CSC263: Assignment 2

Jingqi Zhang, Yulin WANG, Tianjing Ruan

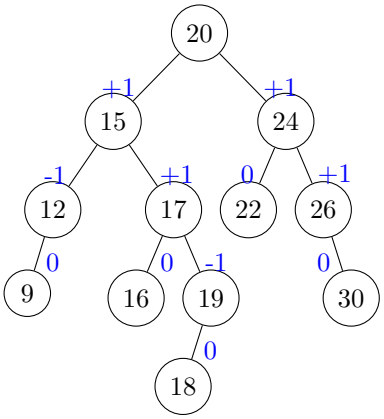
Due Date: *06-10-2019*

Question	Written By	Checked By
1	Tianjing Ruan	Yulin WANG
2	Tianjing Ruan	Jingqi Zhang
3	Yulin WANG	Jingqi Zhang
4	Jingqi Zhang	Yulin WANG

Question 1

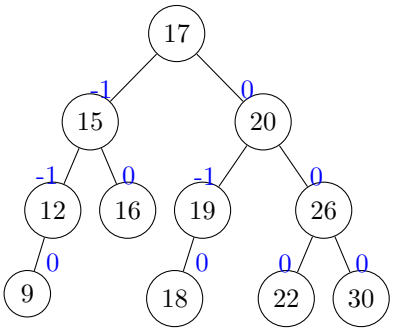
a)

The BF of the root is -1.

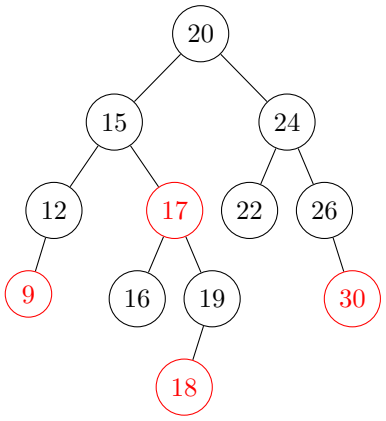


b)

The BF of the root is 0.



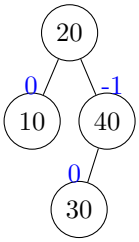
c)



Question 2

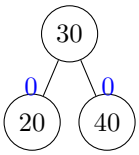
a)

Disprove:
example:
T: The BF of the root is +1.



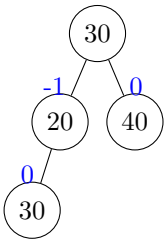
We delete 10 from T

T': The BF of the root is 0.



We add 10 back to T'

T": The BF of the root is -1.

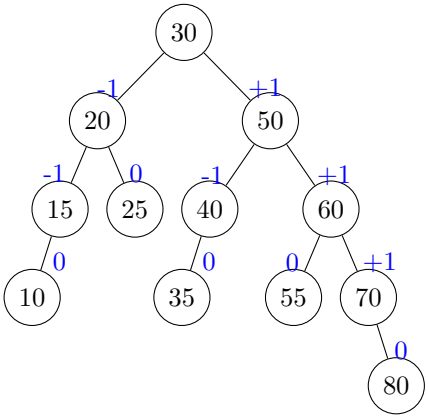


T" $\neq$ T

Therefore, the statement is false.

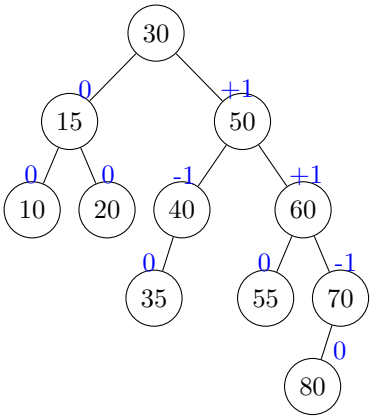
b)

T: The BF of the root is +1.



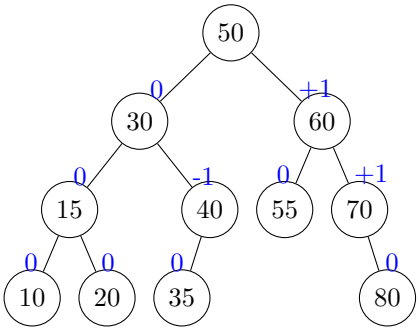
We preform DELETE(T, 25) to T

After first rotation, T': The BF of the root is +2.



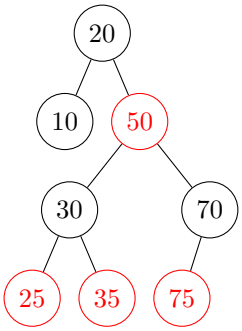
Then we apply the second rebalancing operation to T'

So the result of rebalancing operations is T": The BF of the root is 0.



c)

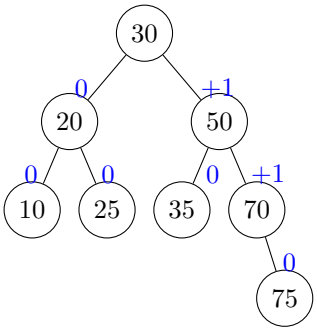
Disprove:
example:
If we insert 20, 10, 50, 30, 70, 25, 35, 75 into an initially empty red-black tree, the result will be:



We can notice that the red black tree is not height balanced

However, If we insert 20, 10, 50, 30, 70, 25, 35, 75 into an initially empty AVL tree, the result will be:

The BF of the root is +1.



Therefore, the statement that "any red-black tree is also AVL Tree, that is that every red-black tree is height balanced" is false.

Question 3

Given any node u of a BST T, procedure isBalanced(u) does the following things:

1.If the subtree rooted at u is balanced then it returns the height of the subtree.

2.If the subtree rooted at u is unbalanced then it returns -1.

Below is the pseudo-code.

isBalanced(node u)
if u = NIL then return 0
else:
    H_L = isBalanced(u.left)
    H_R = isBalanced(u.right)
    if H_L = -1 or H_R = -1 or $|H_L - H_R| > 1$
        then return -1
        else return max(H_L + H_R) + 1

If isBalanced(root) > 0 then return True, otherwise return False.

Explain.
Start from the root, travel all the way down to leaf nodes and then go up. While going up, calculate the left and right subtree height. If the difference between them is greater than 1, return -1, else return Max(leftHeight, rightHeight) +1.

The algorithm recursively visits every node of the tree T exactly once, and for each node, it takes constant time to execute the if...then...else part of the algorithm. That is, updating on each node takes constant time. And since there are n nodes in T, so we update at most n times, thus the worst-case running time is $\Theta(n)$.

Question 4

The data structure for this algorithm I choose is Max Heap. It works as 2 parts:

1.BuildUp our Max Heap M1 :

-insert each of the first m input keys into M1 (using the Insert op of Max Heap).

Now our M1 is with heap size of m, containing the current m smallest keys since there are only m keys having been input. And by assumption query will not occurs in the first m keys input, so we don't need to care about performing that.

2.Process each input x:

•if x is a key input (x is an integer):

-insert this key x into M1(using Insert op of Max Heap);

-remove the maximum key of M1 (using the ExtractMax op of Max Heap);

After each time process, we guarantee out M1 keeping size m (since +1-1 = 0) and conatins the current m smallest keys amount all input before the next.

• if x is a query operation(x == "query"):

Since all output is actually keys in M1 we stored, now we sort the order.

-build up a new array B of size m;

-get a copy M2 of M1 (to keep our current Max Heap from any change by below).

-get the maximum from M2 (using the ExtractMax op of Max Heap) and store it into the array B, until M2 is empty (that is actually m times).

Now array B contains values what we need and from max to min key order. So

-linearly print out the element in B as an reversed order (from B[m-1] to B[0]).

Below is the pseudo-code.

Algorithm(x, m)

```
BuildUpMaxHeap(M1);
IF M1.size < m:
    MaxHeap.Insert(M1, x);
IF x == INT:
    MaxHeap.Insert(M1,x);
    MaxHeap.ExtractMax(M1);
IF x == "query":
    B[0:m] ← NIL;
    Index ← 0;
    M2 ← M1.COPY();                          #linearly copy m keys, O(m).
    WHILE Index < m:                         #loop m times
          max ← MaxHeap.ExtractMax(M2);
          B[Index] ← max;
          Index ++;
    WHILE Index <= 0;                        # loop m times
          Print(B[Index]);
          Index −−;
```

Now I explain the time complexity.

• When process each key input, either that the key input is the very first m times input (which is BuildUp part, using one Insert operation, O(log m)), or after the first m input(which is Processing part, using one Insert and Extract-Max operation, O(log m) + O(1) that is O(log m)), we cost O(log m) as code above noted.

•When perform query operation, except the constant time, we did a copy (O(m)), ExtractMax operation for m times (m * O(1) in total) , print output for m keys (m * O(1) in total), so we cost O(m)as code above noted.