Exercises 4: Heaps (and Priority Queues)

Questions

1. If you are rusty on heaps, see the exercises from my COMP 250 web page.

2. Suppose you were to use a balanced binary search tree (BST) to represent a priority queue. Both a balanced BST and a heap require O(log n) to remove the minimum element. However, a balanced BST also requires O(log n) to find (but not remove) the minimum element, whereas a heap requires only O(1) to find the minimum element. How could you augment a balanced BST so that finding the minimum element is O(1)?

3. We have seen how to use an indexed heap implementation of a priority queue that allows us to change the key (priority). Suppose we used a binary search tree to represent the priorities, instead of a heap, and suppose we wanted to be able to change the priorities (as in Assignment 1) and finally suppose we want to do this in time O(log n). Would we need to augment the BST to be able to do so? Why or why not?

4. Suppose you have a heap which supports the usual add() and removeMin() operations, but also supports a changePriority(name, newPriority) operation. How could you combine these operations to define a remove(name) operation?

5. Suppose you have a heap which supports the usual add() and removeMin() operations, but also supports a removeName(name) operation. How could you combine these operations to define a changePriority(name, key) operation ?

6. Suppose you used an unordered list (either linked or array) to implement a priority queue. How long would the operations removeMin(), add(element, key), findMin() take?

7. Suppose you used an ordered array to implement a priority queue. Give the O( ) time for the operations removeMin(), add(element, key), findMin() take?

8. [reworded Jan. 29] An indexed priority queue is a priority queue which allows you to access any object by its name, using a map (such as a hashmap as in Assignment 1). Is it really necessary to have a separate map though? The object might be located anywhere in the queue but you could always find an object by traversing the priority queue. What then is the benefit of having a separate map ?

9. Suppose you used an ordered array to implement an indexed priority queue. That is, suppose the priorities were ordered rather than being stored in a heap. Would there be any benefit in doing this? How long would it take in the worst case to execute changePriority() ?

10. In the lecture I gave an algorithm, based on downHeap, for building a heap in O(n) time. Why is the algorithm correct i.e. how can we be sure that it indeed builds a heap ?

11. Do priority queues need to have distinct priorities? Is there a potential danger if they don't?

12. Will you have to know for the exam that amazing Calculus trick where you take the derivative of x^l and substitute x=2, in order to prove you can build a heap in O(n) ?

13. [added Jan 17] To change a priority in a heap, I argued that we need to augment the heap with a nameToIndex map which allows us to know where each object (name) is. Is this really necessary though? We can access any object in the heap just be doing a linear search through the heap array. This will take time O(n) which is obviously slow. But we need to spend at least O(n) anyhow to build the array. So if we're already at O(n), then why does it matter if we add another O(n) operation ?

Answers

1. See my COMP 250 web page for solutions.

2. Easy. Just add an extra variable to reference the minimum element. Note that whenever you add or delete a new element, you also need to examine whether the minimum element in the balanced BST has changed and if so then update this extra variable. This check can be done at the end of the add/remove routine. This would require O(log n) to check the minimum element, but add/remove takes O(log n) anyhow, so there's no extra cost here in the O( ) sense.

3. Yes you would. The BST is ordered by the priority, then so you won't be able to index the object whose priority you want to change. You would also need a map from the names of objects to nodes in the BST, and you would need to update this map if you change the BST. You can do all this in O(log n). e.g. Suppose you used a hash table for the map from names to BST nodes; then you can access this map in O(1). Once you change the priority of a node, you need to update the BST so it is well ordered by priority. For that, you can remove the priority, THEN change it, and then add it back in (in its correct place). For a balanced binary tree, the remove and add can each be done in O(log n).

   Note that if you didn't use the extra map and you just traversed the BST to find the name you're looking for, this would take time O(n) – too slow!

4. Check the current minimum and find out what its priority is. (You can remove the min, check its value, and then add it back in.) Then use the changePriority() method to reduce the priority

of the element that you wish to remove, such that its new priority is less than that of the current minimum. Then, remove the (new) minimum.

5. Suppose you wish to change the priority of the element that has some name. Simply use removeName() to remove the object from the heap. Now add(name, newPriority).

6. removeMin and findMin would take O(n) since you might have to scan the whole list. add() would take O(1) if you wanted to be fast (add to the front of a linked list, add to the end of an array).

7. If you order from small to large then removeMin() would be O(n). It would be better would be to order from large to small so that removeMin would be O(1). Adding an arbitrary element would be O(n) since you would have to shift all the elements in the worst case. findMin would be O(1) since the array is ordered.

8. To traverse the priority queue takes time O(n). For example, to traverse a heap and look for a particular object (or object name), you might need to look at all elements in the heap. By contrast, using a map (e.g. a hashmap as in Assignment 1), you only need time O(1) to index an arbitrary element. See also Q13 below.

9. [updated Jan. 29] The only benefit would be if you wanted to access objects by their priority, rather than by their name. The idea here is that if the objects were ordered by priority, then you could do a binary search and access objects having the desired priority in O(log n).

   If you just wanted to use the indexed priority queue in the usual way, though (namely add elements, removeMin, and change priority) then there would be no advantage in using an ordered array and indeed there would be a disadvantage: it would take O(n) to change a priority, since in worst case you might require a shift of all elements of the array to maintain the proper ordering of priorities. Note that this is worse than O(log n) which is the amount of time you need to maintain a heap, after a changePriority or removeMin operation. (Finally, note that if you used a map to index the objects by their name, then it would take O(1) to access the object as in the usual case. Nothing has changed there.)

10. The algorithm starts at index n/2 where n is the number of nodes. [Verify that nodes n/2 + 1, … , n are all leaves, using the parent child indexing relationship.] In particular, each leaf is (trivially) the roots of a (sub)heap. Now, for any iteration i in the algorithm, if the left and right children of node i are themselves the roots of heaps, then after you apply downHeap on node i, you can be sure that node i will be the root of a heap. (I prove this below.) But then when the algorithm terminates when i=1, we are sure that the node i=1 will be the root of the heap, that is, the whole tree is a heap.

11. Its no problem if two objects objects have the same priority. You just have to accept that ties will be broken arbitrarily. But this doesn't cause problems with the algorithms (at least, I don't see how). That's different from, say binary search trees or other maps, where we insist that each key has AT MOST one value, i.e. there are no two pairs that have the same key.

    (The proof mentioned above is by induction on the height of the tree: the base case is trivial i.e. the tree is a single node. Now suppose the claim is true for trees up to height k. Consider a node that is the root of a tree of height k+1, such that the left and right subtrees of the node are heaps. Then what does downheap do? It begins by swapping the key at the root with the child having the smaller key, and then recursively calls itself on the child. But by the induction hypothesis, after the first swap, the child is the root of a heap of height k. Its easy to verify that the root at height k+1 is now also a heap, since its root key is smaller than the keys of its two children, and the two children are each heaps. That's the end of the proof by induction.)

12. No, but you should be able to understand it. And you should know the formula for a geometric series. Its easy to derive using $(1 + x + x^2 + \ldots x^n)*(1 – x) = ?$

13. [updated Feb 2] It only matters if you changing priorities m times. If you used a linear search to find the object name each time, then that would take O(n m) which is bigger than the O(n) time it took to build the heap initially. By contrast, if you use say a nameToIndex hashmap, then that would be only O(log n) for each changePriority, namely O(1) for each find O(log n) because you need to upheap or downheap. Thus, it takes O(m log n) to change m priorities, which is less than the O(m n) which would be used if you just linearly scanned the array to do the find.