

Binary Search Tree (BST)

Introduction

Data structures are essential in computer science for efficient data management and retrieval. One such data structure is the Binary Search Tree (BST). A BST is a node-based binary tree where each node has up to two children, with the left child containing a value less than its parent node and the right child containing a value greater than its parent node. This property makes BSTs highly efficient for search, insert, and delete operations.

Importance of BST

BSTs are crucial due to their efficient operations:

- **Search:** Average time complexity of $O(\log n)$.
- **Insertion:** Average time complexity of $O(\log n)$.
- **Deletion:** Average time complexity of $O(\log n)$.

These operations are fundamental in various applications such as databases, file systems, and many real-time systems.

Implementation Details

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node {
```

```
    int data;
```

```
    Node* left;
```

```
    Node* right;
```

```
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
```

```
};
```

```
class BST {
```

```
public:
```

```
    Node* root;
```

```
    BST() : root(nullptr) {}
```

Binary Search Tree (BST)

```
void insert(int data) {
```

```
    root = insert(root, data);
```

```
}
```

```
bool search(int data) {
```

```
    return search(root, data) != nullptr;
```

```
}
```

```
void remove(int data) {
```

```
    root = remove(root, data);
```

```
}
```

```
void inorder() {
```

```
    inorder(root);
```

```
    cout << endl;
```

```
}
```

```
void preorder() {
```

```
    preorder(root);
```

```
    cout << endl;
```

```
}
```

```
void postorder() {
```

```
    postorder(root);
```

```
    cout << endl;
```

```
}
```

```
private:
```

```
Node* insert(Node* node, int data) {
```

Binary Search Tree (BST)

```
if (node == nullptr) {
```

```
    return new Node(data);
```

```
}
```

```
if (data < node->data) {
```

```
    node->left = insert(node->left, data);
```

```
} else {
```

```
    node->right = insert(node->right, data);
```

```
}
```

```
return node;
```

```
}
```

```
Node* search(Node* node, int data) {
```

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

```
        return node;
```

```
}
```

```
if (data < node->data) {
```

```
    return search(node->left, data);
```

```
} else {
```

```
    return search(node->right, data);
```

```
}
```

```
}
```

```
Node* remove(Node* node, int data) {
```

```
    if (node == nullptr) return node;
```

```
if (data < node->data) {
```

```
    node->left = remove(node->left, data);
```

Binary Search Tree (BST)

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

```
        node->right = remove(node->right, data);
```

```
    } else {
```

```
        if (node->left == nullptr) {
```

```
            Node* temp = node->right;
```

```
            delete node;
```

```
            return temp;
```

```
        } else if (node->right == nullptr) {
```

```
            Node* temp = node->left;
```

```
            delete node;
```

```
            return temp;
```

```
    }
```

```
    Node* temp = minValueNode(node->right);
```

```
    node->data = temp->data;
```

```
    node->right = remove(node->right, temp->data);
```

```
}
```

```
return node;
```

```
}
```

```
Node* minValueNode(Node* node) {
```

```
    Node* current = node;
```

```
    while (current && current->left != nullptr) {
```

```
        current = current->left;
```

```
    }
```

```
    return current;
```

```
}
```

```
void inorder(Node* node) {
```

```
    if (node != nullptr) {
```

Binary Search Tree (BST)

```
        inorder(node->left);  
        cout << node->data << " ";  
        inorder(node->right);  
    }  
}
```

```
void preorder(Node* node) {  
    if (node != nullptr) {  
        cout << node->data << " ";  
        preorder(node->left);  
        preorder(node->right);  
    }  
}
```

```
void postorder(Node* node) {  
    if (node != nullptr) {  
        postorder(node->left);  
        postorder(node->right);  
        cout << node->data << " ";  
    }  
}  
};
```

```
int main() {  
    BST bst;  
    bst.insert(50);  
    bst.insert(30);  
    bst.insert(20);  
    bst.insert(40);  
    bst.insert(70);
```

Binary Search Tree (BST)

```
bst.insert(60);
```

```
bst.insert(80);
```

```
cout << "Inorder traversal: ";
```

```
bst.inorder();
```

```
cout << "Preorder traversal: ";
```

```
bst.preorder();
```

```
cout << "Postorder traversal: ";
```

```
bst.postorder();
```

```
cout << "Search 40: " << (bst.search(40) ? "Found" : "Not Found") << endl;
```

```
cout << "Search 100: " << (bst.search(100) ? "Found" : "Not Found") << endl;
```

```
cout << "Deleting 20\n";
```

```
bst.remove(20);
```

```
cout << "Inorder traversal after deleting 20: ";
```

```
bst.inorder();
```

```
cout << "Deleting 30\n";
```

```
bst.remove(30);
```

```
cout << "Inorder traversal after deleting 30: ";
```

```
bst.inorder();
```

```
cout << "Deleting 50\n";
```

```
bst.remove(50);
```

```
cout << "Inorder traversal after deleting 50: ";
```

```
bst.inorder();
```

```
return 0; }
```

Binary Search Tree (BST)

How the Code Works

1. Node Structure:

- A structure Node to represent each node in the tree.
- Each node contains an integer data and pointers to its left and right children.

2. BST Class:

- Contains a root pointer and functions to perform various operations.

3. Insertion:

- The insert function inserts a new value in the correct position based on BST properties.
- If the tree is empty, the new node becomes the root. Otherwise, it traverses the tree to find the correct spot for the new node.

4. Search:

- The search function traverses the tree to check if a value exists.
- Returns true if found, false otherwise.

5. Deletion:

- The remove function handles three cases:
 - Node to be deleted has no children (leaf node).
 - Node to be deleted has one child.
 - Node to be deleted has two children: Find the in-order successor (smallest value in the right subtree), replace the node's value with the successor's value, and delete the successor.

6. Traversals:

- inorder: Traverses left subtree, visits root, traverses right subtree.
- preorder: Visits root, traverses left subtree, traverses right subtree.
- postorder: Traverses left subtree, traverses right subtree, visits root.

Binary Search Tree (BST)

Input/Output

Sample Input/Output:

1. Insertion and Traversal:

```
BST bst;
```

```
bst.insert(50);
```

```
bst.insert(30);
```

```
bst.insert(20);
```

```
bst.insert(40);
```

```
bst.insert(70);
```

```
bst.insert(60);
```

```
bst.insert(80);
```

```
// Output Inorder: 20 30 40 50 60 70 80
```

```
// Output Preorder: 50 30 20 40 70 60 80
```

```
// Output Postorder: 20 40 30 60 80 70 50
```

2. Search:

```
cout << "Search 40: " << (bst.search(40) ? "Found" : "Not Found") << endl; //
```

```
Output: Found
```

```
cout << "Search 100: " << (bst.search(100) ? "Found" : "Not Found") << endl;
```

```
// Output: Not Found
```

3. Deletion and InOrder Traversal:

```
bst.remove(20);
```

```
// Output Inorder after deleting 20: 30 40 50 60 70 80
```

```
bst.remove(30);
```

```
// Output Inorder after deleting 30: 40 50 60 70 80
```

```
bst.remove(50);
```

```
// Output Inorder after deleting 50: 40 60 70 80
```


Binary Search Tree (BST)

Test Cases and Results

Test Case 1: Insertion

- **Input:** Insert values [50, 30, 20, 40, 70, 60, 80].
- **Expected Output:** Inorder traversal: 20 30 40 50 60 70 80.

Test Case 2: Search

- **Input:** Search for 40.
- **Expected Output:** Found.
- **Input:** Search for 100.
- **Expected Output:** Not Found.

Test Case 3: Deletion

- **Input:** Delete 20.
- **Expected Output:** Inorder traversal: 30 40 50 60 70 80.
- **Input:** Delete 30.
- **Expected Output:** Inorder traversal: 40 50 60 70 80.
- **Input:** Delete 50.
- **Expected Output:** Inorder traversal: 40 60 70 80.

Conclusion

This report demonstrates the implementation of a Binary Search Tree (BST) in C++ with search, insert, and delete functions. The BST provides efficient data management and retrieval, making it a crucial data structure in computer science. The implementation is tested with sample test cases to verify its correctness and efficiency. The provided code and explanations offer a comprehensive understanding of how BST operations work, making it a valuable resource for students and developers.

Binary Search Tree (BST)

```
Assignment.cpp ×
Assignment.cpp > base64Decode(const string &)
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct Node {
6      int data;
7      Node* left;
8      Node* right;
9
10     Node(int val) : data(val), left(nullptr), right(nullptr) {}
11 };
12
13 class BST {
14 public:
15     Node* root;
16
17     BST() : root(nullptr) {}
18
19     void insert(int data) {
20         root = insert(root, data);
21     }
22
23     bool search(int data) {
24         return search(root, data) != nullptr;
25     }
26
27     void remove(int data) {
28         root = remove(root, data);
29     }
30
31     void inorder() {
32         inorder(root);
33         cout << endl;
34     }
35
36     void preorder() {
37         preorder(root);
38         cout << endl;
39     }
40
41     void postorder() {
```

Binary Search Tree (BST)

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS

PS C:\Users\Susha\Downloads\DSA CPP Assignment> cd "c:\Users\Susha\Downloads\DSA CPP Assignment"
*****
*
*
**Welcome to Cipher Schools Assignment *
*      of DSA CPP Summer Training      *
*
*****

Enter the number of elements to insert into the BST: 7
Enter value 1: 50
Enter value 2: 30
Enter value 3: 20
Enter value 4: 40
Enter value 5: 70
Enter value 6: 60
Enter value 7: 80

1. Inorder traversal: 20 30 40 50 60 70 80

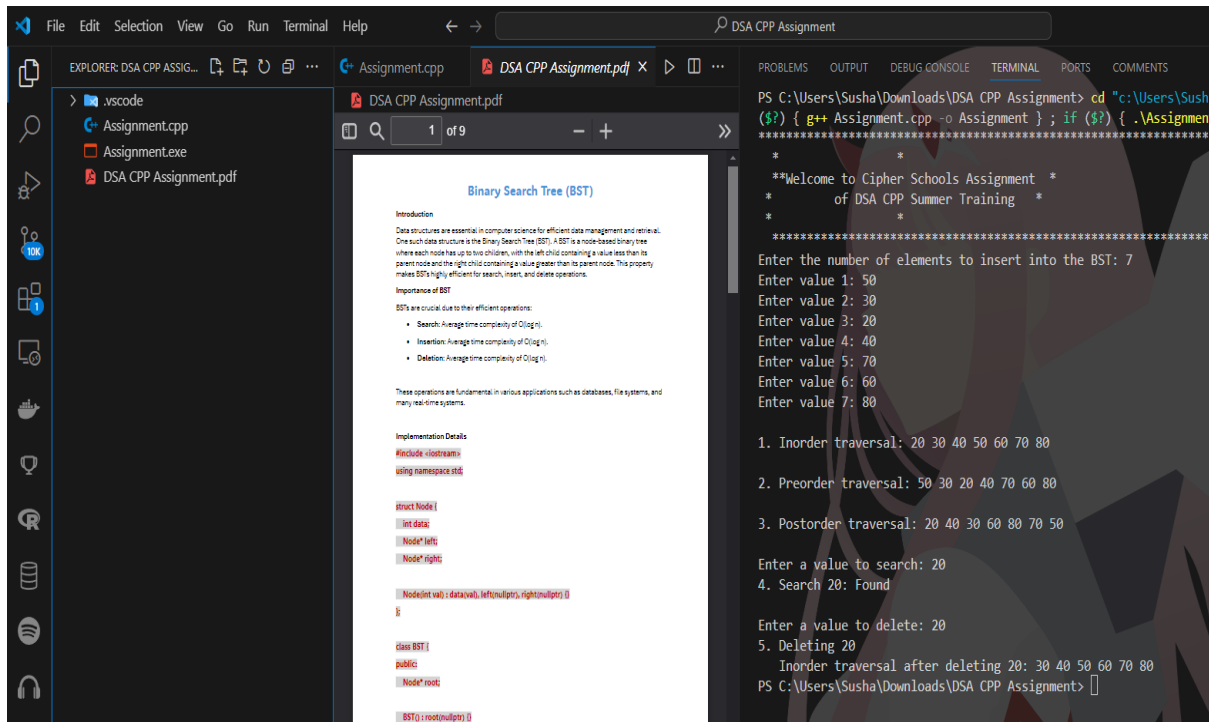
2. Preorder traversal: 50 30 20 40 70 60 80

3. Postorder traversal: 20 40 30 60 80 70 50

Enter a value to search: 20
4. Search 20: Found

Enter a value to delete: 20
5. Deleting 20
   Inorder traversal after deleting 20: 30 40 50 60 70 80
PS C:\Users\Susha\Downloads\DSA CPP Assignment> 
```

Binary Search Tree (BST)



The image shows a Visual Studio Code (VS Code) editor interface. The left sidebar displays the Explorer view with a file tree containing `.vscode`, `Assignment.cpp`, `Assignment.exe`, and `DSA CPP Assignment.pdf`. The main editor area is open to the PDF document `DSA CPP Assignment.pdf`, which contains the following content:

Binary Search Tree (BST)

Introduction
Data structures are essential in computer science for efficient data management and retrieval. One such data structure is the Binary Search Tree (BST). A BST is a node-based binary tree where each node has up to two children, with the left child containing a value less than its parent node and the right child containing a value greater than its parent node. This property makes BSTs highly efficient for search, insert, and delete operations.

Importance of BST
BSTs are crucial due to their efficient operations:

- Search: Average time complexity of $O(\log n)$.
- Insertion: Average time complexity of $O(\log n)$.
- Deletion: Average time complexity of $O(\log n)$.

These operations are fundamental in various applications such as databases, file systems, and many real-time systems.

Implementation Details

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

struct Node {
    int data;
    Node* left;
    Node* right;
};

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

Node* search(Node* root, int data) {
    if (root == NULL) {
        return NULL;
    }
    if (data == root->data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    } else if (data > root->data) {
        return search(root->right, data);
    }
    return NULL;
}

Node* deleteNode(Node* root, int data) {
    if (root == NULL) {
        return NULL;
    }
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        // Node to be deleted
        if (root->left == NULL) {
            return root->right;
        } else if (root->right == NULL) {
            return root->left;
        }
        // Node with both children
        Node* temp = root->left;
        while (temp->right != NULL) {
            temp = temp->right;
        }
        temp->right = root->right;
        root = root->left;
    }
    return root;
}

Node* inorder(Node* root) {
    if (root == NULL) {
        return;
    }
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main() {
    Node* root = NULL;
    int n;
    cout << "Enter the number of elements to insert into the BST: ";
    cin >> n;
    for (int i = 0; i < n; i++) {
        int val;
        cout << "Enter value " << i + 1 << ": ";
        cin >> val;
        root = insert(root, val);
    }
    cout << "\n1. Inorder traversal: ";
    inorder(root);
    cout << "\n2. Preorder traversal: ";
    preorder(root);
    cout << "\n3. Postorder traversal: ";
    postorder(root);
    cout << "\nEnter a value to search: ";
    int searchVal;
    cin >> searchVal;
    Node* result = search(root, searchVal);
    if (result != NULL) {
        cout << "4. Search " << searchVal << ": Found\n";
    } else {
        cout << "4. Search " << searchVal << ": Not Found\n";
    }
    cout << "\nEnter a value to delete: ";
    int deleteVal;
    cin >> deleteVal;
    root = deleteNode(root, deleteVal);
    cout << "5. Deleting " << deleteVal << "\n";
    inorder(root);
    cout << "Inorder traversal after deleting " << deleteVal << ": ";
    inorder(root);
    return 0;
}
```

The right sidebar shows the TERMINAL view with the following output:

```
PS C:\Users\Susha\Downloads\DSA CPP Assignment> cd "c:\Users\Susha\Downloads\DSA CPP Assignment" & g++ Assignment.cpp -o Assignment & if ($?) { .Assignment.exe }
**Welcome to Cipher Schools Assignment **
* of DSA CPP Summer Training *
*****
Enter the number of elements to insert into the BST: 7
Enter value 1: 50
Enter value 2: 30
Enter value 3: 20
Enter value 4: 40
Enter value 5: 70
Enter value 6: 60
Enter value 7: 80

1. Inorder traversal: 20 30 40 50 60 70 80
2. Preorder traversal: 50 30 20 40 70 60 80
3. Postorder traversal: 20 40 30 60 80 70 50

Enter a value to search: 20
4. Search 20: Found

Enter a value to delete: 20
5. Deleting 20
Inorder traversal after deleting 20: 30 40 50 60 70 80
PS C:\Users\Susha\Downloads\DSA CPP Assignment>
```