EC-200 Data Structures

Lab Manual 09

**Course Instructor:** Lecturer Anum Abdul Salam

**Lab Engineer:** Lab Engineer Ayesha Batool

**Student Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Degree/ Syndicate: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Trait** | **Obtained Marks** | **Maximum Marks** |
| **R1** | **Application Functionality 20%** |  | 20 |
| **R2** | **Specification & Data structure implementation**  **30%** |  | 30 |
| **R3** | **Reusability**  **10%** |  | 10 |
| **R4** | **Input Validation**  **10%** |  | 10 |
| **R5** | **Efficiency**  **20%** |  | 20 |
| **R6** | **Delivery**  **10%** |  | 10 |
| **R7** | **Plagiarism above 80%** |  | 1 |
|  | **Total** |  | 10 |

**Total Marks = O**𝒃𝒕𝒂𝒊𝒏𝒆𝒅 𝑴𝒂𝒓𝒌𝒔 (∑6𝟏 𝑹𝒊 ∗ 𝑹7)

# LAB # 09: BINARY SEARCH TREES

## Lab Objective:

To Implement Binary Search tree ADT.

## Lab Description:

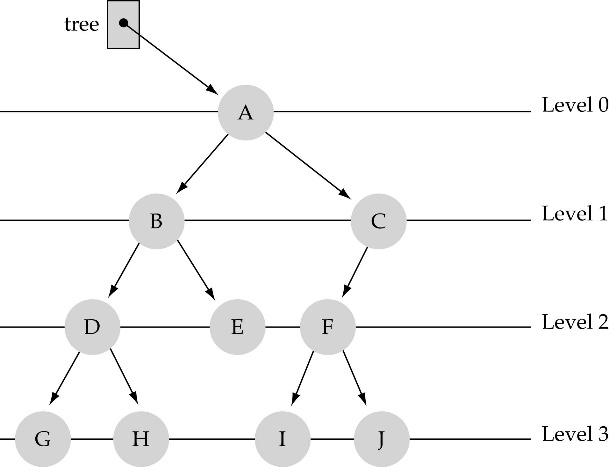
Complexity of data Structures we have previously learned and practiced is linear i.e., O(N). In case of linear complexity, size of data structure determines the time and space consumed by different operations performed in data structure. To reduce the complexity Binary Search trees are being used. Binary search trees are so called because each node can have up to two successor nodes. In Binary Search trees a parent node’s left child value should be small than parent node and right-side value should be large than parent node.

**Left child data Right child**



**Figure 9.1:** Tree node structure

* The predecessor node of a node is called its parent.
* The "beginning" node is called the root (no parent).
* A node without children is called a leaf.
* **Size** of tree is the number of nodes in the tree.



**Figure 9.2:** Level of Binary search tree

The **level** of a node N in a tree is the length of the path from root to N. Root is at level 0. The **depth** of a tree is the length of the longest path from root to any node.



**Figure 9.3:** Binary Search trees

**Types of Binary Search Trees**

1. **Full binary tree:** a binary tree is which each node was exactly 2 or 0 children.
2. **Complete binary tree:** a binary tree in which every level, except possibly the deepest is completely filled. At depth n, the height of the tree, all nodes are as far left as possible
3. **Perfect binary tree:** a binary tree with all leaf nodes at the same depth. All internal nodes have exactly two children. A perfect binary tree has the maximum number of nodes for a given height a perfect binary tree has 2(n+1) - 1 nodes where n is the height of a tree.

**Operations in Binary Search Tree (BST) ADT**

Some of the operations in BST ADT are

1. Insert
2. Delete
3. Traverse
4. Search
5. isEmpty

**Insertion**

Maintain key property (invariant).

1. Smaller values in left subtree.
2. Larger values in right subtree

**Algorithm:**

1. Perform search for value X
2. Search will end at node Y (if X not in tree)
3. If X < Y, insert new leaf X as new left subtree for Y
4. If X > Y, insert new leaf X as new right subtree for Y

|  |  |
| --- | --- |
| Insert in BST | |
| 10 < 20, right  30 > 20, left  25 > 20, left  Insert 20 on left  void insert (Node\* root, int key)  {      // if the root is null, create a new node and return it      // if key is less than the root node (key < root->data), recur for the left subtree (insert(root->left, key));      // if key is greater than the root node (key > root->data), recur for the right subtree (insert(root->right, key))  } |  |

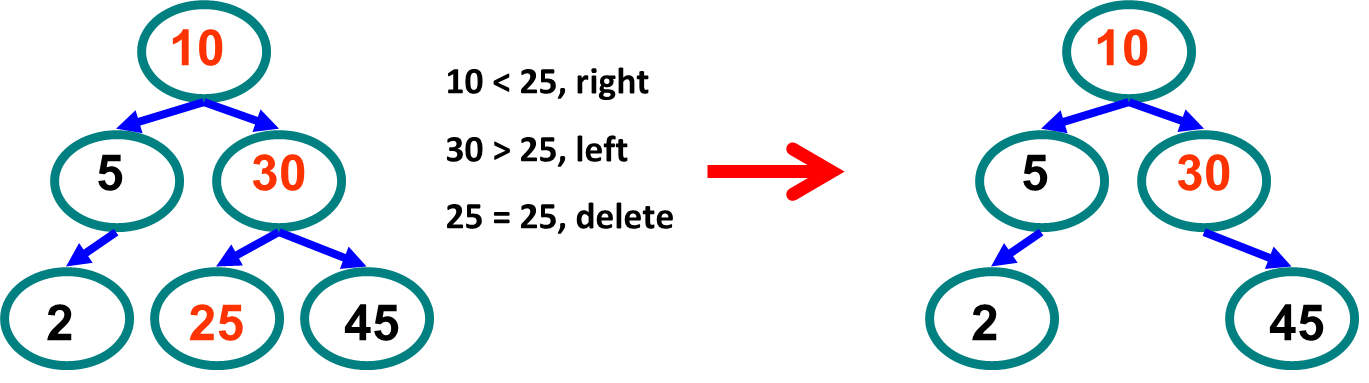
**Deletion**

Deletion in BST should be performed such that the remaining BST do not gets disturbed. In BST deletion three conditions should be checked. Node to be deleted is

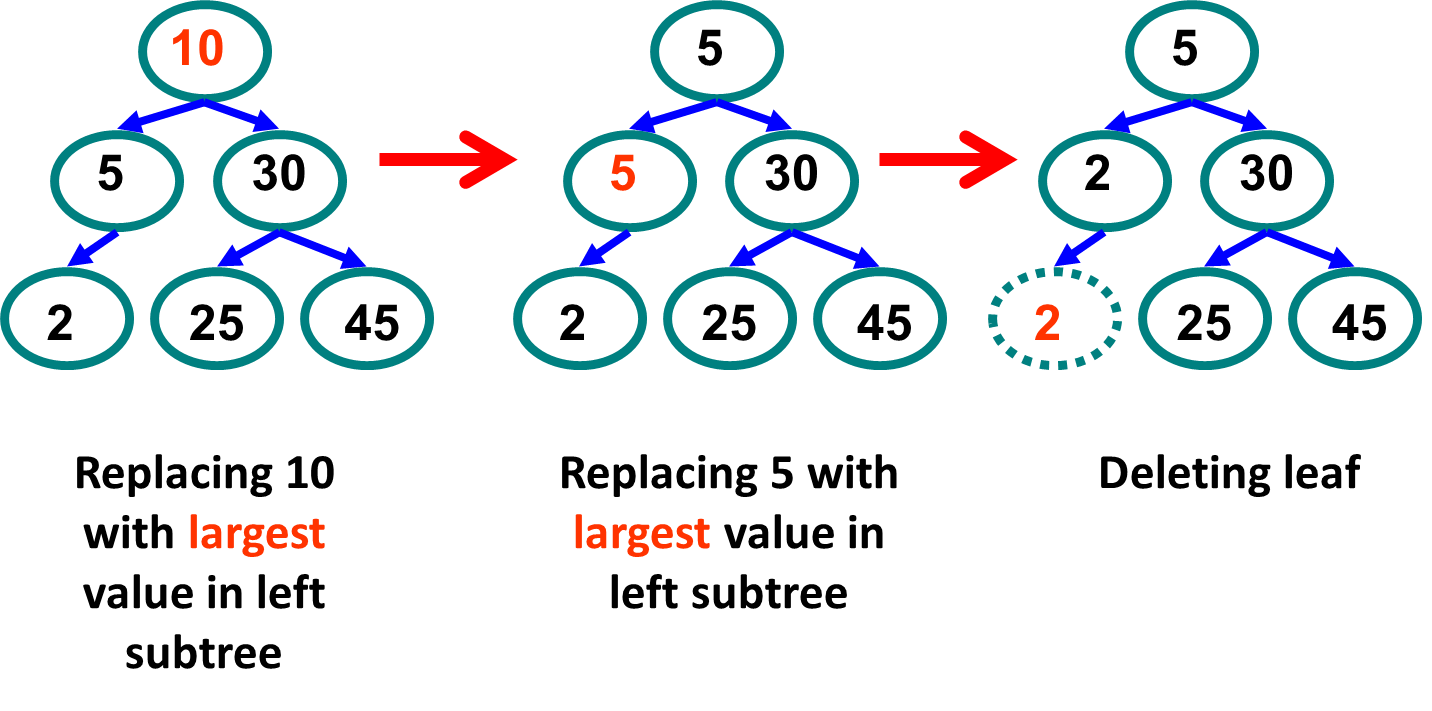
1. A parent node with two Childs
2. A parent node with one Child
3. A leaf node

**Algorithm:**

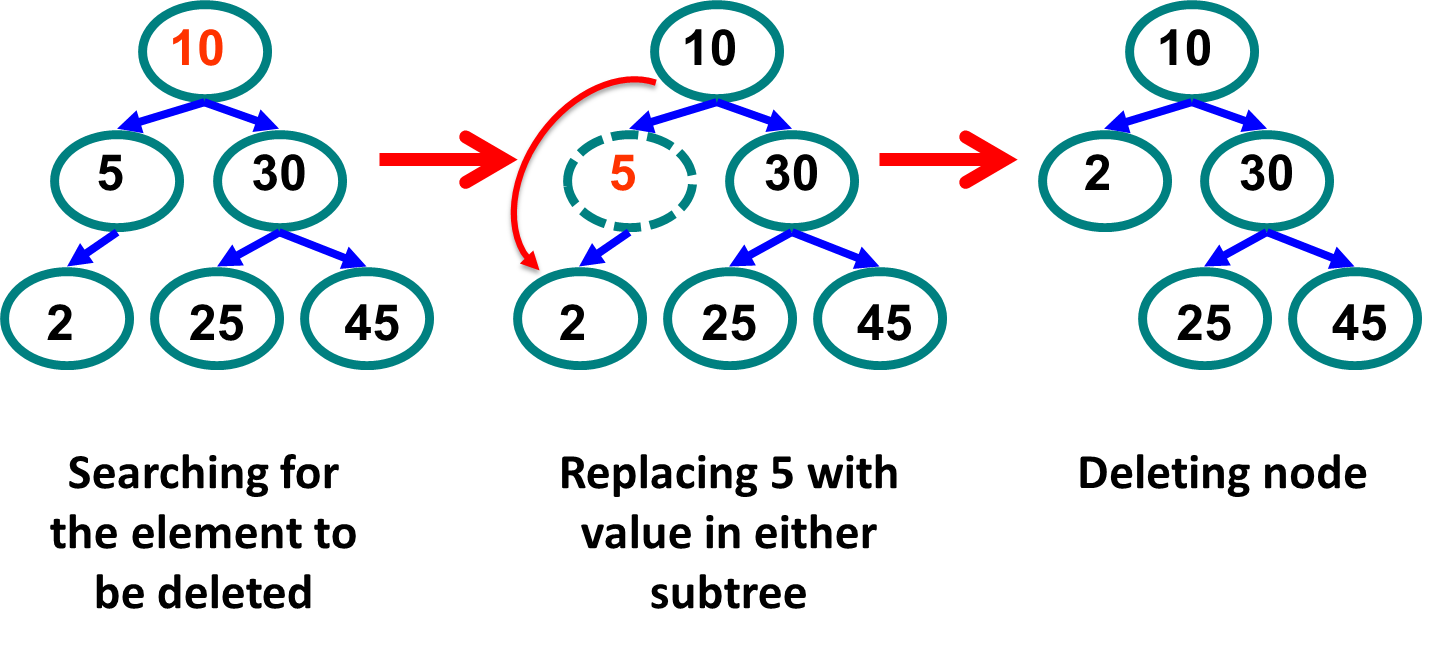
1. Perform search for value X
2. If X is a leaf, delete X
3. Else // must delete internal node
   * 1. Replace with largest value Y on left subtree OR smallest value Z on right subtree
     2. Delete replacement value (Y or Z) from subtree



**Figure 9.4:** Deleting a leaf node from BST



**Figure 9.5:** Deleting an internal node (node with two Childs)



**Figure 9.6:** Deleting an internal node (node with one Child)

|  |
| --- |
|  |

**Search**

Search in BST includes searching a particular value (if exists?) and return address of the node containing value.

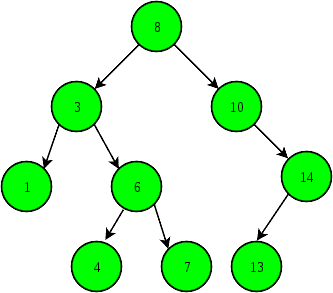
**Algorithm:**

|  |
| --- |
| struct node\* search (struct node\* root, int key)  {      // Base Cases: if root is null or key is present at root, return root;      // if key is greater than root's key, return search(root->right, key);      // Key is smaller than root's key, return search(root->left, key);  } |

**Traversal**

Traversal of BST Includes

1. Inorder (Traverse the Left subtree, Visit the root, Traverse the right subtree)
2. Preorder (Visit the root, Traverse the Left subtree, Traverse the right subtree)
3. Postorder (Traverse the Left subtree, Traverse the right subtree, Visit the root)



**Figure 9.7:** In order tree traversal

**void Postorder(Node \* root)**

**{**

**If (root!=NULL) {**

**Postorder(root->Left);**

**Postorder(root->Right);**

**cout>>root->data;}**

**}**

**void Preorder(Node \* root)**

**{**

**If (root!=NULL) {**

**cout>>root->data;**

**Preorder(root->Left);**

**Preorder(root->Right); }**

**}**

**void Inorder(Node \* root)**

**{**

**if (root!=NULL) {**

**Inorder(root->Left);**

**cout>>root->data;**

**Inorder(root->Right);}**

**}**

## LAB TASKS

Design Tree ADT and provide all the above functionalities including insertion, deletion, searching, traversing (Inorder, preorder, postorder) and isEmpty.

**TEST PLAN:**

1. **Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Operations** | **Expected Results** | **Results/Status** |
| **1.** | Create an empty tree **myTree** | root=NULL |  |
| **2.** | Insert numbers 21, 22, 10, 19, 17 |  |  |
| **3.** | Inorder Traverse **myTree**  Preorder Traverse **myTree**  Postorder Traverse **myTree** | 10,17,19,21,22  21 10 19 17 22  17 19 10 22 21 |  |
| **4.** | Search 6 in the tree | Element **not** found |  |
| **5.** | Search 21 in the tree | Element found |  |
| **6.** | Insert 22 in the tree | Number already exists |  |
| **7.** | Delete 78 from the tree | Element not found |  |
| **8.** | Delete 10 from the tree |  |  |
| **9.** | Inorder Traverse **myTree** | 17,19,21,22 |  |
| **10.** | Check if **myTree** is empty? | Not empty |  |
| **11.** | Create a tree **copyTree** from **myTree** (referring to copy constructor-> deep copy) |  |  |
| **12.** | Check if **copyTree** is empty? | Not empty |  |
| **13.** | Delete 17 from **copyTree** |  |  |
| **14.** | Inorder Traverse **myTree** | 17,19,21,22 |  |
| **15.** | Inorder Traverse **copyTree** | 19,21,22 |  |
| **16.** | Delete 19 from **copyTree** |  |  |
| **17.** | Delete 21 from **copyTree** |  |  |
| **18.** | Delete 22 from **copyTree** |  |  |
| **19.** | Inorder Traverse **copyTree** | Tree is empty |  |
| **20.** | Inorder Traverse **myTree** | 17,19,21,22 |  |
| **21.** | Preorder Traverse **myTree** | 21 19 17 22 |  |
| **22.** | Postorder Traverse **myTree** | 17 19 22 21 |  |

**THINK?**

1. Think & write at least 3 scenarios where trees are preferable than lists. Why?
2. What will be the complexity of a balanced binary search tree?
3. If following data is to be entered in a BST in the sequence given, write the complexity of searching in a tree. Consider worst case scenario?29,30,45,50,60,65,70
4. Compare the efficiency of searchIter & searchRecur functions in BST ADT.