Assignment 6

Fu Guanshujie, Ge Yuhao, Lou Haina, Qiu Xiaomin

March 2021

1

See the code in attached files.

2

For the original delete operation, the delete and merge only take constant time, and the most complex part is to find the place of a node through a value which is in $O(logn)$. However, for the delete with reinsert, it needs to conduct an insertion for every nodes beneath the node to be deleted, and each insertion is in $O(logn)$. The number of nodes beneath one node is proportional to n, so the general complexity is in $O(nlogn)$.

3

3.1 Insert

Firstly, we compare each node value with the inserted value. if the inserted value is less than node value, turn to left successor node. if the inserted value is more than node value, turn to right successor node. Keep track of all nodes on the path from the root and put the nodes onto stack with balance information. If the checking left or right successor become NULL, stop. Then we add the node to that position and update the balance values on the stack. If balance >= 1 or <=-1, do rotation operation. If we detect balance to be 0 from front to back of stack, we can stop. The core part of **Insert** code is:

Algorithm 1 INSERT

Input: $*pt\ avilnode < T >$
$Tval$
Output: AVL after insert

```
1:  function INSERT(i)
2:      while *pt! = NULL do
3:          if val > (*pt).value then
4:              *temp ← *pt
5:              *pt ← *pt.getright()
6:          else
7:              temp ← *pt
8:              *pt ← *pt.getleft()
9:          end if
10:     end while
11:     if value > *temp.value then
12:         *temp.setright(new_node)
13:     else
14:         *temp.setleft(new_node)
15:     end if
16: end function
```

Then do some operation for rotation as template after checking balance.

3.2 Delete

We do the same method as insert: compare and find the node, push them onto the stack with potential information about balance. Once the node is found, we delete it. If the deleted node has node child, we then update balance and do rotating if necessary. If the node has one child then swap the child into the deleted position. If it has two children, swap the maximum in the left successor tree or minimum in the right successor tree and do rotate if necessary. **Delete**:

Algorithm 2 Delete

Input: $*pt\ avilnode < T >$
$Tval$
Output: AVL after delete

```
1: function DELETE(i)
2:     while *pt! = NULL do
3:         if val == (*pt).value then
4:             delete(*pt)
5:             Return
6:         end if
7:         if value < *temp.value then
8:             *pt ← *pt.getleft()
9:         else
10:            *pt ← *pt.getright()
11:        end if
12:    end while
13: end function
```

Delete operation is to delete the node, and we need to check balance and rotate the operation.

Difference:
The main difference between iterative and recursive method is that iterative uses stack to keep track of nodes and store balance. The stack has space complexity of $O(\log n)$. They all relate to the height of AVL tree, with time complexity of $O(\log n)$. But recursive method is more effective processing with mass data.

4

4.1

Consider an AVL tree with n nodes has median property. If $n = 2k$ which indicates that n is even, then the left subtree should contain $\frac{n}{2}$ nodes while the right subtree contains $\frac{n}{2} - 1$ nodes. With nodes in the AVL tree satisfying this property, all nodes' left and right subtrees should also maintain above relation and hence the tree is must balanced. If $n = 2k + 1$ which indicates n is odd, the left and right subtrees should contain same number of nodes and all nodes should also satisfying such property. Clearly, the tree should also be balanced.

4.2 Insertion

Let's define the *median* of an AVL tree node as m, the element to be inserted as x, the number of nodes in left and right subtree of a node as $Number(l)$ and $Number(r)$. As the AVL tree will maintain the *median* property after insertion, there should exist four cases for insertion.
1. $Number(l) = Number(r)$ **and** $x < m$**.** In this case, the new element will go to the left subtree. Clearly, after insertion the left tree will contain one more element than right tree with new median $m' = \frac{x+x'}{2}$, x' represents the maximum element in the left subtree.
2. $Number(l) = Number(r)$ **and** $x > m$**.** In this case, the new element will go to the right subtree. After insertion, the median property will be broken and we need to adjust the nodes in the right subtree by searching for minimum element in the right tree and set it as the new root. The original root will be inserted into the left subtree and the new median will be $m' = \frac{x+x'}{2}$.
3. $Number(l)$ **-** $Number(r)$ **= 1 and** $x > m$**.** In this case, the left subtree contains one more element than the right subtree and the new element will go to the right subtree. By insertion, the new median will be the value of

the current root.
4. $Number(l)$ **-** $Number(r)$ **= 1 and** $x < m$**.** In this case, the left subtree contains one more element than the right subtree and the new element will go to the left subtree. By insertion, the left subtree will then contains two more elements than right and adjustment is needed. After insertion, we search for the maximum element in the left subtree and set it as the new root. The original root will be inserted into the right subtree while the new median m' will be the value of the new root.

4.3 Deletion

Similar to the insertion, the deletion also has four cases to be considered. Let's define the *median* of an AVL tree node as m, the element to be deleted as x.
1. $Number(l) = Number(r)$ **and** $x < m$**.** In this case, the element will be deleted from the left subtree. Clearly, after deletion the right tree will contain one more element than left tree. Adjustment is needed to maintain the median property by searching for the minimum element element in right subtree and set it as the new root. The original root will be inserted into the left subtree.
2. $Number(l) = Number(r)$ **and** $x > m$**.** In this case, the new element will be deleted from the right subtree. After deletion, the median property still holds with the new median be $m' = \frac{x+x'}{2}$.
3. $Number(l)$ **-** $Number(r)$ **= 1 and** $x > m$**.** In this case, the left subtree contains one more element than the right subtree and the element will be deleted from the right subtree. Adjustment is needed by searching for the maxmum element in the left tree and set it as the new root. By deletion, the new median will be the value of the current root.
4. $Number(l)$ **-** $Number(r)$ **= 1 and** $x < m$**.** In this case, the left subtree contains one more element than the right subtree and the element will be deleted from the left subtree. The median property holds after deletion and the new median will be the current root.

4.4 Worst Case Complexity

Let's consider the worst case complexity of insertion as both deletion and insertion works in a similar way. Assume the complexity of the inseriton as $P(n)$. Clearly, the "insert" part has complecity of $O(\log(n))$. The real work is keeping all nodes to maintain median property. We can derive the following recursive equation:

$$P(n) = 2P(n) + O(\log(n))$$

Obviously, $P(n)$ should be in $O(n)$. Same for deletion.