Binary Search Trees (BST)

Objective

The objective of this lab is to understand implementation of Binary Search Tree.

Task

1. Tree can be traversed in the following possible ways:
   In-order (LNR), Pre-order (NLR), Post-order (LRN).
   Implements recursive methods for all three traversals in linked list based implementation.

2. Deleting a node in BST using linked list implementation.

Procedure

- Deleting a node is the most common operation required for binary search trees. You start by finding the node you want to delete, using the same approach as you did in find() and insert() method. once node is found then there are three possible cases in binary search tree to consider:

  1. **The node to be deleted is a leaf (has no children):**
     To delete a leaf node, you simply change the appropriate child field in the node's parent to point to null, instead of the node.

  2. **The node to be deleted has one child:**
     The node has only two connections: to its parent and to it's only a child. You want to cut the node out of this sequence by connecting its parent directly to its child.

  3. **The node to be deleted has two children:**
     To delete a node with two children, replace the node with its in-order **successor** and delete the successor node as discuss above in case 1 or case 2.

Finding the Successor:

First, the program goes to the original node's right child, which must have a value larger than the node. Then it goes to this right child's left child (if it has one), and to this left child's left child, and so on, following down the path of left children. The last left child in this path is the successor of the original node. If the right child of the original node has no left children, this right child is itself the successor.

```
Delete(Object element){
        node[] ref=find(data) //find node to delete that return node t and its parent p references
        If(node t has no child) { // call deleteNochild(t,p) }
        If(node t has one child) { // call deleteOnechild(t,p) }
        else{ //(node t has two children)
            Find node minNode with minimum value on t's right subtree
           Replace t data with this minNode value
          If(minNode is noChiled node) // remove minNode
               // call deleteNochild(t,p);
        else
              // call deleteOnechild(t,p)
        }
}
```

```
class node<T>{
  T data;
  node<T> left;
  node<T> right;

  node(T d){
  data=d;
  }
}
public class BSTlinklist<T extends Comparable<T>> {
    node<T> root;

public void insert(T d){   .....    }

public void TraverseLNR(node n){ ....}

public node[] find(T key){ … }

protected void delNoChild(node parent,node temp){ .... }

protected void delOneChild(node parent,node temp){ ... }

public void delete(T key){
    node<T> temp=root;
    node<T> parent=root;
    node[] ref=find(key);
    parent=ref[0];
    temp=ref[1];

if(temp!=null&&key.compareTo(temp.data)==0){
            if(temp.left==null && temp.right==null){            // no child case
                delNoChild(parent,temp);
             }                                                  // one child case
            elseif((temp.left!=null &&temp.right==null)||(temp.left==null && temp.right!=null)){
                  delOneChild(parent,temp);
               }
            else{                                               // two child case

             }
  }
else{       System.out.println("key not found");
 }

}
}
```