데이터 구조 7장 실습과제

20223100 박신조

7.3 원형 연결 리스트

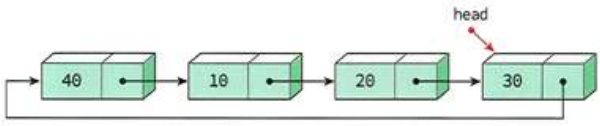
```c
#include <stdio.h>
#include <stdlib.h>

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

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);
}

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;
}

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;
    }
    return head;
}

int main( )
{
    ListNode* head = NULL;

    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;
}
```

```
7.3 원형연결리스트 결과
10->20->30->40->
```

원형 연결 리스트란

일반 연결 리스트는 끝 마지막 노드가 NULL을 가리킨다
원형 리스트는 마지막 노드가 다시 첫 번째 노드를 가리켜서 원처럼 연결되는 리스트이다
따라서 리스트를 순회할 때 NULL이 아닌 시작 노드를 다시 만나는지로 끝을 판단해야한다.

위 코드는 원형 연결 리스트를 구현한 코드이다.
데이터를 삽입할 때 시작부분에 데이터를 추가하는 것과
끝 부분에 데이터를 추가하는 2가지 경우를 확인해 볼 수 있다.



typedef int element

element의 타입을 int로 지정

typedef struct { } ListNode

리스트의 노드부분을 구현

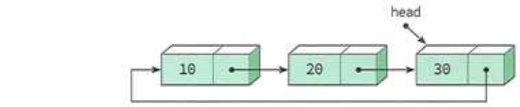
멤버 - data : 노드에 저장되는 정수 값, link : 다음 노드를 나타내는 포인터

void print_list(ListNode* head)

리스트의 원소를 출력하는 함수

head가 비어 있으면 바로 리턴

그렇지 않으면 head->link부터 시작해서 다시 head를 만날 때까지 순회하며 값



ListNode* insert_first(ListNode* head, element data)

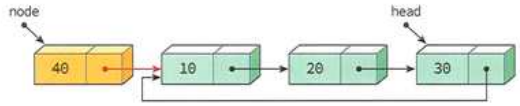
리스트의 시작 부분에 새 노드를 삽입.

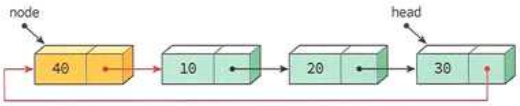
만약 리스트가 공백 상태 라면 새 노드를 만들고, 자기 자신을 가리키게 함

그렇지 않다면새 노드를 head 뒤에 끼워 넣음.

head->link가 가리키던 걸 새 노드가 가리키고,

head는 여전히 마지막 노드를 가리킴.





ListNode* insert_last(ListNode* head, element data)

리스트의 끝 부분에 새 노드를 삽입.

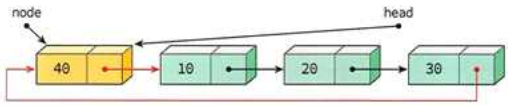
만약 리스트가 공백 상태 라면 새 노드를 만들고 자기 자신을 가리키게 함.

리스트가 비어 있지 않으면:

새 노드를 head 뒤에 삽입하고,

head를 새 노드로 바꿈.

- 항상 가장 마지막 노드를 head로 유지



int main()

연결리스트의 head를 null로 초기화

insert_first, insert_last 함수를 호출하여 노드들 추가.

print_list 함수를 호출하여 원형리스트의 현재 상태 출력

insert_first를 호출하면 head는 여전히 마지막 노드를 가리킴

insert_last를 호출하면 head가 새로 만들어진 노드를 가리킴

7.4 multigame

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

typedef char element[100];
typedef struct {
    element data;
    struct ListNode* link;
}ListNode;

ListNode* insert_first(ListNode* head, element data)
{
    ListNode* node = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(node->data, sizeof(node->data), data);
    if (head == NULL) {
        head = node;
        node->link = head;
    }
    else {
        node->link = head->link;
        head->link = node;
    }
    return head;
}

ListNode* insert_last(ListNode* head, element data)
{
    ListNode* node = (ListNode*)malloc(sizeof(ListNode));
    strcpy_s(node->data, sizeof(node->data), data);
    if (head == NULL) {
        head = node;
        node->link = head;
    }
    else {
        node->link = head->link;
        head->link = node;
        head = node;
    }
    return head;
}
int main()
{
    ListNode* head1 = NULL;
    ListNode* head2 = NULL;
    printf("7.4 multigame 결과\n");
    head1 = insert_first(head1, "KIM");
    head1 = insert_first(head1, "PARK");
    head1 = insert_first(head1, "CHOI");

    head2 = insert_last(head2, "KIM");
    head2 = insert_last(head2, "PARK");
    head2 = insert_last(head2, "CHOI");

    ListNode* p1 = head1;
    ListNode* p2 = head2;
    printf("insert_first호출시\n");
    for (int i = 0; i < 10; i++) {
        printf("현재차례=%s \n", p1->data);
        p1 = p1->link;
    }
    printf("\ninsert_last호출시\n");
    for (int i = 0; i < 10; i++) {
        printf("현재차례=%s \n", p2->data);
        p2 = p2->link;
    }
    return 0;
}
```

```
7.4 multigame 결과
insert_first호출시
현재차례=KIM
현재차례=CHOI
현재차례=PARK
현재차례=KIM
현재차례=CHOI
현재차례=PARK
현재차례=KIM
현재차례=CHOI
현재차례=PARK
현재차례=KIM

insert_last호출시
현재차례=CHOI
현재차례=KIM
현재차례=PARK
현재차례=CHOI
현재차례=KIM
현재차례=PARK
현재차례=CHOI
현재차례=KIM
현재차례=PARK
현재차례=CHOI
```

7.3에서 구현한 원형 연결 리스트를 통해 multigame이라는 코드이다.

typedef char element[100]

element을 문자열(char[100])로 지정

typedef struct { } ListNode

리스트의 노드부분을 구현

멤버 - data : 노드에 저장되는 문자열 값, link : 다음 노드를 나타내는 포인터

ListNode* insert_first(ListNode* head, element data)

리스트의 시작 부분에 새 노드를 삽입.
만약 리스트가 공백 상태 라면 새 노드를 만들고, 자기 자신을 가리키게 함
그렇지 않다면새 노드를 head 뒤에 끼워 넣음.
head->link가 가리키던 걸 새 노드가 가리키고,
head는 여전히 마지막 노드를 가리킴.

ListNode* insert_last(ListNode* head, element data)

리스트의 끝 부분에 새 노드를 삽입.
만약 리스트가 공백 상태 라면 새 노드를 만들고 자기 자신을 가리키게 함.
리스트가 비어 있지 않으면:
새 노드를 head 뒤에 삽입하고,
head를 새 노드로 바꿈.
 - 항상 가장 마지막 노드를 head로 유지

int main()

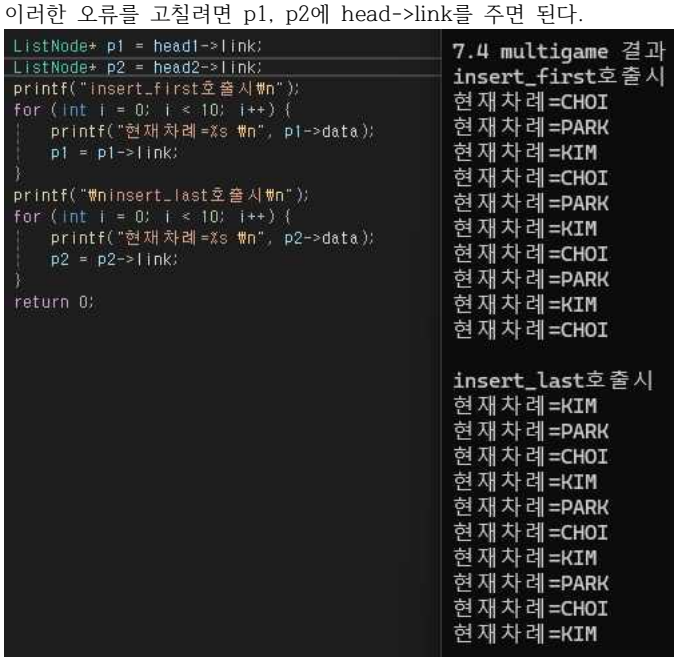
두 리스트의 head를 초기화
insert_first와 insert_last의 차이점을 알기 위해 두 개의 리스트 생성
각각 KIM, PARK, CHOI의 문자열을 차례대로 삽입
head1은 insert_first로, head2는 insert_last로
이후 각각 for문을 통해 출력

head1은 KIM -> PARK -> CHOI , haed2는 CHOI -> KIM -> PARK으로 출력된다.
하지만 여기에는 약간의 오류가 있다.
첫번째 노드부터 출력하지 않고 head부터 출력하기 때문에 맨 마지막 노드부터 출력이 된다.

insert_first에서는 head는 맨 마지막 노드를 계속 유지하는데
삽입 순서는 "CHOI" → "PARK" → "KIM" (head는 "KIM"을 가리킴) 이지만
출력 순서는 “KIM” → "CHOI" → "PARK" 가 된다.

insert_last에서는 매번 head가 새로 삽입된 노드를 가리키게 된다.
삽입 순서는 "KIM" → "PARK" → "CHOI" (head는 "CHOI"을 가리킴) 이지만
출력 순서는 "CHOI" → "KIM" → "PARK" 가 된다.

이러한 오류를 고칠려면 p1, p2에 head->link를 주면 된다.

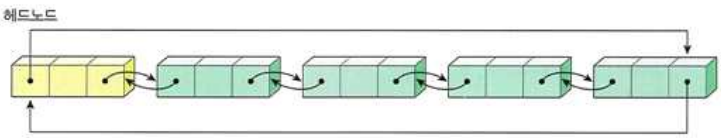


7.7 이중 연결 리스트

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

typedef int element;
typedef struct DListNode {
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;

void init(DListNode* phead)
{
    phead->llink = phead;
    phead->rlink = phead;
}

void print_dlist(DListNode* phead)
{
    DListNode* p;
    for (p = phead->rlink; p != phead; p = p->rlink) {
        printf("<-| |%d| |-> ", p->data);
    }
    printf("₩n");
}

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

void ddelete(DListNode* head, DListNode* removed)
{
    if (removed == head) return;
    removed->llink->rlink = removed->rlink;
    removed->rlink->llink = removed->llink;
    free(removed);
}
int main( )
{
    DListNode* head = (DListNode*)malloc(sizeof(DListNode));
    init(head);
    printf("추가단계₩n");
    for (int i = 0; i < 5; i++) {
        dinsert(head, i);
        print_dlist(head);
    }
    printf("₩n삭제단계₩n");
    for (int i = 0; i < 5; i++) {
        print_dlist(head);
        ddelete(head, head->rlink);
    }
    free(head);
    return 0;
}
```

```
7.7 이중연결리스트 결과
추가단계
<-| |0| |->
<-| |1| |-> <-| |0| |->
<-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |4| |-> <-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->

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

이중연결리스트를 구현한 코드이다.
원형 연결리스트와는 다르게 이전노드, 다음노드에 연결되어 있어 양방향으로 연결되어 있는 구조이다.

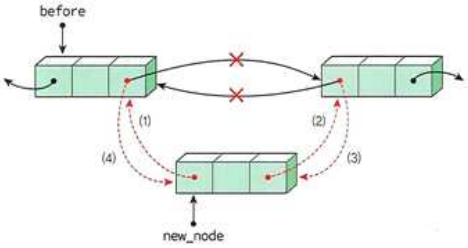


typedef int element

element의 타입을 int로 지정

typedef struct DListNode { } DListNode

이중연결 리스트의 노드를 구현
멤버 - data : 노드에 저장되는 문자열 값, llink : 이전 노드를 나타내는 포인터, rlink : 다음 노드를 나타내는 포인터



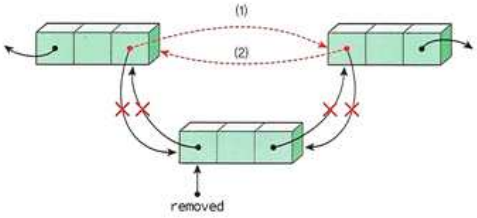
void init(DListNode* phead)

리스트를 초기화 해 주는 함수
자기 자신을 왼쪽과 오른쪽으로 가리키게 해서 원형 구조를 만듦.

void dinsert(DListNode* before, element data)

주어진 before 노드 뒤쪽에 새 노드를 삽입하는 함수
새 노드의 왼쪽은 before, 오른쪽은 before->rlink
기존 노드들의 링크도 이중 연결 방식으로 변경

void ddelete(DListNode* head, DListNode* removed)



removed 노드를 리스트에서 삭제하는 함수
삭제할 노드가 head이면 리턴
삭제할 노드 이전, 다음 노드끼리 연결 시켜 준 후 메모리 반납

void print_dlist(DListNode* phead)

리스트의 원소를 출력하는 함수
phead->rlink에서 시작함 (첫 번째 노드부터 출력 시작)
p != phead인 동안 반복순회 (다시 phead를 만나면 종료)

int main()

이중연결리스트 head를 동적으로 할당
init 함수를 통해 초기화
값을 삽입하고 리스트의 현재 상태 출력
 - 삽입되는 순서를 보면 리스트의 앞쪽에 삽입되는 것처럼 보임 (head 뒤에 매번 새 노드 삽입).
값을 하나씩 삭제하며 리스트의 현재 상태 출력
 - 순서대로 맨 앞 노드를 하나씩 제거

7.9 연결된 스택

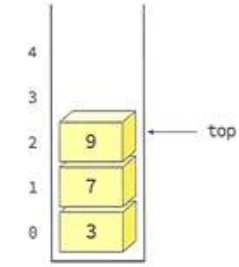
```c
#include <stdio.h>
#include <malloc.h>

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

typedef struct {
    StackNode* top;
} LinkedStackType;

void init(LinkedStackType* s)
{
    s->top = NULL;
}

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

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

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");
}
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;
    }
}

element peek(LinkedStackType* s)
{
    if (is_empty(s)) {
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else {
        return s->top->data;
    }
}

int main()
{
    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;
}
```

```
7.9 연결된 스택 결과
1->NULL
2->1->NULL
3->2->1->NULL
2->1->NULL
1->NULL
NULL
```

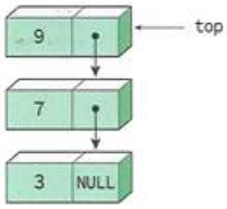
단일 연결 리스트를 활용하여 스택을 구현한 코드이다.
연결된 스택은 단일 연결 리스트에서 맨 앞에 데이터를 삽입하는 것과 동일하다.
연결된 스택에서는 head가 top으로 불리는 것으로 구조적으로는 단일 연결 리스트와 동일하다.
배열 스택와 연결리스트 스택의 가장 큰 차이점은 크기의 제한이다.
연결리스트 스택은 동적으로 메모리를 할당받기 때문에 필요에 따라 크기를 늘릴 수 있다.

typedef int element

element의 타입을 int로 지정



(a) 배열을 이용한 스택



(b) 연결 리스트를 이용한 스택

typedef struct StackNode { } StackNode

스택의 노드를 나타내는 구조체

멤버 - data : 노드에 저장되는 정수 값, link : 다음 노드를 나타내는 포인터

typedef struct { } LinkedStackType

스택의 마지막 데이터를 가리키는 구조체

멤버 - top: 현재 스택의 가장 위(top)를 가리킴

void init(LinkedStackType* s)

스택을 초기화 시키는 함수

스택에서 시작 노드를 NULL로 설정

int is_empty(LinkedStackType* s)

스택이 공백 생태인지 확인하는 함수

top이 null값이면 1 리턴 아니면 0 리턴

int is_full(LinkedStackType* s)

스택이 포화상태인지 확인하는 함수

연결리스트는 데이터가 삽입될 때 마다 메모리를 할당하기 때문에 끝없이 확장 가능

그렇기에 0 리턴

void push(LinkedStackType* s, element item)

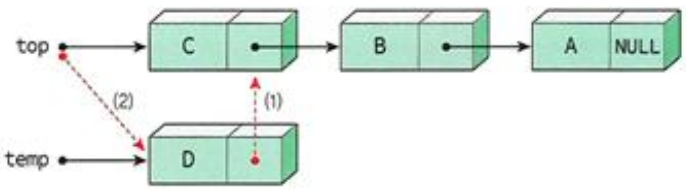
스택에 값을 추가하는 함수(연결리스트에서 맨 앞에 값 추가)

동적으로 새 노드 생성

새로 할당받은 구조체의 data에 데이터 저장

현재 top(추가되기 이전의 노드)을 새 노드의 link에 연결

이후 top을 새 노드로 변경



void print_stack(LinkedStackType* s)

스택의 현재 상태를 출력하는 함수

top부터 시작하여 link를 따라가며 null이 올때까지 하나하나 출력

element pop(LinkedStackType* s)

스택에서 가장 위의 데이터를 삭제하고 반환하는 함수(후입선출)

우선 스택이 공백상태인지 확인 - 공백상태라면 에러문구 출력 후 종료

아니라면 임시 구조체 포인터 temp에 마지막 노드(최근에 생성한 노드) 주소 저장

임시변수 data에 마지막 노드의 data값 저장

top을 한 칸 아래로 이동(마지막 노드가 가리키던 노드로 이동)

temp 메모리 해제(마지막 노드 반환)

data 리턴

element peek(LinkedStackType* s)

마지막 노드에 들어있는 데이터 값을 반환하는 함수

top에 있는 값만 확인 - pop과 다르게 메모리 반환 x

int main()

스택 구조체 선언 및 init 함수 호출하여 스택 초기화

이후 push 함수 호출하여 데이터 추가

값을 모두 추가한 후 pop 함수 호출하여 최근 추가된 노드부터 하나씩 삭제

스택의 구조가 변경될 때 마다 print_stack 함수를 호출하여 스택의 현재상태 출력

7.12 연결된 큐

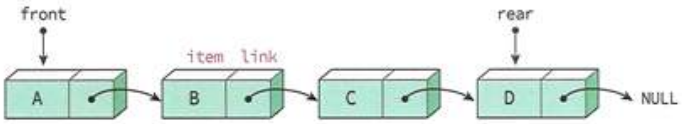
```c
#include <stdio.h>
#include <stdlib.h>

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

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

void init(LinkedQueueType* q){  q->front = q->rear = 0;}

int is_empty(LinkedQueueType* q) {  return (q->front == NULL);}

int is_full(LinkedQueueType* q){ return 0;}

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;
        q->rear = temp;
    }
}

element dequeue(LinkedQueueType* q){
    QueueNode* temp = q->front;
    element data;
    if (is_empty(q)) {
        fprintf(stderr, "큐가 비어있음₩n");
        exit(1);
    }
    else {
        data = temp->data;
        q->front = q->front->link;
        if (q->front == NULL)
            q->rear = NULL;
        free(temp);
        return data;
    }
}
void print_queue(LinkedQueueType* q){
    QueueNode* p;
    for (p = q->front; p != NULL; p = p->link)
        printf("%d->", p->data);
    printf("NULL₩n");
}

int main(void)
{
    LinkedQueueType queue;

    init(&queue);

    enqueue(&queue, 1);
    print_queue(&queue);
    enqueue(&queue, 2);
    print_queue(&queue);
    enqueue(&queue, 3);
    print_queue(&queue);
    dequeue(&queue);
    print_queue(&queue);
    dequeue(&queue);
    print_queue(&queue);
    dequeue(&queue);
    print_queue(&queue);
    return 0;
}
```

```
7.12 연결된 덱 결과
1->NULL
1->2->NULL
1->2->3->NULL
2->3->NULL
3->NULL
NULL
```

단일 연결 리스트를 활용하여 큐을 구현한 코드이다.
연결된 큐은 단일 연결 리스트에서 맨 뒤에 데이터를 삽입하는 것과 동일하다.
단일 연결 리스트에서는 포인터 2개(front, rear)를 추가한 것과 같다.
배열 큐와 연결리스트 큐의 가장 큰 차이점은 크기의 제한이다.
연결리스트 큐는 동적으로 메모리를 할당받기 때문에 필요에 따라 크기를 늘릴 수 있다.

typedef int element

element의 타입을 int로 지정



typedef struct QueueNode { } QueueNode

큐의 노드를 나타내는 구조체

멤버 - data : 노드에 저장되는 정수 값, link : 다음 노드를 나타내는 포인터

typedef struct { } LinkedQueueType

큐의 첫 번째 노드와 마지막 노드를 가리키는 구조체

멤버 - front : 첫 번째 노드를 가리킴, rear : 마지막 노드를 가리킴

void init(LinkedQueueType* q)

큐를 초기화 시키는 함수

큐의 첫 번째 노드와 마지막 노드를 0으로 지정

-> 0으로 지정하면 포인터가 주소 0을 가리키고 있음으로, 원래는 null로 초기화 시켜줘야함

int is_empty(LinkedQueueType* q)

큐가 공백 생태인지 확인하는 함수

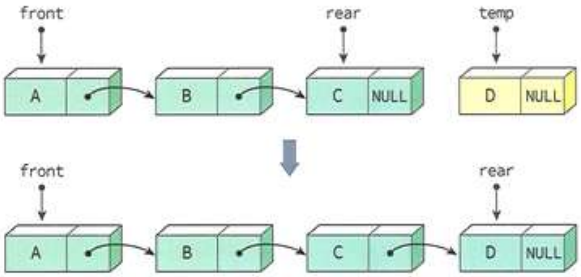
top이 null값이면 1 리턴 아니면 0 리턴

int is_full(LinkedQueueType* q)

큐가 포화상태인지 확인하는 함수

연결리스트는 데이터가 삽입될 때 마다 메모리를 할당하기 때문에 끝없이 확장 가능

그렇기에 0 리턴



void enqueue(LinkedQueueType* q, element data)

큐에 값을 추가하는 함수(연결리스트에서 마지막에 값 추가)

새로운 노드 동적으로 할당 후 데이터 저장

새로운 노드의 link는 null로 지정(마지막 노드이기 때문)

큐가 비어있다면 front와 rear 모두 새 노드를 가리키도록 지정

아니라면 기존 rear의 link를 새 노드로 설정, rear를 새로운 노드로 변경 - rear은 마지막 노드를 가리키는 포인터.

element dequeue(LinkedQueueType* q)

큐에서 첫 번째 노드를 삭제 및 반환하는 함수(선입선출)

임시 구조체 포인터 temp를 삭제할 노드 주소로 설정

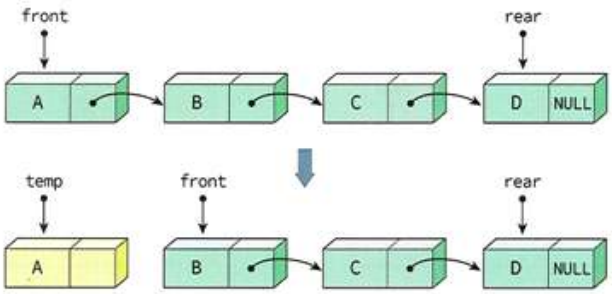
큐가 공백 상태 인지 확인 - 공백 상태 라면 에러 문구 출력 후 종료

임시변수 data에 삭제할 노드의 data 값 저장

front를 첫 번째 노드에서 두 번째 노드로 이동

만약 front가 null이라면 rear도 null 로 변경(큐 공백상태)

이후 첫 번째 노드의 메모리 반환, data값 리턴



void print_queue(LinkedQueueType* q)

큐의 현재 상태를 출력하는 함수

front부터 rear까지 순서대로 출력

int main(void)

큐 구조체 선언 및 init 함수 호출하여 큐 초기화

이후 enqueue 함수 호출하여 데이터 추가

값을 모두 추가한 후 dequeue 함수 호출하여 첫 번째 노드부터 하나씩 삭제

스택의 구조가 변경될 때 마다 print_stack 함수를 호출하여 스택의 현재상태 출력