Algorithms HW 8 - Trees

1. 12-2 from book

    - Pseudocode-eqsue Scala/Python
    - Never constructs objects, so you'd have to add that if actually implementing it
    - also a little bit of haskell list syntax thrown in because it looks nice

```
class Tree {
    // To use:
    //  1. Insert values into tree
    //  2. Then run unwrapTree
    value:String = nil
    left:Tree = nil
    right:Tree = nil

    insert(str): {
        insertHelper(str, "")
    }

    insertHelper(str, old_str):Tree {
        if str is empty then
            value = old_str
            return this
        else
            if str[0] is 'O' then
                return insertHelper(left, str[1..len(str)], old_str + str[0])
            else
                return insertHelper(right, str[1..len(str)], old_str + str[0])
    }

    unwrapTree():List_of_strings {
        retVal:list_of_strings = [] // empty list
        if left is not nil then
            retVal = retVal ++ unwrapTree(left)
        if value not nil then
            retVal += entry
        if right not nil then
            retVal ++ unwrapTree(right)
        return retVal
    }
}
```

2. From website

Aim: create a function, set-depths-below that sets the depths of each node below the node that is called. I will implement the function such that it sets the depth of the node in the paramater and recursively sets the depth of the nodes further in the tree.

```
class Node {
    depth:Int := nil
    key:Int := nil
    value:Value := nil
    parent:Node := nil
    left:Node := nil
    right:Node := nil

    set-depths-below():Unit {
        // do a pre-order traverse, i.e. hit parents before children

        // work with current node:
        if node.parent is not nil then
            // invariant here: we know node.parent's depth
            node.depth := node.parent.depth + 1
        else // this is the case if we are looking at root node
            node.depth := 0

        // hit the children
        if node.left is not nil then
            set.depths-below(node.left)
        if node.right is not nil then
            set.depths-below(node.right)
    }
}
```

3. From website

   Now let Node also have attributes pred:Node and succ:Node where pred and succ refer to the preceding and succeeding node if the nodes are lined up in order from least to greatest according to their key (a doubly linked list). The goal here is to implement this as we insert nodes where the insertion function uses an iterative method (ie while loop).

   The intuitition: First let's think about managing pred. If we are at a node and move to the left, then the pred isn't going to change. It's going to be something dependent on the nodes above. If we move to the right, the pred is going to change, it's going to become the parent of the new node. Likewise with succ, if we move to the right, then succ doesn't change, but if we move the left, then the temporary succ changes to the parent.

```
T.INSERT(k):
    x := T.root
    y := NIL
    d := 0
    succPtr:Node := nil // new line
    predPtr:Node := nil // new line
    while not (x = NIL) do:
        y := x
        d := d + 1
        if k < x.key then
            // if we go left
            x := x.left
            succPtr := x.parent // new line
        else
            // if we go right
            x := x.right
            predPtr := x.parent // new line
    z := new BSTnode()
    z.key := k
    z.depth := d
    z.parent := y
    z.pred := predPtr // new line
    z.succ:= succPtr // new line
    if y = NIL
        then T.root := z
        else if k < y.key
            then y.left := z
            else y.right := z
```