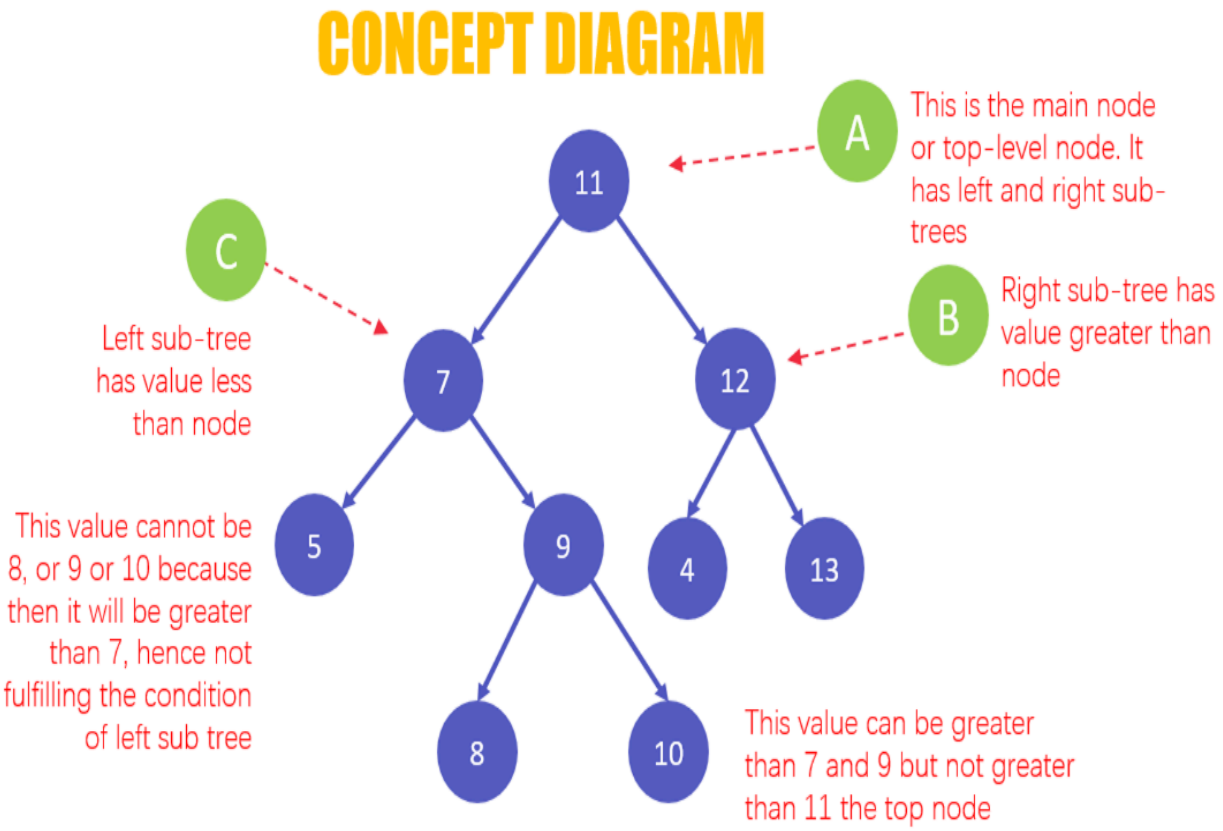
DSA Project

on:-

Binary Search Tree Implementation



Submitted By – Jaskaran Singh

Submitted to – Cipherschools.com

Project Details

Project Overview

This project is a C++ implementation of a Binary Search Tree (BST). A Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with values lesser than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- The left and right subtree each must also be a binary search tree.

Technologies and Tools

- **Programming Language**: C++
- **Development Environment**: Any C++ Integrated Development Environment (IDE) or Compiler (e.g., GCC, Visual Studio)
- **Libraries**: Standard Template Library (STL) for input and output operations

Features and Functionalities

1. **Node Creation**:
    - A function to create a new node with a given value.
2. **Insertion**:
    - Functionality to insert a new value into the BST while maintaining its properties.
3. **Search**:
    - A function to search for a specific value within the BST.
4. **Deletion**:

- Functionality to delete a node with a given value from the BST. The deletion process handles three cases:
  - Node with no children (leaf node).
  - Node with one child.
  - Node with two children.

5. **Traversal**:
   - **In-order Traversal**: Traverses the BST in ascending order of values.
   - **Pre-order Traversal**: Traverses the BST starting from the root node, then the left subtree, and finally the right subtree.
   - **Post-order Traversal**: Traverses the BST starting with the left subtree, then the right subtree, and finally the root node.

bstproject.cpp

C: > Users > user > Desktop > BST Project > bstproject.cpp

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

// Structure representing a node in the BST
struct Node {
    int value;
    Node* left;
    Node* right;
};

// Create a new node with the given value
Node* createNode(int val) {
    Node* newNode = new Node();
    newNode->value = val;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert  value
Node* insert(Node* node, int val) {
    if (node == nullptr) return createNode(val);

    if (val < node->value) {
        node->left = insert(node->left, val);
    } else {
        node->right = insert(node->right, val);
    }
    return node;
}

// Search value in the BST
Node* search(Node* node, int val) {
    if (node == nullptr || node->value == val) return node;

    if (val < node->value) return search(node->left, val);

    return search(node->right, val);
}
```

Code Functionality

Node Structure

- The `Node` struct defines the structure of a node in the BST, containing an integer value, and pointers to the left and right children.

Create Node

- The `createNode` function initializes a new node with a given value.

Insert Node

- The `insert` function recursively inserts a new value into the BST at the correct position.

Search Node

- The `search` function recursively searches for a value in the BST and returns the node if found.

Delete Node

- The `deleteNode` function deletes a node with a given value and ensures the BST properties are maintained. It also handles the deletion of nodes with no children, one child, or two children.

Traversals

- The `inOrder`, `preOrder`, and `postOrder` functions perform the respective tree traversals and print the values of the nodes.

Main Function

- The `main` function provides a menu-driven interface for the user to interact with the BST. It allows the user to insert, search, delete nodes, and perform various traversals on the BST. The program runs in a loop until the user decides to exit.

Example Usage

```
1. Insert
2. Search
3. Delete
4. InOrder Traversal
5. PreOrder Traversal
6. PostOrder Traversal
7. Exit
Enter your choice: 1
Enter value to insert: 22
1. Insert
2. Search
3. Delete
4. InOrder Traversal
5. PreOrder Traversal
6. PostOrder Traversal
7. Exit
Enter your choice: 2
Enter value to search: 22
Value found.
1. Insert
2. Search
3. Delete
4. InOrder Traversal
5. PreOrder Traversal
6. PostOrder Traversal
7. Exit
```

```
Enter your choice: 6
PostOrder Traversal: 99 66 44 22
1. Insert
2. Search
3. Delete
4. InOrder Traversal
5. PreOrder Traversal
6. PostOrder Traversal
7. Exit
Enter your choice: 4
InOrder Traversal: 22 44 66 99
1. Insert
2. Search
3. Delete
4. InOrder Traversal
5. PreOrder Traversal
6. PostOrder Traversal
7. Exit
Enter your choice:
```

- Users can choose an option from the menu to perform the desired operation on the BST. For example, choosing option 1 allows the user to insert a new value into the BST.

Use Cases of Binary Search Tree (BST) in Industry

1. **Database Indexing**:
    - **Use Case**: Efficient data retrieval
    - **Description**: BSTs can be used to implement database indexes, allowing quick look-up, insertion, and deletion operations. By organizing data hierarchically, BSTs improve the performance of query operations, which is crucial for large databases

      .

2. **Filesystem and Directory Management**:
    - **Use Case**: File search and organization
    - **Description**: Filesystems often use tree structures, including BSTs, to manage and organize files and directories. This allows for efficient searching, inserting, and deleting of files and directories.

3. **Memory Management**:
    - **Use Case**: Efficient allocation and deallocation of memory
    - **Description**: BSTs are used in memory management systems to keep track of free and allocated memory blocks. This helps in efficiently finding free blocks of the required size and merging adjacent free blocks.

4. **Network Routing Algorithms**:
    - **Use Case**: Optimal path finding
    - **Description**: BSTs can be utilized in network routing algorithms to find the most efficient path for data

transmission. They help in managing and updating the routing tables dynamically.

5. **Compiler Design**:
    - **Use Case**: Syntax analysis and optimization
    - **Description**: Compilers use BSTs in various phases, such as syntax analysis (parsing) and optimization. Abstract Syntax Trees (ASTs), a type of binary tree, are used to represent the structure of source code.

6. **Priority Queues**:
    - **Use Case**: Task scheduling
    - **Description**: BSTs can be used to implement priority queues, which are essential for task scheduling in operating systems. They allow for efficient extraction of the highest or lowest priority element.

7. **Autocomplete and Spell Check**:
    - **Use Case**: Fast and efficient suggestions
    - **Description**: BSTs, specifically in the form of Ternary Search Trees (a variant of BST), are used in autocomplete systems and spell checkers to provide quick suggestions and corrections.

8. **Game Development**:
    - **Use Case**: Real-time searching and decision making
    - **Description**: BSTs are used in game development for managing various in-game elements, such as object hierarchies, spatial partitioning, and AI decision trees, allowing for efficient real-time searching and decision making.

9. **Load Balancing**:
    - **Use Case**: Distributing tasks evenly
    - **Description**: BSTs can help in load balancing by keeping track of the load on different servers and ensuring that tasks are evenly distributed. This is crucial for maintaining system performance and avoiding overload on any single server.

10. **Search Engines**:
    - **Use Case**: Indexing web pages
    - **Description**: BSTs are used in search engines to index web pages and provide fast search results. They help in organizing and retrieving relevant web pages based on user queries efficiently.

Conclusion

Binary Search Trees are a fundamental data structure with numerous applications across various industries. Their ability to organize data hierarchically and perform quick search, insertion, and deletion operations makes them indispensable for optimizing performance in real-world applications.