LAPORAN PRAKTIKUM
Modul 10
“Tree Bagian Pertama”

Disusun Oleh:
Dimastian Aji Wibowo (2311104058)
SE-07-02

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY
PURWOKERTO
2024

Telkom University Purwokerto

1. Tujuan

- Memahami konsep penggunaan fungsi rekursif.
- Mengimplementasikan bentuk-bentuk fungsi rekursif.
- Mengaplikasikan struktur data tree dalam sebuah kasus pemrograman.
- Mengimplementasikan struktur data tree, khususnya Binary Tree.

2. Landasan Teori

A. 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 :

```
Fungsi 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

B. 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).

C. 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.

D. Contoh Rekursif

Rekursif berarti suatu fungsi dapat memanggil fungsi yang merupakan dirinya sendiri. Berikut adalah contoh program untuk rekursif menghitung nilai pangkat sebuah bilangan.

Algortima	C++
Program coba_rekursif	`#include <conio.h>`

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 <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.

Algortima	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
	`#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();`
	`}`

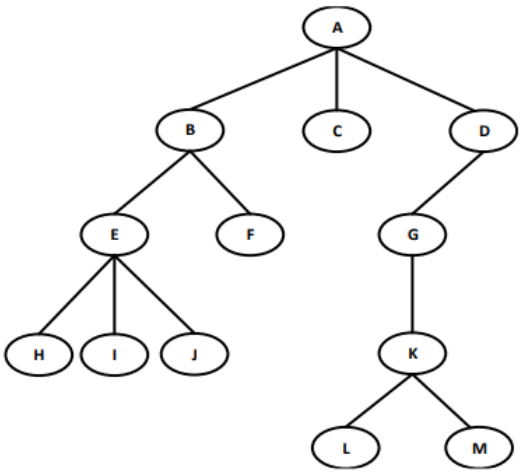
→ 0	`long int faktorial(long int y){`
	`if (a==1 || a==0){`
	`return(1);`
	`}else if (a>1){`
	`return(a*faktorial(a-1));`
	`}else{`
	`return 0;`
	`}`
	`}`

E. 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.



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

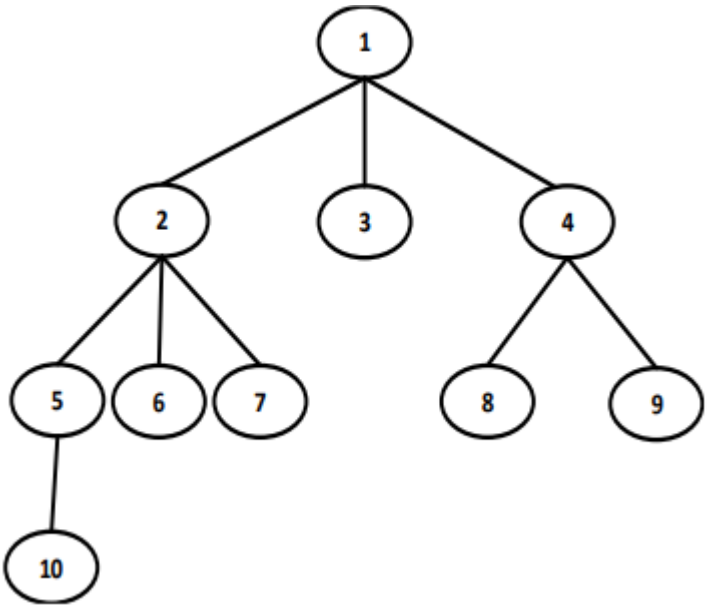
lintasan dari A ke J adalah 3.

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.

F. Jenis – jenis Tree

1. Ordered Tree

Yaitu pohon yang urutan anak-anaknya penting.



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 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.

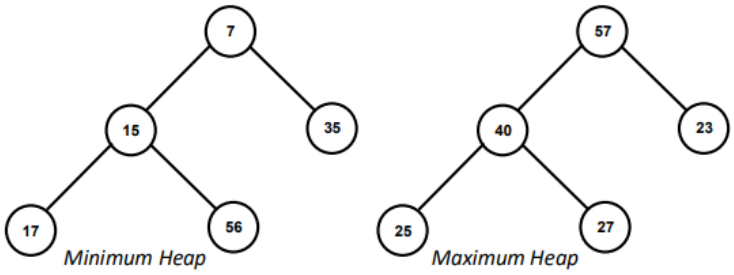
e. Heap Tree

   Adalah tree yang memenuhi persamaan berikut: R[i] < r[2i] and R[i] < r[2i+1] STRUKTUR DATA 93 Heap juga disebut Complete Binary Tree, karena jika suatu node mempunyai child, maka jumlah childnya 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.



G. Operasi – operasi dalam 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.

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int key;
    struct node *left, *right;
};

// Fungsi utilitas untuk membuat sebuah node BST
struct node *newNode(int item) {
    struct node *temp = (struct
node*)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* Fungsi utilitas untuk memasukkan sebuah node dengan
kunci
   yang diberikan ke dalam BST */
struct node* insert(struct node* node, int key) {
    // Jika tree kosong, return node yang baru
    if (node == NULL) {
        return newNode(key);
    }
```

```
    // Jika tidak, masukkan node pada subtree yang
sesuai
    if (key < node->key) {
        node->left = insert(node->left, key);
    }
    else if (key > node->key) {
        node->right = insert(node->right, key);
    }

    // Kembalikan pointer root yang tidak berubah
    return node;
}

// Fungsi untuk menampilkan BST dalam traversal
inorder
void inorder(struct node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Driver program untuk menguji kode
int main() {
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal dari BST: \n");
    inorder(root);

    return 0;
}
```

b. Update

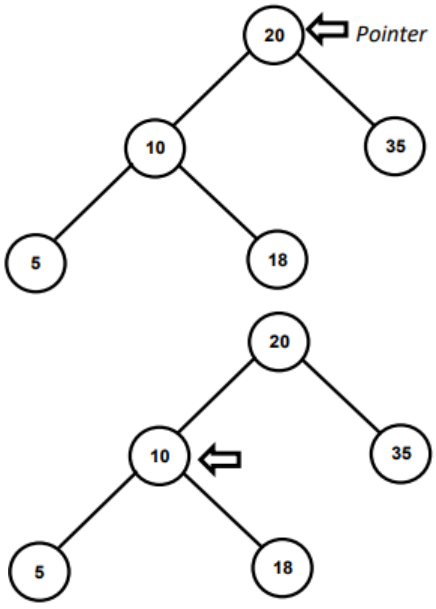
Jika setelah diupdate posisi/lokasi node yang bersangkutan tidak sesuai dengan ketentuan, maka harus dilakukan dengan proses REGENERASI agar tetap memenuhi kriteria Binary Search Tree

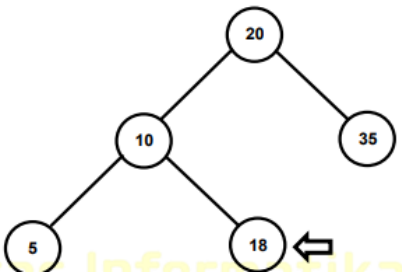
c. Search

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

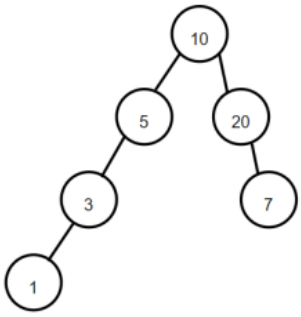
Nilai dicari : 18





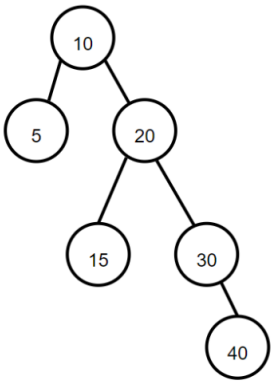
d. Delete
   1. LEAF, tidak perlu dilakukan modifikasi.
   2. Node dengan 1 Child, maka child langsung menggantikan posisi Parent.
   3. Node dengan 2 Child:
      - Left Subtree, yang diambil adalah node yang paling kiri (nilai terbesar).
      - Right Subtree, yang diambil adalah node yang paling kanan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

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.

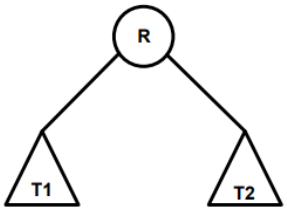


f. Most-Right
   Most-right node adalah node yang berada paling kanan dalam tree. Dalam konteks binary search tree (BST), most-right node adalah node dengan nilai terbesar, yang dapat ditemukan dengan mengikuti anak kanan (right child) dari root hingga mencapai node yang tidak memiliki

anak kanan lagi



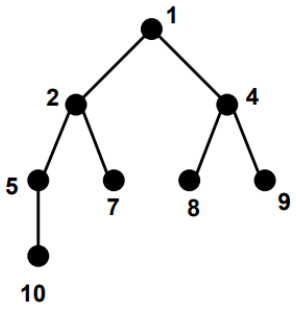
H. Traversal pada Binary Tree



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



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,
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 elemen tsb, dan Nil
jika sebaliknya */

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

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

/******** Insert Elemen Tree ********/
void InsertBinTree(Tree T, address P);
/* I.S P Tree bisa saja kosong
   F.S. Memasukkan 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 */

/******** Traversal Tree ********/
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 InOrder */

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

#endif
```

3. Guided

1. Membuat struct Pohon yang berisi char data untuk menyimpan nilai atau data pada node, Pohon *left untuk pointer ke anak kiri node, Pohon *right untuk pointer ke anak kanan node, dan Pohon *parent untuk pointer ke node induk.
2. Membuat variabel Pohon *root untuk menyimpan pointer ke akar pohon, dan Pohon *baru digunakan untuk membuat node baru.
3. Membuat fungsi init() untuk mengatur root menjadi null yang menandakan pohon kosong.
4. Fungsi isEmpty() untuk mengembalikan true jika root bernilai null yang menandakan pohon kosong.
5. Fungsi buatNode(char data) dengan memanggil isEmpty untuk memeriksa isEmpty() untuk memeriksa apakah pohon kosong, jika kosong maka root dibuat sebagai node baru dengan data yang diberikan, dan jika pohon sudah ada, maka menampilkan pesan bahwa root telah dibuat.

```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
    }
}
```

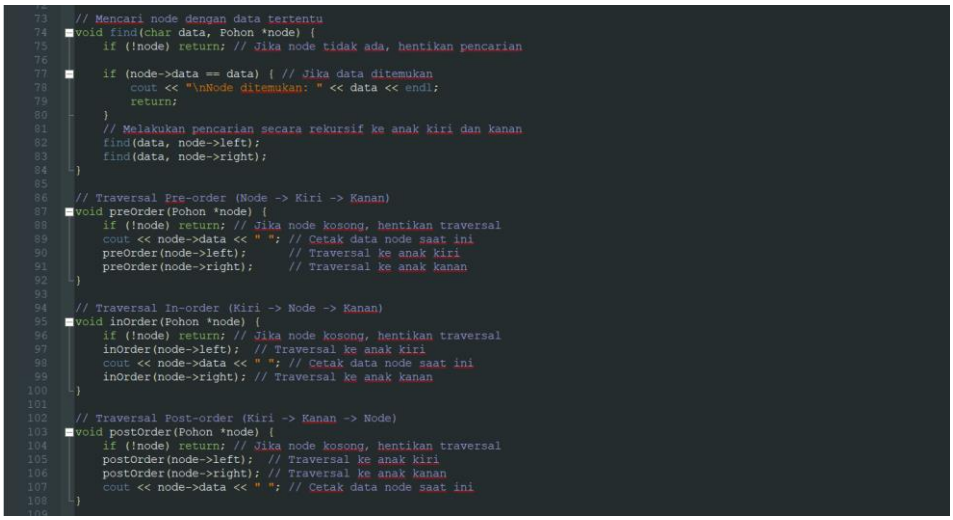
6. Fungsi insertLeft(char data, Pohon *node) digunakan untuk menambahkan node baru sebagai anak kiri dari node tertentu dengan memeriksa apakah anak kiri sudah ada, jika anak kiri sudah ada maka fungsi mengembalikan pesan bahwa anak kiri telah ada, dan jika tidak ada maka node baru dibuat dan dihubungkan ke node induk sebagai anak kiri.
7. Fungsi insertRight(char data, Pohon *node) mirip dengan fungsi insertLeft akan tetapi fungsi ini menambahkan node sebagai anak kanan dengan memeriksa apakah anak kanan sudah ada, jika belum maka membuat node baru dan menghubungkannya sebagai anak kanan.
8. Fungsi update(char data, Pohon *node) untuk mengubah data dalam sebuah node dengan data lama pada node disimpan sementara dan data node diperbarui dengan nilai baru.

```cpp
// 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;
}
```

9. Fungsi find(char data, Pohon *node) digunakan untuk mencari node dengan data tertentu secara rekursif dengan mencari node dan memeriksa

anak kiri dan kanan dari node saat ini dan jika data ditemukan maka menampilkann pesan bahwa node ditemukan.

10. Fungsi Traversal (Pre-order, In-order, Post-order) digunakan untuk mencetak elemen pohon dalam urutan tertentu, pre-order dengan cetak data saat ini lalu traversal anak kiri dan traversal anak kanan, in-order dengan traversal anak kiri lalu cetak data saat ini dan traversal anak kanan, post-order dengan traversal anak kiri lalu traversal anak kanan dan cetak data saat ini.



```
// 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
}
```

11. Fungsi deleteNode(Pohon *node, char data) digunakan untuk menghapus node dengan data tertentu dengan jika node tidak memiliki anak kiri maka node kanan akan menggantikannya, jika node tidak memiliki anak kanan maka node kiri akan menggantikannya, jika node memiliki dua anak maka node terkecil di subtree kanan digunakan untuk menggantikan data node yang dihapus.
12. mostLeft(Pohon *node) fungsi ini menemukan node paling kiri (anak kiri terdalam) dalam pohon.

```
// 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;
}
```

13. mostRight(Pohon *node) fungsi ini menemukan node paling kanan (anak kanan terdalam) dalam pohon.
14. Fungsi main() menginisialisasi pohon menggunakan init(), membuat root dengan data suatu data, menambahkan beberapa node anak kiri dan kanan dari root menggunakan insertLeft() dan insertRight(), traversal pohon dilakukan untuk menampilkan node dalam pre-order, in-order, dan post-order, node paling kiri dan kanan ditampilkan menggunakan mostLeft() dan mostRight(), dan menghapus node menggunakan deleteNode() dan traversal in-order.

```
// 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

A. Unguided 1

1. Mendefinisikan variabel pilihan untuk menyimpan pilihan menu dari pengguna, data untuk menyimpan data yang akan dimasukkan ke dalam pohon biner, parentData untuk menyimpan data node induk yang akan dipilih untuk ditambahkan anak kiri atau kanan, parentNode pointer yang digunakan untuk menyimpan node induk yang ditemukan berdasarkan parentData.
2. Mendefinisikan fungsi menu dengan menampilkan menu utama dengan berbagai pilihan operasi pada pohon biner, dan program akan mengeksekusi operasi tertentu pada pohon biner berdasarkan pilihan yang dimasukkan pengguna.

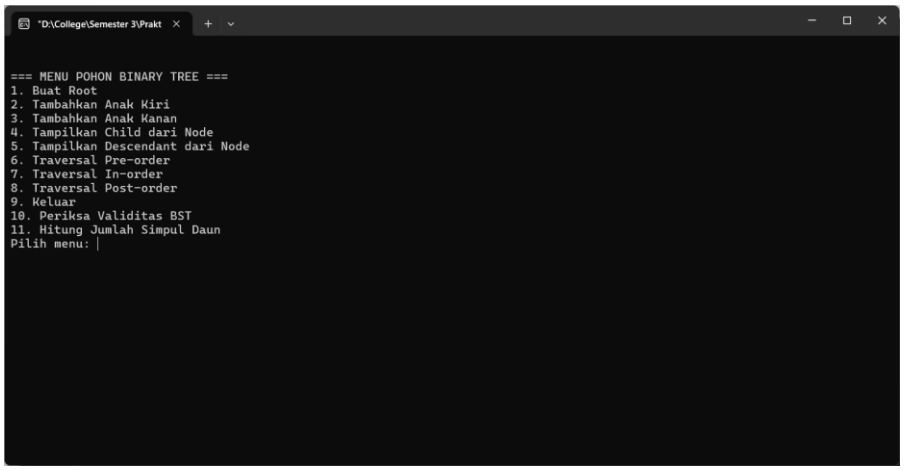
```
154  // Menu
155  void menu() {
156      int pilihan;
157      char data, parentData;
158      Pohon *parentNode;
159
160      do {
161          cout << "\n\n=== MENU POHON BINARY TREE ===\n";
162          cout << "1. Buat Root\n";
163          cout << "2. Tambahkan Anak Kiri\n";
164          cout << "3. Tambahkan Anak Kanan\n";
165          cout << "4. Tampilkan Child dari Node\n";
166          cout << "5. Tampilkan Descendant dari Node\n";
167          cout << "6. Traversal Pre-order\n";
168          cout << "7. Traversal In-order\n";
169          cout << "8. Traversal Post-order\n";
170          cout << "9. Keluar\n";
171          cout << "10. Periksa Validitas BST\n";
172          cout << "11. Hitung Jumlah Simpul Daun\n";
173          cout << "Pilih menu: ";
174          cin >> pilihan;
175
176          switch (pilihan) {
177              case 1: {
178                  cout << "\nMasukkan data root: ";
179                  cin >> data;
180                  buatNode(data);
181                  break;
182              }
```

3. Fungsi main() untuk menjalankan program dengan memanggil fungsi init() untuk menginisialisasi pohon kosong dan memanggil fungsi menu() untuk menjalankan menu interaktif dan memungkinkan pengguna melakukan operasi pada pohon biner.

```
184          case 2: {
185              cout << "\nMasukkan data parent: ";
186              cin >> parentData;
187              cout << "Masukkan data anak kiri: ";
188              cin >> data;
189
190              parentNode = root; // Mulai pencarian dari root
191              find(parentData, root);
192              if (parentNode)
193                  insertLeft(data, parentNode);
194              break;
195          }
196          case 3: {
197              cout << "\nMasukkan data parent: ";
198              cin >> parentData;
199              cout << "Masukkan data anak kanan: ";
200              cin >> data;
201
202              parentNode = root; // Mulai pencarian dari root
203              find(parentData, root);
204              if (parentNode)
205                  insertRight(data, parentNode);
206              break;
207          }
208          case 4: {
209              cout << "\nMasukkan data node: ";
210              cin >> data;
211
212              parentNode = root;
213              find(data, root);
214              tampilkanChild(parentNode);
215              break;
216          }
217          case 5: {
218              cout << "\nMasukkan data node: ";
219              cin >> data;
220
221              parentNode = root;
222              find(data, root);
223              cout << "\nDescendant dari node " << data << ":\n";
224              tampilkanDescendant(parentNode);
```

```
            break;
        }
        case 6: {
            cout << "\nTraversal Pre-order: ";
            preOrder(root);
            break;
        }
        case 7: {
            cout << "\nTraversal In-order: ";
            inOrder(root);
            break;
        }
        case 8: {
            cout << "\nTraversal Post-order: ";
            postOrder(root);
            break;
        }
        case 9: {
            cout << "\nKeluar dari program.\n";
            break;
        }
        case 10: {
            // Cek apakah pohon valid sebagai BST
            cout << "\nMemeriksa apakah pohon valid sebagai BST...\n";
            if (isValidBST(root))
                cout << "Pohon ini adalah Binary Search Tree (BST).\n";
            else
                cout << "Pohon ini BUKAN Binary Search Tree (BST).\n";
            break;
        }
        case 11: {
            // Hitung jumlah simpul daun
            int jumlahDaun = cari_simpul_daun(root);
            cout << "\nJumlah simpul daun dalam pohon: " << jumlahDaun << endl;
            break;
        }

        default:
            cout << "\nPilihan tidak valid.\n";
        }
    } while (pilihan != 9);
}
int main() {
    init();
    menu();
    return 0;
}
```

4. Berikut merupakan outputnya



B. Unguided 2

1. Fungsi tampilkanChild digunakan untuk menampilkan anak kiri dan kanan dari node tertentu pada pohon biner dengan memeriksa apakah node adalah null, jika node null maka menampilkan pesan bahwa node tidak ditemukan dan fungsi akan berhenti.
2. Jika node ditemukan maka menampilkan pesan anak dari node lalu memeriksa apakah node memiliki anak kiri, jika ada maka menampilkan anak kiri, jika tidak ada maka menampilkan anak kiri null, dan dilakukan pada anak kanan.
3. Fungsi tampilkanDescendant digunakan untuk menampilkan semua keturunan (descendant) dari node tertentu pada pohon biner dengan

memeriksa apakah node yang diberikan adalah null, jika node null maka fungsi akan berhenti.

4. Jika node ditemukan dan memiliki anak maka mencetak data dari node tersebut dan anak – anaknya, dengan mencetak node saat ini diikuti dengan anak kiri dan kanan (jika ada).
5. Setelah mencetak node dan keturunannya maka memanggil fungsi tampilkanDescendant secara rekursif untuk anak kiri dan anak kanan.

```
void tampilkanChild(Pohon *node) {
    if (!node) {
        cout << "\nNode tidak ditemukan!\n";
        return;
    }
    cout << "\nChild dari node " << node->data << ":\n";
    if (node->left)
        cout << "Anak Kiri: " << node->left->data << endl;
    else
        cout << "Anak Kiri: NULL\n";

    if (node->right)
        cout << "Anak Kanan: " << node->right->data << endl;
    else
        cout << "Anak Kanan: NULL\n";
}
void tampilkanDescendant(Pohon *node) {
    if (!node) return;

    if (node->left || node->right) {
        cout << node->data << " -> ";
        if (node->left) cout << node->left->data << " ";
        if (node->right) cout << node->right->data << " ";
        cout << endl;
    }
    tampilkanDescendant(node->left);
    tampilkanDescendant(node->right);
}
```

C. Unguided 3

1. Fungsi is_valid_bst digunakan untuk memeriksa apakah suatu pohon biner memenuhi properti dari Binary Search Tree (BST) dengan memeriksa apakah node kosong, jika tidak kosong maka fungsi memeriksa apakah nilai node berada di luar batas yang valid dan jika salah satu kondisi ini terpenuhi (nilai node tidak berada di antara batas yang valid), maka fungsi mengembalikan false karena pohon tidak memenuhi properti BST.
2. Fungsi kemudian memanggil dirinya sendiri secara rekursif untuk memeriksa subpohon kiri dan kanan dari node.
3. Fungsi isValidBST adalah pembungkus untuk memulai pemeriksaan pada root pohon.

4. Fungsi cari_simpul_daun digunakan untuk menghitung jumlah simpul daun dalam pohon biner dengan memeriksa apakah node kosong, jika tidak kosong maka fungsi memeriksa apakah node tersebut merupakan simpul daun dengan mengecek apakah kedua anak kiri dan kanan node adalah null.
5. Jika node memiliki anak, fungsi akan memanggil dirinya sendiri secara rekursif untuk menghitung jumlah simpul daun pada subpohon kiri dan kanan dan fungsi akan mengembalikan jumlah simpul daun dengan menjumlahkan hasil dari subpohon kiri dan kanan.

```
bool is_valid_bst(Pohon* node, char min_val, char max_val) {
    if (node == NULL) 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);
}
bool isValidBST(Pohon* root) {
    return is_valid_bst(root, CHAR_MIN, CHAR_MAX);
}
int cari_simpul_daun(Pohon* node) {
    if (node == NULL) return 0;
    if (node->left == NULL && node->right == NULL)
        return 1;
    return cari_simpul_daun(node->left) + cari_simpul_daun(node->right);
}
```

5. Kesimpulan

Dalam struktur data dan algoritma, rekursif, pohon biner (binary tree), dan traversal sangat penting. Dengan melakukan berbagai operasi pada pohon biner, seperti menambah anak kiri dan kanan, dan dengan menggunakan traversal seperti pre-order, in-order, dan post-order, kita dapat memahami cara menavigasi dan memanipulasi struktur pohon secara efisien. Selain itu, penggunaan Binary Search Tree (BST) menunjukkan bagaimana aturan tertentu diterapkan pada struktur pohon untuk memastikan bahwa elemen pada subpohon kiri lebih kecil dan elemen pada subpohon kanan lebih besar daripada nilai node saat ini. Keunggulan rekursif dalam menyelesaikan masalah yang melibatkan struktur data pohon ditunjukkan oleh fungsi rekursif yang digunakan dalam operasi seperti perhitungan simpul daun dan pemeriksaan validitas BST.