

MODUL PRAKTIKUM

STRUKTUR DATA

S1 TEKNOLOGI INFORMASI



LEMBAR PENGESAHAN

Saya yang bertanda tangan di bawah ini:

Nama : Febriyanti Sthevanie, S.T., M.T.

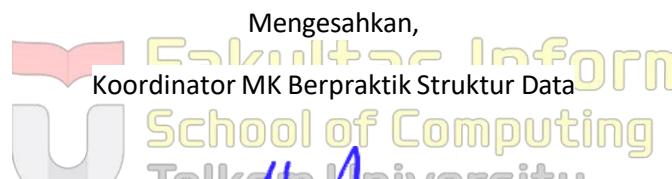
NIP : 14880014

Koordinator Mata Kuliah : Struktur Data

Program Studi : S1 Teknologi Informasi

Menerangkan dengan sesungguhnya bahwa modul ini digunakan untuk pelaksanaan praktikum di Semester Ganjil Tahun Ajaran 2024/2025 di Laboratorium Informatika Fakultas Informatika Universitas Telkom.

Bandung, 24 Agustus 2024



Mengesahkan,
Koordinator MK Berpraktik Struktur Data

S. Th.
Febriyanti Sthevanie, S.T., M.T.

NIP. 14880014

Mengetahui,
Kepala Program Studi S1 Teknologi Informatika

Rio Guntur Utomo, S.T., M.T., Ph.D.

NIP. 00760045

PERATURAN PRAKTIKUM

LABORATORIUM INFORMATIKA 2023/2024

1. Praktikum diampu oleh dosen kelas dan dibantu oleh asisten laboratorium dan asisten praktikum.
2. Praktikum dilaksanakan di Gedung TULT (Telkom University Landmark Tower) lantai 6 dan 7 sesuai jadwal yang ditentukan.
3. Praktikan wajib membawa modul praktikum, kartu praktikum, dan alat tulis.
4. Praktikan wajib mengecek kehadiran di igracias dan sheet yang dibagikan asisten.
5. Durasi kegiatan praktikum S-1 = 2 jam (100 menit).
6. Jumlah pertemuan praktikum:
 - 16 kali pertemuan
7. Praktikan wajib hadir minimal 75% dari seluruh pertemuan praktikum di lab.
8. Praktikan yang datang terlambat :
 - <= 5 menit : diperbolehkan mengikuti praktikum tanpa tambahan praktikuM
 - >= 30 menit : tidak diperbolehkan mengikuti praktikum
9. Saat praktikum berlangsung, asisten praktikum dan praktikan:
 - Wajib menggunakan seragam sesuai aturan institusi.
 - Wajib mematikan/ mengkondisikan semua alat komunikasi.
 - Dilarang membuka aplikasi yang tidak berhubungan dengan praktikum yang berlangsung.
 - Dilarang mengubah pengaturan *software* maupun *hardware* komputer tanpa ijin.
 - Dilarang membawa makanan maupun minuman di ruang praktikum.
 - Dilarang memberikan jawaban ke praktikan lain.
 - Dilarang menyebarkan soal praktikum.
 - Dilarang membuang sampah di ruangan praktikum.
 - Wajib meletakkan alas kaki dengan rapi pada tempat yang telah disediakan.
10. Setiap praktikan dapat mengikuti praktikum susulan maksimal dua modul untuk satu mata kuliah praktikum.
 - Praktikan yang dapat mengikuti praktikum susulan hanyalah praktikan yang memenuhi syarat sesuai ketentuan institusi, yaitu: sakit (dibuktikan dengan surat keterangan medis), tugas dari institusi (dibuktikan dengan surat dinas atau dispensasi dari institusi), atau mendapat musibah atau kedukaan (menunjukkan surat keterangan dari orangtua/wali mahasiswa.)
 - Persyaratan untuk praktikum susulan diserahkan sesegera mungkin kepada asisten laboratorium untuk keperluan administrasi.
 - Praktikan yang diijinkan menjadi peserta praktikum susulan ditetapkan oleh Lab Informatika dan tidak dapat diganggu gugat.
11. Ketidakhadiran pada kelas praktikum:
 - **Nilai Modul = 0**
12. Meminta, mendapatkan, dan menyebarluaskan soal dan atau kunci jawaban praktikum:
 - **Penyebar soal dan kunci jawaban: Pengajuan sanksi kepada Komisi Disiplin Fakultas**
 - **Penerima soal dan kunci jawaban: Nilai '0' pada (seluruh assessment) praktikum**
13. Lupa menghapus file praktikum:
 - **Pengurangan nilai modul 20%**
14. Memicu kegaduhan, sehingga membuat situasi tidak kondusif (jalan-jalan, mengganggu teman, mengobrol, dll), asisten praktikum diwajibkan menegur sebanyak 3x
 - **Pengurangan nilai modul 50%**
15. Menyalahgunakan fitur lms
 - **Siap menerima sanksi lebih lanjut dari IFLAB**

TATA CARA KOMPLAIN PRAKTIKUM IFLAB MELALUI IGRACIAS

1. Login IGracias
2. Pilih Menu **Masukan dan Komplain**, pilih *Input Tiket*



3. Pilih Fakultas/Bagian: **Bidang Akademik (FIF)**
4. Pilih Program Studi/Urusan: **Urusan Laboratorium/Bengkel/Studio (FIF)**
5. Pilih Layanan: **Praktikum**
6. Pilih Kategori: **Pelaksanaan Praktikum**, lalu pilih **Sub Kategori**.
7. Isi **Deskripsi** sesuai komplain yang ingin disampaikan.

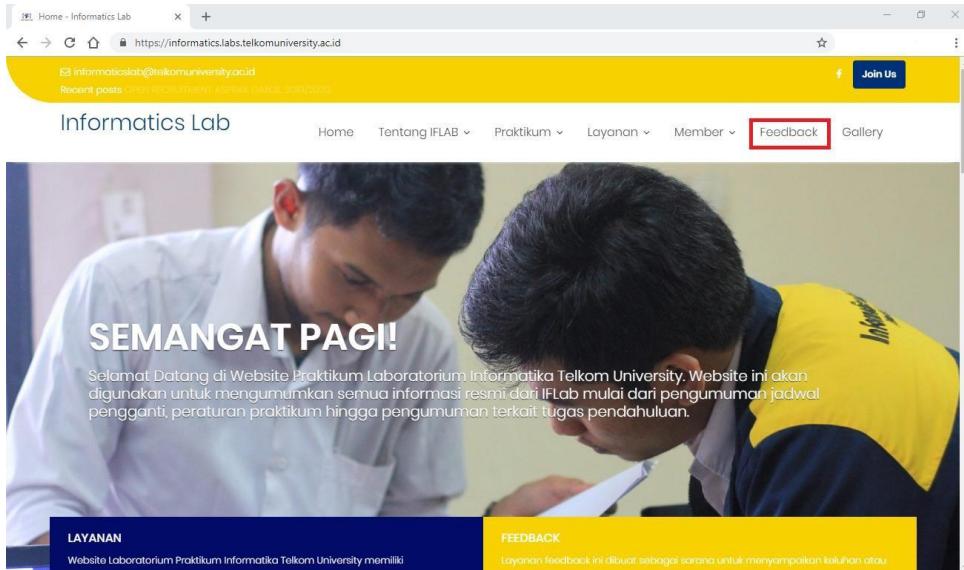
The screenshot shows the 'Input Keluhan' (File Complaint) form with the following fields filled:

Fakultas / Bagian	: BIDANG AKADEMIK (FIF)
Program Studi / Urusan	: URUSAN LABORATORIUM/BENGKEL/STUDIO (FIF)
Pelapor	: RIZQILLAH ZAHRA LESTARI
Layanan	: PRAKTIKUM
Kategori	: Pelaksanaan Praktikum
Sub Kategori	: Please Select...
Tipe Masukan	: <input checked="" type="radio"/> Komplain <input type="radio"/> Masukan
Deskripsi	: [Text area with rich text editor toolbar]

Lampirkan *file* jika perlu. Lalu klik Kirim.

TATA CARA KOMPLAIN PRAKTIKUM IFLAB MELALUI WEBSITE

1. Buka website <https://informatics.labs.telkomuniversity.ac.id/> melalui browser.
2. Pilih menu **Feedback** pada navigation bar website.



3. Pilih tombol **Link Form Feedback**.



4. Lakukan *login* menggunakan akun **SSO Telkom University** untuk mengakses *form feedback*.
5. Isi *form* sesuai dengan *feedback* yang ingin diberikan.

DAFTAR ISI

LEMBAR PENGESAHAN	i
PERATURAN PRAKTIKUM LABORATORIUM INFORMATIKA 2023/2024	ii
TATA CARA KOMPLAIN PRAKTIKUM IFLAB MELALUI IGRCIAS.....	iii
TATA CARA KOMPLAIN PRAKTIKUM IFLAB MELALUI WEBSITE.....	iv
DAFTAR ISI	v
DAFTAR GAMBAR	viii
Modul 1 CODE BLOCKS IDE & PENGENALAN BAHASA C++ (BAGIAN PERTAMA).....	12
1.1 Pengenalan Code Blocks.....	12
1.1.1 Instalasi Code Blocks	12
1.1.2 Cara Menggunakan <i>Code Blocks</i>	12
1.2 Sekilas tentang C++.....	16
1.3 Dasar Pemrograman.....	17
1.3.1 Struktur Program C++.....	17
1.3.2 Pengenal (<i>Identifier</i>)	17
1.3.3 Tipe data dasar	17
1.3.4 Variabel	18
1.3.5 Konstanta	18
1.4 Input / Output	18
1.4.1 Output	18
1.4.2 Input	20
1.5 Operator	21
1.6 Pemodifikasi Tipe	24
1.6.1 Unsigned.....	24
1.6.2 Short	24
1.6.3 Long	24
1.7 Kondisional	25
1.7.1 Bentuk 1	25
1.7.2 Bentuk 2	25
1.7.3 Bentuk 3	26
1.8 Perulangan.....	27
1.8.1 Perulangan dengan <i>for</i> dan <i>while</i>	27
1.8.2 Perulangan dengan <i>do ... while</i>	28
1.9 Struktur	29
1.10 Blok Program	30
1.11 Latihan.....	31
Modul 2 PENGENALAN BAHASA C++ (BAGIAN KEDUA).....	32
2.1 Array	32
2.1.1 <i>Array</i> Satu Dimensi	32
2.1.2 <i>Array</i> Dua Dimensi.....	32
2.1.3 <i>Array</i> Berdimensi Banyak	33
2.2 Pointer	33

2.2.1	Data dan Memori	33
2.2.2	<i>Pointer</i> dan Alamat.....	34
2.2.3	Pointer dan Array	35
2.2.4	Pointer dan String.....	36
2.3	Fungsi	38
2.4	Prosedur.....	39
2.5	Parameter Fungsi	39
2.5.1	Paramater Formal dan Parameter Aktual.....	39
2.5.2	Cara melewatkkan Parameter.....	40
Modul 3	ABSTRACT DATA TYPE (ADT)	43
3.1	Abstract Data Type (ADT)	43
3.2	Latihan.....	45
Modul 4	SINGLE LINKED LIST (BAGIAN PERTAMA)	47
4.1	Linked List dengan Pointer	47
4.2	Single Linked List	48
4.2.1	Pembentukan Komponen-Komponen <i>List</i>	49
4.2.2	Insert	50
4.2.3	View.....	53
4.3	Latihan.....	54
4.4	Delete	55
4.5	Update	57
Modul 5	SINGLE LINKED LIST (BAGIAN KEDUA)	58
5.1	Searching	58
5.2	Latihan.....	60
Modul 6	DOUBLE LINKED LIST (BAGIAN PERTAMA)	62
6.1	Double Linked List	62
6.1.1	Insert	63
6.1.2	Delete	65
6.2	Latihan.....	68
Modul 7	STACK	70
7.1	Pengertian Stack.....	70
7.2	Komponen-Komponen dalam Stack.....	70
7.3	Operasi-Operasi dalam Stack	71
7.3.1	Push	71
7.3.2	Pop.....	72
7.4	Primitif-Primitif dalam Stack.....	72
7.5	Stack (Representasi Tabel)	73
7.5.1	Operasi-operasi Dalam <i>Stack</i>	73
7.5.2	Primitif-primitif Dalam <i>Stack</i>	74
7.6	Latihan Stack	75
Modul 8	QUEUE	77
8.1	Pengertian Queue	77
8.2	Operasi-Operasi dalam Queue	77

8.2.1	Insert (Enqueue)	78
8.2.2	Delete (Dequeue)	78
8.3	Primitif-Primitif dalam Queue	79
8.4	Queue (Representasi Tabel)	81
8.4.1	Pengertian	81
8.4.2	Primitif-Primitif dalam Queue	85
8.5	Latihan Queue	86
Modul 9	ASSESSMENT CLO 1 (SLL & DLL)	87
Modul 10	TREE (BAGIAN PERTAMA)	88
10.1	Pengertian Rekursif	88
10.2	Kriteria Rekursif	89
10.3	Kekurangan Rekursif.....	89
10.4	Contoh Rekursif.....	89
10.5	Pengertian Tree	90
10.6	Jenis-Jenis Tree	92
10.6.1	Ordered Tree	92
10.6.2	Binary Tree	92
10.7	Operasi-Operasi dalam Binary Search Tree	93
10.8	Traversal pada Binary Tree	95
10.9	Latihan.....	97
Modul 11	TREE (BAGIAN KEDUA)	100
Modul 12	ASISTENSI TUGAS BESAR	102
Modul 13	MULTI LINKED LIST	103
13.1	Multi Linked List	103
13.1.1	Insert	103
13.1.2	Delete	105
13.2	Latihan.....	110
Modul 14	GRAPH	113
14.1	Pengertian	113
14.2	Jenis-Jenis Graph	113
14.2.1	Graph Berarah (Directed Graph)	113
14.2.2	Graph Tidak Berarah (Undirected Graph).....	117
14.3	Latihan.....	121
Modul 15	ASSESSMENT CLO 2 (STACK & QUEUE)	122
DAFTAR PUSTAKA	123

DAFTAR GAMBAR

Gambar 1-1 Code Blocks IDE	12
Gambar 1-2 Membuat <i>project</i> baru	13
Gambar 1-3 Menulis sintak	13
Gambar 1-4 Membuat <i>class</i> baru	13
Gambar 1-5 Centang all in build target	14
Gambar 1-6 <i>Target options</i>	14
Gambar 1-7 Contoh <i>file</i>	15
Gambar 1-8 <i>Clean</i>	15
Gambar 1-9 <i>Error Message</i>	16
Gambar 1-10 <i>Hello World</i>	17
Gambar 1-11 Penentu Lebar <i>Field</i>	19
Gambar 1-12 <i>Output cout</i> dengan Penentu Lebar <i>Field</i>	20
Gambar 1-13 <i>Increment</i> di Belakang	24
Gambar 1-14 <i>Increment</i> di Depan	24
Gambar 1-15 Contoh	31
Gambar 1-16 <i>Mirror</i>	31
Gambar 2-1 Ilustrasi Array Dua Dimensi	32
Gambar 2-2 Ilustrasi <i>Memory</i>	33
Gambar 2-3 Ilustrasi Alokasi <i>Memory</i>	33
Gambar 2-4 Ilustrasi Alokasi <i>Pointer</i>	34
Gambar 2-5 <i>Output Pointer</i>	35
Gambar 2-6 <i>Array</i>	35
Gambar 2-7 <i>Pointer</i> dan <i>Array</i>	35
Gambar 2-8 <i>Pointer</i> dan <i>String</i>	38
Gambar 3-1 Main.cpp pelajaran	45
Gambar 3-2 <i>output</i> pelajaran	45
Gambar 4-1 Elemen <i>Single Linked list</i>	48
Gambar 4-2 <i>Single Linked list</i> dengan Elemen Kosong	48
Gambar 4-3 <i>Single Linked list</i> dengan 3 Elemen	48
Gambar 4-4 <i>Single Linked list Insert First</i> (1)	51
Gambar 4-5 <i>Single Linked list Insert First</i> (2)	51
Gambar 4-6 <i>Single Linked list Insert First</i> (3)	51
Gambar 4-7 <i>Single Linked list Insert Last</i> 1	52
Gambar 4-8 <i>Single Linked list Insert Last</i> 2	52
Gambar 4-9 <i>Single Linked list Insert Last</i> 3	52
Gambar 4-10 <i>Single Linked list Insert After</i> 1	52
Gambar 4-11 <i>Single Linked list Insert After</i> 2	53
Gambar 4-12 <i>Single Linked list Insert After</i> 3	53
Gambar 4-13 Ilustrasi elemen	54
Gambar 4-14 <i>Output singlelist</i>	55
Gambar 4-15 <i>Single Linked List Delete First</i> 1	55
Gambar 4-16 <i>Single Linked list Delete First</i> 2	55
Gambar 4-17 <i>Single Linked list Delete First</i> 3	56

Gambar 4-18 Single Linked list Delete Last 1.....	56
Gambar 4-19 Single Linked list Delete Last 2.....	56
Gambar 4-20 Single Linked list Delete Last 3.....	56
Gambar 4-21 Single Linked list Delete After 1	57
Gambar 4-22 Single Linked list Delete After 2	57
Gambar 4-23 Single Linked list Delete After 3	57
Gambar 5-1 Ilustrasi elemen	60
Gambar 5-2 Output singlelist.....	61
Gambar 5-3 Output pencarian 8.....	61
Gambar 5-4 Output total info elemen.....	61
Gambar 6-1 Double Linked list dengan Elemen Kosong	62
Gambar 6-2 Double Linked list dengan 3 Elemen.....	62
Gambar 6-3 Double Linked list Insert First 1.....	63
Gambar 6-4 Double Linked list Insert First 2.....	63
Gambar 6-5 Double Linked list Insert First 3.....	63
Gambar 6-6 Double Linked list Insert First 4.....	63
Gambar 6-7 Double Linked list Insert Last 1	64
Gambar 6-8 Double Linked list Insert Last 2	64
Gambar 6-9 Double Linked list Insert Last 3	64
Gambar 6-10 Double Linked list Insert Last 4	64
Gambar 6-11 Double Linked list Insert After 1.....	65
Gambar 6-12 Double Linked list Insert After 2.....	65
Gambar 6-13 Double Linked list Insert After 3.....	65
Gambar 6-14 Double Linked list Delete First 1.....	65
Gambar 6-15 Double Linked list Delete First 2.....	66
Gambar 6-16 Double Linked list Delete First 3.....	66
Gambar 6-17 Double Linked list Delete Last 1	66
Gambar 6-18 Double Linked list Delete Last 2	66
Gambar 6-19 Double Linked list Delete Last 3	67
Gambar 6-20 Double Linked list Delete After 1	67
Gambar 6-21 Double Linked list Delete After 2	67
Gambar 6-22 Double Linked list Delete After 3	67
Gambar 6-23 Output kasus kendaraan.....	69
Gambar 6-24 Output mencari nomor polisi	69
Gambar 6-25 Output menghapus data nomor polisi.....	69
Gambar 7-1 Stack dengan 3 Elemen	70
Gambar 7-2 Stack Push 1.....	71
Gambar 7-3 Stack Push 2.....	71
Gambar 7-4 Stack Push 3.....	71
Gambar 7-5 Stack Pop 1	72
Gambar 7-6 Stack Pop 2	72
Gambar 7-7 Stack Pop 3	72
Gambar 7-8 Stack Kosong dengan Representasi Table	73
Gambar 7-9 Push Elemen dengan Representasi Tabel.....	73

Gambar 7-10 Pop Elemen dengan Representasi Tabel	74
Gambar 7-11 Output stack	75
Gambar 7-12 Main stack	75
Gambar 7-14 Main stack dengan push ascending.....	75
Gambar 7-13 Output stack push ascending.....	75
Gambar 7-16 Main stack dengan input stream.....	76
Gambar 7-15 Output stack	76
Gambar 8-1 Queue	77
Gambar 8-2 Queue Insert 1	78
Gambar 8-3 Queue Insert 2	78
Gambar 8-4 Queue Insert 3	78
Gambar 8-5 Queue Delete 1.....	79
Gambar 8-6 Queue Delete 2.....	79
Gambar 8-7 Queue Delete 3.....	79
Gambar 8-8 Queue 1 Tidak Kosong Representasi Tabel.....	81
Gambar 8-9 Queue 1 Kosong Representasi Tabel	81
Gambar 8-10 Queue 2 Tidak Kosong 1 Representasi Table	82
Gambar 8-11 Queue 2 Tidak Kosong 2 Representasi Tabel.....	82
Gambar 8-12 Queue 2 Kosong Representasi Tabel	82
Gambar 8-13 Queue 2 Penuh “semu”	82
Gambar 8-14 Queue 3 Tidak Kosong 1 Representasi Table	83
Gambar 8-15 Queue 3 Tidak Kosong 2 Representasi Table	83
Gambar 8-16 Queue 3 Tidak Kosong 3 Representasi Table	83
Gambar 8-17 Output Queue	86
Gambar 8-18 Main Queue	86
Gambar 10-1 Tree.....	91
Gambar 10-2 Ordered Tree.....	92
Gambar 10-3 Heap Tree	93
Gambar 10-4 Binary Search Tree Insert.....	93
Gambar 10-5 Search pada Binary Search Tree traversal ke-1	94
Gambar 10-6 Search pada Binary Search Tree traversal ke-1	95
Gambar 10-7 Search pada Binary Search Tree traversal ke-1 : Nilai ketemu	95
Gambar 10-8 Traversal pada Binary Tree 1.....	95
Gambar 10-9 Traversal pada Binary Tree 2.....	96
Gambar 10-10 Main.cpp.....	98
Gambar 10-11 Output	98
Gambar 10-12 Main	98
Gambar 10-13 Output	98
Gambar 10-14 Ilustrasi Tree	99
Gambar 11-1 Binary Search Tree sebelum di-Delete.....	100
Gambar 11-2 Binary Search Tree setelah di-Delete.....	100
Gambar 11-3 most-left tree	101
Gambar 11-4 most-right tree	101
Gambar 13-1 Multi Linked list	103

Gambar 13-2 <i>Multi Linked list Insert Anak 1</i>	104
Gambar 13-3 <i>Multi Linked list Insert Anak 2</i>	104
Gambar 13-4 <i>Multi Linked list Delete Anak 1</i>	105
Gambar 13-5 <i>Multi Linked list Delete Anak 2</i>	106
Gambar 13-6 <i>Multi Linked list Delete Induk 1</i>	106
Gambar 13-7 <i>Multi Linked list Delete Induk 2</i>	107
Gambar 13-8 Ilustrasi data	111
Gambar 13-9 Fungsi <i>create</i>	111
Gambar 13-10 <i>Output</i>	112
Gambar 13-11 Main.cpp.....	112
Gambar 14-1 <i>Graph Kost</i> dan <i>Common Lab</i>	113
Gambar 14-2 <i>Graph Berarah (Directed Graph)</i>	113
Gambar 14-3 <i>Graph Representasi Multilist</i>	113
Gambar 14-4 <i>Graph</i>	114
Gambar 14-5 Contoh <i>Graph</i>	114
Gambar 14-6 Urutan Linier <i>Graph</i>	114
Gambar 14-7 Solusi 1	116
Gambar 14-8 <i>Graph Tidak Berarah (Undirected Graph)</i>	117
Gambar 14-9 Representasi <i>Graph Array</i> 2 Dimensi.....	118
Gambar 14-10 Representasi <i>Graph Multi Linked list</i>	118
Gambar 14-11 <i>Graph Jarak Antar kota</i>	118
Gambar 14-12 <i>Graph Breadth First Search (BFS)</i>	120
Gambar 14-13 <i>Graph Depth First Search (DFS)</i>	120
Gambar 14-14 Ilustrasi <i>Graph</i>	121



Modul 1 CODE BLOCKS IDE & PENGENALAN BAHASA C++ (BAGIAN PERTAMA)

TUJUAN PRAKTIKUM

1. Mengenal *environment* Code Blocks dengan baik.
2. Memahami cara menggunakan dan *troubleshooting* Code Blocks IDE.
3. Mengimplementasikan operator-operator dalam program.
4. Memahami cara membuat program sederhana dalam bahasa C++.
5. Memahami penggunaan tipe data dan variabel dalam bahasa C++.
6. Menggunakan operator-operator *input/output* dengan tepat.
7. Memahami dan mengimplementasikan fungsi kondisional dalam program.

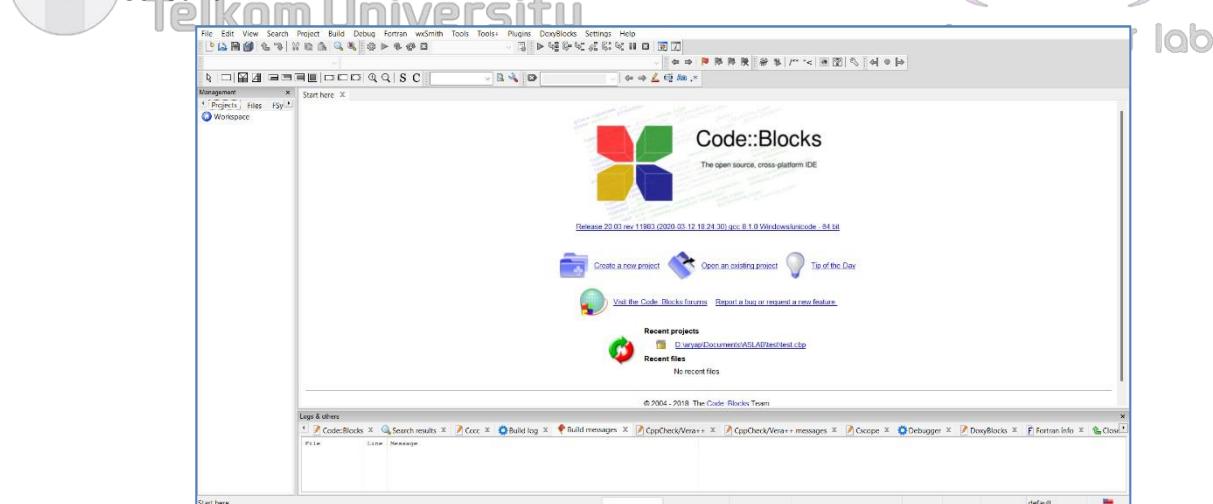
1.1 Pengenalan Code Blocks

Pada praktikum Struktur Data ini, kertas (*tool*) yang digunakan adalah Code Blocks. Kertas ini merupakan *free*, *open-source*, dan *cross-platform IDE*. Saat ini, Code Blocks berorientasi pada C/C++/Fortran (codeblocks, 2016).

1.1.1 Instalasi Code Blocks

Adapun cara menginstall Code Blocks adalah sebagai berikut.

1. Download terlebih dahulu *file exe* pada <http://www.codeblocks.org/downloads>. Pilih *Download the binary release* kemudian pilih *file* yang menggunakan mingw-setup (e.g. codeblocks-20.03mingw-setup.exe).
2. Setelah itu install *file* tersebut, akan muncul tampilan seperti pada **Error! Reference source not found..**

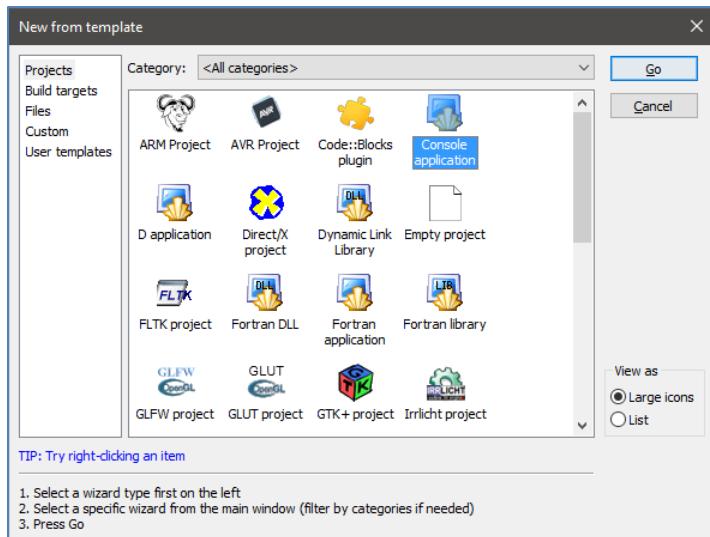


Gambar 1-1 Code Blocks IDE

1.1.2 Cara Menggunakan Code Blocks

Adapun cara menggunakan Code Blocks adalah sebagai berikut.

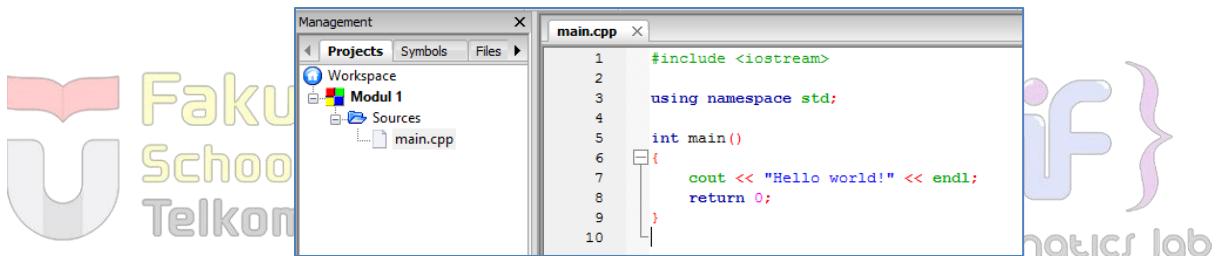
1. Membuat *Project* Baru dengan cara memilih *File > New > Projects*. Kemudian pada panel kiri pilih *Project*, pada panel kanan pilih *Console application* kemudian klik *Go* seperti pada **Error! Reference source not found..**.



Gambar 1-2 Membuat *project* baru

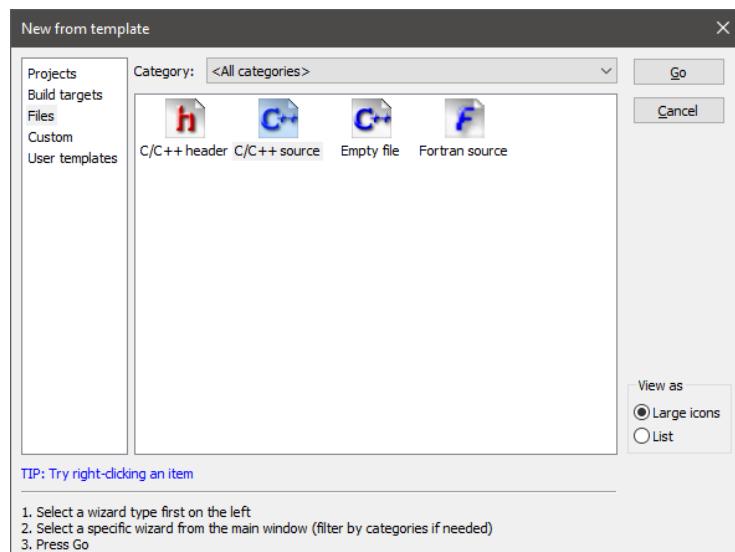
Setelah itu klik **Next >** Pilih Bahasa Pemrograman yang akan digunakan > Isi *Project title* dan *Folder to create project in* (tempat menyimpan *project*) > Klik **Finish**.

2. Menulis Sintak pada *editor* seperti pada **Error! Reference source not found..**



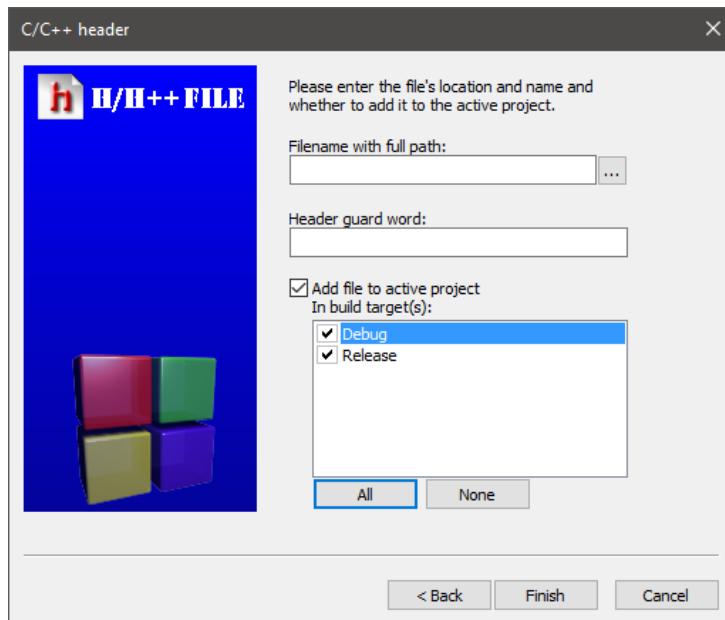
Gambar 1-3 Menulis sintak

3. Membuat *Class* Baru dengan cara klik **File > New > File**. Pada panel kiri pilih *Files*, dan pada panel kanan pilih *C/C++ source* Kemudian Klik **Go** seperti pada **Error! Reference source not found..**



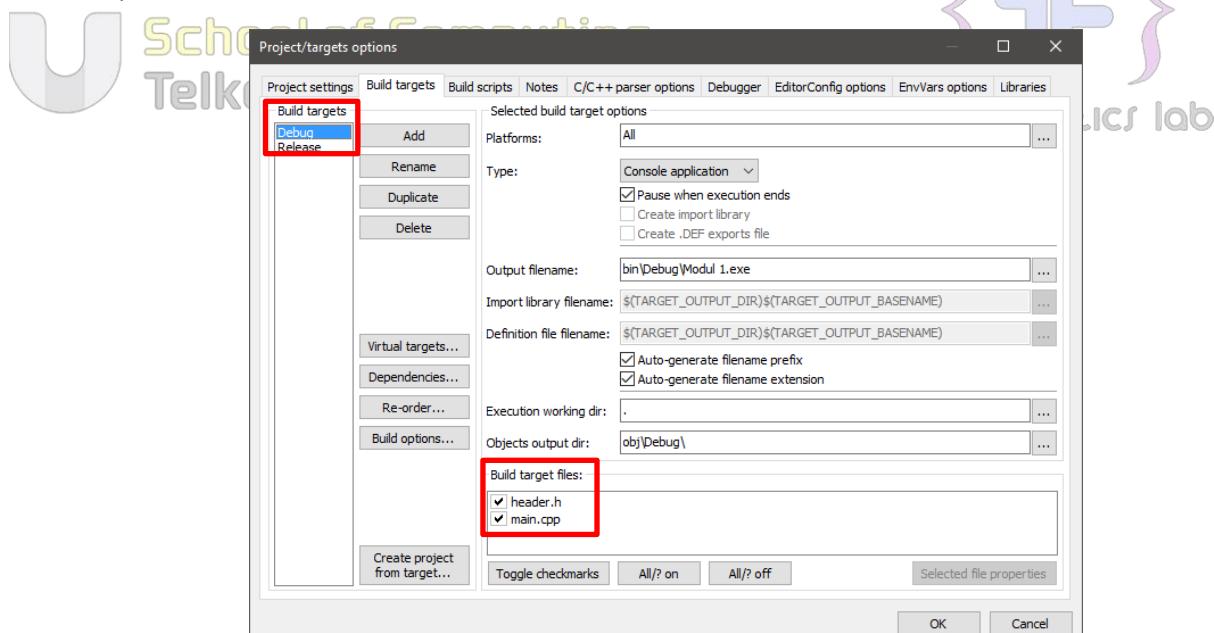
Gambar 1-4 Membuat *class* baru

Kemudian klik **Next** > Pilih bahasa pemrograman > Isi *Filename with full path* > Centang *all in build target* > **Finish**, seperti pada **Error! Reference source not found..**



Gambar 1-5 Centang all in build target

Jika anda lupa mencentang *build target*, dapat dilakukan *setting manual* dengan cara klik kanan pada *project* > *properties* > *Build targets* > *Debug* > Centang semua *target files* seperti pada **Error! Reference source not found..**



Gambar 1-6 Target options

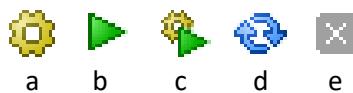
Setiap *class* harus memiliki sebuah nama yang dapat digunakan untuk membedakannya dari *class* lain. Penamaan *class* menggunakan kata benda tunggal yang merupakan abstraksi yang terbaik. Penulisan nama *class*, huruf pertama dari setiap kata pada nama *class* ditulis dengan menggunakan huruf kapital. Contohnya, Major dan StudentName. Jangan lupa untuk selalu menyimpan hasil pekerjaan anda. *Shortcut* untuk *save* satu *file* adalah *Ctrl+S* sedangkan

untuk menyimpan seluruh *file* adalah Ctrl+Shift+S. *Class* yang belum tersimpan akan ditandai dengan * pada bagian kiri nama *class*, seperti pada **Error! Reference source not found..**

```
main.cpp X *header.h X ← belum tersimpan
sudah tersimpan 1 #ifndef HEADER_H_INCLUDED
2 #define HEADER_H_INCLUDED
3
4
5
6 #endif // HEADER_H_INCLUDED
```

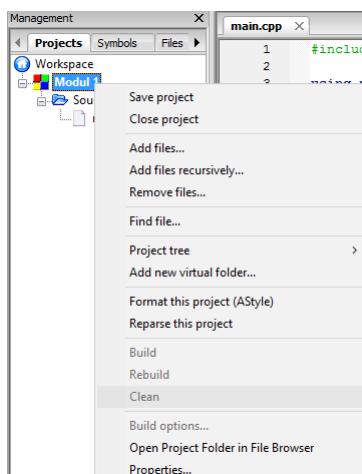
Gambar 1-7 Contoh *file*

4. *Compile* program, yaitu aksi untuk menjalankan program yang telah ditulis sehingga penulis dapat melihat *output* dari program.



- Build** merupakan aksi untuk membangun sintak menjadi sebuah program. Adapun *shortcut* yang digunakan adalah Ctrl-F9.
- Run** merupakan aksi untuk menjalankan program yang telah di-build. Program tidak akan berjalan sebelum dilakukan build. Adapun *shortcut* yang digunakan adalah Ctrl-F10.
- Build and Run** adalah aksi yang mengizinkan *Build* dan *Run* berjalan berurutan secara otomatis. Adapun *shortcut* yang digunakan adalah F9.
- Rebuild** merupakan aksi untuk membangun kembali program. Adapun *shortcut* yang digunakan adalah Ctrl-F11.
- Abort** merupakan aksi untuk mematikan program yang sedang berjalan.

5. **Clean**. Terkadang program yang kita buat tidak dapat di *run*, untuk itu perlu dilakukan *clean project* dengan cara klik kanan pada *project* kemudian klik *Clean* seperti pada **Error! Reference source not found..**. Setelah dilakukan *clean*, program akan dapat berjalan kembali.



Gambar 1-8 *Clean*

6. **Close dan Open Project**. Untuk menutup *project* dilakukan dengan cara klik kanan pada *project* kemudian pilih close. *Project* yang di *close* tidak akan terhapus dan tetap ada pada *directory*.

Sedangkan untuk membuka *project* yang telah di *close* dilakukan dengan cara memilih *File > Open > Pilih File *.cbp*

7. **Error Message** akan muncul ketika terjadi kesalahan pada penulisan sintak. Contoh seperti pada **Error! Reference source not found.**, terdapat *error* pada *line 8*. Pada error message diharapkan *semicolon* (;) sebelum *return* dan ternyata pada *line 7* dapat dilihat bahwa penulisan sintak belum diakhiri untuk fungsi *cout*.

```
main.cpp x
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hello world!" << endl
8     | return 0;
9 }
10

Logs & others
Code::Blocks x Search results x Ccc x Build log x Build messages x CppCheck x CppCheck message x
File Line Message
C:\Users\tst\D... == Build: Debug in Modul 1 (compiler: GNU GCC Compiler) ==
In function 'int main()':
C:\Users\tst\D... 8 error: expected ';' before 'return'
== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ==
```

Gambar 1-9 Error Message

1.2 Sekilas tentang C++

Bahasa C++ diciptakan oleh Bjarne Stroustrup di AT&T Bell Laboratories awal tahun 1980-an berdasarkan C ANSI (American National Standard Institute). Pertama kali, *prototype* C++ muncul sebagai C yang dipercanggih dengan fasilitas kelas. Bahasa tersebut disebut C dengan kelas (C with class). Selama tahun 1983-1984, C dengan kelas disempurnakan dengan menambahkan fasilitas pembebanlebihan operator dan fungsi yang kemudian melahirkan apa yang disebut C++. Simbol ++ merupakan operator C untuk operasi penaikan, muncul untuk menunjukkan bahwa bahasa baru ini merupakan versi yang lebih canggih dari C.

Borland International merilis *compiler* Borland C++ dan Turbo C++. Kedua *compiler* ini sama-sama dapat digunakan untuk mengkompilasi kode C++. Bedanya, Borland C++ selain dapat digunakan dibawah lingkungan DOS, juga dapat digunakan untuk pemrograman Windows. Selain Borland International, beberapa perusahaan lain juga merilis *compiler* C++, seperti Topspeed C++, Zortech C++ dan Code Blocks. Dalam praktikum ini, kita akan menggunakan bahasa C++.

```
1 #include <iostream>
2 #include <conio.h>
3
4 using namespace std;
5 int main(){
6     cout<<"Hello World!"<<endl;
7     getch();
8     return 0;
9 }
```

Program 1 Hello World

Setelah di-*compile* (F9) dan di-*run* (Ctrl+F9) maka hasil keluaran programnya adalah sebagai berikut:

```
Hello World!
```

Gambar 1-10 Hello World

1.3 Dasar Pemrograman

1.3.1 Struktur Program C++

Secara umum, pembagian struktur bahasa pemrograman C++ adalah sebagai berikut.

C++	Keterangan
#include <iostream> #include <conio.h> #include "newlibrary.h"	Pendeklarasian <i>library</i> yang akan digunakan di dalam program
#define PHI 3.14 const int constant1; const float constant2 = 0.5;	Pendefinisian konstanta
struct new_record_type { int element1; float element2; }	Pendefinisian tipe data bentukan / <i>record type</i> / struktur
int var1; float var2[2];	Pendeklarasian variabel
int function_A(){ // } void procedure_B(){ // }	Pendeklarasian fungsi dan prosedur
int main(){ // blok program return 0; }	Program utama

1.3.2 Pengenal (*Identifier*)

Identifier merupakan nama yang biasa digunakan untuk variabel, konstanta, fungsi atau objek lain yang didefinisikan oleh program.

Aturan yang digunakan untuk menentukan *identifier*:

1. Harus diawali dengan huruf (A....Z, a....z) atau garis bawah (_).
2. Karakter selanjutnya bisa berupa huruf, digit atau karakter garis bawah (_) atau dollar (\$).
3. Panjang maksimal *identifier* adalah 32 karakter, jika lebih maka yang dianggap adalah 32 karakter awal.
4. Tidak boleh mengandung spasi.
5. Tidak boleh menggunakan operator aritmatika (+ - * %).

Bahasa C bersifat *case sensitive*, jadi huruf besar dan huruf kecil dianggap berbeda.

Contoh:

```
panjang (berbeda dengan: Panjang)  
nilai4  
luas_total  
harga_beli$
```

1.3.3 Tipe data dasar

Data merupakan suatu nilai yang dapat dinyatakan dalam bentuk konstanta atau variabel. Data berdasarkan jenisnya dibagi dalam 5 kelompok, yang dinamakan sebagai tipe data dasar. Kelima kelompok tersebut:

1. Bilangan bulat (*integer*).
2. Bilangan *real* presisi – tunggal.
3. Bilangan *real* presisi – ganda.
4. Karakter.
5. Tak-bertipe.

Daftar tipe data dasar:

Tabel 1-1 Tipe Data Dasar

Type data	Contoh	Ukuran	Jangkauan nilai
Char	char nama[20];	1 Byte	-128 s.d. +127
Int	int nilai; int jumlah = 0;	2 Byte	-32768 s.d +32767
Long	long selisih;	4 Byte	-2.147.438.648 s.d +2.147.438.647
Float	float jumlah;	4 Byte	3.4e-38 s.d 3.4e+38
Double	double hasil;	8 Byte	1.7e-308 s.d 1.7e+308

1.3.4 Variabel

Variabel dalam program digunakan untuk menyimpan nilai, nilai variabel bisa berubah – ubah selama program berjalan. Aturan penamaan variabel sesuai dengan aturan penamaan *identifier*. Bentuk umum pendeklarasian suatu variabel dalam bahasa pemrograman C dapat ditulis sebagai berikut.

```
tipe_data nama_variabel;
```

Contoh:

```
int x,y;
char ch;
```

Kita juga dapat langsung memberikan nilai awal (inisialisasi) pada suatu variabel pada saat variabel tersebut di deklarasikan.

Contoh :

```
int x=20, y=6;
char nama[30] = "Budi";
```



1.3.5 Konstanta

Konstanta menyatakan nilai yang selalu tetap. Seperti halnya dengan variabel, konstanta juga mempunyai tipe.

Untuk mendeklarasikan suatu nilai yang sifatnya konstan kita cukup menambahkan kata **const** di depan tipe data dan variabel.

Contoh:

```
const float phi = 3.14;
const int n = 20;
```

1.4 Input / Output

1.4.1 Output

A. Fungsi cout()

Fungsi ini digunakan untuk mencetak data baik yang bertipe numerik, ataupun teks, baik konstanta ataupun variabel.

Algoritma	Program coba_output Algoritma Output("saya lagi belajar bahasa C++ nih!!!")
C++	#include <iostream> using namespace std; int main(){ cout<<"saya lagi belajar bahasa C++ nih!!!"<<endl; return 0; }

Output:

saya lagi belajar bahasa C++ nih!!!

B. Penentu Format

Penggunaan penentu format sangat berkaitan erat dengan suatu tipe data. Artinya suatu tipe data memiliki penentu format masing-masing. Format tersebut dipakai di bahasa C, sedangkan pada C++ tidak harus dipakai.

Tabel 1-2 Penentu Format

Tipe Data	Penentu Format
Integer	%d
Floating Point Bentuk Desimal	%f
Floating Point Bentuk Berpangkat	%e
Floating Point yang lebih pendek antara Desimal dan Berpangkat	%g
Double Precision	%lf
Character	%c
String	%s
Unsigned integer	%u
Long Integer	%ld
Long Unsigned Integer	%lu
Unsigned hexadecimal integer	%x
Unsigned octal integer	%o

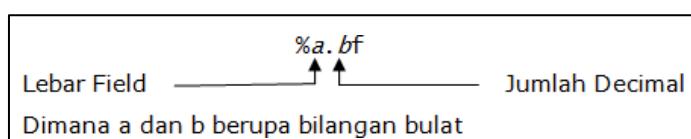
C. Penentu Lebar Field

Bila kita mencetak data bertipe *float*, seringkali tampilan yang diberikan tampak kurang manis. Misalnya desimal yang dicetak terlalu banyak.

```

1 #include <stdio.h>
2 int main(){
3     float bil;
4     bil = 2.5;
5     cout<< "bilangan = " <<bil;
6     return 0;
7 }
```

Sebenarnya kita dapat mengatur lebar *field* dan jumlah desimal yang ingin dicetak pada layar dengan cara memberikan tambahan format %f dengan bentuk sebagai berikut:

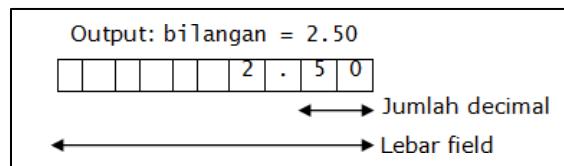


Gambar 1-11 Penentu Lebar Field

```

1 #include <stdio.h>
2 int main(){
3     float bil;
4     bil = 2.5;
5     cout<< "bilangan = %10.2f\n" << bil;
6     return 0;
7 }

```



Gambar 1-12 Output cout dengan Penentu Lebar Field

D. Escape Sequence

Disebut *escape sequence*, karena notasi '\' dianggap sebagai karakter "*escape*" (menghindar), dalam arti bahwa karakter yang terdapat setelah tanda '\' dianggap merupakan bukan teks biasa.

Berikut ini beberapa *escape sequence*.

Tabel 1-3 Escape Sequence

Escape sequence	Pengertian
\b	Backspace
\f	Formfeed
\n	Baris baru
\r	Carriage return
\t	Tabulasi
\'	Tanda kutip tunggal
\\"	Tanda kutip ganda
\\\	Backslash
\xaa	Kode ASCII dalam hexadecimal (aa: menunjukkan angka ASCII)

1.4.2 Input

A. Fungsi cin()

Cin() merupakan salah satu fungsi yang digunakan untuk meminta inputan *keyboard* dari user. Bentuk umumnya pendeklarasiannya adalah sebagai berikut :

cin>>nama_variabel;

Algoritma	C++
<u>Program</u> coba_input <u>Kamus</u> inp : int <u>Algoritma</u> <u>Input</u> (inp) <u>Output</u> ("nilai = ", inp)	<pre>#include <iostream> using namespace std; int main(){ int inp; cin >> inp; cout << "nilai = " << inp; return 0; }</pre>

B. Penentu Format

Penentu format untuk fungsi `cin()` sama seperti penentu format untuk fungsi `printf()` (digunakan pada bahasa C).

C. Fungsi `getche()` dan `getch()`

Fungsi `input getche()` memiliki sifat yang sedikit berbeda dari `cin()`. Jika dalam `cin()` jumlah karakter yang diinputkan boleh bebas, dalam `getche()` hanya sebuah karakter yang dapat diterima. Bila `cin()` membutuhkan *return (enter)* untuk mengakhiri *input*, `getche` tidak membutuhkan *return (enter)*. *Input* akan dianggap selesai begitu anda selesai mengetikkan satu karakter. Dan secara otomatis akan melanjutkan perintah-perintah berikutnya. Perlu diketahui juga bahwa inputan yang diterima `getche()` akan disimpan dalam variabel dengan tipe data karakter yang sebelumnya harus di deklarasikan terlebih dahulu.

```
1 #include <iostream>
2 #include <conio.h>
3 using namespace std;
4 int main(){
5     char ch;
6     ch = getche();
7     cout << " anda telah menekan tombol "<< ch << endl;
8     return 0;
9 }
```

Fungsi `input` lain yang mirip dengan `getche()` adalah `getch()`. Satu-satunya perbedaan antara `getche()` dan `getch()` adalah bahwa `getche()` akan menampilkan karakter yang diinput ke *layer* sedangkan `getch()` hanya akan menyimpan karakter masukan ke memori tanpa menampilkannya ke layar. Untuk pendeklarasiannya sama seperti fungsi `getche()`.

1.5 Operator

Operator adalah suatu simbol yang digunakan untuk melakukan suatu operasi atau manipulasi. Bahasa C merupakan bahasa yang kaya dengan operator yaitu: Operator Aritmatika, Operator Penggeraan (*assignment operator*), Operator Logika, Operator Unary, Operator Bitwise, Operator Kondisional, dan lain-lain.

Tabel 1-4 Operator, Arah Proses, dan Jenjangnya

Kategori (Arti)	Operator	Kategori (Arti)	Operator
Panggilan fungsi, subscript <i>array</i> , dan elemen struktur data	() [] ->	Operator Hubungan (sama dengan, tidak sama dengan)	== !=
Operator Unary (NOT, komplemen, negasi, inkremen, dekremen , <i>address</i> , <i>indirection</i>)	! ~ - ++ -- & *	Operator Bitwise AND	&
Operator Aritmatika(Perkalian, pembagian, Sisa Pembagian/mod)	*	Operator Bitwise XOR	^
Operator Aritmatika (Pertambahan, Pengurangan)	/ %	Operator Bitwise OR	
	+	Operator Logika AND	&&
	-	Operator Logika OR	
		Operator Kondisional	?:
		Operator Penggeraan Aritmatika (<i>assignment, assignment</i> perkalian, <i>assignment</i> pembagian, <i>assignment</i> mod, <i>assignment</i> penjumlahan, <i>assignment</i> pengurangan)	= *= /= %= += -=

Operator Bitwise Pergeseran Bit (shift kiri, shift kanan)	<< >>	Operator Penggerjaan Bitwise (assignment AND bitwise, assignment OR bitwise, assignment XOR bitwise, assignment shift kanan)	&=
Operasi Hubungan (kurang dari, kurang dari atau sama dengan lebih dari, lebih dari atau sama dengan)	< <= > >=	=	
Operator Koma		<=>	>=

A. Operator Aritmatika

Misalkan terdapat ungkapan sebagai berikut : $A + B / C + D$. Untuk mengubah jenjang dapat digunakan tanda kurung '()' (sebagai operator jenjang tertinggi) sebagai berikut: $(A + B)/(C + D)$.

```

1 #include <iostream>
2 #include <stdio.h>
3 using namespace std;
4 int main(){
5     int W, X, Y; float Z;
6     X = 7; Y = 3; W = 1;
7     Z = (X + Y)/(Y + W);
8     cout<< "Nilai z = " << Z << endl;
9     return 0;
10 }
```

B. Operator Penggerjaan (Assignment)

Operator ini digunakan untuk memindahkan nilai dari suatu ungkapan ke suatu pengenal. Di samping operator penggerjaan $=$ yang umumnya digunakan di bahasa-bahasa pemrograman, bahasa C menyediakan beberapa operator penggerjaan lain.

Misalnya: $A += 7$, ekuivalen dengan, $A = A + 7$

C. Operator Logika

Operasi logika membandingkan dua buah nilai logika. Nilai logika adalah nilai benar atau salah.

Misalnya:

1. $(\text{kar} > 'A') \&\& (\text{kar} < 'Z')$
Hasil operasi $\&\&$ bernilai benar hanya jika $\text{kar} > 'A'$ dan $\text{kar} < 'Z'$
2. $(\text{pilihan} == 'Y') || (\text{pilihan} == 'y')$
Hasil operasi logika $||$ bernilai benar jika pilihan berupa 'Y' atau 'y'.
3. $! \text{operand}$
Hasil operand $!$ akan bernilai benar jika operand bernilai salah dan sebaliknya.

D. Operator Unary

1) Operator Unary tipe

Jika pernyataan $Z = (X + Y)/(Y + W)$ yang memiliki *output* 2.000000 diubah menjadi $Z = (\text{float})(X + Y)/(Y + W)$ maka outputnya menjadi : Nilai $Z = 2.500000$. Mengapa demikian?

Itulah peranan dari operator tipe ini. Operator ini akan menghasilkan tipe data yang diinginkan walau berasal dari operand-operand dengan tipe berbeda.

2) Operator Unary sizeof

Operator Unary *sizeof* digunakan untuk mengetahui ukuran *memory* dari operandnya dalam satuan *byte*. Misalnya untuk komputer IBM PC, ukuran dari nilai *float* adalah 4 *byte*, kemungkinan di komputer yang lain, ukuran *float* bukan 4 *byte*. Contoh penggalan program penggunaan *sizeof*:

```
printf("Ukuran karakter = %d byte \n", sizeof(char));
```

E. Operator sizeof

Operator sizeof menghasilkan ukuran dari variabel atau suatu tipe pada saat kompilasi program. Penggunaan sizeof dengan operan (berupa variabel atau tipe) yang ditempatkan dalam tanda kurung.

Contoh:

```
sizeof(char) → 1  
sizeof(int) → 2
```

Operator ini sangat bermanfaat untuk menghitung besarnya sebarang tipe atau variabel, terutama untuk variabel dan tipe yang kompleks (seperti struktur).

```
1 #include <iostream>  
2 #include <conio.h>  
3 using namespace std;  
4 int main(){  
5     char x;  
6     float y;  
7     cout << "ukuran variabel char " << sizeof(x) << endl;  
8     cout << "ukuran variabel float " << sizeof(y) << endl;  
9     cout << "ukuran variabel int " << sizeof(int) << endl;  
10    getch();  
11    return 0;  
12 }
```

F. Operator Increment dan Decrement

Bahasa C++ menyediakan 2 operator yang tidak biasa untuk *increment* dan *decrement*. Operator *increment* ++ akan menambahkan nilai 1 dari variabel, sedangkan operator *decrement* -- akan mengurangi variabel dengan nilai 1.

Contoh:

```
++i; //ekuivalen dengan i=i+1  
--j; //ekuivalen dengan j=j-1
```

Kedua operator ini bisa diletakkan sebelum variabel (prefix, contoh: ++i), dan sesudah variabel (postfix, contoh: i++) . Effect dari keduanya sama yaitu akan menambahkan nilai 1 ke variabel i. Akan tetapi, untuk ekspresi ++i dan i++ ada bedanya. Jika ekspresi ++i maka variabel akan di-*increment* dulu sebelum digunakan sedangkan untuk operator i++ maka nilai i akan digunakan terlebih dahulu setelah itu baru di-*increment*.

Untuk lebih jelasnya perhatikan penggalan program berikut.

Algoritma	C++
<pre>Program coba_increment_belakang Kamus r, s : integer Algoritma r ← 10 r ← r + 1 s ← 10 + r output (r,s)</pre>	<pre>#include <iostream> #include <conio.h> #include <stdlib.h> using namespace std; int main(){ int r = 10; int s; system("cls"); s=10 + ++r; cout<< "Nilai r= "<<r<<endl; cout<< "Nilai s= "<<s<<endl; getch(); return 0; }</pre>

```
Nilai r=11  
Nilai s=21_
```

Gambar 1-13 Increment di Belakang

Penjelasan: pada contoh di atas nilai r pertama di-*increment* terlebih dahulu sehingga nilai r=11 dan kemudian dijumlahkan dengan 10 sehingga nilai s sama dengan 21. *Increment* ini biasanya disebut **pre-increment**.

Algoritma	C++
<pre>Program coba_increment_depan Kamus r,s : integer Algoritma r ← 10 s ← 10 + r r ← r + 1 output (r,s)</pre>	<pre>#include <iostream> #include <conio.h> #include <stdlib.h> using namespace std; int main(){ int r = 10; int s; system("cls"); s=10 + r++; cout<< "Nilai r= "<<r<<endl; cout<< "Nilai s= "<<s<<endl; getch(); return 0; }</pre>

```
Nilai r=11
Nilai s=20_
```

Gambar 1-14 Increment di Depan

Penjelasan: pada contoh di atas nilai s diisi terlebih dahulu dengan penjumlahan antara 10 dengan r sehingga nilai s=20. Kemudian setelah itu baru nilai r di-*increment* sehingga nilai r=11. *Increment* dengan cara seperti ini biasanya disebut **post-increment**.

1.6 Pemodifikasi Tipe

Pemodifikasi tipe (*type modifier*) dapat dikenakan diawal tipe data kecuali untuk void.

1.6.1 Unsigned

Tipe data ini digunakan bila kita hanya ingin bekerja dengan data yang bernilai positif saja. Misalnya *unsigned integer* akan menerima data dari 0 – 65.535. (tidak lagi dari -32.768 hingga 32.768).

Contoh cara deklarasinya:

```
unsigned int jumlah;
unsigned char ch;
```

1.6.2 Short

Tipe data ini kadangkala disamakan dengan *integer* dan kadangkala juga dibedakan, tergantung pada sistem dan jenis komputer yang digunakan.

1.6.3 Long

Tipe data ini digunakan untuk menaikkan kapasitas dari suatu variabel. Misalnya *long integer* memiliki bilangan bulat dari -2.147.483.648 hingga 2.147.483.647

Contoh Pendeklarasiannya :

```
unsigned long int harga_rumah;
```

1.7 Kondisional

Untuk menyelesaikan suatu masalah diperlukan pengambilan keputusan, Bahasa C menyediakan beberapa jenis pernyataan berupa operator kondisi sebagai berikut.

1. Pernyataan **if**
2. Pernyataan **if-else**
3. Pernyataan **switch**

Pernyataan pengambilan keputusan di atas memerlukan suatu kondisi sebagai basis pada pengambilan keputusan. Kondisi umum yang dipakai keadaan benar atau salah.

1.7.1 Bentuk 1

```
if (kondisi)  
    pernyataan ;
```

Arti dari perintah if di atas adalah jika kondisi benar maka pernyataan akan dijalankan. Sedangkan kondisi ditulis di antara tanda kurung, dapat berupa ungkapan yang memiliki nilai benar atau salah. Dan pernyataan berupa sebuah pernyataan tunggal pernyataan majemuk atau pernyataan kosong.

Contoh Bentuk 1	
Algoritma	C++
<pre>Program coba_if kamus tot_pembelian, diskon : real algoritma input(tot_pembelian) diskon ← 0 if (tot_pembelian >= 100000) then diskon ← 0.05 * tot_pembelian output(diskon)</pre>	<pre>/* contoh penggunaan 'if' */ #include <iostream> using namespace std; int main(){ double tot_pembelian, diskon; cout<<"total pembelian: Rp"; cin>>tot_pembelian; diskon = 0; if(tot_pembelian >= 100000) diskon = 0.05*tot_pembelian; cout<<"besar diskon = Rp" <<diskon; }</pre>

1.7.2 Bentuk 2

```
if (kondisi)  
    pernyataan1 ;  
else  
    pernyataan2 ;
```

Arti dari pernyataan *if-else* di atas adalah:

1. Jika kondisi benar, maka pernyataan1 dijalankan.
2. Jika kondisi salah, maka pernyataan2 yang akan dijalankan.

pernyataan1 dan pernyataan2 dapat berupa sebuah pernyataan tunggal, pernyataan majemuk ataupun pernyataan kosong. Bentuk ini dapat disederhanakan dengan kondisional ekspresi dengan bentuk umumnya sebagai berikut.

```
expr1 ? expr2 : expr3
```

Jika *expr1* benar maka *expr2* yang dijalankan, sedangkan jika salah maka *expr3* yang dijalankan. Untuk lebih jelasnya perhatikan contoh berikut.

Contoh Bentuk 2

Algoritma	C++
<pre>Program coba_if kamus tot_pembelian, diskon : real algoritma input(tot_pembelian) diskon ← 0 if (tot_pembelian >= 100000) then diskon ← 0.05 * tot_pembelian else diskon ← 0 output(diskon)</pre>	<pre>/* contoh penggunaan 'if else' */ #include <iostream> using namespace std; int main(){ double tot_pembelian, diskon; cout<<"total pembelian: Rp"; cin>>tot_pembelian; diskon = 0; if(tot_pembelian >= 100000) diskon = 0.05*tot_pembelian; else diskon = 0; cout<<"besar diskon = Rp" <<diskon; }</pre>
Penyederhanaan	
<pre>* contoh penggunaan 'simple if else' */ #include <iostream> using namespace std; int main(){ double tot_pembelian, diskon; cout<<"total pembelian: Rp"; cin>>tot_pembelian; diskon = (tot_pembelian >= 100000) ? 0.05*tot_pembelian : 0; cout<<"besar diskon = Rp" <<diskon; }</pre>	

1.7.3 Bentuk 3

Bentuk ketiga menggunakan pernyataan *switch*, merupakan pernyataan yang dirancang khusus untuk menangani pengambilan keputusan yang melibatkan banyak alternatif.

Bentuk umumnya:

```
Switch (Variabel) {
Case kondisi1: pernyataan1;
break;
Case kondisi2 : pernyataan2;
break;
default: pernyataan_n;
break;
}
```

Pengujian pada *switch* akan dimulai dari kondisi1, kalau nilai kondisi1 cocok maka pernyataan1 dilakukan, bila tidak cocok akan diteruskan pada pengecekan pernyataan2. Bila tidak ditemukan kondisi yang cocok maka statement pada default akan dilakukan. Contoh penggunaan *switch*:

Algoritma	C++
<pre>Program coba_switch Kamus Kode_hari : integer Algoritma Input(kode_hari) Depend on (kode_hari) kode_hari = 1: output("Hari Senin") kode_hari = 2: output ("Hari Selasa") kode_hari = 3: output ("Hari Rabu") kode_hari = 4: output ("Hari Kamis") kode_hari = 5:</pre>	<pre>#include <stdio.h> int main(){ int kode_hari; puts("Menentukan hari\n"); puts("1=Senin 3=Rabu 5=Jumat 7=Minggu "); puts("2=Selasa 4=Kamis 6=Sabtu "); cin>>kode_hari; switch(kode_hari){ case 1: puts("Hari Senin"); break; case 2: puts("Hari Selasa"); break; case 3: puts("Hari Rabu"); break; case 4: puts("Hari Kamis"); break; case 5: puts("Hari Minggu"); break; } }</pre>

```

        output ("Hari Jumat")
kode_hari = 6:
    output ("Hari Sabtu")
kode_hari = 7:
    output ("Hari Minggu")
else:
    output("kode
salah!!!");
}

puts("Hari Rabu");
break;
case 4:
    puts("Hari Kamis");
    break;
case 5:
    puts("Hari Jumat");
    break;
case 6:
    puts("Hari Sabtu");
    break;
case 7:
    puts("Hari Minggu");
    break;
default:
    puts("Kode masukan salah!!!");
}
return 0;
}

```

1.8 Perulangan

Apabila kita ingin menuliskan angka 1 s.d. 5 secara berurutan maka kita bisa saja menuliskan semua angka tersebut secara manual karena *range* yang ditulis masih kecil. Lain halnya apabila angka yang ingin kita tulis *range*-nya dari 1 s.d. 10000 apakah kita akan menulisnya secara manual 1, 2, 3, 4, ..., 10000? Tentu tidak. Diperlukan suatu cara yaitu perulangan (looping). Perulangan digunakan untuk mengefisienkan waktu dan meringkas kode program dalam pengeksekusian sub-program yang sama. Hal yang terpenting dalam perulangan adalah harus ada kondisi berhenti.



1.8.1 Perulangan dengan *for* dan *while*

Perulangan *for* dan *while* biasa digunakan saat kondisi ekspresi terpenuhi. Jika tidak maka perulangan akan terhenti.

Di bawah ini merupakan bentuk umum perulangan *for*:

```
for (initialization; condition; increment/decrement)
    statement;
```

Bentuk *for* di atas ekuivalen dengan bentuk *while* di bawah ini:

```
initialization;
while (condition) {
    statement;
    increment/decrement;
}
```

Keterangan :

1. *initialization*: pernyataan untuk menyatakan keadaan awal dari variabel control.
2. *condition*: ekspresi relasi yang menyatakan kondisi untuk keluar dari perulangan.
3. *increment/decrement*: pengatur perubahan nilai variabel kontrol.

Berikut adalah contoh program perulangan dengan *for*:

Algoritma	C++
<pre>Program coba_for Kamus Jum, i : integer Algoritma Input(jum) i traversal [1..jum] output("saya pintar")</pre>	<pre>#include <iostream> #include <conio.h> using namespace std; int main(){ int jum; cout<<"jumlah perulangan: "; cin>>jum; for(int i=0; i<jum; i++){ cout<<"saya pintar\n"; } getch(); return 0; }</pre>

Berikut adalah contoh program perulangan dengan *while*:

Algoritma	C++
<pre>Program coba_while Kamus Jum, i : integer Algoritma i ← 1 Input(jum) while (i<=jum) do output("saya pintar") i ← i + 1</pre>	<pre>#include <iostream> #include <conio.h> using namespace std; int main(){ int i=1; int jum; cout<<"masukan banyak baris: "; cin>>jum; while(i<=jum){ cout<<"baris ke-"<<i<<endl; i++; //sama dengan i=i+1 } getch(); return 0; }</pre>

1.8.2 Perulangan dengan *do ... while*

Pada dasarnya struktur perulangan *do...while* sama saja dengan struktur *while*, hanya saja pada proses perulangan dengan *while*, seleksi berada di *while* yang letaknya di atas sementara pada perulangan *do...while*, seleksi *while* berada di bawah batas perulangan. Jadi dengan menggunakan struktur *do...while* sekurang-kurangnya akan terjadi satu kali perulangan.

Bentuk umum perulangan *do...while* adalah sebagai berikut:

```
do {
    statement;
} while (condition);
```

Algoritma	C++
<pre>Program coba_do_while Kamus Jum, i : integer Algoritma i ← 0 Input(jum) repeat output("saya pintar") i ← i + 1 until (i >= jum)</pre>	<pre>#include <iostream> #include <conio.h> using namespace std; int main(){ int i=1; int jum; cin>>jum; do{ cout<<"baris ke-"<<i+1<<endl; i++; }while(i<jum); getch(); return 0; }</pre>

1.9 Struktur

Struktur merupakan tipe data bentukan berupa kumpulan dari variabel yang dinyatakan dalam sebuah nama, setiap variabel bisa memiliki tipe yang berlainan. Struktur bisa digunakan untuk mengelompokkan beberapa informasi yang saling berkaitan menjadi satu kesatuan (dalam bahasa pascal, struktur disebut dengan **record**).

Bentuk umum pendeklarasian struktur:

```
struct nama_tipe_struktur {
    tipe field1;
    tipe field2;
    .
    tipe fieldN;
} variabel_struktur1 ... variabel_strukturN;
```

Mengakses elemen struktur menggunakan tanda dot atau (.):

```
variabel_struktur.nama_field
```

Contoh pendeklarasian:

Algoritma	C++
<pre>Tipe Type tanggal < Tanggal : integer Bulan : integer Tahun : integer > Type data_rekam < Nama : char[31] tgl_lahir : tanggal > kamus Info : data rekam</pre>	<pre>struct tanggal{ int tanggal; int bulan; int tahun; }; struct data_rekam{ char nama[31]; tanggal tgl_lahir; }; data_rekam info</pre>

Contoh pengaksesan:

```
cout<<info.nama;
cout<< info.tgl_lahir.tanggal << info.tgl_lahir.bulan << info.tgl_lahir.tahun
<< endl;
```

Penggunaan struktur sering dikombinasikan dengan *array*. Contoh penggunaan struktur dengan *array* misalnya untuk menyimpan data siswa.

Algoritma	C++
<pre>program coba_array_struct Type Type data < nama : char[40] nilai : integer > konstant maks ← 5 kamus siswa : data[maks] algoritma i traversal [1..maks] output("siswa ",i) input(siswa[i].nama) input(siswa[i].nilai) i traversal [1..maks] output("siswa ",i) output(siswa[i].nama) output(siswa[i].nilai)</pre>	<pre>/* contoh array dan struktur */ #include <iostream> #include <conio.h> #define MAX 5 using namespace std; int main(){ int i; struct data{ char nama[40]; int nilai; }; data siswa[MAX]; for(i=0; i<MAX; i++){ cout<<"masukkan data ke-" <<i+1<<endl; cout<<"nama = "; cin>>siswa[i].nama; cout<<"nilai = "; cin>>siswa[i].nilai; } cout<<"\nData siswa\n";</pre>

```

cout<<"======" ;
for(i=0; i<MAX; i++) {
    cout<<"\n\n data ke-"<<i+1;
    cout<<"\n\n nama
=><<siswa[i].nama;
    cout<<"\n\n nilai
=><<siswa[i].nilai;
}
getch();
return 0;
}

```

Tipe data struktur ini cukup rumit, tetapi sangat penting dalam pemrograman bahasa C, terutama dalam struktur data. Untuk merepresentasikan data sebagian besar akan menggunakan struktur yang dikombinasikan dengan *array* maupun *pointer* (akan dibahas pada bab selanjutnya).

1.10 Blok Program

Setiap bahasa komputer disusun dengan struktur yang berbeda. Untuk dapat mengerti bagaimana membuat program maka kita harus dapat memahami struktur dari program tersebut terlebih dahulu. Struktur program dari bahasa C terdiri dari fungsi-fungsi, seperti berikut.

```
fungsi_lain() { // fungsi lain yang ditulis programmer
    /* bagian ini berisi statement */
}
main () { // fungsi utama
    /* bagian ini berisi statement */
```

Perhatikan contoh di bawah ini:

Algoritma	C++
<pre>Program coba1 kamus celcius, farenheit : real</pre>	<pre>#include <iostream> using namespace std; /*deklarasi fungsi */ float ctof(float celcius); int main() { float celcius, fahrenheit; cout<<"nilai celcius? "; cin>>celcius; /*hitung konversi*/ fahrenheit = ctof(celcius); cout<<celcius<<" celcius adalah " <<fahrenheit<<"Fahrenheit ="<<endl; return 0; }</pre>
<pre>function ctof(in:celcius:real):real algoritma input(celcius) farenheit ← ctot(celcius) output(celcius)</pre>	<pre>/*pendefinisian fungsi*/ float ctot(float celcius){ return celcius * 1.8 + 32; }</pre>
<pre>function ctot(in: celcius: real): real algoritma → celcius * 1.8 + 32</pre>	

Output :

Nilai celcius? 10
10.000000 celcius adalah 50.000000 fahrenheit

Pembahasan :

1. Komentar bebas dapat diletakkan di manapun dalam blok program, diapit tanda /* dan */.

- Contoh dari program di atas: /* program pertama kita */
2. *File judul (header file)* adalah *file* berisi *prototype* sekumpulan fungsi pustaka.
 3. Fungsi pustakanya sendiri terdapat pada *file* pustaka (*library file*).
Contoh: Penggunaan fungsi pustaka `printf()` dan `scanf()` harus menyertakan *file* judul `stdio.h` yang berisi *prototype* fungsi pustaka operasi *input* dan *output* serta menggunakan preposesor `#include`
 4. Penulisan statemen dalam bahasa C diakhiri dengan titik koma (;
 5. Semua variabel yang digunakan harus telah deklarasikan dahulu.
Contoh: float celcius, fahrenheit;

1.11 Latihan

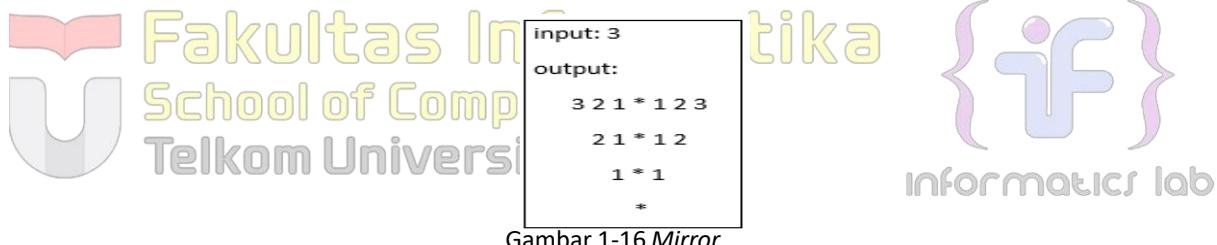
1. Buatlah program yang menerima *input*-an dua buah bilangan bertipe float, kemudian memberikan *output*-an hasil penjumlahan, pengurangan, perkalian, dan pembagian dari dua bilangan tersebut.
2. Buatlah sebuah program yang menerima masukan angka dan mengeluarkan *output* nilai angka tersebut dalam bentuk tulisan. Angka yang akan di-*input*-kan user adalah bilangan bulat positif mulai dari 0 s.d 100

contoh:

79 : tujuh puluh Sembilan

Gambar 1-15 Contoh

3. Buatlah program yang dapat memberikan *input* dan *output* sbb.



input: 3
output:
3 2 1 * 1 2 3
2 1 * 1 2
1 * 1
*

Gambar 1-16 Mirror

Modul 2 PENGENALAN BAHASA C++ (BAGIAN KEDUA)

TUJUAN PRAKTIKUM

1. Memahami penggunaan *pointer* dan alamat memori
2. Mengimplementasikan fungsi dan prosedur dalam program

2.1 Array

Array merupakan kumpulan data dengan nama yang sama dan setiap elemen bertipe data sama. Untuk mengakses setiap komponen / elemen *array* berdasarkan indeks dari setiap elemen.

2.1.1 Array Satu Dimensi

Adalah *array* yang hanya terdiri dari satu larik data saja. Cara pendeklarasian *array* satu dimensi:

```
tipe_data nama_var[ukuran]
```

Keterangan:

Tipe_data → menyatakan jenis elemen *array* (int, char, float, dll).

Ukuran → menyatakan jumlah maksimum *array*.

Contoh:

```
int nilai[10];
```

Menyatakan bahwa *array* nilai mengandung 10 elemen dan bertipe *integer*.

Dalam C++ data *array* disimpan dalam memori pada lokasi yang berurutan. Elemen pertama memiliki indeks 0 dan selemen selanjutnya memiliki indeks 1 dan seterusnya. Jadi jika terdapat *array* dengan 5 elemen maka elemen pertama memiliki indeks 0 dan elemen terakhir memiliki indeks 4.

nama_var[indeks]

nilai[5] → elemen ke-5 dari *array* nilai. Contoh memasukkan data ke dalam *array*:

```
nilai[4] = 90;      /*memasukkan 90 ke dalam array nilai indeks ke-4*/
cin << nilai[4];   /*membaca input-an dari keyboard*/
```

2.1.2 Array Dua Dimensi

Bentuk *array* dua dimensi ini mirip seperti tabel. Jadi *array* dua dimensi bisa digunakan untuk menyimpan data dalam bentuk tabel. Terbagi menjadi dua bagian, dimensi pertama dan dimensi kedua. Cara akses, deklarasi, inisialisasi, dan menampilkan data sama dengan *array* satu dimensi, hanya saja indeks yang digunakan ada dua.

Contoh:

```
int data_nilai[4][3];
nilai[2][0] = 10;
```

	0	1	2
0			
1			
2	10		
3			

Gambar 2-1 Ilustrasi Array Dua Dimensi

2.1.3 Array Berdimensi Banyak

Merupakan *array* yang mempunyai indeks banyak, lebih dari dua. Indeks inilah yang menyatakan dimensi *array*. *Array* berdimensi banyak lebih susah dibayangkan, sejalan dengan jumlah dimensi dalam *array*.

Cara deklarasi:

```
tipe_data nama_var[ukuran1][ukuran2]...[ukuran-N];
```

Contoh:

```
int data_rumit[4][6][6];
```

Array sebenarnya masih banyak pengembangannya untuk penyimpanan berbagai bentuk data, pengembangan *array* misalnya untuk *array* tak berukuran.

2.2 Pointer

2.2.1 Data dan Memori

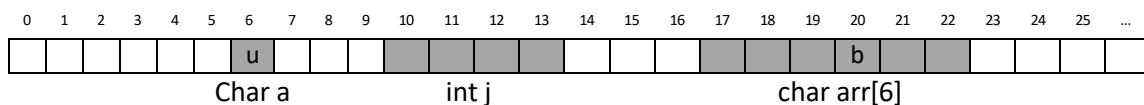
Semua data yang ada digunakan oleh program komputer disimpan di dalam memori (RAM) komputer. Memori dapat digambarkan sebagai sebuah *array* 1 dimensi yang berukuran sangat besar. Seperti layaknya *array*, setiap *cell memory* memiliki “indeks” atau “alamat” unik yang berguna untuk identitas yang biasa kita sebut sebagai “*address*”

Saat program berjalan, Sistem Operasi (OS) akan mengalokasikan *space memory* untuk setiap variabel, objek, atau *array* yang kita buat. Lokasi pengalokasian memori bisa sangat teracak sesuai proses yang ada di dalam OS masing-masing. Perhatikan ilustrasi berikut



Digambarkan sebuah *memory* diasumsikan setiap *cell* menyimpan 1 *byte* data. Pada saat komputer pertama kali berjalan keadaan memori adalah kosong. Saat variabel dideklarasikan, OS akan mencari *cell* kosong untuk dialokasikan sebagai memori variabel tersebut.

```
char a;  
int j;  
char arr[6];  
arr[3] = 'b'  
a = 'u'
```



Gambar 2-3 Ilustrasi Alokasi Memory

Pada contoh di atas variabel a dialokasikan di *memory* alamat x6, variabel j dialokasi kan di alamat x10-13, dan variabel arr dialokasiakan di alamat x17-22. Nilai variabel yang ada di dalam memori dapat dipanggil menggunakan alamat dari *cell* yang menyimpannya. Untuk mengetahui alamat memori tempat di mana suatu variabel dialokasikan, kita bisa menggunakan *keyword* “&” yang ditempatkan di depan nama variabel yang ingin kita cari alamatnya.

C++	Output	Keterangan
<pre>Cout << a << endl; Cout << &a << endl; Cout << j << endl; Cout << &j << endl; Cout << &(arr[4]) << endl;</pre>	<pre>'u' x6 0 X10 X21</pre>	Nilai variabel a Alamat variabel a Nilai variabel j Alamat variabel j Alamat variabel arr[4]

2.2.2 Pointer dan Alamat

Variabel *pointer* merupakan dasar tipe variabel yang berisi *integer* dalam format heksadesimal. *Pointer* digunakan untuk menyimpan alamat memori variabel lain sehingga *pointer* dapat mengakses nilai dari variabel yang alamatnya ditunjuk.

Cara pendeklarasian variabel *pointer* adalah sebagai berikut:

```
type *nama_variabel;
```

Contoh:

```
int *p_int;
/* p_int merupakan variabel pointer yang menunjuk ke data bertipe int */
```

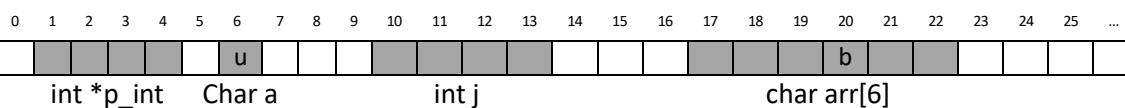
Agar suatu *pointer* menunjuk ke variabel lain, mula-mula *pointer* harus diisi dengan alamat memori yang ditunjuk.

Pernyataan di atas berarti bahwa p_int diberi nilai berupa alamat dari variabel j. Setelah pernyataan tersebut di eksekusi maka dapat dikatakan bahwa p_int menunjuk ke variabel j. Jika suatu variabel sudah ditunjuk oleh *pointer*. Maka, variabel yang ditunjuk oleh *pointer* dapat diakses melalui variabel itu sendiri ataupun melalui *pointer*.

Untuk mendapatkan nilai dari variabel yang ditunjuk *pointer*, gunakan tanda * di depan nama variabel *pointer*

Pointer juga merupakan variabel, karena itu *pointer* juga akan menggunakan *space memory* dan memiliki alamat sendiri

```
int *p_int;
```



Gambar 2-4 Ilustrasi Alokasi Pointer

C++	Output	Keterangan
<pre>int j,k; j =10; int *p_int; p_int = &j; cout<< j << endl; cout<< &j << endl; cout<< p_int << endl; cout<< &p_int << endl; cout<< *p_int << endl; k = *p_int; cout << k << endl;</pre>	<pre>10 x6 x6 X1 10 10</pre>	Nilai variabel j Alamat variabel j Nilai variabel p_int Alamat variabel p_int Nilai variabel yang ditunjuk p_int Nilai variabel k

Berikut ini contoh program sederhana menggunakan *pointer*:

```
1 #include <iostream>
2 #include <conio.h>
3 using namespace std;
4
5 int main(){
6     int x,y; //x dan y bertipe int
7     int *px; //px merupakan variabel pointer menunjuk ke variabel int
8     x =87;
9     px=&x;
10    y=*px;
11    cout<<"Alamat x= "<<&x<<endl;
12    cout<<"Isi px= "<<px<<endl;
13    cout<<"Isi X= "<<x<<endl;
14    cout<<"Nilai yang ditunjuk px= "<<*px<<endl;
15    cout<<"Nilai y= "<<y<<endl;
16    getch();
17    return 0;
18 }
```

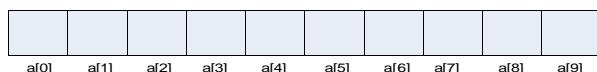
```
Alamat x= 0022FF14
Isi px= 0022FF14
Isi X= 87
Nilai yang ditunjuk px= 87
Nilai y= 87
```

Gambar 2-5 Output Pointer

2.2.3 Pointer dan Array

Ada keterhubungan yang kuat antara *array* dan *pointer*. Banyak operasi yang bisa dilakukan dengan *array* juga bisa dilakukan dengan *pointer*. Pendeklarasian *array*: int a[10];

Mendefinisikan *array* sebesar 10, kemudian blok dari objek *array* tersebut diberi nama a[0],a[1],a[2],....a[9].



Gambar 2-6 Array

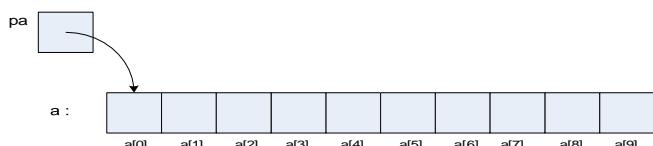
Notasi a[i] akan merujuk elemen ke-i dari *array*. Jika pa merupakan *pointer* yang menunjuk variabel bertipe *integer*, yang di deklarasikan sebagai berikut :

```
int *pa;
```

maka pernyataan :

```
pa=&a[0];
```

akan membuat pa menunjuk ke alamat dari elemen ke-0 dari variabel a. Sehingga, pa akan mengandung alamat dari a[0].



Gambar 2-7 Pointer dan Array

Sekarang, pernyataan :

```
x=*pa;
```

Akan menyalinkan isi dari $a[0]$ ke variabel x.

Jika pa akan menunjuk ke elemen tertentu dari *array*, maka pendefinisian $pa + 1$ akan menunjuk elemen berikutnya, $pa + i$ akan menunjuk elemen ke- i setelah pa , sedangkan $pa - i$ akan menunjuk elemen ke- i sebelum pa sehingga jika pa menunjuk ke $a[0]$ maka $* (pa + 1)$ akan mengandung isi elemen ke $a[1]$. $pa + i$ merupakan alamat dari $a[i]$, dan $* (pa + i)$ akan mengandung isi dari elemen $a[i]$.

```
1 #include <iostream>
2 #include <conio.h>
3 #define MAX 5
4 using namespace std;
5
6 int main(){
7     int i,j;
8     float nilai_total, rata_rata;
9     float nilai[MAX];
10    static int nilai_tahun[MAX][MAX]=
11    {
12        {0,2,2,0,0},
13        {0,1,1,1,0},
14        {0,3,3,3,0},
15        {4,4,0,0,4},
16        {5,0,0,0,5}
17    };
18    /*inisialisasi array dua dimensi */
19    for (i=0; i<MAX; i++){
20        cout<<"masukkan nilai ke-"<<i+1<<endl;
21        cin>>nilai[i];
22    }
23    cout<<"\n data nilai siswa :\n";
24    /*menampilkan array satu dimensi */
25    for (i=0; i<MAX; i++)
26        cout<<"nilai k-"<<i+1<< "=" <<nilai[i]<<endl;
27    cout<<"\n nilai tahunan : \n";
28
29    /* menampilkan array dua dimensi */
30    for(i=0; i<MAX; i++){
31        for(j=0; j<MAX; j++)
32            cout<<nilai_tahun[i][j];
33        cout<<"\n";
34    }
35    getch();
36    return 0;
37 }
```

2.2.4 Pointer dan String

A. String

String merupakan bentuk data yang sering digunakan dalam bahasa pemrograman untuk mengolah data teks atau kalimat. Dalam bahasa C pada dasarnya *string* merupakan kumpulan dari karakter atau *array* dari karakter.

Deklarasi variabel *string*:

```
char nama[50];
```

50 → menyatakan jumlah maksimal karakter dalam *string*.

Memasukkan data *string* dari *keyboard*:

```
gets(nama_array);
```

contoh: `gets(nama);`

jika menggunakan `cin()`:

contoh: `cin>>nama;`

Inisialisasi *string*:

```
char nama[] = {'s','t','r','u','k','d','a','t','\0'};
```

Merupakan variabel nama dengan isi data *string* "strukdat".

Bentuk inisialisasi yang lebih singkat:

```
char nama[] = "strukdat";
```

Menampilkan *string* bisa menggunakan `puts()` atau `cout()`:

```
puts(nama);  
cout << nama;
```

Untuk mengakses data *string* sepihalknya mengakses data pada *array*, pengaksesan dilakukan perkarakter sesuai dengan indeks setiap karakter dalam *string*.

Contoh :

```
Cout<<nama[3]; /*menampilkan karakter ke-3 dari string*/
```

B. Pointer dan String

Sesuai dengan penjelasan di atas , misalkan ada *string* :

"I am string"

Merupakan *array* dari karakter. Dalam representasi internal, *array* diakhiri dengan karakter '\0' sehingga program dapat menemukan akhir dari program. Panjang dari *storage* merupakan panjang dari karakter yang ada dalam tanda petik dua ditambah satu. Ketika karakter *string* tampil dalam sebuah program maka untuk mengaksesnya digunakan *pointer* karakter. Standar *input/output* akan menerima *pointer* dari awal karakter *array* sehingga konstanta *string* akan diakses oleh *pointer* mulai dari elemen pertama.

Jika *pmessage* di deklarasikan :

```
char *pmessage ;
```

Maka pernyataan berikut :

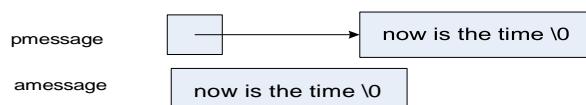
```
pmessage = "now is the time";
```

Akan membuat *pmessage* sebagai *pointer* pada karakter *array*. Ini bukan copy *string*, hanya *pointer* yang terlibat. C tidak menyediakan operator untuk memproses karakter *string* sebagai sebuah unit.

Ada perbedaan yang sangat penting diantara pernyataan berikut :

```
char amessage[] = "now is the time"; //merupakan array  
char *pmessage = "now is the time"; //merupakan pointer
```

Variabel *amessage* merupakan sebuah *array*, hanya cukup besar untuk menampung karakter-karakter sequence tersebut dan karakter null '\0' yang menginisialisasinya. Tiap-tiap karakter dalam *array* bisa saja berubah tapi variabel *amessage* akan selalu menunjuk apada *storage* yang sama. Di sisi lain, *pmessage* merupakan *pointer*, diinisialisasikan menunjuk konstanta *string*, *pointer* bisa di modifikasi untuk menunjuk kemanapun, tapi hasilnya tidak akan terdefinisi jika kamu mencoba untuk mengubah isi *string*.



2.3 Fungsi

Fungsi merupakan blok dari kode yang dirancang untuk melaksanakan tugas khusus dengan tujuan:

1. Program menjadi terstruktur, sehingga mudah dipahami dan mudah dikembangkan. Program dibagi menjadi beberapa modul yang kecil.
2. Dapat mengurangi pengulangan kode (duplicasi kode) sehingga menghemat ukuran program.

Pada umumnya fungsi memerlukan masukan yang dinamakan sebagai parameter. Masukan ini selanjutnya diolah oleh fungsi. Hasil akhir fungsi berupa sebuah nilai (nilai balik fungsi).

Bentuk umum sebuah fungsi:

```
tipe_keluaran nama_fungsi(daftar_parameter) {
    blok pernyataan fungsi ;
}
```

Jika penentu_tipe fungsi merupakan tipe dari nilai balik fungsi, bila tidak disebutkan maka akan dianggap (default) sebagai int.

Algoritma	C++
<pre>Program coba_fungsi Kamus x,y,z : integer function max3(input: a,b,c : integer) : integer algoritma input(x,y,z) output(max3(x,y,z)) function max3(input:a,b,c : integer) : integer kamus temp_max : integer algoritma temp_max ← a if (b>temp_max) then temp_max ← b if (c>temp_max) then temp_max ← c → temp_max</pre>	<pre>#include <conio.h> #include <iostream> #include <stdlib.h> using namespace std; int maks3(int a, int b, int c); /*mendeklarasikan prototype fungsi */ int main(){ system("cls"); int x,y,z; cout<<"masukkan nilai bilangan ke-1 ="; cin>>x; cout<<"masukkan nilai bilangan ke-2 ="; cin>>y; cout<<"masukkan nilai bilangan ke-3 ="; cin>>z; cout<<"nilai maksimumnya adalah = <<maks3(x,y,z); getch(); return 0; } /*badan fungsi */ int maks3(int a, int b, int c){ /* deklarasi variabel lokal dalam fungsi */ Int temp_max =a; if(b>temp_max) temp_max=b; if(c>temp_max) temp_max=c; return (temp_max); }</pre>

2.4 Prosedur

Dalam C sebenarnya tidak ada prosedur, semua berupa fungsi, termasuk main() pun adalah sebuah fungsi. Jadi prosedur dalam C merupakan fungsi yang tidak mengembalikan nilai, biasa diawali dengan kata kunci void di depan nama prosedur.

Bentuk umum sebuah prosedur:

```
void nama_prosedur (daftar_parameter) {  
    blok pernyataan prosedur ;  
}
```

Algoritma	C++
Program coba_procedur Kamus jum : integer procedure tulis(input: x: integer) Algoritma input(jum) tulis(jum) procedure tulis(input: x: integer) kamus i : integer algoritma i traversal [1..x] output("baris ke-", i+1)	#include <iostream> #include <conio.h> #include <stdlib.h> using namespace std; /*prototype fungsi */ void tulis(int x); int main() { System("cls"); int jum; cout << " jumlah baris kata=" ; cin >> jum; tulis(jum); getch(); return 0; } /*badan prosedur*/ void tulis(int x){ for (int i=0;i<x;i++) cout<<"baris ke-"< <i+1<<endl; </i+1<<endl; }

2.5 Parameter Fungsi

2.5.1 Paramater Formal dan Parameter Aktual

Parameter formal adalah variabel yang ada pada daftar parameter ketika mendefinisikan fungsi. Pada fungsi maks3() contoh diatas, a, b dan c merupakan parameter formal.

```
float perkalian (float x, float y) {  
    return (x*y);  
}
```

Pada contoh di atas x dan y adalah parameter formal.

Adapun parameter aktual adalah parameter (tidak selamanya menyatakan variabel) yang dipakai untuk memanggil fungsi.

```
x = perkalian(a, b);  
y = perkalian(10,30);
```

Dari pernyataan diatas a dan b merupakan parameter aktual, begitu pula 10 dan 30. parameter aktual tidak harus berupa variabel, melainkan bisa berupa konstanta atau ungkapan.

2.5.2 Cara melewaskan Parameter

A. Pemanggilan dengan Nilai (*call by value*)

Pada pemanggilan dengan nilai, nilai dari parameter aktual akan disalin kedalam parameter formal, jadi parameter aktual tidak akan berubah meskipun parameter formalnya berubah. Untuk lebih jelasnya perhatikan contoh berikut:

Algoritma	C++
<pre>Program coba_parameter_by_value Kamus a,b : integer procedure tukar(input: x,y : integer) Algoritma a < 4 b < 6 output(a,b) tukar(a,b) output(a,b) procedure tukar(input:x,y : integer) kamus temp : integer algoritma temp < x x < y y < temp output(x,y)</pre>	<pre>#include <iostream> #include <conio.h> #include <stdlib.h> using namespace std; /*prototype fungsi */ void tukar(int x, int y); int main () { int a, b; system("cls"); a=4; b=6; cout << "kondisi sebelum ditukar \n"; cout << " a = "<<a<<" b = "<<b<<endl; tukar(a,b); printf("kondisi setelah ditukar \n"); cout << " a = "<<a<<" b = "<<b<<endl; getch(); return 0; } void tukar (int x, int y) { int temp; temp = x; x = y; y = temp; cout << "nilai akhir pada fungsi tukar \n"; cout << " x = "<<x<<" y = "<<y<<endl; }</pre>

Hasil eksekusi :

Kondisi sebelum tukar

a = 4, b = 6

Nilai akhir pada fungsi tukar

x = 6, y = 4

Kondisi setelah tukar

a = 4, b = 6

Jelas bahwa pada pemanggilan fungsi tukar, yang melewaskan variabel a dan b tidak merubah nilai dari variabel tersebut. Hal ini dikarenakan ketika pemanggilan fungsi tersebut nilai dari a dan b disalin ke variabel formal yaitu x dan y.

B. Pemanggilan dengan Pointer (*call by pointer*)

Pemanggilan dengan *pointer* merupakan cara untuk melewaskan alamat suatu variabel ke dalam suatu fungsi. Dengan cara ini dapat merubah nilai dari variabel aktual yang dilewatkan ke dalam fungsi. Jadi cara ini dapat merubah variabel yang ada diluar fungsi.

Cara penulisan :

```
tukar(int *px, int *py) {
    int temp;
    temp = *px;
```

```

    *px = *py;
    *py = temp;
    ...
}

```

Cara pemanggilan:

```
tukar(&a, &b);
```

Pada ilustrasi tersebut, `*px` merupakan suatu variabel *pointer* yang menunjuk ke suatu variabel *interger*. Pada pemanggilan fungsi `tukar()`, `&a` dan `&b` menyatakan “alamat a” dan “alamat b”. dengan cara diatas maka variabel yang diubah dalam fungsi `tukar()` adalah variabel yang dilewatkan dalam fungsi itu juga, karena yang dilewatkan dalam fungsi adalah alamat dari variabel tersebut, jadi bukan sekedar disalin.

C. Pemanggilan dengan Referensi (*Call by Reference*)

Pemanggilan dengan referensi merupakan cara untuk melewaskan alamat suatu variabel kedalam suatu fungsi. Dengan cara ini dapat merubah nilai dari variabel aktual yang dilewatkan ke dalam fungsi. Jadi cara ini dapat merubah variabel yang ada diluar fungsi. Cara penulisan :

```

tukar(int &px, int &py) {
    int temp;
    temp = px;
    px = py;
    py = temp;
    ...
}

```

Cara pemanggilan:

```
tukar(a, b);
```

untuk melewaskan nilai dengan referensi, argumen dilalui ke fungsi seperti nilai lain. Jadi pada akhirnya, harus mendeklarasikan di parameter awal, serta untuk pemanggilan tidak perlu menggunakan paramter tambahan seperti pada *call by pointer*.

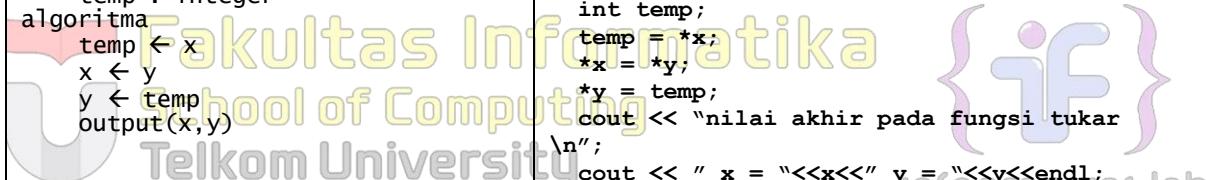
Algoritma	C++
<pre> Program coba_parameter_by_reference Kamus a,b : integer procedure tukar(input/output: x,y : integer) Algoritma a ← 4 b ← 6 output(a,b) tukar(a,b) output(a,b) procedure tukar(input/output :x,y : integer) kamus temp : integer algoritma temp ← x x ← y y ← temp output(x,y) </pre>	<pre> #include <iostream> #include <conio.h> #include <stdlib.h> using namespace std; /*prototype fungsi */ void tukar(int &x, int &y); int main () { int a, b; system("cls"); a=4; b=6; cout << "kondisi sebelum ditukar \n"; cout << " a = "<<a<<" b = "<<b<<endl; tukar(a,b); printf("kondisi setelah ditukar \n"); cout << " a = "<<a<<" b = "<<b<<endl; getch(); return 0; } void tukar (int &x, int &y) { int temp; temp = x; x = y; y = temp; cout<< "nilai akhir pada fungsi tukar \n"; cout << " x = "<<x<<" y = "<<y<<endl; </pre>

Call By
Reference

	}
--	---

Algoritma	C++
<pre> Program coba_parameter_by_pointer Kamus a,b : integer procedure tukar(input/output: x,y : integer) Algoritma a ← 4 b ← 6 output(a,b) tukar(a,b) output(a,b) procedure tukar(input/output :x,y : integer) kamus temp : integer algoritma temp ← x x ← y y ← temp output(x,y) </pre>	<pre> #include <iostream> #include <conio.h> #include <stdlib.h> using namespace std; /*prototype fungsi */ void tukar(int *x, int *y); int main () { int a, b; system("cls"); a=4; b=6; cout << "kondisi sebelum ditukar \n"; cout << " a = "<<a<<" b = "<<b<<endl; tukar(&a,&b); printf("kondisi setelah ditukar \n"); cout << " a = "<<a<<" b = "<<b<<endl; getch(); return 0; } void tukar (int *x, int *y) { int temp; temp = *x; *x = *y; *y = temp; cout << "nilai akhir pada fungsi tukar \n"; cout << " x = "<<x<<" y = "<<y<<endl; } </pre>

Call by
Pointer



Modul 3 ABSTRACT DATA TYPE (ADT)

TUJUAN PRAKTIKUM

1. Memahami konsep Abstract Data Type (ADT) dan penggunaannya dalam pemrograman.

3.1 Abstract Data Type (ADT)

ADT adalah TYPE dan sekumpulan PRIMITIF (operasi dasar) terhadap TYPE tersebut. Selain itu, dalam sebuah ADT yang lengkap, disertakan pula definisi invarian dari TYPE dan aksioma yang berlaku. ADT merupakan definisi STATIK.

Definisi type dari sebuah ADT dapat mengandung sebuah definisi ADT lain. Misalnya :

3. ADT waktu yang terdiri dari ADT JAM dan ADT DATE
4. Garis terdiri dari dua buah ADT POINT

SEGI4 yang terdiri dari pasangan dua buah POINT (*Top,Left*) dan (*Bottom,Right*)

TYPE diterjemahkan menjadi *type* terdefinisi dalam bahasa yang bersangkutan. Jika dalam bahasa C menggunakan struct PRIMITIF, dalam konteks prosedural, diterjemahkan menjadi fungsi atau prosedur. PRIMITIF dikelompokan menjadi:

1. Konstruktor/Kreator, pemebentuk nilai *type*. Semua objek (variabel) bertipe tersebut harus melalui konstruktor. Biasanya namanya diawali *Make*.
2. *Selector*, untuk mengakses tipe komponen(biasanya namanya diawali *Get*).
3. Prosedur pengubah nilai komponen (biasanya namanya diawali *Get*).
4. Tipe validator komponen, yang dipakai untuk mentest apakah dapat membentuk tipe sesuai dengan batasan.
5. Destruktor/Dealokator yaitu untuk “menghancurkan” nilai objek/variabel (sekaligus memori penyimpanannya).
6. Baca/Tulis, untuk interface dengan *input/output device*.
7. Operator relasional, terhadap tipe tersebut untuk mendefinisikan lebih besar, lebih kecil, sama dengan dan sebagainya.
8. Aritmatika terhadap tipe tersebut, karena biasanya aritmatika dalam bahasa C hanya terdefinisi untuk bilangan numerik.
9. Konversi dari tipe tersebut ke tipe dasar dan sebaliknya.

ADT biasanya diimplementasikan menjadi dua buah modul utama dan 1 modul *interface* program utama (*driver*). Dua modul tersebut adalah sebagai berikut:

1. Definisi/Spesifikasi *Type* dan Primitif/*Header* fungsi (.h)
 - Spesifikasi *type* sesuai dengan kaidah bahasa yang dipakai
 - Spesifikasi dari primitif sesuai dengan kaidah dalam konteks prosedural,yaitu:
 - Fungsi : nama, domain, *range*, dan prekondisi jika ada
 - Prosedur : *Initial state*, *Final state*, dan proses yang dilakukan
2. *Body/realisasi* dari primitif (.c)

Berupa kode program dalam bahasa yang bersangkutan (dalam praktikum ini berarti dengan bahasa C++). Realisasi fungsi dan prosedur harus sedapat mungkin memanfaatkan *selector* dan konstruktor. Untuk memahami lebih jelas mengenai konsep ADT, perhatikan ilustrasi berikut.

Algoritma	C++
<pre> Program coba_ADT Type mahasiswa < nim : char[10] nilai1,nilai2 : integer Kamus mhs : mahasiswa procedure inputMhs(input/output m : mahasiswa) function rata2(input: m : mahasiswa) : real Algoritma inputMhs(mhs) output(rata2(mhs)) procedure inputMhs(input/output m : mahasiswa) kamus algoritma input(m.nim, m.nilai1, m.nilai2) function rata2(input: m : mahasiswa) : real kamus algoritma → (m.nilai1 + m.nilai1) / 2 </pre>	<pre> #include <iostream> #include <conio.h> #include <stdlib.h> using namespace std; struct mahasiswa{ char nim[10]; int nilai1,nilai2; }; void inputMhs (mahasiswa &m); float rata2 (mahasiswa m); int main() { mahasiswa mhs; inputMhs (mhs); cout << "rata-rata = " << rata2 (mhs); return 0; } void inputMhs (mahasiswa &m) { cout << "input nama = "; cin >> (m).nim; cout << "input nilai = "; cin >> (m).nilai1; cout << "input nilai2 = "; cin >> (m).nilai2; } float rata2 (mahasiswa m) { return (m.nilai1+m.nilai2)/2; } </pre> <div style="border: 1px solid black; padding: 5px; margin-left: 20px;"> Definisi/ Spesifikasi Type dan Primitif / Header fungsi (&.h) </div> <div style="border: 1px solid black; padding: 5px; margin-left: 20px;"> Body/ relisasi dari primitif (&.c) </div>

Untuk menerapkan konsep ADT, kita harus memisah deklarasi tipe, variabel, dan fungsi dari program ke dalam sebuah file.h dan memisahkan definisi fungsi dari program ke sebuah file.cpp. Sehingga jika kita menerapkan konsep ADT berdasarkan contoh program di atas, bentuk code program akan dipisah menjadi seperti berikut.

Algoritma	C++
<pre> Program coba_ADT Type mahasiswa < nim : char[10] nilai1,nilai2 : integer Kamus mhs : mahasiswa procedure inputMhs(i/o m : mahasiswa) function rata2(input: m : mahasiswa) : real Algoritma inputMhs(mhs) output(rata2(mhs)) procedure inputMhs(input/output m : mahasiswa) kamus algoritma input(m.nim, m.nilai1, m.nilai2) </pre>	<pre> mahasiswa.h #ifndef MAHASISWA_H_INCLUDED #define MAHASISWA_H_INCLUDED struct mahasiswa{ char nim[10]; int nilai1, nilai2; }; void inputMhs (mahasiswa &m); float rata2 (mahasiswa m); #endif // MAHASISWA_H_INCLUDED mahasiswa.cpp void inputMhs (mahasiswa &m) { cout << "input nama = "; cin >> (m).nim; cout << "input nilai = "; cin >> (m).nilai1; cout << "input nilai2 = "; cin >> (m).nilai2; } </pre>

<pre> function rata2(input: m : mahasiswa) : real kamus algoritma → (m.nilai1 + m.nilai2) / 2 </pre>	<pre> float rata2(mahasiswa m){ return (m.nilai1+m.nilai2)/2; } main.cpp </pre>
	<pre> #include <iostream> #include <conio.h> #include <stdlib.h> #include "mahasiswa.cpp" using namespace std; int main() { mahasiswa mhs; inputMhs(mhs); cout << "rata-rata = " << rata2(mhs); return 0; } </pre>

3.2 Latihan

1. Buat program yang dapat menyimpan data mahasiswa (max. 10) ke dalam sebuah *array* dengan field nama, nim, uts, uas, tugas, dan nilai akhir. Nilai akhir diperoleh dari FUNGSI dengan rumus $0.3*uts+0.4*uas+0.3*tugas$.
2. Buatlah ADT pelajaran sebagai berikut di dalam file “pelajaran.h”:

tipe pelajaran <
 namaMapel : string
 kodeMapel : string
 >
 fungsi create_pelajaran(namapel : string, kodepel : string) →
 pelajaran
 prosedur tampil_pelajaran(pel : pelajaran)

Buatlah implementasi ADT pelajaran pada file “pelajaran.cpp”

Cobalah hasil implementasi ADT pelajaran pada file “main.cpp”

```

using namespace std;
int main(){
    string namapel = "Struktur Data";
    string kodepel = "STD";
    pelajaran pel = create_pelajaran(namapel,kodepel);
    tampil_pelajaran(pel);

    return 0;
}

```

Gambar 3-1 Main.cpp pelajaran

Contoh *output* hasil:

```

nama pelajaran : struktur Data
nilai : STD

```

Gambar 3-2 *output* pelajaran

3. Buatlah program dengan ketentuan :
 - 2 buah *array 2D integer* berukuran 3×3 dan 2 buah *pointer integer*
 - fungsi/prosedur yang menampilkan isi sebuah *array integer 2D*
 - fungsi/prosedur yang akan menukar isi dari 2 *array integer 2D* pada posisi tertentu

- fungsi/prosedur yang akan menukar isi dari variabel yang ditunjuk oleh 2 buah *pointer*



Modul 4 SINGLE LINKED LIST (BAGIAN PERTAMA)

TUJUAN PRAKTIKUM

2. Memahami penggunaan *linked list* dengan *pointer* operator- operator dalam program.
3. Memahami operasi-operasi dasar dalam *linked list*.
4. Membuat program dengan menggunakan *linked list* dengan *prototype* yang ada

4.1 *Linked List* dengan *Pointer*

Linked list (biasa disebut *list* saja) adalah salah satu bentuk struktur data (representasi penyimpanan) berupa serangkaian elemen data yang saling berkait (berhubungan) dan bersifat fleksibel karena dapat tumbuh dan mengerut sesuai kebutuhan. Data yang disimpan dalam *Linked list* bisa berupa data tunggal atau data majemuk. Data tunggal merupakan data yang hanya terdiri dari satu data (variabel), misalnya: nama bertipe *string*. Sedangkan data majemuk merupakan sekumpulan data (*record*) yang di dalamnya terdiri dari berbagai tipe data, misalnya: Data Mahasiswa, terdiri dari Nama bertipe *string*, NIM bertipe *long integer*, dan Alamat bertipe *string*.

Linked list dapat diimplementasikan menggunakan *Array* dan *Pointer* (*Linked list*).

Yang akan kita gunakan adalah *pointer*, karena beberapa alasan, yaitu :

1. *Array* bersifat statis, sedangkan *pointer* dinamis.
2. Pada *linked list* bentuk datanya saling bergandengan (berhubungan) sehingga lebih mudah memakai *pointer*.
3. Sifat *linked list* yang fleksibel lebih cocok dengan sifat *pointer* yang dapat diatur sesuai kebutuhan.
4. Karena *array* lebih susah dalam menangani *linked list*, sedangkan *pointer* lebih mudah.
5. *Array* lebih cocok pada kumpulan data yang jumlah elemen maksimumnya sudah diketahui dari awal.

Dalam implementasinya, pengaksesan elemen pada *Linked list* dengan *pointer* bisa menggunakan (->) atau tanda titik (.).

Model-model dari ADT *Linked list* yang kita pelajari adalah :

1. *Single Linked list*
2. *Double Linked list*
3. *Circular Linked list*
4. *Multi Linked list*
5. *Stack* (Tumpukan)
6. *Queue* (Antrian)
7. *Tree*
8. *Graph*

Setiap model ADT *Linked list* di atas memiliki karakteristik tertentu dan dalam penggunaannya disesuaikan dengan kebutuhan.

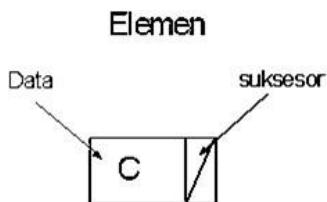
Secara umum operasi-operasi ADT pada *Linked list*, yaitu :

1. Penciptaan dan inisialisasi *list* (*Create List*).
2. Penyisipan elemen *list* (*Insert*).
3. Penghapusan elemen *list* (*Delete*).
4. Penelusuran elemen *list* dan menampilkannya (*View*).
5. Pencarian elemen *list* (*Searching*).
6. Pengubahan isi elemen *list* (*Update*).

4.2 Single Linked List

Single Linked list merupakan model ADT *Linked list* yang hanya memiliki satu arah *pointer*.

Komponen elemen dalam *single linked list*:



Gambar 4-1 Elemen *Single Linked list*

Keterangan:

Elemen: segmen-semen data yang terdapat dalam suatu *list*.

Data: informasi utama yang tersimpan dalam sebuah elemen.

Suksesor: bagian elemen yang berfungsi sebagai penghubung antar elemen.

Sifat dari *Single Linked list*:

1. Hanya memerlukan satu buah *pointer*.
2. *Node* akhir menunjuk ke Nil kecuali untuk *list circular*.
3. Hanya dapat melakukan pembacaan maju.
4. Pencarian sequensial dilakukan jika data tidak terurut.
5. Lebih mudah ketika melakukan penyisipan atau penghapusan di tengah *list*.

Istilah-istilah dalam *Single Linked list* :

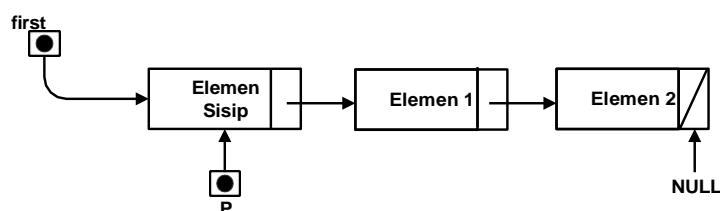
1. *first/head*: *pointer* pada *list* yang menunjuk alamat elemen pertama *list*.
2. *next*: *pointer* pada elemen yang berfungsi sebagai *successor* (penunjuk) alamat elemen di depannya.
3. Null/Nil: artinya tidak memiliki nilai, atau tidak mengacu ke mana pun, atau kosong.
4. *Node/simpul/elemen*: merupakan tempat penyimpanan data pada suatu memori tertentu.

Gambaran sederhana *single linked list* dengan elemen kosong:



Gambar 4-2 *Single Linked list* dengan Elemen Kosong

Gambaran sederhana *single linked list* dengan 3 elemen:



Gambar 4-3 *Single Linked list* dengan 3 Elemen

Contoh deklarasi struktur data *single linked list*:

```
1  /*file : list.h*/
2  #ifndef LIST_H_INCLUDED
3  #define LIST_H_INCLUDED
4
5  #define Nil NULL
6  #define info(P) (P)->info
7  #define next(P) (P)->next
8  #define first(L) ((L).first)
9  using namespace std;
10 /*deklarasi record dan struktur data list*/
11 typedef int infotype;
12 typedef struct elmlist *address;
13 struct elmlist {
14     infotype info;
15     address next;
16 };
17
18 struct list{
19     address first;
20 };
21 #endif // TEST_H_INCLUDED
```

Misal untuk data mahasiswa yang terdiri dari nama dan nim.

```
1  /*file : list.h*/
2  #ifndef LIST_H_INCLUDED
3  #define LIST_H_INCLUDED
4
5  #define Nil NULL
6  #define info(P) (P)->info
7  #define next(P) (P)->next
8  #define first(L) ((L).first)
9
10 using namespace std;
11 /*deklarasi record dan struktur data list*/
12 struct mahasiswa{
13     char nama[30]
14     char nim[10]
15 }
16 typedef mahasiswa infotype;
17
18 typedef struct elmlist *address;
19 struct elmlist {
20     infotype info;
21     address next;
22 };
23
24 struct list{
25     address first;
26 };
27 #endif // TEST_H_INCLUDED
```

4.2.1 Pembentukan Komponen-Komponen List

A. Pembentukan List

Adalah sebuah proses untuk membentuk sebuah *list* baru. Biasanya nama fungsi yang digunakan `createList()`. Fungsi ini akan mengeset nilai awal *list* yaitu `first(list)` dan `last(list)` dengan nilai Nil.

B. Pengalokasian Memori

Adalah proses untuk mengalokasikan memori untuk setiap elemen data yang ada dalam *list*. Fungsi yang biasanya digunakan adalah nama fungsi yang biasa digunakan `alokasi()`.

Sintak alokasi pada C:

```
P = (address) malloc ( sizeof (elmlist));
```

Keterangan:

- | | |
|---------|--|
| P | = variabel <i>pointer</i> yang mengacu pada elemen yang dialokasikan. |
| address | = tipe data <i>pointer</i> dari tipe data elemen yang akan dialokasikan. |
| Elmlist | = tipe data atau <i>record</i> elemen yang dialokasikan. |

Contoh deklarasi struktur data *single linked list*:

Misal untuk data mahasiswa yang terdiri dari nama dan nim.

```
address alokasi(mahasiswa m){  
    address p = (address)malloc(sizeof(elmlist));  
    info(p) = m;  
    return p;  
}
```

Namun pada Cpp. Penggunaan **malloc** dapat dipersingkat menggunakan sintak **new**.

Sintak alokasi pada Cpp:

```
P = new elmlist;
```

Keterangan:

- | | |
|---------|--|
| P | = variabel <i>pointer</i> yang mengacu pada elemen yang dialokasikan. |
| address | = tipe data <i>pointer</i> dari tipe data elemen yang akan dialokasikan. |

Contoh deklarasi struktur data *single linked list*:

Misal untuk data mahasiswa yang terdiri dari nama dan nim.

```
address alokasi(mahasiswa m){  
    address p = new elmlist;  
    info(p) = m;  
    return p;  
}
```

C. Dealokasi

Untuk menghapus sebuah *memory address* yang tersimpan atau telah dialokasikan dalam bahasa pemrograman C digunakan sintak **free**, sedangkan pada Cpp digunakan sintak **delete**, seperti berikut.

Sintak pada C:

```
free( p );
```

Sintak pada Cpp:

```
delete p;
```

D. Pengecekan List

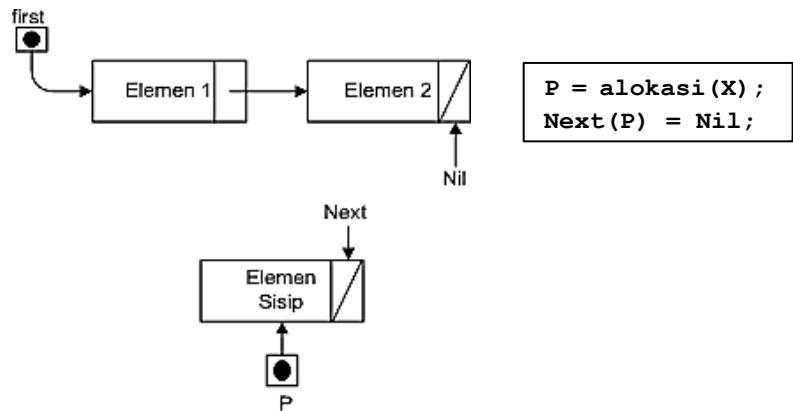
Adalah fungsi untuk mengecek apakah *list* tersebut kosong atau tidak. Akan mengembalikan nilai true jika *list* kosong dan nilai false jika *list* tidak kosong. Fungsi yang digunakan adalah **isEmpty()**.

4.2.2 Insert

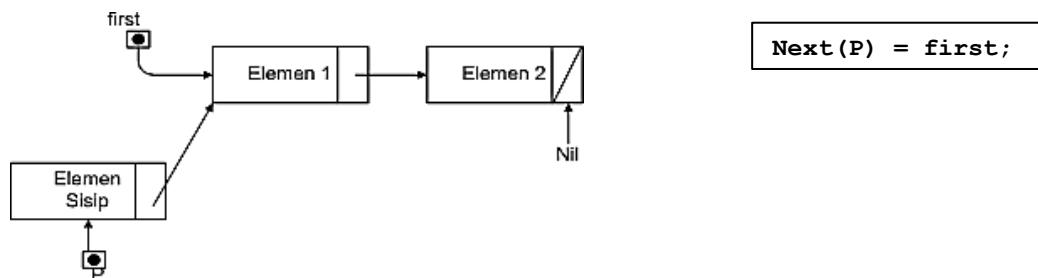
A. Insert First

Merupakan metode memasukkan elemen data ke dalam *list* yang diletakkan pada awal *list*.

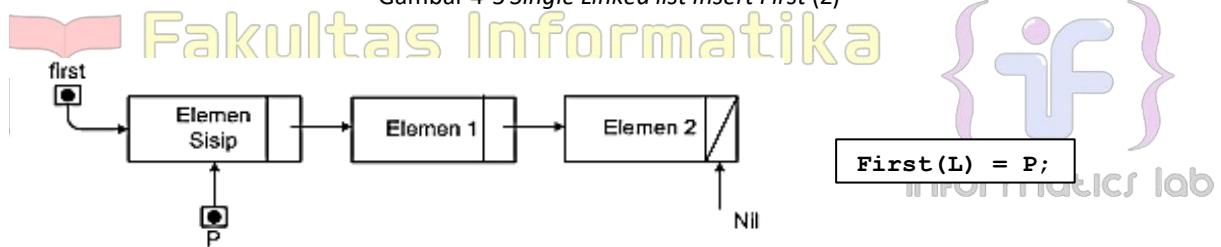
Langkah-langkah dalam proses *insert first*:



Gambar 4-4 Single Linked list Insert First (1)



Gambar 4-5 Single Linked list Insert First (2)



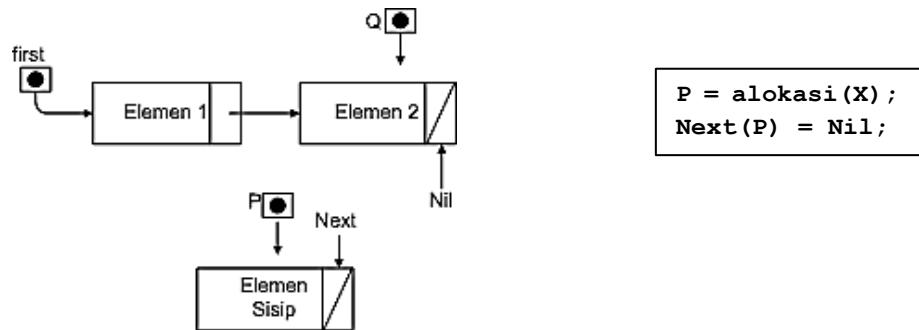
Gambar 4-6 Single Linked list Insert First (3)

```
/* contoh syntax insert first */
void insertFirst(List &L, address &P){
    next (P) = first(L);
    first(L) = P;
}
```

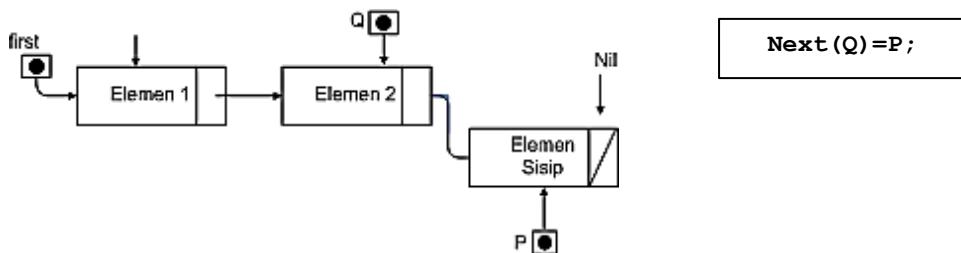
B. Insert Last

Merupakan metode memasukkan elemen data ke dalam *list* yang diletakkan pada akhir *list*.

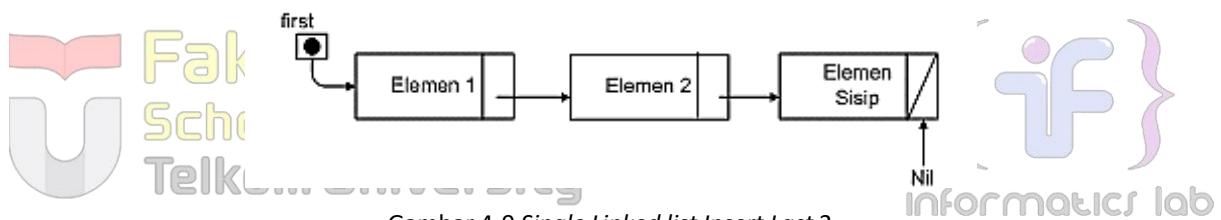
Langkah dalam *insert last* :



Gambar 4-7 Single Linked list Insert Last 1



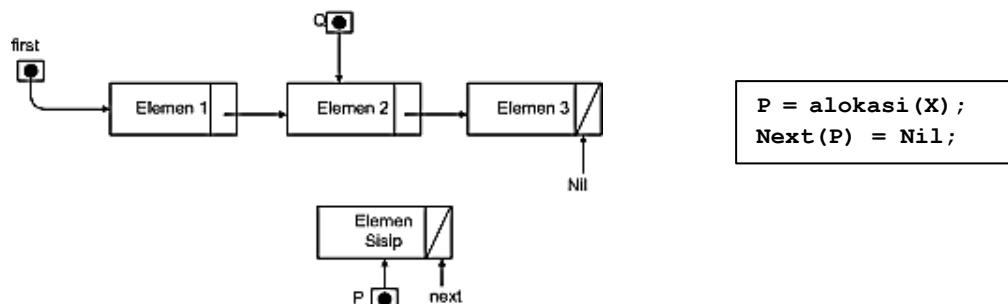
Gambar 4-8 Single Linked list Insert Last 2



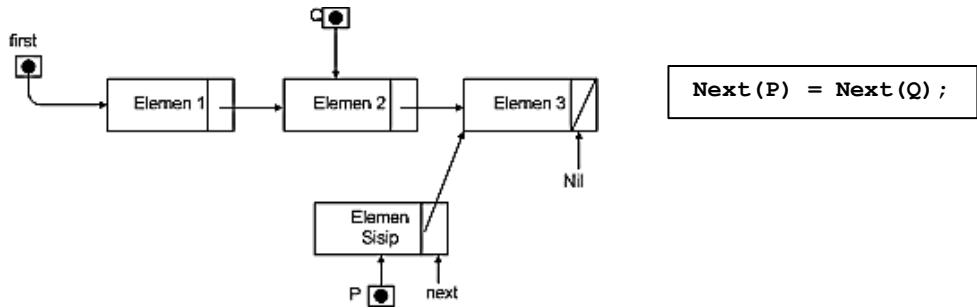
Gambar 4-9 Single Linked list Insert Last 3

C. Insert After

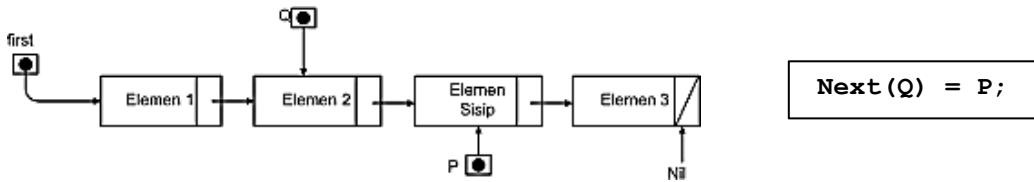
Merupakan metode memasukkan data ke dalam *list* yang diletakkan setelah *node* tertentu yang ditunjuk oleh *user*. Langkah dalam *insert after*:



Gambar 4-10 Single Linked list Insert After 1



Gambar 4-11 Single Linked list Insert After 2



Gambar 4-12 Single Linked list Insert After 3

4.2.3 View

Merupakan operasi dasar pada *list* yang menampilkan isi *node*/simpul dengan suatu penelusuran *list*. Mengunjungi setiap *node* kemudian menampilkan data yang tersimpan pada *node* tersebut.

Semua fungsi dasar diatas merupakan bagian dari ADT dari single *linked list*, dan aplikasi pada bahasa pemrograman Cp semua ADT tersebut tersimpan dalam file *.c dan file *.h.

```

1  /*file : list .h*/
2  /* contoh ADT list berkait dengan representasi fisik pointer*/
3  /* representasi address dengan pointer*/
4  /* info tipe adalah integer */
5  #ifndef list_H
6  #define list_H
7  #include "boolean.h"
8  #include <stdio.h>
9  #define Nil NULL
10 #define info(P) (P)->info
11 #define next(P) (P)->next
12 #define first(L) ((L).first)
13
14 /*deklarasi record dan struktur data list*/
15 typedef int infotype;
16 typedef struct elmlist *address;
17 struct elmlist{
18     infotype info;
19     address next;
20 };
21
22 /* definisi list : */
23 /* list kosong jika First(L)=Nil */
24 /* setiap elemen address P dapat diacu info(P) atau next(P) */
25 struct list {
26     address first;
27 };
28 //***** pengecekan apakah list kosong *****/
29 boolean ListEmpty(list L);
30 /*mengembalikan nilai true jika list kosong*/
31
32 //***** pembuatan list kosong *****/
33 void CreateList(list &L);

```

```

34  /* I.S. sembarang
35      F.S. terbentuk list kosong*/
36
37  ***** manajemen memori *****/
38  void dealokasi(address P);
39  /* I.S. P terdefinisi
40      F.S. memori yang digunakan P dikembalikan ke sistem */
41
42  ***** penambahan elemen *****/
43  void insertFirst(list &L, address P);
44  /* I.S. sembarang, P sudah dialokasikan
45      F.S. menempatkan elemen beralamat P pada awal list */
46
47  void insertAfter(list &L, address P, address Prec);
48  /* I.S. sembarang, P dan Prec alamat salah satu elemen list
49      F.S. menempatkan elemen beralamat P sesudah elemen beralamat Prec */
50
51  void insertLast(list &L, address P);
52  /* I.S. sembarang, P sudah dialokasikan
53      F.S. menempatkan elemen beralamat P pada akhir list */
54
55  ***** proses semua elemen list *****/
56  void printInfo(list L);
57  /* I.S. list mungkin kosong
58      F.S. jika list tidak kosong menampilkan semua info yang ada pada list */
59
60  int nbList(list L);
61  /* mengembalikan jumlah elemen pada list */
62
63  #endiff

```

4.3 Latihan

- Buatlah ADT Single Linked list sebagai berikut di dalam file “singlelist.h”:

```

Type infotype : int
Type address : pointer to ElmList

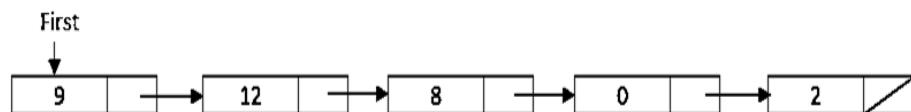
Type ElmList <
    info : infotype
    next : address
>

Type List : < First : address >
    prosedur CreateList( in/out L : List )
    fungsi alokasi( x : infotype ) : address
    prosedur dealokasi( in/out P : address )
    prosedur printInfo( in L : List )
    prosedur insertFirst( in/out L : List, in P : address )

```

Kemudian buat implementasi ADT Single Linked list pada file “singlelist.cpp”.

Adapun isi data



Gambar 4-13 Ilustrasi elemen

Cobalah hasil implementasi ADT pada file “main.cpp”

```

int main()
{
    List L;
    address P1, P2, P3, P4, P5 = NULL;
    createList(L);

    P1 = alokasi(2);
    insertFirst(L,P1);

    P2 = alokasi(0);
    insertFirst(L,P2);

    P3 = alokasi(8);
    insertFirst(L,P3);

    P4 = alokasi(12);
    insertFirst(L,P4);

    P5 = alokasi(9);
    insertFirst(L,P5);

    printInfo(L)
    return 0;
}

```

```

9 12 8 0 2

Process returned 0 (0x0)   execution time : 0.019 s
Press any key to continue.

```

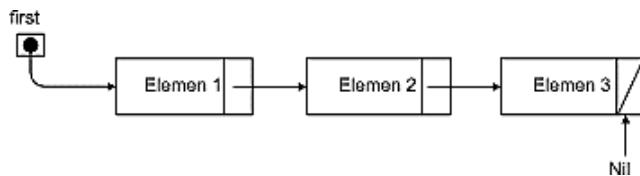


4.4 Delete

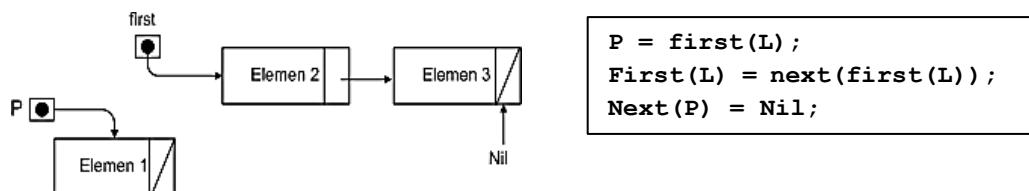
A. Delete First

Adalah pengambilan atau penghapusan sebuah elemen pada awal *list*.

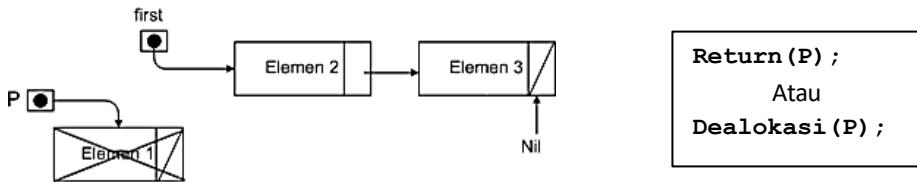
Langkah-langkah dalam *delete first*:



Gambar 4-15 Single Linked List Delete First 1



Gambar 4-16 Single Linked list Delete First 2



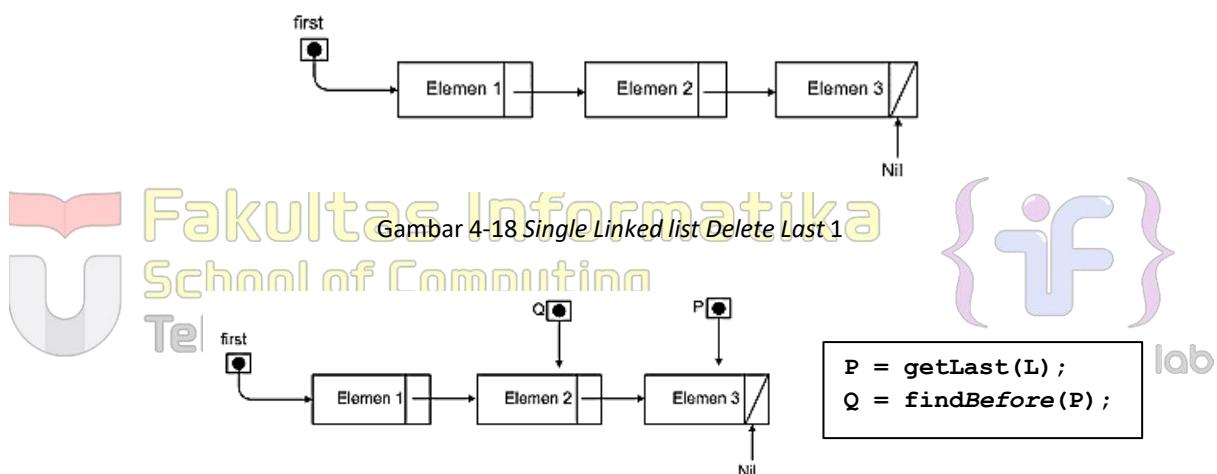
Gambar 4-17 Single Linked list Delete First 3

```
/* contoh syntax delete first */
void deleteFirst(List &L, address &P) {
    P = first(L);
    first(L) = next(first(L));
    next(P) = null;
}
```

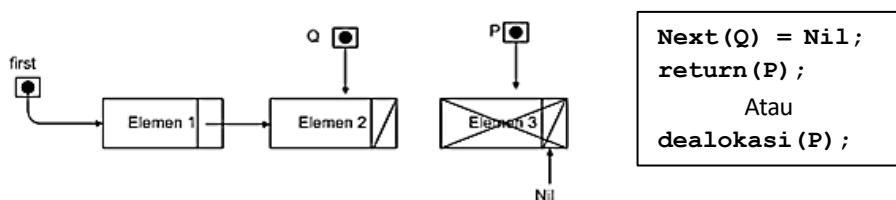
B. Delete Last

Merupakan pengambilan atau penghapusan suatu elemen dari akhir *list*.

Langkah-langkah dalam *delete last*:



Gambar 4-18 Single Linked list Delete Last 1

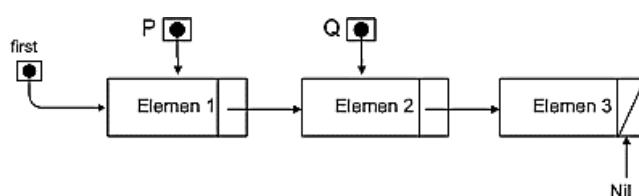


Gambar 4-20 Single Linked list Delete Last 3

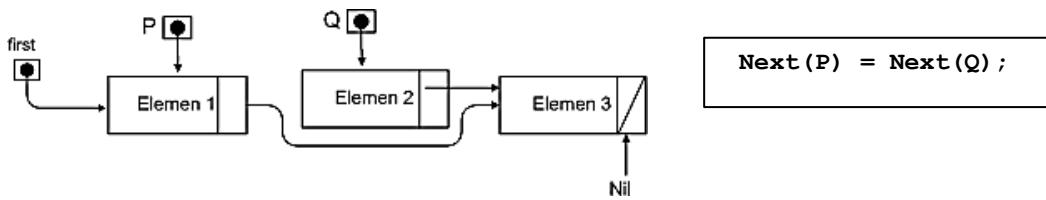
C. Delete After

Merupakan pengambilan atau penghapusan *node* setelah *node* tertentu.

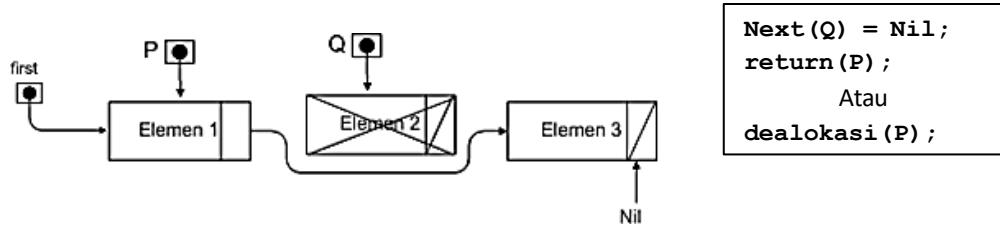
Langkah-langkah dalam *delete after*:



Gambar 4-21 Single Linked list Delete After 1



Gambar 4-22 Single Linked list Delete After 2



Gambar 4-23 Single Linked list Delete After 3

D. Delete Elemen

Adalah operasi yang digunakan untuk menghapus dan membebaskan memori yang dipakai oleh elemen tersebut.

Fungsi yang biasanya dipakai:

1. fungsi dealokasi(P) : membebaskan memori yang dipakai oleh elemen P.
2. fungsi delAll(L) : membebaskan semua memori yang dipakai elemen – elemen yang ada pada list L. Hasil akhir list L menjadi kosong.

Semua operasi-operasi dasar *list* biasa disebut dengan operasi primitif. Primitif-primitif dalam *list* ini merupakan bagian dari ADT *list* yang tersimpan dalam file *.h dan file *.cpp, dengan rincian file *.h untuk menyimpan prototipe primitif-primitif atau fungsi-fungsi dan menyimpan tipe data yang dipergunakan dalam primitif *list* tersebut.

Untuk bisa mengakses semua primitif tersebut yaitu dengan meng-*include* terhadap file *.h-nya.

4.5 Update

Merupakan operasi dasar pada *list* yang digunakan untuk mengupdate data yang ada di dalam *list*. Dengan operasi *update* ini kita dapat meng-update data-data *node* yang ada di dalam *list*. Proses *update* biasanya diawali dengan proses pencarian terhadap data yang akan *di-update*.

Modul 5 SINGLE LINKED LIST (BAGIAN KEDUA)

TUJUAN PRAKTIKUM

5. Memahami penggunaan *linked list* dengan *pointer* operator- operator dalam program.
6. Memahami operasi-operasi dasar dalam *linked list*.
7. Membuat program dengan menggunakan *linked list* dengan *prototype* yang ada

5.1 *Searching*

Searching merupakan operasi dasar *list* dengan melakukan aktivitas pencarian terhadap *node* tertentu. Proses ini berjalan dengan mengunjungi setiap *node* dan berhenti setelah *node* yang dicari ketemu. Dengan melakukan operasi *searching*, operasi-operasi seperti *insert after*, *delete after*, dan *update* akan lebih mudah.

Semua fungsi dasar diatas merupakan bagian dari ADT dari singgle *linked list*, dan aplikasi pada bahasa pemrograman Cp semua ADT tersebut tersimpan dalam file *.c dan file *.h.

```
1  /*file : list .h*/
2  /* contoh ADT list berkait dengan representasi fisik pointer*/
3  /* representasi address dengan pointer*/
4  /* info tipe adalah integer */
5  #ifndef list_H
6  #define list_H
7  #include "boolean.h"
8  #include <stdio.h>
9  #define Nil NULL
10 #define info(P) (P)->info
11 #define next(P) (P)->next
12 #define first(L) ((L).first)
13
14 /*deklarasi record dan struktur data list*/
15 typedef int infotype;
16 typedef struct elmlist *address;
17 struct elmlist{
18     infotype info;
19     address next;
20 };
21
22 /* definisi list : */
23 /* list kosong jika First(L)=Nil */
24 /* setiap elemen address P dapat diacu info(P) atau next(P) */
25 struct list {
26     address first;
27 };
28 /****** pengecekan apakah list kosong *****/
29 boolean ListEmpty(list L);
30 /*mengembalikan nilai true jika list kosong*/
31
32 /****** pembuatan list kosong *****/
33 void CreateList(list &L);
34 /* I.S. sembarang
35   F.S. terbentuk list kosong*/
36
37 /****** manajemen memori *****/
38 void dealokasi(address P);
39 /* I.S. P terdefinisi
40   F.S. memori yang digunakan P dikembalikan ke sistem */
41
42
43 /****** pencarian sebuah elemen list *****/
44 address findElm(list L, infotype X);
45 /* mencari apakah ada elemen list dengan info(P) = x
```

```

46     jika ada, mengembalikan address elemen tab tsb, dan Nil jika sebaliknya
47 */
48
49 boolean fFindElm(list L, address P);
50 /* mencari apakah ada elemen list dengan alamat P
51     mengembalikan true jika ada dan false jika tidak ada */
52
53 address findBefore(list L, address P);
54 /* mengembalikan address elemen sebelum P
55     jika prec berada pada awal list, maka mengembalikan nilai Nil */
56
57 /***** penambahan elemen *****/
58 void insertFirst(list &L, address P);
59 /* I.S. sembarang, P sudah dialokasikan
60     F.S. menempatkan elemen beralamat P pada awal list */
61
62 void insertAfter(list &L, address P, address Prec);
63 /* I.S. sembarang, P dan Prec alamt salah satu elemen list
64     F.S. menempatkan elemen beralamat P sesudah elemen beralamat Prec */
65
66 void insertLast(list &L, address P);
67 /* I.S. sembarang, P sudah dialokasikan
68     F.S. menempatkan elemen beralamat P pada akhir list */
69
70 /***** penghapusan sebuah elemen *****/
71 void delFirst(list &L, address &P);
72 /* I.S. list tidak kosong
73     F.S. adalah alamat dari alamat elemen pertama list
74     sebelum elemen pertama list dihapus
75     elemen pertama list hilang dan list mungkin menjadi kosong
76     first elemen yang baru adalah successor first elemen yang lama */
77
78 void delLast(list &L, address &P);
79 /* I.S. list tidak kosong
80     F.S. adalah alamat dari alamat elemen terakhir list
81     sebelum elemen terakhir list dihapus
82     elemen terakhir list hilang dan list mungkin menjadi kosong
83     last elemen yang baru adalah successor last elemen yang lama */
84
85 void delAfter(list &L, address &P, address Prec);
86 /* I.S. list tidak kosng, Prec alamat salah satu elemen list
87     F.S. P adalah alamatdari next(Prec), menghapus next(Prec) dari list */
88
89 void delP (list &L, infotype X);
90 /* I.S. sembarang
91     F.S. jika ada elemen list dengan alamat P, dimana info(P)=X, maka P
92     dihapus
93     dan P di-dealokasi, jika tidak ada maka list tetap
94     list mungkin akan menjadi kosong karena penghapusan */
95
96 /***** proses semua elemen list *****/
97 void printInfo(list L);
98 /* I.S. list mungkin kosong
99     F.S. jika list tidak kosong menampilkan semua info yang ada pada list */
100
101 int nbList(list L);
102 /* mengembalikan jumlah elemen pada list */
103
104 /***** proses terhadap list *****/
105 void delAll(list &L);
106 /* menghapus semua elemen list dan semua elemen di-dealokasi */
107
108 void invertList(list &L);
109 /* I.S. sembarang
110     F.S. elemen - elemen list dibalik */
111 void copyList(list L1, list &L2)
112 /* I.S. L1 sembarang

```

```

113     F.S. L1 = L2, L1 dan L2 menunjuk pada elemen yang sama */
114
115     list fCopyList(list L);
116     /* mengembalikan list yang merupakan salinan dari L */
117 #endif

```

5.2 Latihan

2. Buatlah ADT *Single Linked list* sebagai berikut di dalam file “**singlist.h**”:

```

Type infotype : int
Type address : pointer to ElmList

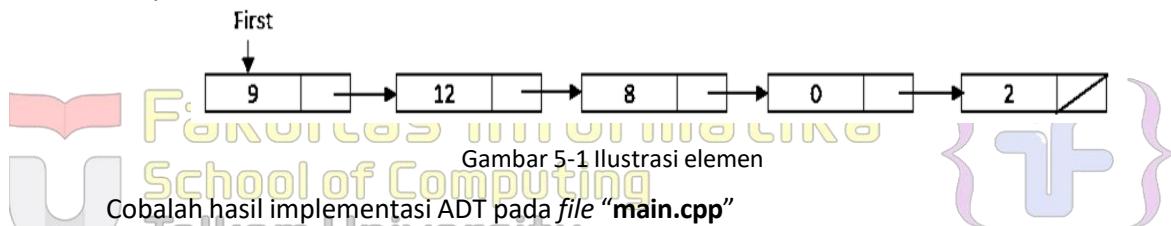
Type ElmList <
    info : infotype
    next : address
>

Type List : < First : address >
prosedur CreateList( in/out L : List )
fungsi alokasi( x : infotype ) : address
prosedur dealokasi( in/out P : address )
prosedur printInfo( in L : List )
prosedur insertFirst( in/out L : List, in P : address )

```

Kemudian buat implementasi ADT *Single Linked list* pada file “**singlist.cpp**”.

Adapun isi data



Cobalah hasil implementasi ADT pada file “**main.cpp**”

```

int main()
{
    List L;
    address P1, P2, P3, P4, P5 = NULL;
    createList(L);

    P1 = alokasi(2);
    insertFirst(L,P1);

    P2 = alokasi(0);
    insertFirst(L,P2);

    P3 = alokasi(8);
    insertFirst(L,P3);

    P4 = alokasi(12);
    insertFirst(L,P4);

    P5 = alokasi(9);
    insertFirst(L,P5);

    printInfo(L)
    return 0;
}

```

```
9 12 8 0 2
Process returned 0 (0x0)    execution time : 0.019 s
Press any key to continue.
```

Gambar 5-2 *Output* singlelist

3. Carilah elemen dengan info 8 dengan membuat fungsi baru.
fungsi findElm(L : List, x : infotype) : address

```
8 ditemukan dalam list
Process returned 0 (0x0)    execution time : 0.020 s
Press any key to continue.
```

Gambar 5-3 *Output* pencarian 8

4. Hitunglah jumlah total info seluruh elemen ($9+12+8+0+2=31$).

```
Total info dari kelima elemen adalah 31
Process returned 0 (0x0)    execution time : 0.019 s
Press any key to continue.
```

Gambar 5-4 *Output* total info elemen



Modul 6 DOUBLE LINKED LIST (BAGIAN PERTAMA)

TUJUAN PRAKTIKUM

1. Memahami konsep modul *linked list*.
2. Mengaplikasikan konsep *double linked list* dengan menggunakan *pointer* dan dengan bahasa C

6.1 Double Linked List

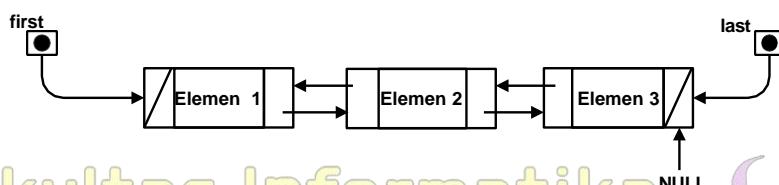
Double Linked list adalah *linked list* yang masing – masing elemennya memiliki 2 *successor*, yaitu *successor* yang menunjuk pada elemen sebelumnya (*prev*) dan *successor* yang menunjuk pada elemen sesudahnya (*next*).

Gambar berikut menunjukkan bentuk *Double Linked list* dengan elemen kosong:



Gambar 6-1 *Double Linked list* dengan Elemen Kosong

Gambar berikut menunjukkan bentuk *Double Linked list* dengan 3 elemen:



Gambar 6-2 *Double Linked list* dengan 3 Elemen

Double linked list juga menggunakan dua buah *successor* utama yang terdapat pada *list*, yaitu *first* (*successor* yang menunjuk elemen pertama) dan *last* (*successor* yang menunjuk elemen terakhir *list*).

Komponen-komponen dalam *double linked list*:

1. *First* : *pointer* pada *list* yang menunjuk pada elemen pertama *list*.
2. *Last* : *pointer* pada *list* yang menunjuk pada elemen terakhir *list*.
3. *Next* : *pointer* pada elemen sebagai *successor* yang menunjuk pada elemen didepannya.
4. *Prev* : *pointer* pada elemen sebagai *successor* yang menunjuk pada elemen dibelakangnya.

Contoh pendeklarasian struktur data untuk *double linked list*:

```
1 #ifndef doublelist_H
2 #define doublelist_H
3 #include "boolean.h"
4 #define Nil NULL
5 #define info(P) (P)->info
6 #define next(P) (P)->next
7 #define prev(P) (P)->prev
8 #define first(L) ((L).first)
9 #define last(L) ((L).last)
10
11 /*deklarasi record dan struktur data double linked list*/
12 typedef int infotype;
13 typedef struct elmlist *address;
14 struct elmlist {
15     infotype info;
16     address next;
17     address prev;
18 };
19
```

```

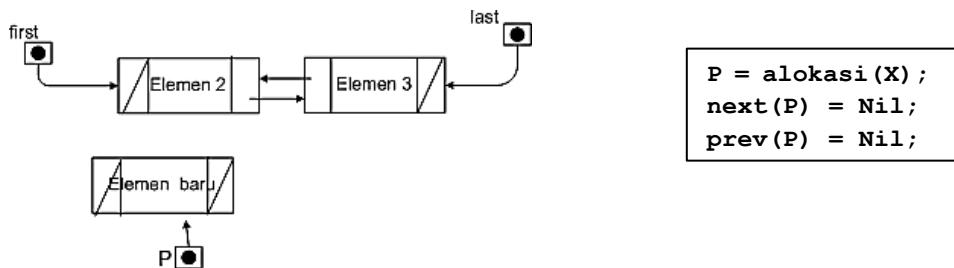
20  /* definisi list: */
21  /* list kosong jika First(L)=Nil */
22  struct list{
23      address first;
24      address last;
25  };
26  #endif

```

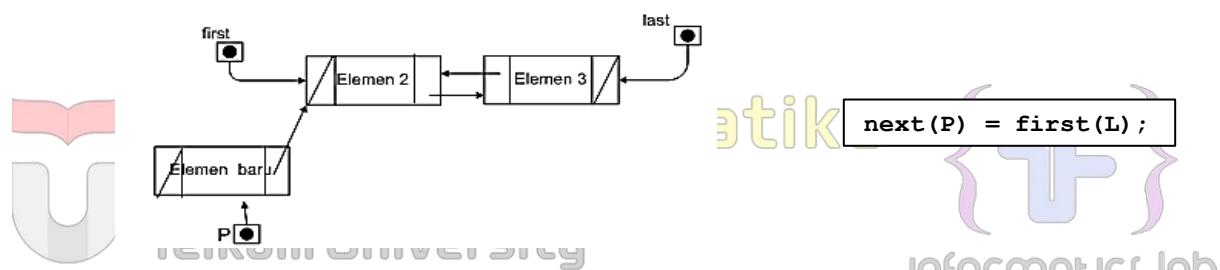
6.1.1 Insert

A. Insert First

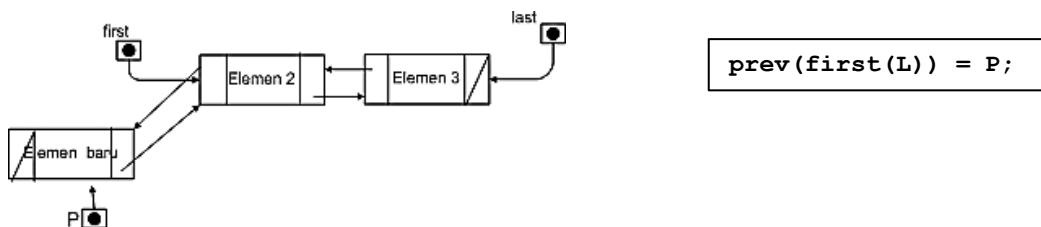
Langkah-langkah dalam proses *insert first*:



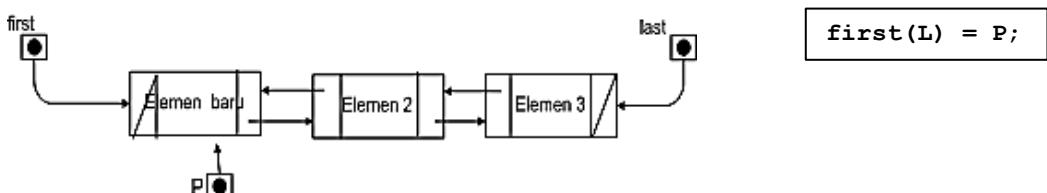
Gambar 6-3 Double Linked list Insert First 1



Gambar 6-4 Double Linked list Insert First 2



Gambar 6-5 Double Linked list Insert First 3



Gambar 6-6 Double Linked list Insert First 4

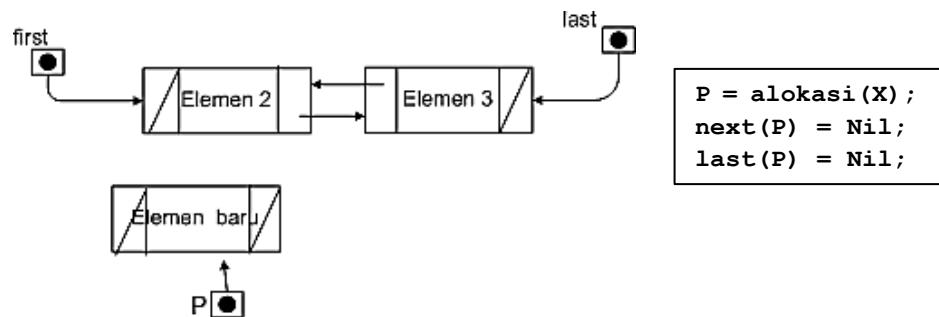
```

void insertFirst(list &L, address &P){
    next(P) = first(L);
    prev(first(L)) = P;
    first(L) = P;
}

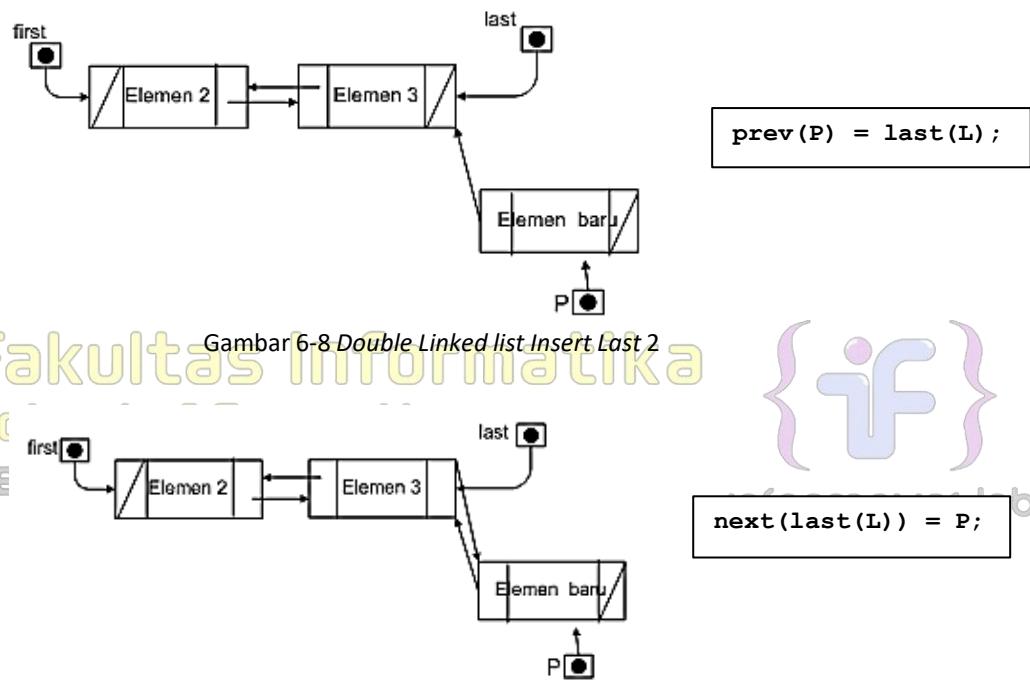
```

B. Insert Last

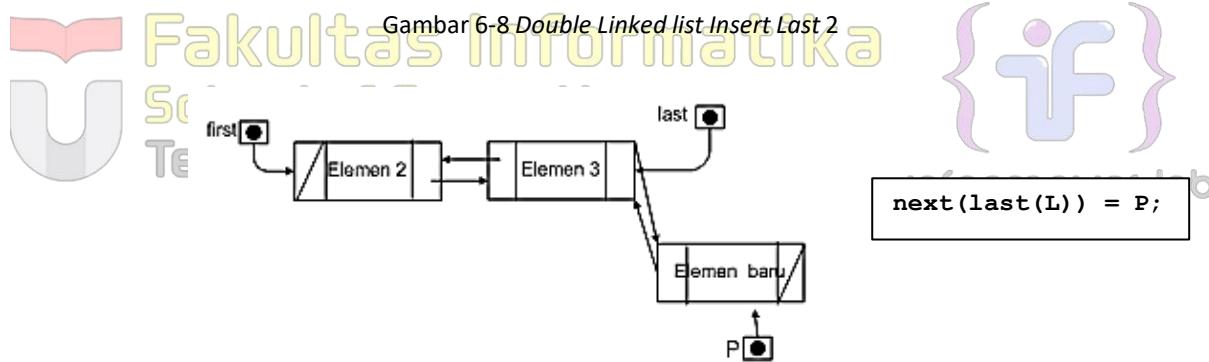
Langkah-langkah dalam proses *insert last*:



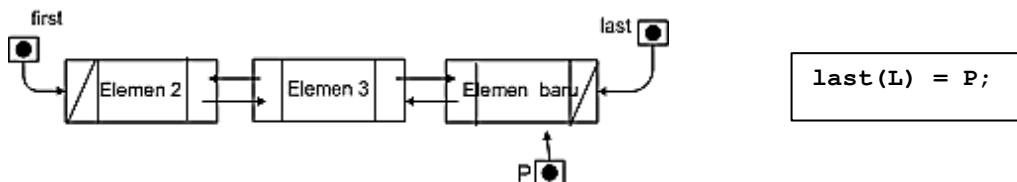
Gambar 6-7 Double Linked list Insert Last 1



Gambar 6-8 Double Linked list Insert Last 2



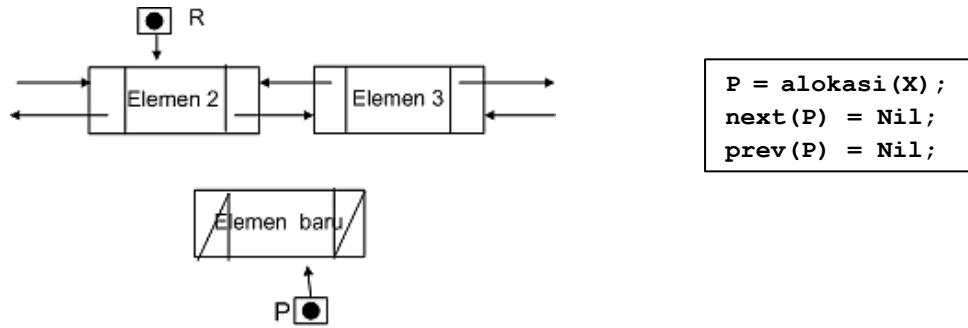
Gambar 6-9 Double Linked list Insert Last 3



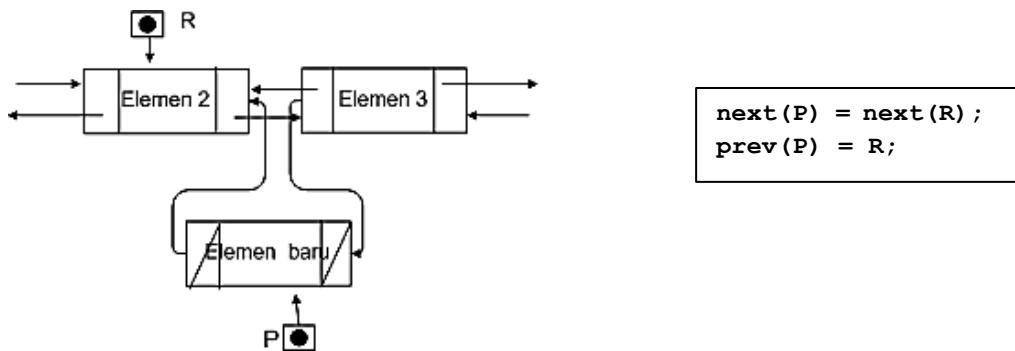
Gambar 6-10 Double Linked list Insert Last 4

C. Insert After

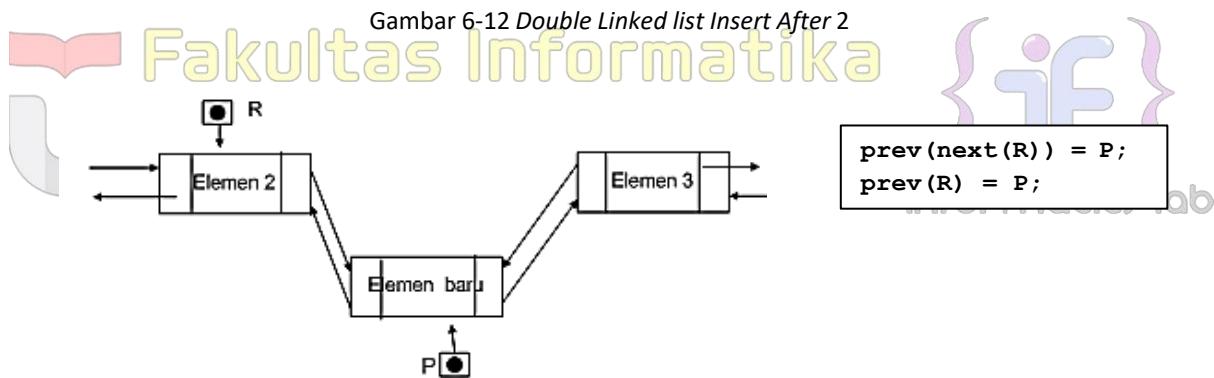
Langkah-langkah dalam proses *insert after*:



Gambar 6-11 Double Linked list Insert After 1



Gambar 6-12 Double Linked list Insert After 2



Gambar 6-13 Double Linked list Insert After 3

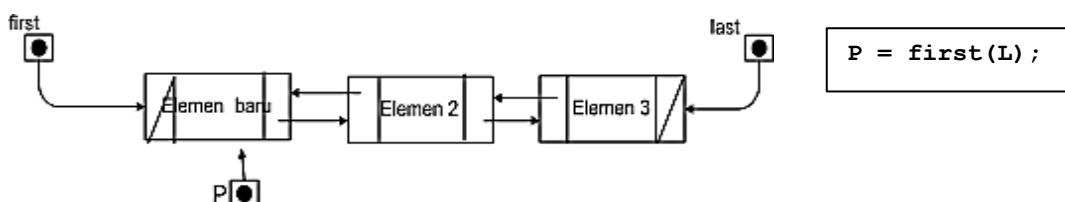
D. Insert Before

Diatas hanya dijelaskan tentang *insert after*. *Insert before* hanya kebalikan dari *insert after*. Perbedaan *Insert After* dan *Insert Before* terletak pada pencarian elemennya.

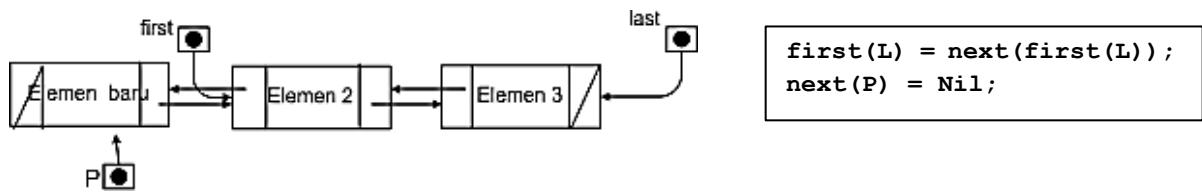
6.1.2 Delete

A. Delete First

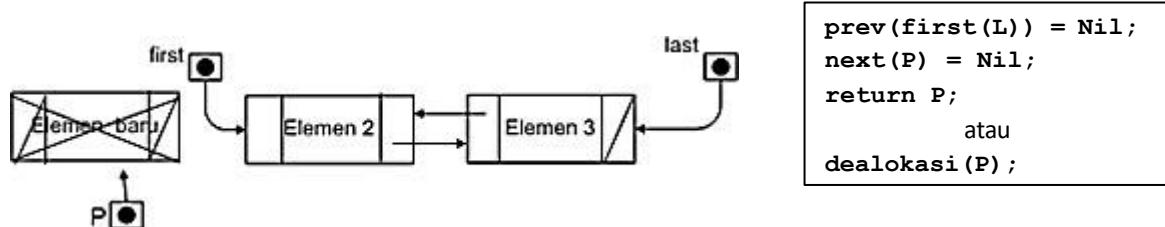
Langkah-langkah dalam proses *delete first*:



Gambar 6-14 Double Linked list Delete First 1



Gambar 6-15 Double Linked list Delete First 2

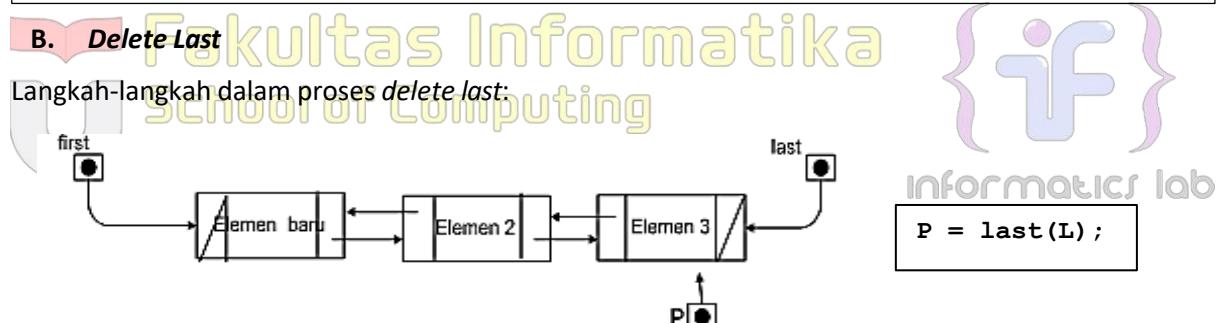


Gambar 6-16 Double Linked list Delete First 3

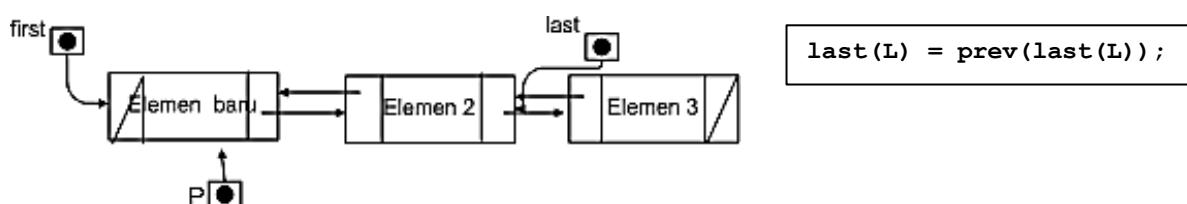
```

/* contoh sintak delete first */
void deleteFirst(list &L, address &P){
    P = first(L);
    first(L) = next(first(L));
    prev(P) = null;
    prev(first(L)) = null;
    next(P) = null;
}

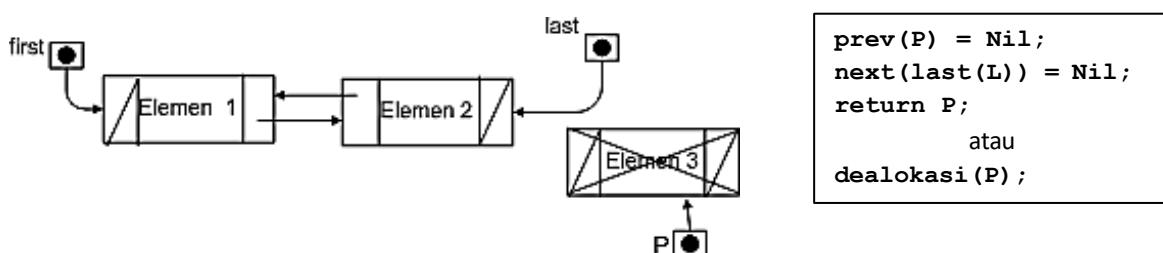
```



Gambar 6-17 Double Linked list Delete Last 1



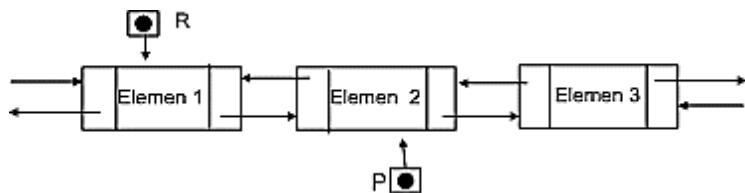
Gambar 6-18 Double Linked list Delete Last 2



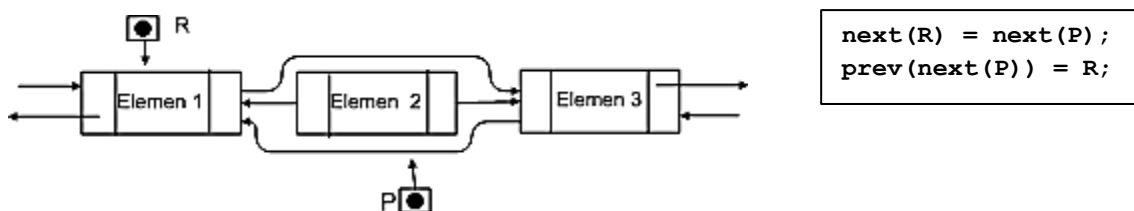
Gambar 6-19 Double Linked list Delete Last 3

C. Delete After

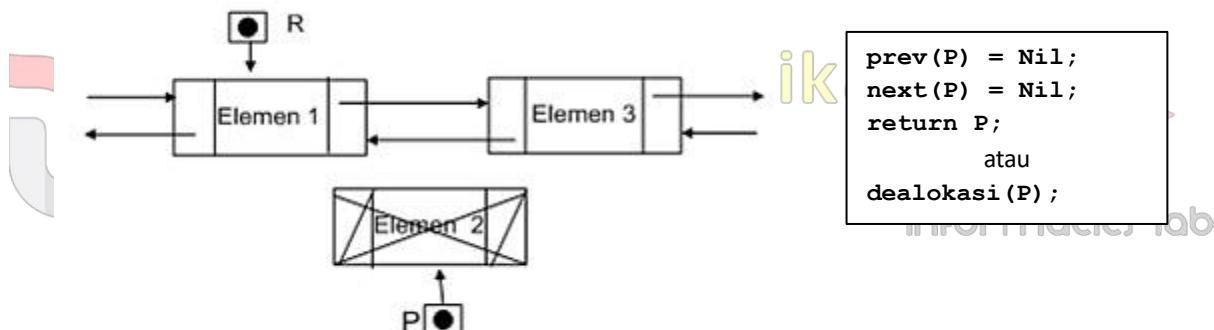
Langkah-langkah dalam proses *delete after*:



Gambar 6-20 Double Linked list Delete After 1



Gambar 6-21 Double Linked list Delete After 2



Gambar 6-22 Double Linked list Delete After 3

D. Delete Before

Diatas hanya dijelaskan tentang *delete after*. *Delete before* hanya kebalikan dari *delete after*. Perbedaan *Delete After* dan *Delete Before* terletak pada pencarian elemennya.

E. Update, View, dan Searching

Proses pencarian, *update* data dan *view* data pada dasarnya sama dengan proses pada *single linked list*. Hanya saja pada *double linked list* lebih mudah dalam melakukan proses akses elemen, karena bisa melakukan iterasi maju dan mundur.

Seperti halnya *single linked list*, *double linked list* juga mempunyai ADT yang pada dasarnya sama dengan ADT yang ada pada *single linked list*.

```

1  /*file : doublelist .h*/
2  /* contoh ADT list berkait dengan representasi fisik pointer*/
3  /* representasi address dengan pointer*/
4  /* info tipe adalah integer */
5  #ifndef doublelist_H
6  #define doublelist_H
7
8  #include <stdio.h>
9  #define Nil NULL
10 #define info(P) (P)->info
11 #define next(P) (P)->next
12 #define prev(P) (P)->prev
13 #define first(L) ((L).first)
14 #define last(L) ((L).last)
15
16 typedef int infotype;
17 typedef struct elmlist *address;
18 /* pendefinisian tipe data bentukan elemen list
19   dengan dua successor, yaitu next dan prev */
20 struct elmlist{
21     infotype info;
22     address prev;
23     address next;
24 };
25
26 /* definisi double linked list : list kosong jika first(L)=Nil
27   setiap elemen address P dapat diacu info(P) atau next(P)
28   elemen terakhir adalah last
29   nama tipe list yang dipakai adalah 'list', sama dengan pada single list*/
30 struct list {
31     address first,last;
32 };
33
34 /** Deklarasi fungsi primitif lain**/
35 /** Sama dengan Single Linked list */

```



informatics lab

6.2 Latihan

- Buatlah ADT Double Linked list sebagai berikut di dalam file “doublelist.h”:

```

Type infotype : kendaraan <
    nopol : string
    warna : string
    thnBuat : integer
>
Type address : pointer to ElmList
Type ElmList <
    info : infotype
    next :address
    prev : address
>

Type List <
    First : address
    Last : address
>
prosedur CreateList( in/out L : List )
fungsi alokasi( x : infotype ) : address
prosedur dealokasi( in/out P : address )
prosedur printInfo( in L : List )
prosedur insertLast( in/out L : List, in P : address )

```

Buatlah implementasi ADT Double Linked list pada file “doublelist.cpp” dan coba hasil implementasi ADT pada file “main.cpp”.

Contoh Output :

```

masukkan nomor polisi: D001
masukkan warna kendaraan: hitam
masukkan tahun kendaraan: 90

masukkan nomor polisi: D003
masukkan warna kendaraan: putih
masukkan tahun kendaraan: 70

masukkan nomor polisi: D001
masukkan warna kendaraan: merah
masukkan tahun kendaraan: 80
nomor polisi sudah terdaftar

masukkan nomor polisi: D004
masukkan warna kendaraan: kuning
masukkan tahun kendaraan: 90

DATA LIST 1

no polisi : D004
warna      : kuning
tahun       : 90
no polisi : D003
warna      : putih
tahun       : 70
no polisi : D001
warna      : hitam
tahun       : 90

```

Gambar 6-23 *Output* kasus kendaraan

- Carilah elemen dengan nomor polisi D001 dengan membuat fungsi baru.
fungsi `findElm(L : List, x : infotype) : address`



**Fakultas
School of Computing
Telkom University**



Gambar 6-24 *Output* mencari nomor polisi

- Hapus elemen dengan nomor polisi D003 dengan prosedur *delete*.
 - prosedur `deleteFirst(in/out L : List, in/out P : address)`
 - prosedur `deleteLast(in/out L : List, in/out P : address)`
 - prosedur `deleteAfter(in Prec : address, in/out P : address)`

```

Masukkan Nomor Polisi yang akan dihapus : D003
Data dengan nomor polisi D003 berhasil dihapus.

DATA LIST 1

Nomor Polisi : D004
Warna        : kuning
Tahun         : 90
Nomor Polisi : D001
Warna        : hitam
Tahun         : 90

```

Gambar 6-25 *Output* menghapus data nomor polisi

Modul 7 STACK

TUJUAN PRAKTIKUM

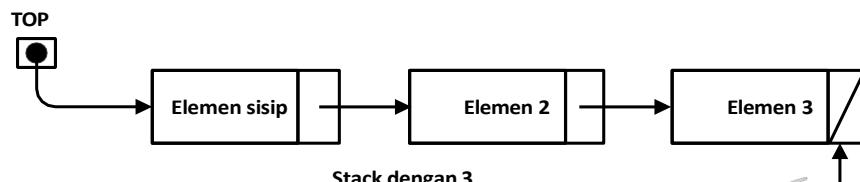
1. Memahami konsep *stack*.
2. Mengimplementasikan *stack* dengan menggunakan representasi *pointer* dan tabel.
3. Memahami konsep *queue*.
4. Mengimplementasikan *queue* dengan menggunakan *pointer* dan tabel.

7.1 Pengertian Stack

Stack merupakan salah satu bentuk struktur data dimana prinsip operasi yang digunakan seperti tumpukan. Seperti halnya tumpukan, elemen yang bisa diambil terlebih dahulu adalah elemen yang paling atas, atau elemen yang pertama kali masuk, prinsip ini biasa disebut **LIFO** (*Last In First Out*).

7.2 Komponen-Komponen dalam Stack

Komponen – komponen dalam *stack* pada dasarnya sama dengan komponen pada *single linked list*. Hanya saja akses pada *stack* hanya bisa dilakukan pada awal *stack* saja.



Gambar 7-1 Stack dengan 3 Elemen

Seperti terlihat pada gambar diatas bentuk *stack* mirip seperti *list* linier, yang terdiri dari elemen – elemen yang saling terkait. Komponen utama dalam *stack* yang berfungsi untuk mengakses data dalam *stack* adalah elemen paling awal saja yang disebut "Top". Pendeklarasian tipedata *stack*:

```
1 #ifndef stack_H
2 #define stack_H
3
4 #define Nil NULL
5 #define info(P) (P)->info
6 #define next(P) (P)->next
7 #define Top(S) ((S).Top)
8
9 typedef int infotype; /* tipe data dalam stack */
10 typedef struct elmStack *address;
11 /* tipe data pointe untuk elemen stack */
12 struct elmStack {
13     infotype info;
14     address next;
15 }; /*tipe data elemen stack */
16
17 /* pendeklarasian tipe data stack */
18 struct stack {
19     address Top;
20 };
21 #endif
```

Keterangan:

1. Dalam *stack* hanya terdapat TOP.

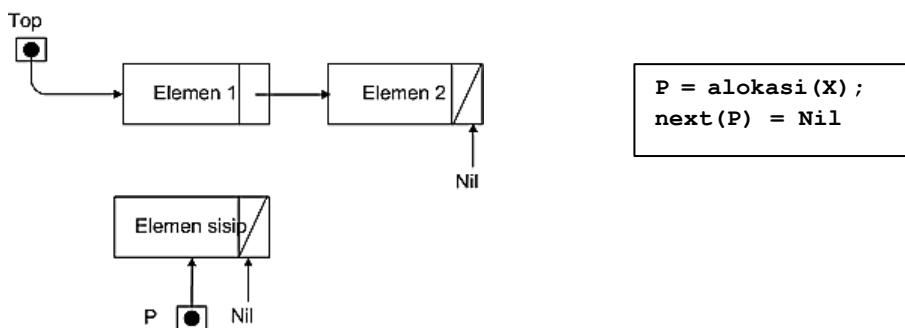
2. Tipe *address* adalah tipe elemen *stack* yang sama dengan elemen dalam *list* lainnya.

7.3 Operasi-Operasi dalam Stack

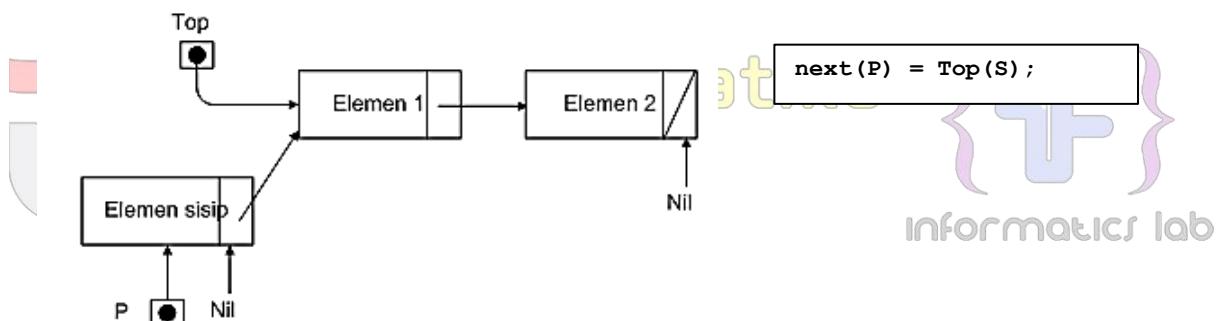
Dalam *stack* ada dua operasi utama, yaitu operasi penyisipan (*Push*) dan operasi pengambilan (*Pop*).

7.3.1 Push

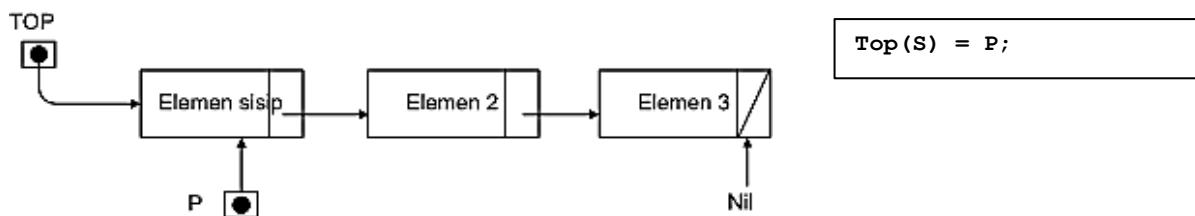
Adalah operasi menyisipkan elemen pada tumpukan data. Fungsi ini sama dengan fungsi *insert first* pada *list* biasa. Langkah – langkah dalam proses *Push*:



Gambar 7-2 Stack Push 1



Gambar 7-3 Stack Push 2



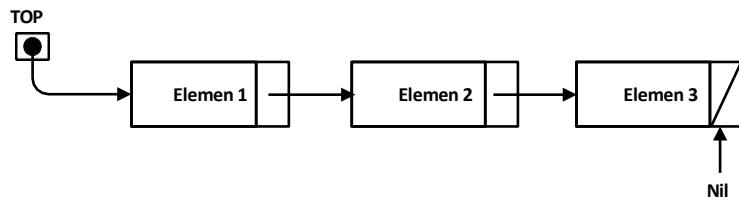
Gambar 7-4 Stack Push 3

```
/* buat dahulu elemen yang akan disisipkan */
address createElm(int x){
    address p = alokasi(x);
    next(p) = null;
    return p;
}

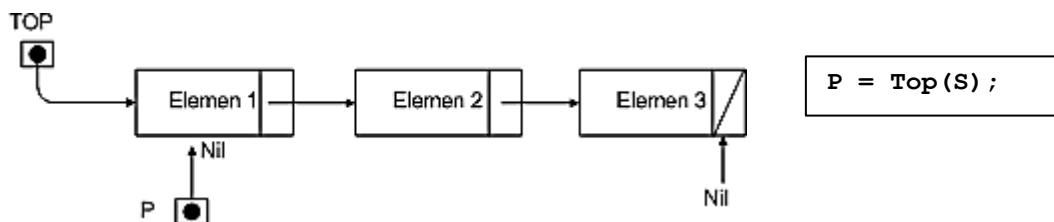
/* contoh sintak push */
void push(address p){
    next(p) = top(s);
    top(s) = p;
}
```

7.3.2 Pop

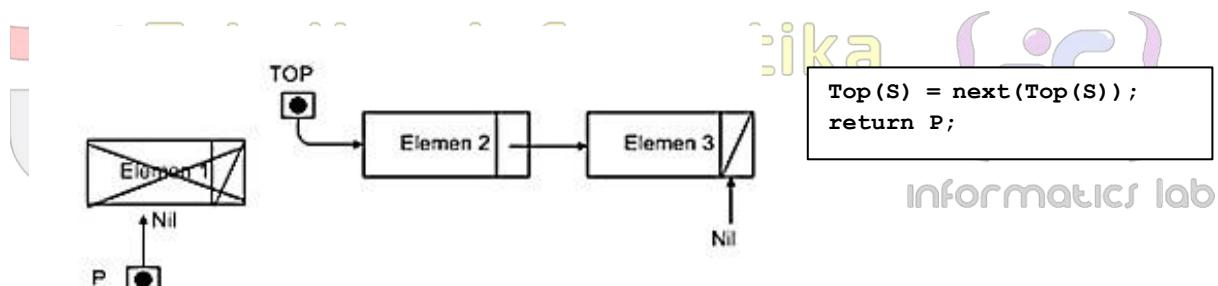
Adalah operasi pengambilan data dalam *list*. Operasi ini mirip dengan operasi *delete first* dalam *list linear*, karena elemen yang paling pertama kali diakses adalah elemen paling atas atau elemen paling awal saja. Langkah-langkah dalam proses *Pop*:



Gambar 7-5 Stack Pop 1



Gambar 7-6 Stack Pop 2



Gambar 7-7 Stack Pop 3

```
/* contoh sintak pop */
address pop(address p){
    p = top(s);
    top(s) = next(top(s));
    return p;
}
```

7.4 Primitif-Primitif dalam Stack

Primitif-primitif dalam *stack* pada dasarnya sama dengan primitif-primitif pada *list* lainnya. Malahan primitif dalam *stack* lebih sedikit, karena dalam *stack* hanya melakukan operasi-operasi terhadap elemen paling atas.

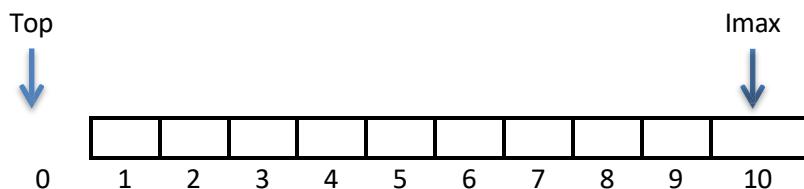
Primitif-primitif dalam *stack* :

1. `createStack()`.
2. `isEmpty()`.
3. `alokasi()`.
4. `de alokasi()`.
5. Fungsi-fungsi pencarian.
6. Dan fungsi-fungsi primitif lainnya.

Seperti halnya pada model *list* yang lain, primitif-primitifnya tersimpan pada file `*.c` dan file `*.h`.

7.5 Stack (Representasi Tabel)

Pada prinsipnya representasi menggunakan tabel sama halnya dengan menggunakan *pointer*. Perbedaannya terletak pada pendeklarasian struktur datanya, menggunakan *array* berindeks dan jumlah tumpukan yang terbatas.



Gambar 7-8 Stack Kosong dengan Representasi Table

Gambar di atas menunjukkan *stack* maksimum terdapat pada indeks $lmax=10$, sedangkan *Stack* masih kosong karena $Top = 0$.

7.5.1 Operasi-operasi Dalam Stack

Operasi-operasi dalam *stack* representasi tabel pada dasarnya sama dengan representasi *pointer*, yaitu **PUSH** dan **POP**.

A. Push

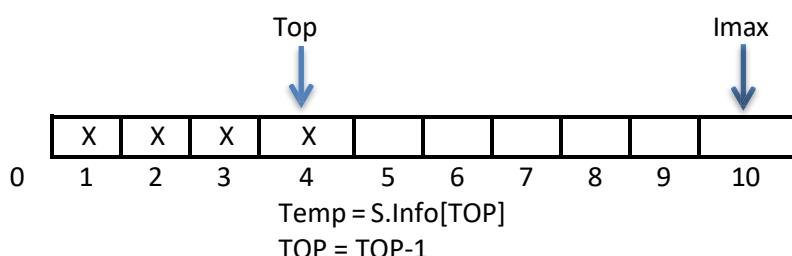
Push merupakan operasi penyisipan data ke dalam *stack*, penyisipan dilakukan dengan menggeser indeks dari **TOP** ke indeks berikutnya. Perhatikan contoh dibawah ini:

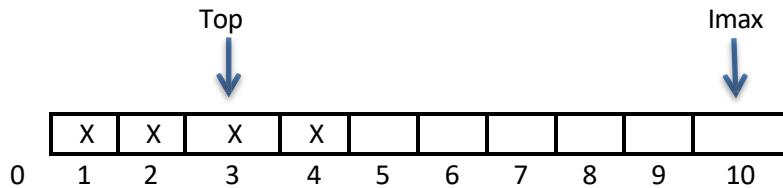


Gambar 7-9 Push Elemen dengan Representasi Tabel

B. Pop

Pop merupakan operasi pengambilan data di posisi indeks **TOP** berada dalam sebuah *stack*. Setelah data diambil, indeks **TOP** akan bergeser ke indeks sebelum **TOP** tanpa menghilangkan *info* dari indeks **TOP** sebelumnya. Perhatikan contoh dibawah ini:





Gambar 7-10 Pop Elemen dengan Representasi Tabel

7.5.2 Primitif-primitif Dalam Stack

Primitif-primitif pada *stack* representasi tabel pada dasarnya sama dengan representasi *pointer*. Perbedaanya hanya pada manajemen memori, pada representasi tabel tidak diperlukan manajemen memori. antara lain sebagai berikut,

1. `createStack()`.
2. `isEmpty()`.
3. Fungsi – fungsi pencarian.
4. Dan fungsi – fungsi primitif lainnya.

Seperti halnya pada model *list* yang lain, primitif-primitifnya tersimpan pada file *.c dan file *.h.

Untuk lebih memahami struktur data dari *stack* representasi tabel, berikut ini contoh ADT *stack* representasi tabel dalam file *.h:

```

1  /* file : stack.h
2   contoh ADT stack dengan representasi tabel */
3 #ifndef STACK_H_INCLUDED
4 #define STACK_H_INCLUDED
5
6 #include <stdio.h>
7 #include <conio.h>
8 struct infotype {
9     char nim[20];
10    char nama[20];
11 };
12 struct Stack {
13     infotype info[10];
14     int Top;
15 };
16
17 /* prototype */
18 //***** pengecekan apakah Stack kosong *****/
19 int isEmpty(Stack S);
20 /* mengembalikan nilai 0 jika stack kosong */
21 //***** pembuatan Stack *****/
22 void createStack(Stack &S);
23 /* I.S. sembarang
24     F.S. terbentuk stack dengan Top=0 */
25 //***** penambahan elemen pada Stack *****/
26 void push(Stack &S, infotype X);
27 /* I.S. Stack mungkin kosong
28     F.S. menambahkan elemen pada stack dengan nilai X, TOP=TOP+1 */
29 //***** penghapusan elemen pada list *****/
30 void pop(Stack &S, infotype &X);
31 /* I.S. stack tidak kosong
32     F.S. nilai info pada indeks TOP disimpan pada X, kemudian TOP=TOP-1 */
33 //***** proses semua elemen list *****/
34 void viewStack(Stack S);
35 /* I.S. stack mungkin kosong
36     F.S. jika stack tidak kosong menampilkan semua info yang ada pada stack
37 */
38#endif // STACK_H_INCLUDED

```

7.6 Latihan Stack

- Buatlah ADT Stack menggunakan ARRAY sebagai berikut di dalam file "stack.h":

```
Type infotype : integer
Type Stack <
    info : array [20] of integer
    top : integer
>
prosedur CreateStack( in/out S : Stack )
prosedur push( in/out S : Stack, in x : infotype)
fungsi pop( in/out S : Stack ) : infotype
prosedur printInfo( in S : Stack )
prosedur balikStack( in/out S : Stack )
```

Program 2 Stack.h

Buatlah implementasi ADT Stack menggunakan Array pada file "stack.cpp" dan "main.cpp"

```
int main()
{
    cout << "Hello world!" << endl;
    Stack S;
    createStack(S);
    Push(S,3);
    Push(S,4);
    Push(S,8);
    pop(S);
    Push(S,2);
    Push(S,3);
    pop(S);
    Push(S,9);
    printInfo(S);
    cout<<"balik stack"<<endl;
    balikStack(S);
    printInfo(S);
    return 0;
}
```

```
Hello world!
[TOP] 9 2 4 3
balik stack
[TOP] 3 4 2 9
```

Gambar 7-11 Output stack



Fakultas Informatika
School of Computing
Telkom University



Gambar 7-12 Main stack

- Tambahkan prosedur **pushAscending**(in/out S : Stack, in x : integer)

```
int main()
{
    cout << "Hello world!" << endl;
    Stack S;
    createStack(S);
    pushAscending(S,3);
    pushAscending(S,4);
    pushAscending(S,8);
    pushAscending(S,2);
    pushAscending(S,3);
    pushAscending(S,9);
    printInfo(S);
    cout<<"balik stack"<<endl;
    balikStack(S);
    printInfo(S);
    return 0;
}
```

```
Hello world!
[TOP] 9 8 4 3 3 2
balik stack
[TOP] 2 3 3 4 8 9
```

Gambar 7-13 Output stack push ascending

Gambar 7-14 Main stack dengan push ascending

3. Tambahkan prosedur `getInputStream(in/out S : Stack)`. Prosedur akan terus membaca dan menerima *input* user dan memasukkan setiap *input* ke dalam *stack* hingga user menekan tombol enter. Contoh: gunakan `cin.get()` untuk mendapatkan inputan user.

```
int main()
{
    cout << "Hello world!" << endl;
    Stack S;
    createStack(S);
    getInputStream(S);
    printInfo(S);
    cout<<"balik stack"<<endl;
    balikStack(S);
    printInfo(S);
    return 0;
}
```

Gambar 7-16 Main stack dengan *input* stream

```
Hello world!
4729601
[TOP] 1 0 6 9 2 7 4
balik stack
[TOP] 4 7 2 9 6 0 1
```

Gambar 7-15 Output stack

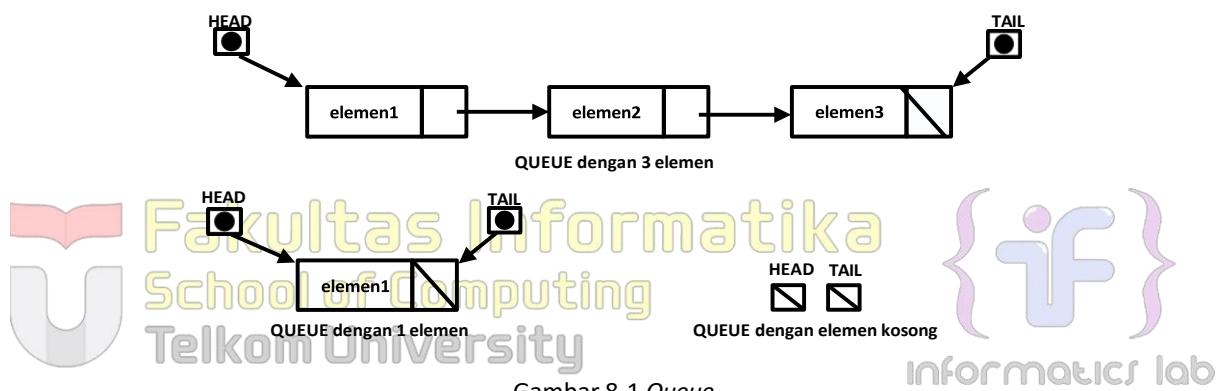


Modul 8 QUEUE

8.1 Pengertian Queue

Queue (dibaca : kyu) merupakan struktur data yang dapat diumpamakan seperti sebuah antrian. Misalkan antrian pada loket pembelian tiket Kereta Api. Orang yang akan mendapatkan pelayanan yang pertama adalah orang pertamakali masuk dalam antrian tersebut dan yang terakhir masuk dia akan mendapatkan layanan yang terakhir pula. Jadi prinsip dasar dalam *Queue* adalah **FIFO** (*First in First out*), proses yang pertama masuk akan diakses terlebih dahulu. Dalam pengimplementasian struktur *Queue* dalam C dapat menggunakan tipe data *array* dan *linked list*.

Dalam praktikum ini hanya akan dibahas pengimplementasian *Queue* dalam bentuk *linked list*. Implementasi *Queue* dalam *linked list* sebenarnya tidak jauh berbeda dengan operasi *list* biasa, malahan lebih sederhana. Karena sesuai dengan sifat FIFO dimana proses *delete* hanya dilakukan pada bagian **Head** (depan *list*) dan proses *insert* selalu dilakukan pada bagian **Tail** (belakang *list*) atau sebaliknya, tergantung dari persepsi masing-masing. Dalam penerapannya *Queue* dapat diterapkan dalam *single linked list* dan *double linked list*.



Contoh pendeklarasian struktur data *queue*:

```
1 #ifndef queue_H
2 #define queue_H
3 #define Nil NULL
4 #define info(P) (P)->info
5 #define next(P) (P)->next
6 #define head(Q) ((Q).head)
7 #define tail(Q) ((Q).tail)
8
9 typedef int infotype; /*tipe data dalam queue */
10 typedef struct elmQueue *address; /* tipe data pointer untuk elemen queue */
11 struct elmQueue{
12     infotype info;
13     address next;
14 }; /*tipe data elemen queue */
15 /* pendeklarasian tipe data queue */
16 struct queue {
17     address head, tail;
18 };
19 #endif
```

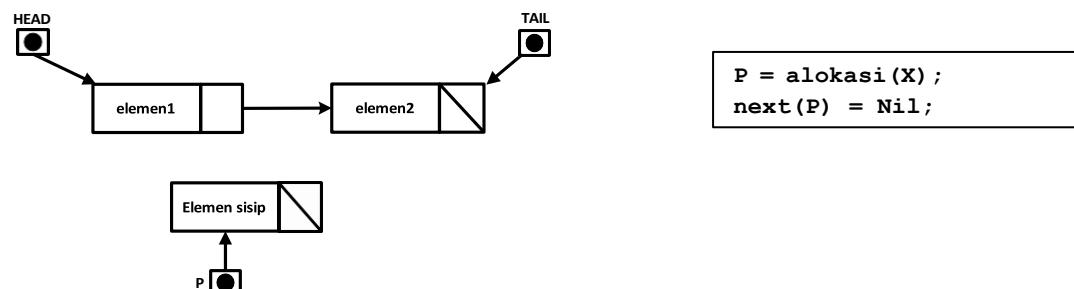
8.2 Operasi-Operasi dalam Queue

Dalam *queue* ada dua operasi utama, yaitu operasi penyisipan (*Insert/Enqueue*) dan operasi pengambilan (*Delete/Dequeue*).

8.2.1 Insert (Enqueue)

Operasi penyisipan selalu dilakukan pada akhir (*tail*).

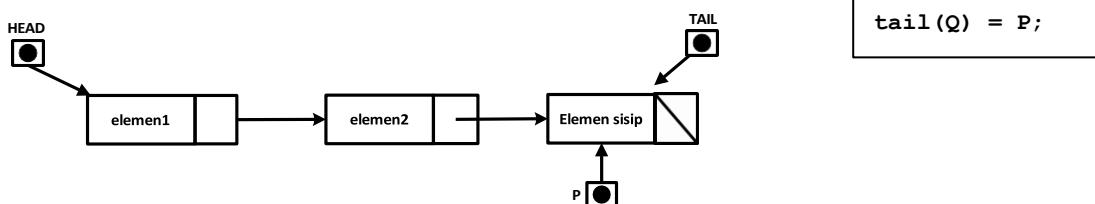
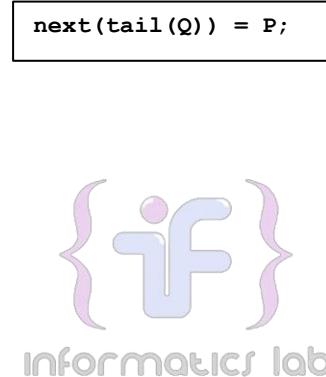
Langkah – langkah dalam proses Enqueue:



Gambar 8-2 Queue Insert 1



Gambar 8-3 Queue Insert 2

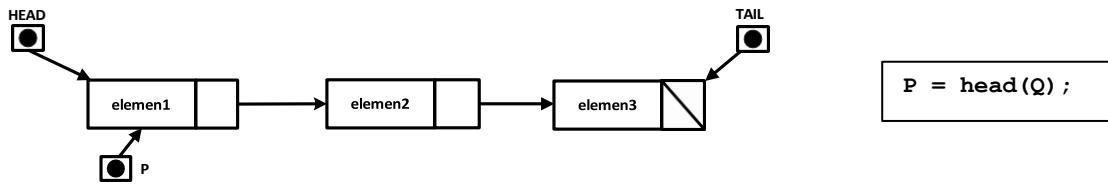


Gambar 8-4 Queue Insert 3

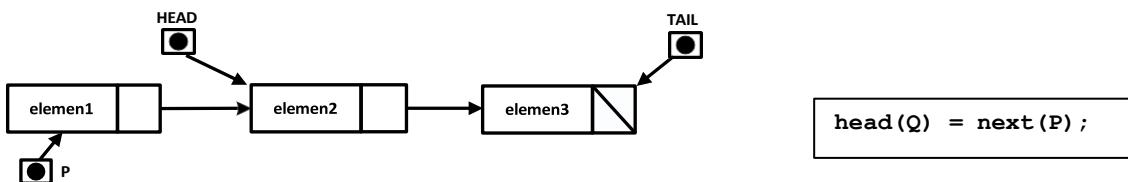
```
1 /* buat dahulu elemen yang akan disisipkan */  
2 address createElm(int x){  
3     address p = alokasi(x);  
4     next(p) = null;  
5     return p;  
6 }  
7  
8 /* contoh sintak queue insert */  
9 void queue(address p){  
10    next(tail(Q)) = p;  
11    tail(Q) = p;  
12 }
```

8.2.2 Delete (Dequeue)

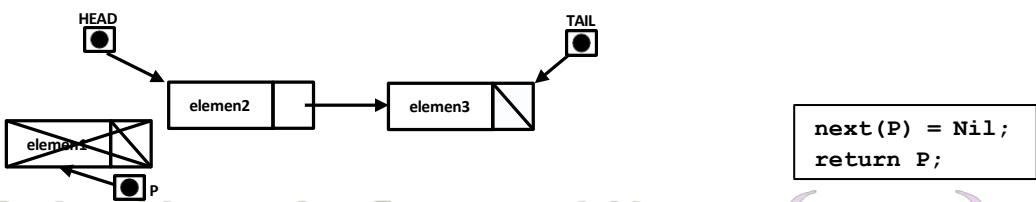
Operasi *delete* dilakukan pada awal (*head*).



Gambar 8-5 Queue Delete 1



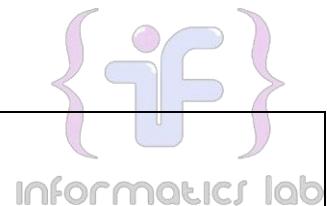
Gambar 8-6 Queue Delete 2



Gambar 8-7 Queue Delete 3

```

1 /*contoh sintak dequeue */
2 address dequeue(address p) {
3     p = head(Q);
4     head(Q) = next(p);
5     next(p) = null;
6     return p;
7 }
```



8.3 Primitif-Primitif dalam Queue

Primitif-primitif pada *queue* tersimpan pada ADT *queue*, seperti pada materi sebelumnya, primitif-primitifnya tersimpan pada file *.h dan *.c.

File *.h untuk ADT *queue*:

```

1 /*file : queue .h
2 contoh ADT queue dengan representasi fisik pointer
3 representasi address dengan pointer
4 info tipe adalah integer */

5
6 #ifndef QUEUE_H_INCLUDE
7 #define QUEUE_H_INCLUDE
8 #include <stdio.h>
9
10#define Nil NULL
11#define info(P) ((P)->info)
12#define next(P) ((P)->next)
13#define head(S) ((S).head)
14#define tail(S) ((S).tail)
15typedef int infotype;
16typedef struct elmQ *address;
```

```

18 struct elmQ{
19     infotype info;
20     address next;
21 };
22 /* deklarasi tipe data queue, terdiri dari head dan tail, queue kosong jika
23     head = Nil */
24 struct queue {
25     address head, tail;
26 };
27
28 /*prototype*/
29 /***** pengecekan apakah queue kosong *****/
30 boolean isEmpty(queue Q);
31 /*mengembalikan nilai true jika queue kosong*/
32
33 /***** pembuatan queue kosong *****/
34 void CreateQueue(queue &Q);
35 /* I.S. sembarang
36     F.S. terbentuk queue kosong*/
37
38
39 /***** manajemen memori *****/
40 void alokasi(infotype X);
41 /* mengirimkan address dari alokasi elemen
42     jika alokasi berhasil, maka nilai address tidak Nil dan jika gagal, nilai
43     address Nil */
44
45 void dealokasi(address P);
46 /* I.S. P terdefinisi
47     F.S. memori yang digunakan P dikembalikan ke sistem */
48
49 /***** pencarian sebuah elemen list *****/
50 address findElm(queue Q, infotype X);
51 /* mencari apakah ada elemen queue dengan info(P) = X
52     jika ada, mengembalikan address elemen tab tsb, dan Nil jika sebaliknya */
53
54 boolean fFindElm(queue Q address P);
55 /* mencari apakah ada elemen queue dengan alamat P
56     mengembalikan true jika ada dan false jika tidak ada */
57
58 /***** penambahan elemen *****/
59 void insert(queue &Q, address P);
60 /* I.S. sembarang, P sudah dialokasikan
61     F.S. menempatkan elemen beralamat P pada akhir queue*/
62
63 /***** penghapusan sebuah elemen *****/
64 void delete(queue &Q, address &P);
65 /* I.S. queue tidak kosong
66     F.S. menunjuk elemen pertama queue, head dari queue menunjuk pada next
67     elemen head yang lama. Queue mungkin menjadi kosong */
68
69 /***** proses semua elemen queue*****/
70 void printInfo(queue Q);
71 /* I.S. queue mungkin tidak kosong
72     F.S. jika queue tidak kosong, maka menampilkan semua info yang ada pada
73     queue */
74
75 void nbList(queue Q);
76 /* mengembalikan jumlah elemen pada queue */
77
78 void delAll(queue &Q);
79 /* menghapus semua elemen pada queue dan semua elemen di-dealokasi */
80
81 #endif

```

8.4 Queue (Representasi Tabel)

8.4.1 Pengertian

Pada dasarnya representasi *queue* menggunakan tabel sama halnya dengan menggunakan *pointer*. Perbedaan yang mendasar adalah pada management memori serta keterbatasan jumlah antriannya. Untuk lebih jelasnya perhatikan perbedaan representasi tabel dan *pointer* pada *queue* dibawah ini.

Tabel 8-1 Perbedaan Queue Representasi Table dan Pointer

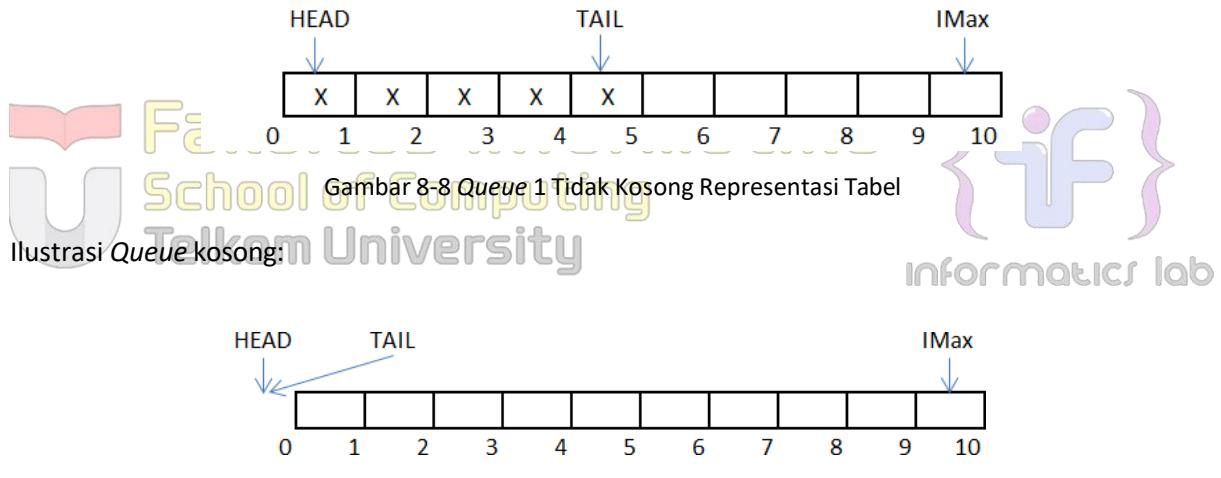
No	Representasi Table	Representasi Pointer
1	Jumlah Queue terbatas	Jumlah Queue tak-terbatas
2	Tidak ada management memori	ada management memori

Ada beberapa macam-macam bentuk *queue*, yaitu:

A. Alternatif 1

Tabel dengan hanya representasi *TAIL* adalah indeks elemen terakhir, *HEAD* selalu di-set sama dengan 1 jika *Queue* tidak kosong. Jika *Queue* kosong, maka *HEAD*=0.

Ilustrasi *Queue* tidak kosong, dengan 5 elemen :

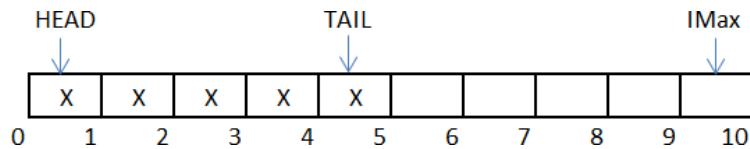


Gambar 8-9 Queue 1 Kosong Representasi Tabel

Algoritma paling sederhana untuk penambahan elemen jika masih ada tempat adalah dengan “memajukan” *TAIL*. Kasus khusus untuk *Queue* kosong karena *HEAD* harus diset nilainya menjadi 1. Algoritma paling sederhana dan “naif” untuk penghapusan elemen jika *Queue* tidak kosong: ambil nilai elemen *HEAD*, geser semua elemen mulai dari *HEAD*+1 s/d *TAIL* (jika ada), kemudian *TAIL* “mundur”. Kasus khusus untuk *Queue* dengan keadaan awal berelemen 1, yaitu menyesuaikan *HEAD* dan *TAIL* dengan DEFINISI. Algoritma ini mencerminkan pergeseran orang yang sedang mengantri di dunia nyata, tapi tidak efisien.

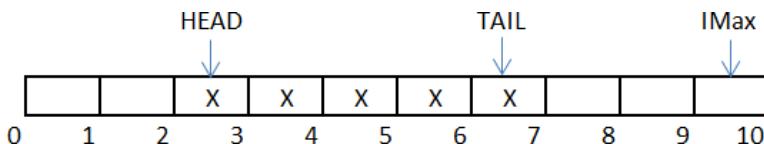
B. Alternatif 2

Tabel dengan representasi *HEAD* dan *TAIL*, *HEAD* “bergerak” ketika sebuah elemen dihapus jika *Queue* tidak kosong. Jika *Queue* kosong, maka *HEAD*=0. Ilustrasi *Queue* tidak kosong, dengan 5 elemen, kemungkinan pertama *HEAD* sedang berada di posisi awal:



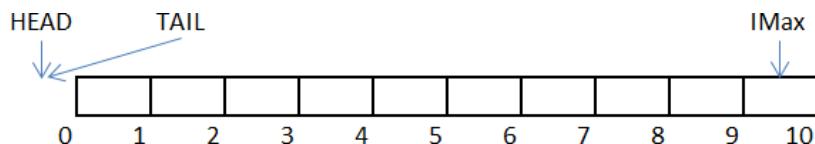
Gambar 8-10 Queue 2 Tidak Kosong 1 Representasi Table

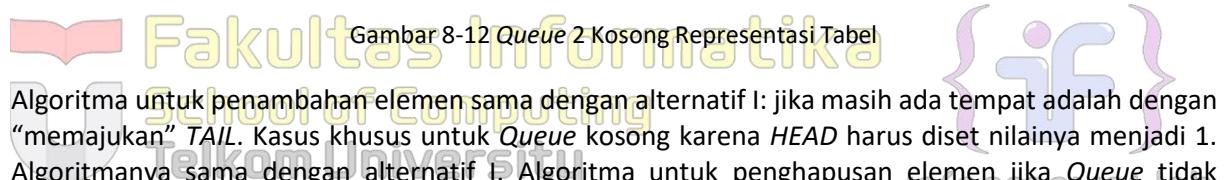
Ilustrasi *Queue* tidak kosong, dengan 5 elemen, kemungkinan pertama *HEAD* tidak berada di posisi awal. Hal ini terjadi akibat algoritma penghapusan yang dilakukan (lihat keterangan).



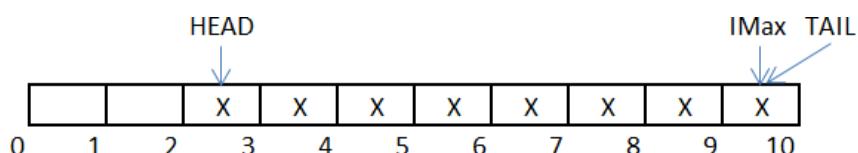
Gambar 8-11 Queue 2 Tidak Kosong 2 Representasi Tabel

Ilustrasi *Queue* kosong:




Algoritma untuk penambahan elemen sama dengan alternatif I: jika masih ada tempat adalah dengan “memajukan” *TAIL*. Kasus khusus untuk *Queue* kosong karena *HEAD* harus diset nilainya menjadi 1. Algoritmanya sama dengan alternatif I. Algoritma untuk penghapusan elemen jika *Queue* tidak kosong: ambil nilai elemen *HEAD*, kemudian *HEAD* “maju”. Kasus khusus untuk *Queue* dengan keadaan awal berelemen 1, yaitu menyesuaikan *HEAD* dan *TAIL* dengan DEFINISI.

Algoritma ini TIDAK mencerminkan pergeseran orang yang sedang mengantri di dunia nyata, tapi efisien. Namun suatu saat terjadi keadaan *Queue* penuh tetapi “semu” sebagai berikut :

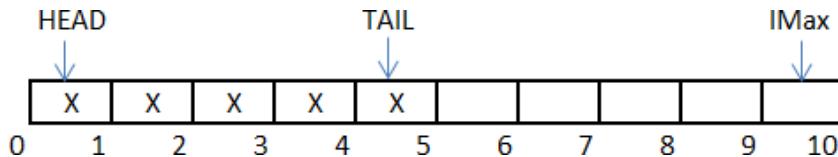


Gambar 8-13 Queue 2 Penuh “semu”

Jika keadaan ini terjadi, haruslah dilakukan aksi menggeser elemen untuk menciptakan ruangan kosong. Pergeseran hanya dilakukan jika dan hanya jika *TAIL* sudah mencapai *IndexMax*.

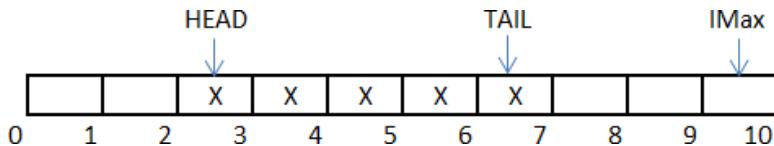
C. Alternatif 3

Tabel dengan representasi *HEAD* dan *TAIL* yang “berputar” mengelilingi indeks tabel dari awal sampai akhir, kemudian kembali ke awal. Jika *Queue* kosong, maka *HEAD*=0. Representasi ini memungkinkan tidak perlu lagi ada pergeseran yang harus dilakukan seperti pada alternatif II pada saat penambahan elemen. Ilustrasi *Queue* tidak kosong, dengan 5 elemen, dengan *HEAD* “sedang” berada di posisi awal:



Gambar 8-14 Queue 3 Tidak Kosong 1 Representasi Table

Ilustrasi *Queue* tidak kosong, dengan 5 elemen, dengan *HEAD* tidak berada diposisi awal, tetapi masih “lebih kecil” atau “sebelum” *TAIL*. Hal ini terjadi akibat algoritma penghapusan/Penambahan yang dilakukan (lihat keterangan).



Gambar 8-15 Queue 3 Tidak Kosong 2 Representasi Table

Ilustrasi *Queue* tidak kosong, dengan 5 elemen, *HEAD* tidak berada di posisi awal, tetapi “lebih besar” atau “sesudah” *TAIL*. Hal ini terjadi akibat algoritma penghapusan/Penambahan yang dilakukan (lihat keterangan)



Gambar 8-16 Queue 3 Tidak Kosong 3 Representasi Table

Algoritma untuk penambahan elemen: jika masih ada tempat adalah dengan “memajukan” *TAIL*. Tapi jika *TAIL* sudah mencapai *IdxMax*, maka *successor* dari *IdxMax* adalah 1 sehingga *TAIL* yang baru adalah 1. Jika *TAIL* belum mencapai *IdxMax*, maka algoritma penambahan elemen sama dengan alternatif II. Kasus khusus untuk *Queue* kosong karena *HEAD* harus diset nilainya menjadi 1.

Algoritma untuk penghapusan elemen jika *Queue* tidak kosong: ambil nilai elemen *HEAD*, kemudian *HEAD* “maju”. Penentuan suatu *successor* dari indeks yang diubah/“maju” dibuat Seperti pada algoritma penambahan elemen: jika *HEAD* mencapai *IdxMAX*, maka *successor* dari *HEAD* adalah 1. Kasus khusus untuk *Queue* dengan keadaan awal berelemen 1, yaitu menyesuaikan *HEAD* dan *TAIL* dengan DEFINISI. Algoritma ini efisien karena tidak perlu pergeseran, dan seringkali strategi pemakaian tabel semacam ini disebut sebagai “circular buffer”, dimana tabel penyimpan elemen dianggap sebagai “buffer”.

Salah satu variasi dari representasi pada alternatif III adalah : menggantikan representasi *TAIL* dengan COUNT, yaitu banyaknya elemen *Queue*. Dengan representasi ini, banyaknya elemen diketahui secara eksplisit, tetapi untuk melakukan penambahan elemen harus dilakukan kalkulasi *TAIL*. Buatlah sebagai latihan.

Contoh Kasus Alternatif 2

Antrian kosong (*Head=0, Tail=0*)



Setelah penambahan elemen 8,3,6,9,15,1 (*Head=1, Tail=6*)

8	3	6	9	15	1				
1	2	3	4	5	6	7	8	9	10

Setelah penghapusan sebuah elemen (*Head=2, Tail=6*)

	3	6	9	15	1				
1	2	3	4	5	6	7	8	9	10

Algoritma	C++
<u>Kamus Umum</u> constant NMax: integer = 100 constant Nil: integer = 0 type infotype = integer type Queue = <TabQ: array[1..NMax] of infotype; Head,Tail: integer Function EmptyQ(Q:Queue) → boolean {True jika Q antrian kosong} Procedure CreateEmpty(output Q:Queue) {I.Q. - F.Q. Terdefinisi antrian kosong Q}	<pre>const int NMax=100; const int Nil=0; typedef int infotype queue { infotype TabQ[NMax]; int head, tail; } bool Empty(Queue Q){ /* True jika Q merupakan antrian kosong */ } void CreateEmpty(){ /* membuat antrian kosong */ } bool IsFull(Queue Q){ /* True jika Q penuh */ if(head(Q)=1 && tail(Q)=Nmax){ return true; } }</pre>
function IsFullQ(Q:Queue) → boolean {True Q penuh} kamus: algoritma → (Q.Head = 1) and (Q.Tail=NMax)	void addQ(Queue Q, infotype x){ int i; if (isFull(Q)){ cout >> "Antrian Penuh"; }else{ if(Empty(Q)){ head(Q) = 1; tail(Q) = 1; }else{ if(tail(Q)>NMax){ tail(Q)++; }else{ for(i=head(Q);i<=tail(Q);i++){ TabQ[i+head(Q)+1] = TabQ[i]; } tail(Q) = tail(Q)-head(Q)+2; head(Q) = 1; TabQ[tail(Q)](Q) = x; /*menyisipkan x */ } } } }
Procedure AddQ(input/output Q:Queue; input X:infotype) {I.Q. Q antrian, terdefinisi, mungkin kosong} {F.Q. X elemen terakhir Q jika antrian tidak penuh} Kamus i:integer algoritma if IsFull(Q) then output('Antrian penuh') else if Empty(Q) then Q.Head ← 1 Q.Tail ← 1 else if Q.Tail<NMax then Q.Tail ← Q.Tail + 1 else for i←Q.Head to Q.Tail do Q.TabQ[i-Q.Head+1] ← Q.TabQ[i] Q.Tail ← Q.Tail-Q.Head+2 Q.Head ← 1 Q.TabQ[Q.Tail] ← X {menyisipkan X}	

```

Procedure      DelQ(input/output    Q:Queue;      output
X:infotype)
{I.Q. Q antrian, terdefinisi, tidak kosong}
{F.Q. X elemen antrian yang dihapus}
Kamus
  i:integer
Algoritma
  X ← Q.TabQ[Q.Head]
  if Q.Head=Q.Tail then
    CreateEmpty(Q)
  else
    Q.Head ← Q.Head+1

```

```

void DelQ(Queue Q, infotype x){
  int i;
  x= TabQ[head(Q)](Q);
  if(head(Q)==tail(Q)){
    CreateEmpty(Q);
  }else{
    head(Q);
  }
}

```

8.4.2 Primitif-Primitif dalam Queue

Berikut ini contoh program *queue.h* dalam ADT *queue* menggunakan representasi tabel.

```

1  /*file : queue .h
2   contoh ADT queue dengan representasi tabel*/
3
4  #ifndef QUEUE_H_INCLUDE
5  #define QUEUE_H_INCLUDE
6  #include <stdio.h>
7  #include <conio.h>
8
9  struct infotype {
10    char id[20];
11    char nama[20];
12  };
13
14 struct Queue {
15   infotype info[3];
16   int head;
17   int tail;
18 };
19
20 /* prototype */
21 //***** pengecekan apakah Queue penuh *****/
22 int isFull(queue Q):
23 /* mengembalikan nilai 0 jika queue penuh */
24
25 //***** pengecekan apakah Queue kosong *****/
26 int isEmpty(queue Q);
27 /* mengembalikan nilai 0 jika queue kosong */
28
29 //***** pembuatan queue *****/
30 void CreateQueue(queue &Q);
31 /* I.S. sembarang
32   F.S. terbentuk queue dengan head = -1 dan tail = -1 */
33
34 //***** penambahan elemen pada queue *****/
35 void enQueue(queue &Q, infotype X);
36 /* I.S. queue mungkin kosong
37   F.S. menambahkan elemen pada stack dengan nilai X */
38
39 //***** penghapusan elemen pada queue *****/
40 void deQueue(queue &Q);
41 /* I.S. queue tidak kosong
42   F.S. head = head + 1 */
43
44 //***** proses semua elemen pada queue *****/
45 void viewQueue(queue Q);
46 /* I.S. queue mungkin kosong
47   F.S. jika queue tidak kosong menampilkan semua info yang ada pada queue */
48
49 #endif

```

8.5 Latihan Queue

- Buatlah ADT *Queue* menggunakan *ARRAY* sebagai berikut di dalam file “*queue.h*”:

```
Type infotype: integer
Type Queue: <
    info : array [5] of infotype {index array dalam C++
        dimulai dari 0}
        head, tail : integer
    >
prosedur CreateQueue (in/out Q: Queue)
fungsi isEmptyQueue (Q: Queue) → boolean
fungsi isFullQueue (Q: Queue) → boolean
prosedur enqueue (in/out Q: Queue, in x: infotype)
fungsi dequeue (in/out Q: Queue) → infotype
prosedur printInfo (in Q: Queue)
```

Buatlah implementasi ADT *Queue* pada file “*queue.cpp*” dengan menerapkan mekanisme *queue* Alternatif 1 (*head* diam, *tail* bergerak).

```
1 int main() {
2     cout << "Hello World" << endl;
3     Queue Q;
4     createQueue (Q);
5
6     cout<<" -----" << endl;
7     cout<<" H - T \t | Queue Info" << endl;
8     cout<<" -----" << endl;
9     printInfo (Q);
10    enqueue (Q,5); printInfo (Q);
11    enqueue (Q,2); printInfo (Q);
12    enqueue (Q,7); printInfo (Q);
13    dequeue (Q); printInfo (Q);
14    enqueue (Q,4); printInfo (Q);
15    dequeue (Q); printInfo (Q);
16    dequeue (Q); printInfo (Q);
17
18    return 0;
19 }
```

Hello world!		
H	-	T
-1	-	-1
0	-	0
0	-	1
0	-	2
0	-	2
0	-	7
0	-	2
0	-	7
0	-	4
0	-	4
-1	-	-1
empty queue		

Gambar 8-17 Output Queue

informatics lab

Gambar 8-18 Main Queue

- Buatlah implementasi ADT *Queue* pada file “*queue.cpp*” dengan menerapkan mekanisme *queue* Alternatif 2 (*head* bergerak, *tail* bergerak).
- Buatlah implementasi ADT *Queue* pada file “*queue.cpp*” dengan menerapkan mekanisme *queue* Alternatif 3 (*head* dan *tail* berputar).

Modul 9 ASSESSMENT CLO 1 (SLL & DLL)

TUJUAN PRAKTIKUM

1. Mengevaluasi pemahaman materi *Single Linked List* dan *Double Linked List*.

Pada modul 9, praktikan akan dievaluasi pemahamannya terhadap materi *Single Linked List* dan *Double Linked List*. Praktikan diminta untuk menyelesaikan soal-soal yang diberikan dalam bentuk *Assessment*.



Modul 10 TREE (BAGIAN PERTAMA)

TUJUAN PRAKTIKUM

1. Memahami konsep penggunaan fungsi rekursif.
2. Mengimplementasikan bentuk-bentuk fungsi rekursif.
3. Mengaplikasikan struktur data *tree* dalam sebuah kasus pemrograman.
4. Mengimplementasikan struktur data *tree*, khususnya *Binary Tree*.

10.1 Pengertian Rekursif

Secara harfiah, rekursif berarti suatu proses pengulangan sesuatu dengan cara kesamaan-diri atau suatu proses yang memanggil dirinya sendiri. Prosedur dan fungsi merupakan sub program yang sangat bermanfaat dalam pemrograman, terutama untuk program atau proyek yang besar.

Manfaat penggunaan sub program antara lain adalah :

1. meningkatkan *readibility*, yaitu mempermudah pembacaan program
2. meningkatkan *modularity*, yaitu memecah sesuatu yang besar menjadi modul-modul atau bagian-bagian yang lebih kecil sesuai dengan fungsinya, sehingga mempermudah pengecekan, *testing* dan lokalisasi kesalahan.
3. meningkatkan *reusability*, yaitu suatu sub program dapat dipakai berulang kali dengan hanya memanggil sub program tersebut tanpa menuliskan perintah-perintah yang semestinya diulang-ulang.

Sub Program Rekursif adalah sub program yang memanggil dirinya sendiri selama kondisi pemanggilan dipenuhi. Prinsip rekursif sangat berkaitan erat dengan bentuk induksi matematika. Berikut adalah contoh fungsi rekursif pada rumus pangkat 2:

Kita ketahui bahwa secara umum perhitungan pangkat 2 dapat dituliskan sebagai berikut

$$\begin{aligned} 2^0 &= 1 \\ 2^n &= 2 * 2^{n-1} \end{aligned}$$

Secara matematis, rumus pangkat 2 dapat dituliskan sebagai

$$f(x) = \begin{cases} 1 & |x| = 0 \\ 2 * f(x-1) & |x| > 0 \end{cases}$$

Berdasarkan rumus matematika tersebut, kita dapat bangun algoritma rekursif untuk menghitung hasil pangkat 2 sebagai berikut :

```
Fungsi pangkat_2 ( x : integer ) : integer
Kamus
Algoritma
  If( x = 0 ) then
    → 1
  Else
    → 2 * pangkat_2( x - 1 )
```

Jika kita jalankan algoritma di atas dengan $x = 4$, maka algoritma di atas akan menghasilkan

Pangkat_2 (4)
→ 2 * pangkat_2 (3)
→ 2 * (2 * pangkat_2 (2))
→ 2 * (2 * (2 * pangkat_2 (1)))
→ 2 * (2 * (2 * (2 * pangkat_2 (0))))
→ 2 * (2 * (2 * (2 * 1)))
→ 2 * (2 * (2 * 2))
→ 2 * (2 * 4)
→ 2 * 8 → 16

10.2 Kriteria Rekursif

Dengan melihat sifat sub program rekursif di atas maka sub program rekursif harus memiliki :

1. Kondisi yang menyebabkan pemanggilan dirinya berhenti (disebut kondisi khusus atau special condition)
2. Pemanggilan diri sub program (yaitu bila kondisi khusus tidak dipenuhi)

Secara umum bentuk dari sub program rekursif memiliki statemen kondisional :

- if kondisi khusus tak dipenuhi
- then panggil diri-sendiri dengan parameter yang sesuai
- else lakukan instruksi yang akan dieksekusi bila kondisi khusus dipenuhi

Sub program rekursif umumnya dipakai untuk permasalahan yang memiliki langkah penyelesaian yang terpola atau langkah-langkah yang teratur. Bila kita memiliki suatu permasalahan dan kita mengetahui algoritma penyelesaiannya, kadang-kadang sub program rekursif menjadi pilihan kita bila memang memungkinkan untuk dipergunakan. Secara algoritmik (dari segi algoritma, yaitu bila kita mempertimbangkan penggunaan memori, waktu eksekusi sub program) sub program rekursif sering bersifat tidak efisien.

Dengan demikian sub program rekursif umumnya memiliki efisiensi dalam penulisan perintah, tetapi kadang tidak efisien secara algoritmik. Meskipun demikian banyak pula permasalahan-permasalahan yang lebih sesuai diselesaikan dengan cara rekursif (misalnya dalam pencarian / *searching*, yang akan dibahas pada pertemuan-pertemuan yang akan datang).

10.3 Kekurangan Rekursif

Konsep penggunaan yang terlihat mudah karena fungsi rekursif dapat menyederhanakan solusi dari suatu permasalahan, sehingga sering kali menghasilkan bentuk algoritma dan program yang lebih singkat dan lebih mudah dimengerti.

Kendati demikian, penggunaan rekursif memiliki beberapa kekurangan antara lain:

1. Memerlukan memori yang lebih banyak untuk menyimpan *activation record* dan variabel lokal. *Activation record* diperlukan waktu proses kembali kepada pemanggil
2. Memerlukan waktu yang lebih banyak untuk menangani *activation record*.

Secara umum gunakan penyelesaian rekursif hanya jika:

- Penyelesaian sulit dilaksanakan secara iteratif.
- Efisiensi dengan cara rekursif sudah memadai.
- Efisiensi bukan masalah dibandingkan dengan kejelasan logika program.

10.4 Contoh Rekursif

Rekursif berarti suatu fungsi dapat memanggil fungsi yang merupakan dirinya sendiri.

Berikut adalah contoh program untuk rekursif menghitung nilai pangkat sebuah bilangan.

Algoritma	C++
Program coba_rekursif Kamus bil, bil_pkt : integer function pangkat (input: x,y: integer) Algoritma	#include <conio.h> #include <iostream> #include <stdlib.h> using namespace std; /* prototype fungsi rekursif */ int pangkat(int x, int y); /* fungsi utama */ int main(){ system("cls");

<pre> input(bil, bil_pkt) output(pangkat(bil, bil_pkt)) function pangkat (input: x,y: integer) kamus algoritma if (y = 1) then → x else → x * pangkat(x,y-1) </pre>	<pre> int bil, bil_pkt; cout<<"menghitung x^y \n"; cout<<"x="; cin>>bil; cout<<"y="; cin>>bil_pkt; /* pemanggilan fungsi rekursif */ cout<<"\n "<< bil<<"^"<<bil_pkt << "="<<pangkat(bil,bil_pkt); getche(); return 0; } /* badan fungsi rekursif */ int pangkat(int x, int y){ if (y==1) return(x); else /* bentuk penulisan rekursif */ return(x*pangkat(x,y-1)); } </pre>
--	---

Berikut adalah contoh program untuk rekursif menghitung nilai faktorial sebuah bilangan.

Algoritma	C++
<pre> Program rekursif_factorial Kamus faktor, n : integer function faktorial (input: a: integer) Algoritma input(n) faktor =faktorial(n) output(faktor) function faktorial (input: a: integer) kamus algoritma if (a == 1 a == 0) then → 1 else if (a > 1) then → a* faktorial(a-1) else → 0 </pre>	 <pre> #include <conio.h> #include <iostream> long int faktorial(long int a); main(){ long int faktor; long int n; cout<<"Masukkan nilai faktorial "; cin>>n; faktor =faktorial(n); cout<<n<<"!="<<faktor<<endl; getch(); } long int faktorial(long int y){ if (a==1 a==0){ return(1); }else if (a>1){ return(a*faktorial(a-1)); }else{ return 0; } } </pre>

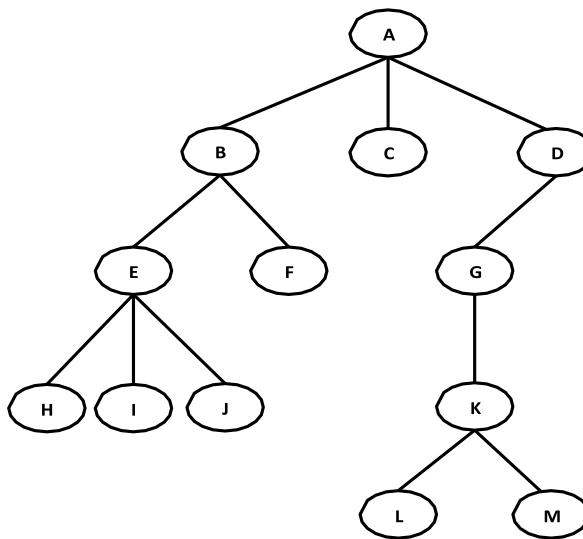
10.5 Pengertian Tree

Kita telah mengenal dan mempelajari jenis-jenis strukur data yang *linear*, seperti : *list*, *stack* dan *queue*. Adapun jenis struktur data yang kita pelajari kali ini adalah struktur data yang non-liniar (*non-linear data structure*) yang disebut *tree*.

Tree digambarkan sebagai suatu *graph* tak berarah terhubung dan tidak terhubung dan tidak mengandung sirkuit.

Karateristik dari suatu *tree* T adalah :

1. T kosong berarti *empty tree*
2. Hanya terdapat satu *node* tanpa pendahulu, disebut akar (*root*)
3. Semua *node* lainnya hanya mempunyai satu *node* pendahulu.



Gambar 10-1 Tree

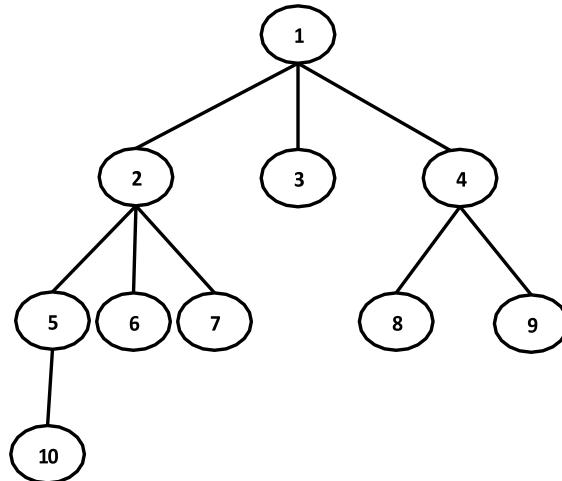
Berdasarkan gambar diatas dapat digambarkan beberapa terminologinya, yaitu

1. Anak (*child* atau *children*) dan Orangtua (*parent*). B, C, dan D adalah anak-anak simpul A, A adalah orangtua dari anak-anak itu.
2. Lintasan (*path*). Lintasan dari A ke J adalah A, B, E, J. Panjang lintasan dari A ke J adalah 3.
3. Saudara kandung (*sibling*). F adalah saudara kandung E, tetapi G bukan saudara kandung E, karena orangtua mereka berbeda.
4. Derajat(*degree*). Derajat sebuah simpul adalah jumlah pohon (atau jumlah anak) pada simpul tersebut. Derajat A = 3, derajat D = 1 dan derajat C = 0. Derajat maksimum dari semua simpul merupakan derajat pohon itu sendiri. Pohon diatas berderajat 3.
5. Daun (*leaf*). Simpul yang berderajat nol (atau tidak mempunyai anak) disebut daun. Simpul H, I, J, F, C, L, dan M adalah daun.
6. Simpul Dalam (*internal nodes*). Simpul yang mempunyai anak disebut simpul dalam. Simpul B, D, E, G, dan K adalah simpul dalam.
7. Tinggi (*height*) atau Kedalaman (*depth*). Jumlah maksimum *node* yang terdapat di cabang *tree* tersebut. Pohon diatas mempunyai tinggi 4.

10.6 Jenis-Jenis Tree

10.6.1 Ordered Tree

Yaitu pohon yang urutan anak-anaknya penting.



Gambar 10-2 Ordered Tree

10.6.2 Binary Tree

Setiap *node* di *Binary Tree* hanya dapat mempunyai maksimum 2 *children* tanpa pengecualian. *Level* dari suatu *tree* dapat menunjukkan berapa kemungkinan jumlah *maximum nodes* yang terdapat pada *tree* tersebut. Misalnya, *level tree* adalah r , maka *node* maksimum yang mungkin adalah 2^r .

A. Complete Binary Tree

Suatu *binary tree* dapat dikatakan lengkap (*complete*), jika pada setiap level yang mempunyai jumlah maksimum dari kemungkinan *node* yang dapat dipunyai, dengan pengecualian *node* terakhir. *Complete tree* T_n yang unik memiliki n *nodes*. Untuk menentukan jumlah *left children* dan *right children* *tree* T_n di *node K* dapat dilakukan dengan cara:

1. Menentukan *left children*: 2^*K
2. Menentukan *right children*: $2^*(K + 1)$
3. Menentukan *parent*: $[K/2]$

B. Extended Binary Tree

Suatu *binary tree* yang terdiri atas *tree T* yang masing-masing *node*-nya terdiri dari tepat 0 atau 2 *children* disebut *2-tree* atau **extended binary tree**. Jika setiap *node N* mempunyai 0 atau 2 *children* disebut *internal nodes* dan *node* dengan 0 *children* disebut *external nodes*.

C. Binary Search Tree

Binary search tree adalah *Binary tree* yang terurut dengan ketentuan:

1. Semua **LEFTCHILD** harus lebih kecil dari *parent*-nya.
2. Semua **RIGHTCHILD** harus lebih besar dari *parent*nya dan *leftchild*-nya.

D. AVL Tree

Adalah *binary search tree* yang mempunyai ketentuan bahwa *maximum* perbedaan *height* antara *subtree* kiri dan *subtree* kanan adalah 1.

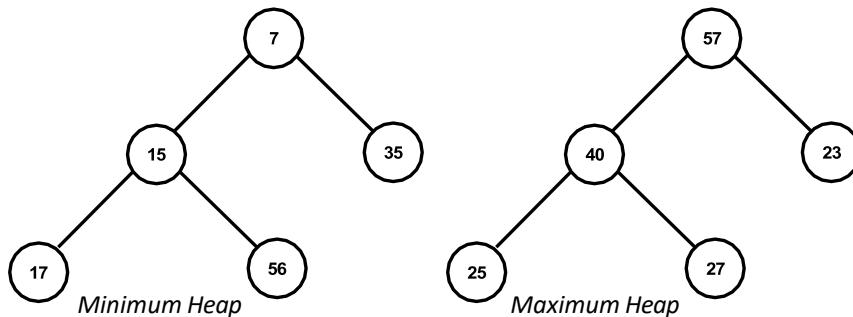
E. Heap Tree

Adalah *tree* yang memenuhi persamaan berikut: $R[i] < r[2i]$ and $R[i] < r[2i+1]$

Heap juga disebut *Complete Binary Tree*, karena jika suatu *node* mempunyai *child*, maka jumlah *child*-nya harus selalu dua.

Minimum Heap: jika *parent*-nya selalu lebih kecil daripada kedua *children*-nya.

Maximum Heap: jika *parent*-nya selalu lebih besar daripada kedua *children*-nya.



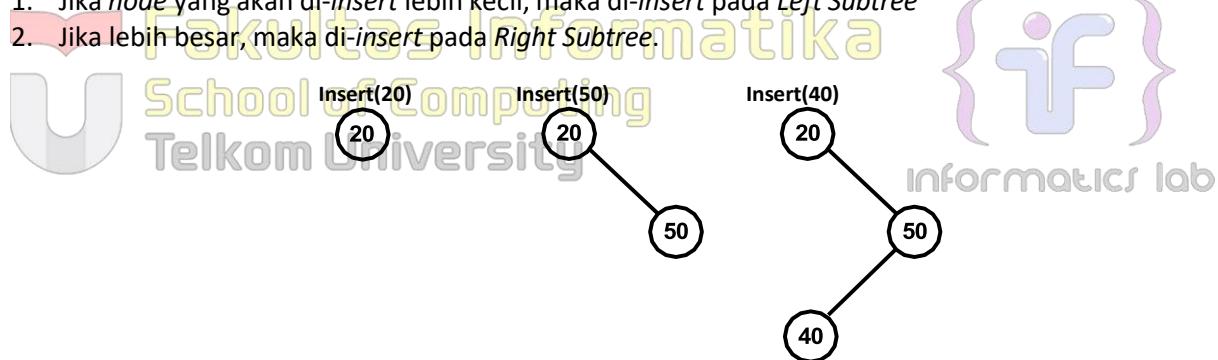
Gambar 10-3 Heap Tree

10.7 Operasi-Operasi dalam Binary Search Tree

Pada praktikum ini, difokuskan pada Pendalaman tentang *Binary Search Tree*.

A. Insert

1. Jika *node* yang akan di-insert lebih kecil, maka di-insert pada *Left Subtree*
2. Jika lebih besar, maka di-insert pada *Right Subtree*.



Gambar 10-4 Binary Search Tree Insert

```
1 struct node{  
2     int key;  
3     struct node *left, *right;  
4 };  
5  
6 // sebuah fungsi utilitas untuk membuat sebuah node BST  
7 struct node *newNode(int item){  
8     struct node *temp = (struct node*)malloc(sizeof(struct node));  
9     key(temp) = item;  
10    left(item)= NULL;  
11    right(item)= NULL;  
12    return temp;  
13 }  
14 /* sebuah fungsi utilitas untuk memasukan sebuah node dengan kunci yang  
15 diberikan kedalam BST */  
16 struct node* insert(struct node* node, int key)  
17 {  
18     /* jika tree kosong, return node yang baru */  
19 }
```

```

19     if (node == NULL){
20         return newNode(key) ;
21     /* jika tidak, kembalikan ke tree */
22     if (key < key(node))
23         left(node) = insert(left(node, key));
24     else if (key > key(node))
25         right(node) = insert(right(node, key));
26     /* mengeluarkan pointer yang tidak berubah */
27     return node;
28 }

```

B. Update

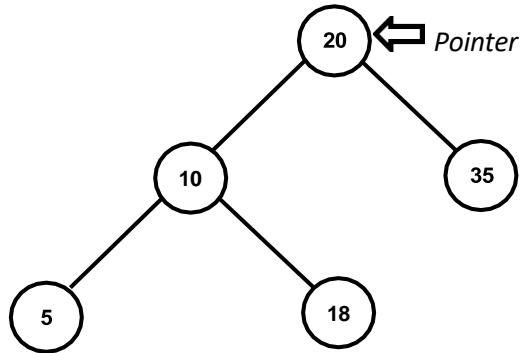
Jika setelah diupdate posisi/lokasi *node* yang bersangkutan tidak sesuai dengan ketentuan, maka harus dilakukan dengan proses **REGENERASI** agar tetap memenuhi kriteria *Binary Search Tree*.

C. Search

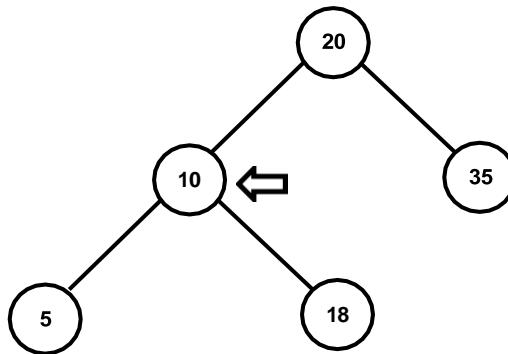
Proses pencarian elemen pada *binary tree* dapat menggunakan algoritma rekursif *binary search*. Berikut adalah algoritma *binary search* :

1. Pencarian pada *binary search tree* dilakukan dengan menaruh *pointer* dan membandingkan nilai yang dicari dengan *node* awal (*root*)
2. Jika nilai yang dicari tidak sama dengan *node*, maka *pointer* akan diganti ke *child* dari *node* yang ditunjuk:
 - a. *Pointer* akan pindah ke *child* kiri bila, nilai dicari lebih kecil dari nilai *node* yang ditunjuk saat itu
 - b. *Pointer* akan pindah ke *child* kanan bila, nilai dicari lebih besar dari nilai *node* yang ditunjuk saat itu
3. Nilai *node* saat itu akan dibandingkan lagi dengan nilai yang dicari dan apabila belum ditemukan, maka perulangan akan kembali ke tahap 2
4. Pencarian akan berhenti saat nilai yang dicari ketemu, atau *pointer* menunjukkan nilai null

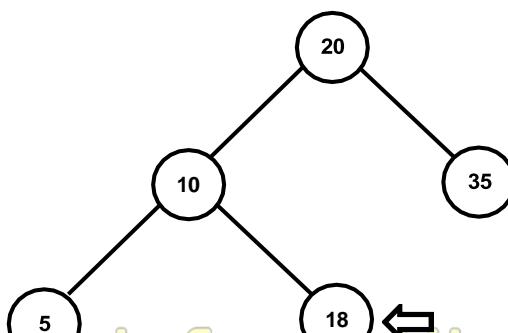
Nilai dicari : 18



Gambar 10-5 Search pada *Binary Search Tree* traversal ke-1

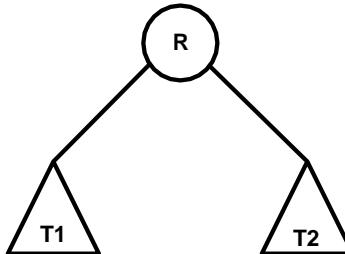


Gambar 10-6 Search pada *Binary Search Tree* traversal ke-1



Gambar 10-7 Search pada *Binary Search Tree* traversal ke-1 : Nilai ketemu

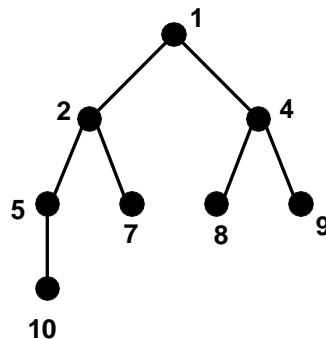
10.8 Traversal pada *Binary Tree*



Gambar 10-8 Traversal pada *Binary Tree* 1

1. *Pre-order* : R, T1, T2
 - kunjungi R
 - kunjungi T1 secara *pre-order*
 - kunjungi T2 secara *pre-order*
2. *In-order* : T1 , R, T2
 - kunjungi T1 secara *in-order*
 - kunjungi R
 - kunjungi T2 secara *in-order*
3. *Post-order* : T1, T2 , R
 - Kunjungi T1 secara *pre-order*
 - kunjungi T2 secara *pre-order*

- kunjungi R



Gambar 10-9 *Traversal* pada *Binary Tree* 2

Sebagai contoh apabila kita mempunyai *tree* dengan representasi seperti di atas ini maka proses *traversal* masing-masing akan menghasilkan output:

1. *Pre-order* : 1-2-5-10-7-4-8-9
2. *In-order*: 10-5-2-7-1-8-4-9
3. *Post-order* : 10-5-7-2-8-9-4-1

Berikut ini ADT untuk *tree* dengan menggunakan representasi *list* linier:

1	#ifndef tree_H
2	#define tree_H
3	#define Nil NULL
4	#define info(P) (P)->info
5	#define right(P) (P)->right
6	#define left(P) (P)->left
7	
8	typedef int infotype;
9	typedef struct Node *address;
10	struct Node{
11	infotype info;
12	address right;
13	address left;
14	};
15	typedef address BinTree;
16	// fungsi primitif pohon biner
17	***** pengecekan apakah tree kosong *****
18	boolean EmptyTree(Tree T);
19	/* mengembalikan nilai true jika tree kosong */
20	
21	***** pembuatan tree kosong *****
22	void CreateTree(Tree &T);
23	/* I.S sembarang
24	F.S. terbentuk Tree kosong */
25	
26	***** manajemen memori *****
27	address alokasi(infotype X);
28	/* mengirimkan address dari alokasi sebuah elemen
29	jika alokasi berhasil maka nilai address tidak Nil dan jika gagal nilai
30	address Nil*/
31	
32	void Dealokasi(address P);
33	/* I.S P terdefinisi
34	F.S. memori yang digunakan P dikembalikan ke sistem */
35	
36	/* Konstruktor */
37	address createElemen(infotype X, address L, address R)
38	

```

39  /* menghasilkan sebuah elemen tree dengan info X dan elemen kiri L dan
40  elemen kanan R
41      mencari elemen tree tertentu */

42
43 address findElmBinTree(Tree T, infotype X);
44 /* mencari apakah ada elemen tree dengan info(P) = X
45     jika ada mengembalikan address element tsb, dan Nil jika sebaliknya */
46
47 address findLeftBinTree(Tree T, infotype X);
48 /* mencari apakah ada elemen sebelah kiri dengan info(P) = X
49     jika ada mengembalikan address element tsb, dan Nil jika sebaliknya */
50
51 address findRigthBinTree(Tree T, infotype X);
52 /* mencari apakah ada elemen sebelah kanan dengan info(P) = X
53     jika ada mengembalikan address element tsb, dan Nil jika sebaliknya */
54
55 /*insert elemen tree */
56 void InsertBinTree(Tree T, address P);
57 /* I.S P Tree bisa saja kosong
58     F.S. memasukka p ke dalam tree terurut sesuai konsep binary tree
59     menghapus elemen tree tertentu*/
60 void DelBinTree(Tree &T, address P);
61 /* I.S P Tree tidak kosong
62     F.S. menghapus p dari Tree selector */
63
64 infotype akar(Tree T);
65 /* mengembalikan nilai dari akar */
66
67 void PreOrder(Tree &T);
68 /* I.S P Tree tidak kosong
69     F.S. menampilkan Tree secara PreOrder */
70
71 void InOrder(Tree &T);
72 /* I.S P Tree tidak kosong
73     F.S. menampilkan Tree secara InOrder */
74
75 void PostOrder(Tree &T);
76 /* I.S P Tree tidak kosong
77     F.S. menampilkan Tree secara PostOrder */
78
#endif

```

10.9 Latihan

- Buatlah ADT *Binary Search Tree* menggunakan *Linked list* sebagai berikut di dalam file “bstree.h”:

```

Type infotype: integer
Type address : pointer to Node
Type Node: <
    info : infotype
    left, right : address
>
    fungsi alokasi( x : infotype ) : address
prosedur insertNode( i/o root : address, i: x : infotype )
function findNode( x : infotype, root : address ) : address
procedure printInorder( root : address )

```

Buatlah implementasi ADT *Binary Search Tree* pada file “bstree.cpp” dan cobalah hasil implementasi ADT pada file “main.cpp”

```

#include <iostream>
#include "bstree.h"

using namespace std;
int main() {
    cout << "Hello World" << endl;

```

```

address root = NULL;
insertNode(root,1);
insertNode(root,2);
insertNode(root,6);
insertNode(root,4);
insertNode(root,5);
insertNode(root,3);
insertNode(root,6);
insertNode(root,7);
InOrder(root);
return 0;
}

```

Gambar 10-10 Main.cpp

```

Hello world!
1 - 2 - 3 - 4 - 5 - 6 - 7 -
Process returned 0 (0x0) execution time : 0.017 s
Press any key to continue.

```

Gambar 10-11 Output

2. Buatlah fungsi untuk menghitung jumlah *node* dengan fungsi berikut.

- fungsi **hitungJumlahNode**(root:address) : integer
/* fungsi mengembalikan integer banyak node yang ada di dalam BST*/

- fungsi **hitungTotalInfo**(root:address, start:integer) : integer
/* fungsi mengembalikan jumlah (total) info dari node-node yang ada di dalam BST*/

- fungsi **hitungKedalaman**(root:address, start:integer) : integer
/* fungsi rekursif mengembalikan integer kedalaman maksimal dari binary tree */

```

int main() {
    cout << "Hello World" << endl;
    address root = NULL;
    insertNode(root,1);
    insertNode(root,2);
    insertNode(root,6);
    insertNode(root,4);
    insertNode(root,5);
    insertNode(root,3);
    insertNode(root,6);
    insertNode(root,7);
    InOrder(root);
    cout<<"\n";
    cout<<"kedalaman : "<<hitungKedalaman(root,0)<<endl;
    cout<<"jumlah Node : "<<hitungNode(root)<<endl;
    cout<<"total : "<<hitungTotal(root)<<endl;
    return 0;
}

```

Gambar 10-12 Main

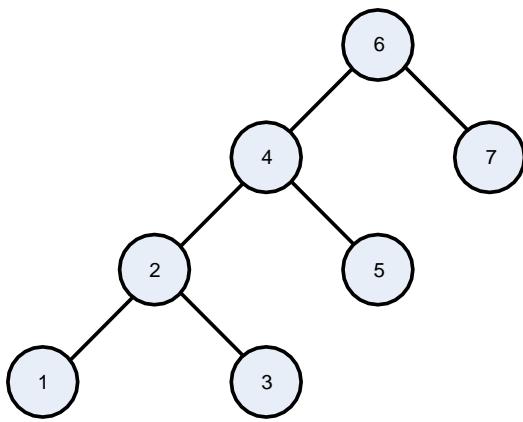
```

Hello world!
1 - 2 - 3 - 4 - 5 - 6 - 7 -
kedalaman : 5
jumlah node : 7
total : 28

```

Gambar 10-13 Output

3. Print *tree* secara *pre-order* dan *post-order*.



Gambar 10-14 Ilustrasi *Tree*



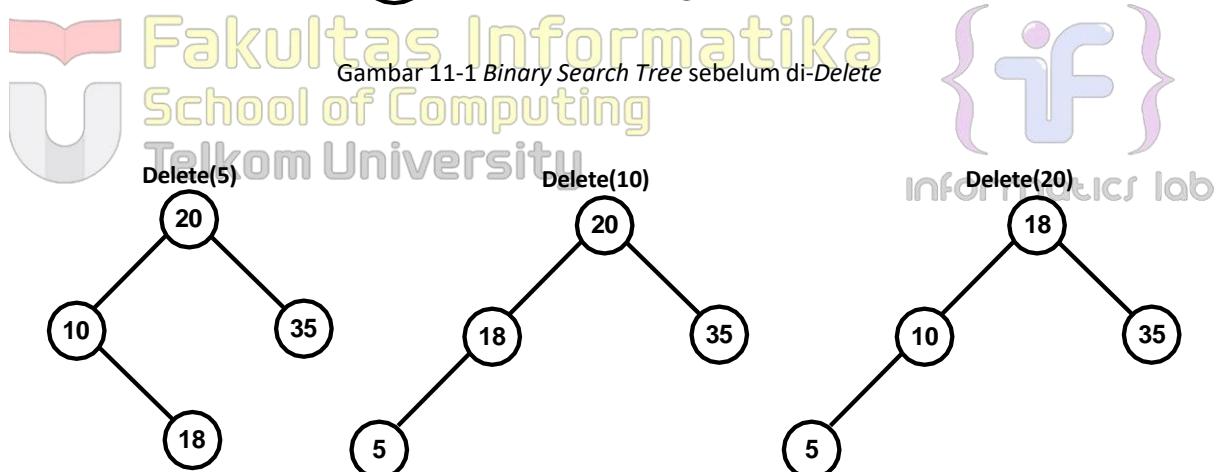
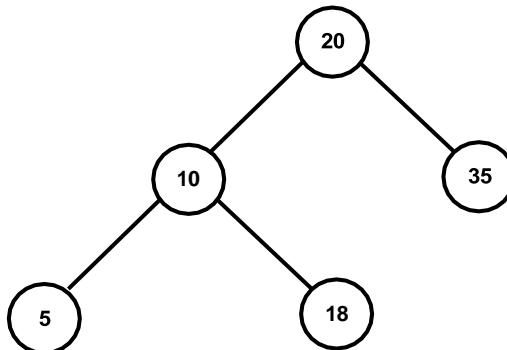
Modul 11 TREE (BAGIAN KEDUA)

TUJUAN PRAKTIKUM

1. Mengimplementasikan struktur data *tree*, khususnya *Binary Tree*.

A. Delete

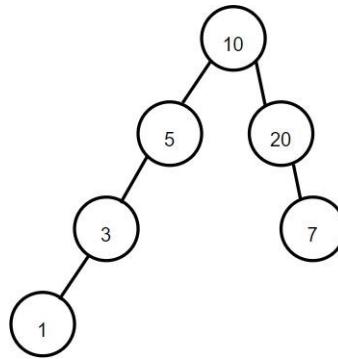
1. *LEAF*, tidak perlu dilakukan modifikasi.
2. *Node* dengan 1 *Child*, maka *child* langsung menggantikan posisi *Parent*.
3. *Node* dengan 2 *Child*:
 - Left *Subtree*, yang diambil adalah *node* yang paling kiri (nilai terbesar).
 - Right *Subtree*, yang diambil adalah *node* yang paling kanan (nilai terkecil).



Gambar 11-2 *Binary Search Tree* setelah di-Delete

B. Most-Left

Most-left node adalah node yang berada paling kiri dalam *tree*. Dalam konteks *binary search tree* (BST), *most-left node* adalah *node* dengan nilai terkecil, yang dapat ditemukan dengan mengikuti anak kiri (*left child*) dari root hingga mencapai node yang tidak memiliki anak kiri lagi.

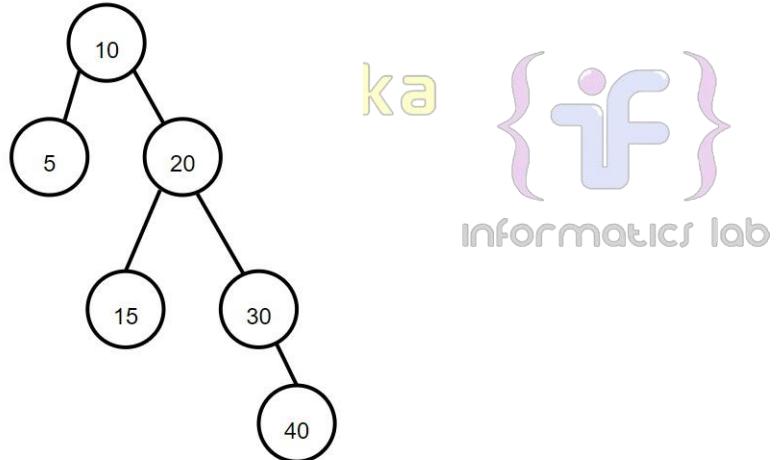


Gambar 11-3 *most-left tree*

Pada *tree* diatas, *most-left tree* adalah = 1

C. Most-Right

Most-right node adalah node yang berada paling kanan dalam *tree*. Dalam konteks *binary search tree* (BST), *most-right node* adalah *node* dengan nilai terbesar, yang dapat ditemukan dengan mengikuti anak kanan (*right child*) dari root hingga mencapai node yang tidak memiliki anak kanan lagi.



Gambar 11-4 *most-right tree*

Pada *tree* diatas, *most-right tree* adalah = 40

Modul 12 ASISTENSI TUGAS BESAR

Pada modul 12, praktikan akan mempresentasikan progress tugas besarnya.



Modul 13 MULTI LINKED LIST

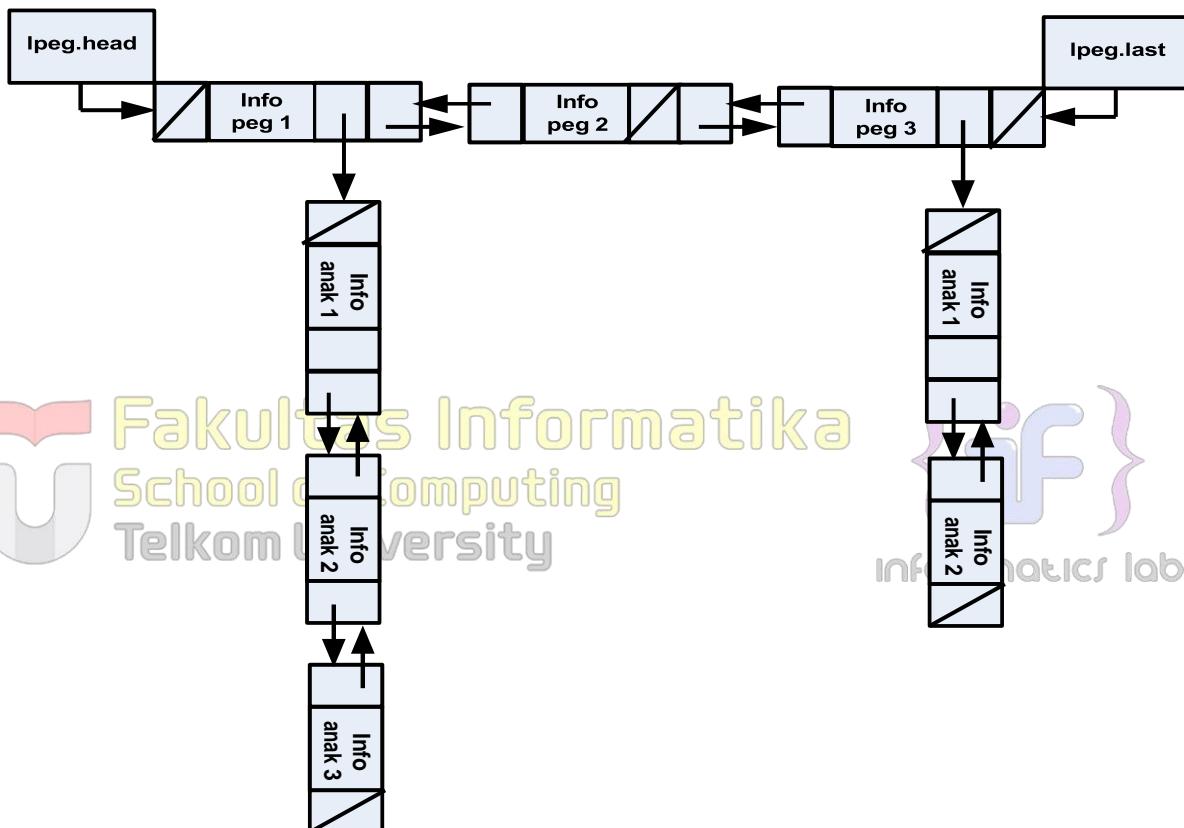
TUJUAN PRAKTIKUM

1. Memahami penggunaan *Multi Linked list*.
2. Mengimplementasikan *Multi Linked list* dalam beberapa studi kasus.

13.1 Multi Linked List

Multi List merupakan sekumpulan *list* yang berbeda yang memiliki suatu keterhubungan satu sama lain. Tiap elemen dalam *multi link list* dapat membentuk *list* sendiri. Biasanya ada yang bersifat sebagai *list* induk dan *list* anak .

Contoh *Multi Linked list* dapat dilihat pada gambar berikut.



Gambar 13-1 Multi *Linked list*

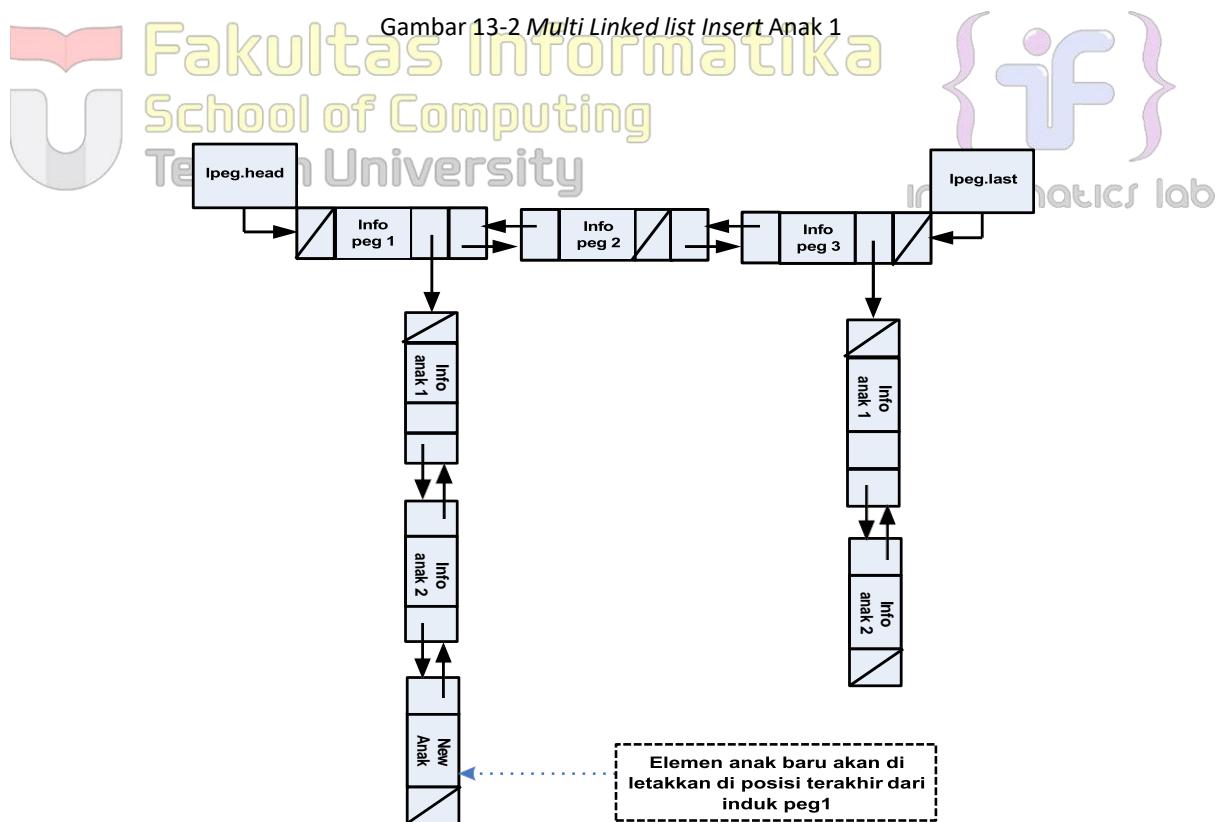
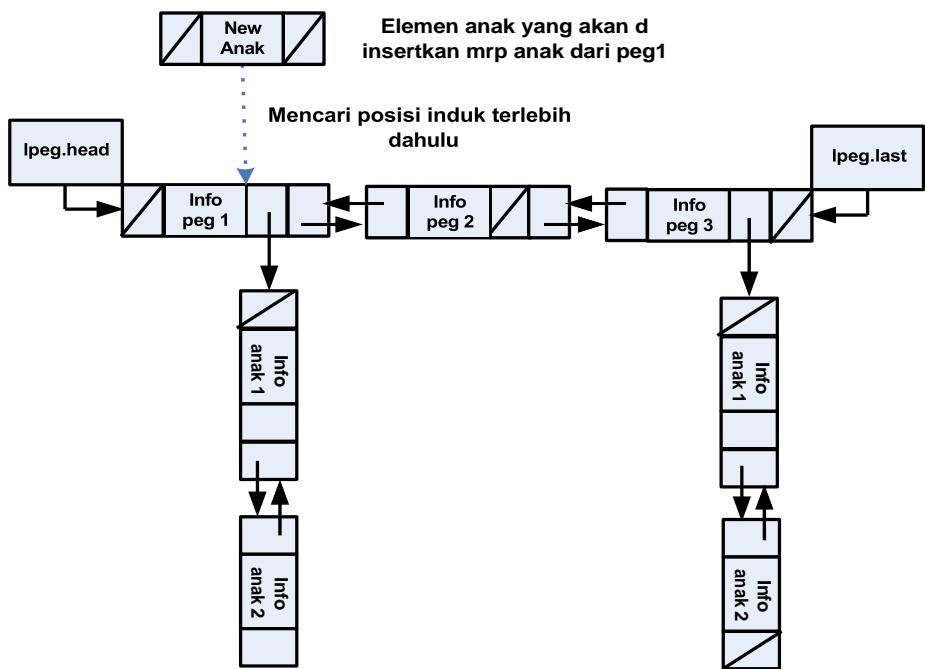
Jadi , dari implementasi di atas akan terdapat dua buah *list*, *list* pegawai dan *list* anak. Dimana untuk *list* pegawai menunjuk satu buah *list* anak. Disini *list* induknya adalah *list* pegawai dan *list* anaknya adalah *list* anak.

13.1.1 Insert

A. Insert Anak

Dalam penambahan elemen anak harus diketahui dulu elemen induknya.

Berikut ini ilustrasi *insert anak* dengan konsep *insert last*:



Gambar 13-3 Multi Linked list Insert Anak 2

```
/* buat dahulu elemen yang akan disisipkan */
address_anak alokasiAnak(infotypeanak X){
```

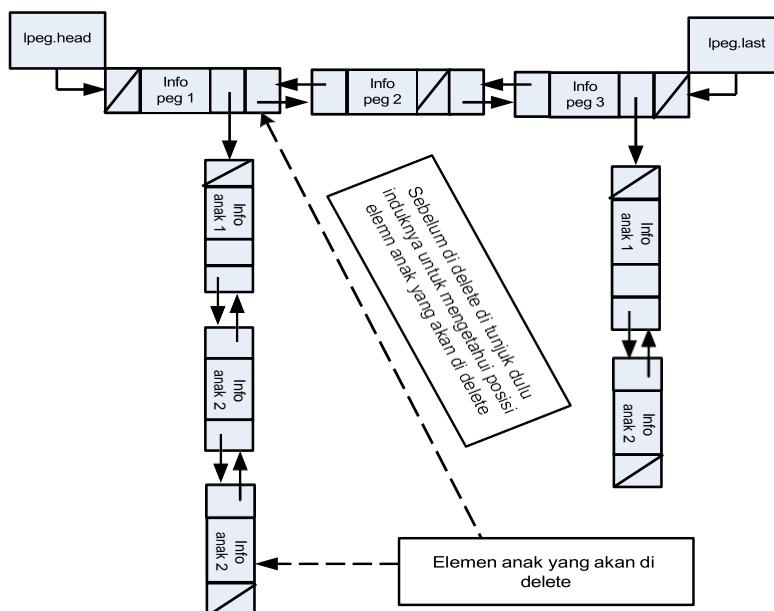
```

address_anak p = alokasi(X);
next(p) = null;
prev(p) = null;
return p;
}
/* mencari apakah ada elemen pegawai dengan info X */
address findElm(listinduk L, infotypeinduk X){
    address cariInduk = head(L);
    do{
        if(cariInduk.info == X){
            return cariInduk;
        }else{
            cariInduk = next(cariInduk);
        }
    }while(cariInduk.info!=X || cariInduk!=last(L))
}
/* menyisipkan anak pada akhir list anak */
void insertLastAnak(listanak &Lanak, address_anak P){
    address_anak ! = head(&Lanak);
    do{
        Q = next(Q);
    }while(next(&Lanak) !=NULL)
    next(Q) = P;
    prev(P) = Q;
    next(P) = NULL;
}

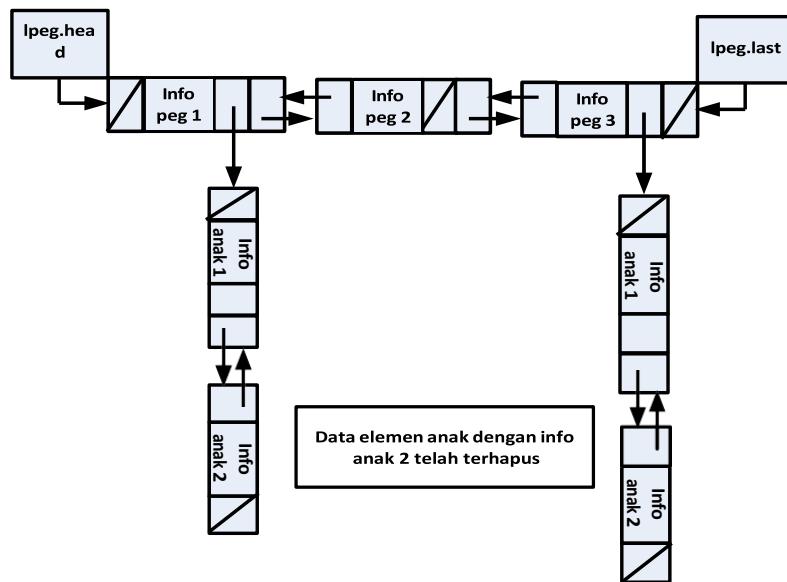
```

B. Insert Induk

Untuk *insert* elemen induk sama dengan konsep *insert* pada *single*, *double* dan *circular linked list*.



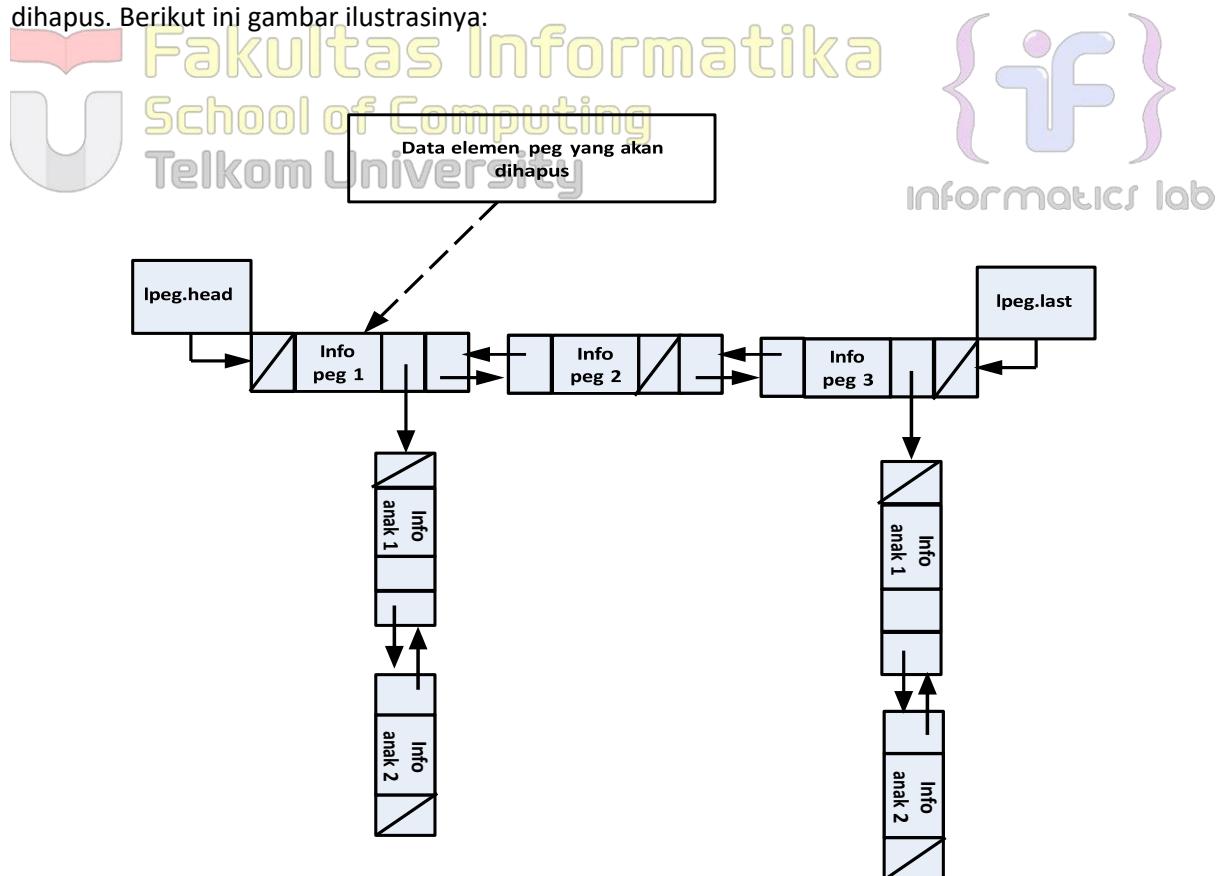
Gambar 13-4 Multi Linked list Delete Anak 1



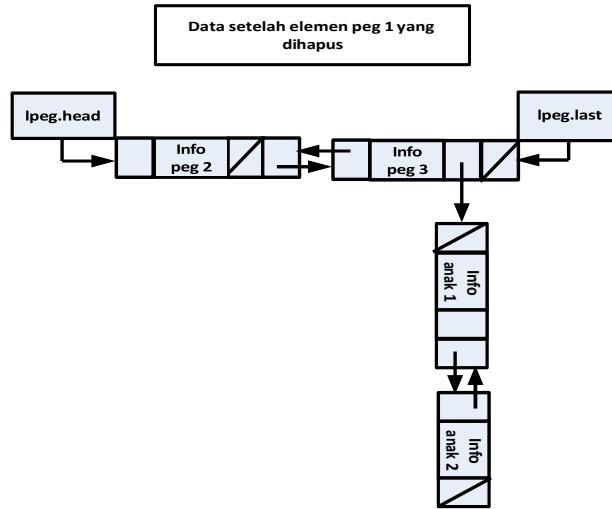
Gambar 13-5 Multi Linked list Delete Anak 2

B. Delete Induk

Untuk *delete* elemen induk maka saat dihapus maka seluruh anak dengan induk tersebut juga harus dihapus. Berikut ini gambar ilustrasinya:



Gambar 13-6 Multi Linked list Delete Induk 1



Gambar 13-7 Multi Linked list Delete Induk 2

```

1  /*file : multilist .h*/
2  /* contoh ADT list berkait dengan representasi fisik pointer*/
3  /* representasi address dengan pointer*/
4
5  /* info tipe adalah integer */
6  #ifndef MULTILIST_H_INCLUDED
7  #define MULTILIST_H_INCLUDED
8  #include <stdio.h>
9  #define Nil NULL
10 #define info(P) (P)->info
11 #define next(P) (P)->next
12 #define first(L) ((L).first)
13 #define last(L) ((L).last)
14
15 typedef int infotypeanak;
16 typedef int infotypeinduk;
17 typedef struct elemen_list_anak *address;
18 typedef struct elemen_list_anak *address_anak;
19 /* define list : */
20
21 /* list kosong jika first(L)=Nil
22 setiap elemen address P dapat diacu info(P) atau next(P)
23 elemen terakhir list jika addressnya last, maka next(last) = Nil */
24 struct elemen_list_anak{
25 /* struct ini untuk menyimpan elemen anak dan pointer penunjuk
26 elemen tetangganya */
27     infotypeanak info;
28     address_anak next;
29     address_anak prev;
30 };
31
32 struct listanak {
33 /* struct ini digunakan untuk menyimpan list anak itu sendiri */
34     address_anak first;
35     address_anak last;
36 };
37
38 struct elemen_list_induk{
39 /* struct ini untuk menyimpan elemen induk dan pointer penunjuk
40 elemen tetangganya */
41     infotypeanak info;
42     listanak lanak;
43     address next;
44     address prev;

```

```

44    };
45    struct listinduk {
46        /* struct ini digunakan untuk menyimpan list induk itu sendiri */
47        address first;
48        address last;
49    };
50
51    ***** pengecekan apakah list kosong *****/
52    boolean ListEmpty(listinduk L);
53    /*mengembalikan nilai true jika list induk kosong*/
54    boolean ListEmptyAnak(listanak L);
55    /*mengembalikan nilai true jika list anak kosong*/
56
57    ***** pembuatan list kosong *****/
58    void CreateList(listinduk &L);
59    /* I.S. sembarang
60        F.S. terbentuk list induk kosong*/
61    void CreateListAnak(listanak &L);
62    /* I.S. sembarang
63        F.S. terbentuk list anak kosong*/
64
65    ***** manajemen memori *****/
66    address alokasi(infotypeinduk P);
67    /* mengirimkan address dari alokasi sebuah elemen induk
68        jika alokasi berhasil, maka nilai address tidak Nil dan jika gagal
69        nilai address Nil */
70
71    address_anak alokasiAnak(infotypeanak P);
72    /* mengirimkan address dari alokasi sebuah elemen anak
73        jika alokasi berhasil, maka nilai address tidak Nil dan jika gagal
74        nilai address_anak Nil */
75
76    void dealokasi(address P);
77    /* I.S. P terdefinisi
78        F.S. memori yang digunakan P dikembalikan ke sistem */
79
80    void dealokasiAnak(address_anak P);
81    /* I.S. P terdefinisi
82        F.S. memori yang digunakan P dikembalikan ke sistem */
83    ***** pencarian sebuah elemen list *****/
84    address findElm(listinduk L, infotypeinduk X);
85    /* mencari apakah ada elemen list dengan info(P) = X
86        jika ada, mengembalikan address elemen tab tsb, dan Nil jika sebaliknya
87    */
88    address_anak findElm(listanak Lanak, infotypeanak X);
89    /* mencari apakah ada elemen list dengan info(P) = X
90        jika ada, mengembalikan address elemen tab tsb, dan Nil jika sebaliknya
91    */
92    boolean fFindElm(listinduk L, address P);
93    /* mencari apakah ada elemen list dengan alamat P
94        mengembalikan true jika ada dan false jika tidak ada */
95    boolean fFindElmanak(listanak Lanak, address_anak P);
96    /* mencari apakah ada elemen list dengan alamat P
97        mengembalikan true jika ada dan false jika tidak ada */
98
99    address findBefore(listinduk L, address P);
100   /* mengembalikan address elemen sebelum P
101      jika P berada pada awal list, maka mengembalikan nilai Nil */
102   address_anak findBeforeAnak(listanak Lanak, infotypeinduk X, address_anak
103   P);
104   /* mengembalikan address elemen sebelum P dimana info(P) = X
105      jika P berada pada awal list, maka mengembalikan nilai Nil */
106
107  ***** penambahan elemen *****/
108  void insertFirst(listinduk &L, address P);
109  /* I.S. sembarang, P sudah dialokasikan
110    F.S. menempatkan elemen beralamat P pada awal list */

```

```

111 void insertAfter(listinduk &L, address P, address Prec);
112 /* I.S. sembarang, P dan Prec alamt salah satu elemen list
113   F.S. menempatkan elemen beralamat P sesudah elemen beralamat Prec */
114
115 void insertLast(listinduk &L, address P);
116 /* I.S. sembarang, P sudah dialokasikan
117   F.S. menempatkan elemen beralamat P pada akhir list */
118
119 void insertFirstAnak(listanak &L, address_anak P);
120 /* I.S. sembarang, P sudah dialokasikan
121   F.S. menempatkan elemen beralamat P pada awal list */
122
123 void insertAfterAnak(listanak &L, address_anak P, address_anak Prec);
124 /* I.S. sembarang, P dan Prec alamt salah satu elemen list
125   F.S. menempatkan elemen beralamat P sesudah elemen beralamat Prec */
126
127 void insertLastAnak(listanak &L, address_anak P);
128 /* I.S. sembarang, P sudah dialokasikan
129   F.S. menempatkan elemen beralamat P pada akhir list */
130
131 /***** penghapusan sebuah elemen *****/
132 void delFirst(listinduk &L, address &P);
133 /* I.S. list tidak kosong
134   F.S. adalah alamat dari alamat elemen pertama list
135   sebelum elemen pertama list dihapus
136   elemen pertama list hilang dan list mungkin menjadi kosong
137   first elemen yang baru adalah successor first elemen yang lama */
138 void delLast(listinduk &L, address &P);
139 /* I.S. list tidak kosong
140   F.S. adalah alamat dari alamat elemen terakhir list
141   sebelum elemen terakhir list dihapus
142   elemen terakhir list hilang dan list mungkin menjadi kosong
143   last elemen yang baru adalah successor last elemen yang lama */
144
145 void delAfter(listinduk &L, address &P, address Prec);
146 /* I.S. list tidak kosng, Prec alamat salah satu elemen list
147   F.S. P adalah alamatdari next(Prec), menghapus next(Prec) dari list */
148 void delP (listinduk &L, infotypeinduk X);
149 /* I.S. sembarang
150   F.S. jika ada elemen list dengan alamat P, dimana info(P)=X, maka P
151   dihapus
152   dan P di-dealokasi, jika tidak ada maka list tetap
153   list mungkin akan menjadi kosong karena penghapusan */
154
155 void delFirstAnak(listanak &L, address_anak &P);
156 /* I.S. list tidak kosong
157   F.S. adalah alamat dari alamat elemen pertama list
158   sebelum elemen pertama list dihapus
159   elemen pertama list hilang dan list mungkin menjadi kosong
160   first elemen yang baru adalah successor first elemen yang lama */
161 void delLastAnak(listanak &L, address_anak &P);
162 /* I.S. list tidak kosong
163   F.S. adalah alamat dari alamat elemen terakhir list
164   sebelum elemen terakhir list dihapus
165   elemen terakhir list hilang dan list mungkin menjadi kosong
166   last elemen yang baru adalah successor last elemen yang lama */
167
168 void delAfterAnak(listanak &L, address_anak &P, address_anak Prec);
169 /* I.S. list tidak kosng, Prec alamat salah satu elemen list
170   F.S. P adalah alamatdari next(Prec), menghapus next(Prec) dari list */
171 void delPanak (listanak &L, infotypeanak X);
172 /* I.S. sembarang
173   F.S. jika ada elemen list dengan alamat P, dimana info(P)=X, maka P
174   dihapus
175   dan P di-dealokasi, jika tidak ada maka list tetap
176   list mungkin akan menjadi kosong karena penghapusan */
177

```

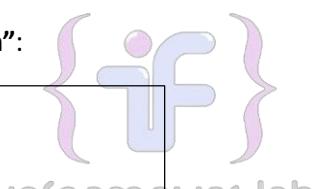
```

178  **** proses semua elemen list *****/
179  void printInfo(list L);
180  /* I.S. list mungkin kosong
181   F.S. jika list tidak kosong menampilkan semua info yang ada pada list
182 */
183
184  int nbList(list L);
185  /* mengembalikan jumlah elemen pada list */
186
187  void printInfoAnak(listanak Lanak);
188  /* I.S. list mungkin kosong
189   F.S. jika list tidak kosong menampilkan semua info yang ada pada list
190 */
191
192  int nbListAnak(listanak Lanak);
193  /* mengembalikan jumlah elemen pada list anak */
194
195  **** proses terhadap list *****/
196  void delAll(listinduk &L);
197  /* menghapus semua elemen list dan semua elemen di-dealokasi */
198
199 #endif

```

13.2 Latihan

2. Perhatikan program 46 **multilist.h**, buat **multilist.cpp** untuk implementasi semua fungsi pada **multilist.h**. Buat **main.cpp** untuk pemanggilan fungsi-fungsi tersebut.
3. Buatlah ADT Multi Linked list sebagai berikut di dalam file “**circularlist.h**”:

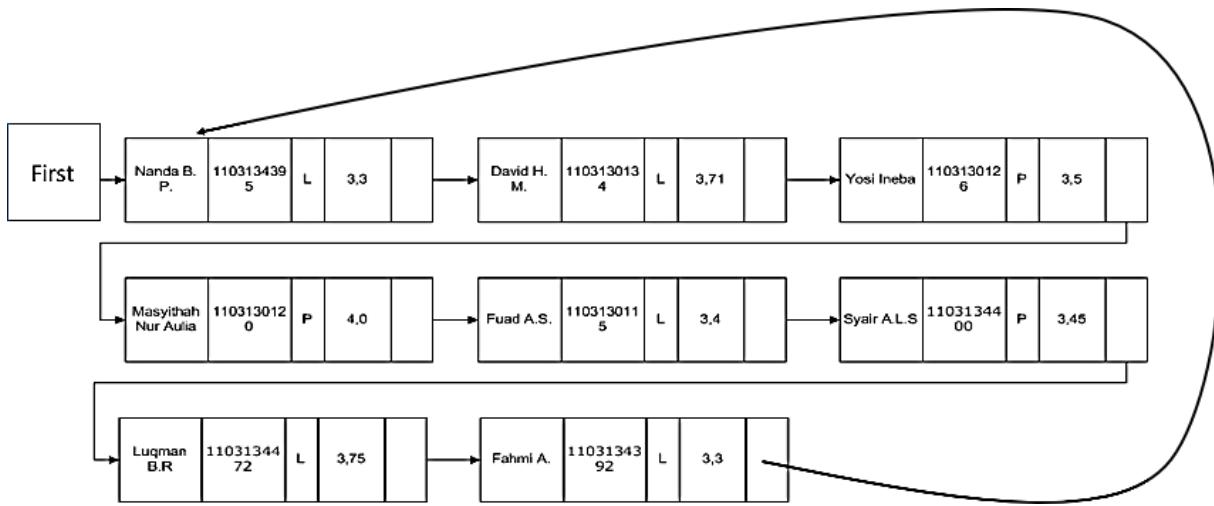



 Type infotype : mahasiswa <
 Nama:string
 Nim:string
 Jenis_kelamin:char
 Ipk:float>
 Type address : pointer to ElmList
 Type ElmList <
 info : infotype
 next :address>
 Type List <
 First : address>

- Terdapat 11 fungsi/prosedur untuk ADT circularlist
 - prosedur **CreateList**(in/out L : List)
 - fungsi **alokasi**(x : infotype) : address
 - prosedur **dealokasi**(in/out P : address)
 - prosedur **insertFirst**(in/out L : List, in P : address)
 - prosedur **insertAfter**(in/out L : List, in Prec : address, P : address)
 - prosedur **insertLast**(in/out L : List, in P : address)
 - prosedur **deleteFirst**(in/out L : List, in/out P : address)
 - prosedur **deleteAfter**(in/out L : List, in Prec : address, in/out P : address)
 - prosedur **deleteLast**(in/out L : List, in/out P : address)
 - fungsi **findElm**(L : List, x : infotype) : address
 - prosedur **printInfo**(in L : List)

Keterangan :

- fungsi **findElm** mencari elemen di dalam *list* L berdasarkan nim
 - fungsi mengembalikan elemen dengan dengan info nim == x.nim jika ditemukan
 - fungsi mengembalikan NIL jika tidak ditemukan



Gambar 13-8 Ilustrasi data

Buatlah implementasi ADT *Double Linked list* pada file “*circularlist.cpp*”. Tambahkan fungsi/prosedur berikut pada file “*main.cpp*”.

- fungsi **create (in nama, nim : string, jenis_kelamin : char, ipk : float)**
 - fungsi disediakan, ketik ulang code yang diberikan
 - fungsi mengalokasikan sebuah elemen *list* dengan info sesuai *input*

```

address createData(string nama, string nim, char jenis_kelamin, float ipk)
{
    /**
     * PR : mengalokasikan sebuah elemen list dengan info dengan info sesuai input
     * FS : address P menunjuk elemen dengan info sesuai input
     */
    infotype x;
    address P;
    x.nama = nama;
    x.nim = nim;
    x.jenis_kelamin = jenis_kelamin;
    x.ipk = ipk;
    P = alokasi(x);
    return P;
}
  
```

Gambar 13-9 Fungsi *create*

Cobalah hasil implementasi ADT pada file “*main.cpp*”

```

int main()
{
    List L, A, B, L2;
    address P1 = NULL;
    address P2 = NULL;
    infotype x;
    createList(L);

    cout<<"coba insert first, last, dan after"<<endl;
    P1 = createData("Danu", "04", 'l', 4.0);
    insertFirst(L,P1);

    P1 = createData("Fahmi", "06", 'l', 3.45);
    insertLast(L,P1);
    P1 = createData("Bobi", "02", 'l', 3.71);
    insertFirst(L,P1);
  
```

```

P1 = createData("Ali", "01", 'l', 3.3);
insertFirst(L,P1);

P1 = createData("Gita", "07", 'p', 3.75);
insertLast(L,P1);

x.nim = "07";
P1 = findElm(L,x);
P2 = createData("Cindi", "03", 'p', 3.5);
insertAfter(L, P1, P2);

x.nim = "02";
P1 = findElm(L,x);
P2 = createData("Hilmi", "08", 'p', 3.3);
insertAfter(L, P1, P2);

x.nim = "04";
P1 = findElm(L,x);
P2 = createData("Eli", "05", 'p', 3.4);
insertAfter(L, P1, P2);
printInfo(L);
return 0;
}

```

```

coba insert first, last, dan after
Nama : Ali
NIM : 01
L/P : l
IPK : 3.3

Nama : Bobi
NIM : 02
L/P : l
IPK : 3.71

Nama : Cindi
NIM : 03
L/P : p
IPK : 3.5

Nama : Danu
NIM : 04
L/P : l
IPK : 4

Nama : Eli
NIM : 05
L/P : p
IPK : 3.4

Nama : Fahmi
NIM : 06
L/P : l
IPK : 3.45

Nama : Gita
NIM : 07
L/P : p
IPK : 3.75

Nama : Hilmi
NIM : 08
L/P : l
IPK : 3.3

```

Gambar 13-10 Output

Gambar 13-11 Main.cpp



Modul 14 GRAPH

TUJUAN PRAKTIKUM

1. Memahami konsep *graph*
2. Mengimplementasikan *graph* dengan menggunakan *pointer*.

14.1 Pengertian

Graph merupakan himpunan tidak kosong dari *node* (*vertec*) dan garis penghubung (*edge*). Contoh sederhana tentang *graph*, yaitu antara Tempat Kost Anda dengan *Common Lab*. Tempat Kost Anda dan *Common Lab* merupakan *node* (*vertec*). Jalan yang menghubungkan tempat Kost dan *Common Lab* merupakan garis penghubung antara keduanya (*edge*).

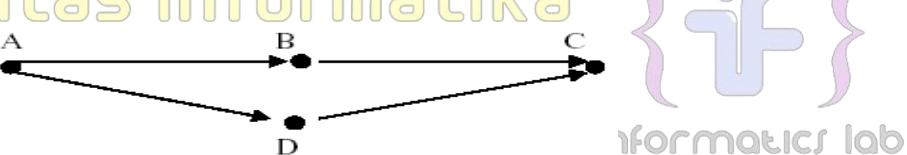


Gambar 14-1 Graph Kost dan Common Lab

14.2 Jenis-Jenis Graph

14.2.1 Graph Berarah (Directed Graph)

Merupakan *graph* dimana tiap *node* memiliki *edge* yang memiliki arah, kemana *node* tersebut dihubungkan.

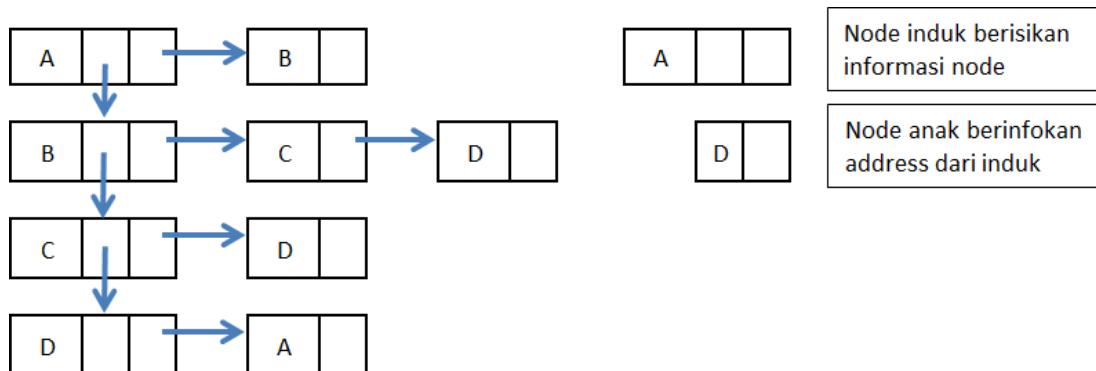


Gambar 14-2 Graph Berarah (Directed Graph)

A. Representasi Graph

Pada dasarnya representasi dari *graph* berarah sama dengan *graph* tak-berarah. Perbedaannya apabila *graph* tak-berarah terdapat *node* A dan *node* B yang terhubung, secara otomatis terbentuk panah bolak-balik dari A ke B dan B ke A. Pada *Graph* berarah *node* A terhubung dengan *node* B, belum tentu *node* B terhubung dengan *node* A.

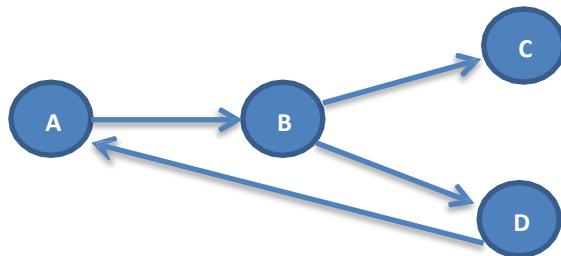
Pada *graph* berarah bisa diwujudkan dalam multilist sebagai berikut,



Gambar 14-3 Graph Representasi Multilist

Dalam praktikum ini untuk merepresentasikan *graph* akan menggunakan *multilist*. Karena sifat *list* yang dinamis.

Dari *multilist* di atas apabila digambarkan dalam bentuk *graph* menjadi :



Gambar 14-4 Graph

B. Topological Sort

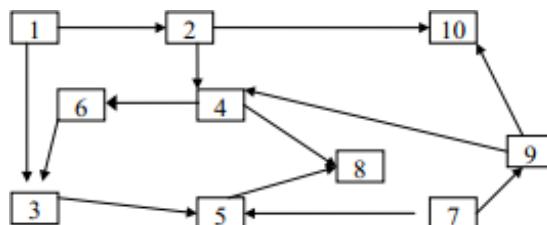
a. Pengertian

Diberikan urutan *partial* dari elemen suatu himpunan, dikehendaki agar elemen yang terurut parsial tersebut mempunyai keterurutan linier. Contoh dari keterurutan parsial banyak dijumpai dalam kehidupan sehari-hari, misalnya:

1. Dalam suatu kurikulum, suatu mata pelajaran mempunyai *prerequisite* mata pelajaran lain. Urutan linier adalah urutan untuk seluruh mata pelajaran dalam kurikulum
2. Dalam suatu proyek, suatu pekerjaan harus dikerjakan lebih dulu dari pekerjaan lain (misalnya membuat fondasi harus sebelum dinding, membuat dinding harus sebelum pintu. Namun pintu dapat dikerjakan bersamaan dengan jendela, dsb)
3. Dalam sebuah program Pascal, pemanggilan prosedur harus sedemikian rupa, sehingga peletakan prosedur pada teks program harus sesuai dengan urutan (*partial*) pemanggilan.

Dalam pembuatan tabel pada basis data, tabel yang di-refer oleh tabel lain harus dideklarasikan terlebih dulu. Jika suatu aplikasi terdiri dari banyak tabel, maka urutan pembuatan tabel harus sesuai dengan definisinya.

Jika $X < Y$ adalah simbol untuk X “sebelum” Y , dan keterurutan *partial* digambarkan sebagai graf, maka graf sebagai berikut :



Gambar 14-5 Contoh Graph

akan dikatakan mempunyai keterurutan *partial*

$$\begin{array}{ccccccc} 1 < 2 & 2 < 4 & 4 < 6 & 2 < 10 & 4 < 8 & 6 < 3 & 1 < 3 \\ 3 < 5 & 5 < 8 & 7 < 5 & 7 < 9 & 9 < 4 & 9 < 10 & \end{array}$$

Dan yang SALAH SATU urutan linier adalah graf sebagai berikut :



Gambar 14-6 Urutan Linier Graph

Kenapa disebut salah satu urutan linier ? Karena suatu urutan *partial* akan mempunyai banyak urutan linier yang mungkin dibentuk dari urutan *partial* tersebut. Elemen yang membentuk urutan linier disebut sebagai “*list*”.

Proses yang dilakukan untuk mendapatkan urutan linier :

1. Andaikata item yang mempunyai keterurutan *partial* adalah anggota himpunan S.
2. Pilih salah satu item yang tidak mempunyai *predecessor*, misalnya X. Minimal adasatu elemen semacam ini. Jika tidak, maka akan looping.
3. Hapus X dari himpunan S, dan *insert* ke dalam *list*
4. Sisa himpunan S masih merupakan himpunan terurut *partial*, maka proses 2-3 dapat dilakukan lagi terhadap sisa dari S
5. Lakukan sampai S menjadi kosong, dan *list* Hasil mempunyai elemen dengan keterurutan linier

Solusi I :

Untuk melakukan hal ini, perlu ditentukan suatu representasi internal. Operasi yang penting adalah memilih elemen tanpa *predecessor* (yaitu jumlah *predecessor* elemen sama dengan nol). Maka setiap elemen mempunyai 3 karakteristik : identifikasi, *list* suksesornya, dan banyaknya *predecessor*. Karena jumlah elemen bervariasi, maka representasi yang paling cocok adalah *list* berkait dengan representasi dinamis (*pointer*). *List* dari *successor* direpresentasi pula secara berkait.

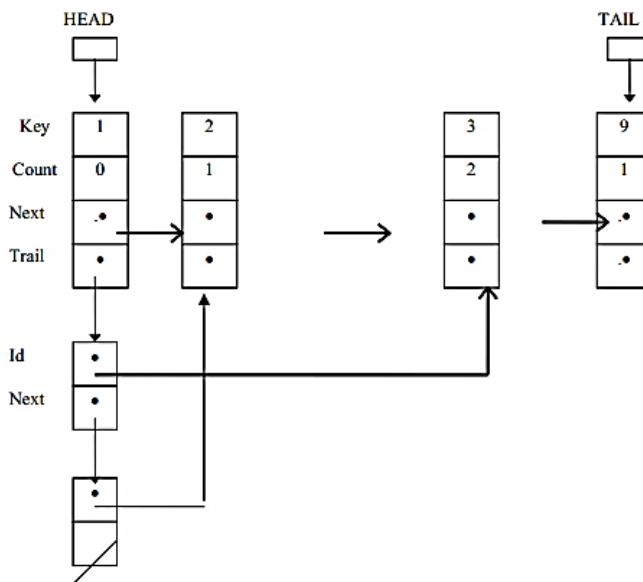
Representasi yang dipilih untuk persoalan ini adalah multilist sebagai berikut :

1. *List* yang digambarkan horisontal adalah *list* dari banyaknya *predecessor* setiap item, disebut *list* “*Leader*”, yang direpresentasi sebagai *list* yang dicatat alamat elemen pertama dan terakhir (*Head-Tail*) serta elemen terurut menurut *key*. *List* ini dibentuk dari pembacaan data. Untuk setiap data keterurutan *partial* $X < Y$: Jika X dan/atau Y belum ada pada *list Leader*, *insert* pada *Tail* dengan metoda *search* dengan sentinel.
2. *List* yang digambarkan vertikal (ke bawah) adalah *list* yang merupakan *indirect addressing* ke setiap *predecessor*, disebut sebagai “*Trailer*”. Untuk setiap elemen *list Leader* X, *list* dari suksesornya disimpan sebagai elemen *list Trailer* yang setiap elemennya berisi alamat dari *successor*. Penyisipan data suatu *successor* ($X < Y$), dengan diketahui X, maka akan dilakukan dengan *InsertFirst* alamat Y sebagai elemen *list Trailer* dengan *key* X.

Algoritma secara keseluruhan terdiri dari dua pass :

1. Bentuk *list leader* dan *Trailer* dari data keterurutan *partial* : baca pasangan nilai ($X < Y$). Temukan alamat X dan Y (jika belum ada sisipkan), kemudian dengan mengetahui alamat X dan Y pada *list Leader*, *InsertFirst* alamat Y sebagai *trailer* X
2. Lakukan *topological sort* dengan melakukan *search list Leader* dengan jumlah *predecessor*=0, kemudian *insert* sebagai elemen *list* linier hasil pengurutan.

Ilustrasi umum dari *list Leader* dan *Trailer* untuk representasi internal persoalan *topological sorting* adalah sebagai berikut.



Gambar 14-7 Solusi 1

Solusi II : pendekatan “fungsional” dengan *list* linier sederhana.

Pada solusi ini, proses untuk mendapatkan urutan linier diterjemahkan secara fungsional, dengan representasi sederhana. Graf *partial* dinyatakan sebagai *list* linier dengan representasi fisik *First-Last* dengan *dummy* seperti representasi pada Solusi I. dengan elemen yang terdiri dari <Precc,Succ>. Contoh: sebuah elemen bernilai <1,2> artinya 1 adalah *predecessor* dari 2.

Langkah :

1. Fase *input*: Bentuk *list* linier yang merepresentasi graf seperti pada solusi I.
2. Fase *output*: Ulangi langkah berikut sampai *list* “habis”, artinya semua elemen *list* selesai ditulis sesuai dengan urutan total.
 - P adalah elemen pertama (*First(L)*)
 - *Search* pada sisa *list*, apakah X=Precc(P) mempunyai *predecessor*.
 - Jika ya, maka elemen ini harus dipertahankan sampai saatnya dapat dihapus dari *list* untuk dioutputkan:
 - *Delete* P, tapi jangan didealokasi
 - *Insert* P sebagai Last(L) yang baru
 - Jika tidak mempunyai *predecessor*, maka X siap untuk di-*output*-kan, tetapi Y masih harus dipertanyakan. Maka langkah yang harus dilakukan :
 - *Output*-kan X
 - *Search* apakah Y masih ada pada sisa *list*, baik sebagai Precc maupun Succ
 - Jika ya, maka Y akan dioutputkan nanti. Hapus elemen pertama yang sedangkan diproses dari *list*
 - Jika tidak muncul sama sekali, berarti Y tidak mempunyai *predecessor*. Maka *output*-kan Y, baru hapus elemen pertama dari *list*

b. Representasi *Topological Sort*

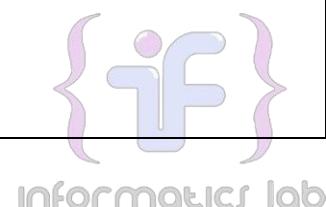
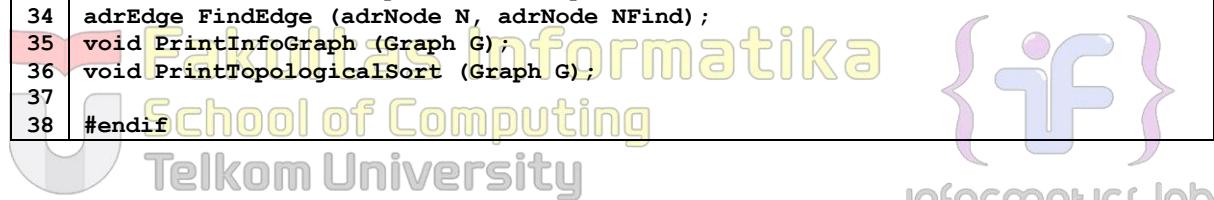
Representasi *graph* untuk *topological sort* sama dengan *graph* berarah pada umumnya.

1	#ifndef GRAPH_H_INCLUDE
2	#define GRAPH_H_INCLUDE
3	#include <stdio.h>

```

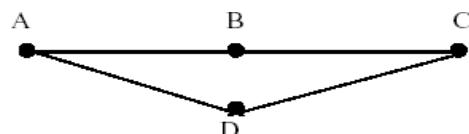
4 #include <stdlib.h>
5 #include <conio.h>
6
7 typedef int infoGraph;
8 typedef struct ElmNode *adrNode;
9 typedef struct ElmEdge *adrEdge;
10
11 struct ElmNode{
12     infoGraph info;
13     int Visited;
14     int Pred;
15     adrEdge firstEdge;
16     adrNode Next;
17 };
18 struct ElmEdge{
19     adrNode Node;
20     adrEdge Next;
21 };
22 struct Graph {
23     adrNode First;
24 };
25
26 adrNode AllocateNode (infoGraph X);
27 adrEdge AllocateEdge (adrNode N);
28 void CreateGraph (Graph &G);
29 void InsertNode (Graph &G, infoGraph X);
30 void DeleteNode (Graph &G, infoGraph X);
31 void ConnectNode (adrNode N1, adrNode N2);
32 void DisconnectNode (adrNode N1, adrNode N2);
33 adrNode FindNode (Graph G, infoGraph X);
34 adrEdge FindEdge (adrNode N, adrNode NFind);
35 void PrintInfoGraph (Graph G);
36 void PrintTopologicalSort (Graph G);
37
38 #endif

```



14.2.2 Graph Tidak Berarah (Undirected Graph)

Merupakan *graph* dimana tiap *node* memiliki edge yang dihubungkan ke *node* lain tanpa arah.



Gambar 14-8 Graph Tidak Berarah (Undirected Graph)

Selain arah, beban atau nilai sering ditambahkan pada *edge*. Misalnya nilai yang merepresentasikan panjang, atau biaya transportasi, dan lain-lain. Hal mendasar lain yang perlu diketahui adalah, suatu *node* A dikatakan bertetangga dengan *node* B jika antara *node* A dan *node* B dihubungkan langsung dengan sebuah *edge*.

Misalnya:

Dari gambar contoh *graph* pada halaman sebelumnya dapat disimpulkan bahwa: A bertetangga dengan B, B bertetangga dengan C, A tidak bertetangga dengan C, B tidak bertetangga dengan D.

Masalah ketetanggaan suatu *node* dengan *node* yang lain harus benar-benar

diperhatikan dalam implementasi pada program. Ketetanggaan dapat diartikan sebagai keterhubungan antar *node* yang nantinya informasi ini dibutuhkan untuk melakukan beberapa proses seperti : mencari lintasan terpendek dari suatu *node* ke *node* yang lain, pengubahan *graph* menjadi *tree* (untuk perancangan jaringan) dan lain-lain.

Tentu anda sudah tidak asing dengan algoritma Djikstra, Kruskal, Prim dsb. Karena waktu praktikum terbatas, kita tidak membahas algoritma tersebut. Di sini anda hanya akan mencoba untuk mengimplementasikan *graph* dalam program.

A. Representasi Graph

Dari definisi *graph* dapat kita simpulkan bahwa *graph* dapat direpresentasikan dengan Matrik Ketetanggan (*Adjacency Matrices*), yaitu matrik yang menyatakan keterhubungan antar *node* dalam *graph*. Implementasi matrik ketetanggan dalam bahasa pemrograman dapat berupa : *Array 2 Dimensi* dan *Multi Linked List*. *Graph* dapat direpresentasikan dengan matrik $n \times n$, dimana n merupakan jumlah *node* dalam *graph* tersebut.

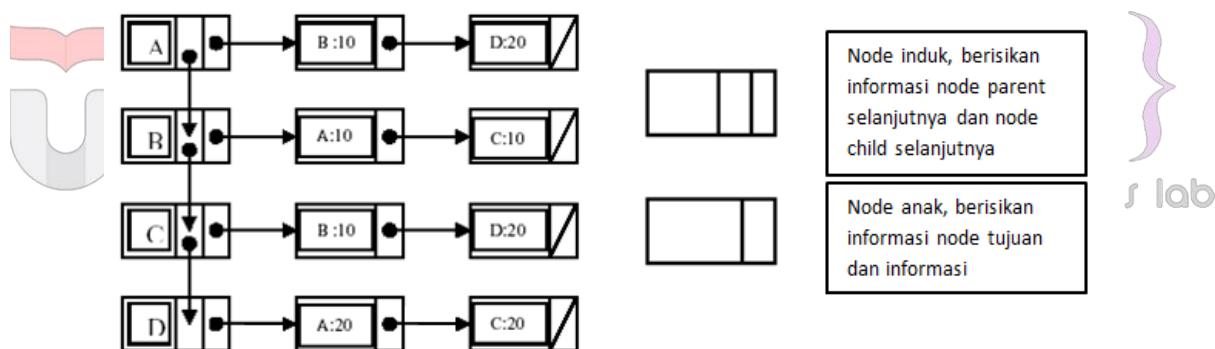
a. Array 2 Dimensi

	A	B	C	D
A	-	1	0	1
B	1	-	1	0
C	0	1	-	1
D	1	0	1	-

Keterangan :
1 bertetangga
0 tidak bertetangga

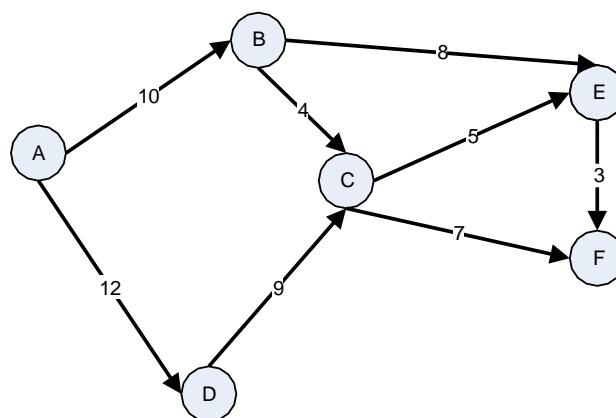
Gambar 14-9 Representasi *Graph* Array 2 Dimensi

b. Multi Linked List



Gambar 14-10 Representasi *Graph* Multi Linked list

Dalam praktikum ini untuk merepresentasikan *graph* akan menggunakan *multi list*. Karena sifat *list* yang dinamis, sehingga data yang bisa ditangani bersifat dinamis. Contoh ada sebuah *graph* yang menggambarkan jarak antar kota:



Gambar 14-11 *Graph* Jarak Antar kota

Gambar *multilist*-nya sama dengan gambar di atas.

Representasi struktur data *graph* pada multilist:

```
1 #ifndef GRAPH_H_INCLUDE
2 #define GRAPH_H_INCLUDE
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <conio.h>
6
7 typedef int infoGraph;
8 typedef struct ElmNode *adrNode;
9 typedef struct ElmEdge *adrEdge;
10
11 struct ElmNode{
12     infoGraph info;
13     int Visited;
14     adrEdge firstEdge;
15     adrNode Next;
16 };
17 struct ElmEdge{
18     adrNode Node;
19     adrEdge Next;
20 };
21 struct Graph {
22     adrNode First;
23 };
```

Berikut adalah contoh fungsi tambah *node* (*addNode*) dan prosedur tambah *edge* (*addEdge*):

```
1 // Adds Node
2 ElmNode addNode (infoGraph a, int b, adrEdge c, adrNode d) {
3     ElmNode newNode;
4     newNode.info = a;
5     newNode.Visited = b;
6     newNode.firstEdge = c;
7     newNode.Next = d;
8     return newNode;
9 }
10
11 // Adds an edge to a graph
12 void addEdge (ElmNode newNode) {
13     ElmEdge newEdge ;
14     newEdge.Node = newNode.Next;
15     newEdge.Next = newNode.firstEdge;
16 }
```

Program 3 Add *newNode* dan *newEdge*

Karena representasinya menggunakan *multilist* maka primitif-primitif *graph* sama dengan primitif-primitif pada *multilist*. Jadi untuk membuat ADT *graph* bisa memanfaatkan ADT yang sudah dibuat pada *multilist*.

B. Metode-Metode Penulusuran *Graph*

a. Breadth First Search (BFS)

Cara kerja algoritma ini adalah dengan mengunjungi *root* (*depth 0*) kemudian ke *depth 1, 2, dan seterusnya*. Kunjungan pada masing-masing *level* dimulai dari kiri ke kanan.

Secara umum, Algoritma BFS pada *graph* adalah sebagai berikut:

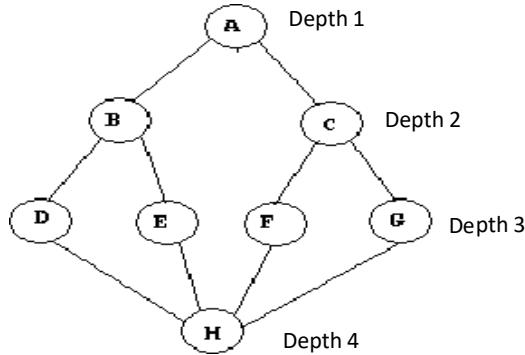
```
Prosedur BFS ( g : graph, start : node )
Kamus
    Q : Queue
    x, w : node
Algoritma
```

```

enqueue ( Q, start )
while ( not isEmpty( Q ) ) do
    x ← dequeue ( Q )
    if ( isvisited( x ) = false ) then
        isvisited( x ) ← true
        output ( x )
        for each node w ∈ Vx
            if ( isvisited( w ) = false ) then
                enqueue( Q, w )

```

Perhatikan *graph* berikut :



Gambar 14-12 Graph Breadth First Search (BFS)

Urutannya hasil penelusuran BFS : A B C D E F G H

b. Depth First Search (DFS)

Cara kerja algoritma ini adalah dengan mengunjungi *root*, kemudian rekursif ke *subtree node* tersebut.

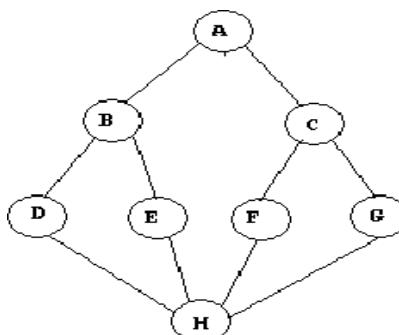
Secara umum, Algoritma DFS pada *graph* adalah sebagai berikut:

```

Prosedur BFS ( g : graph, start : node )
Kamus
    S : Stack
    x, w : node
Algoritma
    push ( S, start )
    while ( not isEmpty( S ) ) do
        x ← pop ( S )
        if ( isvisited( x ) = false ) then
            isvisited( x ) ← true
            output ( x )
            for each node w ∈ Vx
                if ( isvisited( w ) = false ) then
                    push ( S, w )

```

Perhatikan *graph* berikut :



Gambar 14-13 Graph Depth First Search (DFS)

Urutannya : A B D H E F C G

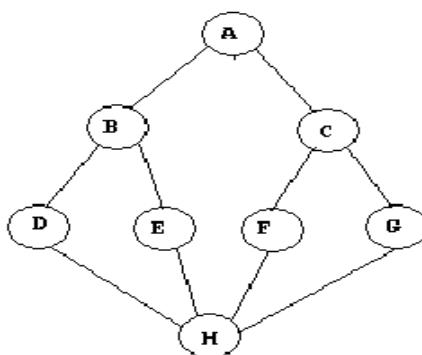
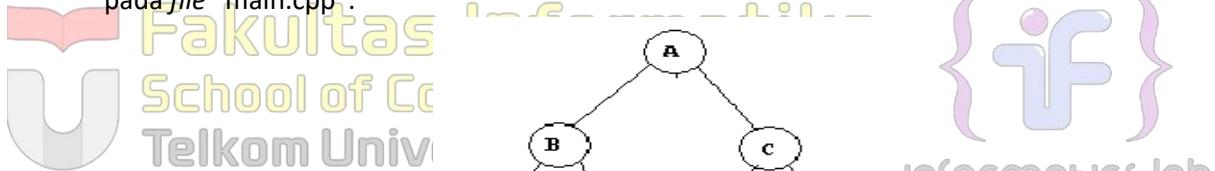
14.3 Latihan

- Buatlah ADT *Graph* tidak berarah file “graph.h”:

```
Type infoGraph: char
Type adrNode : pointer to ElmNode
Type adrEdge : pointer to ElmNode
Type ElmNode <
    info : infoGraph
    visited : integer
    firstEdge : adrEdge
    Next : adrNode
>
Type ElmEdge <
    Node : adrNode
    Next : adrEdge
>
Type Graph <
    first : adrNode
>
prosedur CreateGraph (Graph &G);
prosedur InsertNode (Graph &G, infotype x);
prosedur ConnectNode (adrNode N1, adrNode N2);
prosedur PrintInfoGraph (Graph G);
```

Program 4 Graph.h

Buatlah implementasi ADT *Graph* pada file “graph.cpp” dan cobalah hasil implementasi ADT pada file “main.cpp”.



Gambar 14-14 Ilustrasi Graph

- Buatlah prosedur untuk menampilkan hasil penelusuran DFS.
prosedur PrintDFS (Graph G, adrNode N);
- Buatlah prosedur untuk menampilkan hasil penelusuran BFS.
prosedur PrintBFS (Graph G, adrNode N);

Modul 15 ASSESSMENT CLO 2 (STACK & QUEUE)

TUJUAN PRAKTIKUM

1. Mengevaluasi pemahaman materi *Stack* dan *Queue*.

Pada modul 15, praktikan akan dievaluasi pemahamannya terhadap materi *Stack* dan *Queue*. Praktikan diminta untuk menyelesaikan soal-soal yang diberikan dalam bentuk *Assessment*.



DAFTAR PUSTAKA

- [1] Tim Dosen MK ASD. 2010 .*Modul Algoritma Struktur Data*. Departemen Teknik Informatika. Institut Teknologi Telkom, Bandung.
- [2] Tim Dosen MK ASD. 2011 .*Modul Algoritma Struktur Data*. Departemen Teknik Informatika. Institut Teknologi Telkom, Bandung.
- [3] Tim Dosen MK ASD. 2012 .*Modul Algoritma Struktur Data*. Fakultas Informatika. Institut Teknologi Telkom, Bandung.
- [4] Kernighan, Brian W., Ritchie, Dennis M. 1988. *C Programming Language. Second Ed.* Prentice Hall.
- [5] Liem, Ingriani. 2003. *Diktat Kuliah IF2181 Struktur Data*. Institut Teknologi Bandung, Bandung.
- [6] Standish, Thomas A. 1995. *Data structures, Algorithms, & Software Principles in C*. Addison Wesley Publishing Company.
- [7] Wirth, Niklaus. 1996. *Algorithm + data structure = program*. Prentice Hall.





Kontak Kami :

-  @dky2921g
-  @informaticslab
-  @informaticslab_telu
-  informaticslab@telkomuniversity.ac.id
-  informatics.labs.telkomuniversity.ac.id