CS225 Assignment5 | Group 10

-- *Exercise 1* --

Please refer to folder 'ex1' for code implementations.

-- *Exercise 2* --

Q2.2

In general, this idea should not be recommended since the complexity is

much larger than the original idea.

For the original $delete$ operation, whatever the condition is, the complexity

is always $O(\log_2 n)$. First, we need $O(h)$ steps to find the node to delete at

height h. Then, we have two conditions:

1. If the right successor has a non-empty left successor tree, then we need to

   merge this tree with left successor tree. For the worst case, we need to search

   k times, where k is the height of this non-empty left successor tree. So

   the complexity is in $O(k)$. The overall complexity is $O(h+k) = O(\log_2 n)$.

2. For other cases, we need constant time to transplant new subtree into the place

   where the deleted node occupies, the overall complexity is just $O(h)$.

So the complexity of $delete$ operation is $O(log_2 n)$

But for the $delete\_with\_reinsert$ operation, the situation becomes different. First, we still need $O(h)$ steps to find the node to delete at height h. Then, if the number of nodes in the left and right subtrees is m, we need to reinsert them one by one. Luckily all the nodes in the subtrees are bigger or smaller than the parent of

the deleted node, so we can just insert starting from that parent node.

For the worst case, we reinsert all the nodes linearly, then the complexity is in $O(m^2)$.

Even for average case when the tree after reinsertion is almost balanced, the time complexity is still in $O(m)$ for building a binary tree. So the overall complexity is either $O(h+m^2)$ or $O(h+m)$, which is greater than $O(h+k) = O(\log_2 n)$.

Besides, we have not taken the $free$ operations for the nodes when reinserting and

the space complexity for creating new nodes into consideration.

So considering the time complexity and space complexity, $delete\_with\_reinsert$

operation is not recommended.

-- *Exercise 3* --

Basis For both operations -- A Stack

In the iterative cases, a stack must be manually maintained, which is automatically done by compiler in recursive cases. Thus, we have to initiate an empty stack which keeps record of nodes visited during locating the position for the new node to be inserted. The information stored in this stack allows us to update the balances for nodes affected and execute rotations when necessary during **AVL::Insert** and **AVL::Delete**. Specifically, the elements to be stored in the stack will be comprised of elements of two types:

Information	Type	Value
Node pointer	Node*	A pointer to the node being visited
Direction	bool	left(false) / right(true)

We can use a ***struct*** or c++ STL ***pair*** to pack the above two elements into one single element that can be stored in our stack.

The basic framework for both operations will be the same as the provided recursive implementations. The operations will be sketched below.

An Iterative Implementation of AVL::Insert

> Algorithm Description
>
> - Declaration:
>
> ```
> Node* AVL::Insert(Node* node, int data);
> ```

1. Initiate the empty stack as described above
2. initiate relevant variables
3. If *node* (pointer to the root) is NULL, the tree is empty. Hence we malloc in the memory a ***newnode***, set its value to ***data***. And make this new node the ***root*** by returning ***newnode***.

   'This leaves us with the case where the tree is not empty.'
4. **Location Finding**

   Now traverse down the tree to locate the location for inserting the newnode. In each iteration, we push into the stack information containing the node visited currently and the direction of the next traversal (whether we are going to visit a left child or a right child). This allows us to step back to update balances and adjust trees after the actual insertion. During iterations, several cases are considered:

   1. **current_node.getdata() == data**

      If we encountered the data to be inserted, clean the memory and exit the program since nothing has to be done.
   2. current_node.getdata() > data

      we go to left successor tree for next iteration; push relevant information in the stack
   3. current_node.getdata() < data

      we go to right successor tree for next iteration; push relevant information in the stack
   4. **current_node == NULL**

A location for the insertion has been found. End the iteration and move on to balance update and tree adjustments.

5. **Insertion**

   Insert the node at the location found. if the balance of the parent node of the newly inserted node is not affected, the insertion is done and we can exit the operation here. If not, go to step 6.

6. **Balance Update and Tree Adjustment**

   This part we use another loop structure to implement. The ending condition of this loop is

   ```
   path.empty() == true;
   ```

During iterations of this step, we will need 4 variables,

```
Node* bchild, Node* bgrandchild, Node* current_node, int newbal
```

where

current_node is currently being examined, it always equal to $path.pop() \rightarrow first$

bchild is the value of current_node in the previous iteration

bgrandchild is the value of current_node in the penultimate iteration

newbal is the new balance to be given to current_node (if valid)

We calculate newbal based on the original balance of current_node and the direction information from the stack.

- If newbal is valid ($-1 \leq newbal \leq 1$), we set

```
current_node->balance = newbal;
```

and step to the next iteration

- If newbal is not valid, **single rotations or double rotations** are executed at the local subtree structure.
  - If the balance of the root of the local subtree after rotation does not change, **the operation is done**.
  - Otherwise we step to the next iteration.

7. Finally we return the root node (since it might be modified by rotations) of the tree structure and exit.

Description for An Iterative Insertion Ends Here

Note: You can find a C++ Implementation of iterative Insert in the appendix.

An Iterative Implementation of AVL::Delete

1. Initiate the empty stack as described above.
2. **Locate the node to be Deleted**

   Use a loop to find the location of the node to be deleted. During each iteration we push the node and direction into the stack as described in Insert.

The loop terminates on two iterations

1. `current_node == NULL;`
2. `current_node->data == data;`

For case 1, the element to be deleted is not in the tree, return error and terminate the operation. (the case of an empty tree is handled is this case)

For case 2, we move on to the next step.

else the iteration is continued.

3. **Find and Swap**

   If $current\_node$ has no left/right child, replace $current\_node$ by its right/left child and do balance update and tree adjustments as we have done in Insert. **Then exit the operation.**

   This leaves the case where both left and right successor trees are not empty.

   Now we use another iteration to find the maximum in the left successor tree and swap this maximum with the node to be deleted.

   - First initiate another stack that will keep record of the path between the node to be deleted and the maximum found.
   - Iterate to find the right end (maximum) of the left successor subtree. Push relevant information in the stack in each iteration.
   - Swap the maximum with the node to be deleted, and delete the node.
   - Using the stack information for balance update and tree adjustment.

4. **Tree Maintenance**

   Use another iteration maintain the tree structure by information stored in the stack (the stack used when locating the node to be deleted).
   Since the steps are similar to Insert, we omit details here.

Description for An Iterative Deletion Ends Here

Differences between An Iterative approach and A Recursive approach.

Both approaches will have time complexity in $O(\log(n))$

The major difference lies in spatial complexity of the two approaches. While recursion takes $O(\log(n))$ spatial complexity, an iterative approach takes $O(1)$ spatial complexity.

The reduction in spatial complexity and recursive function calls let iterative approaches have an performance advantage over recursive approaches.

However, recursive implementations are often more intuitive, easier to maintain and cause less burden on programmers. Thus recursive implementation has a "programmer performance" advantage over iterative approaches.

-- Exercise 4 --

Please refer to Q4.pdf for Exercise 4