LAPORAN PRAKTIKUM

MODUL X - XI

“TREE (bagian 1-2)”

Disusun Oleh:

Alya Rabani - 2311104076

S1SE-07-02

Dosen :

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2024

1. Tujuan

- Memahami konsep penggunaan fungsi rekursif.
- Mengimplementasikan bentuk-bentuk fungsi rekursif.
- Mengaplikasikan struktur data tree dalam sebuah kasus pemrograman.
- Mengimplementasikan struktur data tree, khususnya Binary Tree.

2. Landasan Teori

Fungsi rekursif adalah proses di mana suatu fungsi memanggil dirinya sendiri secara langsung atau tidak langsung. Dengan menggunakan algoritma rekrusif, masalah tertentu dapat diselesaikan dengan cukup mudah. Pada beberapa kasus fungsi rekrusif bisa lebih mudah dipahami dan sederhana daripada dengan Solusi iterative. Maka dari itu penggunaan fungsi ini dapat membantu mengurangi jumlah kode yang dibutuhkan serta memudahkan untuk dibaca. Di dalam system pemrograman sendiri, rekursif memiliki dua jenis yaitu, rekrusif langsung dimana fungsi memanggil dirinya sendiri secara langsung dalam proses eksekusi dan tidak langsung merupakan fungsi a memanggi fungsi b dan fungsi b memanggil fungsi a.

Salah satu bentuk rekursi langsung adalah tree recursion di mana fungsi memanggil dirinya sendiri lebih dari satu kali dalam satu langkah eksekusi. Pola ini menciptakan struktur seperti pohon (tree structure), karena setiap pemanggilan menghasilkan cabang baru yang dapat memanggil rekursi lagi.

Ciri-Ciri Tree Recursion:

- Fungsi memanggil dirinya sendiri beberapa kali dalam satu langkah eksekusi.
- Biasanya digunakan untuk masalah yang memiliki sub-masalah bercabang.
- Kompleksitas waktu cenderung meningkat secara eksponensial karena banyaknya pemanggilan ulang fungsi untuk nilai yang sama (kecuali menggunakan teknik seperti memoization).

Pada struktur data tree adalah struktur hierarkis yang terdiri dari Kumpulan simpul (nodes) yang dihubungkan oleh cabang (edges). Tree sering digunakan untuk merepresentasikan data yang memiliki hubungan hierarkis atau bersarang, seperti struktur file di komputer, silsilah keluarga, atau ekspresi matematika. Komponen dasar tree ada Root yang merupakan node utama atau simpul paling atas dari tree, Parent node yang memiliki cabang menuju node lain, child node yang terhubung dari parent, leaf node yang tidak memiliki child, edge hubungan atau garis yang menghubungkan dua node, subtree bagian dari tree yang merupakan tree kecil dengan root sendiri, level jarak suatu node dari root dalam hal jumlah edges, height panjang jalur terpanjang dari root ke leaf, degree jumlah child yang dimiliki oleh suatu node.

Tree memiliki banyak jenis yang paling familiar adalah Binary Search Tree (BST). Digunakan untuk pencarian cepat, penyisipan, dan penghapusan data yang terurut. BST memiliki keunggulan seperti operasi pencarian, penyisipan, dan penghapusan dapat dilakukan dalam waktu rata-rata O(log⁡n)O(\log n)O(logn)

jika tree seimbang. BST biasa digunakan untuk penggunaan basis data untuk indeks sederhana dan aplikasi pencarian sistem yang membutuhkan data terurut, seperti daftar produk. Namun, BST tidak selalu efisien jika tree menjadi tidak seimbang (skewed), sehingga perlu optimisasi seperti AVL atau Red-Black Tree.

Operasi-operasi yang digunakan pada tree seperti berikut:

- Traversal/Penelusuran Tree, adalah proses menjelajahi semua node dalam tree. Contohnya:
    - ✓ Pre-order: Root → Left → Right.
    - ✓ In-order: Left → Root → Right.
    - ✓ Post-order: Left → Right → Root.
    - ✓ Level-order: Traversal per level dari atas ke bawah.
- Insertion/penyisipan, untuk menambahkan node baru sesuai aturan tree.
- Deletion/penghapusan, menghapus node dan menjaga struktur tree tetap valid.
- Search, pencarian node tertentu berdasarkan nilai.

3. Guided

Pada guided ini merupakan implementasi dari struktur data tree biner. Program ini dimulai dengan mendefinisikan struktur tree, yang menyimpan data karakter dan pointer ke anak kiri, anak kanan, serta induk dari setiap node. Fungsi init digunakan untuk menginisialisasi pohon agar kosong, sedangkan fungsi isEmpty memeriksa apakah pohon tersebut kosong. Fungsi buatNode membuat node baru sebagai akar pohon jika pohon masih kosong. Selanjutnya, terdapat fungsi insertLeft dan insertRight untuk menambahkan node baru sebagai anak kiri atau kanan dari node tertentu, dengan pengecekan untuk memastikan bahwa anak tersebut belum ada. Program juga menyediakan fungsi untuk memperbarui data node, mencari node berdasarkan data, serta melakukan traversal pohon dengan metode pre-order, in-order, dan post-order. Selain itu, terdapat fungsi untuk menghapus node tertentu dan menemukan node paling kiri dan kanan dalam pohon. Di dalam fungsi main, pohon diinisialisasi, beberapa node ditambahkan, dan dilakukan traversal untuk menampilkan struktur pohon. Program juga menunjukkan cara menghapus node dan menampilkan hasil traversal setelah penghapusan.

Kode program:

```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);
}
```

```cpp
// 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;
}
```

Hasil output dari program:

```
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
PS D:\tugas yall\praktikum sd\pertemuan 10\output>
```

4. Unguided

Program ini menggunakan struktur data tree untuk menyimpan node yang dibuat agar user yang menginputkannya sendiri, memiliki anak kiri dan kanan serta mendukung operasi tambahan sesuai dengan ketiga soal yang diberikan, seperti mencari node, memeriksa properti Binary Search Tree (BST), dan menghitung jumlah simpul daun.

Node tree berisi data (huruf atau karakter) dan tiga pointer:
left → menunjuk ke anak kiri.
right → menunjuk ke anak kanan.
parent → menunjuk ke induk dari node.
Fungsi pada program yang awalnya sudah ada seperti mengecek apakah tree kosong, membuat node baru sebagai root, serta menambahkan node baru sebagai anak kiri atau anak kanan dari node tertentu.
Fungsi tambahan program seperti soal pertama yang menyuruh untuk menampilkan anak dan descendant dari node tertentu. Kemudian pada soal kedua dibuat pencarian node berdasarkan data secara rekrusif dan juga dilakukan uji fungsi BST pada tree. Soal ketiga membuat fungsi untuk menghitung jumlah simpul daun.

Pada fungsi utama dibuat menu program, untuk membuat menu itu berjalan digunakan switch case yang memberikan menu interaktif kepada pengguna agar mereka dapat memilih aksi tertentu. Setiap case mewakili pilihan menu yang akan dijalankan.

- Case 1: Buat Root
  Membuat node pertama (root) dari pohon. Jika root sudah ada, program akan memberi peringatan bahwa pohon sudah dibuat.

- Case 2: Tambah Child Kiri
  Menambahkan node baru sebagai anak kiri dari node yang ditentukan oleh pengguna. Menggunakan fungsi findNode untuk mencari node induk.
- Case 3: Tambah Child Kanan
  Menambahkan node baru sebagai anak kanan dari node yang ditentukan oleh pengguna. Sama seperti Case 2, menggunakan findNode untuk mencari node induk.
- Case 4: Tampilkan Child Node
  Menampilkan anak kiri dan anak kanan dari node tertentu. Jika node tidak memiliki anak, program akan memberi keterangan bahwa anak tersebut kosong.
- Case 5: Tampilkan Descendant Node
  Menampilkan semua descendant (keturunan) dari node yang diinputkan, termasuk anak, cucu, cicit, dan seterusnya. Descendant dicetak menggunakan traversal Pre-order mulai dari anak kiri ke kanan.
- Case 6: Periksa Apakah Pohon Adalah BST
  Memeriksa apakah pohon memenuhi properti Binary Search Tree (BST). Terdapat syarat BST untuk setiap node yaitu, semua node di anak kiri memiliki nilai lebih kecil dan semua node di anak kanan memiliki nilai lebih besar.
- Case 7: Hitung Jumlah Simpul Daun
  Menghitung jumlah simpul daun dalam pohon biner. Simpul daun adalah node yang tidak memiliki anak kiri maupun kanan.
- Case 0: Keluar
  Program akan keluar dari menu.

```cpp
#include <iostream>
#include <limits> // Untuk nilai batas min dan max
using namespace std;

// Struktur data pohon biner
struct Pohon {
    char data;
    Pohon *left, *right;
    Pohon *parent;
};

Pohon *root, *baru;

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

// Mengecek apakah pohon kosong
bool isEmpty() {
    return root == NULL;
}

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

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

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

// 2 Modifikasi mencari node berdasarkan data secara rekrusif
Pohon* findNode(char data, Pohon *node) {
    if (!node) return NULL; // Basis rekursif: jika node NULL, kembalikan NULL

    if (node->data == data) // Jika data cocok, kembalikan node
        return node;

    // Cari ke anak kiri dan kanan secara rekursif
    Pohon *leftResult = findNode(data, node->left);
    if (leftResult) return leftResult; // Jika ditemukan di anak kiri, kembalikan hasil

    Pohon *rightResult = findNode(data, node->right);
    return rightResult; // Jika ditemukan di anak kanan, kembalikan hasil
}
```

```cpp
// 1 Modifikasi menampilkan child dari node tertentu
void tampilChild(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan.\n";
        return;
    }
    cout << "Node: " << node->data << "\n";
    if (node->left) cout << "Child Kiri: " << node->left->data << "\n";
    else cout << "Child Kiri: NULL\n";
    if (node->right) cout << "Child Kanan: " << node->right->data << "\n";
    else cout << "Child Kanan: NULL\n";
}

// 1 Modifikasi menampilkan descendant dari node tertentu (rekursif)
void tampilDescendant(Pohon *node) {
    if (!node) return;
    if (node->left || node->right)
        cout << node->data << " memiliki descendant: ";
    if (node->left) cout << node->left->data << " ";
    if (node->right) cout << node->right->data << " ";
    cout << "\n";
    tampilDescendant(node->left);
    tampilDescendant(node->right);
}

// 2 Modifikasi menambahkan fungsi validasi apakah pohon adalah 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);
}

// 3 Modifikasi menghitung jumlah simpul daun
int cari_simpul_daun(Pohon *node) {
    if (!node) return 0; // Basis rekursi
    if (!node->left && !node->right) return 1; // Node daun
    return cari_simpul_daun(node->left) + cari_simpul_daun(node->right);
}

// Fungsi utama dengan menu
int main()
{
    int pilihan;
    char data, parent;
    Pohon *temp = NULL;
    init();

    // 1 Modifikasi membuat menu program agar user dapat menginputkan sendiri
    do {
        cout << "\n=== MENU POHON BINARY TREE ===\n";
        cout << "1. Buat Root\n";
        cout << "2. Tambah Child Kiri\n";
        cout << "3. Tambah Child Kanan\n";
        cout << "4. Tampilkan Child Node\n";
        cout << "5. Tampilkan Descendant Node\n";
        cout << "6. Cek Apakah Pohon adalah BST\n";
        cout << "7. Hitung Jumlah Simpul Daun\n";
        cout << "0. 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 anak kiri: ";
                cin >> data;
                temp = findNode(parent, root); // Cari node berdasarkan data
                if (temp) {
                    insertLeft(data, temp);
                    } else {
                        cout << "Node dengan data " << parent << " tidak ditemukan!\n";
                    }
                break;
            case 3:
                cout << "Masukkan data parent: ";
                cin >> parent;
                cout << "Masukkan data anak kanan: ";
                cin >> data;
                temp = findNode(parent, root); // Cari node berdasarkan data
                if (temp) {
                    insertRight(data, temp);
                } else {
                    cout << "Node dengan data " << parent << " tidak ditemukan!\n";
                }
                break;
            case 4:
                cout << "Masukkan node yang ingin ditampilkan child-nya: ";
                cin >> data;
                temp = findNode(data, root); // Cari node berdasarkan data
                if (temp) {
                    tampilChild(temp);
                } else {
                    cout << "Node dengan data " << data << " tidak ditemukan!\n";
                }
                break;
            case 5:
                cout << "Masukkan node yang ingin ditampilkan descendant-nya: ";
                cin >> data;
                tampilDescendant(findNode(data, root));
                break;
            case 6:
                if (is_valid_bst(root, numeric_limits<char>::min(), numeric_limits<char>::max()))
                    cout << "Pohon adalah BST yang valid.\n";
                else
                    cout << "Pohon bukan BST yang valid.\n";
                break;
            case 7:
                cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << "\n";
                break;
            case 0:
                cout << "Keluar...\n";
                break;
            default:
                cout << "Pilihan tidak valid!\n";
        }
    } while (pilihan != 0);

    return 0;
}

```

Hasil output dari program akan menjadi seperti berikut:

```
=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 1
Masukkan data root: 7
Node 7 berhasil dibuat sebagai root.

=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 2
Masukkan data parent: 7
Masukkan data anak kiri: 5
Node 5 ditambahkan ke kiri dari 7
```

```
=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 3
Masukkan data parent: 7
Masukkan data anak kanan: 6
Node 6 ditambahkan ke kanan dari 7

=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 4
Masukkan node yang ingin ditampilkan child-nya: 7
Node: 7
Child Kiri: 5
Child Kanan: 6
```

```
=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 5
Masukkan node yang ingin ditampilkan descendant-nya: 7
7 memiliki descendant: 5 6



=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 6
Pohon bukan BST yang valid.
```

```
=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 7
Jumlah simpul daun: 2

=== MENU POHON BINARY TREE ===
1. Buat Root
2. Tambah Child Kiri
3. Tambah Child Kanan
4. Tampilkan Child Node
5. Tampilkan Descendant Node
6. Cek Apakah Pohon adalah BST
7. Hitung Jumlah Simpul Daun
0. Keluar
Pilihan: 0
Keluar...
```

5. Kesimpulan

Pada laporan ini menyajikan implementasi struktur data Binary Tree menggunakan konsep rekursi. Pembuatan program yang dapat melakukan berbagai operasi pada pohon biner, mulai dari membuat pohon baru, menambahkan atau menghapus simpul, hingga melakukan penelusuran pohon dengan berbagai metode (pre-order, in-order, post-order). Selain itu, pada guided dibuat modifikasi program yang dilengkapi fitur tambahan seperti menampilkan keturunan suatu simpul, memeriksa apakah pohon memenuhi syarat sebagai pohon pencarian biner (Binary Search Tree), serta menghitung jumlah daun pada pohon. Melalui praktikum ini, pemahaman mengenai penerapan rekursi dan struktur pohon biner dalam menyelesaikan masalah pemrograman yang bersifat hierarkis semakin diperdalam.