Finding the longest monotonically increasing subsequence of a sequence of n numbers in $O(n \lg n)$

Jiman Hwang (pingdummy1@gmail.com)

October 13, 2017

Disclaimer: This document is for self-study only. If you find an error, feel free to contact me.

1 Problem

Consider the longest monotonically increasing subsequence(LMIS) of a sequence of n numbers(problem 15.4-6 at [THCS09]). For example, given a sequence $[3,2,5,3,6,8,1,6]$, the LMIS are $[3,5,6,8]$, $[2,5,6,8]$, $[3,3,6,6]$, $\cdots$. Give an $O(n \ln n)$-time algorithm to find any of them.

2 Solution

2.1 Main idea

Given the input is given as an array $A[1..n]$, let $LMIS_i$ denote the LMIS of $A[1..i]$, and $LMIS'_i$ the LMIS that ends with $A[i]$. Note that the last value of $LMIS_n$ should be one of the input values. To put it another way,

$$LMIS_n \in \underset{\forall i\ LMIS'_i}{\arg\max} \left|LMIS'_i\right|$$

Therefore, what we have to do is gather all $LMIS'_i$s and pick up the maximum. We use induction for this. Clearly $LMIS'_1 = A[1]$ for initial case. Suppose we already know $LMIS'_j$ for $j = 1, \cdots, i-1$. We pick up one monotonically increasing subsequence of $A[1..(i-1)]$ and link it to $A[i]$ for gaining $LMIS'_i$. Note that any monotonically increasing subsequence of $A[1..(i-1)]$ ends with one of $A[1], \cdots, A[i-1]$. For each case, the LMISs are $LMIS'_1, \cdots, LMIS'_{i-1}$, respectively. This means we have only $i-1$ candidates to which we append $A[i]$ because we're looking for the longest one. Also, we need to filter out $LMIS'_j$ whose last value is larger than $A[i]$. Otherwise, there will be decreasing subsequence, or a contradiction, when we build $LMIS'_i$. All things considered,

$$LMIS'_i \in \left\{ LMIS'_j \parallel A[i] : \left(\text{the last value of } LMIS'_j\right) \le A[i] \ \wedge\ LMIS'_j \in \underset{\forall k<i\ LMIS'_k}{\arg\max} \left|LMIS'_k\right| \right\}$$

where $\parallel$ means concatenation. There may be multiple $LMIS'_i$ candidates, and we choose any of them.

2.2 Data structure

The auxiliary data structure is a binary search tree whose each node X has following properties. Let $LMIS'_X = LMIS'_i$ when X contains i^{th} input value, and T_X denote the subtree rooted at X.

- basic binary search tree properties : *key*, *parent*, *left*, *right*
- *len* : the length of $LMIS'_X$
- *prev-node* : the pointer to the node whose key is the previous one of X's key in $LMIS'_X$
- *max-node* : the pointer to the node whose *len* is maximum in T_X

Each input is inserted into *key*. *prev-node* is for rebuilding $LMIS'_X$, acquired by tracing back through *prev-node*. *max-node* helps to compare each *len* throughout a subtree.

In this document we use the following illustration. For a node,

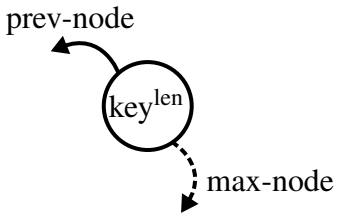

Figure 1: Illustration of a node

A node is presented as a circle that contains *key* and *len*(superscripted), a normal arrow that points *prev-node*, and a dashed arrow that points *max-node*. To illustrate, suppose the input sequence is $[6,5,8,7]$. We have built the tree as follow.

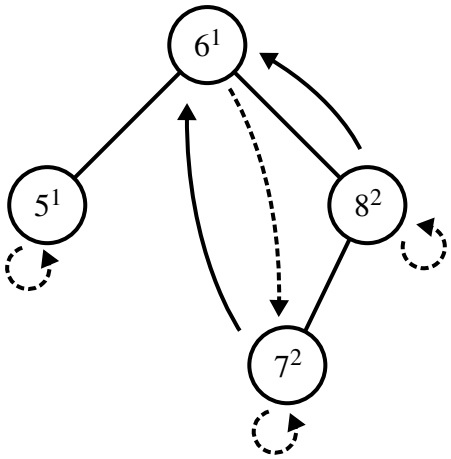

Figure 2: An example of tree formed with the input $[6,5,8,7]$

We will explain how to build it later. For now, let us shed light on each property.

First of all, tree is formed using *key*s as a binary search tree.

Next, $LMIS'_5 = 5, LMIS'_6 = 6$. Therefore, their length, 1, is marked as superscripted one on each key. Meanwhile, there are two $LMIS'_8$ candidates: <5, 8>, <6, 8>. Since the longest one has length of 2, the node 8 shows up with *len* 2. Similarly, so does 7.

Because $\left|LMIS'_5\right| = \left|LMIS'_6\right| = 1$, their *prev-node*s must be empty, shown as no normal arrow. At the same time, the node 8 has the previous key in $LMIS'_8$: 5 or 6. In this illustration, we chose 6, but this may vary depending on implementation. The normal arrow from node 8 to node 6 is showing the *prev-node* of node 8. Similarly, we chose 6 as *prev-node* of node 7, which presented normal arrow from node 7 to node 6.

Lastly, *max-node*s are presented as dashed arrows. Observe that all leaves' *max-node*s are pointing themselves because of no child. Node 8, however, has a child, node 7. Hence, we should compare *len* of node 7 and that of 8. In case of deuce, again, the choice hinges on the implementation. In our example, it's itself. For *max-node* of node 6, we pick up node 7 between node 7 and node 8, whose *len*s are the largest as 2. Once again, you can select node 8 instead of node 7.

2.3 Max-node correction

Before we start out, let us introduce an auxiliary function.

```
FIX-MAX-NODE(x)
1  if x.left ≠ NIL and x.left.max-node.len > x.len
2      x.max-node = x.left.max-node
3  elseif x.right ≠ NIL and x.right.max-node.len > x.len
4      x.max-node = x.right.max-node
5  else
6      x.max-node = x
```

Assuming that each child of a node x has the correct *max-node* or is a NIL, FIX-MAX-NODE(x) set $x.max\text{-}node$ to the correct one. By definition, *len* of all nodes in T_x needs to be compared. However, we already have the maximum values of $T_{x.left}$ and $T_{x.right}$. Hence, we just need to find the largest value among $x.left.max\text{-}node.len$, $x.right.max\text{-}node.len$, and $x.len$. Setting the node as *max-node* of the largest value finishes the job.

2.4 Building the tree

By inserting each input one by one from the first one to the last one, we acquire LMIS. Let's take a look at one insertion first. The TREE-INSERT(T, k) takes a tree T and a key k, or input value, and inserts k into T.

```
TREE-INSERT(T, k)
     // Find the right position of k maintaining prev-node
 1   t = T.root
 2   prev = NIL
 3   parent-node = NIL
 4   while t ≠ NIL
 5       parent-node = t
 6       if t.key ≤ k
             // prev vs t
 7           if prev == NIL or prev.len < t.len
 8               prev = t

             // prev vs t.left.max-node
 9           if t.left ≠ NIL and prev.len < t.left.max-node.len
10               prev = t.left.max-node

11           t = t.right
12       else
13           t = t.left

     // insert the new node
14   Y = (a new node)
15   (Insert Y into T as binary search tree using parent.)
16   Y.key = k
17   Y.max-node = Y
18   Y.prev-node = prev
19   if prev == NIL
20       Y.len = 1
21   else
22       Y.len = prev.len + 1

     // Fix up along from Y to the root
23   t = Y.parent
24   while t ≠ NIL
25       FIX-MAX-NODE(t)
26       t = t.parent
```

Like a normal binary search tree, we find the leaf node to add a new node using keys.

Initialization is shown in lines 1-3. *prev* is used for the *prev-node* of the new node. *parent-node* points t's parent, solely for linking as binary search tree.

From lines 4-13, it's the procedure to find the position of new node, except lines 7-10. Hence, let's focus on lines 7-10, which maintains *prev*. We will see the following loop invariant is true throughout lines 4-13.

If $prev \neq$ NIL, then $LMIS'_{prev}$ is the longest among all $LMIS'_X$s that satisfies all conditions below. 1. $X.key \leq k$. 2. $LMIS'_X$ consists of the nodes in T but T_t.	(1)

Proof.

Initialization

Before entering the loop, *prev* is NIL. This makes (1) true.

Maintenance

We'll look into two cases, whether $t.key \leq k$ or not. For each case, assume (1) is true at the start of iteration.

(i) $t.key \leq k$

t and all nodes in $T_{t.left}$ have keys less than k. Considering each node X is the last node of $LMIS'_X$, t and all nodes in $T_{t.left}$ satisfy the first condition of (1) when t becomes $t.right$. Next, we compare their *len*s. Instead of checking all of them, we take advantage of $t.left.max\text{-}node$, which contains the largest *len* in $T_{t.left}$. Hence our task is confined to comparing and choosing the node whose *len* is the largest among t, $t.left.max\text{-}node$, and *prev*. Lines 7-10 do this task.

Meanwhile, we shows that $LMIS'_{prev}$, $LMIS'_t$, and $LMIS'_{t.left.max\text{-}node}$ satisfy the second condition of (1) respectively when t becomes $t.right$. Hence, no matter what variable is chosen at lines 7-10 as *prev*, (1) is true at the end of iteration.

$LMIS'_{prev}$ satisfies the second condition of (1) because T_t becomes shrink into $T_{t.right}$.

Suppose $LMIS'_t$ includes a node z in $T_{t.right}$. This implies z appears earlier than t in $LMIS'_t$. However we know that every new node is added to a leaf, thus t must precede z in the input. Therefore this is a contradiction, and the second condition of (1) is true.

As far as $LMIS'_{t.left.max\text{-}node}$ is concerned, we know $t.left.max\text{-}node.key < t.key \leq k$. Considering that all keys in $T_{t.right}$ are bigger than or equal to k, and those in $LMIS'_{t.left.max\text{-}node}$ are less than or equal to $t.left.max\text{-}node.key$, the second condition of (1) is also true.

(ii) $t.key > k$

No node among t and the nodes in $T_{t.right}$ can replace *prev* because their keys are larger than k, denying the first condition of (1). Therefore, there's no change of *prev* through iteration. With the same way as the above case does, $LMIS'_r$ ending with *prev* satisfies all conditions of (1) when t becomes $t.left$. Hence (1) is true at the end of iteration.

Termination

When we exit the loop, t becomes NIL. This means that $LMIS'_{prev}$ is the longest one in T in constraint of $prev.key \leq k$.

□

Referring the main idea, let's say this call for TREE-INSERT was i^{th} one. It means we have $LMIS'_1, \cdots, LMIS'_{i-1}$(stored as each node, restorable through *prev-node*). *prev* from lines 1-13 is equivalent to $LMIS'_{prev}$ to which we'll append $A[i]$. This concatenation is done by setting *prev-node*.

The new node is created and attached to T in lines 14-22. Since Y is a leaf node, *max-node* should be itself. *key* is, of course, the input value, $A[i]$. *prev-node* must be *prev*, which makes Y equivalent to $LMIS'_{prev} \parallel A[i]$, or $LMIS'_i$. For *len*, if *prev* is NIL, this implies Y is the only node in $LMIS'_i$. Hence, the length must be 1. Otherwise, we set $Y.len$ to $prev.len + 1$.

After adding Y, all node properties should not be violated. Observing what properties are related to Y, we finds *max-node* is the only one that may rely on Y. In particular, all nodes on the simple path from the root to Y may need modifications. This is because $Y.len$ may become the largest value in the subtree rooted by each node on the simple path. The fixing up procedure is shown in lines 23-26. Knowing $Y.max\text{-}node$ is correct, we start from Y. From bottom to top, this fix up is done until the root has the right max-node value. At each iteration, one child of t is previously inserted one or a NIL. The other child of t is corrected by FIX-MAX-NODE or manually set(in case of initial iteration). Both children of t have correct *max-node*, satisfying the assumption for calling FIX-MAX-NODE. In sum, after all iterations, all nodes become fixed.

2.5 Eliciting the answer

GET-LMIS takes the input array filled with real numbers of size n, and returns the stack that contains the LMIS and its length.

```
GET-LMIS(A)
     // Build the tree
 1   T = (an empty binary search tree)
 2   for i = 1 to A.len
 3       TREE-INSERT(T, A[i])

     // Elicit the LMIS
 4   t = T.root.max-node
 5   lmis-len = t.len
 6   lmis = (an empty stack)
 7   while t ≠ NIL
 8       STACK-PUSH(lmis, t.key)
 9       t = t.prev-node

10   return lmis, lmis-len
```

Lines 1-3 forms the tree by inserting each input one by one. Note that the order must be from the first to the last. After every iteration, the *max-node* of root of T is always the last node in the LMIS of the inserted inputs so far.

After the tree is built, the *max-node* of the root is the last node in LMIS of the entire input. Since each node stores which node is used for making the LMIS, we can trace back in reverse order, revealing the LMIS of the input. Lines 4-9 does this job.

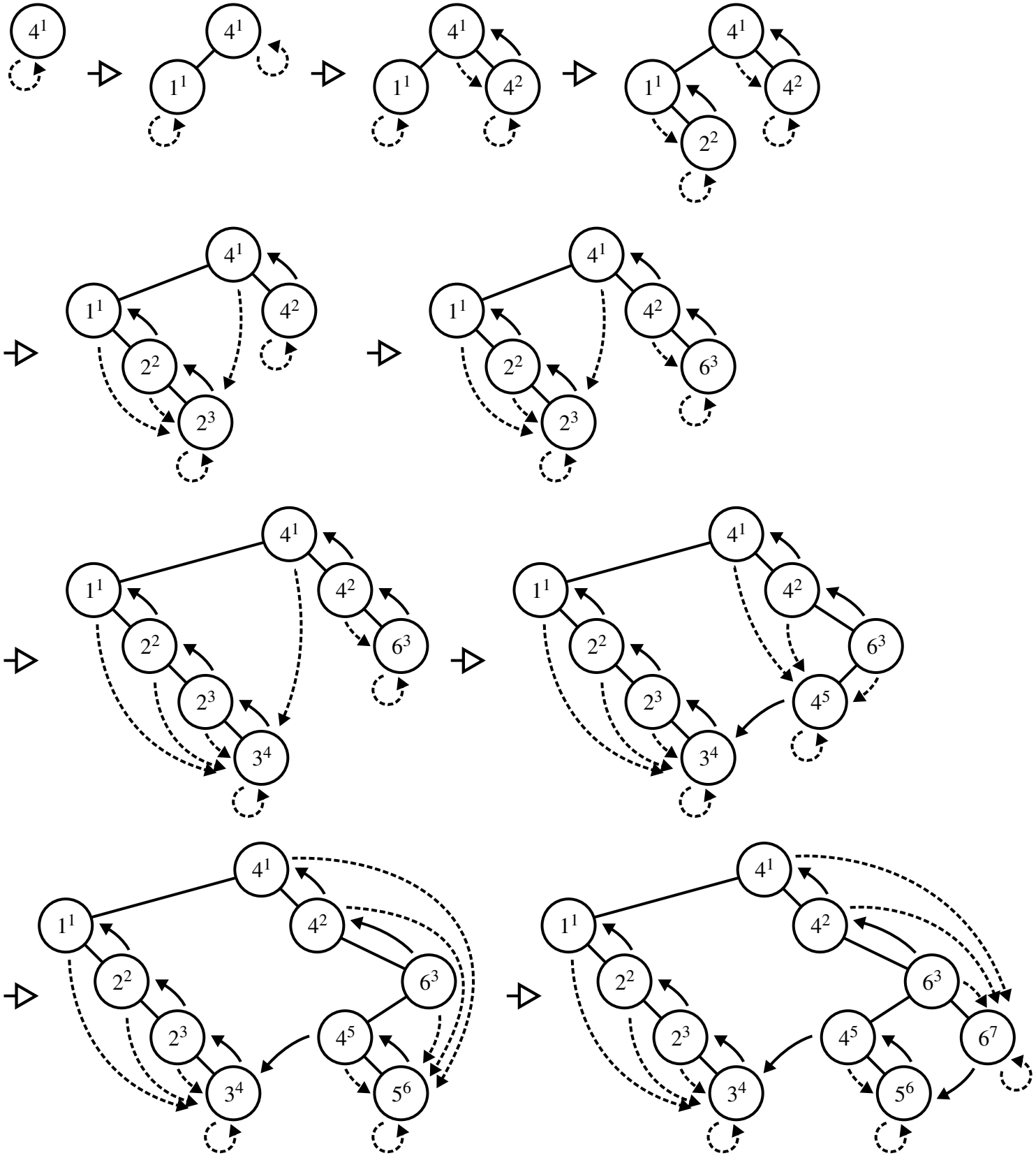

Figure 3: An example of tree formation process with the input $4, 1, 4, 2, 2, 6, 3, 4, 5, 6$

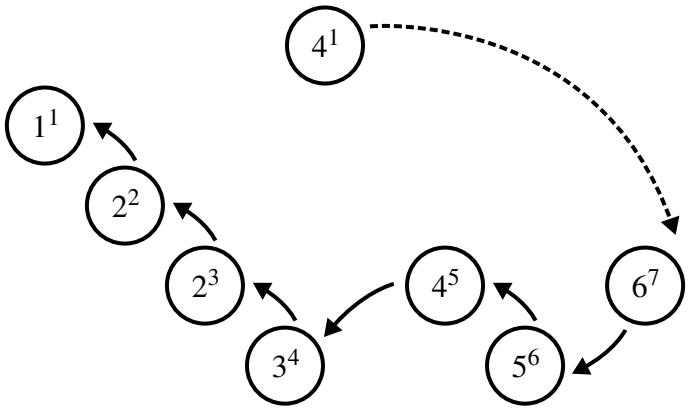

Figure 4: The answer of Fig 3: $1,2,2,3,4,5,6$

Fig 3 shows the entire procedure with the input $4,1,4,2,2,6,3,4,5,6$. The illustration looks complicated, yet you will be in a better position to understand the algorithm. After the tree is built, we acquire the answer $1,2,2,3,4,5,6$ by tracing back from $T.root.$*max-node* using *prev-node* until NIL appears, shown in Fig 4.

2.6 Performance

Let's start with TREE-INSERT. Lines 1-13 takes $O(h)$ where h is the height of T as t goes from the root to a leaf, and it takes $O(1)$ at every iteration. Lines 14-22 takes $O(1)$. Lines 23-26 takes $O(h)$ as t from a leaf to the root, and FIX-MAX-NODE takes $O(1)$. In sum, TREE-INSERT takes $O(h)$.

GET-LMIS calls TREE-INSERT n times at line 3. This consumes $O(nH)$ where H is the maximum height of T during iterations. Lines 4-9 takes $O(n)$ as the longest length of possible LMIS is n. To sum up, GET-LMIS takes $O(nH)$

2.7 Combination with balanced tree

As you have seen the above, the time complexity relies on the maximum height of the tree. This is where a balanced tree comes in for meeting the goal $O(n\lg n)$. In this document, we take advantage of balanced trees which harnesses rotations in common, such as red-black or AVL tree. If we maintain node properties during rotation in $O(1)$, we can attach additional codes for maintaining balanced tree properties at the end of TREE-INSERT, keeping the performance $O(h)$.

Aside from basic binary search properties, *len* and *prev-node* don't change by a modification of tree structure. *max-node* is the only property we have to deal with. Using symmetry, we explain how to preserve *max-node* during a right-rotation.

In Fig 5, we right-rotate y. α, β, and γ are representing subtrees respectively. *max-node* of each node in α, β, and γ doesn't need change because *max-node* is affected only by its subtree, and there's no change of the subtree. For this reason, our concern is constrained with x and y. To fix possible error of x and y, call FIX-MAX-NODE twice.

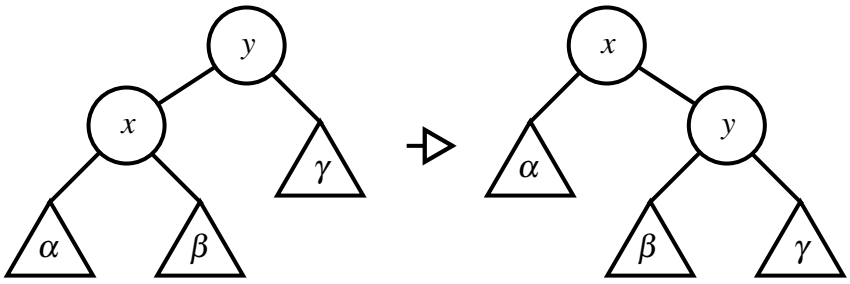

Figure 5: Right-rotation of y

RIGHT-ROTATION(T, y)

1 (Do right-rotation on node y in tree T)
2 FIX-MAX-NODE(y)
3 FIX-MAX-NODE(x)

Note that RIGHT-ROTATION takes $O(1)$. It follows the performance of TREE-INSERT still remains $O(h)$, and GET-LMIS $O(nH)$. Since balanced trees ensure the height is $O(\lg n)$, or $H = O(\lg n)$, GET-LMIS takes $O(n \lg n)$ indeed.

Yet, this is not the end of story. We should rethink the loop invariant (1). In its proof, Initialization, case (ii) in Maintenance, and Termination are out of concern because they work well even with rotations. In contrast, the rest part, case (i) in Maintenance doesn't. Thus, we need to introduce another proof. One counter example for the old proof is on the part "$LMIS'_t$". Suppose there was no rotation on x and y in the left one in Fig 5. As adding a new node is done at a leaf only, y precede x in the input. When a right-rotation is executed on y, y becomes the right child of x. The old proof utilizes this point in which x must precede y in the input, so proof is broken.

Instead of the old proof, here's another version for case (i) in Maintenance in (1). It's trickier but works.

Proof. First of all, we define a terminology. Given two nodes x and y in a tree T, x is on the left of y if

$$\exists\, z \in T \quad \left[x \in z \cup T_{z.\,left}\right] \wedge \left[y \in z \cup T_{z.\,right}\right]$$

and is represented by $x \diamondsuit y$. Let *middle node* denote the node z above. For instance, in Fig 2, (node 6) $\diamondsuit$ (node 7) by taking the middle node 6. Note that $\diamondsuit$ is transitive.

$$x \diamondsuit y \;\wedge\; y \diamondsuit z \quad \rightarrow \quad x \diamondsuit z$$

Proof. Let $\alpha \sim \beta$ denote the simple path between arbitrary nodes α and β. Observe that the middle node between α and β, $m_{\alpha\beta}$, is the node whose height is maximum on $\alpha \sim \beta$. Suppose the height of m_{xy} is less than equal to that of m_{yz}. Then m_{xy} should be on $y \sim z$. Otherwise, $m_{xy}.parent = \text{NIL}$ but $m_{xy} \neq T.root$, a contradiction. Hence $y \sim z$ is split into $y \sim m_{xy}$, $m_{xy} \sim m_{yz}$, and $m_{yz} \sim z$. Linking $x \sim m_{xy}$ to $m_{xy} \sim m_{yz}$, and $m_{yz} \sim z$ gives us $x \sim z$ on which the maximum-height node is m_{yz}. We know $m_{yz} \cup T_{m_{yz}.\,right}$ contains z. We're also aware that $m_{yz} \cup T_{m_{yz}.\,left}$ contains m_{xy}, and $T_{m_{xy}}$ does x. Therefore, $x \diamondsuit z$. □

To set a boundary for every node in $LMIS'_r$ for any r, let us introduce a corollary.

$$\forall\, x \quad x.prev\text{-}node \diamondsuit x$$

This is true because we set $x.prev\text{-}node$ using only the node on the left of x. This relationship also remains true when a node goes through a rotation(proof is simple, so skipped here). By taking transitivity and the above corollary, we know

$$\forall\, y \in LMIS'_x \quad y \diamondsuit x \tag{2}$$

Remember that every node in $LMIS'_x$ appears when we follow through *prev-node* from x recursively.

Now we're ready to prove the case (i) in (1). Similar to the original method, we show $LMIS'_{prev}$, $LMIS'_t$, and $LMIS'_{t.left.max\text{-}node}$ satisfy (1) respectively when t becomes $t.right$.

The old proof for $LMIS'_{prev}$ doesn't get bothered by rotation. Hence we use it again.

Both $LMIS'_t$, and $LMIS'_{t.left.max\text{-}node}$ satisfies the first condition of (1) by means of

$$t.left.max\text{-}node.key \leq t.key \leq k$$

Also, this is true.

$$\forall\, x \in T_{t.right} \quad t \diamondsuit x$$

using t as the middle node. Furthermore, it follows $t.left.max\text{-}node \diamondsuit t$. Considering all together with (2), each node in the $LMIS'_t$, and $LMIS'_{t.left.max\text{-}node}$ is on the left of every node in $T_{t.right}$. Because t is not in $T_{t.right}$, the second condition of (1) is true.

In summary, no matter what variable is chosen at lines 7-10 as *prev*, (1) is true at the end of iteration even with rotations. □

References

[THCS09] Ronald Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithm*, chapter 15, page 397. MIT Press, 3 edition, 2009.