The exam will be in class (one hour) and closed book. You may use one page of notes (one side of an 8.5×11" piece of paper, hard copy required), which will be handed in with the exam. The sums and recurrence relations tables and logarithm/exponent rules from class will be provided if needed.

It is strongly recommended that you treat this as a practice exam and work on it in a similar environment.

---

1. You need to implement an ADT characterized by the following operations:

   - `insert(x)` — insert element x
   - `replace(x,y)` — replace element x with y
   - `rank(x)` — return the number of elements larger than x
   - `successor(x)` — return the smallest element larger than x, if any

   Explain how to implement this ADT using various data structures. You can refer to standard building blocks like "traverse", "sequential search", or "bubble up" without further explanation, but do need to explain "insert", "remove", and "find" instead of just citing them as standard operations.

   Also give the running time for each operation. It should be clear from your description of how the operation is carried out where the running time comes from so you don't need additional explanation.

   (a) unsorted array

   (i) `insert(x)`
   Add x at the end of the array, $O(1)$.

   (ii) `rank(x)`
   Since the array is unsorted, it is necessary to traverse the array, counting the number of elements larger than x. $O(n)$

   (b) sorted array

   (i) `replace(x,y)`
   Binary search can be used to find x, but then everything between x and y needs to be shifted to make room to insert y in the correct spot in the sorted order. $O(\log n)$ for binary search but up to n elements may need to be shifted so $O(n)$ in total.

   (ii) `rank(x)`
   Binary search can be used to find x, then arithmetic (size minus the index of x minus 1) to compute the rank because, assuming no duplicates, everything after x in the sorted order is exactly everything larger than x. $O(\log n)$

If duplicates are possible, find the first element y larger than x by scanning right once x's position is found. x's rank is then size minus the index of y. $O(\log n + s)$ where s is the maximum number of copies of any one element — this could be $O(n)$ in total if every element is the same.

(iii) `successor(x)`
Binary search to find x, then x's successor is the next element after x (if there is one). $O(\log n)$
If duplicates are possible, find the first element y larger than x by scanning right once x's position is found. y is the successor. $O(\log n + s)$ where s is the maximum number of copies of any one element — this could be $O(n)$ in total if every element is the same.

(c) unsorted linked list

(i) `insert(x)`
Put x at the head, $O(1)$.

(ii) `successor(x)`
Since the array is unsorted, it is necessary to traverse the array looking for the smallest element larger than x. $O(n)$

(d) sorted linked list

(i) `replace(x,y)`
Delete the node containing x, then insert a new node containing y. Both of these operations involve scanning forward from the head of the list to find the node before x or before the insertion point of y, then a constant number of steps to create a new node (for insert) and relink the nodes on either side (for both delete and insert). $O(n)$

(ii) `successor(x)`
Traverse the list from the head until x is found, then return the element in the next node. If duplicates are possible, scan right from x's position until the first larger element is found. $O(n)$ in both cases.

(e) balanced search tree (AVL or 2-4)

(i) `replace(x,y)`
Delete x, then insert y. Both operations involve searching for an element: start at the root and (for an AVL tree) go left if the element is less than or equal to the current node's value and right otherwise. Two $O(\log n)$ operations is $O(\log n)$ in total.

(ii) `rank(x)`
Have each node also store the number of elements in its subtree (including its own). Search for x, adding size(right)+1 to the running total each time the search goes left. When the node containing x is found, add size(right) to account for the descendants of x that are bigger than x. $O(\log n)$ since this involves $O(1)$ work at each level of the tree.

If duplicate elements are possible, assume that they have gone into the left subtree (so the right subtree of a node only contains elements that are strictly greater). Then the same procedure works — the first x found is the last one according to an in-order traversal (all other xs will be to the left and none will be to the right).

(iii) `successor(x)`
Search for x. If the node containing x has a non-leaf right child, go right and then go left until a node y whose left child is an (empty) leaf is found. If x only has a leaf right child, go back up the tree until the first ancestor y of x where $y > x$ is found. (This is the first ancestor of x where x is in the left subtree.) In both cases, y is x's successor. If there is no such ancestor of x (x is always in the right subtree), x is the bigget element and there is no successor. $O(\log n)$ because there are at most two passes through the tree: one root to leaf and one leaf to root.
If duplicate elements are possible, assume that they have gone into the left subtree (so the right subtree of a node only contains elements that are strictly greater). Then the same procedure works — the first x found is the last one according to an in-order traversal (all other xs will be to the left and none will be to the right).

(f) heap

Assume a max-heap.

(i) `replace(x,y)`
Locating x in a heap requires $O(n)$ traversal. Once found, replace x with y and then bubble to fix the ordering property — if $y > x$, bubble up; if $y < x$, bubble down. Bubbling is $O(\log n)$ but replace is $O(n)$ overall because of needing to find x in the first place.

(ii) `rank(x)`
Traverse the heap and count the number of elements larger than x. $O(n)$
However, since in a max heap bigger elements are higher up, with the largest element of all at the root, it is possible to do a DFS or BFS traversal and only look at 3*rank(x) elements — the rank(x) elements larger than x that are being counted plus up to two children $\leq x$ each. (Once an element $\leq x$ is found, the heap ordering property means all of its descendants will also be $\leq x$.) This is $O(\text{rank}(x))$ — which still ends up being $O(n)$ worst case and on average.

(iii) `successor(x)`
This is similar to rank, but instead of counting the number of elements larger than x, keep track of the smallest such element encountered. $O(n)$

(g) hashtable (separate chaining or open addressing)

(i) `replace(x,y)`

Delete x, insert y. With separate chaining, hash x and then search the elements in slot $h(x)$ to find x. To insert, hash y and then add y at the head of the list in slot $h(y)$. $O(1)$ expected time.

(ii) `rank(x)`
The elements in a hashtable are not sorted, so traverse the hashtable counting the number of elements $> x$. $O(N + n)$ for separate chaining and $O(N)$ for open addressing because every slot of the array must be checked, even if empty.

(iii) `successor(x)`
This is similar to rank, but instead of counting the number of elements larger than x, keep track of the smallest such element encountered. $O(N + n)$ or $O(N)$

2. Design a data structure which holds a collection of elements, each of which has a key and a numeric value. It should efficiently support the following operations:

   - `insert(k,v)` — insert the value v associated with the key k
   - `replace(k,v)` — replace the value associated with key k with v
   - `getValue(k)` — return the value associated with key k
   - `getMinValue()` — return the smallest value in the collection
   - `add(k,a)` — add a to the value associated with key k
   - `addAll(a)` — add a to all of the values

   Give an efficient implementation for this ADT. No operation should have a running time worse than $O(\log n)$ in the worst case, though for full credit you should obtain $O(1)$ time where possible.

   Describe how the elements are stored in your data structure and how each of the operations are carried out. You can refer to standard operations like "insert into a balanced binary search tree" without further explanation.

   Also give the running time for each operation. It should be clear from your description of how the operation is carried out where the running time comes from so you don't need additional explanation.

   The idea is to use multiple structures: have a hashtable storing key-value pairs to support `getValue`, a min-heap storing elements to support `getMinValue`, and a global offset to support `addAll`. The offset records what has been added to all of the elements so the values stored in the heap and hashtable are actually `offset` less than the real values.

   - `insert(k,v)`

Insert $(k, v - \texttt{offset})$ into the hashtable and $v - \texttt{offset}$ into the heap. $O(1)$ expected for the hashtable insert, $O(\log n)$ for the heap insert so $O(\log n)$ total.

(Subtracting the offset is needed when the value is stored so that an immediate `getValue(`k`)`, which returns the looked-up value plus the offset, will correctly return v.)

- `replace(k,v)`

  Both the heap and the hashtable need to be updated.

  The heap stores elements — look up the old element v' associated with k in the hashtable, then remove v' from the heap and add $v - \texttt{offset}$. $O(1)$ to look up v', $O(\log n)$ to remove and insert.

  The hashtable stores key-value pairs — look up the entry associated with key k and replace the element with $v - \texttt{offset}$. The location in the hashtable doesn't change so there's no need to remove the old and insert the new. $O(1)$ expected to look up k, $O(1)$ to replace the element.

  Total: $O(\log n)$

- `getValue(k)`

  Look up the value v associated with k in the hashtable, then return $v +$ `offset`. $O(1)$ expected.

- `getMinValue()`

  Return `heap.getMin()` + `offset`. $O(1)$

- `add(k,a)`

  a is added to only one value, so implement this as `replace(k,getValue(`k`)+a)`. $O(\log n)$

- `addAll(a)`

  Add a to the global offset. (Since `getValue` and `getMinValue` add `offset` to what they return, this has the effect of adding a to every value.) $O(1)$