HW07

Homework 7 (Due 11:59pm Monday, November 8, 2021)

Submit via SVN

Preliminaries

This homework should be done in Full Java (using DrJava, IntelliJ, Eclipse, or a text editor and command line compilation and execution). The Functional Java language in DrJava does not work for more complex OO code such involving the visitor pattern. In this assignment, you will re-implement some of the functions on IntLists assigned in Homework 7 using the visitor pattern.

As before, your program must support the object-oriented formulation of lists of integers defined the composite class hierarchy where

- **`IntList`** is an abstract list of **`int`**.
- **`EmptyIntList`** is an **`IntList`**
- **`ConsIntList(first, rest)`**, where **`first`** is an **`int`** and **`rest`** is an **`IntList`**, is an **`IntList`**

The Homework Support files **`IntList.java`**, **`IntListVisitor.java`**, **`LengthVisitor`**, and **`IntListTest.java`** provide a starting point for your code. Feel free to edit these files and omit files that are not needed in this homework assignment.

Problems

Apply the visitor design pattern to define the methods below as visitor classes implementing the **`IntListVistor`** interface. Develop a JUnit test class, **`IntListTest`** to test all of your the methods in the **`IntList`** class and your visitor classes. Use the **`LengthVisitor`** example as a guide for defining your new visitor classes. To form your **`IntListTest`** class, you can augment the provided test class **`IntListTest.java`** to include test methods for each of your visitor classes. Confine your documentation to writing contracts for each visitor using **`javadoc`** notation (a comment preceding the corresponding definition) beginning with **`/**`** and closing with **`*/`** for each visitor class. Use the documentation of **`LengthVisitor`** in the respository as an example.

- (15 pts.) **`IntList reverse()`** constructs a list that is the reversal of **`this`**. Name your visitor class **`ReverseVisitor`**. **Hint**: this computation is faster and simpler if you introduce a help "method" (also a visitor).that takes an argument.
- (15 pts.) **`IntList notGreaterThan(int bound)`** returns a list of elements in this list that are less than or equal to **`bound`** . Name your visitor class **`NotGreaterThanVisitor`**.
- (15 pts.) **`IntList remove(int key)`** returns a list of all elements in this list that are not equal to **`key`**. Name your visitor class **`RemoveVisitor`**
- (15 pts.) **`IntList subst(int oldN, int newN)`** returns a list of all elements in this list with **`oldN`** replaced by **`newN`**. Name your visitor class **`SubstVisitor`**

- (20 pts.) **`IntList merge(IntList other)`** merges this list with the input list other, assuming that this list and other are sorted in ascending order. Note that the lists need not have the same length. Name your visitor class **`MergeVisitor`**. **Hint**: add a "method" **`mergeHelp(ConsIntList other)`** that does all of the work if one list is non-empty (a **`ConsIntList`**). Only **`mergeHelp`** is recursive. Use dynamic dispatch on the list that may be empty. Recall that **`a.merge(b)`** is equivalent to **`b.merge(a)`**. You should formulate help methods as visitors. The **`mergeHelp`** visitor should be placed in its own class **`MergeHelpVisitor`** like every other visitor.
- (20 pts.) **`IntList mergeSort().`** Leveraging the **`merge`** "method" you just wrote (as a visitor), write a **`mergeSort()`** that sorts an **`IntList`** formulated as a visitor. Name your visitor class (for **`mergeSort`**) **`MergeSortVisitor`**. Recall that you need to write a help function **`partition`** that splits a list approximately in two. Name your visitor class for this help function **`PartitionVisitor`**.

In summary, you need to write eight visitor classes: **`ReverseVisitor`**, **`NotGreaterThanVisitor`**, **`RemoveVisitor`**, **`SubstVisitor`**, **`MergeVisitor`**, **`MergeHelpVisitor`**, **`MergeSortVisitor`**, and **`PartitionVisitor`** and create tests for them all in **`IntListTest.`**

Coding Tricks

In writing solutions to this assignment, I discovered that the Oracle Java 8 JDK is better than any of the OpenJDK distributions. When I stated testing more complex programs, DrJava running on OpenJDK 8 JVMs would sometimes fail to re-enable the “test” button (which is disabled during compilation). In addition, it would occasionally hang during testing. In contrast, when running on the Oracle JVM (on which DrJava was developed using Java 2, 3, 4, 5, and 6, I experienced no glitches. I suspect that there are subtle differences in the support for concurrent threads (locking and scheduling) between OpenJDK JVMs and Oracle JVMs. I am begrudgingly going to use the Oracle JDK for all subsequent DrJava usage in this course. You can download the Oracle Java 8 JDK at https://www.java.com/en/; the download protocol requires that you create a free account (if you do not aleady have one) and agree that you will abide by the terms of the license which authorizes free personal use and free use for individual developers).

Of course, you can use other IDEs and versions of Java provided that you only use constructs available on Oracle JDK 8 (which is a superset of OpenJDK 8). The other minor advantage of the Oracle JDK 8 is that includes the JavaFX libraries which were dropped from all OpenJDK distributions and from Oracle JDK distributions starting with Java 9. You can get an open version of JavaFX (called OpenJFX I think) for OpenJDK 9 and above. Why do the JavaFX libraries matter? There is an implementation of the generic pair type **`Pair<A,B>`** in JavaFX (but not in the Java core libraries) in **`javafx.util.Pair`**. This class comes in handy when writing the code for the Partition method. You can also define your own Pair class in about 6 lines of code. I prefer to use standard libraries if available, but this case is borderline. You should be able to use raw versions of the JavaFX **`Pair`** class if you are avoiding generics. If you code your own, you can make it type specific.

Testing Tricks

In Racket, the **`equal?`** function performs structural equality. Java does not include such a built-in operation. For the **`IntList`** composite type, we overrode the inherited **`equals`** method (trivially defined in class **`Object`**) by an **`equals`** method that implements structural equality but it is slightly more complex than you might expect. Recall that the argument passed to **`equal`** has type **`Object`**. Hence, we have to worry about the class of the argument; the simple (and IMO best) definition of structural equality is to mandate that objects cannot be equal unless they are instances of the same class. Study the definition of the **`equals`** method in class **`ConsIntList`**. Unfortunately, we cannot write the body of this method as the

`return` of a boolean-valued expression, because Java does not support a notion of **`local`** or **`let`** at the level of expressions. So the body is of our **`equals`** override is an **`if`** statement with explicit **`return`** statements in the consequent statement and alternative statement. Notice that we still programmed in a functional style without any mutation.

To test the computations that yield results of composite type, we can either define structural equality over the composite type (as we did for **`IntList`**) or write an intelligible **`toString`** method for the composite type (which I strongly recommend for debugging purposes) and compare the **`toString()`** representations of the composite type. But beware that **`toString()`** equality may not imply structural equality and vice versa. You should always endeavor to make them agree. Moreover, you must explicity apply the **`toString()`** method to any expression of reference type that you are comparing to a String in Junit.

Extra Credit

If you do the assignment using generic types (**`List<T>`** instead of **`IntList)`**, you can earn an additional 50 points. In my recent experience, this is a lot of work.