LAPORAN PRAKTIKUM
Modul 10
TREE (BAGIAN PERTAMA)

Disusun Oleh:

Muhammad Shafiq Rasuna - 2311104043

Kelas :

S1SE-07-02

Dosen :

Wahyu Andi Saputra, S.Pd, M.Eng

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY

PURWOKERTO

2024

1. 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.

2. Dasar 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.

2.2 Kriteria Rekursif

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).

2.3 Kekurangan Rekursif

Konsep penggunaan yang terlihat mudah karena fungsi rekursif dapat menyederhanakan solusi dari suatu permasalahan, sehingga sering kali menghasilkan bentuk algoritma dan program yang lebih singkat dan lebih mudah dimengerti.

3. Guided

Kode programnya:

```cpp
#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 << "\nNode " << data << " berhasil ditambahkan ke child kiri " << 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 ditambahkan ke child kanan " << node->data << endl;
    return baru;
}

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;
}
```

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

    if (node->data == data) {
        cout << "\nNode ditemukan: " << data << 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 << "\nIn-order Traversal: ";
    inOrder(root);
    cout << "\nPost-order Traversal: ";
    postOrder(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;
}
```

Hasil runnya:

```
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
c:\Users\ASUS\OneDrive\Dokumen\tugas smt  3\Pemograman Struktur Data 3\pertemuan10>
```

4. 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!

   Kode program :

```cpp
#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 << "\nNode " << data << " berhasil ditambahkan ke child kiri " << 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 ditambahkan ke child kanan " << node->data << endl;
    return baru;
}

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

```cpp
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 << " ";
}

void tampilkanChild(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan!" << endl;
        return;
    }
    cout << "\nNode " << node->data << " memiliki child: ";
    if (node->left) cout << "Kiri: " << node->left->data << " ";
    else cout << "Kiri: NULL ";
    if (node->right) cout << "Kanan: " << node->right->data << endl;
    else cout << "Kanan: NULL" << endl;
}

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

Pohon* cariNode(Pohon *node, char data) {
    if (!node) return NULL;
    if (node->data == data) return node;
    Pohon *leftSearch = cariNode(node->left, data);
    if (leftSearch) return leftSearch;
    return cariNode(node->right, data);
}

int main() {
    init();
    char pilihan, data, parentData;
    Pohon *node;

    do {
        cout << "\nMenu:\n";
        cout << "1. Buat Node Root\n";
        cout << "2. Tambah Child Kiri\n";
        cout << "3. Tambah Child Kanan\n";
        cout << "4. Tampilkan Pre-order\n";
        cout << "5. Tampilkan In-order\n";
        cout << "6. Tampilkan Post-order\n";
        cout << "7. Tampilkan Child dari Node\n";
        cout << "8. Tampilkan Descendant dari Node\n";
        cout << "9. Keluar\n";
        cout << "Pilih: ";
        cin >> pilihan;
```

```cpp
        switch (pilihan) {
            case '1':
                cout << "Masukkan data untuk root: ";
                cin >> data;
                buatNode(data);
                break;
            case '2':
                cout << "Masukkan data untuk node parent: ";
                cin >> parentData;
                node = cariNode(root, parentData);
                if (node) {
                    cout << "Masukkan data untuk child kiri: ";
                    cin >> data;
                    insertLeft(data, node);
                } else {
                    cout << "Node parent tidak ditemukan!" << endl;
                }
                break;
            case '3':
                cout << "Masukkan data untuk node parent: ";
                cin >> parentData;
                node = cariNode(root, parentData);
                if (node) {
                    cout << "Masukkan data untuk child kanan: ";
                    cin >> data;
                    insertRight(data, node);
                } else {
                    cout << "Node parent tidak ditemukan!" << endl;
                }
                break;
            case '4':
                cout << "\nPre-order Traversal: ";
                preOrder(root);
                cout << endl;
                break;
            case '5':
                cout << "\nIn-order Traversal: ";
                inOrder(root);
                cout << endl;
                break;
            case '6':
                cout << "\nPost-order Traversal: ";
                postOrder(root);
                cout << endl;
                break;
            case '7':
                cout << "Masukkan data node untuk melihat child: ";
                cin >> data;
                node = cariNode(root, data);
                tampilkanChild(node);
                break;
            case '8':
                cout << "Masukkan data node untuk melihat descendant: ";
                cin >> data;
                node = cariNode(root, data);
                if (node) {
                    cout << "\nDescendant dari node " << node->data << ": ";
                    tampilkanDescendant(node);
                    cout << endl;
                } else {
                    cout << "Node tidak ditemukan!" << endl;
                }
                break;
            case '9':
                cout << "Keluar dari program.\n";
                break;
            default:
                cout << "Pilihan tidak valid!\n";
        }
    } while (pilihan != '9');

    return 0;
}

```

Hasil outputnya :

```
Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 1
Masukkan data untuk root: W

Node W berhasil dibuat menjadi root.

Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 2
Masukkan data untuk node parent: W
Masukkan data untuk child kiri: Z

Node Z berhasil ditambahkan ke child kiri
```

```
Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 3
Masukkan data untuk node parent: W
Masukkan data untuk child kanan: Y

Node Y berhasil ditambahkan ke child kanan W

Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 4

Pre-order Traversal: W Z Y
```

```
Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 5

In-order Traversal: Z W Y

Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 6

Post-order Traversal: Z Y W
```

```
Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 7
Masukkan data node untuk melihat child: W

Node W memiliki child: Kiri: Z Kanan: Y

Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 8
Masukkan data node untuk melihat descendant: W

Descendant dari node W: W Z Y
```

```
Menu:
1. Buat Node Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Pre-order
5. Tampilkan In-order
6. Tampilkan Post-order
7. Tampilkan Child dari Node
8. Tampilkan Descendant dari Node
9. Keluar
Pilih: 9
Keluar dari program.

c:\Users\ASUS\OneDrive\Dokumen\tugas smt  3\Pemograman Struktur Data 3\pertemuan10>
```

Penjelasan Program :

1. **Menu Interaktif**:
   - Pengguna dapat memilih operasi dari menu.
   - Operasi termasuk pembuatan root, penambahan child kiri/kanan, traversal, dan menampilkan child/descendant.
2. **Fungsi Tambahan**:
   - tampilkanChild: Menampilkan child kiri dan kanan dari node yang dipilih.
   - tampilkanDescendant: Menampilkan semua descendant dari node yang dipilih.
   - cariNode: Mencari node berdasarkan data yang diberikan.

2. Buatlah fungsi rekursif is_valid_bst(node, min_val, max_val) untuk memeriksa apakah suatu pohon memenuhi properti Binary Search Tree. Uji fungsi ini pada berbagai pohon, baik yang valid maupun tidak valid sebagai BST.
Kode programnya:

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

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

Pohon* root = NULL;

void buatNode(char data) {
    root = new Pohon{data, NULL, NULL};
}

Pohon* insertLeft(Pohon* node, char data) {
    node->left = new Pohon{data, NULL, NULL};
    return node->left;
}

Pohon* insertRight(Pohon* node, char data) {
    node->right = new Pohon{data, NULL, NULL};
    return 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;
    }
```

```cpp
return is_valid_bst(node->left, min_val, node->data) &&
           is_valid_bst(node->right, node->data, max_val);
}

bool checkBST(Pohon* root) {
    return is_valid_bst(root, numeric_limits<char>::min(), numeric_limits<char>::max());
}

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

int main() {
    cout << "Membuat pohon BST yang valid:" << endl;
    buatNode('F');
    Pohon* nodeB = insertLeft(root, 'B');
    Pohon* nodeG = insertRight(root, 'G');
    insertLeft(nodeB, 'A');
    insertRight(nodeB, 'D');
    insertLeft(nodeG, 'I');
    insertRight(nodeG, 'H');

    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;

    if (checkBST(root)) {
        cout << "Pohon ini adalah BST yang valid." << endl;
    } else {
        cout << "Pohon ini BUKAN BST yang valid." << endl;
    }

    cout << "\nMembuat pohon yang TIDAK valid sebagai BST:" << endl;
    buatNode('F');
    nodeB = insertLeft(root, 'G');
    nodeG = insertRight(root, 'B');

    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;

    if (checkBST(root)) {
        cout << "Pohon ini adalah BST yang valid." << endl;
    } else {
        cout << "Pohon ini BUKAN BST yang valid." << endl;
    }

    return 0;
}
```

Output nya :

```
Membuat pohon BST yang valid:
In-order Traversal: A B D F I G H
Pohon ini BUKAN BST yang valid.

Membuat pohon yang TIDAK valid sebagai BST:
In-order Traversal: G F B
Pohon ini BUKAN BST yang valid.

c:\Users\ASUS\OneDrive\Dokumen\tugas smt  3\Pemograman Struktur Data 3\pertemuan10>
```

Penjelasan

1. **Parameter**:
   - node: Pointer ke node yang sedang diperiksa.
   - min_val: Nilai minimum yang diperbolehkan untuk node.
   - max_val: Nilai maksimum yang diperbolehkan untuk node.
2. **Base Case**: Jika node adalah NULL, kembalikan true.
3. **Validasi Node**: Periksa apakah node->data berada di antara min_val dan max_val.
4. **Rekursif**:
   - Panggil is_valid_bst untuk subtree kiri dengan rentang [min_val, node->data].
   - Panggil is_valid_bst untuk subtree kanan dengan rentang [node->data, max_val].

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.

Kode Programnya :

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

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

Pohon* root = NULL;

void buatNode(char data) {
    root = new Pohon{data, NULL, NULL};
}

Pohon* insertLeft(Pohon* node, char data) {
    node->left = new Pohon{data, NULL, NULL};
    return node->left;
}

Pohon* insertRight(Pohon* node, char data) {
    node->right = new Pohon{data, NULL, NULL};
    return node->right;
}

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

int main() {
    buatNode('A');
    Pohon* nodeB = insertLeft(root, 'B');
    Pohon* nodeC = insertRight(root, 'C');
    insertLeft(nodeB, 'D');
    insertRight(nodeB, 'E');
    insertRight(nodeC, 'F');

    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;

    int jumlahDaun = cari_simpul_daun(root);
    cout << "Jumlah simpul daun dalam pohon: " << jumlahDaun << endl;

    return 0;
}
```

Output nya:

```
In-order Traversal: D B E A C F
Jumlah simpul daun dalam pohon: 3

c:\Users\ASUS\OneDrive\Dokumen\tugas smt  3\Pemograman Struktur Data 3\pertemuan10>
```

Penjelasan

1. **Base Case:**
   - Jika node adalah NULL, kembalikan 0 karena tidak ada simpul.
2. **Simpul Daun:**
   - Jika node tidak memiliki anak kiri dan anak kanan (!node->left && !node->right), maka node adalah simpul daun. Kembalikan 1.
3. **Rekursi:**
   - Panggil cari_simpul_daun pada subtree kiri dan subtree kanan.
   - Jumlahkan hasil dari kedua subtree untuk mendapatkan total jumlah simpul daun.

5. Kesimpulan

Pada praktikum ini, tujuan utama adalah memahami dan mengimplementasikan fungsi rekursif serta mengaplikasikan struktur data tree, khususnya Binary Tree, dalam pemrograman. Rekursi merupakan konsep penting yang memanfaatkan fungsi yang memanggil dirinya sendiri untuk menyelesaikan masalah yang memiliki pola penyelesaian berulang. Penggunaan fungsi rekursif memberikan keuntungan dalam hal readability (kemudahan membaca kode), modularity (pemecahan masalah menjadi bagian-bagian yang lebih kecil), dan reusability (penggunaan kembali subprogram tanpa menulis ulang kode).

Namun, rekursi juga memiliki kekurangan karena sering kali kurang efisien dalam penggunaan memori dan waktu eksekusi. Praktikum ini menunjukkan implementasi berbagai fungsi rekursif dalam konteks Binary Tree, seperti validasi pohon sebagai Binary Search Tree (BST) menggunakan is_valid_bst dan menghitung jumlah simpul daun dengan fungsi cari_simpul_daun. Selain itu, implementasi program dengan menu interaktif memungkinkan pengguna untuk menambahkan node secara dinamis, melakukan traversal pohon, serta menampilkan child dan descendant dari node tertentu.

Dengan praktikum ini, peserta dapat memahami bahwa meskipun rekursi memiliki kelemahan dalam efisiensi eksekusi, ia sering kali menghasilkan solusi yang lebih sederhana dan mudah dipahami untuk masalah kompleks, terutama dalam struktur data tree seperti Binary Tree dan Binary Search Tree.