Q1. (a) Consider the following algorithm:

Algorithm 1 IS-SMALLER(A, x, k)

Input: A min-heap with n distinct keys in an array $A[1, \ldots, n]$, a value x, and an integer k.
Output: “yes” if the k-th smallest key in the heap is smaller than x; “no” otherwise.

```
1:  c ← 0                      // counts the number of keys a_c < x; if c = k, then we're done!
2:  S ← ∅                      // sets an empty stack for processing suitable keys
3:  S.push(1)                  // commence at the root
4:  while S.size() > 0 do
5:      i ← S.pop()
6:      if i ≤ A.size() and A[i] < x then
7:          c ← c + 1
8:          if c = k then
9:              return "yes"
10:         end if
11:         l ← 2i
12:         r ← 2i + 1
13:         if l ≤ A.size() then
14:             S.push(l)
15:         end if
16:         if r ≤ A.size() then
17:             S.push(r)
18:         end if
19:     end if
20: end while
21: if c < k then
22:     return "no"
23: else
24:     return "yes"
25: end if
```

Algorithm 1 just counts, via iterative DFS, the number of keys less or equal to x starting from the root. In the event that k of those are exhausted, it returns “yes”; otherwise, it returns “no”.

(b) In analyzing the runtime of Algorithm 1, we notice that all lines are canonical constant time operations, except the while loop in line 4, which bears the bottleneck of the algorithm. So, in order to show that Algorithm 1 runs in $O(k)$, it suffices to show that the while loop in line 4—representing the total number of keys processed by S—is linear in k.

> **Claim** — For arbitrary inputs A, x, and k, the number of keys processed by S does not exceed $2k - 1$.

Proof. The number of keys added to S with value less or equal to x is at most k. Each key with value greater than x is (by lines 11 and 12) a child of a key less or equal to x. Because every key except the root must have a parent key with value smaller or equal to x, the number of keys with value greater than x must be at most $2(k-1) - (k-1)$ since $k-1$ of the $2(k-1)$ child keys have values smaller or equal to x. So, in the worst case, S processes k keys smaller or equal to x and $k-1$ greater than x, which amounts to $2k-1$ keys in total! □

With this established, $T(n) = O(2k-1) + O(1) = O(k)$, as desired!

Q2. (a) We proceed with the following strategy: we augment T by transforming it into an order statistic tree and supplement its nodes with one additional field, call it *size*. This new field, for a given node, say v, will store the size of the subtree rooted in v, which in more practical terms amounts to the number of descendants of v plus one (see Figure 1 for details). With this strategy in mind, we wish to maintain the following invariance over *size*: $v.size = v.left.size + v.right.size + 1$.

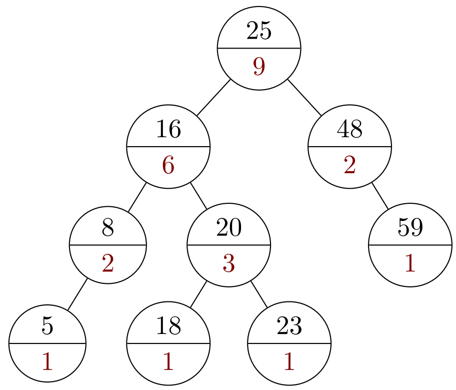

Figure 1. Order Statistic Tree constructed from T. Note each node is augmented with an additional field (in red) denoting its *size*.

(b) Now, consider the following algorithm, which given a value x, computes its *rank*:

Algorithm 2 RANK(T, x)

Input: An augmented and balanced order statistic tree and a value x.
Output: The position of x (one-indexed!) in the linear sorted list of keys of the tree.

1: $r \leftarrow 0$ // the rank of x in T
2: $v \leftarrow T.root$
3: **while** $v \neq$ NIL **do**
4:     **if** $v.key \leq x$ **then**
5:         **if** $v.left \neq$ NIL **then**
6:             $r \leftarrow r + v.left.size + 1$ // count the # of elements less than v + v itself
7:         **else**
8:             $r \leftarrow r + 1$ // if there isn't a left, just count v
9:         **end if**
10:         **if** $v.key = x$ **then**
11:             **return** r
12:         **end if**
13:         $v \leftarrow v.right$ // explore the right sub-tree
14:     **else**
15:         $v \leftarrow v.left$ // explore the left sub-tree
16:     **end if**
17: **end while**
18: **return** $r + 1$ // plus one to account for key x not in the tree

Suppose T is balanced and augmented. Then at every iteration of the while loop in line 3, Algorithm 2 reduces the problem space by half by carefully choosing to explore either the left sub-tree or the right sub-tree. This means Algorithm 2 only traverses a single path in T to find the *rank* of x. But any path's length in T is bounded by T's height—$\log n$. So, in the worst case, the length of the path traced by Algorithm 2 is exactly $\log n$, which incurs an $O(\log n)$ in its runtime.

(c) Let v be an arbitrary node in T. Then, since the *size* attribute of v only depends on information on $v.left$ and $v.right$, according to the **Main Theorem**, we can maintain the values of *size* in all nodes of T during *insertion* and *deletion* without affecting the desired $O(\log n)$ time performance of these operations. Moreover, since *size* only equips T with additional information, the time performance of *search* operations remains unchanged.

Q3. (a) Consider the following algorithm on a balanced binary search tree T whose keys are distinct:

Algorithm 3 RANGE-QUERY(v, x_l, x_r)

Input: A node v in T and two real numbers x_l and x_r, denoting a range.
Output: All keys x stored in T such that $x_l \leq x \leq x_r$.

1: **if** $v \neq$ NIL **then**
2:     **if** $x_l \leq v.key$ **then**
3:         RANGE-QUERY$(v.left, x_l, x_r)$ `// search left for potential candidates`
4:     **end if**
5:     **if** $x_l \leq v.key \leq x_r$ **then**
6:         PRINT$(v.key)$
7:     **end if**
8:     **if** $v.key \leq x_r$ **then**
9:         RANGE-QUERY$(v.right, x_l, x_r)$ `// search right for potential candidates`
10:     **end if**
11: **end if**

(b) Let k be number of keys of T who fall in the range $[x_l, x_r]$. Then, in either execution path, Algorithm 3 always terminates by reporting exactly k values of T which means the PRINT statement in line 6 is called exactly k times—one for each value in the range. Assume this operation is constant, since we have k of those we get an $O(k)$ cost for reporting/printing the values in the range. Added to this cost are the two recursive calls in line 3 and 9 that search (through only one path!) down to the height of T. Now, since T is balanced the two recursive calls incur both an $O(\log n)$ time cost each.

So, putting all together, if we let $T(n)$ represent the total time taken for Algorithm 3 to report k of n keys of T in the range $[x_l, x_r]$ then $T(n) = O(k) + O(\log n) + O(\log n) = O(k + \log n)$. This last equality follows from the fact that k may very well exhaust all nodes in the tree, in which case $k = n \gg \log n$.

Q4. (a) We augment T as follows: for every entry node v in T, we supplement v with one additional field, call it *sum*. This new field will store the cumulative sum of all the values descendants of v as well as the value of v itself (see Figure 2 for details). With this augmentation, we wish to maintain the following invariance over *sum*: $v.sum = v.left.sum + v.right.sum + v.key$.

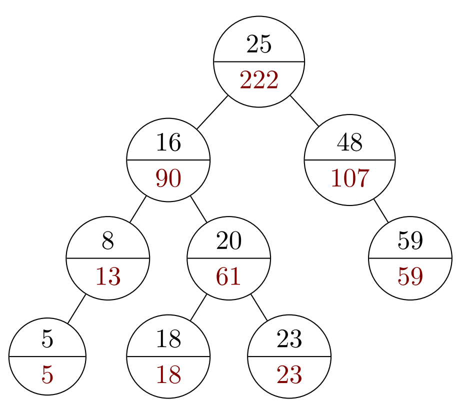

Figure 2. Augmented and Balanced Binary Search Tree constructed from T. Note each node v is augmented with an additional field (in red) representing its *sum*.

(b) Now consider the following algorithm, which given a range $[x_l, x_r]$, returns the sum of all keys of T in that range:

Algorithm 4 RANGE-SUM(T, x_l, x_r)

Input: An augmented and balanced binary search tree and two real numbers x_l and x_r, denoting a range.

Output: The sum of all keys of T in the range $[x_l, x_r]$.

```
1:  S_L ← 0                                   // sum of all keys < x_l
2:  v ← T.root
3:  while v ≠ NIL do
4:      if v.key < x_l then
5:          if v.left ≠ NIL then
6:              S_L ← S_L + v.left.sum + v.key
7:          else
8:              S_L ← S_L + v.key
9:          end if
10:         v ← v.right
11:     else
12:         v ← v.left
13:     end if
14: end while
15:
16: S_R ← 0                                   // sum of all keys ≤ x_r
17: v ← T.root
18: while v ≠ NIL do
19:     if v.key ≤ x_r then
20:         if v.left ≠ NIL then
21:             S_R ← S_R + v.left.sum + v.key
22:         else
23:             S_R ← S_R + v.key
24:         end if
25:         v ← v.right
26:     else
27:         v ← v.left
28:     end if
29: end while
30:
31: return S_R − S_L                          // sum of all keys in [x_l, x_r]
```

In analyzing the runtime of Algorithm 4, we notice that there are two almost identical pieces in its structure (namely, lines 1–14 and lines 16–29). In both of these pieces, we start at the root of the tree and make a decision to either walk left or right. We stop until a leaf node is reached. So, the paths traced by these lines (1–14 and 16–29) cannot exceed the height of T—$\log n$. This is because every time we make a decision to walk left or right, we are effectively narrowing the search space by half. But since the search space initially starts with n (the number of nodes in the tree), we are restricted to at most $\log n$ steps in that walk until n gets reduced to 1, in which case, we are in the presence of a leaf node. So, lines 1–14 and 16–29 both incur an $O(\log n)$ cost each, and a direct consequence of this is that the total running time of Algorithm 4 is:

$$T(n) = O(\log n) + O(\log n) + O(1) = O(\log n)$$

The last equality follows from the fact that $O(1)$ is also $O(\log n)$, so we get $3O(\log n) = O(\log n)$ as required!

(c) Let v be an arbitrary node in T. Then, since—per *sum* invariance in part a—the *sum* attribute of v only depends on information on v, $v.left$, and $v.right$, according to the **Main Theorem**, we can maintain the values of *sum* in all nodes of T during *insertion* and *deletion* without affecting the desired $O(\log n)$ time performance of these operations. Moreover, since *sum* only equips T with additional information, the time performance of *search* operations remains unchanged.