CS201L

Fall2021

Lab 9

For this lab we’re going to do a very simple class hierarchy showing the use of inheritance. We’re going to implement a simple container, a *Bag*, and then derive a child class with slightly different behavior but the same public interface.

A *Bag* is an unordered container. We’re going to use a Bag of integers for convenience, but we could use this for any data type. A Bag has the following interface:

* A default constructor that makes a new, empty Bag.
* An add\_item(int) method that takes an integer parameter and adds the item to the Bag. Duplicate items are allowed; there is no checking or trying to avoid duplicate items. This returns true if the item was added successfully, otherwise false.
* A remove\_item() method that returns an item from the Bag. The item returned is chosen from the Bag’s contents using any convenient method. A Bag does not promise any particular ordering to how items are removed. Think of reaching into a ‘grab bag’ and pulling out something. You get something that was in the Bag, and when you’re done, that item is no longer in the Bag.
* A delete\_item(const int&) method that removes a value from the Bag, without returning it. The Bag’s contents are scanned until an instance of the desired item is found; it is then removed from the Bag’s contents. The delete\_item() method returns true if an instance of the item was found and deleted, otherwise returns false.
* A delete\_all(const int&) method that removes *all* instances of the desired item from the Bag. It returns a Boolean value reporting whether any items were removed.
* A size() method that returns the number of items currently in the Bag. This is a const method.
* A clear() method that removes everything from the Bag.

You have a file of test data showing various operations that should be carried out. Each operation is on a line by itself and consists of a command; in some cases the command is followed by a second parameter, an integer. The commands are:

* A. Add item. This will be followed by the integer that should be added. Example: A 36
* R. Remove item. No parameter is specified because all a Bag promises is that we’ll get something from the Bag.
* D. Delete item. This is followed by an integer showing what should be deleted. Example: D 347
* Z. Delete all. (‘D’ and ‘A’ are already taken.) This will be followed by an integer showing what should be deleted. Example: Z 36

Put the Bag class into its own header and .cpp file, separate from your main program. Your program should read through the input file, producing an output file logging what’s happening at each step: What’s being added to the Bag, what was returned by Remove, and so forth. After all commands are printed, your program should also report in the log how many items are still in the Bag. Put this into a function that takes an istream, an ostream, and a Bag, all passed by reference, as its parameters.

DO THIS PART FIRST, TEST IT, AND MAKE SURE IT WORKS. Note that your Bag might return different items than mine, depending on implementation details. As long as the Bag returns an item that it had stored, and removes it appropriately, there’s no problem. The entire point of a Bag is that we don’t bother enforcing any particular discipline on what we return.

**Implementing a Bag:**

The simplest way to implement a Bag is to use a vector inside it. Make the vector protected, not private, since it’ll be convenient for the child class to have access to it.

Adding an item: The vector’s push\_back() method can be used to add items.

Removing an item: We can return the last item in the vector, copying it into a temporary variable before calling pop\_back() to remove it from the vector. Since we can return any item we please, it may as well be something convenient.

Deleting an item: Removing an item from the middle of a vector requires a bit of care, as we can’t just snip it out. BUT, we’re under no obligation to preserve any ordering. So the simplest way to remove an item is to search through the vector and locate an instance of the item, swap it with the last element in the vector, then call pop\_back() to remove it. Return true if this was done, false if no instance of the item was ever found.

Deleting all instances of an item: We can call delete\_item() repeatedly until it returns false.

Clear: The vector has a method for this.

Part 2: Deriving a child class.

Now we’re going to make a child class with slightly different behavior but the same interface. The OrderedBag is a child class, derived with public inheritance from Bag. Most functionality is the same, but an Ordered Bag always returns the smallest item it contains when remove\_item() is called. Go back to Bag.h and make the remove\_item() method virtual. Then, declare the child class. It doesn’t need any additional data or new methods, but it will need its own implementation of the remove\_item() method.

It is not necessary to keep the vector in an OrderedBag sorted. That’s much more work than is needed. Furthermore, any later operations are likely to destroy any ordering, so it would have to be re-done each time we remove something just in case something had changed. We just need to find the smallest item. Search through the vector, find the smallest item, and use a swap() to move it to the back of the vector, where the pop\_back() method can remove it conveniently.

Testing your OrderedBag: Modify your program to read through the input file twice. You wrote a function that logged what happened as you read through the file. Call that function twice, once with a Bag, once with an OrderedBag. (Obviously, you’ll need to close/clear/re-open the input file.) The OrderedBag should return different items than a Bag, but the same code ought to work for both.

Your program will be tested against a different data file with the same format.

Save all of your files, and either commit/sync to GitHub or delete the contents of Debug and upload them to Canvas as usual.

**Development notes:** This method of manually scanning the vector to find the smallest item is fine for a small collection such as we’re working with here. If we expect to be working with a larger data set, it becomes inefficient; we have to look at all of our data to find just 1 item, so it takes O(n) time for each removal. For a larger data set, we’d use a structure called a *heap* to find and return the smallest item in O(lg n) time, so finding the smallest item in a million would take 20 comparisons rather than a million. (Actually, finding it would be instant; removing it & replacing it with the next-smallest item would be O(lg n).) FaYou’ll learn about heaps in CS 303, Data Structures.

Also, here’s the cplusplus.com reference page on vectors:

<https://cplusplus.com/reference/vector/vector/?kw=vector>