Discrete Mathematics Project

PROJECT NO: 1

A Project Submitted
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Technology
in
CSE

SUBMITTED TO: Dr. Rishi Asthana

SUBMITTED BY: Group Number-1

Group Members:

Godavarthi Sai Nikhil (210214)

Dasariraju Deepak (210374)

Sara Tarun Goud (210223)

Suresh Raj (210229)

BML MUNJAL UNIVERSITY™

SCHOOL OF ENGINEERING AND TECHNOLOGY
BML MUNJAL UNIVERSITY GURGAON
December, 2023

DECLARATION BY THE CANDIDATES

We hereby declare that Project Number **"1"** has been carried out to fulfill the requirements for the completion of the course **Discrete Mathematics** by the 5th semester of the Bachelor of Technology program in the Department of Computer Science and Engineering during the academic year - 2023-24(odd semester). This experimental work has been carried out by us and submitted to the course instructor **Dr. Rishi Asthana**. Due acknowledgment has been made in the text of the project to all other materials used. This project has been prepared in full compliance with the requirements and constraints of the prescribed curriculum.

Godavarthi Sai Nikhil (210214)
Dasariraju Deepak (210374)
Sara Tarun Goud (210223)
Suresh Raj (210229)

Place: BML Munjal University
Date: 3 December, 2023

ACKNOWLEDGEMENT

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend our sincere thanks to all of them.

We would like to express our gratitude toward our respected professor
Dr. Rishi Asthana & members of TEAM for their kind cooperation and encouragement in the completion of this project.

The project helped us learn more about the trees and their applications and we also learned about many new things while doing the project.

Our thanks and appreciation also go to our colleagues in developing the project and the people who have willingly helped us out with their abilities.

Contents:

1. Problem Statement

Given Problem:

Study rooted trees in deeper details and write a report on basic results about rooted trees and their connection with graph theory. Discuss applications of tress such as Binary Search Trees and Decision Trees. Wherever required you can use available software or write your own codes in C/C++ as the case may be.

Problem Description:

The study of rooted trees and their deep connections with graph theory represents a foundational exploration at the intersection of computer science and mathematics.

Rooted trees, serving as hierarchical structures, wield significant influence across various applications, ranging from facilitating efficient data storage to powering sophisticated search algorithms and informing pivotal decision-making processes.

This project is propelled by the overarching objective of delving deeply into the intricacies of rooted trees, not only to unravel their fundamental properties but also to establish robust connections with the broader field of graph theory.

The primary objectives of this study include:

- Understanding **the Rooted Trees**
- Connection with **Graph Theory**
- Applications in **Binary Search Trees (BST)** and **Decision Trees**.

As an integral part of this research, we embark on practically implementing our theoretical insights. Utilizing the **C/C++ programming languages**, we aim to develop code implementations that reflect the fundamental operations associated with rooted trees. This coding aspect serves not only as a validation of our theoretical findings but also as a hands-on exploration of the practical aspects of rooted trees.

In summation, this comprehensive study aspires to yield a holistic understanding of rooted trees, their interconnectedness with graph theory, and the tangible applications that underscore their significance in algorithmic problem-solving. By amalgamating theoretical insights with practical implementations, we aim to contribute to the broader discourse surrounding rooted trees and their multifaceted role in the computer science and mathematics landscape.

2. Introduction

In the field of discrete mathematics and graph theory, a tree is a special kind of connected graph that doesn't contain any loops or simple circuits. Unlike other graph structures, a tree cannot have cycles, multiple edges, or loops.

2.1 Hierarchical Structure:

A tree has a highly organized and structured layout, following a hierarchical order, like a family tree where you have parents and child nodes. In a tree, there's always one special node right at the top called the "root." It's like the head of the tree, and all other nodes stem from it.

2.2 Trees in Computer Science:

Trees play a very important role in computer science, finding applications across a wide range of algorithms. Their Hierarchical structures are well-defined properties that make them particularly useful in problem-solving and optimization. Here are some key areas where trees are used mostly:

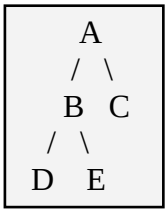
a. **Item Locations:** Trees are employed to create efficient algorithms for locating items in a list.
b. **Huffman Coding:** Trees are used for developing effective codes in algorithms, with a focus on minimizing costs in data transmission and storage.
c. **Game Strategy Analysis:** Trees can be used to study games such as checkers and chess and can help determine winning strategies for playing these games.
d. **Modeling Sequential Decision Procedures:** Trees can be used to model procedures carried out using a sequence of decisions. Constructing these models can help determine the computational complexity of algorithms based on a sequence of decisions, such as sorting algorithms.

Note: An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

3. Rooted trees:

A rooted tree is a hierarchical structure comprising nodes connected by edges, where each node, except for the root, has exactly one parent and zero or more children. The root node serves as the starting point, and each edge represents a relationship between parent and child nodes, directed away from the root (Rosen, 2012). This concept finds applications in diverse fields, including data structures, computer science algorithms, and decision-making processes.

Example:



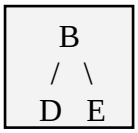
In this example, 'A' is the root node, and 'B' and 'C' are its children. Nodes 'D' and 'E' are children of 'B'.

3.1 Components of Rooted Trees:

1) Node (The Fundamental Element of a Tree):

With respect to trees, a node stands as the elemental building block, representing a fundamental entity within the hierarchical structure. Each node in a tree encapsulates information and may establish relationships with other nodes, thereby forming the intricate web of connections that characterizes the tree.

Example:

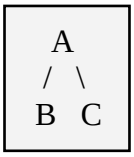


Here, 'B' is a node containing information, and 'D' and 'E' are its children.

2) Root:

The root is the topmost node in the hierarchy and serves as the starting point for traversing the tree. It does not have a parent but may have one or more children.

Example:

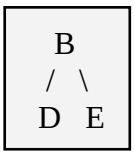


'A' is the root node in this case.

3) Parent Node:

Every node, except the root, has precisely one parent node. The parent node is the immediate ancestor from which the current node is descended.

Example:

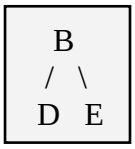


'B' is the parent of 'D' and 'E'.

4) Child Nodes:

Child nodes are nodes that stem from a parent node. A parent node can have zero or more children, forming branches that extend further into the tree.

Example:

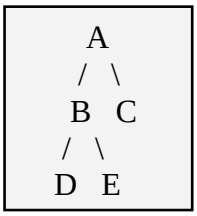


'D' and 'E' are children of 'B'.

5) Leaf Nodes (Terminal Nodes):

Leaf nodes are nodes without children. They reside at the periphery of the tree and represent endpoints in the hierarchy.

Example:

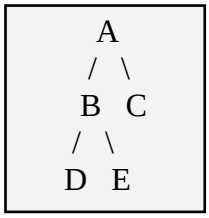


'D' and 'E' are leaf nodes in this tree.

6) Sibling Nodes:

In a rooted tree, sibling nodes are nodes that share the same parent. These nodes form a horizontal relationship within the tree structure, as they are direct offspring of the same parent node.

Example:



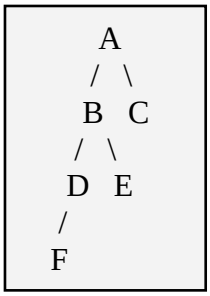
In this example, 'B' and 'C' are sibling nodes, as they both share 'A' as their parent.

7) Ancestor Nodes:

Ancestor nodes are nodes that lie on the path from a given node to the root. These nodes

represent the lineage of the current node, tracing back to the topmost node in the hierarchy.

Example:

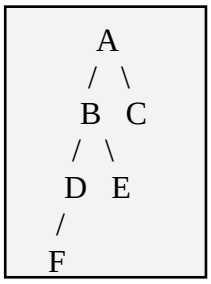


In this tree, 'D' has 'B' and 'A' as its ancestors.

8) Descendant Nodes:

Conversely, descendant nodes of a given node are all the nodes that can be reached by traversing down the tree from that node. These nodes make up the hierarchy below the current node.

Example:



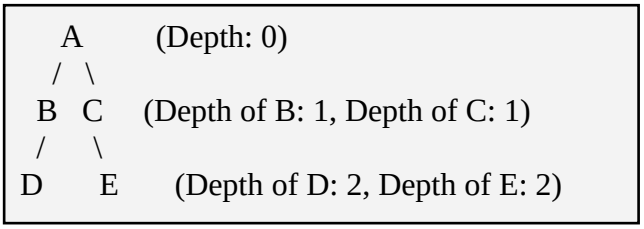
'B' has 'D', 'E', and 'F' as its descendants.

3.2 Properties of Rooted Trees:

1) Depth:

The depth of a node is the length of the path from the root to that node. The depth of the root is typically considered as zero.

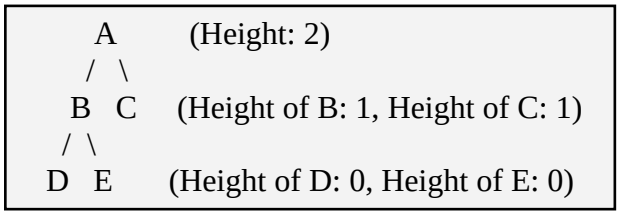
Example:



2) Height:

The height of a node is the length of the longest path from that node to a leaf. The height of the tree is the height of the root node.

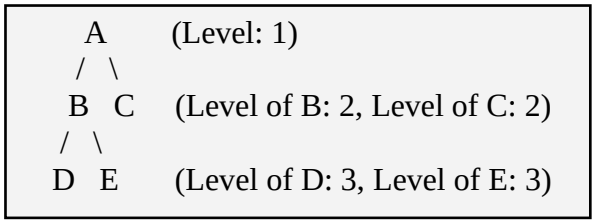
Example:



3) Level:

The level of a node is its depth plus one. The root node is at level 1, its children are at level 2, and so on.

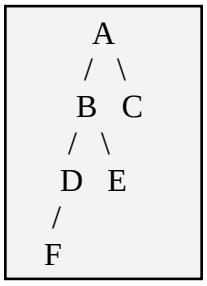
Example:



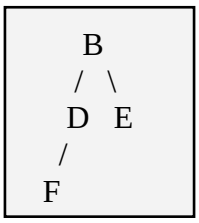
4) Subtrees:

A subtree is a portion of the tree that is itself a rooted tree. It consists of a node and all its descendants, including the node itself.

Example:



Subtree rooted at 'B':

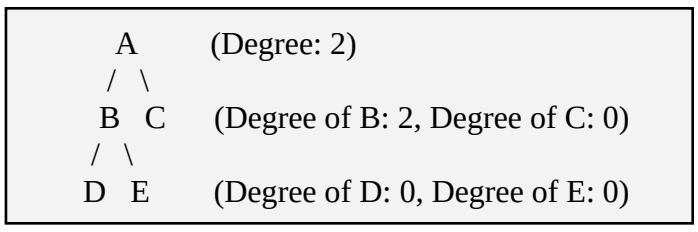


Here, the subtree rooted at 'B' includes 'B', 'D', 'E', and 'F'.

5) Degree of a Node:

The degree of a node in a rooted tree is the number of children it has. A node with no children is referred to as a leaf node and has a degree of zero. The degree of the root is the maximum degree in the tree.

Example:

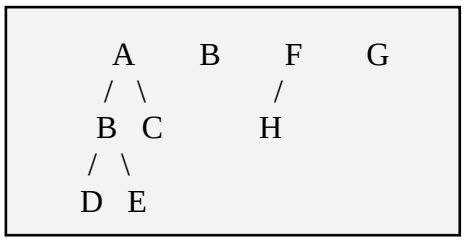


Here, 'A' has a degree of 2, 'B' has a degree of 2, and 'D' and 'E' have degrees of 0.

6) Forest:

A forest is a collection of disjoint trees, each considered as a separate entity. In other words, a forest is a set of rooted trees.

Example:

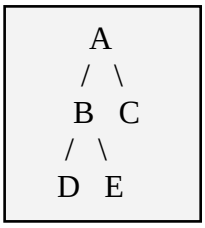


In this example, the collection consists of two rooted trees: one with 'A', 'B', 'C', 'D', and 'E', and another with 'F', 'G', and 'H'.

7) Degree of a Tree:

The degree of a tree is the maximum degree among its nodes. In other words, it is the highest number of children any node in the tree has.

Example:

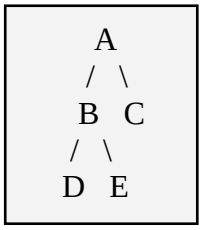


In this tree, the degree of the tree is 2.

8) Balanced Trees:

A rooted tree is considered balanced if, for every node in the tree, the heights of its subtrees differ by at most one. Balanced trees are particularly useful in maintaining efficient search and retrieval operations.

Example:



This tree is balanced as the heights of subtrees rooted at 'B' and 'C' differ by at most one.

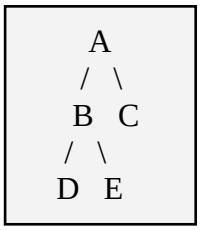
3.3 Connection with Graph Theory:

Rooted trees are closely connected to graph theory, serving as a specialized form of directed acyclic graphs (DAGs). Understanding this connection provides insights into the properties and applications of both rooted trees and graphs. Here are several aspects of this connection:

1) **Directed Acyclic Graphs (DAGs):**

Rooted trees can be considered a specific type of directed acyclic graph where each node has at most one incoming edge (except for the root) and may have zero or more outgoing edges. The acyclic nature of trees ensures that there are no cycles, making them a subset of DAGs.

Example:



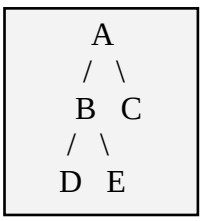
This is a rooted tree where 'A' is the root, and 'B' and 'C' are its children. Each node has at most one incoming edge (except for the root), and there are no cycles, making it a DAG.

2) **Hierarchy and Parent-Child Relationships:**

In graph theory, edges in a directed graph typically represent relationships between nodes. In the case of rooted trees, the edges explicitly indicate parent-child relationships. The hierarchical structure of rooted trees captures a specific type of directed relationship inherent in many real-world scenarios.

3) **Paths and Connectivity:**

Paths in rooted trees are analogous to directed paths in graphs. The concept of connectivity and the exploration of paths between nodes in a graph find direct applications in rooted trees. The length of paths in trees corresponds to the depth and height of nodes.
Example:



The path from 'A' to 'E' is A -> B -> E, highlighting the concept of paths in a tree. Connectivity is established through these paths.

4) **Tree as a Specialized Graph:**

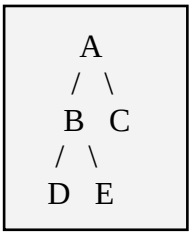
A tree is essentially a connected and acyclic graph. By understanding the properties of trees, one gains a deeper understanding of more general graph structures. Conversely, graph algorithms and concepts can be applied to analyze and manipulate rooted trees.

5) **Depth-first and Breadth-First Traversal:**

Graph traversal algorithms, such as depth-first search (DFS) and breadth-first search (BFS), are directly applicable to rooted trees. These algorithms can be employed to visit nodes

systematically, revealing the structure and relationships within the tree.

Example:



Depth-First Traversal (DFS): A ->B -> D -> E -> C
Breadth-First Traversal (BFS): A -> B -> C -> D -> E

6) Subtrees and Connected Components:

Subtrees in a rooted tree correspond to connected components in a graph. Analyzing subtrees provides insights into the decomposition of the tree into smaller, more manageable parts, mirroring the concept of connected components in graphs.

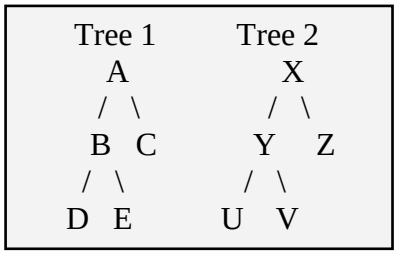
7) Application of Graph Algorithms:

Algorithms designed for graphs, such as finding paths, detecting cycles, and determining connectivity, can be adapted, and applied to rooted trees. This adaptability underscores the shared fundamental principles between these structures.

8) Isomorphism:

The concept of graph isomorphism, where two graphs are structurally identical, extends to rooted trees. Isomorphic trees share the same structure even if their node labels differ. Understanding isomorphism in the context of rooted trees provides a bridge to the broader study of graph isomorphism.

Example:

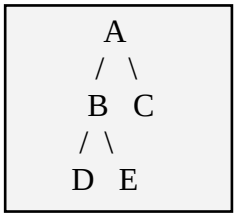


Tree 1 and Tree 2 are isomorphic, even though the node labels differ. Both trees share the same structural arrangement.

9) Binary Trees and Graph Theory:

Binary trees, a specific type of rooted tree where each node has at most two children, have a direct connection with graph theory. Binary trees can be seen as a specialized form of directed graph where each node has at most two outgoing edges. The study of binary trees aligns with the exploration of directed graphs with a degree constraint.

Example:

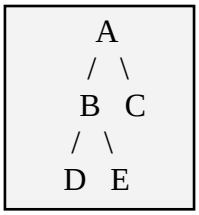


This binary tree can be seen as a directed graph with nodes {A, B, C, D, E} and edges {(A, B), (A, C), (B, D), (B, E)}. Each node has at most two outgoing edges, aligning with the properties of a binary tree.

10) Topological Ordering:

Rooted trees provide a natural example of a topological ordering, where nodes are linearly ordered based on the parent-child relationships. Topological ordering is a concept widely used in graph theory, especially in directed acyclic graphs, to represent a linear ordering of nodes that respects the partial order defined by the edges.

Example:

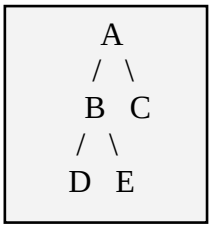


The topological ordering of the nodes based on parent-child relationships is A, B, D, E, C. This linear ordering respects the partial order defined by the edges in the tree.

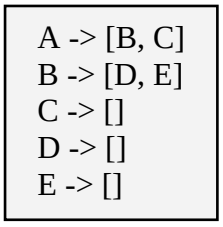
11) Graph Representations:

Rooted trees can be represented using various graph representations, such as adjacency lists or adjacency matrices, which are common in graph theory. The choice of graph representation impacts the efficiency of algorithms and operations, providing a link between data structures used for trees and those used for general graphs.

Example:



The adjacency list representation of this tree is:

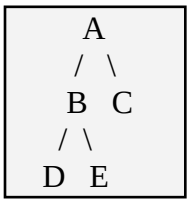


This adjacency list representation is a common graph representation in graph theory.

12) Minimum Spanning Trees (MST):

Minimum spanning trees are important in graph theory for connecting all nodes in a graph with the minimum possible total edge weight. In the context of rooted trees, the minimum spanning tree can be found by selecting a root and spanning the entire tree with minimum weight edges.

Example:

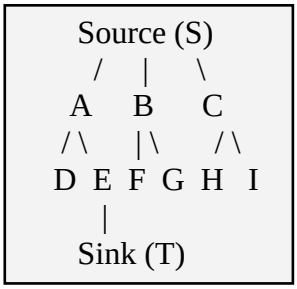


Assuming weights on edges (A, B), (A, C), (B, D), (B, E), the minimum spanning tree is the entire tree itself, as it connects all nodes with the minimum possible total edge weight.

13) Network Flow in Trees:

Concepts from network flow theory, such as flow conservation and max flow-min cut, can be applied to rooted trees. The branching structure of trees allows for an exploration of flow dynamics and conservation within the tree.

Example: Consider a rooted tree representing a water distribution network:

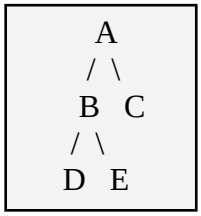


This tree can be analyzed for network flow dynamics, such as how water flows from the source to the sink through different branches.

14) Planar Graphs and Tree Embedding:

Rooted trees have a planar embedding, meaning they can be drawn in the plane without any edge intersections. Understanding planarity in rooted trees extends to the study of planar graphs in general graph theory.

Example:

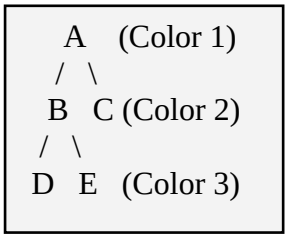


This tree can be drawn in the plane without any edge intersections, showcasing its planar embedding.

15) Graph Coloring and Tree Labeling:

Graph coloring techniques, such as chromatic numbers and proper colorings, can be applied to rooted trees. In this context, the labeling of nodes in a rooted tree corresponds to assigning colors to satisfy specific coloring conditions, mirroring graph coloring concepts.

Example:



This example illustrates the application of graph coloring concepts to a rooted tree, where adjacent nodes must have different colors.

4. Applications of Rooted Trees

Trees are fundamental structures in discrete mathematics that have wide-ranging applications in computer science, network design, and optimization problems. In this report, we will explore the various real-world applications of trees and their significance in solving complex problems efficiently.

Some of the applications are Binary Search Trees (BST), Decision Trees, Heap Data Structures, Expression Trees, File System Structures, and Game Trees.

4.1 Binary Search Trees (BST):

Binary Search Trees (BST) are fundamental data structures that play a crucial role in various computer science applications. A BST is a hierarchical structure where each node has at most two children, and nodes to the left contain values smaller than the parent, while nodes to the right contain values greater than the parent (Binary Search Tree, n.d.). This ordered arrangement allows for efficient search, insertion, and deletion operations, making BSTs indispensable in diverse domains.

Applications of BST:

1. **Database Management Systems (DBMS):**

   BSTs are extensively used in DBMS for indexing and searching operations. The ordered nature of BSTs enables quick retrieval of data based on keys, optimizing the efficiency of database queries.

   Binary search trees are used in database management systems to efficiently store and retrieve data. The tree's structure allows for quick searching, insertion, and deletion of records, making it ideal for applications where fast data access is crucial.

2. **Symbol Tables:**

   Symbol tables are vital data structures in compilers and interpreters to manage identifiers such as variables and functions. BSTs are employed to store and organize these symbols, allowing for quick lookups during the compilation or interpretation process. The hierarchical nature of the tree allows for efficient insertion and retrieval of symbols based on their names.

3. **File Systems:**

   File systems use BSTs to maintain the hierarchical structure of directories and files. Each node in the BST represents a directory or file, and the tree structure reflects the relationships between them. This organization facilitates efficient searching and retrieval of files, as the tree's ordered nature allows for quick navigation through the directory structure.

4. **Network Routing Algorithms:**

   BSTs are applied in networking for routing algorithms, especially in scenarios where routing decisions are based on certain criteria, such as IP addresses. The hierarchical structure of the BST corresponds to the network topology, allowing for efficient routing decisions.

Nodes in the tree represent routers or network nodes, and the tree structure guides the path selection for data packets.

5. **Auto-Complete and Spell Checking:**

   In applications like text editors or search engines, BSTs are used for implementing auto-complete features. The ordered nature of the tree enables efficient prefix searches, making it easy to suggest words based on partial input. Additionally, BSTs can be employed in spell-checking algorithms to quickly identify and correct misspelled words.

6. **Compression Algorithms:**

   Some compression algorithms, such as Huffman coding, utilize BSTs to represent hierarchical structures of characters in each dataset. In Huffman coding, characters are assigned variable-length codes based on their frequency in the dataset. A BST is constructed in a way that characters with higher frequencies have shorter codes, leading to more efficient compression of the data.

4.2 Decision Trees:

Decision Trees are powerful tools employed in machine learning, data analysis, and decision-making processes. These tree-like structures consist of nodes representing decisions, branches representing possible outcomes, and leaves containing the final decisions or predictions. Decision Trees are versatile and find application in various domains due to their ability to model complex decision-making scenarios in a comprehensible manner (Decision Tree, n.d.).

Applications of Decision Trees:

1. **Machine Learning and Data Mining:**

   Decision Trees are extensively used in machine learning for classification and regression tasks. They can efficiently model and analyze complex datasets, making them valuable in predictive modeling.

2. **Game Playing:**

   Decision Trees find application in game-playing algorithms, aiding in decision-making processes by simulating possible moves and outcomes. This is evident in games such as chess and tic-tac-toe.

3. **Business Decision Making:**

   In business, Decision Trees are applied for decision analysis and strategic planning. They help visualize potential outcomes and choices, facilitating informed decision-making.

4. **Medical Diagnosis:**

   Decision Trees are employed in medical diagnosis systems to analyze patient data and determine potential conditions based on symptoms, test results, and medical history.

5. **Credit Scoring:**

   Financial institutions use Decision Trees in credit scoring models to assess the creditworthiness of individuals. The tree structure aids in evaluating various factors to predict the likelihood of loan default.

6. **Fault Diagnosis in Engineering:**

   Decision Trees are utilized in engineering and maintenance for fault diagnosis in systems. They help identify and troubleshoot potential issues by modeling the relationships between different components and their behaviors.

5. Implementing Applications of Rooted Trees

5.1 Binary Search Trees:

1. Database Management Systems (DBMS):

Implementing insertion (itsadityash, n.d.), deletion, and searching of keys using C++

ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

procedure *insertion*(T: binary search tree, x: item)
$v :=$ root of T
{a vertex not present in T has the value *null*}
while $v \neq null$ and $label(v) \neq x$
  if $x < label(v)$ **then**
    if left child of $v \neq null$ **then** $v :=$ left child of v
    else add *new vertex* as a left child of v and set $v := null$
  else
    if right child of $v \neq null$ **then** $v :=$ right child of v
    else add *new vertex* as a right child of v and set $v := null$
if root of $T = null$ **then** add a vertex v to the tree and label it with x
else if v is null or $label(v) \neq x$ **then** label *new vertex* with x and let v be this new vertex
return v {$v =$ location of x}

Code:

Click Here for the Code: https://onlinegdb.com/LnikGwCs7

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

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int value) : key(value), left(nullptr), right(nullptr){}
};

Node* insert(Node* root, int key) {
    if (root == nullptr) {
        return new Node(key);
    }
    if(key < root->key) {
        root->left = insert(root->left, key);
    }else if (key > root->key){
        root->right = insert(root->right, key);
    }
    return root;
}

bool search(Node* root, int key) {
    if (root == nullptr) {
        return false;
    }
    if (key == root->key) {
```

```
        return true;
    } else if (key < root->key) {
        return search(root->left, key);
    } else {
        return search(root->right, key);
    }
}

Node* deleteNode(Node* root, int key) {
    if (root == nullptr) {
        return root;
    }
        if (key < root->key)
        {
            root->left = deleteNode(root->left, key);
        } else if (key > root->key)
        {
            root->right = deleteNode(root->right, key);
        }
    else {
        if (root->left == nullptr)
        {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = root->right;
        while (temp->left != nullptr) {
            temp = temp->left;
        }
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
void printPreorder(Node* root) {
    if (root != nullptr) {
        cout << root->key << " ";
        printPreorder(root->left);
        printPreorder(root->right);
    }
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
```

```
    root = insert(root, 40);
    cout << "Initial Tree (Preorder): ";
    printPreorder(root);
    cout << endl;
    int choice, value;
    do{
        cout << "\nOptions:\n";
        cout << "1. Search for a key\n";
        cout << "2. Insert a new key\n";
        cout << "3. Delete a key\n";
        cout << "4. Print the tree in preorder\n";
        cout << "0. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter the key to search: ";
                cin >> value;
                if (search(root, value)) {
                    cout << "Key " << value << " found in the tree.\n";
                } else {
                    cout << "Key " << value << " not found in the tree.\n";
                }
                break;
            case 2:
                cout << "Enter the key to insert: ";
                cin >> value;
                root = insert(root, value);
                cout << "Key " << value << " inserted.\n";
                break;
            case 3:
                cout << "Enter the key to delete: ";
                cin >> value;
                root = deleteNode(root, value);
                cout << "Key " << value << " deleted.\n";
                break;
            case 4:
                cout << "Tree in preorder: ";
                printPreorder(root);
                cout << endl;
                break;
            case 0:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid choice. Try again.\n";
        }
    } while (choice != 0);
    return 0;
}
```

Time and Space Complexities:

Operations	Best	Worst	Space
Insertion	O(log n)	O(n)	O(1)
Search	O(1)	O(n)	O(1)
Deletion	O(log n)	O(n)	O(1)
Preorder Traversal	O(n)		O(h)

2. **Implementing Binary Tree Visualization for Preorder Traversal:**

   Implementing Preorder traversal Using Python

   Code:

   Click Here for the Code: https://colab.research.google.com/drive/1HFZVS76kIpV-Zc8BvrKVPG4CxLjOh05M?usp=sharing

```
import networkx as nx
import matplotlib.pyplot as plt

class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def build_balanced_tree(numbers, start, end):
    if start > end:
        return None

    mid = (start + end) // 2
    root = TreeNode(numbers[mid])

    root.left = build_balanced_tree(numbers, start, mid - 1)
    root.right = build_balanced_tree(numbers, mid + 1, end)

    return root

def print_preorder(node):
    if node is not None:
        print(node.value, end=' ')
        print_preorder(node.left)
        print_preorder(node.right)

def visualize_tree(tree):
    G = nx.Graph()
```

```
    pos = {}

    def add_nodes_edges(node, parent=None, x=0, y=0, layer=1, width=2.0,
vert_gap=0.4):
        if node is None:
            return

        G.add_node(node.value, pos=(x, -layer * vert_gap))
        if parent is not None:
            G.add_edge(parent.value, node.value)

        add_nodes_edges(node.left, node, x=x - width / 2.0, y=y - 1,
layer=layer + 1, width=width / 2.0, vert_gap=vert_gap)
        add_nodes_edges(node.right, node, x=x + width / 2.0, y=y - 1,
layer=layer + 1, width=width / 2.0, vert_gap=vert_gap)

    add_nodes_edges(tree)

    # Draw the graph
    labels = {node: node for node in G.nodes()}
    nx.draw(G, pos=nx.get_node_attributes(G, 'pos'), with_labels=True,
labels=labels,
            node_size=700, node_color="skyblue", font_size=8,
font_color="black")
    plt.show()

# Example usage:
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
root = build_balanced_tree(numbers, 0, len(numbers) - 1)

print("Preorder traversal of binary tree is:")
print_preorder(root)

print("\nVisual representation of the binary tree:")
visualize_tree(root)
```

5.2 Decision Trees:

1. Game Playing (Tic-Tac-Toe):

Implementing a game-playing agent using minimax algorithm and alpha-beta purring (Minimax algorithm and alpha-beta pruning, n.d.) using C++.

Code:

Click Here for the Code: https://www.onlinegdb.com/nppwqZMVk

```
#include <bits/stdc++.h>
using namespace std;
const int boardSize = 3;
void printBoard(char board[boardSize][boardSize]) {
    cout << "Tic-Tac-Toe Board:\n";
    for (int i = 0; i < boardSize; i++) {
        for (int j = 0; j < boardSize; j++) {
            cout << " " << board[i][j] << " ";
            if (j < boardSize - 1)
                cout << "|";
        }
        cout << endl;
        if (i < boardSize - 1)
            cout << "-----------\n";
    }
    cout << endl;
}

// Function to check if the game is over (win or draw)
bool isGameOver(char board[boardSize][boardSize]) {
    for (int i = 0; i < boardSize; i++) {
        if (board[i][0] == board[i][1] && board[i][1] == board[i][2] && board[i][0]
!= ' ')
            return true;
    }

    for (int j = 0; j < boardSize; j++) {
        if (board[0][j] == board[1][j] && board[1][j] == board[2][j] && board[0][j]
!= ' ')
            return true;
    }

    if (board[0][0] == board[1][1] && board[1][1] == board[2][2] && board[0][0] !=
' ')
        return true;

    if (board[0][2] == board[1][1] && board[1][1] == board[2][0] && board[0][2] !=
' ')
        return true;

    for (int i = 0; i < boardSize; i++) {
```

```
        for (int j = 0; j < boardSize; j++) {
            if (board[i][j] == ' ')
                return false;
        }
    }

    return true;
}
int evaluate(char board[boardSize][boardSize]) {
    for (int i = 0; i < boardSize; i++) {
        if (board[i][0] == board[i][1] && board[i][1] == board[i][2]) {
            if (board[i][0] == 'X') return 10;
            else if (board[i][0] == 'O') return -10;
        }
    }

    for (int j = 0; j < boardSize; j++) {
        if (board[0][j] == board[1][j] && board[1][j] == board[2][j]) {
            if (board[0][j] == 'X') return 10;
            else if (board[0][j] == 'O') return -10;
        }
    }
    if (board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
        if (board[0][0] == 'X') return 10;
        else if (board[0][0] == 'O') return -10;
    }

    if (board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
        if (board[0][2] == 'X') return 10;
        else if (board[0][2] == 'O') return -10;
    }
    return 0;
}
int minimax(char board[boardSize][boardSize], int depth, bool isMax, int alpha, int
beta) {
    if (isGameOver(board)) {
        int score = evaluate(board);
        return score;
    }

    if (isMax) {
        int bestScore = INT_MIN;
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                if (board[i][j] == ' ') {
                    board[i][j] = 'X';
                    bestScore = max(bestScore, minimax(board, depth + 1, !isMax,
alpha, beta));
                    board[i][j] = ' ';
                    alpha = max(alpha, bestScore);
                    if (beta <= alpha)
                        break;
```

```
                }
            }
        }
        return bestScore;
    } else {
        int bestScore = INT_MAX;
        for (int i = 0; i < boardSize; i++) {
            for (int j = 0; j < boardSize; j++) {
                if (board[i][j] == ' ') {
                    board[i][j] = 'O';
                    bestScore = min(bestScore, minimax(board, depth + 1, !isMax,
alpha, beta));
                    board[i][j] = ' ';
                    beta = min(beta, bestScore);
                    if (beta <= alpha)
                        break; // Prune the branch
                }
            }
        }
        return bestScore;
    }
}
pair<int, int> findBestMove(char board[boardSize][boardSize]) {
    int bestScore = INT_MIN;
    pair<int, int> bestMove;
    bestMove.first = -1;
    bestMove.second = -1;

    for (int i = 0; i < boardSize; i++) {
        for (int j = 0; j < boardSize; j++) {
            if (board[i][j] == ' ') {
                board[i][j] = 'X';
                int moveScore = minimax(board, 0, false, INT_MIN, INT_MAX);
                board[i][j] = ' ';

                if (moveScore > bestScore) {
                    bestScore = moveScore;
                    bestMove.first = i;
                    bestMove.second = j;
                }
            }
        }
    }

    return bestMove;
}

int main() {
    char board[boardSize][boardSize] = {
        {' ', ' ', ' '},
        {' ', ' ', ' '},
        {' ', ' ', ' '}
```

```
    };

    cout << "Tic-Tac-Toe Game\n";

    while (!isGameOver(board)) {
        printBoard(board);
        pair<int, int> move = findBestMove(board);
        board[move.first][move.second] = 'X';
        cout << "Player 'X' (Max) makes a move:\n";
        if (isGameOver(board)) {
            printBoard(board);
            break;
        }
        printBoard(board);
        int row, col;
        cout << "Enter the row and column for Player 'O' (Min): ";
        cin >> row >> col;
        if (row >= 0 && row < boardSize && col >= 0 && col < boardSize &&
board[row][col] == ' ') {
            board[row][col] = 'O';
            cout << "Player 'O' (Min) makes a move:\n";
        } else {
            cout << "Invalid move. Try again.\n";
        }
    }
    printBoard(board);
    if (evaluate(board) == 10)
        cout << "Player 'X' (Max) wins!\n";
    else if (evaluate(board) == -10)
        cout << "Player 'O' (Min) wins!\n";
    else
        cout << "It's a draw!\n";
    return 0;
}
```

2. Sorting Student's marks:

Implementing a decision Tree in Sorting Student's marks using C++.

The provided C++ code demonstrates the implementation of a Decision Tree to sort students based on their marks. Each node in the tree represents a decision point, where students are divided into branches based on their marks. The tree is constructed dynamically as student information is input.

Code:

Click Here for the Code: https://onlinegdb.com/vnfRDvmzl

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Node {
    int studentNumber;
    int marks;
    Node* left;
    Node* right;

    Node(int studentNumber, int marks) : studentNumber(studentNumber),
marks(marks), left(nullptr), right(nullptr) {}
};

class DecisionTree {
private:
    Node* root;

    Node* insert(Node* node, int studentNumber, int marks) {
        if (node == nullptr) {
            return new Node(studentNumber, marks);
        }

        if (marks <= node->marks) {
            node->left = insert(node->left, studentNumber, marks);
        } else {
            node->right = insert(node->right, studentNumber, marks);
        }

        return node;
    }

    void displayTree(Node* node, int depth) {
        if (node != nullptr) {
            displayTree(node->right, depth + 1);
            for (int i = 0; i < depth; ++i) {
                std::cout << "    ";
            }
            std::cout << node->studentNumber << "." << node->marks << "\n";
            displayTree(node->left, depth + 1);
```

```
        }
    }

    void inOrderTraversal(Node* node, std::vector<Node*>& sortedList) {
        if (node != nullptr) {
            inOrderTraversal(node->left, sortedList);
            sortedList.push_back(node);
            inOrderTraversal(node->right, sortedList);
        }
    }

    Node* buildTreeFromSorted(std::vector<Node*>& sortedList, int start, int end) {
        if (start > end) {
            return nullptr;
        }

        int mid = (start + end) / 2;
        Node* node = sortedList[mid];

        node->left = buildTreeFromSorted(sortedList, start, mid - 1);
        node->right = buildTreeFromSorted(sortedList, mid + 1, end);

        return node;
    }

public:
    DecisionTree() : root(nullptr) {}

    void insert(int studentNumber, int marks) {
        root = insert(root, studentNumber, marks);
    }

    void displayTree() {
        displayTree(root, 0);
    }

    void displaySortedList() {
        std::vector<Node*> sortedList;
        inOrderTraversal(root, sortedList);

        std::cout << "Sorted List of Students:\n";
        for (const auto& node : sortedList) {
            std::cout << node->studentNumber << "." << node->marks << " ";
        }
        std::cout << "\n";
    }

    void buildTreeFromSorted() {
        std::vector<Node*> sortedList;
        inOrderTraversal(root, sortedList);
        int size = sortedList.size();
```

```
        root = buildTreeFromSorted(sortedList, 0, size - 1);
    }
};

int main() {
    int numStudents;

    std::cout << "Enter the number of students: ";
    std::cin >> numStudents;

    DecisionTree tree;

    for (int i = 0; i < numStudents; ++i) {
        int marks;
        std::cout << "Enter marks for student " << i + 1 << ": ";
        std::cin >> marks;
        tree.insert(i + 1, marks);
    }

    tree.buildTreeFromSorted();

    std::cout << "\nDecision Tree for Sorting Students by Marks:\n";
    tree.displayTree();

    tree.displaySortedList();

    return 0;
}
```

6. Results:

Database Management Systems (DBMS):

```
Initial Tree (Preorder): 50 30 20 40

Options:
1. Search for a key
2. Insert a new key
3. Delete a key
4. Print the tree in preorder
0. Exit
Enter your choice: 2
Enter the key to insert: 22
Key 22 inserted.

Options:
1. Search for a key
2. Insert a new key
3. Delete a key
4. Print the tree in preorder
0. Exit
Enter your choice: 1
Enter the key to search: 30
Key 30 found in the tree.

Options:
1. Search for a key
2. Insert a new key
3. Delete a key
4. Print the tree in preorder
0. Exit
Enter your choice: 3
Enter the key to delete: 40
Key 40 deleted.
```

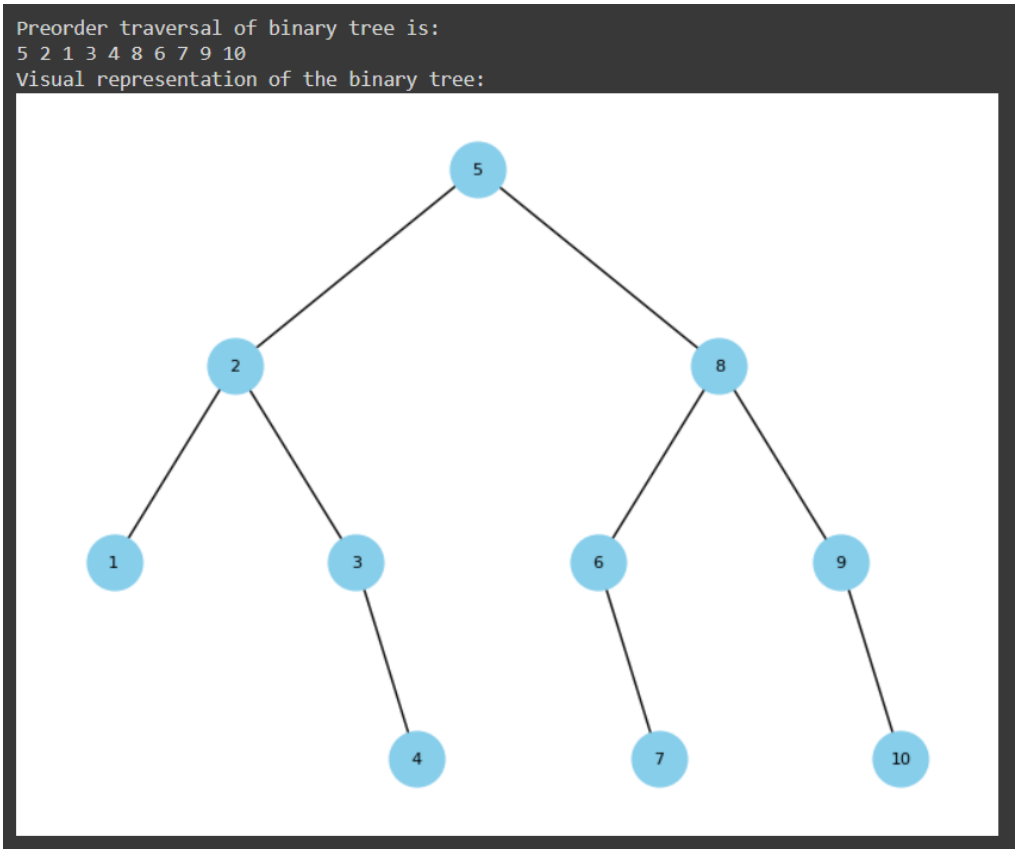
```
Options:
1. Search for a key
2. Insert a new key
3. Delete a key
4. Print the tree in preorder
0. Exit
Enter your choice: 4
Tree in preorder: 50 30 20 22

Options:
1. Search for a key
2. Insert a new key
3. Delete a key
4. Print the tree in preorder
0. Exit
Enter your choice: 0
Exiting...


...Program finished with exit code 0
Press ENTER to exit console.
```

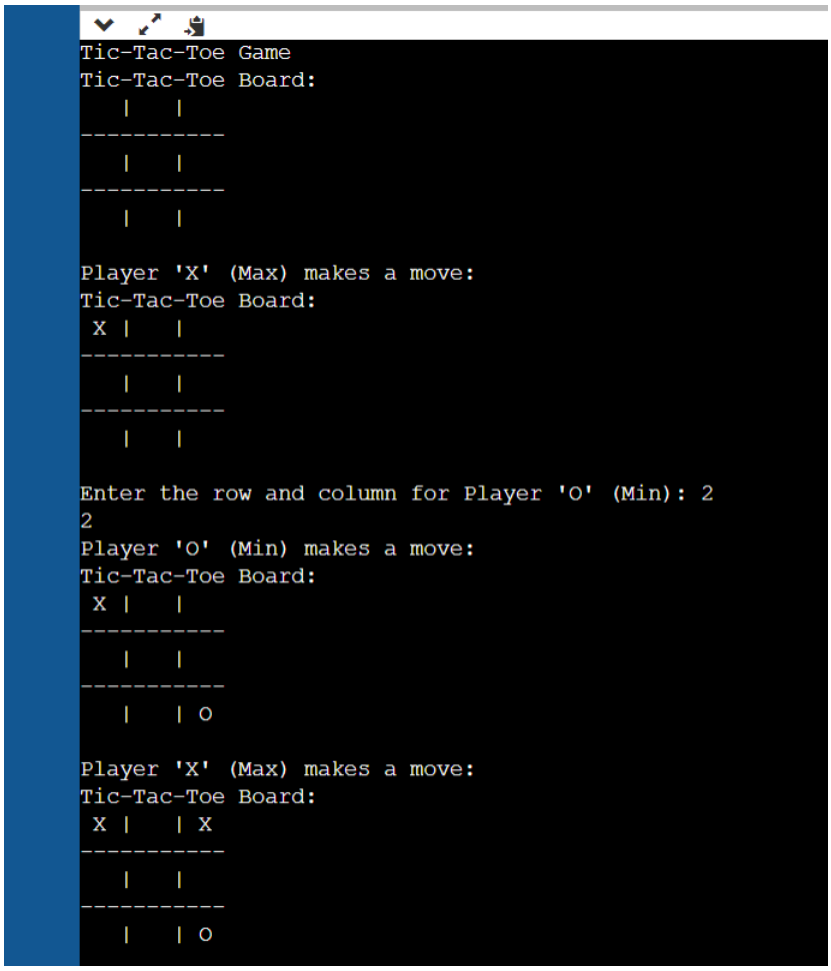
Implementing Binary Tree Visualization for Preorder Traversal:

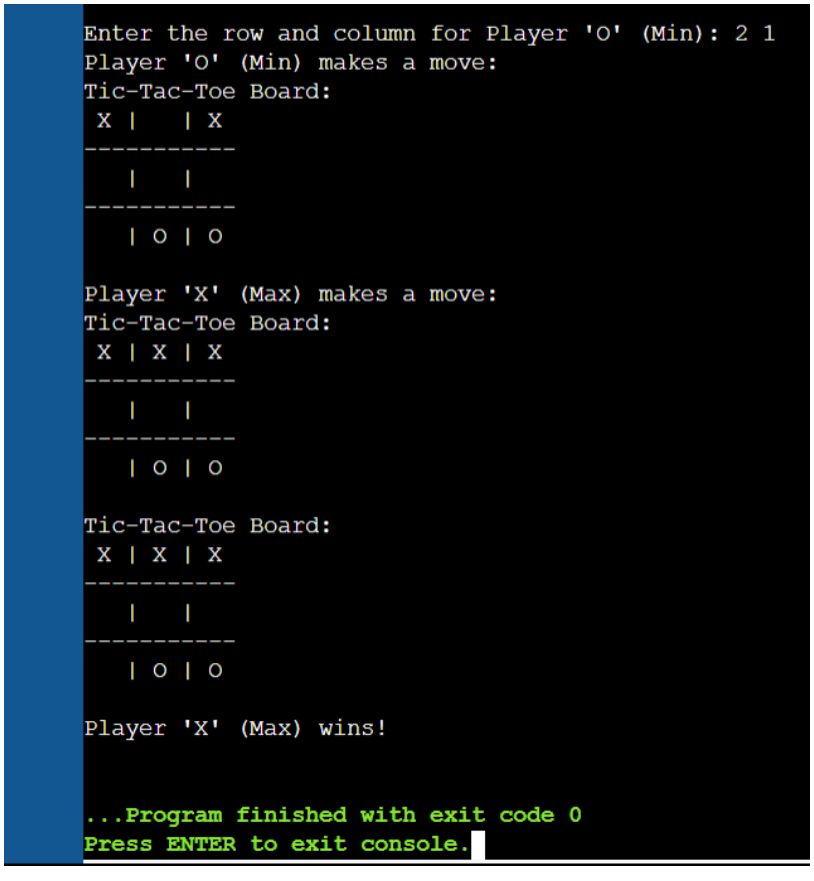
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



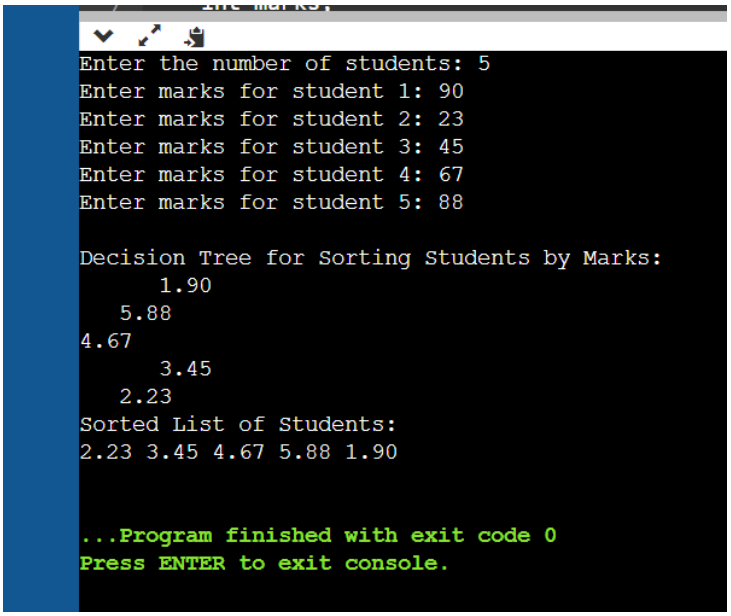
The combined output of the preorder traversal and the visual representation allows for a comprehensive understanding of the binary tree. The visualization provides a clear and concise representation of the tree's structure, aiding in the interpretation of hierarchical relationships within the binary tree.

Game Playing (Tic-Tac-Toe):





Sorting Student's marks:



Explanation:

The root node represents the initial decision based on the average marks of all students.

In this case, the root node is 67, and it divides the students into two branches based on whether their marks are less than or equal to 67 (Left Branch) or greater than 67 (Right Branch).

The Decision Tree effectively organizes students based on their average marks, creating a hierarchical structure for easy sorting and analysis. Students are sorted into different branches of the tree based on specific criteria, providing a clear visual representation of the decision-making process. The sorted list at the end of the output provides the order in which students appear in the tree structure.

The representation 3.45 represents "3"- Student-3 & "45"-Student-3's marks.

7. Conclusion

Rooted trees serve as fundamental structures in discrete mathematics, particularly in graph theory. The exploration of rooted trees has provided insights into their basic properties and intricate connections with graph theory. Throughout this study, we've uncovered key concepts, including nodes, edges, depth, and height, forming the foundation of rooted trees.

In conclusion, the study of rooted trees offers valuable insights into both theoretical and practical aspects of computer science. Rooted trees, a fundamental structure in graph theory, serve as a foundation for various applications, including Binary Search Trees (BST) and Decision Trees.

Binary Search Trees offer an efficient solution for database management systems (DBMS). Through C++ implementation, we explored the insertion, deletion, and searching of keys. The tree's hierarchical structure enables quick access to data, making it particularly suitable for applications where fast data retrieval is essential.

On the other hand, Decision Trees play a crucial role in game-playing scenarios, as demonstrated by the implementation of a Tic-Tac-Toe game-playing agent in C++. The use of algorithms such as minimax and alpha-beta pruning showcases how Decision Trees contribute to strategic decision-making in games.

As technology advances, rooted trees remain a relevant and impactful area of study, influencing the design and optimization of algorithms across various domains. Understanding their principles provides a solid foundation for addressing complex computational challenges and developing innovative solutions in the ever-evolving field of computer science.

8. Reference

Binary Search Tree. (n.d.). Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/binary-search-tree-data-structure/

Decision Tree. (n.d.). Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/decision-tree/

itsadityash. (n.d.). *Insertion in Binary Search Tree (BST)*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/insertion-in-binary-search-tree/?ref=lbp

Minimax algorithm and alpha-beta pruning. (n.d.). Retrieved from mathsapp: https://mathspp.com/blog/minimax-algorithm-and-alpha-beta-pruning

Rosen, K. H. (2012). *Discrete Mathematics and its Applications Seventh Edition.* New York: McGraw-Hill Companies. Retrieved from https://faculty.ksu.edu.sa/sites/default/files/rosen_discrete_mathematics_and_its_applications_7th_edition.pdf