Persistent Dynamic Sets

Objectives:

* Read, use and develop established code
* Adjust given Binary Search Trees (called BSTs from now on) code for a practical application
* Get familiar with Recursion

Persistent Dynamic Sets:

During the course of an algorithm, sometimes it is necessary to maintain a past version of a dynamic set as it is updated. We call such a set persistent. One way to implement a persistent set is to copy the entire set whenever it is modified. However, this approach can slow down a program and also consumes much space, especially when the dynamic set is very large.



Consider a persistent set S, which is implemented using BSTs as shown in Figure 1 (a). Now, we are going to insert more elements. However, we want to maintain this BST for S. We do it as follows: in order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. Again, the new node with key 8 becomes the right child of a new root with key 4 whose left child is the existing node with key 3. As a result, we copy only part of the tree and share some of the nodes in the original tree, as shown in Figrue 1(b). Now, let us insert the key 6 into the set and we obtain Figure 1(c). When the key 9 is inserted, the node with key 9 becomes the left child of a new node with key 10, since we cannot modify the existing node with key 10. The new node with key 10 becomes the right child with key 8, as shown in Figure 1(d).

In this project, as input, we are given two sets S and T, where S is persistent and T is not. We assume S is fixed once given. T is dynamic, i.e., remove/add operations are allowed in this set. We also assume that the elements in S and T are all distinct. For S, you need to build a BST according to the order of the elements in S. Then, you are required to insert the elements in T into the above tree while maintaining the structure . In the above example, S={4, 3, 2, 8, 7, 10} and T={5, 6, 9}.

|  |
| --- |
| Figure 1 |
| ../../../../../Desktop/Screen%20Shot%202020-09-14%20at%2010.  ../../../../../Desktop/Screen%20Shot%202020-09-14%20at%2010. |

Source Code given

To get you started, [some code](https://drive.google.com/drive/folders/1g1Qh89mHmWoV9bZ7-sE9d7qDKuMJsnkA?usp=share_link) has been given. You are NOT to change any of the code. You are only allowed to add code. Compared to explaining each function here in the project document, that is in the code provided.

Implementation

1. The main goal of this project is to build the structure for S and T similar to Figure 1.
2. In the project, first build the BST for S. Then, add T into to build the BST . and are created based on the order of the elements in S and T.
3. Since will share part of nodes in , we need to distinguish these nodes. We do this by adding a **bool variable status** in each node. **status** is 1 if this node belongs to the tree , and **status** equals 0 if it is a node in (it can also be a common node in ).
4. When inserting an element x, implement two cases: x is in S or in T. These two cases are different as explained in Figure 1.
5. In the destructor functions or makeEmpty(), pay attention to the common nodes in and . One suggestion is to make empty first and then . Note that the common nodes in and cannot be deleted when emptying , and they have to be dealt with finally when emptying . Deleting the common nodes when emptying will lead to memory leak and some other problems.
6. The implementation of the function printTree() is given, which is to display the two trees and their common nodes. Please do not change its code since it is a part of the test.
7. In the Driver.cpp, several tests are given and the output is in the file “sample\_output.txt”. You can use them to test your code. Each test is small, and you can “draw” what the BSTs look like.
8. When overloading the operator=, copy one BST is easier. The difficulty lies in coping the second BST. Pay attention to the common nodes in the two BSTs.

Hints

1. Reading ALL of the code given is imperative. Some of you will just start to do and end of repeating stuff I already gave you wasting time.



1. Notice all of the debugging functions. I am not trying to give you extra work. I promise. These are to help you step by step to get closer to a successful project. USE THEM!

Submitting

After successful testing of your project, simply zip all .h and .cpp file(s) together and upload in Canvas.

