1. D(x) 이외의 부분에서 과제 I 프로그램의 수정 사항

수정사항 없음

2. 구현한 곱하기 알고리즘에 대한 설명 (해당 부분 C 코드 캡처 이미지를 삽입하고 설명)

```c
#include <stdio.h>
#define _CRT_SECURE_NO_WARNINGS

typedef struct listNode* polyPointer; //listNode 구조체를 가리키는 포인터 선언
typedef struct listNode {
    int coef;
    int exp;
    polyPointer link; //listNode 구조체 자기 자신을 가르키는 포인터
} listNode;
```

listNode 구조체 자체를 가르키는 포인터를 선언하고, 이 포인터를 polyPointer 로 이름붙였습니다. listNode 안에는 항의 지수, 계수를 데이터로 저장할 수 있도록 했고, 앞에 선언했던 구조체를 참조하는 포인터를 저장할 수 있도록 해 다음 구조체를 참조할 수 있도록 했습니다.

```c
void attach(int coefficient, int exponent, polyPointer* ptr) {//노드를 삽입하는 함수
    polyPointer temp = malloc(sizeof(*temp)); //노드의 메모리를 할당
    temp->coef = coefficient; //메모리가 할당된 노드에 데이터 삽입
    temp->exp = exponent;
    (*ptr)->link = temp; //*ptr이 가르키는 노드 link에 현재 노드의 위치 삽입
    *ptr = temp; //*ptr이 현재 노드를 참조하도록 *ptr에 현재 노드 위치 삽입
}
```

항을 연결 리스트에 추가해주는 함수입니다. Coefficients, exponent 를 받아 데이터로 저장하고, 노드를 추가하려는 위치 바로 앞에 있는 노드의 주소값을 받습니다. 그래서 그 노드의 link 에 현재 노드의 위치를 저장해주고, 노드를 추가하려는 위치 앞에 있는 노드를 가리키는 *ptr 이 자신을 가리키도록 갱신합니다.

```
main() {

    for (int t = 0; t < 3; t++) {
        int A[10] = { 0, };
        int B[10] = { 0, };
        int a, b;


        printf("₩nA(x)를 입력해주세요.");
        for (int i = 0; i < 3; i++) {//A(x)를 계수 배열로 저장
            scanf_s("%dx%d", &a, &b);
            A[b] = a;
        }

        printf("B(x)를 입력해주세요.");
        for (int i = 0; i < 3; i++) {//B(x)를 계수 배열로 저장
            scanf_s("%dx%d", &a, &b);
            B[b] = a;
        }
```

전에 제출했던 과제와 동일한 부분입니다. A(x), B(x)를 입력받고 계수 배열로 저장합니다.

```
polyPointer D = NULL;
polyPointer rear = malloc(sizeof(*rear));
D = rear;
```

listNode 를 가리키는 포인터 D 를 선언해줍니다. 그리고 마지막 노드를 가리키는 rear 포인터도 선언해줍니다. 아직 노드가 없으므로 rear 와 D 가 같은 곳을 가리킵니다.

```
for (int k = 18; k > 9; k--) {//D(x) 안의 항의 지수가 9 초과일 경우
    int coef = 0;
    int exp = -1;
    for (int i = 9; i >= 0; i--) {
        if (k - i < 10 && A[i] != 0 && B[k - i] != 0) {
            coef += A[i] * B[k - i];
            int exp = k;
        }
    }
    if (coef!=0&&exp != -1) {
        attach(coef, exp, &rear);
    }
}
```

```
for (int k = 9; k >= 0; k--) {//D(x) 안의 항의 지수가 9 이하일 경우
    int coef = 0;
    int exp = -1;
    for (int i = k; i >= 0; i--) {
        if (A[i] != 0 && B[k - i] != 0) {
            coef += A[i] * B[k - i];
            exp = k;
        }
    }
    if (coef!=0&&exp != -1) {
        attach(coef, exp, &rear);
    }
}
rear->link = NULL;
D = D->link;
```

Coef 와 exp 를 계산해주는 알고리즘은 저번 과제 알고리즘을 재사용했습니다. 바뀐 부분은 먼저 coef 와 exp 를 0 과 -1 로 선언하고 만약 곱해진 해당 항이 없으면 exp 가 여전히 -1 이므로 attach 가 일어나지 않도록 했습니다. 모든 계산이 끝나고 항 저장이 끝난 후에, 가장 마지막 항인 rear 의 link 에 NULL 을 삽입해 연결 리스트가 끝났음을 알려줍니다. 그리고 첫 attach 가 일어났을 때 원래 앞의 노드였던 rear 는 비어있는 상태였으므로 현재 연결리스트의 제일 첫 노드에는 쓰레기값이 들어 있습니다. 다음 노드부터 D 가 시작되도록 D 의 시작을 두 번째 노드부터로 옮겨줍니다.

-A(x)의 1 개 항 axi 과 B(x)의 1 개 항 bxj 을 곱해서 항 abxi+j 를 산출했을 때, D(x)를 저장하는 linked list 에 지수=i+j 인 노드가

(1) 없고 list 가 empty 인 경우 (즉, 항 abxi+j 를 표현한 노드가 linked list 의 유일한 노드가 되는 경우) 두 번째 노드부터 attach 함수를 통해 값이 저장되고, rear 포인터가 입력된 노드를 참조합니다.

(2) 없고 list 가 empty 가 아닌 경우, attach 함수를 통해 다음 노드에 값이 저장되고, rear 포인터가 입력된 노드를 참조합니다.

(3) 이미 있고 항 abxi+j 를 반영하였을 때 계수가 0 이 아닌 경우, 같은 지수를 가진 경우, coef 변수에 계속 반복문을 통해 더해주고 한 번에 반복문을 빠져나오기 때문에 이미 있을 경우가 없습니다.

(4) 이미 있고 항 abxi+j 를 반영하였을 때 계수가 0 이 되는 경우, 같은 지수를 가진 경우, coef 변수에 계속 반복문을 통해 더해주고 한 번에 반복문을 빠져나오기 때문에 이미 있을 경우가 없습니다. 계수가 0 인 경우 attach 가 일어나지 않습니다.

이 방법을 사용할 경우, coef 와 exp 가 한 번에 계산되어서 나오기 때문에 그때그때 노드를 찾아서 데이터를 변경해주고 하지 않아도 되는 것이 장점입니다. 그러나 해당 지수를 가진 항이 있는지 반복문을 n^2 번 돌려야 하기 때문에 시간복잡도 부분에서 단점이 있는 것 같습니다.

```
    printf("\nA(x)=");
    for (int i = 0; i < 10; i++) {
        printf("%dx%d", A[i], i);
        if (A[i + 1] >= 0) {
            printf("+");
        }
    }
    printf("\nB(x)=");
    for (int i = 0; i < 10; i++) {
        printf("%dx%d", B[i], i);
        if (B[i + 1] >= 0) {
            printf("+");
        }
    }


    printf("\nD(x)=");
    for (int i=0;D;D=D->link,i++) {//D(x) 출력
        int coef = D->coef;
        int exp = D->exp;
        if (coef > 0 &&i!=0) {
            printf("+");
        }
        printf("%dx%d", D->coef, D->exp);
    }
}
```

저번 과제의 코드를 재사용한 출력 알고리즘입니다. 반복문을 사용해서 출력합니다.

3. 프로그래밍의 어떤 부분에 있어 그것의 구현 방식이 다양한 경우, (1) 어떤 부분이며, (2) 어떤 구현 방식을 선택했는지, 그리고 (3) 자신의 선택의 이유를 각 경우마다 설명: 과제 I 을 예로 들자면, “A(x)와 B(x)의 입력” 부분 등이 구현 방식이 다양한 경우에 해당됨

앞에서도 언급했던 것처럼 노드에 저장하기 전, 그 지수를 가진 모든 항을 찾은 다음 각 계수를 모두 더해서 노드에 저장할 것인지, 아니면 계산을 할 때마다 해당 지숫값을 가진 노드가 있는지 search 하고, 있다면 계수를 더해주고, 없다면 맞는 위치에 노드를 insert 해주는 방식을 사용할 것인지 두 가지 구현 방식이 있는 것 같습니다. 두 번째 방법은 search 를 하는 과정, 찾을 때마다 노드를 삽입하는 과정(노드의 링크를 끊고 삽입하고 하는 과정)에서 시간 복잡도 부분에서 손실이 있습니다. 첫 번째 방식은 모든 항을 찾는 과정에서 반복문을 돌리고, 그 과정에서 시간 복잡도 부분 손실이 있습니다. 저는 저번 과제에서 지수를 먼저 정하고 그 지수에 맞는 항을 찾아 모두 더하는 방식을 활용했었기 때문에 코드 재사용을 하기에는 첫 번째 방식이 효율적이라고 생각했고 첫 번째 방식을 활용했습니다.

4. 프로그램 실행결과 화면 캡쳐 및 설명

첫번째 실행결과는 과제 공지에 나온 예시이고, 두 번째 예제는 더해서 0 이 되는 항이 있는 경우의 예시입니다.(2(4)에 대한 예시) 세 번째 예제는 같은 지수를 가진 항이 나와 항의 개수가 9 개 미만이 되는 경우입니다. (2(3)에 대한 예시)