**I.**

The balanceTreeTwo() function works by doing a series of right rotations on the tree until it is completely right leaning. Once the right rotations are complete the tree is in ascending order (lowest number is located at the root). The algorithm than rotates a certain number of odd nodes to the left. The number of odd nodes that is rotated left is based off the total number of nodes. After all the left rotations, the remaining right leaning nodes of the tree are the ones that weren't rotated. At this point there are either M or M-1 (M = number of odd nodes rotated) subtrees that have one child to the left. After those rotations several more left rotations are completed on the remaining right leaning odd nodes until the tree is balanced. The nodes that are rotated left depend on the variable K.

This function balances the tree by making the node with the middle value the root of the tree when there is an odd number of nodes. If the tree has an even number of nodes either the node with the lower middle value or the upper middle value will become the root of the tree. This ensures that the absolute difference between the height of the left sub tree and the right sub tree is never larger than one. The function also makes it show that the left child of the root is the node with the middle number relative to the nodes that have a smaller value than the root. If there is even number of nodes that have a value lower than the root either the node with the upper middle number or lower middle number will act as the middle. The right child of the root is the node with the middle number relative to the nodes that have a larger value than the root after the function runs. If there is even number of nodes that have a value greater than the root either the node with the upper middle number or lower middle number will act as the middle. The function makes each sub parent and its children using a similar process. Since each sub parent is the middle value relative to its left and right subtrees, the absolute difference between the number of nodes to its right and its left can never be greater than one.

**II.**

```
node* balanceTreeOne() {
		int *arr = sortedTree();
		int size = returnSize();
		top2 = sortedArrayToBST(arr, 0, size-1);
		return top2;
	}
```

The code shown above showcases the balanceTreeOne function. The run time for the first line of code (yellow) is equal to the run time of sorted tree. The run time of the second line of code (cyan) is equal to the run time of returnSize, and the run time of the third line of code (green) is equal to the run time of `sortedArrayToBST`. This means the total run time of balanceTreeOne is the sum of the run time of the three functions.

```
int returnSize() {
              cnt = 0;
              cntInOrderPri(top);
              return cnt;
       }
```

The code shown above showcases the returnSize function. The functions determines the number of nodes in the BST by performing a in order traversal of the tree and increases the value of cnt by one every time a node is reached. The run time of a in order traversal of a binary tree is equal to O(n) where n is equal to the number of nodes. Since every node must be visited the run time will always be O(n). Since the in-order traversal is the only part of returnSize function that doesn’t take constant time to run, the total run time of returnSize in all cases is O(n).

```
int* sortedTree() {
              int *arr;
              int size = returnSize();
              sortedarr = new int[size];
              arr = new int[size];
              cnt = 0;
              arrInOrderPri(top);
              for (int i = 0; i < size; ++i) {
                     arr[i]= sortedarr[i];
              }
              return arr;
       }
```

The code shown above showcases the sortedTree function. The only part of the function that doesn’t run for a constant time is the call to returnSize (yellow) and the for loop shown in green. As stated in the previous paragraph the run time of returnSize is O(n). The run time of the for loop is also O(n). This means that the total run time of this function is O(n) + O(n) = O(n).

```
node* sortedArrayToBST(int arr[], int start, int end)  {
              if (start > end)
                     return NULL;

              int mid = (start + end) / 2;
              node *root = createLeaf(arr[mid]);

              root->left = sortedArrayToBST(arr, start, mid - 1);
              root->right = sortedArrayToBST(arr, mid + 1, end);

              return root;
       }
```

The code shown above showcases the sortedArrayToBST function. The only portion of this function that does not have constant run time in this function is the two recursive function calls shown in green. The run time of the recursion can be determined using the master theorem. The generic form of theorem is $T(n) = aT\left(\frac{n}{b}\right) +$ f(n) . a is the number of problems in the recursion b is the amount each subproblem is reduced by after each recursive function call, and f(n) is equal to the amount of time taken at the top level. Since each recursive call looks at the relative half of the array b = 2.  There are only two recursive calls in the function which means a is also equal to 2. The run time of the function excluding the recursive calls is O(1) which means f(n) = O(1).  Based of this information the run time of this function in all cases is O(n).

```
node* balanceTreeOne() {
```

```
                int *arr = sortedTree();
                int size = returnSize();
                top2 = sortedArrayToBST(arr, 0, size-1);
                return top2;
        }
```

All of the information in the previous paragraphs indicates that the run time of balanceTreeOne is O(n) + O(n) + O(n) = O(n).

**III.**

```
node* balanceTreeOne() {
                int *arr = sortedTree();
                int size = returnSize();
                top2 = sortedArrayToBST(arr, 0, size-1);
                return top2;
        }
```

Similar to the time complexity, the space complexity is equal to the sum of the space complexity for each function.

```
int returnSize() {
                cnt = 0;
                cntInOrderPri(top);
                return cnt;
        }
```

The returnSize function only creates one variable when its called which means its space complexity is equal to O(1).

```
int* sortedTree() {
                int *arr;
                int size = returnSize();
                sortedarr = new int[size];
                arr = new int[size];
                cnt = 0;
                arrInOrderPri(top);
                for (int i = 0; i < size; ++i) {
                        arr[i]= sortedarr[i];
                }
                return arr;
        }
```

The sortedTree function creates two arrays and the size of both arrays is equal to the number of nodes in the tree. This means that the space complexity of sortedTree is equal O(2n) = O(n).

```
node* sortedArrayToBST(int arr[], int start, int end)  {
                if (start > end)
                        return NULL;

                int mid = (start + end) / 2;
                node *root = createLeaf(arr[mid]);

                root->left = sortedArrayToBST(arr, start, mid - 1);
                root->right = sortedArrayToBST(arr, mid + 1, end);

                return root;
```

```
        }
```

Since each recursive call sees half the data there are a total of O(log n) levels. On each level the amount of space used is O(1). This means that the space complexity of sortedArrayToBST is O(log n).

Based of the information in the previous paragraphs the total space complexity is O(n) + O(1) + O(log n) = O(n).

**IV.**

```
void transformtoListPri(node* ptr) {

                while (ptr->left != NULL) {
                        rotateRightLink(ptr);

                        if ( ptr->parent == NULL) {
                                break;
                        }
                        else {
                                ptr=ptr->parent;
                        }
                }
                if (ptr->right != NULL) {
                        transformtoListPri(ptr->right);
                }
        }
```

The code shown above showcases the transformtoListPri function. The function performs a certain number of right rotations on the current parent node and then traverse the tree to the right via recursion and repeats the same process. The time complexity of each rotation is O(1). The total number of times the rotation function is called O(n). This means that the total time complexity of transformtoListPri is equal to O(1) ∗ O(n) = O(n).

```
void balanceTreeTwo() {
                transformtoList();
                int size = returnSize();
                int logint = log2(size);
                int M = (size+1)-pow(2,floor(log2(size)));
                node* temp = top;
                node ** pointers;
                int K = log2(size)-1;
                cout <<  endl;
                cout <<"The value of K is " << K << endl;
                for (int i = 0; i < M; ++i) {
                        rotateLeftLink(temp);
                                temp = temp->parent->right;
                }
                temp = top;
                int cnt2=0;//counts how many pointers to make
                while (temp != NULL ) {
                        ++cnt2;
                        if (temp->right == NULL) {
                                break;
                        }
                        else {
```

```
                temp = temp->right->right;
            }
        }
        temp = top;
        cout << endl;
        while (K!=0) {
            if (K == 1) {
                for (int i = 0; i < 1; ++i) {
                    rotateLeftLink(temp);
                }
                cout << "Ran K==1 case " << endl;
            }
            else {
                for (int i = 0; i < cnt2; ++i) {
                    rotateLeftLink(temp);

            if (temp->parent != NULL && temp->parent->right != NULL) {
                temp = temp->parent->right;
            }

                }
            }
            temp = top;
            cout << "Out of for loop" << endl;
            cnt2 = 0;//counts how many pointers to make
            while (temp != NULL) {
                ++cnt2;
                if (temp->right == NULL) {
                    break;
                }
                else {
                    temp = temp->right->right;
                }
            }
            temp = top;
            --K;
            cout <<endl;
        }
    }
```

The code shown above showcases the balanceTreeTwo function. The constraints on the run time of the function are the for loops shown in green, the while loops shown in cyan, and the call to the transformtoList function shown in yellow. Based of the information in the previous paragraph the time complexity of transformtoList is O(n). The first for loop runs for M times where $M = (n+1) - 2^{[log_2 n]}$. The lowest number M can be is 1 and the highest number M can be is $\frac{n+1}{2}$. This means that in most situations M is equal to O(n). The first while loop runs for a total of $\left(\frac{n-M}{2}\right)$ [rounded up], since that is the number of odd leaning right nodes after M odd nodes have been rotated. This means that the first while loop runs a total of O(n) times. The second while loop runs for K times where $K = [log_2 n] - 1$. The first for loop inside the second while loop will always run one time and has a run time of O(1). The second for loop inside the second while loop will run K-1 times and the run time of the for loop is equal to the current number of odd leaning right nodes after each pass of the while loop. The most amount of odd leaning right nodes there can be at this point in the code is n-1 and the least amount is $O\left(\frac{n}{2}\right)$. This means that runtime of the second for loop inside the while loop is O(n). The while loop inside the second while loop also runs for O(n), since the amount of time it runs is equal to the number of odd leaning right nodes. This means that the run time of second while loop is $O(K * O(1)) + \big(O(n) * (K - 1)\big) + (K * O(n)) =$

$O(nlog_2n)$. All of this information indicates that the total run time of balanceTreeTwo is $O(nlog_2n) + \mathrm{O(n)} + \mathrm{O(n)} + \mathrm{O(n)} = \mathrm{O(n}log_2n)$

**V.**

All of the variables created inside the balanceTreeTwo function take up constant space. The pointers created inside the rotate function can be called outside of the function without any issues. They are called inside the functions to make the code easier to view. This means that the rotate functions does not take any space, which means transformtoListPri function takes O(1) space. All of this information indicates that the total space complexity of balanceTreeTwo is O(1).