**LAPORAN PRAKTIKUM**

**ALGORITMA & STRUKTUR DATA**

**MODUL 7**

****

**TREE**

**Oleh:**

**Indra Suryadilaga NIM. 2410817310014**

**PROGRAM STUDI TEKNOLOGI INFORMASI**

**FAKULTAS TEKNIK**

**UNIVERSITAS LAMBUNG MANGKURAT**

**JUNI 2025**

# LEMBAR PENGESAHAN

**LAPORAN PRAKTIKUM ALGORITMA & STRUKTUR DATA**

**MODUL 7**

Laporan Praktikum Algoritma & Struktur Data Modul 6: Tree ini disusun sebagai syarat lulus mata kuliah Praktikum Algoritma & Struktur Data. Laporan Prakitkum ini dikerjakan oleh:

Nama Praktikan : Indra Suryadilaga

NIM : 2410817310014

|  |  |
| --- | --- |
| Menyetujui,  Asisten Praktikum  Harry Pratama Yunus  NIM. 2310817210010 | Mengetahui,  Dosen Penanggung Jawab Praktikum  Andreyan Rizky Baskara, S.Kom., M.Kom  NIP. 199307032019031011 |

# DAFTAR ISI

[LEMBAR PENGESAHAN 2](#_Toc201150346)

[DAFTAR ISI 3](#_Toc201150347)

[DAFTAR GAMBAR 4](#_Toc201150348)

[DAFTAR TABEL 5](#_Toc201150349)

[SOAL 1 6](#_Toc201150350)

[A. Source Code 8](#_Toc201150351)

[B. Output Program 12](#_Toc201150352)

[C. Pembahasan 13](#_Toc201150353)

[GITHUB 17](#_Toc201150354)

# DAFTAR GAMBAR

[Gambar 1. Soal No. 1 Source Code Tree 7](#_Toc201150372)

[Gambar 2. Output Opsi 1 Insert 12](#_Toc201150373)

[Gambar 3. Output Opsi 2 PreOrder 12](#_Toc201150374)

[Gambar 4. Output Opsi 3 InOrder 13](#_Toc201150375)

[Gambar 5. Output Opsi 4 PostOrder 13](#_Toc201150376)

# DAFTAR TABEL

[Table 1. Source Code Tree 8](#_Toc201150383)

# SOAL 1

Cobalah program berikut, perbaiki *output*, lengkapi fungsi *inOrder* dan *postOrder* pada *coding*, *running*, simpan program !

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

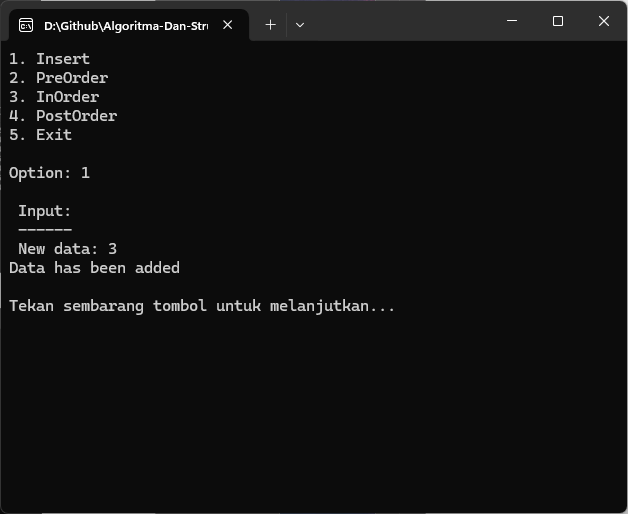
Gambar 1. Soal No. 1 Source Code Tree

### Source Code

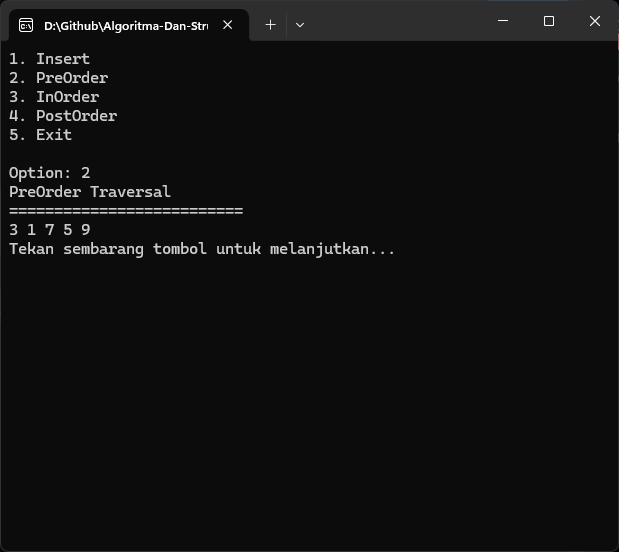
Table 1. Source Code Tree

|  |  |
| --- | --- |
|  | #include <iostream> |
|  | #include <conio.h> |
|  | #include <stdlib.h> |
|  |  |
|  | using namespace std; |
|  |  |
|  | struct Node { |
|  | int data; |
|  | Node \*left; |
|  | Node \*right; |
|  | }; |
|  |  |
|  | void insert(Node \*\*root, int newData){ |
|  | if (\*root == nullptr){ |
|  | Node \*newNode; |
|  | newNode = new Node; |
|  |  |
|  | newNode -> data = newData; |
|  | newNode -> left = nullptr; |
|  | newNode -> right = nullptr; |
|  |  |
|  | \*root = newNode; |
|  |  |
|  | cout << "Data has been added\n"; |
|  | } |
|  | else if (newData < (\*root) -> data){ |
|  | insert(&((\*root)->left), newData); |
|  | } |
|  | else if (newData > (\*root) -> data){ |
|  | insert(&((\*root)->right), newData); |
|  | } |
|  | else if (newData == (\*root) -> data){ |
|  | cout << "Data is already exist\n"; |
|  | } |
|  | } |
|  |  |
|  | void preOrder(Node \*root){ |
|  | if (root != NULL){ |
|  | cout << root->data << " "; |
|  | preOrder(root->left); |
|  | preOrder(root->right); |
|  | } |
|  | } |
|  |  |
|  | void inOrder(Node \*root){ |
|  | if (root != NULL){ |
|  | inOrder(root->left); |
|  | cout << root->data << " "; |
|  | inOrder(root->right); |
|  | } |
|  | } |
|  |  |
|  | void postOrder(Node \*root){ |
|  | if (root != NULL){ |
|  | postOrder(root->left); |
|  | postOrder(root->right); |
|  | cout << root->data << " "; |
|  | } |
|  | } |
|  |  |
|  | void destroyTree(Node \*root) { |
|  | if (root != nullptr) { |
|  | destroyTree(root->left); |
|  | destroyTree(root->right); |
|  | delete root; |
|  | } |
|  | } |
|  |  |
|  | int main(){ |
|  | int opt, val; |
|  | Node \*tree; |
|  | tree = NULL; |
|  |  |
|  | do { |
|  | system("cls"); |
|  |  |
|  | cout << "1. Insert\n"; |
|  | cout << "2. PreOrder\n"; |
|  | cout << "3. InOrder\n"; |
|  | cout << "4. PostOrder\n"; |
|  | cout << "5. Exit\n"; |
|  |  |
|  | cout << "\nOption: "; cin >> opt; |
|  |  |
|  | switch (opt) { |
|  |  |
|  | case 1: |
|  | cout << "\n Input:"; |
|  | cout << "\n ------"; |
|  | cout << "\n New data: "; |
|  | cin >> val; |
|  | insert(&tree, val); |
|  | break; |
|  |  |
|  | case 2: |
|  | cout << "PreOrder Traversal\n"; |
|  | cout << "==========================\n"; |
|  | if (tree == NULL) { |
|  | cout << "Tree is empty!\n"; |
|  | } |
|  | else { |
|  | preOrder(tree); |
|  | } |
|  | break; |
|  |  |
|  | case 3: |
|  | cout << "InOrder Traversal\n"; |
|  | cout << "==========================\n"; |
|  | if (tree == NULL) { |
|  | cout << "Tree is empty!\n"; |
|  | } |
|  | else { |
|  | inOrder(tree); |
|  | } |
|  | break; |
|  |  |
|  | case 4: |
|  | cout << "PostOrder Traversal\n"; |
|  | cout << "==========================\n"; |
|  | if (tree == NULL) { |
|  | cout << "Tree is empty!\n"; |
|  | } |
|  | else { |
|  | postOrder(tree); |
|  | } |
|  | break; |
|  |  |
|  | case 5: |
|  | cout << "\nExiting program and cleaning up memory...\n"; |
|  | destroyTree(tree); |
|  | tree = nullptr; |
|  | cout << "Memory cleaned up. Goodbye!\n"; |
|  | break; |
|  |  |
|  | default: |
|  | cout << "Option is not valid! Please re-enter your option"; |
|  | break; |
|  | } |
|  |  |
|  | if (opt != 5) { |
|  | cout << "\nTekan sembarang tombol untuk melanjutkan..."; |
|  | getch(); |
|  | } |
|  |  |
|  | } while(opt != 5); |
|  | return 0; |
|  | } |

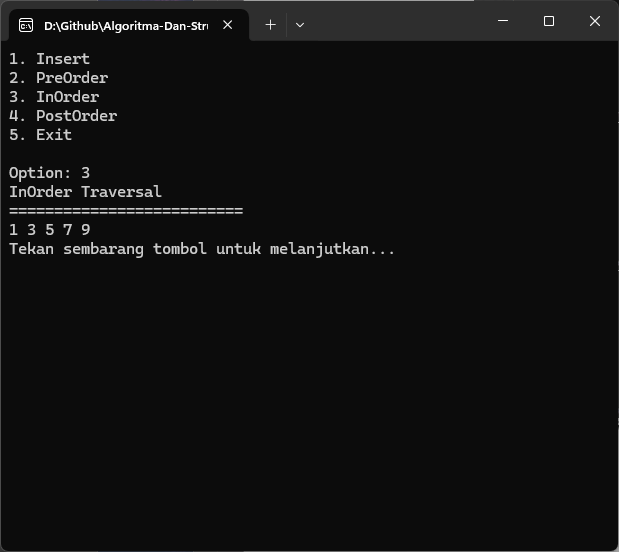
### Output Program



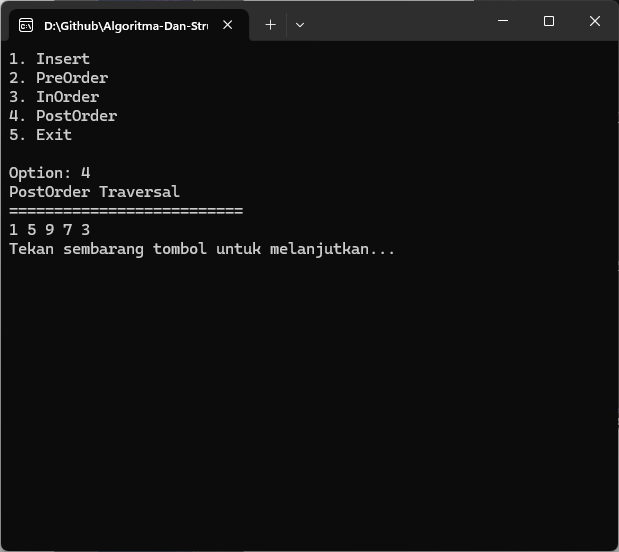
Gambar 2. Output Opsi 1 Insert



Gambar 3. Output Opsi 2 PreOrder



Gambar 4. Output Opsi 3 InOrder



Gambar 5. Output Opsi 4 PostOrder

### Pembahasan

**Alur Program Binary Search Tree**

Ketika program dijalankan, sistem akan menampilkan menu utama dengan 5 pilihan yang tersedia menggunakan do-while loop yang terus berjalan hingga user memilih opsi 5 (Exit). Setiap kali user memilih opsi 1-4, program akan melakukan proses sesuai dengan pilihan yang dipilih, kemudian layar akan dibersihkan menggunakan system("cls") dan menu utama akan ditampilkan kembali.

Untuk opsi 1 (Insert), program akan meminta user untuk memasukkan data baru yang ingin ditambahkan ke dalam tree. Program menggunakan fungsi insert() yang bekerja secara rekursif untuk menemukan posisi yang tepat sesuai dengan aturan BST (Binary Search Tree). Data yang lebih kecil akan ditempatkan di subtree kiri, sedangkan data yang lebih besar akan ditempatkan di subtree kanan. Jika data yang dimasukkan sudah ada dalam tree, program akan menampilkan pesan "Data is already exist".

Untuk opsi 2, 3, dan 4 (PreOrder, InOrder, PostOrder), program akan melakukan traversal atau penelusuran tree menggunakan tiga metode yang berbeda. Sebelum melakukan traversal, program akan memeriksa apakah tree kosong. Jika kosong, program akan menampilkan pesan "Tree is empty!". Jika tidak kosong, program akan menampilkan hasil traversal sesuai dengan metode yang dipilih.

Untuk opsi 5, program akan melakukan cleanup memory dengan memanggil fungsi destroyTree() untuk menghapus semua node yang telah dialokasikan secara dinamis, kemudian mengatur pointer tree menjadi nullptr sebelum mengakhiri program. Hal ini dilakukan untuk mencegah memory leak.

Pada akhir setiap operasi (kecuali exit), program akan menampilkan pesan "Tekan sembarang tombol untuk melanjutkan..." dan menunggu input menggunakan getch(). Setelah user menekan tombol apapun, layar akan dibersihkan dan menu utama akan ditampilkan kembali. Loop ini akan terus berlanjut hingga user memilih opsi 5 untuk keluar dari program.

**Analisis Struktur Data Binary Search Tree**

Binary Search Tree (BST) adalah struktur data tree berbasis node di mana setiap node memiliki maksimal dua child (anak), yaitu left child dan right child. Struktur Node dalam implementasi ini terdiri dari tiga komponen: data bertipe integer yang menyimpan nilai, dan dua pointer (left dan right) yang menunjuk ke child nodes.

Aturan fundamental BST adalah bahwa untuk setiap node, semua nilai di subtree kiri harus lebih kecil dari nilai node tersebut, dan semua nilai di subtree kanan harus lebih besar dari nilai node tersebut. Aturan ini memungkinkan operasi pencarian, penyisipan, dan penghapusan dapat dilakukan dengan efisien.

Program ini menggunakan implementasi pointer-to-pointer (double pointer) pada fungsi insert() untuk memungkinkan modifikasi langsung terhadap pointer root. Hal ini memungkinkan fungsi untuk mengubah nilai pointer yang diteruskan dari fungsi pemanggil, yang sangat penting ketika menambahkan node pertama ke tree yang kosong.

**Analisis Algoritma Insert**

Algoritma insert() bekerja secara rekursif dengan menggunakan prinsip divide and conquer. Proses dimulai dengan memeriksa apakah root tree kosong (nullptr). Jika kosong, program akan membuat node baru menggunakan dynamic memory allocation dengan operator new, mengisi data node tersebut, dan mengatur kedua pointer child menjadi nullptr.

Jika tree tidak kosong, program akan membandingkan data baru dengan data pada node saat ini. Jika data baru lebih kecil, fungsi akan memanggil dirinya sendiri secara rekursif untuk subtree kiri. Jika data baru lebih besar, fungsi akan memanggil dirinya sendiri secara rekursif untuk subtree kanan. Jika data baru sama dengan data yang sudah ada, program akan menampilkan pesan bahwa data sudah exist dan tidak melakukan penyisipan.

Kelebihan dari algoritma insert ini adalah dapat mempertahankan properti BST secara otomatis dan memiliki implementasi yang elegant menggunakan rekursi. Time complexity untuk operasi insert pada average case adalah O(log n), di mana n adalah jumlah node dalam tree. Namun pada worst case (ketika tree menjadi skewed atau tidak seimbang), time complexity bisa mencapai O(n).

Space complexity untuk operasi insert adalah O(h) di mana h adalah height dari tree, karena menggunakan recursive call stack. Pada balanced tree, space complexity adalah O(log n), namun pada skewed tree bisa mencapai O(n).

**Analisis Algoritma Traversal**

Program implementasi ini menyediakan tiga metode traversal yang berbeda: PreOrder, InOrder, dan PostOrder. Ketiga algoritma ini menggunakan pendekatan rekursif untuk mengunjungi setiap node dalam tree dengan urutan yang berbeda.

**PreOrder Traversal (Root-Left-Right)**

Algoritma PreOrder mengunjungi node dalam urutan: root, kemudian subtree kiri, lalu subtree kanan. Proses dimulai dengan mencetak data node saat ini, kemudian memanggil preOrder() secara rekursif untuk left child, dan terakhir memanggil preOrder() secara rekursif untuk right child. Traversal ini berguna untuk membuat copy dari tree atau untuk mengevaluasi expression tree.

**InOrder Traversal (Left-Root-Right)**

Algoritma InOrder mengunjungi node dalam urutan: subtree kiri, root, kemudian subtree kanan. Proses dimulai dengan memanggil inOrder() secara rekursif untuk left child, kemudian mencetak data node saat ini, dan terakhir memanggil inOrder() secara rekursif untuk right child. Keunggulan utama InOrder traversal pada BST adalah menghasilkan output yang terurut secara ascending, sehingga sangat berguna untuk mendapatkan data dalam urutan yang sorted.

**PostOrder Traversal (Left-Right-Root)**

Algoritma PostOrder mengunjungi node dalam urutan: subtree kiri, subtree kanan, kemudian root. Proses dimulai dengan memanggil postOrder() secara rekursif untuk left child, kemudian memanggil postOrder() secara rekursif untuk right child, dan terakhir mencetak data node saat ini. Traversal ini sangat berguna untuk operasi penghapusan tree karena memproses children sebelum parent, sehingga aman untuk menghapus node tanpa kehilangan referensi ke children.

Time complexity untuk semua metode traversal adalah O(n) karena setiap node harus dikunjungi tepat satu kali. Space complexity adalah O(h) dimana h adalah height tree, karena menggunakan recursive call stack yang dalam worst case bisa mencapai height maksimum tree.

# GITHUB

<https://github.com/IndraSuryadilaga/Algoritma-Dan-Struktur-Data>