

# COMP 282 Summer 2020,

## Homework 01 Due: Monday, 7/20/2020

### Linked List

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, L1 and L2, create a new list L3 that contains the intersection (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;
    }
}
```

## 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.

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 be 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 coefficients 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.
- 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. Every time 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.

Submit **EVERYTHING EVERY TIME**:

With **each** submission, including **the current version of all work products to-date, even if incomplete, even if un-changed**. This includes:

- Source Code with complete explanation and comments
- Executable program
- Entire software project files if you use any IDE
- Picture files that show you successfully compile your Java program with a set of output results.

Software Project files (example: if MS Visual Studio is used, include the ENTIRE project folder containing all project files – such as .vcxproj.)

**Note:** All deliverables must be put in a folder, then Zip the folder and name it as HW01\_your CSUN ID and submit it via Canvas.

If you plan to use GitHub, you can only send me the URL of your project. Make sure it is accessible.