**Question 4:**

Start with an arbitrary n-node binary tree.

It can be shown that any arbitrary n-node binary tree can be transformed into a right-going chain with a maximum of n-1 rotations. The algorithm for that can be shown as:

1. Start at the root node. If there is a left child, do a right rotation so that the left child is the new root node. Repeat step 1.
2. If there is no left child, recursively traverse down to the right child and repeat step 1.
3. If there are no more left or right children, the binary tree has been converted to a right-going chain.

With the algorithm above, each rotation moves exactly one node into the right-going chain. The original root node does not need to be rotated into the chain because it starts off as part of the right-going chain. Therefore, it will take a maximum of n-1 rotations to move all n-nodes into a right-going chain.

A left rotation is the inverse of a right rotation. If a right rotation is performed followed immediately by a left rotation on the same nodes, the binary tree will remain the same. It follows from this that the right-going chain can be reverted back into any arbitrary n-node tree by some amount of left rotations. It will take a maximum of n-1 left rotations to achieve any n-node binary tree because it took at most n-1 right rotations to convert any binary tree into a right-going chain, and performing the left-rotations is merely performing the inverse of right-rotations.

To summarize, a maximum of n-1 right rotations is required to turn an arbitrary n-node binary tree into a right-going chain. It takes an additional maximum of n-1 left rotations to turn a right-going chain into any arbitrary n-node tree. Combining the two gives 2n-2, which is the maximum number of rotations required to change any arbitrary n-node binary tree into any other arbitrary n-node binary tree.

**Question 5:**

SearchIndex(a, leftindex, rightindex){

index = floor((rightindex+leftindex)/2);

if(index == a[index])

return index;

if(index<a[index]){

if(leftindex==index)

return -1;

return SearchIndex(a,leftindex,(index-1));

}

if(index>a[index]){

if(rightindex==index)

return -1;

return SearchIndex(a,rightindex,(index+1));

}

}

Algorithm for question 5 is fairly straightforward and assumes that the array is sorted smallest to largest. This algorithm looks at the middle index in each recursion and subdivides the array by half each time. The algorithm recursively only looks at either the left or right partition but not both. This reduces the search space by a half each time and runs in O(logn) time.

The code runs recursively until it finds a[i] == i and returns index or returns -1 if it is not found. It runs by taking the array and looking at the middle index and checking if a[middleindex] == middleindex. If this is true, it is finished; else it checks if the key is bigger or smaller than the index. If the key is bigger than the index then any instances of a[i] == i must occur to the left of the middleindex. This is because as the index increases by one each time, the corresponding keys must increment by at least one since the array is sorted with distinct integers. If this is the case, then a[i] == i can never occur to the right of middleindex because the key must increment by at least one each time the index increases but the index can only increment by one. Therefore, the index can never catch up to the value of the key and we should look in the left partition. Similar logic follows if the key is smaller than the index – it follows that we should look to the right because moving towards the left, the index can only decrement by a maximum of one while the key must also decrement by a minimum of one. The index can never catch up to the key. If the algorithm finds that index is equal to the right or left index (depending on which direction it is traversing to), it terminates because there is no a[i] == i.