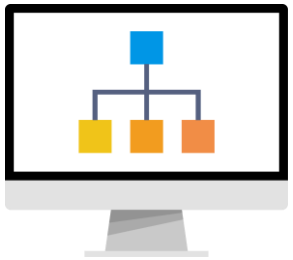
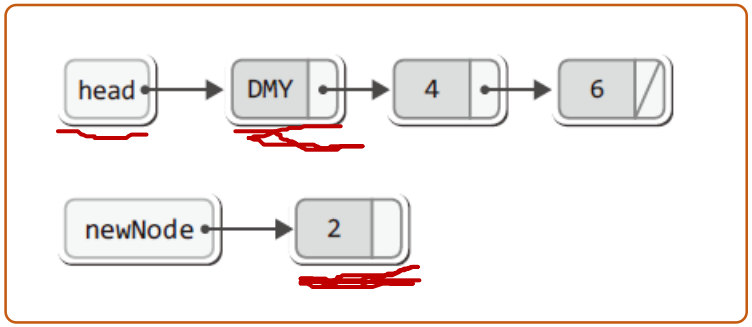


# Data Structure & Algorithm
# 자료구조 및 알고리즘

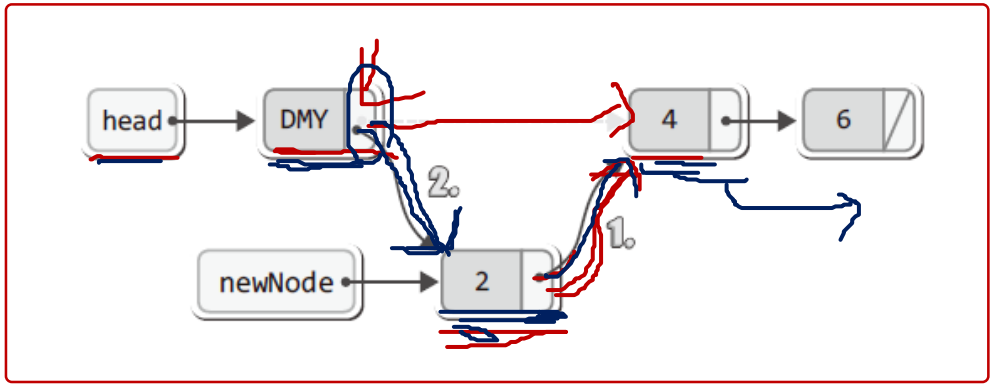
## 7. 연결 리스트 (Chapter 4, Linked List part 3)

# 더미 노드 연결 리스트 구현: 삽입

```
void FInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;

    newNode->next = plist->head->next;
    plist->head->next = newNode;

    (plist->numOfData)++;
}
```

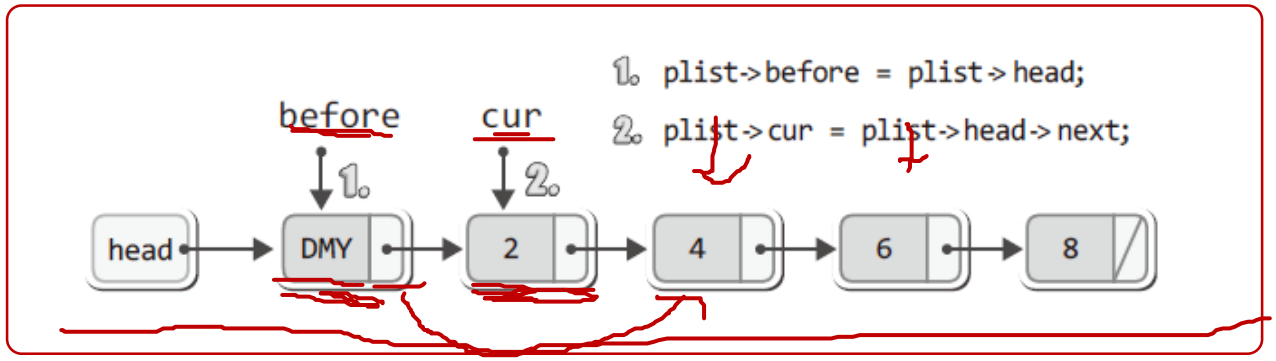



모든 경우에 있어서 동일한 삽입과정을 거친다는 것이 더미 노드 기반 연결 리스트의 장점!



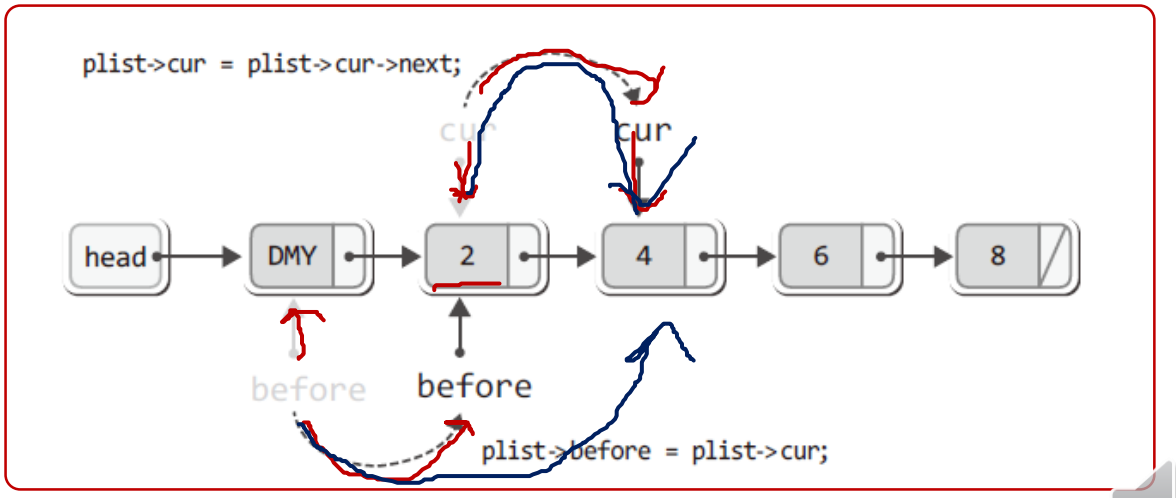

# 더미 노드 연결 리스트 구현: 참조1

```
int LFirst(List * plist, LData * pdata)
{
    if(plist->head->next == NULL)        // 더미 노드가 NULL을 가리킨다면,
        return FALSE;                    // 반환할 데이터가 없다!

    plist->before = plist->head;         // before는 더미 노드를 가리키게 함
    plist->cur = plist->head->next;      // cur은 첫 번째 노드를 가리키게 함

    *pdata = plist->cur->data;           // 첫 번째 노드의 데이터를 전달
    return TRUE;                         // 데이터 반환 성공!
}
```

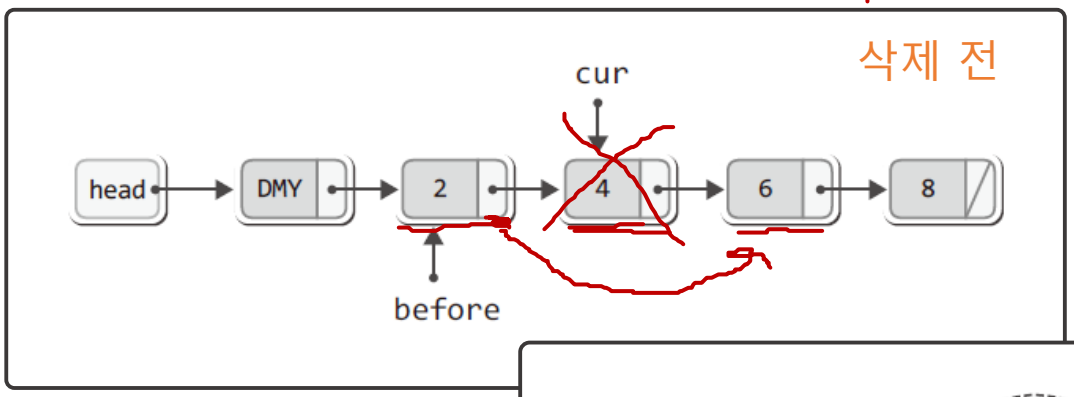


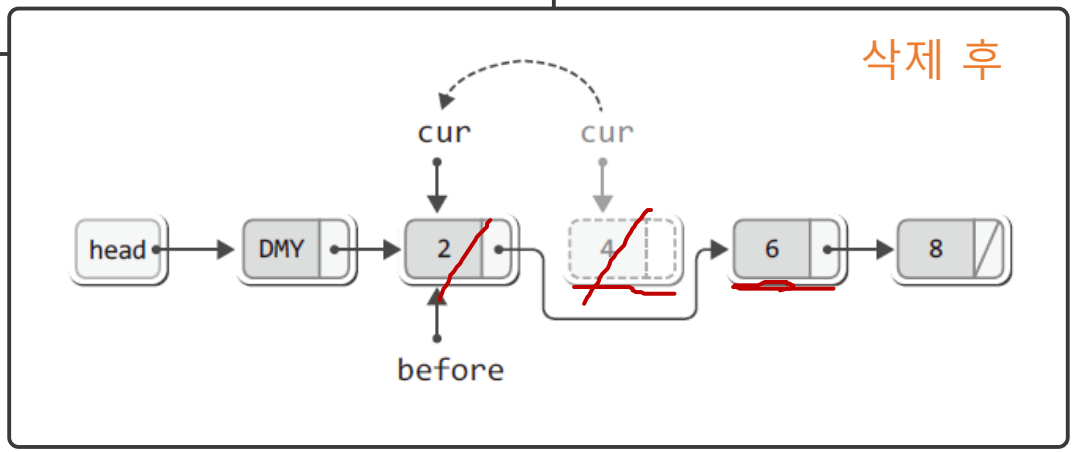

# 더미 노드 연결 리스트 구현: 참조2

```
int LNext(List * plist, LData * pdata)
{
    if(plist->cur->next == NULL)        // 더미 노드가 NULL을 가리킨다면,
        return FALSE;                   // 반환할 데이터가 없다!

    plist->before = plist->cur;         // cur이 가리키던 것을 before가 가리킴
    plist->cur = plist->cur->next;      // cur은 그 다음 노드를 가리킴

    *pdata = plist->cur->data;          // cur이 가리키는 노드의 데이터 전달
    return TRUE;                        // 데이터 반환 성공!
}
```




# 더미 노드 연결 리스트 구현: 삭제1

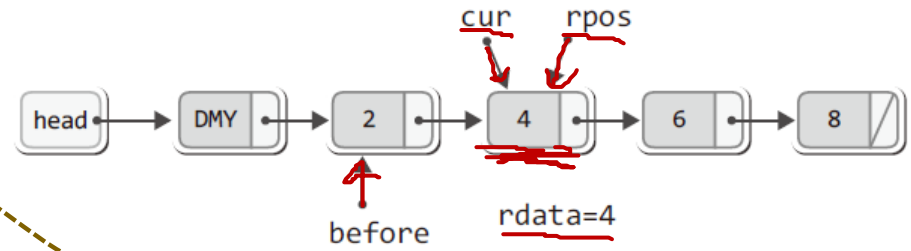


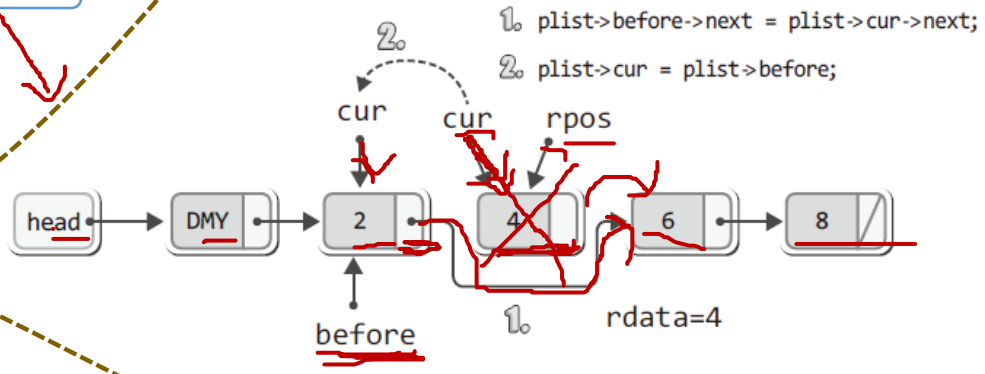



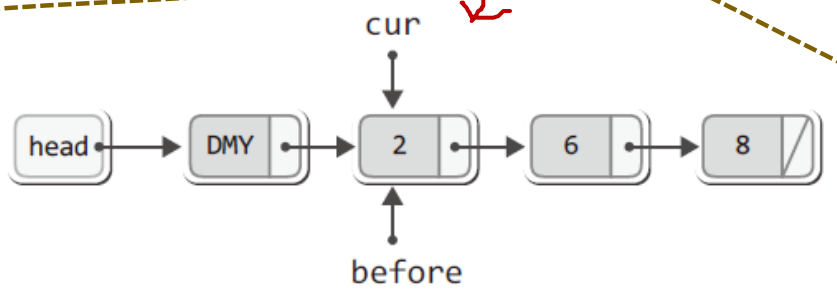

cur은 삭제 후 재조정의 과정을 거쳐야 하지만 before는 LFirst or LNext 호출 시 재설정되므로 재조정의 과정이 불필요하다.

# 더미 노드 연결 리스트 구현: 삭제2

```
LData LRemove(List * plist)
{
    Node * rpos = plist->cur;
    LData rdata = rpos->data;

    plist->before->next = plist->cur->next;
    plist->cur = plist->before;

    free(rpos);
    (plist->numOfData)--;
    return rdata;
}
```



# 더미 기반 단순 연결 리스트

## DLinkedlist.c
## DLinkedlist.h
## DLinkedlistmain.c

**실행결과**

```
현재 데이터의 수: 5
33 22 22 11 11

현재 데이터의 수: 3
33 11 11
```

Chapter 03의 ListMain.c의 main 함수와 완전히 동일하다.
다만 노드를 머리에 추가하는 방식이기 때문에 실행결과에서는 차이가 난다.

# 연결 리스트의 정렬 삽입의 구현

# 정렬기준 설정과 관련된 부분

단순 연결 리스트의 정렬 관련 요소 세 가지

- 정렬기준이 되는 함수를 등록하는 SetSortRule 함수
- SetSortRule 함수 통해 전달된 함수정보 저장을 위한 LinkedList의 멤버 comp
- comp에 등록된 정렬기준을 근거로 데이터를 저장하는 SInsert 함수

하나의 문장으로 구성한 결과

"SetSortRule 함수가 호출되면서 정렬의 기준이 리스트의 멤버 comp에 등록되면, SInsert 함수 내에서는 comp에 등록된 정렬의 기준을 근거로 데이터를 정렬하여 저장한다."

# SetSortRule 함수와 멤버 comp

1. SetSortRule 함수의 호출을 통해서 . . . .

```
void SetSortRule(List * plist, int (*comp)(LData d1, LData d2))
{
    plist->comp = comp;
}
```

```
typedef struct _linkedList
{
    Node * head;
    Node * cur;
    Node * before;
    int numOfData;
    int (*comp)(LData d1, LData d2);
 } LinkedList;
```

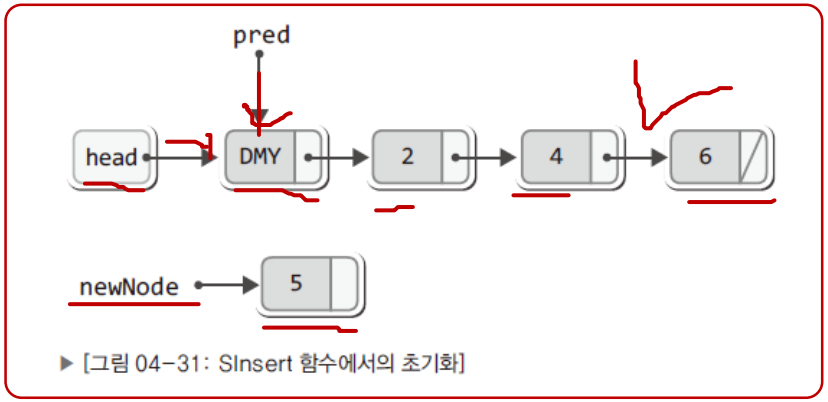
2. 멤버 comp가 초기화되면. . . .

```
void LInsert(List * plist, LData data)
{
    if(plist->comp == NULL)
        FInsert(plist, data);
    else
        SInsert(plist, data);
 }
```

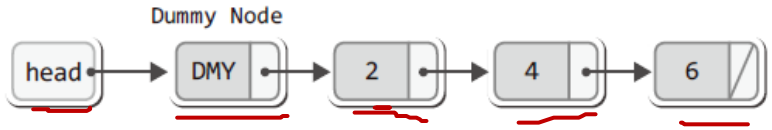
3. 정렬 관련 SInsert 함수가 호출된다.

# SInsert 함수1

```
void SInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    Node * pred = plist->head;
    newNode->data = data;

    // 새 노드가 들어갈 위치를 찾기 위한 반복문!
    while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
    {
        pred = pred->next; // 다음 노드로 이동
    }
    newNode->next = pred->next;
    pred->next = newNode;
    (plist->numOfData)++;
}
```




▶ [그림 04-31: SInsert 함수에서의 초기화]




▶ [그림 04-30: 값의 대소가 정렬의 기준인 연결 리스트]

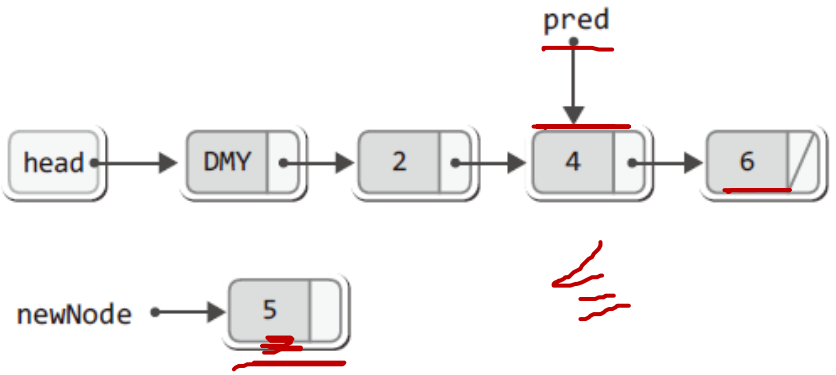
위 상황에서 다음 문장이 실행되었다고 가정!

SInsert(&slist, 5);

# SInsert 함수2

```
void SInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    Node * pred = plist->head;
    newNode->data = data;

    // 새 노드가 들어갈 위치를 찾기 위한 반복문!
    while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
    {
        pred = pred->next; // 다음 노드로 이동
    }
    newNode->next = pred->next;
    pred->next = newNode;
    (plist->numOfData)++;
}
```

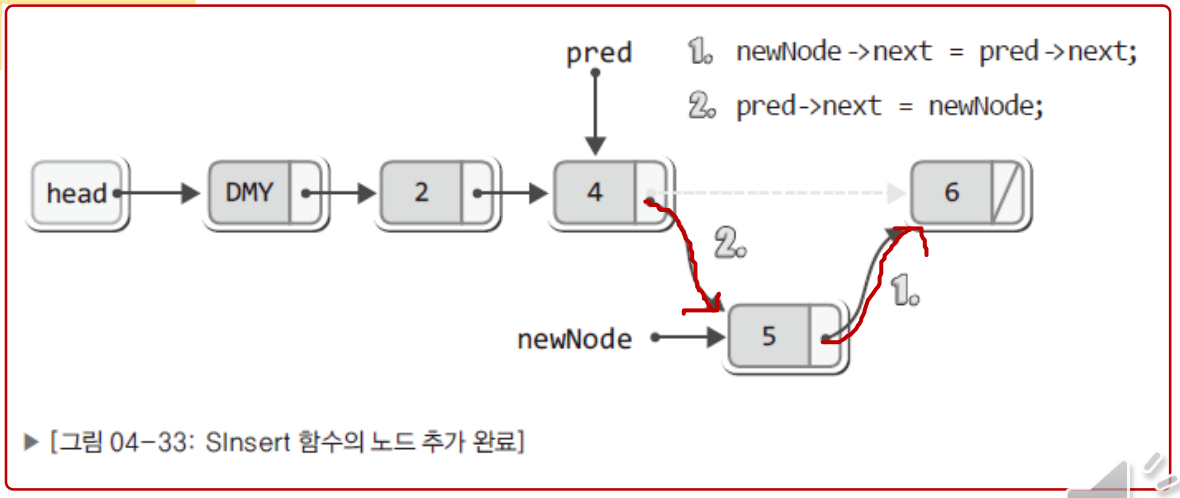
comp가 0을 반환한다는 것은 첫 번째 인자인 data가 정렬 순서상 앞서기 때문에 head에 가까워야 한다는 의미!




▶ [그림 04-32: SInsert 함수의 while문 탈출 이후]

# SInsert 함수3

```
void SInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    Node * pred = plist->head;
    newNode->data = data;

    // 새 노드가 들어갈 위치를 찾기 위한 반복문!
    while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
    {
        pred = pred->next; // 다음 노드로 이동
    }
    newNode->next = pred->next;
    pred->next = newNode;
    (plist->numOfData)++;
}
```




▶ [그림 04-33: SInsert 함수의 노드 추가 완료]

# 정렬의 핵심인 while 반복문

· 반복의 조건 1 pred->next != NULL

pred가 리스트의 마지막 노드를 가리키는지 묻기 위한 연산

· 반복의 조건 2 plist->comp(data, pred->next->data) != 0

새 데이터와 pred의 다음 노드에 저장된 데이터의 우선순위 비교를 위한 함수호출

```
while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
{
    pred = pred->next;    // 다음 노드로 이동
}
```

# comp의 반환 값과 그 의미

```
while(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
{
    pred = pred->next;    // 다음 노드로 이동
}
```

우리의 결정 내용! 이 내용을 근거로 SInsert 함수를 정의하였다.

- comp가 0을 반환

  첫 번째 인자인 data가 정렬 순서상 앞서서 head에 더 가까워야 하는 경우

- comp가 1을 반환

  두 번째 인자인 pred->next->data가 정렬 순서상 앞서서 head에 더 가까워야 하는 경우

# 정렬의 기준을 설정하기 위한 함수의 정의

함수의 정의 기준

- 두 개의 인자를 전달받도록 함수를 정의한다.
- 첫 번째 인자의 정렬 우선순위가 높으면 0을, 그렇지 않으면 1을 반환한다.

```
int WhoIsPrecede(int d1, int d2)
{
    if(d1 < d2)
        return 0;        // d1이 정렬 순서상 앞선다.
    else
        return 1;        // d2가 정렬 순서상 앞서거나 같다.
}
```

오름차순 정렬을 위한 함수의 정의

작은 수의 우선 순위가 높다

= 작은 수가 앞에 온다

= 뒤로 갈수록 커진다 = 오름차순

정렬 관련된 함수를 DLinkedListSortMain.c에 포함시켜야 함을 이해한다.

- Dlinkedlist.c
- DLinkedlist.h
- DLinkedlistSortmain.c

```
현재 데이터의 수: 5
11 11 22 22 33

현재 데이터의 수: 3
11 11 33
```

실행결과

# 요약

- 삭제 시 해당하는 노드를 free 함수 호출을 통해 지우고 앞뒤의 노드들을 포인터를 활용해 연결하는 것에 유의한다.

- 함수 포인터를 이용하여 사용자가 자료 구조의 동작(정렬 기준)에 개입할 수 있도록 할 수 있다.

- 정렬 삽입 시: 뒤로 가본다! 언제까지? 끝에 도착하거나 새로 추가된 값 보다 더 큰 값을 찾을 때까지!

# 출석 인정을 위한 보고서 작성

- A4 반 장 이상으로 아래 질문에 답한 후 포털에 있는 과제 제출란에 제출

1) 연결 리스트는 배열과 달리 *i*번째 노드의 데이터를 가져오는 인터페이스가 없다. 오늘 배운 함수들을 활용하여 리스트 plist의 *i*번째 데이터를 반환하는 함수 LData LIndex(List *plist, int i)를 의사 코드로 나타내 보세요.
   - 비효율적이더라도 괜찮음.

2) 어떤 연결 리스트 L1의 순서를 뒤집은 L2를 만들고 싶다. 어떻게 하면 될까?
   - 비효율적이더라도 괜찮음.
   - 정렬 규칙을 사용하지 않는다고 가정