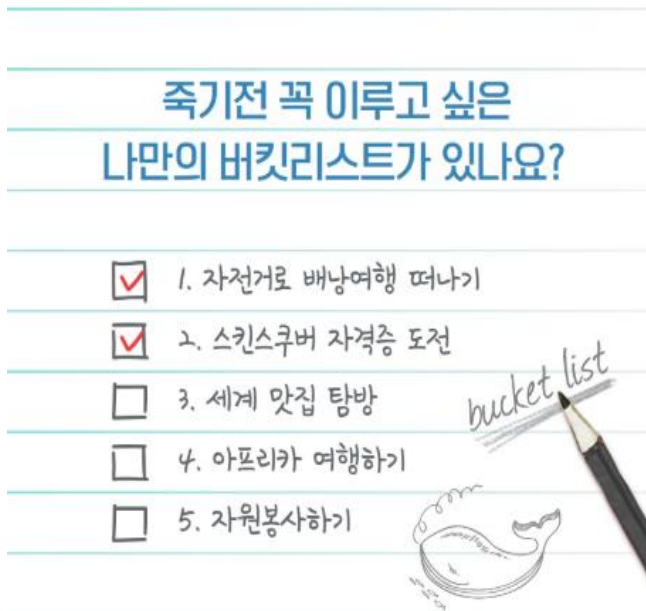

연결 리스트

2020.01.16

리스트

- 리스트 : 자료를 정리하는 방법중의 하나. -> 항목들이 차례대로 저장되어 있다.
(항목들은 순서 or 위치를 가진다. <-> 집합 : 항목 간에 순서의 개념이 없다.)
표시 : $L = (\text{item0}, \text{item1}, \text{item2})$, 할 수 있는 연산 : 삽입, 삭제, 탐색

Ex) 스택과 큐도 리스트의 일종이다.



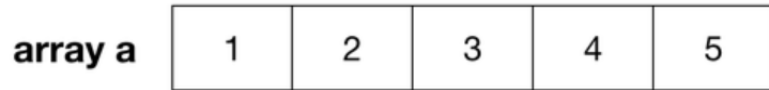
<스택>



<큐>

리스트 구현

■ 배열을 이용

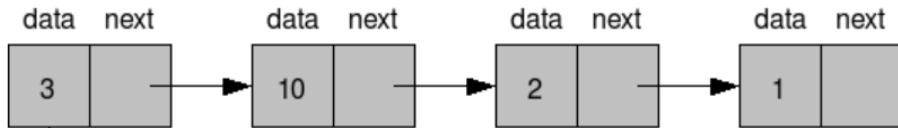


특징: 연속된 공간에 생성된다. -> 구현이 간단하고 검색속도가 빠르다.

단점: 1. 크기가 고정되어 있다. -> 공간이 부족하면 더 큰 배열을 만들어 복사해야 한다.

2. 중간에 삽입, 삭제를 위해서는 기존데이터들을 이동해야 한다.

■ 포인터를 이용 -> Linked List(연결 리스트)



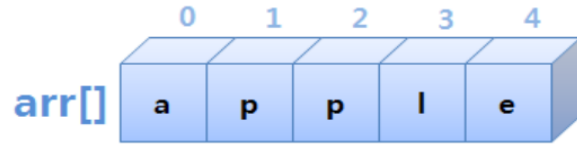
특징: 크기 제한이 없다. -> 중간에서 쉽게 삽입하거나 삭제할 수 있다.

단점: 1. 구현이 복잡하다.

2. 임의의 항목을 추출할 때 배열을 사용하는 것보다 시간이 오래 걸린다.

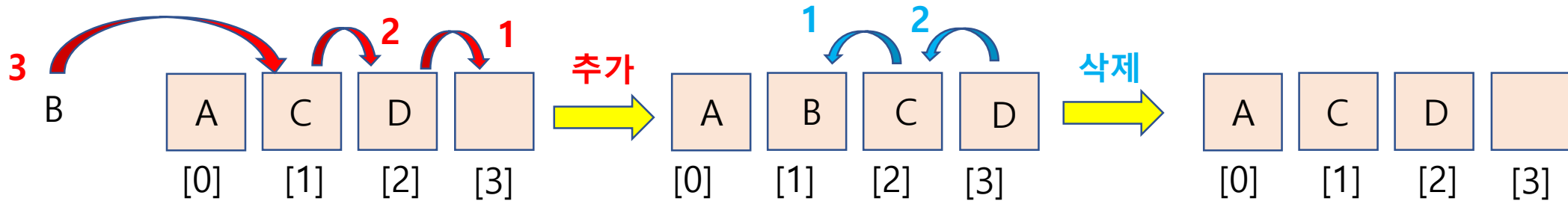
배열로 구현된 리스트

-> 순차적인 메모리 공간이 할당 (Sequential representation)



```
#define MAXLISTSIZE 100;
typedef int element;
typedef struct{
    element array[MAXLISTSIZE];
    int size;
}ArrayListType;
```

<기초 연산> – 오류 처리, 리스트 초기화, 비어있는 확인, 리스트 출력, 항목 추가 등
▶ 모든 연산들은 구조체 포인터를 받아 처리한다 (∵구조체를 변경할 필요도 있기 때문)



<항목 추가/삭제 연산>

배열로 구현된 리스트

- 항목 추가 함수

```
void insert(ArrayListType *L, int pos, element item){
    if (!is_full(L) && (pos >= 0) && (pos <= L->size)) {
        for (int i = (L->size - 1); i >= pos; i--) L->array[i + 1] = L->array[i];
        L->array[pos] = item;
        L->size++;
    }
}
```

<중간에 삽입>

```
void insert_last(ArrayListType *L, element item) {
    if( L->size >= MAX_LIST_SIZE ) {
        error("리스트 오버플로우");
    }
    L->array[L->size++] = item;
}
```

<마지막에 삽입>

- 항목 삭제 함수

```
element delete(ArrayListType *L, int pos){
    element item;
    if (pos < 0 || pos >= L->size) error("위치 오류");
    item = L->array[pos];
    for (int i = pos; i < (L->size - 1); i++) L->array[i] = L->array[i + 1];
    L->size--;
    return item;
}
```

<삭제>

연결 리스트

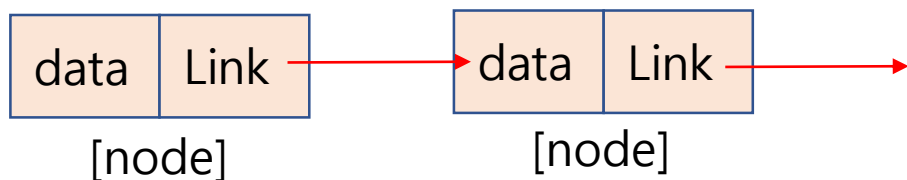
-> 포인터를 사용하여 데이터들을 연결한다.

※ 연결된 표현은 리스트의 구현에서만 사용되는 것이 아니고 다른 여러가지 자료구조등을 구현하는데도 많이 사용된다.

연결 리스트 : 물리적으로 흩어져 있는 자료들을 서로 연결하여 하나로 묶는 방법.

-> 첫 번째 데이터만 알 수 있으면 연결리스트의 나머지 데이터들은 줄만 따라가면 된다.

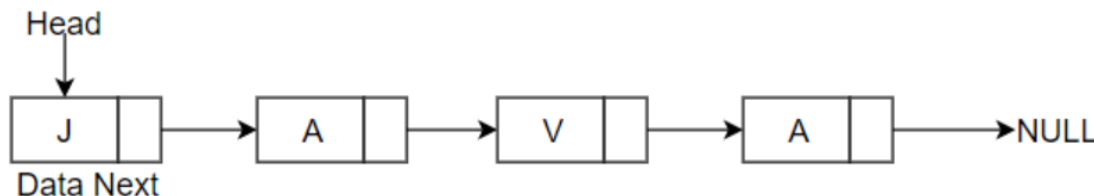
단점: 구현이 어렵다. 메모리 공간을 많이 차지한다. 데이터에 순차 접근 하는데 시간이 오래 걸린다.



※ 연결리스트는 node들의 집합
※ node = data field + Link field

(다른 노드를 가리키는 포인터가 저장)

헤드 포인터 : 첫 번째 노드를 가리키는 변수 (∵ 연결리스트의 첫번째 노드를 알아야 전체 노드에 접근이 가능하다.)

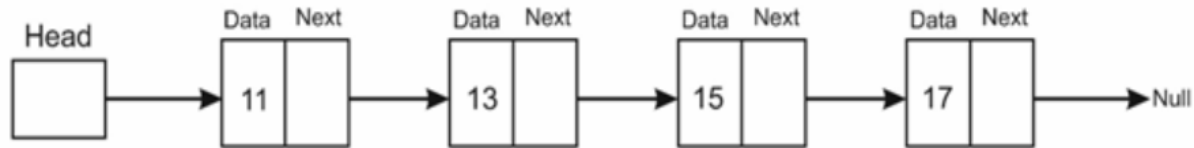


※ 마지막 node의 link field = NULL 로 설정
-> 더 이상 연결된 노드가 없음을 의미.

※ 연결리스트의 node들은 필요할 때마다 **malloc ()** 을 이용하여 동적으로 생성.

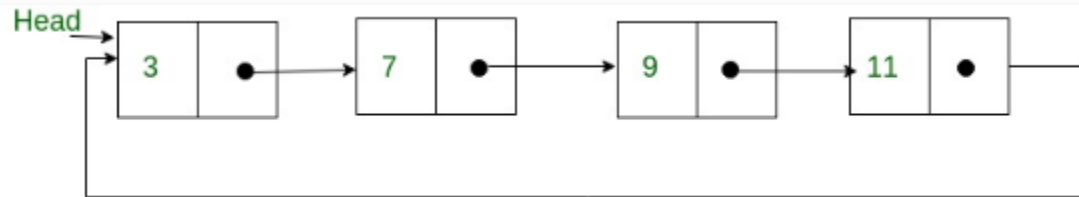
연결 리스트 종류

1. Singly linked list



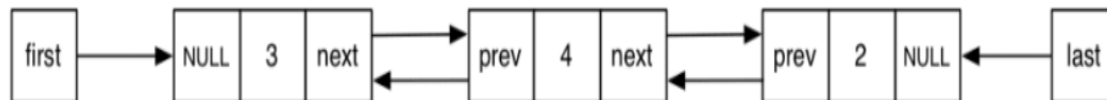
※ 하나의 방향으로만 연결

2. Circular linked list



※ 마지막 노드의 링크가 첫 번째 노드를 가리킨다.

3. Doubly linked list



※ 각 노드 마다 앞, 뒤 노드를 가리키는 2개의 링크가 존재

단순 연결 리스트

C언어 구현 -> 1. 노드는 어떻게 정의할 것인가? -> 자기 참조 구조체를 이용한다.
2. 노드는 어떻게 생성할 것인가? -> malloc () 을 호출하여 동적 메모리로 생성한다.
3. 노드는 어떻게 삭제할 것인가? -> free () 을 호출하여 동적 메모리를 해체한다.

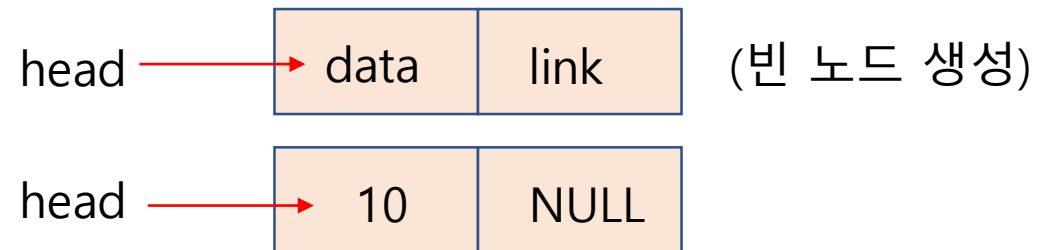
자기 참조 구조체 : 자기 자신을 참조하는 포함하는 구조체 (※질문 : 왜 자기 참조 구조체를 사용해야 하는가?)

<node 정의>

```
typedef int element;  
typedef struct ListNode { // 노드 타입을 구조체로 정의한다.  
    element data;  
    struct ListNode *link;  
} ListNode;
```

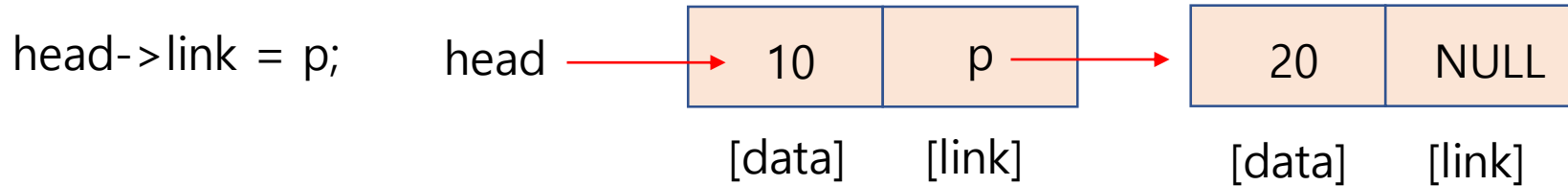
<List 생성>

```
ListNode *head = NULL;  
head = (ListNode *)malloc(sizeof(ListNode));  
head->data = 10;  
head->link = NULL;
```



단순 연결 리스트

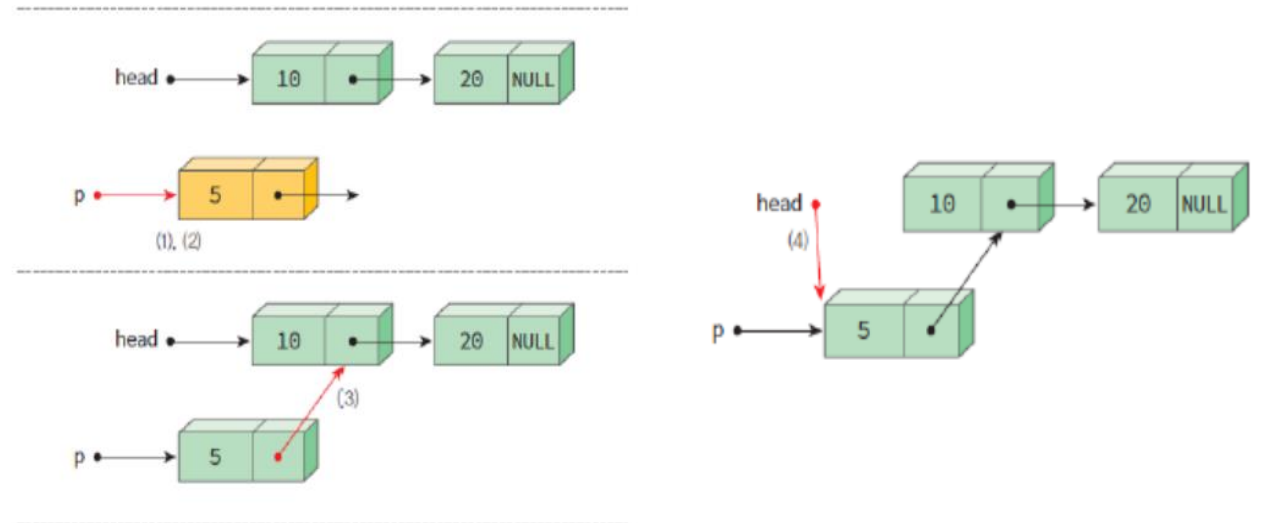
<노드의 연결>



<단순 연결 리스트의 연산 구현>

1. 삽입 연산 (처음 삽입)

```
ListNode* insert_first(ListNode *head, int value) {  
    ListNode *p =(ListNode *)malloc(sizeof(ListNode));  
    p->data = value;  
    p->link = head;  
    head = p;  
    return head;  
}
```



3. 리스트 방문 알고리즘 -> 출력!

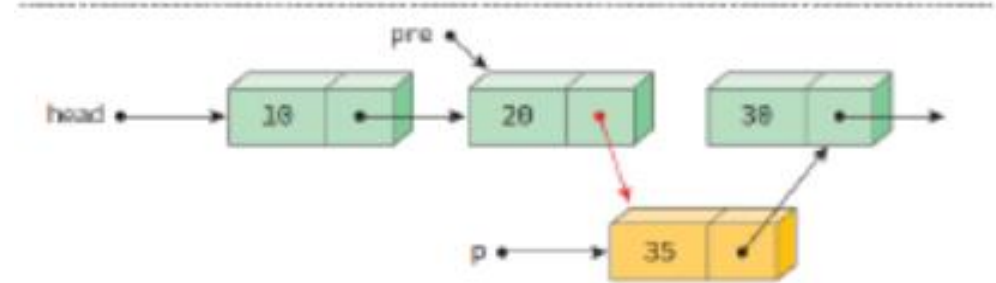
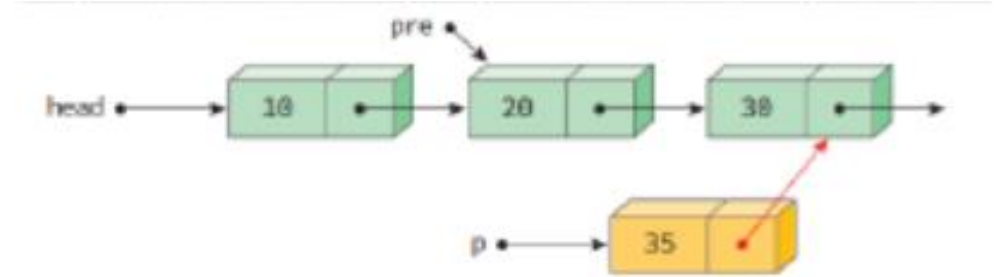
2. 삭제 연산 (처음 삭제, 중간 삭제)

단순 연결 리스트

<단순 연결 리스트의 연산 구현>

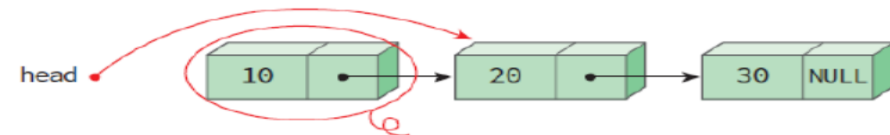
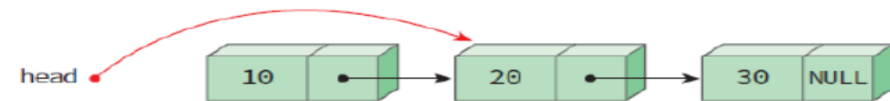
1. 삽입 연산 (중간 삽입)

```
ListNode* insert(ListNode *head, ListNode *pre, element value) {  
    ListNode *p = (ListNode *)malloc(sizeof(ListNode));  
    p->data = value;  
    p->link = pre->link;  
    pre->link = p;  
    return head;  
}
```



2. 삭제 연산 (처음 삭제)

```
ListNode* delete_first(ListNode *head){  
    ListNode *removed;  
    if (head == NULL) return NULL;  
    removed = head;  
    head = removed->link;  
    free(removed);  
    return head;  
}
```

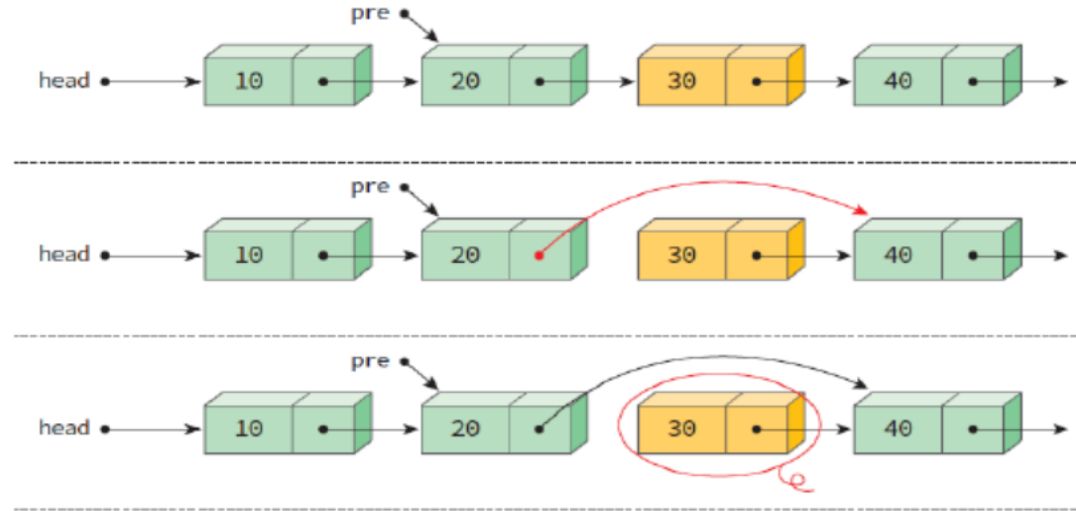


단순 연결 리스트

<단순 연결 리스트의 연산 구현>

2. 삭제 연산 (중간 삭제)

```
ListNode* delete(ListNode *head, ListNode *pre){  
    ListNode *removed;  
    removed = pre->link;  
    pre->link = removed->link;  
    free(removed);  
    return head;  
}
```



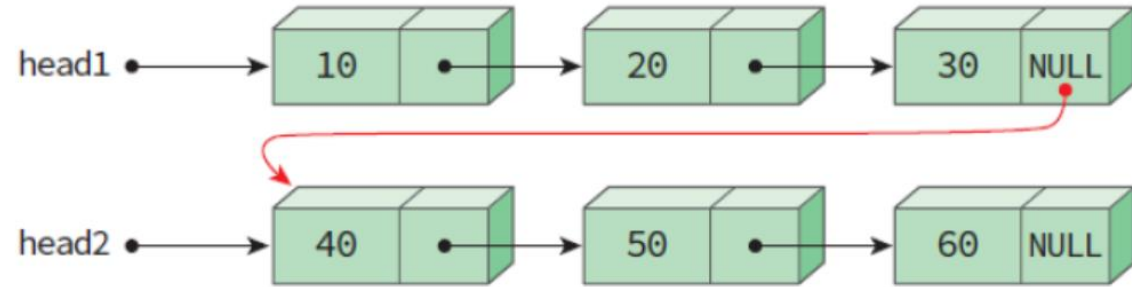
3. 리스트 방문

```
void print_list(ListNode *head){  
    for (ListNode *p = head; p != NULL; p = p->link) printf("%d->", p->data);  
    printf("NULL \n");  
}
```

연결 리스트

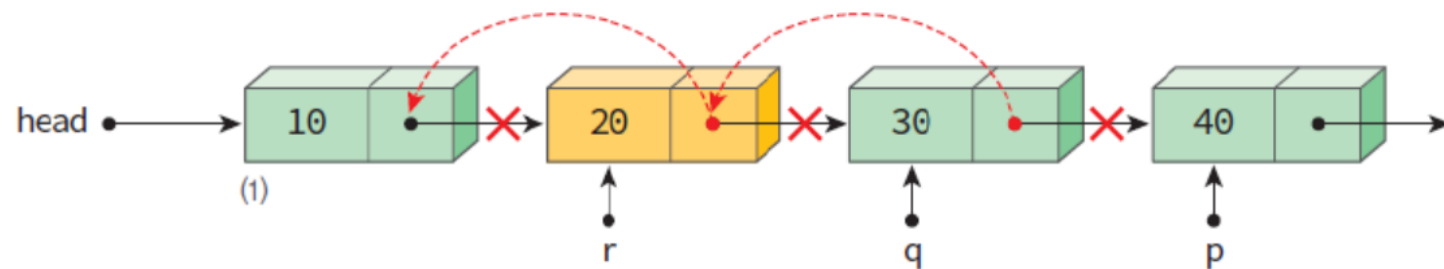
<두개 리스트를 하나로 합치는 함수>

```
ListNode* concat_list(ListNode *head1, ListNode *head2){
    if (head1 == NULL) return head2;
    else if (head2 == NULL) return head1;
    else {
        ListNode *p;
        p = head1;
        while (p->link != NULL)
            p = p->link;
        p->link = head2;
        return head1;
    }
}
```



<리스트를 역순으로 만드는 함수>

```
ListNode* reverse(ListNode *head){
    // 순회 포인터로 p, q, r을 사용
    ListNode *p, *q, *r;
    p = head; // p는 역순으로 만들 리스트
    q = NULL; // q는 역순으로 만들 노드
    while (p != NULL) {
        r = q; // r은 역순으로 된 리스트.
        // r은 q, q는 p를 차례로 따라간다.
        q = p;
        p = p->link;
        q->link = r; // q의 링크 방향을 바꾼다.
    }
    return q;
}
```

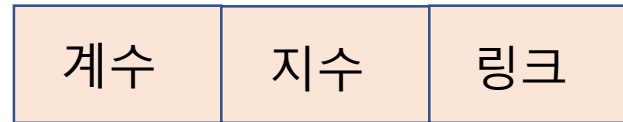


연결 리스트

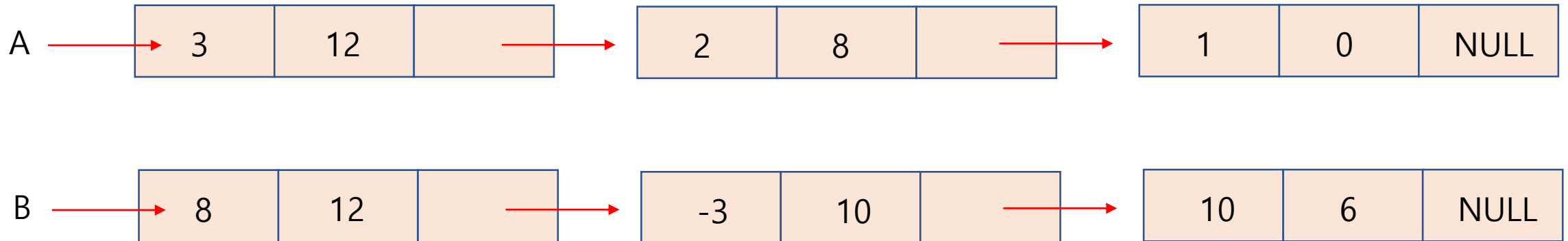
<연결리스트의 응용 : 다항식 >

(구조체 정의)

```
typedef struct ListNode {  
    int coef; //계수  
    int expon; //지수  
    struct ListNode *link;  
} ListNode;
```



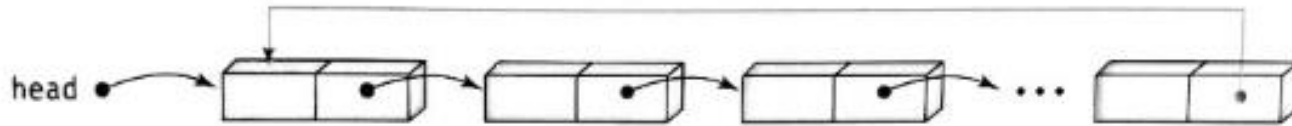
$$A=3x^{12}+2x^8+1, B=8x^{12}-3x^{10}+10x^6$$



원형 연결 리스트

원형 연결 리스트란

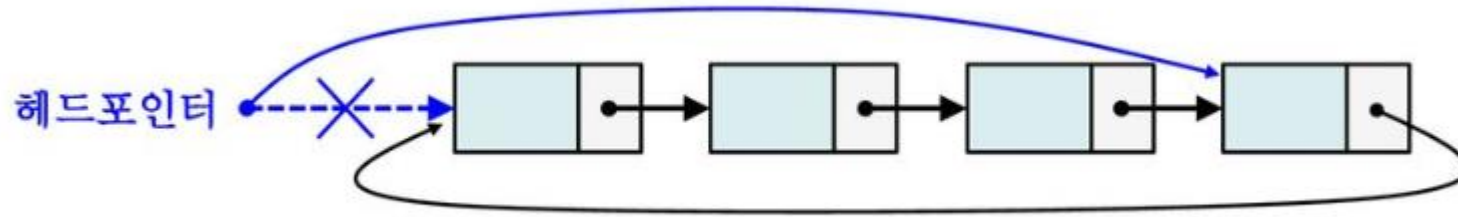
리스트의 마지막 노드의 링크가 첫 번째 노드를 가리키는 연결 리스트이다.



<장점> : 한 노드에서 다른 모든 노드로의 접근이 가능하다. (삽입, 삭제가 단순)

<단점> : 노드의 삽입, 삭제시 선행 노드의 포인터가 필요하다.

변형된 원형 연결 리스트



리스트의 끝에 노드를 추가하는 경우

<단순 연결리스트> : 첫 번째 노드에서부터 링크를 따라서 마지막 노드로 이동
→ 시간이 오래 걸린다.

<변형된 원형 연결리스트> : 리스트의 처음과 끝에 노드 삽입이 쉽다.

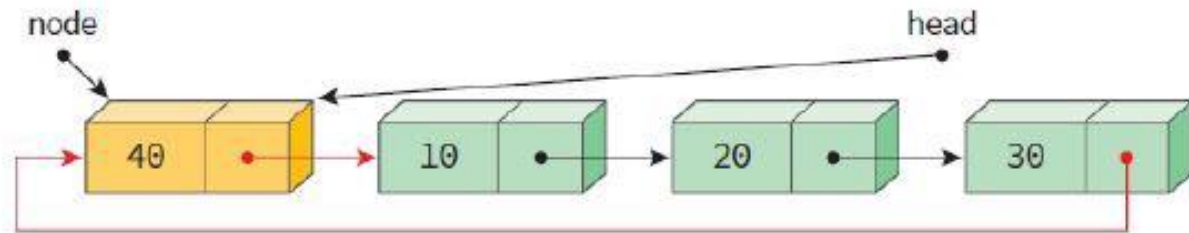
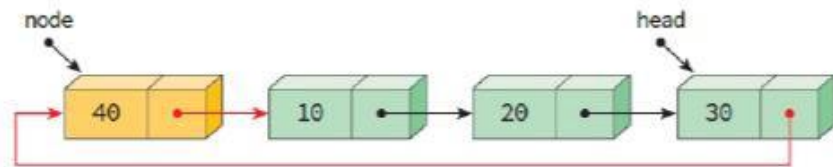
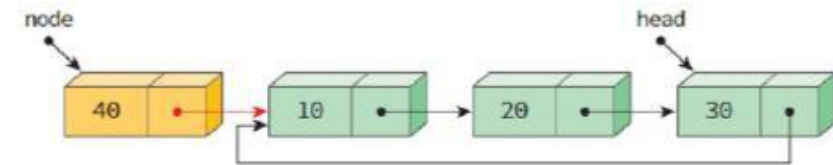
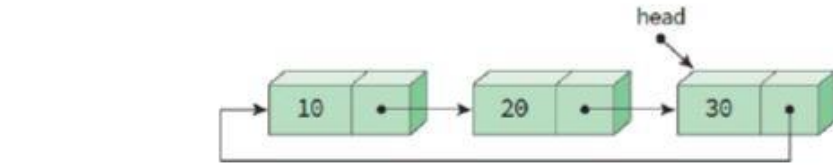
원형 연결 리스트

<원형 연결 리스트의 연산 구현>

1. 삽입 연산 (처음 삽입, 마지막 삽입)

```
ListNode* insert_first(ListNode* head, element data)
{
    ListNode *node=(ListNode *)malloc(sizeof(ListNode));
    node->data=data;
    if(head==NULL){
        head = node;
        node->link = head;
    }
    else {
        node->link = head->link;
        head->link = node;
    }
    return head; // 변경된 헤드 포인터를 반환한다.
}
```

```
ListNode* insert_last(ListNode* head, element data)
{
    ListNode *node=(ListNode *)malloc(sizeof(ListNode));
    node->data=data;
    if(head==NULL){
        head = node;
        node->link = head;
    }
    else {
        node->link = head->link;
        head->link = node;
        head = node;
    }
    return head; // 변경된 헤드 포인터를 반환한다.
}
```



원형 연결 리스트

리스트의 끝에 도달했는지 확인하는 방법

```
void print_list(ListNode* head)
{
    ListNode* p;

    if(head == NULL) return;
    p = head->link;
    do {
        printf("%d->", p->data);
        p = p->link;
    } while(p != head);
    printf("%d->", p->data);
}
```

```
void display(ListNode *head)
{
    ListNode *p;
    if( head == NULL ) return;

    p = head;
    do {
        printf("%d->", p->data);
        p = p->link;
    } while(p != head);
}
```

<단순 연결리스트> : 마지막 노드의 링크가 NULL 인지 아닌지 비교

<원형 연결리스트> : 헤드 포인터와 비교

원형 연결 리스트

<원형 연결 리스트의 활용>

1. 컴퓨터에서 여러 응용 프로그램을 하나의 CPU를 이용하여 실행할 때

- 모든 응용 프로그램을 원형 연결 리스트에 보관하고, 운영 체제는 모든 응용 프로그램이 완료될 때까지 원형 연결 리스트를 계속 순회한다.

2. 멀티 플레이어 게임

- 모든 플레이어를 원형 연결 리스트에 저장하고, 포인터를 통해 이동한다.

3. 원형 큐를 만드는데 사용

- 두개의 포인터 front, rear가 있어야 한다.

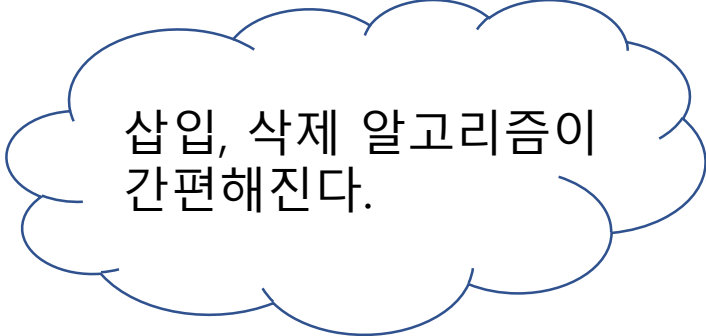
이중 연결 리스트

<왜 이중 연결 리스트가 나왔을까>

- 단순 연결 리스트에서 후속 노드를 찾기는 쉽지만, 선행 노드를 찾기는 어렵다.
- 원형 연결 리스트에서도 거의 전체 노드를 거쳐서 돌아 와야 한다.

<이중 연결 리스트의 특징>

- 링크가 양방향이므로 양방향으로 검색이 가능하다.○
- 데이터를 가지고 있지 않는 헤드 노드가 존재하므로, 헤드 포인터는 필요 없다.
- 사용하기 전에 반드시 헤드 노드의 링크필드들이 자기 자신을 가리키도록 초기화를 해야 한다.
- 단점 : 공간을 많이 차지하고, 코드가 복잡해진다.



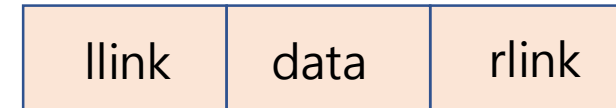
삽입, 삭제 알고리즘이
간편해진다.

이중 연결 리스트

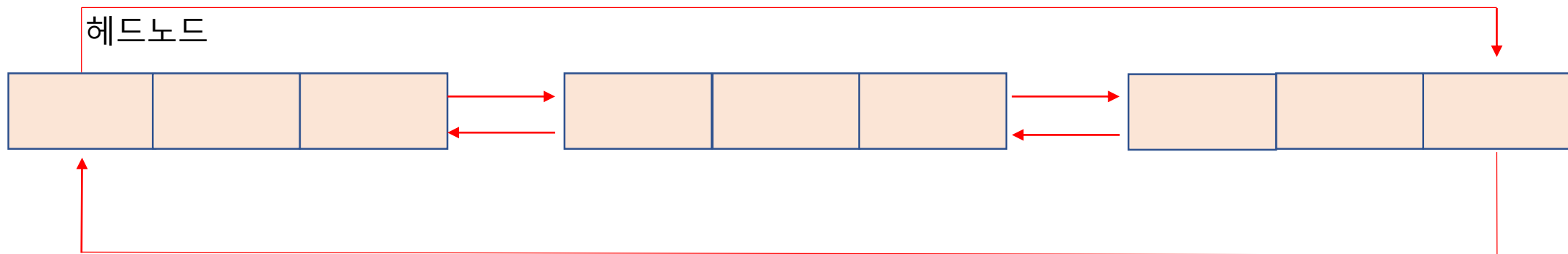
<node 정의>

```
typedef int element;
typedef struct DListNode { // 이중 연결 노드 타입
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;
```

<node 구조>



임의의 노드를 가리키는 포인터를 p이라 하면,
 $p = p->llink->rlink = p->rlink->llink$ 이 항상 성립한다.



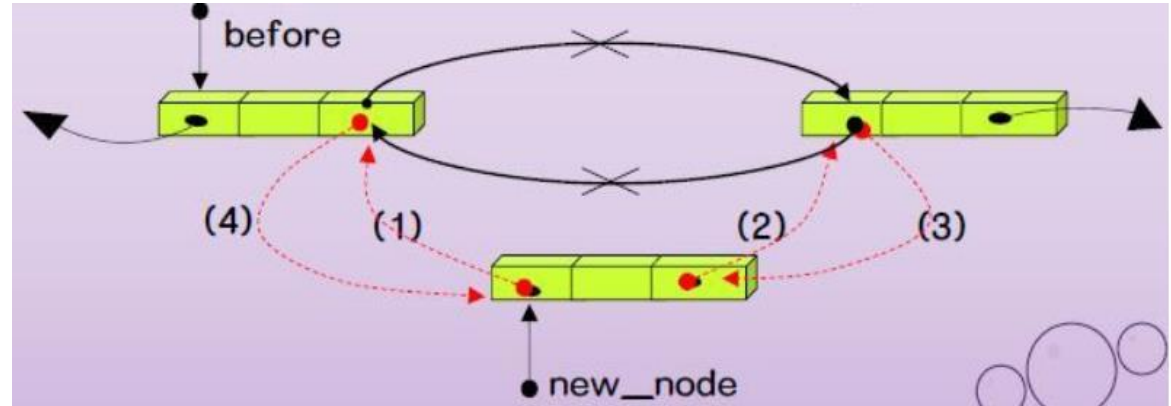
※ 실제 응용에서는 이중 연결 리스트와 원형 연결 리스트를 혼합한 형태가 많이 사용된다.

이중 연결 리스트

<이중 연결 리스트의 연산 구현>

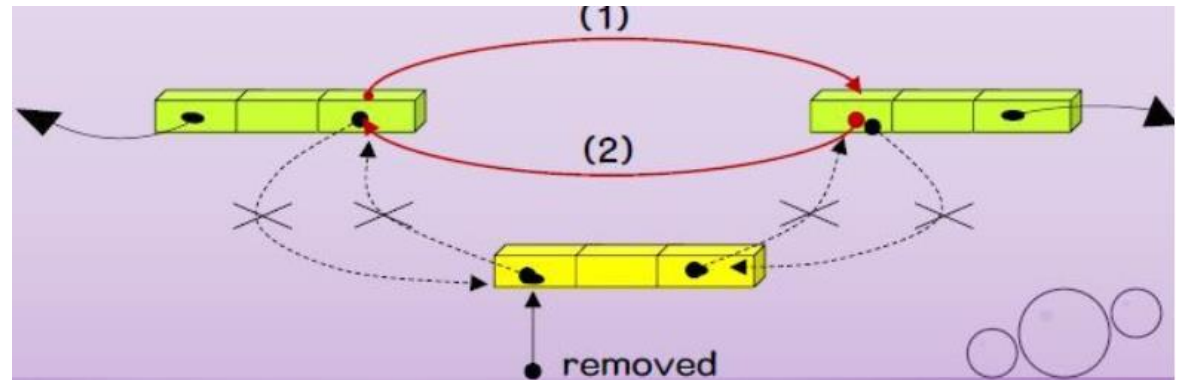
1. 삽입 연산

```
void dinsert(DListNode *before, element data)
{
    DListNode *newnode = (DListNode *)malloc(sizeof(DListNode));
    strcpy(newnode->data, data);
    newnode->llink = before;
    newnode->rlink = before->rlink;
    before->rlink->llink = newnode;
    before->rlink = newnode;
}
```



2. 삭제 연산

```
void ddelete(DListNode *head, DListNode *removed)
{
    if(removed == head) return;
    removed->llink->rlink = removed->rlink;
    removed->rlink->llink = removed->llink;
    free(removed);
}
```



이중 연결 리스트

<mp3 재생 프로그램 만들기>

- 왜 이중 연결 리스트를 사용해야 될까?

Mp3를 재생하다 보면 이전 곡으로 가기도 하고, 다음 곡으로 가기도 한다.
따라서 현재항목에서 이전 항목이나 다음 항목으로 쉽게 이동할 수 있는 자료 구조를 사용해야한다.



배열을 이용한 스택 vs 연결 리스트를 이용한 스택

<공통점>

제공되는 외부 인터페이스는 완전히 동일

<차이점>

연결 리스트를 사용할 경우, 동적 메모리 할당만 할 수 있다면 스택에 새로운 요소 삽입 가능하다.

하지만, 동적 메모리 할당이나 해제를 해야 하므로 삽입이나 삭제 시간이 조금 더 길다.

연결 리스트의 경우, 공백 상태는 top 포인터가 NULL인 경우 이고 포화 상태는 동적 메모리 할당만 된다면 노드를 생성할 수 있기 때문에 없는 것이나 마찬가지이다.

연결된 스택

<연결된 스택(linked stack) node 정의>

```
typedef int element;  
typedef struct StackNode {  
    element data;  
    struct StackNode *link;  
} StackNode;
```

```
typedef struct {  
    StackNode *top;  
} LinkedStackType;
```

- top은 포인터뿐이지만 일관성을 위하여 구조체 타입으로 정의
- 삽입 연산은 단순 연결 리스트에서 맨 앞에 데이터를 삽입하는 것과 동일
- 연결된 스택에서 헤드 포인터가 top이라는 이름으로 불리는 것 이외에는 차이점 없음.

연결된 스택

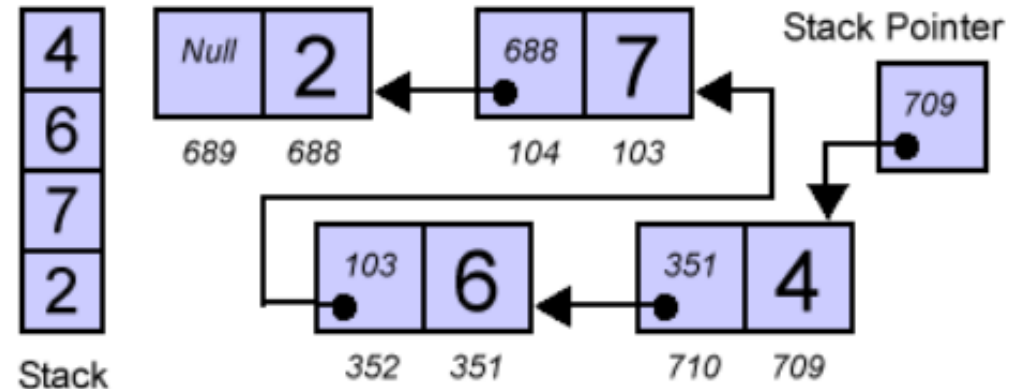
<연결된 스택(linked stack) 연산 구현>

1. 삽입 연산

```
void push(LinkedStackType *s, element item)
{
    StackNode *temp = (StackNode *)malloc(sizeof(StackNode));
    temp->data = item;
    temp->link = s->top;
    s->top = temp;
}
```

2. 삭제 연산

```
element pop(LinkedStackType *s)
{
    if(is_empty(s)) {
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else {
        StackNode *temp = s->top;
        int data = temp->data;
        s->top = s->top->link;
        free(temp);
        return data;
    }
}
```



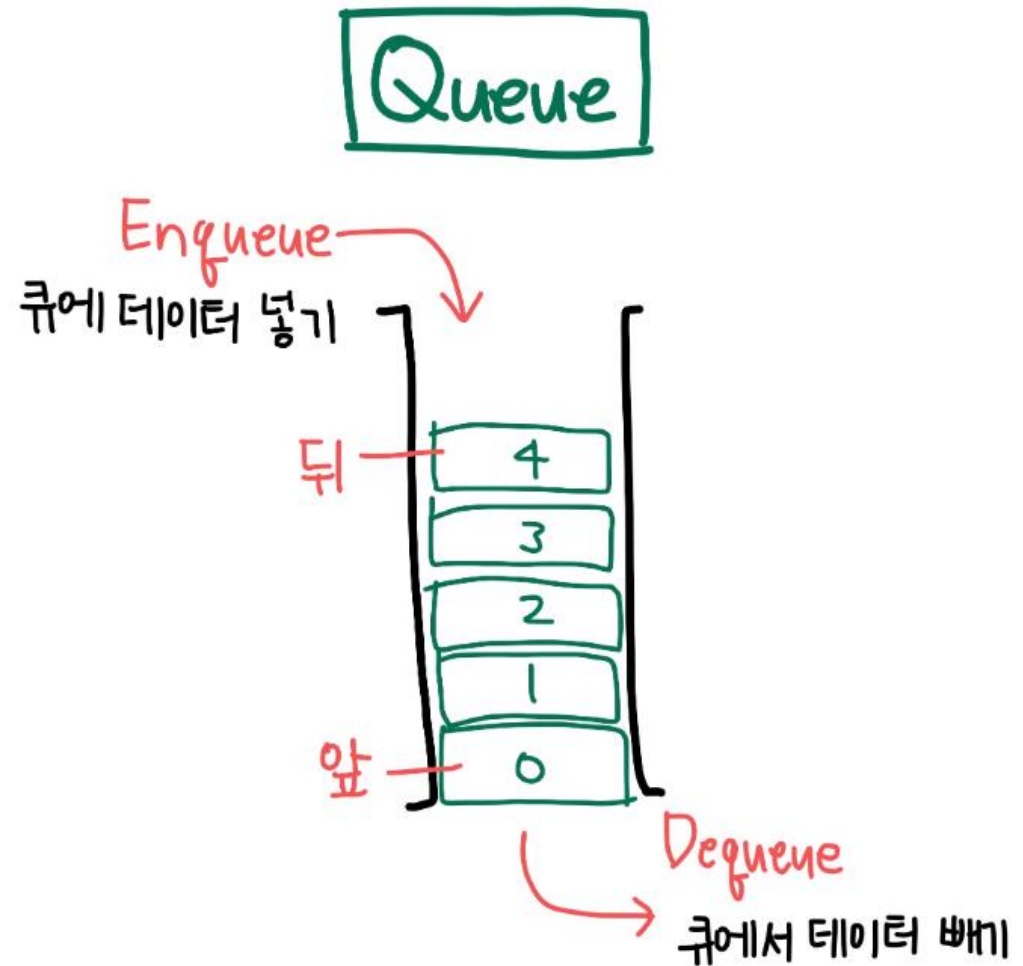
연결된 큐

<연결된 큐(linked queue) node 정의>

```
typedef int element;  
typedef struct QueueNode {  
    element data;  
    struct StackNode *link;  
} QueueNode;
```

```
typedef struct {  
    QueueNode *front, *rear;  
} LinkedQueueType;
```

공백상태인 경우는 front, rear가 모두 NULL인 경우이다.



연결된 큐

<연결된 큐(linked queue) 연산 구현>

1. 삽입 연산

```
void enqueue(LinkedQueueType *q, element data)
{
    QueueNode *temp = (QueueNode *)malloc(sizeof(QueueNode));
    temp->data = data;
    temp->link = NULL;
    if(is_empty(q)) {
        q->front = temp;
        q->rear = temp;
    }
    else {
        q->rear->link = temp; // 순서가 중요
        q->rear = temp;
    }
}
```

2. 삭제 연산

```
element dequeue(LinkedQueueType *q)
{
    QueueNode *temp = q->front;
    element data;
    if(is_empty(q)) {
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else {
        data = temp->data;
        q->front = q->front->link;
        if(q->front == NULL)
            q->rear = NULL;
        free(temp);
        return data;
    }
}
```

감사합니다.