LAPORAN PRAKTIKUM

MODUL 10

TREE (BAGIAN PERTAMA)

Disusun Oleh:

Rizaldy Aulia Rachman (2311104051)

S1SE-07-02

Dosen :

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

PROGRAM STUDI S1 SOFTWARE ENGINEERING

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY PURWOKERTO

2024

I. 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*.

II. LANDASAN 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

Dengan melihat sifat sub program rekursif di atas maka sub program rekursif harus memiliki :

1. Kondisi yang menyebabkan pemanggilan dirinya berhenti (disebut kondisi khusus atau special condition)
2. Pemanggilan diri sub program (yaitu bila kondisi khusus tidak dipenuhi)

Secara umum bentuk dari sub program rekursif memiliki statemen kondisional :

- if kondisi khusus tak dipenuhi
- then panggil diri-sendiri dengan parameter yang sesuai
- else lakukan instruksi yang akan dieksekusi bila kondisi khusus dipenuhi

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.
Kendati demikian, penggunaan rekursif memiliki beberapa kekurangan antara lain:

1. Memerlukan memori yang lebih banyak untuk menyimpan *activation record* dan variabel lokal. *Activation record* diperlukan waktu proses kembali kepada pemanggil
2. Memerlukan waktu yang lebih banyak untuk menangani *activation record*.

Secara umum gunakan penyelesaian rekursif hanya jika:

- Penyelesaian sulit dilaksanakan secara iteratif.
- Efisiensi dengan cara rekursif sudah memadai.
- Efisiensi bukan masalah dibandingkan dengan kejelasan logika program.

2.4 Pengertian Tree

Kita telah mengenal dan mempelajari jenis-jenis strukur data yang *linear*, seperti : *list*, *stack* dan *queue*. Adapun jenis struktur data yang kita pelajari kali ini adalah struktur data yang non-liniar (*nonlinear data structure*) yang disebut *tree*.

Tree digambarkan sebagai suatu *graph* tak berarah terhubung dan tidak terhubung dan tidak mengandung sirkuit.

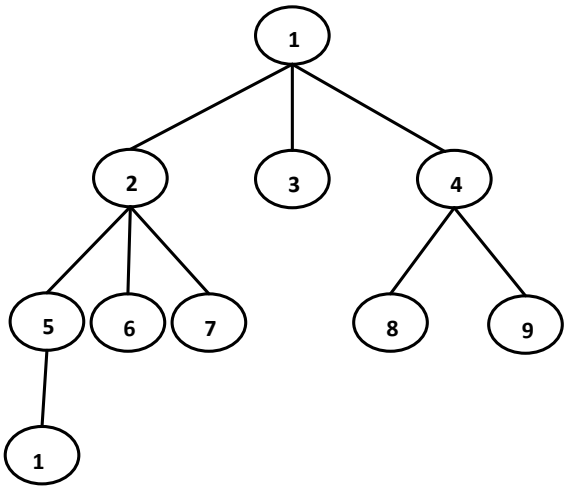
Karateristik dari suatu *tree* T adalah :

1. T kosong berarti *empty tree*
2. Hanya terdapat satu *node* tanpa pendahulu, disebut akar (*root*)
3. Semua *node* lainnya hanya mempunyai satu *node* pendahulu.

2.5 Jenis-jenis Tree

1. **Ordered Tree**

Yaitu pohon yang urutan anak-anaknya penting.



Gambar 10-2 *Ordered Tree*

2. Binary Tree

Setiap *node* di *Binary Tree* hanya dapat mempunyai maksimum 2 *children* tanpa pengecualian. *Level* dari suatu *tree* dapat menunjukan berapa kemungkinan jumlah *maximum nodes* yang terdapat pada *tree* tersebut. Misalnya, *level tree* adalah r, maka *node* maksimum yang mungkin adalah 2^r.

A. Complete Binary Tree

Suatu *binary tree* dapat dikatakan lengkap (*complete*), jika pada setiap level yang mempunyai jumlah maksimum dari kemungkinan *node* yang dapat dipunyai, dengan pengecualian *node* terakhir. Complete *tree* T_n yang unik memiliki n *nodes*. Untuk menentukan jumlah *left children* dan *right children tree* T_n di *node* K dapat dilakukan dengan cara:

1. Menentukan left *children*: 2*K
2. Menentukan right *children*: 2 *(K + 1)
3. Menentukan *parent*: [K/2] **B. *Extended Binary Tree***

Suatu *binary tree* yang terdiri atas *tree* T yang masing-masing *node*-nya terdiri dari tepat 0 atau 2 *children* disebut 2-*tree* atau ***extended binary* tree.** Jika setiap *node* N mempunyai 0 atau 2 *children* disebut internal *nodes* dan *node* dengan 0 *children* disebut *external nodes.*

C. Binary Search Tree

Binary search tree adalah *Binary tree* yang terurut dengan ketentuan:

1. Semua **LEFTCHILD** harus lebih kecil dari *parent*-nya.
2. Semua **RIGHTCHILD** harus lebih besar dari parentnya dan *leftchild*-nya.

D. AVL *Tree*

Adalah *binary search tree* yang mempunyai ketentuan bahwa *maximum* perbedaan *height* antara *subtree* kiri dan *subtree* kanan adalah 1.

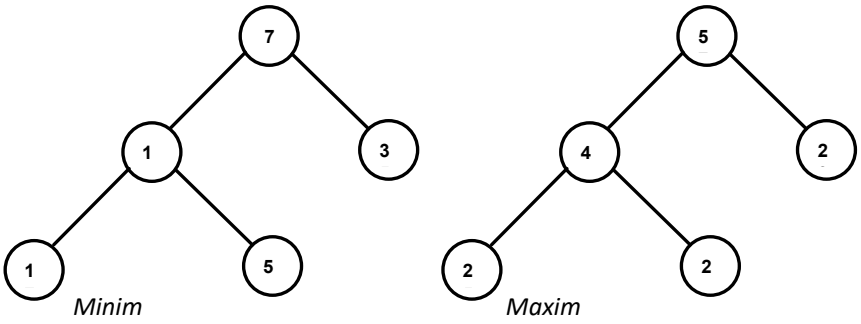
E. Heap Tree

Adalah *tree* yang memenuhi persamaan berikut: R[i] < r[2i] and R[i] < r[2i+1]

Heap juga disebut *Complete Binary Tree*, karena jika suatu *node* mempunyai *child*, maka jumlah *child*nya harus selalu dua.

Minimum Heap: jika *parent*-nya selalu lebih kecil daripada kedua *children*-nya.

Maximum Heap : jika *parent*-nya selalu lebih besar daripada kedua *children*-nya.



Gambar 10-3 *Heap Tree*

III. GUIDED

1. Guided1

Code:

```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);
}

// 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; // Tukar 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
PS C:\Praktikum Struktur data\pertemuan9>
```

IV. 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.

Jawaban:

Code:

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

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

Pohon *root = NULL;

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

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

// Menambahkan node ke kiri
Pohon* insertLeft(char data, Pohon *node) {
    if (!node) {
        cout << "Node parent tidak ditemukan." << endl;
        return NULL;
    }
    if (node->left != NULL) {
        cout << "Node " << node->data << " sudah ada anak kiri." << endl;
        return NULL;
    }
    node->left = new Pohon{data, NULL, NULL, node};
    cout << "Node " << data << " berhasil ditambahkan ke kiri " << node->data << endl;
    return node->left;
}

// Menambahkan node ke kanan
Pohon* insertRight(char data, Pohon *node) {
    if (!node) {
        cout << "Node parent tidak ditemukan." << endl;
        return NULL;
    }
    if (node->right != NULL) {
        cout << "Node " << node->data << " sudah ada anak kanan." << endl;
        return NULL;
    }
    node->right = new Pohon{data, NULL, NULL, node};
    cout << "Node " << data << " berhasil ditambahkan ke kanan " << node->data << endl;
    return node->right;
}

// Mencari node berdasarkan data
Pohon* find(char data, Pohon *node) {
    if (!node) return NULL;
    if (node->data == data) return node;
    Pohon *leftResult = find(data, node->left);
    if (leftResult) return leftResult;
    return find(data, node->right);
}

// Menampilkan anak langsung
void tampilkanChild(Pohon *node) {
    if (!node) {
        cout << "Node tidak ditemukan." << endl;
        return;
    }
    cout << "Node " << node->data << " memiliki:" << endl;
    if (node->left) cout << "  Anak kiri: " << node->left->data << endl;
    if (node->right) cout << "  Anak kanan: " << node->right->data << endl;
    if (!node->left && !node->right) cout << "  Tidak ada anak langsung." << endl;
}

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

// Memeriksa 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);
}

// 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);
}

// Menu utama
void menu() {
    char pilihan, data, parent_data;
    Pohon *parent_node;

    do {
        cout << "\nMenu:\n";
        cout << "1. Buat Root\n";
        cout << "2. Tambah Anak Kiri\n";
        cout << "3. Tambah Anak Kanan\n";
        cout << "4. Tampilkan Anak\n";
        cout << "5. Tampilkan Descendant\n";
        cout << "6. Periksa BST\n";
        cout << "7. Hitung Simpul Daun\n";
        cout << "0. Keluar\n";
        cout << "Pilih: ";
        cin >> pilihan;

        switch (pilihan) {
        case '1':
            cout << "Masukkan data root: ";
            cin >> data;
            buatNode(data);
            break;
        case '2':
            cout << "Masukkan data parent: ";
            cin >> parent_data;
            cout << "Masukkan data anak kiri: ";
            cin >> data;
            parent_node = find(parent_data, root);
            if (parent_node) insertLeft(data, parent_node);
            else cout << "Node parent tidak ditemukan.\n";
            break;
        case '3':
            cout << "Masukkan data parent: ";
            cin >> parent_data;
            cout << "Masukkan data anak kanan: ";
            cin >> data;
            parent_node = find(parent_data, root);
            if (parent_node) insertRight(data, parent_node);
            else cout << "Node parent tidak ditemukan.\n";
            break;
        case '4':
            cout << "Masukkan data node: ";
            cin >> data;
            parent_node = find(data, root);
            tampilkanChild(parent_node);
            break;
        case '5':
            cout << "Masukkan data node: ";
            cin >> data;
            parent_node = find(data, root);
            if (parent_node) {
                cout << "Descendant dari " << data << ": ";
                tampilkanDescendant(parent_node);
                cout << endl;
            } else cout << "Node tidak ditemukan.\n";
            break;
        case '6':
            cout << (is_valid_bst(root, 'A', 'Z') ? "Pohon adalah BST.\n" : "Pohon bukan BST.\n");
            break;
        case '7':
            cout << "Jumlah simpul daun: " << cari_simpul_daun(root) << endl;
            break;
        case '0':
            cout << "Keluar.\n";
            break;
        default:
            cout << "Pilihan tidak valid.\n";
        }
    } while (pilihan != '0');
}

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

Output:

```
Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 1
Masukkan data root: F
Node F berhasil dibuat menjadi root.

Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 2
Masukkan data parent: F
Masukkan data anak kiri: B
Node B berhasil ditambahkan ke kiri F
```

```
Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 3
Masukkan data parent: F
Masukkan data anak kanan: G
Node G berhasil ditambahkan ke kanan F

Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 4
Masukkan data node: F
Node F memiliki:
  Anak kiri: B
  Anak kanan: G
```

```
Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 5
Masukkan data node: F
Descendant dari F: F B G

Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 6
Pohon adalah BST.
```

```
Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 7
Jumlah simpul daun: 2

Menu:
1. Buat Root
2. Tambah Anak Kiri
3. Tambah Anak Kanan
4. Tampilkan Anak
5. Tampilkan Descendant
6. Periksa BST
7. Hitung Simpul Daun
0. Keluar
Pilih: 0
Keluar.
```