LAPORAN PRAKTIKUM

PERTEMUAN 9

Modul 10-11

Disusun Oleh:

Naura Aisha Zahira (2311104078)

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. Landasan Teori

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.

3. Guided

1. Guided 1

Sourcecode:

```
#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
```

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!
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.
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.

Code:

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

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

Pohon *root;

void init() {
   root = NULL;
}

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

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

```
Pohon* insertLeft(char data, Pohon* node) {
    if (!node) return NULL;
    if (node->left) {
        cout << "Child kiri sudah ada!\n";
        return NULL;
    }
    node->left = buatNode(data, node);
    return node->left;
}

Pohon* insertRight(char data, Pohon* node) {
    if (!node) return NULL;
    if (node->right) {
        cout << "Child kanan sudah ada!\n";
        return NULL;
    }
    node->right = buatNode(data, node);
    return 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) {
```

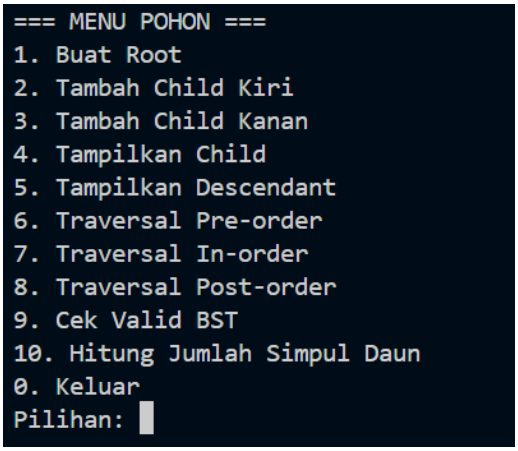
```
    if (!node) return;
    postOrder(node->left);
    postOrder(node->right);
    cout << node->data << " ";
}

void tampilkanChild(Pohon* node) {
    if (!node) return;
    cout << "Node " << node->data << " memiliki:\n";
    if (node->left) cout << "- Child kiri: " << node->left->data << endl;
    else cout << "- Tidak ada child kiri.\n";
    if (node->right) cout << "- Child kanan: " << node->right->data << endl;
    else cout << "- Tidak ada child kanan.\n";
}

void tampilkanDescendant(Pohon* node) {
    if (!node) return;
    cout << "Descendant dari " << node->data << ": ";
    preOrder(node);
    cout << endl;
}

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, parent;
    Pohon* currentNode = NULL;

    do {
        cout << "\n=== MENU POHON ===\n";
        cout << "1. Buat Root\n";
        cout << "2. Tambah Child Kiri\n";
        cout << "3. Tambah Child Kanan\n";
        cout << "4. Tampilkan Child\n";
        cout << "5. Tampilkan Descendant\n";
        cout << "6. Traversal Pre-order\n";
        cout << "7. Traversal In-order\n";
        cout << "8. Traversal Post-order\n";
        cout << "9. Cek Valid BST\n";
        cout << "10. Hitung Jumlah Simpul Daun\n";
        cout << "0. Keluar\n";
        cout << "Pilihan: ";
        cin >> pilihan;

        switch (pilihan) {
            case 1:
                if (!root) {
                    cout << "Masukkan data root: ";
                    cin >> data;
                    root = buatNode(data);
                    cout << "Root berhasil dibuat.\n";
                } else {
                    cout << "Root sudah ada.\n";
                }
                break;
            case 2:
                cout << "Masukkan parent node: ";
                cin >> parent;
```

```
            cout << "Masukkan data: ";
            cin >> data;
            currentNode = insertLeft(data, root);
            if (currentNode) cout << "Child kiri berhasil ditambahkan.\n";
            break;
        case 3:
            cout << "Masukkan parent node: ";
            cin >> parent;
            cout << "Masukkan data: ";
            cin >> data;
            currentNode = insertRight(data, root);
            if (currentNode) cout << "Child kanan berhasil ditambahkan.\n";
            break;
        case 4:
            cout << "Masukkan node: ";
            cin >> data;
            tampilkanChild(root);
            break;
        case 5:
            cout << "Masukkan node: ";
            cin >> data;
            tampilkanDescendant(root);
            break;
        case 6:
            cout << "Pre-order Traversal: ";
            preOrder(root);
            cout << endl;
            break;
        case 7:
            cout << "In-order Traversal: ";
            inOrder(root);
            cout << endl;
            break;
        case 8:
            cout << "Post-order Traversal: ";
```

```
            postOrder(root);
            cout << endl;
            break;
        case 9:
            if (is_valid_bst(root, numeric_limits<char>::min(), numeric_limits<char>::max()))
{
                cout << "Pohon adalah Binary Search Tree.\n";
            } else {
                cout << "Pohon bukan Binary Search Tree.\n";
            }
            break;
        case 10:
            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() {
   init();
   menu();
   return 0;
}
```

Output:


=== MENU POHON ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child
5. Tampilkan Descendant
6. Traversal Pre-order
7. Traversal In-order
8. Traversal Post-order
9. Cek Valid BST
10. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan:

5. Kesimpulan

Praktikum ini berhasil menerapkan konsep rekursif dan struktur data binary tree untuk berbagai operasi, seperti pembuatan node, traversal, pencarian, dan validasi Binary Search Tree, serta menghitung jumlah simpul daun. Hal ini meningkatkan pemahaman tentang pemrograman modular dan efisien.