Ex2

(ii) The average time complexity for the original ***delete*** is O(logn) for BST, and the worst time complexity is O(n). However, if we re-insert all elements after deleting one element, the time complexity is O(nlogn), which has worse performance. Therefore, this idea should not be recommended.

Ex3

For insertion, we take two main steps:

1) Do the insert operation just as in BST trees. Use a while loop while(1) (always do the loop), and in the loop body, it divides into two conditions of entering either the left successor tree or the right one of the judging node (start from the root node). If any successor tree should be entered but there is nothing in that tree, then insertion breaks after creating a new leaf node there with the inserted key (and balance 0); otherwise, continue judging the "root node" with that successor tree.

Besides, for any node being judged, store them in a stack. (This is essential for the second step.)

2) Do the rebalancing. For all the nodes stored in the stack built in 1) (we can use a while loop which stops when the stack is empty), determine if they have a balance 2 or -2. If so, do a single rotation if the new element is inserted in the same direction as the imbalance (both left or both right), or do a double rotation case (LR rotation for LR insertion and RL rotation for RL insertion) otherwise.

For deletion, still need two main steps:

i) Do the deletion. Firstly use a while(1) loop to find the node with the corresponding key value and breaks when the node is being found or we reach a leaf node with the node not being found (deletion failure). After finding the node, find the min value node in its right successor tree or the max value node in its left successor tree, then swap these two nodes (if it is a leaf node, skip this). Finally delete the node with the given value (it must be at the position of leaf node).

Besides, during the procedure of finding the two kinds of nodes, store all the nodes being visited in a stack.

ii) Do the rebalancing. This step is the same as the corresponding step of insertion, with the stack being used replaced by the one in i).

Difference:

The main difference is that the recursive implementations use the function in itself to track the path of insertion and make the rebalancing, while the iterative way uses an extra stack to track that path. For the complexity, the time complexities for them are almost the same, but the space complexity of the recursive implementations will be higher as it calls functions many times.

Ex4

(i).

According to the definition of median property, if an AVL tree has median property, then the number of nodes in the left tree of the root must be equal to or one more than the right tree.

According to the meaning of the perfectly balanced tree, all paths from the root to a leaf have length h or h-1. That means the h-1 layers of AVL tree are full. Then assume one AVL tree has median property, and it is perfectly balanced, but the h-1 layer of this tree is not full. Then there must be some nodes whose number of nodes in the left subtree minus the number of nodes in the right subtree is not 0 or 1 in this case, which is contradictory to the assumption that this AVL tree has median property. Then we get the conclusion, if an AVL tree satisfies the median property, then it is perfectly balanced.

(ii).

First we use _insert() function to insert the new value to the tree. Then we have gotten an AVL tree, and then what we need to do is just make it has the median property.

Here we use a new function me_balance() to make a new balance.

We need to calculate the number of nodes of left tree minus the number of right tree for every node, and it must have the value of 1 or 0, otherwise rebalance is needed. Let us call this number be c.

If c is 0 or 1 for one node: do nothing.

If c is -1: we should go to the right tree to find the new root's data, which is the minimum number of the right tree. Call this

number a. And the root's data is r. First delete the node whose data is a. Then set the data of root be a. Then insert the original root's data to the AVL tree. Use recursion to make a new balance for the whole tree. When we come to a null node, just return.
If c is 2: we should go to the left tree to find the new root's data, which is the maximum number of the left tree. Call this number a. And the root's data is r. First use _delete() function to delete the node whose data is a. Then set the data of root be a. Then use _insert() function to insert the original root's data to the AVL tree. Use recursion to make a new balance for the whole tree. When we come to a null node, just return.
(iii)
First we use _delete() function to delete node of chosen value from the tree. Then we have gotten an AVL tree, and then what we need to do is just make it has the median property.
Here we use a new function me_balance() to make a new balance.
Just like (ii), we need to calculate the number of nodes of left tree minus the number of right tree for every node, and it must have the value of 1 or 0, otherwise rebalance is needed. Let us call this number be c.
If c is 0 or 1 for one node: do nothing.
If c is -1: we should go to the right tree to find the new root's data, which is the minimum number of the right tree. Call this number a. And the root's data is r. First delete the node whose data is a. Then set the data of root be a. Then insert the original root's data to the AVL tree. Use recursion to make a new balance for the whole tree. When we come to a null node, just return.
If c is 2: we should go to the left tree to find the new root's data, which is the maximum number of the left tree. Call this number a. And the root's data is r. First use _delete() function to delete the node whose data is a. Then set the data of root be a. Then use _insert() function to insert the original root's data to the AVL tree. Use recursion to make a new balance for the whole tree. When we come to a null node, just return.
(iv)
We know the time complexity of _insert() and _delete() is O(log i) from the lecture.(i is the height of the subtree). Similarly, the time complexity of left_max() and right_max() is also O(log i). After modified, the insert and delete operation both include me_balance(). And because we need to check for every node, for the worst case, we need to operate n times me_balance(). The time complexity is related to the height of the subtree. For every me_balance operation, the time complexity is O(log i) because it includes one _insert() operation and one _delete() and also the left_max(), right_min(), operations and i is the height of the subtree. As a total, for all the nodes, the total time complexity is:

$$O\left(\sum_{i=1}^{\log_2 n} \log_2 i\right) = O\left(\log_2 \prod_{i=1}^{\log_2 n} i\right) = O(\log_2(\log_2 n)!) = O((\log_2 n)^2)$$