# CSC265 Fall 2020 Homework Assignment 3

Student 1: Yizhou Yang
Student 2: JiaWei Chen

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:**
   We define the notion of size as the number of leaves the subtree contains. We have an leaf oriented red-black tree that arranges leaves in the order of the sequence, therefore this is not a binary search tree, but can still have a height of log(n) due to its r-b property. In the leaves we store 4 parameters: value, color, a pointer to parent, and size (which is always 1 for the leaves). In the non-leave nodes, we store the size, the maximum value of its leaves, and pointers to parent and subtrees.

   (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:**
   Suppose that we are given a tree S. we can access its root via S.root, which will be used in code. First, we travel to the ith node and jth node from root using the size element of each node's subtree.

```
1  TRAVEL(x, node, height)
2       while node.left != NIL //by the structure of the tree, right is not NIL also
3            if x>node.left.size
4                 node = node.right
5                 x = x-node.left.size
6            else
7                 node = node.left
8            height = height + 1
9       return node
10
```

   Rename the ith node begin, and the jth node end. Suppose their first common ancestor (the ancestor with the greatest height) is node g. This g will be used in the later parts. The function has two modes: when considering the node begin and its ancestors, CLIMB first goes to its parent, then returns the maximum value of the current max and the maximum of the right subtree if begin is the left subtree. When considering the node end and its ancestors, it returns the current max and the maximum of the left subtree

if end is the right subtree.

```
10
11 //for begin and its ancestors, the climb's mode is 0. For end and its ancestors it is 1.
12 //MAX() returns the larger of the two parameters
13 CLIMB(node, mode, curr_max)
14       p = node.parent
15       if mode==0 && node == p.left
16             return MAX(curr_max, p.right.max)
17       else if mode ==1 && node == p.right
18             return MAX(curr_max, p.left.max)
19       return curr_max
20
```

For SUBSEQMAX, we first find the nodes representing begin and end. Then, we CLIMB the node with a greater height until they are equal in height, so that we would know when we reach the two node's common ancestor. Then, we CLIMB them both by one level each time until they reach their common ancestor. When the two nodes meet, the current maximum value will be the answer.

```
21 //height1 and height2 are global, so that after TRAVEL, they reflect the depth of begin and end.
22 height1 = 1
23 height2 = 1
24 SUBSEQMAX(S, i, j)
25       count = 0;
26       begin = TRAVEL(i, S.root, height1)
27       end = TRAVEL(j, S.root, height2)
28       curr_max = MAX(begin.value, end.value)
29       if height1>height2
30             h = height1 - height2
31             while h>0
32                   curr_max = CLIMB(begin, 0, begin.value)
33                   h = h-1
34                   begin = begin.parent
35       else
36             h = height2 - height1
37             while h>0
38                   curr_max = CLIMB(end, 1, end.value)
39                   h = h-1
40                   end = end.parent
41
42     while begin != end
43             num1 = CLIMB(begin, 0, curr_max)
44             num2 = CLIMB(end, 1, curr_max)
45             curr_max = MAX(num1, num2)
46             begin = begin.parent
47             end = end.parent
48     height1 = 1
49     height2 = 1 //reset height1 and height2 for the next SUBSEQMAX call
50     return curr_max
```

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

**Solution:**

Lemma: for any ancestor of begin with height greater than g, if begin is in left subtree, then end is not in right subtree.

proof: We prove by contradiction. If begin is in left subtree, and end is in right subtree, then this node is an ancestor of both begin and end. Therefore, this node will either be g itself or be g's ancestor, which contradicts the fact that it has height greater than g.

Remark: similarly, if end is in right subtree, then begin is not in left subtree.

Lemma 1: for any ancestor of begin with height greater than g, if begin is in left subtree, then the all leaves in the right subtree are values between S[i] and S[j].

proof: initially, consider the parent of begin. Assume it is not g and begin is in its left subtree. If end is in its right subtree, then it will be g, so end is not in its subtree, but to the right of it, since end¿begin if they do not equal. By the definition of the tree, we stored data in the order of the sequence, so the right subtree's leaves will be between S[i] and S[j].

Now consider an arbitrary ancestor of begin with height greater than g and assume this claim is true for all its children. Assume begin is in its left subtree. Consider its left child, it is an ancestor of begin, so it satisfies the inductive hypothesis. Consider its right child. Since this node has greater height than g, it is not an ancestor of end by the definition of g. Therefore, end is to the right of every leaf in its right subtree since end exists and end is greater than begin. Therefore, all the right subtree's leaves will be between S[i] and S[j]. Therefore the lemma is proven by induction.

Remark 1: Following the exact same proof structure, we can prove that for any ancestor of end with height greater than g, if end is in right subtree, then the all leaves in the left subtree are between i and j.

Lemma 2: This tree's height will still be O(log(n)) proof: we can use the exact same proof for proving the LOBS's height in HW2 that involved black height, because this tree follows the same r-b constraints.

For any ancestor of begin with height greater than g, denote this node p, denote its rightmost leaf S[m]. by lemma 0 we know m is smaller than j. Lemma 3: CLIMB(p,0,currmax) returns the maximum value among the ith through mth elements of sequence S, where currmax is the maximum value returned by CLIMB from the ancestor of begin with height 1 more than p.

base case: initially, consider the parent p of begin, currmax = begin.value. If begin is p's right child, then all leaves in p's left subtree will be before S[i]. CLIMB returns begin.value, which is as expected.

Inductive step: Now, consider an arbitrary ancestor p of begin. before CLIMB, currmax is the maximum value returned by CLIMB from the ancestor of begin with height 1 more than p, call it f.

Case 1: f is the left child of p. Then by lemma 1, p's right subtree will be between S[i] and S[j]. line 15 will be triggered. By definition, p.right.max will store the maximum value of all its leaves. if the max value of S[i] to S[m] is in the right subtree of p, then CLIMB will return this max, as expected. if the max value of S[i] to S[m] is in the

left subtree of p, then CLIMB will return the return value of the CLIMB call on f. By inductive hypothesis, this the max value of S[i] to S[m'], where S[m'] is the rightmost node in p's left subtree. This is also correct.

Case 2: f is the right child of p. Then, the left child of p's leaves will all be to the left of begin, so their values will not be between S[i] and S[j]. CLIMB returns the return value of CLIMB on f, which by inductive hypothesis, this the max value of S[i] to S[m'], where S[m'] is the rightmost node in p's left subtree. This is also correct.

Therefore, by induction, CLIMB(p,0,currmax) returns the maximum value among the ith through mth elements of sequence S.

Remark 2: Similarly, For any ancestor of begin with height greater than g, denote this node q, denote its leftmost leaf S[n]. CLIMB(q,1,currmax) returns the maximum value among the nth through jth elements of sequence S.

Lemma 4: after lines 29-40, begin and end will have the same height, they will still be ancestors of the node representing S[i] and S[j]. If line 29 is triggered, currmax will store the max value of S[i] to S[m'], where S[m'] is the rightmost node in begin's right subtree.

proof: assume line 29 is triggered. Initially, h will be height1-height2, which is greater than 0. begin is the node that represent S[i], and begin has no subtrees. Consider an arbitrary iteration and assume this claim is true before the iteration. Then, line 30 will make currmax store the max value of S[i] to S[m'], as proven in Lemma 3, line 33 will reduce the height of h by 1, which corresponds the decrease of begin's height by 1, and begin points to its parent. The new begin still contains S[i], and the claim is maintained. When the loop terminates after h iteration, begin's level is raised up h times, which is exactly the difference between begin and end, therefore they will have the same height.

Remark 3: Similarly, if line 35 is triggered, currmax will store the max value of S[m'] to S[j], where S[m'] is the leftmost node in end's left subtree.

Lemma 5: At any iteration of TRAVEL, S[x] is the x-th leaf of node, node either is or is the ancestor of S[x], and TRAVEL(x,S.root) will return the node that represents S[x] (so that begin and end is what we think it is)

proof: Initially, node points to the root. If there is only one node in the tree, then this root is S[x], else root is the ancestor of any leaf. x is equal to i initially, and S[i] is the i-th leaf of the root.

Consider an arbitrary iteration and assume the first part of the claim is true before the iteration. Case 1: S[i] is in the left child of node. then x will be smaller or equal to node.left.size, since there are node.left.size number of leaves in the left subtree, and

S[i] is the x-th leaf of it by the claim, so we point node to its left subtree as in line 7, and the claim is maintained. Case 2: S[i] is in the right child of node, then there will be node.left.size number of leaves in the left subtree, and since S[i] is in the right subtree, x will be the x-node.left.size element in the right subtree, so we reassign x to be x-node.left.size element and point node to its subtree, and the claim is maintained. Therefore, at termination, since node has no children, it will be S[x] itself.

Lemma 6: at the beginning of every iteration of loop in line 42, begin is the ancestor of S[i], end is the ancestor of S[j], and after the loop terminates, currmax is the max value of S[i] to S[j].

proof: Initially, begin is either itself if line 29 did not execute, or the ancestor of S[i] by lemma 4. currmax is the larger of begin and end. If the loop did not execute, then the two nodes are the same and we return the node, which is correct. Begin and end also have the same height by lemma 4.

We first prove that we can find g by reducing both height one at a time. We know that g will not be in any node greater or equal this height because the two leaves must converge on an internal node with is both their parent, and at this height one node is still a leaf, which mean that they can not have a common ancestor. So g is less than this height, and since g is a single node, it must have a single height. Since for any two nodes of a r-b tree, they must have a common ancestor (trivially by the property of a tree), this g must exist. We then prove by contradiction. Assume that when we found g, the height of begin and end do not equal. Then since g is when begin and end meet, its height is equal to begin. Since g is when begin and end meet, its height is equal to end. This contradicts the fact that it has only one height.

recall a definition: For any ancestor of begin with height greater than g, denote this node p, denote its rightmost leaf S[m]. by lemma 0 we know m is smaller than j. Similarly for end we denote this node q, denote its leftmost leaf S[m'].

Suppose we can find g when line 42 terminates, by lemma 3, CLIMB(p,0,currmax) returns the maximum value among the ith through mth elements of sequence S, where m is where currmax is the maximum value returned by CLIMB from the ancestor of begin with height 1 more than p. By remark 2, CLIMB(q,1,currmax) returns the maximum value among the m'th through ith elements of sequence S, where currmax is the maximum value returned by CLIMB from the ancestor of end with height 1 more than q. When begin and end converge, CLIMB(q,1,currmax) will return the greatest value of g's right subtree that is between S[i] and S[j] by lemma 1 and lemma 3, and CLIMB(p,0,currmax) will return the greatest value of g's right subtree that is between S[i] and S[j] by lemma 1 and lemma 3. So their MAX() will return the greatest value of g's leaves between S[i] and S[j].

We then prove that there is no leaf between S[i] and S[j] outside g's leaves. We know that S[i] is in g.left, and any node to the left of it is not in S[i] to S[j] by the structure

of the tree. We know that S[j] is in g.right, and any node to the right of it is not in S[i] to S[j] by the structure of the tree. Consider any leaf that is not g's children, then it is either to the left of g's rightmost leaf or to the left of g's leftmost leaf because g's leaves are consecutive. So it is not in S[i] to S[j]. Therefore when line 42 terminates, currmax is the maximum value between S[i] and S[j].

Theorem 1: SUBSEQMAX(i,j) returns the maximum value among the ith through jth elements of the sequence S, which in our tree is represented by the ith and jth leaves.

proof: by lemma 5, begin is the node representing S[i], end is the node representing S[j]. If i=j, then neither line 29 nor line 42 executes, and S[i] is returned, as expected. Assume they are not equal. by lemma 4, before line 42, begin and end will have the same height, they will still be ancestors of the node represents S[i] and S[j]. Since only one of begin and end is changed, the other is still a leaf and that g is by definition a internal node, initially begin is not equal to end. By lemma 6, after the loop terminates, currmax will store the maximum between S[i] and S[j] , which will be returned, as expected. Therefore SUBSEQMAX is correct.

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

**Solution:**

Consider TRAVEL. By lemma 2 and HW2, the height of the tree is at most 3log(n+1). let h denote the height of node. initially, h is 1. For each iteration of the while loop, node either becomes its left child if line 3 is executed, or becomes its right child if the else clause in line 6 executes. either way, h will increase by 1. The loop terminates by the test in line 2, which means that node is currently a leaf. By the structure of the tree, this means that the height is at most the tree's maximum height. Since h increases by 1 each iteration, there will be O(h) = O(logn) iterations. For each iteration, 3-5 or line 6-7 executes, and both take constant time, so TRAVEL's worst case complexity is O(logn).

CLIMB takes up constant time because lines 15-16 or line 17-18 both takes constant time if executed, and line 14 and line 19 are constant time as well.

Consider SUBSEQMAX. At most one of the branches(or neither) of line 29 or 35 will execute depending on the height difference of begin and end. Assume we execute line 29, because line 35 can be proven using the same proof. Since their height is at most 3log(n+1), so is their height difference, therefore, h is in O(logn). Since for each iteration of line 31, h decreases by 1, the while loop will execute O(logn) times. Each iteration of the loop takes constant time since CLIMB takes constant time. Therefore, the loop in either branch 29-34 or 35-41 is in O(logn).

Consider the while loop in line 42. From the previous proof of correctness we know that begin and end will have a common ancestor g after an arbitrary number of iterations of line 42. Since thee are at most 3log(n+1) levels in the tree and that each iteration of the loop raises both begin and end by one level, after at most 3log(n+1) iterations they will meet. each iteration involves constant time operations and subfunctions, so this loop is O(logn).

Therefore, SUBSEQMAX take O(logn) time to complete.

Notice that in the worst case, the tree has height of 3log(n+1). In the case that either i or j is the node with the greatest height, TRAVEL on a tree with length 3log(n+1) has $\Omega(logn)$ worst case time complexity, since the height start at 0, each iteration increase the height by 1, and after termination height is 3log(n+1). Therefore, even disregarding other subprograms, SUBSEQMAX will have $\Omega(logn)$ worst case time complexity.

Therefore, SUBSEQMAX take $\Theta$(logn) time to complete.

(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.

**Solution:**

For APPEND(S,x), assume there are n nodes in the tree S(which is the value of parameter size of the root). we first TRAVEL(S,n) to find the last node. Then, using the methods described in HW2, we replace the last node with a 3-node subtree with the original last node as the left leaf and the appended node as right leaf. We rotate the tree if it violate any of the red-black specifications, then update the depth and size of the inserted node's parents and all rotated nodes. For INSERT(S,i,x), we TRAVEL(s,i) to find the i-th node, then we replace the last node with a 3-node subtree with the original ith node as the right leaf and the inserted node as the right leaf. We then rotate the tree and update the depth and size of the inserted node's parents and all rotated nodes.

Consider the size parameter of internal nodes. Tree rotation according to HW2 preserves the red black property and a relative height of O(logn). For any node, its size is just the size of the left subtree and right subtree, so by CLRS theorem 14.1, this property is preserves in both insert and append.

Consider the maximum value of internal nodes. Since rotation only rotates the internal nodes, the leaves that belonged to the subtree's root still belong to the subtree's new root. For any node, its maximum value is just the max of the left subtree and right subtree, since all the values are stored in the leaves. By CLRS theorem 14.1, this property is preserves in both insert and append.

The color and pointers are updated according to HW2, so it is correct and takes O(logn) time during both insert and append.

Since all the attributes are maintained in O(logn) time, and all the parameters will be correct after rotations, both APPEND an d INSERT are correct and takes O(logn) time.

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

ACCESS(3), ACCESS(1), ACCESS(3), ACCESS(2), 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 implementation of CACHE consists of two counters and two red-black trees. We will call them tree $P$ and tree $Q$. The first counter $s$ keeps track of the size of subset. The second counter $t$ keeps track of the number of times $ACCESS$ has been called.

Each node in tree $P$ stores a key $p$ that correspond to the number accessed and a pointer to a node in tree $Q$ that represents the time accessed (For example, if the node is $A$, we will refer to them as $A.key$ and $A.time$). The keys in this tree represent the subset $C$ specified in the question.

Each node in tree $Q$ stores a key $q$ that records $t$ when ACCESS is called, and a pointer that that points to a node in $Q$ that represents the number accessed (if the node is $B$, we will refer to them as $B.key$ and $B.value$).

Initially, both counters are 0, and both trees are empty. (And we expect $t$ to be a non-negative integer, and $s$ to be an integer that satisfies $0 \leq s \leq k$)

When ACCESS($p$) is performed, the following steps take place:

(In this passage, DELETE, INSERT and SEARCH refer to the corresponding operations in the standard red black tree.)

First, perform SEARCH($p$) in tree $P$ and then check the value of $s$. We can divide the situations into three cases.

CASE A. SEARCH($p$) in $Q$ finds node $A$.

1. DELETE $A.time$ from tree $Q$
2. INSERT a new node $B$ with key $B.key = t$ into $Q$
3. Set $A.time = B$, and set $B.value = A$.
4. Increment $t$ by 1.

CASE B. SEARCH($p$) returns $nil$ (that is, key $p$ is not in $P$) and $s < k$.

1. INSERT a new node $A$ with $A.key = p$ into $P$.
2. INSERT a new node $B$ with $B.key = t$ into $Q$.
3. Set $A.time = B$, and set $B.value = A$.

4. Increment $s$ by 1.
5. Increment $t$ by 1.

CASE C. SEARCH($p$) returns $nil$ and $s = k$.

1. Look for the node with the smallest key in node $Q$ (This is implemented by always taking the left path until a leaf is found).
2. DELETE that node and its corresponding node in $P$.
3. INSERT a new node $A$ with $A.key = p$ into $P$.
4. INSERT a new node $B$ with $B.key = t$ into $Q$.
5. Set $A.time = B$, and set $B.value = A$.
6. Increment $t$ by 1.

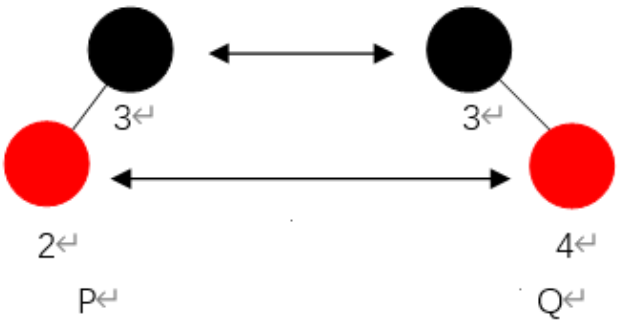


Figure 1: This diagram represents the data structure after the sequence of ACCESS.

**LEMMA 1:** After ACCESS runs, $t$ is a non-negative integer, and $s$ is an integer that satisfies $0 \leq s \leq k$.

**Proof:**

Base case: Initially $t = 0, s = 0$, then ACCESS uses the CASE B. After the execution, $t = 1, s = 1$. The conditions still hold.

Inductive case: Assume $t$ is a non-negative integer, and $s$ is an integer that satisfies $0 \leq s \leq k$ after ACCESS has been called $r$ times. We call ACCESS once more, it uses one of the three cases.

Regardless of which case it performs, $t$ increments by 1. $t+1$ is certainly a non-negative integer because $t$ is a non-negative integer by assumption.

$s$ increment by 1 if $s < k$ (Case B). Otherwise it doesn't change at all. As $k$ is also an integer, $s < k$ implies $s + 1 \leq k$. This is enough to show that $s$ remains $0 \leq s \leq k$ after ACCESS.

By induction on $r$, the statement is true. □

**REMARK 1:** The counter $t$ equals the number of times ACCESS has been called, as it starts with $t = 0$ and increments each time ACCESS runs. Moreover, the value $t$ is never the same for two different calls of ACCESS.

**REMARK 2:** Tree $P$ and $Q$ always have the same size, and the size equals $s$.

**LEMMA 2:** Nodes in $P$ have distinct keys. Nodes in $Q$ have distinct keys.

> **Proof:** This is clearly true for $Q$ because its keys are the value of counter $t$ when the nodes are created. And by REMARK 1, $t$ is distinct for distinct calls of ACCESS.
>
> The statement is also true for $P$. Because before insert new node with key $p$ to $P$, we need to check that SEARCH($p$) returns *nil* (referring to Case B and C). And Case A does not add new node to it. □

**LEMMA 3:** The node in $P$ that represents the least recently accessed number is paired with the node with the smallest key in $Q$.

> **Proof:** Let's say $A$ is the node that represents the least recently accessed number. Suppose the lemma is false, then there exists a node $B$ in $Q$ that has a smaller key.
>
> This implies that $B$ was created in an ACCESS call before $A.time$. So $B.value$ would be the node that represents the least recently accessed number by definition.
>
> By contradiction, this lemma is true. □

**REMARK 3:** By LEMMA 3, line 1,2 in case B removes the pair of nodes that represents the element that was least recently the parameter of an access operation.

**THEOREM 1:** The algorithm described above is correct.

> **Proof:** By the lemmas and remarks above, it's evident that the algorithm in each case modifies the data structure as we required. Also, the three cases cover all possible situation disjointly.
>
> The program does not directly run any loops or recursive structure. (Although it used DELETE, INSERT, SEARCH of the red-black tree that depends on loop/recursion. We assume the correctness of those operations.) So the program must terminate correctly.
>
> In other words, the algorithm is correct. □

**THEOREM 2:** ACCESS runs in $O(\log k)$ time.

> **Proof:** Need to show that ACCESS runs in $O(\log k)$ time for each of the three cases.
>
> Before we enter any of the cases, we need to check the condition first. Checking the value of $s$ is a constant-time operation, and performing SEARCH($p$) takes $O(\log k)$ as the tree has at most $k$ nodes.
>
> > For CASE A: line 1, 2 invoke DELETE and INSERT, which run in $O(\log k)$ time. Line 3, 4 runs in $O(1)$ time. Overall, CASE A takes $O(\log k)$ time.
> >
> > For CASE B: line 1, 2 invoke INSERT, which run in $O(\log k)$ time. Line 3, 4, 5 runs in $O(1)$ time. Overall, CASE B takes $O(\log k)$ time.
> >
> > For CASE C: line 1 takes $O(height)$ steps. Since height is $O(\log k)$ for a red black tree with at most $k$ nodes. Line 2, 3, 4 invoke INSERT, which run in $O(\log k)$ time. Line 5, 6 runs in $O(1)$ time. Overall, CASE C takes $O(\log k)$ time.

So ACCESS is an $O(\log k)$ operation. $\square$

**THEOREM 3:** The data structure has space complexity $O(k \log m)$.

**Proof:** Consider a number of the order $2^6$, for example. It takes 7 bits to represent this number. For something of the order $2^{6+n}$, then it takes $7 + n$ bits to store it. We can deduce that the bits it takes to store an integer $m$ has $O(\log m)$ space complexity. In the worst case scenario, all of the numbers accessed are of the same order as $m$, i.e. each takes up $O(\log m)$ space.

Each tree can have at most $k$ nodes. Then the number of nodes is $2k$ in total. So the total space required is $2kO(\log m) = O(k \log m)$. $\square$