Lab 12

8 April 2021

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 moving all code to the `.h` file, replacing `char` with `Thing`, and preceding the class with

```
  template <typename 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 heavily 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 (for demonstrating and testing the set).

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 `TSet`. Its implementation will use a binary search tree to store the elements, and like `VSet` 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. 2 April 2021) */
```

are how we do citations in much of 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 `TSet` 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).) Though not required by the `Set` interface, your `TSet` should have a working `operator==`, and a friend `operator<<` function that prints out the contents by making a call to your `inPrint` function—which will be used as a *private method* of the `TSet` class. You can assume that any `Thing` will also have a well-defined `<<` operator.

The first thing that is not already written for you in some form is `remove`, which we roughed out a basic design for in class but still has some implementation and testing gaps. Enter the test cases we did devise and add some to represent the ones we didn't write yet; implement it piece-by-piece, focusing on getting one TC to pass before working on the next. (Otherwise you're typing in dozens of lines all at once and sort of hoping there aren't any tricky bugs in it.)

Finally, adjust your code to test and implement `size`, `isEmpty`, and the `==` operator. They should work correctly in all cases and be *relatively* efficient, although for this lab `==` only has to guarantee $O(n \lg n)$ performance on typical trees, not $O(n)$. The `==` operator should be built to work with *any* `Set` as the right-hand-side operand.

Hand in your work electronically as `lab12`, by lab time next Thursday. (It would be due Wednesday but we have no classes Wednesday....)

RUBRIC

1	Present/engaged
1	Good readme
Gluing together existing code and design	
1	`TSet` is a subclass of `Set` ♣
1	`add` and `contains` implemented from pseudocode
½	`<<` prints contents of `TSet` with inorder traversal using templated `inPrint`
½	Fixtures with suitable examples
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
Other methods	
½	`size` and `isEmpty` are tested, correct, and $O(1)$
½	`==` is tested, correct, and $O(n \lg n)$ if both operands are `TSet` and balanced

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