LAPORAN PRAKTIKUM

MODUL 10

TREE (BAGIAN PERTAMA)

Disusun Oleh:

Alvin Bagus Firmansyah - 2311104070

Kelas

SE-07-02

Dosen :

…….

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY

PURWOKERTO

2024

1. Tujuan

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.

2. Landasan Teori

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 :

<u>Fungsi</u> pangkat_2 (x : integer) : integer

<u>Kamus</u> <u>Algoritma</u>

<u>If</u>(x = 0) <u>then</u>

→ 1

<u>Else</u>

→ 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).

10.3 *Kekurangan Rekursif*

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

Kendati demikian, penggunaan rekursif memiliki beberapa kekurangan antara lain:

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

Secara umum gunakan penyelesaian rekursif hanya jika:

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

10.4 *Contoh Rekursif*

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

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

Algoritma	C++

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

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

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)
 	
Algoritma
input(n)
	#include <conio.h>
	#include <iostream>
	
	long int faktorial(long int a);
	main(){
	cout<<"Masukkan nilai fak
	cin>>n;
	faktor =faktorial(n);
	cout<<n<<"!="<<faktor<<en
	getch();
	}
	long int faktor;
	long int n;

faktor =faktorial(n)
output(faktor)
function faktorial (input:
a: integer)
kamus
algoritma
if (a == 1 || a == 0) then
→ 1
else if (a > 1) then
→ a* faktorial(a-1)
else
→ 0
	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;
	}
	}

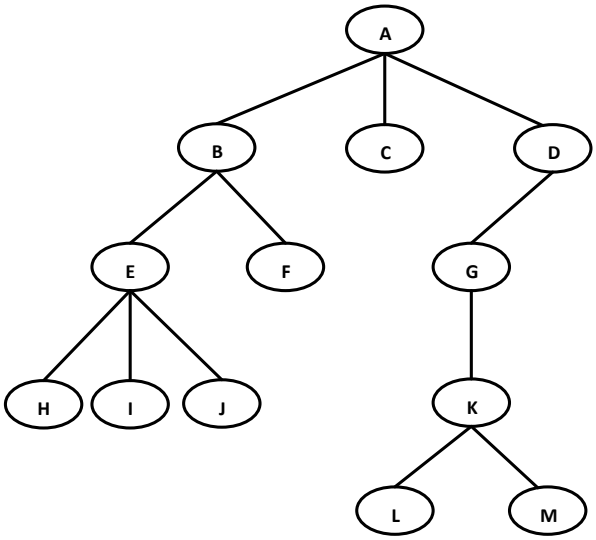
10.5 *Pengertian Tree*

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 (*nonlinear data structure*) yang disebut *tree*.

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

Karateristik dari suatu *tree* T adalah :

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



Gambar 10-1 *Tree*

Berdasarkan gambar diatas dapat digambarkan beberapa terminologinya, yaitu

1. **Anak (*child* atau *children*) dan Orangtua (*parent*). B, C, dan D adalah anak-anak simpul A, A adalah Orangtua dari anak-anak itu.**

2. **Lintasan (*path*). Lintasan dari A ke J adalah A, B, E, J. Panjang lintasan dari A ke J adalah 3.**

3. **Saudara kandung (*sibling*). F adalah saudara kandung E, tetapi G bukan saudara kandung E, karena orangtua mereka berbeda.**

4. **Derajat(*degree*). Derajat sebuah simpul adalah jumlah pohon (atau jumlah anak) pada simpul tersebut. Derajat A = 3, derajat D = 1 dan derajat C = 0. Derajat maksimum dari semua simpul merupakan derajat pohon itu sendiri. Pohon diatas berderajat 3.**

5. **Daun (*leaf*). Simpul yang berderajat nol (atau tidak mempunyai anak) disebut daun. Simpul H, I, J, F, C, L, dan M adalah daun.**

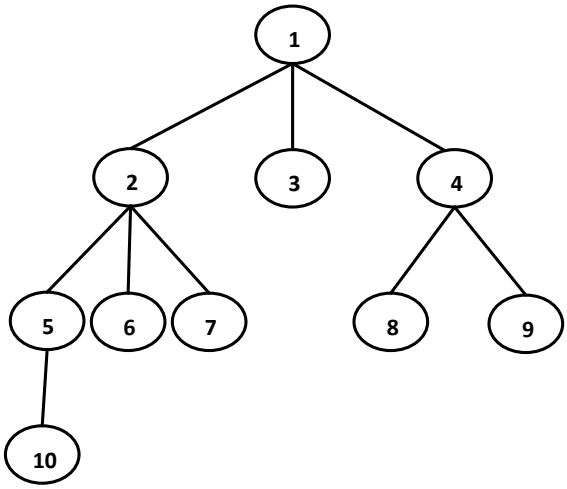
6. **Simpul Dalam (*internal nodes*). Simpul yang mempunyai anak disebut simpul dalam. Simpul B, D, E, G, dan K adalah simpul dalam.**

7. **Tinggi (*height*) atau Kedalaman (*depth*). Jumlah maksimum *node* yang terdapat di cabang *tree* tersebut. Pohon diatas mempunyai tinggi 4.**

10.6 *Jenis-Jenis Tree*

10.6.1 Ordered Tree

Yaitu pohon yang urutan anak-anaknya penting.



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.

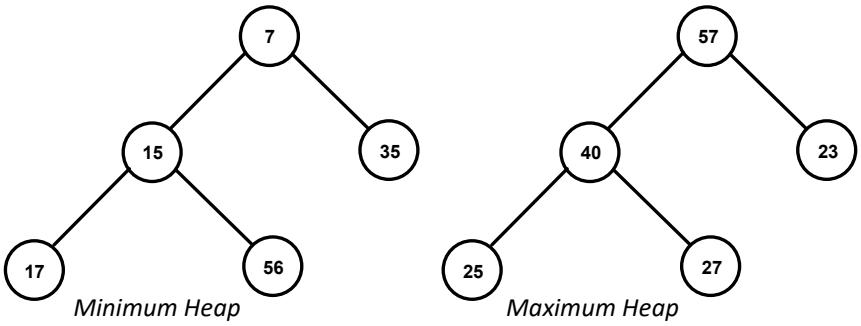
E. *Heap Tree*

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.**



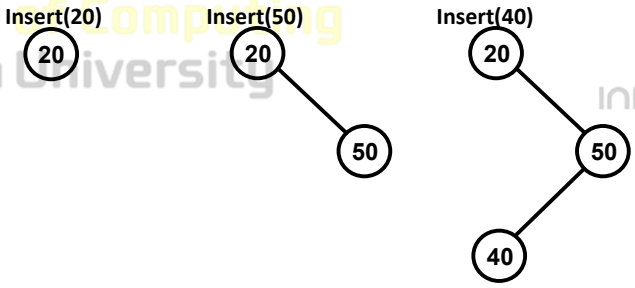
Gambar 10-3 *Heap Tree*

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*.



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

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

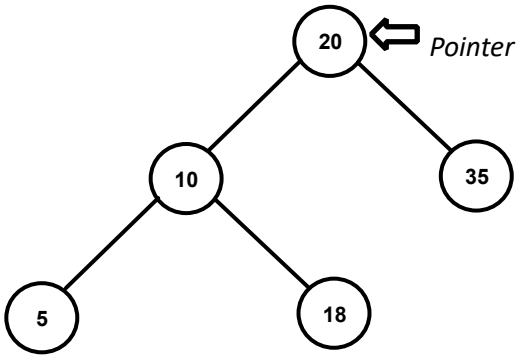
   a. ***Pointer* akan pindah ke *child* kiri bila, nilai dicari lebih kecil dari nilai *node* yang ditunjuk saat itu**

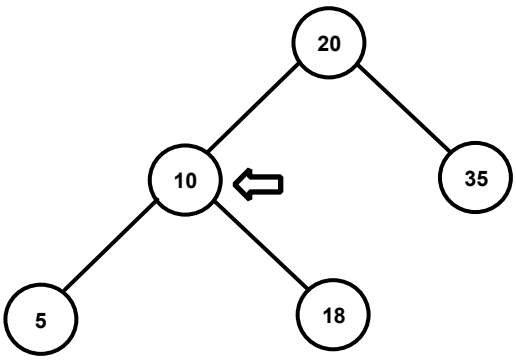
   b. ***Pointer* akan pindah ke *child* kanan bila, nilai dicari lebih besar dari nilai *node* yang ditunjuk**

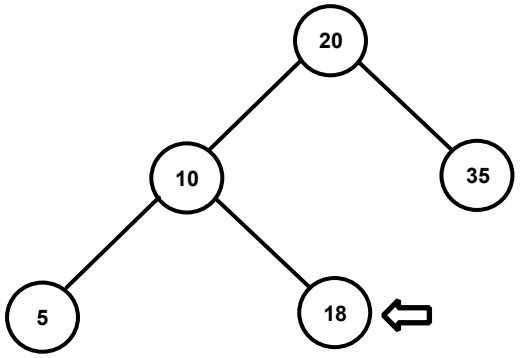
   saat itu

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

Nilai dicari : 18

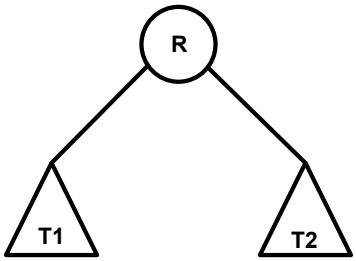






-

10.8 *Traversal pada Binary Tree*

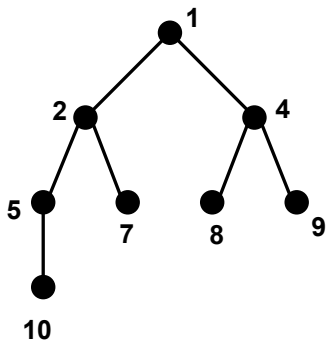


Gambar 10-8 *Traversal* pada *Binary Tree* 1

1. ***Pre-order* : R, T1, T2**
   - **kunjungi R**
   - **kunjungi T1 secara *pre-order***
   - **kunjungi T2 secara *pre-order***

2. ***In-order* : T1 , R, T2**
   - **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**



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

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

```
#ifndef tree_H
#define tree_H
#define Nil NULL
#define info(P) (P)->info
#define right(P) (P)->right
#define left(P) (P)->left
```

typedef int infotype; typedef struct Node *address; struct Node{ infotype info; add
right;
address left;
};
typedef address BinTree; // fungsi primitif pohon biner
/******** pengecekan apakah tree kosong ********/ boolean EmptyTree(Tree T);
/* mengembalikan nilai true jika tree kosong */

/******** pembuatan tree kosong ********/ void CreateTree(Tree &T);
/* I.S sembarang
F.S. terbentuk Tree kosong */

/******** manajemen memori ********/ address alokasi(infotype X);
/* mengirimkan address dari alokasi sebuah elemen
jika alokasi berhasil maka nilai address tidak Nil dan jika gagal nilai address Nil*/

void Dealokasi(address P);
/* I.S P terdefinisi
F.S. memori yang digunakan P dikembalikan ke sistem */

/* Konstruktor */
address createElemen(infotype X, address L, address R)

31	
32	
33	
34	
35	
36	
37	
38	

```
39      /*  menghasilkan sebuah elemen tree dengan info X dan elemen kiri L dan
40  elemen kanan R
41       mencari elemen tree tertentu */
42
43      address findElmBinTree(Tree T, infotype X);
44      /* mencari apakah ada elemen tree dengan info(P) = X
45       jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */
46
47      address findLeftBinTree(Tree T, infotype X);
48      /* mencari apakah ada elemen sebelah kiri dengan info(P) = X
49       jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */
50
51      address findRigthBinTree(Tree T, infotype X);
52      /* mencari apakah ada elemen sebelah kanan dengan info(P) = X
53       jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */
54      /*insert elemen tree */
55      void InsertBinTree(Tree T, address P);
56      /*  I.S P Tree bisa saja kosong
57       F.S. memasukka p ke dalam tree terurut sesuai konsep binary tree
58              menghapus elemen tree tertentu*/ void DelBinTree(Tree &T, address P);
59      /*  I.S P Tree tidak kosong
60       F.S. menghapus p dari Tree selector */
61
62      infotype akar(Tree T);
63      /* mengembalikan nilai dari akar */
64
65      void PreOrder(Tree &T); /*  I.S P Tree tidak kosong
66
67
68
```

```
69          F.S. menampilkan Tree secara PreOrder */
70  void InOrder(Tree &T);
71  /*  I.S P Tree tidak kosong
            F.S. menampilkan Tree secara IOrder */
72
73          void PostOrder(Tree &T); /*  I.S P Tree tidak kosong
74           F.S. menampilkan Tree secara PostOrder */
75
76          #endif
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**
>
> **> fungsi alokasi(x : infotype) : address prosedur insertNode(i/o root : address, i: x : infotype) function findNode(x : infotype, root : address) : address procedure printInorder(root : address)**

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

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

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

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

        }
```

Gambar 10-10 Main.cpp

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

Gambar 10- 11 *Output*

2. **Buatlah fungsi untuk menghitung jumlah *node* dengan fungsi berikut.**
   - **fungsi hitungJumlahNode(root:address) : integer**

/* fungsi mengembalikan integer banyak node yang ada di dalam BST*/

   - **fungsi hitungTotalInfo(root:address, start:integer) : integer**

/* fungsi mengembalikan jumlah (total) info dari node-node yang ada di dalam BST*/

- **fungsi hitungKedalaman(root:address, start:integer) : integer**

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

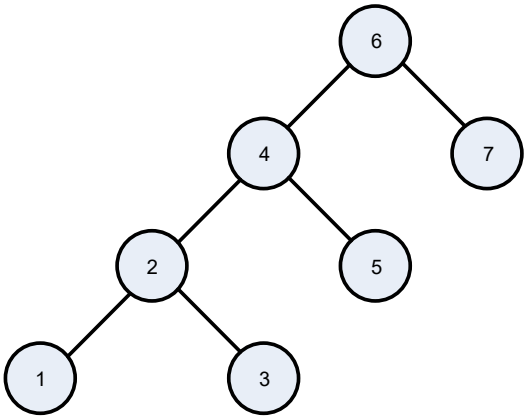
```
int main() {

        cout << "Hello World" << endl;     address root = NULL;
        insertNode(root,1);   insertNode(root,2);   insertNode(root,6);
        insertNode(root,4);   insertNode(root,5);   insertNode(root,3);
        insertNode(root,6);   insertNode(root,7);   InOrder(root);
        cout<<"\n";

        cout<<"kedalaman : "<<hitungKedalaman(root,0)<<endl;
        cout<<"jumlah Node : "<<hitungNode(root)<<endl;
        cout<<"total : "<<hitungTotal(root)<<endl;        return 0;

}
```

Gambar 10-12 *Main*

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

Gambar 10-13 *Output*

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



3. Guided

Code Progam:

main.cpp

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

/// PROGRAM BINARY TREE

// Struktur data pohon biner untuk menyimpan data dan pointer ke anak kiri, kanan, dan induk
struct Pohon {
    char data;                  // Data yang disimpan di node (tipe char)
    Pohon *left, *right;        // Pointer ke anak kiri dan anak kanan
    Pohon *parent;              // Pointer ke node induk
};

// Variabel global untuk menyimpan root (akar) pohon dan node baru
Pohon *root, *baru;

// Inisialisasi pohon agar kosong
void init() {
    root = NULL; // Mengatur root sebagai NULL (pohon kosong)
}

// Mengecek apakah pohon kosong
bool isEmpty() {
    return root == NULL; // Mengembalikan true jika root adalah NULL
}

// Membuat node baru sebagai root pohon
void buatNode(char data) {
    if (isEmpty()) { // Jika pohon kosong
        root = new Pohon{data, NULL, NULL, NULL}; // Membuat node baru sebagai root
        cout << "\nNode " << data << " berhasil dibuat menjadi root." << endl;
    } else {
        cout << "\nPohon sudah dibuat." << endl; // Root sudah ada, tidak membuat node baru
    }
}

// Menambahkan node baru sebagai anak kiri dari node tertentu
Pohon* insertLeft(char data, Pohon *node) {
```

```cpp
Pohon* insertLeft(char data, Pohon *node) {
    if (node->left != NULL) { // Jika anak kiri sudah ada
        cout << "\nNode " << node->data << " sudah ada child kiri!" << endl;
        return NULL; // Tidak menambahkan node baru
    }
    // Membuat node baru dan menghubungkannya ke node sebagai anak kiri
    baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kiri " << node->data << endl;
    return baru; // Mengembalikan pointer ke node baru
}

// Menambahkan node baru sebagai anak kanan dari node tertentu
Pohon* insertRight(char data, Pohon *node) {
    if (node->right != NULL) { // Jika anak kanan sudah ada
        cout << "\nNode " << node->data << " sudah ada child kanan!" << endl;
        return NULL; // Tidak menambahkan node baru
    }
    // Membuat node baru dan menghubungkannya ke node sebagai anak kanan
    baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kanan " << node->data << endl;
    return baru; // Mengembalikan pointer ke node baru
}

// Mengubah data di dalam sebuah node
void update(char data, Pohon *node) {
    if (!node) { // Jika node tidak ditemukan
        cout << "\nNode yang ingin diubah tidak ditemukan!" << endl;
        return;
    }
    char temp = node->data; // Menyimpan data lama
    node->data = data;      // Mengubah data dengan nilai baru
    cout << "\nNode " << temp << " berhasil diubah menjadi " << data << endl;
}

// Mencari node dengan data tertentu
```

main.cpp

```cpp
// Mencari node dengan data tertentu
void find(char data, Pohon *node) {
    if (!node) return; // Jika node tidak ada, hentikan pencarian

    if (node->data == data) { // Jika data ditemukan
        cout << "\nNode ditemukan: " << data << endl;
        return;
    }
    // Melakukan pencarian secara rekursif ke anak kiri dan kanan
    find(data, node->left);
    find(data, node->right);
}

// Traversal Pre-order (Node -> Kiri -> Kanan)
void preOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    cout << node->data << " "; // Cetak data node saat ini
    preOrder(node->left);      // Traversal ke anak kiri
    preOrder(node->right);     // Traversal ke anak kanan
}

// Traversal In-order (Kiri -> Node -> Kanan)
void inOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    inOrder(node->left);  // Traversal ke anak kiri
    cout << node->data << " "; // Cetak data node saat ini
    inOrder(node->right); // Traversal ke anak kanan
}

// Traversal Post-order (Kiri -> Kanan -> Node)
void postOrder(Pohon *node) {
    if (!node) return; // Jika node kosong, hentikan traversal
    postOrder(node->left);  // Traversal ke anak kiri
    postOrder(node->right); // Traversal ke anak kanan
    cout << node->data << " "; // Cetak data node saat ini
}
```

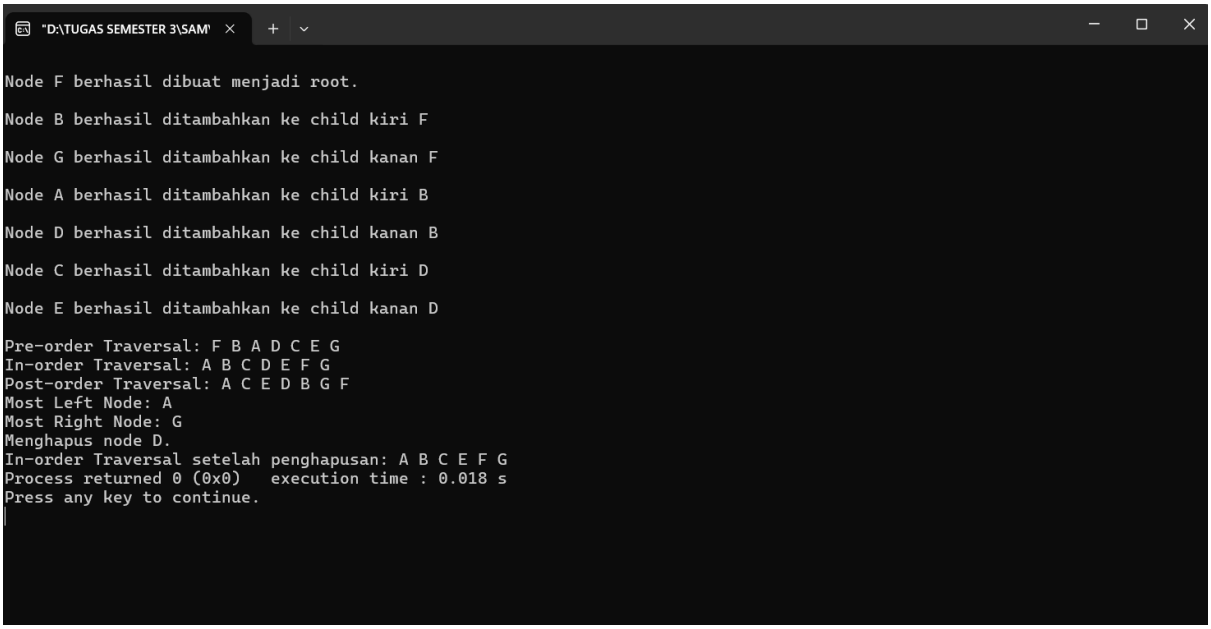
main.cpp

```cpp

// Menghapus node dengan data tertentu
Pohon* deleteNode(Pohon *node, char data) {
    if (!node) return NULL; // Jika node kosong, hentikan

    // Rekursif mencari node yang akan dihapus
    if (data < node->data) {
        node->left = deleteNode(node->left, data); // Cari di anak kiri
    } else if (data > node->data) {
        node->right = deleteNode(node->right, data); // Cari di anak kanan
    } else {
        // Jika node ditemukan
        if (!node->left) { // Jika tidak ada anak kiri
            Pohon *temp = node->right; // Anak kanan menggantikan posisi node
            delete node;
            return temp;
        } else if (!node->right) { // Jika tidak ada anak kanan
            Pohon *temp = node->left; // Anak kiri menggantikan posisi node
            delete node;
            return temp;
        }

        // Jika node memiliki dua anak, cari node pengganti (successor)
        Pohon *successor = node->right;
        while (successor->left) successor = successor->left; // Cari node terkecil di anak kanan
        node->data = successor->data; // Gantikan data dengan successor
        node->right = deleteNode(node->right, successor->data); // Hapus successor
    }
    return node;
}

// Menemukan node paling kiri
Pohon* mostLeft(Pohon *node) {
    if (!node) return NULL; // Jika node kosong, hentikan
    while (node->left) node = node->left; // Iterasi ke anak kiri hingga mentok
    return node;
}
```

```cpp

// Menemukan node paling kanan
Pohon* mostRight(Pohon *node) {
    if (!node) return NULL; // Jika node kosong, hentikan
    while (node->right) node = node->right; // Iterasi ke anak kanan hingga mentok
    return node;
}

// Fungsi utama
int main() {
    init(); // Inisialisasi pohon
    buatNode('F'); // Membuat root dengan data 'F'
    insertLeft('B', root); // Menambahkan 'B' ke anak kiri root
    insertRight('G', root); // Menambahkan 'G' ke anak kanan root
    insertLeft('A', root->left); // Menambahkan 'A' ke anak kiri dari 'B'
    insertRight('D', root->left); // Menambahkan 'D' ke anak kanan dari 'B'
    insertLeft('C', root->left->right); // Menambahkan 'C' ke anak kiri dari 'D'
    insertRight('E', root->left->right); // Menambahkan 'E' ke anak kanan dari 'D'

    // Traversal pohon
    cout << "\nPre-order Traversal: ";
    preOrder(root);
    cout << "\nIn-order Traversal: ";
    inOrder(root);
    cout << "\nPost-order Traversal: ";
    postOrder(root);

    // Menampilkan node paling kiri dan kanan
    cout << "\nMost Left Node: " << mostLeft(root)->data;
    cout << "\nMost Right Node: " << mostRight(root)->data;

    // Menghapus node
    cout << "\nMenghapus node D.";
    root = deleteNode(root, 'D');
    cout << "\nIn-order Traversal setelah penghapusan: ";
    inOrder(root);
```

```cpp
    cout << "\nIn-order Traversal setelah penghapusan: ";
    inOrder(root);

    return 0;
}
```

Hasil Outputnya:



4. Unguided

Code Progam:

ain.cpp

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

struct Pohon {
    char data;
    Pohon *left, *right, *parent;
};

Pohon *root = nullptr;

void init() {
    root = nullptr;
}

bool isEmpty() {
    return root == nullptr;
}

Pohon* buatNode(char data, Pohon* parent = nullptr) {
    Pohon* baru = new Pohon;
    baru->data = data;
    baru->left = baru->right = nullptr;
    baru->parent = parent;
    return baru;
}

void insertLeft(char data, Pohon* node) {
    if (node->left != nullptr) {
        cout << "Node " << node->data << " sudah memiliki child kiri!" << endl;
    } else {
        node->left = buatNode(data, node);
        cout << "Node " << data << " berhasil ditambahkan ke child kiri " << node->data << endl;
    }
}

void insertRight(char data, Pohon* node) {
    if (node->right != nullptr) {
        cout << "Node " << node->data << " sudah memiliki child kanan!" << endl;
```

main.cpp

```
    } else {
        node->right = buatNode(data, node);
        cout << "Node " << data << " berhasil ditambahkan ke child kanan " << node->data << endl;
    }
}

void displayChild(Pohon* node) {
    if (node == nullptr) {
        cout << "Node tidak ditemukan." << endl;
        return;
    }
    cout << "Node: " << node->data << endl;
    cout << "Left Child: " << (node->left ? node->left->data : '-') << endl;
    cout << "Right Child: " << (node->right ? node->right->data : '-') << endl;
}

void displayDescendants(Pohon* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    displayDescendants(node->left);
    displayDescendants(node->right);
}

bool is_valid_bst(Pohon* node, char min_val, char max_val) {
    if (node == nullptr) return true;
    if (node->data <= min_val || node->data >= max_val) return false;
    return is_valid_bst(node->left, min_val, node->data) &&
           is_valid_bst(node->right, node->data, max_val);
}

int cari_simpul_daun(Pohon* node) {
    if (node == nullptr) return 0;
    if (node->left == nullptr && node->right == nullptr) return 1;
    return cari_simpul_daun(node->left) + cari_simpul_daun(node->right);
}

void inOrder(Pohon* node) {
    if (node == nullptr) return;
    inOrder(node->left);
```

main.cpp

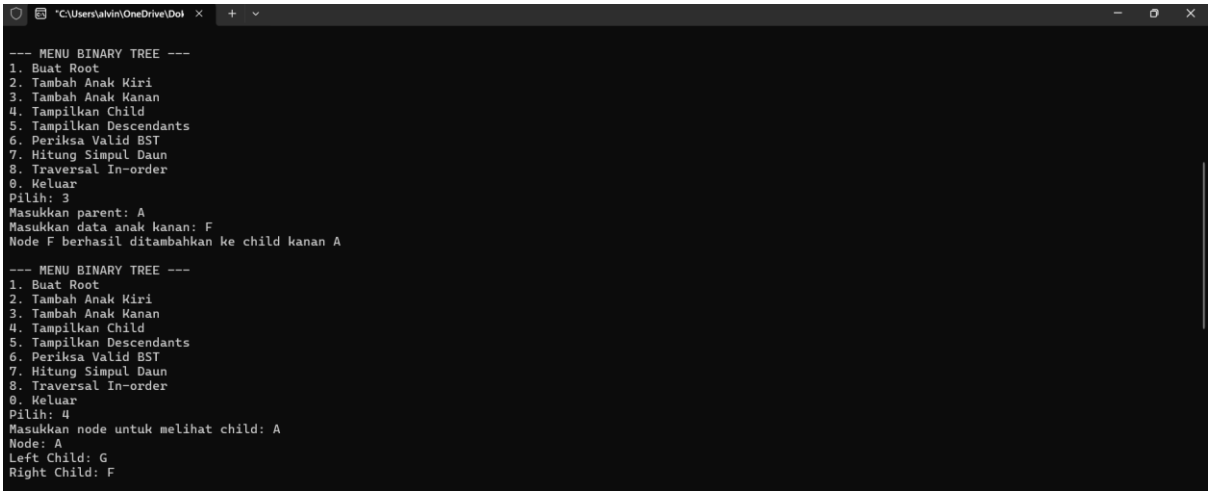
```
    cout << node->data << " ";
    inOrder(node->right);
}

void menu() {
    int choice;
    char data, parentData;
    Pohon* parentNode;

    do {
        cout << "\n--- MENU BINARY TREE ---\n";
        cout << "1. Buat Root\n";
        cout << "2. Tambah Anak Kiri\n";
        cout << "3. Tambah Anak Kanan\n";
        cout << "4. Tampilkan Child\n";
        cout << "5. Tampilkan Descendants\n";
        cout << "6. Periksa Valid BST\n";
        cout << "7. Hitung Simpul Daun\n";
        cout << "8. Traversal In-order\n";
        cout << "0. Keluar\n";
        cout << "Pilih: ";
        cin >> choice;

        switch (choice) {
            case 1:
                if (!isEmpty()) {
                    cout << "Root sudah dibuat!" << endl;
                } else {
                    cout << "Masukkan data root: ";
                    cin >> data;
                    root = buatNode(data);
                    cout << "Root " << data << " berhasil dibuat." << endl;
                }
                break;

            case 2:
                if (isEmpty()) {
                    cout << "Pohon belum dibuat. Buat root terlebih dahulu!" << endl;
                } else {
```

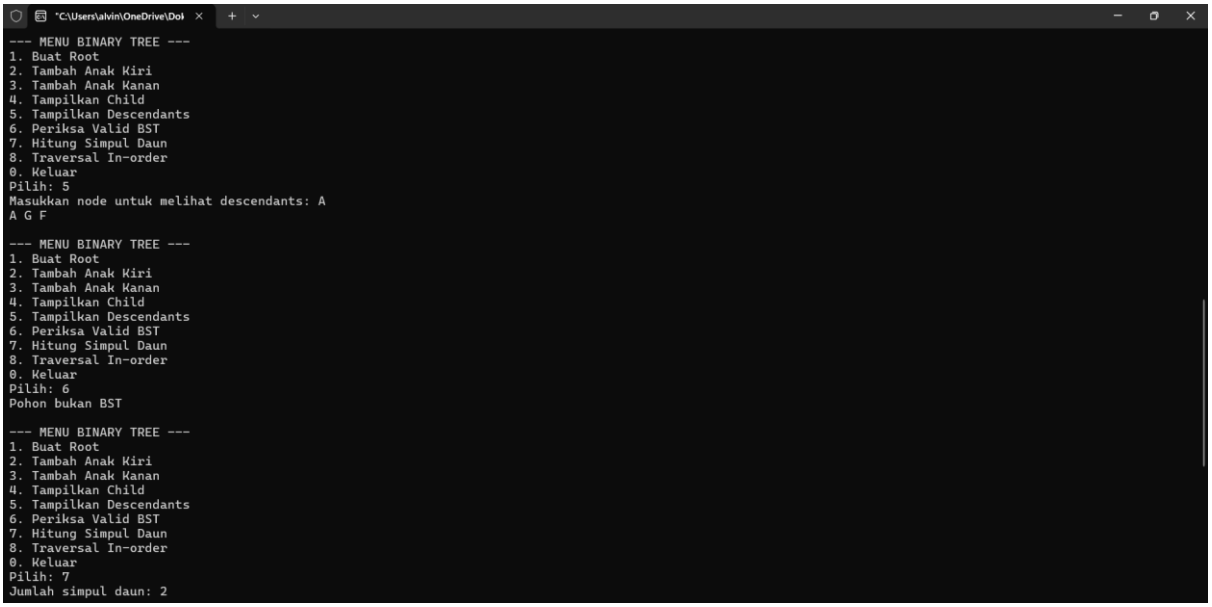
main.cpp

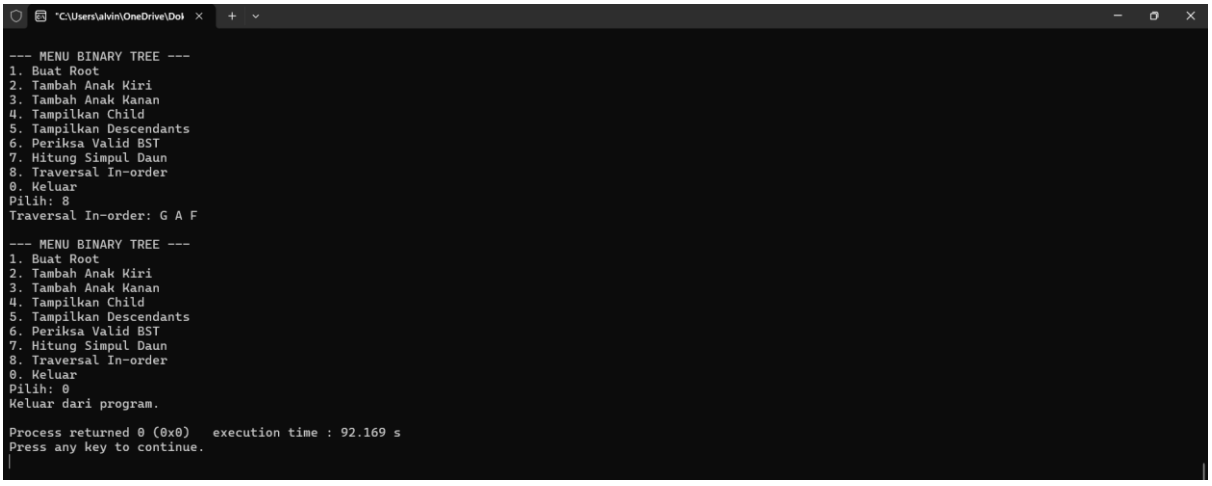
```
                    cout << "Masukkan parent: ";
                    cin >> parentData;
                    cout << "Masukkan data anak kiri: ";
                    cin >> data;
                    parentNode = root;
                    insertLeft(data, parentNode);
                }
                break;

            case 3:
                if (isEmpty()) {
                    cout << "Pohon belum dibuat. Buat root terlebih dahulu!" << endl;
                } else {
                    cout << "Masukkan parent: ";
                    cin >> parentData;
                    cout << "Masukkan data anak kanan: ";
                    cin >> data;
                    parentNode = root;
                    insertRight(data, parentNode);
                }
                break;

            case 4:
                if (isEmpty()) {
                    cout << "Pohon kosong." << endl;
                } else {
                    cout << "Masukkan node untuk melihat child: ";
                    cin >> data;
                    displayChild(root);
                }
                break;

            case 5:
                if (isEmpty()) {
                    cout << "Pohon kosong." << endl;
                } else {
                    cout << "Masukkan node untuk melihat descendants: ";
                    cin >> data;
                    displayDescendants(root);
                    cout << endl;
                }
                break;

            case 6:
                cout << (is_valid_bst(root, CHAR_MIN, CHAR_MAX) ? "Pohon adalah BST" : "Pohon bukan BST") << endl;
                break;

            case 7:
                cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << endl;
                break;

            case 8:
                cout << "Traversal In-order: ";
                inOrder(root);
                cout << endl;
                break;

            case 0:
                cout << "Keluar dari program." << endl;
                break;

            default:
                cout << "Pilihan tidak valid!" << endl;
        }
    } while (choice != 0);
}

int main() {
    init();
    menu();
    return 0;
}
```

Hasil Outputnya:

```
"C:\Users\alvin\OneDrive\Dok

--- MENU BINARY TREE ---
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Periksa Valid BST
7. Hitung Simpul Daun
8. Traversal In-order
0. Keluar
Pilih: 1
Masukkan data root: A
Root A berhasil dibuat.

--- MENU BINARY TREE ---
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Periksa Valid BST
7. Hitung Simpul Daun
8. Traversal In-order
0. Keluar
Pilih: 2
Masukkan parent: A
Masukkan data anak kiri: G
Node G berhasil ditambahkan ke child kiri A
```







5. Kesimpulan

Rekursif adalah metode dalam pemrograman di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah. Metode ini membuat program menjadi

lebih sederhana, mudah dibaca, dan efisien dalam menyelesaikan masalah yang berulang. Namun, rekursif juga membutuhkan lebih banyak memori dan waktu karena setiap pemanggilan fungsi menyimpan informasi.

Salah satu struktur data yang sering menggunakan rekursi adalah pohon (tree). Pohon adalah struktur data non-linear yang terdiri dari beberapa elemen yang saling terhubung. Setiap elemen disebut node, dan terdapat tiga komponen utama dalam pohon: akar (root), anak (children), dan daun (leaf). Salah satu jenis pohon yang sering digunakan adalah Binary Tree, yaitu pohon di mana setiap node hanya bisa memiliki dua anak.

Pada Binary Search Tree (BST), ada aturan khusus: anak kiri dari sebuah node selalu lebih kecil, sementara anak kanan lebih besar dari node tersebut. Dalam BST, kita bisa melakukan beberapa operasi seperti penyisipan (insert), pembaruan (update), pencarian (search), dan traversal (mengunjungi setiap node). Ada tiga jenis traversal utama dalam BST, yaitu pre-order, in-order, dan post-order. Dalam implementasinya, pohon biner biasanya disusun menggunakan linked list, dengan fungsi untuk alokasi memori, penyisipan, pencarian, dan traversal node.