**Lab 3: Exercise 5**

## Indicate the time complexity of the operations BiIterator::operator++ and BiIterator::operator--, in the best and worst cases. Use Big-Oh notation and motivate clearly your answer.

Tree with *n* nodes. Time complexity of **Operator—**is identical to **Operator++** since the exact same operations are performed except that each parent->right check is parent->left.

**Worst case:**

Tree is generated from a sorted list, i.e. it is unbalanced with only right (or left) children from root. If we then tries to use Operator++ at the last node (or Operator—from first node) the time complexity is O(*n*).

Operations to count:

1. Check if current node exists and if node has a right child. Evaluates to false. 2 steps.
2. Allocate memory for a previous node pointer and assign the current node pointer to it. 2 steps
3. Ascend the tree until a parent node whose left child is the previous node is reached

[Worst case scenario is that all nodes have a single, right child => no parent node whose left child is previous node is ever reached. Thus each step in the ascension iteration is repeated *n* times.]

* 1. For i = 1, 2, 3….(*n*-1), (*n*)
     1. Check if current node is a valid node. 1 step
     2. Check if current node’s left child is equal to the previous node. (false) 1 step
     3. Assign the current node to the previous node pointer. 1 step
     4. Assign the current node’s parent node to the current node pointer. 1 step (for retrieving the parent pointer) + 1 step (for assignment) = 2 steps

T(*n*) = 2 + 2 + *n* \* (1 + 1 + 1 + 1 + 2) = 4 + 6*n* = O(*n*)

**Best case:**

The iterator is currently standing in a leaf node, whose parent has not been visited yet (i.e. the current node is left child to the parent, for operator++). This results in a time complexity of O(1).

Operations to count:

1. Check if current node exists and if node has a right child. Evaluates to false. 2 steps.
2. Allocate memory for a previous node pointer and assign the current node pointer to it. 2 steps
3. Ascend the tree until a parent node whose left child is the previous node is reached

[Best case scenario is that the current node is left child to its parent => the ascension iteration is executed just once.]

* 1. Check if current node is a valid node. 1 step
  2. Check if current node’s left child is equal to the previous node. (true) 1 step

T(*n*) = 2 + 2 + 1 \* (1 + 1) = 6 = O(1)