CSE213L: DSA Lab
LNMIIT, Jaipur
Training Set 05

In this training set, students are coached to write a program to add (and subtract) two algebraic expressions of the form: **`a[n]*x^n +`** … **`a[0]*x^0+`** … **`a[-m]*x^-m`** using linked list data structures (Symbol `^` denotes *power of* or *exponent* operator). Linked-list based representation for the algebraic expressions is efficient if there are only a few nonzero-coefficient terms and n+m has a large value. In all programs in this training set, we will assume every coefficients `a[i]` to be an integer. Symbols m and n are non-negative integer values representing the minimum (negative) and maximum (positive) exponents in the expression.

I am sure some of you will be surprised to note that decimal number 50002.345 and binary number 1001.01 are two commonly used (but conveniently expressed) examples of the algebraic expression. Why?

```
50002.345 = 5*10^4 + 2*10^0 + 3*10-1 + 4*10^-2 + 5*10^-3
1001.01 = 1*2^3 + 1*2^0 + 1*2^-2
```

Please feel free to skip these indented paragraphs if you find them challenging.
Common decimal arithmetic uses an additional trick called *carry* to impose a normalisation (also called, canonicalising) requirement which we do not cover in this training set. A student may try to add code to manage carry if they so decide. We describe the idea as applicable to a twos-compliment like representation of the binary numbers.

All coefficients `c[i]` over the domain (`n > i >= -m`) are constrained to value range `0 <= c[i] < x`. Only coefficient that can have a negative value is for the term with the largest power. (Terms *power* and *exponent* are used interchangeably in this document). Assuming n to be the largest power in the expression, coefficient `c[n]` can have value range `-1 <= c[n] < x`.
End of optional section.

We have another programming goal for this training set. Previous training exercises have been designed around a single instance of the main data-structure. Here in this exercise, we need three linked lists for three algebraic expressions. Two algebraic expressions represent the arguments to be added (or to be operated by some other binary operation). And, the third data-structure is needed to take the result. Each expression will have its own private values of the coefficients and exponents; but all data structures will share the common ADT implementation code.

In this training set, we will use singly linked lists of terms (see Figure for declaration of `struct term`). To meet our programming needs, each term has two data members – one member represents the coefficient value of the term, and the other member represents the exponent (or power) of the term. We will consistently arrange terms in the lists in the

decreasing order of the power values. The program variable representing the expression will point at the first term in the list holding the largest exponent value for the algebraic expression.

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

Training Set 05: Task 01

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

Output (Slightly rearranged to fit page margins)

```
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
```

Training Set 05: Task 02

There are some ADT functions listed in header file `expr.h` but these interface functions are not implemented in the supplied code. Write codes for these ADT functions in file `expr.c`.

Demonstrate to your tutor a run of your program, where you copy algebraic expression `expr_1` to make a new algebraic expression `expr_2`. Compute and print expressions `expr_1 + expr_2` and `expr_1 - expr_2`.

Obviously, coefficients of all terms in the last expression is 0. You may wish to remove these terms from the algebraic expression. In fact, an empty algebraic expression should be printed as a single value 0.

Training Set 05: Task 03

The final task planned for this training set is to modify the provided code in function `main.c` to add two expressions. The code provided in the function can be improved and made shorter by using ADT interface functions `makeExprCopy()` and `addTerm()`.

Function `makeExprCopy()` returns a reference to a new copy of its parameter. Function `addTerm()` supports the add operations as described in the 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);
```

Use these two ADT functions to adjust the code in function `main.c` as indicated in the task.

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");
}
```

Vishu Malhotra
01 May 2020
The LNMIIT, Rupa ki Nangal
Jaipur Rajasthan 302031