Ade Mulyana, dkk.

Cara Mudah Mempelajari Algoritma dan Struktur Data

DIVA Press

CARA MUDAH MEMPELAJARI ALGORITMA DAN STRUKTUR DATA

Penulis: **Ade Mulyana, dkk.**
Editor: **Chacha**
Tata Sampul: **xxx**
Tata Isi: **Adelia**
Pracetak: **Antini, Dwi, Wardi**

Cetakan Pertama, **Oktober 2021**

Penerbit
DIVA Press
(Anggota IKAPI)
Sampangan Gg. Perkutut No.325-B
Jl. Wonosari, Baturetno
Banguntapan Yogyakarta
Telp: (0274) 4353776, 081804374879
Fax: (0274) 4353776
E-mail:redaksi_divapress@yahoo.com
sekred2.divapress@gmail.com
Blog: www.blogdivapress.com
Website: www.divapress-online.com

Perpustakaan Nasional: Katalog Dalam Terbitan (KDT)

Mulyana, Ade, dkk.

Cara Mudah Mempelajari Algoritma dan Struktur Data/Ade Mulyana, dkk.; editor, Chacha–cet. 1–Yogyakarta: DIVA Press, 2021

168 hlmn; 14 x 20 cm
ISBN -

I. Judul
II. Chacha

Daftar Isi

1

Mengenal Struktur Data

Tujuan:
- Mahasiswa memahami maksud struktur data.

A. Pendahuluan

Struktur data adalah cara penyimpanan, penyusunan, dan pengaturan data di dalam media penyimpanan komputer, sehingga data tersebut dapat digunakan secara efisien. Struktur data juga bisa berarti tata letak data yang berisi kolom-kolom data, baik kolom yang tampak oleh pengguna (*user*) ataupun kolom yang hanya digunakan untuk keperluan pemrograman yang tidak tampak oleh pengguna.

1. Hubungan antara Algoritma dan Struktur Data

Kumpulan instruksi komputer dinamakan program, sedangkan metode dan tahapan sistematis dalam program disebut algoritma. Program ditulis menggunakan bahasa pemrograman. Dengan demikian, program merupakan suatu

implementasi bahasa pemrograman. Terkait hal ini, beberapa pakar memberikan formula bahwa:

Program = Struktur Data + Algoritma

Algoritma adalah suatu langkah atau prosedur yang ditujukan untuk memanipulasi data. Misalnya, algoritma diperlukan untuk memasukkan data ke dalam suatu struktur data atau mencari suatu data yang tersimpan dalam struktur data. Algoritma yang baik namun tanpa pemilihan struktur data yang tepat akan membuat program menjadi kurang baik, demikian pula sebaliknya. Beberapa contoh struktur data antara lain array, tumpukan, antrean, senarai berantai, pohon biner, dan tabel Hash.

2. Kelebihan dan Kekurangan Struktur Data

Berikut beberapa kelebihan dan kekurangan struktur data:

a. Array

- Kelebihan

  Array merupakan struktur data yang paling mudah, memori ekonomis apabila semua elemen terisi, dan waktu akses sama ke setiap elemen.

- Kekurangan:

  Array boros memori jika banyak elemen yang tidak digunakan, serta struktur data bersifat statis.

b. Array yang Terurutkan

- Kelebihan:

  Jika dibandingkan dengan *array* biasa, *array* yang terurutkan lebih cepat dalam hal pencarian data.

- Kekurangan:

  Array yang terurutkan lambat dalam pengisian dan penghapusan data, serta ukurannya tetap walaupun *array* tidak terisi penuh.

c. Tumpukan

- Kelebihan:

  Penambahan dan penghapusan data dapat dilakukan dengan cepat, yaitu O (1) di mana selama memori masih tersedia, penambahan data bisa terus dilakukan. Dengan demikian, tidak ada kekhawatiran terjadinya *stack overflow*.

- Kekurangan:

  Setiap sel tidak hanya menyimpan *value*, tetapi juga *pointer* ke sel berikutnya. Hal ini menyebabkan implementasi *stack* menggunakan *linked list* akan memerlukan memori yang lebih banyak daripada jika diimplementasikan dengan *array*. Tiap elemen pada *linked list* hanya dapat diakses dengan cara sekuensial, sehingga lambat, yaitu O (n).

d. Antrean

- ✧ Kelebihan:

  Data yang pertama masuk, maka data itulah akan pertama kali dilayani.

- ✧ Kekurangan:

  Apabila waktu pelayanan habis, data yang terakhir masuk kemungkinan bisa tidak dilayani.

e. Senarai Berantai

- ✧ Kelebihan:

  Penyisipan dan penghapusan data mudah dilakukan.

- ✧ Kekurangan:

  Pencarian data cukup lama.

f. Pohon Biner

- ✧ Kelebihan:

  Pencarian dan penyisipan data mudah dilakukan.

- ✧ Kekurangan:

  Penghapusan kompleks.

g. Tabel Hash

- ✧ Kelebihan:

  Akses cepat apabila kunci diketahui serta penyisipan data cepat.

- ✧ Kekurangan:

  Algoritma penghapusan ada yang sederhana dan ada pula yang kompleks, serta akses lambat apabila kunci tidak diketahui.

Gambar 1.1
Ilustrasi Struktur Data

B. Tujuan Struktur Data

Struktur data bertujuan agar cara merepresentasikan data dalam membuat program dapat dilakukan dengan efisien dalam pengolahan di memori serta mempermudah melakukan pengolahan penyimpanan dari program ke *storage*. Struktur data sebenarnya juga meliputi larik (*array*) dan

rekaman (*record*) pada berkas beruntun (*sequential file*) yang dipelajari dalam algoritma. Pemrograman pada dasarnya juga merupakan bagian dari struktur data untuk penyimpanan data di dalam memori sebagai *array* atau di dalam *file* sebagai *record*. Penyimpanan *record* di dalam *file* adalah cikal bakal adanya aplikasi basis data, karena aplikasi basis data pada dasarnya berbasis pada konsep penyimpanan *record* di dalam *file*.

Pembuatan struktur data dimulai dari analisis perancangan data yang harus dimanipulasi di memori komputer agar program yang dibuat lebih efisien. Langkah selanjutnya adalah mengimplementasikan struktur data dalam bahasa pemrograman, kemudian menggunakan struktur data yang telah dibuat untuk memanipulasi data di memori dalam sebuah program. Sebagai contoh, perhatikan ilustrasi langkah-langkah berikut:

Ilustrasi	Keterangan
nama | alamat | No_ktp | No_telp ? | ? | ? | ?	Misalkan ada sebuah data manusia yang terdiri dari: Nama Alamat No_ktp (nomor KTP) No_telp (nomor Telepon) Dan diperlukan untuk menyimpan data manusia, maka dalam logika akan dipersiapkan tempat untuk menyimpan sebuah data manusia, maka dibuat sebuah tipe data bentuk untuk menyimpan data manusia.
nama | alamat | no_ktp | no_telp nofri | riau | 1234322 | 08123453	Misalkan dari data manusia yang ada diisi dengan data seorang manusia.
nama | alamat | no_ktp | no_telp Dzul | Riau | 123453433 | 081254333 Ayuk | Riau | 231445432 | 081233343 Yaya n | riau | 223433234 | 081399887	Misalkan dari data manusia yang ada digunakan untuk menampung bebrapa data manusia.

Berdasarkan ilustrasi di atas dapat diketahui bahwa sebuah struktur data adalah cara menyediakan tempat yang baik dan tersusun secara terstruktur agar data yang disimpan dapat dibaca dengan mudah.

C. Jenis/Tipe Data (Data Type)

Jenis data terdiri dari:

1. Set Nilai Data
2. Set Operasi yang Dapat Diterapkan pada Nilai Tersebut

D. Klasifikasi Jenis Data

Jenis-jenis data dikelompokkan sebagai berikut:

1. Simple Data Type (Jenis Data Sederhana)
2. Item Data Individual
3. Data Structures/Data Aggregates (Struktur Data)
4. Kombinasi dari Item Data Individual
5. Membentuk Item Data Lain

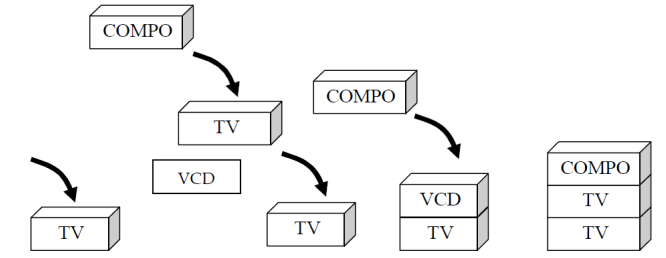
2

Stack

Tujuan:
- Mahasiswa memahami LIFO pada stack dan operasinya.

A. Definisi Stack

Stack adalah suatu tumpukan dari benda. Konsep utamanya adalah LIFO (*last in first out*), yaitu benda yang terakhir masuk dalam *stack* akan menjadi benda pertama yang dikeluarkan dari *stack*.



Pada gambar di atas, jika kita ingin mengambil sesuatu dari tumpukan, maka kita harus mengambil benda paling atas terlebih dahulu, yakni compo. Apabila VCD langsung diambil, maka compo akan jatuh. Prinsip *stack* ini dapat diterapkan dalam pemrograman. Di C++, ada dua cara penerapan prinsip *stack*, yakni dengan *array* dan *linked list*.

Setidaknya, *stack* harus memiliki operasi-operasi sebagai berikut:

1. Push ➔ Untuk menambahkan *item* pada tumpukan paling atas.
2. Pop ➔ Untuk mengambil *item* teratas.
3. Clear ➔ Untuk mengosongkan *stack*.
4. IsEmpty ➔ Untuk memeriksa apakah *stack* kosong.
5. IsFull ➔ Untuk memeriksa apakah *stack* sudah penuh.
6. Retrieve ➔ Untuk mendapatkan nilai dari *item* teratas.

B. Stack dengan Array

Sesuai dengan sifat *stack*, pengambilan/penghapusan pada elemen dalam *stack* harus dimulai dari elemen teratas. Operasi-operasi *stack* dengan *array*, antara lain:

1. IsFull

Fungsi ini berguna untuk memeriksa apakah *stack* yang ada sudah penuh. *Stack* penuh jika puncak *stack* terdapat tepat di bawah jumlah maksimum yang bisa ditampung oleh *stack*, atau dengan kata lain Top = MAX_STACK -1.

2. Push

Fungsi ini berguna untuk menambahkan sebuah elemen ke dalam *stack* dan tidak bisa dilakukan lagi apabila *stack* sudah penuh.

3. IsEmpty

Fungsi ini berguna untuk menentukan apakah *stack* kosong atau tidak. Tanda bahwa *stack* kosong adalah Top bernilai kurang dari nol.

4. Pop

Fungsi ini berguna untuk mengambil elemen teratas dari *stack*, dengan syarat *stack* tidak boleh kosong.

5. Clear

Fungsi ini berguna untuk mengosongkan *stack* dengan cara mengeset Top dengan -1. Jika Top bernilai kurang dari nol, maka *stack* dianggap kosong.

6. Retrieve

Fungsi ini berguna untuk melihat nilai yang berada pada posisi tumpukan teratas.

C. Double Stack dengan Array

Metode ini merupakan teknik khusus yang dikembangkan untuk menghemat pemakaian memori dalam pembuatan dua *stack* dengan *array*. Intinya adalah penggunaan satu buah *array* untuk menampung dua *stack*. Dari sini terlihat jelas bahwa sebuah *array* dapat dibagi untuk dua *stack*, di mana *stack* 1 bergerak ke atas dan *stack* 2 bergerak ke bawah. Jika Top 1 (elemen teratas dari *stack* 1) bertemu dengan Top 2 (elemen teratas dari *stack* 2), maka *double stack* telah penuh. Implementasi *double stack* dengan *array* adalah dengan memanfaatkan operasi-operasi yang tidak berbeda jauh dengan operasi *single stack* dengan *array*.

Adapun operasi-operasi *double stack* dengan *array*, antara lain:

1. IsFull

Operasi ini berfungsi memeriksa apakah *double stack* sudah penuh. *Stack* dianggap penuh jika Top[0] dan Top[1] bersentuhan, sehingga *stack* tidak memiliki ruang kosong. Dengan kata lain, (Top[0] + 1) > Top[1].

2. Push

Operasi ini berfungsi memasukkan sebuah elemen ke salah satu *stack*.

3. IsEmpty

Operasi ini berfungsi memeriksa apakah *stack* pertama atau kedua kosong. *Stack* pertama dianggap kosong jika puncak *stack* bernilai kurang dari nol, sedangkan *stack* kedua dianggap kosong jika puncak *stack* sama atau melebihi MAX_STACK.

4. Pop

Operasi ini berfungsi mengeluarkan elemen teratas dari salah satu *stack*.

5. Clear

Operasi ini berfungsi mengosongkan salah satu *stack*.

D. Stack dengan Single Linked List

Selain implementasi *stack* dengan *array* sebagaimana telah dijelaskan pada pembahasan sebelumnya, terdapat cara lain untuk mengimplementasikan *stack* dalam C++, yakni dengan *single linked list*. Keunggulan *linked list* dibandingkan *array* adalah penggunaan alokasi memori yang dinamis sehingga menghindari pemborosan memori. Misalnya, pada *stack* dengan *array* disediakan tempat untuk *stack* berisi 150 elemen. Ketika dipakai oleh *user*, *stack* hanya diisi 50 elemen. Ini berarti telah terjadi pemborosan memori untuk sisa 100 elemen yang tak terpakai. Namun, dengan menggunakan *linked list*, tempat yang disediakan akan sesuai dengan

banyaknya elemen yang mengisi *stack*. Oleh karena itu pula, dalam *stack* dengan *linked list* tidak ada istilah *full*. Sebab, biasanya program tidak menentukan jumlah elemen *stack* yang mungkin ada (kecuali jika sudah dibatasi oleh pembuatnya). Namun demikian, *stack* ini pun sebenarnya memiliki batas kapasitas, yakni dibatasi oleh jumlah memori yang tersedia.

Adapun operasi-operasi untuk *stack* dengan *linked list*, di antaranya sebagai berikut:

1. IsEmpty

Operasi ini berfungsi memeriksa apakah *stack* yang ada masih kosong.

2. Push

Operasi ini berfungsi memasukkan elemen baru ke dalam *stack*. *Push* di sini mirip dengan *insert* dalam *single linked list* biasa.

3. Pop

Operasi ini berfungsi mengeluarkan elemen teratas dari *stack*.

4. Clear

Operasi ini berfungsi menghapus *stack* yang ada.

3

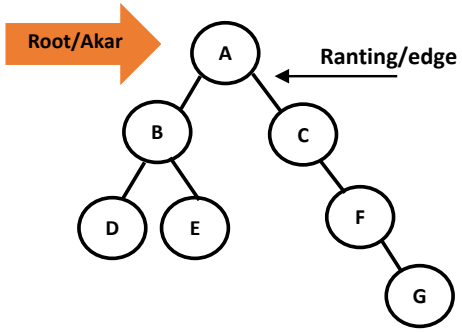
Binary Tree

Tujuan:
- Mahasiswa memahami struktur pohon biner (*binary tree*).

A. Definisi Binary Tree

Binary tree adalah struktur pohon yang di dalamnya terdapat data beserta dua buah *link* untuk kedua anak cabangnya. *Binary tree* digunakan untuk memperkecil indeks data, sehingga waktu operasional penelusuran data dalam koleksi ADT bisa lebih cepat dibandingkan struktur sekuensial, seperti *array*, *list*, ataupun *linked list*.

B. Sifat Pohon Biner



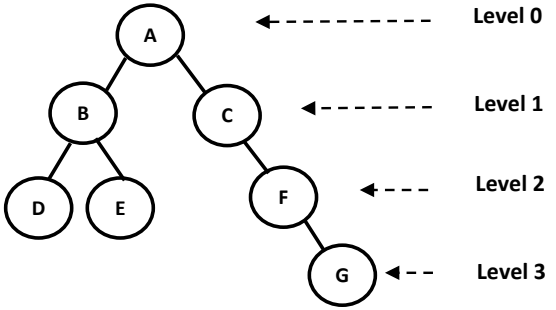
Gambar 3.1
Struktur Binary Tree

1. Simpul dan Ranting

Pada gambar pohon biner (*binary tree*) di atas, A, B, C, D, E, F, dan G disebut simpul/*node*/*vertex* dan disimbolkan dengan m. Pohon biner pada gambar tersebut mempunyai ranting (*edge*) sebanyak **m – 1**, yaitu **7 – 1 = 6** ranting.

2. Tingkatan (Level) dan Kedalaman (Depth)

Tingkatan atau level pada pohon biner dimulai dari 0, 1, 2, ... dst. Sedangkan, untuk kedalaman (*depth*) adalah level tertinggi + 1. Sebagai contoh, perhatikan Gambar 3.2.



Gambar 3.2
Tingkatan/Level Binary Tree

Keterangan:

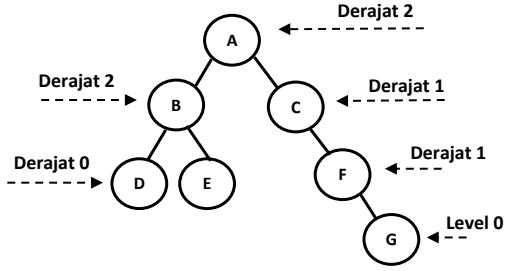
Dari Gambar 3.2 terlihat bahwa:

- simpul A → level 0
- simpul B, C → level 1
- simpul D, E, F → level 2
- simpul G → level 3

Dengan demikian, dapat diketahui bahwa level tertinggi pohon biner tersebut adalah 3. Jadi, kedalaman pohon biner tersebut = level tertinggi + 1 = 3 + 1 = 4.

3. Derajat Simpul

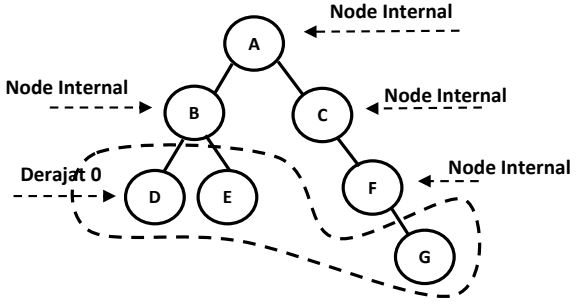
Derajat simpul adalah jumlah anak yang dimiliki oleh sebuah simpul.



Gambar 3.3
Derajat Simpul

4. Node Internal dan Node Eksternal

Node internal adalah *node* yang mempunyai anak, sedangkan *node* eksternal adalah *node* yang tidak memiliki anak atau yang biasa disebut daun (*leaf*).



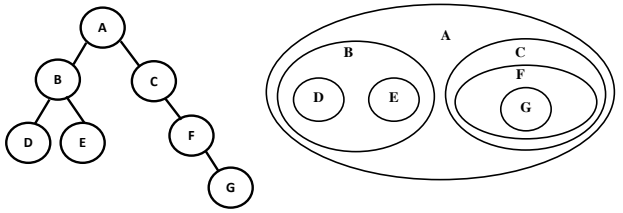
Gambar 3.4
Node Internal dan Node Eksternal

Keterangan:

Berdasarkan Gambar 3.4, maka jumlah daun pada pohon biner tersebut adalah 3, yaitu simpul D, E, dan G.

C. Notasi Pohon

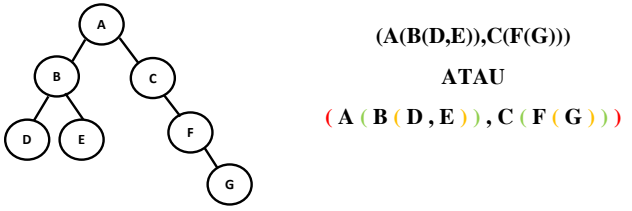
1. Diagram Venn



Gambar 3.5
Diagram Venn

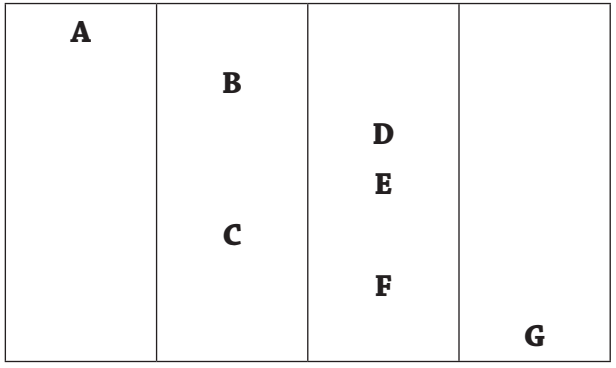
2. Notasi Kurung

Notasi kurung pada pohon biner adalah sebagai berikut:



Gambar 3.6
Notasi Kurung

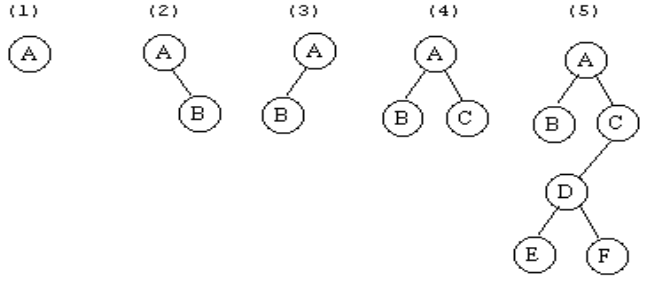
3. Notasi Tingkat



Gambar 3.7
Notasi Tingkat

D. Struktur Pohon

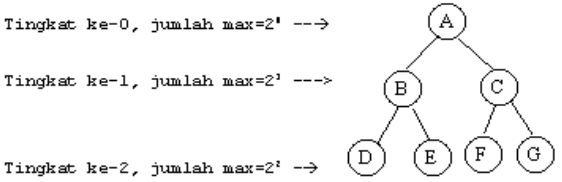
Struktur data pada pohon biner maksimal mempunyai dua anak.



Gambar 3.8
Beberapa Contoh Pohon Biner

E. Jumlah Maksimum Node

Jumlah maksimum node pada setiap tingkatan adalah 2^n.



Gambar 3.9
Jumlah Maksimum Node

F. Kamus Data

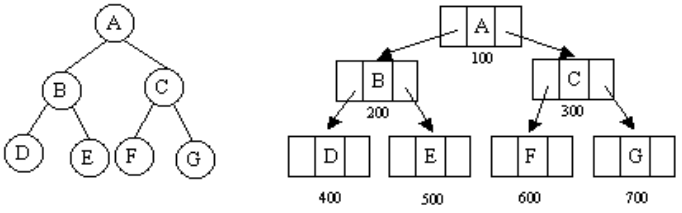
Kamus data pada pohon biner adalah sebagai berikut:

```
Kamus Data
Type BTree = record <
             Kiri : BTree
             Info : char
             Kanan : BTree >
     P : BTree
```

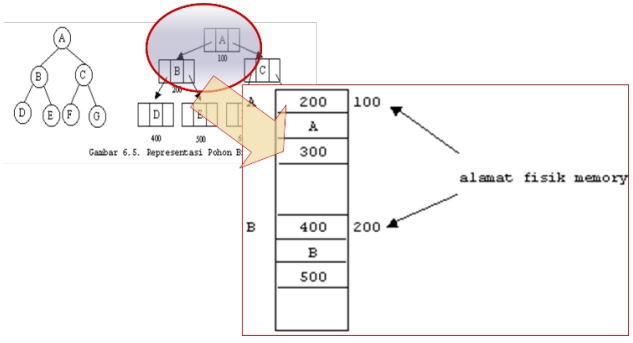
Kiri	Info	Kanan

Gambar 3.10
Kamus Data

G. Fisik Pohon Biner



Gambar 6.5. Representasi Pohon Biner



Gambar 3.11
Representasi Pohon Biner

H. Operasi Pohon Biner

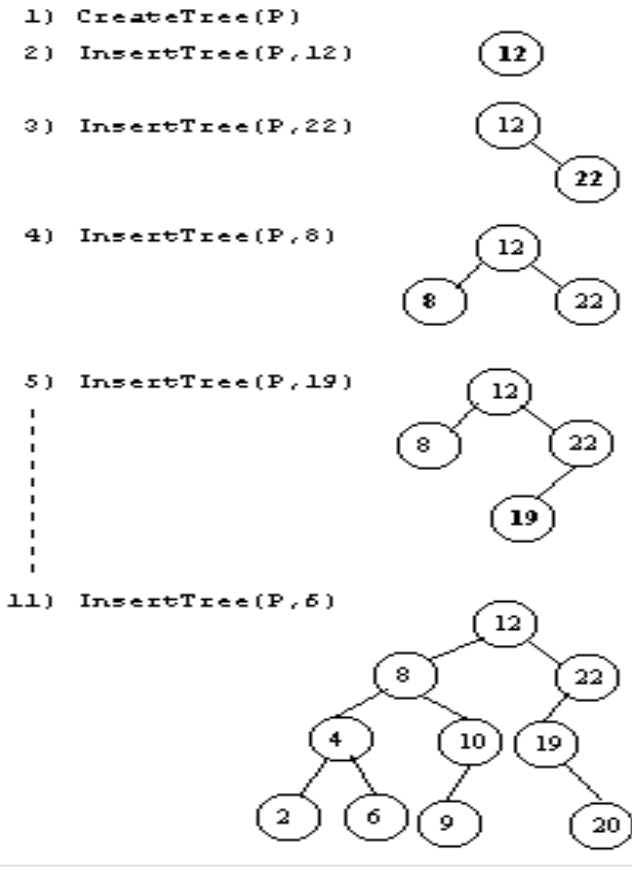
Operasi-operasi yang ada pohon biner dapat dilihat pada tabel berikut:

Nama Operasi	Fungsi
Create	Membentuk *binary tree* baru yang masih kosong.
Clear	Mengosongkan *binary tree* yang sudah ada.
Empty Function	Memeriksa apakah *binary tree* masih kosong.
Insert	Memasukkan sebuah *node* ke dalam *tree*. Ada tiga pilihan *insert*, yaitu: 1. *root*, 2. *left child*, dan 3. *right child*. Khusus *insert* sebagai *root*, *tree* harus dalam keadaan kosong.
Find	Mencari *root*, *parent*, *left child*, atau *right child* dari suatu *node* (*tree* tidak boleh kosong).
Update	Mengubah isi *node* yang ditunjuk oleh *pointer current* (*tree* tidak boleh kosong).
Retrieve	Mengetahui isi *node* yang ditunjuk oleh *pointer current* (*tree* tidak boleh kosong).
DeleteSub	Menghapus sebuah *subtree* (*node* beserta seluruh *descendant*-nya) yang ditunjuk *current*. *Tree* tidak boleh kosong. Setelah itu, *pointer current* akan berpindah ke *parent* dari *node* yang dihapus.
Characteristic	Mengetahui karakteristik dari suatu *tree*, yakni *size*, *height*, dan *average length*. *Tree* tidak boleh kosong.
Traverse	Mengunjungi seluruh *node* pada *tree*, masing-masing sekali. Hasilnya adalah urutan informasi secara linear yang tersimpan di dalam *tree*. Ada tiga cara *traverse*,yaitu *preorder*, *inorder*, dan *postorder*.

I. Pohon Biner Terurut

Contoh:

12 22 8 19 10 9 20 4 2 6



Gambar 3.12
Pohon Biner Terurut

Aturan yang digunakan untuk menyisipkan simpul adalah simpul yang lebih kecil diletakkan di sebelah kiri simpul.

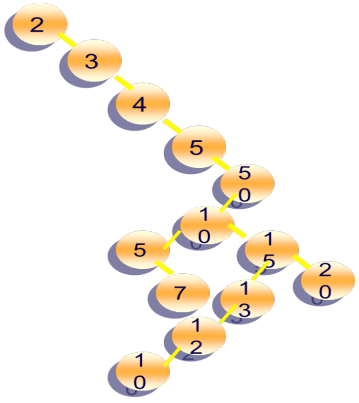
```
Procedure SisipUrutBTree(input/output P:BTree, input N:integer)
  If EmptyTree(P) then
        CreateTree(P)
        InsertTree(P,N) {untuk info(P)}
  Else  If N < info(P) then
            SisipUrutBTree(P↑.kiri,N)
        else
            SisipUrutBTree(P↑.kanan,N)
        Endif
Endif
```

Gambar 3.13
Prosedur Penyisipan Simpul

Studi Kasus:

Buatlah pohon biner terurut dari data berikut: 2, 3, 4, 5, 50, 10, 15, 13, 20, 12, 10, 5, 7.

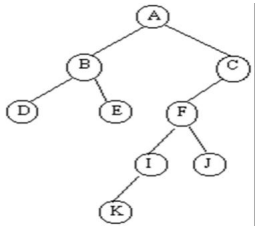
Penyelesaian:



J. Penelusuran Pohon Biner (Traverse)

Penelusuran pohon biner (*traverse*) dapat dilakukan melalui tiga metode sebagai berikut:

1. Preorder: cetak isi *node* yang dikunjungi, kunjungi *left child*, lalu kunjungi *right child*.
2. Inorder: kunjungi *left child*, cetak isi *node* yang dikunjungi, lalu kunjungi *right child*.
3. Postorder: kunjungi *left child*, kunjungi *right child*, lalu cetak isi *node* yang dikunjungi.

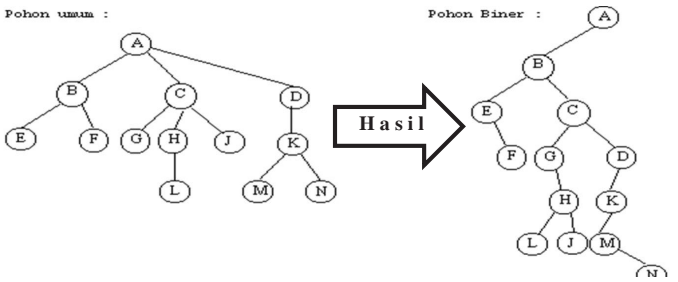


Preorder	**: ABDECFIKJ**
Postorder	**: DEBKIJFCA**
Inorder	**: DBEAKIFJC**
Levelorder	**: ABCDEFIJK**

Gambar 3.14
Penelusuran Pohon Biner

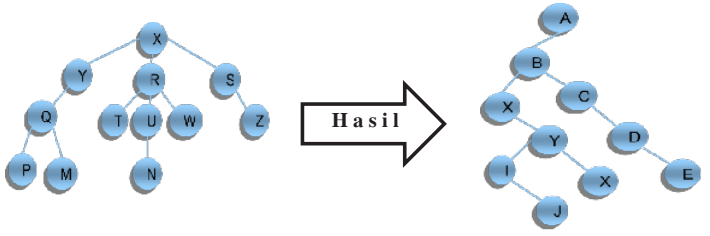
K. Konversi Pohon ke Pohon Biner

Dalam proses konversi ini, anak pertama menjadi anak kiri, anak kedua menjadi cucu kanan, anak ketiga menjadi cicit kanan, dan seterusnya.



Gambar 3.15
Pohon Biner Umum dan Pohon Biner Hasil Konversinya

Studi Kasus:

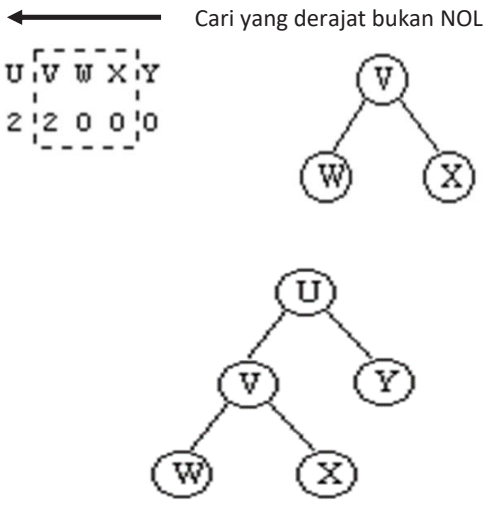


L. Pembentukan Pohon dari Hasil Traversal dan Derajat Simpul

Preorder : U V W X Y

Derajat : 2 2 0 0 0

Hasilnya :



Gambar 3.14
Pohon Biner dari Hasil Traversal dan Diketahui Derajatnya

M. Jenis-Jenis Tree

Tree terdiri dari beberapa jenis, antara lain *binary tree*, *full binary tree*, *complete binary tree*, *skewed binary tree*, *dan implementasi binary tree*.

1. Binary Tree

Binary tree adalah *tree* dengan syarat bahwa tiap *node* hanya boleh memiliki maksimal dua *subtree* dan kedua *subtree* tersebut harus terpisah. Dengan demikian, tiap *node* dalam *binary tree* hanya boleh memiliki paling banyak dua *child*.

2. Full Binary Tree

Jenis *binary tree* ini tiap *node*-nya (kecuali *leaf*) memiliki dua *child* dan tiap *subtree* harus mempunyai panjang *path* yang sama.

3. Complete Binary Tree

Complete binary tree mirip dengan *full binary tree*, namun tiap *subtree* boleh memiliki panjang *path* yang berbeda dan setiap *node* kecuali *leaf* hanya boleh memiliki dua *child*.

4. Skewed Binary Tree

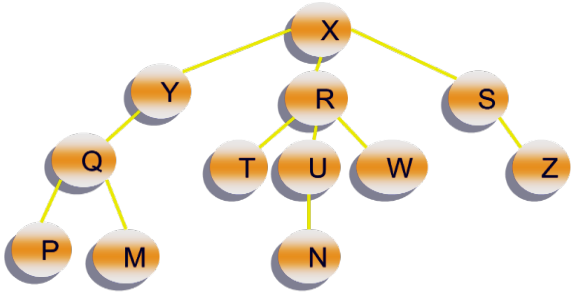
Skewed binary tree adalah *binary tree* yang semua *node*-nya (kecuali *leaf*) hanya memiliki satu *child*.

5. Implementasi Binary Tree

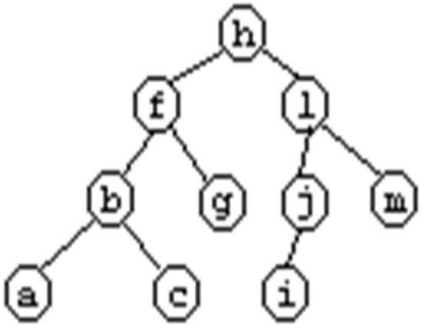
Implementasi binary tree adalah *binary tree* yang dapat diimplementasikan dalam C++ dengan menggunakan *double linked list*.

Tugas

1. Buatlah diagram Venn, notasi kurung, dan notasi tingkat dari pohon biner berikut!



2. Buatlah penelusuran pohon biner (preorder, postorder, dan inorder) berikut!



4 Linked List (Senarai)

Tujuan:
- Mahasiswa memahami struktur *linked list*.

A. Pendahuluan

Pada materi sebelumnya telah dijelaskan mengenai variabel *array* yang bersifat statis (ukuran dan urutannya sudah pasti). Selain itu, ruang memori yang dipakai oleh *array* tidak dapat dihapus apabila *array* tersebut sudah tidak digunakan lagi saat program dijalankan. Untuk memecahkan masalah tersebut, kita dapat menggunakan variabel *pointer*. Tipe data *pointer* bersifat dinamis, di mana variabel akan dialokasikan hanya saat dibutuhkan dan dapat direlokasikan kembali setelah tidak dibutuhkan.

Setiap ingin menambahkan data, Anda selalu menggunakan variabel *pointer* yang baru, akibatnya Anda akan membutuhkan banyak sekali *pointer*. Oleh karena itu, ada baiknya jika Anda hanya menggunakan satu variabel *pointer* untuk

menyimpan banyak data dengan metode yang kita sebut *linked list*. *Linked list* adalah sekumpulan elemen bertipe sama yang mempunyai keterurutan tertentu, yang setiap elemennya terdiri dari dua bagian.

B. Definisi

Linked list merupakan struktur data yang terdiri dari rantaian elemen sejenis yang saling berhubungan. Setiap elemen memiliki pendahulu dan penerusnya (kecuali elemen terakhir).

Contoh:

Struktur pada gambar di bawah mirip kereta api, di mana kepalanya seperti lokomotif, elemennya seperti gerbong kereta, dan datanya seperti penumpang/barang.

Bentuk umum:

```
typedef struct telmtlist
    {
      infotype info;
      address next;
    }elmtlist;
```

infotype → sebuah tipe terdefinisi yang menyimpan informasi sebuah elemen *list*

next → *address* dari elemen berikutnya (suksesor)

Jika L adalah *list* dan P adalah *address*, maka alamat elemen pertama *list* L dapat diacu dengan notasi sebagai berikut:

```
first(L)
```

Sebelum digunakan harus dideklarasikan terlebih dahulu dengan:

```
#define first(L) (L)
```

Elemen yang diacu oleh P dapat dikonsultasi informasinya dengan notasi sebagai berikut:

`info(P)` deklarasi `#define next(P) (P)->next`

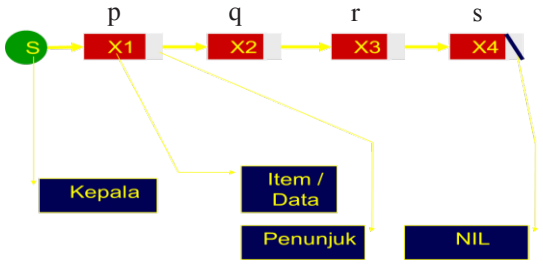
`next(P)` deklarasi `#define info(P) (P)->info`

Beberapa definisi:

1. List l adalah list kosong, jika First(L) = Nil.
2. Elemen terakhir dikenali, dengan salah satu cara yaitu karena Next(Last) = Nil.

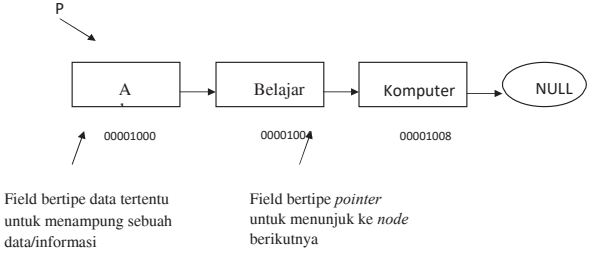
Nil adalah pengganti Null. Perubahan ini dituliskan dengan #define Nil Null.

C. Model Senarai



Gambar 4.1
Model Senarai

D. Single Linked List



Gambar 4.2
Single Linked List

Pada gambar di atas terlihat bahwa sebuah data terletak pada sebuah lokasi memori area. Tempat yang disediakan pada satu area memori tertentu untuk menyimpan data dikenal dengan sebutan *node* atau simpul. Setiap *node* memiliki *pointer* yang menunjuk ke simpul berikutnya sehingga terbentuk satu untaian, dengan demikian hanya diperlukan sebuah variabel

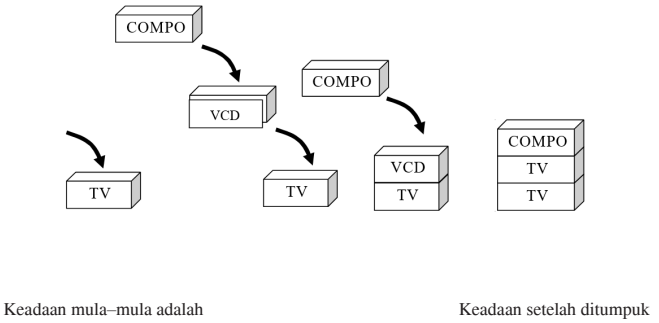
pointer. Susunan berupa untaian semacam ini disebut *single linked list* (NULL memiliki nilai khusus yang artinya tidak menunjuk ke mana-mana. Biasanya, *linked list* pada titik akhirnya akan menunjuk ke NULL).

Pembuatan *single linked list* dapat dilakukan menggunakan dua jenis metode, yaitu:

1. LIFO (*last in first out*), aplikasinya: *stack* (tumpukan).
2. FIFO (*first in first out*), aplikasinya: *queue* (antrean).

E. LIFO (Last In First Out)

LIFO adalah suatu metode pembuatan *linked list*, di mana data yang masuk paling akhir adalah data yang keluar paling awal. Dalam kehidupan sehari-hari, hal ini dapat dianalogikan dengan saat Anda menumpuk barang seperti gambar berikut



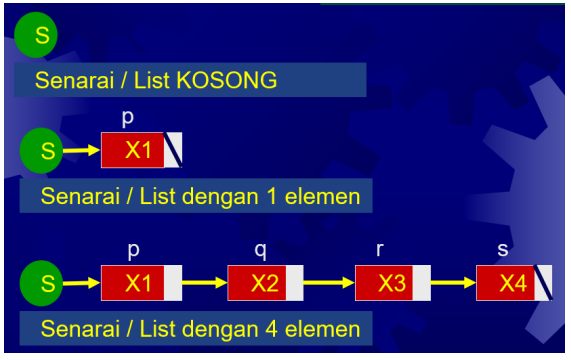
Gambar 4.3
Ilustrasi Single Linked List dengan metode LIFO

Pembuatan sebuah simpul dalam suatu *linked list* dapat diilustrasikan seperti pada Gambar 4.3. Jika *linked list* dibuat dengan metode LIFO, maka terjadi penambahan simpul di belakang, yang dikenal dengan istilah *insert*.

F. FIFO (First In First Out)

FIFO adalah suatu metode pembuatan *linked list*, di mana data yang masuk paling awal adalah data yang keluar paling awal juga. Dalam kehidupan sehari-hari, hal ini dapat dianalogikan dengan sekelompok orang yang datang mengantre (*enqueue*) hendak membeli tiket di loket. Jika *linked list* dibuat dengan metode FIFO, maka terjadi penambahan (*insert*) simpul di depan.

G. Kondisi Senarai



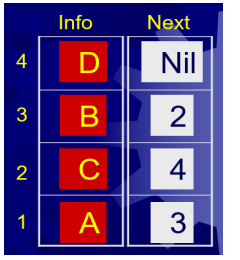
Gambar 4.4
Ilustrasi Kondisi Senarai

H. Kamus Data Senarai

Kamus Data:

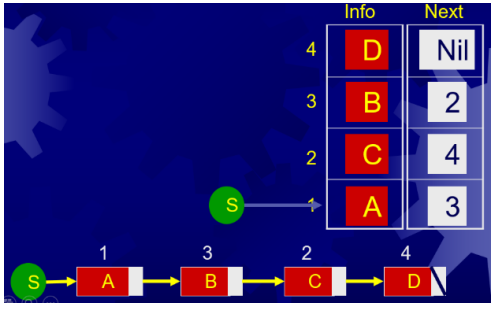
Info: array [1..4] of String

Next: array [1..4] of Integer



Gambar 4.5
Kamus Data Senarai

I. Linked List (List Berkait)

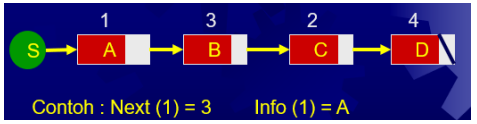


Gambar 4.6
Representasi Senarai dengan Array

J. Notasi Info dan Next

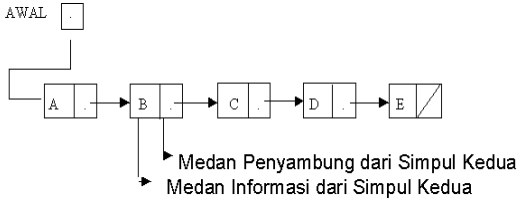
Terdapat dua notasi, yaitu:

1. Info (x): data yang ada di alamat X.
2. Next (x): alamat elemen berikut setelah X.

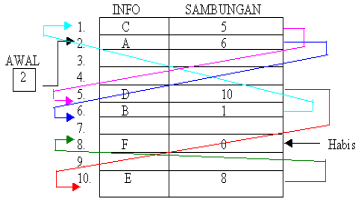


Gambar 4.7
Notasi Info dan Next

Linked juga mengandung sebuah variabel penunjuk *list*, yang biasanya diberi nama *start* (awal), yang berisi alamat dari simpul pertama dalam *list*.



K. Penyajian Linked List dalam Memory



Keterangan :

AWAL	= 2	,	Maka	INFO[2]	=	'A'
SAMBUNGAN[2]	= 6	,	Maka	INFO[6]	=	'B'
SAMBUNGAN[6]	= 1	,	Maka	INFO[1]	=	'C'
SAMBUNGAN[1]	= 5	,	Maka	INFO[5]	=	'D'
SAMBUNGAN[5]	= 10	,	Maka	INFO[10]	=	'E'
SAMBUNGAN[10]	= 8	,	Maka	INFO[8]	=	'F'
SAMBUNGAN[8]	= 0	,	Maka	Akhir Linked List		

Dari contoh diatas diperoleh untai **'ABCDEF'**

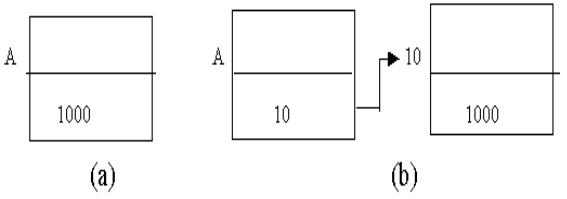
L. Konsep Pointer dan Linked List

Untuk mengolah data yang banyaknya tidak bisa ditentukan sebelumnya, disediakan satu fasilitas yang memungkinkan untuk menggunakan suatu peubah atau variabel yang disebut dengan peubah dinamis (*dynamic variable*). Peubah dinamis adalah suatu peubah yang akan dialokasikan hanya saat diperlukan, yaitu setelah program dieksekusi.

M. Perbedaan Peubah Statis dan Dinamis

Pada peubah statis, isi *memory* pada lokasi tertentu (nilai peubah) adalah data sesungguhnya yang akan diolah. Sementara, pada peubah dinamis, nilai peubah adalah alamat lokasi lain yang menyimpan data sesungguhnya. Dengan demikian, data yang sesungguhnya dapat dimasukkan secara langsung.

Dalam hal cara pemasukan data, skema prosesnya dapat diilustrasikan sebagai berikut:



N. Deklarasi Pointer

Pointer digunakan sebagai penunjuk ke suatu alamat memori. Dalam pemrograman C++, *type data pointer* dideklarasikan dengan bentuk umum sebagai berikut:

Type Data * Nama Variabel;

Type data dapat berupa sembarang, misalnya *char*, *int*, atau *float*. Sedangkan, nama variabel merupakan nama variabel *pointer*.

Contoh penggunaan *pointer* dalam program C++:

```
Void main()
{
 int x,y,*z;
 x = 75;     //nilai x = 75
 y = x;      //nilai y diambil dari nilai x
 z = &x;     //nilai z menunjuk ke alamat pointer dari nilai x
 getch();
}
```


O. Perbedaan Karakteristik Array dan Linked List

Perbedaan karakteristik *array* dan *linked list* dapat dilihat pada tabel berikut:

ARRAY	LINKED LIST
Statis	Dinamis
Penambahan / penghapusan data terbatas	Penambahan / penghapusan data tidak terbatas
Random access	Sequential access
Penghapusan array tidak mungkin	Penghapusan linked list mudah

Setiap simpul dalam suatu *linked list* terbagi menjadi dua bagian, yaitu:

1. Medan informasi, berisi informasi yang akan disimpan dan diolah.
2. Medan penyambung (*link field*), berisi alamat berikutnya. Bernilai 0, jika *link* tersebut tidak menunjuk ke data (simpul) lainnya. Penunjuk ini disebut penunjuk nol.

P. Double Linked List

Salah satu kelemahan *single linked list* adalah *pointer* (penunjuk) hanya dapat bergerak satu arah, yaitu maju/mundur atau kanan/kiri sehingga pencarian data pada *single linked list* hanya dapat bergerak dalam satu arah. Untuk mengatasi kelemahan ini, kita bisa menggunakan metode *double linked list*. *Linked list* ini dikenal dengan nama *linked list* berpointer ganda atau *double linked list*.

Q. Circular Double Linked List

Circular double linked list adalah *double linked list* yang simpul terakhirnya menunjuk ke simpul terakhirnya menunjuk ke simpul awalnya menunjuk ke simpul akhir sehingga membentuk suatu lingkaran.

R. Operasi-Operasi yang Ada dalam Linked List

Terdapat beberapa operasi yang ada dalam *linked list*, di antaranya sebagai berikut:

1. Insert

Istilah *insert* berarti menambahkan sebuah simpul baru ke dalam suatu *linked list*.

2. IsEmpty

Fungsi ini menentukan apakah *linked list* kosong atau tidak.

3. Find First

Fungsi ini mencari elemen pertama dari *linked list*.

4. Find Next

Fungsi ini mencari elemen sesudah elemen yang ditunjuk *now*.

5. Retrieve

Fungsi ini mengambil elemen yang ditunjuk oleh *now*. Elemen ini kemudian dikembalikan oleh fungsi.

6. Update

Fungsi ini mengubah elemen yang ditunjuk oleh *now* dengan isi dari sesuatu.

7. Delete Now

Fungsi ini menghapus elemen yang ditunjuk oleh *now*. Jika yang dihapus adalah elemen pertama dari *linked list* (*head*), maka *head* akan berpindah ke elemen berikutnya.

8. Delete Head

Fungsi ini menghapus elemen yang ditunjuk oleh *head*. *Head* kemudian berpindah ke elemen sesudahnya.

9. Clear

Fungsi ini menghapus *linked list* yang sudah ada. Fungsi ini wajib dilakukan jika Anda ingin mengakhiri program yang menggunakan *linked list*. Jika Anda melakukannya, data-data yang dialokasikan ke memori pada program sebelumnya akan tetap tertinggal di dalam memori.

Contoh Program:

1. Membuat Single Linked List

```
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <conio.h>

#define Nil NULL
#define info(P) P->info
#define next(P) P->next
#define First(L) (L)

typedef int InfoType;
typedef struct telmtlist *address;
typedef struct telmtlist
{
   InfoType info;
   address next;
}elmtlist;

typedef address list;

void CiptaSenarai(list *L)
{
   First(*L) = Nil;
}
```

```
list NodBaru(int m)
{
   list n;
   n = (list) malloc(sizeof(elmtlist));
   if (n != NULL)
   {
      info(n) = m;
      next(n) = Nil;
   }

   return n;
}
```



```
void SisipSenarai (list *L, list t, list p)
{
   if (p == Nil)
   {
      t->next = *L;
      *L = t;
   }
   else
   {
      t->next = p->next;
      p->next = t;
   }
}

void CetakSenarai (list L)
{
   list ps;
   for (ps=L; ps!=Nil; ps=ps->next)
   {
      cout<<" "<<info(ps)<<" -->";
   }
   cout<<" NULL"<<endl;

}
```

```
int main()
{
   list pel;
   list n;
   int i,k,nilai;

   CiptaSenarai(&pel);
   cout<<"Masukkan Banyak Data = ";
   cin>>k;
   for (i=1; i<=k; i++)
   {
      cout<<"Masukkan Data Senarai ke-"<<i<<" = ";
      cin>>nilai;
      n = NodBaru(nilai);
      SisipSenarai (&pel, n, NULL);
   }

   CetakSenarai(pel);
   return 0;
}
```

Output:

```
Masukkan Banyak Data = 7
Masukkan Data Senarai ke-1 = 6
Masukkan Data Senarai ke-2 = 2
Masukkan Data Senarai ke-3 = 6
Masukkan Data Senarai ke-4 = 1
Masukkan Data Senarai ke-5 = 7
Masukkan Data Senarai ke-6 = 5
Masukkan Data Senarai ke-7 = 8
 8 --> 5 --> 7 --> 1 --> 6 --> 2 --> 6 --> NULL
```

2. Pencarian Nilai Terkecil dan Nilai Terbesar dalam Sebuah Single Linked List

```
#include <iostream.h>
#include <stdlib.h>
#include <malloc.h>
#include <conio.h>

#define Nil NULL
#define info(P) P->info
#define next(P) P->next
#define First(L) (L)

typedef int InfoType;
typedef struct telmtlist *address;
typedef struct telmtlist
{
   InfoType info;
   address next;
}elmtlist;

typedef address list;

void CiptaSenarai(list *L)
{
   First(*L) = Nil;
}
```



```
list NodBaru(int m)
{
   list n;
   n = (list) malloc(sizeof(elmtlist));
   if (n != NULL)
   {
      info(n) = m;
      next(n) = Nil;
   }

   return n;
}

void SisipSenarai (list *L, list t, list p)
{
   if (p == Nil)
   {
      t->next = *L;
      *L = t;
   }
   else
   {
      t->next = p->next;
      p->next = t;
   }
}

void CetakSenarai (list L)
{
   list ps;
   for (ps=L; ps!=Nil; ps=ps->next)
   {
      cout<<" "<<info(ps)<<" -->";
   }
   cout<<" NULL"<<endl;

}
```

```
InfoType Max(list L)
{
   address Pmax,Pt;

   Pmax = First(L);

   if (next(Pmax) == Nil)
   {
      return (info(Pmax));
   }
   else
   {
      Pt = next(Pmax);
      while (Pt != Nil)
      {
         if (info(Pmax) < info(Pt))
         {
            Pmax = Pt;
         }
         else
         {
            Pt = next(Pt);
         }
      }

      return (info(Pmax));
   }
}

InfoType Min(list L)
{
   address Pmin,Pt;

   Pmin = First(L);

   if (next(Pmin) == Nil)
   {
      return (info(Pmin));
   }
   else
   {
      Pt = next(Pmin);
      while (Pt != Nil)
      {
         if (info(Pmin) > info(Pt))
         {
            Pmin = Pt;
         }
```



```
         else
         {
            Pt = next(Pt);
         }
      }

      return (info(Pmin));
   }
}

void main()
{
   list pel;
   list n;
   int i,k,nilai,maks,min;

   CiptaSenarai(&pel);
   cout<<"Masukkan Banyak Data = ";
   cin>>k;
   for (i=1; i<=k; i++)
   {
      cout<<"Masukkan Data Senarai ke-"<<i<<" = ";
      cin>>nilai;
      n = NodBaru(nilai);
      SisipSenarai (&pel, n, NULL);
   }


   cout<<endl;
   CetakSenarai(pel);
   maks = Max(pel);
   min = Min(pel);

   cout<<endl;
   cout<<"Nilai Terbesar : "<<maks;
   cout<<endl;
   cout<<"Nilai Terkecil : "<<min;
}
```

Output:

```
Masukkan Banyak Data = 5
Masukkan Data Senarai ke-1 = 3
Masukkan Data Senarai ke-2 = 11
Masukkan Data Senarai ke-3 = 54
Masukkan Data Senarai ke-4 = 1
Masukkan Data Senarai ke-5 = 26

 26 --> 1 --> 54 --> 11 --> 3 --> NULL

Nilai Terbesar : 54
Nilai Terkecil : 1
```

5

Queue (Antrean)

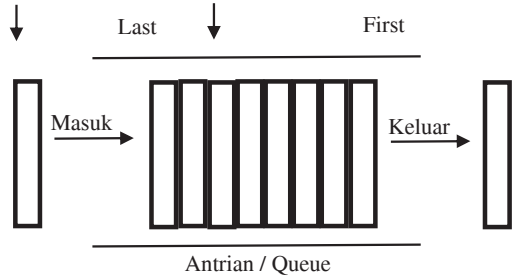
Tujuan:
- Mahasiswa memahami *queue* (antrean) dalam bahasa pemrograman.

A. Pendahuluan

Antrean atau *queue* (baca: qyu) adalah salah satu konsep struktur data yang memiliki sistem kerja, di mana yang pertama masuk maka akan menjadi yang pertama keluar (FIFO = *first in first out*), seperti halnya antrean yang terjadi dalam dunia nyata. Namun, hal ini tidak berlaku pada antrean berprioritas. Sebab, pada antrean berprioritas, urutan keluar dari sebuah antrean ditentukan berdasarkan prioritas masing-masing elemen dalam antrean untuk diproses terlebih dahulu.

Pada sebuah antrean, elemen hanya dapat ditambahkan melalui sisi belakang *queue* dan hanya dapat diambil dari sisi bagian depan *queue*. Oleh karena itu, ada dua buah elemen pada sebuah *queue*, yaitu belakang (*last* atau *rear*) sebagai penunjuk paling belakang dan depan (*first* atau *front*) sebagai

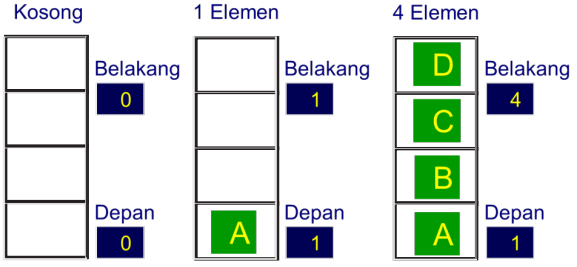
penunjuk elemen bagian depan. Seperti halnya *stack* dan *list*, *queue* juga merupakan pemikiran struktur data yang dapat direpresentasikan secara statis menggunakan *array* atau secara dinamis menggunakan *pointer*. Begitu pun dengan elemen sebuah *queue* dapat diisi dengan data sesuai kebutuhan.



Gambar 5.1
Representasi Antrean

B. Model Antrean

Berikut adalah model antrean yang direpresentasikan dalam bentuk data:



Gambar 5.2
Model Antrean

C. Operasi Dasar pada Antrean

Deklarasi secara umum:

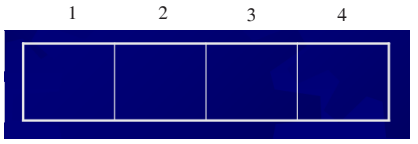
Const max_antrian = ... Type elemen = Array[1..max_antrian] of char Var antrian : elemen depan, belakang : integer	Kamus Data : Q : array [1..4] of Char Depan : Integer Belakang : Integer

Ada dua operasi dasar pada antrean (*queue*), yaitu:

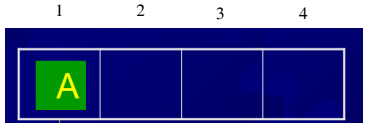
- Tambah
- Ambil

1. Penambahan Elemen

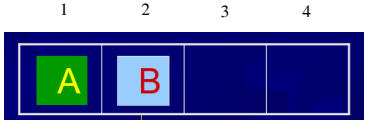
Depan/Front = 0
Belakang/Rear = 0



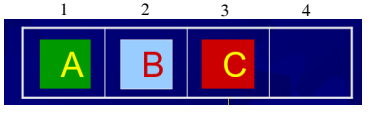
Depan/Front = 1
Belakang/Rear = 1



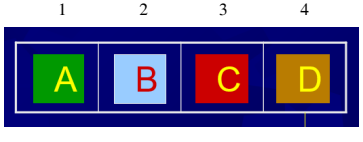
Depan/Front = 1
Belakang/Rear = 2



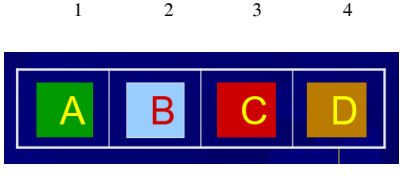
Depan/Front = 1
Belakang/Rear = 3



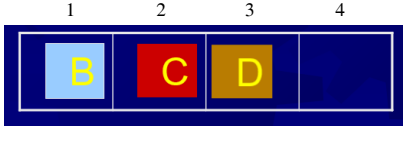
Depan/Front = 1
Belakang/Rear = 4



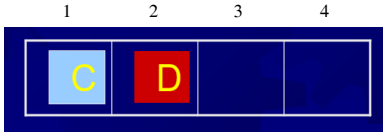
2. Pengurangan Elemen



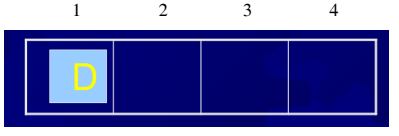
Ambil 1 elemen
Geser antrean
Depan/Front = 1
Belakang/Rear = 3



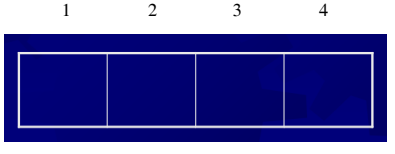
Ambil 1 elemen
Geser antrean
Depan/Front = 1
Belakang/Rear = 2



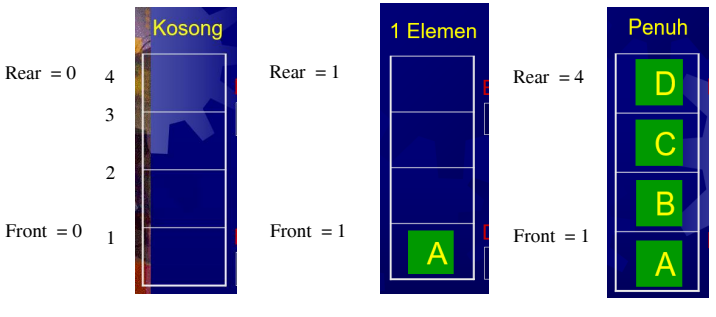
Ambil 1 elemen
Geser antrean
Depan/Front = 1
Belakang/Rear = 1



Ambil 1 elemen
Geser antrean
Depan/Front = 0
Belakang/Rear = 0



D. Kondisi Antrean

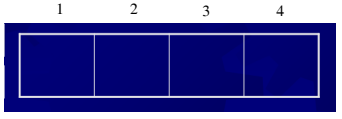


E. Antrean Sirkular atau Berputar

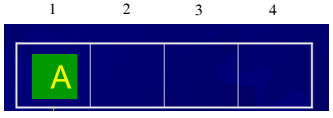
Model ini sama dengan antrean biasa, hanya saja: **tidak ada pergeseran**.

1. Penambahan Elemen

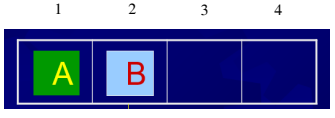
Depan/Front = 0
Belakang/Rear = 0



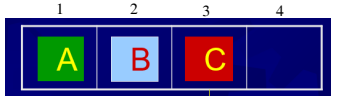
Depan/Front = 1
Belakang/Rear = 1



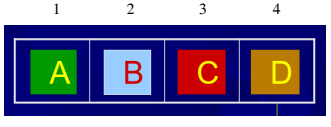
Depan/Front = 1
Belakang/Rear = 2



Depan/Front = 1
Belakang/Rear = 3

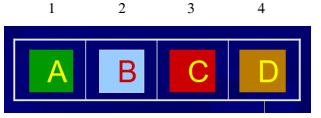


Depan/Front = 1
Belakang/Rear = 4

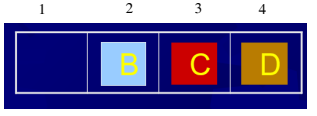


2. Pengambilan Elemen

Depan/Front = 1
Belakang/Rear = 4



Depan/Front = 2
Belakang/Rear = 4



Ambil 1 Elemen

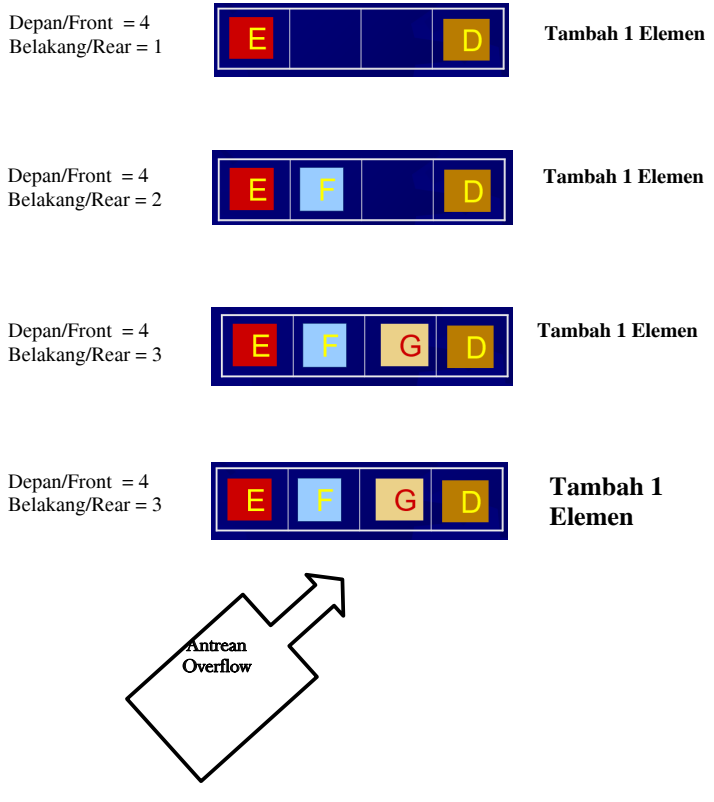
Depan/Front = 3
Belakang/Rear = 4

Ambil 1 Elemen

Depan/Front = 4
Belakang/Rear = 4

Ambil 1 Elemen

3. Penambahan Data pada Antrean Sirkular



6

Array (Larik)

Tujuan:

- Mahasiswa mampu memahami *array* (larik) satu dan dua dimensi.

A. Definisi

Array (larik) ialah penampung sejumlah data sejenis (homogen) yang menggunakan satu *identifier* (pengenal). Masing-masing elemen larik diakses menggunakan indeks (*subscript*) dari nol sampai n-1 (n menyatakan jumlah elemen larik).

Pengolahan data larik harus per elemen. Elemen larik dapat diakses langsung (acak). Tujuannya adalah untuk memanipulasi elemen keempat tidak harus melalui elemen pertama, kedua, dan ketiga. Berdasarkan banyaknya indeks, larik dibagi menjadi larik satu dimensi dan multidimensi (dua dimensi dan tiga dimensi).

B. Larik Satu Dimensi

Bentuk umum larik satu dimensi dapat dideklarasikan sebagai berikut:

```
tipe_data  nama_larik[ukuran];
```

Keterangan:

tipe_data : menyatakan jenis elemen larik (*int*, *float*, *char*, *unsigned*, dan lain-lain), tidak boleh jenis *void*.

nama_larik: menyatakan nama larik, harus memenuhi ketentuan pengenal.

ukuran : menyatakan jumlah maksimal elemen larik, normalnya lebih dari satu.

Contoh:

int nilai[4];

?	?	?	?

nilai[0] nilai[1] nilai[2] nilai[3]

Untuk memberi nilai ke elemen larik dapat dilakukan dengan cara:

1. Memberikan nilai langsung (*assignment*)

   nilai[2] ¬ 5 (nilai[2]=5;), artinya kita memberikan nilai 5 ke elemen larik yang berindeks 2;

2. Memasukkan nilai melalui papan ketik (*keyboard*) cin>> nilai[2]; atau scanf("%d", &nilai[2]);

Untuk mengakses (membaca) elemen larik, dilakukan dengan cara akses berikut:

nama_larik[indeks]; contoh: nilai[2];

atau

cout<<nilai[2]; atau printf("%d",nilai[2]);

Elemen larik juga dapat langsung diberi nilai awal saat larik dideklarasikan. Dalam hal ini, ukuran larik boleh dituliskan atau dikosongkan.

```
tipe_data nama_larik[ ]= {konstanta_1, konstanta_2,…,konstanta_n};
```

Konstanta_1, konstanta_2,..., konstanta_n adalah nilai awal elemen larik dan harus setipe.

Contoh deklarasi larik:

char huruf[] = {'a', 'b', 'c'};

'a'	'b'	'c'

huruf[0] huruf[1] huruf[2]

maksudnya huruf[0] = 'a'; huruf[1]='b'; huruf[2]='c';

Latihan program larik:

```
#include
<stdio.h>
#include
<conio.h>

 main()
 { int bil[7], i;
        clrscr();
        printf("elemen pertama ? "); scanf("%d", &bil[0]);
        bil[1] = 5;
        bil[2] = bil[1] + 20;
        for (i = 4; i < 7; i++) bil[i] = i * 10;
        bil[3] = bil[bil[1]];
        for (i = 0; i < 7; i++)
        printf("bil[%d] = %d \n", i,
        bil[i]);
 }
```

C. Menghitung Jumlah Elemen Array

Karena fungsi *sizeof()* mengembalikan jumlah *byte* yang sesuai dengan argumennya, maka operator tersebut dapat digunakan untuk menemukan jumlah elemen *array*. Misalnya:

```
int array[ ] = {26,7,82,166};
cout<<sizeof(array)/sizeof(int);
```

akan mengembalikan nilai 4, yaitu sama dengan jumlah elemen yang dimiliki *array*.

D. Melewatkan Array sebagai Argumen Fungsi

Array dapat dikirim dan dikembalikan oleh fungsi saat *array* dikirim ke dalam fungsi, serta nilai aktualnya dapat dimanipulasi.

Contoh:

```
#include <iostream.h> void ubah(int x[]);
void main()
{
int ujian[] = {90,95,78,85};
ubah(ujian);
cout<<"Elemen kedua dari array ujian adalah"<<ujian[1]<<endl;
}

void ubah(int x[])
{
x[1] = 100;
}
```

Keluarannya: elemen kedua dari *array* ujian adalah 100.

E. Larik Dua Dimensi (Matriks)

Matriks adalah sekumpulan informasi yang setiap individu elemennya diacu dengan menggunakan dua buah indeks (biasanya dikonotasikan dengan baris dan kolom).

Konsep umum larik juga berlaku pada matriks, yaitu:

1. Kumpulan elemen bertipe sama;
2. Setiap elemen data dapat diakses secara acak, jika indeksnya (baris dan kolom) sudah diketahui;
3. Merupakan struktur data statis, artinya jumlah elemennya sudah ditentukan terlebih dahulu di dalam kamus dan tidak bisa diubah selama pelaksanaan program.

Bentuk umum matriks dapat dideklarasikan sebagai berikut:

```
tipe_data nama_matriks[baris] [kolom];
```

Keterangan:

tipe_data : menyatakan jenis elemen matriks (*int*, *float*, *char*, *unsigned*, dan lain-lain), tidak oleh jenis *void*.

nama_matriks : menyatakan nama matriks, harus memenuhi ketentuan pengenal.

baris : menyatakan jumlah maksimal elemen baris matriks.

kolom : menyatakan jumlah maksimal elemen kolom matriks.

Contoh:

int B[3][4];

	0	1	2	3
0	B[0][0]	B[0][1]	B[0][2]	B[0][3]
1	B[1][0]	B[1][1]	B[1][2]	B[1][3]
2	B[2][0]	B[2][1]	B[2][2]	B[2][3]

Pemberian nilai matriks dapat dilakukan dengan cara:

1. Memberikan nilai langsung (*assignment*)
   B[2][1] ¬ 5 (B[2][1]=5;), artinya kita memberikan nilai 5 ke matriks pada baris 2 dan kolom 1;
2. Memasukkan nilai melalui papan ketik (*keyboard*) cin>> B[2][1]; atau scanf("%d", &B[2][1]);

Untuk mengakses (membaca) elemen matriks, dilakukan dengan cara akses berikut:

nama_larik[baris][kolom]; contoh B[2][1];

atau

cout<<B[2][1]; atau printf("%d",B[2][1]);

Contoh program memberi (mengisi) nilai suatu matriks:

```
#include <stdio.h>
#include <conio.h>

main()
  { int B[3][4],i,j;
            clrscr();
            for (i=0;i<=2;i++)
            {
                        for (j=0;j<=3;j++)
                          {
                                    printf("B[%d,%d] = ",i+1,j+1); scanf("%d", &B[i][j]);
                          }
            }
            printf("\n");
            for (i=0; i<=2;i++)
            {
                        for (j=0; j<=3;j++)
                          printf("%d
                          ",B[i][j]); if (j=3)
                          printf("\n");
            }
            return 0;
  }
```

F. Inisialisasi Matriks

Misalkan ada matriks sebagai berikut:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Maka, matriks tersebut dapat dinyatakan dengan perintah:

```
int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
```


G. Menjumlahkan Dua Buah Matriks

Menjumlahkan dua buah matriks A dan B akan menghasilkan matriks C atau A + B = C hanya dapat dilakukan jika ukuran matriks A dan B sama, serta kedua matriks sudah terdefinisi harganya (elemennya sudah terisi). Hasilnya, matriks C juga berukuran sama dengan matriks A dan B. Penjumlahan matriks A dan B didefinisikan sebagai berikut:

C[i,j] = A[i,j] + B[i,j] untuk setiap i dan j

Contoh:

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & 2 & 4 \\ 1 & 4 & 3 \\ 2 & 5 & 2 \end{bmatrix}$$

Adapun prosedur untuk menjumlahkan matriks adalah sebagai berikut:

```
Kamus

const min_baris : integer = 1 const
maks_baris  :  integer  =  3  const
min_kolom  :  integer  =  1  const
maks_kolom : integer = 3

type matriks : matrix[min_baris..maks_baris, min_kolom..maks_kolom] of integer
```

atau

```
Kamus

const int maks_baris = 3 const
int maks_kolom = 3

int matriks [maks_baris] [maks_kolom]
```

```
procedure Jumlah_matriks(input A : matriks, input B : matriks, output C : matriks)

{menjumlahkan matriks A dan B, yaitu A + B = C

 K.Awal : matriks A dan B sudah terdefinisi elemen-elemennya

 K.Akhir : elemen matriks C berisi penjumlahan A dan B}
```

```
Kamus

  i : integer {indeks baris}

  j : integer {indeks kolom}
```

```
Algoritma

  for i ¬ min_baris to mak_baris do

      for j ¬ min_kolom to mak_kolom do C[i,j]

          ¬ A[i,j] + B[i,j]
```

```
      endfor

  endfor
```

```
void Jumlah_matriks(input A : matriks, input B : matriks, output C : matriks)
{menjumlahkan matriks A dan B, yaitu A + B = C
 K.Awal : matriks A dan B sudah terdefinisi elemen-elemennya
 K.Akhir : elemen matriks C berisi penjumlahan A dan B}

Kamus
  int i {indeks baris}
  int j {indeks kolom}

Algoritma
  for (i ¬ min_baris; i £ mak_baris; i++)
      for (j ¬ min_kolom; j £ mak_kolom; j++) C[i,j]
          ¬ A[i,j] + B[i,j]
      endfor
  endfor
```


7

Hashing Table (Tabel Hash)

Tujuan:
- Mahasiswa mampu membuat dan mendeklarasikan struktur algoritma tabel hash.
- Mahasiswa mampu menerapkan dan mengimplementasikan struktur algoritma tabel hash.

A. Dasar Teori

Tabel hash merupakan salah satu struktur data yang digunakan dalam penyimpanan data sementara. Tujuan penggunaan tabel hash adalah mempercepat pencarian kembali dari banyak data yang disimpan. Tabel hash menggunakan suatu teknik penyimpanan sehingga waktu yang dibutuhkan untuk penambahan data (*insertions*), penghapusan data (*deletions*), dan pencarian data (*searching*) relatif sama dibanding struktur data atau algoritma yang lain.

Dari topik yang sebelumnya sudah dipelajari dapat diketahui bahwa beberapa struktur data dan algoritma pencarian (*searching*) memiliki kelebihan serta kekurangan masing-masing. Begitu pula dengan tabel hash yang memiliki kelebihan dan kekurangan. Adapun kelebihan tabel hash antara lain:

- relatif lebih cepat; dan
- kecepatan dalam *insertions*, *deletions*, maupun *searching* relatif sama.

Tabel hash menggunakan memori penyimpanan utama berbentuk *array* dengan tambahan algoritma untuk mempercepat pemrosesan data. Pada intinya, tabel hash merupakan penyimpanan data menggunakan *key value* yang didapat dari nilai data itu sendiri. Dengan *key value* tersebut, diperoleh *hash value*. Jadi, *hash function* merupakan suatu fungsi sederhana untuk mendapatkan *hash value* dari *key value* suatu data. Beberapa hal yang perlu diperhatikan untuk membuat *hash function* adalah:

- ukuran *array*/*table size*(m),
- *key value*/nilai yang didapat dari data(k), dan
- *hash value*/*hash index*/indeks yang dituju(h).

Berikut contoh penggunaan tabel hash dengan *hash function* sederhana yaitu memodulus *key value* dengan ukuran *array*:

$$H(k) = k \text{ MOD } m$$

Contoh:

Misal kita memiliki *array* dengan ukuran 13, maka *hash function*-nya adalah h = k (mod 13).

Dengan *hash function* tersebut, maka diperoleh:

k	H
7	7
13	0
25	12
27	1
39	0

Perhatikan *range* dari h untuk sembarang nilai k.

Dengan demikian, data 7 akan disimpan pada index 7, data 13 akan disimpan pada index 0, dan seterusnya.

Untuk mencari kembali suatu data, kita hanya perlu menggunakan *hash function* yang sama sehingga mendapatkan *hash index* yang sama pula.

Misal: mencari data 25 → h = 25 (mod 13) = 12

Namun, dalam penerapannya seperti contoh di atas terdapat tabrakan (*collision*) pada k = 13 dan k = 39. *Collision* berarti ada lebih dari satu data yang memiliki *hash index* yang sama. Padahal sebagaimana kita ketahui, satu alamat/satu *index array* hanya dapat menyimpan satu data.

B. Cara-Cara Mengatasi Collision

Untuk meminimalkan *collision*, gunakan *hash function* yang dapat mencapai seluruh indeks/alamat. Dalam contoh

di atas, gunakan m untuk me-modulo k. Perhatikan apabila kita menggunakan angka m untuk me-modulo k, maka pada indeks yang lebih besar dari dan sama dengan m di *hash table* tidak akan pernah terisi (memori yang terpakai semakin kecil), kemungkinan terjadi *collision* juga semakin besar.

Karena memori yang terbatas dan untuk masukan data yang belum diketahui, tentu *collision* tidak dapat dihindari. Berikut beberapa cara untuk mengatasi *collision*.

1. Closed Hashing (Open Addressing)

Close hashing menyelesaikan *collision* dengan menggunakan memori yang masih ada tanpa menggunakan memori di luar *array* yang dipakai. *Closed hashing* mencari alamat lain apabila alamat yang akan dituju sudah terisi oleh data. Terdapat tiga cara untuk mencari alamat lain tersebut, yaitu *linear probing*, *quadratic probing*, dan *double hashing*.

a. Linear Probing (Metode Pembagian)

Apabila telah terisi, l*inear probing* mencari alamat lain dengan bergeser satu indeks dari alamat sebelumnya hingga ditemukan alamat yang belum terisi data, dengan rumus sebagai berikut:

$$(h+1) \bmod m$$

Contoh:

Disediakan nomor mahasiswa terdiri dari lima digit. Selain itu, disediakan pula larik (l) dari 100 buah alamat yang masing-masing alamat terdiri dari dua karakter, yaitu 00 ... 99. Nomor mahasiswa yang diketahui adalah 10347, 87492, 34212, dan 88688. Untuk menentukan alamat dari keempat nomor mahasiswa tersebut, kita pilih bilangan prima yang mendekati 99 yaitu 97, sehingga m = 97. Selanjutnya, masukkan hit di atas ke dalam rumus berikut:

$$H(k) \bmod m + 1$$

H(K) 1 = 10347 MOD 97 + 1 → 66

H(K) 2 = 87492 MOD 97 + 1 → 96

H(K) 3 = 34212 MOD 97 + 1 → 69

H(K) 4 = 88688 MOD 97 + 1 → 31

H(K) 5 = 88588 MOD 97 + 1 → 28

H(K) 6 = 87578 MOD 97 + 1 → 85

b. Quadratic Probing (Metode Midsquare/Nilai Tengah)

Quadratic probing mencari alamat baru untuk ditempati dengan proses perhitungan kuadratik yang lebih kompleks. Tidak ada formula baku pada *quadratic probing* ini. Anda dapat menentukan sendiri formula yang akan digunakan.

Berikut adalah salah satu contoh formula *quadratic probing* untuk mencari alamat baru:

h,(h+i2)mod m,(h-i2)mod m, … ,(h+((m-1)/2)2)mod m, (h-((m-1)/2)2)mod m

dengan i = 1,2,3,4, … , ((m-1)/2)

Maksud formula di atas adalah jika alamat h telah terisi, maka alamat lain yang digunakan adalah (h+1)mod m. Jika alamat telah terisi, gunakan alamat (h-1)mod m. Jika alamat telah terisi, gunakan alamat (h+4)mod m. Jika alamat telah terisi, gunakan alamat (h-4)mod m, dan seterusnya. Jadi, jika m = 23, maka nilai maksimal i adalah ((23-1)/2) = 11.

Contoh:

Pada metode ini, kunci yang diketahui dikuadratkan.

K	10347	87492
K2	01 07 06 04 09	76 54 85 00 64
H(k)	**06**	**85**

c. Double Hashing (Metode Penjumlahan Digit)

Sesuai dengan namanya, alamat baru untuk menyimpan data yang belum dapat masuk ke dalam tabel diperoleh dengan menggunakan *hash function* lagi. *Hash function* kedua yang digunakan setelah alamat yang dihasilkan oleh *hash function* awal telah terisi tentu saja berbeda dengan *hash function* awal itu sendiri.

Metode *closed hashing* memiliki kelemahan yaitu ukuran *array* yang disediakan harus lebih besar dari jumlah data. Selain itu, dibutuhkan memori yang lebih besar untuk meminimalkan *collision*.

Contoh:

10347 = 01 + 03 + 47 = 51

87492 = 08 + 74 + 92 = 174 = 01 + 74 = 75

Intinya:

Pada metode ini, kunci yang diketahui bisa dipecah menjadi beberapa kelompok. Pemecahan dan penjumlahan terus dilakukan jika keseluruhan kelompok yang ada masih lebih besar dari banyaknya alamat yang akan dipakai.

2. Open Hashing (Separate Chaining)

Pada dasarnya, *open hashing* atau s*eparate chaining* membuat tabel yang digunakan untuk proses *hashing* menjadi sebuah *array of pointer* yang masing-masing pointernya diikuti oleh sebuah *linked lis*t, dengan *chain* (mata rantai) 1 terletak pada *array of pointer*, sedangkan *chain* 2 dan seterusnya berhubungan dengan *chain* 1 secara memanjang. Kelemahan

metode ini yaitu terjadi *linked list* yang panjang apabila data menumpuk pada satu atau sedikit indeks.

8

Pengurutan (Sorting)

Tujuan:
- Mahasiswa mampu menunjukkan beberapa algoritma dalam pengurutan.
- Mahasiswa mampu menunjukkan bahwa pengurutan merupakan suatu persoalan yang bisa diselesaikan dengan sejumlah algoritma yang berbeda satu sama lain lengkap dengan kelebihan dan kekurangannya.
- Mahasiswa dapat memilih algoritma yang paling sesuai untuk menyelesaikan suatu permasalahan pemrograman

A. Definisi

Pengurutan data (*sorting*) didefinisikan sebagai suatu proses untuk menyusun kembali himpunan objek menggunakan aturan tertentu. Menurut Microsoft Book-shelf, algoritma pengurutan adalah algoritma untuk meletakkan kumpulan elemen data ke dalam urutan tertentu berdasarkan satu atau beberapa kunci dalam tiap-tiap elemen.

Ada dua macam urutan yang biasa digunakan dalam proses pengurutan, yaitu:

1. Urut naik (*ascending*), merupakan urutan dari data yang mempunyai nilai terkecil sampai terbesar.
2. Urut turun (*descending*), merupakan urutan data yang mempunyai nilai terbesar sampai terkecil.

Contoh:

Data bilangan 5, 2, 6, dan 4 dapat diurutkan naik menjadi 2, 4, 5, 6 atau diurutkan turun menjadi 6, 5, 4, 2. Pada data yang bertipe *char*, nilai data dikatakan lebih kecil atau lebih besar dari yang lain didasarkan pada urutan relatif (*collating sequence*) seperti dinyatakan dalam tabel ASCII (Lampiran).

Keuntungan dari data yang sudah dalam keadaan terurutkan, antara lain:

1. Data mudah dicari (misalnya dalam buku telepon atau kamus bahasa) serta mudah untuk dibetulkan, dihapus, disisipi, atau digabungkan. Dalam keadaan terurutkan, kita mudah melakukan pengecekan adakah ada data yang hilang.
2. Melakukan kompilasi program komputer jika tabel-tabel simbol harus dibentuk untuk mempercepat proses pencarian data yang harus dilakukan berulang kali.

Data yang diurutkan sangat bervariasi, baik dalam hal jumlah data maupun jenis data yang akan diurutkan. Tidak ada algoritma terbaik untuk setiap situasi yang kita hadapi. Bahkan, cukup sulit untuk menentukan algoritma mana yang

paling baik untuk situasi tertentu, karena ada beberapa faktor yang memengaruhi efektivitas algoritma pengurutan.

Beberapa faktor yang berpengaruh terhadap efektivitas suatu algoritma pengurutan, antara lain:

1. banyaknya data yang diurutkan;
2. kapasitas pengingat apakah mampu menyimpan semua data yang kita miliki; dan
3. tempat penyimpanan data, misalnya piringan, pita, kartu, atau media penyimpan yang lain.

Pemilihan algoritma sangat ditentukan oleh struktur data yang digunakan. Metode pengurutan yang digunakan dapat diklasifikasikan menjadi dua kategori, yaitu:

1. Pengurutan internal, merupakan pengurutan dengan menggunakan larik (*array*). Larik tersimpan dalam memori utama komputer.
2. Pengurutan eksternal, merupakan pengurutan dengan menggunakan berkas (*sequential access file*). Berkas tersimpan dalam pengingat luar, misalnya cakram atau pita magnetis.

Untuk menggambarkan pengurutan dengan larik, bisa kita bayangkan semua kartu terletak di hadapan kita sehingga semua kartu tersebut terlihat dengan jelas nomornya. Pada penyusunan kartu sebagai sebuah berkas, kita bayangkan semua kartu kita tumpuk sehingga hanya kartu bagian atas yang bisa kita lihat nomornya.

B. Klasifikasi Pengurutan

Algoritma-algoritma pengurutan (*sorting*) dapat dikelompokkan berdasarkan teknik yang digunakan dalam algoritma tersebut. Adapun pengelompokannya adalah sebagai berikut.

1. Brute Force

Brute force adalah sebuah pendekatan yang lempang (*straightforward*) untuk memecahkan suatu masalah (*problem statement*) dan definisi konsep yang dilibatkan. Algoritma *brute force* memecahkan masalah dengan sangat sederhana, langsung, dan dengan cara yang jelas (*obvious way*).

Adapun jenis-jenis algoritma yang termasuk dalam algoritma *brute force*, antara lain:

a. Bubble Sort

Bubble sort merupakan algoritma pengurutan paling tua dengan metode pengurutan paling sederhana. Pengurutan dilakukan dengan membandingkan masing-masing *item* dalam suatu *list* secara berpasangan, menukar *item* jika diperlukan, dan mengulanginya sampai akhir *list* secara berurutan, sehingga tidak ada lagi *item* yang dapat ditukar.

b. Bidirectional Bubble Sort

Bidirectional bubble sort merupakan variasi dari algoritma *bubble sort*, di mana *item* dalam suatu *list* dibandingkan secara berpasangan, menukarnya jika diperlukan, dan mempunyai alternatif melalui *list* secara berurutan dari awal sampai akhir kemudian dari akhir sampai awal lagi hingga tidak ada pertukaran (*swap*) yang dapat dilakukan.

2. Divide and Conquer

Divide and conquer adalah metode pemecahan masalah yang bekerja dengan membagi masalah (problem) menjadi beberapa submasalah (subproblem) yang lebih kecil, kemudian menyelesaikan masing-masing submasalah secara independen dan akhirnya menggabung solusi masing-masing submasalah sehingga menjadi solusi masalah semula.

Adapun yang termasuk dalam algoritma *divide and conquer* adalah sebagai berikut:

a. Merge Sort

Merge sort adalah sebuah algoritma pengurutan yang membagi *item* yang akan diurutkan menjadi dua bagian, dan secara rekursif mengurutkan masing-masing bagian tersebut lalu menggabungkannya sampai berakhir.

b. Insertion Sort

Insertion sort adalah sebuah algoritma pengurutan dengan mengambil *item* dan memasukkannya ke dalam struktur data yang secara berulang kali dalam susunan yang tepat.

c. Quick Sort

Quick sort adalah sebuah algoritma pengurutan yang mengambil sebuah elemen dalam larik sebagai pivot, mempartisi elemen sisanya menjadi elemen yang lebih besar dan elemen yang lebih kecil daripada pivot, serta secara rekursif mengurutkan hasil dari partisi tersebut.

d. Selection Sort

Selection sort adalah sebuah algoritma pengurutan yang secara berulang mencari *item* yang belum terurut dan mencari paling sedikit satu untuk dimasukkan ke lokasi akhir.

e. Shell Sort

Shell sort merupakan algoritma yang sejenis dengan *insertion sort*, di mana pada setiap nilai i dalam n/i *item* diurutkan. Pada setiap pergantian nilai, i dikurangi sampai 1 sebagai nilai terakhir.

f. Radix Sort

Secara kompleksitas waktu, *radix sort* termasuk algoritma *divide and conquer*. Namun, dari segi algoritma untuk melakukan proses pengurutan, *radix sort* tidak termasuk dalam *divide and conquer*. *Radix sort* merupakan sebuah algoritma pengurutan yang mengatur pengurutan nilai tanpa melakukan beberapa perbandingan pada data yang dimasukkan.

C. Deklarasi Larik

Pada pengurutan larik, ada beberapa aspek yang perlu dipertimbangkan, antara lain aspek menyangkut kapasitas pengingat yang ada dan aspek waktu, yaitu waktu yang diperlukan untuk melakukan permutasi sehingga semua elemen akhirnya menjadi terurutkan. Deklarasi larik yang digunakan adalah larik dimensi satu (vektor) dengan elemennya bertipe integer.

```
#define Max 100;

int Data[Max];
```

Pada deklarasi di atas, Max adalah banyaknya elemen vektor. Anda bisa mengubah nilai konstanta Max sesuai kebutuhan. Indeks larik dimulai dari 0. Data yang sebenarnya disimpan mulai dari indeks 0. Selain deklarasi di atas, proses yang juga selalu digunakan pada algoritma pengurutan adalah proses menukarkan elemen. Berikut adalah satu contoh prosedur sederhana untuk menukarkan nilai dua buah elemen A dan B.

```
void Tukar (int *a, int *b)
{
int temp; temp = *a;
*a = *b;
*b = temp;
}
```

Program 10.1
Prosedur Penukaran Dua Bilangan

D. Metode Penyisipan Langsung (Straight Insertion Sort)

Proses pengurutan dengan metode penyisipan langsung dapat dijelaskan sebagai berikut. Data dicek satu per satu mulai dari yang kedua sampai dengan yang terakhir. Apabila ditemukan data yang lebih kecil daripada data sebelumnya, maka data tersebut disisipkan pada posisi yang sesuai. Akan lebih mudah apabila membayangkan pengurutan kartu. Pertama-tama, Anda meletakkan kartu-kartu tersebut di atas meja, kemudian melihatnya dari kiri ke kanan. Apabila kartu di sebelah kanan lebih kecil daripada kartu di sebelah kiri, maka ambillah kartu tersebut dan sisipkan di tempat yang sesuai.

Algoritma penyisipan langsung dapat dituliskan sebagai berikut:

1 i ← 1
2 selama (i < N) kerjakan baris 3 sampai dengan 9
3 x ← Data[i]

4 j ← i – 1
5 selama (x < Data[j]) kerjakan baris 6 dan 7
6 Data[j + 1] ← Data[j]
7 j ← j – 1
8 Data[j+1] ← x
9 i ← i + 1

Untuk memperjelas langkah-langkah algoritma penyisipan langsung dapat dilihat pada Tabel 10.1. Proses pengurutan pada tabel 10.1 dapat dijelaskan sebagai berikut:

- Pada saat i = 1, x sama dengan Data[1] = 35 dan j = 0. Karena Data[0] = 12 dan 35 > 12, maka proses dilanjutkan untuk i = 2.
- Pada saat i = 2, x = Data[2] = 9 dan j = 1. Karena Data[1] = 35 dan 9 < 35, maka dilakukan pergeseran sampai ditemukan data yang lebih kecil dari 9. Hasil pergeseran ini, Data[1] = 12 dan Data[2] = 35 sedangkan Data[0] = x = 9.
- Pada saat i = 3, x = Data[3] = 11 dan j = 3. Karena Data[2] = 35 dan 11 < 35, maka dilakukan pergeseran sampai ditemukan data yang lebih kecil dari 11. Hasil pergeseran ini, Data[2] = 12 dan Data[3] = 35 sedangkan Data[1] = x = 11.
- Dan, seterusnya.

Iterasi	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
Awal	12	35	9	11	3	17	23	15	31	20
i=1	12	**35**	9	11	3	17	23	15	31	20
i=2	12	35	**9**	11	3	17	23	15	31	20
i=3	9	12	35	**11**	3	17	23	15	31	20
i=4	9	11	12	35	**3**	17	23	15	31	20
i=5	3	9	11	12	35	**17**	23	15	31	20
i=6	3	9	11	12	17	35	**23**	15	31	20
i=7	3	9	11	12	17	23	35	**15**	31	20
i=8	3	9	11	12	15	17	23	35	**31**	20
i=9	3	9	11	12	15	17	23	31	35	**20**
Akhir	3	9	11	12	15	17	20	23	31	35

Tabel 10.1 Proses Pengurutan dengan Metode Penyisipan Langsung

Berikut adalah prosedur yang menggunakan metode penyisipan langsung:

```
void StraighInsertSort()
{
int i, j, x;
for(i=1; i<Max; i++){ x = Data[i];
j = i - 1;
while (x < Data[j]){

Data[j+1] = Data[j]; j--;
}
Data[j+1] = x;
}
}
```

Program 10.2
Prosedur Pengurutan dengan Metode Penyisipan Langsung

Dari algoritma dan prosedur di atas, jumlah perbandingan (=C) minimum, rata-rata, dan maksimum pada metode penyisipan langsung adalah:

$$\mathbf{C_{min} = N - 1}$$

$$\mathbf{C_{rata\text{-}rata} = (N^2 + N + 2)/4}$$

$$\mathbf{C_{max} = (N^2 + N - 2)/2}$$

Jumlah perbandingan minimum terjadi jika data sudah dalam keadaan urut. Sebaliknya, jumlah perbandingan maksimum terjadi jika data dalam keadaan urut terbalik.

Cara menghitung C_{min} adalah dengan melihat kalang paling luar yaitu i, pengulangan ini dimulai dari 1 sampai dengan N-1 atau sama dengan N-1.

Cara menghitung C_{max} yaitu dengan menganggap selalu terjadi pergeseran. Kalang dalam (*while*) di atas akan melakukan pengulangan dari 0 sampai dengan i. Apabila hal ini dilakukan sebanyak N-1 kali, maka Cmax adalah jumlah dari deret aritmetika 2, 3, 4, ..., N atau (N – 1) / 2 . (2 + N).

Cara menghitung $C_{rata\text{-}rata}$ adalah dengan menjumlahkan C_{min} dan C_{max}, lalu dibagi dengan 2.

Jumlah penggeseran (=M) minimum, rata-rata, dan maksimum untuk metode penyisipan langsung adalah:

$$M_{min} = 2(N - 1)$$

$$M_{rata\text{-}rata} = (N^2 + 7N - 8)/4$$

$$M_{max} = (N^2 + 3N - 4)/2$$

E. Metode Penyisipan Biner (Binary Insertion Sort)

Metode ini merupakan pengembangan dari metode penyisipan langsung. Dengan cara penyisipan langsung, perbandingan selalu dimulai dari elemen pertama (data ke-0). Sehingga, untuk menyisipkan elemen ke-i, kita harus melakukan perbandingan sebanyak i-1 kali. Ide dari metode ini didasarkan pada kenyataan bahwa saat menggeser data ke-i, data ke-0 s/d i-1 sebenarnya sudah dalam keadaan terurut. Sebagai contoh pada Tabel 10.1 di atas, saat i = 4, data ke-0 s/d 3 sudah dalam keadaan urut, yaitu 3, 9, 12, 35.

Dengan demikian, posisi dari data ke-i sebenarnya dapat ditentukan dengan pencarian biner. Misalnya pada saat i = 7, data yang akan dipindah adalah 15 sedangkan data di sebelah kiri 15 adalah sebagai berikut:

3	9	11	12	17	23	35	**15**

Pertama-tama, dicari data pada posisi paling tengah di antara data di atas. Data yang terletak di tengah adalah data ke-3, yaitu 12. Karena 12 < 15, berarti 15 harus disisipkan di sebelah kanan 12. Oleh karena itu, proses pencarian dilanjutkan lagi untuk data berikut:

17	23	35

Dari hasil ini, didapatkan data tengahnya adalah 23. Karena 15 < 23, berarti 15 harus disisipkan di sebelah kiri 23. Proses dilanjutkan kembali untuk data berikut:

17

Karena 17 > 15, berarti 15 harus disisipkan di sebelah kiri 17. Algoritma penyisipan biner dapat dituliskan sebagai berikut:

```
1   i ← 1
2   selama (i < N) kerjakan baris 3 sampai dengan 14
3   x ← Data[i]
4   l ← 0
5   r ← i – 1
6   selama (l<=r) kerjakan baris 7 dan 8
7   m ← (l + r) / 2
8   jika (x < Data[m]) maka r ← m – 1, jika tidak l ← m + 1
9   j ← i – 1
10  selama ( j >=l) kerjakan baris 11 dan 12
11  Data[j+1] ← Data[j] 12     j ← j – 1
13  Data[l] ← x
14  I ← i + 1
```

Berikut merupakan prosedur yang menggunakan metode penyisipan biner:

```
void BinaryInsertSort()
{
int i, j, l, r, m, x; for (i=1; i<Max; i++){
x = Data[i]; l = 0;
r = i - 1; while(l <= r){
m = (l + r) / 2; if(x < Data[m])
r = m - 1; else
l = m + 1;
}
for(j=i-1; j>=l; j--) Data[j+1] = Data[j];
Data[l]=x;
}
}
```

Program 10.3
Prosedur Pengurutan dengan Metode Penyisipan Biner

Dari algoritma dan prosedur di atas, jumlah perbandingan (=C) untuk metode penyisipan biner adalah:

$$C = \Sigma\ [^2\log(i)]$$

Sementara, jumlah penggeseran (=M) untuk metode penyisipan biner sama dengan metode penyisipan langsung.

F. Metode Seleksi (Selection Sort)

Metode seleksi melakukan pengurutan dengan cara mencari data yang terkecil kemudian menukarkannya dengan data yang digunakan sebagai acuan atau sering dinamakan pivot.

Proses pengurutan dengan metode seleksi dapat dijelaskan sebagai berikut:

1. Langkah pertama dicari data terkecil dari data pertama sampai terakhir. Kemudian, data terkecil ditukar dengan data pertama. Dengan demikian, data pertama sekarang mempunyai nilai paling kecil dibanding data yang lain.
2. Langkah kedua, data terkecil kita cari mulai dari data kedua sampai terakhir. Data terkecil yang kita peroleh ditukar dengan data kedua dan demikian seterusnya sampai semua elemen dalam keadaan terurutkan.

Algoritma seleksi dapat dituliskan sebagai berikut:

1 i ← 0
2 selama (i < N-1) kerjakan baris 3 sampai dengan 9
3 k ← i
4 j ← i + 1
5 Selama (j < N) kerjakan baris 6 dan 7
6 Jika (Data[k] > Data[j]) maka k ← j
7 j ← j + 1
8 Tukar Data[i] dengan Data[k]
9 i ← i + 1

Untuk lebih jelas mengenai langkah-langkah algoritma seleksi, dapat dilihat pada tabel 10.2. Proses pengurutan pada Tabel 10.2 dapat dijelaskan sebagai berikut:

1. Pada saat i = 0, data terkecil antara data ke-1 s/d ke-9 adalah data ke-4, yaitu 3, maka data ke-0 yaitu 12 ditukar dengan data ke-4 yaitu 3.
2. Pada saat i = 1, data terkecil antara data ke-2 s/d ke-9 adalah data ke-2, yaitu 9, maka data ke-1 yaitu 35 ditukar dengan data ke-2 yaitu 9.
3. Pada saat i = 2, data terkecil antara data ke-3 s/d ke-9 adalah data ke-3, yaitu 11, maka data ke-2 yaitu 35 ditukar dengan data ke-3 yaitu 11.
4. Pada saat i = 3, data terkecil antara data ke-4 s/d ke-9 adalah data ke-4, yaitu 12, maka data ke-3 yaitu 35 ditukar dengan data ke-4 yaitu 12.
5. Dan, seterusnya.

Iterasi	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
Awal	12	35	9	11	3	17	23	15	31	20
i=0	**12**	35	9	11	**3**	17	23	15	31	20
i=1	3	**35**	**9**	11	12	17	23	15	31	20
i=2	3	9	**35**	**11**	12	17	23	15	31	20
i=3	3	9	11	**35**	**12**	17	23	15	31	20
i=4	3	9	11	12	**35**	17	23	**15**	31	20
i=5	3	9	11	12	15	**17**	23	35	31	20
i=6	3	9	11	12	15	17	**23**	35	31	**20**
i=7	3	9	11	12	15	17	20	**35**	31	**23**
i=8	3	9	11	12	15	17	20	23	**31**	35
Akhir	3	9	11	12	15	17	20	23	31	35

Tabel 10.2
Proses Pengurutan dengan Metode Seleksi

Berikut merupakan prosedur yang menggunakan metode seleksi:

```
void SelectionSort()
{
        int i, j, k;
        for(i=0; i<Max-1;i++){
                k = i;
                for (j=i+1; j<Max; j++)
                        if(Data[k] > Data[j])
                                k = j;
                Tukar(&Data[i], &Data[k]);
        }
}
```

Program 10.4
Prosedur Pengurutan dengan Metode Seleksi

Dari algoritma dan prosedur di atas, jumlah perbandingan (=C) metode seleksi adalah:

$$C = N(N - 1) / 2$$

Jumlah penukaran (=M) pada metode seleksi tergantung keadaan datanya. Penukaran minimum terjadi bila data sudah dalam keadaan urut, sebaliknya jumlah penukaran maksimum terjadi bila data dalam keadaan urut terbalik. Jumlah penukaran minimum dan maksimum dapat dirumuskan sebagai berikut:

$$M_{min} = 3(N - 1)$$

$$M_{max} = [N^2 / 4] + 3(N - 1)$$

G. Metode Gelembung (Bubble Sort)

Metode gelembung (*bubble sort*) atau sering disebut dengan metode penukaran (*exchange sort*) adalah metode yang mengurutkan data dengan cara membandingkan masing-masing elemen, kemudian melakukan penukaran bila perlu. Metode ini mudah dipahami dan diprogram. Tetapi, bila dibandingkan dengan metode lain yang kita pelajari, metode ini merupakan metode yang paling tidak efisien.

Proses pengurutan metode gelembung ini menggunakan dua kalang. Kalang pertama melakukan pengulangan dari elemen ke-2 sampai dengan elemen ke-N-1 (misalnya variabel i). Sedangkan, kalang kedua melakukan pengulangan menurun dari elemen ke-N sampai elemen ke-i (misalnya variabel j). Pada setiap pengulangan, elemen ke j-1 dibandingkan dengan elemen ke j. Apabila data ke j-1 lebih besar daripada data ke j, dilakukan penukaran.

Algoritma gelembung dapat dituliskan sebagai berikut:

```
1  i ← 0
2  selama (i < N-1) kerjakan baris 3 sampai dengan 7
3  j ← N - 1
4  Selama (j >= i) kerjakan baris 5 sampai dengan 7
5  Jika (Data[j-1] > Data[j]) maka tukar Data[j-1] dengan
   Data[j]
6  j ← j – 1
7  i ← i + 1
```


Untuk lebih jelasnya mengenai langkah-langkah algoritma gelembung, dapat dilihat pada Tabel 10.3. Proses pengurutan pada Tabel 10.3 dapat dijelaskan sebagai berikut:

1. Pada saat i = 1, nilai j diulang dari 9 sampai dengan 1. Pada pengulangan pertama, Data[9] dibandingkan Data[8], karena 20 < 31 maka Data[9] dan Data[8] ditukar. Pada pengulangan kedua, Data[8] dibandingkan Data[7], karena 20 > 15 maka proses dilanjutkan. Demikian seterusnya sampai j = 1.
2. Pada saat i = 2, nilai j diulang dari 9 sampai dengan 2. Pada pengulangan pertama, Data[9] dibandingkan Data[8], karena 31 > 20 maka proses dilanjutkan. Pada pengulangan kedua, Data[8] dibandingkan Data[7], karena 20 < 23 Data[8] dan Data[7] ditukar. Demikian seterusnya sampai j = 2.
3. Pada saat i = 3, nilai j diulang dari 9 sampai dengan 3. Pada pengulangan pertama, Data[9] dibandingkan Data[8], karena 31 > 23 maka proses dilanjutkan. Pada pengulangan kedua, Data[8] dibandingkan Data[7], karena 23 > 20 maka proses dilanjutkan. Demikian seterusnya sampai j = 3.
4. Dan, seterusnya sampai dengan i = 8.

Iterasi	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
Awal	12	35	9	11	3	17	23	15	31	20
i=1; j=9	12	35	9	11	3	17	23	15	**31**	**20**
j=8	12	35	9	11	3	17	23	**15**	**20**	31
j=7	12	35	9	11	3	17	**23**	**15**	20	31
j=6	12	35	9	11	3	**17**	**15**	23	20	31
j=5	12	35	9	11	**3**	**15**	17	23	20	31
j=4	12	35	9	**11**	**3**	15	17	23	20	31
j=3	12	35	**9**	**3**	11	15	17	23	20	31
j=2	12	**35**	**3**	9	11	15	17	23	20	31
j=1	**12**	**3**	35	9	11	15	17	23	20	31
i=2; j=9	3	12	35	9	11	15	17	23	**20**	**31**
j=8	3	12	35	9	11	15	17	**23**	**20**	31
j=7	3	12	35	9	11	15	**17**	**20**	23	31
j=6	3	12	35	9	11	**16**	**17**	20	23	31
j=5	3	12	35	9	**11**	**15**	17	20	23	31
j=4	3	12	35	**9**	**11**	15	17	20	23	31
j=3	3	12	**35**	**9**	11	15	17	20	23	31
j=2	3	**12**	**9**	35	11	15	17	20	23	31
i=3; j=9	3	9	12	35	11	15	17	20	**23**	**31**
j=8	3	9	12	35	11	15	17	**20**	**23**	31
j=7	3	9	12	35	11	15	**17**	**20**	23	31
j=6	3	9	12	35	11	**15**	**17**	20	23	31
j=5	3	9	12	35	**11**	**15**	17	20	23	31
j=4	3	9	12	**35**	**11**	15	17	20	23	31
j=3	3	9	**12**	**11**	35	15	17	20	23	31
i=4; j=9	3	9	11	12	35	15	17	20	**23**	**31**
j=8	3	9	11	12	35	15	17	**20**	**23**	31
j=7	3	9	11	12	35	15	**17**	**20**	23	31
j=6	3	9	11	12	35	**15**	**17**	20	23	31
j=5	3	9	11	12	**35**	**15**	17	20	23	31
i=4	3	9	11	**12**	**15**	35	17	20	23	31
i=5; j=9	3	9	11	12	15	35	17	20	**23**	**31**
j=8	3	9	11	12	15	35	17	**20**	**23**	31
j=7	3	9	11	12	15	35	**17**	**20**	23	31
j=6	3	9	11	12	15	**35**	**17**	20	23	31
j=5	3	9	11	12	**15**	**17**	35	20	23	31
i=6; j=9	3	9	11	12	15	17	35	20	**23**	**31**
j=8	3	9	11	12	15	17	35	**20**	**23**	31
j=7	3	9	11	12	15	17	**35**	**20**	23	31
j=6	3	9	11	12	15	**17**	**20**	35	23	31
i=7; j=9	3	9	11	12	15	17	20	35	**23**	**31**
j=8	3	9	11	12	15	17	20	**35**	**23**	31
j=7	3	9	11	12	15	17	**20**	**23**	35	31
i=8; j=9	3	9	11	12	15	17	20	23	**35**	**31**
j=8	3	9	11	12	15	17	20	23	31	35
Akhir	3	9	11	12	15	17	20	23	31	35

Tabel 10.3
Proses Pengurutan dengan Metode Gelembung

Berikut merupakan prosedur yang menggunakan metode gelembung:

```
void BubbleSort()
{
        int i, j;
        for(i=1; i<Max-1; i++)
                for(j=Max-1; j>=i; j--)
                        if(Data[j-1] > Data[j])
                                Tukar(&Data[j-1], &Data[j]);
}
```

Program 10.5
Prosedur Pengurutan dengan Metode Gelembung

Dari algoritma dan prosedur di atas, jumlah perbandingan (=C) metode gelembung selalu konstan, yaitu:

$$C = (N^2 - N)/2$$

Jumlah penukaran data (=M) pada metode gelembung tergantung keadaan data. Jumlah penukaran minimum terjadi bila data sudah dalam keadaan urut, sebaliknya jumlah penukaran maksimum terjadi bila data dalam keadaan urut terbalik. Adapun rumus dari jumlah penukaran pada metode gelembung adalah sebagai berikut:

$$M_{min} = 0$$

$$M_{rata\text{-}rata} = 3(N^2 - N) / 4$$

$$M_{max} = 3(N^2 - N) / 2$$

H. Metode Shell (Shell Sort)

Metode ini disebut juga dengan metode pertambahan menurun (*diminishing increment*). Metode ini dikembangkan oleh Donald L. Shell pada tahun 1959, sehingga sering disebut dengan metode *shell sort*. Metode ini mengurutkan data dengan cara membandingkan suatu data dengan data lain yang memiliki jarak tertentu, kemudian dilakukan penukaran bila diperlukan

Proses pengurutan dengan metode *shell sort* dapat dijelaskan sebagai berikut. Pertama-tama adalah menentukan jarak mula-mula dari data yang akan dibandingkan yaitu N/2. Data pertama dibandingkan dengan data dengan jarak N/2. Apabila data pertama lebih besar dari data ke-N/2 maka kedua data tersebut ditukar. Kemudian, data kedua dibandingkan dengan jarak yang sama yaitu N/2. Demikian seterusnya sampai seluruh data dibandingkan, sehingga semua data ke-j selalu lebih kecil daripada data ke-(j + N/2).

Pada proses berikutnya, digunakan jarak (N/2)/2 atau N/4. Data pertama dibandingkan dengan data dengan jarak N/4. Apabila data pertama lebih besar dari data ke-N/4 maka kedua data tersebut ditukar. Kemudian, data kedua dibandingkan dengan jarak yang sama yaitu N/4. Demikian seterusnya sampai seluruh data dibandingkan sehingga semua data ke-j lebih kecil daripada data ke-(j + N/4).

Pada proses berikutnya, digunakan jarak (N/4)/2 atau N/8. Demikian seterusnya sampai jarak yang digunakan adalah 1.

Algoritma metode *shell sort* dapat dituliskan sebagai berikut:

1 Jarak ← N
2 Selama (Jarak > 1) kerjakan baris 3 sampai dengan 9
3 Jarak ← Jarak / 2. Sudah ← false
4 Kerjakan baris 4 sampai dengan 8 selama Sudah = false
5 Sudah ← true
6 j ← 0
7 Selama (j < N – Jarak) kerjakan baris 8 dan 9
8 Jika (Data[j] > Data[j + Jarak] maka tukar Data[j], Data[j + Jarak].
Sudah ← true
9 j ← j + 1

Untuk memperjelas pemahaman mengenai langkah-langkah algoritma penyisipan langsung, dapat dilihat pada Tabel 10.4. Proses pengurutan pada Tabel 10.4 dapat dijelaskan sebagai berikut:

1. Pada saat Jarak = 5, j diulang dari 0 sampai dengan 4. Pada pengulangan pertama, Data[0] dibandingkan dengan Data[5]. Karena 12 < 17, maka tidak terjadi penukaran. Kemudian, Data[1] dibandingkan dengan Data[6]. Karena

35 > 23, maka Data[1] ditukar dengan Data[6]. Demikian seterusnya sampai j = 4.

2. Pada saat Jarak = 5/2 = 2, j diulang dari 0 sampai dengan 7. Pada pengulangan pertama, Data[0] dibandingkan dengan Data[2]. Karena 12 > 9, maka Data[0] ditukar dengan Data[2]. Kemudian, Data[1] dibandingkan dengan Data[3], dan terjadi penukaran karena 23 > 11. Demikian seterusnya sampai j = 7. Perhatikan untuk Jarak = 2, proses pengulangan harus dilakukan lagi karena ternyata Data[0] > Data[2]. Proses pengulangan ini berhenti bila Sudah=true.
3. Demikian seterusnya sampai Jarak = 1.

Iterasi	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
Awal	12	35	9	11	3	17	23	15	31	20
Jarak=5	12	35	9	11	3	17	23	15	31	20
i=6	12	**35**	9	11	3	17	**23**	15	31	20
Jarak=2	12	23	9	11	3	17	35	15	31	20
i=2	**12**	23	**9**	11	3	17	35	15	31	20
i=3	9	**23**	12	**11**	3	17	35	15	31	20
i=4	9	11	**12**	23	**3**	17	35	15	31	20
i=5	9	11	3	**23**	12	**17**	35	15	31	20
i=7	9	11	3	17	12	**23**	35	**15**	31	20
i=8	9	11	3	17	12	15	**35**	23	**31**	20
i=9	9	11	3	17	12	15	31	**23**	35	**20**
i=2	**9**	11	**3**	17	12	15	31	20	35	23
i=3	3	**11**	9	**17**	12	15	31	20	35	23
Jarak=1	3	11	9	15	12	17	31	20	35	23
i=2	3	**11**	**9**	15	12	17	31	20	35	23
i=4	3	9	11	**15**	**12**	17	31	20	35	23
i=7	3	9	11	12	15	17	**31**	**20**	35	23
i=9	3	9	11	12	15	17	20	31	**35**	**23**
i=1	**3**	**9**	11	12	15	17	20	31	23	35
i=8	3	9	11	12	15	17	20	**31**	**23**	35
i=9	3	9	11	12	15	17	20	23	**31**	**35**
Akhir	3	9	11	12	15	17	20	23	31	35

Tabel 10.4
Proses Pengurutan dengan Metode Shell

Berikut merupakan prosedur yang menggunakan metode Shell:

```
void ShellSort(int N)
{
        int Jarak, i, j;
        bool Sudah;
        Jarak = N;
        while(Lompat > 1){
                Jarak = Jarak / 2;
                Sudah = false;
                while(!Sudah){
                        Sudah = true;
                        for(j=0; j<N-Jarak; j++){
                                i = j + Jarak;
                                if(Data[j] > Data[i]){

                                Tukar(&Data[j], &Data[i]);
                                Sudah = false;
                                }
                        }
                }
        }
}
```

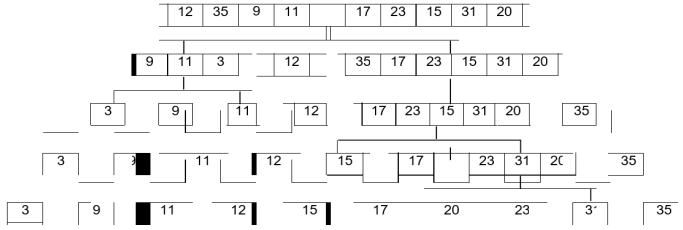
Program 10.6
Prosedur Pengurutan dengan Metode Shell

I. Metode Quick (Quick Sort)

Metode *quick* sering disebut juga metode partisi (partition exchange sort). Metode ini diperkenalkan pertama kali oleh C.A.R. Hoare pada tahun 1962. Untuk mempertinggi efektivitas metode ini, digunakan teknik menukarkan dua elemen dengan jarak yang cukup besar.

Proses penukaran dengan metode *quick* dapat dijelaskan sebagai berikut. Mula-mula dipilih data tertentu yang disebut pivot, misalnya x. Pivot dipilih untuk mengatur data di sebelah kiri agar lebih kecil daripada pivot dan data di sebelah kanan agar lebih besar daripada pivot. Pivot ini diletakkan

pada posisi ke-j sedemikian sehingga data antara 1 sampai dengan j-1 lebih kecil daripada x. Sedangkan, data pada posisi ke-j+1 sampai N lebih besar daripada x, caranya yaitu dengan menukarkan data di antara posisi 1 sampai dengan j-1 yang lebih besar daripada x dengan data di antara posisi j+1 sampai dengan N yang lebih kecil daripada x. Ilustrasi dari metode *quick* dapat dilihat pada Gambar 10.1.



Gambar 10.1
Ilustrasi Metode Quick Sort

Gambar 10.1 menunjukkan pembagian data menjadi sub-subbagian. Pivot dipilih dari data pertama tiap bagian maupun subbagian, tetapi sebenarnya kita bisa memilih sembarang data sebagai pivot. Dari ilustrasi tersebut bisa kita lihat bahwa metode *quick* dapat kita implementasikan menggunakan dua cara, yaitu dengan cara nonrekursif dan rekursif. Pada kedua cara tersebut, persoalan utama yang perlu kita perhatikan adalah bagaimana kita meletakkan suatu data pada posisinya yang tepat sehingga memenuhi ketentuan dan bagaimana menyimpan batas-batas subbagian. Dengan cara seperti yang ditunjukkan pada Gambar 10.1, kita hanya menggerakkan

data pertama sampai di suatu tempat yang sesuai. Dalam hal ini, kita hanya bergerak dari satu arah. Untuk mempercepat penempatan suatu data, kita bisa bergerak dari dua arah, yaitu kiri dan kanan. Caranya adalah sebagai berikut:

Misalnya, kita mempunyai 10 data (N=9) sebagai berikut:

12	**35**	**9**	**11**	**3**	**17**	**23**	**15**	**31**	**20**
i = 0									**j = 9**

Pertama-tama, ditentukan i = 0 (untuk bergerak dari kiri ke kanan), dan j = N (untuk bergerak dari kanan ke kiri). Proses akan dihentikan jika nilai i lebih besar atau sama dengan j. Sebagai contoh, kita akan menempatkan elemen pertama, 12 pada posisinya yang tepat dengan bergerak dari dua arah, dari kiri ke kanan dan dari kanan ke kiri secara bergantian. Dimulai dari data terakhir bergerak dari kanan ke kiri (j dikurangi 1), dilakukan pembandingan data sampai ditemukan data yang nilainya lebih kecil dari 12 yaitu 3. Kedua elemen data ini kemudian kita tukarkan sehingga diperoleh:

3	**35**	**9**	**11**	**12**	**17**	**23**	**15**	**31**	**20**
i = 0				**j = 4**					

Selanjutnya, bergerak dari kiri ke kanan dimulai dari data 3 (i ditambah 1), dilakukan pembandingan pada setiap data yang dilalui dengan 12, sampai ditemukan data yang nilainya lebih besar dari 12 yaitu 35. Kedua data ini lalu kita tukarkan sehingga diperoleh:

3	12	9	11	35	17	23	15	31	20
	i = 1			j = 4					

Berikutnya, bergerak dari kanan ke kiri dimulai dari 11. Dan, ternyata data 11 lebih kecil dari 12. Kedua data ini kemudian ditukarkan sehingga diperoleh:

3	11	9	12	35	17	23	15	31	20
	i = 1		j = 3						

Kemudian, dimulai dari 9 bergerak dari kiri ke kanan. Pada langkah ini ternyata tidak ditemukan data yang lebih besar dari 12 sampai nilai i = j. Hal ini berarti proses penempatan data yang bernilai 12 telah selesai, sehingga semua data yang lebih kecil dari 12 berada di sebelah kiri dan data yang lebih besar dari 12 berada di sebelah kanan seperti terlihat sebagai berikut:

3	11	9

12

35	17	23	15	31	20

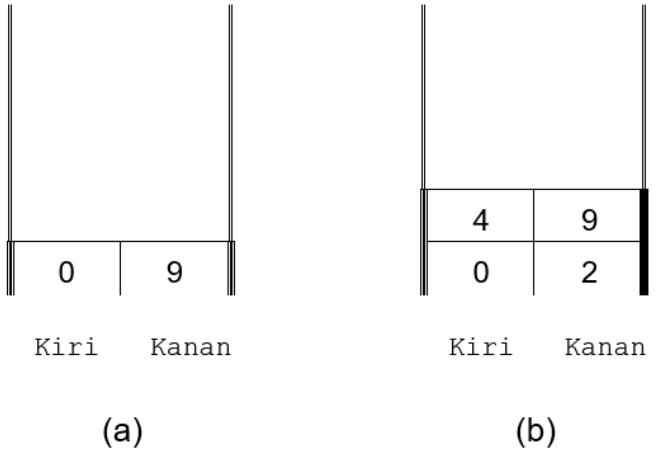
Sebagaimana telah disebutkan sebelumnya, implementasi metode ini dapat dilakukan secara nonrekursif maupun rekursif.

1. Metode Quick Sort Nonrekursif

Implementasi metode *quick sort* secara nonrekursif memerlukan dua buah tumpukan (*stack*) yang digunakan untuk menyimpan batas-batas subbagian. Pada prosedur

ini, digunakan tumpukan yang bertipe *record* (struktur), yang terdiri dari elemen kiri (untuk mencatat batas kiri) dan kanan (untuk mencatat batas kanan. Tumpukan dalam hal ini dideklarasikan sebagai *array*.

Dengan mengacu pada contoh di atas, sebelum metode ini dijalankan data pertama tumpukan, kiri diatur sama dengan 0 dan kanan sama dengan N-1 (pada contoh sama dengan 9). Lihat Gambar 10.2a. Setelah data 12 berada pada posisi yang tepat, maka isi tumpukan berubah seperti Gambar 10.2b.



Gambar 10.2
Perubahan Isi Tumpukan; (a) Saat Mulai Iterasi, dan (b) Setelah Satu Elemen Ditempatkan

Algoritma *quick sort* nonrekursif dapat dituliskan sebagai berikut:

1 Tumpukan[1].Kiri ← 0

```
2  Tumpukan[1].Kanan ← N-1
3  Selama ujung ◷ 0 kerjakan baris 4 sampai dengan 22
4  L ← Tumpukan[ujung].Kiri
5  R ← Tumpukan[ujung].Kanan
6  ujung ← ujung – 1
7  Selama (R > L) kerjakan baris sampai 8 dengan 22
8  i ← L
9  j ← R
10 x ← Data[(L + R) / 2]
11 Selama i <= j kerjakan baris 12 sampai dengan 14
12 Selama (Data[i] < x), i ← i + 1
13 Selama (x < Data[j]), j ← j – 1
14 Jika (i <= j) maka kerjakan baris 15 sampai dengan 17,
   jika tidak ke baris 11
15 Tukar Data[i] dengan Data[j]
16 i ← i + 1
17 j ← j –1
18 Jika (L < i) maka kerjakan baris 19 sampai dengan 21
19 ujung ← ujung + 1
20 Tumpukan[ujung].Kiri = i
21 Tumpukan[ujung].Kanan = R
22 R ← j
```


Untuk memperjelas pemahaman mengenai langkah-langkah algoritma *quick*, dapat dilihat pada Tabel 10.5. Proses pengurutan pada Tabel 10.5 dapat dijelaskan sebagai berikut:

1. Mula-mula, L = 0 dan R = 9 dan pivot adalah pada data ke-4 yaitu 3. Kita mencari data di sebelah kiri pivot yang lebih besar daripada 3, ternyata data ke-0 yaitu 12 lebih besar daripada 3. Untuk data di sebelah kanan pivot, ternyata tidak ada data yang lebih kecil daripada 3, yang berarti 3 adalah data terkecil. Sekarang, 3 harus ditukar dengan 12 seperti terlihat pada Tabel 10.5.

12	35	9	11	3	17	23	15	31	20
				⇓					
3	35	9	11	12	17	23	15	31	20

2. Langkah berikutnya yaitu membuat dua kumpulan data baru berdasarkan hasil ini. Kumpulan data pertama adalah data yang memiliki indeks 0 s/d (4 - 0) = 4, yaitu:

3 35 9 11

3. Kumpulan data kedua adalah data yang memiliki indeks 0 + 1 = 1 s/d 9. Kumpulan data kedua ini belum bisa ditentukan sekarang karena masih tergantung dari hasil pengurutan kumpulan data pertama.
4. Kembali ke kumpulan data pertama, dicari pivot kemudian digunakan aturan yang serupa. Pivot pada kumpulan data ini adalah data ke-2 yaitu 9. Perhatikan bahwa

terdapat data yang lebih besar dari 9 di sebelah kiri yaitu 35, sehingga 35 ditukar dengan 9.

3 9 35 11

5. Langkah selanjutnya adalah membuat dua kumpulan data lagi. Kumpulan data pertama mempunyai indeks 0 sampai dengan (4 - 1) = 3. Jadi, kumpulan data pertama menjadi:

3 9 35

6. Pivot dari kumpulan data ini adalah 9. Perhatikan bahwa tidak ada data yang lebih besar dari 9 di sebelah kiri dan lebih kecil dari 9 di sebelah kanan.
7. Kumpulan kedua dari indeks 0 s/d 4 adalah data ke-2 dan ke-(4 - 1 = 3), yaitu 35 dan 11. Ternyata 35 lebih besar dari 11 sehingga kedua data ini ditukar.
8. Kembali ke kumpulan data kedua yang memiliki indeks 1 s/d 9. Kumpulan ini dibagi menjadi dua yaitu kumpulan data berindeks 1 s/d 4 dan kumpulan data berindeks 5 s/d 9. Pivot data ini adalah data ke-5 yaitu 17. Kumpulan data ini dapat dituliskan sebagai berikut:

9 11 35 12 17 23 15 31 20

9. Data yang lebih besar dari 17 di sebelah kiri adalah 35 dan data yang lebih kecil dari 17 di sebelah kanan adalah 15, jadi kedua data ini ditukar menjadi:

9 11 15 12 17 23 35 31 20

10. Dari hasil penukaran ini dilakukan pembagian menjadi 2 kumpulan data. Kumpulan pertama yaitu dari indeks 2 s/d 4, pivot pada data ke-3 dan terjadi penukaran data 15 dan 12 sehingga menjadi:

11 12 15

11. Kumpulan kedua yaitu dari indeks 4 s/d 5 tidak terjadi pertukaran data.
12. Kembali ke kumpulan kedua dari indeks 5 s/d 9. Pivot dari kumpulan ini adalah 35. Selanjutnya, dicari data yang lebih besar dari 35 di sebelah kiri dan data yang lebih kecil dari 35 di sebelah kanan. Ternyata, terjadi pertukaran data 35 dan 20.
13. Dari hasil pertukaran ini, kemudian dilakukan pembagian kumpulan data yaitu data yang mempunyai indeks 6 s/d 7 dan 8 s/9. Kumpulan data pertama terjadi pertukaran data 23 dan 20, sedangkan kumpulan data kedua tidak terjadi pertukaran data.

Iterasi	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
Awal	12	35	9	11	3	17	23	15	31	20
L=0;R=9	**12**	35	9	11	**3**	17	23	15	31	20
L=0;R=4	3	**35**	**9**	11	12	17	23	15	31	20
L=1;R=3	3	9	**35**	**11**	12	17	23	15	31	20
L=1;R=9	3	9	11	**35**	12	17	23	**15**	31	20
L=2;R=4	3	9	11	**15**	**12**	17	23	35	31	20
L=5;R=9	3	9	11	12	15	17	23	**35**	31	**20**
L=6;R=7	3	9	11	12	15	17	**23**	**20**	31	35
L=8;R=9	3	9	11	12	15	17	20	23	31	35
Akhir	3	9	11	12	15	17	20	23	31	35

Tabel 10.5
Proses Pengurutan dengan Metode Quick

Berikut merupakan prosedur yang menggunakan metode *quick* nonrekursif:

```
void QuickSortNonRekursif(int N)
{
        const M = MaxStack;
        struct tump {
                int Kiri;
                int Kanan;
        }Tumpukan[M];

        int i, j, L, R, x, ujung = 1;
        Tumpukan[1].Kiri = 0;
        Tumpukan[1].Kanan = N-1;

        while (ujung!=0){
                L = Tumpukan[ujung].Kiri;
                R = Tumpukan[ujung].Kanan;
                ujung--;
                while(R > L){
                        i = L;
                        j = R;
                        x = Data[(L+R)/2];

                while(i <= j){
                        while(Data[i] < x)
                                i++;
                        while(x < Data[j])
                                j--;
                        if(i <= j){
                                Tukar(&Data[i], &Data[j]); i++;
                                j--;
                        }
                }
                if(L < i){
                        ujung++;
                        Tumpukan[ujung].Kiri = i;
                        Tumpukan[ujung].Kanan = R;
                }
                R = j;
        }
}
}
```

Program 10.7
Prosedur Pengurutan dengan Metode Quick Nonrekursif

2. Metode Quick Sort Rekursif

Algoritma metode *quick* rekursif dapat dituliskan sebagai berikut:

1 x ← Data[(L + R) / 2]

2 i ← L

3 j ← R

4 Selama (i <= j) kerjakan baris 5 sampai dengan 12

5 Selama (Data[i] < x) kerjakan i ← i + 1

6 Selama (Data[j] > x) kerjakan j ← j – 1

7 Jika (i <= j) maka kerjakan baris 8 sampai dengan 10; jika tidak kerjakan baris 11

8 Tukar Data[i] dengan Data[j]

9 i ← i + 1

10 j ← j –1

11 Jika (L < j) kerjakan lagi baris 1 dengan R = j

12 Jika (i < R) kerjakan lagi baris 1 dengan L = i

Berikut merupakan prosedur yang menggunakan metode *quick sort* dengan rekursif:

```
void QuickSortRekursif(int L, int R)
{
        int i, j, x;
        x = data[(L+R)/2];
        i = L;
        j = R;
        while (i <= j){
                while(Data[i] < x)
                        i++;
                while(Data[j] > x)
                        j--;
                if(i <= j){
                        Tukar(&Data[i], &Data[j]);
                        i++;
                        j--;
                }
        }
        if(L < j)
                QuickSortRekursif(L, j);
        if(i < R)
                QuickSortRekursif(i, R);
}
```

Program 10.8
Prosedur Pengurutan dengan Metode Quick Sort Rekursif

J. Metode Penggabungan (Merge Sort)

Metode penggabungan biasanya digunakan pada pengurutan berkas. Prinsip metode ini yaitu mula-mula diberikan dua kumpulan data yang sudah dalam keadaan urut. Kedua kumpulan data tersebut harus dijadikan satu tabel sehingga dalam keadaan urut.

Misalnya, kumpulan data pertama (T1) adalah sebagai berikut:

3 11 12 23 31

Sedangkan, kumpulan data kedua (T2) adalah sebagai berikut:

9 15 17 20 35

Maka, proses penggabungan kedua data tersebut dapat dijelaskan sebagai berikut. Mula-mula, diambil data pertama dari T1 yaitu 3 dan data pertama dari T2 yaitu 9. Data ini dibandingkan, kemudian yang lebih kecil diletakkan sebagai data pertama dari hasil pengurutan, misalnya T3. Jadi, T3 akan memiliki satu data yaitu 3. Data yang lebih besar yaitu 9 kemudian dibandingkan dengan data kedua dari T1 yaitu 11. Ternyata, 9 lebih kecil dari 11, sehingga 9 diletakkan sebagai data kedua dari T3. Demikian seterusnya sehingga didapat hasil sebagai berikut:

3 9 11 12 15 17 20 23 31 35

Algoritma penggabungan dapat dituliskan sebagai berikut:

1 i ← 0
2 j ← 0
3 J3 ← 0
4 Kerjakan baris 5 sampai dengan 7 selama (i < J1) atau (j < J2)
5 J3 ← J3 + 1
6 Jika (T1[i] < T2[j]) maka T3[J3] ← T1[i], i ← i + 1
7 Jika (T1[i] >= T2[j]) maka T3[J3] ← T2[j], j ← j + 1

8 Jika (i > J1) maka kerjakan baris 9, jika tidak kerjakan baris 15

9 i ← j

10 Selama (i < J2) kerjakan baris 11 sampai dengan 13

11 J3 ← J3 + 1

12 T3[J3] ← T2[i]

13 i ← i + 1

14 Selesai

15 j ← i

16 Selama (j < J1) kerjakan baris 17 sampai dengan 19

17 J3 ← J3 + 1

18 T3[J3] ← T1[j]

19 j ← j + 1

Berikut merupakan prosedur penggabungan dua kumpulan data yang sudah dalam keadaan urut:

```
void MergeSort(int T1[],int T2[],int J1,int J2, int T3[],int *J3)
{
        int i=0, j=0;
        int t=0;
        while ((i<J1)||(j<J2)){
                if(T1[i]<T2[j]){
                        T3[t] = T1[i];
                        i++;
                }
                else{
                        T3[t] = T2[j];
                        j++;
                }
                t++;
}

if(i>J1)
        for(i=j; i<J2; i++){
                T3[t] = T2[i];
                t++;
        }

if(j>J2)
        for(j=i; j<J1; j++){
        T3[t] = T1[j]; t++;
        }
*J3 = t;
}
```

Program 10.9
Prosedur Pengurutan dengan Metode Merge

K. Metode Radix Sort

Radix sort adalah metode *sorting* ajaib yang mengatur pengurutan nilai tanpa melakukan beberapa perbandingan pada data yang dimasukkan. Secara harfiah, *radix* dapat diartikan sebagai posisi dalam angka. Pada sistem desimal, *radix* adalah digit dalam angka desimal. Misal, angka 42 mempunyai 2 digit, yaitu 4 dan 2. *Radix sort* memperoleh namanya dari digit-digit tersebut, karena metode ini mula-mula

mengurutkan nilai-nilai *input* berdasarkan *radix* pertamanya, lalu *radix* keduanya, dan seterusnya.

1. Implementasi Radix Sort

Radix sort merupakan algoritma pengurutan yang cepat, mudah, dan sangat efektif. Namun, banyak orang yang berpikir bahwa algoritma ini memiliki banyak batasan. Untuk kasus-kasus tertentu tidak dapat dilakukan dengan algoritma ini. Misalnya, pada pengurutan bilangan pecahan dan bilangan negatif, adanya kompleksitas *bit* dan *word*, dan pengurutan pada *multiple keys*. *Radix sort* hanya bisa digunakan pada bilangan integer. Untuk bilangan pecahan bisa dilakukan dengan perantara *bucket sort* atau metode berbasis perbandingan yang lain. Dalam perilakunya yang melihat digit-digit angka sebagai pengontrolnya, *radix sort* dapat diimplementasikan dalam pengurutan bilangan desimal dan bilangan *bit*.

Contoh:

Kita mempunyai data sebagai berikut:

342 547 321 213 453 621 120 958 124 854

DATA	KELOMPOK									
	0	1	2	3	4	5	6	7	8	9
342										
547										
321										
213										
453										
621										
120										
958										
124										
854										
HASIL										

Gambar 10. 3
Tabel Radix Sort Kosong

Kemudian, data diurutkan secara kolom. Di sini, lalu kita masukan mulai dari nilai satuan untuk mencapai iterasi 1. Misalnya, 342 terdiri atas 300 sebagai ratusan, 40 sebagai puluhan, dan 2 adalah satuan.

DATA	KELOMPOK									
	0	1	2	3	4	5	6	7	8	9
342			342							
547								547		
321		321								
213				213						
453				453						
621		621								
120	120									
958									958	
124					124					
854					854					
HASIL	120	321	621	342	213	453	124	854	547	958

Gambar 10.4
Tabel Iterasi 1 Radix Sort

Setelah mendapatkan hasil dari sortir satuan, kemudian kita masukkan sebagai data di iterasi 2, digit puluhan yang akan dimasukkan dalam tabel, di sini digit tengahnya yang dimasukkan.

DATA	KELOMPOK									
	0	1	2	3	4	5	6	7	8	9
120			120							
321			321							
621			621							
342					342					
213		213								
453						453				
124			124							
854						854				
547					547					
958						958				
HASIL	213	120	321	621	124	342	547	453	854	958

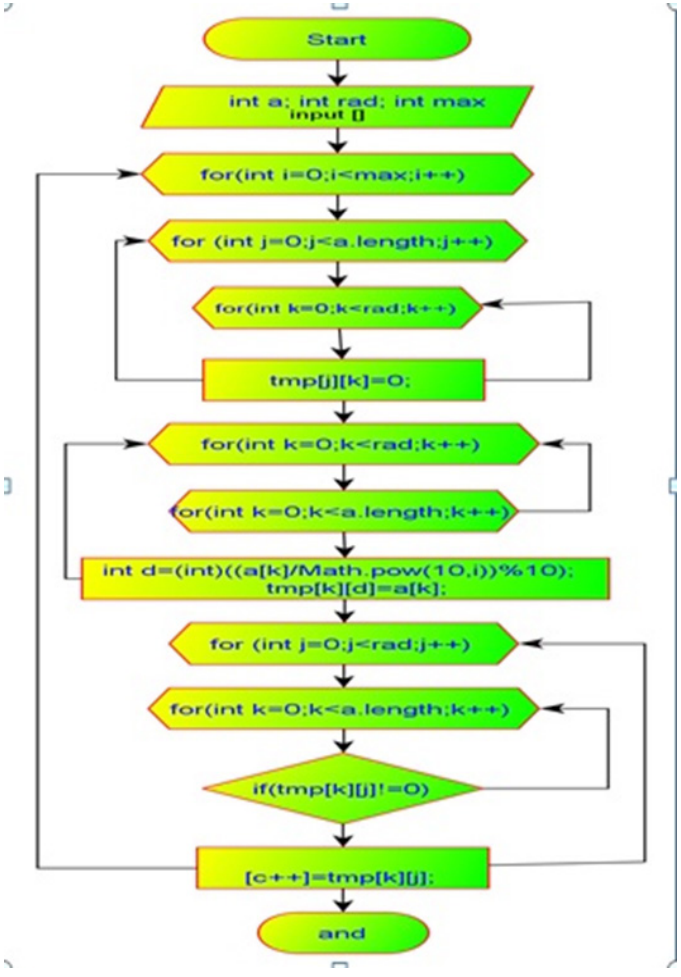
Gambar 10.5 Tabel Iterasi 2 Radix Sort

Setelah mendapatkan hasil dari sortir puluhan, kemudian kita masukan sebagai data di iterasi 3, digit ratusan yang akan dimasukkan dalam tabel, di sini digit pertamanya yang dimasukkan.

DATA	KELOMPOK									
	0	1	2	3	4	5	6	7	8	9
213			213							
120		120								
321				321						
621							621			
124		124								
342				342						
547						547				
453					453					
854									854	
958										958
HASIL	120	124	218	321	342	453	547	621	854	958

Gambar 10.6
Tabel Iterasi 3 Radix Sort

2. Flowchart Radix Sort



9

Graph

Tujuan:
- Mahasiswa mampu memahami struktur data *graph*.
- Mahasiswa mampu mengimplementasikan algoritma pencarian jalur terpendek.

A. Teori

Graph sering digunakan untuk merepresentasikan sebuah objek dan hubungannya dengan objek lain. Sejarah teori *graph* bermula saat ahli matematika Swiss, Leonhard Euler, memecahkan masalah jembatan Königsberg. Masalah jembatan Königsberg adalah teka-teki lama mengenai kemungkinan menemukan jalan setapak di tujuh jembatan yang membentang di sepanjang sebuah sungai bercabang yang melewati sebuah pulau namun tanpa melewati jembatan dua kali. Euler berpendapat bahwa tidak ada jalan semacam itu. Buktinya memang hanya mengacu pada susunan fisik jembatan, namun intinya ia membuktikan teorema pertama dalam teori *graph* (Carlson, 2017).

Seperti yang digunakan dalam teori grafik, grafik istilah tidak mengacu pada grafik data, seperti grafik garis atau grafik batang. Sebaliknya, ini mengacu pada sekumpulan simpul (yaitu titik atau simpul) dan tepi (atau garis) yang menghubungkan simpul. Bila dua simpul digabungkan lebih dari satu tepi, grafiknya disebut *multigraph*. Grafik tanpa *loop* dan paling banyak satu tepi antara dua simpul disebut grafik sederhana. Kecuali dinyatakan lain, grafik diasumsikan mengacu pada grafik sederhana. Bila setiap simpul dihubungkan oleh ujung ke setiap titik lainnya, grafik disebut grafik lengkap. Bila sesuai, arah dapat diberikan ke masing-masing ujung untuk menghasilkan apa yang dikenal sebagai grafik terarah, atau digraf (Carlson, 2017).

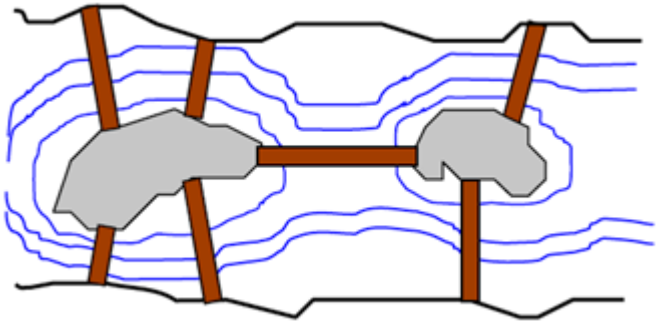
Graph pada dasarnya mempunyai komponen berupa simpul dan sisi dan pada *graph* tersebut sehingga membentuk *graph* terbuka dan *graph* tertutup, sehingga membentuk sejumlah lintasan dan sirkuit. Dengan demikian, teorema *graph* telah dapat menyelesaikan tanda tanya dalam penyelesaian teka-teki jembatan Konigsberg dan dengan solusi masalah yang sama (Wirdasari, 2011).

B. Sejarah

1. Masalah di Konigsberg (7 Crossing Point on Progel River)

Euler adalah seorang ahli matematika yang mencoba untuk memecahkan teka-teki jembatan Konigsberg dan lebih

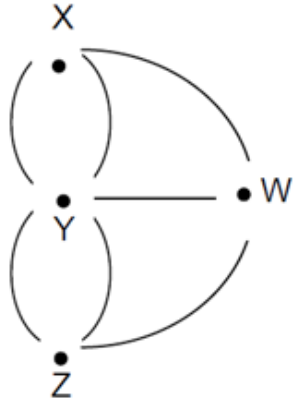
dikenal dengan masalah Jembatan Konigsberg (Wirdasari, 2011). Terdapat tujuh buah jembatan yang dapat menghubungkan dua pulau dan sebuah sungai, seperti yang ditunjukkan pada Gambar 11.1.



Gambar 11.1
Jembatan Konigsberg

2. Urban Planning Problem

Dalam mencari solusi tersebut, Euler mencoba metode dari masalah ini yaitu dengan membentuk model dari jembatan Konigsberg yang dikenal dengan *multigraph*, seperti diperlihatkan pada Gambar 11.2. *Multigraph* tersebut memiliki dua elemen yaitu himpunan verteks (titik/node) dan himpunan *edge* (garis) yang saling menghubungkan garis antar-verteks (Wirdasari, 2011).

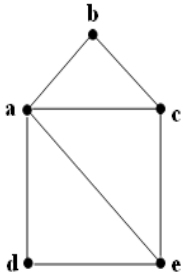


Gambar 11.2
Representasi Multigraph Jembatan Konigsberg

Titik-titik yang diberi label X, Y, Z, dan W pada Gambar 11.2 itulah yang disebut verteks, dan dengan garis saling menghubungkan antartitik itulah yang disebut dengan *edge*.

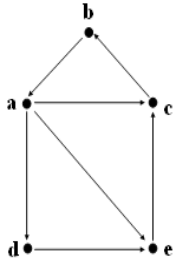
Pada semua *multigraph*, Euler telah membuat sebuah aturan yang dapat dipakai dalam mencari solusi pada jembatan Konigsberg, sehingga aturan ini disebut dengan sebutan *Eulerian path*, yang berbunyi:

> "Andai kita mempunyai sebuah *multigraph* untuk beberapa pasang verteks, sehingga akan terdapat sebuah *path* (lintasan) di antara verteks-verteks tersebut. *Multigraph* tersebut memiliki *eulerian path* dan jika terdapat 0 atau 2 verteks tersebut, maka banyak *edge* yang meninggalkan verteks tersebut akan berjumlah ganjil."



Gambar 11.3
Graph Tak Berarah

Pada *multigraph* jembatan Konigsberg tersebut terdapat empat verteks, dan keempat verteks tersebut memiliki *edge* sehingga meninggalkan verteks yang berjumlah ganjil. Maka, *eulirian path* tersebut tidak dimiliki pada *multigraph* jembatan Konigsberg. *Multigraph* yang ditunjukkan pada Gambar 11.3 tidak memiliki panah, sehingga disebut dengan *undirected graph* (*graph* tak berarah). Adapun yang disebut dengan *directed graph* (*graph* berarah) adalah *multigraph* yang memiliki panah, yang ditujukan pada Gambar 11.4.



Gambar 11.4
Graph Berarah

Definisi 1. Sebuah *simple graph* (*undirected graph*) adalah pasangan dari G = (V, E), di mana:

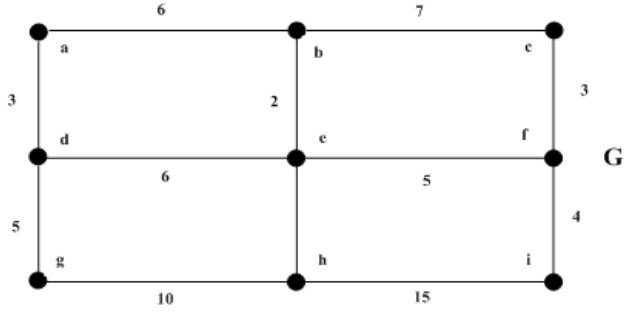
V = himpunan berhingga dari elemen yang disebut verteks

E = sebuah relasi yang irrefleksif dan simetri pada V

Pasangan berurutan pada E disebut *edge* dari *graph* yang berurutan. Lebih spesifik, jika e = (u, v) Î E maka dikatakan bahwa *edge* e adalah antara u dan v (juga antara v dan u), dan dikatakan bahwa u *adjacent* ke v . Lebih jauh, dapat dikatakan bahwa e *incident* ke u (juga v). Karena E simetri, maka kita dapat menotasikan e sebagai pasangan tak berurut {u, v}.

3. Pemecahan oleh Euler

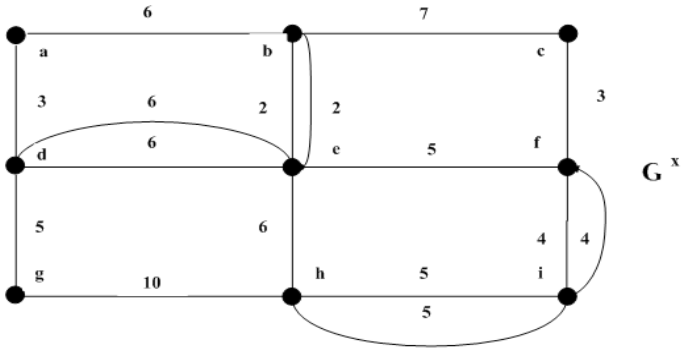
Hasil dari teka-teki jembatan Konigsberg berdampak luar biasa terhadap ilmu pengetahuan. Teka-teki tersebut sangat berguna dan telah membuka jalan bagi terciptanya teorema baru yang disebut teorema *graph***.**



Gambar 11.5
Graph Berbobot dari Teka-teki Tukang Pos Cina

Dari teka-teki tujuh jembatan Konigsberg dihasilkan solusi permasalahan yang didapatkan melalui dianalogikannya setiap jembatan sebagai sisi dan setiap daratan yang diperoleh sebagai simpul pada *graph*, sehingga dapat terbentuk *graph* yang lengkap. Dengan memperhitungkan derajat dalam *graph* dari setiap simpulnya, maka dengan menggunakan metode seperti yang telah diungkapkan dalam pembuktian di atas, kita akan dapat mengetahui apakah *graph* tersebut merupakan suatu lintasan di mana setiap sisi dilalui hanya satu kali (Studi & Informatika, n.d.).

Fakta bahwa teorema *graph* yang dihasilkan oleh Euler telah menyelesaikan masalah berdasarkan teka-teki pada jembatan Konigsberg yang menyatakan hubungan tersendiri antara jaringan spasial (seperti jalur transportasi) pada *graph*. Dalam memodelkan jaringan spasial, ada beberapa tambahan yaitu selain simpul dan sisi, biasanya diberikan sebuah nilai selaku bobot berupa panjang atau satuan ukuran nilai lainnya yang menyatakan kuantitas segmen jalan yang diwakili oleh sisi pada *graph* tersebut. Maka, selanjutnya kita dapat mencari suatu rute pada jaringan spasial tersebut, yaitu menggunakan sarana atau beban yang minimal seperti pada solusi Teka-teki Tukang Pos Cina.



Gambar 11.6
Berat Minimum Matching Sempurna dari Graph Lengkap K

Manfaat mempelajari salah satu aplikasi Teorema Euler yaitu dapat diketahui dari hasil kerja Euler mengenai *graph* yang menjadi salah satu kunci penting dalam keberhasilan penyelesaian berbagai aplikasi masalah yang ada di dunia nyata.

C. Pengertian Graph

Graph adalah kumpulan *node* (simpul) di dalam bidang dua dimensi yang dihubungkan dengan sekumpulan garis (sisi). *Graph* dapat digunakan untuk merepresentasikan objek-objek diskret dan hubungan antara objek-objek tersebut. Representasi visual dari *graph* adalah dengan menyatakan objek sebagai *node*, bulatan, atau titik (*vertex*), sedangkan hubungan antara objek dinyatakan dengan garis (*edge*).

$$G = (V, E)$$

Keterangan:

G = graph

V = simpul atau titik (*vertex* atau *node*)

E = busur atau *edge*, atau *arc*

Jalur pada *graph* dinotasikan sebagai berikut:

$$P = (V0, V1, \ldots, Vn)$$

Keterangan:

P = jalur

Vi = titik jalur

n = jumlah titik jalur

Graph merupakan suatu cabang ilmu yang memiliki banyak terapan. Banyak sekali struktur yang bisa direpresentasikan dengan *graph*, dan banyak masalah yang dapat diselesaikan dengan bantuan *graph*. Seringkali, *graph* digunakan untuk merepresentasikan suatu jaringan. Misalkan, jaringan jalan raya dimodelkan *graph* dengan kota sebagai simpul (*vertex*/*node*) dan jalan yang menghubungkan setiap kotanya sebagai sisi (*edge*) yang bobotnya (*weight*) adalah panjang dari jalan tersebut. Ada beberapa cara untuk menyimpan *graph* di dalam sistem komputer.

Struktur data bergantung pada struktur *graph* dan algoritma yang digunakan untuk memanipulasi *graph*. Secara teori, salah satu dari keduanya dapat dibedakan antara struktur *linked list* dan matriks (*array* dimensi 2). tetapi, dalam penggunaannya, struktur terbaik yang sering digunakan adalah kombinasi keduanya.

D. Istilah pada Graph

Terdapat beberapa istilah yang berkaitan dengan *graph*, antara lain:

1. Vertex

Vertex yaitu himpunan *node*/titik pada sebuah *graph*.

2. Edge

Edge yaitu himpunan garis yang menghubungkan tiap *node*/*vertex*.

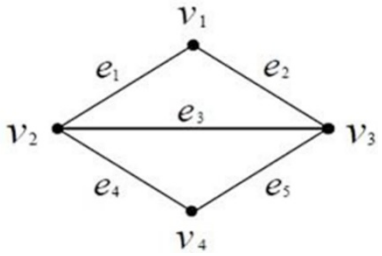
3. Adjacent

Dua buah titik dikatakan berdekatan (*adjacent*) jika keduanya terhubung dengan sebuah sisi.

Contoh:

Sisi e3 = v2v3 *incident* dengan titik v2 dan v3, tetapi sisi e3 = v2v3 tidak *incident* dengan titik v1 dan v4. Titik

v1 *adjacent* dengan titik v2 dan v3, tetapi titik v1 tidak *adjacent* dengan titik v4.



Gambar 11.7
Adjacent pada Graph

4. Weight

Sebuah *graph* G = (V, E) disebut sebuah *graph* berbobot (*weight graph*) apabila terdapat sebuah fungsi bobot bernilai real W pada himpunan E,

$$\mathbf{W : E \rightarrow R}$$

Nilai W(e) disebut bobot untuk sisi e, $\forall$ e $\in$ E. *Graph* berbobot tersebut dinyatakan pula sebagai: G = (V, E, W).

Graph berbobot G = (V, E, W) dapat menyatakan:

a. Suatu sistem perhubungan udara, di mana:
   V = himpunan kota-kota
   E = himpunan penerbangan langsung dari satu kota ke kota lain

W = fungsi bernilai real pada E yang menyatakan jarak atau ongkos atau waktu

b. Suatu sistem jaringan komputer, di mana:

V = himpunan komputer

E = himpunan jalur komunikasi langsung antar dua komputer

W = fungsi bernilai real pada E yang menyatakan jarak atau ongkos atau waktu

5. Path

Path adalah jalur dengan setiap *vertex* berbeda. Contoh: P = D5B4C2A. Sebuah jalur (W) didefinisikan sebagai urutan (tidak nol) *vertex* dan *edge*. Diawali *origin vertex* (*vertex* awal) dan diakhiri *terminus vertex* (*vertex* akhir). Dan, setiap dua garis berurutan adalah series. Contoh: W = A1B3C4B1A2.

6. Cycle

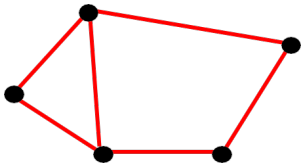
Cycle (siklus) atau *circuit* (sirkuit) merupakan lintasan yang berawal dan berakhir pada simpul yang sama.

E. Jenis-Jenis Graph

Terdapat beberapa jenis *graph*, yaitu:

1. Graph Tak Berarah (Undirected Graph atau Nondirected Graph)

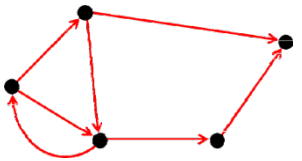
Pada jenis *graph* ini, urutan simpul dalam sebuah busur tidak dipentingkan. Misal, busur e1 dapat disebut busur AB atau BA.



Gambar 11.8
Graph Tak Berarah

2. Graph Berarah (Directed Graph)

Pada jenis *graph* ini, urutan simpul mempunyai arti. Misal, busur AB adalah e1 sedangkan busur BA adalah e8.

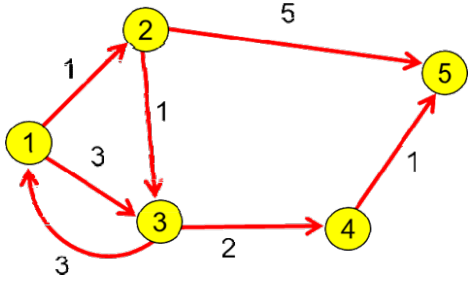


Gambar 11.9
Graph Berarah

3. Graph Berbobot (Weighted Graph)

Jika setiap busur mempunyai nilai yang menyatakan hubungan antara dua buah simpul, maka busur tersebut dinyatakan memiliki bobot. Bobot sebuah busur dapat

menyatakan panjang sebuah jalan dari dua buah titik, jumlah rata-rata kendaraan per hari yang melalui sebuah jalan, dan lain-lain.



Gambar 11.10
Graph Berbobot

F. Representasi Graph dengan Matriks (Array Dimensi 2)

Dalam pemrograman, agar data yang ada dalam *graph* dapat diolah, maka *graph* harus dinyatakan dalam suatu struktur data yang dapat mewakili *graph* tersebut. Dalam hal ini, *graph* perlu direpresentasikan ke dalam bentuk *array* dan dimensi yang sering disebut *matrix*.

Lintasan terpendek merupakan salah satu dari masalah yang dapat diselesaikan dengan *graph*. Jika diberikan sebuah *graph* berbobot, masalah lintasan terpendek adalah bagaimana kita mencari sebuah jalur pada *graph* yang meminimalkan jumlah bobot sisi pembentuk jalur tersebut.

Terdapat beberapa macam persoalan lintasan terpendek, antara lain:

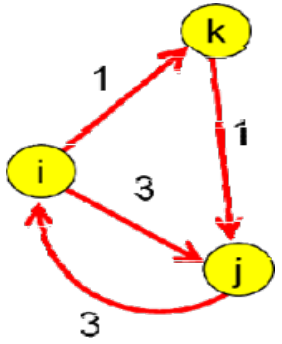
1. Lintasan terpendek antara dua buah simpul tertentu (*a pair shortest path*).
2. Lintasan terpendek antara semua pasangan simpul (*all pairs shortest path*).
3. Lintasan terpendek dari simpul tertentu ke semua simpul yang lain (*single-source shortest path*).
4. Lintasan terpendek antara dua buah simpul yang melalui beberapa simpul tertentu (*intermediate shortest path*).

G. Algoritma Warshall

Algoritma Floyd-Warshall menghitung jarak terpendek (*shortest path*) untuk semua pasangan titik pada sebuah *graph*, dan melakukannya dalam waktu berorde kubik. Algoritma Warshall digunakan untuk menyelesaikan permasalahan jalur terpendek *multipath*.

Algoritma Floyd-Warshall memiliki input *graph* berarah dan berbobot (V,E), yang berupa daftar titik (*node/vertex* V) dan daftar sisi (*edge* E). Jumlah bobot sisi-sisi pada sebuah jalur adalah bobot jalur tersebut. Sisi pada E diperbolehkan memiliki bobot negatif, akan tetapi tidak diperbolehkan bagi *graph* ini untuk memiliki siklus dengan bobot negatif. Algoritma ini menghitung bobot terkecil dari semua jalur yang

menghubungkan sebuah pasangan titik, dan melakukannya sekaligus untuk semua pasangan titik.



Gambar 11.11
Ilustrasi Graph

Berdasarkan ilustrasi pada Gambar 12.10, algoritma melakukan pengecekan apakah beban langsung Q(i,j) memang lebih kecil daripada beban melalui titik perantara Q(i,k)+Q(k,j).

if ((Q(i,k)+Q(k,j))< Q(i, j)) Q(i, j) ← Q(i,k)+Q(k,j)

Pada contoh kasus Gambar 12.2, langkah-langkah penyelesaian dengan algoritma Warshall adalah sebagai berikut:

1. Representasi matriks beban di bawah ini menjadi *array* dua dimensi Q.

	1	2	3	4	5
1		1	3	-	-
2	-		1	-	5
3	3	-		2	-
4	-	-	-		1
5	-	-	-	-	

⇨

Q	1	2	3	4	5
1	M	1	3	M	M
2	M	M	1	M	5
3	3	M	M	2	M
4	M	M	M	M	1
5	M	M	M	M	M

Di mana M adalah *big integer*.

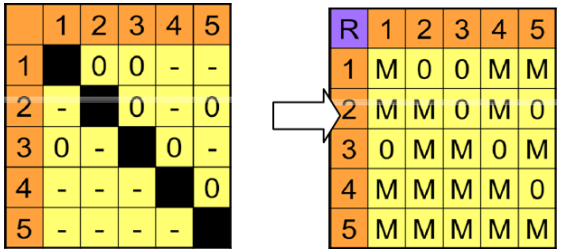
2. Representasi matriks jalur di bawah ini menjadi *array* dua dimensi P.

	1	2	3	4	5
1		√	√	-	-
2	-		√	-	√
3	√	-		√	-
4	-	-	-		√
5	-	-	-	-	

⇨

P	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	0	1
3	1	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	0

3. Representasi matriks rute di bawah ini menjadi *array* dua dimensi.



	1	2	3	4	5
1		0	0	-	-
2	-		0	-	0
3	0	-		0	-
4	-	-	-		0
5	-	-	-	-	

R	1	2	3	4	5
1	M	0	0	M	M
2	M	M	0	M	0
3	0	M	M	0	M
4	M	M	M	M	0
5	M	M	M	M	M

Di mana M adalah *big integer*.

4. Melakukan pengecekan beban langsung Q(1,3) = 3 dengan beban tak langsung.

$$Q(1,1) + Q(1,3) = M+3$$

$$Q(1,2) + Q(2,3) = 2$$

$$Q(1,3) + Q(3,3) = 3+M$$

$$Q(1,4) + Q(4,3) = M+M$$

$$Q(1,5) + Q(5,3) = M+M$$

Sehingga, beban terkecil adalah Q(1,3) = 2.

Algoritma Warshall untuk beban adalah sebagai berikut:

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to
        n
            if ((Q(i,k) + Q(k,j)) < Q(i,j))
                Q(i,j) ← (Q(i,k)+Q(k,j)
```

Algortitma Warshall untuk jalur adalah sebagai berikut:

```
for k = 1 to n
    for i = 1 to n
        for j = 1 to n

P(i,j) ← P(i,j) OR (P(i,k) AND P(k,j))
```

Algoritma Warshall untuk rute adalah sebagai berikut:

```
for k = 1 to n
  for i = 1 to n
    for j = 1 to
    n
        if ((Q(i,k) + Q(k,j)) <
          Q(i,j)) if (R(k,j) = 0)
            R(i,j) ← k
          else
            R(i,j)=R(k,j)
```


Latihan 1. Mendeklarasikan Matriks Beban, Jalur, dan Rute

```
#include<stdio.h>

#define N 5
#define M 1000


void Tampil(int data[N][N], char *judul)
{
      printf("%s = \n",judul); for(int
      i=0; i<N; i++) {
                  for(int j=0; j<N; j++)
                        if(data[i][j] >= M)
                                    printf("M ");
                        else
                                    printf("%d ", data[i][j]);
                  printf("\n");
      }
}

void main()
{
            int Beban[N][N] = {M,1,3,M,M,
                                    M,M,1,M,5,
                                    3,M,M,2,M,
                                       M,M,M,M,1,
                                       M,M,M,M,M};

                  int Jalur[N][N] = {0,1,1,0,0,
                                       0,0,1,0,1,
                                       1,0,0,1,0,
                                       0,0,0,0,1,
                                       0,0,0,0,0};

               int Rute[N][N] = {M,0,0,M,M,
                                    M,M,0,M,0,
                                    0,M,M,0,M,
                                       M,M,M,M,0,
                                       M,M,M,M,M};

      Tampil(Beban,      "Beban");
      Tampil(Jalur,           "Jalur");
      Tampil(Rute, "Rute");
}
```

Latihan 2. Algoritma Warshall untuk Pencarian Jalur Terpendek Multipath

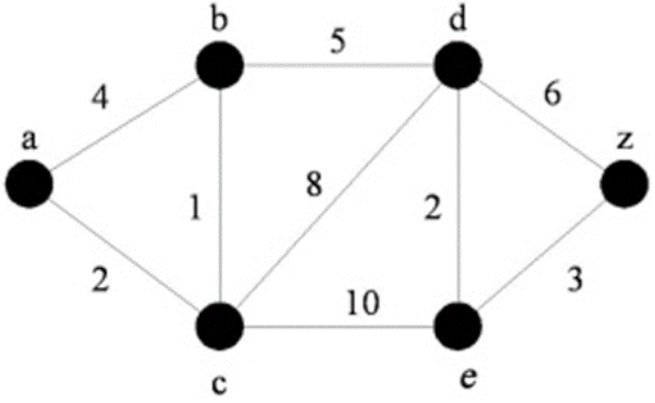
```
#include<stdio.h>

#define N 5
#define M 1000


void Tampil(int data[N][N], char *judul)
{
      printf("%s = \n",judul); for(int
      i=0; i<N; i++) {
                              for(int j=0; j<N; j++)
                                  if(data[i][j] >= M)
                                        printf("M ");
                        else
                                    printf("%d ", data[i][j]); printf("\n");
      }
}

void Warshall(int Q[N][N], int P[N][N], int R[N][N])
{
      for(int k=0; k<N; k++)
                for (int i=0; i<N; i++)
                          for (int j=0; j<N; j++){
P[i][j] = P[i][j] | (P[i][k] & P[k][j]);
                                    if ((Q[i][k] + Q[k][j]) < Q[i][j]) {
                                      Q[i][j] = Q[i][k] + Q[k][j]; if (R[k][j] ==
                                          0)
                                              R[i][j] = k+1;
                                          else
                                              R[i][j] = R[k][j];
                                    }
                              }
}
```



```
void main()
{
                int Beban[N][N] = {M,1,3,M,M,
                                   M,M,1,M,5,
                                   3,M,M,2,M,
                                  M,M,M,M,1,
                                  M,M,M,M,M};

                    int Jalur[N][N] = {0,1,1,0,0,
                                       0,0,1,0,1,
                                       1,0,0,1,0,
                                       0,0,0,0,1,
                                      0,0,0,0,0};

                int Rute[N][N] = {M,0,0,M,M,
                                   M,M,0,M,0,
                                   0,M,M,0,M,
                                  M,M,M,M,0,
                                  M,M,M,M,M};

      Tampil(Beban, "Beban"); Tampil(Jalur,
      "Jalur"); Tampil(Rute, "Rute");
      Warshall(Beban, Jalur, Rute);
      printf("Matriks setelah Algoritma Warshall : \n"); Tampil(Beban,
      "Beban");
      Tampil(Jalur, "Jalur");
      Tampil(Rute, "Rute");
}
```

Tugas

Berdasarkan *graph* di bawah ini, representasikan matriks. Gunakan algoritma Warshall untuk mencari rute terpendek.



Daftar Pustaka

Jurnal:

Alfatwa, Dean Fathony, Eriek Rahman Syah P., dan Fahris Mumtaza Ahsan, "Implementasi Algoritma Radix Sort dalam Berbagai Kasus Bilangan Dibandingkan Algoritma Pengurutan yang Lain", Departemen Teknik Informatika, Institut Teknologi Bandung.

Buku/Diktat:

Munir, Rinaldi. 2005. *Diktat Kuliah IF2251 Strategi Algoritmik*. Bandung: Laboratorium Ilmu dan Rekayasa Komputasi, Departemen Teknik Informatika, Institut Teknologi Bandung.

Website:

http://elektro.um.ac.id/wp-content/uploads/2016/04/Struktur-Data-Modul-Praktikum-11-Hashing-Table.pdf, diakses pada 25 April 2020.

http://hackerunikan.blogspot.com/2014/02/kelebihan-dan-kekurangan-gambaran.html, diakses pada 20 April 2020.

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/Makalah/MakalahStmik06.pdf, diakses pada 27 April 2020.

https://informatika11d.wordpress.com/2012/11/22/struktur-data-hash-table/, diakses pada 25 April 2020.

https://mti.binus.ac.id/2018/03/05/teori-graph-sejarah-dan-manfaatnya/, diakses pada 1 Mei 2020.

https://nofwan.blogspot.com/2016/12/pengenalan-struktur-data-sebelum-ngoding.html, diakses pada 22 April 2020.

https://www.researchgate.net/publication/259289777_Teori_dan_Aplikasi_Struktur_Data_Menggunakan_C, diakses pada 20 April 2020.

Paul E. Black et. al., http://www.nist.gov, diakses pada 30 Mei 2020 pukul 22:00 WIB.

Modul:

Modul Praktikum Struktur Data Politeknik Elektronika Negeri Surabaya.

Daftar Istilah

Algoritma
Langkah-langkah menyelesaikan suatu masalah yang disusun secara logis dan berurutan.

Animasi
Gambar yang tampak bergerak, terdiri dari banyak gambar tunggal (disebut *frame*) yang ditampilkan satu per satu secara bergantian dengan cepat, sehingga objek dalam gambar terlihat seolah-olah bergerak.

Anting-Anting (Pendant Vertex)
Simpul graf yang berderajat.

Array
Struktur data yang memiliki banyak elemen di dalamnya, dengan masing-masing elemen memiliki tipe data yang sama.

Clear
Menghapus secara keseluruhan, biasanya digunakan sebagai nama fungsi/metode yang bertujuan untuk mengosongkan *list* atau menghapus keseluruhan elemen.

Console
Istilah dalam komputer yang menunjuk pada antarmuka antara pemakai dengan komputer yang berbasiskan teks. Cara kerja konsol sangat sederhana yaitu menggunakan standar *input* untuk membaca *input* dari *keyboard* dan standar *output* untuk menampilkan teks ke layer monitor.

Daftar/Senarai Ketetanggaan (Adjacency List)
Daftar yang menunjukkan hasil enumerasi simpul yang bertetangga dengan simpul lainnya pada suatu graf.

Data
Informasi yang disimpan oleh komputer, dapat berbentuk teks, gambar, suara, video, dan sebagainya.

Delete
Menghapus sebuah elemen, biasanya digunakan sebagai nama fungsi/metode yang bertujuan untuk menghapus sebuah elemen dalam suatu *list*/*tree*.

Deret Geometric
Deretan bilangan yang setiap bilangan merupakan hasil kali bilangan sebelumnya dengan suatu konstanta.

Destruktor
Metode khusus dalam sebuah kelas untuk menghapus objek hasil instansiasi kelas tersebut.

Diagram Sirkulasi
Diagram yang dibuat arsitek untuk menganalisis arus pengunjung atau mengatur tata letak ruangan dalam gedung besar.

Definisi Graf
Pasangan tak berurutan yang terdiri dari himpunan tak kosong berupa himpunan titik/simpul (*vertex*) dan himpunan boleh kosong berupa himpunan sisi (*edge*).

Derajat Masuk (In-Degree) dan Derajat Keluar (Out-Degree)
Jumlah busur yang masuk ~ keluar suatu simpul pada graf berarah.

Derajat Simpul (Vertex Degree)
Jumlah sisi yang bersisian/keluar dari simpul.

Dimensi
Jumlah indeks yang diperlukan untuk menyatakan sebuah elemen dalam *array*.

Edge Induced Subgraph
Subgraf yang himpunan titiknya adalah titik-titik ujung sisi pada himpunan sisi subset dari graf.

Elemen
Sebuah data tunggal yang paling kecil dari sebuah *array* atau *list*. Data tunggal di sini tidak perlu data sederhana, melainkan bisa berupa kumpulan data atau *list* yang lain.

Empty
Keadaan di mana *list* berada dalam keadaan kosong.

Fibonacci
Barisan bilangan yang setiap bilangan merupakan jumlah dari dua bilangan sebelumnya.

Field
Data yang dimiliki oleh sebuah objek.

FIFO
First in first out; sifat suatu kumpulan data. Jika sebuah elemen A dimasukkan lebih dulu dari B, maka A harus dikeluarkan dulu dari B.

FPB
Faktor persekutuan terbesar; faktor yang paling besar jika sejumlah bilangan memiliki beberapa faktor yang sama.

Full
Keadaan di mana *list* penuh dan tidak boleh menerima data lagi.

Fungsi
Suatu modul atau bagian program yang mengerjakan suatu program tertentu.

Gelang/Kalang (Loop atau Self-Loop atau Buckle)
Sisi yang titik ujungnya sama.

Graf Berarah (Directed Graph)
Graf yang sisinya mengandung orientasi arah. Sisinya disebut busur (*arc*).

Graf Beraturan-r (Regular-r Graph)
Graf yang semua titiknya berderajat.

Graf Berbobot (Weighted Graph)
Graf yang setiap sisinya diberi bobot/nilai.

Graf Bidang (Plane Graph)
Representasi graf planar yang digambarkan dengan sisi-sisinya tidak saling berpotongan.

Graf Bipartisi (Bipartite Graph)
Graf yang himpunan titiknya dapat dipartisi menjadi beberapa bagian.

Graf Bipartisi Komplet (Complete Bipartite Graph)
Graf bipartisi dengan himpunan partisi dan masing-masing titik didihubungkan dengan masing-masing titik oleh tepat satu sisi.

Graf Ganda/Rangkap (Multiple Graph)
Graf yang mengandung sisi rangkap (tidak boleh memuat gelang).

Graf Hamilton (Hamiltonian Graph)
Graf yang memuat sikel Hamilton.

Graf Komplet/Graf Lengkap (Complete Graph)
Graf sederhana dengan setiap pasang titik yang berbeda dihubungkan oleh satu sisi.

Graf Kosong (Null/Empty Graph)
Graf yang tidak memiliki sisi.

Graf Lingkaran (Circle Graph)
Graf sederhana yang semua simpulnya berderajat dua.

Graf Lintasan
Graf yang hanya memuat satu lintasan.

Graf Planar (Planar Graph)
Graf yang dapat digambar pada bidang datar dengan sisi-sisi yang tidak saling berpotongan.

Graf Sederhana (Simple Graph)
Graf yang tidak memuat sisi rangkap maupun gelang (*loop*).

Graf Semu/Graf Palsu/Pseudograf (Pseudograph)
Graf yang mengandung gelang (jika ada sisi rangkap, juga disebut demikian).

Graf Tak Berarah (Undirected Graph)
Graf yang sisinya tidak mengandung orientasi arah (berupa garis saja).

Graf Tak Sederhana (Unsimple Graph)
Graf yang memuat sisi rangkap atau gelang (*loop*).

Graf Terhubung (Connected Graph)
Graf dengan setiap pasang simpulnya mengandung lintasan.

Graf Trivial (Trivial Graph)
Graf yang hanya memiliki satu titik/simpul (tanpa sisi).

Himpunan
Kumpulan dari objek-objek, misalnya sebuah himpunan dari buah-buahan dapat terdiri dari pisang, mangga, jambu, dan lain-lain.

Indeks
Bilangan yang digunakan untuk menyatakan posisi suatu elemen dalam *array* atau *list*.

Input
Data masukan; dalam fungsi berarti parameter yang dimasukkan, sedangkan dalam program secara keseluruhan berarti data yang dimasukkan oleh pemakai, bisa melalui parameter program, *file*, maupun *keyboard*.

Insert
Memasukkan sebuah elemen baru ke dalam *list*. Biasanya *insert* dilakukan baik di tengah-tengah, awal, maupun akhir *list*.

Iterasi
Perulangan dengan struktur perulangan, *while*, *do while*, maupun *for*.

Jalan (Walk)
Barisan simpul dan sisi yang saling terhubung pada suatu graf.

Jalan Trivial (Trivial Walk)
Jalan yang tidak memuat sisi.

Jalur/Jejak (Trail)
Jalan yang semua sisinya berbeda.

Jembatan/Sisi Potong(Cut Set/Bridge)
Sisi graf terhubung yang bila dihilangkan menyebabkan graf menjadi tidak terhubung.

Kardinalitas Graf
Jumlah simpul/titik pada graf.

Kelas
Suatu struktur yang digunakan sebagai *template* bagi objek-objek yang sama sifatnya.

Kompilasi
Proses menerjemahkan bahasa sumber (*source code*) ke dalam bahasa lain, biasanya bahasa mesin, untuk dapat dijalankan langsung oleh komputer melalui sistem operasi.

Kompiler
Program yang mengerjakan kompilasi.

Konstruktor
Metode khusus yang dimiliki suatu kelas untuk membentuk suatu objek baru berdasarkan kelas tersebut.

Lema Jabat Tangan (Handshaking Lemma)
Jumlah semua derajat simpul pada suatu graf sama dengan dua kali jumlah sisinya.

Library
Kumpulan fungsi, makro, *template*, dan kelas yang disediakan bersama kompiler C++.

LIFO
Last in first out; sifat kumpulan data, kebalikan dari FIFO.

Lilitan (Girth)
Panjang sikel terpendek pada graf.

Linked List
List yang didesain dengan cara mendefinisikan sebuah elemen yang memiliki hubungan atau *link* dengan elemen lain yang dihubungkan dengan elemen yang lain lagi.

Lintasan (Path)
Jalan yang semua titiknya berbeda.

Lintasan Euler (Euler Path)
Lintasan yang memuat semua sisi di graf.

Lintasan Hamilton (Hamiltonian Path)
Lintasan yang melalui tiap simpul di dalam graf tepat satu kali.

Matriks
Dalam matematika berarti kumpulan bilangan yang disusun dalam bentuk kolom dan baris.

Matriks Keterkaitan/Bersisian (Incidency Matrix)
Matriks yang merepresentasikan sisi yang bersisian dengan setiap pasang simpul dari suatu graf.

Matriks Ketertanggaan (Adjacency Matrix)
Matriks yang merepresentasikan banyaknya sisi yang menghubungkan setiap dua simpul dari suatu graf.

Metode
Fungsi yang dimiliki suatu objek.

Objek
Struktur data yang terdiri dari data yang lebih sederhana yang disebut *field*, yang memiliki operasi sendiri untuk menangani data-data yang dimilikinya.

Output
Data yang dihasilkan oleh program.

Pohon (Tree)
Graf terhubung yang tidak memuat sikel.

Pointer
Tipe data khusus yang umumnya berukuran 32 bit, yang berfungsi menampung bilangan tertentu yang menunjuk pada lokasi memori tertentu.

Pop
Mengeluarkan satu elemen dari dalam *list* dengan cara menyalin data elemen tersebut, kemudian menghapus elemen tersebut dari *list*, biasanya digunakan untuk *stack*.

Prima
Bilangan yang tidak memiliki faktor selain 1 dan bilangan itu sendiri.

Push
Memasukkan sebuah elemen baru ke dalam *list*.

Queue
Struktur *list* dengan sifat FIFO, cara kerjanya seperti antrean manusia.

Record
Struktur data yang terdiri dari satu atau lebih elemen yang tipe datanya bisa berbeda.

Rekursi
Jenis perulangan yang tidak menggunakan struktur perulangan, tetapi dengan memanggil fungsi yang bersangkutan.

Sikel (Cycle)
Sirkuit yang titik dalamnya berbeda.

Sikel Hamilton (Hamiltonian Cycle)
Sikel yang memuat semua titik dari suatu graf.

Simpul Asal (Initial Vertex) dan Simpul Terminal (Terminal Vertex)
Istilah khusus untuk simpul awal dan simpul akhir pada graf berarah.

Simpul Terpencil (Isolated Vertex)
Simpul yang tidak memiliki sisi yang bersisian dengannya.

Sisi Rangkap/Sisi Ganda (Multiple Edges)
Dua sisi atau lebih yang menghubungkan sepasang titik.

Sirkuit (Circuit) atau Trail Tertutup
Jalan tertutup yang semua sisinya berbeda.

Sirkuit Euler (Euler Circuit)
Sirkuit yang memuat semua sisi di graf.

Sirkuit Hamilton (Hamiltonian Circuit)
Lintasan Hamilton yang kembali ke titik asal sehingga membentuk lintasan tertutup.

Sisi Disjoin
Dua subgraf dari graf yang tidak memiliki sisi yang sama.

Sort
Menyusun elemen-elemen suatu *list* secara berurutan.

Source Code
Program yang ditulis menggunakan bahasa pemrograman tertentu. Kode sumber belum dapat dijalankan oleh komputer dan perlu menjalani proses kompilasi sehingga dapat dijalankan.

Stack
List yang memiliki sifat LIFO. Data yang hendak dikeluarkan dari *stack* haruslah data yang paling terakhir dari *stack*.

STL
Standar template library; merupakan kumpulan yang disertakan dalam setiap kompiler C++ yang memenuhi standar ANSI C++, yang menyediakan berbagai struktur data, algoritma, dan *template* yang sering dipakai.

Stream
Aliran; merupakan konsep dalam C++ untuk *input* dan *output* data tanpa mempedulikan isi data maupun media penampung data tersebut.

Struktur Control
Struktur yang digunakan untuk mengontrol jalannya program.

Subgraf Merentang (Spanning Subgraph)
Subgraf yang mengandung semua titik dari graf induknya.

Subgraf/Upagraf/Graf Bagian (Subgraph)
Graf yang himpunan titik dan sisinya merupakan himpunan bagian dari graf yang lain.

Teks
Data yang terdiri dari karakter-karakter yang dapat dibaca (huruf, bilangan, tanda baca).

Terhubung Langsung/Bertetangga (Adjacent)
Dua buah titik pada graf tak berarah dikatakan terhubung langsung/bertetangga jika kedua titik itu dihubungkan oleh sebuah sisi.

Terkait/Bersisian (Incident)
Untuk sembarang sisi yang menghubungkan dua titik pada graf, maka sisi itu dikatakan terkait/bersisian dengan dua titik itu.

Titik Dalam (Internal Vertex)
Titik selain titik ujung pada suatu jalan.

Titik Disjoin
Dua subgraf dari graf yang tidak memiliki titik yang sama.

Titik Potong (Cut Vertex)
Titik graf terhubung yang bila dihilangkan menyebabkan graf menjadi tidak terhubung.

Tree
Suatu struktur data yang setiap elemennya terhubung sedemikian rupa sehingga berbentuk seperti pohon.

Vertex Induced Subgraph
Subgraf yang himpunan titik dan himpunan sisinya beranggotakan semua sisi yang mempunyai titik ujung.