The Hong Kong University of Science and Technology
Department of Computer Science and Engineering
COMP2012: Object-Oriented Programming and Data Structures
(Spring 2014)

Midterm Examination

Date: Saturday, 22 March 2014

Venue: LT-F & LT-G for L1; LT-E for L2 **Time: 10:30 – 12:30 (2 hours)**

- This is a closed-book examination. However, you are allowed to bring with you a piece of A4-sized paper with notes written, drawn or typed on both sides for use in the examination.
- Your answers will be graded according to correctness, efficiency, precision, and clarity.
- During the examination, you must put aside your calculators, mobile phones, tablets and all other electronic devices. All mobile phones must be turned off.
- This booklet consists of single-sided pages. Please check that all pages are properly printed. You may use the reverse side of the pages for your rough work. If you decide to use the reverse side to continue your work, please clearly write down the question number.

Student Name	Solution
English nickname (if any)	
Student ID	
ITSC email	_____ @stu.ust.hk

I have not violated the Academic Honor Code in this examination (signature):_____

Question	**Score / Max. score**
Q1. Class member functions	/55
Q2. Constructors and Destructor	/17
Q3. Operator overloading	/20
Q4. Using the Polynomial class	/8
Total	**/100**

A Polynomial Class using Linked List *[100 points total]*

In this midterm examination, you are going to implement a polynomial class using linked list. Please note that memory leak is a bug, and deep copy should be used if data is to copied from one linked list to another.

In mathematics, a polynomial is an expression consisting of variables with degrees and coefficients. For example, the polynomial $3x^2 - 2x + 1$ is with variable *x*, where the coefficient for degree 2 is 3, and the coefficient for degree 1 is -2, and the constant term is 1.

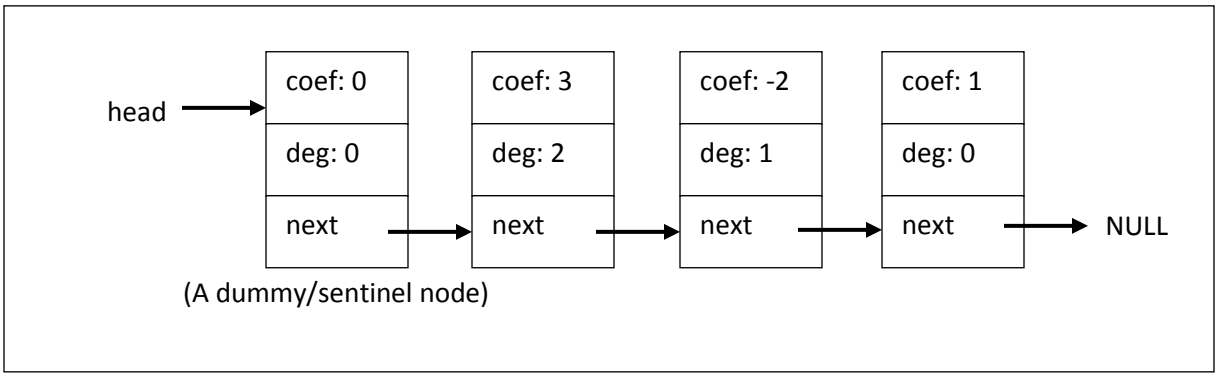
In this problem, please consider that:

- The variable takes on integer values;
- The degrees are unsigned (positive) integers;
- The coefficients should be non-zero integers except only the polynomial 0. Therefore, the polynomial $0x^2 + 1$ is not valid, which should be *reduced* to the valid polynomial 1. The polynomial $-4x^5 + 0$ is invalid and should be *reduced* to $-4x^5$. The polynomial 0 is a valid one.

Our polynomial is implemented using linked list where nodes are linked in decreasing order of degree.

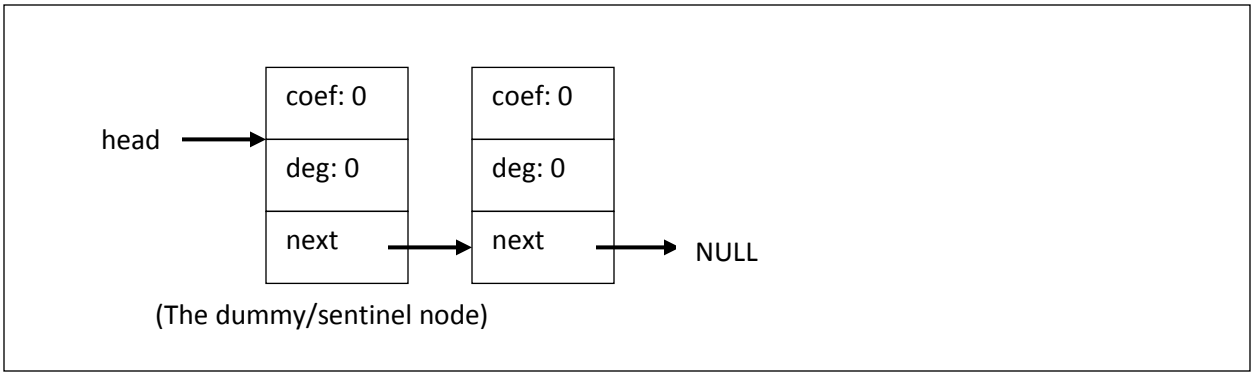
The head of the linked list points to the dummy (sentinel) node. The coefficient and degree of the dummy node should both be set to 0. This dummy node simplifies the operations of insertion and deletion. With the dummy node, there is no need to handle the boundary cases of insertion and deletion of the first node in the list as they are no longer a special case in linked list operations.

For example, a polynomial $3x^2 - 2x + 1$ should be represented by the following linked list:



In the above example, the first data node (coef:3, deg:2) has a predecessor dummy node (coef:0, deg:0) instead of being pointed by the *head* pointer. Therefore, the operations required for inserting before this node or deleting this node is the same as inserting before or deleting the other data node in the list.

Another example, polynomial 0, is represented below. Again, we see that the first node is the dummy sentinel node and the second node is the data node:



The header file `Polynomial.h` defines the class `Polynomial` as follows. You will implement the functions in the implementation file `Polynomial.cpp`:

```
struct Node {
      int coef;
      unsigned int deg;
      Node *next;
};

class Polynomial {
public:
      Polynomial();                                        // See Q2a
      ~Polynomial();                                       // See Q2b
      Polynomial(const Polynomial& other);                 // See Q2c

     void addTerm(int coef, unsigned int deg);            // See Q1d
      void read(istream& is);                              // See Q1e
      int evaluate(int x) const ;                          // See Q1f

      Polynomial& operator=(const Polynomial& other);     // See Q3a
      void operator+=(const Polynomial& other);           // See Q3b

     //… some other functions not relevant to this questions

private:
      void cleanUp(Node *ptr);                             // See Q1a
      void copyFrom(const Polynomial& other);              // See Q1b
     void reduce();                                       // See Q1c

     // pointing to the dummy sentinel node
      Node *head;
};

Polynomial operator+
(const Polynomial& first, const Polynomial& second);    // See Q3c
```

Q1 Class member functions *[55 marks]*

(a) ***[5 marks]*** Show the codes for the member function *cleanUp* in the implementation file. *cleanUp* is a private utility function which deletes the nodes of the linked list starting at the node pointed by *ptr*, including the node itself up to the end of the linked list. Note that *ptr* can be NULL, in which case no action would be taken.

```
void Polynomial::cleanUp(Node *ptr) {


      Node *tmp = NULL;
      while ( ptr != NULL ) {
            tmp = ptr;
            ptr = ptr->next;
            delete tmp;
      }


}
```

(b) ***[8 marks]*** Show the codes of the member function *copyFrom* in the implementation file. *copyFrom* is a private utility function which makes a deep copy, and hence replication, of the polynomial *other*. You may invoke other member functions if necessary.

```
void Polynomial::copyFrom(const Polynomial& other) {

     cleanUp( head -> next );
     head -> next = NULL;

      Node *curr = head;
      Node *otherCurr = other.head->next;

      while (otherCurr != NULL) {
            // Deep copy
            curr->next = new Node;
            curr = curr->next ;
            curr->coef = otherCurr->coef;
            curr->deg = otherCurr->deg;
            curr->next = NULL;

            otherCurr = otherCurr->next;
      }



}
```

(c) ***[12 marks]*** Implement the member function ***reduce***, which removes all the terms with coefficient 0 in the polynomial so as to make it a valid one. For example, the polynomial $3x^7+0x^2-7x+0$ should be reduced to x^7-7x. If all terms are removed, it should result in the polynomial 0 (i.e., solely with a zero constant term).

```
void Polynomial::reduce() {

      Node *prev = head;         // prev points to the dummy node
      Node *curr = head->next;   // curr points to the first data node


     while ( curr != NULL ) {

            if ( curr->coef == 0  ) {

                  // Delete the node with zero coefficient
                  Node *tmp = curr ;
                  prev->next = curr->next;
                  curr = curr->next;
                  delete tmp;

            } else {

                  // Advance 2 pointers
                  prev = curr ;
                  curr = curr->next;
            }

      }

      // Special case: all term having 0 coefficient
      if ( head-> next == NULL ) {
            // Action: Add back a 0 constant term

            Node *zero = new Node;
            zero->deg = 0;
            zero->coef = 0;
            zero->next = NULL;

            head->next = zero;

      }









}
```

(d) ***[15 marks]*** Implement the member function ***addTerm*** which adds a term with coefficient ***coef*** and degree ***deg*** to the polynomial. You can assume that ***coef*** is an integer and ***deg*** is a non-negative integer. You need to consider the following two cases:

- In the original polynomial, the term with degree ***deg*** does not exist:
  - Action: Create a new node and insert it to the appropriate position of the linked list so that the degrees in the linked list are sorted in decreasing order.
- The term with degree ***deg*** exists: In this case, update the coefficient by the sum.

Note that you need to *reduce* the polynomial if there are terms with coefficient 0. You may invoke other member functions if necessary.

```
void Polynomial::addTerm(int coef, unsigned int deg) {
      Node *prev = head;         // prev points to the dummy node
      Node *curr = head->next;   // curr points to the first data node

      If (!coefficient)return;

     while ( curr != NULL ) {
           if ( curr->deg == deg ) {
                 curr->coef += coef;
                 reduce(); // invoke reduce before exiting the function
                 return; // exit the function

           } else if ( curr->deg < deg ) {

                 // Create a new node
                 Node *newNode = new Node;
                 newNode->coef = coef;
                 newNode->deg = deg;

                 prev->next = newNode;
                 newNode->next = curr;

                 reduce(); // invoke reduce before exiting the function
                 return; // exit the function
           }

           prev = curr;
           curr = curr->next;
      }


// (Continuation of the addTerm member function)



      // If the while loop is passed without return
      // Action: Create a new node and append to the end
```

```
        Node *newNode = new Node;
        newNode->coef = coef;
        newNode->deg = deg;
       newNode->next = NULL;
        prev->next = newNode;

        reduce(); // invoke reduce before exiting the function
```

```
} // end of addTerm
```

(d) ***[8 marks]*** Implement the member function *read*, which reads in a polynomial from an input stream. You may invoke other member functions if necessary. You may assume the data in the input file is always in the correct format.

The input format and a sample input are given below:

The input format

The First line: The number of terms in the polynomial N

Remaining N lines: (Note: They are not sorted by degree)
coef1 deg1
coef2 deg2
…

A sample input (for the polynomial $3x^2 - 2x + 1$):

```
3
3   2
1   0
-2  1
```

A sample input for the polynomial 2:

```
1
2   0
```

```
void Polynomial::read(istream& is) {


      int n ;
      int coef;
      unsigned int deg;

     cleanUp( head -> next );
     head->next = NULL;

      is >> n;
      for (int i=0; i<n; i++) {
            is >> coef >> deg;  // Get coef and deg, line by line
            addTerm(coef, deg); // invoke addTerm
      }


}
```

(e) ***[7 marks]*** Implement the member function *evaluate*, which returns the value of the polynomial given the value of the variable. A few evaluated examples are given below:

Polynomial	**Value of x**	**Evaluated result**
$3x^2 - 2x + 1$	0	1
	2	9
	-2	17
0	0	0
	2	0
	-2	0

```
// Given in this question
int power(int x, int n) {
      int result = 1;
      for (int i=0; i<n; i++)
            result = result * x;
      return result;

}

int Polynomial::evaluate(int x) const {

      int result = 0 ;
      int coef, deg;
      Node *ptr = head->next;

      while ( ptr != NULL ) {

            coef = ptr->coef;
            deg = ptr->deg;

            result += coef * power(x, deg);

            ptr = ptr->next;
      }

      return result;




}
```

Q2 Constructors and Destructor *[17 marks]*

(a) ***[8 marks]*** Implement the *default constructor*, where the polynomial should be initialized to *the polynomial 0.*

```
Polynomial::Polynomial() {


      // Pointing to the dummy sentinel node
      head = new Node;
      head->coef = 0;
      head->deg = 0;
      head->next = new Node;

      // First data node: 0, 0
      head->next->coef = 0;
      head->next->deg = 0;
      head->next->next = NULL;




}
```

(b) ***[3 marks]*** Implement the destructor. You may invoke other member functions if necessary.

```
Polynomial::~Polynomial() {


  cleanUp(head); // clean up all the nodes, including the dummy node

}
```

(c) ***[6 marks]*** Implement the copy constructor. You may invoke other member functions if necessary.

```
Polynomial::Polynomial(const Polynomial& other) {

      // Pointing to the dummy sentinel node
      head = new Node;
      head->coef = 0;
      head->deg = 0;
      head->next = NULL;
      copyFrom(other);  // invoke copyFrom


}
```

Q3 Operator overloading *[20 marks]*

(a) ***[6 marks]*** Implement the assignment operator. The assignment operator must support a chain/concatenated assignment (i.e. a = b = c, where a, b and c are polynomials). You may invoke other member functions if necessary.

```
Polynomial& Polynomial::operator=(const Polynomial& other) {

   if(this != & other){
      cleanUp(head->next); // clean up existing nodes
     head -> next = NULL;
      copyFrom(other);     // copy from others
   }
   return *this;         // return its reference



}
```

(b) ***[5 marks]*** Implement the *operator+=* to add up 2 polynomials. For example:

$$\begin{array}{rl} & 3x^2 - 2x + 1 \\ + & 5x^6 + 4x^5 - 3x^2 - x - 1 \\ \hline = & 5x^6 + 4x^5 \quad\quad - 3x \end{array}$$

You may invoke other member functions if necessary.

```
void Polynomial::operator+=(const Polynomial& other) {


      Node *ptr = other.head->next;
      while ( ptr != NULL ) {
            addTerm(ptr->coef, ptr->deg);
            ptr = ptr->next;
      }









}
```

(c) ***[4 marks]*** Implement the global function operator+ to add up 2 polynomials: *first* and *second*. It should support the statement d = a + b + c, where a, b and c are polynomials.

```
Polynomial
operator+(const Polynomial& first, const Polynomial& second) {


      Polynomial tmp(first);
      tmp += second ;
      return tmp;









}
```

(d) ***[5 marks]*** Write down the function prototype in *Polynomial.h* and implement the operator>> in *Polynomial.cpp*, which takes in the same format as the member function *read* from an input stream *istream*, and hence supports statement such as *cin >> a >> b;* where a and b are polynomial objects.

```
// Write down the function prototype of operator >> here

istream & operator>> (istream &, Polynomial &);


// Implement the operator>> here



istream & operator>> (istream& is, Polynomial& s) {
      s.read(is);
     return is;
}
```

Q4 Using the Polynomial class *[8 marks]*

Using the polynomial class, write a program which creates a polynomial object $5x^4 + 4x^3 + 3x^2 + 2x + 1$ and cout the evaluated results of the polynomial for x=1 and x=-1. The expected output is as follows:

```
Result when x=1 : 15
Result when x=-1: 3
```

```
int main() {



    Polynomial result;     // A polynomial 0
    result.addTerm(5,4);   // order is not important for addTerm
    result.addTerm(4,3);
    result.addTerm(3,2);
    result.addTerm(2,1);
    result.addTerm(1,0);
    cout << "Result when x=1 : " << result.evaluate(1) << endl ;
    cout << "Result when x=-1: " << result.evaluate(-1) << endl ;

    return 0;
}
```

=== END OF PAPER ===