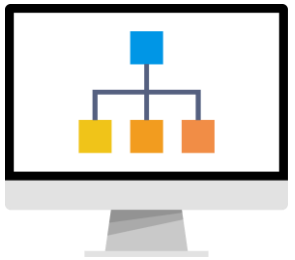


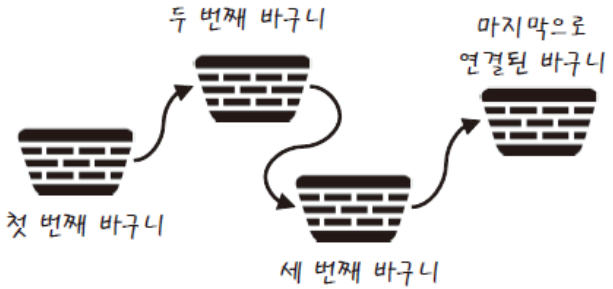
Data Structure & Algorithm
자료구조 및 알고리즘

6. 연결 리스트 (Chapter 4, Linked List part 2)

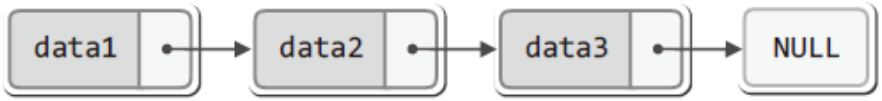
"연결" 리스트

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

일종의 바구니, 연결이 가능한 바구니



▶ [그림 04-1 : 노드의 표현]



▶ [그림 04-2: 노드의 연결]

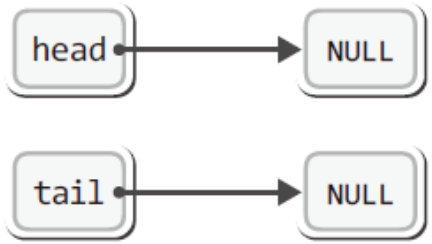
예제 LinkedRead.c의 분석을 시도 바랍니다!

예제 LinkedRead.c의 분석: 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;
    . . . .
}
```

LinkedRead.c의 일부

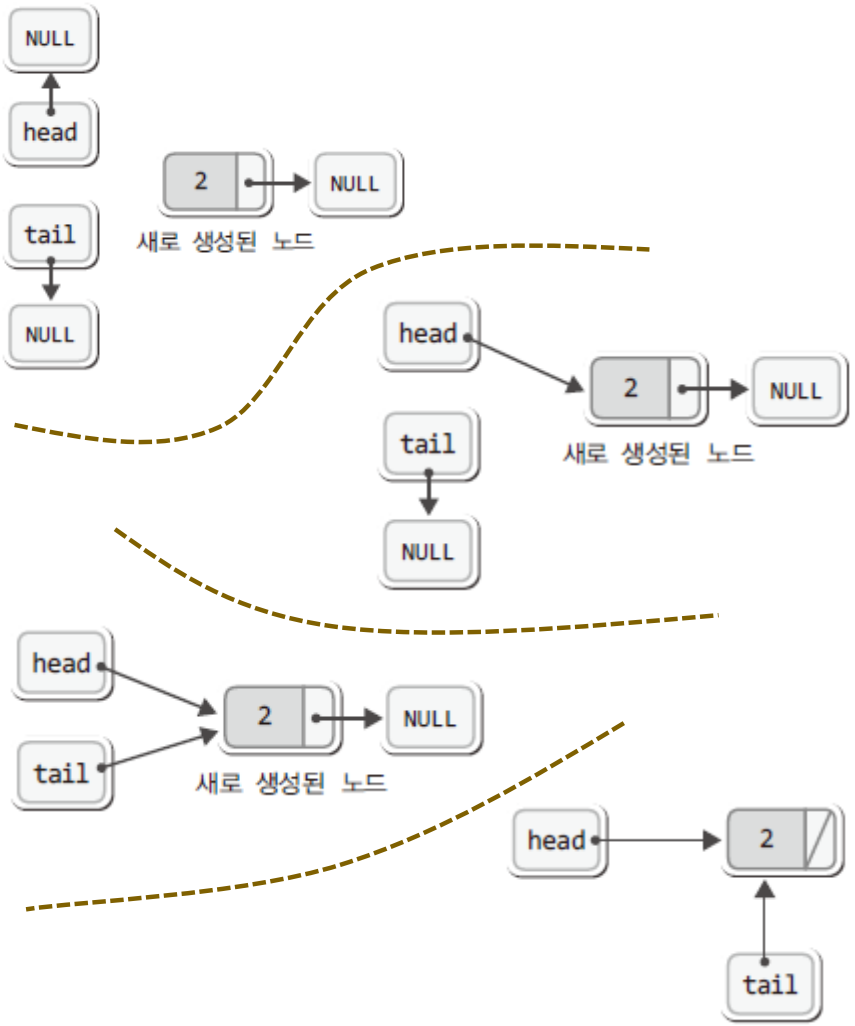
- head, tail, cur이 연결 리스트의 핵심!
- head와 tail은 연결을 추가 및 유지하기 위한것
- cur은 참조 및 조회를 위한것



예제 LinkedRead.c의 분석: 삽입 1회전

```
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;
}
```

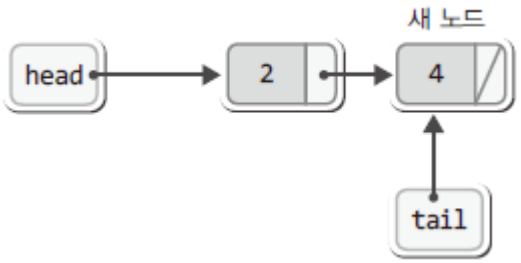
LinkedRead.c의 일부

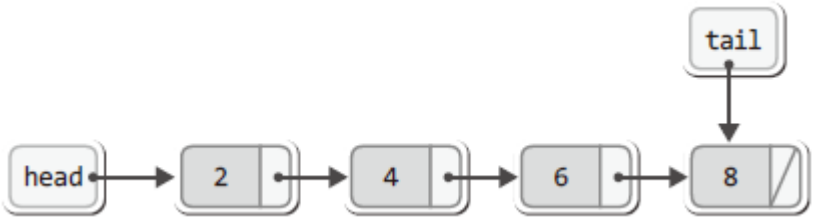


예제 LinkedRead.c의 분석: 삽입 2회전

```
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.c의 일부



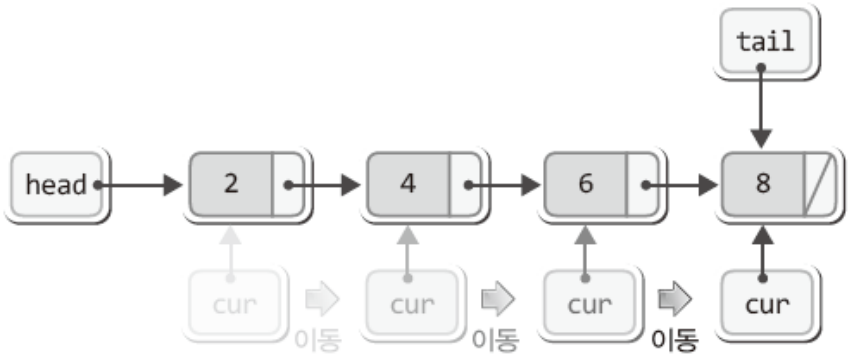


다수의 노드를 저장한 결과

예제 LinkedRead.c의 분석: 데이터 조회

전체 데이터의 출력 과정

```
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);
    }
}
```



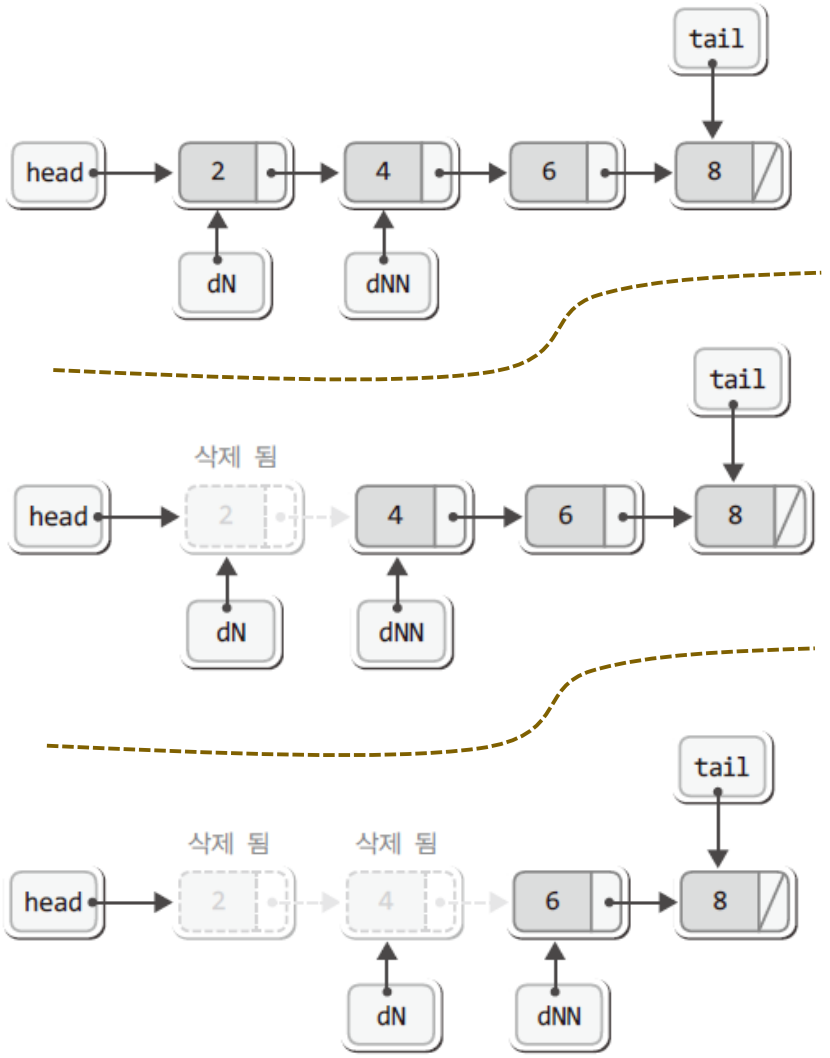
LinkedRead.c의 일부

예제 LinkedRead.c의 분석: 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);
    }
}
```

전체 노드의 삭제 과정

LinkedRead.c의 일부



정렬 기능 추가된 연결 리스트

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

이전과 동일

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

이전과 동일

- `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

이전과 동일

- `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

이전과 동일

정렬 기능 추가된 연결 리스트

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

이전과 동일

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

이전과 동일

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

SetSortRule 함수는 정렬의 기준을 설정하기 위해 정의된 함수! 이 함수의 선언 및 정의를 이해하기 위해서는 '함수 포인터'의 대한 이해가 필요하다.

함수 포인터

- 변수에 대한 포인터를 쓰는 이유?
  - 어떤 변수에 접근해서 값을 변경하고 싶다.
  - 대상이 되는 “어떤 변수”를 가리키기 위해
  - set5(&a); set5(&my_variable); set5(&arr[i]);

- 함수에 대한 포인터를 쓰는 이유?
  - 어떤 함수를 호출해서 반환 값을 얻고 싶다.
  - 대상이 되는 “어떤 함수”를 가리키기 위해

함수 포인터

- 변수에 대한 포인터를 생각했다.
- 함수에 대한 포인터도 생각해보자.
- 정수를 반환하는 함수에 대한 포인터 *f*
  - int (*f)(void);

- 정수에 대한 포인터를 반환하는 함수 *f*
  - int *f(void) { … }

연산자		결합성
() [] -> .		left to right
! ~ ++ -- + - * & (*type*) sizeof		**right to left**
* / %	산술연산자	left to right
+ -		left to right

함수 포인터

```c
#include <stdio.h>

void print_hello() {
    printf("Hello\n");
}

void print_bye() {
    printf("Bye\n");
}

void call(void (*f)()) {
    f();
}

int main(){
    call(print_hello);
    call(print_bye);

    void (*f)() = print_hello;
    call(f);

    return 0;
}
```

리스트에서의 함수 포인터

- 그냥 작은 수가 앞에 오도록 정렬하는 코드를 리스트 요소 추가 코드에 넣으면 안되나요?
  - 큰 수가 앞에 오도록 정렬하고 싶다면?
  - Point 구조체와 같이 대소를 비교하는 기준이 명확하지 않다면?

- 사용자가 “정렬 우선순위를 결정하는 함수”를 자유롭게 등록할 수 있도록 함으로써 유연성을 제공하자.

SetSortRule 함수 선언에 대한 이해

```
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)

```
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.
}
```

인자로 전달이 가능한 함수의 예

정렬의 기준을 결정하는 함수

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

int cr = WhoIsPrecede(D1, D2);

cr에 저장된 값이 0이라면

head . . . D1 . . D2 . . . tail D1이 head에 더 가깝다.

cr에 저장된 값이 1이라면

head . . . D2 . . D1 . . . tail D2가 head에 더 가깝다.

새 노드의 추가 위치에 따른 장점과 단점

새 노드를 연결 리스트의 머리에 추가하는 경우

- 장점 포인터 변수 tail이 불필요하다.
- 단점 저장된 순서를 유지하지 않는다.

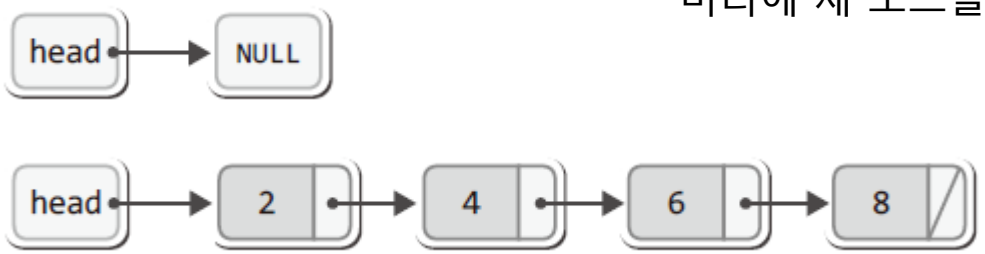
새 노드를 연결 리스트의 꼬리에 추가하는 경우

- 장점 저장된 순서가 유지된다.
- 단점 포인터 변수 tail이 필요하다.

두 가지 다 가능한 방법이다. 다만 tail의 관리를 생략하기 위해서 머리에 추가하는 것을 원칙으로 하자!

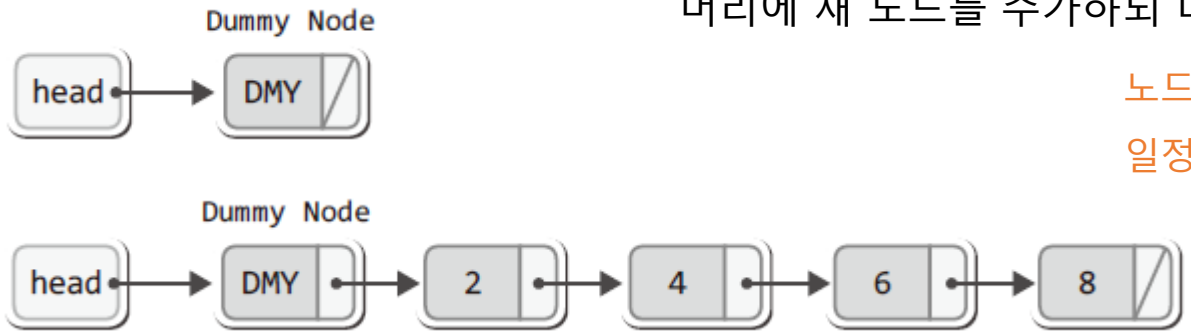
더미 노드 기반 연결 리스트

머리에 새 노드를 추가하되 더미 노드 없는 경우



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

머리에 새 노드를 추가하되 더미 노드 있는 경우



노드의 추가 및 삭제 방식이 항상 일정하다.

LinkedRead.c와 문제 04-2의 답안인 DLinkedRead.c를 비교해보자!

정렬 기능 추가된 연결 리스트의 구조체

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

노드의 구조체 표현

연결 리스트에 필요한 변수들은
구조체로 같이 묶어야 한다.

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

```
#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;
```

앞부분

```
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));
```

뒷부분

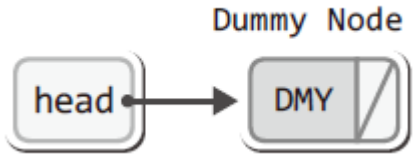
더미 노드 연결 리스트 구현: 초기화

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

초기화 함수의 정의를 위해서
살펴봐야 하는 구조체의 정의

초기화 함수의 정의

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



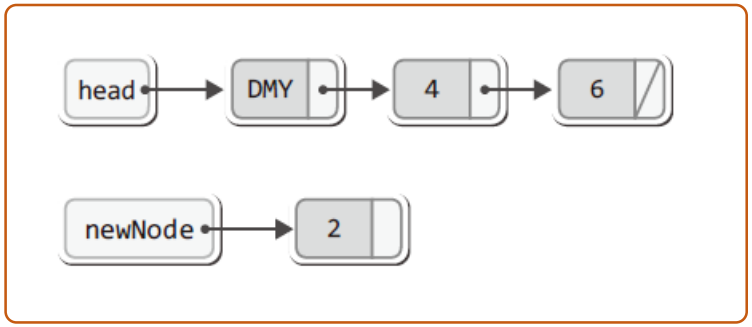
더미 노드 연결 리스트 구현: 삽입1

```
void LInsert(List * plist, LData data)
{
    if(plist->comp == NULL)          // 정렬기준이 마련되지 않았다면,
       FInsert(plist, data);         // 머리에 노드를 추가!

    else                            // 정렬기준이 마련되었다면,
       SInsert(plist, data);        // 정렬기준에 근거하여 노드를 추가!
}
```

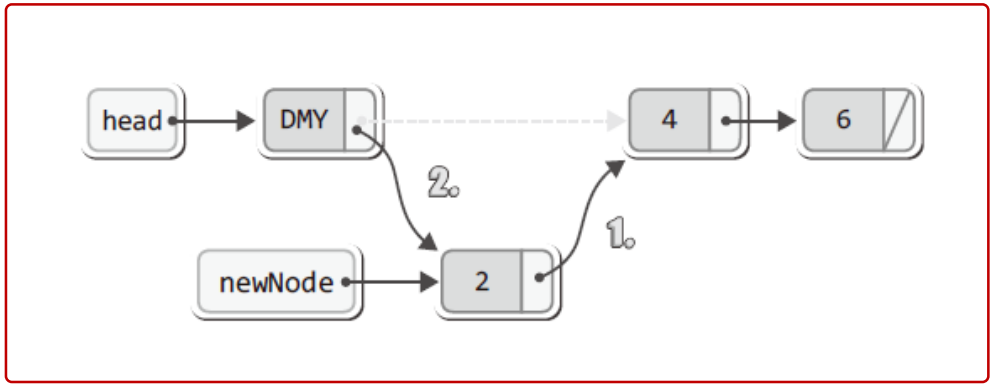
```
void FInsert(List * plist, LData data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));   // 새 노드 생성
    newNode->data = data;                           // 새 노드에 데이터 저장

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

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

더미 노드 연결 리스트 구현: 삽입2

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

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

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



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



요약

- 연결 리스트는 배열 리스트와 다르게 동적 할당을 이용하여 가변 길이의 데이터를 담을 수 있다.
  - 데이터 삽입시 관련된 요소의 next 포인터 값을 잘 바꾸어야 리스트가 끊어지지 않는다!

- 리스트의 맨 앞에 항상 더미 노드를 둠으로써 삽입시 head 포인터를 변경하는 수고를 줄일 수 있다.

출석 인정을 위한 보고서 작성

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

1) 배열 리스트와 연결 리스트의 장단점을 비교하시오.

2) 아래 FInsert 함수에서 아래 별표로 표시한 두 줄이 바뀌면 어떤 문제가 발생하는지 예를 들어 보이시오.

```
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 가리키게 함

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