# CSC265 Fall 2020 Homework Assignment 3

Student 1: Kunal Chawla
Student 2: Andrew Feng

1. Construct a data structure based on a LOBS that represents a sequence of positive integers $S$ of length $n$ and supports the following operations in $O(\log n)$ time:

   SUBSEQMAX$(S, i, j)$: returns the maximum value among the $i$'th through $j$'th elements of the sequence $S$, where $1 \leq i \leq j \leq n$.

   APPEND$(S, v)$: appends the integer $v$ to the end of the sequence $S$.

   INSERT$(S, i, v)$: inserts the integer $v$ immediately before the $i$'th element in the sequence $S$, where $1 \leq i \leq n$.

   For example, if $S = 12, 3, 8, 8$, then SUBSEQMAX$(S, 2, 3)$ returns 8 and INSERT$(S, 2, 2)$ changes $S$ to $12, 2, 3, 8, 8$.

   (a) Clearly explain how your data structure represents a sequence of positive integers.
   **Solution:**
   To represent a sequence of positive integers, we use a LOBS and store all positive integers in the sequence as values of the leafs. In particular, we use a LOBS where every leaf has a distinct key, and if the keys of the leafs are in the order

   $$k_1 < k_2 < ... < k_n,$$

   then the leaf with key $k_i$ stores the $i$ th element in the sequence.
   To assist with the methods of the data structure, we augment a LOBS. At each node $x$ in the LOBS, we store

   i. the number of leafs in the left subtree of $x$, called left-leafs$[x]$,
   ii. the number of leafs in the right subtree of $x$, called right-leafs$[x]$,
   iii. the maximum value store at any leaf in the left subtree of $x$, called left-max$[x]$,
   iv. the maximum value stored at any leaf in the right subtree of $x$, called right-max$[x]$.

   Observe that we can calculate these attributes for any node $x$ using only $left[x]$ and $right[x]$. Indeed, we have

   - left-leafs$[x]$ = left-leafs$[left[x]$ + right-leafs$[left[x]]$,
   - right-leafs$[x]$ = right-leafs$[right[x]]$ + left-leafs$[right[x]$,
   - left-max$[x] = \max($left-max$[left[x]]$, right-max$[left[x]]])$,
   - right-max$[x] = \max($left-max$[right[x]]$, right-max$[right[x]])$.

   So by theorem 14.1 in the book storing these do not change the asymptotic running time of insertion in a LOBS.

(b) Give an algorithm (in pseudocode) for performing SUBSEQMAX$(S, i, j)$ in this data structure. Briefly explain how your algorithm works. Give the high level idea, NOT a line by line description of the pseudocode.

**Solution:**

The idea is to think of bounding all nodes containing elements of the sequence from $i$ to $j$ like an interval. We first find the left endpoint of the interval - a node $x$ in the LOBS such that the rightmost node in the left subtree of $x$ contains the $i$th element of our sequence. Then knowing where this node is, we traverse through the tree until we get to the right endpoint of the interval, taking the maximum of all leafs in between along the way, in such a way that takes $\mathcal{O}(h) \cong \mathcal{O}(\log n)$.

First we'd like to define a term we use throughout. Recall that in a LOBS every non-leaf has a pointer to the rightmost node in its left subtree, this was not given a name in the previous assignment so we use it here.

Let $x$ and $y$ be nodes in a LOBS. We call $y$ a knockoff-predecessor of $x$ if either (i) $x$ is a leaf and $y = x$ or (ii) $x$ is a non-leaf and $y$ is the rightmost node in the left subtree of $x$.

We write the knockoff-predecessor of $x$ as 'knockoff-predecessor[x]'. Observe that this must be a leaf in the tree. If $x$ is a leaf then this is clear, and if $x$ is not a leaf then the rightmost node in the left subtree of $x$ must be a leaf - else by the LOBS property it would have a right child.

Now we get on to outlining our SUBSEQMAX$(S, i, j)$ algorithm:

First we find a node $x$ in the LOBS $S$ such that knockoff-predecessor$[x]$ contains the $i$th element in our sequence.

```
FIND-BOTTOM-ENDPOINT($S, i$)
    $x$ = root[$T$ ]
    while left-leafs[$x$] $\neq i$
        if left-leafs[$x$] $< i$
            $i = i -$ left-leafs[x]
            $x = right[x]$
        else
            $x = left[x]$
    return $x$
```

Next we need to calculate the maximum of the values stored in the leafs between leaf $i$ and leaf $j$. To do this we go as follows:

- Start at a node whose knockoff predecessor is the $i$th leaf and store this value as MAX-CANDIDATE
- Traverse up through the tree, and if the right subtree of our node $x$ contains only leafs between $i$ and $j$, we compare MAX-CANDIDATE to the right-max of our node. We keep traversing up until it is possible to reach leaf $j$ only by traversing down.
- We traverse down to leaf $j$, and if the left subtree of our node $x$ contains only leafs between $i$ and $j$, we compare MAX-CANDIDATE to the left-max of our node.

- we return MAX-CANDIDATE

Broadly, this takes $\mathcal{O}(h)$ steps as we go straight up the tree, and then straight down. Overall, our pseudocode is as follows.

hyperref

```
SUBSEQMAX($S, i, j$)
        bottom-endpoint = FIND-BOTTOM-ENDPOINT($S, i$)

        MAX-CANDIDATE = knockoff-predecessor[x].value
        just-moved-right = TRUE
        while $i +$ right-leafs$[x] < j$
            if just-moved-right
                MAX-CANDIDATE = max{MAX-CANDIDATE, right-max[x]}
                just-moved-right = FALSE
            if $x = right[parent[x]]$
                i = i - left-leafs[x]
            if $x = left[parent[x]]$
                just-moved-right = TRUE
                i = i + right-leafs[x]
            $x = parent[x]$

        if $i +$ right-leafs$[x] = j$
            MAX-CANDIDATE = max{MAX-CANDIDATE, right-max[x]}
            return MAX-CANDIDATE

        x = $right[x]$
        $i = i +$ left-leafs$[x]$

        while $i \neq j$
            if $i < j$
                MAX-CANDIDATE = max{MAX-CANDIDATE, left-max[x]}
                $x = right[x]$
                $i = i +$ left-leafs$[x]$
            else
                $x = left[x]$
                $i = i -$ right-leafs$[x]$
        MAX-CANDIDATE = max{MAX-CANDIDATE, left-max[x]}
        return MAX-CANDIDATE
```

(c) Prove that your algorithm for SUBSEQMAX is correct.
**Solution:**

First we prove correctness of FIND-BOTTOM-ENDPOINT$(S, i)$ As in 240, to denote the value of a variable during the $k$th iteration of the loop we add a subscript $k$ (e.g. $i_k$)

**Lemma 1** *Given a LOBS as specified in part (a) that contains* $n$ *leafs, and given some integer* $i$ *such that* $1 \leq i \leq n$*, BOTTOM-ENDPOINT$(S, i)$ outputs a pointer to a node* $x$ *such that knockoff-predecessor[x] contains the* $i$*th element in the sequence.*

**Proof:** Let $x'$ be the value of $x$ at the beginning of the while loop. We claim that if $i$ is an integer between 1 and $n'$, where $n'$ is the number of leafs in the subtree under $x'$, then the while loop terminates and $x$ points to a node such that knockoff-predecessor$[x']$ contains the $i$th element in our sequence.

We prove this claim by strong induction on $h$, the height of the tree under $x'$. Let $h$ be an arbitrary nonnegative integer and suppose that the claim is true for all $k < h$. Then on the first iteration of the while loop, there are three possibilities, depending on whether $i$ is less than, equal to, or greater than left-leafs$[x']$

If $i <$ left-leafs$[x']$, then $x$ must be a non-leaf, therefore it has both a right and left child. Then the $i$th leaf is contained in the right subtree of $x'$. Also, lines 4–6 will execute. Observe that the $i$th leaf in the subtree under $x'$ is also the $(i -$left-leafs$[x])$th leaf in the subtree under $right[x']$. When we run lines 5 and 6 this will update $i$ and $x$ accordingly, then run the while loop again. At this iteration of the while loop, the subtree under $x$ has height $h - 1$, so by our induction hypothesis this will output a pointer $x$ such that the rightmost node in the left subtree under $x$ is leaf $i$.

In the second case, if left-leafs$[x] = i$, then the while loop does not run. Observe that if left-leafs$[x] = i$, then the rightmost node in the left subtree of $x$ is the $i$th leaf in our tree.

In the third case, $i >$ left-leafs$[x']$, then $x$ must be a non-leaf, therefore it has both a right and left child. Then the $i$th leaf is contained in the left subtree of $x'$. Also, line 8 will execute. Observe that the $i$th leaf in the subtree under $x'$ is also the $i$th leaf in the subtree under $left[x']$. When we run line 8 this will update $x'$ to $left[x']$. At this iteration of the while loop, the subtree under $x'$ has height $h - 1$, so by our induction hypothesis this will output a pointer $x$ such that the rightmost node in the left subtree under $x$ is leaf $i$.

Therefore we have shown that after the while loop runs, the variable $x$ is a pointer to some node in the tree whose rightmost leaf in left subtree is the $i$th leaf in our tree. ■

Now we prove correctness of SUBSEQMAX$(S, i, j)$. To do this we will need two loop invariants.

**Lemma 2** *After the kth iteration of the first while loop, let $a_k, b_k$ be the positions of the leftmost and rightmost leafs in the right subtree of $x_k$ (i.e. which elements of our sequence these leafs hold).*

*Then for every nonnegative integer k, after the kth iteration of the while loop in lines 15–24, the following hold*

- *quantities $a_k$ and $b_k$ increase only if the boolean just-moved-right is TRUE,*
- *when $a_k$ or $b_k$ increase we have $a_k = b_{k-1} + 1$,*
- *the value of $i_k$ is precisely the number of leafs in the tree to the left of $x_k$,*
- *if $j$ is a leaf in the right subtree of $x_k$, then $max \geq value[j]$.*

**Proof:** We prove this by strong induction on $k$. Let $k$ be an arbitrary nonnegative integer and suppose the result holds for all $j < k$. Then suppose the loop runs again. Then we know that $i + \text{right-leafs}[x] = b_{k-1}$ is less than $j$, and by our induction hypothesis we know that $i_{k-1}$ is equal to the number of leafs in our tree to the left of $x$.

If just-moved-right is set to FALSE, we know that $x_{k-1}$ was the right child of $x_k$, therefore the node $x_{k-1}$ was in the right subtree of $x_k$. This tells us that $b_k = b_{k-1}$. Likewise, it tells us that $a_k \leq a_{k-1}$, which proves the first part of our statement.

If just-moved-true is set to TRUE, we know that either (i) the loop is running for the first time, or that $x_{k-1}$ was the left child of $x_k$. In the former case, we know that $a_k \geq a_{k-1}$

In either case, if $p$ is a leaf in the right subtree of $x_k$, then either $p$ was in the right subtree of $x_{k-1}$ or not. In the former case, we know that $\text{MAX-CANDIDATE}_{k-1} \geq value[jp$. In the latter case, this means that just-moved-right is true, therefore on line 17 runs. When this line runs it tells us that $\text{MAX-CANDIDATE}_k \geq \text{MAX-CANDIDATE}_{k-1}$ and $\text{MAX-CANDIDATE}_k \geq \text{right-max}[x_k] \geq value[p]$ because $p$ is in the right subtree of $x_k$.

Now we only need to show that the value of $i_k$ is precisely the number of leafs to the left of $x_k$, inclusive. As this is true on the first iteration of the loop by correctness of FIND-BOTTOM-ENDPOINT, we can assume that $k > 0$. By our induction hypothesis we know that $i_{k-1}$ is the number of leafs on the left of $x_{k-1}$. On each iteration of the loop, we move $x$ to its parent – observe that it must in fact have a parent else $i_{k-1} + \text{right-leafs}[x]$ would be greater than $j$.

There are now two cases to consider. If $x_{k-1}$ is the right child of its parent, then $parent[x_{k-1}]$ has $i_{k-1} - \text{left-leafs}[x]$ leafs on its left. If instead $x_{k-1}$ is the left child of its parent, then $parent[x_{k-1}]$ has $i_{k-1} + right - leafs[x]$ leafs to its left.

Therefore by induction our loop invariant holds after all iterations of the while loop in line 15–24.

■

Also, the quantity $i_k$ is precisely the number of leafs that are to the left of $x_k$.

**Lemma 3** *Similar to before, let $a_k$ and $b_k$ denote the position of the leftmost and rightmost leafs in the left subtree of $x_k$. Then after the kth iteration of the while loop in lines 33–40, the following hold:*

- *The quantity $a_k$ is greater than or equal to $i_0$.*
- *if line 35 is executed, then every leaf in the left subtree of $x_k$ is between the $i$th and $j$th leafs in the tree.*
- *The quantity $i_k$ is precisely the number of leafs to the left of $x_k$.*

**Proof:** We prove this by strong induction on the number of iterations of the loop, $k$. If $k = 0$ then the invariant holds. Indeed, by lines 30–31 we know that the left subtree of $x_0$ contains only leafs from $i$ to $j$, therefore the first part of our invariant hold. Also, by our loop invariant for the first while loop and line 31, we know that when the while loop runs for the first time that $i_0$ is precisely the number of leafs to the left of $x_0$. Also by our loop invariant for the previous while loop, we know that every leaf in the left subtree of $x_0$ before the loop starts is less than or equal to MAX-CANDIDATE$_0$.

Now suppose that $k > 0$ and the loop invariant holds for all nonnegative integers $l < k$. Then either $i_k < j$ or $i_k > j$.

In the first case we know by our induction hypothesis that $i_{k-1}$ is the number of nodes in the left subtree of $x_{k-1}$, therefore every leaf in the left subtree of $x_{k-1}$ is between the $i$th and $j$th leafs. As line 35 is executed this tells us that for every leaf $p$ between $a_k$ and $b_k$, that $value[p] \leq$ MAX=CANDIDATE, Also, in line 36 we update $x$ to the right child of $x$, and update $i_k$ to $i_{k-1} +$left-leafs$[x_{k-1}]$. As $a_k$ is at least as large as $a_{k-1}$, this maintains our loop invariants.

In the second case, we know by induction hypothesis that $i_{k-1}$ is precisely the number of leafs in the tree to the left of $x_{k-1}$, therefore when we update $x$ to $left[x]$, the number of leafs to the left of $left[x]$ is equal to $i_{k-1} -$ right-leafs$[x_{k-1}]$. Also, $a_k = a_{k-1}$. Therefore we have maintained our loop invariant. ■

Now we are ready to prove correctness of SUBSEQMAX$(S, i, j)$

**Theorem 1** *SUBSEQMAX$(S, i, j)$ returns the maximum value among the $i$th through $j$th elements of our sequence.*

**Proof:** By our two loop invariants, we know that for every leaf $p$ in the tree such that $p$ is between the $i$th and $j$th leafs, that $p$ is less than or equal to MAX-CANDIDATE. Therefore MAX-CANDIDATE is greater than or equal to the maximum value of the nodes between the $i$th and $j$th leafs in the tree.

Also, as the only updates to MAX-CANDIDATE were

- setting it equal to the value in the $i$th leaf
- taking the max of MAX-CANDIDATE and right-max$[x]$ in the first while loop
- taking the max of MAX-CANDIDATE and left-max$[x]$ in the second while loop.

Again by both our lemmas about the loops, we know that at each time either right-max (resp. left-max) was called, either the entire right (resp. left) subtree of $x$ contained only leafs from $i$ to $j$. Therefore MAX-CANDIDATE is less than or equal to the maximum of the values in the leafs between the $i$th and $j$th leaf in the tree. Therefore MAX-CANDIDATE is equal to the maximum value among the leafs between the $i$th and $j$th leaf in the tree.

Therefore, when we return MAX-CANDIDATE on line 42, we know that we are returning the maximum value among the $i$th to $j$th elements of our sequence, inclusive. ■

(d) Prove that the worst case time complexity of your algorithm is $\Theta(\log n)$.

**Solution:**

We break our argument into a couple lemmas, working with each part of our algorithm for SUBSEQMAX$(S, i, j)$

Let $d : \text{Node} \times \text{Node} \to \mathbb{N}$ be the distance function on trees giving the length of the shortest path between two nodes. Let **root** denote the root node of the LOBS.

**Lemma 4** *FIND-BOTTOM-ENDPOINT runs in $\Theta(h)$ where $h$ is the height of the tree.*

**Proof:** i. **FIND-BOTTOM-ENDPOINT runs in $O(h)$**: Notice that every line other than the while loop on lines - executes in constant time and the lines within the loop takes constant number of steps per iteration. Thus, it is enough to show that the while loop runs for at most $h$ iterations. Let $x'$ be the value of $x$ on line . We show that number of iteration of the loop is at most $d(\mathbf{root}, x')$ by induction on the number of iterations.

Let $P(k)$ mean: if the while loop on lines - runs for at least $k$ iterations, then $k = d(\mathbf{root}, x_k)$[1]. Note that $P(0)$ holds since distance is 0 before the loop. Now suppose that $P(k')$ holds for $k' = k - 1, k > 0$ and that the loop runs for at least $k$ iterations. By induction hypothesis, we know that $k - 1 = d(\mathbf{root}, x_{k-1})$. By lines and , we know the new distance is 1 greater than last iteration. Thus, $k = d(\mathbf{root}, x_k)$, we can conclude $P(k)$.

Suppose that the while loop runs for $a$ iterations. By above result we get $a = d(\mathbf{root}, x_a) \le h$.

ii. **FIND-BOTTOM-ENDPOINT runs in $\Omega(h)$**: Let $i = n$ and let $x$ denote the value of the variable $x$ right before termination. We will prove that FIND-BOTTOM-ENDPOINT takes at least $h$ steps to complete. We know that the $n$th leaf must be the right most leaf in the right subtree of our LOBS since its parent must contain two children and the $n$th leaf is the last leaf. This means there does not exist a node where the $n$th leaf is the predecessor, so $x$ must be the $n$th leaf itself (by the correctness algorithm, FIND-BOTTOM-ENDPOINT's return value is either: a node whose predecessor is the $n$th leaf, or the $n$th leaf itself). Now, by the property of LOBS, we know that $d(\mathbf{root}, x) = h$, and by the result from above we can conclude the loop runs for $h$ iterations.

Combining the two results, we can conclude FIND-BOTTOM-ENDPOINT runs in $\Theta(h)$ where $h$ is the height of the tree. ■

**Lemma 5** *The height of the tree is $\Theta(\log n)$, where $n$ is the number of elements in our sequence.*

[1] $x_k$ means the value of variable $x$ during the $k$th iteration

**Proof:** From last assignment, we already know that the height of a LOBS is $O(\log n)$, where $n$ is the number of leaves. We now show the height of a LOBS is $\Omega(\log n)$. Suppose a LOBS $T$ has $n$ leaves.

We know a full binary tree with height $h$ has exactly $1 + \ldots + 2^h = 2^{h+1} - 1 = n'$ nodes. Solving for $h$ gives $h = \log_2(n' + 1) - 1$. Thus, a full binary tree with $n$ nodes has height $\lceil \log_2(n + 1) - 1 \rceil$. This allows us to conclude that the height of $T$ is at least $\lceil \log_2(n + 1) - 1 \rceil \in \Omega(\log n)$. From here, we can conclude that a LOBS with $n$ leaves has height at least $\lceil \log_2(n + 1) - 1 \rceil \in \Omega(\log n)$. ■

**Theorem 2** *The worst case time complexity of SUBSEQMAX$(S, i, j)$ is* $\Theta(\log n)$

**Proof:** We first show that the worst case time complexity is $\Theta(h)$, where $h$ is the height of our LOBS.

Note that by lemma 4, we know line 11 runs in $\Theta(h)$ time. This shows that SUBSEQ-MAX runs in $\Omega(h)$, and it remains to show SUBSEQMAX runs in $O(h)$.

First notice that the code outside of the two while loops (lines 15 to 24 for the first one; lines 33 to 41 for the second one) run in constant time. The same is true for the code inside of the loops. It is enough that we show the loops each runs for at most $h$ iterations.

i. **The while loop on lines 15 to 24 runs for at most $h$ iterations**: Let $x_i$ be the value of variable $x$ before the while loop and $x_f$ be the value of variable $x$ immediately after the while loop. We claim that the number of iterations of the loop (call it $t$) is at most $d(x_i, x_f)$. This can be proven by induction as follows.
   Let $x_j$ denote the value of variable $x$ during the $j$th iteration (before line 24). We know that right before the loop executes, $x_i = x_0$. So the distance between the two is 0 as expected. Suppose (as induction hypothesis) that if the loop runs for at least $k - 1$ iterations, then $d(x_i, x_{k-1}) = k - 1$ and [$x_i$ is a descendent of $x_{k-1}$ or $x_i = x_{k-1}$]. Now, if the loop runs for at least $k$ iterations, we can use the hypothesis to conclude that $d(x_i, x_{k-1}) = k - 1$ and [$x_i$ is a descendent of $x_{k-1}$ or $x_i = x_{k-1}$]. By line 1b and the fact that [$x_i$ is a descendent of $x_{k-1}$ or $x_i = x_{k-1}$], the distance must increase by 1. So we know that $d(x_i, x_k) = k$. And since $x_k$ is the parent of $x_{k-1}$ and [$x_i$ is a descendent of $x_{k-1}$ or $x_i = x_{k-1}$], we know that $x_i$ is a descendent of $x_k$. This completes the induction.
   Now, by the correctness proof, we know the loop terminates after some $t$ iterations. By result from the above paragraph, we know $t = d(x_i, x_t)$. And from line 24, we have $x_f$ is the parent of $x_t$. Combine this with $x_t$ being the ancestor of $x_i$, we can say $x_f$ is an ancestor of $x_i$. Since this is a binary tree, $d(x_i, x_f) \leq h$. So,

$$t = d(x_i, x_t) \leq d(x_i, x_f) \leq h.$$

ii. **The while loop on lines 33 to 41 runs for at most $h$ iterations**: If the number iterations this while loop runs is $t$, we show that $t \leq d(x_i, x_f)$, where $x_i$ and $x_f$ are the values of variable $x$ immediately before and after the loop, respectively.
   We do this by induction (similar to the method above). Note that before the loop begins, we know $d(x_0, x_i) = 0$, which is what we expected. Suppose that if the loop

runs for at least $k-1$ iterations, then $d(x_i, x_{k-1}) = k-1$ and [$x_{k-1}$ is a descendant of $x_i$ or $x_{k-1} = x_i$]. Suppose now that the loop runs for at least $k$ iterations. We can use the induction hypothesis that $d(x_i, x_{k-1}) = k-1$ and [$x_{k-1}$ is a descendant of $x_i$ or $x_{k-1} = x_i$]. By lines 36 and 39, we know that $x_k$ is a child of $x_{k-1}$. Combine this with the fact that either $x_{k-1}$ is a descendant of $x_i$ or $x_{k-1} = x_i$, we know $x_k$ is a descendant of $x_i$ and $d(x_i, x_k) = k$. This completes our induction.
By the correctness proof, we know that this loop terminates after some $t$ iterations. We know by lines 36 and 39 that $x_f$ is a child of $x_t$, so we have

$$t = d(x_i, x_t) < d(x_i, x_f) \leq h,$$

where the last inequality comes from the fact that ancestor-to-descendant paths in a tree does not exceed the height of the tree.

■

(e) Clearly explain in English how to perform APPEND and INSERT in $O(\log n)$ time. Do not use pseudocode. Briefly explain why your data structure is correct and satisfies the required time complexity bound. You may use algorithms and results in the solutions to Homework Assignment 2.

**Solution:**
**INSERT:**

To insert a new element into our sequence we do the following:

i. start at the root of the tree
ii. traverse all the way down to the right until we get to our rightmost leaf $x$
iii. insert a new leaf containing new value $v$, with key equal to key$[x] + 1$, using LOBS-INSERT from assignment 2.

As we specified in part (a), in our instance of a LOBS we mandate that no two leafs have an identical key. This ensures that the integer $v$ is inserted at the end of our sequence, and that any rotations that go on during LOBS-INSERT do not affect the order of the leafs in our sequence.

As we specified in part (a), the augmentations we added to our LOBS do not affect the asymptotic running time of LOBS-INSERT. The first two steps in this algorithm take $\mathcal{O}(h)$ where $h$ is the height of the tree. As we showed, this is $\mathcal{O}(m)$ where $m$ is the number of nodes in our tree. Also by a lemma in part (d), we know that if the number of leafs in our lobs is $n$, then $m \in \Theta(n)$, therefore we know that the first two steps take $\mathcal{O}(\log n)$ time complexity in the worst case.

The third step takes $\mathcal{O}(\log m)$ in the number of nodes $m$ in our LOBS by assignment two, and the justification we gave above. Again by the lemma in section (d), we know that $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ where $n$ is the number of elements in our sequence.

Therefore INSERT$(S, v)$ is correct and takes $\mathcal{O}(\log n)$ time in the worst case.

**APPEND:**

To do APPEND$(S, i, v)$, we do the following:

i. If $i = 1$, start at the root of the tree and traverse all the way down until arriving at the leftmost leaf in the tree, denoted by $x$

ii. Else, we know that $i > 1$, so use FIND-BOTTOM-ENDPOINT$(S, i-1)$ to find a node $x$ whose knockoff-predecessor is either the first leaf, or the $(i-1)$th leaf in our sequence.

iii. Insert a new leaf into our tree, containing the value $v$, and whose key is
   - $\frac{key[x]+key[x']-1}{2}$, if $i = 1$,
   - $\frac{key[x]+key[x']}{2}$, if $i > 1$ and $x$ is not a leaf.
   - $\frac{key[x]+key[x']}{2} + \varepsilon$, if $i > 1$ and and $x$ is a leaf, where $\varepsilon$ is strictly less than $key[x] - key[successor[x]]$. Here $x'$ is the $i$th leaf in our tree. If $x'$ does not have a successor we set $\varepsilon = 1/2$

     In this third case we calculate the successor of $x$ using the algorithm TREE-SUCESSOR on page 259 of CLRS, which given a node $x$ outputs a pointer to its successor in $\mathcal{O}(h)$ time where $h$ is the height of the tree.

Now we justify why this is correct.

If $i = 1$, then this inserts $v$ using a node whose key is $\frac{2key[x']-1}{2} < key[x']$, where $x'$ is the leftmost leaf in the tree. Therefore this inputs $v$ directly before the 1st element as specified. As the keys of all leafs are distinct, this maintains all properties of our data structure.

Else, if $i > i$, then the call to FIND-BOTTOM-ENDPOINT$(S, i-1)$ outputs a node $x$ whose knockoff-predecessor is the $(i-1)$th leaf in our tree, which we denote by $x'$. From this fact, either $x$ and $x'$ are the same key or $x$ is the successor of the $(i-1)$th leaf. In either case the key we use while inserting $v$ is strictly between the key of $x'$ and the successor of $x'$. Therefore the node containing the value $v$ is inserted directly after the $(i-1)$th leaf and before the $i$th leaf. As this key is distinct from that of all other leafs in the tree we maintain the property that all leafs have distinct keys. Therefore the algorithm is correct.

Now we justify why it runs in $\mathcal{O}(\log n)$ time, where $n$ is the number of elements in our sequence. If $i = 1$ we traverse all the way down which takes $\mathcal{O}(h)$ steps. Else, we make a call to FIND-BOTTOM-ENDPOINT which also takes $\mathcal{O}(h)$ time in the worst case.

Then on the next line, we calculate the new key that is used to insert our new leaf. In the first two cases, this takes constant time. However in the third case we make a call to TREE-SUCCESSOR from page 259 of CLRS. This takes $\mathcal{O}(h)$ time in the worst case according to CLRS.

Then we run LOBS-INSERT, which by the previous assignment also takes $\mathcal{O}(h)$ time.

Therefore our total algorithm takes $\mathcal{O}(h)$ time in the worst case. However, by a lemma we proved in part (d), the height of a LOBS is asymptotically equal to $\log n$, where $n$ is the number of leafs. Therefore APPEND$(S, i, v)$ takes $\mathcal{O}(\log n)$ time in the worst case.

2. Consider the abstract datatype CACHE. An object of this abstract data type is a subset $C$ of $\{1, \ldots, m\}$ of size at most $k$.

   Initially, $C = \emptyset$. The only operation is ACCESS$(p)$, which adds the number $p \in \{1, \ldots, m\}$

to the set $C$ and, if this causes the size of $C$ to become bigger than $k$, removes the element from $C$ that was least recently the parameter of an access operation.

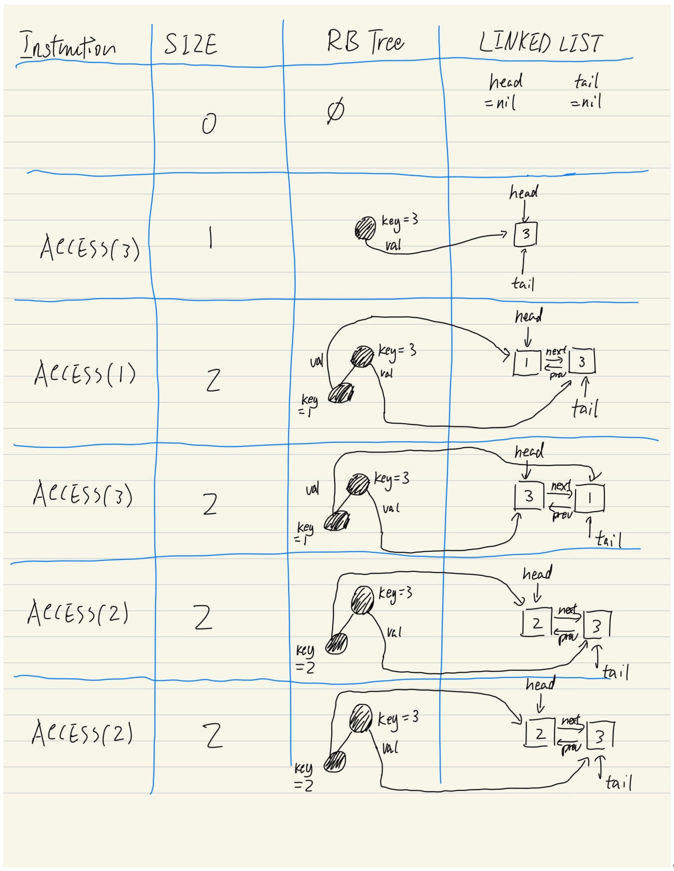
Give a data structure for this abstract data type that uses $O(k \log m)$ space (measured in bits) and has worst case time complexity $O(\log k)$.
Draw a diagram of your data structure when $k = 2$ and the sequence

$$\text{ACCESS}(3),\ \text{ACCESS}(1),\ \text{ACCESS}(3),\ \text{ACCESS}(2),\ \text{ACCESS}(2)$$

is performed starting with $C = \emptyset$.
Clearly describe the representation of your data structure, how to perform ACCESS, why it is correct, and why it satisfies the required complexity bounds.
Do not use pseudocode.

**Solution:**
Our data structure will consist of a red-black tree and a doubly linked list, together with pointers *head*, *tail* (initial values NIL), and integer *size* (initial value 0). For a simple example of how it works, here is a diagram of our data structure executing the sequence

$$\text{ACCESS}(3),\ \text{ACCESS}(1),\ \text{ACCESS}(3),\ \text{ACCESS}(2),\ \text{ACCESS}(2):$$



.

Call the red-black tree $t$, and the linked list $q$. Our data structure stores elements of $C$ in both $l$ and $q$. In $t$, the keys of the nodes are elements of $C$ and the values are pointers to their corresponding nodes in the linked list. In $q$, the nodes store the elements of $C$ along with the necessary pointers for $q$ to be doubly linked. Pointers *head* and *tail* always point to the first and last nodes of $q$, respectively. The intended use of *tail* is to reference the least recently accessed element, and *head* for most recently accessed. In summary $p \in C$ iff $p$ is a key in $t$ and a node in $q$.

Function ACCESS($p$) is performed in the following way.

(a) If $size < k$ and $p$ is not a key in $t$ [2], then: we first create a new node $x(p)$ with value $p$ and make that the new head of $q$ (when we say "make some $x$ the head", we mean we set $x.next = head$, $head.prev = x$, let *head* point to $x$, and adjust *tail* if necessary); after that, we insert $(p, x(p))$ ($p$ as the key and $x(p)$ as the value) into $t$ using RB-INSERT from our text (page 315).

(b) If $p$ is a key in $t$, then we link $x(p).prev$ with $x(p).next$ (if they exist), and then make $x(p)$ the head of $q$.

(c) When $size = k$: we properly remove from $q$ the node $p'$ that *tail* references and point *tail* to the node that preceded $p'$ (if that exists); then, we remove $(p', x(p'))$ from $t$ (this requires a binary search on $t$ to find $p'$); finally, we make $x(p)$ the head and add $(p, x(p))$ to $t$.

We now explain why this algorithm is correct.

Note that when the first node $x$ of $q$ is inserted, we have $x = head = tail$, which is correct since $x$ is indeed the least and most recently accessed element. We can know that $q$ is sorted by most recently accessed to least recently accessed if we can maintain this property every time we ACCESS.

Also, notice that ACCESS($p$) when $p$ is already in $C$ does not change $p$'s membership in $C$: point 2b only rotates the node $x(p)$ to the head. This means our algorithm satisfies the requirement that $C \cup \{p\} = C$ whenever $p \in C$. Also, we are able to maintain $|C| \leq k$ during ACCESS: point 2a maintains $size = |C| \leq k$ since $|C| + 1 \leq k$ when $size < k$; point 2b maintains $|C| \leq k$ since we first remove an element from $t$ and then insert, leaving $|C|$ unchanged.

The case when $|C| < k$ and $p$ is not a key in $t$ is correctly handled in point 2a because we added $(p, x(p))$ to $t$ and $p$ to $q$, and $x(p)$ is made the new head.

In the simple case where $p$ is already in $C$, we did not change the membership of $p$ as described above, and we made $x(p)$ the most recently accessed by making it the head.

Lastly, when $size = k$ and $p$ is not a key in $t$, we know that ACCESS will "cause" $C$ to overflow. So we first decrease the size of $t$ by removing the least recently accessed element, referenced by *tail*. We maintained the invariant by setting the second least recently accessed element the new tail. Then, we added $x(p)$ to $q$ as head, which also maintains the invariant. Lastly $(p, x(p))$ is added to $t$ so that $p \in C$.

Now for space complexity, assume memory addresses take $y$ bits to store (and we know storing a positive integer $\leq m$ takes $c = \lceil \log_2 m \rceil$ bits): *head* and *tail* take up $2y$ bits; each of the

[2] the second condition is verified through a binary search on $t$

at most $k$ node in $q$ takes $2y + c$ bits since the list is doubly linked and the node stores an integer. Therefore the linked list takes $2y + k(2y + c) \in \mathcal{O}(k \log m)$ space to store.

Also, our RB tree takes $\mathcal{O}(k \log m)$ space to store. Indeed, we store $k$ nodes each of which has a an integer value from 1 to $m$, taking up $c$ bits. And each node contains a pointer to the corresponding node in $q$, taking up $y$ bits. Therefore storing the tree takes $k(c + y) \in \mathcal{O}(k \log m)$ space.

Time complexity: in point 2a, checking if $p$ is a key in $t$ requires binary search on $t$, taking $O(\log n)$ since the height is $O(\log n)$ by CRLS page 309; in point 2c, deleting the least recently inserted with RB-Delete from $t$ takes $O(\log n)$ by CLRS page 324. Also by CLRS page 206, LIST-INSERT, which inserts an element into a linked list, takes constant time. Deleting the tail of the linked list takes constant time as well, as it requires us to simply use our pointer to the tail, delete the element, and update the previous element's 'next' pointer to NIL. All other operations are constant time. Thus, ACCESS is $O(\log n)$ time.