#1 MIN-HEAP SELECTION

Given values x and k ($1 \leq k \leq n$) and an array $A[n]$, determine if x is larger than than the k^{th} smallest element of $A[n]$ in $O(k)$ time.

PSEUDO CODE:

```
boolean greaterThanSelection(int[] A, int x, int k) {
    if (A[1] > x) { return false; }

    int[] B = new int[k];
    B[1] = 0;
    int i = 1;
    int j = 2;

    while ((B[i] != 0 || i == 1) && j < k) {
        int left = (B[i]*2);
        int right = (B[i]*2)+1;

        if (A.length > left && A[left] < x) {
            B[j++] = left;
        }
        if (A.length > right && A[right] < x) {
            B[j++] = right;
        }

        i++;
    }

    return j == k;
}
```

DESCRIPTION:

This algorithm treats the min-Heap like a Binary Search Tree in order to count the number of elements smaller than x. Beginning at the root of the heap, if a value is less than x, the index of that value is stored in a secondary array. Only the children of an element stored in $B[k]$ are visited. Once there are no more elements in $A[n]$ less than x or $B[k]$ has been filled, the algorithm finishes searching. The algorithm will return true if $B[k]$ has been filled as the k^{th} smallest element must be smaller than x. If the algorithm return false, it means there were not k elements smaller than x in $A[n]$ and therefore the k^{th} smallest element is not smaller than x.

ANALYSIS:

One iteration of the algorithm takes constant time to complete, as only comparisons and constant time insertions are performed. The while-loop will only be iterated at most k times to check if there are k elements smaller than x. Therefore the total complexity of this algorithm can be defined as:

$$O(c * k) = O(k)$$

#2 BINARY TREE SUCCESSOR

Given a Binary Search Tree (BST) and a value x, determine what the in-order successor of x would be without adding it to the BST, in $O(h)$ time (where h is the height of the tree).

PSEUDO CODE:

```
private Node locateSuccessor(Node root, int x, Node successor) {
    if (root == null) { return successor; }

    if (root.key == x) {
        return root;
    } else if (x > root.key) {
        return getSuccessor(root.right, x, successor);
    } else {
        return getSuccessor(root.left, x, root);
    }
}

public Node getSuccessor(Node root, int x) {
        return locateSuccessor(root, x, null);
}
```

DESCRIPTION:

This algorithm works very similarly to a binary search using a BST. However the current successor must be passed into each recursive call. When the method is first called, the parameter *successor* would be passed a *null* value. The algorithm drops into the tree looking for the location at which x would be inserted. Anytime the algorithm goes to visit the left child, it stores the current Node as the current successor. When the algorithm reaches the bottom of the tree the variable *successor* will contain the in-order successor of x, unless there is none.

ANALYSIS:

This algorithm will always travel to the bottom of the tree as if it were going to insert x into the tree. If the algorithm happens to follow the longest branch of the tree it will take h recursions to reach the bottom and h returns to exit the recursion. Each level of the recursion performs comparisons at a constant time c. Thus the total complexity of the algorithm is:

$$O(h * c) = O(h)$$

#3 BST RANGE QUERY

Given a BST and a range x_l to x_r report, in sorted order, each element of the BST that falls within the given range in $O(h+k)$ time. Where h is the height of the tree and k is the number of elements in the BST that fall between x_l and x_r.

PSUEDO CODE:

```
void rangeQuery(Node root, int leftBound, int rightBound, int[] range) {
    if (root == null) { return; }

    if (root.key > leftBound) {
        rangeQuery(root.left, leftBound, rightBound);
    }

    if (root.key >= leftBound && root.key <= rightBound) {
        range.add(root.key);
    }

    if (root.key < rightBound) {
        rangeQuery(root.right, leftBound, rightBound);
    }
}
```

DESCRIPTION:

Similar to the last problem, this algorithm adapts the binary search method for a BST. The algorithm will traverse left into the tree to find the left bound, x_l, of the range. Once the successor of x_l is identified, each node is visited in-order until the algorithm finds the right bound, x_r, of the given range. Each key that falls within the range is stored in an array. Essentially, this algorithm is an in-order traversal of the tree that halts traversal anytime it steps out of bounds of the given range.

ANALYSIS:

The algorithm will first locate the successor of x_l, which takes at most $O(h)$ time. Once the left bound is identified each subsequent node is visited in-order and saved to an array. Adding each element to an array takes constant time, so if there are k elements between x_l and x_r this process will takes $O(k)$ time. Thus the total complexity of the algorithm is:

$$O(h * c * k) = O(h + k)$$

#4 BST RANGE SUM

Augment a Binary Search Tree such that all operations, including a range query that returns the total sum of all numbers between x_l and x_r, in $O(h)$ time. Include the range query sum algorithm.

AUGMENTED BST:

Only a simple modification to a traditional BST is needed to accomplish the above requirements. If each node stores the sum of its subtree, including itself, operations like $insert, delete$, and $search$ will not be slowed and will enable $rangeQuerySum$ to be performed in $O(h)$ time. Whenever $insert$ or $delete$ is called, the sum of each ancestor of the newly added or deleted node will be updated by adding or subtracting the value of the node as the method propagates back up the recursion.

PSUEDO CODE:

```
public int rangeQuerySum(Node root, int leftBound, int rightBound) {
    if (root == null) { return 0; }
    return root.sum − sumBeyondLeft(root, leftBound) − sumBeyondRight(root, rightBound);
}

private int sumBeyondLeft(Node root, int leftBound) {
    if (root == null) { return 0; }
    if (root.key < leftBound) {
        if (root.right == null) { return root.sum; }
        else { return root.sum − rightSum + sumBeyondLeft(root.right, leftBound); }
    } else {
        return sumBeyondLeft(root.left, leftBound);
    }
}

private int sumBeyondRight(Node root, int rightBound) {
    if (root == null) { return 0; }
    if (root.key > rightBound) {
        if (root. left == null) { return root.sum; }
        else { return root.sum − leftSum + sumBeyondRight(root.left, rightBound); }
    } else {
        return sumBeyondRight(root.right, rightBound);
    }
}
```

DESCRIPTION:

Both private methods look for the successor and predecessor of the left and right bounds. As they locate the bounds, each method tracks the sum of the nodes outside of the range by summing the subtree sums. The returned results are subtracted from the total sum of all the nodes in the tree.

ANALYSIS:

Each method will work its way to the bottom of the tree once, which takes $O(h)$ time. All other operations take constant time. Thus the total complexity for this algorithm is:

$$O(2 * c * h) = O(h)$$