CSC263H Tutorial 4

Problem Set

Winter 2024

Work on these exercises *before* the tutorial. You don't have to come up with a complete solution, but you should be prepared to discuss them with your TA. We encourage you to work on these problems in groups of 2-3.

Recall the Ordered Set ADT from lecture:
Sets with $Insert$, $Delete$, $Search$ and:

- *Rank*(k): return the *rank* of key k, i.e., the index of k in the sorted order of all set elements (where indexing starts at 1).
  Note: This operation does NOT necessarily involve sorting the keys; the "'*sorted order*" is used only to define this precisely.
- *Select*(r): return the key with rank r.

Example: If the set contains $\{27, 56, 30, 3, 15\}$, then $Rank(15) = 2$ and $Select(4) = 30$ (because the sorted order is $[3, 15, 27, 30, 56]$).

1. Explain, briefly and at a high level, how to use AVL trees (with no additional information) to implement an Ordered Set.
   Describe the implementation of each new operation (*Rank* and *Select*).
   Determine the worst-case complexity of each new operation.

   Solutions: For both *Rank* and *Select*, we start by traversing the given AVL tree in Inorder. For $Rank(i)$, we return the number of nodes visited before finding the target node i. For $Select(j)$, we return the jth node visited node. The worst-case complexity for both operations are $O(n)$ where n is the size of the AVL tree since we might have to visit every node in the tree.

2. Explain, briefly and at a high level, how to use AVL trees augmented with *node.rank* (the rank of the node) to implement an Ordered Set. Describe the implementation of each new operation and necessary modifications to each old operation. Determine the worst-case complexity of each operation (new and old).

   Solutions: For $Rank(i)$ we first find the target node i via binary search on the keys and return the rank attribute of the node with key i.
   For $Select(j)$ we just need to do a binary search on the rank attribute of nodes.
   The worst-case running time of both operations is $O(log(n))$ because it just involves a binary search.

   When we insert a new node like x into the AVL, we need to update the rank attribute of all ancestors of x, plus the rank of all nodes in the right subtrees of the the rank. In the worst-case the new inserted node is the with minimum key and therefore all the nodes must be visited. Thus, the worst-case running time of $AVLInsert$ with this augmentation is $O(n)$. Similarly, whenever we delete a node like x, we need to update the rank attribute of all ancestors of x, plus the rank of all nodes in the right subtrees of the the rank. In the worst-case, we delete the node with the minimum key value and therefore must update rank attribute of $n-1$ nodes, hence the worst-case running time of $AVLDelete$ with this augmentation becomes $O(n)$.

3. Explain, in more detail, how to use AVL trees augmented with *node.size* (the size of the sub-tree rooted at node) to implement an Ordered Set. Describe the implementation of each new operation and necessary modifications to each old operation. Determine the worst-case complexity of each operation (new and old).

Solutions: For $Rank(i)$ we first find the target node x with the key value i via binary search. $Rank(i)$ is equal to 1 plus (1) the number of node in the left subtree of x plus (2) any ancestors of x that x is in their right subtree plus (3) the number of nodes in the left subtree of any ancestors of x that x is in their right subtree.
Note that we can count the size of the left child of x, as well as the size of left subtrees of all ancestors of x directly since *node.size* is stored at all nodes.
The complexity of *Rank* is $O(log(n))$ since the the time complexity of binary search is $O(log(n))$ and there are $log(n)$ ancestors of x.

For $Select(j)$, we try to find the jth element in the AVL tree. Since *node.size* are given, we only need to explore a child of *node* if j is in the sub-tree of the child node. Here's the complete pseudo-code:

```
SELECT(root, r)
    x = root
    p = x.left.size + 1 # rank of root in the current subtree
    if p == r: # found it
        return x
    elif r < p: # the target is in left subtree
        return SELECT(x.left, r)
    else: # the target is in right subtree
        return SELECT(x.right, r - p)
```

The time complexity is $O(log(n))$ since we only need to explore at most $log(n)$ child nodes before finding the jth node in the AVL tree.

Since for all nodes, *node.size* can be calculated by using the some of the size attribute of the left and right subtrees of the node, this augmentation does not change the runtime complexity of $AVLInsert$ and $AVLDelete$.