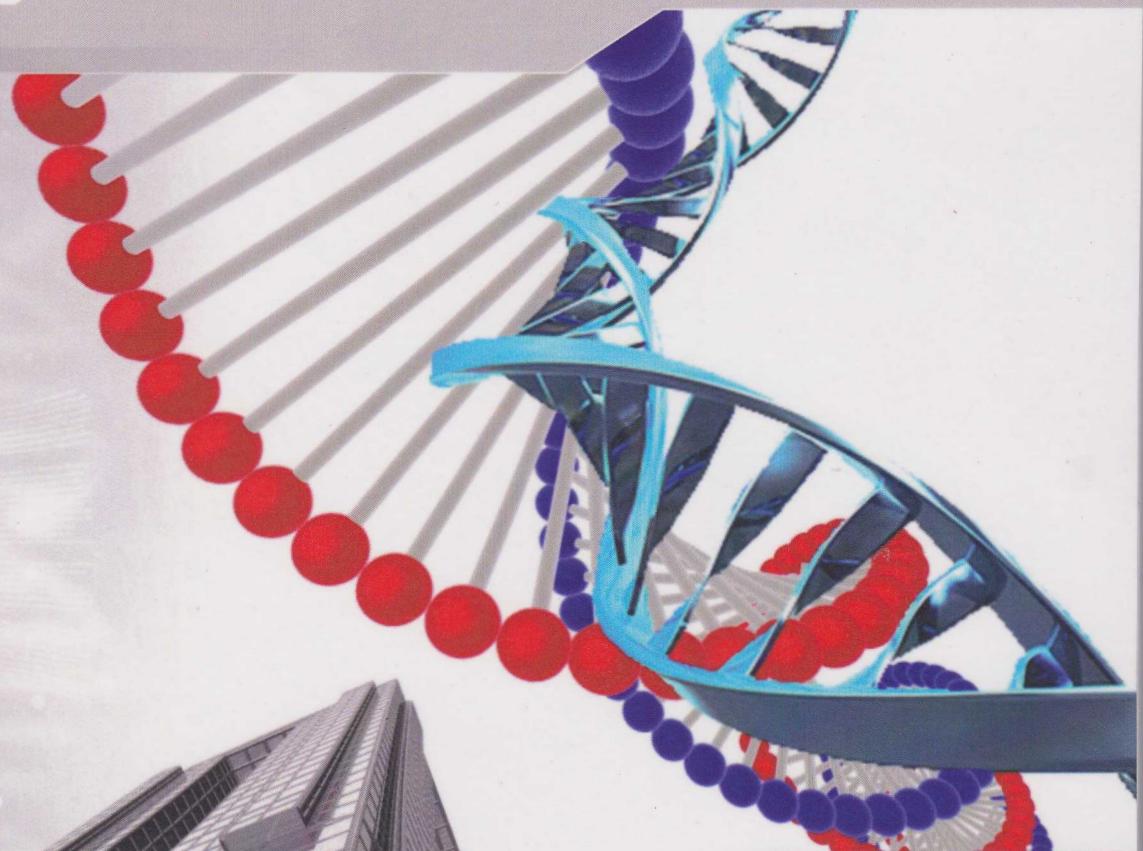


STRUKTUR DATA



Emy Setyaningsih

AKPRIND PRESS

STRUKTUR DATA

Dalam memahami teknologi informasi kita perlu memahami konsep-konsep dasar dan bagaimana teknologi tersebut berinteraksi dengan dunia nyata. Dalam hal ini struktur data merupakan salah satu konsep dasar yang perlu dipahami. Struktur data merupakan cara penyimpanan dan pengorganisasian data dalam suatu sistem komputer agar mudah diakses dan dimanfaatkan.

Buku berjudul *Struktur Data* ditulis untuk mahasiswa dan profesional pernah mempelajari strukturnya dalam bahasa pemrograman C/C++ dan Java. Untuk mahasiswa yang belum mengikuti kelas pemrograman. Pada akhirnya buku ini akan memberikan pengetahuan tentang fungsi-fungsi dasar strukturnya. Mengingat teknologi informasi saat ini semakin maju, maka buku ini akan menjadi buku referensi bagi mahasiswa dan profesional dalam struktur data.

Penulis menyadari masih banyak kerumitan dalam penulisan buku ini dalam penyelesaian materi, dan tentu saja ini. Semoga diharapkan dengan ketekunan untuk membuat buku ini menjadi lebih baik lagi dan lebih berguna bagi pembaca.

Pada kesempatan ini penulis menyampaikan ucapan terimakasih dan penghargaan kepada Keluarga Besar Institut Sains dan Teknologi AKPRIND Yogyakarta, suami dan anak-anaknya tercinta (Melinda, Nofira dan Dedi) yang selalu mendukung dan memberikan sinyal positif yang telah membentuk dirinya sejauh ini.

Emy Setyaningsih

Akhirmnya penulis berharap agar buku ini bermanfaat dapat berfungsi sebagai pedoman

Yogyakarta, 11 November 2012

Lembaran ILMU PENGETAHUAN DAN KETEKNOLOGIAN

JASA MASYARAKAT DAN KONSEP HUMANISME

Desain Grafis : Zulfikar Alifqarni

Didesain Oleh : AKPRIND PRESS

ISBN: 978-602-3814-03-8

AKPRIND PRESS

Undang- Undang Nomor 7 Tahun 1987
Tentang Hak Cipta
Pasal 44

- (1) Barang siapa dengan sengaja mengumumkan atau memperbanyak suatu ciptaan atau memberi izin untuk itu, dipidana dengan pidana penjara paling lama 7 (tujuh) tahun dan/atau denda paling banyak Rp. 100.000.000,00 (seratus juta rupiah).
- (2) Barang siapa dengan sengaja menyiarakan, memamerkan, mengedarkan, atau menjual kepada umum suatu ciptaan atau barang hasil pelanggaran Hak Cipta sebagaimana di maksud dalam ayat (1), dipidana dengan pidana penjara paling lama 5 (lima) tahun dan/atau denda paling banyak Rp. 50.000.000,00 (lima puluh juta rupiah).

STRUKTUR DATA

Hak cipta 2012 pada penulis, dilarang keras mengutip, menjiplak,

Memphoto copy baik sebagian atau keseluruhan isi buku ini

Tanpa mendapat izin tertulis dari pengarang dan penerbit

Penulis : Emy Setyaningsih

Page Make Up : Rochmad Haryanto

Desain Cover : Arham Arifuddin

Dicetak Oleh : AKPRIND PRESS

ISBN :978-602-7619-03-6

HAK CIPTA DI LINDUNGI OLEH UNDANG-UNDANG

KATA PENGANTAR

Dengan memanjatkan syukur alhamdulillah, segala puji dipanjangkan kehadirat Allah Subhanahu Wa ta'ala. Karena hanya dengan bantuan dan rahmatnya akhirnya buku ini dapat terselesaikan.

Buku berjudul *Struktur Data* disusun dengan materi yang cukup padat namun berusaha disajikan dalam bahasa yang sederhana dan banyak contoh yang ditampilkan dengan menggunakan bahasa pemrograman Pascal dengan harapan pembaca dapat lebih mudah memahaminya. Mengingat cukup lengkapnya materi yang disajikan dalam buku ini, maka buku ini dapat dijadikan sebagai buku teks maupun buku referensi bagi mahasiswa maupun dosen untuk mata kuliah struktur data.

Penulis menyadari masih banyak kekurangan dalam penulisan buku ini baik dalam penyajian materi, dan tata bahasa. Semoga dihari hari mendatang ada kesempatan untuk membuat buku ini menjadi lebih baik lagi dan lebih berguna bagi pembaca.

Pada kesempatan ini penulis menyampaikan ucapan terima kasih dan penghargaan kepada Keluarga Besar Institut Sains dan Teknologi AKPRIND Yogyakarta, suami dan anak-anakku tercinta (Malecita, Nohan dan Salsabila), serta pihak-pihak lain yang tidak dapat kami sebutkan satu persatu yang telah mendukung terselesainya buku ini.

Akhirnya penulis berharap agar buku ini benar-benar dapat bermanfaat.

Yogyakarta, November 2012

Penulis

DAFTAR ISI

	Hal
KATA PENGANTAR	i
DAFTAR ISI	ii
DAFTAR GAMBAR	iv
DAFTAR TABEL	vii
BAB I. STRUKTUR DATA DENGAN PASCAL	1
1.1. Pendahuluan	1
1.2. Hirarki Tipe Data Pada Pascal	2
1.2.1. Tipe data sederhana	2
1.2.2. Tipe data terstruktur	4
1.2.3. Tipe data pointer	10
1.3. Struktur Data Statis	14
1.4. Struktur Data Dinamis	15
1.5. Procedure dan Function	15
1.6. Latihan	17
BAB II. SORTING	19
2.1. Pendahuluan	19
2.2. Exchange Sort	20
2.2.1. Bubble sort.....	20
2.2.2. Quicksort.....	23
2.3. Selection Sort	26
2.3.1. Selection sort.....	26
2.3.2. Heap sort.....	29
2.4. Insertion Sort	32
2.5. Radix Sort	34
2.6. Latihan	36
BAB III. LINKED LIST	37
3.1. Single Linked List	37
3.1.1. Deklarasi single linked list	39
3.1.2. Operasi pada single linked list	40
3.1.3. Membaca isi linked list	49
3.1.4. Mencari data dalam linked list	51
3.2. Double Linked List	52

3.2.1. Pendeklarasian struktur dan varia-bel double linked list	52
3.2.2. View data	53
3.2.3. Tambah data	53
3.2.4. Hapus data	60
3.2.5. Pencarian data.....	67
3.3. Latihan	70
BAB IV. STACK (TUMPUKAN)	71
4.1. Representasi Stack Dengan Array	72
4.1.1. Single Stack.....	73
4.1.2. Double Stack.....	76
4.2. Representasi Stack dengan Single Linked List	79
4.3. Implementasi Stack Untuk Mengkonversi Bilangan Desimal ke Bilangan Biner	82
4.4. Latihan	86
BAB V. QUEUE (ANTRIAN)	87
5.1. Representasi Queue dengan Array	88
5.1.1. Representasi queue dengan peng-geseran.....	91
5.1.2. Queue melingkar	93
5.2. Representasi Queue Dengan Linked List	95
5.3. Contoh Penerapan Queue	98
5.4. Latihan	100
BAB VI. TREE	101
6.1. Terminologi Tree.....	101
6.2. Binary Tree.....	102
6.2.1. Jenis binary tree	102
6.2.2. Aplikasi pada binary tree.....	103
6.3. Binary Search Tree.....	104
6.3.1. Operasi pada binary search tree	104
6.3.2. Model kunjungan pada binary search tree	106
6.3.3. Notasi Prefix, Infix, dan Postfix	108
6.4. Implementasi Binary Tree.....	108
6.5. Latihan	112
DAFTAR PUSTAKA	113

DAFTAR GAMBAR

	Hal
Gambar 1.1 Struktur Array	5
Gambar 1.2. Ilustrasi Perubah Statis	11
Gambar 1.3. Ilustrasi Perubah Dinamis	11
Gambar 1.4. Program dibagi-bagi menjadi beberapa subprogram	16
Gambar 2.1. Ilustrasi algoritma bubble sort untuk pengurutan secara ascending	20
Gambar 2.2. Ilustrasi algoritma bubble sort pada proses 3 s.d 5	21
Gambar 2.3. Flowchart dari algoritma bubble sort	22
Gambar 2.4. Pohon rekursif untuk best case pada algoritma quick sort	23
Gambar 2.5. Pohon rekursif untuk worst case pada algoritma quick sort	23
Gambar 2.6. Pohon rekursif untuk average case pada algoritma quick sort	24
Gambar 2.7. Ilustrasi algoritma quick Sort	24
Gambar 2.8. Ilustrasi algortima quick sort	25
Gambar 2.9. Flowchart algortima quick sort	25
Gambar 2.10. Flowchart algortima selection sort	28
Gambar 2.11. Urutan pengisian data pada heap	29
Gambar 2.12. Proses insert heap	29
Gambar 2.13. Pengurutan menggunakan metode upheap	29
Gambar 2.14. Pengurutan menggunakan metode downhap	30
Gambar 2.15. Proses 1 dari proses heap sort	30
Gambar 2.16. Hasil heap sort pada iterasi pertama	31
Gambar 2.17. Proses delete heap	31
Gambar 2.18. Heap sort pada iterasi ke dua	31
Gambar 2.19. Heap Sort pada iterasi ke tiga	31
Gambar 2.20. Flowchart algortima insertion sort	33
Gambar 2.21. Program procedure algoritma insertion sort	33
Gambar 3.1. Bagian linked list	38
Gambar 3.2. Ilustrasi Single Linked List	38
Gambar 3.3. Linked list dengan medan informasi berisi 1 field data	39
Gambar 3.4. Linked list dengan medan informasi berisi 2 field	40
Gambar 3.5. Ilustrasi penambahan simpul dibelakang pada saat linked list masih kosong	41
Gambar 3.6. Ilustrasi penambahan simpul di awal pada saat linked list masih kosong	42
Gambar 3.7. Ilustrasi linked list kosong	45
Gambar 3.8. Ilustrasi penghapusan simpul pada linked list yang berisi 1 simpul	45
Gambar 3.9. Ilustrasi proses membaca isi simpul secara maju	50

Gambar 3.10. Ilustrasi proses membaca isi simpul secara mundur	50
Gambar 3.11. Ilustrasi double linked list	52
Gambar 3.12. Dekripsi simpul double linked list	52
Gambar 3.13. Dekripsi simpul double linked list setelah dideklarasikan	52
Gambar 3.14. Ilustrasi simpul double linked list yang berisi 1 simpul.....	53
Gambar 3.15. Ilustrasi simpul double linked list dalam kondisi kosong	54
Gambar 3.16. Ilustrasi simpul double linked list yang berisi 1 simpul.....	54
Gambar 3.17. Ilustrasi penambahan di akhir simpul double linked list yang berisi 1 simpul	55
Gambar 3.18. Simpul double linked list kosong yang telah ditambahkan simpul baru	56
Gambar 3.19. Contoh double linked list dengan 4 simpul	57
Gambar 3.20. Ilustrasi double linked list yang akan disisipi simpul pada posisi ke-3	59
Gambar 4.1. Contoh stack	71
Gambar 4.2. Operasi Push	72
Gambar 4.3. Operasi POP	72
Gambar 4.4. Contoh operasi dasar pada stack	72
Gambar 4.5. Ilustrasi Single Stack	73
Gambar 4.6. Representasi stack menggunakan array	73
Gambar 4.7. Ilustrasi double stack	76
Gambar 4.8. Representasi stack menggunakan single linked list	79
Gambar 4.9. Struktur untuk node stack	79
Gambar 5.1. Ilustrasi queue	87
Gambar 5.2. Contoh model antian	87
Gambar 5.3. Contoh model antian pasien	88
Gambar 5.4. Contoh antrian dengan 6 elemen	88
Gambar 5.5. Ilustrasi penambahan elemen pada antrian	89
Gambar 5.6. Ilustrasi penghapusan elemen pada antrian	89
Gambar 5.7. Ilustrasi penambahan dan pengurangan pada antrian	90
Gambar 5.8. Ilustrasi penambahan dan pengurangan pada antrian menggunakan peng- Geseran	92
Gambar 5.9. Ilustrasi circular array	93
Gambar 5.10. Ilustrasi penambahan dan pengurangan pada antrian menggunakan circu- lar aray	94
Gambar 5.11. Ilustasi queue menggunakan linked list	96
Gambar 5.13. Ilustrasi dequeue	98
Gambar 6.1. Contoh Tree	102
Gambar 6.2. Contoh binary Trees	102
Gambar 6.3. Contoh full binary Tree	102

Gambar 6.4.	Contoh Complete binary Tree	103
Gambar 6.5.	Contoh skewed binary Tree	103
Gambar 6.6.	Contoh arithmetic expressions Tree	103
Gambar 6.7.	Contoh decision tree	104
Gambar 6.8.	Sifat binary search Tree	104
Gambar 6.9.	Operasi insert pada binary search Tree	105
Gambar 6.10.	Operasi delete (4) binary search Tree	105
<i>Gambar 6.11.</i>	<i>Operasi delete (3) binary search Tree</i>	105
Gambar 6.12.	Kunjungan Preorder	106
Gambar 6.13.	Kunjungan Inorder	106
Gambar 6.14.	Kunjungan PostOrder	107
Gambar 6.15.	Kunjungan level order	107
<i>Gambar 6.16.</i>	<i>Notasi Prefix</i>	108
Gambar 6.17.	Implementasi Binary Tree	109

DAFTAR TABEL

	Hal
Tabel 1.1 Macam-macam tipe bilangan bulat	2
Tabel 1.2. Macam-macam tipe bilangan real	4

BAB I

STRUKTUR DATA DENGAN PASCAL

1.1. Pendahuluan

Ketika kita mempelajari suatu bahasa pemrograman, kita akan menjumpai elemen-elemen yang pada dasarnya serupa antara satu bahasa dengan bahasa yang lain. Hal itu dikarenakan elemen-elemen tersebut merupakan bagian dari tata bahasa pemrograman yang bersangkutan.

Definisi data adalah fakta atau kenyataan yang tercatat mengenai suatu obyek. Pengertian data ini menyiratkan suatu nilai yang bisa dinyatakan dalam bentuk konstanta atau variabel. Struktur data adalah abstraksi model penyimpanan/pengaturan/susunan data di dalam memori/RAM komputer atau koleksi dari suatu variabel yang dapat dinyatakan dengan sebuah nama, dengan sifat setiap variabel dapat memiliki tipe yang berlainan. Struktur data biasa dipakai untuk mengelompokkan beberapa informasi yang berkaitan menjadi sebuah kesatuan.

Struktur data diperlukan dalam rangka membuat program komputer. Untuk menyusun sebuah program komputer diperlukan tiga macam komponen dasar, yaitu:

1. Algoritma
2. Bahasa pemrograman
3. Struktur data

Aspek yang berkaitan dengan algoritma adalah efisiensi algoritma yang sering disebut ukuran algoritma. Ukuran algoritma ditentukan oleh dua hal, yaitu:

1. Efisiensi waktu
2. Efisiensi memori

Aspek yang berkaitan dengan bahasa pemrograman adalah meliputi:

1. Sintaks
2. Reserved word
3. Function
4. Procedure

Aspek yang berkaitan dengan struktur data adalah meliputi:

1. Nilai data (data value), yaitu numerik atau non numerik
2. Relasi antar data
3. Prosedur/fungsi atau operasi pada data

Operasi pada data dapat dibedakan menjadi dua macam, yaitu:

1. Operasi menambahkan (insert) data
2. Operasi menghapus (delete) data

1.2. Hirarki Type Data Pada Pascal

Pascal telah menyediakan beberapa tipe data yang sudah siap dipakai. Pada saat mendeklarasikan sebuah variabel secara otomatis harus mendeklarasikan tipe data yang dapat ditampung oleh variabel tersebut.

Tipe data dalam Pascal dibagi menjadi 3 bagian

1. Tipe data Sederhana
2. Type data Terstruktur
3. Type data Pointer

1.2.1. Tipe data sederhana

Tipe data sederhana merupakan tipe data yang paling kecil, yang hanya melibatkan satu item data, misalnya tipe data integer, string, real, Boolean, dan sebagainya. Kita dapat juga mendefinisikan sendiri tipe data ini. Tipe data yang didefinisikan sendiri tersebut diistilahkan enumerated data type

(1) Tipe bilangan bulat

Tipe data ini digunakan untuk menyimpan bilangan bulat. Macam-macam tipe bilangan bulat dalam Pascal dapat dilihat pada Tabel 1.1.

Tabel 1.1. Macam-macam tipe bilangan bulat

Tipe	Jangkauan	Ukuran
Shortint	-128 ... 127	8 bit
Integer	-32768 ... 32767	16 bit
Longint	-2147483648 ... 2147483647	32 bit
Byte	0 ... 255	8 bit
Word	0 ... 65535	16 bit

Untuk memberi nilai pada tipe bilangan bulat dapat menggunakan basis decimal maupun heksadesimal yang ditandai dengan tanda \$

Contoh :

```
Var
  x, y : integer;
begin
  x := 16;      { dengan decimal }
  y := $0A;    { dengan hexadecimal }
end.
```

(2) Tipe boolean

Tipe data ini hanya dapat bernilai benar dan salah. Tipe Boolean ukurannya 1 byte.

Contoh :

```
Var  
  B1 : boolean;  
begin  
  b1 := true;  
  b1 := false;  
end.
```

(3) Tipe karakter

Tipe data ini digunakan untuk menyimpan data alfanumeris seperti 'A', 'Z', '@', dsb.. Tipe data char ukurannya 1 byte.

Contoh :

```
Var  
  ch : char;  
begin  
  ch := 'A';  
  ch := #65 { sama artinya dengan ch := 'A' }  
  ch := chr(65); { sama artinya dengan ch := 'A' }  
end.
```

(4) Tipe subjangkauan

Tipe data ini memungkinkan Anda mendeklarasikan tipe yang berada pada jangkauan tertentu.

Tipe ini hamper sama dengan tipe bilangan bulat, bedanya Anda bebas menentukan jangkauan dari tipe ini.

Contoh :

```
Type  
  Bulan = 1 .. 12;  
Var  
  Januari : Bulan;  
begin  
  Januari := 1;  
End.
```

(5) Tipe terbilang

Tipe data ini memungkinkan Anda memberi nama pada beberapa nilai tertentu.

Contoh :

```
Type  
  TipeHari=(Minggu,Senin,Selasa,Rabu,Kamis,Jumat, Sabtu);  
Var  
  hari : TipeHari;  
begin  
  hari := Minggu;  
  hari := Senin;  
end.
```

(6) Tipe real

Tipe data ini digunakan untuk menyimpan bilangan real. Macam-macamnya tipe bilangan real dalam Pascal dapat dilihat pada Tabel 1.2.

Tabel 1.2. Macam-macam tipe bilangan real

Tipe	Jangkauan	Digit penting	Ukuran
Real	$2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$	11 – 12	6 byte
Single	$1.5 \times 10^{-45} \dots 3.4 \times 10^{38}$	7 – 8	4 byte
Double	$5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$	15 – 16	8 byte
Extended	$3.4 \times 10^{-4932} \dots 1.1 \times 10^{4932}$	19 – 20	10 byte
Comp	$-2^{63} + 1 \dots 2^{63} - 1$	19 – 20	8 byte

Contoh :

```
Var
  x, y : real;

begin
  x := 123.45;      {menuliskan nilai dengan tanda titik}
  y := 1.2345E+2 { menuliskan nilai dengan eksponen}
end.
```

(7) Tipe string

Tipe data ini digunakan untuk menyimpan data yang berupa unaian karakter.

Contoh :

```
Var
  kalimat : string;

begin
  kalimat := 'IST AKPRIND';
end.
```

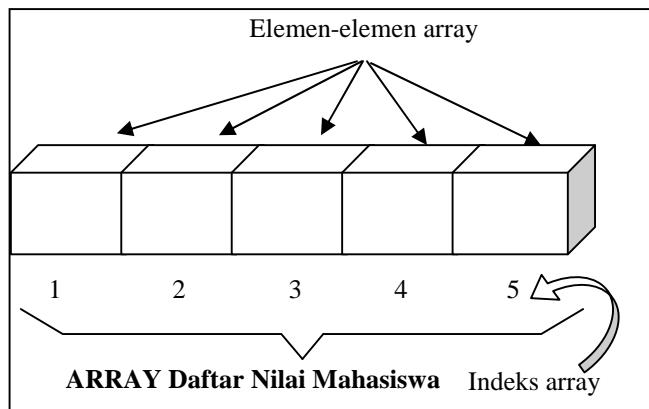
String adalah suatu jenis variabel yang tidak dapat ‘di operasikan’ dengan operator matematika. String lebih banyak menyebut sebagai variabel yang berupa teks. Suatu variabel string tidak harus berisi suatu teks (rangkaian karakter/huruf) tetapi bisa juga berisi suatu angka.

1.2.2. Tipe data terstruktur

Tipe data terstruktur merupakan tipe data yang menampung beberapa item data. Bentuk dari tipe data ini dapat berupa array (terdiri dari item-item yang memiliki tipe data yang sama) ataupun record (terdiri dari item-item yang boleh memiliki tipe data yang berbeda).

(1) Type larik (array)

Array adalah suatu tipe data terstruktur yang terdapat dalam memori yang terdiri dari sejumlah elemen (tempat) yang mempunyai tipe data yang sama dan merupakan gabungan dari beberapa variabel sejenis serta memiliki jumlah komponen yang jumlahnya tetap.



Gambar 1.1. Struktur Array

Array (biasa juga disebut **larik**) merupakan tipe data terstruktur yang berguna untuk menyimpan sejumlah data yang bersifat sama. Bagian yang menyusun array biasa dinamakan elemen array. Masing-masing elemen dapat diakses tersendiri, melalui **indeks array**, seperti terlihat pada gambar 1.1.

Elemen-elemen dari array tersusun secara sequential dalam memori komputer. Array dapat berupa satu dimensi, dua dimensi, tiga dimensi ataupun banyak dimensi.

a. Array berdimensi satu

Array berdimensi satu dapat digambarkan sebagai kotak panjang yang terdiri atas beberapa kotak kecil seperti terlihat pada Gambar 1.1. Dalam gambar tersebut, array memiliki 5 buah elemen. Untuk membentuk array seperti pada gambar 1.1, diperlukan pendeklarasian sebagai berikut:

```
const
    maks_elemen = 5;
var
    x : array [1 .. maks_elemen] of real;
```

Pada contoh ini, X dapat menampung 5 buah elemen bertipe **Real**. Yang menjadi kata-kata kunci pendeklarasian array adalah kata cadang **ARRAY**. Banyaknya komponen dalam suatu larik ditunjukkan oleh suatu indeks yang disebut dengan tipe indeks (index type). Tiap-tiap komponen larik dapat diakses dengan menunjukkan nilai indeksnya.

Bentuk umum :

Var *nama* : *array[index] of tipe*

dengan :

var, array, of kata cadangan yang harus ada.
 nama nama larik yang akan dideklarasikan.
 index batas yang akan ada pada larik yg akan dideklarasikan(cacah elemnya).
 tipe tipe larik.

Array dapat bertipe data sederhana seperti byte, word, integer, real, boolean, char, string dan tipe data scalar atau subrange.

Contoh :

```
Var
  x : array[1..10] of integer;
  b : integer;

begin
  b := 3;
  x[1] := 39;
  x[2] := 42;
  x[3] := x[1] + 50;
  x[4] := x[2] + b;
  x[5] := x[3] + x[4];
end.
```

b. Array berdimensi dua

Sejauh ini struktur yang telah dibahas merupakan array yang bekerja dengan daftar linear yang dikases dengan satu subskrip, yang disebut array satu dimensi. Array satu dimensi sebagai item-item kolom tunggal yang semua itemnya bertipe sama. Kadang-kadang kita perlu membuat struktur yang lebih kompleks yang mempunyai dua dimensi yaitu berupa baris dan kolom.

Bentuk umum :

Var nama : array[index1,index2] of tipe

dengan :

var, array, of kata cadangan yang harus ada.
 nama nama larik yang akan dideklarasikan.
 Index1 batas pertama yang akan ada pada larik yg akan dideklarasikan(cacah elemen pada baris).
 Index2 batas ke dua yang akan ada pada larik yg akan dideklarasikan(cacah elemen pada kolom).
 tipe tipe larik.

Sebagai contoh, andaikata kita akan menyimpan 3 tes untuk 100 mahasiswa kita dapat membuat tabel sebagai berikut :

	Test 1	Test 2	Test 3
Mahasiswa 1			
Mahasiswa 2			
Mahasiswa 3			

Subskrip pertama mewakili jumlah mahasiswa dan subskrip kedua mewakili jumlah tes. Jika nama arraynya TestMhs, maka TestMhs[1,2] berarti mengandung nilai untuk mahasiswa pertama, tes yang ke dua. TestMhs[3,1] mengandung nilai untuk mahasiswa ke tiga test yang pertama.

Struktur ini dapat dideklarasikan :

```
Const  
  MakBaris = 100;  
  MakKolom = 3;  
  
Type  
  TipeArray=array[1..MakBaris,1..MakKolom] of Real;  
  
Var  
  A : TipeArray;  
  N, M, indekBaris,indekKolom : integer;
```

Atau dengan menentukan maksimum baris dan kolom dituliskan dalam type array seperti contoh berikut :

```
Type  
  TipeArray = array[1..100,1..10] of Real;  
  
Var  
  A : TipeArray;  
  N, M, indekBaris,indekKolom : integer;
```

Dua variabel N dan M mewakili jumlah baris dan jumlah kolom yang digunakan untuk menyimpan data. Dengan segmen program ini, kita dapat menulis prosedur untuk memuat data ke dalam array. Pemakai pertama kali memasukkan nilai untuk N dan M dan kemudian memasukkan dalam setiap posisi pada array tersebut, pada saat yang berbeda. Contoh Prosedur tersebut adalah :

```
Begin  
  Write('Masukkan jumlah Baris dan Kolom'); Readln(N,M);  
  { prosedur memasukkan nilai tes urut baris }  
  for indekBaris :=1 to N do begin  
    for indekKolom := 1 to M do  
      begin  
        write('Masukkan nilai tes mahasiswa ke-','indekBaris,' Nilai Tes ke-','indekkolom);  
        readln(A[indekBaris,indekKolom]);  
      end;  
  end;
```

c. Array Dengan Tiga Atau Lebih Dimensi

Bentuk umum :

Var nama : array[index1,index2,...,indexn] of tipe

dengan :

var, array, of	kata cadangan yang harus ada.
nama	: nama larik yang akan dideklarasikan.
Index1	: batas pertama yang akan ada pada larik yg akan dideklarasikan
Index2	: batas ke dua yang akan ada pada larik yg akan dideklarasikan
Indexn	: batas ke n yang akan ada pada larik yg akan dideklarasikan
tipe	: tipe larik.

Contoh :

Menghitung rata-rata nilai tes beasiswa untuk semua peserta di semua gelombang, jika diketahui Gel : menunjukkan gelombang tes yang terdiri dari 2 , Sn : jumlah peserta tes beasiswa tiap gelombang dan Nt :Nilai tes yang terdiri dari 3 materi tes.

```

Var
    Nilai          : array[1..2,1..100,1..3] of Real;
    Gel, Sn, jmlpeserta, Nt,N: integer;
    Rata,ratanil,totnil,total   : real;
Begin
    Total := 0; N:=0;
    For Gel := 1 to 2 do begin
        Write( Jumlah peserta tes beasiswa gel ke- ,Gel,  adalah : );readln(jmlpeserta);
        For Sn := 1 to jmlpeserta do Begin
            Totnil := 0;
            For Nt := 1 to 3 do
                Write( Nilai Tes ke- ,Nt,  adalah : );readln(Nilai[Gel,Sn,Nt]);
                Totnil := Totnil + Nilai[Gel,Sn,Nt];
            End;
            Total := Total + Totnil/3;
        End;
        N := N + jmlpeserta;
    End;
    Rata := Total / N;
End.

```

(2) Tipe rekaman

Seperti pada larik, rekaman (record) adalah kumpulan data. Perbedaan antara larik dengan rekaman adalah bahwa dalam larik semua elemennya harus bertipe sama, tetapi dalam rekaman setiap elemen dapat mempunyai tipe data yang berbeda satu sama lainnya. Untuk mendeklarasikan rekaman selalu diawali dengan nama record tanda sama dengan (=) dan kata kunci record serta diakhiri oleh kata kunci end. Field-field dari record tersebut diletakkan diantara kata kunci record dan end.

Bentuk umum :

```
type
  pengenal1 = record
    medan1 : tipe1;
    medan2 : tipe2;
    .
    medann : tipen;
  end;
```

dimana :

pengenal : pengenal yang menunjukkan tipe data yang akan dideklarasikan.
 medan : nama medan yang akan digunakan.
 tipe : sembarang tipe data yang telah dideklarasikan sebelumnya.

Contoh :

```
Type
  data = record
    Nim   : string[10];
    Nama  : string;
    IPK   : real;
  End;
Var
  Mhs1 : data;
  Mhs2 : array[1..3] of data;
Begin
  Mhs1.nim    := 1110710001 ;
  Mhs1.nama   := MALECITA ;
  Mhs1.IPK    := 3.89;
  Mhs2[1].nim := 1110710005 ;
  Mhs2[1].nama := NOHAN ;
  Mhs2[1].IPK := 3.75;
  Mhs2[2].nim := 1110710015 ;
  Mhs2[2].nama := SALSA ;
  Mhs2[2].IPK := 3.80;
End.
```

(3) Tipe himpunan

Set merupakan tipe data terstruktur yang terdiri dari elemen yang disebut Anggota set. Anggota set ini tidak memiliki urutan dan tidak boleh ada dua anggota set yang sama. Sebuah set dalam Pascal hanya dapat memuat maksimal 255 anggota dan hanya mempunyai satu tipe.

Bentuk Umum :

Type

<namatipe> = set of <tipedata>;

Contoh :

```
Type
  Karakter = set of char ;
  Tanggal = set of 1..31;
  Hari   = set of (Senin, Selasa, Rabu, Kamis, Jumat,Sabtu, Minggu);
Var
  Kar   : karakter;
  tgl  : tanggal;
  smg  : hari;
```

nilai-nilai dalam suatu set dapat dituliskan dengan beberapa cara, yaitu disebutkan satu persatu(enumerasi), atau dituliskan dalam rentang tertentu.

a. Notasi set enumerasi

Elemen-elemen yang terdapat dalam set dinyatakan satu persatu.

Contoh :

```
Angka := [1,2,3,4,5];
Huruf := [ 'A', 'B', 'C'];
```

b. Notasi set rentang

Elemen-elemen yang dinyatakan secara rentang berdasarkan tipe dasar set tersebut.

Contoh :

```
Angka := [1..6];
Huruf := [ 'A'.. 'C', 'G'.. 'Z'];
```

Pascal menyediakan operator hubungan yang dapat digunakan untuk mengetahui (tes) keanggotaan himpunan. Contoh operator **In** yang berarti termasuk sebagai anggota. ‘a’ in [‘a’,‘b’,‘c’] adalah true.

Contoh :

(X >= 1) and (X <= 10)

hubungan ini true(benar) ketika x dalam jangkauan 1..10 sehingga dapat ditulis:

X in [1..10];

Tipe ini digunakan untuk menyimpan kumpulan nilai yang bertipe sama.

1.2.3. Tipe data pointer

(1) Pengertian pointer

Tipe data pointer digunakan untuk menunjuk pada alamat memory suatu data yang lain. Jadi tipe data pointer pada dasarnya tidak menyimpan nilai data secara langsung,melainkan hanya menyimpan alamat dimana data berada. Dengan melihat sifat-sifat perubah statis, bisa kita katakan bahwa banyaknya data yang bisa diolah adalah terbatas. Sebagai contoh, misalnya kita mempunyai perubah yang kita deklarasikan sebagai :

```
var Tabel : array[1..100, 1..50] of integer;
```

Perubah **Tabel** di atas hanya mampu untuk menyimpan data sebanyak $100 \times 50 = 5000$ buah data. Jika kita tetap nekat untuk menambah data pada perubah tersebut, eksekusi program akan terhenti karena deklarasi lariknya kurang. Memang kita bisa mengubah deklarasi di atas untuk memperbesar ukurannya. Tetapi jika setiap kali kita harus mengubah deklarasi di atas, sementara banyaknya data tidak bisa kita tentukan lebih dahulu, hal ini tentu merupakan pekerjaan yang membosankan. Sekarang bagaimana jika kita ingin mengolah data yang banyaknya tidak bisa kita pastikan sebelumnya, sehingga kita tidak yakin dengan deklarasi larik kita?

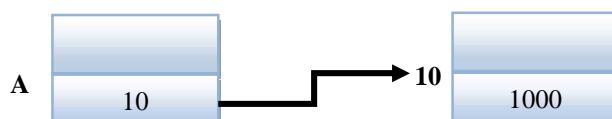
Untuk menjawab pertanyaan di atas, Pascal menyediakan satu fasilitas yang memungkinkan kita menggunakan suatu perubah yang disebut dengan perubah dinamis (*dynamic variable*). Perubah dinamis adalah suatu perubah yang akan dialokasikan hanya pada saat diperlukan, yaitu setelah program dieksekusi. Dengan kata lain, pada saat program dikompilasi, lokasi untuk perubah tersebut belum ditentukan. Kompiler hanya akan mencatat bahwa suatu perubah akan diperlakukan sebagai perubah dinamis. Hal ini membawa keuntungan pula, bahwa perubah-perubah dinamis tersebut bisa dihapus pada saat program dieksekusi, sehingga ukuran memori akan selalu berubah. Hal inilah yang menyebabkan perubah dinamakan sebagai perubah dinamis.

Pada perubah statis, isi memori pada lokasi tertentu (nilai perubah) adalah data sesungguhnya yang akan kita olah. Pada perubah dinamis, nilai perubah adalah alamat lokasi lain yang menyimpan data yang sesungguhnya. Dengan demikian data yang sesungguhnya tidak bisa diakses secara langsung.

Cara pengaksesan data bisa diilustrasikan seperti gambar di bawah ini :



Gambar 1.2. Ilustrasi Perubah Statis



Gambar 1.3. Ilustrasi Perubah Dinamis

Pada gambar 1.2. perubah A adalah perubah statis. Dalam hal ini 1000 adalah nilai data yang sesungguhnya dan disimpan pada perubah (lokasi) A. Pada gambar 1.3. perubah A adalah perubah dinamis. Nilai perubah ini, misalnya adalah 10. Nilai ini bukan nilai data yang sesungguhnya, tetapi lokasi dimana data yang sesungguhnya berada. Jadi dalam hal ini nilai data yang sesungguhnya tersimpan pada lokasi 10.

Dari ilustrasi di atas bisa dilihat bahwa nilai perubah dinamis akan digunakan untuk menunjuk ke lokasi lain yang berisi data sesungguhnya yang akan diproses. Karena alasan inilah perubah dinamis lebih dikenal dengan sebutan pointer yang artinya menunjuk ke sesuatu. Dalam

perubah dinamis, nilai data yang ditunjuk oleh suatu pointer biasanya disebut sebagai simpul/node.

(2) Deklarasi pointer

Pendeklarasian pointer hampir sama dengan pendeklarasian variabel biasa, bedanya hanya dengan menambahkan tanda ^ di depan tipe pointer.

Bentuk umum deklarasi pointer adalah :

type

pengenal = ^**simpul**;

simpul = **tipe**;

dengan

pengenal : nama pengenal yang menyatakan data bertipe pointer

simpul : nama simpul

tipe : tipe data dari simpul

Tanda ^ di depan nama **simpul** harus ditulis seperti apa adanya dan menunjukkan bahwa **pengenal** adalah suatu tipe data pointer. Tipe data simpul yang dinyatakan dalam **tipe** bisa berupa sembarang tipe data, misalnya **char**, **integer**, atau **real**.

Sebagai contoh :

Type

Bulat = ^**integer**;

Dalam contoh di atas **Bulat** menunjukkan tipe data baru, yaitu bertipe pointer. Dalam hal ini pointer tersebut akan menunjuk ke suatu data yang bertipe integer. Dengan deklarasi diatas, maka kita bisa mempunyai deklarasi perubah, misalnya:

var

X, K : **Bulat**;

Yang menunjukkan bahwa **X** dan **K** adalah perubah bertipe pointer yang hanya bisa mengakses data yang bertipe **integer**. Dalam kebanyakan program-program terapan, biasanya terdapat sekumpulan data yang dikumpulkan dalam sebuah rekaman (**record**), sehingga anda akan banyak menjumpai tipe data pointer yang elemennya (data yang ditunjuk) adalah sebuah rekaman. Perhatikan contoh berikut :

type

Pmhs = ^**Data**;

Data = **record**

Nim : **Str[10]**;

Nama : **Str[30]**;

IPK : **real**;

end;

Dengan deklarasi tipe data seperti di atas, kita bisa mendeklarasikan perubah, misalnya :

var

P1,P2 : **Pmhs**;

A,B,C : **String**;

Pada contoh di atas, **P1** dan **P2** masing-masing bertipe pointer; **A**, **B** dan **C** perubah statis yang bertipe **string**.

Pada saat program dikompilasi, perubah P1 dan P2 akan menempati lokasi tertentu dalam memori. Kedua perubah ini masing-masing belum menunjuk ke suatu simpul. Pointer yang belum menunjuk ke suatu simpul lainnya dinyatakan sebagai **nil**.

Untuk mengalokasikan simpul dalam memori, statemen yang digunakan adalah statemen **new**, yang mempunyai bentuk umum :

new (perubah);

dengan **perubah** adalah nama perubah yang bertipe pointer.

Sebagai contoh, dengan deklarasi perubah seperti di atas dan statemen:

new (P1);

new (P2);

Contoh :

```
Var
  P1 : ^integer;
  P2 : ^char;
```

(3) Operasi pointer

Secara umum ada dua operasi dasar yang bisa kita lakukan menggunakan data yang bertipe pointer. ada 2 operasi dasar yang bisa dilakukan menggunakan data yang bertipe pointer *yang mempunyai deklarasi yang sama* :

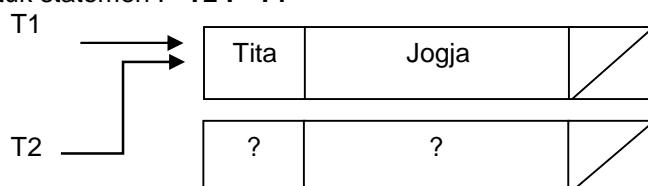
a. Mengkopi pointer

mengakibatkan sebuah simpul akan ditunjuk oleh lebih dari sebuah pointer. Jika didalam statemen pemberian tanda **^** tidak ditulis, operasinya disebut operasi mengkopi pointer.

Konsekuensinya simpul yang semula ditunjukkan oleh pointer akan terlepas

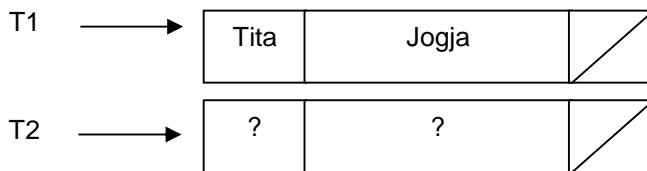


Untuk statemen : **T2 := T1**



b. Mengkopi isi simpul

mengakibatkan dua atau lebih simpul yang ditunjuk oleh pointer yang berbeda mempunyai isi yang sama. Jika statemen pemberian tanda **^** ditulis, operasinya merupakan mengkopi isi simpul pointer, konsekuensinya isi dua simpul atau lebih akan sama.



Untuk statemen : **T2^A := T1^A**



(4) Menghapus pointer

Di atas telah dijelaskan bahwa pointer yang telah dialokasikan (dibentuk) bisa didealokasikan (dihapus) kembali pada saat program dieksekusi. Setelah suatu pointer dihapus, maka lokasi yang semula ditempati oleh simpul yang ditunjuk oleh pointer tersebut akan bebas, sehingga bisa digunakan oleh perubah lain.

Statemen untuk menghapus pointer adalah **dispose**, yang mempunyai bentuk umum :

dispose (perubah)

dengan perubah adalah sembarang perubah yang bertipe pointer. Sebagai contoh, dengan menggunakan deklarasi :

```
type
  Pmhs = ^Data;
  Data = record
    Nim   : Str[10];
    Nama  : Str[30];
    IPK   : real;
  end;
var
  Mhs, Mhs1 : Pmhs;
```

Kemudian kita membentuk simpul baru, yaitu :

```
new (Mhs);
Mhs1 := Mhs;
```

Pada suatu saat, simpul yang ditunjuk oleh pointer **Mhs1** tidak digunakan lagi, maka bisa dihapus dengan menggunakan statemen :

dispose (Mhs1)

Demikian penjelasan tentang perubah dinamis yang lebih dikenal dengan sebutan pointer..

1.3. Struktur Data Statis

Struktur data statis adalah struktur data yang kebutuhan memorinya tetap/*fixed* selama program dijalankan. Struktur data statis mempunyai kelemahan, yaitu:

1. Kebutuhan memori terbatas sesuai definisi larik/array
2. Kebutuhan memori tidak fleksibel

Sedangkan keuntungan struktur data statis adalah:

1. Pemrograman relatif mudah
2. Pemrograman praktis
3. Pemrograman sederhana

1.4. Struktur Data Dinamis

Bersifat dinamis/fleksibel, dalam arti bisa berubah-ubah selama program berjalan. Struktur data dinamis mempunyai kelemahan, yaitu:

1. Pemrograman relatif rumit
2. Pemrograman tidak praktis

Sedangkan keuntungan struktur data dinamis adalah:

1. Kebutuhan memori lebih efisien
2. Kebutuhan memori fleksibel

Struktur data dinamis diimplementasikan menggunakan tipe data pointer (penunjuk alamat memori).

Dengan demikian, struktur data dinamis akan memuat dua elemen, yaitu:

1. Nilai data
2. Penunjuk alamat berikutnya (link pointer)

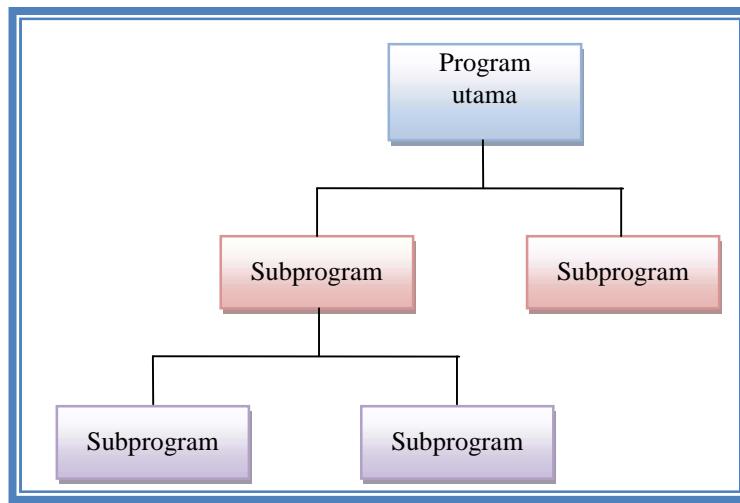
1.5. Proceduer dan Function

Suatu teknik yang biasa diterapkan dalam pemrograman terstruktur adalah teknik rancang atas-bawah (top-down design). Berdasarkan falsafah rancang atas-bawah, maka suatu program dipecah menjadi beberapa bagian yang lebih kecil, dengan harapan bagian-bagian yang kecil menjadi lebih mudah dikodekan dan juga menjadi lebih mudah dipahami oleh program. Bagian-bagian pemrograman seperti ini dikenal dengan sebutan subprogram atau subrutin.

Keuntungan lain dengan adanya subprogram adalah penghematan kode program. Ini terjadi jika ada beberapa bagian program yang sama dipanggil di beberapa tempat didalam program.

Untuk membuat subroutine Pascal menyediakan dua pilihan, yaitu procedure dan fungsi. Kedua jenis subroutine ini memiliki kegunaan yang sama, yaitu melakukan tugas tertentu.

Prosedur dan fungsi memungkinkan kita untuk menambahkan sekelompok statemen yang seolah-olah terpisah dari program utama tetapi sesungguhnya merupakan bagian dari program utama. Prosedur diaktifkan menggunakan statemen **procedure** (pemanggil prosedur) dan fungsi diaktifkan dengan suatu ungkapan yang hasilnya akan dikembalikan lagi sebagai nilai baru dari ungkapan tersebut.



Gambar 1.4. Program dibagi-bagi menjadi beberapa subprogram

Berikut contoh program menggunakan *procedure* dan *function* beserta cara pemanggilannya.

Contoh :

```

Program Prosedur;
Procedure tambah(a,b : integer; var c: integer); {Parameter acuan}
begin
  c:= a + b;
end;

{Program Utama}
var
  x,y,z : integer; {Parameter lokal}
begin
  clrscr;
  x:= 2; y:= 3;
  tambah(x,y,z);
  writeln('X = ',x,' Y = ',y,' Z = ',z);
end.
  
```

```

Program fungs1;
var a: real; {Parameter global}

function hasil(a : real) : real;
begin
  hasil := a*60;
end;

{Program utama}
begin
  clrscr;
  writeln('Masukkan jam : '); readln(a);
  writeln(a:5:2,' jam ada : ',hasil(a):8:2,' menit');
end.
  
```

1.6. Latihan

1. Buat struct untuk data buku yang berisi tentang : kode buku, nama buku, tahun terbit, pengarang, dan harga. Gunakan array of struct.
2. Buatlah fungsi untuk soal no 1, agar dapat dimanipulasi untuk ADD, EDIT, HAPUS, dan TAMPIL
3. Buatlah program data KTP, dengan menggunakan pointer. Kemudian buatkan fungsi atau prosedur untuk:
 - a) Menambah data
 - b) Mencari data berdasarkan tahun kelahiran tertentu
 - c) Menampilkan data berdasarkan L dan P
 - d) Mengedit data
4. Semua pengaksesan dilakukan dengan menggunakan pointer.

BAB II

SORTING

2.1. Pendahuluan

Data terkadang berada dalam bentuk yang tidak berpola ataupun dengan pola tertentu yang tidak kita inginkan, namun dalam penggunaanya, kita akan selalu ingin menggunakan data-data tersebut dalam bentuk yang rapi atau berpola sesuai dengan yang kita inginkan. Maka dari itu proses sorting adalah proses yang sangat penting dalam struktur data, terlebih untuk pengurutan data yang bertipe numerik ataupun *leksikografi* (urutan secara abjad sesuai kamus).

Sorting adalah proses menyusun kembali data yang sebelumnya telah disusun dengan suatu pola tertentu ataupun secara acak, sehingga menjadi tersusun secara teratur menurut aturan tertentu. Pada umumnya ada 2 macam pengurutan, yaitu: pengurutan secara *ascending* (urut naik) dan pengurutan secara *descending* (urut turun).

Banyak klasifikasi yang dapat digunakan untuk mengklasifikasikan algoritma-algoritma pengurutan, misalnya secara kompleksitas, teknik yang dilakukan, stabilitas, memori yang digunakan, rekursif atau tidak ataupun proses yang terjadi. Secara umum, metode pengurutan dapat dikelompokkan dalam 2 kategori yaitu

1. Metode pengurutan sederhana (*elementary sorting methods*)

Metode pengurutan sederhana meliputi *bubble sort*, *selection sort* dan *insertion sort*.

2. Pengurutan lanjut (*advanced sorting methods*).

Metode pengurutan lanjut diantaranya *shell sort*, *quick sort*, *merge sort* dan *radix sort*.

Algoritma-algoritma ini tentu saja akan mempunyai efek yang berbeda dalam setiap prosesnya, ada yang mudah digunakan, ada yang mempunyai proses yang sangat cepat. Pemilihan algoritma untuk sorting ini tidak hanya asal saja dipilih. Pemilihan ini semestinya berdasarkan kebutuhan yang diperlukan. Tidak semua algoritma yang pendek itu buruk dan tidak semua algoritma yang super cepat juga akan baik dalam semua kondisi. Misal: algoritma *quick sort* adalah algoritma sorting yang tercepat dalam proses pencarinya, namun jika data yang akan diurutkan ternyata sudah hampir terurut atau tidak terlalu banyak, maka algoritma ini malah akan memperlama proses pengurutan itu sendiri, karena akan banyak perulangan tidak perlu yang dilakukan dalam proses sorting ini.

Pada pembahasan sorting ini pengklasifikasian yang disajikan adalah menurut langkah yang terjadi dalam proses pengurutan tersebut. Pengklasifikasianya adalah sebagai berikut :

- *Exchange sort*
- *Selection sort*
- *Insertion sort*
- *Non-comparison sort*

2.2. Exchange Sort

Dalam prosesnya, algoritma-algoritma pengurutan yang diklasifikasikan sebagai *exchange sort* melakukan perbandingan antar data, dan melakukan pertukaran apabila urutan yang didapat belum sesuai. Contohnya adalah : *Bubble sort*, dan *Quicksort*.

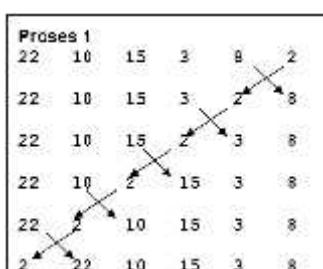
2.2.1. Bubble sort

Keuntungan dari algoritma sorting ini adalah karena paling mudah, dan dapat dijalankan dengan cukup cepat dan efisien untuk mengurutkan list yang urutannya sudah hampir benar. Namun algoritma ini paling lambat dan termasuk sangat tidak efisien untuk dilakukan dibandingkan dengan algoritma yang lain apalagi pengurutan dilakukan terhadap elemen yang banyak jumlahnya. Untuk itu biasanya *bubble sort* hanya digunakan untuk mengenalkan konsep dari algoritma sorting pada pendidikan komputer karena idenya yang cukup sederhana, yaitu mengurutkan data dengan cara membandingkan elemen sekarang dengan elemen berikutnya. Kompleksitas untuk algoritma ini adalah $O(n^2)$.

(1) Konsep algoritma bubble sort

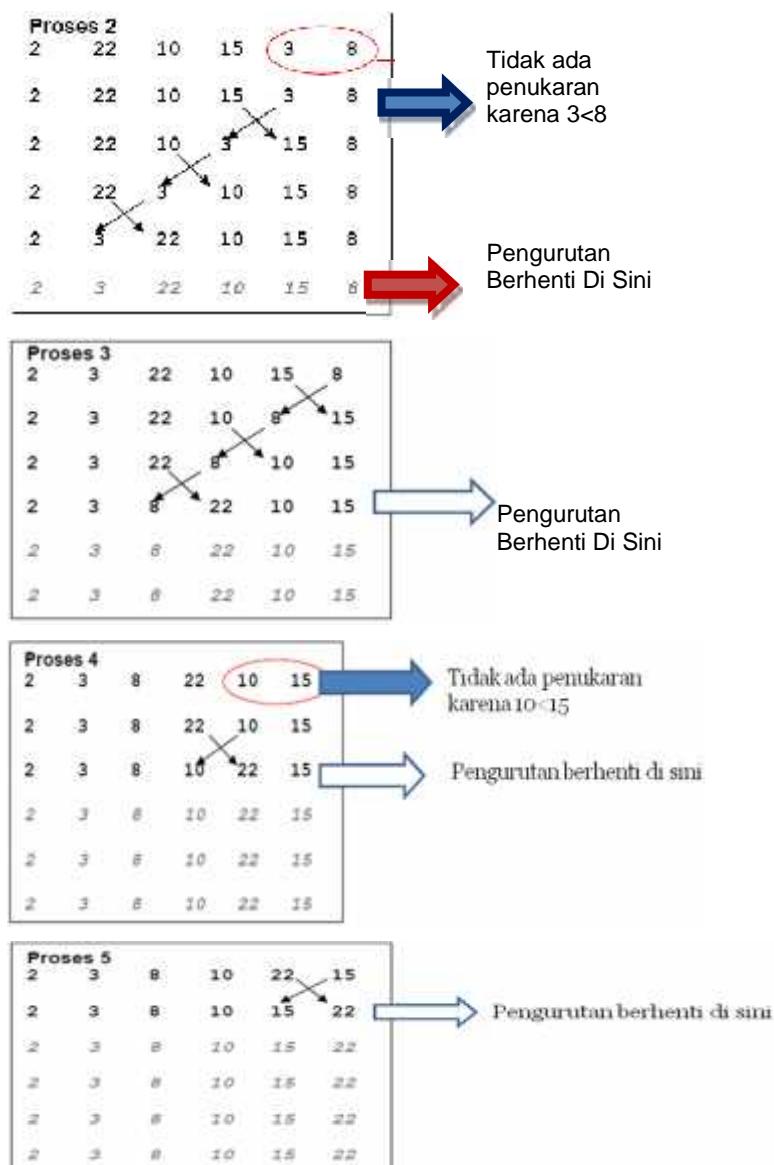
- Algoritma dimulai dari elemen paling awal.
- 2 buah elemen pertama dari list dibandingkan. Jika elemen pertama lebih besar dari elemen kedua atau sebaliknya (urut secara *ascending* atau *descending*), dilakukan pertukaran.
- Langkah 2 dan 3 dilakukan lagi terhadap elemen kedua dan ketiga, seterusnya sampai ke ujung elemen
- Bila sudah sampai ke ujung dilakukan lagi ke awal sampai tidak ada terjadi lagi pertukaran elemen.
- Bila tidak ada pertukaran elemen lagi, maka list elemen sudah terurut.
- Setiap pasangan data: $x[j]$ dengan $x[j+1]$, untuk semua $j=1,\dots,n-1$ harus memenuhi keterurutan, yaitu untuk pengurutan:
 - o Ascending : $x[j] < x[j+1]$
 - o Descending : $x[j] > x[j+1]$

(2) Implementasi bubble sort



Gambar 2.1. Ilustrasi algoritma bubble sort untuk pengurutan secara ascending

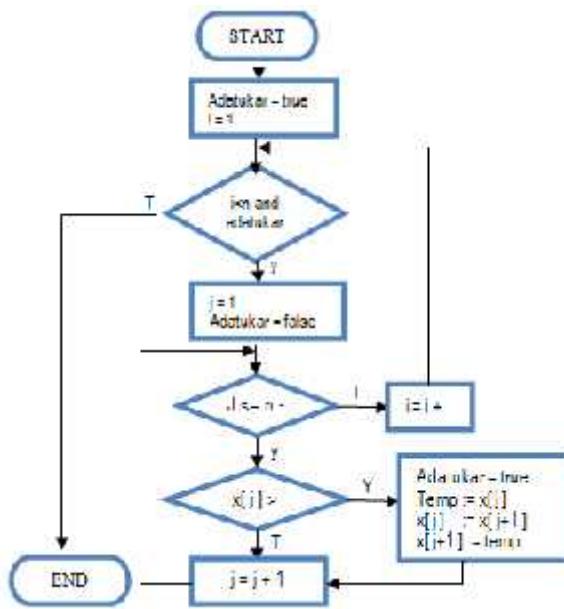
- Pada gambar 2.1, pengecekan dimulai dari data yang paling akhir, kemudian dibandingkan dengan data di depannya, jika data di depannya lebih besar maka akan ditukar
- Pada proses kedua, proses pertama dilanjut dan pengecekan dilakukan sampai dengan data ke-2 karena data pertama pasti sudah paling kecil.
- Proses ketiga dilakukan dengan cara yang sama dengan proses pertama atau proses kedua sampai data sudah dalam kondisi terurut. Proses iterasi maksimal dilakukan sampai dengan $n-1$, dengan n adalah jumlah data yang akan diurutkan.



Gambar 2.2. Ilustrasi algoritma bubble sort pada proses 3 s.d 5

(3) Flowchart algoritma bubble sort

Flowchart procedure dari algoritma *bubble sort* dapat dilihat pada gambar 2.3



Gambar 2.3. Flowchart dari algoritma bubble sort

(4) Procedure bubble sort

```

Type
  Typearray = array[1..100] of integer;
Var
  Adatukar : boolean;
  i, j,temp : integer;

Procedure buble_sort(var x:typearray; n : integer);
begin
  adatukar := true;
  i := 1;
  while ( i<n ) and (adatukar) do
  begin
    j:=1;
    adatukar := false;
    while j <= (n - i) do
    begin
      If x[ j ] > x[ j+1 ] then
      begin
        adatukar := true ;
        temp := x[ j ];
        x[ j ] := x[ j+1 ];
        x[ j+1]:= temp;
      end;
      j := j+1 ;
    end
    i := i+1;
  end;
end;
  
```

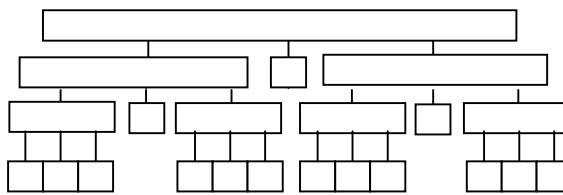
2.2.2. Quick sort

Algoritma Quicksort bersifat *divide and conquer*. Quick sort banyak digunakan untuk proses *sorting*, karena:

- Merupakan proses *sorting* yang umum digunakan
- Mudah untuk diimplementasikan
- Algoritma ini merupakan algoritma pencarian yang sangat cepat (saat ini tercepat).

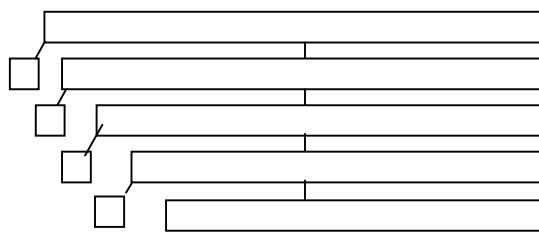
Kelemahan dari algoritma *quicksort* adalah apabila algoritma ini digunakan untuk mengurutkan elemen-elemen yang hanya ada sedikit atau sudah hampir terurut, karena jika menggunakan algoritma ini justru akan melakukan perulangan yang tidak berguna dan lama. Selain itu algoritma ini mempunyai algoritma dan program yang cukup kompleks.

Kompleksitas waktu untuk algoritma quick sort untuk kondisi *best case* seperti terlihat pada gambar 2.4. apabila tiap subproblem berukuran $n/2$ dengan partisi dalam tiap subproblem adalah linier. Jadi total usaha dalam mempartisi 2^k problem dengan ukuran $n/2^k$ adalah $O(n)$. Total partisi di tiap level adalah $O(n)$ dan memerlukan $\log n$ level dari partisi terbaik untuk memperoleh sebuah elemen subproblem. Sehingga kompleksitas waktu untuk *best case* adalah $O(n \cdot \log n)$.



Gambar 2.4. Pohon rekursif untuk *best case* pada algoritma quick sort

Kompleksitas waktu untuk algoritma *quick sort* untuk kondisi *worst case* seperti terlihat pada Gambar 2.5. terjadi jika elemen pivot membagi array tidak sama dengan $n/2$ elemen pada separuh bagian terkecil, maka diperoleh elemen pivot adalah elemen terbesar atau elemen terkecil dalam array. Diperoleh $n-1$ level dari $\log n$, semenjak $n/2$ level pertama masing-masing mempunyai $\geq n/2$ elemen untuk partisi, sehingga waktu untuk *worst case* adalah $O(n^2)$



Gambar 2.5. Pohon rekursif untuk *worst case* pada algoritma quick sort

Kompleksitas waktu untuk algoritma *quick sort* untuk kondisi *average case* seperti terlihat pada Gambar 2.6. terjadi bila elemen pivot dipilih secara acak dalam array dengan n elemen, suatu saat elemen pivot akan berasal dari bagian tengah array yang terurut.

Bila elemen pivot dari posisi $n/4$ ke $3n/4$. maka subarray maksimal mengandung $3n/4$ elemen. Jika diasumsikan bahwa elemen pivot selalu dalam range ini, maka partisi maksimum yang dibutuhkan untuk n elemen turun ke 1 elemen :

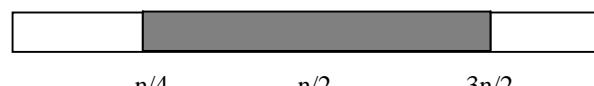
$$\left(\frac{3}{4}\right)^i \cdot n = 1$$

$$n = \left(\frac{4}{3}\right)^i$$

$$\log n = i \cdot \log\left(\frac{4}{3}\right)$$

$$i = \log\left(\frac{4}{3}\right) \cdot \log(n) < 2 \cdot \log n$$

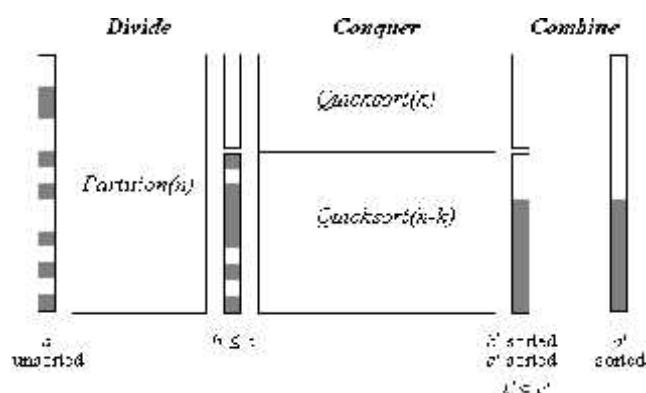
Jika membutuhkan $2 \cdot \log n$ level dari partisi yang baik untuk menyelesaikan permasalahan dan separuh dari partisi acak adalah baik, maka rata-rata pohon rekursif untuk *quick sort* array adalah $\approx 4 \cdot \log n$ level.



Gambar 2.6. Pohon rekursif untuk average case pada algoritma quick sort

(1) Konsep algoritma quick sort

Pertama-tama deret dibagi menjadi dua bagian, misal, semua elemen pada bagian b (bagian pertama) mempunyai kurang dari atau sama dengan semua elemen pada bagian c (bagian kedua). Kemudian kedua bagian tersebut dilakukan proses sorting dengan rekursif secara terpisah dengan prosedur yang sama(*conquer*). Kemudian gabungkan lagi kedua bagian terpisah tersebut.

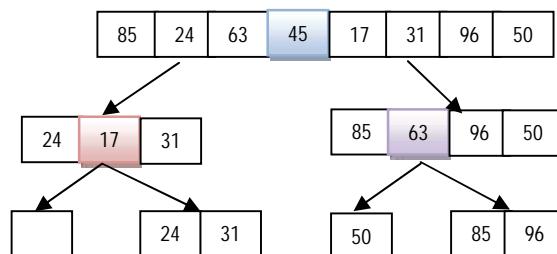


Gambar 2.7. Ilustrasi algoritma quick Sort

- *Select* : Pertama kita pilih elemen yang ditengah sebagai pivot, misalkan X.
- *Partition* : kemudian semua elemen tersebut disusun dengan menempatkan X pada posisi j sedemikian rupa sehingga elemen disebelah kiri1 lebih $< X$ dan elemen sebelah kanan $> X$.
- *Rekursif* : kemudian proses diulang untuk bagian kiri dan kanan elemen X dengan cara yg sama dengan langkah 1 sampai kondisi terurut

(2) Implementasi algoritma quick sort

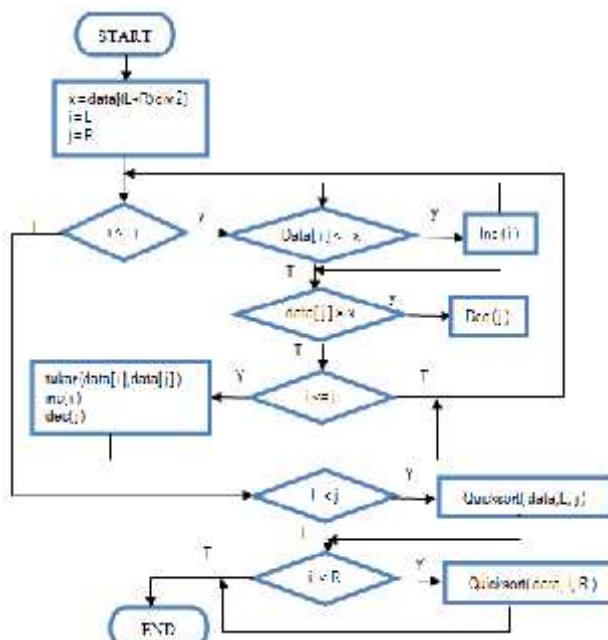
- Diketahui larik (85, 24, 63, 45, 17, 31, 96, 50).
- Pilih 45 sebagai **pivot** dan bagi elemen-elemen kedalam 3 larik: (24, 17, 31), (45), and (85, 63, 96, 50). Karena ukuran larik lebih kecil, diharapkan proses pengurutan menjadi lebih mudah.
- Sekarang kita ambil larik pertama (24, 17, 31) dan tampak bahwa larik tersebut belum terurut, maka kita pilih 17 sebagai pivot dan kita pecah larik ke dalam 2 larik lagi (17) , dan (24, 31). Sekarang seluruh larik-larik kecil sudah terurut.
- Kita kumpulkan kembali larik-larik kecil tadi. Hasilnya adalah larik (17), (24, 31), dan (45) bersama. Hasilnya adalah (17, 24, 31, 45).
- Dengan cara serupa, larik (85, 63, 96, 50) diurutkan. Maka kita pilih 63 sebagai pivot dan kita pecah larik ke dalam 3 larik lagi (50),(63) dan (85, 96). Hasilnya adalah (50, 63, 85, 96).
- Selanjutnya kita kumpulkan larik (17, 24, 31, 45), dan (50, 63, 85, 96) bersama sehingga mendapatkan larik (17, 24, 31, 45, 50, 63, 85, 96).



Gambar 2.8. Ilustrasi algoritma quick sort

(3) Flowchart algoritma quick sort

Flowchart procedure dari algoritma *quick sort* dapat dilihat pada gambar 2.9.



Gambar 2.9. Flowchart algoritma quick sort

(4) Procedure quick sort

```

Procedure quicksort(data, L,R : integer);
Var i,j,x : integer;
Begin
    x = data[(L+R) div 2];
    i = L;
    j = R;
    while ( i <= j ) do
    begin
        while (data[ i ] < x ) do inc( i );
        while ( data[J] > x ) do dec( J );
        If ( i <= J ) then
            tukar(data[i],data[j]);
            inc( I );
            Dec( J );
        End;
    End;
    If ( L < J ) then quicksort(data,L, J );
    If ( I < R ) then quicksort(data,i, R );
End;

```

2.3. Selection Sort

Prinsip utama algoritma dalam klasifikasi ini adalah mencari elemen yang tepat untuk diletakkan di posisi yang telah diketahui, dan meletakkannya di posisi tersebut setelah data tersebut ditemukan. Algoritma yang dapat diklasifikasikan ke dalam kategori ini adalah : *Selection sort, dan Heapsort.*

2.3.1 Selection sort

Selection Sort merupakan kombinasi antara *sorting* dan *searching*. Untuk setiap proses, akan dicari elemen-elemen yang belum diurutkan yang memiliki nilai terkecil atau terbesar akan dipertukarkan ke posisi yang tepat di dalam array.

Kelebihan dan kekurangan Selection Sort:

1. Kompleksitas selection sort relatif lebih kecil
2. Mudah menggabungkannya kembali, tetapi sulit membagi masalah
3. Membutuhkan metode tambahan

(1) Konsep algoritma selection sort

Untuk putaran pertama, akan dicari data dengan nilai terkecil dan data ini akan ditempatkan di indeks terkecil (data[1]), pada putaran kedua akan dicari data kedua terkecil, dan akan ditempatkan di indeks kedua (data[2]). Selama proses, pembandingan dan pengubahan hanya dilakukan pada indeks pembanding saja, pertukaran data secara fisik terjadi pada akhir proses.

Tehnik pengurutan dgn cara pemilihan elemen atau proses kerja dgn memilih elemen data terkecil untuk kemudian dibandingkan & ditukarkan dgn elemen pada data awal, dan seterusnya sampai dengan seluruh elemen sehingga akan menghasilkan pola data yg telah disort.

(2) Implementasi algoritma selection sort

- Bila diketahui data sebelum dilakukan proses sortir

85	63	24	45	17	31	96	50
----	----	----	----	----	----	----	----

- Iterasi 1:

85	63	24	45	17	31	96	50
----	----	----	----	----	----	----	----

85	63	24	45	17	31	96	50
----	----	----	----	----	----	----	----

17	63	24	45	85	31	96	50
----	----	----	----	----	----	----	----

- Iterasi 2:

17	63	24	45	85	31	96	50
----	----	----	----	----	----	----	----

17	24	63	45	85	31	96	50
----	----	----	----	----	----	----	----

- Iterasi 3:

17	24	63	45	85	31	96	50
----	----	----	----	----	----	----	----

17	24	31	45	85	63	96	50
----	----	----	----	----	----	----	----

- Iterasi 4:

Pada iterasi ke empat karena data terkecil adalah data ke-4, maka tidak dilakukan proses pertukaran

17	24	31	45	85	63	96	50
----	----	----	----	----	----	----	----

- Iterasi 5:

17	24	31	45	85	63	96	50
----	----	----	----	----	----	----	----

17	24	31	45	50	63	96	85
----	----	----	----	----	----	----	----

- Iterasi 6:

Pada iterasi ke enam karena data terkecil adalah data ke-6, maka tidak dilakukan proses pertukaran seperti pada iterasi ke-4

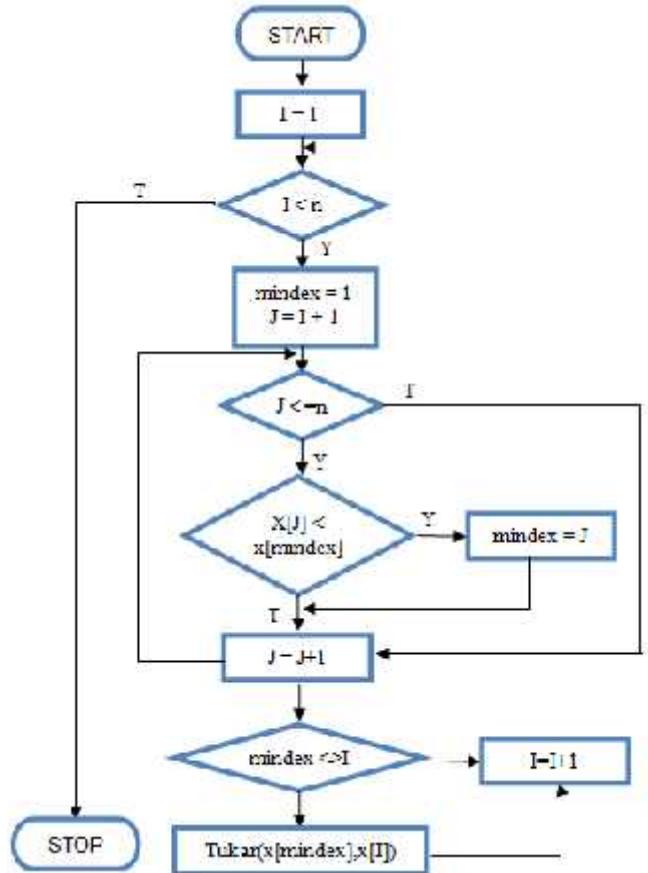
17	24	31	45	50	63	96	85
----	----	----	----	----	----	----	----

- Iterasi 7:

17	24	31	45	50	63	96	85
----	----	----	----	----	----	----	----

17	24	31	45	50	63	85	96
----	----	----	----	----	----	----	----

(3) Flowchart algoritma selection sort



Gambar 2.10. Flowchart algoritma selection sort

(4) Procedure selection sort

```

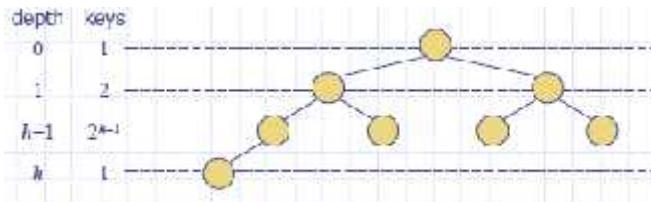
Procedure selection_sort(var x: typearray; n : integer);
Var I, J, mindex : integer;
Begin
  I := 1;
  While I < n do
    Begin
      Mindex := I;
      J := J+1;
      While J <= n do
        Begin
          If x[ J ] < x[ mindex ] then
            Mindex := J;
          J := J+1;
        End;
        If mindex <> I then
          Tukar(x[ I ], x[ mindex ]);
        I := I+1;
      End;
    End;
End;
  
```

2.3.2. Heap sort

(1) Aturan Heap sort

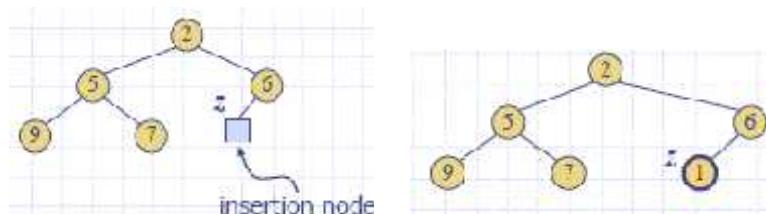
Heap sort adalah *binary tree* dengan menggunakan kunci, dimana mempunyai aturan-aturan sebagai berikut :

- Heap* dibuat dengan cara mengisikan data ke *heap* dimulai dari level 1 sampai ke level dibawahnya, bila dalam level yang sama semua kunci *heap* belum terisi maka tidak boleh mengisi dibawahnya.



Gambar 2.11. Urutan pengisian data pada heap

- Penambahan kunci diletakkan pada posisi terakhir dari level dan disebelah kanan *child* yg terakhir.



Gambar 2.12. Proses insert heap

- Urutkan *heap* dengan aturan *left child > parent* dan *right child > parent* bila akan dilakukan pengurutan secara *descending*. Bila akan diurutkan secara *ascending* maka urutkan *heap* dengan aturan *left child < parent* dan *right child < parent*.

Pada *heap sort* dikenal 2 proses, yaitu metode *down heap* dan *upheap*.

1) Metode Upheap

- bandingkan kunci terakhir dengan *parentnya* apabila *parent > kunci* maka lakukan pertukaran.
- ulangi langkah 1 dengan membandingkan dengan *parent* selanjutnya sampai posisi *parent* di level 1 selesai dibandingkan



Gambar 2.13. Pengurutan menggunakan metode upheap

2) Metode Downheap:

- bandingkan *parent* dengan *leftchild* dan *rightchild* apabila *parent* > *leftchild* atau *rightchild* maka lakukan pertukaran.
- ulangi langkah 1 dengan membandingkan dengan *leftchild* dan *rightchild* pada posisi level dibawahnya sampai posisi di level terakhir selesai dibandingkan.



Gambar 2.14. Pengurutan menggunakan metode downheap

- Delete *heap* dengan cara mengambil isi *heap* dari root (mengambil isi *heap* pada parent di level 1) kemudian dimasukkan ke dalam array pada posisi ke-n selanjutnya posisi parent digantikan posisi kunci terakhir, dan dilakukan sort kembali dengan metode *downheap*.
- Ulangi langkah 3 sampai note pada binary tree kosong. Data yang tersimpan dalam array merupakan data yang telah dalam kondisi terurut turun (*descending*)

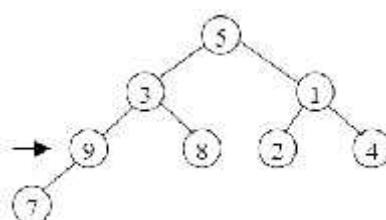
Heap sort mempunyai kompleksitas waktu $O(n \log n)$ untuk semua kasus baik *word case* maupun *base case*. Berbeda dengan *quicksort* dimana secara umum mempunyai kompleksitas $O(n \log n)$ namun untuk kasus *word case* kompleksitas waktunya perlahan menjadi $O(n^2)$. Selain itu *heap sort* mempunyai kritikal waktu aplikasi yang lebih baik, meskipun *quicksort* merupakan algoritma yang paling cepat.

(2) Implementasi Heap sort dengan metode ascending

- Pertama yang kita lakukan pada heap adalah mengkonvert ke dalam array sebagai contoh diketahui $n = 8$

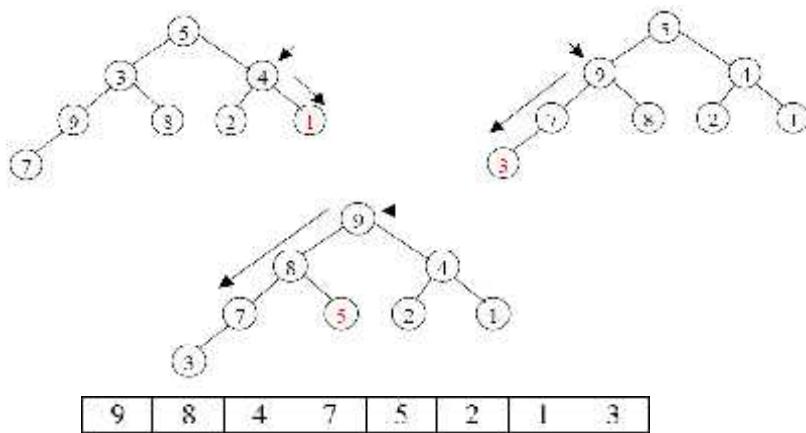
5	3	1	9	8	2	4	7
---	---	---	---	---	---	---	---

Kemudian masukkan data ke dalam binary tree dengan mengisikan dari level 1 ke level berikutnya seperti terlihat pada gambar 2.15

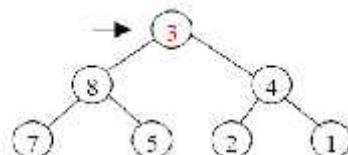


Gambar 2.15. Proses 1 dari proses heap sort

- Kemudian lakukan pengurutan menggunakan metode *up heap* agar pohon biner sesuai dengan aturan *heap sort* yaitu *parent* harus lebih besar daripada *child*

**Gambar 2.16.** Hasil heap sort pada iterasi pertama

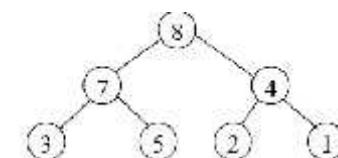
- c. Kemudian lakukan *delete heap*



3	8	4	7	5	2	1	9
---	---	---	---	---	---	---	---

Gambar 2.17. Proses *delete heap*

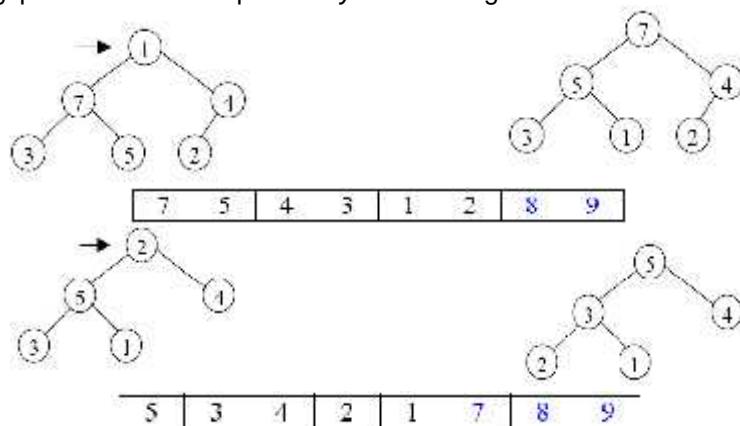
- d. Ulangi proses pengurutan menggunakan metode down heap agar pohon biner sesuai dengan aturan heap sort yaitu bahwa parent harus lebih besar daripada child



8	7	4	3	5	2	1	9
---	---	---	---	---	---	---	---

Gambar 2.18. Heap sort pada iterasi ke dua

- e. Ulangi proses diatas sampai *binary tree* kosong

**Gambar 2.19.** Heap Sort pada iterasi ke tiga

2.4. Insertion Sort

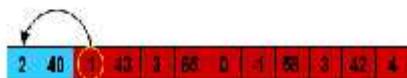
Algoritma pengurutan yang diklasifikasikan kedalam kategori ini mencari tempat yang tepat untuk suatu elemen data yang telah diketahui kedalam subkumpulan data yang telah terurut, kemudian melakukan penyisipan (*insertion*) data di tempat yang tepat tersebut.

(1) Konsep algoritma insertion sort

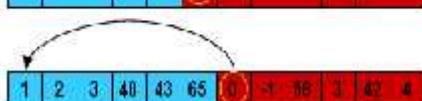
Prinsip dasar *Insertion* adalah secara berulang-ulang menyisipkan/ memasukan setiap elemen ke dalam posisinya/tempatnya yang benar. Mirip dengan cara orang **mengurutkan** kartu, selembar demi selembar kartu diambil dan **disisipkan** (*insert*) ke tempat yang seharusnya. Pengurutan dimulai dari data ke-2 sampai dengan data terakhir, jika ditemukan data yang **lebih kecil**, maka akan ditempatkan (**diinsert**) diposisi yang seharusnya. Pada penyisipan elemen, maka elemen-elemen lain akan bergeser ke belakang

(2) Implementasi Insertion sort

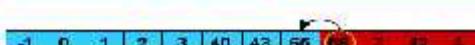
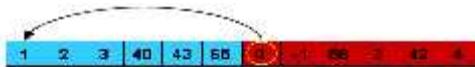
- Kondisi awal:
 - Unsorted list* = data
 - Sorted list* = kosong
- Ambil sembarang elemen dari *unsorted list*, sisipkan (*insert*) pada posisi yang benar dalam *sorted list*. Lakukan terus sampai *unsorted list* habis.
 - Langkah 1:**



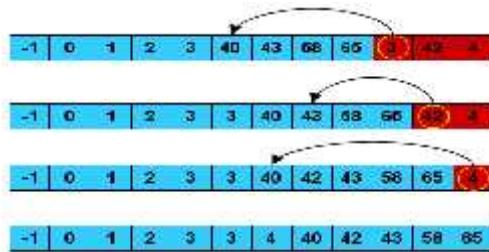
- Langkah 2:**



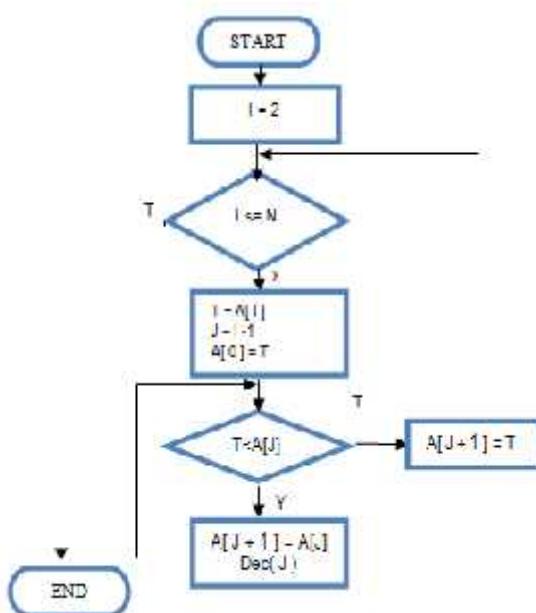
- Langkah 3:**



- Langkah 4:



(3) Flowchart algoritma insertion sort



Gambar 2.20. Flowchart algoritma insertion sort

(4) Procedure Insertion sort

```

Procedure sisip_langsung (var A : larik; N : integer);
Var i, j : integer; T : real;
Begin
  For I := 2 to N do begin
    T := A[ i ];
    J := I - 1;
    A[0] := T;
    While T < A[ J ] do Begin
      A[ J + 1 ] := A[ J ];
      Dec ( J );
    End;
    A[ J + 1 ] := T;
  End;
End;
  
```

Gambar 2.21. Program procedure algoritma insertion sort

2.5. Non comparison sort

Non comparison sort adalah algoritma pengurutan di mana dalam prosesnya tidak melakukan pembandingan antar data. Secara umum proses yang dilakukan dalam algoritma ini adalah mengklasifikasikan data sesuai dengan kategori terurut yang tertentu, dan dalam tiap kategori dilakukan pengklasifikasian lagi, dan seterusnya sesuai dengan kebutuhan, kemudian subkategori-subkategori tersebut digabungkan kembali, yang dilakukan dengan algoritma sederhana *concatenation*.

Dalam kenyataanya seringkali algoritma *non-comparison sort* yang digunakan tidak murni tanpa pembandingan, yang dilakukan dengan menggunakan algoritma pengurutan cepat lainnya untuk menurunkan subkumpulan-subkumpulan datanya.

Secara kompleksitas, dalam berbagai kasus tertentu, algoritma tanpa pembandingan ini dapat bekerja dalam waktu linier, atau dengan kata lain memiliki kompleksitas $O(n)$. Salah satu kelemahan dari algoritma ini adalah selalu diperlukannya memori tambahan.

Salah satu algoritma sorting non comparison sort adalah algoritma *Radix sort*.

(1) Konsep algoritma radix sort

Ide dasar dari algoritma *Radix sort* ini adalah mengkategorikan data-data menjadi sub kumpulan subkumpulan data sesuai dengan nilai *radix*-nya, mengkonkatennasinya, kemudian mengkategorikannya kembali berdasar nilai *radix* lainnya.

(2) Implementasi algoritma

Contohnya adalah pengurutan sebuah kumpulan data bilangan bulat dengan jumlah digit maksimal 3

121	076	823	367	232	434	742	936	274
-----	-----	-----	-----	-----	-----	-----	-----	-----

Langkahnya adalah :

- a. Pertama kali, data dibagi-bagi sesuai dengan digit terkanan :

- b. Hasil pengkategorian tersebut lalu digabung kembali dengan algoritma konkatenasi menjadi:

121	232	742	823	434	274	076	936	367
-----	-----	-----	-----	-----	-----	-----	-----	-----

- c. Kemudian pengkategorian dilakukan kembali, namun kali ini berdasar digit kedua atau digit tengah, dan jangan lupa bahwa urutan pada tiap subkumpulan data harus sesuai dengan urutan kemunculan pada kumpulan data

121	232	742	823	434	274	076	936	367
-----	-----	-----	-----	-----	-----	-----	-----	-----

Kategori digit	Isi
0	-
1	-
2	121, 823
3	232, 434, 936
4	742
5	-
6	367
7	274, 076
8	-
9	-

- d. Yang kemudian dikonkatenasi kembali menjadi

121	823	232	434	936	742	367	274	076
-----	-----	-----	-----	-----	-----	-----	-----	-----

- e. Kemudian langkah ketiga, atau langkah terakhir pada contoh ini adalah pengkategorian kembali berdasar digit yang terkiri, atau yang paling signifikan

121	823	232	434	936	742	367	274	076
Kategori digit	Isi							
0	076							
1	121							
2	232, 274							
3	367							
4	434							
5	-							
6	-							
7	742							
8	823							
9	936							

Yang kemudian dikonkatenasi lagi menjadi hasil akhir dari metode pengurutan ini.

076	121	232	274	367	434	742	823	936
-----	-----	-----	-----	-----	-----	-----	-----	-----

(3) Proceduer algoritma radix sort

Implementasi dari algoritma pada Gambar 2.25 dapat direalisasikan dengan menggunakan *queue* sebagai representasi tiap kategori *radix* untuk pengkategorian. *Array A* adalah *array input*, dan *array B* adalah *array A* yang sudah terurut.

```

Procedure RadixSort (A : TArray; var B : TArray; d : byte);
var
  KatRadix : array [0..9] of Queue;
  i, x, ctr : integer;
  pembagi : longword;
begin
  {--- mengkopi A ke B ---}
  for i:=1 to n do
    B[i] := A[i];
  pembagi := 1;
  for x:=1 to d do begin
    {--- inisialisasi KatRadix ---}
    for i:=0 to 9 do
      InitQueue (KatRadix[i]);
    {--- dikategorikan ---}
    for i:=1 to n do
      Enqueue (KatRadix [(B[i] div pembagi) mod 10], B[i]);
    B[i] := 0;
    {--- dikonkat ---}
    ctr := 0;
    for i:=0 to 9 do begin
      while (NOT IsQueueEmpty (KatRadix[i])) do begin
        ctr := ctr + 1;
        B[ctr]:=DeQueue (KatRadix [i]);
      end;
    end;
    pembagi := pembagi * 10;
  end;
end;

```

2.6. Latihan

- Carilah metode-metode sort yang belum dibahas pada pokok bahasan diatas. Kemudian buatlah perbandingan antara metode yang anda temukan dengan yang metode yang telah dipelajari pada buku ini.
- gambarkan ilustrasi sorting menggunakan algoritma yang dibahas pada buku ini bila diketahui data yang dimasukkan adalah :

60 30 35 65 55 20 45 85 70 80 10

BAB III

LINKED LIST

Linked list dikembangkan tahun 1955-1956 oleh Allen Newell, Cliff Shaw & Herbert Simon di RAND Corporation sebagai struktur data utama untuk bahasa *information processing language* (IPL). IPL adalah pengembangan dari program AI dengan bahasa pemrograman COMMIT. *Linked list* merupakan struktur data dinamis yang paling sederhana yang berlawanan dengan *array*, yang merupakan struktur statis.

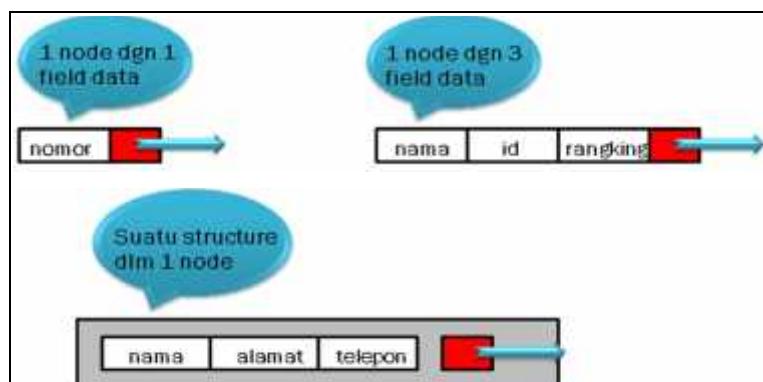
Linked list adalah koleksi dari obyek-obyek homogen dengan sifat setiap obyek (kecuali terakhir) punya penerus dan setiap obyek (kecuali yang pertama) punya pendahulu. Masing-masing data dalam *Linked list* disebut dengan *node* (simpul) yang menempati alokasi memori secara dinamis dan biasanya berupa *struct* yang terdiri dari beberapa *field*. Kumpulan komponen obyek-obyek ini disusun secara berurutan dengan bantuan pointer. Penggunaan pointer untuk mengacu elemen berakibat elemen-elemen bersebelahan secara logik walau tidak bersebelahan secara fisik di memori. Masing-masing *linked list* terbagi menjadi 2 bagian :

1. Medan informasi

Berisi informasi yang akan disimpan dan diolah.

2. Medan penyambung

Berisi alamat simpul berikutnya



Gambar 3.1. Bagian linked list

Operasi pada *Linked list*

1. Menambah simpul

Bisa dipecahkan berdasarkan posisi simpul yaitu simpul baru :

- ditambahkan dibelakang simpul terakhir
- selalu diletakkan sebagai simpul pertama
- menyisip diantara dua simpul yang sudah ada.

2. Menghapus simpul

Dapat dilakukan dengan menghapus didepan, dibelakang atau ditengah dan harus berada sesudah simpul yang ditunjuk oleh suatu pointer.

3. Mencari Informasi pada suatu *Linked list*

Sama dengan membaca isi simpul hanya ditambah dengan test untuk menentukan ada tidaknya data yang dicari

4. Mencetak Simpul

Dengan cara membaca secara maju dan secara mundur

3.1. Single Linked List

Linked list dapat diilustrasikan seperti satu kesatuan rangkaian kereta api. Kereta api terdiri dari beberapa gerbong, masing-masing dari gerbong itulah yang disebut node/tipe data bentukan. Agar gerbong-gerbong tersebut dapat saling bertaut dibutuhkan minimal sebuah kait yang dinamakan sebagai pointer. Setelah mendeklarasikan tipe data dan pointer pada list, selanjutnya kita akan mencoba membuat senarai / *Linked list* tunggal tidak berputar atau sebuah gerbong.

Single *Linked list* adalah senarai berkait yang masing-masing simpul pembentuknya mempunyai satu kait (link) ke simpul lainnya. Pembentukan *linked list* tunggal memerlukan :

1. deklarasi tipe simpul
2. deklarasi variabel pointer penunjuk awal *Linked list*
3. pembentukan simpul baru
4. pengaitan simpul baru ke *Linked list* yang telah terbentuk

Ada beberapa operasi yang dapat kita buat pada senarai tersebut, diantaranya: tambah, hapus dan edit dari gerbong tersebut.



Gambar 3.2. Ilustrasi Single Linked List

Gambar 3.2. mengilustrasikan sebuah rangkaian kereta api dengan 4 buah gerbong. Gerbong A akan disebut sebagai kepala / head (walaupun penamaan ini bebas) dan gerbong D adalah ekor / tail. Tanda panah merupakan kait atau pointer yang menghubungkan satu gerbong dengan yang lainnya.

Pointer yang dimiliki D menuju ke NULL, inilah yang membedakan antara senarai berputar dengan yang tidak berputar. Kalau senarai berputar maka pointer dari D akan menuju ke A lagi.

3.1.1. Deklarasi single linked list

Manipulasi *Linked list* tidak bisa dilakukan langsung ke node yang dituju, melainkan harus menggunakan suatu pointer penunjuk ke node pertama dalam *Linked list* (dalam hal ini adalah *head/kepala*).

Bentuk umum :

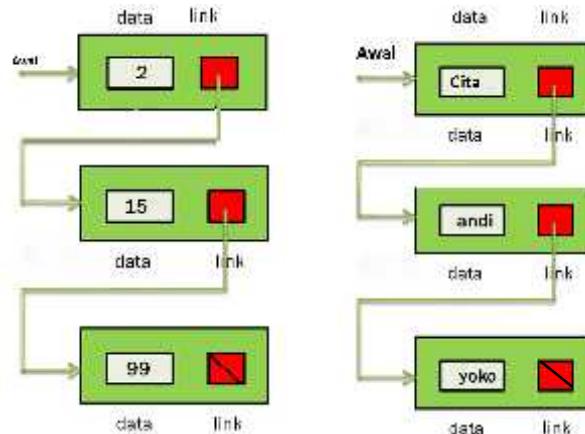
```
Type typeinfo = .....;
simpul = ^typenode;
Typenode = record
    Info : typeinfo;
    Next : simpul;
End;
Var elemen : typeinfo;
Awal, akhir, baru : simpul;
```

Dimana :

Info : sebuah tipe terdefinisi yang menyimpan informasi sebuah simpul
 Next : alamat dari elemen berikutnya (suksesor)

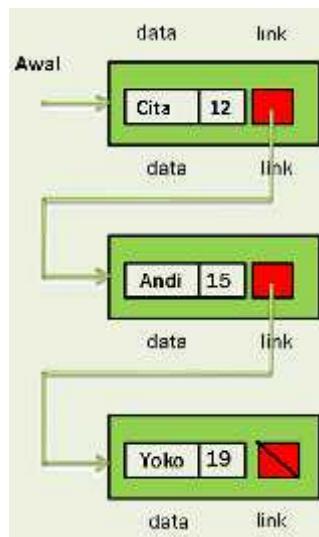
(1) Contoh deklarasi type single linked list dengan 1 node berisi 1 field data

```
Type typeinfo = .....;
simpul = ^typenode;
Typenode = record
    data : typeinfo;
    link : simpul;
End;
Var
Awal : simpul;
```



Gambar 3.3. Linked list dengan median informasi berisi 1 field data

(2) Contoh deklarasi single linked list dengan 1 node berisi 2 field data



Gambar 3.4. Linked list dengan median informasi berisi 2 field

Type

```

simpul = ^typenode;
Typenode = record
    nama : string;
    usia : integer;
    link : simpul;
End;

```

Var

```
Awal : simpul;
```

3.1.2. Operasi pada single linked list

Ada sejumlah operasi yang bisa kita lakukan pada sebuah *Linked list*, yaitu operasi menambah simpul dan menghapus simpul pada sebuah *Linked list*.

3.1.2.1. Menambah simpul

Operasi menambah simpul bisa dipecah berdasarkan posisi simpul baru yang akan disisipkan, yaitu simpul baru selalu ditambahkan dibelakang simpul terakhir, simpul baru selalu diletakkan sebagai simpul pertama, dan simpul baru menyisip diantara dua simpul yang ada.

(1) Menambah simpul di belakang

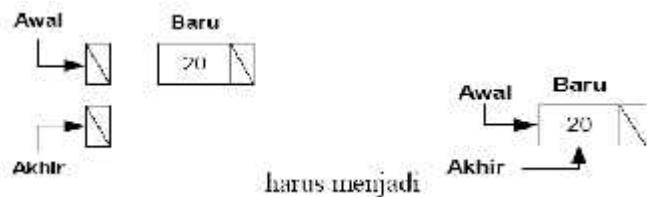
Penambahan simpul di belakang adalah proses penambahan data baru dimana data baru disimpan di posisi terakhir. Setelah proses penambahan selesai, maka variable akhir akan menunjuk ke data baru tersebut.

Ada 2 kondisi yang harus diperiksa yaitu kondisi penambahan akhir pada *linked list* yang masih kosong dan kondisi penambahan akhir pada *linked list* yang sudah mempunyai elemen.

a. Menambah simpul dibelakang ketika linked list masih kosong

Fungsi menambah simpul di belakang, apabila *linked list* masih kosong. Langkah-langkahnya:

- 1) membuat simpul baru kemudian diisi info baru.
- 2) simpul paling akhir dihubungkan ke simpul baru.
- 3) penunjuk pointer akhir dan pointer awal diarahkan ke simpul baru.

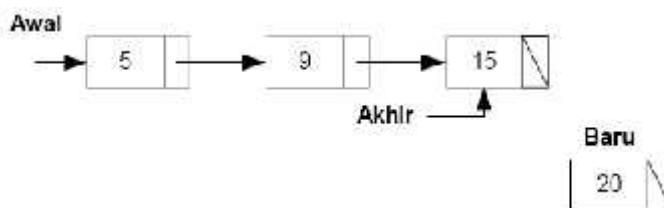


Gambar 3.5. Ilustrasi penambahan simpul dibelakang pada saat linked list masih kosong

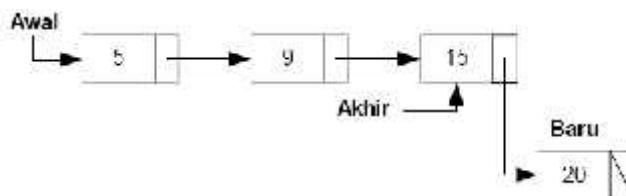
b. Penambahan akhir ketika linked list sudah mempunyai elemen.

Fungsi menambah simpul di belakang, langkah-langkahnya:

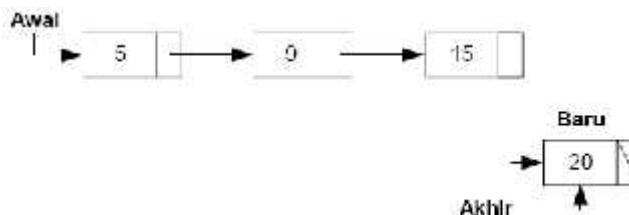
- 1) membuat simpul baru kemudian diisi info baru.



- 2) Setelah elemen baru dibuat, maka sambungkan field *next* milik pointer terakhir *Linked list* ke pointer baru.



- 3) Kemudian variable/pointer akhir dipindahkan ke pointer baru.



c. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```

Type Point = ^Data ;
  Data = record
    Info : integer ;
    next : point;
  End;
Var Elemen : integer ;
Awal, Akhir : point ;

Procedure Tambah_Belakang(var Awal,Akhir: point; Elemen: integer);
Var Baru : point;
Begin
  New(Baru);
  Baru^.Info := elemen;
  If Awal = NIL then
    Awal:= Baru
  Else
    Akhir^.next := Baru;
  Akhir:= Baru;
  Akhir^.next:= NIL;
End;

```

(2) Menambah simpul di awal

Penambahan elemen di posisi awal adalah menambahkan data baru pada posisi awal, sehingga data baru tersebut akan menjadi awal. Ada 2 kondisi yang harus diperhatikan ketika kita melakukan proses penambahan elemen baru di awal yaitu kondisi *Linked list* sedang kosong dan kondisi *Linked list* sudah mempunyai elemen.

a. Penambahan elemen di awal ketika linked list masih kosong

Fungsi menambah simpul di awal, apabila *Linked list* masih kosong. Langkah-langkahnya:

- 1) membuat simpul baru, kemudian diisi info baru.
- 2) simpul baru dihubungkan ke simpul awal.
- 3) penunjuk pointer awal diarahkan ke simpul baru.

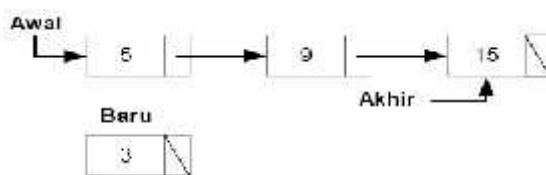


Gambar 3.6. Ilustrasi penambahan simpul di awal pada saat linked list masih kosong

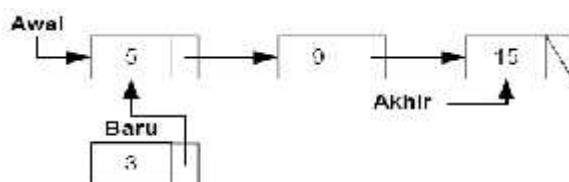
b. Penambahan di awal ketika linked list sudah mempunyai elemen.

Fungsi menambah simpul di belakang, langkah-langkahnya:

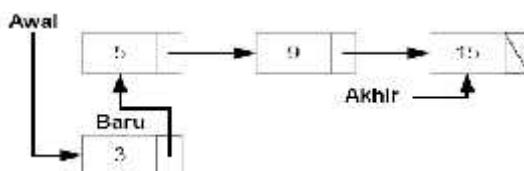
- 1) membuat simpul baru kemudian diisi info baru.



- 2) Setelah elemen baru dibuat, maka sambungkan field next milik pointer baru ke pointer awal.



- 3) Kemudian variable/pointer awal dipindahkan ke pointer baru.



c. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```
Type Point = ^Data ;
  Data = record
    Info      : integer ;
    next     : point;
  End;
Var   Elemen      : integer ;
      Awal, Akhir   : point ;

Procedure Tambah_Depan(var Awal,Akhir: point; Elemen:integer);
Var Baru : point;
Begin
  New(Baru);
  Baru^.Info :=Elemen;
  If Awal= NIL then
    Akhir:= Baru
  Else
    Baru^.next:= Awal;
  Awal := Baru;
End;
```

(3) Menambah / menyisipkan simpul di tengah

Proses penambahan di tengah berarti proses penyisipan data pada posisi tertentu. Oleh karena itu, posisi penyisipan sangat diperlukan.

Ada beberapa kondisi yang harus diperhatikan ketika ingin melakukan penyisipan data, yaitu kondisi ketika *Linked list* masih kosong, dan ketika *Linked list* sudah mempunyai data.

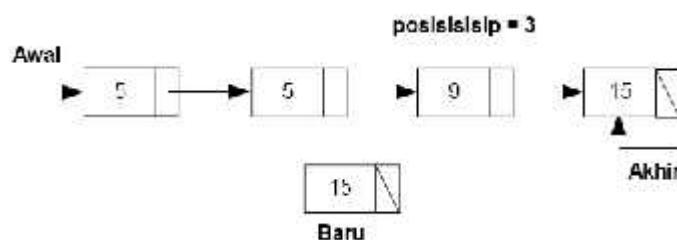
- a. **Posisi penyisipan di luar dari jangkauan Linked list (posisi kurang dari 1 atau melebihi banyak data yang ada di Linked list).**

Pada proses ini, jika terjadi posisi penyisipan kurang dari 1 maka proses yang dilakukan adalah proses penambahan data di awal, dan jika posisi di luar dari banyak data maka proses yang dilakukan adalah proses penambahan data di akhir.

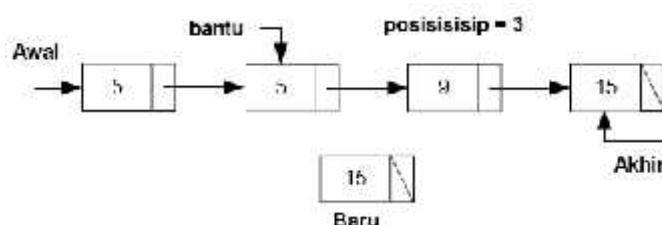
- b. **Posisi penyisipan di dalam jangkauan Linked list.**

Contoh kalau ingin menyisipkan data pada posisi ke-3 (posisisisp=3).

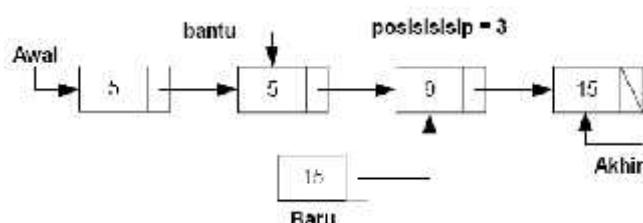
- 1) membuat simpul baru, kemudian diisi info baru.



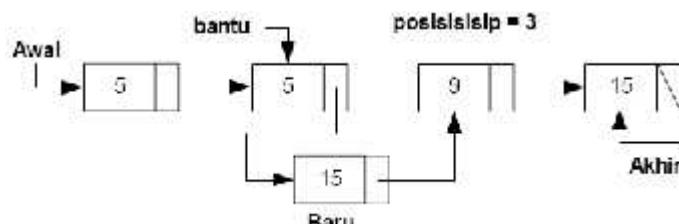
- 2) menentukan di mana simpul baru akan ditambahkan, yaitu dengan menempatkan pointer bantu pada suatu tempat.



- 3) pointer baru dihubungkan ke simpul setelah simpul yang ditunjuk oleh pointer bantu



- 4) kemudian penunjuk pointer bantu diarahkan ke simpul baru.



c. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```
Procedure Tambah_Tengah(Var Awal,Akhir: point; Elemen: integer);
Var Baru,Bantu : point;
Begin
    New(Baru); Baru^.info:= Elemen;
    If Awal = NIL then Begin
        Awal:= Baru;
        Akhir:= Baru;
    End Else Begin
        Bantu:= Awal;
        While (Elemen > Bantu^.Next^.Info) and (Bantu <> NIL) do
            Bantu:= Bantu^.Next;
        Baru^.Next:= Bantu^.Next;
        Bantu^.Next:= Baru;
    End;
End;
```

3.1.2.2. Menghapus simpul

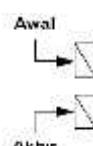
Dalam menghapus simpul ada satu hal yang perlu diperhatikan, yaitu bahwa simpul yang bisa dihapus adalah simpul yang berada sesudah simpul yang ditunjuk oleh suatu pointer, kecuali untuk simpul pertama.

(1) Menghapus simpul pertama

Penghapusan data di awal adalah proses menghapus elemen pertama (awal), sehingga variable awal akan berpindah ke elemen data berikutnya. Ada 3 kondisi yang perlu diperhatikan yaitu kondisi *Linked list* masih kosong, kondisi *Linked list* hanya memiliki 1 simpul, dan kondisi *Linked list* yang memiliki data lebih dari 1 elemen.

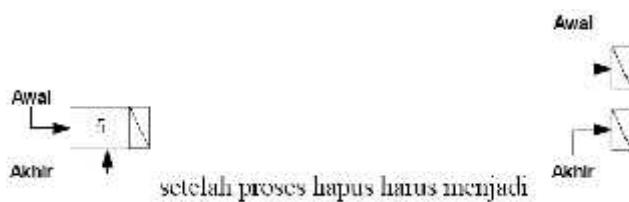
a. Kondisi linked list kosong

Pada kondisi ini proses penghapusan tidak bisa dilakukan.



Gambar 3.7. Ilustrasi linked list kosong

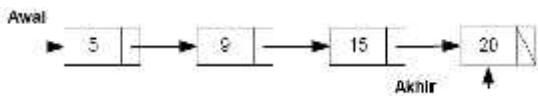
b. Kondisi linked list berisi 1 simpul



Gambar 3.8. Ilustrasi penghapusan simpul pada linked list yang berisi 1 simpul

Langkah yang dilakukan adalah menghapus data yang ada di posisi awal kemudian akhir dan awal di-NULL-kan.

c. Kondisi linked list berisi lebih dari 1 simpul

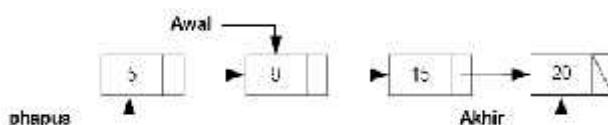


Fungsi menghapus simpul di depan. Langkah-langkahnya:

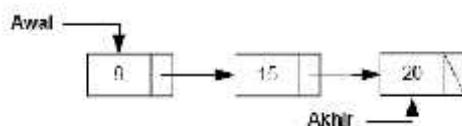
- 1) simpul bantu (phapus) diarahkan pada simpul awal.



- 2) simpul awal diarahkan ke simpul berikutnya.



- 3) menghapus simpul bantu(phapus). Sehingga *Linked list* menjadi seperti di bawah ini.



d. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```
Procedure Hapus_Awal_Simpul(var Awal, Akhir: point; Elemen: integer);
Var phapus: point;
Begin
  If awal=nil then
    Write('List tidak dapat dihapus')
  Else if awal=akhir then begin
    dispose(awal);
    awal := nil;
    akhir := nil;
  end else begin
    phapus := awal;
    Awal:= Awal^.Next;
    Dispose(phapus);
  End;
End;
```

(2) Menghapus simpul di akhir

Penghapusan data akhir adalah proses menghilangkan/menghapus data yang ada di posisi terakhir. Ada 3 kondisi yang harus diperhatikan ketika akan melakukan proses penghapusan data akhir adalah kondisi *Linked list* masih kosong, kondisi *Linked list* hanya berisi 1 data, dan kondisi *Linked list* berisi data lebih dari 1 buah.

a. Kondisi linked list kosong

Pada kondisi ini proses penghapusan tidak bisa dilakukan seperti terlihat pada Gambar 3.7.

b. Kondisi linked list berisi 1 simpul

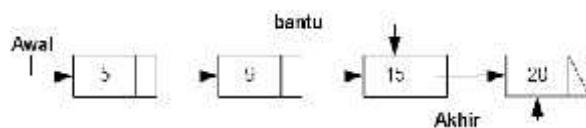
Langkah yang dilakukan adalah menghapus data yang ada di posisi awal kemudian akhir dan awal di-NULL-kan seperti diilustrasikan pada Gambar 3.8.

c. Kondisi linked list berisi lebih dari 1 simpul

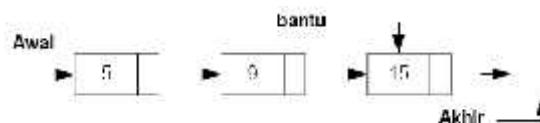


Karena posisi hapus adalah data terakhir, maka nanti posisi akhir harus pindah ke posisi sebelumnya. Oleh karena itu harus dicari posisi data sebelum data terakhir, sebut dengan variabel bantu. Fungsi menghapus simpul di Akhir, langkah-langkahnya:

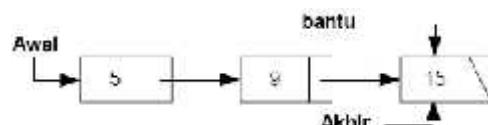
- meletakkan pointer bantu di sebelah kiri simpul yang akan dihapus.



- menghapus simpul akhir. Sehingga *Linked list* menjadi seperti di bawah ini.



- Kemudian posisi akhir yang datanya telah dihapus dipindahkan ke posisi bantu dan kemudian field next dari bantu di-NULL-kan.



d. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```
Procedure Hapus_Akhir(var Awal, Akhir: point; Elemen: integer);
Var Bantu, Hapus : point;
Begin {senarai masih kosong}
  If awal= NIL then Writeln('Senarai berantai masih kosong !')
  Else if awal=akhir then begin
    dispose(awal);
    awal := nil;
    akhir := nil;
  end else Begin {menghapus simpul akhir}
    Bantu:= Awal;
    While (Bantu^.Next <> akhir) do
      Bantu:= Bantu^.Next;
    Dispose(akhir);
    Akhir := bantu;
  End;
End;
```

(3) Menghapus simpul di tengah

Penghapusan data di tengah berarti menghapus suatu elemen data *Linked list* pada posisi tertentu. Ada kondisi-kondisi yang perlu diperhatikan yaitu kondisi ketika *Linked list* masih kosong dan kondisi ketika *Linked list* sudah mempunyai data.

a. Proses penghapusan data ketika linked list masih kosong

Ketika kondisi ini tercapai, maka proses penghapusan tidak dilaksanakan karena *Linked list* masih kosong seperti diilustrasikan pada Gambar 3.7.

b. Proses penghapusan data ketika linked list sudah mempunyai data.

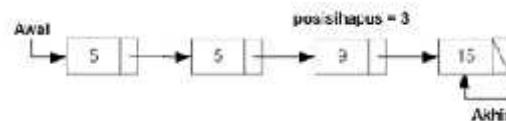
Ada 3 kondisi yang dapat terjadi pada penghapusan yaitu :

- 1) Posisi penghapusan di luar dari jangkauan *Linked list*.

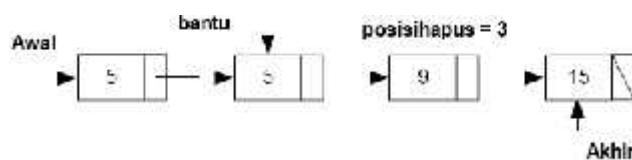
Jika posisi penghapusan berada di luar jangkauan (posisi<1 atau posisi>banyak data) maka proses penghapusan dikerjakan.

- 2) Posisi penghapusan di dalam jangkauan *Linked list*.

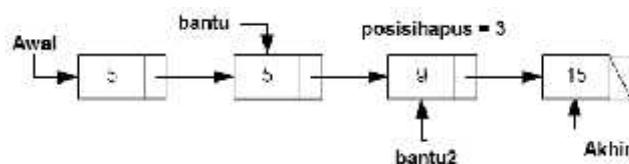
Jika posisi penghapusan sama dengan 1, maka proses yang dikerjakan adalah proses penghapusan data awal, dan jika posisi penghapusan sama dengan banyak data, maka proses penghapusan yang dikerjakan sama dengan proses penghapusan data akhir. Jika posisi penghapusan ada di tengah ($1 < \text{posisi hapus} < \text{banyak data}$), langkah-langkah penghapusannya dapat dilihat pada contoh di bawah ini. Sebagai contoh, penghapusan akan dilakukan pada data 3 dari banyak data sebanyak 4.



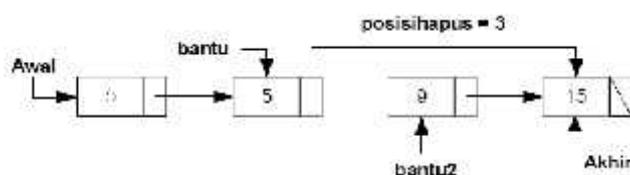
- 3) Kemudian cari posisi elemen sebelum elemen posisi hapus, kemudian simpan dalam variabel bantu.



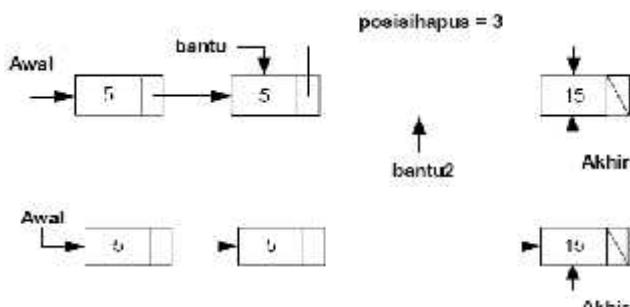
- 4) Kemudian simpan alamat elemen posisi hapus dalam suatu variabel dengan nama bantu2.



- 5) Kemudian pindahkan field next dari bantu ke alamat yang ditunjuk oleh field next dari bantu2.



- 6) Hapus elemen data yang ditunjuk dengan bantu2.



c. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```

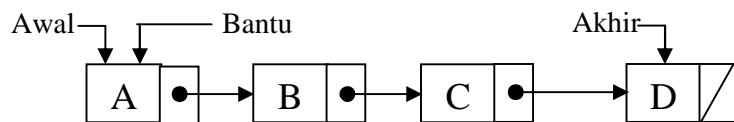
Procedure Hapus_Akhir(var Awal, Akhir: point; Elemen: integer);
Var
    Bantu, Bantu2 : point;
Begin {senarai masih kosong}
    If awal= NIL then
        Writeln('Senarai berantai masih kosong !')
    Else {menghapus simpul akhir}
        Begin
            Bantu:= Awal;
            While (Elemen <> Bantu^.Next^.info) and (Bantu^.Next^.Next <> NIL) do
                Bantu:= Bantu^.Next;
            If bantu^.next^.info = elemen then
                Begin
                    Bantu2:= Bantu^.next;
                    If Bantu2 = Akhir then
                        Begin
                            Akhir:= Bantu;
                            Akhir^.Next:= NIL;
                        End Else
                            Bantu^.Next:= Bantu2^.Next;
                            Dispose(Bantu2);
                    End Else
                        writeln('Tidak ketemu yang dihapus !!');
                End;
            End;
        End;
    End;
```

3.1.3. Membaca isi linked list

Pembacaan isi *Linked list* dapat dilakukan dengan 2 cara : yaitu dengan pembacaan secara maju dan pembacaan secara mundur.

3.1.3.1. Membaca isi linked list secara maju

Pertama kali kita atur supaya pointer Bantu menunjuk ke simpul yang ditunjuk oleh pointer Awal. Setelah isi simpul itu dibaca, maka pointer Bantu kita gerakkan ke kanan untuk membaca simpul berikutnya. Proses ini kita ulang sampai pointer Bantu sama dengan pointer Akhir.

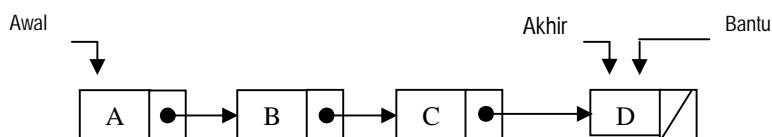


Gambar 3.9. Ilustrasi proses membaca isi simpul secara maju

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```
Procedure Baca_Maju(Awal,Akhir : point);
Var
    Bantu : point;
Begin
    Bantu:= Awal;
    Repeat
        Write(Bantu^.info:2);
        Bantu:= Bantu^.Next;
    Until Bantu = NIL;
    Writeln;
End;
```

3.1.3.2. Membaca isi linked list secara mundur



Gambar 3.10. Ilustrasi proses membaca isi simpul secara mundur

Ada 2 cara membaca mundur isi dari *Linked list*, yaitu dengan cara biasa seperti diatas atau dengan memanfaatkan proses rekursi.

(1) Cara pertama

- Atur pointer Bantu sama dengan pointer Awal.
- Berikutnya, pointer awal kita arahkan ke simpul Akhir dan kita pakai pointer lain misalkan Bantu1 untuk bergerak dari simpul pertama sampai simpul sebelum simpul terakhir.
- Langkah selanjutnya adalah mengarahkan medan pointer dari simpul terakhir ke simpul Bantu1.
- Langkah ini diulang sampai pointer Akhir Berimpit dengan pointer Bantu.
- Procedure cara ini dapat dilihat pada program Pascal dibawah ini.

```

Procedure Baca_Terbalik(var Awal,Akhir: point);
Var Bantu,Bantu1 : point;
Begin
    Bantu:= Awal;
    Awal:= Akhir;
    Repeat
        Bantu1:=Bantu;
        While Bantu1^.Next <> Akhir do
            Bantu1:= Bantu1^.Next;
            Akhir^.Next:= Bantu1;
            Akhir:= Bantu1;
        Until Akhir = Bantu;
        Akhir^.Next:= NIL;
    End;

```

(2) Cara kedua dengan rekursi.

Caranya hampir sama dengan cara membaca *Linked list* secara maju hanya saja pencetakan isi simpul ditunda sampai pointer Bantu sama dengan pointer akhir.

Procedure cara ini dapat dilihat pada program Pascal dibawah ini.

```

Procedure Terbalik(Bantu: point);
Begin
    If Bantu <> NIL then
        Begin
            Terbalik(Bantu^.next);
            Write(Bantu^.Info:2);
        End;
    End;

```

3.1.4. Mencari data dalam linked list

Data yang dicari disimpan dalam suatu variabel misalkan cari. Pertama pointer Bantu dibuat sama dengan pointer awal Isi simpul yang ditunjuk oleh pointer Bantu dibandingkan dengan cari. Jika sama beritahukan bahwa data yang dicari ditemukan, jika belum sama maka pointer Bantu dipindahkan ke simpul disebelah kanannya, dan proses pembandingan diulang. Proses ini diulang sampai data yang dicari ditemukan atau sampai pada simpul akhir.

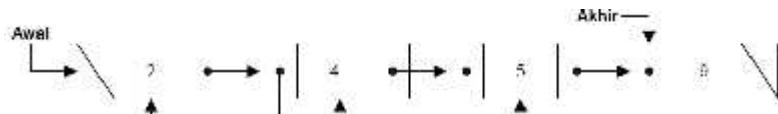
```

Function Cari_Data(Awal: point; Elemen : integer): Boolean;
Var ketemu: Boolean;
Begin
    Ketemu: false;
    Repeat
        If Awal^.info = Elemen then
            Ketemu:= true
        Else
            Awal:= Awal^.Next;
    Until (ketemu) or (Awal = NIL);
    Cari_Data:= ketemu;
end;

```

3.2. Double Linked List

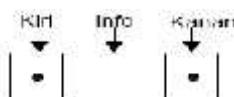
Double linked list adalah suatu *Linked list* yang mempunyai 2 penunjuk yaitu penunjuk ke data sebelumnya dan berikutnya, seperti terlihat pada Gambar 3.20.



Gambar 3.11. Ilustrasi double linked list

3.2.1. Pendeklarasian struktur dan variabel double linked list

Jika dilihat 1 elemen listnya, maka secara umum struktur dari elemen listnya adalah sebagai berikut :



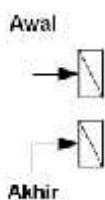
Gambar 3.12. Dekripsi simpul double linked list

Dari gambar 3.21 untuk setiap elemen terdiri dari 3 buah field yaitu kiri (prev), info (data), dan kanan (next). Field kiri dan kanan merupakan sebuah pointer ke data struktur elemen (tdata).

Pendeklarasian struktur *double linked list* dalam bahasa Pascal adalah :

```
Type
    Typedata = .....;
    Point = ^Data ;
    Data = record
        Info          : Typedata ;
        Kiri,kanan   : point;
    End;
    Var     Elemen      : typedata ;
           Awal, Akhir, Baru : point ;
```

Kondisi awal ketika awal dan akhir telah dideklarasikan.



Gambar 3.13. Dekripsi simpul double linked list setelah dideklarasikan.

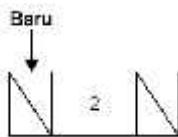
3.2.2. View data

Untuk view data, langkah yang dilakukan adalah dengan menelusuri semua elemen list sampai elemen terakhir. Setelah melakukan penelusuran posisi awal dan akhir elemen tidak boleh berubah sehingga untuk melakukan penelusuran dibutuhkan sebuah variabel bantu. Kelebihan dari view data pada *Linked list* adalah kita dapat menampilkan data dari data terakhir dengan lebih mudah.

3.2.3. Tambah data

3.2.3.1. Tambah di awal

Operasi ini berguna untuk menambahkan elemen baru di posisi pertama. Langkah pertama untuk penambahan data adalah pembuatan elemen baru dan pengisian infonya. Pointer yang menunjuk ke data tersebut dipanggil dengan nama **baru**. Kondisi setelah ada pembuatan elemen baru tersebut adalah :



Gambar 3.14. Ilustrasi simpul double linked list yang berisi 1 simpul.

Perintah pembuatan elemen *double linked list* dalam bahasa Pascal

```

Type
    Typedata = integer;
    Point = ^Data ;
    Data = record
        Info           : Typedata ;
        Kiri, kanan   : point;
    End;
Var
    Elemen          : Typedata ;
    Awal, Akhir,Baru : point ;
Begin
    New(Baru);
    Baru^.Info := elemen;
    Baru^.kiri := nil;
    Baru^.kanan := nil;
End;

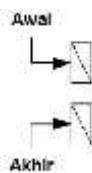
```

Ada 2 kondisi yang harus diperhatikan dalam penambahan data di awal yaitu :

(1) Ketika double linked list masih kosong

Kalau kondisi *double linked list* masih kosong, maka elemen baru akan menjadi awal dan akhir *double linked list*.

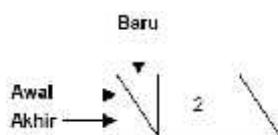
a. Kondisi sebelum disisipkan



Gambar 3.15. Ilustrasi simpul double linked list dalam kondisi kosong

b. Kondisi setelah operasi penambahan

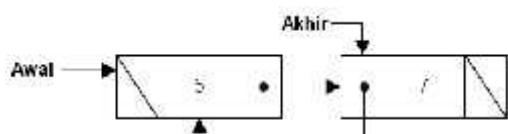
Operasi penambahan awal ketika *double linked list* masih kosong adalah dengan mengisikan alamat pointer baru ke pointer awal dan pointer akhir. Lihat gambar di bawah ini.



Gambar 3.16. Ilustrasi simpul double linked list yang berisi 1 simpul.

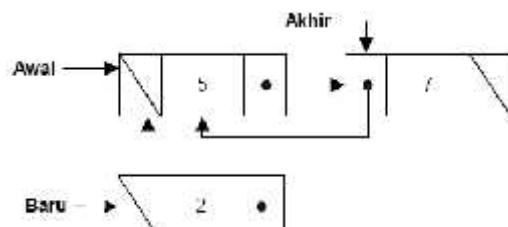
(2) Ketika double linked list sudah mempunyai data

Kondisi *double Linked list* ketika sudah mempunyai data elemen dan elemen yang baru telah dibuat, dapat dilihat di gambar di bawah ini.

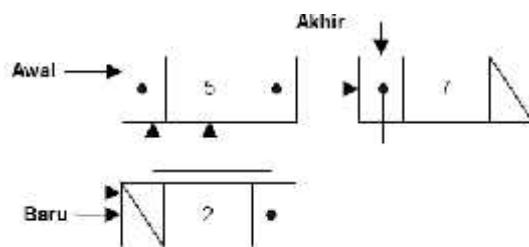


Proses penambahan data di awal *double linked list* adalah :

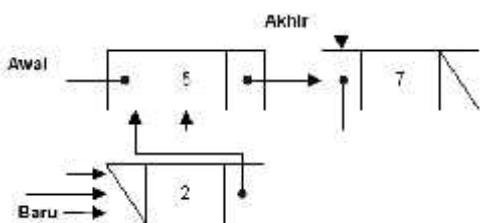
a. Hubungkan pointer baru^.kanan ke pointer awal



b. Hubungkan pointer awal^.kiri agar menunjuk ke posisi pointer baru



c. Pindahkan pointer awal ke pointer baru



d. Implementasi dalam program

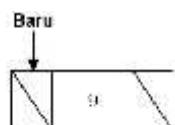
Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal sebagai berikut :

```
Type Point = ^Data ;
  Data = record
    Info      : integer ;
    Kanan,kiri : point;
  End;
Var   Elemen      : integer ;
      Awal, Akhir, Baru : point ;

Procedure Tambah_Awal(Awal,Akhir : Point;Elemen: integer);
Var Baru : point;
Begin
  New(Baru);
  Baru^.Info := elemen;
  Baru^.kiri := nil;
  Baru^.kanan := nil
  If Awal = NIL then begin
    Awal:= Baru;
    Akhir := Baru;
  End Else      begin
    Baru^.kanan := awal;
    Awal^.kiri := Baru;
    Awal:= Baru;
  End;
End;
```

3.2.3.2. Tambah di akhir

Operasi ini berguna untuk menambahkan elemen baru di posisi akhir. Langkah pertama untuk penambahan data adalah pembuatan elemen baru dan pengisian infonya. Pointer yang menunjuk ke data tersebut dipanggil dengan nama **baru** seperti terlihat pada Gambar 3.17.



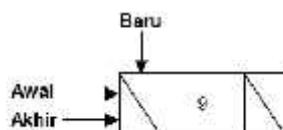
Gambar 3.17. Ilustrasi penambahan di akhir simpul double linked list yang berisi 1 simpul

(1) Ketika double linked list masih kosong

Apabila *double linked list* masih kosong, maka elemen baru akan menjadi awal dan akhir *Linked list*.

- Kondisi sebelum penambahan seperti terlihat pada Gambar 3.15
- Kondisi setelah operasi penambahan

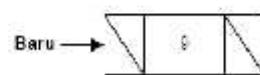
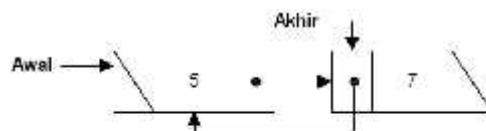
Operasi penambahan awal ketika *double linked list* masih kosong adalah dengan mengisikan alamat pointer baru ke pointer awal dan pointer akhir. Lihat gambar 3.18



Gambar 3.18. Simpul double linked list kosong yang telah ditambahkan simpul baru

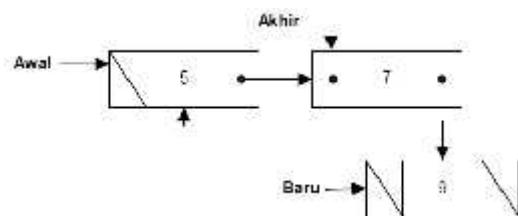
(2) Ketika double linked list ada isinya

Kondisi *double linked list* ketika sudah mempunyai data elemen dan elemen yang baru telah dibuat, dapat dilihat di gambar di bawah ini.

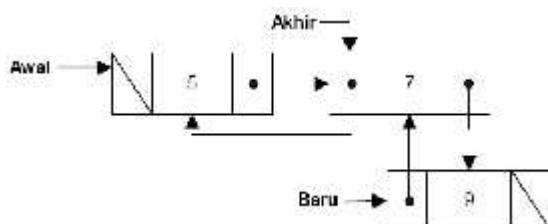


Proses penambahan data di akhir *double linked list* adalah :

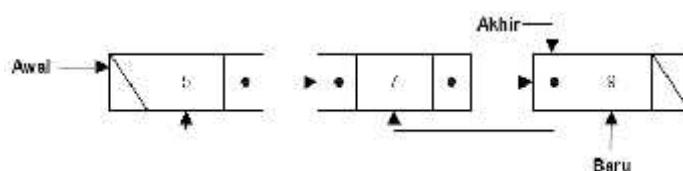
- Hubungkan akhir^.kanan agar menunjuk ke pointer baru**



- Hubungkan baru^.kiri agar menunjuk ke posisi pointer akhir**



- Pindahkan pointer akhir ke pointer baru**



d. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```
Procedure Tambah_Akhir(Awal,Akhir : point; Elemen: Integer);
Var Baru : point;
Begin
    New(Baru);
    Baru^.Info := elemen;
    Baru^.kiri := nil;
    Baru^.kanan := nil
    If Awal = NIL then
        begin
            Awal:= Baru;
            Akhir := Baru;
        End Else
        begin
            Akhir^.kanan := Baru;
            Baru^.kiri := Akhir;
            Akhir:= Baru;
        End;
    End;
```

3.2.3.3. Tambah di tengah

Operasi penyisipan data di tengah *double linked list* adalah suatu operasi menambah data di posisi tertentu di dalam *double linked list*. Contohnya adalah jika ingin menyisipkan data di posisi ke-3 atau ke-4.

Untuk proses tersebut ada 2 hal yang harus diperhatikan yaitu :

- (1) **Ketika double linked list masih kosong atau posisi penyisipan kurang dari atau sama dengan 1**

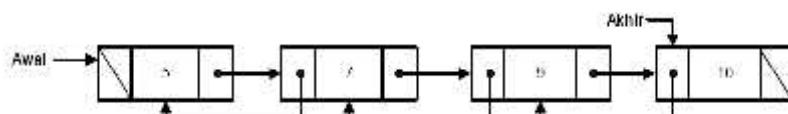
Jika kondisi ini terjadi, maka langkah yang dilakukan adalah sangat mudah yaitu dengan memanggil proses penambahan data awal atau akhir. (untuk lebih jelas lihat penambahan data awal atau akhir ketika kondisi *double linked list* masih kosong).

- (2) **Kondisi double linked list sudah tedapat data**

- a. **Cari posisi pointer pada data ke-posisi sisip.**

Caranya adalah dengan menelusuri *double linked list* sebanyak **posisi sisip** kali.

Contoh *Double Linked list*



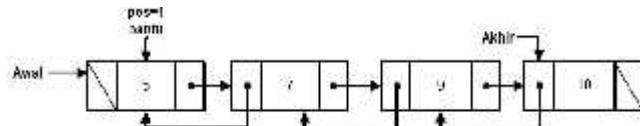
Gambar 3.19. Contoh double linked list dengan 4 simpul

Ketika pencarian posisi pointer pada data ke-**posisi sisip** dicari, maka ada dua kemungkinan yaitu posisi sisip ada di dalam *double linked list* atau diluar *double linked list*

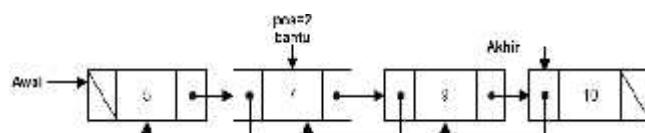
(kalau mengisi posisi lebih besar dari banyak data). Oleh karena itu pencarian pointer posisi sisip ada dua kemungkinan. Perhatikan contoh di bawah ini.

Contoh 1 : Penyisipan di posisi 3 dengan data yang telah ada sebanyak 4 buah.

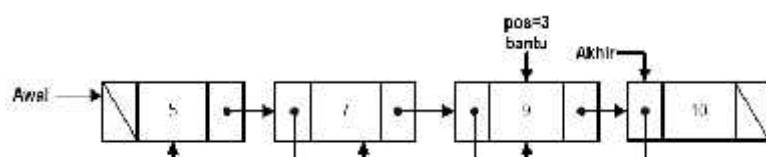
- 1) Pointer bantu diisi dengan data awal, pos diisi 1 (data pertama)



- 2) Jika pos belum sama dengan posisi sisip, maka pindah ke data berikutnya serta pos ditambah 1.



- 3) Karena pos masih lebih kecil dari posisi sisip, maka pindahkan pos dan bantu ke posisi berikutnya dan pos ditambah 1.



- 4) Karena pos telah sama dengan posisi sisip maka perulangan pencarian telah selesai. Itu menunjukkan posisi penyisipan adalah di posisi yang ditunjuk oleh bantu.

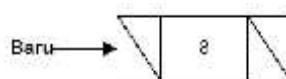
b. Posisi penyisipan (bantu) ditemukan

Jika posisi penyisipan ditemukan maka periksa apakah posisi penyisipan (bantu) bernilai NULL atau tidak. Jika posisi penyisipan (bantu) bernilai NULL berarti posisi sisip berada di luar atau melebihi *double linked list*. Oleh karena itu berarti penambahan datanya dilakukan dengan melakukan operasi penambahan akhir.

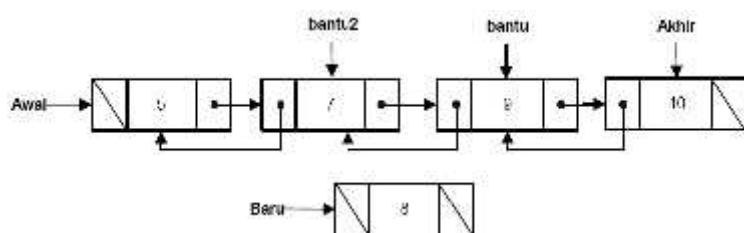
c. Jika posisi penyisipan (bantu) tidak sama dengan NULL

Maka posisi penyisipan ada di dalam jangkauan *double linked list*. Proses yang dilakukan untuk penyisipannya adalah :

- 1) Buat elemen baru di memori dan isi infonya (contoh data info = 8).

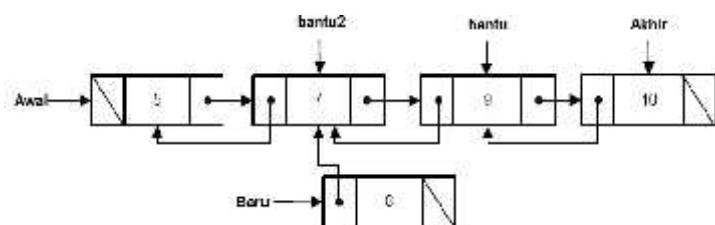


- 2) Untuk mempermudah proses penyambungan/penyisipan data baru, maka buat variabel pointer baru dengan nama bantu2 untuk memegang data di sebelah kiri dari posisi sisip (bantu^.kiri). (Contoh posisi penyisipan adalah 3)

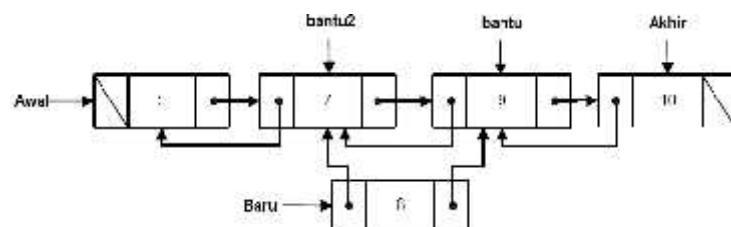


Gambar 3.20. Ilustrasi double linked list yang akan disisipi simpul pada posisi ke-3

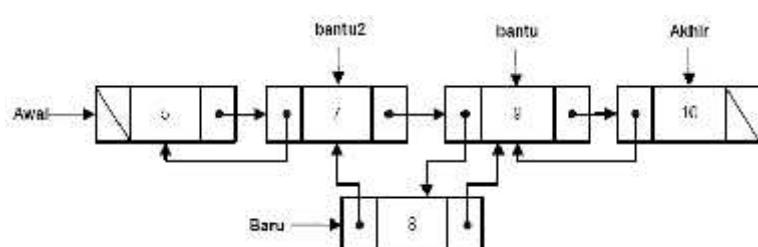
- a) Isi baru^.kiri dengan pointer bantu2



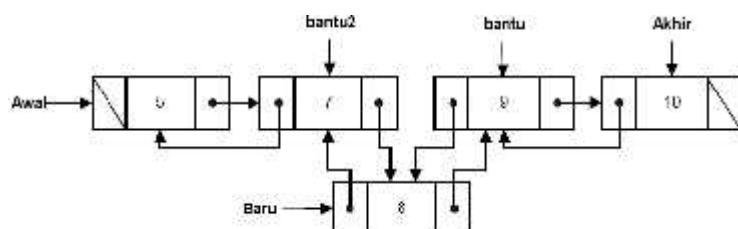
- b) Isi/sambungkan baru^.kanan ke pointer bantu



- c) Isi/sambungkan bantu^.kiri ke posisi pointer baru



- d) Isi/sambungkan bantu2^.kanan ke posisi pointer baru



d. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```

Procedure Sisip_Tengah(Awal,Akhir : point; posisi, Elemen: integer);
Var Baru : point;
    Pos : integer;
Begin
    New(bantu); new(bantu2); New(Baru);
    If (awal = nol) or (posisi <=1)
        tambahawal(awal,akhir,data);
    else begin
        bantu=awal;
        pos=1;
        while((pos<posisi) and (bantu <> Nil)) do
        begin
            pos := pos + 1;
            bantu=bantu^.kanan;
        end;
        if(bantu =Nil)
            tambahakhir(awal,akhir,data);
        else begin
            baru^.kiri=Nil;
            baru^.kanan=Nil;
            baru^.info=data;
            bantu2=bantu^.kiri;
            baru^.kiri=bantu2;
            baru^.kanan=bantu;
            bantu^.kiri=baru;
            bantu2^.kanan=baru;
        end;
    end;
end;

```

3.2.4. Hapus

3.2.4.1. Hapus data di awal

Operasi ini berguna untuk menghapus data pada posisi pertama. Ada 3 keadaan yang mungkin terjadi ketika akan melakukan proses hapus yaitu :

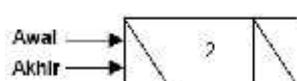
(1) Kondisi double linked list masih kosong

Jika kondisi ini terjadi, maka proses penghapusan data tidak bisa dilakukan karena *Linked list* masih kosong.

(2) Kondisi double linked list hanya memiliki 1 data

Langkah yang perlu dilakukan ketika ingin melakukan proses penghapusan *double linked list* yang memiliki hanya 1 data adalah dengan langsung menghapus data dari memori dan kemudian pointer awal dan akhir di-NULL-kan. Untuk lebih jelas perhatikan urutan penghapusannya di bawah ini :

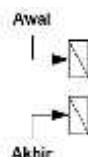
- Kondisi data sebelum dihapus



- b. Proses penghapusan yaitu dengan menghilangkan data dari memori dengan perintah free(awal) atau free(akhir).



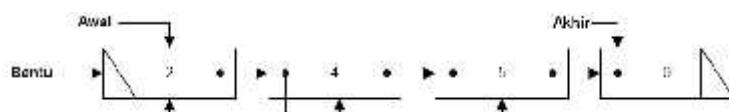
- c. Kemudian pointer awal dan akhir diisi dengan NULL.



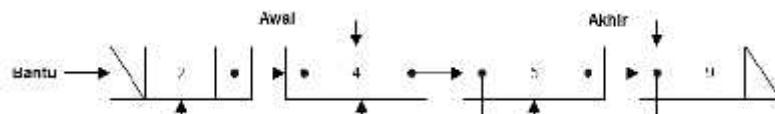
(3) Kondisi linked list memiliki lebih dari 1 data

Untuk operasi penghapusan data di posisi pertama pada *double linked list* yang mempunyai data lebih dari 1 buah adalah :

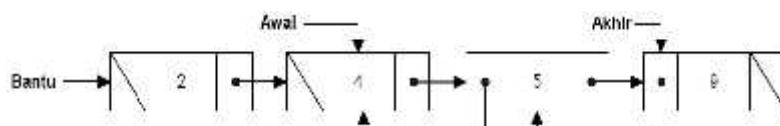
- a. Simpan pointer yang akan dihapus (awal) ke suatu pointer lain yang diberi nama pointer **bantu**.



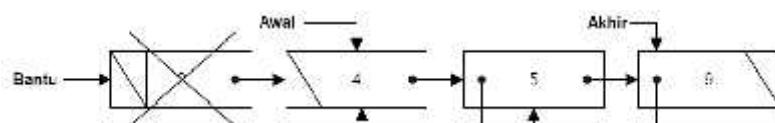
- b. Pindahkan pointer awal ke data berikutnya (bantu^.kanan atau awal^.kanan).



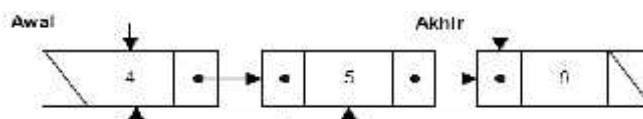
- c. Field kiri dari awal yang baru (awal^.kiri) di-NULL-kan.



- d. Langkah terakhir adalah hapus elemen yang ditunjuk pointer bantu.



- e. Setelah data dihapus, maka kondisi *double linked list* adalah seperti di gambar di bawah ini.



- f. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```

Procedure hapusawal(awal, akhir : point)
Var Bantu : point;
begin
    if(awal=Nil) then writeln('Data kosong. Tidak bisa dihapus');
    if(awal=akhir) then begin
        dispose(awal);
        awal=nil;
        akhir=Nil;
    end else begin
        bantu=awal;
        awal=bantu^.kanan;
        awal^.kiri=Nil;
        dispose(bantu);
    end;
end;

```

3.2.4.2. Hapus data akhir

Operasi ini berguna untuk menghapus data pada posisi terakhir. Ada 3 keadaan yang mungkin terjadi ketika akan melakukan proses hapus yaitu:

(1) Kondisi double linked list masih kosong

Jika kondisi ini terjadi, maka proses penghapusan data tidak bisa dilakukan karena *double linked list* masih kosong.

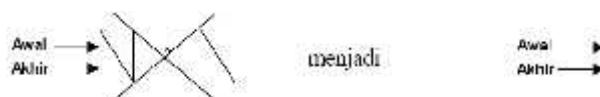
(2) Kondisi double linked list hanya memiliki 1 data

Langkah yang perlu dilakukan ketika ingin melakukan proses penghapusan *double linked list* yang memiliki hanya 1 data adalah dengan langsung menghapus data dari memori dan kemudian pointer awal dan akhir di-NULL-kan. Untuk lebih jelas perhatikan urutan penghapusannya di bawah ini :

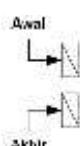
- Kondisi data sebelum dihapus



- Proses penghapusan yaitu dengan menghilangkan data dari memori dengan perintah `dispose(awal)` atau `dispose(akhir)`.



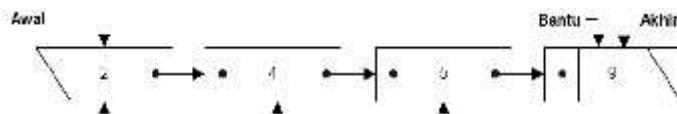
- Kemudian pointer awal dan akhir diisi dengan NULL.



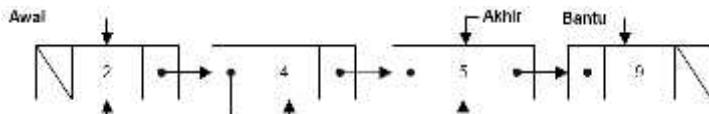
(3) Kondisi double linked list memiliki data lebih dari 1 buah

Untuk operasi penghapusan data di posisi terakhir pada *double linked list* yang mempunyai data lebih dari 1 buah adalah :

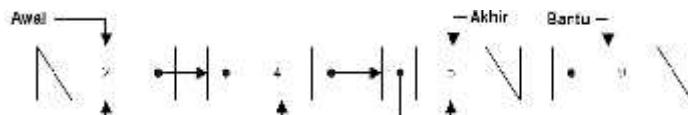
- a. Simpan pointer yang akan dihapus (akhir) ke suatu pointer lain yang diberi nama pointer bantu.



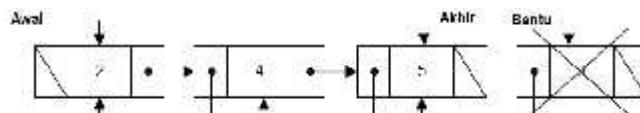
- b. Pindahkan pointer akhir ke data sebelumnya (bantu^.kiri atau akhir^.kiri).



- c. Field kanan dari akhir baru (akhir^.kanan) di-NULL-kan.



- d. Langkah terakhir adalah hapus elemen yang ditunjuk pointer bantu.



- e. Setelah data dihapus, maka kondisi *double linked list* adalah seperti di gambar di bawah ini.



- f. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```

Procedure hapusakhir(awal, akhir : point)
Var   Bantu : point;
begin
    if(awal=Nil) then writeln(Data kosong. Tidak bisa dihapus);
    if(awal=akhir) then begin
        dispose(awal);
        awal = Nil;
        akhir=Nil;
    end else begin
        bantu=akhir;
        akhir=bantu^.kiri;
        akhir^.kanan=Nil;
        dispose(bantu);
    end;
end;
```

3.2.4.3. Hapus data di tengah

Untuk melakukan proses penghapusan di tengah *double linked list*, ada 3 kondisi yang perlu diperhatikan yaitu :

(1) Kondisi ketika double linked list masih kosong atau ketika posisi hapus lebih kecil dari 1.

Ketika kondisi ini terjadi, maka proses penghapusan tidak bisa dilakukan karena data masih kosong atau karena posisi hapus diluar jangkauan *double linked list* (posisi kurang dari 1).

(2) Kondisi ketika posisi hapus sama dengan 1 (hapus data pertama)

Ketika kondisi ini terjadi, maka proses yang dilakukan adalah proses penghapusan di posisi awal (hapusawal).

(3) Kondisi ketika posisi hapus lebih besar dari 1

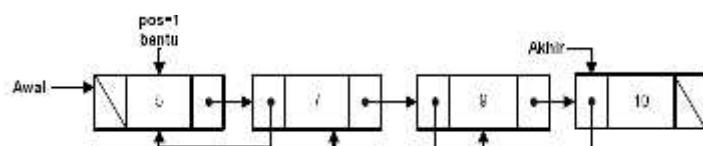
Langkah-langkah untuk penghapusan data di tengah *double linked list* yang posisi hapusnya lebih besar dari 1 adalah :

a. **Cari pointer yang menunjuk ke data pada posisi ke-posisi hapus.**

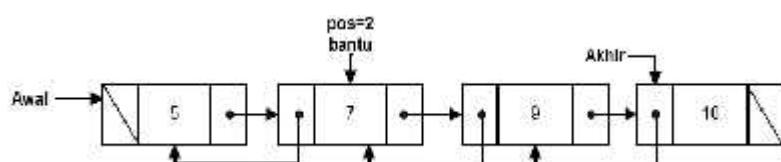
Ketika kondisi ini, ada 2 kemungkinan yang bisa terjadi yaitu posisi hapus ada di dalam jangkauan *double linked list* atau di luar *Linked list*. Untuk lebih jelas perhatikan 2 contoh di bawah ini :

Contoh 1 : Kondisi posisi hapus ada di dalam jangkauan double linked list, contoh penghapusan pada posisi 3.

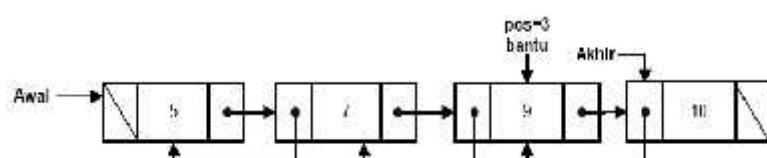
- 1) Pointer bantu diisi dengan data awal, pos diisi 1 (data pertama)



- 2) Jika pos belum sama dengan posisi hapus, maka pindah ke data berikutnya dan pos ditambah 1.



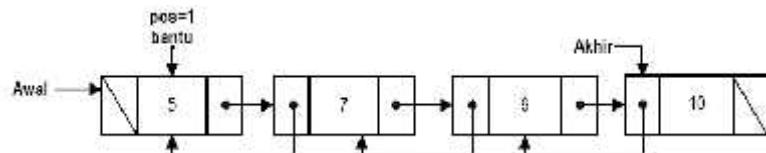
- 3) Karena pos masih lebih kecil dari posisi hapus, maka pindahkan bantu ke posisi berikutnya dan pos ditambah 1.



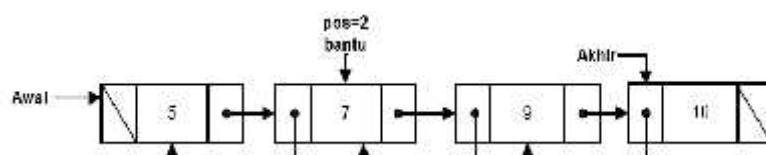
- 4) Karena pos telah sama dengan posisi hapus maka perulangan pencarian posisi penghapusan telah selesai. Itu menunjukkan posisi penghapusan adalah di posisi yang ditunjuk oleh bantu.

Contoh 2 : Kondisi posisi hapus ada di luar/melebihi jangkauan double linked list, contoh penghapusan pada posisi 10 tetapi dalam linked list hanya ada 4 buah data.

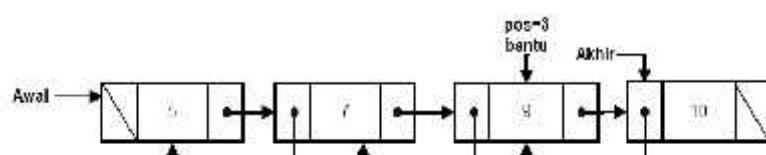
- 1) Pointer bantu diisi dengan data awal, pos diisi 1 (data pertama)



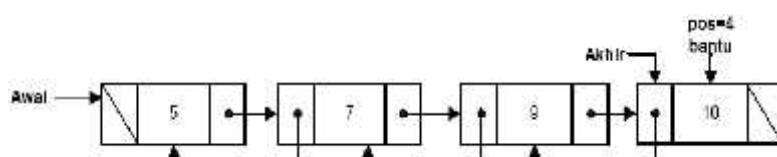
- 2) Jika pos belum sama dengan posisi hapus, maka pindah ke data berikutnya dan pos ditambah 1.



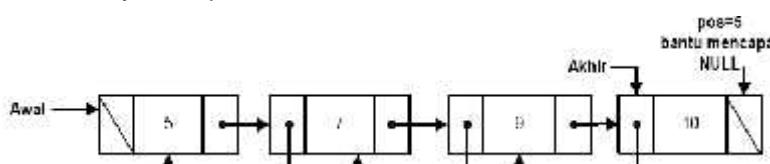
- 3) Karena pos masih lebih kecil dari posisi hapus, maka pindahkan pointer bantu ke posisi berikutnya dan variabel pos ditambah 1.



- 4) Karena pos masih lebih kecil dari posisi hapus, maka pindahkan pos dan bantu ke posisi berikutnya dan pos ditambah 1.



- 5) Karena pos masih lebih kecil dari posisi hapus, maka pindahkan pos dan bantu ke posisi berikutnya dan pos ditambah 1.



- 6) Karena bantu mencapai nilai NULL, maka perulangan harus berhenti karena itu menunjukkan posisi hapus ada di luar double linked list. Kalau ini terjadi, maka penghapusan tidak dapat dilakukan

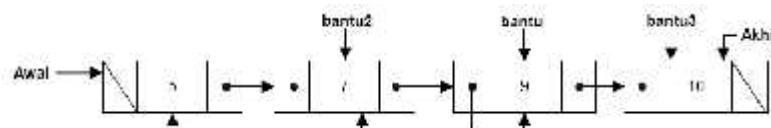
- b. Setelah pointer posisi penghapusan telah ditemukan, maka langkah selanjutnya adalah pemeriksaan.

Pemeriksaan dilakukan untuk mengetahui apakah posisi penghapusan (bantu) tersebut bernilai NULL (diluar jangkauan *double linked list*). Jika posisi penghapusan (bantu) bernilai NULL maka proses penghapusan tidak bisa dilakukan. Tetapi jika bantu tidak bernilai NULL, maka lanjutkan ke langkah berikutnya.

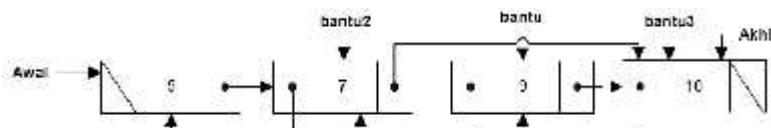
- c. Periksa apakah pointer posisi penghapusan (bantu) sama dengan posisi akhir.

Jika benar maka proses penghapusan yang dilakukan adalah proses penghapusan akhir. Tetapi jika posisi penghapusan tidak sama dengan posisi akhir itu menunjukkan posisi penghapusan ada di tengah, maka proses yang dilakukan adalah :

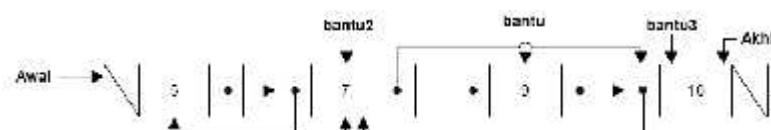
- Untuk mempermudah penghapusan, maka simpan pointer yang menunjuk ke data sebelum posisi penghapusan ($bantu2=bantu^.kiri$) dan data setelah posisi penghapusan ($bantu3=bantu^.kanan$).



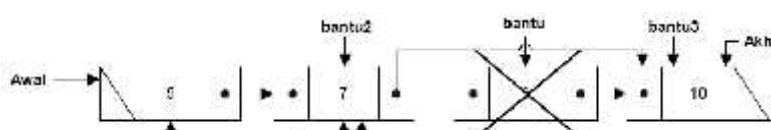
- Isi/sambungkan $bantu2^.kanan$ ke posisi pointer $bantu3$.



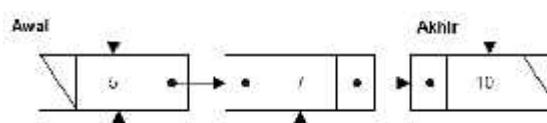
- Isi/sambungkan $bantu3^.kiri$ ke posisi pointer $bantu2$.



- Hapus dari memori data yang ditunjuk oleh pointer $bantu$.



- Setelah data di pointer $bantu$ dihapus, maka kondisi *double linked list* adalah



d. Implementasi dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

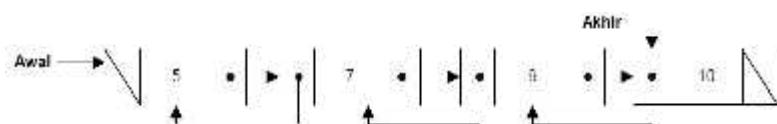
```

Procedure hapustengah(awal, akhir : point; posisi : integer)
Var
    bantu,bantu2,bantu3 : point;
    i : integer;
begin
    if((awal=nil) or (posisi<1)) then begin
        writeln(Data kosong atau posisi kurang dari 1');
        writeln('Tidak bisa dihapus');
    end else if (posisi==1)
        hapusawal(awal,akhir);
    else begin
        bantu=awal;
        i=1;
        while((i<posisi) and (bantu <> Nil)) do begin
            i=i+1;
            bantu=bantu^.kanan;
        end;
        if(bantu=Nil) then
            writeln('Posisi Diluar Jangkauan')
        else if (bantu = akhir) then
            hapusakhir(awal,akhir)
        else begin
            bantu2=bantu^.kiri;
            bantu3=bantu^.kanan;
            bantu2^.kanan=bantu3;
            bantu3^.kiri=bantu2;
            dispose(bantu);
        end
    end;
end;

```

3.2.5. Pencarian data

Pencarian dilakukan dengan memeriksa data yang ada dalam *double linked list* dengan data yang dicari. Pencarian dilakukan dari data pertama sampai data ditemukan atau pointer pencari (bantu) telah mencapai NULL yang menandakan bahwa data yang dicari tidak ditemukan. Agar lebih jelas perhatikan ilustrasi di bawah ini, dengan contoh data adalah :

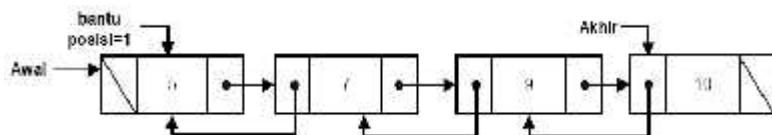


Ada 2 kondisi yang dihasilkan oleh proses pencarian yaitu

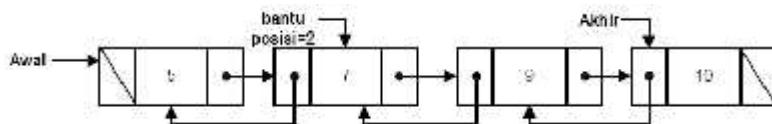
(1) Pencarian dimana data yang dicari dapat ditemukan

Kasus : data yang akan dicari adalah data 9.

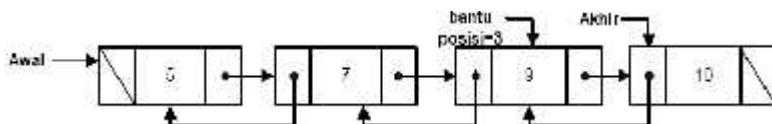
- Isi bantu dengan pointer data awal dan posisi diisi 1



- b. Jika data info yang ditunjuk oleh pointer bantu tidak sama dengan yang dicari, maka bantu pindah ke data berikutnya dan variabel posisi ditambah 1.



- c. Karena info yang ditunjuk oleh bantu belum sama dengan yang dicari, maka bantu pindah lagi ke data berikutnya dan variabel posisi ditambah 1

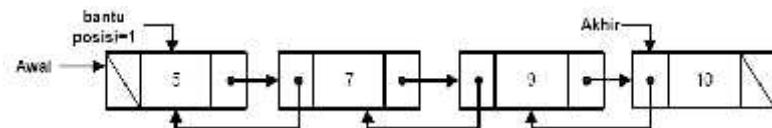


- d. Karena data info yang ditunjuk oleh bantu sama dengan data yang dicari, maka pencarian selesai dan data ditemukan pada lokasi yang dimiliki oleh variabel posisi.

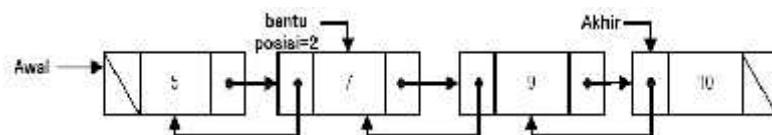
(2) Pencarian dimana data yang dicari tidak ditemukan

Kasus : data yang dicari adalah 20.

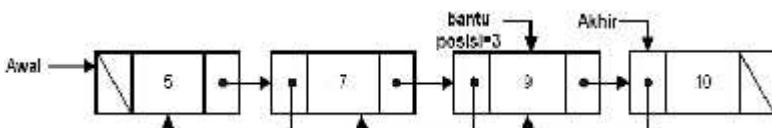
- a. Isi bantu dengan pointer data awal dan posisi diisi 1



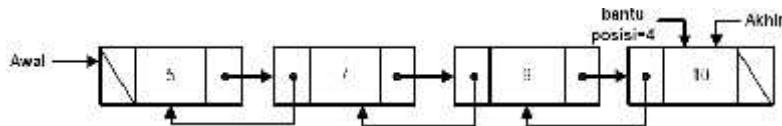
- b. Jika data info yang ditunjuk oleh pointer bantu tidak sama dengan yang dicari, maka bantu pindah ke data berikutnya dan variabel posisi ditambah 1.



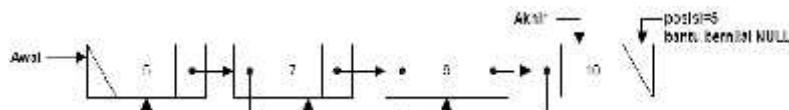
- c. Karena info yang ditunjuk oleh bantu belum sama dengan yang dicari, maka bantu pindah lagi ke data berikutnya dan variabel posisi ditambah 1



- d. Karena info yang ditunjuk oleh bantu belum sama dengan yang dicari, maka bantu pindah lagi ke data berikutnya dan variabel posisi ditambah 1



- e. Karena info yang ditunjuk oleh bantu belum sama dengan yang dicari, maka bantu pindah lagi ke data berikutnya dan variabel posisi ditambah 1



- f. Karena bantu bernilai NULL maka pencarian tidak perlu dilanjutkan lagi karena itu berarti data tidak ditemukan.

- g. Implementasi program dalam program

Dari langkah-langkah di atas, maka dapat diimplementasikan ke dalam bahasa Pascal

```
function pencarian(awal : point; data : integer) : integer;
var
    bantu : point;
    posisi : integer;
begin
    if(awal = Nil) then
    begin
        writeln('Data Kosong');
        exit;
    end else
    begin
        posisi=1;
        bantu=awal;
        while((bantu^.info <> data) and(bantu <> Nil)) do
        begin
            posisi := posisi + 1;
            bantu:=bantu^.kanan;
        end;
        if(bantu <> Nil) then
            return;
        else
            exit;
    end;
end;
```

3.3. Latihan

1. Buatlah sebuah *Linked list* non circular yang berisi nim anda dan nama lengkap anda.
2. Buat fungsi untuk menambahkan node single *Linked list* non circular dimana tiap node mengandung informasi nim dan nama. Peletakan posisi node urut berdasar nim secara ascending, jadi bisa tambah depan, belakang maupun tambah di tengah.
Isikan data nim dan nama lengkap teman sebelah kiri dan kanan anda!!!
3. Buatlah fungsi untuk menampilkan data 3 buah node yang telah anda bentuk sebelumnya. Contoh tampilan

NIM	Nama Lengkap
99071001	Malecita
00071022	Nohan
05071078	Salsabila

4. Buatlah fungsi untuk mencari nama yang telah diinputkan dengan menggunakan NIM.

Contoh tampilan:

Inputkan nim yang dicari = 05071078

Nama yang tercantum Salsabila

5. Buatlah sebuah fungsi untuk menghapus nim yang diinputkan oleh user.

Contoh tampilan:

NIM yang mau dihapus = 00071022

NIM dengan nama Andrew S ditemukan dan telah dihapus

BAB IV

STACK (TUMPUKAN)

Stack bisa diartikan sebagai suatu kumpulan data yang seolah-olah ada data yang diletakkan di atas data yang lain. Di dalam stack, kita dapat menambahkan/menyisipkan dan mengambil/menghapus data melalui ujung yang sama yang disebut sebagai puncak *stack* (*top of stack*). Stack mempunyai sifat LIFO (*Last In First Out*), artinya data yang terakhir masuk adalah data yang pertama keluar.

Contoh dalam kehidupan sehari-hari adalah tumpukan piring di sebuah restoran yang tumpukannya dapat ditambah pada bagian paling atas dan jika mengambilnya pun dari bagian paling atas pula. Lihat gambar 4.1.

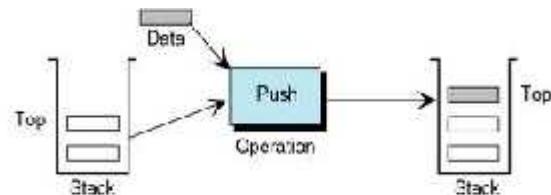


Gambar 4.1. Contoh stack
(sumber : www.cateringjakarta.tumblr.com)

Ada 2 operasi paling dasar dari *stack* yang dapat dilakukan, yaitu :

1. Operasi **push**

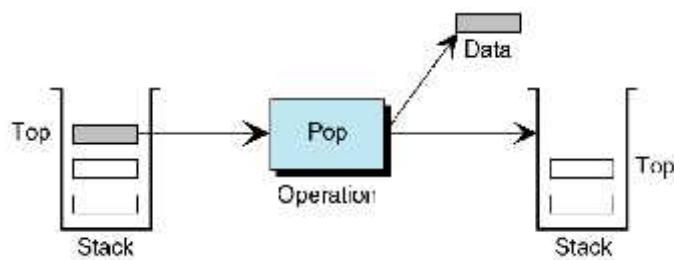
yaitu operasi menambahkan elemen pada urutan terakhir (paling atas). Dengan syarat tumpukan tidak dalam kondisi penuh, jika penuh penuh maka terjadi overflow



Gambar 4.2. Operasi Push
(Sumber : www.algonicox.blogspot.com)

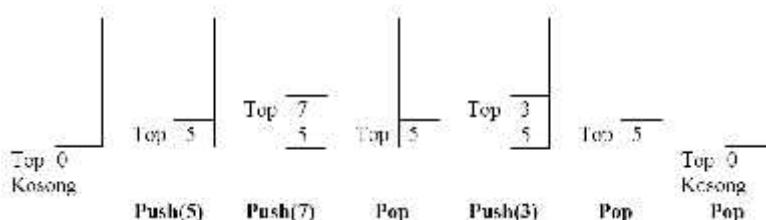
2. Operasi **pop**

yaitu operasi mengambil sebuah elemen data pada urutan terakhir dan menghapus elemen tersebut dari *stack*. Dengan syarat tumpukan tidak dalam kondisi kosong, jika kosong akan terjadi underflow



Gambar 4.3. Operasi POP
(Sumber : www.algonicox.blogspot.com)

Contoh : ada sekumpulan perintah *stack* yaitu *push(5)*, *push(7)*, *pop*, *push(3)*, *pop*. Jika dijalankan, maka yang akan terjadi adalah :



Gambar 4.4. Contoh operasi dasar pada stack

Selain operasi dasar *stack* (*push* dan *stack*), ada lagi operasi lain yang dapat terjadi dalam *stack* yaitu :

1. Proses **deklarasi** yaitu proses pendeklarasian *stack*.
2. Proses **IsEmpty** yaitu proses pemeriksaan apakah *stack* dalam keadaan kosong.
3. Proses **IsFull** yaitu proses pemeriksaan apakah *stack* telah penuh.
4. Proses **inisialisasi** yaitu proses pembuatan *stack* kosong, biasanya dengan pemberian nilai untuk top.

Representasi *stack* dalam pemrograman, dapat dilakukan dengan 2 cara yaitu :

1. Representasi *stack* dengan array
2. Representasi *stack* dengan single linked list

4.1. Representasi Stack Dengan Array

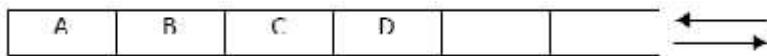
Bentuk penyajian *stack* menggunakan tipe data array sebenarnya kurang tepat karena banyaknya elemen dalam array bersifat statis, sedangkan jumlah elemen *stack* sangat bervariasi atau dinamis. Meskipun demikian, array dapat digunakan untuk penyajian *stack*, tetapi dengan anggapan bahwa banyaknya elemen maksimal suatu *stack* tidak melebihi batas maksimum banyaknya array. Pada suatu saat ukuran *stack* akan sama dengan ukuran array. Bila diteruskan

menambah data, maka akan terjadi *overflow*. Oleh karena itu, perlu ditambahkan data untuk mencatat posisi ujung *stack*.

Ada dua jenis bentuk *stack* menggunakan array, yaitu single *stack* dan double *stack*.

4.1.1. Single Stack

Single *stack* dapat dianalogikan dengan sebuah wadah yang diisi benda melalui satu jalan keluar dan masuk.



Gambar 4.5. Ilustrasi Single Stack

Representasi *stack* dengan menggunakan array dengan maksimal data 9 dapat dilihat pada gambar 4.6

max	→ 9	28
8		45
7		27
6		70
5		60
4		50
3		15
2		25
1		30
0		5

A red arrow points to the value 5 at index 0, with the label "Menyimpan posisi TOS". A blue arrow points to the value 9 above the array, with the label "→ 9".

Gambar 4.6. Representasi stack menggunakan array

Dari gambar 4.6. terlihat bahwa indek array ke-0 digunakan untuk menyimpan posisi Top Of Stack (TOS). Karena isi dari array ke-0 adalah 5 maka untuk TOS = 5, sehingga isi *stack* sebenarnya adalah : **60, 50, 15, 25, 30**. Yaitu posisi array ke-1 sampai dengan ke-5.

Apabila dilakukan operasi *Push(stack,IB)*, maka proses yang terjadi adalah :

1. Tambahkan nilai TOP
2. masukkan nilai IB pada posisi TOP

Sebagai contoh apabila dilakukan operasi: **Push(stack,20)**, maka

1. TOP = 6, posisi 6 dari 70 berubah 20, dan
2. Isi *stack* menjadi : **20, 60, 50, 15, 25, 30**

Apabila dilakukan operasi *Pop(stack,IB)*, maka proses yang terjadi adalah :

1. Ambil elemen pada posisi TOP
2. Turunkan nilai TOP

Sebagai contoh apabila dilakukan operasi: **Pop(stack,X)**, maka

1. Maka posisi 6 diambil dan
2. TOP = 5

Operasi-operasi *stack* pada single *stack* secara lengkap adalah sebagai berikut :

(1) Pendeklarasian stack

Proses pendeklarasian *stack* adalah proses pembuatan struktur *stack* dalam memori. Suatu *stack* memiliki beberapa bagian yaitu

- a. **top** yang menunjuk posisi data terakhir (TOP)
- b. **elemen** yang berisi data yang ada dalam *stack*. Bagian ini lah yang berbentuk array.
- c. **maks_elelen** yaitu variable yang menunjuk maksimal banyaknya elemen dalam *stack*.

Dalam bahasa Pascal, pendeklarasiannya adalah :

```
Const
      Maks_elelen := 100;
Type
      tstack = record
          Elemen : array[1..maks_elelen] of integer;
          Top    : integer;
          Maks   : integer;
      End;
Var
      Stack : tstack;
```

(2) Inisialisasi

Inisialisasi *stack* adalah proses pembuatan suatu *stack* kosong. Proses inisialisasi untuk *stack* yang menggunakan array adalah dengan mengisi nilai field top dengan 0 (nol)

Implementasinya dalam bahasa Pascal adalah sebagai berikut

```
Procedure init(var stack : tstack);
Begin
    Stack.top= 0;
    Stack.maks =maks_elelen;
End;
```

(3) Operasi IsEmpty

Operasi ini digunakan untuk memeriksa apakah *stack* dalam keadaan kosong. Operasi ini penting dilakukan dalam proses pop. Ketika suatu *stack* dalam keadaan kosong, maka proses pop tidak bisa dilakukan. Operasi ini dilakukan hanya dengan memeriksa field top. Jika top bernilai 0, maka berarti *stack* dalam keadaan *empty* (kosong) yang akan menghasilkan nilai true (1) pada function IsEmpty dan jika tidak berarti *stack* mempunyai isi dan menghasilkan nilai false (0) pada function IsEmpty

Implementasi Function IsEmpty dalam bahasa Pascal adalah sebagai berikut

```
Function IsEmpty(stack : tstack) : Boolean;
Begin
    If stack.top = 0 then
        IsEmpty := true;
    Else
        IsEmpty := false;
End;
```

(4) Operasi IsFull

Operasi ini berguna untuk memeriksa keadaan *stack* apakah sudah penuh atau belum. Operasi ini akan menghasilkan nilai true (1) jika *stack* telah penuh dan akan menghasilkan nilai false (0) jika *stack* masih bisa ditambah. Operasi ini akan memberikan nilai true (1) jika field top sama dengan field maks_elelen

Implementasinya Function IsFull dalam bahasa Pascal adalah sebagai berikut

```
Function IsFull(stack : tstack) : Boolean;
Begin
    If stack.top = maks_elelen then
        IsFull := true;
    Else
        IsFull := false;
End;
```

(5) Operasi Push

Operasi **push** adalah operasi dasar dari *stack*. Operasi ini berguna untuk menambah suatu elemen data baru pada *stack* dan disimpan pada posisi top yang akan mengakibatkan posisi top akan berubah. Langkah operasi ini adalah :

- Periksa apakah *stack* penuh (isfull). Jika bernilai false/0 (tidak penuh) maka proses *push* dilaksanakan dan jika pemeriksaan ini bernilai true/1 (*stack* penuh), maka proses *push* digagalkan.
- Proses *push*-nya sendiri adalah dengan menambah field top dengan 1, kemudian elemen pada posisi top diisi dengan elemen data baru.
- Untuk lebih jelas, perhatikan lagi gambar 4.4. mengenai representasi *stack* dengan array.
- Implementasi operasi *Push* dalam bahasa Pascal adalah sebagai berikut

```
Procedure push(var stack : tstack; baru : integer)
Begin
    if(IsFull(stack)= false) then
    begin
        stack.top := stack.top + 1;
        stack.elemen[stack.top]:= baru;
    end else
        writeln('Stack Full. Push Gagal.');
    end;
```

(6) Operasi Pop

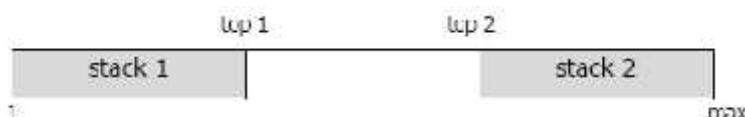
Operasi **pop** adalah salah satu operasi paling dasar dari *stack*. Operasi ini berguna untuk mengambil elemen terakhir (top) dan kemudian menghapus elemen tersebut sehingga posisi top akan berpindah. Operasi ini biasanya menghasilkan nilai sesuai data yang ada di top seperti diperlihatkan pada Gambar 4.6. Langkah operasi pop pada *stack* yang menggunakan array adalah

- terlebih dahulu memeriksa apakah *stack* dalam keadaan keadaan kosong, jika tidak kosong maka data diambil pada posisi yang ditunjuk oleh posisi top, kemudian simpan dalam variable baru dengan nama **data**, kemudian posisi top dikurangi 1,
- kemudian nilai pada variable **data** dikembalikan ke function.
- Implementasi operasi POP dalam bahasa Pascal adalah sebagai berikut

```
Procedure pop(var stack: tstack; var data : integer);
Begin
  if(IsEmpty(stack)= false) then
    begin
      data=stack.elemen[stack.top];
      stack.top := stack.top - 1;
    end
    else
      writeln('Stack Empty. Pop Gagal.');
  end;
```

4.1.2. Double Stack

Double stack merupakan bentuk pengembangan single *stack* dengan maksud untuk menghemat memori. Dalam *double stack* terdapat dua *stack* dalam satu array. *Stack 1* bergerak ke kanan dan *stack 2* bergerak ke kiri. *Double stack* dikatakan penuh apabila puncak *stack 1* bertemu dengan puncak *stack 2*.



Gambar 4.7. Ilustrasi double stack

Operasi-operasi *stack* pada single *stack* secara lengkap adalah sebagai berikut :

(1) Pendeklarasian stack

- Top1** yang menunjuk posisi data terakhir (top1)
- Top2** yang menunjuk posisi data terakhir (top2)
- elemen** yang berisi data yang ada dalam *stack*. Bagian ini lah yang berbentuk array.
- maks_eleme**n yaitu variable yang menunjuk maksimal banyaknya elemen dalam *stack*.

Dalam bahasa Pascal, pendeklarasiannya adalah sebagai berikut

```

Const
    Maks_elelen := 100;
Type
    tstack = record
        Elemen : array[1..maks_elelen] of integer;
        Top1   : integer;
        Top2   : integer
    End;
Var
    Stack : tstack;

```

(2) Inisialisasi

Inisialisasi *double stack* adalah proses pembuatan suatu *double stack* kosong. Proses inisialisasi untuk *double stack* yang menggunakan array adalah dengan mengisi nilai field top1 dengan 0 (nol) dan mengisi nilai field top2 dengan nilai maksimum ditambah 1. Implementasi proses inisialisasi dalam bahasa Pascal adalah sebagai berikut

```

Procedure init(var stack : tstack);
Begin
    Stack.top1= 0;
    Stack.top2 =maks_elelen + 1;
End;

```

(3) Operasi IsEmpty

Operasi ini digunakan untuk memeriksa apakah *double stack* dalam keadaan kosong. Operasi ini penting dilakukan dalam proses pop. Ketika suatu *double stack* dalam keadaan kosong, maka proses pop tidak bisa dilakukan. Operasi ini dilakukan hanya dengan memeriksa field top1 dan top2. Jika top1 bernilai 0 dan top2 bernilai maks_elelen ditambah 1, maka berarti *stack* dalam keadaan *empty* (kosong) yang akan menghasilkan nilai true (1) dan jika tidak berarti *stack* mempunyai isi dan menghasilkan nilai false (0). Implementasi operasi *IsEmpty* dalam bahasa Pascal adalah sebagai berikut

```

Function IsEmpty(stack : tstack; nostack: integer) : Boolean;
Begin
    Case nostack of
        1 : If (stack.top1 = 0) then
            IsEmpty := true;
        Else
            IsEmpty := false;
        2 : if (stack.top2 = maks_elelen + 1) then
            IsEmpty := true;
        Else
            IsEmpty := false;
    Else
        Writeln('Nomor stack salah');
    End;
End;

```

(4) Operasi IsFull

Operasi ini berguna untuk memeriksa keadaan *stack* apakah sudah penuh atau belum. Operasi ini akan menghasilkan nilai true (1) jika *stack* telah penuh dan akan menghasilkan nilai false (0) jika *stack* masih bisa ditambah. Operasi ini akan memberikan nilai true (1) jika field top1 lebih besar dari field top2.

Implementasinya dalam bahasa Pascal adalah :

```
Function IsFull(stack : tstack) : Boolean;
Begin
    If stack.top1 >= stack.top2 then
        IsFull := true;
    Else
        IsFull := false;
End;
```

(5) Operasi Push

Operasi **push** berguna untuk menambah suatu elemen data baru pada salah satu *stack*.

Implementasinya dalam bahasa Pascal sebagai berikut

```
Procedure push(stack : tstack; nostack,baru : integer)
begin
    if (IsFull(stack) = false) then
        begin
            case nostack of
                1 : stack.top1 := stack.top1 + 1;
                      Stack.elemen[stack.top1]:=baru;
                2 : stack.top2 := stack.top2 - 1;
                      Stack.elemen[stack.top2]:=baru;
            Else
                writeln('Invalid PUSH...\\');
            end;
        end else
            writeln('Stack overflow...\\');
    end;
```

(6) Operasi Pop

Operasi **pop** berguna untuk mengambil elemen terakhir (top) dan kemudian menghapus elemen tersebut sehingga posisi top akan berpindah.

Implementasi procedure pop dalam bahasa Pascal adalah sebagai berikut

```

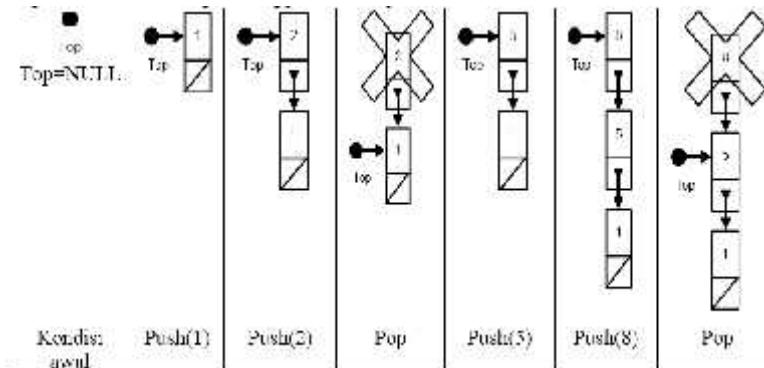
Procedure pop(stack: tstack; nostack: inetegr; var data : integer);
Begin
  if (IsEmpty(nostack) = false) then begin
    case nostack of
      1 : data := stack.elemen[stack.stack[top1]];
      Stack.top1:= stack.top1 - 1;
      2 : data := stack.elemen[stack.stack[top2]];
      Stack.top2 := stack.top2 + 1;
    else
      writeln('Invalid POP...');
    end
  end else
    writeln('Stack underflow..');
end;

```

4.2. Representasi Stack Dengan Single Linked List

Bentuk penyajian *stack* menggunakan single linked list sangat tepat karena banyaknya elemen dalam list bersifat dinamis, sedangkan jumlah elemen *stack* juga sangat bervariasi atau dinamis.

Representasi *stack* menggunakan single linked list dapat dilihat pada gambar 4.8.



Gambar 4.8. Representasi stack menggunakan single linked list
(Sumber : <http://forbiddens.wordpress.com/2010/06/17/stack-atau-tumpukan/>)

Operasi *stack* menggunakan single linked, meliputi operasi

(1) Pendeklarasian stack

Proses pendeklarasian *stack* menggunakan single linked list, hanya memerlukan suatu pointer yang menunjuk ke data terakhir. Setiap elemen linked list mempunyai 2 field yaitu datanya dan link yang menunjuk posisi terakhir sebelum proses *push* seperti terlihat pada gambar 4.9.



Gambar 4.9. Struktur untuk node stack

Pada representasi *stack* menggunakan single linked list TOP selalu berada di penunjuk pointer awal.

Dalam bahasa Pascal, pendeklarasiannya adalah :

```
Type
  pstack = point
  point = record
    data : integer;
    link : pstack;
  End;
Var
  Stack : pstack;
```

(2) Inisialisasi

Proses inisialisasi untuk *stack* yang menggunakan single linked list adalah dengan mengisi nilai pointer *stack* dengan Nil.

Implementasinya dalam bahasa Pascal adalah :

```
Procedure init(stack : pstack);
Begin
  Stack = nil;
End;
```

(3) Operasi IsEmpty

Operasi IsEmpty pada *stack* yang menggunakan single linked list adalah dengan memeriksa apakah pointer *stack* bernilai NIL. Jika *stack* bernilai NIL maka menandakan *stack* sedang keadaan empty (kosong) dan akan me-return-kan nilai 1 dan jika tidak Nil maka menandakan *stack* mempunyai isi (tidak kosong) sehingga operasi tersebut akan me-return-kan nilai false (0).

Implementasi dalam bahasa Pascal adalah :

```
Function IsEmpty(stack : pstack) : Boolean;
Begin
  If stack = nil then
    IsEmpty := true;
  Else
    IsEmpty := false;
End;
```

(4) Operasi IsFull

Karena dalam linked list bersifat dinamis, maka pengecekan isFull adalah dengan memeriksa apakah memori masih dapat digunakan untuk alokasi sebuah elemen *stack*. Jika alokasi dapat dilakukan, maka berarti memori masih belum penuh dan proses *push* dapat dilakukan. Tetapi jika alokasi memori gagal dilakukan, maka berarti memori penuh dan tidak bisa menambah lagi elemen *stack*.

Implementasinya dalam bahasa Pascal adalah :

```
Function IsFull(stack : pstack) : Boolean;
Begin
    If stack = maks_elelen then
        IsFull := true;
    Else
        IsFull := false;
End;
```

(5) Operasi Push

Operasi *push* pada *stack* yang menggunakan single linked list adalah sama dengan proses tambahawal pada operasi linked list. Langkah-langkahnya adalah :

- Periksa apakah memori penuh (*isfull*). Jika bernilai false/0 (tidak penuh) maka proses *push* dilaksanakan dan jika pemeriksaan ini bernilai true/1 (*stack* penuh), maka proses *push* digagalkan.
- Proses *push*-nya sendiri adalah dengan cara mengalokasikan suatu elemen linked list (disebut variable baru), kemudian periksa jika *stack* dalam keadaan kosong maka pointer yang menunjuk ke awal *stack* diisi dengan pointer baru, dan jika dengan menambah field *top* dengan 1, kemudian elemen pada posisi *top* diisi dengan elemen data baru. Tetapi jika pointer penunjuk *stack* sudah menunjuk ke suatu data, maka sambungkan field pointer **link** (penunjuk ke data sebelumnya) dari pointer baru ke pointer penunjuk posisi akhir *stack* (*top*) dan kemudian pindahkan pointer penunjuk posisi akhir *stack* ke pointer baru.

Untuk lebih jelas perhatikan kembali gambar 4.8 mengenai representasi *stack* dengan linked list.

Implementasinya dalam bahasa Pascal adalah :

```
Procedure push(stack : pstack; isi : integer)
var
    Baru : pstack;
begin
    if(isfull(stack)=false) then
    begin
        new(baru);
        baru^.link=Nil;
        baru^.data=isi;
        if(isempty(stack)=true) then
            stack=baru;
        else
            begin
                baru^.link := stack;
                stack := baru;
            end;
    end
    else
        writeln('Memory Full. Push Gagal');
end;
```

(6) Operasi Pop

Langkah operasi pop pada *stack* yang menggunakan single linked list adalah sama dengan proses hapus awal pada operasi single linked list. Prosesnya adalah :

- Periksa apakah *stack* kosong (*isempty*), jika kosong maka proses pop tidak bisa dilakukan. Jika *stack* tidak kosong maka proses pop dijalankan.
- Proses pop-nya sendiri adalah mengambil elemen yang ditunjuk oleh pointer *stack* kemudian simpan dalam variable data. Kemudian buat variable pointer **bantu** yang diisi dengan pointer penunjuk *stack* yang nantinya akan dihapus dari memori. Kemudian pointer penunjuk *stack* dipindahkan ke posisi yang ditunjuk oleh field pointer bawah dari variable **bantu**.

Implementasi operasi ini dalam bahasa Pascal adalah :

```
Procedure pop(stack: tstack; var elemen : integer);
var
  bantu : PStack;
begin
  if(isempty(stack)= false) then
    begin
      elemen=stack^.data;
      bantu=stack;
      stack=bantu^.link;
      dispose(bantu);
    end else
      writeln('Stack dalam kondisi kosong');
  end;
```

4.3. Implementasi Stack Untuk Mengkonversi Bilangan Desimal ke Bilangan Biner

Contoh nyata implementasi *stack* adalah proses pengkonversian dari bilangan decimal (basis 10) ke bilangan biner (basis 2).

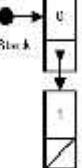
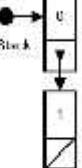
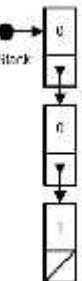
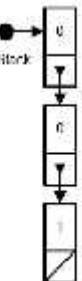
Algoritmanya adalah :

- Ambil sisa pembagian variable **bilangan** dengan angka **2**, kemudian simpan dalam variable **sisa**. Kemudian simpan isi variable **sisa** ke dalam *stack*.
- Bagi variable **bilangan** dengan angka **2**.
- Ulangi langkah 1 dan 2 selama **bilangan** tidak 0. Jika variable **bilangan** telah bernilai 0 maka lanjutkan ke langkah 4,
- Lakukan perulangan untuk langkah 5 dan 6 selama *stack* masih mempunyai isi (tidak kosong).
- ambil (pop) nilai yang ada di *stack* simpan di variable **data**.

(6) Tulis isi variable data ke layar .

(7) Selesai.

Contoh: operasi konversi dari decimal ke biner dengan variable bilangan yang akan dikonversi adalah 25 (sumber : <http://nafnafi.files.wordpress.com/2010/10/stack2.pdf>).

Operasi	Representasi Array	Representasi Single Linked List																
Kondisi Awal	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>4</td></tr> <tr><td>?</td><td>3</td></tr> <tr><td>?</td><td>2</td></tr> <tr><td>?</td><td>1</td></tr> </table> Operasi Stack : inisialisasi(&stack)	?	10	?	9	?	...	?	...	?	4	?	3	?	2	?	1	 Stack
?	10																	
?	9																	
?	...																	
?	...																	
?	4																	
?	3																	
?	2																	
?	1																	
	Top=0 Maks_Elemen=10	Pointer Stack = NULL																
Bilangan = 25, Pembagian : $25 / 2$ Sisa = 1 Bilangan = 12 Operasi Stack : Push(&stack,1)	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>4</td></tr> <tr><td>?</td><td>3</td></tr> <tr><td>?</td><td>2</td></tr> <tr><td>Top</td><td>1</td></tr> </table> 	?	10	?	9	?	...	?	...	?	4	?	3	?	2	Top	1	 Stack
?	10																	
?	9																	
?	...																	
?	...																	
?	4																	
?	3																	
?	2																	
Top	1																	
	Top=1 Maks_Elemen=10																	
Bilangan = 12, Pembagian : $12 / 2$ Sisa = 0 Bilangan = 6 Operasi Stack : Push(&stack,0)	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>4</td></tr> <tr><td>?</td><td>3</td></tr> <tr><td>Top</td><td>0</td></tr> <tr><td></td><td>1</td></tr> </table> 	?	10	?	9	?	...	?	...	?	4	?	3	Top	0		1	 Stack
?	10																	
?	9																	
?	...																	
?	...																	
?	4																	
?	3																	
Top	0																	
	1																	
	Top=2 Maks_Elemen=10																	
Bilangan = 6 Pembagian : $6 / 2$ Sisa : 0 Bilangan : 3 Operasi Stack : Push(&stack,0)	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>?</td><td>4</td></tr> <tr><td>Top</td><td>0</td></tr> <tr><td>0</td><td>2</td></tr> <tr><td>1</td><td>1</td></tr> </table> 	?	10	?	9	?	...	?	...	?	4	Top	0	0	2	1	1	 Stack
?	10																	
?	9																	
?	...																	
?	...																	
?	4																	
Top	0																	
0	2																	
1	1																	
	Top=3 Maks_Elemen=10																	

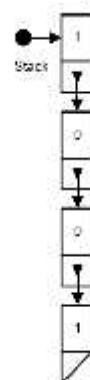
Bilangan = 3

Pembagian : 3 / 2
 Sisa : 1
 Bilangan : 1
 Operasi Stack :

Push(&stack,1)

?	10
?	9
?	...
Top	1
0	3
0	2
1	1

Top=4 Maks_Elemen=10



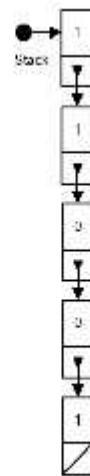
Bilangan = 1

Pembagian : 1 / 2
 Sisa : 1
 Bilangan : 0
 Operasi Stack :

Push(&stack,1)

?	10
?	9
?	...
Top	1
1	4
0	3
0	2
1	1

Top=5 Maks_Elemen=10



Operasi pembagian variable **bilangan** dengan angka 2 berakhir di posisi ini karena nilai variable **bilangan** sudah mencapai 0. Langkah selanjutnya adalah proses menampilkan data biner dari stack, sampai stack habis.

Periksa :

Stack Kosong ? Tidak

Operasi Stack :

Data=poy(&stack);

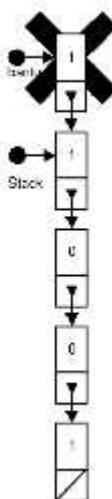
Menghasilkan Data=1

Tampilkan variable Data di layar

Di layar : 1

?	10
?	9
?	...
Top	1
0	3
0	2
1	1

Top=4 Maks_Elemen=10



<p>Periksa : Stack Kosong ? Tidak</p> <p>Opeasi Stack :</p> <table border="1"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>1</td><td>...</td></tr> </table> <p>Data=pop(&stack); Menghasilkan Data=1</p> <p>Tampilkan variable Data di layar</p> <p>Dilayari : 1 1</p>	?	10	?	9	?	...	1	...	<p>Top=3 Maks_Element=10 </p>
?	10								
?	9								
?	...								
1	...								
<p>Periksa : Stack Kosong ? Tidak</p> <p>Opeasi Stack :</p> <table border="1"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>1</td><td>...</td></tr> </table> <p>Data=pop(&stack); Menghasilkan Data=0</p> <p>Tampilkan variable Data di layar</p> <p>Dilayari : 1 1 0</p>	?	10	?	9	?	...	1	...	<p>Top=2 Maks_Element=10 </p>
?	10								
?	9								
?	...								
1	...								
<p>Periksa : Stack Kosong ? Tidak</p> <p>Opeasi Stack :</p> <table border="1"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>1</td><td>...</td></tr> </table> <p>Data=pop(&stack); Menghasilkan Data=0</p> <p>Tampilkan variable Data di layar</p> <p>Dilayari : 1 1 0 0</p>	?	10	?	9	?	...	1	...	<p>Top=1 Maks_Element=10 </p>
?	10								
?	9								
?	...								
1	...								
<p>Periksa : Stack Kosong ? Tidak</p> <p>Opeasi Stack :</p> <table border="1"> <tr><td>?</td><td>10</td></tr> <tr><td>?</td><td>9</td></tr> <tr><td>?</td><td>...</td></tr> <tr><td>1</td><td>...</td></tr> </table> <p>Data=pop(&stack); Menghasilkan Data=1</p> <p>Tampilkan variable Data di layar</p> <p>Dilayari : 1 1 0 0 1</p> <p>Periksa : Stack Kosong ? Ya</p> <p>Pembuktian :</p>	?	10	?	9	?	...	1	...	<p>Top=0 Maks_Element=10 </p> <p>Stack = NULL</p> <p>Proses pop stack selesai karena stack telah kosong. Itu berarti semua bilangan biner yang disimpan dalam stack telah ditampilkan ke layar.</p> <p>$1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 16 + 8 + 1 = 25$.</p>
?	10								
?	9								
?	...								
1	...								

4.4. Latihan

1. Sebutkan 4 (empat) operasi pada *stack* dan beri contoh-contohnya.
2. Pada sebuah *stack*, kapan posisi TOP = BOTTOM ?
3. Buatlah program kalkulator sederhana (pengoperasian 2 bilangan aritmatika sederhana)! Operasi aritmatika tersebut (dalam notasi infiks), harus diubah kedalam bentuk notasi postfiks.

Contoh:

INFIKS	POSTFIKS
$3 + 7$	$3 7 +$
$2 - 9$	$2 9 -$
$8 * 7$	$8 7 *$
$9 / 3$	$9 3 /$

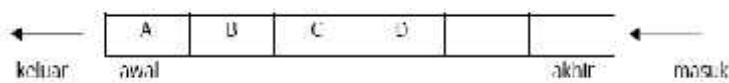
Misal algoritma: $3 7 +$

indeks	stack	process
2	+	add
1	7	<i>push operand</i>
0	3	<i>push operand</i>

BAB V

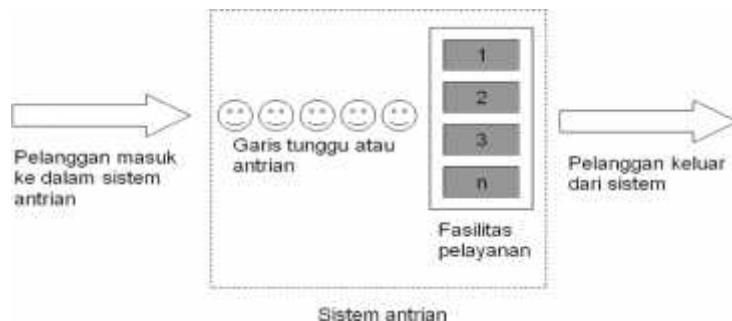
QUEUE (ANTRIAN)

Secara harfiah *queue* dapat diartikan sebagai antrian. *Queue* merupakan kumpulan data dengan penambahan data hanya melalui satu sisi, yaitu belakang (*tail*) dan penghapusan data hanya melalui sisi depan (*head*). Berbeda dengan *stack* yang bersifat LIFO maka *queue* bersifat FIFO(*First In First Out*), yaitu data yang pertama masuk akan keluar terlebih dahulu dan data yang terakhir masuk akan keluar terakhir. Berikut ini adalah gambaran struktur data *queue*.



Gambar 5.1. Ilustrasi queue

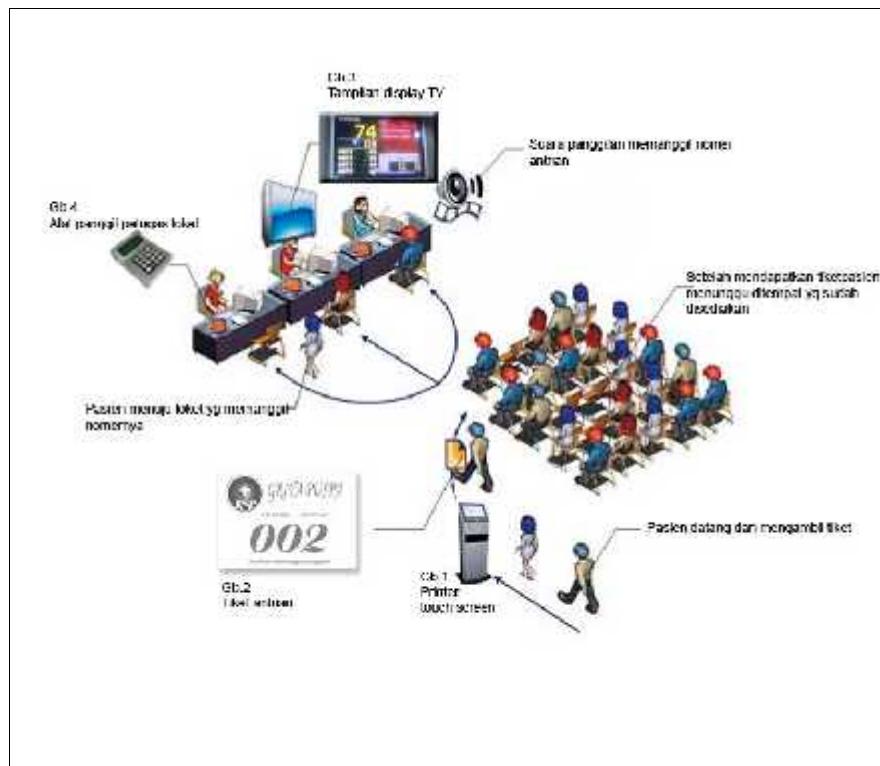
Elemen yang pertama kali masuk ke dalam *queue* disebut elemen depan (*front/head of queue*), sedangkan elemen yang terakhir kali masuk ke *queue* disebut elemen belakang (*rear/tail of queue*). Perbedaan antara *stack* dan *queue* terdapat pada aturan penambahan dan penghapusan elemen. Pada *stack*, operasi penambahan dan penghapusan elemen dilakukan di satu ujung. Elemen yang terakhir kali dimasukkan akan berada paling dekat dengan ujung atau dianggap paling atas sehingga pada operasi penghapusan, elemen teratas tersebut akan dihapus paling awal, sifat demikian dikenal dengan LIFO. Pada *queue*, operasi tersebut dilakukan di tempat yang berbeda. Penambahan elemen selalu dilakukan melalui salah satu ujung, menempati posisi di belakang elemen-elemen yang sudah masuk sebelumnya atau menjadi elemen paling belakang. Sedangkan penghapusan elemen dilakukan di ujung yang berbeda, yaitu pada posisi elemen yang masuk paling awal atau elemen terdepan. Sifat yang demikian dikenal dengan FIFO.



Gambar 5.2. Contoh model antian

Contoh antrian dalam kehidupan sehari – hari :

- Mobil yang antri membeli karcis di pintu jalan tol akan membentuk antrian
- Pembelian tiket pada loket tiket
- Antrian pasien

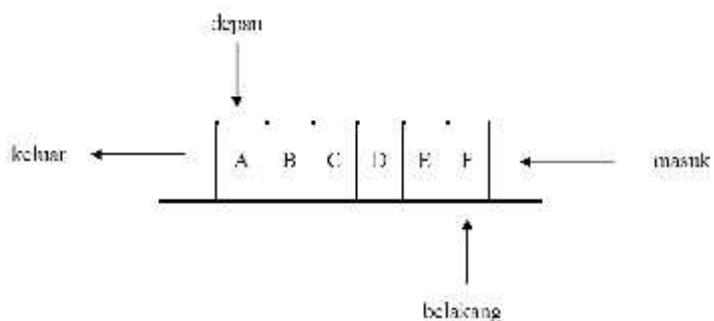


Gambar 5.3. Contoh model antian pasien

(sumber : <http://bopax.wordpress.com/2011/02/14/terjadinya-antrian/>)

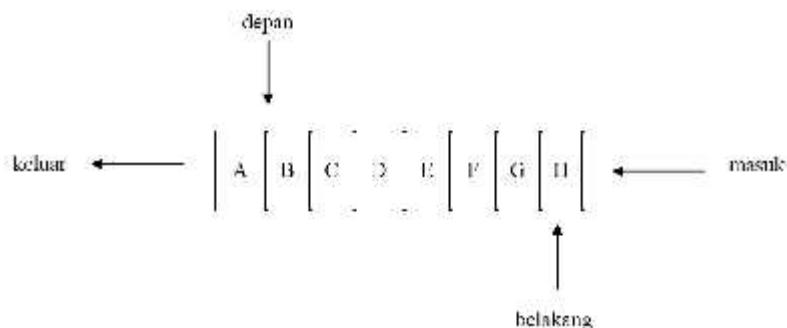
5.1. Representasi Queue Dengan Array

Disebut juga *queue* dengan model fisik, yaitu bagian depan *queue* selalu menempati posisi pertama array.



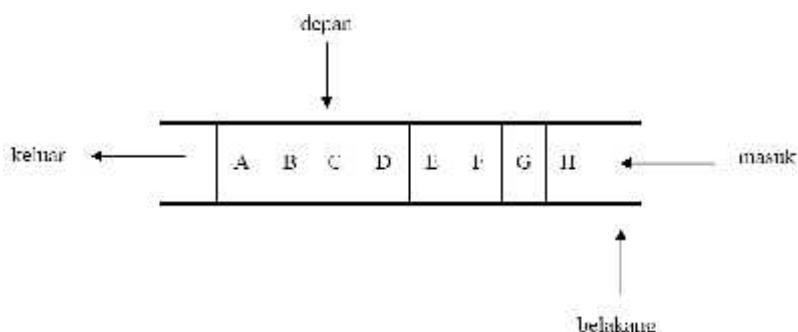
Gambar 5.4. Contoh antrian dengan 6 elemen

Gambar 5.4. di atas menunjukkan contoh penyajian antrian menggunakan larik. Antrian di atas berisi 6 elemen, yaitu A,B,C,D,E dan F. Elemen A terletak di bagian depan antrian dan elemen F terletak dibagian belakang antrian. Dengan demikian, jika ada elemen yang baru masuk, maka ia akan diletakkan disebelah kanan F (pada gambar diatas). Jika ada elemen yang akan dihapus, maka A akan dihapus lebih dahulu. Gambar 5.5. menunjukkan antrian di atas setelah berturut-turut dimasukkan G dan H.



Gambar 5.5. Ilustrasi penambahan elemen pada antrian

Gambar 5.6. menunjukkan antrian Gambar 5.5. setelah elemen A dan B dihapus.



Gambar 5.6. Ilustrasi penghapusan elemen pada antrian

Seperti halnya pada tumpukan, maka dalam antrian kita juga mengenal ada dua operasi dasar, yaitu menambah elemen baru yang akan kita tempatkan di bagian belakang antrian dan menghapus elemen yang terletak di bagian depan antrian. Disamping itu seringkali kita juga perlu melihat apakah antrian mempunyai isi atau dalam keadaan kosong.

Operasi penambahan elemen baru selalu bisa kita lakukan karena tidak ada pembatasan banyaknya elemen dari suatu antrian. Tetapi untuk menghapus elemen, maka kita harus melihat apakah antrian dalam keadaan kosong atau tidak. Tentu saja kita tidak mungkin menghapus elemen dari suatu antrian yang sudah kosong. Untuk menyajikan antrian, menggunakan larik, maka kita membutuhkan deklarasi antrian, misalnya, sebagai berikut :

```

Const nmax = 100;
Type typeinfo = ..... { bisa tipe data apa saja }
    Typearray = array[1..nmax] of typeinfo
    Typequeue = record
        Elemen : typearray
        Depan, Belakang : integer;
    End;
Var
    Antrian : typequeue;

```

Dalam deklarasi di atas, elemen antrian dinyatakan dalam tipe data integer. Peubah **Depan** menunjukkan posisi elemen pertama dalam larik; peubah **Belakang** menunjukkan posisi elemen terakhir dalam larik. Dengan menggunakan larik, maka kejadian *overflow* sangat mungkin, yakni jika antrian telah penuh, sementara kita masih ingin menambah terus. Dengan mengabaikan kemungkinan adanya *overflow*, maka penambahan elemen baru, yang dinyatakan oleh perubah X, bisa kita implementasikan dengan statemen :

```

Belakang := Belakang + 1 ;
Antrian[Belakang] := X;

```

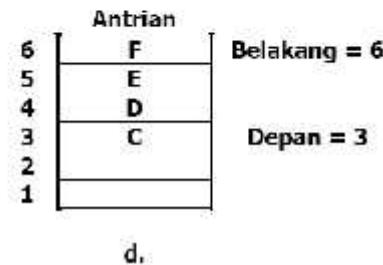
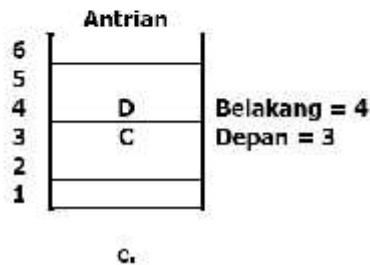
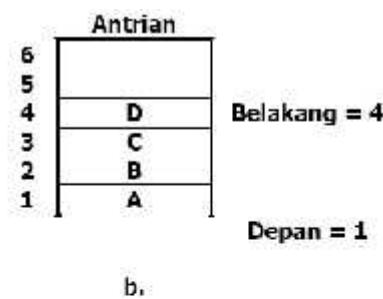
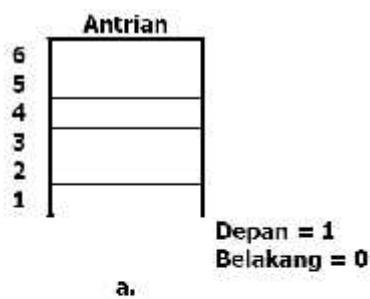
Operasi penghapusannya bisa diimplementasikan dengan :

```

X:= Antrian[Depan];
Depan := Depan + 1;

```

Pada saat permulaan, **Belakang** dibuat sama dengan 0 dan **Depan** dibuat sama dengan 1, dan antrian dikatakan kosong jika **Belakang < Depan**. Banyaknya elemen yang ada dalam antrian dinyatakan sebagai **Belakang – Depan + 1**.



Gambar 5.7. Ilustrasi penambahan dan pengurangan pada antrian

Sekarang marilah kita tinjau implementasi menambah dan menghapus elemen seperti diperlihatkan di atas. Gambar 5.7 menunjukkan larik dengan 6 elemen untuk menyajikan sebuah antrian (dalam hal ini Max_Elemen = 6). Pada saat permulaan (Gambar 5.7a), antrian dalam keadaan kosong. Pada gambar 5.7b terdapat 4 buah elemen yang telah ditambahkan. Dalam hal ini nilai **Depan** = 1 dan **Belakang** = 4. Gambar 5.7c menunjukkan antrian setelah dua elemen dihapus. Gambar 5.6d menunjukkan antrian setelah dua elemen baru ditambahkan. Banyaknya elemen dalam antrian adalah $6 - 3 + 1 = 4$ elemen. Karena larik terdiri dari 6 elemen, maka sebenarnya kita masih bisa menambah elemen lagi. Tetapi, jika kita ingin menambah elemen baru, maka nilai **Belakang** harus ditambah satu, menjadi 7. Padahal larik **Antrian** hanya terdiri dari 6 elemen, sehingga tidak mungkin ditambah lagi, meskipun sebenarnya larik tersebut masih kosong di dua tempat. Bahkan dapat terjadi suatu situasi dimana sebenarnya antriannya dalam keadaan kosong, tetapi tidak ada elemen yang bisa tambahkan kepadanya. Dengan demikian, penyajian di atas tidak dapat diterima.

5.1.1. Representasi Queue dengan penggeseran

Representasi *queue* menggunakan penggeseran dimaksudkan untuk menyelesaikan kasus yang dilustrasikan pada gambar 5.8. Caranya dengan mengubah prosedur untuk menghapus elemen, sedemikian rupa sehingga jika ada elemen yang dihapus, maka semua elemen lain digeser sehingga antrian selalu dimulai dari **Depan** = 1. Dengan cara ini, maka sebenarnya perubah **Depan** ini tidak diperlukan lagi hanya perubah **Belakang** saja yang diperlukan, karena nilai **Depan** selalu sama dengan 1.

Queue dengan linear array menggunakan penggeseran secara umum dapat dideklarasikan sebagai berikut:

```

Const nmax = 100;
Type typeinfo = ..... { bisa tipe data apa saja }
    Typearray      = array[1..nmax] of typeinfo
    Typequeue     = record
        Elemen       : typearray
        Belakang     : integer;
    End;
Var
    Queue : typequeue;
```

Berikut adalah aplikasi untuk operasi *queue* secara lengkap dengan menggunakan model penggeseran

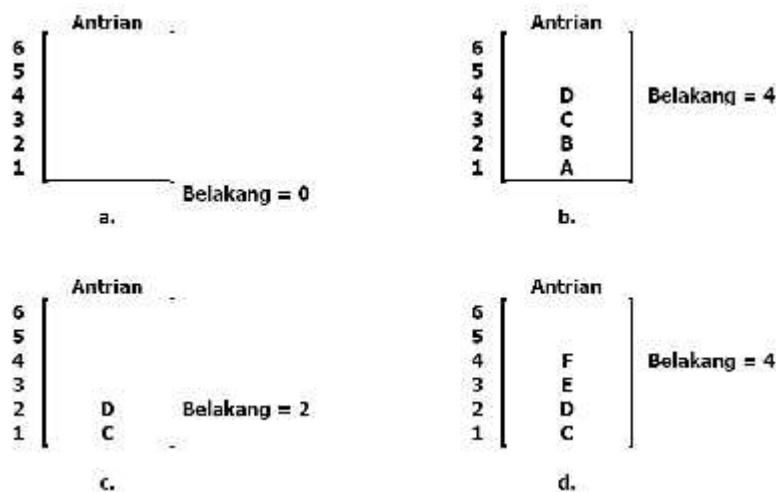
```

Procedure Buatqueue(var queue : typequeue)
Begin
    Queue.belakang := 0;
End;
Function queuekosong(queue : typequeue) : Boolean;
Begin
    Queuekosong := (queue.belakang = 0);
End;
Function queuepenuh(queue : typequeue) : Boolean;
Begin
    Queuepenuh := (queue.belakang = nmax);
End;
Procedure enqueue(var queue : typequeue; IB : typeinfo)
Begin
    If not(queuepenuh(Queue)) then
        Begin
            Queue.belakang := queue.belakang + 1;
            Queue.info[queue.belakang] := IB;
        End;
    End;

Procedure dequeue(var queue : typequeue; var infodec : typeinfo)
Var I : integer;
Begin
    If not(queuekosong(Queue)) then
        Begin
            Infodec := queue.elemen[1];
            For I := 1 to (queue.belakang -1) do
                Queue.elemen[I] := Queue.elemen[I + 1];
            Queue.belakang := queue.belakang - 1;
        End;
    End;

```

Dalam hal ini antrian kosong dinyatakan sebagai nilai **Belakang** = 0. Gambar 5.8. menunjukkan ilustrasi penambahan dan penghapusan elemen pada sebuah antrian dengan penggeseran elemen.



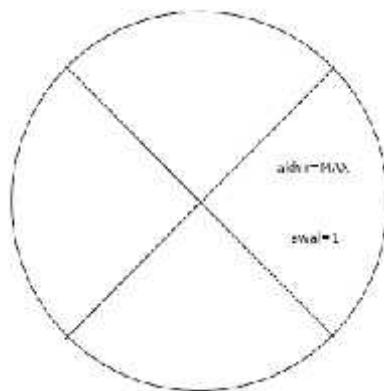
Gambar 5.8. Ilustrasi penambahan dan pengurangan pada antrian menggunakan penggeseran

Cara ini kelihatannya sudah memecahkan persoalan yang kita miliki. Tetapi jika kita tinjau lebih lanjut, maka ada sesuatu yang harus dibayar lebih mahal, yaitu tentang penggeseran elemen itu sendiri. Jika kita mempunyai antrian dengan 1000 elemen, maka waktu terbanyak yang dihabiskan sesungguhnya hanya untuk melakukan proses penggeseran. Hal ini tentu saja sangat tidak efisien. Dengan melihat gambar-gambar ilustrasi, proses penambahan dan penghapusan elemen antrian sebenarnya hanya mengoperasikan sebuah elemen, sehingga tidak perlu ditambah dengan sejumlah operasi lain yang justru menyebabkan tingkat ketidak-efisienan bertambah besar.

5.1.2. Queue melingkar

Pemecahan kasus pada sub bab 5.1.1. adalah dengan memperlakukan larik yang menyimpan elemen antrian sebagai larik yang memutar (*circular*), bukan lurus (*straight*).

Array melingkar (*circular array*), artinya array dapat diakses mulai dari sembarang indeks (indeks awal) ke arah indeks terakhir (maksimum array), lalu memutar ke indeks pertama hingga kembali ke indeks awal. Circular array adalah array yang dibuat seakan-akan merupakan sebuah lingkaran dengan titik awal dan titik akhir saling bersebelahan jika array tersebut masih kosong. Jumlah data yang dapat ditampung oleh array ini adalah besarnya ukuran array dikurangi 1. Misalnya besar array adalah 8, maka jumlah data yang dapat ditampung adalah 7.

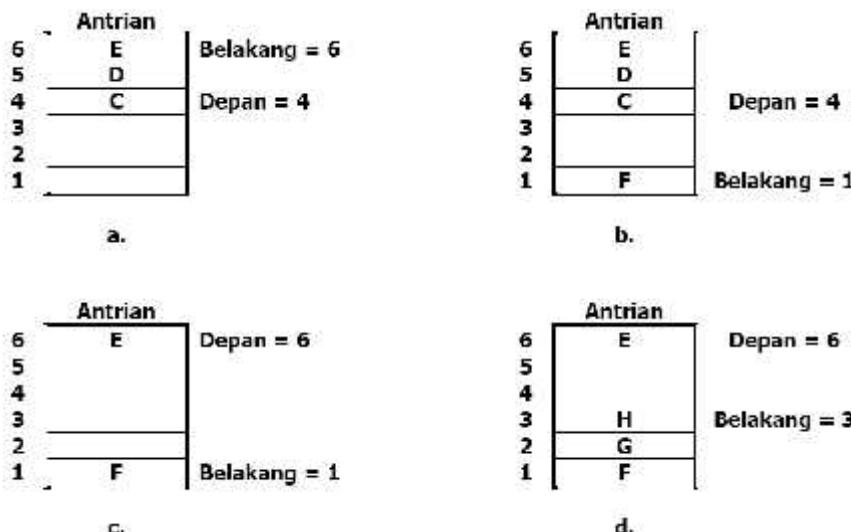


Gambar 5.9. Ilustrasi circular array

Dengan circular array, meskipun posisi terakhir telah terpakai, elemen baru tetap dapat ditambahkan pada posisi pertama jika posisi pertama dalam keadaan kosong.

Perhatikan contoh berikut, larik digambarkan mendatar untuk lebih mempermudah pemahaman. Gambar 5.10a. Menunjukkan antrian telah terisi dengan 3 elemen pada posisi ke 4, 5 dan 6 (dalam hal ini **Max_Elemen** = 6). Gambar 5.10b menunjukkan antrian setelah ditambah dengan sebuah elemen baru, F. Bisa diperhatikan, bahwa karena larik hanya berisi 6 elemen, sedangkan gambar 5.9a posisi ke-6 telah diisi, maka elemen baru akan menempati posisi ke 1. Gambar 5.9c menunjukkan ada 2 elemen yang dihapus dari antrian, dan gambar 6d menunjukkan ada 2 elemen baru yang ditambahkan

ke antrian. Bagaimana jika elemen antrian tersebut akan dihapus lagi? Coba anda lakukan pelacakan ilustrasi tersebut.



Gambar 5.10. Ilustrasi penambahan dan pengurangan pada antrian menggunakan circular array

Dengan melihat pada ilustrasi di atas, kita bisa menyusun prosedur untuk menambah elemen baru ke dalam antrian. Prosedur di bawah ini didasarkan pada deklarasi berikut :

```
const Max_Elemen = 100;
type Antri = array[1..Max_Elemen] of char;
var Antrian : Antri;
Depan, Belakang : Integer;
```

Untuk awalan, maka kita tentukan nilai perubah **Depan** dan **Belakang** sebagai :

```
Depan := 0;
Belakang := 0;
```

Prosedur selengkapnya untuk menambahkan elemen baru adalah sebagai berikut :

```
Procedure TAMBAH (var Q : Antri; X : Char);
begin
  if Belakang := Max_Elemen then
    Belakang := 1
  else
    Belakang := Belakang + 1;
  if Depan = Belakang then
    writeln ('Antrian Sudah Penuh')
  else
    Q[Belakang] := X
end;
```

Untuk menghapus elemen, terlebih dahulu kita harus melihat apakah antrian dalam keadaan kosong atau tidak. Berikut disajikan satu fungsi untuk mencek keadaan antrian.

```

function KOSONG(Q : Antri) : boolean;
begin
    KOSONG := (Depan=Belakang);
end;

```

Dengan memperhatikan fungsi di atas, maka kita bisa menyusun fungsi lain untuk menghapus elemen, yaitu :

```

function HAPUS (var Q : Antri) : char;
begin
    if KOSONG(Q) then
        writeln('ANTRIAN KOSONG...')
    else
        HAPUS := Q[Depan];
    if Depan := Max_Elemen then
        Depan := 1
    Else
        Depan := Depan +1;
    end;
end;

```

Dengan memperhatikan program-program di atas, bisa kita lihat bahwa sebenarnya elemen yang terletak di depan antrian, menempati posisi (subskrip) ke **Depan** + 1 pada larik yang digunakan. Sehingga sesungguhnya, banyaknya elemen maksimum yang bisa disimpan adalah sebesar **Max_Elemen** – 1, kecuali pada saat permulaan.

5.2. Representasi Queue Dengan Linked List

Implementasi antrian secara dinamis dilakukan dengan menggunakan tipe data *pointer*. Pada dasarnya sama dengan implementasi list pada bab III dimana pada *queue* dikenal sebagai sistem FIFO (first in first out).

Operasi yang umum pada *Queue*:

(1) Buat_queue

Mendeklarasikan *queue* yg kosong/menginisialisasi *queue* yg kosong

(2) Enqueue

Menambah elemen pada posisi paling belakang

(3) Dequeue

Mengeluarkan elemen pada posisi paling depan

(4) Rear

melihat elemen paling belakang dari antrian

(5) Front

Melihat elemen terdepan dari antrian

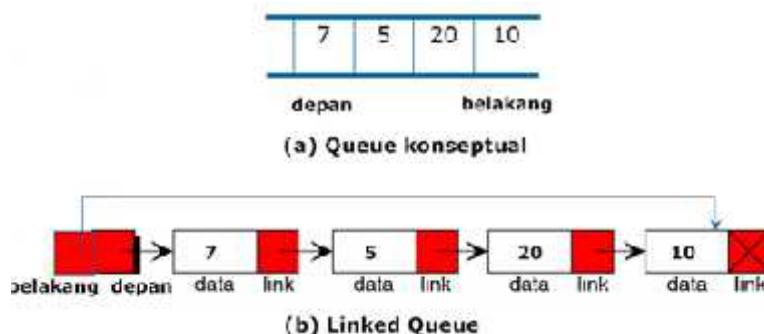
Elemen antrian disimpan sebagai node yang dibuat dengan memori dinamis

(1) Tiap node memiliki field:

- Data : berisi elemen data antrian
- Link : pointer ke node berikut dalam antrian

(2) Struktur suatu antrian memuat field-field berikut:

- Depan : pointer ke node pertama dalam antrian
- Belakang : pointer ke node terakhir dalam antrian



Gambar 5.11. Ilustrasi queue menggunakan linked list

Untuk menyajikan antrian, menggunakan larik, maka kita membutuhkan deklarasi antrian, misalnya, sebagai berikut :

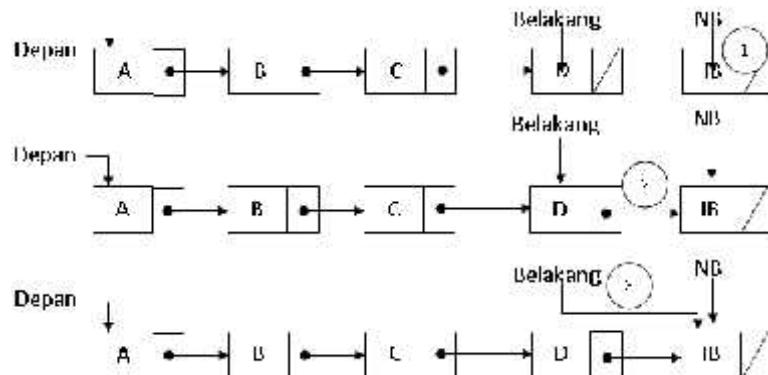
```
Type typeinfo = ....;
Typeptr = ^typenode
Typenode = record
  Info : typeinfo;
  link : typeptr;
End;
Typequeue = record
  Depan, belakang : typeptr;
Var antrian : typequeue;
```

Penambahan elemen pada *queue* disimpan di posisi terakhir. Setelah proses penambahan selesai, maka variable akhir akan menunjuk ke data baru tersebut. Ada 2 kondisi yang harus diperiksa yaitu kondisi penambahan pada *queue* yang masih kosong dan kondisi penambahan *queue* yang sudah mempunyai elemen.

Fungsi tersebut sama dengan fungsi menambah simpul di belakang, pada linked list. Langkah-langkahnya:

- (1) membuat simpul baru kemudian diisi info baru.
- (2) simpul paling belakang dihubungkan ke simpul baru.

- (3) penunjuk pointer belakang diarahkan ke simpul baru



Gambar 5.12. Ilustrasi enqueue

```

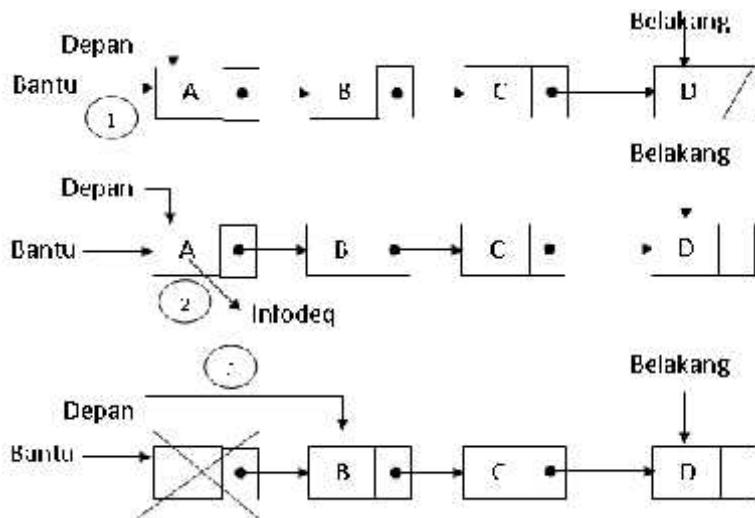
Procedure enqueue(var antrian : typequeue; IB : typeinfo);
Var NB : typeptr;
Begin
  New(NB);
  NB^.info := IB;
  NB^.link := nil;
  If queuekosong(antrian) then
    begin
      antrian.depan := NB;
      antrian.belakang := NB;
    End else Begin
      antrian.belakang^.link := NB;
      antrian.belakang := NB;
    End;
  End;
End;

```

Pengambilan data pada *queue* sama dengan proses penhapusuan elemen pertama (awal) pada linked list, sehingga posisi awal *queue* akan berpindah ke elemen *queue* berikutnya. Ada 3 kondisi yang perlu diperhatikan yaitu kondisi *queue* kosong, kondisi *queue* hanya memiliki 1 data antrian, dan kondisi *queue* yang memiliki data lebih dari 1 elemen.

Langkah-langkahnya:

- (1) simpul bantu diarahkan ke simpul depan
- (2) simpul depan diarahkan ke simpul berikutnya
- (3) simpul bantu dihapus dari *queue*.



Gambar 5.13. Ilustrasi dequeue

```

Procedure dequeue(var antrian : typequeue; Infodeq : typeinfo);
Var bantu : typeptr;
Begin
  If not(queuekosong(antrian)) then
    begin
      bantu := antrian.depan ;
      infodeq := bantu^.info;
      antrian.depan := antrian.depan^.nlink;
      If antrian.depan := nil then
        antrian.belakang := Nil;
      Dispose(bantu);
    End;
End;
  
```

5.3. Contoh Penerapan Queue

Salah satu contoh penerapan struktur data antrian adalah pengurutan data dengan metode Radix (*Radix Sort*). Metoda *Radix Sort* hanya ditujukan untuk pengurutan data yang bertipe numerik saja. Dalam metoda *Radix Sort* proses pengurutan didasarkan pada harga sesungguhnya dari suatu digit pada bilangan-bilangan yang hendak diurutkan. Dalam basis sistem bilangan desimal, maka digit-digit suatu bilangan dapat dikelompokkan menjadi 10 kelompok, yaitu kelompok “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, dan “9”. Dengan demikian harga suatu bilangan dapat diidentifikasi ke dalam kelompok-kelompok digit tersebut.

Sebagai contoh pada bilangan 5426 angka 6 menempati pada digit “satuan”, angka 2 menempati pada digit “puluhan”, angka 4 menempati pada digit “ratusan”, dan angka 5 menempati pada digit “ribuan”. Untuk mengurutkan data-data bertipe numerik, maka dapat dilakukan dengan cara membandingkan setiap digit yang bersesuaian terhadap bilangan yang lain.

Pengurutan data dengan metoda *Radix Sort* dilakukan dengan cara membandingkan setiap digit bilangan yang akan diurutkan mulai dari digit paling kiri yaitu digit yang mempunyai harga paling besar. Jika harganya sama maka kemudian dibandingkan kembali digit pada posisi di sebelah kanannya hingga digit yang terakhir yaitu digit yang ditulis paling kanan. Pada saat dijumpai adanya perbedaan nilai pada digit yang sama, maka kita dapat menentukan bahwa suatu bilangan adalah lebih besar atau lebih kecil berdasarkan harganya. Sebagai contoh, data 6425 adalah lebih besar dari 5425, karena harga pada digit pertama (dari kiri), yaitu 6 lebih besar dari 5.

Contoh:

Data inputan : 0, 1, 3, 2, 4, 7, 5, 8, 9, 6

Hasil urut : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Catatan:

Macam data (=M) input =10, yaitu 0 hingga 9

Secara lebih umum, seandainya data yang akan diurutkan adalah data-data bilangan bulat yang memiliki sejumlah M macam data, maka data-data yang akan diurutkan tersebut dapat dinotasikan sebagai:

$$X_1, X_2, X_3, \dots, X_M$$

Catatan:

X_i : bilangan bulat

Macam data (=M) input : antara 0 hingga M-1

Dengan memanfaatkan prosedur *InitQueue* dan *EnQueue*, maka prosedur *Radix Sort* untuk mengurutkan data tersebut harus diimplementasikan dalam struktur data *Queue* dinamis, karena cacah data yang akan diurutkan (=N) bisa berisi berapapun. Prosedur *Radix Sort* dengan struktur *Queue* dinamis dapat dilakukan dalam 3 langkah. Berikut ini langkah dan implementasi prosedur *Radix Sort*, yaitu:

(1) Inisialisasi M ke Queue, yaitu Q_0 hingga $Q_{(M-1)}$

Larik: array[1..N] of Queue;

For I:=1 To N Do InitQueue(X[I]); (*Inisialisasi*)

(2) Kokatensi M ke Queue

J:=1;

For I:= 0 To M-1 Do

If Q[I].Depan <> Nil Then

Begin

B:=Q[I].Depan;

(3) Transfer data dari Queue ke X

```
While B<>Nil Do
    Begin
        X[J]:=B^.Isi;
        J:=J+1;
    End;
End;
```

5.4. Latihan

1. Jelaskan apa yang dimaksud dengan FIFO pada *queue* ?
2. Apa yang dimaksud dengan *enqueue* dan *dequeue* ?
3. Pada saat kita menggunakan perintah *enqueue*, kita membutuhkan parameter masukan, contoh: *enqueue* (7). Tapi hal ini tidak berlaku untuk perintah *dequeue*, *dequeue* tidak membutuhkan parameter masukan. Hal apakah yang membedakan kedua perintah tersebut ? Jelaskan !
4. Analisa kode sumber *queue* yang telah Anda tulis, dan jelaskan masing-masing method (fungsi atau prosedur) dengan menggambarkan flowchart masing-masing method tersebut.

BAB VI

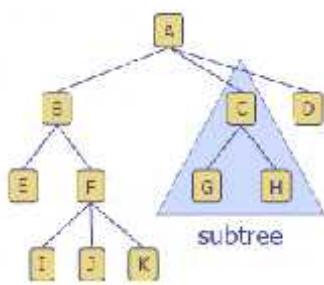
T R E E

Tree adalah kumpulan simpul/node dengan satu elemen khusus yang disebut *root* dan *node* lainnya terbagi menjadi himpunan-himpunan yang saling tidak berhubungan satu sama lain (disebut *sub Tree*).

Sebelumnya kita sudah mengenal struktur data list, yang berupa obyek-obyek yang saling terkait. Dalam list, satu obyek hanya terkait dengan satu obyek berikutnya melalui sebuah pointer. List dapat dikembangkan menjadi struktur data yang lebih kompleks, misalnya dengan menambah jumlah pointer dalam obyek. Misal dengan penambahan satu pointer lagi. Artinya bahwa jika masing-masing obyek memiliki dua pointer, ada dua obyek lain yang ditunjuknya. Struktur yang demikian dikenal sebagai *binary Trees* atau dikenal juga sebagai *Tree Node*. Oleh karena itu *Tree* merupakan salah satu bentuk struktur data tidak linier yang menggambarkan hubungan yang bersifat hirarkis (hubungan *one to many*) antara elemen-elemen.

6.1. Terminologi Tree

<i>Predecessor</i>	: <i>Node</i> yang berada diatas <i>node</i> tertentu.
<i>Successor</i>	: <i>Node</i> yang berada dibawah <i>node</i> tertentu
<i>Ancestor</i>	: Seluruh <i>node</i> yang terletak sebelum <i>node</i> tertentu dan terletak pada jalur yang sama.
<i>Descendant</i>	: Seluruh <i>node</i> yang terletak sesudah <i>node</i> tertentu dan terletak pada jalur yang sama
<i>Parent</i>	: predecessor satu level diatas suatu <i>node</i>
<i>Child</i>	: successor satu level dibawah suatu <i>node</i>
<i>Sibling</i>	: <i>node-node</i> yang memiliki parent yang sama dengan suatu <i>node</i>
<i>SubTree</i>	: bagian dari <i>Tree</i> yang berupa suatu <i>node</i> beserta descendant nya dan memiliki semua karakteristik dari <i>Tree</i> tersebut
<i>Size</i>	: banyaknya <i>node</i> dalam suatu <i>Tree</i>
<i>Height</i>	: banyaknya tingkatan / level dalam suatu <i>Tree</i>
<i>Root</i>	: <i>Node</i> khusus dalam <i>Tree</i> yang tidak punya predecessor
<i>Leaf</i>	: <i>node-node</i> dalam <i>Tree</i> yang tak memiliki successor
<i>Degree</i>	: banyaknya <i>child</i> yang dimiliki suatu <i>node</i>

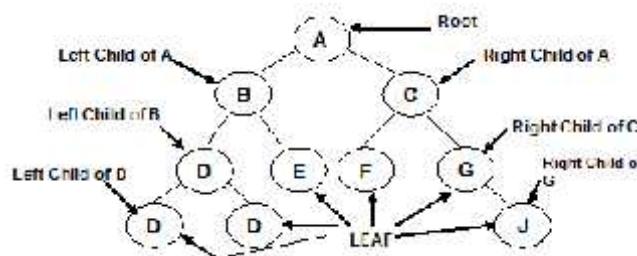


Ancestor (F)	= B, A
Descendant (C)	= G, H
Parent (D)	= A
Child (A)	= B, C , D
Sibling (F)	= E
Size	= 11
Height	= 4
Root	= A
Leaf	= E, I, J, K, G, H, D
Degree (C)	= 2

Gambar 6.1. Contoh Tree

6.2. Binary Tree

Binary Tree adalah Tree dengan syarat bahwa tiap node hanya boleh memiliki dua subTree dan kedua subTree tersebut harus terpisah. Sehingga setiap node hanya boleh memiliki paling banyak 2 child

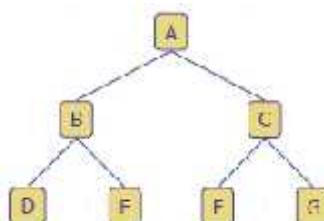


Gambar 6.2. Contoh binary Trees

6.2.1. Jenis binary Tree

(1) Full binary Tree

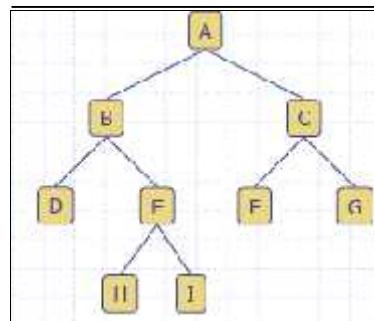
Binary Tree yang tiap nodenya (kecuali leaf) memiliki 2 child dan tiap subTree harus mempunyai panjang path yang sama



Gambar 6.3. Contoh full binary Tree

(2) Complete binary Tree

Mirip dengan Full Binary Tree, namun tiap sub tree boleh memiliki panjang path yang berbeda. Node kecuali leaf memiliki 0 atau 2 child



Gambar 6.4. Contoh Complete binary Tree

(3) Skewed binary Tree

Binary Tree yang semua *node*nya (kecuali *leaf*) hanya memiliki satu *child*



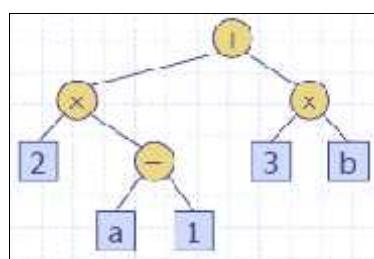
Gambar 6.5. Contoh skewed binary Tree

6.2.2. Aplikasi pada binary Tree

(1) Arithmetic expressions

Kelompok *binary Tree* yang digunakan untuk mengekspresikan notasi aritmatika

Contoh : arithmetic expression *Tree* untuk persamaan $(2 \times (a - 1) + (3 \times b))$



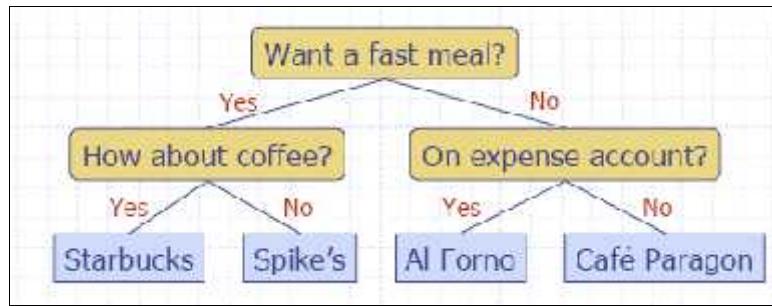
Gambar 6.6. Contoh arithmetic expressions Tree

(2) Decision processes

Kelompok *binary Tree* yang digunakan untuk mengambarkan proses pengambilan keputusan

- a. *internal nodes* : pertanyaan dengan jawaban yes/no
- b. *external nodes* : Keputusan

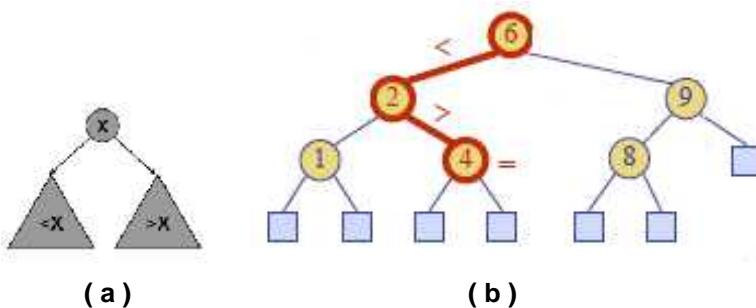
Contoh : Keputusan untuk menentukan jenis makanan untuk makan malam



Gambar 6.7. Contoh decision tree

6.3. Binary Search Tree

Binary Tree dengan sifat bahwa semua *left child* harus lebih kecil dari pada *right child* dan *parent*. Juga semua *right child* harus lebih besar dari *left child* serta parentnya. Hal ini digunakan untuk menutupi kelemahan dari *binary Tree* biasa, sehingga memudahkan proses *searching*.



Gambar 6.8. Sifat binary search Tree

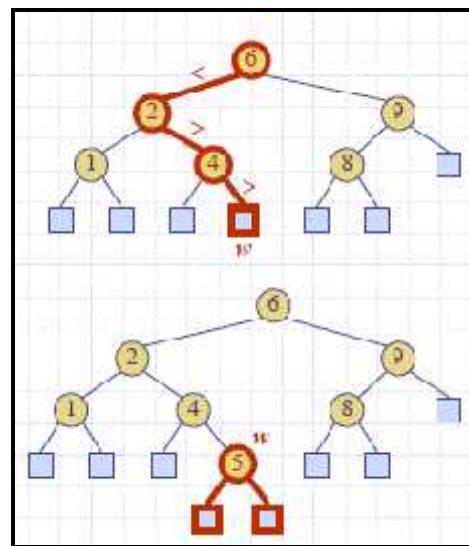
6.3.1. Operasi pada binary search Tree

(1) Operasi Insert

Operasi *insert* dilakukan setelah ditemukan lokasi yang tepat yaitu bila *node* baru $>$ *parent* maka diposisikan sebagai *right child* dan bila *node* baru $<$ *parent* diposisikan sebagai *left child*. Penyisipan sebuah elemen baru dalam *binary search tree*, elemen tersebut pasti akan menjadi *leaf*.

Sebagai contoh apabila kita akan menyisipkan nilai 5 pada pohon biner pada gambar 6.8b, maka langkah penyisipannya adalah sebagai berikut

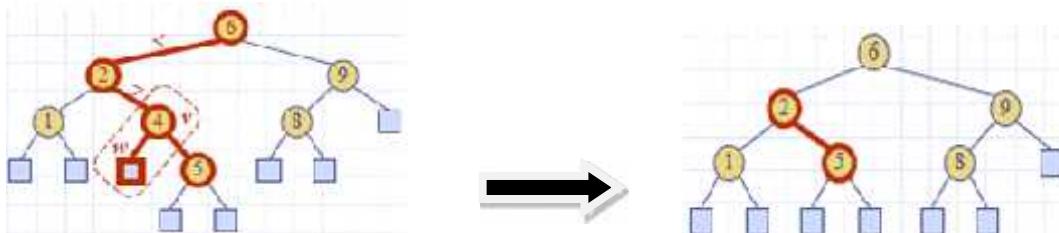
- pertama data yang akan disisipkan akan dibandingkan dengan *node* pada *root* yang bernilai 6 apakah data $<$ dari nilai *root*,
- karena benar maka selanjutnya data yang akan disisipkan akan dibandingkan dengan data yang merupakan *left child* dari *root*
- selanjutnya ulangi lagi langkah tersebut sampai data menjadi *leaf*



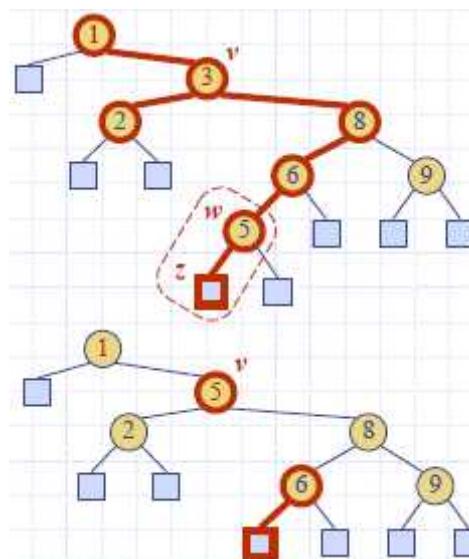
Gambar 6.9. Operasi insert pada binary search Tree

(2) Operasi Delete

Delete dalam *binary search tree* mempengaruhi struktur dari *tree* tersebut. Sehingga apabila *node* yang dihapus mempunyai *child* maka posisi *node* yang dihapus digantikan dengan *leaf* yang berada pada posisi terakhir



Gambar 6.10. Operasi delete (4) binary search Tree



Gambar 6.11. Operasi delete (3) binary search Tree

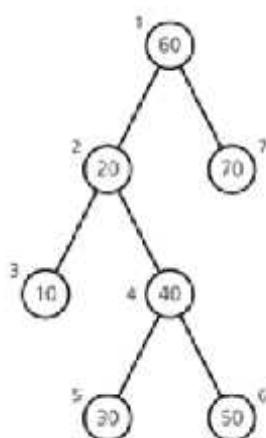
6.3.2. Model kunjungan pada binary search Tree

Pada kunjungan ini sistem yang digunakan adalah LRO (*left To Right Oriented*) artinya kunjungan selalu pada *left child* dahulu baru ke *right child*.

(1) PreOrder (Depth First Order):

- cetak isi *node* yang dikunjungi
- kunjungi *left child*
- kunjungi *right child*

Bila diketahui pohon biner seperti terlihat pada gambar 6.12. maka hasil kunjungan menggunakan metode *Preorder* adalah : **60 20 10 40 30 50 70**

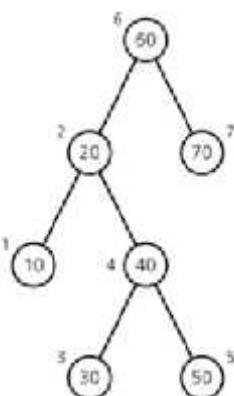


Gambar 6.12. Kunjungan Preorder

(2) InOrder (Symetric Order)

- kunjungi *left child*
- cetak isi *node* yang dikunjungi
- kunjungi *right child*

Bila diketahui pohon biner seperti terlihat pada gambar 6.13. maka hasil kunjungan menggunakan metode *InOrder* adalah : **10 20 30 40 50 60 70**

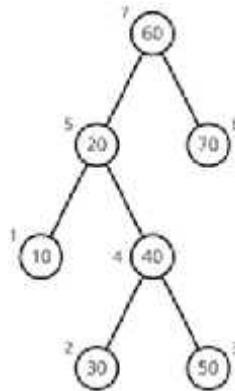


Gambar 6.13. Kunjungan Inorder

(3) PostOrder

- kunjungi left *child*
- kunjungi right *child*
- cetak isi *node* yang dikunjungi

Bila diketahui pohon biner seperti terlihat pada gambar 6.14. maka hasil kunjungan menggunakan metode *InOrder* adalah : **10 30 50 40 20 70 60**



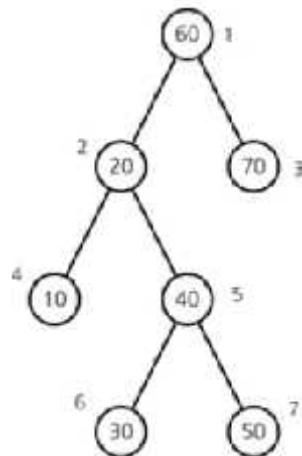
Gambar 6.14. Kunjungan PostOrder

(4) LevelOrder

kunjungi *node* pada tingkat yang sama dimulai dari *root* sampai *node-node* yang merupakan *leaf*-nya.

Bila diketahui pohon biner seperti terlihat pada gambar 6.15. maka hasil kunjungan menggunakan metode *LevelOrder* adalah :

60 20 70 10 40 30 50



Gambar 6.15. Kunjungan level order

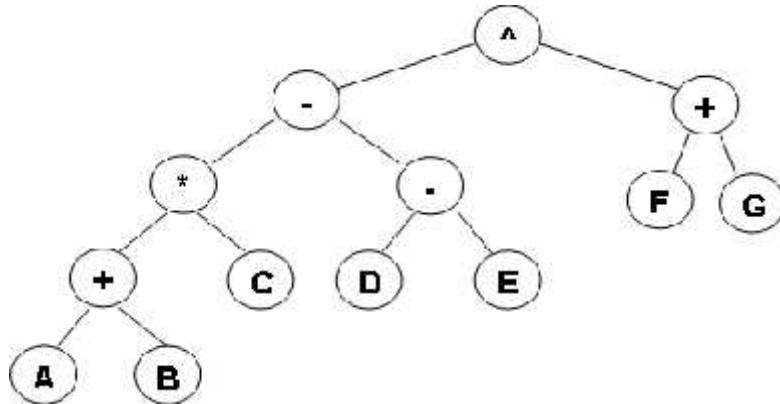
6.3.3. Notasi Prefix, Infix, dan Postfix

(1) Prefix

Sebuah *binary tree* apabila dikunjungi secara *preorder* akan menghasilkan notasi **prefix**

Bila diketahui pohon biner seperti terlihat pada gambar 6.16. maka hasil Notasi *Prefix* adalah : ^

- * + A B C - D E + F G



Gambar 6.16. Notasi Prefix

2) InFix

Sebuah *binary Tree* apabila dikunjungi secara *inorder* akan menghasilkan notasi **Infix**

Bila diketahui pohon biner seperti terlihat pada gambar 6.16. maka hasil notasi *Infix* adalah : ((A

+ B) * C - (D - E)) ^ (F + G)

3) PostFix

Sebuah *binary Tree* apabila dikunjungi secara *postorder* akan menghasilkan notasi **Postfix**

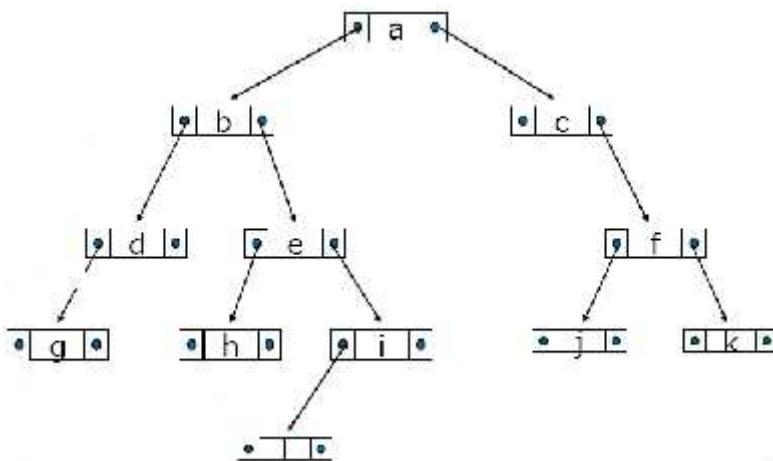
Bila diketahui pohon biner seperti terlihat pada gambar 6.16. maka hasil notasi *Postfix* adalah :

A B + C * D E - - F G + ^

6.4. Implementasi Binary Tree

Implementasi dalam pemrograman ini akan digunakan untuk pohon biner saja. Sebagai contoh data yang digunakan untuk implementasi *binary Tree* adalah data bertipe data char. Masing-masing obyek atau *node* dalam *binary Tree* memiliki dua pointer yang biasa disebut *left* dan *right*. Pointer *left* dan *right* sebuah obyek dapat bernilai nil (tidak menunjuk ke obyek lain) atau dapat menunjuk ke obyek lain. *Node* atau *obyek* yang menunjuk ke *node* lain disebut sebagai *parent*, sedangkan *node* yang ditunjuk disebut sebagai *child*. Tidak semua struktur data berantai yang tersusun atas *Tree* adalah *binary Tree*. Bisa disebut *binary Tree* jika ada satu *node* dalam *Tree* yang tidak memiliki *parent* (disebut *root*) dan setiap *node* dalam *Tree* mempunyai satu *parent*.

Dengan demikian tidak mungkin terjadi loop dalam binary Tree, dengan kata lain tidak mungkin mengikuti rantai pointer dan kemudian kembali lagi ke *node* yang sama.



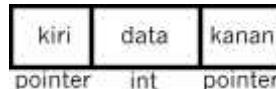
Gambar 6.17. Implementasi Binary Tree

Pada gambar ilustrasi binary Tree di atas, terdapat *node* yang tidak menunjuk ke obyek manapun. *Node* ini disebut sebagai leaf . Ciri-ciri leaf adalah kedua pointernya bernilai nil , karena tidak menunjuk ke *node* manapun.

(1) Deklarasi Tree

Tree tersusun atas *node-node*, sehingga perlu kita deklarasikan adalah komponen *node* itu sendiri. Dalam contoh dibawah, akan kita namai **Node**.

Sebelumnya perlu kita lihat bahwa untuk mewujudkan implementasi *node* ke dalam bahasa pemrograman, diperlukan sebuah struktur yang memiliki susunan berikut ini:



Berikut ini adalah contoh deklarasi obyek dalam binary Tree (dalam Pascal):

```

Type tipedata = integer;
Tree = ^node;
Node = record;
    data : tipedata;
    kiri, kanan : Tree;
End;
  
```

Variabel data digunakan untuk menyimpan nilai angka *node* tersebut, sedangkan kiri dan kanan, bertipe pointer, masing-masing mewakili vektor yang akan menunjuk ke *node* anak kiri dan kanan.

(2) Inisialisasi Tree

Untuk pertama kali, saat kita akan membuat sebuah pohon biner, asumsi awal adalah pohon itu belum bertumbuh, belum memiliki *node* sama sekali, sehingga masih kosong.

Pohon := Nil;

Kita mendeklarasikan sebuah pointer yang akan menunjuk ke akar pohon yang kita buat, dengan nama Pohon. Pointer ini ditujukan untuk menunjuk struktur bertipe *Node*, yang telah dibuat pada bagian 1. Karena pohon tersebut sama sekali belum memiliki *node*, maka pointer **Pohon** ditunjukkan ke **Nil**.

(3) Menambahkan Node Pada Tree

Karena pohon yang kita buat merupakan sebuah pohon biner, maka untuk menambahkan sebuah *node*, secara otomatis penambahan tersebut mengikuti aturan penambahan *node* pada pohon biner:

Jika pohon kosong, maka *node* baru ditempatkan sebagai akar pohon.

Jika pohon tidak kosong, maka dimulai dari *node* akar, dilakukan proses pengecekan berikut:

- Jika nilai *node* baru lebih kecil dari nilai *node* yang sedang dicek, maka lihat ke kiri *node* tersebut. Jika kiri *node* tersebut kosong (belum memiliki kiri), maka *node* baru menjadi kiri *node* yang sedang dicek. Seandainya kiri *node* sudah terisi, lakukan kembali pengecekan a dan b terhadap *node* kiri tersebut. Pengecekan ini dilakukan seterusnya hingga *node* baru dapat ditempatkan.
- Jika nilai *node* baru lebih besar dari nilai *node* yang sedang dicek, maka lihat ke kanan *node* tersebut. Jika kanan *node* tersebut kosong (belum memiliki kanan), maka *node* baru menjadi kanan *node* yang sedang dicek. Seandainya kanan *node* sudah terisi, lakukan kembali pengecekan a dan b terhadap *node* kanan tersebut. Pengecekan ini dilakukan seterusnya hingga *node* baru dapat ditempatkan.

Proses penambahan ini diimplementasikan secara rekursif pada fungsi berikut:

```
Procedure sisip_node(d : typedata; var pohon : Tree);
Begin
  If pohon = nil then
    Begin
      New(pohon);
      Pohon^.data := d;
      Pohon^.kiri := nil;
      Pohon^.kanan := nil;
    End
  else if pohon^.isi < d then sisip_node(d,Pohon^.kanan)
  else if pohon^.isi > d then sisip_node(d,Pohon^.kiri);
end;
```

(4) Membaca dan Menampilkan Node Pada Tree

Untuk membaca dan menampilkan seluruh *node* yang terdapat pada pohon biner, terdapat 3 macam cara, atau yang biasa disebut *kunjungan* (visit). Semua kunjungan diawali dengan mengunjungi akar pohon. Karena proses kunjungan ini memerlukan perulangan proses yang sama namun untuk *depth* (kedalaman) yang berbeda, maka ketiganya diimplementasikan dengan fungsi rekursif.

a. Kunjungan Pre-Order.

Kunjungan pre-order dilakukan mulai dari akar pohon, dengan urutan:

1. Cetak isi (data) *node* yang sedang dikunjungi
2. Kunjungi kiri *node* tersebut,
 - a) Jika kiri bukan kosong (tidak NIL) mulai lagi dari langkah pertama, terapkan untuk kiri tersebut.
 - b) Jika kiri kosong (NIL), lanjut ke langkah ketiga.
3. Kunjungi kanan *node* tersebut,
 - a) Jika kanan bukan kosong (tidak NIL) mulai lagi dari langkah pertama, terapkan untuk kanan tersebut.
 - b) Jika kanan kosong (NIL), proses untuk *node* ini selesai, tuntaskan proses yang sama untuk *node* yang dikunjungi sebelumnya.

Implementasi dalam bahasa Pascal :

```
Procedure Preorder(Pohon : Tree);
Begin
  If pohon <> nil then begin
    Write(pohon^.isi);
    Preorder(pohon^.kiri);
    Preorder(pohon^.kanan);
  End;
End;
```

b. Kunjungan In-Order.

1. Kunjungi kiri *node* tersebut,
 - a) Jika kiri bukan kosong (tidak NIL) mulai lagi dari langkah pertama, terapkan untuk kiri tersebut.
 - b) Jika kiri kosong (NIL), lanjut ke langkah kedua.
2. Cetak isi (data) *node* yang sedang dikunjungi
3. Kunjungi kanan *node* tersebut,
 - a) Jika kanan bukan kosong (tidak NIL) mulai lagi dari langkah pertama, terapkan untuk kanan tersebut.
 - b) Jika kanan kosong (NIL), proses untuk *node* ini selesai, tuntaskan proses yang sama untuk *node* yang dikunjungi sebelumnya.

Implementasi dalam bahasa Pascal :

```
Procedure Inorder(Pohon : Tree);
Begin
  If pohon <> nil then begin
    Inorder(pohon^.kiri);
    Write(pohon^.isi);
    Inorder(pohon^.kanan);
  End;
End;
```

c. Kunjungan Post-Order.

1. Kunjungi kiri *node* tersebut,

- a) Jika kiri bukan kosong (tidak NIL) mulai lagi dari langkah pertama, terapkan untuk kiri tersebut.
- b) Jika kiri kosong (NIL), lanjut ke langkah kedua.
2. Kunjungi kanan *node* tersebut,
 - a) Jika kanan bukan kosong (tidak NIL) mulai lagi dari langkah pertama, terapkan untuk kanan tersebut.
 - b) Jika kanan kosong (NIL), lanjut ke langkah ketiga.
3. Cetak isi (data) *node* yang sedang dikunjungi. Proses untuk *node* ini selesai, tuntaskan proses yang sama untuk *node* yang dikunjungi sebelumnya.

Implementasi dalam bahasa Pascal :

```
Procedure Postorder(Pohon : Tree);
Begin
  If pohon <> nil then
    begin
      Postorder(pohon^.kiri);
      Postorder(pohon^.kanan);
      Write(pohon^.isi);
    End;
  End;
```

6.5. Latihan

1. Dipunyai sekelompok data yang akan disimpan dalam pohon telusur biner (Binary Search Tree) dengan urutan pemasukan sebagai berikut :

M A L E C I T A N U R A T A L A S I N G G I H

- a. gambarkan pohon telusur biner yang terjadi
 - b. tuliskan urutan data jika ditelusuri secara Pre-order
 - c. tuliskan urutan data jika ditelusuri secara Post-order
 - d. jika data **C** dihapus gambarkan pohon telusur biner yang terjadi
-
2. Bila diketahui suatu fungsi Y sebagai berikut : $(a + (b^c - d^e * f)^g) / (h^i)$
 - a. Buatlah dalam bentuk binary three
 - b. Tuliskan notasi prefix
 - c. Tuliskan notasi postfix
 - 3) Buatlah fungsi untuk mencetak semua *leaf* (daun) yang ada pada pohon biner.
 - 4) Buatlah fungsi untuk mencetak nilai *node* minimum (terkecil) pada pohon biner.

DAFTAR PUSTAKA

- Aho , A., J.Hopcroft and J. Ulman, 1988, *Data Structure and Algorithms*, Adison Wesley
- Bambang, W., 2004, *Pengantar Struktur Data Dan Algoritma*, Andi Yogyakarta
- Brassard, G., and Bratley, P., 1996, *Fundamentals of Algorithmics*, Prentice Hall, Englewood Cliffs, New Jersey.
- Fadilah, R., 2008, *Algoritma & Struktur Data*, <http://rijalfadilah.files.wordpress.com>
- Haryanto, B., 2000, *Struktur Data*, Informatika, Bandung
- Munir, R. dkk, 1998, *Algoritma dan Pemrograman 2*, Informatika, Bandung
- Sanjaya, D., 2001, *Bertualang dengan Struktur Data di Planet Pascal*, J & J Learning, Yogyakarta.
- Santosa, I., 1992, *Struktur Data menggunakan Turbo Pascal 6.0*, Andi Offset, Yogyakarta.
- Zakaria, T. M. dan Prijono, 2005, A., *Konsep dan Implementasi Struktur Data*, Informatika.



Buku Struktur Data ini mempunyai susunan dan struktur materi yang terbagi dalam 6 bab, yaitu Pendahuluan, Sorting, Linked List, Stack, Queue dan Tree.

- Bab 1 yang merupakan bagian pendahuluan membahas struktur data dengan Pascal mulai dari hirarki struktur data pascal, struktur data statis, struktur data dinamis serta procedure dan function.
- Bab 2 membahas materi tentang sorting dengan menggunakan metode exchange sort, selection sort, insertion sort, serta radix sort.
- Bab 3 membahas tentang single linked list serta double linked list.
- Bab 4 membahas tentang representasi stack menggunakan array, representasi stack menggunakan linked list serta implementasi stack untuk mengkonversi bilangan desimal ke biner.
- Bab 5 membahas tentang representasi queue menggunakan array, representasi queue menggunakan linked list serta contoh penerapan queue.
- Bab 6 membahasa terminologi tree, binary tree, binary search tree, serta implementasi binary tree.
- Buku ini dapat digunakan sebagai salah satu referensi untuk matakuliah struktur data.

ISBN 978-602-7619-03-6



9 786027 619036