① Input: Minheap with n distinct keys stored in an array A[1...n].
Given "x" and "k" with $1 \le k \le n$

Output: prove whether there will be k numbers in the heap which are smaller than x.

Time: O(K)

Algorithm:

```
count = 0;
minheap (root, x, k, count)
{
    if root.key < x
        count ++;
        if count >= k
            return true;
        minheap (root.left, x, k, count);
        minheap (root.right, x, k, count);
    if root.key ≥ x
        exit;
}
```

① First, set the count value to zero. Now start from the root node and check whether the root node is less than x.

② If the root node is less than the given value x. then increase the count by one and check whether the count value is greater than or equal to k. The algorithm ends if this condition satisfies.

③ If the count is less than k, then solve the problem recursively either left minheap or right min heap until the count >= k condition satisfies.

④ If the root.key is greater than or equal to x, then the algorithm ends as there is no children which is smaller than root node.

⇒ In this algorithm, the number of nodes smaller than x will be atmost k. We are using DFS to make it possible. So the time complexity is O(k).

(2) Input: Binary Search Tree T.
x value.

Output: successor of x in T

Time: O(h).

Algorithm

```
Successor (T, x) ---> successor = NULL;
{   if (root == null)
        return null;
    if root > x
        successor = root;
        if (root.left != null)
            successor (root.left, x)
        return successor;
    if root < x
        if root.right != null
            successor (root.right, x)
        return successor;
    if root = x
        return root;
}
```

// Discussed with Shivakesh Reddy & Sahiti

(1) This algorithm consists of mainly three conditions. Here, the root node is taken initially and compared directly with the given x value.

(2) Initially, the successor is taken as NULL.
Basecase: If the root node is equal to Null, then return 0 or Null.

(3) Next, if the root node is greater than the given x value, then there may be a chance that this root node might be a successor. So, this root node is assigned to the successor and the root.left (left subtree) is passed to successor() recursively. The left subtree is always checked whether it is null or not before recursion to make sure it doesn't have ~~any~~ one child. Finally, the successor is returned.

(4) If the root node is less than x, then the successor must be in the right subtree. So the right subtree is passed to successor() to find the successor recursively. The right subtree is always checked whether it is null

or not before recursion to make sure it does have one child. Finally, the successor is returned.

In this way, we can find the successor of a given

① If the root node is equal to the given x value, then return the root as successor.

In this way, we can find the successor of given element in ~~O(n)~~ O(h) time where h is the height of the tree. In each level we are comparing only once with the given x-value. So time complexity is O(h).

(3) Input: Binary Search Tree T (keys - real numbers)
Range $[x_1, x_2]$

Output: all keys sorted in T s.t $x_1 \leq x \leq x_2$.

Time: $O(h+k)$

h = height of tree T

k = number of keys of T in range $[x_1, x_2]$

Condition: All keys must be in a sorted order.

This algorithm is similar to the range-min algorithm, but the difference is that we have to keep them in a sorted. So, here we use in-order traversal to print all the keys in rang $[x_1, x_2]$.

(1) Find the lowest common ancestor of (x_1, x_2). It is the highest node in the given tree where its key $\in (x_1, x_2)$. We can find it in $O(h)$ time.

(2) Left subtree of lowest common ancestor

x_1 is a key of T, then v_1 is node of T whose key is x_1. Or let x_1 be the new leaf node created for x_1, if we want to insert x_1 into T.

Now we take the nodes from v_2 to the lowest common ancestor (lca).

* if v.key $\in$ range $[x_2, x_1]$, print it
* if v has right subtree, use in-order traversal to print all keys in right subtree of v.

③ Right subtree of lowest common ancestor

Similarly we do the same in right subtree also.

If $x_{r1} \longrightarrow$ key(T), then $v_{r1} \longrightarrow$ node of T whose key is x_{r1}.

O.t let $v_{r1} \longrightarrow$ ~~node(T)~~ new leaf node for x_{r1} if x_{r1} is inserted into T.

Now we take the nodes from v_{r1} to the lowest common ancestor.

* if v.key $\in$ range $[x_2, x_1]$, print it.
* if v has a left subtree, use in-order traversal to print all keys in left subtree of v.

Algorithm :

```
range-keys (V, x1, x2)
{
    if V = NULL return;
    if V.key > x2
        range-keys (V.left, x1, x2);
    if V.key < x1
        range-keys (V.right, x1, x2);
    else
        range-keys (V.left, x1, x2);
        print V.key;
        range-keys (V.right, x1, x2);
}
```

The time for finding lowest common ancestor is $O(h)$. The time for finding V_1 and V_2 is also $O(h)$. Time for finding nodes from V_1 to lca and lca to V_2 is $O(h)$. The inorder traversal will take $O(k)$ where k is the number of nodes in range $[x_1, x_2]$. So, the total time complexity is $O(h+k)$.

④ Input: Binary Search tree T; Range $[x_l, x_r]$
Output: Sum of keys in T, s.t $x_l \leq x \leq x_r$.
Time:
Algorithm for implementing range-sum(x_l, x_r).

```
range-sum(T, x_l, x_r)
{
    u = LowestCommonAncestor(x_l, x_r)
    // find lca as discussed in the class.
    if u = NULL return "no key in [x_l, x_r]";
  * sum = u.key
    v = u.left
    while (v ≠ NULL)
    { if v.key > x_l
            sum = sum + v.key;
            if v.right ≠ NULL
                sum = sum + v.rightsum;
            v = v.left;
      if v.key < x_l
            v = v.right;
      if v.key == x_l
            sum = sum + v.key;
            if v.right ≠ NULL
                sum = sum + v.rightsum;
            break;
    }
```

```
v = u.right
while (v ≠ NULL)
{   if  v.key < x1
        sum = sum + v.leftsum;
        sum = sum + v.key;
        if v.left ≠ NULL
            sum = sum + v.leftsum;
        v = v.right;
    if  v.key > x1
        v = v.left;
    if  v.key = = x1
        sum = sum + x.key;
        if v.left ≠ NULL
            sum = sum + x.leftsum
        break;
}
return sum;
```

① Design of the data structure

For all nodes in the binary search tree T, v.sum is assigned which is equal to the sum of its children.

① Find lowest common ancestor of (x_1, x_2). It is the highest node in the given tree where its key $\in (x_1, x_2)$. we can find it in $O(h)$ time.

② Left subtree of lowest common ancestor.

x_l is a key of T, $v_l \rightarrow$ node(T) whose key is x_l.
or, let v_l be the new leaf node created for x_l if we want to insert x_l into T.

First, sum = u.key.

Now we take the nodes from v_l to the lowest common ancestor (lca). (set sum = sum + v.rightsum)

* if v.key ∈ $[x_l, x_r]$, sum = sum + v.key.

At last, sum = sum of keys in T in range $[x_l, x_r]$.

③ Right subtree of lowest common ancestor

x_r is a key of T, $v_r \rightarrow$ node(T) whose key is x_r.
let v_r be the new leaf node created for x_r if we want to insert x_r into T.

Now we take the nodes from v_r to the lowest common ancestor. set sum = sum + v.leftsum

* if v.key ∈ $(x_l, x_r]$, sum = sum + v.key.

The time for finding lowest common ancestor is O(h).
The time for finding sum of all nodes is O(n)
Time for search, insert, delete operations is O(h).