Implementation

Base on the assignment illustrate, it is required that we need to implement two kinds of BBST trees and then compare them. But unfortunately, since the problem involves coding, this part is quite challenging for me. I need to collect additional information by online to understand, learn and try in the coding part, it took me a lot of time. Therefore, in the final result, I can only achieve one kind of BBST tree which is the AVL tree. For the AVL tree implementation, my code can successful pass all the testcase on OJ and within the running time 1 second.

AVL tree

The requirement of the problem is to maintain a multiset and have the following operation: insert, erase, find order, x-th smallest, precursor and successor. Also, the time requirement is to complete all test cases within 1 second. Therefore, the ordinary binary search tree must not be completed within the specified time. But the AVL tree always balance, so all the operation can search within O(logn). For the above reasons, I chose the AVL tree. Realizing balance in the AVL tree is done through the rotate root method. The key to determining whether to rotate is to calculate the height of each Node. When the child node is null, the height is regarded as -1, and the rest are left and right child nodes. The maximum height + 1 is the current node height. Now we have the basic structure of AVL tree, but it is only valid that the entered Node data are not repeated. If there is duplicate Node data, it will not insert. So, I need to improve on the basic structure. As mentioned earlier, if want to make all functions can be completed in O (longn) time. Therefore, for operation 4, 5 and 6, the data stored in each Node needs extra data information that is to calculate how many Nodes on the left subtree and the right subtree of each Node. Also, for operation 2, To decide whether to execute deletion it is to check whether there is any data of this Node before insert. For this reason, I use unordered_map to constantly check whether the data in the tree exists. The following will briefly explain the parts that need to be changed and how to create each function.

Operation 1: insert

To change this part, it needs to set up new conditions. If the Node data is larger than the root Node, it places the new Node to right child nodes. If the Node data is smaller than or equal to the root Node, it places the new Node to left child nodes. According to the new conditions, use the recursion method to place the new Node in the NULL position. The left subtree count or right subtree count of the passed root is +1 on the corresponding side indicates that a new node has been added. Next, update the unordered_map value. After finishing the insertion, then do the left rotate or right rotate if necessary and make the tree balance.

Operation: delete

It's almost the same as the insertion function. The only difference is the left subtree count or right subtree count of the passed root is -1.

Operation 3: find the order

This function is input the number find the correct order of this number. This is also can done by recursion. Compare the root Node of each level until fund this number. If found, return left subtree count + 1, the value of return is the order of this number. But one of the tricky part is don’t know if this return value is the smallest repeating number. So, the solution is not to return this value immediately, keep finding is it have the same number in the tree and the order is smaller. If yes, update the return value, until looking for the smallest in the tree.

Operation 4: find x-th small

This function is use recursion find the number. If input number is smaller than root left subtree count + 1, it goes to left child Node. If input number is larger than root left subtree count + 1, update input number to input number – (root left subtree count + 1) and it goes to right child Node. Last case is if input number is equal to root left subtree count + 1, it returns root value.

Operation 5: find the precursor

The recursion function again. The idea is use temp Node save the up one level root Node. Until find the input value return the up one level root Node.

Operation 6: find the successor

The method is like operation 5 do the same. But is doing opposite direction.

Data

- Large amounts of data

  The AVL tree is very efficient in dealing with large amounts of data. Because the characteristic of AVL tree is balance, the speed of the search is based on the height of the tree which is O(logn).

- Ordered data

  If use the ordinary binary search tree, the insert order will be both Node on left side or right side. It is not balance. Such as ascending order 1, 2, 3, 4, the tree will all insert to right side and height is 4. For descending order 4, 3, 2, 1 the tree will all insert to left side and height is 4. It is takes longer to searching number. But AVL tree is doing the same things, but it will rotate the tree, so that make the tree balance.

Strengths

The height of the AVL tree is always balanced. The height never grows beyond log N, where N is the total number of nodes in the tree. When use recursion function it is very fast can get the result. Therefore, if have large amounts of data and the data is unordered data, it is suited to use AVL tree. It gives better search time complexity when compared to simple Binary Search trees. AVL trees have self-balancing capabilities.

Weaknesses

The code is difficult to implement, because it needs to manage the height of each node, the left subtree count, right subtree count. All the value needs to update after insert a new Node or delete a Node. Each step must be calculation is correct. Also needs to clear the rotation method. So, this is also the reason why I use a lot of times to learn and study AVL tree, I can’t have time to create second kind of tree which is splay tree.

Compare

Even I don’t have time to implement splay, but I have done the research of splay tree. Splay tree also has rotation function try to make the tree balance. And insert, delete is also can done by O(logn) times. But splay trees can only guarantee that any sequence of n operations takes at most O(n log n) time. Therefore, the time of AVL tree is faster.