IMPERATIVE PROGRAMMING 2018

MOISE GABRIEL

QUESTION 5

(a)

```
class CountTree{

    class Tree ( var word : String, var count: int, var left: Tree , var right: Tree)
```

(b)

```
private var root : Tree = null
// DTI: Let B(t)= B(t. left) ++ [(t.word, t.count)] ++ B(t.right) && B(null) = []
// Then, we have B(t) sorted increasingly (lexicographically) and finite after the first element of each
// pair, and all the first elements are distinct and all the second elements are positive integers.
```

(c)

```
def addToTree (word : String, t: Tree) : Tree = {
    var current = t
    if (current == null) current = new Tree (word, 1, null, null)
    else if (current.word > word) current. left = addToTree (word, current.left)
    else if (current. word < word) current. right = addToTree (word, current. right)
    else current. count += 1
    current
}
```

(d)

(i)
```
// DTI: Let L(a,b) = if (a != b) then (a.word, a.count): L (a.right, b) else [(b. word,
// b. count)]. Then, we have L(start, end) is lexicographically-increasingly sorted and finite
// and a. left. right = a.right. left = a for all a except start and end && start. right. left=
// start && end.left.right = end and L(start, end) contains only pairs whose second elements
// are positive integers
```

(ii)
```
def flatten (t: Tree) : (Tree, Tree) = {
    if (t==null) return (null, null)
    else if ((t.left == null) && (t. right == null)) return (t,t)
    else if (t. left == null)
    {   var (a,b) = flatten (t. right)
        a. left = t
        t. right = a
        return (t, b)
```

1.

```
    else if (t.right == null)
    {
      var (a,b) = flatten (t. left)
      b. right = t
      t. left = b
      return (a,t)
    }
    else
    {
      var (x,a)= flatten(t. left)
      var (b,y)= flatten (t. right)
      a. right =t ; b.left=t
      t. left = a ; t. right = b
      return (x,y)
    }
}
```

(iii)

```
def flattenIter (t: Tree): (Tree, Tree) = {
    val stack = new scala. collection. mutable. Stack [Tree]
    var current = t
    var start : Tree = null
    var end: Tree = null
    while ((current != null) || (! stack.isEmpty))
    { if (current != null) {stack.push (current) ; current = current. left}
      else
      {
        current = stack.pop
        if (start == null)
        { start = current
          end = current
        }
        else
        { end. right = current
          current. left = end
          end = current
        }
        current = current. right
      }
    (start, end)
}
```