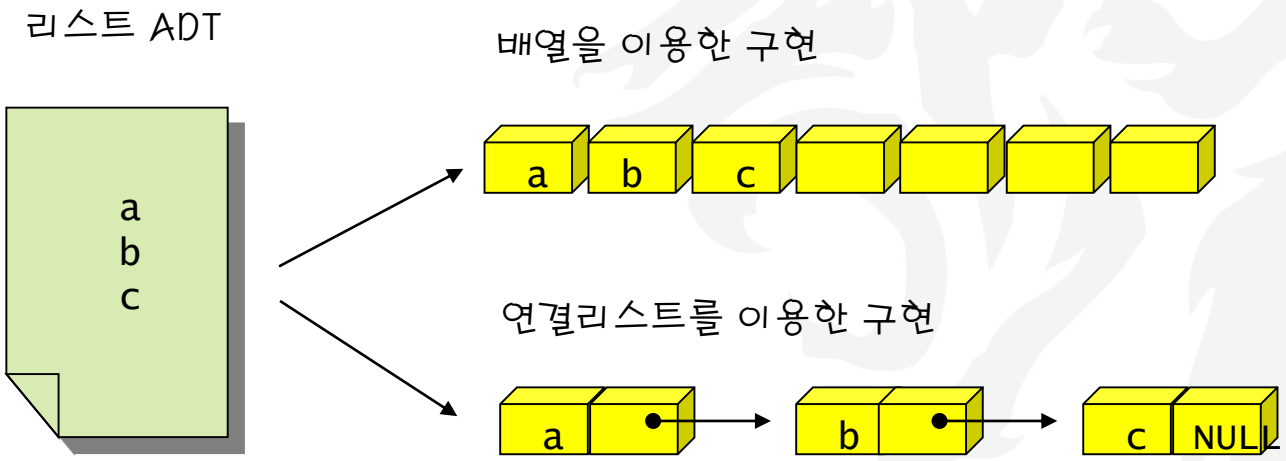
CSE2010 자료구조론

Week 3: Linked List 1

ICT융합학부 한진영

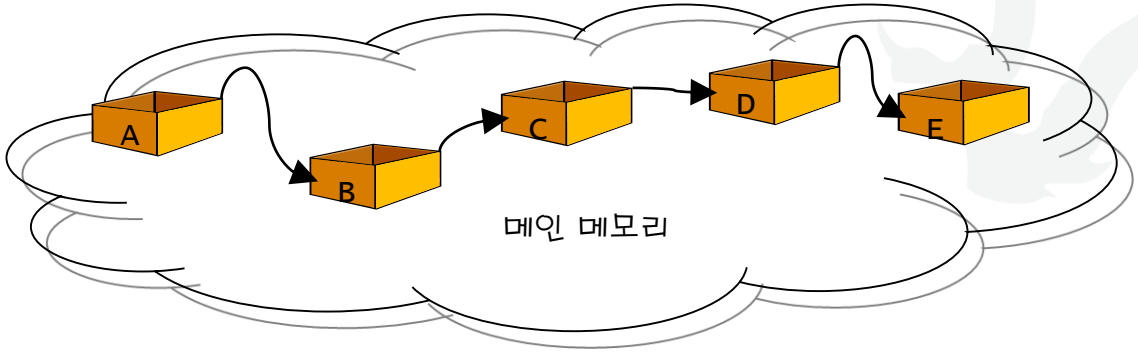
연결 리스트(Linked List)

- 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 연결



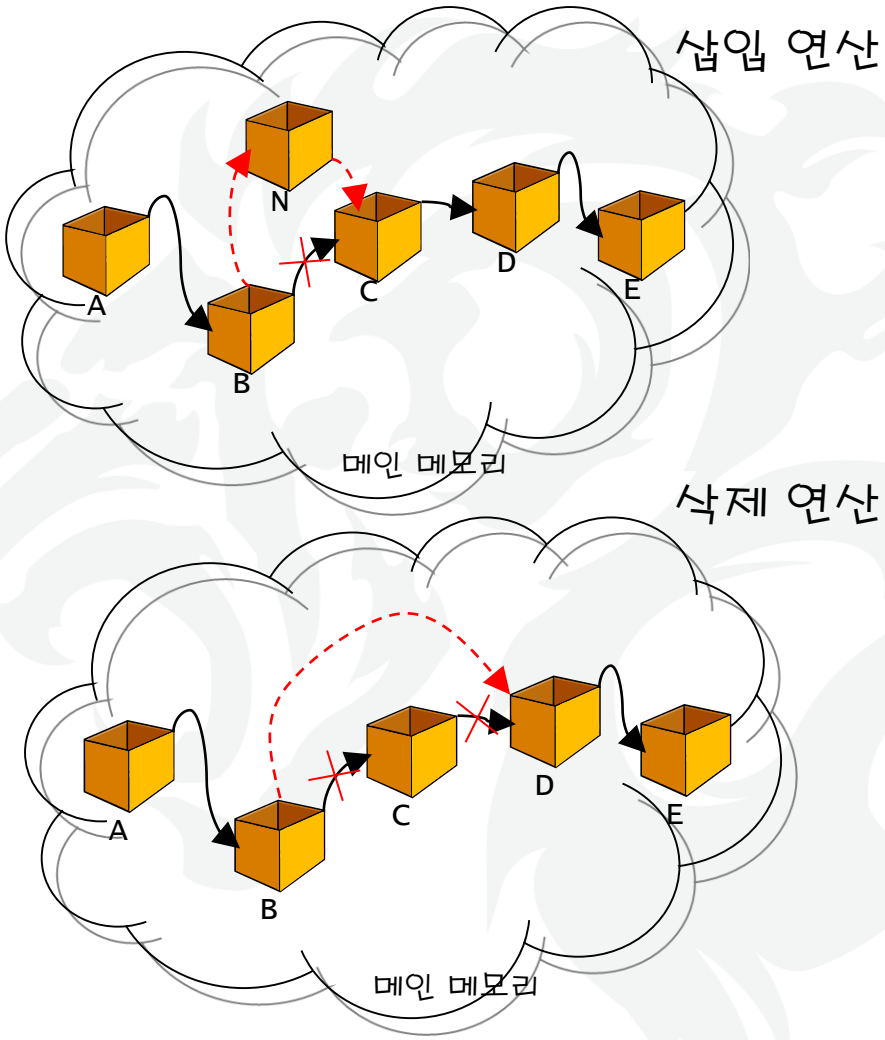
연결된 표현(Linked Representation)

- 연결된 표현
  - 데이터와 링크로 구성됨
  - 다양한 동적구조(리스트, 스택, 큐, 트리, 그래프 등)에서 활용됨
  - 리스트의 항목들을 노드(node)라고 하는 곳에 분산하여 저장
  - 다음 항목을 가리키는 주소도 같이 저장
- 노드(node) : <항목, 주소> 쌍
  - 노드는 데이타 필드와 링크 필드로 구성
  - 데이타 필드: 리스트의 원소, 즉 데이타 값을 저장하는 곳
  - 링크 필드: 다른 노드의 주소값을 저장하는 장소(포인터)
- 메모리안에서의 노드의 물리적 순서가 리스트의 논리적 순서와 일치할 필요 없음



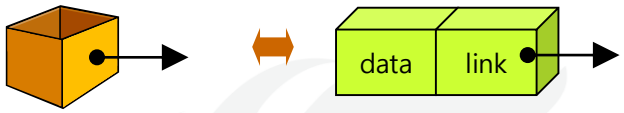
연결된 표현의 장단점

- 장점
  - 삽입, 삭제가 용이
  - 연속된 메모리 공간이 필요 없음
  - 크기 제한이 없음
- 단점
  - 구현이 상대적으로 어려움
  - 오류가 발생하기 쉬움

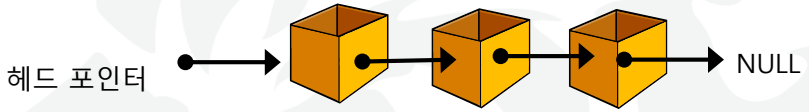


연결 리스트의 구조

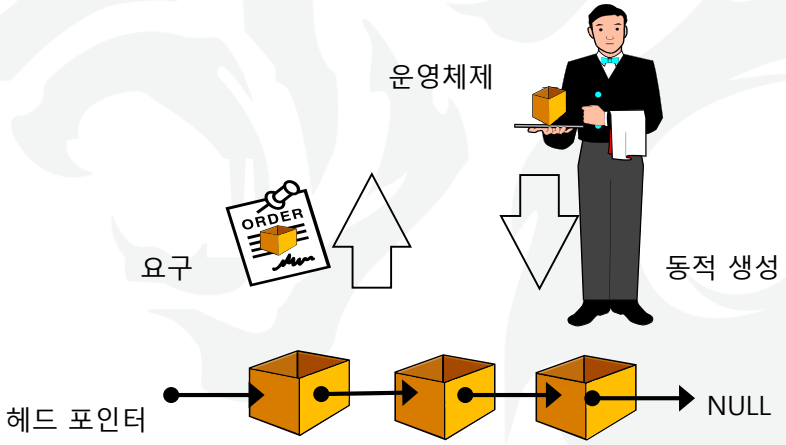
- 노드 = 데이터 필드 + 링크 필드



- 헤드 포인터(head pointer)
  - 리스트의 첫 번째 노드를 가리키는 변수

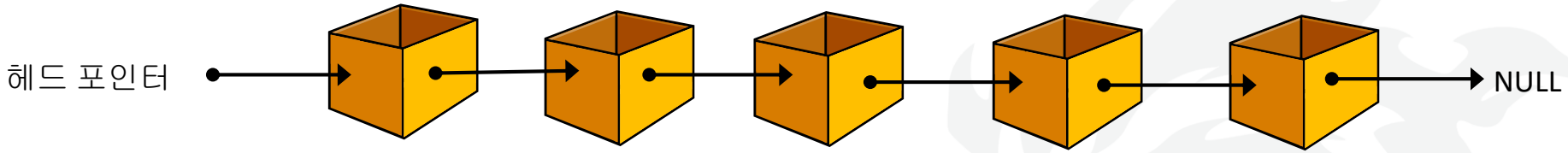


- 노드의 생성
  - 필요시 동적으로 메모리 생성함



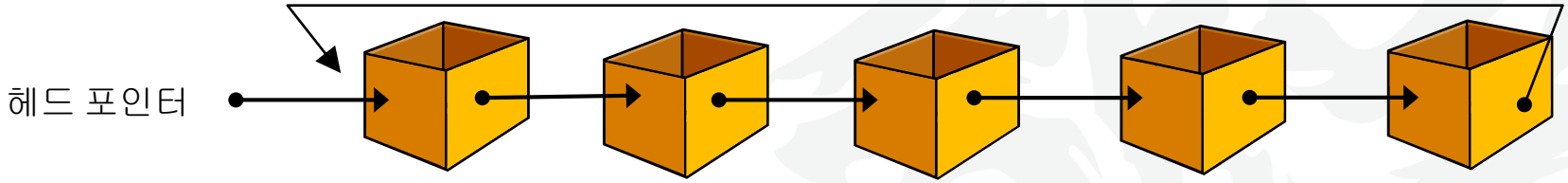
연결 리스트의 종류

- 단순 연결 리스트



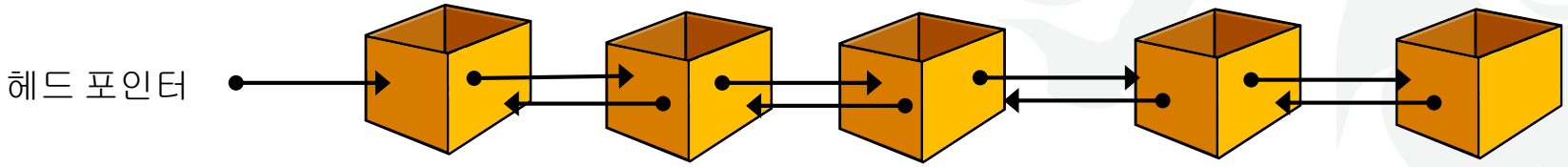
단순 연결 리스트(singly linked list)

- 원형 연결 리스트



원형 연결 리스트(circular linked list)

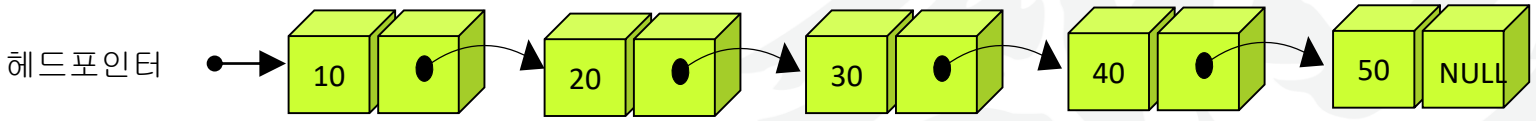
- 이중 연결 리스트



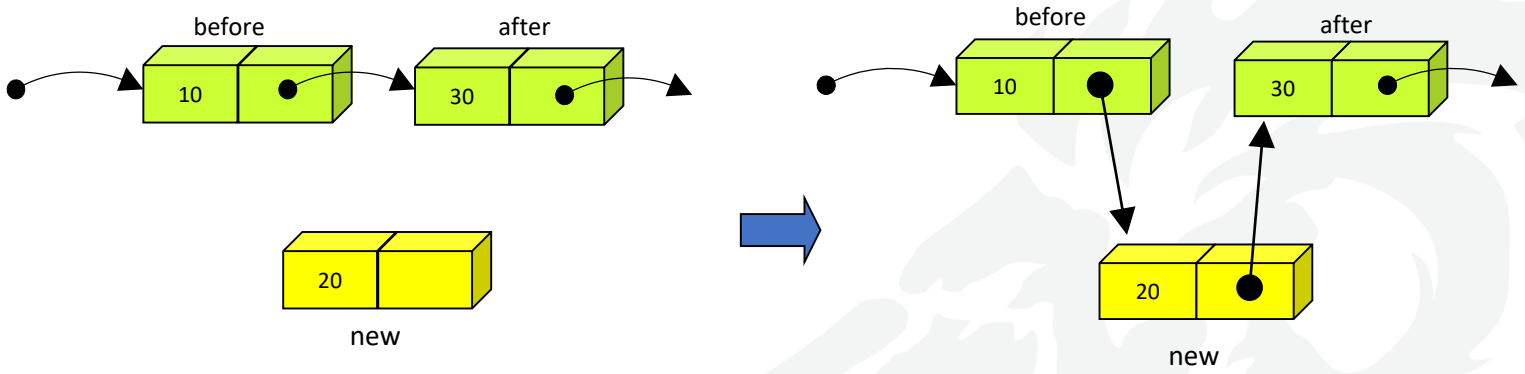
이중 연결 리스트(doubly linked list)

단순 연결 리스트

- 하나의 링크 필드를 이용하여 연결
- 마지막 노드의 링크값은 'NULL'



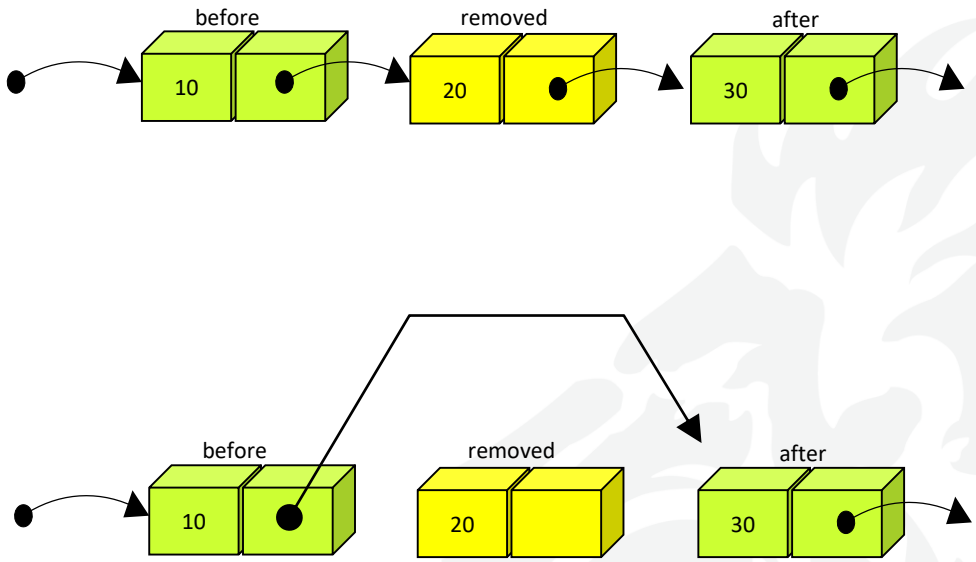
단순 연결 리스트: 삽입 연산




```
insert_node(L, before, new)

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

단순 연결 리스트: 삭제 연산




```
remove_node(L, before, removed)

if L ≠ NULL
    then   before.link←removed.link
      destroy(removed)
```

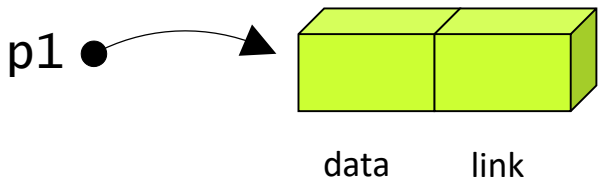
단순 연결 리스트 구현: 노드 생성

- 노드: 구조체로 정의
  - 데이터 필드 + 링크 필드(포인터 사용)

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

- 노드의 생성
  - 동적 메모리 할당

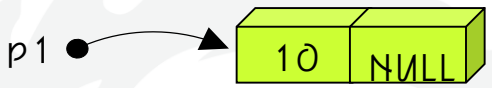
```
ListNode *p1;
p1 = (ListNode *)malloc(sizeof(ListNode));
```



단순 연결 리스트 구현: 노드 연결

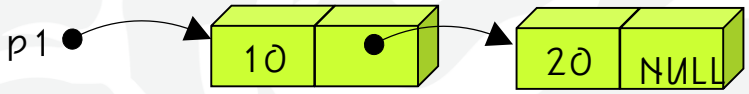
- 데이터 필드와 링크 필드 설정

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



- 두번째 노드 생성 및 첫번째 노드와의 연결

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



- 헤드 포인터(head pointer)
  - 연결 리스트의 맨 첫번째 노드를 카리키는 포인터

단순 연결 리스트 구현: 삽입 연산(1)

- 삽입 함수의 프로토타입

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

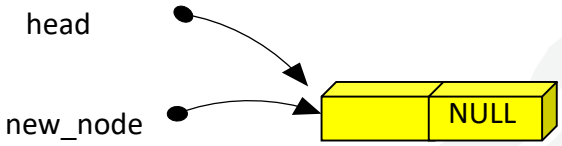
phead: 헤드 포인터 head에 대한 포인터
p: **삽입될 위치의 선행 노드**를 가리키는 포인터, 이 노드 다음에 삽입된다.
new_node: **새로운 노드**를 가리키는 포인터

* 헤드포인터가 함수 안에서 변경되므로 헤드포인터의 포인터 필요

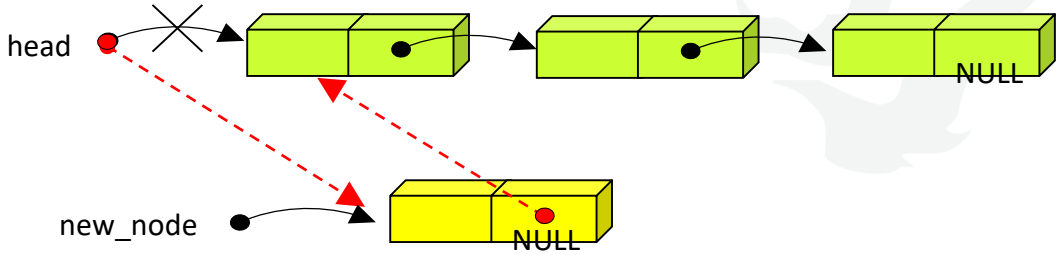
- 삽입의 3가지 경우
  - (1) head가 NULL인 경우: 공백 리스트에 삽입
  - (2) p가 NULL인 경우: 리스트의 맨처음에 삽입
  - (3) 일반적인 경우: 리스트의 중간에 삽입

단순 연결 리스트 구현: 삽입 연산(2)

- (1) head가 NULL인 경우
  - head가 NULL이라면 현재 삽입하려는 노드가 첫 번째 노드가 됨
  - 따라서 head의 값만 변경

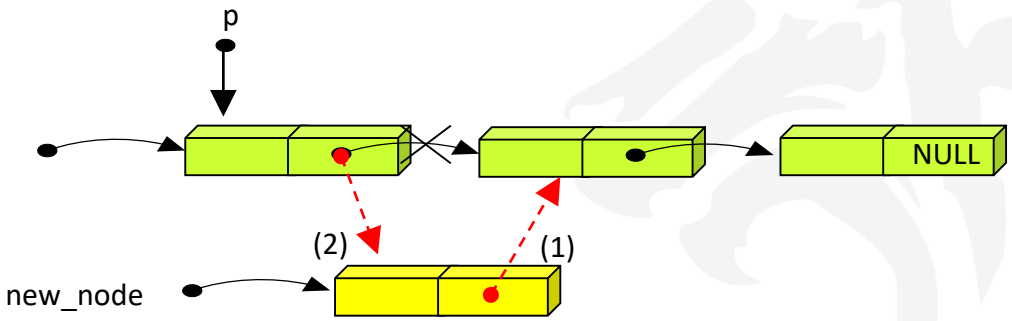


- (2) p가 NULL인 경우
  - 새로운 노드를 리스트의 맨 앞에 삽입



단순 연결 리스트 구현: 삽입 연산(3)

- (3) head와 p가 NULL이 아닌 경우
  - 가장 일반적인 경우
  - new_node의 link에 p->link값을 복사한 다음에 p->link가 new_node를 가리킴



단순 연결 리스트 구현: 삽입 연산 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;
        }
}
```

단순 연결 리스트 구현: 삭제 연산(1)

- 삭제 함수의 프로토타입

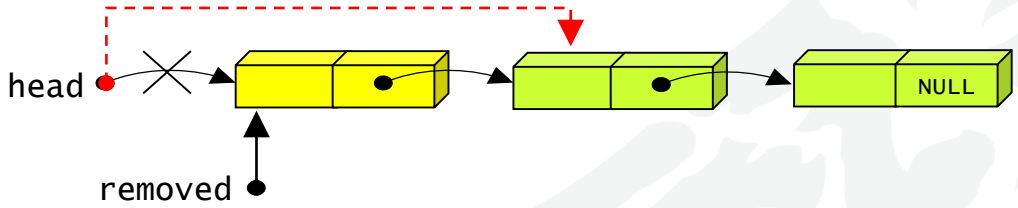
```
//phead: 헤드 포인터 head의 포인터
//p: 삭제될 노드의 선행 노드를 가리키는 포인터
//removed: 삭제될 노드를 가리키는 포인터

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

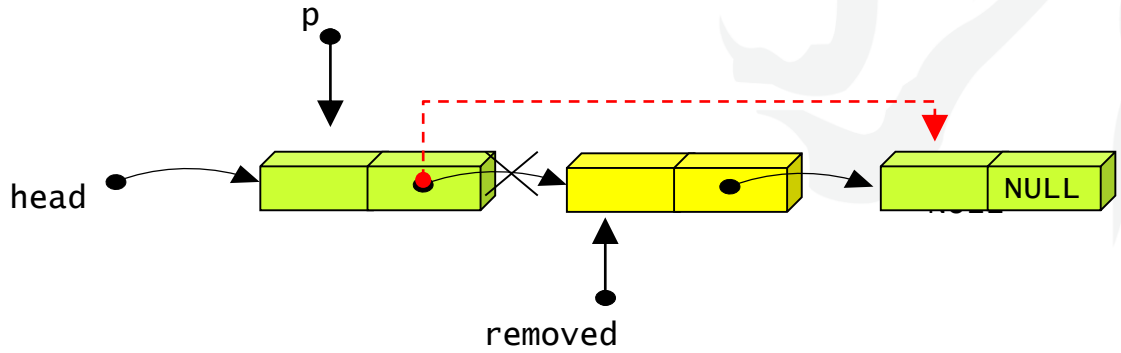
- 삭제의 3가지 경우
  - (1) p가 NULL인 경우: 맨 앞의 노드를 삭제
  - (2) p가 NULL이 아닌 경우: 중간 노드를 삭제

단순 연결 리스트 구현: 삭제 연산(2)

- (1) p가 NULL인 경우
  - 연결 리스트의 첫 번째 노드를 삭제함
  - 헤드포인터 변경



- (2) p가 NULL이 아닌 경우
  - removed 앞의 노드인 p의 링크가 removed 다음 노드를 가리킴



단순 연결 리스트 구현: 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);
}
```

단순 연결 리스트 구현: 방문 연산 코드

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

[반복]

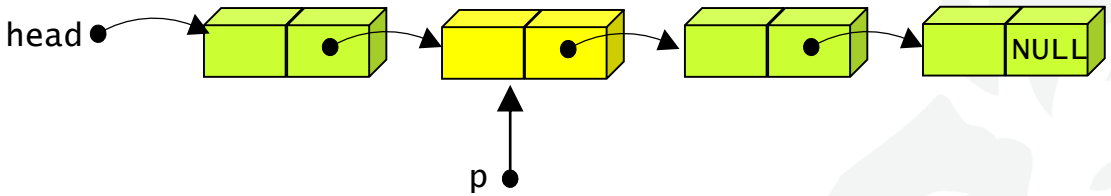
```
void display(ListNode *head)
{
     ListNode *p=head;
     while( p != NULL ){
          printf("%d->", p->data);
          p = p->link;
     }
     printf("\n");
}
```

[순환]

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

단순 연결 리스트 구현: 탐색 연산 코드

- 탐색 연산
  - 특정한 데이터 값을 갖는 노드를 찾는 연산

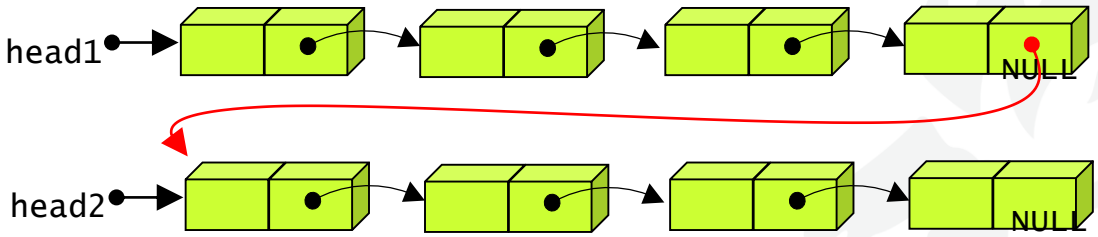



```
ListNode *search(ListNode *head, int x)
{
        ListNode *p;
        p = head;
        while( p != NULL ){
                if( p->data == x ) return p;  // 탐색 성공
                p = p->link;
        }
        return p;  // 탐색 실패일 경우 NULL 반환
}
```

* 포인터 p가 첫번째 노드를 가리키도록 하고, 순서대로 링크를 따라가면서 노드 데이터와 비교

단순 연결 리스트 구현: 합병 연산 코드

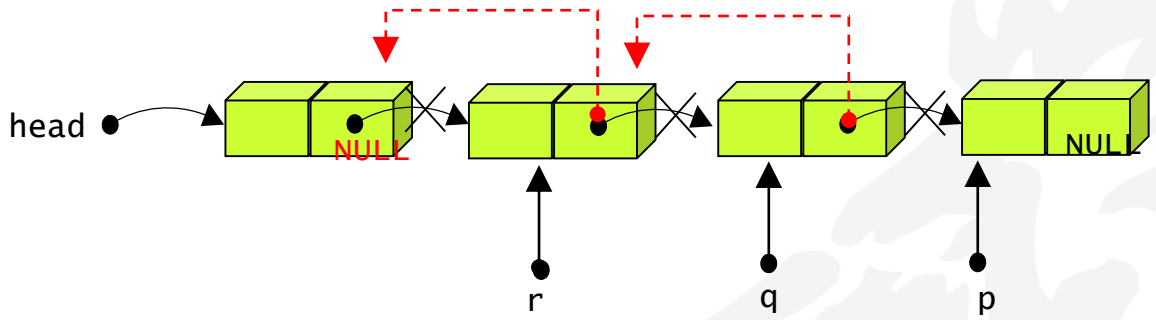
- 합병 연산
  - 2개의 리스트를 합하는 연산




```
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;
     }
}
```

단순 연결 리스트 구현: 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 아직 처리되지 않은 노드
  q = NULL;       // q는 역순으로 만들 노드
  while (p != NULL){
     r = q;  // r은 역순으로 된 노드.   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
}
```

Week 3: Linked List 1