중간고사 1번 보고서

Create, Search, Insert, Remove ADT를 head와 tail 포인터를 사용하는 linked list에 구현하기 전에 먼저 ADT(Abstract Data Type)가 무엇인지 알아야 합니다.

ADT는 구체적인 기능의 완성과정을 언급하지 않고, 순수하게 기능이 무엇인지 나열한 것을 말합니다. 저희가 배우는 책에서는 지갑을 예로 들어 코드로 설명하기도 하였습니다. 그런데 ADT라 부르는 이유는 컴퓨터 공학적 측면에서 어떠한 자료형을 단순히 메모리 구조로 정의 내리지 않고, 그러한 메모리 구조를 기반으로 하는 연산의 종류를 결정하였을 때, 해당 자료형의 정의가 완성된다고 보기 때문이라 책에서 서술하였습니다. 즉, '자료형'의 정의에서 '기능' 혹은 '연산'과 관련된 내용을 명시할 수 있다는 것입니다. 책과 달리 제가 다시 한 번 간단히 설명하면, 어떤 제품의 사용 설명서와 같은 것이 'ADT'라는 것입니다. 해당 자료형에 대한 설명서 정도로 생각하면 될 것입니다. 사용 설명서에는 사용할 때 필요한 정보만 담고 있지 그 외에 필요 없는 내부 구현 같은 것들은 포함되어 있지 않다는 것입니다.

List는 데이터구조에서 구현 방법에 따라 크게 2가지로 나뉘게 되는데 그 중 linked list는 메모리 동작 할당을 기반으로 구현된 리스트로 일반 순차List와 다르게 길이에 제약이 걸려있지 않고 데 데이터 추가될 때 중복된 데이터의 저장을 막지 않고, 삭제 같은 기능에는 용이합니다. 하지만 데이터 참조가 어렵다는 단점이 있습니다.

제가 구현하는 'linked list'는 '단일 linked list' 입니다. 가장 기본이 되는 연결 리스트입니다. 'Linked list' 에서는 노드에는 다음 노드를 가리킬 포인터 변수와 해당 데이터 변수가 존재합니다.

```
typedef struct _node       // 기본 구조체
{
   int data;               // 데이터 변수
   struct _node * next;    // 연결 도구
} Node;
```

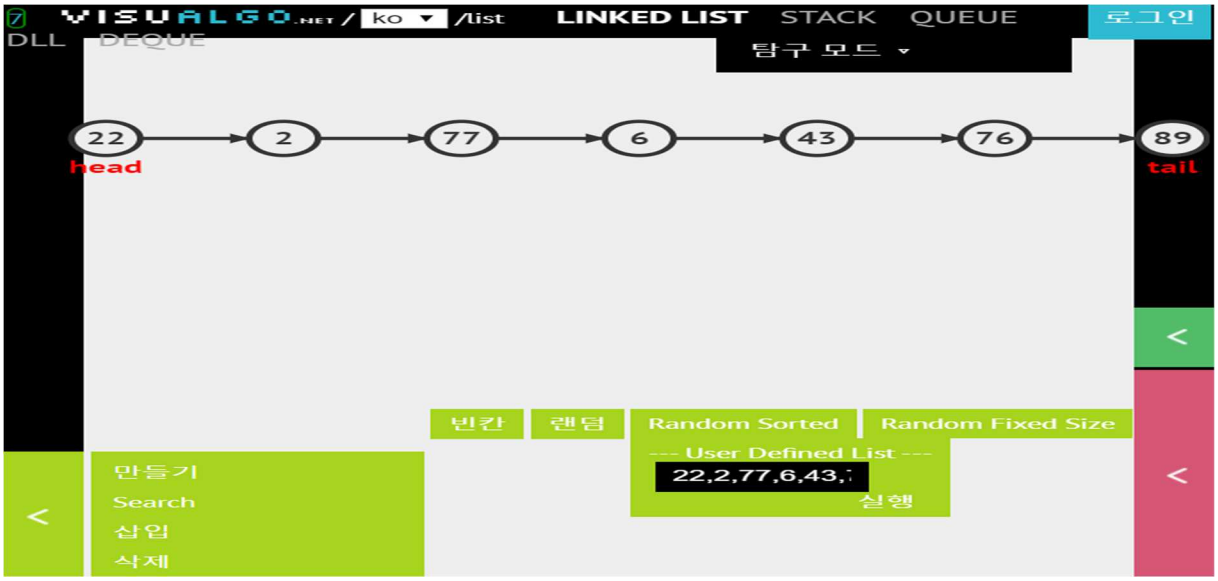
기본 노드 선언

```
typedef struct _list  // 구조체 선언
{
   struct _node* cur;  //위치참조 포인터변수
   struct _node* head; //머리를 나타내는 포인터 변수
   struct _node* tail; //꼬리를 나타내는 포인터 변수
   int numOfData;
} List;
```

구조체 선언

저 같은 경우 교재내용을 참고하여 제일 먼저 각 데이터를 연결할 '노드' 를 선언하고 바로 구조체 기반 연결리스트를 선언했습니다. 앞에서도 언급했던 것처럼 연결리스트는 각 노드로 연결시킨 구조체기반 리스트입니다. 이제 visualgo사이트로 구현한 ADT 연결리스트를 코드와 그림으로 설명해 보겠습니다.

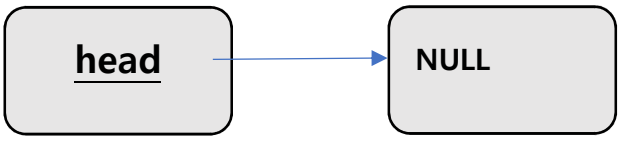
1.생성(Create)

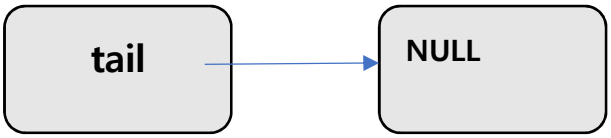


위 사진의 좌측 하단에 주요기능 4가지 중 제일 첫번째에 있는 만들기 즉 배열을 생성에 대해 설명해보겠습니다.

Create는 저희 교재에 있는 Listinit과 같은 개념입니다. 즉 연결리스트를 생성하기 전에 모든 노드 포인터들을 NULL로 초기화 시키는 것입니다. 그리고 저는 이번 더미노드를 사용하지 않았습니다. 왜냐하면 제가 visualgo로 연결리스트를 구현해보았을 때 더미노드를 쓰지 않았기 때문입니다.

```
void Create(List* plist)
{
    Node* head = NULL;      // NULL 포인터 초기화, 리스트 머리를 가르키는 포인터변수
    Node* tail = NULL;      // NULL 포인터 초기화, 리스트 꼬리를 가르키는 포인터변수
    Node* cur = NULL;       // NULL 포인터 초기화, 저장된 데이터의 조회하는 포인터변
    plist->numOfData = 0;   // 데이터 수
}
```





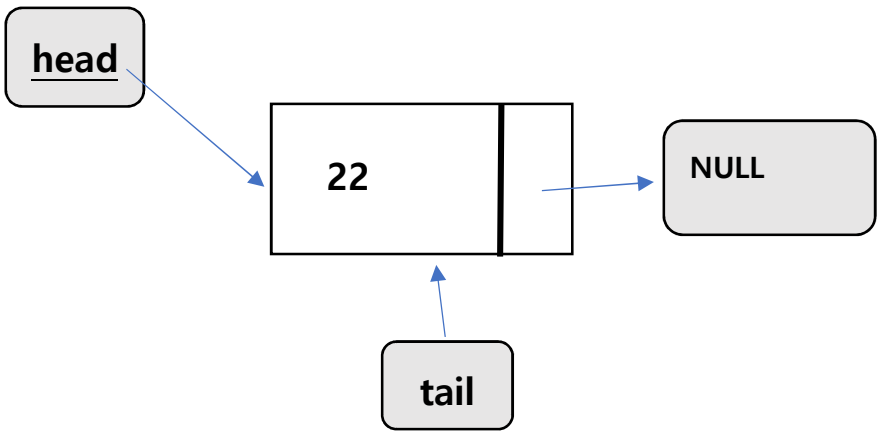
위 그림처럼 각 포인터 변수 머리(head)와 꼬리(tail)를 초기화 시킵니다. 이 머리와 꼬리, cur 그리고 나중에 나올 tmp는 연결리스트에서 핵심이라 말할 수 있습니다.

교재에서는 첫 데이터추가함수, 정렬조건이 있는 데이터 추가함수로 나누어 배열을 생성했습니다. 하지만 전 visualgo사이트에서 생성 버튼은 한 번에 변수 초기화와 연결리스트의 데이터를 추가하는 식입니다. 그래서 저는 따로 add함수를 구현하여 설명하겠습니다.

```
/***추가함수(데이터 추가)*****/
void Add(List* plist, int pdata)
{
    Node* newNode = NULL; // NULL 포인터 초기화, 추가되는 노트의 포인터변수
    newNode = (Node*)malloc(sizeof(Node));
    newNode->data = pdata;
    newNode->next = NULL;
    if (plist->head == NULL && plist->tail == NULL) //머리 꼬리가 비어있을 경우
        plist->head = plist->tail = newNode;
    else
    {
        plist->tail->next = newNode;
        plist->tail = newNode;
    }
    (plist->numOfData)++;
}
```

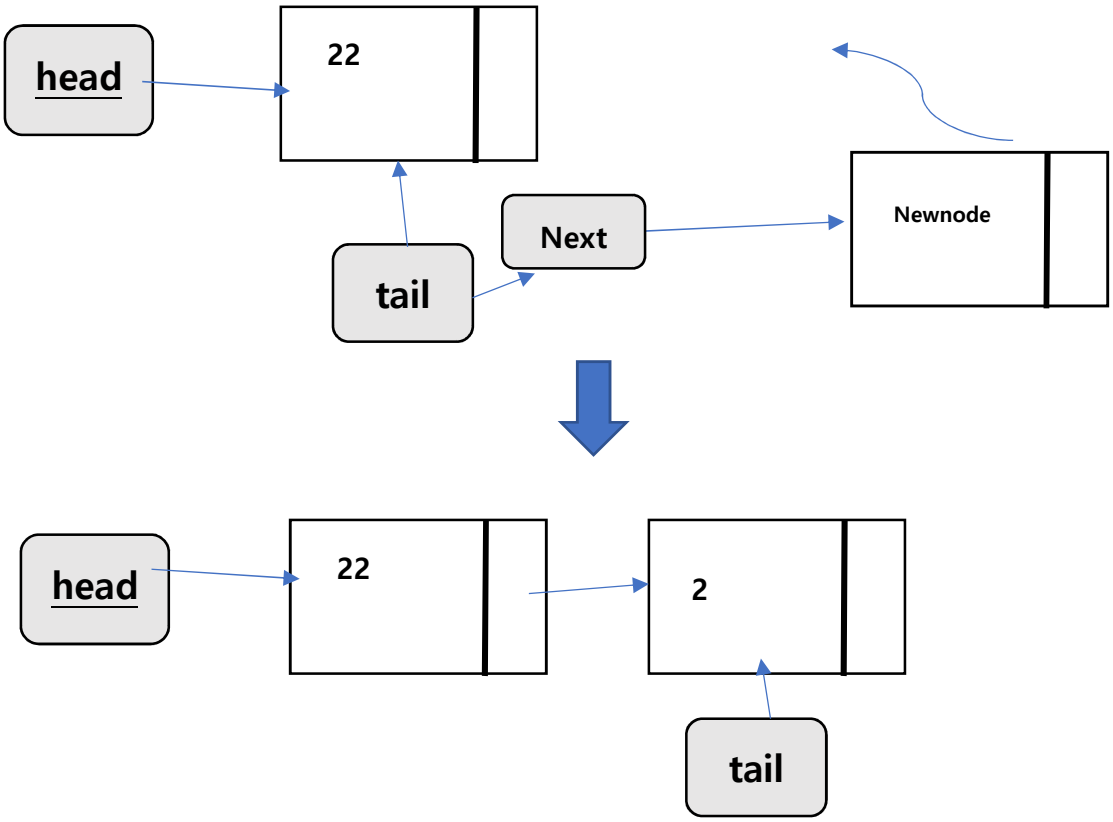
저희가 구현하는 연결리스트는 길이를 따로 정하지 않고 바로 그때 그때 메모리가 할당해주는 동적할당을 사용합니다. 먼저 새로 추가될 newNode란 노드을 앞에서 머리와 꼬리가 했던 것처럼 널로 초기화 시킵니다. 그 다음 동적할당을 시켜주고 그 newNode를 데이터를 가르키게 하면 즉 데이터 값을 넣게끔 선언합니다. 그리고 저는 next란 선언해서 데이터가 연결리스트에 들어갈 때 일어날 경우를 대비했습니다.

맨 처음 데이터를 넣을 경우 if조건문을 사용하여 머리와 꼬리가 비여있을 경우 그 데이터를 머리와 꼬리가 향하도록 설정했습니다.

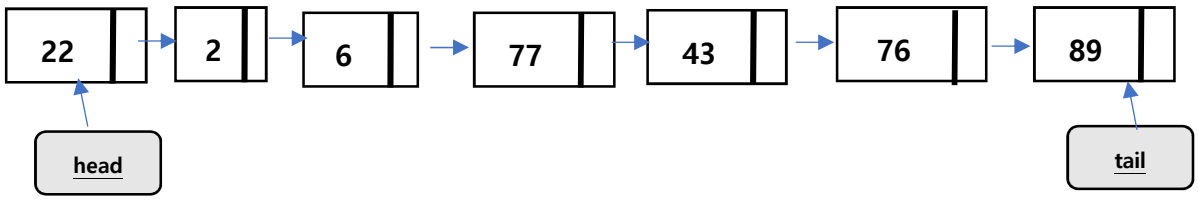


(첫 데이터가 생성될 경우)

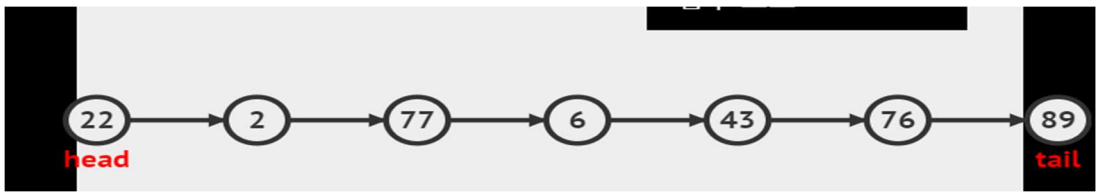
그 외의 경우에는 추가도 생성되는 데이터들을 꼬리의 그 다음 즉 next가 받도록 하였습니다.



(이 후 데이터가 생성될 경우)



이런 식으로 위 visualgo에서 생성된 연결리스트를 생성합니다.



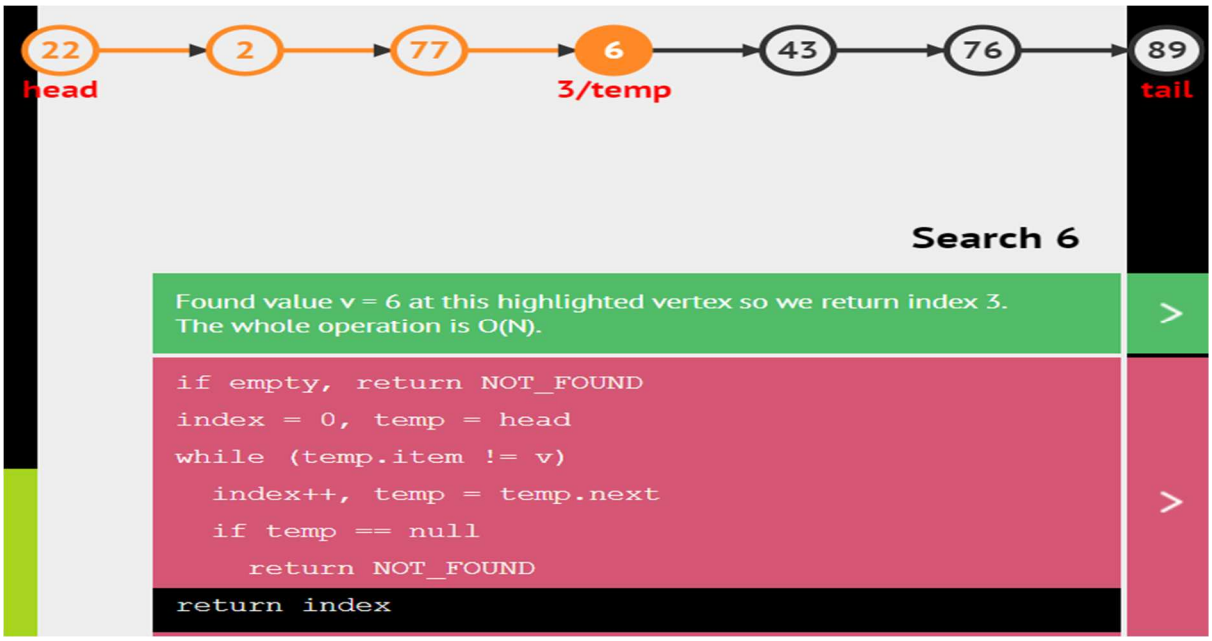

```
printf("노드 생성 전 초기화합니다. (Create 구현)₩n₩n");
Create(plist); //생성
Add(plist, 22);
Add(plist, 2);
Add(plist, 77);
Add(plist, 6);
Add(plist, 43);
Add(plist, 76);
Add(plist, 89);
```

```
노드 생성 전 초기화합니다. (Create 구현)

입력 받은 데이터의 전체출력!
22  2  77  6  43  76  89
```

2.Search(탐색)

위와 같이 연결리스트에 데이터 값들이 많이 저장되면 특정 데이터가 어디에 위치하는지 인간인 우리가 받아들이기 쉽게 인덱스로 반환되어 확인하는 것이 가장 편합니다.

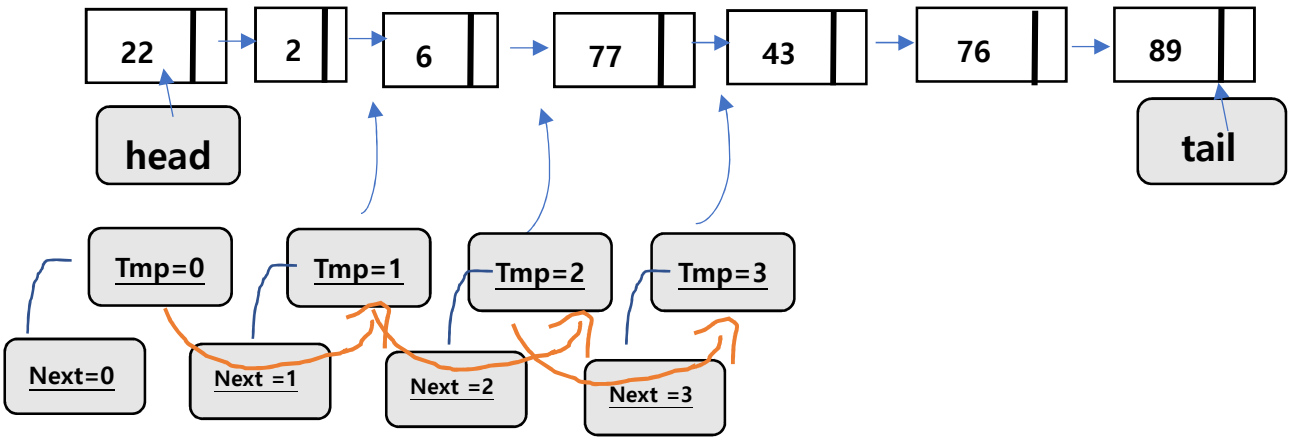


교재에서 선언한 cur이란 포인터로 저 tmp처럼 사용해도 되지만 저는 tmp란 포인터변수를 선언하여 구현했습니다.

```
void Search(Node* Head, int data)
{
    Node* temp = Head->next;
    int n = 1;
    while (temp != NULL)
    {
        Node* next = temp->next;
        if (data == (int)temp->data)
        {
            printf("%d번째 시도만에 인덱스 %d에 있다는 것을 확인. ₩n",n,n-1);
        }
        n++;
        temp =next;
    }
}
```

visualgo처럼 tmp란 노드포인터변수가 머리의 그 다음 값을 가리키도록 했습니다. Idx를 n으로 설정했습니다. 무한 루프인 while을 사용하여 tmp가 비어 있지 않을 때 계속 반복하도록 하여 위치를 찾도록 구현했습니다.

즉 tmp가 NULL이면 찾을 값은 인덱스 0입니다, 그 이 외의 경우인 tmp가 NULL아니면 next 노드를 이용해 하나씩 참조하여 특정값을 찾는 방식입니다.

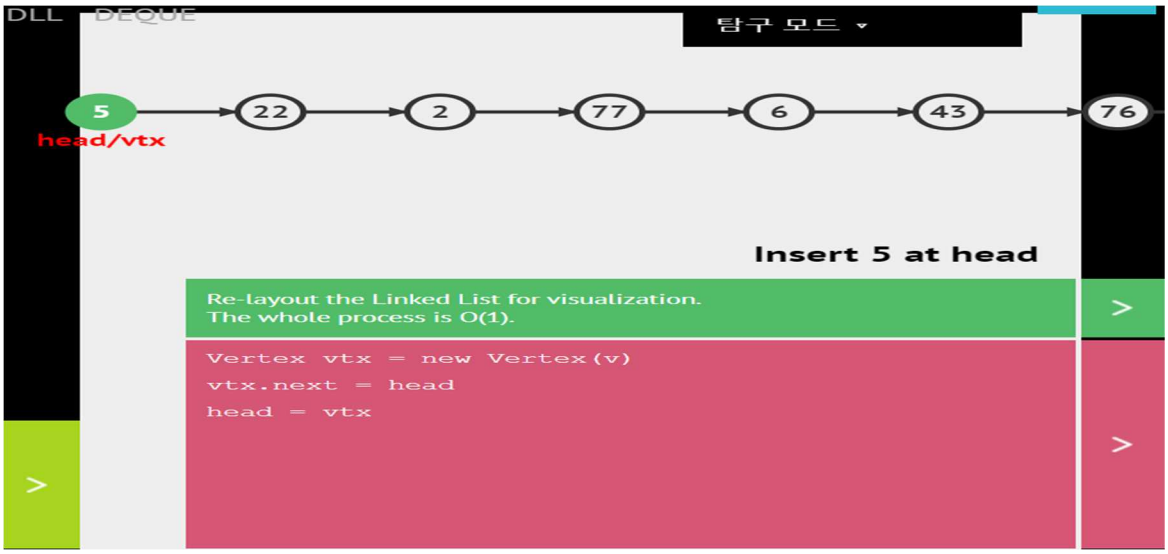


(주황색 이동을 나타냅니다.) 조건 달성 바로 printf!!

```
/**** 데이터 6을 탐색(Search) ****/
printf("노드 6을 찾습니다. (Search 구현)₩n");
Search(plist,6);
printf("₩n");
```

```
노드 6을 찾습니다. (Search 구현)
4번째 시도만에 인덱스 3에 있다는 것을 확인.
```

3.Insert(삽입)



이제 이미 있는 연결리스트에 특정 노드위치에 삽입하는 것을 구현할 것입니다. 특정위치는 크게 세가지로 구분할 수 있습니다. 첫번째 연결리스트의 머리에 삽입하는 경우, 두번째 연결리스트의

꼬리에 삽입하는 경우, 마지막 세번째 특정위치에 데이터를 삽입하는 경우가 있습니다. 두번째 같이 연결리스트 꼬리에 삽입하는 경우는 앞에서 잠깐 했던 add함수를 이용해도 되지만 저는 Insert함수로 저 3가지를 할 수 있도록 구현해 보았습니다.

아래의 코드를 보면

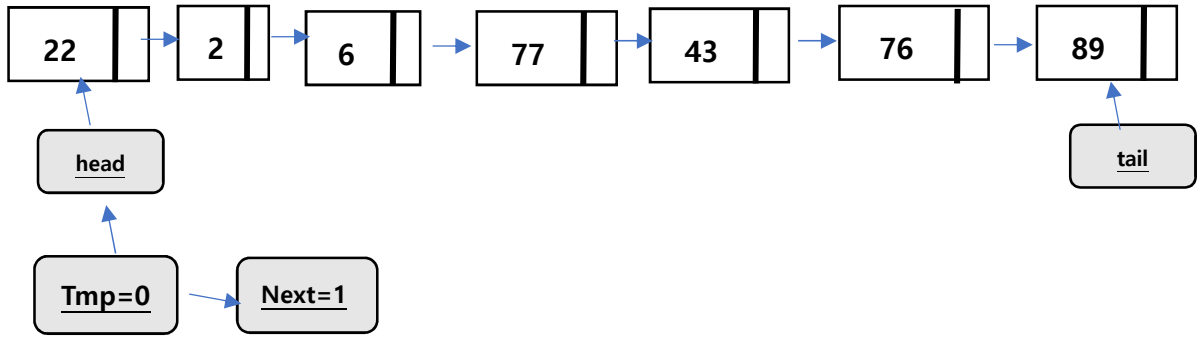
```
/**** 삽입함수 => Insert ****/
void Insert(List* plist,int pdata, int pos)
{
    //추가할 노드 만들기
    Node* newNode = NULL; // NULL 포인터 초기화, 추가되는 노트의 포인터변수
    newNode = (Node*)malloc(sizeof(Node));
    newNode->data = pdata;
    newNode->next = NULL;

    if (pos > 1 && pos <= plist->numOfData + 1)
    {
        Node* tmp = plist->head;
        for (int j = 1; j < pos - 1; j++)
        {
            tmp = tmp->next;
        }
        newNode->next = tmp->next;
        tmp->next =newNode;
        plist->tail = newNode;
    }
    else if(pos == 1)
    {
        newNode->next = plist->head;
        plist->head = newNode;
    }
```

먼저 새로 들어오는 노드 'newNode'를 선언합니다. 이 부분은 앞에서 했으니 자세한 설명은 생략하겠습니다.

```
newNode->data = pdata;
newNode->next = NULL;
```

이번에 next노드를 이용합니다. 이번에도 NULL로 초기화시켜놓습니다. 이제 저 `void Insert(List* plist,int pdata, int pos)`에서 주목을 하실 매개변수는 `int pos` 입니다. Int pos는 제가 삽입하고 싶은 위치 즉 배열에서의 인덱스와 같은 것을 담는 변수입니다.

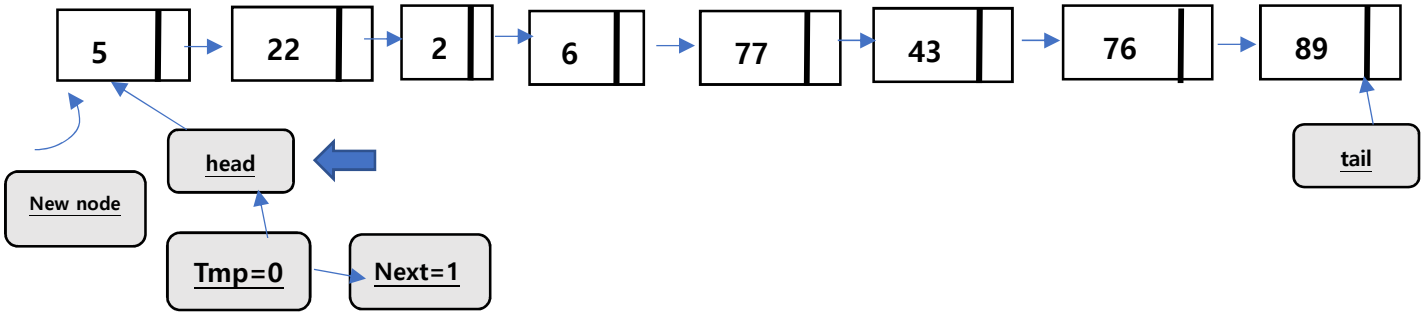


1)머리에 삽입하기

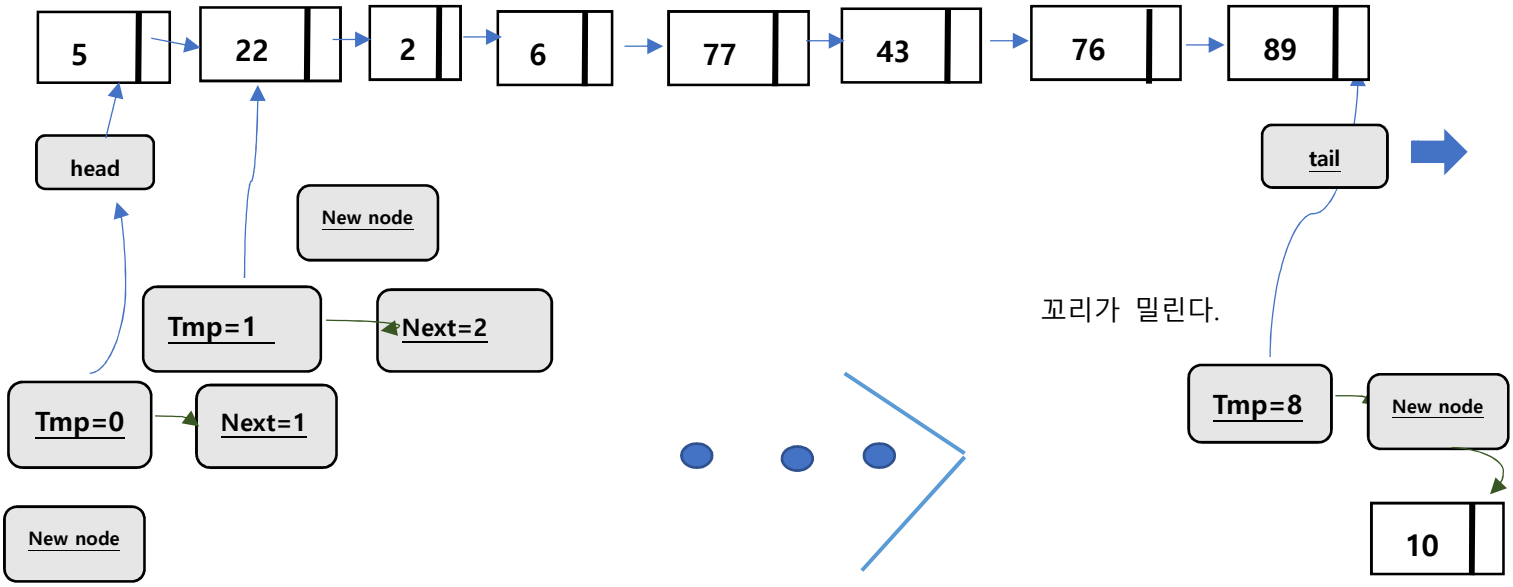
위 그림처럼 기존 연결리스트가 있습니다. 여기서 머리 위치에 삽입을 하면 저는 pos에 1쓸 것입니다.

그럼 if조건문의 else if조건에 딱 걸립니다. 만약 pos가 1이면 NewNode가 머리가 되는 것입니다.

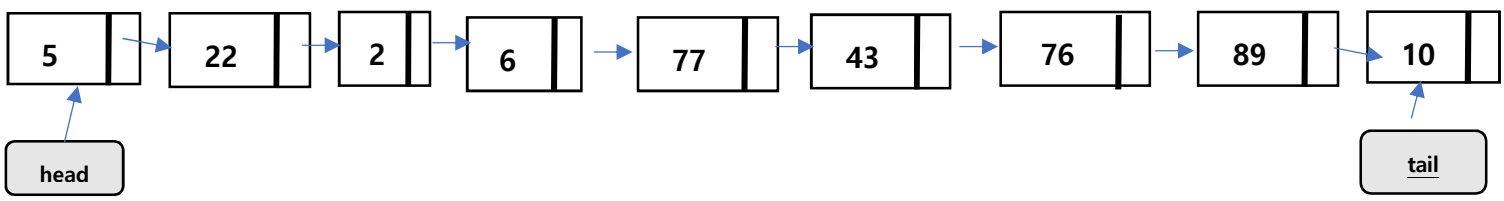
것입니다. 즉 22는 밀리게 되는 것입니다.



2)꼬리에 삽입하기



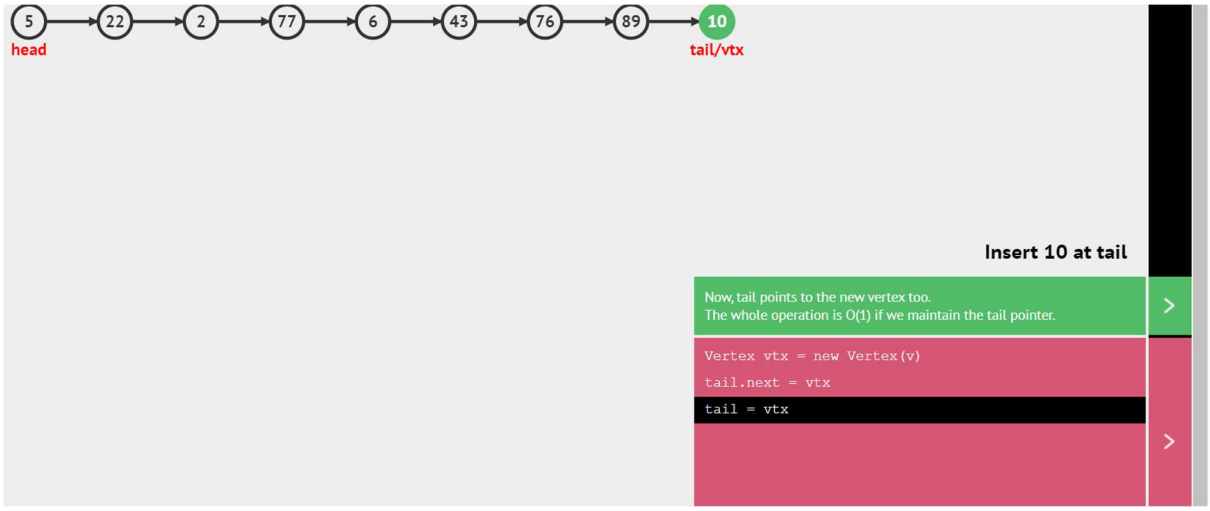
노트 포인터변수 tmp가 입력받은 들어갈 위치를 찾습니다. Tmp는 0부터 꼬리의 위치치까지 증가되는데 0 1 2 3 4 이런 식으로 증가되면서 꼬리의 위치로 다가가서 꼬리 위치에 삽입되고 그 전에 있던 꼬리는 뒤로 밀리게 됩니다.



```
/**** 꼬리에 10을 삽입(Insert) ****/
printf("꼬리에 노드 10을 삽입합니다. (Insert 구현)₩n");
Insert(plist, 10, 9);
```

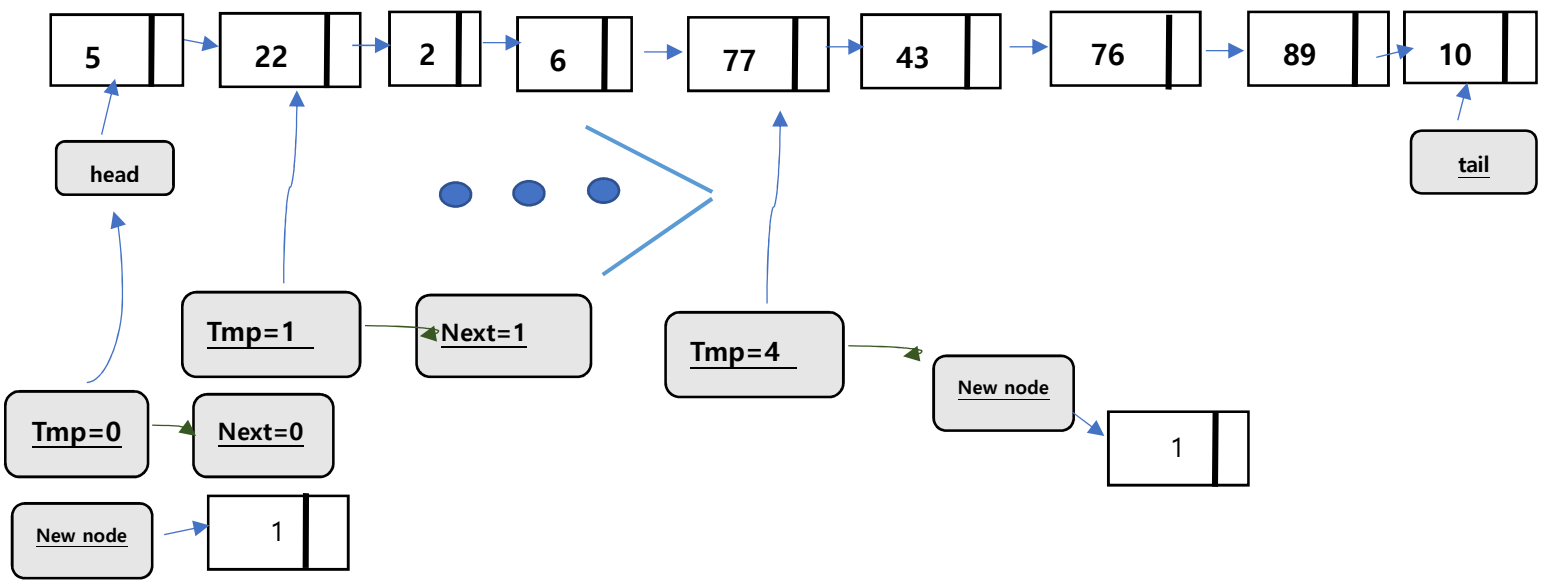
여기서 tmp보다 +1시킨 이유는 위 코드에서 보시면 아시다싶이 insert함수에서는 pos-1로 했기 때문입니다.

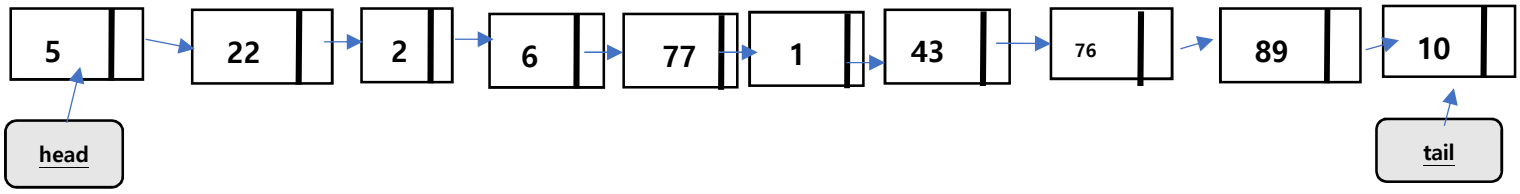
```
꼬리에 노드 10을 삽입합니다. (Insert 구현)
현재 연결리스트 출력!
5  22  2  77  6  43  76  89  10
```

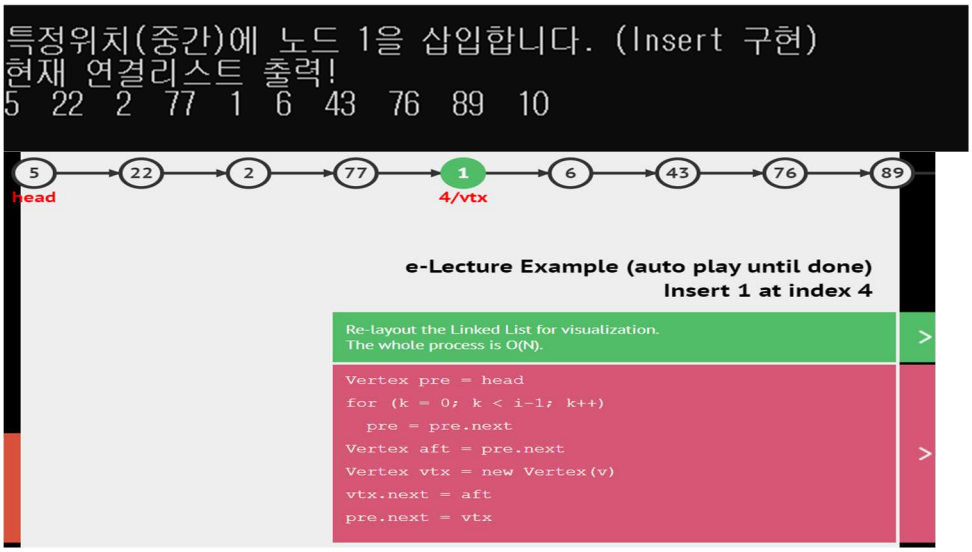


3)특정위치에 삽입하기

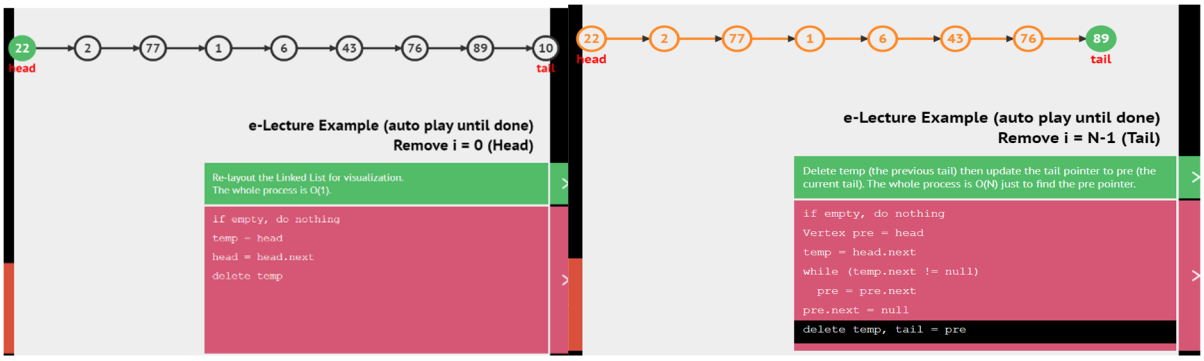
"노트 포인터변수 tmp가 입력받은 들어갈 위치를 찾습니다. Tmp는 0부터 꼬리의 위치치까지 증가되는데 0 1 2 3 4 이런 식으로 증가되면서 꼬리의 위치로 다가가서 꼬리 위치에 삽입되고 그 전에 있던 꼬리는 뒤로 밀리게 됩니다."라고 앞에서 설명했던 것에서 꼬리가 아니라 사용자가 임의로 입력한 위치입니다. 하지만 동일한 논리로 구현됩니다.







4.Remove(삭제)



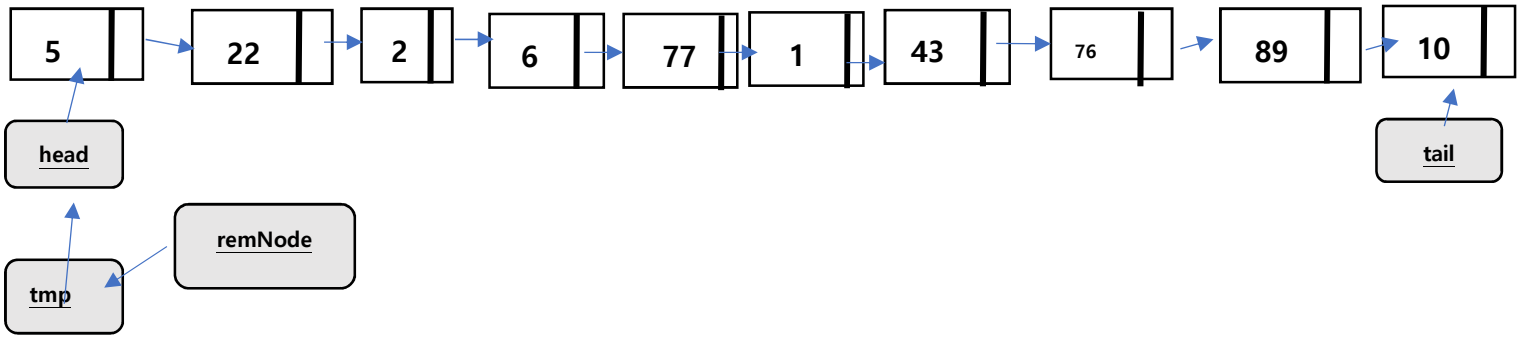
이제 머리, 꼬리, 특정 위치에 있는 노드를 삭제합니다. 이것의 논리도 앞에서 했던 insert와 매우 비슷합니다. 먼저 코드를 보시면

```
/**** 제거함수 => Remove ****/
void Remove(List* plist, int pos)
{
    if (pos < 1 && pos <= plist->numOfData + 1)
    {
        printf("해당 위치에 삭제할 값이 없습니다.₩n");
        return;
    }
    else
    {
        Node* tmp = plist->head;
        Node* remNode = plist->head;

        if (pos == 1)
        {
            plist->head= plist->head->next;
        }
```

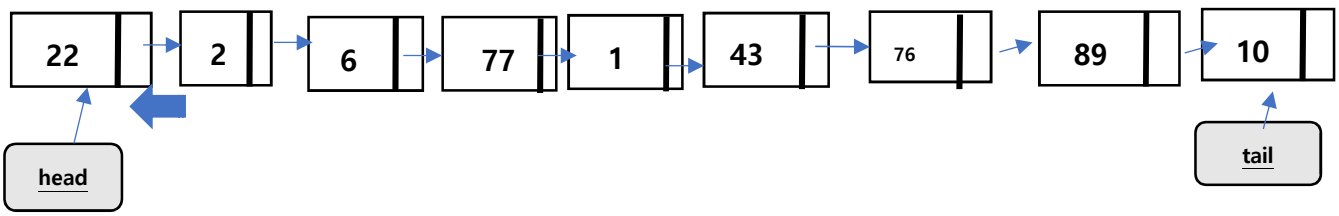
```
        else
        {
            for (int k = 1; k < pos - 1; k++)
            {
                tmp = tmp->next;
            }
            remNode = tmp->next;
            tmp->next = tmp->next->next;
        }
        free(remNode);
        (plist->numOfData)--;
    }
}
```

먼저 첫 번째 조건 연결리스트의 범위를 넘어 갔을 경우에는 저런 문구가 출력되도록 했습니다. 그리고 노드 포인터 변수 tmp는 머리를 향하게 설정하고 제거할 노트 포인터 변수를 선언하고 그것을 머리를 향하게 합니다.



1) 머리 노드 삭제

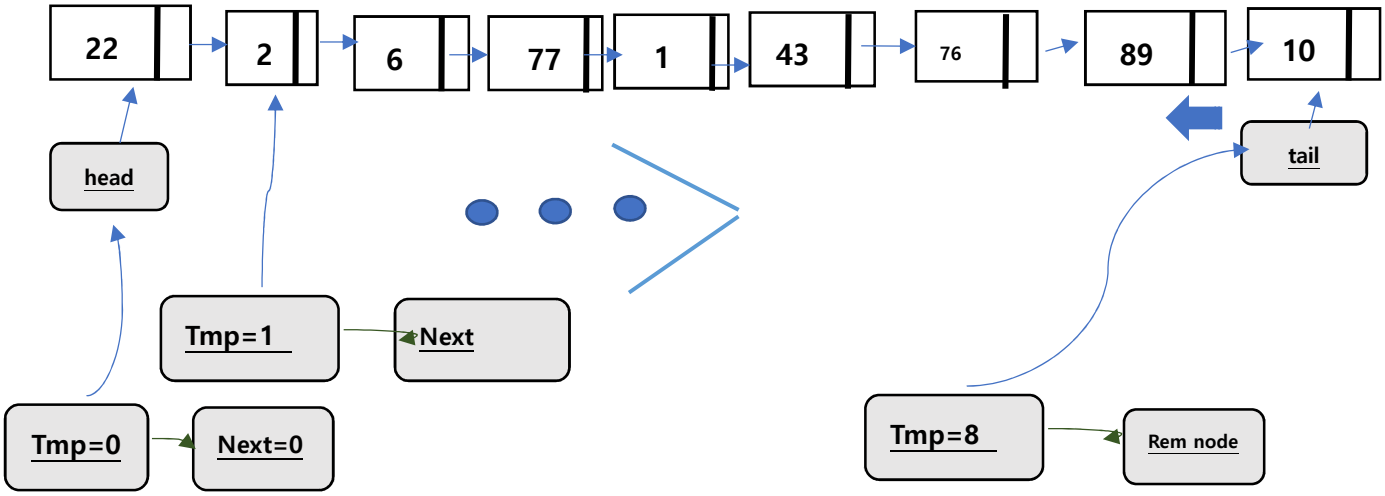
원하는 삭제위치가 머리라면 tmp가 돌 필요도 없이 바로 삭제됩니다.



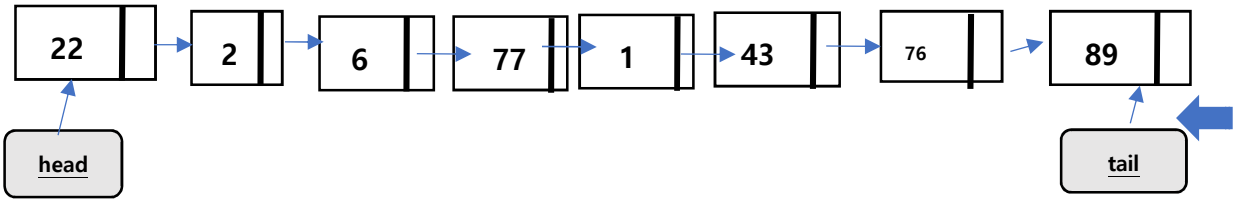

```
/**** 머리 노드 삭제(Remove) ****/
printf("머리 노드를 삭제합니다. (Remove 구현)₩n");
Remove(plist, 1);
```

```
머리 노드를 삭제합니다. (Remove 구현)
현재 연결리스트 출력!
22  2  77  1  6  43  76  89  10
```

2) 꼬리 노드 삭제



이런 식으로 꼬리 노드를 삭제하면

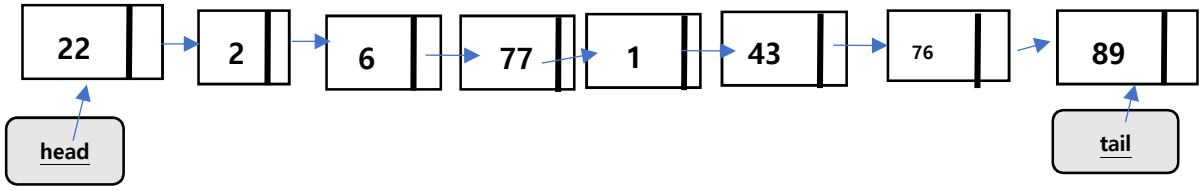


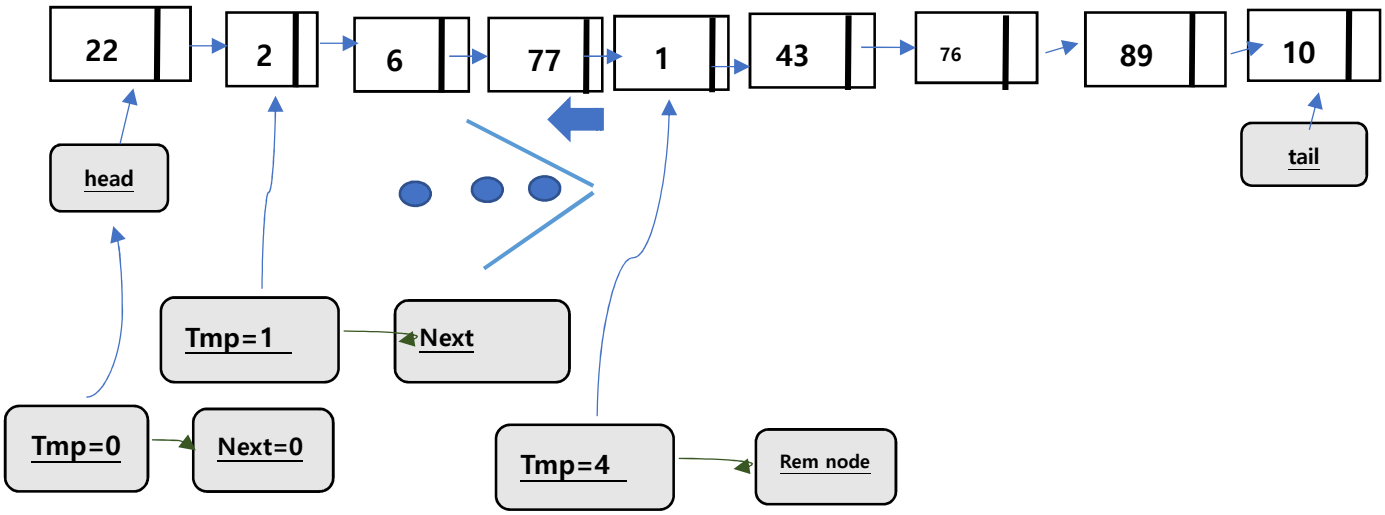

```
/**** 꼬리 노드 삭제(Remove) ****/
printf("꼬리 노드를 삭제합니다. (Remove 구현)₩n");
Remove(plist, 9);
```

```
꼬리 노드를 삭제합니다. (Remove 구현)
현재 연결리스트 출력!
22  2  77  1  6  43  76  89
```

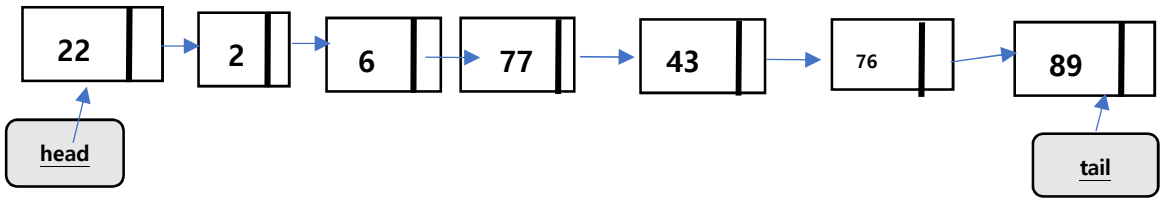
3) 특정 노드 삭제

이번에도 삭제할 노드의 위치가 머리, 꼬리가 아니라 임의로 지정한 위치이면 이번에도 동일하게 하면 됩니다.





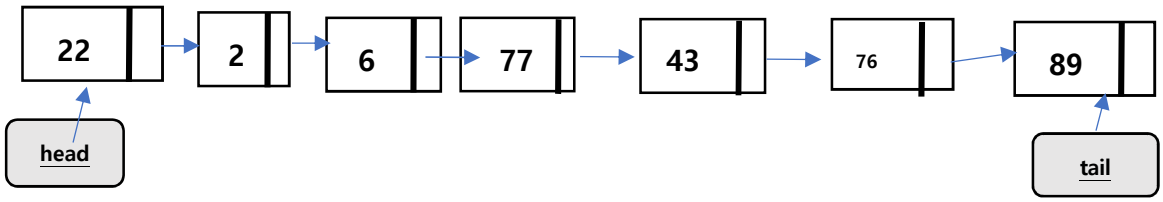
삭제하고 싶은 특정 노트 찾았음! 데이터수-1




```
/**** 특정 노드 삭제(Remove) ****/
printf("특정 노드(아까 1노드를 추가했던 )를 삭제합니다. (Remove 구현)₩n");
Remove(plist, 4);
```

```
특정 노드(아까 1노드를 추가했던 )를 삭제합니다. (Remove 구현)
현재 연결리스트 출력!
22  2  77  6  43  76  89
```

4) 모든 노드 삭제하기



모든 노드를 삭제 할꺼면 이제는 하나씩 꼬리부터 머리까지 모드 다 지우면 됩니다.

```
모든 노드를 삭제합니다. (Remove 구현)
22을(를) 삭제합니다.
2을(를) 삭제합니다.
77을(를) 삭제합니다.
6을(를) 삭제합니다.
43을(를) 삭제합니다.
76을(를) 삭제합니다.
89을(를) 삭제합니다.
```