THE HONG KONG UNIVERSITY OF SCIENCE & TECHNOLOGY
Department of Computer Science

COMP 171 : Data Structures and Algorithms

Fall 2005
Programming Assignment 1
Due date and time : October 4, 2005, 23:59

---

## 1 Rules, Submission procedures, and Grading

**Rules and submission procedures.** There are two programming tasks in this assignment: problem A and problem B. You need to produce ONE ZIPPED FILE FOR EACH PROBLEM. Each zipped file must:

- contain your source files and documentation (including your acknowledgments).
- be completely self-contained; your program can only refer to the normal C++ libraries.
- compile and run using `VC++` under `Windows`.

A good programming practice is to keep separate `.h` files and `.cpp` files in the zipped file, but this is not a grading criteria. You MUST name your zipped files as indicated below:

| Task Name | Name of Zipped File |
|---|---|
| Problem A | `pa1A.zip` |
| Problem B | `pa1B.zip` |

**Grading.** Your solution will be graded on:

- Correctness (80 percent).
  - When your program does not compile, you may receive partial credits at our discretion.
  - When your program does not run correctly, we will inspect the results and your code visually and you may receive up to half marks for that operation.
- Programming style (10 percent).
- Program documentation (10 percent).

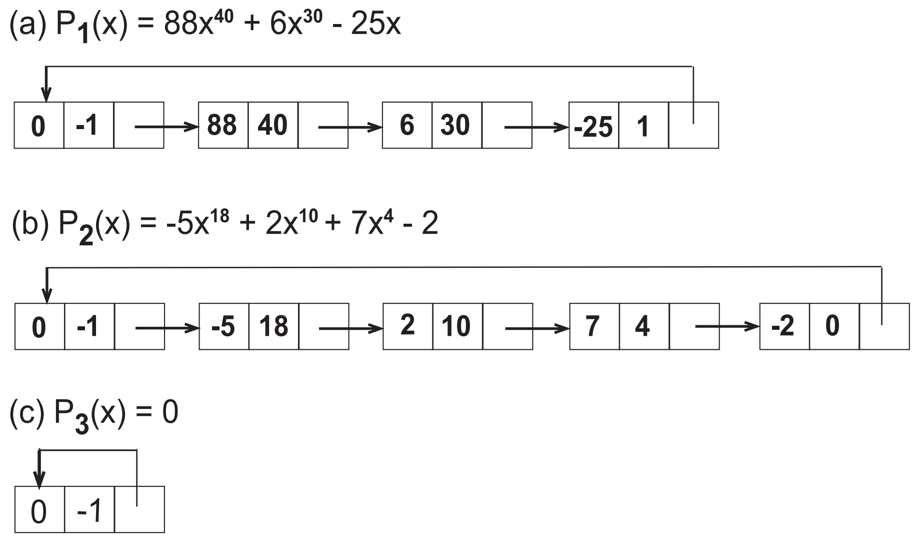
  This includes your acknowledgments. You must explicitly acknowledge the resources (books, web sites, and so on) that you have consulted and the people you have discussed the assignment with (if they are other COMP 171 students, state their full names and student ids).

  If you have not consulted anyone, then state that explicitly as follows: *The work included in this program is all my own work.*

## 2 Problem A: Univariate Polynomial Manipulation

**Task description.** Design, implement and test an ADT, `Polynomial`, that provides some of the basic operations for univariate polynomials. For example, $P(x) = 3x^4 - 7x + 18$ is such a polynomial.

Your ADT `Polynomial` should have a data member belonging to the class `CircularList` which keeps the terms (term consists of coefficient and exponent) in the polynomial in a circular linked list. The circular list representation of a polynomial has one node for each term that has non-zero coefficient. The terms are in decreasing order of exponent and the head node has its coefficient and exponent field equal to 0 and -1 respectively. The following figure gives some examples.


(a) P1(x) = 88x40 + 6x30 - 25x
0 -1
88 40
6 30
-25 1
(b) P2(x) = -5x18 + 2x10 + 7x4 - 2
0 -1
-5 18
2 10
7 4
-2 0
(c) P3(x) = 0
0 -1

You must use the three classes `Node`, `ListIterator`, and `List` (should be modified to adapt the change to circular structure) covered in class to do the job:

- You can modify and/or simplify the specific details of these three classes, but you must adhere to the principles of hiding `Node` from the public and accessing via list iterators instead of explicit node pointers.
- In particular, `Node` must be hidden from your `Polynomial` class.

*Note that it is a good programming style to destroy nodes when they are no longer needed.*

The ADT `Polynomial` should support the following operations. Note that some operators should be overloaded to make your code more readable. You may add any other public/private member functions that you think are necessary.

(a) `Polynomial()`
- Create the zero polynomial, that is $P(x) = 0$. `Polynomial()` is the class default constructor.

(b) `friend istream& operator>>(istream&, const Polynomial&);`
- Read in a polynomial from `cin`. Each polynomial has the following form:

$$c_1\ e_1\ c_2\ e_2\ \cdots\ c_m\ e_m\ 0\ -1$$

where $c_i$ and $e_i$ are integers denoting the coefficient and exponent of the $i$th term, respectively. The last pair $0 - 1$ denotes the end of polynomial.
You can assume that the exponents are in decreasing order; that is $e_1 > e_2 > \ldots > e_m \geq 0$, and there is no zero coefficient in the input; that is $c_i \neq 0$ for all $i$.

(c) `friend ostream& operator<<(ostream&, const Polynomial&);`
- Output the polynomial to `cout`. The output format should be the same as the input format. That is, the exponents should be in decreasing order and all coefficients are non-zero. Also it should end with the pair $0\ -1$.

(d) `friend Polynomial& operator+(const Polynomial& p1, const Polynomial& p2);`
- Add the two polynomials `p1` and `p2` and return the result.

(e) `friend Polynomial& operator-(const Polynomial& p1, const Polynomial& p2);`
- Subtract the first polynomial `p1` from the second polynomial `p2` and return the result.

(f) `friend Polynomial& operator*(const Polynomial& p1, const Polynomial& p2);`
- Multiply the two polynomials `p1` and `p2` and return the result.

**Input and output.** You need to write a main program which obtains an input line from users. The end of input is signalled by the Ctrl-Z character (EOF character in VC++) which your program should detect. Each input line is a sequence of integers that are separated by blanks and have one of the following three possible formats:

```
1 <polynomial> <polynomial>
2 <polynomial> <polynomial>
3 <polynomial> <polynomial>
```

where `<polynomial>` represents one polynomial. 1 means that the two polynomials are to be added; 2 means that the second polynomial should be subtracted from the first polynomial; and 3 means that the two polynomials are to be multiplied.

For each input line, your program should output to `cout` one separate line containing the result using the `<polynomial>` format.

*Note that you should make use of the overloaded input and output operator functions to get polynomials and output the result respectively. Similarity, you should use the overloaded arithmetic operators to calculate the results.*

## 3 Problem B: Infix to Postfix Conversion

**Task description.** Implement and test a program to convert standard arithmetic expressions (infix expressions) into postfix expressions.

An infix expression is the usual way such mathematical expressions are written. It is called infix because each operator appears in between its operands. In postfix expression, the operator is written after its operands. For example,

Infix form: `a+b-c+d*e-a*c`
Postfix form: `ab+c-de*+ac*-`

As you can see from the above example, the order of the operands is the same in infix and postfix. When we scan an expression for the first time, we can form the postfix by immediately passing any operands to the output. To handle the operators, we need a storage to store them until it is time to pass them to the output.

For this problem, you should use a stack ADT as the storage. You can either implement your own stack ADT or use the stack ADT in the standard C++ library. You should first read Weiss, section 3.3.3.

**Input and output.** You need to write a main program which obtains an input line from users. The end of input is signalled by the Ctrl-Z character (EOF character in VC++) which your program should detect. Each input line contains one string which is a legal infix expression. Assume that the input infix expression:

- is a legal infix expression that contains at most 100 characters.
- has only three operators: multiplication (*), addition (+) and subtraction (-) and possibly parentheses "(" and ")".
- has variables taken from the set {a, b, . . . , z}.
- contains no blank.

For each input line, your program should output to `cout` a separate line containing containing the corresponding postfix expression.