<6장, 7장 연결 리스트(LINKED LISTS), 김성현, 2019107783>

6.1 리스트 추상 데이터 타입(ADT)	6.2 리스트 구현 방법(배열과 연결된 리스트)	6.3 연결 리스트	6.4 단순 연결 리스트 - 구현
		6.6 연결 리스트 구현: 다항식	6.5 단순 연결 리스트의 연산 구현

7.1 원형 연결 리스트	7.2 원형 연결 리스트는 어디에 사용될까?	7.3 이중 연결 리스트	7.4 에제: mp3 재생 프로그램 만들기
		7.6 연결 리스트로 구현한 큐	7.5 연결 리스트로 구현한 스택

6.1. 리스트 추상 데이터 타입(ADT) → 전형적으로 필요한 것들(추가되거나 삭제되는 연산들)
└ 리스트: 여러 element가 순서가 있다!
└ 실제 현장에서 프로젝트를 짤 때 이 ADT를 많이 구현한다.

(p.176 ~ p.177) 리스트(List) ⇒ 우리들이 자료를 정리하는 방법 중 하나.
- → 항목들이 차례대로 저장, 리스트의 항목들은 순서 또는 위치를 가진다.
- → 앞서 살펴본 스택과 큐도 넓게 보면 리스트의 일종
- → 리스트는 집합과는 다르다, 집합은 각 항목 간의 순서의 개념이 없지만, 리스트는 있다!!

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

• 리스트를 가지고 할 수 있는 연산.
- 리스트에 새로운 항목을 추가한다 → <추가연산>
- 리스트에서 항목을 삭제한다. → <삭제연산>
- 리스트에서 특정한 항목을 찾는다. → <탐색 연산>.

• 리스트 에서
- 특정 위치로 이동
- 현재 항목 등등

p.177

• 리스트 ADT

순서가 살고, 그 순서가 유지, 순서가 중요!!! (정렬과는 무관하다, 다르다, 정렬은 아니다.)

• 객체 : list는 순서(order)가 있는 요소(element)들의 모임.

• 연산 :

position 정보 O / position 정보 X

- insert(list, pos, item) ::= list의 pos 위치에 item을 추가. (└ 위치정보 / 순서(position, array index 아니고, 순서 위치))
- insert_last(list, item) ::= list의 끝 위치에 item을 추가. (└ 그냥 끝에, 맨 마지막)
- insert_first(list, item) ::= list의 시작 위치에 item을 추가. (└ 맨 처음에)

(element 개수 증가 (+1))

- delete(list, pos) ::= list의 pos 위치의 요소를 삭제 (element 개수 감소 (-1) or all)
- clear(list) ::= list의 모든 요소를 삭제 (전체 삭제) → element가 0개가 됨 ⇒ "delete보다는 clear가 실용 [illegible] 더 적다."
- get_entry(list, pos) ::= list의 pos 위치의 요소를 반환. (└ 정보만 알려주는 것 (pos 위치))
- get_length(list) ::= list의 요소의 개수를 반환. (└ list의 길이라고 할 수도 있다.)
- is_empty(list) ::= list가 empty인지 검사.
- is_full(list) ::= list가 full인지 검사. → array를 사용해서 list를 구현하면 full인 경우가 문제가 발생하지만, pointer를 사용해서 구현한 경우 full인 경우는 거의 없다고 할 수 있다.
- print_list(list) ::= list의 모든 요소를 출력

- 앞에서 배운 [illegible] 다르게 삭제, 추가 위치가 정해져 있지 않다 / 원하는 원소를 넣거나 뺄 수 있는 것이 가능하다.

이거 확실 ⇒ 꼭 하는 출력하는 '방문' 하는 함수라고 이해하는게 더 좋다

→ list의 모든 요소들을 다 '방문'해야 하기 때문에 (traversal) 그래야 출력할 수 있기 때문에 이게 중요하다.

"insert, delete는 실패할 요지가 다소 있다."

"꼭 그렇게나 [illegible] 불가능하게 다 성공적으로 끝나는 것은 아니다."

"프로그램의 성공과 실패를 다 생각할 수 있게 하고 처리 가능하게 바로 '전문가'이다."

성공과 실패를 항상 같이 공부해야 한다!!

p.177~185 6.2 리스트 구현 방법

p.177~178. ① ○ 배열(array)을 사용한 구현. → (느리다☆)

- 장점
  - ⊙ 구현은 간단 (+속도 빠름) + 오류가 발생할 확률↓
- 단점
  - ⊙ insert, delete 동작이 비효율적임 (항목들의 이동이 많음)
    - ↳ 다 이동, 밀려서 최대 열어야되서 쓰다 이동 하니 이동이 많음)
  - ⊙ 최대 요소의 개수 제한 있음 (배열의 크기 때문)
    - 컴퓨터의 resource가 있는데도 그걸 못쓴다.

문제의 해결 ☆☆☆
↳ 1. 정확성
2. 효율성
3. 메모리 낭비 + resource 적게 사용

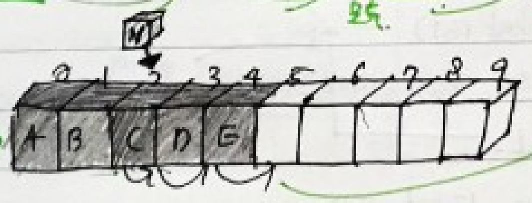
② ○ 연결 리스트 (linked list)를 이용한 구현. → pointer → 또 3가지로 나뉨: 1. simple 2. Circular 3. Doubly.

- ⊙ 구현은 복잡 ← pointer를 써야하기 때문
- ⊙ insert, delete 동작이 매우 효율적. 밀려서 하는게 없음. 굉장히 빠르게 동작이 가능함.
- ⊙ 최대 요소의 개수에 제한 없음. (링크 리스트의 resource가 있는 한)
  - ↳ malloc을 써서 메모리를 할당 받기 때문에 제한은 없기 때문.

p.178~ • 배열(Array)을 이용한 구현.

⊙ 리스트의 모습: 1차원 배열에 요소들을 이어서 저장 (최대 10개 제한)
빈 공간 없이 이어서 저장 (순서가 있기 때문)

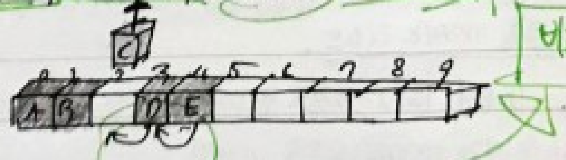
⊙ insert 동작: 삽입할 위치 및 오른쪽 항목들을 모두 한칸씩 이동 → 비효율적
모두.



"어떤 operation이 있으면 그걸 생각할 줄 알고 예측할 수 있어야 하는 것이 매우 중요하다!!"

<리스트의 순차적 표현>
(sequential representation)

⊙ delete 동작: 제거할 위치 오른쪽 항목들을 한칸씩 이동. → 비효율적 ☆



(각 사례 따른 코드 구현은 교재 p.179~183 참고)

p.183. *실행시간 분석.

- 임의의 항목에 접근하는 연산 get-entry 연산 → 인덱스 사용, 바로 접근 가능 = O(1) ☆
- 삽입, 삭제 연산 → 다른 항목들을 이동하는 경우가 많으니 최악의 경우 O(n) ☆
  - (제일 오른쪽 뒤 case)
  - → 리스트의 맨 끝 삽입 시 → O(1) ☆

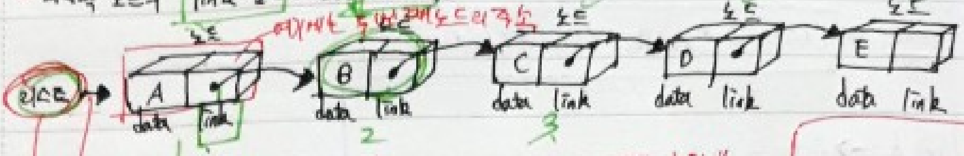
p.187~190 6.4 단순 연결리스트

* 단순(Simple) 연결 리스트

○ 연결 리스트는 리스트의 첫 노드의 주소를 가지고 있음.

○ 각 노드는 다음 노드를 가리키는 하나의 link 필드를 가지고 있음.
(└> pointer이다!!)

○ 마지막 노드의 link 필드의 값은 NULL임.



화살표가 이렇게 되었다면 "화살표 시작이 끝의 주소를 가지고 있다." => 화살표: 프로그램에서 포인터(pointer)

=> "리스트"라는 변수가 첫번째 노드의 주소를 가지고 있다!
<헤드(head)>

p.190~199 6.5 단순 연결리스트의 연산 구현

• 단순 연결리스트의 구현보기 (1/4)
(단순 구조 연결 리스트 구현)

○ 하나의 노드를 표현하는 ListNode는 data 필드와 link 필드로 구성되는 구조체 형.

```
typedef int element;

typedef struct ListNode {
    element data;            // data 필드
    struct ListNode *link;   // Link필드
} ListNode;
```

하나의 구조체

type: struct ListNode * => 자기 리스트 노드의 포인터이다.

자기 참조 포인터 (자기 struct type의 포인터를 가리킨다).

○ 하나의 노드 생성은 동적 메모리 할당 함수인 malloc 사용.

```
ListNode *Node;        -head 라고도 한다.

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

-> ListNode의 크기만큼 할당 (node의 type이 ListNode이니까)

node는 ListNode의 주소가지고있다.

=> 하나의 일반적인 노드 구조

8byte

오른쪽의 자료가 완전히 들어있다!!

메모리 할당 8byte!!

(화살표는 pointer!!)

p.190

• 단순 연결 리스트 구현 보기 - (2/4)

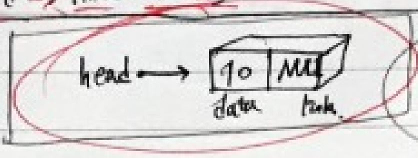
○ 하나의 노드 head를 생성, data 및 link 필드에 각각 10 및 NULL을 저장함
(head는 하나의 노드를 가지는 리스트임)

```
ListNode *head;

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

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

*head는 나중에 리스트의 시작 부분이 됨!! (리스트 이름처럼 됨)



이렇게 그리면 이렇게 완성됐다! 라고 이 두 개를 동시에 생각할 수 있어야 함!!

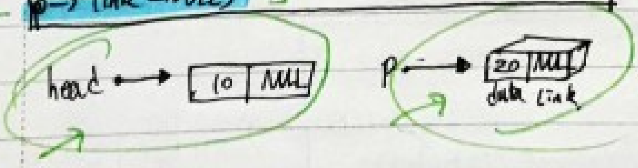
• 단순 연결 리스트 구현 보기 - (3/4)

○ 하나의 노드 p를 생성하고 data 및 link 필드에 각각 20 및 NULL을 저장함.

```
ListNode *p;

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

p->data = 20;
p->link = NULL;
```



하나 하나 일일히 노드를 다 만드는 방식
↓
+ 연결시키면 완성

• 단순 연결 리스트 구현 보기 - (4/4)

리스트 "head"

○ 리스트 head 뒤에 노드 p를 연결하여 리스트 head가 2개 노드로 구성되게 함!!

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

head가 리스트의 시작, 리스트 이름 그자체가 됨!!

2개의 노드가 됨.

연결되게 됨.

NULL → p를 가리키게 됨 (두 번째 노드의 주소를 가리키게 됨)

이 결과에서는 얘가 노드 2개 있는 리스트가 되어버림

p.190~199. ☆ 단순 연결 리스트의 연산

head는 리스트로서 첫번째 노드의 주소를 가지고 있다.

p.190.

- ListNode* insert_first(ListNode *head, int value): 리스트 head의 시작 위치에 data가 value인 새 노드를 추가.
  - 첫번째 → 5번째가 되고, …value가 첫번째에 새로운 노드.
- ListNode* insert(ListNode *head, ListNode *pre, element value): 리스트 head에서 노드 pre의 다음 노드로 data가 value인 새 노드를 추가.
  - pre가 가리키는 노드의 다음 노드에 새노드를 추가!
  - 노드의 link가 다음 노드를 가리키게 pre다음에 넣게 되도록
  - 새노드 먼저 생성!! (노드 다음에 다음 노드를 가리키니까!)
- ListNode* delete_first(ListNode *head): 리스트 head의 첫 노드를 삭제.
  - 노드는 head가 첫 주소를 가리키고 있으니 이를 argument로 갖게!!
- ListNode* delete(ListNode *head, ListNode *pre): 리스트 head에서 노드 pre의 다음 노드를 삭제.
  - pre다음노드를 삭제!!
  - 링크가 다음 노드를 가리키는 것을 생략!!
- ListNode* reverse(ListNode *head): 리스트 head의 역 리스트 만들기.
  - 〈굉장히 까다로움〉 리스트가 완전히 역순으로 됨.
- void print_list(ListNode *head): 리스트 head의 각 노드를 순서대로 출력.
  - 하나하나 '방문'하면서 출력한다!!!

이렇게 생긴 함수들.

p.191~192. • 함수 insert_first()

"함수가 어디인지 알기 항상 알고 있어야 한다!" ⇓ "문제를 풀기전에 자신이 쓸게 깊게 이해하고 풀게 중요하다!!"

```
// 리스트 head의 시작 위치에 data가
// value인 새 노드를 추가.

ListNode* insert_first(ListNode *head, int value)
{
    ListNode *p;

    p = (ListNode *) malloc (sizeof(ListNode));   // 1
    p -> data = value;                            // 2
    p -> link = head;                             // 3
    head = p;                                     // 4.

    return head;
}
```

(한 줄하고 그림을 또 한 줄하고) 공부할 때 왔다갔다 하면서 공부하자!

head가 첫번째 노드의 주소니 이걸 p의 link로 주어 연결시켜 link가 head를 가리키게 되며 head 리스트 앞에 하나가 삽입된 형태가 된다.

☆ 알고리즘 설명

- 1: 삽입할 새노드 p 생성 → 새 노드 공간 확보
- 2: data를 p에 저장
- 3: p→link을 현재의 head값으로 연결 → 첫째 노드를 가리키게 됨
- 4: head를 p값으로 연결 → p가 첫째노드가 됨

/5. 마무리로 새로 삽입된 head반환

p.192 ~194

• 함수 insert()

```
// 리스트 head에서 노드 pre의 다음
// 노드로 data가 value인 새 노드를 추가.

ListNode * insert(ListNode *head, ListNode *pre, element value)
{
    ListNode *p;

    p = (ListNode *)malloc(sizeof(ListNode));   // 1
    p->data = value;                            // 2
    p->link = pre->link;                        // 3
    pre->link = p;                              // 4

    return head;
}
```

(나중에 헷갈리지 않도록 이 코드를 쓸 수 있게 되어야 한다!!)

Q: 왜 pre노드의 뒤, 다음 노드로 삽입될까?

=> 리스트의 노드는

"다음 노드를 가리키기 때문!!

(다음 노드를 가리켜서 link에 넣어도 다음 노드를 가리키게 되는 거임)

p.197 삽입과정을 설명.

1. 새로운 노드를 생성, 변수 p를 가리킨다.
2. data 저장 (value)
3. p의 링크를 pre의 링크로 해서 다음 노드를 가리키게 함 (3)
4. pre의 link를 p를 가리키게 해서 p를 연결시킴(4).

p.194 ~ 195

• delete_first()

★ 새로운 노드 추가하는 삭제에서 사용하는 함수.

```
// 리스트 head의 첫 노드를 삭제

ListNode * delete_first(ListNode *head, int value)
{
    ListNode *removed;

    if (head == NULL) return NULL;
    removed = head;        //1
    head = head->link;     //2.
    free(removed);         //3..

    return head;
;
```

→ head가 NULL이면 삭제할 노드가 없기에 NULL 반환

→ removed가 첫노드를 가리키게

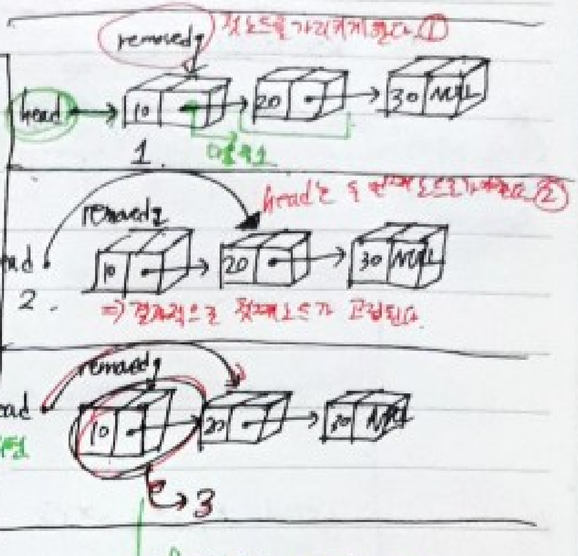
★ 굉장히 중요한 부분!!

삭제를 해야하는 head는 두번째 노드를 가리키는데 head = head→link이다, 왜일까?

⇒ head의 link는 두번째 노드를 가리키고 있기 때문!!

★ (head는 첫째노드를 가리키고있다) ★

★ 해당 메모리를 삭제 ★



★ 알고리즘 설명
1. 헤드 포인터값 → removed에 복사 (삭제 준비)
2. 헤드 포인터 값 → head→link로 변경
3. removed가 가리키는 동적 메모리 반환!

p.195 ~ 197.

• 함수 delete()

★ pre의 뒤것을 삭제!!

```
// 리스트 head에서 노드 pre의 다음
// 노드를 삭제

ListNode * delete(ListNode *head, ListNode *pre)
{
    ListNode *removed;
    removed = pre->link;              //1
    pre->link = removed->link;        //2
    free(removed);                    //3
    return head;
}
```

→ pre가 다음노드를 가리키니 이것 removed에 담는다.

→ 그 다음 (다다음)가리키니 pre가 다다음노드를 가리키게 한다 ⇒ 중간노드가 고립

★ 이는 이전노드로 다른노드의 링크가 다음노드의 주소를 가리키게 pre노드의 뒤, 다음노드가 삭제되는 것이다!!

★ 알고리즘 설명
1. 삭제할 노드 찾기
2. 노드 건너뛰기 연결
3. 고립된 노드 메모리 제거.

<아이디어>

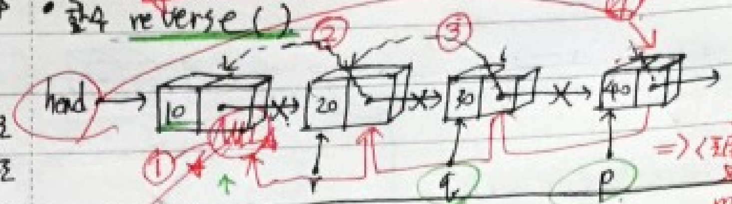
① head는 끝 노드를 가리키게 한다.
② 각 노드의 link 값은 각 앞 노드를 가리키게 한다.
③ 첫 노드의 링크는 NULL로 넣는다.

→ 프로그램 순서는 ③ → ② → ① 이다.

자주 나오는 중요

• 함수 reverse()

리스트의 역순 리스트 return.
⇒ 노드 생성, 삭제 X,
기존 리스트를 그대로 이용해서 link만 변경, return해주면 된다.
data는 그대로 두고!! link만 변경!!



⇒ <최종 상태>
malloc, free X.

(r–q–p 순서로 head부터 오른쪽으로 순서대로 이동하며 역순 처리한다).

```
// 리스트 head의 역 리스트 만들기.

ListNode *reverse(ListNode *head)
{
    ListNode *p, *q, *r;   // 3개 포인터 p, q, r을 사용.

    p = head; q = NULL;    // p는 head로, q는 NULL로 초기화.
    while(p != NULL){
        r = q;             // r은 q를 따라감.
        q = p;             // q는 p를 따라감.
        p = p->link;       // p는 p->link를 따라감
        q->link = r;       // q의 link를 역 방향인 r로 변경.
    }
    return q;              // q가 역 리스트
}
```

r → q → p 순서 (p가 이동하면 따라서 이동!!)
<한 칸씩 차례대로 이동>

처음 q는 NULL이다
처음 r도 NULL이 된다

포인터 3개로 p, q, r을 활용
r → q, q → p를 따라간다.
<p, q, r의 관계>

다음 노드로 넘어가기

역순으로 하면 첫 노드의 link는 NULL이 된다.

q의 link → 한 칸씩 p가 가리킴
그리고 q의 link에 r을 넣으면? ⇒ 화살표가 반대로 향함
→ 계속 진행해서 역순으로 바뀌게 됨!!

역순 변경의 역할 ⇒ q

r ⇒ 역순 변경의 대상 (무엇이)
p ⇒ 리더, 앞서가는 것?

즉, 전체적인 아이디어는
계속해서 <반복>적으로 오른쪽으로 가며
"한 칸"씩 가서 q의 link를 가리키게 하고,
또 "한 칸" 가서 r을 가리키게 하고
⋮
계속 한 칸씩 나아가며
p가 NULL이 될 때까지 간다!!
(끝까지 도달)

• 함수 print_list()

```
// 리스트 head의 각 노드를 순서대로 출력.

void print_list(ListNode *head)
{
    ListNode *p;

    for (p = head; p != NULL; p = p->link)
        printf("%d", p->data);
}
```

p가 NULL이 될 때까지 끝노드까지 하나씩 출력 돌린다!!

포인터 조건 3개
중요!! 항상 알고 있어야 한다!!

p.209~ 6.6 연결 리스트의 응용: 다항식.

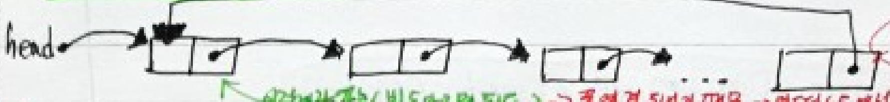
$A(x) = a_{m-1}x^{e_{m-1}} + \cdots a_0 x^{e_0}$ 형태의 다항식을 단순 연결 리스트로 처리가능.

p.209부터 참고

7.1 원형 연결 리스트

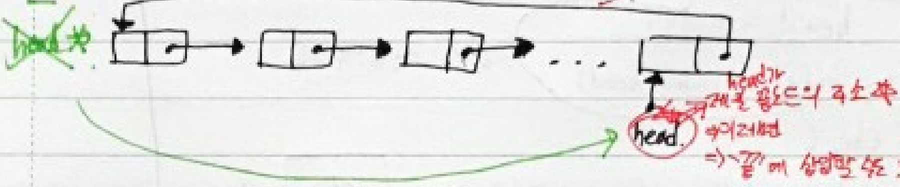
p.222 ~ p.229. • 원형 연결 리스트 (Circular linked list) <교재 p.222 원형>

○ 원형 연결 리스트는 단순 연결 리스트에서 마지막 노드의 link가 NULL이 아니고 첫 노드를 가리키는 리스트로 한 노드에서 다른 모든 노드로의 접근이 가능.

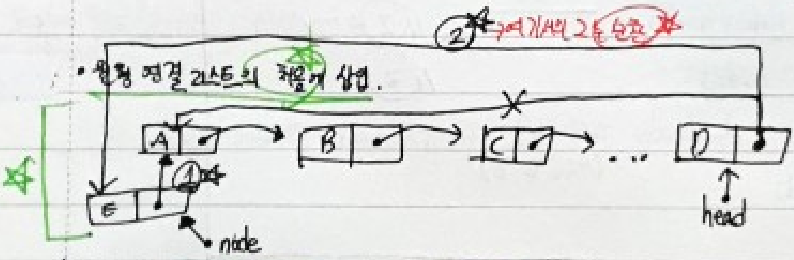


○ 원형 연결 리스트에서 head가 마지막 노드를 가리키게 하면 리스트의 시작 위치 혹은 끝 위치에 노드를 삽입하는 연산이 단순 연결 리스트에 비하여 용이함.

길이 원형이라 빙 돌아갈 수 있다!



p.223~224 • 원형 연결 리스트의 처음에 삽입.

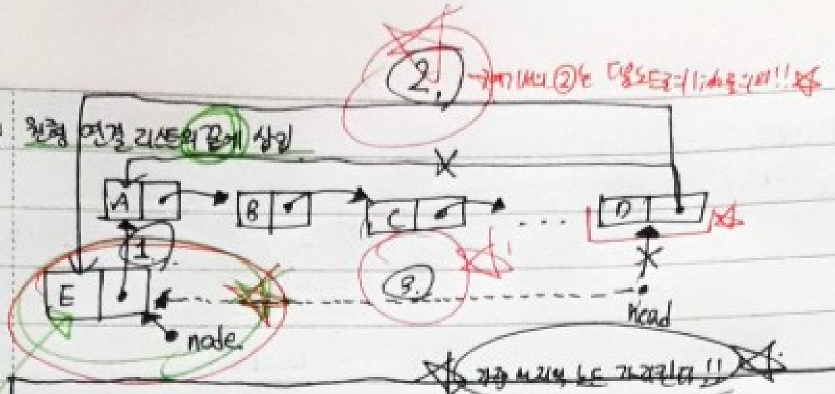



```
ListNode* insert_first(ListNode *head, element data)
{
  ListNode *node;

  node = (ListNode *)malloc(sizeof(ListNode));
  node->data = data;
  if(head == NULL) {
    head = node;
    node->link = head;
  } else {
    node->link = head->link;   //1
    head->link = node;         //2.
  }
  return head;
}
```

- head: 맨 마지막 노드 가리키기!!
- node = malloc / node->data = data: 노드 하나 만들고, data 넣기 <기본>
- if(head == NULL): head가 NULL은 "노드가 하나도 없다, empty 상태."라는 의미.
  ⇓ 단일모드, 노드가 1개인 경우이다. — 노드의 1개 생성하는 느낌.
- head = node; → head가 node를 가리키게 해서 빈 list에 1개 추가.
- node->link = head; → 자기 자신을 가리키게 함
- else: → (head) 첫 노드 앞에 새로운 (node) 첫노드 삽입.
  <원형에서 head->link가 가리키는 곳이 첫노드이다!!>
- head->link = node; └ 마지막 노드의 순환을 새로운 첫 삽입노드로 연결

p.224 원형 연결 리스트의 끝에 삽입




```
ListNode * insert_last (ListNode * head, element data)
{
    ListNode *node;

    node = (ListNode*)malloc(sizeof(ListNode));
    node -> data = data;
    if (head == NULL) {              → empty list
        head = node;
        node -> link = head;         알과 작성 다 봤음.
    } else {
        node -> link = head -> link;     // 1 ☆ → 첫노드 가리키게 함 (원형 형성 중에)
        head -> link = node;             // 2 ☆ → 현재의 끝노드를 E인 node로 바꾸서
                                                   head가 끝이 아닌
                                                   새로운 끝을 알린다!
        head = node;                     // 3 ☆
    }   └ head도 다시 끝을 가리키게 해서 마무리를 한다.
          (새로운 끝, E)
    return head;
}
```

7.2 원형 연결 리스트는 어디에 사용될까?

1. 컴퓨터가 여러 응용 프로그램을 하나의 CPU로 할 때 필요

2. 멀티 플레이어 게임 => 모든 플레이어는 원형 연결리스트에 저장되며
한 플레이어의 기회가 끝나면 포인터를 다시 앞으로 옮겨서
다음 플레이어의 순서가 된다.

<그 외의 설명은 p.227 ~ 의 설명 참고>

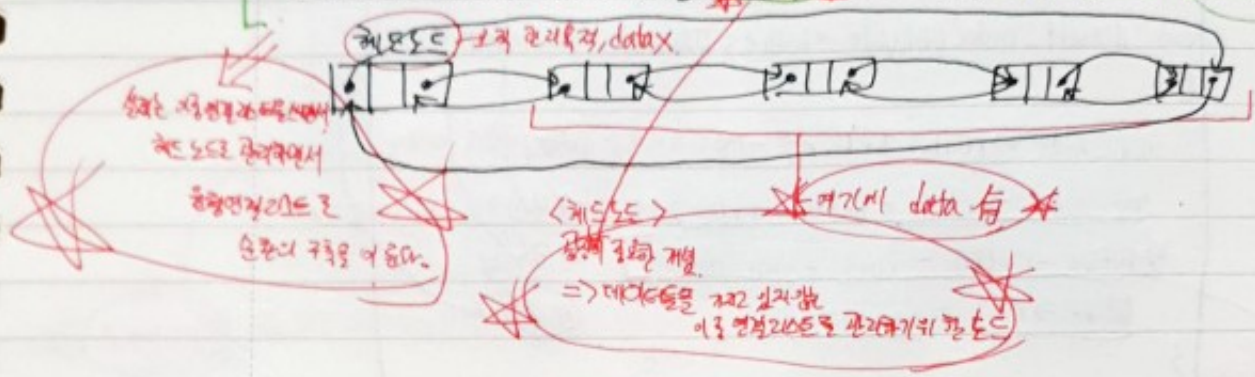
p.230 ~236. 7.3 이중 연결 리스트

• 이중 연결리스트 (Doubly linked list)

○ 단순 연결 리스트에서 어떤 노드의 삽입 혹은 삭제 시에는 반드시 이전 노드가 필요하다. 어떤 노드의 이전 노드를 찾기가 어려움.

○ 이중 연결 리스트는 하나의 노드가 이전노드와 다음 노드를 가리키는 두 개의 link를 가지는 리스트로 어떤 노드에서 양방향으로 노드 검색이 가능.

○ 실제 사용되는 이중 연결 리스트 (여러 그림) = 헤드 노드 + 이중 연결리스트 + 원형 연결 리스트 일.



• 이중 연결 리스트 헤드 노드

○ 헤드 노드(head node) : 데이터는 없이 단지 삽입,삭제 코드를 간단하게할 목적으로 만들어진 노드.

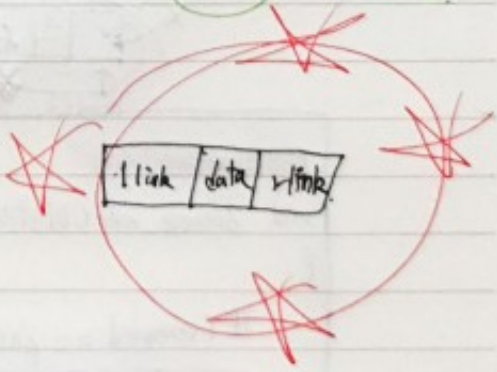
⊙ 빈(empty) 리스트인 경우에는 헤드 노드만 존재.

빈 리스트 (only 헤드노드 有)

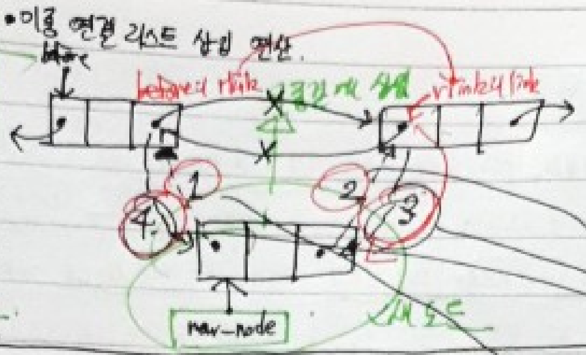
들어오기(리스트를 가리킬)

○ 이중 연결 리스트에서의 노드 구조체.

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



p.232 • 이중 연결 리스트 삽입 연산.

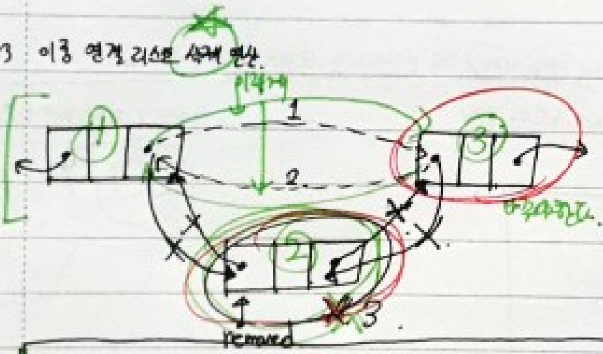



```
// 노드 new_node를 노드 before의 오른쪽에 삽입.
void dinsert_node(DlistNode *before, DlistNode *new_node)
{
    new_node->llink = before;            — 새노드의 왼쪽을 before의 이전을 가리키게
    new_node->rlink = before->rlink;     새노드의 오른쪽을 다음 노드를 가리키게
    before->rlink->llink = new_node;     //3
    before->rlink = new_node;            //4
}
```

before의 rlink의 llink → 다음노드의 llink

쏙 들어감.

p.233 이중 연결 리스트 삭제 연산.



* 여태까지는 삭제는 항상 그 구의 두개값을 삭제하는 것이었지만, 이중 연결리스트는 그냥 그자리를 주면 그것의 삭제가 가능하다!!

```
// 노드 removed를 삭제.
void dremove_node(DlistNode *phead_node, DlistNode *removed)
{                                                   너가 삭제
    if (removed == phead_node) return;
①  removed->llink->rlink = removed->rlink;   //1.   ③의 llink
③  removed->rlink->llink = removed->llink;   //2.   └①의 rlink
    free(removed);                            //3.
}   └free, 삭제!
```

※ 삭제하고자 하는 노드의 전노드의 rlink가 다음노드의 llink가 되고, 다음노드의 llink가 전노드의 rlink가 되었습니다!!

① ⇄ ③ 해주면 된다!!

p.234~238 7.4 예제 : mp3 재생 프로그램 만들기.

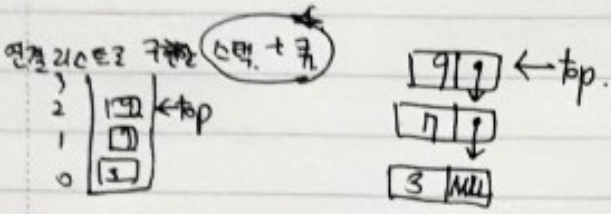
• mp3 ⇒ 현재 곡에서 이전 곡 or 다음 곡 간다 + 재생 or 정지버튼을 가정한다.

⇒ 따라 곡들에서 이전 항목과 다음 항목을 쉽게 이동할 수 있는 자료구조 사용

⇒ 이중 연결 리스트 !!

(p.239~238의 프로그램 코드 참조)

p.239~243 7.5 연결리스트로 구현한 스택 + 큐

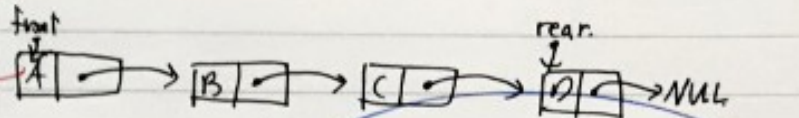


(a) 배열 스택 (b) 연결리스트 스택

⇒ 연결된 스택 (linked stack)

(코드는 p.241~242 참조)

덱(큐) ※ 크기의 제한이 없어 배열보다 훨씬 쉽게 삽입, 삭제 가능



⇒ 연결된 큐 (linked queue)