Ryan Tick

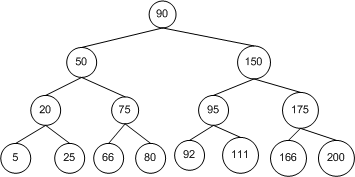
CSE 40113

Assignment 4

Due: 10/26/2015

For each solution of a data structure problem, you must present: (1) the design of your data structure, and (2) the algorithm and time analysis for each operation.

1. For this problem, I would use a binary search tree. At each node, my data structure would store the value of the current node as well as a token storing the number of nodes that are contained within the right branch of the current node, plus itself. With this in mind, we can look at the operations that our data structure should support:  
   1. **Insert**(x): Binary Search Tree already has a standard insert function, so we would use that. This would also allow us to check and see if x is already in the binary tree; if we find the value when we traverse to insert the new value, then we know not to insert the new node as it is already present. You would have to slightly alter this insert function, however, to match our data structure. When we want to insert a new node, x, any time we go down a right branch, we would have to increase the counter (our token) by one. This has a runtime complexity of O (log n).
   2. **Delete**(x): Again, just like we did for insert, we would use the standard delete function of the Binary Search tree. As we go down the tree to check for the existence of x, any time we take a right branch, we would decrease our counter (our token value) by one. If we find the value, then we would delete it. However, if we don’t find the value then we would not do anything. This has a runtime complexity of O (log n).
   3. **Search**(x): This would be really easy to do using the standard Binary Search Tree function search. You would start at the root of the tree. At each node, if the value you are looking for is smaller, than you would move to the left. If the value is bigger, you would move to the right. If you move from root to leaf without finding the value, then we know that the value doesn’t exist within the tree. This has a runtime complexity of O (log n).
   4. **Rank-among-Larger-Keys**(x,k): For this algorithm, we are supplying an x (which is the floor value) and we are given a k (which describes the k-smallest value we are looking for above our floor value, x). In other words, if our x is 10 and our k is 2 then we would be looking for the second smallest value bigger than 10. With this in mind, we can think about how we would implement our algorithm. First, we would start at the top of the tree. If would traverse down the tree until we found a node value closest to x. Once we did this, we could look at the value of our tokens. If we are at a node that is the closest value to x, then we are looking for k nodes bigger than it. Consider the following tree.



<https://i-msdn.sec.s-msft.com/dynimg/IC520.gif>

Let’s say we call the algorithm by Rank-among-Larger-Keys(76,3). If we do this, then our algorithm would search and find 75, since 75 is the closest value less than 76. So our x would be the node with value 75. We are looking for the third largest value after this. If we look at the token of our x node, we see that there is only one value in the right sub-tree. So our token would be 1. This is no good, since we are looking for the third largest value. We would then go to the root, since this would be the second largest value. Next, we would traverse all the way down the left most branch of the right sub-tree, and this would be our answer. In other words, our algorithm would do the following:

* Check the token value of the current node.
  + If the value is greater than or equal to k, than traverse down k nodes and return.
  + Else, check the value of the parent node.
    - If the parent node is larger (meaning you were looking at a left child originally), then check the token value of the parent.
      * If the token value is greater than or equal to k, than traverse down k nodes and return.
      * Else, recursively call this step.
    - Else, return to the root. This will be the next largest value. If k equals 1, then you can return the root of the tree.
  + However, if k is larger than 1, you would traverse all the way down the left most branch of the right side of your original tree. Once you are at a leaf of the tree, this will be the next smallest value after the root. From this point, if k is equal to 2 you would return this leaf value. Otherwise, you would traverse up the tree following the nodes in order of value and return the eventual node.
  + If you get to the very top of the tree again, that means that k was too large of a value. In this example, if our x node was still 75 and k was 100, then we can clearly see that we would run into a problem since there isn’t 100 values bigger than 75 on our tree.
  + This has a runtime complexity of O (log n) since it is basically an augmented search.

1. Maybe use a fenwick tree?

<http://stackoverflow.com/questions/5712092/data-structure-supporting-add-and-partial-sum>

1. **Problem 9.3-5, page 223**

For this problem we can modify example 9.3 on page 220 slightly. Our algorithm would do the following:

* 1. We would first check the size of our array. If the size of our array is 1, then we would return the single element of the array and this would be our median.
  2. Otherwise we would find the median of the given array. We could do this like the book does by dividing the array into five different groups. We would find the midpoint of each of these groups by insertion sorting them, and then we would pick the midpoint since it would be simple at this point.
     1. If there is an even number of elements, the midpoint is the smaller one.
  3. Next we would partition on that midpoint. Then we would let k = 1 + (number of elements on the low side of the partition).
  4. If (i == k) then we would return the value of our array at q (return A[q]).
  5. Else if (i > k) then we would call SELECT again on the high side of our partition looking for i where i = i – k.
  6. Else if (i < k) then we would call SELECT again on the low side of our partition looking for i.

T(n) = 2T(n/2) + c(n) which would be a linear big-O runtime complexity.