

Data Structure & Algorithm 자료구조 및 알고리즘

9. 연결 리스트 (Chapter 5, Linked List part 4)



원형 연결 리스트

(Circular Linked List)

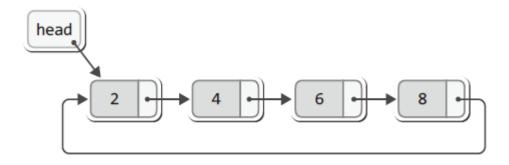
원형 연결 리스트의 이해





▶ [그림 05-1: 단순 연결 리스트]

단순 연결 리스트의 마지막 노드는 NULL을 가리킨다.

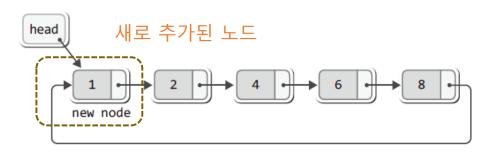


▶ [그림 05-2: 원형 연결 리스트]

원형 연결 리스트의 마지막 노드는 첫 번째 노드를 가리킨다.

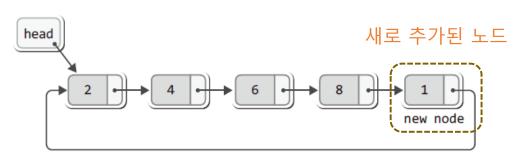
원형 연결 리스트의 노드 추가





이렇듯 모든 노드가 원의 형태를 이루면서 연결되어 있기 때문에 원형 연결 리스트에서는 사실상 머리와 꼬리의 구분이 없다.

▶ [그림 05-3: 원형 연결 리스트의 머리에 노드 추가]



▶ [그림 05-4: 원형 연결 리스트의 꼬리에 노드 추가]

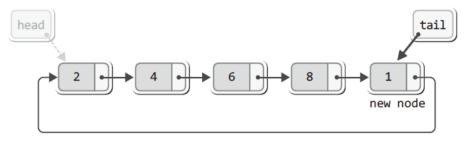
- · 같다 두 경우의 노드 연결 순서가 같다.
- · 다르다 head가 가리키는 노드가 다르다.

원형 연결 리스트의 대표적인 장점



"단순 연결 리스트처럼 머리와 꼬리를 가리키는 포인터 변수를 각각 두지 않아도, 하나의 포인터 변수만 있어도 머리 또는 꼬리에 노드를 간단히 추가할 수 있다."

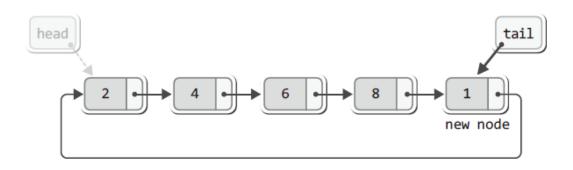
앞서 소개한 모델을 기반으로는 위의 장점을 살리기 어렵다 (꼬리를 못 찾기 때문에). 그래서 우리는 변형된, 그러나 보다 일반적이라고 인식이 되고 있는 변경된 원형 연결 리스트를 구현한다.



▶ [그림 05-6: 변형된 원형 연결 리스트]

변형된 원형 연결 리스트





▶ [그림 05-6: 변형된 원형 연결 리스트]

· 꼬리를 가리키는 포인터 변수는? tail 입니다!

· 머리를 가리키는 포인터 변수는? tail->next 입니다!

이렇듯 리스트의 꼬리와 머리의 주소 값을 쉽게 확인할 수 있기 때문에 연결리스트를 가리키는 포인터 변수는 하나만 있으면 된다.

변형된 원형 연결 리스트의 구현범위



· 조회관련 LFirst	이전에 구현한 연결 리스트와 기능 동일
· 조회관련 LNext	원형 연결 리스트를 계속해서 순환하는 형태로 변경!
· 삭제관련 LRemove	이전에 구현한 연결 리스트와 기능 동일
· 삽입관련	앞과 뒤에 삽입이 가능하도록 두 개의 함수 정의!
· 정렬관련	정렬과 관련된 부분 전부 제거
・이외의 부분	이전에 구현한 연결 리스트와 기능 동일

원형 연결 리스트는 그 구조상 끝이 존재하지 않는다. 따라서 LNext 함수는 계속해서 호출이 가능하고, 이로 인해서 리스트를 순환하면서 저장된 값을 반환하도록 구현한다.

원형 연결 리스트의 헤더파일과



```
typedef int Data;

typedef struct _node
{
    Data data;
    struct _node * next;
} Node;

typedef struct _CLL
{
    Node * tail;
    Node * cur;
    Node * before;
    int numOfData;
} CList;
```

```
void ListInit(List * plist); 노드를 꼬리에 추가!
void LInsert(List * plist, Data data);
void LInsertFront(List * plist, Data data);
int LFirst(List * plist, Data * pdata);
int LNext(List * plist, Data * pdata);
Data LRemove(List * plist);
int LCount(List * plist);
```

```
void ListInit(List * plist)
{
    plist->tail = NULL;
    plist->cur = NULL;
    plist->before = NULL;
    plist->numOfData = 0;
}
```

모든 멤버를 NULL과 0으로 초기화한다.

원형 연결 리스트 구현: 첫 번째 노드 삽입



```
// LInsert & LInsertFront의 공통 부분
void LInser~(List * plist, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
                           첫 번째 노드라면
                                                                           tail
    if(plist->tail == NULL)
                                          tail•
                                                NULL
                                                           $$
       plist->tail = newNode;
                                                         추가 완료
       newNode->next = newNode;
                                                                      new node
                                         new node
    else
          두 번째 이후의 노드라면
                                          첫 번째 노드는 그 자체로 머리이자 꼬리이기
       . . . . 차이가 나는 부분 . . . .
                                          때문에 노드를 뒤에 추가하건 앞에 추가하건 그
                                          결과가 동일하다!
    (plist->numOfData)++;
```

원형 연결 리스트 구현: 두 번째 이후 노드 머리로 삽입

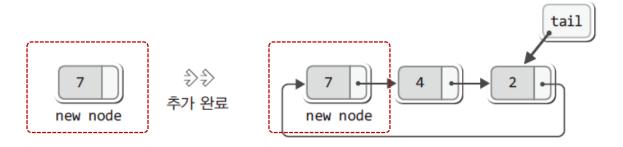


두 번째 이후의 노드 머리에 추가!

```
void LInsertFront(List * plist, Data data)
                                                                           tail
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    if(plist->tail == NULL)
    {
        plist->tail = newNode;
                                                     새 노드 추기
        newNode->next = newNode;
                                                                       new node
                                                                                     tail
    else
    {
        newNode->next = plist->tail->next;
                                                             new node
        plist->tail->next = newNode;
                                                          두 번째 이후 노드를 머리에 추가
    (plist->numOfData)++;
```

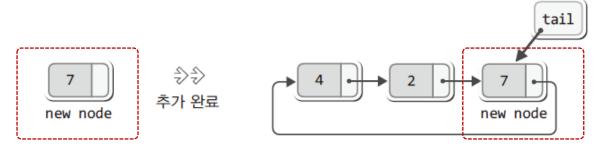
원형 연결 리스트 구현: 앞과 뒤의 삽입 과정 비교1





노드를 머리에 추가한 결과

그림상에서 보았을 때 이 둘의 실질적인 차이점은?



노드를 꼬리에 추가한 결과

원형 연결 리스트 구현: 앞과 뒤의 삽입 과정 비교2



두 번째 이후의 노드 꼬리에 추가!

```
tail의 위치가 유일한 차이점이다!
void LInsert(List * plist, Data data)
                                                              new node
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
                                                 $$
                                                꼬리 추가
    if(plist->tail == NULL)
                                                                                  new node
       plist->tail = newNode;
       newNode->next = newNode;
                                                                     $$
                                                                    머리 추가
    else
       newNode->next = plist->tail->next;
       plist->tail->next = newNode;
       plist->tail = newNode;
                              LinsertFront 함수에는 없는 문장!
                              -<u>LinsertFront</u>-함수와의 유일한 차이점!
    (plist->numOfData)++;
```

원형 연결 리스트 구현: 조회 LFirst



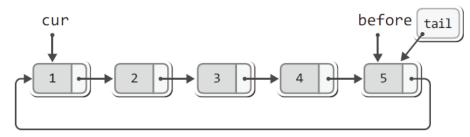
```
int LFirst(List * plist, Data * pdata)
{
    if(plist->tail == NULL)
        return FALSE;

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

*pdata = plist->cur->data;
return TRUE;
}
```

```
typedef struct _CLL
{
    Node * tail;
    Node * cur;
    Node * before;
    int numOfData;
} CList;
```

cur이 가리키는 노드가 머리!



▶ [그림 05-12: LFirst 함수의 호출결과]

원형 연결 리스트 구현: 조회 LNext

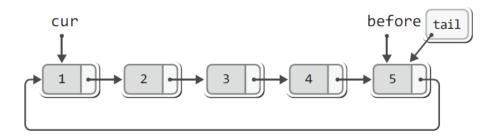


```
int LNext(List * plist, Data * pdata)
{
    if(plist->tail == NULL)
        return FALSE;

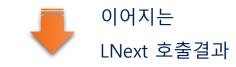
    plist->before = plist->cur;
    plist->cur = plist->cur->next;

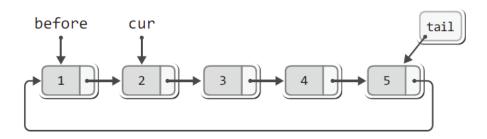
    *pdata = plist->cur->data;
    return TRUE;
}
```

원형 연결 리스트이므로 리스트의 끝을 검사하는 코드가 없다!



▶ [그림 05-12: LFirst 함수의 호출결과]





▶ [그림 05-13: LNext 함수의 호출결과]

원형 연결 리스트 구현: 노드의 삭제(복습)



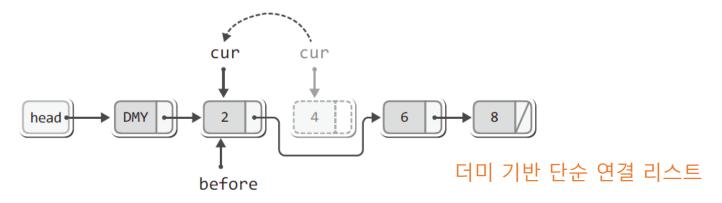
더미 노드 기반 연결 리스트의 삭제 과정 복습!

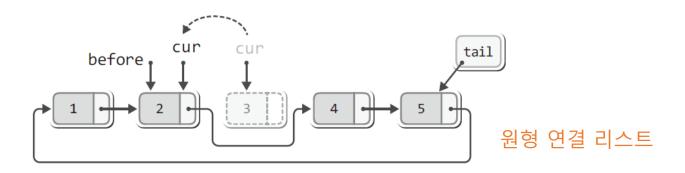
- · 핵심연산 1.
 - 삭제할 노드의 이전 노드가, 삭제할 노드의 다음 노드를 가리키게 한다.
- · 핵심연산 2.
 - 포인터 변수 cur을 한 칸 뒤로 이동시킨다.

```
LData LRemove(List * plist)
                       더미 기반 단순 연결 리스트의 삭제!
                                                             핵심연산 2
   Node * rpos = plist->cur;
                                                         cur
                                                               cur
   LData rdata = rpos->data;
                             핵심연산 /
   plist->before->next = plist->cur->next;
                                                              핵심연산 /
   plist->cur = plist->before;
                                                        before
                     핵심연산 2
   free(rpos);
   (plist->numOfData)--;
   return rdata;
                                 원형 연결 리스트의 삭제도 이와 별반 다르지 않다!
```

원형 연결 리스트 구현: 노드의 삭제(그림 비교)







그림상으로는 두 연결 리스트의 삭제 과정이 비슷해 보이나 원형 연결 리스트에는 더미 노드가 없기 때문에 삭제의 과정이 상황에 따라서 달라진다.

원형 연결 리스트 노드 삭제 구현



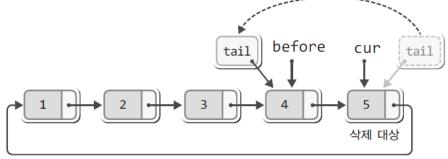
```
plist->before->next = plist->cur->next;
plist->cur = plist->before;

단순 연결 리스트와 동일한 코드
free(rpos);
(plist->numOfData)--;
return rdata;
```

- · 예외적인 상황 1 삭제할 노드를 tail이 가리키는 경우
- · 예외적인 상황 2

 삭제할 노드를 tail이 가리키는데,

 그 노드가 마지막 노드라면



▶ [그림 05-16: 삭제의 예외적인 경우]

예외적인 상황 1에 해당하는 경우

원형 연결 리스트 구현: 하나로 묶기



Clinkedlist.c CLinkedlist.h CLinkedlistmain.c

실행결과

1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

1 3 5

원형 연결 리스트 구현: 하나로 묶기



Clinkedlist.c CLinkedlist.h CLinkedlistmain.c

실행결과

1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

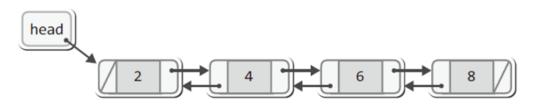
1 3 5

양방향 연결 리스트

(Doubly Linked List)

양방향 연결 리스트의 이해

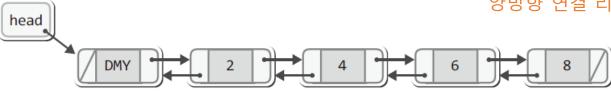




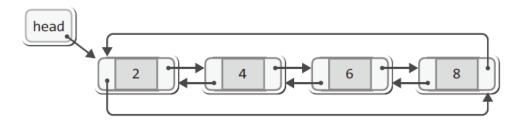
▶ [그림 05-17: 기본적인 양방향 연결 리스트]

```
typedef struct _node
{
    Data data;
    struct _node * next;
    struct _node * prev;
} Node;
```

양방향 연결 리스트를 위한 노드의 표현



▶ [그림 05-18: 더미 노드 양방향 연결 리스트]



▶ [그림 05-19: 원형 연결 기반의 양방향 연결 리스트]

양방향으로 노드를 연결하는 이유!



```
int LNext(List * plist, Data * pdata)
{
    if(plist->tail == NULL)
        return FALSE;

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

    *pdata = plist->cur->data;
    return TRUE;
}
```

```
{
   if(plist->cur->next == NULL)
     return FALSE;

plist->cur = plist->cur->next;
*pdata = plist->cur->data;

return TRUE;
}

단순 연결 리스트와 같이
  before를 유지할 필요가 없다!
```

int LNext(List * plist, Data * pdata)

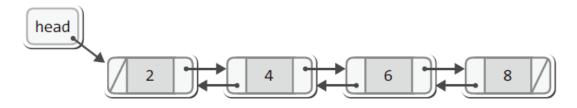
단순 연결 리스트의 LNext

양방향 연결 리스트의 LNext

- √ 오른쪽 노드로의 이동이 용이하다! 양방향으로 이동이 가능하다!
- √ 위의 코드에서 보이듯이 양방향으로 연결한다 하여 더 복잡한 것은 아니다!그렇게 느낀다면 선입견이다.

우리가 구현할 양방향 연결 리스트 모델





- ▶ [그림 05-20: 함께 구현할 양방향 연결 리스트의 구조]
- √ LRemove 함수를 ADT에서 제외시킨다.
- √ 대신에 왼쪽 노드의 데이터를 참조하는 LPrevious 함수를 ADT에 추가시킨다.
- √ 새 노드는 머리에 추가한다.

양방향 연결 리스트의 헤더파일



```
typedef int Data;
typedef struct node
    Data data;
    struct node * next;
    struct node * prev;
} Node;
typedef struct DLinkedList
   Node * head;
   Node * cur;
    int numOfData;
} DBLinkedList;
```

```
typedef DBLinkedList List;

void ListInit(List * plist);
void LInsert(List * plist, Data data);

int LFirst(List * plist, Data * pdata);
int LNext(List * plist, Data * pdata);
int LPrevious(List * plist, Data * pdata);
int LCount(List * plist);
```

LPrevious 함수는 LNext 함수와 반대로 이동한다!

양방향 연결 리스트의 활용의 예



```
int main(void)
{

    // 양방향 연결 리스트의 생성 및 초기화 //////
    List list;
    int data;
    ListInit(&list);

    // 8개의 데이터 저장 //////
    LInsert(&list, 1); LInsert(&list, 2);
    LInsert(&list, 3); LInsert(&list, 4);
    LInsert(&list, 5); LInsert(&list, 6);
    LInsert(&list, 7); LInsert(&list, 8);
```



추가된 이후의 연결 순서

```
head \rightarrow 8 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1
```

```
// 저장된 데이터의 조회 //////
if(LFirst(&list, &data))
   printf("%d ", data);
   // 오른쪽 노드로 이동하며 데이터 조회
   while(LNext(&list, &data))
       printf("%d ", data);
   // 왼쪽 노드로 이동하며 데이터 조회
   while(LPrevious(&list, &data))
       printf("%d ", data);
   printf("\n\n");
return 0;
```

실행결과

8 7 6 5 4 3 2 1 2 3 4 5 6 7 8

연결 리스트의 구현: 리스트의 초기화



```
typedef struct _dbLinkedList
{
    Node * head;
    Node * cur;
    int numOfData;
} DBLinkedList;
```

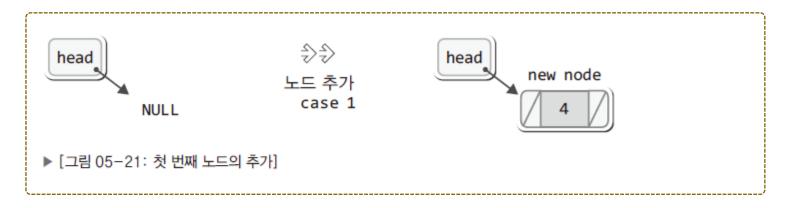
ListInit 함수의 정의에 참조해야 하는 구조체의 정의

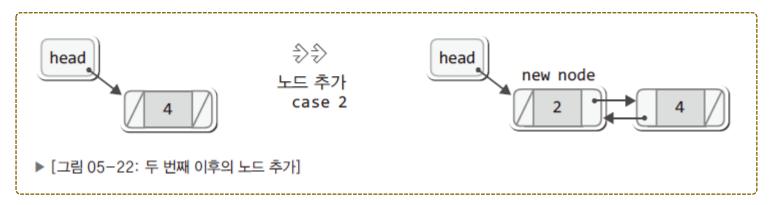
```
void ListInit(List * plist)
{
    plist->head = NULL;
    plist->numOfData = 0;
}
```

멤버 cur은 조회의 과정에서 초기화 되는 멤버이니 head와 numOfData만 초기화 하면 된다.

연결 리스트의 구현: 노드 삽입의 구분







더미 노드를 기반으로 하지 않기 때문에 노드의 삽입 방법은 두 가지로 구분이 된다.

연결 리스트의 구현: 첫 번째 노드의 삽입



첫 번째 노드의 추가 과정만을 담은 결과

```
void LInsert(List * plist, Data data)
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    // 아래 문장에서 plist->head는 NULL이다!
    newNode->next = plist->head;
    newNode->prev = NULL;
    plist->head = newNode;
    (plist->numOfData)++;
```



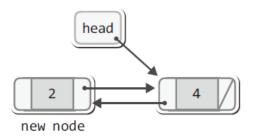
new node

NULL

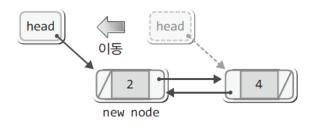
연결 리스트의 구현: 두 번째 이후 노드의 삽입



```
void LInsert(List * plist, Data data)
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = plist->head;
    if(plist->head != NULL)
        plist->head->prev = newNode;
    newNode->prev = NULL;
    plist->head = newNode;
    (plist->numOfData)++;
```



▶ [그림 05-23: 두 번째 노드의 추가과정 1/2]



▶ [그림 05-24: 두 번째 노드의 추가과정 2/2]

첫 번째 노드의 추가 과정에 덧붙여서, if문이 포함하는 문장이 두 번째 이후의 노드 추가 과정에서는 요구된다.

연결 리스트의 구현: 데이터 조회



```
int LFirst(List * plist, Data * pdata)
{
    if(plist->head == NULL)
        return FALSE;

    plist->cur = plist->head;
    *pdata = plist->cur->data;
    return TRUE;
}
```

```
int LNext(List * plist, Data * pdata)
{
    if(plist->cur->next == NULL)
        return FALSE;

    plist->cur = plist->cur->next;
    *pdata = plist->cur->data;
    return TRUE;
}
```

```
int LPrevious(List * plist, Data * pdata)
{
    if(plist->cur->prev == NULL)
        return FALSE;

    plist->cur = plist->cur->prev;
    *pdata = plist->cur->data;
    return TRUE;
}
```

LFirst 함수와 LNext 함수는 사실상 단방향 연결 리스트의 경우와 차이가 없다. 그리고 LPrevious 함수는 LNext 함수와 이동 방향에서만 차이가 난다.

양방향 연결 리스트의 구현을 하나로 묶기



DBLinkedlist.c
DBLinkedlist.h
DBLinkedlistmain.c

실행결과

876543212345678

요약



- 원형 연결 리스트는 머리와 꼬리가 이어진 리스트
 - 머리에 대한 포인터를 저장하지 않고, 꼬리에 대한 포인터를 저장하여 꼬리와(tail) 머리에(tail->next) 쉽게 접근 가능
 - 단순 연결 리스트와 차이점: 꼬리가 가리키는 노드가 삭제될 때 tail을 변경해야 함
- 양방향 연결 리스트는 각 노드가 이전 노드와 다음 노드를 가리키는 포인터를 가짐
 - 단순 연결 리스트와 차이점: 노드가 추가될 때 기존 머리 노드의 prev 포인터를 바꿔 줘야 함

출석 인정을 위한 보고서 작성



- A4 반 장 이상으로 아래 질문에 답한 후 포털에 있는 과제 제출 란에 제출
- 이중 연결 리스트에서 삭제는 어떻게 구현하면 될 지 LRemove 함수를 의사 코드로 표현해 보세요 (가능하다면, C 코드도 괜찮음)
 - head 포인터가 가리키고 있는 노드가 지워질 때 head 포인터를 바꿔야 함에 주의
 - 삭제 후 앞, 뒤 노드를 서로 연결해야 함에 주의