Assignment 3, 6.006, Introduction to Algorithms

Bryan Zhang

May 9, 2018

1 Problem 3-1. Range Queries

1.1 a

Anwsers: 4

The range Queries requires that the data has sub-linear running time for inserting a value and getting minimum and maximum.

Thus **Min-heap** and **Max-heap** is not suitable because they can only return one extreme of the ranging ends in constant time but can not return the other one without futhur enhancement.

Binary Search Tree does all operation in $O(h)$ time, in which h is height of the search tree.But We don't know the BST is balanced or not. h is between $\log(n)$ and n. So BST is not working here.

AVL trees is a nearly balanced BST. So it works.

B-trees insert at $O(\log(n))$. But the every node may contains more than one key. Thus is not suitable for range queries.

1.2 b

Answer: 3

1.3 c

Answer: 3

1.4 d

Answer: 3

1.5 e

Answer: 6

The " + 1" is to compensate for the l key.

1.6 f

Answer: 5

1.7 g

Answer: 6

1.8 h

Answer: 5

1.9 i

Answer: 4

Choice 1, 2, 3, 6 are doing nothing to find out the rank of the node. And the rank of the node can not be given its subtree and its property only. The number of nodes in the subtree, or the subtree size, can be give the sum of subtree sizes of the node's children and plus itself.We can use compute the rank.

This is the pseudo-code for the Rank():

Algorithm 1 Rank()

```
1: function RANK(tree, k)        ▷ Where tree - the AVL tree, k - the key to find its rank
2:     rank = 0
3:     node = tree.root                                  ▷ start from the root to find k
4:     while node ≠ NIL do
5:         if k < node.key then
6:             node = node.left                          ▷ thus rank is not updated
7:         else
8:             if node.left ≠ NIL then
9:                 rank = rank + 1 + node.left.γ
10:            else
11:                rank = rank + 1
12:            end if
13:            if node.key == k then
14:                return rank
15:            else
16:                node = node.right
17:            end if
18:        end if
19:    end while
20:    return rank
21: end function
```

1.10 j

Answer: 3

The number to store is maximum N. In bits, $\log(N)$.

1.11 k

Answer: 1

No children for N_4.

1.12 l

$N_3.\gamma = 3$.

1.13 m

Answer: 4

$N_2.\gamma = 6$.

1.14 n

Answer: 4

$N_1.\gamma = 10$.

1.15 o

Answer: 3, 4, 5

We only need to modify **ROTATE-LEFT**, **ROTATE-RIGHT**, **REBALANCE**, all of which call the update_height function.

1.16 p

Answer: 3

Since the $Rank(tree, k)$ algorithm and find method of AVL tree run at $O(\log(n))$ time, the count algorithm also runs at $O(\log(n))$.

The Count algorithm is below:

Algorithm 2 Count()

```
1: function COUNT(tree, l, h)
2:     if tree.find(l) == 0 then
3:         return Rank(tree, h) - Rank(tree, l)
4:     else
5:         return Rank(tree, h) - Rank(tree, l) + 1
6:     end if
7: end function
```

1.17 q

Answer: 2

We can see the node is always trying to get closer to the lower bound. Once the node hit the range, it is returned. So it will be the **lowest common ancestor** .

1.18 r

Answer: 3

The worst case is that the l and h is two neighbor leaves and their common ancestors is just one layer above them. So the running time is $O(\log(n) - 1)$, which equals $O(\log(n))$.

1.19 s

Answer: 8 I chose the 6 incorrectly. Let V be number of nodes that the NODE-LIST algorithm visits. So its running time is $O(V)$. Let X is the number of nodes visited bu t not output by the LIST algorithm. Since $V = L + X$, our goal becomes to prove X = $O(h)$.

Our focus is the nodes visited but not in range [l, h]. But We can run through NODE-LIST to find out where they and nodes in the range [l, h] are.

The first node the program visits is the LCA. So the search goes into LCA left subtree and right subtree.

In the left subtree, all keys < LCA.key < h, since h is in the right subtree of LCA. In this case, We only need to care about the lower bound. If the key of the node < l, then line 5 will prune the left subtree of the node, and going to right subtree of the node. In this case, the algorithm just follow a path to l. Even if l is not in the tree, the path is going to the nearest node that are greater or equal to l.

In the right subtree of the LCA, we can have the same result. Thus we have proved that X is smaller than but at the scale of the height of the tree. Since the BST is an AVL, The total running time is $O(L + \log(N))$.

1.20 s

Answer: 8

The line 2 takes $O(\log(N))$ and line 3 takes $O(L + \log(N))$.

1.21 u

Solution: We will prove LCA is correct by two stage: l and h is in the subtree of the every node LCA visits and the returned node is LCA. LCA is defined as the root of the smallest subtree that contains l and h. If l and/or h is not in the tree, we will augment the tree with imaginary nodes created by inserting l and h into the tree but not re-balancing the tree.

LCA maintains the invariant that l and h belong to the subtree that is rooted at the node. LCA starts at the root of the tree. So the invariant is true at the beginning. Because the loop will continues under two conditions: either node.key < l or node.key > h.If node.key < l , then l, h must be located in the right subtree of the node, which keep the invariant by line 6. In the same reason, the invariant is also kept in line 3 if the node.key > h.

LCA returns node that is the root of the smallest subtree that contains l and h. We prove this point by contradiction. If r is node returned by LCA and c is the root of the smallest subtree containing l and h.We assume $c \neq r$. l and h must be in the subtree of the node r. Since $c \neq r$, then c must be in the subtree of the node r too. If c is in the left subtree of r, then by BST invariant, l and h < r.key. If this is the case, r will not be returned. Similarly, when c is in the right subtree of node r, r will also not be returned. This contradicts with our assumption that r is the node returned by LCA. q.e.d.