DESCRIBE AN ALGORITHM THAT, GIVEN N INTEGERS IN THE RANGE 0 TO K, PREPROCESSES ITS INPUT AND THEN ANSWERS ANY QUERY ABOUT HOW MANY OF THE N INTEGERS FALL INTO RANGE $[a \ldots b]$ IN $O(1)$ TIME. YOUR ALGORITHM SHOULD USE $\Theta(n+k)$ PREPROCESSING TIME.

An algorithm as described could be implemented using an algorithm very similar to COUNTING-SORT to preprocess the values and then accessing the returned array to handle the queries.

Using this algorithm based on COUNTING-SORT to preprocess the values can be done in $\Theta(n+k)$ time:

```
Algorithm CSPREPROCESS(A, k)
In: An array of integers, A, containing the input [a ... b] and the largest element in A,
integer k.
Out: An array that contains the count of each element in array A.

let C[0…k] be a new array

for i = 0 to k do
        C[i] = 0
for j = 1 to A.length do
        C[A[j]] = C[A[j]] + 1
for i = 1 to k do
        C[i] = C[i] + C[i - 1]

return C
```

The above algorithm is essentially COUNTING-SORT, minus the final third, as it is unnecessary for this application. Without the last third, the algorithm still preforms the preprocessing in $\Theta(n+k)$, however the hidden constant is smaller: The first for loop preforms $c \cdot k$ operations to initialize the array; the second preforms $c' \cdot n$ operations to count the number of elements equal i at $C[i]$; and the final preforms $c'' \cdot k$ operations to count the number of elements less than or equal to i. Summing each operation and discarding constants leaves us with $\Theta(n+k)$.

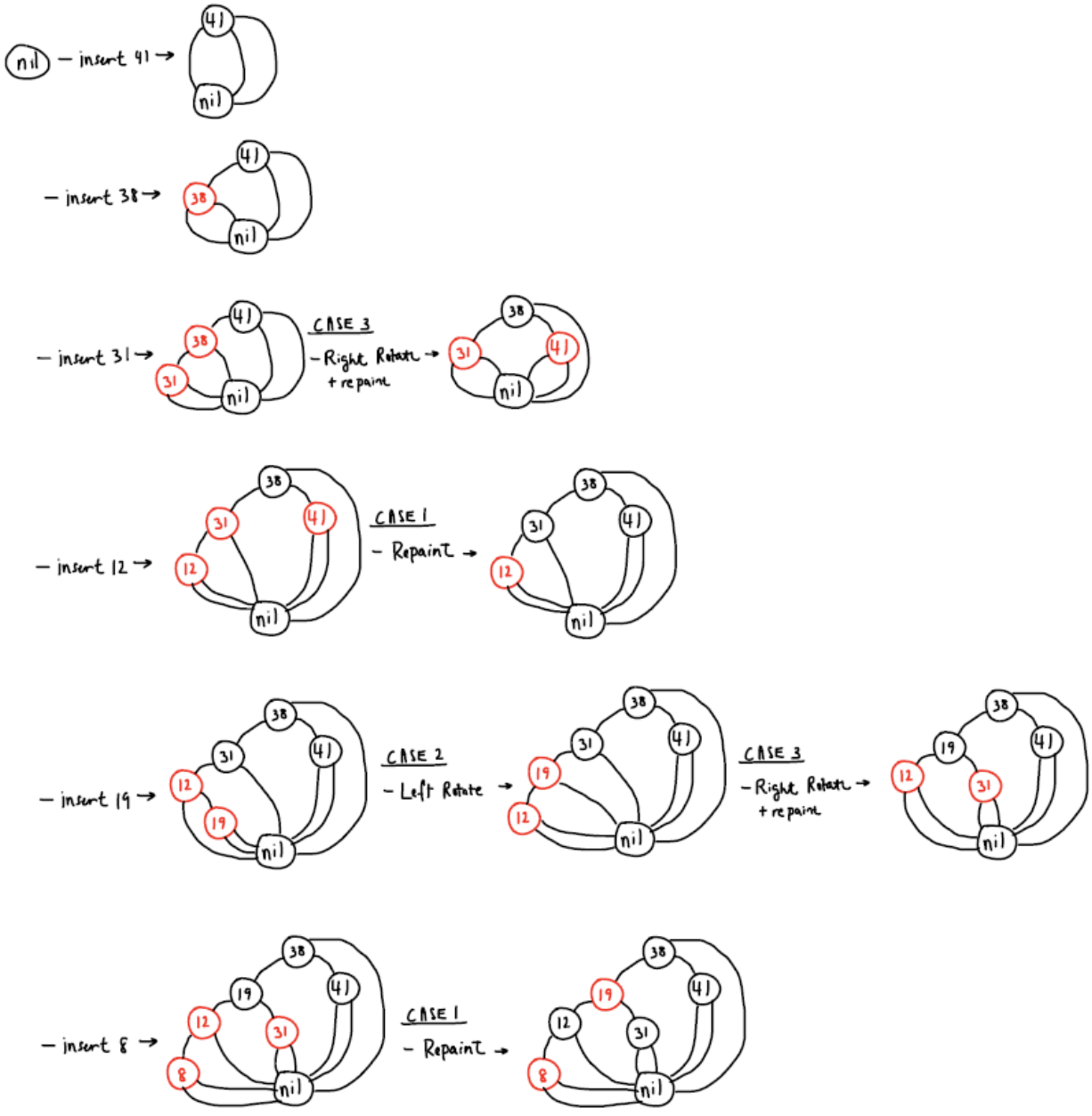
Using the array returned from CSPREPROCESS, one can devise an algorithm based on the properties of the array to return the number of elements that fit in the range $[a \ldots b]$ in constant time:

```
Algorithm QUERY(C, a, b, k)
In: An array that contains the count of each element in array A, integers a and b which
represent the boundaries for the range, the largest element in A, integer k.
Out: the number of elements in the original array that fit in the range defined by a and b.

a = a -1
if a < 0 then a = 0
if b > k then b = k

return C[b]-C[a]
```

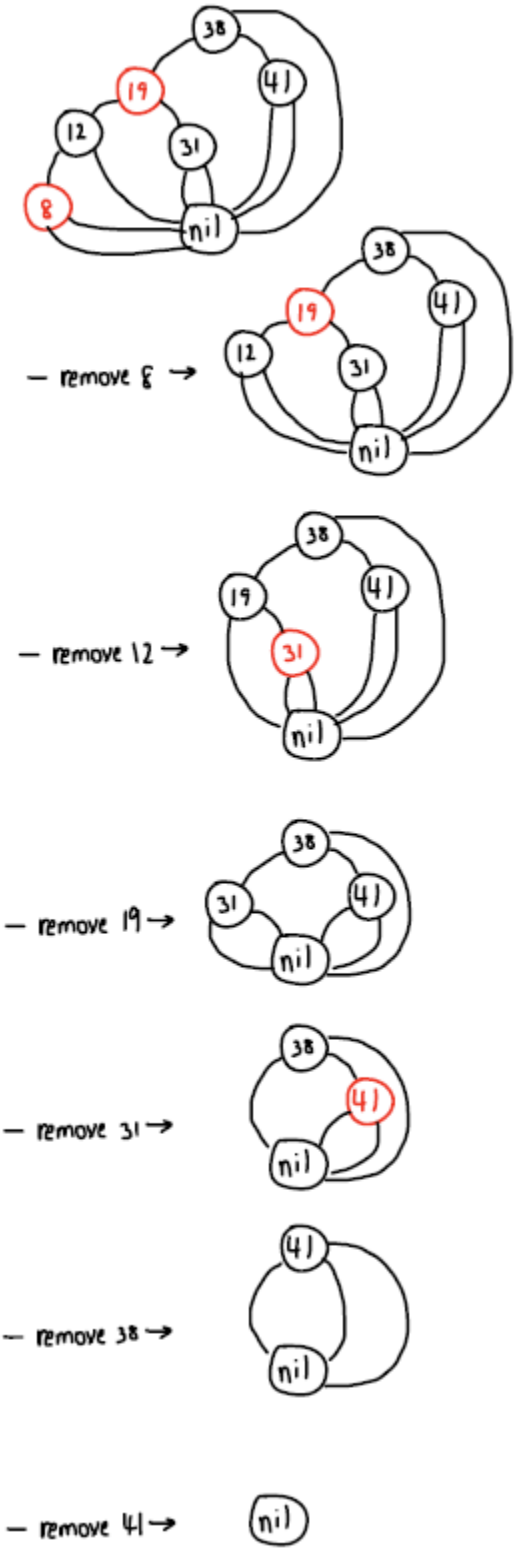
The above algorithm preforms three operations, each independent of n or k, and thus runs in $\Theta(1)$ time.

By subtracting the values less than a ($C[a]$) from the total of those less than or equal to b ($C[b]$), the elements that fit in the range $[a \ldots b]$ can be found by using the preprocessing algorithm, which runs in in $\Theta(n+k)$ time, and query algorithm, which runs in $\Theta(1)$ time

SHOW THE RED-BLACK TREES THAT RESULT AFTER SUCCESSIVELY INSERTING THE KEYS 41, 38, 31, 12, 19, 8 INTO AN INITIALLY EMPTY RED-BLACK TREE.



SHOW THE RED-BLACK TREE THAT RESULT FROM THE SUCCESSIVE DELETION OF THE KEYS IN THE ORDER 8, 12, 19, 31, 38, 41 FROM THE RED-BLACK TREE RESULTED FROM QUESTION 1.



PROVE THAT AN AVL TREE WITH N NODES HAS HEIGHT $O(\log n)$.
(HINT: PROVE THAT AN AVL TREE OF HEIGHT h HAS AT LEAST F_h NODES, WHERE F_h IS THE h^{th} FIBONACCI NUMBER.)

The number of nodes in a binary search tree (since the set of AVL trees belong to the set of binary search trees, this applies to them also) can be defined inductively as:

$$N(h) = N(h_L) + N(h_R) + 1$$

where h_L and h_R are the heights of the two subtrees of T, who's height is represented by h. Combining this definition with the fact that each subtree in an AVL tree can only differ in height by at most 1, we get:

$$N(h) = N(h-1) + N(h-2) + 1$$

as the definition for the minimum number of nodes in a AVL tree of height h.

Proof by induction

Base Case:

$$N(0) \geq F_0$$
$$\geq 0$$

This holds as a tree of height 0 has either 0 or 1 nodes.

Inductive Step:

$$F_h \leq N(h)$$

$$F_{h+1} \leq N(h+1)$$
$$N(h+1) = N(h) + N(h-1) + 1$$
$$F_{h+1} = F(h) + F(h-1)$$
$$F(h) + F(h-1) \leq N(h) + N(h-1) + 1$$

Knowing that $N(h+1) \geq N(h)$ and $F(h+1) \geq F(h)$, we can deduce, based on the above definition, that, $N(h+1) \geq F(h+1)$.

The Fibonacci sequence F_h can also be defined as $F_h = \phi^h$, where $\phi \approx 1.61803398875$. Thus, $F_h \geq 1.6^h$.

Since:

$$N(h) \geq F_h \geq 1.6^h$$
$$N(h) \geq 1.6^h$$
$$\lg N(h) \geq h$$

And since $N(h)$ is a function that represents the number of nodes:

$$N(h) = n$$
$$\lg n \geq h$$
$$\therefore h = O(\lg n) \ \blacksquare$$

DESIGN AN EFFICIENT DATA STRUCTURE USING (MODIFIED) RED-BLACK TREES THAT SUPPORTS THE FOLLOWING OPERATIONS:

INSERT(x): insert the key x into the data structure if it is not already there.
DELETE(x): delete the key x from the data structure if it is there.
FIND-SMALLEST(k): find the kth smallest key in the data structure.

WHAT ARE THE TIME COMPLEXITIES OF THESE OPERATIONS?

For this modified Red-Black tree, each node will contain an attribute storing the number of elements in its subtree (i.e. itself and all its descendants). In doing so we are able to implement FIND-SMALLEST more efficiently, while making changes so minor to INSERT and DELETE, that they will not affect their overall time complexity.

Implementing INSERT and DELETE would require a find method that would traverse the tree to find and return the node containing the element to be added or removed respectively. Since this find method does not change the structure of the tree, it is exactly like that of a binary search tree, which traverses at most he height of the tree. That being the case, the find method would be of $O(height)$; however, considering that Red-Black trees are generally well balanced, the height of the tree is $\lg n$, resulting in a time complexity of $O(\lg n)$.

As the number of descendants of any particular node will change when a new node is added, the INSERT method must be modified to adjust for this. For every node that is processed upon traversing the tree to find a place to insert the new node, its number of descendants must be increased as it will now have one more child. Since new elements can only be inserted at the leaf level, the new element will have just itself as its initial descendants. Additionally, since the INSERT method is reliant on RB-INSERT-FIXUP, and since that method alters the structure of the tree through the LEFT-ROTATE and RIGHT-ROTATE methods, these will also have to update the number of descendants of the nodes affected. Thus, upon a rotation, the parent node will swap its number of descendants with that of the child doing the rotation. These changes can all be done in $O(1)$ time, leaving each of these methods unchanged in the way of time complexity. Finally, before each insertion, a find operation will need to be conducted to make sure that the element is not already in the tree, and thus the INSERT method will be modified to include this.

Likewise, the delete algorithm will be modified in the same way as the insertion algorithm. DELETE will be modified such that each visited node during its traversal will have its number of descendants decremented by one. Also like the insertion algorithm, a find operation will need to be conducted to make sure the value exists in the tree beforehand to avoid false changes to the number of descendants of each node. Deletion also makes use of the LEFT-ROTATE and RIGHT-ROTATE methods through its RB-DELETE-FIXUP. As these methods are already modified, they need no further modification. Finally, since deletion can occur at any level in the tree, the RB-TRANSPLANT method will need to be modified so that the node replacing the deleted node inherits its number of descendants -1. As with the previous changes, all of these are done in $O(1)$ and have a minimal effect on the time complexity of the algorithm.

The final method, FIND-SMALLEST, inherits all the benefits of storing the number of children at each subtree. Considering that we are seeking the k smallest elements in the tree, we are to recursively call this method on the children of each node. To decide which node to call it on, the algorithm must compare the number of children at each subtree. Should the number of descendants be greater than k, then the k^{th} smallest value must be in the node's left subtree, so FIND-SMALLEST should be called on its left subtree. Should the number of descendants be less than k, then the problem becomes more complex: we must look to the right subtree to find the $k -$ the number of nodes in the left subtree (including its root), since each of those nodes are smaller than that those on the right, by calling FIND-SMALLEST on the right descendants. Should the number of descendants equal k, then the subtree containing the smallest k elements has been found, and will be returned. Using this algorithm to implement FIND-SMALLEST, the method can achieve a time complexity of $O(\lg n)$ as each recursive call of the method can only travel down the tree if the node is not found—resulting in a time complexity of $O(height)$, which in a balanced tree like Red-Black trees, is of $O(\lg n)$.

GIVEN N ELEMENTS AND AN INTEGER K. DESIGN AN ALGORITHM TO OUTPUT A SORTED SEQUENCE OF SMALLEST K ELEMENTS WITH TIME COMPLEXITY $O(n)$ WHEN $k \cdot \log n \leq n$. CAN YOU SOLVE THE SAME PROBLEM FOR GENERAL k WITH TIME COMPLEXITY $O(k \cdot \log(k) + n)$? (HINT: $k \cdot \log n \leq k \cdot \log(k) + n$.)

Using a min priority heap, sorting and outputting the elements can traditionally be done in $O(n \log n)$ time. Adjusting the algorithm to output only the smallest k elements could be done by limiting the number of calls to MIN-HEAPIFY to k in HEAPSORT. Traditionally, the time complexity of HEAPIFY is calculated by adding the cost of the call to BUILD-MIN-HEAP, which is done in $O(n)$ time, to the $n-1$ calls to MIN-HEAPIFY, which are done in $O(\log n)$ time each to get a total time complexity of $O(n + (n-1) \cdot \log n) = O(n \log n)$. By modifying the number of calls to MIN-HEAPIFY from $n-1$ to k. The time complexity would be calculated as $O(n + k \log n)$, and since $k \cdot \log n \leq n$, this would simplify to $O(n)$.

Such a modified algorithm would recycle the existing BUILD-MAX-HEAP AS BUILD-MIN-HEAP and use the following modified algorithms:

```
Algorithm HEAPSORT(A, k)
In: An array of integers, A and an integer k which represents the number of elements to
output.
Out: A sorted sequence of the smallest k elements in A.

BUILD-MIN-HEAP
for i = k downto 1 do
        exchange A[1] with A[i]
        A.heap-size = A.heap-size - 1
        MIN-HEAPIFY
return A
```

```
Algorithm MIN-HEAPIFY(A, i)
In: An array of integers, A and an index in the array, i.
Out: A min-priority heap where the subtree rooted at i obeys the min-heap property

left = Left(i)
right = Right(i)
if left ≤ A.heap-size and A[left] < A[i]
        smallest = left
else smallest = i
if right ≤ A.heap-size and A[right] < A[i]
        smallest = right
if smallest ≠ i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```

Knowing that the above algorithm creates the min heap in $O(n)$ and returns the smallest k elements in $O(k \cdot \log n)$ time, the total time complexity is of $O(k \cdot \log n + n) = O(k \cdot \log n)$. Considering that $k \cdot \log n \leq k \cdot \log(k) + n$, $k \cdot \log n$ is upper bounded by $k \cdot \log(k) + n$, and is thus $O(k \cdot \log k + n)$.

PROVE THAT IF WE ORDER THE CHARACTERS IN AN ALPHABET SO THAT THEIR FREQUENCIES ARE MONOTONICALLY DECREASING, THEN THERE EXISTS AN OPTIMAL CODE WHOSE CODEWORD LENGTHS ARE MONOTONICALLY INCREASING.

Assuming we are using Huffman's algorithm for encoding, a set of characters whose frequencies are monotonically decreasing would be: $C\{a: 26, b: 25, c: 24, \dots, x: 3, y: 2, z: 1\}$. Such a set would be stored in a min-priority queue, Q like so: $\langle z: 1, y: 2, x: 3, \dots c: 24, b: 25, a: 26\rangle$.

Huffman's algorithm processes Q by first creating a new node and assigning its left and right child to the first and second elements in the queue. It then sums their frequencies and adds the new node into its appropriate place in Q based on its frequency, preserving the min-priority property of Q. This process repeats itself $|C|$ times, which is when there is one element in the queue which remains—the root of the Huffman Tree.

With each iteration, the number of edges in the tree increases proportionately to the number of elements processed (ie. two edges per iteration). Thus, the number of edges that exist in the tree can be modelled by:

$$|E| = 2 \cdot (|C| - 1)$$

Since each edge is represented by a bit (ether 1 or 0), each new edgepath must be represented by a new, unique binary string. Thus, the number of unique binary strings is directly proportional to the number of edges. This being the case, the number of unique binary strings $|S|$, or "codewords," can therefore be defined using the same function as that for the number of edges:

$$\begin{aligned} |S| &= |E| \\ &= 2 \cdot (|C| - 1) \\ \frac{d|S|}{d|C|} &= 2 \end{aligned}$$

Thus, this function is monotonically increasing. However, regarding the length of the codewords, there are two ways of creating a new unique codeword: either toggle any of the digits so that it produces a unique codeword or add a new bit. Since the length of the codewords must either remain the same or increase each time a new edge is added, the lengths will never decrease and, will increase each time enough new edges are added. This abides by the definition of a monotonically increasing function, whereby, for all defined values of x and y:

$$y \leq x, \qquad f(y) \leq f(x)$$

Thus, the codeword lengths must be monotonically increasing. ■

PROVE THAT EVERY NODE HAS RANK AT MOST $\lfloor \lg n \rfloor$.

A node of rank r must be the root of a subtree which has a size of at least 2^r. Let $S(N_r)$ represent the size of a node N of rank r.

Proof by induction

Base Case:

$$2^0 = 1$$

This holds as the base case represents the first operation, Make_set, which makes a set of one node, containing itself.

Inductive Case:

$$S(N_r) \leq 2^r$$

Since the only way for N to attain the rank of $r + 1$ would be for it to become the root of a union of two trees of rank r:

$$S(N_{r+1}) \leq 2^r + 2^r$$
$$S(N_{r+1}) \leq 2 \cdot 2^r$$
$$S(N_{r+1}) \leq 2^{r+1}$$

Given that we have n nodes, with 2^r nodes in each tree of rank r:

$$2^r \leq n$$
$$r \leq \lg n$$
$$\therefore r = O(\lg n)$$
$$\because r \in \mathbb{Z}^+, r \leq \lfloor \lg n \rfloor \quad \blacksquare$$

IN LIGHT OF THE PAST EXERCISE, HOW MANY BITS ARE NECESSARY TO STORE X.RANK FOR EACH NODE X?

As each x.rank is the upper bound of the height of x, we would need at most $\lfloor \lg n \rfloor$ bits to store each x.rank, as $\lfloor \lg n \rfloor$ is the upper bound on the rank of each x (as proven above). Therefore, $n \cdot \lfloor \lg n \rfloor$ bits would be the most bits necessary to store x.rank for every node x, thus the number of bits necessary to store x.rank for every node x is of $O(n \cdot \lg n)$.