Bucknell UNIVERSITY

CSCI 204 – Introduction to Computer Science II

Lab 8 – Linked Lists

1. Objectives

In this lab, you will:

- Learn basics of nested classes in Java
- Learn more about linked lists by adding extensions to a LinkedList<T> implementation
- Get more experience with JUnit

2. Introduction

The standard List ADT has been covered in class. For this lab, you will make a few modifications to the LinkedList<T> ADT as presented in class to make it more versatile.

3. Getting Started

As usual, you should set up your Eclipse workspace for today's lab. Import your files you need from

`~csci204/2011-spring/student/labs/lab08`

You should have two files, `LinkedList.java`, and a JUnit test file, `LinkedListTest.java`.

1. Adding the JUnit Library to Your Project

Add the JUnit library to your path by right-clicking on the project name lab09, and select **Properties**. In the right hand pane, select **Java Build Path**, then select the **Libraries** tab. Select **Add Library**, and then select **JUnit**. Specify version 4 of the JUnit library, then click **Finish** when complete.

Create a `readme.txt` file to store some answers to questions you will be asked throughout this lab. Put the `readme.txt` file in the main project folder so that it gets committed properly.

Be sure to commit everything into SVN, and then recommit your work as you are completing your exercises.

4. The `LinkedList<T>` class

The `LinkedList<T>` class provided is pretty much the implementation that we've covered in class. You will notice that it has the following essential ADT methods:

- `T get(int index)` – Returns the object at location index

- `void add(T obj)` – Adds a new object to the end of the list
- `void insert(int index, T obj)` – Inserts a new object at a specified location
- `void remove(int index)` – Removes an object from a specified location, and returns it
- `int find(T obj)` – Searches for a specific object

There are also two important helper functions:

- `public String toString()` – usual overridden method of the `Object` class that outputs the contents of the list
- `public T[] toArray()` – this is a useful method that converts the contents of the linked list into an array of exact size. You should take a moment to examine the implementation of this method. It comes in quite handy in some of the JUnit tests that are already given to you.

1. A brief note about nested classes in Java.

Since the `LinkedList<T>` implementation uses nested classes, it is worth discussing. The Java programming language allows you to define a class within another class. This type of class is called a *nested class*.

> NESTED CLASSES ARE DIVIDED INTO TWO CATEGORIES: STATIC AND NON-STATIC. NESTED CLASSES THAT ARE DECLARED STATIC ARE SIMPLY CALLED **STATIC NESTED CLASSES**. NON-STATIC NESTED CLASSES ARE CALLED **INNER CLASSES**.

Static nested classes are accessible outside of the outer class using the syntax:

```
OuterClass.StaticNestedClass object = new OuterClass.StaticNestedClass();
```

When using a *static nested class* inside the outer class as we do here, we can simplify the syntax to the familar:

```
StaticNestedClass object = new StaticNestedClass();
```

Inner classes can only be declared inside the outer class and have direct access to the methods and fields of the outer class. *Inner classes* are declared and instantiated using the familiar syntax:

```
InnerClass object = new InnerClass();
```

Nested classes provide a wonderful means of logically grouping classes together that are only used in one place. It also increases encapsulation by enforcing not only specific members of a class to be private, but also the entire definition of the nested type can be private as well.

As you scan through `LinkedList.java`, you should notice that there are TWO nested classes:

- `LinkedListIterator<T>` - an public *inner class* that defines an iterator on an instance of LinkedList.
- `Node<T>` - a private *static nested class* that defines a node that encapsulates an object in the list, and the reference to the next object in the list. It is private because we want to design our linked list class to hide internal representation details from the public API. In object-oriented terminology, this increases the encapsulation in the design of our linked list. There is no reason that someone using a linked list should ever need to

know implementation details of individual nodes, and how they are joined together. Therefore, this nested class is private. It is a static nested class (as opposed to an inner class) because nodes, in theory, could exist independently of a specific instance of a linked list.

We are making use of both types of nested classes in this design, though at this point, you should just be familiar with the concept.

5. Exercises

You'll be working with a JUnit test class to ensure that each method of the ADT is working properly. Work through each exercise below:

1. Exercise 1a: Implement `public int size()`

There is no method that returns the number of objects in the list. However, this number is readily available as a private member (instance field). Locate and complete the implementation of `size()`.

There is already a test developed to ensure it works properly. In `LinkedListTest.java`, the test is called `testSize()`. You need not do anything to the test. After you finish your `size()` implementation, run the JUnit test, and make sure that this test works.

2. Exercise 1b: Implement `public void clear()`

There is no method that allows you to clear the list of all objects. Locate and complete the implementation of clear. It should be a public void method that erases the list, and resets the count. (HINT: You do not need to iterate through the list and call remove on every object. Just initialize the internal references to null, and set the count to 0.)

3. Exercise 1c: Implement `public void addFirst(T obj)`

Currently, there is only one method that adds objects to the linked list. The `add(T obj)` method adds an object to the end of the list. It would be useful to have a method that adds an element to the beginning of the list instead. Call this new method addFirst(T obj). It should add the new object to the beginning of the list. NOTE: **You may NOT simply call the insert method with an index of 0**.

Implement a new JUnit test case in the LinkedListTest class right next to add() that tests that addFirst is working correctly. Call the new test case `testAddFirst()`

4. – Exercise 1d: Refactor `add(T obj)`

Rename this method to be called `addLast(T obj)`. Use the refactoring features of Eclipse to make your job easy.

5. Exercise 1e: Refactor `find(T obj)`

Rename this method to be called `findFirst(T obj)`. This method only retrieves the very first instance of the object found in the linked list. Verify that the test case for find still works. Also, you should rename the test case in `LinkedListTest` to be `testFindFirst()`. The test as designed places two instances of a known value at specific locations, and verifies that the first one is returned.

6. Exercise 1f: Implement `public int findLast(T obj).`

Implement a new method in `LinkedList<T>` called `findLast(T obj)` that will return the index of the last occurrence of the object in the list, or -1 if the object does not exist.

7. Exercise 1g: Implement a new test case for `findLast` .

Go to the file `LinkedListTest.java`, and add a new case for `findLast`, called `testFindLast()`. Model it after the `testFindFirst()` test case. Verify that your `findLast` method works correctly.

At this point, every test case should pass.

6. Exercise 2 – Implement a main() program to test LinkedList

Write a main program that does the following:

1. Instantiates a new `LinkedList` instance of Integer objects.
2. Fill the list with `SIZE` random integers between 1 and 10. Define `SIZE` to be a constant in `main()`, initialized to 500000. Use `java.util.Random` to generate the random numbers. (See the API.)
3. Write a loop that will accumulate the frequency (or tally) of the occurrence of each integer between 1 and 10. NOTE: This can easily be done in one pass through the loop. (HINT: Use an array to accumulate the tally! How?) Use must use an `Iterator` to iterator through your list. Either instantiate the iterator using the `iterator()` method, or use a for-each loop.
4. Output the frequency of occurrence of all 10 numbers, along with the fraction of the number of times each occurred next to it as a percentage. Format your output nicely. Be sure that the sum of your tally adds up to 100.

Run your main program, and store your output in a new file called `readme.txt`.

7. When you are done

Submit everything to SVN when complete