Lab 9

21 March 2019

Today you'll start development on a project that provides a (small) library of classes to a potential user. Specifically, it will be a group of classes that store elements without duplication—a set.

Sets

What is a set? Its fundamental properties are that it

- contains elements,
- does not count or distinguish duplicates, and
- does not guarantee anything about their order.

That means that it can't, for instance, retrieve an element at a particular index, because indices imply order and sets don't (promise to) preserve order. Think about it, and in your notebook, write down the key methods that a `Set` class will have to have. There are three or four really important ones, plus a few that would be more optional. Make sure to mark which ones would be `const`.

Once you're pretty confident about your list, write a file `Set.h` that encodes this information in the form of valid C++ method headers. It will look pretty similar to some of the header files we've written in class (such as `ArrayList.h`), except that it won't have instance variables (ie no `private` section) and the methods won't be defined. We would like to make our `Set`s able to hold any type of element; recall that we can use templates for that. To make that happen, you just need to precede the class header with

```
template <class Thing>
```

and then use `Thing` as the name of the type the `Set` would hold, whenever you add a value or search for a value or anything like that. (Feel free to use a different name than `Thing`—in class we've used `ItemType` or just `T`. Up to you!)

Because our `Set` class is meant to define an interface, we want to mark its methods as "pure virtual": the implications of this we'll discuss in class, but the mechanics simply involve marking it `virtual` and setting the body to zero. That is, if you had written a method

```
  int getSomeValue() const;
```

you would mark it pure virtual by writing

```
  virtual int getSomeValue() const = 0;
```

Go ahead and do that (add `virtual` and `= 0` to each of your method declarations) in `Set.h`.

Then, write a simple test file called `test_VectorSet.u` that, for now, just `#include`s your `Set.h` file and has an empty test suite. Compile that file to confirm that your header has no errors.

Test cases

Now that we have a public interface, we can start planning our test cases. In your notebook (*not* yet in the `.u` file), describe a few useful examples (which will eventually become the test fixture). Then, write some sequences of method calls, using those examples, that collectively verify that a `Set` would correctly contain its elements, and does not count or distinguish duplicates.

Starting an implementation

Eventually, we'll write `Set` implementations that run efficiently and mimic the standard implementations, but before we worry about efficiency we have to aim for correctness. Our first implementation will be `VectorSet`, and will use the `vector` built in to C++ to store the data.[1] Its main inefficiency will be that when the user requests to add an element, it will have to check to see that it's not already in the set before adding it.

Edit a file `VectorSet.h` to start working on the class definition. The `VectorSet` will declare itself to be a subclass of `Set` by using the following class header:

[1]Note that a `VectorSet` object "has-a" `vector`, but "is-a" `Set`.

```
template <class Thing>
class VectorSet : public Set<Thing>
```

(again, feel free to use a word other than `Thing`). Inside the class, you'll start by making a private instance variable that is a `vector` to hold the data; and then for every pure virtual method in the `Set` definition, you'll write a stub method in the `VectorSet` definition (for now). Note: because it is a templated class, *all* the code for `VectorSet` will go in the `.h` file. Other than the "`: public Set<Thing>`", the `.h` file will be structurally quite similar to the `ArrayList.h` and `LinkedList.h` files we've been working on in lecture (and the `Bag` implementations before them).

Testing it

Now that you have the bare bones of an implementation, go ahead and type the test cases you wrote out earlier into the file `test_VectorSet.u` you created earlier. For reasons I'll explain tomorrow, I want you to use references here; in your fixture, you'll have lines that look like this:

```
Set<int>& example1 = VectorSet<int>{};
```

You'll probably have more than one, and some of them could be sets of string or whatever, and you should use names more descriptive than "`example1`". If you have something you want to do to some of them as part of setting up your test fixture, remember that you can create a `setup` block with arbitrary code. (Again, use our work on `ArrayList` and others as a model for the general structure of this file.)

Once you have your test file typed in, compile it and run it to confirm that everything compiles. If you run your test now, most if not all of the tests will still be failing—they're still just stubs!

Actually writing it

Now go back and start filling in the stub methods. At this point you can compile and test fairly frequently. The more frequently you do so, the easier it will be to find bugs that you inadvertently introduce.

Several of the methods will be quite short, and can simply call an existing method of `vector`! Don't write more than you have to.

Another implementation

Once you've finished `VectorSet`, write a different class called `LazyVectorSet`. From a user perspective, the results it gives should be exactly the same (but may take more or less time) as a `VectorSet`. The difference is that when the user requests to add an element, it *always* justs adds it (using `push_back`) to the internal `vector`, even if this creates duplicates—making this a cheap operation—but then it has to do a bit more work when it removes something and when it computes how many distinct elements are in the set.

Testing that one

The tests for `LazyVectorSet` should be identical to the ones for the other set, right? Copy your existing test file to one called `test_LazyVectorSet.u` and replace all occurrences of `VectorSet` (which should only be at the top of the fixture) with `LazyVectorSet`, and compile the test suite and run it. Debug your `LazyVectorSet` and keep testing it until it passes as well.

Handing in

Hand your code in by 4pm Wednesday, as `lab9` .

RUBRIC

1	Present in lab with preview stuff done
1	Appropriate readme
Set	
1	Method headers
$^{1}/_{2}$	pure virtual
$^{1}/_{2}$	compiles ♣
VectorSet	
1	Class definition as subclass ♣
1	Test suite tests correct behaviour (fail ok) ♣
1	Either add or contains is defined and correct
1	Add, contains, and remove are correct ♣
LazyVectorSet	
1	Class definition, subclass, add is correct ♣
$^{1}/_{2}$	Remove and size are effectively tested (fail ok) ♣
$^{1}/_{2}$	Remove is correctly defined

♣ indicates point is only available if the code compiles, with at least a stub for the relevant method(s).

Extra

Produce a table of times and a group of graphs à la Lab 8 to show the efficiency differences between `VectorSet` and `LazyVectorSet`.