1DV516 assignment 1

Björne Söderberg Laanemäe - bl222ey

6 oktober 2018

1 Exercise 1

1.1 Most similar value

This one is very very similar to insert, or a supposed «find». We traverse the tree like if we were looking for the specific number, all the while saving the best match yet. In this way, at every node we can safely discard the left or right subtree, because one of them will always be further from our goal than we already are.

The complexity of this depends on how the tree looks. A worst case tree has a depth of N, where every level has only 1 element. That is, a tree where every node has only one child - a straight line. Furthermore, the element we seek would be in the bottom. This gives us a complexity of $O(N)$.

My innermost loop is a `while(true)`, which looks daunting. It is exited with `break` in three cases - if only a left tree is interesting but there is none, if only the right tree is interesting but there is none, or if we found the exact value. This gives us (per iteration) a $O(3)$ for having to update the most similar value and $O(2)$ for traversing (which is the most costly path). In the worst case tree from earlier, the loop is run N times. Before this we have a static $O(3)$. All of this gives us $O(3 + 5N)$, which can be simplified to $O(N)$.

1.2 Print by levels

I used the approach that was brought up in the lectures, adding the children of the root to a queue and then continuing like this for all nodes in the queue. In this fashion we walk through the tree per level, left to right. I had some problems in knowing when a certain level was completed so i added some counters. I also had to implement a queue.

In our worst-case tree, the depth is N and the ”width” of every level is 1. This means that our innermost loop `while (i<steps)` will run only 1 time. Inside it, `n.dequeue()` is $O(1)$, and then we have some other statements totalling the inner loop up to $O(9) = O(1)$.

The outer loop has a complexity of 6 plus the inner loops $O(1)$, but will repeat N times, giving us $7N = O(N)$. Theres a ”setup cost” as well with with a static complexity of 4. Thus the total comes down to $O(N)$.

1.3 Conclusion

This cost of $O(N)$ is quite high, and could probably be improved, by for example changing this whole thing into an AVL tree so that it is constantly balanced. I plan to do this if i got the time after completing exercise 2.

2 Exercise 2

2.1 Proposal, discussion

The requirements of $O(1)$ worst-case are quite harsh and does not leave much room. The insert/remove operations can be done with a doubly linked list, where i keep track of both ends. However, the «findMin» is puzzling. My first thought was a heap but that would worsen the complexity of the remove operation, or i'd have to build the heap each time we call findMin, giving a complexity of $O(\log N)$, which is out of bounds. I also looked at implementing a fibbonacci heaps since they have amortized $O(1)$ for a lot of operations... but not «deleteMin». Unfortunate, since i need that to be constant.

I also thought of keeping a separate simply linked list, operating like a stack where every time i insert a new node, i would compare it with the top. If the new node is smaller, it's pushed to the stack. Then, the minimum element is guaranteed to be on top, and furthermore we could end up with some of the next smallest values. However this all falls apart quickly once the list becomes empty, since recreating it from the nodes already in my main list would be costly. Also, it doesn't really work in the first place.

2.2 Analysis

I'm going for the doubly linked list described above.

2.2.1 Insert

Since the list is doubly linked, insertion should be the same in both ends, except for variables used. I'm using the left end here as an example.

1. Create a new node with the new value. $O(1)$
2. Retrieve the leftmost node. $O(1)$
3. Change the left-pointer of the old node to point to the new one. $O(1)$
4. Change the right-pointer of the new node to point to the old one. $O(1)$
5. Change the pointer for leftmost node to the new node.

And that's basically it. In case of there being no previous node (ie. the new one is the first in the list), we need to set one more pointer. In any case, this comes down to $5 * O(1) = O(1)$.

2.2.2 Remove

This one is symmetric as well. Removing in both ends should look the same (but mirrored). Again, i'm using the left end in my example.

1. Retrieve the value from the leftmost node. $O(1)$
2. Change the pointer for leftmost node to the second leftmost node. $O(1)$
3. Change the left-pointer of the second leftmost node to null. $O(1)$
4. Return the value retrieved in step one. $O(1)$

Java is garbage collected so i do not have to care about what happens to the leftmost node. All in all, this is $4 * O(1) = O(1)$. As with insert, in the edge case of removing the last node, i would need to change an additional pointer, still leaving us with $O(1)$.

2.2.3 findMin

Without any extra internal data structures my list will not find the minimum better than in linear time - I will do a good old linear search (and i'm not proud, but i know that it works).

1. Say that the leftmost node is the current node. $O(1)$
2. Say that the leftmost node value is the smallest value. $O(1)$
3. If there is a node to the right of the current node, say that it is the current one. If not, goto step 6. $O(1)$
4. If the current node value is lower than the saved smallest value, say that the current node value is the smallest. $2 * O(1)$
5. Goto step 3. $O(N)$
6. Return the lowest value. $O(1)$

This is a loop that iterates through all the items exactly once. Inside loop are a bunch of statements, supposedly 3 (one comparison and two assignments), which is $3N$. We have some housekeeping statements outside, say three. In the end, this is $3N + 3 = O(N)$.