해달 자료구조

2차시 – 큐

순서

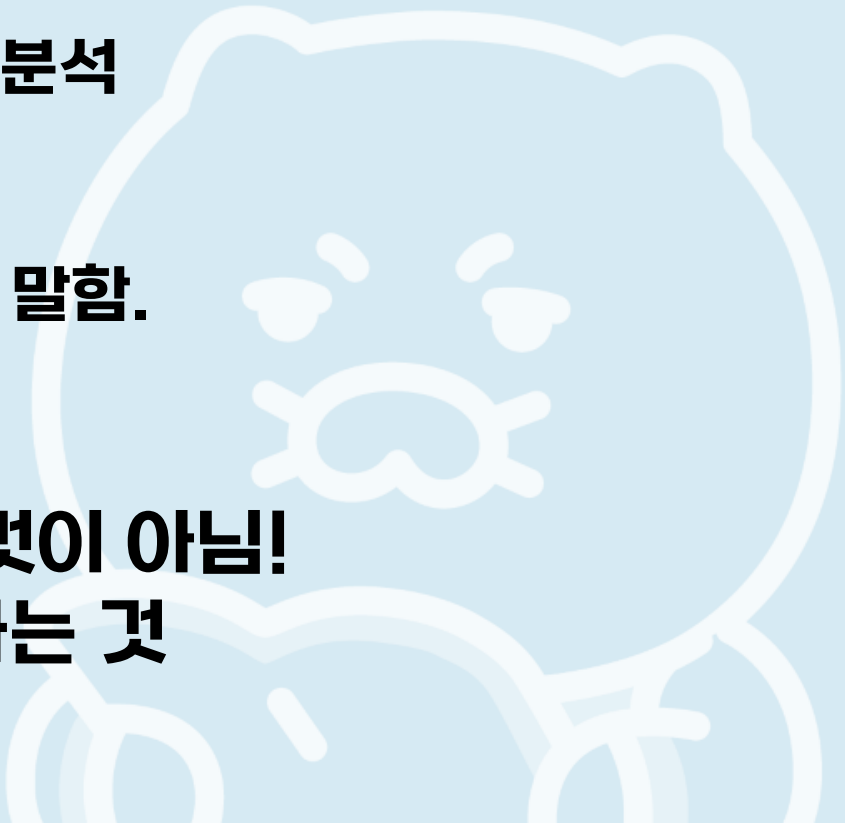
1. 시간복잡도
2. 큐
3. 덱

시간 복잡도(Time complexity)

알고리즘 성능 분석에는 2가지 측면이 고려됨
1. 시간 복잡도 - 알고리즘의 수행 시간 분석
2. 공간 복잡도 – 알고리즘이 사용하는 기억 공간 분석

알고리즘의 복잡도를 이야기할 때 대개는 시간 복잡도를 말함.

시간 복잡도는 절대적인 수행 시간을 나타내는 것이 아님!
연산들이 몇 번이나 수행되는지를 숫자로 표시하는 것



빅오 표기법(Big-O notation)

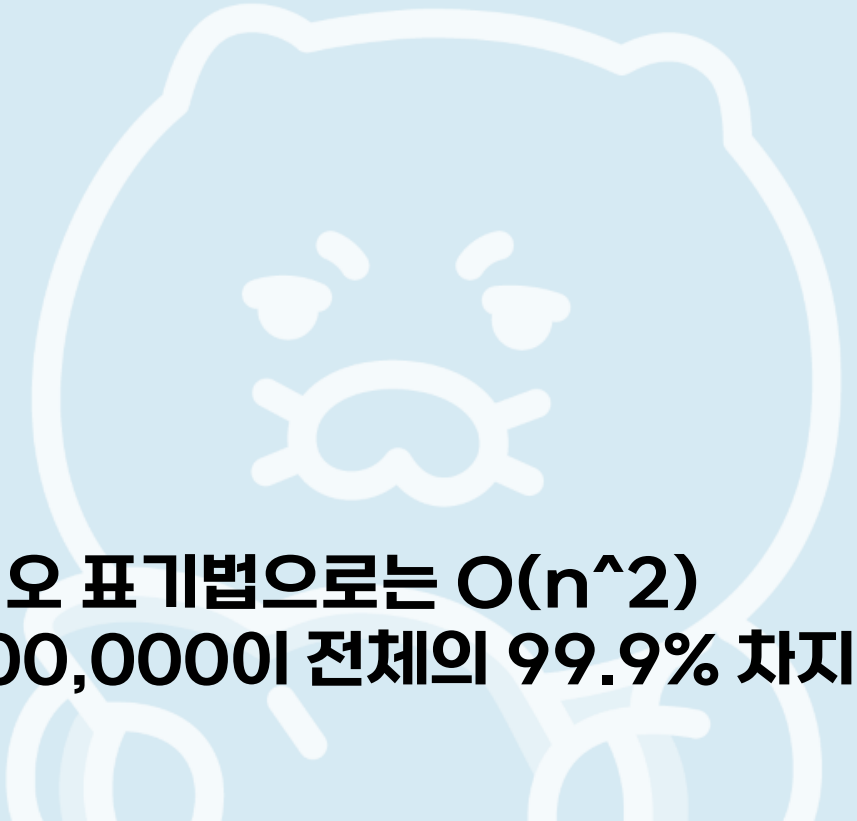
연산의 횟수를 대략적(점진적)으로 표기한 것. 함수의 상한(upper bound) 표시.

→ 알고리즘의 최악의 실행 시간을 표기

빅오 표기법은 다음과 같은 특징을 가짐

1. 상수항 무시
2. 계수 무시
3. 최고차항만 표기

예를 들어 시간 복잡도 함수가 n^2+n+1 이라고 하면 빅오 표기법으로는 O(n^2)
n = 1000 일 때, 1,001,001. 첫 번째 항인 n^2 = 1,000,0000이 전체의 99.9% 차지



빅오 표기법(Big-O notation) 예시

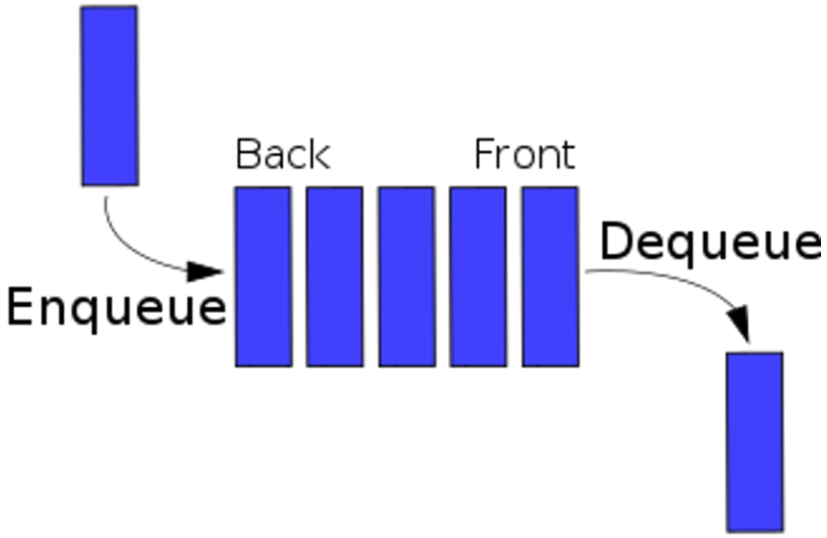
알고리즘 A	알고리즘 B	알고리즘 C

sum ←n*n;	for i←1 to n do
	  sum ←sum + n;
		for i←1 to n do
		  for j←1 to n do
		  sum ←sum + 1;

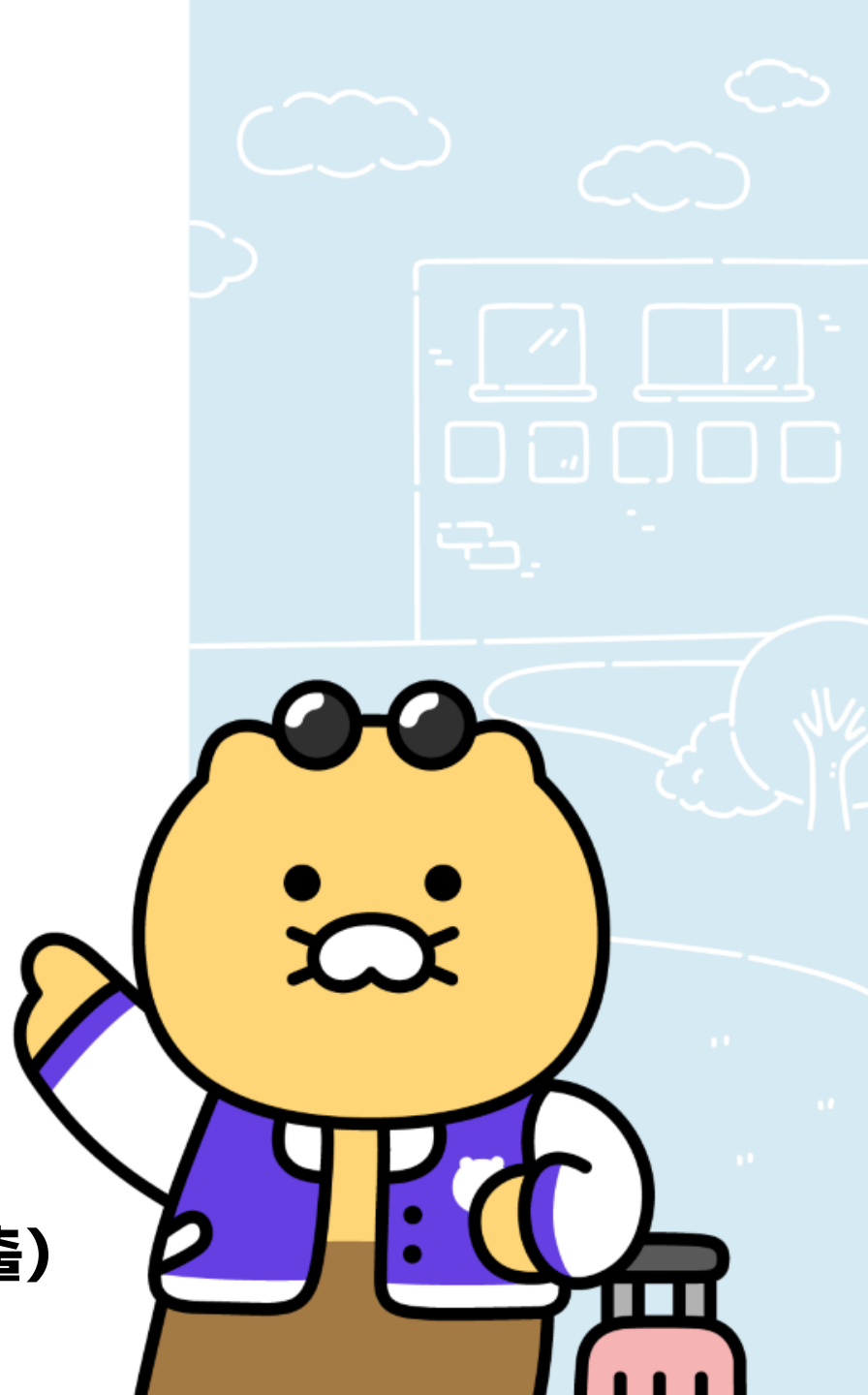
알고리즘 A → O(1)
알고리즘 B → O(n)
알고리즘 C → O(n^2)

큐(Queue)

queue: 줄, 대기 행렬, 줄을 서서 기다리다



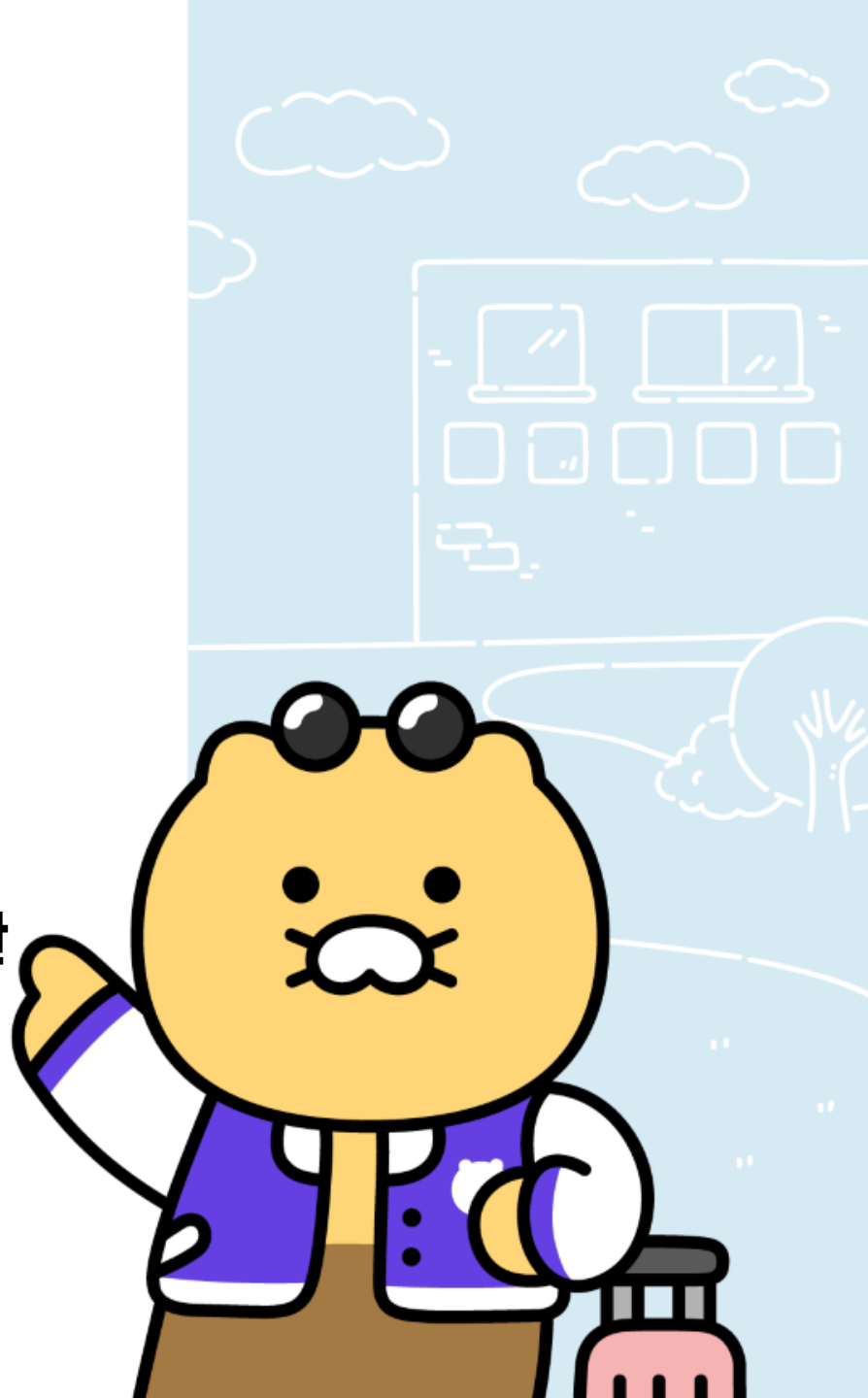
데이터들을 줄지어 놓는 것
먼저 들어온 데이터가 먼저 나가는 구조(FIFO/선입선출)



ADT

큐는 0개 이상의 요소들로 구성된 선형 리스트로 정의됨
다음과 같은 연산들로 이루어져 있음

- is_empty
- is_full
- enqueue: 삽입 연산
- dequeue: 삭제 연산. 맨 앞의 요소를 제거하고 반환
- peek: 맨 앞의 요소를 반환



선형 큐(Linear queue)

1차원 배열을 이용해 구현한 큐.
1차원 배열과 삽입, 삭제를 위한 변수 front와 rear 선언
front와 rear의 초기값은 -1

is_empty

공백 상태 검사.
front와 rear의 값이 같다면
공백 상태로 판단

is_full

포화 상태 검사
rear의 값이
'배열의 크기 – 1'과 같다면
포화 상태로 판단

enqueue

삽입 연산
현재 위치(rear)에 데이터를
저장하고 rear값 증가

dequeue

삭제 연산
현재 위치(front)에
데이터를 삭제하고 front 값
증가

선형 큐 코드

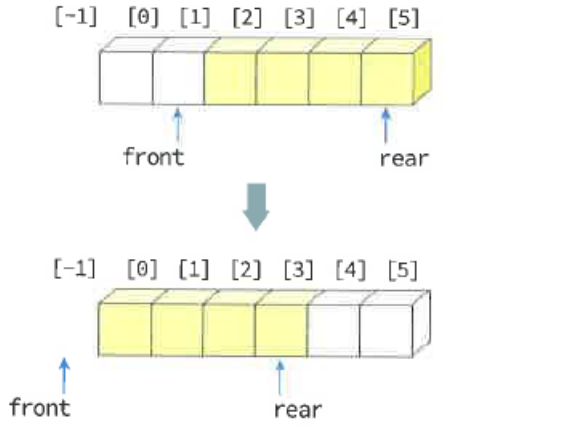
```c
typedef int element;
typedef struct{
   int front;
   int rear;
   element data[MAX_QUEUE_SIZE];
} QueueType;
```

선형 큐 코드

```c
void init_queue(QueueType *q)
{
  q->rear = -1;
  q->front = -1;
}

int is_full(QueueType *q)
{
  if (q->rear == MAX_QUEUE_SIZE - 1)
    return 1;
  else
    return 0;
}

int is_empty(QueueType *q)
{
  if (q->front == q->rear)
    return 1;
  else
    return 0;
}
```

```c
void enqueue(QueueType *q, int item)
{
  if(is_full(q)) {
    error("큐가 포화상태입니다.");
    return;
  }
  q->data[++(q->rear)] = item;
}

int dequeue(QueueType *a)
{
  if(is_empty(q)) {
    error("큐가 공백상태입니다.");
    return -1;
  }
  int item = q->data[++(q->front)];
  return item;
}
```

선형 큐의 단점



[그림 5-6] 선형큐의 경우 이동이 필요하다.

front와 rear의 값이 증가하기만 함

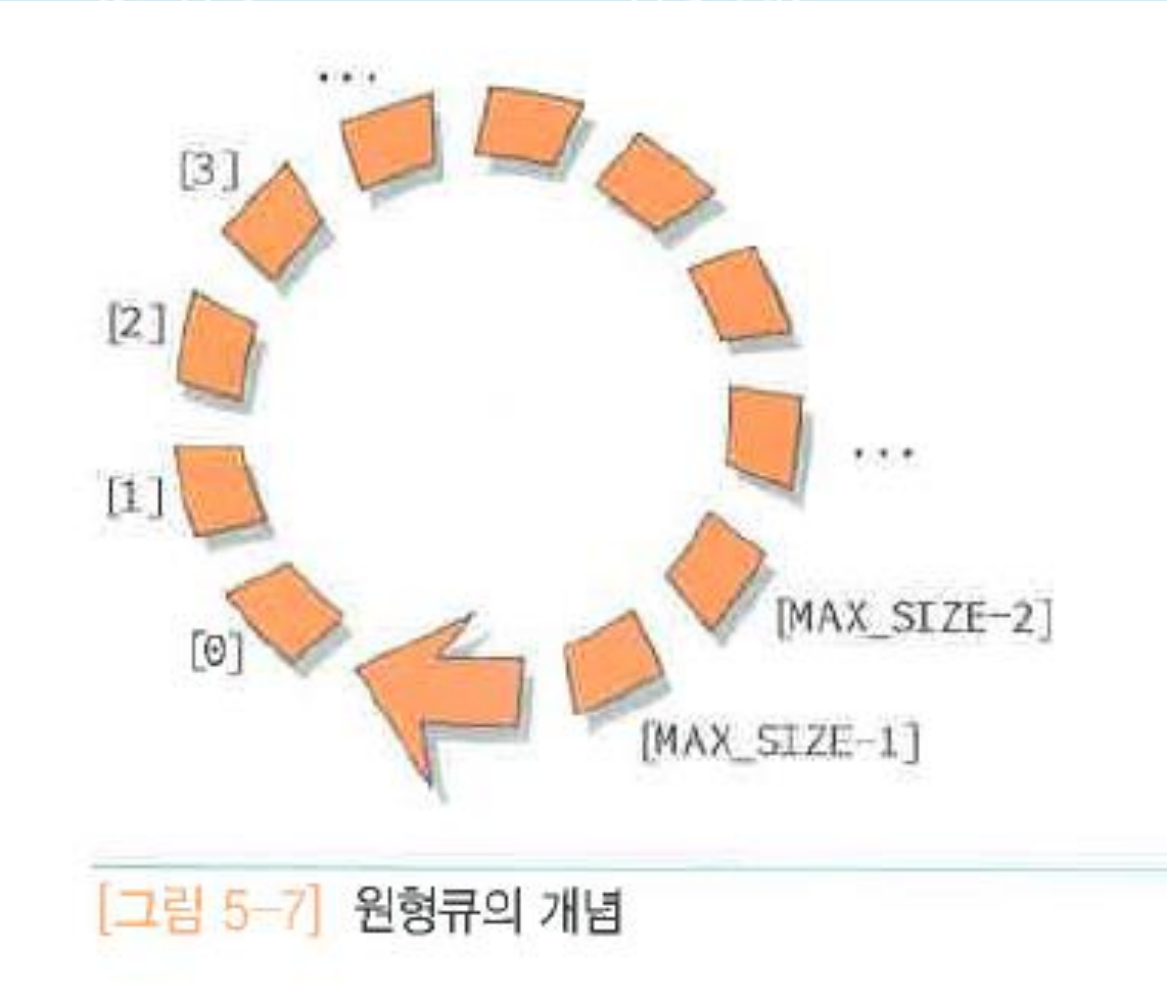
언젠가는 배열의 끝에 도달하게 되고, 배열의 앞부분을 사용할 수 없음

주기적으로 배열의 값들을 이동시키면 해결은 되지만..

원형 큐

선형 큐의 문제를 해결하기 위함
배열을 선형이 아니라 원형으로 생각하는 것

다만, 실제로 배열이 원형으로 변화되는 것은 아니며 개념 상으로 배열의 인덱스를 변화시켜주는 것임에 유의



[그림 5-7] 원형큐의 개념

선형 큐와의 차이점

원형 큐에서 변경되는 부분

front, rear

초기값은 -1이 아닌 0

front는 항상
첫 번째 요소의 하나 앞을,

rear는 마지막 요소를
가리킴

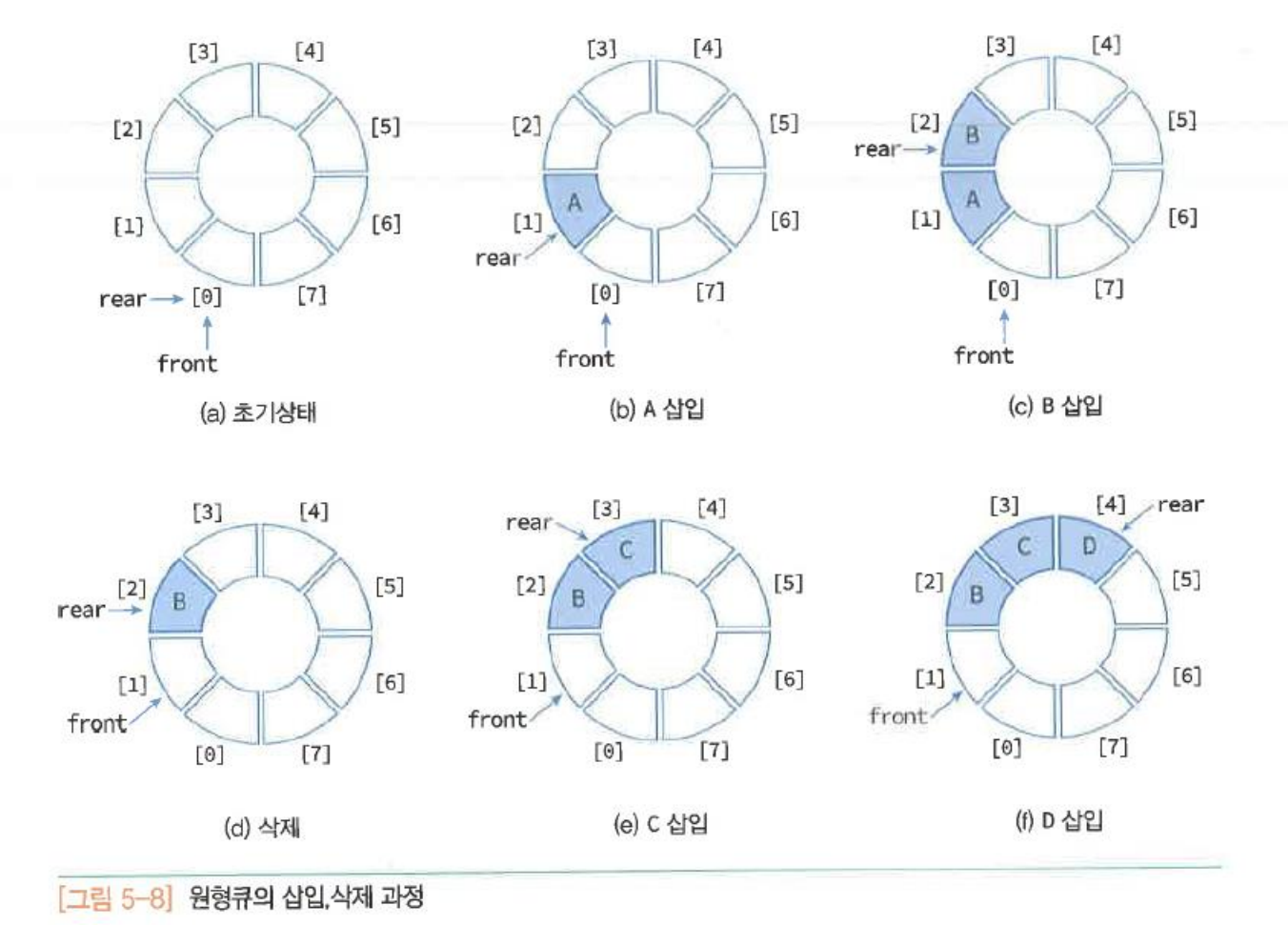
삽입 연산

rear가 먼저 증가되고,
증가된 위치에 새로운
데이터 삽입

삭제 연산

front가 먼저 증가되고,
증가된 위치의 데이터
삭제

원형 큐에서의 삽입, 삭제



[그림 5-8] 원형큐의 삽입,삭제 과정

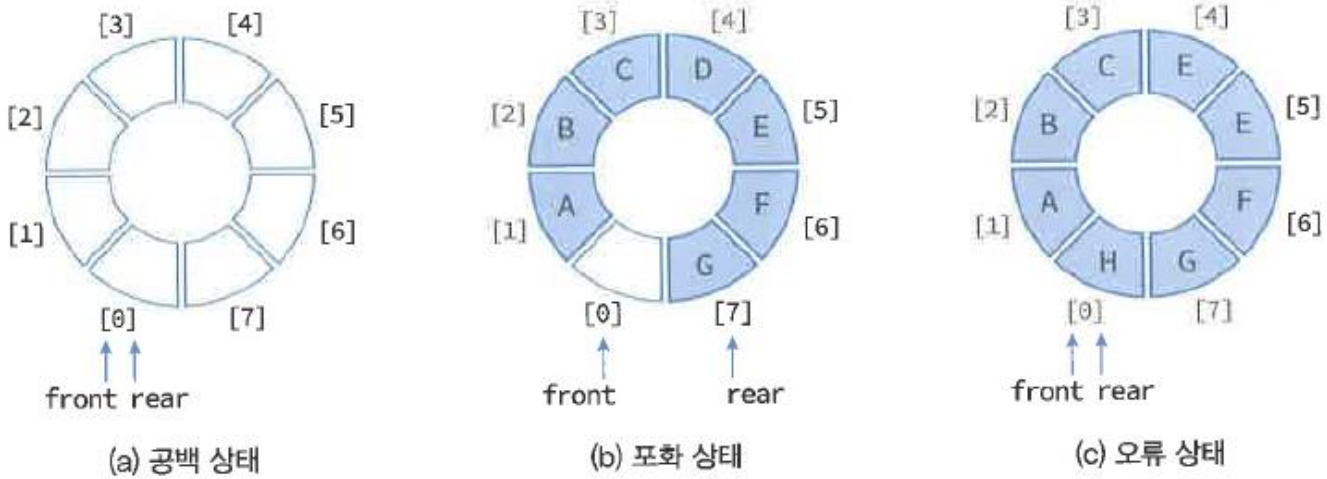
여기서 하나 유의해야할 것은,
삭제 연산이 일어났을 때
데이터가 **실제로 삭제되는 것이 아님**

front값을 변화시키는 것으로
그렇게 가정하는 것

또한 원형 큐에서 하나의 자리는
항상 비워둬야 함

한 자리를 비워야 하는 이유

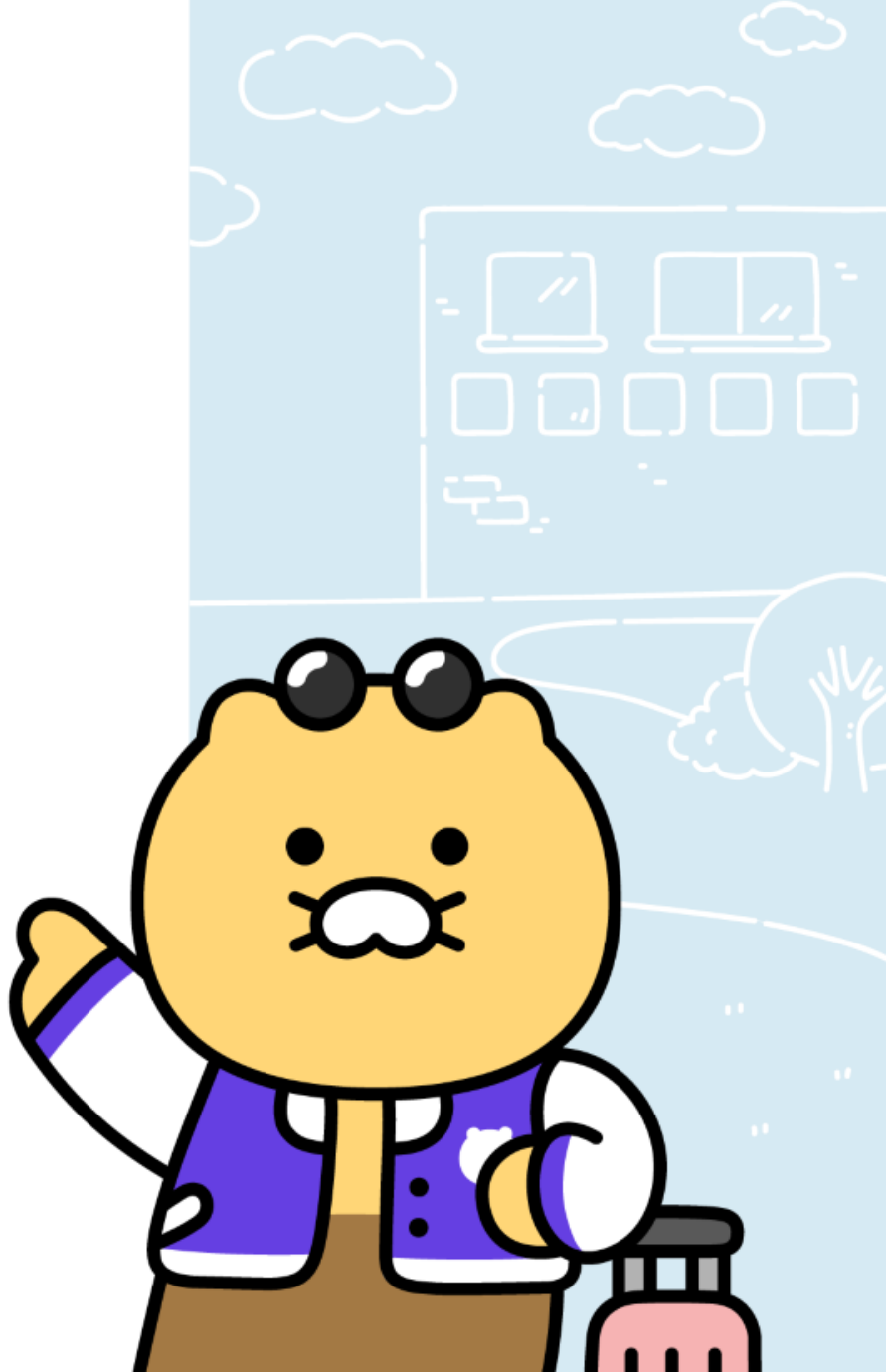
공백/포화 상태를 구분하기 위함



[그림 5-9] 원형큐의 공백상태와 오류상태

한 자리를 비워 두지 않는다면 (c)와 같은 상태일 때 공백 상태인지 포화 상태인지 구분할 수 없음

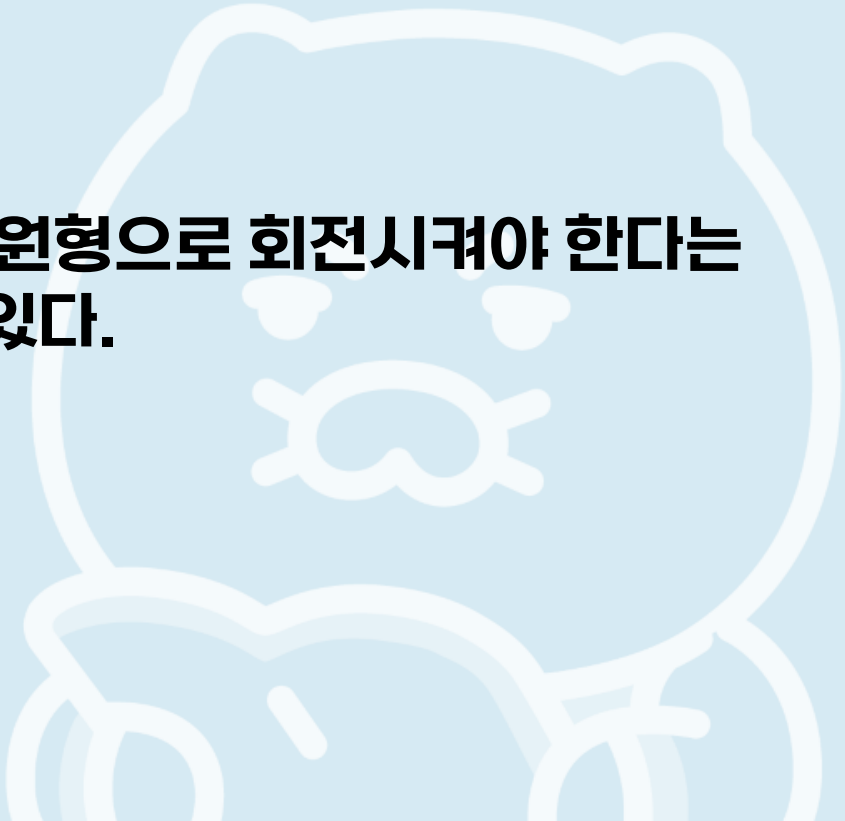
단, 요소들의 개수를 저장하고 있는 추가적인 변수를 사용할 수 있다면 비워 두지 않아도 괜찮음



원형 큐에서의 공백/포화 상태 검사

front == rear이면 공백 상태가 되고
front가 rear보다 하나 앞에 있으면(front == (rear + 1) % M, 여기서 M은 큐의 최대 크기) 포화 상태가 된다.

원형큐의 구현에 있어서 중요한 것은 front와 rear를 원형으로 회전시켜야 한다는 것인데, 이는 나머지 연산자를 이용하여 쉽게 구현할 수 있다.
front ← (front + 1) % M
rear ← (rear + 1) % M



원형 큐 코드(1)

```c
void init_queue(QueueType *q)
{
    q->front = q->rear = 0;
}

// 공백 상태 검출 함수
int is_empty(QueueType *q)
{
    return (q->front == q->rear);
}

// 포화 상태 검출 함수
int is_full(QueueType *q)
{
    return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
}
```

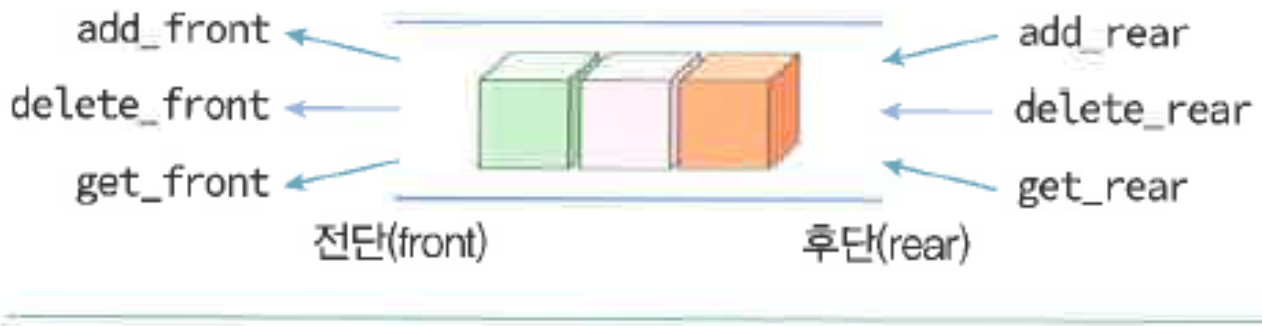
원형 큐 코드(2)

```c
// 삽입 함수
void enqueue(QueueType *q, element item)
{
    if (is_full(q))
        error("큐가 포화상태입니다");
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}
```

```c
// 삭제 함수
element dequeue(QueueType *q)
{
    if (is_empty(q))
        error("큐가 공백상태입니다");
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}
```

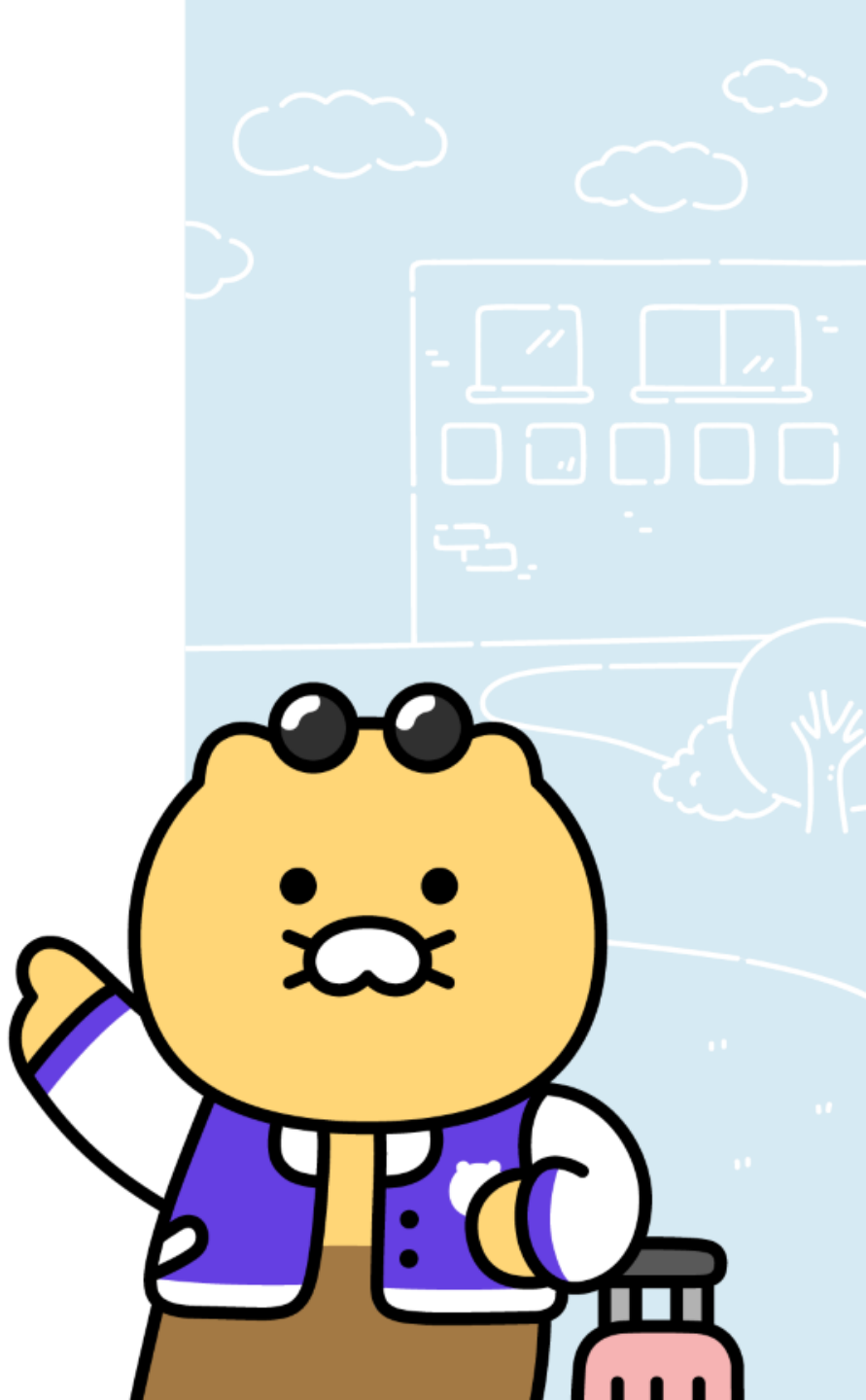
덱(Deque)

double-ended queue



[그림 5-11] 덱의 구조

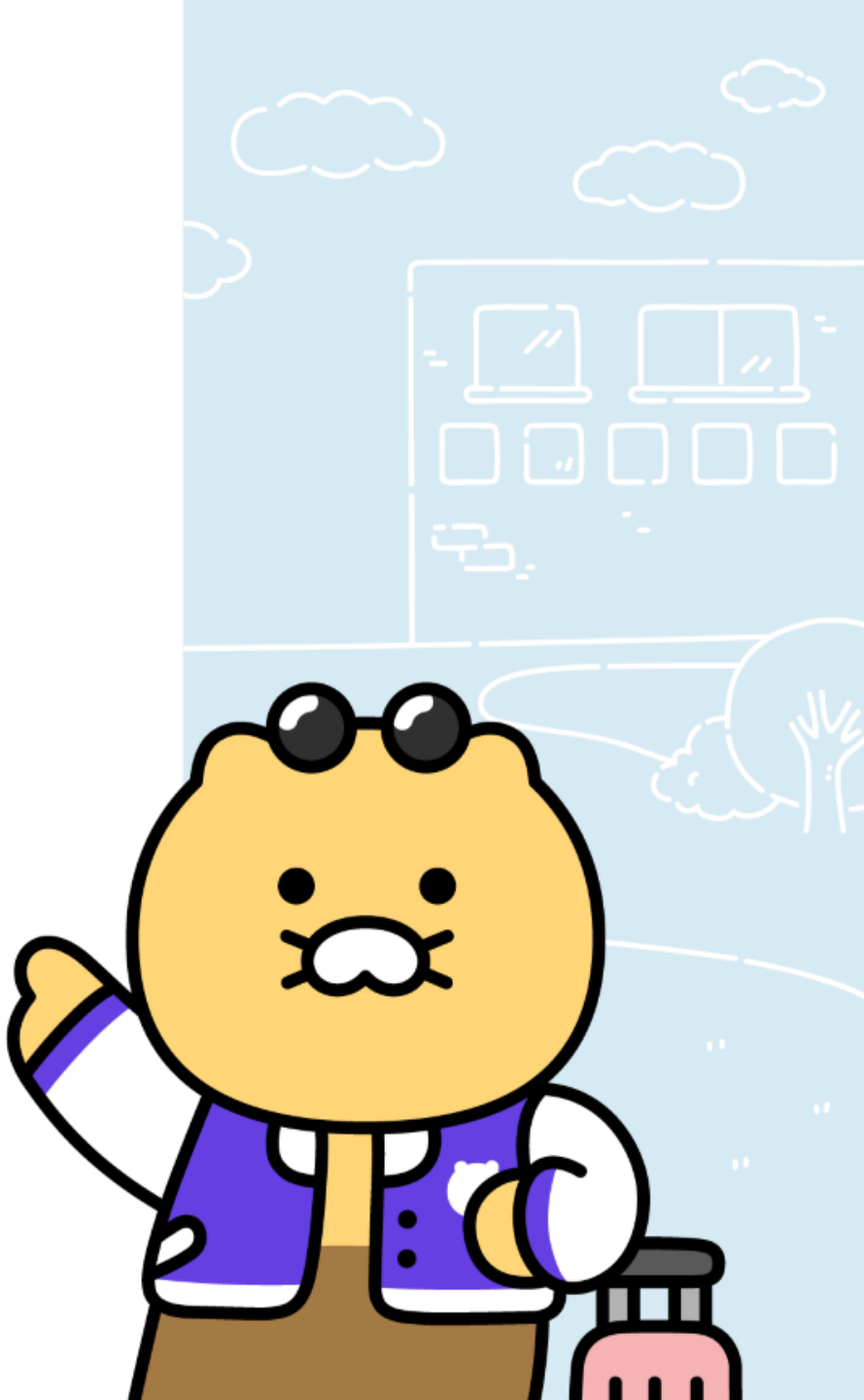
큐의 전단과 후단 모두에서 삽입과 삭제가 가능한 큐



ADT

- is_empty
- is_full
- add_front: 덱의 앞(front)에 삽입
- add_rear: 덱의 뒤(rear)에 삽입
- delete_front: 덱의 앞의 데이터 삭제 후 반환
- delete_rear: 덱의 뒤의 데이터 삭제 후 반환
- get_front: 덱의 앞의 데이터 반환
- get_rear: 덱의 뒤의 데이터 반환

front에서의 삽입과 rear에서의 삭제가 추가됨



덱 코드(1)

원형 큐 코드를 확장하면 덱을 쉽게 구현할 수 있음

```
// 삽입 함수
void add_rear(DequeType *q, element item)
{
    if (is_full(q))
        error("큐가 포화상태입니다");
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}

// 삭제 함수
element delete_front(DequeType *q)
{
    if (is_empty(q))
        error("큐가 공백상태입니다");
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}
```

add_rear와 delete_front는
각각 enqueuer, dequeue와 동일

덱 코드(2)

원형 큐 코드를 확장하면 덱을 쉽게 구현할 수 있음

```
void add_front(DequeType *q, element val)
{
    if (is_full(q))
        error("큐가 포화상태입니다");
    q->data[q->front] = val;
    q->front = (q->front - 1 + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
}

element delete_rear(DequeType *q)
{
    int prev = q->rear;
    if (is_empty(q))
        error("큐가 공백상태입니다");
    q->rear = (q->rear - 1 + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
    return q->data[prev];
}
```

기존의 원형큐 코드와는 다르게 반대 방향으로 front/rear를 옮겨주어야 함.

그렇기에 front/rear를 감소시키고, 덱의 크기만큼 더해준다. (front/rear가 0이면 음수가 되기 때문)

감사합니다 :)