Assignment 3 Problem 2

In this question, we want to add support for an operation `ithSuccessor` on AVL trees (in addition to the standard operations `insert, delete, find`). The operation `ithSuccessor` has two parameters, x and $i \geq 0$, and returns the ith inorder successor of the node x. If $i = 0$, then the node x itself is returned. You may assume that all input is valid; i.e. the successor exists (but may not be in the subtree rooted at x).

We assume that the nodes have the following fields:

- `key` – the key of the node;
- `left` – pointer to the left child;
- `right` – pointer to the right child;
- `balance` – balance factor of the node;
- `parent` – pointer to the parent of the node;
- `isLeft` – is true if the node is a left child of its parent;
- `isRight` – is true if the node is a right child of its parent;
- `numLeft` – holds the number of nodes in the left subtree of the node;
- `numRight` – holds the number of nodes in the right subtree of the node.

a) Give an algorithm `ithNode(x, i)` which returns the ith inorder node in the subtree rooted at x. For example, suppose the subtree contains m nodes, when $i = 1$, the minimum element in the subtree is returned and when $i = m$ the maximum element in the subtree is returned. You may assume that the subtree has at least i elements. Your algorithm should take worst-case $O(\log(m))$ time. Briefly justify that your algorithm achieves this runtime.

The algorithmn of ithNode(x, i) should do the following:
if i = the number of nodes in the left subtree + 1, then return x
else if the number of nodes in the left subtree of x is less than i, then recursively call ithNode() but with x's right child and i-x.numLeft-1 else if the number of nodes in the left subtree of x is greater than or equal to i, then recursively call ithNode() but with x's left child and same i.

The pseudo-code should be the following:
ithNode(x, i)
    if(i = x.numLeft + 1) then return x
    else if(i > x.numLeft) then

```
        ithNode(x.right, i-x.numLeft-1)
    else if(i ≤ x.numLeft) then
        ithNode(x.left, i)
```

The recursion will terminate on the leaves, although sometimes it will terminate earlier, we consider the worst-case which means that ithNode() terminate on a leaf. Since the function runs on a AVL-tree, and from lecture, we know that the height h of a m-node AVL-tree is $\Theta(\log m)$, for each recursion, what we do is judge the ith inorder node is in the left or right subtree and then we move to the corresponding subtree which has a height 1 less. And the other operations are in the constant time, therefore, the algorithm has runtime in the worst-case $\Theta(height) = \Theta(\log m)$

b) Give the algorithm for `ithSuccessor(x, i)` for an AVL tree with n nodes. Your algorithm should take worst-case $O(\log(n))$ time and must use `ithNode(x, i)` from above. Briefly justify that your algorithm achieves this runtime.

The ithSuccessor(x,i) should do the following:
We firstly judge whether i is less or equal to the number of nodes in the right subtree of x, if so, we directly call ithNode(x.right, i), if not, we move upwards(towards the root) to the ancestors of x, until we find a node (say y) that x is in the left subtree of y, then we do the exactly same thing as we did for x, but this time we consider if i = x.right + 1, then return y, else if i ≤ x.right + 1 + y.right, and call ithNode(y.right, i-1-x.right).

The pseudo-code should be the following:

```
ithSuccessor(x, i)
    if(i ¡= x.numRight) then
        return ithNode(x.right, i)
    temp ← x.numRight
    y ← x
    while(true)
        if(y.isLeft)
            y = y.parent
            if(i ≤ temp + 1 + y.numRight) break
            temp = temp + 1 + y.numRight
        else
            y = y.parent
    if(i = temp + 1) return y
    else return ithNode(y.right, i - 1 - temp)
```

This algorithm is in $O(\log n)$, because what we do is find the ancestors of a node and calling ithNode() only once. And other statements in the algorithm are in constant time, therefore, in the worst-case, if we starting from a deepest leaf, and find the ancestors to the root, then call ithNode() at root, we have the running time $\Theta(height)+\Theta(\log m)$ where m is the number of node in the right subtree of root. We know that height h is $\Theta(\log n)$ and m is less than n which implies $m \in O(n)$. Therefore, the running time for the algorithm should be $\Theta(\log n) + \Theta(\log O(n)) = O(\log n)$