Assignment 6

Due: December 4, 2018 (11:59pm)

Assignment

Implement the bag template class from Section 10.5, using a binary search tree to store the items.

Purpose

Ensure that you understand and can use binary search tree.

Before starting

Read Chapter 10, with particular attention to 10.3 and 10.5.

How to Turn In

Submit all source code files (`*.cpp`, `*.cxx`, `*.h`) in a collection as a zip on Blackboard.

Files

- `bstbag.h`: Header file for this version of the bag class.
- `bstbag.cpp`: The implementation file for the new bag class. In contrast to the previous assignments, much of this code is already written. You only need to fill in the parts marked with "STUDENT WORK". `bintree.h` and `bintree.cpp`: This is the binary tree node template class from Section 10.3.
- `bagtest.cpp`: A simple interactive test program.
- `bagexam.cpp`: A non-interactive test program that will be used to grade the correctness of your bag class.

`bstbag.cpp` and `bintree.cpp` are actually template files. Do not include them in your compile list (they will be automatically used by the headers). The reason they are using the extension `cpp` instead of `template` is because the IDEs complain otherwise about what exactly to tell you about the files, and this is

the quickest solution without everyone needing to change their IDE settings. A `CMakeLists.txt` file is included to make it easier for your to get things up and running. (However, if your compiler does not have the correct version you may need to create a new `CMakeLists.txt` from scratch. You can ignore this if you are not using an IDE.)

Description

Note, much of this code is already written. You only need to fill in the parts marked with "STUDENT WORK". Begin by examining and understanding the existing code.

You may find writing additional functions helpful (particularly for writing a helper recursive function for another function). You may add additional functions, but you must still provide the functions being requested.

A bag does *not* have an order to the items. However, the binary search tree does have a structure. What this means is that the bag is still correct even if the items in the tree are rearranged. This also means that if multiple of the same value are added to the bag, and then you need to delete one of those values, it doesn't matter if you delete an older version of that value in the bag or a newer one.

Note the difference between a pointer parameter (e.g. `foo(binary_tree_node*& p)`) and a reference to a pointer parameter (e.g. `bar(binary_tree_node* p)`). Both pointers can be used to change the *object they point to* in memory, which will change that object for the calling function. However, if the *pointer* is changed to point to a new object the two methods will differ. In the case of the pointer parameter, this *will not* change the pointer from the calling function (since it is a copy of the pointer). In the reference to a pointer parameter case, this *will* change the original pointer in the calling function. In other words, in the non-reference version, the copy of the pointer points to the same object, but there are two different pointers to that object, and those pointers could be made to point to different things. This note is also true for functions *returning* a pointer or pointer reference.

I presented some binary search tree functions in class. You may use that code as a guide, but note that the implementation in this assignment will be written in a slightly different fashion. In particular, where the delete shown in class would use the minimum value of the right subtree as the value to replace the one being deleted, in this assignment you will use the maximum value of the minimum subtree. This accomplishes more or less the same result.

Optional (for extra 10% points)

Add the operators - and -=: For - operator, the value returned by `x - y` contains all the items of `x` with each instance of an element in `y` having been removed if it exists. That is, if `x` contains [1, 3, 4, 1] and `y` contains [2, 1, 3], the result of `x - y` will contain [1, 4]. Note, because `y` contained only a single 1, and `x` contained two 1s, the result still contains a 1. Also, for any element contained

in `y` which is not in `x`, `x` does not change. The statement `x -= y` leaves `x` in the state that would be equivalent to `x = x - y`. Note, you must consider the special cases of `x - x` and `x -= x`.