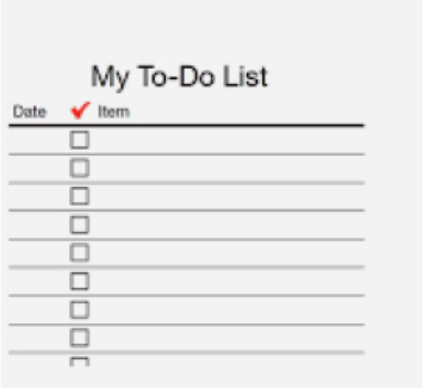
6장 연결 리스트 I

일상생활에서의 리스트

- 오늘 해야 할 일: (청소, 쇼핑, 영화관람)
- 버킷 리스트: (세계여행하기, 새로운 언어 배우기, 마라톤 뛰기)
- 요일들: (일요일, 월요일, ... ,토요일)
- 카드 한 벌의 값: (Ace, 2, 3,..., King)



리스트의 기본 연산

$$L = (item_0, item_1, item_2, . . ., item_{n-1})$$

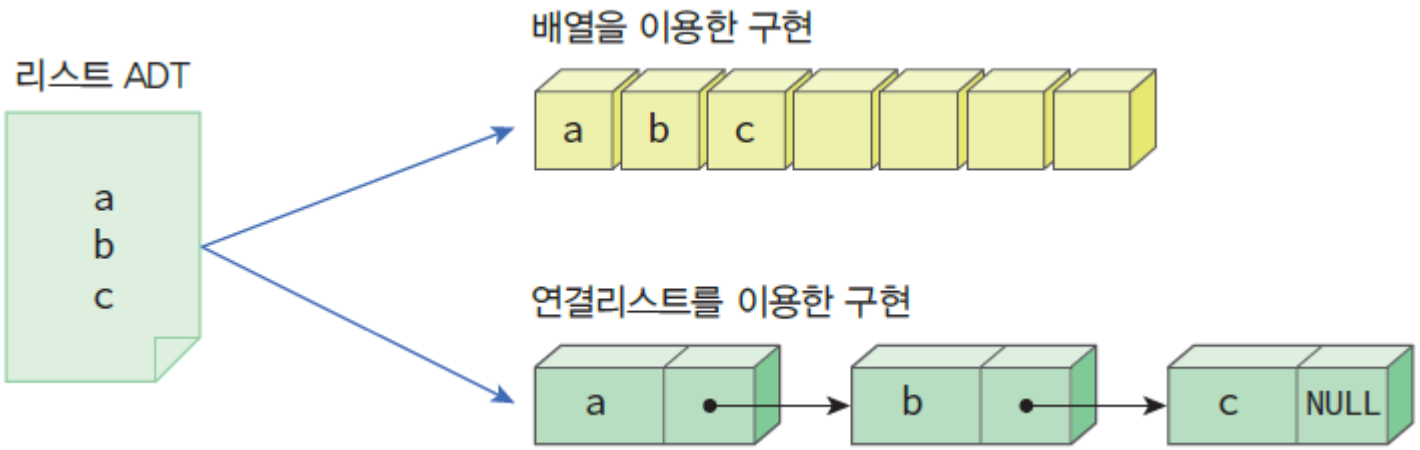
- 리스트에 새로운 항목을 추가한다(삽입 연산).
- 리스트에서 항목을 삭제한다(삭제 연산).
- 리스트에서 특정한 항목을 찾는다(탐색 연산).

리스트 ADT

· 객체: n개의 element형으로 구성된 순서 있는 모임
· 연산:

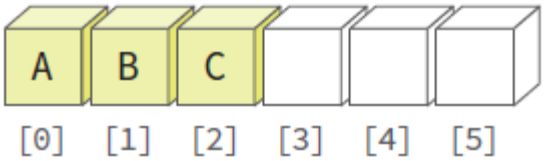
insert(list, pos, item) ::= pos 위치에 요소를 추가한다.
insert_last(list, item) ::= 맨 끝에 요소를 추가한다.
insert_first(list, item) ::= 맨 처음에 요소를 추가한다.
delete(list, pos) ::= pos 위치의 요소를 제거한다.
clear(list) ::= 리스트의 모든 요소를 제거한다.
get_entry(list, pos) ::= pos 위치의 요소를 반환한다.
get_length(list) ::= 리스트의 길이를 구한다.
is_empty(list) ::= 리스트가 비었는지를 검사한다.
is_full(list) ::= 리스트가 꽉 찼는지를 검사한다.
print_list(list) ::= 리스트의 모든 요소를 표시한다.

리스트 구현 방법



배열로 구현된 리스트

- 배열을 이용하여 리스트를 구현하면 순차적인 메모리 공간이 할당되므로, 이것을 리스트의 순차적 표현(sequential representation)이라고 한다.



ArrayListType의 구현

```
#define MAX_LIST_SIZE 100 // 리스트의 최대크기

typedef int element; // 항목의 정의

typedef struct {
        element array[MAX_LIST_SIZE]; // 배열 정의
        int size; // 현재 리스트에 저장된 항목들의 개수
} ArrayListType;
```

ArrayListType의 구현

```
// 오류 처리 함수
void error(char *message)
{
        fprintf(stderr, "%s\n", message);
        exit(1);
}
// 리스트 초기화 함수
void init(ArrayListType *L)
{
        L->size = 0;
}
// 리스트가 비어 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_empty(ArrayListType *L)
{
        return L->size == 0;
}
// 리스트가 가득 차 있으면 1을 반환
// 그렇지 않으면 1을 반환
int is_full(ArrayListType *L)
{
        return L->size == MAX_LIST_SIZE;
}
```

ArrayListType의 구현

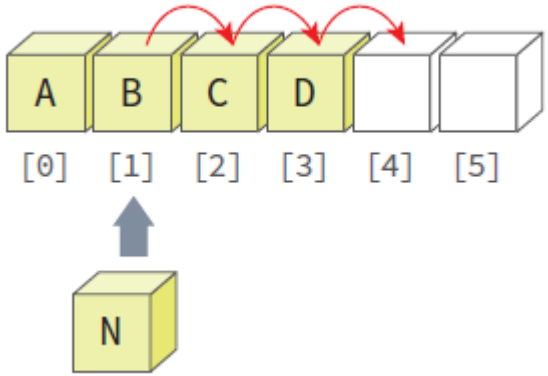
```
element get_entry(ArrayListType *L, int pos)
{
        if (pos < 0 || pos >= L->size)
                error("위치 오류");
        return L->array[pos];
}
// 리스트 출력
void print_list(ArrayListType *L)
{
        int i;
        for (i = 0; i<L->size; i++)
                printf("%d->", L->array[i]);
        printf("\n");
}
```

ArrayListType의 구현

```
void insert_last(ArrayListType *L, element item)
{
        if( L->size >= MAX_LIST_SIZE ) {
                error("리스트 오버플로우");
        }
        L->array[L->size++] = item;
}
```

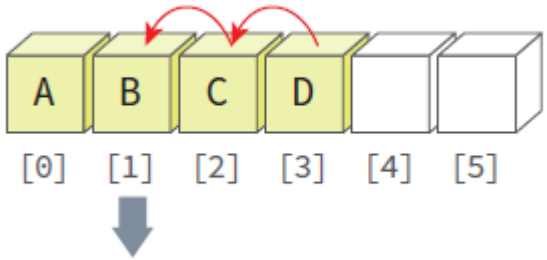
ArrayListType의 구현

```
void insert(ArrayListType *L, int pos, element item)
{
        if (!is_full(L) && (pos >= 0) && (pos <= L->size)) {
                for (int i = (L->size - 1); i >= pos; i--)
                        L->array[i + 1] = L->array[i];
                L->array[pos] = item;
                L->size++;
        }
}
```



ArrayListType의 구현

```
element delete(ArrayListType *L, int pos)
{
        element item;

        if (pos < 0 || pos >= L->size)
                error("위치 오류");
        item = L->array[pos];
        for (int i = pos; i<(L->size - 1); i++)
                L->array[i] = L->array[i + 1];
        L->size--;
        return item;
}
```



ArrayListType의 구현

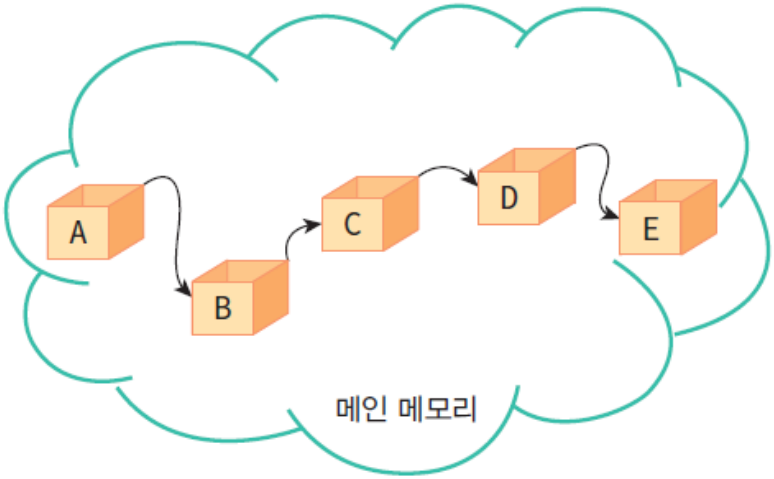
```
int main(void)
{
        // ArrayListType를 정적으로 생성하고 ArrayListType를
        // 가리키는 포인터를 함수의 매개변수로 전달한다.
        ArrayListType list;

        init(&list);
        insert(&list, 0, 10);     print_list(&list);        // 0번째 위치에 10 추가
        insert(&list, 0, 20);     print_list(&list);        // 0번째 위치에 20 추가
        insert(&list, 0, 30);     print_list(&list);        // 0번째 위치에 30 추가
        insert_last(&list, 40);   print_list(&list);        // 맨 끝에 40 추가
        delete(&list, 0);                    print_list(&list);        // 0번째 항목 삭제
        return 0;
}
```

실행결과

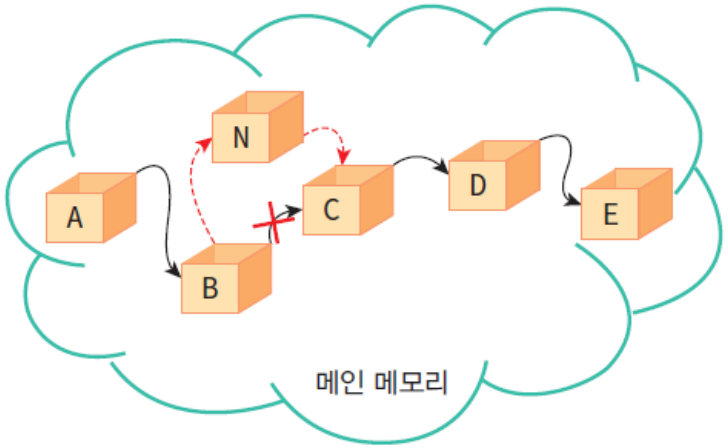
```
10->
20->10->
30->20->10->
30->20->10->40->
20->10->40->
```

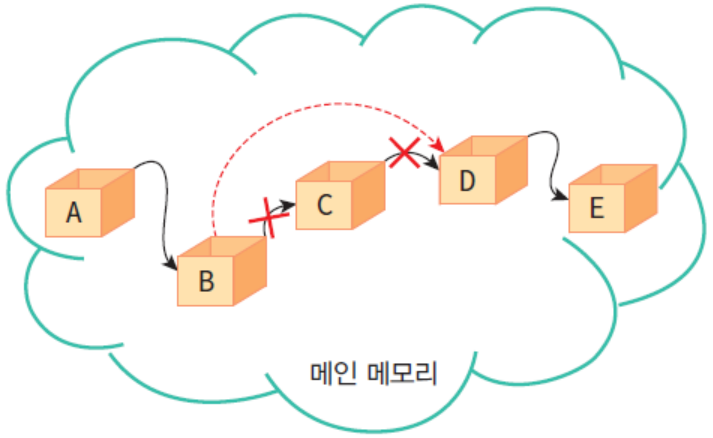
연결된 표현

- 리스트의 항목들을 노드(node)라고 하는 곳에 분산하여 저장
- 노드는 데이타 필드와 링크 필드로 구성
  - 데이타 필드 – 리스트의 원소, 즉 데이타 값을 저장하는 곳
  - 링크 필드 – 다른 노드의 주소값을 저장하는 장소 (포인터)



삽입과 삭제





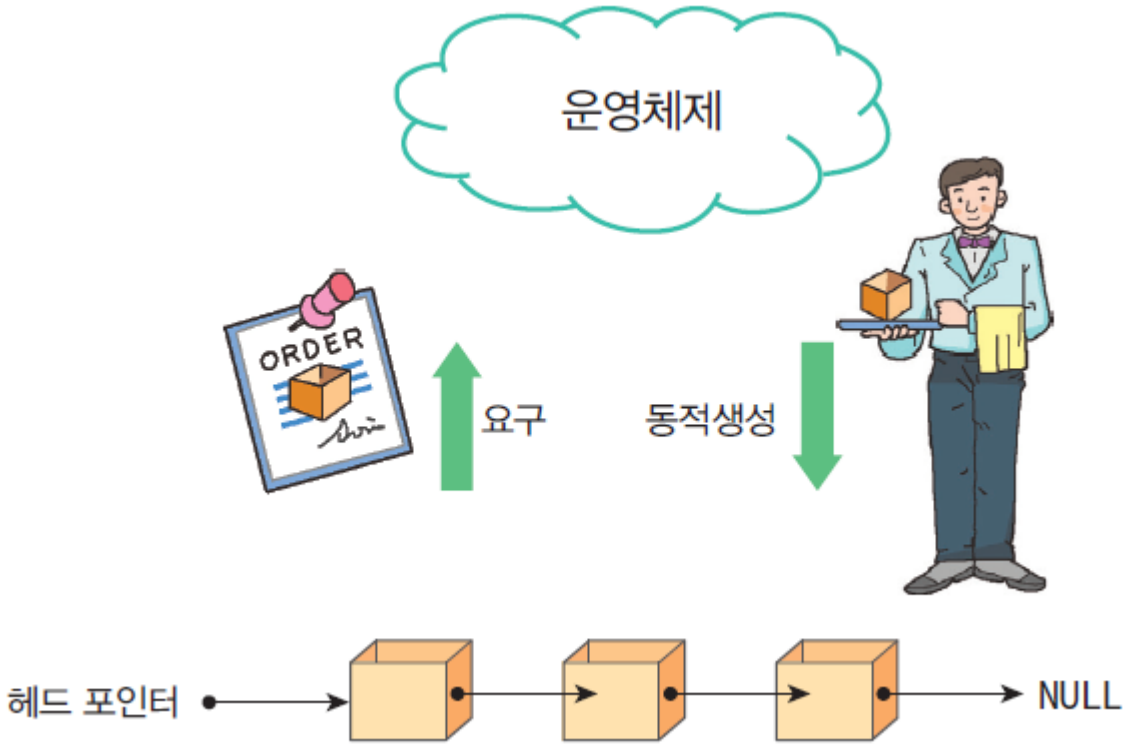
연결된 표현의 장단점

- 장점
  - 삽입, 삭제가 보다 용이하다.
  - 연속된 메모리 공간이 필요 없다.
  - 크기 제한이 없다
- 단점
  - 구현이 어렵다.
  - 오류가 발생하기 쉽다.

노드의 구조

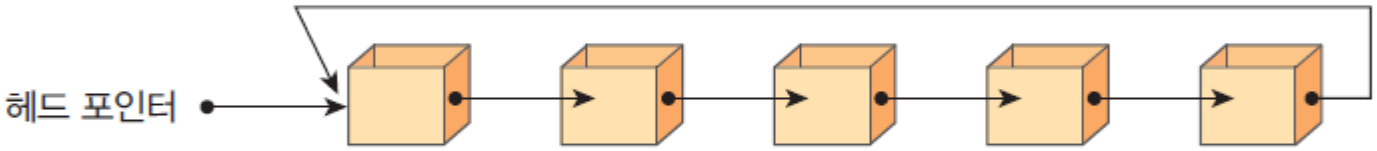
- 노드 = 데이터 필드 + 링크 필드

헤드 포인터와 노드의 생성

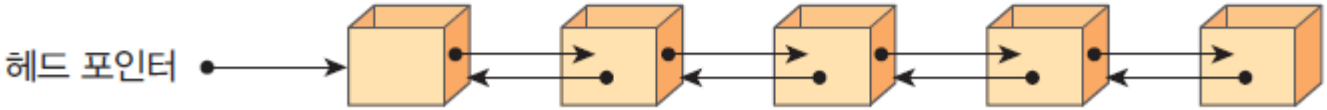


연결 리스트의 종류

단순 연결 리스트



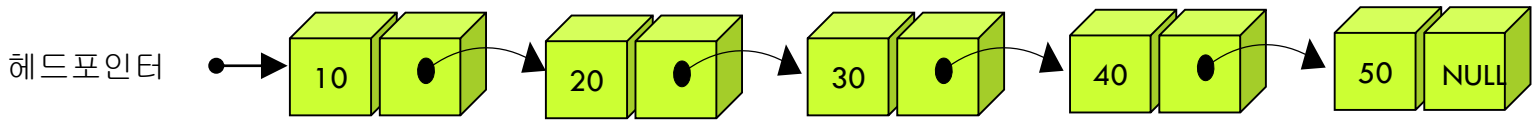
원형 연결 리스트



이중 연결 리스트

단순 연결 리스트

- 하나의 링크 필드를 이용하여 연결
- 마지막 노드의 링크 값은 NULL

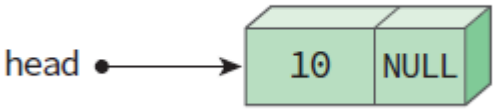


노드의 정의

```
typedef int element;

typedef struct ListNode {       // 노드 타입을 구조체로 정의한다.
            element data;
            struct ListNode *link;
} ListNode;
```

리스트의 생성

```
ListNode *head = NULL;

head = (ListNode *)malloc(sizeof(ListNode));

head->data = 10;
head->link = NULL;
```

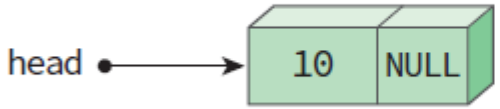


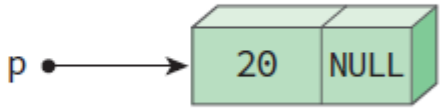
2번째 노드 생성

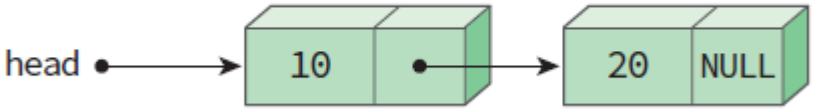
```
ListNode *p;
p = (ListNode *)malloc(sizeof(ListNode));
p->data = 20;
p->link = NULL;
```

노드의 연결

```
head->link = p;
```





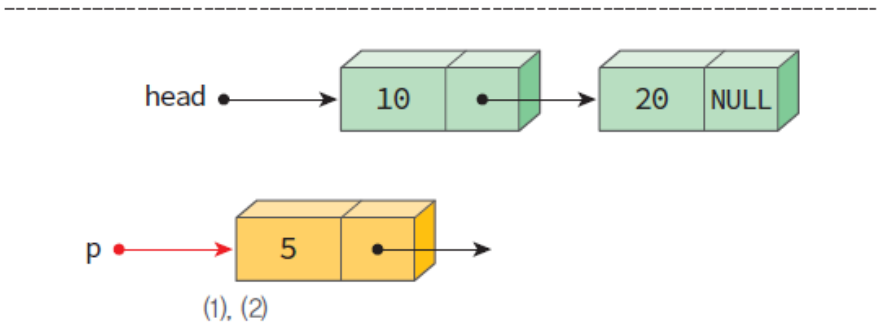


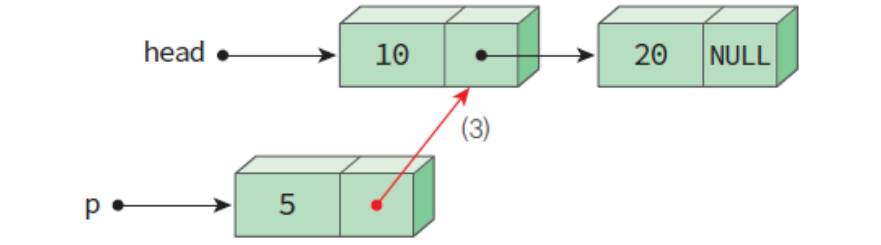
단순 연결 리스트의 연산

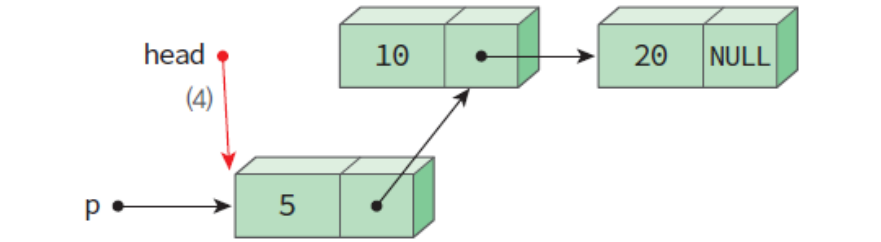
- insert_first(): 리스트의 시작 부분에 항목을 삽입하는 함수
- insert(): 리스트의 중간 부분에 항목을 삽입하는 함수
- delete_first(): 리스트의 첫 번째 항목을 삭제하는 함수
- delete(): 리스트의 중간 항목을 삭제하는 함수(도전 문제)
- print_list(): 리스트를 방문하여 모든 항목을 출력하는 함수

단순 연결 리스트(삽입연산)

```
ListNode* insert_first(ListNode *head, int value)
{
        ListNode *p =
         (ListNode *)malloc(sizeof(ListNode));//(1)
        p->data = value;
                  // (2)
        p->link = head;                          //(3)
        head = p;   //(4)
        return head;
}
```

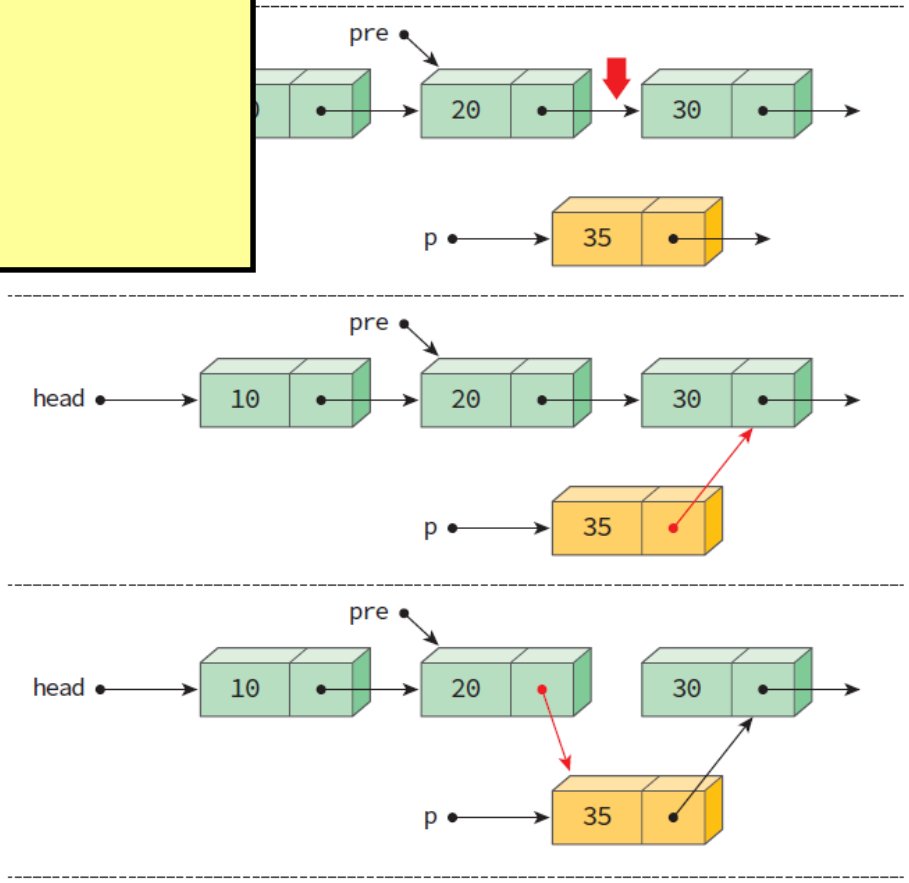






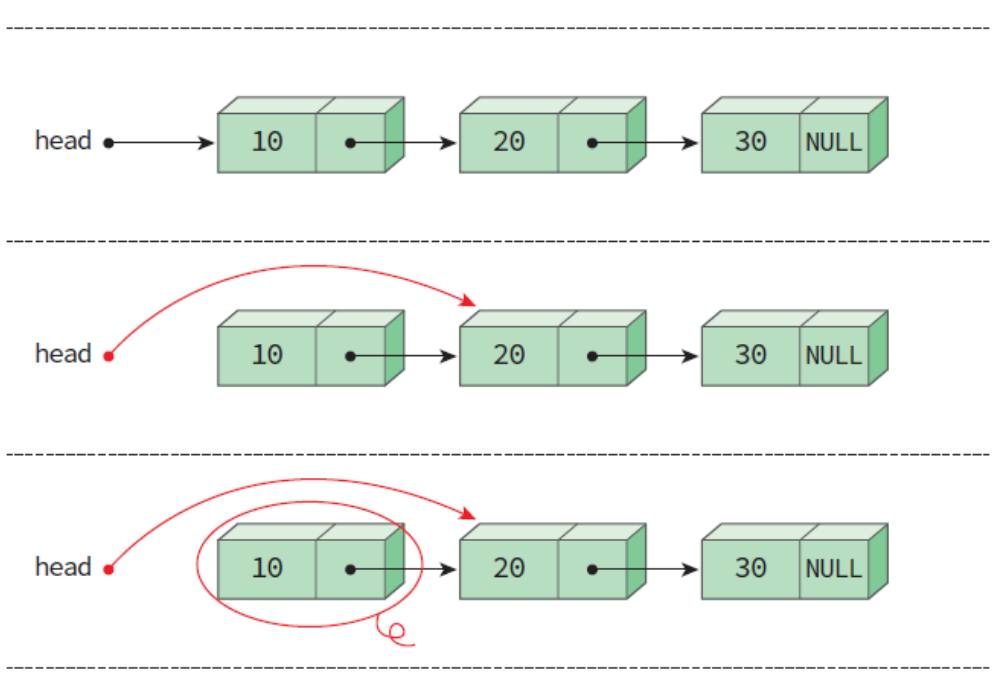
단순 연결 리스트(삽입연산)

```
// 노드 pre 뒤에 새로운 노드 삽입
ListNode*  insert(ListNode *head, ListNode *pre, element value)
{
        ListNode  *p  =  (ListNode  *)malloc(sizeof(ListNode));
        //(1)
        p->data = value;                    //(2)
        p->link = pre->link;      //(3)
        pre->link = p;                      //(4)
        return head;                        //(5)
}
```



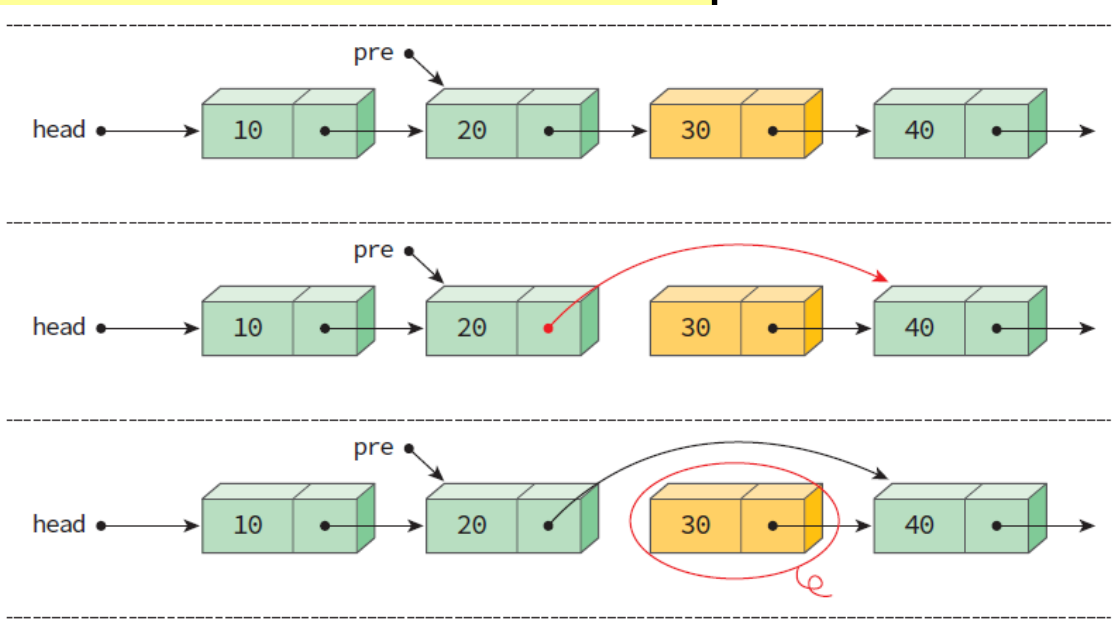
단순 연결 리스트(삭제연산)

```
ListNode* delete_first(ListNode *head)
{
        ListNode *removed;
        if (head == NULL) return NULL;
        removed = head;     // (1)
        head = removed->link;          // (2)
        free(removed);
        return head;
}
```



단순 연결 리스트(삭제연산)

```
// pre가 가리키는 노드의 다음 노드를 삭제한다.
ListNode* delete(ListNode *head, ListNode *pre)
{
        ListNode *removed;
        removed = pre->link;
        pre->link = removed->link;          // (2)
        free(removed);                      // (3)
        return head;                        // (4)
}
```



방문 연산 코드

```
void print_list(ListNode *head)
{
        for (ListNode *p = head; p != NULL; p = p->link)
                printf("%d->", p->data);
        printf("NULL \n");
}
```

테스트 프로그램

```
// 테스트 프로그램
int main(void)
{
        ListNode *head = NULL;

        for (int i = 0; i < 5; i++) {
                head = insert_first(head, i);
                print_list(head);
        }
        for (int i = 0; i < 5; i++) {
                head = delete_first(head);
                print_list(head);
        }
        return 0;
}
```

실행결과

```
0->NULL
1->0->NULL
2->1->0->NULL
3->2->1->0->NULL
4->3->2->1->0->NULL
3->2->1->0->NULL
2->1->0->NULL
1->0->NULL
0->NULL
NULL
```

Lab: 단어들을 저장하고 있는 연결리스트

APPLE->NULL
KIWI->APPLE->NULL
BANANA->KIWI->APPLE->NULL

Solution

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

typedef struct {
        char name[100];
} element;

typedef struct ListNode {       // 노드 타입
        element data;
        struct ListNode *link;
} ListNode;

// 오류 처리 함수
void error(char *message)
{
        fprintf(stderr, "%s\n", message);
        exit(1);
}
```

Solution

```
ListNode* insert_first(ListNode *head, element value)
{
        ListNode *p = (ListNode *)malloc(sizeof(ListNode));          //(1)
        p->data = value;                                             // (2)
        p->link = head;      // 헤드 포인터의 값을 복사    //(3)
        head = p; // 헤드 포인터 변경              //(4)
        return head;
}

void print_list(ListNode *head)
{
        for (ListNode *p = head; p != NULL; p = p->link)
                printf("%s->", p->data.name);
        printf("NULL \n");
}
```

Solution

```
// 테스트 프로그램
int main(void)
{
        ListNode *head = NULL;
        element data;

        strcpy(data.name, "APPLE");
        head = insert_first(head, data);
        print_list(head);

        strcpy(data.name, "KIWI");
        head = insert_first(head, data);
        print_list(head);

        strcpy(data.name, "BANANA");
        head = insert_first(head, data);
        print_list(head);
        return 0;
}
```

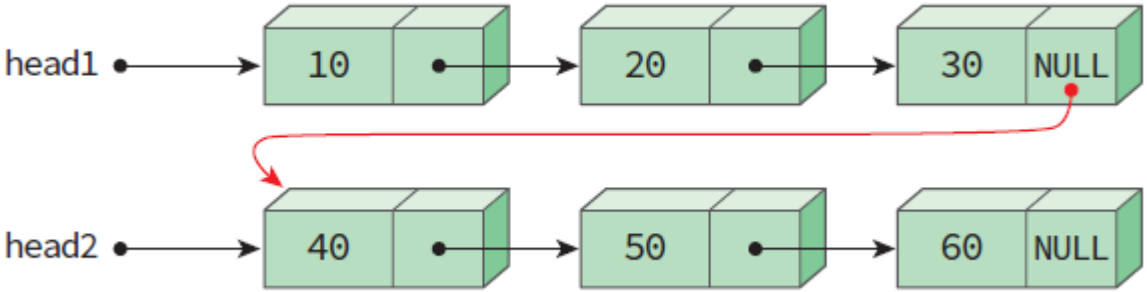
Lab: 특정한 값을 탐색하는 함수

```
10->NULL
20->10->NULL
30->20->10->NULL
리스트에서 30을 찾았습니다.
```

```
ListNode* search_list(ListNode *head, element x)
{
        ListNode *p = head;

        while (p != NULL) {
                if (p->data == x) return p;
                p = p->link;
        }
        return NULL;            // 탐색 실패
}

// 테스트 프로그램
int main(void)
{
        ListNode *head = NULL;

        head = insert_first(head, 10);
        print_list(head);
        head = insert_first(head, 20);
        print_list(head);
        head = insert_first(head, 30);
        print_list(head);
        if (search_list(head, 30) != NULL)
                printf("리스트에서 30을 찾았습니다. \n");
        else
                printf("리스트에서 30을 찾지 못했습니다. \n");
        return 0;
}
```

Lab: 2개의 리스트를 합하는 함수

```
30->20->10->NULL
50->40->NULL
30->20->10->50->40->NULL
```

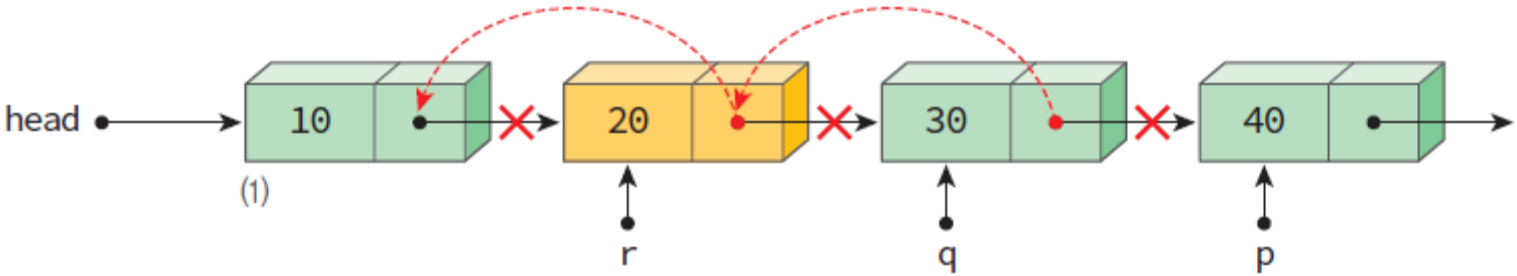


Solution

```
ListNode* concat_list(ListNode *head1, ListNode *head2)
{
        if (head1 == NULL) return head2;
        else if (head2 == NULL) return head2;
        else {
                ListNode *p;
                p = head1;
                while (p->link != NULL)
                        p = p->link;
                p->link = head2;
                return head1;
        }
}
```

Lab: 리스트를 역순으로 만드는 연산

30->20->10->NULL
10->20->30->NULL

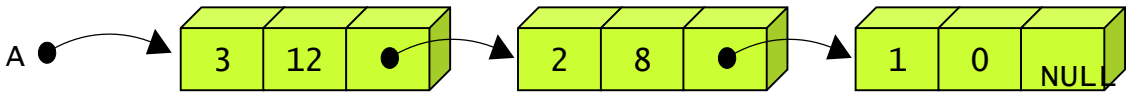


Solution

```
ListNode* reverse(ListNode *head)
{
        // 순회 포인터로 p, q, r을 사용
        ListNode *p, *q, *r;

        p = head;       // p는 역순으로 만들 리스트
        q = NULL;       // q는 역순으로 만들 노드
        while (p != NULL) {
                r = q;          // r은 역순으로 된 리스트.
                                // r은 q, q는 p를 차례로 따라간다.
                q = p;
                p = p->link;
                q->link = r;    // q의 링크 방향을 바꾼다.
        }
        return q;
}
```

연결리스트의 응용: 다항식

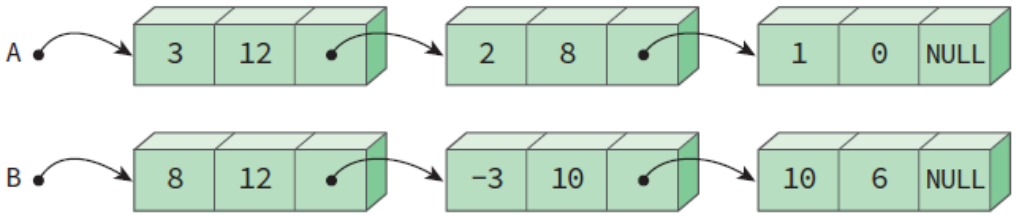
- 다항식을 컴퓨터로 처리하기 위한 자료구조
  - 다항식의 덧셈, 뺄셈...
- 하나의 다항식을 하나의 연결리스트로 표현
  - $A=3x^{12}+2x^{8}+1$



연결리스트의 응용: 다항식

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

예를 들면 다항식 $A(x)=3x^{12}+2x^{8}+1$과 $B(x)=8x^{12}-3x^{10}+10x^{6}$은 다음과 같이 표현된다.

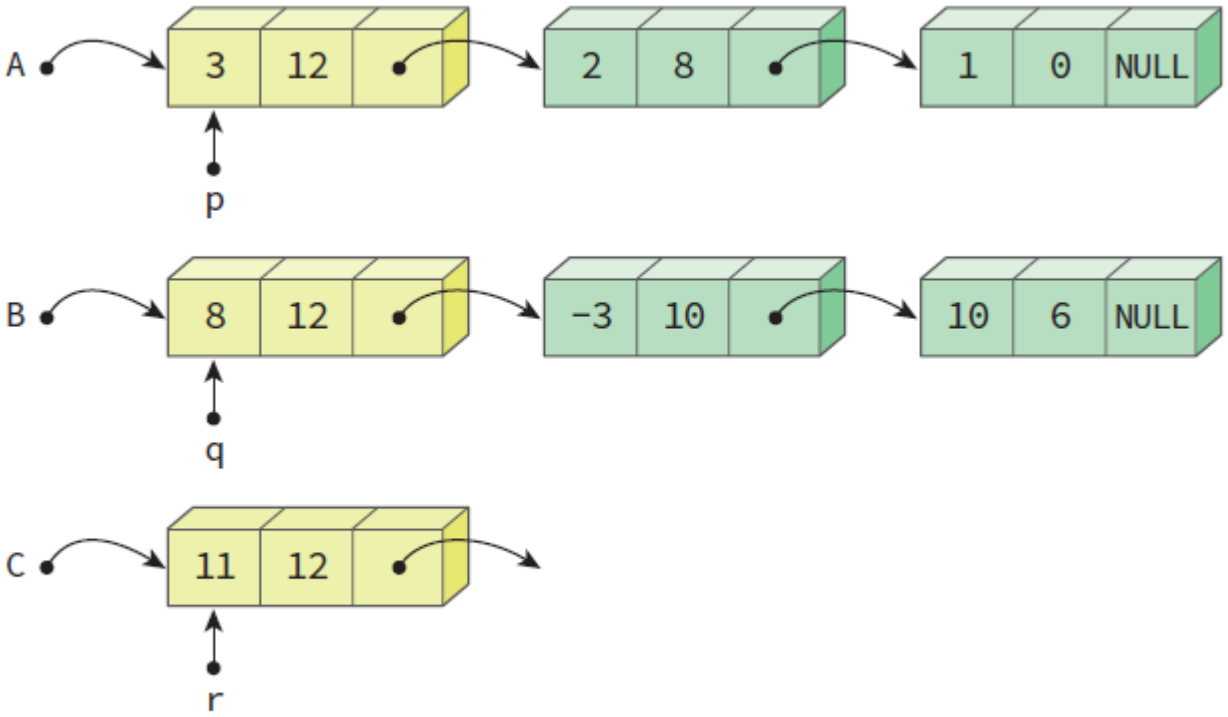


다항식의 덧셈 구현

- 2개의 다항식을 더하는 덧셈 연산을 구현

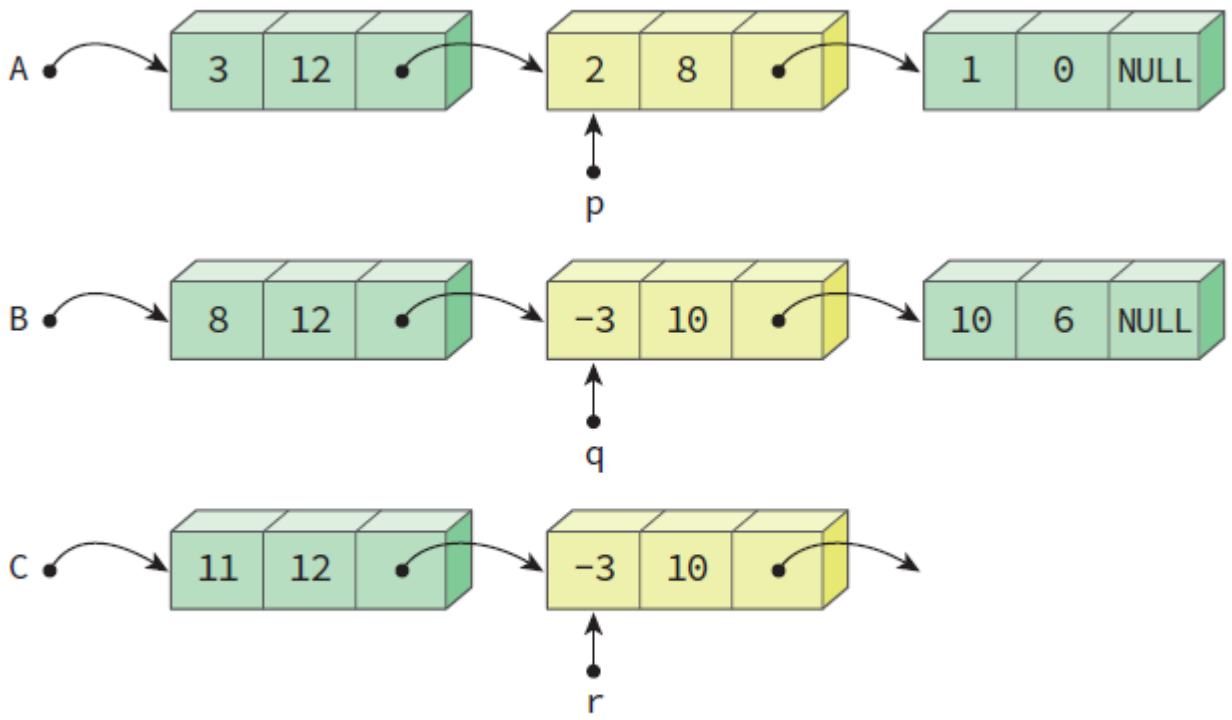
- $A=3x^{12}+2x^{8}+1$, $B=8x^{12}-3x^{10}+10x^{6}$이면
- $A+B=11x^{12}-3x^{10}+2x^{8}+10x^{6}+1$

다항식의 덧셈



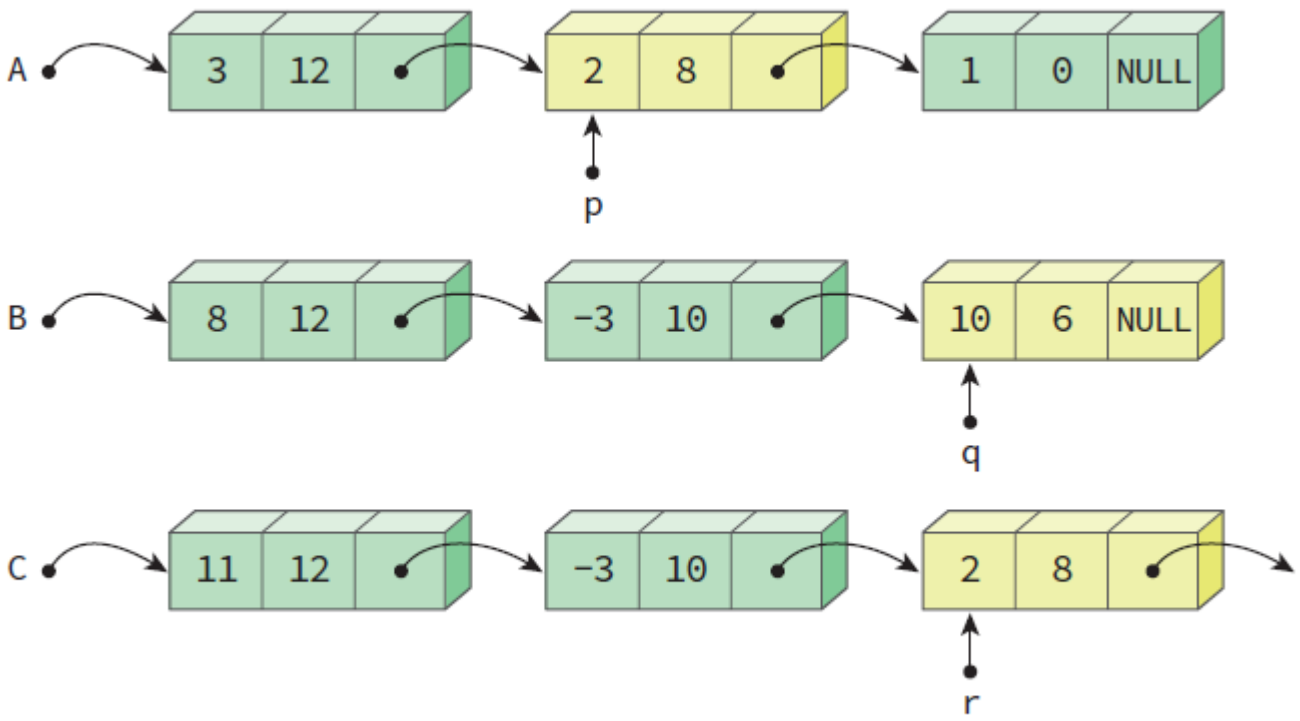
(a) p와 q가 가리키는 항들의 지수가 같으면 계수를 더한다.

다항식의 덧셈



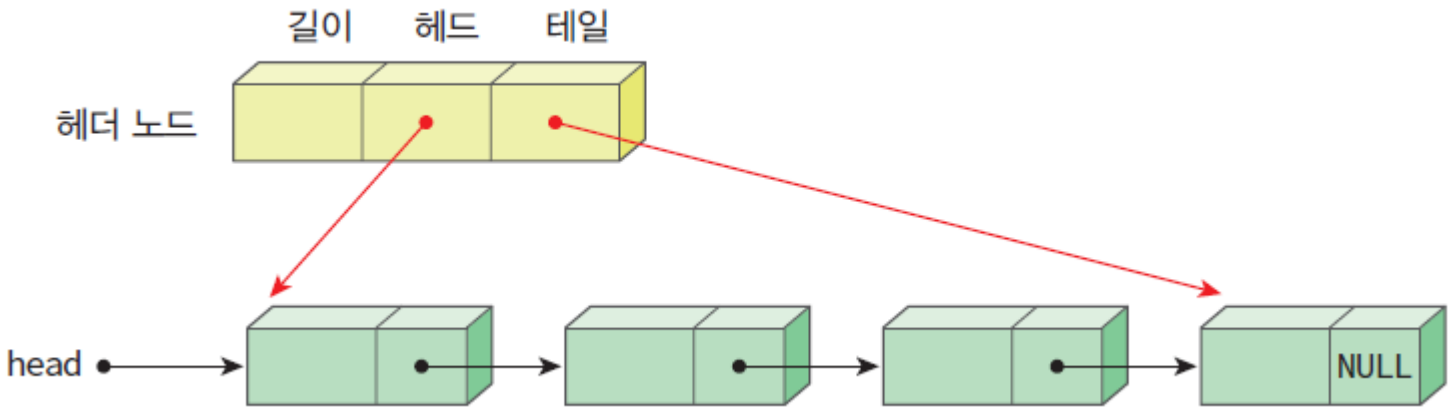
(b) q가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.

다항식의 덧셈



(c) p가 가리키는 항의 지수가 높으면 그대로 C로 옮긴다.

헤더 노드의 개념



다항식 프로그램

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

// 연결 리스트 헤더
typedef struct ListType { // 리스트 헤더 타입
        int size;
        ListNode *head;
        ListNode *tail;
} ListType;

// 오류 함수
void error(char *message)
{
        fprintf(stderr, "%s\n", message);
        exit(1);
}
```

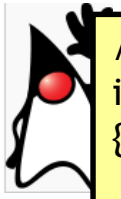
```
// 리스트 헤더 생성 함수
ListType* create()
{
        ListType *plist = (ListType *)malloc(sizeof(ListType));
        plist->size = 0;
        plist->head = plist->tail = NULL;
        return plist;
}

// plist는 연결 리스트의 헤더를 가리키는 포인터, coef는 계수,
// expon는 지수
void insert_last(ListType* 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->size++;
}
```

다항식 프로그램

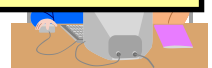
```
// list3 = list1 + list2
void poly_add(ListType* plist1, ListType* plist2, ListType* plist3)
{
        ListNode* a = plist1->head;
        ListNode* b = plist2->head;
        int sum;

        while (a && b) {
                if (a->expon == b->expon) {    // a의 차수 > b의 차수
                        sum = a->coef + b->coef;
                        if (sum != 0) insert_last(plist3, sum, a->expon);
                        a = a->link; b = b->link;
                }
                else if (a->expon > b->expon) {  // a의 차수 == b의 차수
                        insert_last(plist3, a->coef, a->expon);
                        a = a->link;
                }
                else {                                                  // a의 차수 < b의 차수
                        insert_last(plist3, b->coef, b->expon);
                        b = b->link;
                }
        }
```

다항식 프로그램

```
                // a나 b중의 하나가 먼저 끝나게 되면 남아있는 항들을 모두
                // 결과 다항식으로 복사
                for (; a != NULL; a = a->link)
                        insert_last(plist3, a->coef, a->expon);
                for (; b != NULL; b = b->link)
                        insert_last(plist3, b->coef, b->expon);
}
//
//
void poly_print(ListType* plist)
{
                ListNode* p = plist->head;

                printf("polynomial = ");
                for (; p; p = p->link) {
                        printf("%d^%d + ", p->coef, p->expon);
                }
                printf("\n");
}
```



```
//
int main(void)
{
        ListType *list1, *list2, *list3;

        // 연결 리스트 헤더 생성
        list1 = create();
        list2 = create();
        list3 = create();

        // 다항식 1을 생성
        insert_last(list1, 3, 12);
        insert_last(list1, 2, 8);
        insert_last(list1, 1, 0);

        // 다항식 2를 생성
        insert_last(list2, 8, 12);
        insert_last(list2, -3, 10);
        insert_last(list2, 10, 6);

        poly_print(list1);
        poly_print(list2);

        // 다항식 3 = 다항식 1 + 다항식 2
        poly_add(list1, list2, list3);
        poly_print(list3);

        free(list1); free(list2); free(list3);
}
```

실행결과

```
polynomial = 3^12 + 2^8 + 1^0 +
polynomial = 8^12 + -3^10 + 10^6 +
polynomial = 11^12 + -3^10 + 2^8 + 10^6 + 1^0 +
```