Modul 10 TREE (BAGIAN PERTAMA)

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

10.1 *Pengertian Rekursif*

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

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

Sub Program Rekursif adalah sub program yang memanggil dirinya sendiri selama kondisi pemanggilan dipenuhi. Prinsip rekursif sangat berkaitan erat dengan bentuk induksi matematika. Berikut adalah contoh fungsi rekursif pada rumus pangkat 2:
Kita ketahui bahwa secara umum perhitungan pangkat 2 dapat dituliskan sebagai berikut

$$20 = 1$$
$$2n = 2 * 2n - 1$$

Secara matematis, rumus pangkat 2 dapat dituliskan sebagai

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

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

```
function pangkat_2 ( x : integer )→integer
kamus

algoritma
       if( x = 0 ) then
              → 1
       else
              → 2 * pangkat_2( x - 1 )
```

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

Pangkat_2 (4)
➔ 2 * pangkat_2 (3)
➔ 2 * (2 * pangkat_2 (2))
➔ 2 * (2 * (2 * pangkat_2 (1)))
➔ 2 * (2 * (2 * (2 * pangkat_2 (0))))
➔ 2 * (2 * (2 * (2 * 1)))
➔ 2 * (2 * (2 * 2))
➔ 2 * (2 * 4)

➔ 2 * 8 ➔ 16

10.2 *Kriteria Rekursif*

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

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

Secara umum bentuk dari sub program rekursif memiliki statemen kondisional :

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

Sub program rekursif umumnya dipakai untuk permasalahan yang memiliki langkah penyelesaian yang terpola atau langkah-langkah yang teratur. Bila kita memiliki suatu permasalahan dan kita mengetahui algoritma penyelesaiannya, kadang-kadang sub program rekursif menjadi pilihan kita bila memang memungkinkan untuk dipergunakan. Secara algoritmis (dari segi algoritma, yaitu bila kita mempertimbangkan penggunaan memori, waktu eksekusi sub program) sub program rekursif sering bersifat tidak efisien.
Dengan demikian sub program rekursif umumnya memiliki efisiensi dalam penulisan perintah, tetapi kadang tidak efisien secara algoritmis. Meskipun demikian banyak pula permasalahan-permasalahan yang lebih sesuai diselesaikan dengan cara rekursif (misalnya dalam pencarian / *searching*, yang akan dibahas pada pertemuan-pertemuan yang akan datang).

Fakultas Informatika
School of Computing
Telkom University
Informatics Lab

10.3 *Kekurangan Rekursif*

Konsep penggunaan yang terlihat mudah karena fungsi rekursif dapat menyederhanakan solusi dari suatu permasalahan, sehingga sering kali menghasilkan bentuk algoritma dan program yang lebih singkat dan lebih mudah dimengerti.
Kendati demikian, penggunaan rekursif memiliki beberapa kekurangan antara lain:

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

Secara umum gunakan penyelesaian rekursif hanya jika:

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

10.4 *Contoh Rekursif*

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

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

Algoritma	C++

Program coba_rekursif
 	
kamus
  bil, bil_pkt : integer
 	
  function pangkat (input x,y: integer)
	#include <iostream>
	using namespace std;
	/* prototype fungsi rekursif */
	int pangkat(int x, int y);
	/* fungsi utama */
	int main(){
	  int bil, bil_pkt;

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

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

Algoritma	C++

Program rekursif_factorial
 	
kamus
  faktor, n : integer
 	
  function faktorial (input
  a: integer)→integer
 	
algoritma
  input(n)
  faktor =faktorial(n)
  output(faktor)
endprogram
 	
function faktorial (input
  a: integer)→integer
kamus
algoritma
  if (a == 1 || a == 0) then
  → 1
  else if (a > 1) then
  → a* faktorial(a-1)
  else
  → 0
  endif
endfunction
	#include <iostream>
	
	long int faktorial(long int a);
	
	main(){
	long int faktor;
	long int n;
	cout<<"Masukkan nilai faktorial ";
	cin>>n;
	faktor =faktorial(n);
	cout<<n<<"!="<<faktor<<endl;
	}
	
	long int faktorial(long int y){
	if (a==1 || a==0){
	return(1);
	}else if (a>1){
	return(a*faktorial(a-1));
	}else{
	return 0;
	}
	}

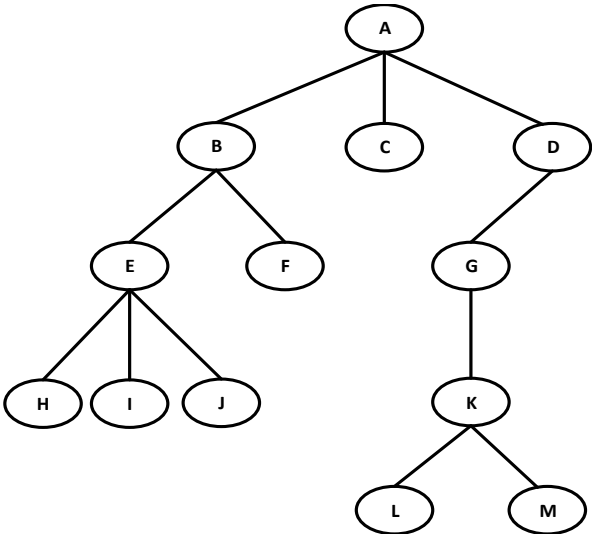
10.5 *Pengertian Tree*

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

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

Karateristik dari suatu *tree* T adalah :

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



Gambar 10-1 *Tree*

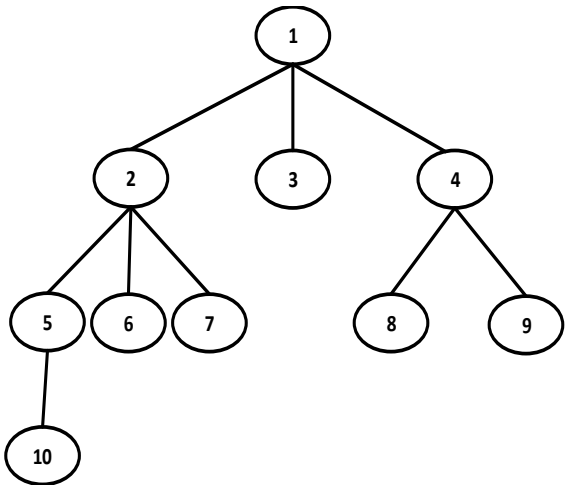
Berdasarkan gambar di atas dapat digambarkan beberapa terminologinya, yaitu

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

10.6 *Jenis-Jenis Tree*

10.6.1 Ordered Tree

Yaitu pohon yang urutan anak-anaknya penting.



Gambar 10-2 *Ordered Tree*

10.6.2 Binary Tree

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

A. *Complete Binary Tree*

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

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

B. *Extended Binary Tree*

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

C. *Binary Search Tree*

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

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

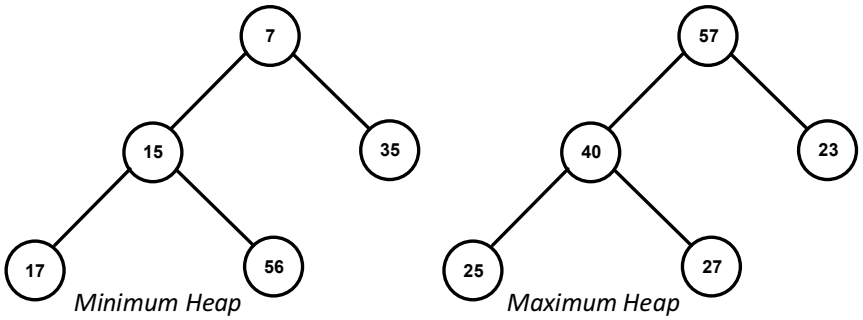
D. AVL *Tree*

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

E. *Heap Tree*

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

Heap juga disebut *Complete Binary Tree*, karena jika suatu *node* mempunyai *child*, maka jumlah *child*-nya harus selalu dua.
Minimum Heap: jika *parent*-nya selalu lebih kecil daripada kedua *children*-nya.
Maximum Heap: jika *parent*-nya selalu lebih besar daripada kedua *children*-nya.



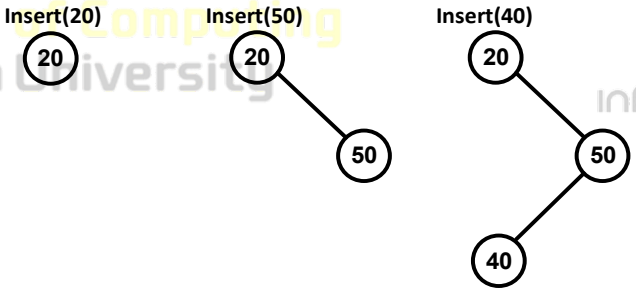
Gambar 10-3 *Heap Tree*

10.7 *Operasi-Operasi dalam Binary Search Tree*

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

A. *Insert*

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



Gambar 10-4 *Binary Search Tree Insert*

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

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

B. *Update*

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

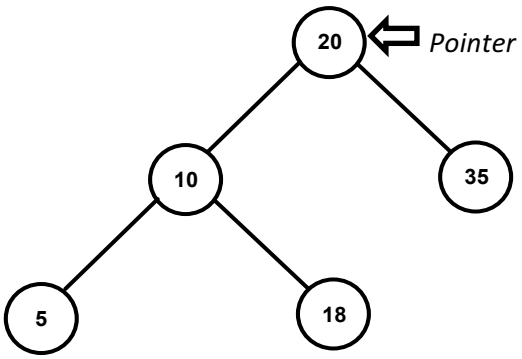
C. *Search*

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

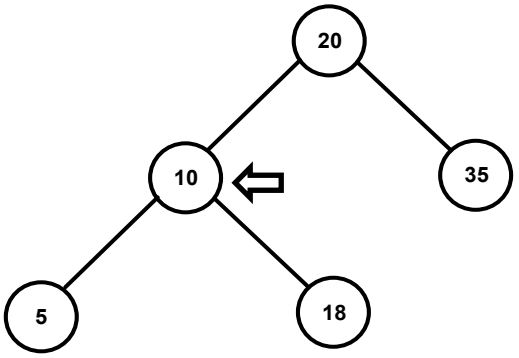
1. Pencarian pada *binary search tree* dilakukan dengan menaruh *pointer* dan membandingkan nilai yang dicari dengan *node* awal (*root*)
2. Jika nilai yang dicari tidak sama dengan *node*, maka *pointer* akan diganti ke *child* dari *node* yang ditunjuk:
   a. *Pointer* akan pindah ke *child* kiri bila,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

fakultas Informatika
School of Computing
Telkom University
informatics lab

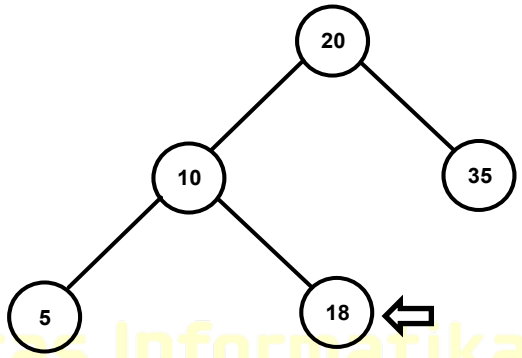
Nilai dicari : 18



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



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



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

Fakultas Informatika
School of Computing
Telkom University
informatics lab

D. *Delete*

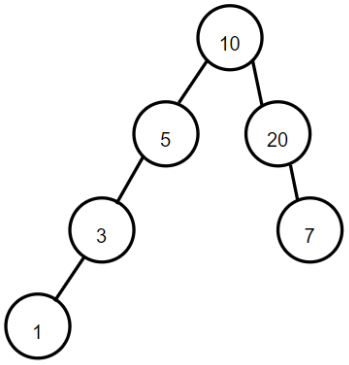
1. *LEAF*, tidak perlu dilakukan modifikasi.
2. *Node* dengan 1 *Child*, maka *child* langsung menggantikan posisi *Parent*.
3. *Node* dengan 2 *Child*:
   - Left *Subtree*, yang diambil adalah *node* yang paling kiri (nilai terbesar).
   - Right *Subtree*, yang diambil adalah *node* yang paling kanan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

Gambar 108 *Binary Search Tree* sebelum di-*Delete*

Gambar 109 *Binary Search Tree* setelah di-*Delete*

E. *Most-Left*

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

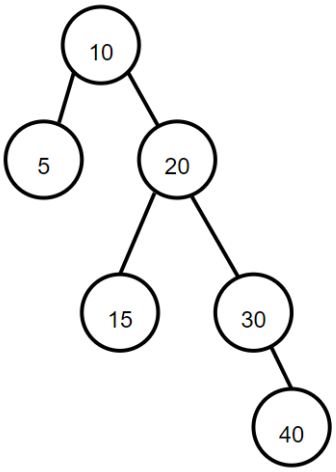


Gambar 1010 *most-left tree*

Pada *tree* di atas, *most-left tree* adalah = 1

F. *Most-Right*

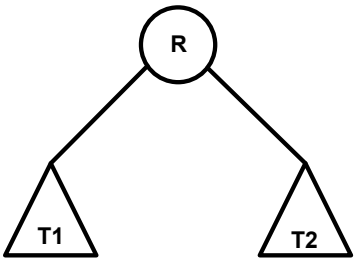
Most-right node adalah node yang berada paling kanan dalam tree. Dalam konteks *binary search tree* (BST), *most-right node* adalah *node* dengan nilai terbesar, yang dapat ditemukan dengan mengikuti anak kanan *(right child)* dari root hingga mencapai node yang tidak memiliki anak kanan lagi.



Gambar 1011 *most-right tree*

Pada *tree* di atas, *most-right tree* adalah = 40

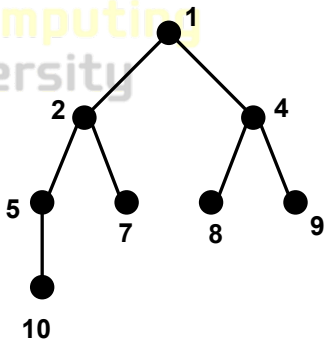
10.8 *Traversal pada Binary Tree*



Gambar 10-12 *Traversal* pada *Binary Tree* 1

1. *Pre-order* : R, T1, T2
   - kunjungi R
   - kunjungi T1 secara *pre-order*
   - kunjungi T2 secara *pre-order*
2. *In-order* : T1 , R, T2
   - kunjungiT1 secara *in-order*
   - kunjungi R
   - kunjungi T2 secara *in-order*
3. *Post-order* : T1, T2 , R
   - Kunjungi T1 secara *pre-order*
   - kunjungi T2 secara *pre-order*
   - kunjungi R

Fakultas Informatika
School of Computing
Telkom University
informatics lab



Gambar 10-13 *Traversal* pada *Binary Tree* 2

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

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

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

```
1  #ifndef tree_H
2  #define tree_H
3  #define Nil NULL
4
5  typedef int infotype;
6  typedef struct Node *address;
```

```
7   struct Node{
8       infotype info;
9       address right;
10      address left;
11  };
12  typedef address BinTree;
13  // fungsi primitif pohon biner
14  /******** pengecekan apakah tree kosong ********/
15  boolean EmptyTree(Tree T);
16  /* mengembalikan nilai true jika tree kosong */
17
18  /******** pembuatan tree kosong ********/
19  void CreateTree(Tree &T);
20  /*  I.S sembarang
21      F.S. terbentuk Tree kosong */
22
23  /******** manajemen memori ********/
24  address alokasi(infotype X);
25  /*  mengirimkan address dari alokasi sebuah elemen
26      jika alokasi berhasil maka nilai address tidak Nil dan jika gagal nilai
27  address Nil*/
28
29  void Dealokasi(address P);
30  /*  I.S P terdefinisi
31      F.S. memori yang digunakan P dikembalikan ke sistem */
32
33  /* Konstruktor */
34  address createElemen(infotype X, address L, address R)
35  /*  menghasilkan sebuah elemen tree dengan info X dan elemen kiri L dan
36  elemen kanan R
37      mencari elemen tree tertentu */
38
39  address findElmBinTree(Tree T, infotype X);
40  /* mencari apakah ada elemen tree dengan P→info = X
41      jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */
42
43  address findLeftBinTree(Tree T, infotype X);
44  /* mencari apakah ada elemen sebelah kiri dengan P→info = X
45      jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */
46
47  address findRigthBinTree(Tree T, infotype X);
48  /* mencari apakah ada elemen sebelah kanan dengan P→info = X
49      jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */
50
51  /*insert elemen tree */
52  void InsertBinTree(Tree T, address P);
53  /*  I.S P Tree bisa saja kosong
54      F.S. memasukka p ke dalam tree terurut sesuai konsep binary tree
55           menghapus elemen tree tertentu*/
56  void DelBinTree(Tree &T, address P);
57  /*  I.S P Tree tidak kosong
58      F.S. menghapus p dari Tree selector */
59
60  infotype akar(Tree T);
61  /* mengembalikan nilai dari akar */
62
63  void PreOrder(Tree &T);
64  /*  I.S P Tree tidak kosong
65      F.S. menampilkan Tree secara PreOrder */
66
67  void InOrder(Tree &T);
68  /*  I.S P Tree tidak kosong
69      F.S. menampilkan Tree secara IOrder */
70
71  void PostOrder(Tree &T);
72  /*  I.S P Tree tidak kosong
73      F.S. menampilkan Tree secara PostOrder */
```

Fakultas Informatika Telkom University informatics lab

```
74
75  #endif
76
77
78
```

10.9 *Latihan*

1. Buatlah ADT *Binary Search Tree* menggunakan *Linked list* sebagai berikut di dalam *file* **"bstree.h"**:

```
Type infotype: integer
Type address : pointer to Node
Type Node:  <
    info : infotype
    left, right : address
>
    function alokasi( x : infotype ) → address
procedure insertNode( input/output root : address,
   input x : infotype )
function findNode( x : infotype, root : address )→address
procedure printInorder( input root : address )
```

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

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

using namespace std;
int main() {
    cout << "Hello World" << endl;
    address root = Nil;
    insertNode(root,1);
    insertNode(root,2);
    insertNode(root,6);
    insertNode(root,4);
    insertNode(root,5);
    insertNode(root,3);
    insertNode(root,6);
    insertNode(root,7);
    InOrder(root);
    return 0;
}
```

Gambar 10-14 main.cpp

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

Gambar 10-15 *Output*

2. Buatlah fungsi untuk menghitung jumlah *node* dengan fungsi berikut.
   - fungsi **hitungJumlahNode**(root:address) : integer
     /* fungsi mengembalikan integer banyak node yang ada di dalam BST*/
   - fungsi **hitungTotalInfo**(root:address, start:integer) : integer
     /* fungsi mengembalikan jumlah (total) info dari node-node yang ada di dalam BST*/
   - fungsi **hitungKedalaman**(root:address, start:integer) : integer

/* fungsi rekursif mengembalikan integer kedalaman maksimal dari binary tree */

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

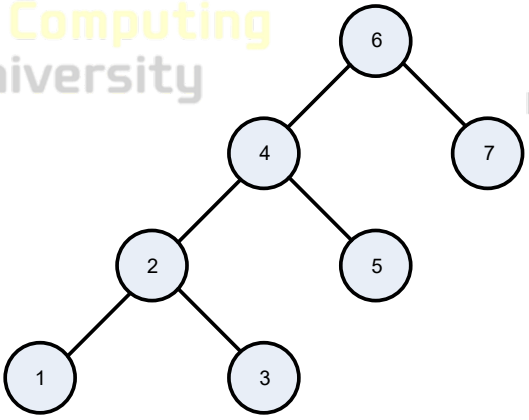
Gambar 10-16 *main*

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

Gambar 10-17 *Output*

Fakultas Informatika
School of Computing
Telkom University
informatics lab

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



Gambar 10-18 Ilustrasi *Tree*