

Binary Search, Binary Search Trees, and Visitors

Due: 3/15/2011 10:00pm

Portfolio Problems

Problem

In a class `Algorithms` design a more general version of the binary search algorithm shown in lectures. It should consume an instance of `java.util.List`, a `java.util.Comparator`, and an element of the list to be found, and returns the *index* of the element in the list. See the following web pages for necessary documentation:

```
http://download.oracle.com/javase/6/docs/api/java/util/List.html
http://download.oracle.com/javase/6/docs/api/java/util/Comparator.html
```

Because the `Algorithms` class operates on the `List` as a client (from the outside), your method should be parametrized by the type of elements in the list. If the given element is found your method should produce its index in the list. Remember that the data in the list must be sorted. If the element is not found, you should throw a `RuntimeException`.

In your examples, use instances of `ArrayList` to test your algorithm. Choose any type of data (e.g., `String`s, `Integer`s, etc.) to test your method and try it out with several versions of `Comparator`. Be sure to test that your method also throws the correct exception.

Pair Programming Assignment

8.1 Problem: Abstractions

Work out Exercises 34.11 through 34.15 from the textbook. Create a new project for these problems named `Assignment-08-1` in your pair's repository. Make sure your java files are in the `src` folder of the project.

For the drawing portion of the problems, use the JavaWorld library instead of the `idraw/draw` mentioned in the problems and extend `VoidWorld` for problem 34.15.

8.2 Problem: Traversals and Visitors

Start with the code given in the **BSTs.zip** file. You should have the following files:

- `Book.java`: a class that represents books, and includes two implementations of `Comparator<Book>`.
- `ABST.java`: an abstract class that represents a generic/parametrized binary search tree.
- `Leaf.java`: a class that represents a Leaf of a BST.
- `Node.java`: a class that represents a Node of a BST.
- `Examples.java`: contains several examples of BSTs and some tests.
- `ABSTVisitor.java`: an interface that represents a parametrized Visitor for generic BSTs allows us to define functions over BSTs without modifying the `ABST`, `Leaf`, and `Node` classes.

Create a new project for this problem named `Assignment-08-2` in your pair's repository. Make sure your java files are in the `src` folder of the project.

8.2.1 Problem: Traversals and Visitors

In this problem you will work with the `Traversal` interface (provided by the `tester` library) and see both its advantages and its shortcomings.

1. Run `tester.Main` for your project. Don't worry about the failing tests for now. Build additional examples of `Book` BSTs using comparison by `price` and/or `title`.
2. Add new tests the examples you have defined.
3. The `Node` class incorrectly implements the `getFirst` and `getRest` methods from the `Traversal` interface. Design the correct implementations for these methods so that all the tests pass.

4. In the `Algorithms` class design the method `totalPrice` that computes the total price of all the `Books` in the given `Traversal`.

5. In the `Algorithms` class design the method `makeString` that produces a `String` representation of all data in the given `Traversal`. You may add *separators* (i.e., commas, new lines, or semicolons) between the individual data items to make them easy to read.

8.2.2 Visitors

The `Traversal` interface is good if we want to walk through the elements of a data structure in some order (in this case specified by a `Comparator`) but we loose information about the organization of the data. For instance, we cannot design a method that computes structure specific values (e.g., the height of a tree) using `Traversal` interface.

Visitors are an object-oriented mechanism to allow functions to be implemented from outside of a class hierarchy, while providing access to the organization of the data. It is known as the *Visitor Pattern*, and we implement it for a class hierarchy by implementing an `accept` method for each class that invokes the corresponding method defined in the *Visitor* interface passing its fields (similar to our double-dispatch trick for implementing equality).

The programmer can implement new Visitors to add functionality over a class hierarchy that has been provided as a library. Each class that implements the *Visitor* interface represents the implementation of a new function for this class hierarchy. The `ABSTVisitor.java` file provides the Visitor interface for `ABST`s and an example implementation of the `countNodes` function through the class `CountNodes`.

1. Look at the `ABSTVisitor` interface and at `CountNodes` class. Add additional tests in the `Examples` class for your new examples defined earlier.

2. Design the class `ComputeHeight` that implements the `ABSTVisitor` interface by defining methods that compute the height of a binary search tree. *Hint*: use `CountNodes` as a guide, and see HtDP for a description of BST height.

3. Design the class `Contains` that implements the `ABSTVisitor` interface by defining methods that determine whether the given element

(a field of the class) matches any of the data items in a binary search tree.

Hint: Each `ABST` already has something that can tell you whether or not two data elements are the same (or otherwise ordered).