CHAP 4리스트

C로 쉽게 풀어쓴 자료구조

생능출판사

2011

<<table of contents>>

<<table of contents>>

리스트란

- 리스트(list), 선형 리스트(linear list): 순서를 가진 항목들의 모임
  - 집합: 항목간의 순서의 개념이 없음

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

- 리스트의 예
  - 요일: (일요일, 월요일, …, 토요일)
  - 한글 자음의 모임: (ㄱ,ㄴ,…,ㅎ)
  - 카드: (Ace, 2,3,…,King)
  - 핸드폰의 문자 메시지 리스트

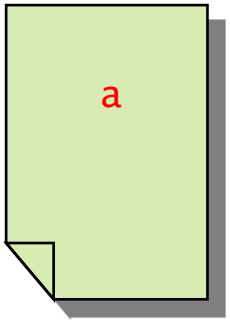
리스트의 연산

- 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.
- 리스트 안의 항목의 개수를 센다.
- 리스트가 비었는지, 꽉 찼는지를 체크한다.
- 리스트 안의 모든 항목을 표시한다.

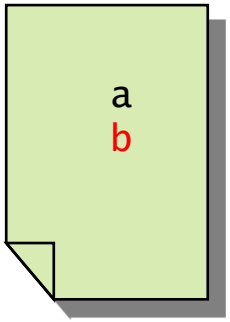
리스트 ADT

- 객체: n개의 element형으로 구성된 순서있는 모임
- 연산:
  - add_last(list, item) ::= 맨끝에 요소를 추가한다.
  - add_first(list, item) ::= 맨앞에 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 item로 바꾼다.
  - is_in_list(list, item) ::= item이 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.

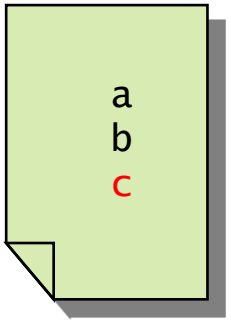
리스트 ADT 사용예 1



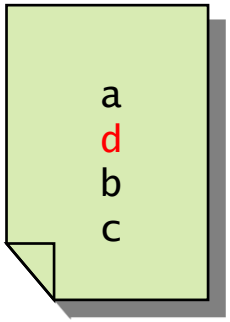
add_last(list1,a)



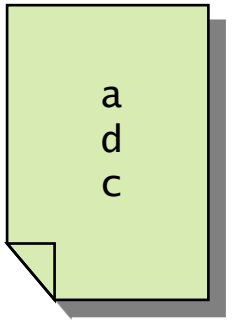
add_last(list1,b)



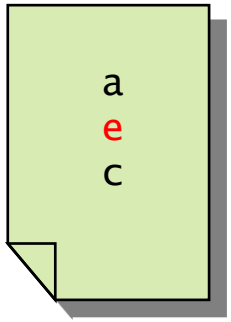
add(list1,2,c)



add(list1,1,d)



delete(list1,2)



replace(list1,1,e)

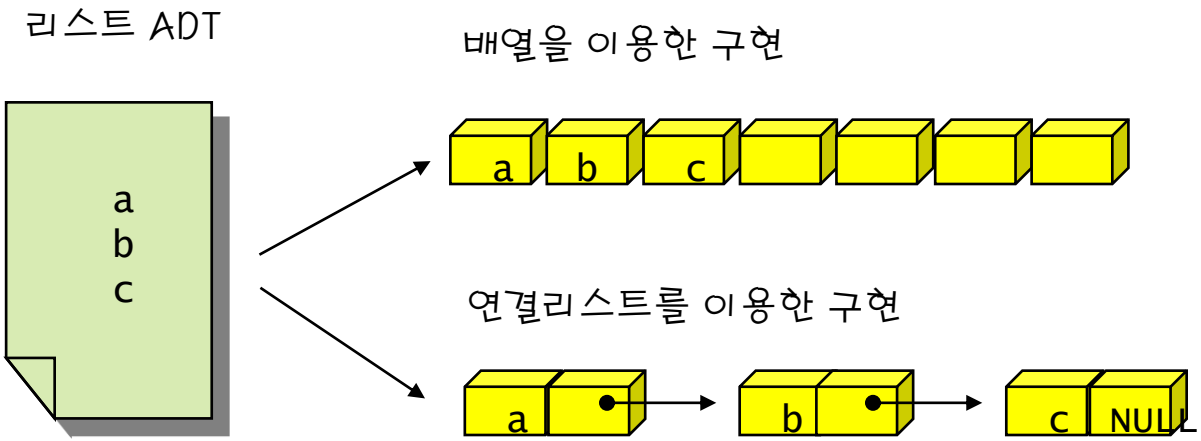
리스트 ADT 사용예 2

```
main()
{
        int i, n;

        // list2를 생성한다: 구현방법에 따라 약간씩 다름
        ListType list2;
        add_last(&list2,"마요네즈");    // 리스트의 포인트를 전달
        add_last(&list2,"빵");
        add_last(&list2,"치즈");
        add_last(&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));
}
```

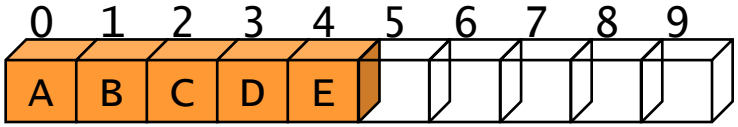
리스트 구현 방법

- 배열을 이용하는 방법
  - 구현이 간단
  - 삽입, 삭제시 오버헤드
  - 항목의 개수 제한
- 연결리스트를 이용하는 방법
  - 구현이 복잡
  - 삽입, 삭제가 효율적
  - 크기가 제한되지 않음

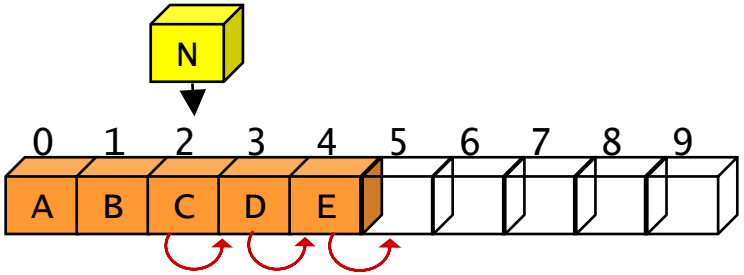


배열로 구현된 리스트

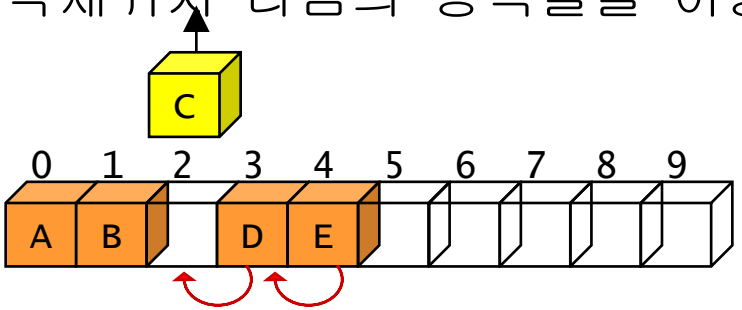
- 1차원 배열에 항목들을 순서대로 저장
  - L=(A, B, C, D, E)



- 삽입연산: 삽입위치 다음의 항목들을 이동하여야 함.



- 삭제연산: 삭제위치 다음의 항목들을 이동하여야 함



- 가상실습 4.1 : 배열로 구현된 리스트 실습

ArrayListType의 구현

- 항목들의 타입은 element로 정의
- list라는 1차원 배열에 항목들을 차례대로 저장
- length에 항목의 갯수 저장

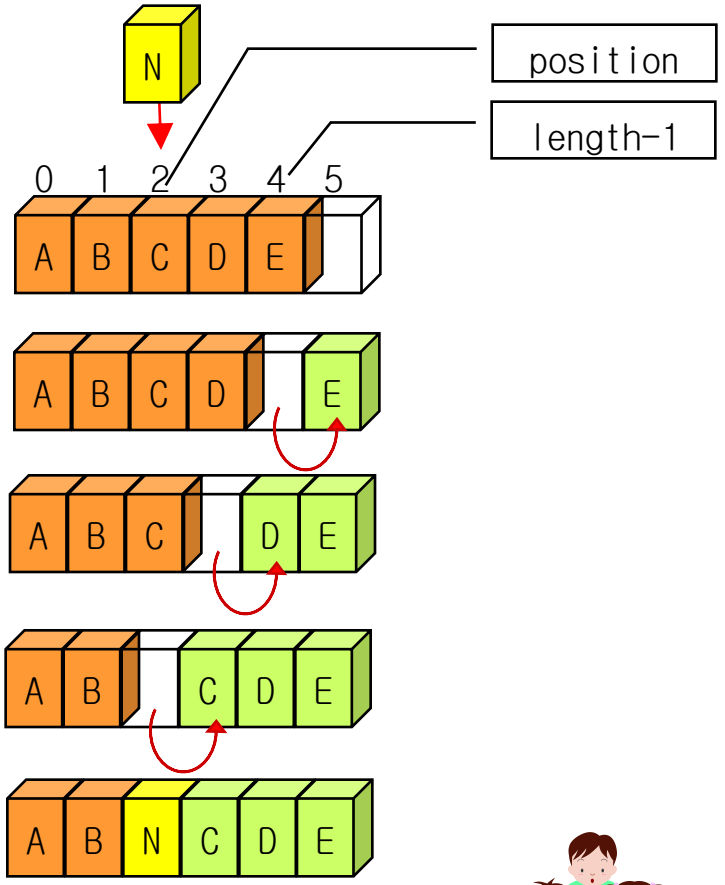
```
typedef int element;
typedef  struct {
            int list[MAX_LIST_SIZE];        // 배열 정의
            int length;                     // 현재 배열에 저장된 항목들의 개수
} ArrayListType;
```

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

ArrayListType의 삽입 연산

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

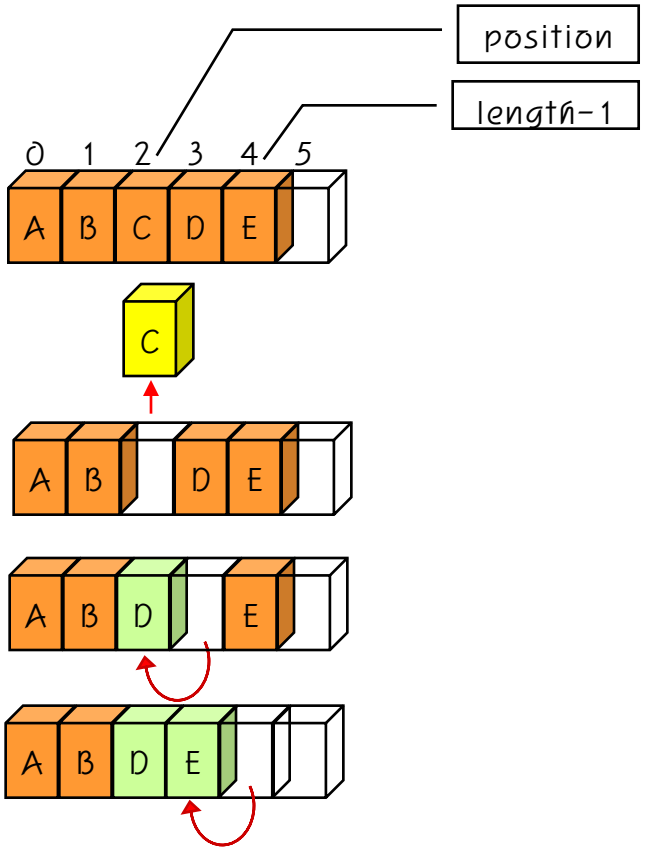
```
// position: 삽입하고자 하는 위치
// item: 삽입하고자 하는 자료
void add(ArrayListType *L, int position, element item)
{
  if( !is_full(L) && (position >= 0) &&
              (position <= L->length) ){
    int i;
    for(i=(L->length-1); i>=position;i--)
              L->list[i+1] = L->list[i];
    L->list[position]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



ArrayListType의 삭제 연산

1. 삭제 위치를 검사한다.
2. 삭제위치부터 맨끝까지의 자료를 한칸씩 앞으로 옮긴다.

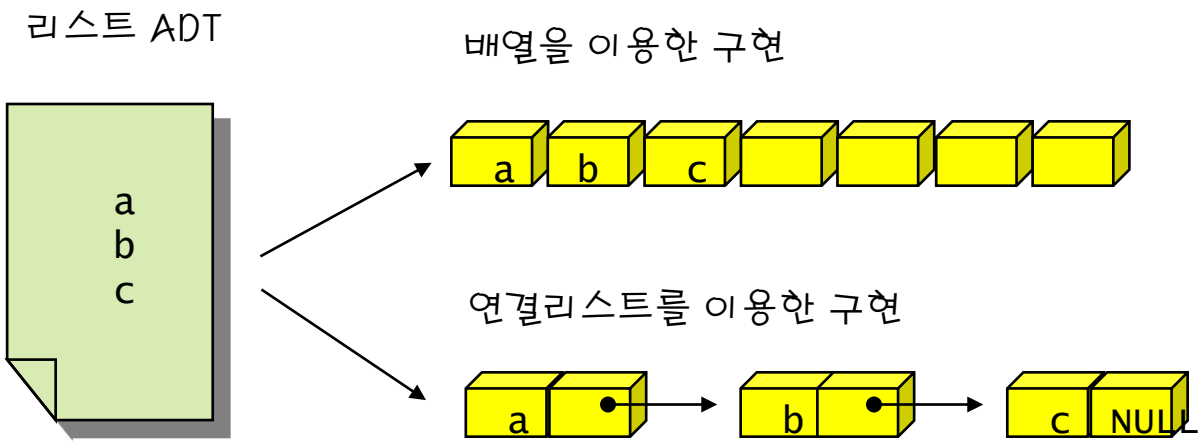
```
// position: 삭제하고자 하는 위치
// 반환값: 삭제되는 자료
element delete(ArrayListType *L, int position)
{
        int i;
        element item;
        if( position < 0 || position >= L->length )
                error("위치 오류");
        item = L->list[position];
        for(i=position; i<(L->length-1);i++)
                L->list[i] = L->list[i+1];
        L->length--;
        return item;
}
```



- 프로그램 4.4 : 배열을 이용한 리스트 테스트 프로그램

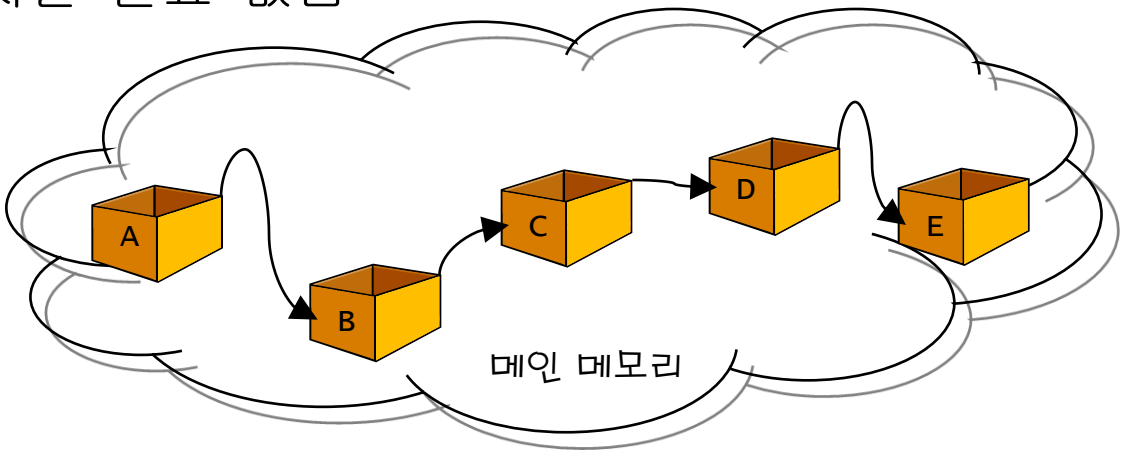
연결 리스트

- 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.



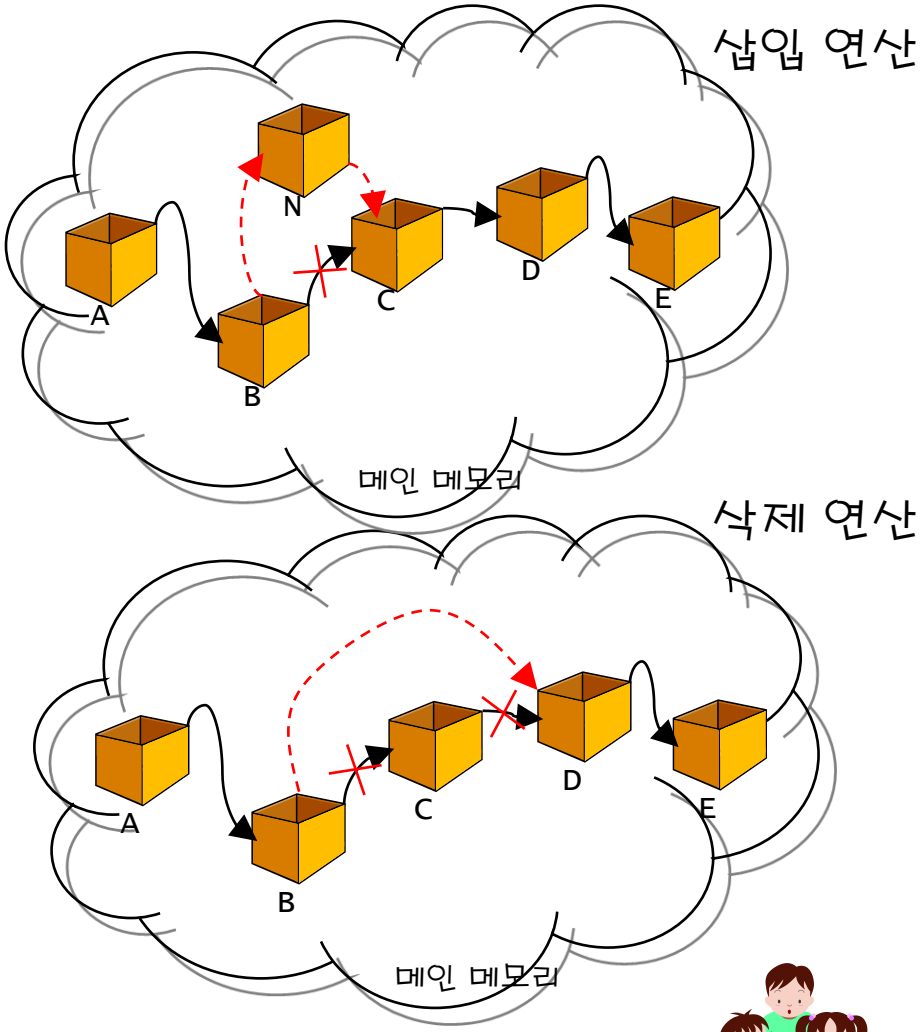
연결된 표현

- 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)
- 메모리 안에서의 노드의 물리적 순서가 리스트의 논리적 순서와 일치할 필요 없음



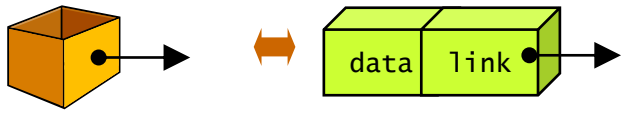
연결된 표현의 장단점

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

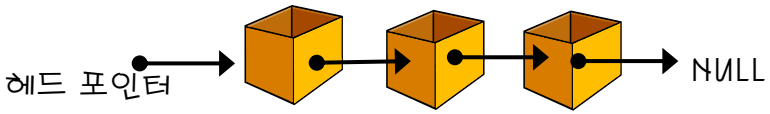


연결 리스트의 구조

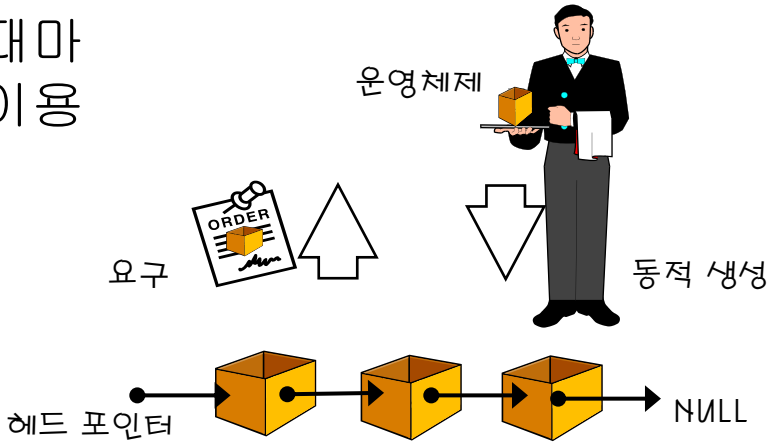
- 노드 = 데이터 필드 + 링크 필드



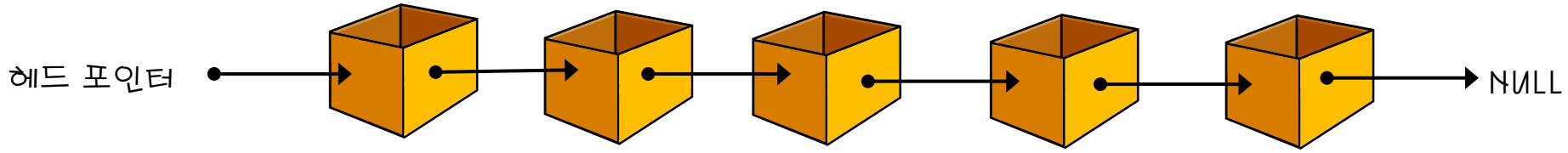
- 헤드 포인터(head pointer): 리스트의 첫번째 노드를 가리키는 변수



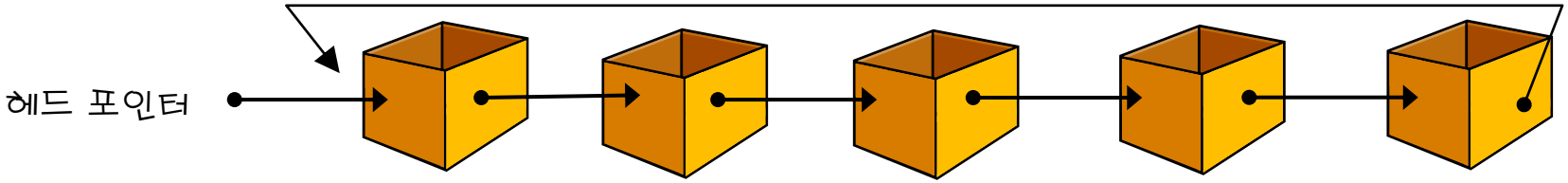
- 노드의 생성: 필요할 때마다 동적 메모리 생성 이용하여 노드를 생성



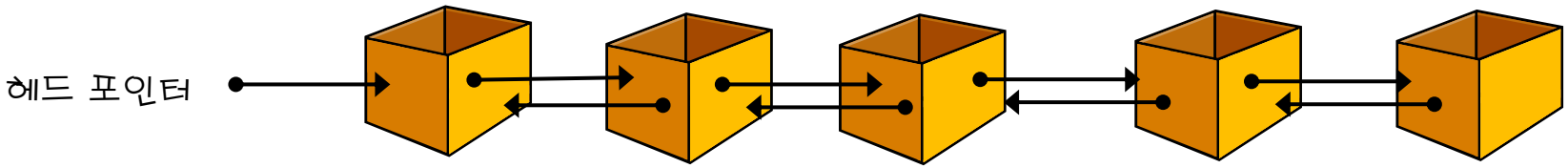
연결 리스트의 종류



단순 연결 리스트



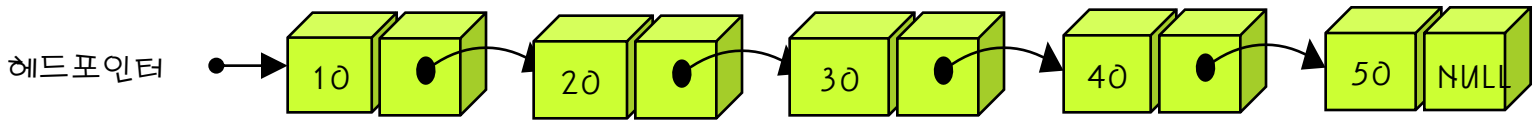
원형 연결 리스트



이중 연결 리스트

단순 연결 리스트

- 하나의 링크 필드를 이용하여 연결
- 마지막 노드의 링크값은 NULL



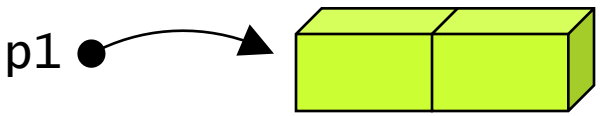
단순 연결 리스트의 구현

- 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 함수 이용

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



삽입연산의 코드

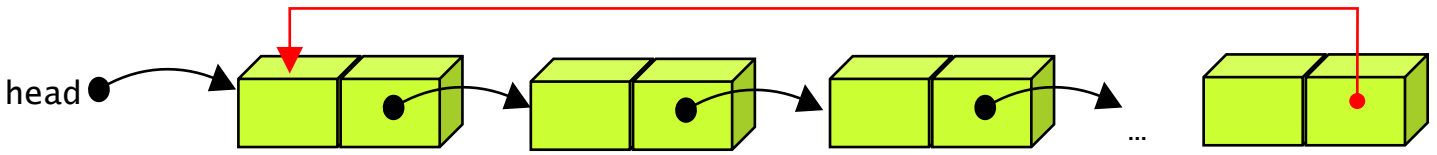
```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드
// new_node : 삽입될 노드
void insert_node(ListNode **phead, ListNode *p, ListNode *new_node)
{
        if( *phead == NULL ){          // 공백리스트인 경우
                new_node->link = NULL;
                *phead = new_node;
        }
        else if( p == NULL ){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

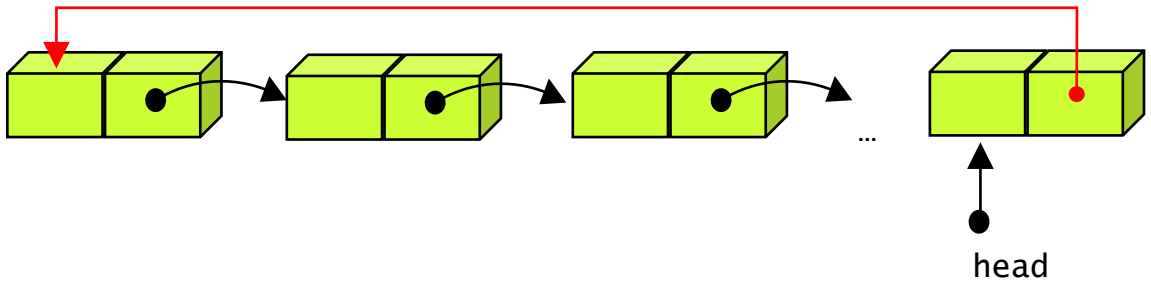
```
// phead : 헤드 포인터에 대한 포인터
// p: 삭제될 노드의 선행 노드
// removed: 삭제될 노드
void remove_node(ListNode **phead, ListNode *p, ListNode *removed)
{
        if( p == NULL )
                *phead = (*phead)->link;
        else
                p->link = removed->link;
        free(removed);
}
```

원형 연결 리스트

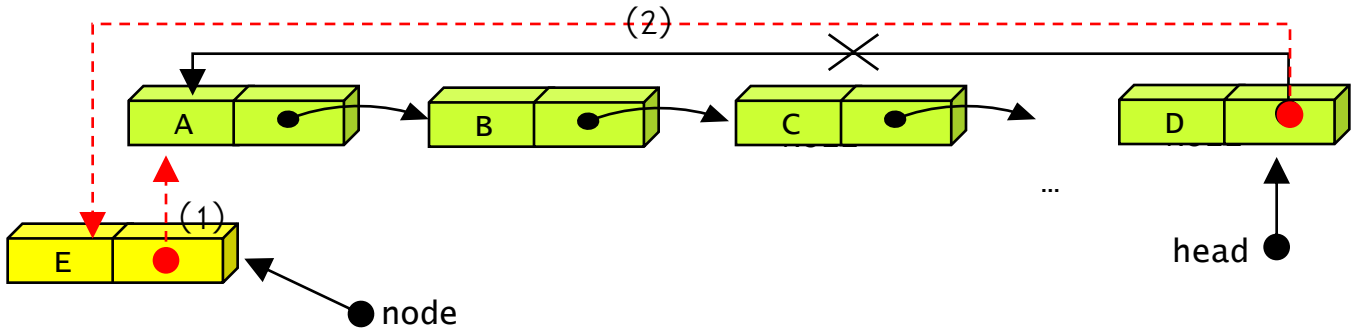
- 마지막 노드의 링크가 첫번째 노드를 가리키는 리스트
- 한 노드에서 다른 모든 노드로의 접근이 가능



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

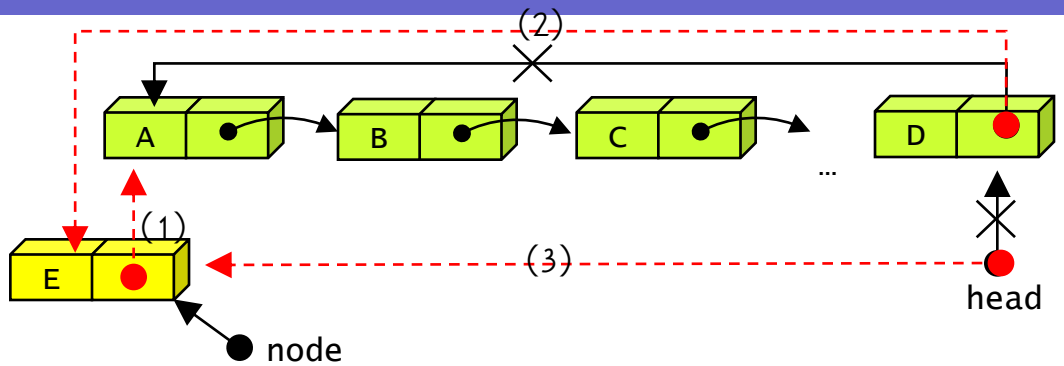


리스트의 처음에 삽입




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

리스트의 끝에 삽입

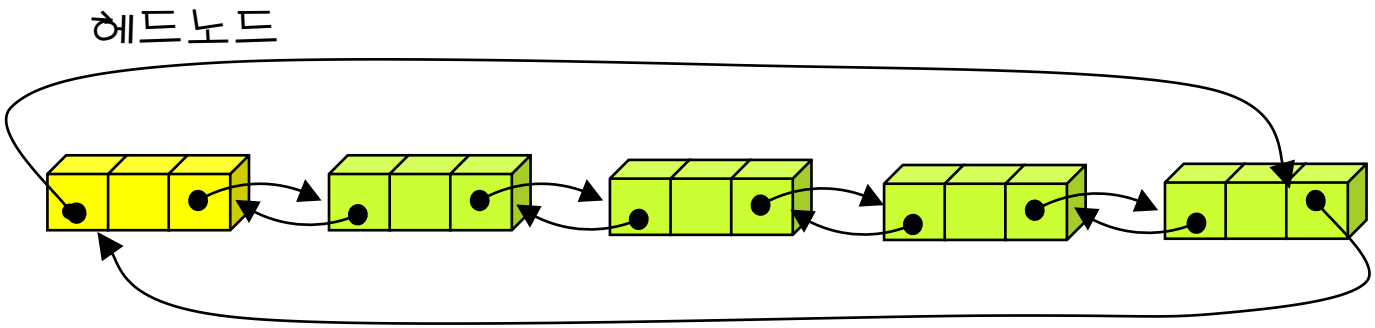



```
// phead: 리스트의 헤드 포인터의 포인터
// p : 선행 노드
// node : 삽입될 노드
void insert_last(ListNode **phead, ListNode *node)
{
        if( *phead == NULL ){
                *phead = node;
                node->link = node;
        }
        else {
                node->link = (*phead)->link;
                (*phead)->link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

- 프로그램 4.15 : 원형 연결 리스트 테스트 프로그램

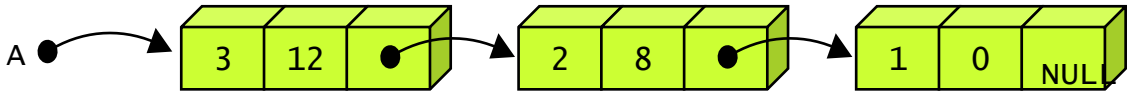
이중 연결 리스트

- 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



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;
```

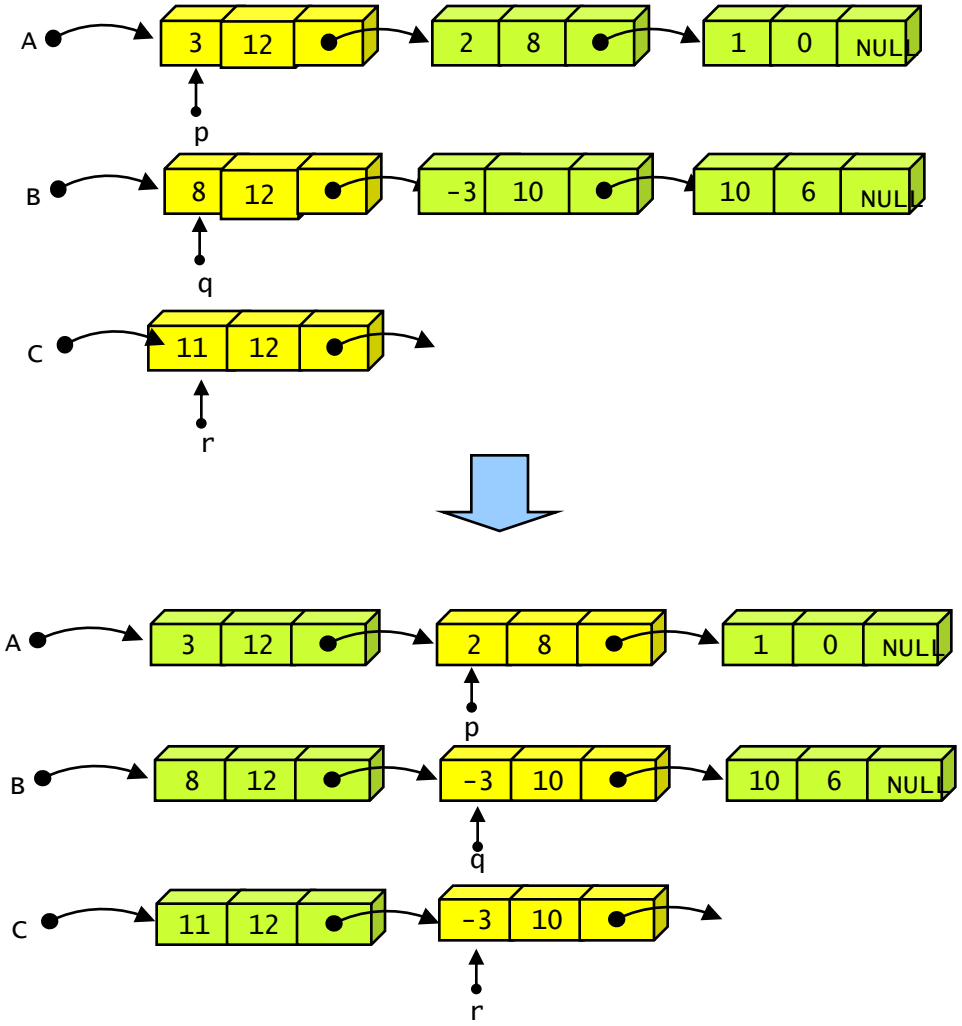
다항식의 덧셈 구현

- 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.

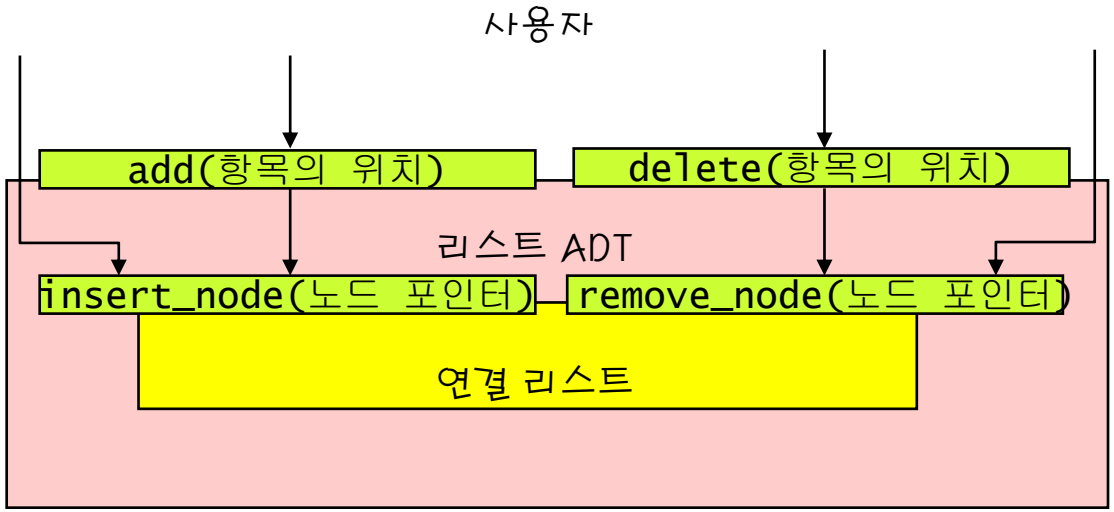
① p.expon == q.expon :
두 계수를 더해서 0이 아니면 새로운 항을 만들어 결과 다항식 c에 추가한다. 그리고 p와 q는 모두 다음 항으로 이동한다.
② p.expon < q.expon :
q가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 c에 추가한다. 그리고 q만 다음 항으로 이동한다.
③ p.expon > q.expon :
p가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 c에 추가한다. 그리고 p만 다음 항으로 이동한다.

다항식의 덧셈



연결리스트를 이용한 리스트 ADT의 구현

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



리스트 ADT의 구현

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

LinkedListType list1;
```

첫 번째 노드를 가리키는 헤드 포인터

연결 리스트내의 존재하는 노드의 개수

리스트 ADT의 생성

is_empty, get_length 연산의 구현

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

```
// 리스트의 항목의 개수를 반환한다.
int get_length(LinkedListType *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(LinkedListType *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;
}
```

delete 연산의 구현

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

```
// 주어진 위치의 데이터를 삭제한다.
void delete(LinkedListType *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--;
        }
}
```

get_entry, clear 연산의 구현

```
// 주어진 위치 pos 에 있는 데이터를 반환한다
element get_entry(LinkedListType *list, int pos)
{
        ListNode *p;
        if( pos >= list->length ) error("위치 오류");
        p = get_node_at(list, pos);
        return p->data;
}

// 모든 노드를 지운다.
void clear(LinkedListType *list)
{
        int i;
        for (i = 0; i < list->length; i++)
                delete(list, i);
}
```

전체 프로그램

```
//
int main()
{
        LinkedListType list1;
        init(&list1);
        add(&list1, 0, 20);
        add_last(&list1, 30);
        add_first(&list1, 10);
        add_last(&list1, 40);
        // list1 = (10, 20, 30, 40)
        display(&list1);
```