Data Structures 4

Question 1:

|  |  |  |
| --- | --- | --- |
| Type | Usage | Space Complexity |
| Deque<Node> backtrackRotated | Saves the pointer to a node that was rotated during the insertion (if it exists). To know which node to delete during backtracking. | One Node is added in all insertions except NO\_IMBALANCE. |
| Deque<Node> backtrackInsert | Saves the pointer to a node that was inserted. To know which node to delete during backtracking. | In insertion only one is added (a pointer to the inserted node). |
| Deque<String> backtrackType | Saves the type of the insertion, to know which backtrack type to use. | In insertions LEFT\_LEFT, RIGHT\_RIGHT and NO\_IMBALANCE only one is added. In insertion with double rotation (LEFT\_RIGHT and RIGHT\_LEFT) three are added. |
| Deque<Integer> backtrackValue | Marks if a node is in the tree. Used during backtracking, needed for backtracking double rotation. If node was inserted add 1, otherwise add 0. | In insertions LEFT\_RIGHT and RIGHT\_LEFT two are added (one 1 and one 0). In all other insertions only one 1 is added. |

Question 2:

|  |  |  |
| --- | --- | --- |
| Operation | Number of repetitions | Total time complexity |
| AVLTree.Node public void updateHeight() | In the worst case as the height of the tree \* 2 () \* 2. It is (\* 2) because during backtracking insertion with double rotation we call Backtrack() twice. | In the worst case it is () \* 2. . Total: . |
| AVLTree.Node public void updateSize() | In the worst case as the height of the tree \* 2 () \* 2. It is (\* 2) because during backtracking insertion with double rotation we call Backtrack() twice. | In the worst case it is () \* 2. . Total: . |
| BacktrackingAVL public void Backtrack() | In the worst case it is 2 (during backtracking insertion with double rotation, otherwise it is 0). | It is had operations with time complexity and operations with time complexity. Therefore, total complexity time is . |

Question 3:

|  |  |  |
| --- | --- | --- |
| Type | Usage | Space Complexity |
| Deque<Node> backtrackNodes | Saves the node before the insertion operation. Used for regular insertion and for insertion with split. | In regular only one added. In insertion with splits we add as many splits we have. |
| Deque<Node> backtrackParent | In split, saves the node that median value joins. | In regular only one added. In insertion with splits we add as many splits we have. |
| Deque<Integer> backtrackMidIndex | In split, saves the index of median value in its new node. | In regular only one added. In insertion with splits we add as many splits we have. |
| Deque<T> backtrackInserted | Saves the value that was inserted. To know which value to delete in backtrack. | In regular only one added. In insertion with splits we add as many splits we have. |
| Deque<Character> backtrackType | Saves the type of the insertion, to know which backtrack type to use. | In regular only one added. In insertion with splits we add as many splits we have. |

Question 4:

|  |  |  |
| --- | --- | --- |
| Operation | Number of repetitions | Total time complexity |
| Backtrack without merge operation: Deletion of last inserted value from the node we saved. | 1 | In regular backtrack (no split involved), we use the deques we saved to simple delete because we have pointer to the node. |
| Backtrack with merge operation:  For each split that was made we merge the nodes that were splitted. In the **first** merge of a backtrack we search the node that has the value that was last inserted and delete it. In each merge we also build the merged node. | As many splits we had, for most | As long as, we see the same value in deque, we do a merge. In the first merge we do a operation (when search and delete, as explained). Building the merged node for each merge is (fixed=קבוע). |

Question 5:

Danny's implementation takes in time complexity for **the backtrack operation** (not for insert), because changing the root of the tree is a single operation.

Question 6:

Danny's implementation is not better than ours, because instead of doing the insert operation in , it is done in for time complexity.

And in each insertion, we save the whole tree so it's in space complexity, instead of .