Telkom University Purwokerto

Aturan Praktikum Struktur Data

1. **Akun GitHub:** Setiap praktikan wajib memiliki akun GitHubyang aktif dan digunakan selama praktikum berlangsung.
2. **Invite Collaborator:** Setiap praktikan diwajibkan untuk menambahkan collaborator di setiap repository
    a. Asisten Praktikum: AndiniNH
    b. AsistenPraktikum:4ldiputra
3. **Repository Praktikum:** Setiap praktikan diwajibkan untuk membuat satu repository di GitHub yang akan digunakan untuk seluruh tugas dan laporan praktikum. Repository ini harus diatur dengan rapi dan sesuai dengan instruksi yang akan diberikan di lampiran.
4. **Penamaan Folder:** Penamaan folderdalam repositoryakan dibahas secara rinci di lampiran. Praktikan wajib mengikuti aturan penamaan yang telah ditentukan.

Nomor	Pertemuan	Penamaan
1	Penganalan Bahasa C++ Bagian Pertama	01_Pengenalan_CPP_Bagian_1
2	Pengenalan Bahasa C++ Bagian Kedua	02_Pengenalan_CPP_Bagian_2
3	Abstract Data Type	03_Abstract_Data_Type
4	Single Linked List Bagian Pertama	04_Single_Linked_List_Bagian_1
5	Single Linked List Bagian Kedua	05_Single_Linked_List_Bagian_2
6	Double Linked List Bagian Pertama	06_Double_Linked_List_Bagian_1
7	Stack	07_Stack
8	Queue	08_Queue
9	Assessment Bagian Pertama	09_Assessment_Bagian_1
10	Tree Bagian Pertama	10_Tree_Bagian_1
11	Tree Bagian Kedua	11_Tree_Bagian_2
12	Asistensi Tugas Besar	12_Asistensi_Tugas_Besar
13	Multi Linked List	13_Multi_Linked_List
14	Graph	14_Graph
15	Assessment Bagian Kedua	15_Assessment_Bagian_2
16	Tugas Besar	16_Tugas_Besar

5. **Jam Praktikum:**
    - Jam masuk praktikum adalah **1 jam lebih lambat** dari jadwal yang tercantum. Sebagai contoh, jika jadwalpraktikumadalah pukul 06.30 - 09.30, maka aturan praktikum akan diatur sebagai berikut:
        - **06.30 - 07.30**: Waktuini digunakan untuk **Tugas Praktikum dan Laporan Praktikum** yang dilakukan di luar laboratorium.
        - **07.30 - 0G.30**: Sesi ini mencakup **tutorial, diskusi, dan kasus problem- solving**. Kegiatan ini berlangsung di dalam laboratorium dengan alokasi waktu sebagai berikut:
            - **60 menit pertama**: Tugas terbimbing.
            - **60 menit kedua**: Tugas mandiri.
6. **Pengumpulan Tugasn Pendahuluan:** Tugas Pendahuluan (TP) wajib dikumpulkan melalui GitHub sesuai dengan format berikut:

    nama_repo/nama_pertemuan/TP_Pertemuan_Ke.md

    Sebagai contoh:

    STD_Yudha_Islalmi_Sulistya_XXXXXXXX/01_Running_Modul/TP_01.md

7. **Pengecekan Tugas Pendahuluan:** Pengumpulan laporan praktikum akan diperiksa **1 hari sebelum praktikum selanjutnya** dimulai. Pastikan tugas telah diunggah tepat waktu untuk menghindari sanksi.

LAPORAN PRAKTIKUM
MODUL 10 & 11
TREE 1 & 2

Disusun Oleh :

Zaenarif Putra ‘Ainurdin – 2311104049

Kelas :

SE-07-02

Dosen :

Wahyu Andi Saputra, S.pd,M.Eng

PROGRAM STUDI 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.
5. Mengimplementasikan struktur data tree, khususnya Binary Tree.

II. LANDASAN TEORI

Struktur data tree, khususnya Binary Search Tree (BST), adalah representasi hierarkis yang terdiri dari node yang terhubung, di mana setiap node memiliki maksimum dua anak (left dan right). BST memiliki sifat unik di mana nilai di subtree kiri selalu lebih kecil dari nilai node induk, dan nilai di subtree kanan selalu lebih besar. Ini memungkinkan pencarian, penyisipan, dan penghapusan elemen dilakukan dengan efisien, biasanya dalam waktu O(log n) pada tree yang seimbang.

Rekursi adalah metode pemrograman di mana fungsi memanggil dirinya sendiri untuk menyelesaikan sub-masalah yang lebih kecil. Dalam konteks BST, rekursi sering digunakan untuk melakukan operasi seperti penyisipan (insert), pencarian (search), dan traversal (pre-order, in-order, post-order). Setiap fungsi rekursif harus memiliki kondisi pemberhentian untuk mencegah pemanggilan tak terbatas, serta pemanggilan diri dengan parameter yang dimodifikasi untuk mendekati kondisi pemberhentian.

Meskipun rekursi dapat menyederhanakan kode dan meningkatkan kejelasan, penggunaannya juga memiliki kekurangan, seperti penggunaan memori yang lebih tinggi dan waktu eksekusi yang lebih lama dibandingkan dengan pendekatan iteratif. Traversal pada BST memungkinkan kita untuk mengunjungi setiap node dalam urutan tertentu, yang berguna untuk berbagai aplikasi, termasuk penghitungan jumlah node, total informasi, dan kedalaman tree. Dengan memahami konsep-konsep ini, kita dapat lebih efektif dalam mengimplementasikan dan memanfaatkan struktur data tree dan teknik rekursif dalam pemrograman.v

III. GUIDE

1. Guide

   a. Syntax

```
#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;
}

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 == NULL) 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;
}
```

b. Penjelasan syntax

- Struktur Pohon digunakan untuk merepresentasikan sebuah node dalam pohon biner. Setiap node memiliki atribut data untuk menyimpan nilai, serta pointer left dan right untuk menunjuk ke anak kiri dan anak kanan. Pointer parent menyimpan referensi ke node induk, yang berguna untuk navigasi dalam pohon.
- Inisialisasi dan Pengecekan Pohon Fungsi init digunakan untuk menginisialisasi pohon dengan mengatur root menjadi NULL, menandakan bahwa pohon dalam keadaan kosong. Fungsi isEmpty mengecek apakah pohon kosong dengan memeriksa apakah root bernilai NULL.
- Membuat Node Root Fungsi buatNode membuat node pertama dalam pohon, yang menjadi root. Jika root sudah ada, fungsi ini tidak akan menambahkan node baru untuk mencegah duplikasi.
- Menambahkan Node Anak insertLeft: Menambahkan node baru sebagai anak kiri dari node tertentu. insertRight: Menambahkan node baru sebagai anak kanan dari node tertentu. Kedua fungsi ini memastikan tidak ada anak ganda di posisi kiri atau kanan sebelum menambahkan node baru.
- Mengubah Nilai Node Fungsi update digunakan untuk mengubah nilai data pada sebuah node tanpa mengubah struktur pohon. Hal ini berguna jika nilai data perlu diperbarui.
- Traversal Pohon Pre-order (preOrder): Menampilkan node dalam urutan root → anak kiri → anak kanan. Berguna untuk menyalin struktur pohon. In-order (inOrder): Menampilkan node dalam urutan anak kiri → root → anak kanan. Traversal ini menghasilkan urutan elemen yang terurut pada pohon biner pencarian (BST). Post-order (postOrder): Menampilkan node dalam urutan anak kiri → anak kanan → root. Traversal ini berguna untuk menghapus atau membongkar pohon secara berurutan.
- Fungsi find mencari node berdasarkan nilai tertentu (data). Jika ditemukan, program akan menampilkan node yang dimaksud. Fungsi ini penting untuk operasi yang memerlukan akses ke node tertentu.
- Node Paling Kiri dan Kanan mostLeft: Menemukan node paling kiri, yaitu node dengan nilai terkecil dalam pohon.
- mostRight: Menemukan node paling kanan, yaitu node dengan nilai terbesar dalam pohon. Fungsi ini berguna untuk mencari elemen ekstrem pada pohon biner.
- Fungsi deleteNode menghapus node berdasarkan nilai tertentu (data).

Jika node memiliki anak, algoritma ini mengganti node yang dihapus dengan nilai dari node pengganti yang paling dekat (successor atau predecessor). Fungsi ini memastikan struktur pohon tetap valid setelah penghapusan.

- Dalam fungsi main, program akan menginisialisasi pohon dengan root 'F'. Menambahkan node-node baru sebagai anak kiri dan kanan. Melakukan traversal (Pre-order, In-order, Post-order) untuk menampilkan isi pohon. Menemukan node dengan nilai terkecil dan terbesar. Menghapus sebuah node ('D') dari pohon, dan memastikan struktur pohon tetap konsisten setelah penghapusan.

c. Output

```
PS C:\Users\LENOVO\OneDrive - Telkom University\Documents\ALL Matkul\StrukturData\peretmuan9\Guide\output>

Node F berhasil dibuat menjadi root.

Node B berhasil ditambahkan ke child kiriF

Node G berhasil ditambahkan ke child kanan F

Node A berhasil ditambahkan ke child kiriB

Node D berhasil ditambahkan ke child kanan B

Node C berhasil ditambahkan ke child kiriD

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:\Users\LENOVO\OneDrive - Telkom University\Documents\ALL Matkul\StrukturData\peretmuan9\Guide\output>
```

IV. UNGUIDED

1. Unguided 1,2,3

a. Syntax :

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

struct Pohon {
    char data; // Untuk pohon biner
    int intData; // Untuk pohon BST
    Pohon *left, *right, *parent;
};

// Variabel global untuk pohon biner
Pohon *root = NULL;

// Fungsi untuk inisialisasi pohon biner
void init() {
    root = NULL;
}
```

```
// Fungsi untuk memeriksa apakah pohon biner kosong
bool isEmpty() {
    return root == NULL;
}

// Fungsi untuk membuat node baru sebagai root
void buatNode(char data) {
    if (isEmpty()) {
        root = new Pohon{data, 0, NULL, NULL, NULL};
        cout << "Node " << data << " berhasil dibuat sebagai root.\n";
    } else {
        cout << "Root sudah ada.\n";
    }
}

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

// Fungsi untuk menambahkan anak kiri
Pohon* insertLeft(char data, Pohon *node) {
    if (node->left != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kiri.\n";
        return NULL;
    }
    Pohon *baru = new Pohon{data, 0, NULL, NULL, node};
    node->left = baru;
    cout << "Node " << data << " berhasil ditambahkan ke anak kiri " <<
node->data << ".\n";
    return baru;
}

// Fungsi untuk menambahkan anak kanan
Pohon* insertRight(char data, Pohon *node) {
    if (node->right != NULL) {
        cout << "Node " << node->data << " sudah memiliki anak kanan.\n";
        return NULL;
    }
    Pohon *baru = new Pohon{data, 0, NULL, NULL, node};
    node->right = baru;
    cout << "Node " << data << " berhasil ditambahkan ke anak kanan "
<< node->data << ".\n";
    return baru;
}
```

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

// Fungsi untuk menampilkan keturunan dari node tertentu
void tampilkanDescendants(Pohon *node) {
    if (!node) return;
    if (node->left || node->right) cout << node->data << ": ";
    if (node->left) cout << node->left->data << " ";
    if (node->right) cout << node->right->data << " ";
    cout << "\n";
    tampilkanDescendants(node->left);
    tampilkanDescendants(node->right);
}

// Fungsi untuk memeriksa validitas BST
bool is_valid_bst(Pohon* node, int min_val, int max_val) {
    if (!node) return true;
    if (node->intData <= min_val || node->intData >= max_val)
        return false;
    return is_valid_bst(node->left, min_val, node->intData) &&
           is_valid_bst(node->right, node->intData, max_val);
}

// Fungsi 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);
}

// Menu untuk interaksi pengguna
void menu() {
    int pilihan;
    char data, parent_data;
    Pohon *parent_node;

    do {
        cout << "\nMenu Binary Tree:";
        cout << "\n1. Add Root";
```

```
        cout << "\n2. Tambahkan Anak Kiri";
        cout << "\n3. Tambahkan Anak Kanan";
        cout << "\n4. Tampilkan Child";
        cout << "\n5. Tampilkan Descendants";
        cout << "\n6. Cek Validitas BST";
        cout << "\n7. Hitung Jumlah Simpul Daun";
        cout << "\n8. Keluar";
        cout << "\nPilih: ";
        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 = findNode(root, parent_data);
                if (parent_node) insertLeft(data, parent_node);
                else cout << "Node " << parent_data << " tidak ditemukan.\n";
                break;

            case 3:
                cout << "Masukkan data parent: ";
                cin >> parent_data;
                cout << "Masukkan data anak kanan: ";
                cin >> data;
                parent_node = findNode(root, parent_data);
                if (parent_node) insertRight(data, parent_node);
                else cout << "Node " << parent_data << " tidak ditemukan.\n";
                break;

            case 4:
                cout << "Masukkan data node: ";
                cin >> data;
                parent_node = findNode(root, data);
                tampilkanChild(parent_node);
                break;

            case 5:
                cout << "Masukkan data node: ";
                cin >> data;
                parent_node = findNode(root, data);
                tampilkanDescendants(parent_node);
                break;
```

```
            case 6:
                cout << "Masukkan data node: ";
                cin >> data;
                parent_node = findNode(root, data);
                  cout << "Pohon valid BST: " << (is_valid_bst(parent_node,
INT_MIN, INT_MAX) ? "Ya" : "Tidak") << endl;
                break;

            case 7:
                cout << "Jumlah simpul daun: " << cari_simpul_daun(root) <<
endl;
                break;

            case 8:
                cout << "Keluar.\n";
                break;

            default:
                cout << "Pilihan tidak valid.\n";
        }
    } while (pilihan != 8);
}

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

b. Penjelasan Syntax

- Deklarasi Struktur Pohon: Struktur Pohon digunakan untuk merepresentasikan sebuah node dalam pohon biner atau pohon BST. Setiap node memiliki data berupa karakter (data) untuk pohon biner, atau integer (intData) untuk pohon BST. Selain itu, setiap node memiliki pointer ke anak kiri (left), anak kanan (right), dan parent (parent).
- Inisialisasi dan Cek Kekosongan Pohon: Fungsi init() digunakan untuk menginisialisasi pohon dengan menjadikan root sebagai NULL, yang menunjukkan bahwa pohon masih kosong. Fungsi isEmpty() memeriksa apakah pohon kosong dengan memeriksa apakah root adalah NULL.
- Membuat Node sebagai Root: Fungsi buatNode(data) akan membuat node baru dan menjadikannya sebagai root jika pohon masih kosong. Jika root sudah ada, maka fungsi akan memberi pesan bahwa root sudah ada.
- Menambahkan Anak Kiri atau Kanan: Fungsi insertLeft(data, node) menambahkan sebuah node baru sebagai anak kiri dari node yang diberikan, jika anak kiri belum ada. Demikian pula, fungsi insertRight(data, node) menambahkan sebuah node baru sebagai anak

kanan jika anak kanan belum ada.

- Menampilkan Anak dari Node Tertentu: Fungsi tampilkanChild(node) digunakan untuk menampilkan anak kiri dan kanan dari node yang diberikan. Jika anak kiri atau kanan tidak ada, maka akan ditampilkan sebagai NULL.
- Menampilkan Semua Keturunan (Descendants): Fungsi tampilkanDescendants(node) akan menampilkan seluruh keturunan dari node yang diberikan, termasuk anak, cucu, dan seterusnya. Fungsi ini akan menampilkan data anak dari setiap node yang ditemukan dalam pohon.
- Memeriksa Validitas BST: Fungsi is_valid_bst(node, min_val, max_val) digunakan untuk memeriksa apakah pohon tersebut adalah pohon pencarian biner (BST) yang valid. Fungsi ini memastikan bahwa data dalam setiap node berada dalam rentang yang sesuai, yakni lebih besar dari nilai minimum yang ditentukan dan lebih kecil dari nilai maksimum yang ditentukan.
- Menghitung Jumlah Simpul Daun: Fungsi cari_simpul_daun(node) menghitung jumlah simpul daun dalam pohon, yaitu node yang tidak memiliki anak kiri maupun kanan. Fungsi ini akan mengembalikan jumlah simpul daun dalam pohon biner.
- Menu Interaktif Pengguna: Fungsi menu() menyediakan menu interaktif bagi pengguna untuk memilih berbagai operasi seperti menambahkan node root, menambahkan anak kiri/kanan, menampilkan anak atau keturunan, memeriksa validitas BST, dan menghitung jumlah simpul daun. Program akan terus berjalan hingga pengguna memilih untuk keluar.
- Fungsi main() menginisialisasi pohon dengan memanggil fungsi init(), lalu memulai menu interaktif dengan memanggil fungsi menu(). Program ini akan terus berjalan hingga pengguna memilih untuk keluar dari menu.

c. Output

```
Menu Binary Tree:
1. Add Root
2. Tambahkan Anak Kiri
3. Tambahkan Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Cek Validitas BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 1
Masukkan data root: 15
Node 1 berhasil dibuat sebagai root.
```

```
Menu Binary Tree:
1. Add Root
2. Tambahkan Anak Kiri
3. Tambahkan Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Cek Validitas BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 2
Masukkan data parent: 15
Masukkan data anak kiri: Node 5 berhasil ditambahkan ke anak kiri 1.
```

```
Menu Binary Tree:
1. Add Root
2. Tambahkan Anak Kiri
3. Tambahkan Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Cek Validitas BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 3
Masukkan data parent: 15
Masukkan data anak kanan: Node 5 berhasil ditambahkan ke anak kanan 1.
```

```
Menu Binary Tree:
1. Add Root
2. Tambahkan Anak Kiri
3. Tambahkan Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Cek Validitas BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 4
Masukkan data node: 15
Child dari 1:
Kiri: 5
Kanan: 5
```

```
Menu Binary Tree:
1. Add Root
2. Tambahkan Anak Kiri
3. Tambahkan Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Cek Validitas BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 6
Masukkan data node: 15
Pohon valid BST: Tidak
```

```
Menu Binary Tree:
1. Add Root
2. Tambahkan Anak Kiri
3. Tambahkan Anak Kanan
4. Tampilkan Child
5. Tampilkan Descendants
6. Cek Validitas BST
7. Hitung Jumlah Simpul Daun
8. Keluar
Pilih: 7
Jumlah simpul daun: 2
```

V. KESIMPULAN

Tujuan praktik 10 dan 11 tentang struktur data pohon adalah untuk mendapatkan pemahaman dan penerapan konsep dasar pohon, khususnya Binary Search Tree (BST), menggunakan pendekatan rekursif. Modul-modul ini mencakup operasi dasar seperti menambahkan, menghapus, dan melintasi (pre-order, in-order, dan post-order), serta konsep lanjutan seperti paling kiri, paling kanan, dan regenerasi struktur pohon. Peserta praktik juga diajarkan untuk membuat ADT untuk BST menggunakan daftar yang terhubung dan Praktik ini menegaskan efisiensi logika rekursif dalam menyelesaikan masalah non-linear. Ini juga mendorong teori untuk diterapkan dalam dunia nyata pemrograman.