

4장: 리스트

- 목차
 - 1. 리스트란?
 - 2. 배열로 구현된 리스트
 - 3. 연결 리스트
 - 4. 연결 리스트로 구현된 리스트
 - 5. 선형 리스트의 응용 : 다항식

1. 리스트란?

- 리스트(list), 선형리스트(linear list): 순서를 가진 항목들의 모임

- 집합: 항목간의 순서의 개념이 없음

$$L = (item_0, item_1, \dots, item_{n-1})$$

- 리스트의 예

- 요일: (일요일, 월요일, ..., 토요일)

- 한글 자음의 모임: (ㄱ, ㄴ, ..., ㅎ)

- 카드: (Ace, 2, 3, ..., King)

- 핸드폰의 문자 메시지 리스트



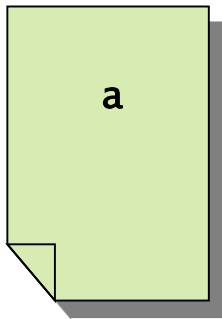
리스트의 연산

- 새로운 항목을 리스트의 끝, 처음, 중간에 추가한다.
- 기존의 항목을 리스트의 임의의 위치에서 삭제한다.
- 모든 항목을 삭제한다.
- 기존의 항목을 대치한다.
- 리스트가 특정한 항목을 가지고 있는지를 살핀다.
- 리스트의 특정위치의 항목을 반환한다.
- 리스트안의 항목의 개수를 센다.
- 리스트가 비었는지, 꽉 찼는지를 체크한다.
- 리스트안의 모든 항목을 표시한다.

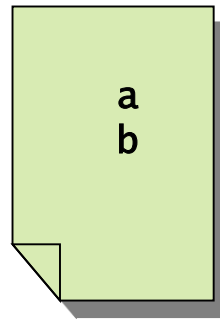
리스트 ADT

- 객체: n개의 element형으로 구성된 순서있는 모임
- 연산:
 - `add_last(list, item)` ::= 맨끝에 요소를 추가한다.
 - `add_first(list, item)` ::= 맨처음에 요소를 추가한다.
 - `add(list, pos, item)` ::= `pos` 위치에 요소를 추가한다.
 - `delete(list, pos)` ::= `pos` 위치의 요소를 제거한다.
 - `clear(list)` ::= 리스트의 모든 요소를 제거한다.
 - `replace(list, pos, item)` ::=
pos 위치의 요소를 `item`로 바꾼다.
 - `is_in_list(list, item)` ::= `item`이 리스트안에 있는지를 검사한다.
 - `get_entry(list, pos)` ::= `pos` 위치의 요소를 반환한다.
 - `get_length(list)` ::= 리스트의 길이를 구한다.
 - `is_empty(list)` ::= 리스트가 비었는지를 검사한다.
 - `is_full(list)` ::= 리스트가 꽉찼는지를 검사한다.
 - `display(list)` ::= 리스트의 모든 요소를 표시한다.

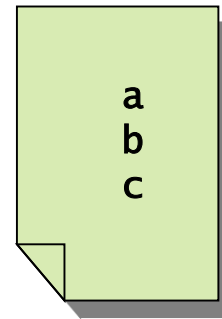
리스트 ADT 사용예 #1



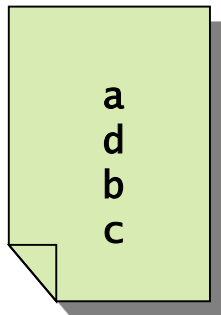
`addTail(list1,a)`



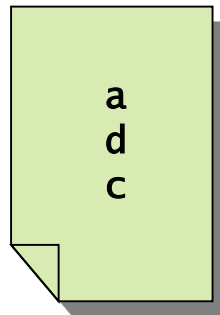
`addTail(list1,b)`



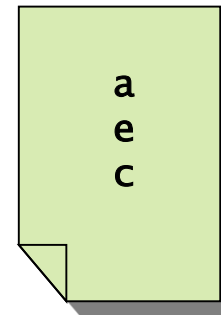
`add(list1,2,c)`



`add(list1,1,d)`



`delete(list1,2)`



`replace(list1,1,e)`

리스트 ADT 사용예 #2

```
•main()
{
    int i, n;

    // list2를 생성한다: 구현방법에 따라 약간씩 다름
    ListType list2;
    add_tail(&list2, "마요네즈"); // 리스트의 포인터를 전달
    add_tail(&list2, "빵");
    add_tail(&list2, "치즈");
    add_tail(&list2, "우유");
    display(&list2);
    n = get_length(&list2);
    printf("쇼핑해야할 항목수는 %d입니다.\n", n);
    for(i=0; i<n; i++)
        printf("%d항목은 %s입니다. ", i, get_entry(&list2, i));
}
```

리스트 구현 방법

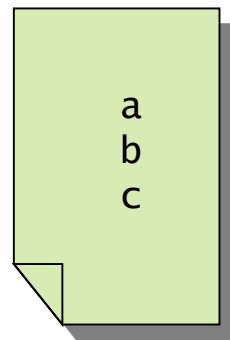
- 배열을 이용하는 방법

- 구현이 간단
- 삽입, 삭제시 오버헤드
- 항목의 개수 제한

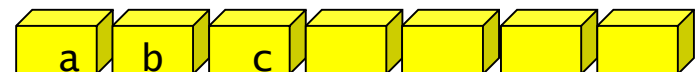
- 연결리스트를 이용하는 방법

- 구현이 복잡
- 삽입, 삭제가 효율적
- 크기가 제한되지 않음

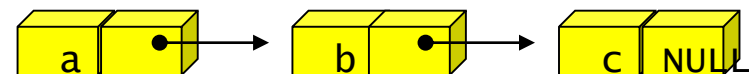
리스트 ADT



배열을 이용한 구현

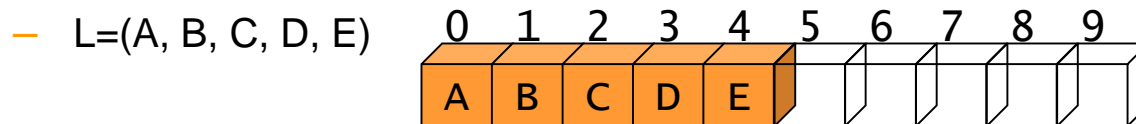


연결리스트를 이용한 구현

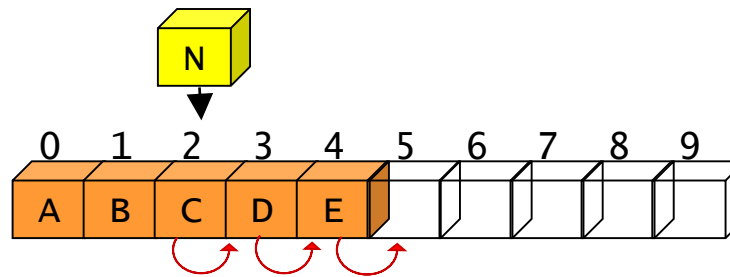


2. 배열로 구현된 리스트

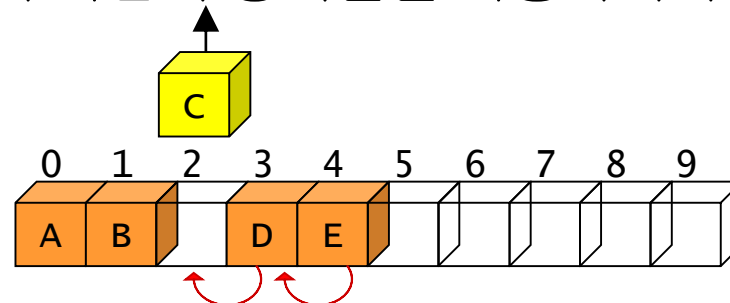
- 1차원 배열에 항목들을 순서대로 저장



- 삽입연산: 삽입위치 다음의 항목들을 이동하여야 함.



- 삭제연산: 삭제위치 다음의 항목들을 이동하여야 함



ArrayListType의 구현

- 항목들의 타입은 `element`로 정의
- `list`라는 1차원 배열에 항목들을 차례대로 저장
- `length`에 항목의 개수 저장

```
typedef int element;
typedef struct {
    int list[MAX_LIST_SIZE]; // 배열 정의
    int length;               // 현재 배열에 저장된 항목들의 개수
} ArrayListType;
```

```
// 리스트 초기화
void init(ArrayListType *L)
{
    L->length = 0;
}
```

ArrayListType의 구현

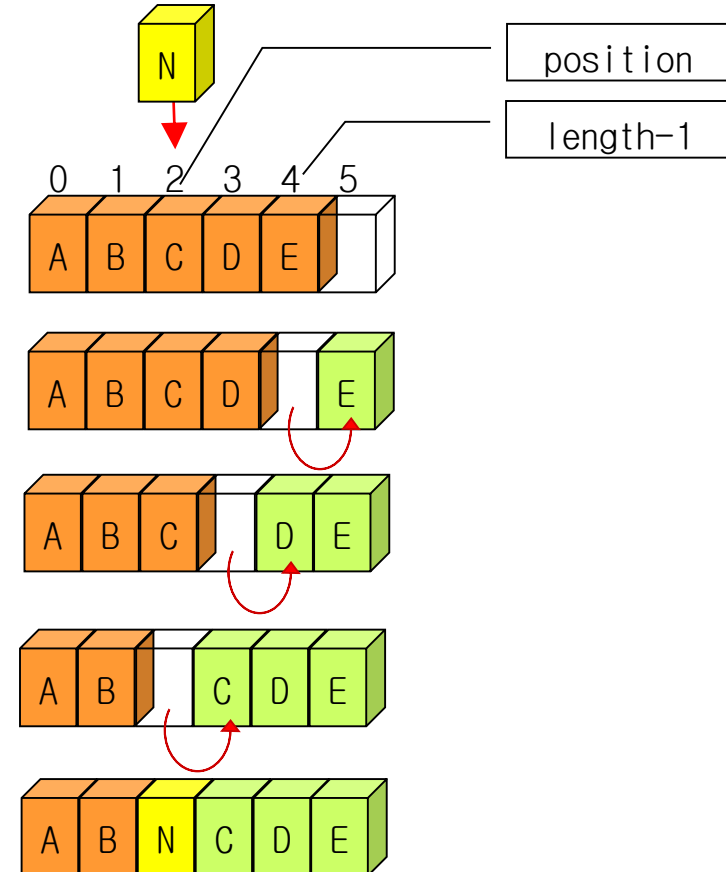
- is_empty 연산과 is_full 연산의 구현

```
// 리스트가 비어 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_empty(ArrayListType *L)
{
    return L->length == 0;
}
// 리스트가 가득 차 있으면 1을 반환
// 그렇지 않으면 0을 반환
int is_full(ArrayListType *L)
{
    return L->length == MAX_LIST_SIZE;
}
```

ArrayListType의 삽입 연산

- add 함수는 먼저 배열이 포화상태인지를 검사하고 삽입위치가 적합한 범위에 있는지를 검사한다.
- 삽입 위치 다음에 있는 자료들을 한칸씩 뒤로 이동한다

```
// position: 삽입하고자 하는 위치
// item: 삽입하고자 하는 자료
void add(ArrayListType *L, int position, element item)
{
    if( !is_full(L) && (position >= 0) &&
        (position <= L->length) ){
        int i;
        for(i=(L->length-1); i>=position;i--){
            L->list[i+1] = L->list[i];
        }
        L->list[position] = item;
        L->length++;
    }
}
```

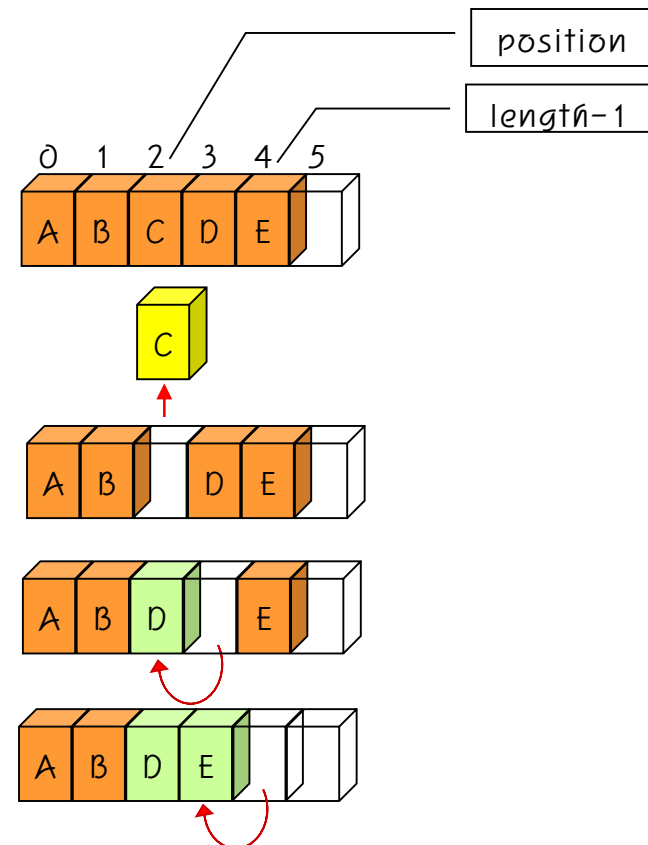


ArrayListType의 삭제 연산

- 삭제 위치를 검사한다.
- 삭제위치부터 맨끝까지의 자료를 한칸씩 앞으로 옮긴다.

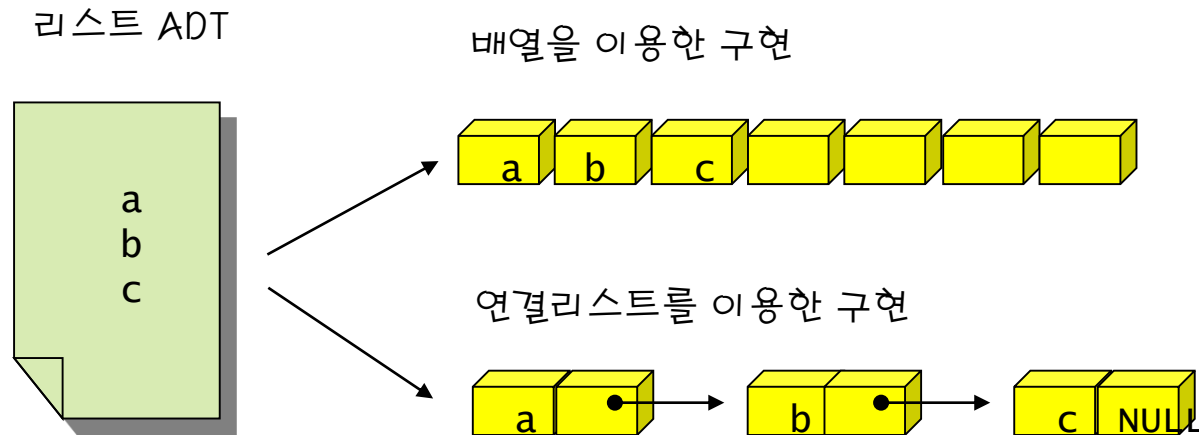
```
// position: 삭제하고자 하는 위치
// 반환값: 삭제되는 자료
element delete(ArrayListType *L, int position)
{
    int i;
    element item;

    if( position < 0 || position >= L->length )
        error("위치 오류");
    item = L->list[position];
    for(i=position; i<(L->length-1);i++)
        L->list[i] = L->list[i+1];
    L->length--;
    return item;
}
```



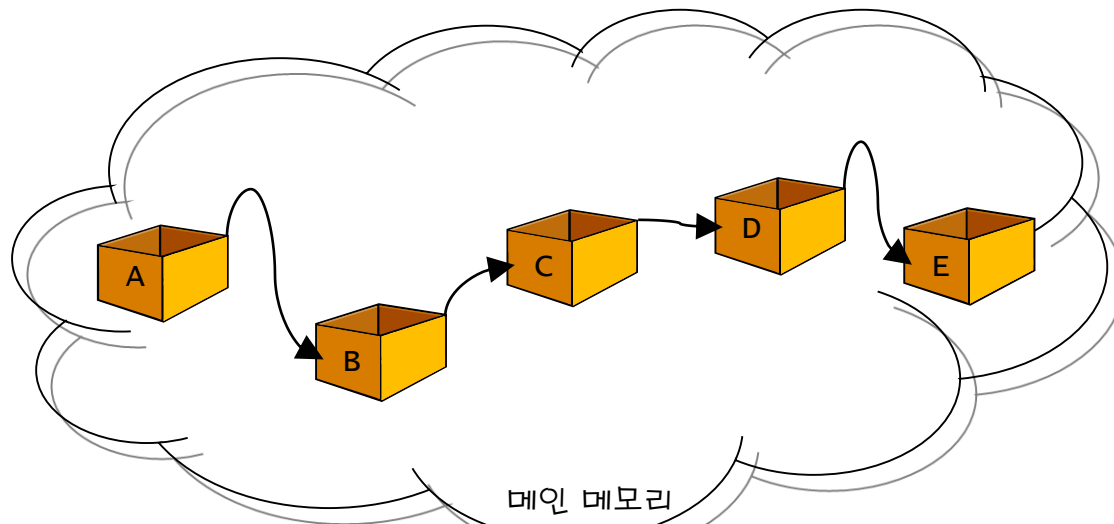
3. 연결 리스트

- 리스트 표현의 2가지 방법
 - 순차 표현: 배열을 이용한 리스트 표현
 - 연결된 표현: 연결 리스트를 사용한 리스트 표현, 하나의 노드가 데이터와 링크로 구성되어 있고 링크가 노드들을 연결한다.



연결된 표현

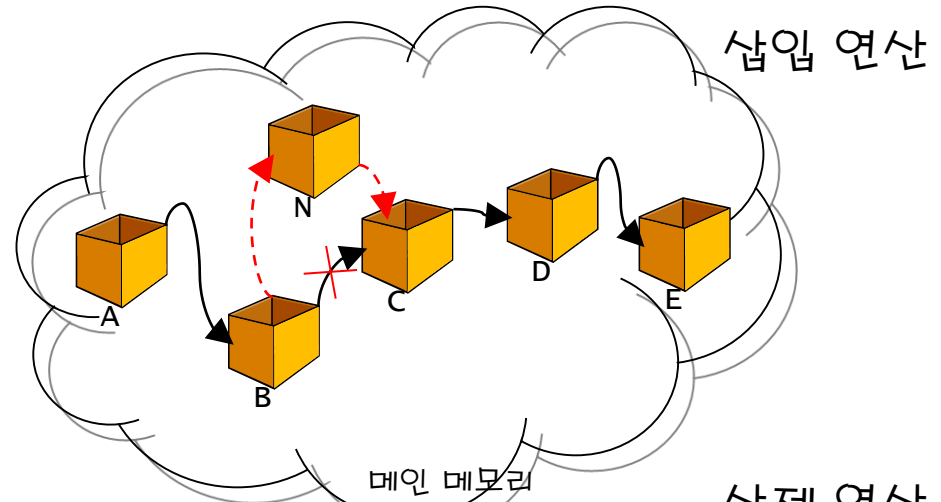
- 리스트의 항목들을 노드(node)라고 하는 곳에 분산하여 저장
- 다음 항목을 가리키는 주소도 같이 저장
- 노드 (node) : <항목, 주소> 쌍
- 노드는 데이터 필드와 링크 필드로 구성
 - 데이터 필드 - 리스트의 원소, 즉 데이터 값을 저장하는 곳
 - 링크 필드 - 다른 노드의 주소값을 저장하는 장소 (포인터)
- 메모리안에서의 노드의 물리적 순서가 리스트의 논리적 순서와 일치할 필요 없음



연결된 표현의 장단점

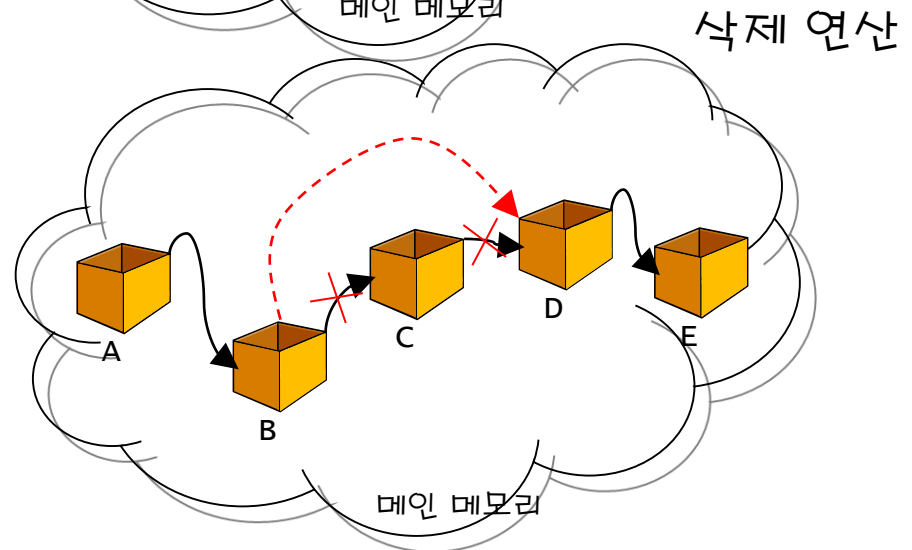
장점

- 삽입, 삭제가 보다 용이하다.
- 연속된 메모리 공간이 필요없다.
- 크기 제한이 없다



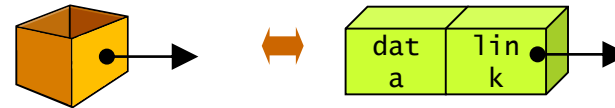
단점

- 구현이 어렵다.
- 오류가 발생하기 쉽다.

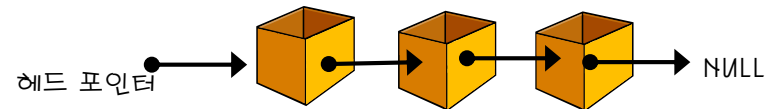


연결 리스트의 구조

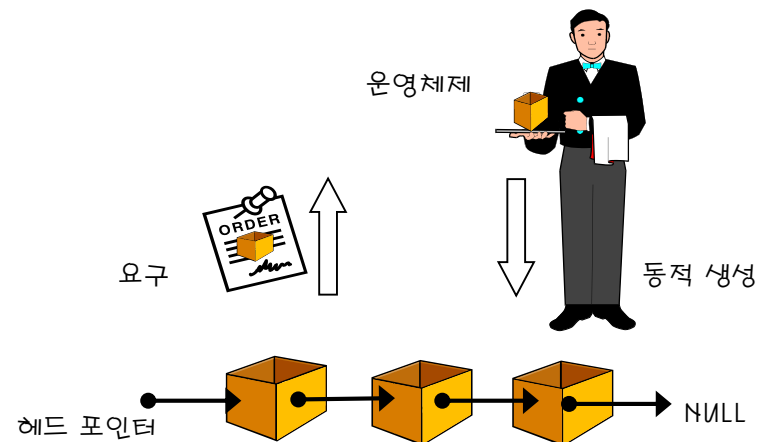
- 노드 = 데이터 필드 + 링크 필드



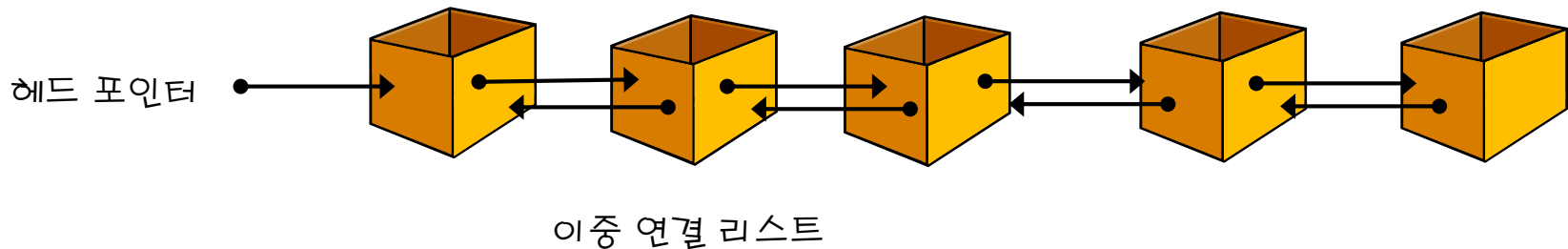
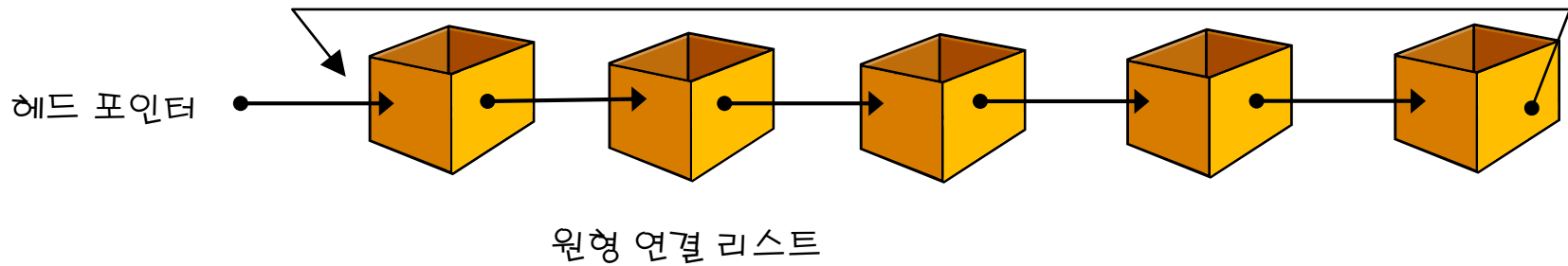
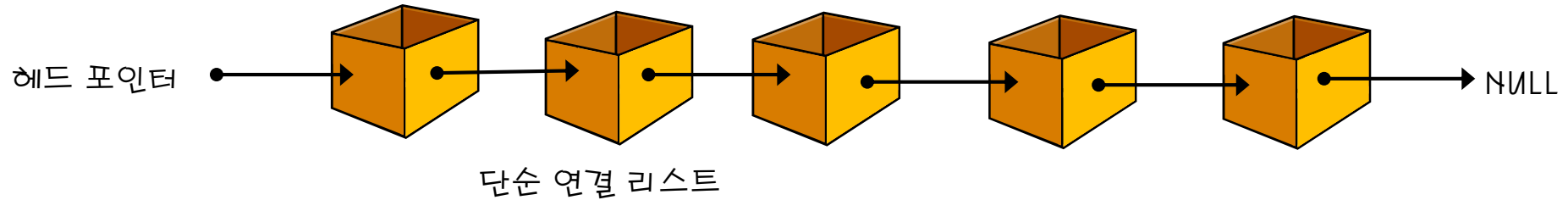
- 헤드 포인터(head pointer): 리스트의 첫번째 노드를 가리키는 변수



- 노드의 생성: 필요할 때마다 동적 메모리 생성 이용하여 노드를 생성

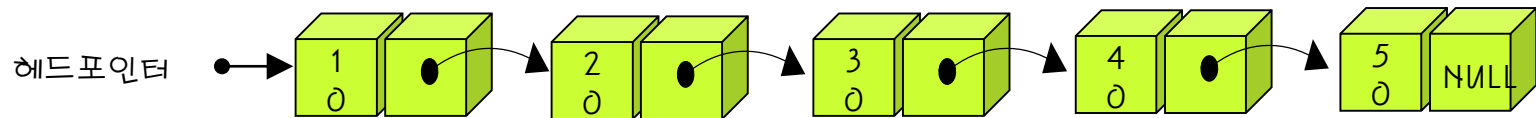


연결 리스트의 종류

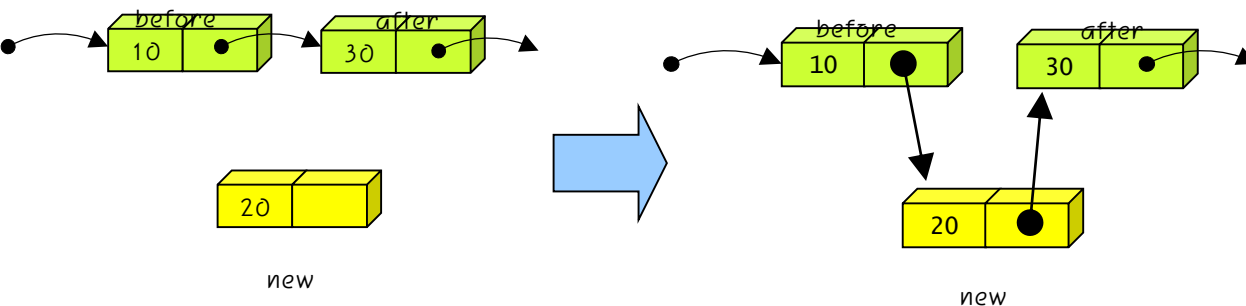


4. 연결 리스트로 구현된 리스트

- 하나의 링크 필드를 이용하여 연결
- 마지막 노드의 링크값은 NULL



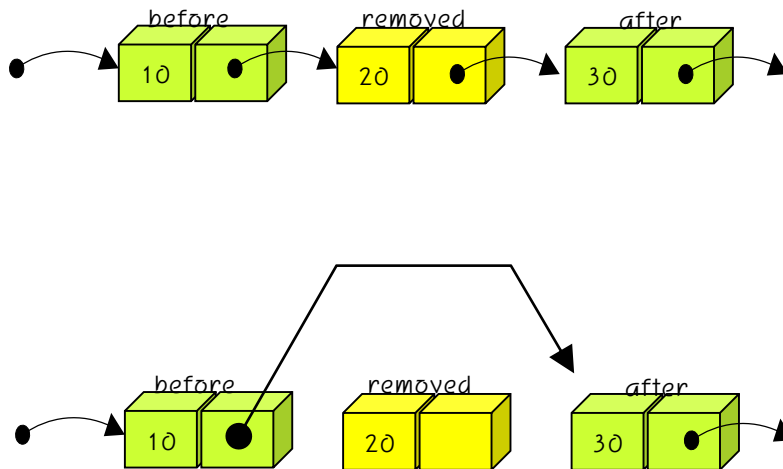
- 삽입연산



```
insert_node(L, before, new)
```

```
if L = NULL
then L ← new
else new.link ← before.link
    before.link ← new
```

단순연결리스트(삭제연산)



```
remove_node(L, before, removed)
```

```
if L ≠ NULL
```

```
then before.link ← removed.link  
destroy(removed)
```

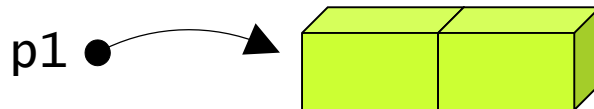
단순 연결 리스트의 구현

- 데이터 필드: 구조체로 정의
- 링크 필드: 포인터 사용

```
typedef int element;  
typedef struct ListNode {  
    element data;  
    struct ListNode *link;  
} ListNode;
```

- 노드의 생성: 동적 메모리 생성 라이브러리 malloc 함수이용

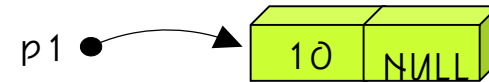
```
ListNode *p1;  
p1 = (ListNode *)malloc(sizeof(ListNode));
```



단순 연결 리스트의 구현

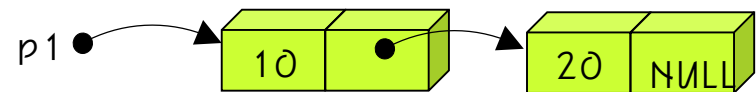
- 데이터 필드와 링크 필드 설정

```
p1->data = 10;  
p1->link = NULL;
```



- 두번째 노드 생성과 첫번째 노드와의 연결

```
ListNode *p2;  
p2 = (ListNode *)malloc(sizeof(ListNode));  
p2->data = 20;  
p2->link = NULL;  
p1->link = p2;
```



- 헤드포인터(head pointer): 연결 리스트의 맨 처음 노드를 가리키는 포인터

단순 연결 리스트의 삽입연산

- 삽입 함수의 프로토타입

```
void insert_node(ListNode **phead, ListNode *p, ListNode *new_node)
```

phead: 헤드 포인터 *head*에 대한 포인터

p: 삽입될 위치의 선행노드를 가리키는 포인터, 이 노드 다음에 삽입된다.

new_node: 새로운 노드를 가리키는 포인터

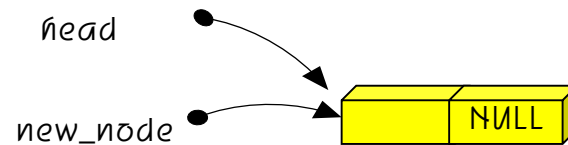
- 헤드포인터가 함수 안에서 변경되므로 헤드포인터의 포인터 필요

- 삽입의 3가지 경우

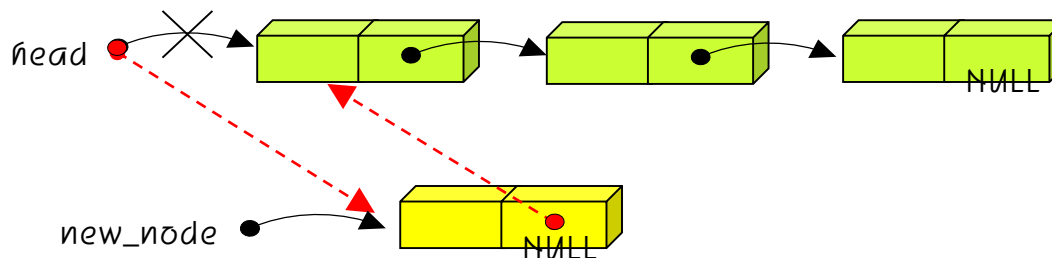
- *head*가 NULL인 경우: 공백 리스트에 삽입
- *p*가 NULL인 경우: 리스트의 맨처음에 삽입
- 일반적인 경우: 리스트의 중간에 삽입

삽입연산

- (1) head가 NULL인 경우: head가 NULL이라면 현재 삽입하려는 노드가 첫 번째 노드가 된다. 따라서 head의 값만 변경하면 된다.

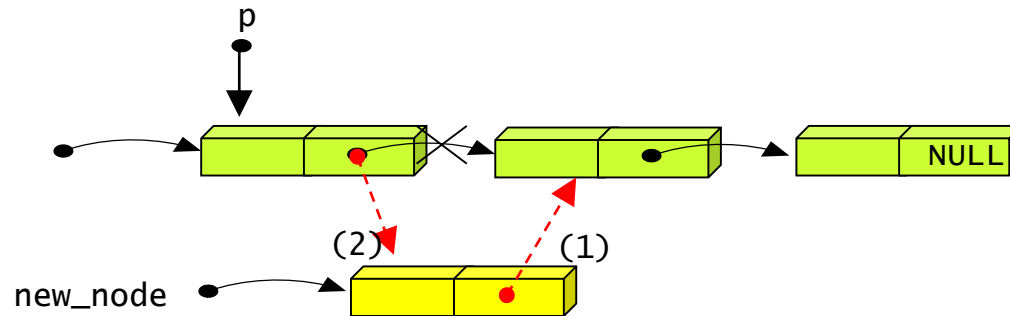


- (2) p가 NULL인 경우: 새로운 노드를 리스트의 맨앞에 삽입한다.



삽입연산

- (3) head와 p가 NULL이 아닌 경우: 가장 일반적인 경우이다.
new_node의 link에 p->link값을 복사한 다음, p->link가 new_node를 가리키도록 한다.



삽입연산의 코드

```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드
// new_node : 삽입될 노드
void insert_node(ListNode **phead, ListNode *p, ListNode *new_node)
{
    if( *phead == NULL )
    {
        // 공백리스트인 경우
        new_node->link = NULL;
        *phead = new_node;
    }
    else if( p == NULL )
    {
        // p가 NULL이면 첫번째 노드로 삽입
        new_node->link = *phead;
        *phead = new_node;
    }
    else
    {
        // p 다음에 삽입
        new_node->link = p->link;
        p->link = new_node;
    }
}
```

삭제 연산

- 삭제함수의 프로토타입

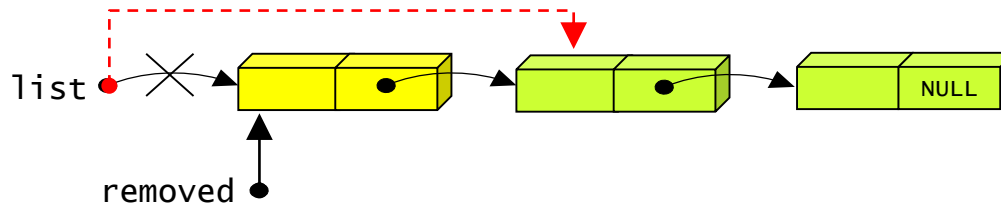
```
//phead: 헤드 포인터 head의 포인터  
//p: 삭제될 노드의 선행 노드를 가리키는 포인터  
//removed: 삭제될 노드를 가리키는 포인터
```

```
void remove_node(ListNode **phead, ListNode *p, ListNode *removed)
```

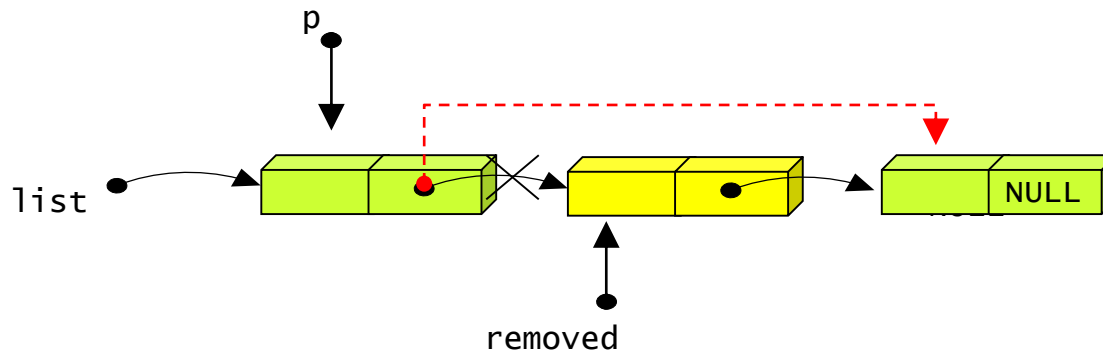
- 삭제의 2가지 경우
 - p가 NULL인 경우: 맨 앞의 노드를 삭제
 - p가 NULL이 아닌 경우: 중간 노드를 삭제

삭제 연산

- p가 NULL인 경우: 연결 리스트의 첫 번째 노드를 삭제한다.
헤드포인터 변경



- p가 NULL이 아닌 경우: removed 앞의 노드인 p의 링크가 removed 다음 노드를 가리키도록 변경



삭제 연산 코드

```
// phead : 헤드 포인터에 대한 포인터
// p: 삭제될 노드의 선행 노드
// removed: 삭제될 노드
void remove_node(ListNode **phead, ListNode *p, ListNode *removed)
{
    if( p == NULL )
        *phead = (*phead)->link;
    else
        p->link = removed->link;
    free(removed);
}
```

방문 연산 코드

- 방문 연산: 리스트 상의 노드를 순차적으로 방문
- 반복과 순환기법을 모두 사용가능
- 반복버전

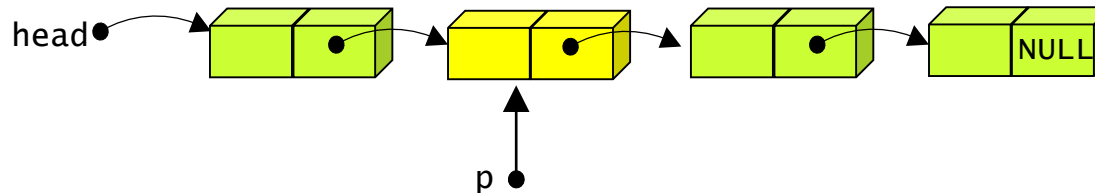
```
void display(ListNode *head)
{
    ListNode *p=head;
    while( p != NULL )
    {
        printf("%d->", p->data);
        p = p->link;
    }
    printf("\n");
}
```

- 순환버전

```
void display_recur(ListNode *head)
{
    ListNode *p=head;
    if( p != NULL )
    {
        printf("%d->", p->data);
        display_recur(p->link);
    }
}
```

탐색 연산 코드

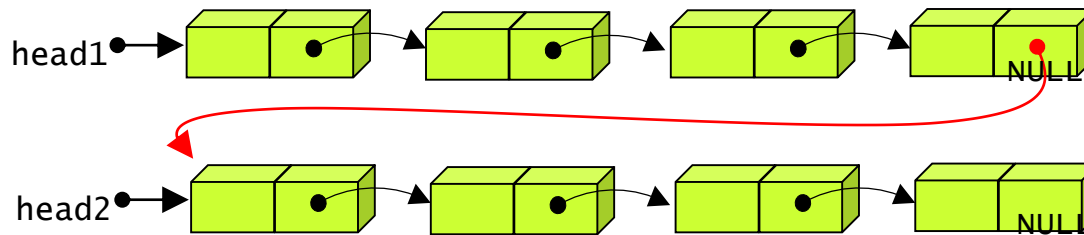
- 탐색 연산: 특정한 데이터값을 갖는 노드를 찾는 연산



```
ListNode *search(ListNode *head, int x)
{
    ListNode *p;
    p = head;
    while( p != NULL )
    {
        if( p->data == x ) return p; // 탐색 성공
        p = p->link;
    }
    return p; // 탐색 실패일 경우 NULL 반환
}
```

합병 연산 코드

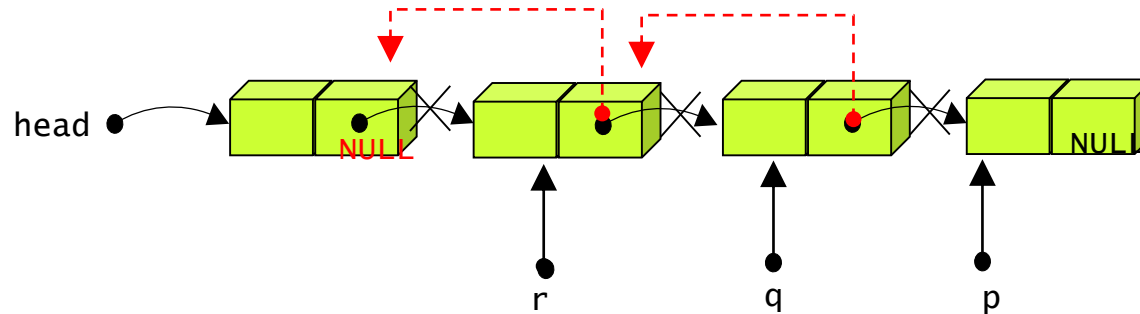
- 합병 연산: 2개의 리스트를 합하는 연산



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

역순 연산 코드

- 역순 연산: 리스트의 노드들을 역순으로 만드는 연산



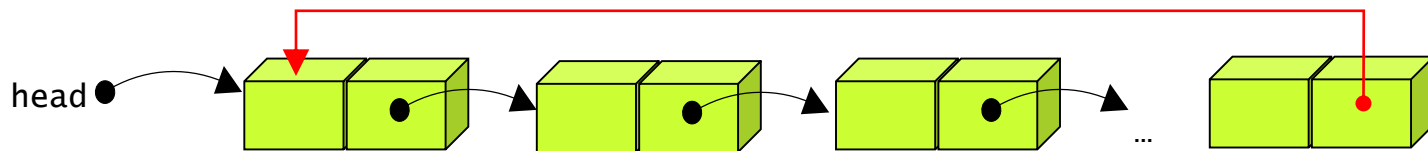
```

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; // q는 역순으로 된 리스트의 헤드 포인터
}

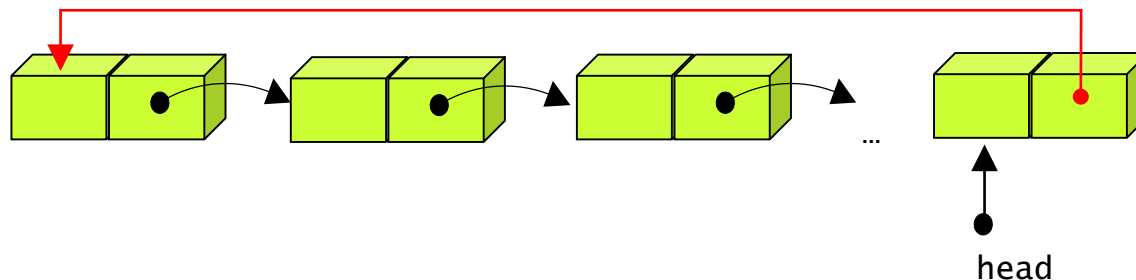
```


원형 연결 리스트

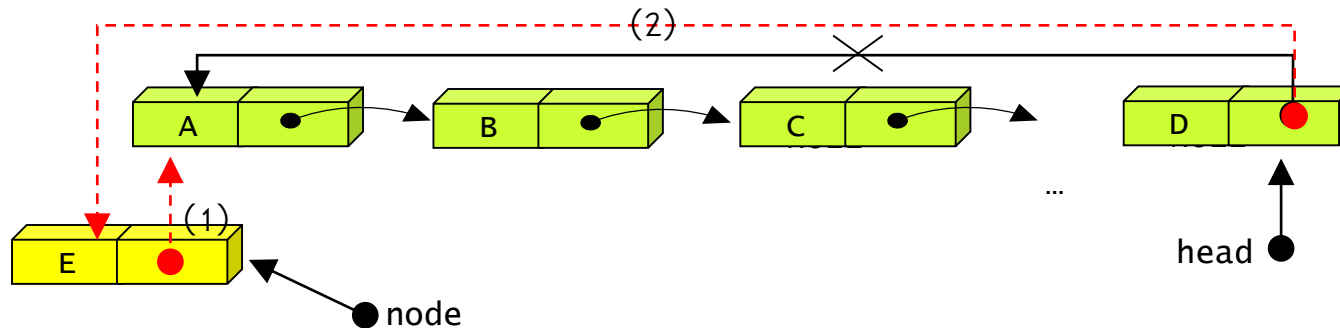
- 마지막 노드의 링크가 첫번째 노드를 가리키는 리스트
- 한 노드에서 다른 모든 노드로의 접근이 가능



- 보통 헤드포인터가 마지막 노드를 가리키게끔 구성하면 리스트의 처음이나 마지막에 노드를 삽입하는 연산이 단순 연결 리스트에 비하여 용이

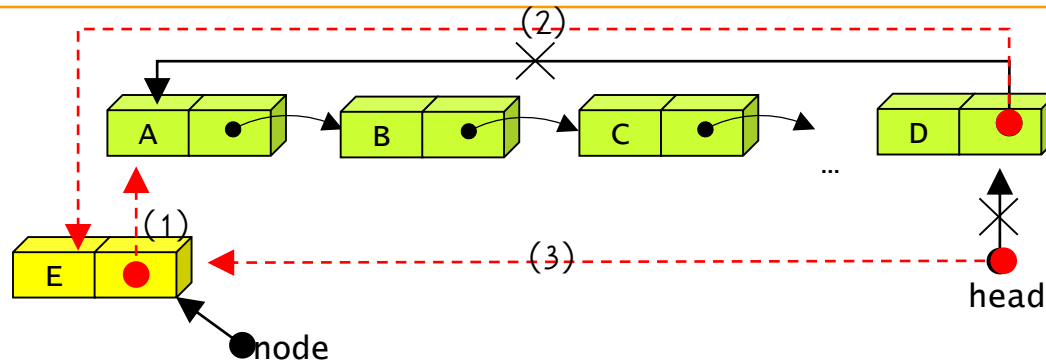


리스트의 처음에 삽입



```
// phead: 리스트의 헤드 포인터의 포인터
// 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;
    }
}
```

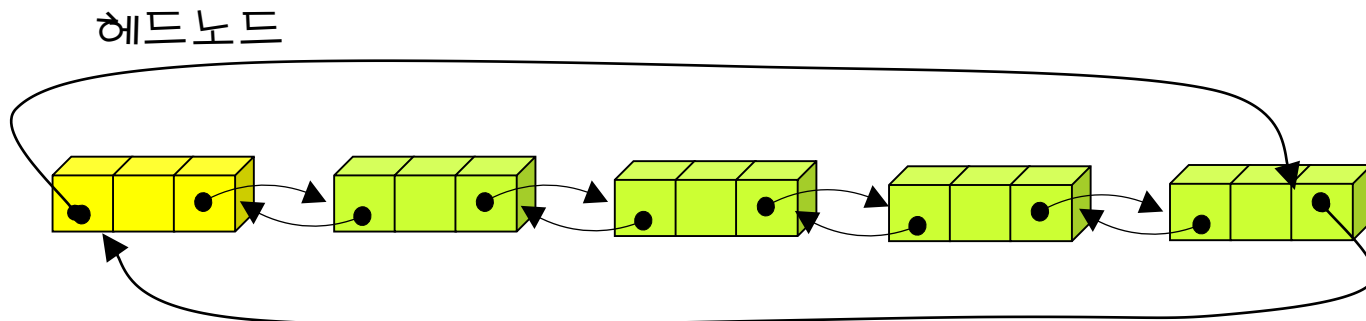
리스트의 끝에 삽입



```
// 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;
    }
}
```

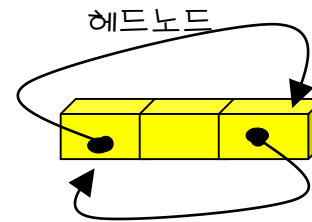
이중 연결 리스트

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



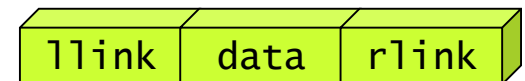
헤드노드

- 헤드노드(head node): 데이터를 가지지 않고 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드
- 헤드 포인터와의 구별 필요
- 공백상태에서는 헤드 노드만 존재

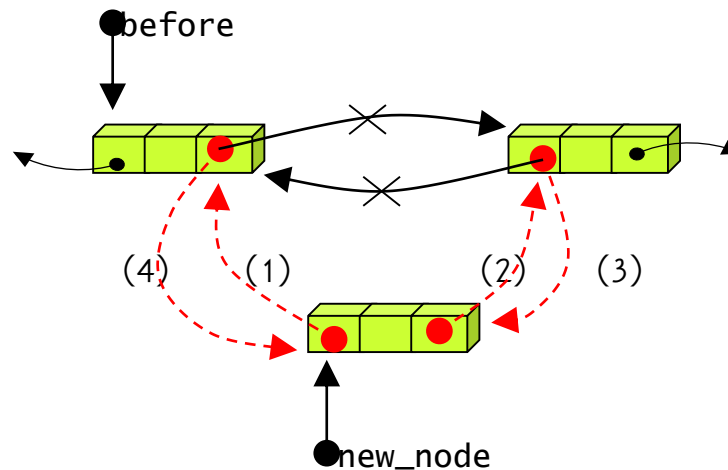


- 이중연결리스트에서의 노드의 구조

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

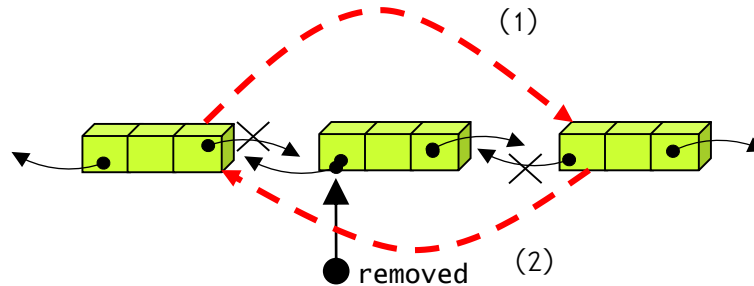


삽입연산



```
// 노드 new_node를 노드 before의 오른쪽에 삽입한다.  
void dinser_node(DListNode *before, DListNode *new_node)  
{  
    new_node->llink = before;  
    new_node->rlink = before->rlink;  
    before->rlink->llink = new_node;  
    before->rlink = new_node;  
}
```

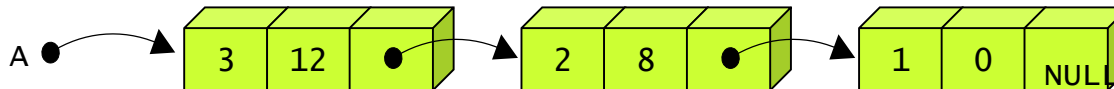
삭제연산



```
// 노드 removed를 삭제한다.  
void dremove_node(DListNode *phead_node,  
                  DListNode *removed)  
{  
    if( removed == phead_node ) return;  
    removed->llink->rlink = removed->rlink;  
    removed->rlink->llink = removed->llink;  
    free(removed);  
}
```

5. 연결리스트의 응용: 다항식

- 다항식을 컴퓨터로 처리하기 위한 자료구조
 - 다항식의 덧셈, 뺄셈...
- 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 구현

- 2개의 다항식을 더하는 덧셈 연산을 구현

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

- 다항식 A와 B의 항들을 따라 순회하면서 각 항들을 더한다.

① $p.\text{expon} == q.\text{expon}$:

두 계수를 더해서 0이 아니면 새로운 항을 만들어 결과 다항식 C에 추가한다.
그리고 p와 q는 모두 다음 항으로 이동한다.

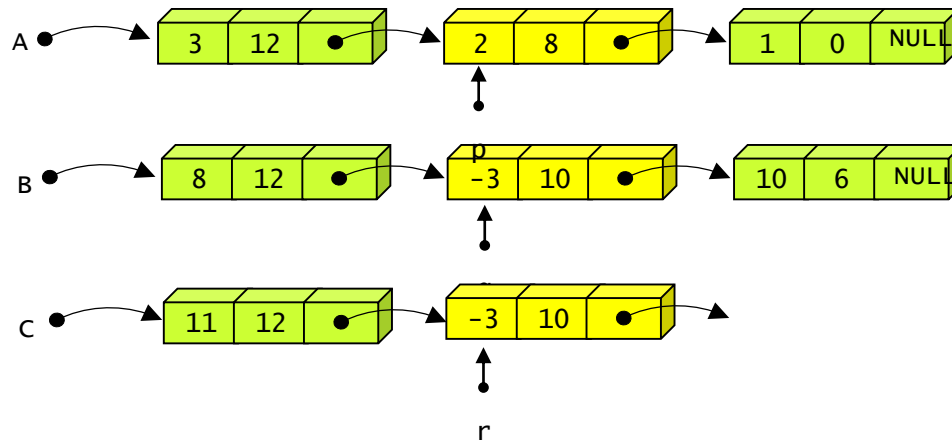
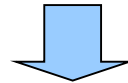
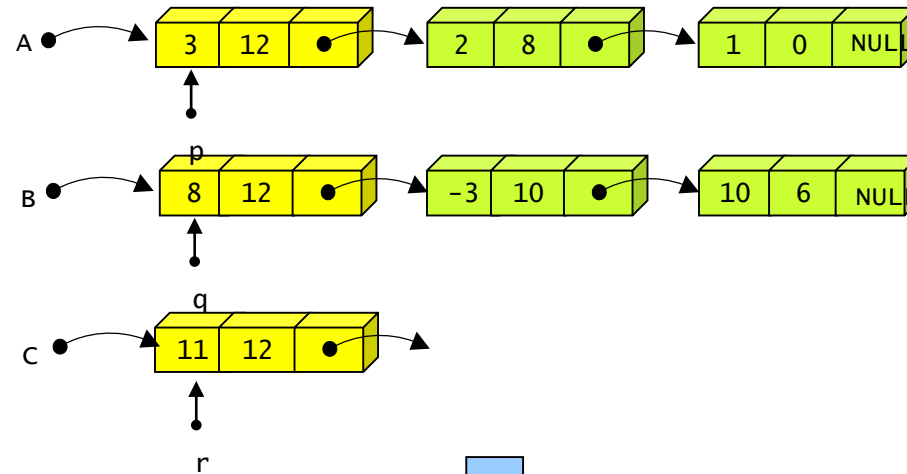
② $p.\text{expon} < q.\text{expon}$:

q가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C에 추가한다. 그리고
q만 다음 항으로 이동한다.

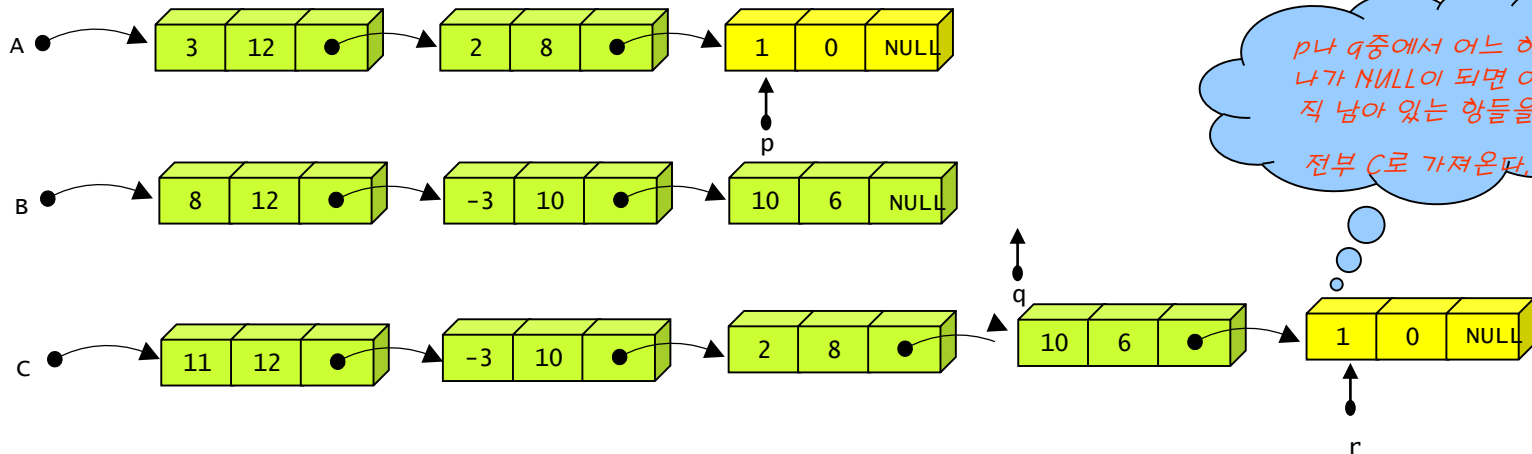
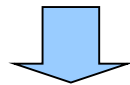
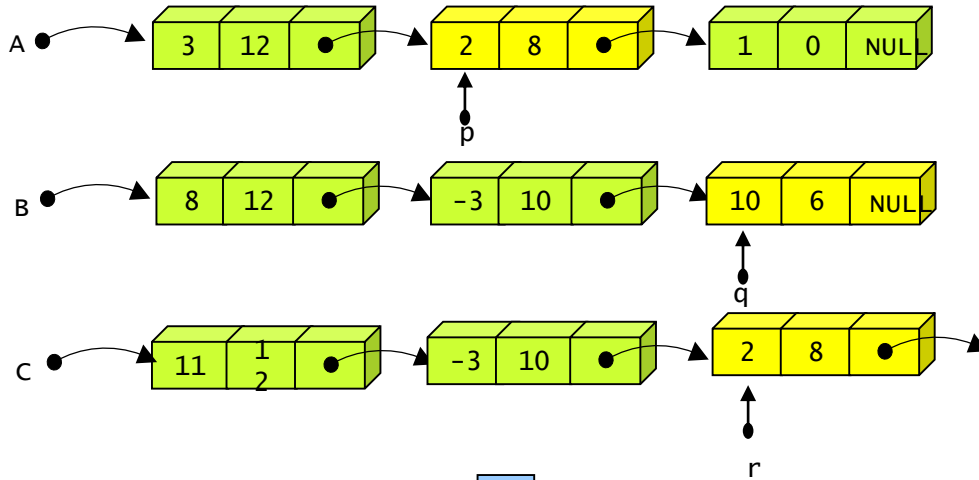
③ $p.\text{expon} > q.\text{expon}$:

p가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C에 추가한다. 그리고
p만 다음 항으로 이동한다.

다항식의 덧셈



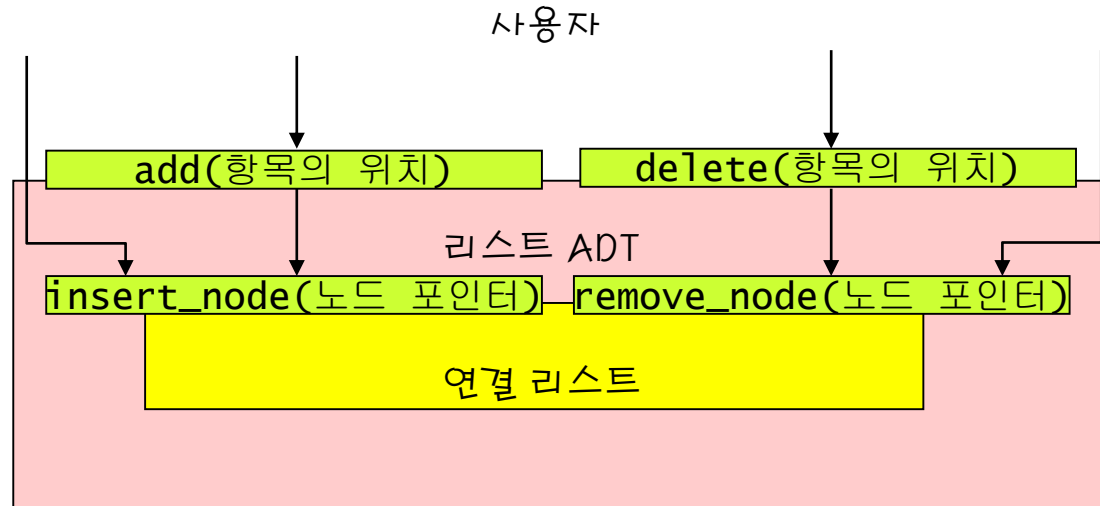
다항식의 덧셈



p나 q중에서 어느 하나가 NULL이 되면 아직 남아 있는 항들을 전부 C로 가져온다.

연결리스트를 이용한 리스트 ADT의 구현

- 리스트 ADT의 연산을 연결리스트를 이용하여 구현
- 리스트 ADT의 add, delete 연산의 파라미터는 위치
- 연결리스트의 insert_node, remove_node의 파라미터는 노드 포인터
- 상황에 따라 연산을 적절하게 선택하여야 함



리스트 ADT의 구현

```
typedef struct {  
    ListNode *head;    // 헤드 포인터  
    int length;        // 노드의 개수  
} ListType;  
  
ListType list1;
```

첫 번째 노드를 가리키는 헤드 포인터

연결 리스트내의 존재하는 노드의 개수

리스트 ADT의 생성

is_empty, get_length 연산의 구현

```
int is_empty(ListType *list)
{
    if( list->head == NULL ) return 1;
    else return 0;
}
```

```
// 리스트의 항목의 개수를 반환한다.
int get_length(ListType *list)
{
    return list->length;
}
```

add 연산의 구현

- 새로운 데이터를 임의의 위치에 삽입
- 항목의 위치를 노드 포인터로 변환해주는 함수 `get_node_at` 필요

```
// 리스트안에서 pos 위치의 노드를 반환한다.
ListNode *get_node_at(ListType *list, int pos)
{
    int i;
    ListNode *tmp_node = list->head;
    if( pos < 0 ) return NULL;
    for (i=0; i<pos; i++)
        tmp_node = tmp_node->link;
    return tmp_node;
}
// 주어진 위치에 데이터를 삽입한다.
void add(ListType *list, int position, element data)
{
    ListNode *p;
    if ((position >= 0) && (position <= list->length)){
        ListNode*node = (ListNode *)malloc(sizeof(ListNode));
        if( node == NULL ) error("메모리 할당에러");
        node->data = data;
        p = get_node_at(list, position-1);
        insert_node(&(list->head), p, node);
        list->length++;
    }
}
```

delete 연산의 구현

- 임의의 위치의 데이터를 삭제
- 항목의 위치를 노드 포인터로 변환해주는 함수 `get_node_at` 필요

```
// 주어진 위치의 데이터를 삭제한다.  
void delete(ListType *list, int pos)  
{  
    if (!is_empty(list) && (pos >= 0) && (pos < list->length)){  
        ListNode *p = get_node_at(list, pos-1);  
        remove_node(&(list->head), p, (p!=NULL)?p->link:NULL);  
        list->length--;  
    }  
}
```