Lab 12

11 April 2019

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.

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 `BinaryNode.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` 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.

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. 10 Apr 2019) */
```

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`, by 4pm on Wednesday.

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 ♣
½ `<<` prints contents of `TreeSet` with inorder traversal
½ `contains` exploits BST property
1 Test cases convincingly confirm that `add`, `contains`, `<<` work (fail ok) ♣

Implementing `remove`

1 Test cases convincingly confirm that `remove` works (fail ok) ♣
½ `remove` correctly removes values in leaves on multi-element trees
½ `remove` correctly removes value from one-element tree
½ `remove` correctly removes values in nodes with exactly one child
½ `remove` correctly removes values in internal nodes with two children

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