LAPORAN PRAKTIKUM
Pertemuan 09
Modul 10-11
Tree

Disusun Oleh:
Zhafir Zaidan Avail
S1-SE-07-2

Dosen :
Wahyu Andi Saputra, S.Pd., M.Eng

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.
    5. Mengimplementasikan struktur data tree, khususnya Binary Tree.
2. **Landasan Teori**
    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.Algoritma rekursif untuk menghitung hasil pangkat 2 sebagai berikut:

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

    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).

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.

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
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)
	#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");
	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)
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
	#include <conio.h>
	#include <iostream>
	
	long int faktorial(long int a);
	
	main(){
	 long int faktor;
	 long int n;
	 cout<<"Masukkan nilai faktorial ";
	 cin>>n;
	 faktor =faktorial(n);
	 cout<<n<<"!="<<faktor<<endl;
	 getch();
	}
	
	long int faktorial(long int y){
	 if (a==1 || a==0){
	 return(1);
	 }else if (a>1){
	 return(a*faktorial(a-1));
	 }else{
	 return 0;
	 }
	}

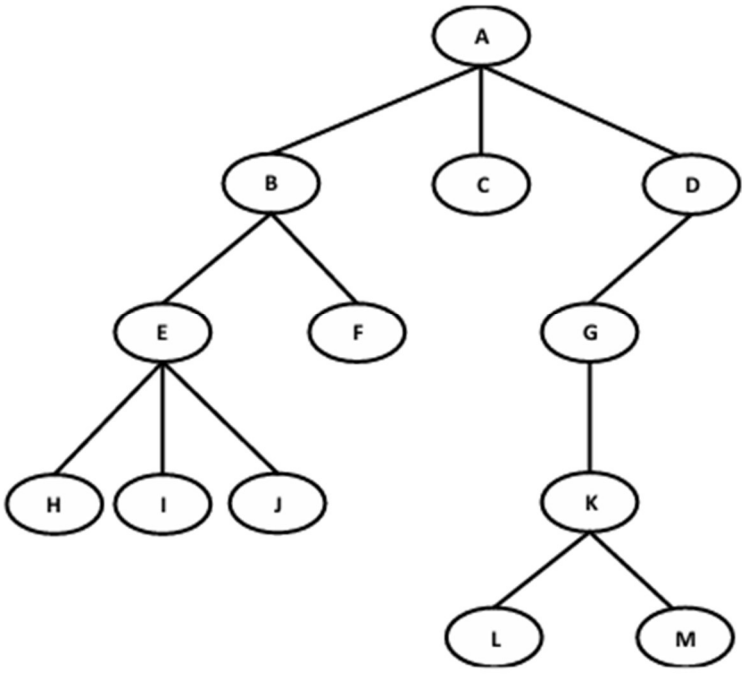
5. Pengertian Tree

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

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

Karateristik dari suatu tree T adalah :

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



Gambar 10-1 Tree

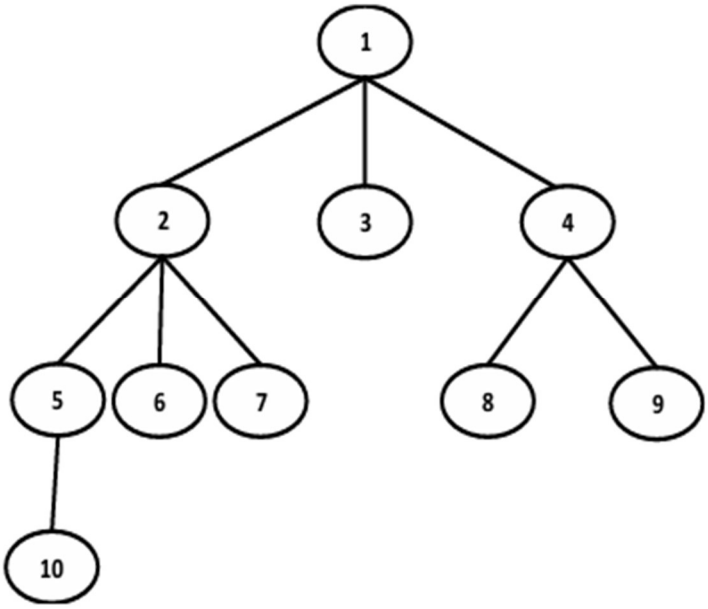
Berdasarkan gambar diatas dapat digambarkan beberapa terminologinya, yaitu

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

6. Jenis-Jenis Tree

i. Ordered Tree

Yaitu pohon yang urutan anak-anaknya penting.



Gambar 10-2 Ordered Tree

ii. 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 2r.

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 Tn yang unik memiliki n nodes. Untuk menentukan jumlah left children dan right children tree Tn 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.

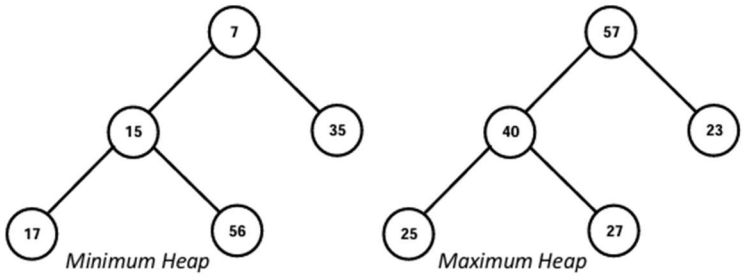
D. AVL Tree

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.



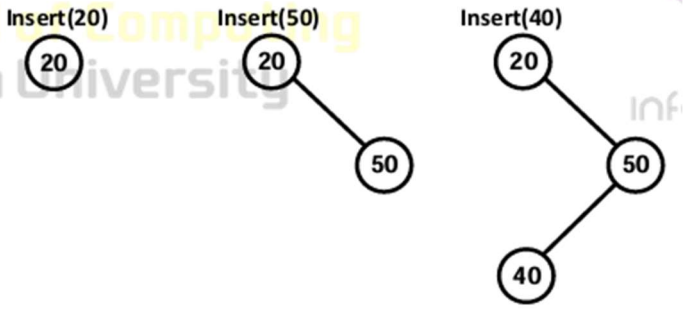
Gambar 10-3 Heap Tree

7. Operasi-Operasi dalam Binary Search Tree

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

i. 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

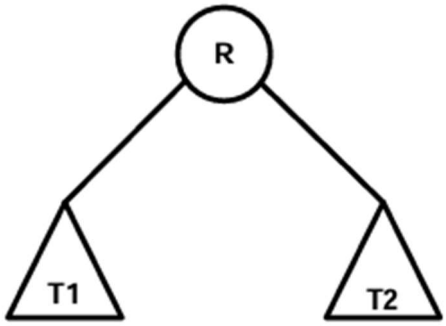
```
struct node{
int key;
struct node *left, *right;
};
// sebuah fungsi utilitas untuk membuat sebuah node BST
struct node *newNode(int item){
struct node *temp = (struct node*)malloc(sizeof(struct node));
key(temp) = item;
left(item)= NULL;
right(item)= NULL;
return temp;
}
/* sebuah fungsi utilitas untuk memasukan sebuah node dengan kunci yang
diberikan kedalam BST */
struct node* insert(struct node* node, int key)
{
/* jika tree kosong, return node yang baru */
if (node == NULL){
return newNode(key); }
/* jika tidak, kembali ke tree */
if (key < key(node))
left(node) = insert(left(node,key));
else if (key > key(node))
right(node) = insert(right(node,key));
/* mengeluarkan pointer yang tidak berubah */
return node;
```

```
}
```

ii. **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.

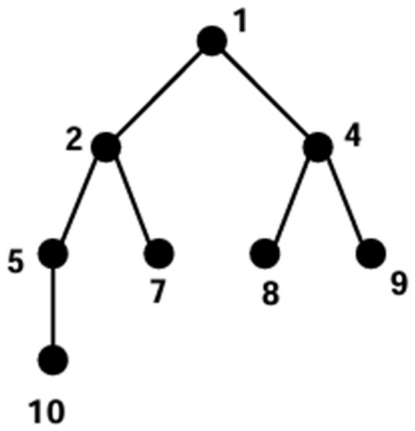
iii. **Search**
Proses pencarian elemen pada binary tree dapat menggunakan algoritma rekursif binary search. Berikut adalah algoritma binary search :
1. Pencarian pada binary search tree dilakukan dengan menaruh pointer dan membandingkan nilai yang dicari dengan node awal (root)
2. Jika nilai yang dicari 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

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;
 address 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)
/*  menghasilkan sebuah elemen tree dengan info X dan elemen kiri L dan
elemen kanan R
```

```
 mencari elemen tree tertentu */

address findElmBinTree(Tree T, infotype X);
/* mencari apakah ada elemen tree dengan info(P) = X
 jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */

address findLeftBinTree(Tree T, infotype X);
/* mencari apakah ada elemen sebelah kiri dengan info(P) = X
 jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */

address findRigthBinTree(Tree T, infotype X);
/* mencari apakah ada elemen sebelah kanan dengan info(P) = X
 jika ada mengembalikan address element tsb, dan Nil jika sebalikanya */

/*insert elemen tree */
void InsertBinTree(Tree T, address P);
/*  I.S P Tree bisa saja kosong
 F.S. memasukka p ke dalam tree terurut sesuai konsep binary tree
   menghapus elemen tree tertentu*/
void DelBinTree(Tree &T, address P);
/*  I.S P Tree tidak kosong
 F.S. menghapus p dari Tree selector */

infotype akar(Tree T);
/* mengembalikan nilai dari akar */

void PreOrder(Tree &T);
/*  I.S P Tree tidak kosong
 F.S. menampilkan Tree secara PreOrder */

void InOrder(Tree &T);
/*  I.S P Tree tidak kosong
 F.S. menampilkan Tree secara IOrder */

void PostOrder(Tree &T);
/*  I.S P Tree tidak kosong
 F.S. menampilkan Tree secara PostOrder */

#endif
```

3. Guided

```
#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) {
    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
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
}

// 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;
}

// 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);

    return 0;
}
```

4. Unguided

```
#include <iostream>
#include <queue>
#include <limits>
#include <climits> // Untuk nilai INT_MIN dan INT_MAX
using namespace std;

struct TreeNode {
    char value;
    TreeNode *left, *right, *parent;

    TreeNode(char value, TreeNode *left = NULL, TreeNode *right = NULL,
TreeNode *parent = NULL)
        : value(value), left(left), right(right), parent(parent) {}
};

TreeNode *root;
```

```
void initialize() {
    root = NULL;
}

bool isTreeEmpty() {
    return root == NULL;
}

void createRootNode(char value) {
    if (isTreeEmpty()) {
        root = new TreeNode(value);
        cout << "Node " << value << " berhasil dibuat sebagai root." <<
endl;
    } else {
        cout << "Root sudah ada." << endl;
    }
}

TreeNode *insertLeftNode(char value, TreeNode *node) {
    if (!node) {
        cout << "Node parent tidak ditemukan." << endl;
        return NULL;
    }
    if (node->left != NULL) {
        cout << "Node " << node->value << " sudah memiliki anak kiri."
<< endl;
        return node->left;
    }
    TreeNode *newNode = new TreeNode(value, NULL, NULL, node);
    node->left = newNode;
    cout << "Node " << value << " berhasil ditambahkan sebagai anak kiri
dari " << node->value << "." << endl;
    return newNode;
}

TreeNode *insertRightNode(char value, TreeNode *node) {
    if (!node) {
        cout << "Node parent tidak ditemukan." << endl;
        return NULL;
    }
    if (node->right != NULL) {
        cout << "Node " << node->value << " sudah memiliki anak kanan."
<< endl;
        return node->right;
    }
    TreeNode *newNode = new TreeNode(value, NULL, NULL, node);
    node->right = newNode;
    cout << "Node " << value << " berhasil ditambahkan sebagai anak
kanan dari " << node->value << "." << endl;
    return newNode;
}

void preOrderTraversal(TreeNode *node) {
    if (!node) return;
    cout << node->value << " ";
    preOrderTraversal(node->left);
    preOrderTraversal(node->right);
}

void inOrderTraversal(TreeNode *node) {
    if (!node) return;
    inOrderTraversal(node->left);
    cout << node->value << " ";
    inOrderTraversal(node->right);
}
```

```
void postOrderTraversal(TreeNode *node) {
    if (!node) return;
    postOrderTraversal(node->left);
    postOrderTraversal(node->right);
    cout << node->value << " ";
}

void findNode(char value, TreeNode *node, TreeNode *&result) {
    if (!node) return;
    if (node->value == value) {
        result = node;
        return;
    }
    findNode(value, node->left, result);
    findNode(value, node->right, result);
}

void displayChildren(TreeNode *node) {
    if (!node) {
        cout << "Node tidak ditemukan." << endl;
        return;
    }
    cout << "Node " << node->value << " memiliki anak: ";
    if (node->left) cout << "Kiri: " << node->left->value << " ";
    if (node->right) cout << "Kanan: " << node->right->value << " ";
    if (!node->left && !node->right) cout << "tidak ada.";
    cout << endl;
}

void displayDescendants(TreeNode *node) {
    if (!node) {
        cout << "Node tidak ditemukan." << endl;
        return;
    }
    cout << "Descendant dari node " << node->value << ": ";
    queue<TreeNode *> q;
    q.push(node);
    while (!q.empty()) {
        TreeNode *current = q.front();
        q.pop();
        if (current->left) {
            cout << current->left->value << " ";
            q.push(current->left);
        }
        if (current->right) {
            cout << current->right->value << " ";
            q.push(current->right);
        }
    }
    cout << endl;
}

bool is_valid_bst(TreeNode *node, int min_val, int max_val) {
    if (!node) return true;

    if (node->value <= min_val || node->value >= max_val) return false;

    return is_valid_bst(node->left, min_val, node->value) &&
           is_valid_bst(node->right, node->value, max_val);
}

int countLeafNodes(TreeNode *node) {
    if (!node) return 0;

    if (!node->left && !node->right) return 1;
```

```
    return countLeafNodes(node->left) + countLeafNodes(node->right);
}

int main() {
    initialize();
    int choice;
    char value, parentValue;
    TreeNode *parent = NULL;

    do {
        cout << "\n=== Menu Pohon Biner ===";
        cout << "\n1. Tambah Root";
        cout << "\n2. Tambah Anak Kiri";
        cout << "\n3. Tambah Anak Kanan";
        cout << "\n4. Tampilkan Pre-order Traversal";
        cout << "\n5. Tampilkan In-order Traversal";
        cout << "\n6. Tampilkan Post-order Traversal";
        cout << "\n7. Tampilkan Anak Node";
        cout << "\n8. Tampilkan Descendant Node";
        cout << "\n9. Periksa Apakah BST";
        cout << "\n10. Hitung Jumlah Simpul Daun";
        cout << "\n0. Keluar";
        cout << "\nPilihan Anda: ";
        cin >> choice;

        if (cin.fail()) { // Jika terjadi kesalahan input
            cin.clear();  // Bersihkan kesalahan pada cin
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); //
Hapus sisa input
            cout << "Input tidak valid. Silakan masukkan angka.\n";
            continue; // Ulangi loop tanpa keluar
        }

        cin.ignore(numeric_limits<streamsize>::max(), '\n'); //
Bersihkan buffer input setelah membaca

        switch (choice) {
            case 1:
                cout << "Masukkan data root: ";
                cin >> value;
                cin.ignore(); // Bersihkan buffer input
                createRootNode(value);
                break;
            case 2:
                cout << "Masukkan data parent: ";
                cin >> parentValue;
                cin.ignore();
                findNode(parentValue, root, parent);
                if (parent) {
                    cout << "Masukkan data anak kiri: ";
                    cin >> value;
                    cin.ignore();
                    insertLeftNode(value, parent);
                } else {
                    cout << "Parent tidak ditemukan." << endl;
                }
                break;
            case 3:
                cout << "Masukkan data parent: ";
                cin >> parentValue;
                cin.ignore();
                findNode(parentValue, root, parent);
                if (parent) {
                    cout << "Masukkan data anak kanan: ";
                    cin >> value;
```

```
                    cin.ignore();
                    insertRightNode(value, parent);
                } else {
                    cout << "Parent tidak ditemukan." << endl;
                }
                break;
            case 4:
                cout << "Pre-order Traversal: ";
                preOrderTraversal(root);
                cout << endl;
                break;
            case 5:
                cout << "In-order Traversal: ";
                inOrderTraversal(root);
                cout << endl;
                break;
            case 6:
                cout << "Post-order Traversal: ";
                postOrderTraversal(root);
                cout << endl;
                break;
            case 7:
                cout << "Masukkan data node: ";
                cin >> value;
                cin.ignore();
                findNode(value, root, parent);
                displayChildren(parent);
                break;
            case 8:
                cout << "Masukkan data node: ";
                cin >> value;
                cin.ignore();
                findNode(value, root, parent);
                displayDescendants(parent);
                break;
            case 9:
                cout << "Apakah pohon ini adalah BST? "
                     << (is_valid_bst(root, INT_MIN, INT_MAX) ? "Ya" :
"Tidak") << endl;
                break;
            case 10:
                cout << "Jumlah simpul daun: " << countLeafNodes(root)
<< endl;
                break;
            case 0:
                cout << "Keluar dari program." << endl;
                break;
            default:
                cout << "Pilihan tidak valid. Silakan coba lagi." <<
endl;
        }

    } while (choice != 0);

    return 0;
}
```

Output

```
=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
```

```
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 1
Masukkan data root: 2
Node 2 berhasil dibuat sebagai root.

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 2
Masukkan data parent: 2
Masukkan data anak kiri: 1
Node 1 berhasil ditambahkan sebagai anak kiri dari 2.

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 2
Masukkan data parent: 1
Masukkan data anak kiri: 5
Node 5 berhasil ditambahkan sebagai anak kiri dari 1.

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 3
Masukkan data parent: 1
Masukkan data anak kanan: 1
Node 1 berhasil ditambahkan sebagai anak kanan dari 1.

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
```

```
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 3
Masukkan data parent: 2
Masukkan data anak kanan: 3
Node 3 berhasil ditambahkan sebagai anak kanan dari 2.

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 4
Pre-order Traversal: 2 1 5 1 3

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 5
In-order Traversal: 5 1 1 2 3

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 6
Post-order Traversal: 5 1 1 3 2

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
```

```
0. Keluar
Pilihan Anda: 7
Masukkan data node: 2
Node 2 memiliki anak: Kiri: 1 Kanan: 3

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 8
Masukkan data node: 1
Descendant dari node 1: 5 1

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 9
Apakah pohon ini adalah BST? Tidak

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 10
Jumlah simpul daun: 3

=== Menu Pohon Biner ===
1. Tambah Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Pre-order Traversal
5. Tampilkan In-order Traversal
6. Tampilkan Post-order Traversal
7. Tampilkan Anak Node
8. Tampilkan Descendant Node
9. Periksa Apakah BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan Anda: 0
Keluar dari program.
```

5. Kesimpulan

Rekursif adalah proses dalam program yang memungkinkan sebuah fungsi memanggil dirinya sendiri. Dengan manfaat seperti meningkatkan keterbacaan, modularitas, dan efisiensi penulisan kode, rekursif mempermudah pengembangan program, terutama untuk masalah yang solusinya bersifat berulang. Prinsip utama rekursif melibatkan dua elemen: kondisi pemutus dan pemanggilan diri sendiri, yang diimplementasikan melalui struktur logika yang teratur seperti pernyataan bersyarat.

Rekursif sering digunakan untuk memecahkan masalah seperti faktorial, penghitungan pangkat, dan manipulasi struktur data seperti pohon. Rekursif memberikan cara yang sederhana untuk menyelesaikan masalah yang kompleks. Namun, penggunaannya memiliki kelemahan, yaitu konsumsi memori lebih besar dan waktu eksekusi tambahan untuk menangani *activation records*. Oleh karena itu, rekursif ideal digunakan saat solusi iteratif sulit diimplementasikan atau ketika efisiensi bukan prioritas utama dibandingkan dengan kejelasan logika.

Tree adalah struktur data non-linear yang digunakan untuk merepresentasikan hubungan hierarkis antar data. Jenis-jenis tree seperti binary tree, complete binary tree, dan binary search tree (BST) digunakan untuk efisiensi penyimpanan dan pencarian data. Operasi seperti penyisipan, pencarian, pembaruan, dan traversal (pre-order, in-order, post-order) menjadikan tree sebagai struktur data penting dalam berbagai aplikasi. Namun, desain dan manajemen tree memerlukan pemahaman mendalam tentang properti seperti left child, right child, parent node, dan aturan-aturan pengurutan dalam BST.