LAPORAN PRAKTIKUM
Modul XI
“Tree”

Disusun Oleh:
Dewi Atika Muthi -2211104042
SE-07-02

Dosen:
Wahyu Andi Saputra

PROGRAM STUDI S1 SOFTWARE ENGINEERING
FAKULTAS INFORMATIKA
TELKOM UNIVERSITY
PURWOKERTO

2024

1. **Tujuan**

Tujuan praktikum ini:

1. Memahami konsep penggunaan fungsi rekursif dalam implementasi tree.
2. Mengimplementasikan bentuk-bentuk fungsi rekursif untuk operasi pada binary tree.
3. Mengaplikasikan struktur data tree dalam program.
4. Memahami konsep binary search tree, termasuk operasi traversal, pencarian, dan penghapusan node.

2. **Landasan Teori**

Tree merupakan salah satu struktur data non-linear yang digunakan untuk merepresentasikan hubungan hierarkis antara elemen-elemen data. Tree tersusun atas node-node yang terhubung oleh cabang (edges). Node dalam tree memiliki beberapa elemen penting:

- Root: Node pertama yang menjadi akar tree.
- Child: Node yang memiliki induk (parent).
- Leaf: Node tanpa child.
- Parent: Node yang memiliki child.
- Siblings: Node-node dengan parent yang sama.

Tree memiliki berbagai jenis, salah satunya adalah Binary Tree, di mana setiap node memiliki maksimal dua child (left dan right). Salah satu implementasi binary tree yang sering digunakan adalah Binary Search Tree (BST), yang memiliki aturan sebagai berikut:

- Nilai di subtree kiri lebih kecil dari nilai parent.
- Nilai di subtree kanan lebih besar dari nilai parent.

Jenis Traversal dalam Tree:

Traversal adalah proses untuk mengunjungi setiap node di dalam tree. Ada tiga jenis traversal utama:

- Pre-order (Root, Left, Right): Kunjungi root terlebih dahulu, kemudian traversal ke subtree kiri, lalu ke subtree kanan.
- In-order (Left, Root, Right): Traversal dimulai dari subtree kiri, lalu ke root, kemudian subtree kanan.
- Post-order (Left, Right, Root): Traversal dimulai dari subtree kiri, kemudian subtree kanan, dan terakhir ke root.

Operasi Dasar pada Binary Tree:

- Pembuatan Tree: Membuat node root dan menambahkan child ke tree.
- Pencarian Node: Mencari node dengan nilai tertentu.
- Pembaruan Node: Mengubah nilai node tertentu.
- Penghapusan Node: Menghapus node dengan memperhatikan kondisi berikut:

- Node tanpa child (leaf).
- Node dengan satu child (child menggantikan posisi node).
- Node dengan dua child (mengganti node dengan successor atau predecessor).

Konsep Rekursi dalam Tree:

Rekursi adalah metode pemrograman di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah. Dalam operasi tree, rekursi banyak digunakan untuk mempermudah navigasi melalui node-node tree. Misalnya:

- Traversal tree menggunakan fungsi rekursif untuk mengunjungi setiap node.
- Fungsi pencarian dan penghapusan node juga memanfaatkan rekursi untuk menavigasi tree.

Manfaat dan Kekurangan Rekursi:

Manfaat:

- Membuat kode lebih ringkas dan mudah dipahami.
- Mempermudah implementasi struktur data hierarkis, seperti tree.

Kekurangan:

- Menggunakan lebih banyak memori karena setiap pemanggilan fungsi membutuhkan ruang pada call stack.
- Jika tidak dirancang dengan baik, rekursi dapat menyebabkan stack overflow.
- Binary Tree dalam Kehidupan Nyata

Binary tree digunakan di berbagai bidang, seperti:

- Organisasi Data: Digunakan dalam database untuk menyimpan data dengan struktur hierarkis.
- Pencarian Cepat: BST digunakan untuk melakukan pencarian data dengan kompleksitas O(log n).
- Pemrograman Game: Digunakan dalam AI untuk pengambilan keputusan, seperti pohon pencarian game.

3. Guided

1) Tree praktikum

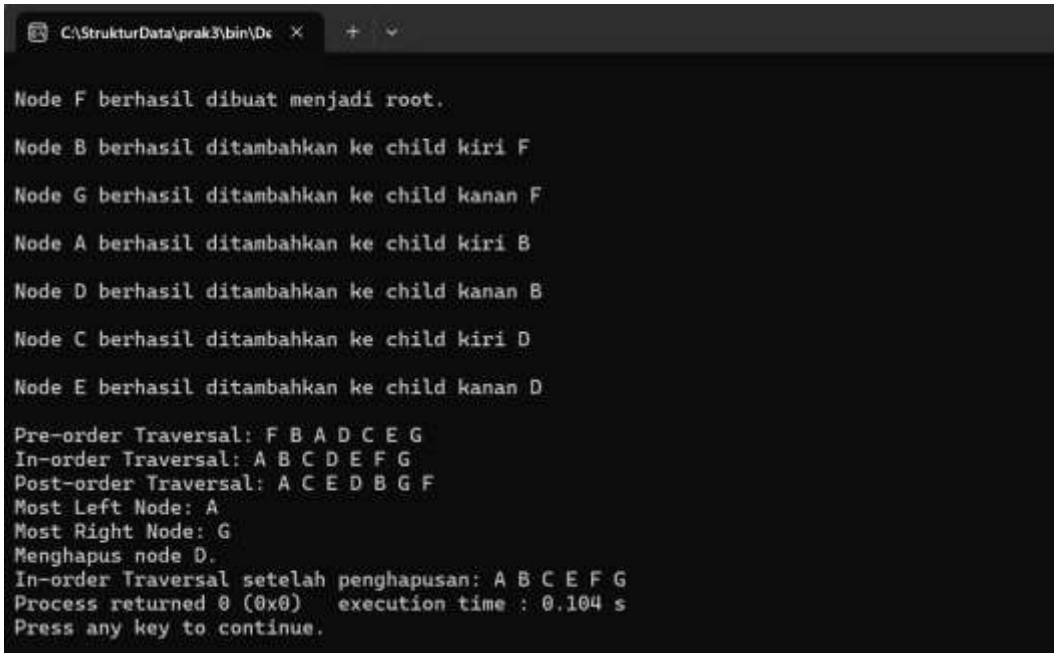
```
#include <iostream>
using namespace std;

/// 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:



Penjelasan Program:

Program ini membangun binary tree dengan fungsi untuk menambahkan node, traversal (pre-order, in-order, post-order), pencarian node, pembaruan data node, dan penghapusan node. Operasi dilakukan menggunakan pendekatan rekursif untuk mempermudah navigasi melalui node.

4. Unguided

a) Modifikasi guided tree diatas dengan program menu menggunakan input data tree
b) dari user dan berikan fungsi tambahan untuk menampilkan node child dan descendant dari node yang diinputkan!
c) 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.
d) 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.

Source code:

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

// Struktur data tree seperti pada guided
struct Pohon {
    char data;
    Pohon *left, *right, *parent;
};

// Variabel global
Pohon *root = NULL, *baru = NULL;
```

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

// Membuat node baru sebagai root
void buatNode(char data) {
    if (!root) {
        root = new Pohon{data, NULL, NULL, NULL};
        cout << "Node " << data << " berhasil dibuat sebagai root.\n";
    } else {
        cout << "Root sudah ada.\n";
    }
}

// Menambahkan node ke anak kiri
Pohon* insertLeft(char data, Pohon *node) {
    if (node->left != NULL) {
        cout << "Node " << node->data << " sudah memiliki child
   kiri!\n";
        return NULL;
    }
    baru = new Pohon{data, NULL, NULL, node};
    node->left = baru;
    return baru;
}

// Menambahkan node ke anak kanan
Pohon* insertRight(char data, Pohon *node) {
    if (node->right != NULL) {
        cout << "Node " << node->data << " sudah memiliki child
   kanan!\n";
        return NULL;
    }
    baru = new Pohon{data, NULL, NULL, node};
    node->right = baru;
    return baru;
}

// Fungsi untuk mencari node
Pohon* findNode(char data, Pohon *node) {
    if (!node) return NULL;
    if (node->data == data) return node;
    Pohon *leftResult = findNode(data, node->left);
    if (leftResult) return leftResult;
    return findNode(data, node->right);
}

// Menampilkan child dari node tertentu
void tampilkanChild(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan.\n";
        return;
    }
    cout << "Node " << node->data << ":\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";
```

```
}

// Menampilkan descendant dari node tertentu
void tampilkanDescendant(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan.\n";
        return;
    }
    queue<Pohon*> q;
    q.push(node);
    cout << "Descendant dari node " << node->data << ": ";
    while (!q.empty()) {
        Pohon *current = q.front();
        q.pop();
        if (current != node) cout << current->data << " ";
        if (current->left) q.push(current->left);
        if (current->right) q.push(current->right);
    }
    cout << endl;
}

// Fungsi rekursif untuk memeriksa apakah tree valid 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 menu utama
void menu() {
    char pilihan, data, parent;
    while (true) {
        cout << "\n=== Menu Binary Tree ===\n";
        cout << "1. Tambah Root\n2. Tambah Child\n3. Tampilkan
   Child\n4. Tampilkan Descendant\n";
        cout << "5. Periksa Validitas BST\n6. Hitung Simpul Daun\n7.
   Keluar\n";
        cout << "Pilihan: ";
        cin >> pilihan;

        switch (pilihan) {
            case '1':
                cout << "Masukkan data root: ";
                cin >> data;
                buatNode(data);
                break;
            case '2':
                cout << "Masukkan data parent: ";
                cin >> parent;
                cout << "Masukkan data child: ";
                cin >> data;
                cout << "Kiri (L) atau Kanan (R)? ";
```

```
                char arah;
                cin >> arah;
                if (arah == 'L' || arah == 'l') {
                    insertLeft(data, findNode(parent, root));
                } else {
                    insertRight(data, findNode(parent, root));
                }
                break;
            case '3':
                cout << "Masukkan data node: ";
                cin >> data;
                tampilkanChild(findNode(data, root));
                break;
            case '4':
                cout << "Masukkan data node: ";
                cin >> data;
                tampilkanDescendant(findNode(data, root));
                break;
            case '5':
                if (is_valid_bst(root, CHAR_MIN, CHAR_MAX))
                    cout << "Tree adalah Binary Search Tree yang
    valid.\n";
                else
                    cout << "Tree bukan Binary Search Tree yang
    valid.\n";
                break;
            case '6':
                cout << "Jumlah simpul daun: " <<
    cari_simpul_daun(root) << endl;
                break;
            case '7':
                return;
            default:
                cout << "Pilihan tidak valid.\n";
        }
    }
}

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

Output:

Binary Search Tree tidak valid:

```
=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 1
Masukkan data root: A
Node A berhasil dibuat sebagai root.

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: A
Masukkan data child: B
Kiri (L) atau Kanan (R)? L

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: B
Masukkan data child: C
Kiri (L) atau Kanan (R)? L
```

```
=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: B
Masukkan data child: D
Kiri (L) atau Kanan (R)? R

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 4
Masukkan data node: A
Descendant dari node A: B C D

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 5
Tree bukan Binary Search Tree yang valid.

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 6
Jumlah simpul daun: 2
```

Binary search tree valid:

(Tambah root dan child)

```
=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 1
Masukkan data root: F
Node F berhasil dibuat sebagai root.

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: F
Masukkan data child: B
Kiri (L) atau Kanan (R)? L

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: F
Masukkan data child: G
Kiri (L) atau Kanan (R)? R
```

```
=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: B
Masukkan data child: A
Kiri (L) atau Kanan (R)? L

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: B
Masukkan data child: D
Kiri (L) atau Kanan (R)? R
```

```
=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: D
Masukkan data child: C
Kiri (L) atau Kanan (R)? L

=== Menu Binary Tree ===
1. Tambah Root
2. Tambah Child
3. Tampilkan Child
4. Tampilkan Descendant
5. Periksa Validitas BST
6. Hitung Simpul Daun
7. Keluar
Pilihan: 2
Masukkan data parent: D
Masukkan data child: E
Kiri (L) atau Kanan (R)? R
```

(Menampilkan child dan meenampilkan descendant)

(Periksa Validitas BTS, hitung simpul daun dan keluar)

Penjelasan progam:

Tambahan Menu:

- Tambah Root: Membuat root dari tree dengan data masukan pengguna.
- Tambah Child: Menambahkan anak kiri atau kanan ke node yang sudah ada.
- Tampilkan Child: Menampilkan anak kiri dan kanan dari node tertentu.
- Tampilkan Descendant: Menampilkan semua keturunan (descendant) dari

node tertentu.
- Periksa Validitas BST: Memeriksa apakah tree yang dibuat memenuhi aturan BST.
- Hitung Simpul Daun: Menghitung jumlah simpul daun pada tree.

Fungsi is_valid_bst:
Fungsi ini memeriksa apakah semua node dalam tree memenuhi aturan BST dengan membandingkan nilai node dengan batas bawah (min_val) dan batas atas (max_val).

Fungsi cari_simpul_daun:
Fungsi ini menghitung jumlah simpul daun dalam tree secara rekursif. Simpul daun adalah node yang tidak memiliki child kiri maupun kanan.

5. Kesimpulan

Dari praktikum ini, diperoleh pemahaman tentang:

1) Penggunaan rekursi untuk operasi pada binary tree.
2) Cara membangun tree secara dinamis dan melakukan traversal.
3) Penghapusan node dalam tree, termasuk pengelolaan tree setelah penghapusan.
4) Validasi struktur BST dan perhitungan simpul daun menggunakan pendekatan rekursif.