CodeU Coding Exercises - Session 2

Lists, Trees, and Graphs

Please Help - Share your Feedback:

Session 2 Feedback Form

Exercise 1 (DO THIS OR DO Exercise 2): List Filtering

This is the same basic problem as Exercise 1 in Session 1, but using Lists instead of Arrays. If you need a refresher on Lists, here is one place to start: http://tutorials.jenkov.com/java-collections/list.html.

Use the following class “FilterList” as your starting point, and provide an implementation for “evens”.

```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

public class FilterList {
    // Write a function named "evens" that takes as input a
    // list of Integer (almost but not quite int) and returns
    // a new list of ints containing only the even elements
    // of the input.
    public static List<Integer> evens(List<Integer> input) {
        // Here are some reminders:
        //
        // You can find input's length using input.size().
        //
        // You can find the remainder of a division using %. For instance,
        // 2 == 11%3 and 1 == 25 % 4.
        //
        // You can declare a new list named "clown" of length n with the
```

```
        // syntax:
        //
        // List<Integer> clown = new ArrayList<Integer>(10);

        // STUDENTS, WRITE CODE HERE.
    }

    public static void main(String[] args) {
        List<Integer> test1 =
            new ArrayList<Integer>(Arrays.asList(8,6,7,5,3,0,9));
        List<Integer> ans = evens(test1);
        // Expected output: 8, 6, 0
        for (Integer n: ans) {
            System.out.print(Integer.valueOf(n) + ", ");
        }
        System.out.println();

        List<Integer> test2 =
            new ArrayList<Integer>(Arrays.asList(2,7,18,28,18,28,45,90,45));
        ans = evens(test2);
        // Expected output: 2, 18, 28, 18, 28, 90
        for (Integer n: ans) {
            System.out.print(Integer.valueOf(n) + ", ");
        }
        System.out.println();

        // STUDENTS: ADD YOUR TEST CASES HERE.
    }
}
```

Exercise 2 (DO THIS OR DO Exercise 1): Rotated List

You will write a function **isRotation()** that takes two lists as inputs and returns true if the first list is a rotation of the second list. For example, if **isRotation()** is
given the inputs (1,2,3,4) and (3,4,1,2), it should return true, and for the inputs (1,2,3,4) and (3,4,2,1) it should return false.

This class also includes the framework for a test framework that you should fill in and
use when you write your tests.

```
import java.util.Arrays;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

public class RotatedList {

    public static boolean isRotation(List<Integer> list1, List<Integer> list2) {
```

```
    // STUDENT: PROVIDE IMPLEMENTATION
}

public static class Test_IsRotation {
    // Test framework for isRotation()
    // Instead of calling isRotation directly, call expectFailure or expectSuccess
    // with the desired input arguments.  Call init() to start collecting results
    // data for a series of tests, and call report() to print a short report on the
    // outcomes so far.
    private static int failCount;
    private static int successCount;

    public static void init() {
        failCount = 0;
        successCount = 0;
    }

    public static void expectFailure(List<Integer> a, List<Integer> b) {
        // STUDENT: PROVIDE IMPLEMENTATION
    }

    public static void expectSuccess(List<Integer> a, List<Integer> b) {
        // STUDENT: PROVIDE IMPLEMENTATION
    }

    private static void report(List<Integer> a, List<Integer> b, boolean got) {
        // Helper routine that may be used by expectFailure and expectSuccess.
        // It should be called only if the unexpected occurs. In other words,
        // if all tests work as expected, then the program will be silent.

        // STUDENT: PROVIDE IMPLEMENTATION
    }

    public static void reportSummary() {
        if (successCount == 0) {
            System.out.println("Summary: all " + failCount +
                               " tests FAILED!");
        } else if (failCount == 0) {
            System.out.println("Summary: all " + successCount +
                               " tests SUCCEEDED!");
        } else {
            System.out.println("Summary: " + successCount +
                               " tests succeeded, " + failCount +
                               " tests failed.");
        }
    }
}
```

```
    public static void main(String[] args) {

        Test_IsRotation.init();

        List<Integer> list1, list2;
        list1 = new LinkedList<Integer>(Arrays.asList(1,2,3,4));
        list2 = new LinkedList<Integer>(Arrays.asList(3,4,1,2));
        Test_IsRotation.expectSuccess(list1,list2);
        list1 = new LinkedList<Integer>(Arrays.asList(1,2,3,4));
        list2 = new LinkedList<Integer>(Arrays.asList(3,4,2,1));
        Test_IsRotation.expectFailure(list1,list2);

        // STUDENT: ADD TEST CASES HERE

        // Check performance
        int N = 10000;
        Integer[] a1 = new Integer[N];
        Integer[] a2 = new Integer[N];
        for (int i = 0; i < N; ++i) {
            a1[i] = 0;
            a2[i] = 0;
        }
        // (STUDENT: WHY THIS CASE?)
        a1[0] = 1;
        a2[N/2] = 1;
        list1 = new LinkedList<Integer>(Arrays.asList(a1));
        list2 = new LinkedList<Integer>(Arrays.asList(a2));
        long start_time, end_time;
        start_time = System.nanoTime();
        Test_IsRotation.expectSuccess(list1, list2);
        end_time = System.nanoTime();
        System.out.println("10000 elements: " +
                           (end_time - start_time) + " nsec");

        // STUDENT: DESIGN SOME BETTER TESTS FOR EVALUATING PERFORMANCE

        Test_IsRotation.reportSummary();
    }
}
```

Exercise 3 (DO THIS): Flatten a Tree

In this exercise you will define a **BinaryTree** class. Your binary tree should be composed of **BinaryTreeNode** objects. Each BinaryTreeNode instance has a reference to two child nodes (traditionally called left and right, or leftChild, rightChild) and a single string for its “payload”. The left and/or right references may be null. If both are null, the node is a “leaf” node. Your BinaryTree instance should reference a single “root” node (which may be null).

Your primary method will be called **flatten**. It will take the current binary tree, starting with the root, and produce a List containing the data from all the nodes of the tree. You will also need one or more methods to build the tree to test your "flattener".

Exercise 4 (OPTIONAL): Build and Query a Genealogical Tree

In this exercise you will create an AncestorDatabase class and implement some useful functions for it. The AncestorDatabase consists of zero or more AncestorTree objects. Each AncestorTree object has a single individual at the root. Each individual has a mother and/or a father (one or both may be unknown for a given individual). Each mother and father also has a mother and a father, and so forth, as deep as you want to go.

Please do the following:

1. Design the **AncestorDatabase** and **AncestorTree** classes.
2. Implement an **addPerson** method. It should accept three strings, for person, mother, and father. Use the string "UNKNOWN" if the mother or father are unknown. Assume that each name string is unique. For example, if you process **addPerson("John", "Mary", "Jack")** and **addPerson("Jack", "Martha", "Fred")**, then Fred must be John's grandfather.
3. Implement an **isAncestor** method. It takes two names and returns a positive integer if the second name is an ancestor of the first. The integer indicates the number generations between the two. For example, **isAncestor("John", "Martha")** should return 2. If person 2 is not an ancestor of person 1, return -1.
4. Implement an **isDescendant** method. It takes two names and returns a positive integer if the second name is a descendant of the first. The integer indicates the number generations between the two. For example, **isDescendant("Jack", "John")** should return 1. If person 2 is not an ancestor of person 1, return -1.

In your main routine (or a test framework) write a series of addPerson() calls to populate your database. Then write a series of queries to verify that the database is properly constructed.

This is strictly optional: you might choose to add methods to enter people and queries using console I/O.

Also strictly optional: develop an extension of this system to return the chain of relationships between two persons, or name the actual relationship (e.g., **listRelationship("Jack", "Fred")** might return **"Fred->Jack->John"**, or **describeRelationship("John", "Fred")** might return **"Fred is the grandfather of John"**).

Exercise 5 (OPTIONAL): Collatz Sequences Revisited

If you did the Collatz exercise last session you realized that the sequences had some very interesting behaviors. Some of you also started thinking about performance, and/or how far you could go with the computations.

Last session's exercise looked only at the length of the sequence across a range of inputs. This session we will keep the results of computing each sequence in a graph.

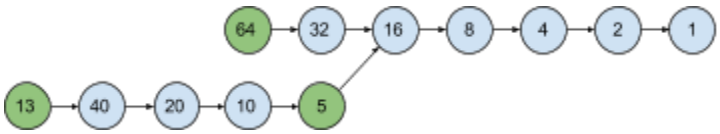
Consider the following three sequences:

```
collatz(64) -> 64, 32, 16, 8, 4, 2, 1

collatz(5) -> 5, 16, 8, 4, 2, 1
```

```
collatz(13) -> 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Note that when two sequences hit a value in common, the rest of the sequence is identical:



Note also that we've computed the Collatz sequences for a lot of other intermediate values. If we save what we've already computed, then if we compute collatz(13) first, we have already computed collatz(5). If we compute collatz(5) first, then when we later compute collatz(13) we can stop as soon as we reach 5. If we compute either, we've already computed collatz(16) which is used by collatz(32), collatz(64), etc. The individual computations are not expensive, but in more realistic situations where the computation for each step is expensive this technique can make a huge difference.

```
class CollatzGraph {
  // Start with loopCount from Session 1. Every time it generates
  // a new member of the sequence, add a node for the integer to the
  // graph (unless a node for that integer is already present).
  // If such a node is already present, loopCount should not have to
  // generate the rest of the sequence. However, it still must return
  // the length of the sequence starting from x.
  // NOTE: loopCount is no longer static.  You will need to use one
  // or more instance variables to store information (the current
  // graph, for example) between calls.
  int loopCount(int x) {
    // STUDENTS: FILL IN CODE HERE!
  }

  // This method sets the initial state of the graph and any other
  // instance variables of the class.
  // HINT: This is a good place to create an initial node for the integer 1.
  // If you do this, then your termination check will be a lot simpler!
  void initialize() {
    // STUDENTS: FILL IN CODE HERE!
  }

  // Using loopCount, fill in the function maxLoop so that it returns
  // the maximum sequence length for any sequence that starts with a
  // number greater than or equal to x and less than y.
  int maxLoop(int x, int y) {
    // STUDENTS: FILL IN CODE HERE!
  }

  public static void main(String[] args) {
    CollatzGraph graph = new CollatzGraph();
    graph.initialize();
    System.out.println(graph.maxLoop(1,1000));
```

```
    // STUDENTS: maxloop and loopcount have constructed a graph that
    // contains the collatz sequence for every integer from 2 to 1000
    // as well as for every integer contained in those sequences. Think
    // of a question about the graph and write a query that answers it.
    // Some interesting (IMO) questions that come to mind are:
    //   (1) how many nodes are in the graph?  How fast is it growing?
    //   (2) what is the largest integer in the graph?
    //   (3) what is the smallest integer that is not in the graph?
  }
}
```

Exercise 6 (OPTIONAL): Sparse Matrix Revisited - Sparse Matrix Input

Last session you wrote a function to output a sparse matrix. Now you will write the input function. If you didn't do the sparse matrix output, that's OK, you can still do this - just review last week's exercise, or do it now, or team up with someone who's already done it :).

Write a function that reads from the console, scans a sequence of entries of the form "[x, y]: value" and builds the corresponding matrix. Each line should have the following format (or you can modify the syntax if you wish:

```
[row_index, column_index]: value
```

For example, for the following input:

```
[1, 1]: 6
[2, 0]: 8
[2, 3]: 4
```

the resulting matrix (in memory) should be:

```
| 0 0 0 0 |
| 0 6 0 0 |
| 8 0 0 4 |
```

There are many ways of inputting information from the console. Consider using the Scanner class (java.util.scanner). If you have the sparse matrix output routine, you can combine them to output the matrix that you've input (or re-input the matrix you've output). Think about overall testability. You can use diff to verify console input vs. output, but you may also need a method to compare two matrices for equivalence.

Hint: there is one thing your sparse matrix input method will have to handle that was not an issue for the output method. You will have to figure out what it is and how to handle it.

Exercise 7 (OPTIONAL): Real Sparse Matrices

You know how to input and output sparse matrices, but in memory you've been working with full-sized matrices. Until now. Create a class (e.g., SparseMatrix) that supports some basic matrix operations but uses a denser internal format. If you think about the 3x4 sample matrix used in the previous exercises, it requires a minimum of 12 integers for storage (not counting overhead). The sample contained 3 non-zero values. If you use an internal representation of a list of objects, each containing a row index, a column

index, and a value, then you can store the same matrix in 9 integers (again ignoring overhead). OK, not so hot, but if the matrix had been 1000x1000 and contained 1000 non-zero values, we're talking about 1000000 vs 3000 integers, so the sparse representation uses 0.3% of the space of the full representation (ignoring overhead again, but it is hard to imaging the overhead canceling out the space savings).

Your internal representation of the matrix should be a list of rows, and each row will contain a list of zero or more points. Your SparseMatrix class should have the following methods:

- A **constructor** that creates a new empty matrix of a given "virtual" size. The matrix cannot be resized after creation (yet)
- A **constructor** that creates a copy of the input matrix.
- **int rowDimension()** - return the number of rows in the matrix
- **int columnDimension()** - return the number of columns in the matrix
- **int size()** - return the number of elements in the matrix
- **double density()** - compute the space requirements of the matrix as a fraction of the non-sparse representation. An empty matrix should return 0.0, and equal space should return 1.0 (can it ever be > 1.0?).
- **void set(int row_index, int column_index, int value)** - set the element at [row_index, column_index] to value.
- **int get(int row_index, int column_index)** - return the current value at [row_index, column_index]
- **SparseMatrix add(SparseMatrix m2)** - perform a matrix add. return (this + m2).

Also, port your input and output functions to your SparseMatrix class.

Although your basic representation is a "list of lists", you will have a number of design decisions to make. Think about your options and make deliberate choices. You are probably already thinking that a list of lists may not be the most efficient representation, but stay with that for now. If you want more to do, add two more methods to your class:

- Matrix **extract(int x_origin, int y_origin, int x_size, y_size)** - extract a rectangular patch from the current SparseMatrix.
- Matrix void **replace(int x_origin, int y_origin, Matrix replacement)** - overwrite a rectangular patch in the current SparseMatrix from the input Matrix.

Finally, think about testing. In particular, how will you verify that SparseMatrix::add() is correct?

Please Help - Share your Feedback:

Session 2 Feedback Form