Jonathan Petersen
CS 5050
October 29, 2015

Homework 4

1. Given k sorted lists $L_1, L_2, \ldots, L_k$, with $1 \le k \le n$, such that each list contains some numbers in sorted order (e.g., in descending order) and the total number of numbers in all lists is n, using a heap, design an O(n log k) (not O(n log n)) time algorithm for sorting all n numbers in the k sorted lists.

Create a max-heap with the first element of the k sublists. Place the root into the sorted result set, then add the next element from the list that the root came from. Percolate down, spending log(k) time to reheapify the heap. Continue this procedure until no elements remain in the sorted sublists, and you'll end up with the fully sorted result set. This will take log(k) time for each of the n elements, or O(n log(k)).

2. Let A be an array of n distinct elements that store a max heap (a max heap is one that stores the largest key at its root). Each element of A is also called a "key" of the heap. The largest key of A can clearly be reported in constant time (simply by looking at the key stored at A[1], without removing it from A). We denote the operation of reporting the largest key in A (without removing that key from A) by Report-Max(1). A generalized operation of Report-Max(1), denoted by Report-Max(k) can be defined as follows: Given an integer k as input, with $1 \le k \le n$, report the largest k keys (i.e., the 1st, 2nd, ..., kth largest keys) in A without removing them from A. Design an algorithm to implement Report-Max(k) in O(k log k) time. (Note: Although k can be equal to n in the worst case, k in general is much smaller than n).

After working on this problem by myself for a while, I talked about it with Richard Hansen (who also did not know the answer) and we discovered this solution:

For k = 1, return A[1]. Then recursively examine A[2] to A[n], and percolate down the top value. This should create another max-heap from A[2] to A[n]. Repeat this procedure k times, and then once you return the first k elements of A should be the sorted set of 1st through kth largest elements.

3. 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 (you may assume no two keys in T are the same). Let h denote the height of T. The range query is 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 $x_l \leq x_r$. Note that x_l and x_r need not be keys stored in T.

Assuming a search function that locates the next-highest node for a search parameter that does not exist, begin by recursively searching for x_l. Begin the result set from x_l, then begin traversing back up the tree from child to parent, appending the key value to the result set and then adding the in-order traversal of each right subtree to the list of results. Once you reach the lowest common ancestor between x_l and x_r, add it to the result set and begin traversing down the search path to x_r. Add the in-order traversal of each left subtree followed by the node key to the result, finally ending at x_r. If the last node is equal to x_r (x_r is in the tree), add it, otherwise ignore it. If the search algorithm finds the next-smallest number, you'll need to do this comparison on x_l instead, or if the search algorithm is indeterminate the comparison must be run on both endpoints.

This approach requires two searches at O(h) time and several in-order traversals at a combined O(k) time, where k is the number of nodes in between x_l and x_r. Therefore this algorithm takes O(2h + k) or O(h + k) time.

4. 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 $x_l \leq x_r$, the operation range-sum(x_l,x_r) reports

the sum of the keys in T that are in the range $[x_l,x_r]$.

For each node, maintain a second data member that stores the sum of that node's data and all of its children's data. This will take constant time, as for any node v you can compute the sum with v.left.sum + v.data + v.right.sum.

Then, exactly like the range-min operator we did in class, starting from the common ancestor of x_l and x_r traverse the tree down to x_l, adding up the values of any nodes greater than x_l and the sums of their right subtrees. Then add up all of the data values and sums of left subtrees down the search path to x_r, ignoring any nodes that are greater than x_r. Finally, add these two values together with the data value from the common ancestor, and return this value as the result.

This algorithm requires O(h) time for the two branch traversals, and O(1) time for the sum calculations, leading to a combined time complexity of O(h).