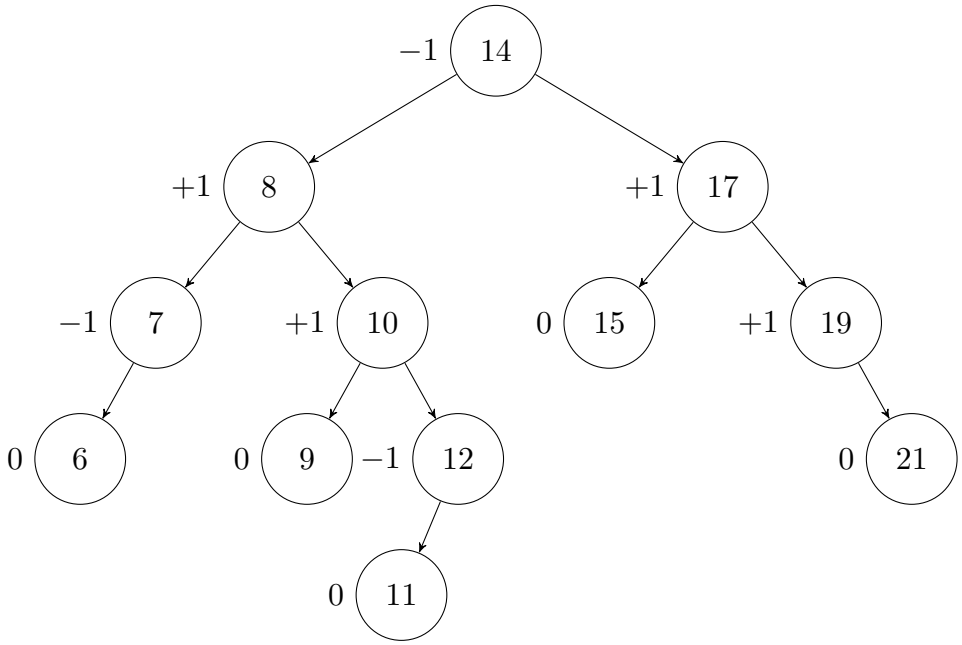
CSC263 Assignment 2

Yueen Ma, Zikun Chen, Yue Li

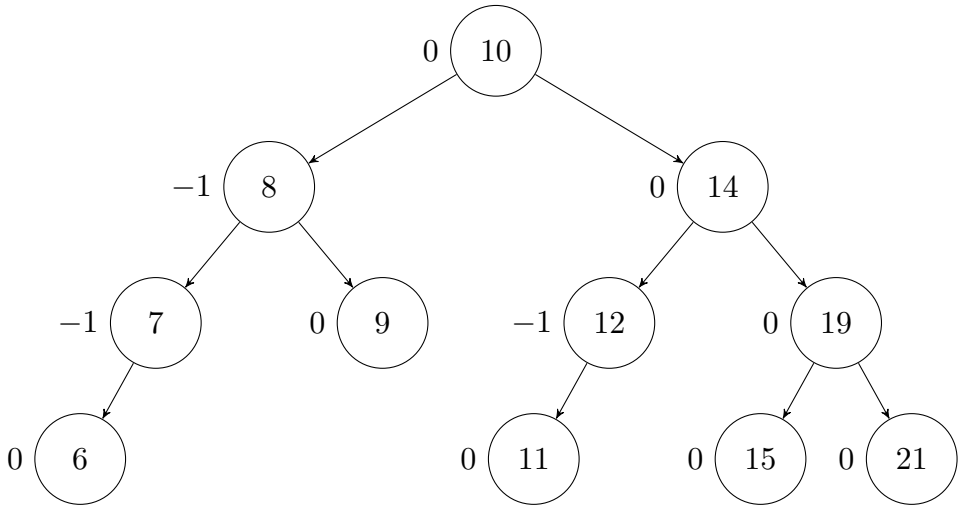
September 7, 2018

Question 1

(Written by Yueen Ma, Read by Zikun Chen, Yue Li)
a.



b.

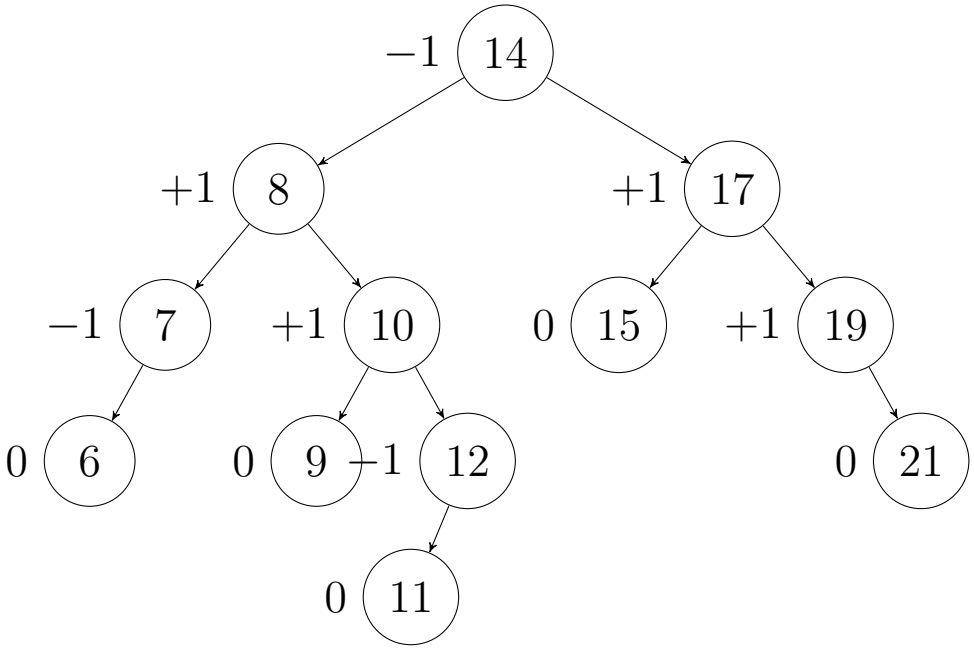


Question 2

(Written by Yueen Ma, Read by Zikun Chen, Yue Li)

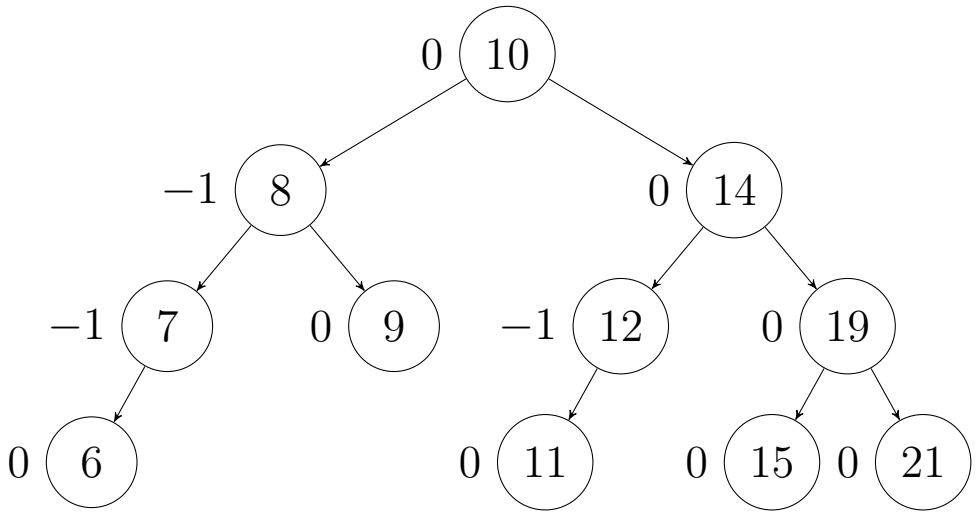
a. Disprove:

$T =$



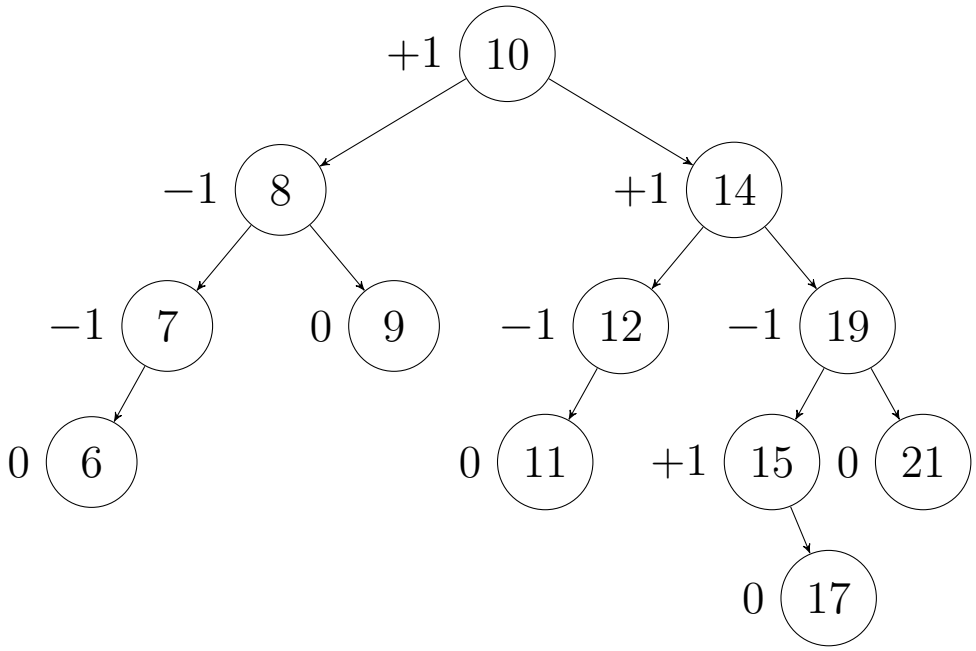
After DELETE$(T, 17)$, we get:

$T' =$



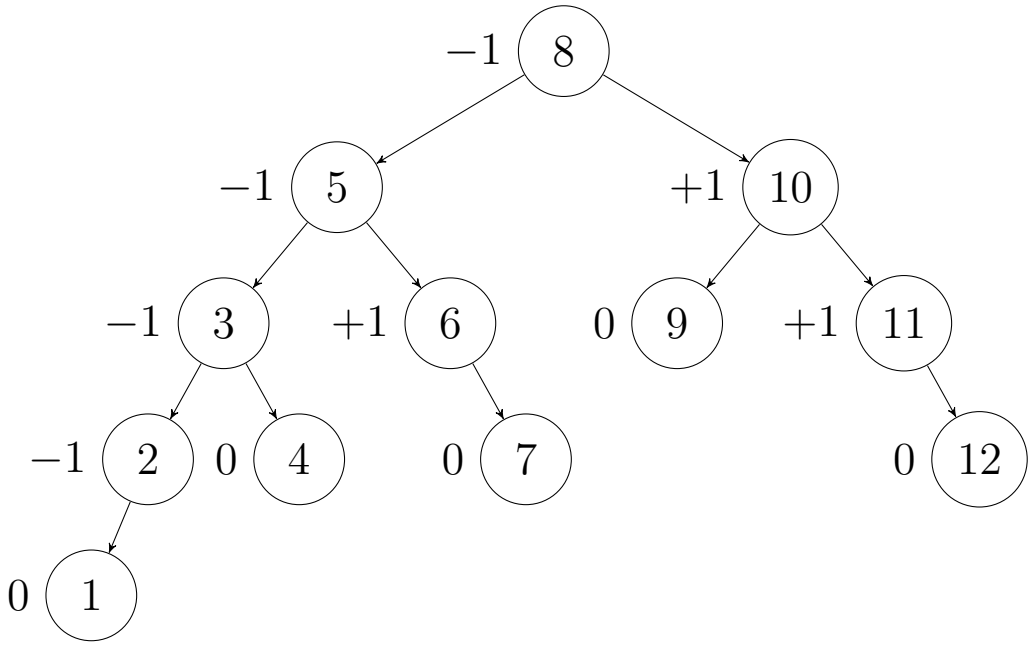
After INSERT$(T', 17)$, we get:

$T'' =$

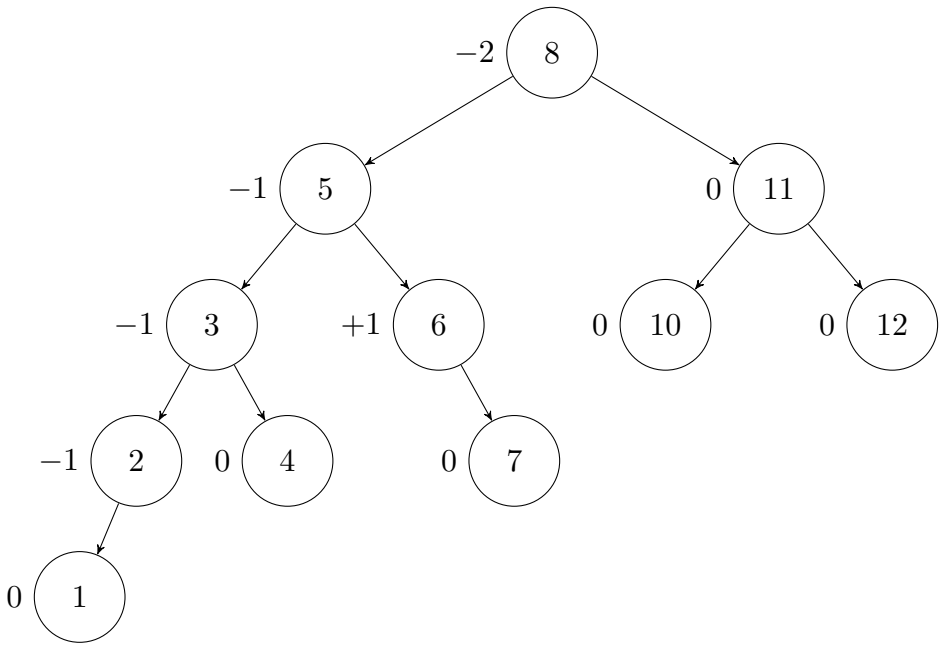


b.

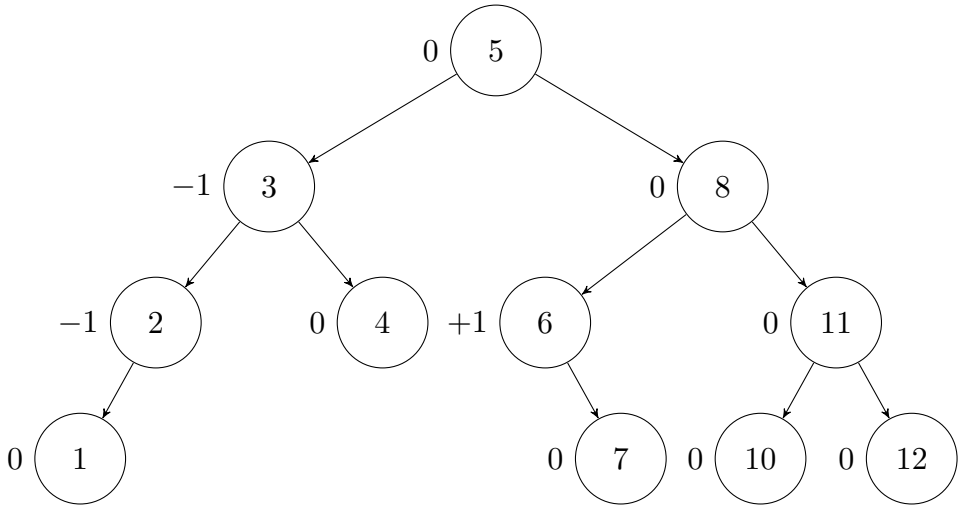
$T =$



We do DELETE(T, 9)
Result of the first rebalancing(left rotation at the node with key '10'):



Result of the second rebalancing(right rotation at the root):



Question 3

(Written by Yue Li, Read by Zikun Chen, Yueen Ma)

```
# CheckHeight is a helper function:
# It returns the height of tree at root if input is a balanced AVL tree,
# and returns -2 otherwise.
def CheckHeight(root):
    if root == Nil:
            return -1

    left_height = CheckHeight(root.lchild)
    if left_height == -2: return -2

    right_height = CheckHeight(root.rchild)
    if right_height == -2: return -2

    //Check absolute value of balance factor
    if absolute_value(left_height - right_height) > 1:
        return -2

    return 1 + max(left_height, right_height)

def CheckBalance(root):
    if CheckHeight(root) != -2:
        return True
    return False
```

Explanation:

- **Algorithm Idea:**

  CheckHeight takes pointer of root of a binary tee, and recursively compute the height of node's left child and right child, and then check balance factor of current node.

  If absolute value of balance factor is greater than 1 for a particular node, program will continue returning -2 until pointer reaches the root of tree, and finally returns -2 to CheckBalance, the program returns False.

  Otherwise, each recursive call will return the height of each subtree rooted at the left child and right child and finally CheckHeight returns the height of the whole tree at root.

  Our program CheckBalance will return False if CheckHeight returns -2, indicating that at least one of nodes in given binary tree has unacceptable balance factor, return True if CheckHeight returns value other than -2.

- **Time Complexity:**
  Let T(n) be the worst-case running time of CheckHeight, which is the worst-case runtime of CheckBalance differ only by a constant. So their big-O complexities are the same.

  To prove T(n) is Θ (n), need to prove both O(n) and Ω(n)

  Prove O(n):
  For worst case, assume every line of code will be executed.
  Note except the recursive call in the function CheckHeight, the other lines have constant run time O(1), say they take c_1 steps.
  Let n denoted the number of nodes in given tree. Let constant c denotes the constant time cost in each function call, then:
  $T(n) = T(\lfloor \frac{n-1}{2} \rfloor) + T(\lceil \frac{n-1}{2} \rceil) + c_1$, when $n \geq 2$
  $T(1) = c_2$, when $n = 1$
  -Note:Use floor and ceiling in case that the total number of nodes of the two children is odd.

  Since each call contains two sub calls on left and right subtree, and T(n) depends on the number of nodes each tree has. Then we get time complexity $T(n)$ by unwinding as follows:

$$
\begin{aligned}
T(n) &= T(\lfloor \frac{n-1}{2} \rfloor) + T(\lceil \frac{n-1}{2} \rceil) + c_1 \\
&\leq 2T(\lceil \frac{n-1}{2} \rceil) + c_1 \\
&\leq 2^2 T(\lceil \frac{n-1-2}{2^2} \rceil) + c_1 + 2c_2 \\
&\leq 2^3 T(\lceil \frac{n-1-2-2^2}{2^3} \rceil) + c_1 + 2c_1 + 2^2 c_1 \\
&\leq ..... && \text{keep unwinding} \\
&\leq 2^n T(\lceil \frac{n-(1+2+2^2+...+2^{n-1})}{2^n} \rceil) + c_1(1+2+...+2^n) \\
&= 2^n T(\lceil \frac{n-(2^n-1)}{2^n} \rceil) + c_1(2^{n+1}-1) \\
&Let \lceil (\frac{n-(2^n-1)}{2^n}) \rceil = 1 && \text{recursively call until we reach base case} \\
&\Rightarrow n+1 = lg(n+1) && \text{lg is base 2} \\
&\Rightarrow n = lg(n+1) - 1 \\
&\Rightarrow T(n) = 2^{lg(n+1)-1} c_2 + c_1 2^{lg(n+1)} - c_1 \\
&= c_2 \frac{n+1}{2} + c_1(n+1) - c_1 \\
&= \frac{n(2c_1+c_2)+c_2}{2} \in O(n)
\end{aligned}
$$

Therefore T(n) is in O(n)

Prove $\Omega(n)$

In the worst case, consider the input as a **full binary tree**(every node other than the leaves has two children). In this case. every node has balance factor 0, which means that program cannot find any node with unacceptable balance factor.

Then program has to check height of **every substree** and compute balance factor of **every node** from leaves to root.

In this case, time cost T(n) depends on the number of nodes, which is n.

So T(n) takes at least n time. Then T(n) is in Ω(n).

Therefore programs' worst-case time complexity is in Θ (n)

Question 4

(Written by Zikun Chen, Yueen Ma, Read by Yue Li)

Data Structure:

We are using augmented AVL tree data structure (learned in class) with BST-ordering.

We are also using linked list data structure as the result to return in function QUERY at the end.

Items contained:

Augmented AVL Tree:

It is a binary tree structure, so it has individual nodes and each nodes contains references to its parent, left child and right child.

Each node stores its value. And also both the balance factor (the size of the right child minus the size of the left child) and the size of the subtree rooted at that node.

The value of a node's left child is smaller than the node's value, and the value of a node's right child is larger than the node's value (BST ordering).

It contains operations: SEARCH, INSERT, DELETE, RANK, and SELECT which all has $O(log(m))$ worst-case running time, where m is the number of nodes in the tree.

Linked List:

It is a Linked List structure, which consists of one or more nodes.

Each node has value, indicating the value this node holds. Next, indicating the next node connected to the current node.

Each list has a head, indicating the first node of the current node and a tail, indicating the last node of current linked list.

The only linked list operation we are using is INSERT, where we point the tail's next node to a new node and make the new node the new tail.

Algorithm:

PROCESS:

The function takes in a reference for the root node of the tree, the new key and the fixed number $m \geq 2$.

First of all, the algorithm checks if the size of the tree rooted at the root node is m, in which case we find out the largest node value in the tree by using SELECT operation and compare it to the value of the new key. If the new key has a smaller value than the largest node, which means that it should be one of the m smallest nodes in the sequence so far, we want to DELETE this largest node and INSERT the new key into the tree. Otherwise, we don't do anything and keep the tree as it is since the new key has a bigger value than all the m nodes already in the tree.

On the other hand, where the size of the tree rooted at the root node is less than m, we simply INSERT the new key into the tree.

This way, we can keep the size of the tree to be fixed at m while we add potential infinitely more new keys one at a time and the node values in the tree will be the smallest m because it is updated when new keys are received.

QUERY:

The function takes in the reference of the root node.

We simply do a recursive in-order traversal of the tree rooted at root and store each value in a linked list, which we return at the end.

We want to concatenate the linked-list returned by the QUERY operation on the left child with the root value and the linked list returned by the QUERY operation on the right child using

the helper function LINK, which takes constant time to concatenate two linked lists by changing a few references of the head, tail and next node of the tail.

This way, the values in the resulting linked list will be in sorted order from the smallest to the largest.

Pseudo Code:

```
# root is a pointer to the root node of the AVL tree
PROCESS(root, key, m):
    if (root.size == m):
        largest = root.SELECT(m)
        if (largest > key):
            root.DELETE(largest)
            root.INSERT(key)
    else: root.INSERT(key)

# Pre-condition: QUERY called after root.size reached m
QUERY(root):
    #in-order traversal
    result = empty linked list
    if (root != NIL):
        result = LINK(result, QUERY(root.left))
        result.INSERT(root.value) # linked-list INSERT
        result = LINK(result, QUERY(root.right))
    return result

# Helper funciton: Link two linked list L1 and L2 together.
# L1 and L2 are two linked lists
LINK(L1, L2):
    if (L1 = NIL): return L2
    if (L2 = NIL): return L1
    else:
        L1.tail.next = L2.head
        L1.tail = L2.tail
        return L1
```

Run Time Analysis:

- $O(logm)$ to process each input key:
  Each PROCESS(root, key , m) requires:
  1. if (root.size == m):
     (a) one SELECT(m), which takes $O(logm)$ time in the worst case, and
     (b) at most one DELETE(largest), which takes $O(logm)$ time in the worst case, and
     (c) at most one INSERT(key), which takes $O(logm)$ time in the worst case
     Thus the worst-case time complexity for the case "root.size == m" is $O(logm)$
  2. else: (root.size != m)
     one INSERT(key), which takes $O(logm)$ time in the worst case,
     Thus the worst-case time complexity for the case "root.size != m" is $O(logm)$

So the worst-case time complexity of PROCESS(root, key , m) is: $O(logm)$

- $O(m)$ to perform each QUERY(root):
  Let $T(n)$ denote the time cost of QUERY($Node$), with the size of the tree rooted at $Node$ being n for any natural number n.
  The helper function LINK(L1, L2) is O(1) because linking two linked list only involves changing references of head and tail, which are constant operations.
  Linked list's INSERT(x) is also O(1) because we simply point tail's next node to the new node x, which is a constant operation.
  Therefore, in the recursive algorithm, except QUERY(root.left) and QUERY(root.left), the other lines have constant runtime, say c_1 operations.

  - Note below, the floor and ceiling accounts for the case when the total number of nodes of the two children is odd.

$$
\begin{aligned}
T(m) &= T(\lfloor \frac{m-1}{2} \rfloor) + T(\lceil \frac{m-1}{2} \rceil) + c_1 \\
&\leq 2T(\lceil \frac{m-1}{2} \rceil) + c_1 \\
&\leq 2^2 T(\lceil \frac{\lceil \frac{m-1}{2} \rceil - 1}{2} \rceil) + (2^1 + 1)c_1 \\
&\leq 2^3 T(\lceil \frac{\lceil \frac{\lceil \frac{m-1}{2} \rceil - 1}{2} \rceil - 1}{2} \rceil) + (2^2 + 2^1 + 1)c_1 \\
&\cdots\cdots \\
&\leq 2^{log(m)} T(\lceil 1 \rceil) + (2^{log(m-1)} + \cdots + 2^1 + 1)c_1 \text{ (log is base 2)}
\end{aligned}
$$

  when we reach T(1), it means that we have already traversed all log(m) layers of the tree
  so, the inequality stops here

$$
\begin{aligned}
&= 2^{log(m)} T(1) + (2^{log(m-1)} + \cdots + 2^1 + 1)c_1 \\
&= 2^{log(m)} T(1) + \frac{1 - 2^{log(m-1)+1}}{1-2} c_1 \text{ (by geometric series } r = 2) \\
&= 2^{log(m)} T(1) + (2^{log(m-1)+1} - 1)c_1 \\
&\leq 2^{log(m)} c_2 + 2^{log(m)} 2c_1
\end{aligned}
$$

  since linked-list INSERT and helper function LINK have constant run time so T(1) takes time O(1)

$$
\begin{aligned}
&= (c_2 + 2c_1)2^{log(m)} \\
&= (c_2 + 2c_1)m
\end{aligned}
$$

  So the worst-case time complexity of QUERY($Node$) is: $O(m)$