Homework 2 *

COT5405

September 2021

1. **Prove or disprove that a HEAP can support the following operations in $o(\log n)$ time: (i) `insert`(x)`:` insert an element x into the HEAP and (ii) `extract_min_max`: extract either the minimum or the maximum element (the HEAP won't tell if the extracted element is the min or max) [Hint: Prove by contraction via sorting lower bound].**

   (a) heap insertion:

   Assume we can perform insertion in worst case $o(\log n)$ time. Binary heaps, by definition, are complete binary trees, in that all levels of the heap (except perhaps the last level) are full. Assume we have a min heap, and we insert the contents of an array sorted in descending order. During the bubble-up process of insertion, in the worst case a node will need to travel all the way up to the root from the leaf level. To achieve $o(\log n)$ insertion, we would have to need less than $\log n$ comparisons for the bubble-up process of insertion in the worst case. But, since the distance an inserted node travels will be proportional to the height of the tree, which is $h = \log n$ in balanced binary trees, the average- and worst-case runtime of insertion is $O(\log n)$, so $o(\log n)$ insertion is not possible except in the best case (or in amortized analysis).

   (b) heap min/max extraction:

   Assume we can do this in $o(\log n)$ time. After every extraction, we must compare the previously-extracted element with the currently-extracted element and note whether it is greater than or less than it. Repeating this for all n elements in the heap will let us know whether the heap extracted the min or max. Simply extracting the root n times and sorting the results to check would take $O(n \log n)$ time. We are claiming to achieve the result faster (in $o(\log n)$ time), which contradicts the proven lower bound for sorting (we cannot extract all the roots and compare them in less than $\log n$ time by definition; all nodes must be processed after all). Also, more empirically, since extracting the root takes $\log n$ time, and this is done n times (with $O(1)$ comparisons to the previously-extracted elements each time), we already see that the $o(\log n)$ runtime is exceeded.

2. **A binomial heap can perform both insertions and extract min operations in $O(\log n)$ worst-case time. By proposing an appropriate potential function, show that the amortized cost of insertion is $O(1)$ and the amortized cost of extract min is $O(\log n)$.**

   - let the potential function $\Phi(h)$ be defined as follows:

     $\Phi(h_i) = c \cdot k$, where k is the number of merged binomial trees in the heap after the i^{th} operation

     Of course, for an initialized binomial heap $\Phi(h_0) = 0$, and the number of trees only grows, so $\Phi(h_i) \geq 0$ for $i \geq 0$.

   - $O(1)$ insertion:

     The actual cost of insertion is the number of merged binomial trees already in the binomial heap, plus one extra:

     actual cost: $c_i = k + 1$

     The amortized cost of the operation is the actual cost plus the change in potential from performing the operation:

     $m = c_i + \Delta\Phi(h_i)$

     $= k + 1 +$

*For some questions, I received guidance from or collaborated with: TA (office hours)

3. **[Range Query Data Structure]: Let S be a (dynamic) set of items, where each item is associated with a unique "key" (a positive number) and a unique "weight" (a positive number). Therefore each item can be represented as a (key, weight) pair. Design an $O(n)$ space data structure that can support the following operations on S in O(log n) time, where n is the cardinality of S: (i) insert a new item into S (ii) RangeQuery[a,b]: report the item with the highest weight among all items in S with the key within the range [a,b].**

   We can approach this problem using a modified form of a binary search tree. To see how, I'll first be laying out some things we need to store in each node of our data structure for S:

   - the key for a node, which will be the primary means of "indexing" in our data structure (in other words, the binary tree ordering property will be followed by the keys)
   - the weight/value for the node
   - the maximum weight of the node's left subtree
   - the maximum weight of the node's right subtree

   The reason I choose to store the max weights of both subtrees is to make searching for the max weight take less than an additional $O(k)$ time, where $k = b - a$. If these values were not stored, we would first have to find the smallest key $\geq a$ and the largest key $\leq b$, and then search every node with keys between those values exhaustively. But, since we maintain and update these values as we insert nodes into the tree, all we have to do is find the node $\geq a$ and the node $\leq b$, and do a series of $O(1)$ comparison between a couple of values along the way between the two. The comparisons needed would follow these cases:

   Putting this all together, here's a more exact description of how the data structure and its functions:

   - inserting a new item into the data structure:
     (a) create a node that contains the given key, the given value, and `maxleft` and `maxright`, which may be initialized to $-\infty$
     (b) insert the node into the data structure following a BST's insertion steps; that is, do a binary search for the node's key in the tree and create a left/right child for the last node found, as appropriate
     (c) as you are traversing the array in this binary search fashion, also update the "maxleft" or "maxright" values of the nodes you pass through as needed. For example, if insertion makes the node to be inserted go right of the current node, update the current node's "maxright" value with $max($`curr.right`$,$ `node.weight`$)$. This ensures that the "left" and "right" values of all nodes are kept up-to-date as we insert more nodes. Since this comparison incurs only constant time for each step spent traversing the tree, we can keep this in $O(\log n)$.
   - RangeQuery[a, b]:
     (a) find the node with key $\geq a$ using binary search. Store the current max value seen, which will just be `curr.weight`
     (b) we now search for the node with key $\leq b$. Rather than search for it directly, we should start from the lowest common ancestor of this node and the node found in (a) (which we can easily determine as the node where the search for a and the search for b cause us to take different child nodes). From this ancestor's right child onward, we will compute $max($`curr.weight, curr.maxleft, currMaxWeight`$)$ until we reach the last node $\leq b$. By the end of this traversal, the max value within $[a, b]$ will be stored in `currMaxWeight`, and we can return it.
     (c) The reason we start from the right child of the ancestor of the nodes roughly corresponding to a and b is to ensure that we don't accidentally include weights outside the interval. For example, if a left child of a's node was larger than a's weight, it would also be in the `maxleft` of the ancestor, which would cause us to arrive at an incorrect result when performing step (b) (since we check the `maxlefts` there). So for that reason, we start from the common ancestor's right child when doing the `maxleft` comparisons.
     (d) as we move along in search of the node with key $\leq b$, we compare our current max value with the max of the current node's weight and its `maxleft` value, and take it if it's larger. By doing this, means we don't have to actually search the left subtree.

- the above operations in this data structure run in $O(\log n)$ time. Insertion into the tree takes $O(\log n)$ time (since we are just doing $O(1)$ comparisons for the subtree maxes, and these are done $\log n$ times). The range query also only incurs time from searching the tree for the lowest common ancestor (divergence point) of a and b, which can be done in $O(\log n)$ time as well. Finding the nodes with the smallest key $\geq a$ and largest key $\leq b$ is just an extension of the search, which is also done in logarithmic time. When we walk along the nodes from the ancestor towards b, we also only look at one node per level, and incur logarithmic time. These are all additive (and some, like the ancestor search and search for a and b, can be done simultaneously), so our operations all run in $O(\log n)$ time with a relatively balanced binary search tree. If we are worried about the worst case where the BST devolves to a linked list, we can always just use a self-balancing BST like an AVL tree, and maintain the logarithmic runtime for the two operations.

4. **Design an $O(k \log k)$ time algorithm to find the kth smallest number from a set of n numbers arranged in the form of a binary min-heap (where $n \gg k$). Improve the time complexity to $O(k \log \log k)$. Note that the trivial solution takes $O(k \log n)$ time (i.e., by simply doing extract min k times). [Hint]**

   To achieve an $O(k \log k)$ algorithm, we can use a two-heap strategy. We can achieve this time by observing that the k^{th} smallest element in a min-heap will be at or above the k^{th} level in the heap, which means that section of the heap is all we need to interact with. However, we can't perform any of the heap-related operations in our original heap since it contains all n elements, which would give us a $O(k \log n)$ runtime. This is why we need the second heap, which we can keep at a size of at most k, allowing us to achieve $O(\log k)$ heap operations.

   We will first start with the original heap, and a second heap that initially only contains the root of the original heap. Then, for $k - 1$ steps, we can do the following:

   (a) extract the min element from the second heap

   (b) find the extracted element's children in the original heap (they'll be at indices $2i$ and $2i + 1$, where i was the index of the extracted element in the original heap) and insert them into the second heap

   By the end of the $k - 1$ steps, the min element in the second heap will be the k^{th} smallest element in the original heap. For the k iterations, we incur $\log k$ work to perform heap insertion and removal operations in the second heap (remember, we do not actually do any re-heapification on the original heap, as we just do read-only interactions with children nodes via array indexing; all heapify operations are done in the second heap). This gives us a final runtime of $O(k \log k)$.

   To reach an $O(k \log \log k)$ runtime, we can use the approach suggested in the paper, which is more nuanced but is still somewhat of an extension of the previous algorithm, in that we are still using an auxiliary heap but are interested in solving the problem in subtrees of the heap at a time.

   Before describing the algorithm, I've described some terminology from the paper that must be discussed for it to make sense:

   - a **clan** is a grouping of the elements from the input heap. In this algorithm, all clans will have a size of $\lfloor \log k \rfloor$. The first clan (C_1) would contain the first $\lfloor \log k \rfloor$ smallest elements, the second clan C_2 would have the 2nd $\lfloor \log k \rfloor$ smallest, etc. Clans are generated from previous clans, one by one in an iterative manner, as we progress through the algorithm.
   - **offspring** of the i^{th} clan, denoted $os(C_i)$ in the paper, are simply the $\lfloor \log k \rfloor$ smallest children of the elements of a clan (from the original heap) that are not also in the clan. For example, if an arbitrary node (that has 2 children) and its right child are both part of a clan, its offspring set could include the node's left child and both of its right child's children, but not the right child itself.
     - in the paper's example, a clan being "grown from" another clan simply means that the previous clan's $\lfloor \log k \rfloor$ smallest offspring are selected to form the new clan. Keep in mind that a clan can have more than $\lfloor \log k \rfloor$ offspring, but here only the smallest $\lfloor \log k \rfloor$ are selected

- the **poor relations** of a clan C_i are any offspring of the C_{i-1}-st clan than were not part of C_i. As mentioned in the previous point, a clan can have more than $\lfloor \log k \rfloor$ offspring; these leftover offspring form the poor relations of the next clan.
- the **representative** of a clan is just the largest element in that clan

Without further ado, here's how the SEL1 algorithm would go about finding the k^{th} smallest element in $O(k \log \log k)$ time:

(a) we start with the input heap H_0 and an auxiliary heap H_2

(b) use H_2 to find the $\lfloor \log k \rfloor$ smallest elements in H_0; these will form our first clan. Determine the set of this clan's offspring $os(C_1)$, and set $pr(C_1)$ to the empty set (as there are no poor relations for the first clan).

(c) find the representative of C_1 and insert it into a different heap, H_1

(d) For $\lceil \frac{k}{\lfloor \log k \rfloor} \rceil$ steps, do the following:

i. extract the min element of H_1 (initially the representative of C_1)
   - let C_j be the clan that the extracted element was a part of
   - with H_2, find the $\lfloor \log k \rfloor$ smallest offspring of C_j; these elements will form clan C_i (i represents the number of clans generated until now)

ii. determine this new clan's offspring set and it's associated poor relations set

iii. if the poor relations set is non-empty, again find the $\lfloor \log k \rfloor$ smallest offspring, but this time relative to the elements in the poor relations set. These will be clan C_{i+1}

iv. again determine the offspring set and poor relations set for C_{i+1}

v. insert the representatives of both C_i (the clan formed from the previous clan's offspring) and C_{i+1} (the clan formed from the new clan's poor relations, i.e. the next $\lfloor \log k \rfloor$ elements that were the previous clan's offspring but were not chosen to form C_i) into H_1

(e) using the last element removed from H_1 at the end of the loop, find the set of elements in H_0 that are $\leq$ that last-removed element. Note that the last-removed element from H_1 will be the largest of the $\lceil \frac{k}{\lfloor \log k \rfloor} \rceil$ clan representatives that we traversed through in the above loop, making it at least as large as the k^{th} largest element.

(f) from the set of elements achieved in step (e), directly select the k^{th} smallest element, which can be done in using a linear-time selection algorithm like median of medians or PICK (in this case, $O(k)$, since the set of elements selected will be approximately k)

The runtime of this algorithm is $O(k \log \log k)$. The loop executes $\lceil \frac{k}{\lfloor \log k \rfloor} \rceil$ steps, and each time we do $\log k$ work (for the heap operations in H_1), giving us $O(k)$. However, in each step, we are also forming clans, which we'll have at most $2\lceil \frac{k}{\lfloor \log k \rfloor} \rceil + 1$ of, and the time taken to create each one is $O(\log k \log \log k)$. Simplifying a bit by disregarding coefficients and constants, we see that we'd thus take $\frac{k}{\log k} \cdot \log k \log \log k = O(k \log \log k)$ time to create all the needed clans. Finally, after the loop ends, we end up with the set containing $O(k)$ elements that are less than the element last removed from H_1, and selecting the k^{th} smallest element from this set would take linear time $O(k)$. So, we'd have a runtime of $O(k \log \log k + k) = O(k \log \log k)$.

5. **Recall how we solved the following problem in class: Given a (read-only) array A[1,n] of n numbers, if it exists, find the number that appears more than n/2 times. Note that the time complexity for that approach is O(n). Also note that A is "read-only," and the auxiliary space used (i.e., to maintain two variables) is O(1). Now, let's generalize this problem: Given a (read-only) array A of n numbers and an integer $k > 1$, determine the numbers that appear more than n/k times, if they exist. Design an O(n log k) time solution that uses only O(k) auxiliary space.**

   We can solve this problem in a manner similar to the one discussed in class, just modified to track more than one possible frequently-occurring element. The algorithm would go like this:

   (a) initialize an auxiliary data structure like an associative binary search tree (or hash map to give us a better expected runtime but worse worst-case runtime) of size $k-1$ to hold

the elements that can *potentially* occur more than $\frac{n}{k}$ times. Note that if $k = 2$, we'd arrive at the problem discussed in class and the array would be of size 1 (to hold the one element appearing more than $\frac{n}{2}$ times).

(b) Like the algorithm discussed in class, walk through the array element by element, but now we track $k - 1$ elements by mapping the key (the element of the array) to a "count-like" value. The first time the element is seen, this value would be 1.

(c) Again similar to the base algorithm, we'd increment a key's count if we see it again, but only if its key already exists in the array. If it doesn't (i.e. this is the first time we're seeing the element), we'd make a new key in the array for it (also initialized to 1).

(d) In the case that the array is already filled and we can't make a new key for a new unique element, we'd instead decrement the values of *all* keys in the array by 1. And as was the case in the base algorithm, if the count of a key ever reaches 0, we remove it (this is analogous to what we did before, in that it will allow a new, potentially unique element into the "top-k-frequently-occurring" group, and previously that group had a size of 1).

(e) Continue this process as described in steps 2 - 4 (inclusive) until the end of the input array is reached. At this point, the array of $k-1$ elements will contain all the candidates for occurring more than $\frac{n}{k}$ times; however, it's not guaranteed that *every* element in that array will occur more than $\frac{n}{k}$ times (for example, the last element in the array could've entered the array just once at the very end and taken a slot thus only having a count of 1).

(f) Because of this, it's necessary to again scan the input array to confirm whether the final $k - 1$ candidates actually occur more than $\frac{n}{k}$ times. We'd simply iterate over the input array again and maintain counts of only the final candidates; any that occur less than or equal to $\frac{n}{k}$ times can be safely discarded. The remaining candidates will be our final result.

This approach runs in $O(n \log k)$ time with $O(k)$ space if we use a binary search tree/sorted array approach for storing the size $k - 1$ candidate pool. This is because, for every element in the array, we need to find whether its key exists by performing a binary search in the tree, which will take $O(\log k)$ time. If the key doesn't exist, we'd have to create it in the tree (which also takes $O(\log k)$ time). Incrementing/decrementing the count attribute of a key would be a constant-time operation. Thus, for all n elements in the array, we'd incur $O(\log k)$ work, giving us a runtime of $O(n \log k)$. The scan to confirm the validity of the final $k-1$ candidates would also take $O(n \log k)$ time since for all n elements, we do a binary search to determine whether it is a final candidate, and if it is, maintain a count for it.

If we instead use a hash map, we'd get an expected runtime of $O(n)$ with $O(k)$ space, as each lookup or key insertion into the size $k - 1$ hash map would be amortized $O(1)$. Of course, in the worst case, the runtime would be $O(n \cdot k)$ due to the possibility of hash collisions.