게임 자료구조와 알고리즘
-CHAPTER7-

SOULSEEK

목차

추상 자료형(ADT : ABSTRACT DATA TYPE)

1. 추상 자료형(ADT : ABSTRACT DATA TYPE)

추상 자료형**(ADT : Abstract Data Type)**

1. 구체적인 기능의 완성을 언급하지 않고**,** 순수하게 기능이 무엇인지 나열한 것**.**

2. 자료형의 전의에 기능혹은 연산과 관련된 내용을 명시할 수 있다**.**
- 구조체에 필요로 하는 연산을 함수를 이용해 정의하고 있고 함수의 정의까지 끝나면 그것이 구조체에 대한 정의가 끝나는 것이다**.**

```
Type struct _wallet // 동전 및 지폐 일부만을 대상으로 표현한 지갑
{
    int coin100Num; // 100원짜리 동전의 수
    int bill5000Num; // 5000원짜리 지폐의 수
}Wallet;

int TakeOutMoney(Wallet* pw, int coinNum, int billNum); //돈을 꺼내는 연산
int PutMoney(Wallet* pw, int coinNum, int billNum); //돈을 넣는 연산
```

- **하지만, 실제 구현에 있어서 구현 함수와 구조체 할당만으로도 해당 구조체가 어떤 것들을 가지고 있는지 대충 알 수가 있다 그렇기 때문에 ADT를 명시할 때, 구조체의 선언 부분은 생략 할 수 있다.**

1. 추상 자료형(ADT : ABSTRACT DATA TYPE)

Wallet의 ADT

- **int TakeOutMoney(Wallet* pw, int coinNum, int billNum)**
  - 첫 번째 인자로 전달된 주소의 지갑에서 돈을 꺼낸다.
  - 두 번째 인자로 꺼낼 동전의 수 , 세 번째 인자로 꺼낼 지폐의 수를 전달 한다.
  - 꺼내고자 하는 돈의 총액이 반환된다. 그리고 그만큼 돈은 차감된다.

- **int PutMoney(Wallet* pw, int coinNum, int billNum)**
  - 첫 번째 인자로 전달된 주소의 지갑에 돈을 넣는다.
  - 두 번째 인자로 넣을 동전의 수, 세 번째 인자로 넣을 지폐의 수를 전달 한다.
  - 넣은 만큼 동전과 지폐의 수가 증가한다.

자료구조의 ADT를 정의하고 ADT를 근거로 활용하는 main을 작성함수를 정의한 뒤 ADT를 근거로 해당 자료구조를 구현하는 형식으로 진행 하는 것이 좋은 방법!!

ADT는 표준이 존재하지 않으며 구조를 설정하는 사람마다 다를 수 있다!!

배열을 이용한 리스트의 구현

2. 배열을 이용한 리스트의 구현

List의 이해

- **LinkedList**만을 의미하는 것이 아니다.
- 순차 리스트(배열을 기반으로 구현된 리스트), 연결 리스트(메모리의 동적 할당을 기반으로 구현된 리스트 - **LinkedList**)를 모두 말한다.
- 데이터를 나란히 저장하고 중복된 데이터의 저장을 막지 않는다.

2. 배열을 이용한 리스트의 구현

List 자료구조의 ADT

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

- **void LInsert(List* plist, LData data); - 삽입**
  - 리스트에 데이터를 저장한다. 매개 변수 **data**에 전달된 값을 저장한다.

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

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

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

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

2. 배열을 이용한 리스트의 구현

```
#ifndef __ARRAY_LIST_H__
#define __ARRAY_LIST_H__

#define TRUE		1 // '참'을 표현하기 위한 매크로 정의
#define FALSE		0 // '거짓'을 표현하기 위한 매크로 정의

#define LIST_LEN  100

typedef int LData; // LData에 대한 typedef 선언

typedef struct __ArrayList // 배열기반 리스트를 정의한 구조체
{
	LData arr[LIST_LEN]; // 리스트의 저장소인 배열
	int numOfData; // 저장된 데이터의 수
	int curPosition; // 데이터 참조위치를 기록
}ArrayList;

typedef ArrayList List;

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

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

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

#endif
```

2. 배열을 이용한 리스트의 구현

삽입과 조회

초기화

```
void LIstInit(list* plist)
{
    (plist -> numOfData)   = 0; // 리스트에 저장된 데이터의 수는 0!
    (plist -> curPosition)   = -1; // 현재 아무 위치도 가리키지 않음!
}
```

삽입

```
void Linsert(List* plist, LData data)
{
    if(plist -> numOfData >= LIST_LEN) // 더 이상 저장할 공간이 없다면...
    {
        puts("저장이 불가능합니다.")
        return;
    }

    plist -> arr[plist -> numOfData] = data; // 데이터 저장
    (plist -> numOfData)++; // 저장된 데이터의 수 증가
}
```

2. 배열을 이용한 리스트의 구현

조회

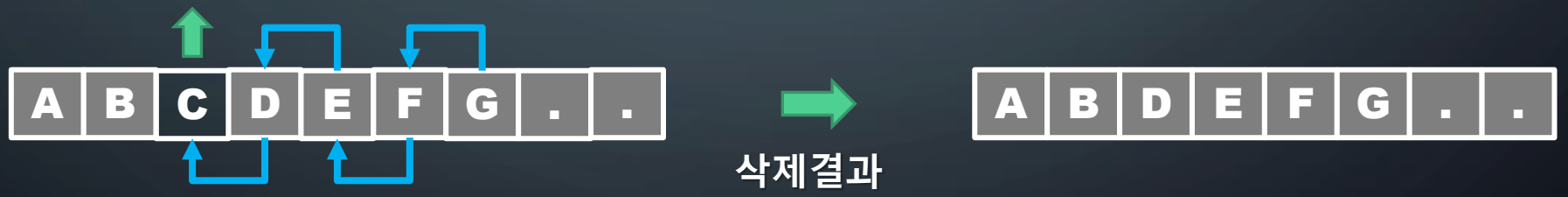
- **int LFirst(List* plist, LData* pdata);** // 첫 번째 조회
- **int LNext(List* plist, LData* pdata);** // 두 번째 이후의 조회

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

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

	return TRUE;
}

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

	(plist -> curPosition)++;
	*pdata = plist -> arr[plist->curPosition];

	return TRUE;
}
```

2. 배열을 이용한 리스트의 구현

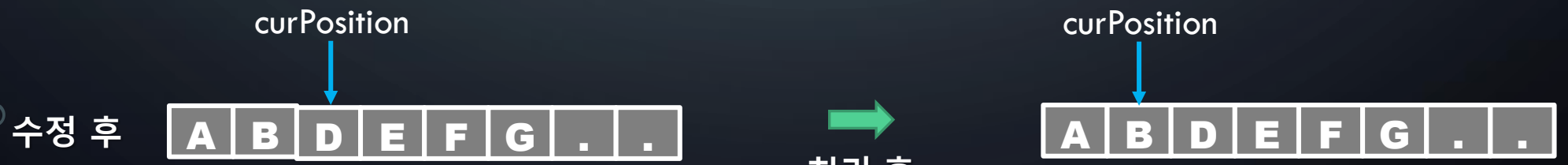
삭제

- **LRemove** 함수가 호출되면 리스트의 멤버 **curPosition**을 확인해서, 조회가 이뤄진 데이터의 위치를 확인한 다음 그 데이터를 삭제.
- 배열로 된 리스트는 앞에서부터 데이터가 순차적으로 채워지는게 원칙이니 중간의 데이터가 사라지면 그 공간을 뒤에 저장된 데이터가 앞으로 이동하면서 채워줘야 한다.



2. 배열을 이용한 리스트의 구현

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

	// 삭제를 위한 데이터의 이동을 진행하는 반복문
	for(int i = rpos; i < num – 1; i++)
		plist -> arr[i] = plist -> arr[i + 1];

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



2. 배열을 이용한 리스트의 구현

LData의 자료형을 구조체 변수로 저장해보기

- **void SetPointPos(Point* ppos, int xpos, int ypos);** // 값을 저장
- **void ShowPointPos(Point* ppos);** //정보 출력
- **void PointComp(Point* pos1, Point* pos2);** // 비교
  - 두 **Point**변수의 멤버 **xpos**만 같으면 **1**
  - 두 **Point**변수의 멤버 **ypos**만 같으면 **2**
  - 두 **Point**변수의 멤버가 모두 같으면 **0**
  - 두 **Point**변수의 멤버가 모두 다르면 **-1**

```
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);
```

ArrayList_Point의 코드를 확인해보자. 달라진 부분과 활용 부분을 꼼꼼히 살펴보자.

2. 배열을 이용한 리스트의 구현

배열 기반 리스트(순차 리스트)의 단점 – LinkedList와 비교했을 때

- 배열의 길이가 초기에 결정되어야 한다. 변경이 불가능하다.
- 삭제의 과정에서 데이터의 이동(복사)가 매우 빈번히 일어난다.

배열 기반 리스트(순차 리스트)의 장점 – LinkedList와 비교했을 때

- 데이터의 참조가 쉽다. 인덱스 값을 기준으로 어디든 한 번에 참조가 가능하다.

배열 기반 리스트(순차 리스트)는 데이터의 수가 정해져 있고 변화가 없는 상황에서 참조만해서 사용할 경우에만 사용하는 것이 바람직하다.

2. 배열을 이용한 리스트의 구현

학습과제

- 문제 문서에 나와 있는 내용을 작성해 보자.(문제 제출 폴더 – **Chapter7** 참조)

LINKEDLIST의 개념적인 이해

3. LINKEDLIST의 개념적인 이해

순차 리스트는 크기가 정적이기 때문에 유연하지 못하다 연결 리스트에서는 이것을 보완해서 유연한 동적 할당을 이용해 할 수 있게 구현해야 한다.

구조체를 가리키는 포인터의 이용

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

순차 리스트에서 보았듯이 서로가 서로를 '**연결**'되어 있는 형태가 되어 있었다 그중 이미 다음 인덱스와 연결이 되어 있는 배열을 이용해 참조하는 곳만 이동시켜서 간편하게 구현 할 수 있었다 하지만 동적 할당의 형태에 제약이 있었다. 그래서 동적할당을 위해서 '**구조체를 가리키는 포인터**'를 이용하면 데이터가 필요할 때마다 하나씩 추가해서 사용할 수 있고 우리는 이를 통상 **Node(서로를 가리키는 통로)**라고 명명하자.

3. LINKEDLIST의 개념적인 이해

삽입

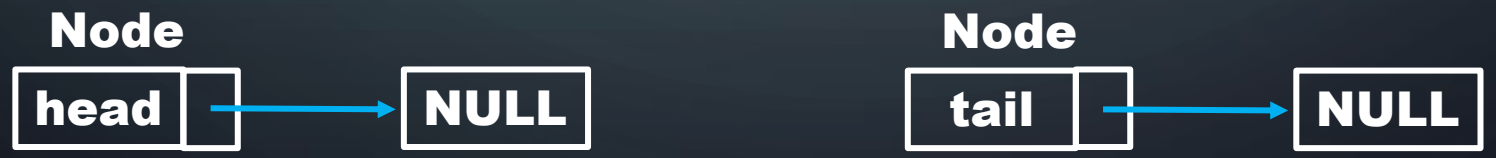
- **LinkedList**는 시작과 끝이 있고 각 **Data**끼리 연결되어 있기 때문에 **Head Node, Tail Node**가 필요하고 검색에 사용하기 위해 **curPoint**가 필요하다**.**

Main에 사용될 때

Node* head = NULL; // 리스트의 머리를 가리키는 포인터 변수
Node* tail = NULL; // 리스트의 꼬리를 가리키는 포인터 변수
Node* cur = NULL; // 저장된 데이터의 조회에 사용되는 포인터 변수

라고 각각 선언할 수 있다**.**

- 필요한 **Node**의 구성이 끝나면 삽입부분을 구현해 보자**.**



3. LINKEDLIST의 개념적인 이해

삽입

- 삽입을 실행하게 되면 데이터가 새로 생성되어서 연결 되어야 한다는 뜻이다.

  1. Node를 생성해주고 Node에 데이터를 저장해주고 해당 Node의 다음 Node를 NULL로 초기화 한다.

  2. 생성한 Node가 첫 번째 Node라면 head부분이 첫 번째 Node를 가리키게 하고 두 번째 이 후 Node라면 다음 Node를 가리키게 한다.

```
While(1)
{
    newNode = (Node*)malloc(sizeof(Node)); // Node 생성
    newNode -> data = readData;            // Node에 데이터 저장
    newNode -> next = NULL;                // Node의 next를 NULL로 초기화

    if(head == NULL)                       // 첫 번째 Node라면!
        head -> next = newNode;            // 첫 번째 Node를 head가 가리키게.
    else                                   // 두 번째 Node라면!
        tail -> next = newNode;

    tail = newNode;                        //Node의 끝을 가리키게 한다.
}
```

1번 과정

2번 과정

3. LINKEDLIST의 개념적인 이해

첫 번째 Node 추가 과정

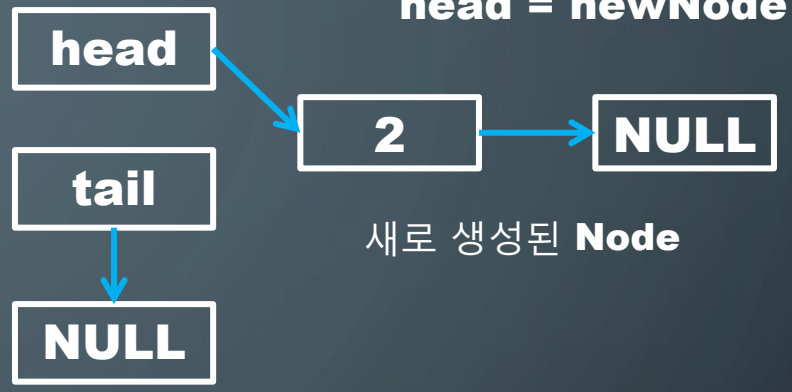
1번 과정

새로 생성된 **Node**

```
newNode = (Node*)malloc(sizeof(Node));
newNode->data = readData;
newNode->next = NULL;
```

2번 과정

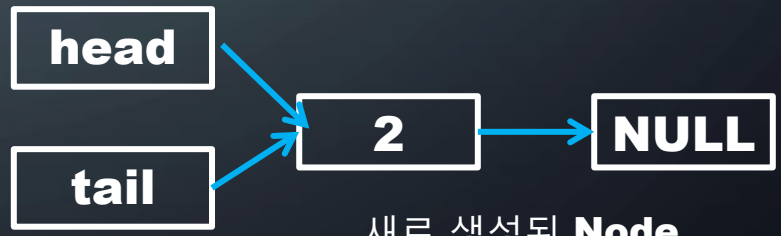
if(head == NULL)
head = newNode



새로 생성된 **Node**

2번 과정

Tail = newNode;

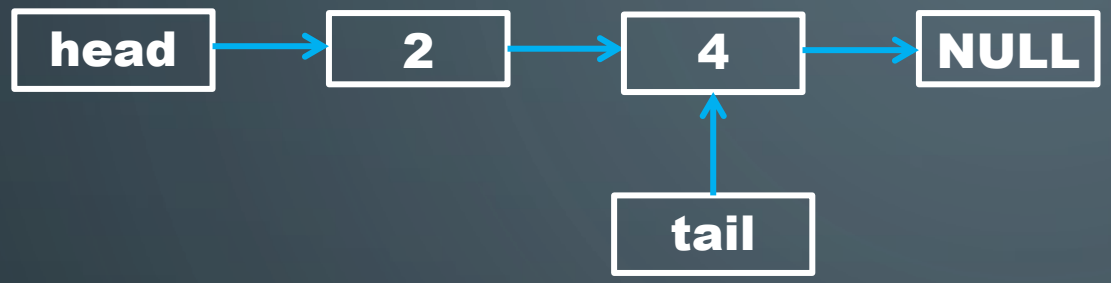


새로 생성된 **Node**

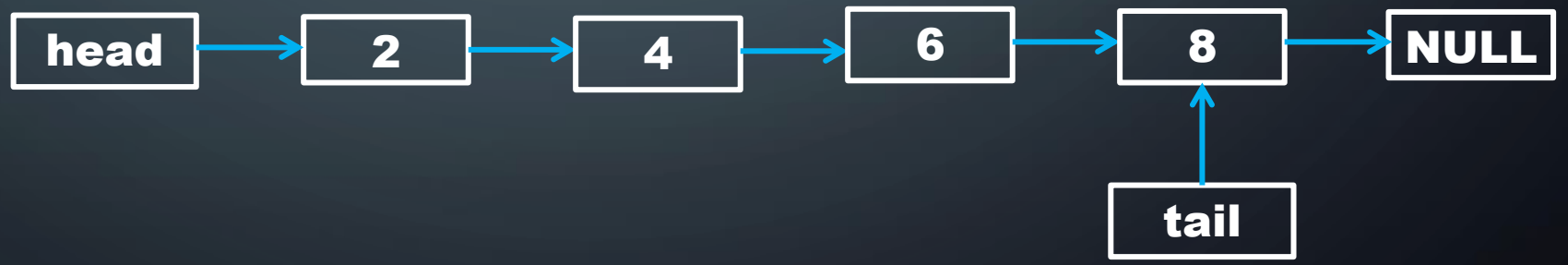
3. LINKEDLIST의 개념적인 이해

두 번째 이후 Node 추가과정

추가 생성된 **Node**



다수의 데이터가 추가 생성된 **Node**



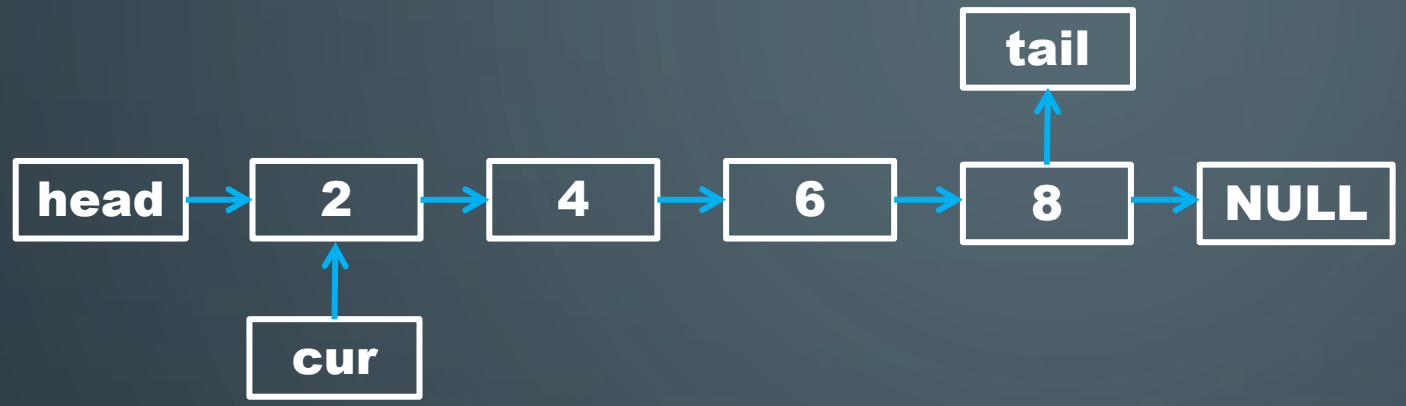
3. LINKEDLIST의 개념적인 이해

조회

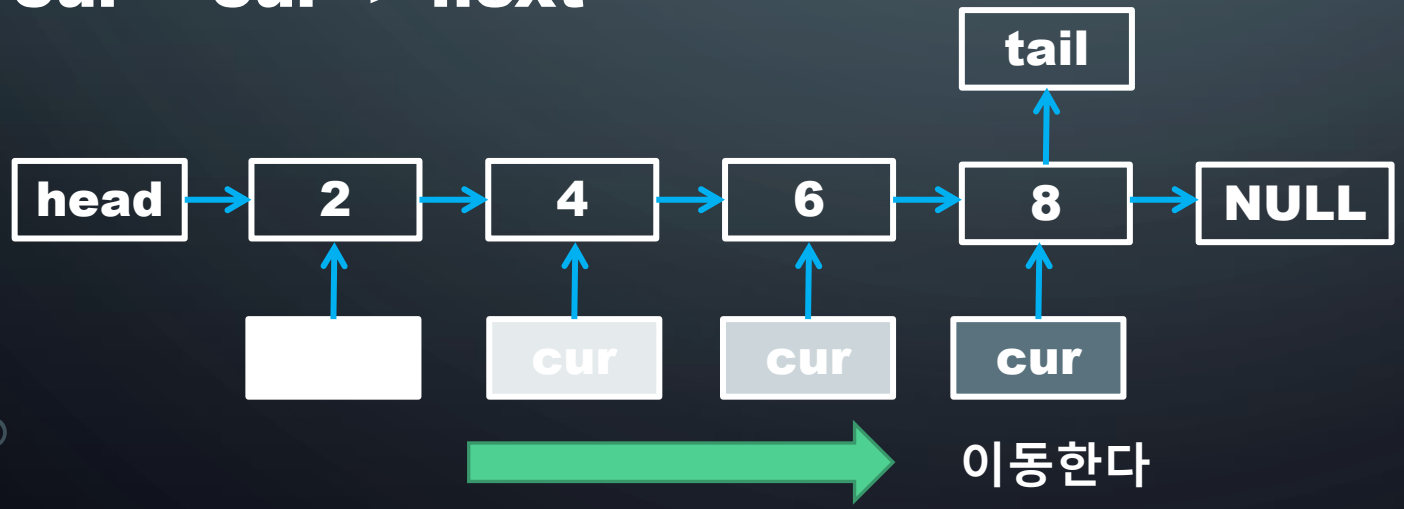
```
if(head == NULL)                    // head가 가리키는 것이 존재하지 않다면..
{
    printf("저장된 자연수가 존재하지 않습니다. \n");
}
else
{
    cur = head;                     // cur이 리스트의 첫 번째 Node를 가리킨다.
    printf("%d ", cur -> data);     // 첫 번째 데이터 출력

    while(cur -> next != NULL)      // 연결된 Node가 존재한다면....
    {
        cur = cur->next;            // cur이 다음 Node를 가리키게 한다.
        printf("%d ", cur -> data); // cur이 가리키는 Node를 출력한다.
    }
}
```

3. LINKEDLIST의 개념적인 이해

cur = head;



cur = cur -> next



3. LINKEDLIST의 개념적인 이해

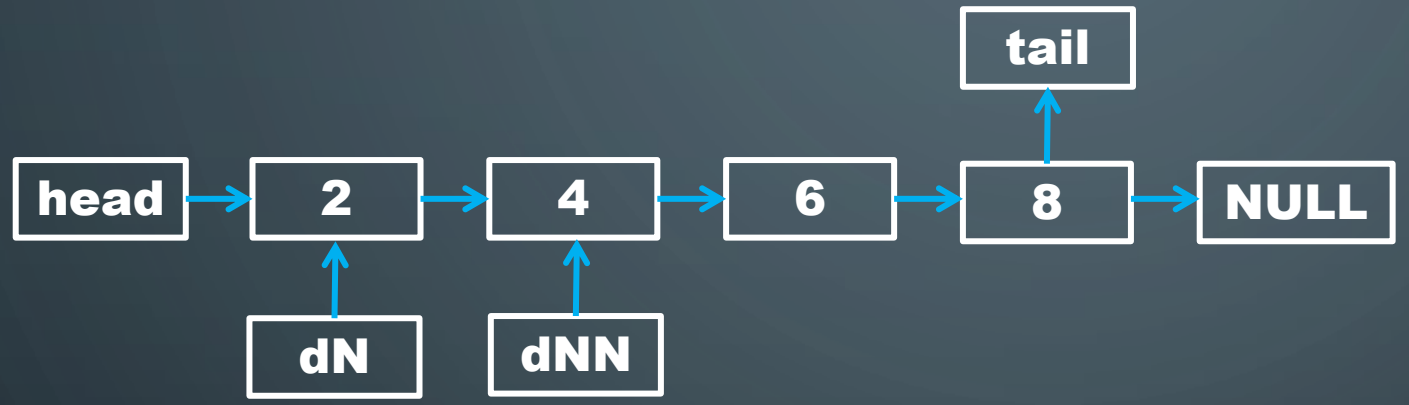
삭제

```
if(head == NULL)
{
	return 0;
}
else
{
	Node* delNode = head;
	Node* delNextNode = head -> next;

	printf("%d을(를) 삭제합니다. \n", head->data);
	free(delNode);                                  // 첫 번째 Node삭제

	while(deleNextNode != NULL)                     // 두 번째 이후 Node 삭제
	{
		delNode = delNextNode;
		delNextNode = delNextNode -> next;

		printf("%d을(를) 삭제합니다. \n", delNode->data);
		free(delNode);
	}
}
```

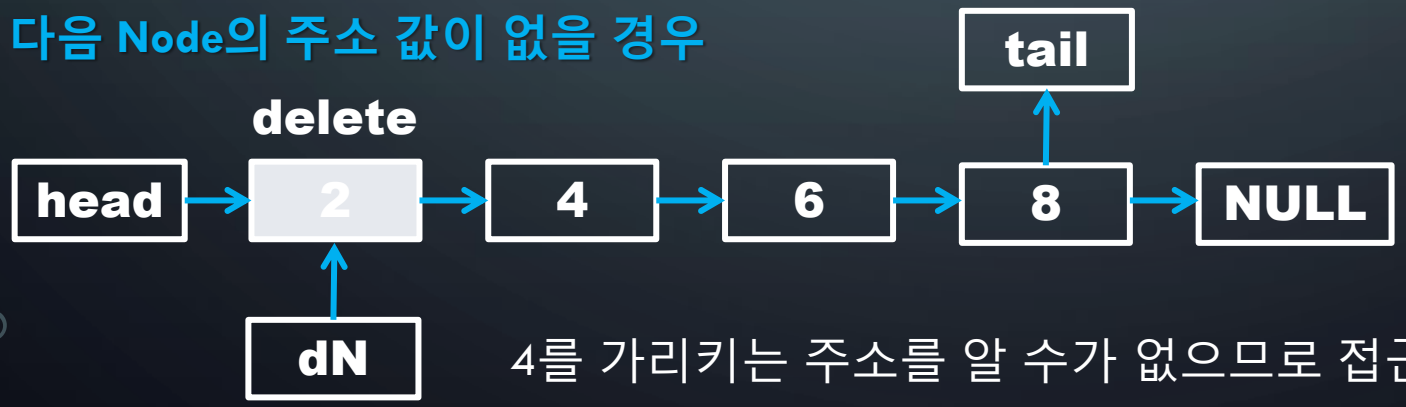
3. LINKEDLIST의 개념적인 이해

Node* delNode = head;
Node* delNextNode = head->next;

- 삭제가 될 Node가 가리키고 있는 다음 Node을 별도로 저장하지 않으면 접근이 불가능 하기 때문에 다음 Node의 주소 값을 별도로 저장할 필요가 있다.



다음 Node의 주소 값이 없을 경우



4를 가리키는 주소를 알 수가 없으므로 접근 불가

3. LINKEDLIST의 개념적인 이해

학습과제

- 문제 문서에 나와 있는 내용을 작성해 보자.(학습과제 폴더 – **Chapter7_LinkedList**의 이해)

단순 **LINKEDLIST**의 **ADT** 구현

4. 단순 LINKEDLIST의 ADT 구현

정렬 기능이 추가된 리스트 자료구조의 ADT

ArrayList를 구현과 차이는 없고 정렬기능만 추가 되었다.

- **void ListInit(List* plist); -** 초기화
  - 초기화 리스트의 주소 값을 인자로 전달한다.
  - 리스트 생성 후 제일 먼저 호출되어야 하는 함수이다.
- **void Linsert(List* plist, LData data); -** 삽입
  - 리스트에 데이터를 저장한다. 매개 변수 **data**에 전달된 값을 저장한다.
- **int Lfirst(List* plist, LData data); -** 조회
  - 첫 번째 데이터가 **pdata**가 가리키는 메모리에 저장된다.
  - 데이터의 참조를 위한 초기화가 진행된다.
  - 참조 성공 시 **TRUE(1),** 실패 시 **FALSE(0)**반환
- **int LNext(List* plist, LData data); -** 조회
  - 참조된 데이터의 다음 데이터가 **pdata**가 가리키는 메모리에 저장된다.
  - 순차적인 참조를 위해서 반복 호출이 가능하다.
  - 참조를 새로 시작 하려면 먼저 **LFirst** 함수를 호출해야 한다.
  - 참조 성공 시 **TRUE(1),** 실패 시 **FALSE(0)** 반환
- **LData LRemove(List* plist); -** 삭제
  - **LFirst** 또는 **LNext** 함수의 마지막 반환 데이터를 삭제한다.
  - 삭제된 데이터는 반환 된다.
  - 마지막 반환 제이터를 삭제하므로 연이은 반복 호출을 허용하지 않는다.
- **void SetSortRule(List* plist, int (*comp)(LData d1, LData d2)); -** 정렬
  - 리스트에 정렬의 기준이 되는 함수를 등록한다.
- **int LCount(List* plist); -** 데이터의 수 검색
  - 리스트에 저장되어 있는 데이터의 수를 반환한다.

4. 단순 **LINKEDLIST**의 **ADT** 구현

구현 사항을 결정해야하는 과정이 필요하다.

1. **head**와 **tail**이 있는데 **tail**로 데이터가 추가되지 않다면 **tail**이 필요 없지 않을까**?**

   - **head**로 추가가 되는 것과 **tail**로 추가되는 것이 장단점은 존재한다**.**
   - **head**로 추가 했을 때 앞에서 부터 순서대로 들어가는 저장순서를 유지 할 수 없는 단점은 자료구조에서 필요 사항이 아니므로 **tail**을 사용해서 생가는 코드적인 코스트를 제거하는 것이 더 좋기때문에 **head**로 **Data**가 추가 되도록 한다**.**

2. 항상 첫 번째 **Node**상황과 두 번째 **Node** 상황이 나뉘지 않고 항상 같은 상황으로 만들 수 있지 않을까**?(**첫 번째 **Node**상황일때를 제외**)**

   - **Head**다음에 **DummyNode**를 추가하면 항상 유효한 데이터가 추가되는 **Node**가 구조상 두 번째 노드가 되므로 노드의 추가**,** 삭제 및 조회코드의 과정을 일관된 형태로 진행이 가능해진다**.**

4. 단순 LINKEDLIST의 ADT 구현

구현 하게 될 LinkedList의 모습

- 제공된 LinkedRead.cpp에 head로 추가되고 Dummy Node를 추가해서 아래와 같은 모습이 되게 바꿔보자.

4. 단순 LINKEDLIST의 ADT 구현

필요한 구조체를 선언

Node

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

Main에서 필요한 **List** 선언

- 단일 변수로 구현하게 되면 **List**가 여러 개일 때 어려움을 겪게 된다.

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

4. 단순 LINKEDLIST의 ADT 구현

헤더파일의 구현

```
#define TRUE1
#define FALSE0

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));
```

4. 단순 LINKEDLIST의 ADT 구현

초기화 구현

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

삽입 구현

```
Void LInsert(List* plist, Ldata data)
{
	if(plist -> comp == NULL) // 정렬기준이 설정되어 있지 않으면..(정렬 기준이 없을 수도 있다.)
		FInsert(plist, data); // 머리에 Node를 추가!
	else
		SInsert(plist, data); // 정렬기준에 근거하여 Node를 추가!
}
```

4. 단순 LINKEDLIST의 ADT 구현

FInsert 구현

```
void FInsert(List * plist, Ldata data)
{
	Node * newNode = (Node*)malloc(sizeof(Node));		// 새 노드를 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 가리키게 함

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

4. 단순 LINKEDLIST의 ADT 구현

- **head**가 **NULL**이 아닌 더미를 가리키고 이미 **4, 6**이라는 데이터가 있는 상태에서 **2**가 삽입되는 걸 가정.

FInsert(plist, 2); // **plist**는 리스트의 주소를 담고 있는 포인터 변수

- **Node**를 추가하는 함수를 실행하면 새로운 **Node**를 생성하고 **Node**에 데이터를 저장한다.

```
Node* newNode = (Node*)malloc(sizeof(Node));
newNode -> data = data;
```

1. **newNode->next = plist->head->next;** // **New Node**가 다른 **Node**를 가리키게 함.
2. **Plist->head->next = newNode;** // **Dummy Node**가 **New Node**를 가리키게 함.

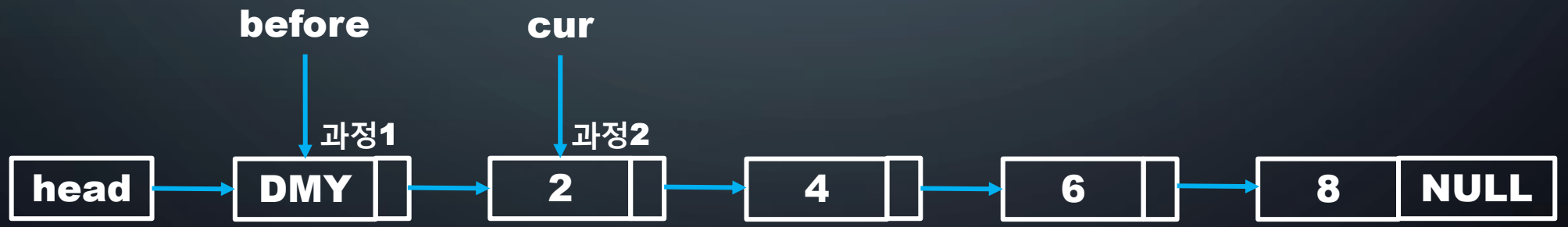
SInsert함수는 정렬 삽입 과정에서 함께

4. 단순 LINKEDLIST의 ADT 구현

조회

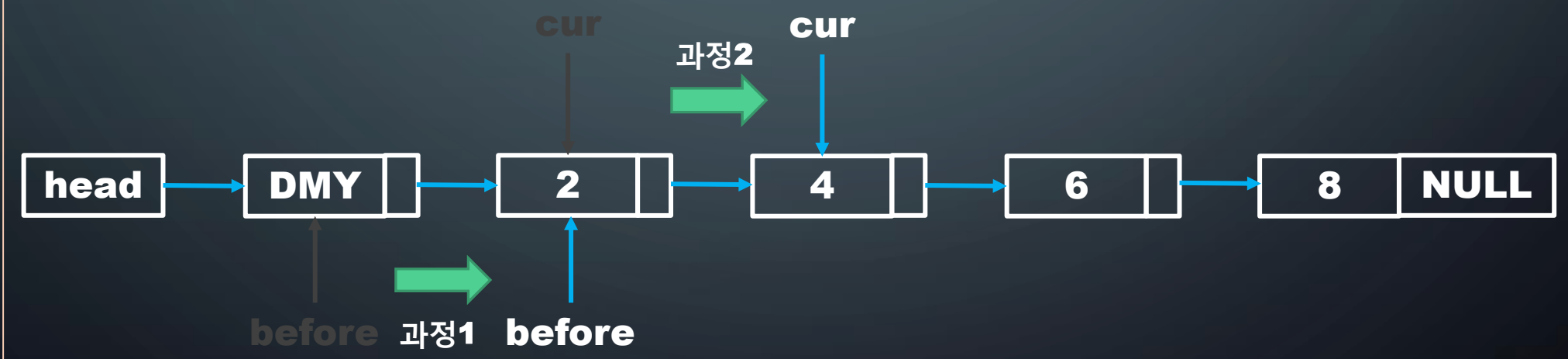
- **DummyNode** 다음에 오는 **Node**를 첫 번째 **Node**이다.

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

과정1 plist -> before = plist -> head;       // befor는 Dummy Node를 가리키게 함.
과정2 plist -> cur = plist -> head -> next;  // cur은 첫 번째 Node를 가리키게 함

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



4. 단순 LINKEDLIST의 ADT 구현

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

과정1Plist->before = plist->cur;           // cur이 가리키던 것을 befor가 가리킴
과정2Plist->cur = plist->cur->next;        // cur은 그 다음 Node를 가리킴

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



4. 단순 LINKEDLIST의 ADT 구현

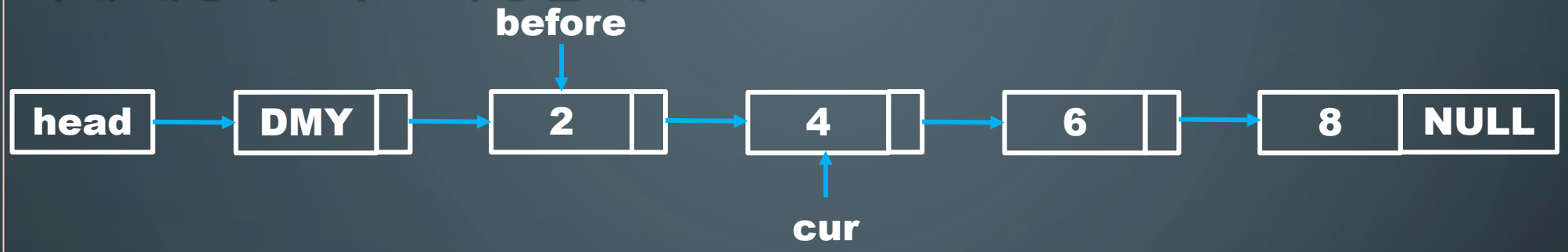
삭제

- **cur**의 값을 **before**의 값만 동일한 곳을 가리키게 만들어주면 **LNext**에 의해 **before**가 이동하기 때문에 삭제를 할 때 **before**의 이동처리는 하지 않아도 된다

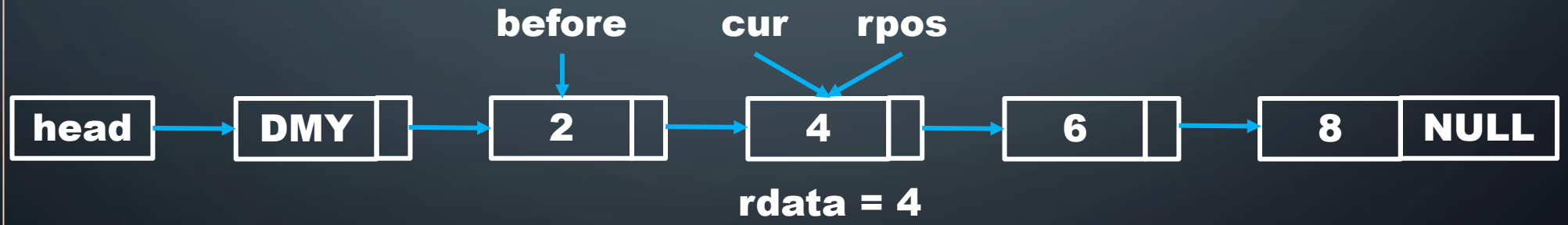
```
LData LRemove(List * plist)
{
    Node * rpos = plist->cur;                    // 소멸 대상의 주소 값을 rpos에 저장
    Ldata rdata = rpos->data;                    // 소멸 대상의 데이터를 rdata에 저장

    plist->before->next = plist->cur->next;      // 소멸 대상을 리스트에서 제거
    plist->cur = plist->before;                  // cur이 가리키는 위치를 재조정!

    free(rpos);                                  // 리스트에서 제거된 Node 소멸
    (plist->numOfData)--;                        // 저장된 데이터의 수 하나 감소
    return rdata;                                // 제거된 Node의 데이터 반환
}
```

4. 단순 LINKEDLIST의 ADT 구현

삭제 대상이 4라고 가정할 때

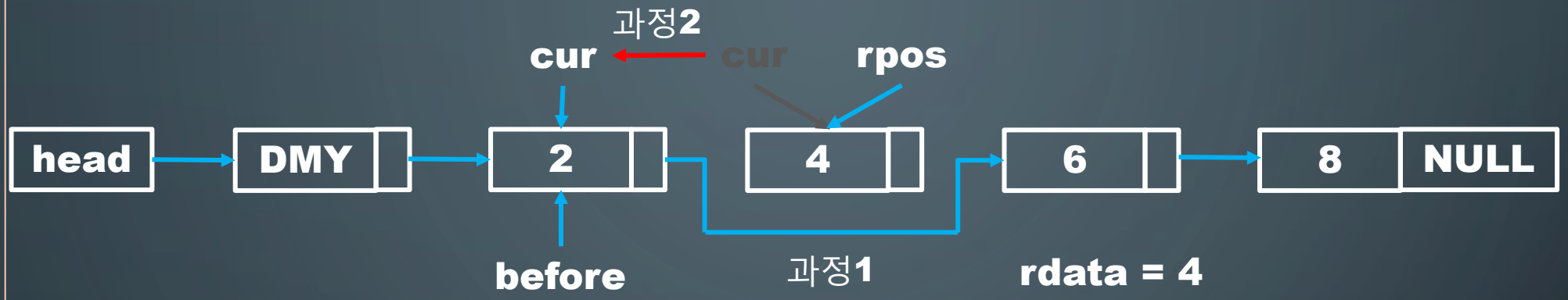



```
Node * rpos = plist->cur;      // 소멸 대상의 주소 값을 rpos에 저장
Ldata rdata = rpos->data;      // 소멸 대상의 데이터를 rdata에 저장
```

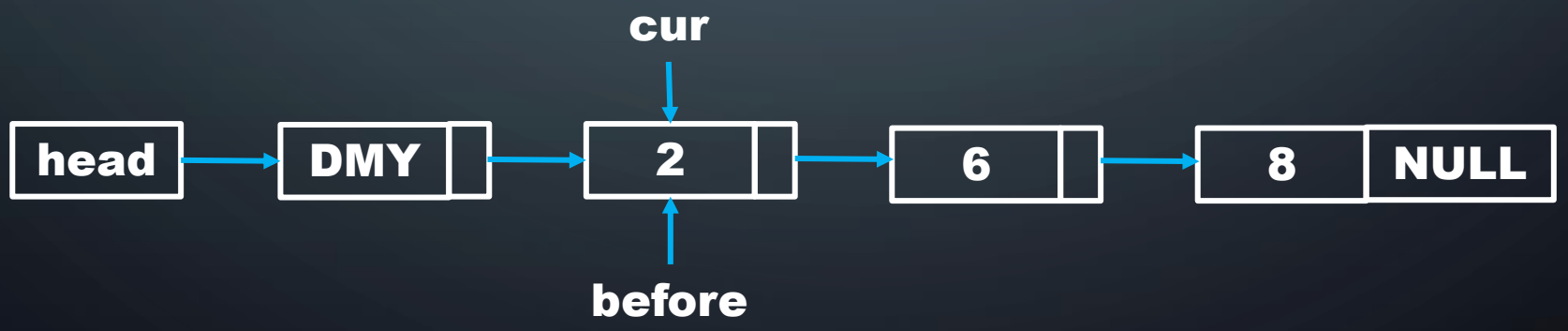


4. 단순 LINKEDLIST의 ADT 구현

```
과정1 plist->before->next = plist->cur->next;   // 소멸 대상을 리스트에서 제거
과정2 plist->cur = plist->before;               // cur이 가리키는 위치를 재조정!
```




```
free(rpos);                 // 리스트에서 제거된 Node 소멸
(plist->numOfData)--;       // 저장된 데이터의 수 하나 감소
return rdata;               // 제거된 Node의 데이터 반환
```



4. 단순 LINKEDLIST의 ADT 구현

```
int main(void)
{
	// List의 생성 및 초기화 ////////////////////////////
	List list;
	int data;
	ListInit(&list);

	// 5개의 데이터 저장 ////////////////////////////
	LInsert(&list, 11);  LInsert(&list, 11);
	LInsert(&list, 22);  LInsert(&list, 22);
	LInsert(&list, 33);

	// 저장된 데이터의 전체 출력 ////////////////////////
	printf("현재 데이터의 수: %d \n", LCount(&list));

	// 첫 번째 데이터 조회
	if (LFirst(&list, &data))
	{
		printf("%d ", data);

		// 두 번째 이후의 데이터 조회
		while (LNext(&list, &data))
			printf("%d ", data);
	}

	printf("\n\n");

	// 숫자 22을 검색하여 모두 삭제 ////////////////////////
	if (LFirst(&list, &data))
	{
		if (data == 22)
		LRemove(&list);

		while (LNext(&list, &data))
		{
			if (data == 22)
				LRemove(&list);
		}
	}

	// 삭제 후 남아있는 데이터 전체 출력 ////////////////////
	printf("현재 데이터의 수: %d \n", LCount(&list));

	if (LFirst(&list, &data))
	{
		printf("%d ", data);

		while (LNext(&list, &data))
			printf("%d ", data);
	}

	printf("\n\n");
	return 0;
}
```

4. 단순 LINKEDLIST의 ADT 구현

학습과제

앞선 과제인 **Point** 구조체를 이용한 활용 예제에 사용했던 **Point.h, Point.cpp, ArrayList.h, ArrayList.cpp, PointListMain.cpp** 중에서 **ArrayList.h, ArrayList.cpp**를 **DLinkedList.h, DLinkedList.cpp**로 대처해서 구현해 보자.

4. 단순 LINKEDLIST의 ADT 구현

정렬 삽입 구현

- **LinkedList**의 정렬기준이 되는 함수를 등록하는 **SetSortRule**함수
- **SetSortRule** 함수를 통해서 전달된 함수정보를 저장하기 위한 **LinkedList**의 멤버 **comp**
- **Comp**에 등록된 정렬기준을 근거로 데이터를 저장하는 **SInsert**함수

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

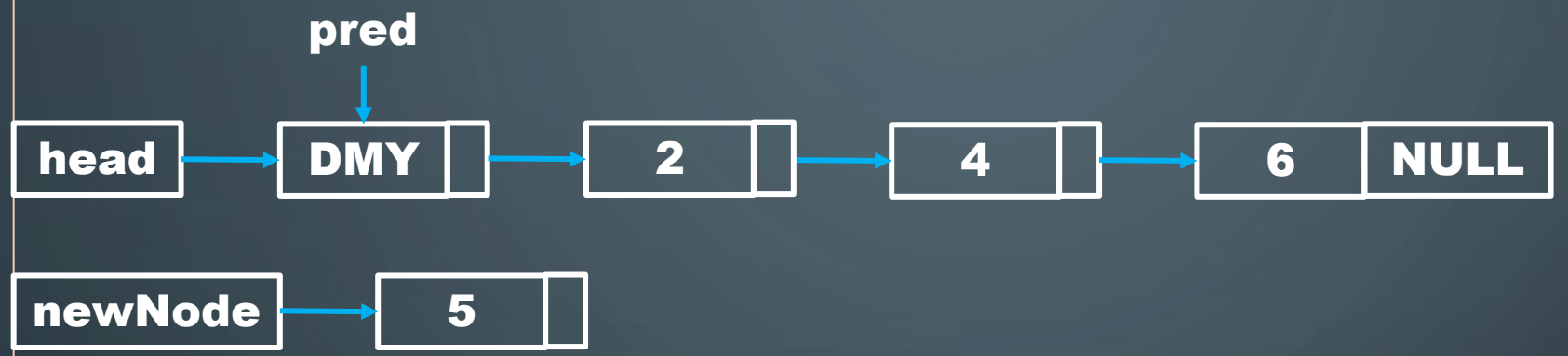
```
void SetSortRule(List* plist, int(*comp)(LData d1, LData d2))
{
    plist->comp = comp;
}
```

4. 단순 LINKEDLIST의 ADT 구현

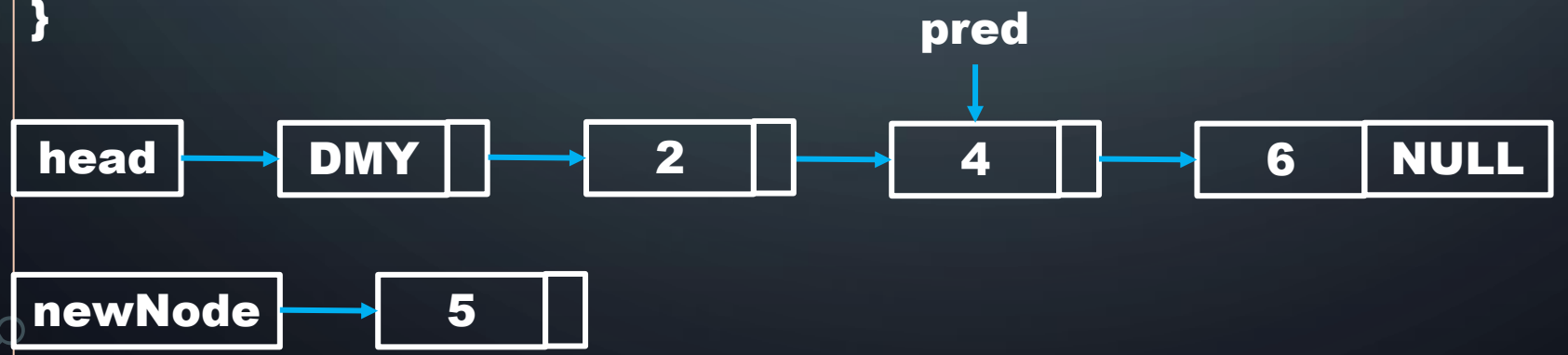
```
void SInsert(List * plist, LData data)
{
	Node * newNode = (Node*)malloc(sizeof(Node));  // New Node의 생성
	Node * pred = plist->head;                     // pred는 더미 노드를 가리킴
	newNode->data = data;                          // New Node에 데이터 저장

	//New Node가 들어갈 위치를 찾기 위한 반복문!
	While (pred->next != NULL & plist->comp(data, pred->next->data) != 0)
	{
		pred = pred->next;                     // Next Node로 이동
	}

	newNode->next = pred->next;                    // New Node의 오른쪽을 연결
	pred->next = newNode;                          // New Node의 왼쪽을 연결

	(plist->numOfData)++;                          // 저장된 데이터의 수 하나 증가
}
```

4. 단순 LINKEDLIST의 ADT 구현

```
Node * newNode = (Node*)malloc(sizeof(Node));      // New Node의 생성
Node * pred = plist->head;                         // pred는 더미 노드를 가리킴
newNode->data = data;                              // New Node에 데이터 저장
```



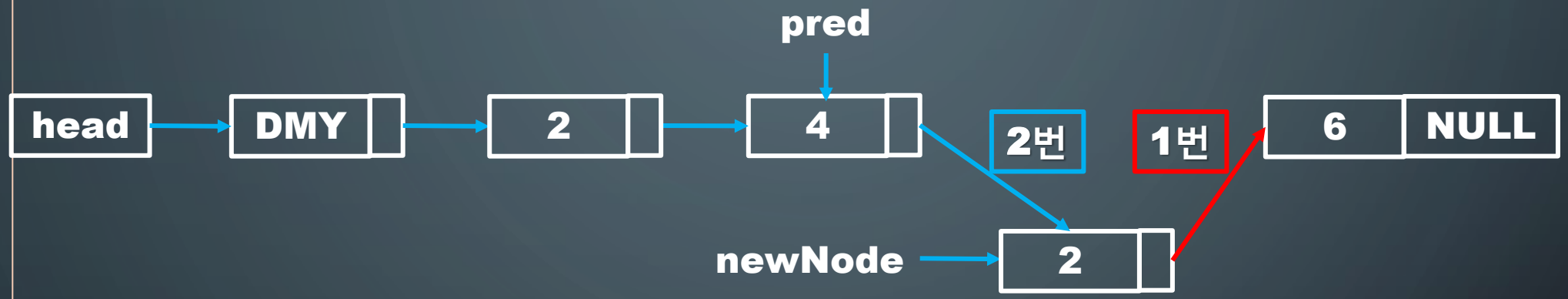

```
While(pred->next != NULL && plist->comp(data, pred->next->data) != 0)
{
    pred = pred->next;                             // Next Node로 이동
}
```



4. 단순 LINKEDLIST의 ADT 구현

조건1 **newNode->next = pred->next;** // **New Node**의 오른쪽을 연결
조건2 **pred->next = newNode;** // **New Node**의 왼쪽을 연결

(plist->numOfData)++; // 저장된 데이터의 수 하나 증가



SInsert함수의 While반복문

반복의 조건 **1 pred->next != NULL**

- **Pred**가 리스트의 **Last Node**를 가리키는지 묻기 위한 연산

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

- 새 데이터와 **pred**의 **Next Node**에 저장된 데이터의 우선순위 비교를 위한 함수호출
- **0**을 반환하면 **data**가 정렬순서상 앞서서 **head**에 더 가까워야 하고**, 1**을 반환하면 **pred->next->data**가 정렬순서상 앞서거나 같은 경우이다**.**

“pred가 마지막 Node를 가리키는 것도 아니고, 새 데이터가 들어갈 자리도 아직 찾지 못했다면 pred를 Next Node로 이동시킨다.”

4. 단순 LINKEDLIST의 ADT 구현

정렬 기준을 설정하는 함수 정의

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

- **SInsert**의 **While** 반복문에서 어떤 반환 값으로 표현 할지가 결정되면 함수의 정의 형태가 달라진다.
- **DLinkedList.cpp**에서는 정렬을 고정해서 계속 변경되는 코드를 만드는 것보다 정렬을 표현하고 결과를 반환해주는 형태로 표현해주면 그것에 맞게 정렬해주는 과정을 만들어주는 유연함을 보여줘야하기 때문에 **main.cpp**에 위치하는 것이 올바르다.

4. 단순 LINKEDLIST의 ADT 구현

학습과제

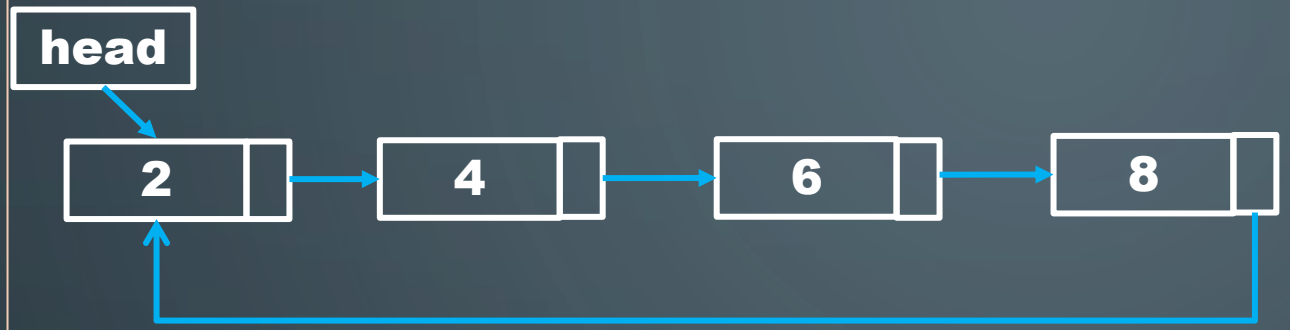
- 앞서 적용한 Point.h, Point.cpp를 이용한 DLinkedList.cpp, DLinkedList.h를 작성했다. 그것을 다음조건에 충족하도록 main함수를 적용해보자.

  조건1. x좌표의 값을 기준으로 오름차순 정렬이 되게 한다.
  조건2. x좌표의 값이 같은 경우에는 y좌표를 대상으로 오름차순 정렬이 되게 한다.

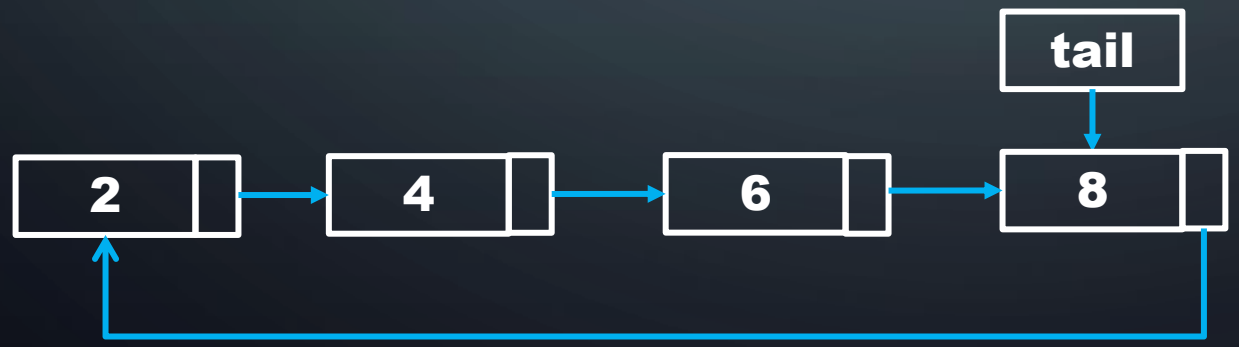
원형 연결 리스트(CIRCUAL LINKEDLIST)

4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

Last Node가 **First Node**를 가리키게 하는 리스트



- 해당 원형 연결 리스트는 시작점은 있지만 끝 마지막 지점을 특정할 수 없다.
- 무한으로 순회하면서 비교를 하는 것이 원형 연결 리스트의 특징이지만 시작점을 중심으로 마지막 지점을 중심으로 데이터를 추가할 때 문제가 발생한다.
- 마지막 지점을 위해 **tail**을 추가하게 되면 원형 연결 계속 돌면서 포인터 하나만 쓰기 위해 만들었던 것인데 결국 두 개를 쓰게 된다.
- **Head**대신 **tail**만 존재하게 된다면 **tail**이 마지막 지점이면서 **tail->next**가 시작점이 된다.



4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

```
#ifndef __C_LINKED_LIST_H__
#define __C_LINKED_LIST_H__

#define TRUE1
#define FALSE0

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);

//꼬리에 Node를 추가
void LInsert(List * plist, Data data);
//머리에 Node를 추가
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);

#endif
```

4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

초기화

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

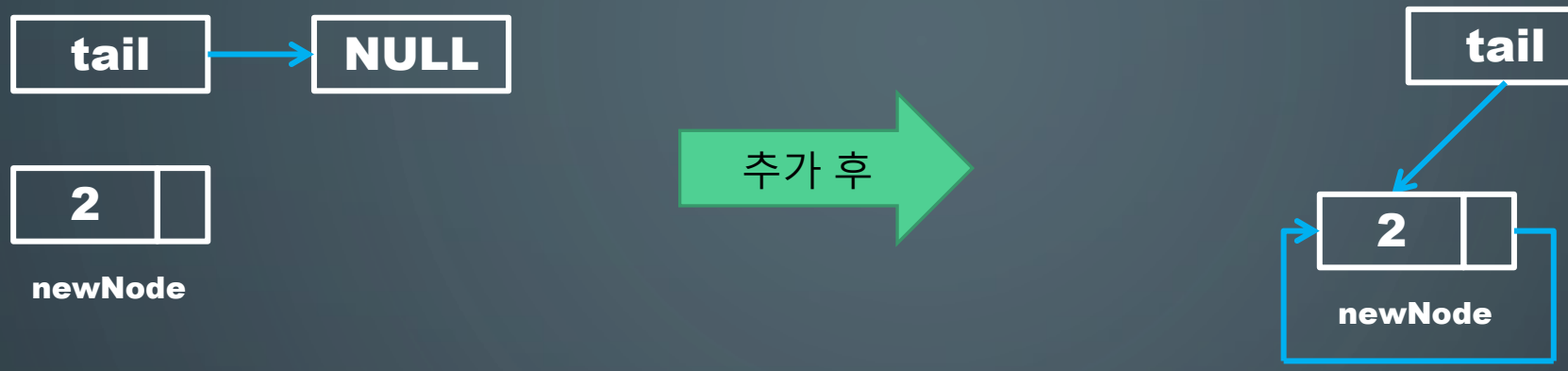
삽입

- 두 가지의 삽입 함수가 존재 하지만 첫 번째 Node 일 경우에는 동일한 처리를 하고 있다.

```
Node * newNode = (Node*)malloc(sizeof(Node));
newNode->data = data;

if (plist->tail == NULL)
{
	plist->tail = newNode;
	newNode->next = newNode;
}
else
{
	// 두 함수가 다른 부분..
}
(plist->numOfData)++;
```

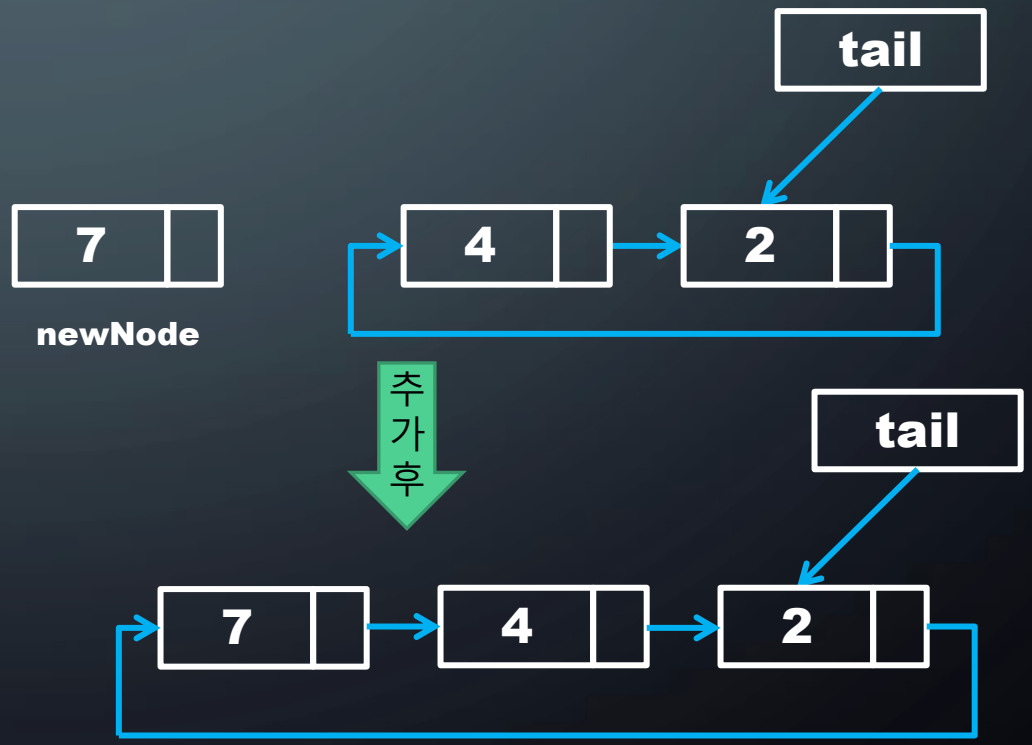
4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

Node를 처음 추가 할 때의 모습



머리에 Node를 추가 했을 때

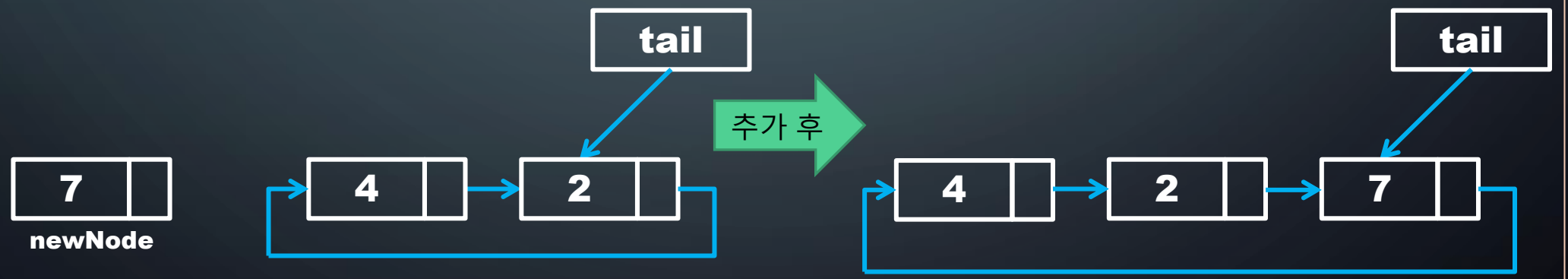
```
void LInsertFront(List * plist, Data data)
{
      .....공통부분 생략.....

      else
      {
            newNode->next = plist->tail->next;
            plist->tail->next = newNode;
      }

      .....공통부분 생략.....
}
```



4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

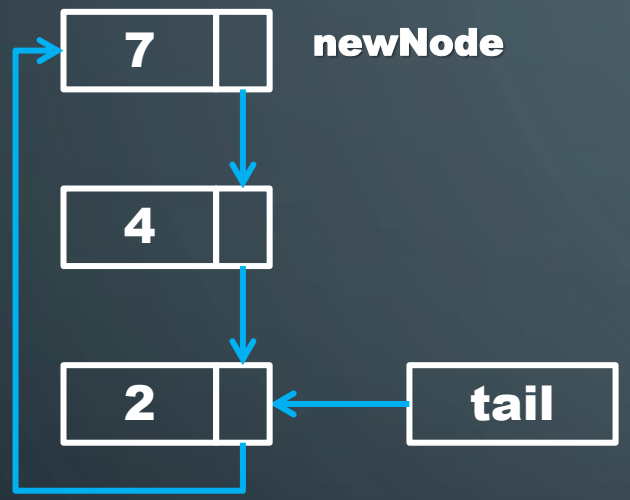
꼬리에 **Node**를 추가 했을 때

```
void LInsert(List * plist, Data data)
{
	else
	{
		newNode->next = plist->tail->next;
		plist->tail->next = newNode;
		plist->tail = newNode;
	}
}
```

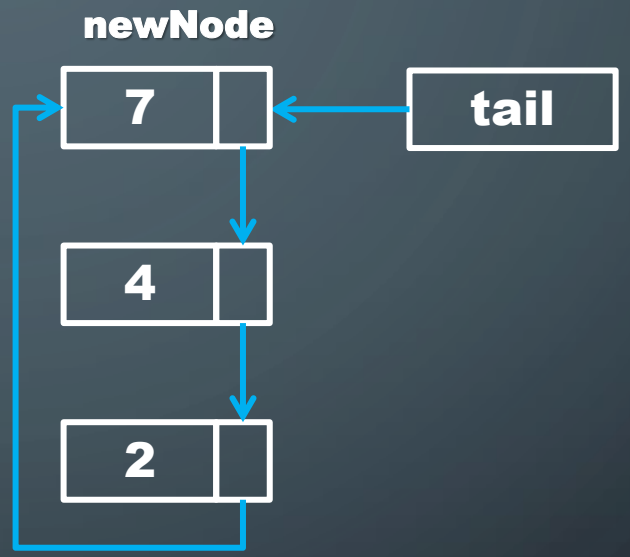


4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

머리추가와 꼬리추가의 비교



머리 추가



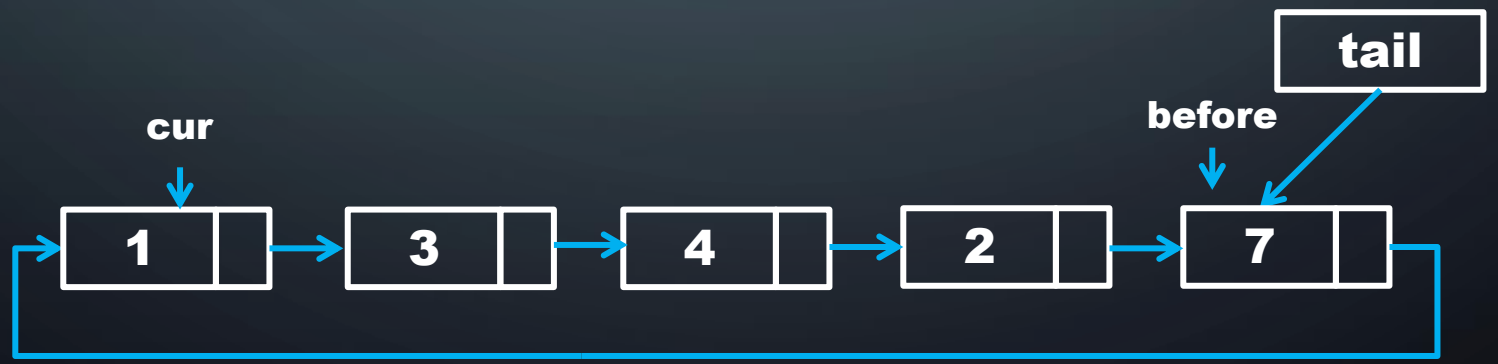
꼬리 추가

4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

조회

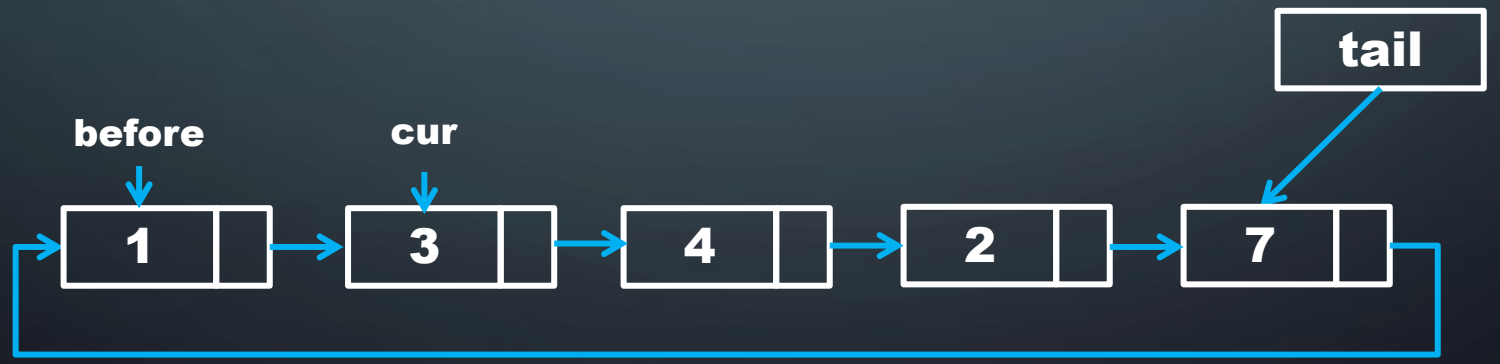
- **LFirst**를 이용한 초기화와 최초 참조 지점
- 단일 **LinkedList**의 구현부분과 달라지는 건 마지막 지점에 대한 처리

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

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

	*pdata = plist->cur->data;

	return TRUE;
}
```



4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

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

	plist->before = plist->cur;		// before가 Next Node를 가리키게 한다.
	plist->cur = plist->cur->next;		// cur가 Next Node를 가리키게 한다.

	*pdata = plist->cur->data;		// cur이 가리키는 Node의 데이터 반환
	return TRUE;
}
```

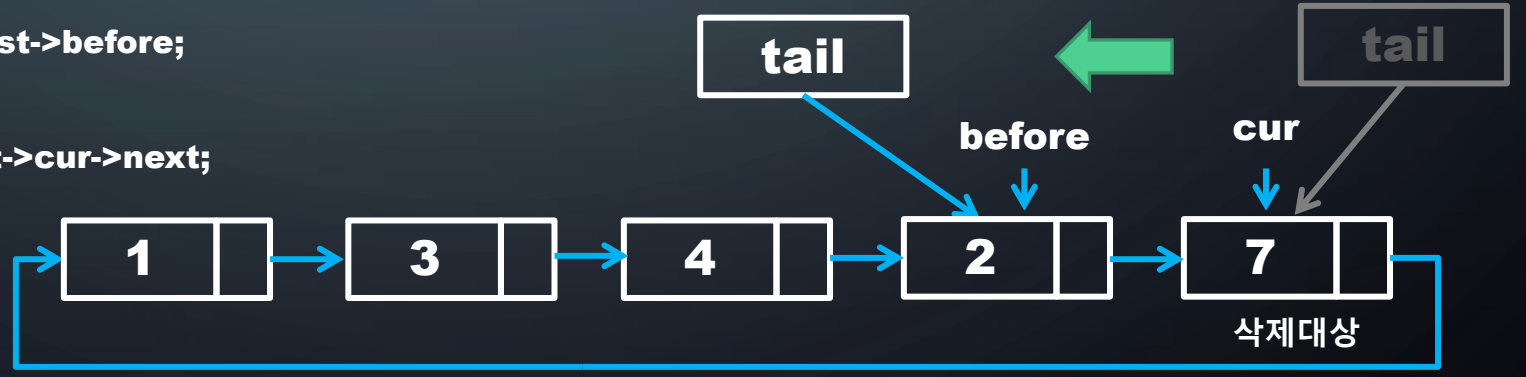


4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

삭제

- 단순 연결 리스트와 구조는 동일 하기 때문에 방법도 유사하다.
- 삭제할 **Node**의 **before Node**가, 삭제할 **Node**의 **Next Node**를 가리키게 한다.
- 포인터 변수 **cur**을 한 칸 뒤로 이동시킨다.
- 삭제할 **Node**를 **tail**이 가리키는 경우 **tail**이 다른 **Node**를 가리키게 해야 한다.
- 삭제할 **Node**가 **List**에 홀로 남은 경우 **tail**이 가리킬 **Node**가 존재하지 않기 때문에 **NULL**을 가리키게 한다.

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

	if(rpos == plist->tail)
	{
		if(plist->tail == plist->tail->next)
			plist->tail = NULL;
		else
			plist->tail = plist->before;
	}

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

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



4. 원형 연결 리스트(CIRCUAL LINKEDLIST)

학습과제

CircularLinkedList예제를 공부해보자
학습과제 폴더의 Chapter7_CircualLinkedList 문서를 참고하자.

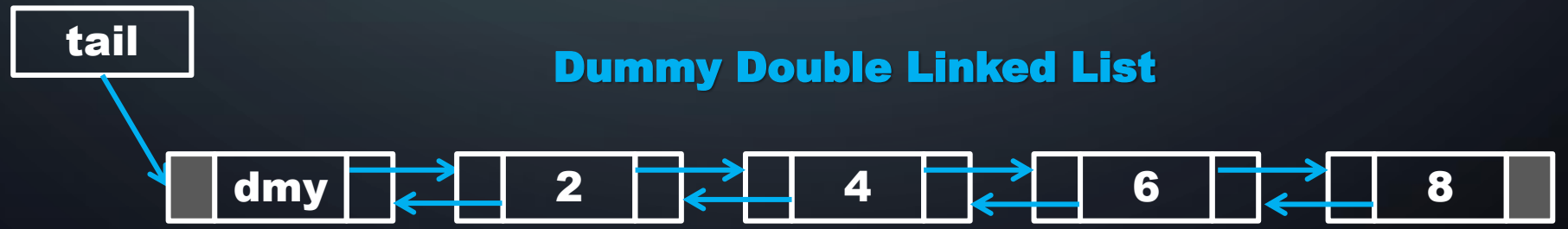
양방향 연결 리스트(DOUBLE LINKEDLIST)

4. 양방향 연결 리스트(DOUBLE LINKEDLIST)

- '**이중 연결 리스트**'라고 부르기도 한다.
- 왼쪽 Node가 오른쪽 Node를 가리킴과 동시에 오른쪽 Node도 왼쪽 Node를 가리키는 구조이다.
- 단순 연결 리스트에서 참조를 위해 썼던 before을 생략할 수 있다.

Node 구조체

```
Typedef struct _node
{
    Data data;
    struct _node* next;        //오른쪽 Node를 가리키는 포인터 변수
    struct _node* prev;        //왼쪽 Node를 가리키는 포인터 변수
}Node;
```



4. 양방향 연결 리스트(DOUBLE LINKEDLIST)

단순 연결 리스트

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

	plist->before = plist->cur;		// cur이 가리키는 위치를 재조정!
	plist->cur = plist->cur->next;

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

양방향 연결 리스트

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

	plist->before = plist->cur;		// Node가 서로 가리키고 있기때문에 필요 없다.
	plist->cur = plist->cur->next;
	*pdata = plist->cur->data;

	return TRUE;
}
```

4. 양방향 연결 리스트(DOUBLE LINKEDLIST)

```
#define TRUE1
#define FALSE0

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);

//Node의 왼쪽으로 이동해서 해당 Node를 참조, Data반환
int LPrevious(List * plist, Data * pdata);

int LCount(List * plist);
```

4. 양방향 연결 리스트(DOUBLE LINKEDLIST)

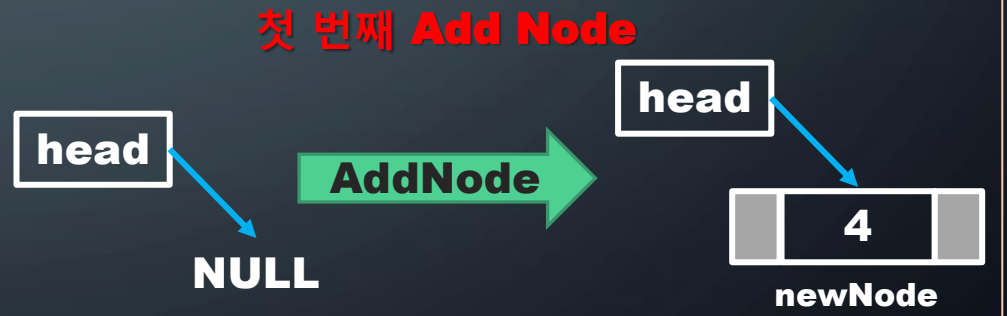
초기화

- **cur**은 **LFirst**함수가 호출됨과 동시에 초기화된다.

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

삽입

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

	//NewNode의 next를 NULL로 초기화
	newNode->next = plist->head;

	//두 번째 이후의 Node를 추가할 때만 실행
	if (plist->head != NULL)
		plist->head->prev = newNode;

	newNode->prev = NULL; //NewNode의 prev를 NULL로 초기화
	plist->head = newNode; //포인터 변수 head가 NewNode를 가리키게 한다.

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

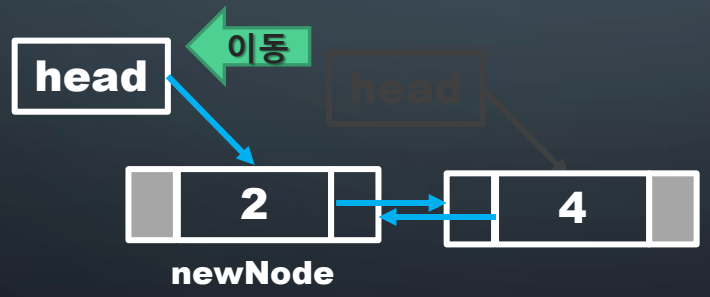


4. 양방향 연결 리스트(DOUBLE LINKEDLIST)

```
newNode->next = plist->head;        // 1단계 – newNode가 기존 Node를 가리키게 한다.
plist->head->prev = newNode;        // 2단계 – 기존 Node가 newNode를 가리키게 한다.
```

두 번째 이후의 Add Node 1단계

두 번째 이후의 Add Node 2단계



4. 양방향 연결 리스트(DOUBLE LINKEDLIST)

조회

```
int LFirst(List * plist, Data * pdata)          // 첫 번째 Node의 데이터 조회
int LNext(List * plist, Data * pdata)           // 두 번째 이후의 Node 데이터 조회
Int Lprevious(List * plist, Data * pdata)       // LNext의 반대 방향으로 데이터 조회
```

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

        plist->cur = plist->head;               // cur이 첫 번째 Node를 가리키게 함
        *pdata = plist->cur->data;              // cur이 가리키는 Node의 데이터 반환

        return TRUE;
}
```

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

        // cur을 오른쪽으로 이동
        Plist->cur = plist->cur->next;
        // cur이 가리키는 Node의 데이터 반환
        *pdata = plist->cur->data;

        return TRUE;
}
```

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

        // cur을 왼쪽으로 이동
        plist->cur = plist->cur->prev;
        // cur이 가리키는 노드의 데이터 반환
        *pdata = plist->cur->data;

        return TRUE;
}
```

4. 양방향 연결 리스트(DOUBLE LINKEDLIST)

학습과제

- 예제폴더에 있는 DoubleLinkedList를 공부해보자.
- 학습과제 폴더에 있는 문제를 해결해 보자(Chapter7_DoubleLinkedList 확인)