0.0.1 Implementing Stacks and Trees

Since stacks and queues are a prerequistite to our function class, we will design, implement and test them now.

Our stack does not need to become infinitely big, as the size of the stack, as discussed in the last section, is m where m is the number of operands. Therefore we can initialize an array of size m and manipulate the array to make it act like a stack.

Our binary tree will be quite simple, containing only the bare minimum of what our binary tree needs, with an added traversal function which returns the post-order depth-first traversal of the tree. For simplicity we will use a private static helper function, which will take a tree and list as parameter. This list which is passed will be edited and returned in every recursive call. Also we will make our left and right tree attributes public. This is because our root tree will be public, and as the left and right sub-trees are simply different copies of the root tree, it isn't consistent to make the right and left private while keeping the root tree public.

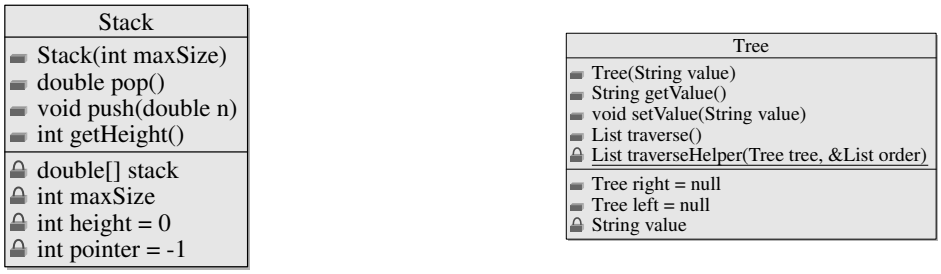

Figure 1: Class Diagrams

Algorithm 1: Post-Order Depth-First Traversal Helper

```
1  function traverseHelper(Tree tree,List &order):
2      if tree == null then
3          return;
4      else
5          traverse(tree.left,order)
6          traverse(root.right,order)
7          list.add(root.data)
8      end
9      return list
10 end
```

Algorithm 2: Post-Order Depth-First Traversal

```
1  function traverse():
2      List order
3      traverseHelper(this,order)
4      return order
5  end
```