Brigham Michaelis
October 21 2017
A00364835

Homework 4

The solution to this problem was discovered while working with Josh Dawson, one of the students in the class.

Problem 1.

(20 points) Suppose we have a min-heap with n distinct keys that are stored in an array A[1 . . . n] (a min-heap is one that stores the smallest key at its root). Given a value x and an integer k with 1 ≤ k ≤ n, design an algorithm to determine whether the k-th smallest key in the heap is smaller than x. The running time of your algorithm should be O(k), independent of the size of the heap.

Remark. If we were to find the k-th smallest key of the heap, denoted by y, then the best way would be to perform k times deleteMin operations, which would take O(k log n) time. Our above problem, however, is actually a decision problem. Namely, you only need to decide whether y is smaller than x, and you do not have to know what the exact value of y is. Hence, the problem is easier and we are able to solve it in a faster way, i.e., O(k) time.

Description

The main question is "_is the kth_value_smaller_than_x?" return true or false.

For this algorithm we are basically doing a modified version of a tree traversal. First note that a heap is an array. So we would have to access the children of a node by:
left(i): 2*i;
right(i): 2i+1;
parent(i):(i/2);
Now to perform the algorithm we are simply going to do a recursive tree traversal. And while we are traversing the tree we are going to count how many values(nodes) are smaller than x(which is given). We will call this variable "count"

```
1. Start at the root; If the root is smaller than x count++;
        else return;
2. Inspect the left child;
3. Inspect the right child;
```

There are some rules to follow however,

1. **If we inspect a node that is larger than x then we return and we do not traverse anything in that node's subtree.**
2. **We do not follow children that would overreach the bounds of the array.**
3. **If the count is ever greater than or equal to k(which is given). Then we immediately stop and return true;**

In this fashion we traverse only the nodes of the tree that are less than x and as we keep track of the values that are smaller than x (this is our count variable) we are guaranteed to have a worst case runtime of less than 2k, which is still O(k)

Psuedo Code
Not needed
Correctness
This algorithm works because we do a check on each node before looking at it’s children. If the node is smaller than x the we increment the count and also check the children. If the node is larger than x we return and if the count is ever greater than or equal to k we return true. **We return false only when we are done traversing the tree(following the rules above) and our count is less than k.**

Runtime
The runtime for this algorithm is O(k) best time and O(2k) worst time.

Problem 2.

(20 points) Suppose you are given a binary search tree T of n nodes (as discussed in class, each node v has $v.left$, $v.right$, and $v.key$). We assume that no two nodes of T have the same key. Given a value x, the *successor* of x in T is defined as follows: (1) If x is larger than every key in T, then x does not have a successor; (2) if x is equal to a key in T, then the successor of x is x itself; otherwise, the successor of x is the smallest key of T that is larger than x.

For example, in Figure 1, the successor of 19 is 20, the successor of 48 is 48, and 70 does not have a successor.

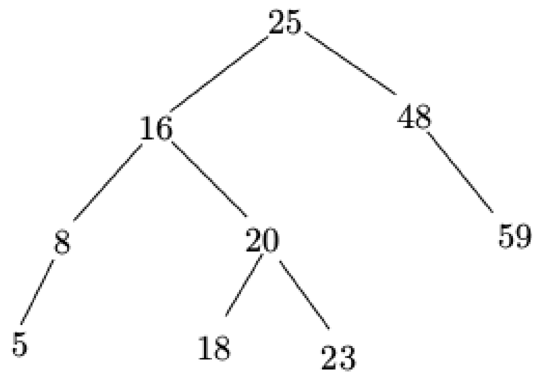

Figure 1: A binary search tree.

Let h be the height of the tree T. Design an $O(h)$ time algorithm to perform the **successor** operations: Given any value x, your algorithm should return the successor of x in T, and simply return "NULL" if x does not have a successor in T.

Note: You are encouraged to give pseudocode for this problem.

Description

For this algorithm we are going to first check the largest value in the tree. (Start at root and go to the farthest right child). This will take O(h) time. If x is larger than the largest value then there is no successor and we return null.

Now here, because the first check did not return, we know that there exists a value that is larger than or equal to x in the tree.
We are going to perform a search(we discussed this algorithm in class it takes O(h) time). If x is in the tree then we will simply find that value and return it.
The main part of this problem comes about if x is not in the tree. To return the smallest value in the tree that is larger than x we perform the following algorithm.

Note that at this point in the successor algorithm we know two facts.
1. x is not larger than the largest value in the tree
2. The value of x does not exist in the tree.

With these two facts we can now present our algorithm that will return the smallest value in the tree T that is larger than x.

First we have a variable called “closest-successor” we initialize this variable with the value stored at the root of the tree.

Now we are just going to follow the same steps used in the search algorithm for a binary tree. The only change is that whenever we move to new node in the tree the “closest-successor” is assigned the value of that node if and only if that node’s value is greater than x otherwise it remains unchanged. Once this modified version of a search is completed(it should return false) instead of returning false we are simply going to return the value stored in “closest-successor”.

Psuedo Code
Not required
Correctness
These algorithms are basically just performing a search which we already know runs in O(h). We are also checking for the three conditions outlined in the problem description.

Runtime
The runtime is O(h), because we are just basically doing 1-3 searches each of which runs in O(h) runtime.

Problem 3.
(20 points) This problem is concerned with range queries (we have discussed a similar problem in class) on a binary search tree T whose keys are real numbers (no two keys in T are equal). Let h denote the height of T . The range query is a generalization of the ordinary search operation. The range of a range query on T is defined by a pair [x l , x r], where x l and x r are real numbers and x l ≤ x r . Note that x l and x r may not be the keys stored in T .
You already know that the binary search tree T can support the ordinary search, insert, and delete operations, each in O(h) time. You are asked to design an algorithm to efficiently perform the range queries. That is, in each range query, you are given a range [x l , x r], and your algorithm should report all keys x stored in T such that x l ≤ x ≤ x r . Your algorithm should run in O(h + k) time, where k is the number of keys of T in the range [x l , x r]. In addition, it is required that all keys in [x l , x r] be reported in a sorted order.

Remark. Such an algorithm of O(h + k) time is normally called an output-sensitive algorithm because the running time (i.e., O(h + k)) is also a function of the output size k.

Description
This is really easy we are just going to do an in-order tree traversal. The only thing we are changing is in the "in-order" algorithm we are just going to add two lines.
The basic steps to an in-order tree traversal are the following:

Step 1. Recursively traverse left subtree
Step 2. Visit root node
Step 3. Recursively traverse right subtree

We will modify this as seen below:

```
if (node !<= xl)
{
   Step 1. Recursively traverse left subtree;
}
// otherwise proceed to step 2.
Step 2. Print root node;
if(node !>= xr)
{
   Step 3. Recursively traverse right subtree;
}
```

And with that we are done!
Psuedo Code
Not required
Correctness
This algorithm will work in O(h+k) time; h is how far down the tree we have to go before we hit xl, then we just do our in-order traversal from xl to xr. And we traverse nothing else.
Runtime
The runtime is O(h + k), h-time to get to xl and k-time to get to xr from xl in an in-order traversal. Only touching the nodes that will be printed.

Problem 4.
(20 points) Consider one more operation on the above binary search tree T in Question 3:
range-sum(x l , x r). Given any range [x l , x r] with x l ≤ x r , the operation range-sum(x l , x r)
reports the sum of the keys in T that are in the range [x l , x r].
You are asked to augment the binary search tree T , such that the range-sum(x l , x r) operations,
as well as the normal search, insert, and delete operations, all take O(h) time each, where h
is the height of T .
You must present: (1) the design of your data structure (i.e., how you augment T); (2) the
algorithm for implementing the range-sum(x l , x r) operation.

Description
1. The design of our data structure is only modified in that each node is going to keep track of the total sum of it's subtree. We discussed this idea in class, suffice it to say, that each node knows, in addition to it's own value the sum of itself plus its left and right children.
2. The algorithm for this is now very easy. Starting at the "lowest common ancestor" we discussed this algorithm in class. We start by storing the total sum of the "lowest common ancestor" in a variable called "range-sum". We then traverse down the tree to xl until we hit its successor(see problem 2 above) from here we will subtract the total sum of the successor's left child from "range-sum". Finally we traverse to the successor of xr(again see problem 2 above), and subtract the total sum of its right child from "range-sum". And we are done. Now we just return the variable "range-sum".
Psuedo Code
Not needed
Correctness
This algorithm is correct because in our worst case we are traversing the height of the tree just 3 times. Once to find the "lowest common ancestor", once get to the successor of xl from the "lowest common ancestor", and one final time to get to the successor of xr, again from the "lowest common ancestor".
Runtime
The runtime(worst-case) is O(3h) which is still O(h).

On a side note the insert and delete functions would have to be slightly modified to update the nodes and their respective parents. But this can be done with destroying their O(h) runtime. It just isn't required.