Data Structure Algorithms & Applications (CT-159)
Lab 09

Binary Search Tree

Objectives

The objective of the lab is to get students familiar with Binary Search Tree and its operations i.e., insert, remove, search, traversal.

Tools Required

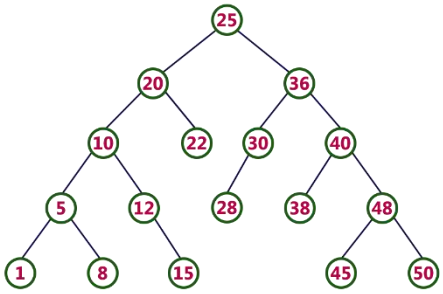
Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Binary Search Tree

A **Binary Search Tree (BST)** is a type of binary tree where each node has at most two children, commonly referred to as the left child and the right child. For each node:

- **Left Subtree**: All the nodes in the left subtree have values less than the node's value.
- **Right Subtree**: All the nodes in the right subtree have values greater than the node's value.
- Both left and right subtrees must also be binary search trees.

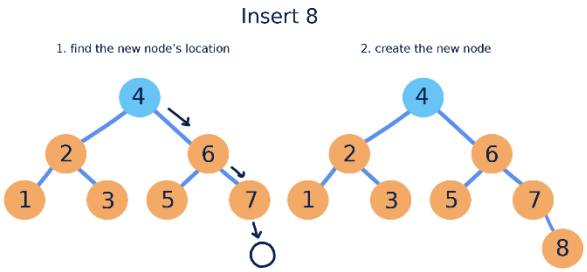


Properties:

- **Binary Tree Structure**: A BST is a special case of a binary tree where the order property is maintained.
- **In-Order Traversal**: In-order traversal of a BST yields the values in **sorted (ascending) order**.
- **Time Complexity**: **Search, Insert, and Delete**: On average, these operations are O(logn), where n is the number of nodes. However, in the worst case (if the tree becomes unbalanced), the time complexity is O(n).
- **Height**: The height of a balanced BST is O(logn), but it can degrade to O(n) in a degenerate case (if the tree becomes skewed).

Insert a Node in a BST:

If root is nullptr: Create a new node with value
If value < root.data: Set root.left to insertNode(root.left, value) (recursively insert in the left subtree)
Else If value > root.data: Set root.right to insertNode(root.right, value) (recursively insert in the right subtree)



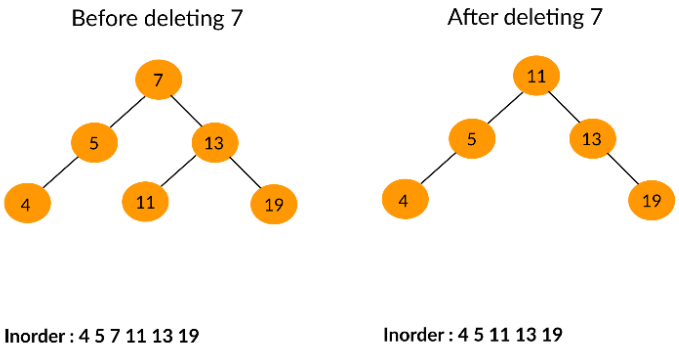
Delete a Node from a BST:

Find the Node: Start from the root and traverse the tree based on the value you want to delete.

- Move left if the value is smaller.
- Move right if the value is larger.

Delete the Node:

- **Case 1**: Node has **no children**: Simply remove the node.
- **Case 2**: Node has **one child**: Replace the node with its only child.
- **Case 3**: Node has **two children**: Find the **inorder successor** (smallest value in the right subtree), copy its value to the node to be deleted, and recursively delete the inorder successor.



Example 01: Implementation of a binary search tree.

```
#include <iostream>
using namespace std;
class TreeNode {
public:
   int val;
   TreeNode* left;
   TreeNode* right;
   TreeNode(int value) : val(value), left(NULL), right(NULL) {}
};

class BinarySearchTree {
public:
   TreeNode* root;
   BinarySearchTree() : root(NULL) {}

   void insert(int key) {
      root = insertRec(root, key);
   }
   void deleteNode(int key) {
      root = deleteRec(root, key);
   }

   void inorder() {
      inorderRec(root);
      cout << endl;
   }

private:
   // Recursive function to insert a node
   TreeNode* insertRec(TreeNode* node, int key) {
      // Base case: if the tree is empty, create a new node
      if (node == NULL)
         return new TreeNode(key);

      // Otherwise, recur down the tree
      if (key < node->val)
         node->left = insertRec(node->left, key);
      else if (key > node->val)
         node->right = insertRec(node->right, key);

      return node;
   }

   // Recursive function to delete a node
   TreeNode* deleteRec(TreeNode* node, int key) {
      // Base case: if the tree is empty
      if (node == NULL)
         return node;

      // Recur down the tree
      if (key < node->val)
         node->left = deleteRec(node->left, key);
      else if (key > node->val)
         node->right = deleteRec(node->right, key);
      else {
         // Node with only one child or no child
         if (node->left == NULL) {
            TreeNode* temp = node->right;
            delete node;
            return temp;
         } else if (node->right == NULL) {
            TreeNode* temp = node->left;
            delete node;
            return temp;
         }

         // Node with two children: Get the inorder successor (smallest in the right subtree)
         TreeNode* temp = minValueNode(node->right);
         // Copy the inorder successor's content to this node
         node->val = temp->val;
```

```
        // Delete the inorder successor
        node->right = deleteRec(node->right, temp->val);
    }
    return node;
  }
  // Find the node with the minimum value (used for deleting nodes)
  TreeNode* minValueNode(TreeNode* node) {
    TreeNode* current = node;
    while (current && current->left != NULL)
      current = current->left;
    return current;
  }
  // Inorder traversal function
  void inorderRec(TreeNode* root) {
    if (root != NULL) {
      inorderRec(root->left);
      cout << root->val << " ";
      inorderRec(root->right);
    }
  }
};
int main() {
  BinarySearchTree bst;
  bst.insert(50);
  bst.insert(30);
  bst.insert(20);
  bst.insert(40);
  bst.insert(70);
  bst.insert(60);
  bst.insert(80);
  cout << "Inorder traversal of the tree: ";
  bst.inorder();
  cout << "Delete 20\n";
  bst.deleteNode(20);
  cout << "Inorder traversal of the tree after deleting 20: ";
  bst.inorder();
}
```

Issues with BST

- **Unbalanced Tree**: In the worst case (e.g., inserting sorted data), the tree can become unbalanced, degrading to a linked list, which results in O(n) time complexity for operations.
- **Rebalancing Required**: To maintain efficiency, balancing techniques such as AVL or Red-Black Trees are often used, but these come with added complexity.
- **Memory Usage**: BSTs use more memory than arrays or linked lists due to the storage overhead for pointers (two for each node).

Computational Complexity Comparison of Data Structures covered so far

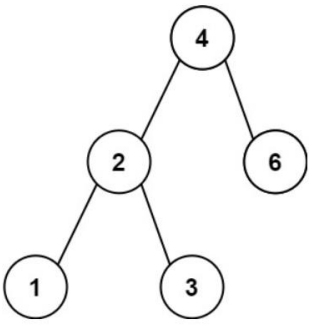
Data Structure	Insertion	Deletion	Search	Access (Indexing)
Linked List	O(1) (at head/tail)	O(n) (unsorted)	O(n)	O(n)
Array	O(n) (due to shifting)	O(n) (due to shifting)	O(n) (unsorted)	O(1)
Binary Tree	O(n)	O(n)	O(n)	N/A (no direct indexing)
Binary Search Tree	O(h) = O(log n) (balanced), O(n) (unbalanced)	O(h) = O(log n) (balanced), O(n) (unbalanced)	O(h) = O(log n) (balanced), O(n) (unbalanced)	N/A (no direct indexing)

Exercise

1. Define following methods in Example 01 by using stack.
    - Preorder traversal
    - Postorder travsersal
2. You are tasked with designing an employee management system for a small company. Each employee has a unique ID, name, and department. You are required to store and manage employee records in a way that allows quick insertion, deletion, and search operations based on the employee ID. You decide to use a Binary Search Tree (BST) to store the employee records. Each node in the BST will store: Employee ID (used as the key for the BST), Employee Name, Employee Department.

- Create a C++ class EmployeeNode representing each employee in the BST. Each node should store: int id (Employee ID), string name (Employee Name), string department (Employee Department), A pointer to the left child (EmployeeNode* left), a pointer to the right child (EmployeeNode* right).
- Create a class EmployeeBST with the following member functions:
    - insert(int id, string name, string department): Inserts a new employee into the BST.
    - search(int id): Searches for an employee by ID. If found, return their name and department; otherwise, print an appropriate message.
    - deleteNode(int id): Deletes an employee from the BST based on their ID.
    - inOrderTraversal(): Prints all employees in ascending order of their ID, showing their ID, name, and department.
    - findMin(): Returns the name and department of the employee with the smallest ID.
    - findMax(): Returns the name and department of the employee with the largest ID.
- Also, Handle edge cases such as: Inserting an employee with a duplicate ID, Deleting an employee that does not exist, Deleting nodes with no children, one child, and two children, Handling an empty tree for search or delete operations.

3. Given the two nodes of a binary search tree, return their least common ancestor.
4. Given the root of a binary search tree, recursively find the sum of all nodes of the tree.
5. Given the root of a Binary Search Tree (BST), return the minimum difference between the values of any two different nodes in the tree.




```
Input: root = [4,2,6,1,3]
Output: 1
```

Lab 09 Evaluation		
Student Name:	**Student ID:**	**Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		