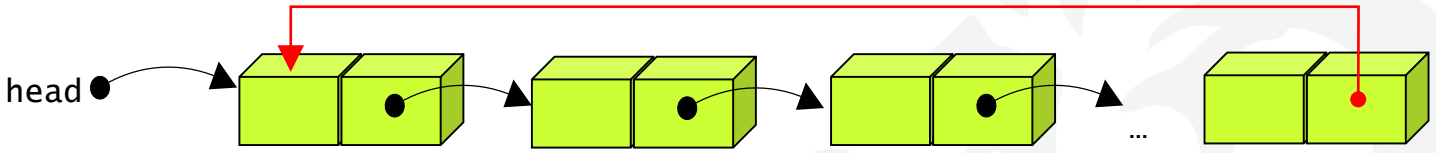

CSE2010 자료구조론

# Week 3: Linked List 2

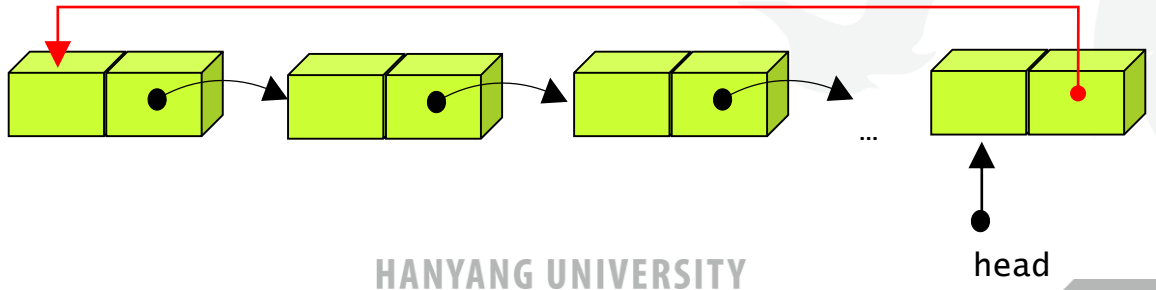
ICT융합학부 한진영

# 원형 연결 리스트

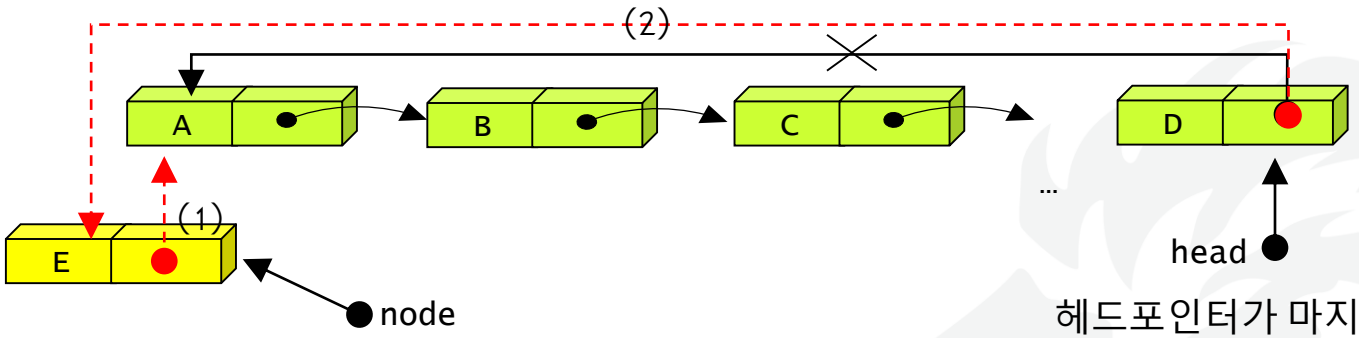
- 원형 연결 리스트
  - 마지막 노드의 링크가 첫 번째 노드를 가리키는 리스트
  - 마지막 노드의 링크 필드가 NULL이 아니고, 첫번째 노드를 가리킴




  - 한 노드에서 다른 모든 노드로의 접근이 가능
  - 삽입/삭제 연산이 단순연결리스트보다 용이함
    - cf) 단순연결리스트에서 삽입/삭제시 항상 선행 노드의 포인터가 필요함
  - 보통 헤드포인터가 마지막 노드를 가리키게끔 구성하면, 리스트 끝 또는 앞에 노드를 삽입하는 연산이 단순 연결 리스트에 비하여 용이함



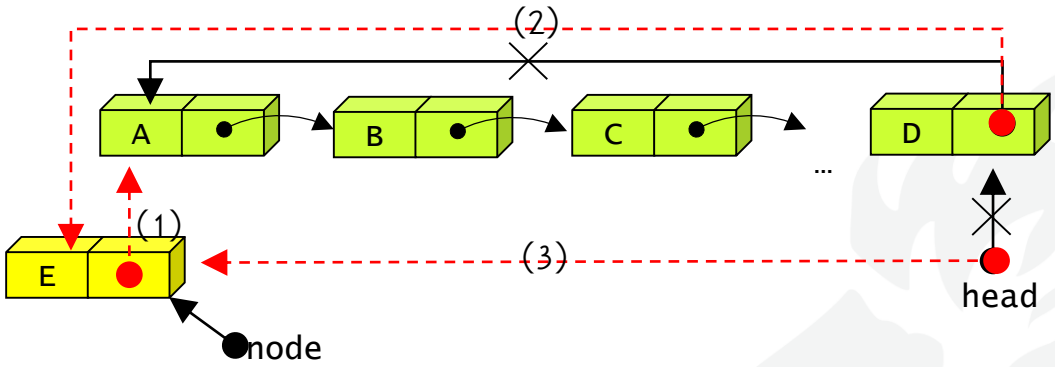
# 원형 연결 리스트: 리스트 처음에 삽입




헤드포인터가 마지막 노드를 가리킴

```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드
// node : 삽입될 노드
void insert_first(ListNode **phead, ListNode *node)
{
     if( *phead == NULL ){
          *phead = node;
          node->link = node;
     }
     else {
          node->link = (*phead)->link;
          (*phead)->link = node;
     }
}
```

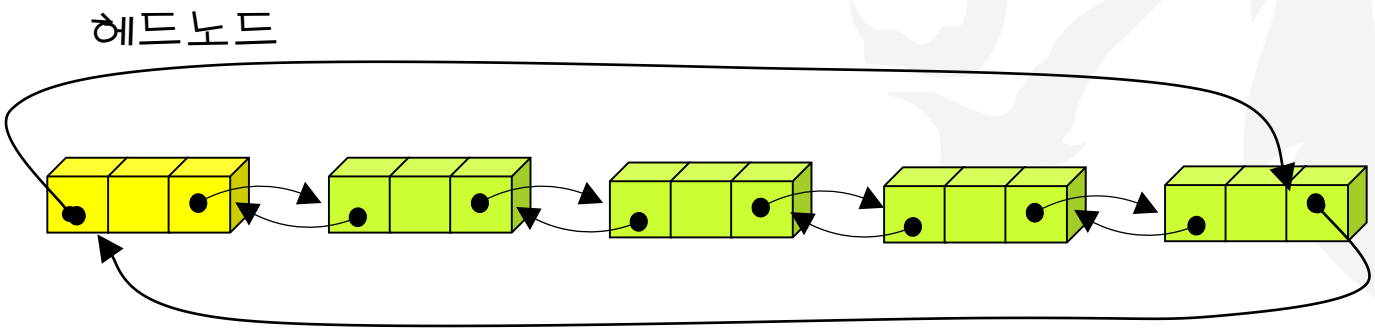
# 원형 연결 리스트: 리스트 끝에 삽입




```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드
// node : 삽입될 노드
void insert_last(ListNode **phead, ListNode *node)
{
     if( *phead == NULL ){
          *phead = node;
          node->link = node;
     }
     else {
          node->link = (*phead)->link;
          (*phead)->link = node;
          *phead = node;
     }
}
```

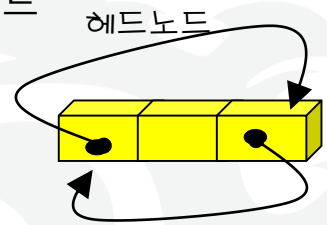
# 이중 연결 리스트

- 단순 연결 리스트의 문제점
  - 선행 노드를 찾기가 어려움
  - 삽입이나 삭제 시에는 반드시 선행 노드가 필요
- 이중 연결 리스트
  - 하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트
  - 링크가 양방향이므로 양방향으로 검색이 가능
  - 단점: 공간을 많이 차지하고 코드가 복잡
- 실제 사용되는 이중연결 리스트의 형태
  - **헤드노드+ 이중연결 리스트+ 원형연결 리스트**

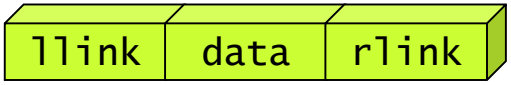


# 이중 연결 리스트: 헤드노드

- 헤드 노드(head node)
  - 데이터를 갖지 않고, 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드
  - 헤드 포인터와의 구별 필요
    - 헤드 포인터는 첫번째 노드를 가리키는 포인터
  - 공백상태에서는 헤드 노드만 존재




- 이중연결리스트에서의 노드의 구조
  - (왼쪽 링크 필드, 데이터 필드, 오른쪽 링크 필드)

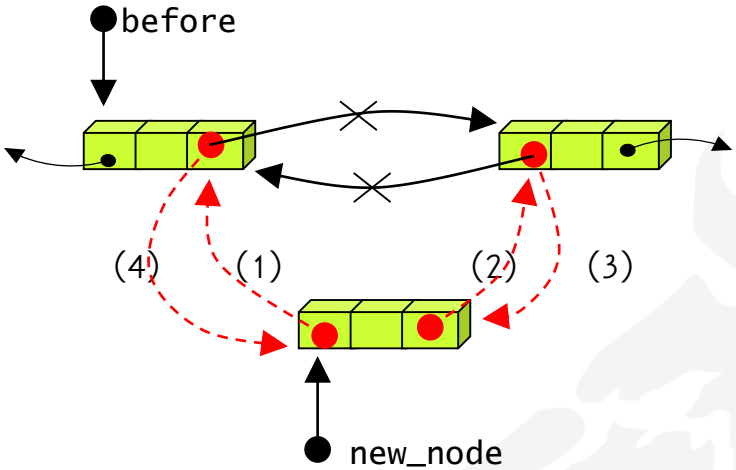



```
typedef int element;
typedef struct DlistNode {
    element data;
    struct DlistNode *llink;
    struct DlistNode *rlink;
} DlistNode;
```

  - 임의의 노드를 가리키는 포인터를 p라고 할 때,

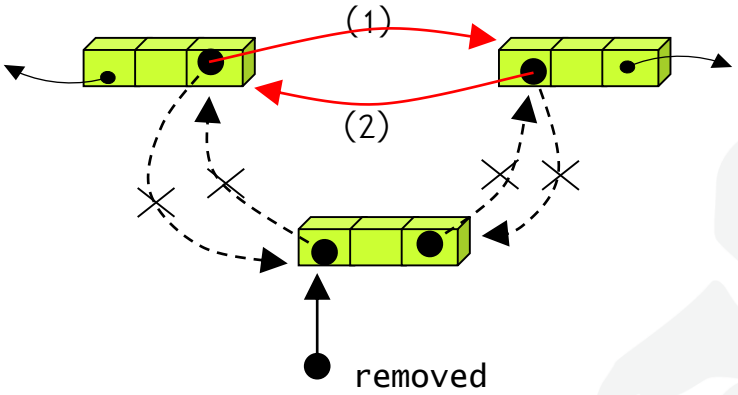
    항상, p == p -> llink -> rlink == p ->rlink->llink 관계 성립

# 이중 연결 리스트: 삽입 연산




```
// 노드 new_node를 노드 before의 오른쪽에 삽입한다.
void dinsert_node(DlistNode *before, DlistNode *new_node)
{
    new_node->llink = before;     (1)
    new_node->rlink = before->rlink;    (2)
    before->rlink->llink = new_node;    (3)
    before->rlink = new_node;           (4)
}
```

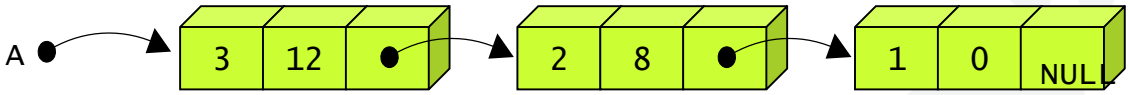
# 이중 연결 리스트: 삭제 연산




```
// 노드 removed를 삭제
void dremove_node(DlistNode *phead_node, DlistNode *removed)
{
    if( removed == phead_node ) return;
    removed->llink->rlink = removed->rlink; (1)
    removed->rlink->llink = removed->llink; (2)
    free(removed);
}
```

# 연결리스트의 응용: 다항식

- 하나의 다항식을 하나의 연결리스트로 표현
  - $A=3x^{12}+2x^{8}+1$




```
typedef struct ListNode {
    int coef;
    int expon;
    struct ListNode *link;
} ListNode;

ListNode *A, *B;
```

# 다항식의 덧셈(1)

- 2개의 다항식을 더하는 덧셈 연산을 구현
  - $A=3x^{12}+2x^{8}+1$, $B=8x^{12}-3x^{10}+10x^{6}$ 이면,

    $A+B=11x^{12}-3x^{10}+2x^{8}+10x^{6}+1$
- 다항식 A와 B의 항들을 따라 순회하면서 각 항들을 더함

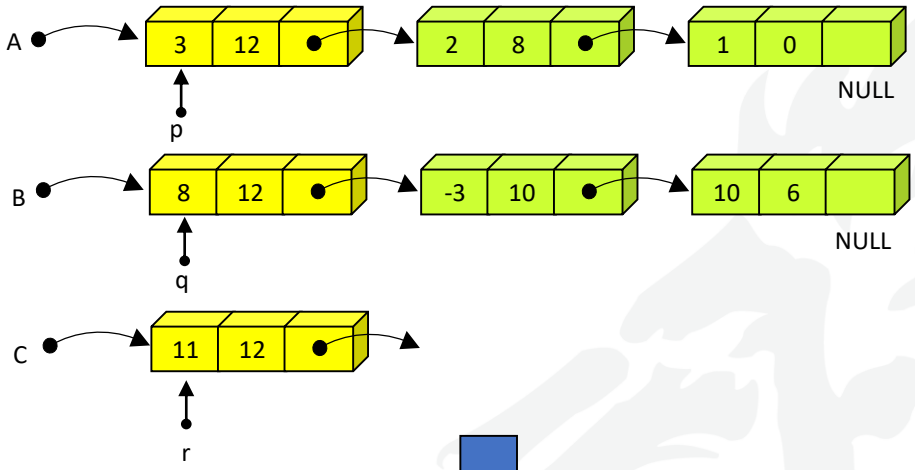
① p.expon == q.expon :
두 계수를 더해서 0이 아니면 새로운 항을 만들어 결과 다항식 C에 추가한다.
그리고 p와 q는 모두 다음 항으로 이동한다.

② p.expon < q.expon :
q가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C에 추가한다.
그리고 q만 다음 항으로 이동한다.

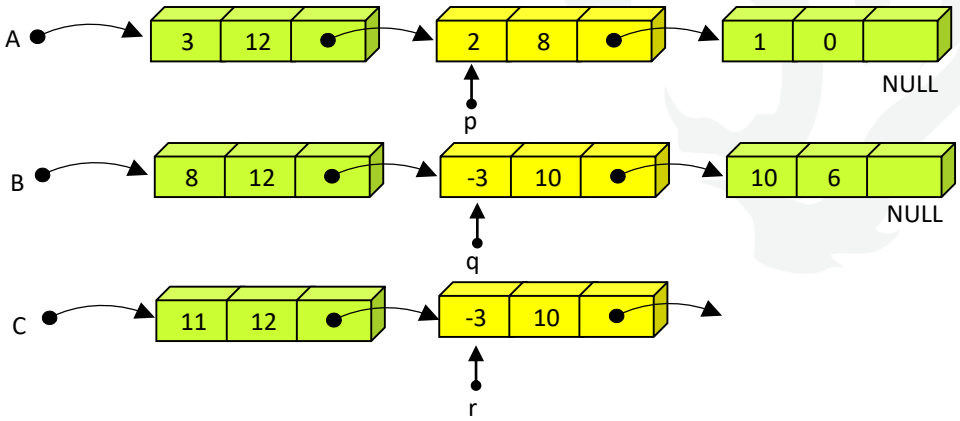
③ p.expon > q.expon :
p가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C에 추가한다.
그리고 p만 다음 항으로 이동한다.

# 다항식의 덧셈(2)

① p.expon == q.expon

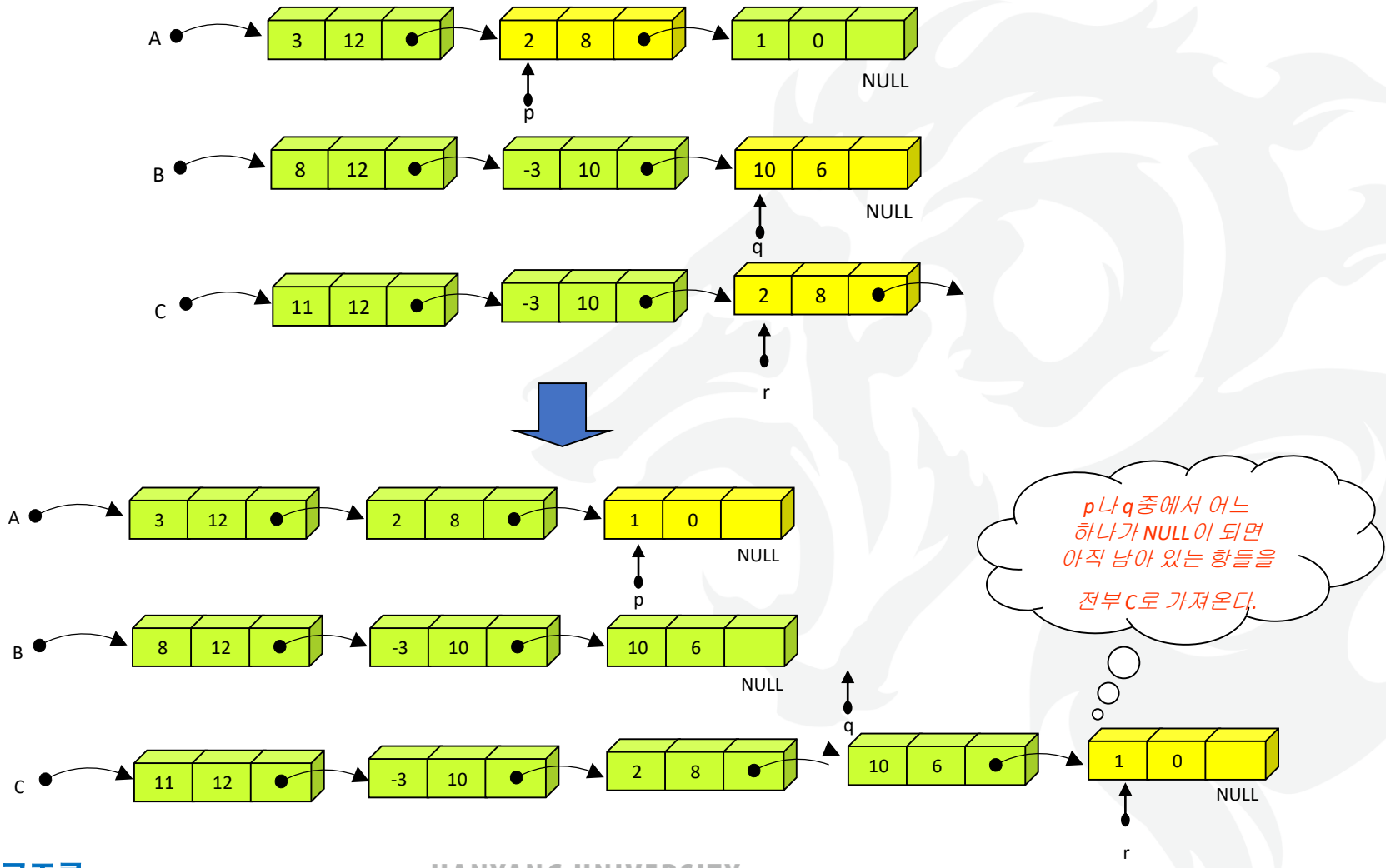



② p.expon < q.expon



# 다항식의 덧셈(3)

③ p.expon > q.expon



# 다항식 구현 코드(1)

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

// 연결 리스트의 노드의 구조
typedef struct ListNode {
	int coef;
	int expon;
	struct ListNode *link;
} ListNode;

// 연결 리스트 헤더 노드 구조
typedef struct ListHeader {
	int length;
	ListNode *head;
	ListNode *tail;
} ListHeader;
```

# 다항식 구현 코드(2)

```
// 초기화 함수
void init(ListHeader *plist)
{
        plist->length = 0;
        plist->head = plist->tail = NULL;
}
// plist는 연결 리스트의 헤더를 가리키는 포인터, coef는 계수, expon는 지수
// 새로운 노드를 만들어서 다항식의 마지막에 추가
void insert_node_last(ListHeader *plist, int coef, int expon)
{
        ListNode *temp = (ListNode *)malloc(sizeof(ListNode));
        if( temp == NULL ) error("메모리 할당 에러");
        temp->coef=coef;
        temp->expon=expon;
        temp->link=NULL;
        if( plist->tail == NULL ){
                plist->head = plist->tail = temp;
        }
        else {
                plist->tail->link = temp;
                plist->tail = temp;
        }
        plist->length++;
}
```

# 다항식 구현 코드(3)

```
// list3 = list1 + list2
void poly_add(ListHeader *plist1, ListHeader *plist2, ListHeader *plist3)
{
        ListNode *a = plist1->head;
        ListNode *b = plist2->head;
        int sum;
        while (a && b) {
                if (a->expon == b->expon) { // a의 차수 = b의 차수
                        sum = a->coef + b->coef;
                        if (sum != 0) insert_node_last(plist3, sum, a->expon);
                        a = a->link; b = b->link;
                }
                else if (a->expon > b->expon) { // a의 차수 > b의 차수
                        insert_node_last(plist3, a->coef, a->expon);
                        a = a->link;
                }
                else {          // a의 차수 < b의 차수
                        insert_node_last(plist3, b->coef, b->expon);
                        b = b->link;
                }
        }
```

# 다항식 구현 코드(4)

```
// a나 b중의 하나가 먼저 끝나게 되면 남아있는 항들을 모두 결과 다항식으로 복사
        for (; a != NULL; a = a->link)
                insert_node_last(plist3, a->coef, a->expon);
        for (; b != NULL; b = b->link)
                insert_node_last(plist3, b->coef, b->expon);
}
//
void poly_print(ListHeader *plist)
{
        ListNode *p = plist->head;
        for (; p; p = p->link) {
                printf("%d %d\n", p->coef, p->expon);
        }
}
```

# 다항식 구현 코드(5)

```
void main()
{
	ListHeader list1, list2, list3;

	// 연결 리스트의 초기화
	init(&list1);
	init(&list2);
	init(&list3);
	// 다항식 1을 생성
	insert_node_last(&list1, 3,12);
	insert_node_last(&list1, 2,8);
	insert_node_last(&list1, 1,0);
	// 다항식 2를 생성
	insert_node_last(&list2, 8,12);
	insert_node_last(&list2, -3,10);
	insert_node_last(&list2, 10,6);
	// 다항식 3 = 다항식 1 + 다항식 2
	poly_add(&list1, &list2, &list3);
	poly_print(&list3);
}
```

# Week 3: Linked List 2