Questions:

i. How does the algorithm of balanceTreeTwo(); balance the tree -- why does this work? Explain it for all steps.

1. First, restructure the tree by calling transformToList(); to turn it into a right-leaning linked list.

We now have a list from the tree we started with from this first part

2. Compute the parameter $M = (N + 1) - 2 \lfloor \log_2 N \rfloor$. a. N is the total number of nodes, $\lfloor \log_2 N \rfloor$ means to compute the floor of log2 *N*.

We computed the total number of nodes in order to be able to determine number of comparisons which will be used for sorting.

3. From the root of the tree (which is now a right-leaning linked list), perform left rotations on every odd node until *M* odd nodes have been rotated. a. Example: Imagine a list 0->1->2->3->4. If $M = 2$, then we perform rotateLeft() on the '0' node and the '2' node. That's exactly *M* odd nodes.

We do an odd number of rotations left because it is a right leaning linked list thus is separates the nodes properly.

4. Compute the parameter $K = \lfloor \log_2 N \rfloor - 1$.

We need to know this value in order to know the number of comparisons needed for sorting in order to sort the list we have.

5. Compute this loop: a. While $K > 1$, rotate every odd node to the left. b. Decrement K by 1 when finishing a set of rotations. c. When $K == 1$, only rotate the parent node and complete the algorithm

By computing the given loop with odd nodes to the left it is sorting the list. When K=1 it means that it is completed sorting and does not need to move values anywhere else thus sorting the list.

ii. What is the total time complexity of the balanceTreeOne(); function in the average case? Consider how this function worked. balanceTreeOne(); first calls sortedTree(); then creates a new BST out of that. The cost of this should be $O(BalanceTreeOne) = O(sortedTree) + O(put(int[a]))$. You will fill this in for the actual values.

If we started with a BST for example 3-2-1 we would need to, for example. for searching in a BST, we have to traverse all elements (in order 3, 2, 1). This means that searching in BST has worst case complexity of O(n). However in average the time complexity is O(h) where h is height of BST. In order to perform the action of for example inserting element 0, it must be inserted as the left child of 1. This means another traversal of all elements 3-2-1 is needed to insert 0 which has worst case complexity of O(n).Again the average time complexity is O(h). Thus, for a BST we can conclude log(n) as the average case. So for us we have log(n) + log(n) = 2log(n).

iii. What is the space complexity of the balanceTreeOne(); function?

Space complexity proportional to N, where N is the number of elements in the tree. This makes it O(N) so it depends on user input as to the amount of elements they desire to input.

iv. What is the total time complexity of the balanceTreeTwo(); function? Perform a similar analysis to I, where you sum up the cost of the internal operations.

In terms of time complexity, it can be proportional in the worst case. Since the worst case is sought to be avoided, we can balance the tree and get the complexity proportional to logarithm N. For a linked list we have time complexity O(N) which is then added to the BST's log N giving us O(N)+log(N).

v. What is the space complexity of balanceTreeTwo();?

In terms of space complexity, it is O(n) depending on n elements similar to before.