LAPORAN PRAKTIKUM

Modul 09

“Tree”

Disusun Oleh:

Marvel Sanjaya Setiawan (2311104053)

SE-07-02

Dosen :

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY

PURWOKERTO

2024

1. **Tujuan**
    - Konsep fungsi rekursif.
    - Implementasi fungsi rekursif.
    - Aplikasi struktur data tree.
    - Implementasi Binary Tree.
2. **Landasan Teori**

Rekursif

Rekursif: Fungsi yang memanggil dirinya sendiri, berguna untuk menyelesaikan masalah dengan pola langkah teratur. Contoh: pangkat dua, faktorial.

Tree

Tree: Struktur data non-linear dengan node sebagai elemen utama. Node memiliki root, parent, child, sibling, leaf, dan internal node.

Jenis-jenis Tree:

1. Ordered Tree: Urutan anak penting.
2. Binary Tree: Setiap node maksimal memiliki 2 child.
    - Complete Binary Tree: Semua level penuh kecuali terakhir.
    - Binary Search Tree (BST): Anak kiri < parent < anak kanan.
    - AVL Tree: Selisih tinggi subtree ≤ 1.
    - Heap Tree: Parent lebih kecil/lebih besar dari kedua child.

Operasi pada Binary Search Tree:

1. Insert: Tambahkan node ke posisi sesuai aturan BST.
2. Update: Perbarui node dan lakukan regenerasi jika melanggar aturan BST.
3. Search: Cari elemen dengan algoritma binary search.

Tranversal Binary Tree:

1. Pre-order: Root → Subtree kiri → Subtree kanan.
2. In-order: Subtree kiri → Root → Subtree kanan.
3. Post-order: Subtree kiri → Subtree kanan → Root.

Telkom University Purwokerto

3. Guided

```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;
    }
    // Melanjutkan 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; // Ganti nilai 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('D', root); // Menambahkan 'D' ke anak kiri root
    insertRight('G', root); // Menambahkan 'G' ke anak kanan root
    insertLeft('A', root->left); // Menambahkan 'A' ke anak kiri dari 'D'
    insertRight('E', root->left); // Menambahkan 'E' ke anak kanan dari 'D'
    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 << "\n\nMenghapus node D.";
    root = deleteNode(root, 'D');
    cout << "\nIn-order Traversal setelah penghapusan: ";
    inOrder(root);

    return 0;
}
```

```
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
```

Cara Kerja:

1. Struktur Data: Membuat struktur Pohon dengan data, pointer anak kiri, kanan, dan induk.
2. Inisialisasi: Mengatur root sebagai NULL untuk pohon kosong.
3. Periksa Kosong: Fungsi isEmpty() mengembalikan true jika root adalah NULL.
4. Buat Root: Fungsi buatNode membuat node baru sebagai root jika pohon kosong.
5. Tambah Anak Kiri/Kanan: Fungsi insertLeft dan insertRight menambah node baru sebagai anak kiri/kanan dari node tertentu jika anak tersebut belum ada.
6. Ubah Data Node: Fungsi update mengganti data dalam node dengan nilai baru.
7. Cari Node: Fungsi find mencari node dengan data tertentu secara rekursif.
8. Traversal Pre-order: Fungsi preOrder mengunjungi node saat ini, anak kiri, dan anak kanan.
9. Traversal In-order: Fungsi inOrder mengunjungi anak kiri, node saat ini, dan anak kanan.
10. Traversal Post-order: Fungsi postOrder mengunjungi anak kiri, anak kanan, dan node saat ini.

11. Hapus Node: Fungsi deleteNode menghapus node dengan data tertentu dan mengatur penggantinya.

12. Node Paling Kiri/Kanan: Fungsi mostLeft dan mostRight menemukan node paling kiri/kanan di pohon.

13. Fungsi Utama: Inisialisasi pohon, membuat node root dan anak-anaknya, melakukan traversal, menampilkan node kiri/kanan, dan menghapus node tertentu.

4. Unguided

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

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

// Variabel global untuk menyimpan root dan node baru
Pohon *root, *baru;

// Inisialisasi pohon agar kosong
void init() {
    root = NULL;
}

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

// Membuat node baru sebagai root pohon
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;
    }
}

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

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

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

// Mencari node dengan data tertentu
Pohon* find(char data, Pohon *node) {
    if (!node) return NULL;

    if (node->data == data) {
        cout << "\nNode ditemukan: " << data << endl;
        return node;
    }
    Pohon* foundNode = find(data, node->left);
    if (foundNode == NULL) foundNode = find(data, node->right);
    return foundNode;
}

// Traversal Pre-order (Node -> Kiri -> Kanan)
void preOrder(Pohon *node) {
    if (!node) return;
    cout << node->data << " ";
    preOrder(node->left);
    preOrder(node->right);
}

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

// Traversal Post-order (Kiri -> Kanan -> Node)
void postOrder(Pohon *node) {
    if (!node) return;
    postOrder(node->left);
    postOrder(node->right);
    cout << node->data << " ";
}

// Menampilkan child dan descendant dari node tertentu
void showChildDescendant(Pohon *node) {
    if (!node) return;
    cout << "Child dari node " << node->data << ": ";
    if (node->left) cout << node->left->data << " ";
    if (node->right) cout << node->right->data << " ";
    cout << endl << "Descendant dari node " << node->data << ": ";
    preOrder(node);
    cout << endl;
}

// Fungsi untuk memeriksa apakah pohon memenuhi properti Binary Search Tree (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);
}

// Fungsi rekursif untuk 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 utama dengan menu
int main() {
    init();
    char pilihan;
    char data, parentData;
    Pohon *parentNode;

    do {
        cout << "\nMenu Program Binary Tree:";
        cout << "\n1. Buat Node Root";
        cout << "\n2. Tambah Anak Kiri";
        cout << "\n3. Tambah Anak Kanan";
        cout << "\n4. Tampilkan Child";
        cout << "\n5. Tampilkan Descendant";
        cout << "\n6. Cek Valid BST";
        cout << "\n7. Hitung Jumlah Simpul Daun";
        cout << "\n8. Keluar";
        cout << "\nPilih: ";
        cin >> pilihan;

        switch (pilihan) {
            case '1':
                cout << "Masukkan data untuk node root: ";
                cin >> data;
                buatNode(data);
                break;
            case '2':
                cout << "Masukkan data untuk node baru: ";
                cin >> data;
                cout << "Masukkan data node induk: ";
                cin >> parentData;
                parentNode = find(parentData, root);
                if (parentNode) insertLeft(data, parentNode);
                break;
            case '3':
                cout << "Masukkan data untuk node baru: ";
                cin >> data;
                cout << "Masukkan data node induk: ";
                cin >> parentData;
                parentNode = find(parentData, root);
                if (parentNode) insertRight(data, parentNode);
                break;
            case '4':
                cout << "Masukkan data node: ";
                cin >> data;
                parentNode = find(data, root);
                if (parentNode) {
                    showChildDescendant(parentNode);
                } else {
                    cout << "Node tidak ditemukan." << endl;
                }
                break;
            case '5':
                cout << "Masukkan data node: ";
                cin >> data;
                parentNode = find(data, root);
                if (parentNode) {
                    showChildDescendant(parentNode);
                } else {
                    cout << "Node tidak ditemukan." << endl;
                }
                break;
            case '6':
                if (is_valid_bst(root, '0', 'z')) {
                    cout << "Pohon adalah Binary Search Tree yang valid." << endl;
                } else {
                    cout << "Pohon bukan Binary Search Tree yang valid." << endl;
                }
                break;
            case '7':
                cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << endl;
                break;
            case '8':
                cout << "Keluar." << endl;
                break;
            default:
                cout << "Pilihan tidak valid!" << endl;
        }
    } while (pilihan != '8');

    return 0;
}
```

Telkom University Purwokerto

```
Menu Program Binary Tree:
1. Buat Node Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 1
Masukkan data untuk node root: 1

Node 1 berhasil dibuat menjadi root.

Menu Program Binary Tree:
1. Buat Node Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 2
Masukkan data untuk node baru: 3
Masukkan data pada induk: 2
Menu Program Binary Tree:
1. Buat Node Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 6
Pohon adalah Binary Search Tree yang valid.
```

```
Menu Program Binary Tree:
1. Buat Node Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Cek Valid BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 7
Jumlah simpul daun: 1
```

Cara Kerja:

1. Struktur Data: Struct Pohon menyimpan data node, pointer anak kiri, kanan, dan induk.
2. Inisialisasi: Fungsi init mengatur root sebagai NULL.
3. Periksa Kosong: Fungsi isEmpty mengembalikan true jika root adalah NULL.
4. Buat Node Root: Fungsi buatNode membuat node baru sebagai root jika pohon kosong.
5. Tambah Anak Kiri/Kanan: Fungsi insertLeft dan insertRight menambah node baru sebagai anak kiri/kanan dari node tertentu.
6. Ubah Data Node: Fungsi update mengganti data dalam node.
7. Cari Node: Fungsi find mencari node dengan data tertentu secara rekursif.
8. Traversal:
    - preOrder: Mengunjungi node saat ini, lalu anak kiri, dan anak kanan.
    - inOrder: Mengunjungi anak kiri, node saat ini, lalu anak kanan.
    - postOrder: Mengunjungi anak kiri, anak kanan, lalu node saat ini.
9. Tampilkan Child dan Descendant: Fungsi showChildDescendant menampilkan anak kiri/kanan dan seluruh descendant dari node tertentu.
10. Cek Valid BST: Fungsi is_valid_bst memeriksa apakah pohon memenuhi properti Binary Search Tree.
11. Hitung Simpul Daun: Fungsi cari_simpul_daun menghitung jumlah simpul daun dalam pohon.
12. Fungsi Utama (Main): Menu interaktif untuk mengelola pohon biner: membuat node root, menambah anak kiri/kanan, menampilkan child dan descendant, memeriksa validitas BST, menghitung simpul daun, dan keluar dari program.

5. Kesimpulan

Fungsi rekursif dan struktur data pohon, terutama Binary Tree, memungkinkan efisiennya pembuatan, penambahan, pengubahan, pencarian, dan penghapusan node serta traversal data yang terstruktur. Pemahaman ini fundamental untuk efisiensi pemrograman dan pengelolaan data.