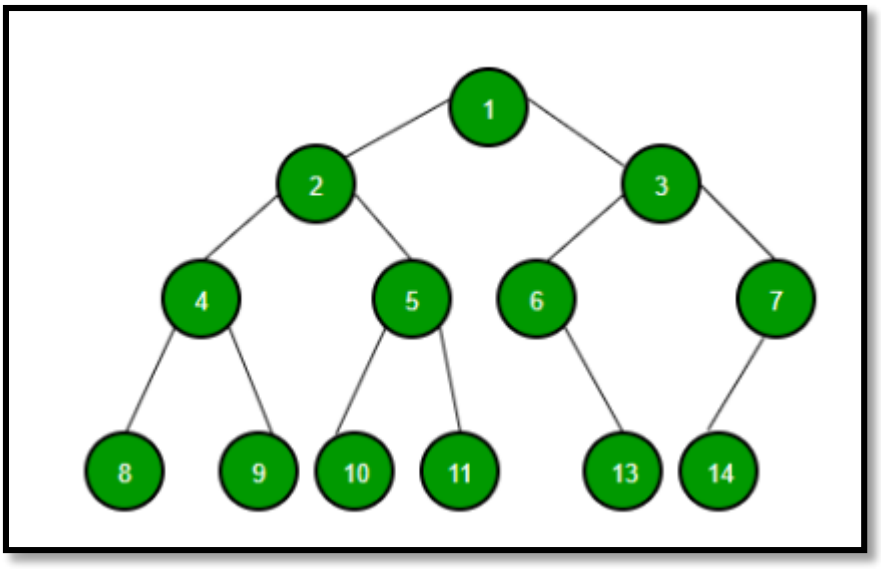
Course Code: CL-2001	Course : Data Structures - Lab
Instructor(s) :	Abeer Gauher, Sobia Iftikhar

LAB - 9

Binary Trees and Binary Search Trees

Binary Trees

Binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



Binary Tree Representation

A Binary tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL.

Binary Tree node contains the following parts:

1. Data
2. Pointer to left child
3. Pointer to right child

Basic Operation On Binary Tree:

1. Inserting an element.
2. Removing an element.
3. Searching for an element.

Using Linked List to implement trees:

```
class Node {
   int value;
   Node left;
   Node right;

   Node(int value) {
      this.value = value;
      right = null;
      left = null;
   }
}
```

Traversals

1. Inorder Traversal

1. Traverse the left subtree, i.e., call Inorder(left->subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right->subtree

Uses of Inorder Traversal:

In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Code:

```
void printInorder(Node node)
   {
      if (node == null)
         return;

      /* first recur on left child */
      printInorder(node.left);

      /* then print the data of node */
      System.out.print(node.key + " ");

      /* now recur on right child */
      printInorder(node.right);
   }
```

2. Preorder Traversal

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left->subtree)
3. Traverse the right subtree, i.e., call Preorder(right->subtree)

Uses of Preorder:

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

Code:

```
void printPreorder(Node node)
   {
      if (node == null)
         return;

      /* first print data of node */
      System.out.print(node.key + " ");

      /* then recur on left subtree */
      printPreorder(node.left);

      /* now recur on right subtree */
      printPreorder(node.right);
   }
```

3. Postorder Traversal

1. Traverse the left subtree, i.e., call Postorder(left->subtree)
2. Traverse the right subtree, i.e., call Postorder(right->subtree)
3. Visit the root

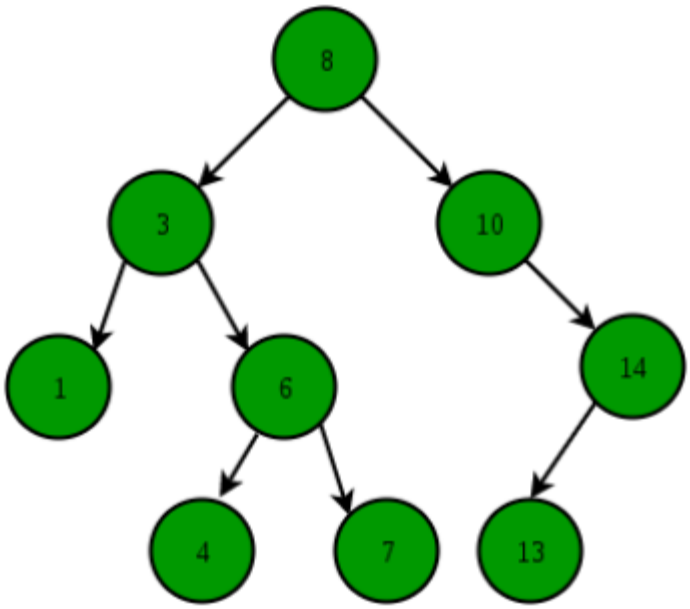
Code:

```
void printPostorder(Node node)
   {
      if (node == null)
         return;

      // first recur on left subtree
      printPostorder(node.left);

      // then recur on right subtree
      printPostorder(node.right);

      // now deal with the node
      System.out.print(node.key + " ");
   }
```

Binary Search Trees

Binary Search Tree is a node-based binary tree data structure which has the following properties:

1. The left subtree of a node contains only nodes with keys lesser than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.



How to search a key in given Binary Tree?

In binary search, we start with 'n' elements in search space and if the mid element is not the element that we are looking for, we reduce the search space to 'n/2' we keep reducing the search space until we either find the record that we are looking for or we get to only one element in search space and be done with this whole reduction.

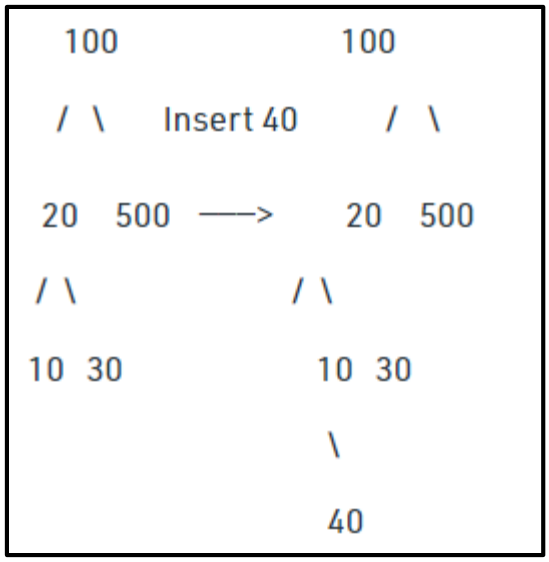
Search operations in binary search trees will be very similar. Let's say we want to search for the number, we start at the root, and then we compare the value to be searched with the value of the root, if it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.

Code:

```
public Node search(Node root, int key)
{
   // Base Cases: root is null or key is present at root
   if (root==null || root.key==key)
      return root;

   // Key is greater than root's key
   if (root.key < key)
     return search(root.right, key);

   // Key is smaller than root's key
   return search(root.left, key);
}
```

Insertion of a key :

A new key is always inserted at the leaf. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

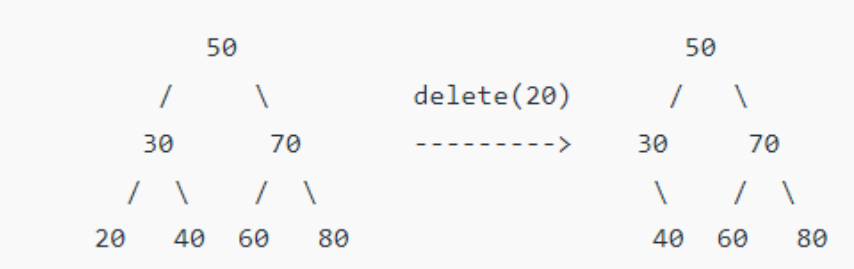


Code:

```
Node insertRec(Node root, int key)
  {
    /* If the tree is empty,
      return a new node */
    if (root == null) {
      root = new Node(key);
      return root;
    }
    /* Otherwise, recur down the tree */
    else if (key < root.key)
      root.left = insertRec(root.left, key);
    else if (key > root.key)
      root.right = insertRec(root.right, key);

    /* return the (unchanged) node pointer */
    return root;
  }
```

The delete operation:

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

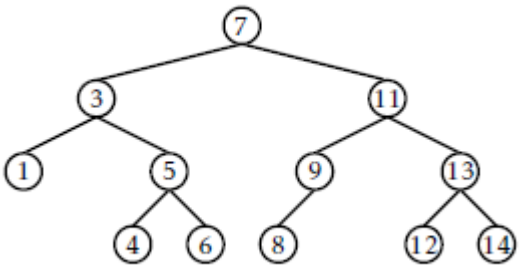
3. Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

Code:

```
Node deleteRec(Node root, int key)
   {
      /* Base Case: If the tree is empty */
      if (root == null)
         return root;
      /* Otherwise, recur down the tree */
      if (key < root.key)
         root.left = deleteRec(root.left, key);
      else if (key > root.key)
         root.right = deleteRec(root.right, key);

      // if key is same as root's  key, then This is the node to be deleted
      else {
         // node with only one child or no child
         if (root.left == null)
            return root.right;
         else if (root.right == null)
            return root.left;

         // node with two children: Get the inorder successor (smallest in the right subtree)
         root.key = minValue(root.right);

         // Delete the inorder successor
         root.right = deleteRec(root.right, root.key);
      }

      return root;
   }

   int minValue(Node root)
   {   int minv = root.key;
      while (root.left != null)
      {
         minv = root.left.key;
         root = root.left;
      }
      return minv;
   }
```

Lab Tasks:

1.



a) Add the values 3.5 and then 4.5 to the binary search tree in Figure.

b) Remove the values 3 and then 5 from the binary search tree in Figure.

2. Write a recursive function named checkBST that, given the pointer to the root of a binary tree, will return true if the tree is a BST, and false if it is not. Implement the function in the main class.

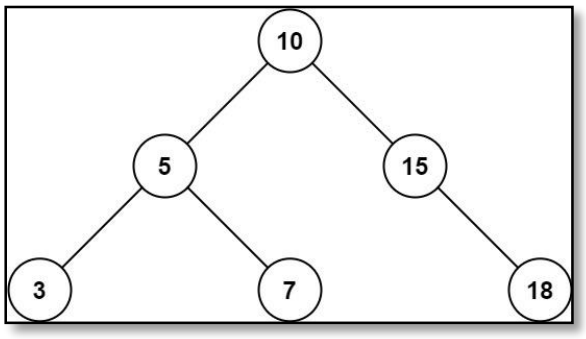
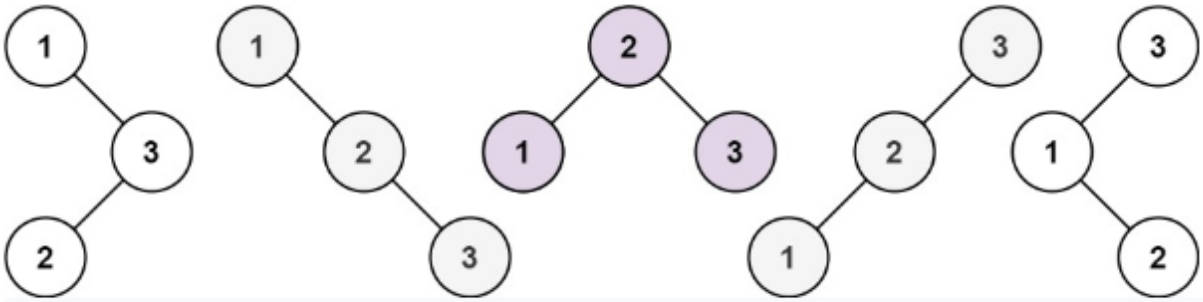


Figure 2

3. Assume that a given binary tree stores integer values in its nodes. Write a recursive function that traverses a binary tree, and prints the value of every node who's parent has a value that is a multiple of five. Function should be implemented in the main program. Use the figure 2.

4. Write a recursive function named smallcount that, given the pointer to the root of a BST and a key K, returns the number of nodes having key values less than or equal to K. Function should be implemented in the main program. Use the figure 2.

5. Given an integer n, return the number of unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n.




```
Input: n = 3
Output: 5
```