

BUKU AJAR

STRUKTUR DATA

Tim Dosen Struktur Data
Prodi Informatika Fakultas Ilmu Komputer

Daftar Isi

Daftar Isi.....	i
Daftar Gambar	vi
Daftar Tabel	x
Pengantar	2
1.1 Pengertian Struktur Data.....	3
1.2 Penerapan struktur data.....	4
1.3 Klasifikasi struktur data	4
1.3.1 Linear Vs Non-Linear	4
1.3.2 Jenis Struktur Data.....	5
1.4 Manfaat mempelajari struktur data.....	6
1.5 Studi Kasus: Statistik Sederhana	7
1.6 Latihan	8
Array	9
2.1 Pengertian.....	10
2.2 Array 1 Dimensi	11
2.3 Array Multidimensi	14
2.4 Manipulasi	17
2.5 Studi Kasus	18
2.6 Latihan	20
Pointer.....	21
3.1 Pengertian.....	22
3.1.1 Deklarasi.....	23
3.1.2 Operator.....	23
3.2 Penggunaan	25
3.2.1 Aritmatika Pointer	25
3.2.2 Array dan Pointer	26
2.2.3 Fungsi dan Pointer	27
3.3 Reference	30
3.4 Ukuran Data	31
3.5 Alokasi Dinamis	34
3.5.1 <i>Wild Pointer</i>	34

3.5.2 Alokasi dan Dealokasi.....	35
3.6 Studi Kasus	38
3.7 Latihan	39
Pencarian Data	40
4.1 Pengertian.....	41
4.2 Metode	41
4.2.1 <i>Sequential Search</i>	41
4.2.2 <i>Binary Search</i>	43
4.3 Standard Library <code>std::find()</code>	46
4.4 Studi Kasus	47
4.5 Latihan	48
Pengurutan Data.....	49
5.1 Pengertian.....	50
5.2 Metode Pencarian Dasar	50
5.2.1 <i>Bubble Sort</i>	51
5.2.2 <i>Selection Sort</i>	55
5.2.3 <i>Insertion Sort</i>	59
5.3 Metode Pengurutan Data Tingkat Lanjut	63
5.3.1 <i>Shell Sort</i>	63
5.3.2 <i>Merge Sort</i>	65
5.3.3 <i>Quick Sort</i>	65
5.4 Standard Library <code>std::sort()</code>	66
5.5 Studi Kasus	67
5.6 Latihan	70
Struct	71
6.1 Pengertian.....	72
6.1.1 Struct	72
6.1.2 Class.....	72
6.2 Deklarasi	73
6.3 Pengaksesan	74
6.3.1 Pengaksesan Struct Bersarang.....	74
6.4 <i>Array of Struct</i>	75
6.4.1 Deklarasi.....	75
6.4.2 Pengaksesan	76

6.5 Studi Kasus	77
6.6 Latihan	79
Stack	81
7.1 Pengertian.....	82
7.2 Operasi	82
7.2.1 Push	84
7.2.2 Pop	84
7.2.3 Clear	84
7.2.4 Show.....	85
7.2.5 Simulasi.....	85
7.3 <i>Double Stack</i>	87
7.3.1 Operasi	87
7.3.2 Simulasi.....	90
7.4 Penerapan.....	91
7.5 Studi Kasus	93
7.6 Latihan	94
Queue.....	96
8.1 Pengertian.....	97
8.2 Operasi	98
8.2.1 Store	98
8.2.2 Retrieve.....	99
8.2.3 Clear	99
8.2.4 Show.....	99
8.2.5 Simulasi.....	100
8.3 <i>Circular Queue</i>	102
8.3.1 Operasi.....	103
8.3.2 Simulasi.....	105
8.4 Penerapan Queue	107
8.5 Studi Kasus	107
8.6 Latihan	110
Linked List.....	111
9.1 Pengertian.....	112
9.2 Operasi	116
9.2.1 Penambahan	117

9.2.2 Penghapusan.....	123
9.2.4 Pembacaan.....	128
9.2.5 Simulasi.....	128
9.3 Stack dan Queue dengan Linked List	130
9.3.1. Stack dengan Singly Linked List.....	130
9.3.2. Queue dengan Doubly Linked List.....	131
9.3.3 Queue dengan Linked List VS Queue dengan Array.....	132
9.4 Studi Kasus	133
9.4 Latihan	138
Graph	139
10.1 Pengertian.....	140
10.1.1 Jenis-jenis Graph	141
10.1.2 Representasi Graph	143
10.1.3 Mengambil <i>graph</i> sebagai input dari keyboard dan menyimpannya ke dalam memori.....	153
10.2 Operasi	154
10.3 Istilah	154
10.4 Algoritma Traversal Graph	155
10.4.1 <i>Breadth-First Search</i> (BFS)	155
10.5 Studi Kasus.....	160
10.6 Latihan	161
Tree	163
11.1 Pengertian.....	164
11.2 Pohon Biner (Binary Tree)	166
11.2.1 Terminologi Pohon Berakar	169
11.2.2 Jenis-jenis <i>Binary Tree</i>	171
11.3 Representasi Tree	172
11.3.1 Implementasi program tree	177
11.4 <i>Traversal</i>	178
11.5 Operasi-operasi Tree.....	180
11.5.1 Insert Node.....	181
11.5.2 Hapus Node.....	181
11.6 <i>Binary Search Tree</i> (BST).....	184
11.6.1 Konversi <i>Tree</i> biasa ke <i>Binary Tree</i>	184

11.6.2 Ilustrasi Searching dengan BST.....	185
11.7 Studi Kasus.....	188
11.8 Latihan	192
Daftar Pustaka.....	194

Daftar Gambar

Gambar 1. 1 Klasifikasi Struktur Data	5
Gambar 2. 1 Ilustrasi Array 1 Dimensi	11
Gambar 2. 2 Inisialisasi Array.....	12
Gambar 2. 3 Ilustrasi pengisian slot array umur di memori	13
Gambar 2. 4 Ilustrasi Array 2 Dimensi	14
Gambar 2. 5 Ilustrasi pengalokasian array multidimensi di memori	15
Gambar 3. 1 Alokasi memori untuk variabel.....	22
Gambar 3. 2 Pointer	22
Gambar 3. 3 Ilustrasi pointer dalam memori	23
Gambar 3. 4 Pemetaan Memori pada Variabel dan Pointer	24
Gambar 3. 5 Pesan Kesalahan Wild Pointer	35
Gambar 4. 1 Langkah 1 Ilustrasi Sequential Search	42
Gambar 4. 2 Langkah 2 Ilustrasi Sequential Search	43
Gambar 4. 3 Langkah 3 Ilustrasi Sequential Search	43
Gambar 4. 4 Langkah 4 Ilustrasi Sequential Search	43
Gambar 4. 5 Langkah 1 Ilustrasi Binary Search	45
Gambar 4. 6 Langkah 2 Ilustrasi Binary Search	45
Gambar 4. 7 Langkah 3 Ilustrasi Binary Search	45
Gambar 5. 1 Pengurutan Data	50
Gambar 5. 2 Kondisi awal (array blm urut).....	51
Gambar 5. 3 Langkah 1 Pengurutan Bubble Sort.....	51
Gambar 5. 4 Langkah 2 Pengurutan Bubble Sort.....	52
Gambar 5. 5 Langkah 3 Pengurutan Bubble Sort.....	52
Gambar 5. 6 Langkah 4 Pengurutan Bubble Sort.....	52
Gambar 5. 7 Langkah 5 Pengurutan Bubble Sort.....	53
Gambar 5. 8 Hasil pengurutan dengan Bubble Sort.....	53
Gambar 5. 9 Rangkuman proses pengurutan data dengan Bubble Sort.....	53
Gambar 5. 10 Kondisi awal (data belum urut)	55
Gambar 5. 11 Langkah 1 Pengurutan Selection Sort.....	55
Gambar 5. 12 Langkah 2 Pengurutan Selection Sort.....	56
Gambar 5. 13 Langkah 3 Pengurutan Selection Sort.....	56
Gambar 5. 14 Langkah 4 Pengurutan Selection Sort.....	57
Gambar 5. 15 Langkah 5 Pengurutan Selection Sort.....	57
Gambar 5. 16 Hasil pengurutan Selection Sort.....	57
Gambar 5. 17 rangkuman proses pengurutan data dengan Selection Sort	58
Gambar 5. 18 Kondisi awal (data belum urut)	59
Gambar 5. 19 Langkah 1 Pengurutan Insertion Sort.....	59
Gambar 5. 20 Langkah 2 Pengurutan Insertion Sort	60
Gambar 5. 21 Langkah 3 Pengurutan Insertion Sort	60
Gambar 5. 22 Langkah 4 Pengurutan Insertion Sort	60
Gambar 5. 23 Langkah 5 Pengurutan Insertion Sort	61
Gambar 5. 24 Hasil pengurutan Selection Sort.....	61

Gambar 5. 25 Proses pengurutan data dengan Insertion Sort	61
Gambar 5. 26 Pembandingan dengan gap sebesar 4	64
Gambar 5. 27 Hasil akhir setelah pembandingan dengan gap sebesar 1	64
Gambar 5. 28 Ilustrasi Merge sort.....	65
Gambar 5. 29 Ilustrasi prosedur pemartisian	66
Gambar 7. 1 Ilustrasi stack	83
Gambar 7. 2 Ilustrasi Stack Kondisi Awal.....	85
Gambar 7. 3 Langkah 1 Ilustrasi Stack	85
Gambar 7. 4 Langkah 2 Ilustrasi Stack	86
Gambar 7. 5 Langkah 3 Ilustrasi Stack	86
Gambar 7. 6 Langkah 4 Ilustrasi Stack	86
Gambar 7. 7 Langkah 5 Ilustrasi Stack	86
Gambar 7. 8 Langkah 6 Ilustrasi Stack	87
Gambar 7. 9 Contoh Double Stack.....	87
Gambar 7. 10 Langkah 1 Ilustrasi Double Stack.....	90
Gambar 7. 11 Langkah 2 Ilustrasi Double Stack.....	90
Gambar 7. 12 Langkah 3 Ilustrasi Double Stack.....	90
Gambar 7. 13 Langkah 4 Ilustrasi Double Stack.....	90
Gambar 7. 14 Langkah 5 Ilustrasi Double Stack.....	91
Gambar 7. 15 Langkah 6 Ilustrasi Double Stack.....	91
Gambar 7. 16 Tumpukan untuk membalikan string	91
Gambar 8. 1 Ilustrasi perbedaan stack dan queue	97
Gambar 8. 2 Ilustrasi queue	98
Gambar 8. 3 Ilustrasi Queue Kondisi Awal	100
Gambar 8. 4 Langkah 1 Ilustrasi Queue	100
Gambar 8. 5 Langkah 2 Ilustrasi Queue	100
Gambar 8. 6 Langkah 3 Ilustrasi Queue	100
Gambar 8. 7 Langkah 4 Ilustrasi Queue	101
Gambar 8. 8 Langkah 5 Ilustrasi Queue	101
Gambar 8. 9 Langkah 6 Ilustrasi Queue	101
Gambar 8. 10 Langkah 7 Ilustrasi Queue	101
Gambar 8. 11 Contoh Circular Queue	102
Gambar 8. 12 Langkah 1 Ilustrasi Circular Queue.....	105
Gambar 8. 13 Langkah 2 Ilustrasi Circular Queue.....	105
Gambar 8. 14 Langkah 3 Ilustrasi Circular Queue.....	106
Gambar 8. 15 Langkah 4 Ilustrasi Circular Queue.....	106
Gambar 8. 16 Langkah 5 Ilustrasi Circular Queue.....	106
Gambar 8. 17 Langkah 6 Ilustrasi Circular Queue.....	106
Gambar 8. 18 Langkah 7 Ilustrasi Circular Queue.....	107
Gambar 9. 1 Ilustrasi linked list.....	112
Gambar 9. 2 Ilustrasi singly linked list non circular.....	113
Gambar 9. 3 Ilustrasi doubly linked list	114
Gambar 9. 4 Ilustrasi singly linked list circular	115
Gambar 9. 5 Ilustrasi doubly linked list circular	115
Gambar 9. 6 Penyisipan depan dengan linked list belum ada	118
Gambar 9. 7 Penyisipan simpul depan	118

Gambar 9. 8 Penyisipan simpul belakang.....	120
Gambar 9. 9 Penyisipan simpul sebelum simpul tertentu	122
Gambar 9. 10 Penghapusan simpul depan	124
Gambar 9. 11 Penghapusan simpul belakang.....	125
Gambar 9. 12 Penghapusan simpul tertentu di tengah	127
Gambar 9. 13 Langkah 1 Penambahan data linked list	128
Gambar 9. 14 Langkah 2 Penambahan data linked list	128
Gambar 9. 15 Langkah 3.1 Penambahan data linked list	129
Gambar 9. 16 Langkah 3.2 Penambahan data linked list	129
Gambar 9. 17 Langkah 4.1 Penambahan data linked list	129
Gambar 9. 18 Langkah 4.2 Penambahan data linked list	130
Gambar 9. 19 Hasil penambahan data linked list	130
Gambar 10. 1 Contoh graph	140
Gambar 10. 2 (a) Graph Connected (b) Disconnected (c) Complete	141
Gambar 10. 3 Graph tidak terarah.....	141
Gambar 10. 4 Graph berarah.....	142
Gambar 10. 5 (a) Graph Weakly Connected (b) Strongly Connected	142
Gambar 10. 6 Graph berbobot	143
Gambar 10. 7 Contoh simulasi graph tak berarah	146
Gambar 10. 8 Langkah 1 Pengisian Adjacency Matrix Graph Tak Berarah	146
Gambar 10. 9 Langkah 2 Pengisian Adjacency Matrix Graph Tak Berarah	147
Gambar 10. 10 Langkah 3 Pengisian Adjacency Matrix Graph Tak Berarah	147
Gambar 10. 11 Langkah 4 Pengisian Adjacency Matrix Graph Tak Berarah	148
Gambar 10. 12 Langkah 5 Pengisian Adjacency Matrix Graph Tak Berarah	148
Gambar 10. 13 Contoh Simulasi Graph Berarah	149
Gambar 10. 14 Langkah 1 Pengisian Adjacency Matrix Graph Berarah	149
Gambar 10. 15 Langkah 2 Pengisian Adjacency Matrix Graph Berarah	150
Gambar 10. 16 Langkah 3 Pengisian Adjacency Matrix Graph Berarah	150
Gambar 10. 17 Langkah 4 Pengisian Adjacency Matrix Graph Berarah	151
Gambar 10. 18 Langkah 5 Pengisian Adjacency Matrix Graph Berarah	151
Gambar 10. 19 Representasi Adjacency Matrix Graph Berarah dan Berbobot	152
Gambar 10. 20 Representasi Graph pada Linked List	152
Gambar 10. 21 Contoh Representasi Graph Tak Berarah pada Linked List.....	153
Gambar 10. 22 Graph tertutup	154
Gambar 10. 23 Contoh Graph pada Ilustrasi Algoritma Traversal.....	155
Gambar 10. 24 Langkah 1 Algoritma Traversal Graph: BFS	156
Gambar 10. 25 Langkah 2 Algoritma Traversal Graph: BFS	156
Gambar 10. 26 Langkah 3 Algoritma Traversal Graph: BFS	156
Gambar 10. 27 Langkah 4 Algoritma Traversal Graph: BFS	156
Gambar 10. 28 Langkah 5 Algoritma Traversal Graph: BFS	157
Gambar 10. 29 Langkah 6 Algoritma Traversal Graph: BFS	157
Gambar 10. 30 Langkah 7 Algoritma Traversal Graph: BFS	157
Gambar 10. 31 Langkah 8 Algoritma Traversal Graph: BFS	157
Gambar 10. 32 Langkah 1 Algoritma Traversal Graph: DFS	158
Gambar 10. 33 Langkah 2 Algoritma Traversal Graph: DFS	158
Gambar 10. 34 Langkah 3 Algoritma Traversal Graph: DFS	159

Gambar 10. 35 Langkah 4 Algoritma Traversal Graph: DFS	159
Gambar 10. 36 Langkah 5 Algoritma Traversal Graph: DFS	159
Gambar 10. 37 Langkah 6 Algoritma Traversal Graph: DFS	159
Gambar 10. 38 Langkah 7 Algoritma Traversal Graph: DFS	160
Gambar 10. 39 Langkah 8 Algoritma Traversal Graph: DFS	160
Gambar 10. 40 Ilustrasi proses insert	181
Gambar 10. 41 Menghapus Node Root 25.....	182
Gambar 10. 42 Mencari pengganti Node Root dengan nilai yang tepat	182
Gambar 10. 43 Menghapus node yang memiliki 1 anak	183
Gambar 10. 44 Menghapus Node Leaf.....	183
Gambar 10. 45 Ilustrasi proses penghapusan node dengan 2 anak	184
Gambar 10. 46 Langkah 1 proses pencarian BST	186
Gambar 10. 47 Langkah 2 proses pencarian BST	187
Gambar 10. 48 Langkah 3 proses pencarian BST	187
Gambar 11. 1 Pohon berakar.....	165
Gambar 11. 2 Pohon biner dengan 11 simpul (node)	167
Gambar 11. 3 Dua pohon biner yang similar	168
Gambar 11. 4 Dua pohon biner yang disebut copies	168
Gambar 11. 5 Terminologi Pohon Berakar	170
Gambar 11. 6 Complete Binary Tree.....	171
Gambar 11. 7 Pohon condong kanan (a) dan condong kiri (b).....	171
Gambar 11. 8 Sebuah pohon beserta tingkatannya (level)	172
Gambar 11. 9 Diagram Venn	172
Gambar 11. 10 Notasi Tingkat	173
Gambar 11. 11 Representasi pohon biner.....	174
Gambar 11. 12 Penggambaran skema penyajian kait pada pohon biner	175
Gambar 11. 13 Skema memori penyajian kait dari pohon biner	175
Gambar 11. 14 Contoh gambaran pohon biner pada records.....	176
Gambar 11. 15 Skema dari Gambar 11.8	176
Gambar 11. 16 Proses Traversal	179
Gambar 11. 17 Contoh Ilustrasi traversal pada ekspresi matematika	180
Gambar 11. 18 Konversi menjadi pohon biner	185
Gambar 11. 19 Ilustrasi Searching BST.....	185

Daftar Tabel

Tabel 1. 1 Kelebihan dan Kekurangan macam-macam struktur data.....	6
Tabel 5. 1 Nilai gap pada shell sort.....	63
Tabel 11. 1 Perbedaan Graph dan Tree.....	164

1

Pengantar

Sub CPMK

Mahasiswa mampu menjelaskan kembali teori dan konsep dasar struktur data serta mampu menerapkan contoh pengelolaan data dalam program (CPMK35)

Bab ini akan membahas tentang Pengantar Struktur Data. Pembahasan dimulai dari penjelasan singkat tentang struktur data dan kaitannya dengan algoritma. Kemudian dilanjutkan dengan bahasan mengenai gambaran penggunaan struktur data dan klasifikasi struktur data. Terdapat contoh studi kasus untuk membuat program statistik sederhana untuk mencari bilangan terbesar dari 3 buah bilangan, menentukan ganjil genap dan luas persegi. Terakhir akan diberikan beberapa soal latihan.

1.1 Pengertian Struktur Data

Struktur data adalah representasi dari hubungan logis yang ada antara elemen individu data. Struktur data mendefinisikan cara mengatur semua item data yang mempertimbangkan tidak hanya elemen yang disimpan tetapi juga hubungan antara satu sama lain. Data disimpan supaya bisa diakses atau diproses setelahnya. Penyimpanan data diperlukan sebuah struktur supaya lebih mudah dan efisien dalam pengaksesan dan pemrosesan data tersebut. Contoh dari struktur data adalah array, pointer, struct, tumpukan dan pohon biner.

Algoritma adalah metode sistematis yang menggambarkan langkah atau proses yang ditujukan untuk memanipulasi data dalam memecahkan berbagai jenis masalah. Sebagai contoh, algoritma diperlukan untuk memasukkan data kedalam struktur data atau untuk mencari suatu data yang tersimpan dalam struktur data. Abdul Kadir menggambarkan hubungan antara struktur data dengan algoritma seperti sebuah wadah data sebagai struktur datanya, sedangkan algoritma adalah manipulasi datanya [1]. Oleh karena itu, program merupakan representasi dari algoritma yang dikombinasikan dengan struktur data.

Komponen dari pemrograman terstruktur antara lain:

1. Sekuensial / urutan (*Sequence*)
Eksekusi statement atau instruksi secara teratur.
2. Seleksi (*Selection*)
Eksekusi salah satu statement bergantung pada kondisi tertentu.
3. Pengulangan (*Repetition*)
Eksekusi sebuah statement hingga mencapai kondisi tertentu.

Struktur pertama, sekuensial merupakan langkah atau proses yang berurutan. Langkah yang berurutan menentukan urutan eksekusi program. Contohnya pada penukaran nilai A dan nilai B.

```
X = A;  
A = B;  
B = X;
```

Berdasarkan algoritma diatas, dapat dilihat urutan langkah sebagai berikut :

1. X diberi inputan nilai A
2. A diberi inputan nilai B
3. B diberi inputan nilai X

Apabila urutan program diubah maka tidak ada proses penukaran antara nilai A dan nilai B.

Struktur kedua, struktur seleksi mempunyai manfaat untuk memilih Tindakan berdasar pada kondisi. Misalkan, jika nilai A dan nilai B akan ditukar hanya jika

memenuhi syarat A lebih besar dari pada B ($A > B$). Dalam C++ pernyataan seleksi menggunakan pernyataan IF, pernyataan dapat dituliskan :

```
if (a>b)
{
    x = a;
    a = b;
    b = x;
}
```

Proses pertukaran hanya dieksekusi jika syarat/kondisi yang ditentukan terpenuhi. Apabila terdapat dua jenis Tindakan berbeda yang akan diproses maka struktur seleksi dapat menggunakan ELSE.

1.2 Penerapan struktur data

Struktur data pada penerapannya dikehidupan sehari-hari dapat kita ilustrasikan dengan kartu belanja. Kini data transaksi pembelanjaan dari pelanggan sudah menjadi data penting yang harus dimiliki oleh supermarket untuk mengenali pola belanja dari pelanggannya. Setiap data pelanggan akan disimpan dalam suatu kartu.

Saat dilakukan pencarian maka nama pelanggan akan digunakan sebagai key atau acuan, sehingga huruf awal nama customer akan digunakan. Data pelanggan akan disusun berdasarkan abjad, contohnya: Ana, Ami, Asti, Budi, Dani, Dika, Ika. Nama yang mempunyai huruf awalan abjad sama akan disimpan dalam satu tempat yang sama. Dengan cara tersebut akan memudahkan pencarian kartu anggota daripada pencarian dilakukan pada data acak yang tidak diurutkan.

1.3 Klasifikasi struktur data

1.3.1 Linear Vs Non-Linear

Suatu struktur data dikatakan linier jika elemen-elemennya membentuk barisan atau daftar linier. Struktur data linier contohnya seperti larik, tumpukan, antrian, dan daftar tertaut mengatur data dalam urutan linier.

Suatu struktur data dikatakan tidak linier jika elemen-elemennya membentuk klasifikasi hierarkis dimana item data muncul pada berbagai level. Pohon dan Grafik adalah struktur data non-linier yang banyak digunakan. Struktur pohon dan grafik yang mewakili hubungan hierarkis antara elemen data individual. Grafik tidak lain adalah pohon dengan batasan tertentu yang dihilangkan.

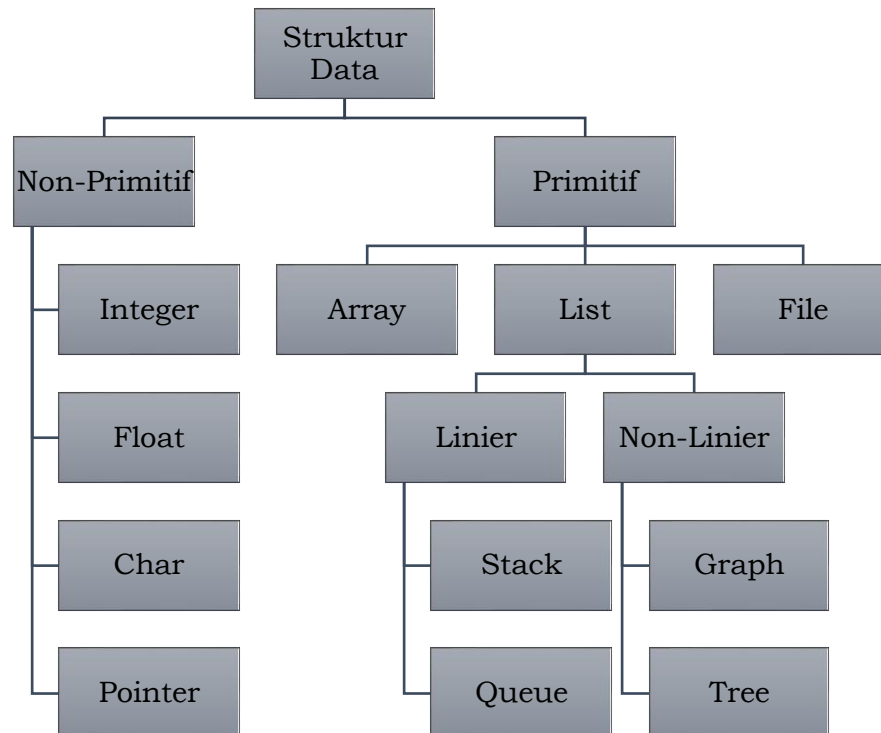
1.3.2 Jenis Struktur Data

Struktur data dibagi menjadi dua jenis:

- 1) Struktur data primitif.
- 2) Struktur data non-primitif.

Struktur data primitif adalah struktur data dasar yang disediakan oleh bahasa pemrograman atau langsung beroperasi pada instruksi mesin. Mereka memiliki representasi yang berbeda pada komputer yang berbeda atau bahasa pemrograman. Bilangan bulat, angka floating point, konstanta karakter, konstanta string, dan pointer termasuk dalam kategori ini.

Struktur data non-primitif adalah struktur data yang lebih rumit dan diturunkan dari struktur data primitif. Mereka menekankan pada pengelompokan item data yang sama atau berbeda dengan hubungan antara setiap item data. Array, daftar dan file termasuk dalam kategori ini. Gambaran dari struktur data ini dapat dilihat pada Gambar 1.1.



Gambar 1. 1 Klasifikasi Struktur Data

Masing-masing struktur data memiliki kelebihan dan kekurangan, hal tersebut dapat digambarkan pada Tabel 1.1.

Tabel 1. 1 Kelebihan dan Kekurangan macam-macam struktur data

Struktur Data	Kelebihan	Kekurangan
<i>Array</i>	Tidak Terurut: Penambahan data dibelakang mudah dilakukan Terurut: Pencarian lebih cepat	Tidak Terurut: Ukuran tetap, penghapusan lambat, pencarian lama Terurut: Ukuran tetap, penghapusan lambat, penyisipan lama.
<i>Tumpukan (Stack)</i>	Penambahan cepat Akses data terakhir masuk cepat	Pencarian dan penghapusan lambat
<i>Antrian (Queue)</i>	Data yang pertama kali masuk mudah diakses	Akses data lainnya lambat
<i>Linked List</i>	Penyisipan dan penghapusan mudah	Pencarian lama
<i>Binary Tree</i>	Pencarian dan penyisipan mudah	Penghapusan kompleks

1.4 Manfaat mempelajari struktur data

Dari penjelasan awal sudah dijelaskan struktur data adalah pengaturan data di dalam memori komputer atau terkadang di dalam disk dengan tujuan agar data dapat diakses secara efisien. Bagaimana mengetahui sudah efisien atau belum? Efisien ini dapat diukur darimana?

Efisiensi sebuah algoritma dapat diukur berdasarkan waktu pemrosesan ataupun penggunaan memori. Suatu algoritma dikatakan sangat efisien dalam hal waktu kalau waktu yang digunakan untuk melaksanakan eksekusi sebuah program lebih cepat daripada algoritma lainnya. Sedangkan untuk efisien dalam penggunaan memori jika pemilihan struktur data tepat sesuai kebutuhan sehingga proses eksekusi sebuah program dapat menghemat penggunaan memori atau lebih hemat memori daripada algoritma lainnya.

Kompleksitas suatu algoritma merupakan ukuran seberapa banyak komputasi yang dibutuhkan algoritma tersebut untuk mendapatkan hasil yang diinginkan. Hal-hal yang mempengaruhi kompleksitas waktu:

1. Jumlah masukan data untuk suatu algoritma (n).
2. Waktu yang dibutuhkan untuk menjalankan algoritma tersebut.
3. Ruang memori yang dibutuhkan untuk menjalankan algoritma yang berkaitan dengan struktur data dari program.

Kompleksitas mempengaruhi performa atau kinerja dari suatu algoritma. Kompleksitas dibagi menjadi 3 jenis yaitu, *worst case*, *best case*, dan *average case*. Masing-masing jenis kompleksitas ini menunjukkan kecepatan atau waktu yang dibutuhkan algoritma untuk mengeksekusi sejumlah kode [2].

1.5 Studi Kasus: Statistik Sederhana

Program untuk mencari bilangan terbesar dari 3 buah bilangan.

Ketentuan:

- 1) Data disimpan dalam variabel A, B dan C
- 2) Data dapat diinputkan secara dinamis saat program berjalan.

Penyelesaian :

```
#include<iostream>
using namespace std;
int main()
{
    int i=0, a, b, c;
    cout << "Program Mencari Bilangan Terbesar dari 3 buah bilangan"
    << endl;
    while(i < 1){
        cout << "masukkan nilai A: "; cin >> a;
        cout << "masukkan nilai B: "; cin >> b;
        cout << "masukkan nilai C: "; cin >> c;
        if(a > b && a > c){
            cout << "bilangan terbesar adalah "<< a << endl;
        }else if(b > a && b > c){
            cout << "bilangan terbesar adalah "<< b << endl;
        }else if(c > a && c > b){
            cout << "bilangan terbesar adalah "<< c << endl;
        }
        cout << '\n';
        string x;
        cout << "Coba Lagi (y/n)? "; cin >> x;
        if (x == "y"){ b=0; }
        else if(x == "n"){ i=1; }
        cout << '\n';
    }
}
```

1.6 Latihan

Buatlah program sederhana untuk:

1. Mencari bilangan terkecil dan rata-rata dari 3 buah bilangan.
2. Menentukan bilangan ganjil dan genap
3. Menghitung luas persegi
4. Menghitung akar-akar persamaan kuadrat
5. Mencari nilai Faktor Persekutuan Terbesar (FPB)

2

Array

Sub CPMK

1. Mahasiswa mampu menjelaskan kembali konsep, pengertian dan prinsip dasar Array
2. Mahasiswa mampu merancang program yang menerapkan struktur pemrograman dan prinsip dasar Array dalam bahasa pemrograman C++
3. Mahasiswa mampu mengevaluasi permasalahan pemrograman Array dalam studi kasus.

Bab ini akan membahas tentang array atau larik. Pembahasan dimulai dari pengertian array, cara pendeklarasian, dan cara memanipulasinya. Pembahasan selanjutnya tentang macam-macam array yaitu array 1 dimensi dan array multidimensi. Terakhir adalah contoh studi kasus penggunaan array untuk membuat program statistik sederhana yang dapat melakukan perhitungan rata - rata, pencarian nilai terbesar dan terkecil.

2.1 Pengertian

Array merupakan fungsi variabel di dalam program digunakan untuk menyimpan suatu nilai tertentu yang dapat diubah-ubah. Tiap variabel mempunyai nama dan tipe. Hanya data yang mempunyai tipe sama dengan variabel yang dapat disimpan di dalam variabel tersebut.

Array atau sering disebut larik adalah sekumpulan elemen yang mempunyai tipe data sama dalam urutan tertentu dan menggunakan nama yang sama. Elemen-elemen array tersusun secara berderet dan dapat diakses secara random di dalam memori yang posisinya dapat diakses melalui indeks. Array memiliki alamat yang besebelahan/berdampingan tergantung lebar tipe datanya. Array sangat cocok jika digunakan pada kumpulan data homogen yang ukuran atau jumlah elemen maksimumnya telah diketahui dari awal.

Array dapat berupa array 1 dimensi, 2 dimensi, bahkan n-dimensi dapat dilihat berdasarkan penunjuk indeks [3]. Elemen-elemen array bertipe data sama dan bisa berisi nilai yang sama atau berbeda-beda. Array merupakan struktur data yang statis, yaitu jumlah elemen yang ada harus ditentukan terlebih dahulu dan tak bisa di ubah saat program berjalan.

Hal-hal yang perlu diperhatikan apabila memasukkan deretan data bertipe array sebagai berikut:

1. Mengetahui tipe data yang digunakan dalam variabel array. Variabel array numerik hanya dapat menerima data numerik dan variabel array string hanya dapat menerima data karakter. String merupakan kumpulan dari karakter, biasanya merupakan kumpulan dari huruf dalam alfabet. String didefinisikan sebagai kumpulan dari tipe data char. Dalam bahasa C string didefinisikan sebagai sekumpulan char yang diakhiri dengan null character (C-style string).
2. Banyaknya data harus lebih kecil atau sama dengan besarnya ukuran array.
3. Struktur perulangan lazim digunakan untuk membaca nilai – nilai dalam array.
4. Banyaknya indeks yang digunakan menunjukkan banyaknya ruang memori yang dialokasikan. Supaya tidak terjadi pemborosan ruang memori, maka banyaknya indeks harus disesuaikan dengan banyaknya data.

Penentuan dimensi array harus disesuaikan:

1. Model data dari kasus yang akan diselesaikan.
2. Model pengorganisasian data untuk kemudahan dalam pengolahan data.

2.2 Array 1 Dimensi

Disebut sebagai array 1 dimensi karena indeks hanya satu. Pendeklarasian variabel array satu dimensi sebenarnya hampir sama dengan pendeklarasian variabel yang lain. Hanya saja pendeklarasian array ini diikuti dengan maksimum elemen yang akan dialokasikan. Pendeklarasian ukuran array dituliskan dalam pasangan tanda kurung siku atau *square bracket* [..].

Pendeklarasian:

```
TipeData NamaVarArray[UkuranIndeks1];
```

dimana :

tipe data : menyatakan jenis tipe data elemen array (int, char, float, dll)
nama_var_array : menyatakan nama variabel yang dipakai
ukuran : menunjukkan maksimal jumlah elemen array

Contoh:

```
int ar[8];
```

Deklarasi diatas berarti bahwa `ar` adalah sebuah array bertipe int dengan ukuran 8. Ilustrasi visual deklarasi array diatas dapat dilihat pada Gambar 2.1.

Ilustrasi Array 1 Dimensi

0	1	2	3	4	5	6	7	indeks
								value
21da	21db	21dc	21dd	21de	21df	21e0	21e1	alamat

Gambar 2. 1 Ilustrasi Array 1 Dimensi

Saat dideklarasikan `ar` akan disediakan 8 area untuk menampung elemen. Tiap – tiap elemen memiliki index dari 0 sampai 7.

Catatan:

Index atau subscript pada array digunakan untuk menandai tiap – tiap nilai yang tersimpan di dalamnya. Satu nilai yang tersimpan memiliki satu index yang unik. Index array selalu dimulai dari 0 sehingga untuk array dengan panjang n akan memiliki index terakhir $n-1$.

Mengisi nilai atau value dari elemen array dapat dilakukan dengan 3 cara, yaitu pada saat variabel dideklarasikan (inisialisasi), menggunakan penugasan (=) dan dibaca dari media masukan saat program berjalan.

Inisialisasi:

Inisialisasi array merupakan langkah untuk memberikan nilai awal pada array saat dideklarasikan. Pemberian nilai awal tersebut dapat dilakukan dengan dua acara, yaitu:

1) Statis

pemberian nilai dilakukan serentak saat deklarasi. Berikut ini adalah contoh pemberian nilai secara statis:

```
int ar[] = {4, 8, 9, 2, 7};  
int ar[] {4, 8, 9, 2, 7}; // atau menggunakan direct  
// initialization list
```

Ukuran array secara implisit akan disesuaikan dengan jumlah elemen yang dimasukkan. Namun, bisa juga pemberian ukuran array dilakukan secara eksplisit seperti berikut ini:

```
int ar[5] = {4, 8, 9, 2, 7};  
int ar[5] {4, 8, 9, 2, 7}; // direct initialization list
```

Setiap elemen akan otomatis memiliki index sesuai urutan yang dimulai dari index 0. Pendeklarasian di atas bila dilakukan satu persatu adalah seperti berikut:

```
ar[0] = 4;  
ar[1] = 8;  
ar[2] = 9;  
ar[3] = 2;  
ar[4] = 7;
```

Ilustrasi visual dari contoh pemberian nilai (inisialisasi) di atas adalah seperti pada Gambar 2.2.

0	1	2	3	4
4	8	9	2	7

Gambar 2. 2 Inisialisasi Array

Contoh lain pendeklarasian array 1 dimensi:

```
char huruf[9];  
int umur[10] = {8,12,17,20,25,30,55};  
int kondisi[2] = {0,1}  
int arr_dinamis[] = {1,2,3}
```

- Tanda [] disebut juga “elemen yang ke- ...”. Misalnya kondisi[0] berarti array kondisi elemen yang ke nol.

- Contoh pada baris ke-2, misalnya array dengan nama umur dengan jumlah elemen 10. Artinya kita memesan 10 tempat di memori, tempat atau slot tersebut tidak harus diisi semuanya, bisa saja hanya diisi 7 elemen saja, baik secara berurutan maupun tidak. Ilustrasi dari pendeklarasian array umur digambarkan oleh Gambar 1.3.

0	1	2	3	4	5	6	7	8	9	indeks
8	12	17	20	25	30	55				value

Gambar 2. 3 Ilustrasi pengisian slot array umur di memori

Namun pada kondisi yang tidak sepenuhnya terisi tersebut, tempat pemesanan di memori tetap sebanyak 10 tempat, jadi tempat yang tidak terisi tetap akan terpesan dan dibiarkan kosong.

Kita **tidak dapat** mendeklarasikan array dinamis tanpa inisialisasi!

2) **Dinamis** (menggunakan struktur perulangan)

Pemberian nilai dilakukan setelah pendeklarasian yang biasanya berasal dari input pengguna. Berikut ini adalah contoh kode untuk pemberian nilai secara dinamis:

```
for(int i = 0; i < 5; ++i) {
    cout << "elemen ke-" << i << ": ";
    cin >> ar[i];
}
```

Ketika array dideklarasikan dan belum mendapatkan input maka tiap elemen akan berisi nilai yang tidak diketahui, kecuali bila array (dan variabel tunggal) dideklarasikan dalam lingkup global maka nilai default-nya adalah 0;

Pengaksesan:

Elemen-elemen array dapat diakses oleh program menggunakan suatu indeks tertentu secara random ataupun berurutan. Pengisian dan pengambilan nilai pada indeks tertentu dapat dilakukan dengan mengeset nilai atau menampilkan nilai pada indeks yang dimaksud. Berikut ini adalah contoh kode untuk menampilkan nilai setiap elemen dalam array:

```
cout << "elemen ke-0: " << ar[0] << endl;
cout << "elemen ke-1: " << ar[1] << endl;
cout << "elemen ke-2: " << ar[2] << endl;
cout << "elemen ke-3: " << ar[3] << endl;
cout << "elemen ke-4: " << ar[4] << endl;
```

Pengaksesan elemen di atas dapat juga dilakukan menggunakan struktur perulangan seperti berikut ini:

```
for(int i = 0; i < 5; ++i)
    cout << "elemen ke-" << i << ": "
    << ar[i] << endl;
```

Kedua potongan kode di atas akan menghasilkan output yang sama seperti berikut:

```
elemen ke-0: 4
elemen ke-1: 8
elemen ke-2: 9
elemen ke-3: 2
elemen ke-4: 7
```

Selain menggunakan struktur for tradisional pengaksesan elemen dapat pula menggunakan range-based-for seperti berikut ini:

```
for(auto n : ar) {
    cout << n << " ";
}
```

Struktur di atas dapat dibaca sebagai “untuk semua n dalam ar, lakukan ...”.

Dalam C, tidak terdapat error handling terhadap batasan nilai indeks, apakah indeks tersebut berada di dalam indeks array yang sudah didefinisikan atau belum. Hal ini merupakan tanggung jawab programmer. Sehingga jika programmer mengakses indeks yang salah, maka nilai yang dihasilkan akan berbeda atau rusak karena mengakses alamat memori yang tidak sesuai.

2.3 Array Multidimensi

Array multidimensi merupakan perluasan dari array satu dimensi. Dapat digambarkan sebuah matriks. Jika array satu dimensi hanya terdiri dari satu baris dan beberapa kolom, maka array dua dimensi tersusun dari beberapa baris dan kolom, dimana indeks pertama menyatakan baris dan indeks kedua menyatakan kolom [3]. Array 2 dimensi dapat diilustrasikan dalam bentuk tabel seperti pada Gambar 2.4 berikut ini:

Ilustrasi Array 2 Dimensi

	0	1	2	3	4
0					
1					
2					

Gambar 2. 4 Ilustrasi Array 2 Dimensi

Ilustrasi gambar 2.4 di atas dapat dibaca sebagai array 2 dimensi dengan ukuran 3 x 4. Dimensi pertama menyatakan baris dan dimensi kedua adalah kolom. Sebagai contoh nilai ‘m’ dapat dirujuk pada posisi baris ke-2 kolom ke-1. Berdasar ilustrasi

gambar di atas dapat dikatakan bahwa array multidimensi adalah array di dalam array. Tabel adalah sebuah array yang berisi baris – baris, dan di setiap baris adalah array yang berisi kolom – kolom.

Sebenarnya array banyak dimensi ini tidak terlalu sering dipakai seperti halnya array satu dimensi, dua dimensi, dan tiga dimensi. Array banyak dimensi ini pada dasarnya sama dengan array sebelumnya kecuali pada jumlah dimensinya.

Pendeklarasian:

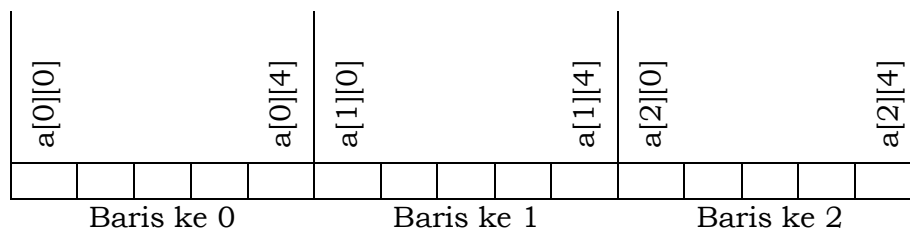
TipeData NamaVarArray[Indeks1] [Indeks2]... [IndeksN];

Pendeklarasian array multidimensi sama seperti pada array 1 dimensi dengan jumlah pasangan square bracket lebih dari satu yang menyatakan dimensinya.

Contoh Pendeklarasian Array Multidimensi:

```
char a[3][5];
```

- Deklarasi di atas menghasilkan array bertipe char dengan nama a
- Pendeklarasian array dengan indeks [3][5] sama dengan matriks berukuran 3x5
- Pengalokasian tempat atau slot di memori diilustrasikan oleh Gambar 2.5.



Gambar 2. 5 Ilustrasi pengalokasian array multidimensi di memori

Inisialisasi

Seperti halnya dalam array 1 dimensi pemberian nilai pada array multidimensi dapat dilakukan melalui 2 cara yaitu:

1) Statis

pemberian nilai dilakukan pada saat pendeklarasian. Berikut ini adalah contoh pemberian nilai pada array 2 dimensi:

```
char ar2d[4][3] = {
    { 'd', 'z', 'x' },
    { 'c', 'v', 'b' },
    { 'a', 'm', 'n' },
    { 's', 'd', 'f' }
};
```

Atau dapat dilakukan satu persatu seperti berikut ini:

```
ar2d[0][0] = 'd';
ar2d[0][1] = 'z';
ar2d[0][2] = 'x';
ar2d[1][0] = 'c';
ar2d[1][1] = 'v';
ar2d[1][2] = 'b';
ar2d[2][0] = 'a';
ar2d[2][1] = 'm';
ar2d[2][2] = 'n';
ar2d[3][0] = 's';
ar2d[3][1] = 'd';
ar2d[3][2] = 'f';
```

2) Dinamis

pemberian nilai dilakukan dari input pengguna. Berikut ini adalah contoh kode untuk pemberian nilai secara dinamis:

```
for(int i = 0; i < 4; ++i)
{
    for(int j = 0; j < 3; ++j)
    {
        cout << "elemen [" << i << ", " << j << "]:" ;
        cin >> ar2d[i][j];
    }
}
```

Berdasar potongan kode di atas dapat diketahui bahwa perulangan luar (counter i) digunakan untuk index dimensi pertama dan perulangan dalam (counter j) digunakan untuk index dimensi kedua. Berikut ini adalah contoh lain pemberian nilai secara dinamis pada array 3 dimensi bertipe char yang bernama ar3d dengan ukuran 4x3x2:

```
for(int i = 0; i < 4; ++i)
{
    for(int j = 0; j < 3; ++j)
    {
        for(int k = 0; k < 2; ++k)
        {
            cout << "elemen [" << i << ", "
                << j << ", "
                << k << "]: ";
            cin >> ar3d[i][j][k];
        }
    }
}
```

Pengaksesan

Elemen – elemen array multidimensi dapat diakses seperti pada array 1 dimensi yaitu dengan merujuk pada nomor – nomor index tiap elemen. Pengaksesan dapat dilakukan satu demi satu atau serentak menggunakan struktur perulangan. Berikut ini adalah contoh mengakses elemen pada array 2 dimensi:

```
for(int i = 0; i < 4; ++i)
{
    for(int j = 0; j < 3; ++j)
        cout << arr2d[i][j] << "\t" ;
    cout << '\n';
}
```

Contoh output potongan kode di atas adalah sebagai berikut:

d	z	x
c	v	b
a	m	n
s	d	f

2.4 Manipulasi

Operasi manipulasi yang dapat dilakukan pada array antara lain:

1. Penambahan elemen array
2. Menampilkan elemen array
3. Pencarian elemen array
 - a. Cari, jika ditemukan, katakan KETEMU!
4. Penghapusan elemen array
 - a. Cari, jika ditemukan kemudian dihapus!
5. Pengeditan elemen array
 - a. Cari, jika ditemukan kemudian diedit!

2.5 Studi Kasus

Program untuk perhitungan statistik sederhana yang dapat mencari data terbesar, terkecil, dan nilai rata – rata.

Ketentuan:

1. Data disimpan dalam array berukuran maksimal 20
2. Input dilakukan secara dinamis oleh pengguna melalui keyboard dengan jumlah maksimal 20 data (boleh kurang)

Penyelesaian:

```
/**
 * program statistik sederhana
 * menampilkan data terbesar, terkecil, dan rata - rata
 *
 **/

#include <iostream>
using namespace std;

int get_max(int ar[], int sz);
int get_min(int ar[], int sz);
double get_avg(int ar[], int sz);

int main()
{
    int data[20];
    int n;

    cout << "banyak data (max 20): ";
    cin >> n;

    for(int i = 0; i < n; ++i)
    {
        cout << "data ke-" << i << ": ";
        cin >> data[i];
    }
    cout << "max: " << get_max(data, n) << endl;
    cout << "min: " << get_min(data, n) << endl;
    cout << "avg: " << get_avg(data, n) << endl;

    return 0;
}

int get_max(int ar[], int sz)
{
    int imax{0};
    for(int i = 1; i < sz; ++i)
        if(ar[i] > ar[imax])
            imax = i;
}
```

```
        return ar[imax];
    }

int get_min(int ar[], int sz)
{
    int imin{0};
    for(int i = 1; i < sz; ++i)
        if(ar[i] < ar[imin])
            imin = i;
    return ar[imin];
}

double get_avg(int ar[], int sz)
{
    double sum{0};
    for(int i = 0; i < sz; ++i)
        sum += ar[i];
    return sum / sz;
}
```

2.6 Latihan

1. Tuliskan nilai setiap elemen dari variabel array a di bawah ini

```
int a[3][3] = { {1, 8}, {2, 4, 6}, {5} };
```

2. Buatlah fungsi rekursif untuk membalik isi sebuah variabel array
3. Buatlah sebuah fungsi bernama testPalindrome untuk menguji apakah sebuah string bersifat palindrome, artinya dibaca dari kiri sama dengan dibaca dari kanan. Fungsi akan menghasilkan nilai 1 jika benar, dan 0 bila tidak.
4. Buatlah fungsi rekursif untuk mendapatkan bilangan terkecil dari n buah bilangan bulat yang tersimpan dalam sebuah variabel array.
5. Apa yang dilakukan oleh program berikut:

```
#include <stdio.h>
#define SIZE 10

int whatIsThis(int [], int);

main() {
    int total, a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    total = whatIsThis(a, SIZE);
    printf("\nNilai variabel total adalah %d", total);
    return 0;
}

int whatIsThis(int b[], int size) {
    if (size == 1)
        return b[0];
    else
        return b[size-1] + whatIsThis(b, size-1);
}
```

3

Pointer

Sub CPMK

Mahasiswa mampu membuat dan mengevaluasi algoritma pengelolaan data dengan menggunakan pointer kedalam program

Bab ini akan membahas tentang deklarasi Pointer, bagaimana data disimpan dan dimanipulasi dalam memori. Pembahasan meliputi operasi – operasi dasar pointer, keterkaitan antara pointer dengan array, dan penggunaan pointer dalam fungsi. Selain itu akan dipaparkan pula mengenai pengalokasian memori secara dinamis. Di akhir bab akan diberikan contoh sebuah studi kasus berupa pembuatan fungsi random generator yang dapat menghasilkan sekumpulan bilangan acak. cara pembuatan pointer sesuai dengan aturan/sintaks penulisan Bahasa C++ yang benar.

3.1 Pengertian

Pada dasarnya semua data yang dimanipulasi oleh program tersimpan dalam memori. Data tersebut tersimpan dalam memori di mana masing-masing memiliki alamat unik. Bila terdapat pendeklarasian variabel maka di memori akan dialokasikan tempat untuk variabel tersebut. Sebagai contoh berikut ini adalah deklarasi beberapa variabel.

```
int x{5};  
auto y = x;  
char z{'a'};
```

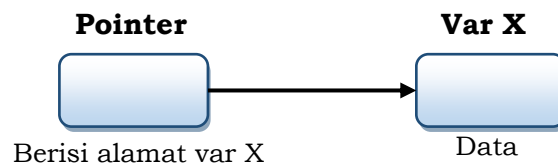
Deklarasi di atas dapat diilustrasikan dalam area memori seperti ditunjukkan Gambar 3.1.

Alamat	Isi	var
1014		
1012		
1010		
1008	a	z
1006		
1004	5	y
1002		
1000	5	x

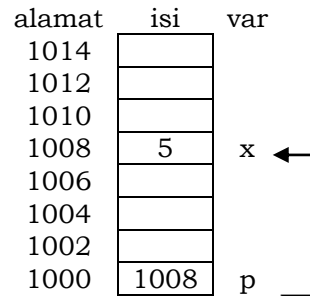
Gambar 3. 1 Alokasi memori untuk variabel

Dari ilustrasi di atas terlihat bahwa variabel x berada pada alamat 1000, y 1004, dan z 1008. Setiap alamat menyimpan sebuah nilai sesuai tipe datanya. Seperti analogi sebuah loker yang terdiri dari banyak tempat penyimpanan. Tiap tempat penyimpanan memiliki nomor dan tiap nomor menyimpan isinya masing – masing.

Pointer merupakan tipe data primitif. Pointer sendiri merupakan suatu variabel penunjuk, yaitu variabel yang tidak berisi data, melainkan berisi nilai yang menunjuk alamat memori dari variabel lain di dalam memori. Dengan kata lain pointer adalah sebuah objek dalam memori yang menunjuk ke sebuah nilai yang tersimpan di memori berdasarkan alamat nilai tersebut. Sebuah pointer dapat mereferensi ke sebuah alamat dan mengambil isinya (dereferensi). Gambar 3.2 merupakan ilustrasi sebuah pointer yang menunjuk ke sebuah alamat.



Gambar 3. 2 Pointer



Gambar 3. 3 Ilustrasi pointer dalam memori

Gambar 3.3 merupakan ilustrasi data yang disimpan di memori pada pointer hanya menyimpan alamat dari variabel yang ditunjuk. Sekilas, pointer *p* terlihat seperti variabel biasa, tetapi nilai yang tersimpan adalah 1008, di mana nilai tersebut adalah alamat dari variabel *x* yang berisi 5.

3.1.1 Deklarasi

Pendeklarasian pointer dapat dilakukan seperti variabel biasa dengan menambahkan operator unary *. Berikut ini adalah contoh pendeklarasian pointer.

Pendeklarasian:

```
TipeData *variabel;
```

Contoh:

```
int *a;
char *p;
```

a dan *p* adalah variabel pointer. *a* merupakan pointer yang menunjuk ke tipe data bertipe *int* sedangkan *p* pointer menunjuk ke tipe data *char*. Sebuah pointer yang dideklarasikan menunjuk ke tipe data tertentu hanya boleh menunjuk alamat memori yang menyimpan nilai dengan tipe data tersebut.

3.1.2 Operator

Suatu pointer dapat berisi alamat dari suatu variable lain untuk dapat mengakses nilai yang ada dalam variable pointer secara langsung dapat dilakukan dengan operator. Operator pointer yang disediakan c++ yaitu :

- 1) Operator Reference atau unary asterix (*)
- 2) Operator Dereference atau ampersand (&).

Kedua operator tersebut bersifat komplemen, artinya satu operator adalah pelengkap operator yang lain. Berikut ini adalah penggunaan kedua operator tersebut:

1. Operator Reference / asterix (*)

Digunakan untuk mendeklarasikan bahwa sebuah variabel adalah pointer. Operator asterix (*) digunakan untuk melakukan pengambilan nilai (dereferensi) suatu alamat memori yang ditunjuk.

Reference (*) merupakan suatu operator yang berfungsi menyatakan suatu variabel adalah variabel pointer. Sama halnya dengan operator dereference, peletakan simbol operator reference diletakan diawal variabel. Operator reference ini akan membuat suatu variabel pointer untuk menampung alamat.

2. Operator Dereference / ampersand (&)

Bersifat unary (hanya memerlukan satu operand saja), digunakan untuk mengambil alamat suatu memori (referensi) melalui nama variabel.

Dereference (&) merupakan suatu operator yang berfungsi untuk menanyakan alamat dari suatu variabel. Apabila kamu memberikan simbol & pada awal variabel dan mencetak hasilnya pada jendela CLI, maka yang akan tercetak adalah alamat dari variabel tersebut bukan nilai yang ditampung oleh variabel tersebut.

Alamat variabel 'a' pada setiap komputer akan berbeda-beda tergantung kompiler dalam mengalokasikan memori untuk suatu variabel.

Berikut ini adalah contoh penggunaan operator * dan &.

```
int *a;
int b{4};
int c{b};
a = &b;
```

Deklarasi di atas menerangkan bahwa a sebuah pointer bertipe int, b dan c variabel bertipe int. Kemudian variabel b diisi nilai 4, dan variabel c berisi salinan isi variabel b. Baris berikutnya a menunjuk ke alamat dari variabel b dengan operasi &b. Gambar 3.4 berikut ini adalah ilustrasi pemetaan area di memori untuk deklarasi di atas.

alamat	isi	var
1014		
1012	5	c
1010		
1008	5	b
1006		
1004		
1002		
1000	1008	a

Gambar 3. 4 Pemetaan Memori pada Variabel dan Pointer

Terlihat bahwa pointer a menyimpan (menunjuk, mereferensi) alamat variabel b yaitu 1008. Pointer a dapat mengakses (dereferensi) isi variabel b dengan operasi *a. Bila deklarasi di atas dilanjutkan dengan operasi berikut:

```
cout << "nilai b : " << b << '\n';  
cout << "nilai c : " << c << '\n';  
cout << "alamat b: " << a << '\n';  
cout << "nilai b : " << *a << '\n';
```

akan menghasilkan tampilan:

```
nilai b : 5  
nilai c : 5  
alamat b: 1008  
nilai b : 5
```

Pointer a menyimpan alamat b, ketika ditampilkan pointer a bernilai 1008 yaitu alamat b. Baris terakhir menampilkan nilai variabel b melalui pointer a. Operasi dilanjutkan seperti berikut:

```
*a = 9;  
cout << "nilai b: " << b << '\n';  
cout << "nilai c: " << c << '\n';
```

Baris pertama berarti “nilai yang ditunjuk oleh a diisi oleh 9”, yang sama artinya dengan mengubah nilai variabel b dengan 9. Berikut ini adalah tampilan yang dihasilkan:

```
nilai b: 9  
nilai c: 5
```

Nilai variabel c tidak terpengaruh dengan perubahan di variabel b karena variabel c hanya menyalin nilai variabel b di operasi sebelumnya. Analoginya seperti mengkopi selembar catatan, ketika lembar catatan asli dicoret – coret maka tidak akan mempengaruhi lembar salinannya, demikian sebaliknya.

3.2 Penggunaan

3.2.1 Aritmatika Pointer

Seperti telah dibahas sebelumnya pointer menyimpan sebuah alamat memori, yang berupa nilai numerik. Oleh karena itu sebuah pointer dapat dikenai operasi aritmatika seperti tipe numerik pada umumnya, hanya saja operasinya terbatas pada operasi penambahan dan pengurangan. Operasi penambahan dengan suatu nilai menunjukkan lokasi data berikutnya (index selanjutnya) dalam memori, begitu juga operasi pengurangan akan menunjuk lokasi data sebelumnya.

Sebagai ilustrasi awal potongan kode berikut ini menunjukkan sebuah pointer yang dikenai operasi aritmatik penjumlahan.

```
int x{6};
int* y{&x};
cout << y << '\n';
y++;
cout << y << '\n';
```

Kode di atas berisi sebuah pointer y ke tipe int yang menunjuk ke alamat variabel x. Pointer y kemudian dikenai operasi increment sehingga naik 1 'tingkat'. Berikut ini adalah output yang muncul (hasil bisa berbeda).

```
0x28fea8
0x28feac
```

Berdasar output di atas terlihat bahwa alamat yang ditunjuk oleh pointer y sebelum dan sesudah operasi increment selisih 4 byte bukannya 1 byte. Hal ini dikarenakan ukuran tipe int adalah 4 byte. Berikut ini adalah contoh lain operasi aritmatik pointer ke tipe double.

```
double x{'a'};
double* y{&x};
cout << y << '\n';
cout << (y + 2) << '\n';
```

Tipe double memiliki ukuran 8 byte sehingga ketika pointer y ditambahkan dengan 2 maka dapat dipastikan bahwa selisih alamat yang ditunjuk oleh y dan (y + 2) sebesar $(1 + 2 * 8)$ byte yaitu 16 byte. Berikut ini adalah output (hasil bisa berbeda) dari potongan kode di atas.

```
0x28fea0
0x28feb0
```

Dari dua ilustrasi di atas dapat disimpulkan bahwa akibat operasi aritmatik pada pointer adalah pergeseran alamat memori yang ditunjuk dengan kelipatan ukuran dari tipe data. Operasi ini dapat digunakan untuk melakukan 'penjelajahan' (traverse) elemen – elemen dalam array.

3.2.2 Array dan Pointer

Aritmatika pointer dapat digunakan untuk melakukan *traversing* elemen – elemen dalam array. Ketika sebuah pointer menunjuk ke sebuah array maka yang ditunjuk adalah alamat dari elemen ke-0. Elemen – elemen dalam array tersusun secara berurutan sehingga dengan menggeser pointer ke arah memori sebelumnya berarti sama juga artinya dengan menunjuk ke elemen array yang di sebelumnya. Besarnya 1 kali pergeseran adalah sebanyak ukuran tipe data pada array. Dengan kata lain bila

elemen array bertipe int maka tiap terjadi 1 kali pergeseran akan melompat sebanyak 4 *byte*.

Potongan kode berikut ini adalah contoh operasi aritmatik pointer terhadap array.

```
int x[] {8, 2, 9, 4, 6, 3, 1};
int* p = x; // p menunjuk x[0]

p++; // p menunjuk x[1]
cout << *p // 2
cout << *(p + 2) // 4
```

Berikut ini adalah output ketika kode dijalankan:

```
2
4
```

Berikut ini adalah contoh lain operasi aritmatik pointer untuk *traversing* seluruh elemen array.

```
int x[] {9, 2, 1, 7, 4};
int* p = x;

for(auto i = 0; i < 5; ++i)
    cout << p++ << ' ';
```

Berikut ini adalah output ketika kode dijalankan:

```
9 2 1 7 4
```

Array dalam bahasa C (*C-style array*) tidak menyimpan informasi mengenai ukuran sehingga mungkin saja terjadi operasi traversing yang di luar batasan (*out of bound*). Sebagai contoh bila perulangan pada kode di atas diubah menjadi seperti berikut ini:

```
for(auto i = 0; i < 8; ++i)
```

Maka yang terjadi adalah pengaksesan pointer ke area memori di luar alokasi untuk elemen array x.

2.2.3 Fungsi dan Pointer

3.2.3.1 Passing Argument by Pointer

Secara default ketika ada sebuah nilai yang dilewatkan (sebagai parameter atau argumen) ke fungsi maka yang terjadi adalah proses penyalinan (*copy*) dari nilai asli. Dengan kata lain jika terjadi manipulasi terhadap nilai yang masuk tersebut tidak akan mempengaruhi nilai aslinya.

Sebagai ilustrasi berikut ini adalah kode untuk fungsi swap() yang digunakan untuk menukar 2 nilai yang dilewatkan.

```
void swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}

int main()
{
    int a{3};
    int b{4};
    swap(a, b);
    // ...
}
```

Pada main() terdapat 2 variabel yaitu a dengan nilai 3 dan b dengan nilai 4. Saat terjadi pemanggilan terhadap fungsi swap() dengan a dan b sebagai argumennya akan terjadi

penyalinan nilai a ke x dan b ke y. Proses penukaran terjadi di dalam fungsi swap() melibatkan variabel lokalnya (x, y, dan tmp). Ketika pemanggilan selesai dan proses dikembalikan ke pemanggil (fungsi main()) maka seluruh variabel lokal di fungsi swap() akan dihapus.

Berdasar kondisi tersebut dapat dikatakan bahwa proses penukaran justru terjadi terhadap x dan y sebagai variabel lokal swap(), bukan pada a dan b sebagai nilai asli yang hendak ditukar.

Pelewatan argumen dengan menyalin nilai asli ke dalam fungsi disebut *passing argument by value* atau sering disingkat *pass by value*. Cara lain untuk melewatkan argumen ke fungsi adalah dengan *pass by pointer*. Cara ini 'hanya' melewatkan alamat nilai asli ke dalam fungsi.

Berikut ini adalah penggunaan pass by value dalam fungsi swap().

```
void swap(int* x, int* y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main()
{
    int a{3};
```

```
int b{4};  
swap(&a, &b);  
// ...  
}
```

Argumen formal dalam fungsi `swap()` (`x` dan `y`) yang merupakan pointer yang menunjuk ke alamat yang dilewatkan. Saat terjadi pemanggilan fungsi `swap()` dengan argumen berupa alamat dari variabel `a` dan `b` maka `x` akan menjadi 'handle' untuk `a` dan `y` untuk `b`. Proses penukaran dalam fungsi `swap()` benar – benar terjadi pada `a` dan `b`.

Selain *pass by pointer* masih terdapat cara lain yaitu *pass by reference* yang akan di bahas pada bab berikutnya (3.3).

3.2.3.2 Pointer ke Fungsi

Sebagaimana variabel fungsi juga memiliki alamat memori. Alamat memori dari sebuah fungsi dapat ditunjuk oleh pointer. Bila pointer ke variabel harus memiliki tipe yang sama maka pointer ke fungsi juga harus memiliki signature yang sama. Berikut ini adalah contoh deklarasi pointer ke fungsi:

```
int (*f)(int, int)
```

f adalah sebuah pointer ke fungsi dengan signature berupa return type int dan 2 parameter bertipe int. Semua fungsi yang memiliki signature sama dengan f dapat ditunjuk. Berikut ini adalah lanjutan kode deklarasi di atas:

```
int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }

f = &add;
cout << (*f)(4, 5); // 9

f = &mul;
cout << (*f)(4, 5) // 45
```

Pemanggilan f harus diawali dengan tanda asterik seperti halnya pada pointer ke variabel. Dari kode di atas f dapat dipandang seperti layaknya fungsi yang sesungguhnya, tetapi dapat berubah perilaku (*polymorphic*). Pointer ke fungsi seperti ini dalam bahasa C# disebut sebagai tipe delegate.

3.3 Reference

Sebuah nilai akan dialokasikan pada alamat memori tertentu. Setiap alamat dapat direferensi oleh pointer. Berbeda dari bahasa C di C++ terdapat cara lain untuk melakukan referensi selain menggunakan pointer yaitu dengan *reference*. Berbeda dengan pointer yang bisa menunjuk ke alamat yang berbeda – beda maka reference hanya bisa ke satu alamat saja.

Sekali *reference* diinisiasi terhadap alamat tertentu maka selama daur hidup tidak boleh berpindah ke alamat yang lain. Selain itu pengoperasian *reference* lebih natural, seperti halnya variabel biasanya; tidak seperti pointer yang menggunakan notasi asterik untuk menunjuk ke nilai yang direferensi. Potongan kode berikut ini adalah contoh pendeklarasian *reference*:

```
int x{8};
int& r = x;
```

Deklarasi di atas menjelaskan bahwa r (ditandai dengan ampersand) merupakan reference ke x. Perbedaan mendasar antara pointer dengan *reference* adalah pada lokasinya. Pada pointer alamat antara pointer itu sendiri dengan alamat yang direferensi berbeda. Reference memiliki alamat yang sama dengan alamat yang

direferensinya sehingga dapat dikatakan bahwa reference adalah alias dari sebuah alamat. Untuk membuktikan perhatikan kode berikut yang merupakan lanjutan deklarasi sebelumnya:

```
cout << &x;  
cout << &r;.
```

Setelah dijalankan dapat dilihat bahwa alamat x dan r adalah sama.

Seperti telah disebutkan pada bahasan sebelumnya (3.2.3.1) tentang *passing argument by pointer* terdapat cara lain untuk melewati argumen yaitu *by reference*. Berikut ini adalah contoh fungsi swap() yang menggunakan passing argument by reference.

```
void swap(int& x, int& y)  
{  
    int tmp = x;  
    x = y;  
    y = tmp;  
}  
  
int main()  
{  
    int a{3};  
    int b{4};  
    swap(a, b);  
    // ...  
}
```

Dari kode di atas dapat dilihat bahwa argumen formal pada fungsi swap() adalah reference ke argumen aktual ketika terjadi pemanggilan.

3.4 Ukuran Data

Ukuran sebuah variabel dapat dihitung menggunakan operator sizeof. Besarnya ukuran tergantung dari besarnya tipe data yang digunakan. Sebagai contoh ukuran tipe int dapat dihitung dengan sizeof(int). Berikut ini adalah contoh potongan kode untuk menampilkan beberapa ukuran tipe data menggunakan operator sizeof.

```
cout << "ukuran int : " << sizeof(int) << endl;  
cout << "ukuran char : " << sizeof(char) << endl;  
cout << "ukuran float : " << sizeof(float) << endl;  
cout << "ukuran double : " << sizeof(double) << endl;
```

Tampilan setelah dijalankan adalah sebagai berikut.

```
ukuran int : 4
```

```
ukuran char : 1
ukuran float : 4
ukuran double : 8
```

Satuan ukuran data seperti yang tertampil pada contoh output di atas adalah byte. Hal yang sama juga berlaku untuk menghitung variabel bertipe array atau struct. Sebagai contoh sebuah array berukuran 5 bertipe float akan memiliki ukuran sebesar 20 byte (4 *byte* x 5).

Potongan kode berikut ini adalah contoh lain untuk perhitungan ukuran array dan struct.

```
struct pixel {
    int x;
    int y;
} p;

float a[5];

cout << "ukuran array a : " << sizeof(a) << endl;
cout << "ukuran struct p : " << sizeof(p);
```

Contoh output untuk potongan kode di atas adalah sebagai berikut.

```
ukuran array a : 20
ukuran struct p : 8
```

Berdasarkan contoh tampilan output di atas ukuran struct p adalah 8 byte karena member dari struct adalah x dan y yang masing – masing bertipe int (4 byte x 2). Namun hal yang berbeda kemungkinan akan terjadi ketika dalam sebuah struct berisi member dengan tipe yang beragam. Berikut ini contoh pendeklarasian struct dengan member beragam sekaligus menampilkan ukurannya.

```
struct pixel {
    int x;
    int y;
    char z;
} p;

...
cout << "ukuran member struct p" << endl;
cout << " p.x: " << sizeof(p.x) << endl;
cout << " p.y: " << sizeof(p.y) << endl;
cout << " p.z: " << sizeof(p.z) << endl;
cout << "ukuran struct p: " << sizeof(p) << endl;
```

Berikut ini adalah contoh tampilan ketika potongan kode di atas dijalankan.

```
ukuran member struct p
p.x: 4
p.y: 4
p.z: 1

ukuran struct p: 12
```

Berdasar contoh tampilan di atas ternyata ukuran struct p berbeda dari jumlah ukuran ketiga membernya. Bila dijumlah total ukuran member struct p adalah: 4 (x: int) + 4 (y: int) + 1 (z:char) = 9. Namun, ketika dihitung secara utuh struct p berukuran 12 sehingga ada selisih 3 byte. Hal ini terjadi berdasar ukuran word dalam sistem yang digunakan adalah 4 byte.

Catatan:

Word adalah satuan natural untuk data yang digunakan oleh prosesor. Ukuran word berupa bit (binary digit). Seluruh implementasi pada buku ini menggunakan sistem 32 bit, artinya panjang word yang digunakan adalah 32 bit. Bila 1 byte terdiri dari 8 bit, maka dapat dikatakan dalam sistem 32 bit panjang word-nya adalah 4 byte (32 / 8 byte).

Dalam contoh di atas member x dan y bertipe int di mana ukurannya sama dengan word sehingga tidak ada penambahan. Pada member z yang berukuran 1 *byte* akan ditambahkan 3 byte agar sesuai dengan jumlah byte dalam word yaitu 4 (1 + 3). Penambahan byte ini biasa disebut sebagai *byte padding*. Seandainya ditambahkan lagi 1 member bertipe char pada struct p maka ukurannya akan tetap 12 byte. Berikut ini adalah contoh modifikasinya.

```
struct pixel {
    int x;
    int y;
    char z;
    char c;
} p;

...
cout << "ukuran member struct" << endl;
cout << " p.x: " << sizeof(p.x) << endl;
cout << " p.y: " << sizeof(p.y) << endl;
cout << " p.z: " << sizeeof(p.z) << endl;
cout << " p.c: " << sizeof(p.c) << endl;
cout << "ukuran struct p: " << sizeof(p) << endl;
```

Bila dijalankan akan menghasilkan output sebagai berikut.

```
ukuran member struct p
p.x: 4
p.y: 4
```

```
p.z: 1  
p.c: 1  
ukuran struct p: 12
```

Berdasar tampilan output di atas bila ukuran tiap member dijumlah akan menghasilkan 10 byte, tetapi ukuran struct tetap 12 byte seperti sebelum ada penambahan member c yang bertipe char.

Selain itu urutan pendeklarasian member dalam struct juga berpengaruh terhadap ukuran total sebuah struct. Sebuah struct dengan ukuran tertentu bisa berubah ukurannya karena perubahan urutan pendeklarasian member.

3.5 Alokasi Dinamis

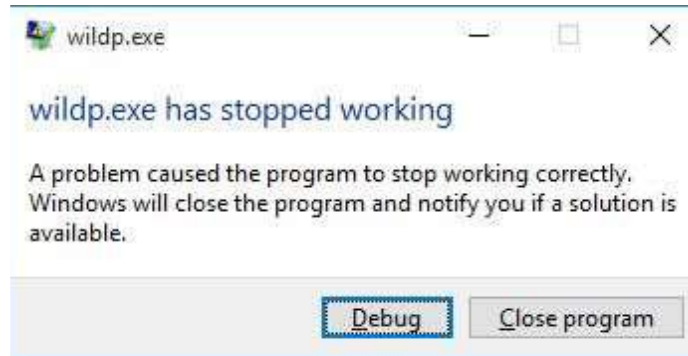
3.5.1 *Wild Pointer*

Seperti telah dibahas sebelumnya pendeklarasian sebuah pointer dapat dilakukan seperti variabel pada umumnya. Ketika sebuah variabel dideklarasikan maka akan dialokasikan sebuah tempat di memori dan akan diisi dengan nilai default sesuai dengan tipe data variabel yang bersangkutan.

Sebuah pointer ketika dideklarasikan akan secara otomatis menunjuk ke sebuah alamat memori secara acak. Setelah pendeklarasian pointer dapat melakukan pengaksesan maupun pengubahan nilai pada alamat yang ditunjuk. Berikut contoh deklarasi 2 buah pointer ke tipe int dan char.

```
int *a;  
char *b;  
  
*a = 9;  
*b = 'a';  
  
cout << "*a: " << *a << ", " << *b: " << *b << endl;
```

Bila potongan kode di atas dijalankan maka kemungkinan besar akan terjadi crash. Berikut ini adalah contoh pesan kesalahan yang muncul ketika program dijalankan.



Gambar 3. 5 Pesan Kesalahan Wild Pointer

Gambar 3.5 adalah tampilan pesan kesalahan yang muncul di lingkungan sistem Windows 10 menggunakan compiler GCC/g++ versi 4.9.2

Pada saat pointer dideklarasikan maka secara otomatis akan menunjuk ke sebuah alamat memori secara acak. Apa yang terjadi ketika pointer menunjuk ke alamat yang sedang digunakan oleh proses lain di luar, kemudian mengubah nilai pada alamat yang ditunjuk tersebut? Maka sistem operasi akan menganggap itu sebagai operasi yang ilegal sehingga akan terblokir.

Pendeklarasian pointer seperti di atas sering di sebut sebagai '*wild pointer*'. Untuk mengantisipasi hal tersebut biasanya pada saat deklarasi pointer akan diinisialisasi dengan nilai `nullptr` sebelum akhirnya menunjuk ke sebuah alamat yang memang sudah ditentukan.

Berikut ini modifikasi deklarasi pada potongan kode sebelumnya.

```
int *a = nullptr;  
char *b = nullptr;
```

Penunjukan ke nilai `nullptr` berarti pointer tidak menunjuk ke alamat manapun.

3.5.2 Alokasi dan Dealokasi

Pembahasan '*wild pointer*' di atas menunjukkan bahwa sebuah pointer harus menunjuk ke alamat yang memang sudah ditentukan untuk menghindari resiko pelanggaran operasi. Selain itu pendeklarasian pointer seperti bahasan sebelumnya bersifat 'statis', artinya sekali pointer dideklarasikan maka akan mengambil sebuah alamat (seperti variabel pada umumnya) untuk menyimpan nilai yang juga berupa alamat dan akan hidup sepanjang program (atau fungsi) berjalan, walaupun sudah tidak terpakai.

Beberapa struktur data memanfaatkan variabel dinamis untuk mengurangi pembrosan memori. Variabel dinamis adalah variabel yang bisa diciptakan Ketika program dieksekusi. Selain dialokasikan dengan cara 'statis' seperti sebelumnya, pointer dapat dialokasikan juga secara dinamis. Pointer yang dialokasikan secara dinamis artinya variabel dapat digunakan hanya pada saat diperlukan saja dan dapat

dihapus (dealokasi) setelah selesai. Untuk menghindari pemborosan memori, alokasi atau dealokasi pointer dapat ditempatkan waktu run.

Variabel dinamis diciptakan dengan operator **new**. Untuk mendealokasi memory (yang baru ditempatkan pada new operator) menggunakan **delete** operator. Berikut ini adalah contoh potongan kode yang menggunakan alokasi memori secara dinamis.

```
int *a = new int;
char *b = new char;

*a = 9;
*b = 'A';

cout << "a menunjuk ke: " << a << ", nilai: " << *a << endl;
cout << "b menunjuk ke: " << b << ", nilai: " << *b << endl;
```

Pada saat deklarasi, pointer **a** dan **b** akan dialokasikan memori untuk ditunjuk menggunakan operator **new**. Operator new akan mengalokasikan memori seukuran tipe data yang dialokasikan. Berdasar deklarasi di atas alokasi untuk pointer a adalah seukuran tipe data int, dan b seukuran tipe char. Ukuran sebuah tipe data didapat menggunakan operator sizeof.

Berikut ini adalah contoh tampilan bila potongan kode di atas dijalankan (alamat bisa berbeda).

```
a menunjuk ke: 008A0F38, nilai: 9
b menunjuk ke: 008A0FB8, nilai: A
```

Potongan kode di atas tidak menyebabkan crash walaupun alamat yang ditunjuk oleh pointer a dan b adalah acak dan diubah nilainya. Hal tersebut karena ketika terjadi pengalokasian memori secara dinamis, compiler akan memastikan hanya memori yang bebas saja yang akan dialokasikan sehingga tidak menyebabkan pelanggaran akses. Selain itu juga alokasi memori yang sudah tidak digunakan dapat dibebaskan (dealokasi) sehingga penggunaan memori lebih efisien. Berikut ini adalah modifikasi dari potongan kode sebelumnya.

```
int *a = new int; //allocation
char *b = new char;

*a = 9;
*b = 'A';

cout << "a menunjuk ke: " << a << ", nilai: " << *a << endl;
cout << "b menunjuk ke: " << b << ", nilai: " << *b << endl;

delete a; // deallocation
delete b;
```

```
cout << "setelah dealokasi\n";

cout << "a menunjuk ke: " << "\n, nilai: ", a, *a);
printf("b menunjuk ke: %p, nilai: %c\n", b, *b);
```

Berikut ini adalah contoh tampilan bila potongan kode di atas dijalankan (alamat bisa berbeda).

```
a menunjuk ke: 008A0F38, nilai: 9
b menunjuk ke: 008A0FB8, nilai: A
setelah dealokasi
a menunjuk ke: 008A0F38, nilai: 7868344
b menunjuk ke: 008A0FB8, nilai: P
```

Dari contoh tampilan di atas setelah didealokasikan pointer tetap menunjuk alamat yang sama. Walaupun alamat yang ditunjuk masih sama tetapi sudah bukan ‘milik’ pointer yang bersangkutan sehingga bila dipaksa untuk diisi nilai kemungkinan besar akan terjadi crash, kecuali sebelumnya di re-alokasi.

Selain keuntungan berupa efisiensi penggunaan memori, pengalokasian memori dinamis ini juga mengakibatkan ‘sampah memori’ ketika pemrogram lupa untuk mendefinisikan memori yang sudah tidak digunakan (berbeda dengan bahasa Java yang memiliki garbage collector untuk membersihkan sampah memori secara otomatis setelah memori tidak digunakan). Oleh karena itu penanganan alokasi memori secara dinamis harus dilakukan dengan sangat cermat.

3.6 Studi Kasus

Sebuah program membutuhkan fungsi yang dapat menghasilkan n bilangan acak antara $1 - n$, misal fungsi tersebut diberi nama *getrandom()*.

Penyelesaian:

Fungsi *getrandom()* membutuhkan sebuah parameter, misal bernama n , bertipe `int` untuk menentukan batas bilangan acak yang dihasilkan. Berikut ini adalah implementasi fungsi *getrandom()*.

```
int *getrand(int n)
{
    int* r(new int[10]);
    srand((unsigned)time(NULL));
    for(auto i = 0; i < n; ++i)
        r[i] = rand() % n + 1;
    return r;
}
```

Dalam fungsi *getrandom()* dideklarasikan pointer r yang menunjuk ke array berukuran n yang dialokasikan secara dinamis.

Berikut ini adalah contoh penggunaannya.

```
int main() {
    int* rands{getrand(5)};
    for(auto i = 0; i < n; ++i)
        cout << rands[i] << endl;
    // dealokasi
    delete [] rands;
    return 0;
}
```

Contoh tampilan setelah dijalankan adalah sebagai berikut.

```
4
3
5
2
2
```


3.7 Latihan

1. Pada fungsi main() di contoh penggunaan fungsi getrand() di atas,
 - a. Mengapa harus melakukan dealokasi?
 - b. Mengapa dealokasi dilakukan di fungsi main(), bukan di getrand()?
2. Diketahui sebuah deklarasi sebagai berikut:

```
int a[]{2, 3, 4, 5, 6};  
int *x, *y, z;
```

Kemudian dikenai operasi – operasi berikut secara berurutan:

(1) $x = a$;

$z = *x$;

Berapa nilai z ?

(2) $*(x + 1) = z$;

Berapa nilai array a sekarang?

(3) $y = \&a[2]$;

$*y = (*x)++ + z$;

Berapa nilai array a sekarang?

Buatlah fungsi untuk mengacak sebuah string yang dimasukkan. Fungsi memiliki sebuah parameter bertipe string sebagai masukan dan mengembalikan nilai string yang sudah teracak. (nama fungsi: scramble())

4

Pencarian Data

Sub CPMK

Mahasiswa mampu merancang algoritma pencarian data (searching) yaitu pencarian berurutan (sequential search) dan pencarian biner (binary search)

Bab ini membahas tentang pencarian sekumpulan data menggunakan 2 metode yaitu *sequential search* dan *binary search*. Pembahasan tiap metode disertai contoh kasus dan ilustrasi secara visual untuk menggambarkan tiap prosesnya. Di akhir pembahasan juga diperkenalkan standard library C++ untuk proses pencarian.

4.1 Pengertian

Aktifitas pencarian data sering kita lakukan dalam kehidupan sehari-hari. Contoh sederhananya adalah seorang kurir melakukan pencarian rumah berdasar pada alamat yang tercantum dalam paket yang dikirimkan. Pada saat melakukan pencarian rumah maka akan dilihat satu persatu nomor tiap rumah apakah sesuai dengan yang tertulis pada paket. Proses pencarian tersebut akan terus dilakukan sampai nomor rumah yang dicari sesuai atau semua rumah sudah dicek dan tidak ditemukan sama sekali.

Dalam pemrograman masalah pencarian adalah salah satu hal yang penting dan mendasar. Pencarian merupakan tindakan untuk mendapatkan suatu data dalam kumpulan data berdasarkan kunci (*key*) atau acuan data [3]. Pencarian ini berkaitan dengan data yang biasanya tersimpan secara terstruktur baik dalam bentuk array atau struktur data yang lain.

Contoh struktur data yang dipakai dalam proses pencarian antara lain: array, list dan pohon biner. Namun pada bab ini hanya membahas pencarian data pada array, struktur data lainnya akan kita bahas di bab selanjutnya.

4.2 Metode

Secara garis besar ada 2 metode dalam pencarian data yaitu pencarian beruntun (*sequential search*) dan pencarian biner (*binary search*). Kedua metode tersebut sebenarnya mengadopsi dari cara pencarian dalam kehidupan sehari – hari.

4.2.1 *Sequential Search*

Sequential search merupakan algoritma pencarian yang paling mudah. Metode ini melakukan pencarian data dalam array (1 dimensi) yang akan menelusuri semua elemen-elemen array dari awal sampai akhir, dimana data-data tidak perlu diurutkan terlebih dahulu.

Metode *sequential search* dapat diterapkan untuk melakukan pencarian data baik pada array yang sudah terurut maupun yang belum terurut. Proses yang terjadi pada metode pencarian ini seperti tahapan berikut:

1. Menentukan data yang dicari.
2. Pada data yang tidak urut, pencarian akan membandingkan data yang dicari dengan data dalam array satu persatu mulai dari indeks pertama data[0] sampai index terakhir data[n-1], tetapi
3. Jika data sudah urut, perbandingan antara data yang dicari dengan data dalam array cocok atau lebih besar dari data yang dicari maka proses pengurutan dihentikan.

Kemungkinan terbaik (*best case*) adalah jika data yang dicari terletak di indeks array terdepan (elemen array pertama) sehingga waktu yang dibutuhkan untuk pencarian data sangat sebentar (minimal). Kompleksitas algoritmanya dinyatakan $O(1)$

Kemungkinan terburuk (*worst case*) adalah jika data yang dicari terletak di indeks array terakhir (elemen array terakhir) sehingga waktu yang dibutuhkan untuk pencarian data sangat lama (maksimal). Kompleksitas algoritmanya $O(N)$

Sebagai ilustrasi ada sebuah array berukuran 7 bertipe `int` dengan nama `ar` berisi data {5, 9, 2, 7, 8, 1, 6}. Data yang akan dicari adalah 7 yang dinyatakan sebagai `x`. Berikut ini adalah contoh proses pencarian yang bisa digunakan.

```
bool found{false};
for(auto i = 0; i < 7; ++i)
{
    if(x == arr[i])
    {
        found = true;
        break;
    }
}
```

Pada potongan kode di atas terdapat variabel `x` sebagai penampung data yang di cari yaitu 7 dan variabel `found` sebagai penanda apakah data yang dicari ketemu. Sebagai nilai awal variabel `found` adalah `false` sebagai penanda bahwa data belum ditemukan.

Pada saat proses perulangan ke-*i* nilai `x` akan selalu dibandingkan dengan nilai `arr[i]`. Jika perbandingan sesuai maka nilai `found` akan diset menjadi `true` dan perulangan dihentikan.

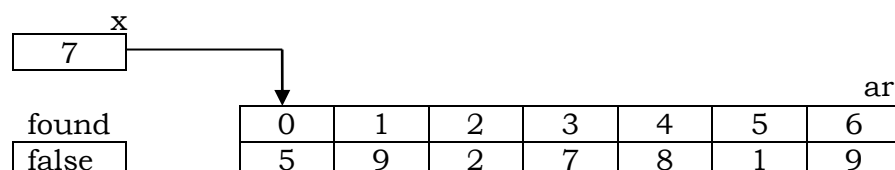
Akhir proses dari contoh di atas akan didapat nilai *k* adalah 3, sehingga pencarian dikatakan berhasil yaitu data ketemu pada index yang ke-3.

Ilustrasi visual pencarian secara *sequential* adalah sebagai berikut:

Contoh:

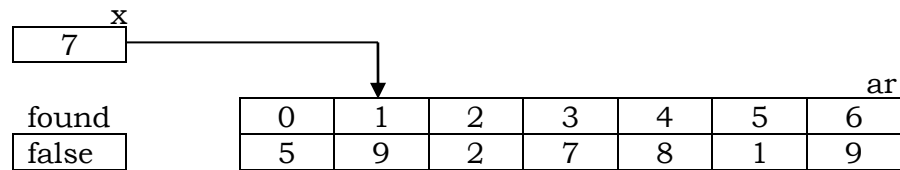
Mencari 7 (target $x = 7$)

Perulangan $i = 0$



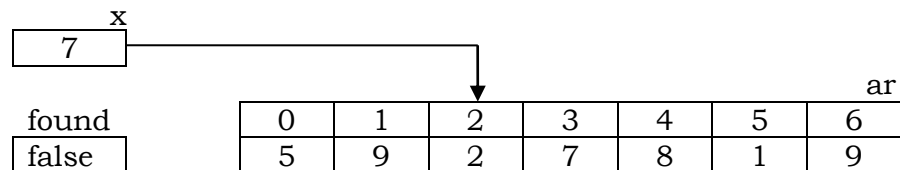
Gambar 4. 1 Langkah 1 Ilustrasi Sequential Search

Perulangan i = 1



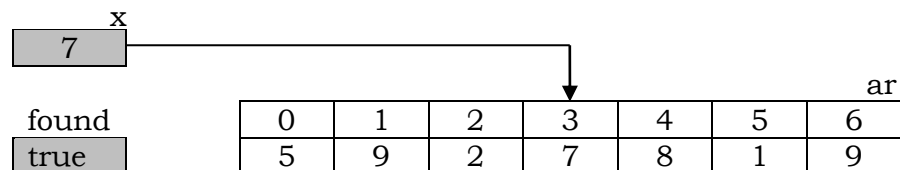
Gambar 4. 2 Langkah 2 Ilustrasi Sequential Search

Perulangan i = 2



Gambar 4. 3 Langkah 3 Ilustrasi Sequential Search

Perulangan i = 3



Gambar 4. 4 Langkah 4 Ilustrasi Sequential Search

Dari gambar 4.1 hingga Gambar 4.4 dapat dilihat pada perulangan $i=3$ ditemukan target $x=ar[3]$ yaitu sama-sama bernilai 7 sehingga perulangan dihentikan.

4.2.2 Binary Search

Merupakan teknik pencarian data dalam array dengan cara membagi array menjadi dua bagian setiap kali terjadi proses pengurutan. Data yang ada harus diurutkan terlebih dahulu berdasarkan suatu urutan tertentu yang dijadikan kunci pencarian.

Teknik pencarian data dalam dengan cara membagi data menjadi dua bagian setiap kali terjadi proses pencarian. Metode *binary search* hanya digunakan untuk pencarian data yang sudah terurut. Proses yang terjadi pada pencarian dengan metode ini adalah sebagai berikut:

1. Menentukan data yang akan dicari.
2. Menentukan elemen tengah dari array.
Rumus: $(\text{posisi awal} + \text{posisi akhir}) / 2$

3. Data yang dicari dibandingkan dengan data yang di tengah, apakah sama atau lebih kecil, atau lebih besar?
 - a. Jika nilai elemen tengah sama dengan data yang dicari maka pencarian selesai
 - b. Jika nilai elemen tengah lebih besar daripada data yang dicari maka pencarian dilakukan pada setengah array pertama, tetapi
 - c. Jika nilai elemen tengah lebih kecil daripada data yang dicari maka pencarian dilakukan pada setengah array berikutnya.
4. Ulangi langkah 3 selama data belum ketemu dan data masih ada.

Sebagai ilustrasi terdapat array berukuran 11 bertipe `int` dengan nama `ar` berisi data terurut naik {1, 2, 5, 7, 8, 9, 9, 11, 15, 16, 20}. Sebagai contoh data yang akan dicari adalah 5 dinyatakan sebagai `x` dan penanda bila data ketemu adalah `found`. Proses pencarian dapat dilakukan seperti berikut.

```
int l{0};
int r{10};
bool found{false};

while(l <= r && !found)
{
    m = l + (r - l) / 2;

    if(x == arr[m])
        found = true;

    if(x < arr[m])
        r = m - 1;
    else if(x > arr[m])
        l = m + 1;
}
```

Dari potongan kode di atas terdapat 3 variabel lagi yaitu `l`, `m`, dan `r` masing – masing digunakan sebagai penanda posisi kiri (`l`), tengah (`m`), dan kanan (`r`) sebagai batas pencarian.

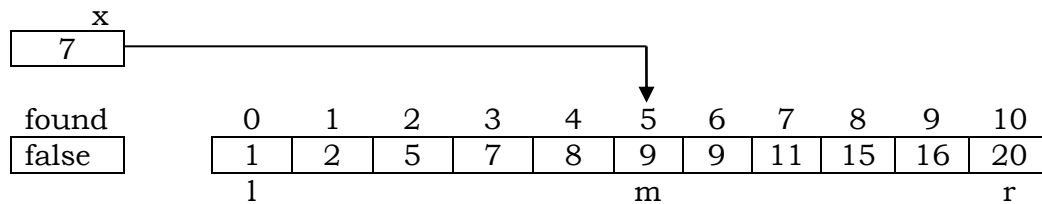
Proses pencarian akan dilakukan selama posisi index yang ditunjuk `l <= r`. Proses pencocokan selalu mengacu terhadap data pada posisi `m` di mana `x == arr[m]`. Berdasar ilustrasi di atas maka data ditemukan pada index yang ke-3.

Ilustrasi visual dari proses pencarian secara *binary* sebagai berikut:

Contoh:

Mencari 7 (target $x = 7$)

Perulangan pertama

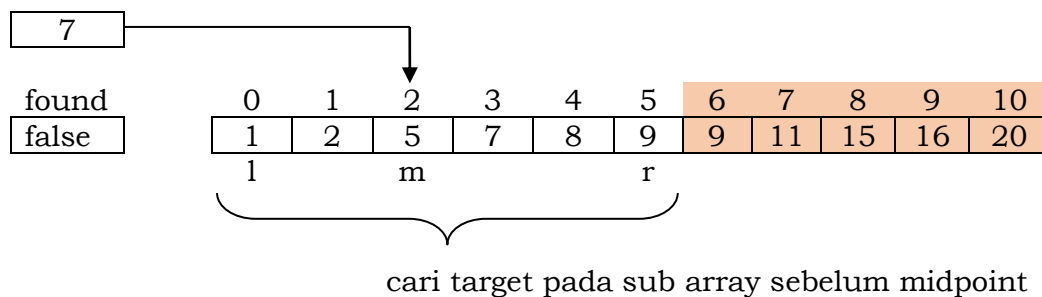


Gambar 4. 5 Langkah 1 Ilustrasi Binary Search

Apakah $7 = m$?

Apakah $7 < m$? YA

Perulangan Kedua



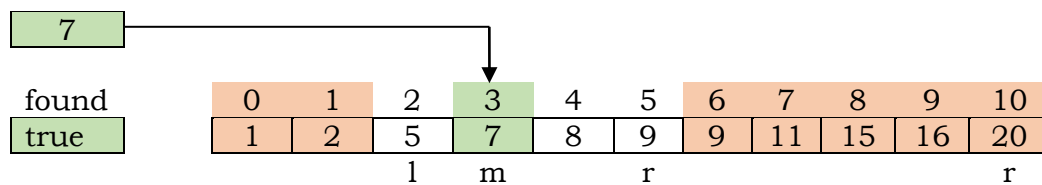
Gambar 4. 6 Langkah 2 Ilustrasi Binary Search

Apakah $7 = m$?

Apakah $7 < m$?

Apakah $7 > m$? YA

Perulangan Ketiga



Apakah $7 = m$? YA

Gambar 4. 7 Langkah 3 Ilustrasi Binary Search

4.2.2.1 Binary Search dengan Rekursi

Selain menggunakan perulangan seperti ilustrasi di atas binary search dapat pula dibuat dengan rekursi. Berikut ini adalah contoh fungsi rekursif untuk binary search.

```
bool binsearch(int n[], int x, int l, int r)
{
    int m = l + (r - l) / 2;

    if(x == n[m])
        return true;

    if(x < n[m])
        return binsearch(x, l, m - 1);

    if(x > n[m])
        return binsearch(x, m + 1, r);

    if(l > r)
        return false;
}
```

4.3 Standard Library `std::find()`

Dalam praktek pemrograman sesungguhnya seorang pemrogram C++ tidak harus mengimplementasikan sendiri algoritma untuk pencarian. C++ menyediakan beberapa fungsi dalam standard library untuk keperluan pencarian, diantaranya yang sering digunakan dan relevan dengan pembahasan di bab ini adalah `std::find()` dan `std::find_if()`. Definisi kedua fungsi tersebut ada dalam header `<algorithm>`.

4.4 Studi Kasus

Terdapat sebuah array dengan ukuran maksimal 20. Elemen array diisi dengan input oleh pengguna saat program berjalan. Program harus bisa melakukan pencarian terhadap elemen array dan bila ketemu maka harus menampilkan semua posisi index-nya bila terdapat lebih dari satu elemen yang sama nilainya.

Penyelesaian

```
/**
 * program sequential search
 * dapat menampilkan semua index data yang ditemukan
 */

#include <iostream>
using namespace std;
const int MAX_SIZE{20}

int data[MAX_SIZE]; // array data
int idx[MAX_SIZE];  // array untuk menyimpan index elemen yang
ditemukan
int count{0}; // counter, menghitung ada berapa banyak data yang
ditemukan

void search(int x);

int main(void) {
    int n;
    cout << "jumlah data: ";
    cin << n;
    for(auto i = 0; i < n; ++i) {
        cout << "data ke-<< i << "i: ";
        cin >> data[i];
    }

    int x;
    cout << "cari: ";
    cin >> x;

    search(x);
    // jika counter > 0, berarti ada data yang ditemukan
    if(count > 0)
    {
        cout << "ditemukan pada index: ";
        for(auto i = 0; i < count; ++i)
        {
            cout << idx[i] << ", ";
        }
    }
}
```

```

}
else
{
    cout << "data tidak ditemukan\n";
}

return 0;
}

void search(int x) {
    for(auto i = 0; i < n; ++i)
    {
        // jika x ditemukan pada data[i]
        if(x == data[i])
        {
            // simpan index i ke array idx
            idx[count++] = i;
        }
    }
}

```

4.5 Latihan

1. Pencarian nomor halaman buku dapat dilakukan dengan metode apa? Berikan alasan dan jelaskan dengan contoh.
2. Buatlah data string untuk menyimpan nama kota dan provinsi kemudian coba lakukan pencarian untuk kota tempat tinggal Anda dan tampilkan nama provinsinya.
3. Buatlah program phonebook yang berisi data nama dan nomor telepon. Phonebook memiliki kapasitas maksimal 50.

Ketentuan:

Program memiliki 2 array, 1 array untuk menyimpan nama dan 1 lagi untuk menyimpan nomor telepon. Di dalam program terdapat fungsi untuk melakukan penambahan dan pencarian data.

5

Pengurutan Data

Sub CPMK

Mahasiswa mampu merancang algoritma pengurutan data (*sorting*) yang tepat terhadap data dalam larik (array)

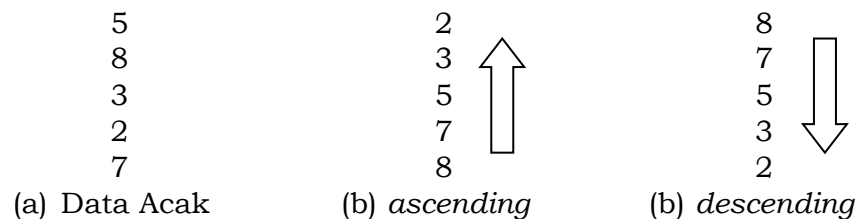
Bab ini membahas tentang metode pengurutan (*sorting*) data dasar dan lanjutan. Pembahasan awal berfokus pada 3 metode pengurutan dasar yaitu *Bubble sort*, *Insertion sort*, dan *Selection sort*. Tiap metode dibahas menggunakan contoh dan disertai dengan ilustrasi visual untuk menggambarkan tiap prosesnya. Pembahasan dilanjutkan dengan contoh kasus berupa perbandingan kecepatan di antara ketiga metode. Di pembahasan akhir akan diperkenalkan dengan metode pencarian lanjut seperti *Shell sort*, *Merge sort* dan *Quick sort*. Selain itu juga diperkenalkan secara singkat tentang fungsi dalam standard library C++ untuk pengurutan.

5.1 Pengertian

Pengurutan atau sorting diartikan sebagai proses penyusunan kembali sekumpulan obyek ke dalam urutan tertentu. Tujuan pengurutan untuk mendapatkan kemudahan dalam pencarian anggota dari suatu himpunan disamping dapat mempercepat mengetahui data terbesar dan data terkecil, misalnya untuk mengetahui perolehan nilai tertinggi dan nilai terendah dari hasil ujian. Contoh obyek terurutkan seperti daftar isi, daftar pustaka dan lain-lain.

Dalam kehidupan sehari – hari juga sering ditemukan aktivitas pengurutan seperti pengurutan nilai, pengurutan susunan buku, dan lain sebagainya.

Pengurutan data dalam struktur data sangat penting untuk data yang bertipe data numerik ataupun karakter. Pengurutan dapat dibedakan menjadi dua jenis, yakni pengurutan secara *ascending* (urut naik) dan *descending* (urut turun). Pengurutan secara *ascending* biasa dinyatakan dengan A ke Z dan pengurutan *descending* sering dinyatakan Z ke A. Gambar 5.1 merupakan contoh ilustrasi pengurutan data.



Gambar 5. 1 Pengurutan Data

Secara umum proses yang terjadi pada pengurutan adalah sebagai berikut:

1. Perbandingan data
2. Pertukaran data

5.2 Metode Pencarian Dasar

Ada beberapa metode pengurutan dasar yang dapat digunakan. Masing – masing metode memiliki kelebihan dan kekurangan. Hal yang perlu menjadi pertimbangan saat memilih metode pencarian adalah jumlah data, kompleksitas waktu dan lama waktu untuk membuat program.

Beberapa metode pengurutan dasar diantaranya adalah sebagai berikut:

1. Pengurutan berdasarkan perbandingan (*comparison-based sorting*)
 - a. *Bubble sort*,
 - b. *Exchange sort*
2. Pengurutan berdasarkan prioritas (*priority queue sorting method*)
 - a. *Selection sort*,
 - b. *Heap sort* (menggunakan tree)

3. Pengurutan berdasarkan penyisipan dan penjagaan terurut (*insert and keep sorted method*)
 - a. *Insertion sort, Tree sort*
4. Pengurutan berdasarkan pembagian dan penguasaan (*divide and conquer method*)
 - a. *Quick sort, Merge sort*
5. Pengurutan berkurang menurun (*diminishing increment sort method*)
 - a. *Shell sort* (pengembangan insertion)

Di sini ini hanya akan membahas beberapa metode sorting yang paling sederhana yang merupakan cara pengurutan dasar yaitu *bubble sort*, *selection sort*, dan *insertion sort*.

5.2.1 *Bubble Sort*

Metode *bubble* mengadopsi dari sifat gelembung air, di mana gelembung yang lebih besar akan lebih mudah dan lebih cepat mencapai permukaan. Prosesnya adalah dengan membandingkan dua demi dua data yang saling berdekatan, jika diperlukan. Bila data di kiri lebih besar (atau lebih kecil) maka akan ditukar. Proses akan terus berlangsung hingga seluruh array telah diperiksa dan tidak ada pertukaran lagi yang bisa dilakukan, serta semua data “digelembungkan” sesuai urutan besarnya nilai.

Metode *bubble sort* disebut juga *exchange sort*. Metode yang mengurutkan data dengan cara membandingkan masing-masing elemen, kemudian melakukan penukaran bila perlu.

Pembandingan elemen dapat dimulai dari awal atau mulai dari paling akhir. Dengan contoh array, ilustrasi pengurutan naik dari elemen awal menggunakan metode *bubble* adalah sebagai berikut:

1. Kondisi awal

0	1	2	3	4	5
5	9	7	3	6	4

Gambar 5. 2 Kondisi awal (array blm urut)

2. Langkah 1


Mengelembungkan data terbesar ke-1

0	1	2	3	4	5
5	9	7	3	6	4
5	9	7	3	6	4
5	7	9	3	6	4
5	7	3	9	6	4
5	7	3	6	9	4
5	7	3	6	4	9

→ Tidak ada penukaran karena $5 < 9$

Gambar 5. 3 Langkah 1 Pengurutan Bubble Sort

Keterangan:

 Data terurut

Dari Gambar 5.3 dapat terlihat bahwa tidak ada pertukaran data di awal proses karena 5 sudah lebih kecil dari 9, setelah itu terlihat bahwa angka 9 menggelembung dan bergeser hingga ke ujung array indeks terakhir sehingga **didapat data terurut 9**.

3. Langkah 2

Menggelembungkan data terbesar ke-2

0	1	2	3	4	5
5	7	3	6	4	9
5	7	3	6	4	9
5	3	7	6	4	9
5	3	6	7	4	9
5	3	6	4	7	9

→ Tidak ada penukaran karena $5 < 7$

Gambar 5. 4 Langkah 2 Pengurutan Bubble Sort

Didapat data terurut 7 dan 9

4. Langkah 3

Menggelembungkan data terbesar ke-3

0	1	2	3	4	5
5	3	6	4	7	9
3	5	6	4	7	9
3	5	6	4	7	9
3	5	4	6	7	9

Gambar 5. 5 Langkah 3 Pengurutan Bubble Sort

Didapat data terurut 6, 7 dan 9

5. Langkah 4

Menggelembungkan data terbesar ke-4

0	1	2	3	4	5
3	5	4	6	7	9
3	5	4	6	7	9
3	4	5	6	7	9

Gambar 5. 6 Langkah 4 Pengurutan Bubble Sort

Didapat data terurut 5,6, 7 dan 9

6. Langkah 5
Menggelembungkan data terbesar ke-5

0	1	2	3	4	5
3	4	5	6	7	9
3	4	5	6	7	9

Gambar 5. 7 Langkah 5 Pengurutan Bubble Sort

Didapat data terurut 4, 5, 6, 7 dan 9

7. Hasil akhir semua data sudah terurut

0	1	2	3	4	5
3	4	5	6	7	9

Gambar 5. 8 Hasil pengurutan dengan Bubble Sort

Jika jumlah data sebanyak n , terjadi $n-1$ tahap pengurutan. Jika contoh diatas jumlah $n = 6$ maka jumlah langkah pengurutan sebanyak 5 tahap pengurutan data. Gambar 5.9 mengilustrasikan keadaan awal sampai akhir hingga semua data terurut.

	0	1	2	3	4	5
Awal	5	9	7	3	6	4
Langkah 1	5	7	3	6	4	9
Langkah 2	5	3	6	4	7	9
Langkah 3	3	5	4	6	7	9
Langkah 4	3	4	5	6	7	9
Langkah 5	3	4	5	6	7	9
Akhir	3	4	5	6	7	9

Gambar 5. 9 Rangkuman proses pengurutan data dengan Bubble Sort

Potongan kode untuk proses di atas adalah sebagai berikut:

```
const int SIZE{6};
...
for(int i = 0; i < SIZE; ++i)
{
    for(int j = 0; j < SIZE - i - 1; ++j)
    {
        if(MyArray[j] > MyArray[j + 1])
        {
            int tmp = MyArray[j];
            MyArray[j] = MyArray[j + 1];
            MyArray[j + 1] = tmp;
        }
    }
}
```

```
}  
}
```

Untuk melakukan pengurutan turun dapat dilakukan dengan cara sebaliknya, pada kondisi:

```
if(MyArray[j] > MyArray[j + i])
```

tanda lebih besar (>) diganti dengan tanda lebih kecil (<).

Metode *bubble* merupakan metode *sorting* termudah dan paling sederhana, dengan sifatnya sangat cepat menangani data yang sedikit dan akan menyita waktu yang lama untuk data yang sangat banyak.

Berdasarkan algoritma diatas, jumlah perbandingan selama pengurutan sebanyak:
 $(N-1) + (N-2) + (N-3) + \dots + 1 = N * (N-1)/2$

Jumlah tersebut berlaku pada *best case*, *worst case* ataupun *average case*. Untuk $N = 6$, jumlah perbandingan data sebanyak $(6*5/2) = 15$ kali. Dengan demikian kompleksitas waktu dari *bubble sort* ini dinyatakan serupa dengan $O(N*(N-1)/2)$ atau sama dengan $O(N^2)$.

Dengan algoritma semua data akan tetap dicek dengan data disampingnya selama perulangan masih berlangsung, sehingga *best case* saat data sudah kondisi terurut tidak dapat tercapai. Beberapa perbaikan perlu dilakukan dengan menambahkan **break** pada kondisi saat sudah tidak terjadi pertukaran data untuk menghentikan perulangan. Prinsip dasarnya adalah jika tidak ada pertukaran data yang terjadi berarti data telah urut, sehingga perulangan dapat dihentikan.

Best case

- Array sudah dalam keadaan terurut naik
- Jumlah perbandingan key: $n-1$
- Jumlah swap = 0
- Jumlah pergeseran = 0

Worst case

- Array dalam urutan kebalikannya
- Jumlah perbandingan key:
 $(1 + 2 + \dots + n-1) = n * (n-1)/2$
- Jumlah pergeseran = $3 * n * (n-1)/2$

5.2.2 Selection Sort

Merupakan kombinasi antara algoritma *sorting* dan *searching*. Metode *selection sort* akan memilih elemen terkecil dari sisa array yang belum terurut. Dalam setiap prosesnya, akan dicari elemen-elemen yang belum diurutkan yang memiliki nilai terkecil atau terbesar akan dipertukarkan ke posisi yang tepat di dalam array.

Misalnya untuk putaran pertama, akan dicari data dengan nilai terkecil dan data ini akan ditempatkan di indeks terkecil (data[0]), pada putaran kedua akan dicari data kedua terkecil, dan akan ditempatkan di indeks kedua (data[1]).

Selama proses, perbandingan dan pengubahan hanya dilakukan pada indeks perbandingan saja, pertukaran data secara fisik terjadi pada akhir proses.

Masih menggunakan contoh array yang sama, MyArray, seperti sebelumnya, proses pengurutan menggunakan metode *selection sort* secara pengurutan naik (*ascending*) dimulai dari sisi kiri / depan dapat digambarkan seperti langkah-langkah berikut ini.

1. Kondisi awal

0	1	2	3	4	5
5	9	7	3	6	4

Gambar 5. 10 Kondisi awal (data belum urut)

2. Langkah 1

Mencari nilai terkecil pada data yang belum diurutkan untuk ditempatkan di index-0.

	0	1	2	3	4	5
awal	5	9	7	3	6	4
akhir	3	9	7	5	6	4

Gambar 5. 11 Langkah 1 Pengurutan Selection Sort

Keterangan:

Data terurut

Proses:

Nilai min awal = 5

Membandingkan elemen pertama dengan elemen lainnya, jika lebih kecil akan disimpan sebagai nilai min

$5 < 9$ (min = 5)

$5 < 7$ (min = 5)

$5 > 3$ (min = 3)

$3 < 6$ (min = 3)

$3 < 4$ (min = **3**)

Tukarkan nilai min (index-3) dengan nilai pada index-0.

Didapatkan data terurut 3.

3. Langkah 2

Mencari nilai terkecil pada data yg belum terurut untuk ditempatkan di index-1.

	0	1	2	3	4	5
awal	3	9	7	5	6	4
akhir	3	4	7	5	6	9

Gambar 5. 12 Langkah 2 Pengurutan Selection Sort

Nilai min awal = 9

Mencari nilai min:

$9 > 7$ (min = 7)

$7 > 5$ (min = 5)

$5 < 6$ (min = 5)

$5 > 4$ (min = **4**)

Tukarkan nilai min (index-5) dengan nilai pada index-1.

Didapatkan data terurut 3 dan 4.

4. Langkah 3

Mencari nilai terkecil pada data yg belum terurut untuk ditempatkan di index-2.

	0	1	2	3	4	5
awal	3	4	7	5	6	9
akhir	3	4	5	7	6	9

Gambar 5. 13 Langkah 3 Pengurutan Selection Sort

Nilai min awal = 7

Mencari nilai min:

$7 > 5$ (min = 5)

$5 < 6$ (min = 5)

$5 < 9$ (min = **5**)

Tukarkan nilai min (index-3) dengan nilai pada index-2.

Didapatkan data terurut 3, 4 dan 5.

5. Langkah 4

Mencari nilai terkecil pada data yg belum terurut untuk ditempatkan di index-3.

	0	1	2	3	4	5
awal	3	4	5	7	6	9
akhir	3	4	5	6	7	9

Gambar 5. 14 Langkah 4 Pengurutan Selection Sort

Nilai min awal = 7

Mencari nilai min:

$7 > 6$ (min = 6)

$6 < 9$ (min = 6)

Tukarkan nilai min (index-4) dengan nilai pada index-3.

Didapatkan data terurut 3, 4, 5 dan 6.

6. Langkah 5

Mencari nilai terkecil pada data yg belum terurut untuk ditempatkan di index-3.

	0	1	2	3	4	5
awal	3	4	5	6	7	9
akhir	3	4	5	6	7	9

Gambar 5. 15 Langkah 5 Pengurutan Selection Sort

Nilai min awal = 7

Mencari nilai min:

$7 < 9$ (min=7)

Tidak ada pertukaran data karena $7 < 9$.

Didapatkan data terurut 3, 4, 5, 6, 7 dan 9.

Hasil akhir semua data sudah terurut

0	1	2	3	4	5
3	4	5	6	7	9

Gambar 5. 16 Hasil pengurutan Selection Sort

Gambar 5.17 mengilustrasikan keseluruhan proses pengurutan dengan *selection sort* dari keadaan awal sampai akhir hingga semua data terurut.

	0	1	2	3	4	5
Awal	5	9	7	3	6	4
Langkah 1	3	9	7	5	6	4
Langkah 2	3	4	7	5	6	9
Langkah 3	3	4	5	7	6	9
Langkah 4	3	4	5	6	7	9
Langkah 5	3	4	5	6	7	9
Akhir	3	4	5	6	7	9

Gambar 5. 17 rangkuman proses pengurutan data dengan Selection Sort

Potongan kode untuk proses selection sort adalah

```
const int SIZE{6};
...
for(int i = 0; i < SIZE-1; ++i)
{
    pos = i;
    for(int j = i+1; j < SIZE - i - 1; ++j)
    {
        if(MyArray[j] < MyArray[pos])
        {
            pos = j;
        }
    }

    if(pos != i)
    {
        int tmp = MyArray[j];
        MyArray[j] = MyArray[pos];
        MyArray[pos] = tmp;
    }
}
```

Untuk melakukan pengurutan turun dapat dilakukan dengan cara sebaliknya, pada kondisi:

```
if(MyArray[j] > MyArray[pos])
```

tanda lebih besar (>) diganti dengan tanda lebih kecil (<)

Analisa Selection Sort

- Operasi dasarnya `MyArray[j] < MyArray[min]`, jika dilihat dari program diatas maka nilai min disimpan oleh variabel pos.
- Tidak ada *best case* dan *worst case*.
- Total pergeseran
 $M = 3 * n - 1$ (pada setiap penukaran terjadi 3x pergeseran)

- Jumlah pembandingannya

$$C = 1 + 2 + \dots + n-1$$

$$= n * (n-1)/2$$

5.2.3 Insertion Sort

Metode *insertion sort* mengadopsi dari kehidupan nyata, seperti pengurutan dokumen pada filing kabinet. Prosesnya adalah dengan cara mengambil satu demi satu data yang masih acak untuk dicarikan posisi yang sesuai di deratan data yang sudah terurut kemudian disisipkan (*insert*). Proses akan terus berlangsung hingga semuanya berada pada posisi yang sudah sesuai. Singkatnya dapat dipahami bahwa *insertion sort* adalah proses menyisipkan data ke array yang sudah terurut.

Masih menggunakan contoh array yang sama, MyArray, seperti sebelumnya, proses pengurutan menggunakan metode *insertion* dapat digambarkan seperti berikut.

1. Kondisi awal

Menganggap data index-0 sudah terurut.

0	1	2	3	4	5
5	9	7	3	6	4

Gambar 5. 18 Kondisi awal (data belum urut)

Didapatkan data terurut 5.

2. Langkah 1

Mencari posisi nilai indeks-1 yaitu 9 dengan cara membandingkan nilai data pada data yang sudah terurut.

0	1	2	3	4	5
5	9	7	3	6	4
5	9	7	3	6	4

Gambar 5. 19 Langkah 1 Pengurutan Insertion Sort

Data 9 dibandingkan dengan data 5. Tidak ada pergeseran posisi karena $5 < 9$.

Didapatkan data terurut 5 dan 9.

3. Langkah 2

Mencari posisi nilai indeks-2, yaitu 7

0	1	2	3	4	5
5	9	7	3	6	4
5	7	9	3	6	4

Gambar 5. 20 Langkah 2 Pengurutan Insertion Sort

Data 7 dibandingkan dengan data 9, karena $7 < 9$ maka data 7 ke posisi index 1.

Data 7 dibandingkan lagi dengan data 5, karena $7 > 5$ maka tidak ada pergeseran ke index 0.

Didapat data terurut 5, 7 dan 9.

4. Langkah 3

Mencari posisi nilai indeks-3, yaitu 3

0	1	2	3	4	5
5	7	9	3	6	4
3	5	7	9	6	4

Gambar 5. 21 Langkah 3 Pengurutan Insertion Sort

Data 3 dibandingkan dengan data 9, karena $3 < 9$ maka data 3 ke posisi index 2.

Data 3 dibandingkan lagi dengan data 7, karena $3 < 7$ maka data 3 ke posisi index 1.

Data 3 dibandingkan lagi dengan data 5, karena $3 < 5$ maka data 3 ke posisi index 0.

Didapat data terurut 5, 7 dan 9.

5. Langkah 4

Mencari posisi nilai indeks-4, yaitu 6

0	1	2	3	4	5
3	5	7	9	6	4
3	5	6	7	9	4

Gambar 5. 22 Langkah 4 Pengurutan Insertion Sort

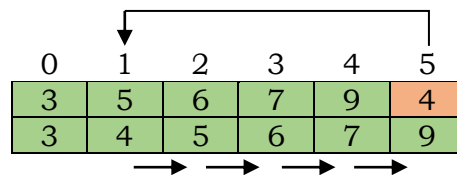
Data 6 dibandingkan dengan data 9, karena $6 < 9$ maka data 6 ke posisi index 3.

Data 6 dibandingkan lagi dengan data 7, karena $6 < 7$ maka data 6 ke posisi index 2.

Data 6 dibandingkan lagi dengan data 5, karena $6 > 5$ maka tidak ada pergeseran.

6. Langkah 5

Mencari posisi nilai indeks-5, yaitu 4



Gambar 5. 23 Langkah 5 Pengurutan Insertion Sort

Data 4 dibandingkan dengan data 9, karena $4 < 9$ maka data 4 ke posisi index 4.

Data 4 dibandingkan lagi dengan data 7, karena $4 < 7$ maka data 4 ke posisi index 3.

Data 4 dibandingkan lagi dengan data 6, karena $4 < 6$ maka data 4 ke posisi index 2.

Data 4 dibandingkan lagi dengan data 5, karena $4 < 5$ maka data 4 ke posisi index 1.

Data 4 dibandingkan lagi dengan data 3, karena $4 > 3$ maka tidak ada pergeseran.

Hasil akhir semua data sudah terurut

0	1	2	3	4	5
3	4	5	6	7	9

Gambar 5. 24 Hasil pengurutan Selection Sort

Gambar 5.25 mengilustrasikan proses pengurutan dengan *insertion sort* dari keadaan awal sampai akhir hingga semua data terurut.

	0	1	2	3	4	5
Awal	5	9	7	3	6	4
Langkah 1	5	9	7	3	6	4
Langkah 2	5	7	9	3	6	4
Langkah 3	3	5	7	9	6	4
Langkah 4	3	5	6	7	9	4
Langkah 5	3	4	5	6	7	9
Akhir	3	4	5	6	7	9

Gambar 5. 25 Proses pengurutan data dengan Insertion Sort

Potongan kode untuk proses di atas adalah sebagai berikut.

```
const int SIZE{6};
...
for(int i = 0; i < SIZE; ++i)
{
    for(int j = 0; j < SIZE - i - 1; ++j)
    {
        if(MyArray[j] > MyArray[j + 1])
        {
            int tmp = MyArray[j];
            MyArray[j] = MyArray[j + 1];
            MyArray[j + 1] = tmp;
        }
    }
}
```

Analisa *insertion sort*

Operasi dasar terdapat pada perbandingan key (`MyArray[j]>key`).

Sehingga untuk melakukan pengurutan turun dapat dilakukan dengan cara sebaliknya, pada kondisi:

```
if(MyArray[j] > MyArray[j + i])
```

tanda lebih besar (>) diganti dengan tanda lebih kecil (<).

Best case

- Array sudah dalam keadaan terurut naik
- Data ke-k yang akan diurutkan dibandingkan sebanyak satu kali dengan data ke-(k-1)
- Loop terdalam tidak pernah dieksekusi
- Jumlah pergeseran (*movement*) $\rightarrow M = 0$.
- Jumlah pembandingan key (*comparison*) $\rightarrow C = n-1$

Worst case

- Array dalam urutan kebalikannya.
- Loop terdalam dieksekusi sebanyak p-1 kali, untuk $p = 2, 3, \dots, n$
- Jumlah pergeseran
 $M = (n-1) + (1 + 2 + \dots + n-1)$
 $M = ((n-1) + n * (n-1) / 2)$
- Jumlah pembandingan key
 $C = (1 + 2 + \dots + n-1) = n * (n-1) / 2$

5.3 Metode Pengurutan Data Tingkat Lanjut

Beberapa metode pencarian lanjut diantaranya sebagai berikut:

1. *Shell Sort*
2. *Merge Sort*
3. *Quick Sort*

5.3.1 *Shell Sort*

Metode pengurutan shell sort dibuat oleh Donald Shell tahun 1959. Nama lain dari algoritma ini adalah *dimishing increment sort*. Algoritma pengurutan ini dibuat untuk memperbaiki kelemahan dari metode *insertion sort*. Tujuan utamanya adalah menghindari pemindahan data dalam jumlah banyak. Caranya dengan membandingkan elemen-elemen yang letaknya berjauhan dengan jarak atau gap tertentu, kemudian secara perlahan melakukan hal serupa dengan gap yang lebih kecil, hingga akhirnya menuju ke gap 1.

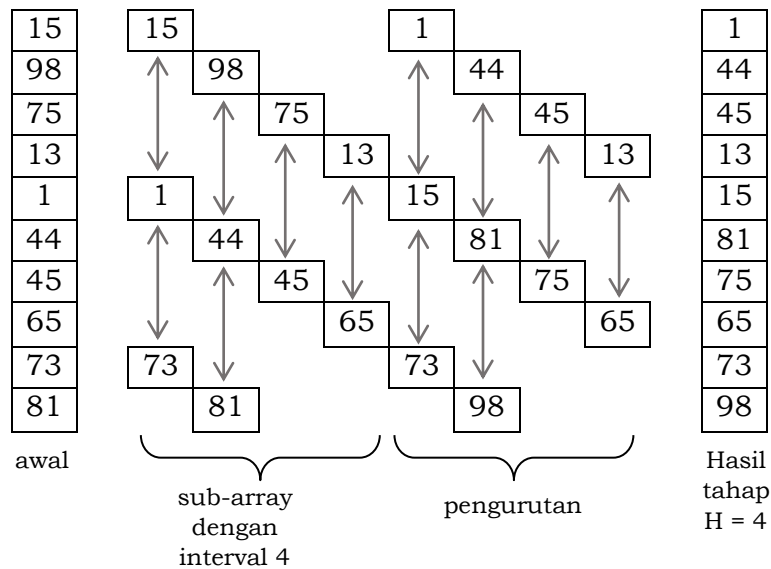
Prinsip pemilihan nilai awal gap adalah bebas, namun yang terpenting adalah gap yang terakhir berupa 1. Jika jumlah data dinyatakan dalam N , maka gap pertama disarankan sebesar $N-2$. Gap kedua dapat bernilai setengah dari gap pertama, dan seterusnya sampai gap bernilai 1.

Donal Knuth menyarankan penggunaan nilai gap 1,4,13,40,121 dan seterusnya sebagai interval dari pengurutan. Angka tersebut berdasarkan rumus $3h+1$ dengan h menyatakan nilai gap sebelumnya. Tabel 5.1 berikut ini akan memperlihatkan hubungan angka-angka dalam penanganan Shell Sort.

Tabel 5. 1 Nilai gap pada shell sort

H	$3H+1$	$(H-1)/3$
1	4	
4	13	1
13	40	4
40	121	13
121	364	40
364	1093	121
1093	3280	364
3280	9841	1093

Ilustrasi pengurutan data menggunakan *Shell sort* ditunjukkan pada Gambar 5.26. Gambar tersebut memperlihatkan keadaan setelah H bernilai 4. Nilai 4 diperoleh dari kondisi $H > N/3$, dengan N menyatakan jumlah data (yaitu 10).



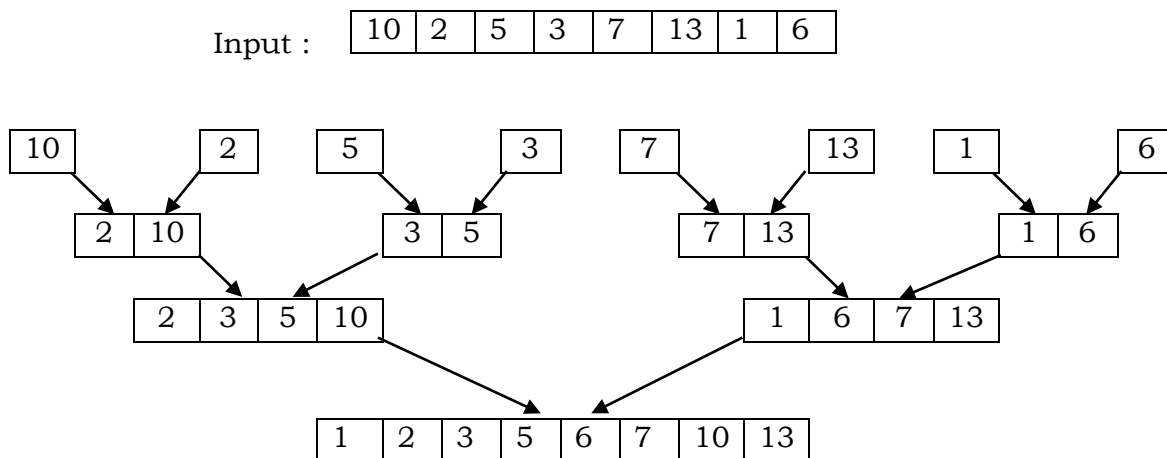
5.3.2 Merge Sort

Merge sort merupakan algoritma dengan kompleksitas waktu lebih baik daripada *Bubble sort*. Jika *bubble sort* memiliki kompleksitas waktu berupa $O(N^2)$ maka *merge sort* memiliki kompleksitas waktu sebesar $O(N \log N)$. Apabila kita simulasikan dengan 10.000 data maka $N^2 = 100.000.000$, sedangkan $N \log N = 80.000$, sangat jauh sekali perbedaannya.

Prinsip dasar *merge sort* adalah menggabungkan dua buah array yang sudahurut, alurnya dapat dilihat sebagai berikut:

1. Bagi n elemen menjadi 2 bagian, masing-masing berisi $n/2$ elemen.
2. Urutkan masing-masing bagian.
3. Gabungkan 2 bagian sehingga menjadi larik yang terurut.

Perhatikan Gambar 5.28 yang memberikan ilustrasi proses dari *merge sort*.



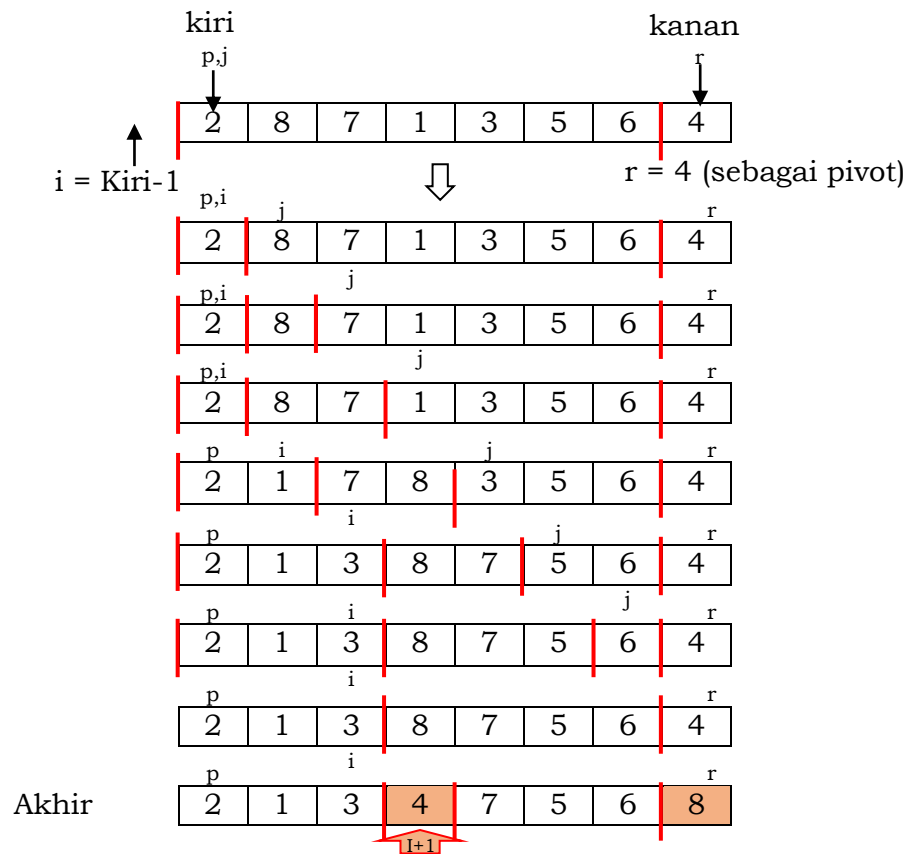
Gambar 5. 28 Ilustrasi Merge sort

Worst case maupun *average case* pada *Merge sort* berupa $O(N \log N)$. jika dibandingkan dengan *Bubble sort*, *Insertion sort* dan *Selection sort*, maka *Merge sort* lebih cepat.

5.3.3 Quick Sort

Metode pengurutan data ini versi dasarnya ditemukan oleh C.A.R. Hoare pada tahun 1960, namun secara formal baru diperkenalkan tahun 1962. Metode penyelesaiannya menggunakan pendekatan rekursif yang didasarkan pada strategi *divide-and-conquer*. Mekanismenya sebagai berikut:

1. Array dipartisi menjadi dua bagian dengan nilai pada bagian kiri selalu lebih kecil daripada nilai pada bagian kanan.
2. Bagian kiri diproses Kembali dimulai pada Langkah 1.
3. Bagian kanan diproses Kembali dimulai pada Langkah 1.



Gambar 5. 29 Ilustrasi prosedur pemartisian

Pada Gambar 5.30 dijelaskan ilustrasi proses pengurutan dengan metode quicksort. $I-1$ menjadi indeks untuk pivot. Nilai yang berada di sebelah kiri pivot bernilai lebih kecil daripada nilai pivot, sedangkan yang disebelah kanan pivot bernilai lebih besar daripada nilai pivot.

Beberapa literatur menyatakan bahwa nilai awal pivot dianjurkan tidak berupa nilai pada indeks pertama atau nilai terakhir. Misalnya, pivot diambil dari elemen yang terletak di tengah array.

Worst case pada *quicksort* terjadi ketika data sudah terurutkan. Pada keadaan itu, perbandingan yang dilakukan sebanyak $n(n-1)/2$ kali. Sehingga kompleksitas algoritmanya $O(N^2)$. *Average case* terjadi ketika array terpartisi di tengah dengan kompleksitas $O(N)$.

5.4 Standard Library `std::sort()`

Seperti telah diuraikan pada bab *searching* sebelumnya, dalam praktek pemrograman menggunakan C++ yang sesungguhnya seorang pemrogram tidak harus mengimplementasikan sendiri algoritma sorting untuk keperluan pengurutan data. C++ menyediakan standard library berupa fungsi `std::sort()` untuk keperluan

pengurutan data dalam array atau sortable data structure lain yang lebih kompleks. Definisi fungsi ini terdapat dalam header <algorithm>.

5.5 Studi Kasus

Program untuk membandingkan kecepatan pengurutan metode *selection*, *bubble*, dan *insertion*. Data dalam array di-generate secara acak. Tiap metode akan dicatat waktu sebelum dan sesudah proses pengurutan yang kemudian akan diperoleh selisih waktu dalam satuan milisecond. Berikut ini contoh tampilan program waktu berjalan.

```
Perbandingan Metode Pengurutan
Data Acak:
41 44 43 12 89 49 57 88 76 90 46 11 61 6 45 33 59 97 3 42 79 78 83 45
28

Bubble Sort
data terurut:
3 6 11 12 28 33 41 42 43 44 45 45 46 49 57 59 61 76 78 79 83 88 89 90
97
waktu: 5 milisecond
-----

Selection Sort
data terurut:
3 6 11 12 28 33 41 42 43 44 45 45 46 49 57 59 61 76 78 79 83 88 89 90
97
waktu: 4 milisecond
-----

Insertion Sort
data terurut:
3 6 11 12 28 33 41 42 43 44 45 45 46 49 57 59 61 76 78 79 83 88 89 90
97
waktu: 2 milisecond
-----
```

Penyelesaian:

```
/*
 * Program Perbandingan Metode Sorting:
 * Bubble, Selection, dan Insertion
 *
 * Bayu Setiaji 2013
 * http://yipsoft.com
 */

#include <iostream>
#include <ctime>
#include <windows.h>

using namespace std;
```

```

const int SIZE{25};
const int BUBBLE{0};
const int SELECTION{1};
const int INSERTION{2};

int nsrc[SIZE];
int ndata[SIZE];

void init_data();
void load_data();
void view_data();
void bubsort();
void selsort();
void inssort();
void run_sort(int);

int main(void) {
    cout << "Perbandingan Metode Pengurutan\n";
    init_data();
    load_data();

    cout << "Data Acak:\n";
    view_data();
    cout << "\n\n";

    // Bubble Sort
    cout << "Bubble Sort\n";
    load_data();
    run_sort(BUBBLE);

    // Selection Sort
    cout << "Selection Sort\n";
    load_data();
    run_sort(SELECTION);

    // Insertion Sort
    load_data();
    cout << "Insertion Sort\n";
    run_sort(INSERTION);
    return 0;
}

void init_data(void)
{
    srand(time(NULL));
    for(int i = 0; i < SIZE; ++i)
        nsrc[i] = rand() % 100;
}

void load_data(void)
{

```

```

        for(int i = 0; i < SIZE; ++i)
            ndata[i] = nsrc[i];
    }

    void view_data(void)
    {
        for(int i = 0; i < SIZE; ++i)
            printf("%d ", ndata[i]);
        cout << endl;
    }

    void bubsort(void)
    {
        for(int i = 0; i < SIZE - 1; ++i)
        {
            for(int j = 0; j < SIZE - i - 1; ++j)
            {
                if(ndata[j] > ndata[j + 1])
                {
                    int tmp = ndata[j];
                    ndata[j] = ndata[j + 1];
                    ndata[j + 1] = tmp;
                    Sleep(10);
                }
                Sleep(10);
            }
            Sleep(10);
        }
    }

    void selsort(void)
    {
        for(int i = 0; i < SIZE; ++i)
        {
            int min = i;
            for(int j = i; j < SIZE; ++j)
            {
                if(ndata[j] < ndata[min])
                {
                    min = j;
                    Sleep(10);
                }
                Sleep(10);
            }

            int tmp = ndata[i];
            ndata[i] = ndata[min];
            ndata[min] = tmp;
            Sleep(10);
        }
    }
}

```

```

void inssort(void)
{
    for(int i = 1; i < SIZE; ++i)
    {
        int m = ndata[i];
        s = i;
        while(s >= 0 && m < ndata[s - 1])
        {
            ndata[s] = ndata[--s];
            Sleep(10);
        }
        ndata[s] = m;
        Sleep(10);
    }
}

void run_sort(int m)
{
    auto t1 = time(NULL);
    switch(m)
    {
        case 0: bubsort(); break;
        case 1: selsort(); break;
        case 2:
            default: inssort(); break;
    }
    auto t2 = time(NULL);
    auto t = t2 - t1;

    cout << "data terurut:\n";
    view_data();
    cout << "waktu: " << t << " milisecond\n";
    cout << "-----\n\n";
}

```

5.6 Latihan

1. Gambarkan langkah pengurutan data pada array sebanyak elemen berikut :
8,4,1,6,20,9,14,17
Pengurutan akan diurutkan menaik (*ascending*) dengan metode *Bubble Sort* dimulai dari belakang.
2. Di antara 6 metode pengurutan dasar dan lanjutan yang sudah dibahas, manakah yang paling cepat? Mengapa?
3. Berdasarkan proses perbandingan dan pertukarannya, manakah yang lebih cepat antara metode *bubble*, *selection* dan *insertion* untuk menangani data yang hampir terurut? Mengapa?

Bila memungkinkan, modifikasilah proses pengurutan dari beberapa metode yang sudah dibahas di atas ke dalam bentuk rekursif.

6

Struct

Sub CPMK

Mahasiswa mampu merancang metode struktur (struct) untuk pengelolaan data

Dalam bab ini akan dibahas tentang pengertian struct, deklarasi struct, cara pengaksesan struct, dan *array of struct*. Terakhir akan diberikan contoh studi kasus yang dilengkapi dengan soal latihan.

6.1 Pengertian

6.1.1 Struct

Struct merupakan struktur data bentukan yang berisi berbagai jenis data. Karena suatu objek biasanya mempunyai beberapa atribut yang perlu dicatat, maka kita bisa menggunakan struktur data ini. Seseorang yang bernama John setidaknya memiliki atribut seperti nama, tanggal lahir, pekerjaan, status, dan alamat yang tertera di KTP. Ketika John menjadi mahasiswa, ia memiliki atribut lain seperti jumlah mahasiswa, jurusan, program studi, mata kuliah yang diambil, nilai mata kuliah, IPK dan sebagainya. Jika John menjadi atlet, mungkin atributnya adalah olahraga, daftar prestasi, dan jadwal pertandingan tahun ini. Oleh karena itu, objek yang sama dapat memiliki atribut yang berbeda tergantung pada sudut di mana objek tersebut ditampilkan. Untuk mengumpulkan semua jenis atribut atau data yang berbeda, Anda dapat menggunakan struktur data yang disebut struct (terkadang disebut kumpulan data dalam bahasa lain).

Struktur adalah kumpulan elemen data dan fungsi yang digabungkan menjadi satu kesatuan. Setiap elemen data disebut bidang atau elemen struktural. Secara umum, jumlah field dalam suatu struktur tidak dibatasi, tetapi untuk objek, field yang perlu direkam dipilih sesuai dengan kebutuhan Anda. Yang perlu Anda ketahui adalah bagaimana menentukan bidang mana yang akan direkam dalam suatu objek. Bidang data dapat memiliki tipe data yang sama atau tipe data yang berbeda. Meskipun bidang-bidang ini memiliki struktur terpadu, masing-masing bidang ini masih dapat diakses secara individual.

Struct digunakan untuk mengelompokkan beberapa informasi yang berkaitan. Dengan sebuah satu kesatuan. Field - field tersebut digabungkan menjadi satu struktur dengan tujuan untuk memudahkan.

Array dan struct mempunyai persamaan serta perbedaan. Persamaan antara array dan struct adalah alokasi memori untuk elemen - elemennya sudah ditentukan sebelum program dijalankan (statis). Sedangkan perbedaan antara array dengan struct adalah sebagai berikut:

1. Array adalah struktur data yang tipe data dari elemen - elemennya sama (homogen) dan elemen - elemennya diakses atau diidentifikasi menggunakan index.
2. Struct adalah struktur data yang tipe data dari elemen - elemennya boleh tidak sama (heterogen) dan elemen - elemennya diakses atau diidentifikasi menggunakan identifier atau nama variabel.

6.1.2 Class

Dalam bahasa C++ class adalah sebuah struktur data yang sama seperti struct. Perbedaannya adalah pada aksesibilitas field yang ada di dalamnya. Istilah class biasa digunakan dalam Object Oriented Programming (OOP). Untuk pembahasan materi selanjutnya hanya akan focus pada penggunaan struct saja.

6.2 Deklarasi

Pendeklarasian struct selalu diawali dengan kata kunci struct yang diikuti dengan nama dari struct. Field atau anggota yang dikumpulkan dalam sebuah struct diletakkan di antara tanda kurung kurawal buka { dan kurung kurawal tutup }, kemudian diakhiri dengan tanda titik koma (;).

Berikut ini adalah bentuk umum pendeklarasian struct:

```
struct>NamaStruct {  
    tipe field1;  
    tipe field2;  
    ...  
};  
  
NamaStruct obj;
```

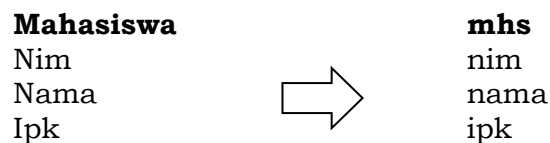
Dari contoh deklarasi di atas, dapat dilihat bahwa pendeklarasian field – field yang ada dalam struct pada dasarnya sama persis seperti deklarasi variabel biasa yang sudah dipelajari.

Struct dapat diartikan juga sebagai sebuah tipe data baru yang nantinya akan digunakan sebagai tipe oleh variabel atau objek. Potongan kode di atas menunjukkan bahwa obj adalah objek (variabel) yang bertipe>NamaStruct.

Berikut ini adalah contoh pendeklarasian struct Mahasiswa dengan anggota berupa nim,nama, dan ipk:

```
struct>Mahasiswa {  
    string nim;  
    string nama;  
    double ipk;  
};  
Mahasiswa mhs;
```

Ilustrasi untuk deklarasi di atas adalah sebagai berikut:



Struct dapat diibaratkan sebagai sebuah cetakan sedangkan objek adalah benda bentukan hasil cetakan sehingga memiliki bentuk yang identik. Proses pembentukan objek ini biasa disebut sebagai instansiasi.

6.3 Pengaksesan

Telah dijelaskan sebelumnya bahwa struct adalah sebuah cetakan yang tidak dapat digunakan secara langsung sehingga dibutuhkan objek hasil bentukan struct tersebut untuk dapat mengakses anggotanya. Cara pengaksesan anggota adalah sebagai berikut:

```
obj.field;
```

Berikut ini adalah contoh memberikan nilai untuk objek mhs dari struct Mahasiswa yang sudah dideklarasikan sebelumnya.

```
mhs.nim = "02.11.0127";  
mhs.nama, = "Ami";  
mhs.ipk = 3.47;
```

Contoh kode berikut ini digunakan untuk menampilkan nilai anggota dari objek mhs:

```
cout << "NIM : " << mhs.nim;  
cout << "Nama: " << mhs.nama;  
cout << "IPK : " << mhs.ipk;
```

Output potongan kode di atas adalah sebagai berikut:

```
NIM : 02.11.0127  
Nama: Ami  
IPK : 3.47
```

6.3.1 Pengaksesan Struct Bersarang

Sebuah struct memungkinkan dapat memuat anggota dengan tipe struct yang lain. Sebagai contoh adalah struct Mahasiswa dapat memiliki anggota berupa struct MataKuliah.

Berikut ini adalah contoh deklarasi struct MataKuliah dan Mahasiswa:

```
struct MataKuliah {  
    string kode;  
    string nama;  
    int bobot;  
};  
struct Mahasiswa {  
    string nim;  
    string nama;  
    MataKuliah mata_kuliah;  
};  
Mahasiswa mhs;
```

Berdasar deklarasi di atas, untuk mengakses anggota pada objek bentukan struct MataKuliah harus melewati object mhs yang di dalamnya terdapt deklarasi anggota berupa objek mkdiambil hasil bentukan dari struct MataKuliah. Berikut ini adalah cara pengisian nilai untuk objek mkdiambil:

```
mhs.mata_kuliah.kode = "ST015";  
mhs.mata_kuliah.nama = "Struktur Data";  
mhs.mata_kuliah.bobot = 4;
```

Untuk menampilkan dapat dilakukan dengan cara yang sama, seperti berikut ini:

```
cout << "Kode MK   : " << mhs.mata_kuliah.kode;  
cout << "Nama MK   : " << mhs.mata_kuliah.nama;  
cout << "Bobot SKS: " << mhs.mata_kuliah.bobot;
```

6.4 Array of Struct

Sama halnya dengan tipe – tipe data primitif struct juga dapat dibuat array. Sebagai ilustrasi suatu ketika dibutuhkan sebuah program yang dapat menyimpan 100 data mahasiswa di mana informasi tiap mahasiswa didefinisikan dalam sebuah struct sehingga dibutuhkan 100 objek yang dibentuk dari struct tersebut. Bagaimana bila jumlahnya menjadi 1000 atau 10000? Akan menjadi hal yang sangat menyulitkan jika harus mendeklarasikan 100, 1000, atau 10000 objek. Untuk itu cukup dibuat 1 objek sebagai array dengan ukuran yang ditentukan sesuai kebutuhan.

6.4.1 Deklarasi

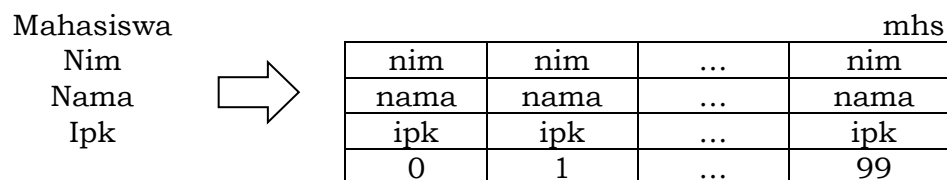
Pendeklarasia array struct dapat dilakukan dengan cara membuat objek bentukan dari sebuah struct menjadi array. Bentuk umum deklarasinya adalah sebagai berikut.

```
NamaStruct obj[ukuran];
```

Contohnya adalah pendeklarasian objek mhs dari struct Mahasiswa dengan ukuran 100 seperti berikut ini:

```
Mahasiswa mhs[100];
```

Ilustrasi visual dari contoh deklarasi di atas adalah sebagai berikut:



Ada 100 objek mhs terbentuk yang masing – masing diidentifikasi dengan nomor index dari 0 sampai 99. Berdasar ilustrasi di atas, dapat dianalogikan bahwa data

tersimpan dalam sebuah tabel. Tiap field dalam struct melambangkan kolom – kolomnya, sedangkan array data yang tersimpan adalah barisnya.

6.4.2 Pengaksesan

Tiap objek diidentifikasi dengan nomor index sehingga pengaksesan anggota tiap objek harus merujuk pada nomor index yang sesuai. Berikut ini adalah bentuk umum pengaksesan anggota dalam array objek dari struct:

```
obj[index].field;
```

Bentuk tersebut berlaku juga untuk pengisian nilai pada object struct. Berikut ini adalah contoh potongan kode untuk memasukkan 100 data mahasiswa ke dalam objek mhs:

```
// semua input dibaca sebagai string, sehingga
// membutuhkan variabel bantu tipe string untuk ipk
string str_ipk;
for(int i = 0; i < 100; ++i) {
    cout << "Data mahasiswa ke-" << i << endl;
    cout << "NIM : ";
    getline(cin, mhs[i].nim);
    cin << "Nama: ";
    getline(cin, mhs[i].nama);
    cout << "IPK : ";
    getline(cin, str_ipk);
    // konversi dari string ke double menggunakan fungsi stod()
    mhs[i].ipk = stod(str_ipk);
}
```

6.5 Studi Kasus

Program buku alamat sederhana untuk menyimpan data yang memuat kode, nama, alamat, dan nomor telepon. Program memiliki menu yang berisi:

4. Penambahan data
5. Penampilan semua data
6. Keluar program

Penyelesaian:

```
/*
 * program buku alamat sederhana
 * menyimpan data alamat: nama, alamat, nomor telepon
 */

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

struct PhoneBook
{
    string name;
    string address;
    string number;
};

// ukuran array
const int SIZE = 100;
// counter, menghitung jumlah data yang tersimpan
int counter = 0;

// array untuk menyimpan data
PhoneBook pb[SIZE];

void add_contact();
void show_contact();
char get_menu();

int main()
{
    while(true)
    {
        char select = get_menu();
        if(select == '1')
            add_contact();
        else if(select == '2')
            show_contact();
    }
}
```

```

        else if(select == '3')
            break;
    }
    return 0;
}

void add_contact()
{
    system("cls");
    cout << "Add New Contact\n";

    cout << "- Name : ";
    getline(cin, pb[counter].name);

    cout << "- Address: ";
    getline(cin, pb[counter].address);

    cout << "- Phone : ";
    getline(cin, pb[counter].number);

    ++counter;
}

void show_contact()
{
    system("cls");
    cout << "Show Contact\n";
    cout << "-----\n";
    cout << setw(4) << "#";
    cout << setw(15) << "Name";
    cout << setw(30) << "Address";
    cout << setw(13) << "Number";
    cout << endl;
    cout << "-----\n";

    for(int i = 0; i < counter; ++i)
    {
        cout << setw(4) << i + 1;
        cout << setw(15) << pb[i].name;
        cout << setw(30) << pb[i].address;
        cout << setw(13) << pb[i].number;
        cout << endl;
    }

    cout << "-----\n";
    system("pause");
}

```



```

}

char get_menu()
{
    system("cls");
    cout << "Phone Book\n";
    cout << " [1] Add Contact\n";
    cout << " [2] Show Contact\n";
    cout << " [3] Exit\n";
    cout << "Select: ";
    string sel;
    getline(cin, sel);

    return sel[0]; // hanya mengambil karakter ke-0
}

```

6.6 Latihan

1. Jelaskan mengapa anggota struct tidak dapat diakses secara langsung menggunakan nama struct tersebut?
2. Dari contoh kasus di atas, tambahkan menu untuk fungsi pencarian data dan pengeditan data.
3. Buatlah program untuk menghitung IPK sejumlah mahasiswa yang dimasukkan.

Ketentuan:

- a. Informasi tiap mahasiswa harus memuat nim, nama, dan mata kuliah diambil.
- b. Informasi tiap mata kuliah harus memuat kode, nama, bobot SKS, dan nilai huruf (A – E)
- c. Tiap mahasiswa dapat mengambil maksimal 5 mata kuliah.
- d. Program harus dapat menampilkan KHS tiap mahasiswa.

Berikut ini adalah contoh tampilan program ketika dijalankan:

```

Program KHS Mahasiswa
Jumlah mahasiswa: 1
-----
1. NIM : 02.11.0127
Nama: John Doe
Jumlah mata kuliah diambil: 3
1. Kode: ST068
Nama: Algoritma dan Pemrograman
Bobot SKS: 4
Nilai: B
-----
2. Kode: ST021

```

Nama: Pemrograman

Bobot SKS: 2

Nilai: A

3. Kode: ST015

Nama: Struktur Data

Bobot SKS: 4

Nilai: A

```
+-----+
| Kartu Hasil Studi |
+-----+
| NIM : 02.11.0127 |
| Nama: John Doe |
+-----+
| KDMK Mata Kuliah SKS Nilai Bobot |
+-----+
| ST068 Algoritma dan Pemrograman 4 B 12 |
| ST021 Pemrograman 2 A 8 |
| ST015 Struktur Data 4 A 16 |
+-----+
| Jumlah SKS Total 10 |
| Indeks Prestasi Kumulatif 3,6 |
+-----+
```

7

Stack

Sub CPMK

Mahasiswa mampu merancang metode tumpukan (stack) untuk pengelolaan data dalam larik

Dalam bab ini akan dibahas tentang metode tumpukan (stack) data. Pembahasan diawali dengan pengertian kemudian akan dibahas operasi push dan pop. Masing-masing pembahasan operasi disertai dengan ilustrasi visual untuk menggambarkan tiap prosesnya. Kemudian dibahas juga tentang fungsi empty, clear dan show. Pada akhir pembahasan dilanjutkan studi kasus tentang konversi basis bilangan desimal ke biner.

7.1 Pengertian

Tumpukan adalah struktur data yang memungkinkan Anda memasukkan dan mengambil data dari ujung yang disebut atas. Seperti namanya, tumpukan bisa diibaratkan sekumpulan data yg seolah-olah terdapat suatu data yg diletakkan pada atas data yg lain. Dalam kehidupan sehari-hari kita mungkin tak jarang menciptakan sebuah tumpukan atau stack. Tumpukan bisa ditambah dan diambil isinya menggunakan cara yang sama.

Misalkan sebuah tumpukan piring atau buku. Ketika ditambahkan buku baru maka cara paling gampang merupakan yakni diletakkan pada atas buku-buku lain yg telah terdapat lebih dulu. Semakin ditambah buku baru yang diletakkan maka semakin atas posisinya. Demikian saat buku-buku itu diambil, maka yg posisinya paling atas lah yang paling gampang diambil dahulu dibanding yg bawahnya.

Ada kalanya diperlukan struktur data yg memungkinkan penambahan & pengurangan elemennya dilakukan dalam bagian ujung puncak (top), baik depan atau belakang, bukan pada tengah. Hal ini merepresentasikan suatu tumpukan. Dalam suatu tumpukan yg diletakkan terakhir lah yang akan pada diambil pertama. Sifat ini dikenal menjadi kata LIFO (*Last In First Out*).

Penyajian stack bisa memakai tipe data array. Namun, tipe ini kurang sempurna lantaran banyaknya elemen pada array merupakan statis, sedangkan pada stack banyaknya elemen sangat bervariasi atau dinamis. Pada suatu waktu berukuran stack akan sama menggunakan berukuran array. Jika diteruskan menambah data maka akan terjadi overflow. Oleh karenanya perlu dibubuhi pencatat posisi ujung stack.

Selain memakai array stack bisa juga tersaji menggunakan linked list yg jumlah isinya lebih dinamis. Namun, pada bab ini stack akan tersaji pada array lantaran linked list merupakan struktur data yg wajib dibahas secara terpisah. Pembahasan linked list & penggunaannya (terasuk buat penyajian stack) akan diterangkan dalam bab selanjutnya.

7.2 Operasi

Dalam stack ada beberapa operasi yang dapat dilakukan. Operasi utama dalam stack adalah push (menambah) dan pop (mengambil). Selain itu terdapat juga operasi – operasi lain sebagai pelengkap. Berikut pengertian dari masing-masing operasi pada stack:

1. Inisialisasi

Pengesetan nilai top dengan 0, karena array dalam bahasa C/C++ dimulai dari 0, yang berarti bahwa data stack adalah KOSONG.

2. Top

Variabel penanda bertipe int dalam stack yang menunjukkan elemen teratas data stack sekarang. Top akan selalu bergerak hingga mencapai MAX of STACK yang menyebabkan stack PENUH!

3. Push

Digunakan untuk menambah data pada stack melalui tumpukan paling atas.

4. Pop

Digunakan untuk menghapus data pada stack melalui tumpukan paling atas.

5. Clear

Digunakan untuk mengosongkan stack.

6. IsEmpty

Digunakan untuk mengecek apakah stack kosong. Fungsi ini menghasilkan nilai Boolean, benar (nilai 1) jika tumpukan dalam keadaan kosong.

7. IsFull

Digunakan untuk mengecek apakah stack penuh.

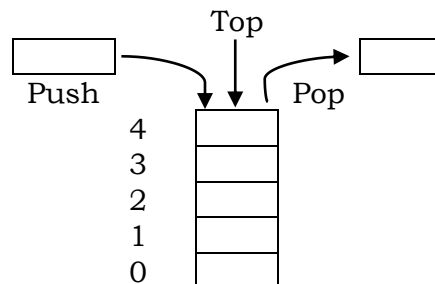
8. Print

Digunakan untuk mencetak seluruh isi stack.

Sebagai bahan untuk mempermudah ilustrasi pembahasan operasi stack, diberikan sebuah deklarasi sebagai berikut.

```
constexpr int size{5}
char stack[size];
int top{0};
```

Berdasar deklarasi di atas terdapat sebuah stack bertipe char dan memiliki ukuran 5. Selain itu didapat pula variabel top dengan nilai awal 0, yang akan digunakan sebagai pencatat posisi ujung stack. Ilustrasi stack dapat dilihat pada Gambar 7.1



Gambar 7. 1 Ilustrasi stack

7.2.1 Push

Push adalah operasi memasukkan elemen ke dalam stack. Urutan dalam operasi push adalah:

4. Memeriksa apakah stack penuh, jika penuh maka operasi dihentikan, tetapi jika tidak maka:
5. Elemen baru dimasukkan ke posisi yang ditunjuk oleh nilai **top** sekarang
6. Naikkan nilai **top**, yang berarti nilai top menunjuk satu tingkat di atasnya.

Berikut ini adalah kode untuk fungsi push.

```
void push(char e) {  
    if(top == size)  
    {  
        cout << "stack overflow\n";  
        return;  
    }  
    stack[top++] = e;  
}
```

7.2.2 Pop

Pop adalah operasi untuk mengeluarkan elemen dari stack. Urutan dalam operasi pop adalah:

1. Memeriksa apakah stack kosong, jika kosong maka operasi dihentikan, tetapi jika tidak maka:
2. Nilai top diturunkan satu tingkat.
3. Elemen stack pada posisi top yang sekarang dikeluarkan.

Berikut ini adalah kode untuk fungsi pop.

```
char pop()  
{  
    if(top == 0)  
    {  
        cout << "stack underflow\n";  
        return('\0');  
    }  
    return stack[--top];  
}
```

7.2.3 Clear

Clear adalah operasi untuk mengosongkan stack dengan cara memberikan nilai 0 untuk top.

Berikut ini adalah kode untuk fungsi clear.

```
void clear()
```

```

{
    top = 0;
}

```

7.2.4 Show

Show adalah operasi untuk menampilkan keseluruhan isi stack. Isi stack ditampilkan dari posisi paling bawah sampai sebelum top. Berikut ini adalah kode untuk fungsi show.

```

void show()
{
    int i;
    for(i = 0; i < top; i++)
        cout << stack[i];
}

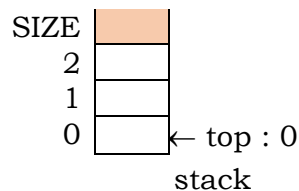
```

Beberapa deklarasi dan definisi fungsi di atas dapat disesuaikan dengan tipe data yang digunakan pada stack.

7.2.5 Simulasi

Supaya lebih mudah memahami konsep dari tumpukan, berikut ini disajikan ilustrasi visual simulasi operasi stack berdasar deklarasi dan definisi fungsi sebelumnya. Top atau puncak akan diilustrasikan dengan SIZE yang menyatakan kondisi pada posisi puncak dalam tumpukan.

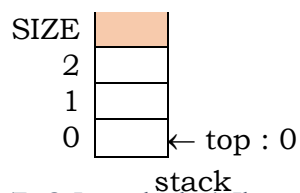
Kondisi awal stack (SIZE: 3) masih kosong. Top diilustrasikan sebagai panah ke kiri.



Gambar 7. 2 Ilustrasi Stack Kondisi Awal

Kemudian dikenai operasi berturut – turut:

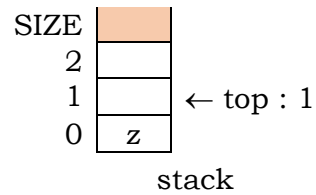
1. pop()



Gambar 7. 3 Langkah 1 Ilustrasi Stack

Operasi pop tidak akan dijalankan karena stack dalam kondisi kosong. Muncul pesan “stack underflow”.

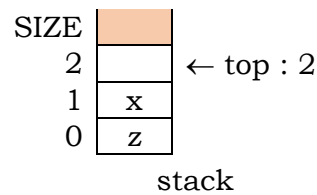
2. push('z')



Gambar 7. 4 Langkah 2 Ilustrasi Stack

'z' menempati posisi 0 dan posisi top naik menunjuk 1. Nilai top dapat juga diartikan sebagai jumlah elemen yang sudah masuk.

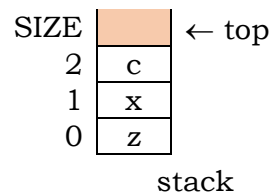
3. push('x')



Gambar 7. 5 Langkah 3 Ilustrasi Stack

Top menunjuk ke 2 artinya isi stack sekarang ada 2 elemen.

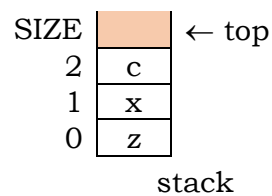
4. push('c')



Gambar 7. 6 Langkah 4 Ilustrasi Stack

Top menunjuk SIZE artinya stack sudah penuh.

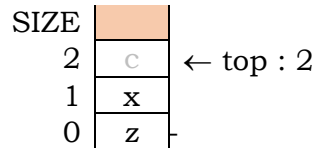
5. push('v')



Gambar 7. 7 Langkah 5 Ilustrasi Stack

Operasi push tidak akan dijalankan karena stack sudah penuh. Muncul pesan “stack overflow”.

6. pop()

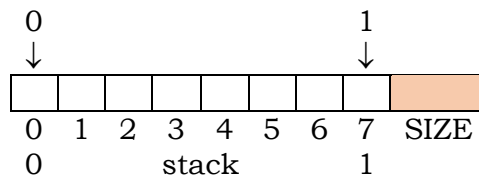


Gambar 7. 8 Langkah 6 Ilustrasi Stack

Top turun menunjuk ke posisi 2. Elemen di posisi 2 dikeluarkan.

7.3 Double Stack

Double stack merupakan bentuk pengembangan dari *single stack* dengan maksud untuk menghemat memori. Prinsip dari *double stack* adalah dalam satu array terdapat dua stack. Dalam *double stack* terdapat 2 top yang mewakili tiap – tiap stack.



Gambar 7. 9 Contoh Double Stack

Seperti terlihat dalam ilustrasi Gambar 7.9 top-0 digunakan untuk stack-0 dan top-1 untuk stack-1.

Bedasar ilustrasi di atas, maka stack dapat dideklarasikan seperti berikut.

```
constexpr int size{8};
char stack[size];
int top1{0};
int top2{size - 1};
```

7.3.1 Operasi

Operasi pada *double stack* secara prinsip sama dengan *single stack* yang sudah dibahas sebelumnya. Bedanya pada *double stack* tiap operasi akan dispesifikan untuk stack pertama atau kedua.

1. Push

Operasi push pada double stack sama seperti single stack. Perbedaannya adalah pada pemeriksaan kondisi penuh stack. Pada double stack, dikatakan penuh ketika $top1 > top2$, sehingga fungsi push() didefinisikan seperti berikut:

```
void push(char e, int s) {
    if(top1 > top2) {
        cout << "stack overflow";
        return;
    }
    if(s == 1) {
        stack[top1++] = e;
        return;
    }
    if(s == 2) {
        stack[top2--] = 2;
        return;
    }
    cout << "invalid stack";
}
```

2. Pop

Operasi pop pada double stack sama seperti pada single stack. Perbedaannya pada saat pemeriksaan kondisi kosong untuk stack-2. Stack-2 dikatakan kosong jika top-2 pada posisi $SIZE - 1$. Berikut ini adalah kode untuk fungsi pop():

```
char pop(int s) {
    if(s == 1) {
        if(top1 <= 0) {
            cout << "stack 1 underflow";
            return '\0'; // '\0', null character
        }
        return stack[--top1];
    }
    if(s == 2) {
        if(top2 >= SIZE - 1) {
            cout << "stack 2 underflow";
            return '\0';
        }
        return stack[++top2];
    }
    printf("invalid stack");
    return '\0';
}
```

3. Clear

Operasi clear sama seperti pada single stack. Kode untuk fungsi clear() adalah sebagai berikut.

```
void clear(int s) {
    if(s == 1)
        top1 = 0;
    else if(s == 2)
        top2 = SIZE - 1;
    else
        cout << "invalid stack";
}
```

4. Show

Fungsi show() sama seperti pada single stack. Berikut ini adalah kodenya.

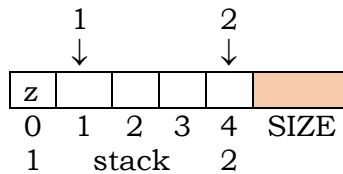
```
void show(int s) {
    if(s == 1) {
        if(top1 <= 0) {
            cout << "empty";
            return;
        }
        for(i = 0; i < top1; i++)
            cout << stack[i] << ' ';
        return;
    }
    if(s == 2) {
        if(top2 >= SIZE - 1) {
            cout << "empty";
            return;
        }
        for(i = SIZE - 1; i > top2; i--)
            cout << stack[i] << ' ';
        return;
    }
    cout << "invalid stack";
}
```

7.3.2 Simulasi

Berikut ini disimulasikan operasi double stack (SIZE: 5) berdasar pada deklarasi dan definisi fungsi sebelumnya.

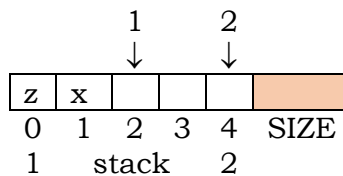
Mula – mula *double stack* dalam kondisi kosong (top1: 0, top2: 4)

1. push('z', 1)



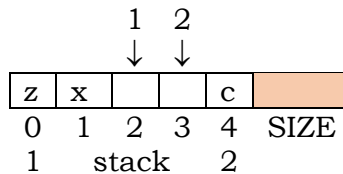
Gambar 7. 10 Langkah 1 Ilustrasi Double Stack

2. push('x', 1)



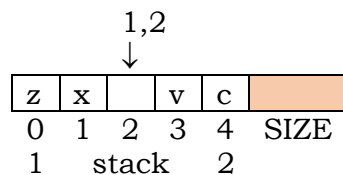
Gambar 7. 11 Langkah 2 Ilustrasi Double Stack

3. push('c', 2)



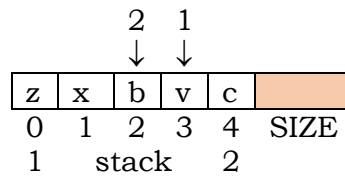
Gambar 7. 12 Langkah 3 Ilustrasi Double Stack

4. push('v', 2)



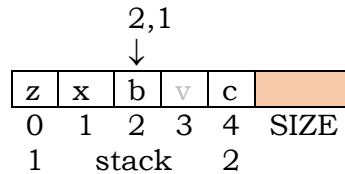
Gambar 7. 13 Langkah 4 Ilustrasi Double Stack

5. push('b', 1)



Gambar 7. 14 Langkah 5 Ilustrasi Double Stack

6. pop(2)



Gambar 7. 15 Langkah 6 Ilustrasi Double Stack

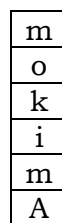
7.4 Penerapan

Penerapan tumpukan pada aplikasi ada banyak, beberapa contohnya sebagai berikut:

a. Membalik string

Jika Anda memproses string dari kiri dan mendorong setiap karakter ke tumpukan, karakter paling kiri akan berada di bagian bawah tumpukan. Kemudian, jika Anda mengambil karakter dalam tumpukan satu per satu dan menempatkannya dari kiri ke kanan, string akan terbentuk dalam urutan terbalik dari karakter aslinya, seperti yang ditunjukkan pada gambar.

String asal : Amikom → diproses ke tumpukan



Gambar 7. 16 Tumpukan untuk membalikan string

Berdasarkan Gambar 7.16 jika kata Amikon diambil hurufnya satu per satu kemudian disusun dari kiri ke kanan menjadi mokimA.

b. Mengkonversi bilangan

Konversi bilangan desimal ke biner, contoh penerapannya akan dibahas pada studi kasus sub bab berikutnya (7.5).

c. Backtracking

Pada suatu aplikasi memungkinkan kita dapat kembali ke proses yang sebelumnya atau setelahnya, misalkan saja dengan mengklik tombol “Back” atau “Undo” dan juga “Next” ataupun “Redo”.

d. Mengevaluasi ekspresi aritmatika

Pada perhitungan aritmatika terdapat *operator precedence* yang harus diutamakan prosesnya. Tumpukan dapat dimanfaatkan untuk memproses perhitungan dengan beberapa prioritas yang berbeda.

e. Memproses pasangan tanda kurung dalam suatu ekspresi

Tanda kurung awal dan akhir dengan berbagai variasi dapat dites valid tidaknya ekspresi tersebut menggunakan pasangan kurungnya.

Misalkan: $(a\{b+c\}[])$ dan $(a\{b+c\})$

Ekspresi pertama dianggap valid karena pasangan kurung tutup sudah sesuai dengan kurung awalnya. Sedangkan ekspresi kedua dianggap tidak valid karena kurung penutup tidak sesuai dengan kurung awalnya.

f. Fungsi rekursif

Pemanggilan fungsi secara rekursif pada umumnya akan diproses menggunakan tumpukan oleh compiler.

7.5 Studi Kasus

Program dengan fungsi untuk konversi basis bilangan desimal ke biner dengan memanfaatkan operasi stack.

Penyelesaian

```
#include <iostream>

constexpr int MAX = 20;

int stack[MAX];
int top = 0;

void push(int e);
int pop();
void dec2bin(int d);

int main(void) {
    int d;
    cout << "bilangan desimal: ";
    cin >> d;

    cout << "konversi biner : ";
    dec2bin(d);

    return 0;
}

void push(int e) {
    if(top >= MAX) {
        cout << "Stack penuh\n";
        return;
    }
    stack[top++] = e;
}

int pop() {
    if(top <= 0) {
        cout << "Stack kosong\n";
        return;
    }
    return stack[--top];
}

void dec2bin(int d) {
    do {
        push(d % 2);
        d = d / 2;
    } while(d > 0);
}
```

```

    } while(d > 0);
    for(int i = top - 1; i >= 0; i--)
        cout << pop();
}

```

7.6 Latihan

1. Diketahui sebuah stack (SIZE: 5, tipe: int) mula – mula dalam kondisi kosong. Kemudian stack dikenai operasi secara berturut – turut:

```

push(8),
push(9),
push(3),
pop(),
push(5),
push(6),
push(4),
push(7),
pop(),
pop().

```

Berapa saja isi stack sekarang?

2. Diketahui sebuah operasi aritmatik:
 $5 + 3 - 2 * 4 / 2$
 Bila operasi aritmatik tersebut disimulasikan dalam operasi stack (push, pop) maka tuliskanlah urutan – urutannya.
3. Buatlah program yang mensimulasikan tumpukan buku. Buku dimodelkan dalam bentuk struct yang memuat judul dan nama pengarang. Ketentuan
 - a. Data tumpukan stack diisi oleh pengguna
 - b. Program harus bisa mensimulasikan langkah pengambilan buku.

Berikut ini adalah contoh tampilan program (setelah isi tumpukan diinput) Ketika dijalankan:

```

Isi tumpukan buku:
5 > "Buku ZXC"
4 > "Buku ABC"
3 > "Buku ASD"
2 > "Buku QWR"
1 > "Buku MNO"
0 > "Buku JKL"

Ambil buku nomor: 3 -> input pengguna

```


Simulasi:

Buku yang diambil: "Buku ASD"

Langkah:

Angkat: "BUKU ZXC"

Angkat: "BUKU ABC"

Angkat: "BUKU ASD"

Ambil : "Buku ASD"

Tumpuk: "BUKU ABC"

Tumpuk: "BUKU ZXC"

Isi tumpukan buku:

4 > "Buku ZXC"

3 > "Buku ABC"

2 > "Buku QWR"

1 > "Buku MNO"

0 > "Buku JKL"

Ambil buku nomor :

8

Queue

Sub CPMK

Mahasiswa mampu merancang metode antrian (queue) untuk pengelolaan data

Dalam bab ini akan dibahas tentang metode *queue* (antrian) data. Pembahasan diawali dengan pengertian kemudian akan dibahas dua operasi utama yaitu *store* dan *retieve*. Masing-masing pembahasan operasi disertai dengan ilustrasi visual untuk menggambarkan tiap prosesnya. Dibahas pula *circular queue* yang disertai simulasinya. Pada akhir materi akan diberikan Studi Kasus beserta Latihan.

8.1 Pengertian

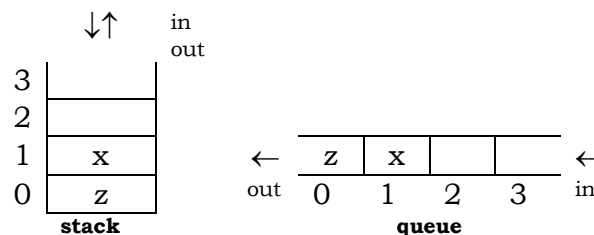
Queue atau antrian merupakan kumpulan data dimana penambahan dan pengambilannya melalui dua jalan yang berbeda. Penambahan elemen hanya bisa dilakukan pada suatu ujung yang disebut dengan sisi belakang (rear), dan penghapusan (pengambilan elemen) dilakukan lewat ujung lain (disebut dengan sisi depan atau front). Contoh *queue* dalam kehidupan sehari – hari misalnya antrian pembayaran di kasir, antrian mobil saat pengisian BBM di SPBU, dll.

Sebagai ilustrasi saat pengisian BBM di SPBU, maka mobil yang datang pertamalah yang akan diisi pertama, demikian seterusnya sampai yang mendapat giliran terakhir adalah yang datang terakhir. Oleh karena itu, struktur data ini mencerminkan konsep antrian yang sering kita alami di dunia nyata.

Hal yang sama juga berlaku pada data, yaitu data yang masuk pertama akan keluar pertama juga dan data yang terakhir masuk akan keluar terakhir. Sifat ini sering disebut dengan istilah FIFO (*First In First Out*).

Dari cara penyajian *queue* hampir sama dengan *stack*. Perbedaannya pada *queue* masuk dan keluarnya data menggunakan jalan yang berbeda. Perbedaan antara *stack* dengan *queue* yaitu jika *stack* atau tumpukan menggunakan prinsip “Masuk terakhir keluar pertama” atau LIFO (*Last In First Out*), maka pada *Queue* atau antrian menggunakan prinsip “Masuk Pertama Keluar Pertama” atau FIFO (*First In First Out*).

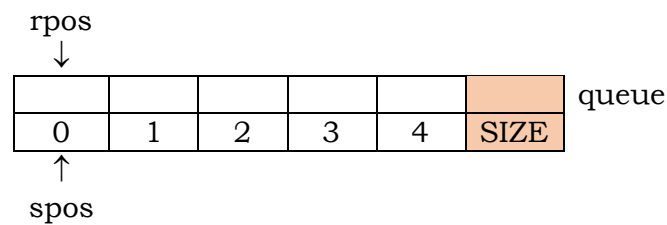
Berikut ini Gambar 8.1 adalah contoh perbedaan *stack* dengan *queue*.



Gambar 8. 1 Ilustrasi perbedaan *stack* dan *queue*

Dalam struktur data *queue* dapat disajikan sebagai sebuah array seperti halnya *stack*. Perbedaannya, pada *stack* proses masuk keluarnya data diacu pada satu penunjuk (top) sedangkan pada *queue* masuk dan keluarnya data diacu pada penunjuk yang berbeda.

Pada *Queue* atau antrian terdapat satu buah pintu masuk di suatu ujung dan satu buah pintu keluar di ujung satunya dimana membutuhkan variabel Head dan Tail (depan/front, belakang/rear). Berikut ini adalah ilustrasi sebuah *queue* berukuran (SIZE) 5. Terdapat 2 penunjuk sebagai head dan tail yaitu spos dan rpos. spos adalah penunjuk yang digunakan untuk proses pemasukan data, dan rpos adalah penunjuk untuk pengeluaran.



Gambar 8. 2 Ilustrasi queue

Ilustrasi Gambar 8.2 di atas dapat disajikan dalam array. Berikut ini adalah deklarasi untuk queue di atas.

```
constexpr int SIZE = 5;
char queue[SIZE];
int spos = 0;
int rpos = 0;
```

Berdasar deklarasi tersebut queue adalah array bertipe char, yang selanjutnya akan digunakan sebagai bahan ilustrasi operasi – operasinya.

8.2 Operasi

Dalam queue terdapat 2 operasi utama yaitu *store* dan *retrieve*. Selain itu juga terdapat operasi – operasi yang lain sebagai pelengkap.

8.2.1 Store

Store adalah operasi pemasukan data atau elemen ke dalam queue. Ada pula yang menyebut proses ini sebagai insert ataupun **enqueue**. Urutan prosesnya adalah sebagai berikut:

1. Memeriksa apakah queue penuh, ditandai dengan posisi spos menunjuk pada posisi SIZE. Jika penuh maka operasi dihentikan, tetapi jika tidak maka
2. Memasukkan elemen pada posisi yang ditunjuk oleh spos sekarang
3. Menaikkan nilai spos dengan 1 sehingga menunjuk di sebelah kanan elemen.

Berikut ini adalah kode untuk fungsi store() void store(char e)

```
{
    if(spos >= SIZE) {
        cout << "queue is full";
        return;
    }
    queue[spos++] = e;
}
```

8.2.2 Retrieve

Retrieve adalah proses pengeluaran elemen dari queue. Operasi ini sering disebut juga sebagai remove atau ***dequeue***. Data yang berada paling depan dalam antrian akan dikeluarkan. Urutan prosesnya adalah:

1. Memeriksa apakah queue dalam kondisi kosong, ditandai dengan spos dan rpos menunjuk posisi yang sama. jika kosong maka operasi dihentikan, tetapi jika tidak maka
2. Memasukkan elemen dari queue
3. Meningkatkan nilai rpos dengan 1 sehingga bergeser ke kanan.

Berikut ini adalah kode untuk fungsi retrieve()

```
char retrieve()
{
    if(rpos == spos) {
        cout << "queue is empty";
        return '\\0'; // null char
    }
    return queue[rpos++];
}
```

8.2.3 Clear

Clear adalah operasi mengosongkan queue. Prosesnya dengan menempatkan spos dan rpos pada posisi 0. Berikut ini adalah kode untuk fungsi clear()

```
void clear()
{
    spos = 0;
    rpos = 0;
}
```

8.2.4 Show

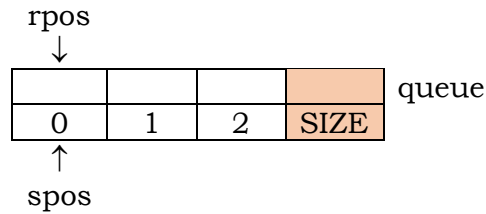
Show adalah operasi untuk menampilkan isi queue. Prosesnya adalah membaca keseluruhan elemen dari posisi rpos sampai sebelum spos. Berikut ini adalah kode untuk fungsi show()

```
void show()
{
    for(int i = rpos; i < spos; ++i)
        cout << queue[i] << ' ';
}
```

8.2.5 Simulasi

Supaya lebih mudah dalam memahami proses antrian, berikut ini diberikan simulasi pengoperasian queue berdasar deklarasi dan definisi fungsi sebelumnya.

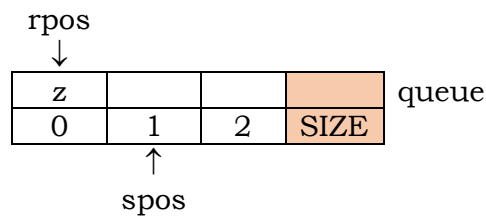
Diberikan sebuah queue (SIZE: 3, tipe: char) mula – mula dalam kondisi kosong.



Gambar 8. 3 Ilustrasi Queue Kondisi Awal

Kemudian secara berurutan diberikan operasi – operasi berikut:

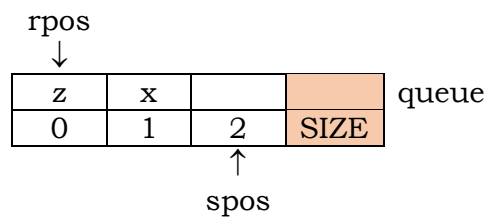
1. store('z')



Gambar 8. 4 Langkah 1 Ilustrasi Queue

'z' masuk dan menempati posisi 0, spos bergeser menunjuk ke posisi 1.

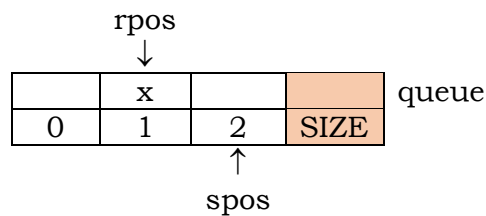
2. store('x')



Gambar 8. 5 Langkah 2 Ilustrasi Queue

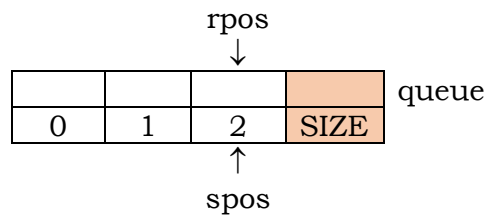
'x' masuk dan menempati posisi 1, spos bergeser menunjuk ke posisi 2.

3. retrieve()



Gambar 8. 6 Langkah 3 Ilustrasi Queue

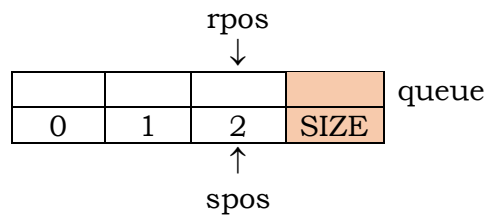
4. retrieve()



Gambar 8. 7 Langkah 4 Ilustrasi Queue

'z' dikeluarkan dan rpos bergeser menunjuk ke posisi 2

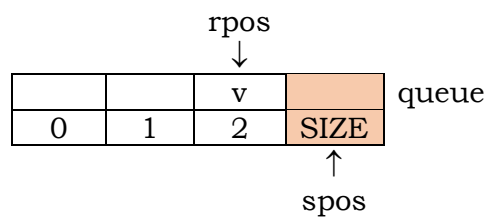
5. retrieve()



Gambar 8. 8 Langkah 5 Ilustrasi Queue

Operasi tidak dijalankan karena queue dalam kondisi kosong, ditandai dengan rpos dan spos menunjuk posisi yang sama. Muncul pesan "queue is empty".

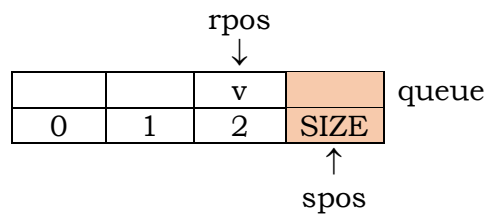
6. store('v')



Gambar 8. 9 Langkah 6 Ilustrasi Queue

'v' masuk dan menempati posisi 2, spos bergeser menunjuk posisi SIZE.

7. store('b')



Gambar 8. 10 Langkah 7 Ilustrasi Queue

Operasi tidak dijalankan karena queue penuh, ditandai dengan spos menunjuk ke posisi SIZE. Muncul pesan “*queue is full*”.

Dari simulasi di atas terlihat bahwa operasi pemasukan dan pengeluaran elemen membuat spos dan rpos bergerak ke arah SIZE. Bandingkan dengan stack di mana top bisa bergerak ke arah size saat pemasukan dan bergerak ke arah 0 saat pengeluaran sehingga satu posisi dalam stack kemungkinan bisa ditempati lebih dari satu kali.

Berbeda dengan stack, satu posisi dalam queue hanya bisa ditempati satu kali saja, disebabkan oleh spos dan rpos yang selalu bergerak ke arah SIZE, sehingga queue hanya dapat digunakan dalam satu siklus saja.

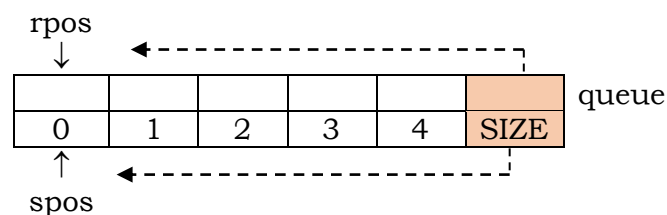
Hal di atas mirip dengan antrian di sebuah customer service, di mana para pelanggan harus mengambil kartu antrian terlebih dahulu dan akan dipanggil satu persatu sesuai nomor urutnya. Setelah selesai kartu nomor antrian tersebut sudah tidak bisa digunakan lagi. Nomor posisi dalam queue dapat dianalogikan sebagai kartu nomor antrian tersebut.

Agar queue dapat digunakan berulang kali maka operasi – operasi yang berkaitan dengan pemasukan dan pengeluaran elemen harus ditambah dengan mekanisme untuk me-recycle penggunaan nomor posisi. Implementasi queue jenis ini menggunakan circular array.

8.3 Circular Queue

Berbeda dengan implementasi queue sebelumnya yang menggunakan linear array maka queue yang menggunakan *circular array* (selanjutnya disebut *circular queue*) dapat digunakan dalam siklus yang berulang. Sebagai ilustrasi di kehidupan nyata adalah revolver pistol yang dapat diisi peluru kembali setelah peluru yang sebelumnya ditembakkan. Peluru yang baru dimasukkan akan ditembakkan pada siklus berikutnya setelah siklus sebelumnya selesai.

Berikut ini Gambar 8.11 merupakan ilustrasi *circular queue*.



Gambar 8. 11 Contoh Circular Queue

Berdasar ilustrasi di atas spos dan rpos akan dikembalikan ke posisi 0 setelah mencapai ujung antrian. Contoh deklarasi circular queue adalah sebagai berikut.

```
constexpr SIZE = 5;
char queue[SIZE];
int rpos = 0;
int spos = 0;
int diff = 0;
```

Deklarasi *circular queue* di atas hampir sama dengan *linear queue*. Perbedaannya pada *circular queue* terdapat variabel tambahan bernama *diff* yang digunakan untuk mengatur apakah proses store dan retrieve berada pada siklus yang sama atau tidak. Pengisian nilai awal 0 terhadap variabel *diff* menandakan bahwa mula - mula queue masih dalam siklus yang sama.

8.3.1 Operasi

Secara umum operasi *circular queue* sama dengan *linear queue* dengan tambahan proses pengaturan siklus antrian dan pemutaran penunjuk keluar masuknya elemen. Di bawah ini adalah penjelasan tiap operasi dengan mengacu pada deklarasi *circular queue* di atas.

1. Store

Sama halnya dalam *linear queue*, sebelum operasi dijalankan akan diperiksa terlebih dahulu apakah queue dalam kondisi penuh atau tidak. Kondisi penuh dalam *circular queue* di tandai dengan spos dan rpos berada pada posisi yang sama dan dalam siklus yang berbeda (*diff* != 0). Berikut ini adalah kode untuk fungsi *store()*.

```
void store(char e)
{
    if(spos == rpos && diff) {
        cout << "queue is full";
        return;
    }

    queue[spos++];
    if(spos >= SIZE) {
        diff = !diff;
        spos = 0;
    }
}
```

Seperti terlihat pada kode di atas, spos akan dikembalikan ke posisi 0 ketika setelah operasi mencapai nilai SIZE, dan nilai diff akan diisi kebalikannya,

yang menandakan bahwa dalam queue sudah terdapat 2 siklus yang berbeda (diff != 0).

2. Retrieve

Sebelum operasi dijalankan akan diperiksa terlebih dahulu apakah queue dalam kondisi kosong atau tidak. Queue dalam kondisi kosong ketika rpos dan spos menunjuk posisi yang sama dan berada pada siklus yang sama (diff = 0). Berikut ini adalah kode untuk fungsi store().

```
char retrieve()
{
    char e;
    if(rpos == spos && !diff){
        cout << "queue is empty";
        return '\\0';
    }

    e = queue[rpos++];

    if(rpos >= SIZE) {
        rpos = 0;
        diff = !diff;
    }
    return e;
}
```

3. Clear

Operasi clear dilakukan dengan memberi nilai 0 pada semua penunjuk. Berikut ini adalah kode untuk fungsi clear()

```
void clear()
{
    spos = 0;
    rpos = 0;
    diff = 0;
}
```

4. Show

Menampilkan keseluruhan isi queue. Berikut ini adalah kode untuk fungsi show().

```
void show()
{
    if(rpos == spos && !diff) {
        cout << "queue is empty";
    }
}
```

```

        return;
    }

    int i = rpos;
    do {
        cout << "%c ", queue[i++];
        if(i >= SIZE)
            i = 0;
    } while( i != spos);
}

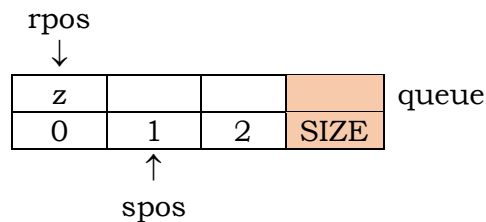
```

8.3.2 Simulasi

Untuk membantu pemahaman tentang circular queue berikut ini disimulasikan operasi –operasi berdasar deklarasi sebelumnya.

Diberikan sebuah circular queue(SIZE: 3, tipe: char) mula – mula dalam kondisi kosong, kemudian diberikan beberapa operasi secara berurutan sebagai berikut:

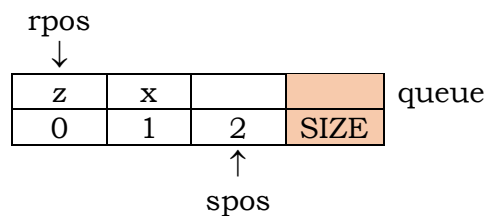
1. store('z')



Gambar 8. 12 Langkah 1 Ilustrasi Circular Queue

'z' masuk dalam queue menempati posisi 0, kemudian spos digeser ke posisi 1.

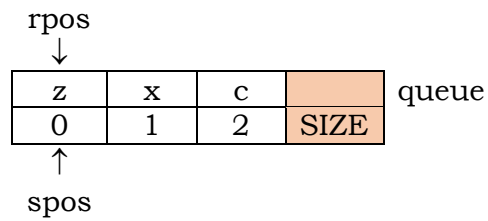
2. store('x')



Gambar 8. 13 Langkah 2 Ilustrasi Circular Queue

'x' masuk, spos bergeser ke posisi 2.

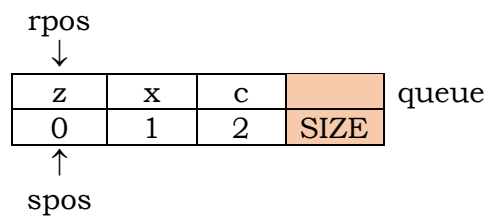
3. store('c')



Gambar 8. 14 Langkah 3 Ilustrasi Circular Queue

'c' masuk dan spos bergeser ke posisi SIZE sehingga spos diputar kembali ke posisi 0

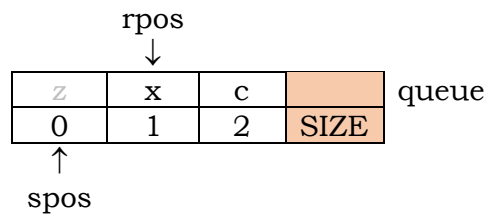
4. store('v')



Gambar 8. 15 Langkah 4 Ilustrasi Circular Queue

Operasi tidak dijalankan, karena kondisi queue penuh.

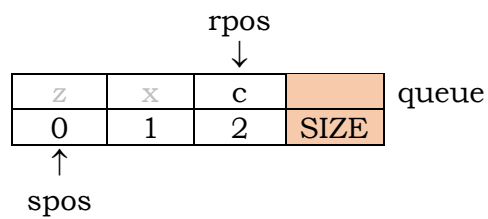
5. retrieve()



Gambar 8. 16 Langkah 5 Ilustrasi Circular Queue

'z' dikeluarkan dari antrian, rpos bergeser ke posisi 1.

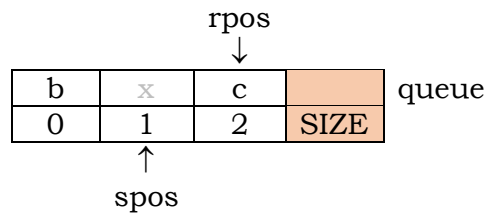
6. retrieve()



Gambar 8. 17 Langkah 6 Ilustrasi Circular Queue

'x' dikeluarkan dari antrian, rpos bergeser ke posisi 2.

7. store('b')



Gambar 8. 18 Langkah 7 Ilustrasi Circular Queue

'b' masuk, spos bergeser ke posisi 1. Bila dibaca maka isi antrian adalah: 'c', 'b'.

8.4 Penerapan Queue

Penerapan antrian dalam data sangat banyak, dicontohkan sebagai berikut:

- Pengetikan dengan keyboard
- Antrian Pembayaran Kasir, Bank, Pembuatan Resep
- Antrian Pelayanan Pemeriksaan Pasien

8.5 Studi Kasus

Membuat antrian dengan kapasitas 10 elemen.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Antrian
{
private:
    vector<string> data;
    int depan, belakang;
    int maksElemen;
public:
    // Konstruktor
    Antrian(int ukuran)
    {
        depan = 0;
        belakang = 0;
```

```

        maksElemen = ukuran;
        data.resize(ukuran); // Ukuran vector
    }

    // Memasukkan data ke antrian
    // Nilai balik tidak ada
    void insert(string x)
    {
        int posisiBelakang;

        // Geser belakang ke posisi berikutnya
        if (belakang == maksElemen)
            posisiBelakang = 1;
        else
            posisiBelakang = belakang + 1;

        // Cek belakang apa sama dengan Depan
        if (posisiBelakang == depan)
            cout << "Antrian penuh" << endl;
        else
        {
            belakang = posisiBelakang;

            // Masukkan data
            data[belakang] = x;
        }
    }

string remove(void)
{
    if (empty())
    {
        cout << "Antrian kosong" << endl;
        return "";
    }

    if (depan == maksElemen)
        depan = 1;
    else
        depan = depan + 1;
}

```

```

        return data[depan];
    }

    bool empty(void)
    {
        if (depan == belakang)
            return true;
        else
            return false;
    }
};

int main()
{
    int ukuran = 10;
    Antrian daftar(ukuran); // Buat objek

    // Masukkan 5 buah nama
    daftar.insert("Aman");
    daftar.insert("Budi");
    daftar.insert("Caca");
    daftar.insert("Didi");
    daftar.insert("Edi");

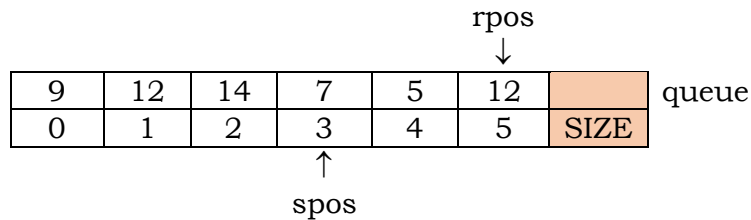
    // Kosongkan isi antrian dan tampilkan
    while (! daftar.empty())
    {
        string nama = daftar.remove();
        cout << nama << endl;
    }

    return 0;
}

```

8.6 Latihan

1. Ilustrasikan antrian untuk daftar pemanggilan operasi berikut:
store('Ana')
store('Bina')
store('Cika')
retrieve
store('Dana')
retrieve
store('Eka')
store('Fina')
retrieve
2. Saat kita mengalokasikan antrian menggunakan array dengan jumlah n elemen maka data yang dapat kita isikan hanya $n-1$ saja, mengapa?
3. Terdapat ilustrasi antrian sebagai berikut :



- Sebutkan jumlah dari data antrian diatas dan bagaimana urutan antriannya?
Apabila ditambahkan operasi remove diakhir proses maka gambarlah ilustrasi hasil akhirnya.
4. Apakah memungkinkan dilakukan penambahan elemen didepan? Mengapa?

9

Linked List

Sub CPMK

Mahasiswa mampu mengevaluasi, merancang dan mengimplementasikan struktur data nonlinear seperti matriks, dan multiple linked list kedalam program

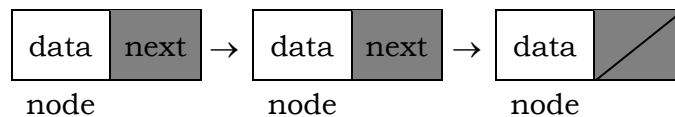
Dalam bab ini akan dibahas tentang Linked List atau sering disebut sebagai senarai berantai. Pembahasan diawali dengan pengertian kemudian akan dibahas dua operasi utama yaitu penambahan dan penghapusan. Masing-masing pembahasan operasi disertai dengan ilustrasi visual untuk menggambarkan tiap prosesnya. Pada akhir materi akan diberikan Studi Kasus beserta Latihan.

9.1 Pengertian

Linked list sering disebut juga senarai berantai. Linked list dapat dianalogikan sebagai rantai besi yang terdiri dari beberapa besi bulatan yang saling terhubung. Pengertian linked list adalah struktur data yang terdiri dari sekelompok node (simpul) yang membentuk rangkaian secara runtut, sekuensial, saling sambung menyambung dan dinamis. Linked list saling terhubung dengan elemen lain melalui bantuan variabel pointer. Penggunaan pointer sangat mendukung dalam pembentukan struktur data dinamis. Salah satu struktur data dinamis adalah linked list. Masing-masing data dalam linked list disebut dengan node (simpul) yang menempati alokasi memori secara dinamis dan biasanya berupa struct yang terdiri dari beberapa field.

Linked list dalam bentuk yang sederhana setiap node terdiri dari 2 bagian, yaitu data dan *next* (penyambung ke node berikutnya). Analoginya mirip dengan rangkaian gerbong kereta api. Masing-masing gerbong itulah yang disebut struct/tipe data bentukan. Agar gerbong kereta itu saling bertautan maka dibutuhkan sebuah kait yang disebut sebagai pointer.

Berikut ini Gambar 9.1 adalah ilustrasi dari linked list sederhana.



Gambar 9. 1 Ilustrasi linked list

Gambar 9.1 dari sebuah node:

- Bagian **data**, disebut medan informasi, berisi informasi yang akan disimpan dan diolah.
- Bagian **next**, disebut medan penyambung (link field), merupakan pointer yang berisi alamat simpul berikutnya.

Dalam ilustrasi linked list sederhana setiap node memiliki segmen untuk data dan satu segmen sebagai penghubung ke node di belakangnya. Pointer awal menunjuk ke simpul pertama dari senarai tersebut. Medan penyambung (pointer) dari suatu simpul yang tidak menunjuk simpul lain disebut pointer kosong, yang nilainya dinyatakan sebagai null (null adalah kata baku yang berarti bahwa pointer 0 atau bilangan negatif). Jadi kita bisa melihat bahwa dengan hanya sebuah pointer awal saja maka kita bisa membaca semua informasi yang tersimpan dalam senarai.

Penggunaan linked list memiliki beberapa keuntungan dibandingkan dengan array konvensional, di antaranya:

1. Bersifat dinamis, pengalokasian memori hanya sesuai dengan yang dibutuhkan dan didealokasikan saat tidak diperlukan lagi (penggunaan memori yang efisien).

2. Memori tidak dipesan terlebih dahulu
3. Operasi penambahan dan penghapusan dapat diimplementasikan dengan mudah.
4. Penyisipan dan penghapusan dapat dilakukan di sembarang tempat sesuai keperluan.
5. Dapat diimplementasikan dengan mudah untuk struktur data linear seperti stack dan queue.

Selain kelebihan linked list juga memiliki beberapa kekurangan, diantaranya:

1. Penggunaan memori tambahan untuk pointer di setiap node.
2. Dalam linked list sederhana pembacaan hanya dapat dilakukan secara sekuensial dari depan ke belakang (forward list).
3. Pengaksesan node tidak bisa dilakukan secara acak.
4. Proses pencarian lebih sulit dan *time consuming*.

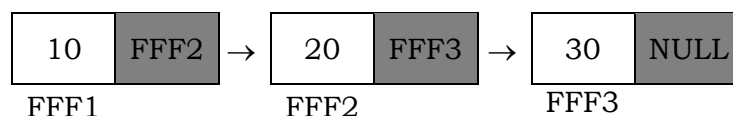
Beberapa contoh penerapan linked list antara lain:

1. Dapat untuk merepresentasikan dan memanipulasi polinomial.
2. Dapat merepresentasikan bilangan yang sangat besar dan operasi-operasinya.
3. Dapat untuk mengimplementasikan stack, queue, tree and graf.
4. Mengimplementasikan tabel simbol pada compiler construction

Linked list dapat disajikan dengan 2 bagian besar yaitu *Singly List* dan *Doubly List*. Baik *Singly List* dan *Doubly List* yang dapat disajikan secara melingkar (*circular*). Jenis-jenis Linked List antara lain:

1. ***Singly Linked List***.

- Merupakan *linked list* paling sederhana.
- Semua node dirangkai bersama dengan cara sekuensial, disebut juga *linked list linier*.
- Setiap simpul dibagi menjadi 2 bagian yaitu satu bagian isi dan satu bagian pointer.
Bagian isi berisi data yang disimpan oleh simpul. Bagian pointer, yaitu next berisi alamat yang menunjuk ke setiap node berikutnya.
- NULL memiliki nilai khusus yang artinya tidak menunjuk ke mana-mana. Biasanya *linked list* pada titik akhirnya akan menunjuk ke NULL).



Gambar 9. 2 Ilustrasi singly linked list non circular

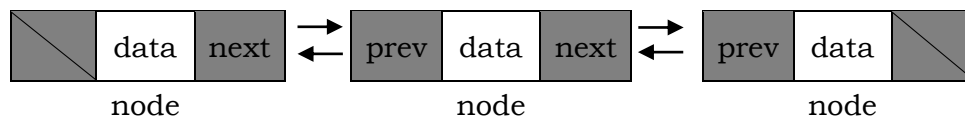
Pembuatan *Singly Linked List* dapat menggunakan 2 metode:

- a) LIFO (Last In First Out), aplikasinya: Stack (Tumpukan)
- b) FIFO (First In First Out), aplikasinya: Queue (Antrean)

Salah satu kelemahan *singly linked list* adalah pointer (penunjuk) hanya dapat bergerak satu arah saja, maju/mundur, atau kanan/kiri sehingga pencarian data pada *singly linked list* hanya dapat bergerak dalam satu arah saja.

2. **Doubly Linked List.**

- List berpointer ganda atau metode *doubly linked list* hadir untuk mengatasi kelemahan *single linked list* yang hanya dapat bergerak satu arah saja, maju/mundur, atau kanan/kiri saja.
- Terdiri dari 3 bagian, yaitu untuk menyimpan nilai dan dua reference yang menunjuk ke node. Node dirangkai dengan 2 link yang dapat mempermudah pengaksesan *successor node* (*next node*) dan *predecessor node* (*previous node*) dari sembarang node.
- Setiap node memiliki 2 variabel pointer yaitu yang menunjuk node kirinya (*previous*) dan yang menunjuk ke node kanannya (*next*). Ini berguna untuk melakukan penelusuran dengan arah *forward* dan *backward*.



Gambar 9. 3 Ilustrasi *doubly linked list*

Keberadaan 2 pointer penunjuk (*next* dan *prev*) menjadikan *Doubly Linked List* menjadi lebih fleksibel dibandingkan *Singly Linked List*, namun membutuhkan memori tambahan dengan adanya pointer tambahan tersebut. *Doubly Linked List* mempunyai reference **front** untuk menandai awal node dan reference **back** untuk menandai akhir list

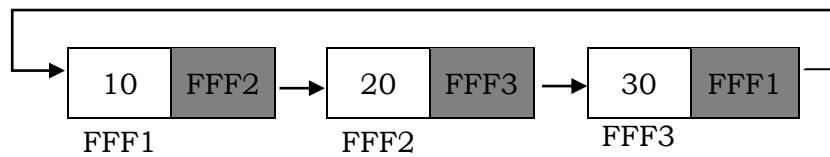
Pembacaan pada Doubly Linked List

Double Linked List dapat dibaca melalui dua arah, yaitu:

- Pembacaan maju (forward scan) yaitu membaca double linked list dimulai dari reference front dan berakhir pada reference back.
- Pembacaan mundur (backward scan) yaitu membaca double linked list dimulai dari reference back dan berakhir pada reference front.

3. **Circular Linked List.**

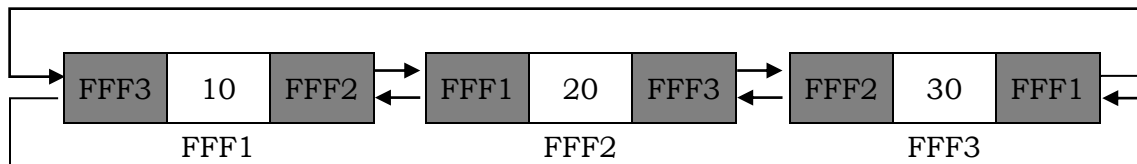
- Tidak memiliki awal maupun akhir.
- *Single linked list* dapat diubah menjadi circular linked list dengan menyimpan alamat dari node pertama pada variabel pointer node yang terakhir.



Gambar 9. 4 Ilustrasi singly linked list circular

4. **Circular Doubly Linked List.**

- Adalah *circular linked list* yang memiliki baik *successor pointer* maupun *predecessor pointer*.
- Merupakan *doubly linked list* yang simpul terakhirnya menunjuk ke simpul awalnya serta simpul awalnya menunjuk ke simpul akhir sehingga membentuk suatu lingkaran.



Gambar 9. 5 Ilustrasi doubly linked list circular

Materi selanjutnya akan kita bahas lebih dalam mengenai *Singly Linked List Non-Circular* atau *Linked List Linier*. Setiap node memiliki 2 field/variabel:

- Field "*data*" untuk menyimpan data
- Field "*next*" yang berupa pointer yang menunjuk pada next node.

Node pertama disebut sebagai start node atau head node. Pointer yang disebut "*start*" atau "*head*" menunjuk ke node pertama pada *linked list*.

Sebuah node pada *linked list* yang paling sederhana dapat didefinisikan menggunakan struct beranggotakan satu elemen dan satu pointer yang menunjuk ke node berikutnya. Berikut ini adalah contoh definisi struct Node untuk *linked list* dengan elemen bertipe int.

```
Node
{
    int elm;
    Node* next;
};
```

Struct Node berisi anggota elm bertipe int sebagai representasi data yang tersimpan dalam linked list dan pointer next bertipe Node sebagai penghubung dengan node yang lain. Linked list dapat dibentuk dari beberapa pointer bertipe Node sebagai handle. Ada 2 handle yang dapat dibentuk, yaitu sebagai pemegang awal (head)

rangkaian dan pemegang akhir (tail) rangkaian. Berikut ini adalah deklarasi untuk 2 handle tersebut:

```
Node* hd; // head, handle awal
Node* tl; // tail, handle akhir
size_t sz; // counter jumlah node
```

Rangkaian *linked list* dapat dialokasikan secara dinamis dalam heap menggunakan 2 handle yang sudah dibentuk. Selain itu terdapat sebuah variabel `sz` bertipe `size_t` yang digunakan sebagai pencatat jumlah node dalam linked list. `sz` akan selalu berubah nilainya setiap kali terjadi operasi penambahan ataupun pengurangan node.

Sebelum digunakan linked list harus diinisialisasi terlebih dahulu. Inisialisasi sebaiknya dilakukan saat pendeklarasian, sehingga kode sebelumnya dapat dimodifikasi seperti berikut ini.

```
Node* hd{nullptr}; // head, handle awal
Node* tl{nullptr}; // tail, handle akhir
size_t sz{0}; // counter jumlah node
// inisialisasi dengan 0
```

Nilai `sz` diinisialisasi dengan 0 yang menyatakan bahwa dalam *linked list* belum terdapat rangkaian node sama sekali. Semua handle menunjuk ke nilai `nullptr` yang menyatakan bahwa ketiganya tidak menunjuk ke manapun.

9.2 Operasi

Semua operasi dalam linked list disusun dalam fungsi – fungsi yang bersesuaian. Operasi meliputi penambahan depan, penambahan belakang, penghapusan depan, penghapusan belakang, dan pembersihan.

Operasi-Operasi yang ada pada Linked List antara lain:

- **Insert**
Istilah Insert berarti menambahkan sebuah simpul baru ke dalam suatu linked list.
- **IsEmpty**
Fungsi ini menentukan apakah linked list kosong atau tidak.
- **Find First**
Fungsi ini mencari elemen pertama dari linked list
- **Find Next**
Fungsi ini mencari elemen sesudah elemen yang ditunjuk now

- **Retrieve**
Fungsi ini mengambil elemen yang ditunjuk oleh now. Elemen tersebut lalu dikembalikan oleh fungsi.
- **Update**
Fungsi ini mengubah elemen yang ditunjuk oleh now dengan isi dari sesuatu
- **Delete Now**
Fungsi ini menghapus elemen yang ditunjuk oleh now. Jika yang dihapus adalah elemen pertama dari linked list (head), head akan berpindah ke elemen berikut.
- **Delete Head**
Fungsi ini menghapus elemen yang ditunjuk head. Head berpindah ke elemen sesudahnya.
- **Clear**
Fungsi ini menghapus linked list yang sudah ada. Fungsi ini wajib dilakukan bila anda ingin mengakhiri program yang menggunakan linked list. Jika anda melakukannya, data-data yang dialokasikan ke memori pada program sebelumnya akan tetap tertinggal di dalam memori.

9.2.1 Penambahan

Operasi yang digunakan untuk menyisipkan simpul di posisi tertentu. Terdapat tiga operasi penambahan yang dapat dilakukan yaitu penyisipan simpul di posisi depan, penyisipan simpul di belakang dan penyisipan simpul di tengah. Saat melakukan penambahan terdapat dua kemungkinan yaitu linked list masih kosong atau sudah berisi. Ketika masih kosong ($sz == 0$) maka penambahan depan atau belakang akan sama saja sehingga perlu dibuat sebuah fungsi tambahan untuk menangani hal ini, misal bernama `insert_new()`. Berikut ini adalah definisi fungsi `insert_new()`.

```
void insert_new(int e)
{
    Node* tmp = new Node;
    if(tmp)
    {
        tmp->elm = e;
        tmp->next = nullptr;
        hd = tmp;
        tl = tmp;
        ++sz;
    }
}
```

`insert_new()` dipanggil oleh fungsi penambahan depan dan belakang hanya ketika linked list masih kosong.

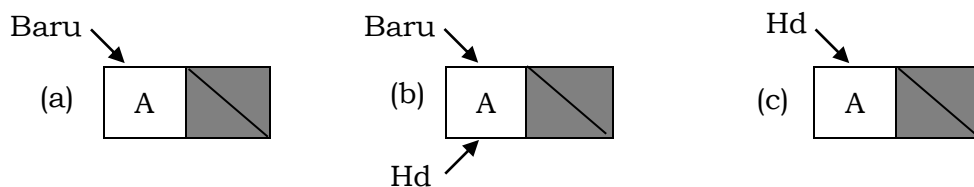
9.2.1.1 Tambah Depan: insert_front()

Penambahan node dari depan rangkaian linked list ini diimplementasikan dalam fungsi `insert_front()`. Operasi ini dilakukan dengan cara mengalokasikan node baru yang kemudian langsung dipegang oleh head.

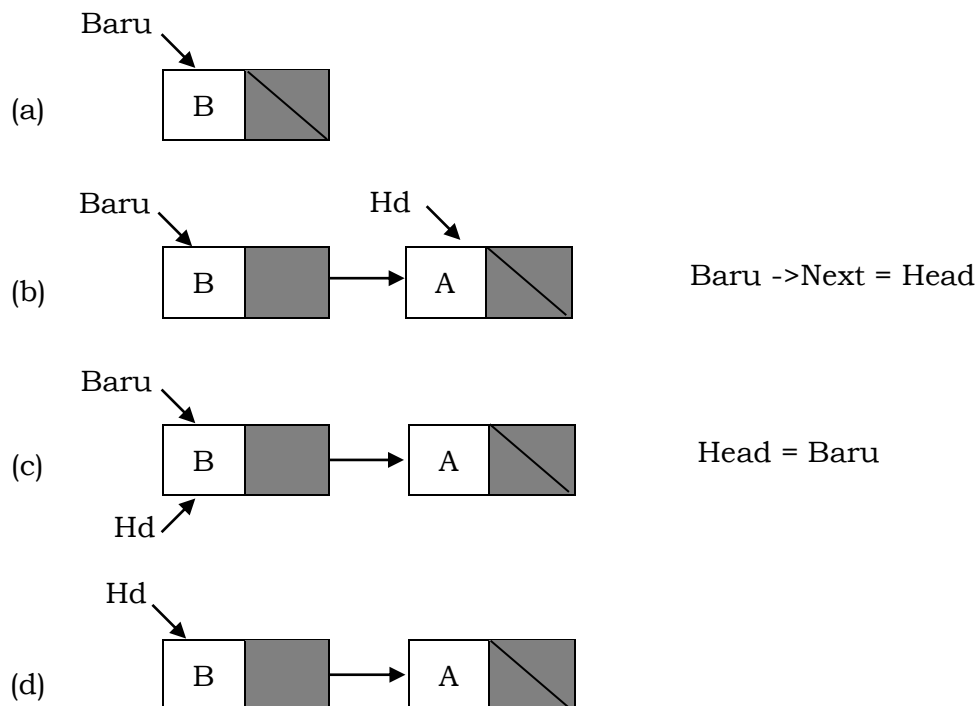
Langkah-langkah penyisipan simpul dapat dilakukan dengan cara:

- Membuat simpul baru yang akan disisipkan
- Jika linked list belum ada maka simpul baru menjadi linked list (Head = Baru).
- Jika linked list sudah ada maka penyisipan dilakukan dengan cara Pointer next simpul baru menunjuk Head (Baru->Next = Head).
Pointer Head dipindahkan ke Baru (Head = Baru).

Skema penyisipan simpul depan dapat dilihat pada Gambar 9.6 (Linked list belum ada) dan Gambar 9.7.



Gambar 9. 6 Penyisipan depan dengan linked list belum ada



Gambar 9. 7 Penyisipan simpul depan

Berikut ini adalah kode untuk fungsi `insert_front()`.

```
void insert_front(int e)
{
    if(sz == 0)
        insert_new(e);
    else
    {
        Node* tmp = new Node;
        if(tmp)
        {
            tmp->elm = e;
            tmp->next = hd;
            hd = tmp;
            ++sz;
        }
    }
}
```

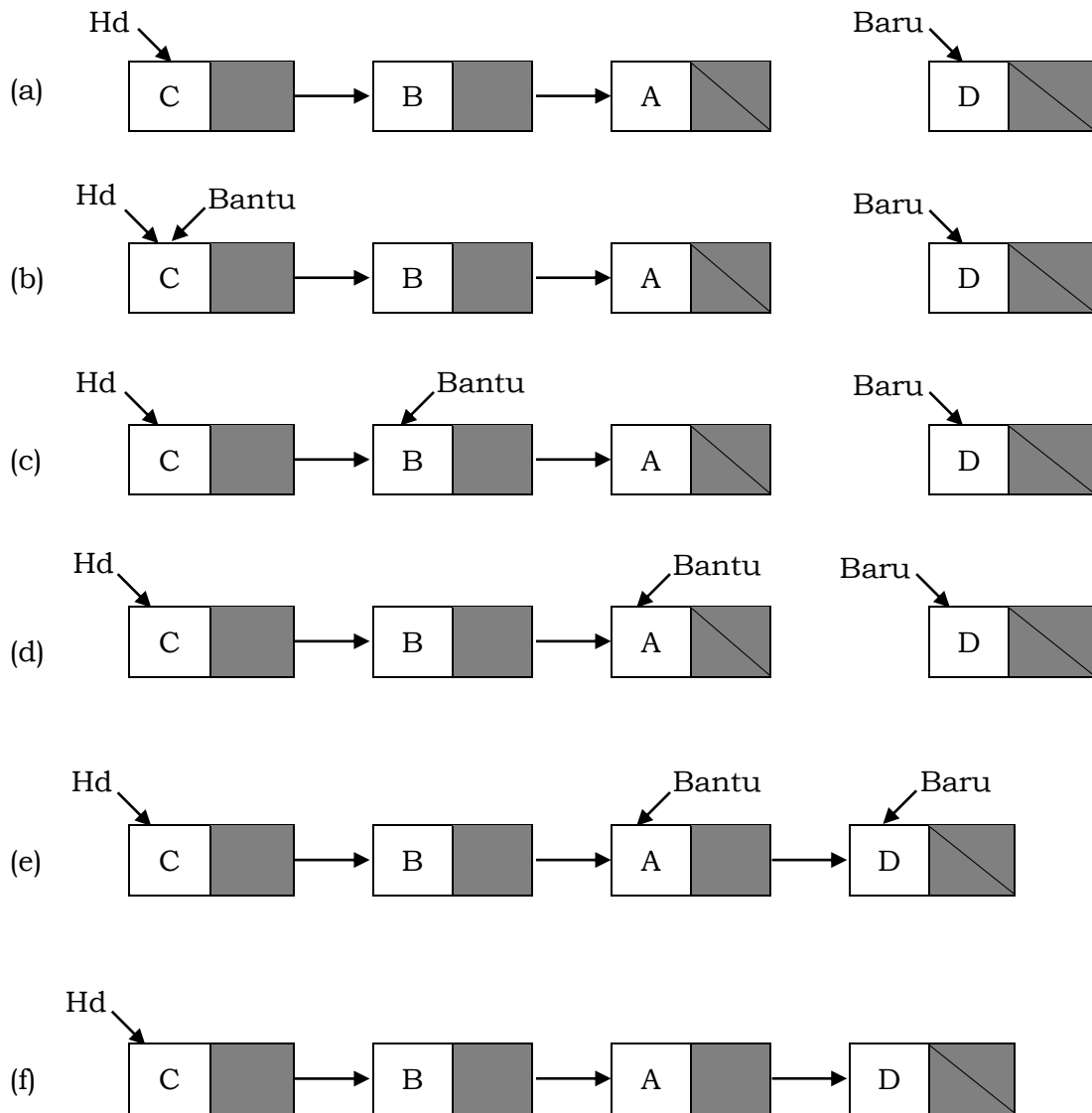
9.2.1.2 Tambah Belakang: `insert_back()`

Penambahan node dari belakang rangkaian *linked list* diimplementasikan dalam fungsi `insert_back()`. Operasi ini dilakukan dengan cara mengalokasikan node baru untuk ditempatkan dibelakang rangkaian yang langsung dipegang oleh tail.

Langkah-langkah penyisipan simpul belakang dapat dilakukan dengan:

- a) Membuat simpul baru yang akan disisipkan
- b) Jika linked list belum ada maka simpul baru menjadi linked list (Head = Baru).
- c) Jika linked list sudah ada maka penyisipan dilakukan dengan cara
 - Buat suatu pointer yang dapat digerakkan, misalnya pointer bantu yang menunjuk simpul pertama dari linked list (Bantu = Head). Hal ini dilakukan karena pointer Head tidak boleh digerakkan dari simpul depan. Karena jika pointer Head digerakkan maka informasi dari simpul yang ditinggalkan pointer Head tidak dapat diakses lagi.
 - Gerakkan pointer bantu hingga ke simpul paling belakang (tail) dari linked list (`while(Bantu->Next!=NULL) Bantu=Bantu->Next;`).
 - Sambungkan linked list dengan simpul baru (`Bantu->Next=Baru`).

Skema penyisipan simpul belakang saat Linked list belum ada dapat dilihat pada Gambar 9.6. Sedangkan penyisipan simpul belakang saat linked list sudah ada dapat dilihat pada Gambar 9.8.



Gambar 9. 8 Penyisipan simpul belakang

Berikut ini adalah kode untuk fungsi insert_back().

```
void insert_back(int e)
{
    if(sz == 0)
        insert_new(e);
    else
    {
        Node* tmp = new Node;
        if(tmp)
        {
            tmp->elm = e;
            tmp->next = nullptr;
            tl->next = tmp;
        }
    }
}
```

```

        t1 = tmp;
        ++sz;
    }
}

```

9.2.1.3 Tambah Tengah: `insert_mid()`

Penambahan node dari tengah rangkaian *linked list* diimplementasikan dalam fungsi `insert_mid()`. Operasi ini dilakukan dengan cara mengalokasikan node baru untuk ditempatkan ditengah rangkaian tertentu atau setelah simpul tertentu. Penyisipan simpul tengah hanya dapat dilakukan jika *linked list* tidak kosong.

a. Menambah simpul **sebelum** simpul tertentu

Sebelum melakukan operasi penambahan node sebelum simpul tertentu maka kita harus mengetahui simpul manakah yang dimaksud, yaitu simpul yang berisi informasi dimana simpul akan disisipkan.

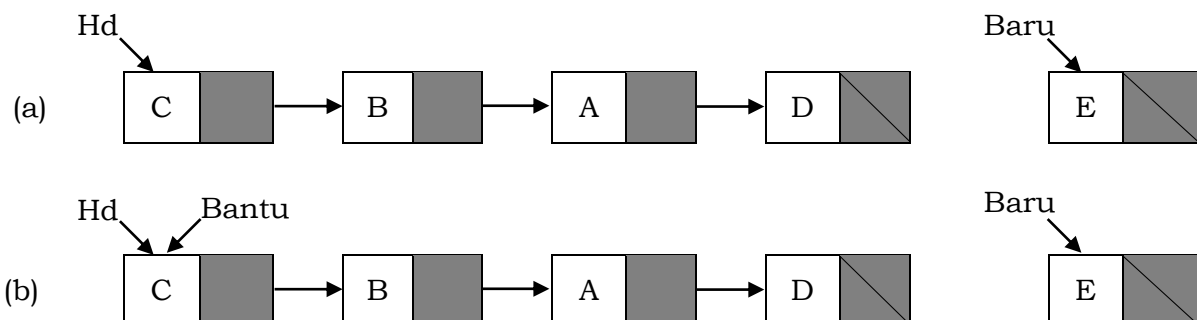
Misalnya:

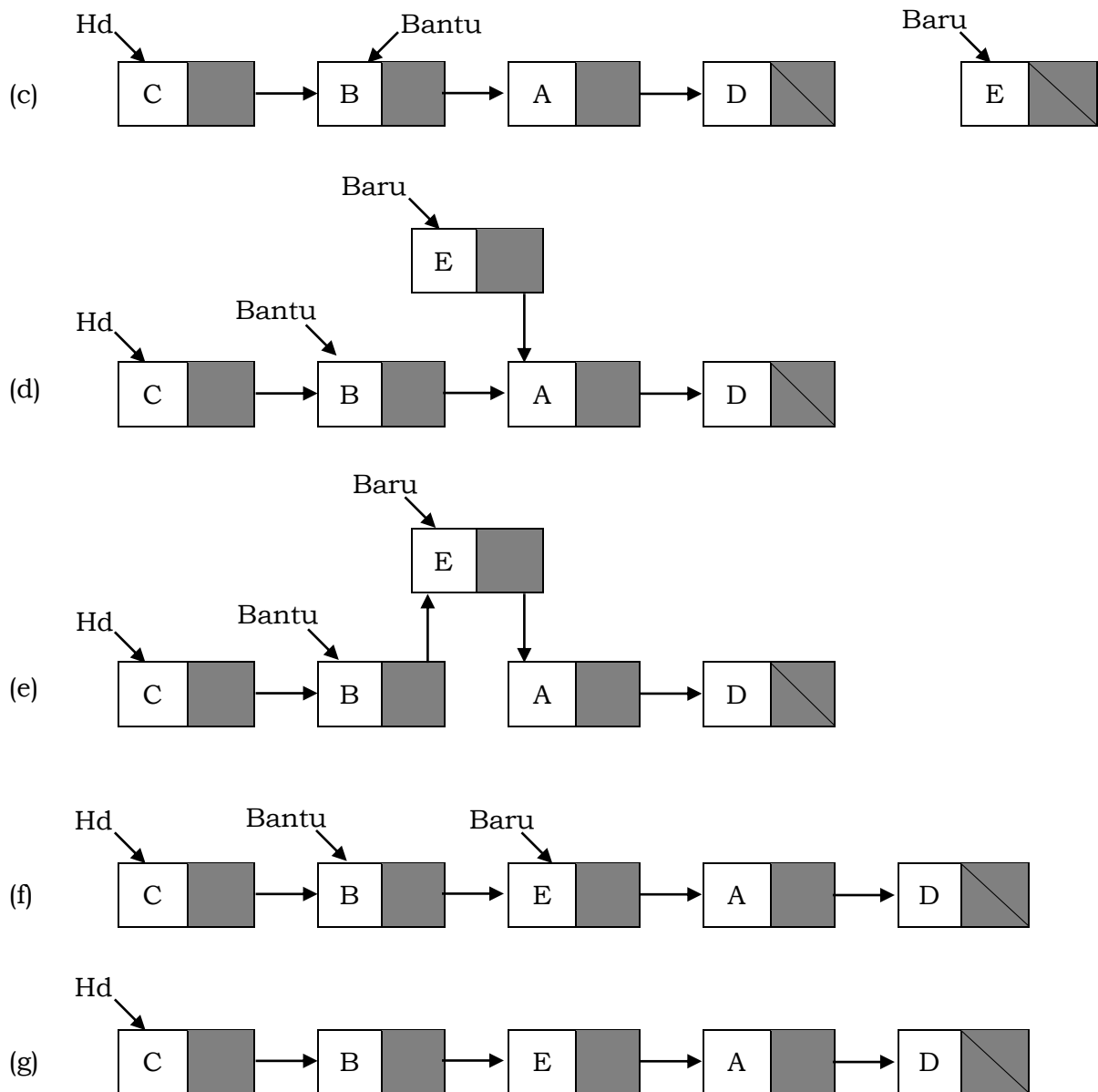
Terdapat 3 simpul *linked list* yang masing-masing berisi informasi C, B, A dan D. Kemudian kita akan menambahkan simpul baru yang berisi informasi E. simpul baru tersebut akan disisipkan sebelum simpul yang berisi informasi A.

Langkah-langkah yang dapat dilakukan adalah:

- Membuat simpul baru yang akan disisipkan
- Buat suatu pointer yang dapat digerakkan, misalnya pointer bantu yang menunjuk simpul pertama dari *linked list* (Bantu = Head). Hal ini dilakukan karena pointer Head tidak boleh digerakkan dari simpul depan. Karena jika pointer Head digerakkan maka informasi dari simpul yang ditinggalkan pointer Head tidak dapat diakses lagi.
- Gerakkan pointer bantu hingga ke simpul sebelum simpul tujuan, yaitu sebelum simpul yang berisi informasi A dari *linked list* (`while(Bantu->Next->Isi != 'A') Bantu=Bantu->Next;`).
- Sambungkan simpul baru dengan simpul berikutnya (`Baru->Next=Bantu->Next`).
- Sambungkan simpul Bantu dengan simpul Baru (`Bantu->Next=Baru`).

Skema penyisipan simpul sebelum simpul tertentu dapat dilihat pada Gambar 9.9.





Gambar 9. 9 Penyisipan simpul sebelum simpul tertentu

b. Menambah simpul **setelah** simpul tertentu

Operasi penyisipan simpul setelah simpul tertentu dengan simpul sebelum simpul tertentu memiliki proses yang sangat mirip. Perbedaan proses hanya berada pada tahapan gerakan pointer bantu hingga simpul berisi informasi tertentu, misalnya menggunakan contoh kasus sebelumnya maka akan dicari simpul yang berisi informasi A. (`while(Bantu -> Isi != 'A') Bantu=Bantu->Next;`).

9.2.2 Penghapusan

Operasi penghapusan ini dimaksudkan untuk menghapus suatu simpul dari linked list. Sama halnya dengan penambahan, terdapat dua operasi penghapusan node yaitu di depan, di belakang dan di tengah. Operasi penghapusan tidak hanya memutus node di ujung depan atau belakang, tetapi harus memastikan bahwa node yang terputus tersebut benar – benar didealokasikan.

Hal yang perlu diperhatikan saat operasi penghapusan simpul ini dilakukan adalah linked list tidak boleh kosong dan linked list tidak boleh terputus.

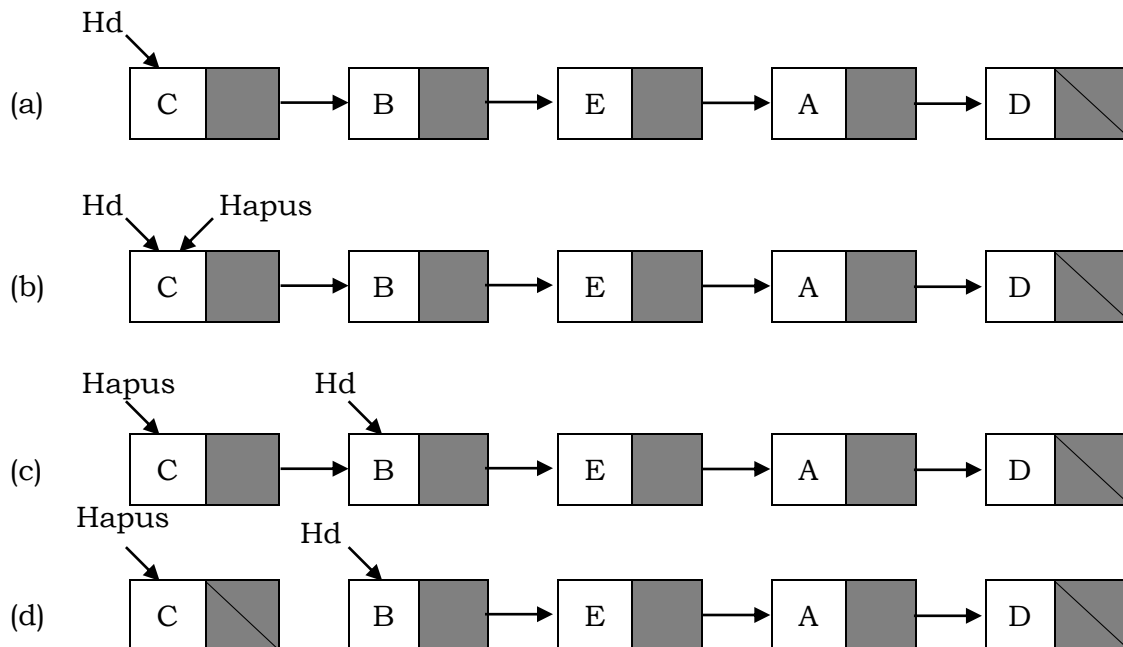
9.2.2.1 Hapus Depan: `remove_front()`

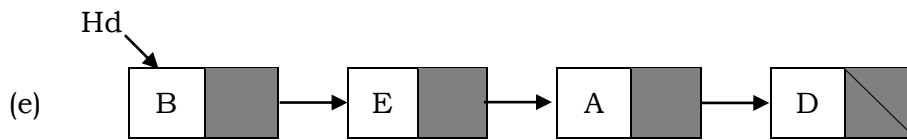
Penghapusan depan dapat diimplementasikan dalam fungsi `remove_front()`. Operasi ini dilakukan dengan cara memutus node pertama yang langsung dipegang oleh head. Kemudian dilakukan dealokasi terhadap node yang terputus tersebut.

Langkah-langkah penghapusan simpul depan dari linked list dapat dilakukan dengan cara berikut:

- Buat sebuah pointer misalnya pointer hapus, pointer ini akan menunjuk simpul yang akan dihapus dan Head untuk menunjuk linked list.
- Simpul pertama ditunjuk oleh pointer hapus (Hapus = Head).
- Pointer Head digerakkan satu simpul berikutnya (Head = Head->Next).
- Putuskan simpul pertama dari Head (Hapus->Next=NULL).

Skema penghapusan simpul dapat dilihat pada Gambar 9.10.





Gambar 9. 10 Penghapusan simpul depan

Berikut ini adalah kode untuk fungsi `remove_front()`.

```
void remove_front()
{
    if(!empty())
    {
        Node* it = hd;
        hd = hd->next;
        delete it;
        --sz;
    }
}
```

9.2.2.2 Hapus Belakang: `remove_back()`

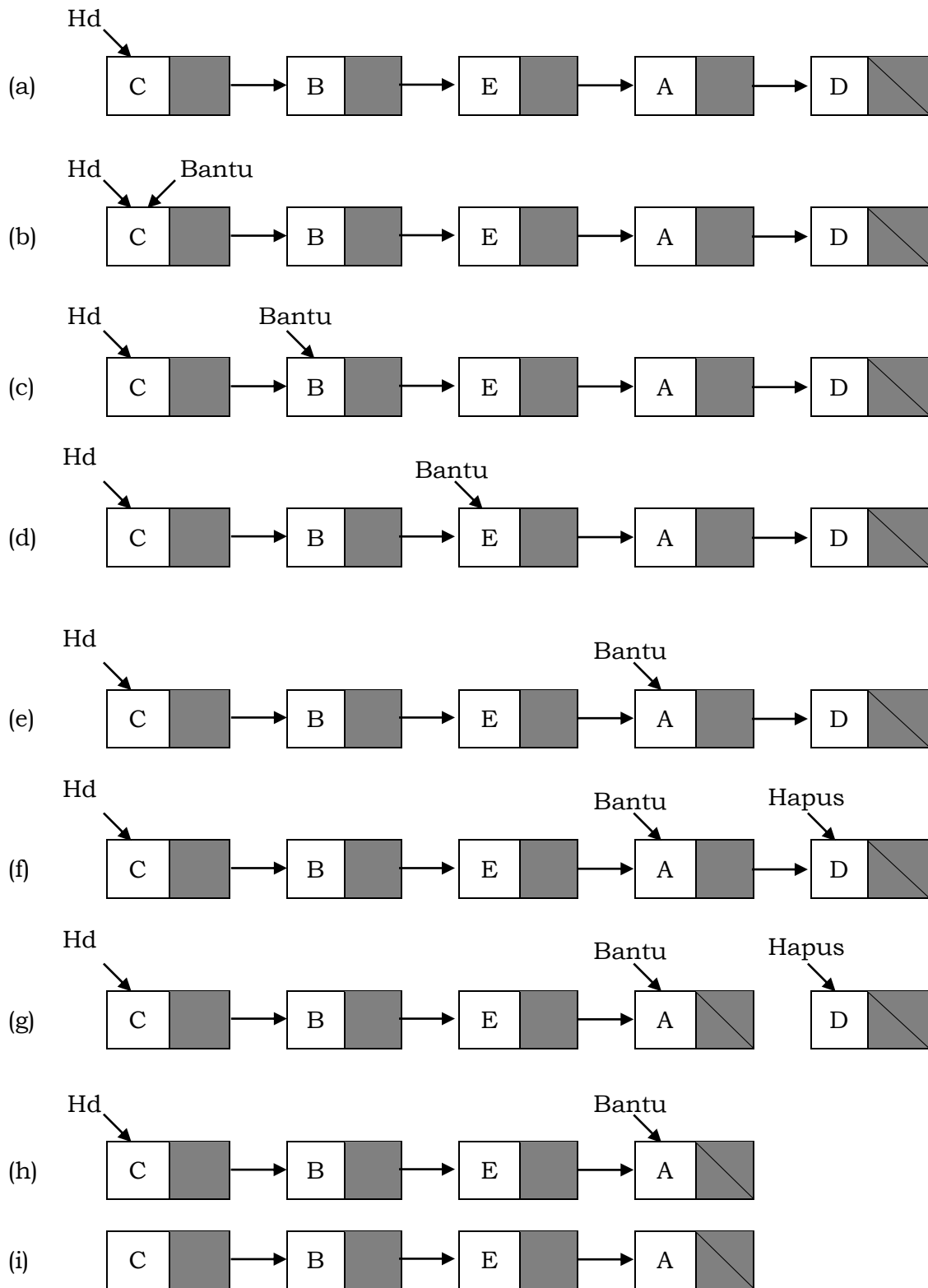
Penghapusan belakang dapat diimplementasikan dalam fungsi `remove_back()`. Operasi ini dilakukan dengan cara memutus node terakhir yang langsung dipegang oleh tail. Kemudian dilakukan dealokasi terhadap node yang terputus tersebut. Perlu diperhatikan juga bahwa operasi ini bisa dijalankan jika linked list tidak kosong.

Berbeda dengan `remove_front()` fungsi ini harus melakukan penelusuran dari head terlebih dahulu untuk mencapai tail, sehingga dibutuhkan variabel bantu. Dengan menggunakan variabel bantu maka memungkinkan tidak adanya simpul yang hilang atau terputus. Selain itu juga dibutuhkan pointer Hapus untuk menunjuk simpul yang akan dihapus.

Langkah-langkah penghapusan simpul belakang dapat dilakukan sebagai berikut:

- Letakkan pointer bantu pada simpul pertama (Bantu = Head).
- Gerakkan pointer bantu hingga pada satu simpul sebelum simpul terakhir (`while(Bantu->Next->Next!=NULL)Bantu=Bantu->Next;`).
- Simpul terakhir ditunjuk oleh pointer hapus (`Hapus = Bantu->Next`).
- Putuskan simpul terakhir dari Head (`Bantu->Next=NULL`).

Skema penghapusan simpul belakang dapat dilihat pada Gambar .11.



Gambar 9. 11 Penghapusan simpul belakang

Berikut ini adalah kode untuk fungsi `remove_back()`.

```
void remove_back()
{
    if(!empty())
    {
        Node* it = hd;
        while(it->next != tl)
            it = it->next;
        delete tl;
        tl = it;
        --sz;
    }
}
```

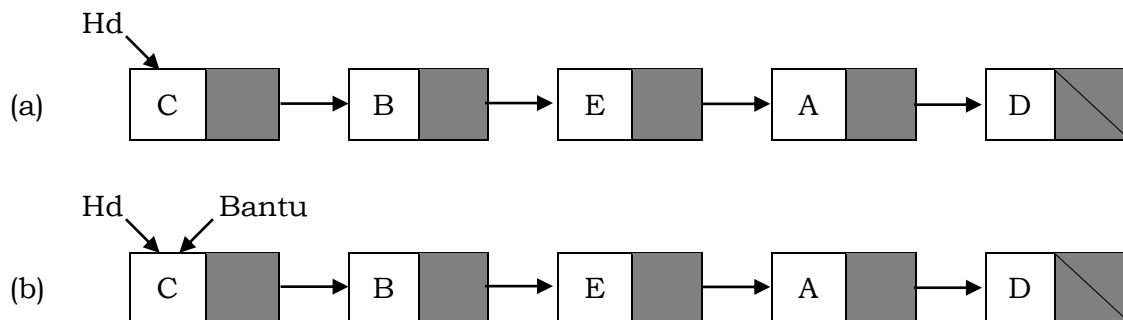
9.2.2.3 Hapus Tengah: `remove_mid()`

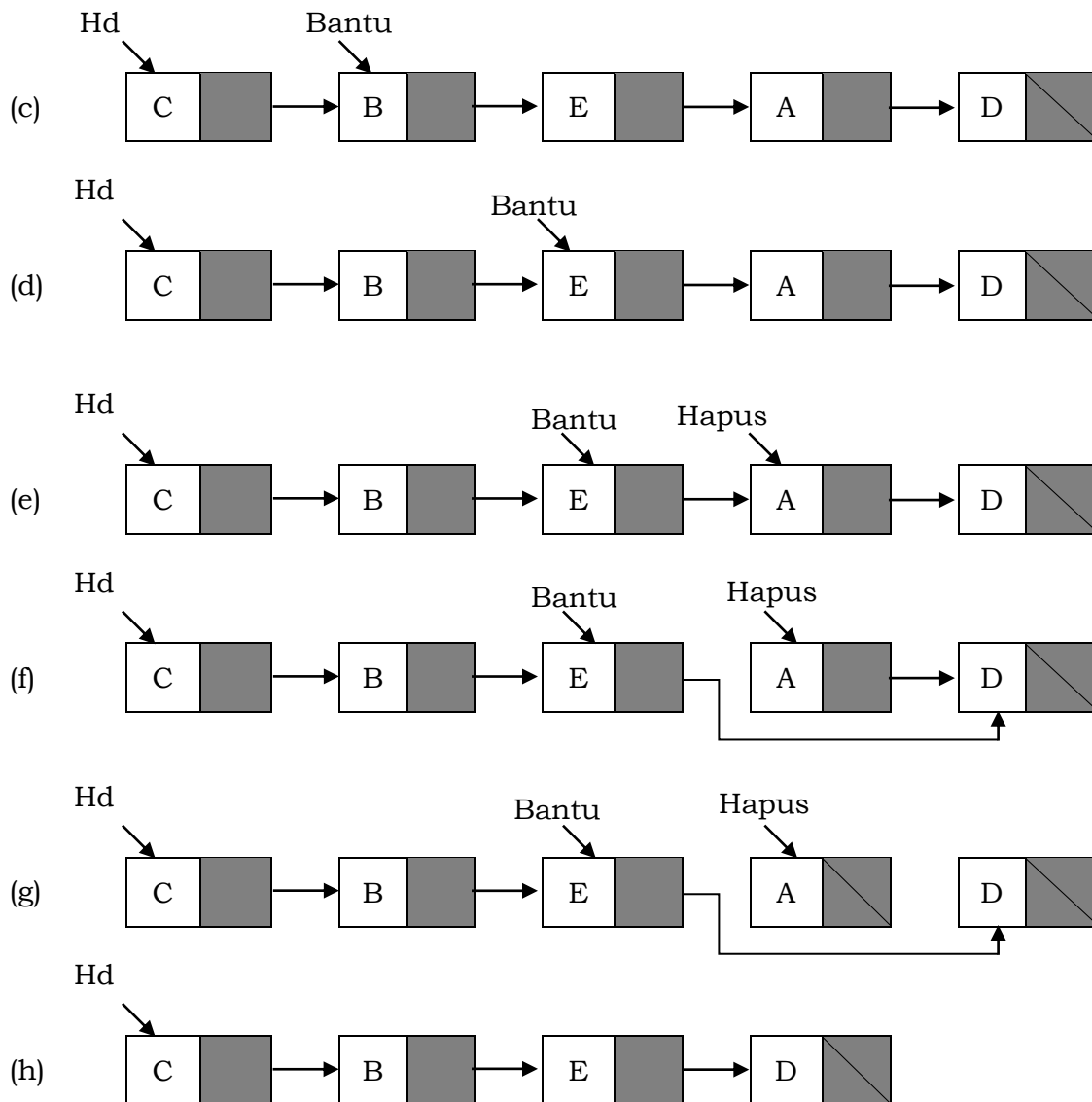
Menghapus simpul yang berada di posisi tengah dari linked list. Hal yang sama yang perlu diperhatikan adalah linked list tidak boleh kosong. Pointer Bantu akan digerakkan dari head hingga berada pada posisi sebelum simpul yang akan dihapus. Pointer Hapus juga digunakan untuk menunjuk pada simpul yang akan dihapus.

Langkah-langkah penghapusan simpul tengah jika menggunakan contoh kasus seperti sebelumnya yang menghapus simpul berisi informasi A maka dapat dilakukan sebagai berikut:

- Letakkan pointer bantu pada simpul pertama (Bantu = Head).
- Gerakkan pointer bantu hingga pada satu simpul sebelum simpul yang akan dihapus (`while(Bantu->Next->Isi!="A")Bantu=Bantu->Next;`).
- Letakkan pointer hapus pada simpul yang akan dihapus (Hapus = Bantu->Next).
- Pointer bantu menunjuk simpul setelah simpul yang ditunjuk oleh hapus (`Bantu->Next=Hapus->Next` atau `Bantu->Next=Bantu->Next->Next`).
- Putuskan simpul yang ditunjuk hapus dari Linked List (`Hapus->Next=NULL`).

Skema penghapusan simpul tengah dapat dilihat pada Gambar 9.11





Gambar 9. 12 Penghapusan simpul tertentu di tengah

9.2.3 Pembersihan: clear()

Pembersihan linked list dilakukan dengan cara menghapus satu persatu node yang masih terangkai. Pembersihan ini diimplementasikan dalam fungsi clear() yang berisi pemanggilan fungsi remove_front(). Pemanggilan ini dilaksanakan dalam perulangan dalam kondisi selama linked list masih belum kosong.

Berikut ini adalah kode untuk fungsi clear().

```
void clear()
{
    do{
        remove_front();
```

```

        } while(!empty());
    }

```

9.2.4 Pembacaan

Pembacaan linked list hanya dapat dilakukan secara satu arah dari depan ke belakang dengan cara mengunjungi node satu persatu (traverse). Pembacaan dilakukan oleh “iterator” yang bertipe Node.

Berikut ini adalah contoh kode untuk melakukan pembacaan keseluruhan node dalam linked list dan menampilkan ke layar;

```

Node* it = hd;
while(it)
{
    std::cout << it->elm << '\n';
    it = it->next;
}

```

9.2.5 Simulasi

Supaya lebih memahami proses penambahan pada linked list dapat memperhatikan simulasi pada gambar 9.2. Apabila secara berturut-turut dimasukkan data pada linked list sebagai berikut: Ana, Caca dan Baba maka ilustrasi penyimpanan pada linked list seperti gambar berikut ini:.

Tahap 1:

List masih kosong (head = NULL)

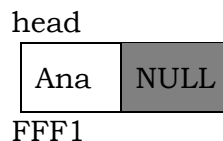


Gambar 9. 13 Langkah 1 Penambahan data linked list

Tahap 2:

Masuk data baru, ('Ana')

Jika list kosong, maka node baru sebagai start atau head

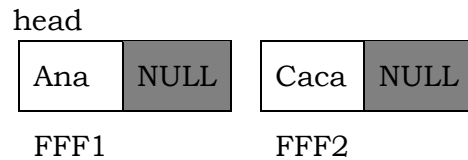


Gambar 9. 14 Langkah 2 Penambahan data linked list

Tahap 3:

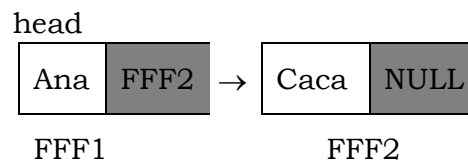
Masuk lagi data baru, **penambahan di belakang**, ('Caca')

- (1) Buat node baru



Gambar 9. 15 Langkah 3.1 Penambahan data linked list

- (2) Temp menunjuk pada start.
- (3) Temp dijalankan sampai menunjuk pada node terakhir
- (4) Tambahkan node baru pada akhir dari linked list dengan mengubah next dari node terakhir pada linked list.

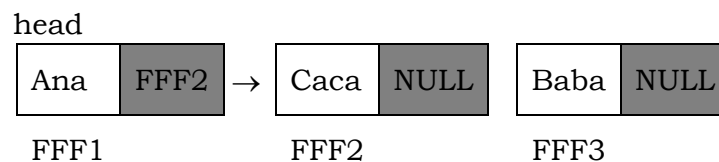


Gambar 9. 16 Langkah 3.2 Penambahan data linked list

Tahap 4:

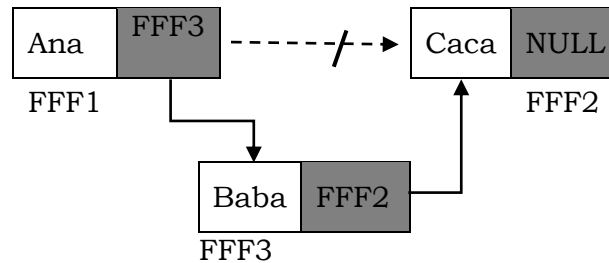
Masuk lagi data baru, **penyisipan di tengah**, pada node ke-2, ('Baba')

- (1) Buat node baru



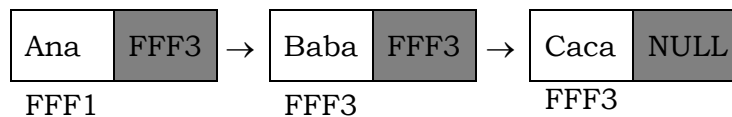
Gambar 9. 17 Langkah 4.1 Penambahan data linked list

- (2) Buat 2 variabel bantu yaitu prev dan temp.
- (3) Prev dijalankan dari head sampai menunjuk pada node yang seharusnya untuk node baru.
- (4) Jika sudah tahu tempat node seharusnya, maka node sebelumnya ditunjuk oleh prev, node sesudahnya oleh temp.
- (5) Node baru disisipkan antara prev dan temp dengan:
 - a) mengubah next dari prev agar menunjuk node baru
 - b) mengeset next dari node baru ke temp



Gambar 9. 18 Langkah 4.2 Penambahan data linked list

Hasil akhir:



Gambar 9. 19 Hasil penambahan data linked list

Dari Gambar 9.19 diatas dapat dilihat bahwa Ana merupakan data yang dimasukkan pertama kali berada di ujung kanan, sedangkan data yang terakhir dimasukkan berada di ujung kiri yang ditunjuk pointer pertama kali. Kesimpulannya adalah pointer pertama akan selalu menunjuk pada data yang terakhir dimasukkan. Sifat ini memiliki kemiripan dengan stack atau tumpukan, yaitu data baru akan selalu berada di bagian atas. Namun, perbedaan antara linked list dengan tumpukan yaitu penghapusan datanya bisa dilakukan dimana saja.

9.3 Stack dan Queue dengan Linked List

9.3.1. Stack dengan Singly Linked List

Selain implementasi stack dengan array seperti telah dijelaskan sebelumnya, stack dapat diimplementasikan dengan single linked list. Keunggulannya dibandingkan array adalah penggunaan alokasi memori yang dinamis sehingga menghindari pemborosan memori.

Misalnya pada stack dengan array disediakan tempat untuk stack berisi 150 elemen, sementara ketika dipakai oleh user stack hanya diisi 50 elemen, maka telah terjadi pemborosan memori untuk sisa 100 elemen, yang tak terpakai. Dengan penggunaan linked list maka tempat yang disediakan akan sesuai dengan banyaknya elemen yang mengisi stack.

Dalam stack dengan linked list tidak ada istilah **full**, sebab biasanya program tidak menentukan jumlah elemen stack yang mungkin ada (kecuali jika sudah dibatasi oleh pembuatnya). Namun demikian sebenarnya stack ini pun memiliki batas kapasitas, yakni dibatasi oleh jumlah memori yang tersedia.

Operasi-operasi untuk Stack dengan Linked List

- **IsEmpty**
Fungsi memeriksa apakah stack yang ada masih kosong.
- **Push**
Fungsi memasukkan elemen baru ke dalam stack. Push di sini mirip dengan insert dalam single linked list biasa.
- **Pop**
Fungsi ini mengeluarkan elemen teratas dari stack.
- **Clear**
Fungsi ini akan menghapus stack yang ada.

9.3.2. Queue dengan Doubly Linked List

Selain menggunakan array, queue juga dapat dibuat dengan linked list. Metode linked list yang digunakan adalah double linked list.

Operasi-operasi Queue dengan Doubly Linked List

- **IsEmpty**
Fungsi IsEmpty berguna untuk mengecek apakah queue masih kosong atau sudah berisi data. Hal ini dilakukan dengan mengecek apakah head masih menunjukkan pada Null atau tidak. Jika benar berarti queue masih kosong.
- **IsFull**
Fungsi IsFull berguna untuk mengecek apakah queue sudah penuh atau masih bisa menampung data dengan cara mengecek apakah Jumlah Queue sudah sama dengan MAX_QUEUE atau belum. Jika benar maka queue sudah penuh.
- **EnQueue**
Fungsi EnQueue berguna untuk memasukkan sebuah elemen ke dalam queue (head dan tail mula-mula menunjukan ke NULL).
- **DeQueue**
Prosedure DeQueue berguna untuk mengambil sebuah elemen dari queue. Hal ini dilakukan dengan cara menghapus satu simpul yang terletak paling depan (head).

Pada linked list tidak dikenal istilah full. Hal ini berkaitan dengan penggunaan alokasi memori pada linked list yang lebih dinamis jika dibandingkan dengan array, sehingga pemborosan memory dapat dihindari. Program tidak menentukan jumlah elemen stack yang mungkin ada. Kecuali dibatasi oleh pembuat program dan jumlah memory yang tersedia. Tempat akan sesuai dengan banyaknya elemen yang mengisi stack.

9.3.3 Queue dengan Linked List VS Queue dengan Array

Implementasi queue menggunakan array

- Implementasi sederhana
- Ukuran memori harus ditentukan ketika sebuah objek queue dideklarasikan
- Pemborosan tempat (memori) ketika menggunakan jumlah data yang lebih sedikit dari alokasi memori
- Tidak dapat menambahkan data melebihi maksimal ukuran array yang telah dideklarasikan

Implementasi queue menggunakan linked list

- Pengalokasian memori dinamis
- Menggunakan 2 buah pointer, head dan tail, untuk menandai posisi depan dan belakang dari queue

Contoh perbandingan implementasi Queue, Array VS Linked List

- Memory requirements
 - Array-based implementation
 - Diasumsikan ukuran queue 100 (string @80bytes)
 - Diasumsikan index membutuhkan 2 bytes
 - Total memory: (80 bytes x 101 slots) + (2 bytes x 2 indexes) = 8084 bytes
 - Linked-list-based implementation
 - Diasumsikan pointers membutuhkan 4 bytes
 - Total memory per node: 80 bytes + 4 bytes = 84 bytes

Kesimpulan yang bisa diambil dari perbandingan antara Stack-Queue dengan Linked List Vs Stack-Queue dengan Array antara lain:

- Untuk stack dan queue yang berukuran besar, terutama jumlah maksimal data tidak diketahui, lebih baik menggunakan linked list.
- Untuk perangkat yang memiliki memori terbatas, seperti small handheld devices, linked list memiliki performa yang lebih bagus.

9.4 Studi Kasus

```
#include <iostream>
using namespace std;

// deklarasi single linked list
struct Buku{

    // komponen / member
    string judul, pengarang;
    int tahunTerbit;

    Buku *next;

};

Buku *head, *tail, *cur, *newNode, *del, *before;

// create single linked list
void createSingleLinkedList(string judul, string pengarang, int tB){
    head = new Buku();
    head->judul = judul;
    head->pengarang = pengarang;
    head->tahunTerbit = tB;
    head->next = NULL;
    tail = head;
}

// print single linked list
int countSingleLinkedList(){
    cur = head;
    int jumlah = 0;
    while( cur != NULL ){
        jumlah++;
        cur = cur->next;
    }
    return jumlah;
}

// tambahAwal Single linked list
void addFirst(string judul, string pengarang, int tB){
    newNode = new Buku();
    newNode->judul = judul;
    newNode->pengarang = pengarang;
    newNode->tahunTerbit = tB;
    newNode->next = head;
    head = newNode;
}
```

```

}

// tambahAkhir Single linked list
void addLast(string judul, string pengarang, int tB){
    newNode = new Buku();
    newNode->judul = judul;
    newNode->pengarang = pengarang;
    newNode->tahunTerbit = tB;
    newNode->next = NULL;
    tail->next = newNode;
    tail = newNode;
}

// tambah tengah single linked list
void addMiddle(string judul, string pengarang, int tB, int posisi){
    if( posisi < 1 || posisi > countSingleLinkedList() ){
        cout << "Posisi diluar jangkauan" << endl;
    }else if( posisi == 1){
        cout << "Posisi bukan posisi tengah" << endl;
    }else{
        newNode = new Buku();
        newNode->judul = judul;
        newNode->pengarang = pengarang;
        newNode->tahunTerbit = tB;

        // tranversing
        cur = head;
        int nomor = 1;
        while( nomor < posisi - 1 ){
            cur = cur->next;
            nomor++;
        }
        newNode->next = cur->next;
        cur->next = newNode;
    }
}

// Remove First
void removeFirst(){
    del = head;
    head = head->next;
    delete del;
}

// Remove Last
void removeLast(){
    del = tail;

```



```

    cur = head;
    while( cur->next != tail ){
        cur = cur->next;
    }
    tail = cur;
    tail->next = NULL;
    delete del;
}

// remove middle
void removeMiddle(int posisi){
    if( posisi < 1 || posisi > countSingleLinkedList() ){
        cout << "Posisi diluar jangkauan" << endl;
    }else if( posisi == 1){
        cout << "Posisi bukan posisi tengah" << endl;
    }else{
        int nomor = 1;
        cur = head;
        while( nomor <= posisi ){
            if( nomor == posisi-1 ){
                before = cur;
            }
            if( nomor == posisi ){
                del = cur;
            }
            cur = cur->next;
            nomor++;
        }
        before->next = cur;
        delete del;
    }
}

// ubahAwal Single linked list
void changeFirst(string judul, string pengarang, int tB){
    head->judul = judul;
    head->pengarang = pengarang;
    head->tahunTerbit = tB;
}

// ubahAkhir Single linked list
void changeLast(string judul, string pengarang, int tB){
    tail->judul = judul;
    tail->pengarang = pengarang;
    tail->tahunTerbit = tB;
}

// ubah Tengah Single linked list

```

```

void changeMiddle(string judul, string pengarang, int tB, int posisi){
    if( posisi < 1 || posisi > countSingleLinkedList() ){
        cout << "Posisi diluar jangkauan" << endl;
    }else if( posisi == 1 || posisi == countSingleLinkedList() ){
        cout << "Posisi bukan posisi tengah" << endl;
    }else{
        cur = head;
        int nomor = 1;
        while( nomor < posisi ){
            cur = cur->next;
            nomor++;
        }
        cur->judul = judul;
        cur->pengarang = pengarang;
        cur->tahunTerbit = tB;
    }
}

// print single linked list
void printSingleLinkedList(){
    cout << "Jumlah data ada : " << countSingleLinkedList() << endl;
    cur = head;
    while( cur != NULL ){
        cout << "Judul Buku : " << cur->judul << endl;
        cout << "Pengarang Buku : " << cur->pengarang << endl;
        cout << "Tahun Terbit Buku : " << cur->tahunTerbit << endl;

        cur = cur->next;
    }
}

int main(){
    createSingleLinkedList("Kata", "Geez & Aan", 2018);
    printSingleLinkedList();
    cout << "\n\n" << endl;

    addFirst("Dia adalah Kakakku", "Tere Liye", 2009);
    printSingleLinkedList();
    cout << "\n\n" << endl;

    addLast("Aroma Karsa", "Dee Lestari", 2018);
    printSingleLinkedList();
    cout << "\n\n" << endl;

    removeFirst();
    printSingleLinkedList();
    cout << "\n\n" << endl;
}

```

```
addLast("11.11", "Fiersa Besari", 2018);
printSingleLinkedList();
cout << "\n\n" << endl;

removeLast();
printSingleLinkedList();
cout << "\n\n" << endl;

changeFirst("Berhenti di Kamu", "Gia Pratama", 2018);
printSingleLinkedList();
cout << "\n\n" << endl;

addMiddle("Bumi Manusia", "Pramoedya Anata Toer", 2005, 2);
printSingleLinkedList();
cout << "\n\n" << endl;

addMiddle("Negeri 5 Menara", "Ahmad Fuadi", 2009, 2);
printSingleLinkedList();
cout << "\n\n" << endl;

removeMiddle(5);
printSingleLinkedList();
cout << "\n\n" << endl;

changeMiddle("Sang Pemimpi", "Andrea Hirata", 2006, 2);
printSingleLinkedList();

cout << "\n\n" << endl;

}
```

9.4 Latihan

1. Terangkan proses penyisipan data pada Linked List, sebutkan 3 macam kondisi yang mungkin?
2. Tambahkan fungsi pencarian data pada Linked List!
3. Tambahkan fungsi pengurutan data berdasarkan tahun terbit!
4. Terangkan mekanisme penghapusan data pada Linked List?
5. Jelaskan dan ilustrasikan perbedaan Linked List Non-Circular dengan Linked List Circular!

10

Graph

Sub CPMK

Mahasiswa mampu merancang metode *graph* untuk pengelolaan data

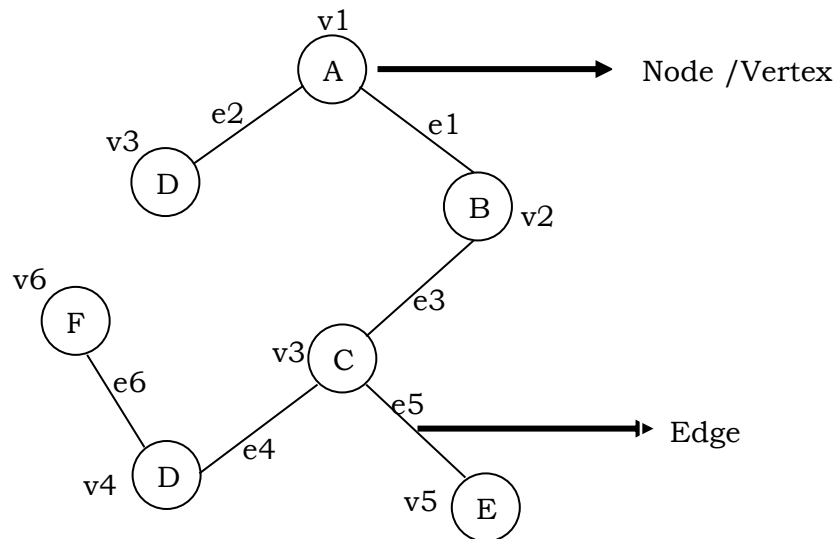
Dalam bab ini akan dibahas tentang struktur data *graph*. Pembahasan diawali dengan pengertian *graph* kemudian istilah-istilah dalam *graph*. Setelah itu akan dibahas representasi *graph*, penerapan *graph* pada array dan operasi pada *graph*. Pada akhir materi akan diberikan Studi Kasus beserta Latihan.

10.1 Pengertian

Dalam matematika dan ilmu komputer, teori *graph* adalah studi tentang *graph*: struktur matematika yang digunakan untuk memodelkan hubungan berpasangan antara objek dari koleksi tertentu.

Graph dalam konteks ini mengacu pada kumpulan *vertex* atau *node* dan kumpulan *edge* yang menghubungkan pasangan *vertex*. Sebuah *graph* mungkin tidak berarah (yang berarti tidak ada perbedaan antara dua simpul yang berasosiasi dengan masing-masing *edge*) atau *edge* dapat diarahkan dari satu *vertex* ke *vertex* lainnya (yang berarti sisi-sisi yang ada memiliki arah tertentu dari sebuah simpul ke simpul lainnya, namun belum tentu ada untuk arah sebaliknya).

Jika suatu nilai diberikan pada *edge* suatu *graph*, maka *graph* tersebut disebut *graph* berbobot. Jika bobot semua *edge*-nya sama, maka *graph* tersebut disebut *graph* tidak berbobot.



Gambar 10. 1 Contoh graph

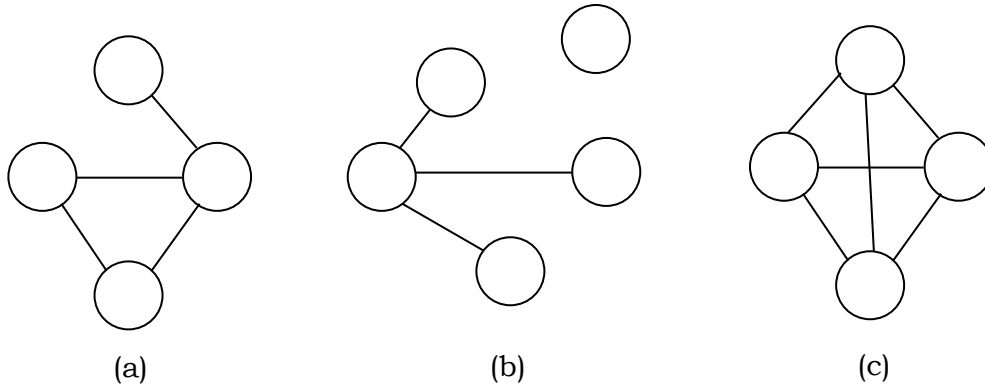
$$V = \{v1, v2, \dots, v6\}$$

$$E = \{e1, e2, \dots, e6\} \text{ atau}$$

$$E = \{(v1,v2), (v1,v3), \dots, (v4,v6)\}$$

$v2$ bertetangga dengan $v1$ dan $v3$

Dua titik u, v dikatakan *adjacent to u*, jika u dan v dihubungkan dengan garis. Path antara dua titik v dan w adalah sekumpulan garis yang menghubungkan titik v ke titik w . Panjang path adalah jumlah garis dalam path. Graph dikatakan *connected*/terhubung jika ada sebuah path yang menghubungkan sembarang dua titik berbeda. Ilustrasi dari graph yang dikatakan *connected*, *disconnected* dan *complete* dapat dilihat pada Gambar 10.2.



Gambar 10. 2 (a) Graph Connected (b) Disconnected (c) Complete

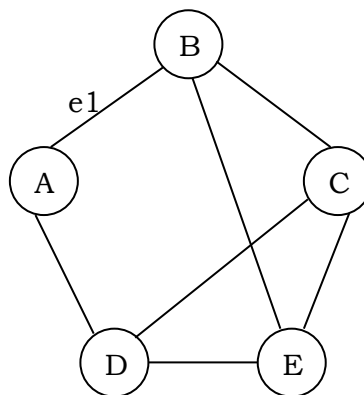
10.1.1 Jenis-jenis Graph

Jenis *graph* yang paling umum dalam struktur data yaitu :

1. *Undirected Graph* / Tidak terarah:

Pada *graph* tak berarah (*undirected graph*), garis tidak mempunyai arah dan dituliskan sebagai pasangan $\{u,v\}$ atau $u \leftrightarrow v$. *Graph* tak berarah merupakan *graph* berarah jika setiap garis tak berarah $\{u,v\}$ merupakan dua garis berarah $\langle u,v \rangle$ dan $\langle v,u \rangle$.

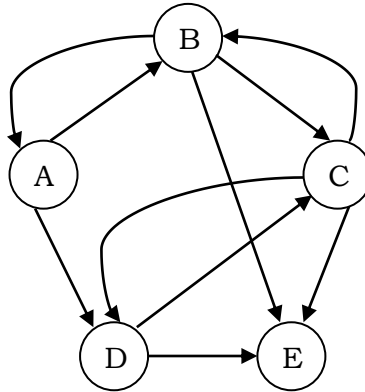
Jadi, dapat dikatakan bahwa *graph* yang semua sisinya dua arah. Ujung-ujungnya tidak menunjuk ke arah tertentu sehingga urutan simpul dalam sebuah busur tidak dipentingkan. Misalnya saja pada Gambar 10.3 busur e_1 dapat disebut busur AB atau BA.



Gambar 10. 3 Graph tidak terarah

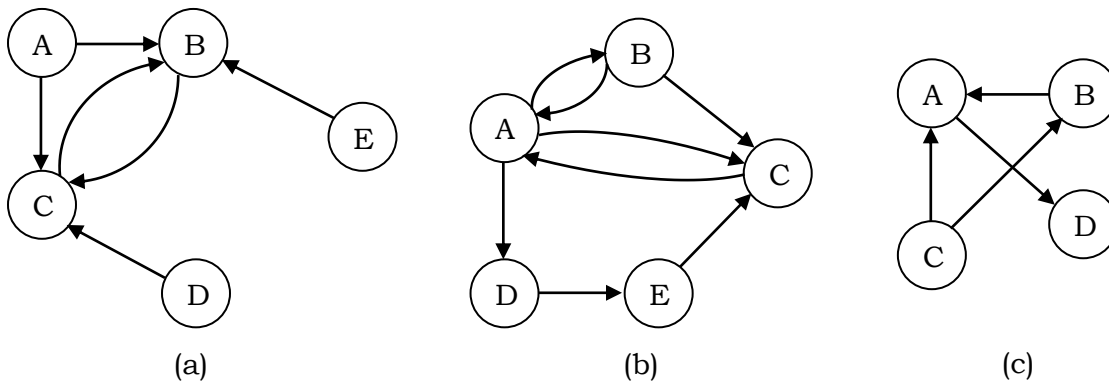
2. *Directed Graph* / Terarah:

Merupakan *graph* yang semua sisinya satu arah. Ujung-ujungnya menunjuk ke satu arah. Contoh sederhananya dapat dilihat pada Gambar 10.4.



Gambar 10. 4 Graph berarah

Pada graph berarah (*directed graph/ digraph*), setiap garis mempunyai arah dari u ke v dan dituliskan sebagai pasangan $\langle u,v \rangle$ atau $u \rightarrow v$. *Complete graph* adalah *graph* yang terhubung (*connected graph*) dan setiap pasang titik dihubungkan dengan garis. *Digraph* disebut *strongly connected* jika ada sebuah path dari sembarang titik ke semua titik. *Digraph* disebut *weakly conneted* jika titik u dan v jika terdapat path (u,v) atau path (v,u) . Ilustrasi dari *connected graph* dapat dilihat pada Gambar 10.5.



Gambar 10. 5 (a) Graph Weakly Connected (b) Strongly Connected
(c) Weakly Connected

Berdasarkan Gambar 10.5 (a) merupakan *digraph* yang *Not Strongly* atau *Weakly Connected* karena tidak ada path dari E ke D atau dari D ke E. Sedangkan Gambar 10.5 (c) merupakan *digraph* yang *Weakly Connected* karena tidak ada path dari D ke simpul lainnya.

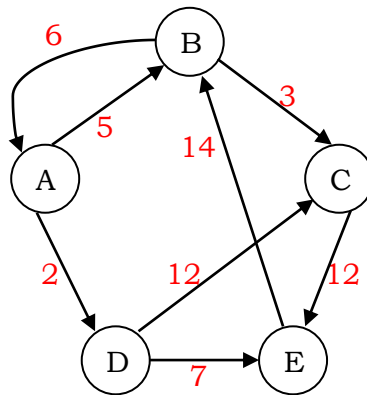
3. *Weighted Graph* / Graf Berbobot:

Baik graph berarah maupun tak berarah dapat diberikan bobot (*weight*). Bobot diberikan untuk setiap garis. Hal ini biasanya digunakan untuk menggambarkan jarak antara dua kota, biaya transportasi, waktu tempuh, harga tiket, kapasitas

elektrik suatu kabel atau ukuran lain yang berhubungan dengan garis. *Graph* berbobot biasanya juga digunakan dalam pemodelan jaringan komputer.

Sisi dalam *graph* berbobot direpresentasikan sebagai (u, v, w) , di mana:

- u adalah simpul sumber
- v adalah simpul tujuan
- w mewakili bobot yang terkait untuk berpindah dari u ke v



Gambar 10. 6 *Graph* berbobot

Graph banyak digunakan untuk menggambarkan jaringan dan peta jalan, jalan kereta api, lintasan pesawat, system perpipaan, saluran telepon, koneksi elektrik, ketergantungan diantara task pada sistem manufaktur dan lain-lain. Terdapat banyak hasil dan struktur penting yang didapatkan dari perhitungan dengan *graph*.

4. *Unweighted Graph* / *Graph* Tidak Berbobot: *Graph* yang tidak ada nilai atau bobot yang terkait dengan edge. Semua *graph* tidak berbobot secara default kecuali ada nilai yang terkait.

Sisi dari *graph* tak berbobot direpresentasikan sebagai (u, v) , di mana:

- u mewakili simpul sumber
- v adalah simpul tujuan

10.1.2 Representasi Graph

Graph biasanya direpresentasikan dalam dua cara:

1. *Adjacency List*

Dalam representasi ini, setiap simpul memiliki daftar simpul mana yang berdekatan / *adjacent*. Hal ini menyebabkan redundansi dalam *graph* tak terarah: misalnya, jika simpul A dan B bertetangga, maka *adjacency list* A berisi B, sedangkan *list* B berisi A. Query *adjacency* lebih cepat, dengan biaya ruang penyimpanan ekstra.

Sebagai contoh, *graph* tak berarah pada Gambar 10.1 dapat direpresentasikan menggunakan *adjacency list* sebagai berikut:

Tabel 10.1 Representasi *Adjacency List*

Node	<i>Adjacency List</i>
A	B,F
B	A,D
C	D
D	B,C,E
E	D,G
F	A
G	E

Contoh program C++ *Adjacency List* seperti terdapat pada kode program di bawah ini :

```
// Representasi sederhana dari graph menggunakan STL
#include <bits/stdc++.h>
using namespace std;
// Fungsi utilitas untuk menambahkan edge pada graph tak berarah.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
// Fungsi utilitas untuk mencetak representasi adjacency list dari
graph
void printGraph(vector<int> adj[], int V)
{
    for (int v = 0; v < V; ++v) {
        cout << "\n Adjacency list of vertex " << v
            << "\n head ";
        for (auto x : adj[v])
            cout << "-> " << x;
        printf("\n");
    }
}

// Driver code
int main()
{
    int V = 7;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 5);
    addEdge(adj, 1, 3);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    addEdge(adj, 4, 6);
```

```

    printGraph(adj, V);
    return 0;
}

```

2. Adjacency Matrix

Adjacency Matrix adalah matriks n kali n , di mana n adalah jumlah simpul dalam *graph*. Jika ada *edge* dari suatu *vertex* x ke beberapa *vertex* y , maka elemen $a_{x,y}$ adalah 1 (atau, dalam kasus *graph* berbobot, bobot sisi yang menghubungkan x dan y), selain itu adalah 0. Dalam komputasi, matriks ini memudahkan untuk menemukan sub *graph*, dan membalikkan *graph* berarah.

Sebagai contoh, *graph* tak berarah pada Gambar 10.1 dapat direpresentasikan menggunakan matriks ketetanggaan sebagai berikut:

Tabel 10.2 *Matriks Adjacency*

	A	1	B	C	D	E	F
A	0	1	0	0	0	1	0
B	1	0	0	1	0	0	0
C	0	0	0	1	0	0	0
D	0	1	1	0	1	0	0
E	0	0	0	1	0	0	1
F	1	0	0	0	0	0	0
G	0	0	0	0	1	0	0

Contoh program C++ *Adjacency Matrix* seperti terdapat pada kode program di bawah ini :

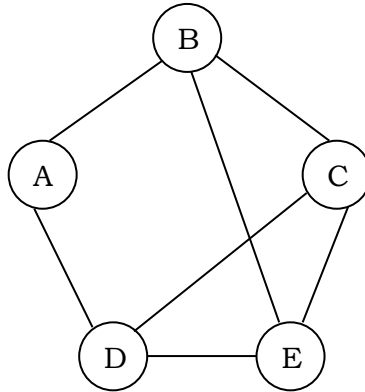
```

#include <iostream>
using namespace std;
int main()
{
    int n, m;
    cin >> n >> m ;
    int adjMat[n + 1][m + 1];
    for(int i = 0; i < m; i++){
        int u , v ;
        cin >> u >> v ;
        adjMat[u][v] = 1 ;
        adjMat[v][u] = 1 ;
    }
    return 0;
}

```

Contoh lainnya dalam membuat *adjacency matrix* akan dibahas dalam simulasi berikut ini:

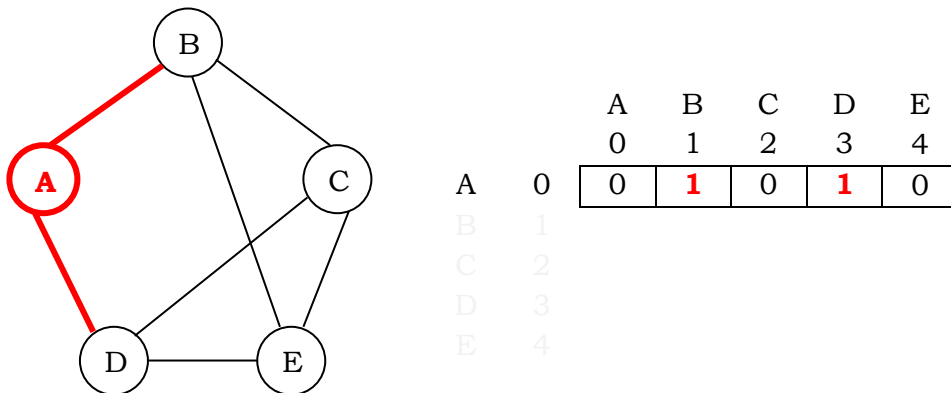
Simulasi 1. Graph tak berarah



Gambar 10. 7 Contoh simulasi graph tak berarah

Tahap 1.

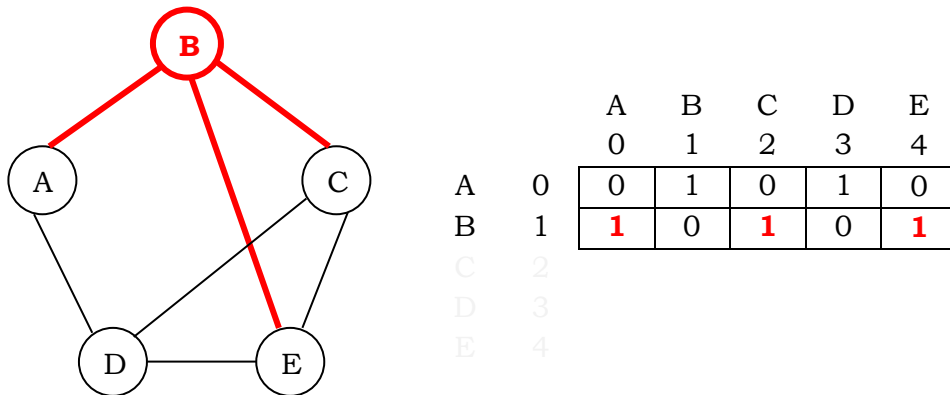
Perhatikan node pertama, yaitu A, node lain yang terhubung dengan node A ini akan kita beri nilai logika 1 dan kita isikan di baris pertama.



Gambar 10. 8 Langkah 1 Pengisian Adjacency Matrix Graph Tak Berarah

Tahap 2.

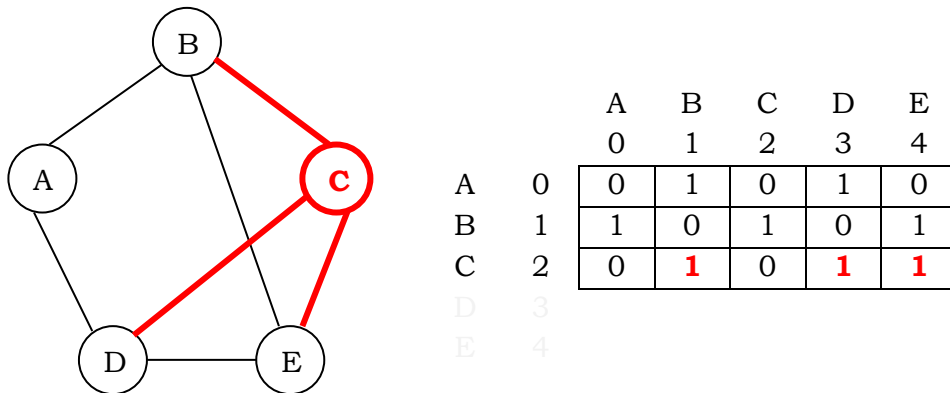
Perhatikan node kedua, yaitu B, node lain yang terhubung dengan node B ini akan kita beri nilai logika 1 dan kita isikan di baris kedua.



Gambar 10. 9 Langkah 2 Pengisian Adjacency Matrix Graph Tak Berarah

Tahap 3.

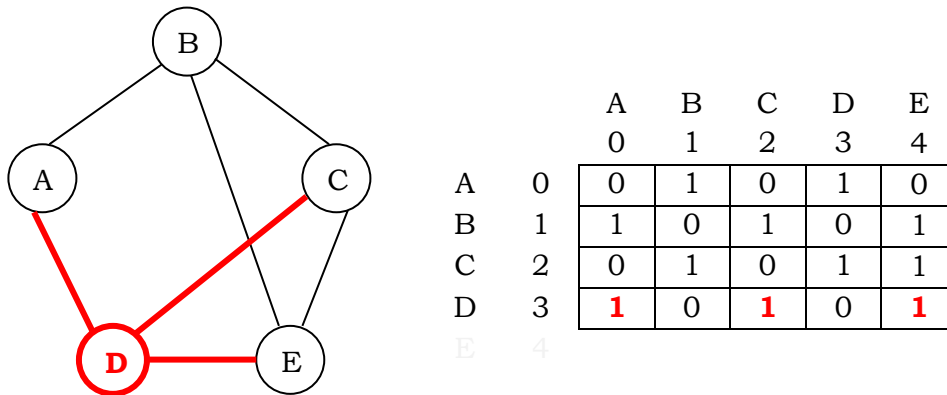
Perhatikan node ketiga, yaitu C, node lain yang terhubung dengan node C ini akan kita beri nilai logika 1 dan kita isikan di baris ketiga.



Gambar 10. 10 Langkah 3 Pengisian Adjacency Matrix Graph Tak Berarah

Tahap 4.

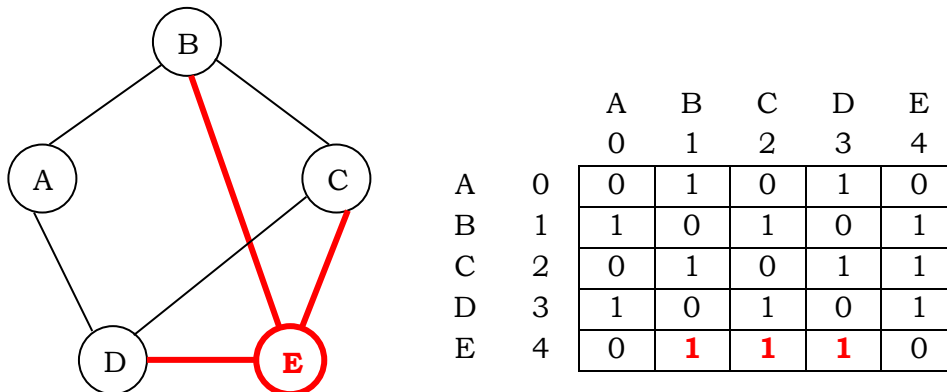
Perhatikan node pertama, yaitu D, node lain yang terhubung dengan node D ini akan kita beri nilai logika 1 dan kita isikan di baris keempat.



Gambar 10. 11 Langkah 4 Pengisian Adjacency Matrix Graph Tak Berarah

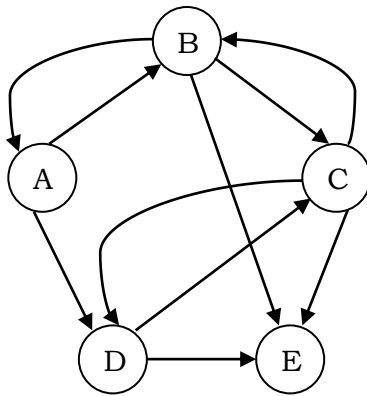
Tahap 5.

Perhatikan node pertama, yaitu E, node lain yang terhubung dengan node E ini akan kita beri nilai logika 1 dan kita isikan di baris kelima.



Gambar 10. 12 Langkah 5 Pengisian Adjacency Matrix Graph Tak Berarah

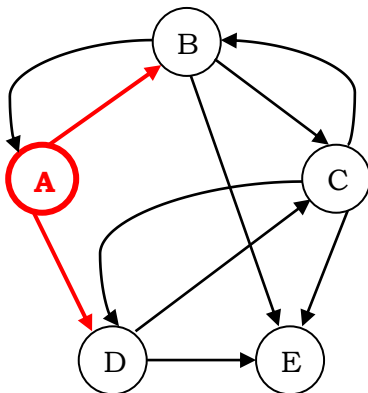
Simulasi 2. Graph Berarah



Gambar 10. 13 Contoh Simulasi Graph Berarah

Tahap 1.

Perhatikan node pertama, yaitu A, node lain yang dituju oleh arah panah dari node A ini akan kita beri nilai logika 1 dan kita isikan di baris pertama.

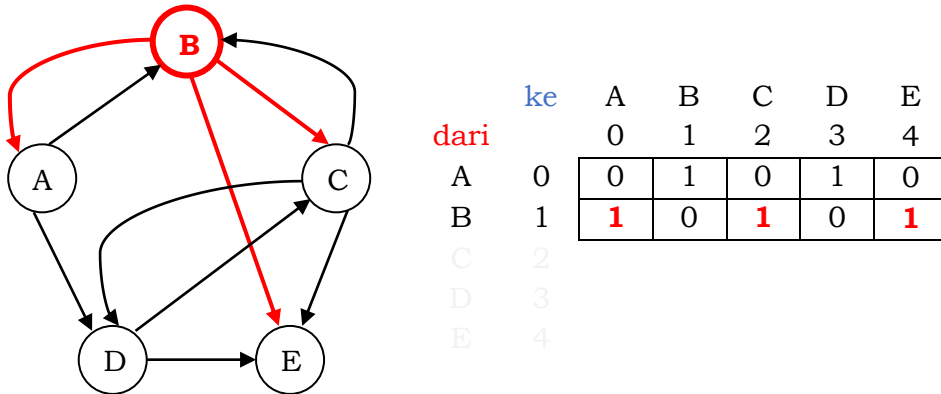


dari	ke	A	B	C	D	E
		0	1	2	3	4
A	0	0	1	0	1	0
B	1	1	0	1	0	1
C	2	0	1	0	1	1
D	3	0	0	1	0	1
E	4	0	0	0	0	0

Gambar 10. 14 Langkah 1 Pengisian Adjacency Matrix Graph Berarah

Tahap 2.

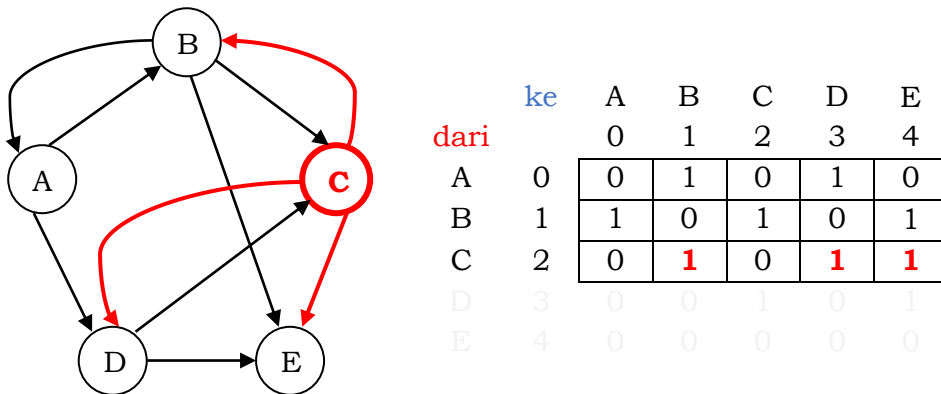
Perhatikan node kedua, yaitu B, node lain yang dituju oleh arah panah dari node B ini akan kita beri nilai logika 1 dan kita isikan di baris pertama.



Gambar 10. 15 Langkah 2 Pengisian Adjacency Matrix Graph Berarah

Tahap 3.

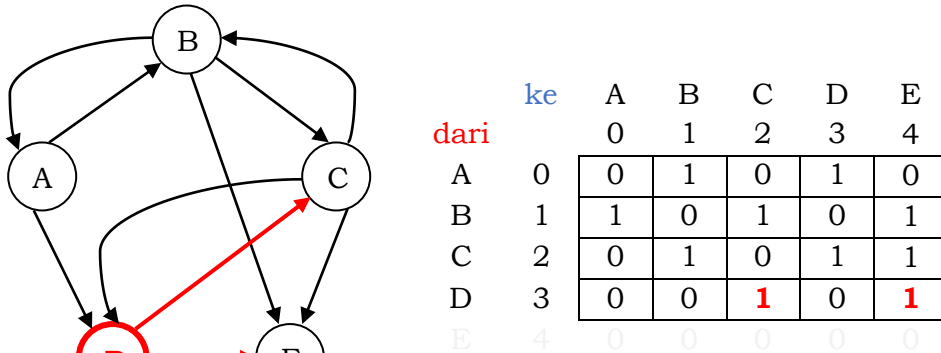
Perhatikan node pertama, yaitu C, node lain yang dituju oleh arah panah dari node C ini akan kita beri nilai logika 1 dan kita isikan di baris pertama.



Gambar 10. 16 Langkah 3 Pengisian Adjacency Matrix Graph Berarah

Tahap 4.

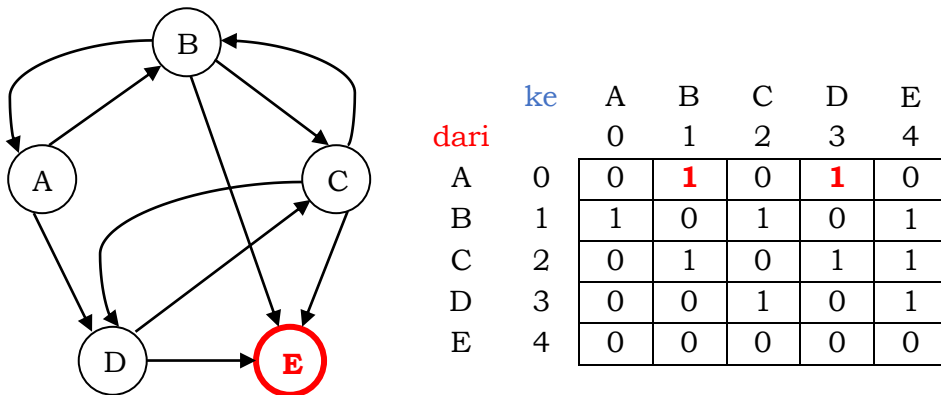
Perhatikan node pertama, yaitu D, node lain yang dituju oleh arah panah dari node D ini akan kita beri nilai logika 1 dan kita isikan di baris pertama.



Gambar 10. 17 Langkah 4 Pengisian Adjacency Matrix Graph Berarah

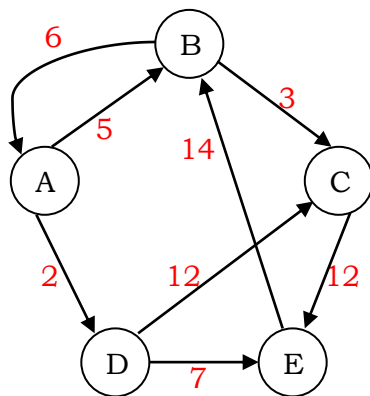
Tahap 5.

Perhatikan node E, arahnya menuju ke node mana saja. Karena node E tidak mengarah ke node lainnya sehingga semua nilainya 0.



Gambar 10. 18 Langkah 5 Pengisian Adjacency Matrix Graph Berarah

Simulasi 3. Graph berbobot

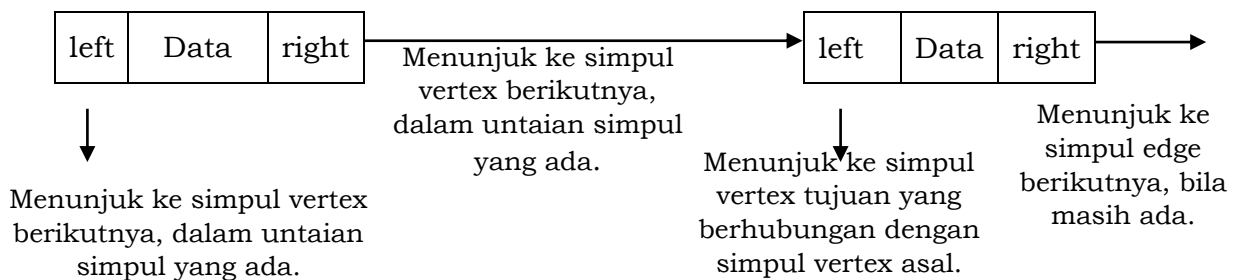


	ke	A	B	C	D	E
dari		0	1	2	3	4
A	0	0	5	0	2	0
B	1	6	0	3	0	0
C	2	0	0	0	0	9
D	3	0	0	12	0	7
E	4	0	14	0	0	0

Gambar 10. 19 Representasi Adjacency Matrix Graph Berarah dan Berbobot

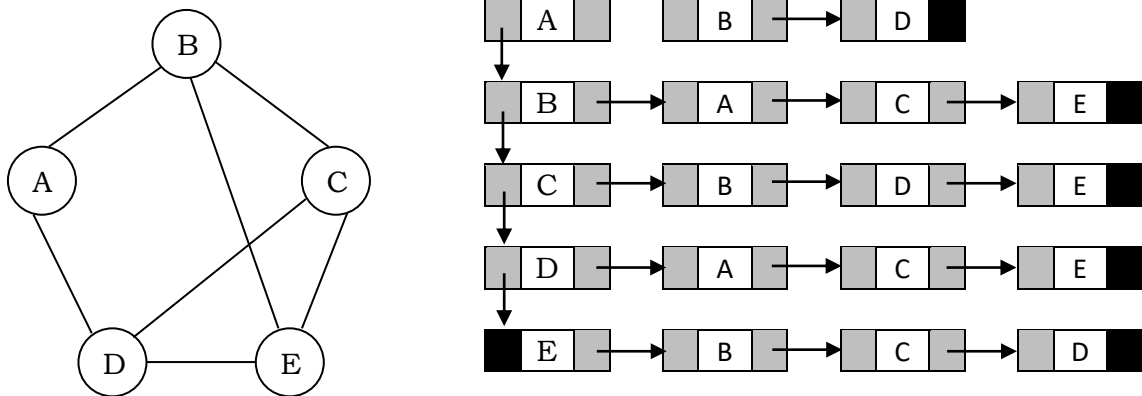
3. Adjacency List dalam bentuk Linked List

Representasi graph dalam bentuk *Linked List* ini memiliki kemiripan dengan *adjacency list* dalam vector yang dibahas sebelumnya, setiap simpul memiliki daftar simpul mana yang berdekatan / *adjacent*. Hal ini menyebabkan redundansi dalam *graph* tak terarah. Dalam representasinya dapat menggunakan *Doubly Linked List* yang memiliki sebuah simpul dengan 2 pointer left dan right.



Gambar 10. 20 Representasi Graph pada Linked List

Sebagai contoh, *graph* tak berarah dapat direpresentasikan menggunakan adjacency list dalam *linked list* seperti pada Gambar 10.21.



Gambar 10. 21 Contoh Representasi Graph Tak Berarah pada Linked List

10.1.3 Mengambil *graph* sebagai input dari keyboard dan menyimpannya ke dalam memori

Lebih mudah untuk menyimpan *graph* ke dalam memori dan melakukan operasi di atasnya menggunakan matriks adjacency daripada daftar adjacency. Pertama, kita perlu mendeklarasikan array 2D global. Kemudian, kita harus meminta pengguna untuk memasukkan *edge*. Pengguna harus memberikan 2 angka (x, y) yang mewakili *edge* antara dua simpul tersebut (simpul akan dilambangkan sebagai angka, bukan huruf). Untuk setiap pasangan (x, y), lokasi yang sesuai a_x, y pada matriks akan diisi dengan '1'. Sebagai contoh adalah kasus untuk *graph* tak berbobot, tetapi berarah. Kasus *graph* tidak berbobot, tetapi *graph* berarah, lokasi a_y, x juga harus diisi dengan '1'. Dalam kasus *graph* berbobot, pengguna harus memasukkan nomor lain yang menunjukkan bobot, dan nomor itu harus disimpan di lokasi alih-alih '1'. Setelah input tepi, semua lokasi lain harus diisi dengan '0'. Namun, dalam kasus C, C++ atau Java, ini dilakukan secara otomatis karena variabel global secara otomatis terisi nol saat diinisialisasi.

Program C/C++ sederhana untuk memasukkan dan menyimpan *graph* tak berbobot terarah

```

#include <stdio.h>
int graph[100][100]; //Matriks untuk menyimpan graph yang berisi
maksimum 100 node
/* int main(int argc, char** argv) { */
    printf("Masukkan Jumlah Edge: ");
    int edges;
    scanf("%d", &edges);
    /* for (int i = 1; i <= edges; i++) { */

```

```

        printf("Masukkan Edge %d: ", i);
        int x, y;                                //Atau, int x, y, weight; - for
storing weight of edge
        scanf("%d %d", &x, &y); //Atau, scanf("%d %d %d", &x, &y,
&weight); - for weighted graph
        graph[x][y] = 1;                        //Atau, graph[x][y] = weight; - for
weighted graph
        //graph[y][x] = 1;                      //Baris ini harus ditambahkan untuk
graph tidak berarah
    }
    return 0;
}

```

10.2 Operasi

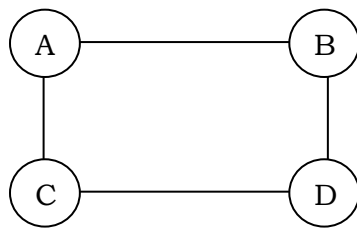
Berikut ini adalah operasi grafik dasar dalam struktur data:

1. Add/Remove Vertex – Menambah atau menghapus vertex dalam *graph*.
2. Tambah/Hapus Tepi – Menambah atau menghapus tepi di antara dua simpul.
3. Periksa apakah *graph* berisi nilai yang diberikan.
4. Temukan jalur dari satu simpul ke simpul lainnya.

10.3 Istilah

Ada beberapa istilah dalam graph, yaitu :

1. Walk disebut tertutup, yang menghubungkan V_1 dan V_n , yaitu setiap Ruas menghubungkan simpul awal dan akhir.



Gambar 10. 22 Graph tertutup

2. Trail adalah Walk dengan semua ruas dalam barisan adalah berbeda
3. Path atau jalur adalah Walk yang semua simpul dalam barisan adalah berbeda.
Jadi suatu path pastilah sebuah trail

10.4 Algoritma Traversal Graph

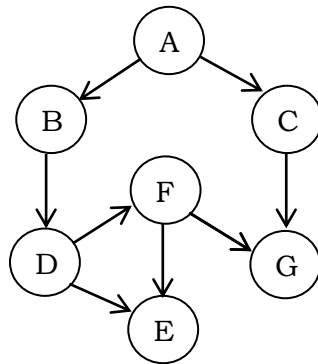
10.4.1 *Breadth-First Search (BFS)*

Algoritma pencarian *Breadth-First Search* (BFS) disebut juga Pencarian Melebar Pertama. Dalam teori *graph*, *Breadth-First Search* (BFS) adalah pencarian *graph* algoritma yang dimulai pada simpul akar (level n). Kunjungan selanjutnya adalah simpul-simpul pada level $n+1$. Pencarian dimulai dari vertex awal terus ke level ke-1 dari kiri ke kanan, kemudian berpindah ke level berikutnya demikian pula dari kiri ke kanan sampai ditemukannya solusi.

Algoritma BFS

1. Tandai semua Vertex yang terdapat pada Graph dengan warna WHITE
2. Tentukan Vertex Awal
3. Buat sebuah Queue, masukkan vertex awal ke Queue, tandai dengan warna GRAY
4. Ambil vertex dari Queue (sebut Vertex P), tandai dengan BLACK, langsung tulis jika Vertex $P \neq \text{GOAL}$, diganti dengan NEIGHBORS/tetangganya (pilih Vertex yang masih berwarna WHITE), masukkan dalam QUEUE, tandai Vertex-vertex tersebut dengan warna GRAY. Lakukan pengulangan langkah 5.
5. Bila vertex $P = \text{GOAL}$, selesai

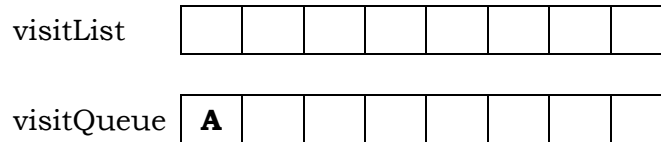
Sebagai contoh, kita lakukan algoritma traversal pada graph pada gambar 10.22



Gambar 10. 23 Contoh Graph pada Ilustrasi Algoritma Traversal

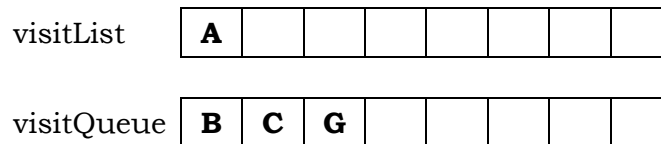
Langkah-langkah algoritma traversal Graph menggunakan BFS.

- Terdapat dua QUEUE yaitu visitQueue untuk menyimpan Vertex-Vertex yang sedang dikunjungi dan visitList untuk Vertex-Vertex yang sudah dikunjungi.
- Warnai semua Vertex dengan WHITE dan masukan Vertex Awal yaitu A ke visitQueue, warna Vertex A dengan GRAY



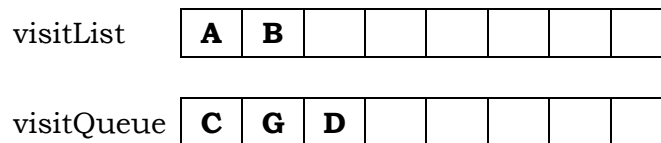
Gambar 10. 24 Langkah 1 Algoritma Traversal Graph: BFS

- Ambil A dari visitQueue, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex A yang masih berwarna WHITE yaitu B, C dan G, masukkan dalam visitQueue. Warnai Vertex B, C, G dengan GRAY.



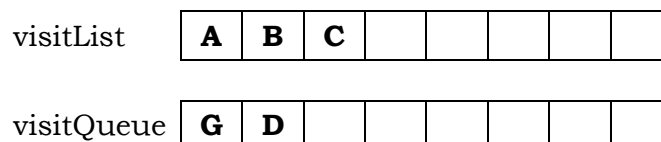
Gambar 10. 25 Langkah 2 Algoritma Traversal Graph: BFS

- Ambil B dari visitQueue, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex B yang masih berwarna WHITE yaitu D, masukkan dalam visitQueue. Warnai Vertex D dengan GRAY.



Gambar 10. 26 Langkah 3 Algoritma Traversal Graph: BFS

- Ambil C dari visitQueue, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex C yang masih berwarna WHITE. Tetangga dari Vertex C adalah G tapi berwarna GRAY, sehingga tidak ada Vertex yang dapat dimasukkan dalam visitQueue.



Gambar 10. 27 Langkah 4 Algoritma Traversal Graph: BFS

- Ambil G dari visitQueue, warnai dengan BLACK dan masukkan dalam visitList. Vertex G tidak memiliki tetangga sehingga tidak ada Vertex yang dapat dimasukkan dalam visitQueue.

visitList	A	B	C	G				
-----------	----------	----------	----------	----------	--	--	--	--

visitQueue	D							
------------	----------	--	--	--	--	--	--	--

Gambar 10. 28 Langkah 5 Algoritma Traversal Graph: BFS

- Ambil D dari visitQueue, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex D yang masih berwarna WHITE yaitu E dan F, masukkan dalam visitQueue. Warnai Vertex E dan F dengan GRAY.

visitList	A	B	C	G	D			
-----------	----------	----------	----------	----------	----------	--	--	--

visitQueue	E	F						
------------	----------	----------	--	--	--	--	--	--

Gambar 10. 29 Langkah 6 Algoritma Traversal Graph: BFS

- Ambil E dari visitQueue, warnai dengan BLACK dan masukkan dalam visitList. Vertex E tidak memiliki tetangga sehingga tidak ada Vertex yang dapat dimasukkan dalam visitQueue.

visitList	A	B	C	G	D	E		
-----------	----------	----------	----------	----------	----------	----------	--	--

visitQueue	F							
------------	----------	--	--	--	--	--	--	--

Gambar 10. 30 Langkah 7 Algoritma Traversal Graph: BFS

- Ambil F dari visitQueue, warnai dengan BLACK dan masukkan dalam visitList. Vertex F tidak memiliki tetangga sehingga tidak ada Vertex yang dapat dimasukkan dalam visitQueue.

visitList	A	B	C	G	D	E	F	
-----------	----------	----------	----------	----------	----------	----------	----------	--

visitQueue								
------------	--	--	--	--	--	--	--	--

Gambar 10. 31 Langkah 8 Algoritma Traversal Graph: BFS

- Karena Queue kosong maka proses traversal selesai

10.4.2 Depth First Search (DFS)

Algoritma pencarian *Depth-First Search* (BFS) disebut juga Pencarian Mendalam Pertama. Pada Depth First Search, proses pencarian akan dilaksanakan pada semua anaknya sebelum dilakukan pencarian ke node-node yang selevel. Pencarian dimulai dari node akar ke level yang lebih tinggi. Proses ini diulangi terus hingga ditemukannya solusi.

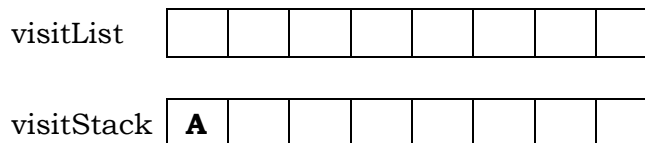
Algoritma DFS

1. Tandai semua Vertex yang terdapat pada Graph dengan warna WHITE
2. Tentukan Vertex Awal
3. Buat sebuah Stack, masukkan vertex awal ke Stack, tandai dengan warna GRAY
4. Ambil vertex dari Stack (sebut Vertex P), tandai dengan BLACK, langsung tulis
5. jika Vertex P \neq GOAL, diganti dengan NEIGHBORS/tetangganya (pilih Vertex yang masih berwarna WHITE), masukkan dalam Stack, tandai Vertex-vertex tersebut dengan warna GRAY. Lakukan pengulangan langkah 5.
6. Bila vertex P = GOAL, selesai

Sebagai contoh, kita lakukan algoritma traversal pada graph pada gambar 10.21.

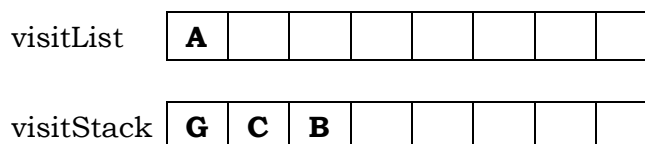
Langkah-langkah algoritma traversal Graph menggunakan DFS:

- Terdapat Stack yaitu visitStack untuk menyimpan Vertex-Vertex yang sedang dikunjungi dan Queue untuk visitList untuk Vertex-Vertex yang sudah dikunjungi.
- Warnai semua Vertex dengan WHITE dan masukan Vertex Awal yaitu A ke visitStack, ubah warna Vertex A menjadi GRAY



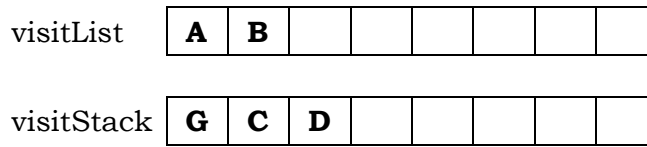
Gambar 10. 32 Langkah 1 Algoritma Traversal Graph: DFS

- Ambil A dari visitStack, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex A yang masih berwarna WHITE yaitu B, C dan G, masukkan dalam visitStack. Warnai Vertex B, C, G dengan GRAY.



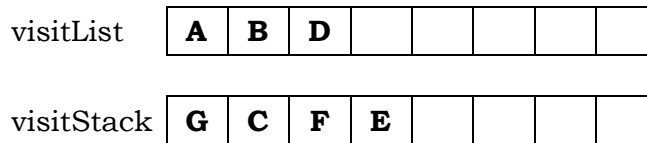
Gambar 10. 33 Langkah 2 Algoritma Traversal Graph: DFS

- Ambil B dari visitStack, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex B yang masih berwarna WHITE yaitu D, masukkan dalam visitStack. Warnai Vertex D dengan GRAY.



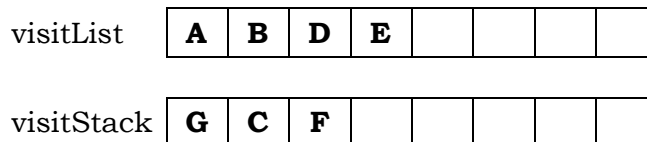
Gambar 10. 34 Langkah 3 Algoritma Traversal Graph: DFS

- Ambil D dari visitStack, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex D yang masih berwarna WHITE yaitu E dan F.



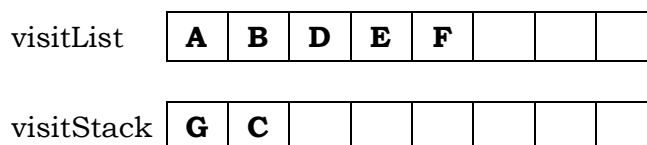
Gambar 10. 35 Langkah 4 Algoritma Traversal Graph: DFS

- Ambil E dari visitStack, warnai dengan BLACK dan masukkan dalam visitList. Vertex E tidak memiliki tetangga sehingga tidak ada Vertex yang dapat dimasukkan dalam visitStack.



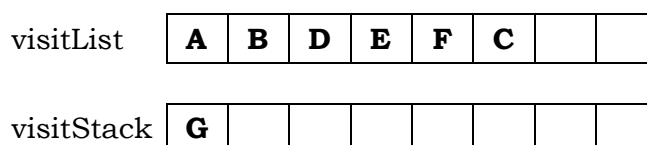
Gambar 10. 36 Langkah 5 Algoritma Traversal Graph: DFS

- Ambil F dari visitStack, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex F yang masih berwarna WHITE. Karena tetangga dari E dan G tidak berwarna WHITE maka tidak ada yang dimasukkan ke visitStack.



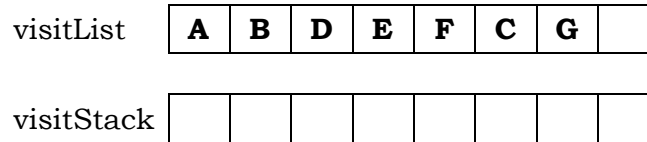
Gambar 10. 37 Langkah 6 Algoritma Traversal Graph: DFS

- Ambil C dari visitStack, warnai dengan BLACK dan masukkan dalam visitList. Cari tetangga dari Vertex C yang masih berwarna WHITE. Karena tetangga dari C adalah G tidak berwarna WHITE maka tidak ada yang dimasukkan ke visitStack.



Gambar 10. 38 Langkah 7 Algoritma Traversal Graph: DFS

- Ambil G dari visitStack, warnai dengan BLACK dan masukkan dalam visitList. Vertex G tidak memiliki tetangga sehingga tidak ada Vertex yang dapat dimasukkan dalam visitStack.



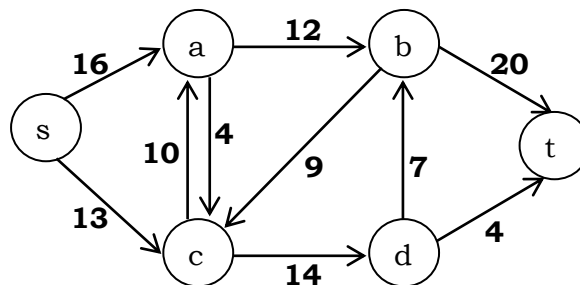
Gambar 10. 39 Langkah 8 Algoritma Traversal Graph: DFS

- Karena Stack kosong maka proses traversal selesai

10.5 Studi Kasus

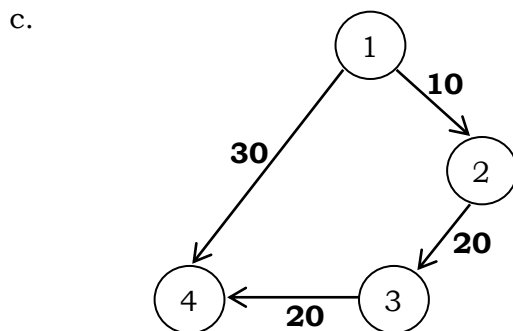
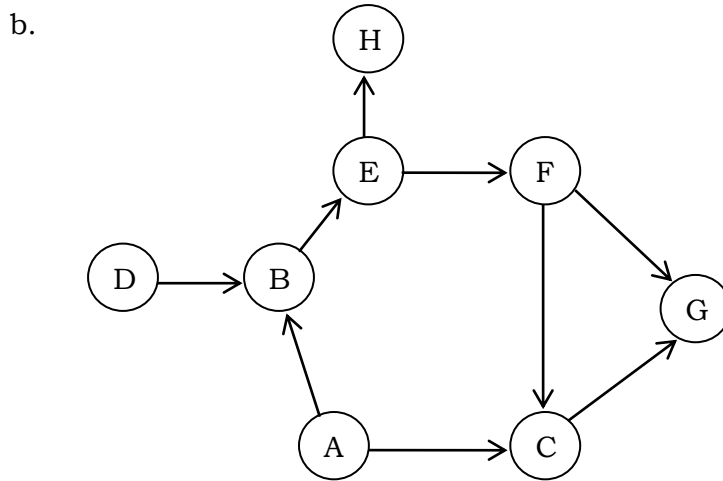
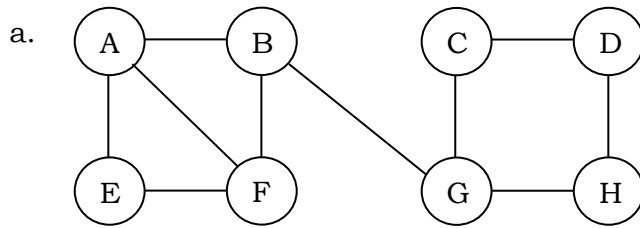
Perintah:

- Tentukan output pembacaan graph
- Ubah garis AC dengan 7
- Hapus link CA, tampilkan output graph
- Tambahkan link CB dengan bobot 2, selanjutnya tampilkan output graph

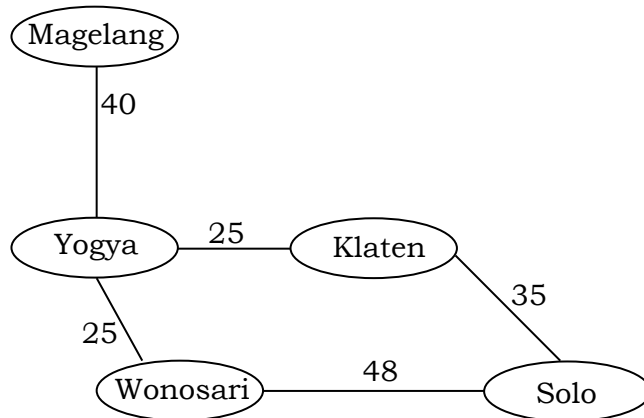


10.6 Latihan

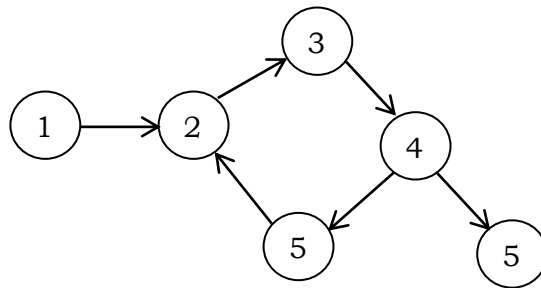
1. Buatlah Adjacency List dalam bentuk matriks untuk graph dibawah ini



2. Buatlah Adjacency List dalam Matrix untuk graph di bawah ini dan tentukan rute paling efektif dari Kota Magelang ke Solo!



3. Buatlah representasi Adjacency List untuk graph dalam bentuk Linked List untuk graph berarah berikut ini



4. Buatlah review mengenai Algoritma BFS dan DFS!
5. Berikan 1 contoh graph yang diselesaikan dengan algoritma traversal BFS dan DFS!

11

Tree

Sub CPMK

Mahasiswa mampu mengevaluasi, merancang dan mengimplementasikan struktur data nonlinear tree kedalam program

Dalam bab ini akan dibahas tentang struktur data Tree. Pembahasan diawali dengan pengertian kemudian akan dibahas representasi tree, penerapan tree pada array dan operasi pada tree. Pada akhir materi akan diberikan Studi Kasus beserta Latihan.

11.1 Pengertian

Pohon (*tree*) adalah salah satu bentuk *graph* terhubung yang tidak mengandung sirkuit. Karena merupakan *graph* terhubung, maka pada pohon selalu terdapat *path* atau jalur yang menghubungkan setiap dua simpul dalam pohon. Struktur pohon adalah suatu cara merepresentasikan suatu struktur hirarki (*one-to-many*) secara grafis yang mirip sebuah pohon, walaupun pohon tersebut hanya tampak sebagai kumpulan node-node dari atas ke bawah. Suatu struktur data yang tidak linier yang menggambarkan hubungan yang hirarkis (*one-to-many*) dan tidak linier antara elemen-elemennya.

Terdapat beberapa perbedaan Tree dan Graph dijelaskan pada Tabel 11.1 berikut ini:

Tabel 11. 1 Perbedaan Graph dan Tree

Graph	Tree
Tidak mempunyai root, proses traversal dapat dilakukan di sembarang vertex	Mempunyai root, proses traversal dilakukan dengan tiga cara
Graph memungkinkan memiliki cycle yang menghasilkan multiple visit ke sebuah vertex	Terdapat path unik dari root menuju ke vertex tertentu.

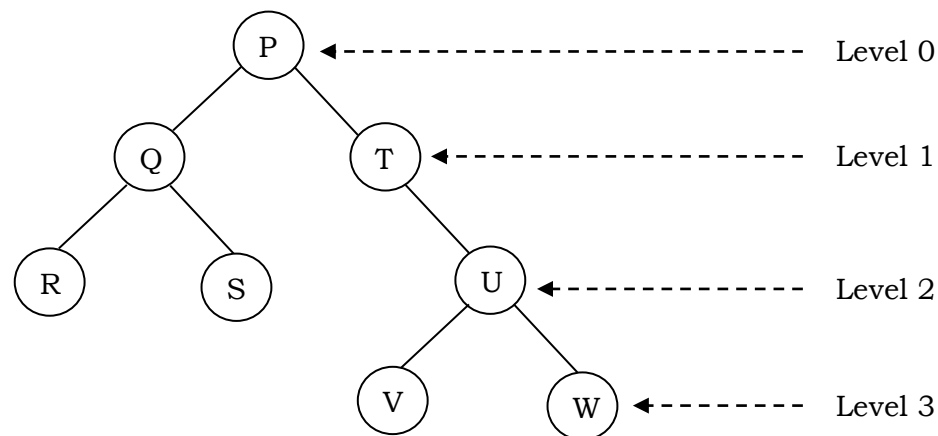
Ada dua macam jenis tree, yaitu :

- **Tree Statik:** isi node-nodenya tetap karena bentuk pohonnya sudah ditentukan.
- **Tree Dinamik:** isi nodenya berubah-ubah karena proses penambahan (insert) dan penghapusan (delete)

Suatu bentuk pohon dilengkapi dengan apa yang disebut “akar” atau “*root*”. Pohon yang salah satu simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi arah sehingga menjadi graf berarah dinamakan pohon berakar (*rooted tree*). Node root dalam sebuah tree adalah suatu node yang memiliki hiarki tertinggi dan dapat juga memiliki node-node anak. Semua node dapat ditelusuri dari node root tersebut. Node root adalah node khusus yang tercipta pertama kalinya. Node-node lain di bawah node root saling terhubung satu sama lain dan disebut subtree. Berikut ini merupakan contoh penggunaan struktur pohon :

- Silsilah keluarga
- Parse Tree (pada compiler)
- Struktur File
- Struktur organisasi
- Hasil pertandingan berbentuk turnamen

Contoh di Gambar 11.1 yang disebut pohon berakar P.



Gambar 11. 1 Pohon berakar

Sifat utama sebuah pohon berakar adalah :

1. Jika pohon mempunyai simpul sebanyak n , maka banyaknya ruas atau *edge* adalah $(n-1)$. Pada pohon P di Gambar 11.1, banyak simpul adalah $n = 8$, dan banyak *edge* $(n - 1) = 8 - 1 = 7$
2. Mempunyai simpul khusus yang disebut “*root*,” yang merupakan simpul yang memiliki derajat keluar ≥ 0 , dan derajat masuk = 0. Simpul P merupakan *root* pada pohon di Gambar 11.1 di atas.
3. Mempunyai simpul yang disebut sebagai “daun” atau “*leaf*,” yang merupakan simpul berderajat keluar 0, dan berderajat masuk = 1. Simpul-simpul R, S, V, W merupakan daun pada pohon di Gambar 11.1.
4. Setiap simpul mempunyai tingkatan atau *level*, yang dimulai dari *root* sebagai *level 0*, sampai dengan *level n* pada daun paling bawah.
Pada pohon P di Gambar 11.1:
P berlevel 0
Q dan T berlevel 1
R, S dan U berlevel 2
V dan W berlevel 3
“Simpul yang mempunyai *level* sama disebut “bersaudara” atau “brother” atau “*siblings*”.

5. Pohon mempunyai ketinggian atau kedalaman atau “*height*,” yang merupakan level tertinggi + 1. Pohon di Gambar 11.1 mempunyai ketinggian atau kedalaman $3+1 = 4$.
6. Pohon mempunyai berat atau bobot atau “*weight*,” yang merupakan banyaknya daun pada pohon. Pohon di Gambar 11.1 mempunyai bobot = 4.

Selanjutnya, lebih khusus lagi dibahas tentang pohon berakar yang disebut “pohon biner” atau “*binary tree*”.

11.2 Pohon Biner (Binary Tree)

Dalam struktur data, pohon memegang peranan yang cukup penting. Struktur ini biasanya digunakan terutama untuk menyajikan data yang mengandung hubungan hirarkikal antara elemen-elemen mereka. Kita lihat misalnya, data pada *record*, keluarga dari pohon, ataupun isi dari tabel. Mereka mempunyai hubungan hirarkikal.

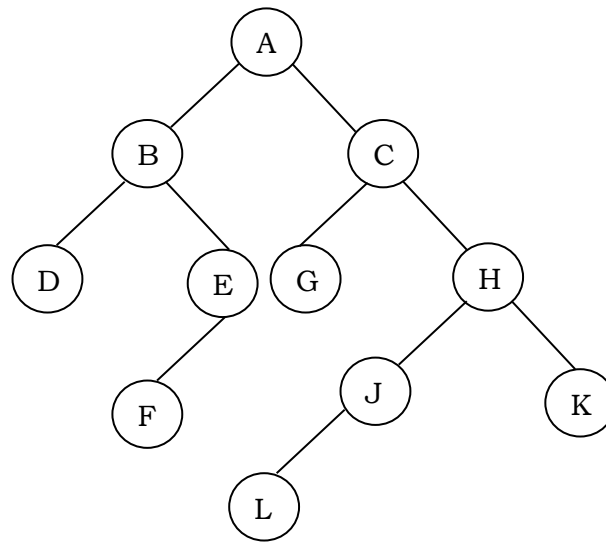
Bentuk pohon berakar yang khusus, yang lebih mudah kita kelola dalam komputer adalah pohon biner (*binary tree*). Sebuah *binary tree* adalah sebuah pengorganisasian secara hirarki dari beberapa buah simpul, dimana masing-masing simpul tidak mempunyai anak lebih dari 2. Bentuk pohon berakar yang umum, kita kenal sebagai “pohon umum” atau “*general tree*.”

Binary Tree adalah struktur data yang hampir mirip juga dengan Linked List untuk menyimpan koleksi dari data. *Linked List* dapat dianalogikan sebagai rantai linier sedangkan *Binary Tree* bisa digambarkan sebagai rantai tidak linier. *Binary Tree* dikelompokkan menjadi *unordered Binary Tree* (pohon yang tidak berurut) dan *ordered Binary Tree* (pohon yang terurut).

Sebuah pohon biner T didefinisikan terdiri atas sebuah himpunan hingga elemen yang disebut simpul (*node*), sedemikian sehingga:

- (a) T adalah hampa (disebut pohon *null*) atau;
- (b) T mengandung simpul R yang dipilih (dibedakan dari yang lain), disebut “akar” atau “*root*” dari T , dan simpul sisanya membentuk 2 pohon biner (subpohon kiri dan subpohon kanan dari akar R) T_1 dan T_2 yang saling lepas.

Perhatikan bahwa pendefinisian pohon biner di atas adalah rekursif. Jika T_1 tidak hampa, maka simpul akarnya disebut suksesor kiri dari R . Hal serupa untuk akar dari T_2 (tidak hampa) disebut suksesor kanan dari R .



Gambar 11. 2 Pohon biner dengan 11 simpul (node)

Pohon biner seringkali disajikan dalam bentuk diagram. Perhatikan contoh pohon biner pada Gambar 11.2. Pohon biner tersebut mempunyai 11 simpul yang diberi label huruf A sampai L (tak termasuk I).

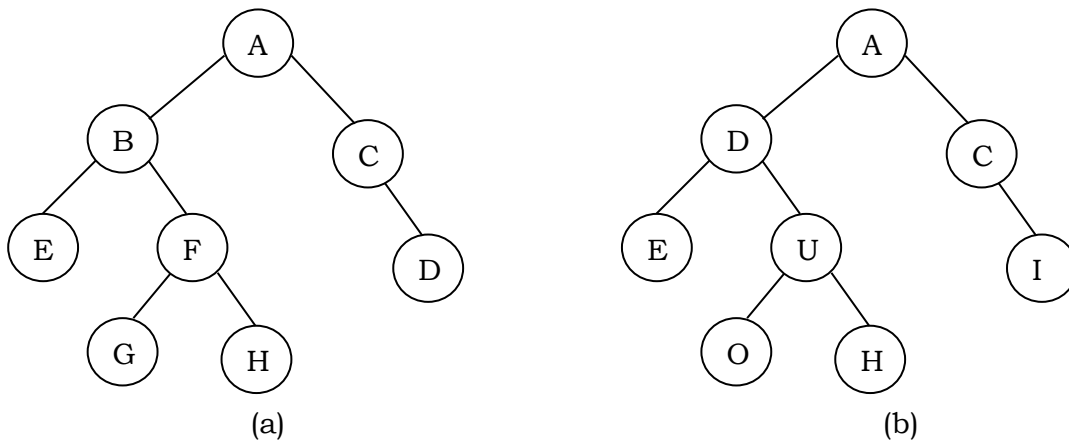
Simpul akar adalah simpul yang digambar pada bagian paling atas. Untuk menggambarkan suksesor kiri serta suksesor kanan, dibuat garis ke kiri bawah dan ke kanan bawah. Perhatikan pada Gambar tersebut bahwa B adalah suksesor kiri dari A, sedangkan C adalah suksesor kanan dari A. Subpohon kiri dari A mengandung simpul-simpul B, D, E dan F, sedangkan subpohon kanannya mengandung simpul-simpul C, G, H, J, K dan L.

Kita dapat melihat bahwa jika N adalah sebarang simpul dari pohon biner T, maka N mempunyai 0, 1 atau 2 buah suksesor. Simpul yang berada dibawah sebuah simpul (suksesor) dinamakan anak (*child*). Jadi simpul yang berada diatas sebuah simpul (predesesor) dinamakan, contohnya adalah simpul N tersebut boleh kita sebut orang-tua (*parent*) dari suksesornya.

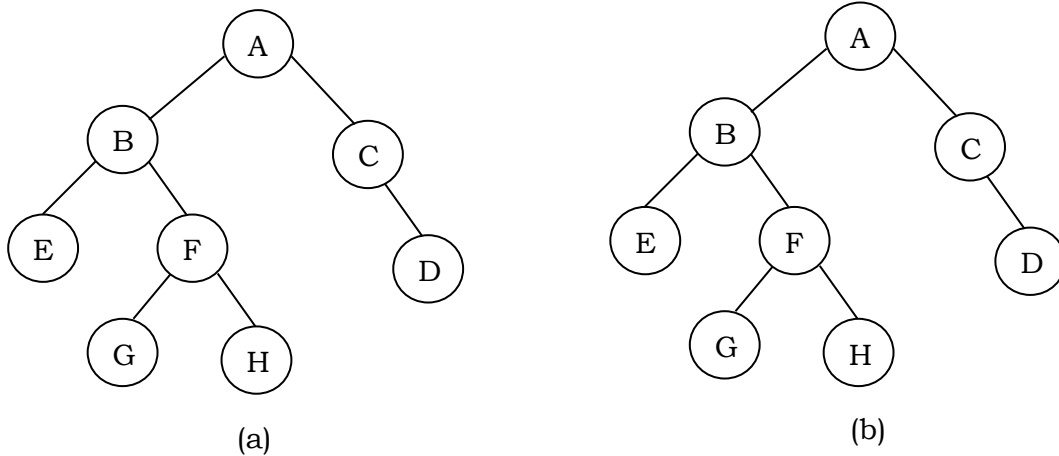
Pada contoh kita, Simpul A, B, C dan H mempunyai 2 anak, simpul E dan J mempunyai satu anak. Sementara itu simpul-simpul D, F, G, L dan K tidak mempunyai satu anakpun. Simpul yang tidak mempunyai anak disebut “daun” atau “*leaf*”

Sekali lagi perhatikan bahwa definisi pohon biner di atas adalah rekursif. T didefinisikan berdasarkan subpohon biner T1 dan T2. Ini berarti bahwa setiap simpul N dari pohon mengandung subpohon kiri dan kanan. Jika simpul N adalah daun maka kedua subpohon kiri dan kanannya adalah hampa.

Dua pohon biner T dan U disebut “*similar*” jika mereka mempunyai bangun (susunan) yang sama. Dua pohon biner pada Gambar 11.3 dibawah ini adalah similar, sementara itu, pohon pada Gambar 11.4 menggambarkan dua pohon yang tidak saja *similar* tetapi juga sama persis antara satu dengan lainnya (baik susunan atau struktur pohon, maupun isi dari setiap simpulnya) yang disebut dengan salinan (*copy / copies*).



Gambar 11. 3 Dua pohon biner yang similar



Gambar 11. 4 Dua pohon binar yang disebut copies

Kelebihan struktur *Binary Tree*:

1. Mudah dalam penyusunan algoritma sorting
2. Searching data relative cepat
3. Fleksibel dalam penambahan dan penghapusan data

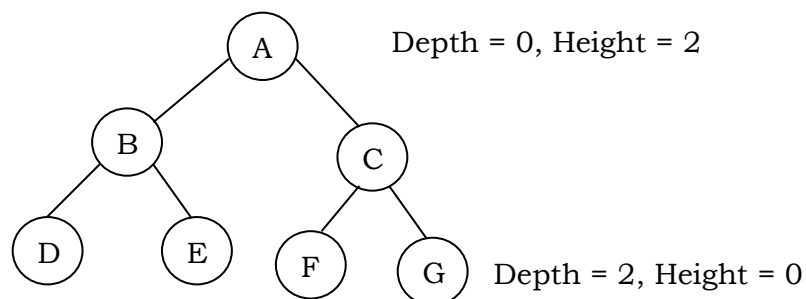
11.2.1 Terminologi Pohon Berakar

Terminologi dari pohon berakar yang dapat dilihat dari Gambar 11.2 antarlain:

- **Root**
 - Node khusus yang tidak memiliki predecessor.
- **Predecessor**
 - Node yang berada diatas node tertentu.
- **Successor**
 - Node yang berada dibawah node tertentu.
- **Ancestor**
 - Seluruh node yang terletak sebelum (berada di atas) antara node tertentu dengan *root* dan terletak pada jalur yang sama.
- **Descendant**
 - Seluruh node yang terletak setelah (berada di bawah) node tertentu dan terletak pada jalur yang sama
- **Orang tua (Parent)**
 - Predecessor satu level diatas suatu node Node yang berada diatas node tertentu
 - a adalah orangtua dari node b dan c
- **Anak (child atau children)**
 - Successor satu level di bawah suatu node
 - b dan c adalah anak-anak simpul a,
- **Sibling**
 - Node-node yang memiliki parent yang sama
- **Subtree**
 - Suatu node beserta descendantnya.
- **Lintasan (path)**
 - Lintasan dari a ke f adalah a, b, e, f.
 - Panjang lintasan = jumlah sisi.
 - Panjang lintasan dari a ke f adalah 3.
- **Saudara kandung (sibling)**
 - Anak lain dari *parent* yang sama dengan simpul.
 - d adalah saudara kandung e, tetapi g bukan saudara kandung e, karena orangtua mereka berbeda.

- **Daun (leaf)**
 - Simpul yang berderajat nol (atau tidak mempunyai anak) disebut daun. Simpul d,f,g,l dan k adalah daun.
- **Simpul Dalam (internal nodes)**
 - Simpul selain akar yang mempunyai anak disebut simpul dalam.
 - Simpul b, e, c, h dan j adalah simpul dalam.
- **Aras (level) atau Tingkat**
 - Terdapat 5 level yaitu level 0 – level 4.
- **Size**
 - Banyaknya node dalam suatu tree.
- **Tinggi (height)**
 - Banyaknya tingkatan dalam suatu node.
 - Disebut juga aras maksimum dari suatu pohon.
 - Height node N = Panjang lintasan terpanjang dari N ke daun. Pohon di atas mempunyai tinggi 4.
- **Kedalaman (depth)**
 - Kedalaman pohon tersebut dari akar disebut kedalaman (*depth*).
 - Depth dari node N = Panjang path dari root ke N
- **Degree**
 - Banyaknya child dalam suatu node.

Berikut ini Gambar 11.5 sebagai contoh terminologi dari pohon berakar.



Gambar 11. 5 Terminologi Pohon Berakar

Ancestor (F)	= C,A	Height	= 3
Descendant (C)	= F,G	Root	= A
Pareng (D)	= B	Leaf	= D,E,F,G
Child (A)	= B,C	Degree (C)	= 2

Sibling (F) = G
Size = 7

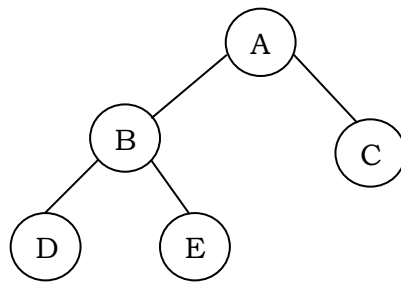
11.2.2 Jenis-jenis Binary Tree

1) Full Binary Tree

Semua node kecuali leaf pasti memiliki 2 anak dan tiap subtree memiliki panjang path yang sama. Contoh seperti pada Gambar 11.5.

2) Complete Binary Tree

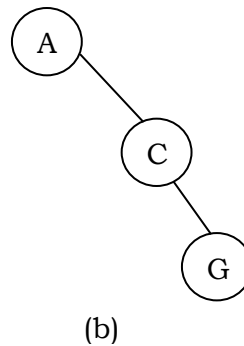
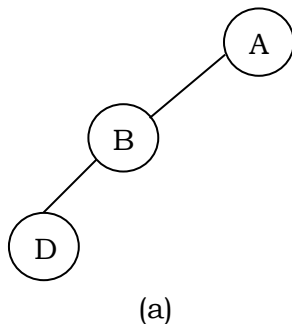
Mirip dengan full binary tree, namun tiap subtree boleh memiliki Panjang path yang berbeda dan tiap node (kecuali leaf) memiliki 2 anak. Complete binary tree digambarkan seperti Gambar 11.6 berikut



Gambar 11. 6 Complete Binary Tree

3) Skewed Binary Tree

Merupakan *binary tree* yang semua nodenya (kecuali leaf) hanya memiliki satu anak.

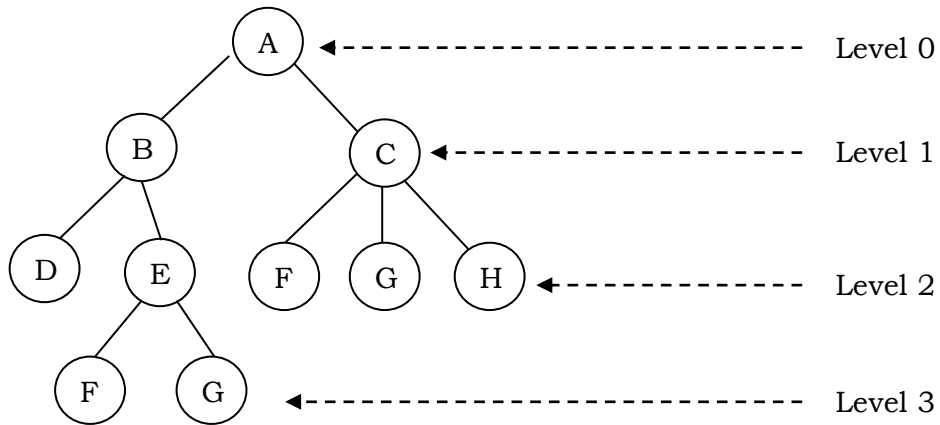


Gambar 11. 7 Pohon condong kanan (a) dan condong kiri (b)

11.3 Representasi Tree

Kita dapat menyajikan suatu pohon biner T dalam beberapa cara, antara lain:

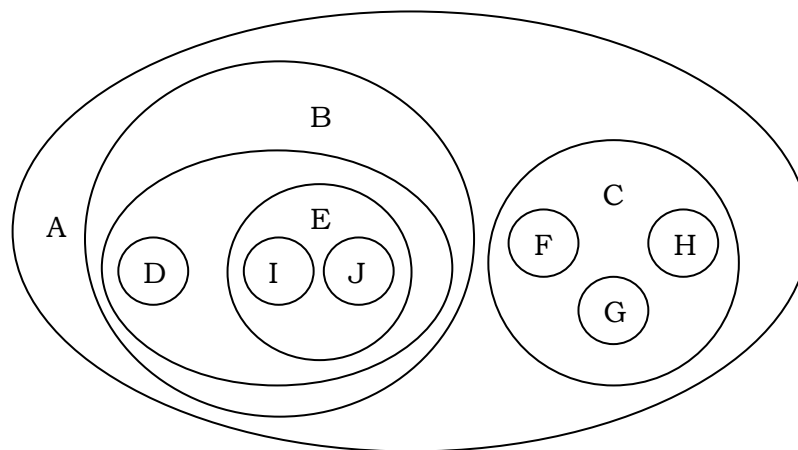
1. Sebuah pohon beserta tingkatannya (*level*)
Digambarkan oleh Gambar 11.5 berikut ini



Gambar 11. 8 Sebuah pohon beserta tingkatannya (level)

2. Diagram Venn

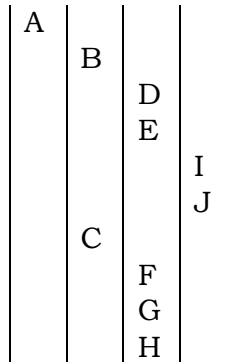
Dari Gambar 11.5 dapat pula direpresentasikan dengan diagram venn seperti digambarkan oleh Gambar 11.6 berikut ini:



Gambar 11. 9 Diagram Venn

3. Notasi Tingkat

Representasi lainnya dari Gambar 11.5 dapat direpresentasikan dengan Notasi Tingkat seperti digambarkan oleh Gambar 11.7 berikut ini:



Gambar 11. 10 Notasi Tingkat

4. Notasi Kurung

Jika Gambar 11.8 akan dituliskan dalam representasi notasi kurung maka ditulis:

$(A(B(D,E(I,J)),C(F,G,H)))$

Sekarang kita akan melihat dari sisi lainnya lagi. Representasi tree di dalam memori dapat disajikan dengan dua cara. Cara pertama adalah penyajian kait (link). Cara ini biasa digunakan dengan list berkaitan (linked list) ketika disajikan dalam memori, yaitu dengan *doubly linked list non circular*. Cara kedua adalah dengan menggunakan sebuah array tunggal disebut penyajian sekuensial dari T.

Kebutuhan utama yang harus dipenuhi pada setiap penyajian dari T adalah bahwa seseorang dapat mempunyai akses langsung ke akar R dan T, dan bila diberikan sembarang simpul N, seseorang harus dapat akses langsung ke anak dari N.

Suatu pohon biner T akan disimpan dalam memori secara penyajian kait atau linked list. Penyajian ini menggunakan tiga array sejajar INFO, LEFT, dan RIGHT, serta sebuah variabel penuding ROOT. Masing-masing simpul N dari pohon T berkorespondensi dengan suatu lokasi K, sedemikian sehingga:

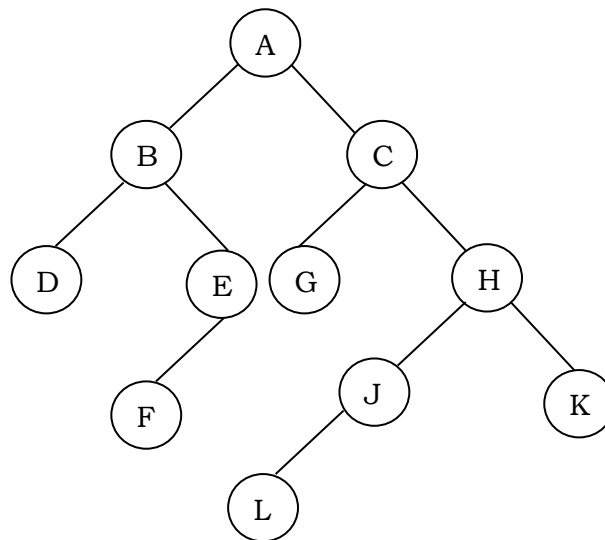
- (1) INFO[K] berisi data pada simpul N
- (2) LEFT[K] berisi lokasi dari anak kiri simpul N
- (3) RIGHT[K] berisi lokasi dari anak kanan simpul N.

ROOT akan berisi lokasi dari akar R dari pohon T. Jika suatu subpohon kosong, maka pointer yang bersangkutan akan berisi harga nol. Jika suatu pohon T sendiri kosong, maka ROOT akan berisi nilai nol.

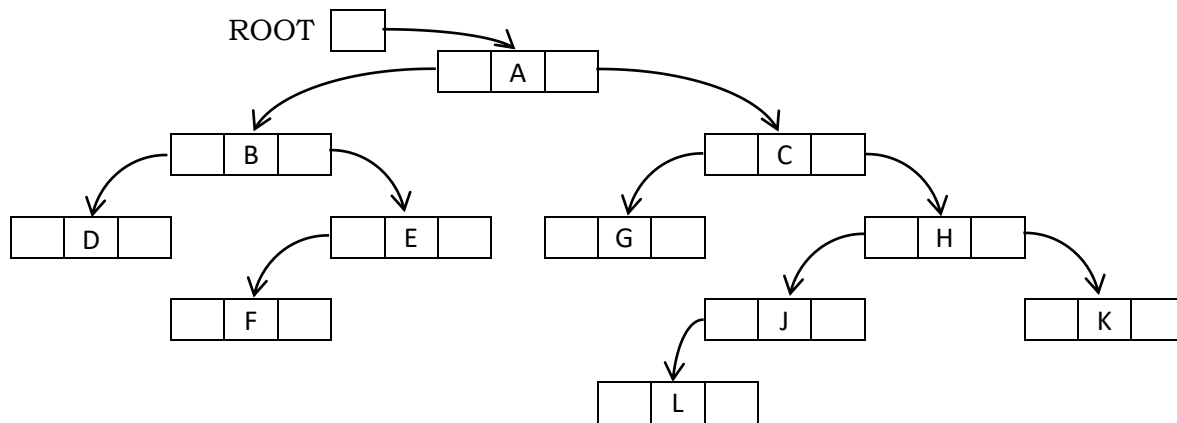
Dapat dicatat bahwa simpul dari pohon bisa saja berisi lebih dari satu informasi. Pada prakteknya simpul biasanya berisi sebuah record. Jadi INFO sebenarnya berupa linear array dari record ataupun berupa sebuah koleksi array sejajar. Selain itu, karena suatu simpul boleh diselipkan sebagai simpul pohon baru, atau simpul lama boleh dihapus dari pohon, kita juga secara implisit mengasumsikan bahwa lokasi hampa dalam array INFO, LEFT dan RIGHT membentuk sebuah list berkaitan dengan pointer AVAIL. Kita biasanya memisalkan array LEFT berisi pointer untuk list AVAIL.

Untuk menunjuk ke alamat yang invalid (subpohon hampa), dipilih penuding nol yang dinyatakan sebagai NULL. Kenyataannya dalam praktek, kita gunakan 0 atau bilangan negatif sebagai isi dari NULL.

Sebagai contoh, Gambar 11.9 menggambarkan skema dari penyajian kait pohon biner Gambar 11.10. Terlihat bahwa setiap simpul digambar terdiri atas tiga field. Terlihat juga di sini, subpohon hampa digambar berlabel x untuk mengisi penuding nol. Selanjutnya Gambar 11.11 menunjukkan bagaimana penyajian kait pohon biner yang bersangkutan. Sebagai contoh, misalkan diketahui berkas personalia suatu perusahaan kecil yang berisi data 9 pegawainya dengan fields: NAME, SOCIAL-SECURITY-NUMBER (SSN), SEX, serta SALARY. Berkas tersebut disimpan dalam memori sebagai pohon biner, seperti terlihat pada Gambar 11.12



Gambar 11. 11 Representasi pohon biner

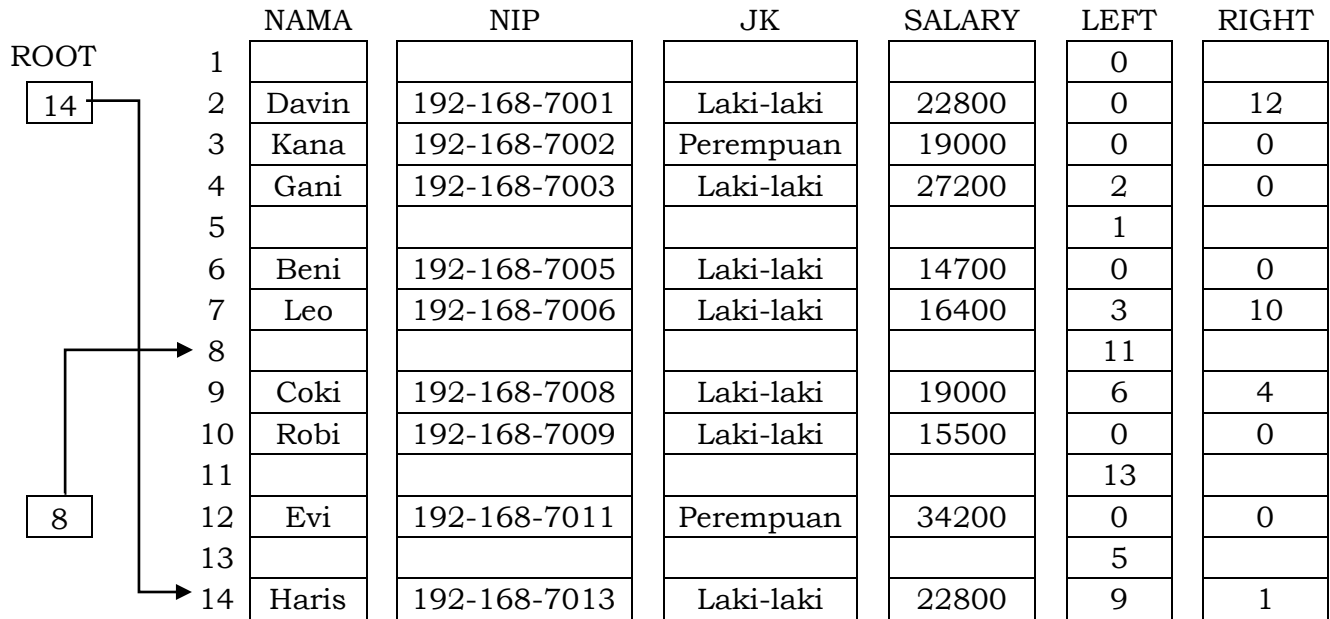


Gambar 11. 12 Penggambaran skema penyajian kait pada pohon biner

	INFO	LEFT	RIGHT
1	K	0	0
2	G	3	6
3	C	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	

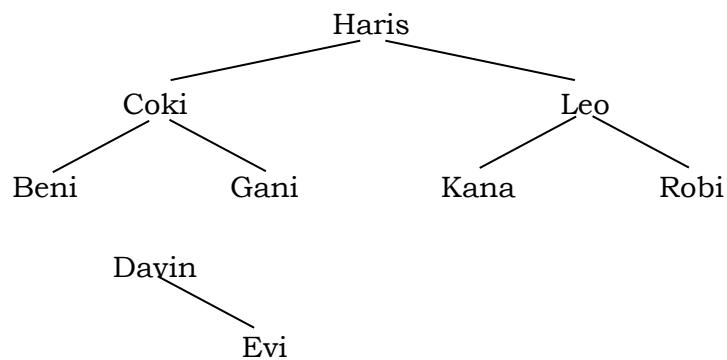
ROOT [5] → 5
 AVAIL [8] → 8

Gambar 11. 13 Skema memori penyajian kait dari pohon biner



Gambar 11. 14 Contoh gambaran pohon biner pada records

Diagram pohon biner dari Gambar 11.8 dapat dilihat di Gambar 11.9 berikut ini :



Gambar 11. 15 Skema dari Gambar 11.8

Untuk memudahkan, kita menulis label dari simpul hanya berupa field kunci, NAMA.

Kita membentuk pohon pada Gambar 11.12 tersebut sebagai berikut :

- (1) Harga dari ROOT = 14 menunjukkan bahwa record nomor 14 dengan NAMA = "Haris" adalah akar dari Pohon.

- (2) $LEFT[14] = 9$ menunjukkan bahwa Coki (record nomor 9) adalah anak kiri dari Harris, dan $RIGHT[14] = 7$ menunjukkan bahwa Leo adalah anak kanan dari Harris.

Dengan mengulangi langkah (2) kita peroleh diagram pohon biner seperti Gambar 11.15 Ingat bila $LEFT[N]$ atau $RIGHT[N]$ bernilai 0 menandakan bahwa simpul N tidak mempunyai anak kiri /kanan.

11.3.1 Implementasi program tree

Menggunakan struct

```
Typedef struct Node {
    Int data;
    Node *kiri;
    Node *kanan;
};
```

Deklarasi variabel

```
Node *pohon;
```

Opptasi

Create: yaitu operasi yang membentuk tree baru yang kosong.

```
pohon = NULL;
```

Insert : yaitu operasi untuk menambah node ke dalam Tree

Jika data yang akan dimasukkan lebih besar daripada elemen root maka data akan diletakkan pada node sebelah kanan. Sebaliknya, jika data yang akan dimasukkan nilainya lebih kecil daripada elemen root maka data akan diletakkan pada node sebelah kiri. Untuk data pertama akan menjadi elemen root.

```
void tambah(Node **root, int databaru)
{
    If((*root) == NULL) {
        Node *baru;
        baru = new Node;
        baru->data = databaru;
        baru->kiri = NULL;
        baru->kanan = NULL;
        (*root) = baru;
        (*root)->kiri = NULL;
        (*root)->kanan = NULL;
        printf("Data bertambah")
    }
    else if(databaru < (*root)->data)
        tambah(&(*root)->kiri, databaru);
```

```

else if(databaru > (*root)->data)
    tambah(&(*root)->kanan,databaru);
else if(databaru == (*root)->data)
    printf"Data sudah ada!";

}

```

11.4 Traversal

Proses traversal adalah proses kunjungan terhadap node-node di dalam suatu pohon dimana masing-masing node akan dikunjungi satu kali. Dari daftar kunjungan secara lengkap tersebut kita dapat mengetahui urutan informasi secara linier yang tersimpan di dalam sebuah tree. Terdapat tiga teknik rekursif yang berguna untuk melakukan traversal, di mana kita dapat secara sistematis mengurutkan semua simpul pohon [4]. Ketiga teknik traversal atau kunjungan tersebut yaitu:

- **PreOrder** (*depth first order*)
cetak node yang dikunjungi, kunjungi left, kunjungi right
Algoritma:
 - Jika tree kosong, maka keluar
 - kunjungi R (node *root*)
 - kunjungi T1 (subtree kiri) secara preorder
 - kunjungi T2 (subtree kanan) secara preorder

Implementasi:

```

void preOrder(Node *root){
    If(root != NULL){
        printf("%d ",root->data);
        preOrder(root->kiri);
        preOrder(root->kanan);
    }
}

```

- **InOrder** (*symmetric order*)
kunjungi left, cetak node yang dikunjungi, kunjungi right
Algoritma:
 - Jika tree kosong, maka keluar
 - kunjungi T1 (subtree kiri) secara inorder
 - kunjungi R (node *root*)
 - kunjungi T2 (subtree kanan) secara inorder

Implementasi:

```

void inOrder(Node *root){
    If(root != NULL){
        inOrder(root->kiri);

```

```

        printf("%d ", root->data);
        inOrder(root->kanan);
    }
}

```

- **PostOrder**

kunjungi left, kunjungi right, cetak node yang dikunjungi

Algoritma:

- Jika tree kosong, maka keluar
- kunjungi T1 (subtree kiri) secara postorder
- kunjungi T2 (subtree kanan) secara postorder
- kunjungi R (node *root*)

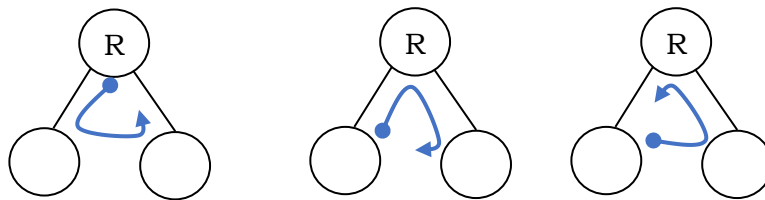
Implementasi:

```

void postOrder(Node *root){
    If (root != NULL) {
        postOrder(root->kiri);
        postOrder(root->kanan);
        printf("%d ", root->data);
    }
}

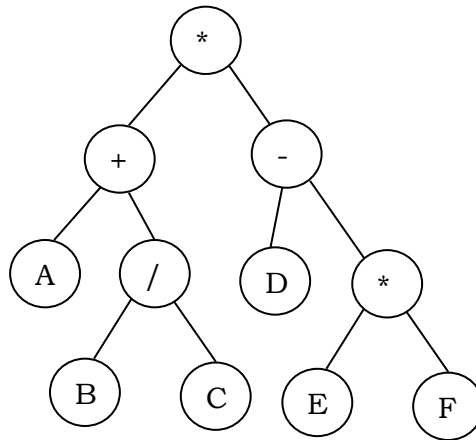
```

Semua algoritma traversal (*preorder*, *inorder*, *postorder*) yang diberikan di atas berupa algoritma rekursif, dan sebenarnya dapat dikerjakan secara iteratif dengan bantuan stack. Ketiga proses tersebut dapat diilustrasikan oleh Gambar 11.16.



Gambar 11. 16 Proses Traversal

Selanjutnya akan diberi contoh ekspresi matematika $((A + (B / C)) * (D - (E * F)))$, apabila digambarkan dalam bentuk binary tree. ilustrasi kunjungan dalam sebuah kasus operasi aritmatika, akan ditunjukkan Gambar 11.17



Gambar 11. 17 Contoh Ilustrasi traversal pada ekspresi matematika

Apabila binary tree di atas dikunjungi secara:

- *Preorder (prefix)* : * + a / b c - d * e f
- *Inorder (infix)* : a + b / c * d - e * f
- *Postorder (postfix)* : a b c / + d e f * - *

11.5 Operasi-operasi Tree

Ada beberapa operasi yang bisa dilakukan pada struktur data tree yaitu:

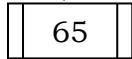
- **Insert:**
menambah node ke dalam Tree secara rekursif. Jika data yang akan dimasukkan lebih besar daripada elemen root, maka akan diletakkan di node sebelah kanan, sebaliknya jika lebih kecil maka akan diletakkan di node sebelah kiri. Untuk data pertama akan menjadi elemen root.
- **Find:**
mencari node di dalam Tree secara rekursif sampai node tersebut ditemukan dengan menggunakan variable bantuan ketemu. Syaratnya adalah tree tidak boleh kosong.
- **Traverse:**
yaitu operasi kunjungan terhadap node-node dalam pohon dimana masing-masing node akan dikunjungi sekali.
- **Count:**
menghitung jumlah node dalam Tree.
- **Height:**
mengetahui kedalaman sebuah Tree.
- **Find Min dan Find Max:**
mencari nilai terkecil dan terbesar pada Tree.

- **Child:**
mengetahui anak dari sebuah node (jika punya).

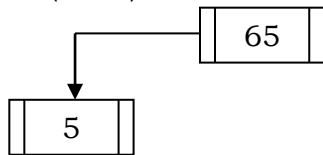
11.5.1 Insert Node

Supaya lebih mudah memahami prosesnya akan digambarkan ilustrasi operasi insert sebagai berikut:

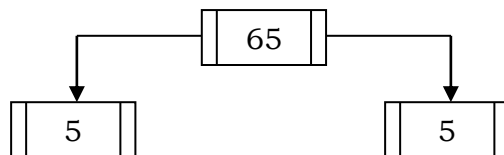
1. Insert (root, 65)



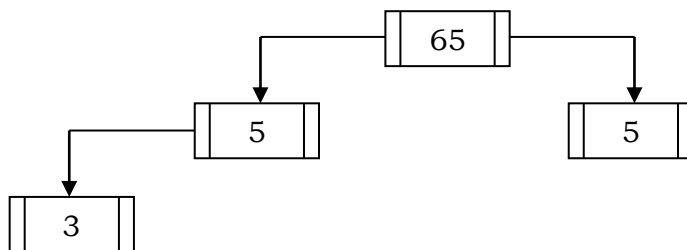
2. Insert (left,5)



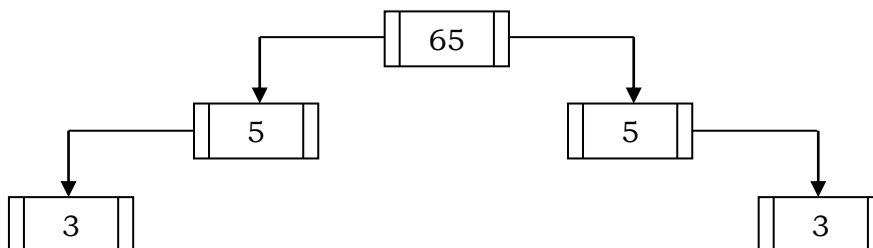
3. Insert (right, 70)



4. Insert (left,3)



5. Insert (right, 10)



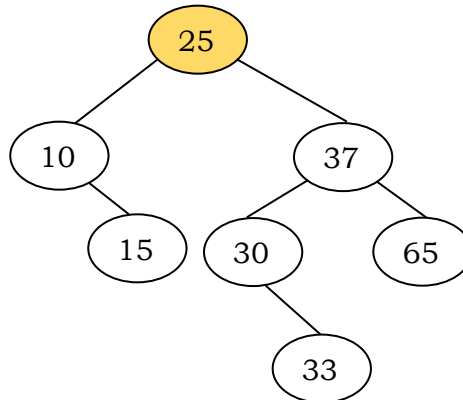
Gambar 10. 40 Ilustrasi proses insert

11.5.2 Menghapus Node

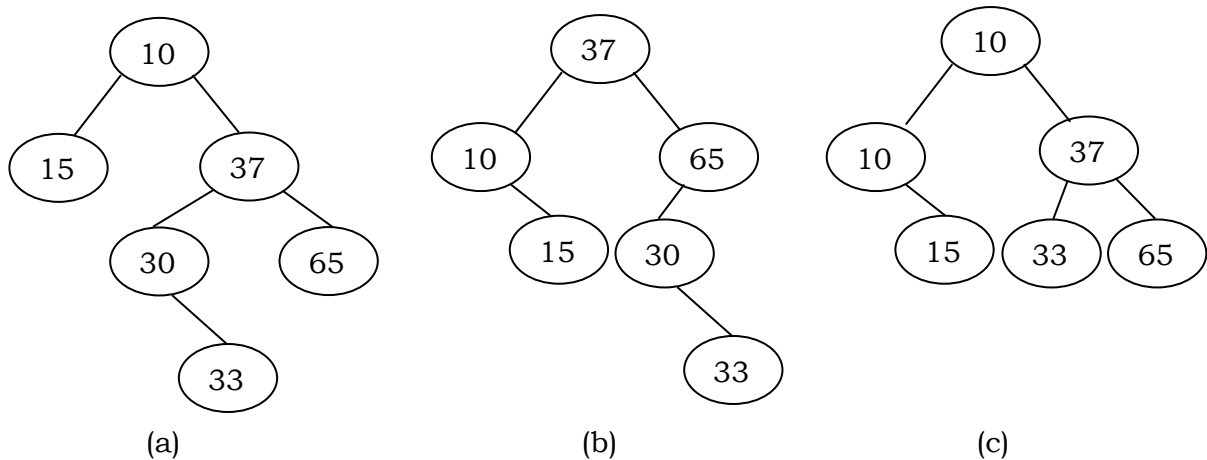
Terdapat aturan dalam menghapus node pada *Binary Search Tree*. Sehingga proses penghapusan dapat dibagi menjadi beberapa kasus, antara lain:

- **Root pada BST tidak dapat dihapus,**

karena terdapat dua subtree pada node root. Dapat diganti dengan node yang tepat yang terdapat pada BST.



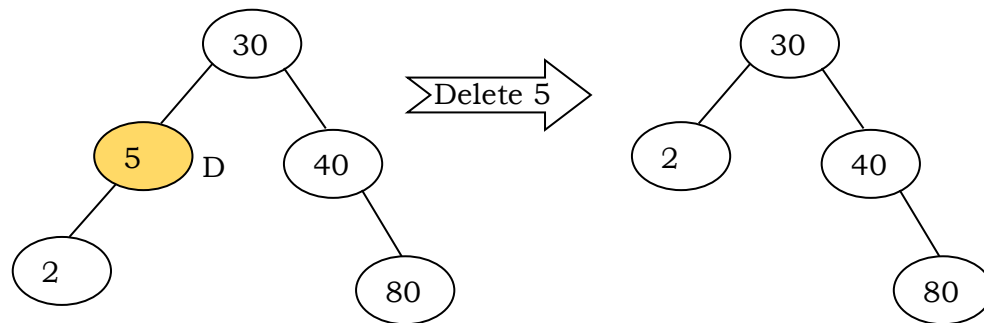
Gambar 10. 41 Menghapus Node Root 25



Gambar 10. 42 Mencari pengganti Node Root dengan nilai yang tepat

Sebagai contoh pada Gambar 10.41, node root 25 akan menghapus. Pada Gambar 11.42(a), jika node 25 diganti dengan node 10, bukan merupakan solusi yang tepat, karena ada node yang tidak sesuai yaitu node 15. Pada Gambar 11.42 (b), jika node 25 diganti dengan node 37, bukan merupakan solusi yang tepat, karena ada node yang tidak sesuai yaitu node 30 dan 33. Pada gambar 11.42 (c), jika node 25 diganti dengan node 30, merupakan solusi yang tepat, karena semua node mengikuti aturan dari BST.

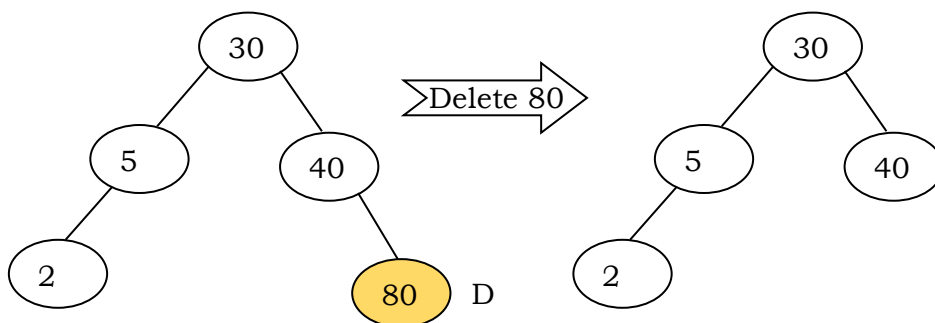
- **Menghapus sebuah node yang memiliki satu anak.**
Operasi Delete (5)



Gambar 10. 43 Menghapus node yang memiliki 1 anak

Pada gambar 10.43 node 5 bukan node leaf, node 5 mempunyai subtree kosong. Node 5 diberi tanda dengan D, Parent dari node 5 yaitu node 30 adalah P dan anak dari node 5 yaitu node 2 ditandai dengan L. Selanjutnya hapus node 5, dan kaitkan L(node 2) sebagai anak dari P(node 30).

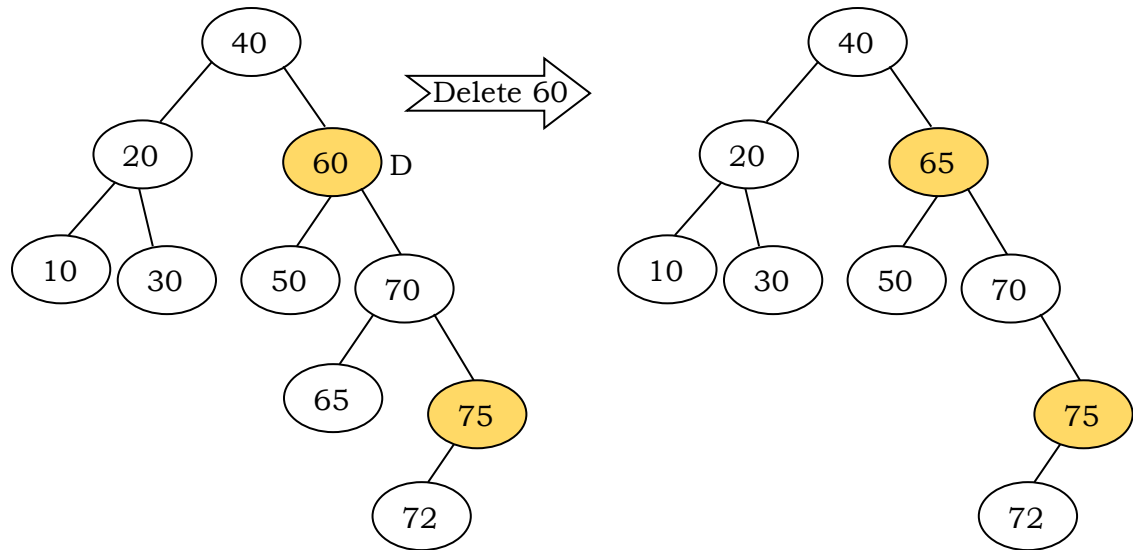
- **Menghapus node leaf (tidak memiliki anak),**
D tidak memiliki anak
Operasi Delete(80)



Gambar 10. 44 Menghapus Node Leaf

menggunakan cara yang sama dengan menghapus node yang bukan leaf memiliki 1 anak. Pada Gambar 10.44, node yang akan dihapus node 80 ditandai dengan D, parent dari node 80 yaitu node 40 ditandai dengan P, dan anak dari node 80 ditandai dengan R. R bernilai null, karena node yang akan dihapus adalah node leaf. Hapus node 80(D), kaitkan R(null) menjadi anak dari node P(40).

- **Menghapus node yang memiliki dua anak.**



Gambar 10. 45 Ilustrasi proses penghapusan node dengan 2 anak

Pilih R (node yang berada di subtree dari node 60) sebagai node dengan nilai terkecil, tapi lebih besar dari nilai node yang dihapus. Pada Gambar 10.45, node 60 sebagai node yang akan dihapus, cari node R yang terkecil, tapi lebih besar dari nilai node yang dihapus (node 60) yaitu node 65. Hapus node 60 dan kaitkan R ke parent dari node yang dihapus.

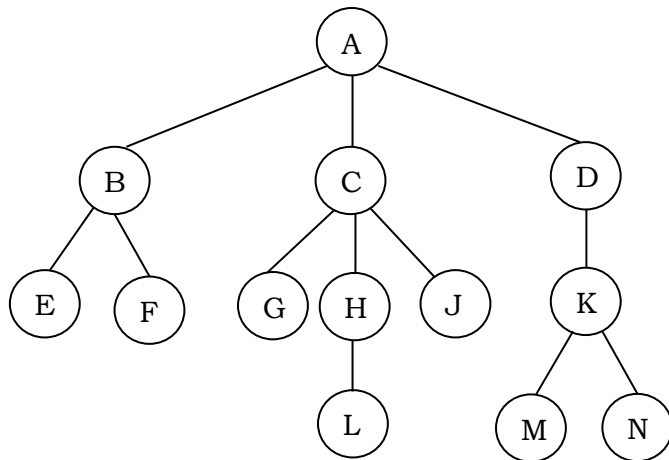
11.6 Binary Search Tree (BST)

Pencarian dilakukan secara rekursif, dimulai dari node root, jika data yang dicari lebih kecil daripada data node root, maka pencarian dilakukan di sub node sebelah kiri, sedangkan jika data yang dicari lebih besar daripada data node root, maka pencarian dilakukan di sub node sebelah kanan, jika data yang dicari sama dengan data suatu node berarti kembalikan node tersebut dan berarti data ditemukan.

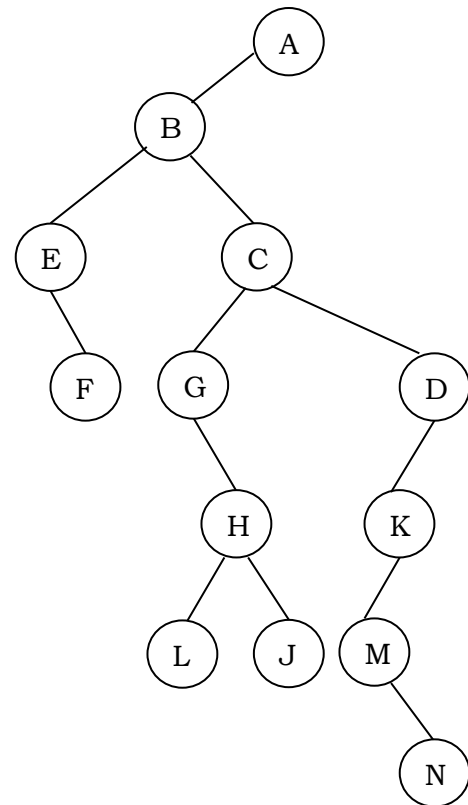
11.6.1 Konversi Tree biasa ke Binary Tree

Anak pertama menjadi anak kiri, anak ke-2 menjadi cucu kanan, ke-3 jadi cicit kanan dan seterusnya.

Pohon umum:



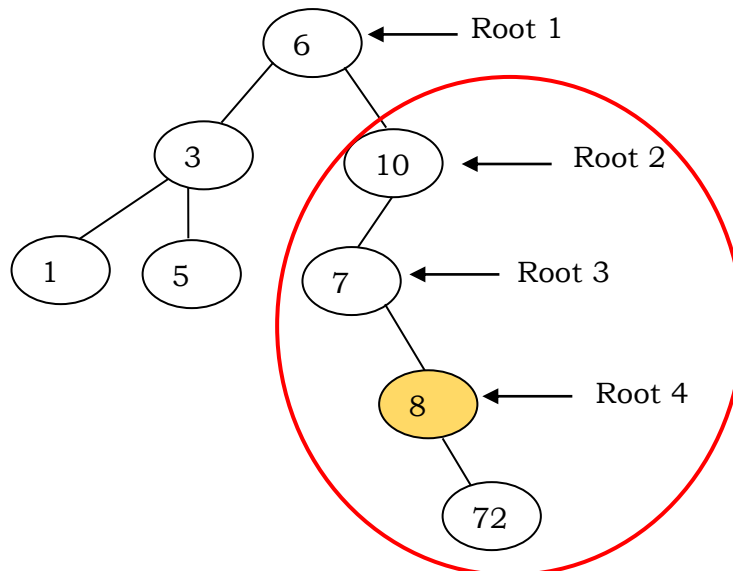
Pohon Biner:



Gambar 11. 18 Konversi menjadi pohon biner

11.6.2 Ilustrasi Searching dengan BST

Misalnya: dicari data **8**



Gambar 11. 19 Ilustrasi Searching BST

Dicari data 8

Mid point = root,

Root = 6, karena $8 > 6$, maka akan dicari di sub node bagian kanan root.

Root = 10, karena $8 < 10$, maka akan dicari di sub node bagian kiri root.

Root = 7, karena $8 > 7$, maka akan dicari di sub node bagian kanan root.

Root = 8, berarti $8 = 8$, maka akan dikembalikan node tersebut dan dianggap **Data Ditemukan!**

Contoh lain dari proses pencarian pohon biner dapat dilihat pada Gambar 11.20, diberikan ilustrasi pertahapan saat melakukan pencarian dengan *Binary Search Tree*:

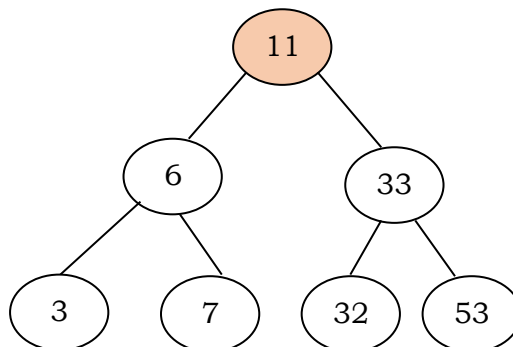
Pencarian Nilai Target = 7

Langkah 1:

Tentukan midpoint:

3	6	7	11	32	33	53
---	---	---	----	----	----	----

Mulai dari root



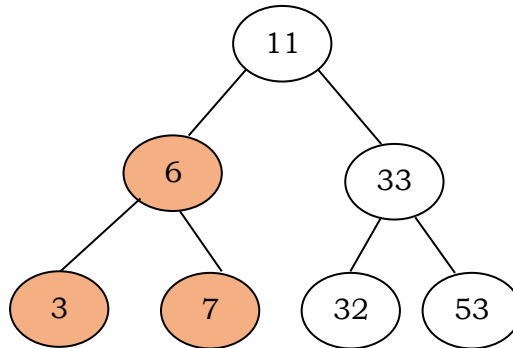
Gambar 10. 46 Langkah 1 proses pencarian BST

Root = 11, karena $7 > 11$, maka akan dicari di sub node bagian sub-array bagian kiri (anak kiri).

Langkah 2:

Cari pada sub-array bagian kiri (anak kiri):

3	6	7	11	32	33	53
---	---	---	----	----	----	----



Gambar 10. 47 Langkah 2 proses pencarian BST

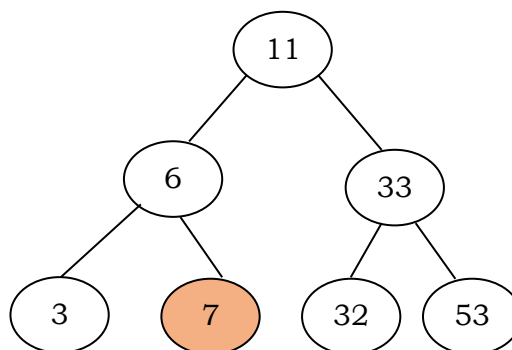
Tentukan midpoint dari sub-array kemudian dijadikan root selanjutnya.

Langkah 3:

Kunjungi root dari sub-tree.

3	6	7	11	32	33	53
---	---	---	----	----	----	----

Root = 6, karena **7** > 6, maka akan dicari di sub node bagian kanan, kemudian dijadikan root selanjutnya.



Gambar 10. 48 Langkah 3 proses pencarian BST

Root = 7, berarti $7 = 7$, maka akan dikembalikan node tersebut dan dianggap **Data Ditemukan!**

11.7 Studi Kasus

Mengimplementasikan proses penambahan data (*insert*) dan menampilkan data secara *preorder*, *inorder* dan *postorder*

```
//header file
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

//pendeklarasian struct sebuah tree awal
struct Node{
    int data;
    Node *kiri;
    Node *kanan;
};

//fungsi untuk menambahkan node baru
void tambah(Node **root, int databaru)
{
    //jika root masih kosong
    if((*root) == NULL)
    {
        //pembuatan node baru
        Node *baru;
        //pengalokasian memori dari node yang telah dibuat
        baru = new Node;
        //inisialisasi awal node yang baru dibuat
        baru->data = databaru;
        baru->kiri = NULL;
        baru->kanan = NULL;
        (*root) = baru;
        (*root)->kiri = NULL;
        (*root)->kanan = NULL;
        printf("Data bertambah!");
    }
    //jika data yang akan dimasukkan lebih kecil daripada elemen
    root, maka akan diletakkan di node sebelah kiri.
    else if(databaru < (*root)->data)
        tambah(&(*root)->kiri, databaru);
    //jika data yang akan dimasukkan lebih besar daripada elemen
    root, maka akan diletakkan di node sebelah kanan
    else if(databaru > (*root)->data)
        tambah(&(*root)->kanan, databaru);
    //jika saat dicek data yang akan dimasukkan memiliki nilai
    yang sama dengan data pada root
    else if(databaru == (*root)->data)
        printf("Data sudah ada!");
}
```

```

}

//fungsi yang digunakan untuk mencetak tree secara preOrder
void preOrder(Node *root)
{
    if(root != NULL){
        printf("%d ", root->data);
        preOrder(root->kiri);
        preOrder(root->kanan);
    }
}

//fungsi yang digunakan untuk mencetak tree secara inOrder
void inOrder(Node *root)
{
    if(root != NULL){
        inOrder(root->kiri);
        printf("%d ", root->data);
        inOrder(root->kanan);
    }
}

//fungsi yang digunakan untuk mencetak tree secara postOrder
void postOrder(Node *root)
{
    if(root != NULL){
        postOrder(root->kiri);
        postOrder(root->kanan);
        printf("%d ", root->data);
    }
}

//fungsi utama
int main()
{
    //deklarasikan variabel
    int pil, data; // c;
    Node *pohon; // *t;
    pohon = NULL; //inisialisasi node pohon
    //perulangan do-while
    do
    {
        system("cls"); //bersihkan layar
        printf("\t#PROGRAM TREE C++#");
        printf("\n\t=====");
        printf("\nMENU");
        printf("\n----\n");
        printf("1. Tambah\n");
    }

```

```

printf("2. Lihat pre-order\n");
printf("3. Lihat in-order\n");
printf("4. Lihat post-order\n");
printf("5. Exit\n");
printf("Pilihan : ");
scanf("%d", &pil);
switch(pil)
{
//jika pil bernilai 1
case 1 :
    printf("\nINPUT : ");
    printf("\n-----");
    printf("\nData baru : ");
    scanf("%d", &data);
    //panggil fungsi untuk menambah node yang
berisi data pada tree
    tambah(&pohon, data);
    break;

//jika pil bernilai 2
case 2 :
    printf("\nOUTPUT PRE ORDER : ");
    printf("\n-----\n");
    if(pohon!=NULL)
        //panggil fungsi untuk mencetak data
secara preOrder
        preOrder(pohon);
    else
        printf("Masih kosong!");
    break;

//jika pil bernilai 3
case 3 :
    printf("\nOUTPUT IN ORDER : ");
    printf("\n-----\n");
    if(pohon!=NULL)
        //panggil fungsi untuk mencetak data
secara inOrder
        inOrder(pohon);
    else
        printf("Masih kosong!");
    break;

//jika pil bernilai 4
case 4 :
    printf("\nOUTPUT POST ORDER : ");
    printf("\n-----\n");
    if(pohon!=NULL)

```



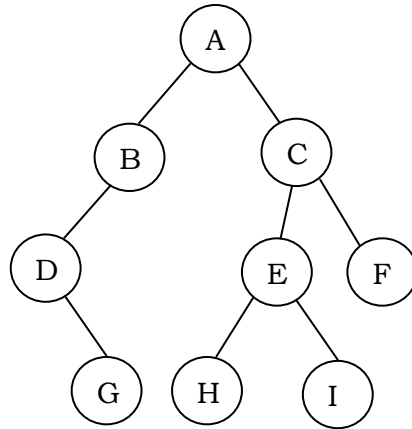
```

//panggil fungsi untuk mencetak data
secara postOrder
    postOrder(pohon);
else
    printf("Masih kosong!");
    break;
}
_getch();
}while(pil != 5); //akan diulang jika input tidak
samadengan 5
return EXIT_FAILURE;
}

```

11.8 Latihan

1. Ada tiga cara traversal dalam pohon biner, sebutkan daftar kunjungannya untuk pohon berikut secara *preorder*, *inorder* dan *postorder*.

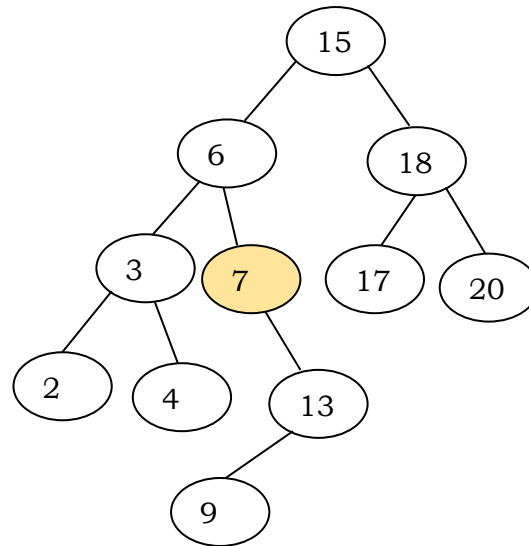


2. Buatlah Binary Tree dari notasi infix berikut. Selanjutnya lakukan pembacaan *Binary Tree* dengan *Traversal Inorder*, *Preorder* dan *Postorder*.
 - a) $A + B - C + D$
 - b) $(A + B) * (C - D)$
 - c) $(A + (B - C)) / D * E$
 - d) $A * B - C ^ D + E / F$
 - e) $((A + B) - (C ^ D)) ^ ((E - F) / (G * H))$
3. Gambarlah pohon biner yang berakar pada indeks no 6, direpresentasikan dengan atribut seperti di bawah ini.

Indeks	Key	Left	Right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

4. Tentukan proses untuk mencari data 13 pada BST di samping. Tentukan pula proses untuk mencari successor dari 4.

5. Jika node dengan data 7 dihapus dari binary search tree di samping, tentukan Langkah-Langkah penghapusannya. Jelaskan pula prosesnya jika kemudian node dengan data 6 dihapus.



6. Buatlah pohon biner dari barisan bilangan berikut:
1. 12, 22, 8, 19, 10, 9, 20, 4, 2, 6
 2. 2, 3, 4, 5, 50, 10, 15, 13, 20, 12, 10, 7
 3. 7, 13, 4, 6, 5, 9, 15, 20, 60, 14, 40, 70
7. Buatlah tambahan menu penghapusan dan pencarian data untuk program manipulasi dan simulasi *tree* berbasis menu yang sudah dibahas pada Studi Kasus!

Daftar Pustaka

- [1] Abdul Kadir, *Teori dan Aplikasi Struktur Data menggunakan C++*. Yogyakarta: Penerbit Andi, 2013.
- [2] A. Sonita and F. Nurtaneo, “Analisis Perbandingan Algoritma Bubble Sort , Merge Sort , Dan Quick Sort Dalam Proses Pengurutan Kombinasi Angka dan Huruf,” *Pseudocode*, vol. II, no. September, pp. 75–80, 2015.
- [3] L. Sitorus and D. J. M. Sembiring, *Konsep dan Implementasi Struktur Data dengan C++*. Yogyakarta: Penerbit Andi, 2012.
- [4] A. V Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. 2001.