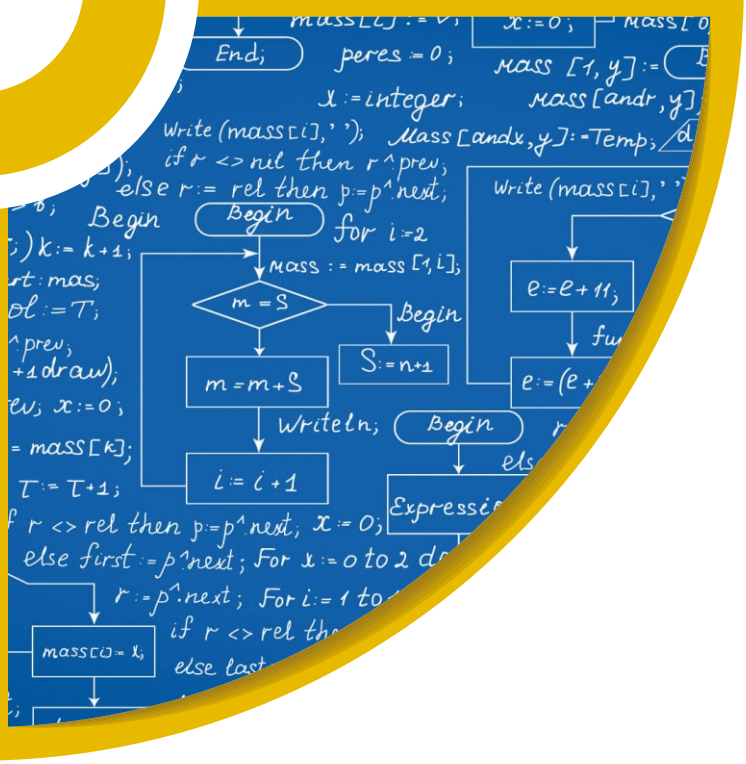
STRUKTUR DATA dan ALGORITMA

(Bagian 1)



Dr. Joseph Teguh Santoso, S.Kom, M.Kom

YAYASAN PRIMA AGUS TEKNIK

STRUKTUR DATA dan ALGORITMA

(Bagian 1)

Dr. Joseph Teguh Santoso, S.Kom, M.Kom

BIODATA PENULIS

Dr. Joseph Teguh Santoso, S.Kom, M.Kom adalah Rektor dari Universitas Sains & Teknologi Komputer (Universitas STEKOM) Semarang yang memiliki banyak pengalaman praktis dalam bidang *e-commerce*sejak Tahun 2002. Beliau mempunyai 3 (tiga) toko *Official Online Store* di China untuk merek Sepeda Raleigh, dengan omzet tahunan pada Tahun 2019 mencapai lebih dari Rp. 35 Milyar rupiah dan terus meningkat. Dr. Joseph T.S memiliki lisensi tunggal sepeda merek “Raleigh” untuk penjualan *Online* di seluruh China. Di samping itu beliau juga memiliki pabrik sepeda dan sepeda listrik merek “Fengjiu”, yaitu Pabrik Sepeda Listrik yang masih tergolong kecil di China. Pengalaman beliau malang melintang di dunia *online store* di China seperti Alibaba, Tmall, Taobao, JD, Aliexpress sangat membantu mahasiswa untuk memiliki pengalaman teknis dan praktis untuk membuka toko *online* bersama beliau.

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-623-5734-16-3 (jil.1 PDF)

STRUKTUR DATA dan ALGORITMA

(Bagian 1)

Dr. Joseph Teguh Santoso, S.Kom, M.Kom

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

STRUKTUR DATA dan ALGORITMA (Bagian 1)

Penulis :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom

ISBN : 9 786235 734163

Editor :
Muhammad Sholikan, M.Kom

Penyunting :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : info@stekom.ac.id

KATA PENGANTAR

Puji syukur pada Tuhan Yang Maha Esa bahwa buku yang berjudul "*Struktur Data dan Algoritma (Bagian 1)*" ini dapat diselesaikan dengan baik. Dalam pemrograman atau aplikasi sebuah komputer, hal penting untuk dipahami ialah bagaimana logika kita dalam mengolah cara pikir untuk mendapatkan solusi, inovasi, dan bahkan untuk menyelesaikan masalah pemrograman yang akan dibangun secara kompleks dan berurutan. Struktur data adalah cara mengatur, menyimpan, maupun mengelola data di media penyimpanan (Warehouse) sehingga data dapat dimanfaatkan secara efisien. Dalam teknik pemrograman, struktur data bisa diartikan tata letak data yang berisi kolom data, baik kolom data yang hanya digunakan untuk tujuan pemrograman maupun kolom data yang ditampilkan oleh aplikasi sehingga dapat digunakan pengguna.

Dalam buku ini dijelaskan bahwa Struktur data meliputi Data Sederhana dan Data Majemuk. Data Majemuk terdiri dari Data Majemuk Linier dan Non Linier. Yang masing-masing struktur data akan dijelaskan mendetail dalam buku ini.

Definisi Algoritma menurut buku ini adalah Runtutan pengambilan putusan secara logis untuk pemecahan masalah. Tiap bab dalam buku ini membahas detail tentang struktur data secara lengkap, karena dilengkapi dengan contoh soal, kasus dan penyelesaian.

Belajar bahasa pemrograman sangat penting, kita dituntut untuk bisa memahami dan mengerti jenis-jenis data yang akan dipergunakan dalam membangun sebuah Program atau Aplikasi, jadi buku ini sangat cocok untuk pembaca yang ingin belajar tentang database.

Algoritma atau dengan kata lain Logic, merupakan komponen dasar dalam pembuatan sebuah Aplikasi Program. Ciri-ciri algoritma yang baik meliputi: *Input, Output, Definite, Efective* dan *Terminate.* Maksudnya adalah sebuah Algoritma harus memiliki Masukan dan Hasil, Juga memiliki kejelasan apa yang dilakukan, serta langkah penyelesaian yang efektif dan langkah tersebut dapat berhenti atau dapat diberhentikan secara jelas. Semoga buku ini dapat memberi manfaat yang besar pada para pembacanya.

Semarang, Desember 2021
Penulis

Dr. Joseph Teguh Santoso, S.Kom, M.Kom

DAFTAR ISI

BAB 1
PENGANTAR

Tujuan dari bab ini adalah untuk menjelaskan pentingnya analisis algoritma, notasinya, hubungan dan pemecahan masalah sebanyak mungkin. Mari kita fokus pada pemahaman elemen dasar algoritma, pentingnya analisis algoritma, dan kemudian perlahan-lahan beralih ke topik lain seperti yang disebutkan di atas. Setelah menyelesaikan bab ini, Anda seharusnya dapat menemukan kompleksitas dari setiap algoritma yang diberikan (terutama fungsi rekursif).

1.1 VARIABEL

Sebelum pergi ke definisi variabel, mari kita menghubungkannya dengan persamaan matematika lama. Kita semua telah memecahkan banyak persamaan matematika sejak kecil. Sebagai contoh, perhatikan persamaan di bawah ini:

$$x^2 + 2y - 2 = 1$$

Kita tidak perlu khawatir tentang penggunaan persamaan ini. Hal penting yang perlu kita pahami adalah bahwa persamaan memiliki nama (x dan y), yang menyimpan nilai (data). Itu berarti nama (x dan y) adalah pengganti untuk mewakili data. Demikian pula, dalam pemrograman ilmu komputer kita membutuhkan sesuatu untuk menyimpan data, dan variabel adalah caranya.

1.2 TIPE DATA

Dalam persamaan di atas, variabel x dan y dapat mengambil nilai apa pun seperti bilangan integral (10, 20), bilangan real (0,23, 5.5), atau hanya 0 dan 1. Untuk menyelesaikan persamaan, kita perlu menghubungkannya untuk jenis nilai yang dapat mereka ambil, dan tipe data adalah nama yang digunakan dalam pemrograman ilmu komputer untuk tujuan ini. Tipe data dalam bahasa pemrograman adalah kumpulan data dengan nilai yang telah ditentukan sebelumnya. Contoh tipe data adalah: integer, floating point, nomor unit, karakter, string, dll.

Memori komputer semua diisi dengan nol dan satu. Jika Kita memiliki masalah dan Kita ingin mengkodekannya, sangat sulit untuk memberikan solusi dalam hal nol dan satu. Untuk membantu pengguna, bahasa pemrograman dan kompiler memberi Kita tipe data. Misalnya, integer membutuhkan 2 byte (nilai sebenarnya tergantung pada compiler), float membutuhkan 4 byte, dll. Ini mengatakan bahwa dalam memori kita menggabungkan 2 byte (16 bit) dan menyebutnya integer. Demikian pula, menggabungkan 4 byte (32 bit) dan menyebutnya float. Tipe data mengurangi upaya pengkodean. Di tingkat atas, ada dua jenis tipe data:

- Tipe data yang ditentukan sistem (juga disebut tipe data Primitif)

- Tipe data yang ditentukan pengguna

Tipe data yang ditentukan sistem (Tipe data primitif)

Tipe data yang didefinisikan oleh sistem disebut tipe data primitif. Tipe data primitif yang disediakan oleh banyak bahasa pemrograman adalah: int, float, char, double, bool, dll. Jumlah bit yang dialokasikan untuk setiap tipe data primitif tergantung pada bahasa pemrograman, kompiler, dan sistem operasi. Untuk tipe data primitif yang sama, bahasa yang berbeda dapat menggunakan ukuran yang berbeda. Tergantung pada ukuran tipe data, nilai total yang tersedia (domain) juga akan berubah.

Misalnya, "int" dapat mengambil 2 byte atau 4 byte. Jika dibutuhkan 2 byte (16 bit), maka total nilai yang mungkin adalah minus 32.768 hingga ditambah 32.767 (-2^{15} hingga 2^{15}-1). Jika dibutuhkan 4 byte (32 bit), maka nilai yang mungkin adalah antara -2.147.483.648 dan +2.147.483.647 (-2^{31} hingga 2^{31}-1). Sama halnya dengan tipe data lainnya.

Tipe data yang ditentukan pengguna

Jika tipe data yang ditentukan sistem tidak cukup, maka sebagian besar bahasa pemrograman memungkinkan pengguna untuk menentukan tipe data mereka sendiri, yang disebut tipe data yang ditentukan pengguna. Contoh yang baik dari tipe data yang ditentukan pengguna adalah: struktur di C/C++ dan kelas di Java. Misalnya, dalam cuplikan di bawah ini, Kita menggabungkan banyak tipe data yang ditentukan sistem dan memanggil tipe data yang ditentukan pengguna dengan nama "tipe baru". Hal ini memberikan lebih banyak fleksibilitas dan kenyamanan dalam menangani memori komputer.

```
struct newType {
    int data1;
    float data 2;
    ...
    char data;
};
```

1.3 STRUKTUR DATA

Berdasarkan pembahasan di atas, setelah kita memiliki data dalam variabel, kita memerlukan beberapa mekanisme untuk memanipulasi data tersebut untuk memecahkan masalah. Struktur data adalah cara tertentu untuk menyimpan dan mengatur data dalam komputer sehingga dapat digunakan secara efisien. Struktur data adalah format khusus untuk mengatur dan menyimpan data. Jenis struktur data umum termasuk array, file, daftar tertaut, tumpukan, antrian, pohon, grafik, dan sebagainya.

Tergantung pada organisasi elemen, struktur data diklasifikasikan menjadi dua jenis:

1) Struktur data linier: Elemen diakses secara berurutan tetapi tidak wajib untuk menyimpan semua elemen secara berurutan. Contoh: Daftar Tertaut, Tumpukan, dan Antrian.
2) Struktur data non-linear: Elemen struktur data ini disimpan/diakses dalam urutan non-linear. Contoh: Pohon dan grafik.

1.4 TIPE DATA ABSTRAK (ADT)

Sebelum mendefinisikan tipe data abstrak, mari kita pertimbangkan pandangan yang berbeda dari tipe data yang ditentukan sistem. Kita semua tahu bahwa, secara default, semua tipe data primitif (int, float, dll.) mendukung operasi dasar seperti penambahan dan pengurangan. Sistem menyediakan implementasi untuk tipe data primitif. Untuk tipe data yang ditentukan pengguna, kita juga perlu mendefinisikan operasi. Implementasi untuk operasi ini dapat dilakukan ketika kita ingin benar-benar menggunakannya. Itu berarti, secara umum, tipe data yang ditentukan pengguna didefinisikan bersama dengan operasinya.

Untuk menyederhanakan proses pemecahan masalah, Kita menggabungkan struktur data dengan operasinya dan Kita menyebutnya Tipe Data Abstrak (ADT). Sebuah ADT terdiri dari dua bagian:

1. Deklarasi data
2. Deklarasi operasi

ADT yang umum digunakan meliputi: *Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs*, dan banyak lainnya. Misalnya, *stack* menggunakan mekanisme LIFO (*Last-In-First-Out*) saat menyimpan data dalam struktur data. Elemen terakhir yang dimasukkan ke dalam tumpukan adalah elemen pertama yang dihapus. Operasi umum itu adalah: membuat tumpukan, mendorong elemen ke tumpukan, mengeluarkan elemen dari tumpukan, menemukan bagian atas tumpukan saat ini, menemukan jumlah elemen dalam tumpukan, dll.

Saat mendefinisikan ADT, jangan khawatir tentang detail implementasi. Mereka datang ke dalam gambar hanya ketika kita ingin menggunakannya. Berbagai jenis ADT cocok untuk berbagai jenis aplikasi, dan beberapa sangat khusus untuk tugas tertentu. Pada akhir buku ini, kita akan membahas banyak dari mereka dan Anda akan berada dalam posisi untuk menghubungkan struktur data dengan jenis masalah yang mereka pecahkan.

1.5 APA ITU ALGORITMA?

Mari kita pertimbangkan masalah menyiapkan telur dadar. Untuk menyiapkan telur dadar, Kita mengikuti langkah-langkah yang diberikan di bawah ini:

1) Siapkan penggorengan.
2) Ambil minyaknya.
    a. Apakah kita punya minyak?
        - Jika ya, masukkan ke dalam panci.
        - Jika tidak, apakah kita ingin membeli minyak?
            1. Jika ya, maka keluarlah dan beli.
            2. Jika tidak, Kita dapat mengakhiri.
3) Nyalakan kompor, dll...

Apa yang Kita lakukan adalah, untuk masalah tertentu (menyiapkan telur dadar), Kita menyediakan prosedur langkah demi langkah untuk menyelesaikannya. Definisi formal dari suatu algoritma dapat dinyatakan sebagai:

Algoritma adalah instruksi langkah-demi-langkah yang jelas untuk memecahkan masalah yang diberikan.

Dalam studi tradisional algoritma, ada dua kriteria utama untuk menilai manfaat dari algoritma: kebenaran (apakah algoritma memberikan solusi untuk masalah dalam jumlah langkah yang terbatas?) dan efisiensi (berapa banyak sumber daya (dalam hal memori dan waktu?)) yang diperlukan untuk mengeksekusi).

Catatan: *Kita tidak perlu membuktikan setiap langkah dari algoritma.*

1.6 MENGAPA ANALISIS ALGORITMA?

Untuk pergi dari kota "A" ke kota "B", ada banyak cara untuk mencapai ini: dengan penerbangan, dengan bus, dengan kereta api dan juga dengan sepeda. Tergantung pada ketersediaan dan kenyamanan, Kita memilih salah satu yang cocok untuk Kita. Demikian pula, dalam ilmu komputer, beberapa algoritma tersedia untuk memecahkan masalah yang sama (misalnya, masalah pengurutan memiliki banyak algoritma, seperti pengurutan penyisipan, pengurutan pemilihan, pengurutan cepat, dan banyak lagi). Analisis algoritma membantu kita untuk menentukan algoritma mana yang paling efisien dalam hal waktu dan ruang yang dikonsumsi.

1.7 TUJUAN ANALISIS ALGORITMA

Tujuan dari analisis algoritma adalah untuk membandingkan algoritma (atau solusi) terutama dalam hal waktu berjalan tetapi juga dalam hal faktor lain (misalnya, memori, upaya pengembang, dll.)

1.8 APA ITU ANALISIS WAKTU BERJALAN?

Ini adalah proses menentukan bagaimana waktu pemrosesan meningkat ketika ukuran masalah (ukuran input) meningkat. Ukuran input adalah jumlah elemen dalam input, dan tergantung pada jenis masalah, input mungkin dari jenis yang berbeda. Berikut ini adalah jenis input yang umum.

- Ukuran array
- Derajat polinomial
- Jumlah elemen dalam matriks
- Jumlah bit dalam representasi biner dari input
- Titik-titik dan sisi-sisi dalam suatu graf.

1.9 BAGAIMANA MEMBANDINGKAN ALGORITMA

Untuk membandingkan algoritma, mari kita definisikan beberapa ukuran objektif:

Waktu eksekusi? Bukan ukuran yang baik karena waktu eksekusi khusus untuk komputer tertentu.

Jumlah pernyataan yang dieksekusi? Bukan ukuran yang baik, karena jumlah pernyataan bervariasi dengan bahasa pemrograman serta gaya masing-masing programmer.

Solusi ideal? Mari kita asumsikan bahwa kita menyatakan waktu berjalan dari suatu algoritma yang diberikan sebagai fungsi dari ukuran input n (yaitu, f(n)) dan membandingkan fungsi-fungsi yang berbeda ini sesuai dengan waktu berjalan. Perbandingan semacam ini tidak tergantung pada waktu mesin, gaya pemrograman, dll.

1.10 APA ITU LAJU PERTUMBUHAN?

Tingkat di mana waktu berjalan meningkat sebagai fungsi input disebut tingkat pertumbuhan. Mari kita asumsikan bahwa Anda pergi ke toko untuk membeli mobil dan sepeda. Jika teman Anda melihat Anda di sana dan bertanya apa yang Anda beli, maka secara umum Anda mengatakan membeli mobil. Ini karena biaya mobil lebih tinggi dibandingkan dengan biaya sepeda (mendekatkan biaya sepeda dengan biaya mobil).

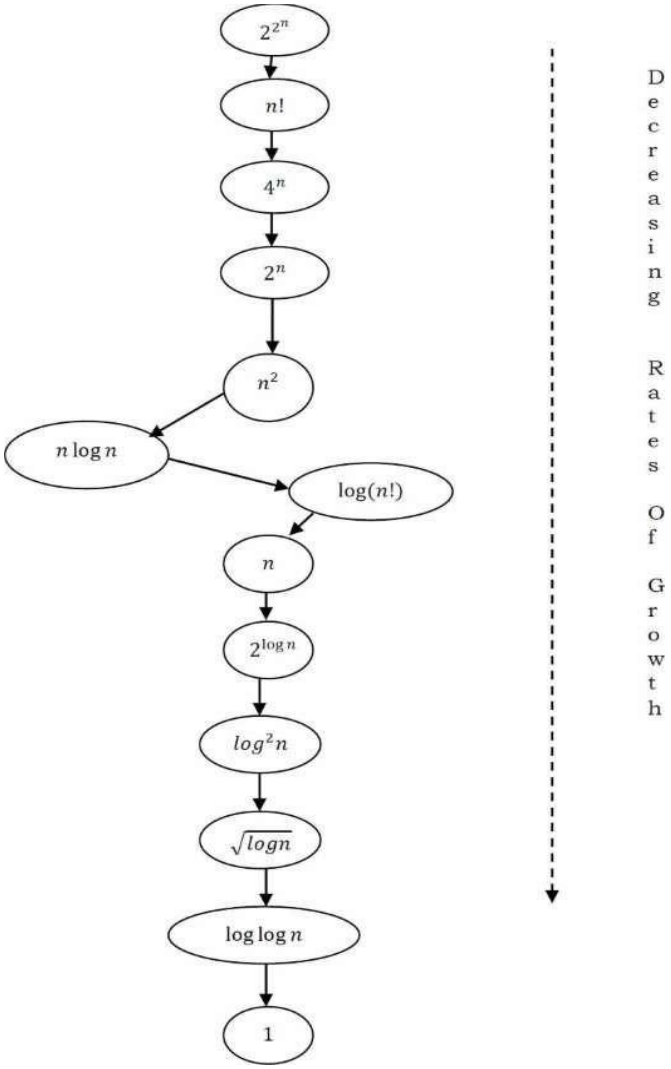
$$Total\ Cost = cost\_of\_car + cost\_of\_bicycle$$
$$Total\ Cost \approx cost\_of\_car\ (approximation)$$

Untuk contoh yang disebutkan di atas, Kita dapat mewakili biaya mobil dan biaya sepeda dalam hal fungsi, dan untuk fungsi yang diberikan mengabaikan suku orde rendah yang relatif tidak signifikan (untuk nilai besar ukuran input, n) . Sebagai contoh, dalam kasus di bawah, n4, 2n2, 100n dan 500 adalah biaya individu dari beberapa fungsi dan mendekati n4 karena n4 adalah tingkat pertumbuhan tertinggi.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

1.11 TINGKAT PERTUMBUHAN YANG UMUM DIGUNAKAN

Diagram di bawah ini menunjukkan hubungan antara tingkat pertumbuhan yang berbeda.



Gambar 1.1 Tingkat Pertumbuhan yang berbeda

Di bawah ini adalah daftar tingkat pertumbuhan yang akan Anda temui dalam bab-bab berikutnya.

Tabel 1.1 Pembahasan lebih Lanjut masalah dibahas dalam buku ini

Kompleksitas Waktu	Nama	Contoh
1	konstanta	Menambahkan elemen ke depan daftar tertaut
logn	Logaritma	Menemukan elemen dalam array yang diurutkan
n	Linier	Menemukan elemen dalam array yang tidak disortir
nlogn	Logaritma Linier	Mengurutkan n item berdasarkan 'bagi dan taklukkan' - Mergesort
n^2	Kuadrat	Jalur terpendek antara dua node dalam grafik
n^3	Kubik	Perkalian Matriks
2^n	Eksponensial	Masalah Menara Hanoi

1.12 JENIS ANALISIS

Untuk menganalisis algoritma yang diberikan, kita perlu mengetahui dengan input mana algoritma membutuhkan waktu lebih sedikit (berkinerja baik1) dan dengan input mana algoritma membutuhkan waktu lama. Kita telah melihat bahwa suatu algoritma dapat direpresentasikan dalam bentuk ekspresi. Itu berarti Kita mewakili algoritma dengan beberapa ekspresi: satu untuk kasus yang membutuhkan waktu lebih sedikit dan yang lainnya untuk kasus yang membutuhkan lebih banyak waktu.

Secara umum, kasus pertama disebut kasus terbaik dan kasus kedua disebut kasus terburuk untuk algoritma. Untuk menganalisis suatu algoritma, kita memerlukan semacam sintaks, dan yang membentuk dasar untuk analisis/notasi asimtotik. Ada tiga jenis analisis:

- Kasus terburuk
  - Mendefinisikan input yang algoritmanya membutuhkan waktu lama (waktu paling lambat untuk diselesaikan).
  - Input adalah yang menjalankan algoritma paling lambat.
- Kasus terbaik
  - Mendefinisikan input yang algoritmanya membutuhkan waktu paling sedikit (waktu tercepat untuk diselesaikan).
  - Input adalah salah satu yang algoritmanya berjalan paling cepat.
- Kasus rata-rata
  - Memberikan prediksi tentang waktu berjalan dari algoritma.
  - Jalankan algoritma berkali-kali, menggunakan banyak input berbeda yang berasal dari beberapa distribusi yang menghasilkan input ini, hitung total waktu berjalan (dengan menambahkan waktu individual), dan bagi dengan jumlah percobaan.
  - Diasumsikan bahwa inputnya acak.

Batas Bawah <= Waktu Rata-rata <= Batas Atas

Untuk algoritma tertentu, kita dapat merepresentasikan kasus terbaik, terburuk, dan rata-rata dalam bentuk ekspresi. Sebagai contoh, misalkan f(n) adalah fungsi yang merepresentasikan algoritma yang diberikan.

$$f(n) = n^2 + 500, \text{ for worst case}$$
$$f(n) = n + 100n + 500, \text{ for best case}$$

Demikian pula untuk kasus rata-rata. Ekspresi mendefinisikan input dengan mana algoritma mengambil waktu berjalan rata-rata (atau memori).

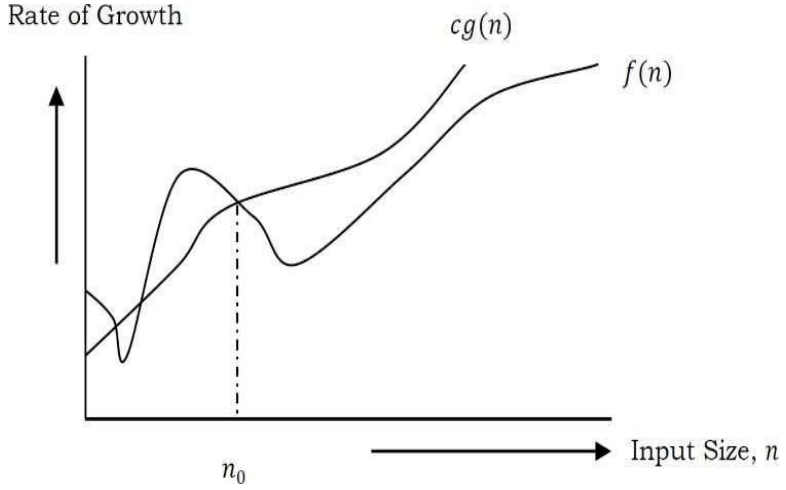
1.13 NOTASI ASIMTOTIK

Memiliki ekspresi untuk kasus terbaik, rata-rata dan terburuk, untuk ketiga kasus kita perlu mengidentifikasi batas atas dan bawah. Untuk merepresentasikan batas atas dan bawah

ini, kita memerlukan semacam sintaks, dan itu adalah subjek dari diskusi berikut. Mari kita asumsikan bahwa algoritma yang diberikan direpresentasikan dalam bentuk fungsi f(n).

1.14 NOTASI BIG-O [FUNGSI BATAS ATAS]

Notasi ini memberikan batas atas yang ketat dari fungsi yang diberikan. Umumnya direpresentasikan sebagai f(n) = O(g(n)). Artinya, pada nilai n yang lebih besar, batas atas f(n) adalah g(n). Misalnya, jika f(n) = n4 + 100n2 + 10n + 50 adalah algoritma yang diberikan, maka n4 adalah g(n). Itu berarti g(n) memberikan tingkat pertumbuhan maksimum untuk f(n) pada nilai n yang lebih besar.



Gambar 1.2 Grafik Tingkat Pertumbuhan untuk f(n)

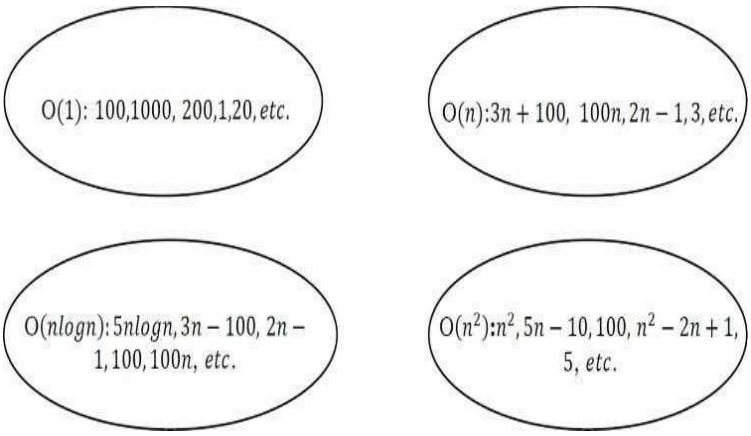
Mari kita lihat notasi O dengan sedikit lebih detail. O–notasi didefinisikan sebagai O(g(n)) = {f(n): terdapat konstanta positif c dan n0 sedemikian rupa sehingga 0 f(n) cg(n) untuk semua n > n0}. g(n) adalah batas atas ketat asimtotik untuk f(n). Tujuan Kita adalah memberikan laju pertumbuhan terkecil g(n) yang lebih besar dari atau sama dengan laju pertumbuhan algoritma yang diberikan /(n).

Umumnya kita membuang nilai n yang lebih rendah. Itu berarti laju pertumbuhan pada nilai n yang lebih rendah tidak penting. Pada gambar, n0 adalah titik dari mana kita perlu mempertimbangkan tingkat pertumbuhan untuk algoritma yang diberikan. Di bawah n0, tingkat pertumbuhan bisa berbeda. n0 disebut ambang batas untuk
fungsi yang diberikan.

Visualisasi Big-O

O(g(n)) adalah himpunan fungsi dengan orde pertumbuhan yang lebih kecil atau sama dengan g(n). Sebagai contoh; O(n2) termasuk O(1), O(n), O(nlogn), dll.

Catatan: *Analisis algoritma hanya pada nilai n yang lebih besar. Artinya, di bawah n0 kita tidak peduli dengan laju pertumbuhan*.



Gambar 1.3 Notasi Big-O

Contoh Big-O

Contoh-1 Temukan batas atas untuk f(n) = 3n + 8

Solusi: $3n + 8 \leq 4n$, untuk semua $n \geq 8$

$\therefore 3n + 8 = O(n)$ dengan c = 4 dan n0 = 8

Contoh-2 Temukan batas atas untuk f(n) = $n^2 + 1$

Solusi: $n^2 + 1 \leq 2n^2$, untuk semua $n \geq 1$

$\therefore n^2 + 1 = O(n^2)$ dengan c = 2 dan $n_0 = 1$

Contoh-3 Temukan batas atas untuk f(n) = $n^4 + 100n^2 + 50$

Solusi: $n^4 + 100n^2 + 50 \leq 2n^4$, untuk semua $n \geq 11$

$\therefore n^4 + 100n^2 + 50 = O(n^4)$ dengan c = 2 dan $n_0 = 11$

Contoh-4 Temukan batas atas untuk f(n) = $2n^3 - 2n^2$

Solusi: $2n^3 - 2n^2 \leq 2n^3$, untuk semua $n > 1$

$\therefore 2n^3 - 2n^2 = O(n^3)$ dengan c = 2 dan $n_0 = 1$

Contoh-5 Temukan batas atas untuk f(n) = n

Solusi: $n \leq n$, untuk semua $n \geq 1$

$\therefore n = O(n)$ dengan c = 1 dan $n_0 = 1$

Contoh-6 Temukan batas atas untuk f(n) = 410

Solusi: $410 \leq 410$, untuk semua $n > 1$

$\therefore 410 = O(1)$ dengan c = 1 dan $n_0 = 1$

Tidak Ada Keunikan?

Tidak ada himpunan nilai unik untuk n0 dan c dalam membuktikan batas asimtotik. Mari kita pertimbangkan, 100n + 5 = O(n). Untuk fungsi ini ada beberapa nilai n_0 dan c yang mungkin.

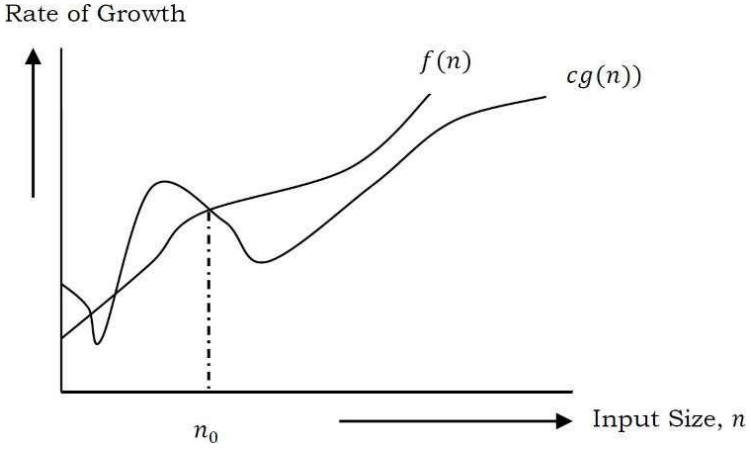
Solusi1: $100n + 5 \leq 100n + n = 101n \leq 101n$, untuk semua $n \geq 5$, $n_0 = 5$ dan c = 101 adalah solusi.

Solusi2: $100n + 5 \leq 100n + 5n = 105n \leq 105n$, untuk semua $n > 1$, $n_0 = 1$ dan c = 105 juga merupakan solusi.

1.15 NOTASI OMEGA-Q [FUNGSI BATAS BAWAH]

Mirip dengan diskusi O, notasi ini memberikan batas bawah yang lebih ketat dari algoritma yang diberikan dan Kita menyatakannya sebagai $f(n) = \Omega(g(n))$. Artinya, pada nilai n yang lebih besar, batas bawah yang lebih rapat dari f(n) adalah g(n). Misalnya, jika $f(n) = 100n^2 + 10n + 50$, g(n) adalah $\Omega(n^2)$.

Notasi dapat didefinisikan sebagai $\Omega(g(n)) = \{f(n)$: terdapat konstanta positif c dan n_0 sedemikian rupa sehingga $0 \leq cg(n) \leq f(n)$ untuk semua $n \geq n0\}$. g(n) adalah batas bawah ketat asimtotik untuk f(n). Tujuan Kita adalah untuk memberikan laju pertumbuhan terbesar g(n) yang kurang dari atau sama dengan laju pertumbuhan algoritma yang diberikan f(n).



Gambar 1.4 Grafik Notasi Omega-Q

Contoh Ω

Contoh-1 Temukan batas bawah untuk $f(n) = 5n^2$.

Solusi: ∃c, n_0 Sehingga: $0\ cn^2 \leq 5n^2\ cn^2\ 5n^2$ c = 5 dan $n_0 = 1$

$\therefore 5n^2 = \Omega(n^2)$ dengan c = 5 dan $n_0 = 1$

Contoh-2 Buktikan $f(n) = 100n + 5 \neq \Omega(n^2)$.

Solusi: ∃c, n_0 Sehingga: $0\ cn^2\ 100n + 5$

$100n + 5 \leq 100n + 5n(\forall n \geq 1) = 105n$

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$

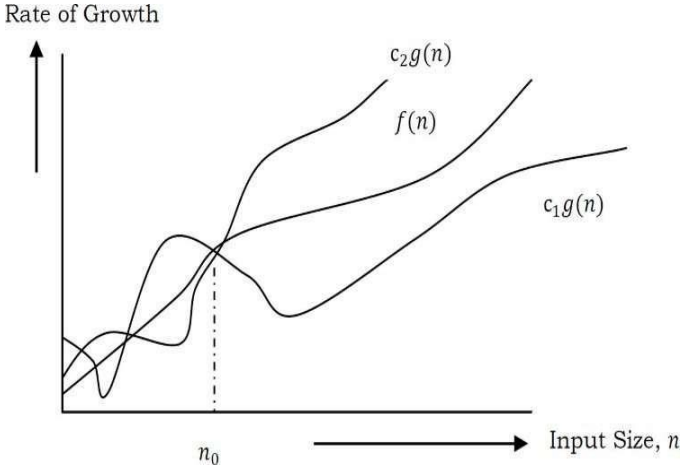
Karena n positif $cn - 105 \leq 0 \Rightarrow n \leq 105/c$

$\Rightarrow$ Kontradiksi: n tidak boleh lebih kecil dari konstanta

Contoh-3 $2n = Q(n)$, $n^3 = Q(n^3)$, $= O(\log n)$.

1.16 NOTASI THETA-Θ [FUNGSI URUTAN]

Notasi ini memutuskan apakah batas atas dan batas bawah dari suatu fungsi (algoritma) yang diberikan adalah sama. Rata-rata waktu berjalan dari suatu algoritma selalu antara batas bawah dan batas atas. Jika batas atas (O) dan batas bawah (Ω) memberikan hasil yang sama, maka notasi juga akan memiliki laju pertumbuhan yang sama.



Gambar 1.5 Grafik Notasi Theta

Sebagai contoh, mari kita asumsikan bahwa f(n) = 10n + n adalah ekspresi. Kemudian, batas atasnya yang ketat g(n) adalah O(n). Laju pertumbuhan dalam kasus terbaik adalah g(n) = O(n). Dalam hal ini, tingkat pertumbuhan dalam kasus terbaik dan kasus terburuk adalah sama. Akibatnya, kasus rata-rata juga akan sama. Untuk suatu fungsi (algoritma), jika laju pertumbuhan (batas) untuk O dan tidak sama, maka laju pertumbuhan untuk kasus mungkin tidak sama. Dalam hal ini, kita perlu mempertimbangkan semua kemungkinan kompleksitas waktu dan mengambil rata-ratanya (misalnya, untuk kasus rata-rata sortir cepat, lihat bab Sorting).

Sekarang perhatikan definisi notasi Θ .Didefinisikan sebagai Θ (g(n)) = {f(n): terdapat konstanta positif c_1,c_2 dan n_0 sedemikian rupa sehingga $0 \leq c_1g(n) \leq f(n)$ $c_2g(n)$ untuk semua $n \geq n_0$ }. g(n) adalah ikatan ketat asimtotik untuk f(n). Θ(g(n)) adalah himpunan fungsi dengan orde pertumbuhan yang sama dengan g(n).

Contoh Θ

Contoh 1 Temukan Θ terikat untuk $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solusi: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$ untuk semua, $n \geq 2$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ dengan $c_1 = 1/5$, $c_2 = 1$ dan $n_0 = 2$

Contoh 2 Buktikan $n \neq \Theta(n^2)$

Solusi: $c_1 n^2 \leq n \leq c_2 n^2$ hanya berlaku untuk: $n \leq 1/c_1$

$\therefore n \neq \Theta(n^2)$

Contoh 3 Buktikan $6n^3 \neq \Theta(n^2)$

Solusi: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ hanya berlaku untuk: $n \leq c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Contoh 4 Buktikan $n \neq \Theta(\log n)$

Solusi: $c_1 \log n \leq n \leq c_2 \log n \Rightarrow c_2 \geq \frac{n}{\log n}$, $\forall n \geq n_0$ – Tidak mungkin

1.17 CATATAN PENTING

Untuk analisis (kasus terbaik, kasus terburuk dan rata-rata), Kita mencoba memberikan batas atas (O) dan batas bawah (Ω) dan waktu berjalan rata-rata (Θ). Dari contoh di atas, juga harus jelas bahwa, untuk suatu fungsi (algoritma), mendapatkan batas atas (O) dan batas bawah (Ω) dan waktu berjalan rata-rata (Θ) tidak selalu mungkin. Misalnya, jika kita membahas kasus terbaik dari suatu algoritma, kita mencoba memberikan batas atas (O) dan batas bawah (Ω) dan waktu berjalan rata-rata (Θ).

Dalam bab-bab selanjutnya, kita biasanya fokus pada batas atas (O) karena mengetahui batas bawah (Ω) dari suatu algoritma tidak penting secara praktis, dan kita menggunakan notasi jika batas atas (O) dan batas bawah (Ω) adalah sama.

1.18 MENGAPA DISEBUT ANALISIS ASIMTOTIK?

Dari pembahasan di atas (untuk ketiga notasi: kasus terburuk, kasus terbaik, dan kasus rata-rata), kita dapat dengan mudah memahami bahwa, dalam setiap kasus untuk fungsi tertentu f(n) kita mencoba mencari fungsi lain g(n) yang mendekati f(n) pada nilai n yang lebih tinggi. Itu berarti g(n) juga merupakan kurva yang mendekati f(n) pada nilai n yang lebih tinggi.

Dalam matematika kita menyebut kurva seperti itu sebagai kurva asimtotik. Dalam istilah lain, g(n) adalah kurva asimtotik untuk f(n). Untuk alasan ini, Kita menyebut analisis algoritma analisis asimtotik.

1.19 PEDOMAN ANALISIS ASIMTOTIK

Ada beberapa aturan umum untuk membantu kita menentukan waktu berjalan dari suatu algoritma.

1) **Loop:** Waktu berjalan dari sebuah loop, paling banyak, adalah waktu berjalan dari pernyataan di dalam loop (termasuk tes) dikalikan dengan jumlah iterasi.

```
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; // constant time, c
```

Total waktu = konstanta c × n = c n = O(n).

2) **Loop bersarang:** Analisis dari dalam ke luar. Total waktu berjalan adalah produk dari ukuran semua loop.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executes n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total waktu = c × n × n = cn^2 = $O(n^2)$.

3) **Pernyataan berurutan:** Tambahkan kompleksitas waktu dari setiap pernyataan.

```
x = x +1; //constant time
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executes n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total waktu = $c_0 + c_1n + c_2n^2$ = $O(n^2)$.

4) **Pernyataan if-then-else:** Waktu berjalan kasus terburuk: tes, ditambah bagian then atau bagian else (mana yang lebih besar).

```
//test: constant
if(length( ) == 0 ) {
    return false; //then part: constant
}
else {// else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++) {
      // another if : constant + constant (no else part)
      if(!list[n].equals(otherList.list[n]))
          //constant
        return false;
    }
}
```

Total waktu = c0 + c1 + (c2 + c3) * n = O(n).

5) **Kompleksitas logaritma:** Suatu algoritma adalah O(logn) jika dibutuhkan waktu yang konstan untuk memotong ukuran masalah dengan pecahan (biasanya dengan). Sebagai contoh mari kita perhatikan program berikut:

```
for (i=1; i<=n;)
   i = i*2;
```

Jika kita amati dengan seksama, nilai i berlipat ganda setiap saat. Awalnya i = 1, pada langkah berikutnya I = 2, dan pada langkah selanjutnya i = 4,8 dan seterusnya. Mari kita asumsikan bahwa loop dieksekusi beberapa k kali. Pada langkah ke-k^{th} $2^k = n$, dan pada langkah $(k + 1)^{th}$ kita keluar dari loop. Mengambil logaritma di kedua sisi, memberikan

$log(2^k) = logn$
$klog2 = logn$
$k = logn$ //if we assume base-2

Total waktu = O(logn).

Catatan: Demikian pula untuk kasus di bawah ini, laju pertumbuhan kasus terburuk adalah O(logn). Diskusi yang sama berlaku untuk urutan menurun juga.

```
for (i=n; i>=1;)
   i = i/2;
```

Contoh lain: pencarian biner (menemukan kata dalam kamus n halaman)

- Lihat titik tengah di kamus
- Apakah kata mengarah ke kiri atau kanan tengah?
- Ulangi proses dengan bagian kiri atau kanan kamus sampai kata ditemukan.

1.20 MENYEDERHANAKAN SIFAT NOTASI ASIMTOTIK

- Transitivitas: f(n) = Θ(g(n)) dan g(n) = Θ(h(n)) ⇒ f(n) = Θ (h(n)). Berlaku untuk O dan juga.
- Refleksivitas: f(n) = Θ (f(n)). Berlaku untuk O dan Ω.
- Simetri: f(n) = Θ (g(n)) jika dan hanya jika g(n) = Θ (f(n)).
- Transpose simetri: f(n) = O(g(n)) jika dan hanya jika g(n) = Ω (f(n)).
- Jika f(n) ada dalam O(kg(n)) untuk sembarang konstanta k > 0, maka f(n) ada di O(g(n)).
- Jika $f_1(n)$ ada di $O(g_1(n))$ dan $f_2(n)$ ada di $O(g_2(n))$, maka $(f_1 + f_2)(n)$ ada di $O(max(g_1(n))$, (g1(n))).
- Jika $f_1(n)$ ada di $O(g_1(n))$ dan $f_2(n)$ ada di $O(g_2(n))$ maka $f_1(n)$ $f_2(n)$ ada di $O(g_1(n)$ $g_1(n))$.

1.21 LOGARITMA DAN PENJUMLAHAN YANG UMUM DIGUNAKAN

- **Logaritma**

$log\ x^y = y\ log\ x$ $\quad\quad$ $logn = log_{10}^n$

$log\ xy = logx + logy$ $\quad\quad$ $log^k n = (logn)^k$

$log\ logn = log(logn)$ $\quad\quad$ $log\frac{x}{y} = logx - logy$

$a^{log_b^x} = x^{log_b^a}$ $\quad\quad$ $log_b^x = \frac{log_a^x}{log_a^b}$

- **Deret aritmatika**

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

- **Deret geometris**

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

- **Seri harmonic**

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx log\ n$$

- **Formula penting lainnya**

$$\sum_{k=1}^{n} log\ k \approx nlogn$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

1.22 TEOREMA UTAMA UNTUK PERULANGAN DIVIDE DAN CONQUER

Semua algoritma bagi dan taklukkan (juga dibahas secara rinci dalam bab Membagi dan Menaklukkan) membagi masalah menjadi sub-masalah, yang masing-masing merupakan bagian dari masalah asli, dan kemudian melakukan beberapa pekerjaan tambahan untuk menghitung jawaban akhir. Sebagai contoh, algoritma pengurutan gabungan [untuk detailnya, lihat bab Pengurutan] beroperasi pada dua sub-masalah, yang masing-masing berukuran setengah dari yang asli, dan kemudian melakukan O(n) pekerjaan tambahan untuk penggabungan. Ini memberikan persamaan waktu berjalan:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Teorema berikut dapat digunakan untuk menentukan waktu berjalan dari algoritma Divide and Conquer. Untuk program (algoritma) yang diberikan, pertama-tama kita mencoba mencari relasi rekurensi untuk masalah tersebut. Jika perulangan seperti di bawah ini maka kita bisa langsung memberikan jawabannya tanpa menyelesaikannya sepenuhnya. Jika perulangan berbentuk

$T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, di mana a 1,b > 1,k 0 dan p adalah bilangan real, maka:

1) Jika a > b^k, maka $T(n) = \Theta\left(n^{log_b^a}\right)$
2) Jika a = b^k
   a. Jika p > -1, maka $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. Jika p = -1, maka $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. Jika p < -1, maka $T(n) = \Theta\left(n^{log_b^a}\right)$
3) Jika a < b^k
   a. Jika p $\geq$0, maka T(n) = Θ ($n^k log^p n$)
   b. Jika p < 0, maka T(n) = O(n^k)

1.23 TEOREMA DIVIDE DAN CONQUER: MASALAH & SOLUSI

Untuk setiap perulangan berikut, berikan ekspresi untuk runtime T(n) jika perulangan dapat diselesaikan dengan Teorema Master. Jika tidak, tunjukkan bahwa Teorema Utama tidak berlaku.

Soal-1 $T(n) = 3T(n/2) + n^2$

Solusi: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Teorema Utama Kasus 3.a)

Soal-2 $T(n) = 4T(n/2) + n^2$

Solusi: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 logn)$ (Teorema Utama Kasus 2.a)

Soal-3 $T(n) = T(n/2) + n^2$

Solusi: $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Teorema Utama Kasus 3.a)

Soal-4 $T(n) = 2^n T(n/2) + n^n$

Solusi: $T(n) = 2^n T(n/2) + n^n$ => Tidak berlaku (a tidak konstan)

Soal-5 $T(n) = 16T(n/4) + n$

Solusi: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Teorema Utama Kasus 1)

Soal-6 $T(n) = 2T(n/2) + nlogn$

Solusi: $T(n) = 2T(n/2) + nlogn \Rightarrow T(n) = \Theta(nlog^2 n)$ (Teorema Utama Kasus 2.a)

Soal-7 T(n) = 2T(n/2) + n/logn
Solusi: T(n) = 2T(n/2)+ n/logn =>T(n) = Θ (nloglogn) (Teorema Utama Kasus 2. b)

Soal-8 T(n) = 2T (n/4) + n^{051}
Solusi: T(n) = 2T(n/4) + n^{051} => T (n) = ($n^{0.51}$) (Teorema Utama Kasus 3.b)

Soal-9 T(n) = 0,5T(n/2) + 1/n
Solusi: T(n) = 0.5T(n/2) + 1/n => Tidak berlaku (a < 1)

Soal-10 T (n) = 6T(n/3)+ n^2 logn
Solusi: T(n) = 6T(n/3) + n^2logn => T(n) = Θ (n^2logn) (Teorema Utama Kasus 3.a)

Soal-11 T(n) = 64T(n/8) – n^2logn
Solusi: T(n) = 64T(n/8) – n^2logn => Tidak berlaku (fungsi tidak positif)

Soal-12 T(n) = 7T(n/3) + n^2
Solusi: T(n) = 7T(n/3) + n^2 => T(n) = Θ (n^2) (Teorema Utama Kasus 3.as)

Soal-13 T(n) = 4T(n/2) + logn
Solusi: T(n) = 4T(n/2) + logn => T(n) = Θ (n^2) (Teorema Utama Kasus 1)

Soal-14 T(n) = 16T (n/4) + n!
Solusi: T(n) = 16T (n/4) + n! => T(n) = (n!) (Teorema Utama Kasus 3.a)

Soal-15 T(n) = $\sqrt{2}$T(n/2) + logn
Solusi: T(n) = $\sqrt{2}$T(n/2) + logn => T(n) = Θ ($\sqrt{n}$) (Teorema Utama Kasus 1)

Soal-16 T(n) = 3T(n/2) + n
Solusi: T(n) = 3T(n/2) + n =>T(n) = Θ (n^{log3}) (Teorema Utama Kasus 1)

Soal-17 T(n) = 3T(n/3) + $\sqrt{n}$
Solusi: T(n) = 3T(n/3) + $\sqrt{n}$ => T(n) = Θ (n) (Teorema Utama Kasus 1)

Soal-18 T(n) = 4T(n/2) + cn
Solusi: T(n) = 4T(n/2) + cn => T(n) = Θ (n^2) (Teorema Utama Kasus 1)

Soal-19 T(n) = 3T(n/4) + nlogn
Solusi: T(n) = 3T(n/4) + nlogn => T(n) = Θ (n^{logn}) (Teorema Utama Kasus 3.a)

Soal-20 T (n) = 3T(n/3) + n/2

Solusi: T(n) = 3T(n/3)+ n/2 => T (n) = Θ (n^{logn}) (Teorema Utama Kasus 2.a)

1.24 TEOREMA UTAMA UNTUK PENGURANGAN DAN PENAKLUKAN PERULANGAN

Biarkan T(n) menjadi fungsi yang didefinisikan pada n positif, dan memiliki properti

$$T(n) = \begin{cases} c, & \text{if } n \le 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

untuk beberapa konstanta c,a > 0,b 0,k 0, dan fungsi f(n). Jika f(n) dalam O(nk), maka

$$T(n) = \begin{cases} O(n^k), & \text{if a} < 1 \\ O(n^{k+1}), & \text{if a} = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if a} > 1 \end{cases}$$

1.25 VARIAN DARI PENGURANGAN DAN PENAKLUKAN MASTER TEOREMA

Solusi persamaan T(n) = T(α n) + T((1 –)n) + n, di mana 0 < α < 1 dan > 0 adalah konstanta, adalah O(nlogn).

1.26 METODE MENEBAK DAN PEMBUKTIAN

Sekarang, mari kita bahas metode yang dapat digunakan untuk menyelesaikan setiap pengulangan. Ide dasar di balik metode ini adalah:

Tebak jawabannya; kemudian buktikan kebenarannya dengan induksi.

Dengan kata lain, ini menjawab pertanyaan: Bagaimana jika pengulangan yang diberikan tampaknya tidak cocok dengan salah satu metode (teorema induk) ini? Jika kita menebak solusi dan kemudian mencoba memverifikasi tebakan kita secara induktif, biasanya buktinya akan berhasil (dalam hal ini kita selesai), atau buktinya akan gagal (dalam hal ini kegagalan akan membantu kita memperbaiki tebakan kita).

Sebagai contoh, perhatikan pengulangan $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$. Ini tidak sesuai dengan bentuk yang disyaratkan oleh Teorema Master. Mengamati pengulangan dengan cermat memberi kita kesan bahwa itu mirip dengan metode membagi dan menaklukkan (membagi masalah menjadi $\sqrt{n}$ submasalah masing-masing dengan ukuran $\sqrt{n}$). Seperti yang dapat kita lihat, ukuran submasalah pada rekursi tingkat pertama adalah n. Jadi, mari kita tebak bahwa T(n) = O(nlogn), dan kemudian coba buktikan bahwa tebakan kita benar. Mari kita mulai dengan mencoba membuktikan batas atas T(n) < cnlogn:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.\, c\sqrt{n}\, log\sqrt{n} + n \\ &= n.\, c\, log\sqrt{n} + n \\ &= n.c.\frac{1}{2}.logn + n \\ &\leq cnlogn \end{aligned}$$

Pertidaksamaan terakhir hanya mengasumsikan bahwa 1 ≤ c. $\frac{1}{2}$ logn. Ini benar jika n cukup besar dan untuk sembarang konstanta c, tidak peduli seberapa kecilnya. Dari bukti di atas, kita dapat melihat bahwa tebakan kita benar untuk batas atas. Sekarang, mari kita buktikan batas bawah untuk perulangan ini.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.\, k\, \sqrt{n}\, log\sqrt{n} + n \\ &= n.\, k\, log\sqrt{n} + n \\ &= n.k.\frac{1}{2}.logn + n \\ &\geq knlogn \end{aligned}$$

Pertidaksamaan terakhir hanya mengasumsikan bahwa 1≥ k. $\frac{1}{2}$ logn. Ini salah jika n cukup besar dan untuk setiap konstanta k. Dari bukti di atas, kita dapat melihat bahwa tebakan kita salah untuk batas bawah.

Dari pembahasan di atas, Kita memahami bahwa Θ (nlogn) terlalu besar. Bagaimana dengan (n)? Batas bawah mudah dibuktikan secara langsung:

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n \quad \geq n$$

Sekarang, mari kita buktikan batas atas untuk Θ (n) ini.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n} + n \\ &= n.\, c + n \\ &= n\,(c + 1) \\ &\not\leq cn \end{aligned}$$

Dari induksi di atas, kita memahami bahwa Θ (n) terlalu kecil dan Θ (nlogn) terlalu besar. Jadi, kita membutuhkan sesuatu yang lebih besar dari n dan lebih kecil dari nlogn. Bagaimana tentang $n\sqrt{logn}$?

Membuktikan batas atas untuk $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\, c.\frac{1}{\sqrt{2}}\, log\sqrt{n} + n \\ &\leq cnlog\sqrt{n} \end{aligned}$$

Buktikan batas bawah untuk $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ k.\frac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\not\geq knlog\sqrt{n} \end{aligned}$$

Langkah terakhir tidak berhasil. Jadi, Θ ($n\sqrt{logn}$) tidak bekerja. Apa lagi antara n dan nlogn? Bagaimana dengan nloglogn? Membuktikan batas atas untuk nloglogn:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ c.loglogn\text{-}c.n + n \\ &\leq cnloglogn, \text{ if } c \geq 1 \end{aligned}$$

Membuktikan batas bawah untuk nloglogn:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ k.loglogn\text{-}k.n + n \\ &\geq knloglogn, \text{ if } k \leq 1 \end{aligned}$$

Dari pembuktian di atas, kita dapat melihat bahwa T(n) cnloglogn, jika c 1 dan T(n) knloglogn, jika k 1. Secara teknis, kita masih kehilangan kasus dasar di kedua pembuktian, tetapi kita dapat cukup yakin pada titik ini bahwa T(n) = (nloglogn).

1.27 ANALISIS DIAMORTISASI

Analisis diamortisasi mengacu pada penentuan waktu rata-rata waktu berjalan untuk urutan operasi. Ini berbeda dari analisis kasus rata-rata, karena analisis yang diamortisasi tidak membuat asumsi tentang distribusi nilai data, sedangkan analisis kasus rata-rata mengasumsikan data tidak "buruk" (misalnya, beberapa algoritma pengurutan rata-rata bekerja dengan baik di semua pengurutan input. tetapi sangat buruk pada pemesanan input tertentu). Artinya, analisis diamortisasi adalah analisis kasus terburuk, tetapi untuk urutan operasi daripada untuk operasi individu.

Motivasi untuk analisis yang diamortisasi adalah untuk lebih memahami waktu berjalan dari teknik tertentu, di mana analisis kasus terburuk standar memberikan batasan yang terlalu pesimis. Analisis diamortisasi umumnya berlaku untuk metode yang terdiri dari urutan operasi, di mana sebagian besar operasi murah, tetapi beberapa operasi mahal. Jika kita dapat menunjukkan bahwa operasi mahal sangat jarang, kita dapat mengubahnya menjadi operasi murah, dan hanya mengikat operasi murah.

Pendekatan umumnya adalah membebankan biaya buatan untuk setiap operasi dalam urutan, sehingga total biaya buatan untuk urutan operasi membatasi total biaya nyata untuk urutan tersebut. Biaya artifisial ini disebut biaya operasi yang diamortisasi. Untuk menganalisis waktu berjalan, biaya diamortisasi dengan demikian adalah cara yang benar untuk memahami waktu berjalan secara keseluruhan – tetapi perhatikan bahwa operasi tertentu masih dapat memakan waktu lebih lama sehingga ini bukan cara untuk membatasi waktu berjalan dari setiap operasi individu dalam urutan. Ketika satu peristiwa dalam urutan mempengaruhi biaya peristiwa selanjutnya:

- Satu tugas tertentu mungkin mahal.
- Tapi mungkin meninggalkan struktur data dalam keadaan bahwa beberapa operasi berikutnya menjadi lebih mudah.

Contoh: Mari kita pertimbangkan sebuah array elemen dari mana kita ingin menemukan elemen terkecil ke-k. Kita dapat memecahkan masalah ini menggunakan pengurutan. Setelah mengurutkan array yang diberikan, kita hanya perlu mengembalikan elemen ke-k darinya. Biaya melakukan pengurutan (dengan asumsi algoritma pengurutan berbasis perbandingan) adalah O(nlogn). Jika kita melakukan n pilihan seperti itu, maka biaya rata-rata dari setiap pilihan adalah O(nlogn/n) = O(logn). Ini dengan jelas menunjukkan bahwa penyortiran sekali mengurangi kompleksitas operasi selanjutnya.

1.28 ANALISIS ALGORITMA: MASALAH & SOLUSI

Catatan: Dari soal-soal berikut, coba pahami kasus-kasus yang memiliki kompleksitas berbeda (O(n), O(logn), O(loglogn) dll.).

Soal-21 Temukan kompleksitas perulangan di bawah ini:

$$T(n) = \begin{cases} 3T(n-1), if\ n > 0, \\ 1, \qquad otherwise \end{cases}$$

Solusi: Mari kita coba menyelesaikan fungsi ini dengan substitusi.

$T(n) = 3T(n-1)$

$T(n) = 3(3T(n-2)) = 3^2T(n-2)$

$T(n) = 3^2(3T(n-3))$

.

.

$T(n) = 3^nT(n-n) = 3^nT(0) = 3^n$

Ini jelas menunjukkan bahwa kompleksitas fungsi ini adalah O(3n).

Catatan: Kita dapat menggunakan teorema master Pengurangan dan Penaklukan untuk masalah ini.

Soal-22 Temukan kompleksitas perulangan di bawah ini:

$$T(n) = \begin{cases} 2T(n-1) - 1, if\ n > 0, \\ 1, \qquad otherwise \end{cases}$$

Solusi: Mari kita coba menyelesaikan fungsi ini dengan substitusi.

$T(n) = 2T(n-1) - 1$

$T(n) = 2(2T(n-2) - 1) - 1 = 2^2T(n-2) - 2 - 1$

$T(n) = 2^2(2T(n-3) - 2 - 1) - 1 = 2^3T(n-4) - 2^2 - 2^1 - 2^0$

$T(n) = 2^nT(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \quad 2^2 - 2^1 - 2^0$

$T(n) = 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \quad 2^2 - 2^1 - 2^0$

$T(n) = 2^n - (2^n - 1)$ [*note:* $2^{n-1} + 2^{n-2} + \ldots + 2^0 = 2^n$]

$T(n) = 1$

Kompleksitas Waktu adalah O(1). Perhatikan bahwa sementara relasi perulangan terlihat eksponensial, solusi untuk relasi perulangan di sini memberikan hasil yang berbeda.

Soal-23 Berapa waktu berjalan dari fungsi berikut?

```
void Function(int n) {
        int i=1, s=1;
        while( s <= n) {
                i++;
                s= s+i;
                printf("*");
        }
}
```

Solusi Pertimbangkan komentar di fungsi di bawah ini:

```
void Function (int n) {
   int i=1, s=1;
   // s is increasing not at rate 1 but i
   while( s <= n) {
      i++;
      s= s+i;
      printf("*");
   }
}
```

Kita dapat mendefinisikan istilah 's' sesuai dengan relasi $s_i = s_{i-1} + i$. Nilai 'sering' meningkat 1 untuk setiap iterasi. Nilai yang terdapat dalam 's' pada iterasi ke-i adalah jumlah dari '('bilangan bulat positif pertama. Jika k adalah jumlah total iterasi yang diambil oleh program, maka loop while berakhir jika:

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

Soal-24 Temukan kompleksitas fungsi yang diberikan di bawah ini.

```
void function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;

 }
```

Solusi:

```
void function(int n) {
      int i, count =0;
      for(i=1; i*i<=n; i++)
              count++;
}
```

Dalam fungsi yang disebutkan di atas, loop akan berakhir, jika $i^2 > n \Rightarrow T(n) = O(\sqrt{n})$. Ini mirip dengan Soal-23.

Soal-25 Berapa kompleksitas program yang diberikan di bawah ini:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j= j+1)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solusi: Perhatikan komentar pada fungsi berikut.

```
void function(int n) {
      int i, j, k , count =0;
        //outer loop execute n/2 times
      for(i=n/2; i<=n; i++)
              //middle loop executes n/2 times
              for(j=1; j + n/2<=n; j= j+1)
                      //inner loop execute logn times
                      for(k=1; k<=n; k= k * 2)
                              count++;
}
```

Kompleksitas fungsi di atas adalah $O(n^2 \log n)$.

Soal-26 Berapakah kompleksitas program yang diberikan di bawah ini:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

```
void function(int n) {
    int i, j, k , count =0;
    //outer loop execute n/2 times
    for(i=n/2; i<=n; i++)
        //middle loop executes logn times
        for(j=1; j<=n; j= 2 * j)
            //inner loop execute logn times
            for(k=1; k<=n; k= k*2)
                count++;
}
```

Solusi: Perhatikan komentar pada fungsi berikut.

Kompleksitas fungsi di atas adalah $O(n\log^2 n)$.

Soal-27 Temukan kerumitan program di bawah ini.

```
function( int n ) {
    if(n == 1) return;
    for(int i = 1 ; i <= n ; i + + ) {
        for(int j= 1 ; j <= n ; j + + ) {
            printf("*" );
            break;
        }
    }
}
```

Solusi: Perhatikan komentar pada fungsi di bawah ini.

```
function( int n ) {
   //constant time
   if( n == 1 ) return;
   //outer loop execute n times
   for(int i = 1 ; i <= n ; i + + ) {
      // inner loop executes only time due to break statement.
      for(int j= 1 ; j <= n ; j + + ) {
         printf("*" );
         break;
      }
   }
}
```

Kompleksitas fungsi di atas adalah O(n). Meskipun loop dalam dibatasi oleh n, karena pernyataan break, loop hanya dieksekusi sekali.

Soal-28 Tulislah sebuah fungsi rekursif untuk waktu berjalan T(n) dari fungsi yang diberikan di bawah ini. Buktikan dengan metode iteratif bahwa $T(n) = \Theta(n^3)$.

```
function( int n ) {
        if( n == 1 ) return;
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j + + )
                        printf("*");
        function( n-3 );
}
```

Solusi: Perhatikan komentar pada fungsi di bawah ini:

```
function (int n) {
   //constant time
   if( n == 1 ) return;
   //outer loop execute n times
   for(int i = 1 ; i <= n ; i + + )
       //inner loop executes n times
      for(int j = 1 ; j <= n ; j + + )
         //constant time
         printf("*" ) ;
      function( n-3 );
}
```

Pengulangan untuk kode ini jelas T(n) = T(n – 3) + cn^2 untuk beberapa konstanta c > 0 karena setiap panggilan mencetak n^2 tanda bintang dan memanggil dirinya sendiri secara rekursif pada n – 3. Menggunakan metode iteratif kita mendapatkan: T (n) = T(n – 3) + cn^2. Menggunakan teorema master Pengurangan dan Penaklukan, kita mendapatkan T(n) = (n^3).

Soal-29 Tentukan Θ batas untuk relasi perulangan: $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$

Solusi Dengan menggunakan teorema master Divide and Conquer, kita mendapatkan O($nlog^2n$).

Soal-30 Tentukan Θ batas untuk perulangan:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$$

Solusi Substitusi ke persamaan rekurensi, kita dapatkan:

$$T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn \leq k * n$$

dimana k adalah konstanta. Ini dengan jelas mengatakan Θ (n).

Soal-31 Tentukan batas untuk relasi perulangan: T(n) = T(⌈n/2⌉) + 7.

Solusi: Menggunakan Teorema Master kita mendapatkan: Θ (logn).

Soal-32 Buktikan bahwa waktu berjalan dari kode di bawah ini adalah Ω(logn).

```
void Read(int n) {
        int k = 1;
        while( k < n )
                k = 3*k;
}
```

Solusi Perulangan while akan berakhir setelah nilai 'k' lebih besar dari atau sama dengan nilai 'n'. Pada setiap iterasi nilai 'k' dikalikan 3. Jika i adalah jumlah iterasi, maka 'k' memiliki nilai 3^i setelah i iterasi. Loop dihentikan setelah mencapai i iterasi ketika 3^i n $\leftrightarrow$ i $\log_3$ n, yang menunjukkan bahwa i = Ω (logn).

Soal-33 Selesaikan perulangan berikut.

$$T(n) = \begin{cases} 1, & if\ n = 1 \\ T(n-1) + n(n-1), & if\ n \geq 2 \end{cases}$$

Solusi Dengan iterasi:

$$T(n) = T(n-2) + (n-1)(n-2) + n(n-1)$$
$$\cdots$$
$$T(n) = T(1) + \sum_{i=1}^{n} i(i-1)$$
$$T(n) = T(1) + \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i$$
$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$
$$T(n) = \Theta(n^3)$$

Catatan: Kita dapat menggunakan teorema master Pengurangan dan Penaklukan untuk masalah ini.

Soal-34 Perhatikan program berikut:

```
Fib[n]
if(n==0) then return 0
else if(n==1) then return 1
else return Fib[n-1]+Fib[n-2]
```

Solusi Relasi perulangan untuk running time program ini adalah: T(n) = T(n – 1) + T(n – 2) + c. Catatan T(n) memiliki dua panggilan perulangan yang menunjukkan pohon biner. Setiap langkah secara rekursif memanggil program untuk n dikurangi 1 dan 2, sehingga kedalaman pohon perulangan adalah O(n). Jumlah daun pada kedalaman n adalah 2^n karena ini adalah pohon biner penuh, dan setiap daun membutuhkan setidaknya O(1) perhitungan untuk faktor konstan. Waktu berjalan jelas eksponensial dalam n dan itu adalah O(2^n).

Soal-35 Waktu menjalankan program berikut?

```
function(n) {
        for(int i = 1; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j+ = i )
                        printf(" * ");
}
```

Solusi : Perhatikan komentar pada fungsi di bawah ini:

```
function (n) {
   //this loop executes n times
   for(int i = 1;  i <= n ; i + + )
      //this loop executes j times with j increase by the rate of i
      for(int j = 1 ; j <= n ; j+ = i )
         printf( " * " );
}
```

Dalam kode di atas, loop dalam dieksekusi n/i kali untuk setiap nilai i. Waktu berjalannya adalah

$$n \times (\sum_{i=1}^{n} n/i) = O(nlogn)$$

Soal-36 Apa kompleksitas dari $\sum_{i=1}^{n} log\, i$?

Solusi Menggunakan properti logaritmik, logxy = logx + logy, kita dapat melihat bahwa masalah ini setara dengan

$$\sum_{i=1}^{n} logi = log\,1 + log\,2 + \cdots + log\,n = log(1 \times 2 \times \ldots \times n) = log(n!) \leq log(n^n) \leq nlogn$$

Hal ini menunjukkan bahwa kompleksitas waktu = O(nlogn).

Soal-37 Berapa waktu berjalan dari fungsi rekursif berikut (ditentukan sebagai fungsi dari nilai input n)? Pertama tulis rumus perulangan dan kemudian temukan kerumitannya.

```
function(int n) {
        if(n <= 1) return;
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

Solusi Pertimbangkan komentar di fungsi di bawah ini:

```
function (int n) {
   //constant time
   if(n <= 1) return;
   //this loop executes with recursive loop of n/3 value
   for (int i=1 ; i <= 3; i++ )
      f(⌈n/3⌉);
}
```

Kita dapat mengasumsikan bahwa untuk analisis asimtotik k = $\lceil k \rceil$ untuk setiap bilangan bulat k $\geq$1. Perulangan untuk kode ini adalah $T(n) = 3T(\frac{n}{3}) + \Theta(1)$

Menggunakan teorema utama, kita mendapatkan T(n) = Θ (n).

Soal-38 Berapa waktu berjalan dari fungsi rekursif berikut (ditentukan sebagai fungsi dari nilai input n)? Pertama tulis rumus perulangan, dan tunjukkan solusinya menggunakan induksi.

```
function(int n) {
        if(n <= 1) return;

        for (int i=1 ; i <= 3 ; i++ )
                function (n − 1).
}
```

Solusi Perhatikan komentar pada fungsi di bawah ini:

```
function (int n) {
   //constant time
   if(n <= 1) return;
   //this loop executes 3 times with recursive call of n-1 value
   for (int i=1 ; i <= 3 ; i++ )
      function (n − 1).
}
```

Pernyataan if membutuhkan waktu yang konstan [O(1)]. Dengan loop for, kita mengabaikan loop overhead dan hanya menghitung tiga kali fungsi dipanggil secara rekursif. Ini menyiratkan pengulangan kompleksitas waktu:

$$T(n) = c, if\ n \leq 1;$$
$$= c + 3T(n - 1), if\ n > 1.$$

Menggunakan teorema master Pengurangan dan Penaklukan, kita mendapatkan T(n) = Θ (3^n).

Soal-39 Tulislah rumus rekursi untuk waktu berjalan T(n) dari fungsi yang kodenya di bawah ini.

```
function (int n) {
        if(n <= 1)  return;
        for(int i = 1; i < n; i + +)
                printf(" * ");
        function ( 0.8n ) ;
}
```

Solusi: Perhatikan komentar pada fungsi di bawah ini:

```
function (int n) {
   if(n <= 1)  return;   //constant time
   // this loop executes n times with constant time loop
   for(int i = 1; i < n; i + +)
      printf(" * ");
   //recursive call with 0.8n
   function ( 0.8n ) ;
}
```

Pengulangan untuk potongan kode ini adalah T(n) = T(.8n) + O(n) = T(4/5n) + O(n) =4/5 T(n) + O(n). Menerapkan teorema utama, kita mendapatkan T(n) = O(n).

Soal-40 Temukan kompleksitas perulangan: T(n) = 2T($\sqrt{n}$) + logn

Solusi Pengulangan yang diberikan tidak dalam format teorema master. Mari kita coba mengubahnya menjadi format teorema master dengan asumsi n = 2^m. Menerapkan logaritma di kedua sisi memberikan, logn = mlogl ⇒m = logn. Sekarang, fungsi yang diberikan menjadi:

$T(n) = T(2^m) = 2T(\sqrt{2^m}) + m = 2T\left(2^{\frac{m}{2}}\right) + m.$

Untuk membuatnya sederhana, kita asumsikan.

$S(m) = T(2^m) \Longrightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Longrightarrow S(m) = 2S\left(\frac{m}{2}\right) + m$

Menerapkan format teorema master akan menghasilkan S(m) = O(mlogm).
Jika kita substitusi m = logn kembali, T(n) = S(logn) = O((logn) loglogn).

Soal-41 Temukan kompleksitas perulangan: T(n) = T($\sqrt{n}$) + 1

Solusi Menerapkan logika Soal-40 memberikan $S(m) = S\left(\frac{m}{2}\right) + 1$
Menerapkan master teorema akan menghasilkan S(m) = O($logm$)
Mengganti m = logn, menghasilkan T(n) = S(logn) = O(loglogn).

Soal-42 Temukan kompleksitas perulangan: T(n) = 2T($\sqrt{n}$) + 1

Solusi Menerapkan logika Soal-40 memberikan: $S(m) = 2S\left(\frac{m}{2}\right) + 1$.
Menggunakan teorema induk hasil S(m)= O($m^{log_2^2}$) .
Mengganti m = logn menghasilkan T(n) =O(logn).

Soal-43 Temukan kompleksitas fungsi di bawah ini.

```
int Function (int n) {
        if(n <= 2) return 1;
        else return (Function (floor(sqrt(n))) + 1);
}
```

Solusi Perhatikan komentar pada fungsi di bawah ini:

```
int Function (int n) {
   if(n <= 2) return 1;              //constant time
   else                              // executes √n + 1 times
      return (Function (floor(sqrt(n))) + 1);
}
```

Untuk kode di atas, fungsi perulangan dapat diberikan sebagai: T(n) = T($\sqrt{n}$+1) . Ini sama dengan Soal-41.

Soal-44 Analisis waktu berjalan pseudo-code rekursif berikut sebagai fungsi dari n.

```
void function(int n) {
        if( n < 2 ) return;
        else counter = 0;
        for i = 1 to 8 do
                function (n/2);
        for i =1 to n^3 do
                counter = counter + 1;
}
```

Solusi Perhatikan komentar pada pseudo-code di bawah ini dan panggil waktu berjalan fungsi(n) sebagai T(n).

```
void function(int n) {
    if( n < 2 ) return;    //constant time
    else      counter = 0;
    // this loop executes 8 times with n value half in every call
    for i = 1 to 8 do
        function(n/2);
    // this loop executes n^3 times with constant time loop
    for i =1 to n^3 do
            counter = counter + 1;
}
```

T(n) dapat didefinisikan sebagai berikut:

$$T(n) = 1 \; if \; n < 2,$$
$$= 8T(\frac{n}{2}) + n^3 + 1 \; otherwise.$$

Menggunakan teorema master memberikan:

$$T(n) = \Theta(n^{log_2^8} logn) = \Theta(n^3 logn)$$

Soal-45 Temukan kompleksitas pseudocode di bawah ini:

```
temp  = 1
repeat
        for i = 1 to n
                temp =  temp + 1;
        n  = n/2;
until n <= 1
```

Solusi Perhatikan komentar pada pseudocode di bawah ini:

```
temp  = 1   //const time
repeat          // this loops executes n times
   for i = 1 to n
      temp =  temp + 1;
      //recursive call with n/2 value
      n  = n/2;
until n <= 1
```

Pengulangan untuk fungsi ini adalah T(n) = T(n/2) + n. Menggunakan teorema utama, kita mendapatkan T(n) = O(n).

Soal-46 Waktu menjalankan program berikut?

```
function(int n) {
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j * = 2 )
                        printf( " * " );
}
```

Solusi: Perhatikan komentar pada fungsi di bawah ini:

```
function(int n) {
    for(int i = 1 ; i <= n ; i + + ) // this loops executes n times
        // this loops executes logn times from our logarithms guideline
        for(int j = 1 ; j <= n ; j * = 2 )
            printf( " * " );
}
```

Kompleksitas program di atas adalah: O(nlogn).

Soal-47 Waktu menjalankan program berikut?

```
function(int n) {
        for(int i = 1 ; i <= n/3 ; i + + )
                for(int j = 1 ; j <= n ; j += 4 )
                        printf( " * " );
}
```

Solusi: Perhatikan komentar pada fungsi di bawah ini:

```
function(int n) {       // this loops executes n/3 times
    for(int i = 1 ; i <= n/3 ; i + + )
        // this loops executes n/4 times
        for(int j = 1 ; j <= n ; j += 4)
            printf( " * " );
}
```

Kompleksitas waktu program ini adalah: O(n2).

Soal-48 Temukan kompleksitas fungsi di bawah ini:

```
void function(int n) {
    if(n <=  1) return;
    if(n > 1) {
            printf (" * ");
            function( n/2 );
            function( n/2 );
        }
}
```

Solusi: Perhatikan komentar pada fungsi di bawah ini:

```
void function(int n) {
        if(n <= 1) return; //constant time
        if(n > 1) {
                //constant time
                printf (" * ");
                //recursion with n/2 value
                function( n/2 );
                //recursion with n/2 value
                function( n/2 );
        }
}
```

Pengulangan untuk fungsi ini adalah: $T(n) = 2T\left(\frac{n}{2}\right) + 1$.

Menggunakan teorema utama, kita mendapatkan T(n) = O(n).

Soal-49 Temukan kompleksitas fungsi di bawah ini:

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                    j = j/2;
                i=2*i;
        } // i
}
```

Solusi:

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                    j = j/2;   //logn code
                i=2*i; //logn times
        } // i
}
```

Kompleksitas Waktu:O(logn * logn) = O($\log^2 n$).

Soal-50 $\sum_{i \le k \le n}$ O(n), di mana O(n) singkatan dari orde n adalah:

(A) O(n)

(B) O(n^2)

(C) O(n^3)

(D) O($3n^2$)

(E) O($1.5n^2$)

Solusi: (B). $\sum_{i \le k \le n}$ = O(n) $\sum_{i \le k \le n}$ 1 = O(n^2).

Soal-51 Manakah dari tiga pernyataan dibawah ini yang benar?

I. $(n + k)^m = \Theta(n^m)$, di mana k dan m adalah konstanta
II. $2^{n+1} = O(2^n)$
III. $22^{n+1} = O(2^n)$

A. I dan II
B. I dan III
C. II dan III
D. I, II dan III

Solusi: (A). (I) $(n + k)^m = n^h + c1*n^{k-1} + ... k^m = \Theta(n^h)$ dan (II) $2^{n+1} = 2*2^n = O(2^n)$

Soal-52 Perhatikan fungsi berikut:

$f(n) = 2^n$
$g(n) = n!$
$h(n) = n^{\log n}$

Manakah pernyataan yang benar tentang perilaku asimtotik f(n), g(n), dan h(n) ?

(A) $f(n) = O(g(n))$; $g(n) = O(h(n))$
(B) $f(n) = \Omega(g(n))$; $g(n) = O(h(n))$
(C) $g(n) = O(f(n))$; $h(n) = O(f(n))$
(D) $h(n) = O(f(n))$; $g(n) = \Omega(f(n))$

Solusi: (D). Menurut kecepatan pertumbuhan: $h(n) < f(n) < g(n)$ ($g(n)$ secara asimtotik lebih besar dari $f(n)$, dan $f(n)$ secara asimtotik lebih besar dari $h(n)$). Kita dapat melihat dengan mudah urutan di atas dengan mengambil logaritma dari 3 fungsi yang diberikan: $\log n \log n < n < \log(n!)$. Perhatikan bahwa, $\log(n!) = O(n \log n)$.

Soal-53 Perhatikan segmen kode-C berikut:

```
int j=1, n;
while (j <=n)
        j = j*2;
```

Banyaknya perbandingan yang dibuat dalam eksekusi loop untuk setiap n > 0 adalah:

(A) $\text{ceil}(log_2^n) + 1$
(B) *n*

(C) $\text{ceil}(log_2^n)$
(D) $\text{floor}(log_2^n) + 1$

Solusi: (a). Mari kita asumsikan bahwa loop mengeksekusi k kali. Setelah langkah ke-k nilai j adalah 2^k. Mengambil logaritma di kedua sisi memberikan k = log_2^n . Karena kita melakukan satu perbandingan lagi untuk keluar dari loop, jawabannya adalah ceil $(log_2^n) + 1$.

Soal-54 Perhatikan segmen kode C berikut. Misalkan T(n) menyatakan berapa kali perulangan for dieksekusi oleh program pada input n. Manakah jawaban yang benar berdasarkan perintah dibawah ini?

```
int IsPrime(int n){
    for(int i=2;i<=sqrt(n);i++)
    if(n%i == 0){
        printf("Not Prime\n");
        return 0;
    }
    return 1;
}
```

(A) T(n) = O($\sqrt{n}$) dan T(n) = Ω($\sqrt{n}$)
(B) T(n) = O($\sqrt{n}$) dan T(n) = Ω (1)
(C) T(n) = O(n) dan T(n) = Ω ($\sqrt{n}$)
(D) Tidak satu pun di atas

Solusi: (B). Notasi Big O menggambarkan batas atas yang ketat dan notasi Big Omega menggambarkan batas bawah yang ketat untuk suatu algoritma. Perulangan for dalam pertanyaan dijalankan dengan waktu maksimum dan minimal 1 kali. Oleh karena itu, T(n) = O($\sqrt{n}$) dan T(n) = Ω (1).

Soal-55 Dalam fungsi C berikut, misalkan n ≥ m. Berapa banyak panggilan rekursif yang dibuat oleh fungsi ini?

```
int gcd(n,m){
        if (n%m ==0)
            return m;
        n = n%m;
        return gcd(m,n);
}
```

(A) $\Theta(log_2^n)$

(B) Ω (n)
(C) $\Theta(log_2 log_2^n)$
(D) Θ (n)

Solusi: Tidak ada opsi yang benar. Notasi Big O menggambarkan batas atas yang ketat dan notasi Big Omega menggambarkan batas bawah yang ketat untuk suatu algoritma. Untuk m = 2 dan untuk semua n = 2^i, waktu berjalan adalah O(1) yang bertentangan dengan setiap opsi.

Soal-56 Misalkan T(n) = 2T(n/2) + n, T(O)=T(1)=1. Manakah jawaban yang tidak sesuai?
(A) T(n) = O(n^2)
(B) T(n) = Θ (nlogn)
(C) T(n) = Q(n^2)
(D) T(n) = O(nlogn)

Solusi: (C). Notasi Big O menggambarkan batas atas yang ketat dan notasi Big Omega menggambarkan batas bawah yang ketat untuk suatu algoritma. Berdasarkan teorema utama, kita mendapatkan T(n) = Θ (nlogn). Hal ini menunjukkan bahwa batas bawah yang ketat dan batas atas yang ketat adalah sama. Itu berarti, O(nlogn) dan Ω (nlogn) benar untuk pengulangan yang diberikan. Jadi pilihan (C) salah.

Soal-57 Temukan kompleksitas pada fungsi di bawah ini:

```
function(int n) {
    for (int i = 0; i<n; i++)
        for(int j=i; j<i*i; j++)
            if (j %i == 0){
                for (int k = 0; k < j; k++)
                    printf(" * ");
            }

}
```

Solusi:

```
function(int n) {
   for (int i = 0; i<n; i++)                  // Executes n times
      for(int j=i; j<i*i; j++)                // Executes n*n times
         if (j %i == 0){
            for (int k = 0; k < j; k++)       // Executes j times = (n*n) times
               printf(" * ");
         }
}
```

Kompleksitas Waktu: O(n^5).

Soal-58 Untuk menghitung 9n, berikan algoritma dan diskusikan kompleksitasnya.
Solusi: Mulailah dengan 1 dan kalikan dengan 9 hingga mencapai 9^n.

Kompleksitas Waktu: Ada n – 1 perkalian dan masing-masing membutuhkan waktu yang konstan memberikan algoritma Θ (n).

Soal-59 Untuk Soal-58, dapatkah kita meningkatkan kompleksitas waktu?
Solusi: Lihat bab Divide and Conquer.

Soal-60 Temukan kompleksitas waktu perulangan $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$.

Solusi: Mari kita selesaikan masalah ini dengan metode menebak. Ukuran total pada setiap tingkat pohon perulangan kurang dari n, jadi kita menduga bahwa f(n) = n akan mendominasi. Asumsikan untuk semua i < n bahwa $c_1 n$ T(i) < $c_2 n$. Kemudian,

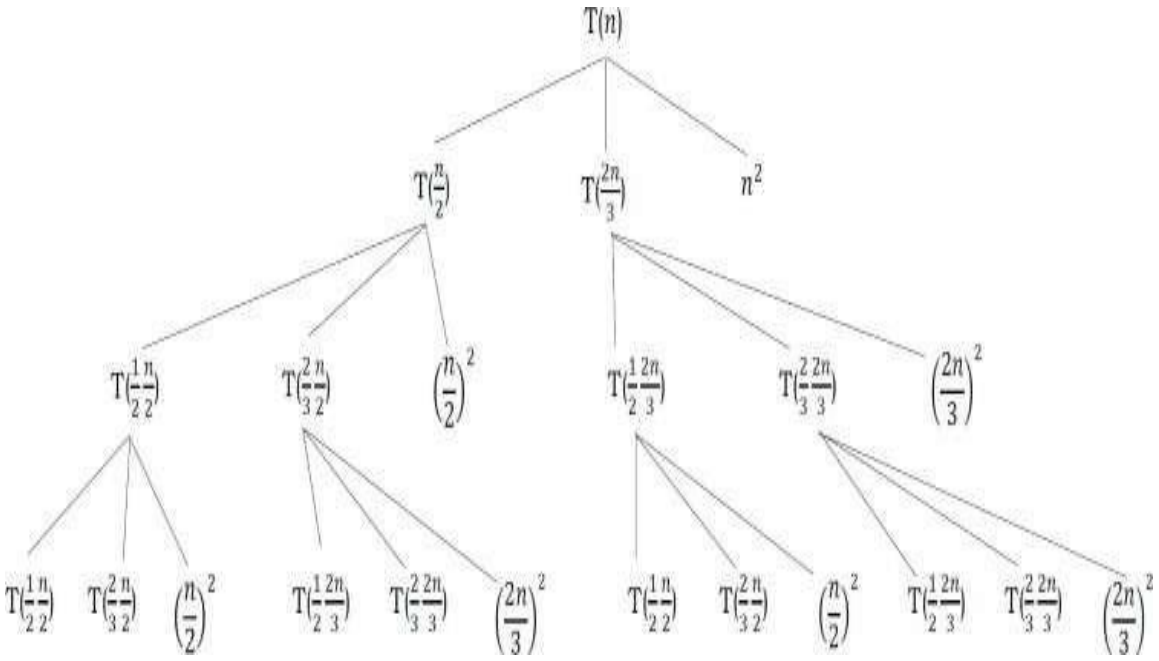
$$\begin{aligned} c_1\frac{n}{2} + c_1\frac{n}{4} + c_1\frac{n}{8} + kn \quad &\leq T(n) \leq \quad c_2\frac{n}{2} + c_2\frac{n}{4} + c_2\frac{n}{8} + kn \\ c_1 n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_1}) \quad &\leq T(n) \leq \quad c_2 n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_2}) \\ c_1 n(\frac{7}{8} + \frac{k}{c_1}) \quad &\leq T(n) \leq \quad c_2 n(\frac{7}{8} + \frac{k}{c_2}) \end{aligned}$$

Jika $c_1 \geq 8k$ dan $c_2 \leq 8k$, maka $c_1 n = T(n) = c_2 n$. Jadi, T(n) = Θ (n). Secara umum, jika Anda memiliki beberapa panggilan rekursif, jumlah argumen untuk panggilan tersebut kurang dari n (dalam hal ini $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} < n$), dan f(n) cukup besar, jawaban yang tepat adalah T(n) = Θ (f(n)).

Soal-61 Selesaikan relasi perulangan berikut menggunakan metode pohon rekursi:

$$T(n) = T(\frac{n}{2}) + T(\frac{2n}{3}) + n^2$$

Solusi: Berapa banyak pekerjaan yang kita lakukan di setiap tingkat pohon rekursi?



Di level 0, kita mengambil n2 waktu. Pada level 1, dua submasalah membutuhkan waktu:

$$\left(\frac{1}{2}n\right)^2+\left(\frac{2}{3}n\right)^2=\left(\frac{1}{4}+\frac{4}{9}\right)n^2=\left(\frac{25}{36}\right)n^2$$

Pada level 2, empat submasalah memiliki ukuran $\frac{1}{2}\frac{n}{2}$, $\frac{2}{3}\frac{n}{2}$, $\frac{1}{2}\frac{2n}{3}$ dan $\frac{2}{3}\frac{2n}{3}$ masing-masing.

Kedua submasalah ini membutuhkan waktu:

$$\left(\frac{1}{4}n\right)^2+\left(\frac{1}{3}n\right)^2+\left(\frac{1}{3}\right)n^2+\left(\frac{4}{9}\right)n^2=\frac{625}{1296}n^2=\left(\frac{25}{36}\right)^2n^2$$

Demikian pula jumlah pekerjaan pada tingkat k paling banyak $\left(\frac{25}{36}\right)^k n^2$.

Biarkan $\alpha = \frac{25}{36}$, total runtime adalah:

$$\begin{aligned} T(n) &\leq \sum_{k=0}^{\infty} \alpha^k n^2 \\ &= \frac{1}{1-\alpha} n^2 \\ &= \frac{1}{1-\frac{25}{36}} n^2 \\ &= \frac{1}{\frac{11}{36}} n^2 \\ &= \frac{36}{11} n^2 \\ &= O(n^2) \end{aligned}$$

Artinya, level pertama menyediakan fraksi konstan dari total runtime.

Soal-62 Urutkan fungsi-fungsi berikut berdasarkan urutan pertumbuhannya: (n + 1)!, n!, 4^n, n × 3^n, $3^n + n^2 + 20n$, $(\frac{3}{2})^n$, $n^2 + 200$, 20n + 500, 2^{lgn}, $n^{2/3}$, 1.

Solusi:

Function	Rate of Growth	
$(n+1)!$	$O(n!)$	Decreasing rate of growths
$n!$	$O(n!)$	
4^n	$O(4^n)$	
$n \times 3^n$	$O(n3^n)$	
$3^n + n^2 + 20n$	$O(3^n)$	
$(\frac{3}{2})^n$	$O((\frac{3}{2})^n)$	
$4n^2$	$O(n^2)$	
4^{lgn}	$O(n^2)$	
$n^2 + 200$	$O(n^2)$	
$20n + 500$	$O(n)$	
2^{lgn}	$O(n)$	
$n^{2/3}$	$O(n^{2/3})$	
1	$O(1)$	

Soal-63 Temukan kompleksitas fungsi di bawah ini:

```
function(int n) {
   int sum = 0;
   for (int i = 0; i<n; i++)
      if (i>j)
         sum = sum +1;
      else {
          for (int k = 0; k < n; k++)
              sum = sum -1;
       }
   }
}
```

Solusi: Perhatikan kasus terburuk.

```
function(int n) {
   int sum = 0;
      for (int i = 0; i<n; i++)                  // Executes n times
      if (i>j)
         sum = sum +1;                           // Executes n times
      else {
           for (int k = 0; k < n; k++)           // Executes n times
              sum = sum -1;
      }
   }
}
```

Kompleksitas Waktu: O(n2).

Soal-64 Bisakah kita sebut $3^{n^{0.75}} = O(3^n)$??

Solusi: Ya: karena $3^{n^{0.75}} < 3^{n^1}$

Soal-65 Bisakah kita sebut $2^{3n} = O(2^n)$?

Solusi: Tidak: karena $2^{3n} = (2^3)^n = 8^n$ tidak kurang dari 2^n.

BAB 2
REKURSI DAN BACKTRACKING

2.1 PENDAHULUAN

Dalam bab ini, kita akan melihat salah satu topik penting, "rekursi", yang akan digunakan di hampir setiap bab, dan juga "mundur" relatifnya.

2.2 APA ITU REKURSI?

Setiap fungsi yang memanggil dirinya sendiri disebut rekursif. Metode rekursif memecahkan masalah dengan memanggil salinan dirinya sendiri untuk mengerjakan masalah yang lebih kecil. Ini disebut langkah rekursi. Langkah rekursi dapat menghasilkan lebih banyak panggilan rekursif seperti itu.

Penting untuk memastikan bahwa rekursi berakhir. Setiap kali fungsi memanggil dirinya sendiri dengan versi masalah asli yang sedikit lebih sederhana. Urutan masalah yang lebih kecil pada akhirnya harus bertemu pada kasus dasar.

2.3 MENGAPA REKURSI?

Rekursi adalah teknik yang berguna dipinjam dari matematika. Kode rekursif umumnya lebih pendek dan lebih mudah ditulis daripada kode berulang. Umumnya, loop diubah menjadi fungsi rekursif ketika dikompilasi atau diinterpretasikan. Rekursi paling berguna untuk tugas-tugas yang dapat didefinisikan dalam hal subtugas serupa. Misalnya, masalah sortir, pencarian, dan traversal sering kali memiliki solusi rekursif sederhana.

2.4 FORMAT FUNGSI REKURSI

Fungsi rekursif yaitu melakukan sebagian tugas dengan memanggil dirinya sendiri untuk melakukan subtugas. Pada titik tertentu, fungsi tersebut dapat menemukan subtugas yang dijalankannya tanpa memanggil dirinya sendiri. Dalam kasus ini, jika fungsinya tidak berulang, disebut kasus dasar. Yang pertama, di mana fungsi memanggil dirinya sendiri untuk melakukan subtugas, disebut sebagai kasus ekursif. Kita dapat menulis semua fungsi rekursif menggunakan format:

```
if(test for the base case)
   return some base case value
else if(test for another base case)
   return some other base case value
// the recursive case
else
   return (some work and then a recursive call)
```

Sebagai contoh perhatikan fungsi faktorial: n! adalah produk dari semua bilangan bulat antara n dan 1. Definisi faktorial rekursif terlihat seperti:

$$n! = 1, \quad \text{if } n = 0$$
$$n! = n * (n - 1)! \text{ if } n > 0$$

Definisi ini dapat dengan mudah diubah menjadi implementasi rekursif. Di sini masalahnya adalah menentukan nilai n!, dan submasalahnya adalah menentukan nilai (n – l)!. Dalam kasus rekursif, ketika n lebih besar dari 1, fungsi memanggil dirinya sendiri untuk menentukan nilai (n – l)! dan mengalikannya dengan n.

Dalam kasus dasar, ketika n adalah 0 atau 1, fungsi hanya mengembalikan 1. Contohnya sebagai berikut:

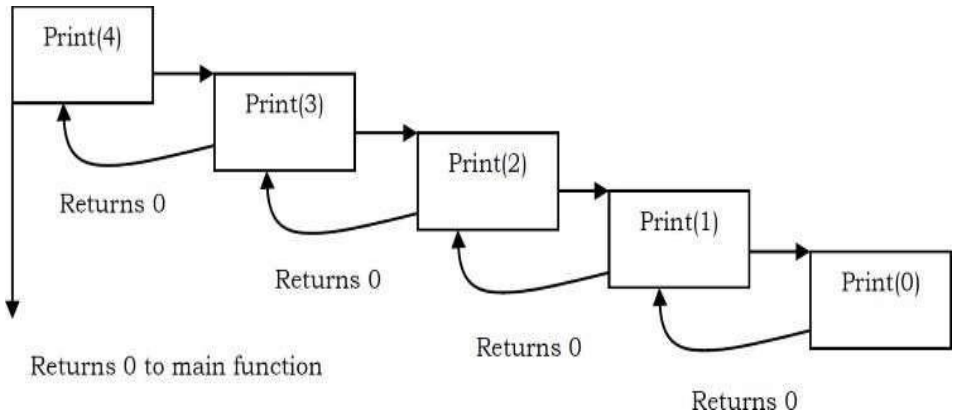
```
// calculates factorial of a positive integer
int Fact(int n) {
   if(n == 1)    // base cases: fact of 0 or 1 is 1
      return 1;
   else if(n == 0)
      return 1;
   else     // recursive case: multiply n by (n – 1) factorial
      return n*Fact(n-1);
}
```

2.5 REKURSI DAN MEMORI (VISUALISASI)

Setiap panggilan rekursif membuat salinan baru dari metode itu (sebenarnya hanya variabel) di memori. Setelah metode berakhir (yaitu, mengembalikan beberapa data), salinan metode pengembalian tersebut dihapus dari memori. Solusi rekursif terlihat sederhana tetapi visualisasi dan penelusuran membutuhkan waktu. Untuk pemahaman yang lebih baik, mari kita perhatikan contoh berikut.

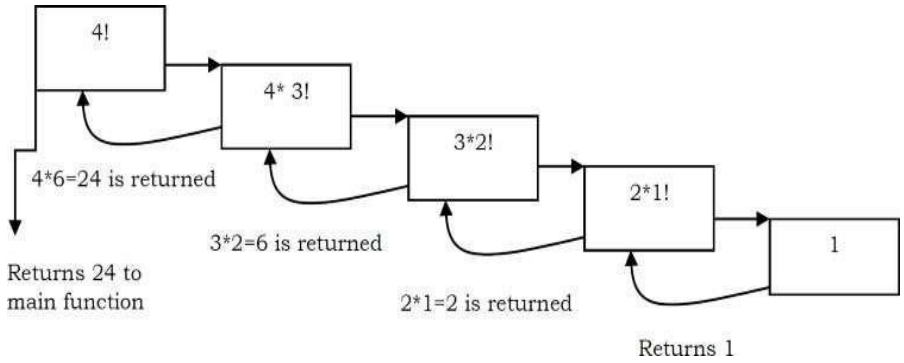
```
//print numbers 1 to n backward
int Print(int n) {
 if( n = = 0) // this is the terminating base case
    return 0;
 else {
    printf ("%d",n);
    return Print(n-1);       // recursive call to itself again
 }
}
```

Untuk contoh ini, jika kita memanggil fungsi print dengan n=4, secara visual tugas memori kita mungkin terlihat seperti:



Gambar 2.1 Prosedur pernyataan diatas

Sekarang, mari kita perhatikan fungsi faktorial kita. Visualisasi fungsi faktorial dengan n=4 akan terlihat seperti:



Gambar 2.2 Prosedur faktorial

2.6 REKURSI VERSUS ITERASI

Saat membahas rekursi, pertanyaan mendasar yang muncul di benak adalah: jalan mana yang lebih baik? - iterasi atau rekursi? Jawaban atas pertanyaan ini tergantung pada apa yang kita coba lakukan. Pendekatan rekursif mencerminkan masalah yang Kita coba pecahkan. Pendekatan rekursif membuatnya lebih mudah untuk memecahkan masalah yang mungkin tidak memiliki jawaban yang paling jelas. Namun, rekursi menambahkan overhead untuk setiap panggilan rekursif (membutuhkan ruang pada bingkai tumpukan).

Pengulangan

- Berakhir ketika kasus dasar tercapai.
- Setiap panggilan rekursif membutuhkan ruang ekstra pada bingkai tumpukan (memori).
- Jika kita mendapatkan rekursi tak terbatas, program mungkin kehabisan memori dan mengakibatkan tumpukan overflow.
- Solusi untuk beberapa masalah lebih mudah dirumuskan secara rekursif.

Pengulangan

- Berakhir ketika suatu kondisi terbukti salah.
- Setiap iterasi tidak membutuhkan ruang ekstra.
- Perulangan tak terbatas dapat berulang selamanya karena tidak ada memori tambahan yang dibuat.
- Solusi berulang untuk suatu masalah mungkin tidak selalu sejelas solusi rekursif.

2.7 CATATAN TENTANG REKURSI

- Algoritma rekursif memiliki dua jenis kasus, kasus rekursif dan kasus dasar.
- Pada setiap kasus, fungsi rekursif harus berakhir pada kasus dasar.
- Umumnya, solusi iteratif lebih efisien daripada solusi rekursif [karena overhead pemanggilan fungsi].
- Algoritma rekursif dapat diimplementasikan tanpa pemanggilan fungsi rekursif menggunakan tumpukan, tetapi biasanya lebih banyak masalah daripada nilainya. Itu berarti setiap masalah yang dapat diselesaikan secara rekursif juga dapat diselesaikan secara iteratif.
- Untuk beberapa masalah, tidak ada algoritma iteratif yang jelas.
- Beberapa masalah yang paling cocok hanya untuk solusi rekursif sementara tidak ada yang lain.

2.8 CONTOH ALGORITMA REKURSI

- Deret Fibonacci, Temuan Faktorial
- Gabung Sortir, Sortir Cepat
- Pencarian Biner
- Traversal Pohon dan banyak Masalah Pohon: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] dan BFS [Breadth First Search]
- Contoh Pemrograman Dinamis
- Algoritma Bagi dan Taklukkan
- Menara Hanoi
- Algoritma Backtracking [akan kita bahas di bagian selanjutnya]

2.9 REKURSI: MASALAH & SOLUSI

Dalam bab ini kita membahas beberapa masalah dengan rekursi dan kita akan membahas sisanya di bab lain. Setelah Anda selesai membaca seluruh isi buku, Anda akan menghadapi banyak masalah rekursi.

Soal-1 Diskusikan teka-teki Menara Hanoi.

Solusi: Menara Hanoi adalah teka-teki matematika. Ini terdiri dari tiga batang (atau pasak atau menara), dan sejumlah disk dengan ukuran berbeda yang dapat

meluncur ke batang apa pun. Teka-teki dimulai dengan disk pada satu batang dalam urutan ukuran, yang terkecil di bagian atas, sehingga membuat bentuk kerucut. Tujuan dari teka-teki ini adalah untuk memindahkan seluruh tumpukan ke batang lain dengan memenuhi aturan sebagai berikut:

- Hanya satu disk yang dapat dipindahkan pada satu waktu.
- Setiap gerakan terdiri dari mengambil piringan atas dari salah satu tongkat dan menggesernya ke tongkat lain, di atas piringan lain yang mungkin sudah ada pada tongkat itu.
- Tidak ada disk yang dapat ditempatkan di atas disk yang lebih kecil.

Algoritma:

- Pindahkan n – 1 disk teratas dari Source ke Auxiliary tower,
- Pindahkan disk ke-n dari menara Sumber ke Tujuan,
- Pindahkan n – 1 disk dari Auxiliary tower ke Destination tower.
- Memindahkan n – 1 disk teratas dari Source ke menara Auxiliary dapat dianggap lagi sebagai masalah baru dan dapat diselesaikan dengan cara yang sama. Setelah kita menyelesaikan Towers of Hanoi dengan tiga disk, kita dapat menyelesaikannya dengan sejumlah disk dengan algoritma di atas.

```
void TowersOfHanoi(int n, char frompeg, char topeg, char auxpeg) {
    /* If only 1 disk, make the move and return */
    if(n==1) {
        printf("Move disk 1 from peg %c to peg %c",frompeg, topeg);
        return;
    }
    /* Move top n-1 disks from A to B, using C as auxiliary */
    TowersOfHanoi(n-1, frompeg, auxpeg, topeg);
    /* Move remaining disks from A to C */
    printf("\nMove disk %d from peg %c to peg %c", n, frompeg, topeg);
    /* Move n-1 disks from B to C using A as auxiliary */
    TowersOfHanoi(n-1, auxpeg, topeg, frompeg);
}
```

Soal-2 Diberikan sebuah array, periksa apakah array diurutkan dengan rekursi.

Solusi:

```
int isArrayInSortedOrder(int A[],int n){
    if(n == 1)
      return 1;
    return (A[n-1] < A[n-2])?0:isArrayInSortedOrder(A,n-1);
}
```

Kompleksitas Waktu: O(n). Kompleksitas Ruang: O(n) untuk ruang tumpukan rekursif.

2.10 APA ITU BACKTRACKING?

Backtracking adalah perbaikan dari pendekatan brute force. Perbaikan ini secara sistematis mencari solusi dari masalah diantara semua opsi yang tersedia. Dalam backtracking, kita mulai dengan satu pilihan yang memungkinkan dari banyak pilihan yang tersedia dan mencoba untuk memecahkan masalah jika kita mampu memecahkan masalah dengan langkah yang dipilih maka kita akan mencetak solusi lain kita akan backtrack dan memilih beberapa pilihan lain dan mencoba untuk menyelesaikannya. Jika tidak ada jika opsi berhasil, kita akan mengklaim bahwa tidak ada solusi untuk masalah tersebut.

Backtracking adalah bentuk rekursi. Skenario pada umumnya adalah Anda dihadapkan pada sejumlah opsi, dan Anda harus memilih salah satunya. Setelah Anda membuat pilihan, Anda akan mendapatkan satu set baru pilihan; set pilihan apa yang Anda dapatkan tergantung pada pilihan apa yang Anda buat. Prosedur ini diulangi berulang-ulang sampai Anda mencapai keadaan akhir. Jika Anda membuat urutan pilihan yang baik, keadaan akhir Anda adalah keadaan tujuan; jika Anda tidak melakukannya, bukan itu.

Backtracking dapat dianggap sebagai metode traversal pohon/grafik selektif. Pohon adalah cara untuk mewakili beberapa posisi awal (simpul akar) dan status tujuan akhir (salah satu daun). Backtracking memungkinkan kita untuk menghadapi situasi di mana pendekatan brute force mentah akan dipecah menjadi sejumlah opsi yang mustahil untuk dipertimbangkan. Backtracking adalah semacam kekuatan kasar yang disempurnakan. Di setiap node, kita menghilangkan pilihan yang jelas tidak mungkin dan melanjutkan memeriksa secara rekursif yang hanya memiliki potensi.

Yang menarik dari backtracking adalah bahwa kita hanya mencadangkan sejauh yang diperlukan untuk mencapai titik keputusan sebelumnya dengan alternatif yang belum dijelajahi. Secara umum, itu akan menjadi titik keputusan terbaru. Akhirnya, semakin banyak poin keputusan ini yang telah sepenuhnya dieksplorasi, dan kita harus mundur lebih jauh. Jika kita mundur sepenuhnya ke keadaan awal kita dan telah menjelajahi semua alternatif dari sana, kita dapat menyimpulkan bahwa masalah tertentu tidak dapat dipecahkan. Dalam kasus seperti itu, kita akan melakukan semua pekerjaan rekursi lengkap dan mengetahui bahwa tidak ada solusi yang mungkin.

- Terkadang algoritma terbaik untuk suatu masalah adalah mencoba semua kemungkinan.
- Ini selalu lambat, tetapi ada alat standar yang dapat digunakan untuk membantu.
- Alat: algoritma untuk menghasilkan objek dasar, seperti string biner [2n kemungkinan untuk n-bit string], permutasi [n!], kombinasi [n!/r!(n – r)!], string umum [k –ary string dengan panjang n memiliki kn kemungkinan], dll...
- Backtracking mempercepat pencarian lengkap dengan memangkas.

2.11 CONTOH ALGORITMA *BACKTRACKING*

- String Biner: menghasilkan semua string biner

- Membangkitkan k – ary String
- Masalah N-Queens
- Masalah Ransel
- String Umum
- Siklus Hamilton [lihat bab Grafik]
- Soal Pewarnaan Grafik

2.12 *BACKTRACKING:* MASALAH & SOLUSI

Soal-3 Hasilkan semua string n bit. Asumsikan A[0..n – 1] adalah array berukuran n.

Solusi :

```
void Binary(int n) {
    if(n < 1)
        printf("%s", A);              //Assume array A is a global variable
    else {
        A[n-1] = 0;
        Binary(n - 1);
        A[n-1] = 1;
        Binary(n - 1);
    }
}
```

Biarkan T(n) menjadi waktu berjalan biner(n). Asumsikan fungsi printf membutuhkan waktu O(1).

$$T(n) = \begin{cases} c, & \text{if } n < 0 \\ 2T(n-1) + d, & \text{otherwise} \end{cases}$$

Dengan menggunakan teorema Subtraction and Conquer Master, kita mendapatkan: T(n) = $O(2^n)$. Ini berarti algoritma untuk menghasilkan bit-string optimal.

Soal-4 Hasilkan semua string dengan panjang n yang ditarik dari 0... k – 1.

Solusi: Mari kita asumsikan kita menyimpan string k-ary saat ini dalam array A[0.. n – 1]. Panggil fungsi k- string(n, k):

```
void k-string(int n, int k) {
    //process all k-ary strings of length m
    if(n < 1)
       printf("%s",A);              //Assume array A is a global variable
    else {
       for (int j = 0 ; j < k ; j++) {
            A[n-1] = j;
            k-string(n- 1, k);
       }
    }
}
```

Misalkan T(n) adalah waktu berjalan dari k – string(n). Kemudian,

$$T(n) = \begin{cases} c, & if\ n < 0 \\ kT(n-1) + d, & otherwise \end{cases}$$

Dengan menggunakan teorema Subtraction and Conquer Master, kita mendapatkan: T(n) = O(kn).

Catatan: Untuk masalah lebih lanjut, lihat bab Algoritma String.

Soal-5 Temukan panjang sel-sel yang terhubung dari 1s (daerah) dalam matriks Os dan 1s: Diberikan sebuah matriks, yang masing-masing mungkin 1 atau 0. Sel-sel terisi yang terhubung membentuk suatu daerah. Dua sel dikatakan terhubung

Sample Input: 11000 Sample Output: 5
01100
00101
10001
01011

jika mereka berdekatan satu sama lain secara horizontal, vertikal atau diagonal. Mungkin ada beberapa daerah dalam matriks. Bagaimana cara Anda menemukan wilayah terbesar (dalam hal jumlah sel) dalam matriks?

Solusi: Ide paling sederhana adalah: untuk setiap lokasi, lintasi di semua 8 arah dan di masing-masing arah tersebut lacak wilayah maksimum yang ditemukan.

```
int getval(int (*A)[5],int i,int j,int L, int H){
    if (i< 0 || i >= L || j< 0 || j >= H)
       return 0;
    else
       return A[i][j];
}
void findMaxBlock(int (*A)[5], int r, int c,int L,int H,int size, bool **cntarr,int &maxsize){
    if ( r >= L || c >= H)
       return;
    cntarr[r][c]=true;
    size++;
    if (size > maxsize)
       maxsize = size;
    //search in eight directions
    int direction[][2]={{-1,0},{-1,-1},{0,-1},{1,-1},{1,0},{1,1},{0,1},{-1,1}};
    for(int i=0; i<8; i++)      {
       int newi =r+direction[i][0];
       int newj=c+direction[i][1];
       int val=getval (A,newi,newj,L,H);
       if (val>0  && (cntarr[newi][newj]==false)){
          findMaxBlock(A,newi,newj,L,H,size,cntarr,maxsize);
       }
    }
    cntarr[r][c]=false;
}
int getMaxOnes(int (*A)[5], int rmax, int colmax){
    int maxsize=0;
    int size=0;
    bool **cntarr=create2darr(rmax,colmax);
    for(int i=0; i< rmax; i++){
       for(int j=0; j< colmax; j++){
          if (A[i][j] == 1){
             findMaxBlock(A,i,j,rmax,colmax, 0,cntarr,maxsize);
          }
       }
    }
    return maxsize;
}
```

Contoh Panggilan:

```
int zarr[][5]={{1,1,0,0,0},{0,1,1,0,1},{0,0,0,1,1},{1,0,0,1,1},{0,1,0,1,1}};
cout << "Number of maximum 1s are " << getMaxOnes(zarr,5,5) << endl;
```

Soal-6 Selesaikan perulangan $T(n) = 2T(n - 1) + 2^n$.

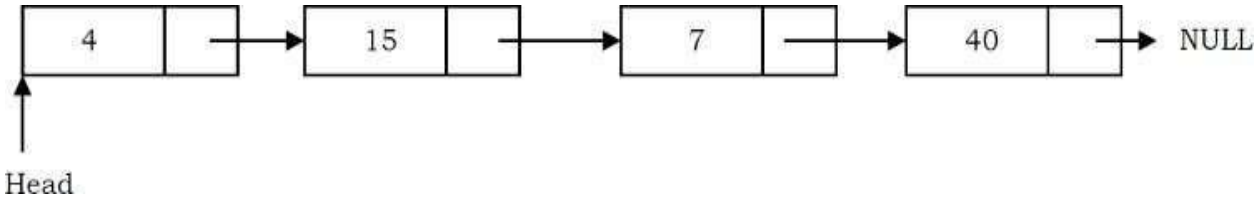
Solusi: Pada setiap tingkat pohon perulangan, jumlah masalah adalah dua kali lipat dari tingkat sebelumnya, sedangkan jumlah pekerjaan yang dilakukan di setiap masalah adalah setengah dari tingkat sebelumnya. Secara formal, level ke-i memiliki masalah 2i, masing-masing membutuhkan pekerjaan 2^{n-i}. Jadi tingkat ke-i membutuhkan tepat 2^n kerja. Kedalaman pohon ini adalah n, karena pada tingkat ke-i, panggilan asal adalah $T(n - i)$. Jadi kompleksitas total untuk T(n) adalah $T(n2^n)$.

BAB 3
DAFTAR TERTAUT
(LINKED LIST)

3.1 APA ITU DAFTAR TERTAUT?

Daftar tertaut adalah struktur data yang digunakan untuk menyimpan kumpulan data. Daftar tertaut memiliki properti berikut:

- Elemen yang berurutan dihubungkan oleh pointer
- Elemen terakhir menunjuk ke NULL
- Dapat tumbuh atau menyusut dalam ukuran selama eksekusi program
- Dapat dibuat selama diperlukan (sampai memori sistem habis)
- Tidak membuang-buang ruang memori (tetapi membutuhkan memori ekstra untuk pointer). Ini mengalokasikan memori saat daftar bertambah.



Gambar 3.1 Linked List

3.2 DAFTAR TERTAUT ADT

Operasi berikut membuat daftar tertaut menjadi ADT:

Operasi Daftar Tertaut Utama

- Sisipkan: menyisipkan elemen ke dalam daftar
- Hapus: menghapus dan mengembalikan elemen posisi tertentu dari daftar

Operasi Daftar Tertaut Bantu

- Hapus Daftar: menghapus semua elemen daftar (membuang daftar)
- Hitung: mengembalikan jumlah elemen dalam daftar
- Temukan simpul ke-n dari akhir daftar

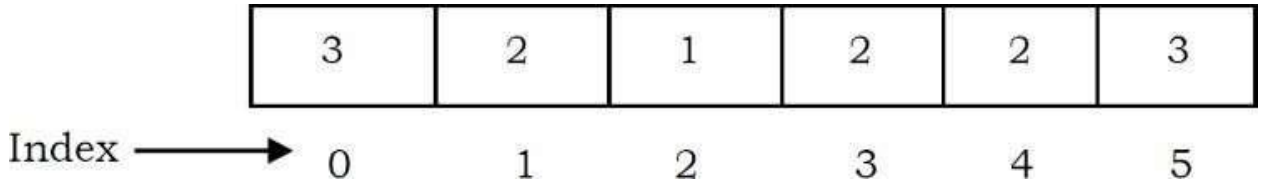
3.3 MENGAPA DAFTAR TERTAUT?

Ada banyak struktur data lain yang melakukan hal yang sama seperti daftar tertaut. Sebelum membahas daftar tertaut, penting untuk memahami perbedaan antara daftar tertaut dan larik. Daftar tertaut dan larik digunakan untuk menyimpan kumpulan data, dan karena

keduanya digunakan untuk tujuan yang sama, kita perlu membedakan penggunaannya. Berarti dalam kasus mana array cocok dan dalam kasus apa daftar tertaut yang cocok.

3.4 ARRAY

Satu blok memori dialokasikan untuk seluruh array untuk menampung elemen-elemen array. Elemen array dapat diakses dalam waktu yang konstan dengan menggunakan indeks elemen tertentu sebagai subskrip.



Gambar 3.2 blok memori dialokasikan ke Array

Mengapa Waktu Konstan untuk Mengakses Elemen Array?

Untuk mengakses elemen array, alamat elemen dihitung sebagai offset dari alamat dasar array dan satu perkalian diperlukan untuk menghitung apa yang seharusnya ditambahkan ke alamat dasar untuk mendapatkan alamat memori elemen. Pertama ukuran elemen dari tipe data tersebut dihitung dan kemudian dikalikan dengan indeks elemen untuk mendapatkan nilai yang akan ditambahkan ke alamat dasar.

Proses ini membutuhkan satu perkalian dan satu penambahan. Karena kedua operasi ini membutuhkan waktu yang konstan, kita dapat mengatakan bahwa akses array dapat dilakukan dalam waktu yang konstan.

Keuntungan dari Array

- Sederhana dan mudah digunakan
- Akses lebih cepat ke elemen (akses konstan)

Kekurangan Array

- Mengalokasikan semua memori yang dibutuhkan di depan dan membuang-buang ruang memori untuk indeks dalam array yang kosong.
- Ukuran tetap: Ukuran larik bersifat statis (tentukan ukuran larik sebelum menggunakannya).
- Alokasi satu blok: Untuk mengalokasikan array itu sendiri di awal, terkadang tidak mungkin untuk mendapatkan memori untuk array yang lengkap (jika ukuran array besar).
- Penyisipan berbasis posisi kompleks: Untuk menyisipkan elemen pada posisi tertentu, kita mungkin perlu menggeser elemen yang ada. Ini akan membuat posisi bagi kita

untuk memasukkan elemen baru pada posisi yang diinginkan. Jika posisi di mana kita ingin menambahkan elemen di awal, maka operasi pemindahan lebih mahal.

Array Dinamis

Array dinamis (juga disebut sebagai array yang dapat ditumbuhkan, array yang dapat diubah ukurannya, tabel dinamis, atau daftar array) adalah akses acak, struktur data daftar ukuran variabel yang memungkinkan elemen ditambahkan atau dihapus.

Salah satu cara sederhana untuk mengimplementasikan array dinamis adalah memulai dengan beberapa array ukuran tetap. Segera setelah array itu menjadi penuh, buat array baru dua kali lipat ukuran array asli. Demikian pula, kurangi ukuran array menjadi setengah jika elemen dalam array kurang dari setengah.

Catatan: *Kita akan melihat implementasi untuk array dinamis di Tumpukans, Queues dan Hashing bab.*

Keuntungan dari Daftar Tertaut

Daftar tertaut memiliki kelebihan dan kekurangan. Keuntungan dari daftar tertaut adalah mereka dapat diperluas dalam waktu yang konstan. Untuk membuat array, kita harus mengalokasikan memori untuk sejumlah elemen tertentu. Untuk menambahkan lebih banyak elemen ke array saat penuh, kita harus membuat array baru dan menyalin array lama ke dalam array baru. Ini bisa memakan banyak waktu.

Kita dapat mencegahnya dengan mengalokasikan banyak ruang pada awalnya, tetapi kemudian kita mungkin mengalokasikan lebih dari yang kita butuhkan dan membuang-buang memori. Dengan daftar tertaut, kita dapat memulai dengan ruang hanya untuk satu elemen yang dialokasikan dan menambahkan elemen baru dengan mudah tanpa perlu melakukan penyalinan dan pengalokasian ulang.

Kerugian Daftar Tertaut

Ada sejumlah masalah dengan daftar tertaut. Kerugian utama dari daftar tertaut adalah waktu akses ke elemen individual. Array adalah akses acak, yang berarti dibutuhkan O(1) untuk mengakses elemen apa pun dalam array. Daftar tertaut mengambil O(n) untuk akses ke elemen dalam daftar dalam kasus terburuk. Keuntungan lain dari array dalam waktu akses adalah lokalitas spasial dalam memori. Array didefinisikan sebagai blok memori yang berdekatan, sehingga setiap elemen array akan secara fisik berada di dekat tetangganya. Ini sangat diuntungkan dari metode caching CPU modern.

Meskipun alokasi penyimpanan dinamis merupakan keuntungan besar, overhead dengan menyimpan dan mengambil data dapat membuat perbedaan besar. Terkadang daftar tertaut sulit untuk dimanipulasi. Jika item terakhir dihapus, yang terakhir tetapi satu harus mengubah penunjuknya untuk menyimpan referensi NULL. Ini mengharuskan daftar dilintasi

untuk menemukan tautan terakhir kecuali satu, dan penunjuknya disetel ke referensi NULL. Akhirnya, daftar tertaut membuang-buang memori dalam hal poin referensi tambahan.

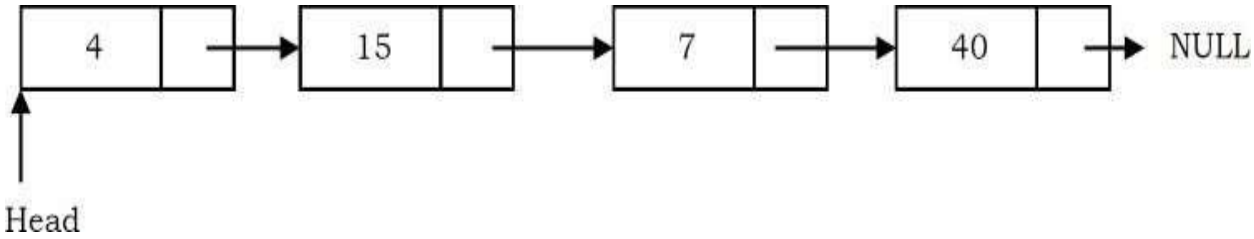
3.5 PERBANDINGAN DAFTAR TERTAUT DENGAN ARRAY & ARRAY DINAMIS

Tabel 3.1 daftar tertaut dengan array & array dinamis

Parameter	Daftar Tertaut	Himpunan
Pengindeksan	O(n)	O(1)
Penyisipan/penghapusan di awal	O(1)	O(n), jika array tidak penuh (untuk menggeser elemen)
Penyisipan di akhir	O(n)	O(1), jika array tidak penuh
Penghapusan di akhir	O(n)	O(1)
Sisipan di tengah	O(n)	O(n), jika array tidak penuh (untuk menggeser elemen)
Penghapusan di tengah	O(n)	O(n), jika array tidak penuh (untuk menggeser elemen)
Ruang yang terbuang	O(n) untuk pointer	0

3.6 DAFTAR TERTAUT TUNGGAL

Umumnya "daftar tertaut" berarti daftar tertaut tunggal. Daftar ini terdiri dari sejumlah node di mana setiap node memiliki pointer berikutnya ke elemen berikut. Tautan simpul terakhir dalam daftar adalah NULL, yang menunjukkan akhir daftar.



Gambar 3.3 daftar yang terdiri dari sejumlah node

Berikut ini adalah deklarasi tipe untuk daftar bilangan bulat yang ditautkan:

```
struct ListNode {
   int data;
   struct ListNode *next;
};
```

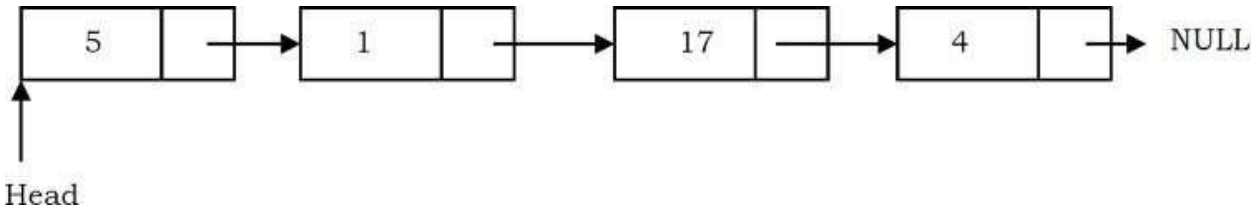
Operasi Dasar pada Daftar

- Melintasi daftar
- Memasukkan item ke dalam daftar
- Menghapus item dari daftar

Melintasi Daftar Tertaut

Mari kita asumsikan bahwa kepala menunjuk ke simpul pertama dari daftar. Untuk melintasi daftar, Kita melakukan hal berikut:

- Ikuti petunjuknya.
- Menampilkan isi dari node (atau count) saat dilintasi.
- Berhenti ketika penunjuk berikutnya menunjuk ke NULL.



Gambar 3.4 kepala menunjuk ke simpul pertama dari daftar

Fungsi ListLength() mengambil daftar tertaut sebagai input dan menghitung jumlah node dalam daftar. Fungsi yang diberikan di bawah ini dapat digunakan untuk mencetak data daftar dengan fungsi cetak tambahan.

```
int ListLength(struct ListNode *head) {
    struct ListNode *current = head;
    int count = 0;

    while (current != NULL) {
        count++;
        current = current→next;
    }
    return count;
}
```

Kompleksitas Waktu: O(n), untuk memindai daftar ukuran n.

Kompleksitas Ruang: O(1), untuk membuat variabel sementara.

Penyisipan Daftar Tertaut Tunggal

Penyisipan ke dalam daftar tertaut tunggal memiliki tiga kasus:

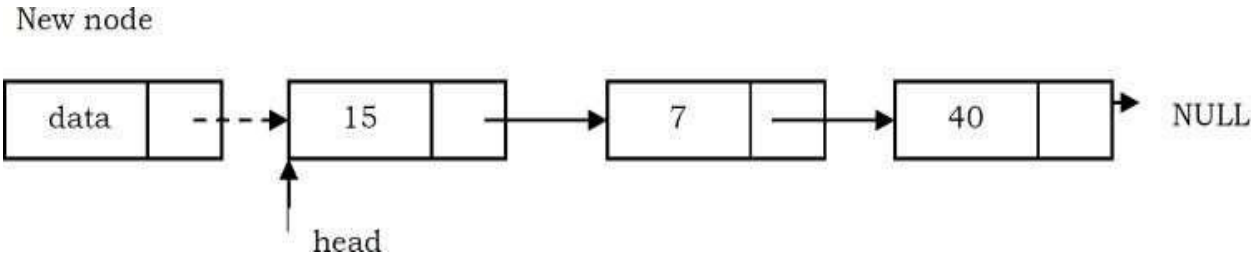
- Memasukkan node baru sebelum head (di awal)
- Memasukkan simpul baru setelah ekor (di akhir daftar)
- Memasukkan node baru di tengah daftar (lokasi acak)

Catatan: *Untuk menyisipkan elemen dalam daftar tertaut di beberapa posisi p, asumsikan bahwa setelah memasukkan elemen posisi simpul baru ini adalah p.*

Memasukkan Node dalam Daftar Tertaut Tunggal di Awal

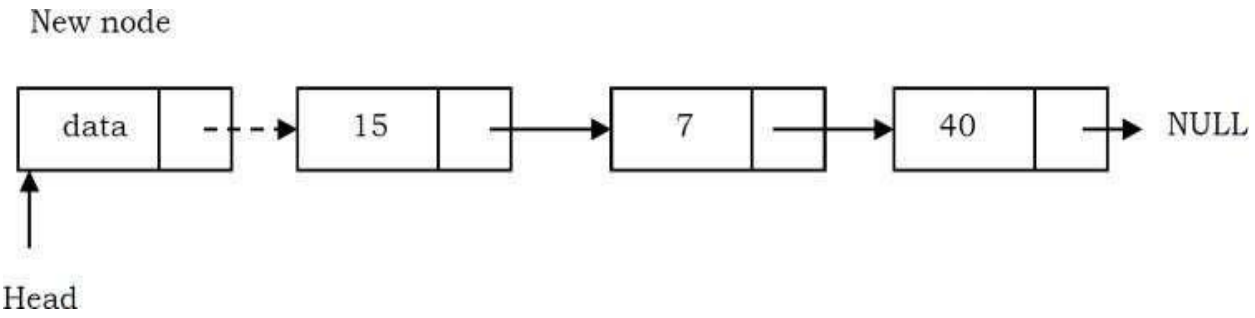
Dalam hal ini, simpul baru dimasukkan sebelum simpul kepala saat ini. Hanya satu pointer berikutnya yang perlu dimodifikasi (pointer berikutnya node baru) dan itu dapat dilakukan dalam dua langkah:

- Perbarui penunjuk berikutnya dari simpul baru, untuk menunjuk ke kepala saat ini.



Gambar 3.5 memperbarui penunjuk dari simpul baru

- Perbarui penunjuk kepala untuk menunjuk ke simpul baru.

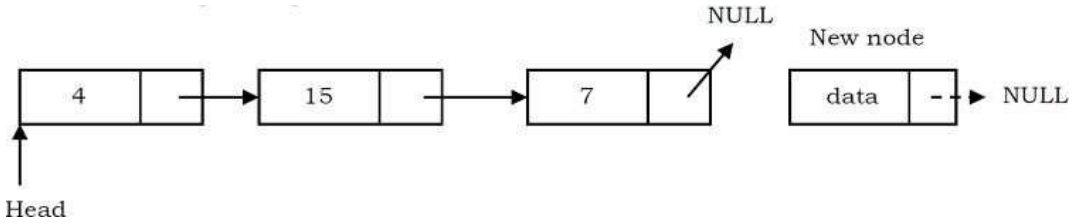


Gambar 3.6 kepala berada di simpul baru

Memasukkan Node dalam Daftar Tertaut Tunggal di Akhir

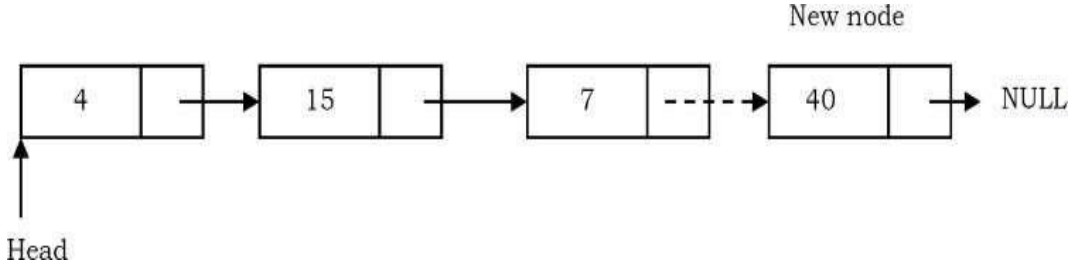
Dalam hal ini, kita perlu memodifikasi dua pointer berikutnya (node terakhir pointer berikutnya dan node baru pointer berikutnya).

- Node baru pointer berikutnya menunjuk ke NULL.



Gambar 3.7 modifikasi dua pointer

- Node terakhir pointer berikutnya menunjuk ke node baru.

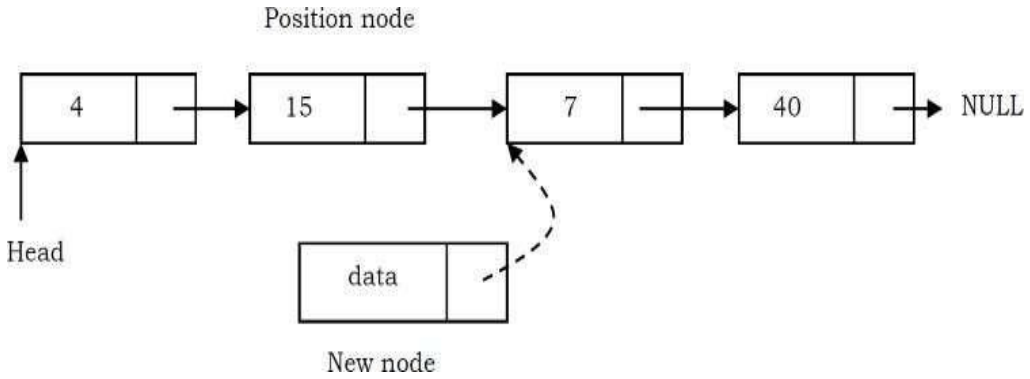


Gambar 3.8 node berikutnya menunjuk ke node baru

Memasukkan Node dalam Daftar Tertaut Tunggal di Tengah

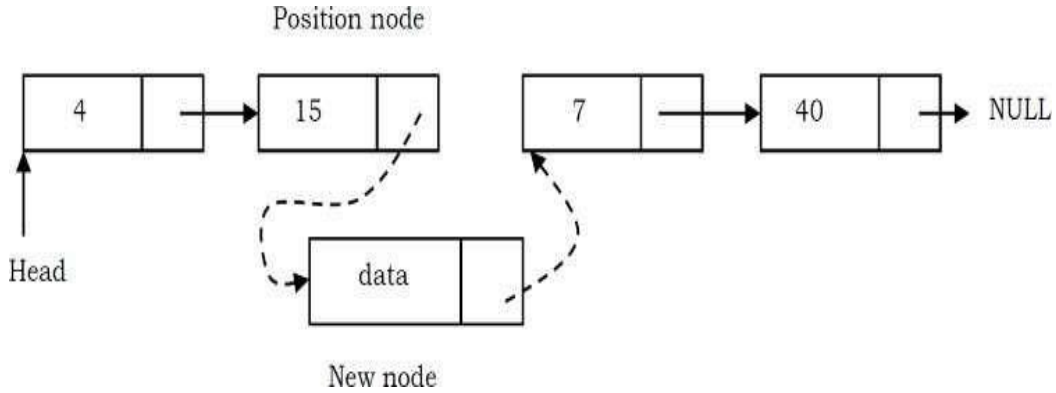
Mari kita asumsikan bahwa kita diberi posisi di mana kita ingin memasukkan node baru. Dalam hal ini juga, kita perlu memodifikasi dua pointer berikutnya.

- Jika kita ingin menambahkan elemen di posisi 3 maka kita berhenti di posisi 2. Itu berarti kita melintasi 2 node dan memasukkan node baru. Untuk mempermudah, mari kita asumsikan bahwa simpul kedua disebut simpul posisi. Node baru menunjuk ke node berikutnya dari posisi di mana kita ingin menambahkan node ini.



Gambar 3.9 memasukan node ditengah

- Posisi pointer node berikutnya sekarang menunjuk ke node baru.



Gambar 3.10 pointer menunjuk ke node berikutnya

Mari kita menulis kode untuk ketiga kasus. Kita harus memperbarui pointer elemen pertama dalam fungsi pemanggil, bukan hanya pada fungsi yang dipanggil. Untuk alasan ini kita perlu mengirim pointer ganda. Kode berikut menyisipkan simpul dalam daftar tertaut tunggal.

```
void InsertInLinkedList(struct ListNode **head,int data,int position) {
   int k=1;
   struct ListNode *p,*q,*newNode;
   newNode = (ListNode *)malloc(sizeof(struct ListNode));
   if(!newNode){
      printf("Memory Error");
      return;
   }
   newNode→data=data;
   p=*head;
   //Inserting at the beginning
   if(position == 1){
      newNode→next=p;
      *head=newNode;
   }
   else{
      //Traverse the list until the position where we want to insert
      while((p!=NULL) && (k<position)){
         k++;
         q=p;
         p=p→next;
      }
      q→next=newNode;   //more optimum way to do this
      newNode→next=p;
   }
}
```

Catatan: *Kita dapat menerapkan tiga variasi operasi penyisipan secara terpisah.*

Kompleksitas Waktu: O(n), karena, dalam kasus terburuk, kita mungkin perlu memasukkan simpul di akhir daftar.
Kompleksitas Ruang: O(1), untuk membuat satu variabel sementara.

Penghapusan Daftar Tertaut Tunggal

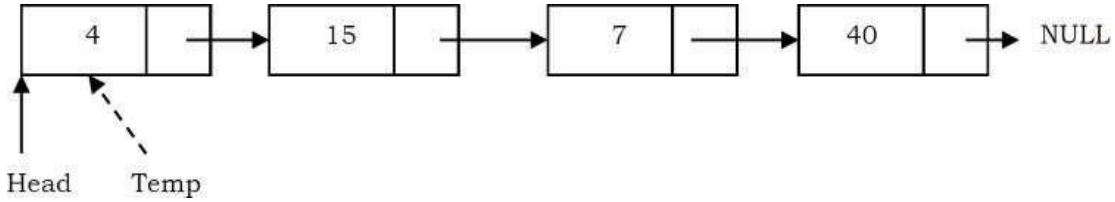
Mirip dengan penyisipan, di sini Kita juga memiliki tiga kasus.

- Menghapus simpul pertama
- Menghapus simpul terakhir
- Menghapus node perantara.

Menghapus Node Pertama di Single Linked List

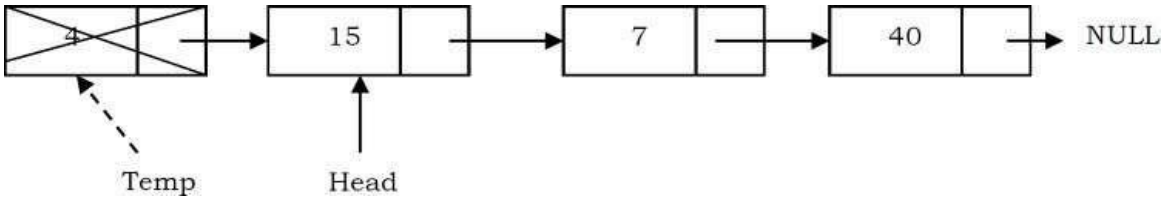
Node pertama (node kepala saat ini) dihapus dari daftar. Itu dapat dilakukan dalam dua langkah:

- Buat simpul sementara yang akan menunjuk ke simpul yang sama dengan simpul kepala.



Gambar 3.11 membuat simpul pertama

- Sekarang, pindahkan penunjuk simpul kepala ke simpul berikutnya dan buang simpul sementara.

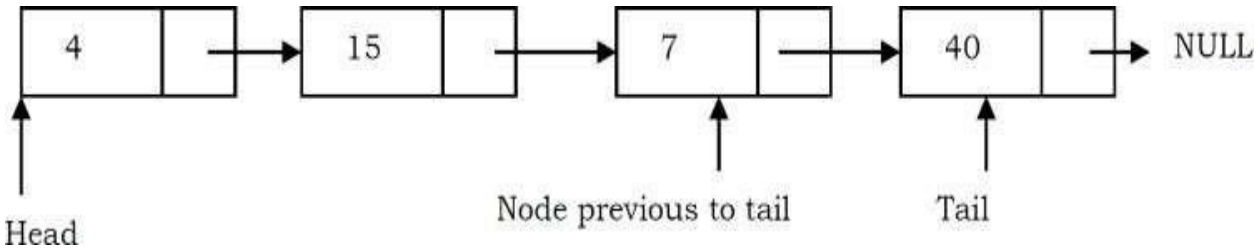


Gambar 3.12 memindahkan penunjuk simpul

Menghapus Node Terakhir di Single Linked List

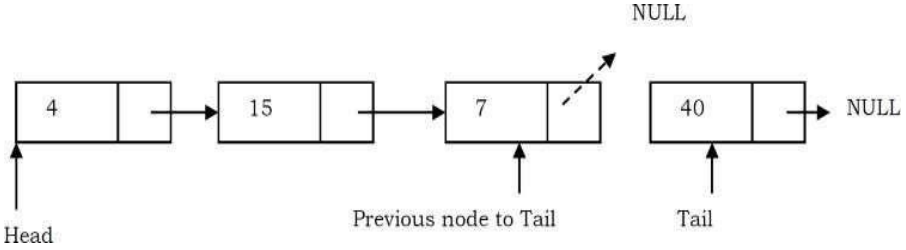
Dalam hal ini, simpul terakhir dihapus dari daftar. Operasi ini sedikit lebih sulit daripada menghapus node pertama, karena algoritma harus menemukan node, yang sebelumnya ke ekor. Itu dapat dilakukan dalam tiga langkah:

- Traverse daftar dan saat melintasi mempertahankan alamat node sebelumnya juga. Pada saat kita mencapai akhir daftar, kita akan memiliki dua pointer, satu menunjuk ke simpul ekor dan yang lainnya menunjuk ke simpul sebelum simpul ekor.



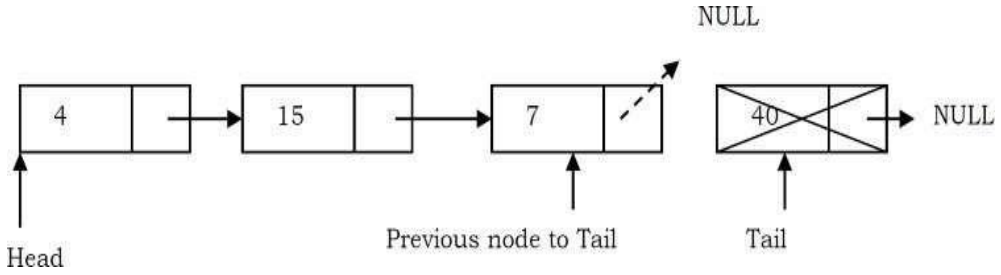
Gambar 3.13 pointer menunjuk ke simpul ekor dan sebelum simpul ekor

- Perbarui penunjuk berikutnya dari simpul sebelumnya dengan NULL.



Gambar 3.14 memperbarui simpul dengan *NULL*

- Buang simpul ekor.

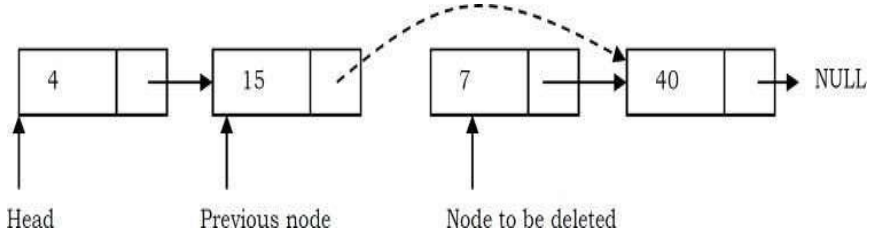


Gambar 3.15 membuang simpul ekor

Menghapus Node Menengah dalam Daftar Tertaut Tunggal

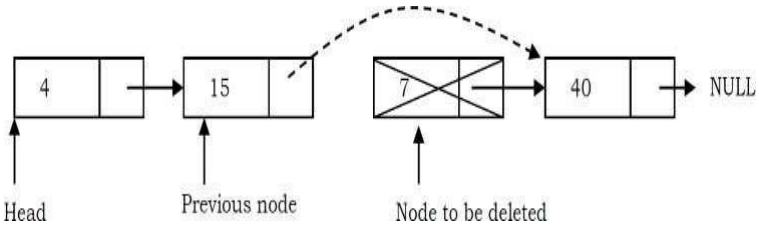
Dalam hal ini, node yang akan dihapus selalu terletak di antara dua node. Tautan kepala dan ekor tidak diperbarui dalam kasus ini. Penghapusan semacam itu dapat dilakukan dalam dua langkah:

- Mirip dengan kasus sebelumnya, pertahankan node sebelumnya saat melintasi daftar. Setelah Kita menemukan simpul yang akan dihapus, ubah penunjuk berikutnya dari simpul sebelumnya ke penunjuk berikutnya dari simpul yang akan dihapus.



Gambar 3.16 mempertahankan node ketika melewati daftar

- Buang node saat ini yang akan dihapus.



Gambar 3.17 menghilangkan node yang dihapus

Kompleksitas Waktu: O(n). Dalam kasus terburuk, kita mungkin perlu menghapus node di akhir daftar. Kompleksitas Ruang: O(1), untuk satu variabel sementara.

Menghapus Daftar Tertaut Tunggal

Ini bekerja dengan menyimpan simpul saat ini di beberapa variabel sementara dan membebaskan simpul saat ini. Setelah membebaskan node saat ini, pergi ke node berikutnya dengan variabel sementara dan ulangi proses ini untuk semua node.

```
void DeleteNodeFromLinkedList (struct ListNode **head, int position) {
    int k = 1;
    struct ListNode *p, *q;
    if(*head == NULL) {
        printf ("List Empty");
        return;
    }
    p = *head;
    /* from the beginning */
    if(position == 1) {
        *head = (*head)→next;
        free (p);
        return;
    }
    else {
        //Traverse the list until arriving at the position from which we want to delete
        while ((p != NULL) && (k < position )) {
            k++;
            q = p;
            p = p→next;
        }
        if(p == NULL)                    /* At the end */
            printf ("Position does not exist.");
        else {                           /* From the middle */
            q→next = p→next;
            free(p);
        }
    }
}
```

```
void DeleteLinkedList(struct ListNode **head) {
    struct ListNode *auxilaryNode, *iterator;
    iterator = *head;
    while (iterator) {
        auxilaryNode = iterator→next;
        free(iterator);
        iterator = auxilaryNode;
    }
    *head = NULL;                        // to affect the real head back in the caller.
}
```

Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n. Kompleksitas Ruang: O(1), untuk membuat satu variabel sementara.

3.7 DAFTAR TERTAUT GANDA

Keuntungan dari daftar tertaut ganda (juga disebut daftar tertaut dua arah) adalah bahwa dengan adanya simpul dalam daftar, kita dapat menavigasi ke dua arah. Sebuah node dalam daftar tertaut tunggal tidak dapat dihapus kecuali kita memiliki penunjuk ke pendahulunya. Tetapi dalam daftar tertaut ganda, kita dapat menghapus sebuah node bahkan jika kita tidak memiliki alamat node sebelumnya (karena setiap node memiliki pointer kiri yang menunjuk ke node sebelumnya dan dapat bergerak mundur).

Kerugian utama dari daftar tertaut ganda adalah:

- Setiap node membutuhkan pointer tambahan, membutuhkan lebih banyak ruang.

- Penyisipan atau penghapusan node membutuhkan waktu sedikit lebih lama (lebih banyak operasi pointer).

Mirip dengan daftar tertaut tunggal, mari kita terapkan operasi daftar tertaut ganda. Jika Anda memahami operasi daftar tertaut tunggal, maka operasi daftar tertaut ganda sudah jelas. Berikut ini adalah deklarasi tipe untuk daftar bilangan bulat yang ditautkan ganda:

```
struct DLLNode {
   int data;
   struct DLLNode *next;
   struct DLLNode *prev;
};
```

Penyisipan Daftar Tertaut Ganda

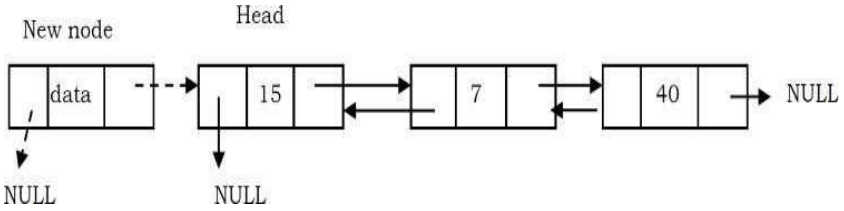
Penyisipan ke dalam daftar tertaut ganda memiliki tiga kasus (sama dengan daftar tertaut tunggal):

- Memasukkan node baru sebelum head.
- Memasukkan simpul baru setelah ekor (di akhir daftar).
- Memasukkan node baru di tengah daftar.

Memasukkan Node dalam Daftar Tertaut Ganda di Awal

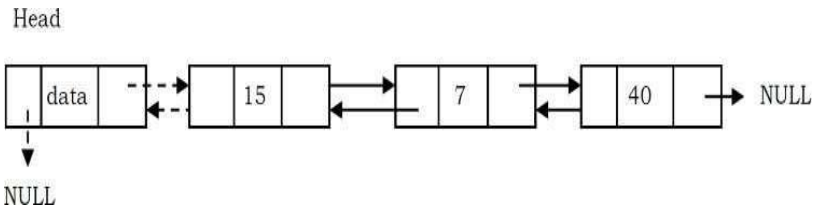
Dalam hal ini, simpul baru dimasukkan sebelum simpul kepala. Petunjuk sebelumnya dan selanjutnya perlu dimodifikasi dan dapat dilakukan dalam dua langkah:

- Perbarui penunjuk kanan dari simpul baru untuk menunjuk ke simpul kepala saat ini (tautan putus-putus pada gambar di bawah) dan juga buat penunjuk kiri simpul baru sebagai NULL.



Gambar 3.19 penunjuk diperbarui dari simpul baru ke simpul kepala

- Perbarui penunjuk kiri simpul kepala untuk menunjuk ke simpul baru dan jadikan simpul baru sebagai kepala.

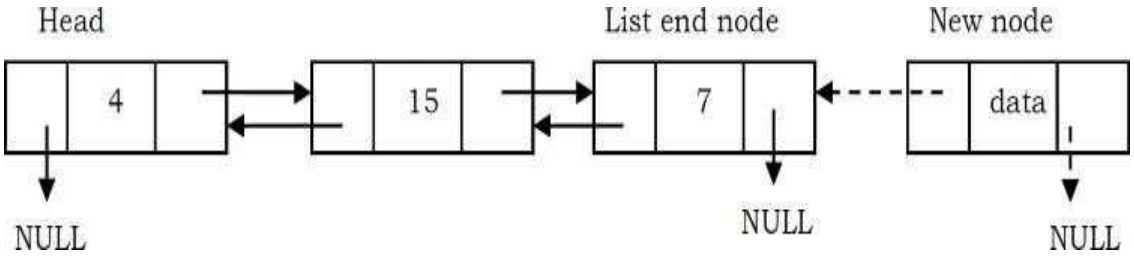


Gambar 3.20 menjadikan simpul baru sebagai kepala

Memasukkan Node dalam Daftar Tertaut Ganda di Akhir

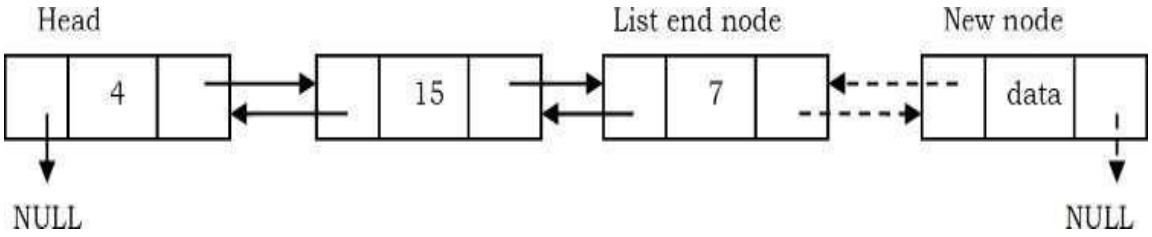
Dalam hal ini, telusuri daftar sampai akhir dan masukkan simpul baru.

- Penunjuk kanan simpul baru menunjuk ke NULL dan penunjuk kiri menunjuk ke akhir daftar.



Gambar 3.21 simpul baru menunjuk ke *NULL*

- Perbarui penunjuk kanan dari simpul terakhir untuk menunjuk ke simpul baru.

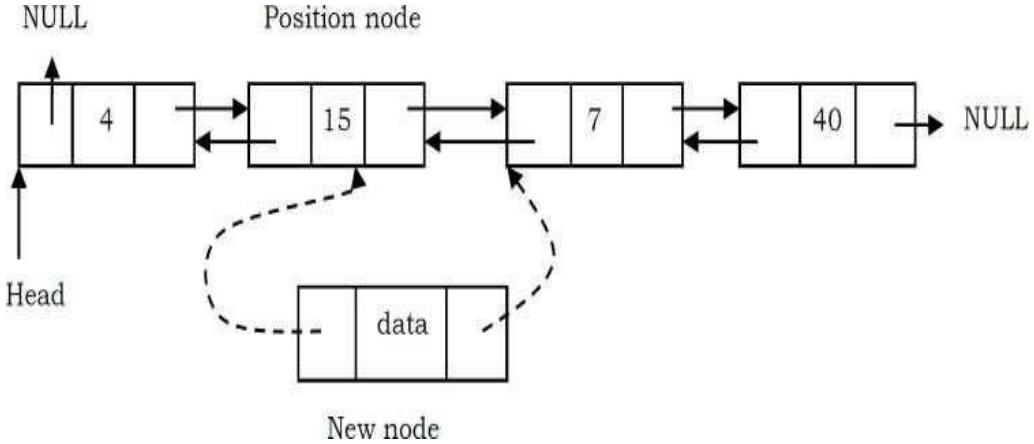


Gambar 3.22 simpul terakhir menunjuk ke simpul baru

Memasukkan Node di Double Linked List di Tengah

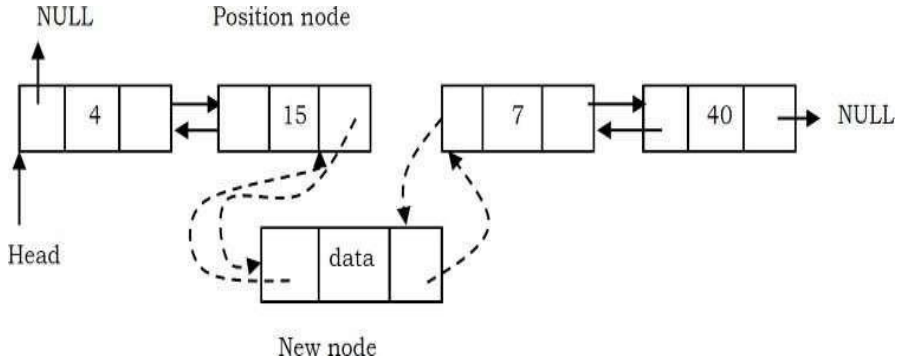
Seperti yang dibahas dalam daftar tertaut tunggal, telusuri daftar ke node posisi dan masukkan node baru.

- Penunjuk kanan simpul baru menunjuk ke simpul berikutnya dari simpul posisi di mana kita ingin menyisipkan simpul baru. Juga, penunjuk kiri simpul baru menunjuk ke simpul posisi.



Gambar 3.23 simpul baru menunjuk ke simpul posisi

- Posisi penunjuk kanan simpul menunjuk ke simpul baru dan simpul berikutnya dari simpul posisi penunjuk kiri menunjuk ke simpul baru.



Gambar 3.24 menunjuk ke simpul baru

Sekarang, mari kita tulis kode untuk ketiga kasus ini. Kita harus memperbarui pointer elemen pertama dalam fungsi pemanggil, bukan hanya pada fungsi yang dipanggil. Untuk

```
void DLLInsert(struct DLLNode **head, int data, int position) {
    int k = 1;
    struct DLLNode *temp, *newNode;
    newNode = (struct DLLNode *) malloc(sizeof ( struct DLLNode ));
    if(!newNode) {                          //Always check for memory errors
       printf ("Memory Error");
       return;
    }
    newNode→data = data;
    if(position == 1) {                     //Inserting a node at the beginning
        newNode→next = *head;
        newNode→prev = NULL;

        if(*head)
          (*head)→prev = newNode;

        *head = newNode;
        return;
    }
    temp = *head;
    while ( (k < position - 1) && temp→next!=NULL) {
        temp = temp→next;
        k++;
    }
    if(k!=position){
       printf("Desired position does not exist\n");
    }
    newNode→next=temp→next;
    newNode→prev=temp;
    if(temp→next)
       temp→next→prev=newNode;

    temp→next=newNode;
    return;
}
```

alasan ini kita perlu mengirim pointer ganda. Berikut kode menyisipkan simpul dalam daftar tertaut ganda.

Kompleksitas Waktu: O(n). Dalam kasus terburuk, kita mungkin perlu memasukkan simpul di akhir daftar. Kompleksitas Ruang: O(1), untuk membuat satu variabel sementara.

Penghapusan Daftar Tertaut Ganda

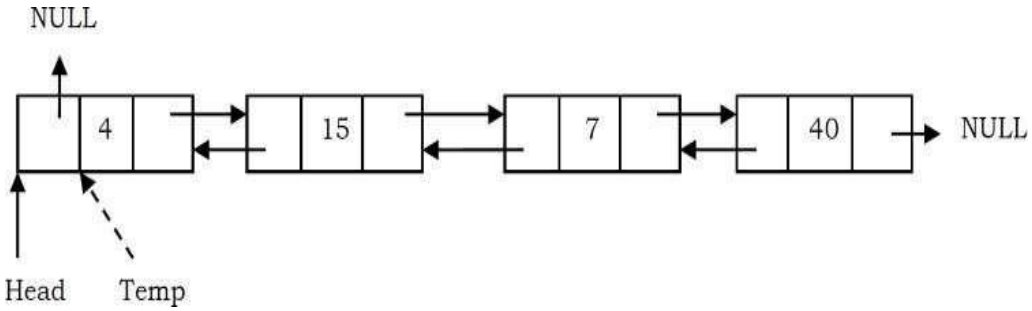
Mirip dengan penghapusan daftar tertaut tunggal, di sini Kita memiliki tiga kasus:

- Menghapus simpul pertama
- Menghapus simpul terakhir
- Menghapus simpul perantara

Menghapus Node Pertama di Double Linked List

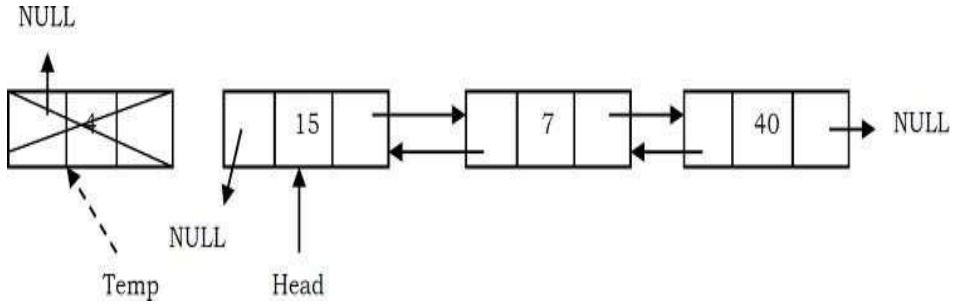
Dalam hal ini, simpul pertama (simpul kepala saat ini) dihapus dari daftar. Dapat dilakukan dalam dua langkah:

- Buat simpul sementara yang akan menunjuk ke simpul yang sama dengan simpul kepala.



Gambar 3.25 membuat simpul sementara

- Sekarang, pindahkan penunjuk simpul kepala ke simpul berikutnya dan ubah penunjuk kiri kepala ke NULL. Kemudian, buang simpul sementara.

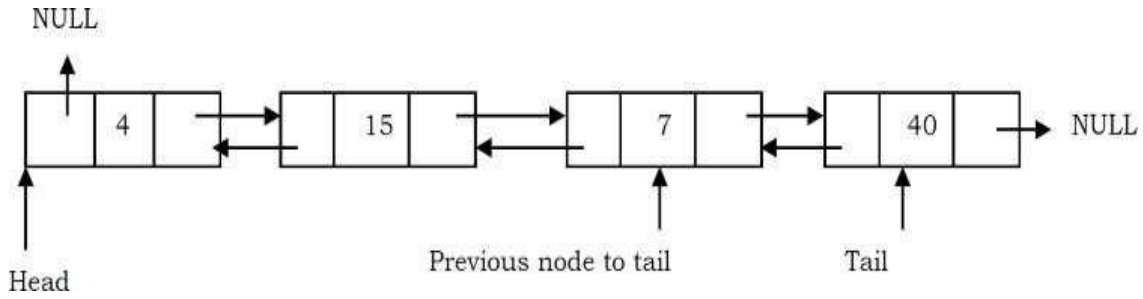


Gambar 3.26 memindahkan simpul kepala dan penunjuk kiri kepala ke *NULL*

Menghapus Node Terakhir di Double Linked List

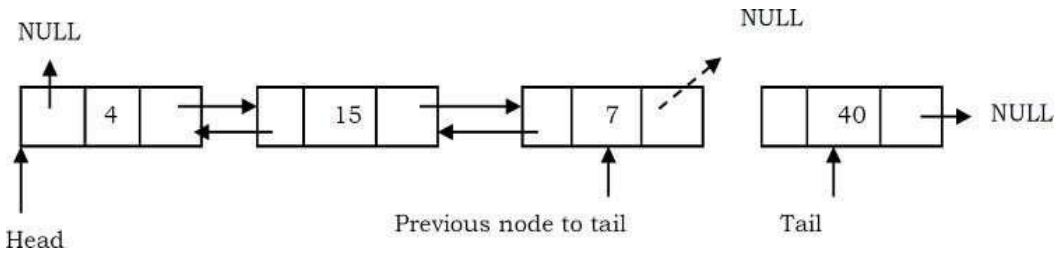
Operasi ini sedikit lebih sulit daripada menghapus node pertama, karena algoritma harus menemukan node, yang sebelumnya ke ekor terlebih dahulu. Ini dapat dilakukan dalam tiga langkah:

- Traverse daftar dan saat melintasi mempertahankan alamat node sebelumnya juga. Pada saat kita mencapai akhir daftar, kita akan memiliki dua pointer, satu menunjuk ke ekor dan yang lainnya menunjuk ke simpul sebelum ekor.

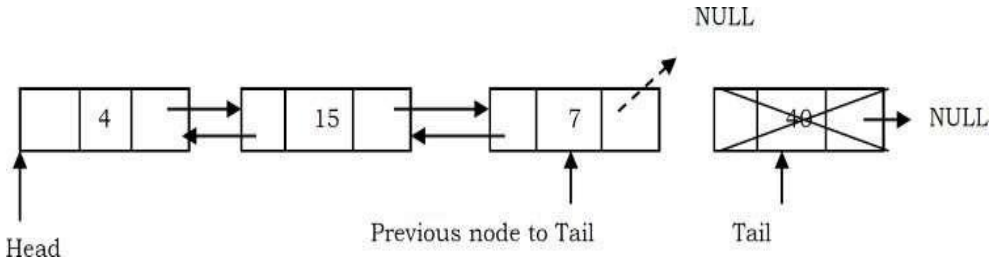


Gambar 3.27 dua pointer menunjuk ke ekor lainnya

- Perbarui penunjuk berikutnya dari simpul sebelumnya ke simpul ekor dengan NULL.



Gambar 3.28 memperbarui simpul ekor *NULL*



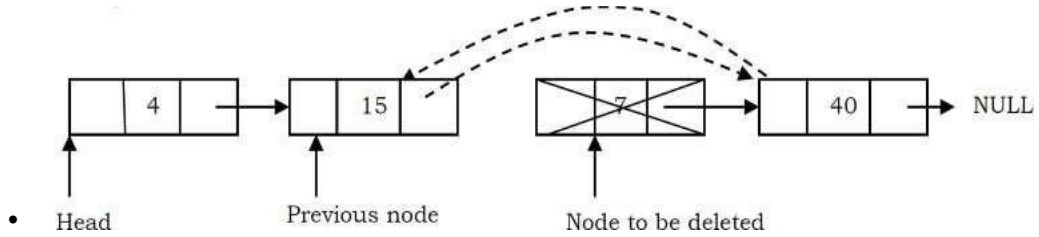
- Buang simpul ekor.

Gambar 3.29 membuang simpul ekor

Menghapus Node Menengah dalam Daftar Tertaut Ganda

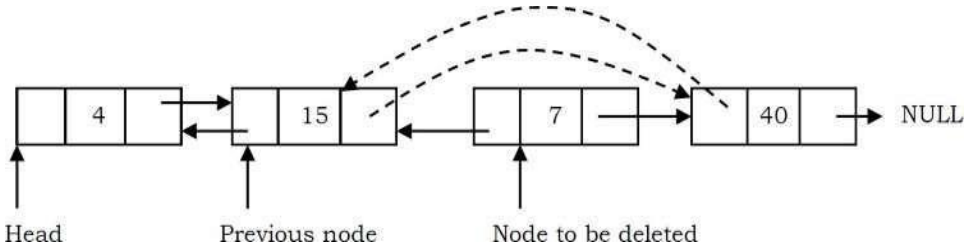
Dalam hal ini, simpul yang akan dihapus selalu terletak di antara dua simpul, dan tautan kepala dan ekor tidak diperbarui. Penghapusan dapat dilakukan dalam dua langkah:

- Mirip dengan kasus sebelumnya, pertahankan node sebelumnya sambil juga melintasi daftar. Setelah menemukan node yang akan dihapus, ubah pointer berikutnya dari node sebelumnya ke node berikutnya dari node yang akan dihapus.



Gambar 3.30 mengubah pointer

- Buang node saat ini yang akan dihapus.



Gambar 3.31 buang node yang dihapus

Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n.
Kompleksitas Ruang: O(1), untuk membuat satu variabel sementara.

```
void DLLDelete(struct DLLNode **head, int position) {
    struct DLLNode *temp, *temp2, temp = *head;
    int k = 1;
    if(*head ==  NULL) {
       printf("List is empty");
       return;
    }
    if(position == 1) {
       *head = (*head)→next;

    if(*head != NULL)
        (*head)→prev = NULL;
       free(temp);
       return;
    }
    while((k < position) && temp→next!=NULL) {
       temp = temp→next;
       k++;
    }
    if(k!=position-1){
       printf("Desired position does not exist\n");
    }

    temp2=temp→prev;
    temp2→next=temp→next;

    if(temp→next) // Deletion from Intermediate Node
       temp→next→prev=temp2;
    free(temp);
    return;
}
```

3.8 DAFTAR TERTAUT MELINGKAR

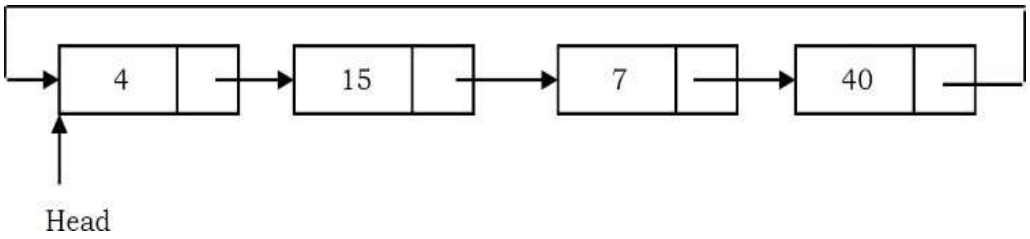
Dalam daftar tertaut tunggal dan daftar tertaut ganda, akhir daftar ditunjukkan dengan nilai NULL. Tetapi daftar tertaut melingkar tidak memiliki akhir. Saat melintasi daftar tertaut melingkar, kita harus berhati-hati; jika tidak, kita akan melintasi daftar tanpa batas. Dalam daftar tertaut melingkar, setiap simpul memiliki penerus. Perhatikan bahwa tidak seperti daftar tertaut tunggal, tidak ada simpul dengan penunjuk NULL dalam daftar tertaut sirkular. Dalam beberapa situasi, daftar tertaut melingkar berguna.

Misalnya, ketika beberapa proses menggunakan sumber daya komputer (CPU) yang sama untuk jumlah waktu yang sama, kita harus memastikan bahwa tidak ada proses yang mengakses sumber daya sebelum semua proses lain melakukannya (algoritma round robin). Berikut ini adalah deklarasi tipe untuk daftar bilangan bulat tertaut melingkar:

```
typedef struct CLLNode {
   int data;
   struct ListNode *next;
};
```

Dalam daftar tertaut melingkar, kita mengakses elemen menggunakan simpul kepala (mirip dengan simpul kepala dalam daftar tertaut tunggal dan daftar tertaut ganda).

Menghitung Node dalam Daftar Tertaut Melingkar



Gambar 3.32 node daftar tertaut melingkar

Daftar melingkar dapat diakses melalui node yang ditandai kepala. Untuk menghitung node, daftar harus dilalui dari node yang ditandai head, dengan bantuan arus node dummy, dan menghentikan penghitungan saat arus mencapai node awal.

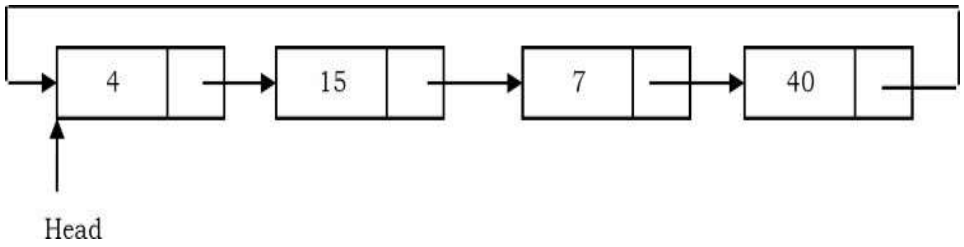
Jika daftar kosong, head akan menjadi NULL, dan dalam hal ini set count = 0. Jika tidak, setel pointer saat ini ke node pertama, dan terus menghitung sampai pointer saat ini mencapai node awal.

```
int CircularListLength(struct CLLNode *head) {
    struct CLLNode *current = head;
    int count = 0;
    if(head == NULL)
        return 0;

    do {
        current = current→next;
        count++;
    } while (current != head);

    return count;
}
```

Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n.
Kompleksitas Ruang: O(1), untuk membuat satu variabel sementara.

Mencetak Isi dari Daftar Tertaut Edaran

Kita berasumsi disini bahwa daftar sedang diakses oleh simpul kepalanya. Karena semua simpul disusun dalam bentuk melingkar, simpul ekor dari daftar akan menjadi simpul sebelum simpul kepala. Mari kita asumsikan kita ingin mencetak isi node yang dimulai dengan node kepala. Cetak isinya, pindah ke simpul berikutnya dan lanjutkan pencetakan sampai kita mencapai simpul kepala lagi.



Gambar 3.33 cetak daftar tertaut edaran

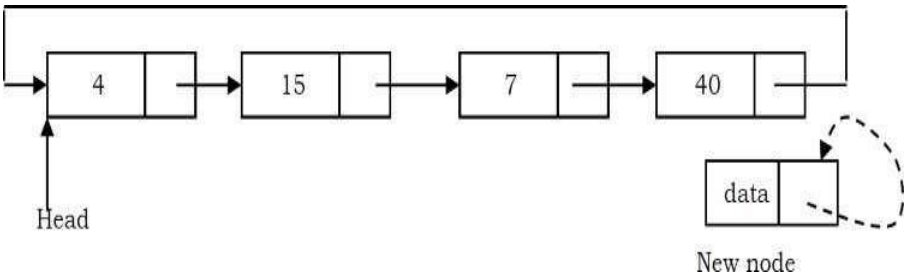
```
void PrintCircularListData(struct CLLNode *head) {
    struct CLLNode *current = head;
    if(head == NULL)
        return;

    do {
        printf ("%d", current→data);
        current = current→next;
    } while (current != head);
}
```

Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n.
Kompleksitas Ruang: O(1), untuk variabel sementara.

Memasukkan Node di Akhir Daftar Tertaut Melingkar

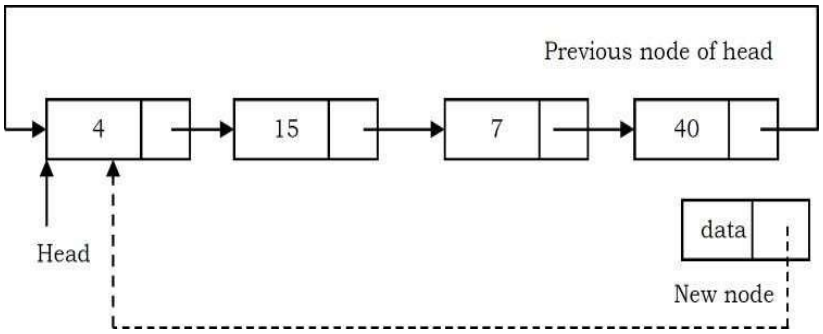
Mari kita tambahkan simpul yang berisi data, di akhir daftar (daftar melingkar) yang dipimpin oleh kepala. Node baru akan ditempatkan tepat setelah node ekor (yang merupakan node terakhir dari daftar), yang berarti harus disisipkan di antara node ekor dan node pertama.

- Buat simpul baru dan mula-mula pertahankan penunjuk berikutnya tetap menunjuk ke dirinya sendiri.



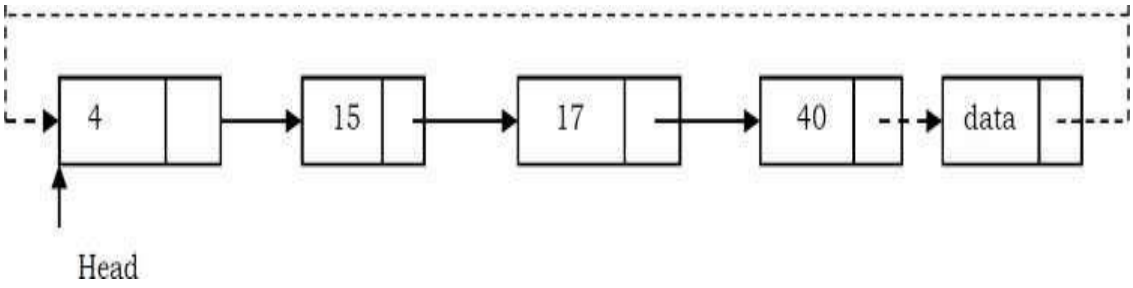
Gambar 3.34 membuat simpul baru

- Perbarui penunjuk berikutnya dari simpul baru dengan simpul kepala dan juga telusuri daftar ke ekor. Itu berarti dalam daftar melingkar kita harus berhenti di simpul yang simpul berikutnya adalah kepala.



Gambar 3.35 memperbarui penunjuk berikutnya

- Perbarui pointer berikutnya dari node sebelumnya untuk menunjuk ke node baru dan kita mendapatkan daftar seperti yang ditunjukkan di bawah ini.



Gambar 3.36 memperbarui pointer dari node sebelumnya

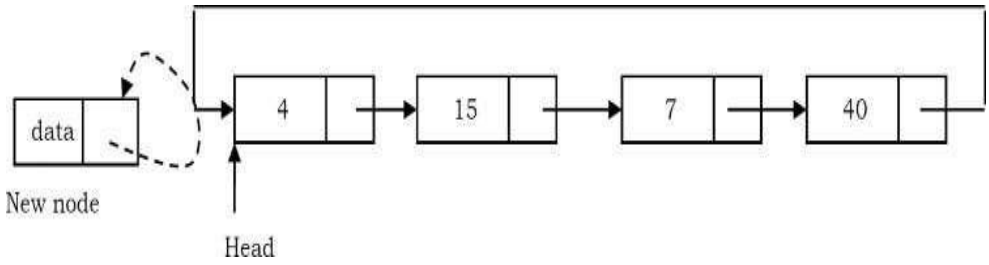
```
void InsertAtEndInCLL (struct CLLNode **head, int data) {
    struct CLLNode *current = *head;
    struct CLLNode *newNode = (struct CLLNode *) (malloc(sizeof(struct CLLNode)));
    if(!newNode) {
        printf("Memory Error");
        return;
    }
    newNode→data = data;
    while (current→next != *head)
        current = current→next;

    newNode→next = newNode;

    if(*head ==NULL)
        *head = newNode;
    else {
        newNode→next = *head;
        current→next = newNode;
    }
}
```

Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n.
Kompleksitas Ruang: O(1), untuk variabel sementara.

Memasukkan Node di Depan Daftar Tertaut Melingkar

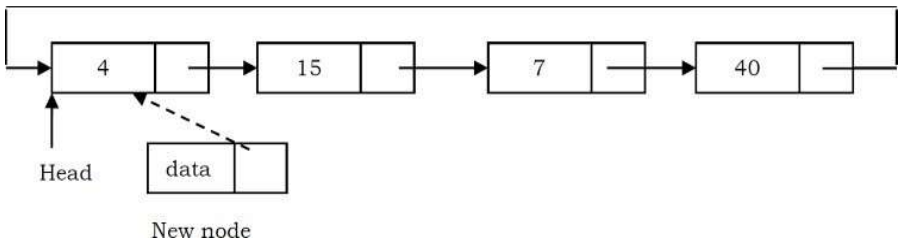
Satu-satunya perbedaan antara menyisipkan simpul di awal dan di akhir adalah, setelah menyisipkan simpul baru, kita hanya perlu memperbarui penunjuk. Langkah-langkah untuk melakukannya sebagai berikut:

- Buat simpul baru dan mula-mula pertahankan penunjuk berikutnya tetap menunjuk ke dirinya sendiri.



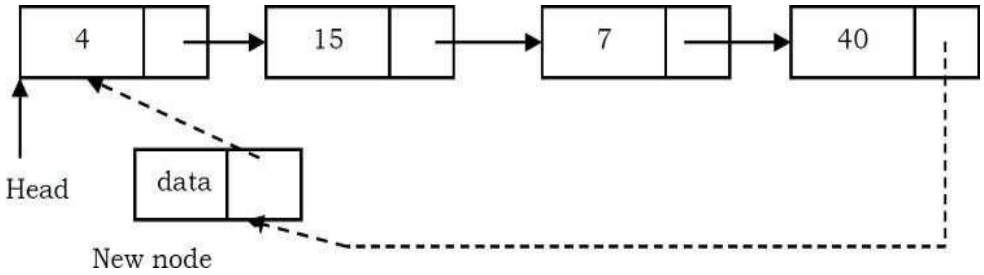
Gambar 3.37 simpul baru

- Perbarui penunjuk berikutnya dari simpul baru dengan simpul kepala dan juga telusuri daftar hingga ekor. Itu berarti dalam daftar melingkar kita harus berhenti di simpul yang merupakan simpul sebelumnya dalam daftar.



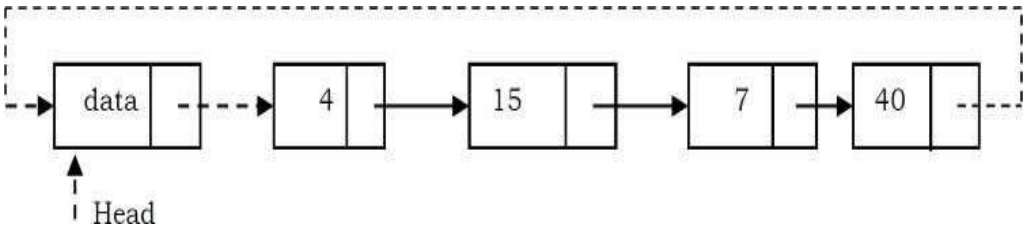
Gambar 3.38 memperbarui dan menelusuri daftar hingga ekor

- Perbarui node kepala sebelumnya dalam daftar untuk menunjuk ke node baru.



Gambar 3.39 memperbarui untuk menunjuk node baru

- Jadikan simpul baru sebagai kepala.



Gambar 3.40 simpul sebagai kepala

```
void InsertAtBeginInCLL (struct CLLNode **head, int data) {
    struct CLLNode *current = *head;
    struct CLLNode * newNode = (struct CLLNode *) (malloc(sizeof(struct CLLNode)));
    if(!newNode) {
        printf("Memory Error");
        return;
    }
    newNode→data = data;
    while (current→next != *head)
        current = current→next;
    newNode→next = newNode;
    if(*head ==NULL)
        *head = newNode;
    else {
        newNode→next = *head;
        current→next = newNode;
        *head = newNode;
    }
    return;
}
```

Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n.

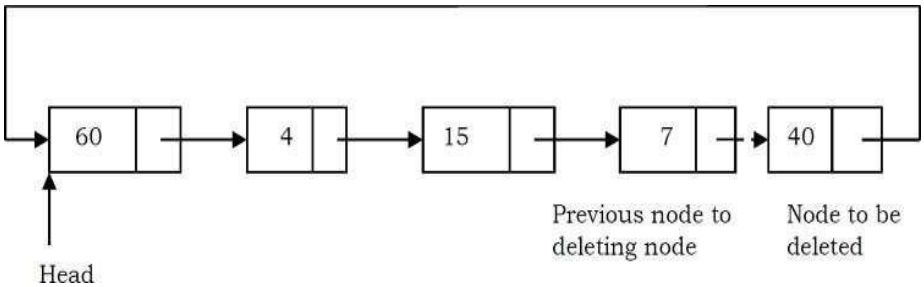
Kompleksitas Ruang: O(1), untuk variabel sementara.

Menghapus Node Terakhir di Circular Linked List

Daftar harus dilalui untuk mencapai simpul terakhir kecuali satu. Ini harus dinamai sebagai simpul ekor, dan bidang berikutnya harus menunjuk ke simpul pertama. Pertimbangkan daftar berikut.

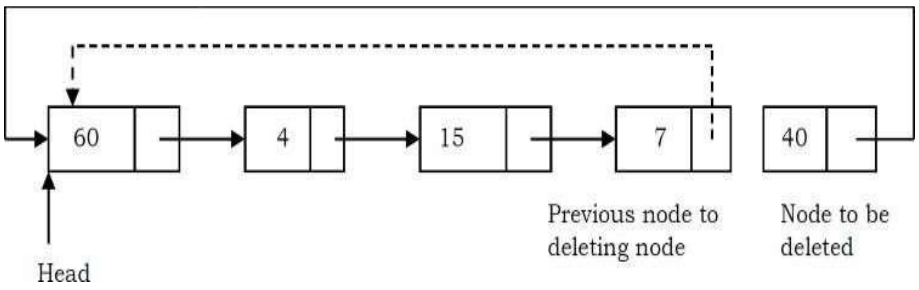
Untuk menghapus simpul terakhir 40, daftar harus dilalui sampai Anda mencapai 7. Bidang berikutnya dari 7 harus diubah menjadi titik 60, dan node ini harus diganti namanya menjadi pTail.

- Telusuri daftar dan temukan simpul ekor dan simpul sebelumnya.



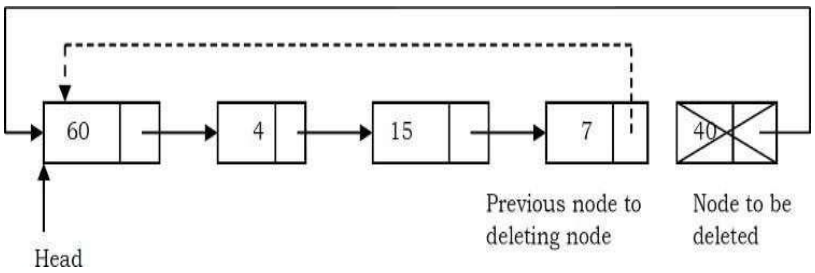
Gambar 3.41 telusuri daftar untuk menemukan simpul ekor

- Perbarui penunjuk berikutnya dari simpul sebelumnya dari simpul ekor untuk menunjuk ke kepala.



Gambar 3.42 perbarui petunjuk untuk menunjuk ke kepala

- Buang simpul ekor.



Gambar 3.43 membuang simpul ekor

```
void DeleteLastNodeFromCLL (struct CLLNode **head) {
    struct CLLNode *temp = *head, *current = *head;
    if(*head == NULL) {
        printf( "List Empty");  return;
    }
    while (current→next != *head) {
        temp = current;
        current = current→next;
    }
    temp→next = current→next;
    free(current);
    return;
}
```

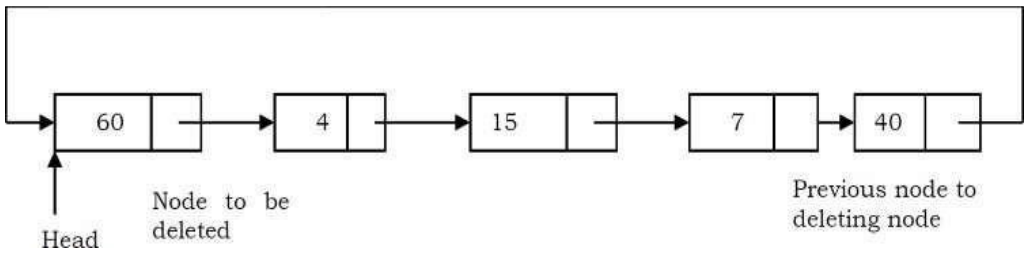
Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n.

Kompleksitas Ruang: O(1), untuk variabel sementara.

Menghapus Node Pertama dalam Daftar Melingkar

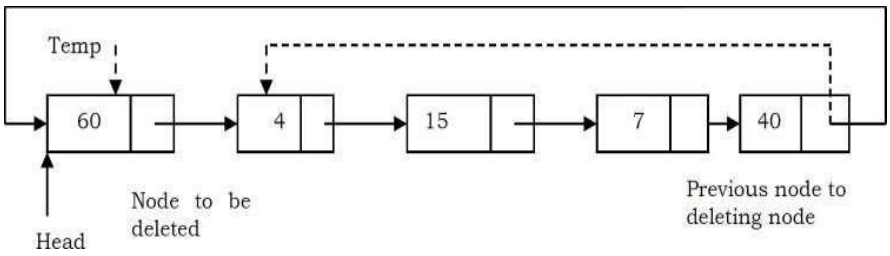
Node pertama dapat dihapus hanya dengan mengganti bidang berikutnya dari simpul ekor dengan bidang berikutnya dari simpul pertama.

- Temukan simpul ekor dari daftar tertaut dengan melintasi daftar. Tail node adalah node sebelumnya ke head node yang ingin kita hapus.



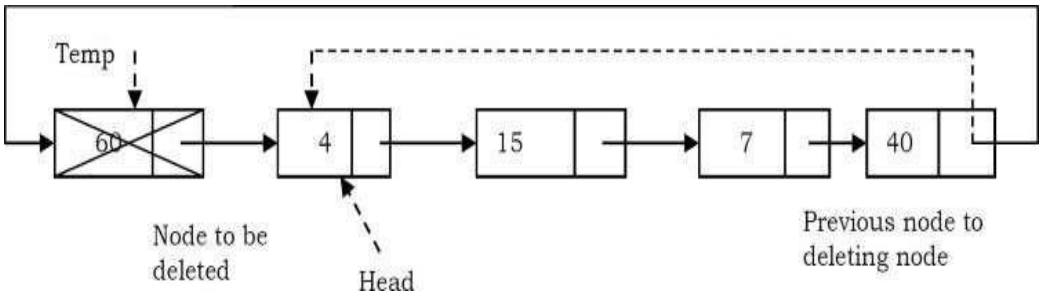
Gambar 3.44 menemukan simpul ekor

- Buat simpul sementara yang akan menunjuk ke kepala. Juga, perbarui simpul ekor penunjuk berikutnya untuk menunjuk ke simpul kepala berikutnya (seperti yang ditunjukkan di bawah).



Gambar 3.45 membuat simpul sementara

- Sekarang, pindahkan penunjuk kepala ke simpul berikutnya. Buat simpul sementara yang akan menunjuk ke kepala. Juga, perbarui simpul ekor penunjuk berikutnya untuk menunjuk ke simpul kepala berikutnya (seperti yang ditunjukkan di bawah).



Gambar 3.46 memindahkan penunjuk kepala ke simpul berikutnya

```
void DeleteFrontNodeFromCLL (struct CLLNode **head) {
    struct CLLNode *temp = *head;
    struct CLLNode *current = *head;

    if(*head == NULL) {
       printf("List Empty");
       return;
    }
    while (current→next != *head)
       current = current→next;

    current→ next = *head→next;
    *head = *head→next;

    free(temp);
    return;
}
```

Kompleksitas Waktu: O(n), untuk memindai daftar lengkap ukuran n.

Kompleksitas Ruang: O(1), untuk variabel sementara.

Aplikasi Daftar Edaran

Daftar tertaut melingkar digunakan dalam mengelola sumber daya komputasi komputer. Kita dapat menggunakan daftar melingkar untuk mengimplementasikan tumpukan dan antrian.

3.9 DAFTAR TERTAUT GANDA YANG HEMAT MEMORI

Dalam implementasi konvensional, kita perlu menyimpan penunjuk maju ke item berikutnya dalam daftar dan penunjuk mundur ke item sebelumnya. Itu berarti elemen dalam implementasi daftar tertaut ganda terdiri dari data, penunjuk ke simpul berikutnya dan penunjuk ke simpul sebelumnya dalam daftar seperti yang ditunjukkan di bawah ini.

Definisi Node Konvensional

```
typedef struct ListNode {
   int data;
   struct ListNode * prev;
   struct ListNode * next;
};
```

Baru-baru ini sebuah jurnal (Sinha) menyajikan implementasi alternatif dari ADT daftar tertaut ganda, dengan operasi penyisipan, traversal, dan penghapusan. Implementasi ini didasarkan pada perbedaan pointer. Setiap node hanya menggunakan satu bidang penunjuk untuk melintasi daftar bolak-balik.

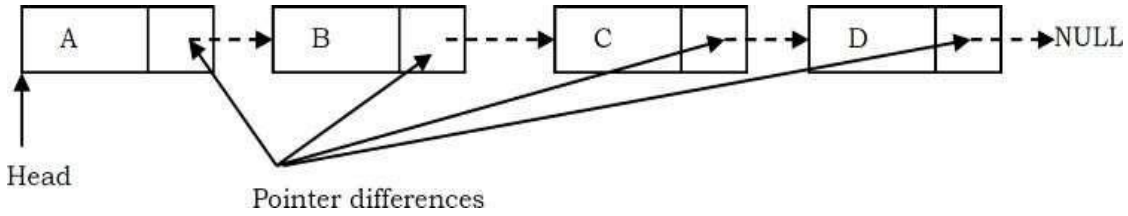
Definisi Node Baru

```
typedef struct ListNode {
  int data;
  struct ListNode * ptrdiff;
};
```

Bidang penunjuk ptrdiff berisi perbedaan antara penunjuk ke simpul berikutnya dan penunjuk ke simpul sebelumnya. Perbedaan pointer dihitung dengan menggunakan operasi eksklusif-atau (⊕).

ptrdiff = penunjuk ke simpul (⊕) *sebelumnya penunjuk ke simpul berikutnya.*

ptrdiff dari node awal (head node) adalah dari NULL dan node berikutnya (next node to head). Demikian pula, ptrdiff node akhir adalah dari node sebelumnya (sebelum node akhir) dan NULL. Sebagai contoh, perhatikan daftar tertaut berikut.



Gambar 3.47 contoh daftar tertaut

Pada contoh di atas,

- Pointer A berikutnya adalah: NULL ⊕ B
- Pointer B berikutnya adalah: A ⊕ C
- Pointer C berikutnya adalah: B ⊕ D
- Pointer D berikutnya adalah: C ⊕ NULL

Mengapa itu berhasil?

Untuk menemukan jawaban atas pertanyaan ini mari kita perhatikan sifat-sifat dari :

X ⊕ X=0

X ⊕ 0 = X

X ⊕ Y = Y ⊕ X (simetris)

(X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z) (transitif)

Untuk contoh di atas, mari kita asumsikan bahwa kita berada di simpul C dan ingin pindah ke B. Kita tahu bahwa ptrdiff C didefinisikan sebagai B ⊕ D. Jika kita ingin pindah ke B, melakukan ⊕ pada ptrdiff C dengan D akan memberikan B. Hal ini disebabkan oleh fakta bahwa

(B ⊕ D) ⊕ D = B(karena, D ⊕ D= 0)

Demikian pula, jika kita ingin pindah ke D, maka kita harus menerapkan ⊕ ke ptrdiff C dengan B untuk memberikan D.

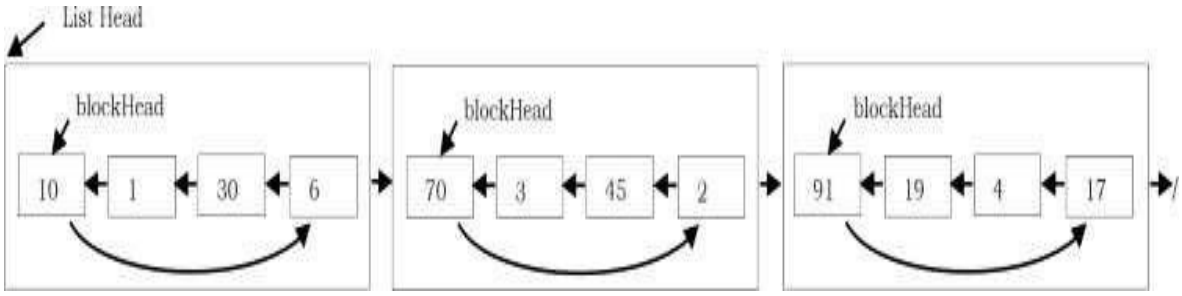
(B ⊕ D) ⊕ B = D (karena, B ⊕ B=0)

Dari pembahasan di atas kita dapat melihat bahwa hanya dengan menggunakan satu pointer, kita dapat bergerak maju mundur. Implementasi yang hemat memori dari daftar tertaut ganda dimungkinkan dengan sedikit mengorbankan efisiensi waktu.

3.10 DAFTAR TAUTAN YANG TIDAK DIGULUNG

Salah satu keuntungan terbesar dari daftar tertaut dibandingkan array adalah memasukkan elemen di lokasi mana pun hanya membutuhkan waktu O(1). Namun, dibutuhkan O(n) untuk mencari elemen dalam daftar tertaut. Ada variasi sederhana dari daftar tertaut tunggal yang disebut daftar tertaut yang tidak digulung.

Daftar tertaut yang tidak digulung menyimpan banyak elemen di setiap simpul (sebut saja blok untuk kenyamanan kita). Di setiap blok, daftar tertaut melingkar digunakan untuk menghubungkan semua node.



Gambar 3.48 daftar tertaut tidak digulung

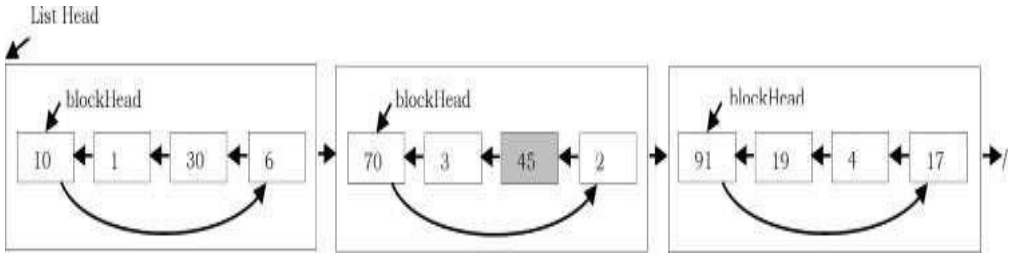
Asumsikan bahwa tidak akan ada lebih dari n elemen dalam daftar tertaut yang belum dibuka setiap saat. Untuk menyederhanakan masalah ini, semua blok, kecuali yang terakhir, harus mengandung elemen yang tepat. Dengan demikian, tidak akan ada lebih dari $\lceil \sqrt{n} \rceil$ blok setiap saat.

Mencari elemen di Daftar Tertaut yang Dibuka

Dalam daftar tertaut yang tidak digulung, kita dapat menemukan elemen ke-k di O($\lceil \sqrt{n} \rceil$):

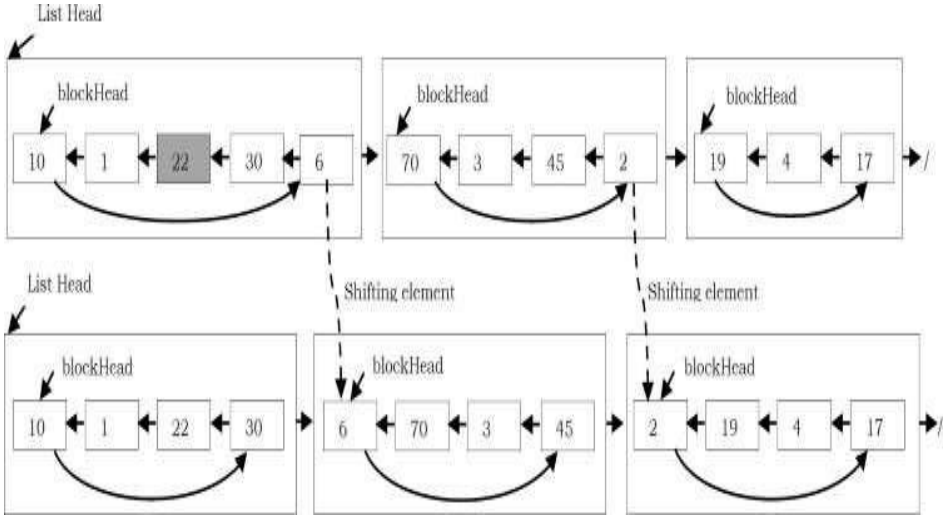
1. Traverse daftar blok ke salah satu yang berisi simpul ke-k, yaitu, $\left\lceil \frac{k}{\lceil \sqrt{n} \rceil} \right\rceil$th memblokir. Dibutuhkan O($\sqrt{n}$) karena kita dapat menemukannya dengan melewati tidak lebih $\sqrt{n}$ dari blok.

2. Temukan simpul (k mod $\lceil\sqrt{n}\rceil$) ke dalam daftar tertaut melingkar dari blok ini. Ini juga membutuhkan O($\sqrt{n}$) karena tidak ada lebih dari $\lceil\sqrt{n}\rceil$ node dalam satu blok.



Gambar 3.49 mencari elemen daftar tertaut dibuka

Memasukkan elemen dalam Daftar Tertaut yang Tidak Digulung



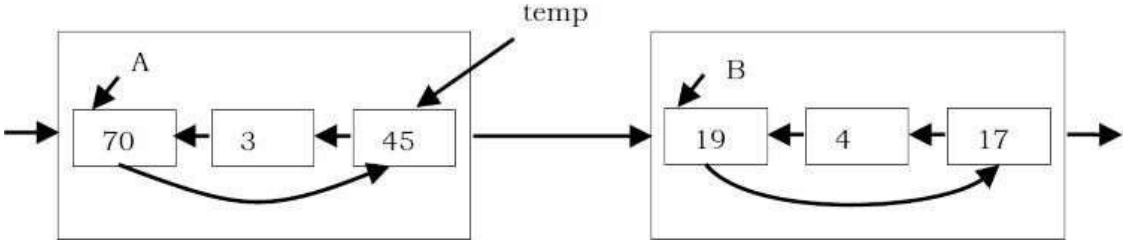
Gambar 3.50 input elemen daftar tertaut tidak digulung

Saat menyisipkan sebuah node, kita harus mengatur ulang node dalam unrolled linked list untuk mempertahankan properti yang disebutkan sebelumnya, bahwa setiap blok berisi $\lceil\sqrt{n}\rceil$ node. Misalkan kita menyisipkan simpul x setelah simpul ke-i, dan x harus ditempatkan di blok ke-j. Node di blok ke-j dan di blok setelah blok ke-j harus digeser ke arah ekor daftar sehingga masing-masing masih memiliki $\lceil\sqrt{n}\rceil$ node. Selain itu, blok baru perlu ditambahkan ke ekor jika blok terakhir dari daftar kehabisan ruang, yaitu memiliki lebih dari $\lceil\sqrt{n}\rceil$ node.

Melakukan Operasi Shift

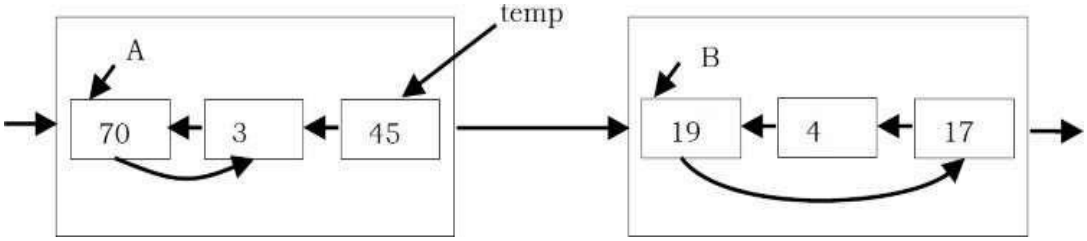
Perhatikan bahwa setiap operasi shift, yang mencakup penghapusan simpul dari ekor daftar tertaut melingkar di blok dan memasukkan simpul ke kepala daftar tertaut melingkar di blok setelahnya, hanya membutuhkan O(1). Kompleksitas waktu total dari operasi penyisipan untuk daftar tertaut yang belum dibuka adalah O($\lceil\sqrt{n}\rceil$); ada paling banyak O($\lceil\sqrt{n}\rceil$) blok dan oleh karena itu paling banyak O($\lceil\sqrt{n}\rceil$) operasi shift.

1. Pointer sementara diperlukan untuk menyimpan ekor A.



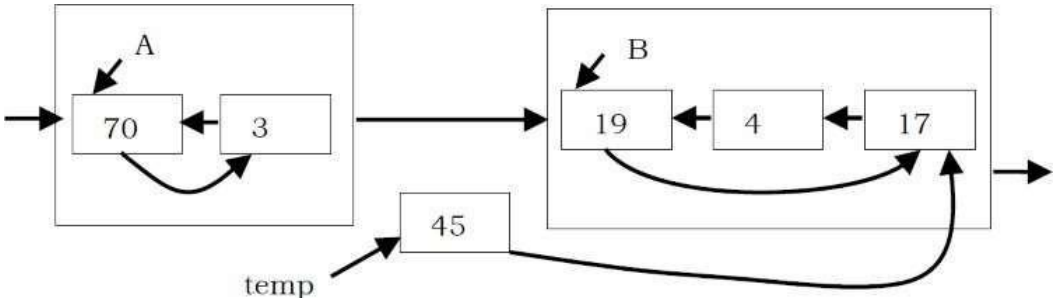
Gambar 3.51 fungsi pointer sementara

2. Di blok A, pindahkan penunjuk berikutnya dari simpul kepala untuk menunjuk ke simpul kedua hingga terakhir, sehingga simpul ekor A dapat dihilangkan.



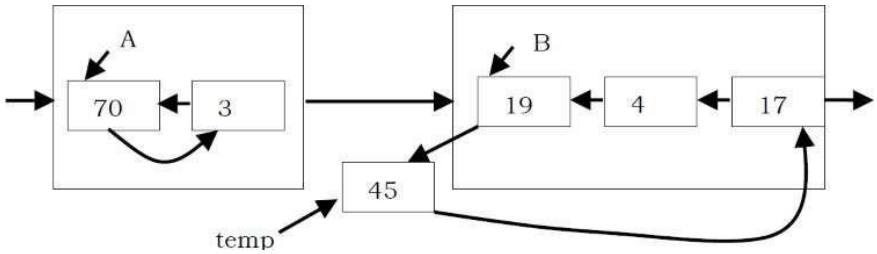
Gambar 3.52 cara menghilangkan simpul ekor A

3. Biarkan penunjuk berikutnya dari simpul yang akan digeser (simpul ekor A), menunjuk ke simpul ekor B.



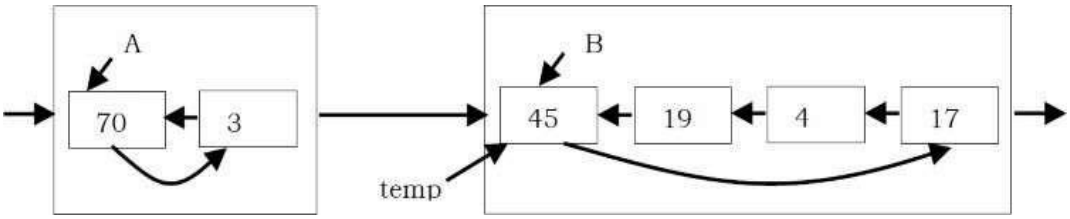
Gambar 3.53 simpul ekor A menunjuk simpul ekor B

4. Biarkan penunjuk berikutnya dari simpul kepala B menunjuk ke titik suhu simpul.



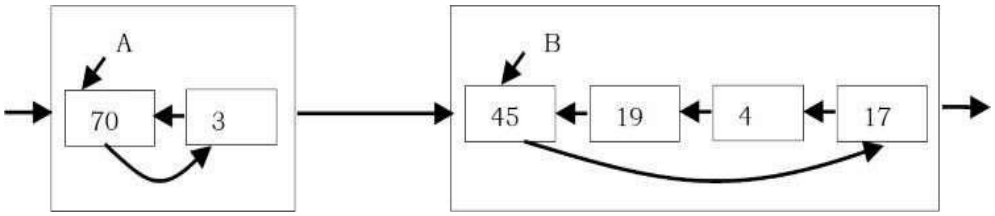
Gambar 3.54 simpul kepala B menunjuk suhu simpul

5. Terakhir, atur penunjuk kepala B untuk menunjuk ke titik temp node. Sekarang node temp menunjuk menjadi node kepala baru dari B.



Gambar 3.55 mengatur penunjuk kepala B

6. penunjuk suhu dapat dibuang. Kita telah menyelesaikan operasi shift untuk memindahkan simpul ekor asli A menjadi simpul kepala baru B.



Gambar 3.56 penyelesaian operasi shift

Pertunjukan

Dengan daftar tertaut yang belum dibuka, ada beberapa keuntungan, satu dalam kecepatan dan satu dalam ruang. Pertama, jika jumlah elemen di setiap blok berukuran tepat (misalnya, paling banyak ukuran satu baris cache), kita mendapatkan kinerja cache yang jauh lebih baik dari lokalitas memori yang ditingkatkan. Kedua, karena kita memiliki tautan O(n/m), di mana n adalah jumlah elemen dalam daftar tertaut yang belum dibuka dan m adalah jumlah elemen yang dapat kita simpan di blok mana pun, kita juga dapat menghemat jumlah ruang yang cukup besar, yang sangat terlihat jika setiap elemen kecil.

Membandingkan Linked List dan Linked Lists Unroled

Untuk membandingkan overhead untuk unrolled list, elemen dalam implementasi double linked list terdiri dari data, pointer ke node berikutnya, dan pointer ke node sebelumnya dalam daftar, seperti yang ditunjukkan di bawah ini.

```
struct ListNode {
    int data;
    struct ListNode *prev;
    struct ListNode *next;
};
```

Dengan asumsi kita memiliki 4 byte pointer, setiap node akan mengambil 8 byte. Tetapi alokasi overhead untuk node dapat berkisar antara 8 dan 16 byte. Mari kita pergi dengan kasus terbaik dan menganggap itu akan menjadi 8 byte. Jadi, jika kita ingin menyimpan item IK dalam daftar ini, kita akan memiliki overhead 16KB.

Sekarang, mari kita pikirkan tentang node daftar tertaut yang belum dibuka (sebut saja LinkedBlock). Ini akan terlihat seperti ini:

```
struct LinkedBlock{
   struct LinkedBlock *next;
   struct ListNode *head;
   int nodeCount;
};
```

Oleh karena itu, mengalokasikan satu node (12 byte + 8 byte overhead) dengan larik 100 elemen (400 byte + 8 byte overhead) sekarang akan dikenakan biaya 428 byte, atau 4,28 byte per elemen. Memikirkan item IK Kita dari atas, dibutuhkan sekitar 4.2KB overhead, yang mendekati 4x lebih baik dari daftar asli Kita. Bahkan jika daftar menjadi sangat terfragmentasi dan susunan item rata-rata hanya 1/2 penuh, ini masih merupakan peningkatan. Juga, perhatikan bahwa kita dapat menyetel ukuran array ke apa pun yang memberi kita overhead terbaik untuk aplikasi kita.

Penerapan

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int blockSize; //max number of nodes in a block

struct ListNode{
   struct ListNode* next;
   int value;
};
```

```
struct LinkedBlock{
    struct LinkedBlock *next;
    struct ListNode *head;
    int nodeCount;
};
struct LinkedBlock* blockHead;

//create an empty block
struct LinkedBlock* newLinkedBlock(){
    struct LinkedBlock* block=(struct LinkedBlock*)malloc(sizeof(struct LinkedBlock));
    block->next=NULL;
    block->head=NULL;
    block->nodeCount=0;
    return block;
}
//create a node
struct ListNode* newListNode(int value){
    struct ListNode* temp=(struct ListNode*)malloc(sizeof(struct ListNode));
    temp->next=NULL;
    temp->value=value;
    return temp;
}
void searchElement(int k,struct LinkedBlock **fLinkedBlock,struct ListNode **fListNode){
    //find the block
    int j=(k+blockSize-1)/blockSize; //k-th node is in the j-th block
    struct LinkedBlock* p=blockHead;
    while(--j){
        p=p->next;
    }
    *fLinkedBlock=p;
    //find the node
    struct ListNode* q=p->head;
    k=k%blockSize;
    if(k==0) k=blockSize;
    k=p->nodeCount+1-k;
    while(k--){
        q=q->next;
    }
    *fListNode=q;
}
//start shift operation from block *p
void shift(struct LinkedBlock *A){
    struct LinkedBlock *B;
    struct ListNode* temp;
    while(A->nodeCount > blockSize){ //if this block still have to shift
        if(A->next==NULL){ //reach the end. A little different
            A->next=newLinkedBlock();
            B=A->next;
            temp=A->head->next;
            A->head->next=A->head->next->next;
            B->head=temp;
            temp->next=temp;
            A->nodeCount--;
            B->nodeCount++;
        }else{
            B=A->next;
            temp=A->head->next;
            A->head->next=A->head->next->next;
            temp->next=B->head->next;
            B->head->next=temp;
            B->head=temp
            A->nodeCount--;
```

```
            B→nodeCount++;
        }
        A=B;
    }
}
void addElement(int k,int x){
    struct ListNode *p,*q;
    struct LinkedBlock *r;

    if(!blockHead){ //initial, first node and block
        blockHead=newLinkedBlock();
        blockHead→head=newListNode(x);
        blockHead→head→next=blockHead→head;
        blockHead→nodeCount++;
    }else{
        if(k==0){ //special case for k=0.
        p=blockHead→head;
        q=p→next;
        p→next=newListNode(x);
        p→next→next=q;
        blockHead→head=p→next;
        blockHead→nodeCount++;
        shift(blockHead);
        }else{
            searchElement(k,&r,&p);
            q=p;
            while(q→next!=p) q=q→next;
            q→next=newListNode(x);
            q→next→next=p;
            r→nodeCount++;
            shift(r);
        }
    }
}
int searchElement(int k){
    struct ListNode *p;
    struct LinkedBlock *q;
    searchElement(k,&q,&p);
    return p→value;
}
int testUnRolledLinkedList(){
    int tt=clock();
    int m,i,k,x;
    char cmd[10];
    scanf("%d",&m);
    blockSize=(int)(sqrt(m-0.001))+1;

    for( i=0; i<m; i++ ){
        scanf("%s",cmd);
        if(strcmp(cmd,"add")==0){
            scanf("%d %d",&k,&x);
            addElement(k,x);
        }else if(strcmp(cmd,"search")==0){
            scanf("%d",&k);
            printf("%d\n",searchElement(k));
        }else{
            fprintf(stderr,"Wrong Input\n");
        }
    }
    return 0;
}
```

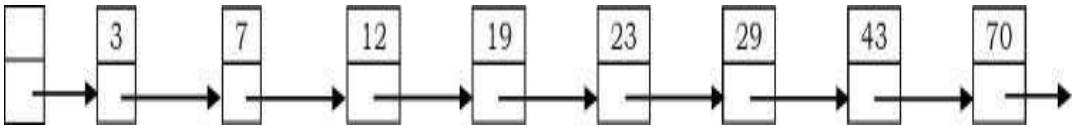
3.11 LEWATI DAFTAR

Pohon biner dapat digunakan untuk mewakili tipe data abstrak seperti kamus dan daftar terurut. Mereka bekerja dengan baik ketika elemen dimasukkan dalam urutan acak. Beberapa urutan operasi, seperti memasukkan elemen secara berurutan, menghasilkan struktur data yang merosot yang memberikan kinerja yang sangat buruk. Jika memungkinkan untuk secara acak mengubah daftar item yang akan dimasukkan, pohon akan bekerja dengan baik dengan probabilitas tinggi untuk setiap urutan input. Dalam kebanyakan kasus, pertanyaan harus dijawab secara online, jadi mengubah input secara acak menjadi tidak praktis. Algoritma pohon seimbang mengatur ulang pohon saat operasi dilakukan untuk mempertahankan kondisi keseimbangan tertentu dan memastikan kinerja yang baik.

Lewati daftar adalah alternatif probabilistik untuk pohon seimbang. Skip list adalah struktur data yang dapat digunakan sebagai alternatif untuk pohon biner seimbang (lihat bab Pohon). Dibandingkan dengan pohon biner, melewatkan daftar memungkinkan pencarian cepat, penyisipan dan penghapusan elemen. Ini dicapai dengan menggunakan penyeimbangan probabilistik daripada menerapkan penyeimbangan secara ketat. Ini pada dasarnya adalah daftar tertaut dengan pointer tambahan sehingga node perantara dapat dilewati. Ini menggunakan generator nomor acak untuk membuat beberapa keputusan.

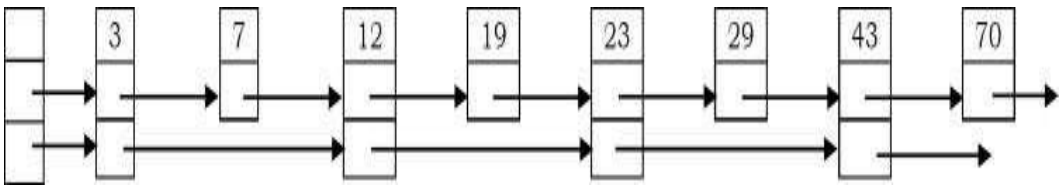
Dalam daftar tertaut yang diurutkan biasa, pencarian, penyisipan, dan penghapusan berada di O(n) karena daftar harus dipindai node-by-node dari kepala untuk menemukan node yang relevan. Jika entah bagaimana Kita dapat memindai daftar dalam langkah yang lebih besar (melompat ke bawah, seolah-olah), Kita akan mengurangi biaya pemindaian. Ini adalah ide mendasar di balik Skip Lists.

Lewati Daftar dengan Satu Level



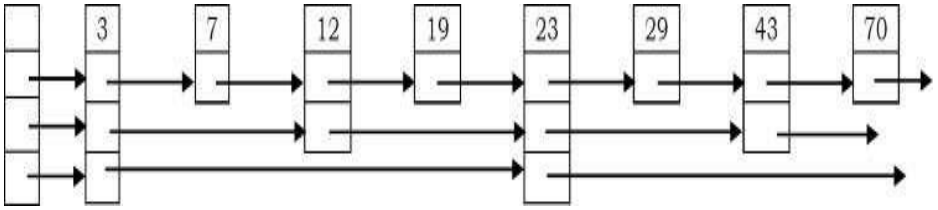
Gambar 3.57 lewati daftar satu kevel

Lewati Daftar dengan Dua Level



Gambar 3.58 lewati daftar dua kevel

Lewati Daftar dengan Tiga Level



Gambar 3.59 lewati daftar tiga Level

Mempertunjukkan

Dalam daftar tertaut sederhana yang terdiri dari n elemen, untuk melakukan pencarian n perbandingan diperlukan dalam kasus terburuk. Jika pointer kedua yang menunjuk dua node di depan ditambahkan ke setiap node, jumlah perbandingan turun menjadi n/2 + 1 dalam kasus terburuk.

Menambahkan satu pointer lagi ke setiap node keempat dan membuatnya menunjuk ke node keempat di depan mengurangi jumlah perbandingan menjadi $\lceil n/2 \rceil + 2$. Jika strategi ini dilanjutkan sehingga setiap node dengan pointer i menunjuk ke 2 * i – 1 node di depan, kinerja O(logn) diperoleh dan jumlah pointer hanya berlipat ganda (n + n/2 + n/4 + n/8 + n/16 + = 2n).

Operasi menemukan, menyisipkan, dan menghapus pada pohon pencarian biner biasa adalah efisien, O(logn), ketika data masukan acak; tetapi kurang efisien, O(n), ketika data input dipesan. Performa Lewati Daftar untuk operasi yang sama ini dan untuk kumpulan data apa pun hampir sama baiknya dengan pohon pencarian biner yang dibuat secara acak - yaitu O(logn).

Membandingkan Daftar Lewati dan Daftar Tertaut yang Dibuka

Secara sederhana, Lewati Daftar diurutkan daftar tertaut dengan dua perbedaan:

- Node dalam daftar biasa memiliki satu referensi berikutnya. Node dalam Daftar Lewati memiliki banyak referensi berikutnya (juga disebut referensi maju).
- Jumlah referensi maju untuk node tertentu ditentukan secara probabilistik.

Kita berbicara tentang node Skip List yang memiliki level, satu level per referensi ke depan. Jumlah level dalam sebuah node disebut ukuran node. Dalam daftar yang diurutkan biasa, operasi penyisipan, penghapusan, dan pencarian memerlukan traversal sekuensial dari daftar. Ini menghasilkan kinerja O(n) per operasi. Lewati Daftar memungkinkan node perantara dalam daftar dilewati selama traversal - menghasilkan kinerja O(logn) yang diharapkan per operasi.

Penerapan

```
#include <stdio.h>
#include <stdlib.h>
#define MAXSKIPLEVEL 5
struct ListNode {
   int data;
   struct ListNode *next[1];
};
struct SkipList {
   struct ListNode *header;
   int listLevel;              //current level of list */
};
struct SkipList list;
struct ListNode *insertElement(int data) {
   int i, newLevel;
   struct ListNode *update[MAXSKIPLEVEL+1];
   struct ListNode *temp;
   temp = list.header;
   for (i = list.listLevel; i >= 0; i--) {
      while (temp→next[i] !=list.header && temp→next[i]→data < data)
         temp = temp→next[i];
      update[i] = temp;
   }
   temp = temp→next[0];
   if (temp != list.header && temp→data == data)
      return(temp);
   //determine level
   for (newLevel = 0; rand() < RAND_MAX/2 && newLevel < MAXSKIPLEVEL; newLevel++);
   if (newLevel > list.listLevel) {
      for (i = list.listLevel + 1; i <= newLevel; i++)
         update[i] = list.header;
      list.listLevel = newLevel;
   }
   // make new node
   if ((temp = malloc(sizeof(Node) +
     newLevel*sizeof(Node *))) == 0) {
      printf ("insufficient memory (insertElement)\n");
      exit(1);
   }
   temp→data = data;
   for (i = 0; i <= newLevel; i++) {    // update next links
      temp→next[i] = update[i]→next[i];
      update[i]→next[i] = temp;
   }
   return(temp);
}
// delete node containing data
void deleteElement(int data) {
   int i;
   struct ListNode *update[MAXSKIPLEVEL+1], *temp;
   temp = list.header;
   for (i = list.listLevel; i >= 0; i--) {
      while (temp→next[i] != list.header && temp→next[i]→data < data)
         temp = temp→next[i];
      update[i] = temp;
   }
   temp = temp→next[0];
   if (temp == list.header | | !(temp→data == data) return;
   //adjust next pointers
   for (i = 0; i <= list.listLevel; i++) {
      if (update[i]→next[i] != temp) break;
      update[i]→next[i] = temp→next[i];
   }
```

```
    free (temp);
    //adjust header level
    while ((list.listLevel > 0) && (list.header→next[list.listLevel] == list.header))
      list.listLevel--;
}
// find node containing data
struct ListNode *findElement(int data) {
    int i;
    struct ListNode *temp = list.header;
    for (i = list.listLevel; i >= 0; i--) {
      while (temp→next[i] != list.header
        && temp→next[i]→data < data)
          temp = temp→next[i];
    }
    temp = temp→next[0];
    if (temp != list.header && temp→data == data) return (temp);
    return(0);
}
// initialize skip list
void initList() {
    int i;
    if ((list.header = malloc(sizeof(struct ListNode) + MAXSKIPLEVEL*sizeof(struct ListNode *))) == 0) {
      printf ("Memory Error\n");
      exit(1);
    }
    for (i = 0; i <= MAXSKIPLEVEL; i++)
      list.header→next[i] = list.header;
    list.listLevel = 0;
}
/* command-line: skipList maxnum skipList 2000: process 2000 sequential records */
int main(int argc, char **argv) {
    int i, *a, maxnum = atoi(argv[1]);
    initList();
    if ((a = malloc(maxnum * sizeof(*a))) == 0) {
      fprintf (stderr, "insufficient memory (a)\n");
      exit(1);
    }
    for (i = 0; i < maxnum; i++) a[i] = rand();
      printf ("Random, %d items\n", maxnum);
    for (i = 0; i < maxnum; i++) {
      insertElement(a[i]);
    }
    for (i = maxnum-1; i >= 0; i--) {
      findElement(a[i]);
    }
    for (i = maxnum-1; i >= 0; i--) {
      deleteElement(a[i]);
    }
    return 0;
}
```

3.12 DAFTAR TERTAUT: MASALAH & SOLUSI

Soal-1 Implementasikan Tumpukan menggunakan Linked List.

Solusi: Lihat bab Tumpukan.

Soal-2 Temukan simpul ke-n dari akhir Daftar Tertaut.

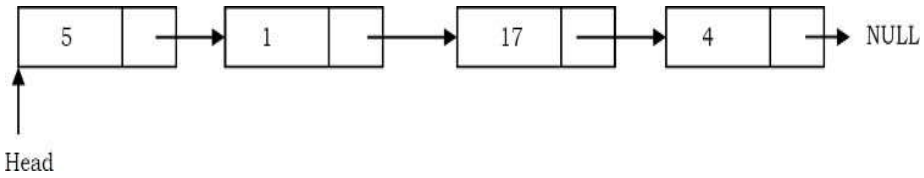
Solusi: Metode Brute-Force: Mulailah dengan node pertama dan hitung jumlah node yang ada setelah node tersebut. Jika jumlah node < n – 1 maka kembali dengan mengatakan "lebih sedikit jumlah node dalam daftar". Jika jumlah node > n – 1 maka lanjutkan ke node berikutnya. Lanjutkan ini sampai jumlah node setelah node saat ini adalah n – 1.

Kompleksitas Waktu: $O(n^2)$, untuk memindai daftar yang tersisa (dari node saat ini) untuk setiap node.

Kompleksitas Ruang: O(1).

Soal-3 Bisakah kita meningkatkan kompleksitas Soal-2?

Solusi: Ya, menggunakan tabel hash. Sebagai contoh perhatikan daftar berikut.



Dalam pendekatan ini, buat tabel hash yang entrinya < posisi node, alamat node >. Artinya, kunci adalah posisi simpul dalam daftar dan nilai adalah alamat simpul itu.

Posisi dalam Daftar	Alamat Node
1	Alamat 5 simpul
2	Alamat 1 simpul
3	Alamat 17 node
4	Alamat 4 simpul

Pada saat kita melintasi daftar lengkap (untuk membuat tabel hash), kita dapat menemukan panjang daftar. Katakanlah panjang daftar adalah M. Untuk menemukan ke-n dari akhir daftar tertaut, kita dapat mengonversinya ke M - n + 1 dari awal. Karena kita sudah tahu panjang daftarnya, itu hanya masalah mengembalikan nilai kunci M- n + 1 dari tabel hash.

Kompleksitas Waktu: Waktu untuk membuat tabel hash, T(m) = O(m).

Kompleksitas Ruang: Karena kita perlu membuat tabel hash dengan ukuran m, O(m).

Soal-4 Bisakah kita menggunakan pendekatan Soal-3 untuk menyelesaikan Soal-2 tanpa membuat tabel hash?

Solusi: Ya. Jika kita mengamati solusi Masalah-3, apa yang sebenarnya kita lakukan adalah menemukan ukuran dari daftar tertaut. Itu berarti kita menggunakan

tabel hash untuk menemukan ukuran daftar tertaut. Kita dapat menemukan panjang daftar tertaut hanya dengan memulai dari simpul kepala dan melintasi daftar.

Jadi, kita dapat menemukan panjang daftar tanpa membuat tabel hash. Setelah menemukan panjangnya, hitung M – n + 1 dan dengan satu scan lagi kita bisa mendapatkan node ke-1 M – n+ dari awal. Solusi ini membutuhkan dua pemindaian: satu untuk menemukan panjang daftar dan yang lainnya untuk menemukan M – n+ simpul ke-1 dari awal.

Kompleksitas Waktu: Waktu untuk menemukan panjang + Waktu untuk menemukan simpul M – n + 1 dari awal. Oleh karena itu, T(n) = O(n) + O(n) O(n). Kompleksitas Ruang: O(1). Oleh karena itu, tidak perlu membuat tabel hash.

Soal-5 Bisakah kita memecahkan Soal-2 dalam sekali scan?

Solusi: Ya. Pendekatan Efisien: Gunakan dua pointer pNthNode dan pTemp. Awalnya, keduanya menunjuk ke simpul kepala daftar. pNthNode mulai bergerak hanya setelah pTemp melakukan n gerakan.

Dari sana keduanya bergerak maju hingga pTemp mencapai akhir daftar. Akibatnya pNthNode menunjuk ke simpul ke-n dari akhir daftar tertaut.

Catatan: *Setiap saat keduanya bergerak satu node pada satu waktu.*

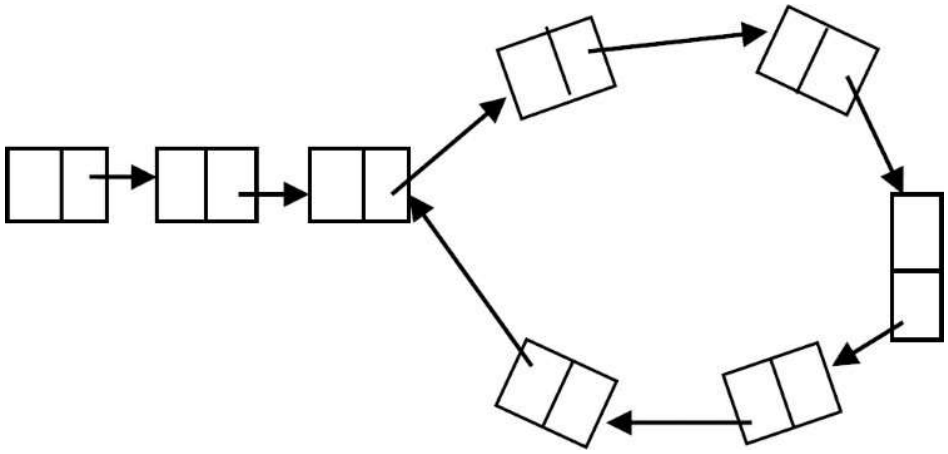
```
struct ListNode *NthNodeFromEnd(struct ListNode *head, int NthNode){
    struct ListNode *pNthNode = NULL, *pTemp = head;
    for(int count =1; count< NthNode;count++) {
        if(pTemp)
            pTemp = pTemp→next;
    }
    while(pTemp) {
        if(pNthNode == NULL)
            pNthNode = head;
        else
            pNthNode = pNthNode→next;
        pTemp = pTemp→next;
    }
    if(pNthNode)
       return pNthNode;
    return NULL;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-6 Periksa apakah daftar tertaut yang diberikan diakhiri dengan NULL atau diakhiri dengan siklus (siklus).

Solusi: Pendekatan Brute-Force. Sebagai contoh, perhatikan daftar tertaut berikut yang memiliki loop di dalamnya. Perbedaan antara daftar ini dan daftar biasa adalah, dalam daftar ini, ada dua simpul yang penunjuk berikutnya sama. Dalam daftar tertaut tunggal reguler (tanpa loop) setiap pointer berikutnya adalah unik.

Itu berarti pengulangan pointer berikutnya menunjukkan adanya loop.



Salah satu cara sederhana dan kasar untuk menyelesaikannya adalah, mulai dengan node pertama dan lihat apakah ada node yang penunjuk berikutnya adalah alamat node saat ini. Jika ada node dengan alamat yang sama maka itu menunjukkan bahwa beberapa node lain menunjuk ke node saat ini dan kita dapat mengatakan bahwa ada loop. Lanjutkan proses ini untuk semua node dari daftar tertaut.

Apakah metode ini berhasil? Sesuai algoritma, Kita memeriksa alamat penunjuk berikutnya, tetapi bagaimana Kita menemukan akhir dari daftar tertaut (jika tidak, Kita akan berakhir di loop tak terbatas)?

Catatan: *Jika kita memulai dengan sebuah simpul dalam satu lingkaran, metode ini dapat bekerja tergantung pada ukuran loop.*

Soal-7 **:** Bisakah kita menggunakan teknik hashing untuk menyelesaikan Soal-6?

Solusi **:** Ya. Menggunakan Tabel Hash kita dapat memecahkan masalah ini.

Algoritma:

- Lintasi node daftar tertaut satu per satu.
- Periksa apakah alamat node tersedia di tabel hash atau tidak.
- Jika sudah tersedia di tabel hash, itu menunjukkan bahwa kita sedang mengunjungi node yang sudah dikunjungi. Ini hanya mungkin jika daftar tertaut yang diberikan memiliki loop di dalamnya.
- Jika alamat node tidak tersedia di tabel hash, masukkan alamat node tersebut ke tabel hash.
- Lanjutkan proses ini sampai kita mencapai akhir dari linked list atau kita menemukan loop.

Kompleksitas Waktu; O(n) untuk memindai daftar tertaut. Perhatikan bahwa Kita melakukan pemindaian hanya pada input.

Kompleksitas Ruang; O(n) untuk tabel hash.

Soal-8 Bisakah kita menyelesaikan Soal-6 menggunakan teknik sortir?

Solusi:Tidak. Perhatikan algoritma berikut yang didasarkan pada pengurutan. Lalu kita lihat mengapa algoritma ini gagal.

Algoritma:

- Lintasi node daftar tertaut satu per satu dan ambil semua nilai pointer berikutnya ke dalam array.
- Urutkan array yang memiliki pointer node berikutnya.
- Jika ada loop dalam linked list, pasti dua pointer node berikutnya akan menunjuk ke node yang sama.
- Setelah mengurutkan jika ada loop dalam daftar, node yang pointer berikutnya adalah sama akan berakhir berdekatan dalam daftar yang diurutkan.
- Jika ada pasangan seperti itu dalam daftar yang diurutkan, maka kita katakan bahwa daftar tertaut memiliki loop di dalamnya.

Kompleksitas Waktu; O(nlogn) untuk menyortir array pointer berikutnya.

Kompleksitas Ruang; O(n) untuk array pointer berikutnya.

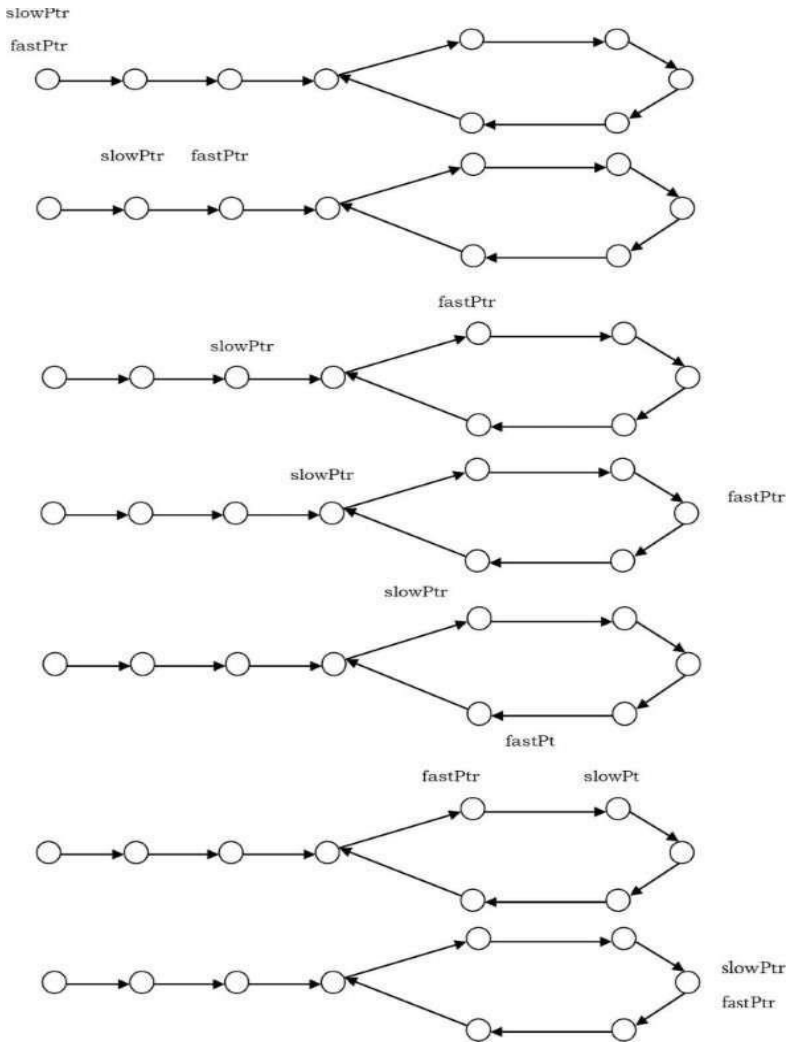
Masalah dengan algoritma di atas: Algoritma di atas hanya berfungsi jika kita dapat menemukan panjang daftar. Tetapi jika daftar memiliki loop maka kita mungkin berakhir di loop tak terbatas. Karena alasan ini algoritma gagal.

Soal-9 Bisakah kita menyelesaikan Soal-6 di O(n)?

Solusi: Ya. Pendekatan Efisien (Pendekatan Tanpa Memori): Masalah ini diselesaikan oleh Floyd. Solusinya diberi nama algoritma pencarian siklus Floyd. Ini menggunakan dua pointer yang bergerak dengan kecepatan berbeda untuk berjalan di daftar tertaut. Begitu mereka memasuki loop, mereka diharapkan bertemu, yang menunjukkan bahwa ada loop.

Ini berfungsi karena satu-satunya cara penunjuk yang bergerak lebih cepat akan menunjuk ke lokasi yang sama dengan penunjuk yang bergerak lebih lambat adalah jika entah bagaimana seluruh daftar atau sebagiannya melingkar. Pikirkan seekor kura-kura dan kelinci yang berlari di lintasan. Kelinci yang berlari lebih cepat akan mengejar kura-kura jika mereka berlari dalam satu lingkaran. Sebagai contoh, perhatikan contoh berikut dan telusuri algoritma Floyd. Dari diagram di bawah ini kita dapat melihat bahwa setelah langkah terakhir mereka bertemu di beberapa titik dalam loop yang mungkin bukan titik awal loop.

Catatan: *slowPtr (kura-kura) menggerakkan satu penunjuk sekaligus dan fastPtr (kelinci) menggerakkan dua penunjuk sekaligus.*




```
int DoesLinkedListHasLoop(struct ListNode * head) {
    struct ListNode *slowPtr = head, *fastPtr = head;
    while (slowPtr && fastPtr && fastPtr→next) {
      slowPtr = slowPtr→next;
      fastPtr = fastPtr→next→next;
      if (slowPtr == fastPtr)
        return 1;
    }
    return 0;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-10 diberi penunjuk ke elemen pertama dari daftar tertaut L. Ada dua kemungkinan untuk L: itu berakhir (ular) atau elemen terakhirnya menunjuk kembali ke salah satu elemen sebelumnya dalam daftar (siput). Berikan algoritma yang menguji apakah daftar L yang diberikan adalah ular atau siput.

Solusi: Sama dengan Soal-6.

Soal-11 Periksa apakah daftar tertaut yang diberikan diakhiri dengan NULL atau tidak. Jika ada siklus, temukan simpul awal dari loop.

Solusi: Solusinya adalah perpanjangan dari solusi di Soal-9. Setelah menemukan loop dalam daftar tertaut, Kita menginisialisasi slowPtr ke kepala daftar tertaut. Sejak saat itu, baik slowPtr maupun fastPtr hanya memindahkan satu node pada satu waktu. Titik di mana mereka bertemu adalah awal dari loop. Umumnya Kita menggunakan metode ini untuk menghilangkan loop.

```
int FindBeginofLoop(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        int loopExists = 0;
        while (slowPtr && fastPtr && fastPtr→next) {
           slowPtr = slowPtr→next;
           fastPtr = fastPtr→next→next;
           if (slowPtr == fastPtr){
             loopExists = 1;
             break;
           }
        }
        if(loopExists) {
          slowPtr = head;
          while(slowPtr  != fastPtr) {
            fastPtr = fastPtr→next;
            slowPtr = slowPtr→next;
          }
          return slowPtr;
        }
        return NULL;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-12 Dari pembahasan dan masalah sebelumnya kita memahami bahwa pertemuan kura-kura dan kelinci menyimpulkan adanya loop, tetapi bagaimana cara memindahkan kura-kura ke awal linked list sambil menjaga kelinci di tempat pertemuan, diikuti dengan memindahkan keduanya selangkah demi selangkah, membuat mereka bertemu di titik awal siklus?

Solusi: Masalah ini adalah inti dari teori bilangan. Dalam algoritma pencarian siklus Floyd, perhatikan bahwa kura-kura dan kelinci akan bertemu ketika mereka n × L, di mana L adalah panjang lingkaran. Selanjutnya, kura-kura berada di titik tengah antara kelinci dan awal urutan karena cara mereka bergerak. Oleh karena itu kura-kura n × L jauh dari awal urutan juga. Jika kita memindahkan keduanya satu langkah pada satu waktu, dari posisi kura-kura dan dari awal urutan, kita tahu bahwa mereka akan bertemu segera setelah keduanya berada di loop, karena mereka n × L, kelipatan dari panjang lingkaran, terpisah. Salah

satunya sudah dalam loop, jadi Kita hanya memindahkan yang lain dalam satu langkah sampai memasuki loop, menjaga n × L lainnya menjauh darinya setiap saat.

Soal-13 Dalam algoritma pencarian siklus Floyd, apakah ini berhasil jika kita menggunakan langkah 2 dan 3, bukan 1 dan 2?

Solusi: Ya, tetapi kerumitannya mungkin tinggi. Menelusuri sebuah contoh.

Soal-14 Periksa apakah daftar tertaut yang diberikan diakhiri dengan NULL. Jika ada siklus, tentukan panjang loop.

Solusi: Solusi ini juga merupakan perluasan dari masalah deteksi siklus dasar. Setelah menemukan loop dalam daftar tertaut, pertahankan slowPtr apa adanya. FastPtr terus bergerak sampai kembali lagi ke slowPtr. Saat bergerak cepatPtr, gunakan variabel penghitung yang bertambah pada tingkat 1.

```
int FindLoopLength(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        int loopExists = 0, counter = 0;
        while (slowPtr && fastPtr && fastPtr→next) {
            slowPtr = slowPtr→next;
            fastPtr = fastPtr→next→next;
            if (slowPtr == fastPtr){
                loopExists = 1;
                break;
            }
        }
        if(loopExists) {
          fastPtr = fastPtr→next;
          while(slowPtr  != fastPtr) {
            fastPtr = fastPtr→next;
            counter++;
          }
          return counter;
        }
        return 0;                                   //If no loops exists
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Soal-15 Menyisipkan simpul dalam daftar tertaut yang diurutkan.

Solusi: Telusuri daftar dan temukan posisi untuk elemen dan masukkan.

```
struct ListNode *InsertInSortedList(struct ListNode * head, struct ListNode * newNode) {
        struct ListNode *current = head, temp;
        if(!head)
              return newNode;
        // traverse the list until you find item bigger the new node value
        while (current != NULL && current→data <  newNode→data){
              temp = current;
             current = current→next;
        }
        //insert the new node before the big item
        newNode→next = current;
        temp→next = newNode;
        return head;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-16 Membalikkan daftar tertaut tunggal.

Solusi:

```
// Iterative version
struct ListNode *ReverseList(struct ListNode *head) {
  struct ListNode *temp = NULL, *nextNode = NULL;
  while (head) {
    nextNode = head→next;
    head→next = temp;
    temp = head;
    head = nextNode;
  }
  return temp;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Versi rekursif: Kita akan lebih mudah memulai dari bawah ke atas, dengan mengajukan dan menjawab pertanyaan-pertanyaan kecil (inilah pendekatan dalam The Little Lisper):

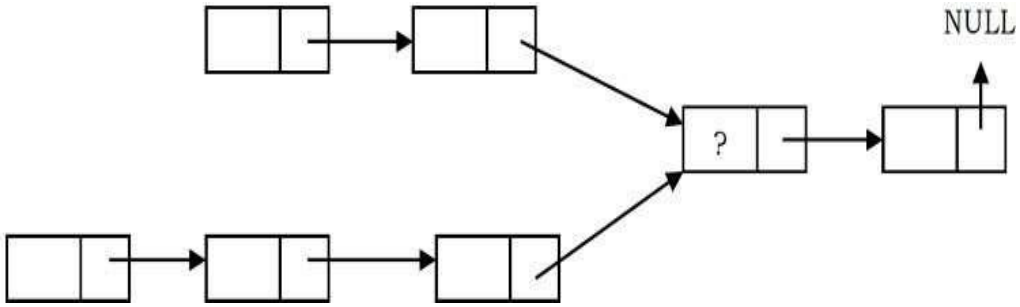
- Apa kebalikan dari NULL (daftar kosong)? BATAL.
- Apa kebalikan dari daftar satu elemen? Elemen itu sendiri.
- Apa kebalikan dari daftar elemen n? Kebalikan dari elemen kedua diikuti oleh elemen pertama.

```
struct ListNode * RecursiveReverse(struct ListNode *head) {
    if (head == NULL)
        return NULL;
    if (head→next == NULL)
        return list;
    struct ListNode *secondElem = head→next;
    // Need to unlink list from the rest or you will get a cycle
    head→next = NULL;
    // reverse everything from the second element on
    struct ListNode *reverseRest = RecursiveReverse(secondElem);
    secondElem→next = head;        // then we join the two lists
    return reverseRest;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk tumpukan rekursif.

Soal-17 Misalkan ada dua daftar tertaut tunggal yang keduanya berpotongan di beberapa titik dan menjadi satu daftar tertaut. Pointer kepala atau awal dari kedua daftar diketahui, tetapi simpul yang berpotongan tidak diketahui. Juga, jumlah node di setiap daftar sebelum mereka berpotongan tidak diketahui dan mungkin berbeda di setiap daftar. List1 mungkin memiliki n node sebelum



mencapai titik persimpangan, dan List2 mungkin memiliki m node sebelum mencapai titik persimpangan di mana m dan n mungkin m = n,m < n atau m > n. Berikan algoritma untuk menemukan titik penggabungan.

Solusi: Pendekatan Brute-Force: Salah satu solusi mudah adalah membandingkan setiap penunjuk simpul di daftar pertama dengan setiap penunjuk simpul lain di daftar kedua di mana penunjuk simpul yang cocok akan membawa kita ke simpul berpotongan. Tetapi, kompleksitas waktu dalam hal ini adalah O(mn) yang akan tinggi.

Kompleksitas Waktu: O(mn).

Kompleksitas Ruang: O(1).

Soal-18 Bisakah kita menyelesaikan Soal-17 dengan menggunakan teknik pengurutan?

Solusi **Tidak.** Pertimbangkan algoritma berikut yang didasarkan pada pengurutan dan lihat mengapa algoritma ini gagal.

Algoritma:

- Ambil pointer node daftar pertama dan simpan dalam beberapa array dan urutkan.
- Ambil pointer node daftar kedua dan simpan dalam beberapa array dan urutkan.
- Setelah pengurutan, gunakan dua indeks: satu untuk larik terurut pertama dan yang lainnya untuk larik terurut kedua.
- Mulai membandingkan nilai pada indeks dan menaikkan indeks menurut mana yang memiliki nilai lebih rendah (kenaikan hanya jika nilainya tidak sama).
- Di sembarang titik, jika kita dapat menemukan dua indeks yang nilainya sama, maka itu menunjukkan bahwa kedua node tersebut menunjuk ke node yang sama dan kita mengembalikan node tersebut.

Kompleksitas Waktu: Waktu untuk menyortir daftar + Waktu untuk memindai (untuk membandingkan)
= O(mlogm) +O(nlogn) +O(m + n) Kita perlu mempertimbangkan salah satu yang memberikan nilai maksimum.

Kompleksitas Ruang: O(1).

Adakah masalah dengan algoritma di atas? Ya.

Dalam algoritma, Kita menyimpan semua penunjuk simpul dari daftar dan pengurutan. Tapi kita melupakan fakta bahwa bisa ada banyak elemen yang berulang. Ini karena setelah titik penggabungan, semua penunjuk simpul sama untuk kedua daftar. Algoritma berfungsi dengan baik hanya dalam satu kasus dan saat kedua daftar memiliki simpul akhir pada titik gabungannya.

Soal-19 Bisakah kita memecahkan Soal-17 menggunakan tabel hash?

Solusi **: Ya**.

Algoritma:

- Pilih daftar yang memiliki jumlah node lebih sedikit (Jika kita tidak mengetahui panjangnya sebelumnya, pilih satu daftar secara acak).
- Sekarang, telusuri daftar lain dan untuk setiap penunjuk simpul dari daftar ini periksa apakah penunjuk simpul yang sama ada di tabel hash.
- Jika ada titik gabungan untuk daftar yang diberikan maka kita pasti akan menemukan penunjuk simpul di tabel hash.

Kompleksitas Waktu: Waktu untuk membuat tabel hash + Waktu untuk memindai daftar kedua = O(m) + O(n) (atau O(n) + O(m), tergantung pada daftar mana yang kita pilih untuk membuat tabel hash. Tapi di keduanya kasus kompleksitas waktu adalah sama.

Kompleksitas Ruang: O(n) atau O(m).

Soal-20 Bisakah kita menggunakan tumpukan untuk menyelesaikan Soal-17?

Solusi **Ya.**

Algoritma:

- Buat dua tumpukan: satu untuk daftar pertama dan satu untuk daftar kedua.
- Telusuri daftar pertama dan dorong semua alamat node ke tumpukan pertama.
- Telusuri daftar kedua dan dorong semua alamat node ke tumpukan kedua.
- Sekarang kedua tumpukan berisi alamat simpul dari daftar yang sesuai.
- Sekarang bandingkan alamat node teratas dari kedua tumpukan.
- Jika sama, ambil elemen teratas dari kedua tumpukan dan simpan dalam beberapa variabel sementara (karena kedua alamat simpul adalah simpul, cukup jika kita menggunakan satu variabel sementara).
- Lanjutkan proses ini sampai alamat node teratas dari tumpukan tidak sama.
- Titik ini adalah titik di mana daftar bergabung menjadi satu daftar.
- Mengembalikan nilai variabel sementara.

Kompleksitas Waktu: O(m + n), untuk memindai kedua daftar.

Kompleksitas Ruang: O(m + n), untuk membuat dua tumpukan untuk kedua daftar.

Soal-21 Apakah ada cara lain untuk menyelesaikan Soal-17?

Solusi: Ya. Menggunakan pendekatan "menemukan angka berulang pertama" dalam array (untuk algoritma lihat bab Pencarian).

Algoritma:

- Buat larik A dan simpan semua penunjuk berikutnya dari kedua daftar dalam larik.
- Dalam larik temukan elemen berulang pertama [Lihat bab Pencarian untuk algoritma].
- Angka pertama yang berulang menunjukkan titik penggabungan kedua daftar.

Kompleksitas Waktu: O(m + n).

Kompleksitas Ruang: O(m + n).

Soal-22 Masih bisakah kita berpikir untuk menemukan solusi alternatif untuk Soal-17?

Solusi: Ya. Dengan menggabungkan teknik penyortiran dan pencarian, kita dapat mengurangi kerumitannya.

Algoritma:

- Buat array A dan simpan semua pointer berikutnya dari daftar pertama dalam array.
- Urutkan elemen array ini.
- Kemudian, untuk setiap elemen daftar kedua, cari dalam larik terurut (mari kita asumsikan bahwa kita menggunakan pencarian biner yang menghasilkan O(logn)).
- Karena kita memindai daftar kedua satu per satu, elemen berulang pertama yang muncul dalam larik tidak lain adalah titik penggabungan.

Kompleksitas Waktu: Waktu untuk menyortir + Waktu untuk mencari = O(Max(mlogm, nlogn)). Kompleksitas Ruang: O(Maks(m, n)).

Soal-23 Bisakah kita meningkatkan kompleksitas Soal-17?

Solusi: Ya. Pendekatan yang Efisien:

- Cari panjang (L1 dan L2) dari kedua daftar - O(n) + O(m) = O(max(m, n)).
- Ambil perbedaan d dari panjang -- O(1).
- Buat d langkah dalam daftar yang lebih panjang -- O(d).
- Langkah di kedua daftar secara paralel sampai link ke node berikutnya cocok -- O(min(m, n)).
- Kompleksitas waktu total = O(maks(m, n)).
- Kompleksitas Ruang = O(1).

```
struct ListNode* FindIntersectingNode(struct ListNode* list1, struct ListNode* list2) {
    int L1=0, L2=0, diff=0;
    struct ListNode *head1 = list1, *head2 = list2;
    while(head1!= NULL) {
       L1++;
       head1 = head1→next;
    }
    while(head2!= NULL) {
       L2++;
       head2 = head2→next;
    }
    if(L1 < L2) {
       head1 = list2;  head2 = list1;  diff = L2 - L1;
    }else{
       head1 = list1;  head2 = list2;  diff = L1 – L2;
    }
    for(int i = 0; i < diff; i++)
       head1 = head1→next;
    while(head1 != NULL && head2 != NULL) {
       if(head1 == head2)
          return head1→data;
       head1= head1→next;
       head2= head2→next;
    }
    return NULL;
}
```

Soal-24 Bagaimana Anda akan menemukan bagian tengah dari daftar tertaut?

Solusi: Pendekatan Brute-Force: Untuk setiap simpul, hitung berapa banyak simpul yang ada dalam daftar, dan lihat apakah itu simpul tengah dari daftar.

Kompleksitas Waktu: O(n2).

Kompleksitas Ruang: O(1).

Soal-25 Bisakah kita meningkatkan kompleksitas Soal-24?

Solusi **Ya.**

Algoritma:

- Telusuri daftar dan temukan panjang daftar.
- Setelah menemukan panjangnya, pindai lagi daftar dan temukan n/2 node dari awal.

Kompleksitas Waktu: Waktu untuk menemukan panjang daftar + Waktu untuk menemukan simpul tengah = O(n) + O(n) O(n).

Kompleksitas Ruang: O(1).

Soal-26 Bisakah kita menggunakan tabel hash untuk menyelesaikan Soal-24?

Solusi: Ya. Alasannya sama dengan Soal-3.

Kompleksitas Waktu: Waktu untuk membuat tabel hash. Oleh karena itu, T(n) = O(n).

Kompleksitas Ruang: O(n). Karena kita perlu membuat tabel hash berukuran n.

Soal-27 Bisakah kita memecahkan Soal-24 hanya dalam satu scan?

Solusi **Pendekatan Efisien:** Gunakan dua pointer. Gerakkan satu penunjuk dengan kecepatan dua kali lipat detik. Ketika penunjuk pertama mencapai akhir daftar, penunjuk kedua akan menunjuk ke simpul tengah.

Catatan: *Jika daftar memiliki jumlah simpul yang genap, simpul tengah adalah ⌈n/2⌉.*

```
struct ListNode * FindMiddle(struct ListNode *head) {
        struct ListNode *ptr1x, *ptr2x;
        ptr1x = ptr2x = head;
        int i=0;
        // keep looping until we reach the tail (next will be NULL for the last node)
        while(ptr1x→next != NULL) {
           if(i == 0) {
              ptr1x = ptr1x→next; //increment only the 1st pointer
              i=1;
           }
           else if( i == 1) {
              ptr1x = ptr1x→next; //increment both pointers
              ptr2x = ptr2x→next;
              i = 0;
           }
        }
        return ptr2x;       //now return the ptr2 which points to the middle node
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Soal-28 Bagaimana Anda akan menampilkan Linked List dari akhir?

Solusi: Traverse secara rekursif hingga akhir daftar tertaut. Saat kembali, mulailah mencetak elemen.

```
//This Function will print the linked list from end
void PrintListFromEnd(struct ListNode *head) {
   if(!head)
      return;

   PrintListFromEnd(head→next);
   printf("%d ",head→data);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n)→ untuk Tumpukan.

Soal-29 Periksa apakah panjang Linked List yang diberikan genap atau ganjil?

Solusi: Gunakan pointer 2x. Ambil pointer yang bergerak 2x [dua node sekaligus]. Pada akhirnya, jika panjangnya genap, maka pointer akan menjadi NULL; jika tidak, itu akan menunjuk ke simpul terakhir.

```
int IsLinkedListLengthEven(struct ListNode * listHead) {
   while(listHead && listHead→next)
      listHead = listHead→next→next;
   if(!listHead)
      return 0;
   return 1;
}
```

Kompleksitas Waktu: O(⌊n/2⌋) O(n).
Kompleksitas Ruang: O(1).

Soal-30 Jika kepala dari Linked List menunjuk ke elemen ke-k, lalu bagaimana Anda akan mendapatkan elemen sebelum elemen ke-k?

Solusi: Gunakan Daftar Tertaut yang Hemat Memori [Daftar Tertaut XOR].

Soal-31 Mengingat dua Daftar Tertaut yang diurutkan, bagaimana cara menggabungkannya ke dalam daftar ketiga dalam urutan yang diurutkan?

Solusi: Asumsikan ukuran daftar adalah m dan n.

Rekursif:

```
struct ListNode *MergeSortedList(struct ListNode *a, struct ListNode *b) {
    struct ListNode *result = NULL;
    if(a == NULL) return b;
    if(b == NULL) return a;
    if(a→data <= b→data) {
        result =a;
        result→next = MergeSortedList(a→next, b);
    }
    else {
        result =b;
        result→next = MergeSortedList(b→next,a);
    }
    return result;
}
```

Kompleksitas Waktu: O(n + m), di mana n dan m adalah panjang dari dua daftar.

Berulang:

```
struct ListNode *MergeSortedListIterative(struct ListNode *head1, struct ListNode *head2){
    struct ListNode * newNode = (struct ListNode*) (malloc(sizeof(struct ListNode)));
    struct ListNode *temp;
    newNode = new Node;
    newNode→next = NULL;
    temp = newNode;
    while (head1!=NULL and head2!=NULL){
        if (head1→data<=head2→data){
            temp→next = head1;
            temp = temp→next;
            head1 = head1→next;
        }else{
            temp→next = head2;
            temp = temp→next;
            head2 = head2→next;
        }
    }
    if (head1!=NULL)
        temp→next = head1;
    else
        temp→next = head2;

    temp = newNode→next;
    free(newNode);
    return temp;
}
```

Kompleksitas Waktu: O(n + m), di mana n dan m adalah panjang dari dua daftar.

Soal-32 Membalikkan daftar tertaut secara berpasangan. Jika Anda memiliki daftar tertaut yang menampung 1 → 2 → 3 → 4 → X, maka setelah fungsi dipanggil, daftar tertaut akan menampung 2 → 1 → 4 → 3 → X.

Solusi: Rekursif:

```
struct ListNode *ReversePairRecursive(struct ListNode *head) {
        struct ListNode *temp;
        if(head ==NULL || head→next ==NULL)
                return;   //base case for empty or 1 element list
        else {
              //Reverse first pair
                temp = head→next;
                head→next = temp→next;
                temp→next = head;
                head = temp;
                //Call the method recursively for the rest of the list
                head→next→next = ReversePairRecursive(head→next→next);
                return head;
        }
}
```

Berulang:

```
struct ListNode *ReversePairIterative(struct ListNode *head) {
        struct ListNode *temp1=NULL, *temp2=NULL, *current = head;
        while(current != NULL && current→next != NULL)  {
              if (temp1 != null) {
                temp1→next→next = current→next;
              }
              temp1 = current→next;
              current→next = current→next→next;
              temp1.next = current;
              if (temp2 == null)
                 temp2 = temp1;
             current = current→next;
        }
       return temp2;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-33 Diberikan pohon biner, konversikan ke daftar tertaut ganda.
Solusi: Lihat bab Pohon.

Soal-34 Bagaimana kita mengurutkan Linked List?
Solusi: Lihat bab Penyortiran.

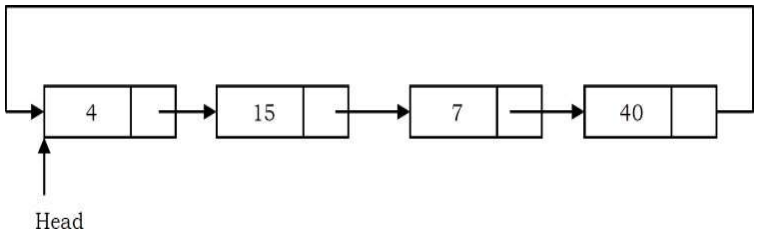
Soal-35 Bagilah sebuah Circular Linked List menjadi dua bagian yang sama. Jika jumlah node dalam daftar ganjil maka buat daftar pertama satu node ekstra dari daftar kedua.

Solusi:

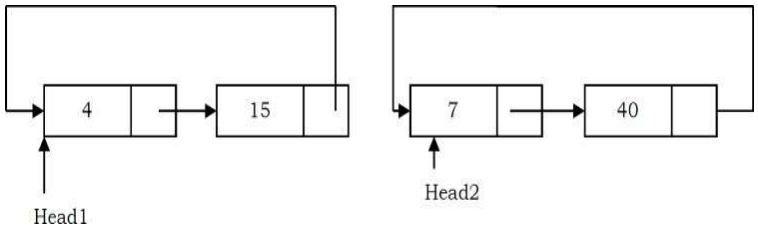
Algoritma:

- Simpan pointer tengah dan terakhir dari daftar tertaut melingkar menggunakan algoritma pencarian siklus Floyd.
- Buat setengah lingkaran kedua.
- Buat setengah lingkaran pertama.
- Setel penunjuk kepala dari dua daftar tertaut.

Sebagai contoh, perhatikan daftar lingkaran berikut.



Setelah dipecah, daftar di atas akan terlihat seperti:



```
// structure for a node
struct ListNode {
        int data;
        struct ListNode *next;
};
void SplitList(struct ListNode *head, struct ListNode **head1, struct ListNode **head2) {
        struct ListNode *slowPtr = head;
        struct ListNode *fastPtr = head;
        if(head == NULL)
                return;
        /* If there are odd nodes in the circular list then fastPtr→next becomes
           head and for even nodes fastPtr→next→next becomes head */
        while(fastPtr→next != head && fastPtr→next→next != head) {
                fastPtr = fastPtr→next→next;
                slowPtr = slowPtr→next;
        }
        // If there are even elements in list then move fastPtr
        if(fastPtr→next→next == head)
                fastPtr = fastPtr→next;
        // Set the head pointer of first half
        *head1 = head;
        // Set the head pointer of second half
        if(head→next != head)
                *head2 = slowPtr→next;
        // Make second half circular
        fastPtr→next = slowPtr→next;
        // Make first half circular
        slowPtr→next = head;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-36 Jika kita ingin menggabungkan dua daftar tertaut, manakah dari berikut ini yang memberikan kompleksitas O(1)?

1) Daftar tertaut tunggal
2) Daftar tertaut ganda
3) Daftar tertaut ganda melingkar

Solusi: Daftar Tertaut Ganda Melingkar. Ini karena untuk daftar tertaut tunggal dan ganda, Kita perlu melintasi daftar pertama sampai akhir dan menambahkan daftar kedua. Tetapi dalam kasus daftar tertaut ganda melingkar, kita tidak harus melintasi daftar.

Soal-37 Bagaimana Anda akan memeriksa apakah daftar tertaut adalah palindrom atau tidak?

Solusi:

Algoritma:

1. Dapatkan bagian tengah dari daftar tertaut.
2. Balikkan paruh kedua daftar tertaut.
3. Bandingkan babak pertama dan babak kedua.
4. Buat daftar tertaut asli dengan membalik bagian kedua lagi dan melampirkannya kembali ke bagian pertama.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-38 Untuk nilai K tertentu (K > 0) membalikkan blok dari K node dalam sebuah daftar.
Contoh: Input: 1 2 3 4 5 6 7 8 9 10. Output untuk nilai K yang berbeda:
Untuk K = 2: 2 1 4 3 6 5 8 7 10 9
Untuk K = 3: 3 2 1 6 5 4 9 8 7 10
Untuk K = 4: 4 3 2 1 8 7 6 5 9 10

Solusi

Algoritma Ini adalah perpanjangan dari pertukaran node dalam daftar tertaut.

1) Periksa apakah daftar yang tersisa memiliki K node.
   A. Jika ya, dapatkan pointer dari K + 1th node.
   B. Lain kembali.
2) Membalikkan K node pertama.
3) Setel berikutnya dari simpul terakhir (setelah pembalikan) ke simpul K + 1.
4) Pindah ke K + 1 node.
5) Pergi ke langkah 1.
6) K – node ke-1 dari K node pertama menjadi head baru jika tersedia. Kalau tidak, kita bisa mengembalikan kepalanya.

```
struct ListNode * GetKPlusOneThNode(int K, struct ListNode *head) {
        struct ListNode *Kth;
        int i = 0;
        if(!head)
            return head;
        for (i = 0, Kth = head; Kth && (i < K); i++, Kth = Kth→next);
        if(i == K && Kth != NULL)
                return Kth;
        return head→next;
}
int HasKnodes(struct ListNode *head, int K) {
        int i =0;
        for(i = 0; head && (i < K); i++, head = head→next);
        if(i == K)
            return 1;
        return 0;
}
```

```
struct ListNode *ReverseBlockOfK-nodesInLinkedList(struct ListNode *head, int K) {
    struct ListNode *cur = head, *temp, *next, newHead;
    int i;
    if(K==0 || K==1)
        return head;
     if(HasKnodes(cur, K-1))
            newHead = GetKPlusOneThNode(K-1, cur);
     else
            newHead = head;
    while(cur && HasKnodes(cur, K)) {
             temp = GetKPlusOneThNode(K, cur);
             i=0;
             while(i < K) {
                next = cur→next;
                cur→next=temp;
                temp = cur;
                cur = next;
                i++;
             }
    }
    return newHead;
}
```

Soal-39 Apakah mungkin untuk mendapatkan waktu akses O(1) untuk Daftar Tertaut?

Solusi: Ya. Buat daftar tertaut dan pada saat yang sama simpan di tabel hash. Untuk n elemen, kita harus menyimpan semua elemen dalam tabel hash yang memberikan waktu pemrosesan awal O(n). Untuk membaca elemen apa pun, kita hanya memerlukan waktu konstan O(1) dan untuk membaca n elemen kita memerlukan n * 1 unit waktu = n satuan Oleh karena itu dengan menggunakan analisis diamortisasi kita dapat mengatakan bahwa akses elemen dapat dilakukan dalam waktu O(1).

Kompleksitas Waktu – O(1) [Diamortisasi].

Kompleksitas Ruang - O(n) untuk Tabel Hash.

Soal-40 Lingkaran Josephus: N orang telah memutuskan untuk memilih seorang pemimpin dengan mengatur diri mereka sendiri dalam sebuah lingkaran dan menghilangkan setiap orang ke-M di sekitar lingkaran, menutup barisan saat setiap orang keluar. Temukan orang mana yang akan menjadi orang terakhir yang tersisa (dengan peringkat 1).

Solusi: Asumsikan input adalah daftar tertaut melingkar dengan N node dan setiap node memiliki nomor (rentang 1 hingga N) yang terkait dengannya. Node kepala memiliki nomor 1 sebagai data.

```
struct ListNode  *GetJosephusPosition(){
        struct ListNode *p, *q;
        printf("Enter N (number of players): ");
        scanf("%d", &N);
        printf("Enter M (every M-th payer gets eliminated): ");
        scanf("%d", &M);
        // Create circular linked list containing all the players:
        p = q = malloc(sizeof(struct node));
        p→data = 1;
        for (int i = 2; i <= N; ++i) {
                p→next = malloc(sizeof(struct node));
                p = p→next;
                p→data = i;
        }
       // Close the circular linked list by having the last node point to the first.
        p→next = q;
        // Eliminate every M-th player as long as more than one player remains:
        for (int count = N; count > 1; --count) {
                for (int i = 0; i < M - 1; i++)
                        p = p→next;
                p→next = p→next→next;  // Remove the eliminated player from the circular linked list.
        }
        printf("Last player left standing (Josephus Position) is %d\n.", p→data);
}
```

Soal-41 Mengingat daftar tertaut terdiri dari data, penunjuk berikutnya dan juga penunjuk acak yang menunjuk ke simpul acak dari daftar. Berikan algoritma untuk mengkloning daftar.

Solusi: Kita dapat menggunakan tabel hash untuk mengasosiasikan node yang baru dibuat dengan instance node dalam daftar yang diberikan.

Algoritma:

- Pindai daftar asli dan untuk setiap simpul X, buat simpul baru Y dengan data X, lalu simpan pasangan (X, Y) dalam tabel hash menggunakan X sebagai kunci. Perhatikan bahwa selama pemindaian ini atur Y → berikutnya dan Y → acak ke NULL dan Kita akan memperbaikinya di pemindaian berikutnya.
- Sekarang untuk setiap node X dalam daftar asli kita memiliki salinan Y yang disimpan dalam tabel hash kita. Kita memindai daftar asli lagi dan mengatur pointer membangun daftar baru.

```
struct ListNode *Clone(struct ListNode *head){
    struct ListNode *X, *Y;
    struct HashTable *HT = CreateHashTable();
    X = head;
    while (X != NULL) {
        Y = (struct ListNode *)malloc(sizeof(struct ListNode *));
        Y→data = X→data;
        Y→next = NULL;
        Y→random = NULL;
        HT.insert(X, Y);
        X = X→next;
    }
    X = head;
    while (X != NULL) {
        // get the node Y corresponding to X from the hash table
        Y = HT.get(X);
        Y→next = HT.get(X→next);
        Y.setRandom = HT.get(X→random);
        X = X→next;
    }
    // Return the head of the new list, that is the Node Y
    return HT.get(head);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-42 Bisakah kita memecahkan Soal-41 tanpa ruang ekstra?

Solusi: Ya.

```
void Clone(struct ListNode *head){
    struct ListNode *temp, *temp2;
    //Step1: put temp→random in temp2→next,
    //so that we can reuse the temp→random field to point to temp2.
    temp = head;
    while (temp != NULL) {
      temp2 = (struct ListNode *)malloc(sizeof(struct ListNode *));
      temp2→data = temp→data;
      temp2→next = temp→random;
      temp→random = temp2;
      temp = temp→next;
    }
```

```
        //Step2: Setting temp2→random.  temp2→next is the old copy of the node that
        // temp2→random should point to, so temp→next→random is the new copy.
        temp = head;
        while (temp != NULL) {
          temp2 = temp→random;
          temp2→random = temp→next→random;
          temp = temp→next;
        }
        //Step3: Repair damage to old list and fill in next pointer in new list.
        temp = head;
        while (temp != NULL) {
        temp2 = temp→random;
        temp→random = temp2→next;
        temp2→next = temp→next→random;
        temp = temp→next;
        }
}
```

Kompleksitas Waktu: O(3n) O(n).
Kompleksitas Ruang: O(1).

Soal-43 Kita diberikan penunjuk ke sebuah simpul (bukan simpul ekor) dalam satu daftar tertaut. Hapus simpul itu dari daftar tertaut.

Solusi: Untuk menghapus sebuah simpul, kita harus menyesuaikan penunjuk berikutnya dari simpul sebelumnya untuk menunjuk ke simpul berikutnya, bukan yang sekarang. Karena kita tidak memiliki pointer ke node sebelumnya, kita tidak dapat mengarahkan pointer berikutnya. Jadi apa yang kita lakukan? Kita dapat dengan mudah lolos dengan memindahkan data dari node berikutnya ke node saat ini dan kemudian menghapus node berikutnya.

```
void deleteaNodeinLinkedList( struct ListNode * node ){
    struct ListNode * temp = node->next;
    node->data = node->next->data;
    node->next = temp->next;
    free(temp);
}
```

Kompleksitas Waktu: O(1).
Kompleksitas Ruang: O(1).

Soal-44 Diberikan daftar tertaut dengan angka genap dan ganjil, buatlah algoritma untuk membuat perubahan pada daftar sedemikian rupa sehingga semua angka genap muncul di awal.

Solusi: Untuk mengatasi masalah ini, kita dapat menggunakan logika pemisahan. Saat melintasi daftar, pisahkan daftar tertaut menjadi dua: satu berisi semua simpul genap dan yang lainnya berisi semua simpul ganjil. Sekarang, untuk

mendapatkan daftar akhir, kita cukup menambahkan daftar tertaut simpul ganjil setelah daftar tertaut simpul genap.

Untuk membagi daftar tertaut, telusuri daftar tertaut asli dan pindahkan semua simpul ganjil ke daftar tertaut terpisah dari semua simpul ganjil. Pada akhir loop, daftar asli akan memiliki semua simpul genap dan daftar simpul ganjil akan memiliki semua simpul ganjil. Untuk menjaga urutan semua node tetap sama, kita harus memasukkan semua node ganjil di akhir daftar node ganjil.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-45 Dalam sebuah linked list dengan n node, waktu yang dibutuhkan untuk menyisipkan sebuah elemen setelah sebuah elemen ditunjuk oleh beberapa pointer adalah

(A) O(1)
(B) O (masuk)
(C) O(n)
(D) O(nlogn)

Solusi **A**

Soal-46 Temukan simpul modular: Diberikan daftar tertaut tunggal, tulis fungsi untuk menemukan elemen terakhir dari awal yang n%k == 0, di mana n adalah jumlah elemen dalam daftar dan k adalah konstanta bilangan bulat. Misalnya, jika n = 19 dan k = 3 maka kita harus mengembalikan simpul ke-18.

Solusi: Untuk masalah ini nilai n tidak diketahui sebelumnya.

```
struct ListNode *modularNodeFromBegin(struct ListNode *head, int k){
    struct ListNode * modularNode;
    int i=0;
    if(k<=0)
        return NULL;
    for (;head != NULL; head = head→next){
        if(i%k == 0){
            modularNode = head;
        }
        i++;
    }
    return modularNode;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-47 Temukan simpul modular dari akhir: Diberikan daftar tertaut tunggal, tulis fungsi untuk menemukan simpul pertama dari akhir yang n%k == 0, di mana n

adalah jumlah elemen dalam daftar dan k adalah konstanta bilangan bulat . Jika n = 19 dan k = 3 maka kita harus mengembalikan simpul ke-16.

Solusi: Untuk masalah ini nilai n tidak diketahui sebelumnya dan sama dengan mencari elemen ke-k dari akhir linked list.

```
struct ListNode *modularNodeFromEnd(struct ListNode *head, int k){
    struct ListNode *modularNode=NULL;
    int i=0;
    if(k<=0)
        return NULL;
    for (i=0; i < k; i++){
        if(head)
            head = head→next;
        else
            return NULL;
    }
    while(head != NULL)
        modularNode = modularNode→next;
        head = head→next;
    }
    return modularNode;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Soal-48 Menemukan simpul pecahan: Diberikan sebuah daftar tertaut tunggal, tulis sebuah fungsi untuk menemukan elemen, di mana n adalah jumlah elemen dalam daftar.

Solusi: Untuk masalah ini nilai n tidak diketahui sebelumnya.

```
struct ListNode *fractionalNodes(struct ListNode *head, int k){
    struct ListNode *fractionalNode = NULL;
    int i=0;
    if(k<=0)
        return NULL;
    for (;head != NULL; head = head→next){
        if(i%k == 0){
            if(fractionalNode == NULL)
                fractionalNode = head;
            else fractionalNode = fractionalNode→next;
        }
        i++;
    }
    return fractionalNode;
}
```

```
    int i=0;
    if(k<=0)
        return NULL;
    for (;head != NULL; head = head→next){
        if(i%k == 0){
            if(fractionalNode == NULL)
                fractionalNode = head;
            else fractionalNode = fractionalNode→next;
        }
        i++;
    }
    return fractionalNode;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-49 Temukan simpul: Diberikan daftar tertaut tunggal, tulis fungsi untuk menemukan elemen, di mana n adalah jumlah elemen dalam daftar. Asumsikan nilai n tidak diketahui sebelumnya.

Solusi: Untuk masalah ini nilai n tidak diketahui sebelumnya.

```
struct ListNode *sqrtNode(struct ListNode *head){
    struct ListNode *sqrtN = NULL;
    int i=1, j=1;
    for (;head != NULL; head = head→next){
        if(i == j*j){
            if(sqrtN == NULL)
                sqrtN = head;
            else
                sqrtN = sqrtN→next;

            j++;
        }
        i++;
    }
    return sqrtN;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-50 Diberikan dua daftar Daftar 1 = {A1, A2, . . . , An) dan Daftar2 = {B1, B2, . . . , Bm} dengan data (keduanya daftar) dalam urutan menaik. Gabungkan mereka ke dalam daftar ketiga dalam urutan menaik sehingga daftar yang digabungkan menjadi:

$$\{A_1, B_1, A_2, B_2 \ldots\ldots A_m, B_m, A_{m+1} \ldots\ldots A_n\} \text{ if } n >= m$$
$$\{A_1, B_1, A_2, B_2 \ldots\ldots A_n, B_n, B_{n+1} \ldots\ldots B_m\} \text{ if } m >= n$$

Solusi:

```
struct ListNode*AlternateMerge(struct ListNode *List1, struct ListNode *List2){
    struct ListNode *newNode = (struct ListNode*) (malloc(sizeof(struct ListNode)));
    struct ListNode *temp;
    newNode→next = NULL;
    temp = newNode;
    while (List1!=NULL and List2!=NULL){
        temp→next = List1;
        temp = temp→next;
        List1 = List1→next;
        temp→next = List2;
        List2 = List2→next;
        temp = temp→next;
    }
    if (List1!=NULL)
        temp→next = List1;
    else
        temp→next = List2;
    temp = newNode→next;
    free(newNode);
    return temp;
}
```

Kompleksitas Waktu: Perulangan while membutuhkan waktu O(min(n,m)) karena akan berjalan selama min(n,m) kali. Langkah-langkah lainnya dijalankan di O(1). Oleh karena itu total kompleksitas waktu adalah O(min(n,m)).

Kompleksitas Ruang: O(1).

Soal-51 Median dalam deret bilangan bulat tak hingga

Solusi: Median adalah angka tengah dalam daftar angka yang diurutkan (jika kita memiliki jumlah elemen ganjil). Jika kita memiliki jumlah elemen genap, median adalah rata-rata dari dua angka tengah dalam daftar angka yang diurutkan. Kita dapat mengatasi masalah ini dengan daftar tertaut (dengan daftar tertaut yang diurutkan dan tidak disortir).

Pertama, mari kita coba dengan daftar tertaut yang tidak disortir. Dalam daftar tertaut yang tidak disortir, kita dapat menyisipkan elemen baik di kepala atau di ekor. Kerugian dengan pendekatan ini adalah bahwa mencari median membutuhkan O(n). Juga, operasi penyisipan membutuhkan O(1).

Sekarang, mari kita coba dengan daftar tertaut yang diurutkan. Kita dapat menemukan median dalam waktu O(1) jika kita melacak elemen tengah. Penyisipan ke lokasi tertentu juga O(1) dalam daftar tertaut mana pun. Namun, menemukan lokasi yang tepat untuk disisipkan bukanlah O(logn) seperti dalam array yang diurutkan, melainkan O(n) karena kita tidak dapat melakukan pencarian biner dalam daftar tertaut meskipun sudah diurutkan. Jadi, menggunakan daftar tertaut yang diurutkan tidak sepadan dengan usaha

karena penyisipan adalah O(n) dan menemukan median adalah O(1), sama dengan array yang diurutkan. Dalam array yang diurutkan, penyisipannya linier karena pergeseran, tetapi di sini linier karena kita tidak dapat melakukan pencarian biner dalam daftar tertaut.

Catatan: Untuk algoritma yang efisien, lihat bab Antrian dan Tumpukan Prioritas.

Soal-52 Diberikan daftar tertaut, bagaimana Anda memodifikasinya sehingga semua angka genap muncul sebelum semua angka ganjil dalam daftar tertaut yang dimodifikasi?

Solusi:

```
struct ListNode *exchangeEvenOddList(struct ListNode *head){
    // initializing the odd and even list headers
    struct ListNode *oddList = NULL, *evenList =NULL;
    // creating tail variables for both the list
    struct ListNode *oddListEnd = NULL, *evenListEnd = NULL;
    struct ListNode *itr=head;
    if( head == NULL ){
        return;
    }
    else{
        while( itr != NULL ){
            if( itr→data % 2 == 0 ){
              if( evenList == NULL ){
                  // first even node
                  evenList = evenListEnd = itr;
              }
```

```
            else{
                // inserting the node at the end of linked list
                evenListEnd→next = itr;
                evenListEnd = itr;
            }
        }
        else{
            if( oddList == NULL ){
                // first odd node
                oddList = oddListEnd = itr;
            }
            else{
                // inserting the node at the end of linked list
                oddListEnd→next = itr;
                oddListEnd = itr;
            }
        }
        itr = itr→next;
    }
    evenListEnd→next = oddList;
    return head;
  }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-53 Diberikan dua daftar tertaut, setiap simpul daftar dengan satu digit bilangan bulat, tambahkan dua daftar tertaut ini. Hasilnya harus disimpan dalam daftar tertaut ketiga. Perhatikan juga bahwa simpul kepala berisi digit angka yang paling signifikan.

Solusi: Karena penjumlahan bilangan bulat dimulai dari digit terkecil, pertama-tama kita harus mengunjungi simpul terakhir dari kedua daftar dan menjumlahkannya, membuat simpul baru untuk menyimpan hasilnya, mengurus carry jika ada, dan menautkan hasilnya simpul ke simpul yang akan ditambahkan ke simpul paling tidak signifikan kedua dan dilanjutkan.

Pertama-tama, kita perlu memperhitungkan perbedaan jumlah digit dalam dua angka. Jadi sebelum memulai rekursi, kita perlu melakukan beberapa perhitungan dan memindahkan penunjuk daftar yang lebih panjang ke tempat yang sesuai sehingga kita membutuhkan simpul terakhir dari kedua daftar secara bersamaan. Hal lain yang perlu kita jaga adalah membawa. Jika dua digit berjumlah lebih dari 10, kita perlu meneruskan carry ke node berikutnya dan menambahkannya. Jika penambahan digit paling signifikan menghasilkan carry, kita perlu membuat node tambahan untuk menyimpan carry.

Fungsi di bawah ini sebenarnya adalah fungsi pembungkus yang melakukan semua pembersihan seperti menghitung panjang daftar, memanggil implementasi rekursif, membuat simpul tambahan untuk dibawa dalam digit paling signifikan, dan menambahkan simpul yang tersisa di daftar yang lebih panjang.

```
void addListNumbersWrapper(struct ListNode *list1, struct ListNode *list2, int *carry, struct ListNode **result){
    int list1Length = 0, list2Length = 0, diff =0;
    struct ListNode  *current  = list1;
    while(current){
          current = current→next;
          list1Length++;
    }
    current = list2;
     while(current){
          current = current→next;
          list2Length++;
    }
     if(list1Length < list2Length){
          current = list1;
          list1 = list2;
          list2 = current;
    }
    diff  = abs(list1Length-list2Length);
    current = list1;
    while(diff--)
          current = current→next;
    addListNumbers(current, list2, carry, result);
    diff  = abs(list1Length-list2Length);
    addRemainingNumbers(list1, carry, result, diff);
    if(*carry){
          struct ListNode  * temp = (struct ListNode  *)malloc(sizeof(struct ListNode ));
          temp→next =  (*result);
          *result = temp;
    }
    return;
}
void addListNumbers(struct ListNode  *list1, struct ListNode  *list2, int *carry, struct ListNode  **result){
    int sum;
    if(!list1)
          return;
    addListNumbers(list1→next, list2→next, carry, result);
    //End of both lists, add them
    struct ListNode  * temp = (struct ListNode  *)malloc(sizeof(struct ListNode ));
    sum = list1→data + list2→data + (*carry);
    // Store carry
    *carry = sum/10;
    sum = sum%10;
    temp→data = sum;
    temp→next =  (*result);
     *result = temp;
    return;
}
void addRemainingNumbers(struct ListNode  * list1, int *carry, struct ListNode  **result, int diff){
    int sum =0;
     if(!list1  || diff == 0)
          return;
    addRemainingNumbers(list1→next, carry, result, diff-1);
    struct ListNode  * temp = (struct ListNode  *)malloc(sizeof(struct ListNode ));
    sum = list1->data + (*carry);
    *carry = sum/10;
    sum = sum%10;
    temp→data = sum;
    temp→next =  (*result);
    *result = temp;
    return;
}
```

Kompleksitas Waktu: O (maks (panjang Daftar1, panjang Daftar2)).
Kompleksitas Ruang: O(min(panjang List1, panjang List1)) untuk tumpukan rekursif.
Catatan: *Ini juga dapat diselesaikan menggunakan tumpukan.*

Soal-54 Algoritma pengurutan mana yang mudah beradaptasi dengan daftar tertaut tunggal?

Solusi: Pengurutan Penyisipan Sederhana mudah disesuaikan dengan daftar tertaut tunggal. Untuk menyisipkan elemen, daftar tertaut dilintasi sampai posisi yang tepat ditemukan, atau sampai akhir daftar tercapai. Itu dimasukkan ke dalam daftar hanya dengan menyesuaikan pointer tanpa menggeser elemen apa pun, tidak seperti dalam array. Ini mengurangi waktu yang diperlukan untuk penyisipan tetapi bukan waktu yang dibutuhkan untuk mencari posisi yang tepat.

Soal-55 Diberikan sebuah daftar, List1 = {A_1, A_2, . . . A_{n-1}; A_n) dengan data, atur ulang menjadi {A_1, A_n,A_2,A_{n-1}} tanpa menggunakan spasi tambahan.

Solusi: Temukan bagian tengah daftar tertaut. Kita bisa melakukannya dengan pendekatan pointer lambat dan cepat. Setelah menemukan simpul tengah, Kita membalikkan separuh kanan kemudian Kita melakukan penggabungan di tempat dari dua bagian dari daftar tertaut.

Soal-56 Diberikan dua daftar tertaut yang diurutkan, diberikan algoritma untuk mencetak elemen umum dari daftar tersebut.

Solusi: Solusinya didasarkan pada logika merge sort. Asumsikan dua daftar tertaut yang diberikan adalah: list1 dan list2. Karena elemen berada dalam urutan yang diurutkan, Kita menjalankan loop hingga Kita mencapai akhir dari salah satu daftar. Kita membandingkan nilai list1 dan list2. Jika nilainya sama, Kita menambahkannya ke daftar umum. Kita memindahkan list1/list2/kedua node ke depan ke pointer berikutnya jika nilai yang ditunjukkan oleh list1 kurang / lebih / sama dengan nilai yang ditunjukkan oleh list2.
Kompleksitas waktu O(m + n), di mana m adalah panjang list1 dan n adalah panjang list2. Kompleksitas Ruang: O(1).

```
struct ListNode  *commonElement(struct ListNode  *list1, struct ListNode  *list2) {
        struct ListNode  *temp =  (struct ListNode  *)malloc(sizeof(struct ListNode));
        struct ListNode  *head = temp;
        while (list1 != null && list2 != null) {
                if (list1→data == list2→data) {
                        head→next = new ListNode(list1→data); // Copy common element.
                        list1 = list1→next;
                        list2 = list2→next;
                        head = head→next;
                } else if (list1→data > list2→data) {
                        list2 = list2→next;
                } else { // list1→data < list2→data
                        list1 = list1→next;
                }
        }
        return temp.next;
}
```

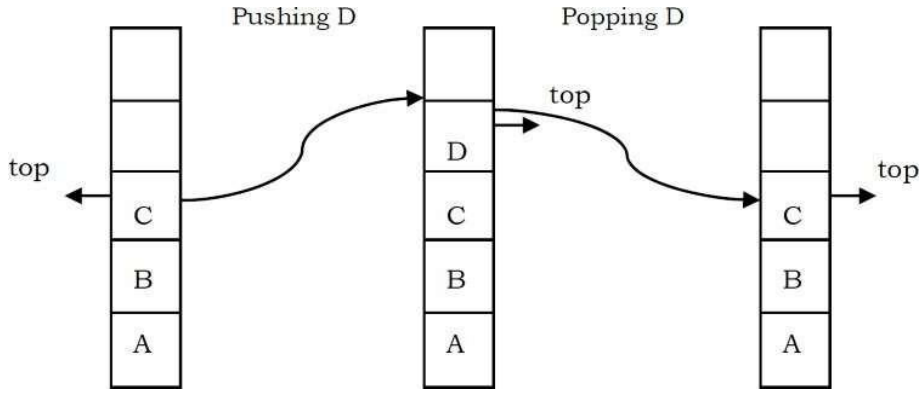
BAB 4
TUMPUKAN *(STACK)*

4.1 APA ITU TUMPUKAN?

Tumpukan (Stack) adalah struktur data sederhana yang digunakan untuk menyimpan data (mirip dengan Daftar Tertaut). Dalam tumpukan, urutan kedatangan data adalah penting. Tumpukan piring di kafetaria adalah contoh tumpukan yang baik. Pelat ditambahkan ke tumpukan saat dibersihkan dan diletakkan di atas. Ketika piring, diperlukan itu diambil dari atas tumpukan. Piring pertama yang ditempatkan di tumpukan adalah yang terakhir digunakan.

Definisi: Tumpukan adalah daftar terurut di mana penyisipan dan penghapusan dilakukan di satu ujung, yang disebut atas. Elemen terakhir yang dimasukkan adalah yang pertama dihapus. Oleh karena itu, ini disebut daftar Last in First out (LIFO) atau First in Last out (FILO).

Nama khusus diberikan untuk dua perubahan yang dapat dilakukan pada tumpukan. Ketika sebuah elemen dimasukkan ke dalam Tumpukan, konsepnya disebut push, dan ketika sebuah elemen dikeluarkan dari Tumpukan, konsepnya disebut pop. Mencoba mengeluarkan tumpukan kosong disebut underflow dan mencoba mendorong elemen dalam tumpukan penuh disebut overflow. Umumnya, Kita memperlakukan mereka sebagai pengecualian. Sebagai contoh, pertimbangkan snapshot dari tumpukan.



Gambar 4.1 pertimbangan snapshot dari tumpukan

4.2 BAGAIMANA TUMPUKAN DIGUNAKAN

Pertimbangkan hari kerja di kantor. Mari kita asumsikan pengembang sedang mengerjakan proyek jangka panjang. Manajer kemudian memberi pengembang tugas baru yang lebih penting. Pengembang mengesampingkan proyek jangka panjang dan mulai mengerjakan tugas baru. Telepon berdering, dan ini adalah prioritas tertinggi karena harus segera dijawab. Pengembang mendorong tugas ini ke baki yang tertunda dan menjawab telepon.

Ketika panggilan selesai, tugas yang ditinggalkan untuk menjawab telepon diambil dari baki yang tertunda dan pekerjaan berlangsung. Untuk menerima panggilan lain, mungkin harus ditangani dengan cara yang sama, tetapi pada akhirnya tugas baru akan selesai, dan pengembang dapat menggambar proyek jangka panjang dari baki yang tertunda dan melanjutkannya.

4.3 TUMPUKAN ADT

Operasi berikut membuat tumpukan menjadi ADT. Untuk kesederhanaan, asumsikan data adalah tipe integer.

Operasi tumpukan utama

- Push (int data): Menyisipkan data ke tumpukan.
- int Pop(): Menghapus dan mengembalikan elemen terakhir yang dimasukkan dari tumpukan.

Operasi tumpukan tambahan

- int Top(): Mengembalikan elemen yang terakhir dimasukkan tanpa menghapusnya.
- int Size(): Mengembalikan jumlah elemen yang disimpan dalam tumpukan.
- int IsEmptyTumpukan(): Menunjukkan apakah ada elemen yang disimpan dalam tumpukan atau tidak.
- int IsFullTumpukan(): Menunjukkan apakah tumpukan penuh atau tidak.

Pengecualian

Mencoba mengeksekusi suatu operasi terkadang dapat menyebabkan kondisi kesalahan, yang disebut pengecualian. Pengecualian dikatakan "dilempar" oleh operasi yang tidak dapat dieksekusi. Di Tumpukan ADT, operasi pop dan top tidak dapat dilakukan jika tumpukan kosong. Mencoba eksekusi pop (atas) pada tumpukan kosong akan memunculkan pengecualian. Mencoba mendorong elemen dalam tumpukan penuh akan menimbulkan pengecualian.

4.4 APLIKASI

Berikut ini adalah beberapa aplikasi di mana tumpukan memainkan peran penting.

Aplikasi langsung

- Keseimbangan simbol
- Konversi infix-to-postfix
- Evaluasi ekspresi postfix
- Menerapkan panggilan fungsi (termasuk rekursi)
- Menemukan rentang (menemukan rentang di pasar saham, lihat bagian Masalah)
- Riwayat kunjungan halaman di browser Web [Tombol Kembali]
- Urungkan urutan dalam editor teks
- Mencocokkan Tag dalam HTML dan XML

Aplikasi tidak langsung

- Struktur data bantu untuk algoritma lain (Contoh: algoritma traversal pohon)
- Komponen struktur data lain (Contoh: Simulasi antrian, lihat Antrian
- Bab)

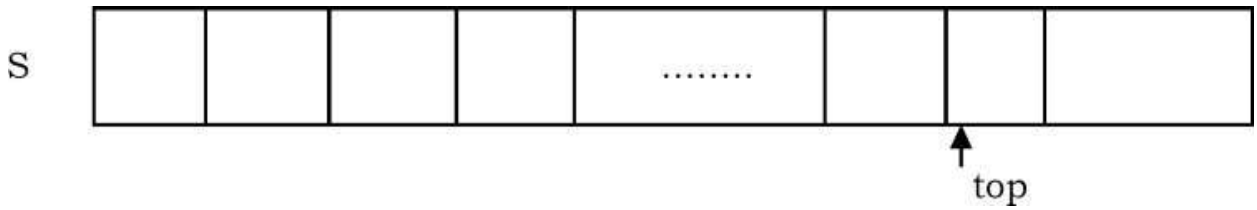
4.5 IMPLEMENTASI

Ada banyak cara untuk mengimplementasikan Tumpukan ADT; di bawah ini adalah metode yang umum digunakan.

- Implementasi berbasis array sederhana
- Implementasi berbasis array dinamis
- Implementasi daftar tertaut

Implementasi Array Sederhana

Implementasi Tumpukan ADT ini menggunakan array. Dalam array, kita menambahkan elemen dari kiri ke kanan dan menggunakan variabel untuk melacak indeks elemen teratas.



Gambar 4.2 implementasi array sederhana

Array yang menyimpan elemen Tumpukan mungkin menjadi penuh. Operasi push kemudian akan melempar pengecualian tumpukan penuh. Demikian pula, jika kita mencoba menghapus elemen dari tumpukan kosong, itu akan membuang pengecualian tumpukan kosong.

```
#define MAXSIZE 10
struct ArrayStack {
        int top;
        int capacity;
        int *array;
};
```

```
struct ArrayStack *CreateStack() {
        struct ArrayStack *S = malloc(sizeof(struct ArrayStack));
        if(!S)
           return NULL;
        S→capacity = MAXSIZE;
        S→top = -1;
        S→array= malloc(S→capacity * sizeof(int));
        if(!S→array)
           return NULL;
        return S;
}
int IsEmptyStack(struct ArrayStack *S) {
        return (S→top == -1);     // if the condition is true then 1 is returned else 0 is returned
}
int IsFullStack(struct ArrayStack *S){
        //if the condition is true then 1 is returned else 0 is returned
        return (S→top == S→capacity - 1);
}
void Push(struct ArrayStack *S, int data){
        /* S→top == capacity -1 indicates that the stack is full*/
        if(IsFullStack(S))
            printf( "Stack Overflow");
        else        /*Increasing the 'top' by 1 and storing the value at 'top' position*/
            S→ array[++S→top]= data;
}
int Pop(struct ArrayStack *S){
      /* S→top == - 1 indicates empty stack*/
        if(IsEmptyStack(S)){
           printf("Stack is Empty");
           return INT_MIN;;
        }
        else /* Removing element from  'top' of the array and reducing  'top' by 1*/
           return (S→ array[S→top--]);
}
void DeleteStack(struct DynArrayStack *S){
      if(S) {
         if(S→array)
            free(S→array);
         free(S);
      }
}
```

Kinerja & Keterbatasan

Kinerja

Misalkan n adalah jumlah elemen dalam tumpukan. Kompleksitas operasi tumpukan dengan representasi ini dapat diberikan sebagai:

Tabel 4.1 kompleksotas operasi tumpukan

Kompleksitas Ruang (untuk n operasi push)	O(*n*)
Kompleksitas Waktu Push()	O(1)
Kompleksitas Waktu Pop()	O(1)
Kompleksitas Waktu Ukuran()	O(1)
Kompleksitas Waktu dari Is Empty Tumpukan ()	O(1)

Kompleksitas Waktu Penuh Tumpukan f)	O(1)
Kompleksitas Waktu Hapus TumpukanQ	O(1)

Keterbatasan

Ukuran maksimum tumpukan harus ditentukan terlebih dahulu dan tidak dapat diubah. Mencoba mendorong elemen baru ke dalam tumpukan penuh menyebabkan pengecualian khusus implementasi.

Implementasi Array Dinamis

Pertama, mari kita pertimbangkan bagaimana kita mengimplementasikan tumpukan berbasis array sederhana. Kita mengambil satu variabel indeks teratas yang menunjuk ke indeks elemen yang paling baru dimasukkan ke dalam tumpukan. Untuk menyisipkan (atau mendorong) suatu elemen, kita menaikkan indeks teratas dan kemudian menempatkan elemen baru pada indeks itu.

Demikian pula, untuk menghapus (atau memunculkan) elemen, kita mengambil elemen di indeks teratas dan kemudian mengurangi indeks teratas. Kita mewakili antrian kosong dengan nilai teratas sama dengan -1. Masalah yang masih perlu diselesaikan adalah apa yang kita lakukan ketika semua slot dalam tumpukan array ukuran tetap terisi?

Percobaan pertama: Bagaimana jika kita menambah ukuran array sebesar 1 setiap kali tumpukan penuh?

- Dorongan(); meningkatkan ukuran S[] sebesar 1
- Pop(): memperkecil ukuran S[] sebanyak 1

Masalah dengan pendekatan ini?

Cara menambah ukuran array ini terlalu mahal. Mari kita lihat alasannya. Misalnya, pada n = 1, untuk mendorong elemen, buat array baru berukuran 2 dan salin semua elemen array lama ke array baru, dan pada akhirnya tambahkan elemen baru. Pada n = 2, untuk mendorong elemen, buat array baru berukuran 3 dan salin semua elemen array lama ke array baru, dan pada akhirnya tambahkan elemen baru.

Demikian pula, pada n = n – 1, jika kita ingin mendorong elemen, buat array baru berukuran n dan salin semua elemen array lama ke array baru dan pada akhirnya tambahkan elemen baru. Setelah n operasi push, total waktu T(n) (jumlah operasi penyalinan) sebanding dengan $1 + 2 + ... + n \approx O(n^2)$.

Pendekatan Alternatif: Penggandaan Berulang

Mari kita tingkatkan kompleksitasnya dengan menggunakan teknik penggandaan array. Jika array penuh, buat array baru dengan ukuran dua kali lipat, dan salin itemnya. Dengan pendekatan ini, mendorong n item membutuhkan waktu yang sebanding dengan n (bukan n^2).

Untuk mempermudah, mari kita asumsikan bahwa awalnya kita mulai dengan n = 1 dan naik ke n = 32. Artinya, kita melakukan penggandaan pada 1, 2,4,8,16. Cara lain untuk

menganalisis pendekatan yang sama adalah: pada n = 1, jika kita ingin menambahkan (push) sebuah elemen, gandakan ukuran larik saat ini dan salin semua elemen larik lama ke larik baru.

Pada n = 1, kita melakukan 1 operasi penyalinan, pada n = 2, kita melakukan 2 operasi penyalinan, dan pada n = 4, kita melakukan 4 operasi penyalinan dan seterusnya. Pada saat kita mencapai n = 32, jumlah total operasi penyalinan adalah 1+2 + 4 + 8+16 = 31 yang kira-kira sama dengan nilai 2n (32). Jika kita amati dengan seksama, kita sedang melakukan operasi penggandaan waktu logn. Sekarang, mari kita generalisasi diskusi. Untuk n operasi push, Kita menggandakan waktu logn ukuran array. Itu berarti, kita akan memiliki istilah logn dalam ekspresi di bawah ini. Total waktu T(n) dari serangkaian n operasi push sebanding dengan T(n) adalah O(n) dan waktu amortisasi dari operasi push adalah O(1) .

$$
\begin{aligned}
1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1 \\
&= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right) \\
&= n(2) \approx 2n = O(n)
\end{aligned}
$$

```
struct DynArrayStack {
    int top;
    int capacity;
    int *array;
};
struct DynArrayStack *CreateStack(){
    struct DynArrayStack *S = malloc(sizeof(struct DynArrayStack));
    if(!S)
        return NULL;
    S→capacity = 1;
    S→top = -1;
    S→array = malloc(S→capacity * sizeof(int));  // allocate an array of size 1 initially
    if(!S→array)
        return NULL;
    return S;
}
```

```
int IsFullStack(struct DynArrayStack *S){
    return (S→top == S→capacity-1);
}
void DoubleStack(struct DynArrayStack *S){
    S→capacity *= 2;
    S→array = realloc(S→array, S→capacity * sizeof(int));
}
void Push(struct DynArrayStack *S, int x){
    // No overflow in this implementation
     if(IsFullStack(S))
        DoubleStack(S);
    S→array[++S→top] = x;
}
int IsEmptyStack(struct DynArrayStack *S){
    return S→top == -1;
}
int Top(struct DynArrayStack *S){
    if(IsEmptyStack(S))
        return INT_MIN;
    return S→array[S→top];
}
int Pop(struct DynArrayStack *S){
    if(IsEmptyStack(S))
        return INT_MIN;
    return S→array[S→top--];
}
void DeleteStack(struct DynArrayStack *S){
    if(S) {
        if(S→array)
            free(S→array);
        free(S);
    }
}
```

Kinerja

Misalkan n adalah jumlah elemen dalam tumpukan. Kompleksitas untuk operasi dengan representasi ini dapat diberikan sebagai:

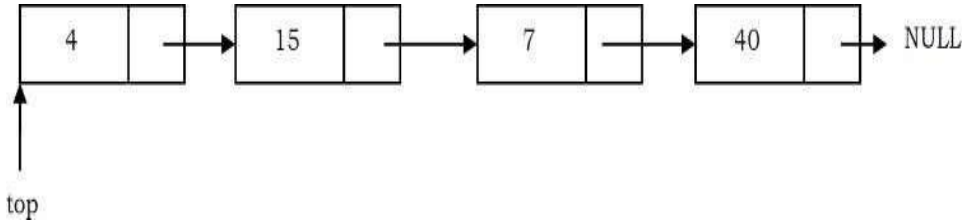
Tabel 4.2 kompleksitas untuk operasi

Kompleksitas Ruang (untuk n operasi push)	O(*n*)
Kompleksitas Waktu Buat Tumpukan()	O(1)
Kompleksitas Waktu PushQ	O(1) (Rata-rata)
Kompleksitas Waktu PopQ	O(1)
Kompleksitas Waktu Atas()	O(1)
Kompleksitas Waktu dari Is Empry Tumpukan f)	O(1))
Kompleksitas Waktu Full Tumpukan f)	O(1)

Kompleksitas Waktu Hapus Tumpukan Q	O(1)

Catatan: *Terlalu banyak penggandaan dapat menyebabkan pengecualian memory overflow.*

Implementasi Daftar Tertaut



Gambar 4.3 implementasi daftar tertaut

Cara lain untuk mengimplementasikan tumpukan adalah dengan menggunakan Daftar tertaut. Operasi push diimplementasikan dengan memasukkan elemen di awal daftar. Operasi pop diimplementasikan dengan menghapus node dari awal (*header/top node*).

```
struct ListNode{
    int data;
    struct ListNode *next;
};
struct Stack *CreateStack(){
    return NULL;
}
void Push(struct Stack **top, int data){
    struct Stack *temp;
    temp = malloc(sizeof(struct Stack));
    if(!temp)
        return NULL;
    temp→data = data;
    temp→next = *top;
    *top = temp;
}
int IsEmptyStack(struct Stack *top){
    return top == NULL;
}
int Pop(struct Stack **top){
    int data;
    struct Stack *temp;
    if(IsEmptyStack(top))
        return INT_MIN;
    temp = *top;
    *top = *top→next;
    data = temp→data;
    free(temp);
    return data;
}
```

```
int Top(struct Stack * top){
    if(IsEmptyStack(top))
        return INT_MIN;
    return top→next→data;
}
void DeleteStack(struct Stack **top){
    struct Stack *temp, *p;
    p = *top;
    while( p→next) {
        temp = p→next;
        p→next = temp→next;
        free(temp);
    }
    free(p);
}
```

Kinerja

Misalkan n adalah jumlah elemen dalam tumpukan. Kompleksitas untuk operasi dengan representasi ini dapat diberikan sebagai:

Tabel 4.3 kompleksitas untuk operasi

Kompleksitas Ruang (untuk n operasi push)	O(*n*)
Kompleksitas Waktu Buat Tumpukan()	O(1)
Kompleksitas Waktu Push()	O(1) (Rata-rata)
Kompleksitas Waktu Pop ()	O(1)
Kompleksitas Waktu Atas()	O(1)
Kompleksitas Waktu dari Is Empty Tumpukan()	O(1)
Kompleksitas Waktu Hapus Tumpukan()	O(*n*)

4.6 PERBANDINGAN IMPLEMENTASI

Membandingkan Strategi Inkremental dan Strategi Penggandaan

Kita membandingkan strategi inkremental dan strategi penggandaan dengan menganalisis total waktu T(n) yang dibutuhkan untuk melakukan serangkaian n operasi push. Kita mulai dengan tumpukan kosong yang diwakili oleh array ukuran 1.

Kita menyebut waktu diamortisasi dari operasi push adalah waktu rata-rata yang diambil oleh push selama serangkaian operasi, yaitu, T(n)/n.

Strategi Inkremental

Waktu diamortisasi (waktu rata-rata per operasi) dari operasi push adalah O(n) [O(n2)/n].

Strategi Menggandakan

Dalam metode ini, waktu diamortisasi dari operasi push adalah O(1) [O(n)/n].

Catatan: Untuk analisis, lihat bagian Implementasi.

Membandingkan Implementasi Array dan Implementasi Linked List

Implementasi Array

- Operasi membutuhkan waktu yang konstan.
- Operasi penggandaan yang mahal sesekali.
- Setiap urutan dari n operasi (dimulai dari tumpukan kosong) – "diamortisasi" terikat membutuhkan waktu yang sebanding dengan n.

Implementasi Daftar Tertaut

- Tumbuh dan menyusut dengan anggun.
- Setiap operasi membutuhkan waktu konstan O(1).
- Setiap operasi menggunakan ruang dan waktu ekstra untuk menangani referensi.

4.7 TUMPUKAN: MASALAH & SOLUSI

Soal-1 Diskusikan bagaimana tumpukan dapat digunakan untuk memeriksa keseimbangan simbol.

Solusi: Tumpukan dapat digunakan untuk memeriksa apakah ekspresi yang diberikan memiliki simbol yang seimbang. Algoritma ini sangat berguna dalam compiler. Setiap kali parser membaca satu karakter pada satu waktu. Jika karakter adalah pembatas pembuka seperti (, {, atau [- maka karakter tersebut ditulis ke tumpukan. Ketika pembatas penutup ditemukan seperti), }, atau] - tumpukan akan muncul.

Pembatas pembukaan dan penutupan kemudian dibandingkan. Jika cocok, penguraian string berlanjut. Jika tidak cocok, pengurai menunjukkan bahwa ada kesalahan pada saluran. Algoritma O(n) linier-waktu berdasarkan tumpukan dapat diberikan sebagai:

Algoritma:

a) Buat tumpukan.
b) while (akhir input tidak tercapai) {
    1) Jika karakter yang dibaca bukan simbol keseimbangan, abaikan saja.
    2) Jika karakter adalah simbol pembuka seperti (, [, {, dorong ke tumpukan
    3) Jika itu adalah simbol penutup seperti),],}, maka jika tumpukan kosong laporkan kesalahan. Jika tidak, pop tumpukan.
    4) Jika simbol yang muncul bukan simbol pembuka yang sesuai, laporkan kesalahan.
    }
c) Di akhir input, jika tumpukan tidak kosong laporkan kesalahan

Contoh:

Contoh	Valid	Deskripsi
(A+B)+(C-D)	Ya	Ekspresi memiliki simbol yang seimbang
((A+B)+(C-D)	Tidak	Satu kurung kurawal hilang
((A+B)+[C-D])	Ya	Kawat gigi pembuka dan penutup langsung sesuai
((A+B)+[C-D]}	Tidak	Kurung penutup terakhir tidak sesuai dengan kurung buka pertama

Untuk menelusuri algoritma, mari kita asumsikan bahwa inputnya adalah: () (() [()])

Simbol Masukan, A[i]	Operasi	Tumpukan	Keluaran
(	Dorongan (	(	
)	Pop (Uji apakah (dan A [i] cocok? YA		
(	Dorongan (	(	
(	Dorongan (	((	
)	Uji apakah (dan A [i] cocok? YA	(	
[	Dorongan [	([	
(	Dorongan (	([(	
)	Uji apakah (dan A [i] cocok? YA	([	
]	Uji apakah [dan A [i] cocok? YA	(	
)	Uji apakah (dan A [i] cocok? YA		
	Uji apakah tumpukan Kosong? YA		BENAR

Kompleksitas Waktu: O(n). Karena Kita memindai input hanya sekali.
Kompleksitas Ruang: O(n) [untuk tumpukan].

Soal-2 Diskusikan algoritma konversi infix ke postfix menggunakan stack.

Solusi: Sebelum membahas algoritma, pertama mari kita lihat definisi ekspresi infix, prefix dan postfix.

Infiks: Ekspresi infiks adalah satu huruf, atau operator, dilanjutkan dengan satu string infiks dan diikuti oleh string Infiks lainnya.

A
A+B
(A+B)+ (C-D)

Awalan: Ekspresi awalan adalah satu huruf, atau operator, diikuti oleh dua string awalan. Setiap string awalan yang lebih panjang dari satu variabel berisi operator, operan pertama, dan operan kedua.

A
+AB
++AB-CD

Postfix: Ekspresi postfix (juga disebut Reverse Polish Notation) adalah satu huruf atau operator, didahului oleh dua string postfix. Setiap string postfix yang lebih panjang dari satu variabel berisi operan pertama dan kedua diikuti oleh operator.

A
AB+
AB+CD-+

Pengertian prefiks dan postfiks adalah metode penulisan ekspresi matematika tanpa tanda kurung. Waktu untuk mengevaluasi ekspresi postfix dan prefix adalah O(n), di mana n adalah jumlah elemen dalam array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

Sekarang, mari kita fokus pada algoritma. Dalam ekspresi infiks, prioritas operator bersifat implisit kecuali jika kita menggunakan tanda kurung. Oleh karena itu, untuk algoritma konversi infix ke postfix kita harus mendefinisikan prioritas (atau prioritas) operator di dalam algoritma.

Tabel menunjukkan prioritas dan asosiatifitasnya (urutan evaluasi) antar operator.

Token	**Operator**	**Hak lebih tinggi**	**Keterkaitan**
( ) **[ ]** **→ .**	Fungsi memanggil struct elemen array atau anggota serikat	17	kiri ke kanan
-- ++	kenaikan, pengurangan	16	kiri ke kanan
-- ++ **!** **-** **- +** **&***	pengurangan, kenaikan logis bukan pelengkap unary minus atau plus alamat atau ukuran tipuan (dalam byte)	15	kanan ke kiri

Ukuran dari			
(tipe)	jenis pemeran	14	kanan ke kiri
***/%**	perkalian	13	kiri ke kanan
+ -	biner menambah atau mengurangi	12	kiri ke kanan
<< >>	menggeser	11	kiri ke kanan
> > = **< < =**	relasional	10	kiri ke kanan
= = ! =	persamaan	9	kiri ke kanan
&	sedikit demi sedikit dan	8	kiri ke kanan
^	bitwise eksklusif atau	7	kiri ke kanan
|	sedikit demi sedikit atau	6	kiri ke kanan
&&	logis dan	5	kiri ke kanan
| |	logis atau	4	kiri ke kanan
?:	bersyarat	3	kanan ke kiri
= += -= / = ***= %=** **<<= >>=** **&= ^=**	penugasan	2	kanan ke kiri
,	koma	1	kiri ke kanan

Properti Penting

- Mari kita perhatikan ekspresi infiks 2 + 3*4 dan ekuivalen postfixnya 234*+. Perhatikan bahwa antara infiks dan postfiks urutan angka (atau operan) tidak berubah. Ini adalah 2 3 4 dalam kedua kasus. Tetapi urutan operator * dan + dipengaruhi dalam dua ekspresi.
- Hanya satu tumpukan yang cukup untuk mengubah ekspresi infix menjadi ekspresi postfix. Tumpukan yang kita gunakan dalam algoritma akan digunakan untuk mengubah urutan operator dari infix ke postfix. Tumpukan yang kita gunakan hanya akan berisi operator dan simbol kurung buka ‘(‘.

Ekspresi postfix tidak mengandung tanda kurung. Kita tidak akan menampilkan tanda kurung dalam output postfix.

Algoritma:

a) Buat tumpukan

b) untuk setiap karakter t dalam aliran input}

c) pop dan keluarkan token sampai tumpukan kosong

```
if(t is an operand)
        output t
else if(t is a right parenthesis){
        Pop and output tokens until a left parenthesis is popped (but not output)
}
else // t is an operator or left parenthesis{
        pop and output tokens until one of lower priority than t is encountered or a left  parenthesis
        is encountered or the stack is empty
        Push t
}
}
```

Untuk pemahaman yang lebih baik mari kita telusuri sebuah contoh: A * B- (C + D) + E

Karakter Masukan	Operasi di Stack	Tumpukan	Ekspresi Postfix
A		Kosong	A
*****	Dorongan	*	A
B		*	AB
-	Periksa dan Dorong	-	AB*
(	Dorongan	-(	AB*
C		-(	AB*C
+	Periksa dan Dorong	-(+	AB*C
D			AB*CD
)	Pop dan tambahkan ke postfix sampai '('	-	AB*CD+
+	Periksa dan Dorong	+	AB*CD+-
E		+	AB*CD+-E
Akhir dari masukan	Pop sampai kosong		AB*CD+-E+

Soal-3 Diskusikan evaluasi postfix menggunakan stack?

Solusi:

Algoritma:

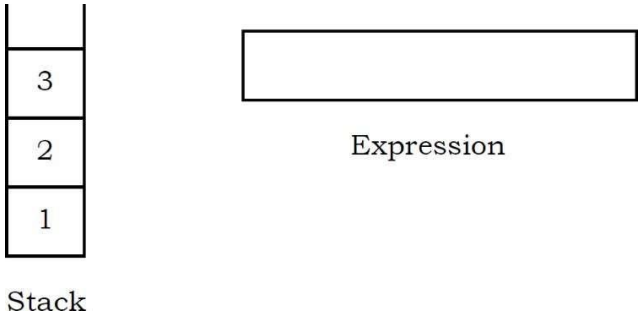
1. Pindai string Postfix dari kiri ke kanan.
2. Inisialisasi tumpukan kosong.
3. Ulangi langkah 4 dan 5 hingga semua karakter dipindai.
4. Jika karakter yang dipindai adalah operan, dorong ke tumpukan.
5. Jika karakter yang dipindai adalah operator, dan jika operator adalah operator unary, maka keluarkan elemen dari tumpukan. Jika operator adalah operator biner, maka

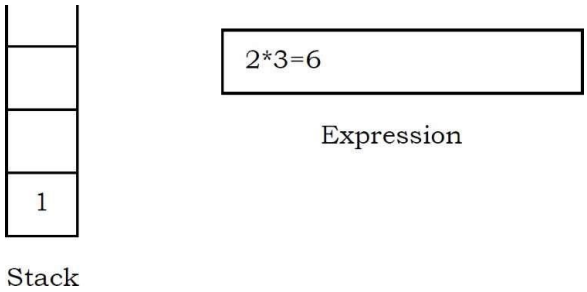
keluarkan dua elemen dari tumpukan. Setelah memunculkan elemen, terapkan operator ke elemen yang muncul. Biarkan hasil operasi ini menjadi retVal ke tumpukan.

6. Setelah semua karakter dipindai, kita hanya akan memiliki satu elemen di tumpukan.
7. Kembalikan bagian atas tumpukan sebagai hasilnya.

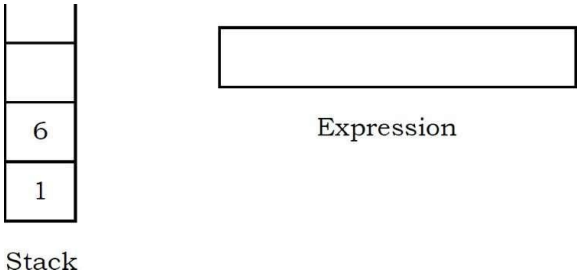
Contoh: Mari kita lihat bagaimana algoritma yang disebutkan di atas bekerja menggunakan sebuah contoh. Asumsikan bahwa string postfix adalah 123*+5-.

Awalnya tumpukan kosong. Sekarang, tiga karakter pertama yang dipindai adalah 1, 2 dan 3, yang merupakan operan. Mereka akan didorong ke dalam tumpukan dalam urutan itu.

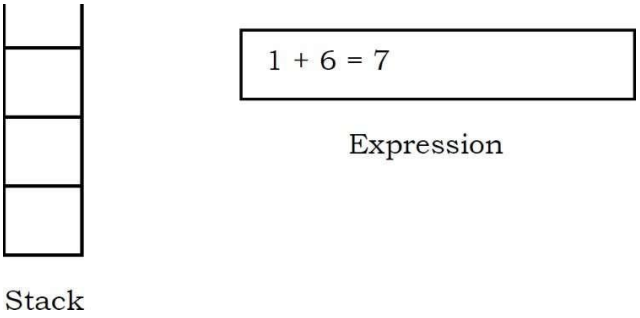




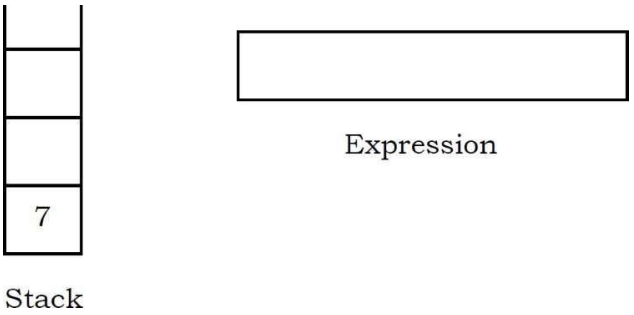
Karakter selanjutnya yang dipindai adalah “*”, yang merupakan operator. Jadi, Kita mengeluarkan dua elemen teratas dari tumpukan dan melakukan operasi "*" dengan dua operan. Operan kedua akan menjadi elemen pertama yang muncul.
Nilai ekspresi (2*3) yang telah dievaluasi (6) didorong ke dalam tumpukan.



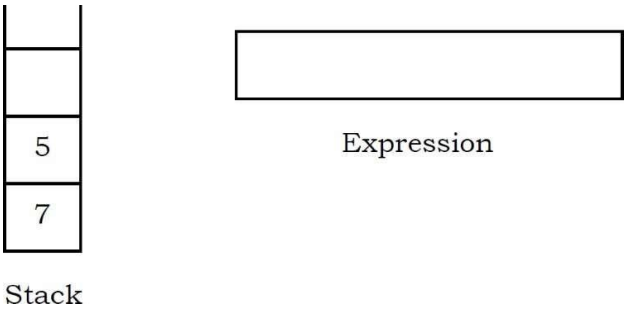
Karakter selanjutnya yang dipindai adalah “+”, yang merupakan operator. Jadi, Kita mengeluarkan dua elemen teratas dari tumpukan dan melakukan operasi "+" dengan dua operan. Operan kedua akan menjadi elemen pertama yang muncul.



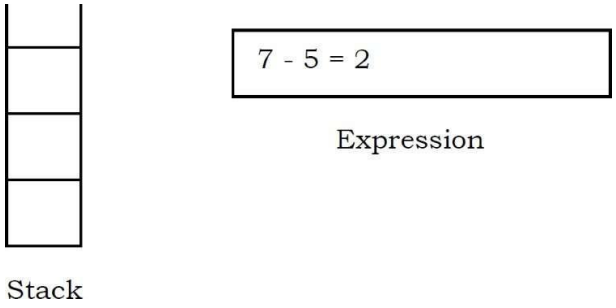
Nilai ekspresi (1+6) yang telah dievaluasi (7) didorong ke dalam tumpukan.



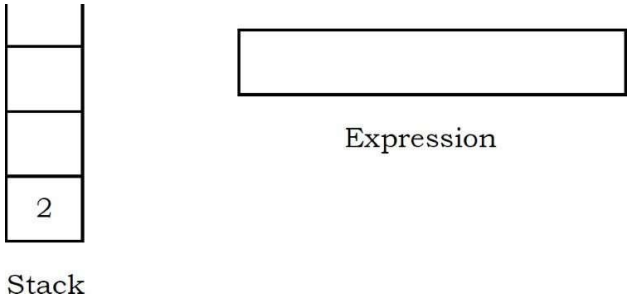
Karakter berikutnya yang dipindai adalah "5", yang ditambahkan ke tumpukan.



Karakter selanjutnya yang dipindai adalah “-”, yang merupakan operator. Jadi, Kita mengeluarkan dua elemen teratas dari tumpukan dan melakukan operasi "-" dengan dua operan. Operan kedua akan menjadi elemen pertama yang muncul.



Nilai ekspresi(7-5) yang telah dievaluasi(23) didorong ke dalam tumpukan.



Sekarang, karena semua karakter dipindai, elemen yang tersisa di tumpukan (hanya akan ada satu elemen di tumpukan) akan dikembalikan. Hasil akhir:

- String Postfix : 123*+5-
- Hasil : 2

Soal-4 Bisakah kita mengevaluasi ekspresi infiks dengan tumpukan dalam satu lintasan?

Solusi: Dengan menggunakan 2 tumpukan, kita dapat mengevaluasi ekspresi infix dalam 1 pass tanpa mengonversi ke postfix.

Algoritma:

1) Buat tumpukan operator kosong
2) Buat tumpukan operan kosong
3) Untuk setiap token dalam string input
   A. Dapatkan token berikutnya di string infix
   B. Jika token berikutnya adalah operan, letakkan di tumpukan operan
   C. Jika token berikutnya adalah operator
      i. Evaluasi operator (operasi berikutnya)
4) Selama tumpukan operator tidak kosong, pop operator dan operan (kiri dan kanan), evaluasi kiri kanan operator dan dorong hasilnya ke tumpukan operan
5) Hasil pop dari tumpukan operator

Soal-5 Bagaimana merancang tumpukan sedemikian rupa sehingga GetMinimum() harus O(1)?

Solusi: Ambil tumpukan tambahan yang mempertahankan minimum semua nilai dalam tumpukan. Juga, asumsikan bahwa setiap elemen tumpukan kurang dari elemen di bawahnya. Untuk mempermudah, mari kita panggil tumpukan tambahan min stack.

Saat Kita mengeluarkan tumpukan utama, pop juga tumpukan min. Saat Kita mendorong tumpukan utama, dorong elemen baru atau minimum saat ini, mana yang lebih rendah. Kapan saja, jika kita ingin mendapatkan minimum, maka kita hanya perlu mengembalikan elemen teratas dari tumpukan min. Mari

kita ambil contoh dan menelusurinya. Awalnya mari kita asumsikan bahwa kita telah mendorong 2, 6, 4, 1 dan 5. Berdasarkan algoritma yang disebutkan di atas, tumpukan min akan terlihat seperti:

Tumpukan utama	**Tumpukan minimal**
5 → atas	1 → atas
1	1
4	2
6	2
2	2

Setelah muncul dua kali kita mendapatkan:

Tumpukan utama	**Tumpukan minimal**
4 -→ atas	2 → atas
6	2
2	2

Berdasarkan pembahasan di atas, sekarang mari kita mengkodekan operasi push, pop dan GetMinimum().

```
struct AdvancedStack{
    struct Stack elementStack;
    struct Stack minStack;
};
void Push(struct AdvancedStack *S, int data ){
    Push (S→elementStack, data);
    if(IsEmptyStack(S→minStack) || Top(S→minStack) >= data)
        Push (S→minStack, data);
    else Push (S→minStack, Top(S→minStack));
}
int Pop(struct AdvancedStack *S ){
    int temp;
    if(IsEmptyStack(S→elementStack))
        return -1;
    temp = Pop (S→elementStack);
    Pop (S→minStack);
    return temp;
}
int GetMinimum(struct AdvancedStack *S){
    return Top(S→minStack);
}
struct AdvancedStack *CreateAdvancedStack(){
    struct AdvancedStack *S = (struct AdvancedStack *)malloc(sizeof(struct AdvancedStack));
    if(!S)
        return NULL;
    S→elementStack = CreateStack();
    S→minStack = CreateStack();
    return S;
}
```

Kompleksitas waktu: O(1).

Kompleksitas ruang: O(n) [untuk tumpukan Min]. Algoritma ini memiliki penggunaan ruang yang jauh lebih baik jika kita jarang mendapatkan "minimal baru atau sama".

Soal-6 Untuk Soal-5 apakah mungkin untuk meningkatkan kompleksitas ruang?

Solusi **Ya.** Masalah utama dari pendekatan sebelumnya adalah, untuk setiap operasi push Kita juga mendorong elemen ke tumpukan min (baik elemen baru atau elemen minimum yang ada). Itu berarti, Kita mendorong elemen minimum duplikat ke tumpukan.

Sekarang, mari kita ubah algoritma untuk meningkatkan kompleksitas ruang. Kita masih memiliki tumpukan min, tetapi Kita hanya muncul darinya ketika nilai yang Kita keluarkan dari tumpukan utama sama dengan yang ada di tumpukan min. Kita hanya mendorong ke tumpukan min ketika nilai yang didorong ke tumpukan utama kurang dari atau sama dengan nilai min saat ini. Dalam algoritma yang dimodifikasi ini juga, jika kita ingin mendapatkan

minimum maka kita hanya perlu mengembalikan elemen teratas dari tumpukan min. Misalnya, mengambil versi asli dan mendorong 1 lagi, Kita akan mendapatkan:

Tumpukan utama	Tumpukan minimal
1 → atas	
5	
1	
4	1 → atas
6	1
2	2

Muncul dari atas muncul dari kedua tumpukan karena 1 == 1, meninggalkan:

Tumpukan utama	tumpukan minimal
5 → atas	
1	
4	
6	1 →atas
2	2

Muncul lagi hanya muncul dari tumpukan utama, karena 5 > 1:

Tumpukan utama	Tumpukan minimal
1 → atas	
4	
6	1 → atas
2	2

Muncul lagi muncul kedua tumpukan karena 1 == 1:

Tumpukan utama	tumpukan minimal
4 → atas	
6	
2	2 → atas

Catatan: Perbedaannya hanya pada operasi push & pop.

```
struct AdvancedStack {
    struct Stack elementStack;
    struct Stack minStack;
};
void Push(struct AdvancedStack *S, int data){
    Push (S→elementStack, data);
    if(IsEmptyStack(S→minStack) || Top(S→minStack) >= data)
        Push (S→minStack, data);
}
int Pop(struct AdvancedStack *S ){
    int temp;
    if(IsEmptyStack(S→elementStack))
        return -1;
    temp = Top (S→elementStack);
    if(Top(S→ minStack) == Pop(S→elementStack))
        Pop (S→ minStack);
    return temp;
}
int GetMinimum(struct AdvancedStack *S){
    return Top(S→minStack);
}
struct AdvancedStack * AdvancedStack(){
    struct AdvancedStack *S = (struct AdvancedStack) malloc (sizeof (struct AdvancedStack));
    if(!S)
        return NULL;
    S→elementStack = CreateStack();
    S→minStack = CreateStack();
    return S;
}
```

Kompleksitas waktu: O(1).

Kompleksitas ruang: O(n) [untuk tumpukan Min]. Tetapi algoritma ini memiliki penggunaan ruang yang jauh lebih baik jika kita jarang mendapatkan "minimal baru atau sama".

Soal-7 Untuk larik tertentu dengan n simbol, berapa banyak permutasi tumpukan yang mungkin?

Solusi Jumlah permutasi tumpukan dengan n simbol diwakili oleh bilangan Catalan dan kita akan membahasnya dalam bab Pemrograman Dinamis.

Soal-8 Diberikan array karakter yang dibentuk dengan a dan b. String ditandai dengan karakter khusus X yang mewakili bagian tengah daftar (misalnya: ababa...ababXbabab baaa). Periksa apakah string palindrom.

Solusi: Ini adalah salah satu algoritma paling sederhana. Apa yang kita lakukan adalah, memulai dua indeks, satu di awal string dan yang lainnya di akhir string. Setiap kali membandingkan apakah nilai pada kedua indeks tersebut sama atau tidak.

Jika nilainya tidak sama maka kita katakan bahwa string yang diberikan bukan palindrom.

Jika nilainya sama maka naikkan indeks kiri dan kurangi indeks kanan. Lanjutkan proses ini sampai kedua indeks bertemu di tengah (di X) atau jika string bukan palindrom.

```
int IsPalindrome(char *A){
    int i=0, j = strlen(A)-1;
    while(i < j && A[i] == A[j]) {
        i++;
        j--;
    }
    if(i < j ) {
        printf("Not a Palindrome");
        return 0;
    }
    else {
        printf("Palindrome");
        return 1;
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Soal-9 Untuk Soal-8, jika input dalam single linked list lalu bagaimana kita memeriksa apakah elemen-elemen daftar membentuk palindrom (Artinya, mundur tidak mungkin).

Solusi Lihat bab Daftar Tertaut.

Soal-10 Bisakah kita menyelesaikan Soal-8 menggunakan tumpukan?

Solusi: Ya.

Algoritma:

- Telusuri daftar sampai kita menemukan X sebagai elemen input.
- Selama traversal, tekan semua elemen (sampai X) ke stack.
- Untuk paruh kedua daftar, bandingkan konten setiap elemen dengan bagian atas tumpukan. Jika mereka sama maka pop stack dan pergi ke elemen berikutnya dalam daftar input.
- Jika tidak sama maka string yang diberikan bukan palindrom.

```
int IsPalindrome(char *A){
    int i=0;
    struct Stack S= CreateStack();
    while(A[i] != 'X') {
        Push(S, A[i]);
        i++;
    }
    i++;
    while(A[i]) {
        if(IsEmptyStack(S) ||A[i] != Pop(S)) {
            printf("Not a Palindrome");
            return 0;
        }
        i++;
    }
    return IsEmptyStack(S);
}
```

Lanjutkan proses ini sampai tumpukan kosong atau string bukan palindrom.
Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n/2) O(n).

Soal-11 Mengingat tumpukan, bagaimana cara membalikkan elemen tumpukan hanya dengan menggunakan operasi tumpukan (Push & pop)?

Solusi:

Algoritma:

- Pertama pop semua elemen tumpukan sampai menjadi kosong.
- Untuk setiap langkah ke atas dalam rekursi, masukkan elemen di bagian bawah tumpukan.

```
void ReverseStack(struct Stack *S){
    int data;
    if(IsEmptyStack(S))
       return;
    data = Pop(S);
    ReverseStack(S);
    InsertAtBottom(S, data);
}
void InsertAtBottom(struct Stack *S, int data){
    int temp;
    if(IsEmptyStack(S)) {
        Push(S, data);
        return;
    }
    temp = Pop(S);
    InsertAtBottom(S, data);
    Push(S, temp);
}
```

Kompleksitas Waktu: O(n2).
Kompleksitas Ruang: O(n), untuk tumpukan rekursif.

Soal-12 Tunjukkan bagaimana menerapkan satu antrian secara efisien menggunakan dua tumpukan. Menganalisis waktu berjalan dari operasi antrian.

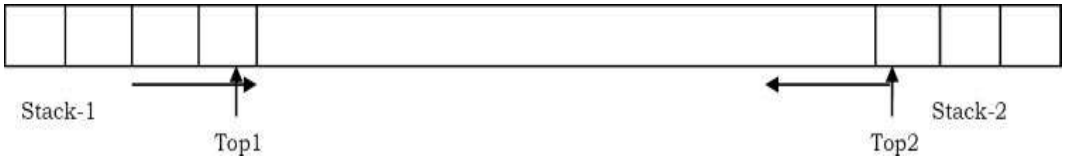
Solusi Lihat bab Antrian.

Soal-13 Tunjukkan bagaimana menerapkan satu tumpukan secara efisien menggunakan dua antrian. Analisis waktu berjalan operasi tumpukan.

Solusi Lihat bab Antrian.

Soal-14 Bagaimana kita mengimplementasikan dua tumpukan hanya dengan menggunakan satu larik? Rutinitas tumpukan Kita seharusnya tidak menunjukkan pengecualian kecuali setiap slot dalam array digunakan?

Solusi:



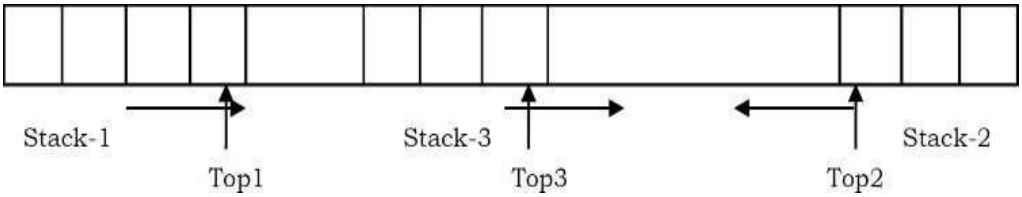
Algoritma:

- Mulai dua indeks satu di ujung kiri dan yang lainnya di ujung kanan.
- Indeks kiri mensimulasikan tumpukan pertama dan indeks kanan mensimulasikan tumpukan kedua.
- Jika kita ingin memasukkan sebuah elemen ke dalam tumpukan pertama maka letakkan elemen tersebut di indeks kiri.
- Demikian pula, jika kita ingin memasukkan sebuah elemen ke dalam tumpukan kedua, maka letakkan elemen tersebut pada indeks yang tepat.
- Tumpukan pertama tumbuh ke kanan, dan tumpukan kedua tumbuh ke kiri.

Kompleksitas waktu push dan pop untuk kedua tumpukan adalah O(1).
Kompleksitas Ruang adalah O(1).

Soal-15 3 tumpukan dalam satu larik: Bagaimana cara menerapkan 3 tumpukan dalam satu larik?

Solusi: Untuk masalah ini, mungkin ada cara lain untuk menyelesaikannya. Diberikan di bawah ini adalah satu kemungkinan dan berfungsi selama ada ruang kosong dalam array.



Untuk menerapkan 3 tumpukan, Kita menyimpan informasi berikut.

- Indeks tumpukan pertama (Topi): ini menunjukkan ukuran tumpukan pertama.

- Indeks tumpukan kedua (Top2): ini menunjukkan ukuran tumpukan kedua.
- Indeks awal tumpukan ketiga (alamat dasar tumpukan ketiga).
- Indeks teratas dari tumpukan ketiga.

Sekarang, mari kita definisikan operasi push dan pop untuk implementasi ini.

Mendorong:

- Untuk mendorong ke tumpukan pertama, kita perlu melihat apakah penambahan elemen baru menyebabkannya menabrak tumpukan ketiga. Jika demikian, coba geser tumpukan ketiga ke atas. Masukkan yang baru elemen di (start1 + Top1).
- Untuk mendorong ke tumpukan kedua, kita perlu melihat apakah menambahkan elemen baru menyebabkannya menabrak tumpukan ketiga. Jika demikian, coba geser tumpukan ketiga ke bawah. Masukkan elemen baru di (start2 - Top2).
- Saat mendorong ke tumpukan ketiga, lihat apakah menabrak tumpukan kedua. Jika demikian, coba geser tumpukan ketiga ke bawah dan coba dorong lagi. Masukkan elemen baru di (start3 + Top3).

Kompleksitas Waktu: O(n). Karena kita mungkin perlu menyesuaikan tumpukan ketiga. Kompleksitas Ruang: O(1). Popping: Untuk popping, kita tidak perlu menggeser, cukup kurangi ukuran tumpukan yang sesuai. Kompleksitas Waktu: O(1). Kompleksitas Ruang: O(1).

Soal-16 Untuk Soal-15, apakah ada cara lain untuk mengimplementasikan middle stack?

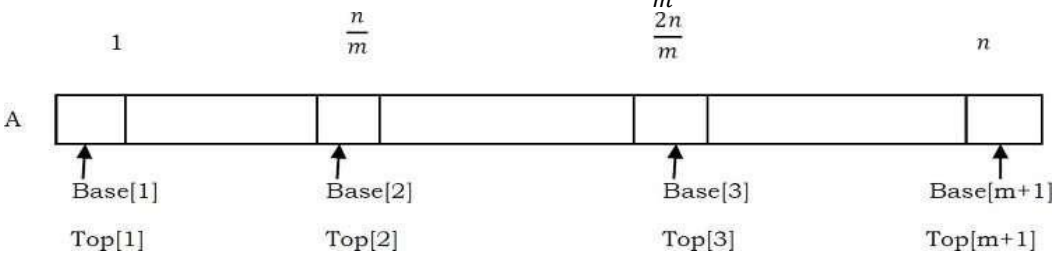
Solusi: Ya. Ketika tumpukan kiri (yang tumbuh ke kanan) atau tumpukan kanan (yang tumbuh ke kiri) menabrak tumpukan tengah, kita perlu menggeser seluruh tumpukan tengah untuk memberi ruang. Hal yang sama terjadi jika dorongan pada tumpukan tengah menyebabkannya menabrak tumpukan kanan.

Untuk mengatasi masalah yang disebutkan di atas (jumlah shift) yang dapat kita lakukan adalah: dorongan bolak-balik dapat ditambahkan di sisi bergantian dari daftar tengah (Misalnya, elemen genap didorong ke kiri, elemen ganjil didorong ke kanan) . Ini akan menjaga tumpukan tengah seimbang di tengah larik tetapi masih perlu digeser ketika menabrak tumpukan kiri atau kanan, baik dengan tumbuh sendiri atau dengan pertumbuhan tumpukan tetangga.

Kita dapat mengoptimalkan lokasi awal dari tiga tumpukan jika mereka tumbuh/menyusut pada tingkat yang berbeda dan jika mereka memiliki ukuran rata-rata yang berbeda. Misalnya, satu tumpukan tidak banyak berubah. Jika kita meletakkannya di sebelah kiri, maka tumpukan tengah pada akhirnya akan terdorong ke arahnya dan meninggalkan celah antara tumpukan tengah dan kanan, yang tumbuh ke arah satu sama lain. Jika mereka bertabrakan, maka kemungkinan kita kehabisan ruang dalam array. Tidak ada perubahan dalam kompleksitas waktu tetapi jumlah rata-rata shift akan berkurang.

Soal-17 Beberapa (m) tumpukan dalam satu larik: Mirip dengan Soal-15, bagaimana jika kita ingin mengimplementasikan m tumpukan dalam satu larik?

Solusi: Mari kita asumsikan bahwa indeks array adalah dari 1 hingga n. Serupa dengan pembahasan pada Soal-15, untuk mengimplementasikan m tumpukan dalam satu larik, kita membagi larik menjadi m bagian (seperti yang ditunjukkan di bawah). NS ukuran masing-masing bagian adalah $\frac{n}{m}$.



Dari representasi di atas kita dapat melihat bahwa, tumpukan pertama dimulai pada indeks 1 (indeks awal disimpan di Base[l]), tumpukan kedua dimulai pada indeks $\frac{n}{m}$ (indeks awal disimpan di Basis [2]), tumpukan ketiga dimulai at index $\frac{2n}{m}$ (indeks awal disimpan di Base[3]), dan seterusnya. Mirip dengan Base array, mari kita asumsikan bahwa Top array menyimpan indeks teratas untuk setiap stack. Pertimbangkan terminologi berikut untuk diskusi.

- Top[i], untuk $1 \leq i \leq m$ akan menunjuk ke elemen paling atas dari stack i.
- Jika Base[i] == Top[i], maka kita dapat mengatakan bahwa stack i kosong.
- Jika Top[i] == Base[i+1], maka kita dapat mengatakan bahwa stack i sudah penuh. Awalnya Basis[i] = Atas[i] = $\frac{n}{m}$ (i – 1), untuk $1 \leq i \leq m$.
- Tumpukan ke-i bertambah dari Basis[i]+1 ke Basis[i+1].

Mendorong ke tumpukan ke-i:

1) Untuk mendorong ke tumpukan ke-i, Kita memeriksa apakah bagian atas tumpukan ke-i mengarah ke Basis[i+1] (kasus ini menentukan bahwa tumpukan ke-i sudah penuh). Itu berarti, kita perlu melihat apakah menambahkan elemen baru menyebabkannya menabrak tumpukan ke-i + 1. Jika demikian, cobalah untuk menggeser tumpukan dari tumpukan ke-i + ke-1 ke tumpukan ke-m ke arah kanan. Masukkan elemen baru di (Base[i] + Top[i]).
2) Jika pemindahan ke kanan tidak memungkinkan, coba geser tumpukan dari tumpukan 1 ke i -1 ke kiri.
3) Jika keduanya tidak memungkinkan maka kita dapat mengatakan bahwa semua tumpukan penuh.

```
void Push(int StackID, int data) {
        if(Top[i] == Base[i+1])
            Print i^th Stack is full and does the necessary action (shifting);
        Top[i] = Top[i]+1;
        A[Top[i]] = data;
}
```

Kompleksitas Waktu: O(n). Karena kita mungkin perlu menyesuaikan tumpukan.
Kompleksitas Ruang: O(1).

Popping from ith stack: Untuk popping, kita tidak perlu menggeser, cukup kurangi ukuran stack yang sesuai. Satu-satunya kasing yang harus diperiksa adalah tumpukan kasing kosong.

```
int Pop(int StackID) {
        if(Top[i] == Base[i])
            Print i^th Stack is empty;
        return  A[Top[i]--];
}
```

Kompleksitas Waktu: O(1).
Kompleksitas Ruang: O(1).

Soal-18 Pertimbangkan setumpuk bilangan bulat kosong. Biarkan angka 1,2,3,4,5,6 didorong ke tumpukan ini dalam urutan kemunculannya dari kiri ke kanan. Biarkan 5 menunjukkan push dan X menunjukkan operasi pop. Bisakah mereka diubah menjadi urutan 325641 (keluaran) dan urutan 154623?

Solusi: Keluaran SSSXXSSXSXXX 325641. 154623 tidak dapat dikeluarkan karena 2 didorong jauh sebelum 3 sehingga hanya dapat muncul setelah 3 dikeluarkan.

Soal-19 Sebelumnya dalam bab ini, kita telah membahas bahwa untuk implementasi array dinamis dari tumpukan, pendekatan 'penggandaan berulang' digunakan. Untuk masalah yang sama, apa kerumitannya jika kita membuat array baru yang ukurannya n + jika bukan menggandakan?

Solusi: Mari kita asumsikan bahwa ukuran tumpukan awal adalah 0. Untuk mempermudah, mari kita asumsikan bahwa K = 10. Untuk menyisipkan elemen, kita membuat array baru yang ukurannya 0 + 10 = 10. Demikian pula, setelah 10 elemen kita kembali membuat array baru yang ukurannya 10 + 10 = 20 dan proses ini berlanjut pada nilai: 30,40 ... Itu berarti, untuk nilai n yang diberikan, Kita membuat array baru di: $\frac{n}{10}, \frac{n}{20}, \frac{n}{30}, \frac{n}{40} \cdots$ Jumlah total operasi penyalinan adalah:

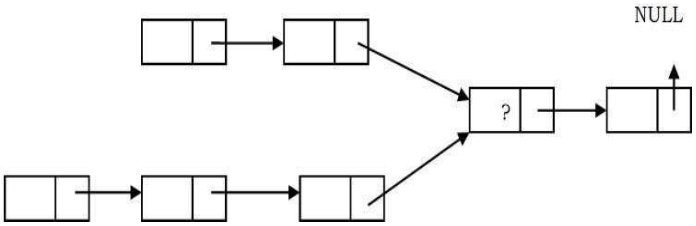
$$= \frac{n}{10} + \frac{n}{20} + \frac{n}{30} + \cdots 1 = \frac{n}{10}\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n}\right) = \frac{n}{10} logn \approx O(nlogn)$$

Jika kita melakukan n operasi push, biaya per operasi adalah O(logn).

Soal-20 Diberikan string yang berisi n S dan n X di mana 5 menunjukkan operasi push dan X menunjukkan operasi pop, dan dengan tumpukan awalnya kosong, rumuskan aturan untuk memeriksa apakah string 5 operasi yang diberikan dapat diterima atau tidak?

Solusi: Diberikan string dengan panjang 2n, Kita ingin memeriksa apakah string operasi yang diberikan diizinkan atau tidak sehubungan dengan fungsinya pada stack. Satu-satunya operasi yang dibatasi adalah pop yang persyaratan sebelumnya adalah bahwa tumpukan tidak boleh kosong. Jadi saat melintasi string dari kiri ke kanan, sebelum pop apa pun, tumpukan tidak boleh kosong, yang berarti jumlah S selalu lebih besar atau sama dengan jumlah X. Oleh karena itu kondisinya pada setiap tahap pemrosesan string, jumlah operasi push (S) harus lebih besar dari jumlah operasi pop (X).

Soal-21 Misalkan ada dua daftar tertaut tunggal yang berpotongan di beberapa titik dan menjadi satu daftar tertaut. Pointer kepala atau awal dari kedua daftar diketahui, tetapi simpul yang berpotongan tidak diketahui. Juga, jumlah node di setiap daftar sebelum mereka berpotongan tidak diketahui dan kedua daftar mungkin memiliki nomor yang berbeda. List1 mungkin memiliki n node sebelum mencapai titik persimpangan dan List2 mungkin memiliki m node sebelum mencapai titik persimpangan di mana m dan n mungkin m = n,m < n atau m > n. Bisakah kita menemukan titik penggabungan menggunakan tumpukan?

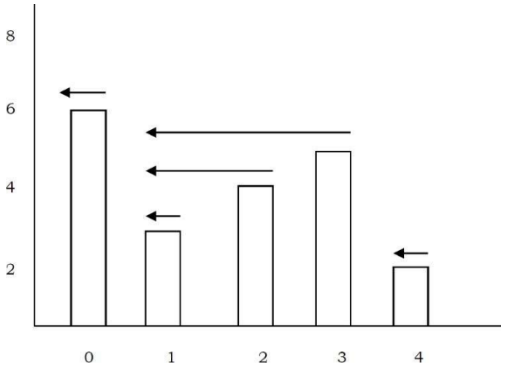


Solusi: Ya. Untuk algoritma lihat bab Linked Lists.

Soal-22 Menemukan Rentang: Diberikan sebuah larik A, rentang S[i] dari A[i] adalah jumlah maksimum elemen berurutan A[j] tepat sebelum A[i] dan sedemikian sehingga A[j] < A[i]?

Cara bertanya lain: Diberikan sebuah array A bilangan bulat, temukan maksimum j – i dikenai kendala A[i] < A[j].

Solusi:



Day: Index i	Input Array A[i]	S[i]: Span of A[i]
0	6	1
1	3	1
2	4	2
3	5	3
4	2	1

Ini adalah masalah yang sangat umum di pasar saham untuk menemukan puncak. Rentang digunakan dalam analisis keuangan (Misalnya, saham pada level tertinggi 52 minggu). Rentang harga saham pada hari tertentu, i, adalah jumlah maksimum hari berturut-turut (sampai dengan hari ini) harga saham telah kurang dari atau sama dengan harga pada i.

Sebagai contoh, mari kita perhatikan tabel dan diagram bentang yang sesuai. Pada gambar, panah menunjukkan panjang bentang. Sekarang, mari kita berkonsentrasi pada algoritma untuk menemukan rentang. Salah satu cara sederhana adalah, setiap hari, periksa berapa hari yang berdekatan memiliki harga saham yang lebih rendah dari harga saat ini.

```
Algorithm: FindingSpans(int A[], int n) {
    //Input: array A of n integers, Output: array S of spans of A
    int i, j, S[n]; //new array of n integers;
    for (i = 0; i < n; i++) {                    Executes n times
            j = 1;                                    n
            while (j <= i && A[i] > A[i-j])           1 + 2 + ...+ (n – 1)
                j = j + 1;                            1 + 2 + ...+ (n – 1)
            S[i] = j;                                 n
    }
    return S;                                         1
}
```

Kompleksitas Waktu: O(n2).

Kompleksitas Ruang: O(1).

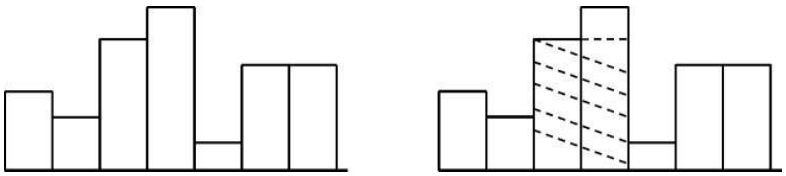
Soal-23 Bisakah kita meningkatkan kompleksitas Soal-22?

Solusi: Dari contoh di atas, kita dapat melihat bahwa rentang S[i] pada hari i dapat dengan mudah dihitung jika kita mengetahui hari terdekat sebelum i, sehingga harga pada hari itu lebih besar daripada harga pada hari i. Mari kita sebut hari seperti P. Jika hari seperti itu ada maka rentang sekarang didefinisikan sebagai S[i] = i – P.

```
Algorithm: FindingSpans(int A[], int n) {
    struct Stack *D = CreateStack();
    int P;
    for (int i = 0 i< n; i++) {
            while (!IsEmptyStack(D) && A[i] > A[Top(D)]) {
                    Pop(D);
            }
            if(IsEmptyStack(D))
                P  = -1;
            else P = Top(D);
            S[i] = i-P;
            Push(D, i);
    }
    return S;
}
```

Kompleksitas Waktu: Setiap indeks array didorong ke dalam tumpukan tepat satu kali dan juga dikeluarkan dari tumpukan paling banyak satu kali. Pernyataan dalam perulangan while dieksekusi paling banyak n kali. Meskipun algoritma memiliki loop bersarang, kompleksitasnya adalah O(n) karena loop dalam hanya dieksekusi n kali selama algoritma (telusuri contoh dan lihat berapa kali loop dalam berhasil). Kompleksitas Ruang: O(n) [untuk tumpukan].

Soal-24 Persegi panjang terbesar di bawah histogram: Histogram adalah poligon yang terdiri dari urutan persegi panjang yang disejajarkan pada garis dasar yang sama. Untuk mempermudah, asumsikan bahwa persegi panjang memiliki lebar yang sama tetapi mungkin memiliki ketinggian yang berbeda. Misalnya, gambar di sebelah kiri menunjukkan histogram yang terdiri dari persegi panjang dengan tinggi 3,2,5,6,1,4,4, diukur dalam satuan di mana 1 adalah lebar persegi panjang. Di sini masalah kita adalah: mengingat sebuah array dengan tinggi persegi panjang (dengan asumsi lebar adalah 1), kita perlu mencari persegi panjang terbesar yang mungkin. Untuk contoh yang diberikan, persegi panjang terbesar adalah bagian yang dibagikan.



Solusi: Jawaban langsung adalah pergi ke setiap batang dalam histogram dan temukan area maksimum yang mungkin dalam histogram untuk itu. Akhirnya, temukan nilai maksimum dari nilai-nilai ini. Ini akan membutuhkan O(n2).

Soal-25 Untuk Soal-24, dapatkah kita meningkatkan kompleksitas waktu?

Solusi: Pencarian linier menggunakan setumpuk sub masalah yang tidak lengkap: Ada banyak cara untuk menyelesaikan masalah ini. Hakim telah memberikan algoritma yang bagus untuk masalah ini yang didasarkan pada tumpukan.

Memproses elemen dalam urutan kiri-ke-kanan dan menyimpan setumpuk informasi tentang sub histogram yang dimulai tetapi belum selesai.

Jika tumpukan kosong, buka sub masalah baru dengan mendorong elemen ke tumpukan. Jika tidak, bandingkan dengan elemen di atas tumpukan. Kalau yang baru lebih besar kita dorong lagi. Jika yang baru sama, kita lewati. Dalam semua kasus ini, Kita melanjutkan dengan elemen baru berikutnya. Jika yang baru kurang, Kita menyelesaikan sub masalah paling atas dengan memperbarui area maksimum sehubungan dengan elemen di bagian atas tumpukan. Kemudian, Kita membuang elemen di atas, dan mengulangi prosedur dengan mempertahankan elemen baru saat ini.

Dengan cara ini, semua sub masalah selesai ketika tumpukan menjadi kosong, atau elemen teratasnya kurang dari atau sama dengan elemen baru, yang mengarah ke tindakan yang dijelaskan di atas. Jika semua elemen telah diproses, dan tumpukan belum kosong, Kita menyelesaikan sub masalah yang tersisa dengan memperbarui area maksimum sehubungan dengan elemen di atas.

```
struct StackItem {
        int height;
        int index;
};
int MaxRectangleArea(int A[], int n) {
        int i, maxArea=-1, top = -1, left, currentArea;
        struct StackItem *S = (struct StackItem *) malloc(sizeof(struct StackItem) * n);
        for(i=0; i<=n; i++) {
                while(top >= 0 && (i==n || S[top]→height > A[i])) {
                        if(top > 0)
                                left = S[top-1]→index;
                        else    left = -1;
                        currentArea = (i – left-1) * S[top]→height;
                        --top;
                        if(currentArea > maxArea)
                                maxArea = currentArea;
                }
                if(i<n) {
                        ++top;
                        S[top]→height = A[i];
                        S[top]→index = i;
                }
        }
        return maxArea;
}
```

Pada kesan pertama, solusi ini tampaknya memiliki kompleksitas O(n2). Tetapi jika kita perhatikan dengan seksama, setiap elemen didorong dan dikeluarkan paling banyak satu kali, dan dalam setiap langkah fungsi setidaknya satu elemen didorong atau dikeluarkan. Karena jumlah pekerjaan untuk keputusan dan

pembaruan konstan, kompleksitas algoritma adalah O(n) dengan analisis diamortisasi. Kompleksitas Ruang: O(n) [untuk tumpukan].

Soal-26 Pada mesin tertentu, bagaimana Anda memeriksa apakah tumpukan bertambah atau berkurang?

Solusi: Coba catat alamat variabel lokal. Panggil fungsi lain dengan variabel lokal yang dideklarasikan di dalamnya dan periksa alamat variabel lokal itu dan bandingkan.

```
int testStackGrowth() {
    int temporary;
    stackGrowth(&temporary);
    exit(0);
}
void stackGrowth(int *temp){
    int temp2;
    printf("\nAddress of first local valuable: %u", temp);
    printf("\nAddress of second local: %u", &temp2);
    if(temp < &temp2)
        printf("\n Stack is growing downwards");
    else
        printf("\n Stack is growing upwards");
}
```

Kompleksitas Waktu: O(1).
Kompleksitas Ruang: O(1).

Soal-27 Diberikan setumpuk bilangan bulat, bagaimana Anda memeriksa apakah setiap pasangan angka yang berurutan dalam tumpukan itu berurutan atau tidak. Pasangan dapat bertambah atau berkurang, dan jika tumpukan memiliki jumlah elemen ganjil, elemen di atas tidak boleh berpasangan. Misalnya, jika tumpukan elemen adalah [4, 5, -2, -3, 11, 10, 5, 6, 20], maka output harus benar karena masing-masing pasangan (4, 5), (- 2, -3), (11, 10), dan (5, 6) terdiri dari angka berurutan.

Solusi: Lihat bab Antrian.

Soal-28 Hapus semua duplikat yang berdekatan secara rekursif: Diberikan string karakter, hapus karakter duplikat yang berdekatan secara rekursif dari string. String keluaran tidak boleh memiliki duplikat yang berdekatan.

Input: careermonk *Output:* camonk	*Input:* mississippi *Output:* m

Solusi: Solusi ini berjalan dengan konsep in-place stack. Ketika elemen di tumpukan tidak cocok dengan karakter saat ini, Kita menambahkannya ke tumpukan. Saat

cocok dengan tumpukan teratas, Kita melewatkan karakter hingga elemen cocok dengan tumpukan teratas dan menghapus elemen dari tumpukan.

```
void removeAdjacentDuplicates(char *str){
   int stkptr=-1;
   int i=0;
   int len=strlen(str);
   while (i<len){
      if (stkptr == -1 || str[stkptr]!=str[i]){
         stkptr++;
         str[stkptr]=str[i];
         i++;
      }else {
         while(i < len&& str[stkptr]==str[i])
            i++;
         stkptr--;
      }
   }
   str[stkptr+1]='\0';
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1) karena simulasi tumpukan dilakukan di tempat.

Soal-29 Diberikan sebuah array elemen, ganti setiap elemen dengan elemen terdekat yang lebih besar di sebelah kanan elemen itu.

Solusi: Satu pendekatan sederhana akan melibatkan pemindaian elemen array dan untuk setiap elemen, pindai elemen yang tersisa dan temukan elemen terdekat yang lebih besar.

```
void replaceWithNearestGreaterElement(int A[], int n){
   int nextNearestGreater = INT_MIN;
   int i = 0, j = 0;
   for (i=0; i<n; i++){
      nextNearestGreater = -INT_MIN;
      for (j = i+1; j<n; j++){
         if (A[i] < A[j]){
            nextNearestGreater = A[j];
            break;
         }
      }
      printf("For the element %d, %d is the nearest greater element\n", A[i], nextNearestGreater);
   }
}
```

Kompleksitas Waktu: O(n2).
Kompleksitas Ruang: O(1).

Soal-30 Untuk Soal-29, dapatkah kita meningkatkan kompleksitasnya?

Solusi: Pendekatannya hampir mirip dengan Soal-22. Buat tumpukan dan dorong elemen pertama. Untuk elemen lainnya, tandai elemen saat ini sebagai

nextNearestGreater. Jika tumpukan tidak kosong, maka keluarkan elemen dari tumpukan dan bandingkan dengan nextNearestGreater. Jika nextNearestGreater lebih besar dari elemen yang muncul, maka nextNearestGreater adalah elemen yang lebih besar berikutnya untuk elemen yang muncul. Terus munculkan dari tumpukan saat elemen yang muncul lebih kecil dari nextNearestGreater. nextNearestGreater menjadi elemen lebih besar berikutnya untuk semua elemen yang muncul. Jika nextNearestGreater lebih kecil dari elemen yang muncul, dorong elemen yang muncul kembali.

```
void replaceWithNearestGreaterElement(int A[], int n){
   int i = 0;
   struct Stack *S = CreateStack();
   int element, nextNearestGreater;
   Push(S, A[0]);
   for (i=1; i<n; i++){
       nextNearestGreater = A[i];
       if (!IsEmptyStack(S)){
       element = Pop(S);
          while (element < nextNearestGreater){
            printf("For the element %d, %d is the nearest greater element\n", A[i], nextNearestGreater);
             if(IsEmptyStack(S))
               break;
             element = Pop(S);
          }
          if (element > nextNearestGreater)
             Push(S, element);
       }
       Push(S, nextNearestGreater);
   }
   while (!IsEmptyStack(S)){
      element = Pop(S);
      nextNearestGreater = -INT_MIN;
      printf("For the element %d, %d is the nearest greater element\n", A[i], nextNearestGreater);
   }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-31 Bagaimana cara mengimplementasikan tumpukan yang akan mendukung operasi berikut dalam kompleksitas waktu O(1)?

- Push yang menambahkan elemen ke atas tumpukan.
- Pop yang menghapus elemen dari atas tumpukan.
- Temukan Tengah yang akan mengembalikan elemen tengah tumpukan.
- Delete Middle yang akan menghapus elemen tengah.

Solusi: Kita dapat menggunakan struktur data LinkedList dengan pointer tambahan ke elemen tengah.

Juga, kita membutuhkan variabel lain untuk menyimpan apakah LinkedList memiliki jumlah elemen genap atau ganjil.

- Push: Tambahkan elemen ke kepala LinkedList. Perbarui pointer ke elemen tengah sesuai dengan variabel.
- Pop: Hapus kepala LinkedList. Perbarui pointer ke elemen tengah sesuai dengan variabel.
- Temukan Tengah: Temukan Tengah yang akan mengembalikan elemen tengah tumpukan.
- Hapus Tengah: Hapus Tengah yang akan menghapus elemen tengah menggunakan logika Soal-43 dari bab Daftar Tertaut.

BAB 5
ANTRIAN *(QUEUE)*

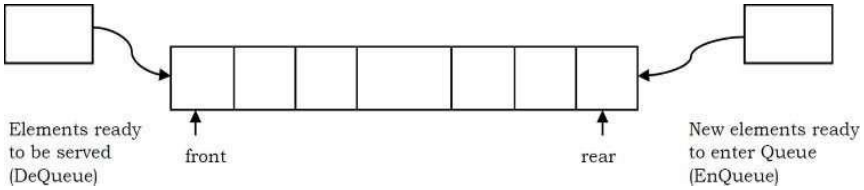
5.1 APA ITU ANTRIAN?

Antrian adalah struktur data yang digunakan untuk menyimpan data (mirip dengan Linked List dan Stacks). Dalam antrian, urutan kedatangan data adalah penting. Secara umum, antrian adalah barisan orang atau benda yang menunggu untuk dilayani secara berurutan mulai dari awal barisan atau urutan tersebut.

Definisi: Antrian adalah daftar berurutan di mana penyisipan dilakukan di satu ujung (belakang) dan penghapusan dilakukan di ujung lain (depan). Elemen pertama yang dimasukkan adalah yang pertama dihapus. Oleh karena itu, ini disebut daftar First in First out (FIFO) atau Last in Last out (LILO).

Mirip dengan Stacks, nama khusus diberikan untuk dua perubahan yang dapat dilakukan pada antrian. Ketika sebuah elemen dimasukkan ke dalam antrian, konsepnya disebut EnQueue, dan ketika sebuah elemen dikeluarkan dari antrian, konsepnya disebut DeQueue.

DeQueueing antrian kosong disebut underflow dan EnQueuing elemen dalam antrian penuh disebut overflow. Umumnya, Kita memperlakukan mereka sebagai pengecualian. Sebagai contoh, pertimbangkan snapshot dari antrian.



Gambar 5.1 snapsot dari antrian

5.2 BAGAIMANA ANTRIAN DIGUNAKAN?

Konsep antrian dapat dijelaskan dengan mengamati antrean di loket reservasi. Ketika kita memasuki garis kita berdiri di ujung garis dan orang yang berada di depan garis adalah orang yang akan dilayani selanjutnya. Dia akan keluar dari antrian dan dilayani.

Ketika ini terjadi, orang berikutnya akan datang di garis depan, akan keluar dari antrian dan akan dilayani. Karena setiap orang di ujung antrean terus keluar dari antrean, Kita bergerak menuju ujung antrean. Akhirnya kita akan mencapai kepala antrean dan kita akan keluar dari antrian dan dilayani. Perilaku ini sangat berguna dalam kasus di mana ada kebutuhan untuk menjaga urutan kedatangan.

5.3 ANTRIAN ADT

Operasi berikut membuat antrian menjadi ADT. Penyisipan dan penghapusan dalam antrian harus mengikuti skema FIFO. Untuk mempermudah, kita asumsikan elemen-elemennya adalah bilangan bulat.

Operasi Antrian Utama

- EnQueue(int data): Menyisipkan elemen di akhir antrian
- int DeQueue(): Menghapus dan mengembalikan elemen di depan antrian

Operasi Antrian Bantu

- int Front(): Mengembalikan elemen di depan tanpa menghapusnya
- int QueueSize(): Mengembalikan jumlah elemen yang disimpan dalam antrian
- int IsEmptyQueueQ: Menunjukkan apakah tidak ada elemen yang disimpan dalam antrian atau tidak

5.4 PENGECUALIAN (*EXCEPTION*)

Serupa dengan ADT lainnya, mengeksekusi DeQueue pada antrian kosong akan memunculkan “Empty Queue Exception” dan mengeksekusi EnQueue pada antrian penuh akan memunculkan “Full Queue Exception”.

5.5 APLIKASI

Berikut ini adalah beberapa aplikasi yang menggunakan antrian.

Aplikasi Langsung

- Sistem operasi menjadwalkan pekerjaan (dengan prioritas yang sama) dalam urutan kedatangan (misalnya, antrian cetak).
- Simulasi antrean dunia nyata seperti antrean di loket tiket atau skenario first-come first-served lainnya membutuhkan antrean.
- Multiprogramming.
- Transfer data asinkron (file IO, pipa, soket).
- Waktu tunggu pelanggan di call center.
- Menentukan jumlah kasir di supermarket.

Aplikasi Tidak Langsung

- Struktur data tambahan untuk algoritma
- Komponen struktur data lainnya

5.6 IMPLEMENTASI

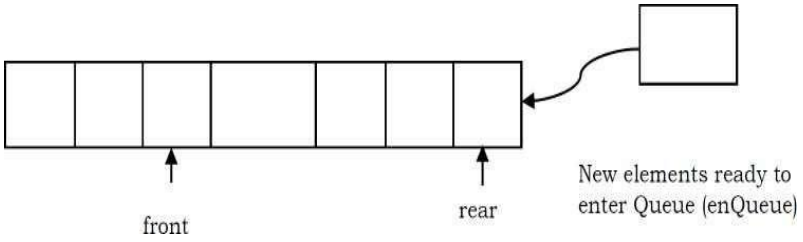
Ada banyak cara (mirip dengan Stacks) untuk mengimplementasikan operasi antrian dan beberapa metode yang umum digunakan tercantum di bawah ini.

- Implementasi berbasis array melingkar sederhana
- Implementasi berbasis array melingkar dinamis
- Implementasi daftar tertaut

Mengapa Array Circle?

Pertama, mari kita lihat apakah kita dapat menggunakan array sederhana untuk mengimplementasikan antrian seperti yang telah kita lakukan untuk tumpukan. Kita tahu bahwa, dalam antrian, penyisipan dilakukan di satu ujung dan penghapusan dilakukan di ujung lainnya. Setelah melakukan beberapa penyisipan dan penghapusan proses menjadi mudah dimengerti.

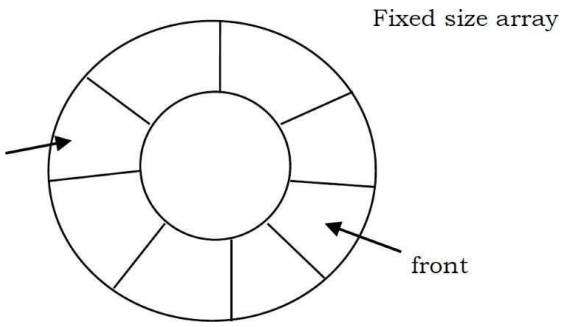
Pada contoh yang ditunjukkan di bawah ini, dapat dilihat dengan jelas bahwa slot awal array semakin terbuang. Jadi, implementasi array sederhana untuk antrian tidak efisien. Untuk mengatasi masalah ini kita menganggap array sebagai array melingkar. Itu berarti, Kita memperlakukan elemen terakhir dan elemen array pertama sebagai berdekatan. Dengan representasi ini, jika ada slot kosong di awal, penunjuk belakang dapat dengan mudah menuju ke slot kosong berikutnya.



Gambar 5.2 implementasi antrian array

Catatan: *Implementasi array melingkar sederhana dan array lingkaran dinamis sangat mirip dengan implementasi array tumpukan. Lihat bab Tumpukan untuk analisis implementasi ini.*

Implementasi Array Circle Sederhana



Gambar 5.3 Implementasi arrya circle sederhana

Implementasi sederhana dari Queue ADT ini menggunakan array. Dalam larik, kita menambahkan elemen secara melingkar dan menggunakan dua variabel untuk melacak elemen awal dan elemen akhir. Umumnya, depan digunakan untuk menunjukkan elemen awal dan belakang digunakan untuk menunjukkan elemen akhir dalam antrian. Array yang menyimpan elemen antrian mungkin menjadi penuh. Operasi EnQueue kemudian akan

mengeluarkan pengecualian antrian penuh. Demikian pula, jika kita mencoba menghapus elemen dari antrian kosong, itu akan membuang pengecualian antrian kosong.

Catatan: *Awalnya, depan dan belakang menunjuk ke -1 yang menunjukkan bahwa antrian kosong.*

```
struct ArrayQueue {
    int front, rear;
    int capacity;
    int *array;
};
struct ArrayQueue *Queue(int size) {
    struct ArrayQueue *Q = malloc(sizeof(struct ArrayQueue));
    if(!Q)
        return NULL;
    Q→capacity = size;
    Q→front = Q→rear = -1;
    Q→array= malloc(Q→capacity * sizeof(int));
    if(!Q→array)
        return NULL;
    return Q;
}
int IsEmptyQueue(struct ArrayQueue *Q) {
    // if the condition is true then 1 is returned else 0 is returned
    return (Q→front == -1);
}
int IsFullQueue(struct ArrayQueue *Q) {
    //if the condition is true then 1 is returned else 0 is returned
    return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
    return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct ArrayQueue *Q, int data) {
    if(IsFullQueue(Q))
            printf("Queue Overflow");
    else {
            Q→rear = (Q→rear+1) % Q→capacity;
            Q→ array[Q→rear]= data;
            if(Q→front == -1)
                    Q→front = Q→rear;
    }
}
int DeQueue(struct ArrayQueue *Q) {
    int data = 0;//or element which does not exist in Queue
    if(IsEmptyQueue(Q)) {
            printf("Queue is Empty");
            return 0;
    }
    else {
        data = Q→array[Q→front];
        if(Q→front == Q→rear)
            Q→front = Q→rear = -1;
        else  Q→front = (Q→front+1) % Q→capacity;
    }
    return data;
}
void DeleteQueue(struct ArrayQueue *Q) {
    if(Q) {
        if(Q→array)
            free(Q→array);
        free(Q);
    }
}
```

Kinerja dan Keterbatasan

Kinerja: Biarkan n menjadi jumlah elemen dalam antrian:

Kompleksitas Ruang (untuk n operasi EnQueue)	O(*n*)
Kompleksitas Waktu EnQueue()	O(1)
Kompleksitas Waktu DeQueue()	O(1)
Kompleksitas Waktu dari Antrian Kosong()	O(1)
Kompleksitas Waktu Antrian Penuh()	O(1)
Kompleksitas Waktu Ukuran Antrian()	O(1)
Kompleksitas Waktu Hapus Antrian()	O(1)

Batasan: Ukuran maksimum antrian harus ditentukan sebelumnya dan tidak dapat diubah. Mencoba memasukkan elemen baru ke dalam antrian penuh menyebabkan pengecualian khusus implementasi.

Implementasi Array Melingkar Dinamis

```
struct DynArrayQueue {
    int front, rear;
    int capacity;
    int *array;
};
struct DynArrayQueue *CreateDynQueue() {
    struct DynArrayQueue *Q = malloc(sizeof(struct DynArrayQueue));
    if(!Q)
        return NULL;
    Q→capacity = 1;
    Q→front = Q→rear = -1;
    Q→array = malloc(Q→capacity * sizeof(int));
    if(!Q→array)
          return NULL;
    return Q;
}
int IsEmptyQueue(struct DynArrayQueue *Q) {
    // if the condition is true then 1 is returned else 0 is returned
    return (Q→front == -1);
}
int IsFullQueue(struct DynArrayQueue *Q) {
    //if the condition is true then 1 is returned else 0 is returned
    return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
    return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct DynArrayQueue *Q, int data) {
    if(IsFullQueue(Q))
        ResizeQueue(Q);
    Q→rear = (Q→rear+1)% Q→capacity;
    Q→ array[Q→rear]= data;

    if(Q→front == -1)
        Q→front = Q→rear;
}
void ResizeQueue(struct DynArrayQueue *Q) {
    int size = Q→capacity;
    Q→capacity = Q→capacity*2;
    Q→array = realloc (Q→array, Q→capacity);
    if(!Q→array) {
        printf("Memory Error");
        return;
    }
    if(Q→front > Q→rear ) {
        for(int i=0; i < Q→front; i++) {
            Q→array[i+size] =Q→array[i];
        }
    Q→rear = Q→rear + size;
    }
}
int DeQueue(struct DynArrayQueue *Q) {
    int data = 0;//or element which does not exist in Queue
    if(IsEmptyQueue(Q)) {
        printf("Queue is Empty");
        return 0;
    }
    else {
        data = Q→array[Q→front];
        if(Q→front== Q→rear)
            Q→front= Q→rear = -1;
        else
            Q→front = (Q→front+1) % Q→capacity;
    }
    return data;
}
void DeleteQueue(struct DynArrayQueue *Q) {
    if(Q) {
        if(Q→array)
            free(Q→array);
        free(Q→array);
    }
}
```

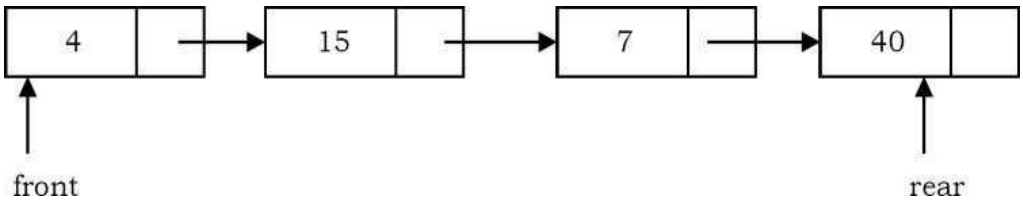
Kinerja

Misalkan n adalah jumlah elemen dalam antrian.

Kompleksitas Ruang (untuk n operasi EnQueue)	O(*n*)
Kompleksitas Waktu EnQueue()	O(1) (Rata rata)
Kompleksitas Waktu DeQueue()	O(1)
Kompleksitas Waktu Ukuran Antrian()	O(1)
Kompleksitas Waktu dari Antrian Kosong()	O(1)
Kompleksitas Waktu Antrian Penuh()	O(1)
Kompleksitas Waktu Ukuran Antrian()	O(1)
Kompleksitas Waktu Hapus Antrian()	O(1)

Implementasi Daftar Tertaut

Cara lain untuk mengimplementasikan antrian adalah dengan menggunakan Daftar tertaut. Operasi EnQueue diimplementasikan dengan menyisipkan elemen di akhir daftar. Operasi DeQueue diimplementasikan dengan menghapus elemen dari awal daftar.



Gambar 5.4 implementasi daftar tertaut

```
struct ListNode {
    int data;
    struct ListNode *next;
};
```

```
struct Queue {
    struct ListNode *front;
    struct ListNode *rear;
};
```

```
struct Queue *CreateQueue() {
    struct Queue *Q;
    struct ListNode *temp;
    Q = malloc(sizeof(struct Queue));
    if(!Q)
        return NULL;
    temp = malloc(sizeof(struct ListNode));
    Q→front = Q→rear = NULL;
    return Q;
}
```

```
int IsEmptyQueue(struct Queue *Q) {
    // if the condition is true then 1 is returned else 0 is returned
    return (Q→front == NULL);
}
void EnQueue(struct Queue *Q, int data) {
    struct ListNode *newNode;
    newNode = malloc(sizeof(struct ListNode));
    if(!newNode)
        return NULL;
    newNode→data = data;
    newNode→next = NULL;
    if(Q→rear) Q→rear→next = newNode;
    Q→rear = newNode;

    if(Q→front == NULL)
        Q→front = Q→rear;
}
int DeQueue(struct Queue *Q) {
    int data = 0;      //or element which does not exist in Queue
    struct ListNode *temp;
    if(IsEmptyQueue(Q)) {
        printf("Queue is empty");
        return 0;
    }
    else {
        temp = Q→front;
        data = Q→front→data;
        Q→front== Q→front→next;
        free(temp);
    }
    return data;
}
void DeleteQueue(struct Queue *Q) {
    struct ListNode *temp;
    while(Q) {
        temp = Q;
        Q = Q→next;
        free(temp);
    }
    free(Q);
}
```

Kinerja

Misalkan n adalah jumlah elemen dalam antrian, maka

Kompleksitas Ruang (untuk n operasi EnQueue)	O(*n*)
Kompleksitas Waktu EnQueue()	O(1) (Rata rata)
Kompleksitas Waktu DeQueue()	O(1)
Kompleksitas Waktu dari Antrian Kosong()	O(1)
Kompleksitas Waktu Hapus Antrian()	O(1)

Perbandingan Implementasi

Catatan: Perbandingan sangat mirip dengan implementasi tumpukan dan bab Tumpukan.

5.7 ANTRIAN: MASALAH & SOLUSI

Soal-1 Berikan algoritma untuk membalikkan antrian Q. Untuk mengakses antrian, kita hanya diperbolehkan menggunakan metode antrian ADT.

Solusi:

```
void ReverseQueue(struct Queue *Q) {
    struct Stack *S = CreateStack();
    while (!IsEmptyQueue(Q))
        Push(S, DeQueue(Q))
    while (!IsEmptyStack(S))
        EnQueue(Q, Pop(S));
}
```

Kompleksitas Waktu: O(n).

Soal-2 Bagaimana Anda bisa mengimplementasikan antrian menggunakan dua tumpukan?

Solusi: Biarkan SI dan S2 menjadi dua tumpukan yang akan digunakan dalam implementasi antrian. Yang harus kita lakukan adalah mendefinisikan operasi EnQueue dan DeQueue untuk antrian.

```
struct Queue {
    struct Stack *S1; // for EnQueue
    struct Stack *S2; // for DeQueue
}
```

Algoritma EnQueue

- Tekan saja untuk menumpuk S1

```
void EnQueue(struct Queue *Q, int data) {
    Push(Q→S1, data);
}
```

Kompleksitas Waktu: O(1).

Algoritma DeQueue

- Jika tumpukan S2 tidak kosong maka keluarkan dari S2 dan kembalikan elemen tersebut.
- Jika tumpukan kosong, pindahkan semua elemen dari SI ke S2 dan keluarkan elemen teratas dari S2 dan kembalikan elemen yang muncul [kita dapat mengoptimalkan kode sedikit dengan mentransfer hanya n – 1 elemen dari SI ke S2 dan memunculkan elemen ke-n dari SI dan kembalikan elemen yang muncul].
- Jika stack S1 juga kosong maka lemparkan error.

```
int DeQueue(struct Queue *Q) {
    if(!IsEmptyStack(Q→S2))
        return Pop(Q→S2);
    else {
        while(!IsEmptyStack(Q→S1))
            Push(Q→S2, Pop(Q→S1));
        return Pop(Q→S2);
    }
}
```

Kompleksitas Waktu: Dari algoritma, jika tumpukan S2 tidak kosong maka kompleksitasnya adalah O(1). Jika tumpukan S2 kosong, maka kita perlu mentransfer elemen dari SI ke S2. Tetapi jika kita amati dengan cermat, jumlah elemen yang ditransfer dan jumlah elemen yang muncul dari S2 adalah sama. Karena ini rata-rata kompleksitas operasi pop dalam kasus ini adalah O(1). Kompleksitas yang diamortisasi dari operasi pop adalah O(1).

Soal-3 Tunjukkan bagaimana Anda dapat mengimplementasikan satu tumpukan secara efisien menggunakan dua antrian. Analisis waktu berjalan operasi tumpukan.

Solusi: Ya, adalah mungkin untuk mengimplementasikan Stack ADT menggunakan 2 implementasi dari Queue ADT. Salah satu antrian akan digunakan untuk menyimpan elemen dan yang lainnya untuk menahannya sementara selama metode pop dan top. Metode push akan mengantrekan elemen yang diberikan ke antrian penyimpanan. Metode teratas akan mentransfer semua kecuali elemen terakhir dari antrian penyimpanan ke antrian sementara, menyimpan elemen depan dari antrian penyimpanan untuk dikembalikan, mentransfer elemen terakhir ke antrian sementara, kemudian mentransfer semua elemen kembali ke antrian penyimpanan. Metode pop akan melakukan hal yang sama seperti top, kecuali alih-alih mentransfer elemen terakhir ke antrian sementara setelah menyimpannya untuk dikembalikan, elemen terakhir itu akan dibuang. Biarkan Q1 dan Q2 menjadi dua antrian yang akan digunakan dalam implementasi stack. Yang harus kita lakukan adalah mendefinisikan operasi push dan pop untuk stack.

```
struct Stack {
    struct Queue *Q1;
    struct Queue *Q2;
}
```

Dalam algoritma di bawah ini, Kita memastikan bahwa satu antrian selalu kosong.

Algoritma Operasi Dorong: Masukkan elemen di antrian mana pun yang tidak kosong.

- Periksa apakah antrian Q1 kosong atau tidak. Jika Q1 kosong maka Enqueue elemen ke Q2.

- Jika tidak, Enqueue elemen ke Q1.

```
Push(struct Stack *S, int data) {
      if(IsEmptyQueue(S→Q1))
          EnQueue(S→Q2, data);
      else      EnQueue(S→Q1, data);
}
```

Kompleksitas Waktu: O(1).

Algoritma Operasi Pop: Transfer n – 1 elemen ke antrian lain dan hapus terakhir dari antrian untuk melakukan operasi pop.

- Jika antrian Q1 tidak kosong maka transfer n – 1 elemen dari Q1 ke Q2 dan kemudian, DeQueue elemen terakhir dari Q1 dan kembalikan.
- Jika antrian Q2 tidak kosong maka transfer n – 1 elemen dari Q2 ke Q1 dan kemudian, DeQueue elemen terakhir dari Q2 dan kembalikan.

```
int Pop(struct Stack *S) {
      int i, size;
      if(IsEmptyQueue(S→Q2)) {
              size = Size(S→Q1);
              i = 0;
              while(i < size-1) {
                EnQueue(S→Q2, DeQueue(S→Q1));
                i++;
              }
              return DeQueue(S→Q1);
      }
      else {
           size = Size(S→Q2);
           while(i < size-1) {
                EnQueue(S→Q1, DeQueue(S→Q2));
                i++;
           }
           return DeQueue(S→Q2);
      }
}
```

Kompleksitas Waktu: Waktu menjalankan operasi pop adalah O(n) karena setiap kali pop dipanggil, Kita mentransfer semua elemen dari satu antrian ke antrian lainnya.

Soal-4 Jumlah maksimum dalam jendela geser: Diberikan larik A[] dengan jendela geser berukuran w yang bergerak dari paling kiri larik ke paling kanan. Asumsikan bahwa kita hanya dapat melihat angka w di jendela. Setiap kali jendela geser bergerak ke kanan dengan satu posisi. Contoh: Array adalah [1 3 -1 -3 5 3 6 7], dan w adalah 3.

Posisi jendela	Maks
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3

1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: Array panjang A[], dan lebar jendela w. Output: Array B[], B[i] adalah nilai maksimum dari A[i] hingga A[i+w-1]. Persyaratan: Temukan cara optimal yang baik untuk mendapatkan B[i]

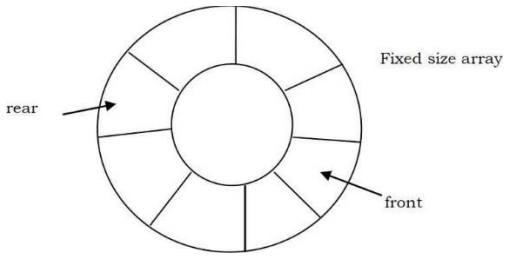
Solusi: Masalah ini dapat diselesaikan dengan antrian berujung ganda (yang mendukung penyisipan dan penghapusan di kedua ujungnya). Lihat bab Antrian Prioritas untuk algoritma.

Soal-5 Diberikan antrian Q yang berisi n elemen, pindahkan item-item ini ke tumpukan S (awalnya kosong) sehingga elemen depan Q muncul di bagian atas tumpukan dan urutan semua item lainnya dipertahankan. Menggunakan operasi enqueue dan dequeue untuk antrian, dan operasi push dan pop untuk stack, menguraikan algoritma O(n) yang efisien untuk menyelesaikan tugas di atas, hanya menggunakan jumlah penyimpanan tambahan yang konstan.

Solusi: Asumsikan elemen antrian Q adalah a1:a2 ...an. Dequeuing semua elemen dan mendorongnya ke tumpukan akan menghasilkan tumpukan dengan a di atas dan a1 di bawah. Ini dilakukan dalam waktu O(n) sebagai dequeue dan setiap push membutuhkan waktu yang konstan per operasi. Antrian sekarang kosong. Dengan memunculkan semua elemen dan mendorongnya ke antrian, kita akan mendapatkan a1 di bagian atas tumpukan. Ini dilakukan lagi dalam waktu O(n). Seperti dalam aritmatika besar, kita dapat mengabaikan faktor konstan. Proses dilakukan dalam waktu O(n). Jumlah penyimpanan tambahan yang dibutuhkan di sini harus cukup besar untuk menampung satu item sementara.

Soal-6 Antrian diatur dalam array melingkar A[O..n - 1] dengan depan dan belakang didefinisikan seperti biasa. Asumsikan bahwa n – 1 lokasi dalam array tersedia untuk menyimpan elemen (dengan elemen lain digunakan untuk mendeteksi kondisi penuh/kosong). Berikan rumus untuk jumlah elemen dalam antrian dalam hal belakang, depan, dan n.

Solusi: Perhatikan gambar berikut untuk mendapatkan gambaran yang jelas tentang antrian.



- Bagian belakang antrian berada di suatu tempat searah jarum jam dari depan.
- Untuk mengantrekan elemen, kita pindah ke belakang satu posisi searah jarum jam dan menulis elemen di posisi itu.
- Untuk dequeue, kita cukup menggeser ke depan satu posisi searah jarum jam.
- Antrian bermigrasi searah jarum jam saat kita enqueue dan dequeue.
- Kekosongan dan kepenuhan harus diperiksa dengan cermat.
- Analisis kemungkinan situasi (buat beberapa gambar untuk melihat di mana depan dan belakang adalah ketika antrian kosong, dan terisi sebagian dan seluruhnya). Kita akan mendapatkan ini:

$$Number\ Of\ Elements = \begin{cases} rear - front + 1 & \text{if rear} == \text{front} \\ rear - front + n & \text{otherwise} \end{cases}$$

Soal-7 Apa struktur data yang paling tepat untuk mencetak elemen antrian dalam urutan terbalik?

Solusi Tumpuk.

Soal-8 Menerapkan antrian berakhir ganda. Antrian berujung ganda adalah struktur data abstrak yang mengimplementasikan antrian yang elemennya hanya dapat ditambahkan atau dihapus dari depan (kepala) atau belakang (ekor). Hal ini juga sering disebut daftar terkait kepala-ekor.

Solusi:

```
void pushBackDEQ(struct ListNode **head, int data){
    struct ListNode *newNode = (struct ListNode*) malloc(sizeof(struct ListNode));
    newNode→data = data;
    if(*head == NULL){
        *head = newNode;
        (*head)→next = *head;
        (*head)→prev = *head;
    }
    else{
        newNode→prev = (*head)→prev;
        newNode→next = *head;
        (*head)→prev→next = newNode;
        (*head)→prev = newNode;
    }
}
void pushFrontDEQ(struct ListNode **head, int data){
    pushBackDEQ(head,data);
    *head = (*head)→prev;
}
int popBackDEQ(struct ListNode **head){
    int data;
    if( (*head)→prev == *head ){
        data = (*head)→data;
        free(*head);
        *head = NULL;
    }
    else{
        struct ListNode *newTail = (*head)→prev→prev;
        data = (*head)→prev→data;
        newTail→next = *head;
        free((*head)→prev);
        (*head) →prev = newTail;
    }
    return data;
}
int popFront(struct ListNode **head){
    int data;
    *head = (*head)→next;
    data = popBackDEQ(head);
    return data;
}
```

Soal-9 Diberikan setumpuk bilangan bulat, bagaimana Anda memeriksa apakah setiap pasangan angka yang berurutan dalam tumpukan itu berurutan atau tidak. Pasangan dapat bertambah atau berkurang, dan jika tumpukan memiliki jumlah elemen ganjil, elemen di atas tidak boleh berpasangan. Misalnya, jika tumpukan elemen adalah [4, 5, -2, -3, 11, 10, 5, 6, 20], maka outputnya harus benar karena setiap pasangan (4, 5), (-2, -3), (11, 10), dan (5, 6) terdiri dari bilangan berurutan.

Solusi:

```
int checkStackPairwiseOrder(struct Stack *s) {
    struct Queue *q = CreateQueue();
    int pairwiseOrdered = 1;
    while (!isEmptyStack(s))
        EnQueue(q, Pop(s));
    while (!IsEmptyQueue(q))
        Push(s, DeQueue(q));
    while (!isEmptyStack(s)) {
        int n = Pop(s);
        EnQueue(q, n);
        if (!isEmptyStack(s)) {
            int m = Pop(s);
            EnQueue(q, m);
            if (abs(n - m) != 1) {
                pairwiseOrdered = 0;
            }
        }
    }
    while (!IsEmptyQueue(q))
        Push(s, DeQueue(q));
    return pairwiseOrdered;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-10 Diberikan antrian bilangan bulat, atur ulang elemen dengan menyisipkan paruh pertama daftar dengan paruh kedua daftar. Misalnya, antrian menyimpan urutan nilai berikut: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Pertimbangkan dua bagian dari daftar ini: babak pertama: [11, 12, 13, 14, 15] babak kedua: [16, 17, 18, 19, 20]. Ini digabungkan secara bergantian untuk membentuk urutan pasangan interleave: nilai pertama dari setiap setengah (11 dan 16), kemudian nilai kedua dari setiap setengah (12 dan 17), kemudian nilai ketiga dari masing-masing setengah (13 dan 18), dan seterusnya. Di setiap pasangan, nilai dari babak pertama muncul sebelum nilai dari babak kedua. Jadi, setelah panggilan, antrian menyimpan nilai-nilai berikut: [11, 16, 12, 17, 13, 18, 14, 19, 15, 20].

Solusi:

```
void interLeavingQueue(struct Queue *q) {
    if (Size(q) % 2 != 0)
        return;
    struct Stack *s = CreateStack();
    int halfSize = Size(q) / 2;
    for (int i = 0; i < halfSize; i++)
        Push(s, DeQueue(q));
    while (!isEmptyStack(s))
        EnQueue(q, Pop(s));
    for (int i = 0; i < halfSize; i++)
        EnQueue(q, DeQueue(q));
    for (int i = 0; i < halfSize; i++)
        Push(s, DeQueue(q));
    while (!isEmptyStack(s)) {
        EnQueue(q, Pop(s));
        EnQueue(q, DeQueue(q));
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-11 Mengingat bilangan bulat k dan antrian bilangan bulat, bagaimana Anda membalikkan urutan k elemen pertama dari antrian, meninggalkan elemen lain dalam urutan relatif yang sama? Misalnya, jika k=4 dan antrian memiliki elemen [10, 20, 30, 40, 50, 60, 70, 80, 90]; outputnya harus [40, 30, 20, 10, 50, 60, 70, 80, 90].

Solusi:

```
void reverseQueueFirstKElements(int k, struct Queue *q) {
    if (q == Null || k > Size(q)) {
        return;
    }
    else if (k > 0) {
        struct Stack *s = CreateStack();
        for (int i = 0; i < k; i++) {
            Push(s, DeQueue(q));
        }
        while (!isEmptyStack(s)) {
            EnQueue(q, Pop(s));
        }
        for (int i = 0; i < Size(q) - k; i++) { // wrap around rest of elements
            EnQueue(q, DeQueue(q));
        }
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

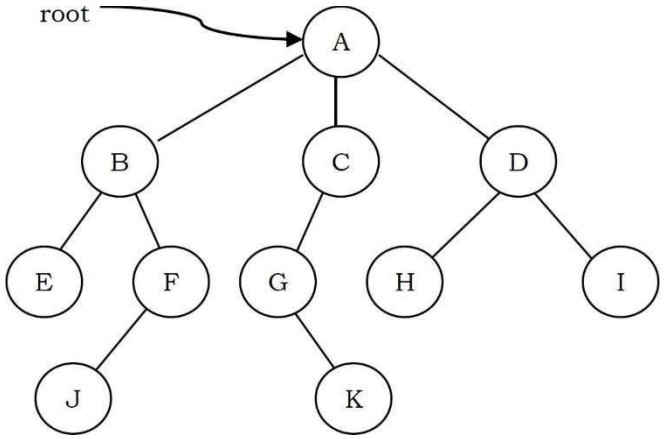
BAB 6
POHON *(TREE)*

6.1 APA ITU POHON?

Pohon adalah struktur data yang mirip dengan daftar tertaut tetapi alih-alih setiap simpul menunjuk hanya ke simpul berikutnya secara linier, setiap simpul menunjuk ke sejumlah simpul. Pohon adalah contoh struktur data non-linier. Struktur pohon adalah cara untuk merepresentasikan sifat hierarkis dari suatu struktur dalam bentuk grafik.

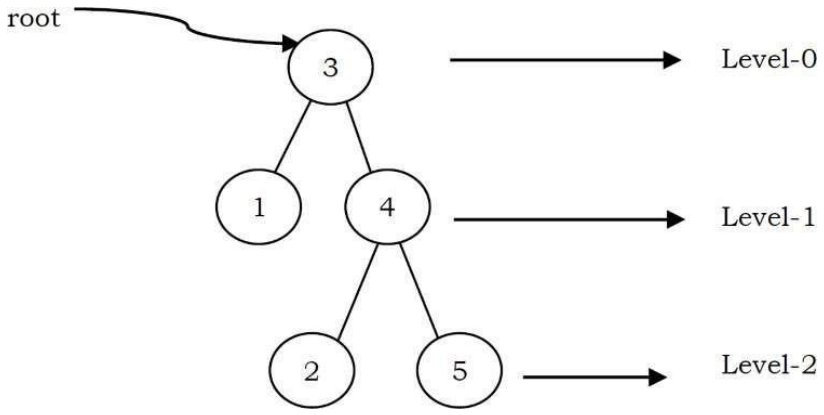
Pada pohon ADT (Abstract Data Type), urutan elemen tidak penting. Jika kita membutuhkan informasi pemesanan, struktur data linier seperti daftar tertaut, tumpukan, antrian, dll. dapat digunakan.

6.2 DAFTAR ISTILAH



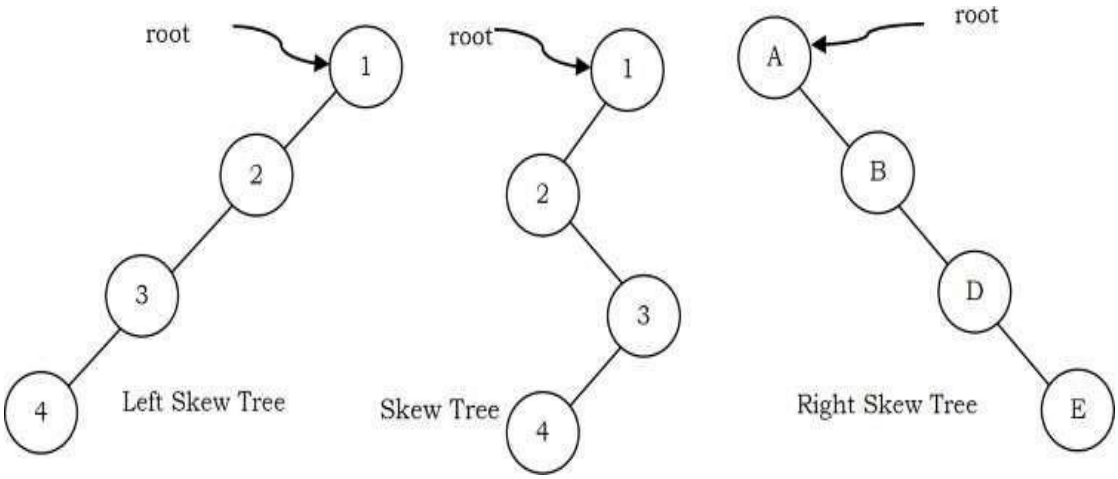
Gambar 6.1 Akar pohon

- Akar pohon adalah simpul tanpa orang tua. Paling banyak ada satu simpul akar dalam sebuah pohon (simpul A dalam contoh di atas).
- Tepi mengacu pada tautan dari induk ke anak (semua tautan dalam gambar).
- Sebuah simpul tanpa anak disebut simpul daun (E,J,K,H dan I).
- Anak dari orang tua yang sama disebut saudara kandung (B,C,D adalah saudara dari A, dan E,F adalah saudara dari B).
- Sebuah node p adalah ancestor dari node q jika terdapat sebuah path dari root ke q dan p muncul pada path tersebut. Simpul q disebut turunan dari p. Misalnya, A,C dan G adalah nenek moyang dari if.
- Himpunan semua node pada kedalaman tertentu disebut level pohon (B, C dan D adalah level yang sama). Node akar berada pada level nol.



Gambar 6.2 level pohon

- Kedalaman suatu simpul adalah panjang lintasan dari akar ke simpul tersebut (kedalaman G adalah 2, A – C – G).
- Ketinggian suatu simpul adalah panjang lintasan dari simpul tersebut ke simpul terdalam. Ketinggian pohon adalah panjang jalan dari akar ke simpul terdalam di pohon. Sebuah pohon (berakar) dengan hanya satu simpul (akar) memiliki ketinggian nol. Pada contoh sebelumnya, tinggi B adalah 2 (B – F – J).
- Tinggi pohon adalah tinggi maksimum di antara semua simpul di pohon dan kedalaman pohon adalah kedalaman maksimum di antara semua simpul di pohon. Untuk pohon tertentu, kedalaman dan tinggi mengembalikan nilai yang sama. Tetapi untuk node individu Kita mungkin mendapatkan hasil yang berbeda.
- Ukuran dari sebuah simpul adalah jumlah keturunan yang dimilikinya termasuk dirinya sendiri (ukuran dari subpohon C adalah 3).
- Jika setiap simpul dalam sebuah pohon hanya memiliki satu anak (kecuali simpul daun) maka kita menyebutnya pohon miring. Jika setiap simpul hanya memiliki anak kiri maka kita menyebutnya pohon miring kiri. Demikian pula, jika setiap simpul hanya memiliki anak kanan maka kita menyebutnya pohon miring kanan.

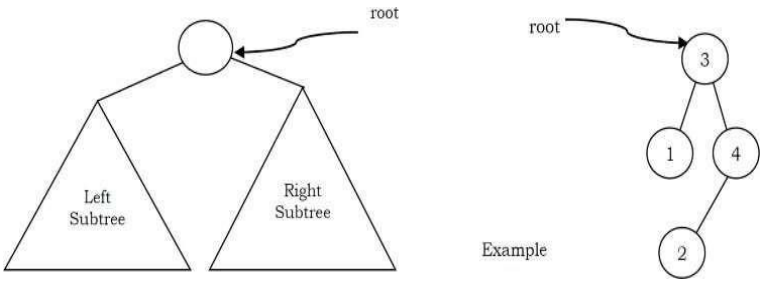


Gambar 6.3 skew tree

6.3 POHON BINER

Sebuah pohon disebut pohon biner jika setiap simpul memiliki nol anak, satu anak atau dua anak. Pohon kosong juga merupakan pohon biner yang valid. Kita dapat memvisualisasikan pohon biner yang terdiri dari akar dan dua pohon biner yang terpisah, yang disebut subpohon kiri dan kanan dari akar.

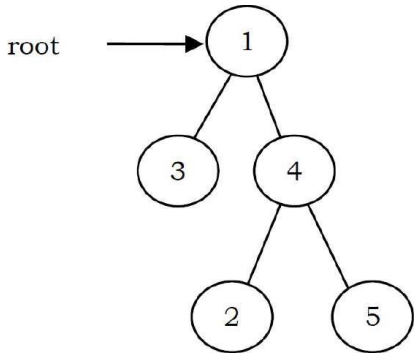
Pohon Biner Generik



Gambar 6.4 pohon biner generik

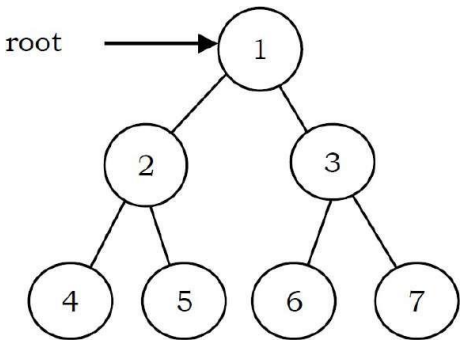
6.4 JENIS POHON BINER

Pohon Biner Ketat: Pohon biner disebut pohon biner ketat jika setiap simpul memiliki tepat dua anak atau tidak ada anak.



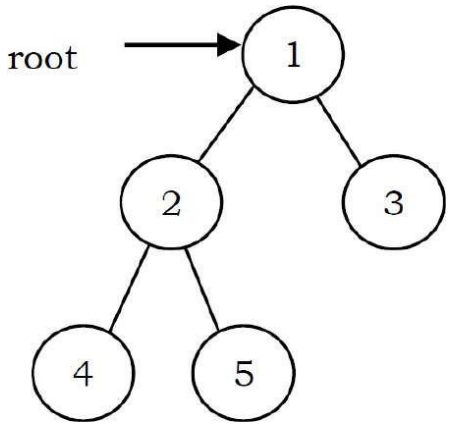
Gambar 6.5 pohon biner ketat

Pohon Biner Penuh: Pohon biner disebut pohon biner penuh jika setiap simpul memiliki tepat dua anak dan semua simpul daun berada pada level yang sama.



Gambar 6.6 pohon biner penuh

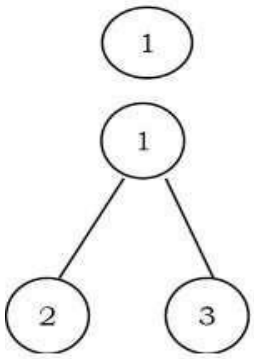
Pohon Biner Lengkap: Sebelum mendefinisikan pohon biner lengkap, mari kita asumsikan bahwa tinggi pohon biner adalah h. Dalam pohon biner lengkap, jika kita memberikan penomoran untuk simpul dengan memulai dari akar (misalkan simpul akar memiliki 1) maka kita mendapatkan urutan lengkap dari 1 hingga jumlah simpul di pohon. Saat melintasi kita harus memberikan penomoran untuk pointer NULL juga. Pohon biner disebut pohon biner lengkap jika semua simpul daun berada pada ketinggian h atau h – 1 dan juga tanpa ada nomor yang hilang dalam urutannya.



Gambar 6.7 pohon biner lengkap

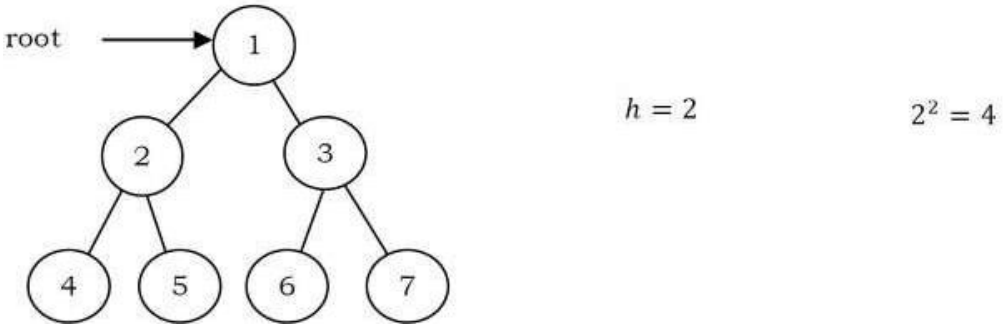
6.5 SIFAT POHON BINER

Untuk sifat-sifat berikut, mari kita asumsikan bahwa tinggi pohon adalah h. Juga, asumsikan bahwa simpul akar berada pada ketinggian nol.



Height	Number of nodes at level h
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$

Gambar 6.8 sifat pohon biner



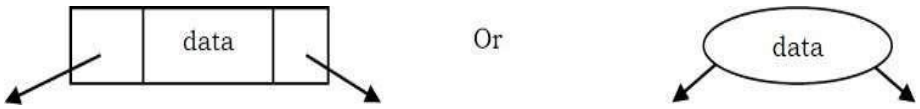
Gambar 6.9 sifat pohon biner (2)

Dari diagram kita dapat menyimpulkan sifat-sifat berikut:

- Jumlah node n dalam pohon biner penuh adalah 2h+1 – 1. Karena ada h level, kita perlu menambahkan semua node di setiap level [20 + 21+ 22 + ••• + 2h = 2h+1 – 1].
- Jumlah node n dalam pohon biner lengkap adalah antara 2h (minimum) dan 2h+1
  - 1 (maksimum). Untuk informasi lebih lanjut tentang ini, lihat bab Antrian Prioritas.
- Jumlah simpul daun dalam pohon biner penuh adalah 2 jam.
- Jumlah link NULL (pointer terbuang) dalam pohon biner lengkap dari n node adalah n + 1.

Struktur Pohon Biner

Sekarang mari kita mendefinisikan struktur pohon biner. Untuk mempermudah, asumsikan bahwa data dari node adalah bilangan bulat. Salah satu cara untuk merepresentasikan sebuah node (yang berisi data) adalah dengan memiliki dua link yang mengarah ke anak kiri dan kanan bersama dengan bidang data seperti yang ditunjukkan di bawah ini:



Gambar 6.10 struktur pohon biner

```
struct BinaryTreeNode {
   int data;
   struct BinaryTreeNode *left;
   struct BinaryTreeNode *right;
};
```

Catatan: Di pohon, aliran default adalah dari induk ke anak dan tidak wajib untuk menampilkan cabang terarah. Untuk diskusi Kita, Kita menganggap kedua representasi yang ditunjukkan di bawah ini sama.

Gambar 6.11 struktur pohon biner

Operasi pada Pohon Biner

Operasi Dasar

- Memasukkan elemen ke dalam pohon
- Menghapus elemen dari pohon
- Mencari elemen
- Melintasi pohon

Operasi bantu

- Mencari ukuran pohon
- Mencari tinggi pohon
- Menemukan level yang memiliki jumlah maksimum
- Mencari LCA (Least Common ancestor) untuk pasangan node tertentu, dan banyak lagi.

Aplikasi Pohon Biner

Berikut adalah beberapa aplikasi di mana pohon biner memainkan peran penting:

- Pohon ekspresi digunakan dalam kompiler.
- Pohon pengkodean Huffman yang digunakan dalam algoritma kompresi data.
- Pohon Pencarian Biner (BST), yang mendukung pencarian, penyisipan dan penghapusan pada koleksi item dalam O(logn) (rata-rata).
- Priority Queue (PQ), yang mendukung pencarian dan penghapusan minimum (atau maksimum) pada kumpulan item dalam waktu logaritmik (dalam kasus terburuk).

6.6 TRAVERSAL POHON BINER

Untuk memproses pohon, kita memerlukan mekanisme untuk melintasinya, dan itu menjadi subjek dari bagiannya. Proses mengunjungi semua simpul pohon disebut traversal pohon. Setiap node diproses hanya sekali tetapi dapat dikunjungi lebih dari sekali. Seperti yang telah kita lihat dalam struktur data linier (seperti daftar tertaut, tumpukan, antrian, dll.), elemen dikunjungi secara berurutan. Tapi, dalam struktur pohon ada banyak cara berbeda.

Penjelajahan pohon seperti mencari pohon, kecuali bahwa dalam penjelajahan tujuannya adalah untuk bergerak melalui pohon dalam urutan tertentu. Selain itu, semua node diproses dalam traversal tetapi pencarian berhenti ketika node yang diperlukan ditemukan.

Kemungkinan Traversal

Mulai dari akar pohon biner, ada tiga langkah utama yang dapat dilakukan dan urutan pelaksanaannya menentukan tipe traversal. Langkah-langkah ini adalah: melakukan tindakan pada node saat ini (disebut sebagai "mengunjungi" node dan dilambangkan dengan "D"), melintasi ke node anak kiri (dilambangkan dengan "L"), dan melintasi ke node anak kanan (

dilambangkan dengan “R”). Proses ini dapat dengan mudah dijelaskan melalui rekursi. Berdasarkan definisi di atas ada 6 kemungkinan:

1. LDR: Proses subpohon kiri, proses data node saat ini dan kemudian proses subpohon kanan
2. LRD: Proses subpohon kiri, proses subpohon kanan dan kemudian proses data node saat ini
3. DLR: Memproses data node saat ini, memproses subpohon kiri dan kemudian memproses subpohon kanan
4. DRL: Memproses data node saat ini, memproses subpohon kanan dan kemudian memproses subpohon kiri
5. RDL: Proses subpohon kanan, proses data node saat ini dan kemudian proses subpohon kiri
6. RLD: Proses subpohon kanan, proses subpohon kiri dan kemudian proses data node saat ini

Mengklasifikasikan Traversal

Urutan di mana entitas ini (node) diproses mendefinisikan metode traversal tertentu. Klasifikasi didasarkan pada urutan di mana node saat ini diproses. Artinya, jika kita mengklasifikasikan berdasarkan simpul saat ini (D) dan jika D datang di tengah maka tidak masalah apakah L di sisi kiri D atau R di sisi kiri D.

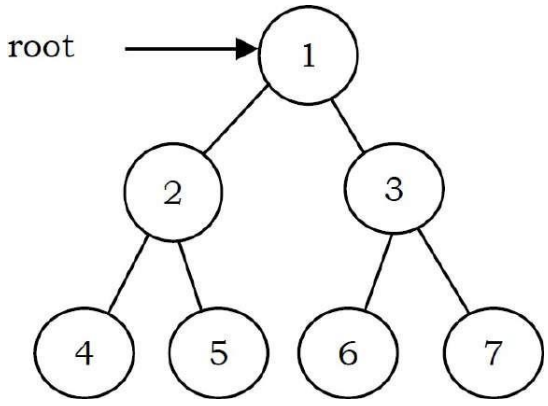
Demikian pula, tidak masalah apakah L di sisi kanan D atau R di sisi kanan D. Karena ini, total 6 kemungkinan dikurangi menjadi 3 dan ini adalah:

- Praorder (DLR) Traversal
- Traversal Inorder (LDR)
- Postorder (LRD) Traversal

Ada metode traversal lain yang tidak bergantung pada perintah di atas dan itu adalah:

- Level Order Traversal: Metode ini terinspirasi dari Breadth First Traversal (algoritma BFS of Graph).

Mari kita gunakan diagram di bawah ini untuk pembahasan selanjutnya.



Gambar 6.12 diagram akar pohon

Praorder Traversal

Dalam traversal preorder, setiap node diproses sebelum (pre) salah satu dari subpohon-nya. Ini adalah traversal paling sederhana untuk dipahami. Namun, meskipun setiap node diproses sebelum subpohon, masih memerlukan beberapa informasi yang harus dipertahankan saat bergerak ke bawah pohon. Pada contoh di atas, 1 diproses terlebih dahulu, kemudian subpohon kiri, dan ini diikuti oleh subpohon kanan.

Oleh karena itu, pemrosesan harus kembali ke subpohon kanan setelah menyelesaikan pemrosesan subpohon kiri. Untuk pindah ke subpohon kanan setelah memproses subpohon kiri, kita harus menjaga informasi root. ADT yang jelas untuk informasi tersebut adalah tumpukan. Karena struktur LIFO-nya, dimungkinkan untuk mendapatkan informasi tentang subpohon kanan kembali dalam urutan terbalik.

Praorder traversal didefinisikan sebagai berikut:

- Kunjungi akarnya.
- Lintasi subpohon kiri di Praorder.
- Lintasi subpohon kanan di Praorder.

Node pohon akan dikunjungi dalam urutan: 1 2 4 5 3 6 7

```
void PreOrder(struct BinaryTreeNode *root){
    if(root) {
        printf("%d",root→data);
        PreOrder(root→left);
        PreOrder (root→right);
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Traversal Preorder Non-Rekursif

Dalam versi rekursif, tumpukan diperlukan karena kita perlu mengingat simpul saat ini sehingga setelah menyelesaikan subpohon kiri kita bisa pergi ke subpohon kanan. Untuk mensimulasikan hal yang sama, pertama-tama kita memproses node saat ini dan sebelum pergi ke subpohon kiri, kita menyimpan node saat ini di stack. Setelah menyelesaikan pemrosesan subpohon kiri, keluarkan elemen dan pergi ke subpohon kanannya. Lanjutkan proses ini sampai tumpukan kosong.

```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            //Process current node
            printf("%d",root→data);
            Push(S,root);
            //If left subtree exists, add to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Traversal InOrder

Dalam Inorder Traversal, root dikunjungi di antara subpohon. Traversal inorder didefinisikan sebagai berikut:

- Lintasi subpohon kiri di Inorder.
- Kunjungi akarnya.
- Lintasi subpohon kanan di Inorder.

Node pohon akan dikunjungi dalam urutan: 4 2 5 1 6 3 7

```
void InOrder(struct BinaryTreeNode *root){
    if(root) {
        InOrder(root→left);
        printf("%d",root→data);
        InOrder(root→right);
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Traversal Inorder Non-Rekursif

Versi non-rekursif dari Inorder traversal mirip dengan Preorder. Satu-satunya perubahan adalah, alih-alih memproses simpul sebelum pergi ke subpohon kiri, memprosesnya setelah muncul (yang ditunjukkan setelah selesainya pemrosesan subpohon kiri)

```
void InOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            Push(S,root);
            //Got left subtree and keep on adding to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        printf("%d", root→data);   //After popping, process the current node
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Traversal PostOrder

Dalam traversal postorder, root dikunjungi setelah kedua subpohon. Traversal postorder didefinisikan sebagai berikut:

- Lintasi subpohon kiri di Postorder.
- Lintasi subpohon kanan di Postorder.
- Kunjungi akarnya.

Node pohon akan dikunjungi dalam urutan: 4 5 2 6 7 3 1

```
void PostOrder(struct BinaryTreeNode *root){
    if(root)        {
            PostOrder(root→left);
            PostOrder(root→right);
            printf("%d",root→data);
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Traversal Postorder Non-Rekursif

Dalam traversal preorder dan inorder, setelah memunculkan elemen stack kita tidak perlu mengunjungi vertex yang sama lagi. Namun dalam traversal postorder, setiap node dikunjungi dua kali. Artinya, setelah memproses subpohon kiri kita akan mengunjungi node saat ini dan setelah memproses subpohon kanan kita akan mengunjungi node saat ini yang sama. Tapi kita harus memproses node selama kunjungan kedua. Di sini masalahnya adalah bagaimana membedakan apakah kita kembali dari subpohon kiri atau subpohon kanan.

Kita menggunakan variabel sebelumnya untuk melacak node yang dilalui sebelumnya. Mari kita asumsikan saat ini adalah simpul saat ini yang ada di atas tumpukan. Ketika sebelumnya adalah induk saat ini, Kita melintasi pohon. Dalam hal ini, Kita mencoba untuk

melintasi ke anak kiri saat ini jika tersedia (mis., Dorong anak kiri ke tumpukan). Jika tidak tersedia, Kita melihat anak kanan saat ini. Jika anak kiri dan kanan tidak ada (yaitu, arus adalah simpul daun), Kita mencetak nilai arus dan mengeluarkannya dari tumpukan.

Jika prev adalah anak kiri saat ini, Kita melintasi pohon dari kiri. Kita melihat anak kanan saat ini. Jika tersedia, lalu telusuri anak kanan (yaitu, dorong anak kanan ke tumpukan); jika tidak, cetak nilai saat ini dan keluarkan dari tumpukan. Jika sebelumnya adalah anak kanan saat ini, Kita melintasi pohon dari kanan. Dalam hal ini, Kita mencetak nilai saat ini dan mengeluarkannya dari tumpukan.

```
void PostOrderNonRecursive(struct BinaryTreeNode *root) {
   struct SimpleArrayStack *S = CreateStack();
   struct BinaryTreeNode *previous = NULL;
   do{
      while (root!=NULL){
         Push(S, root);
         root = root->left;
      }
      while(root == NULL && !IsEmptyStack(S)){
         root = Top(S);
         if(root->right == NULL || root->right == previous){
            printf("%d ", root->data);
            Pop(S);
            previous = root;
            root = NULL;
         }
         else
            root = root->right;
      }
   }while(!IsEmptyStack(S));
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Lintasan Urutan Level

Level order traversal didefinisikan sebagai berikut:

- Kunjungi akarnya.
- Saat melintasi level (, pertahankan semua elemen pada level (+ 1 dalam antrian.
- Pergi ke level berikutnya dan kunjungi semua node di level itu.
- Ulangi ini sampai semua level selesai.

Node pohon dikunjungi dalam urutan: 1 2 3 4 5 6 7

```
void LevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q = CreateQueue();
    if(!root)
        return;
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        //Process current node
        printf("%d", temp→data);
        if(temp→left)
            EnQueue(Q, temp→left);
        if(temp→right)
            EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n). Karena, dalam kasus terburuk, semua node di seluruh level terakhir bisa berada dalam antrian secara bersamaan.

Pohon Biner: Masalah & Solusi

Soal-1 Berikan algoritma untuk mencari elemen maksimum dalam pohon biner.

Solusi: Salah satu cara sederhana untuk menyelesaikan masalah ini adalah: temukan elemen maksimum di subpohon kiri, temukan elemen maksimum di subpohon kanan, bandingkan dengan data akar dan pilih salah satu yang memberikan nilai maksimum. Pendekatan ini dapat dengan mudah diimplementasikan dengan rekursi.

```
int FindMax(struct BinaryTreeNode  *root) {
    int root_val, left, right, max = INT_MIN;
    if(root !=NULL) {
        root_val = root→data;
        left = FindMax(root→left);
        right = FindMax(root→right);
        // Find the largest of the three values.
        if(left > right)
            max = left;
        else max = right;
        if(root_val > max)
            max = root_val;
    }
    return max;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-2 Berikan algoritma untuk mencari elemen maksimum dalam pohon biner tanpa rekursi.

Solusi: Menggunakan traversal urutan level: cukup amati data elemen saat menghapus.

```
int FindMaxUsingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    int max = INT_MIN;
    struct Queue *Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        // largest of the three values
        if(max < temp→data)
            max =  temp→data;
        if(temp→left)
            EnQueue (Q, temp→left);
        if(temp→right)
            EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return max;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-3 Berikan algoritma untuk mencari elemen dalam pohon biner.

Solusi: Diberikan pohon biner, kembalikan true jika simpul dengan data ditemukan di pohon. Recurse down tree, pilih cabang kiri atau kanan dengan membandingkan data dengan data masing-masing node.

```
int FindInBinaryTreeUsingRecursion(struct BinaryTreeNode *root, int data) {
    int temp;
    // Base case == empty tree, in that case, the data is not found so return false
    if(root == NULL)
        return 0;
    else {
            //see if found here
             if(data == root→data)
                    return 1;
             else {
                    // otherwise recur down the correct subtree
                     temp = FindInBinaryTreeUsingRecursion (root→left, data)
                     if(temp != 0)
                          return temp;
                     else return(FindInBinaryTreeUsingRecursion(root→right, data));
             }
    }
    return 0;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-4 Berikan algoritma untuk mencari elemen dalam pohon biner tanpa rekursi.

Solusi: Kita dapat menggunakan level order traversal untuk menyelesaikan masalah ini. Satu-satunya perubahan yang diperlukan dalam traversal urutan level adalah, alih-alih mencetak data, kita hanya perlu memeriksa apakah data root sama dengan elemen yang ingin kita cari.

```
int SearchUsingLevelOrder(struct BinaryTreeNode *root, int data){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    if(!root) return -1;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            //see if found here
            if(data == root→data)
                return 1;
            if(temp→left)
                    EnQueue (Q, temp→left);
            if(temp→right)
                    EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return 0;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-5 Berikan algoritma untuk memasukkan elemen ke dalam pohon biner.

Solusi: Karena pohon yang diberikan adalah pohon biner, kita dapat menyisipkan elemen di mana pun kita mau. Untuk menyisipkan elemen, kita dapat menggunakan traversal urutan level dan menyisipkan elemen di mana pun kita menemukan simpul yang anak kiri atau kanannya NULL.

```
void InsertInBinaryTree(struct BinaryTreeNode *root, int data){
    struct Queue *Q;
    struct BinaryTreeNode *temp;
    struct BinaryTreeNode *newNode;
    newNode = (struct BinaryTreeNode *) malloc(sizeof(struct BinaryTreeNode));
    newNode→left = newNode→right = NULL;
     if(!newNode) {
          printf("Memory Error"); return;
     }
    if(!root) {
          root = newNode;
          return;
    }
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
              temp = DeQueue(Q);
              if(temp→left)
                        EnQueue(Q, temp→left);
              else {
                        temp→left=newNode;
                        DeleteQueue(Q);
                        return;
              }
              if(temp→right)
                        EnQueue(Q, temp→right);
              else {
                        temp→right=newNode;
                        DeleteQueue(Q);
                        return;
              }
    }
    DeleteQueue(Q);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-6 Berikan algoritma untuk mencari ukuran pohon biner.

Solusi: Hitung ukuran subpohon kiri dan kanan secara rekursif, tambahkan 1 (simpul saat ini) dan kembali ke induknya.

```
// Compute the number of nodes in a tree.
int SizeOfBinaryTree(struct BinaryTreeNode *root) {
    if(root==NULL)
        return 0;
    else  return(SizeOfBinaryTree(root→left) + 1 + SizeOfBinaryTree(root→right));
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-7 Bisakah kita menyelesaikan Soal-6 tanpa rekursi?

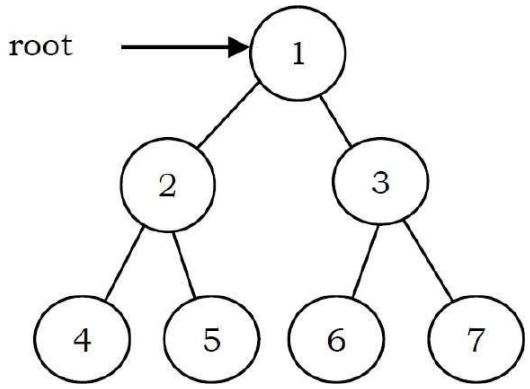
Solusi: Ya, menggunakan level order traversal.

```
int SizeofBTUsingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root)  return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            count++;
            if(temp→left)
                    EnQueue (Q, temp→left);
            if(temp→right)
                    EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return count;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-8 Berikan algoritma untuk mencetak data urutan level dalam urutan terbalik. Misalnya, output untuk pohon di bawah ini harus: 4 5 6 7 2 3 1



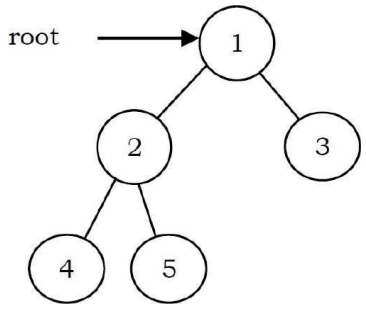
Solusi:

```
void LevelOrderTraversalInReverse(struct BinaryTreeNode *root){
    struct Queue *Q;
    struct Stack *s = CreateStack();
    struct BinaryTreeNode *temp;
    if(!root) return;
    Q = CreateQueue();
    EnQueue(Q, root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            if(temp→right)
                EnQueue(Q, temp→right);
            if(temp→left)
                EnQueue (Q, temp→left);
            Push(s, temp);
    }
    while(!IsEmptyStack(s))
        printf("%d",Pop(s)→data);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-9 Berikan algoritma untuk menghapus pohon.

Solusi:



Untuk menghapus pohon, kita harus melintasi semua simpul pohon dan menghapusnya satu per satu. Jadi traversal mana yang harus kita gunakan: Inorder, Preorder, Postorder atau Level order Traversal?

Sebelum menghapus node induk kita harus menghapus node anak-anaknya terlebih dahulu. Kita dapat menggunakan traversal postorder karena ia bekerja tanpa menyimpan apa pun. Kita dapat menghapus pohon dengan traversal lain juga dengan kompleksitas ruang ekstra. Untuk yang berikut, simpul pohon dihapus secara berurutan – 4,5,2,3,1.

```
void DeleteBinaryTree(struct BinaryTreeNode *root){
    if(root == NULL)
        return;
    /* first delete both subtrees */
    DeleteBinaryTree(root→left);
    DeleteBinaryTree(root→right);
    //Delete current node only after deleting subtrees
    free(root);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-10 Berikan algoritma untuk mencari tinggi (atau kedalaman) pohon biner.

Solusi: Hitung secara rekursif tinggi subpohon kiri dan kanan dari sebuah simpul dan tetapkan tinggi ke simpul tersebut sebagai maksimum dari ketinggian dua anak ditambah 1. Ini mirip dengan traversal pohon PreOrder (dan algoritma DFS dari Graph).

```
int HeightOfBinaryTree(struct BinaryTreeNode   *root){
    int leftheight, rightheight;
    if(root == NULL)
        return 0;
    else {
            /* compute the depth of each subtree */
             leftheight = HeightOfBinaryTree(root→left);
             rightheight = HeightOfBinaryTree(root→right);

             if(leftheight > rightheight)
                  return(leftheight + 1);
             else
                return(rightheight + 1);
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-11 Bisakah kita menyelesaikan Soal-10 tanpa rekursi?

Solusi: Ya, menggunakan level order traversal. Ini mirip dengan algoritma BFS of Graph. Akhir level diidentifikasi dengan NULL.

```
int FindHeightofBinaryTree(struct BinaryTreeNode *root){
    int level = 0;
    struct Queue *Q;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    // End of first level
    EnQueue(Q,NULL);
    while(!IsEmptyQueue(Q)) {
            root=DeQueue(Q);
            // Completion of current level.
            if(root==NULL) {
                    //Put another marker for next level.
                    if(!IsEmptyQueue(Q))
                       EnQueue(Q,NULL);
                    level++;
            }
            else {  if(root→left)
                       EnQueue(Q, root→left);
                    if(root→right)
                       EnQueue(Q, root→right);
            }
    }
    return level;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-12 Berikan algoritma untuk menemukan simpul terdalam dari pohon biner.

Solusi:

```
struct BinaryTreeNode *DeepestNodeinBinaryTree(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    if(!root) return NULL;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            if(temp→left)
               EnQueue(Q, temp→left);
            if(temp→right)
               EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
    return temp;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-13 Berikan algoritma untuk menghapus elemen (dengan asumsi data diberikan) dari pohon biner.

Solusi: Penghapusan node di pohon biner dapat diimplementasikan sebagai

- Mulai dari root, cari node yang ingin kita hapus.
- Temukan simpul terdalam di pohon.

- Ganti data node terdalam dengan node yang akan dihapus.
- Kemudian hapus simpul terdalam.

Soal-14 Berikan algoritma untuk mencari jumlah daun pada pohon biner tanpa menggunakan rekursi.

Solusi: Himpunan simpul yang anak kiri dan kanannya NULL disebut simpul daun.

```
int NumberOfLeavesInBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            if(!temp→left && !temp→right)
                    count++;
            else {  if(temp→left)
                        EnQueue(Q, temp→left);
                    if(temp→right)
                        EnQueue(Q, temp→right);
    }
    }
    DeleteQueue(Q);
    return count;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-15 Berikan algoritma untuk menemukan jumlah simpul penuh dalam pohon biner tanpa menggunakan rekursi.

Solusi: Himpunan semua node dengan anak kiri dan kanan disebut node penuh.

```
int NumberOfFullNodesInBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root)
        return 0;
```

```
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            if(temp→left && temp→right)
                    count++;
            if(temp→left)
                    EnQueue (Q, temp→left);
            if(temp→right)
                    EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return count;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-16 Berikan algoritma untuk menemukan jumlah setengah node (node dengan hanya satu anak) di pohon biner tanpa menggunakan rekursi.

Solusi: Himpunan semua node dengan anak kiri atau kanan (tetapi tidak keduanya) disebut setengah node.

```
int NumberOfHalfNodesInBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int count = 0;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            //we can use this condition also instead of two temp→left ^ temp→right
            if(!temp→left && temp→right || temp→left && !temp→right)
                count++;
            if(temp→left)
                EnQueue (Q, temp→left);
            if(temp→right)
                EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return count;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-17 Diberikan dua pohon biner, kembalikan true jika mereka secara struktural identik.

Solusi:

Algoritma:

- Jika kedua pohon NULL maka kembalikan true.

- Jika kedua pohon bukan NULL, maka bandingkan data dan periksa struktur subpohon kiri dan kanan secara rekursif.

```
//Return true if they are structurally identical.
int AreStructurullySameTrees(struct BinaryTreeNode *root1, struct BinaryTreeNode *root2) {
    // both empty→1
    if(root1==NULL && root2==NULL)
            return 1;
    if(root1==NULL || root2==NULL)
            return 0;
    //  both non-empty→compare them
    return(root1→data == root2→data && AreStructurullySameTrees(root1→left, root2→left) &&
            AreStructurullySameTrees(root1→right, root2→right));
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk tumpukan rekursif.

Soal-18 Berikan algoritma untuk mencari diameter pohon biner. Diameter pohon (kadang-kadang disebut lebar) adalah jumlah simpul pada jalur terpanjang antara dua daun dalam pohon.

Solusi: Untuk mencari diameter pohon, pertama hitung diameter subpohon kiri dan subpohon kanan secara rekursif. Di antara dua nilai ini, kita perlu mengirim nilai maksimum bersama dengan level saat ini (+1).

```
int DiameterOfTree(struct BinaryTreeNode *root, int *ptr){
    int left, right;
    if(!root)
        return 0;
    left = DiameterOfTree(root→left, ptr);
    right = DiameterOfTree(root→right, ptr);
    if(left + right > *ptr)
        *ptr = left + right;
    return Max(left, right)+1;
}
```

```
//Alternative Coding
static int diameter(struct BinaryTreeNode *root) {
    if (root == NULL)
            return 0;
    int lHeight = height(root->eft);
    int rHeight = height(root-right);
    int lDiameter = diameter(root-left);
    int rDiameter = diameter(root-right);
    return max(lHeight + rHeight + 1, max(lDiameter, rDiameter));

}
/*  The function Compute the "height" of a tree. Height is the number of nodes along
the longest path from the root node down to the farthest leaf node.*/
static int  height(Node root) {
    if (root == null)
            return 0;
    return 1 + max(height(root.left), height(root.right));
}
```

Ada solusi lain dan kompleksitasnya adalah O(n). Ide utama dari pendekatan ini adalah bahwa simpul menyimpan diameter maksimum anak kiri dan anak kanannya jika anak simpul adalah "root", oleh karena itu, tidak perlu memanggil metode ketinggian secara rekursif. Kekurangannya adalah kita perlu menambahkan dua variabel tambahan dalam struktur node.

```
int findMaxLen(Node root) {
    int nMaxLen = 0;
    if (root == null)
        return 0;
    if (root.left == null)
        root.nMaxLeft = 0;
    if (root.right == null)
        root.nMaxRight = 0;
    if (root.left != null)
        findMaxLen(root.left);
    if (root.right != null)
        findMaxLen(root.right);
```

```
    if (root.left != null) {
            int nTempMaxLen = 0;
            nTempMaxLen = (root.left.nMaxLeft > root.left.nMaxRight) ?
                    root.left.nMaxLeft : root.left.nMaxRight;
            root.nMaxLeft = nTempMaxLen + 1;
    }
    if (root.right != null) {
            int nTempMaxLen = 0;
            nTempMaxLen = (root.right.nMaxLeft > root.right.nMaxRight) ?
                    root.right.nMaxLeft : root.right.nMaxRight;
            root.nMaxRight = nTempMaxLen + 1;
    }
    if (root.nMaxLeft + root.nMaxRight > nMaxLen)
            nMaxLen = root.nMaxLeft + root.nMaxRight;
    return nMaxLen;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-19 Berikan algoritma untuk menemukan level yang memiliki jumlah maksimum dalam pohon biner.

Solusi: Logikanya sangat mirip dengan menemukan jumlah level. Satu-satunya perubahan adalah, kita perlu melacak jumlahnya juga.

```
int FindLevelwithMaxSum(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    int level=0, maxLevel=0;
    struct Queue *Q;
    int currentSum = 0, maxSum = 0;
    if(!root)
            return 0;
    Q=CreateQueue();
    EnQueue(Q,root);
    EnQueue(Q,NULL);                    //End of first level.
    while(!IsEmptyQueue(Q)) {
            temp =DeQueue(Q);
```

```
            // If the current level is completed then compare sums
            if(temp == NULL) {
                    if(currentSum> maxSum) {
                            maxSum = currentSum;
                            maxLevel = level;
                    }
                    currentSum = 0;
                    //place the indicator for end of next level at the end of queue
                    if(!IsEmptyQueue(Q))
                          EnQueue(Q,NULL);
                    level++;
            }
            else {
                    currentSum   += temp→data;
                    if(temp→left)
                            EnQueue(temp, temp→left);
                    if(root→right)
                            EnQueue(temp, temp→right);
            }
    }
    return maxLevel;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-20 Diberikan pohon biner, cetak semua jalur akar-ke-daunnya.

Solusi: Lihat komentar di functions.php

```
void PrintPathsRecur(struct BinaryTreeNode *root, int path[], int pathLen) {
    if(root ==NULL)
        return;
    // append this node to the path array
    path[pathLen] = root→data;
    pathLen++;
    // it's a leaf, so print the path that led to here
    if(root→left==NULL && root→right==NULL)
        PrintArray(path, pathLen);
    else {
        // otherwise try both subtrees
        PrintPathsRecur(root→left, path, pathLen);
        PrintPathsRecur(root→right, path, pathLen);
    }
}
// Function that prints out an array on a line.
void PrintArray(int ints[], int len) {
    for (int i=0; i<len; i++)
        printf("%d",ints[i]);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk tumpukan rekursif.

Soal-21 Berikan algoritma untuk memeriksa keberadaan jalur dengan jumlah yang diberikan. Itu berarti, jika diberi jumlah, periksa apakah ada jalur dari root ke salah satu node.

Solusi: Untuk masalah ini, strateginya adalah: kurangi nilai simpul dari jumlah sebelum memanggil anak-anaknya secara rekursif, dan periksa untuk melihat apakah jumlahnya 0 ketika kita kehabisan pohon.

```
void PrintPathsRecur(struct BinaryTreeNode *root, int path[], int pathLen) {
    if(root ==NULL)
        return;
    // append this node to the path array
    path[pathLen] = root→data;
    pathLen++;
    // it's a leaf, so print the path that led to here
    if(root→left==NULL && root→right==NULL)
        PrintArray(path, pathLen);
    else {
        // otherwise try both subtrees
        PrintPathsRecur(root→left, path, pathLen);
        PrintPathsRecur(root→right, path, pathLen);
    }
}
// Function that prints out an array on a line.
void PrintArray(int ints[], int len) {
    for (int i=0; i<len; i++)
        printf("%d",ints[i]);
}
        if((root→left && root→right)||(!root→left && !root→right))
                return(HasPathSum(root→left, remainingSum) ||
                        HasPathSum(root→right, remainingSum));
        else if(root→left)
            return HasPathSum(root→left, remainingSum);
        else        return HasPathSum(root→right, remainingSum);
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-22 Berikan algoritma untuk mencari jumlah semua elemen dalam pohon biner.

Solusi: Secara rekursif, panggil jumlah subpohon kiri, jumlah subpohon kanan dan tambahkan nilainya ke data node saat ini.

```
int Add(struct BinaryTreeNode  *root) {
    if(root == NULL)
        return 0;
    else return (root→data + Add(root→left) +  Add(root→right));
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-23 Bisakah kita menyelesaikan Soal-22 tanpa rekursi?

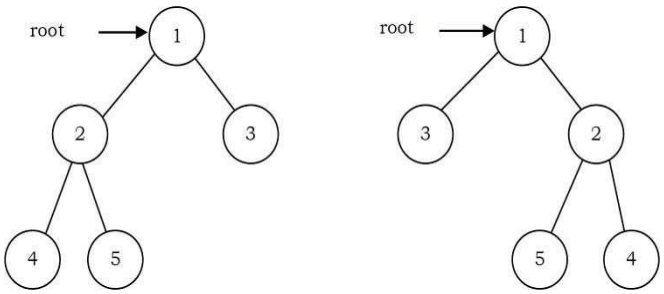
Solusi: Kita bisa menggunakan level order traversal dengan perubahan sederhana. Setiap kali setelah menghapus elemen dari antrian, tambahkan nilai data node ke variabel jumlah.

```
int SumofBTusingLevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    int sum = 0;
    if(!root)
        return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        sum += temp→data;
        if(temp→left)
            EnQueue (Q, temp→left);
        if(temp→right)
            EnQueue (Q, temp→right);
    }
    DeleteQueue(Q);
    return sum;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-24 Berikan algoritma untuk mengubah pohon menjadi cerminnya. Cermin pohon adalah pohon lain dengan anak kiri dan kanan dari semua simpul non-daun dipertukarkan. Pohon-pohon di bawah adalah cermin satu sama lain.



Solusi:

```
struct BinaryTreeNode *MirrorOfBinaryTree(struct BinaryTreeNode *root){
    struct BinaryTreeNode * temp;
    if(root) {
        MirrorOfBinaryTree(root→left);
        MirrorOfBinaryTree(root→right);
        /* swap the pointers in this node */
        temp  = root→left;
        root→left  = root→right;
        root→right = temp;
    }
    return root;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-25 Diberikan dua pohon, berikan algoritma untuk memeriksa apakah keduanya merupakan cermin satu sama lain.

Solusi:

```
int AreMirrors(struct BinaryTreeNode  * root1, struct BinaryTreeNode  * root2) {
    if(root1 == NULL && root2 == NULL)
       return 1;
    if(root1 == NULL || root2 == NULL)
       return 0;
    if(root1→data != root2→data)
       return 0;
    else return AreMirrors(root1→left, root2→right) && AreMirrors(root1→right, root2→left);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-26 Berikan algoritma untuk menemukan LCA (Least Common Ancestor) dari dua node dalam Pohon Biner.

Solusi:

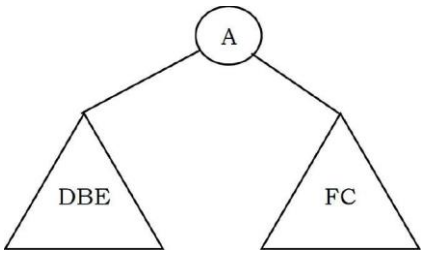
```
struct BinaryTreeNode *LCA(struct BinaryTreeNode *root, struct BinaryTreeNode *α,
                           struct BinaryTreeNode *β){
    struct BinaryTreeNode *left, *right;
    if(root == NULL)
       return root;
    if(root == α || root == β)
       return root;
    left = LCA (root→left, α, β );
    right = LCA (root→right, α, β );
    if(left && right)
       return root;
    else return (left? left: right)
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n) untuk rekursi.

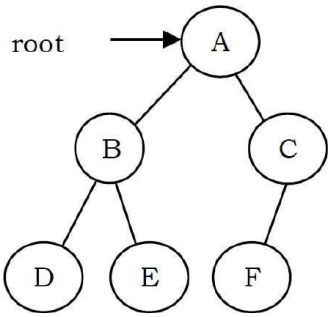
Soal-27 Berikan algoritma untuk membangun pohon biner dari traversal Inorder dan Preorder yang diberikan.

Solusi Mari kita perhatikan traversal di bawah ini:

Urutan berurutan: D B E A F C
Urutan preorder: A B D E C F



Dalam urutan Preorder, elemen paling kiri menunjukkan akar pohon. Jadi kita tahu 'A' adalah akar untuk urutan yang diberikan. Dengan mencari 'A' dalam urutan Inorder kita dapat menemukan semua elemen di sisi kiri 'A', yang berada di bawah subpohon kiri, dan elemen di sisi kanan 'A', yang berada di bawah subpohon kanan. Jadi kita mendapatkan struktur seperti yang terlihat di bawah ini. Kita secara rekursif mengikuti langkah-langkah di atas dan mendapatkan pohon berikut.



Algoritma: BuildTree()

1. Pilih elemen dari Praorder. Tambahkan variabel indeks Preorder (preOrderIndex dalam kode di bawah) untuk memilih elemen berikutnya dalam panggilan rekursif berikutnya.
2. Buat simpul pohon baru (newNode) dari tumpukan dengan data sebagai elemen yang dipilih.
3. Temukan indeks elemen yang dipilih di Inorder. Biarkan indeks menjadi inOrderIndex.
4. Panggil BuildBinaryTree untuk elemen sebelum inOrderIndex dan jadikan pohon yang dibangun sebagai subpohon kiri dari newNode.
5. Panggil BuildBinaryTree untuk elemen setelah inOrderIndex dan jadikan pohon yang dibangun sebagai subpohon kanan dari newNode.
6. mengembalikan newNode.js

```
struct BinaryTreeNode *BuildBinaryTree(int inOrder[], int preOrder[], int inOrderStart, int inOrderEnd){
    static int preOrderIndex = 0;
    struct BinaryTreeNode *newNode;
    if(inOrderStart > inOrderEnd)
        return NULL;
    newNode = (struct BinaryTreeNode *) malloc (sizeof(struct BinaryTreeNode));
    if(!newNode) {
        printf("Memory Error");
        return NULL;
    }
    // Select current node from Preorder traversal using preOrderIndex
    newNode→data = preOrder[preOrderIndex];
    preOrderIndex++;
    if(inOrderStart == inOrderEnd)
        return newNode;
    // find the index of this node in Inorder traversal
    int inOrderIndex = Search(inOrder, inOrderStart, inOrderEnd, newNode→data);
    //Fill the left and right subtrees using index in Inorder traversal
    newNode→left = BuildBinaryTree(inOrder, preOrder, inOrderStart, inOrderIndex -1);
    newNode→right = BuildBinaryTree(inOrder, preOrder, inOrderIndex +1, inOrderEnd);
    return newNode;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-28 Jika kita diberikan dua urutan traversal, dapatkah kita membangun pohon biner secara unik?

Solusi: Tergantung pada traversal apa yang diberikan. Jika salah satu metode traversal adalah Inorder maka pohon dapat dibangun secara unik, sebaliknya tidak.

Oleh karena itu, kombinasi berikut dapat secara unik mengidentifikasi pohon:

- Inorder dan Preorder
- Inorder dan Postorder
- Inorder dan Level-order

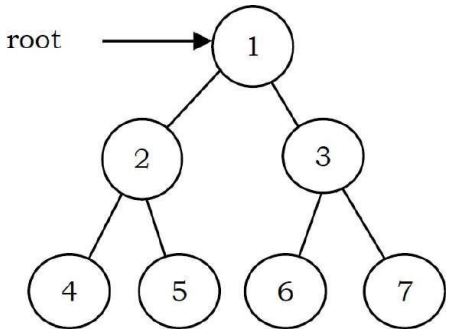
Kombinasi berikut tidak secara unik mengidentifikasi pohon.

- Postorder dan Preorder
- Preorder dan Level-order
- Postorder dan Level-order

Misalnya, traversal Preorder, Level-order, dan Postorder sama untuk pohon di atas:

Jadi, bahkan jika tiga dari mereka (PreOrder, Level-Order dan PostOrder) diberikan, pohon tidak dapat dibangun secara unik.

Soal-29 Berikan algoritma untuk mencetak semua ancestor dari sebuah node dalam pohon Binary. Untuk pohon di bawah, untuk 7 nenek moyangnya adalah 1 3 7.

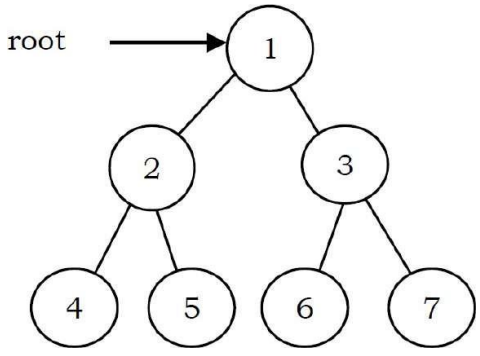


Solusi: Selain Depth First Search dari pohon ini, kita dapat menggunakan cara rekursif berikut untuk mencetak ancestor.

```
int PrintAllAncestors(struct BinaryTreeNode *root, struct BinaryTreeNode *node){
    if(root == NULL) return 0;
    if(root→left == node || root→right == node || PrintAllAncestors(root→left, node) ||
                                                  PrintAllAncestors(root→right, node)) {
        printf("%d",root→data);
        return 1;
    }
    return 0;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n) untuk rekursi.

Soal-30 **Zigzag Tree Traversal:** Berikan algoritma untuk melintasi pohon biner dalam urutan Zigzag. Misalnya, output untuk pohon di bawah ini adalah: 1 3 2 4 5 6 7



Solusi: Masalah ini dapat diselesaikan dengan mudah menggunakan dua tumpukan. Asumsikan dua tumpukan adalah: currentLevel dan nextLevel. Kita juga

membutuhkan variabel untuk melacak urutan level saat ini (apakah itu dari kiri ke kanan atau kanan ke kiri).

Kita muncul dari tumpukan Level saat ini dan mencetak nilai simpul. Setiap kali urutan level saat ini dari kiri ke kanan, dorong anak kiri simpul, lalu anak kanannya, untuk menumpuk Level berikutnya. Karena tumpukan adalah struktur *Last In First Out* (LIFO), saat berikutnya node dikeluarkan dari Level berikutnya, urutannya akan terbalik.

Di sisi lain, ketika urutan level saat ini dari kanan ke kiri, Kita akan mendorong anak kanan node terlebih dahulu, lalu anak kirinya. Terakhir, jangan lupa untuk menukar dua tumpukan itu di akhir setiap level (yaitu, saat Level saat ini kosong).

```
void ZigZagTraversal(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    int leftToRight = 1;
    if(!root)
        return;
    struct Stack *currentLevel = CreateStack(), *nextLevel = CreateStack();
    Push(currentLevel, root);
    while(!IsEmptyStack(currentLevel)) {
            temp = Pop(currentLevel);
            if(temp) {
                    printf("%d",temp→data);
                    if(leftToRight) {
                            if(temp→left)   Push(nextLevel, temp→left);
                            if(temp→right) Push(nextLevel, temp→right);
                    }
                    else {    if(temp→right) Push(nextLevel, temp→right);
                            if(temp→left)    Push(nextLevel, temp→left);
                    }
            }
            if(IsEmptyStack(currentLevel)) {
                    leftToRight = 1-leftToRight;
                    swap(currentLevel, nextLevel);
            }
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: Ruang untuk dua tumpukan = O(n) + O(n) = O(n).

Soal-31 Berikan algoritma untuk mencari jumlah vertikal pohon biner. Misalnya, Pohon itu memiliki 5 garis vertikal

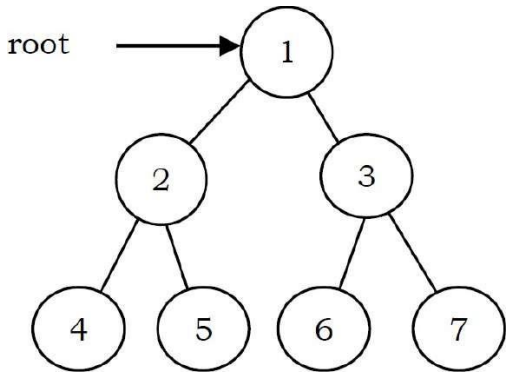
Vertikal-1: node-4 => jumlah vertikal adalah 4

Vertikal-2: node-2 => jumlah vertikal adalah 2

Vertikal-3: node-1,5,6 => jumlah vertikal adalah 1 + 5 + 6 = 12

Vertikal-4: node-3 => jumlah vertikal adalah 3

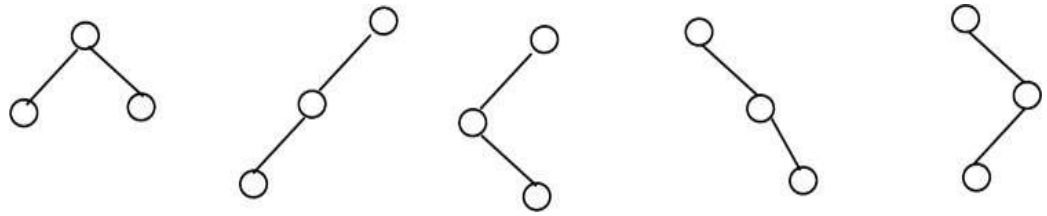
Vertical-5: node-7 => jumlah vertikal adalah 7 perlu menampilkan: 4 2 12 3 7



Solusi: Kita bisa melakukan inorder traversal dan hash kolom. Kita memanggil VerticalSumInBinaryTreefroot, 0) yang berarti root berada di kolom 0. Saat melakukan traversal, hash kolom dan tingkatkan nilainya dengan root → data.

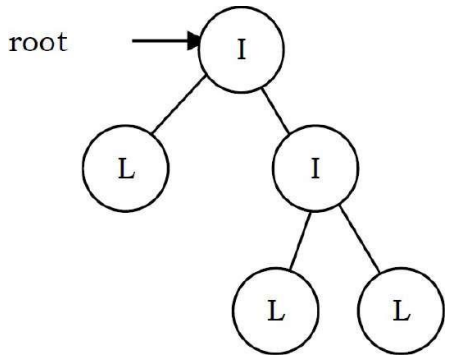
```
void VerticalSumInBinaryTree (struct BinaryTreeNode *root, int column){
    if(root==NULL)
        return;
    VerticalSumInBinaryTree(root→left, column-1);
    //Refer Hashing chapter for implementation of hash table
    Hash[column] += root→data;
    VerticalSumInBinaryTree(root→right, column+1);
}
VerticalSumInBinaryTree(root, 0);
Print Hash;
```

Soal-32 Berapa banyak pohon biner berbeda yang mungkin dengan n simpul?



Solusi: Sebagai contoh, pertimbangkan sebuah pohon dengan 3 node (n = 3). Ini akan memiliki kombinasi maksimum 5 pohon yang berbeda (yaitu, 2^3 -3 = 5).
Secara umum, jika terdapat n node, maka terdapat 2^n –n pohon yang berbeda.

Soal-33 Diberikan pohon dengan properti khusus di mana daun diwakili dengan 'L' dan simpul internal dengan 'I'. Juga, asumsikan bahwa setiap node memiliki 0 atau 2 anak. Mengingat traversal praorder pohon ini, buat pohonnya.
Contoh: String preorder yang diberikan => ILILL



Solusi: Pertama, kita harus melihat bagaimana traversal preorder diatur. Praorder traversal berarti pertama-tama menempatkan simpul akar, kemudian pra-order traversal dari subpohon kiri dan kemudian pre-order traversal dari subpohon kanan. Dalam skenario normal, tidak mungkin mendeteksi di mana subpohon kiri berakhir dan subpohon kanan mulai hanya menggunakan traversal praorder. Karena setiap node memiliki 2 anak atau tidak ada anak, kita dapat mengatakan bahwa jika sebuah node ada maka saudaranya juga ada. Jadi setiap kali kita menghitung subpohon, kita juga perlu menghitung subpohon saudaranya.

Kedua, setiap kali kita mendapatkan 'L' dalam string input, itu adalah daun dan kita dapat berhenti untuk subpohon tertentu pada titik itu. Setelah simpul 'L' ini (anak kiri dari induknya 'L'), saudaranya dimulai. Jika simpul 'L' adalah anak kanan dari induknya, maka kita perlu naik hierarki untuk menemukan subpohon berikutnya untuk dihitung.

Dengan mengingat invarian di atas, kita dapat dengan mudah menentukan kapan subpohon berakhir dan subpohon berikutnya dimulai. Ini berarti bahwa Kita dapat memberikan simpul awal apa pun ke metode Kita dan itu dapat dengan mudah menyelesaikan subpohon yang dihasilkannya di luar simpulnya. Kita hanya perlu berhati-hati untuk meneruskan node awal yang benar ke sub-pohon yang berbeda.

```
struct BinaryTreeNode *BuildTreeFromPreOrder(char* A, int *i){
    struct BinaryTreeNode *newNode;
    newNode = (struct BinaryTreeNode *) malloc(sizeof(struct BinaryTreeNode));
    newNode→data = A[*i];
    newNode→left = newNode→right = NULL;
    if(A == NULL){                              //Boundary Condition
            free(newNode);
            return NULL;
    }
    if(A[*i] == 'L')                            //On reaching leaf node, return
            return newNode;
    *i = *i + 1;                                //Populate left sub tree
    newNode→left = BuildTreeFromPreOrder(A, i);
    *i = *i + 1;                                //Populate right sub tree
    newNode→right = BuildTreeFromPreOrder(A, i);
    return newNode;
}
```

Kompleksitas Waktu: O(n).

Soal-34 Diberikan pohon biner dengan tiga pointer (kiri, kanan dan nextSibling), berikan algoritma untuk mengisi pointer NextSibling dengan asumsi awalnya NULL.

Solusi: Kita dapat menggunakan antrian sederhana (mirip dengan solusi Soal-11). Mari kita asumsikan bahwa struktur pohon biner adalah:

```
struct BinaryTreeNode {
    struct BinaryTreeNode* left;
    struct BinaryTreeNode* right;
    struct BinaryTreeNode* nextSibling;
};

int FillNextSiblings(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    if(!root)
        return 0;
```

```
        Q = CreateQueue();
        EnQueue(Q,root);
        EnQueue(Q,NULL);
        while(!IsEmptyQueue(Q)) {
                temp =DeQueue(Q);
                // Completion of current level.
                if(temp ==NULL) { //Put another marker for next level.
                        if(!IsEmptyQueue(Q))
                            EnQueue(Q,NULL);
                }
                else {
                        temp→nextSibling = QueueFront(Q);
                        if(root→left)
                            EnQueue(Q, temp→left);
                        if(root→right)
                            EnQueue(Q, temp→right);
                }
        }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-35 Apakah ada cara lain untuk menyelesaikan Soal-34?

Solusi: Triknya adalah menggunakan kembali pointer nextSibling yang telah diisi. Seperti yang disebutkan sebelumnya, kita hanya perlu satu langkah lagi untuk bekerja. Sebelum kita meneruskan kiri dan kanan ke fungsi rekursi itu sendiri, kita menghubungkan nextSibling anak kanan ke anak kiri nextSibling node saat ini. Agar ini berfungsi, penunjuk node nextSibling saat ini harus diisi, yang benar dalam kasus ini.

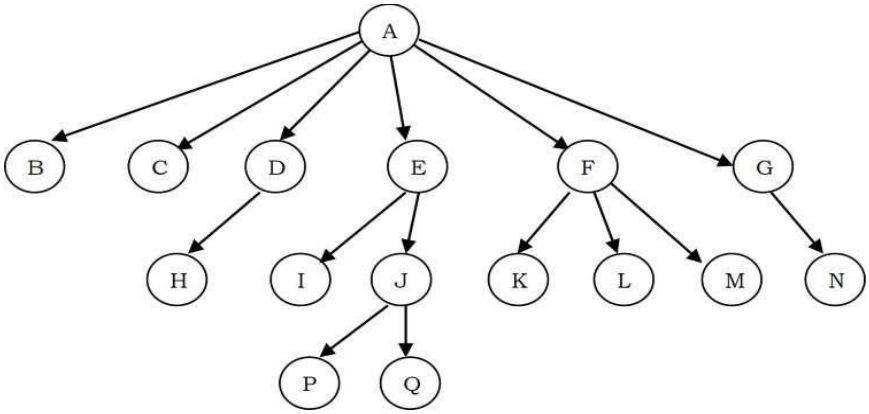
```
void FillNextSiblings(struct BinaryTreeNode* root) {
    if (!root)
        return;
    if (root→left)
        root→left→nextSibling = root→right;
    if (root→right)
        root→right→nextSibling = (root→nextSibling) ? root→nextSibling→left : NULL;
    FillNextSiblings(root→left);
    FillNextSiblings(root→right);
}
```

Kompleksitas Waktu: O(n).

6.7 POHON GENERIK (POHON N-ARY)

Pada bagian sebelumnya kita membahas pohon biner di mana setiap node dapat memiliki maksimal dua anak dan ini diwakili dengan mudah dengan dua pointer. Tetapi misalkan jika kita memiliki sebuah pohon dengan banyak anak pada setiap simpul dan juga jika kita tidak mengetahui berapa banyak anak yang dapat dimiliki sebuah simpul, bagaimana kita merepresentasikannya?

Sebagai contoh, perhatikan pohon yang ditunjukkan di bawah ini.



Gambar 6.13 pohon generik

Bagaimana kita merepresentasikan pohon?

Pada pohon di atas, terdapat simpul dengan 6 anak, dengan 3 anak, dengan 2 anak, dengan 1 anak, dan dengan nol anak (daun). Untuk menyajikan pohon ini kita harus mempertimbangkan kasus terburuk (6 anak) dan mengalokasikan banyak pointer anak untuk setiap node. Berdasarkan ini, representasi node dapat diberikan sebagai:

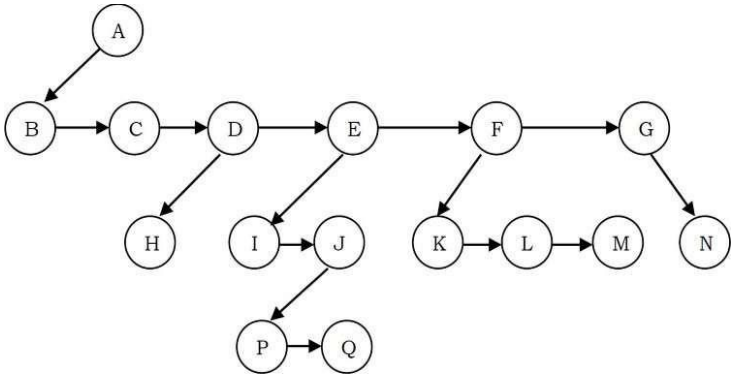
```
struct TreeNode{
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *secondChild;
    struct TreeNode *thirdChild;
    struct TreeNode *fourthChild;
    struct TreeNode *fifthChild;
    struct TreeNode *sixthChild;
};
```

Karena kita tidak menggunakan semua pointer dalam semua kasus, ada banyak pemborosan memori. Masalah lain adalah bahwa kita tidak mengetahui jumlah anak untuk setiap node terlebih dahulu. Untuk mengatasi masalah ini kita membutuhkan representasi yang meminimalkan pemborosan dan juga menerima node dengan sejumlah anak.

Representasi Pohon Generik

Karena tujuan Kita adalah untuk mencapai semua simpul pohon, solusi yang mungkin untuk ini adalah sebagai berikut:

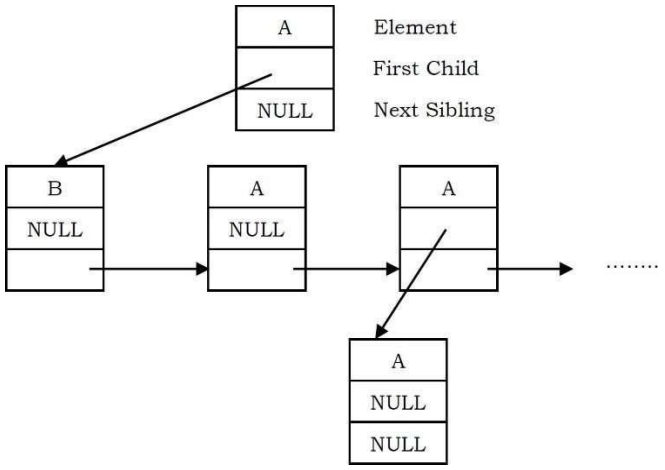
- Pada setiap simpul, hubungkan anak-anak dari orang tua (saudara kandung) yang sama dari kiri ke kanan.
- Hapus link dari orang tua ke semua anak kecuali anak pertama.



Gambar 6.14 Representasi Pohon Generik

Apa yang dikatakan pernyataan di atas adalah jika kita memiliki hubungan antara anak-anak maka kita tidak memerlukan tautan tambahan dari orang tua ke semua anak. Ini karena kita dapat melintasi semua elemen dengan memulai dari anak pertama dari orang tua. Jadi jika kita memiliki hubungan antara orang tua dan anak pertama dan juga hubungan antara semua anak dari orang tua yang sama maka itu menyelesaikan masalah kita.

Representasi ini kadang-kadang disebut representasi anak pertama/saudara berikutnya. Representasi anak pertama/saudara berikutnya dari pohon generik ditunjukkan di atas. Representasi sebenarnya untuk pohon ini adalah:



Gambar 6.15 representasi anak pertama/saudara berikutnya pohon generik

Berdasarkan pembahasan ini, deklarasi simpul pohon untuk pohon umum dapat diberikan sebagai:

```
struct TreeNode {
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
};
```

Catatan: Karena kita dapat mengonversi pohon generik apa pun menjadi representasi biner; dalam praktiknya Kita menggunakan pohon biner. Kita dapat memperlakukan semua pohon generik dengan representasi anak pertama/saudara berikutnya sebagai pohon biner.

Pohon Generik: Masalah & Solusi

Soal-36 Diberikan sebuah pohon, berikan algoritma untuk mencari jumlah semua elemen pohon tersebut.

Solusi: Solusinya mirip dengan apa yang telah kita lakukan untuk pohon biner sederhana. Itu berarti, telusuri daftar lengkap dan terus tambahkan nilainya. Kita bisa menggunakan level order traversal atau rekursi sederhana.

```
int FindSum(struct TreeNode *root){
    if(!root)  return 0;
    return root→data + FindSum(root→firstChild) + FindSum(root→nextSibling);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1) (jika kita tidak mempertimbangkan ruang tumpukan), jika tidak, O(n).

Catatan: Semua masalah yang telah kita diskusikan untuk pohon biner juga berlaku untuk pohon generik. Alih-alih pointer kiri dan kanan, kita hanya perlu menggunakan firstChild dan nextSibling.

Soal-37 Untuk pohon 4-ary (setiap node dapat berisi maksimal 4 anak), berapa tinggi maksimum yang mungkin dengan 100 node? Asumsikan tinggi satu simpul adalah 0.

Solusi: Dalam pohon 4-ary setiap node dapat berisi 0 hingga 4 anak, dan untuk mendapatkan tinggi maksimum, kita hanya perlu menyimpan satu anak untuk setiap orang tua. Dengan 100 node, ketinggian maksimum yang bisa kita dapatkan adalah 99.

Jika kita memiliki batasan bahwa setidaknya satu node memiliki 4 anak, maka kita menyimpan satu node dengan 4 anak dan node yang tersisa dengan 1 anak. Dalam hal ini, ketinggian maksimum yang mungkin adalah 96. Demikian pula, dengan n node, ketinggian maksimum yang mungkin adalah n – 4.

Soal-38 Untuk pohon 4-ary (setiap simpul dapat berisi maksimal 4 anak), berapakah ketinggian minimum yang mungkin dengan n simpul?

Solusi: Mirip dengan pembahasan di atas, jika kita ingin mendapatkan tinggi minimum, maka kita perlu mengisi semua node dengan anak maksimum (dalam hal ini 4). Sekarang mari kita lihat tabel berikut, yang menunjukkan jumlah maksimum node untuk ketinggian tertentu.

Tinggi, h	Node Maksimum pada ketinggian, h= 4^h	Tinggi Total Node $h = \frac{4^{h+1}-1}{3}$
0	1	1
1	4	1 + 4
2	4 x 4	1 + 4 x 4
3	4 x 4 x 4	1 + 4 x 4 + 4 x 4 x 4

Untuk ketinggian tertentu h node maksimum yang mungkin adalah: $\frac{4^{h+1}-1}{3}$. Untuk mendapatkan tinggi minimum, ambil logaritma di kedua sisi:

$$n = \frac{4^{h+1}-1}{3} \Rightarrow 4^{h+1} = 3n+1 \Rightarrow (h+1)log4 = \log(3n+1) \Rightarrow h+1 = \log_4(3n+1) \Rightarrow h = \log_4(3n+1) - 1$$

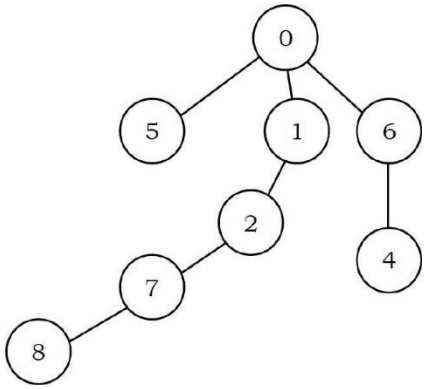
Soal-39 Diberikan sebuah larik induk P, di mana P[i] menunjukkan induk dari simpul ke-i di pohon (asumsikan induk simpul akar ditunjukkan dengan –1). Berikan algoritma untuk mencari tinggi atau kedalaman pohon.

Solusi:

Contoh: jika P adalah

-1	0	1	6	6	0	0	2	7
0	1	2	3	4	5	6	7	8

Pohon yang sesuai adalah:



Dari definisi masalah, array yang diberikan mewakili array induk. Itu berarti, kita perlu mempertimbangkan pohon untuk larik itu dan menemukan kedalaman pohon. Kedalaman pohon yang diberikan ini adalah 4. Jika kita mengamati dengan cermat, kita hanya perlu mulai dari setiap simpul dan terus ke induknya sampai kita mencapai -1 dan juga melacak kedalaman maksimum di antara semua simpul.

```
int FindDepthInGenericTree(int P[], int n){
    int maxDepth =-1, currentDepth =-1, j;
    for (int i = 0; i < n; i++)   {
        currentDepth = 0; j = i;
        while(P[j] != -1) {
            currentDepth++; j = P[j];
        }
        if(currentDepth > maxDepth)
            maxDepth = currentDepth;
    }
    return maxDepth;
}
```

Kompleksitas Waktu: O(n2). Untuk pohon miring Kita akan menghitung ulang nilai yang sama.
Kompleksitas Ruang: O(1).

Catatan: Kita dapat mengoptimalkan kode dengan menyimpan kedalaman node yang dihitung sebelumnya di beberapa tabel hash atau array lainnya. Ini mengurangi kompleksitas waktu tetapi menggunakan ruang ekstra.

Soal-40 Diberikan sebuah simpul pada pohon generik, berikan algoritma untuk menghitung jumlah saudara untuk simpul tersebut.

Solusi: Karena pohon diwakili dengan metode anak pertama/saudara berikutnya, struktur pohon dapat diberikan sebagai:

```
struct TreeNode{
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
};
```

Untuk simpul tertentu di pohon, kita hanya perlu melintasi semua saudara berikutnya.

```
int SiblingsCount(struct TreeNode *current){
    int count = 0;
    while(current) {
        count++;
        current = current→nextSibling;
    }
    reutrn count;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-41 Diberikan sebuah simpul dalam pohon generik, berikan algoritma untuk menghitung jumlah anak untuk simpul tersebut.

Solusi: Karena pohon direpresentasikan sebagai metode anak pertama/saudara berikutnya, struktur pohon dapat diberikan sebagai:

```
struct TreeNode{
    int data;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
};
```

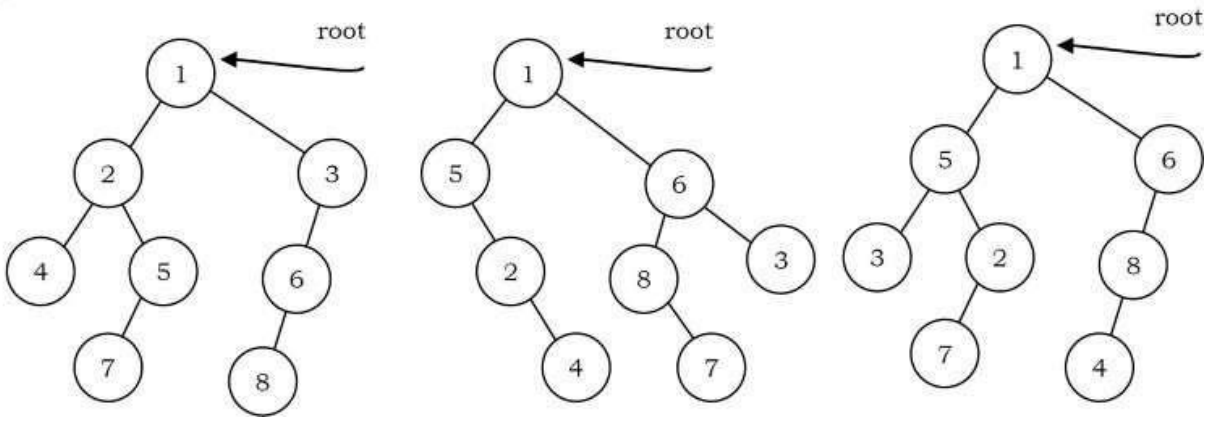
Untuk simpul tertentu di pohon, kita hanya perlu menunjuk ke anak pertamanya dan terus melintasi semua saudara berikutnya.

```
int ChildCount(struct TreeNode *current){
    int count = 0;
    current = current→firstChild;
    while(current)  {
            count++;
            current = current→nextSibling;
    }
    reutrn count;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-42 Diberikan dua pohon bagaimana kita memeriksa apakah pohon-pohon itu isomorfik satu sama lain atau tidak?

Solusi:



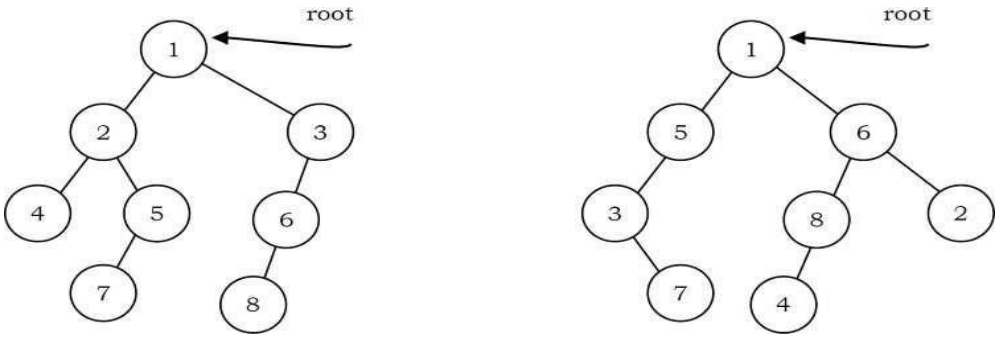
Dua pohon biner root1 dan root2 adalah isomorfik jika mereka memiliki struktur yang sama. Nilai node tidak mempengaruhi apakah dua pohon isomorfik atau tidak. Pada diagram di bawah, pohon di tengah tidak isomorfik dengan pohon lain, tetapi pohon di sebelah kanan isomorfik dengan pohon di sebelah kiri.

```
int IsIsomorphic(struct TreeNode  *root1, struct TreeNode *root2){
    if(!root1 && !root2)
        return 1;
    if((!root1 && root2) || (root1 && !root2))
        return 0;
    return (IsIsomorphic(root1→left, root2→left) && IsIsomorphic(root1→right, root2→right));
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-43 Diberikan dua pohon bagaimana kita memeriksa apakah mereka quasi-isomorphic satu sama lain atau tidak?

Solusi:



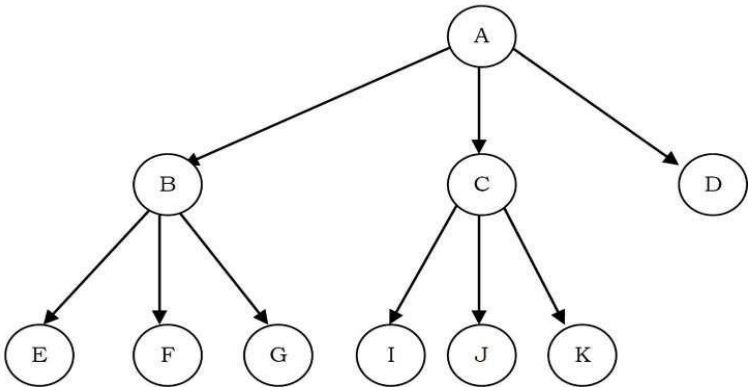
Dua pohon root1 dan root2 adalah quasi-isomorphic jika root1 dapat diubah menjadi root2 dengan menukar anak kiri dan kanan dari beberapa node root1. Data dalam node tidak penting dalam menentukan quasi-isomorphism; hanya bentuknya yang penting. Pohon-pohon di bawah ini adalah quasi-isomorfik karena jika anak-anak dari simpul di sebelah kiri ditukar, pohon di sebelah kanan diperoleh.

```
int QuasiIsomorphic(struct TreeNode  *root1, struct TreeNode  *root2){
    if(!root1 && !root2  return 1;
    if((!root1 && root2) || (root1 && !root2))
        return 0;
    return (QuasiIsomorphic(root1→left, root2→left) && QuasiIsomorphic(root1→right, root2→right)
          || QuasiIsomorphic(root1→right, root2→left) && QuasiIsomorphic(root1→left, root2→right));
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-44 Pohon k –ary penuh adalah pohon di mana setiap simpul memiliki 0 atau k anak. Diberikan sebuah array yang berisi traversal preorder pohon k –ary penuh, berikan algoritma untuk membangun pohon k –ary penuh.

Solusi: Pada k –ary tree, untuk sebuah node pada posisi ke-i anak-anaknya akan berada di k * i + 1 hingga k * i + k. Sebagai contoh, contoh di bawah ini adalah untuk pohon 3-ary penuh.



Seperti yang telah kita lihat, dalam traversal preorder subpohon kiri pertama diproses kemudian diikuti oleh root node dan subpohon kanan. Karena itu, untuk membangun k-ary penuh, kita hanya perlu terus membuat node tanpa mengganggu node yang dibangun sebelumnya. Kita dapat menggunakan trik ini untuk membangun pohon secara rekursif dengan menggunakan satu indeks global. Deklarasi untuk pohon k-ary dapat diberikan sebagai:

```
struct K-aryTreeNode{
    char data;
    struct K-aryTreeNode *child[];
};
int *Ind = 0;
struct K-aryTreeNode *BuildK-aryTree(char A[], int n, int k){
    if(n<=0) return NULL;
    struct K-aryTreeNode *newNode = (struct K-aryTreeNode*) malloc(sizeof(struct K-aryTreeNode));
    if(!newNode) {
            printf("Memory Error");
            return;
    }
    newNode→child = (struct K-aryTreeNode*) malloc( k * sizeof(struct K-aryTreeNode));
    if(!newNode→child) {
            printf("Memory Error");
            return;
    }
    newNode→data = A[Ind];
     for (int i = 0; i<k; i++) {
          if(k * Ind + i <n) {
              Ind++;
              newNode→child[i] = BuildK-aryTree(A, n, k,Ind );
          }
          else  newNode→child[i] =NULL;
     }
     return newNode;
}
```

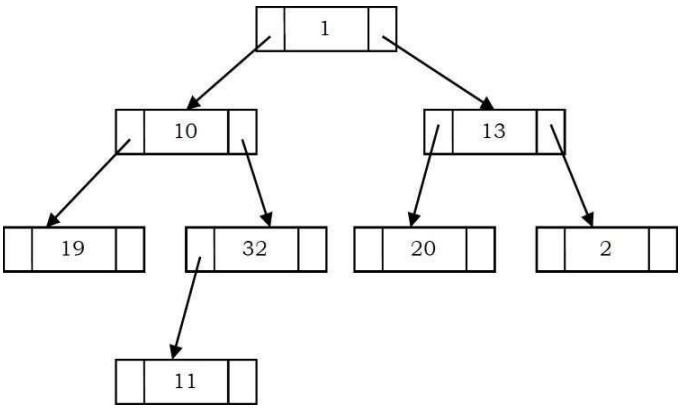
Kompleksitas Waktu: O(n), di mana n adalah ukuran larik pemesanan di muka. Ini karena kita bergerak secara berurutan dan tidak mengunjungi node yang sudah dibangun.

6.8 TRAVERSAL POHON BINER BERULIR (STACK ATAU TRAVERSAL TANPA ANTRIAN)

Pada bagian sebelumnya kita telah melihat bahwa, traversal pohon biner preorder, inorder dan postorder menggunakan tumpukan dan traversal level order menggunakan antrian sebagai struktur data tambahan. Pada bagian ini kita akan membahas algoritma traversal baru yang tidak membutuhkan stack dan queue. Algoritma traversal seperti ini disebut traversal pohon biner berulir atau *stack/queue – less traversal.*

Masalah dengan Traversal Pohon Biner Reguler

- Ruang penyimpanan yang dibutuhkan untuk tumpukan dan antrian besar.
- Mayoritas pointer dalam pohon biner manapun adalah NULL. Misalnya, pohon biner dengan n node memiliki n + 1 pointer NULL dan ini terbuang sia-sia.



Gambar 6.16 traversal pohon biner regular

- Sulit untuk menemukan node penerus (penerus preorder, inorder dan postorder) untuk node yang diberikan.

Motivasi untuk Pohon Biner Berulir

Untuk mengatasi masalah ini, satu ide adalah untuk menyimpan beberapa informasi yang berguna dalam pointer NULL. Jika kita mengamati traversal sebelumnya dengan cermat, stack/queue diperlukan karena kita harus mencatat posisi saat ini untuk pindah ke subpohon kanan setelah memproses subpohon kiri. Jika kita menyimpan informasi yang berguna dalam pointer NULL, maka kita tidak perlu menyimpan informasi tersebut di stack/queue.

Pohon biner yang menyimpan informasi tersebut dalam pointer NULL disebut pohon biner berulir. Dari pembahasan di atas, mari kita asumsikan bahwa kita ingin menyimpan beberapa informasi yang berguna di NULL petunjuk. Pertanyaan selanjutnya adalah apa yang harus disimpan?

Konvensi umum adalah untuk menempatkan informasi pendahulu/penerus. Artinya, jika kita berurusan dengan traversal preorder, maka untuk node yang diberikan, pointer kiri NULL akan berisi informasi pendahulunya preorder dan pointer kanan NULL akan berisi informasi penerus preorder. Pointer khusus ini disebut thread.

Mengklasifikasikan Pohon Biner Berulir

Klasifikasi didasarkan pada apakah kita menyimpan informasi yang berguna di kedua pointer NULL atau hanya di salah satunya.

- Jika kita menyimpan informasi pendahulunya dalam pointer kiri NULL saja, maka kita dapat memanggil pohon biner seperti itu pohon biner berulir kiri.
- Jika kita menyimpan informasi penerus dalam pointer kanan NULL saja, maka kita dapat memanggil pohon biner seperti pohon biner berulir kanan.
- Jika kita menyimpan informasi pendahulu di NULL pointer kiri dan informasi penerus di NULL pointer kanan, maka kita dapat memanggil pohon biner seperti itu pohon biner berulir penuh atau hanya pohon biner berulir.

Catatan: *Untuk pembahasan selanjutnya, Kita hanya mempertimbangkan (sepenuhnya) pohon biner berulir.*

Jenis Pohon Biner Berulir

Berdasarkan pembahasan di atas kita mendapatkan tiga representasi untuk pohon biner berulir.

- Preorder Threaded Binary Trees: Penunjuk kiri NULL akan berisi informasi pendahulunya PreOrder dan penunjuk kanan NULL akan berisi informasi penerus PreOrder.
- Pohon Biner Berulir Inorder: penunjuk kiri NULL akan berisi informasi pendahulu InOrder dan penunjuk kanan NULL akan berisi informasi penerus InOrder.
- Pohon Biner Berulir Postorder: penunjuk kiri NULL akan berisi informasi pendahulu PostOrder dan penunjuk kanan NULL akan berisi informasi penerus PostOrder.

Catatan: Karena representasinya serupa, untuk pembahasan selanjutnya kita akan menggunakan pohon biner berulir InOrder.

Struktur Pohon Biner Berulir

Setiap program yang memeriksa pohon harus dapat membedakan antara pointer kiri/kanan biasa dan thread. Untuk melakukan ini, Kita menggunakan dua bidang tambahan di setiap simpul, memberi Kita, untuk pohon berulir, simpul dengan bentuk berikut:

Gambar 6.17 struktur pohon biner berulir

Perbedaan antara Pohon Biner dan Struktur Pohon Biner Berulir

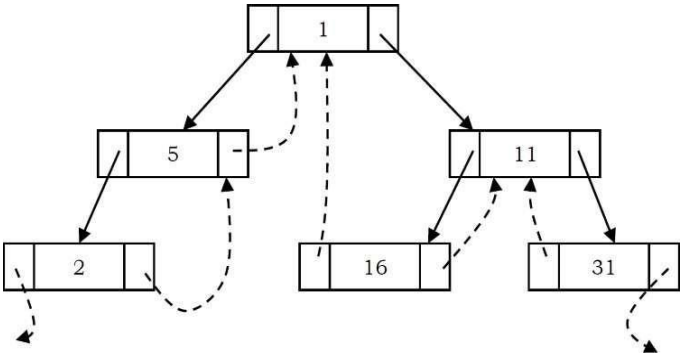
```
struct ThreadedBinaryTreeNode{
    struct ThreadedBinaryTreeNode *left;
    int LTag;
    int data;
    int RTag;
    struct ThreadedBinaryTreeNode *right;
};
```

Tabel 6.1 perbedaan pohon biner dan struktur pohon biner berulir

	Pohon Biner Reguler	Pohon Biner Berulir
jika LTag == 0	BATAL	titik kiri ke urutan pendahulunya
jika Ltag == 1	kiri menunjuk ke anak kiri	titik kiri ke anak kiri
jika RTtag == 0	BATAL	titik kanan ke penerus berurutan
jika RTtag == 1	titik yang tepat untuk anak yang tepat	titik kanan ke anak kanan

Catatan: *Demikian pula, kita dapat mendefinisikan perbedaan preorder/postorder juga*

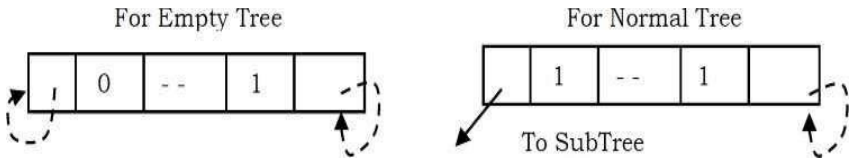
Sebagai contoh, mari kita coba merepresentasikan sebuah pohon dalam bentuk pohon biner berulir berurutan. Pohon di bawah ini menunjukkan seperti apa pohon biner berulir inorder. Panah putus-putus menunjukkan utas. Jika kita amati, penunjuk kiri dari simpul paling kiri (2) dan penunjuk kanan dari simpul paling kanan (31) tergantung.



Gambar 6.18 pohon biner berulir berurutan

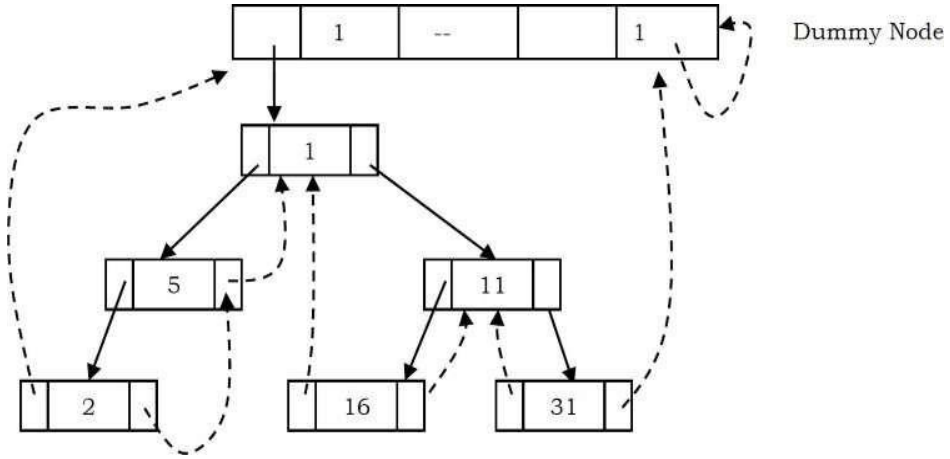
Apa yang harus ditunjukkan oleh pointer paling kiri dan paling kanan?

Dalam representasi pohon biner berulir, akan lebih mudah untuk menggunakan simpul khusus Dummy yang selalu ada bahkan untuk pohon kosong. Perhatikan bahwa tag kanan simpul Dummy adalah 1 dan anak kanannya menunjuk ke dirinya sendiri.



Gambar 6.19 pohon biner berulir berurutan

Dengan konvensi ini pohon di atas dapat direpresentasikan sebagai:



Gambar 6.20 Representasi pohon biner berulir

Menemukan Penerus Inorder di Pohon Biner Berulir Inorder

Untuk menemukan penerus terurut dari suatu simpul tertentu tanpa menggunakan tumpukan, asumsikan bahwa simpul yang ingin dicari penerusnya adalah P.

Strategi: Jika P tidak memiliki subpohon kanan, kembalikan anak kanan P. Jika P memiliki subpohon kanan, kembalikan kiri simpul terdekat yang subpohon kirinya berisi P.

```
struct ThreadedBinaryTreeNode* InorderSuccessor(struct ThreadedBinaryTreeNode *P){
    struct ThreadedBinaryTreeNode *Position;
    if(P→RTag == 0)
        return P→right;
    else {
        Position = P→right;
        while(Position→LTag == 1)
            Position = Position→left;
        return Position;
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Traversal Inorder di Pohon Biner Berulir Inorder

Kita bisa mulai dengan dummy node dan memanggil InorderSuccessor() untuk mengunjungi setiap node sampai kita mencapai simpul boneka.

```
void InorderTraversal(struct ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P = InorderSuccessor(root);
        while(P != root) {
            P = InorderSuccessor(P);
             printf("%d",P→data);
        }
}
```

Pengkodean alternatif:

```
void InorderTraversal(struct ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P = root;
        while(1) {
           P = InorderSuccessor(P);
            if(P == root) return;
            printf("%d",P→data);
        }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Menemukan Penerus PreOrder di Pohon Biner Berulir InOrder

Strategi: Jika P memiliki subpohon kiri, kembalikan anak kiri P. Jika P tidak memiliki subpohon kiri, kembalikan anak kanan dari simpul terdekat yang subpohon kanannya berisi P.

```
struct ThreadedBinaryTreeNode* PreorderSuccessor(struct ThreadedBinaryTreeNode *P){
        struct ThreadedBinaryTreeNode *Position;
        if(P→LTag == 1)
              return P→left;
        else {
              Position = P;
              while(Position→RTag == 0)
                    Position = Position→right;

              return Position→right;
        }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Traversal PreOrder dari Pohon Biner Berulir InOrder

Seperti pada inorder traversal, mulailah dengan dummy node dan panggil PreorderSuccessorf) untuk mengunjungi setiap node sampai kita mendapatkan dummy node lagi

```
void PreorderTraversal(struct ThreadedBinaryTreeNode *root){
    struct ThreadedBinaryTreeNode *P;
    P = PreorderSuccessor(root);
    while(P != root) {
        P = PreorderSuccessor(P);
        printf("%d",P→data);
    }
}
```

Pengkodean alternatif:

```
void PreorderTraversal(struct ThreadedBinaryTreeNode *root) {
    struct ThreadedBinaryTreeNode *P = root;
    while(1){
        P = PreorderSuccessor(P);
        if(P == root) return;
        printf("%d",P→data);
    }
}
```

Kompleksitas Waktu: O(n).

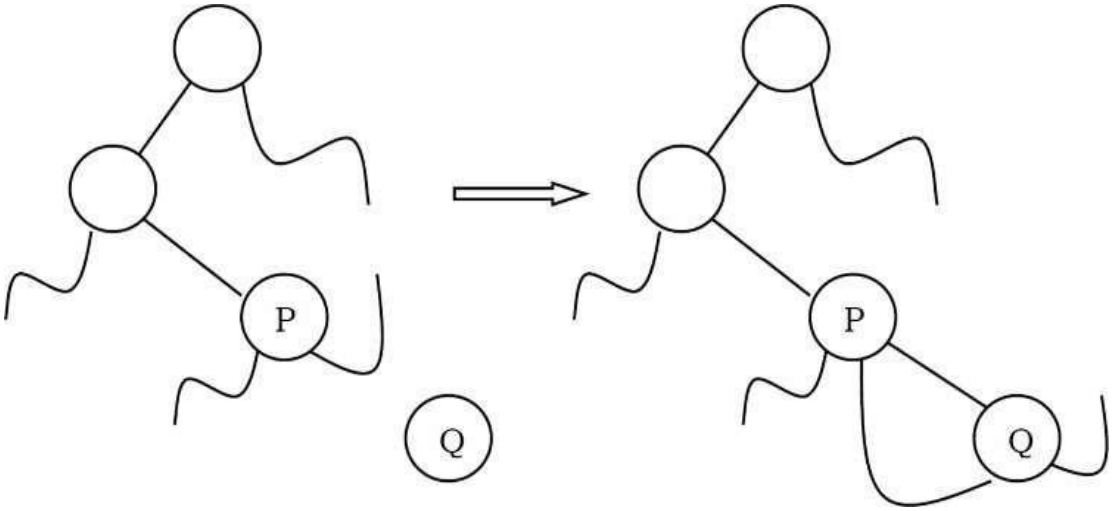
Kompleksitas Ruang: O(1).

Catatan: Dari pembahasan di atas, harus jelas bahwa penemuan suksesor inorder dan preorder mudah dilakukan dengan pohon biner berulir. Tetapi menemukan penerus postorder sangat sulit jika kita tidak menggunakan stack.

Penyisipan Node di InOrder Threaded Binary Trees

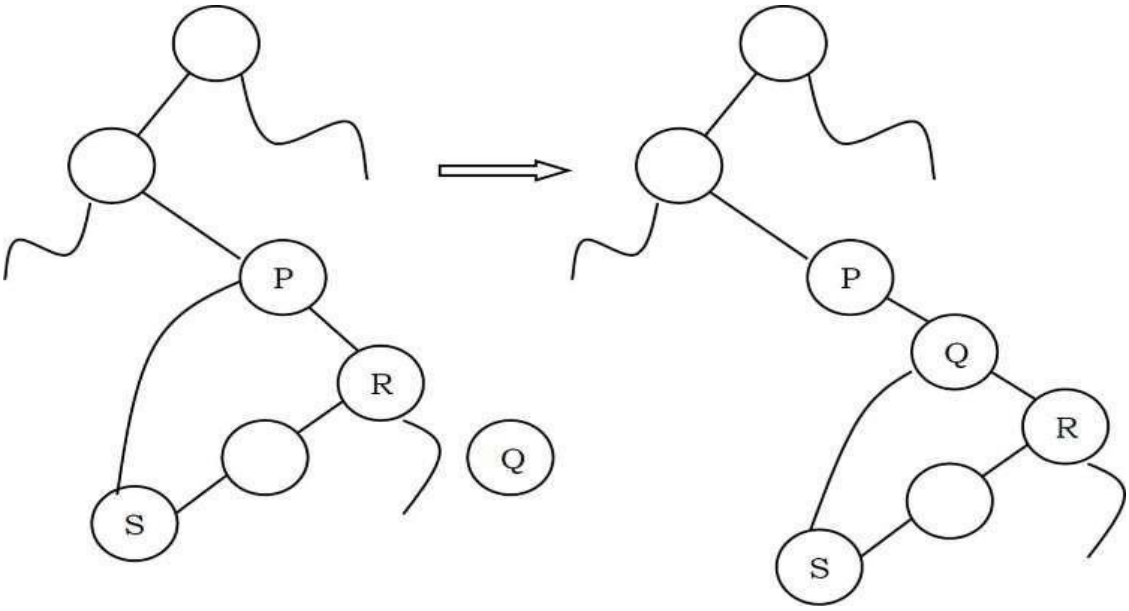
Untuk mempermudah, mari kita asumsikan bahwa ada dua simpul P dan Q dan kita ingin melampirkan Q di sebelah kanan P. Untuk ini kita akan memiliki dua kasus.

- Node P tidak memiliki anak kanan: Dalam hal ini kita hanya perlu melampirkan Q ke P dan mengubah pointer kiri dan kanannya.



Gambar 6.21 node P tidak memiliki anak kanan

- Node P memiliki anak kanan (katakanlah, R): Dalam hal ini kita perlu melintasi subpohon kiri R dan menemukan node paling kiri dan kemudian memperbarui pointer kiri dan kanan dari node tersebut (seperti yang ditunjukkan di bawah).



Gambar 6.22 node P memiliki anak kanan

```
void InsertRightInInorderTBT(struct ThreadedBinaryTreeNode *P,  struct ThreadedBinaryTreeNode *Q){
    struct ThreadedBinaryTreeNode *Temp;
    Q→right = P→right;
    Q→RTag = P→RTag;
    Q→left = P;
    Q→LTag = 0;
    P→right = Q;
    P→RTag = 1;
    if(Q→RTag == 1) {                          //Case-2
        Temp = Q→right;
        while(Temp→LTag)
            Temp = Temp→left;
        Temp→left = Q;
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Pohon Biner Berulir: Masalah & Solusi

Soal-45 Untuk pohon biner tertentu (tidak berulir) bagaimana kita menemukan penerus preorder?

Solusi: Untuk menyelesaikan masalah ini, kita perlu menggunakan stack tambahan S. Pada panggilan pertama, node parameter adalah penunjuk ke kepala pohon,

dan setelah itu nilainya adalah NULL. Karena Kita hanya meminta penerus dari simpul yang Kita dapatkan terakhir kali Kita memanggil fungsi.

Isi tumpukan S dan penunjuk P ke simpul terakhir yang "dikunjungi" perlu dipertahankan dari satu panggilan fungsi ke panggilan berikutnya; mereka didefinisikan sebagai variabel statis.

```
// pre-order successor for an unthreaded binary tree
struct BinaryTreeNode *PreorderSuccessor(struct BinaryTreeNode *node){
    static struct BinaryTreeNode *P;
    static Stack *S = CreateStack();
    if(node != NULL)
        P = node;
    if(P→left != NULL) {
        Push(S,P);
        P = P→left;
    }
    else {
        while (P→right == NULL)
            P = Pop(S);
        P = P→right;
    }
    return P;
}
```

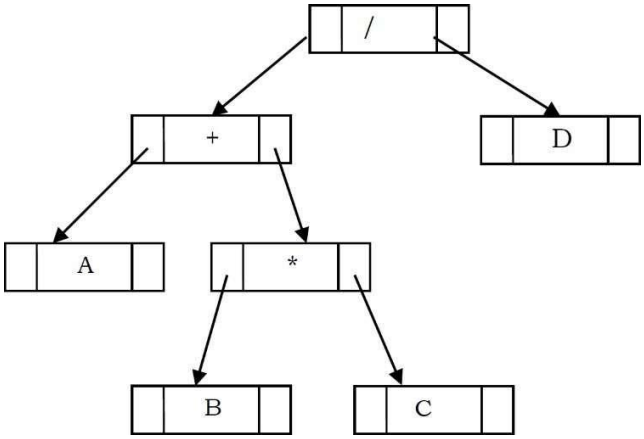
Soal-46 Untuk pohon biner tertentu (tidak berulir) bagaimana kita menemukan penerus dalam urutan?

Solusi: Mirip dengan diskusi di atas, kita dapat menemukan penerus berurutan dari sebuah simpul sebagai:

```
// In-order successor for an unthreaded binary tree
struct BinaryTreeNode *InorderSuccessor(struct BinaryTreeNode *node){
    static struct BinaryTreeNode *P;
    static Stack *S = CreateStack();
    if(node != NULL)
        P = node;
    if(P→right == NULL)
        P = Pop(S);
    else {
        P = P→right;
        while (P→left != NULL)
            Push(S, P);
        P = P→left;
    }
    return P;
}
```

6.9 POHON EKSPRESI

Pohon yang mewakili ekspresi disebut pohon ekspresi. Dalam pohon ekspresi, simpul daun adalah operand dan simpul non-daun adalah operator. Artinya, pohon ekspresi adalah pohon biner di mana simpul internal adalah operator dan daun adalah operan. Pohon ekspresi terdiri dari ekspresi biner. Tetapi untuk operator u-nary, satu subpohon akan kosong. Gambar di bawah menunjukkan pohon ekspresi sederhana untuk (A + B * C) / D.

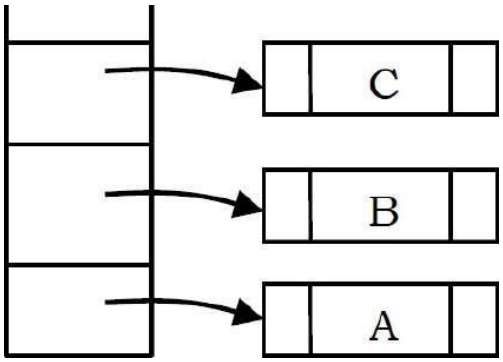


Gambar 6.23 Pohon ekspresi sederhana (A + B * C) / D

Algoritma untuk Membangun Pohon Ekspresi dari Ekspresi Postfix

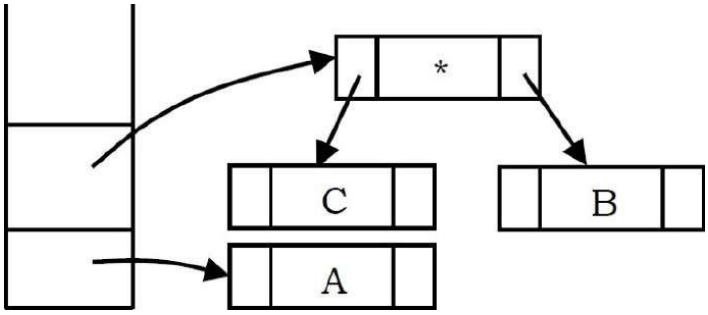
```
struct BinaryTreeNode *BuildExprTree(char postfixExpr[], int size){
    struct Stack *S = Stack(size);
    for (int i = 0; i< size; i++) {
        if(postfixExpr[i] is an operand) {
            struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
                    malloc( sizeof (struct BinaryTreeNode));
            if(!newNode) {
                printf("Memory Error");
                return NULL;
            }
            newNode→data =postfixExpr[i];
            newNode→left = newNode→right = NULL;
            Push(S, newNode);
        }
        else {
            struct BinaryTreeNode *T2 = Pop(S), *T1 = Pop(S);
            struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
                    malloc(sizeof(struct  BinaryTreeNode));
            if(!newNode) {
                printf("Memory Error");
                return NULL;
            }
            newNode→data = postfixExpr[i];
            newNode→left = T1;
            newNode→right = T2;
            Push(S, newNode);
        }
    }
    return S;
}
```

Contoh: Asumsikan bahwa satu 224simbol dibaca pada suatu waktu. Jika simbolnya adalah operan, Kita membuat simpul pohon dan mendorong pointer ke sana ke tumpukan. Jika 224simbol adalah operator, pop pointer ke dua pohon T1 dan T2 dari tumpukan (T1 muncul pertama) dan membentuk pohon baru yang akarnya adalah operator dan yang anak kiri dan kanannya masing-masing menunjuk ke T2 dan T1. Pointer ke pohon baru ini kemudian didorong ke tumpukan.
Sebagai contoh, asumsikan inputnya adalah A B C * + D /. Tiga 224simbol pertama adalah operan, jadi buat simpul pohon dan tekan pointer ke mereka ke tumpukan seperti yang ditunjukkan di bawah ini.



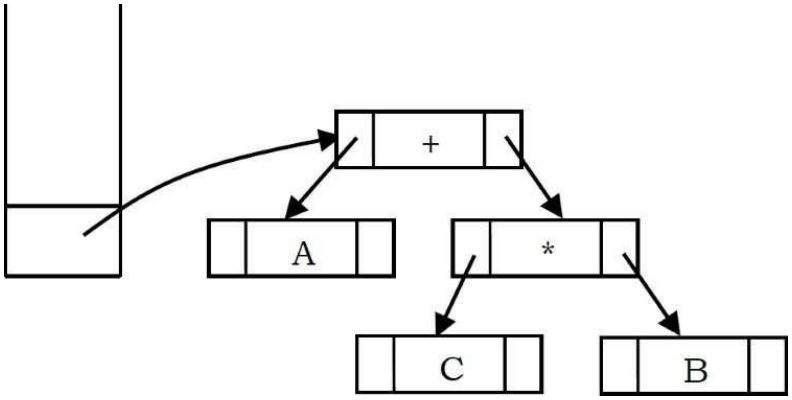
Gambar 6.24 simpul pohon

Selanjutnya, operator '*' dibaca, jadi dua pointer ke pohon muncul, pohon baru terbentuk dan pointer itu didorong ke tumpukan.



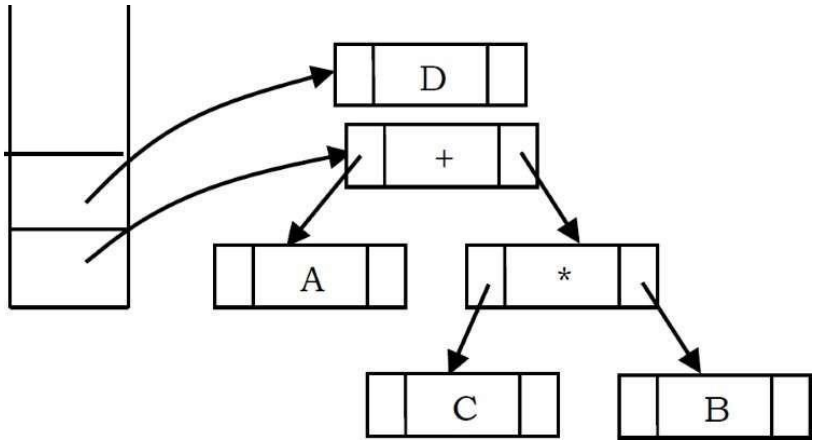
Gambar 6.25 operator '*'

Selanjutnya, operator '+' dibaca, jadi dua pointer ke pohon muncul, pohon baru terbentuk dan pointer ke itu didorong ke tumpukan.



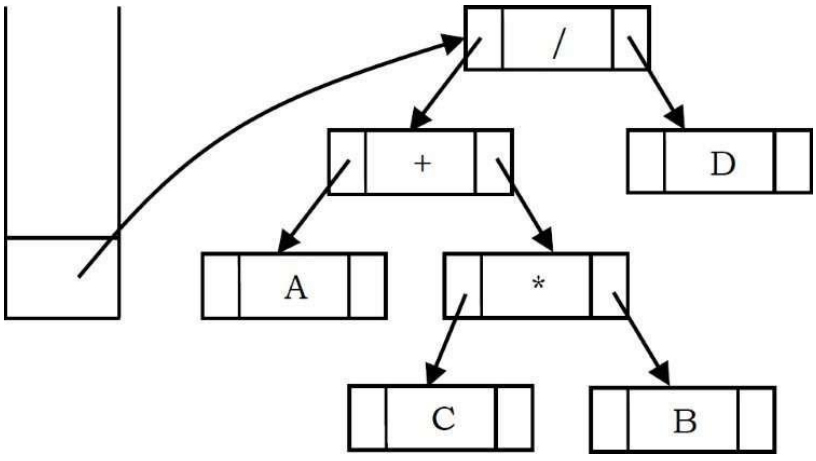
Gambar 6.26 operator '+'

Selanjutnya, operan 'D' dibaca, pohon satu-simpul dibuat dan penunjuk ke pohon yang sesuai didorong ke tumpukan.



Gambar 6.27 operan D

Terakhir, simbol akhir ('/') dibaca, dua pohon digabungkan dan penunjuk ke pohon terakhir tertinggal di tumpukan.

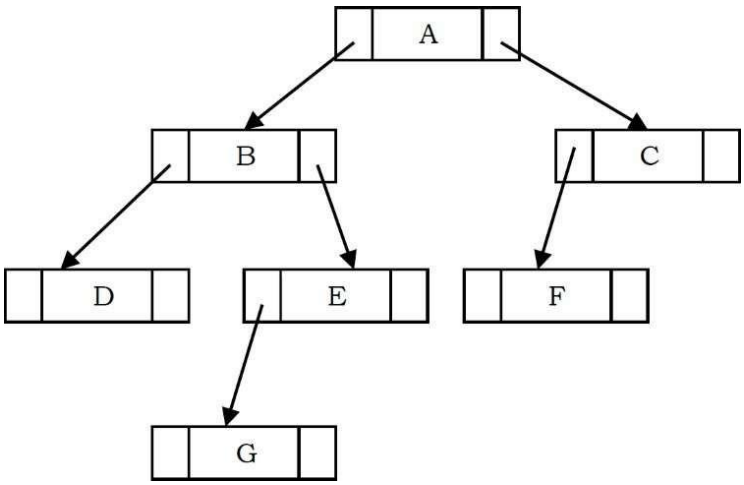


Gambar 6.28 simbol akhir '/'

6.10 POHON XOR

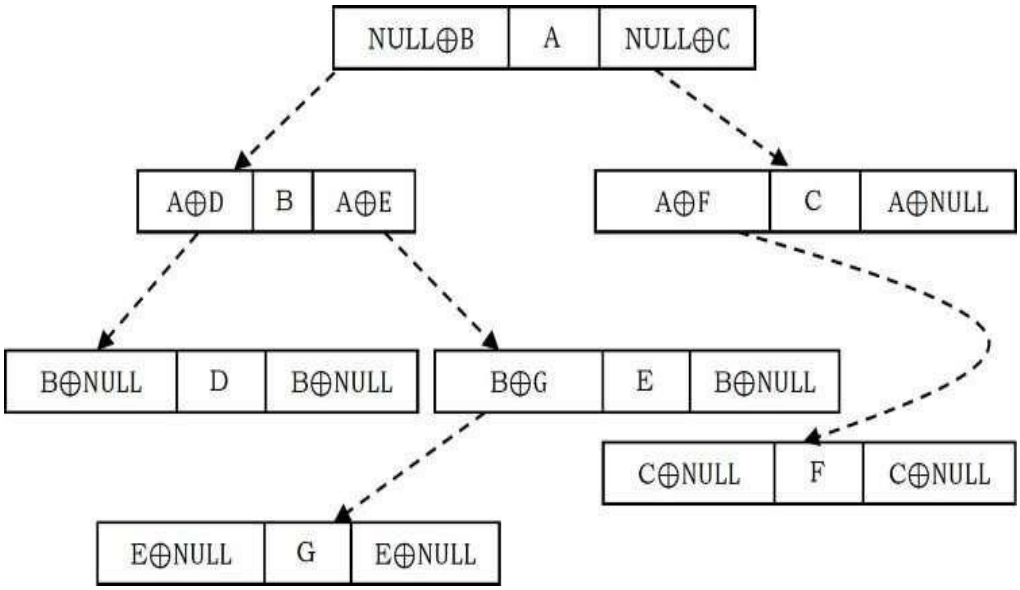
Konsep ini mirip dengan daftar tertaut ganda yang efisien memori dari bab Daftar Tertaut. Juga, seperti pohon biner berulir, representasi ini tidak memerlukan tumpukan atau antrian untuk melintasi pohon. Representasi ini digunakan untuk melintasi bolak-balik (ke orang tua) dan maju (ke anak-anak) menggunakan operasi . Untuk merepresentasikan hal yang sama di pohon XOR, untuk setiap node di bawah ini adalah aturan yang digunakan untuk representasi:

- Setiap node yang tersisa akan memiliki dari induknya dan anak kirinya.
- Setiap node di kanan akan memiliki dari parent dan anak kanannya.
- Orang tua simpul akar adalah NULL dan juga anak simpul daun adalah simpul NULL.



Gambar 6.29 Pohon XOR

Berdasarkan aturan dan diskusi di atas, pohon dapat direpresentasikan sebagai:



Gambar 6.30 Pohon XOR

Tujuan utama dari presentasi ini adalah kemampuan untuk pindah ke orang tua juga untuk anak-anak. Sekarang, mari kita lihat bagaimana menggunakan representasi ini untuk melintasi pohon. Misalnya, jika kita berada di simpul B dan ingin pindah ke simpul induknya A, maka kita hanya perlu melakukan pada konten kirinya dengan alamat anak kirinya (kita dapat menggunakan anak kanan juga untuk pergi ke simpul induk).

Demikian pula, jika kita ingin pindah ke anaknya (katakanlah, anak kiri D) maka kita harus melakukan pada konten kirinya dengan alamat simpul induknya. Satu hal penting yang perlu kita pahami tentang representasi ini adalah: Ketika kita berada di simpul B, bagaimana kita mengetahui alamat anak-anaknya D? Karena traversal dimulai pada node root node, kita dapat menerapkan pada konten kiri root dengan NULL. Sebagai hasilnya kita mendapatkan anak kirinya, B. Ketika kita berada di B, kita dapat menerapkan pada konten kirinya dengan alamat A.

6.11 POHON PENCARIAN BINER (BST)

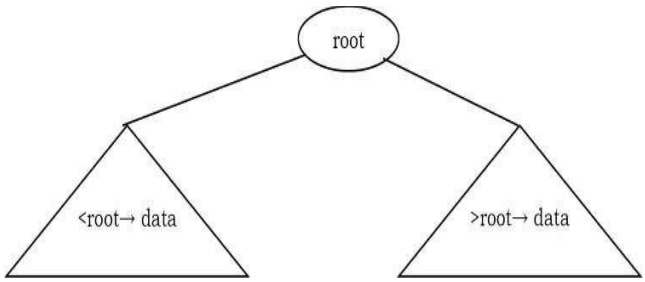
Di bagian sebelumnya kita telah membahas representasi pohon yang berbeda dan semuanya telah kita lakukan tidak memaksakan pembatasan pada data node. Akibatnya, untuk mencari elemen, kita perlu memeriksa baik di subpohon kiri maupun di subpohon kanan. Karena ini, kompleksitas kasus terburuk dari operasi pencarian adalah O(n).

Pada bagian ini, kita akan membahas varian lain dari pohon biner: *Binary Search Trees* (BSTs). Seperti namanya, penggunaan utama representasi ini adalah untuk pencarian. Dalam representasi ini Kita memberlakukan batasan pada jenis data yang dapat ditampung oleh sebuah node. Akibatnya, ini mengurangi operasi pencarian rata-rata kasus terburuk menjadi O(logn).

Properti Pohon Pencarian Biner

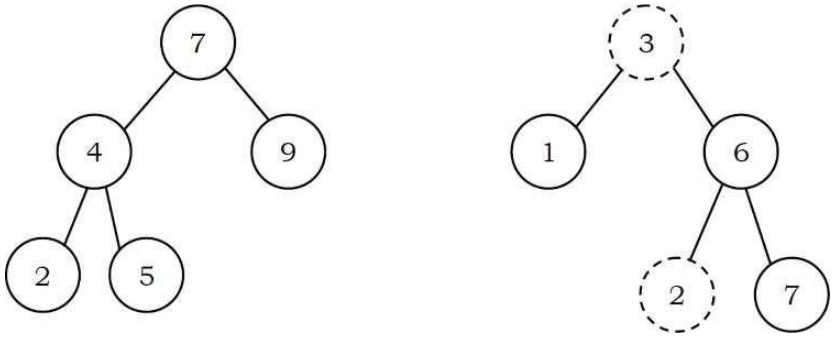
Dalam pohon pencarian biner, semua elemen subpohon kiri harus lebih kecil dari data akar dan semua elemen subpohon kanan harus lebih besar dari data akar. Ini disebut properti pohon pencarian biner. Perhatikan bahwa, properti ini harus dipenuhi di setiap simpul di pohon.

- Subpohon kiri dari sebuah simpul hanya berisi simpul dengan kunci yang lebih kecil dari kunci simpul.
- Subpohon kanan dari sebuah node hanya berisi node dengan kunci lebih besar dari kunci node.
- Baik subpohon kiri dan kanan juga harus berupa pohon pencarian biner.



Gambar 6.31 Pohon pencarian biner

Contoh: Pohon kiri adalah pohon pencarian biner dan pohon kanan bukan pohon pencarian biner (pada simpul 6 tidak memenuhi 228roperty pohon pencarian biner).



Gambar 6.32 Pohon pencarian dan bukan pohon pencarian biner

Deklarasi Pohon Pencarian Biner

Tidak ada perbedaan antara deklarasi pohon biner biasa dan deklarasi pohon pencarian biner. Perbedaannya hanya pada data tetapi tidak pada strukturnya. Tetapi untuk kenyamanan Kita, Kita mengubah nama struktur sebagai:

```
struct BinarySearchTreeNode{
    int data;
    struct BinarySearchTreeNode *left;
    struct BinarySearchTreeNode *right;
};
```

Operasi pada Pohon Pencarian Biner

Operasi utama: Berikut adalah operasi utama yang didukung oleh pohon pencarian biner:

- Temukan/Temukan Minimum/Temukan elemen Maksimum dalam pohon pencarian biner
- Memasukkan elemen dalam pohon pencarian biner
- Menghapus elemen dari pohon pencarian biner

Operasi bantu: Memeriksa apakah pohon yang diberikan adalah pohon pencarian biner atau tidak

- Menemukan elemen terkecil ke-k di pohon
- Menyortir elemen pohon pencarian biner dan banyak lagi

Catatan Penting tentang Pohon Pencarian Biner

- Karena data root selalu berada di antara data subpohon kiri dan data subpohon kanan, melakukan traversal inorder pada pohon pencarian biner menghasilkan daftar yang diurutkan.
- Saat menyelesaikan masalah pada pohon pencarian biner, pertama-tama kita proses subpohon kiri, kemudian data root, dan terakhir kita proses subpohon kanan. Ini berarti, tergantung pada masalahnya, hanya langkah perantara (memproses data root) yang berubah dan Kita tidak menyentuh langkah pertama dan ketiga.
- Jika kita sedang mencari sebuah elemen dan jika data akar subpohon kiri lebih kecil dari elemen yang ingin kita cari, lewati saja. Sama halnya dengan subpohon kanan.. Karena itu, pohon pencarian biner membutuhkan waktu lebih sedikit untuk mencari elemen daripada pohon biner biasa. Dengan kata lain, pohon pencarian biner mempertimbangkan subpohon kiri atau kanan untuk mencari elemen tetapi tidak keduanya.
- Operasi dasar yang dapat dilakukan pada binary search tree (BST) adalah penyisipan elemen, penghapusan elemen, dan pencarian elemen. Saat melakukan operasi ini di BST, ketinggian pohon berubah setiap kali. Oleh karena itu terdapat variasi dalam kompleksitas waktu kasus terbaik, kasus rata-rata, dan kasus terburuk.
- Operasi dasar pada pohon pencarian biner membutuhkan waktu yang sebanding dengan tinggi pohon. Untuk pohon biner lengkap dengan simpul n, operasi tersebut berjalan dalam O(lgn) waktu kasus terburuk. Jika pohon adalah rantai linier dari n simpul (pohon miring), bagaimanapun, operasi yang sama membutuhkan O(n) waktu kasus terburuk.

Menemukan Elemen di Pohon Pencarian Biner

Cari operasi sangat mudah di BST. Mulailah dengan root dan terus bergerak ke kiri atau kanan menggunakan properti BST. Jika data yang Kita cari sama dengan data node maka Kita mengembalikan node saat ini.

Jika data yang kita cari lebih kecil dari data node maka cari subpohon kiri dari node saat ini; jika tidak, cari subpohon kanan dari simpul saat ini. Jika data tidak ada, Kita berakhir di tautan NULL.

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    if( data < root→data )
        return Find(root→left, data);
    else if( data > root→data )
        return( Find( root→right, data );
    return root;
}
```

Kompleksitas Waktu: O(n), dalam kasus terburuk (ketika BST adalah pohon miring).
Kompleksitas Ruang: O(n), untuk tumpukan rekursif.

Versi non rekursif dari algoritma di atas dapat diberikan sebagai:

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    while (root) {
        if(data == root→data)
            return root;
        else if(data > root→data)
            root = root→right;
        else root = root→left;
    }
    return NULL;
}
```

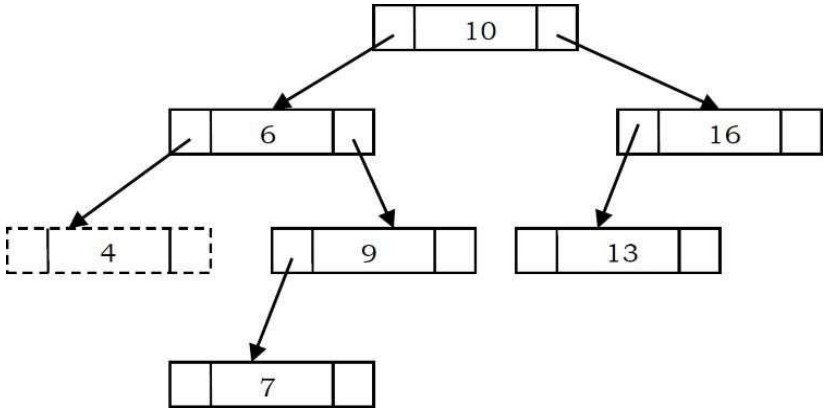
Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Menemukan Elemen Minimum di Pohon Pencarian Biner

Dalam BST, elemen minimum adalah node paling kiri, yang tidak memiliki anak kiri. Dalam BST di bawah ini, elemen minimum adalah 4.

```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode *root){
    if(root == NULL)
        return NULL;
    else  if( root→left == NULL )
        return root;
    else
        return FindMin( root→left );
}
```

Kompleksitas Waktu: O(n), dalam kasus terburuk (ketika BST adalah pohon miring kiri).
Kompleksitas Ruang: O(n), untuk tumpukan rekursif.



Gambar 6.33 elemen minimum di pohon pencarian biner

Versi non rekursif dari algoritma di atas dapat diberikan sebagai:

```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode * root ) {
    if( root == NULL )
        return NULL;
    while( root→left != NULL )
        root = root→left;
    return root;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

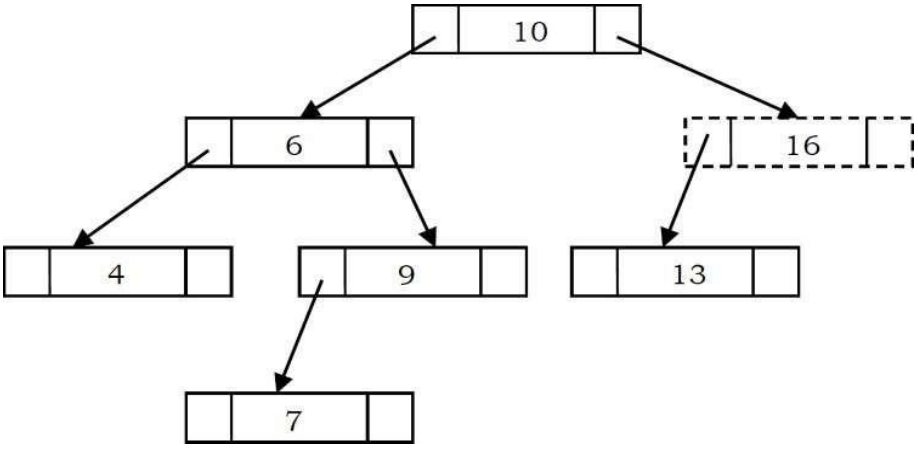
Menemukan Elemen Maksimum di Pohon Pencarian Biner

Dalam BST, elemen maksimum adalah simpul paling kanan, yang tidak memiliki anak kanan. Dalam BST di bawah ini, elemen maksimum adalah 16.

```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode *root) {
    if(root == NULL)
        return NULL;
    else if( root→right == NULL )
        return root;
    else return FindMax( root→right );
}
```

Kompleksitas Waktu: O(n), dalam kasus terburuk (ketika BST adalah pohon miring kanan).

Kompleksitas Ruang: O(n), untuk tumpukan rekursif.



Gambar 6.34 elemen maksimum pohon pencarian biner

Versi non rekursif dari algoritma di atas dapat diberikan sebagai:

```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode * root ) {
    if( root == NULL )
        return NULL;
    while( root→right != NULL )
        root = root→right;
    return root;
}
```

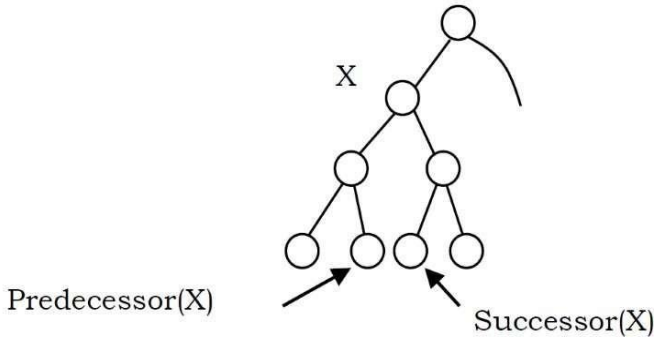
Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Dimana Inorder Predecessor dan Successor?

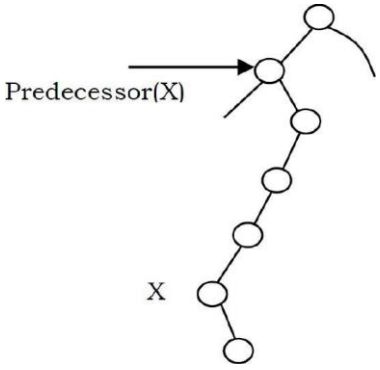
Di mana pendahulu inorder dan penerus simpul X dalam pohon pencarian biner dengan asumsi semua kunci berbeda?

Jika X memiliki dua anak, maka pendahulunya dalam urutannya adalah nilai maksimum pada subpohon kirinya dan penggantinya dalam urutannya adalah nilai minimum pada subpohon kanannya.



Gambar 6.35 predecessor(x) dan Successor(x)

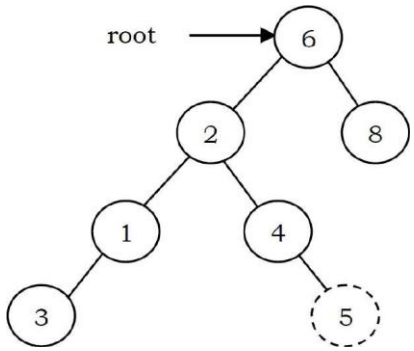
Jika tidak memiliki anak kiri, maka pendahulu dalam urutan node adalah nenek moyang kiri pertama.



Gambar 6.36 predecessor(x)

Memasukkan Elemen dari Pohon Pencarian Biner

Untuk memasukkan data ke dalam pohon pencarian biner, pertama-tama kita perlu menemukan lokasi untuk elemen tersebut. Kita dapat menemukan lokasi penyisipan dengan mengikuti mekanisme yang sama dengan operasi find. Saat mencari lokasi, jika datanya sudah ada maka kita bisa mengabaikan dan keluar begitu saja. Jika tidak, masukkan data di lokasi terakhir pada jalur yang dilalui.



Gambar 6.37 input elemen dari pohon pencarian biner

Sebagai contoh mari kita perhatikan pohon berikut. Node putus-putus menunjukkan elemen (5) yang akan disisipkan. Untuk menyisipkan 5, telusuri pohon menggunakan fungsi find. Pada simpul dengan kunci 4, kita harus ke kanan, tetapi tidak ada subpohon, jadi 5 tidak ada di pohon, dan ini adalah lokasi yang benar untuk penyisipan.

```
struct BinarySearchTreeNode *Insert(struct BinarySearchTreeNode *root, int data) {
    if( root == NULL ) {
        root = (struct BinarySearchTreeNode *) malloc(sizeof(struct  BinarySearchTreeNode));
        if( root == NULL ) {
            printf("Memory Error");
            return;
        }
        else {
            root→data = data;
            root→left = root→right = NULL;
         }
    }
    else {
        if( data < root→data )
            root→left = Insert(root→left,  data);
        else if( data > root→data )
            root→right = Insert(root→right,  data);
    }
    return root;
}
```

Catatan: Dalam kode di atas, setelah menyisipkan elemen di subpohon, pohon dikembalikan ke induknya. Akibatnya, pohon lengkap akan diperbarui.

Kompleksitas Waktu: O(n).

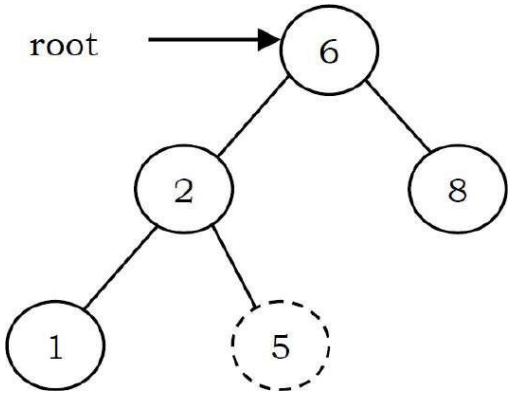
Kompleksitas Ruang: O(n), untuk tumpukan rekursif. Untuk versi iteratif, kompleksitas ruang adalah O(1).

Menghapus Elemen dari Pohon Pencarian Biner

Operasi hapus lebih rumit daripada operasi lainnya. Ini karena elemen yang akan dihapus mungkin bukan simpul daun. Dalam operasi ini juga, pertama-tama kita perlu menemukan lokasi elemen yang ingin kita hapus.

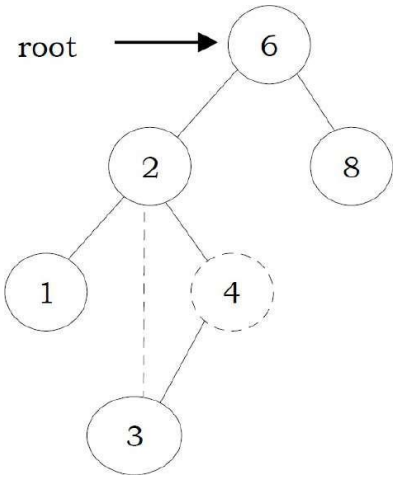
Setelah Kita menemukan simpul yang akan dihapus, pertimbangkan kasus berikut:

- Jika elemen yang akan dihapus adalah simpul daun: kembalikan NULL ke induknya. Itu berarti membuat penunjuk anak yang sesuai NULL. Di pohon di bawah ini untuk menghapus 5, setel NULL ke simpul induknya 2.



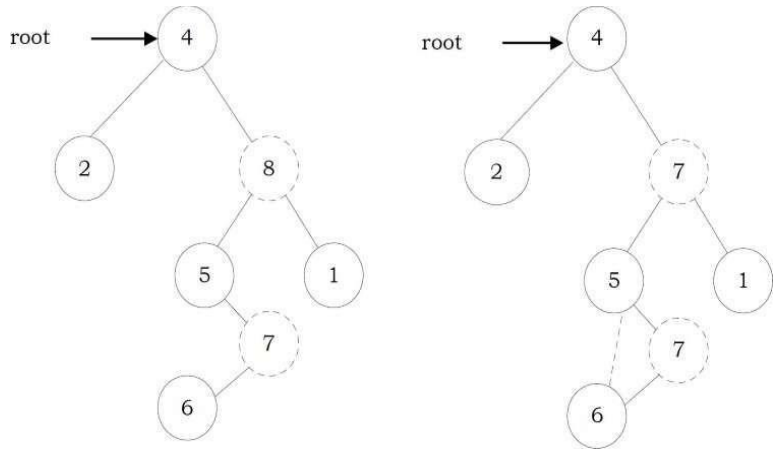
Gambar 6.38 hapus elemen dari pohon pencarian biner

- Jika elemen yang akan dihapus memiliki satu anak: Dalam hal ini kita hanya perlu mengirim anak simpul saat ini ke induknya. Pada pohon di bawah ini, untuk menghapus 4, 4 subpohon kiri diatur ke simpul induknya 2.



Gambar 6.39 elemen dihapus memiliki satu anak

- Jika elemen yang akan dihapus memiliki kedua anak: Strategi umumnya adalah mengganti kunci dari simpul ini dengan elemen terbesar dari subpohon kiri dan menghapus simpul tersebut secara rekursif (yang sekarang kosong). Node terbesar di subpohon kiri tidak dapat memiliki anak kanan, jadi penghapusan kedua adalah yang mudah. Sebagai contoh, mari kita perhatikan pohon berikut. Di pohon di bawah ini, untuk menghapus 8, itu adalah anak kanan dari root. Nilai kuncinya adalah 8. Itu diganti dengan kunci terbesar di subpohon kirinya (7), dan kemudian simpul itu dihapus seperti sebelumnya (kasus kedua).



Gambar 6.40 elemen dihapus memiliki dua anak

Catatan: *Kita juga bisa mengganti dengan elemen minimum di subpohon kanan.*

```
struct BinarySearchTreeNode *Delete(struct BinarySearchTreeNode *root, int data) {
    struct BinarySearchTreeNode *temp;
    if( root == NULL )
        printf("Element not there in tree");
    else if(data < root→data)
            root→left = Delete(root→left, data);
    else if(data > root→data )
        root→right = Delete(root→right, data);
    else {
        //Found element
        if( root→left && root→right ) {
            /* Replace with largest in left subtree */
            temp = FindMax( root→left );
            root→data = temp→data;
            root→left = Delete(root→left, root→data);
        }
        else {
                /* One child */
                temp = root;
                if( root→left == NULL )
                    root = root→right;
                if( root→right == NULL )
                    root = root→left;
                free( temp );
        }
    }
    return root;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n) untuk tumpukan rekursif. Untuk versi iteratif, kompleksitas ruang adalah O(1).

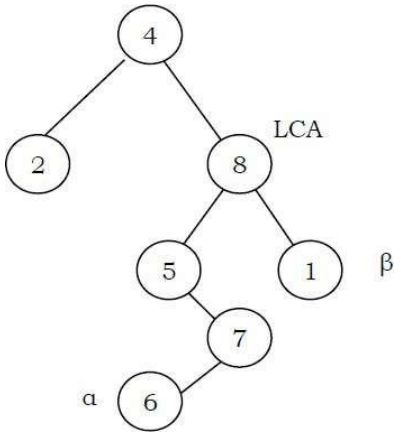
Pohon Pencarian Biner: Masalah & Solusi

Catatan: Untuk mengurutkan masalah terkait dengan pohon pencarian biner dan pohon pencarian biner seimbang,

Traversal inorder memiliki kelebihan dibandingkan yang lain karena memberikan urutan yang diurutkan.

Soal-47 Diberikan pointer ke dua node dalam pohon pencarian biner, temukan leluhur bersama (LCA) terendah. Asumsikan bahwa kedua nilai sudah ada di pohon.

Solusi:



Ide utama dari solusinya adalah: saat melintasi BST dari root ke bawah, node pertama yang kita temui dengan nilai antara dan , yaitu, < node → data < , adalah Least Common Ancestor(LCA) dari dan (dimana <). Jadi cukup lewati BST secara pre-order, dan jika kita menemukan node dengan nilai di antara dan , maka node tersebut adalah LCA. Jika nilainya lebih besar dari dan , maka LCA terletak di sisi kiri node, dan jika nilainya lebih kecil dari dan , maka LCA terletak di sisi kanan.

```
struct BinarySearchTreeNode *FindLCA(struct BinarySearchTreeNode *root, struct BinarySearchTreeNode *α,
                                                    struct BinarySearchTreeNode * β) {
    while(1) {
        if((α→data < root→data && β→data > root→data) ||
                                    (α→data > root→data && β→data < root→data))
            return root;
        if(α→data < root→data)
          root = root→left;
        else root = root→right;
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk pohon miring.

Soal-48 Berikan algoritma untuk menemukan jalur terpendek antara dua node dalam BST.

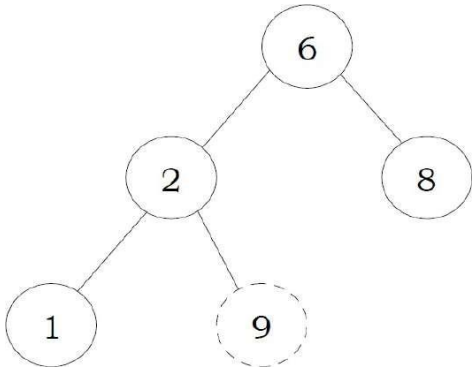
Solusi: Tidak lain adalah menemukan LCA dari dua node di BST.

Soal-49 Berikan algoritma untuk menghitung jumlah BST yang mungkin dengan n node.

Solusi: Ini adalah masalah DP. Lihat bab tentang Pemrograman Dinamis untuk algoritma.

Soal-50 Berikan algoritma untuk memeriksa apakah pohon biner yang diberikan adalah BST atau tidak.

Solusi:



Perhatikan program sederhana berikut. Untuk setiap node, periksa apakah node di sebelah kirinya lebih kecil dan periksa apakah node di sebelah kanannya lebih besar. Pendekatan ini salah karena ini akan mengembalikan true untuk pohon biner di bawah ini. Memeriksa hanya pada node saat ini tidak cukup.

```
int IsBST(struct BinaryTreeNode* root) {
    if(root == NULL)
        return 1;
    // false if left is > than root
    if(root→left != NULL && root→left→data > root→data)
        return 0;
    // false if right is < than root
    if(root→right != NULL && root→right→data < root→data)
        return 0;
    // false if, recursively, the left or right is not a BST
    if(!IsBST(root→left) || !IsBST(root→right))
        return 0;
    // passing all that, it's a BST
    return 1;
}
```

Soal-51 Bisakah kita memikirkan untuk mendapatkan algoritma yang benar?

Solusi: Untuk setiap node, periksa apakah nilai max di subpohon kiri lebih kecil dari data node saat ini dan nilai min di subpohon kanan lebih besar dari data node.

Diasumsikan bahwa kita memiliki fungsi pembantu FindMin() dan FindMax() yang mengembalikan nilai integer min atau maks dari pohon yang tidak kosong.

```
/* Returns true if a binary tree is a binary search tree */
int IsBST(struct BinaryTreeNode* root) {
    if(root == NULL)
        return 1;
    /* false if the max of the left is > than root */
    if(root→left != NULL && FindMax(root→left) > root→data)
        return 0;
    /* false if the min of the right is <= than root */
    if(root→right != NULL && FindMin(root→right) < root→data)
        return 0;
    /* false if, recursively, the left or right is not a BST */
    if(!IsBST(root→left) || !IsBST(root→right))
        return 0;
    /* passing all that, it's a BST */
    return 1;
}
```

Kompleksitas Waktu: O(n2).
Kompleksitas Ruang: O(n).

Soal-52 Bisakah kita meningkatkan kompleksitas Soal-51?

Solusi: Ya. Solusi yang lebih baik adalah dengan melihat setiap node hanya sekali. Triknya adalah dengan menulis fungsi pembantu utilitas IsBSTUtil(struct BinaryTreeNode* root, int min, int max) yang melintasi pohon dengan melacak

```
Initial call: IsBST(root, INT_MIN, INT_MAX);
int IsBST(struct BinaryTreeNode *root, int min, int max) {
    if(!root)
      return 1;
    return (root→data >min && root→data < max &&
            IsBSTUtil(root→left, min, root→data) &&
            IsBSTUtil(root→right, root→data, max));
}
```

nilai min dan max yang dipersempit saat berjalan, melihat setiap node hanya sekali. Nilai awal untuk min dan max harus INT_MIN dan INT_MAX – mereka menyempit dari sana.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n), untuk ruang tumpukan.

Soal-53 Bisakah kita lebih meningkatkan kompleksitas Soal-51?

Solusi: Ya, dengan menggunakan inorder traversal. Ide dibalik solusi ini adalah bahwa inorder traversal dari BST menghasilkan daftar yang diurutkan. Saat melintasi

BST secara berurutan, pada setiap node periksa kondisi bahwa nilai kuncinya harus lebih besar dari nilai kunci dari node yang dikunjungi sebelumnya. Juga, kita perlu menginisialisasi prev dengan kemungkinan nilai integer minimum (katakanlah, INT_MIN).

```
int prev = INT_MIN;
int IsBST(struct BinaryTreeNode *root, int *prev) {
 if(!root) return 1;
 if(!IsBST(root→left, prev))
      return 0;
 if(root→data < *prev)
      return 0;
 *prev = root→data;
 return IsBST(root→right, prev);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk ruang tumpukan.

Soal-54 Berikan algoritma untuk mengubah BST menjadi DLL melingkar dengan kompleksitas ruang O(1).

Solusi: Ubah subpohon kiri dan kanan menjadi DLL dan pertahankan akhir daftar tersebut. Kemudian, sesuaikan pointer.

```
struct BinarySearchTreeNode *BST2DLL(struct BinarySearchTreeNode *root,
                                     struct BinarySearchTreeNode **Ltail) {
    struct BinarySearchTreeNode *left, *ltail, *right, *rtail;
    if(!root) {
        * ltail = NULL;
        return NULL;
    }
    left = BST2DLL(root→left, &ltail);
    right = BST2DLL(root→right, &rtail);
    root→left = ltail;
    root→right = right;
    if(!right)
        * ltail = root;
    else {
        right→left = root;
        * ltail = rtail;
    }
    if(!left)
        return root;
    else {
        ltail→right = root;
        return left;
    }
}
```

Kompleksitas Waktu: O(n).

Soal-55 Untuk Soal-54, apakah ada cara lain untuk menyelesaikannya?

Solusi: Ya. Ada solusi alternatif berdasarkan metode bagi dan taklukkan yang cukup rapi.

```
struct BinarySearchTreeNode *Append(struct BinarySearchTreeNode *a, struct BinarySearchTreeNode *b) {
   struct BinarySearchTreeNode  *aLast, *bLast;
   if (a==NULL)
      return b;
   if (b==NULL)
      return a;
   aLast = a→left;
   bLast = b→left;
   aLast→right = b;
   b→left = aLast;
   bLast→right = a;
   a→left = bLast;
   return a;
}
struct BinarySearchTreeNode* TreeToList(struct BinarySearchTreeNode *root) {
   struct BinarySearchTreeNode *aList, *bList;
   if (root==NULL)
     return NULL;
   aList = TreeToList(root→left);
   bList = TreeToList(root→right);
   root→left = root;
   root→right = root;
   aList = Append(aList, root);
   aList = Append(aList, bList);
   return(aList);
}
```

Kompleksitas Waktu: O(n).

Soal-56 Diberikan daftar tertaut ganda yang diurutkan, berikan algoritma untuk mengubahnya menjadi pohon pencarian biner seimbang.

Solusi: Temukan simpul tengah dan sesuaikan penunjuk.

```
struct DLLNode * DLLtoBalancedBST(struct DLLNode *head) {
   struct DLLNode *temp, *p, *q;
   if( !head || !head→next)
      return head;
   temp = FindMiddleNode(head);
   p = head;
   while(p→next != temp)
      p = p→next;
   p→next = NULL;
   q = temp→next;
   temp→next = NULL;
   temp→prev = DLLtoBalancedBST(head);
   temp→next = DLLtoBalancedBST(q);
   return temp;
}
```

Kompleksitas Waktu: 2T(n/2) + O(n) [untuk menemukan simpul tengah] = O(nlogn).
Catatan: Untuk fungsi FindMiddleNode, lihat bab Daftar Tertaut.

Soal-57 Diberikan array yang diurutkan, berikan algoritma untuk mengonversi array ke BST.

Solusi: Jika kita harus memilih elemen array untuk menjadi root dari BST yang seimbang, elemen mana yang harus kita pilih? Akar dari BST yang seimbang harus menjadi elemen tengah dari array yang diurutkan. Kita akan memilih elemen tengah dari array yang diurutkan di setiap iterasi. Kita kemudian membuat simpul di pohon yang diinisialisasi dengan elemen ini. Setelah elemen dipilih, apa yang tersisa? Bisakah Anda mengidentifikasi sub-masalah dalam masalah?

Ada dua array kiri - satu di kiri dan satu di kanan. Kedua array ini adalah sub-masalah dari masalah asli, karena keduanya diurutkan. Selanjutnya, mereka adalah subpohon dari anak kiri dan kanan simpul saat ini.

Kode di bawah ini membuat BST seimbang dari array yang diurutkan dalam waktu O(n) (n adalah jumlah elemen dalam array). Bandingkan seberapa mirip kodenya dengan algoritma pencarian biner. Keduanya menggunakan metodologi bagi dan taklukkan.

```
struct BinaryTreeNode  *BuildBST(int A[], int left, int right) {
    struct BinaryTreeNode  *newNode;
    int mid;
    if(left > right)
        return NULL;
    newNode = (struct BinaryTreeNode  *)malloc(sizeof(struct BinaryTreeNode));
    if(!newNode) {
       printf("Memory Error");
       return;
    }
    if(left == right) {
       newNode→data = A[left];
       newNode→left = newNode→right = NULL;
    }
    else {    mid = left + (right-left)/ 2;
       newNode→data = A[mid];
       newNode→left = BuildBST(A, left, mid - 1);
       newNode→right = BuildBST(A, mid + 1, right);
    }
    return newNode;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n), untuk ruang tumpukan.

Soal-58 Diberikan daftar tertaut tunggal di mana elemen diurutkan dalam urutan menaik, ubahlah menjadi BST seimbang tinggi.

Solusi: Cara yang naif adalah dengan menerapkan solusi Masalah-56 secara langsung. Dalam setiap panggilan rekursif, kita harus melintasi setengah dari panjang daftar untuk menemukan elemen tengah. Kompleksitas run time jelas O(nlogn), di mana n adalah jumlah total elemen dalam daftar. Ini karena setiap level panggilan rekursif membutuhkan total n/2 langkah traversal dalam daftar, dan ada total jumlah logn level (yaitu, ketinggian pohon seimbang).

Soal-59 Untuk Soal-58, dapatkah kita meningkatkan kompleksitasnya?

Solusi: **Petunjuk:** Bagaimana dengan menyisipkan node mengikuti urutan daftar? Jika kita dapat mencapai ini, kita tidak perlu lagi menemukan elemen tengah karena kita dapat menelusuri daftar sambil menyisipkan node ke pohon.

Solusi Terbaik: Seperti biasa, solusi terbaik mengharuskan kita untuk berpikir dari perspektif lain. Dengan kata lain, kita tidak lagi membuat node di pohon menggunakan pendekatan top-down. Buat node dari bawah ke atas, dan tetapkan ke orang tuanya. Pendekatan bottom-up memungkinkan kita untuk mengakses daftar dalam urutannya sambil membuat node.

Bukankah pendekatan *bottom-up* tepat? Setiap kali kita terjebak dengan pendekatan top-down, kita dapat mencoba bottom-up. Meskipun pendekatan bottom-up bukanlah cara yang paling alami menurut Kita, namun dalam beberapa kasus, pendekatan ini sangat membantu. Namun, kita sebaiknya memilih top-down daripada bottom-up secara umum, karena yang terakhir lebih sulit untuk diverifikasi.

Di bawah ini adalah kode untuk mengonversi daftar tertaut tunggal ke BST seimbang. Harap dicatat bahwa algoritma memerlukan panjang daftar untuk diteruskan sebagai parameter fungsi. Panjang daftar dapat ditemukan dalam waktu O(n) dengan melintasi seluruh daftar satu kali. Panggilan rekursif melintasi daftar dan membuat simpul pohon dengan urutan daftar, yang juga membutuhkan waktu O(n). Oleh karena itu, kompleksitas run time keseluruhan masih O(n).

```
struct BinaryTreeNode* SortedListToBST(struct ListNode *& list, int start, int end) {
    if(start > end)
        return NULL;
    // same as (start+end)/2, avoids overflow
    int mid = start + (end - start) / 2;
    struct BinaryTreeNode *leftChild = SortedListToBST(list, start, mid-1);
    struct BinaryTreeNode  * parent;
    parent = (struct BinaryTreeNode  *)malloc(sizeof(struct BinaryTreeNode));
    if(!parent) {
       printf("Memory Error");
       return;
    }
    parent→data=list→data;
    parent→left = leftChild;
    list = list→next;
    parent→right = SortedListToBST(list, mid+1, end);
    return parent;
}
struct BinaryTreeNode * SortedListToBST(struct ListNode *head, int n) {
    return SortedListToBST(head, 0, n-1);
}
```

Soal-60 Berikan algoritma untuk menemukan elemen terkecil ke-k di BST.

Solusi: Ide di balik solusi ini adalah, inorder traversal dari BST menghasilkan daftar yang diurutkan. Saat melintasi BST secara berurutan, lacak jumlah elemen yang dikunjungi.

```
struct BinarySearchTreeNode *kthSmallestInBST(struct BinarySearchTreeNode *root, int k, int *count){
    if(!root)
        return NULL;
    struct BinarySearchTreeNode *left = kthSmallestInBST(root→left, k, count);
    if( left )
        return left;
    if(++count == k)
        return root;
    return kthSmallestInBST(root→right, k, count);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-61 Lantai dan langit-langit: Jika kunci yang diberikan kurang dari kunci di akar BST maka lantai kunci (kunci terbesar di BST kurang dari atau sama dengan kunci) harus di subpohon kiri. Jika kunci lebih besar dari kunci pada akar, maka lantai dari kunci dapat berada di subpohon kanan, tetapi hanya jika ada kunci yang lebih kecil dari atau sama dengan kunci di subpohon kanan; jika tidak (atau jika kuncinya sama dengan kunci di root) maka kunci di root adalah lantai dari kunci tersebut. Menemukan langit-langit serupa, dengan bertukar kanan dan kiri.

Misalnya, jika array yang diurutkan dengan input adalah {1, 2, 8, 10, 10, 12, 19}, maka Untuk x = 0: lantai tidak ada dalam array, langit-langit = 1, Untuk x = 1: lantai = 1, langit-langit = 1 Untuk x = 5: lantai =2, langit-langit = 8, Untuk x = 20: lantai = 19 , ceil tidak ada dalam array

Solusi: Ide di balik solusi ini adalah, inorder traversal dari BST menghasilkan daftar yang diurutkan. Saat melintasi BST secara berurutan, lacak nilai yang sedang dikunjungi. Jika data akar lebih besar dari nilai yang diberikan maka kembalikan nilai sebelumnya yang telah kita pertahankan selama traversal. Jika data root sama dengan data yang diberikan, maka kembalikan data root.

```
struct BinaryTreeNode *FloorInBST(struct BinaryTreeNode *root, int data){
    struct BinaryTreeNode *prev=NULL;
    return FloorInBSTUtil(root, prev, data);
}
struct BinaryTreeNode *FloorInBSTUtil(struct BinaryTreeNode *root,
                                      struct BinaryTreeNode *prev, int data){
    if(!root)
        return NULL;
    if(!FloorInBSTUtil(root→left, prev, data))
        return 0;
    if(root→data == data)
        return root;
    if(root→data > data)
        return prev;
    prev = root;
    return FloorInBSTUtil(root→right, prev, data);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk ruang tumpukan.

Soal-62 Berikan algoritma untuk menemukan gabungan dan perpotongan BST. Asumsikan pointer induk tersedia (katakanlah pohon biner berulir). Juga, asumsikan panjang dua BST berturut-turut adalah m dan n.

Solusi: Jika pointer induk tersedia maka masalahnya sama dengan menggabungkan dua daftar yang diurutkan. Ini karena jika kita memanggil penerus berurutan setiap kali kita mendapatkan elemen tertinggi berikutnya. Ini hanya masalah InorderSuccessor mana yang akan dipanggil.

Untuk langit-langit, kita hanya perlu memanggil subpohon kanan terlebih dahulu, diikuti oleh subpohon kiri.

```
struct BinaryTreeNode *CeilingInBST(struct BinaryTreeNode *root, int data){
    struct BinaryTreeNode *prev=NULL;
    return CeilingInBSTUtil(root, prev, data);
}

struct BinaryTreeNode *CeilingInBSTUtil(struct BinaryTreeNode *root,
                                        struct BinaryTreeNode *prev, int data){
    if(!root)
        return NULL;
    if(!CeilingInBSTUtil(root→right, prev, data))
        return 0;
    if(root→data == data)
        return root;
    if(root→data < data)
        return prev;
    prev = root;
    return CeilingInBSTUtil(root→left, prev, data);
}
```

Kompleksitas Waktu: O(m + n).
Kompleksitas ruang: O(1).

Soal-63 Untuk Soal-62, bagaimana jika pointer orang tua tidak tersedia?

Solusi: Jika penunjuk induk tidak tersedia, BST dapat dikonversi ke daftar tertaut dan kemudian digabungkan.

1. Ubah kedua BST menjadi daftar terurut ganda dalam waktu O(n + m). Ini menghasilkan 2 daftar yang diurutkan.
2. Gabungkan dua daftar tertaut ganda menjadi satu dan juga pertahankan jumlah elemen total dalam waktu O(n + m).
3. Ubah daftar tertaut ganda yang diurutkan menjadi pohon seimbang tinggi dalam waktu O(n + m).

Soal-64 Untuk Soal-62, apakah ada cara alternatif untuk menyelesaikan masalah?

Solusi: Ya, dengan menggunakan inorder traversal.

- Lakukan inorder traversal pada salah satu BST.
- Saat melakukan traversal simpan dalam tabel (tabel hash).
- Setelah traversal BST pertama selesai, mulailah traversal BST kedua dan bandingkan dengan isi tabel hash.

Kompleksitas Waktu: O(m + n).
Kompleksitas Ruang: O(Maks(m,n)).

Soal-65 Diberikan BST dan dua angka K1 dan K2, berikan algoritma untuk mencetak semua elemen BST dalam rentang K1 dan K2.

Solusi:

```
void RangePrinter(struct BinarySearchTreeNode *root, int  K1, int K2) {
    if(root == NULL)
        return;
    if(root→data >= K1)
        RangePrinter(root→left, K1, K2);
    if(root→data >= K1 && root→data <= K2)
            printf("%d", root→data);
    if(root→data <= K2)
        RangePrinter(root→right, K1, K2);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk ruang tumpukan.

Soal-66 Untuk Soal-65, apakah ada cara alternatif untuk menyelesaikan masalah?

Solusi: Kita bisa menggunakan level order traversal: sambil menambahkan elemen ke antrian, periksa rentangnya.

```
void  RangeSeachLevelOrder(struct BinarySearchTreeNode *root, int  K1, int K2){
    struct BinarySearchTreeNode *temp;
    struct Queue *Q = CreateQueue();
    if(!root)
        return NULL;
    Q = EnQueue(Q, root);
    while(!IsEmptyQueue(Q)) {
            temp=DeQueue(Q);
            if(temp→data >= K1 && temp→data <= K2)
                    printf("%d",temp→data);
            if(temp→left && temp→data >= K1)
                    EnQueue(Q, temp→left);
            if(temp→right && temp→data <= K2)
                    EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
    return NULL;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk antrian.

Soal-67 Untuk Soal-65, masih bisakah kita memikirkan cara alternatif untuk menyelesaikan masalah?

Solusi: Pertama cari K1 dengan pencarian biner normal dan setelah itu gunakan penerus InOrder sampai kita menemukan K2. Untuk algoritma, lihat bagian masalah dari pohon biner berulir.

Soal-68 Diberikan akar pohon Pencarian Biner, rapikan pohonnya, sehingga semua elemen yang dikembalikan di pohon baru berada di antara input A dan B.

Solusi: Ini hanyalah cara lain untuk menanyakan Masalah-65.

Soal-69 Diberikan dua BST, periksa apakah elemen-elemennya sama atau tidak. Misalnya: dua BST dengan data 10 5 20 15 30 dan 10 20 15 30 5 harus mengembalikan true dan dataset dengan 10 5 20 15 30 dan 10 15 30 20 5 harus mengembalikan false. Catatan: Data BST dapat dalam urutan apa pun.

Solusi: Salah satu cara sederhana adalah melakukan traversal inorder pada pohon pertama dan menyimpan datanya dalam tabel hash. Sebagai langkah kedua, lakukan inorder traversal pada pohon kedua dan periksa apakah data tersebut sudah ada di tabel hash atau belum (jika ada di tabel hash maka tandai dengan -1 atau beberapa nilai unik).

Selama traversal pohon kedua jika kita menemukan ketidakcocokan kembali salah. Setelah melintasi pohon kedua, periksa apakah ia memiliki semua -1 di tabel hash atau tidak (ini memastikan data tambahan tersedia di pohon kedua).

Kompleksitas Waktu: O(max(m, n)), di mana m dan n adalah jumlah elemen dalam BST pertama dan kedua. Kompleksitas Ruang: O(maks(m,n)). Ini tergantung pada ukuran pohon pertama.

Soal-70 Untuk Soal-69, dapatkah kita mengurangi kompleksitas waktu?

Solusi: Alih-alih melakukan traversal satu demi satu, kita dapat melakukan traversal berurutan dari kedua pohon secara paralel. Karena in – order traversal memberikan daftar yang diurutkan, kita dapat memeriksa apakah kedua pohon menghasilkan urutan yang sama atau tidak.

Kompleksitas Waktu: O(maks(m,n)).

Kompleksitas Ruang: O(1). Ini tergantung pada ukuran pohon pertama.

Soal-71 Untuk nilai kunci 1... n, berapa banyak BST yang unik secara struktural mungkin yang menyimpan kunci tersebut.

Solusi: Strategi: pertimbangkan bahwa setiap nilai dapat menjadi akar. Temukan ukuran subpohon kiri dan kanan secara rekursif.

```
int CountTrees(int n) {
    if (n <= 1)
        return 1;
    else {
            // there will be one value at the root, with whatever remains on the left and right
            // each forming their own subtrees. Iterate through all the values that could be the root...
            int sum = 0;
            int left, right, root;
            for (root=1; root<=n; root++) {
                    left = CountTrees(root - 1);
                    right = CountTrees(numKeys - root);

                    // number of possible trees with this root == left*right
                    sum += left*right;
            }
            return(sum);
    }
}
```

Soal-72 Mengingat BST ukuran n, di mana setiap simpul r memiliki bidang tambahan r → ukuran, jumlah kunci dalam sub-pohon berakar pada r (termasuk simpul akar r). Berikan algoritma O(h) GreaterthanConstant(r,k) untuk menemukan jumlah kunci yang benar-benar lebih besar dari k (h adalah ketinggian pohon pencarian biner).

Solusi:

```
int GreaterthanConstant (struct BinarySearchTreeNode *r, int k){
    keysCount = 0
    while (r != Null ){
        if (k < r→data){
            keysCount = keysCount + r→right→size + 1;
            r = r→left;
        }
        else if (k > r→data)
            r = r→right;
        else{ // k = r→key
            keysCount = keysCount + r→right→size;
            break;
        }
    }
    return keysCount;
}
```

Algoritma yang disarankan berfungsi dengan baik jika kuncinya adalah nilai unik untuk setiap node. Sebaliknya ketika mencapai k=r→data, kita harus memulai proses perpindahan ke kanan hingga mencapai simpul y dengan kunci yang lebih besar dari k, dan kemudian kita harus mengembalikan keysCount +

y→size. Kompleksitas Waktu: O(h) di mana h=O(n) dalam kasus terburuk dan O(logn) dalam kasus rata-rata.

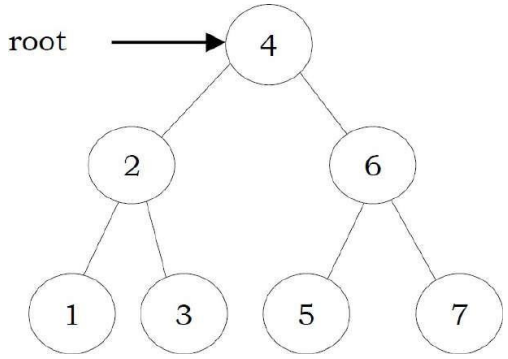
6.12 POHON PENCARIAN BINER SEIMBANG

Pada bagian sebelumnya kita telah melihat pohon berbeda yang kompleksitas kasus terburuknya adalah O(n), di mana n adalah jumlah simpul di pohon. Ini terjadi ketika pohon adalah pohon miring. Pada bagian ini kita akan mencoba untuk mengurangi kompleksitas kasus terburuk ini menjadi O(logn) dengan menerapkan batasan pada ketinggian.

Secara umum, tinggi pohon yang seimbang direpresentasikan dengan HB(k), di mana k adalah selisih antara tinggi subpohon kiri dan tinggi subpohon kanan. Kadang-kadang k disebut faktor keseimbangan.

Pohon Pencarian Biner Seimbang Penuh

Dalam HB(k), jika k = 0 (jika faktor keseimbangan adalah nol), maka kita menyebut pohon pencarian biner tersebut sebagai pohon pencarian biner seimbang penuh. Itu berarti, dalam pohon pencarian biner HB(0), perbedaan antara tinggi subpohon kiri dan tinggi subpohon kanan harus paling nol. Ini memastikan bahwa pohon tersebut adalah pohon biner penuh. Sebagai contoh,



Gambar 6.41 pohon pencarian biner seimbang penuh

Catatan: Untuk membuat pohon HB(0) lihat bagian Masalah.

6.13 POHON AVL(ADELSON-VELSKII DAN LANDIS)

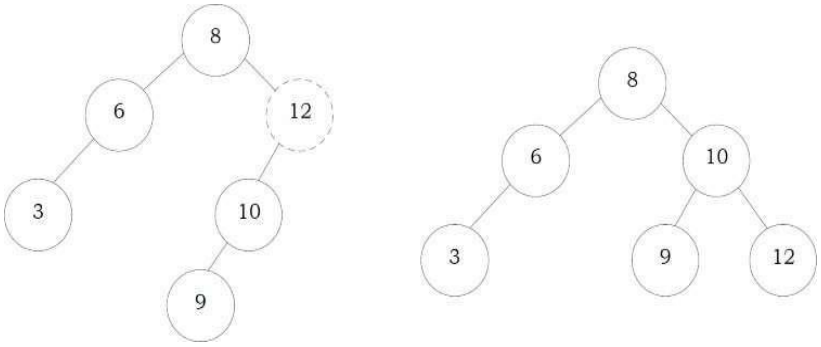
Dalam HB(k), jika k = 1 (jika faktor keseimbangan adalah satu), pohon pencarian biner seperti itu disebut pohon AVL. Artinya pohon AVL adalah pohon pencarian biner dengan kondisi keseimbangan: selisih antara tinggi subpohon kiri dan tinggi subpohon kanan paling banyak 1.

Properti Pohon AVL

Pohon biner dikatakan sebagai pohon AVL, jika:

- Ini adalah pohon pencarian biner, dan

- Untuk sembarang simpul X, tinggi subpohon kiri X dan tinggi subpohon kanan X berbeda paling banyak 1.



Gambar 6.42 Pohon AVL

Sebagai contoh, di antara pohon pencarian biner di atas, yang kiri bukan pohon AVL, sedangkan pohon pencarian biner kanan adalah pohon AVL.

Jumlah Node Minimum/Maksimum di AVL Tree

Untuk mempermudah, mari kita asumsikan bahwa tinggi pohon AVL adalah h dan N(K) menunjukkan jumlah node dalam pohon AVL dengan tinggi h. Untuk mendapatkan jumlah node minimum dengan tinggi h, kita harus mengisi pohon dengan jumlah node minimum yang mungkin. Artinya jika kita mengisi subpohon kiri dengan tinggi h – 1 maka kita harus mengisi subpohon kanan dengan tinggi h – 2. Akibatnya, jumlah minimum node dengan tinggi h adalah:

$$N(h) = N(h-1) + N(h-2) + 1$$

Dalam persamaan di atas:

- N(h – 1) menunjukkan jumlah minimum node dengan tinggi h – 1.
- N(h – 2) menunjukkan jumlah minimum node dengan tinggi h – 2.
- Dalam ekspresi di atas, "1" menunjukkan node saat ini.

Kita dapat memberikan N(h – 1) baik untuk subpohon kiri atau subpohon kanan. Memecahkan pengulangan di atas memberikan:

$$N(h) = O(1.618^h) \Rightarrow h = 1.44 logn \approx O(logn)$$

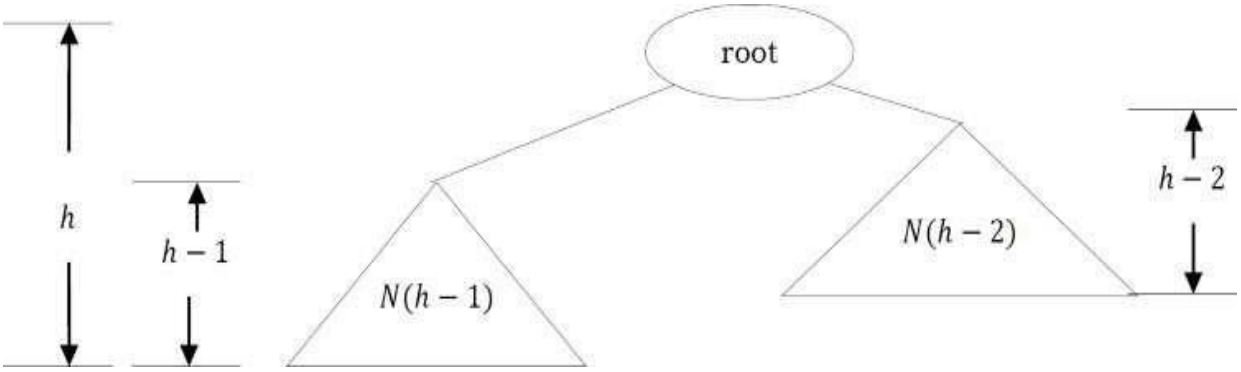
Dimana n adalah jumlah node dalam pohon AVL. Juga, turunan di atas mengatakan bahwa ketinggian maksimum di pohon AVL adalah O(logn). Demikian pula, untuk mendapatkan jumlah maksimum node, kita perlu mengisi subpohon kiri dan kanan dengan tinggi h – 1. Hasilnya, kita mendapatkan:

$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

Ekspresi di atas mendefinisikan kasus pohon biner penuh. Memecahkan pengulangan yang kita dapatkan:

$$N(h) = O(2^h) \Rightarrow h = logn \approx O(logn)$$

∴Dalam kedua kasus, properti pohon AVL memastikan bahwa ketinggian pohon AVL dengan n node adalah O(logn).



Gambar 6.43 node minimum dan maksimum AVL *tree*

Deklarasi Pohon AVL

Karena pohon AVL adalah BST, deklarasi AVL mirip dengan BST. Tetapi hanya untuk menyederhanakan operasi, Kita juga menyertakan tinggi sebagai bagian dari deklarasi.

```
struct AVLTreeNode{
    struct AVLTreeNode *left;
    int data;
    struct AVLTreeNode *right;
    int height;
};
```

Menemukan Ketinggian pohon AVL

```
int Height(struct AVLTreeNode *root ){
    if( !root)
        return -1;
    else
        return root→height;
}
```

Kompleksitas Waktu: O(1).

Rotasi

Ketika struktur pohon berubah (misalnya, dengan penyisipan atau penghapusan), kita perlu memodifikasi pohon untuk memulihkan properti pohon AVL. Ini dapat dilakukan dengan menggunakan rotasi tunggal atau rotasi ganda. Karena penyisipan/penghapusan melibatkan

penambahan/penghapusan satu simpul, ini hanya dapat menambah/mengurangi ketinggian subpohon sebesar 1.

Jadi, jika properti pohon AVL dilanggar pada simpul X, itu berarti ketinggian kiri(X) dan kanan(X) berbeda tepat 2. Hal ini karena, jika kita menyeimbangkan pohon AVL setiap saat, maka pada setiap titik, perbedaan ketinggian kiri(X) dan kanan(X) berbeda persis 2. Rotasi adalah teknik yang digunakan untuk memulihkan properti pohon AVL. Ini berarti, kita perlu menerapkan rotasi untuk simpul X.

Pengamatan: Satu pengamatan penting adalah bahwa, setelah penyisipan, hanya node yang berada di jalur dari titik penyisipan ke root yang mungkin saldonya diubah, karena hanya node tersebut yang mengubah subpohonnya. Untuk memulihkan properti pohon AVL, kita mulai dari titik penyisipan dan terus ke akar pohon.

Saat pindah ke root, kita perlu mempertimbangkan node pertama yang tidak memenuhi properti AVL. Dari node itu dan seterusnya, setiap node di jalur ke root akan memiliki masalah.

Juga, jika Kita memperbaiki masalah untuk simpul pertama itu, maka semua simpul lain di jalur ke akar akan secara otomatis memenuhi properti pohon AVL. Itu berarti kita harus selalu merawat node pertama yang tidak memenuhi properti AVL pada jalur dari titik penyisipan ke root dan memperbaikinya.

Jenis Pelanggaran

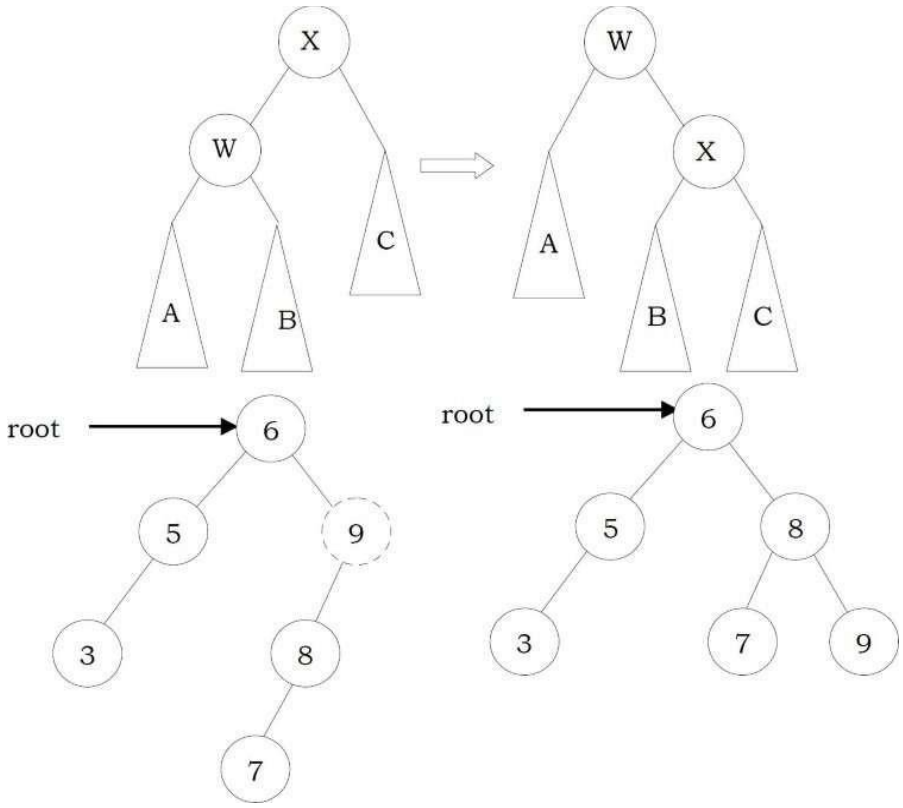
Mari kita asumsikan simpul yang harus diseimbangkan ulang adalah X. Karena simpul mana pun memiliki paling banyak dua anak, dan ketidakseimbangan ketinggian mengharuskan dua ketinggian subpohon X berbeda dua, kita dapat mengamati bahwa pelanggaran mungkin terjadi dalam empat kasus:

1. Penyisipan ke dalam subpohon kiri dari anak kiri X.
2. Penyisipan ke dalam subpohon kanan dari anak kiri X.
3. Penyisipan ke dalam subpohon kiri dari anak kanan X.
4. Penyisipan ke dalam subpohon kanan dari anak kanan X.

Kasus 1 dan 4 simetris dan mudah diselesaikan dengan rotasi tunggal. Demikian pula, kasus 2 dan 3 juga simetris dan dapat diselesaikan dengan rotasi ganda (membutuhkan dua rotasi tunggal).

Rotasi Tunggal

Left Left Rotation (LL Rotation) [Case-1]: Dalam kasus di bawah ini, node X tidak memenuhi properti pohon AVL. Seperti yang telah dibahas sebelumnya, rotasi tidak harus dilakukan pada akar pohon. Secara umum, kita mulai dari simpul yang dimasukkan dan berjalan ke atas pohon, memperbarui informasi keseimbangan di setiap simpul di jalan.



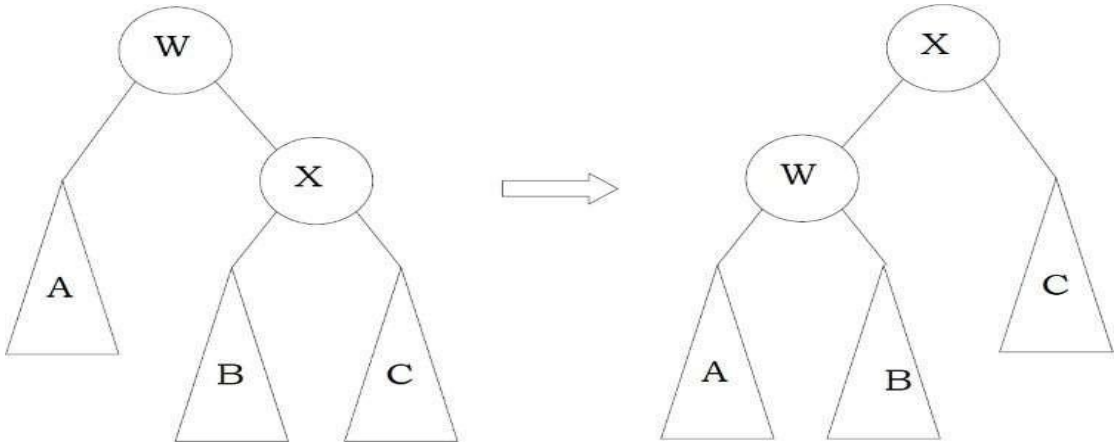
Gambar 6.44 rotasi tunggal

Misalnya, pada gambar di atas, setelah penyisipan 7 di pohon AVL asli di sebelah kiri, simpul 9 menjadi tidak seimbang. Jadi, Kita melakukan satu putaran kiri-kiri pada 9. Hasilnya, Kita mendapatkan pohon di sebelah kanan.

```
struct AVLTreeNode *SingleRotateLeft(struct AVLTreeNode *X ){
    struct AVLTreeNode *W = X→left;
    X→left = W→right;
    W→right = X;
    X→height = max( Height(X→left), Height(X→right) ) + 1;
    W→height = max( Height(W→left), X→height ) + 1;
    return W;  /* New root */
}
```

Kompleksitas Waktu: O(1).
Kompleksitas Ruang: O(1).

Rotasi Kanan Kanan (RR Rotasi) [Kasus-4]: Dalam kasus ini, simpul X tidak memenuhi 254roperty pohon AVL.

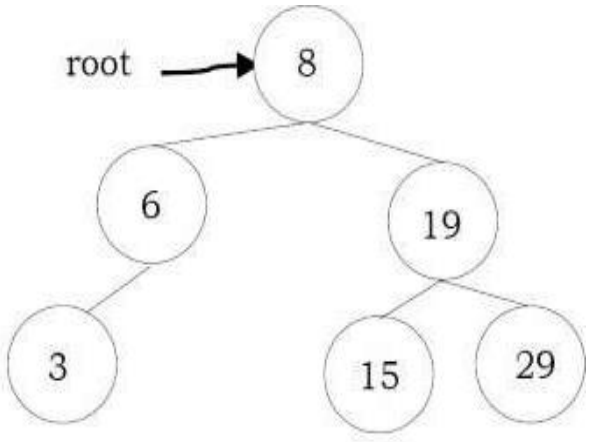


Gambar 6.45 Rotasi kanan kanan

Misalnya, pada gambar, setelah penyisipan 29 di pohon AVL asli di sebelah kiri, simpul 15 menjadi tidak seimbang. Jadi, kita melakukan satu putaran kanan-kanan pada 15. Hasilnya kita mendapatkan pohon di sebelah kanan.

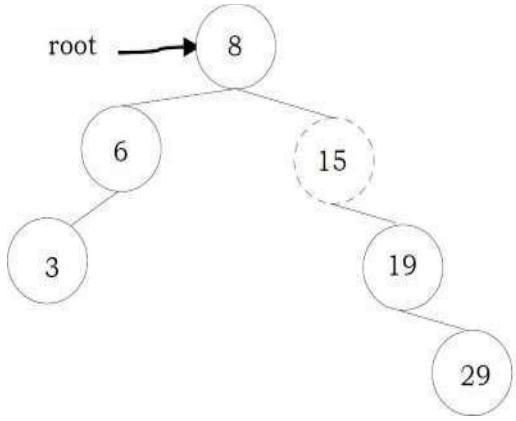
```
struct AVLTreeNode *SingleRotateRight(struct AVLTreeNode *W ) {
    struct AVLTreeNode *X  = W→right;
    W→right = X→left;
    X→left = W;
    W→height = max( Height(W→right), Height(W→left) ) + 1;
    X→height = max( Height(X→right), W→height) + 1;
    return X;
}
```

Kompleksitas Waktu: O(1).
Kompleksitas Ruang: O(1).



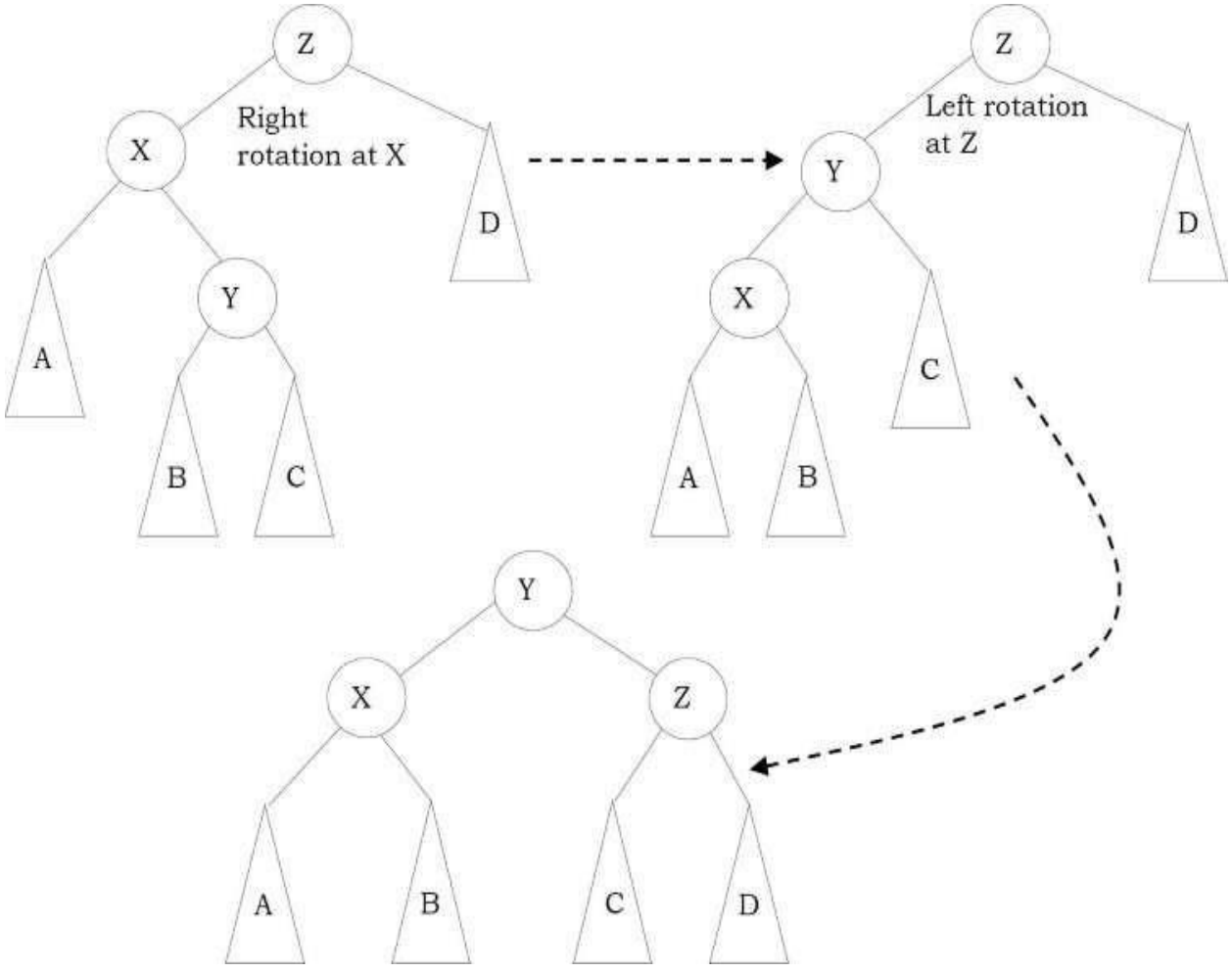
Gambar 6.46 pohon rotasi kanan - kanan

Rotasi Ganda



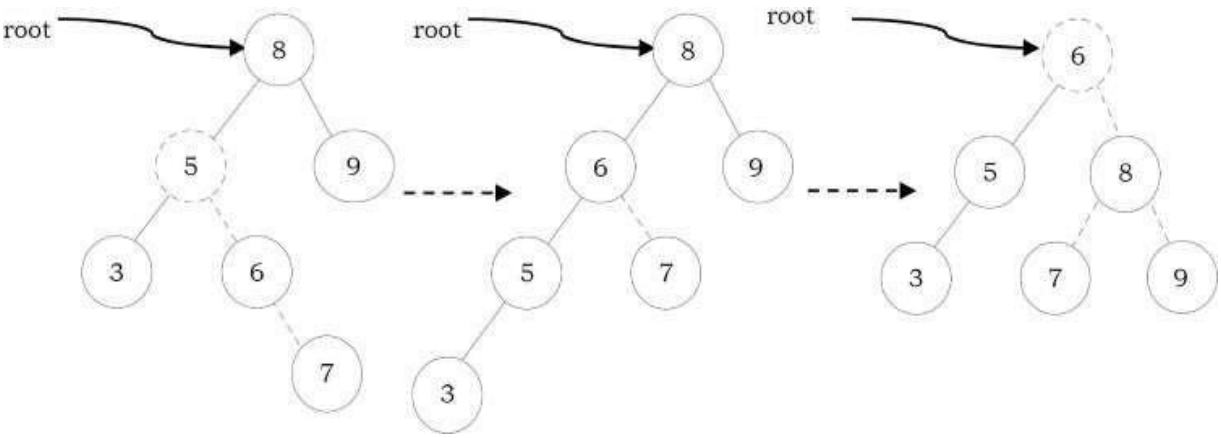
Gambar 6.47 Pohon rotasi kanan – kanan

Rotasi Kiri Kanan (LR Rotasi) [Kasus-2]: Untuk kasus-2 dan kasus-3 rotasi tunggal tidak memperbaiki masalah. Kita perlu melakukan dua putaran.



Gambar 6.43 Rotasi ganda

Sebagai contoh, mari kita perhatikan pohon berikut: Penyisipan 7 membuat 256skenario kasus-2 dan pohon sisi kanan adalah setelah rotasi ganda.

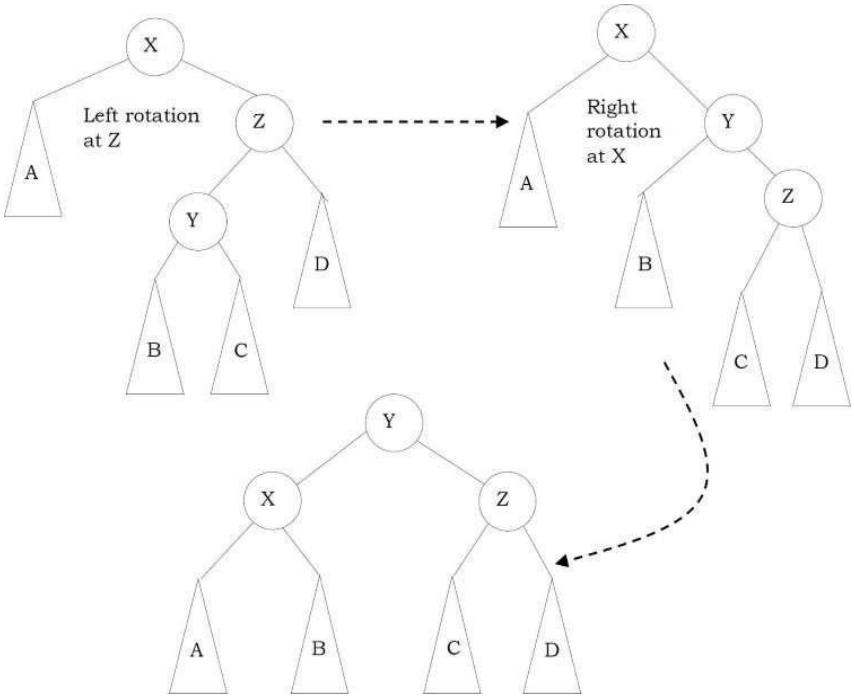


Gambar 6.44 Penyisipan 7 membuat 256skenario kasus-2 dan pohon sisi kanan

Kode untuk rotasi ganda kiri-kanan dapat diberikan sebagai:

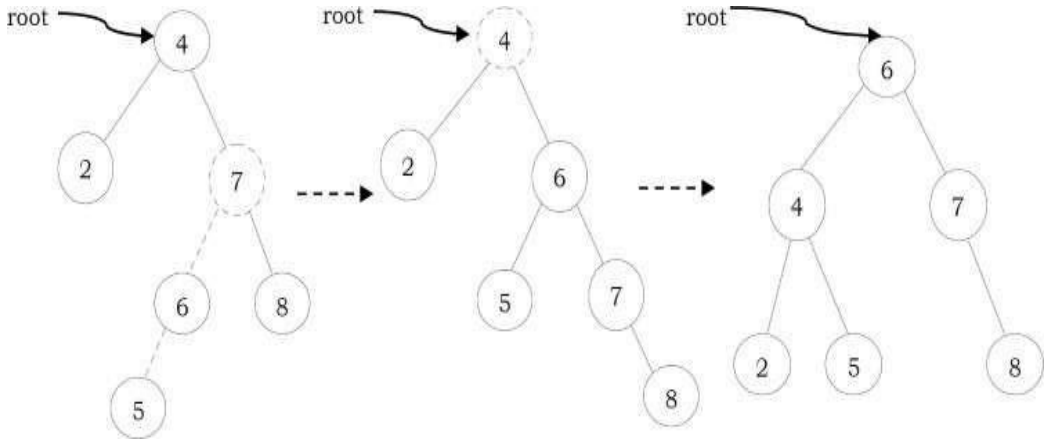
```
struct AVLTreeNode *DoubleRotatewithLeft( struct AVLTreeNode *Z ){
    Z→left = SingleRotateRight( Z→left );
    return SingleRotateLeft(Z);
}
```

Rotasi Kiri Kanan (Rotasi RL) [Kasus-3]: Mirip dengan kasus-2, kita perlu melakukan dua putaran untuk memperbaiki 256cenario ini.



Gambar 6.45 rotasi kiri kanan

Sebagai contoh, mari kita perhatikan pohon berikut: Penyisipan 6 membuat skenario kasus-3 dan pohon sisi kanan adalah setelah rotasi ganda.



Gambar 6.46 Penyisipan 6 membuat 257skenario kasus-3 dan pohon sisi kanan

Penyisipan ke dalam pohon AVL

Penyisipan ke dalam pohon AVL mirip dengan penyisipan BST. Setelah memasukkan elemen, kita hanya perlu memeriksa apakah ada ketidakseimbangan ketinggian. Jika terjadi ketidakseimbangan, panggil fungsi rotasi yang sesuai.

```
struct AVLTreeNode *Insert( struct AVLTreeNode *root, struct AVLTreeNode *parent, int data){
 if( !root) {
        root = (struct AVLTreeNode*) malloc(sizeof (struct AVLTreeNode*) );
        if(!root) {
            printf("Memory Error"); return NULL;
        }
          else {
            root→data = data;
            root→height = 0;
             root→left = root→right = NULL;
          }
}
```

```
        else  if( data < root→data ) {
                root→left = Insert( root→left, root, data );
                if( ( Height( root→left ) - Height( root→right ) ) == 2 ) {
                        if( data < root→left→data )
                                root = SingleRotateLeft( root );
                        else    root = DoubleRotateLeft( root );
                }
        }
        else  if( data > root→data ) {
                root→right = Insert( root→right, root, data );
                if( ( Height( root→right ) - Height( root→left ) ) == 2 ) {
                        if( data < root→right→data )
                            root = SingleRotateRight( root );
                        else  root = DoubleRotateRight( root );
                }
        }
        /* Else data is in the tree already. We'll do nothing */
        root→height = max( Height(root→left), Height(root→right) ) + 1;
        return root;
    }
```

Kompleksitas Waktu: O(logn).

Kompleksitas Ruang: O(logn).

6.14 POHON AVL: MASALAH & SOLUSI

Soal-73 Mengingat ketinggian h, berikan algoritma untuk menghasilkan HB(0).

Solusi: Seperti yang telah kita bahas, HB(0) tidak lain adalah menghasilkan pohon biner penuh. Dalam pohon biner penuh jumlah simpul dengan tinggi h adalah: 2h+1 – 1 (mari kita asumsikan bahwa tinggi pohon dengan satu simpul adalah 0). Akibatnya node dapat diberi nomor sebagai: 1 sampai $2^{h+1} – 1$.

```
struct BinarySearchTreeNode *BuildHB0(int h){
        struct BinarySearchTreeNode *temp;
        if(h == 0) return NULL;
        temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
        temp→left = BuildHB0 (h-1);
        temp→data = count++;      //assume count is a global variable
        temp→right = BuildHB0 (h-1);
        return temp;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(logn), di mana logn menunjukkan ukuran tumpukan maksimum yang sama dengan tinggi pohon.

Soal-74 Apakah ada cara alternatif untuk menyelesaikan Soal-73?

Solusi: Ya, kita bisa menyelesaikannya dengan mengikuti logika Mergesort. Itu berarti, alih-alih bekerja dengan ketinggian, kita bisa mengambil jarak. Dengan pendekatan ini kita tidak membutuhkan counter global untuk dipertahankan. Panggilan awal ke fungsi BuildHBO dapat berupa: BuildHB0(1, 1 h). 1 h melakukan operasi shift untuk menghitung $2^{h+1} - 1$.

```
struct BinarySearchTreeNode *BuildHB0(int l, int r){
    struct BinarySearchTreeNode *temp;
    int mid = l + (r-l)/2;
    if( l > r) return NULL;
    temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
    temp→data = mid;
    temp→left = BuildHB0(l, mid-1);
    temp→right = BuildHB0(mid+1, r);
    return temp;
}
```

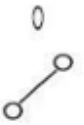
Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(login). Dimana logn menunjukkan ukuran tumpukan maksimum yang sama dengan tinggi pohon.

Soal-75 Membangun pohon AVL minimal dengan tinggi 0,1,2,3,4, dan 5. Berapa jumlah node dalam pohon AVL minimal dengan tinggi 6?

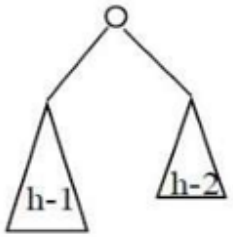
Solusi: Misalkan N(h) adalah jumlah node dalam pohon AVL minimal dengan tinggi h.

$$N(0) = 1$$

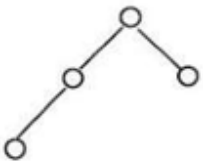


$$N(1) = 2$$

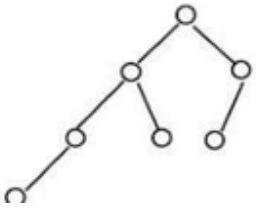
$$N(h) = 1 + N(h-1) + N(h-2)$$



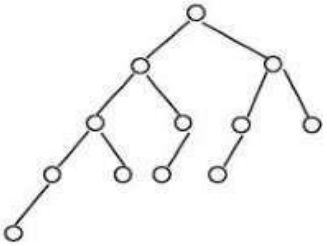
$$N(2) = 1 + N(1) + N(0)$$
$$= 1 + 2 + 1 = 4$$



$$N(3) = 1 + N(2) + N(1)$$
$$= 1 + 4 + 2 = 7$$



$$N(4) = 1 + N(3) + N(2)$$
$$= 1 + 7 + 4 = 12$$



$$N(5) = 1 + N(4) + N(3)$$
$$= 1 + 12 + 7 = 20$$

...

Soal-76 Untuk Soal-73, berapa banyak bentuk berbeda dari pohon AVL minimal setinggi h?

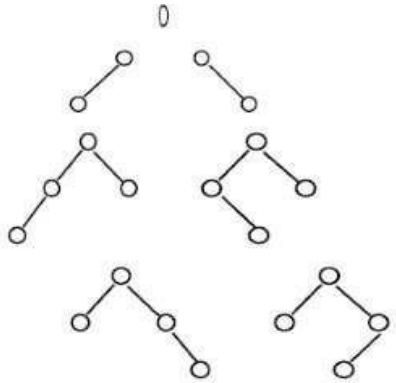
Solusi: Misalkan NS(h) adalah banyaknya bentuk berbeda dari pohon AVL minimal dengan tinggi h.

$$NS(0) = 1$$

$$NS(1) = 2$$

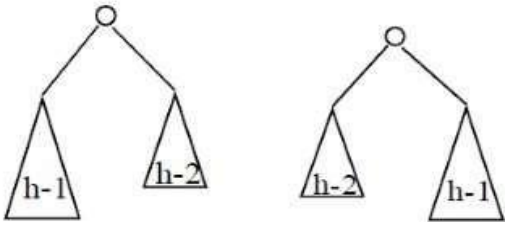
$$NS(2) = 2 * NS(1) * NS(0)$$
$$= 2 * 2 * 1 = 4$$

$$NS(3) = 2 * NS(2) * NS(1)$$
$$= 2 * 4 * 1 = 8$$



...

$$NS(h) = 2 * NS(h-1) * NS(h-2)$$



Soal-77 Diberikan pohon pencarian biner, periksa apakah itu pohon AVL atau bukan?

Solusi: Mari kita asumsikan bahwa IsAVL adalah fungsi yang memeriksa apakah pohon pencarian biner yang diberikan adalah pohon AVL atau bukan. IsAVL mengembalikan -1 jika pohonnya bukan pohon AVL. Selama pemeriksaan, setiap node mengirimkan ketinggiannya ke induknya.

```
int IsAVL(struct BinarySearchTreeNode *root){
        int left, right;
        if(!root)  return 0;
        left = IsAVL(root→left);
        if(left == -1)
            return left;
        right = IsAVL(root→right);
        if(right == -1)
             return right;
        if(abs(left-right)>1)
             return -1;
        return Max(left, right)+1;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

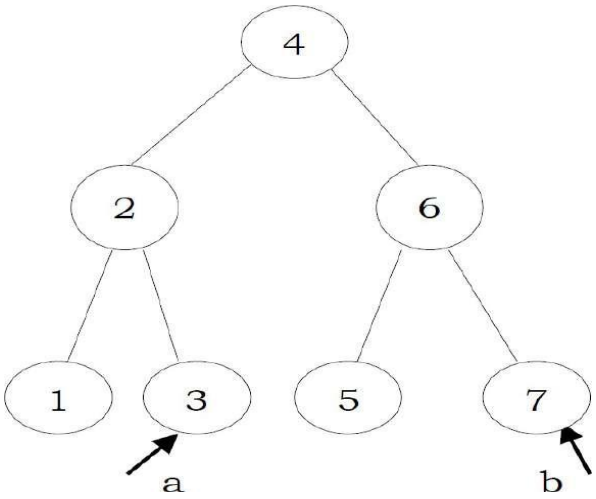
Soal-78 Mengingat ketinggian h, berikan algoritma untuk menghasilkan pohon AVL dengan jumlah node minimum.

Solusi: Untuk mendapatkan jumlah node minimum, isi satu level dengan h – 1 dan yang lainnya dengan h – 2.

```
struct AVLTreeNode *GenerateAVLTree(int h){
        struct AVLTreeNode *temp;
        if(h == 0) return NULL;
        temp = (struct AVLTreeNode *)malloc (sizeof(struct AVLTreeNode));
        temp→left = GenerateAVLTree(h-1);
        temp→data = count++; //assume count is a global variable
        temp→right = GenerateAVLTree(h-2);
        temp→height = temp→left→height+1; // or temp→height = h;
        return temp;
}
```

Soal-79 Diberikan sebuah pohon AVL dengan n item bilangan bulat dan dua bilangan bulat a dan b, di mana a dan b dapat berupa bilangan bulat apa pun dengan a <= b. Menerapkan algoritma untuk menghitung jumlah node dalam rentang [a,b].

Solusi:

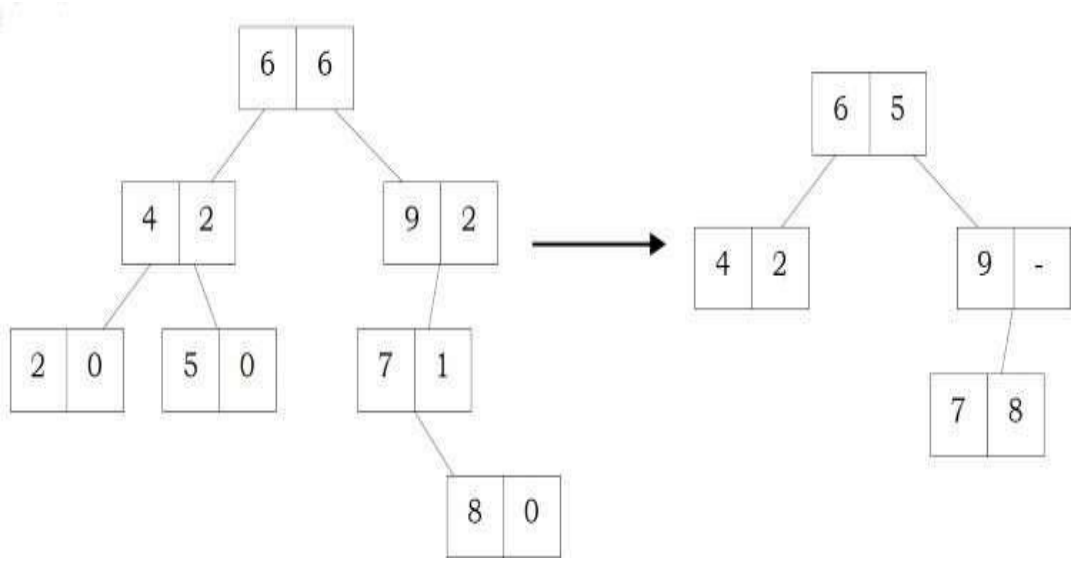


Idenya adalah untuk memanfaatkan properti rekursif dari pohon pencarian biner. Ada tiga kasus yang perlu dipertimbangkan: apakah node saat ini berada dalam rentang [a, b], di sisi kiri rentang [a, b], atau di sisi kanan rentang [a,b]. Hanya subpohon yang mungkin berisi simpul yang akan diproses di bawah masing-masing dari tiga kasus.
Kompleksitasnya mirip dengan traversal berurutan dari pohon tetapi melewatkan sub-pohon kiri atau kanan ketika tidak mengandung jawaban apa pun. Jadi dalam kasus terburuk, jika jangkauan mencakup semua node di pohon, kita perlu melintasi semua n node untuk mendapatkan jawabannya. Oleh karena itu, kompleksitas waktu terburuk adalah O(n).

```
int RangeCount(struct AVLNode *root, int a, int b) {
        if(root == NULL) return 0;
        else if(root→data > b)
                return RangeCount(root→left, a, b);
        else if(root→data < a)
                return RangeCount(root→right, a, b);
        else if(root→data >= a && root→data <= b)
                return RangeCount(root→left, a, b) + RangeCount(root→right, a, b) + 1;
}
```

Jika rentangnya kecil, yang hanya mencakup beberapa elemen dalam subpohon kecil di bagian bawah pohon, kompleksitas waktunya adalah O(h) = O(logn), di mana h adalah tinggi pohon. Ini karena hanya satu jalur yang dilalui untuk mencapai subpohon kecil di bagian bawah dan banyak subpohon tingkat yang lebih tinggi telah dipangkas di sepanjang jalan.
Catatan: Lihat masalah serupa di BST.

Soal-80 Diberikan BST (berlaku untuk pohon AVL juga) di mana setiap node berisi dua elemen data (datanya dan juga jumlah node dalam subpohonnya) seperti yang ditunjukkan di bawah ini. Konversikan pohon ke BST lain dengan mengganti elemen data kedua (jumlah node dalam subpohonnya) dengan data node sebelumnya dalam inorder traversal. Perhatikan bahwa setiap node digabungkan dengan data node sebelumnya secara berurutan. Pastikan juga bahwa konversi terjadi di tempat.



Solusi: Cara paling sederhana adalah dengan menggunakan level order traversal. Jika jumlah elemen di subpohon kiri lebih besar dari jumlah elemen di subpohon kanan, cari elemen maksimum di subpohon kiri dan ganti elemen data kedua simpul saat ini dengannya. Demikian pula, jika jumlah elemen di subpohon kiri lebih kecil dari jumlah elemen di subpohon kanan, cari elemen minimum di subpohon kanan dan ganti elemen data kedua simpul saat ini dengannya.,

```
struct BST *TreeCompression (struct BST *root){
        struct BST *temp, *temp2;
        struct Queue *Q = CreateQueue();
        if(!root) return;
        EnQueue(Q, root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(temp→left && temp→right && temp→left→data2 > temp→right→data2)
                    temp2 = FindMax(temp);
                else temp2 = FindMin(temp);
                 temp→data2 = temp2→data2;   //Process current node
                //Remember to delete this node.
                 DeleteNodeInBST(temp2);
                  if(temp→left)
                            EnQueue(Q, temp→left);
                  if(temp→right)
                            EnQueue(Q, temp→right);
        }
        DeleteQueue(Q);
}
```

Kompleksitas Waktu: Rata-rata O(nlogn) karena BST membutuhkan rata-rata O(logn) untuk menemukan elemen maksimum atau minimum.

Kompleksitas Ruang: O(n). Karena, dalam kasus terburuk, semua node di seluruh level terakhir bisa berada dalam antrian secara bersamaan.

Soal-81 Bisakah kita mengurangi kompleksitas waktu untuk masalah sebelumnya?

Solusi: Mari kita coba menggunakan pendekatan yang mirip dengan apa yang kita ikuti di Soal-60. Ide dibalik solusi ini adalah bahwa inorder traversal dari BST menghasilkan daftar yang diurutkan. Saat melintasi BST secara berurutan, lacak elemen yang dikunjungi dan gabungkan.

```
struct BinarySearchTreeNode * TreeCompression(struct BinarySearchTreeNode *root,
                                              int *previousNodeData){
    if(!root) return NULL;
        TreeCompression(root→left, previousNode);
    if(*previousNodeData == INT_MIN){
        *previousNodeData = root→data;
        free(root);
    }
    if(*previousNodeData != INT_MIN){                    //Process current node
        root→data2 = previousNodeData;
        *previousNodeData = INT_MIN;
    }
    return TreeCompression(root→right, previousNode);
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1). Perhatikan bahwa, Kita masih memiliki ruang tumpukan rekursif untuk traversal inorder.

Soal-82 Diberikan sebuah BST dan sebuah kunci, temukan elemen dalam BST yang paling dekat dengan kunci yang diberikan.

Solusi: Sebagai solusi sederhana, kita dapat menggunakan traversal level-order dan untuk setiap elemen menghitung perbedaan antara kunci yang diberikan dan nilai elemen. Jika selisih tersebut lebih kecil dari selisih yang dipertahankan sebelumnya, maka perbarui selisihnya dengan nilai minimum baru ini. Dengan pendekatan ini, pada akhir traversal kita akan mendapatkan elemen yang paling dekat dengan kunci yang diberikan.

```
int ClosestInBST(struct BinaryTreeNode *root, int key){
        struct BinaryTreeNode *temp, *element;
        struct Queue *Q;
        int difference = INT_MAX;
        if(!root)
            return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(difference > (abs(temp→data-key)){
                        difference = abs(temp→data-key);
                        element = temp;
                }
                if(temp→left)
                        EnQueue (Q, temp→left);
                if(temp→right)
                        EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return element→data;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-83 Untuk Soal-82, dapatkah kita menyelesaikannya dengan menggunakan pendekatan rekursif?

Solusi: Pendekatannya mirip dengan Soal-18. Berikut ini adalah algoritma sederhana untuk menemukan Nilai terdekat di BST.

1. Jika akarnya adalah NULL, maka nilai terdekatnya adalah nol (atau NULL).
2. Jika data root cocok dengan kunci yang diberikan, maka yang terdekat adalah root.
3. Jika tidak, anggap root sebagai yang paling dekat dan lakukan hal berikut:
   A. Jika kunci lebih kecil dari data root, temukan yang terdekat di pohon sisi kiri root secara rekursif dan beri nama temp.
   B. Jika kunci lebih besar dari data root, temukan yang terdekat di pohon sisi kanan root secara rekursif dan beri nama temp.
4. Kembalikan root atau temp tergantung mana yang lebih dekat dengan kunci yang diberikan.

```
struct BinaryTreeNode * ClosestInBST(struct BinaryTreeNode *root, int key){
    struct BinaryTreeNode *temp;
    if(root == NULL)
        return root;
    if(root→data == key)
        return root;
    if(key < root→data){
        if(!root→left)
            return root;
        temp = ClosestInBST(root→left, key);
        return abs(temp→data-key) > abs(root→data-key) ? root : temp;
    }else{
        if(!root→right)
            return root;
        temp = ClosestInBST(root→right, key);
        return abs(temp→data-key) > abs(root→data-key) ? root : temp;
    }
    return NULL;
}
```

Kompleksitas Waktu: O(n) dalam kasus terburuk, dan dalam kasus rata-rata adalah O(logn). Kompleksitas Ruang: O(n) dalam kasus terburuk, dan dalam kasus rata-rata adalah O(logn).

Soal-84 Median dalam deret bilangan bulat tak hingga

Solusi: Median adalah angka tengah dalam daftar angka yang diurutkan (jika kita memiliki jumlah elemen ganjil). Jika kita memiliki jumlah elemen genap, median adalah rata-rata dari dua angka tengah dalam daftar angka yang diurutkan.

Untuk menyelesaikan masalah ini kita dapat menggunakan pohon pencarian biner dengan informasi tambahan pada setiap node, dan jumlah anak pada subpohon kiri dan kanan. Kita juga menyimpan jumlah total node di pohon. Dengan menggunakan informasi tambahan ini, kita dapat menemukan median dalam waktu O(logn), mengambil cabang yang sesuai di pohon berdasarkan jumlah anak di kiri dan kanan node saat ini. Tetapi, kompleksitas penyisipan adalah O(n) karena pohon pencarian biner standar dapat berubah menjadi daftar tertaut jika kita menerima nomor dalam urutan yang diurutkan.

Jadi, mari gunakan pohon pencarian biner seimbang untuk menghindari perilaku terburuk dari pohon pencarian biner standar. Untuk masalah ini, faktor keseimbangan adalah jumlah node di subpohon kiri dikurangi jumlah node di subpohon kanan. Dan hanya node dengan faktor keseimbangan + 1 atau 0 yang dianggap seimbang. Jadi, jumlah node pada subpohon kiri sama dengan atau 1 lebih banyak dari jumlah node pada subpohon kanan, tetapi tidak kurang. Jika kita memastikan faktor keseimbangan ini pada setiap simpul di pohon, maka akar pohon adalah median, jika jumlah elemennya ganjil. Dalam jumlah elemen

genap, median adalah rata-rata dari akar dan penerus terurutnya, yang merupakan keturunan paling kiri dari subpohon kanannya.

Jadi, kompleksitas penyisipan yang mempertahankan kondisi seimbang adalah O(logn) dan menemukan operasi median adalah O(1) dengan asumsi kita menghitung penerus urutan akar pada setiap penyisipan jika jumlah node genap.

Penyisipan dan penyeimbangan sangat mirip dengan pohon AVL. Alih-alih memperbarui ketinggian, Kita memperbarui informasi jumlah node. Pohon pencarian biner seimbang tampaknya menjadi solusi yang paling optimal, penyisipan adalah O(logn) dan median menemukan adalah O(1).

Catatan: Untuk algoritma yang efisien, lihat bab Antrian dan Tumpukan Prioritas.

Soal-85 Mengingat pohon biner, bagaimana Anda menghapus semua setengah node (yang hanya memiliki satu anak)? Perhatikan bahwa kita tidak boleh menyentuh daun.

Solusi: Dengan menggunakan post-order traversal kita dapat menyelesaikan masalah ini secara efisien. Kita pertama-tama memproses anak-anak kiri, lalu anak-anak kanan, dan akhirnya simpul itu sendiri. Jadi kita bentuk pohon baru dari bawah ke atas, mulai dari daun ke arah akar. Pada saat Kita memproses simpul saat ini, subpohon kiri dan kanannya telah diproses.

```
struct BinaryTreeNode *removeHalfNodes(struct BinaryTreeNode *root){
    if (!root)
        return NULL;
    root→left=removeHalfNodes(root→left);
    root→right=removeHalfNodes(root→right);
    if (root→left == NULL && root→right == NULL)
        return root;
    if (root→left == NULL)
        return root→right;
    if (root→right == NULL)
        return root→left;
    return root;
}
```

Kompleksitas Waktu: O(n).

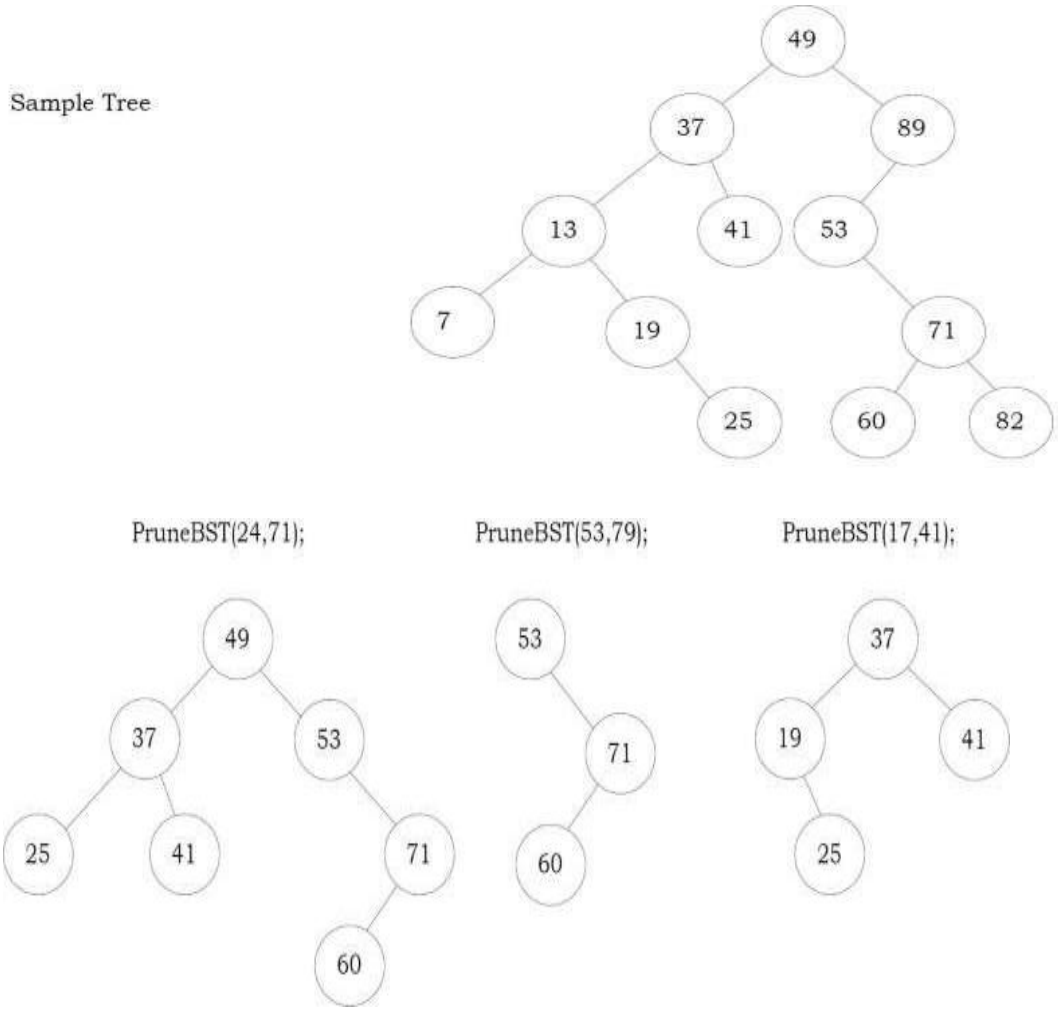
Soal-86 Diberikan pohon biner, bagaimana Anda menghilangkan daunnya?

Solusi: Dengan menggunakan traversal post-order kita dapat menyelesaikan masalah ini (traversal lain juga akan bekerja).

```
struct BinaryTreeNode* removeLeaves(struct BinaryTreeNode* root) {
    if (root != NULL) {
        if (root→left == NULL && root→right == NULL) {
            free(root);
            return NULL;
        } else {
            root→left = removeLeaves(root→left);
            root→right = removeLeaves(root→right);
        }
    }
    return root;
}
```

Kompleksitas Waktu: O(n).

Soal-87 Mengingat BST dan dua bilangan bulat (bilangan bulat minimum dan maksimum) sebagai parameter, bagaimana Anda menghapus (pangkas) elemen yang tidak berada dalam kisaran itu?



Solusi: Pengamatan: Karena kita perlu memeriksa setiap elemen di pohon, dan perubahan subpohon harus tercermin dalam induknya, kita dapat berpikir untuk menggunakan traversal post-order. Jadi Kita memproses node mulai dari daun menuju root. Akibatnya, saat memproses simpul itu sendiri, subpohon kiri dan kanannya adalah BST yang dipangkas yang valid. Pada setiap node Kita akan mengembalikan pointer berdasarkan nilainya, yang kemudian akan ditetapkan ke pointer anak kiri atau kanan induknya, tergantung pada apakah simpul saat ini adalah anak kiri atau kanan dari induknya. Jika nilai node saat ini berada di antara A dan B (A <= data node <= B) maka tidak ada tindakan yang perlu dilakukan, jadi Kita mengembalikan referensi ke node itu sendiri.

Jika nilai simpul saat ini kurang dari A, maka kita mengembalikan referensi ke subpohon kanannya dan membuang subpohon kiri. Karena jika nilai sebuah simpul lebih kecil dari A, maka anak kirinya pasti lebih kecil dari A karena ini adalah pohon pencarian biner. Tetapi anak-anak kanannya mungkin atau mungkin tidak kurang dari A; Kita tidak bisa memastikannya, jadi Kita mengembalikan referensinya. Karena kita melakukan traversal post-order bottom-up, subpohon kanannya sudah merupakan pohon pencarian biner valid yang dipangkas (mungkin NULL), dan subpohon kirinya pasti NULL karena node tersebut pasti kurang dari A dan mereka dihilangkan selama lintas pasca-pesanan.

Situasi serupa terjadi ketika nilai simpul lebih besar dari B, jadi sekarang kita mengembalikan referensi ke subpohon kirinya. Karena jika nilai suatu simpul lebih besar dari B, maka anak kanannya pasti lebih besar dari B. Tetapi anak kirinya bisa lebih besar atau tidak lebih besar dari B; Jadi kita membuang subpohon kanan dan mengembalikan referensi ke subpohon kiri yang sudah valid.

```
struct BinarySearchTreeNode* PruneBST(struct BinarySearchTreeNode *root, int  A, int B){
    if(!root) return NULL;
    root→left= PruneBST(root→left,A,B);
    root→right= PruneBST(root→right,A,B);
    if(A<=root→data && root→data<=B)
        return root;
    if(root→data<A)
        return root→right;
    if(root→data>B)
        return root→left;
}
```

Kompleksitas Waktu: O(n) dalam kasus terburuk dan dalam kasus rata-rata adalah O(logn).

Catatan: *Jika BST yang diberikan adalah pohon AVL maka O(n) adalah kompleksitas waktu rata-rata.*

Soal-88 Mengingat pohon biner, bagaimana Anda menghubungkan semua node yang berdekatan pada tingkat yang sama? Asumsikan bahwa pohon biner yang diberikan memiliki pointer berikutnya bersama dengan pointer kiri dan kanan seperti yang ditunjukkan di bawah ini.

```
struct BinaryTreeNode {
   int data;
   struct BinaryTreeNode *left;
   struct BinaryTreeNode *right;
   struct BinaryTreeNode *next;
};
```

Solusi: Salah satu pendekatan sederhana adalah menggunakan traversal level-order dan terus memperbarui petunjuk berikutnya. Saat melintasi, Kita akan menghubungkan node di level berikutnya.

```
void linkingNodesOfSameLevel(struct BinaryTreeNode *root){
   struct Queue *Q = CreateQueue();
   struct BinaryTreeNode *prev;    // Pointer to the previous node of the current level
   struct BinaryTreeNode *temp;
   int currentLevelNodeCount, nextLevelNodeCount;
   if(!root)
      return;
   EnQueue(Q, root);
   currentLevelNodeCount = 1;
   nextLevelNodeCount = 0;
   prev = NULL;
   while (!IsEmptyQueue(Q)) {
      temp = DeQueue(Q);
      if (temp→left){
         EnQueue(Q, temp→left);
         nextLevelNodeCount++;
      }
      if (temp→right){
         EnQueue(Q, temp→right);
         nextLevelNodeCount++;
      }
      // Link the previous node of the current level to this node
      if (prev)
         prev→next = temp;
      // Set the previous node to the current
      prev = temp;
      currentLevelNodeCount--;
      if (currentLevelNodeCount == 0) {   // if this is the last node of the current level
         currentLevelNodeCount = nextLevelNodeCount;
         nextLevelNodeCount = 0;
         prev = NULL;
      }
   }
}
```

Jika node memiliki node kiri dan kanan, Kita akan menghubungkan kiri ke kanan. Jika simpul memiliki simpul berikutnya, maka tautkan anak paling kanan dari simpul saat ini ke anak paling kiri dari simpul berikutnya.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-89 Bisakah kita meningkatkan kompleksitas ruang untuk Soal-88?

Solusi: Kita dapat memproses pohon tingkat demi tingkat, tetapi tanpa antrian. Bagian logisnya adalah ketika Kita memproses node dari level berikutnya, Kita memastikan bahwa level saat ini telah ditautkan.

```
void linkingNodesOfSameLevel(struct BinaryTreeNode *root) {
    if(!root) return;
    struct BinaryTreeNode *rightMostNode = NULL, *nextHead = NULL, *temp = root;
    //connect next level of current root node level
    while(temp!= NULL){
        if(temp→left!= NULL)
            if(rightMostNode== NULL){
                rightMostNode=temp->left;
                nextHead=temp→left;
            }
            else{
                rightMostNode→next = temp→left;
                rightMostNode = rightMostNode→next;
            }
        if(temp→right!= NULL)
            if(rightMostNode== NULL){
                rightMostNode=temp→right;
                nextHead=temp→right;
            }
            else{
                rightMostNode→next = temp→right;
                rightMostNode = rightMostNode→next;
            }
        temp=temp→next;
    }
    linkingNodesOfSameLevel(nextHead);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O (kedalaman pohon) untuk ruang tumpukan.

Soal-90 Asumsikan bahwa himpunan S dari n bilangan disimpan dalam beberapa bentuk pohon pencarian biner seimbang; yaitu kedalaman pohon adalah O(logn). Selain nilai kunci dan pointer ke anak, asumsikan bahwa setiap node berisi jumlah node di subpohon-nya. Tentukan alasan mengapa pohon biner

seimbang dapat menjadi pilihan yang lebih baik daripada pohon biner lengkap untuk menyimpan himpunan S.

Solusi: Implementasi pohon biner seimbang membutuhkan lebih sedikit ruang RAM karena kita tidak perlu menyimpan pohon lengkap dalam RAM (karena mereka menggunakan pointer).

Soal-91 Untuk Soal-90, tentukan alasan mengapa pohon biner lengkap dapat menjadi pilihan yang lebih baik daripada pohon biner seimbang untuk menyimpan himpunan S.

Solusi: Pohon biner lengkap lebih hemat ruang karena kita tidak memerlukan tanda tambahan. Pohon biner yang seimbang biasanya membutuhkan lebih banyak ruang karena kita perlu menyimpan beberapa flag. Misalnya, di pohon Merah-Hitam kita perlu menyimpan sedikit untuk warnanya. Juga, pohon biner lengkap dapat disimpan dalam RAM sebagai array tanpa menggunakan pointer.

Soal-92 Diberikan pohon biner, temukan jumlah jalur maksimum. Jalur dapat dimulai dan diakhiri pada setiap simpul di pohon.

Solusi:

```
int maxPathSum(struct BinaryTreeNode *root){
        int maxValue = INT_MIN;
        return maxPathSumRec(root);
}
int max(int a, int b){
        if (a>b)  return a;
        else return b;
}
int maxPathSumRec(struct BinaryTreeNode *root){
        if (root == NULL) return 0;
        int leftSum = maxPathSumRec(root→left);
        int rightSum = maxPathSumRec(root→right);
        if (leftSum < 0 && rightSum < 0){
                maxValue = max(maxValue, root→data);
                return root→data;
        }
        if (leftSum>0 && rightSum>0)
                maxValue = max(maxValue, root→data + leftSum + rightSum);
        maxValueUp = max(leftSum, rightSum) + root→data;
        maxValue = max(maxValue, maxValueUp);
        return maxValueUp;
}
```

Soal-93 Biarkan T menjadi pohon biner yang tepat dengan akar r. Perhatikan algoritma berikut.

```
Algorithm TreeTraversal(r):
    if (!r) return 1;
    else {
            a = TreeTraversal(r→left);
            b = TreeTraversal(r→right);
            return a + b;
    }
```

Apa yang dilakukan algoritma?

A. Selalu mengembalikan nilai 1.
B. Ini menghitung jumlah node di pohon.
C. Ini menghitung kedalaman node.
D. Menghitung tinggi pohon.
E. Ini menghitung jumlah daun di pohon.

Solusi: E

6.15 VARIASI LAIN PADA POHON

Pada bagian ini, mari kita menghitung kemungkinan representasi pohon lainnya. Di bagian sebelumnya, kita telah melihat pohon AVL, yang merupakan pohon pencarian biner (BST) dengan properti penyeimbang. Sekarang, mari kita lihat beberapa pohon pencarian biner yang lebih seimbang: Pohon Merah-hitam dan Pohon Splay.

6.15.1 Pohon Merah-Hitam

Di pohon Merah-hitam setiap simpul dikaitkan dengan atribut tambahan: warna, yang merah atau hitam. Untuk mendapatkan kompleksitas logaritmik, Kita memberlakukan batasan berikut.

Definisi: Pohon merah-hitam adalah pohon pencarian biner yang memenuhi properti berikut:

- Properti Root: root berwarna hitam
- Properti Eksternal: setiap daun berwarna hitam
- Properti Internal: anak-anak dari simpul merah berwarna hitam
- Properti Kedalaman: semua daun memiliki warna hitam yang sama

Mirip dengan pohon AVL, jika pohon Merah-hitam menjadi tidak seimbang, maka Kita melakukan rotasi untuk memperkuat properti penyeimbang. Dengan pohon Merah-hitam, kita dapat melakukan operasi berikut dalam O(logn) dalam kasus terburuk, di mana n adalah jumlah node di pohon.

- Penyisipan, Penghapusan
- Menemukan pendahulu, penerus
- Mencari minimum, maksimum

6.15.2 Pohon Bermain

Splay-tree adalah BST dengan properti yang dapat menyesuaikan sendiri. Properti lain yang menarik dari splay-tree adalah: dimulai dengan pohon kosong, setiap urutan operasi K dengan maksimum n node membutuhkan kompleksitas waktu O(Klogn) dalam kasus terburuk.

Pohon splay lebih mudah diprogram dan juga memastikan akses lebih cepat ke item yang baru saja diakses. Mirip dengan pohon AVL dan Merah-Hitam, pada titik mana pun ketika pohon splay menjadi tidak seimbang, kita dapat melakukan rotasi untuk memperkuat properti penyeimbang.

Splay-tree tidak dapat menjamin kompleksitas O(logn) dalam kasus terburuk. Tapi itu memberikan kompleksitas O(logn) yang diamortisasi. Meskipun operasi individu bisa mahal, setiap urutan operasi mendapatkan kompleksitas perilaku logaritmik. Satu operasi mungkin membutuhkan lebih banyak waktu (satu operasi dapat memakan waktu O(n) waktu) tetapi operasi berikutnya mungkin tidak mengambil kompleksitas kasus terburuk dan rata-rata per kompleksitas operasi adalah O{logn).

6.15.3 B-Pohon

B-Tree seperti pohon self-balancing lainnya seperti AVL dan Red-black tree sehingga mempertahankan keseimbangan node saat operasi dilakukan terhadapnya. B-Tree memiliki properti berikut:

- Derajat minimum "£" di mana, kecuali simpul akar, semua simpul lain harus memiliki tidak kurang dari t – 1 kunci
- Setiap node dengan n kunci memiliki n + 1 anak
- Kunci di setiap node berbaris di mana k1 < k2 < .. kn
- Setiap node tidak boleh memiliki lebih dari 2t-l kunci, jadi 2t anak
- Node root setidaknya harus berisi satu kunci. Tidak ada simpul akar jika pohon kosong.
- Pohon tumbuh secara mendalam hanya ketika simpul akar terbelah.

Tidak seperti binary-tree, setiap node dari b-tree mungkin memiliki sejumlah variabel kunci dan anak. Kunci disimpan dalam urutan yang tidak berkurang. Setiap kunci memiliki anak terkait yang merupakan akar dari subpohon yang berisi semua node dengan kunci kurang dari atau sama dengan kunci tetapi lebih besar dari kunci sebelumnya. Sebuah simpul juga memiliki anak paling kanan tambahan yang merupakan akar dari subpohon yang berisi semua kunci yang lebih besar daripada kunci apa pun dalam simpul tersebut.

Sebuah b-tree memiliki jumlah minimal anak yang diijinkan untuk setiap node yang dikenal sebagai faktor minimisasi. Jika t adalah faktor minimalisasi ini, setiap node harus memiliki setidaknya t – 1 kunci. Dalam keadaan tertentu, simpul akar diperbolehkan untuk melanggar properti ini dengan memiliki kurang dari t – 1 kunci. Setiap node mungkin memiliki paling banyak 2t – 1 kunci atau, setara, 2t anak.

Karena setiap node cenderung memiliki faktor percabangan yang besar (sejumlah besar anak), biasanya diperlukan untuk melintasi node yang relatif sedikit sebelum menemukan kunci yang diinginkan. Jika akses ke setiap node memerlukan akses disk, maka B-tree akan meminimalkan jumlah akses disk yang diperlukan. Faktor minimalisasi biasanya dipilih sehingga ukuran total setiap node sesuai dengan kelipatan ukuran blok perangkat penyimpanan yang mendasarinya. Pilihan ini menyederhanakan dan mengoptimalkan akses

disk. Akibatnya, B-tree adalah struktur data yang ideal untuk situasi di mana semua data tidak dapat berada di penyimpanan utama dan akses ke penyimpanan sekunder relatif mahal (atau memakan waktu).

Untuk mencari pohon, ini mirip dengan pohon biner kecuali bahwa kuncinya dibandingkan beberapa kali dalam simpul yang diberikan karena simpul tersebut berisi lebih dari 1 kunci. Jika kunci ditemukan di node, pencarian dihentikan. Jika tidak, ia bergerak ke bawah di mana pada anak ditunjukkan oleh ci di mana kunci k < ki.

Penyisipan kunci dari pohon-B terjadi dari fasion bawah. Ini berarti ia berjalan menuruni pohon dari akar ke simpul anak target terlebih dahulu. Jika anak tidak penuh, kuncinya cukup dimasukkan. Jika penuh, simpul anak terbelah di tengah, kunci median bergerak ke atas ke induk, kemudian kunci baru dimasukkan. Saat memasukkan dan berjalan menuruni pohon, jika simpul akar ditemukan penuh, itu dipecah terlebih dahulu dan Kita memiliki simpul akar baru. Kemudian operasi penyisipan normal dilakukan.

Penghapusan kunci lebih rumit karena perlu mempertahankan jumlah kunci di setiap node untuk memenuhi batasan. Jika kunci ditemukan di simpul daun dan menghapusnya tetap menjaga jumlah kunci di simpul tidak terlalu rendah, itu hanya dilakukan segera. Jika dilakukan pada simpul dalam, pendahulu kunci di simpul anak yang sesuai dipindahkan untuk menggantikan kunci di simpul dalam. Jika memindahkan pendahulunya akan menyebabkan simpul anak melanggar batasan jumlah simpul, simpul anak saudara digabungkan dan kunci di simpul dalam dihapus.

6.15.4 Pohon Yang Diperbesar

Di bagian sebelumnya, kita telah melihat berbagai masalah seperti menemukan elemen Kth – terkecil - di pohon dan yang serupa lainnya. Dari semua masalah kompleksitas terburuk adalah O(n), di mana n adalah jumlah node di pohon. Untuk melakukan operasi seperti itu di O(logn), pohon augmented berguna. Di pohon-pohon ini, informasi tambahan ditambahkan ke setiap simpul dan data tambahan itu tergantung pada masalah yang Kita coba selesaikan.

Misalnya, untuk menemukan elemen Kth dalam pohon pencarian biner, mari kita lihat bagaimana pohon yang diperbesar menyelesaikan masalah. Mari kita asumsikan bahwa kita menggunakan pohon Merah-Hitam sebagai BST seimbang (atau BST seimbang apa pun) dan menambah informasi ukuran dalam data node. Untuk node X yang diberikan di pohon Merah-Hitam dengan ukuran bidang (X) sama dengan jumlah node di subpohon dan dapat dihitung sebagai:

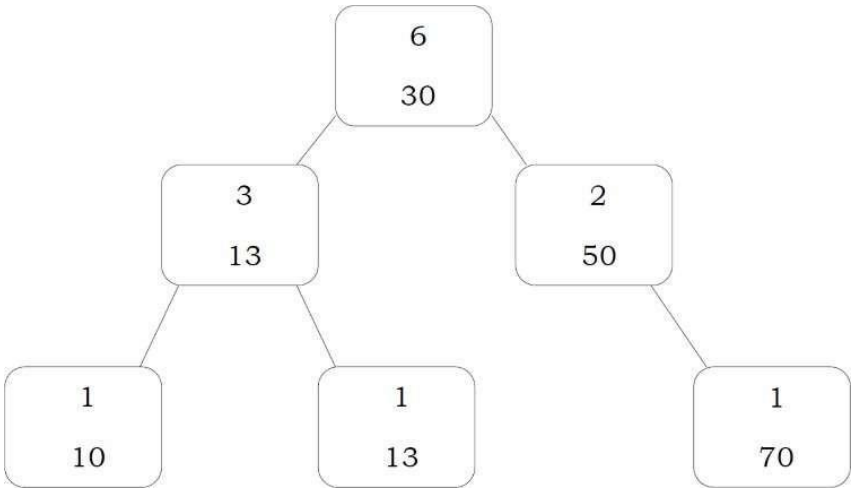
$$size(X) = size(X \rightarrow left) + size(X \rightarrow right)) + 1$$

Kth - terkecil - operasi dapat didefinisikan sebagai:

```
struct BinarySearcTreeNode *KthSmallest (struct BinarySearcTreeNode *X, int K) {
    int r = size(X→left) + 1;
    if(K == r)
        return X;
    if(K < r)
        return KthSmallest (X→left, K);
    if(K > r)
        return KthSmallest (X→right, K-r);
}
```

Kompleksitas Waktu: O(logn).
Kompleksitas Ruang: O(logn).

Contoh: *Dengan informasi ukuran ekstra, pohon yang diperbesar akan terlihat seperti:*



Gambar 6.47 pohon yang diperbesar dengan ukuran infromasi ekstra

6.15.5 Pohon Interval (Pohon Segmen)

Kita sering menghadapi pertanyaan yang melibatkan kueri yang dibuat dalam array berdasarkan rentang. Misalnya, untuk larik bilangan bulat tertentu, berapa jumlah maksimum dalam rentang hingga , di mana dan tentu saja berada dalam batas larik. Untuk mengulangi entri tersebut dengan interval yang berisi nilai tertentu, kita dapat menggunakan array sederhana. Tetapi jika kita membutuhkan akses yang lebih efisien, kita membutuhkan struktur data yang lebih canggih.

Skema penyimpanan berbasis array dan pencarian brute force melalui seluruh array dapat diterima hanya jika pencarian tunggal dilakukan, atau jika jumlah elemen kecil. Misalnya, jika Anda mengetahui semua nilai larik yang diinginkan sebelumnya, Anda hanya perlu melewati satu larik. Namun, jika Anda dapat secara interaktif menentukan operasi pencarian yang berbeda pada waktu yang berbeda, pencarian brute force menjadi tidak praktis karena setiap elemen dalam array harus diperiksa selama setiap operasi pencarian.

Jika Anda mengurutkan larik dalam urutan menaik dari nilai larik, Anda dapat menghentikan pencarian sekuensial ketika Anda mencapai objek yang nilainya rendah lebih

besar dari elemen yang kita cari. Sayangnya, teknik ini menjadi semakin tidak efektif seiring dengan meningkatnya nilai rendah, karena lebih sedikit operasi pencarian yang dihilangkan. Artinya, bagaimana jika kita harus menjawab banyak pertanyaan seperti ini? – apakah kekerasan masih merupakan pilihan yang baik?

Contoh lain adalah ketika kita perlu mengembalikan jumlah dalam rentang tertentu. Kita juga dapat memaksa ini, tetapi masalah untuk sejumlah besar kueri masih tetap ada. Jadi apa yang bisa kita lakukan? Dengan sedikit pemikiran, kita dapat menemukan pendekatan seperti mempertahankan larik terpisah dari n elemen, di mana n adalah ukuran larik asli, di mana setiap indeks menyimpan jumlah semua elemen dari 0 hingga indeks itu. Jadi pada dasarnya Kita memiliki sedikit preprocessing yang menurunkan waktu kueri dari kasus terburuk O(n) ke O(1). Sekarang ini bagus sejauh menyangkut array statis, tetapi, bagaimana jika kita diminta untuk melakukan pembaruan pada array juga?

Pendekatan pertama memberi kita waktu kueri O(n), tetapi waktu pembaruan O(1). Pendekatan kedua, di sisi lain, memberi kita waktu kueri O(1), tetapi waktu pembaruan O(n). Jadi, mana yang kita pilih?

Pohon interval juga merupakan pohon pencarian biner dan mereka menyimpan informasi interval dalam struktur simpul. Itu berarti, Kita mempertahankan satu set interval n [i1, i2] sedemikian rupa sehingga salah satu interval yang berisi titik kueri Q (jika ada) dapat ditemukan secara efisien. Pohon interval digunakan untuk melakukan jangkauan
query secara efisien.

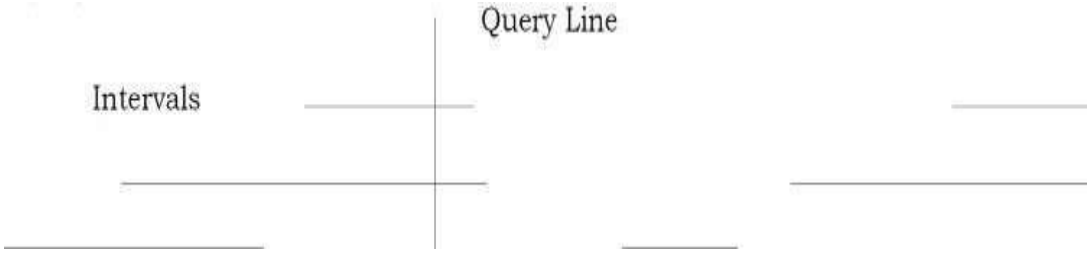
Pohon segmen adalah struktur data seperti tumpukan yang dapat digunakan untuk membuat operasi pembaruan/kueri pada interval array dalam waktu logaritmik. Kita mendefinisikan pohon segmen untuk interval [i,j] dengan cara rekursif berikut:

- Node root (node pertama dalam array) akan menyimpan informasi untuk interval [i,j]
- Jika i < y, anak kiri dan kanan akan menyimpan informasi untuk interval $\left[i,\ \frac{i+j}{2}\right]$ dan $\left[\frac{i+j}{2}+1,\ j\right]$

Pohon segmen (juga disebut pohon segmen dan pohon interval) adalah struktur data yang keren, terutama digunakan untuk kueri rentang. Ini adalah pohon biner tinggi seimbang dengan struktur statis. Node dari pohon segmen sesuai dengan berbagai interval, dan dapat ditambah dengan informasi yang sesuai yang berkaitan dengan interval tersebut. Ini agak kurang kuat daripada pohon biner seimbang karena struktur statisnya, tetapi karena sifat operasi rekursif pada segtree, ini sangat mudah untuk dipikirkan dan dikodekan.

Kita dapat menggunakan pohon segmen untuk memecahkan masalah kueri minimum/maksimum jangkauan. Kompleksitas waktu adalah T(nlogn) di mana O(n) adalah waktu yang dibutuhkan untuk membangun pohon dan setiap kueri membutuhkan waktu O(logn).

Contoh: Diberikan satu set interval: S= {[2-5], [6-7], [6-10], [8-9], [12-15], [15-23], [25- 30]}. Kueri dengan Q = 9 mengembalikan [6,10] atau [8,9] (anggap ini adalah interval yang berisi 9 di antara semua interval). Kueri dengan Q = 23 mengembalikan [15, 23].



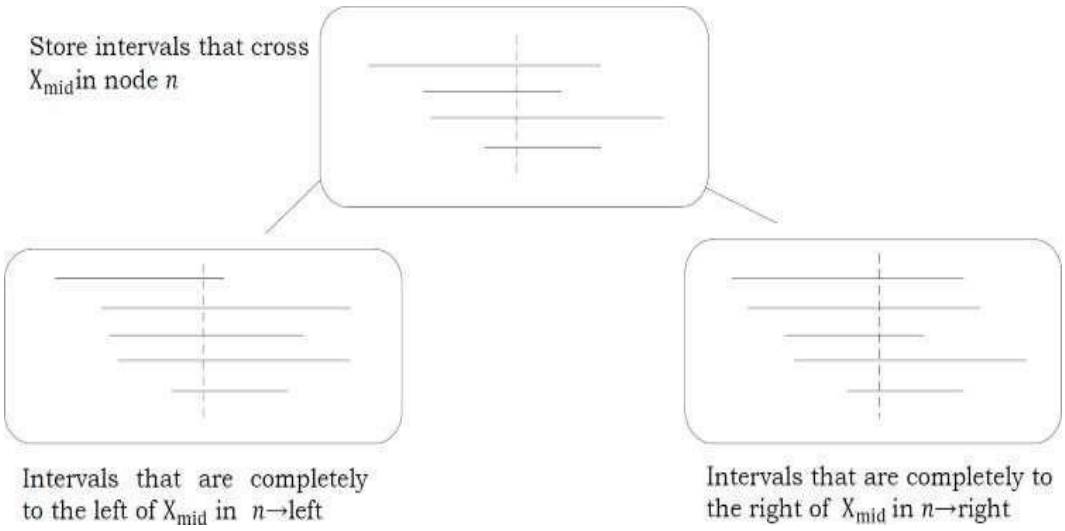
Konstruksi Pohon Interval: Mari kita asumsikan bahwa kita diberikan himpunan S dari n interval (disebut segmen). Interval n ini akan memiliki 2n titik akhir. Sekarang, mari kita lihat bagaimana membangun pohon interval.

Algoritma:

Bangun pohon secara rekursif pada set interval 5 sebagai berikut:

- Urutkan 2n titik akhir
- Biarkan Xmid menjadi titik median

Kompleksitas Waktu untuk membangun pohon interval: O(nlogn). Karena kita memilih median, Pohon Interval kira-kira akan seimbang. Ini memastikan bahwa, Kita membagi set titik akhir menjadi dua setiap kali. Kedalaman pohon adalah O(logn). Untuk mempermudah proses pencarian, umumnya Xmid disimpan dengan setiap node.



6.15.6 Pohon Kambing Hitam

Pohon kambing hitam adalah pohon pencarian biner self-balancing, ditemukan oleh Arne Andersson. Ini memberikan waktu pencarian O(logn) kasus terburuk, dan waktu penyisipan dan penghapusan O(logn) diamortisasi (rata-rata).

Pohon AVL menyeimbangkan kembali setiap kali ketinggian dua subpohon bersaudara berbeda lebih dari satu; pohon kambing hitam menyeimbangkan kembali setiap kali ukuran

anak melebihi rasio tertentu dari orang tuanya, rasio yang dikenal sebagai a. Setelah memasukkan elemen, Kita melintasi kembali pohon. Jika kita menemukan ketidakseimbangan di mana ukuran anak melebihi ukuran induk dikali alfa, kita harus membangun kembali subpohon di induk, kambing hitam.

Mungkin ada lebih dari satu kambing hitam yang mungkin, tetapi kita hanya perlu memilih satu. Kambing hitam yang paling optimal sebenarnya ditentukan oleh keseimbangan tinggi badan. Saat menghapusnya, kita melihat apakah ukuran total pohon kurang dari alfa ukuran terbesar sejak pembangunan kembali terakhir pohon. Jika demikian, Kita membangun kembali seluruh pohon. Alfa untuk pohon kambing hitam dapat berupa angka antara 0,5 dan 1,0. Nilai 0,5 akan memaksa keseimbangan sempurna, sementara 1,0 akan menyebabkan penyeimbangan kembali tidak pernah terjadi, secara efektif mengubahnya menjadi BST.

BAB 7
ANTRIAN PRIORITAS DAN TUMPUKAN

7.1 APA ITU ANTRIAN PRIORITAS

Dalam beberapa situasi kita mungkin perlu menemukan elemen minimum/maksimum di antara kumpulan elemen. Kita dapat melakukan ini dengan bantuan ADT Antrian Prioritas. ADT antrian prioritas adalah struktur data yang mendukung operasi Insert dan DeleteMin (yang mengembalikan dan menghapus elemen minimum) atau DeleteMax (yang mengembalikan dan menghapus elemen maksimum).

Operasi ini setara dengan operasi EnQueue dan DeQueue dari antrian. Perbedaannya adalah, dalam antrian prioritas, urutan elemen yang masuk ke antrian mungkin tidak sama dengan saat elemen tersebut diproses. Contoh penerapan antrian prioritas adalah penjadwalan pekerjaan, yang diprioritaskan daripada melayani di first come first serve.

Gambar 7.1 antrian prioritas

Antrian prioritas disebut antrean prioritas naik, jika item dengan kunci terkecil memiliki prioritas tertinggi (artinya, hapus elemen terkecil selalu). Demikian pula, antrian prioritas dikatakan sebagai antrian prioritas menurun jika item dengan kunci terbesar memiliki prioritas tertinggi (selalu hapus elemen maksimum). Karena kedua jenis ini simetris, Kita akan berkonsentrasi pada salah satunya: antrian prioritas naik.

7.2 ADT ANTRIAN PRIORITAS

Operasi berikut membuat antrian prioritas menjadi ADT.

Operasi Antrian Prioritas Utama

Antrian prioritas adalah wadah elemen, masing-masing memiliki kunci terkait.

- Sisipkan (kunci, data): Menyisipkan data dengan kunci ke antrian prioritas. Elemen diurutkan berdasarkan kunci.
- DeleteMin/DeleteMax: Menghapus dan mengembalikan elemen dengan kunci terkecil/terbesar.
- GetMinimum/GetMaximum: Mengembalikan elemen dengan kunci terkecil/terbesar tanpa menghapusnya.

Operasi Antrian Prioritas Bantu

- kth - Terkecil/kth – Terbesar: Mengembalikan kth -Terkecil/kth –Kunci terbesar dalam antrian prioritas.
- Ukuran: Mengembalikan jumlah elemen dalam antrian prioritas.
- Heap Sort: Mengurutkan elemen dalam antrian prioritas berdasarkan prioritas (kunci).

7.3 APLIKASI ANTRIAN PRIORITAS

Antrian prioritas memiliki banyak aplikasi - beberapa di antaranya tercantum di bawah ini:

- Kompresi data: Algoritma Huffman Coding
- Algoritma jalur terpendek: algoritma Dijkstra
- Algoritma pohon merentang minimum: Algoritma Prim
- Simulasi berbasis peristiwa: pelanggan dalam antrean
- Soal pemilihan: Menemukan k- elemen terkecil

7.4 IMPLEMENTASI ANTRIAN PRIORITAS

Sebelum membahas implementasi yang sebenarnya, mari kita menghitung opsi yang mungkin.

Implementasi Array Tidak Terurut

Elemen dimasukkan ke dalam array tanpa memikirkan urutannya. Penghapusan (DeleteMax) dilakukan dengan mencari kunci dan kemudian menghapus.

Kompleksitas penyisipan: O(1).

Kompleksitas DeleteMin: O(n).

Implementasi Daftar Tidak Terurut

Ini sangat mirip dengan implementasi array, tetapi alih-alih menggunakan array, daftar tertaut digunakan.

Kompleksitas penyisipan: O(1).

Kompleksitas DeleteMin: O(n).

Implementasi Array yang Dipesan

Elemen dimasukkan ke dalam array dalam urutan yang diurutkan berdasarkan bidang kunci. Penghapusan dilakukan hanya pada satu ujung.

Kompleksitas penyisipan: O(n).

Kompleksitas DeleteMin: O(1).

Implementasi Daftar Pesanan

Elemen dimasukkan ke dalam daftar dalam urutan yang diurutkan berdasarkan bidang kunci. Penghapusan dilakukan hanya pada satu ujung, sehingga mempertahankan status antrian prioritas. Semua fungsi lain yang terkait dengan daftar tertaut ADT dilakukan tanpa modifikasi.

Kompleksitas penyisipan: O(n).

Kompleksitas DeleteMin: O(1).

Implementasi Pohon Pencarian Biner

Baik penyisipan dan penghapusan membutuhkan O(logn) rata-rata jika penyisipan dilakukan secara acak (lihat Bab Pohon).

Implementasi Pohon Pencarian Biner Seimbang

Baik penyisipan dan penghapusan mengambil O(logn) dalam kasus terburuk (lihat bab Pohon).

Implementasi Tumpukan Biner

Pada bagian selanjutnya kita akan membahas ini secara lengkap. Untuk saat ini, asumsikan bahwa implementasi tumpukan biner memberikan kompleksitas O(logn) untuk pencarian, penyisipan dan penghapusan dan O(1) untuk menemukan elemen maksimum atau minimum.

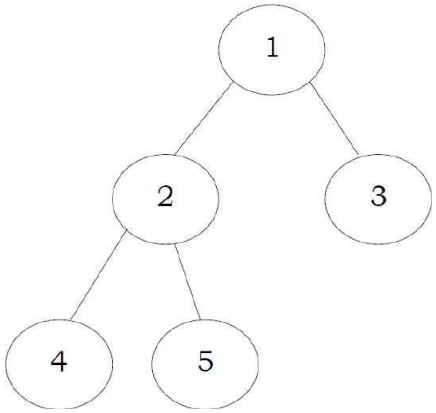
Membandingkan Implementasi

Tabel 7.1 membandingkan implementasi

Implementation	Insertion	Deletion (DeleteMax)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
Binary Search Trees	$logn$ (average)	$logn$ (average)	$logn$ (average)
Balanced Binary Search Trees	$logn$	$logn$	$logn$
Binary Heaps	$logn$	$logn$	1

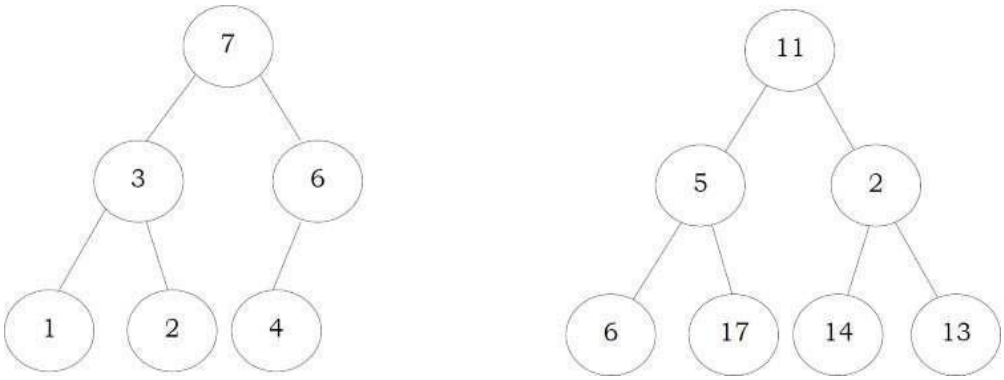
7.5 TUMPUKAN DAN TUMPUKAN BINER

Heap adalah pohon dengan beberapa sifat khusus. Persyaratan dasar dari sebuah heap adalah bahwa nilai dari sebuah node harus (atau) dari nilai anak-anaknya. Ini disebut properti tumpukan. Heap juga memiliki properti tambahan bahwa semua daun harus berada pada level h atau h – 1 (di mana h adalah ketinggian pohon) untuk beberapa h > 0 (pohon biner lengkap). Itu berarti heap harus membentuk pohon biner lengkap (seperti yang ditunjukkan di bawah).



Gambar 7.2 pohon dengan beberapa sifat khusus

Dalam contoh di bawah, pohon kiri adalah tumpukan (setiap elemen lebih besar dari anak-anaknya) dan pohon kanan bukan tumpukan (karena 11 lebih besar dari 2).

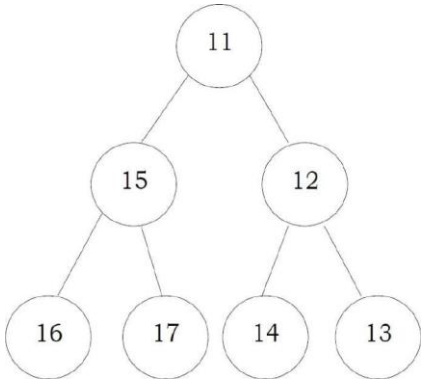


Gambar 7.3 pohon tumpukan dan bukan tumpukan

Jenis Tumpukan?

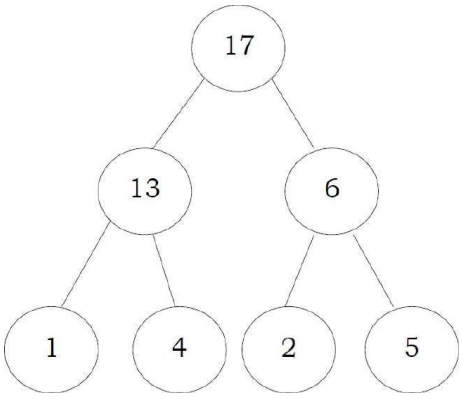
Berdasarkan sifat heap kita dapat mengklasifikasikan heap menjadi dua jenis:

- Min heap: Nilai sebuah simpul harus lebih kecil atau sama dengan nilai turunannya



Gambar 7.4 *min heap*

- **Max heap:** Nilai sebuah simpul harus lebih besar atau sama dengan nilai turunannya

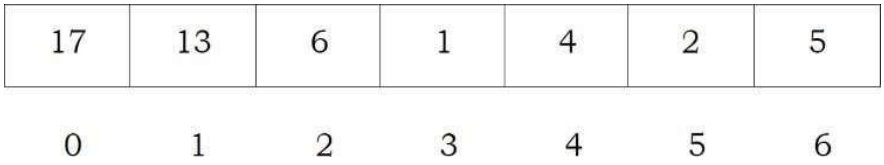


Gambar 7.5 *max heap*

7.6 TUMPUKAN BINER

Dalam tumpukan biner, setiap simpul dapat memiliki hingga dua anak. Dalam praktiknya, tumpukan biner sudah cukup dan Kita berkonsentrasi pada tumpukan min biner dan tumpukan maksimal biner untuk pembahasan selanjutnya.

Mewakili Heap: Sebelum melihat operasi heap, mari kita lihat bagaimana heap dapat direpresentasikan. Salah satu kemungkinan adalah menggunakan array. Karena tumpukan membentuk pohon biner lengkap, tidak akan ada pemborosan lokasi. Untuk pembahasan di bawah ini mari kita asumsikan bahwa elemen disimpan dalam array, yang dimulai pada indeks 0. Max heap sebelumnya dapat direpresentasikan sebagai:



Gambar 7.6 presentasi *max heap*

Catatan: *Untuk pembahasan selanjutnya, mari kita asumsikan bahwa kita sedang melakukan manipulasi di tumpukan maksimal.*

Deklarasi Tumpukan

```
struct Heap {
    int *array;
    int count;          // Number of elements in Heap
    int capacity;       // Size of the heap
    int heap_type;      // Min Heap or Max Heap
};
```

Membuat Tumpukan

```
struct Heap * CreateHeap(int capacity, int heap_type) {
    struct Heap * h = (struct Heap *)malloc(sizeof(struct Heap));
    if(h == NULL) {
          printf("Memory Error");
           return;
    }
```

Kompleksitas Waktu: O(1).

Induk dari sebuah Node

Untuk simpul di lokasi ke-i, induknya ada di lokasi. Pada contoh sebelumnya, elemen 6 berada di lokasi kedua dan induknya berada di lokasi ke-0.

```
int Parent (struct Heap * h, int i) {
    if(i <= 0 || i >= h→count)
        return -1;
    return i-1/2;
}
```

```
    h→heap_type = heap_type;
    h→count = 0;
    h→capacity = capacity;
    h→array = (int *) malloc(sizeof(int) * h→capacity);
    if(h→array == NULL) {
        printf("Memory Error");
        return;
    }
    return h;
}
```

Kompleksitas Waktu: O(1).

Anak-anak dari Node

Mirip dengan pembahasan di atas, untuk sebuah simpul di lokasi ke-i, anak-anaknya berada di lokasi 2 * i + 1 dan 2 * i + 2. Misalnya, pada pohon di atas elemen 6 berada di lokasi kedua dan anak-anaknya 2 dan 5 berada di lokasi 5 (2 * i + 1 = 2 * 2 + 1) dan 6(2 * i + 2 = 2 * 2)

```
int LeftChild(struct Heap *h, int i) {
    int left = 2 * i + 1;
    if(left >= h→count)
        return -1;
    return left;
}
Time Complexity: O(1).
```

```
int RightChild(struct Heap *h, int i) {
    int right = 2 * i + 2;
    if(right >= h→count)
        return -1;
    return right;
}
Time Complexity: O(1).
```

Mendapatkan Elemen Maksimum

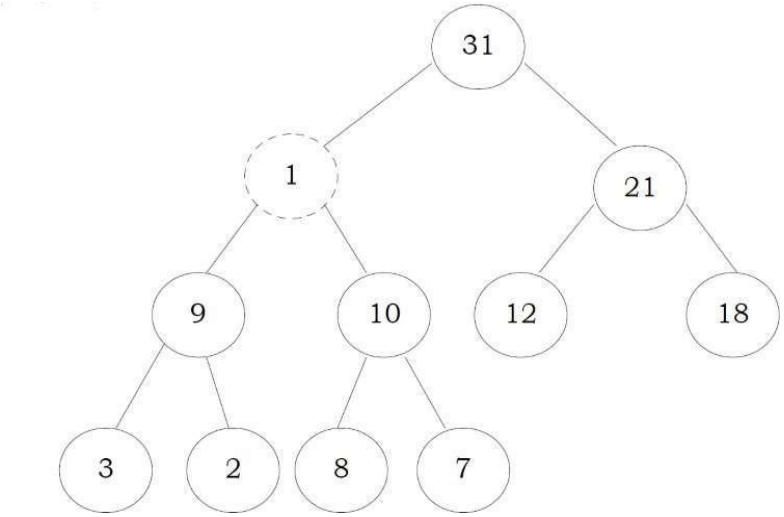
Karena elemen maksimum di max heap selalu di root, maka akan disimpan di h→array[O].

```
int GetMaximum(Heap * h) {
    if(h→count == 0)
        return -1;
    return h→array[0];
}
```

Kompleksitas Waktu: O(1).

Menimbun Elemen

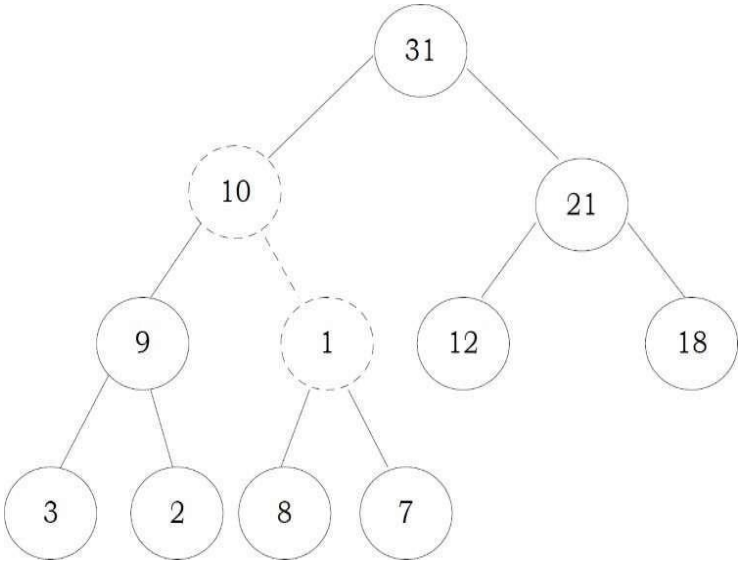
Setelah memasukkan elemen ke dalam heap, mungkin tidak memenuhi properti heap. Dalam hal ini kita perlu menyesuaikan lokasi heap untuk membuatnya heap lagi. Proses ini disebut heapifying. Dalam max-heap, untuk menimbun sebuah elemen, kita harus menemukan maksimum anak-anaknya dan menukarnya dengan elemen saat ini dan melanjutkan proses ini sampai properti heap terpenuhi di setiap node.



Gambar 7.7 *heapfying*

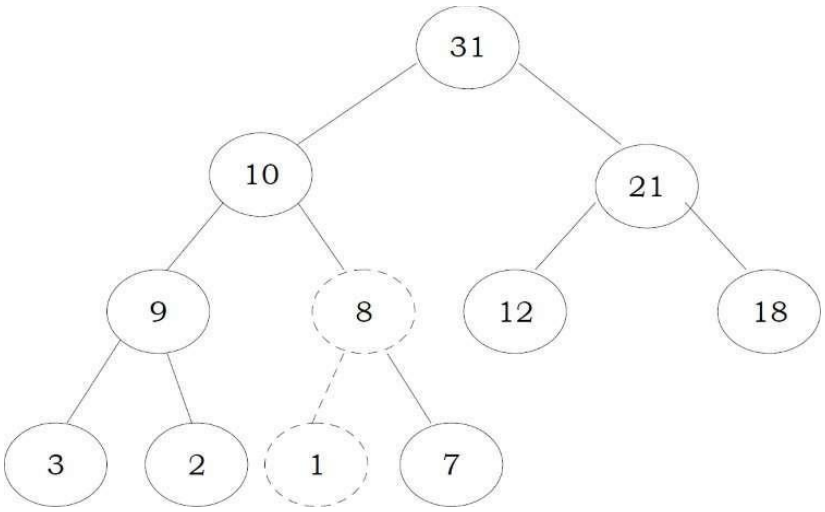
Pengamatan: Salah satu sifat penting dari heap adalah, jika suatu elemen tidak memenuhi sifat heap, maka semua elemen dari elemen tersebut hingga akarnya akan memiliki masalah yang sama. Pada contoh di bawah, elemen 1 tidak memenuhi properti heap dan induknya 31 juga mengalami masalah. Demikian pula, jika kita menumpuk sebuah elemen, maka semua elemen dari elemen tersebut ke root juga akan memenuhi properti heap secara otomatis. Mari kita pergi melalui sebuah contoh. Pada heap di atas, elemen 1 tidak memenuhi properti heap. Mari kita coba menumpuk elemen ini.

Untuk menimbun 1, temukan maksimum anak-anaknya dan tukar dengan itu.



Gambar 7.8 mencari maksimum anak-anaknya

Kita perlu melanjutkan proses ini sampai elemen memenuhi 286roperty heap. Sekarang, tukar 1 dengan 8.



Gambar 7.9 menukar 1 dengan 8

Sekarang pohon memenuhi properti heap. Dalam proses heapifying di atas, karena kita bergerak dari atas ke bawah, proses ini kadang-kadang disebut percolate down. Demikian pula, jika kita mulai menumpuk dari simpul lain ke root, kita dapat proses itu meresap ke atas sebagai gerakan dari bawah ke atas.

```
//Heapifying the element at location i.
void PercolateDown(struct Heap *h, int i) {
    int l, r, max, temp;
    l = LeftChild(h, i);
    r = RightChild(h, i);
    if(l != -1 && h→array[l] > h→array[i])
        max = l;
    else
        max = i;
    if(r != -1&& h→array[r] > h→array[max])
        max = r;
    if(max != i) {
        //Swap h→array[i] and  h→array[max];
        temp = h→array[i];
        h→array[i] = h→array[max];
        h→array[max] = temp;
    }
    PercolateDown(h, max);
}
```

Kompleksitas Waktu: O(logn). Heap adalah pohon biner lengkap dan dalam kasus terburuk kita mulai dari akar dan turun ke daun. Ini sama dengan tinggi pohon biner lengkap. Kompleksitas Ruang: O(1).

Menghapus Elemen

Untuk menghapus elemen dari heap, kita hanya perlu menghapus elemen dari root. Ini adalah satu-satunya operasi (elemen maksimum) yang didukung oleh tumpukan standar.

Setelah menghapus elemen root, salin elemen terakhir dari heap (pohon) dan hapus elemen terakhir itu.

Setelah mengganti elemen terakhir, pohon mungkin tidak memenuhi properti heap. Untuk membuatnya menumpuk lagi, panggil fungsi PercolateDown.

- Salin elemen pertama ke dalam beberapa variabel
- Salin elemen terakhir ke lokasi elemen pertama
- Meresap ke bawah elemen pertama

```
int DeleteMax(struct Heap *h) {
    int data;
    if(h→count == 0)
        return -1;
    data = h→array[0];
    h→array[0] = h→array[h→count-1];
    h→count--; //reducing the heap size
    PercolateDown(h, 0);
    return data;
}
```

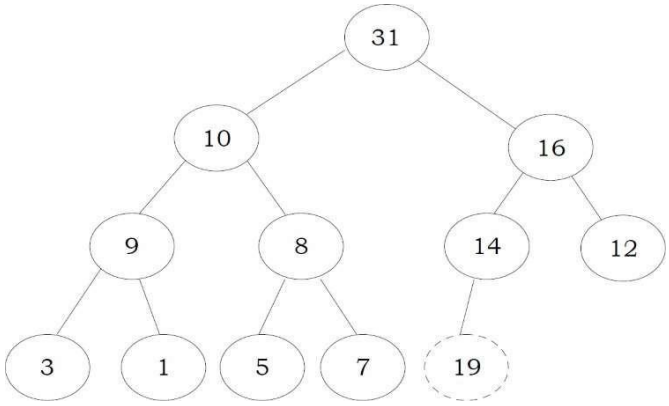
Catatan: *Menghapus elemen menggunakan PercolateDown, dan menyisipkan elemen menggunakan PercolateUp. Kompleksitas Waktu: sama dengan fungsi Heapify dan itu adalah O(logn).*

Memasukkan Elemen

Penyisipan elemen mirip dengan proses heapify dan penghapusan.

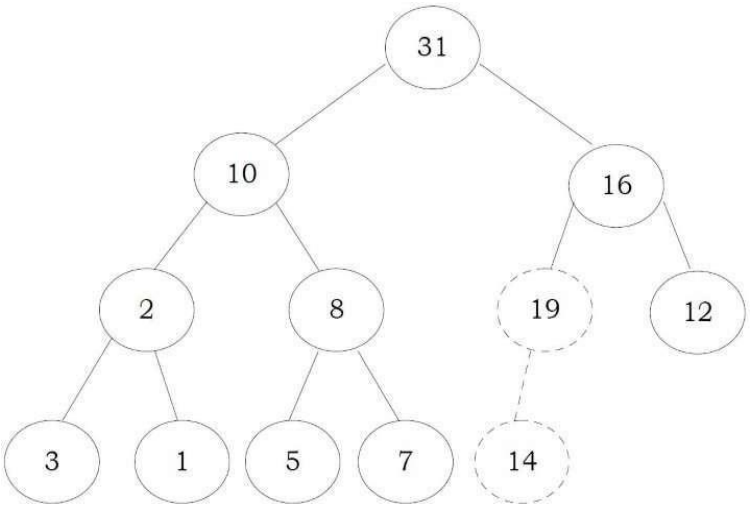
- Tambah ukuran tumpukan
- Simpan elemen baru di akhir heap (pohon)
- Heapify elemen dari bawah ke atas (root)

Sebelum membahas kode, mari kita lihat sebuah contoh. Kita telah memasukkan elemen 19 di akhir heap dan ini tidak memenuhi properti heap.



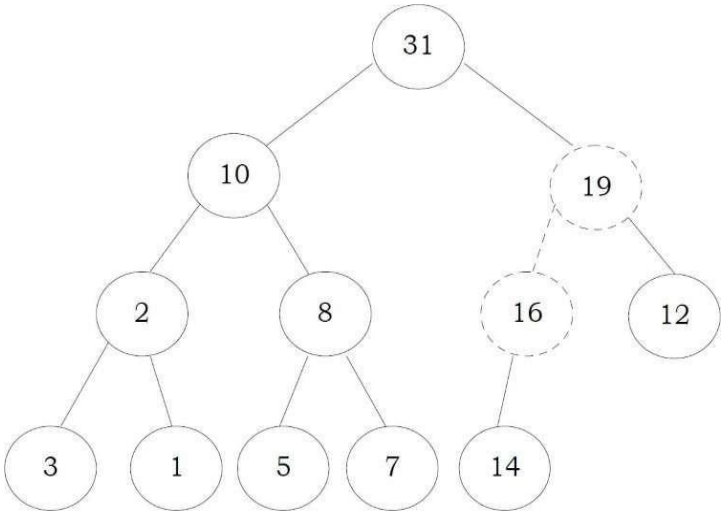
Gambar 7.10 ketika sudah dimasukkan elemen 19

Untuk menumpuk elemen ini (19), kita perlu membandingkannya dengan induknya dan menyesuaikannya. Menukar 19 dan 14 memberi:



Gambar 7.11 membandingkan dan menyesuaikan

Sekali lagi, tukar 19 dan l6:



Gambar 7.12 menukar 19 dan 16

Sekarang pohon memenuhi properti heap. Karena Kita mengikuti pendekatan bottom-up, Kita terkadang menyebut proses ini meresap ke atas.

```
int Insert(struct Heap *h, int data) {
    int i;
    if(h→count == h→capacity)
        ResizeHeap(h);
    h→count++;                  //increasing the heap size to hold this new item
    i = h→count-1;
    while(i>=0 && data > h→array[(i-1)/2]) {
        h→array[i] = h→array[(i-1)/2];
        i = i-1/2;
    }
    h→array[i] = data;
}
void ResizeHeap(struct Heap * h) {
    int *array_old = h→array;
    h→array = (int *) malloc(sizeof(int) * h→capacity * 2);
    if(h→array == NULL) {
        printf("Memory Error");
        return;
    }
    for (int i = 0; i < h→capacity; i ++)
        h→array[i] = array_old[i];
    h→capacity *= 2;
    free(array_old);
}
```

Kompleksitas Waktu: O(logn). Penjelasannya sama dengan fungsi Heapify.

Menghancurkan Tumpukan

```
void DestroyHeap (struct Heap *h) {
    if(h == NULL)
        return;
    free(h→array);
    free(h);
    h = NULL;
}
```

Menumpuk Array

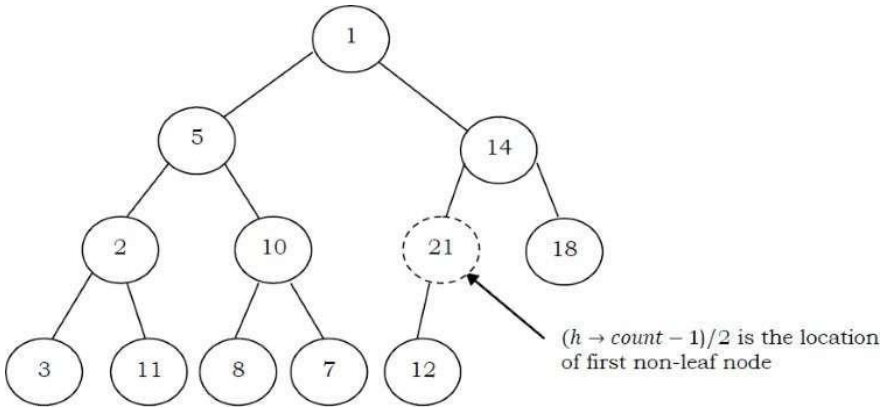
Salah satu pendekatan sederhana untuk membangun heap adalah, ambil n item input dan letakkan di heap kosong. Ini dapat dilakukan dengan n sisipan berturut-turut dan mengambil O(nlogn) dalam kasus terburuk. Hal ini disebabkan oleh fakta bahwa setiap operasi penyisipan membutuhkan O(logn).

Untuk menyelesaikan diskusi kita tentang tumpukan biner, kita akan melihat metode untuk membangun seluruh tumpukan dari daftar kunci. Cara pertama yang mungkin Anda pikirkan mungkin seperti berikut ini. Diberikan daftar kunci, Anda dapat dengan mudah membuat tumpukan dengan memasukkan setiap kunci satu per satu. Karena Anda memulai dengan daftar satu item, daftar tersebut diurutkan dan Anda dapat menggunakan pencarian biner untuk menemukan posisi yang tepat untuk memasukkan kunci berikutnya dengan biaya sekitar O(logn) operasi.

Namun, ingat bahwa menyisipkan item di tengah daftar mungkin memerlukan operasi O(n) untuk menggeser sisa daftar untuk memberi ruang bagi kunci baru. Oleh karena itu, untuk memasukkan n kunci ke dalam heap akan membutuhkan total operasi O(nlogn). Namun, jika

kita mulai dengan seluruh daftar maka kita dapat membangun seluruh tumpukan dalam operasi O(n).

Pengamatan: Node daun selalu memenuhi properti heap dan tidak perlu merawatnya. Elemen daun selalu di akhir dan untuk menumpuk array yang diberikan itu sudah cukup jika kita menumpuk node non-daun. Sekarang mari kita berkonsentrasi untuk menemukan simpul non-daun pertama. Elemen terakhir dari heap berada di lokasi h → count – 1, dan untuk menemukan simpul non-daun pertama cukup dengan menemukan induk dari elemen terakhir.



Gambar 7.13 menemukan simpul non-daun

```
void BuildHeap(struct Heap *h, int A[], int n) {
    if(h == NULL)
        return;
    while (n > h→capacity)
        ResizeHeap(h);
    for (int i = 0; i < n; i ++)
        h→array[i] = A[i];
    h→count = n;
    for (int i = (n-1)/2; i >=0; i --)
        PercolateDown(h, i);
}
```

Kompleksitas Waktu: Batas waktu linier tumpukan bangunan dapat ditunjukkan dengan menghitung jumlah ketinggian semua simpul. Untuk pohon biner lengkap dengan tinggi h yang berisi n = 2h+1- 1 node, jumlah dari tinggi node adalah n – h - 1 = n – logn – 1 (untuk pembuktian lihat Bagian Soal). Artinya, membangun operasi heap dapat dilakukan dalam waktu linier (O(n)) dengan menerapkan fungsi PercolateDown ke node dalam urutan level terbalik.

7.7 HEAPSORT

Salah satu aplikasi utama dari heap ADT adalah pengurutan (heap sort). Algoritma pengurutan tumpukan menyisipkan semua elemen (dari larik yang tidak diurutkan) ke dalam tumpukan, lalu menghapusnya dari akar tumpukan hingga tumpukan kosong. Perhatikan bahwa pengurutan tumpukan dapat dilakukan di tempat dengan larik yang akan diurutkan. Alih-alih menghapus elemen, tukarkan elemen pertama (maksimum) dengan elemen terakhir

dan kurangi ukuran tumpukan (ukuran array). Kemudian, Kita menumpuk elemen pertama. Lanjutkan proses ini sampai jumlah elemen yang tersisa adalah satu.

```
void Heapsort(int A[], in n) {
    struct Heap *h = CreateHeap(n);
    int old_size,  i, temp;
    BuildHeap(h, A, n);
    old_size = h→count;
    for(i = n-1; i > 0; i--) {
        //h→array [0] is the largest element
        temp = h→array[0];
        h→array[0] = h→array[h→count-1];
        h→array[0] = temp;
        h→count--;
        PercolateDown(h, 0);
    }
    h→count = old_size;
}
```

Kompleksitas waktu: Saat Kita menghapus elemen dari heap, nilainya menjadi terurut (karena elemen maksimum selalu hanya root). Karena kompleksitas waktu dari algoritma penyisipan dan algoritma penghapusan adalah O(logn) (di mana n adalah jumlah item dalam heap), kompleksitas waktu dari algoritma heap sort adalah O(nlogn).

7.8 ANTRIAN PRIORITAS [HEAPS]: MASALAH & SOLUSI

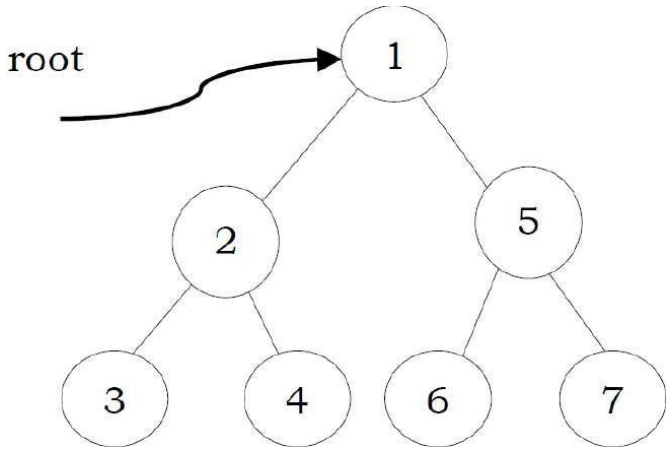
Soal-1 Berapa jumlah minimum dan maksimum elemen dalam tumpukan dengan ketinggian h?

Solusi: Karena heap adalah pohon biner lengkap (semua level berisi node penuh kecuali mungkin level terendah), ia memiliki paling banyak 2h+1 – 1 elemen (jika lengkap). Ini karena, untuk mendapatkan node maksimum, kita perlu mengisi semua level h sepenuhnya dan jumlah maksimum node tidak lain adalah jumlah semua node di semua level h.

Untuk mendapatkan node minimum, kita harus mengisi level h – 1 sepenuhnya dan level terakhir hanya dengan satu elemen. Akibatnya, jumlah minimum node tidak lain adalah jumlah semua node dari h – 1 level ditambah 1 (untuk level terakhir) dan Kita mendapatkan elemen 2h – 1 + 1 = 2h (jika level terendah hanya memiliki 1 elemen dan semua level lainnya selesai).

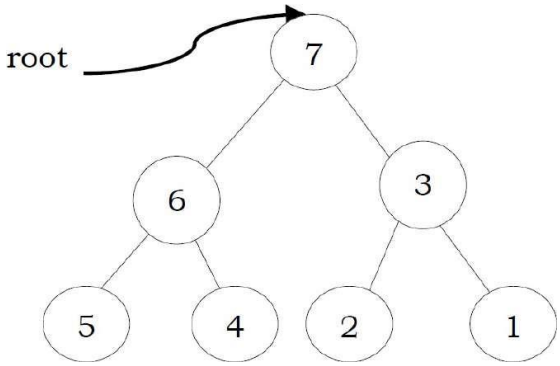
Soal-2 Apakah ada min-heap dengan tujuh elemen berbeda sehingga traversal preordernya memberikan elemen dalam urutan terurut?

Solusi: Ya. Untuk pohon di bawah ini, traversal preorder menghasilkan urutan menaik.



Soal-3 Apakah ada max-heap dengan tujuh elemen berbeda sehingga traversal preordernya memberikan elemen dalam urutan terurut?

Solusi: Ya. Untuk pohon di bawah ini, traversal preorder menghasilkan urutan menurun.

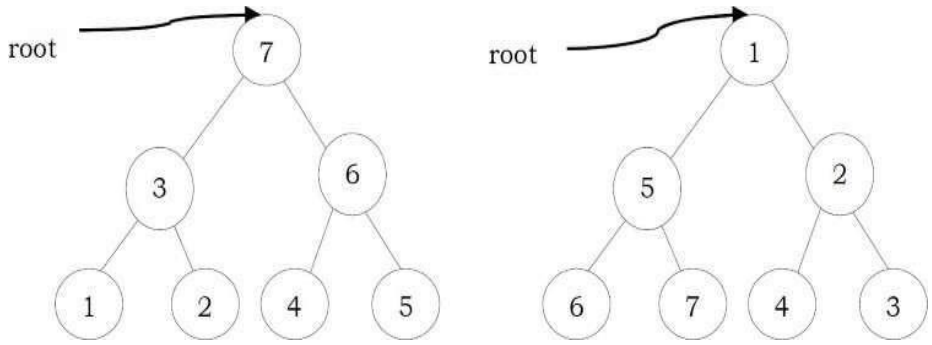


Soal-4 Apakah ada min-heap/max-heap dengan tujuh elemen berbeda sehingga traversal inordernya memberikan elemen dalam urutan terurut?

Solusi:Tidak. Karena heap harus berupa min-heap atau max-heap, root akan menampung elemen terkecil atau terbesar. Sebuah traversal inorder akan mengunjungi akar pohon sebagai langkah kedua, yang bukan tempat yang tepat jika akar pohon mengandung elemen terkecil atau terbesar.

Soal-5 Apakah ada min-heap/max-heap dengan tujuh elemen berbeda sehingga traversal postorder memberikan elemen dalam urutan terurut?

Solusi:



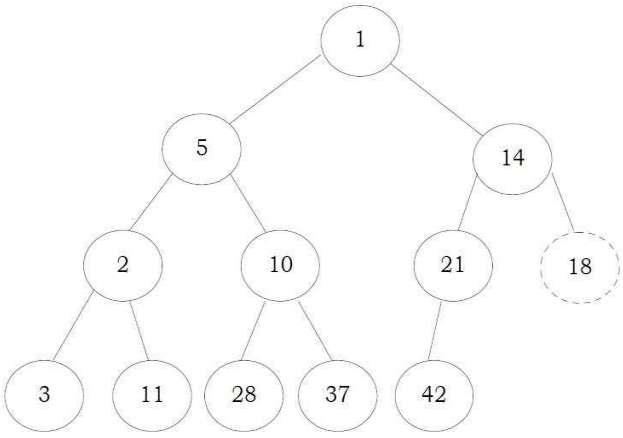
Ya, jika pohon adalah max-heap dan kita ingin urutan menurun (kiri bawah), atau jika pohon adalah min-heap dan kita ingin urutan naik (kanan bawah).

Soal-6 Tunjukkan bahwa tinggi tumpukan dengan n elemen adalah logn?

Solusi: Heap adalah pohon biner lengkap. Semua level, kecuali yang terendah, benar-benar penuh. Tumpukan memiliki setidaknya 2h elemen dan paling banyak elemen 2h n 2h+1 – 1. Ini menyiratkan, h logn h+ 1. Karena h adalah bilangan bulat, h = logn.

Soal-7 Diberikan min-heap, berikan algoritma untuk menemukan elemen maksimum.

Solusi: Untuk tumpukan minimum yang diberikan, elemen maksimum akan selalu berada di daun saja. Sekarang, pertanyaan selanjutnya adalah bagaimana menemukan simpul daun di pohon.



Jika kita amati dengan cermat, simpul berikutnya dari induk elemen terakhir adalah simpul daun pertama. Karena elemen terakhir selalu berada di h → count – lokasi ke-1, simpul berikutnya dari induknya (induk di lokasi $\frac{h \rightarrow count - 1}{2}$ dapat dihitung sebagai:

$$\frac{h \rightarrow count - 1}{2} + 1 \approx \frac{h \rightarrow count + 1}{2}$$

Sekarang, satu-satunya langkah yang tersisa adalah memindai simpul daun dan menemukan maksimum di antara mereka.

```
int FindMaxInMinHeap(struct Heap *h) {
    int Max = -1;
    for(int i = (h→count+1)/2; i < h→count; i++)
            if(h→array[i] > Max)
                Max = h→array[i];
}
```

Kompleksitas Waktu: $O(\frac{n}{2}) \approx O(n)$.

Soal-8 Berikan algoritma untuk menghapus elemen arbitrer dari min heap.

Solusi: Untuk menghapus sebuah elemen, pertama-tama kita perlu mencari sebuah elemen. Mari kita asumsikan bahwa kita menggunakan traversal urutan level untuk menemukan elemen. Setelah menemukan elemen, kita perlu mengikuti proses DeleteMin.

Kompleksitas Waktu = Waktu untuk menemukan elemen + Waktu untuk menghapus elemen
= O(n) + O (logn) ≈ O(n). //Waktu untuk pencarian didominasi.

Soal-9 Berikan algoritma untuk menghapus elemen terindeks ke-i dalam tumpukan-min yang diberikan.

Solusi:

```
int Delete(struct Heap *h, int i) {
    int key;
     if(n < i) {
         printf("Wrong position");
         return;
     }
     key = h→array[i];
     h→array[i]= h→array[h→count-1];
     h→count--;
     PercolateDown(h, i);
     return key;
}
```

Kompleksitas Waktu = O(logn).

Soal-10 Buktikan bahwa, untuk pohon biner lengkap dengan tinggi h, jumlah tinggi semua simpul adalah O(n – h).

Solusi: Sebuah pohon biner lengkap memiliki 2i node pada level (.Juga, sebuah node pada level i memiliki kedalaman i dan tinggi h – i. Mari kita asumsikan bahwa S menunjukkan jumlah ketinggian semua node ini dan S dapat dihitung sebagai :

$$S = \sum_{i=0}^{h} 2^i (h - i)$$

$$S = h + 2(h-1) + 4(h-2) + \cdots + 2^{h-1}(1)$$

Mengalikan dengan 2 di kedua sisi menghasilkan: $2S = 2h + 4(h – 1) + 8(h – 2) + …+ 2^h – 1(1)$
Sekarang, kurangi S dari 2S: $2S – S = – h + 2 + 4 + … + 2^h \Rightarrow S = (2^{h+1} – 1) – (h – 1)$
Namun, kita telah mengetahui bahwa jumlah total node n dalam pohon biner lengkap dengan tinggi h adalah $n = 2^{h+1} – 1$. Ini memberi kita: $h = \log(n + 1)$.
Akhirnya, mengganti $2^{h+1} – 1$ dengan n, menghasilkan: $S = n – (h – 1) = O(n – \log n) = O(n - h)$.

Soal-11 Berikan algoritma untuk menemukan semua elemen yang kurang dari beberapa nilai k dalam tumpukan biner.

Solusi: Mulai dari akar heap. Jika nilai root lebih kecil dari k maka cetak nilainya dan panggil secara rekursif sekali untuk anak kirinya dan sekali untuk anak kanannya. Jika nilai sebuah simpul lebih besar atau sama dengan k maka fungsi berhenti tanpa mencetak nilai tersebut.

Kompleksitas algoritma ini adalah O(n), di mana n adalah jumlah total node dalam heap. Batas ini terjadi dalam kasus terburuk, di mana nilai setiap node di heap akan lebih kecil dari k, sehingga fungsi harus memanggil setiap node dari heap.

Soal-12 Berikan algoritma untuk menggabungkan dua max-heap biner. Mari kita asumsikan bahwa ukuran tumpukan pertama adalah m + n dan ukuran tumpukan kedua adalah n.

Solusi Salah satu cara sederhana untuk menyelesaikan masalah ini adalah:

- Asumsikan bahwa elemen larik pertama (dengan ukuran m + n) berada di awal. Itu berarti, m sel pertama terisi dan n sel tersisa kosong.
- Tanpa mengubah heap pertama, cukup tambahkan heap kedua dan heapify array.
- Karena jumlah total elemen dalam array baru adalah m + n, setiap operasi heapify membutuhkan O(log(m + n)).

Kompleksitas dari algoritma ini adalah : O((m + n)log(m + n)).

Soal-13 Bisakah kita meningkatkan kompleksitas Soal-12?

Solusi: Alih-alih menumpuk semua elemen larik m + n, kita dapat menggunakan teknik "membangun tumpukan dengan larik elemen (menumpukan larik)". Kita bisa mulai dengan node non-daun dan menumpuknya. Algoritma dapat diberikan sebagai:

- Asumsikan bahwa elemen larik pertama (dengan ukuran m + n) berada di awal. Artinya, m sel pertama terisi dan n sel sisanya kosong.
- Tanpa mengubah tumpukan pertama, cukup tambahkan tumpukan kedua.

- Sekarang, temukan simpul non-daun pertama dan mulailah menumpuk dari elemen itu.

Di bagian teori, kita telah melihat bahwa membangun tumpukan dengan n elemen membutuhkan kompleksitas O(n). Kompleksitas penggabungan dengan teknik ini adalah: O(m + n).

Soal-14 Apakah ada algoritma yang efisien untuk menggabungkan 2 max-heap (disimpan sebagai array)? Asumsikan kedua array memiliki n elemen.

Solusi: Solusi alternatif untuk masalah ini tergantung pada jenis tumpukan itu. Jika itu adalah tumpukan standar di mana setiap simpul memiliki hingga dua anak dan yang diisi sehingga daun berada pada maksimum dua baris yang berbeda, kita tidak bisa lebih baik dari O(n) untuk penggabungan.

Ada algoritma O(logm × logn) untuk menggabungkan dua tumpukan biner dengan ukuran m dan n. Untuk m = n, algoritma ini membutuhkan kompleksitas waktu O(log2n). Kita akan melewatkannya karena kesulitan dan cakupannya.

Untuk kinerja penggabungan yang lebih baik, kita dapat menggunakan varian lain dari tumpukan biner seperti Fibonacci- Heap yang dapat digabungkan dalam O(1) rata-rata (diamortisasi).

Soal-15 Berikan algoritma untuk menemukan elemen terkecil ke-k di min-heap.

Solusi: Salah satu solusi sederhana untuk masalah ini adalah: lakukan penghapusan k kali dari min-heap.

```
return PQ.Min();
        //Just delete first k-1 elements and return the k-th element.
         for(int i=0;i<k-1;i++)
             DeleteMin(h);
         return DeleteMin(h);
    }
```

Kompleksitas Waktu: O(klogn). Karena kita melakukan operasi penghapusan sebanyak k kali dan setiap penghapusan membutuhkan O(logn).

Soal-16 Untuk Soal-15, dapatkah kita meningkatkan kompleksitas waktu?

Solusi: Asumsikan bahwa min-heap asli disebut HOrig dan min-heap tambahan bernama HAux. Awalnya, elemen di bagian atas HOrig, yang minimum, dimasukkan ke dalam Haux. Di sini kita tidak melakukan operasi DeleteMin dengan HOrig.

```
Heap HOrig;
Heap HAux;
int FindKthLargestEle( int k ) {
    int heapElement;//Assuming heap data is of integers
    int count=1;
    HAux.Insert(HOrig.Min());
    while( true ) {
            //return the minimum element and delete it from the HA heap
            heapElement = HAux.DeleteMin();
            if(++count == k ) {
                    return heapElement;
            }
            else {    //insert the left and right children in HO into the HA
                    HAux.Insert(heapElement.LeftChild());
                    HAux.Insert(heapElement.RightChild());
            }
    }
}
```

Setiap iterasi while-loop memberikan elemen terkecil ke-k dan kita membutuhkan k loop untuk mendapatkan elemen terkecil ke-k. Karena ukuran tumpukan bantu selalu kurang dari k, setiap iterasi loop sementara ukuran tumpukan tambahan bertambah satu, dan tumpukan asli HOrig tidak beroperasi selama pencarian, waktu berjalan adalah O(klogk).

Catatan: Algoritma di atas berguna jika nilai k terlalu kecil dibandingkan dengan n. Jika nilai k kira-kira sama dengan n, maka kita cukup mengurutkan array (misalnya, menggunakan pengurutan couting atau algoritma pengurutan linier lainnya) dan mengembalikan elemen terkecil ke-k dari larik yang diurutkan. Ini memberikan solusi O(n).

Soal-17 Temukan k elemen maks dari tumpukan maks.

Solusi: Salah satu solusi sederhana untuk masalah ini adalah: membangun max-heap dan melakukan penghapusan k kali.

T(n) = DeleteMin dari heap k kali = Θ((klogn).

Soal-18 Untuk Soal-17, apakah ada solusi alternatif?

Solusi: Kita dapat menggunakan solusi Soal-16. Pada akhirnya, tumpukan bantu berisi k- elemen terbesar. Tanpa menghapus elemen, kita harus terus menambahkan elemen ke Haux.

Soal-19 Bagaimana kita mengimplementasikan stack menggunakan heap?

Solusi: Untuk mengimplementasikan stack menggunakan prioritas antrian PQ (menggunakan min heap), mari kita asumsikan bahwa kita menggunakan satu variabel integer tambahan c. Juga, asumsikan bahwa c diinisialisasi sama dengan nilai yang diketahui (misalnya, 0). Implementasi stack ADT diberikan di

bawah ini. Di sini c digunakan sebagai prioritas saat memasukkan/menghapus elemen dari PQ.

```
void Push(int element) {
    PQ.Insert(c, element);
    c--;
}
int Pop() {
    return PQ.DeleteMin();
}
int Top() {
    return PQ.Min();
}
int Size() {
    return PQ.Size();
}
int IsEmpty() {
    return PQ.IsEmpty();
}
```

Kita juga dapat menambah c kembali saat muncul.

Pengamatan: Kita bisa menggunakan negatif dari waktu sistem saat ini alih-alih c (untuk menghindari luapan). Implementasi berdasarkan ini dapat diberikan sebagai:

```
void Push(int element) {
    PQ.insert(-gettime(),element);
}
```

Soal-20 Bagaimana kita mengimplementasikan Antrian menggunakan heap?

Solusi: Untuk mengimplementasikan antrian menggunakan prioritas antrian PQ (menggunakan min heap), seperti simulasi tumpukan, mari kita asumsikan bahwa kita menggunakan satu variabel integer tambahan, c. Juga, asumsikan bahwa c diinisialisasi sama dengan nilai yang diketahui (misalnya, 0). Implementasi antrian ADT diberikan di bawah ini. Di sini c digunakan sebagai prioritas saat memasukkan/menghapus elemen dari PQ.

```
void Push(int element) {
    PQ.Insert(c, element);
    c++;
}
int Pop() {
    return PQ.DeleteMin();
}
int Top() {
    return PQ.Min();
}
int Size() {
    return PQ.Size();
}
int IsEmpty() {
    return PQ.IsEmpty();
}
```

Catatan: Kita juga dapat mengurangi c saat muncul.

Pengamatan: Kita hanya dapat menggunakan negatif dari waktu sistem saat ini alih-alih c (untuk menghindari luapan). Implementasi berdasarkan ini dapat diberikan sebagai:

```
void Push(int element) {
    PQ.insert(gettime(),element);
}
```

Catatan: Satu-satunya perubahan adalah kita perlu mengambil nilai c positif, bukan negatif.

Soal-21 Mengingat file besar yang berisi miliaran angka, bagaimana Anda dapat menemukan 10 angka maksimum dari file itu?

Solusi: Selalu ingat bahwa ketika Anda perlu menemukan maksimal n elemen, struktur data terbaik untuk digunakan adalah antrian prioritas.

Salah satu solusi untuk masalah ini adalah dengan membagi data dalam kumpulan 1000 elemen (katakanlah 1000) dan buat tumpukannya, lalu ambil 10 elemen dari setiap tumpukan satu per satu. Terakhir, urutkan semua set 10 elemen dan ambil 10 teratas di antara mereka. Tetapi masalah dalam pendekatan ini adalah di mana menyimpan 10 elemen dari setiap heap. Itu mungkin membutuhkan banyak memori karena kita memiliki miliaran angka.

Menggunakan kembali 10 elemen teratas (dari tumpukan sebelumnya) di elemen berikutnya dapat menyelesaikan masalah ini. Itu berarti ambil blok pertama dari 1000 elemen dan blok berikutnya masing-masing 990 elemen. Awalnya, Heapsort set pertama dari 1000 angka, ambil maksimal 10 elemen, dan campur dengan 990 elemen dari set ke-2. Sekali lagi, Heapsort 1000 angka ini (10 dari set pertama dan 990 dari set ke-2), ambil 10 elemen maksimal, dan campurkan dengan 990 elemen dari set ke-3. Ulangi hingga set terakhir dari 990 (atau kurang) elemen dan ambil maksimal 10 elemen dari tumpukan terakhir. 10 elemen ini akan menjadi jawaban Anda.

Kompleksitas Waktu: O(n) = n/1000 ×(kompleksitas Heapsort 1000 elemen) Karena kompleksitas penyortiran tumpukan 1000 elemen akan menjadi konstan sehingga O(n) = n yaitu kompleksitas linier.

Soal-22 Gabungkan k daftar terurut dengan total n elemen: Kita diberikan k daftar terurut dengan total n masukan di semua daftar. Berikan algoritma untuk menggabungkannya menjadi satu daftar tunggal yang diurutkan.

Solusi: Karena ada k daftar dengan ukuran yang sama dengan total n elemen, ukuran setiap daftar adalah Salah satu cara sederhana untuk memecahkan masalah ini adalah:

- Ambil daftar pertama dan gabungkan dengan daftar kedua. Karena ukuran setiap daftar adalah $\frac{\mathbf{n}}{\mathbf{k}}$, langkah ini menghasilkan daftar yang diurutkan

dengan ukuran $\frac{2n}{k}$. Ini mirip dengan menggabungkan logika sortir. Kompleksitas waktu dari langkah ini adalah: $\frac{2n}{k}$. Ini karena kita perlu memindai semua elemen dari kedua daftar.

- Kemudian, gabungkan output daftar kedua dengan daftar ketiga. Akibatnya, langkah ini menghasilkan daftar yang diurutkan dengan ukuran $\frac{3n}{k}$. Kompleksitas waktu dari langkah ini adalah: $\frac{3n}{k}$. Ini karena kita perlu memindai semua elemen dari kedua daftar (satu dengan ukuran $\frac{2n}{k}$ dan yang lainnya dengan ukuran $\frac{n}{k}$).
- Lanjutkan proses ini sampai semua daftar digabung menjadi satu daftar.

Kompleksitas waktu total:

$$= \frac{2n}{k} + \frac{3n}{k} + \frac{4n}{k} + \cdots . \frac{kn}{k} = \sum_{i=2}^{n} \frac{in}{k} = \frac{n}{k} \sum_{i=2}^{n} i \approx \frac{n(k^2)}{k} \approx O(nk)$$

Kompleksitas Ruang: O(1).

Soal-23 Untuk Soal-22, dapatkah kita meningkatkan kompleksitas waktu?

Solusi:

1. Bagilah daftar menjadi pasangan dan gabungkan. Itu berarti, pertama-tama ambil dua daftar sekaligus dan gabungkan mereka sehingga total elemen yang diuraikan untuk semua daftar adalah O(n). Operasi ini memberikan k/2 daftar.
2. Ulangi langkah-1 hingga jumlah daftar menjadi satu.

Kompleksitas waktu: Langkah-1 mengeksekusi waktu logk dan setiap operasi mem-parsing semua n elemen di semua daftar untuk membuat k/2 daftar. Misalnya, jika kita memiliki 8 daftar, maka pass pertama akan membuat 4 daftar dengan menguraikan semua n elemen. Pass kedua akan membuat 2 daftar dengan mem-parsing n elemen lagi dan pass ketiga akan memberikan 1 daftar dengan mem-parsing n elemen lagi. Akibatnya kompleksitas waktu total adalah O(nlogn).

Kompleksitas Ruang: O(n).

Soal-24 Untuk Soal-23, dapatkah kita meningkatkan kompleksitas ruang?

Solusi: Mari kita gunakan tumpukan untuk mengurangi kompleksitas ruang.

1. Bangun max-heap dengan semua elemen pertama dari setiap daftar dalam O(k).
2. Di setiap langkah, ekstrak elemen maksimum dari heap dan tambahkan di akhir output.

3. Tambahkan elemen berikutnya dari daftar yang diekstraksi. Itu berarti kita perlu memilih elemen berikutnya dari daftar yang berisi elemen yang diekstraksi dari langkah sebelumnya.
4. Ulangi langkah-2 dan langkah-3 sampai semua elemen selesai dari semua daftar.

Kompleksitas Waktu = O(nlogk). Pada suatu waktu kita memiliki k elemen max-heap dan untuk semua n elemen kita harus membaca hanya heap dalam waktu logk, jadi total waktu = O(nlogk).
Kompleksitas Ruang: O(k) [untuk Max-heap].

Soal-25 Diberikan 2 larik A dan B masing-masing dengan n elemen. Berikan algoritma untuk menemukan n pasangan terbesar (A[i],B[j]).

Solusi:

Algoritma:

- Heapify A dan B. Langkah ini membutuhkan O(2n) O(n).
- Kemudian terus hapus elemen dari kedua heap. Setiap langkah membutuhkan O(2logn) ≈ O (logn).

Kompleksitas Waktu Total: O(nlogn).

Soal-26 **Min-Max heap:** Berikan algoritma yang mendukung min dan max dalam waktu O(1), masukkan, hapus min, dan hapus maks dalam waktu O(logn). Artinya, rancang struktur data yang mendukung operasi berikut:

Operasi	Kompleksitas
Init	O(*n*)
Insert	O(*logn*)
FindMin	O(1)
FindMax	O(1)
Delete Min	O(*logn*)
Delete Max	O(*lognri*)

Solusi Masalah ini dapat diselesaikan dengan menggunakan dua tumpukan. Katakanlah dua heap adalah: Minimum-Heap H_{min} dan Maximum-Heap H_{max}. Juga, asumsikan bahwa elemen di kedua array memiliki pointer bersama. Artinya, sebuah elemen di H_{min} akan memiliki pointer ke elemen yang sama di H_{max} dan elemen di H_{max} akan memiliki pointer ke elemen yang sama di H_{min}.

Init	Bangun H_{min} dalam O(n) dan H_{max} dalam O(n)
Insert(x)	Masukkan x ke H_{min} di O(logn). Masukkan x ke H_{max} di O(logn). Perbarui pointer di O(1)
FindMin()	Kembalikan root(H_{min}) di O(1)
FindMax	Kembalikan root(H_{max}) di O(1)
Delete Min	Hapus minimum dari H_{min} di O(logn). Hapus elemen yang sama dari H_{max} dengan menggunakan penunjuk bersama di O(logn)
DeleteMax	Hapus maksimum dari H_{max} di O(logn). Hapus elemen yang sama dari H_{min} dengan menggunakan penunjuk bersama di O(logn)

Soal-27 Penemuan median dinamis. Rancang struktur data heap yang mendukung pencarian median.

Solusi: Dalam himpunan n elemen, median adalah elemen tengah, sehingga jumlah elemen yang lebih kecil dari median sama dengan jumlah elemen yang lebih besar dari median. Jika n ganjil, kita dapat mencari median dengan mengurutkan himpunan dan mengambil elemen tengah. Jika n genap, median biasanya didefinisikan sebagai rata-rata dari dua elemen tengah. Algoritma ini bekerja bahkan ketika beberapa elemen dalam daftar sama. Misalnya, median multiset {1, 1, 2, 3, 5} adalah 2, dan median multiset {1, 1, 2, 3, 5, 8} adalah 2,5. "Median heaps" adalah varian dari heap yang memberikan akses ke elemen median. Tumpukan median dapat diimplementasikan menggunakan dua tumpukan, masing-masing berisi setengah elemen. Salah satunya adalah max-heap, berisi elemen terkecil; yang lainnya adalah min-heap, yang mengandung elemen terbesar. Ukuran max-heap mungkin sama dengan ukuran min-heap, jika jumlah elemennya genap. Dalam hal ini, median adalah rata-rata dari elemen maksimum tumpukan-maks dan elemen minimum tumpukan-min. Jika jumlah elemen ganjil, max-heap akan berisi satu elemen lebih banyak daripada min-heap. Median dalam hal ini hanyalah elemen maksimum dari max-heap.

Soal-28 Jumlah maksimum dalam jendela geser: Diberikan larik A[] dengan jendela geser berukuran w yang bergerak dari paling kiri larik ke paling kanan. Asumsikan bahwa kita hanya dapat melihat angka w di jendela. Setiap kali jendela geser bergerak ke kanan dengan satu posisi. Contoh: Array adalah [1 3 -1 -3 5 3 6 7], dan w adalah 3.

Posisi Jendela	**Maks**
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5

1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: Array panjang A[], dan lebar jendela w.
Output: Array B[], B[i] adalah nilai maksimum dari A[i] hingga A[i+w-1]

Persyaratan: Temukan cara optimal yang baik untuk mendapatkan B[i]

Solusi: Solusi brute force adalah, setiap kali jendela dipindahkan kita dapat mencari total w elemen di jendela.

Kompleksitas waktu: O(nw).

Soal-29 Untuk Soal-28, dapatkah kita mengurangi kerumitannya?

Solusi: Ya, kita bisa menggunakan struktur data heap. Ini mengurangi kompleksitas waktu menjadi O(nlogw). Operasi penyisipan membutuhkan waktu O(logw), di mana w adalah ukuran heap. Namun, mendapatkan nilai maksimal itu murah; itu hanya membutuhkan waktu yang konstan karena nilai maksimum selalu disimpan di root (head) heap. Saat jendela meluncur ke kanan, beberapa elemen di heap mungkin tidak valid lagi (rentang berada di luar jendela saat ini). Bagaimana kita harus menghapusnya? Kita perlu agak berhati-hati di sini. Karena Kita hanya menghapus elemen yang berada di luar jangkauan jendela, Kita juga perlu melacak indeks elemen.

Soal-30 Untuk Soal-28, dapatkah kita mengurangi kerumitan lebih lanjut?

Solusi: Ya, Antrian berujung ganda adalah struktur data yang sempurna untuk masalah ini. Mendukung penyisipan/penghapusan dari depan dan belakang. Caranya adalah dengan mencari cara sedemikian rupa sehingga elemen terbesar di jendela akan selalu muncul di depan antrian. Bagaimana Anda mempertahankan persyaratan ini saat Anda mendorong dan mengeluarkan elemen masuk dan keluar dari antrian?

Selain itu, Anda akan melihat bahwa ada beberapa elemen redundan dalam antrian yang seharusnya tidak Kita pertimbangkan. Misalnya, jika antrian saat ini memiliki elemen: [10 5 3], dan elemen baru di jendela memiliki elemen 11. Sekarang, kita dapat mengosongkan antrian tanpa mempertimbangkan elemen 10, 5, dan 3, dan hanya menyisipkan elemen 11 ke dalam antrian.

Biasanya, kebanyakan orang mencoba mempertahankan ukuran antrian sama dengan ukuran jendela. Cobalah untuk melepaskan diri dari pemikiran ini dan berpikir di luar kotak. Menghapus elemen yang berlebihan dan hanya menyimpan elemen yang perlu dipertimbangkan dalam antrian adalah kunci

untuk mencapai solusi O(n) yang efisien di bawah ini. Ini karena setiap elemen dalam daftar dimasukkan dan dihapus paling banyak satu kali. Oleh karena itu, jumlah total operasi insert + delete adalah 2n.

```
void MaxSlidingWindow(int A[], int n, int w, int B[]) {
    struct DoubleEndQueue *Q = CreateDoubleEndQueue();
    for (int i = 0; i < w; i++) {
            while (!IsEmptyQueue(Q) && A[i] >= A[QBack(Q)])
                    PopBack(Q);
            PushBack(Q, i);
    }
    for (int i = w; i < n; i++) {
            B[i-w] = A[QFront(Q)];
            while (!IsEmptyQueue(Q) && A[i] >= A[QBack(Q)])
                    PopBack(Q);
            while (!IsEmptyQueue(Q) && QFront(Q) <= i-w)
                    PopFront(Q);
            PushBack(Q, i);
    }
    B[n-w] = A[QFront(Q)];
}
```

Soal-31 Antrian prioritas adalah daftar item di mana setiap item telah dikaitkan dengan prioritas. Item ditarik dari antrian prioritas dalam urutan prioritasnya dimulai dengan item prioritas tertinggi terlebih dahulu. Jika item prioritas maksimum diperlukan, maka heap dibangun sedemikian rupa sehingga prioritas setiap node lebih besar dari prioritas anak-anaknya.

Rancang tumpukan seperti itu di mana item dengan prioritas tengah ditarik terlebih dahulu. Jika terdapat n item dalam heap, maka jumlah item dengan prioritas lebih kecil dari prioritas tengah adalah jika n ganjil, jika tidak 1.

Jelaskan bagaimana operasi penarikan dan penyisipan bekerja, menghitung kompleksitasnya, dan bagaimana struktur data dibangun.

Solusi: Kita dapat menggunakan satu min heap dan satu max heap sehingga akar dari min heap lebih besar dari akar dari tumpukan maks. Ukuran tumpukan minimum harus sama atau satu kurang dari ukuran tumpukan maksimum. Jadi elemen tengah selalu merupakan akar dari tumpukan maksimal.

Untuk operasi insert, jika item baru kurang dari akar max heap, maka masukkan ke max heap; lain masukkan ke dalam tumpukan min. Setelah operasi penarikan atau penyisipan, jika ukuran tumpukan tidak seperti yang ditentukan di atas, pindahkan elemen akar tumpukan maksimum ke tumpukan minimum atau sebaliknya.

Dengan implementasi ini, operasi insert dan withdraw akan dilakukan dalam waktu O(logn).

Soal-32 Mengingat dua tumpukan, bagaimana Anda menggabungkan (menyatukan) mereka?

Solusi: Binary heap mendukung berbagai operasi dengan cepat: Temukan-min, masukkan, penurunan-kunci. Jika kita memiliki dua min-heap, H1 dan H2, tidak ada cara yang efisien untuk menggabungkannya menjadi satu min-heap.

Untuk memecahkan masalah ini secara efisien, kita dapat menggunakan tumpukan yang dapat digabungkan. Tumpukan yang dapat digabungkan mendukung operasi penyatuan yang efisien. Ini adalah struktur data yang mendukung operasi berikut:

- Create-Heap(): membuat heap kosong
- Insert(H,X,K) : menyisipkan item x dengan kunci K ke dalam heap H
- Temukan-Min(H) : kembalikan item dengan kunci min
- Hapus-Min(H) : kembali dan hapus
- Union(H1, H2) : menggabungkan tumpukan H1 dan H2

Contoh tumpukan yang dapat digabungkan adalah:

- Tumpukan Binomial
- Tumpukan Fibonacci

Kedua tumpukan juga mendukung:

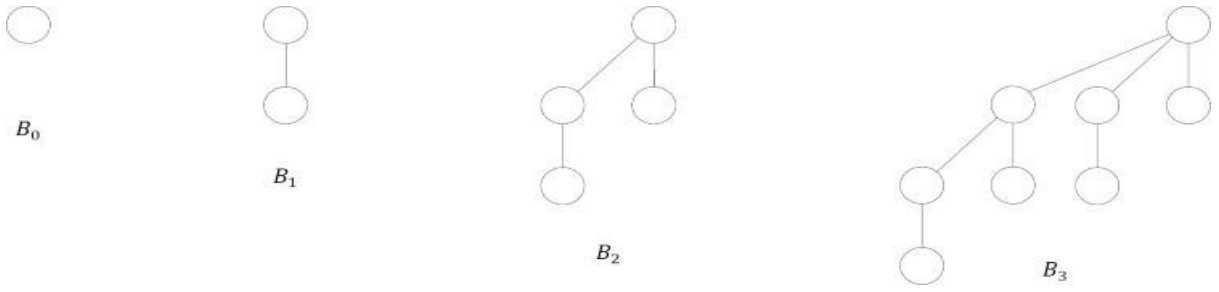
- Tombol Penurunan(H,X,K): menetapkan item Y dengan kunci yang lebih kecil K
- Hapus(H,X) : hapus item X

Tumpukan Binomial: Tidak seperti tumpukan biner yang terdiri dari satu pohon, tumpukan binomial terdiri dari sekumpulan kecil pohon komponen dan tidak perlu membangun kembali semuanya saat penyatuan dilakukan. Setiap pohon komponen dalam format khusus, yang disebut pohon binomial.

Pohon binomial orde k, dilambangkan dengan Bk, didefinisikan secara rekursif sebagai berikut:

- B0 adalah pohon dengan satu simpul
- Untuk k 1, Bk dibentuk dengan menggabungkan dua Bk -1, sehingga akar dari satu pohon menjadi anak paling kiri dari akar yang lain.

Contoh:



Fibonacci Heaps: Fibonacci heap adalah contoh lain dari mergeable heap. Ini tidak memiliki jaminan kasus terburuk yang baik untuk operasi apa pun (kecuali Sisipkan/Buat-Heap). Fibonacci Heaps memiliki biaya diamortisasi yang sangat baik untuk melakukan setiap operasi. Seperti heap binomial, fibonacci heap terdiri dari sekumpulan pohon komponen terurut min-heap.

Namun, tidak seperti tumpukan binomial, ia memiliki

- Tidak ada batasan jumlah pohon (hingga O(n)), dan
- Tidak ada batasan ketinggian pohon (hingga O(n))

Juga, Cari-Min, Hapus-Min, Union, Penurunan-Key, Hapus semua memiliki waktu berjalan O(n) kasus terburuk. Namun, dalam arti diamortisasi, setiap operasi berkinerja sangat cepat.

Operation	Binary Heap	Binomial Heap	Fibonacci Heap
Create-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Find-Min	$\Theta(1)$	$\Theta(logn)$	$\Theta(1)$
Delete-Min	$\Theta(logn)$	$\Theta(logn)$	$\Theta(logn)$
Insert	$\Theta(logn)$	$\Theta(logn)$	$\Theta(1)$
Delete	$\Theta(logn)$	$\Theta(logn)$	$\Theta(logn)$
Decrease-Key	$\Theta(logn)$	$\Theta(logn)$	$\Theta(1)$
Union	$\Theta(n)$	$\Theta(logn)$	$\Theta(1)$

Soal-33 Median dalam deret bilangan bulat tak hingga

Solusi: Median adalah angka tengah dalam daftar angka yang diurutkan (jika kita memiliki jumlah elemen ganjil). Jika kita memiliki jumlah elemen genap, median adalah rata-rata dari dua angka tengah dalam daftar angka yang diurutkan.

Kita dapat memecahkan masalah ini secara efisien dengan menggunakan 2 heap: Satu MaxHeap dan satu MinHeap.

1. MaxHeap berisi setengah terkecil dari bilangan bulat yang diterima
2. MinHeap berisi setengah terbesar dari bilangan bulat yang diterima

Bilangan bulat di MaxHeap selalu kurang dari atau sama dengan bilangan bulat di MinHeap. Juga, jumlah elemen di MaxHeap sama dengan atau 1 lebih banyak dari jumlah elemen di MinHeap.

Dalam aliran jika kita mendapatkan 2n elemen (pada titik waktu mana pun), MaxHeap dan MinHeap keduanya akan berisi jumlah elemen yang sama (dalam hal ini, n elemen di setiap heap). Jika tidak, jika kita telah menerima 2n+1 elemen, MaxHeap akan berisi n+1 dan MinHeap n.

Mari kita cari Median: Jika kita memiliki 2n + 1 elemen (ganjil), Median dari elemen yang diterima akan menjadi elemen terbesar di MaxHeap (tidak lain adalah akar dari MaxHeap). Jika tidak, Median elemen yang diterima akan menjadi rata-rata elemen terbesar di MaxHeap

(tidak lain adalah akar dari MaxHeap) dan elemen terkecil di MinHeap (tidak lain adalah akar dari MinHeap). Ini dapat dihitung dalam O(1).

Memasukkan elemen ke dalam heap dapat dilakukan di O(logn). Perhatikan bahwa, tumpukan apa pun yang berisi n + 1 elemen mungkin memerlukan satu operasi penghapusan (dan penyisipan ke tumpukan lain) juga.

Contoh:

Sisipkan 1: Sisipkan ke MaxHeap.

MaxHeap: {1}, MinHeap:{}

Sisipkan 9: Sisipkan ke MinHeap. Karena 9 lebih besar dari 1 dan MinHeap mempertahankan elemen maksimum.

MaxHeap: {1}, MinHeap:{9}

Sisipkan 2: Sisipkan MinHeap. Karena 2 kurang dari semua elemen MinHeap. MaxHeap: {1,2}, MinHeap:{9}

Sisipkan 0: Karena MaxHeap sudah memiliki lebih dari setengahnya; kita harus menjatuhkan elemen max dari MaxHeap dan memasukkannya ke MinHeap. Jadi, kita harus menghapus 2 dan memasukkan ke dalam MinHeap. Dengan itu menjadi:

MaxHeap: {1},

MinHeap:{2,9}

Sekarang, masukkan 0 ke MaxHeap.

Kompleksitas Waktu Total: O(logn).

Soal-34 Misalkan elemen 7, 2, 10 dan 4 dimasukkan, dalam urutan itu, ke dalam tumpukan maksimum 3 yang valid yang ditemukan dalam pertanyaan di atas, Manakah dari berikut ini yang merupakan urutan item dalam larik yang mewakili resultan tumpukan?

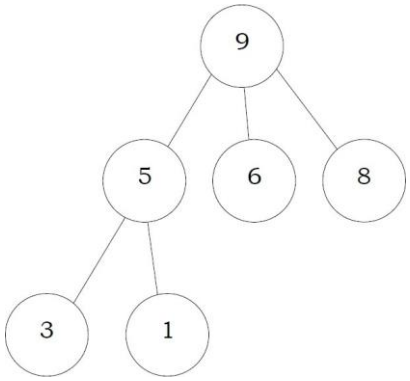
(A) 10, 7, 9, 8, 3, 1, 5, 2, 6, 4

(B) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

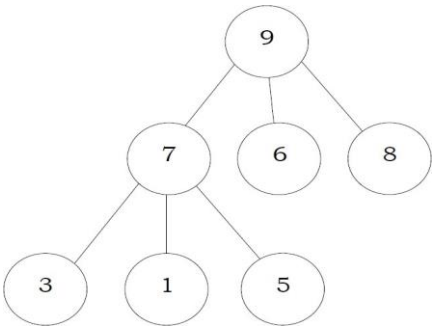
(C) 10, 9, 4, 5, 7, 6, 8, 2, 1, 3

(D) 10, 8, 6, 9, 7, 2, 3, 4, 1, 5

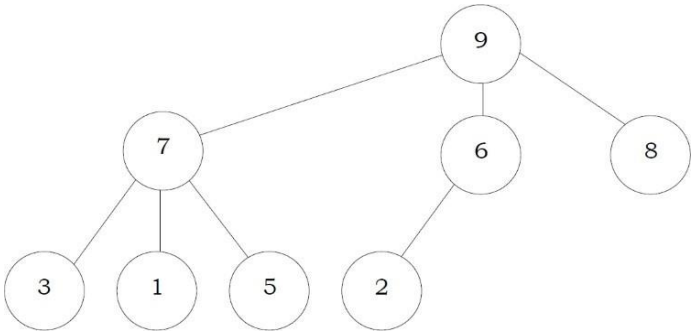
Solusi: Tumpukan maksimum 3-ary dengan elemen 9, 5, 6, 8, 3, 1 adalah:



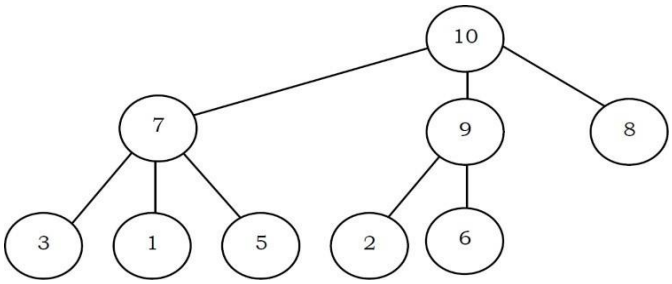
Setelah Penyisipan 7:



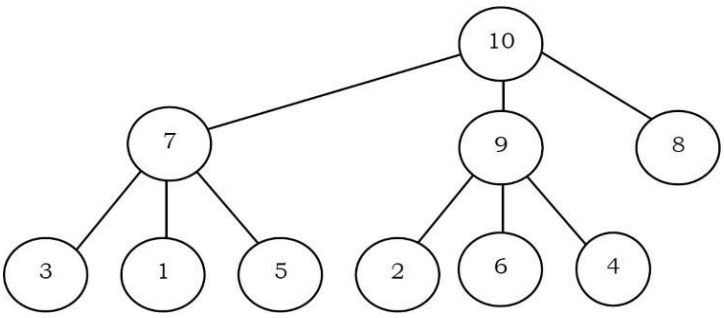
Setelah Penyisipan 2:



Setelah Penyisipan 10:

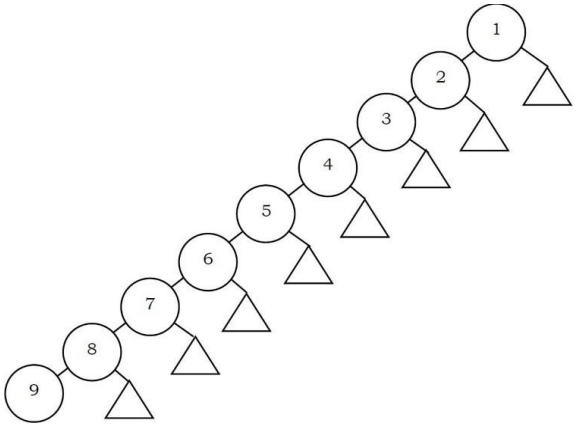


Setelah Penyisipan 4:



Soal-35 Min-heap biner lengkap dibuat dengan memasukkan setiap bilangan bulat di [1,1023] tepat satu kali. Kedalaman sebuah node di heap adalah panjang jalan dari akar heap ke node tersebut. Jadi, akar berada pada kedalaman 0. Kedalaman maksimum di mana bilangan bulat 9 dapat muncul adalah.

Solusi: Seperti yang ditunjukkan pada gambar di bawah, untuk bilangan i yang diberikan, kita dapat memperbaiki elemen i pada tingkat ke-i dan mengatur angka 1 hingga i – 1 ke tingkat di atas. Karena akar berada pada kedalaman nol, kedalaman maksimum elemen ke-i dalam tumpukan-min adalah i – 1. Oleh karena itu, kedalaman maksimum di mana bilangan bulat 9 dapat muncul adalah 8.



Soal-36 Tumpukan d-ary seperti tumpukan biner, tetapi alih-alih 2 anak, simpul memiliki d anak. Bagaimana Anda mewakili tumpukan d-ary dengan n elemen dalam array? Apa ekspresi untuk menentukan induk dari elemen tertentu, Induk(i), dan anak ke-j dari elemen tertentu, Anak(i,j), di mana $1 \leq j \leq d$?

Solusi: Ekspresi berikut menentukan induk dan anak ke-j dari elemen i (di mana $1\ j \leq d$):

$$Parent(i) = \left\lfloor \frac{i+d-2}{d} \right\rfloor$$

$$Child(i,j) = (i-1).d + j + 1$$

BAB 8
SET DISJOINT ADT

8.1 PENDAHULUAN

Dalam bab ini, kita akan merepresentasikan konsep matematika yang penting: himpunan. Ini berarti bagaimana mewakili sekelompok elemen yang tidak memerlukan urutan apa pun. Himpunan lepas ADT adalah yang digunakan untuk tujuan ini. Digunakan untuk menyelesaikan masalah ekivalensi. Ini sangat sederhana untuk diterapkan. Array sederhana dapat digunakan untuk implementasi dan setiap fungsi hanya membutuhkan beberapa baris kode. Set disjoint ADT bertindak sebagai struktur data tambahan untuk banyak algoritma lain (misalnya, algoritma Kruskal dalam teori graf). Sebelum memulai pembahasan kita tentang himpunan lepas ADT, mari kita lihat beberapa sifat dasar himpunan.

8.2 HUBUNGAN EKUIVALENSI DAN KELAS EKUIVALENSI

Untuk pembahasan di bawah ini, mari kita asumsikan bahwa 5 adalah himpunan yang memuat elemen-elemen dan sebuah relasi R didefinisikan di dalamnya. Itu berarti untuk setiap pasangan elemen dalam a,b ∈ 5, a R b bernilai benar atau salah. Jika sebuah R b benar, maka kita katakan a berhubungan dengan b, sebaliknya a tidak berhubungan dengan b. Suatu relasi R disebut an relasi ekivalensi jika memenuhi sifat-sifat berikut:

- *Refleksif:* Untuk setiap elemen a ∈ S.aR a benar.
- *Simetris:* Untuk setiap dua elemen a, b ∈ S, jika a R b benar maka b R a benar.
- *Transitif:* Untuk setiap tiga elemen a, b, c ∈ S, jika a R b dan b R c benar maka a R c adalah benar.

Sebagai contoh, relasi $\leq$ (kurang dari atau sama dengan) dan $\geq$ (lebih besar atau sama dengan) pada himpunan bilangan bulat bukanlah relasi ekivalen. Mereka adalah refleksif (sejak a $\leq$ a) dan transitif (a $\leq$ b dan b $\leq$ c menyiratkan a $\leq$ c) tetapi tidak simetris (a $\leq$ b tidak menyiratkan b $\leq$ a).

Demikian pula, konektivitas kereta api adalah hubungan ekivalensi. Relasi ini bersifat refleksif karena setiap lokasi terhubung dengan dirinya sendiri. Jika ada konektivitas dari kota a ke kota b, maka kota b juga memiliki konektivitas ke kota a, sehingga hubungannya simetris. Akhirnya, jika kota a terhubung dengan kota b dan kota b terhubung dengan kota c, maka kota a juga terhubung dengan kota c.

Kelas ekivalen dari suatu elemen a ∈ S adalah himpunan bagian dari S yang memuat semua elemen yang berelasi dengan a. Kelas ekuivalensi membuat partisi S. Setiap anggota S muncul tepat di satu kelas ekuivalensi. Untuk memutuskan apakah a R b, kita hanya perlu memeriksa apakah a dan b berada dalam kelas (grup) ekivalensi yang sama atau tidak.

Pada contoh di atas, dua kota akan berada dalam kelas ekivalensi yang sama jika memiliki konektivitas rel. Jika mereka tidak memiliki konektivitas maka mereka akan menjadi bagian dari kelas kesetaraan yang berbeda.

Karena perpotongan dua kelas ekivalen adalah kosong (φ), kelas ekivalen kadang-kadang disebut himpunan lepas. Pada bagian selanjutnya, kita akan mencoba melihat operasi yang dapat dilakukan pada kelas ekivalensi.

Operasi yang mungkin dilakukan adalah:

- Membuat kelas ekuivalensi (membuat himpunan)
- Menemukan nama kelas kesetaraan (Temukan)
- Menggabungkan kelas kesetaraan (Union)

8.3 SET TERPISAH ADT

Untuk memanipulasi elemen himpunan kita memerlukan operasi dasar yang didefinisikan pada himpunan. Dalam bab ini, Kita berkonsentrasi pada operasi himpunan berikut:

- MAKESET(X): Membuat set baru yang berisi satu elemen X.
- UNION(X, Y): Membuat himpunan baru yang berisi elemen X dan Y dalam serikatnya dan menghapus himpunan yang berisi elemen X dan Y.
- FIND(X): Mengembalikan nama himpunan yang berisi elemen X.

8.4 APLIKASI

Himpunan Disjoint ADT memiliki banyak aplikasi dan beberapa di antaranya adalah:

- Untuk mewakili konektivitas jaringan
- Pengolahan citra
- Untuk menemukan nenek moyang yang paling tidak sama
- Untuk menentukan kesetaraan automata keadaan terbatas
- Algoritma pohon merentang minimum Kruskal (teori graf)
- Dalam algoritma permainan

8.5 PENGORBANAN DALAM MENGIMPLEMENTASIKAN ADT DISJOINT SET

Mari kita lihat kemungkinan untuk mengimplementasikan operasi himpunan lepas. Awalnya, asumsikan elemen input adalah kumpulan n set, masing-masing dengan satu elemen. Artinya, representasi awal mengasumsikan semua relasi (kecuali relasi refleksif) salah. Setiap himpunan memiliki elemen yang berbeda, sehingga $S_i \cap S_j = \varphi$. Hal ini membuat himpunan terputus-putus.

Untuk menjumlahkan relasi a R b (UNION), pertama-tama kita perlu mengecek apakah a dan b sudah berhubungan atau belum. Ini dapat diverifikasi dengan melakukan FIND pada a dan b dan memeriksa apakah keduanya berada dalam kelas (set) ekivalen yang sama atau tidak.

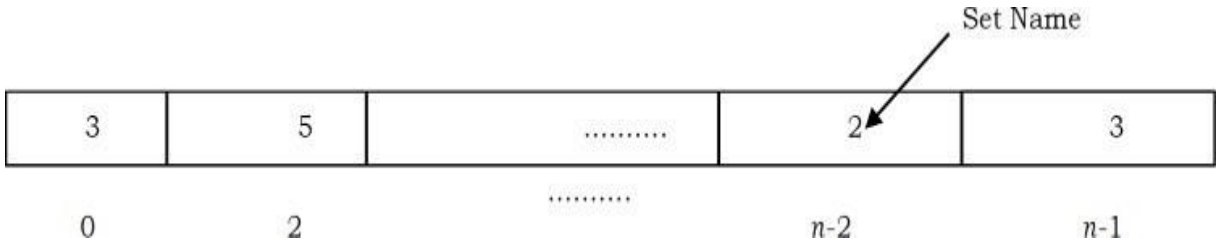
Jika tidak, maka Kita menerapkan UNION. Operasi ini menggabungkan dua kelas ekivalensi yang berisi a dan b ke dalam kelas ekivalensi baru dengan membuat himpunan baru $S_k = S_i \cup S_j$ dan menghapus S_i dan S_j. Pada dasarnya ada dua cara untuk mengimplementasikan operasi FIND/UNION di atas:

- Implementasi FIND FIND (juga disebut Quick FIND)
- Implementasi operasi Fast UNION (juga disebut Quick UNION)

8.6 IMPLEMENTASI FIND

Dalam metode ini, Kita menggunakan array. Sebagai contoh, dalam representasi di bawah array berisi nama yang ditetapkan untuk setiap elemen. Untuk mempermudah, mari kita asumsikan bahwa semua elemen diberi nomor secara berurutan dari 0 hingga n – 1.

Pada contoh di bawah ini, elemen 0 memiliki nama himpunan 3, elemen 1 memiliki nama himpunan 5, dan seterusnya. Dengan representasi ini, FIND hanya membutuhkan O(1) karena untuk elemen apa pun kita dapat menemukan nama yang ditetapkan dengan mengakses lokasi lariknya dalam waktu yang konstan.



Gambar 8.1 Implementasi FIND

Dalam representasi ini, untuk melakukan UNION(a, b) [dengan asumsi bahwa a ada di set i dan b ada di set j] kita perlu memindai array lengkap dan mengubah semua i ke j. Ini membutuhkan O(n).

Urutan n – 1 serikat membutuhkan waktu $O(n^2)$ dalam kasus terburuk. Jika ada operasi FIND $O(n^2)$, kinerja ini baik-baik saja, karena kompleksitas waktu rata-rata adalah O(1) untuk setiap operasi UNION atau FIND. Jika ada lebih sedikit FIND, kerumitan ini tidak dapat diterima.

8.7 IMPLEMENTASI UNION CEPAT (*QUICK UNION*)

Di bagian ini dan selanjutnya, kita akan membahas implementasi UNION yang lebih cepat dan variannya. Ada berbagai cara untuk menerapkan pendekatan ini dan berikut adalah daftar beberapa di antaranya.

- Implementasi UNION yang cepat (*SLOW FIND*)
- Implementasi UNION Cepat (FAST FIND)
- Implementasi UNION yang cepat dengan kompresi jalur

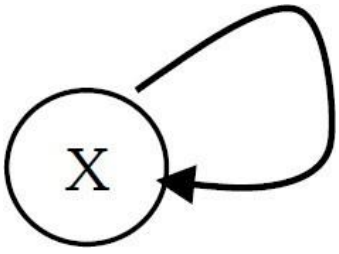
8.8 IMPLEMENTASI UNION LAMBAT (*SLOW FIND*)

Seperti yang telah kita bahas, operasi FIND mengembalikan jawaban yang sama (nama himpunan) jika dan hanya jika mereka berada dalam himpunan yang sama. Dalam merepresentasikan himpunan lepas, tujuan utama kita adalah memberikan nama himpunan yang berbeda untuk setiap kelompok. Secara umum Kita tidak peduli dengan nama himpunan. Salah satu kemungkinan untuk mengimplementasikan himpunan adalah pohon karena setiap elemen hanya memiliki satu akar dan kita dapat menggunakannya sebagai nama himpunan.

Bagaimana ini diwakili? Satu kemungkinan adalah menggunakan array: untuk setiap elemen, pertahankan root sebagai nama yang ditetapkan. Tetapi dengan representasi ini, kita akan memiliki masalah yang sama dengan implementasi array FIND. Untuk mengatasi masalah ini, alih-alih menyimpan root, kita dapat menyimpan induk dari elemen tersebut. Oleh karena itu, menggunakan array yang menyimpan induk dari setiap elemen memecahkan masalah kita.

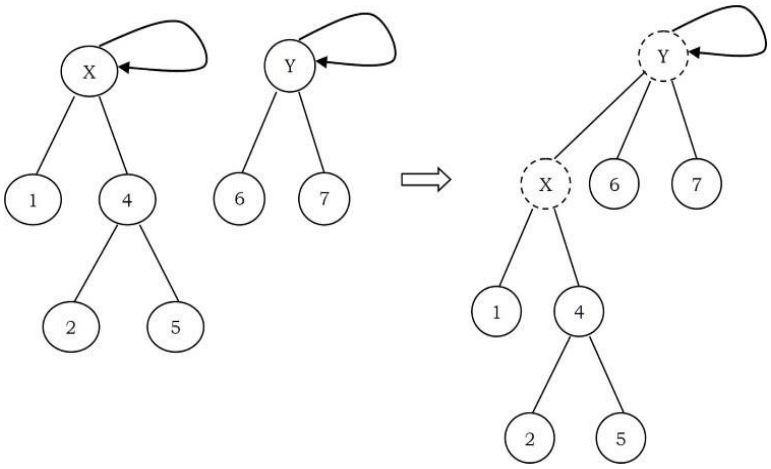
Untuk membedakan simpul akar, mari kita asumsikan induknya sama dengan induk elemen dalam array. Berdasarkan representasi ini, operasi MAKESET, FIND, UNION dapat didefinisikan sebagai:

- (X): Membuat himpunan baru yang berisi satu elemen X dan dalam larik memperbarui induk dari X sebagai X. Itu berarti akar (nama himpunan) dari X adalah X.



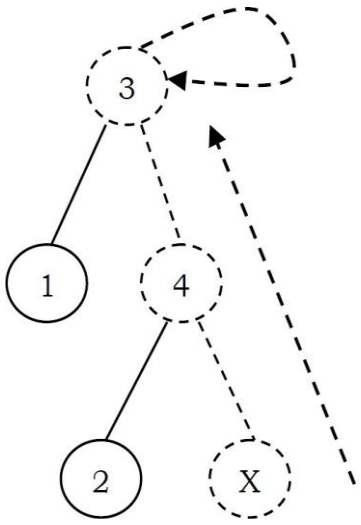
Gambar 8.2 elemen x

- UNION(X, Y): Menggantikan dua set yang berisi X dan Y dengan gabungannya dan dalam array memperbarui induk dari X sebagai Y.



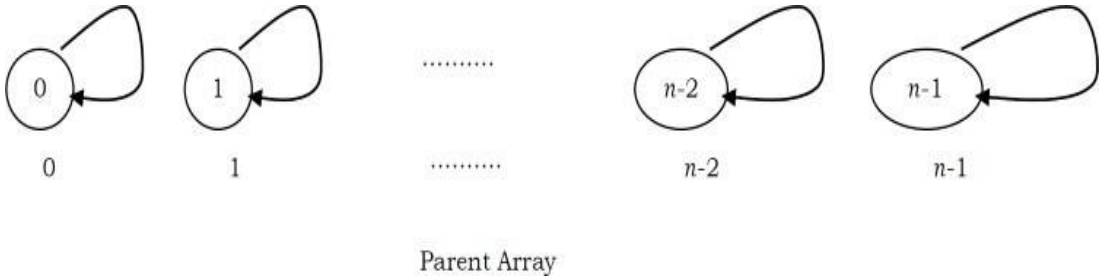
Gambar 8.3 union (x,y)

- FIND(X): Mengembalikan nama himpunan yang berisi elemen X. Kita terus mencari nama himpunan X sampai kita menemukan akar pohon.



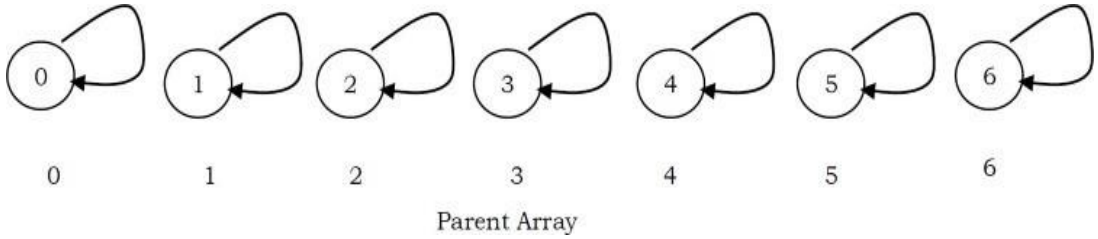
Gambar 8.4 FIND(X)

Untuk elemen 0 hingga n – 1 representasi awalnya adalah:



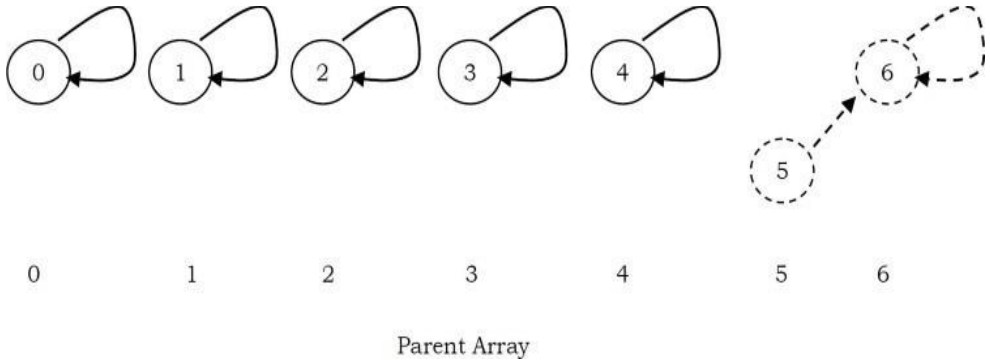
Gambar 8.5 elemen 0 hingga n – 1

Untuk melakukan UNION pada dua set, kita menggabungkan dua pohon dengan membuat akar dari satu pohon menunjuk ke akar yang lain. Konfigurasi Awal untuk elemen 0 hingga 6



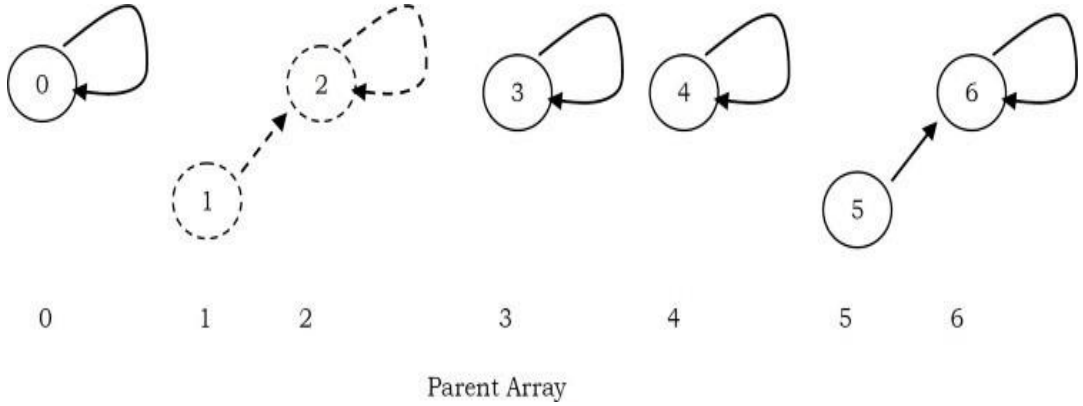
Gambar 8.6 konfigurasi awal elemen 0 – 6

Setelah UNION(5,6)



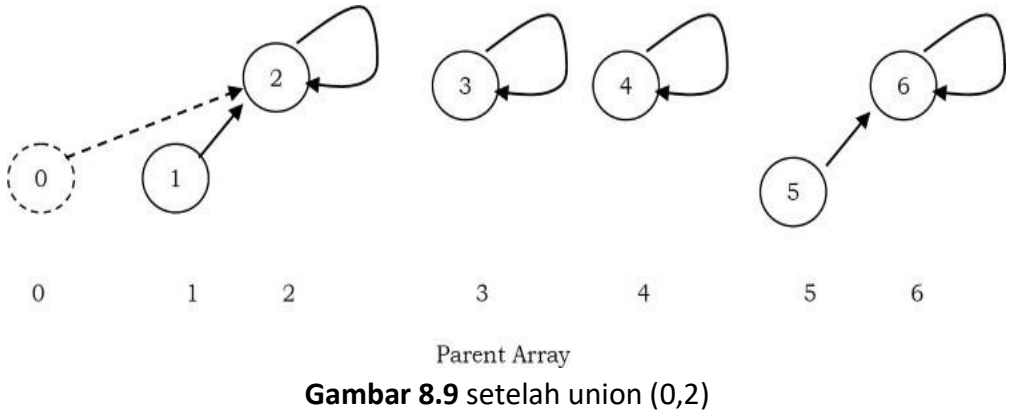
Gambar 8.7 setelah union (5,6)

Setelah UNION(1,2)



Gambar 8.8 setelah union (1,2)

Setelah UNION(0,2)



Gambar 8.9 setelah union (0,2)

Satu hal penting yang harus diperhatikan di sini adalah, operasi UNION hanya mengubah induk root, tetapi tidak untuk semua elemen dalam himpunan. Karena ini, kompleksitas waktu operasi UNION adalah O(1).

FIND(X) pada elemen X dilakukan dengan mengembalikan akar pohon yang mengandung X. Waktu untuk melakukan operasi ini sebanding dengan kedalaman simpul yang mewakili X.

Dengan menggunakan metode ini, dimungkinkan untuk membuat pohon dengan kedalaman n - 1 (Pohon Miring). Waktu berjalan kasus terburuk dari FIND adalah O(n) dan m operasi FIND berturut-turut membutuhkan waktu O(mn) dalam kasus terburuk.

MAKESET

```
void MAKESET( int S[], int size) {
    for(int  i = size-1; i >=0; i-- )
        S[i] = i;
}
```

TEMUKAN

```
int FIND(int S[], int size, int X) {
    if(!(X >= 0 && X < size))
        return -1;
    if( S[X] == X )
        return X;
    else  return FIND(S, S[X]);
}
```

PERSATUAN

```
void UNION( int S[], int size, int root1, int root2 ) {
    if(FIND(S, size, root1) == FIND(S, size, root2))
        return;
    if(!((root1 >= 0 && root1 < size) && (root2 >= 0 && root2 < size)))
        return;
    S[root1] = root2;
}
```

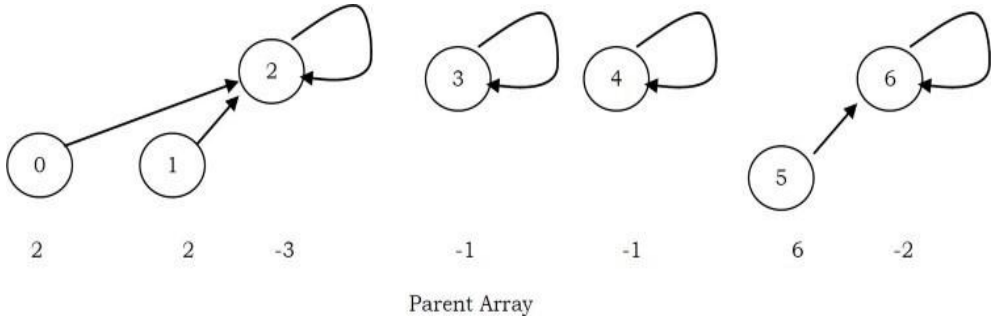
Implementasi Union Cepat (Quick Find)

Masalah utama dengan pendekatan sebelumnya adalah bahwa, dalam kasus terburuk kita mendapatkan pohon miring dan sebagai hasilnya operasi FIND mengambil kompleksitas waktu O(n). Ada dua cara untuk meningkatkannya:

- UNION menurut Ukuran (juga disebut UNION menurut Berat): Jadikan pohon yang lebih kecil sebagai subpohon dari pohon yang lebih besar
- UNION by Height (juga disebut UNION by Rank): Jadikan pohon dengan tinggi lebih kecil sebagai subpohon dari pohon yang lebih tinggi

UNI menurut Ukuran

Dalam representasi sebelumnya, untuk setiap elemen i Kita telah menyimpan i (dalam array induk) untuk elemen root dan untuk elemen lain Kita telah menyimpan induk i. Tetapi dalam pendekatan ini kita menyimpan negatif dari ukuran pohon (artinya, jika ukuran pohon adalah 3 maka simpan –3 dalam larik induk untuk elemen akar). Untuk contoh sebelumnya (setelah UNION(0,2)), representasi baru akan terlihat seperti:



Gambar 8.10 uni menurut ukuran

Asumsikan bahwa ukuran satu set elemen adalah 1 dan simpan – 1. Selain ini, tidak ada perubahan.

MAKESET

```
void MAKESET( int S[], int size) {
    for(int  i = size-1; i >= 0; i-- )
        S[i] = -1;
}
```

TEMUKAN

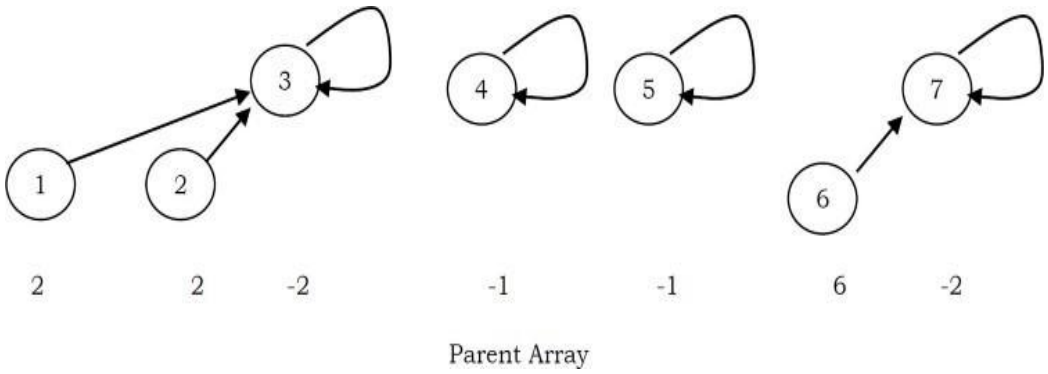
```
int FIND(int S[], int size, int X) {
   if(!(X >= 0 && X < size)) )
      return -1;
   if( S[X] == -1 )
      return X;
   else return FIND(S, S[X]);
}
```

UNI menurut Ukuran

```
void UNIONBySize(int S[], int size, int root1, int root2) {
   if((FIND(S, size, root1) == FIND(S, size, root2)) && FIND(S, size, root1) != -1)
      return;
   if( S[root2] < S[root1] ) {
      S[root1] = root2;
      S[root2] += S[root1];
   }
   else {
      S[root2] = root1;
      S[root1] += S[root2];
   }
}
```

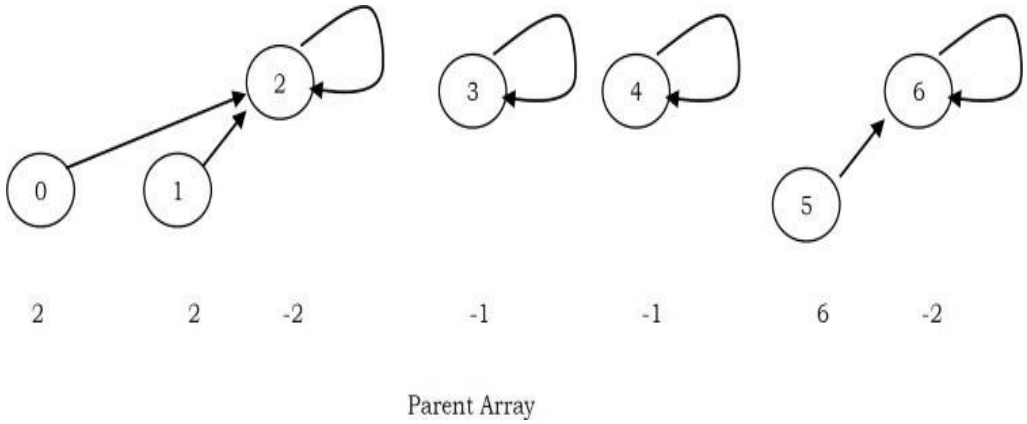
Catatan: *Tidak ada perubahan dalam implementasi operasi FIND.*

UNION berdasarkan Tinggi (UNION berdasarkan Peringkat)



Gambar 8.11 union berdasarkan tinggi

Seperti pada UNION berdasarkan ukuran, dalam metode ini kita menyimpan negatif dari tinggi pohon (artinya, jika tinggi pohon adalah 3 maka kita menyimpan –3 dalam larik induk untuk elemen akar). Kita menganggap tinggi pohon dengan satu set elemen adalah 1. Untuk contoh sebelumnya (setelah UNION(0,2)), representasi baru akan terlihat seperti:



Gambar 8.12 union berdasarkan ukuran

UNION menurut Tinggi

```
void UNIONByHeight(int S[], int size, int root1, int root2) {
    if((FIND(S, size, root1) == FIND(S, size, root2)) && FIND(S, size, root1) != -1)
        return;
    if( S[root2] < S[root1] )
        S[root1] = root2;
    else {
          if( S[root2] == S[root1] ){
               S[root1]--;
              S[root2] = root1;
          }
    }
}
```

Catatan: *Untuk operasi FIND tidak ada perubahan implementasi.*

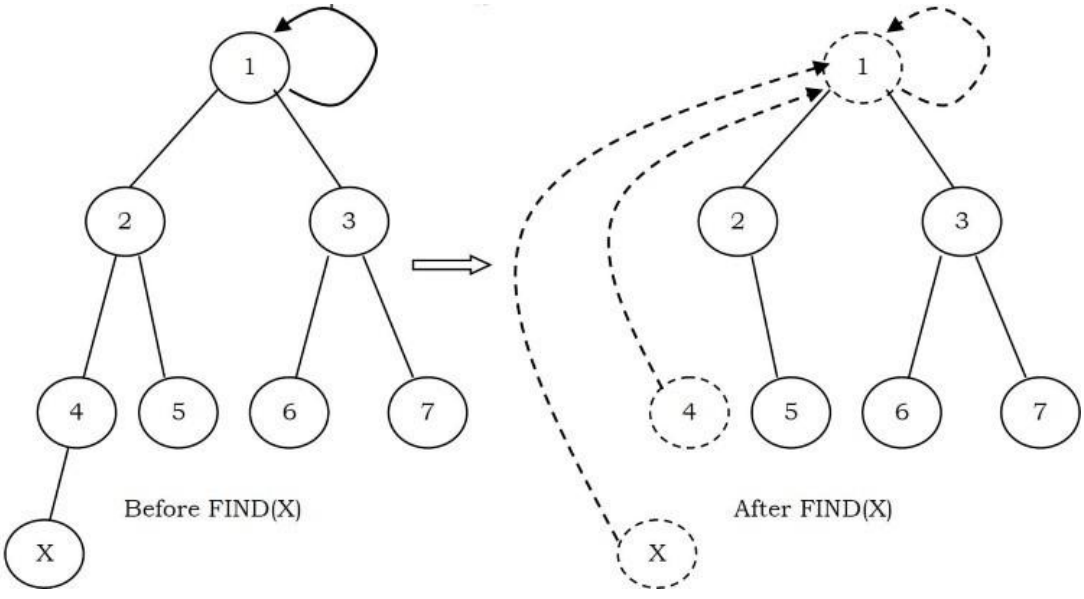
Membandingkan UNION berdasarkan Ukuran dan UNION berdasarkan Tinggi

Dengan UNION berdasarkan ukuran, kedalaman setiap node tidak pernah lebih dari logn. Ini karena sebuah simpul awalnya berada pada kedalaman 0. Ketika kedalamannya meningkat sebagai akibat dari UNION, ia ditempatkan di pohon yang setidaknya dua kali lebih besar dari sebelumnya. Itu berarti kedalamannya dapat ditingkatkan pada sebagian besar waktu logn. Ini berarti bahwa waktu berjalan untuk operasi FIND adalah O(logn), dan urutan m operasi membutuhkan O(m logn).

Demikian pula dengan UNION berdasarkan ketinggian, jika kita mengambil UNION dari dua pohon dengan tinggi yang sama, tinggi UNION adalah satu lebih besar dari tinggi umum, dan sebaliknya sama dengan maksimum dari dua ketinggian. Ini akan menjaga ketinggian pohon dari n node agar tidak tumbuh melewati O(logn). Urutan m UNION dan FIND masih dapat berharga O(m logn).

Kompresi Jalur

Operasi FIND melintasi daftar node dalam perjalanan ke root. Kita dapat membuat operasi FIND selanjutnya menjadi efisien dengan membuat masing-masing simpul ini menunjuk langsung ke akar. Proses ini disebut kompresi jalur. Misalnya, dalam operasi FIND(X), kita melakukan perjalanan dari X ke akar pohon. Efek dari kompresi jalur adalah bahwa setiap node pada jalur dari X ke root memiliki induknya yang berubah menjadi root.



Gambar 8.13 kompresi jalur

Dengan kompresi jalur, satu-satunya perubahan pada fungsi FIND adalah bahwa S[X] dibuat sama dengan nilai yang dikembalikan oleh FIND. Artinya, setelah akar himpunan ditemukan secara rekursif, X dibuat menunjuk langsung ke akar itu. Ini terjadi secara rekursif ke setiap node di jalur ke root.

TEMUKAN dengan kompresi jalur

```
int FIND(int S[], int size, int X) {
    if(!(X >= 0 && X < size))
        return;
    if( S[X] <= 0 )
        return X;
    else return(S[X] = FIND( S, S[X]));
}
```

Catatan: *Kompresi jalur kompatibel dengan UNION berdasarkan ukuran tetapi tidak dengan UNION berdasarkan ketinggian karena tidak ada cara yang efisien untuk mengubah ketinggian pohon.*

8.9 RINGKASAN

Melakukan m operasi union-find pada sekumpulan n objek.

Tabel 8.1 m operasi *union-find* pada sekumpulan n objek

Algoritma	Waktu Kasus Terburuk
Cari Cepat	*m* *n*
serikat cepat	*m* *n*
Quick-Union berdasarkan Ukuran/Tinggi	*n + m logn*
Kompresi jalur	*n + m logn*
Quick-Union berdasarkan Ukuran/Tinggi + Kompresi Jalur	(*m + n*) *logn*

8.10 HIMPUNAN TERPISAH: MASALAH & SOLUSI

Soal-1 Pertimbangkan daftar kota c1; c2,...,cn. Asumsikan bahwa kita memiliki relasi R sedemikian sehingga, untuk setiap i,j, R(ci,cj) adalah 1 jika kota-kota ci dan cj berada dalam keadaan yang sama, dan 0 sebaliknya. Jika R disimpan sebagai tabel, berapa banyak ruang yang dibutuhkan?

Solusi R harus memiliki entri untuk setiap pasangan kota. Ada (n2) di antaranya.

Soal-2 Untuk Soal-1, dengan menggunakan ADT himpunan Disjoint, berikan algoritma yang menempatkan setiap kota dalam himpunan sedemikian rupa sehingga ci dan cj berada dalam himpunan yang sama jika dan hanya jika mereka berada dalam keadaan yang sama.

Solusi:

```
for (i = 1; i<= n; i++) {
    MAKESET(ci);
    for (j = 1; j <= i-1; j++) {
        if(R(cj, ci)) {
            UNION(cj, ci);
            break;
        }
    }
}
```

Soal-3 Untuk Soal-1, ketika kota-kota tersebut disimpan dalam himpunan ADT Disjoint, jika kita diberikan dua kota ci dan cj, bagaimana kita memeriksa apakah mereka berada dalam keadaan yang sama?

Solusi: Kota ci dan cj berada dalam keadaan yang sama jika dan hanya jika FIND(c_i) = FIND(c_j).

Soal-4 Untuk Soal-1, jika kita menggunakan linked-list dengan UNION berdasarkan ukuran untuk mengimplementasikan ADT union-find, berapa banyak ruang yang kita gunakan untuk menyimpan kota?

Solusi: Ada satu simpul per kota, jadi ruangnya adalah Θ (n).

Soal-5 Untuk Soal-1, jika kita menggunakan pohon dengan UNION berdasarkan peringkat, berapa kasus terburuk waktu berjalan dari algoritma dari Soal-2?

Solusi: Setiap kali kita melakukan UNION dalam algoritma dari Soal-2, argumen kedua adalah pohon berukuran 1. Oleh karena itu, semua pohon memiliki tinggi 1, sehingga setiap serikat membutuhkan waktu O(1). Waktu berjalan kasus terburuk adalah Θ (n^2).

Soal-6 Jika kita menggunakan pohon tanpa union-by-rank, berapa kasus terburuk waktu berjalan algoritma dari Soal-2? Apakah ada lebih banyak skenario terburuk daripada Masalah-5?

Solusi: Karena kasus khusus serikat pekerja, serikat berdasarkan peringkat tidak membuat perbedaan untuk algoritma Kita. Oleh karena itu, semuanya sama seperti pada Soal-5.

Soal-7 Dengan algoritma quick-union kita tahu bahwa urutan operasi n (penyatuan dan penemuan) dapat memakan waktu sedikit lebih lama daripada waktu linier dalam kasus terburuk. Jelaskan mengapa jika semua penemuan dilakukan sebelum semua penyatuan, urutan n operasi dijamin memakan waktu O(n).

Solusi: Jika operasi find dilakukan terlebih dahulu, maka operasi find memakan waktu O(1) setiap kali karena setiap item adalah akar dari pohonnya sendiri. Tidak ada

item yang memiliki parent, jadi menemukan set item membutuhkan sejumlah operasi yang tetap. Operasi serikat selalu memakan waktu O(1). Oleh karena itu, urutan n operasi dengan semua temuan sebelum serikat membutuhkan waktu O(n).

Soal-8 Dengan mengacu pada Soal-7, jelaskan mengapa jika semua penyatuan dilakukan sebelum semua temuan, urutan operasi n dijamin memakan waktu O(n).

Solusi: Masalah ini membutuhkan analisis diamortisasi. Operasi pencarian bisa mahal, tetapi operasi pencarian yang mahal ini diimbangi dengan banyak operasi serikat yang murah.

Akuntansinya adalah sebagai berikut. Operasi serikat pekerja selalu memakan waktu O(1), jadi katakanlah mereka memiliki biaya aktual $\sqrt{2}1$. Tetapkan setiap operasi serikat biaya diamortisasi $\sqrt{2}2$, jadi setiap operasi serikat memasukkan $\sqrt{2}1$ ke dalam akun. Setiap operasi serikat menciptakan anak baru. (Beberapa simpul yang bukan merupakan anak dari simpul lain sebelumnya adalah anak sekarang.) Ketika semua operasi gabungan selesai, ada Rp 15.000 di akun untuk setiap anak, atau dengan kata lain, untuk setiap simpul dengan kedalaman satu atau lebih besar. Katakanlah operasi find(u) bernilai $\sqrt{2}1$ jika u adalah root. Untuk setiap node lain, operasi find memerlukan biaya tambahan $\sqrt{2}1$ untuk setiap pointer induk yang dilintasi operasi find. Jadi biaya sebenarnya adalah $\sqrt{2}$ (1 + d), di mana d adalah kedalaman u. Tetapkan setiap operasi pencarian biaya diamortisasi dari $\sqrt{2}2$. Ini mencakup kasus di mana u adalah akar atau anak dari akar. Untuk setiap penunjuk induk tambahan yang dilalui, $\sqrt{2}1$ ditarik dari akun untuk membayarnya.

Untungnya, kompresi jalur mengubah pointer induk dari semua node yang kita bayar 1 untuk dilalui, sehingga node ini menjadi anak-anak dari root. Semua node yang dilalui dengan kedalaman 2 atau lebih besar bergerak ke atas, sehingga kedalamannya sekarang 1. Kita tidak perlu membayar untuk melintasi node ini lagi. Katakanlah sebuah simpul adalah cucu jika kedalamannya 2 atau lebih besar.

Setiap kali find(u) mengunjungi cucu, $\sqrt{2}1$ ditarik dari akun, tetapi cucu tidak lagi menjadi cucu. Jadi jumlah maksimum dolar yang bisa ditarik dari rekening adalah jumlah cucu. Tapi awalnya Kita menaruh Rp 15.000 di bank untuk setiap anak, dan setiap cucu adalah anak, jadi saldo bank tidak akan pernah turun di bawah nol. Oleh karena itu, amortisasi berhasil. Operasi penyatuan dan pencarian keduanya memiliki biaya diamortisasi sebesar $\sqrt{2}2$, jadi setiap urutan n operasi di mana semua penyatuan dilakukan terlebih dahulu membutuhkan waktu O(n).

DAFTAR PUSTAKA

Akash. Programming Interviews, tech-queries.blogspot.com.

Alfred V.Aho,J. E. (1983). Data Structures and Algorithms. Addison-Wesley.

Algorithms.Retrieved from cs.princeton.edu/algs4/home

Anderson., S. E. Bit Twiddling Hacks. Retrieved 2010, from Bit Twiddling Hacks: graphics. Stanford. edu

Bentley, J. AT&T Bell Laboratories. Retrieved from AT&T Bell Laboratories.

Bondalapati, K. Interview Question Bank. Retrieved 2010, from Interview Question Bank: halcyon.usc.edu/~kiran/msqs.html

Chen. Algorithms hawaii.edu/~chenx.

Database, P.Problem Database. Retrieved 2010, from Problem Database: datastructures.net

Drozdek, A. (1996). Data Structures and Algorithms in C++.

Ellis Horowitz, S. S. Fundamentals of Data Structures.

Gilles Brassard, P. B. (1996). Fundamentals of Algorithmics.

Hunter., J. Introduction to Data Structures and Algorithms. Retrieved 2010, from Introduction to Data Structures and Algorithms.

James F. Korsh, L. J. Data Structures, Algorithms and Program Style Using C.

John Mongan, N. S. (2002). Programming Interviews Exposed. Wiley-India. .

Judges. Comments on Problems and Solutions. http://www.informatik.uni-ulm.de/acm/Locals/2003/html/judge, html.

Kalid. P, NP, and NP-Complete. Retrieved from P, NP, and NP-Complete.: cs.princeton.edu/~kazad

Knuth., D. E. (1973). Fundamental Algorithms, volume 1 of The Art of Computer Programming. Addison-Wesley.

Leon, J. S. Computer Algorithms. Retrieved 2010, from Computer Algorithms: math.uic.edu/~leon

Leon., J. S. Computer Algorithms, math.uic.edu/~leon/cs-mcs401-s08.

OCF. Algorithms. Retrieved 2010, from Algorithms: ocf.berkeley.edu

Parlante., N. Binary Trees. Retrieved 2010, from cslibrary.stanford.edu: cslibrary.stanford.edu

Patil., V. Fundamentals of data structures. Nirali Prakashan.

Poundstone., W. HOW WOULD YOU MOVE MOUNT FUJI? New York Boston.: Little, Brown and Company.

Pryor, M. Tech Interview. Retrieved 2010, from Tech Interview: techinterview.org

Questions, A. C. A Collection of Technical Interview Questions. Retrieved 2010, from A Collection of Technical Interview Questions

S. Dasgupta, C. P. Algorithms cs.berkeley.edu/~vazirani.

Sedgewick., R. (1988). Algorithms. Addison-Wesley.

Sells, C. (2010). Interviewing at Microsoft. Retrieved 2010, from Interviewing at Microsoft

Shene, C.-K. Linked Lists Merge Sort Implementation.

Sinha, P. Linux Journal. Retrieved 2010, from: linuxjournal.com/article/6828.

Structures., d. D. www.math-cs.gordon.edu. Retrieved 2010, from www.math- cs.gordon.edu

T. H. Cormen, C. E. (1997). Introduction to Algorithms. Cambridge: The MIT press.

Tsiombikas, J. Pointers Explained, nuclear.sdf-eu.org.

Warren., H. S. (2003). Hackers Delight. Addison-Wesley.

Weiss., M. A. (1992). Data Structures and Algorithm Analysis in C.

SANDRASI http://sandrasi-sw.blogspot.in/