LAPORAN PRAKTIKUM

MODUL 10 & 11

“TREE”

Disusun Oleh:

Dhiemas Tulus Ikhsan 2311104046

SE-07-02

Dosen :

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY

PURWOKERTO

2024

I. TUJUAN

a. Mamahami konsep penggunaan fungsi rekursif.

b. Mengimplementasikan bentuk-bentuk fungsi rekursif.

c. Mengaplikasikan struktur data tree dalam sebuah kasus pemrograman.

d. Mengimplementasikan struktur data tree, khusunya Binary Tree.

II. LANDASAN TEORI

Rekursi merupakan metode pemrograman yang memanfaatkan fungsi untuk memanggil dirinya sendiri selama kondisi tertentu terpenuhi. Pendekatan ini memberikan beberapa keunggulan, seperti meningkatkan keterbacaan kode (readability), modularitas (modularity), dan penggunaan ulang kode (reusability). Rekursi sangat berguna untuk menyelesaikan masalah dengan pola penyelesaian yang terstruktur, seperti menghitung faktorial, pangkat, atau traversal struktur data tertentu. Namun, kekurangan utama rekursi adalah kebutuhan memori yang lebih besar untuk menyimpan activation record serta waktu eksekusi yang lebih lama dibandingkan metode iteratif, terutama jika tidak dirancang dengan efisiensi yang memadai.

Tree adalah salah satu struktur data non-linear yang terdiri dari node-node yang saling terhubung secara hierarkis. Tree memiliki beberapa karakteristik penting, seperti root (simpul pertama tanpa pendahulu), child dan parent (hubungan langsung antara simpul anak dan simpul induk), degree (jumlah anak yang dimiliki sebuah simpul), leaf (simpul tanpa anak), serta height atau depth (tinggi maksimum tree yang dihitung dari root hingga simpul leaf terdalam). Tree sering digunakan dalam representasi data hierarkis, traversal, pencarian, dan pemetaan hubungan dalam berbagai aplikasi.

Binary Tree adalah bentuk khusus dari tree di mana setiap simpul memiliki maksimum dua anak. Beberapa variasi binary tree yang sering digunakan mencakup Complete Binary Tree, yang mengisi semua level hingga penuh kecuali level terakhir; Binary Search Tree (BST), yang mengatur anak kiri lebih kecil dan anak kanan lebih besar dari induknya; AVL Tree, yang menjaga perbedaan tinggi maksimum antara subtree kiri dan kanan tidak lebih dari satu; serta Heap Tree, yang memiliki aturan bahwa nilai pada simpul induk selalu lebih kecil (min-heap) atau lebih besar (max-heap) dari nilai anak-anaknya.

Operasi dasar pada Binary Search Tree meliputi penyisipan (insert), pencarian (search), dan traversal. Penyisipan dilakukan dengan menambahkan simpul baru ke subtree kiri atau kanan sesuai aturan BST, sedangkan pencarian menggunakan algoritma binary search untuk menemukan nilai tertentu dalam tree. Traversal digunakan untuk menjelajahi semua simpul dalam tree dengan pola tertentu, seperti pre-order (mengunjungi root terlebih dahulu, diikuti subtree kiri dan kanan), in-order (mengunjungi subtree kiri, root, dan subtree kanan), serta post-order (mengunjungi subtree kiri, subtree kanan, dan terakhir root). Operasi-operasi ini memungkinkan pengelolaan data dalam tree menjadi lebih terstruktur dan efisien.

III. GUIDED

1. guided_1

```
#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;
}
```

Output:

```
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
```

IV. UNGUIDED

1. Task

```
#include <iostream>
#include <limits>
#include <climits> // Untuk batas minimum dan maksimum karakter
using namespace std;

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

Pohon *root, *baru;

// Fungsi inisialisasi
void init() {
    root = NULL;
}

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

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

// Menambahkan node ke anak kiri
Pohon* insertLeft(char data, Pohon *node) {
    if (!node) return NULL;
```

```
    if (node->left != NULL) {
        cout << "\nNode " << node->data << " sudah memiliki child kiri.\n";
        return NULL;
    }
    baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    cout << "\nNode " << data << " berhasil ditambahkan ke child kiri " << node->data
<< ".\n";
    return baru;
}

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

// Menemukan node berdasarkan data
Pohon* findNode(Pohon *node, char data) {
    if (!node) return NULL;
    if (node->data == data) return node;
    Pohon *leftResult = findNode(node->left, data);
    if (leftResult) return leftResult;
    return findNode(node->right, data);
}

// Menampilkan child dari node
void tampilChild(Pohon *node) {
    if (!node) {
        cout << "\nNode tidak ditemukan.\n";
        return;
    }
    cout << "\nChild dari node " << node->data << ":\n";
    if (node->left) cout << "Kiri: " << node->left->data << "\n";
    else cout << "Kiri: NULL\n";
    if (node->right) cout << "Kanan: " << node->right->data << "\n";
    else cout << "Kanan: NULL\n";
}

// Menampilkan descendant dari node
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);
}

// Traversal In-order
void inOrder(Pohon *node) {
    if (!node) return;
    inOrder(node->left);
    cout << node->data << " ";
    inOrder(node->right);
}

// Validasi apakah pohon merupakan BST
bool is_valid_bst(Pohon *node, char min_val, char 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);
}

// Menghitung jumlah simpul daun
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);
}

// Fungsi Menu
void menu() {
    int pilihan;
    char data, parentData;
    Pohon *node;

    do {
        cout << "\n--- Menu ---\n";
        cout << "1. Buat Root\n";
        cout << "2. Tambah Anak Kiri\n";
        cout << "3. Tambah Anak Kanan\n";
        cout << "4. Tampilkan Child\n";
        cout << "5. Tampilkan Descendant\n";
        cout << "6. In-Order Traversal\n";
        cout << "7. Validasi BST\n";
        cout << "8. Hitung Simpul Daun\n";
        cout << "0. Keluar\n";
        cout << "Pilihan: ";
        cin >> pilihan;

        switch (pilihan) {
        case 1:
            cout << "Masukkan data root: ";
            cin >> data;
            buatNode(data);
            break;
        case 2:
            cout << "Masukkan data parent: ";
            cin >> parentData;
            cout << "Masukkan data anak kiri: ";
            cin >> data;
            node = findNode(root, parentData);
            insertLeft(data, node);
            break;
        case 3:
            cout << "Masukkan data parent: ";
            cin >> parentData;
            cout << "Masukkan data anak kanan: ";
            cin >> data;
            node = findNode(root, parentData);
            insertRight(data, node);
            break;
        case 4:
            cout << "Masukkan data node: ";
            cin >> data;
            node = findNode(root, data);
            tampilChild(node);
            break;
        case 5:
            cout << "Masukkan data node: ";
            cin >> data;
            node = findNode(root, data);
            cout << "\nDescendant dari node " << data << ": ";
            tampilDescendant(node);
            cout << "\n";
```

```
            break;
        case 6:
            cout << "\nIn-Order Traversal: ";
            inOrder(root);
            cout << "\n";
            break;
        case 7:
            cout << "Apakah pohon valid BST? "
                 << (is_valid_bst(root, CHAR_MIN, CHAR_MAX) ? "Ya" : "Tidak") << "\n";
            break;
        case 8:
            cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << "\n";
            break;
        case 0:
            cout << "Keluar dari program.\n";
            break;
        default:
            cout << "Pilihan tidak valid.\n";
        }
    } while (pilihan != 0);
}

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

Output:

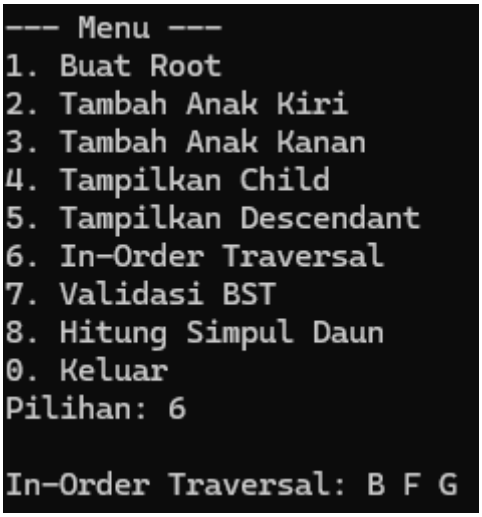
```
--- Menu ---
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. In-Order Traversal
7. Validasi BST
8. Hitung Simpul Daun
0. Keluar
Pilihan: 1
Masukkan data root: F

Node F berhasil dibuat menjadi root.
```

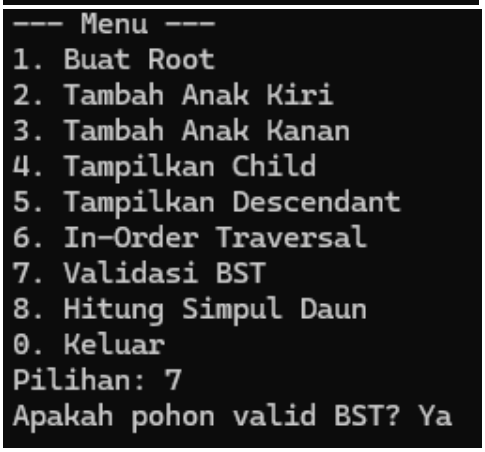
```
--- Menu ---
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. In-Order Traversal
7. Validasi BST
8. Hitung Simpul Daun
0. Keluar
Pilihan: 2
Masukkan data parent: F
Masukkan data anak kiri: B

Node B berhasil ditambahkan ke child kiri F.
```

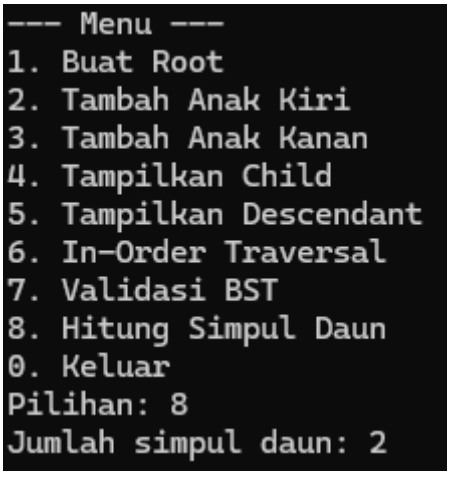
```
--- Menu ---
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. In-Order Traversal
7. Validasi BST
8. Hitung Simpul Daun
0. Keluar
Pilihan: 3
Masukkan data parent: F
Masukkan data anak kanan: G

Node G berhasil ditambahkan ke child kanan F.
```

```
--- Menu ---
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. In-Order Traversal
7. Validasi BST
8. Hitung Simpul Daun
0. Keluar
Pilihan: 4
Masukkan data node: F

Child dari node F:
Kiri: B
Kanan: G
```

```
--- Menu ---
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. In-Order Traversal
7. Validasi BST
8. Hitung Simpul Daun
0. Keluar
Pilihan: 5
Masukkan data node: F

Descendant dari node F: B G
```







V. KESIMPULAN

Kesimpulan dari praktikum rekursi merupakan metode yang efektif untuk menyelesaikan masalah dengan pola langkah yang terstruktur, seperti traversal dan penghitungan nilai matematis. Struktur data pohon, khususnya Binary Tree dan variasinya seperti Binary Search Tree (BST), terbukti menjadi alat yang sangat berguna untuk menyimpan dan mengelola data secara hierarkis. Melalui penerapan operasi dasar seperti pembuatan node, penambahan anak kiri atau kanan, penghapusan node, dan traversal menggunakan metode pre-order, in-order, serta post-order, data dapat diorganisir dengan

efisien. Program ini juga dilengkapi dengan menu interaktif yang memungkinkan pengguna memasukkan data secara dinamis, menampilkan anak (child) dan keturunan (descendant) dari node tertentu, serta menganalisis struktur pohon. Selain itu, fungsi validasi BST menggunakan metode rekursif berhasil diterapkan untuk memastikan pohon memenuhi aturan BST, sementara perhitungan jumlah simpul daun (leaf nodes) menggunakan rekursi juga berhasil diterapkan. Meskipun rekursi memiliki keterbatasan dalam hal efisiensi memori dan waktu eksekusi, metode ini memberikan solusi yang lebih sederhana dan logis untuk permasalahan tertentu. Praktikum ini memberikan pemahaman yang lebih mendalam tentang implementasi pohon biner secara teknis serta penggunaan rekursi dalam manipulasi data dan analisis struktur hierarkis seperti pohon biner.