LAPORAN PRAKTIKUM

MODUL 9

Tree

Disusun Oleh:

Muhammad Ikhsan Al Hakim (2311104064)

S1SE-07-02

Dosen :

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING

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

2.1 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.

2.2 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).

III. GUIDED

1. Guided

Code:

```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
}

// 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('A'); // Membuat root dengan data 'A'
    insertLeft('B', root); // Menambahkan 'B' ke anak kiri root
    insertRight('C', root); // Menambahkan 'C' 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

Code:

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

Node* createNode(int value) {
    return new Node(value);
}

void addLeft(Node* parent, int value) {
    if (parent->left == nullptr) {
        parent->left = new Node(value);
        cout << "Node " << value << " berhasil ditambahkan sebagai anak kiri." << endl;
    } else {
        cout << "Anak kiri sudah ada!" << endl;
    }
}

void addRight(Node* parent, int value) {
    if (parent->right == nullptr) {
        parent->right = new Node(value);
        cout << "Node " << value << " berhasil ditambahkan sebagai anak kanan." << endl;
    } else {
        cout << "Anak kanan sudah ada!" << endl;
    }
}

void displayChildren(Node* node) {
    if (node == nullptr) {
        cout << "Node tidak ada." << endl;
        return;
    }
    cout << "Child dari node " << node->data << ": ";
    if (node->left) cout << "Kiri: " << node->left->data << " ";
    if (node->right) cout << "Kanan: " << node->right->data << " ";
    if (!node->left && !node->right) cout << "Tidak ada child.";
    cout << endl;
}

void displayDescendants(Node* node) {
    if (node == nullptr) {
        cout << "Node tidak ada." << endl;
        return;
    }
    queue<Node*> q;
    q.push(node);
    cout << "Descendant dari node " << node->data << ": ";
    bool hasDescendant = false;
    while (!q.empty()) {
        Node* current = q.front();
        q.pop();
        if (current->left) {
            cout << current->left->data << " ";
            q.push(current->left);
            hasDescendant = true;
        }
        if (current->right) {
            cout << current->right->data << " ";
            q.push(current->right);
            hasDescendant = true;
        }
    }
    if (!hasDescendant) cout << "Tidak ada descendant.";
    cout << endl;
}

bool isBST(Node* node, int min = INT_MIN, int max = INT_MAX) {
    if (node == nullptr) return true;
    if (node->data <= min || node->data >= max) return false;
    return isBST(node->left, min, node->data) && isBST(node->right, node->data, max);
}

int countLeafNodes(Node* node) {
    if (node == nullptr) return 0;
    if (node->left == nullptr && node->right == nullptr) return 1;
    return countLeafNodes(node->left) + countLeafNodes(node->right);
}

void deleteTree(Node* node) {
    if (node == nullptr) return;
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
}

int main() {
    Node* root = nullptr;
    int choice, value;

    do {
        cout << "\nMenu Program Binary Tree:\n";
        cout << "1. Buat Node 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. Cek Valid Bst\n";
        cout << "7. Hitung Jumlah Simpul Daun\n";
        cout << "8. Keluar\n";
        cout << "Pilih: ";
        cin >> choice;

        switch (choice) {
            case 1:
                if (root != nullptr) {
                    cout << "Root sudah dibuat!" << endl;
                } else {
                    cout << "Masukkan data untuk node root: ";
                    cin >> value;
 root = createNode(value);
                    cout << "Node " << value << " berhasil dibuat menjadi root." << endl;
                }
                break;

            case 2:
                if (root == nullptr) {
                    cout << "Buat root terlebih dahulu!" << endl;
                } else {
                    cout << "Masukkan data untuk node baru: ";
                    cin >> value;
                    addLeft(root, value);
                }
                break;

            case 3:
                if (root == nullptr) {
                    cout << "Buat root terlebih dahulu!" << endl;
                } else {
                    cout << "Masukkan data untuk node baru: ";
                    cin >> value;
                    addRight(root, value);
                }
                break;

            case 4:
                if (root == nullptr) {
                    cout << "Buat root terlebih dahulu!" << endl;
                } else {
                    displayChildren(root);
                }
                break;

            case 5:
                if (root == nullptr) {
                    cout << "Buat root terlebih dahulu!" << endl;
                } else {
                    displayDescendants(root);
                }
                break;

            case 6:
                if (isBST(root)) {
                    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: " << countLeafNodes(root) << endl;
                break;

            case 8:
                cout << "Keluar dari program." << endl;
                deleteTree(root);
                break;

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

    return 0;
}
```

Output:

```
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: 20
Node 20 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: 7
Node 7 berhasil ditambahkan sebagai anak kiri.
```

```
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: 3
Masukkan data untuk node baru: 8
Node 8 berhasil ditambahkan sebagai anak kanan.

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: 2
```

Penjelasan:

Pemahaman Kode secara Umum

1. Struktur Node:

Setiap node dalam pohon diwakili oleh struktur Node.

Struktur ini memiliki tiga anggota: data untuk menyimpan nilai pada node, left untuk menunjuk ke anak kiri, dan right untuk menunjuk ke anak kanan.

2. Operasi Pohon:

Pembuatan Pohon: Fungsi createRoot digunakan untuk membuat node akar dari pohon.

Penambahan Anak: Fungsi addLeft dan addRight digunakan untuk menambahkan anak kiri dan kanan pada sebuah node.

Penampilkan Anak: Fungsi displayChildren menampilkan nilai anak kiri dan kanan dari sebuah node.

Penampilkan Descendant: Fungsi displayDescendants melakukan traversal level-order pada pohon untuk menampilkan semua keturunan dari sebuah node.

Mengecek BST: Fungsi isBST memeriksa apakah pohon tersebut adalah Binary Search Tree (BST) dengan membandingkan nilai node dengan anak-anaknya.

Menghitung Daun: Fungsi countLeafNodes menghitung jumlah daun pada pohon.

Menghapus Pohon: Fungsi deleteTree menghapus seluruh node dalam pohon secara rekursif.

3. Menu Interaktif:

Program menyediakan menu interaktif yang memungkinkan pengguna untuk melakukan berbagai operasi pada pohon, seperti membuat node, menambahkan anak, menampilkan informasi, dan lain-lain.

4. Fungsi Utama Setiap Bagian Kode

createRoot: Membuat node akar baru.

addLeft, addRight: Menambahkan anak kiri atau kanan pada sebuah node.

displayChildren: Menampilkan anak langsung dari sebuah node.

displayDescendants: Menampilkan semua keturunan dari sebuah node.

isBST: Memeriksa apakah pohon adalah Binary Search Tree.

countLeafNodes: Menghitung jumlah daun pada pohon.

deleteTree: Menghapus seluruh pohon.