Question 1

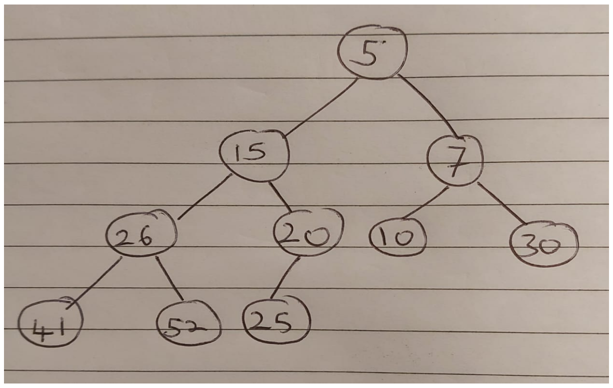
(a) Consider the following algorithm.

Algorithm 1: Is k-th smallest key smaller than x

Input: A min-heap with n distinct keys in an array $A[1,\dots,n]$, a value x and an integer k with $1 \le k \le n$.

Output: Yes if the k-th smallest key in the heap is smaller than x; No otherwise.

```
/* Here's my algorithm.
node:  represents current node (root at start)
count:  counter to keep track (0 at start) */
checkIfSmaller(node, count, k, x)
{
$flag \leftarrow false$
if $node.value != NULL$ and $node.value < x$ then
    count++
    if $count >= k$ then
        flag = true
        return flag
    if $!flag$ then
        checkIfSmaller(node.left, count, k, x)
        checkIfSmaller(node.right, count, k, x)
}

isSmaller(node, count, k, x) {
result = checkIfSmaller(node, count, k, x)
if $result = true$ then
    print Yes
else print No
}
```



Explain the algorithm.
The basic logic as mentioned in the remark is that we just need to know if the node value is lower than x and don't actually need the value.
This is kind of a depth first search approach. We start at the root, check if the node value

is lesser than x. If Yes, then we increase the count and move to the child nodes recursively until we have the count $>= k$ (indicating that, there are already k nodes smaller than x) or there are no more nodes left to check.
Suppose, Case 1: k = 6, x = 30
We check 5, and since 5 < 30 count = 1. We then follow left first traverse acc. to algo.
15 < 30: yes, c =2
26 < 30: yes, c=3
41 < 30: no, since it is more than 30, we don't traverse further
52 < 30:no, same as above
20 < 30: yes, c=4
25 < 30: c=5
7 < 30: c=6
Now we have established that there are atleast 6 values less than x, hence we can say with confidence that k-th smallest key in the heap is smaller than x.

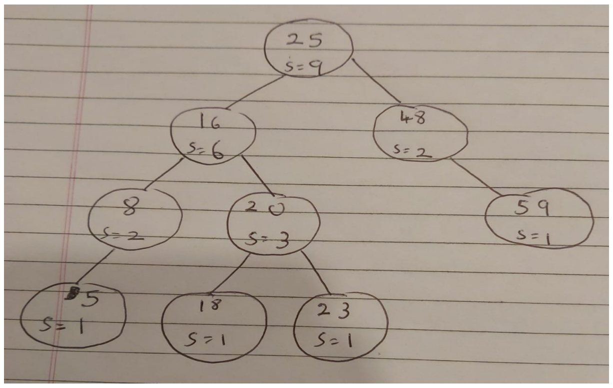
Suppose, Case 2: k=6, x = 20
5 < 20: c=1
15 < 20: c = 2
26 < 20: no, Don't process further
20 < 20:no, Don't process further
7 < 20: c=3
10 < 20: c=4
30 < 20: no, Don't process further
End of Nodes

Hence, by processing just 4 nodes we determined that the answer is no.

(b) Running time analysis.
Since we only Process the nodes with values less than x, We will require at most O(k) time (if answer is yes) and even lesser time (if answer is No).
Example: In case 1 above, k = 6 and we processed 6 nodes to determine the answer. In case 2, k = 6 and we processed 4 nodes to determine the answer.
The worst case time complexity will be O (k).

Question 2

(a) We will augment the tree such that it becomes an AVL tree with each node storing a metadata that contains the size of the node (v.size) where size = number of childs under the node including the node.
v.size = 1 + v.left.size + v.right.size
Refer diagram for the augmented tree.



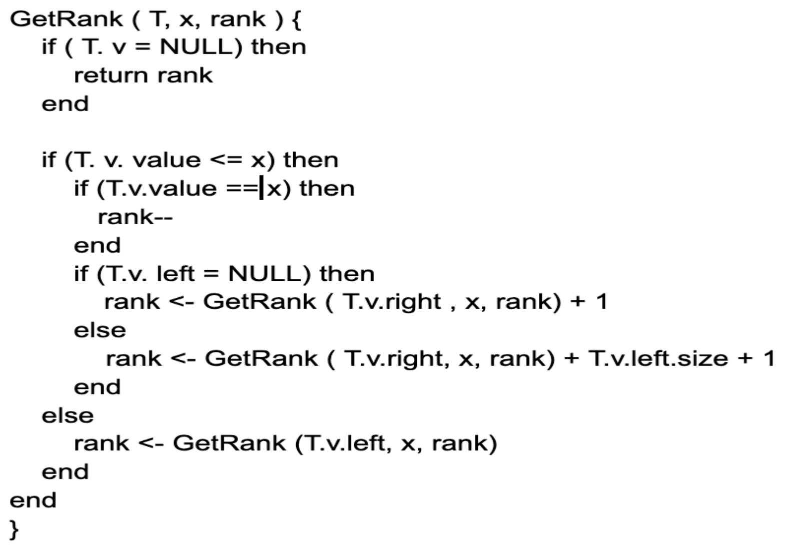
(b) Consider the following algorithm.

Algorithm 2: Determine the range(x) in a BST.

Input: an augmented balanced binary search tree (AVL) T of n nodes (each node v has v.left, v.right, and v.key and stores the size metadata), a value x (to determine rank), rank initialized as 1.

Output: Rank of x.(How does X rank in an ordered list)

```
/* Here's my algorithm.
Rank is initialized as 1 */
```



```
GetRank ( T, x, rank ) {
   if ( T. v = NULL) then
      return rank
   end

   if (T. v. value <= x) then
      if (T.v.value ==x) then
        rank--
      end
      if (T.v. left = NULL) then
         rank <- GetRank ( T.v.right , x, rank) + 1
      else
         rank <- GetRank ( T.v.right, x, rank) + T.v.left.size + 1
      end
   else
      rank <- GetRank (T.v.left, x, rank)
   end
end
}
```

Explanation with Example: case 1: Rank (21)
Current Rank = 1 (initialized value)
25 <= 21 : No, Go to left subtree
16 <= 21: Yes, and since there is a left subtree, rank = current rank + 2 (left tree size) + 1 = 4 (goto right subtree).
20 <= 21: Yes, and since there is a left subtree, rank = curr. rank + 1 (left tree size) + 1 = 6 (goto right subtree).
23 <= 21: No, Go to left subtree. Since left subtree is empty. return rank = 6.

case 2: Rank (48)
Current Rank = 1
25 <= 48 : Yes, and since there is a left subtree, rank = current rank (1) + 6 (left tree size) + 1 = 8 (goto right subtree).
48 <= 48: Yes, and since node value = x, we decrement rank once as the node is present in the tree and we don't have to count it twice current rank = 8 - 1 = 7. There is no left subtree, rank = current rank + 1 = 8 (goto right subtree).
59 <= 48: No, Go to left subtree. Since left subtree is empty. return rank = 8.

Running time analysis. This method just traverses a single path in the tree to find the rank. Worst case, it traverses the entire height of the tree making its time complexity as O (h). Since height of a balanced tree is log (n). The running time complexity of the ALgo is Log (n).

(c) According to the Main Theorm: Let A be an attribute that augments an AVL tree of n nodes, and suppose that the value of A for each node v only depends on the info in the nodes v, v.left and v.right. Then we can maintain the values of A in all nodes of the tree during search, insertion and deletionn without affecting the O (log(n)) performance of the operations. Hence, Here when we augment the tree with the size of its child nodes, performance of the other operations of the tree remains unaffected as the tree balances itself based on the size indicated by the the theorm above.

Question 3

(a) Consider the following algorithm.

Algorithm 3: List all keys in order for the given range [Xl, Xr]

Input: a balanced binary search tree T of n nodes (each node v has v.left, v.right, and v.key, a range [Xl,Xr] such that $Xl <= Xr$

Output: A list of keys in order that lies within the range

```
/* Here's my algorithm. */
```

```
GetKeysInRange( T, Q, Xl, Xr) {
  if (T.v == NULL)
    return
  end

  if (Xl < T.v.key)
    GetKeysInRange( T.v.left, Q, Xl, Xr)
  end

  if (Xl <= T.v.key and Xr >= T.v.key)
    Q.enqueue(T.v.key)
  end

  if (Xr > T.v.key)
    GetKeysInRange( T.v.right, Q, Xl, Xr)
  end
}
```

Explain the algorithm.
We kind of do an in-order traversal over here. We first check all the nodes in left subtree recursively that fall in the given range. move on to the parent and then the right sub tree similarly. This would give us the list of keys in order that lie in the given range. We only traverse those nodes that lie in the range.
Example: Consider the tree in previous example.
Suppose we need to find all keys between 16-48
Xl = 16, Xr = 48.
From root:
$16 < 25$: Yes, Goto left subtree
$16 < 16$: No, Add in queue and go to right subtree(as $48 > 16$).
$16 < 20$: Yes, Goto left subtree.
$16 < 18$: Yes, Goto left subtree.
Since left subtree is null, we return to last recursive call. Hence we print 18 (lies in range) and goto right subtree ($48 > 18$).
Since right subtree is empty, goto prev call, Print 20 and goto right subtree.
$16 < 23$: Yes, Goto left subtree.
Since left subtree is null, we return to last recursive call. Hence we print 23 and goto right subtree ($48 > 23$). Since right subtree is empty, goto prev call, Print 25 and goto right sub-

tree.
16 < 48: Yes, Goto left subtree.
Since left subtree is null, we return to last recursive call. Hence we print 48 and won't goto right subtree as (48 > 48) is not true.

(b) Running time analysis. Worst time it can take to calculate keys in range is Traverse the height of the tree twice (left and right subtree ends). 2 * log (n). and it takes O(k) to report the values obtained. In total it takes O (k + log(n)) time to run the algo in worst case complexity.

Question 4

(a) We will Augment the tree by adding a metadata 'sum', which basically stores the sum of the keys of all child nodes including the node. v.sum = v.key + v.left.sum + v.right.sum

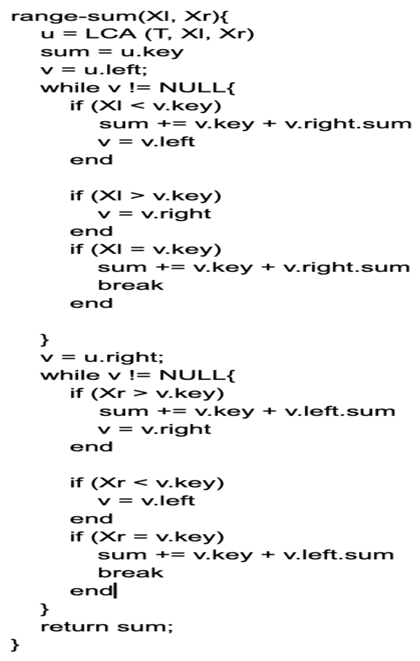
(b) Consider the following algorithm.

Algorithm 4: Calculate sum of all keys that lie in the range in Log n time

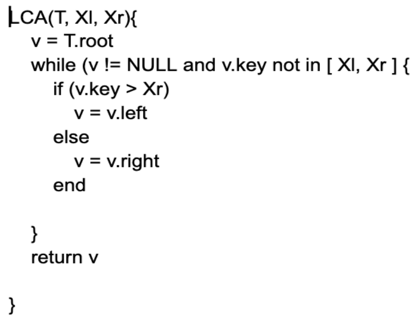
Input: an augmented balanced binary search tree (AVL) T of n nodes (each node v has v.left, v.right, and v.key and stores the sum metadata - v.sum), a value Xl (start range), Xr (end range) such that $Xl <= Xr$.

Output: the sum of the keys in T that are in the range [Xl, Xr]

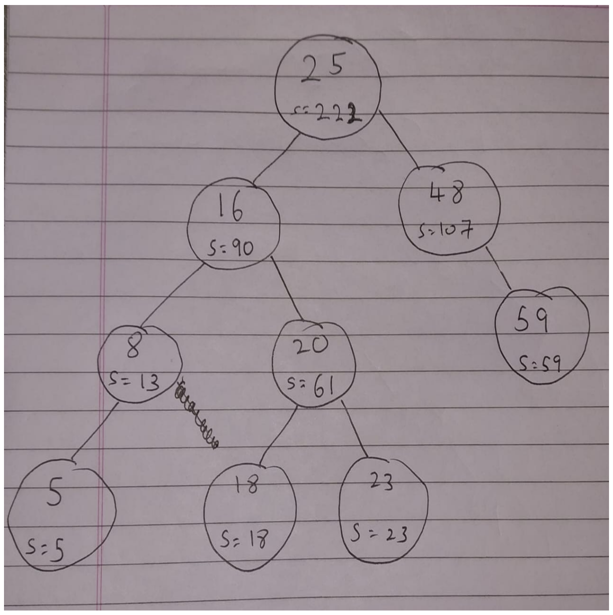
```
/* Here's my algorithm. */
```



```
range-sum(Xl, Xr){
    u = LCA (T, Xl, Xr)
    sum = u.key
    v = u.left;
    while v != NULL{
        if (Xl < v.key)
            sum += v.key + v.right.sum
            v = v.left
        end

        if (Xl > v.key)
            v = v.right
        end
        if (Xl = v.key)
            sum += v.key + v.right.sum
            break
        end

    }
    v = u.right;
    while v != NULL{
        if (Xr > v.key)
            sum += v.key + v.left.sum
            v = v.right
        end

        if (Xr < v.key)
            v = v.left
        end
        if (Xr = v.key)
            sum += v.key + v.left.sum
            break
        end
    }
    return sum;
}
```



```
LCA(T, Xl, Xr){
  v = T.root
  while (v != NULL and v.key not in [ Xl, Xr ] {
    if (v.key > Xr)
      v = v.left
    else
      v = v.right
    end

  }
  return v

}
```



Example:
Given: Xl = 7, Xr = 49
On running LCA we get root node as 25.
sum = 25 (initial for LCA node)
For Left subtree (Start node = 16):
Since $XL < 16$, sum = 25 + 16 + 61 (V.right.sum) = 102
Go left
Since $XL < 8$, sum = 102 + 8 + 0 (V.right.sum) = 110
Go left
Since $XL > 5$, Go right, and since right is null, we end this.
For Right subtree (Start node = 48):
Since $XR > 48$, sum = 110 + 48 + 0 (V.left.sum) = 158
Go right
Since $XR < 59$, Go left, Since node = null, return sum
Final sum = 158

Running time analysis. The worst case time complexity is O (Log n). We first find the LCA and check where does the node splits and we begin our operation range-sum. We first traverse to the left, worst case we reach the leaf and add all the right subtree sums of the Left Path. Similarly on right subtree of our LCA node. Hence augmenting the tree allows to get the range-sum in O (log n)

(c) According to the Main Theorm: Let A be an attribute that augments an AVL tree of n nodes, and suppose that the value of A for each node v only depends on the info in the nodes v, v.left and v.right. Then we can maintain the values of A in all nodes of the tree during search, insertion and deletionn without affecting the O (log(n)) performance of the operations. Hence, Here when we augment the tree with the sum of child nodes, the performance of the other operations of the tree remains unaffected by balancing due to augmentation indicated by the theorm above.