CS W3137 Spring 2014, Assignment 2. DUE: 2/25 in class - extended from 2/20

Non-Programming Problems. If you are asked to write a program or some code, note that you do not have to write actual code. Pseudo-code or use of the Java methods mentioned in the textbook is allowed - it can make this very simple. You may also code the solutions if you wish, but it is not required.

1. (28 points) Assume a simple Linked List data structure such as the one below (NOT a Java Collections API Linked List). Write brief Java methods to do the following:

   (a) return the size of the linked list

   (b) print the linked list

   (c) test if a value x is contained in the linked list

   (d) add a value x if it is not already in the linked list (add at the end of the list)

   (e) remove a value x if it is contained in the list

   (f) reverse the order of the items in the linked list

   (g) Given two lists, Li and L2, create a new list L3 that contains the interesection (common data elements) of L1 and L2.

```
//simple LinkedList class
public class SimpleLinkedList {
    protected Node header;
    public SimpleLinkedList() {
      header=null;
    }
       ...methods go here...
}
public class Node{
   protected Object data;
   protected Node next;
   public Node(Object x, Node n){
     data=x;
     next=n;
   }
   public Node(){
       data=null;
       next=null;
   }
}
```

Programming Problems

1. Maze Search. You will be given a maze file and using Breadth-First-Search, you will find a solution to the maze.

   A maze is an 8x8 grid of cells. Each cell has 4 neighbors: North, East, South, West. Input is 8x8 ascii matrix file, with 0 for empty cell, 1 for obstacle, S for Start, G for Goal, and spaces between entries. Example ascii Maze File:

```
0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0
0 1 1 0 1 G 0 0
0 0 0 0 1 1 1 0
0 0 0 0 0 1 0 0
0 S 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
```

   There is a zip file *MazeHw2.zip* that has some skeleton java code which will get you started. You can access this file from the class homework site:

   `http://www1.cs.columbia.edu/~allen/S14/homework.html`

   This zip file contains the following files:

   (a) MazeShell.java - this is a "skeleton" program that will compile, and it has stubs in the code for the parts you need to write to solve the maze, as well as the GUI interface.

   (b) DrawingCanvas.class - you need this class to compile MazeShell.java. It contains the GUI drawing classes. Youmay use this set of GUI methods (easier) or use your own drawing methods.

   (c) DrawingCanvas.html - documentation for the DrawingCanvas methods.

   (d) maze1.data, maze2.data, maze3.data - sample maze files

   (e) MazeSolutionObs.jar - this is an **executable jar file** that contains a solution to the maze problem so you can see how the program sholdl work. The program has 1 argument which is the maze file name: Execute it as:
   java -jar MazeSolutionObs.jar maze_file_name

   What you need to do:

   (a) (12 pts.) Read in the maze file and display the maze with START, GOAL and OBSTACLES. Use buttons in the display window to control this.

   (b) (20 pts.) click a button to do a breadth-first-search using a queue from the START cell to the GOAL cell, calculating and displaying the distances from the START cell. You may only travel North, East, South or West from a cell, unless one of these directions is an obstacle or the edge of the grid.

(c) (10 pts.) Click a button to find the minimum cost path and display it. Note these‘ paths are not unique, there may be more than one minimum cost path. Cost is simply the number of cells traversed. If no path exists report that in the GUI.

2. Polynomial Operations Using Linked Lists

NOTE: you may not use any of the Java collections classes in this exercise. You must use your own linked list classes, not any of the Java collections classes. Classes in the textbook or from class notes are allowed.

(30 points) Polynomials can be represented by a linked list. Each node of the linked list will correspond to a term of the polynomial. This will allow us to represent polynomials of an arbitrary length. In this exercise, you will write a program to create, manipulate and display polynomials.

A possible Java class for a polynomial term might look like this:

```
public class PolyTerm {
private double coef;
private int exponent;

  PolyTerm()
  {
    coef=0.0;
    exponent=0;
  }
  PolyTerm(double c, int e)
  {
    coef=c;
    exponent=e;
  }
  public void setPolyTerm(double c, int e)
  {
    coef=c;
    exponent=e;
  }
  public int getExponent()
  {    return exponent;  }
  public double getCoef()
  {    return coef;   }
    public String toString()
  {  return coef + " x^" + exponent; }
}
```

These PolyTerm objects form the nodes of a linked list that represents a polynomial. Since we want to manipulate many polynomials (e.g. add polynomial_1 to polynomial_2 resulting in a new polynomial_3), we can allocate an array of linked lists, where each linked list is a polynomial. The array index is the way we identify each of the polynomials (polyarray[0], polyarray[1], etc.)

```
linkedlist[] polyarray = new linkedlist[MAX_POLYS];
```

BE CAREFUL! Besides declaring the array above, you will still have to initialize each linked list using your linked list object class constructor.

Each polynomial will be indexed by its entry in the polyarray, and the contents of that array position will a LinkedList object.

Important: To make the process efficient, we can store the polyterms in order of DECREASING exponent. Then, as we insert terms, we know where in the list to put them.

What you need to do: Below is a sample interface to do polynomial operations:

(a) **Input** - Enter the position in the poly array to store this polynomial, and the number of terms in the polynomial. Then, perform a loop for the number of terms, entering coefficient and exponent pairs. To enter the polynomial $5.1x^4 + 6x^3 - x + 8$ (4 terms) in position 0 of the poly array we would enter at the terminal: (sample menu of choices for user)

```
Please enter what you want:
 i for input
 a for add
 s for subtract
 m for multiply
 e for evaluate
 p for print
 q for quit
i
input: enter index number of polynomial and how many terms
0 4
enter coef and exponent for term 1
5.1 4
enter coef and exponent for term 2
6 3
enter coef and exponent for term 3
-1 1
enter coef and exponent for term 4
8 0
5.1x^4 + 6.0x^3 + -1.0x^1 + 8.0x^0
```

Each time we get a coefficient-exponent pair, we create a term, and insert it into the linked list, ordered by decreasing exponent value. You will need to write an **insertInOrder** method that will scan the linked list and find the appropriate spot to enter each term in the list by decreasing exponent value. When you create a polynomial, you should be able to input the terms in any order: the **insertInOrder** method will find their proper place in the list by decreasing exponent.

(b) **addPoly(LinkedList L1, LinkedList L2, LinkedList L3)** - Once you have input the polynomials, you will have to perform operations on them. To add 2 polynomials, simply specify the indices of the linked list polyarray for the two addends and an index for the linked list that will hold the result. The method above will add the polynomial at polyarray position L1 to the polynomial in polyarray position L2 and create a new polynomial at polyarray position L3.

You might think the way to do the add operation is to process both lists together from the first term to the last (remembering that the terms are ordered by decreasing exponent). If $p1$ is an iterator that points to the first term of polynomial 1, $p2$ is an iterator that points to the first term of polynomial 2, we do the following:

i. If the exponent in the term pointed to by $p1$ is larger than the exponent in the term pointed to by $p2$, we create a new term that is identical to the term pointed to by $p1$, and **insert** this new term into polynomial 3. Now advance the pointer $p1$ to the next term of polynomial 1.

ii. If the exponent in the term pointed to by $p1$ is smaller than the exponent in the term pointed to by $p2$, we create a new term that is identical to the term pointed to by $p2$, and **insert** this new term into polynomial 3. Now move the pointer $p2$ to the next term of polynomial 2.

iii. If both exponents are equal, add the coefficients, and **insert** a new term with the common exponent and coefficient sum into polynomial 3. Now move both pointers $p1$ and $p2$ to the next term.

iv. If one of the polynomials ends, **insert** each of the remaining terms in the other polynomial into polynomial 3.

However, we really don't have to do this. All we have to do is an insertInOrder: simply insert every term in both polynomials into the new polynomial. insertInOrder will automatically add co-efficients if the exponents are the same! So the add operation is essentially for free once we write insertInOrder.

(c) **subtractPoly(LinkedList L1, LinkedList L2, LinkedList L3)**

Same idea as in the addpoly method: subtract polyarray[L2] from polyarray[L1] and leave the result in polyarray[L3].

(d) **multPoly(LinkedList L1, LinkedList L2, LinkedList L3)**

Same idea as in the addpoly method. Make sure you actually do a multiply operation, rather than doing iterative addition as shown in class.

(e) **evalPoly(LinkedList L1, int value)**

This will evaluate the polynomial in position L1 using value as the value of the variable in the polynomial.

(f) **printpoly(L1)** - This method will print out the polynomial in polyarray[L1]. This is very useful in debugging your code. Everytime you do an operation on a polynomial, you can print it out to see if it worked. This method already exists in the Weiss LinkedList class, but you must create a toString method in the PolyTerm class to print out the PolyTerm objects.

Extra Credit 1: Cool User Interface, 4 points - This is something you need to design yourself. You can use the sample menu in this handout as a guideline (no extra credit) or use your own design if you want - make it user-friendly. Also make it defensive: don't allow commands that don't exist (i.e. illegal input) etc. Extra points will be given for a cool interface!

Extra Credit 2: Polynomial Divide, 4 points - Implement a polynomial divide method. dividePoly(L1, L2, L3, L4) divides poly L1 by poly L2, leaving the Quotient in L3 and the Remainder in L4.