LAPORAN PRAKTIKUM
Modul 10
“Tree”

Disusun Oleh:
Ganesha Rahman Gibran -2211104058
Kelas S1SE-07-02

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY
PURWOKERTO
2024

A. TUJUAN PRAKTIKUM

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*.

B. DASAR TEORI

Pengertian Rekursif

Rekursif adalah proses pengulangan melalui pemanggilan dirinya sendiri. Subprogram, seperti prosedur dan fungsi, berguna dalam pemrograman, terutama untuk proyek besar, karena:

1. **Readability**: Membantu memahami program.
2. **Modularity**: Memecah program menjadi bagian kecil sesuai fungsinya, memudahkan pengecekan, pengujian, dan pelacakan kesalahan.
3. **Reusability**: Subprogram dapat digunakan berulang kali tanpa menulis ulang perintahnya.

Fungsi rekursif adalah subprogram yang memanggil dirinya sendiri selama kondisi tertentu terpenuhi. Rekursi terkait erat dengan prinsip induksi matematika.

Contoh perhitungan pangkat dua:

- Secara umum:
  $2^0 = 1$
  $2^n = 2\ x\ 2^{n-1}$
- Algoritma rekursif untuk menghitung 2^n

```
t pangkat_2(int x) {
 if (x == 0) return 1;
 else return 2 * pangkat_2(x - 1);
 }
```

Contoh: pangkat_2(4) menghasilkan $2^4 = 16$ melalui proses rekursif.

Kriteria Rekursif

Fungsi rekursif harus memiliki:

1. Kondisi berhenti: Ketika pemanggilan dirinya tidak diperlukan lagi.
2. Pemanggilan diri: Dilakukan jika kondisi berhenti belum terpenuhi.

Secara umum, struktur fungsi rekursif menggunakan pernyataan kondisional:

- Jika kondisi khusus tidak terpenuhi, panggil fungsi itu sendiri dengan parameter yang sesuai.
- Jika terpenuhi, lakukan instruksi akhir.

Rekursi cocok untuk masalah yang memiliki pola atau langkah-langkah penyelesaian terstruktur. Meskipun lebih sederhana dalam penulisan, rekursi sering kurang efisien secara algoritmis.

Kekurangan Rekursif

Walaupun terlihat mudah dan menghasilkan algoritma yang singkat, rekursi memiliki kelemahan:

1. Membutuhkan lebih banyak memori untuk menyimpan activation record dan variabel lokal.
2. Membutuhkan waktu lebih lama untuk menangani activation record.

Gunakan rekursi hanya jika:

- Penyelesaian secara iteratif sulit dilakukan.
- Efisiensi rekursi memadai.
- Kejelasan logika program lebih penting dibandingkan efisiensi.

Contoh Rekursif

Contoh penggunaan rekursi untuk menghitung pangkat:

```
int pangkat(int x, int y) {
    if (y == 1) return x;
    else return x * pangkat(x, y - 1);
}
```

Contoh penggunaan rekursi untuk menghitung faktorial:

```
long int faktorial(long int a) {
    if (a == 1 || a == 0) return 1;
    else if (a > 1) return a * faktorial(a - 1);
    else return 0;
}
```

Pengertian Pohon

Pohon adalah struktur data non-linear yang memiliki karakteristik berikut:

1. Pohon kosong disebut empty tree.
2. Terdapat satu simpul tanpa pendahulu, yaitu akar (root).
3. Semua simpul lainnya memiliki tepat satu pendahulu.

Terminologi pada pohon:

- Anak dan Orangtua: Simpul yang terhubung langsung.
- Lintasan (path): Urutan simpul dari akar ke simpul tertentu.
- Saudara Kandung: Simpul dengan orangtua yang sama.
- Derajat (degree): Jumlah anak dari sebuah simpul.
- Daun (leaf): Simpul tanpa anak.
- Simpul Dalam (internal node): Simpul yang memiliki anak.
- Tinggi atau Kedalaman (height/depth): Jumlah maksimum simpul dari akar ke daun.

Jenis-Jenis Pohon

1. Ordered Tree: Urutan anak-anak penting.
2. Binary Tree: Setiap simpul memiliki maksimal dua anak. Variannya meliputi:
   - Complete Binary Tree: Semua simpul diisi kecuali simpul terakhir.
   - Extended Binary Tree: Setiap simpul memiliki 0 atau 2 anak.
   - Binary Search Tree (BST): Anak kiri lebih kecil, anak kanan lebih besar.

- AVL Tree: Selisih tinggi subtree kiri dan kanan maksimal 1.
- Heap Tree: Struktur khusus dengan aturan nilai minimum atau maksimum di parent.

Operasi pada Binary Search Tree

- Insert: Menambahkan simpul baru sesuai aturan BST.
- Update: Jika posisi simpul tidak sesuai aturan, regenerasi pohon diperlukan.
- Search: Pencarian dilakukan dengan perbandingan bertahap dari akar hingga simpul yang sesuai

Traversal pada Binary Tree

Traversal adalah proses mengunjungi semua simpul pada pohon dengan pola tertentu:

1. Pre-order: Kunjungi akar, lalu subtree kiri, dan subtree kanan.
2. In-order: Kunjungi subtree kiri, akar, dan subtree kanan.
3. Post-order: Kunjungi subtree kiri, subtree kanan, lalu akar.

C. GUIDED 1

Sourcecode

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

struct Pohon{
    char data;
    Pohon *left, *right;
    Pohon *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};
        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;
    }
    baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
```

```
    cout << "\n Node " << data << " berhasil dibuat menjadi child kiri dari node
" << 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;
    }

    baru = new Pohon{data, NULL, NULL, node};
    node->right=baru;
    cout << "\nNode " << data << " berhasil dibuat menjadi child kanan dari node
" << node->data << endl;
    return baru;
}

void update(char data, Pohon *node){
    if (!node){
        cout << "\nNode " << data << " tidak ditemukan" << endl;
        return;
    }

    char temp = node->data;
    node->data = data;
    cout << "\nData node " << temp << " berhasil diupdate menjadi " << data <<
endl;
}

void find(char data, Pohon *node){
    if(!node) return;
    if (node->data == data) {
        cout << "\nNode " << data << " ditemukan" << endl;
        return;
    }

    find(data, node->left);
    find(data, node->right);
}

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

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

void postOrder(Pohon *node){
    if (!node)
    return;
    postOrder(node->left);
    postOrder(node->right);
```

```
    cout << node -> data << " ";
}

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

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

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

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

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

int main() {
    init();
    buatNode('F');
    insertLeft('B', root);
    insertRight('G', root);
    insertLeft('A', root ->left);
    insertRight('D', root ->left);
    insertLeft('C', root -> left ->right);
    insertRight('E', root->left->right);

    cout << "\nPre-order Traversal: ";
    preOrder(root);
    cout << "\nPost-order Traversal: ";
    postOrder(root);
    cout << "\nIn-order Traversal: ";
    inOrder(root);

    cout << "\nMost Left Node: " << mostLeft(root)->data;
    cout << "\nMost Right Node: " << mostRight(root)->data;

    cout << "\nMenghapus node 'D': ";
```

```
    root = deleteNode(root, 'D');
    cout << "\nIn-Order Traversal setelah penghapusan: ";
    inOrder(root);

    return 0;
}
```

Output

```
PS E:\Struktur Data\2211104058_Ganesha_Rahman_Gibran_SE-06-02\10_Tree_Bagian_1\Unguided\output> cd '
anesha_Rahman_Gibran_SE-06-02\10_Tree_Bagian_1\Guided\output'
PS E:\Struktur Data\2211104058_Ganesha_Rahman_Gibran_SE-06-02\10_Tree_Bagian_1\Guided\output> & .\'g

Node F berhasil dibuat menjadi root

 Node B berhasil dibuat menjadi child kiri dari node F

Node G berhasil dibuat menjadi child kanan dari node F

 Node A berhasil dibuat menjadi child kiri dari node B

Node D berhasil dibuat menjadi child kanan dari node B

 Node C berhasil dibuat menjadi child kiri dari node D

Node E berhasil dibuat menjadi child kanan dari node D

Pre-order Traversal: F B A D C E G
Post-order Traversal: A C E D B G F
In-order Traversal: A B C D E F G
Most Left Node: A
Most Right Node: G
Menghapus node 'D':
In-Order Traversal setelah penghapusan: A B C E F G
PS E:\Struktur Data\2211104058_Ganesha_Rahman_Gibran_SE-06-02\10_Tree_Bagian_1\Guided\output>
```

D. UNGUIDED

1. Modifikasi guided 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 memeriksaapakah 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.

   Sourcecode

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

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

Pohon *root = NULL;

void init() {
    root = NULL;
}
```

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

Pohon* buatNode(char data) {
    return new Pohon{data, NULL, NULL, NULL};
}

void insertNode(char data, Pohon* parent, bool isLeft) {
    if (isLeft) {
        if (parent->left == NULL) {
            parent->left = buatNode(data);
            parent->left->parent = parent;
            cout << "Node " << data << " berhasil ditambahkan ke anak
                        kiri dari " << parent->data << endl;
        } else {
            cout << "Anak kiri dari " << parent->data << " sudah ada!"
                        << endl;
        }
    } else {
        if (parent->right == NULL) {
            parent->right = buatNode(data);
            parent->right->parent = parent;
            cout << "Node " << data << " berhasil ditambahkan ke anak
                        kanan dari " << parent->data << endl;
        } else {
            cout << "Anak kanan dari " << parent->data << " sudah ada!"
                        << endl;
        }
    }
}

void displayChild(Pohon* node) {
    if (!node) {
        cout << "Node tidak ditemukan." << endl;
        return;
    }
    cout << "Node: " << node->data << endl;
    cout << "Anak kiri: " << (node->left ? node->left->data : '-') <<
                        endl;
    cout << "Anak kanan: " << (node->right ? node->right->data : '-')
                        << endl;
}

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

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);
}

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);
}

void menu() {
    int pilihan;
    char data, parentData;
    do {
        cout << "\n=== MENU ===\n";
        cout << "1. Buat Root\n";
        cout << "2. Tambahkan Node\n";
        cout << "3. Tampilkan Child\n";
        cout << "4. Tampilkan Descendant\n";
        cout << "5. Periksa BST Valid\n";
        cout << "6. Hitung Simpul Daun\n";
        cout << "0. Keluar\n";
        cout << "Pilihan: ";
        cin >> pilihan;

        switch (pilihan) {
            case 1:
                if (isEmpty()) {
                    cout << "Masukkan data root: ";
                    cin >> data;
                    root = buatNode(data);
                    cout << "Root " << data << " berhasil dibuat.\n";
                } else {
                    cout << "Root sudah ada.\n";
                }
                break;
            case 2:
                if (isEmpty()) {
                    cout << "Buat root terlebih dahulu.\n";
                } else {
                    cout << "Masukkan data parent: ";
                    cin >> parentData;
                    cout << "Masukkan data node baru: ";
                    cin >> data;
                    cout << "Tambahkan ke (1) kiri atau (2) kanan? ";
                    int pos;
                    cin >> pos;

                    Pohon* current = root;
                    // Pencarian parent
                    while (current && current->data != parentData) {
                        current = (current->left && parentData <
                        current->data) ? current->left : current-
                        >right;
                    }
                    if (!current) {
                        cout << "Parent tidak ditemukan.\n";
                    } else {
                        insertNode(data, current, pos == 1);
                    }
                }
                break;
            case 3:
                if (isEmpty()) {
                    cout << "Pohon kosong.\n";
                } else {
                    cout << "Masukkan data node: ";
                    cin >> data;
```

```
                    Pohon* current = root;
                    // Pencarian node
                    while (current && current->data != data) {
                        current = (current->left && data < current-
                        >data) ? current->left : current->right;
                    }
                    displayChild(current);
                }
                break;
            case 4:
                if (isEmpty()) {
                    cout << "Pohon kosong.\n";
                } else {
                    cout << "Masukkan data node: ";
                    cin >> data;
                    Pohon* current = root;
                    // Pencarian node
                    while (current && current->data != data) {
                        current = (current->left && data < current-
                        >data) ? current->left : current->right;
                    }
                    if (!current) {
                        cout << "Node tidak ditemukan.\n";
                    } else {
                        cout << "Descendant dari " << data << ": ";
                        displayDescendants(current);
                        cout << endl;
                    }
                }
                break;
            case 5:
                if (is_valid_bst(root, numeric_limits<char>::min(),
                        numeric_limits<char>::max())) {
                    cout << "Pohon adalah Binary Search Tree yang
                        valid.\n";
                } else {
                    cout << "Pohon bukan Binary Search Tree yang
                        valid.\n";
                }
                break;
            case 6:
                cout << "Jumlah simpul daun: " <<
                        cari_simpul_daun(root) << endl;
                break;
            case 0:
                cout << "Keluar dari program.\n";
                break;
            default:
                cout << "Pilihan tidak valid.\n";
        }
    } while (pilihan != 0);
}

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

Output

```
=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 1
Masukkan data root: A
Root A berhasil dibuat.

=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 2
Masukkan data parent: A
Masukkan data node baru: B
Tambahkan ke (1) kiri atau (2) kanan? 1
Node B berhasil ditambahkan ke anak kiri dari A

=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 2
Masukkan data parent: A
Masukkan data node baru: C
Tambahkan ke (1) kiri atau (2) kanan? 2
Node C berhasil ditambahkan ke anak kanan dari A

=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 3
Masukkan data node: A
Node: A
Anak kiri: B
Anak kanan: C
```

```
=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 4
Masukkan data node: A
Descendant dari A: B C

=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 5
Pohon bukan Binary Search Tree yang valid.
```

```
=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 6
Jumlah simpul daun: 2

=== MENU ===
1. Buat Root
2. Tambahkan Node
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa BST Valid
6. Hitung Simpul Daun
0. Keluar
Pilihan: 0
Keluar dari program.
PS E:\Struktur Data\2211104058_Ganesha_Rahman_Gibran_SE-06-02\10_Tree_B
```

E. KESIMPULAN

Rekursif adalah metode pemrograman yang memungkinkan sebuah fungsi memanggil dirinya sendiri, sering digunakan untuk menyelesaikan masalah yang memiliki pola penyelesaian berulang atau terstruktur, meskipun kurang efisien dibanding iterasi. Struktur data pohon, khususnya Binary Tree, merupakan representasi hierarkis yang terdiri dari simpul-simpul dengan berbagai operasi seperti pencarian, penyisipan, dan traversal. Pohon memberikan solusi efisien untuk pengelolaan data, terutama dalam pengorganisasian dan pengolahan informasi yang kompleks. Kombinasi rekursi dan struktur pohon sering digunakan untuk menyelesaikan masalah pemrograman secara logis dan terorganisir.