CSE213L: DSA Lab
LNMIIT, Rupa Ki Nangal
Training Set 05

In this training set, students will be guided to write a program to add two (polynomial-like) algebraic expressions of the form: `a[n]*x^n + … a[0]*x^0+ … a[-m]*x^-m` using linked list representations. Linked list representation for algebraic expressions is cost-effective if there are only a few nonzero coefficients with n+m being a large number. In the program(s), we will assume all coefficients a[i] to be integers. Further, symbols m and n denote non-negative integer values.

Sometimes it is easy to miss such algebraic expressions even as we use them in our daily routine activities. Two examples of such expressions are: Decimal number 102.345 and binary number 1001.01.
Why? See below:

102.345 = 1*10^2 + 2*10^0 + 3*10-1 + 4*10^-2 + 5*10^-3
1001.01 = 1*2^3 + 1*2^0 + 1*2^-2

Common decimal arithmetic uses an additional trick called *carry* to extend the process beyond what we will do in this training set. Smart students may try to add code to manage carry, if they so decide. This management process may be termed *normalisation* (or canonicalization). It would require all coefficients `c[i]` (`n > i >= -m`) to be restricted to value ranges `0 <= c < x` and be positive. Only coefficient that can have a negative value is coefficient for exponent n. Coefficient `c[n]` can have value range `-1 <= c[n] < x`. *Please feel free to skip this paragraph if you find it difficult or confusing.*

We have another goal for this training set. Previous training exercises have been built around a single instance of the data-structure. Here in this exercise, we need three linked lists. First two algebraic expressions represent the arguments to be added. And, the third data-structure is needed to store the result.

In this training set, we will use singly linked lists of terms (see code below for the declaration). To suit our purposes, we have declared two pieces of data in each term. Their meaning should be obvious to all bright students of LNMIIT. We will also store terms in a list in the decreasing order of power. The pointer variable referring to the start of the polynomial list will point at the term with the largest power in the list.

Training Set 05: Task 01

Students are provided a working program which adds two algebraic expressions and prints the result. Study the code carefully. After you have understood the code used to add two algebraic expressions, write code that subtracts `expr_2` from `expr_1` and prints the resulting expression.

Training Set 05: Task 02

There are some ADT functions listed in header file `expr.h` but not implemented in the provided code. Add codes for these ADT functions in file `expr.c`. Show your tutor an example where you copy algebraic expression `expr_1` as a new algebraic expression `expr_2` and then print results `expr_1 + expr_2` and `expr_1 - expr_2`.

Obviously, every coefficient in the latter expression is 0. You may wish to remove these terms from the algebraic expression. In fact, an empty algebraic expression may be printed as single value, 0.

Training Set 05: Task 03

In the final task you suggested in this training set is to modify the code written in `main.c` to add two expressions. The code provided by your instructor(s) can be simplified and made shorter by using ADT interface functions `makeExprCopy()` and `addTerm()` that meet the needs of the algebraic expressions.

Function `makeExprCopy()` returns a reference to a new copy of its argument. Function `addTerm()` supports the add operations as described in header file:

```
/* Insert a new term in expression if none have exponent == power.
      Otherwise add coeff to term's coefficient.
      Do not forget to delete term is term's new coefficient == 0 */
void addTerm(expression *prtToExpr, int coeff, int power);
```

Finish this and get your tutor to mark you. Bootstrap code is provided in this document to support students.

Output refers to Task 01

```
Expression_1 = +100*X^2000 +10*X^1000 -10*X^500 -10*X^-1000
Expression_2 = +200*X^3000 +10*X^1000 +110*X^500 +1000*X^0 +10*X^-1000
Expression_1 + Expression_2 = +200*X^3000 +100*X^2000 +20*X^1000 +100*X^500
+1000*X^0
```

expr.h

```
#include <assert.h>

struct term {
      struct term *nextP;
      int coeff;
      int power;
};

/* Expression is a list or terms */
typedef struct term * expression;

/* Gives pointer to the  first term in the expression list */
struct term *getFirstTerm (expression *prtToExpr);

/* Initialise a expression with no term */
void createExpr(expression *prtToExpr);

/* Insert a new term in expression list -- term with power should not exist
*/
void insertTerm(expression *prtToExpr, int coeff, int power);

/* Insert a new term in expression if none have exponent == power.
      Otherwise add coeff to term's coefficient.
      Do not forget to delete term is term's new coefficient == 0 */
void addTerm(expression *prtToExpr, int coeff, int power);

/* Gives next term in the expression after one referenced by termP */
struct term *getNextTerm(expression *prtToExpr, struct term *termP);

/* Prints the expression */
void printExpr(expression *prtToExpr);

/* Search for term with largest exponent value <= argument pow */
struct term *searchTerm(expression *prtToExpr, int pow);

/* Make a copy of src expression and make it accessible through destExpr */
void makeExprCopy(expression *srcExpr, expression *dstExpr);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "expr.h"

expression expr1;
expression expr2;
expression result;
struct term *term1P, *term2P;


int main(void) {
      /* Construct expression 1 */
      createExpr(&expr1); // Get the start pointer set
      insertTerm(&expr1, 10, 1000);
      insertTerm(&expr1, 100, 2000);
      insertTerm(&expr1, -10, 500);
      insertTerm(&expr1, -10, -1000);
      printf("Expression_1 = ");
      printExpr(&expr1);

      /* Construct expression 2 */
      createExpr(&expr2); // Get the start pointer set
      insertTerm(&expr2, 10, 1000);
      insertTerm(&expr2, 200, 3000);
      insertTerm(&expr2, 110, 500);
      insertTerm(&expr2, 1000, 0);
      insertTerm(&expr2, 10, -1000);
      printf("Expression_2 = ");
      printExpr(&expr2);

      /* Construct result = expr1 + expr2 */
      createExpr(&result);
      term1P = getFirstTerm(&expr1);
      term2P = getFirstTerm(&expr2);

      while (term1P!=NULL || term2P!=NULL) {
            if (term1P == NULL) {
                  insertTerm(&result, term2P->coeff, term2P->power);
                  term2P = getNextTerm(&expr2, term2P);
                  continue;
            }
            if (term2P == NULL) {
                  insertTerm(&result, term1P->coeff, term1P->power);
                  term1P = getNextTerm(&expr1, term1P);
                  continue;
            }
            if (term1P->power == term2P->power) {
                  if (term1P->coeff+term2P->coeff!=0)
                        insertTerm(&result,
                              term1P->coeff+term2P->coeff, term2P->power);
                  term2P = getNextTerm(&expr2, term2P);
                  term1P = getNextTerm(&expr1, term1P);
                  continue;
            }
            if (term1P->power > term2P->power) {
                  insertTerm(&result, term1P->coeff, term1P->power);
                  term1P = getNextTerm(&expr1, term1P);
                  continue;
            }
            if (term1P->power < term2P->power) {
                  insertTerm(&result, term2P->coeff, term2P->power);
                  term2P = getNextTerm(&expr2, term2P);
                  continue;
            }
      }
      printf("Expression_1 + Expression_2 = ");
      printExpr(&result);

      return 0;
}
```

expr.c

```
#include "expr.h"
#include <stdio.h>
#include <stdlib.h>

/* Gives pointer to the  first term in the expression list */
struct term *getFirstTerm (expression *ptrToExpr) {
      assert(ptrToExpr!=NULL);
      assert(*ptrToExpr!=NULL);
      return (*ptrToExpr)->nextP;
}

/* Initialise a expression with no term */
void createExpr(expression *ptrToExpr) {
      assert(ptrToExpr!=NULL);
      (*ptrToExpr) = malloc(sizeof(struct term));
      (*ptrToExpr)->nextP = NULL;
}
```

```
/* Insert a new term in expression list -- term with power should not exist
*/
void insertTerm(expression *ptrToExpr, int coeff, int power) {
      struct term *ptr, *prev;
      struct term *newTerm = malloc(sizeof(struct term));
      newTerm->coeff = coeff;
      newTerm->power = power;

      ptr = getFirstTerm(ptrToExpr);
      prev = *ptrToExpr;
      /* Notice that ptr is used only if not NULL */
      while (ptr != NULL && power < ptr->power) {
            // Find the right location for new node
            prev = ptr;
            ptr = getNextTerm(ptrToExpr, ptr);
      }
      assert(ptr == NULL || ptr->power != power);
      newTerm->nextP = ptr;
      prev->nextP = newTerm;
}

/* Gives next term in the expression after one referenced by termP */
struct term *getNextTerm(expression *ptrToExpr, struct term *termP) {
      assert(termP != NULL);
      return termP->nextP;
}

/* Available here but not through expr.h */
static void printTerm(struct term *term) {
      printf("%+d*X^%d ", term->coeff,term->power);
}

/* Prints the expression */
void printExpr(expression *ptrToExpr) {
      struct term *prnt = getFirstTerm(ptrToExpr);

      while (prnt!=NULL){
            printTerm(prnt);
            prnt = getNextTerm(ptrToExpr, prnt);
      }
      printf("\n");
}
```