1. Balance Tree Two works by calculating the floor of the $\log_2 N$ of the linked list, and then rotates all the odd nodes except for the last one to the left. Since this list was right leaning due to the fact that our transformtolist function turned all the nodes to the right to make the list, we have to turn certain nodes back to the left to create the tree. Because it will rotate every other node left, it will by definition create a balanced tree because each parent node will have the correct number of children, with the possible exception of the last one if there are not enough nodes in the tree.

```
public int[] sortedTree() {
		int treeLength = returnSize();
		int sorted[] = new int[treeLength];
		sorted = sortedRecursive(root, sorted);

		mergeSort(sorted, 0, treeLength - 1);
		return sorted;

	}
public void put(int[] a) {
		for (int i = 0; i < a.length; i++) {
			this.put(a[i]);
		}
	}
public int[] sortedRecursive(Node x, int a[]) {
		countarray(x, a, 0);
		return a;

	}
public int countarray(Node x, int a[], int offset) {
		a[offset] = x.val;
		if (x.left != null) {
			offset = countarray(x.left, a, offset + 1);
		}
		if (x.right != null) {
			offset = countarray(x.right, a, offset + 1);
		}
		return offset;
	}

public BST balanceTreeOne() {

		BST balancedTree = new BST();
		int a[] = sortedTree();
		balancedTree.put(this.sortedTree());

		return balancedTree;
```

```
}
public int trueReturn(Node x) {
		int count = 1;
		if (x == null)
			return 1;
		if (x.left != null) {
			count += trueReturn(x.left);
		}
		if (x.right != null) {
			count += trueReturn(x.right);
		}
		return count;

	}

mergeSort(sorted,0,treeLength);

while(first1<=last1 && first2 <= last2) {
			if(a[first1] <=a[first2]) {
				temparray[index] = a[first1];
				first1++;
			}
			else {
				temparray[index] = a[first2];
				first2++;
			}
			index++;
		}
		while(first1 <= last1) {
			temparray[index] = a[first1];
			first1++;
			index++;
			}
		while(first2 <= last2) {
			temparray[index] = a[first2];
			first2++;
			index++;
		}
		for(index = first; index<=last; index++) {
			a[index] = temparray[index];
			}
	public void mergeSort(int a[], int first, int last) {
		if(first<last) {
			int mid = first + (last-first)/2;
			mergeSort(a,first,mid);
			mergeSort(a,mid+1,last);
```

```
merge(a,first,mid,last);
}
```

2. BalanceTreeOne begins by making calls to sortedTree. This function works by calling ReturnSize to determine how many nodes are in the tree, and then sortedTree creates an array of that size. ReturnSize calls trueReturn, which iterates through the entire tree and counts the number of nodes. It does this using divide and conquer, by essentially have one recusive call to search the left of the tree, and one recursive call to search the right of the tree. This yields $O(N\log_2 N)$. Now that we know the size, the array of size N is created. sortedTree then calls sortedRecursive to place all the values into the array. It does this using the same divide and conquer method that we saw in trueReturn. So we have another $O(N\log_2 N)$. Then mergeSort is called, which is yet another divide and conquer algorithm, which divides the array successively by 2 until it no longer can, and sorts. This gives another $O(N\log_2 N)$. Finally, the put function is called, which puts the array into a new tree. Put iterates through the entire array and places the values into a tree, so we have a time complexity of O(N). So our total time complexity is their sum, or $O(N+3N\log_2 N)$.

```
for(index = first; index<=last; index++) {
                        a[index] = temparray[index];
                        }
int a[] = sortedTree();

int sorted[] = new int [treeLength];
```

3. The space complexity of balanceTreeOne is equal to the sum of the space complexity of the sorted array function + N, as we are creating a new tree. The sorted array function uses N extra space to copy the tree into an array, so we're at at least 2N. But it also uses recursive mergesort, which increases its space complexity by O(N) once again, as we generate a copy of the initial array to be sorted. Therefore we have the sum of N+N+N, or O(3N).

```
public Node transformBase(Node x) {
                // Node iter = x;
                if (x == null) {
                        return x;
                }

                while (x.left != null) {
```

```
                        x = rotateRight(x);

                }

                if (x.right != null) {
                        x.right = transformBase(x.right);
                }

                return x;

        }
public void transformToList() {
                transformBase(root);
                Node x = root;


        }
public void balanceTreeTwo() {

                transformToList();

                System.out.print("test");

                int N = returnSize();
                double log2 = Math.log(N) / Math.log(2);
                double pow = Math.floor(log2);
                double subpart = (int) Math.pow(2, pow);
                double M = (N + 1) - subpart;
                // use the floor function for the log2N

                Node iter = root;
                Node iter2 = null;
                Node iter3 = iter.right.right;
                for (int i = 0; i < M; i++) {
                        System.out.println("test1");
                        System.out.println(i);
                        iter2.right = rotateLeft(iter);
                        iter2 = iter2.right;
                        iter = iter3;
                        if(iter.right != null && iter.right.right !=null) {
                                iter = iter.right.right;
                        }
                }

                iter = root;
                iter2 = null;
```

```
                iter3 = iter.right.right;

                double K = Math.floor(pow - 1);

                // new stuff
                while(K > 1) {
                        iter2.right = rotateLeft(iter);
                        iter2 = iter2.right;
                        iter = iter3;

                        if(iter.right != null && iter.right.right !=null) {
                                iter = iter.right.right;
                        }
                        K--;
                }
                if (K == 1) {
                        rotateLeft(root);
                        return;
                }

        }
public Node rotateLeft(Node h) {
                if (h.right == null) {
                        return h;
                } else {
                        Node x = h.right;
                        h.right = x.left;
                        x.left = h;
                        return x;
                }
        }
```

4. balanceTreeTwo's time complexity is equal to the sum of transformToList, transformBase, and 2N-1, as well as the cost of RotateLeft, which is a constant and therefore of low significance. Transform to list consists of nothing but a call of transformBase, so its cost is constant as well. transformBase involves starting at the root and iterating through the entire tree, rotating everything to the right. It then moves right to the next node, and rotates everything to the right. By this logic, it follows that it would have cost of O(N!). balanceTreeTwo consists of iterating through the linked list and rotating every odd node, minus the root. Due to having to iterate through the entire list twice, rotating every other node, it has cost of 2N-1, or just 2N. So we have O(2N+N!).

```
Node iter = root;
                Node iter2 = null;
                Node iter3 = iter.right.right;
                for (int i = 0; i < M; i++) {
                        System.out.println("test1");
                        System.out.println(i);
                        iter2.right = rotateLeft(iter);
                        iter2 = iter2.right;
                        iter = iter3;
                        if(iter.right != null && iter.right.right !=null) {
                                iter = iter.right.right;
                        }
                }

                iter = root;
                iter2 = null;
                iter3 = iter.right.right;

                double K = Math.floor(pow - 1);

                // new stuff
                while(K > 1) {
                        iter2.right = rotateLeft(iter);
                        iter2 = iter2.right;
                        iter = iter3;

                        if(iter.right != null && iter.right.right !=null) {
                                iter = iter.right.right;
                        }
                        K--;
                }
                if (K == 1) {
                        rotateLeft(root);
                        return;
                }

        }
public Node transformBase(Node x) {
                // Node iter = x;
                if (x == null) {
                        return x;
                }

                while (x.left != null) {

                        x = rotateRight(x);
```

```
                }

                if (x.right != null) {
                        x.right = transformBase(x.right);
                }

                return x;

        }
public Node rotateLeft(Node h) {
                if (h.right == null) {
                        return h;
                } else {
                        Node x = h.right;
                        h.right = x.left;
                        x.left = h;
                        return x;
                }
        }
```

5. The space complexity of balanceTreeTwo is equal to the sum of the space complexity of the functions listed in part 4, so the sum of balanceTreeTwo, rotateLeft/Right, transformBase, and balanceTreeTwo does nothing but declare variables and iterate them, therefore it uses constant extra space. transformToList only uses variable declarations and iterates, so it also uses constant extra space. transformBase just declares variables and iterates them, meaning it also uses constant extra space. rotateLeft/Right also just assigns variables and iterates them, meaning that it to uses constant extra space. So we have O(C)+ O(C)+ O(C)+ O(C),, or O(4C) ~ O(C). So balanceTreeTwo uses constant extra space.