LAPORAN PRAKTIKUM

MODUL 10

TREE

Disusun Oleh :

Farhan Kurniawan (2311104073)

Kelas:

SE-07-2

Dosen :

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

PROGRAM STUDI S1 REKAYASA PERANGKAT LUNAK

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2024

I. 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

II. LANDASAN TEORI

Tree adalah salah satu struktur data non-linear yang merepresentasikan data dalam bentuk hierarki, di mana setiap elemen disebut sebagai **node**. Node terdiri atas satu node induk (parent) yang dapat memiliki satu atau lebih node anak (child). Struktur ini banyak digunakan karena kemampuannya untuk mengelola data secara efisien dalam berbagai aplikasi, seperti pengelolaan hierarki file, basis data, dan sistem jaringan. Salah satu jenis tree yang paling umum adalah **Binary Search Tree (BST)**, yang memiliki sifat bahwa nilai node di sub-pohon kiri lebih kecil, dan nilai di sub-pohon kanan lebih besar dari node induknya. Struktur ini memfasilitasi operasi pencarian, penyisipan, dan penghapusan dengan kompleksitas waktu rata-rata O(log⁡n)O(\log n)O(logn). Selain itu, varian tree seperti **Heap Tree**, **Segment Tree**, dan **Trie** sering digunakan untuk menyelesaikan masalah optimasi, pencarian, dan pengelolaan data string. Implementasi tree dalam pemrograman, khususnya dalam bahasa C++, melibatkan penggunaan pointer untuk menghubungkan node satu sama lain, menciptakan struktur dinamis yang dapat disesuaikan dengan kebutuhan.

III. GUIDED

1.

```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) {
    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
}

```

```cpp
// 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;
}
```

Hasil run:

```
PS C:\Users\Farhan Kurniawan> cd "d:\Praktukum\C++\Modul10\" ;

Node F berhasil dibuat menjadi root.

Node B berhasil ditambahkan ke child kiri F

Node G berhasil ditambahkan ke child kanan F

Node A berhasil ditambahkan ke child kiri B

Node D berhasil ditambahkan ke child kanan B

Node C berhasil ditambahkan ke child kiri D

Node E berhasil ditambahkan ke child kanan D

Pre-order Traversal: F B A D C E G
In-order Traversal: A B C D E F G
Post-order Traversal: A C E D B G F
Most Left Node: A
Most Right Node: G
Menghapus node D.
In-order Traversal setelah penghapusan: A B C E F G
PS D:\Praktukum\C++\Modul10>
```

IV. UNGUIDED

1. Modifikasi code tree diatas dengan program menu menggunakan input data tree dari user dan berikan fungsi tambahan untuk menampilkan node child dan descendant dari node yang diinputkan!
2. Buatlah fungsi rekursif is_valid_bst(node, min_val, max_val) untuk memeriksa apakah suatu pohon memenuhi properti Binary Search Tree. Uji fungsi ini pada berbagai pohon, baik yang valid maupun tidak valid sebagai BST.
3. Buatlah fungsi rekursif cari_simpul_daun(node) untuk menghitung jumlah simpul daun dalam Binary Tree. Simpul daun adalah node yang tidak memiliki anak kiri maupun kanan.

Kode Program:

```cpp
#include <iostream>
#include <limits.h>
using namespace std;

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

Pohon *root;

void init() {
    root = NULL;
}

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

void buatNode(char data) {
    if (isEmpty()) {
        root = new Pohon{data, NULL, NULL, NULL};
        cout << "\nNode " << data << " berhasil dibuat menjadi root." << endl;
    } else {
        cout << "\nPohon sudah dibuat." << endl;
    }
}

Pohon* insertLeft(char data, Pohon *node) {
    if (node->left != NULL) {
        cout << "\nNode " << node->data << " sudah ada child kiri!" << endl;
        return NULL;
    }
    Pohon *baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kiri " << node->data << endl;
    return baru;
}

Pohon* insertRight(char data, Pohon *node) {
    if (node->right != NULL) {
        cout << "\nNode " << node->data << " sudah ada child kanan!" << endl;
        return NULL;
    }
    Pohon *baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kanan " << node->data << endl;
    return baru;
}

void tampilChild(Pohon *node) {
    if (!node) {
        cout << "\nNode tidak ditemukan!" << endl;
        return;
    }
    cout << "\nNode " << node->data << " memiliki:\n";
    if (node->left) cout << "Child kiri: " << node->left->data << endl;
    else cout << "Tidak ada child kiri.\n";
    if (node->right) cout << "Child kanan: " << node->right->data << endl;
    else cout << "Tidak ada child kanan.\n";
}

void tampilDescendant(Pohon *node) {
    if (!node) return;
    if (node->left) cout << node->left->data << " ";
    if (node->right) cout << node->right->data << " ";
    tampilDescendant(node->left);
    tampilDescendant(node->right);
}

bool is_valid_bst(Pohon *node, int min_val, int max_val) {
    if (!node) 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) return 0;
    if (!node->left && !node->right) return 1;
    return cari_simpul_daun(node->left) + cari_simpul_daun(node->right);
}

```

```cpp
void menu() {
    int pilihan;
    char data, parent_data;
    Pohon *parent_node = NULL;

    do {
        cout << "\n\n=== MENU ===";
        cout << "\n1. Buat Root";
        cout << "\n2. Tambah Node Kiri";
        cout << "\n3. Tambah Node Kanan";
        cout << "\n4. Tampilkan Child";
        cout << "\n5. Tampilkan Descendant";
        cout << "\n6. Cek Valid BST";
        cout << "\n7. Hitung Jumlah Simpul Daun";
        cout << "\n8. Keluar";
        cout << "\nPilihan: ";
        cin >> pilihan;

        switch (pilihan) {
        case 1:
            cout << "Masukkan data root: ";
            cin >> data;
            buatNode(data);
            break;

        case 2:
            cout << "Masukkan data parent: ";
            cin >> parent_data;
            cout << "Masukkan data node baru: ";
            cin >> data;
            parent_node = root; // Implement pencarian node parent jika perlu
            insertLeft(data, parent_node);
            break;

        case 3:
            cout << "Masukkan data parent: ";
            cin >> parent_data;
            cout << "Masukkan data node baru: ";
            cin >> data;
            parent_node = root; // Implement pencarian node parent jika perlu
            insertRight(data, parent_node);
            break;

        case 4:
            cout << "Masukkan data node: ";
            cin >> data;
            tampilChild(root); // Implement pencarian node jika perlu
            break;

        case 5:
            cout << "Masukkan data node: ";
            cin >> data;
            cout << "Descendant: ";
            tampilDescendant(root); // Implement pencarian node jika perlu
            break;

        case 6:
            cout << (is_valid_bst(root, INT_MIN, INT_MAX) ? "Valid BST" : "Bukan BST") << endl;
            break;

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

        case 8:
            cout << "Keluar program." << endl;
            break;

        default:
            cout << "Pilihan tidak valid!" << endl;
        }
    } while (pilihan != 8);
}

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

V. KESIMPULAN

Mempelajari struktur data tree dalam C++ memberikan pemahaman mendalam tentang bagaimana data dapat diorganisasi secara hierarkis untuk mendukung berbagai aplikasi, seperti pencarian, pengurutan, dan pengelolaan data kompleks. Dengan menggunakan pointer, kita dapat membangun tree secara dinamis, memungkinkan fleksibilitas dalam pengelolaan node. Selain itu, berbagai operasi pada tree, seperti traversal (pre-order, in-order, post-order), penambahan, penghapusan, serta validasi sebagai Binary Search Tree (BST), memberikan wawasan penting dalam menyelesaikan masalah komputasi secara efisien. Dengan memahami implementasi tree dalam C++, pengembang dapat meningkatkan kemampuan pemrograman mereka, terutama dalam menangani struktur data non-linear yang sering digunakan dalam dunia nyata, seperti dalam basis data, sistem file, dan algoritma grafis.