Data Structures and Algorithms using C++

BY CIPHER SCHOOL

INTRODUCTION: In the summer internship with cipher school, data structure algorithm using c++ every important topics have been covered and the learnings have been implemented into this project.

This project focuses on the implementation of a Binary Search Tree (BST) in C++, a fundamental data structure that enables efficient data management through key operations such as insertion, searching, and deletion. Utilizing standard C++ libraries, the BST maintains its properties by organizing integers in a way that smaller values are placed in the left subtree and larger values in the right subtree Users can easily perform various operations, including inserting and deleting elements, searching for specific values, and displaying the tree's contents using different traversal methods.

OBJECTIVE: Implement Binary Search Tree with Following functions:

- Search

- Insert

- Delete

Overview:

The project is a Binary Search Tree (BST) implementation in C++, covering important key operations such as: Insertion, Deletion and Searching.

- Platforms: VsCode, GitHub.
- Tools and Libraries: Standard C++ libraries.
- Features and Functionalities:

1. Insert Elements: This function is used to insert values in a BST. Users can insert integers into the BST. The tree maintains its properties by placing smaller values in the left subtree and larger values in the right subtree.

2. Delete Elements: This function is used to delete the specified node from a BST. Users can delete an element from the BST. The deletion process maintains the structure of the tree, ensuring the BST properties are preserved.

3. Search Elements: This function is used to search a given key in BST. Users can search for a specific integer in the BST. The program will return whether
the element is found or not.

4. Display Elements: The program supports multiple traversal methods to display elements:

- In-order Traversal: Outputs elements in sorted order (left, root, right).
- Pre-order Traversal: Displays elements in pre-order sequence (root, left, right).
- Post-order Traversal: Displays elements in post-order sequence (left, right, root).

5. User-Friendly Menu: A simple text-based menu allows users to interact with the BST and choose various operations easily.
Usage Example:
The user interacts with the program through a menu that allows them to perform various operations on the

Binary Search Tree (BST):

Searching:

Searching for a value in a BST is an efficient process due to the tree's inherent structure. The search algorithm recursively compares the target value with the current node's value and decides whether to move to the left or right subtree based on the comparison result.

```
cpp
bool search(Node* root, int value) {
  if (root == nullptr || root->data == value) {
    return root;
  }
  if (value < root->data) {
    return search(root->left, value);
  }
  return search(root->right, value);
}
```

Example:

- To search for the value 15 in the BST:
  - Compare 15 with the root node's value (50). 15 < 50, so move to the left subtree.
  - Compare 15 with the left child's value (30). 15 < 30, so move to the left subtree.
  - Compare 15 with the left child's value (20). 15 > 20, so move to the right subtree.
  - Compare 15 with the right child's value (25). 15 < 25, so move to the left subtree.
  - 15 is found in the left child of 25.

Insertion

Inserting a value into a BST follows a similar recursive process as searching. The algorithm compares the target value with the current node's value and decides whether to insert it in the left or right subtree based on the comparison result.

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

Example:

- To insert the value 15 into the BST:
  - Compare 15 with the root node's value (50).
  - 15 < 50, so move to the left subtree.
  - Compare 15 with the left child's value (30).
  - 15 < 30, so move to the left subtree.
  - Compare 15 with the left child's value (20).
  - 15 > 20, so move to the right subtree.

- There is no right child, so insert 15 as the right child of 20.

Deletion

Deleting a value from a BST involves three cases:

1. **Node with no children (leaf node)**: Simply remove the node.
2. **Node with one child**: Replace the node with its child.
3. **Node with two children**: Find the in-order successor (smallest value in the right subtree) and replace the node with the successor. Then, delete the successor from the right subtree.

cpp

```
Node* deleteNode(Node* root, int value) {
  if (root == nullptr) {
    return root;
  }
  if (value < root->data) {
    root->left = deleteNode(root->left, value);
  } else if (value > root->data) {
    root->right = deleteNode(root->right, value);
  } else {
    if (root->left == nullptr) {
      return root->right;
    } else if (root->right == nullptr) {
      return root->left;
    }
    root->data = minValue(root->right);
    root->right = deleteNode(root->right, root->data);
  }
  return root;
}
```

Example:

- To delete the value 30 from the BST:
  - Compare 30 with the root node's value (50).
  - 30 < 50, so move to the left subtree.
  - Compare 30 with the left child's value (30).
  - 30 == 30, so this is the node to be deleted.
  - Since 30 has two children, find the in-order successor (smallest value in the right subtree).
  - The in-order successor is 35, so replace 30 with 35.
  - Delete the node with value 35 from the right subtree.

These examples demonstrate how the search, insertion, and deletion operations work in a Binary Search Tree. The recursive nature of these algorithms ensures efficient execution while maintaining the BST properties.

CONCLUSION:

This project showcases the implementation of a Binary Search Tree (BST) in C++, focusing on essential operations that are vital for comprehending data structures. By providing a user-friendly interface, this implementation allows for seamless interaction with the BST, making it an accessible learning tool for those new to the subject. The code serves as a valuable foundational exercise for individuals delving into the world of data structures and algorithms, as it demonstrates the practical application of a BST and its fundamental operations.