**Binary Search Tree (BST) Documentation**

**Introduction**

**Background**

Data structures and algorithms form the backbone of computer science. Efficient data management and manipulation are vital for solving complex computational problems. Among various data structures, trees play a significant role due to their hierarchical nature and flexibility in representing data relationships.

**Objectives**

* Provide an intuitive and interactive educational tool to help students and learners understand the structure and properties of Binary Search Trees (BSTs).
* Enable users to input arrays and observe the dynamic creation of the BST based on the given input.

**Significance**

This project is important because it has real-world implications. In several industries, such as network routing and transportation, effective route planning is crucial. By offering the best routing options, Dijkstra's algorithm implementation can result in advancements in various domains.

**Problem Definition and Requirements**

**Problem Statement**

BST Visualization.

**Software Requirements**

* **C++ Compiler:** GCC or any other standard C++ compiler.
* **IDE:** Visual Studio Code.

**Hardware Requirements**

* Standard PC.

**Proposed Design / Methodology**

**Algorithms Used**

A Binary Search Tree (BST) is a type of binary tree that maintains a specific order property. This property makes BSTs highly efficient for searching, insertion, and deletion operations. Here are the key characteristics and definitions related to a BST:

**Node Structure**

Each node contains a key (or value), a left child, and a right child.

**Ordering**

For any given node:

* All values in the left subtree are less than the node’s value.
* All values in the right subtree are greater than the node’s value.

**Uniqueness**

In a standard BST, each key must be unique. This means no two nodes can have the same value.

**Recursive Nature**

Both the left and right subtrees must also be binary search trees.

**Height**

The height of a BST is defined as the number of edges in the longest path from the root to a leaf. The worst-case height of an unbalanced BST can be as high as 𝑂(𝑛)*O*(*n*) where 𝑛*n* is the number of nodes.

**Average Case Complexity**

The average time complexity for insertion, deletion, and search operations in a balanced BST is 𝑂(log⁡𝑛)*O*(log*n*).

**In-Order Traversal**

An in-order traversal of a BST yields the keys in non-decreasing order. This property allows for efficient sorting.

**Balanced vs. Unbalanced**

A BST can become unbalanced, leading to degraded performance. Balanced versions of BSTs (like AVL trees or Red-Black trees) maintain a balanced structure to ensure logarithmic height

**Implementation**

**BST Node and Tree Structure**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = right = nullptr;

}

};

class BST {

public:

Node\* root;

BST() {

root = nullptr;

}

void insert(int data) {

root = insertRec(root, data);

}

Node\* insertRec(Node\* node, int data) {

if (node == nullptr) {

return new Node(data);

}

if (data < node->data) {

node->left = insertRec(node->left, data);

} else if (data > node->data) {

node->right = insertRec(node->right, data);

}

return node;

}

bool search(int data) {

return searchRec(root, data);

}

bool searchRec(Node\* node, int data) {

if (node == nullptr || node->data == data) {

return node != nullptr;

}

if (node->data < data) {

return searchRec(node->right, data);

}

return searchRec(node->left, data);

}

void inOrder() {

inOrderRec(root);

}

void inOrderRec(Node\* node) {

if (node != nullptr) {

inOrderRec(node->left);

cout << node->data << " ";

inOrderRec(node->right);

}

}

};

int main() {

BST tree;

// Inserting values

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Print in-order traversal

cout << "In-order traversal: ";

tree.inOrder();

cout << endl;

// Search for values

int value = 40;

if (tree.search(value)) {

cout << value << " is found in the BST." << endl;

} else {

cout << value << " is not found in the BST." << endl;

}

value = 90;

if (tree.search(value)) {

cout << value << " is found in the BST." << endl;

} else {

cout << value << " is not found in the BST." << endl;

}

return 0;

}

**Conclusion**

The BST implementation in C++ effectively demonstrates the structure and operations of Binary Search Trees, making it easier for users to understand their properties and behaviors. This document serves as a guide to implement and visualize BST operations such as insertion and searching, offering a solid foundation for further exploration and learning.