List

Data Structures and Algorithms

목차

- 추상 자료형 (ADT)
- 배열을 이용한 리스트의 구현
- 연결 리스트의 개념적 이해
- 연결 리스트의 ADT 구현
- 원형 연결 리스트
- 이중 연결 리스트

추상 자료형
Abstract Data Type

추상 자료형(ADT)의 이해

- 추상 자료형이란?
  - 자료형에 적용 가능한 연산 또는 기능의 명세서

- 예

· 카드의 삽입
· 카드의 추출(카드를 빼냄)
· 동전의 삽입
· 동전의 추출(동전을 빼냄)
· 지폐의 삽입
· 지폐의 추출(지폐를 빼냄)

지갑을 의미하는 구조체 Wallet의 정의

- 자료형과 관련이 있는 연산을 함께 정의해야 완전한 자료형

```
typedef struct
{
    int coin;       // 동전의 수
    int bill;       // 지폐의 수
} Wallet;
```

자료형
Wallet의 정의

+

자료형 Wallet 의
연산의 정의

```
int TakeOutMoney(Wallet * pw, int coin, int bill);  // 돈 꺼내는 연산
void PutMoney(Wallet * pw, int coin, int bill);     // 돈 넣는 연산
```

ADT와 자료구조 학습 순서

- 자료구조의 ADT를 정의한다.
- ADT를 근거로 자료구조를 활용하는 main 함수를 정의
- ADT를 근거로 자료구조를 구현

- 올바른 ADT의 정의
  - 자료구조의 내부 구현이 감춰지도록 정의
- 자료구조를 쉽게 이해하려면 자료구조를 활용하는 main함수를 먼저 작성

연습

- 2차원 좌표계의 한 점의 ADT를 정의해보자
  - 사용 가능한 연산은 각 x, y 좌표의 +, - 계산으로 한정함

배열을 이용한 리스트의 구현

리스트 구분 및 특징

- 구분
  - 순차 리스트: 배열을 기반으로 구현된 리스트
  - 연결 리스트: 메모리의 동적 할당을 기반으로 구현된 리스트

- 특징
  - 저장 형태: 데이터를 나란히(하나의 열로) 저장
  - 저장 특성: 중복이 되는 데이터의 저장을 허용

리스트 자료구조의 ADT

```
typedef int LData;
```

리스트의 초기화

- `void ListInit(List * plist);`
  - 초기화할 리스트의 주소 값을 인자로 전달한다.
  - 리스트 생성 후 제일 먼저 호출되어야 하는 함수이다.

데이터 저장

- `void LInsert(List * plist, LData data);`
  - 리스트에 데이터를 저장한다. 매개변수 data에 전달된 값을 저장한다.

저장된 데이터의 탐색 및 탐색 초기화

- `int LFirst(List * plist, LData * pdata);`
  - 첫 번째 데이터가 pdata가 가리키는 메모리에 저장된다.
  - 데이터의 참조를 위한 초기화가 진행된다.
  - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

리스트 자료구조의 ADT

다음 데이터의 참조(반환)을 목적으로 호출

- `int LNext(List * plist, LData * pdata);`
  - 참조된 데이터의 다음 데이터가 pdata가 가리키는 메모리에 저장된다.
  - 순차적인 참조를 위해서 반복 호출이 가능하다.
  - 참조를 새로 시작하려면 먼저 LFirst 함수를 호출해야 한다.
  - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

바로 이전에 참조(반환)이 이뤄진 데이터의 삭제

- `LData LRemove(List * plist);`
  - LFirst 또는 LNext 함수의 마지막 반환 데이터를 삭제한다.
  - 삭제된 데이터는 반환된다.
  - 마지막 반환 데이터를 삭제하므로 연이은 반복 호출을 허용하지 않는다.

현재 저장되어 있는 데이터의 수를 반환

- `int LCount(List * plist);`
  - 리스트에 저장되어 있는 데이터의 수를 반환한다.

리스트의 ADT를 기반으로 한 main 함수

- 실행을 위해 필요한 파일들
  - ArrayList.h 리스트 자료구조의 헤더파일
  - ArrayList.c 리스트 자료구조의 소스파일
  - ListMain.c 리스트 관련 main 함수가 담긴 소스파일

리스트의 초기화와 데이터 저장 과정

리스트의 초기화

```
int main(void)
{
     List list;   // 리스트의 생성
     . . . .
     ListInit(&list);     // 리스트의 초기화
     . . . .
}
```

초기화된 리스트에 데이터 저장

```
int main(void)
{
     . . . .
     LInsert(&list, 11);      // 리스트에 11을 저장
     LInsert(&list, 22);      // 리스트에 22를 저장
     LInsert(&list, 33);      // 리스트에 33을 저장
     . . . .
}
```

리스트의 노드 순회 과정

- LFirst: 리스트 순회의 시작

```
int main(void)
{
    . . . .
    if( LFirst(&list, &data) )     // 첫 데이터 변수 data에 저장
    {
        printf("%d ", data);
        while( LNext(&list, &data) ) // 다음 데이터 변수 data에 저장
            printf("%d ", data);
    }
    . . . .
}
```

- 데이터 참조 일련의 과정

LFirst → LNext → LNext → LNext → LNext . . . .

리스트의 노드 삭제

- LRemove 함수는 연이은 호출을 허용하지 않음!

```
int main(void)
{
    . . . .
    if( LFirst(&list, &data) )
    {
        if(data == 22)
            LRemove(&list);     // LFirst 함수를 통해 참조한 데이터 삭제!

        while( LNext(&list, &data) )
        {
            if(data == 22)
                LRemove(&list); // LNext 함수를 통해 참조한 데이터 삭제!
        }
    }
    . . . .
}
```

배열 기반 리스트의 헤더파일 정의 1

- 리스트를 배열 기반으로 구현하기 위한 선언을 포함

```
#ifndef __ARRAY_LIST_H__
#define __ARRAY_LIST_H__

#define TRUE 1    // ‘참’ 매크로 정의
#define FALSE 0.  // ‘거짓’ 매크로 정의

/*** ArrayList의 정의 시작 ****/
#define LIST_LEN 100
typedef int LData; //저장할 대상의 자료형을 변경을 위한 typedef 선언

typedef struct __ArrayList  // 배열 기반 리스트를 의미하는 구조체
{
    LData arr[LIST_LEN];    // 리스트의 저장소 배열
    int numOfData;          // 저장된 데이터의 수
    int curPosition;        // 현재 참조 데이터의 위치
} ArrayList;
```

배열 기반 리스트의 헤더파일 정의 2

- 배열 기반 리스트 ADT의 함수들

```
/*** ArrayList와 관련된 연산들 ****/
typedef ArrayList List; // 리스트의 변경을 용이하게 하기 위한 typedef 선언

void ListInit(List * plist);              // 초기화
void LInsert(List * plist, LData data);   // 데이터 저장

int LFirst(List * plist, LData * pdata); // 첫 데이터 참조
int LNext(List * plist, LData * pdata);  // 두 번째 이후 데이터 참조

LData LRemove(List * plist);              // 참조한 데이터 삭제
int LCount(List * plist);                 // 저장된 데이터의 수 반환

#endif
```

배열 기반 리스트의 초기화

- 초기화 대상은 구조체 변수의 멤버이기 때문에 구조체의 정의를 기반으로 함수를 작성

```
void ListInit(List * plist)
{
    (plist->numOfData) = 0;      // 데이터 0개가 리스트에 저장됨
    (plist->curPosition) = -1;   // -1: 참조하지 않았음
}
```

```
#define LIST_LEN 100
typedef int LData; //저장할 대상의 자료형을 변경을 위한 typedef 선언

typedef struct __ArrayList  // 배열 기반 리스트를 의미하는 구조체
{
    LData arr[LIST_LEN];    // 리스트의 저장소 배열
    int numOfData;          // 저장된 데이터의 수
    int curPosition;        // 현재 참조 데이터의 위치
} ArrayList;
```

배열 기반 리스트의 삽입

- 배열에 데이터 저장

```
void LInsert(List * plist, LData data)
{
    if(plist->numOfData > LIST_LEN)     // 더 이상 저장할 공간이 없다면
    {
        printf("저장이 불가능합니다.\n");
        return;
    }
    plist->arr[plist->numOfData] = data;  // 데이터 저장
    (plist->numOfData)++;                 // 저장된 데이터 수 증가
}
```

```
#define LIST_LEN 100
typedef int LData; //저장할 대상의 자료형을 변경을 위한 typedef 선언

typedef struct __ArrayList  // 배열 기반 리스트를 의미하는 구조체
{
    LData arr[LIST_LEN];    // 리스트의 저장소 배열
    int numOfData;          // 저장된 데이터의 수
    int curPosition;        // 현재 참조 데이터의 위치
} ArrayList;
```

배열 기반 리스트 탐색

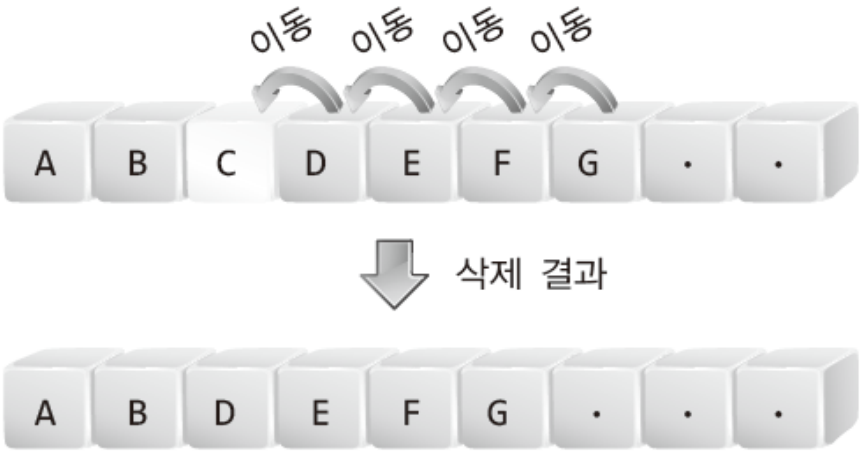
```
int LFirst(List * plist, LData * pdata)  // 초기화, 첫 데이터 참조
{
    if(plist->numOfData == 0)  // 저장된 데이터가 하나도 없다면
        return FALSE;

    (plist->curPosition) = 0;  // 참조 위치 초기화; 첫 데이터 참조를 의미
    *pdata = plist->arr[0];    // pdata가 가리키는 공간에 데이터 저장
    return TRUE;
}

int LNext(List * plist, LData * pdata) // 다음 데이터 참조
{   // 더 이상 참조할 데이터가 없다면
    if(plist->curPosition >= (plist->numOfData)-1)
        return FALSE;

    (plist->curPosition)++;
    *pdata = plist->arr[plist->curPosition];
    return TRUE;
}
```

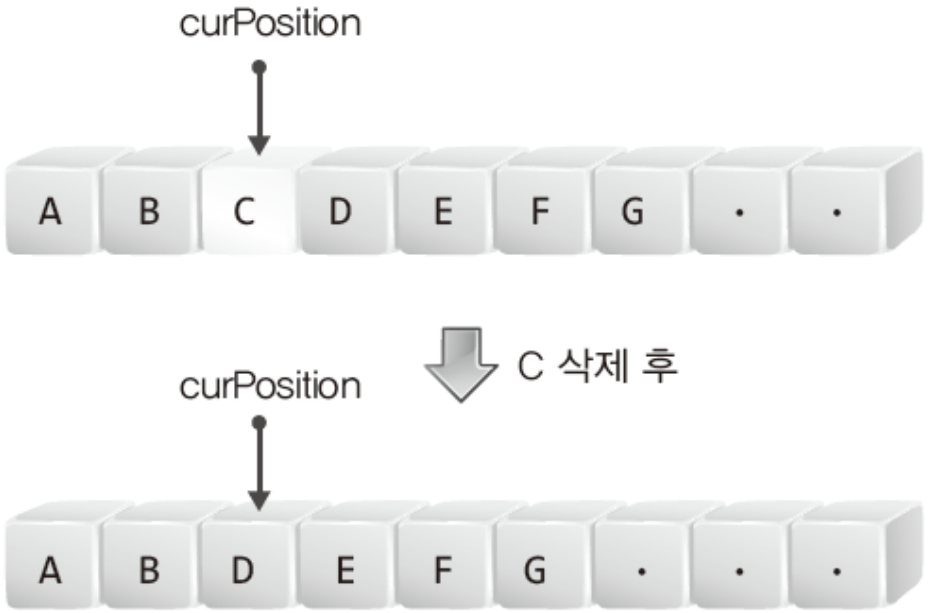
데이터 반환은 매개변수로

함수의 반환은 연산 성공여부

배열 기반 리스트의 삭제

- 삭제 기본 모델





배열 기반 리스트의 삭제

- 삭제된 데이터는 리턴 값으로 전달

```
LData LRemove(List * plist)
{
    int rpos = plist->curPosition;   // 삭제할 데이터의 인덱스 값 참조
    int num = plist->numOfData;
    int i;
    LData rdata = plist->arr[rpos];  // 삭제할 데이터를 임시 저장

    for(i=rpos; i<num-1; i++)
        plist->arr[i] = plist->arr[i+1];

    (plist->numOfData)--;            // 데이터의 수 감소
    (plist->curPosition)--;          // 참조 위치 하나 되돌림
    return rdata;                    // 삭제된 데이터의 반환
}
```

리스트에 구조체 변수 저장하기 1 : Point.h

- Point.h: 리스트에 저장할 Point 구조체 변수의 헤더

```
typedef struct _point
{
    int xpos;
    int ypos;
} Point;

// Point 변수의 xpos, ypos 값 설정
void SetPointPos(Point * ppos, int xpos, int ypos);


// Point 변수의 xpos, ypos 정보 출력
void ShowPointPos(Point * ppos);


// 두 Point 변수의 비교
int PointComp(Point * pos1, Point * pos2);
```

리스트에 구조체 변수 저장하기 2 : Point.c

```
void SetPointPos(Point * ppos, int xpos, int ypos)
{
    ppos->xpos = xpos;
    ppos->ypos = ypos;
}

void ShowPointPos(Point * ppos)
{
    printf("[%d, %d] \n", ppos->xpos, ppos->ypos);
}

int PointComp(Point * pos1, Point * pos2)
{
    if(pos1->xpos == pos2->xpos && pos1->ypos == pos2->ypos)
        return 0;
    else if(pos1->xpos == pos2->xpos)
        return 1;
    else if(pos1->ypos == pos2->ypos)
        return 2;
    else
        return -1;
}
```

리스트에 구조체 변수 저장하기 3

- ArrayList.h

배열

```
3
4  #define TRUE 1
5  #define FALSE  0
6
7  /*** ArrayList의 정의 ****/
8  #define LIST_LEN 100
9  typedef int LData;
10
11 typedef struct __ArrayList
```

구조체

```
4  #include "Point.h"      // 첫 번째 변경
5
6  #define TRUE 1
7  #define FALSE  0
8
9  #define LIST_LEN 100
10
11 // typedef int LData;
12 typedef Point * LData;      // 두 번째 변경
```

- 실행을 위한 파일 구성
  - Point.h, Point.c
  - ArrayList.h, ArrayList.c
  - PointListMain.c

구조체 Point를 위한 파일
배열 기반 리스트
구조체 Point의 변수 저장

PointListMain.c : 초기화 및 저장

```
int main(void)
{
	List list;
	Point compPos;
	Point * ppos;

	ListInit(&list);

	/*** 4개의 데이터 저장 ***/
	ppos = (Point*)malloc(sizeof(Point));
	SetPointPos(ppos, 2, 1);
	LInsert(&list, ppos);

	ppos = (Point*)malloc(sizeof(Point));
	SetPointPos(ppos, 2, 2);
	LInsert(&list, ppos);
	. . . . . .
}
```

PointListMain.c : 참조 및 조회

```
int main(void)
{
     . . . . . .
     /*** 저장된 데이터의 출력 ***/
     printf("현재 데이터의 수: %d \n", LCount(&list));

     if(LFirst(&list, &ppos))
     {
          ShowPointPos(ppos);

          while(LNext(&list, &ppos))
               ShowPointPos(ppos);
     }
     printf("\n");
     . . . . . .
}
```

PointListMain.c : 삭제

```
int main(void)
{
    . . . . . .
    /*** xpos가 2인 모든 데이터 삭제 ***/
    compPos.xpos=2;
    compPos.ypos=0;

    if(LFirst(&list, &ppos)) {
        if(PointComp(ppos, &compPos)==1) {
            ppos=LRemove(&list);
            free(ppos);
        }
        while(LNext(&list, &ppos)) {
            if(PointComp(ppos, &compPos)==1) {
                ppos=LRemove(&list);
                free(ppos);
            }
    }
    . . . . . .
}
```

배열 기반 리스트의 장점과 단점

- 배열 기반 리스트의 단점
  - 배열 길이는 초기에 결정되며, 변경 불가능
  - 삭제 과정에서 데이터 이동(복사)가 매우 빈번함

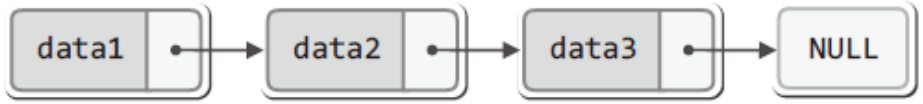
- 배열 기반 리스트의 장점
  - 쉬운 데이터 참조: 인덱스 값으로 어디든 한 번에 참조 가능

연결 리스트의 개념적 이해

Linked의 의미

```
typedef struct _node
{
    int data;          // 데이터를 담을 공간
    struct _node * next;    // 연결의 도구!
} Node;
```

▶ [그림 04-1: 노드의 표현]

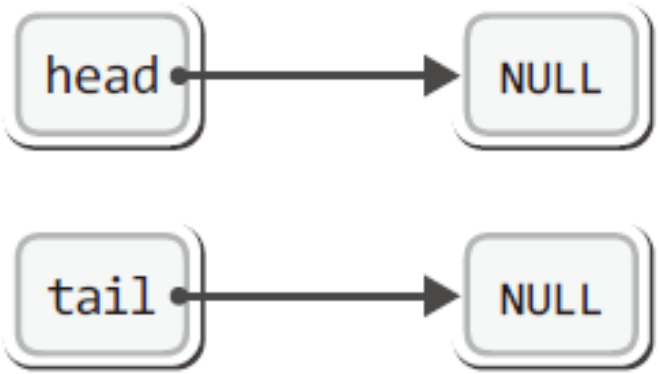


▶ [그림 04-2: 노드의 연결]

LinkedRead.c: 초기화

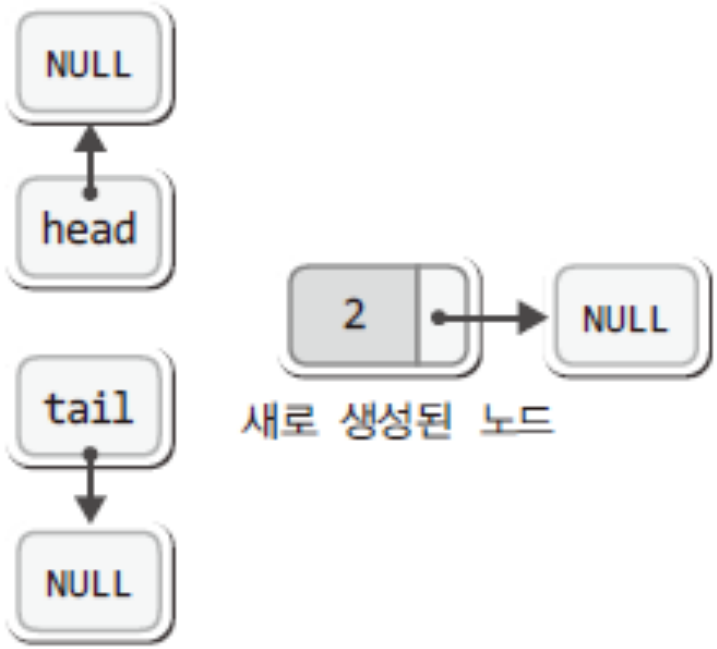
```
typedef struct _node
{
    int data;
    struct _node * next;
} Node;

int main(void)
{
    Node * head = NULL;
    Node * tail = NULL;
    Node * cur = NULL;

    Node * newNode = NULL;
    int readData;
    . . . .
}
```

- head, tail, cur이 연결 리스트의 핵심!
- head와 tail은 연결을 추가 및 유지하기 위한것
- cur은 참조 및 조회를 위한것



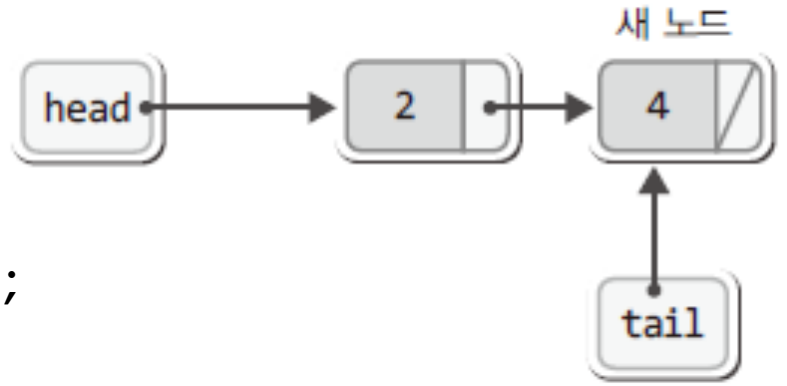
LinkedRead.c: 삽입 1

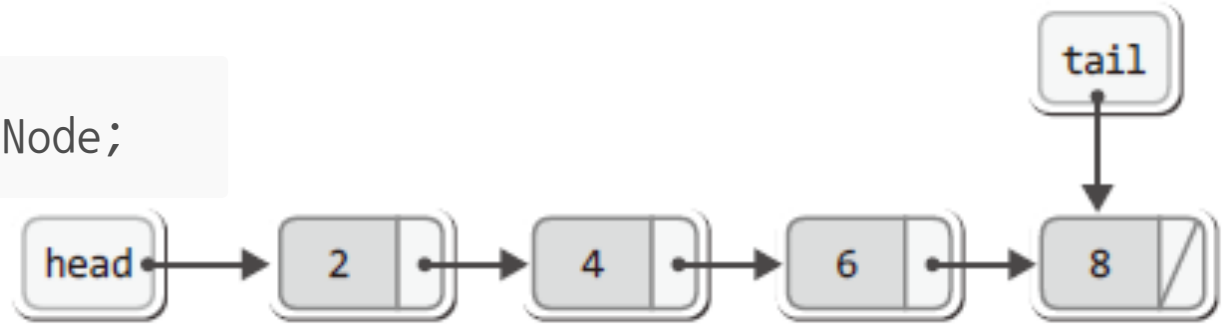
```
while(1)
{
    printf("자연수 입력: ");
    scanf("%d", &readData);
    if(readData < 1)
        break;

    // 노드의 추가과정
    newNode = (Node*)malloc(sizeof(Node));
    newNode->data = readData;
    newNode->next = NULL;

    if(head == NULL)
        head = newNode;
    else
        tail->next = newNode;

    tail = newNode;
 }
```



LinkedRead.c: 삽입 2

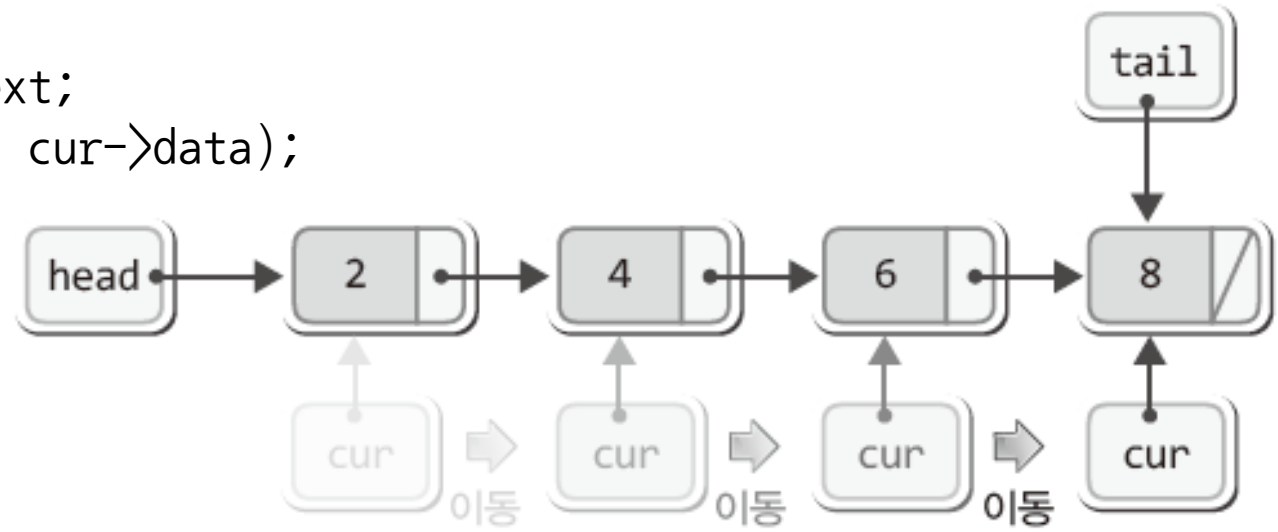
```
while(1)
{
    printf("자연수 입력: ");
    scanf("%d", &readData);
    if(readData < 1)
        break;

    // 노드의 추가과정
    newNode = (Node*)malloc(sizeof(Node));
    newNode->data = readData;
    newNode->next = NULL;

    if(head == NULL)
        head = newNode;
    else
        tail->next = newNode;

    tail = newNode;
}
```





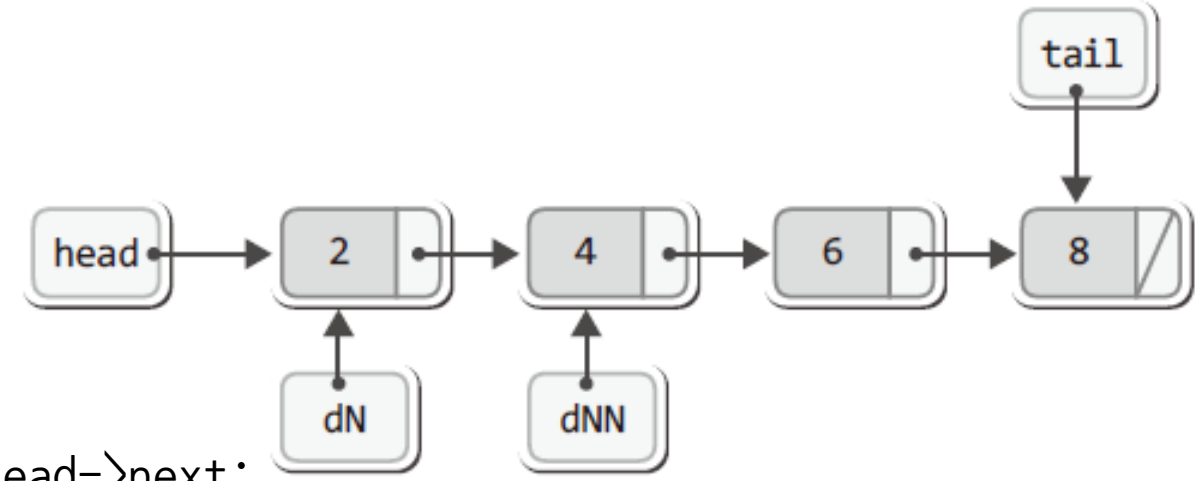
LinkedRead.c: 데이터 순회

```
if(head == NULL)
{
     printf("저장된 자연수가 존재하지 않습니다. \n");
}
else
{
     cur = head;
     printf("%d ", cur->data);
     while(cur->next != NULL)
     {
          cur = cur->next;
          printf("%d ", cur->data);
     }
}
```



LinkedRead.c: 데이터 삭제

```
if(head == NULL)
{
     return 0;
}
else
{
     Node * delNode = head;
     Node * delNextNode = head->next;
     printf("%d을 삭제\n", head->data);
     free(delNode);
     while(delNextNode != NULL)
     {
          delNode = delNextNode;
          delNextNode = delNextNode->next;
          printf("%d을 삭제\n", delNode->data);
          free(delNode);
     }
}
```



연결 리스트의 ADT 구현

정렬 기능 추가된 연결 리스트의 ADT

리스트 자료구조의 ADT

리스트의 초기화

```
typedef int LData;
```

- `void ListInit(List * plist);`
  - 초기화할 리스트의 주소 값을 인자로 전달한다.
  - 리스트 생성 후 제일 먼저 호출되어야 하는 함수이다.

데이터 저장

- `void LInsert(List * plist, LData data);`
  - 리스트에 데이터를 저장한다. 매개변수 data에 전달된 값을 저장한다.

저장된 데이터의 탐색 및 탐색 초기화

- `int LFirst(List * plist, LData * pdata);`
  - 첫 번째 데이터가 pdata가 가리키는 메모리에 저장된다.
  - 데이터의 참조를 위한 초기화가 진행된다.
  - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

리스트 자료구조의 ADT

다음 데이터의 참조(반환)을 목적으로 호출

- `int LNext(List * plist, LData * pdata);`
  - 참조된 데이터의 다음 데이터가 pdata가 가리키는 메모리에 저장된다.
  - 순차적인 참조를 위해서 반복 호출이 가능하다.
  - 참조를 새로 시작하려면 먼저 LFirst 함수를 호출해야 한다.
  - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

바로 이전에 참조(반환)이 이뤄진 데이터의 삭제

- `LData LRemove(List * plist);`
  - LFirst 또는 LNext 함수의 마지막 반환 데이터를 삭제한다.
  - 삭제된 데이터는 반환된다.
  - 마지막 반환 데이터를 삭제하므로 연이은 반복 호출을 허용하지 않는다.

현재 저장되어 있는 데이터의 수를 반환

- `int LCount(List * plist);`
  - 리스트에 저장되어 있는 데이터의 수를 반환한다.

- `void SetSortRule(List * plist, int (*comp)(LData d1, LData d2));`
  - 리스트에 정렬의 기준이 되는 함수를 등록한다.

함수 포인터가 사용됨

새로운 노드의 추가 위치

- 새 노드를 연결 리스트의 머리에 추가
  - 장점 : 포인터 변수 tail이 불필요
  - 단점: 저장된 순서를 유지하지 않음

- 새 노드를 연결 리스트의 꼬리에 추가
  - 장점: 저장된 순서가 유지
  - 단점: 포인터 변수 tail이 필요

SetSortRule 함수 선언

```
void SetSortRule ( List * plist, int (*comp)(LData d1, LData d2) );
```

- 반환형이 int이고, int (*comp)(LData d1, LData d2)
- LData형 인자를 두 개 전달받는, int (*comp)(LData d1, LData d2)
- 함수의 주소 값을 전달해라! int (*comp)(LData d1, LData d2)

인자로 전달이 가능한 함수의 예

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

정렬의 기준을 결정하는 함수에 대한 약속!

약속에 근거한 함수 정의의 예

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

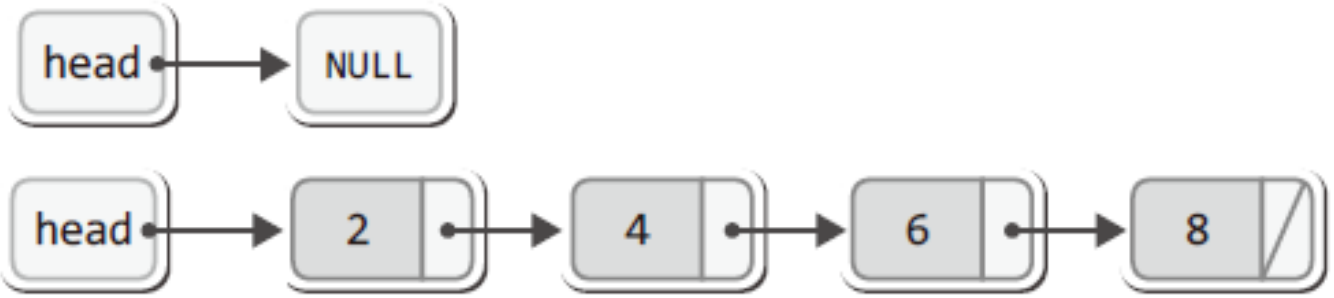
함수 호출 예

```
int cr = WhoIsPrecede(D1, D2);
```

우리가 구현할 더미 노드 기반 연결 리스트

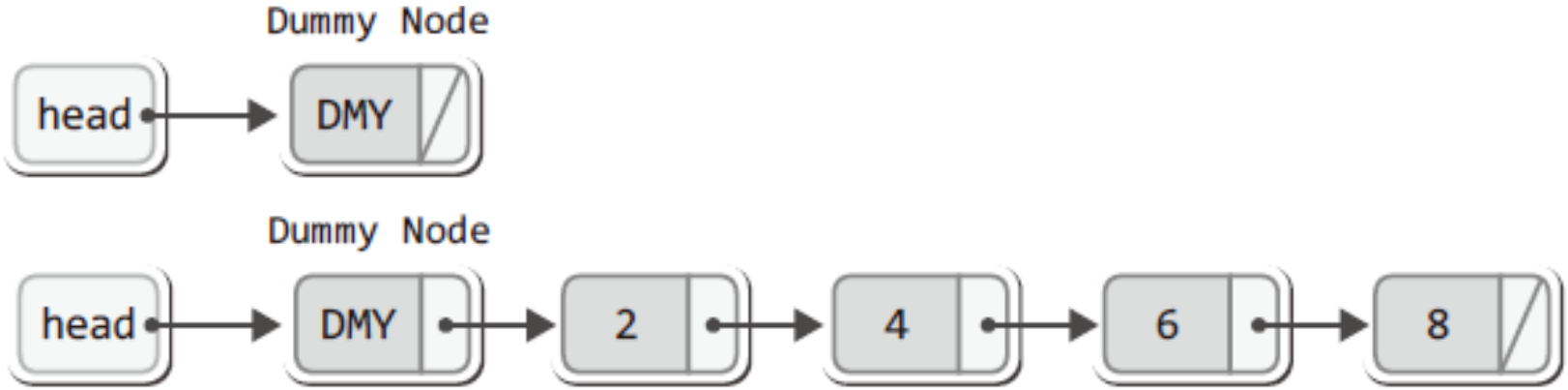
- 머리에 새 노드를 추가하되 더미 노드 없는 경우

  첫 번째 노드와 두 번째 이후의 노드 추가 및 삭제 방식이 다를 수 있음



- 머리에 새 노드를 추가하되 더미 노드 있는 경우

  노드의 추가 및 삭제 방식이 항상 일정



LinkedRead.c와 문제 04-2의 답안인 DLinkedRead.c를 비교

정렬 기능 추가된 연결 리스트의 구조체

- 노드의 구조체 표현

```
typedef struct _node // typedef int LData;
{
    LData data;      // 필요한 변수들은 모두 구조체로 묶어야 함
    struct _node * next;
} Node;
```

- 연결 리스트의 구조체 표현

```
typedef struct _linkedList
{
    Node * head;       // 더미 노드를 가리키는 멤버
    Node * cur;        // 참조 및 삭제를 돕는 멤버
    Node * before;     // 삭제를 돕는 멤버
    int numOfData;     // 저장된 데이터의 수를 기록하기 위한 멤버
    int (*comp)(LData d1, LData d2);  // 정렬의 기준을 등록하기 위한 멤버
} LinkedList;
```

정렬 기능 추가된 연결 리스트 헤더파일: DLinkedList.h

```
#define TRUE  1
#define FALSE 0

typedef int LData;

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

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

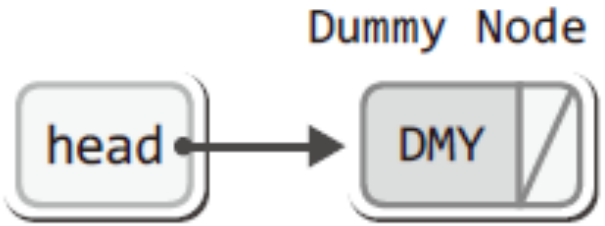
```
typedef LinkedList List;

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

int LFirst(List * plist, LData * pdata);
int LNext(List * plist, LData * pdata);

LData LRemove(List * plist);
int LCount(List * plist);

void SetSortRule(List * plist,
      int (*comp)(LData d1, LData d2));
```

더미 노드 연결 리스트 구현: 초기화

- 초기화 함수

```
void ListInit(List * plist)
{
    plist->head = (Node*)malloc(sizeof(Node)); // 더미노드의 생성
    plist->head->next = NULL;
    plist->comp = NULL;
    plist->numOfData = 0;
}
```

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



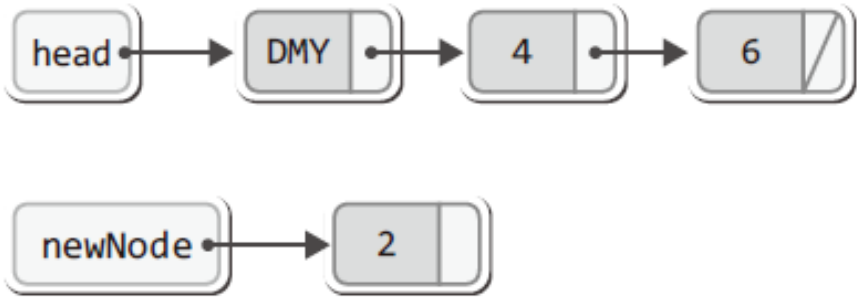
더미 노드 연결 리스트 구현: 삽입1

```
void LInsert(List * plist, LData data)
{
    if(plist->comp == NULL)     // 정렬기준이 없다면
        FInsert(plist, data);   // 머리에 노드를 추가
    else                        // 정렬 기준이 있다면
        SInsert(plist, data);   // 정렬기준에 따라 노드 추가
}

void FInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node)); // 새 노드 생성
    newNode->data = data;                 // 새 노드에 데이터 저장

    newNode->next = plist->head->next; // 새 노드가 다른 노드를 가리키게 함
    plist->head->next = newNode;       // 더미 노드가 새 노드를 가리킴

    (plist->numOfData)++;                 // 저장된 노드의 수를 하나 증가
}
```

더미 노드 연결 리스트 구현: 삽입2

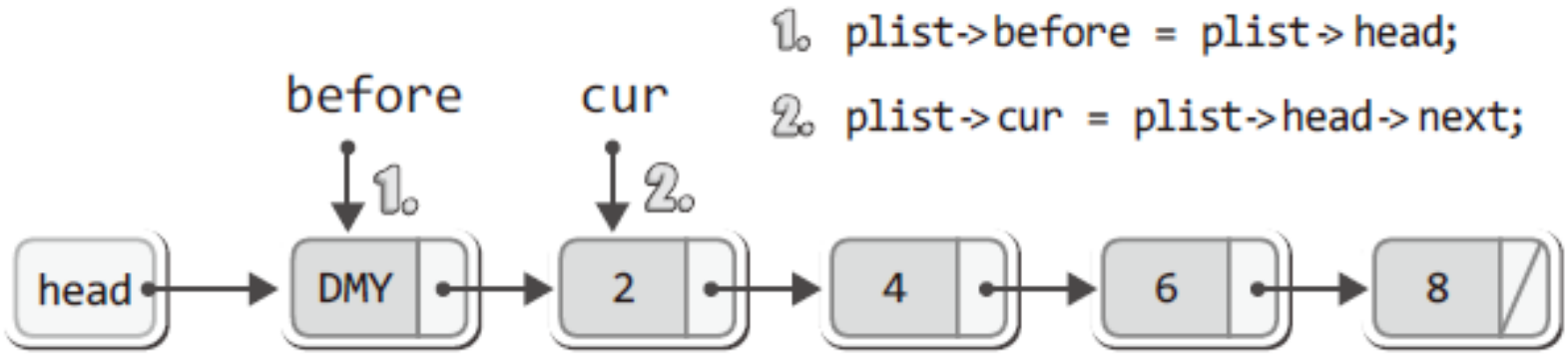
더미 노드를 사용하면 모든 경우에 동일한 삽입과정을 거침

```
void FInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node)); // 새 노드 생성
    newNode->data = data;                         // 새 노드에 데이터 저장

    newNode->next = plist->head->next; // 새 노드가 다른 노드를 가리키게 함
    plist->head->next = newNode;       // 더미 노드가 새 노드를 가리킴

    (plist->numOfData)++;              // 저장된 노드의 수를 하나 증가
}
```



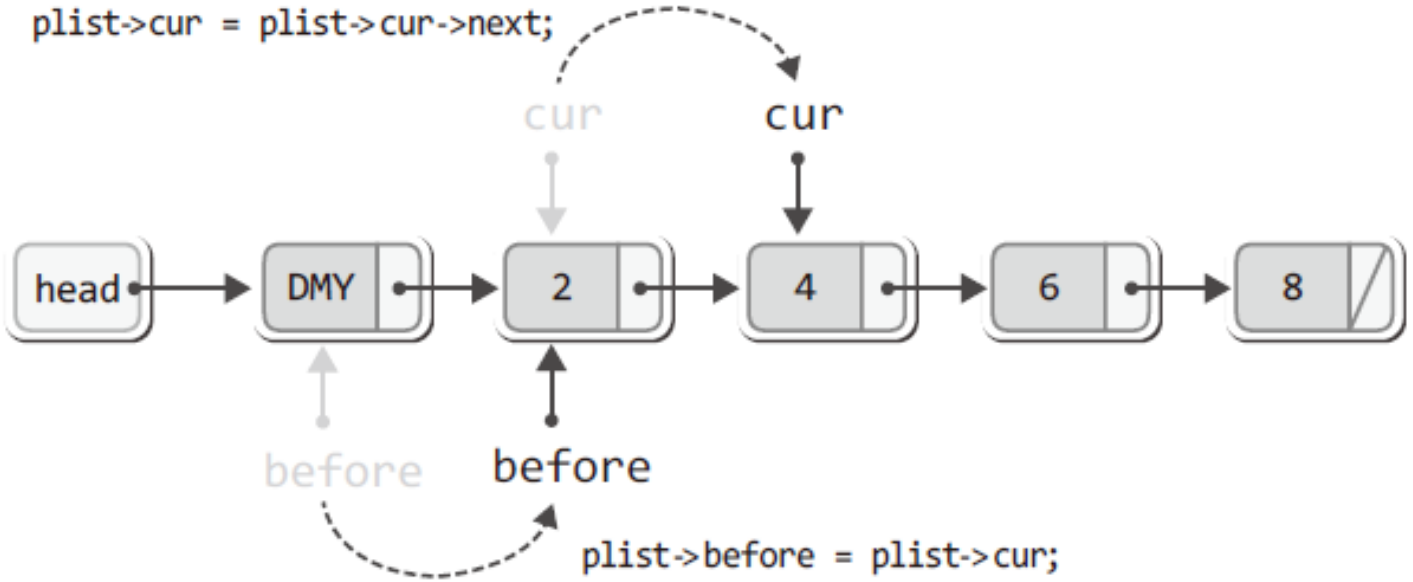
더미 노드 연결 리스트 구현: 참조1

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

    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: 참조2

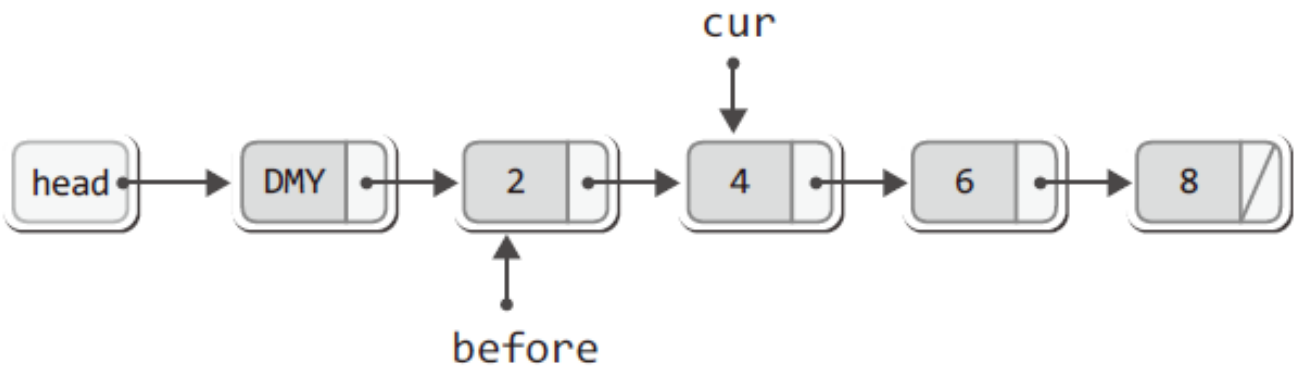
```
int LNext(List * plist, LData * pdata)
{
    if(plist->cur->next == NULL)      // 더미 노드가 NULL을 가리킨다면,
        return FALSE;                 // 반환할 데이터가 없음

    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
}
```



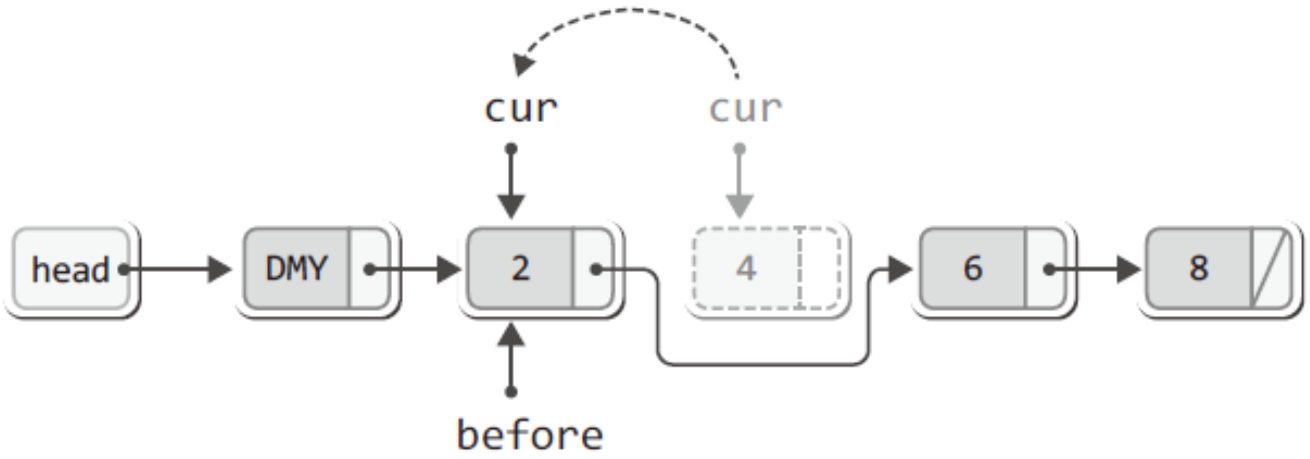
더미 노드 연결 리스트 구현: 삭제1

- cur은 삭제 후 재조정의 과정을 거쳐야 함
- before는 LFirst 또는 LNext 호출 시 재설정 됨

- 삭제 전



- 삭제 후

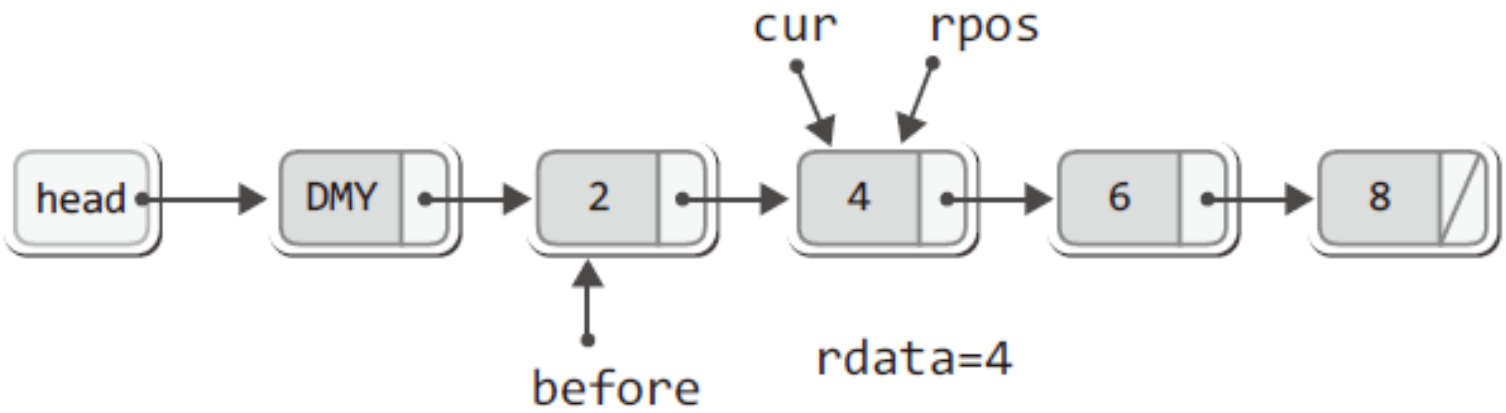


더미 노드 연결 리스트 구현: 삭제2

```
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;
}
```

Dlinkedlist.c
DLinkedlist.h
DLinkedlistmain.c

확인하기



연결 리스트의 정렬 삽입의 구현

정렬 기준 설정

- 단순 연결 리스트의 정렬 관련 요소 세 가지

1

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

SetSortRule 함수 정의에
정렬기준 함수 포함

2

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

SetSortRule 함수를 통해 전달된
함수의 리턴 정보 저장을 위한
LinkedList의 멤버 comp

3

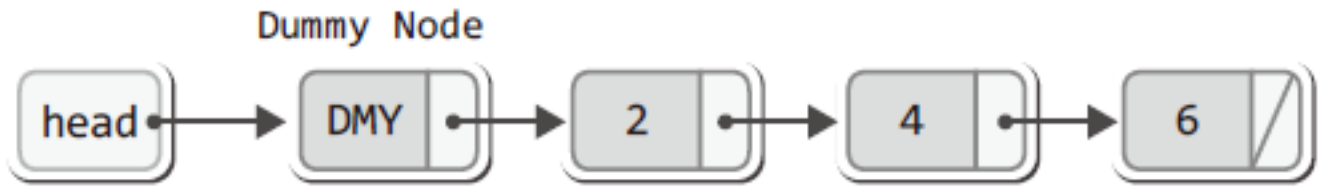
```
void LInsert(List * plist, LData data)
{
    if(plist->comp == NULL)
        FInsert(plist, data);
    else
        SInsert(plist, data);
 }
```

Sinsert 함수가 comp의 정렬기준
대로 데이터를 저장

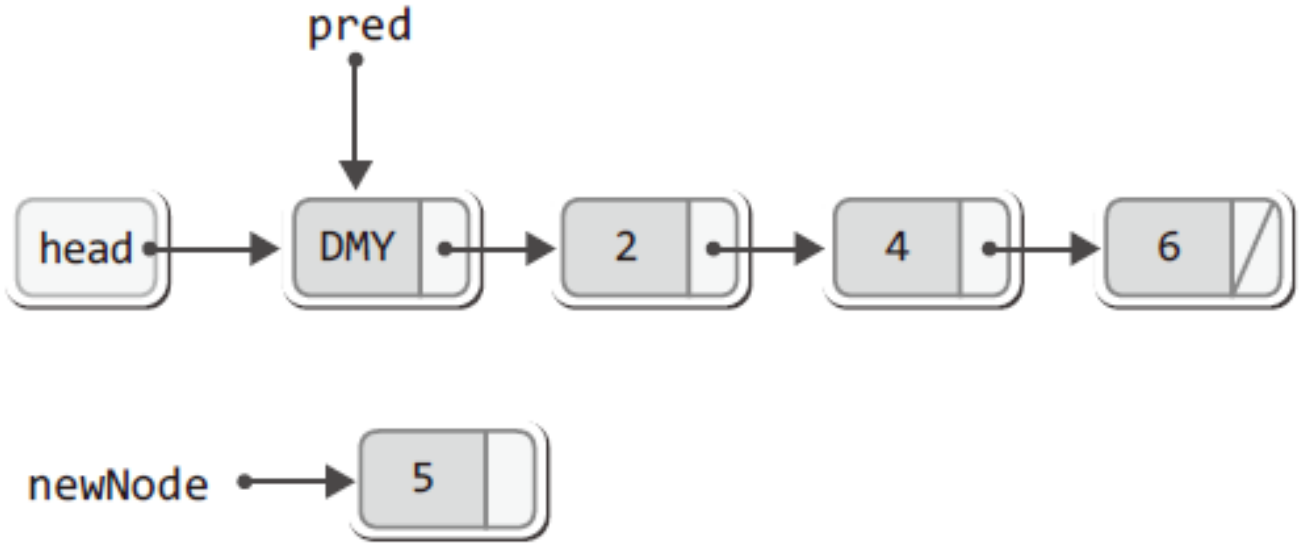
SInsert 함수

```
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)++;
}
```

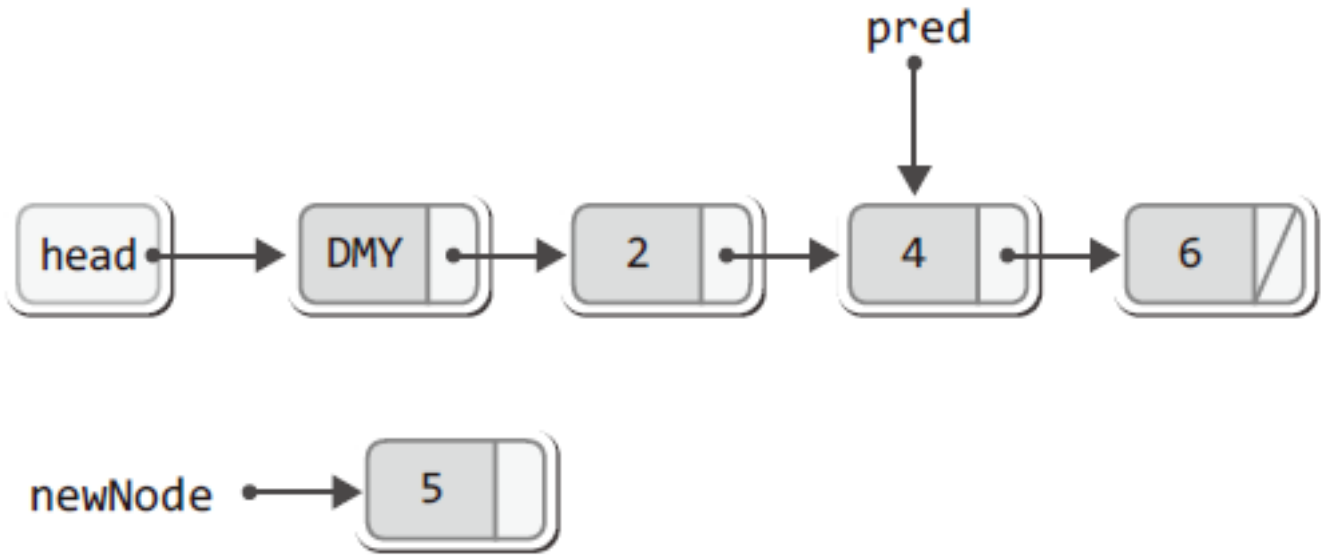
- 다음 상황에서 `SInsert(&slist, 5);` 실행



▶ [그림 04-30: 값의 대소가 정렬의 기준인 연결 리스트]

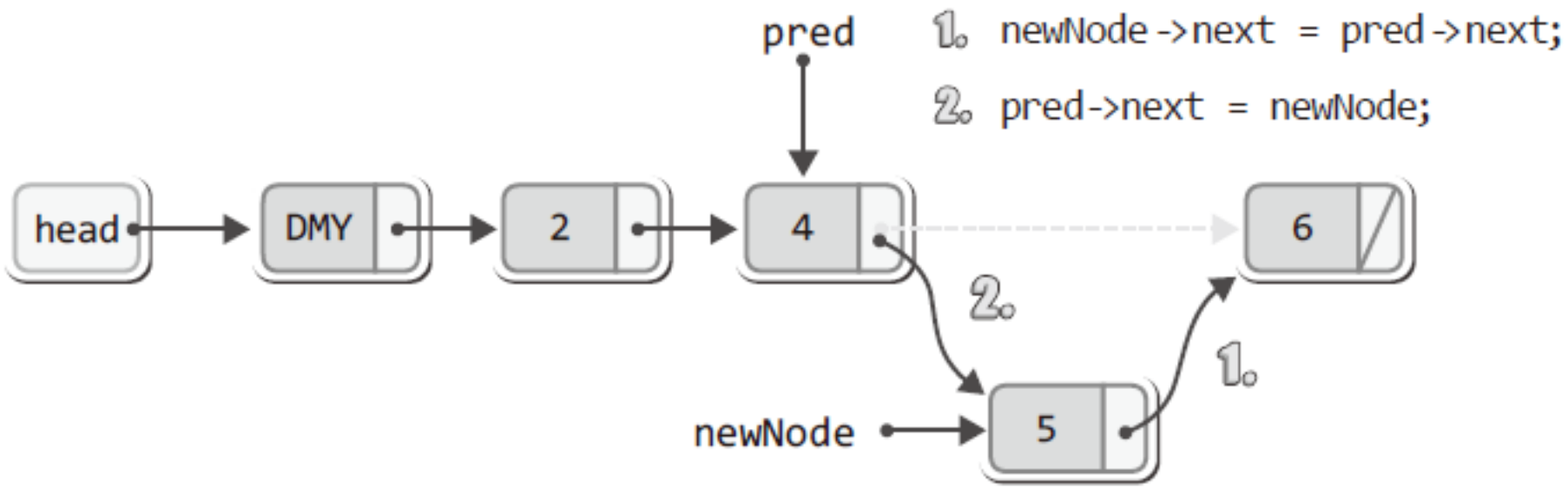


▶ [그림 04-31: SInsert 함수에서의 초기화]



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

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



▶ [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

- 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

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

정렬의 기준을 설정하기 위한 함수의 정의

- 함수의 정의 기준
  - 두 개의 인자를 전달받도록 함수를 정의
  - 첫 번째 인자의 정렬 우선순위가 높으면 0을, 그렇지 않으면 1을 반환

- 오름차순 정렬을 위한 함수의 정의

DLinkedlist.c
DLinkedlist.h
DLinkedlistSortmain.c

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

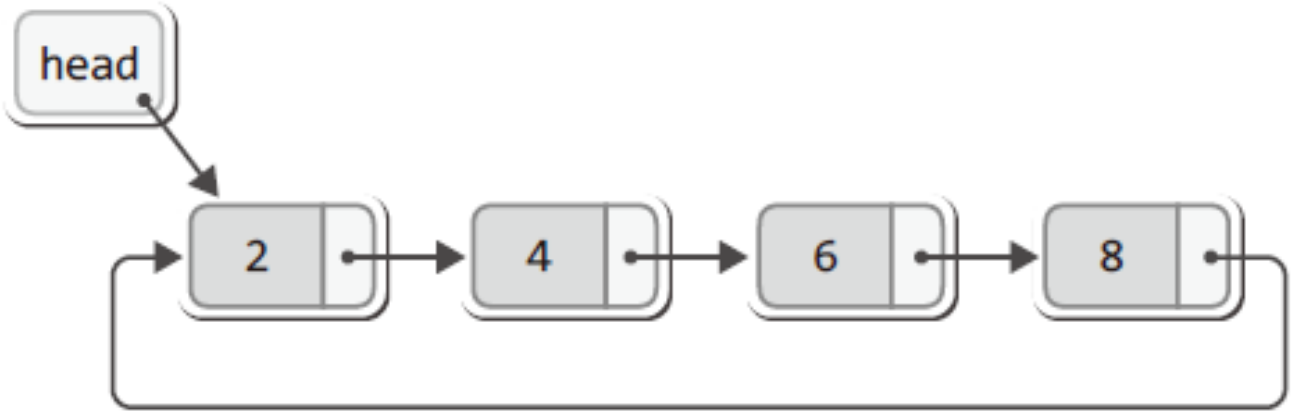
원형 연결 리스트

원형 연결 리스트의 이해

- 단순 연결 리스트의 마지막 노드는 NULL을 가리킴

▶ [그림 05-1 : 단순 연결 리스트]

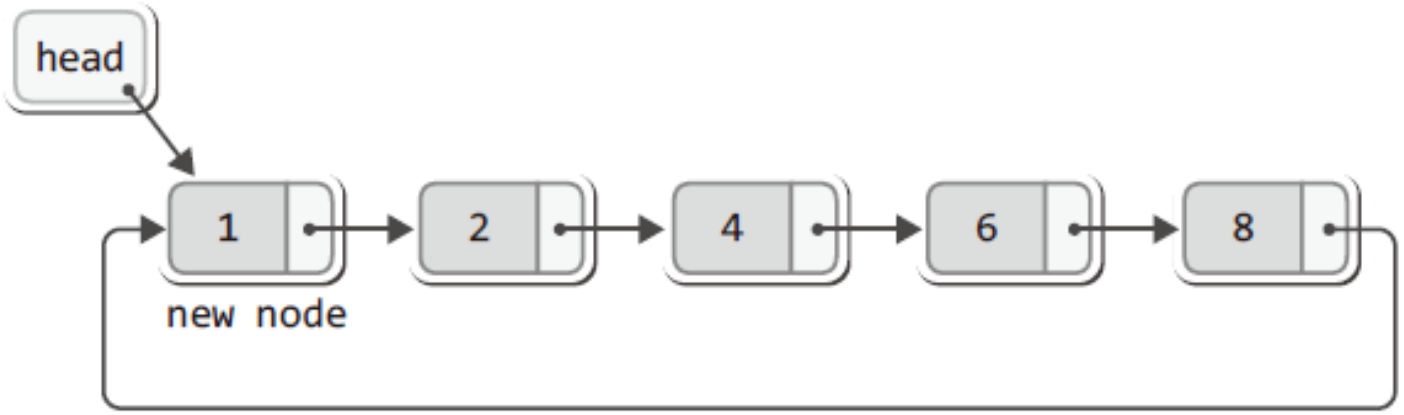
- 원형 연결 리스트의 마지막 노드는 첫 번째 노드를 가리킴



▶ [그림 05-2 : 원형 연결 리스트]

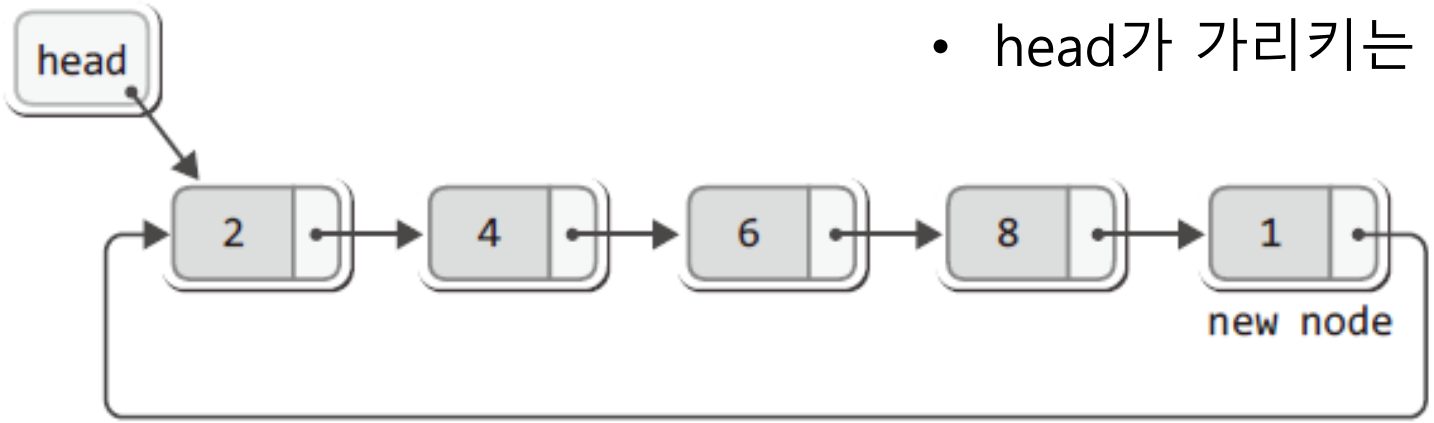
원형 연결 리스트의 노드 추가

- 모든 노드가 연결되어 있기 때문에 머리와 꼬리 구분이 없음



▶ [그림 05-3: 원형 연결 리스트의 머리에 노드 추가]

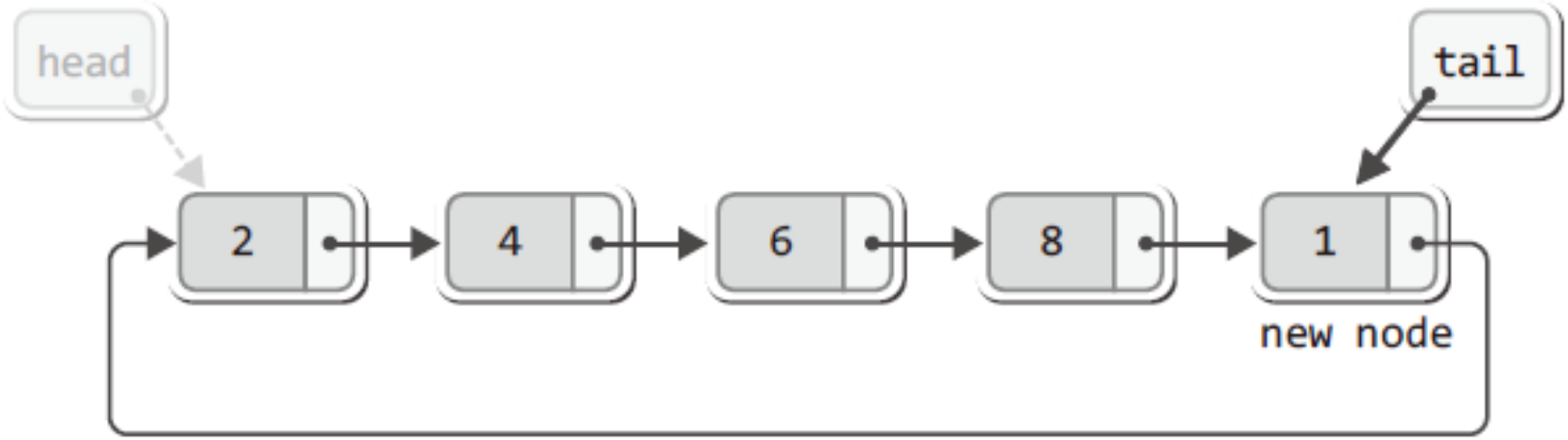
- 두 경우의 노드 연결 순서가 같다
- head가 가리키는 노드가 다르다.



▶ [그림 05-4: 원형 연결 리스트의 꼬리에 노드 추가]

원형 연결 리스트의 장점

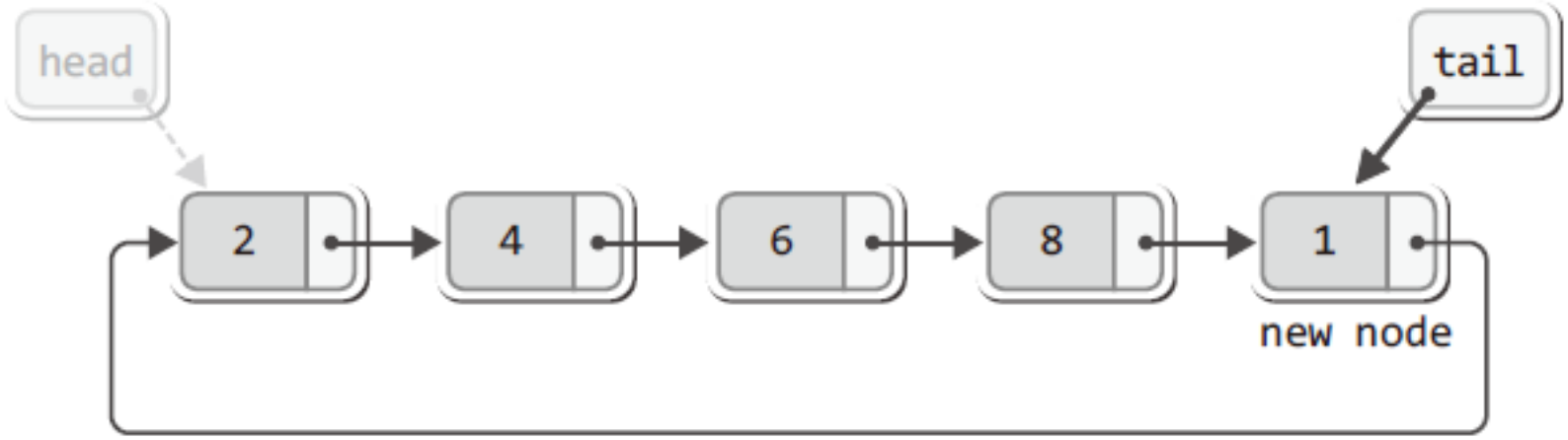
- 단순 연결 리스트처럼 머리와 꼬리를 가리키는 포인터 변수를 각각 두지 않아도, 하나의 포인터 변수만 있어도 머리 또는 꼬리에 노드를 간단히 추가할 수 있다.



▶ [그림 05-6: 변형된 원형 연결 리스트]

원형 연결 리스트

- 연결 리스트를 가리키는 포인터 변수 하나로 머리와 꼬리를 쉽게 확인 가능
  - 꼬리를 가리키는 포인터 변수: tail
  - 머리를 가리키는 포인터 변수: tail->next



▶ [그림 05-6: 변형된 원형 연결 리스트]

변형된 원형 연결 리스트의 구현범위

- 이전에 구현한 연결 리스트와 기능 동일
  - 조회관련 Lfirst
  - 삭제관련 Lremove
  - 이외의 부분

- 조회관련 Lnext
  - 원형 연결 리스트를 계속해서 순환하는 형태로 변경!
- 삽입관련
  - 앞과 뒤에 삽입이 가능하도록 두 개의 함수 정의!
- 정렬관련
  - 정렬과 관련된 부분 전부 제거

원형 연결 리스트의 헤더파일과 초기화 함수

CLinkedList.h

```
typedef int Data;

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

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

typedef CList List;
```

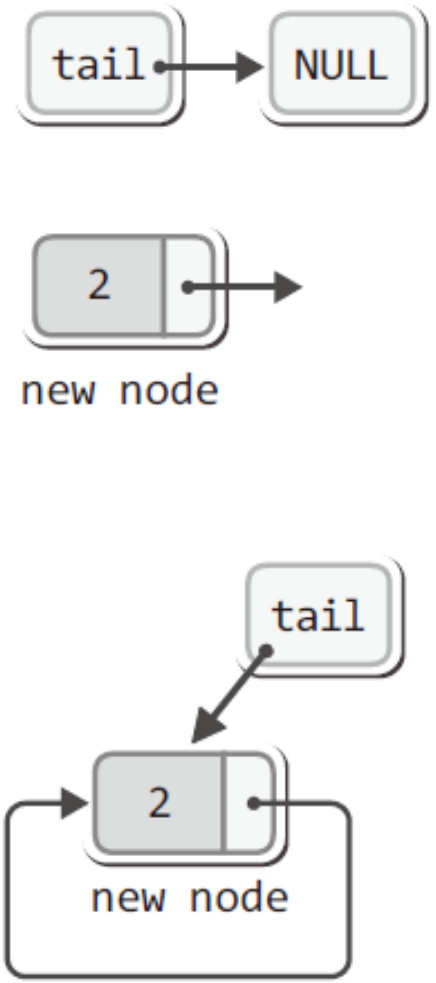
```
void ListInit(List * plist);
void LInsert(List * plist, Data data);
void LInsertFront(List * plist, Data data);

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

CLinkedList.c

```
void ListInit(List * plist)
{
    plist->tail = NULL;
    plist->cur = NULL;
    plist->before = NULL;
    plist->numOfData = 0;
} //모든 멤버를 NULL과 0으로  초기화
```

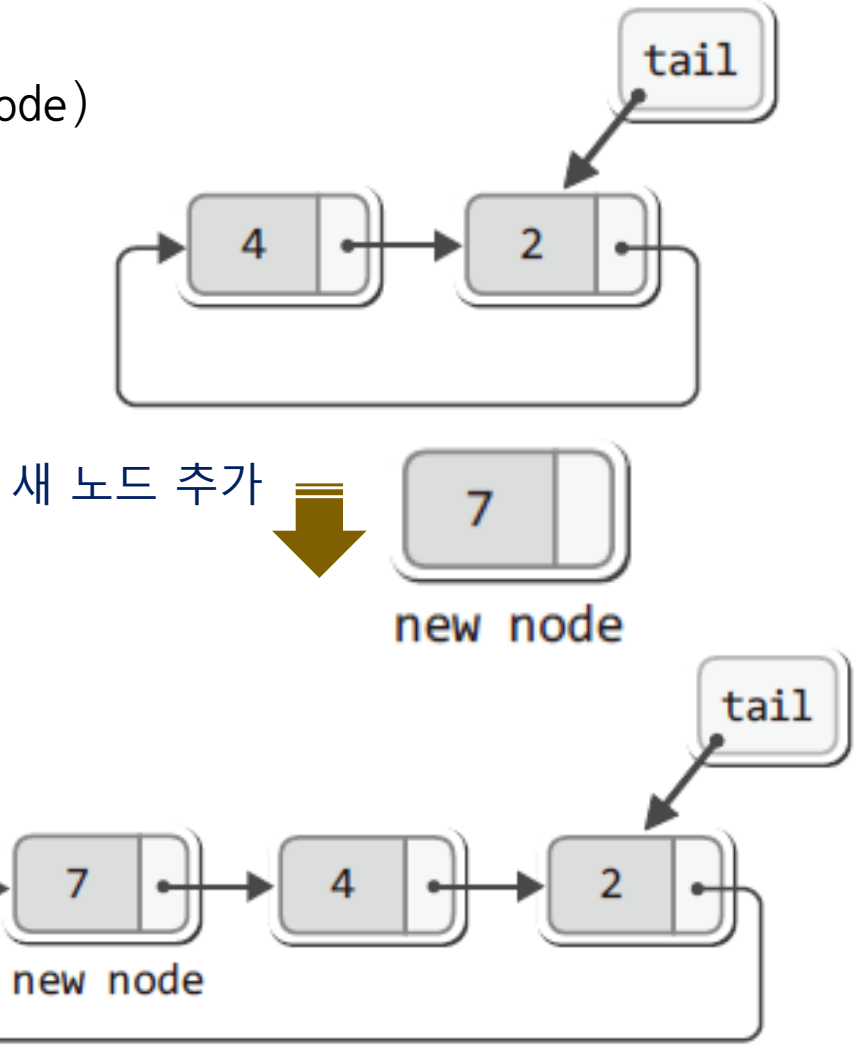
원형 연결 리스트 구현: 첫 번째 노드 삽입

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

    if(plist->tail == NULL) // 첫 노드
    {
        plist->tail = newNode;
        newNode->next = newNode;
    }
    else // 두 번째 노드 이후
    {
        newNode->next = plist->tail->next;
        plist->tail->next = newNode;
    }

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



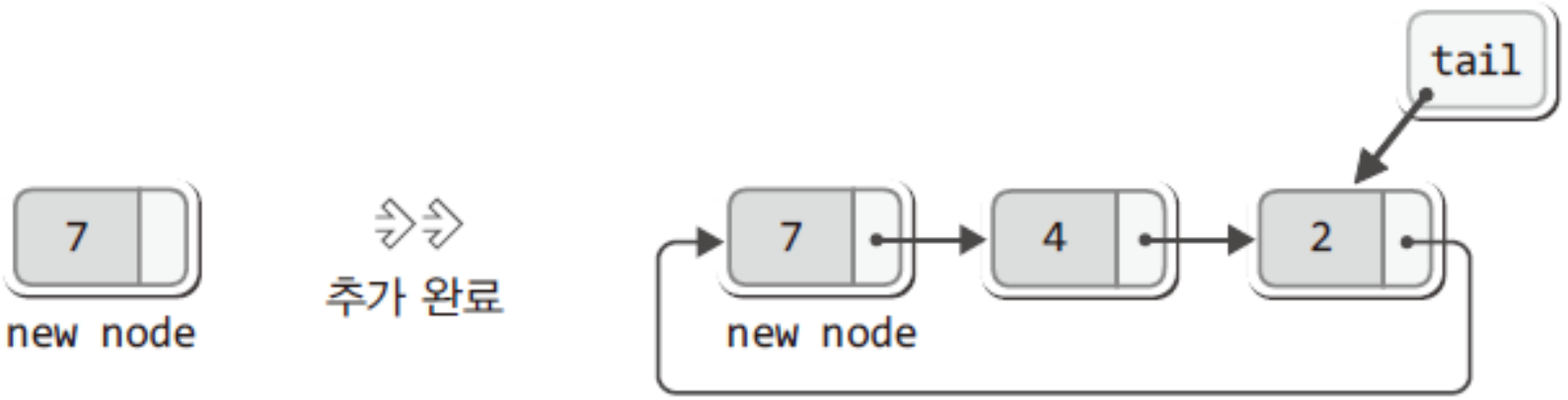
원형 연결 리스트 구현: 두 번째 이후 노드 머리로 삽입

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

    if(plist->tail == NULL) // 첫 노드
    {
        plist->tail = newNode;
        newNode->next = newNode;
    }
    else // 두 번째 노드 이후
    {
        newNode->next = plist->tail->next;
        plist->tail->next = newNode;
    }

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

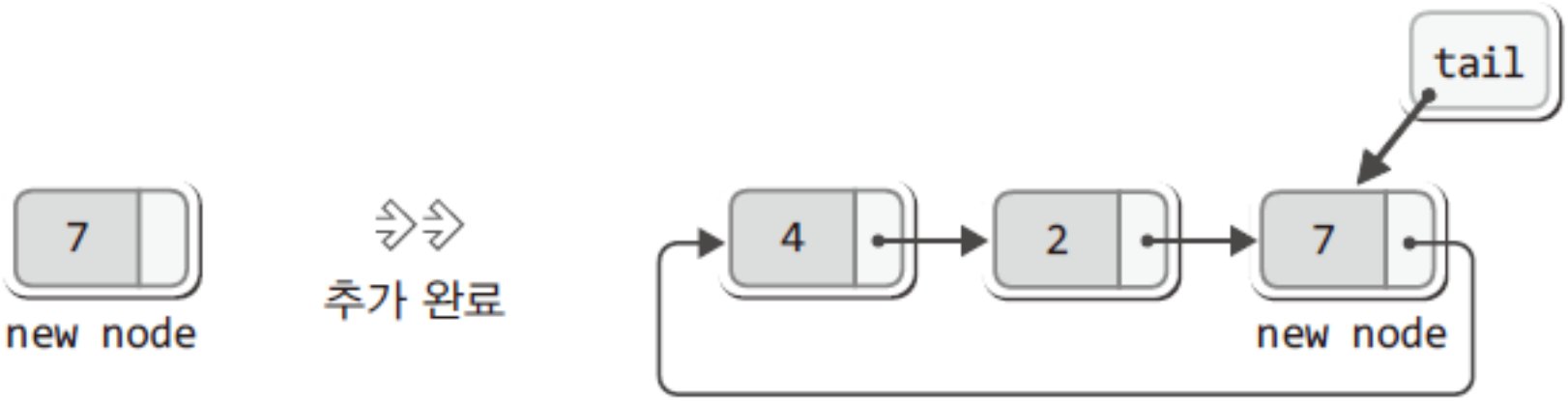


원형 연결 리스트 구현: 앞과 뒤의 삽입 과정 비교1

- 노드를 머리에 추가한 결과



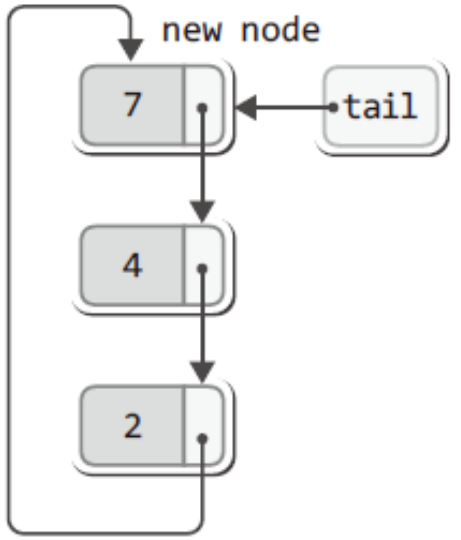
- 노드를 꼬리에 추가한 결과



원형 연결 리스트 구현: 앞과 뒤의 삽입 과정 비교2

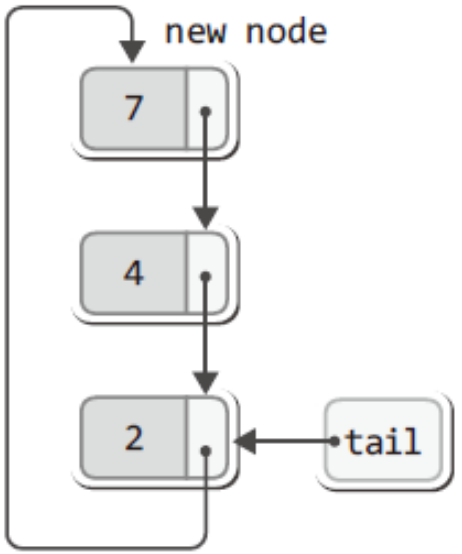
```
void LInsert(List * plist, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;

    if(plist->tail == NULL)
    {
        plist->tail = newNode;
        newNode->next = newNode;
    }
    else
    {
        newNode->next = plist->tail->next;
        plist->tail->next = newNode;
        plist->tail = newNode; //Linsertfront와 차이
    }

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

꼬리 추가



tail의 위치가 다름

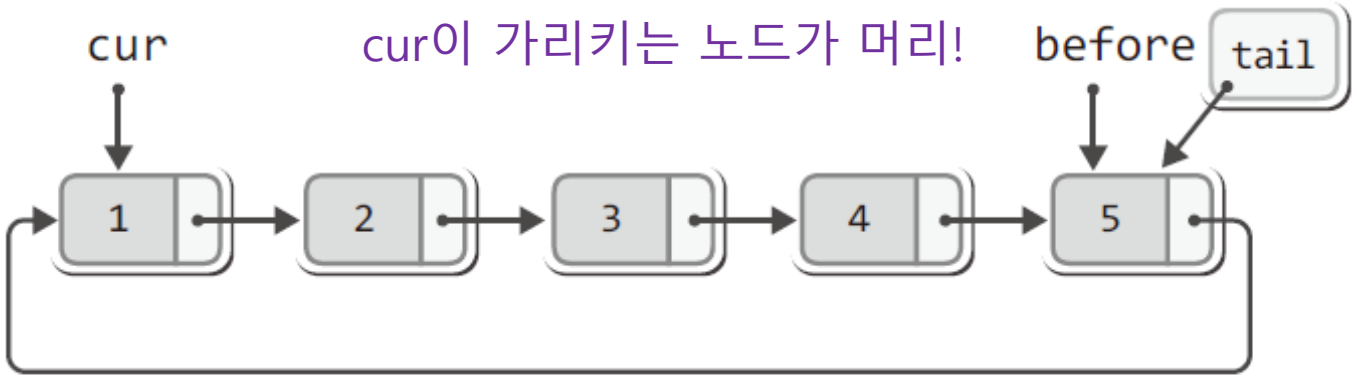
머리 추가



원형 연결 리스트 구현: 조회 LFirst

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

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

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

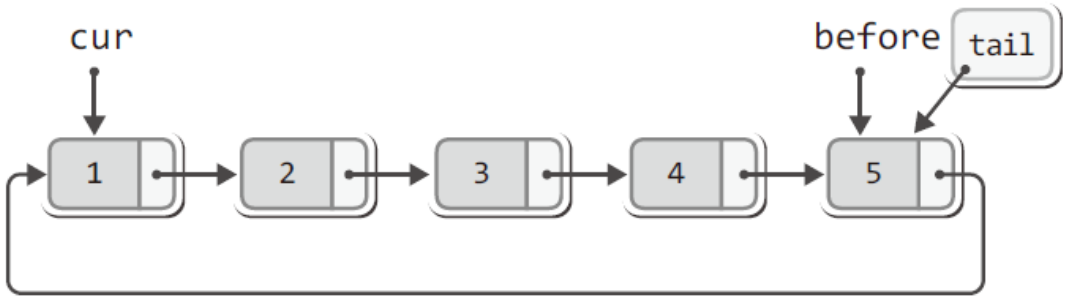
```
typedef struct _CLL
{
    Node * tail;
    Node * cur;
    Node * before;
    int numOfData;
} CList;

typedef CList List;
```



▶ [그림 05-12: LFirst 함수의 호출결과]

원형 연결 리스트 구현: 조회 LNext

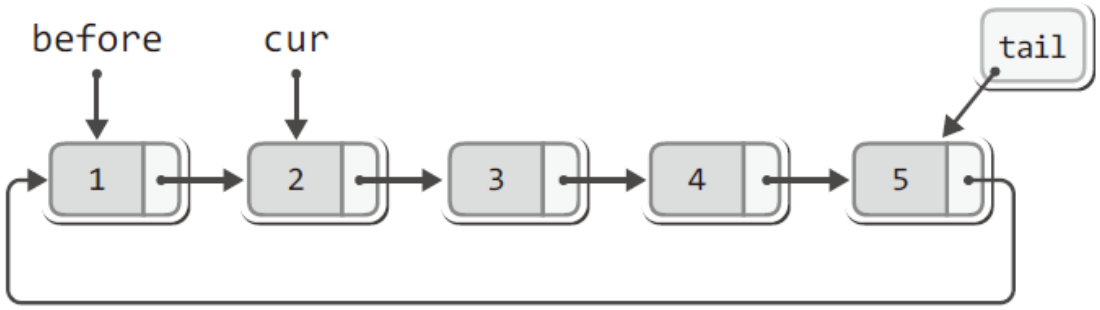
```
int LNext(List * plist, Data * pdata)
{
    if(plist->tail == NULL)    //
        return FALSE;

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

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



▶ [그림 05-12: LFirst 함수의 호출결과]

이어지는

LNext 호출결과



▶ [그림 05-13: LNext 함수의 호출결과]

원형 연결 리스트이므로

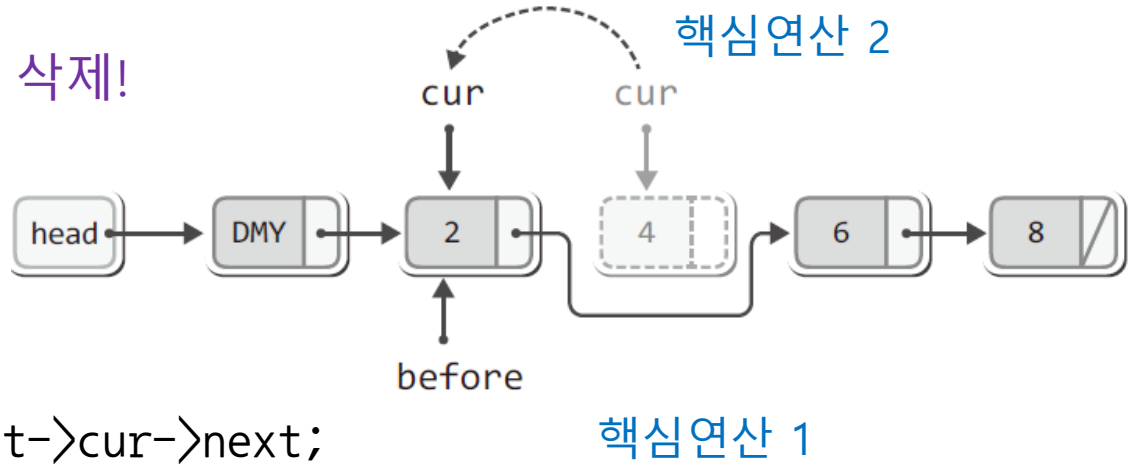
리스트의 끝을 검사하는

코드가 없다!

원형 연결 리스트 구현: 노드의 삭제(복습)

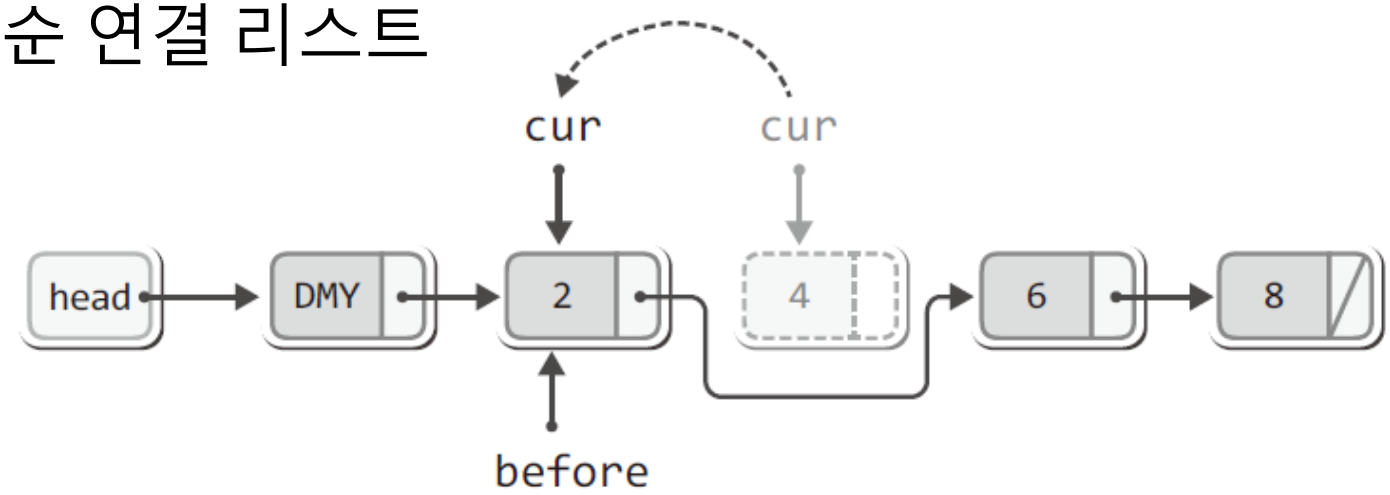
- 핵심연산
  1. 삭제할 노드의 이전 노드가, 삭제할 노드의 다음 노드를 가리키게 한다.
  2. 포인터 변수 cur을 한 칸 뒤로 이동시킨다.

```
// 더미 기반 단순 연결 리스트의 삭제!
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;
}
```

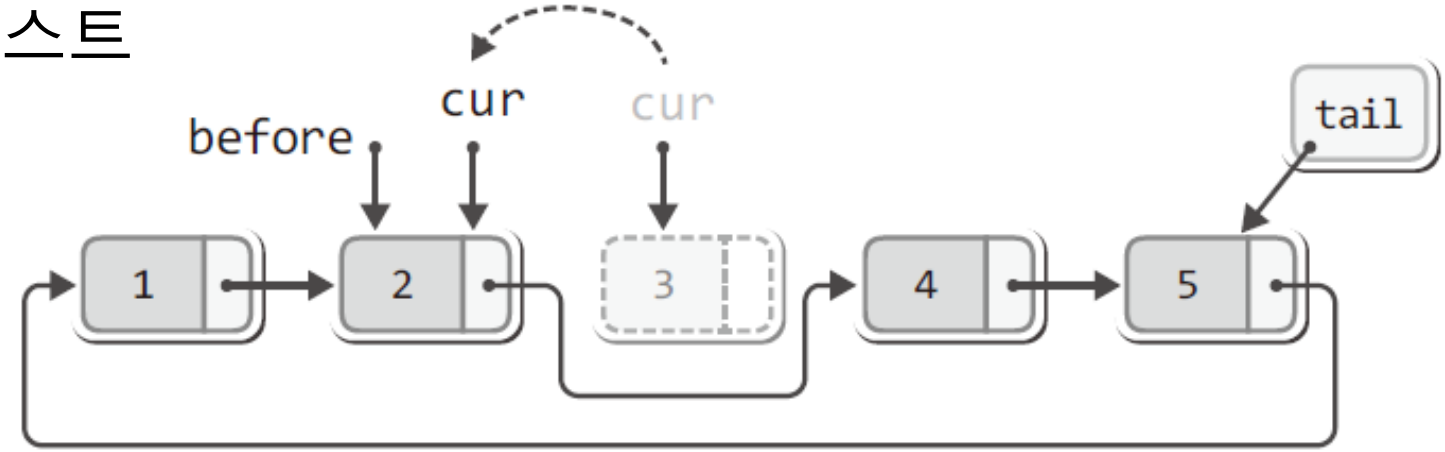


원형 연결 리스트 구현: 노드의 삭제(그림 비교)

- 더미 기반 단순 연결 리스트



- 원형 연결 리스트



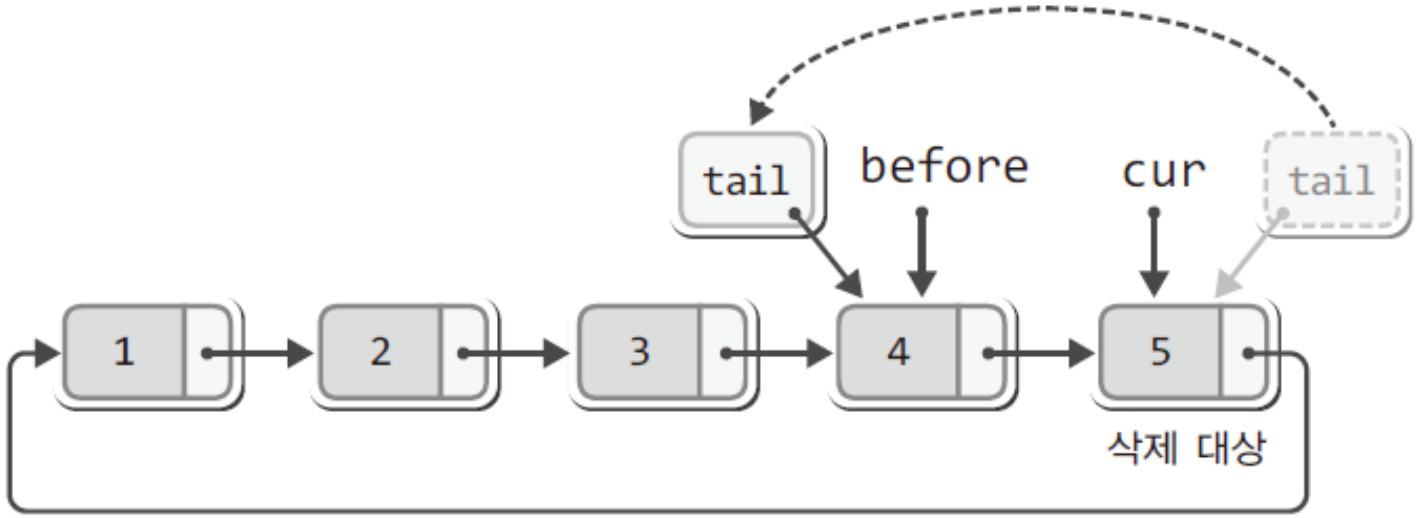
삭제 과정이 비슷해 보이지만 원형 연결 리스트에는 더미 노드가 없기 때문에 상황에 따라 삭제의 과정이 달라짐

원형 연결 리스트 노드 삭제 구현 1

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

    if(rpos == plist->tail)    // 삭제 대상을 tail이 가리킨다면
    {
    if(plist->tail == plist->tail->next)    // 마지막 남은 노드라면
        plist->tail = NULL;
    else
        plist->tail = plist->before;
    }

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

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

원형 연결 리스트 노드 삭제 구현 2

- 예외 상황
  1. 삭제할 노드를 tail이 가리키는 경우
  2. 삭제할 노드를 tail이 가리키는데, 그 노드가 마지막 노드라면



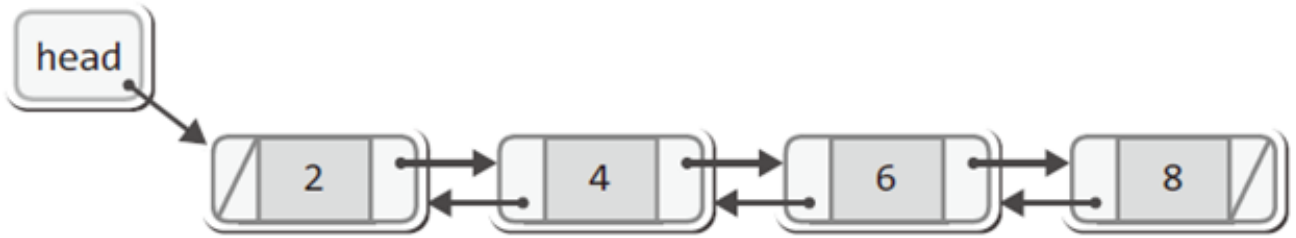
▶ [그림 05-16: 삭제의 예외적인 경우]

이중 연결 리스트

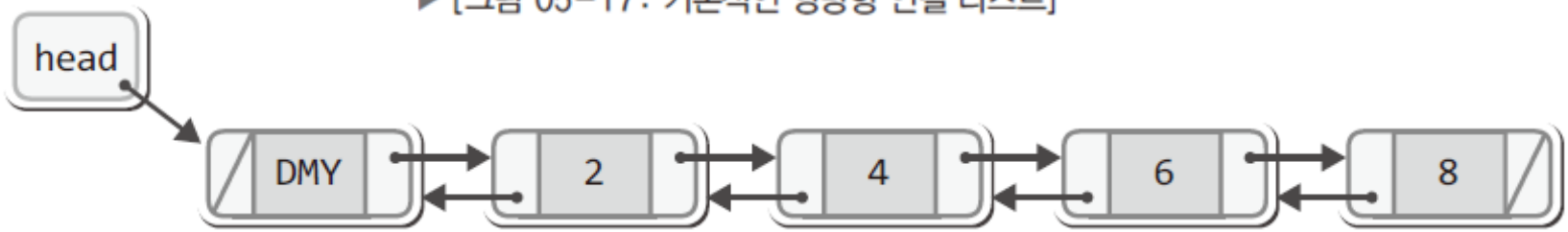
양방향 연결 리스트의 이해

- 양방향 리스트를 위한 노드의 표현

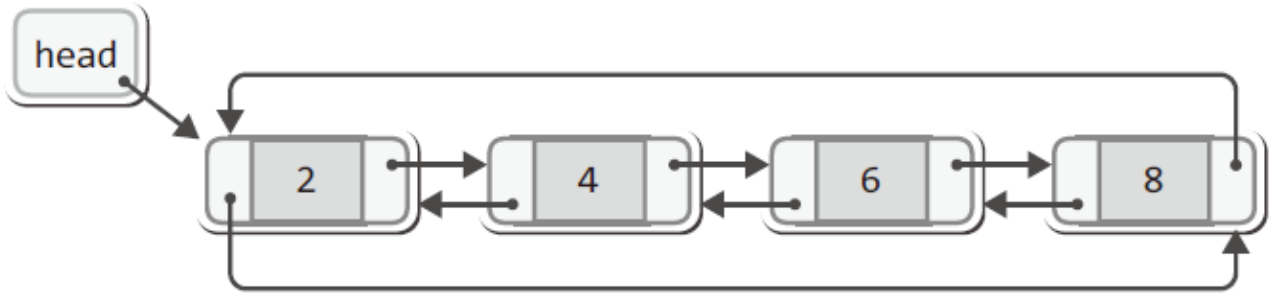
```
typedef struct _node
{
    Data data;
    struct _node * next;
    struct _node * prev;
} Node;
```



▶ [그림 05-17: 기본적인 양방향 연결 리스트]



▶ [그림 05-18: 더미 노드 양방향 연결 리스트]



▶ [그림 05-19: 원형 연결 기반의 양방향 연결 리스트]

양방향으로 노드를 연결하는 이유!

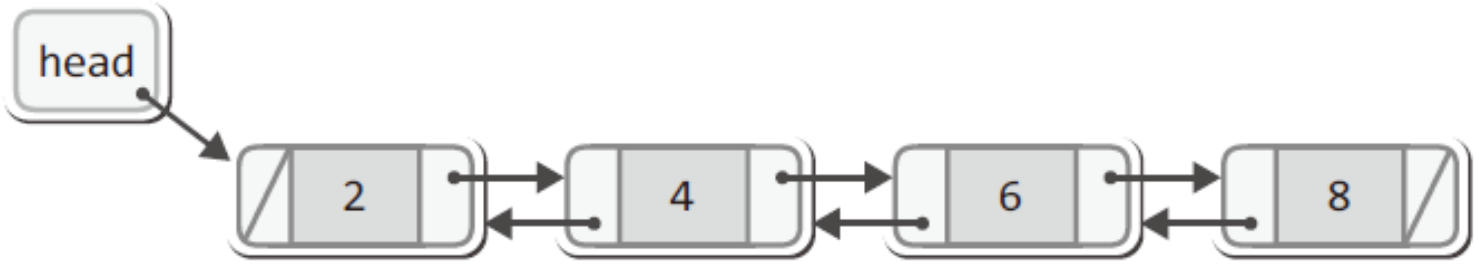
- 오른쪽 노드로의 이동이 쉽고, 양방향으로 이동이 가능하다!
- 양방향으로 연결하는 코드도 복잡하지 않음

```
//단순 연결 리스트의 LNext
int LNext(List * plist, LData * pdata)
{
    if(plist->cur->next == NULL)
        return FALSE;

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

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

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

    // before 유지 불필요
    plist->cur = plist->cur->next;
    *pdata = plist->cur->data;

    return TRUE;
}
```

양방향 연결 리스트 모델

- LRemove 함수를 ADT에서 제외
- ADT에 왼쪽 노드의 데이터를 참조하는 LPrevious 함수를 추가
- 새 노드는 머리에 추가



▶ [그림 05-20: 함께 구현할 양방향 연결 리스트의 구조]

양방향 연결 리스트의 헤더파일: DBLinkedList.h

```
typedef int Data;

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

typedef struct _dbLinkedList
{
    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);

// LPrevious 함수는 LNext 함수와 반대로 이동
int LPrevious(List * plist, Data * pdata);

int LCount(List * plist);
```

연결 리스트의 구현: 리스트의 초기화

- ListInit 함수의 정의에 참조해야 하는 구조체의 정의

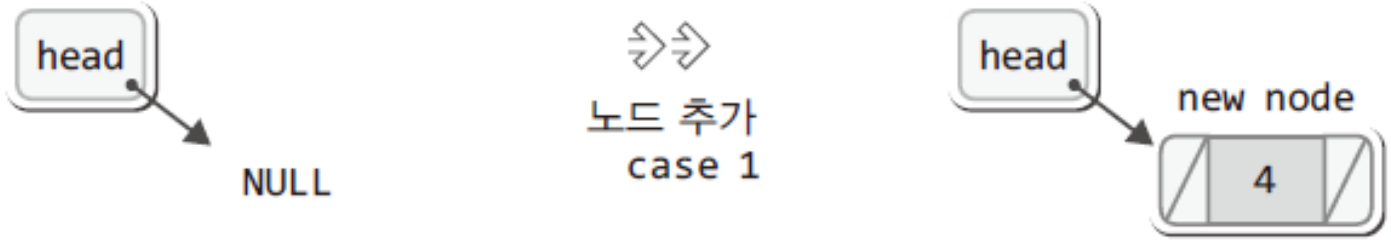
```
typedef struct _dbLinkedList
{
    Node * head;
    Node * cur;
    int numOfData;
} DBLinkedList;
```

- head와 numOfData만 초기화 필요
  - 멤버 cur은 조회의 과정에서 초기화 됨

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

연결 리스트의 구현: 노드 삽입의 구분 1

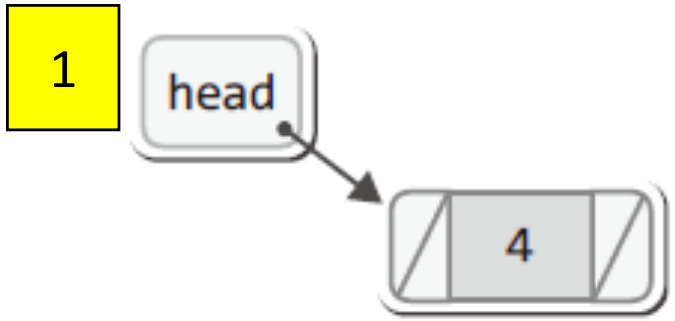
- 빈 리스트에 삽입



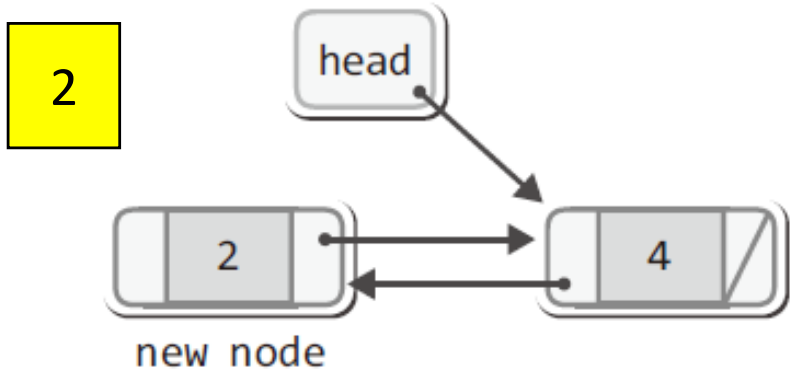
▶ [그림 05-21: 첫 번째 노드의 추가]

연결 리스트의 구현: 노드 삽입의 구분 2

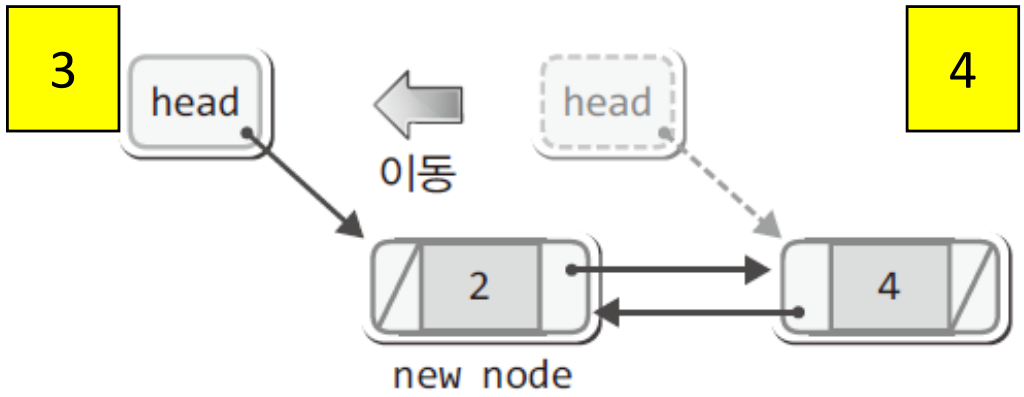
- 두 번째 이후부터의 노드 삽입

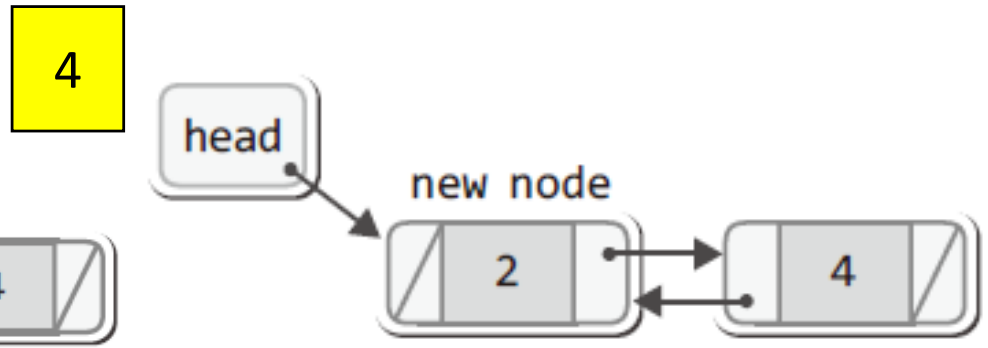


▶ [그림 05-22: 두 번째 이후의 노드 추가



▶ [그림 05-23: 두 번째 노드의 추가과정 1/2]





▶ [그림 05-24: 두 번째 노드의 추가과정 2/2]

연결 리스트의 구현

```
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)++;
}
```

양방향 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;
}
```

- LFirst와 LNext는 단방향 연결 리스트 동일
- LPrevious와 LNext의 차이는 이동방향 뿐