Lab 12

13 November 2014

For today's lab, you'll glue together some of the code from several different sources—a few recent labs, and the last few days of lecture—to make another subclass of `Set`, this time implemented with a BST. But first, a few last tricks...

Vim FOTD: Macros

A macro is a bit like a function for an interface—a stored multi-command sequence that you can access with a quick keystroke or two. In Vim, you can define them in your `.vimrc` file, so go to your home directory and edit that. For this whole section, once you've made a change in the .vimrc, if you save it and then open vim on any file (perhaps in another window), you should see the results of your change.

The first thing to know is that internally, Vim refers to this process as "mapping" keys to macros. So in my `.vimrc` I can type

```
map ! :!
```

and then whenever I'm in command mode, if I press `!` it will act as if I've typed `:!` (which is what I do to run a general-purpose command-line command without leaving vim).

Sometimes, I need to insert characters into a text file that have special meanings to Vim even in insert mode. For instance, if you type a word and press `^W` (that is, Ctrl-W) while still in insert mode, it will normally delete that word. Putting a `^W` character into a file requires a special override that says to Vim: hey, take the next character I type literally instead of as a command. This override is `^V`, so to type a literal `^W` into a text file, I press Ctrl-V then Ctrl-W. (Try it, then delete the `^W` character you made.)

If I want to write a macro command that involves using `^V`, I need to enter that character literally, which means pressing Ctrl-V *twice*. To enter this macro into your `.vimrc`:

```
map ^V  ^V^F
```

you need to press the following key sequence (all in insert mode):

M A P space Ctrl-V Ctrl-V space space Ctrl-V Ctrl-V Ctrl-V Ctrl-F enter

Since `^F` is a command to scroll down one screenful, this mapping makes pressing the spacebar (in command mode) scroll down by one screenful, which I find more useful than simply moving to the right by one character (which is what spacebar does by default).

Two other quick facts before we wrap this up: Another representation of the Escape key is as Control-[or `^[`. (Try it—pressing Control and the left square bracket should take you out of insert mode exactly like Escape does.) And, a regular mapping will scan the macro expansion for further macros, so if you're mapping a key to an expansion that includes that same key, you should in general tell Vim to map it without remapping, to avoid an infinite recursion.

So, here's the visual form of the macro that, in insert mode when the left curly bracket is pressed, inserts both curly brackets at once:

```
  inoremap { {^V^[o}^V^[%a
```

Remember that to type in the control characters you need to press `^V` first (which means that you'll be pressing `^V` three times in a row on part of that line). You may find it instructive to deconstruct that line and see what each element of it is doing.

Assembling the parts you'll need

You'll need most of your files from Lab 9 (at the least `Set.h` and probably your testing code). (I think everybody got at least this much of Lab 9, but if not, you can copy my `Set.h` from the shared directory.)

If you got Lab 11 at least mostly working, you can get your `BinaryTree.h` and related files from there—convert it to be a tree that holds anything by replacing `char` with `Thing` and preceding the class with

```
  template <class Thing>
```

OR you may copy `BinaryNode.h` (a slightly modified version of the book's implementation from Chapter 16) from the shared directory. (But if you have a working `BinaryNode`, it's really better to modify and use that.)

You'll also want to grab your code for `inPrint` in Lab 11; and although you'll be modifying it, get `contains` as well.

You *may* want to bring in your `Card` stuff from Lab 10, particularly if you got the less-than operator working, but this is purely optional.

You'll also want to at look at your notes (and perhaps the board photos) about binary search trees that we've been doing in class the last few days.

Finally, by the time you get this far I should have paused the lab for a brief mini-lecture to finish working out pseudocode for the `add` method. You'll need that!

The task

As in Lab 9, you'll implement a subclass of `Set`, this time called `TreeSet`. Its implementation will use a binary search tree to store the elements, and like `VectorSet` it will not even store duplicate values.

Some of the code for this is already written, and just needs to be adapted to the current task! There is no need here to rewrite something from scratch BUT you should be sure that the source is indicated; comments like

```
  // adapted from class FooBarBaz written in lecture
```

or `@author` lines in class comments, e.g.

```
  /** Description of class
    * @author Don Blaheta
    * @author Your Name
    * @version updated date here (e.g. 13 Nov 2014) */
```

are how we do citations in the programming world.

When you first get started on this lab, your `remove` method should have the following body:

```
    cerr << "Not implemented yet." << endl;
```

until you get everything else pieced together and compiling and tested.

You don't need to worry about keeping the tree balanced; and you can assume that any `Thing` that is used with your `TreeSet` has a working `<` operator as well as `==`. (This is true of all the relevant built-in classes, such as `int`, `char`, and `string`, as well as many user-defined classes, such as our `Card` class (if you got that far).)

In addition, though not required by the `Set` interface, your `TreeSet` should have a friend `operator<<` function that prints out the contents by making a call to your `inPrint` function.

The big thing that is not already written for you in some form is `remove`. Think through what is involved in correctly removing from a BST; draw out a few examples to help you identify the different cases you'll need to deal with. Write some test cases based on those examples. Then, use your examples and test cases to help you write `remove` one piece at a time.

Hand in your work electronically as `lab12`. You should hand in whatever you have by 4pm on Wednesday so I can check on it, but the final version of the lab is due 4pm on Monday, 24 November.

RUBRIC

1	Present
1	Good readme
Gluing together existing code	
1	`TreeSet` is a subclass of `Set` ♣
1	`add` implemented from pseudocode
1	Code for `contains` and `inPrint` imported and templated ♣
$\frac{1}{2}$	`<<` prints contents of `TreeSet` with inorder traversal
$\frac{1}{2}$	`contains` exploits BST property
1	Test cases convincingly confirm that `add`, `contains`, `<<` work (fail ok) ♣
Implementing `remove`	
$\frac{1}{2}$	`remove` correctly removes values in leaves on multi-element trees
$\frac{1}{2}$	`remove` correctly removes value from one-element tree
$\frac{1}{2}$	`remove` correctly removes values in internal nodes
$\frac{1}{2}$	`remove` and destructor correctly `delete` nodes
1	Test cases convincingly confirm that `remove` works (fail ok) ♣

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