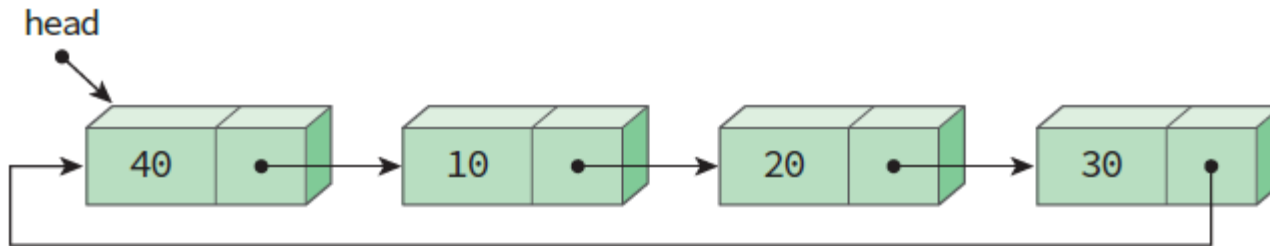


7장 리스트

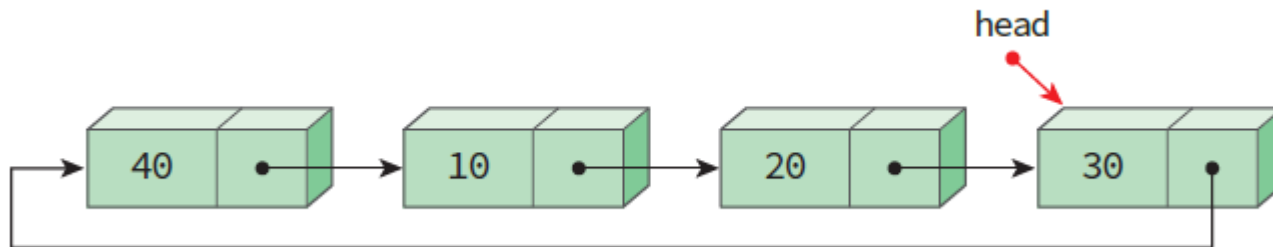


원형 연결 리스트

- 마지막 노드의 링크가 첫 번째 노드를 가리키는 리스트
- 한 노드에서 다른 모든 노드로의 접근이 가능

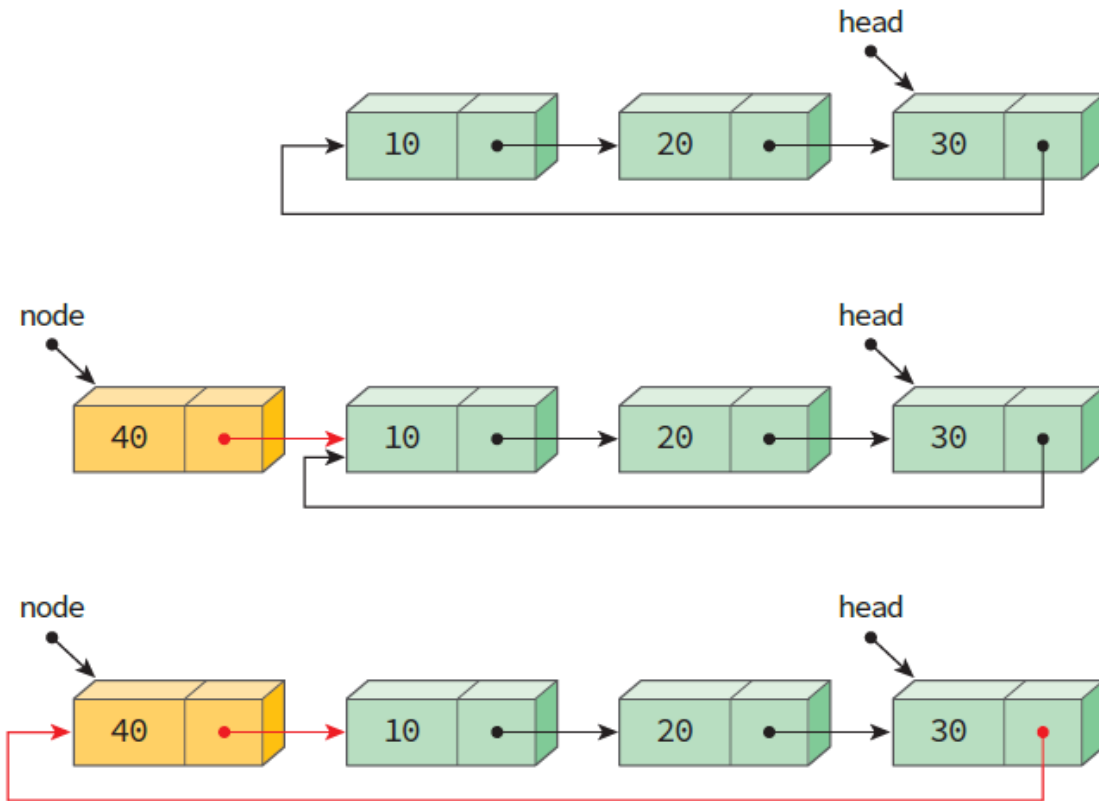


- 보통 헤드포인터가 마지막 노드를 가리키게끔 구성하면 리스트의 처음이나 마지막에 노드를 삽입하는 연산이 단순 연결 리스트에 비하여 용이





연결 리스트의 처음에 삽입





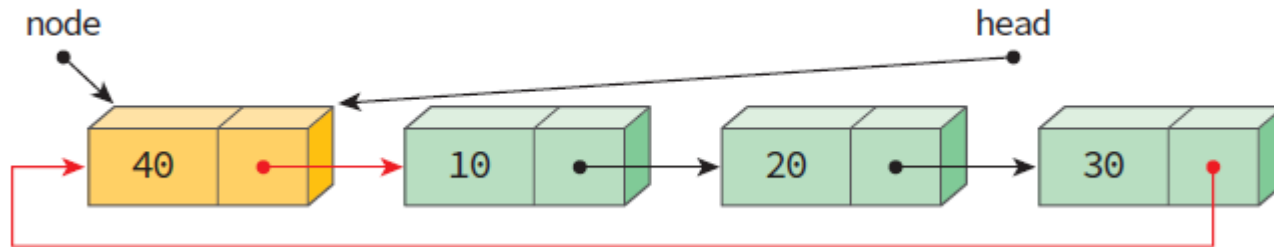
연결 리스트의 처음에 삽입

```
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;    // (1)
        head->link = node;         // (2)
    }
    return head;    // 변경된 헤드 포인터를 반환한다.
}
```





리스트의 끝에 삽입





리스트의 끝에 삽입

```
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;    // (1)
        head->link = node;          // (2)
        head = node;                // (3)
    }
    return head;    // 변경된 헤드 포인터를 반환한다.
}
```





테스트 프로그램

```
#include <stdio.h>
#include <stdlib.h>

typedef int element;
typedef struct ListNode {    // 노드 타입
    element data;
    struct ListNode *link;
} ListNode;

// 리스트의 항목 출력
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); // 마지막 노드 출력
}
```





테스트 프로그램

```
int main(void)
{
    ListNode *head = NULL;

    // list = 10->20->30->40
    head = insert_last(head, 20);
    head = insert_last(head, 30);
    head = insert_last(head, 40);
    head = insert_first(head, 10);
    print_list(head);
    return 0;
}
```

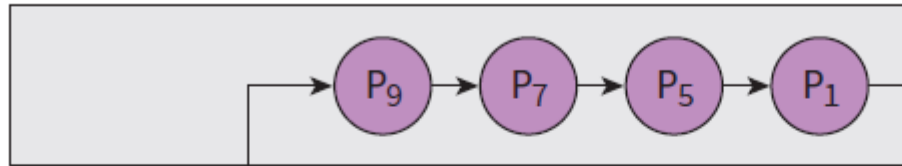
10->20->30->40->



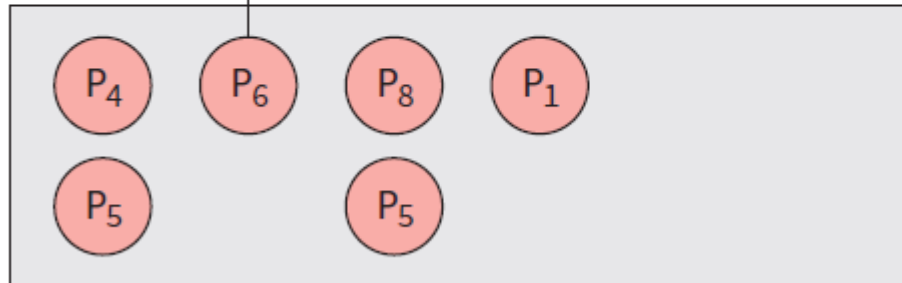


언제 연결 리스트의 응용

준비큐



대기 상태 → 준비 상태



대기 큐

실행



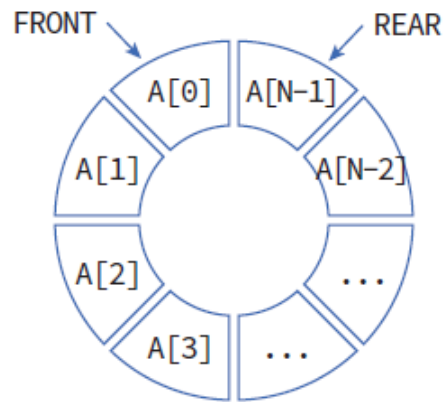
완료

입출력 대기

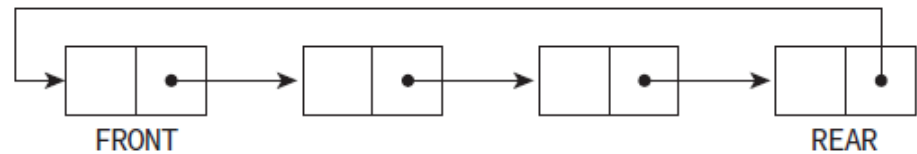




원형 연결 리스트의 응용



배열을 이용한 원형큐



연결 리스트를 이용한 원형큐





멀티 플레이어 게임

현재 차례=KIM
현재 차례=CHOI
현재 차례=PARK
현재 차례=KIM
현재 차례=CHOI
현재 차례=PARK
현재 차례=KIM
현재 차례=CHOI
현재 차례=PARK
현재 차례=KIM





```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
typedef char element[100];
typedef struct ListNode {    // 노드 타입
    element data;
    struct ListNode *link;
} ListNode;
```

```
ListNode* insert_first(ListNode* head, element data)
{
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));
    strcpy(node->data, data);
    if (head == NULL) {
        head = node;
        node->link = head;
    }
    else {
        node->link = head->link;    // (1)
        head->link = node;        // (2)
    }
    return head;    // 변경된 헤드 포인터를 반환한다.
}
```





멀티 플레이어 게임

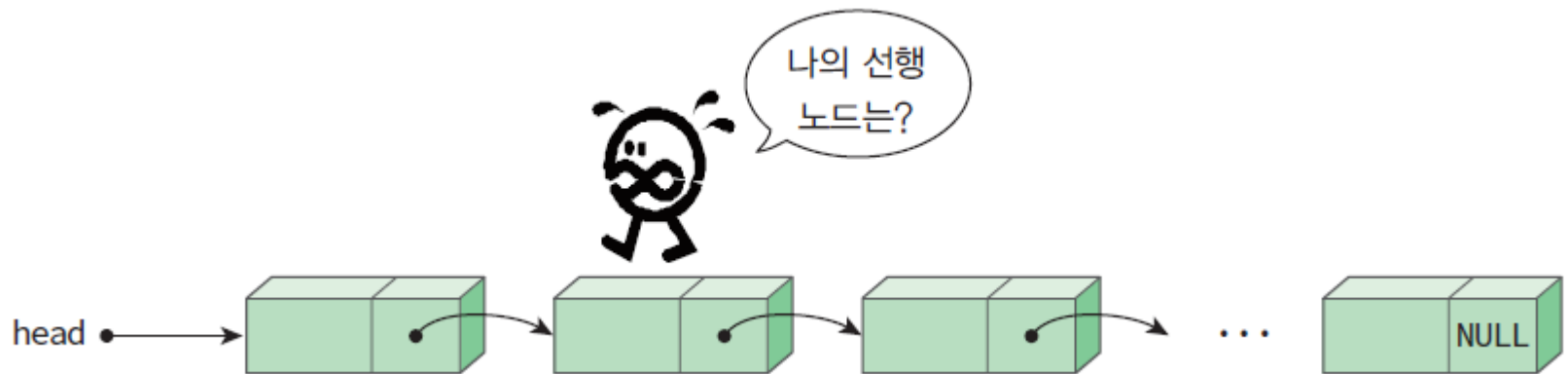
```
// 원형 연결 리스트 테스트 프로그램
int main(void)
{
    ListNode *head = NULL;

    head = insert_first(head, "KIM");
    head = insert_first(head, "PARK");
    head = insert_first(head, "CHOI");

    ListNode* p = head;
    for (int i = 0; i < 10; i++) {
        printf("현재 차례=%s \n", p->data);
        p = p->link;
    }
    return 0;
}
```



- 단순 연결 리스트의 문제점: 선행 노드를 찾기가 힘들다

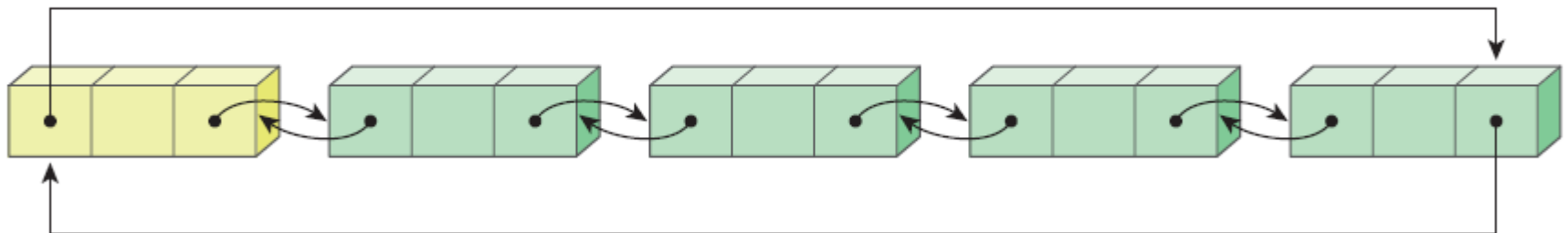




이중 연결 리스트

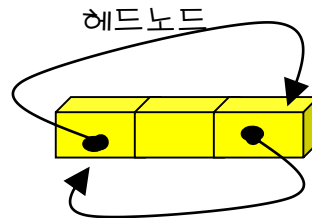
- 이중 연결 리스트: 하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트
- 단점은 공간을 많이 차지하고 코드가 복잡

헤드노드





- 헤드노드(head node): 데이터를 가지지 않고 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드
 - ▣ 헤드 포인터와의 구별 필요
 - ▣ 공백상태에서는 헤드 노드만 존재



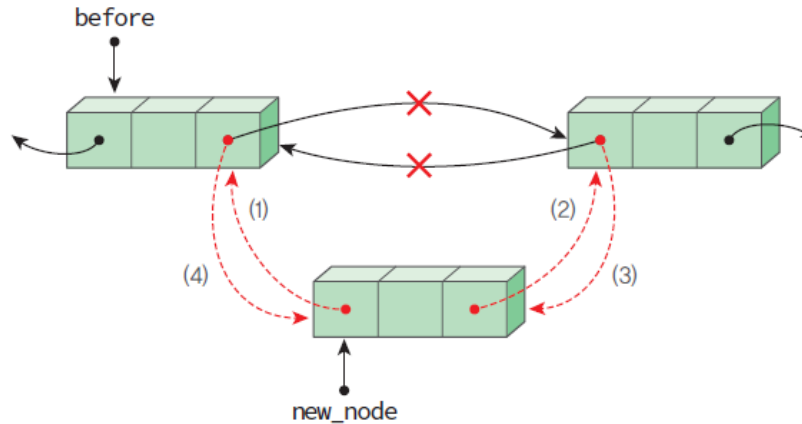


노드의 구조

□ 이중연결리스트에서의 노드의 구조

```
typedef int element;  
typedef struct DlistNode {  
    element data;  
    struct DlistNode *llink;  
    struct DlistNode *rlink;  
} DlistNode;
```





// 새로운 데이터를 노드 before의 오른쪽에 삽입한다.

```
void dinsert(DListNode *before, element data)
```

```
{
```

```
    DListNode *newnode = (DListNode *)malloc(sizeof(DListNode));
```

```
    strcpy(newnode->data, data);
```

```
    newnode->llink = before;
```

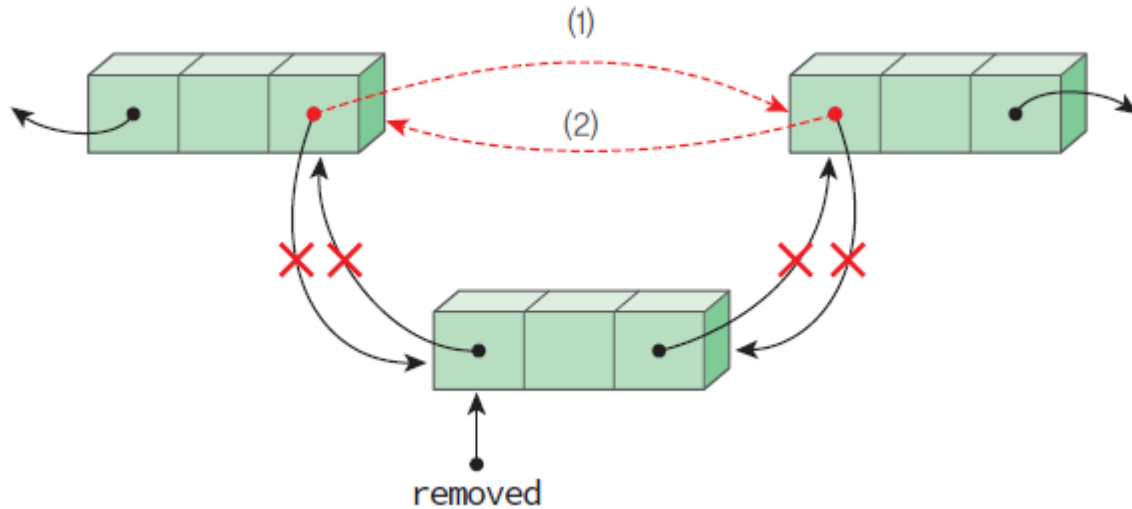
```
    newnode->rlink = before->rlink;
```

```
    before->rlink->llink = newnode;
```

```
    before->rlink = newnode;
```

```
}
```





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





테스트 프로그램

```
#include <stdio.h>
#include <stdlib.h>

typedef int element;
typedef struct DListNode {    // 이중연결 노드 타입
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;

// 이중 연결 리스트를 초기화
void init(DListNode* phead)
{
    phead->llink = phead;
    phead->rlink = phead;
}
```





테스트 프로그램

```
// 이중 연결 리스트의 노드를 출력
void print_dlist(DListNode* phead)
{
    DListNode* p;
    for (p = phead->rlink; p != phead; p = p->rlink) {
        printf("<- | |%d| |-> ", p->data);
    }
    printf("\n");
}

// 새로운 데이터를 노드 before의 오른쪽에 삽입한다.
void dinser(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;
}
```





테스트 프로그램

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





테스트 프로그램

// 이중 연결 리스트 테스트 프로그램

```
int main(void)
```

```
{
```

```
    DListNode* head = (DListNode *)malloc(sizeof(DListNode));
```

```
    init(head);
```

```
    printf("추가 단계\n");
```

```
    for (int i = 0; i < 5; i++) {
```

```
        // 헤드 노드의 오른쪽에 삽입
```

```
        dinsert(head, i);
```

```
        print_dlist(head);
```

```
    }
```

```
    printf("\n삭제 단계\n");
```

```
    for (int i = 0; i < 5; i++) {
```

```
        print_dlist(head);
```

```
        ddelete(head, head->rlink);
```

```
    }
```

```
    free(head);
```

```
    return 0;
```

```
}
```



추가 단계

<- | 0 | ->

<- | 1 | -> <- | 0 | ->

<- | 2 | -> <- | 1 | -> <- | 0 | ->

<- | 3 | -> <- | 2 | -> <- | 1 | -> <- | 0 | ->

<- | 4 | -> <- | 3 | -> <- | 2 | -> <- | 1 | -> <- | 0 | ->

삭제 단계

<- | 4 | -> <- | 3 | -> <- | 2 | -> <- | 1 | -> <- | 0 | ->

<- | 3 | -> <- | 2 | -> <- | 1 | -> <- | 0 | ->

<- | 2 | -> <- | 1 | -> <- | 0 | ->

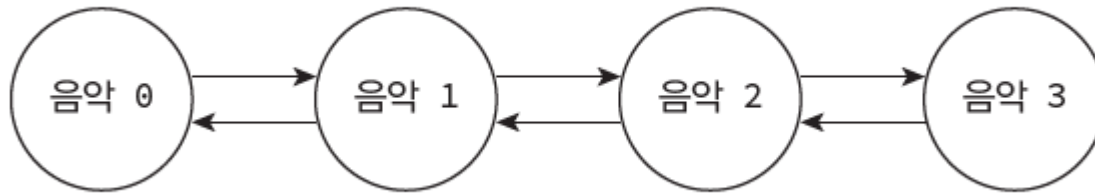
<- | 1 | -> <- | 0 | ->

<- | 0 | ->





mp3 재생 프로그램 만들기



<- | #Fernando# |-> <- | Dancing Queen |-> <- | Mamamia |->

명령어를 입력하시오(<, >, q): >

<- | Fernando |-> <- | #Dancing Queen# |-> <- | Mamamia |->

명령어를 입력하시오(<, >, q): >

<- | Fernando |-> <- | Dancing Queen |-> <- | #Mamamia# |->

명령어를 입력하시오(<, >, q): <

<- | Fernando |-> <- | #Dancing Queen# |-> <- | Mamamia |->

명령어를 입력하시오(<, >, q):





테스트 프로그램

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef char element[100];
typedef struct DListNode {    // 이중연결 노드 타입
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;

DListNode* current;

// 이중 연결 리스트를 초기화
void init(DListNode* phead)
{
    phead->llink = phead;
    phead->rlink = phead;
}
```





테스트 프로그램

```
// 이중 연결 리스트 테스트 프로그램
```

```
int main(void)
```

```
{
```

```
    char ch;
```

```
    DListNode* head = (DListNode *)malloc(sizeof(DListNode));
```

```
    init(head);
```

```
    dinsert(head, "Mamamia");
```

```
    dinsert(head, "Dancing Queen");
```

```
    dinsert(head, "Fernando");
```

```
    current = head->rlink;
```

```
    print_dlist(head);
```





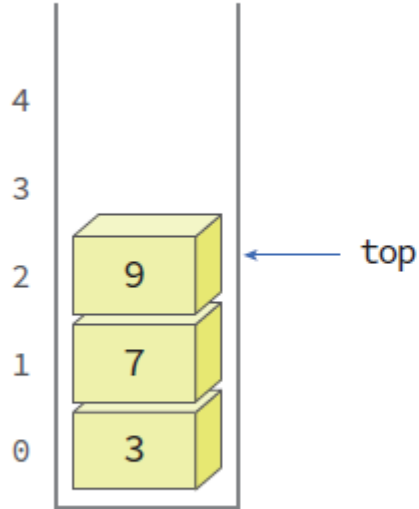
테스트 프로그램

```
do {  
    printf("\n명령어를 입력하시오(<, >, q): ");  
    ch = getchar();  
    if (ch == '<') {  
        current = current->llink;  
        if (current == head)  
            current = current->llink;  
    }  
    else if (ch == '>') {  
        current = current->rlink;  
        if (current == head)  
            current = current->rlink;  
    }  
    print_dlist(head);  
    getchar();  
} while (ch != 'q');  
// 동적 메모리 해제 코드를 여기에  
}
```

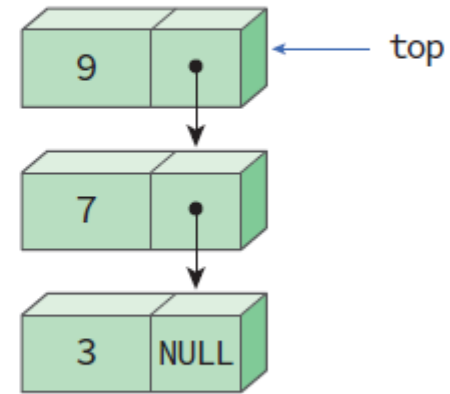




연결 리스트로 구현한 스택



(a) 배열을 이용한 스택



(b) 연결 리스트를 이용한 스택





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

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

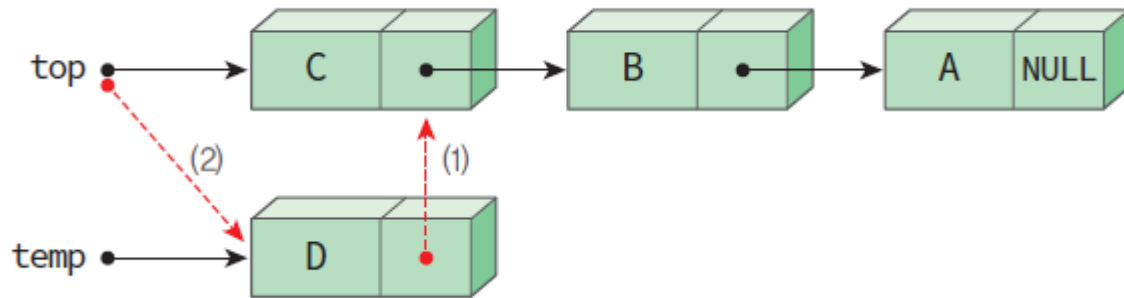




```
typedef int element;
typedef struct StackNode {
    element data;
    struct StackNode *link;
} StackNode;

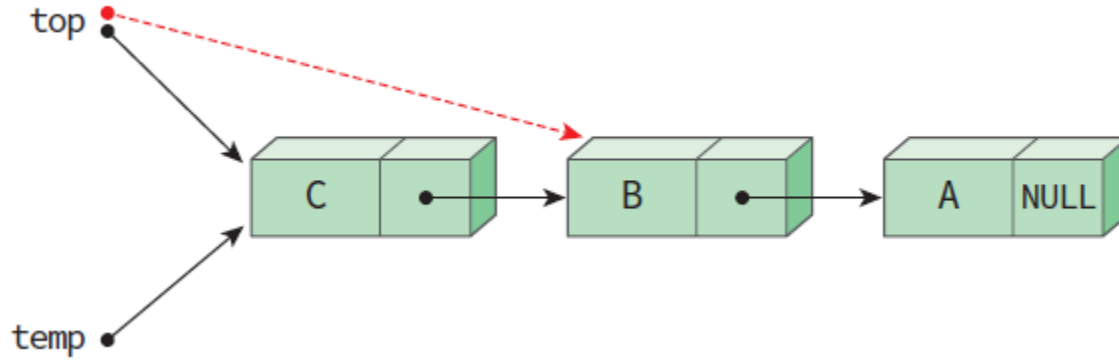
typedef struct {
    StackNode *top;
} LinkedStackType;
```







삽제 연산





```
#include <stdio.h>
#include <malloc.h>

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

typedef struct {
    StackNode *top;
} LinkedStackType;
// 초기화 함수
void init(LinkedStackType *s)
{
    s->top = NULL;
}
```





// 공백 상태 검출 함수

```
int is_empty(LinkedStackType *s)
{
    return (s->top == NULL);
}
```

// 포화 상태 검출 함수

```
int is_full(LinkedStackType *s)
{
    return 0;
}
```

// 삽입 함수

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

void print_stack(LinkedStackType *s)

```
{
    for (StackNode *p = s->top; p != NULL; p = p->link)
        printf("%d->", p->data);
    printf("NULL \n");
}
```





```
// 삭제 함수
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;
    }
}
```





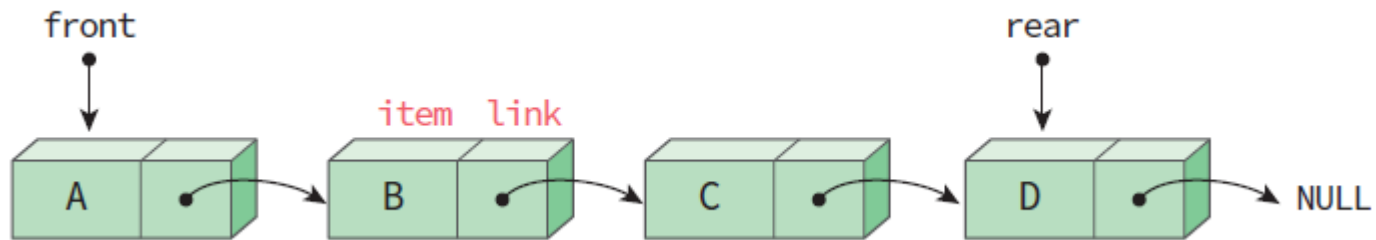
```
// 주 함수
int main(void)
{
    LinkedStackType s;
    init(&s);
    push(&s, 1); print_stack(&s);
    push(&s, 2); print_stack(&s);
    push(&s, 3); print_stack(&s);
    pop(&s); print_stack(&s);
    pop(&s); print_stack(&s);
    pop(&s); print_stack(&s);
    return 0;
}
```

```
1->NULL
2->1->NULL
3->2->1->NULL
2->1->NULL
1->NULL
NULL
```





연결 리스트로 구현한 큐

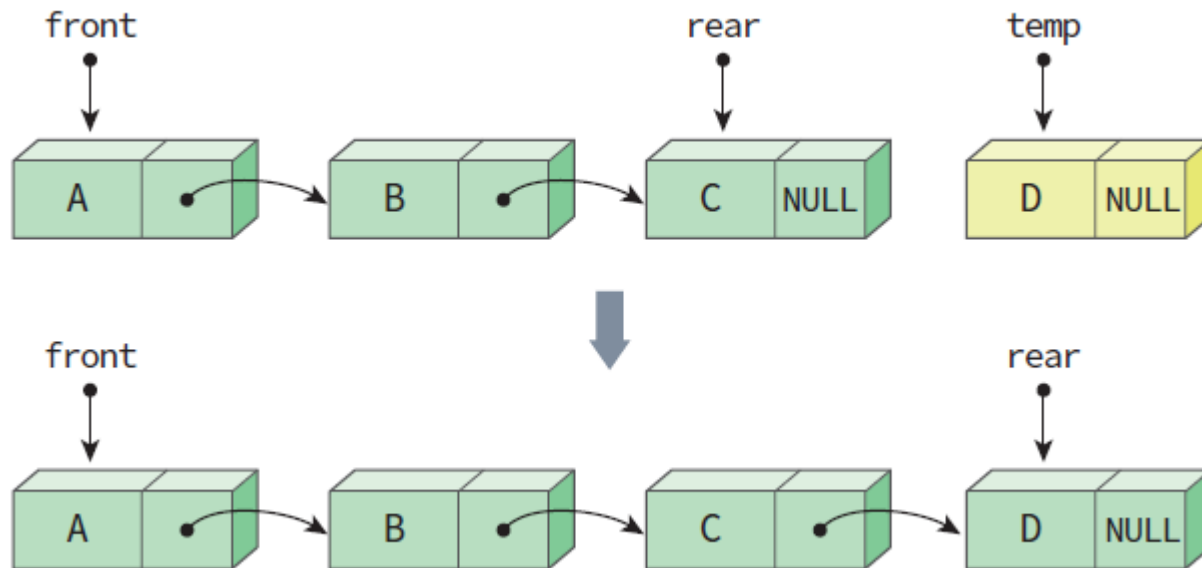




```
typedef int element;           // 요소의 타입
typedef struct QueueNode {     // 큐의 노드의 타입
    element data;
    struct QueueNode *link;
} QueueNode;

typedef struct {               // 큐 ADT 구현
    QueueNode *front, *rear;
} LinkedQueueType;
```

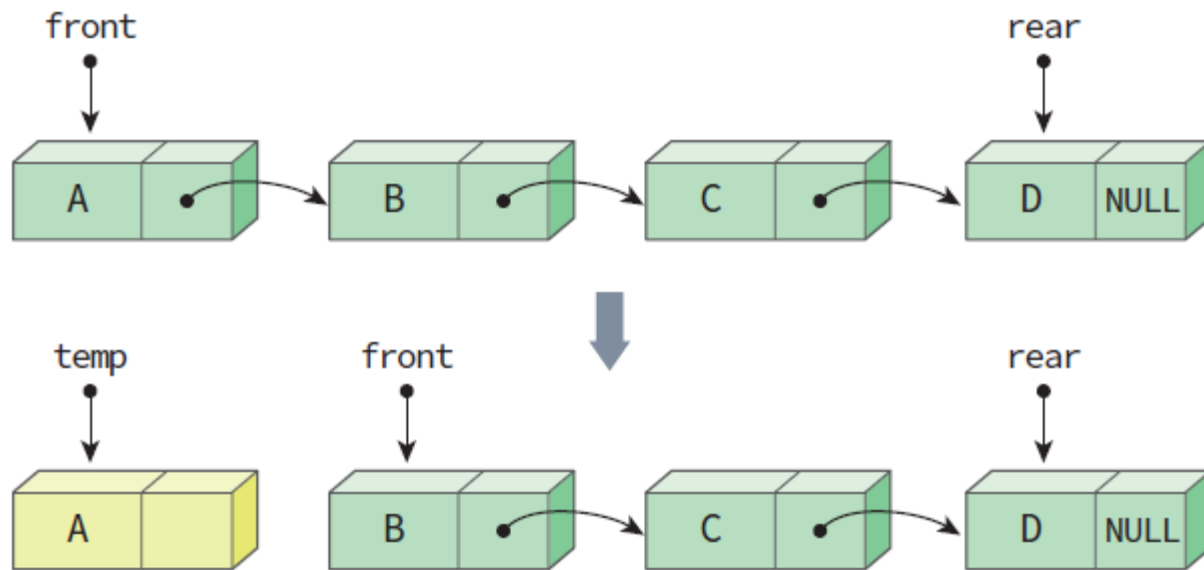






```
// 삽입 함수
void enqueue(LinkedListType *q, element data)
{
    QueueNode *temp = (QueueNode *)malloc(sizeof(QueueNode));
    temp->data = data;                // 데이터 저장
    temp->link = NULL;                // 링크 필드를 NULL
    if (is_empty(q)) {                // 큐가 공백이면
        q->front = temp;
        q->rear = temp;
    }
    else {                            // 큐가 공백이 아니면
        q->rear->link = temp; // 순서가 중요
        q->rear = temp;
    }
}
```







```
// 삭제 함수
element dequeue(LinkedQueueType *q)
{
    QueueNode *temp = q-> front;
    element data;
    if (is_empty(q)) {                // 공백상태
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else {
        data = temp->data;              // 데이터를 꺼낸다.
        q->front = q->front->link; // front로 다음노드
        if (q->front == NULL)         // 공백 상태
            q->rear = NULL;
        free(temp);                   // 동적메모리 해제
        return data;                  // 데이터 반환
    }
}
```

