**Ch3**

To be more comprehensive, we choose several different test case. First we choose the minimal and maximal case, that is having only 1 node and having 10 nodes. After that we choose to test if two tree have different root, which should be not identical. Then it comes the case where the root is the same while one of the child is not the same, which should also not be identical. Afterwards, we test the case in which the original tree having only left child, is perfect binary search tree, is complete binary search tree respectively. Finally, we test some randomly generated trees and test their identity. All the expected result is obtained by drawing the corresponding tree with hand and decide the identity. And the result turned out that we seems to get the right solution.

The test case and the expected output and the actual output is shown below respectively.

**Test case:**

*Case1: only 1 node*

*1 1*

*1*

*1*

*Case2: the tree has the max nodes, that is 10*

*10 9*

*3 2 4 7 5 6 8 9 10 1*

*3 4 2 7 1 8 5 6 9 10*

*3 7 4 2 1 5 6 10 9 8*

*3 2 4 1 7 8 5 6 9 10*

*2 4 6 8 3 5 1 9 10 7*

*3 5 8 7 4 9 10 6 2 1*

*3 8 6 7 10 2 1 9 5 4*

*3 4 1 7 5 2 6 10 8 9*

*3 6 8 9 4 5 7 2 10 1*

*3 9 2 8 1 10 7 5 4 6*

*Case3: hava different root node*

*2 1*

*1 2*

*2 1*

*3 2*

*2 1 3*

*1 2 3*

*3 1 2*

*6 4*

*5 2 3 6 1 4*

*1 3 2 5 4 6*

*2 3 1 6 5 4*

*3 6 2 4 1 5*

*4 3 2 6 1 5*

*Case4: have the same root node but have different child node*

*3 1*

*3 1 2*

*3 2 1*

*4 2*

*3 1 4 2*

*3 2 4 1*

*3 4 2 1*

*Case5: all nodes of the original tree has no right child*

*4 2*

*1 2 3 4*

*1 3 2 4*

*1 4 2 3*

*Case6: the original tree is perfect binary search tree*

*7 5*

*4 2 6 1 3 5 7*

*4 6 2 1 3 5 7*

*4 2 1 3 6 7 5*

*4 2 1 6 5 3 7*

*4 1 2 3 5 6 7*

*4 3 1 5 7 6 2*

*Case7: the original tree is complete binary search tree*

*5 5*

*4 2 5 1 3*

*4 5 2 3 1*

*4 2 3 1 5*

*4 3 2 5 1*

*4 1 2 3 5*

*4 3 5 2 1*

*case8: random choosed tree*

*10 5*

*7 2 9 10 1 8 5 3 6 4*

*7 2 9 10 1 8 6 4 5 3*

*7 6 2 5 4 3 10 8 9 1*

*7 9 4 10 5 8 3 6 1 2*

*7 4 10 8 3 2 5 6 9 1*

*7 1 9 6 2 10 3 5 4 8*

*5 5*

*4 2 3 1 5*

*4 3 5 1 2*

*4 3 1 5 2*

*4 1 3 5 2*

*4 3 5 2 1*

*4 2 3 5 1*

*8 5*

*5 1 2 4 8 6 3 7*

*5 1 2 4 6 8 7 3*

*5 1 4 6 3 7 8 2*

*5 6 7 1 8 2 3 4*

*5 8 4 6 1 7 2 3*

*5 4 2 6 1 3 7 8*

**Expect output:**

*Yes*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*Yes*

*Yes*

*Yes*

*No*

*No*

*Yes*

*Yes*

*No*

*No*

*No*

*Yes*

*No*

*Yes*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*Yes*

*No*

*No*

*No*

*No*

*No*

**Actual output:**

*Yes*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*Yes*

*Yes*

*Yes*

*No*

*No*

*Yes*

*Yes*

*No*

*No*

*No*

*Yes*

*No*

*Yes*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*No*

*Yes*

*No*

*No*

*No*

*No*

*No*

**Ch4**

As you can see above, we choose two algorithm to solve the problem.

The first is the most intuitive one, that is according to the given input to construct the corresponding binary search tree and then check if the trees are the identical. In this manner, first we have to construct the tree. To construct the binary search tree, we have to insert the node one by one, and each takes O(d)(d is the depth of the node after it is inserted). In the worst case, every node ,which is not leave ,has only one child , and then the total complexity will be O(1+2+...+n), which can be proved is O(nlg(n^2)). But in the average case, each insertion would take O(lg(k)) (k is the number of nodes in the tree before insertion), so the average cost of construction would be O(lg(1)+lg(2)+...+lg(n)), which is O(nlg(n)).And as you can see, we need to construct L + 1 trees, so the total cost will be O(L\*n\*lg(n)). After we construct the tree, we have to decide if the two trees are identical. we do this by implementing the checkIdentity function. During the procedural, as we only visit each node at most once, the each check would only take O(n), and because we have to check L times, so the total cost will be O(L\*n). So the final time complexity should be O(L\*n\*lg(n)). For the space complexity, as we only need n node to represent each tree, so the space complexity will be O(L\*n);

For the second method, as we don't need to construct the tree and save the input in the array, and during the procedural, we only visit each element at most once, so the time complexity will be O(L\*n). For the space, we need N int to save the input and at most N\*C(C is the space needed for each system stack), so the space complexity is also O(n). We only implemented the recursive version, it should be possible to implement this algorithm in a nonrecursive manner, but we didn't figured it out.

//algorithm 1 specification

We accept two root as the input. First, we check if the two root are both null, if so, we claim them to be identical. if not, we then check whether one of them is null, which means they are not identical. If they are both not null, then we will call checkIdentity function to recursively check their left child and right child, and claim them to be identical only if their left child and right child both pass the check, otherwise we will claim them to be not identical. To be more vivid, we draw the flow diagram as below [------FLOW CHART SHOULD BE HERE------------- ]

//algorithm 2 specification

The second is implemented by make full use of the property of the binary search tree. First of all, we know that the first element of the sequence must be the root of the binary search tree. And its left child should be the first smaller element followed by it , and by then same token the right child is the first larger element followed by it. But that's not most general case, cause root has no parent while all the other node has a parent. If a node have a parent, then it's child will have more restriction. Take a simple case for example, if given a node which is a left child of its parent, then its left child should not only be smaller than it but also be smaller than its parent, so in this case, the result is the same. but for its right child, that's not true. The right child should not only be larger than it but also be smaller than its parent, so for the right child , the result would be different. In order not to treat the root as a special case, we image that root is the left child of N+1 and right child of 0, which means that root should have of lower bound of 0 and a upper bound of N+1. Given this prerequisite, for each node e,whose value v is in the interval(lower, upper) and ,its left child should be the first element followed it whose value is in the interval (lower, v), and its right child should be the first element followed it whose value is in the interval (v, upper). Now, we can check the identity without constructing the tree. First, we check if the first element of the two sequences are the same, if so, we claim them to be identical, otherwise, we used the method discussed above to find it's left child and check it, and then find its right child and check it.......