LAPORAN PRAKTIKUM STRUKTUR DATA

PERTEMUAN 9

TREE

Nama :

Reyner Atira Prasetyo (2311104057)

S1SE-07-02

Dosen :

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

PROGRAM STUDI S1 REKAYASA PERANGKAT LUNAK

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2024

I. TUJUAN

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

TOOL

1. Visual Studio Code
2. GCC

II. DASAR TEORI

1. Definisi Tree

Tree adalah salah satu struktur data hierarkis yang terdiri dari **simpul (node)** yang dihubungkan dengan **edges (sisi)**. Tree sering digunakan untuk merepresentasikan hubungan hierarkis seperti file sistem, pohon keputusan, dan struktur organisasi.

2. Komponen Tree

- **Root (Akar):** Node utama yang menjadi titik awal tree. Tree hanya memiliki satu root.
- **Parent (Induk):** Node yang memiliki satu atau lebih node anak (children).
- **Child (Anak):** Node yang memiliki node induk.
- **Leaf (Daun):** Node yang tidak memiliki anak.
- **Subtree:** Pohon bagian dari tree yang lebih besar.
- **Level:** Tingkatan dalam tree yang dihitung dari akar (level 0).
- **Height (Tinggi):** Jarak maksimum dari akar ke leaf.

3. Karakteristik Tree

1. Tree memiliki satu node root.
2. Setiap node hanya memiliki satu parent.
3. Tidak ada siklus dalam struktur tree.
4. Tree dapat memiliki sejumlah anak (k-child).

4. Jenis-jenis Tree

1. **Binary Tree:**
    - Setiap node memiliki maksimal dua anak.
    - Terdiri dari **Left Child** dan **Right Child**.
2. **Binary Search Tree (BST):**

- Tree dengan aturan bahwa nilai di node kiri lebih kecil dari node root, dan nilai di node kanan lebih besar dari node root.

3. **Balanced Tree:**
   - Tree di mana perbedaan tinggi antara subtree kiri dan kanan tidak lebih dari satu.
4. **Complete Binary Tree:**
   - Semua level kecuali level terakhir penuh, dan semua node pada level terakhir berada di posisi paling kiri.
5. **Heap Tree:**
   - Tree khusus untuk memenuhi properti heap (Max-Heap atau Min-Heap).

5. Operasi pada Tree

1. **Traversal (Penelusuran):**
   - **Preorder (Root, Left, Right):** Penelusuran dimulai dari root, kemudian anak kiri, dan terakhir anak kanan.
   - **Inorder (Left, Root, Right):** Penelusuran dimulai dari anak kiri, root, dan anak kanan.
   - **Postorder (Left, Right, Root):** Penelusuran dimulai dari anak kiri, anak kanan, dan terakhir root.
   - **Level-order Traversal:** Penelusuran berdasarkan level dari atas ke bawah.
2. **Insertion (Penyisipan):**
   - Menambahkan node baru ke dalam tree berdasarkan aturan tree tertentu.
3. **Deletion (Penghapusan):**
   - Menghapus node dari tree, dengan menyesuaikan struktur agar tetap valid.
4. **Searching (Pencarian):**
   - Mencari node tertentu berdasarkan kriteria tertentu.

6. Penerapan Tree

- Struktur direktori file sistem.
- Pohon ekspresi untuk operasi matematika.
- Pohon keputusan dalam algoritma kecerdasan buatan.
- Representasi database (contoh: B-Tree untuk indeks database).

7. Keunggulan Tree

- Mendukung operasi pencarian yang efisien (terutama dalam BST).

- Struktur data fleksibel untuk berbagai kasus hierarki.
- Mempermudah implementasi algoritma rekursif.

8. Kelemahan Tree

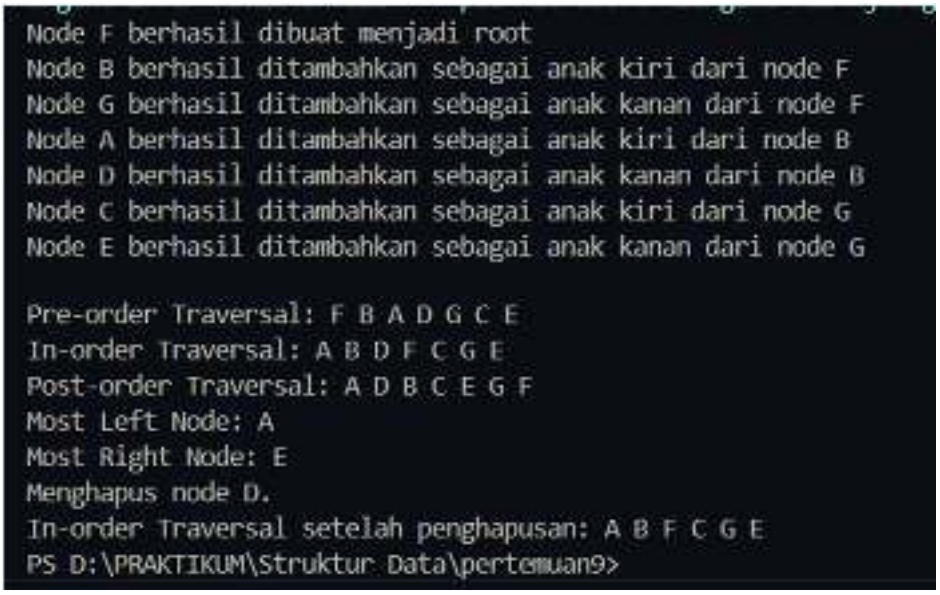
- Kompleksitas implementasi dibandingkan dengan struktur data linear.
- Bergantung pada keseimbangan tree untuk menjaga efisiensi operasinya.

III. GUIDED

1. guided.cpp

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

struct Pohon {
    char data;
    Pohon *kiri, *kanan;
    Pohon *parent;

    Pohon(char data, Pohon *kiri = NULL, Pohon *kanan = NULL, Pohon *parent = NULL)
        : data(data), kiri(kiri), kanan(kanan), parent(parent) {}
};

Pohon *root, *baru;

void init() {
    root = NULL;
}

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

void buatNode(char data){
    if (!isEmpty()) {
        root = new Pohon(data, NULL, NULL, NULL); // membuat node baru menggunakan konstruktor
        cout << "Node " << data << " berhasil dibuat menjadi root" << endl;
    } else {
        cout << "Pohon sudah ada" << endl;
    }
}

Pohon *insertLeft(char data, Pohon *node) {
    if (node->kiri != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kiri " << endl;
        return node->kiri;
    }
    baru = new Pohon(data, NULL, NULL, node);
    node->kiri = baru;
    cout << "Node " << data << " berhasil ditambahkan sebagai anak kiri dari node " << node->data << endl;
    return baru;
}

Pohon *insertRight(char data, Pohon *node) {
    if (node->kanan != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kanan" << endl;
        return node->kanan;
    }
    baru = new Pohon(data, NULL, NULL, node);
    node->kanan = baru;
    cout << "Node " << data << " berhasil ditambahkan sebagai anak kanan dari node " << node->data << endl;
    return baru;
}

void update (char data, Pohon *node) {
    if(!node) {
        cout << "\nNode tidak ditemukan" << endl;
        return;
    }
    char temp = node->data;
    node->data = data;
    cout << "\nNode " << temp << " berhasil diubah menjadi" << data << endl;
}

void find(char data, Pohon *node){
    if(!node) return;

    if (node->data == data) {
        cout << "\nNode ditemukan : "<< data << endl;
    }
    find(data, node->kiri);
    find(data, node->kanan);
}

void preOrder(Pohon *node) {
    if(!node) return;
    cout << node->data << " ";
    preOrder(node->kiri);
    preOrder(node->kanan);
}

void inOrder(Pohon *node) {
    if(!node) return;
    inOrder(node->kiri);
    cout << node->data << " ";
    inOrder(node->kanan);
}

void postOrder(Pohon *node) {
    if(!node) return;
    postOrder(node->kiri);
    postOrder(node->kanan);
    cout << node->data << " ";
}

Pohon* deleteNode(Pohon *node, char data) {
    if (!node) return NULL;

    if (data < node->data) {
        node->kiri = deleteNode(node->kiri, data);
    }
    else if (data > node->data) {
        node->kanan = deleteNode(node->kanan, data);
    }
    else {
        if (!node->kiri) {
            Pohon *temp = node->kiri;
            delete node;
            return temp;
        } else if (!node->kanan) {
            Pohon *temp = node->kiri;
            delete node;
            return temp;
        }

        Pohon *successor = node->kanan;
        while (successor->kiri) successor = successor->kiri;
        node->data = successor->data;
        node->kanan = deleteNode(node->kanan, successor->data);
    }
    return node;
}

Pohon* maxLeft(Pohon *node) {
    if(!node) return NULL;
    while(node->kiri) node = node->kiri;
    return node;
}

Pohon* maxRight(Pohon *node) {
    if(!node) return NULL;
    while(node->kanan) node = node->kanan;
    return node;
}

int main() {
    init();
    buatNode('F');
    insertLeft('B', root);
    insertRight('G', root);
    insertLeft('A', root->kiri);
    insertLeft('D', root->kanan);
    insertRight('I', root->kanan);
    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: " << maxLeft(root)->data;
    cout << "\nMost Right Node: " << maxRight(root)->data;

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

    return 0;
}
```

Hasil Run



IV. UNGUIDED

1. unguided.cpp

NO. 1

```cpp
#include <iostream>
#include <queue> // Untuk traversal level-order (opsional)
using namespace std;

struct Pohon {
    char data;
    Pohon *kiri, *kanan, *parent;

    Pohon(char data, Pohon *kiri = NULL, Pohon *kanan = NULL, Pohon *parent = NULL)
        : data(data), kiri(kiri), kanan(kanan), parent(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 << "Node " << data << " berhasil dibuat sebagai root." << endl;
    } else {
        cout << "Root sudah ada." << endl;
    }
}

Pohon *insertLeft(char data, Pohon *node) {
    if (!node) {
        cout << "Node parent tidak ditemukan." << endl;
        return NULL;
    }
    if (node->kiri != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kiri." << endl;
        return node->kiri;
    }
    Pohon *baru = new Pohon(data, NULL, NULL, node);
    node->kiri = baru;
    cout << "Node " << data << " berhasil ditambahkan sebagai anak kiri dari " << node->data << "." << endl;
    return baru;
}

Pohon *insertRight(char data, Pohon *node) {
    if (!node) {
        cout << "Node parent tidak ditemukan." << endl;
        return NULL;
    }
    if (node->kanan != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kanan." << endl;
        return node->kanan;
    }
    Pohon *baru = new Pohon(data, NULL, NULL, node);
    node->kanan = baru;
    cout << "Node " << data << " berhasil ditambahkan sebagai anak kanan dari " << node->data << "." << endl;
    return baru;
}

void preOrder(Pohon *node) {
    if (!node) return;
    cout << node->data << " ";
    preOrder(node->kiri);
    preOrder(node->kanan);
}

void inOrder(Pohon *node) {
    if (!node) return;
    inOrder(node->kiri);
    cout << node->data << " ";
    inOrder(node->kanan);
}

void postOrder(Pohon *node) {
    if (!node) return;
    postOrder(node->kiri);
    postOrder(node->kanan);
    cout << node->data << " ";
}

void findNode(char data, Pohon *node, Pohon *&result) {
    if (!node) return;
    if (node->data == data) {
        result = node;
        return;
    }
    findNode(data, node->kiri, result);
    findNode(data, node->kanan, result);
}
```

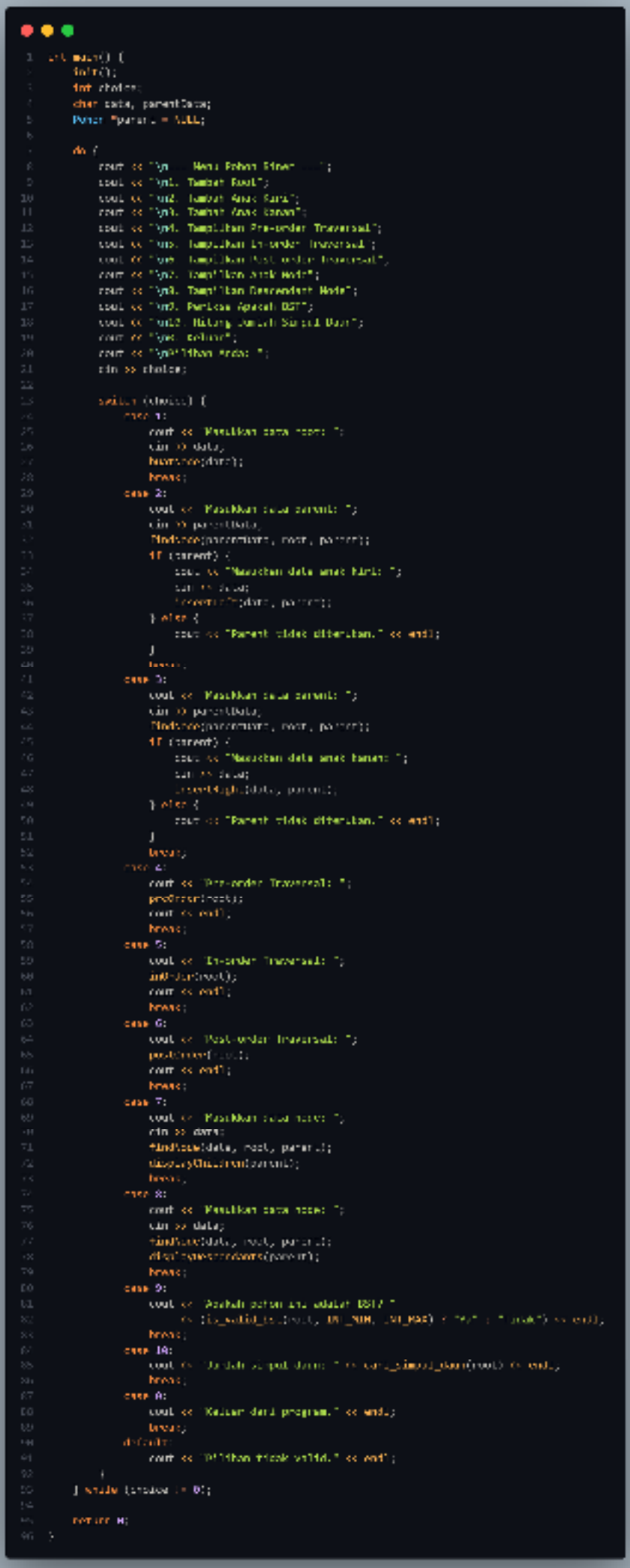
NO. 2

```cpp
void displayChildren(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan." << endl;
        return;
    }
    cout << "Node " << node->data << " memiliki anak: ";
    if (node->kiri) cout << "Kiri: " << node->kiri->data << " ";
    if (node->kanan) cout << "Kanan: " << node->kanan->data << " ";
    if (!node->kiri && !node->kanan) cout << "tidak ada.";
    cout << endl;
}

void displayDescendants(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan." << endl;
        return;
    }
    cout << "Descendant dari node " << node->data << ": ";
    queue<Pohon *> q;
    q.push(node);
    while (!q.empty()) {
        Pohon *current = q.front();
        q.pop();
        if (current->kiri) {
            cout << current->kiri->data << " ";
            q.push(current->kiri);
        }
        if (current->kanan) {
            cout << current->kanan->data << " ";
            q.push(current->kanan);
        }
    }
    cout << endl;
}
```

NO. 3

```c
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->kiri, min_val, node->data) &&
           is_valid_bst(node->kanan, node->data, max_val);
}

// Fungsi menghitung jumlah simpul daun
int cari_simpul_daun(Pohon *node) {
    if (!node) return 0;

    if (!node->kiri && !node->kanan) return 1;

    return cari_simpul_daun(node->kiri) + cari_simpul_daun(node->kanan);
}
```

MAIN FUNCTION



Hasil Run :

```
=== 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: F
Node F 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: F
Masukkan data anak kiri: B
Node B berhasil ditambahkan sebagai anak kiri dari F.
```

```
=== 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: F
Masukkan data anak kanan: G
Node G berhasil ditambahkan sebagai anak kanan dari F.

=== 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: B
Masukkan data anak kiri: A
Node A berhasil ditambahkan sebagai anak kiri dari B.
```

```
=== 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: B
Masukkan data anak kanan: D
Node D berhasil ditambahkan sebagai anak kanan dari B.

=== 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: G
Masukkan data anak kiri: C
Node C berhasil ditambahkan sebagai anak kiri dari G.
```

```
=== 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: G
Masukkan data anak kanan: E
Node E berhasil ditambahkan sebagai anak kanan dari G.

=== 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: F B A D G C E
```

```
=== 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: A B D F C G E

=== 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: A D B C E G F
```

```
=== 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: F
Node F memiliki anak: Kiri: B Kanan: G
```

```
=== 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: F
Descendant dari node F: B G A D C E

=== 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: 4
```

V. KESIMPULAN

Pada praktikum ini, kami mempelajari implementasi struktur data Tree menggunakan bahasa C++. Tree adalah struktur data hierarkis yang terdiri dari kumpulan node, di mana setiap node dapat memiliki sejumlah anak, dan hanya satu node yang bertindak sebagai root. Praktikum ini berfokus pada implementasi Binary Tree, yaitu tree di mana setiap node memiliki maksimal dua anak.

Langkah pertama adalah mendefinisikan struktur data Tree yang terdiri dari node

dengan atribut nilai data, pointer ke anak kiri (left child), dan pointer ke anak kanan (right child). Praktikum ini melibatkan beberapa operasi dasar pada tree, yaitu:

- Insert (menambahkan elemen ke tree),
- Traversal (penelusuran elemen dalam tree),
- Search (mencari elemen tertentu), dan
- Delete (menghapus elemen dari tree).

Implementasi dilakukan dengan dua cara utama:

1. Binary Tree menggunakan array: Elemen-elemen tree diindeks dalam array, di mana anak kiri berada di indeks 2*i + 1 dan anak kanan di indeks 2*i + 2. Cara ini mempermudah akses elemen, tetapi memiliki kelemahan dalam efisiensi memori, terutama jika tree tidak penuh.
2. Binary Tree menggunakan linked list: Setiap node diwakili oleh objek yang memiliki pointer ke anak kiri dan anak kanan. Cara ini lebih fleksibel dalam menangani tree yang dinamis, tetapi memerlukan lebih banyak memori untuk menyimpan pointer dan lebih kompleks dalam implementasi.

Hasil praktikum menunjukkan bahwa implementasi Tree menggunakan array lebih sederhana dan cepat untuk akses elemen jika tree bersifat lengkap, tetapi tidak efisien untuk tree yang tidak penuh. Di sisi lain, implementasi Tree menggunakan linked list lebih fleksibel untuk menambah atau menghapus elemen, tetapi memiliki overhead memori yang lebih besar dan membutuhkan pengelolaan pointer yang hati-hati.

Secara keseluruhan, praktikum ini memberikan pemahaman yang lebih baik tentang cara kerja struktur data Tree, serta kelebihan dan kekurangan implementasi menggunakan array dan linked list dalam bahasa C++.