Question 1) Given a string, write a routine that converts the string to a long, without using the built in functions that would do this. Describe what (if any) limitations the code has.

```
/*
         * stringToLong function takes in a ASCII string which represents a number
         * and converts it into a Long.
         *
         * Known Limitations:
         * This Function requires the string to be clean, i.e without any spaces,
         * commas or any other special characters. The function will print an error  message
         * and return 0 if the string is not clean.
         *
         * Also, this function cannot handle strings longer than range of Long
         *
         */
        private static long stringToLong(String s)
        {
                // Check for null string
                if(s == null || s.length() == 0)
                        return 0;

                boolean isNegative = false;
                int i = 0;

                // Convert String to Character array
                char[] charStr = s.toCharArray();

                // Check if number in string is negative
                if(charStr[i] == '-')
                {
                        isNegative = true;
                        i++;
                }

                long value = 0;

                // Iterate through all characters and multiply by ten during each iteration
                // and add Integer value of character .
                while(i < charStr.length)
                {
                        value *= 10;
                        // Check if current character is a Digit, i.e between 0-9
                        if(Character.isDigit(charStr[i]))
                        {
                                value += (charStr[i] - '0');
                                i++;
                        }
                        else
                        {
                                System.out.println("Unexpected Non - numeric character found in string : "
+charStr[i]);
                                return 0;
                        }
                }

                // Multiply by -1 if number was negative
                if(isNegative)
                {
                        value *= -1;
                }
```

```
        return value;
    }
```

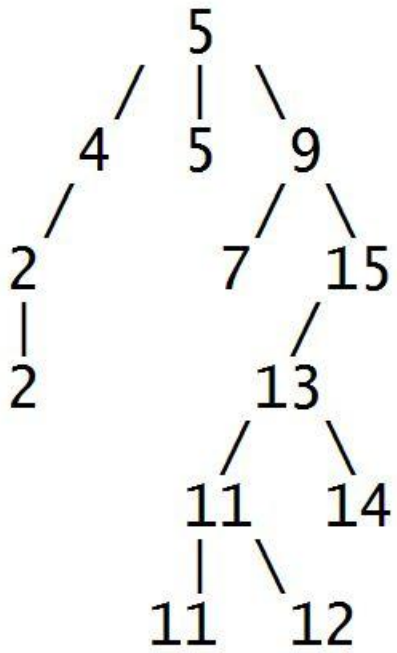
Question 2) Implement insert and delete in a tri-nary tree. A tri-nary tree is much like a binary tree but with three child nodes for each parent instead of two -- with the left node being values less than the parent, the right node values greater than the parent, and the middle nodes values equal to the parent.

Ans:

Please find code below. I have used an iterative approach to perform insert as well as the delete operations. Using iterative approach, I have ensured that the program does not crash due to a stack overflow in the event when there are a large number of nodes.

Test case: Insert order: 5, 4, 9, 5, 7, 2, 2, 15, 13, 14, 11 , 11 ,12

Tree:



The Test program test all cases of deleting a node: 1) Deleting a leaf node 2) Deleting a node with only left child 3) Deleting a node with only right child 4) Deleting a node with both left and right children

Code attached below:

```
/*
 * Tri-nary Tree Insert and Delete Functions
 *
 * Both Insert and Delete Functions have been written
 * using iterative approach. This is an intentional
 * measure taken to ensure that there are no stack overflows
 * which might take place when recursion is used.
 *
 * Using recursion will shorten the length of the code considerably,
 * but I have decided to use the iterative approach to ensure that the
 * code will function correctly without causing a stack overflow.
 *
 */



/*
 * TreeNodeHelper is a Helper class which is used for
 * storing successor and parent of succesor node references
 * for the delete method when the node to be deleted has both left and
 * right children
 */
class TreeNodeHelper
{
	TreeNode successor = null;
	TreeNode parentofSuccessor = null;

}

/*
 * TreeNode class is the actual tri-nary tree class.
 * This also has the insert and delete methods for adding and deleting integer
 * elements fron the tri-nary tree.
 *
 * This also has a helper function which is called from the delete method in
 *  case when the node to be deleted has both left and right subtrees
 */

class TreeNode
{
	int value;
	TreeNode left;
	TreeNode middle;
	TreeNode right;

	TreeNode(int val)
	{
		this.value = val;
		this.left = this.middle = this.right = null;
	}

	/*
	 * Insert method:
	 * Takes in an integer value of the new node to be inserted in the tree
	 */
```

```
public void Insert(int val)
{
    TreeNode newnode = new TreeNode(val);

    TreeNode current = this;
    TreeNode prev = null;

    // Iterative approach to find appropriate place for new node
    while (current != null)
    {
        prev = current;

        if (val > current.value)
         current = current.right;
        else if (val < current.value)
         current = current.left;
        else
         current = current.middle;
    }

    // Attach node to appropriate place in tri-nary tree

    if (val > prev.value)
         prev.right = newnode;
    else if (val < prev.value)
         prev.left = newnode;
    else
         prev.middle = newnode;
}

/*
 * Delete :
 * Delete function takes as its argument, an integer to be deleted from the
 * tri-nary tree.
 * This function, similar to the Insert function uses an iterative approach to
 * move to the node to be deleted
 *
 */

public boolean Delete(int val)
{
    TreeNode root = this;

  if (root == null)
         return false;

    TreeNode current = root;
    TreeNode parent = null;

    // Move to the node to be deleted
    // parent node holds a reference of the node to be deleted
    // which  is used for actually deleting node from tree
    while (current != null)
    {
        if (val > current.value)
```

```
        {
            parent = current;
            current = current.right;
        }
        else if (val < current.value)
        {
            parent = current;
            current = current.left;
        }
        else
        {
         break;
        }
    }

    // If value is not found in tree
    if (current == null)
         return false;

    //Check if the val to be deleted occurs more
    //than once , then delete the bottom most occurrence of it
    if (current.middle != null)
    {
        while (current.middle != null)
        {
            parent = current;
            current = current.middle;
        }
        parent.middle = null;

        return true;
    }

    // Case: Node to be deleted is a leaf node
    if (current.left == null && current.right == null)
    {
        if (current == root)
        {
            root = null;
        }
        else if (parent.left == current)
         parent.left = null;

        else
         parent.right = null;
        return true;
    }

  // Case: Node to be deleted has exactly one child
    if (current.left == null)
    {
        if (current == root)
        {
            root = current.right;
        }
```

```
        else if (parent.left == current)
         parent.left = current.right;

        else
         parent.right = current.right;
    }
    else if (current.right == null)
    {
        if (current == root)
        {
            root = current.left;
        }
        else if (parent.left == current)
         parent.left = current.left;
        else
         parent.right = current.left;
    }
    // Case : Node to be deleted has both left and right children
    else
    {
        // TreeNodeHelper will be used to hold the inorder successor
         // and the parent of the successor of the node to be deleted
         TreeNodeHelper helper = new TreeNodeHelper();

         // Get inorder successor and parent of successor
        getSuccessor(current.right, helper);

        // Re-arrange links to delete node
        if (helper.parentofSuccessor == null)
        {
            current.value = helper.successor.value;
            current.right = helper.successor.right;
        }
        else
        {
            current.value = helper.successor.value;
            if(helper.successor.middle != null)
            {
                helper.parentofSuccessor.left = helper.successor.middle;
                helper.successor.middle.right = helper.successor.right;

            }
            else
                helper.parentofSuccessor.left = helper.successor.right;
        }
    }

    return true;
}

/*
 * getSuccessor method returns the inorder successor of the node to be deleted
 * as well as the parent of the inorder successor.
 */
boolean getSuccessor(TreeNode node, TreeNodeHelper helper)
```

```
    {
       TreeNode parent = null;

       if (node == null)
        {
            parent = null;
            helper.parentofSuccessor = parent;
            helper.successor = node;

            return true;
        }

        while (node.left != null)
        {
            parent = node;
            node = node.left;
        }

        helper.parentofSuccessor = parent;
        helper.successor = node;

        return true;
    }


    void printInorder(TreeNode root)
    {
       if(root !=null)
       {
              printInorder(root.left);
              System.out.println(root.value);
              printInorder(root.middle);
              printInorder(root.right);

       }
    }
}

public class mainClass {

       /**
        * @param args
        */
       public static void main(String[] args) {
              // TODO Auto-generated method stub

              TreeNode root = new TreeNode(5);
              root.Insert(4);
              root.Insert(9);
              root.Insert(5);
              root.Insert(7);
              root.Insert(2);
              root.Insert(2);
              root.Insert(15);
              root.Insert(13);
```

```
            root.Insert(14);
            root.Insert(11);
            root.Insert(11);
            root.Insert(12);



            root.printInorder(root);
            System.out.println("Deleting Node 9 which has left and right children...");
            root.Delete(9);

            root.printInorder(root);

            System.out.println("Deleting middle node 2...");
            root.Delete(2);
            root.printInorder(root);

            System.out.println("Deleting node 4 which has only left child...");
            root.Delete(4);
            root.printInorder(root);

            System.out.println("Deleting node 13 which has only right child...");
            root.Delete(13);
            root.printInorder(root);

        }

}
```