Data Structures Lab 8

Course: Data Structures (CL2001) **Semester:** Fall 2023
Instructor: Shafique Rehman **T.A:** N/A

Note:

- Lab manual cover following below Stack and Queue topics
  {Tree, BST, Design and implement classes for binary tree nodes and nodes for general tree, Traverse the tree with the three common orders, Operation such as searches, insertions, and removals on a binary search tree and its applications}
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

BINARY SEARCH TREE

KeyPoint: A Binary Search Tree (BST) is a binary tree with the following properties:

- The left subtree of a particular node will always contain nodes whose keys are less than that node's key.
- The right subtree of a particular node will always contain nodes with keys greater than that node's key. The left and right subtree of a particular node will also, in turn, be binary search trees

BST Insertion, Deletion and Inorder Traversal

Sample Code of class Nodes

```
// Binary Search Tree operations in Java
class Node {
    int data;
    Node left, right;

    public Node(int item) {
      data = item;
      left = right = null;
    }
  }
class BST{

  Node root;

  BST() {
    root = null;
  }

  void insert(int data) {
```

```
    root = insertVal(root, data);
  }

  // Insert key in the tree
  insertVal(Node root, int data) {

    if (root == null) {
      root = new Node(data);
      return root;
    }
    // Traverse to the left place and insert the node
    if (data < root.data)
      root.left = insertVal(root.left, data);
    // Traverse to the right place and insert the node
    else if (data > root.data)
      root.right = insertVal(root.right, data);

    return root;
  }
// In order Traversal
  void inorder() {
    inorderVisit(root);
  }

  // Inorder Traversal
  void inorderVisit(Node root) {
    if (root != null) {
      inorderVisit(root.left);
      System.out.print(root.data + " -> ");
      inorderRec(root.right);
    }
  }
// deletion in BST
  void delete(int data) {
    root = deleteVal(root, data);
  }

  deleteVal(Node root, int data) {
    // Return if the tree is empty
    if (root == null)
      return root;

    // Find the node to be deleted
    if (data < root.data)
      root.left = deleteVal(root.left, data);
    else if (data > root.data)
```

```
      root.right = deleteVal(root.right, data);
    else {
      // If the node is with only one child or no child
      if (root.left == null)
        return right node;
      else if (root.right == null)
        return left node;

      // If the node has two children
      // Place the inorder successor in position of the node to be deleted
      root.data = minValue(root.right);
      // Delete the inorder successor
      root.right = deleteVal(root.right, root.data);
    }

    return root;
  }

  // Find the inorder successor
  int minValue(Node root) {
    find the min value in the left tree of the right of the node to be
deleted
    }
    return minv;
  }
}

class Lab8_BST {
    public static void main(String[] args) {
        BST bst = new BST();
        bst.insert(8);
        bst.insert(3);
        bst.insert(1);
        bst.insert(6);
        bst.insert(7);
        bst.insert(10);
        bst.insert(14);
        bst.insert(4);

        System.out.print("Inorder traversal: ");
        bst.inorder();

        System.out.println("\n\nAfter deleting 10");
        bst.delete(10);
        System.out.print("Inorder traversal: ");
        bst.inorder();
```

```
    }
}
```

Tree Traversals: Inorder, PreOrder, PostOrder

Pseudo code/steps For Inorder Traversal (iteration)

Step 1: if root is not null
    Step 1.1: recursively call the method on left child inorder(node.right)
    Step 1.2: then print the data(node.data)
    Step 1.3: then recursively call the method on right child inorder(node.right)

Note: when both node.left and node.right will get null values, then both recursive calls will be ended

Pre-Order Traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE.DATA
Step 3: PREORDER(TREE.LEFT)
Step 4: PREORDER(TREE.RIGHT)
[END OF LOOP]
Step 5: END
```

Post-Order Traversal

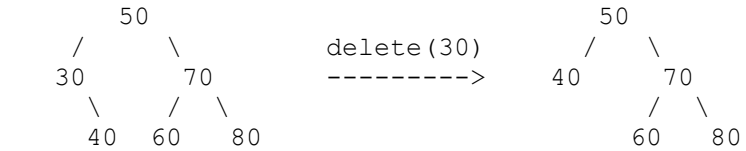
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: POSTORDER(TREE.LEFT)
Step 3: POSTORDER(TREE.RIGHT)
Step 4: Write TREE.DATA
[END OF LOOP]
Step 5: END
```

BST Deletion

BST Deletion

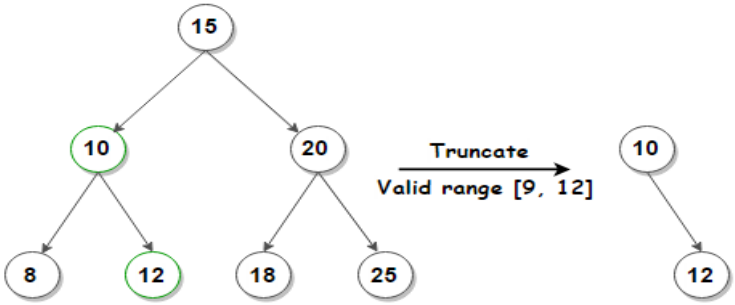
1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child



3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

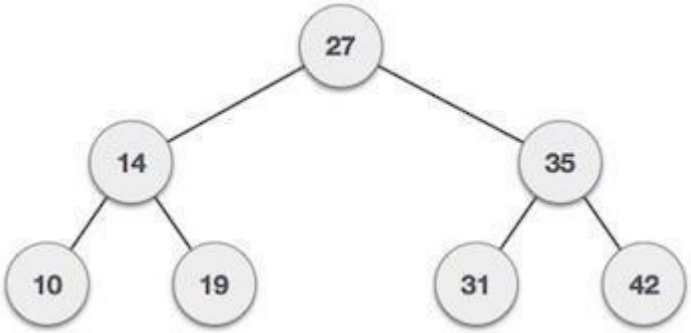


Some points to Note:

Binary search tree (BST)

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

- Each node has a value.
- The key value of the left child of a node is less than to the parent's key value.
- The key value of the right child of a node is greater than (or equal) to the parent's key value.
- And these properties hold true for every node in the tree.



- **Subtree**: any node in a tree and its descendants.
- **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.
- **Depth of a tree**: the maximum depth of any of its leaves.
- **Height of a node**: the length of the longest downward path to a leaf from that node.
- **Full binary tree**: every leaf has the same depth and every nonleaf has two children.
- **Complete binary tree**: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- **Traversal**: an organized way to visit every member in the structure.

Traversals

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.
The algorithms are described below, with Node initialized to the tree's root.

• Preorder Traversal

1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

• In-order Traversal

1. Traverse Node's left sub-tree.
2. Visit Node.
3. Traverse Node's right sub-tree

• Post-order Traversal

1. Traverse Node's left sub-tree.
2. Traverse Node's right sub-tree.
3. Visit Node

Searching

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

Algorithm:

```
If root == NULL
   return NULL;
If number == root.data
   return root.data;
If number < root.data
   return search(root.left)
If number > root.data
   return search(root.right)
```

Lab Tasks:

Q1: Given an array of {45, 10, 7, 90, 12, 50, 13, 39, 57} implement a binary tree accordingly.
Q2: Using the same tree that you made insert the values 6, 8, 9 accordingly. Now check if the tree made is a complete binary tree or if it's a full binary tree if not delete the nodes accordingly to make it a complete and full binary tree.

Q3: Search for the value defined by the user in the tree. If the value does not exist insert it then identify its location based on the level of the tree and if it's the root at level, left child or right child.

Q4: Given a Tree of nodes (Down below)

You are tasked to take a value X from the user and ask whether to ceil or floor the value. If ceil is selected find the closest value to X in our case 7 at level three right child because if we ceil 6 we get 7 (Yes I am familiar with the logic of how ceil and floor works with floating

points but this scenario is different so in the case of ceil add 1 to X and in the case of floor subtract 1 from x then find the value). Also find multiple occurrences if it as well.
Q5: Given two BST's (Down Below) You are tasked to merge them together to form a new BST the output of this will be 1 2 2 3 3 4 5 6 6 7 (You can visualize this by drawing the array to a tree in your notebook)