

<자료구조 및 실습>

5. 큐

한국외국어대학교
컴퓨터.전자시스템공학전공
2016년 1학기
고 석 훈

학습 목표

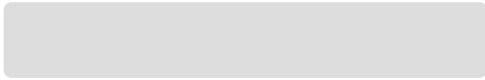
- 자료구조 큐에 대해서 이해한다.
- 큐의 특징과 연산 방법을 알아본다.
- 큐의 응용 분야를 알아본다.

큐(Queue)

- 한쪽에서 데이터를 삽입하고, 반대쪽에서 데이터를 삭제하는 선형 자료구조
 - 큐의 뒤에서는 삽입만 하고, 앞에서는 삭제만 할 수 있는 구조



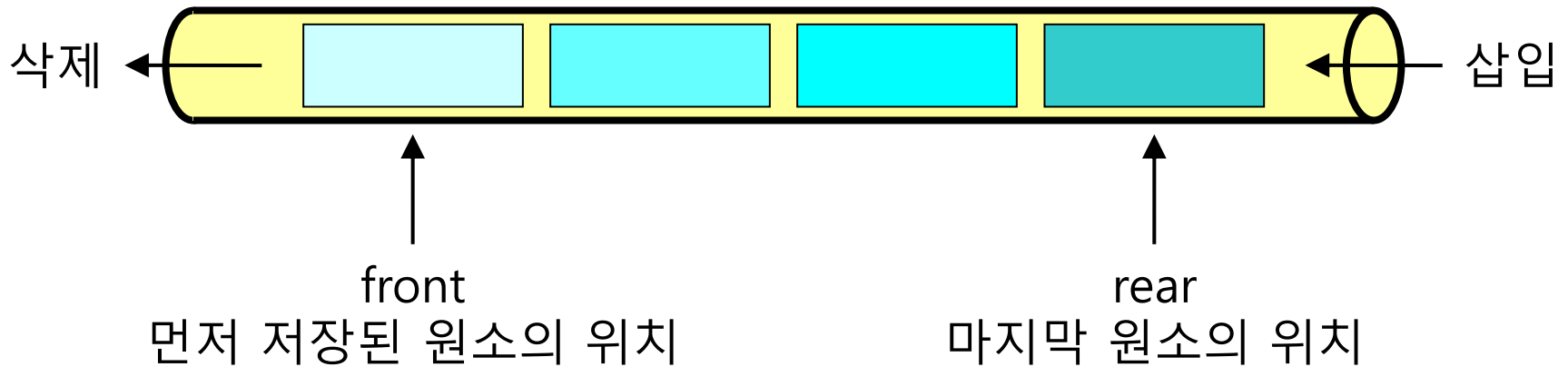
큐의 구조



- 큐에서 삽입이 이루어지는 위치 / 마지막에 저장된 원소의 위치



- 큐에서 삭제가 이루어지는 위치 / 먼저 저장된 원소의 위치



큐의 대표적인 연산

- 큐의 rear에 새로운 원소를 추가한다.
만일, 큐가 가득 찬 상태라면 오류 발생

- 큐의 front부터 원소를 삭제하고, 그 값을 알려준다.
만일, 큐에 원소가 하나도 없으면 오류 발생

스택과 큐의 연산 비교

항목 \ 자료구조	삽입 연산		삭제 연산	
	연산자	삽입 위치	연산자	삭제 위치
스택	push	top	pop	top
큐	enqueue	rear	dequeue	front

큐 ADT

이름: Queue

데이터: 0개 이상의 원소를 가진 유한 순서 리스트

연산: createQueue() ::= create an empty Q;
 // 공백 큐를 생성하는 연산

isEmpty(Q) ::= if (Q is empty) then return true
 else return false;
 // 큐가 공백인지 확인하는 연산

isFull(Q) ::= if (Q is full) then return true
 else return false;
 // 큐가 가득 찼는지 확인하는 연산

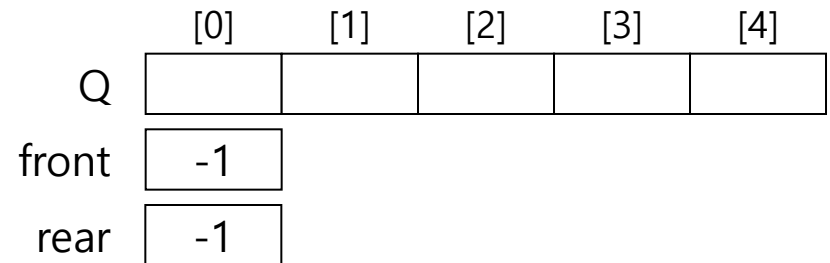
enqueue(Q, item) ::= insert item at the rear of Q;
 // 큐의 rear에 item(원소)을 삽입하는 연산

dequeue(Q) ::= if (isEmpty(Q)) then return error
 else { delete and return the front item of Q };
 // 큐의 front에 있는 원소를 삭제하고 반환하는 연산

큐 구현 1: 순차 자료구조 [1/10]

● 1차원 배열로 구현하는 선형 큐

```
createQueue()  
  Q[n];  
  front ← -1;  
  rear  ← -1;  
end createQueue()
```

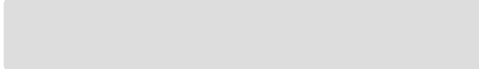


■ 1차원 배열을 이용한 큐

- ◆ 큐의 크기 = 배열의 크기
- ◆ 변수 front : 저장된 첫 번째 원소의 인덱스 -1 저장
- ◆ 변수 rear : 저장된 마지막 원소의 인덱스 저장

큐 구현 1: 순차 자료구조 [2/10]

● 큐의 공백상태와 포화상태 검사 알고리즘

■ 공백상태 : 

■ 포화상태 : 

(n : 배열의 크기, n-1 : 배열의 마지막 인덱스)

isEmpty(Q)

if (front = rear) **then return** true;

else return false;

end isEmpty()

isFull(Q)

if (rear = n - 1) **then return** true;

else return false;

end isFull()

큐 구현 1: 순차 자료구조 [3/10]

● 큐의 삽입 알고리즘

- 큐가 포화상태라면 큐 풀 오류 발생
- 큐가 포화상태가 아니라면, 인덱스 rear의 값을 하나 증가하고 해당하는 배열원소 Q[rear]에 item을 저장

```
enqueue(Q, item)  
  if (isFull(Q)) then queue_full_error;  
  else {  
    rear ← rear + 1;  
    Q[rear] ← item;  
  }  
end enqueue()
```

큐 구현 1: 순차 자료구조 [4/10]

● 큐의 인출 알고리즘

- 큐가 공백상태라면 큐 공백 오류 발생
- 큐가 공백상태가 아니라면, front의 위치를 하나 증가하여 그 자리의 원소를 삭제하여 반환

```
deQueue(Q)
```

```
  if (isEmpty(Q)) then queue_empty_error;
```

```
  else {
```

```
    front  $\leftarrow$  front + 1;
```

```
    return Q[front];
```

```
  }
```

```
end deQueue()
```

큐 구현 1: 순차 자료구조 [5/10]

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

#define Q_SIZE 5

typedef struct {
    char queue[Q_SIZE];
    int front, rear;
} Q_TYPE;

void printQ(Q_TYPE *Q)
{
    int i;

    printf("Queue: ");
    for (i = Q->front + 1; i <= Q->rear; i++)
        printf("[%d] %c ", i, Q->queue[i]);
    printf("\n");
}
```

```
Q_TYPE *createQueue( )
{
    Q_TYPE *Q;
    Q = (Q_TYPE *)malloc(sizeof(Q_TYPE));
    Q->front = Q->rear = -1;
    return Q;
}

int isEmpty(Q_TYPE *Q)
{
    if (Q->front == Q->rear)
        return 1;
    else
        return 0;
}

int isFull(Q_TYPE *Q)
{
    if (Q->rear == Q_SIZE-1)
        return 1;
    else
        return 0;
}
```

큐 구현 1: 순차 자료구조 [6/10]

```
void enqueue(Q_TYPE *Q, char item)
{
    if (isFull(Q)) {
        printf("Error: Queue is Full!\n");
        exit(1);
    }
    else {
        Q->rear++;
        Q->queue[Q->rear] = item;
    }
}

char dequeue(Q_TYPE *Q)
{
    if (isEmpty(Q)) {
        printf("Error: Queue is Empty!\n");
        exit(1);
    }
    else {
        Q->front++;
        return Q->queue[Q->front];
    }
}
```

```
void main(void)
{
    Q_TYPE *Q1 = createQueue();

    printf("createQueue \t");
    printQ(Q1);

    printf("enqueue <- 'A'\t");
    enqueue(Q1, 'A');
    printQ(Q1);

    printf("enqueue <- 'B'\t");
    enqueue(Q1, 'B');
    printQ(Q1);

    printf("enqueue <- 'C'\t");
    enqueue(Q1, 'C');
    printQ(Q1);

    printf("'%'c' <- dequeue\t",
        dequeue(Q1));
    printQ(Q1);

    printf("'%'c' <- dequeue\t",
        dequeue(Q1));
    printQ(Q1);

    printf("enqueue <- 'D'\t");
    enqueue(Q1, 'D');
    printQ(Q1);

    printf("enqueue <- 'E'\t");
    enqueue(Q1, 'E');
    printQ(Q1);

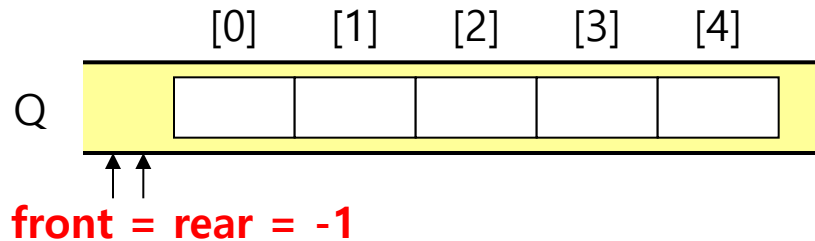
    printf("'%'c' <- dequeue\t",
        dequeue(Q1));
    printQ(Q1);

    printf("enqueue <- 'F'\t");
    enqueue(Q1, 'F');
    printQ(Q1);

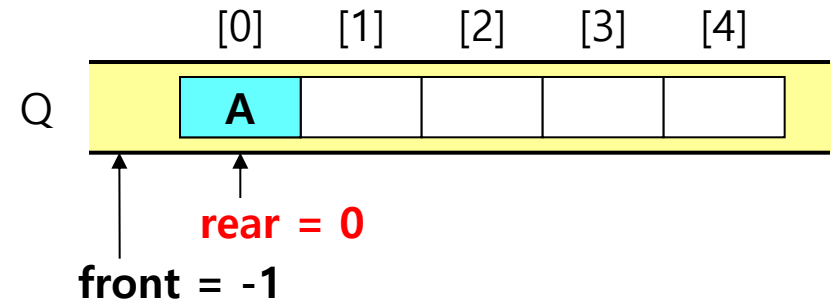
    free(Q1);
}
```

큐 구현 1: 순차 자료구조 [7/10]

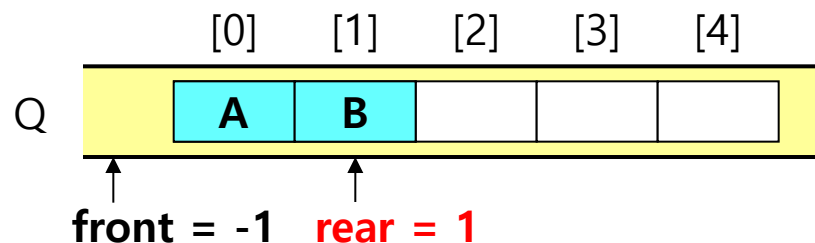
① createQueue()



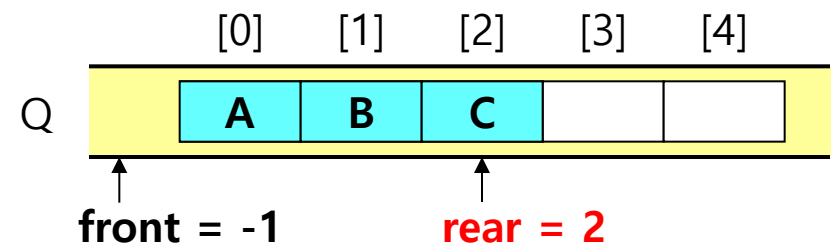
② enqueue(Q, 'A')



③ enqueue(Q, 'B')

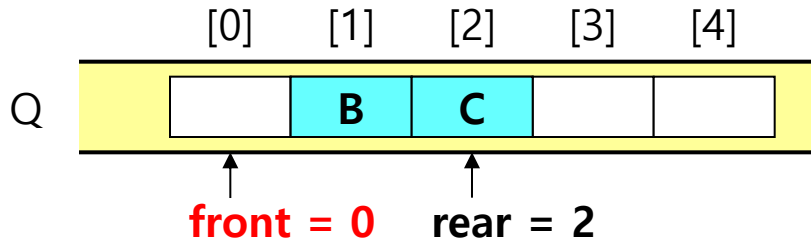


④ enqueue(Q, 'C')

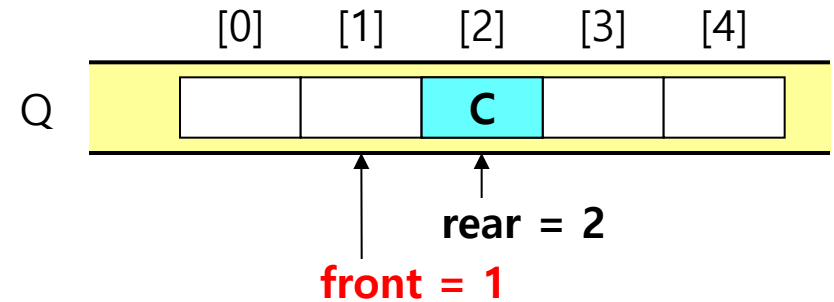


큐 구현 1: 순차 자료구조 [8/10]

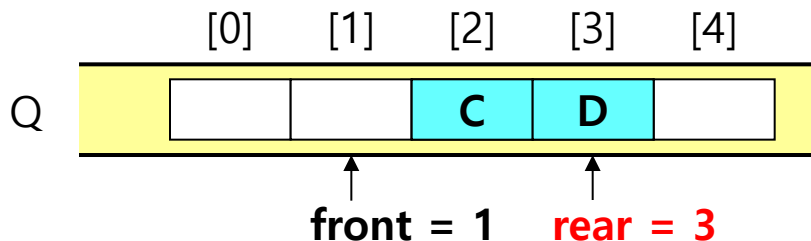
⑤ deQueue(Q)



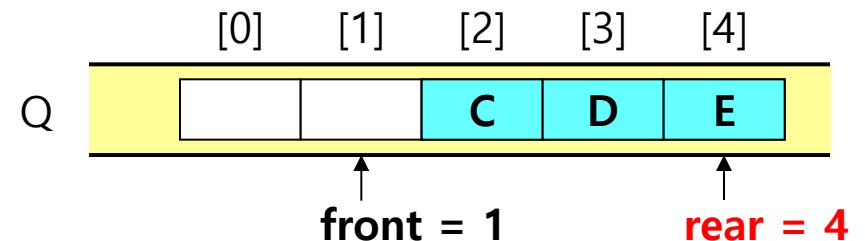
⑥ deQueue(Q)



⑦ enQueue(Q, 'D')

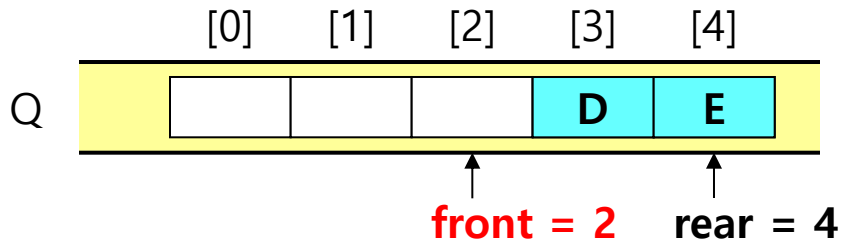


⑧ enQueue(Q, 'E')

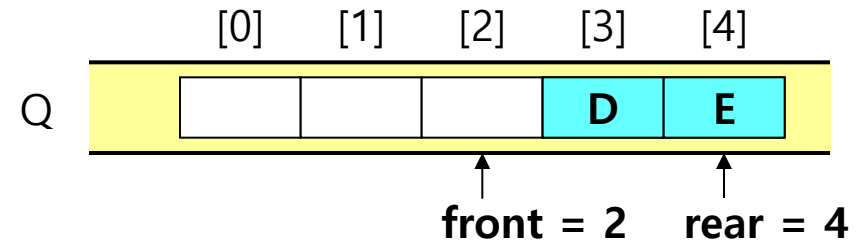


큐 구현 1: 순차 자료구조 [9/10]

⑨ deQueue(Q)



⑩ enqueue(Q, 'F')



Queue is Full!

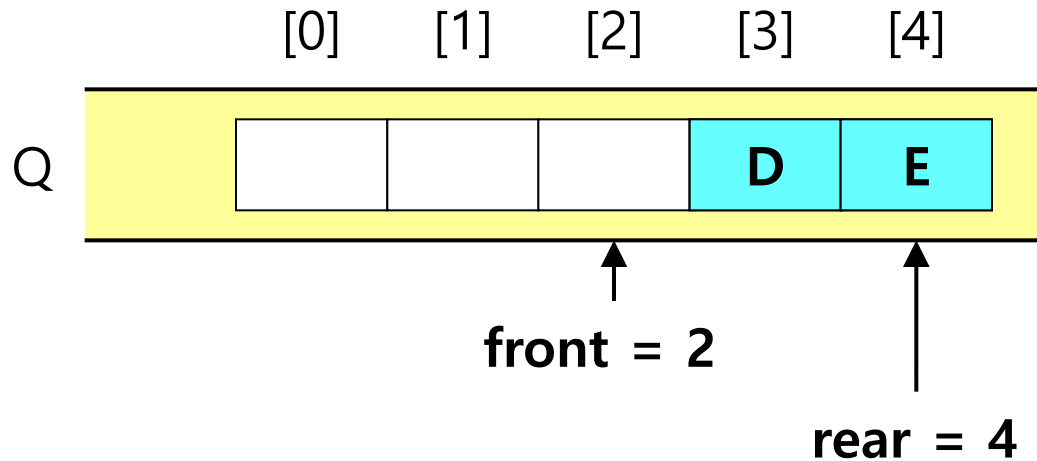
큐 구현 1: 순차 자료구조 [10/10]

```
createQueue      Queue:
enqueue <- 'A'   Queue: [0] A
enqueue <- 'B'   Queue: [0] A [1] B
enqueue <- 'C'   Queue: [0] A [1] B [2] C
'A' <- dequeue   Queue: [1] B [2] C
'B' <- dequeue   Queue: [2] C
enqueue <- 'D'   Queue: [2] C [3] D
enqueue <- 'E'   Queue: [2] C [3] D [4] E
'C' <- dequeue   Queue: [3] D [4] E
enqueue <- 'F'   Error: Queue is Full!
```


선형 큐의 문제점 [1/5]

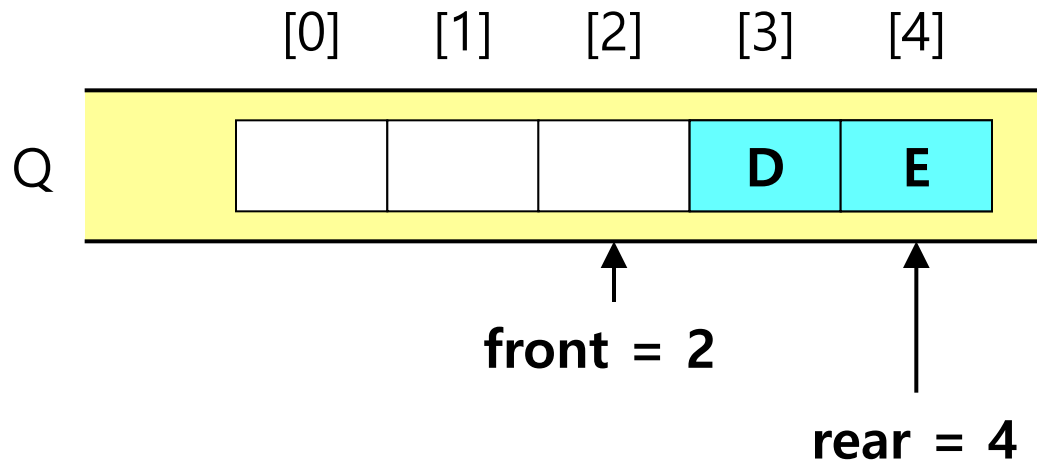
● 잘못된 포화상태

- 삽입, 삭제가 반복되면 삭제된 공간이 비어 있음에도 불구하고, 포화상태가 되어버린다.



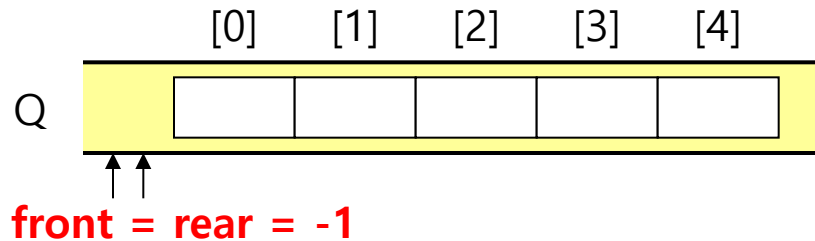
선형 큐의 문제점 [2/5]

- 잘못된 포화상태 해결방법 1: 데이터 이동
 - 삭제가 실행되면, 남아있는 자료를 배열의 앞부분으로 이동한다.
 - front는 항상 -1이 된다.

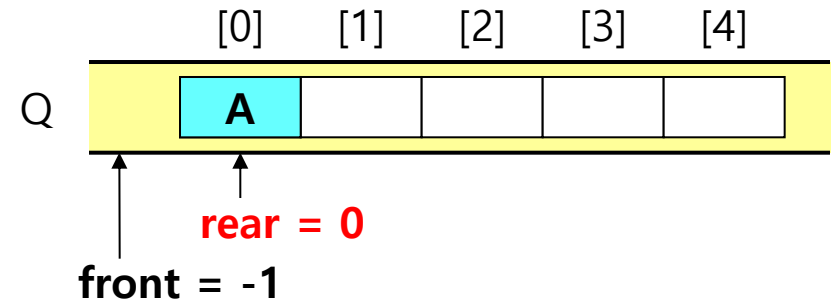


선형 큐의 문제점 [3/5]

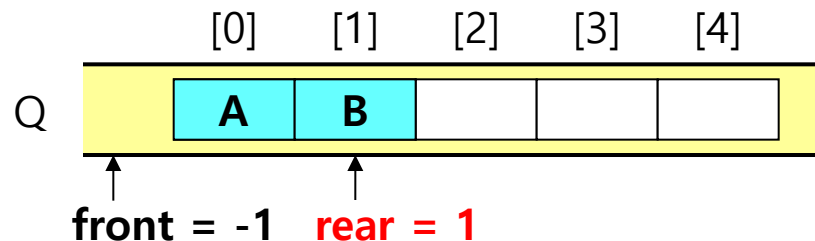
① createQueue()



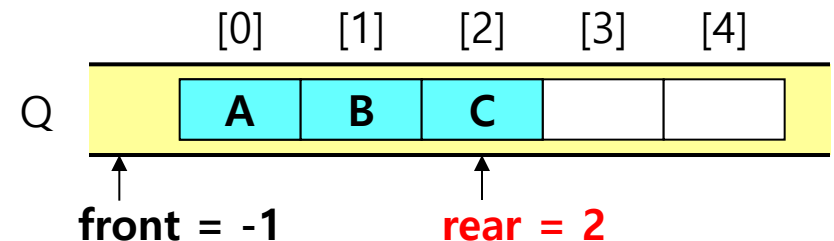
② enqueue(Q, 'A')



③ enqueue(Q, 'B')

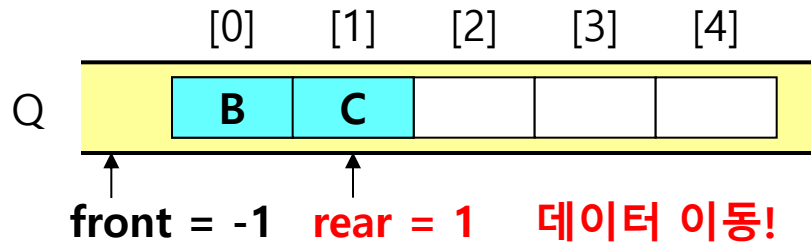


④ enqueue(Q, 'C')

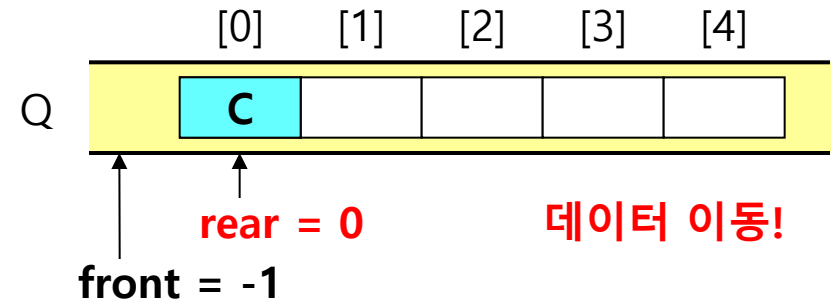


선형 큐의 문제점 [4/5]

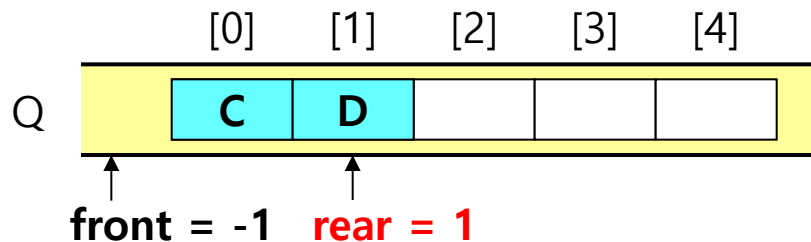
⑤ deQueue(Q)



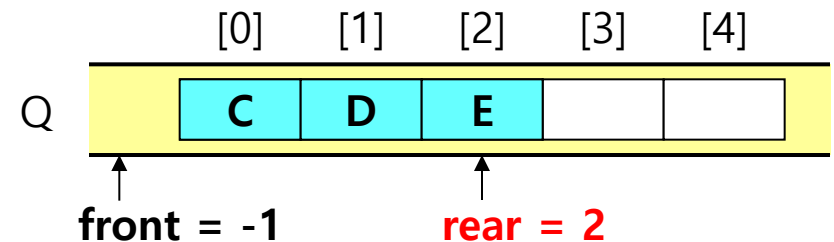
⑥ deQueue(Q)



⑦ enqueue(Q, 'D')

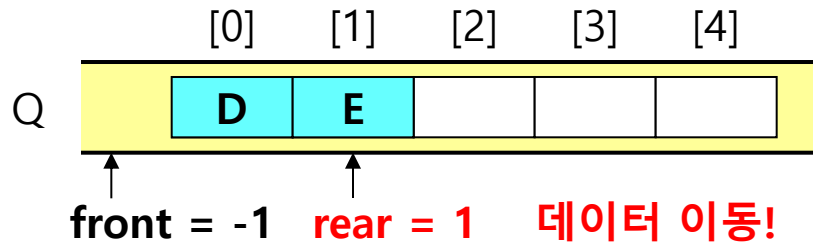


⑧ enqueue(Q, 'E')

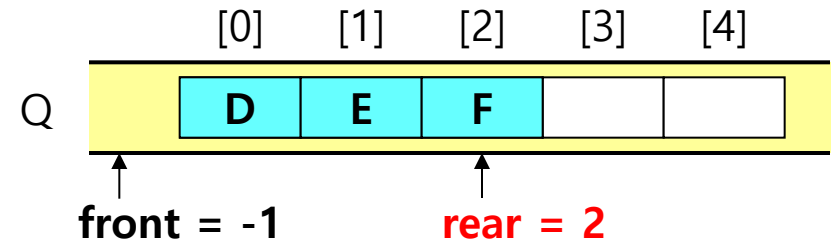


선형 큐의 문제점 [5/5]

⑨ deQueue(Q)



⑩ enqueue(Q, 'F')



데이터 이동으로
연산의 효율 저하

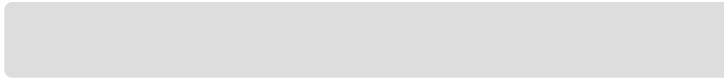
원형 큐

● 잘못된 포화상태 해결방법 2: 원형 큐

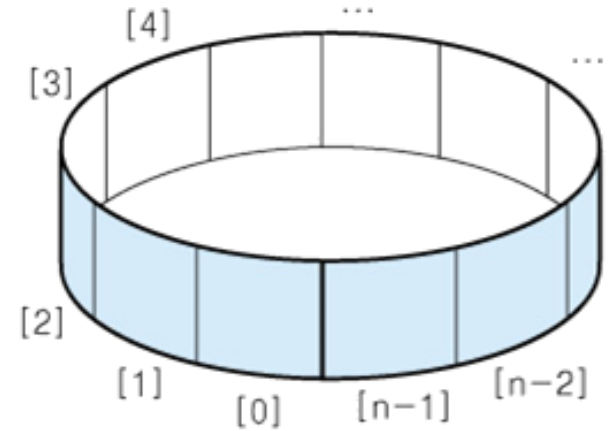
- 1차원 배열을 사용하면서 논리적으로 배열의 처음과 끝을 연결하여 사용

● 원형 큐의 구조

- 초기 공백상태:

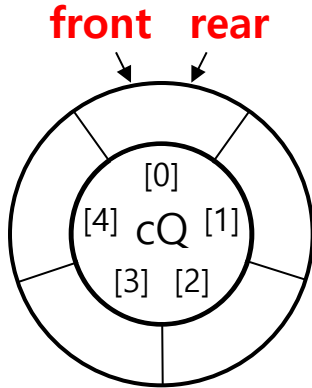


- front와 rear의 위치가 배열의 마지막 인덱스 $n-1$ 에서 논리적인 다음자리인 인덱스 0번으로 이동하기 위해서 나머지 연산자 $\%(\text{modular})$ 를 사용:

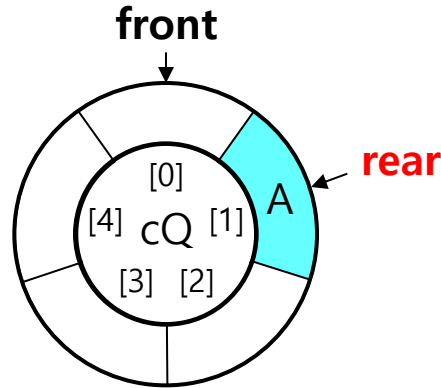


원형 큐의 연산 과정 [1/2]

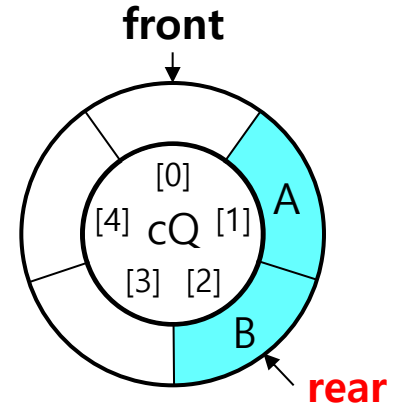
① createQueue()



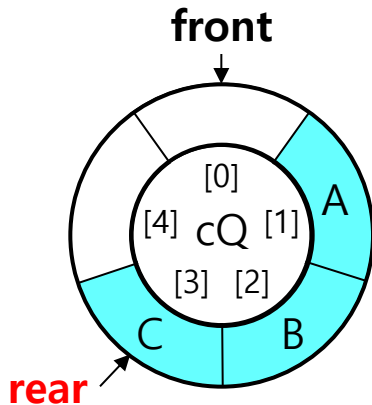
② enqueue(cQ, 'A')



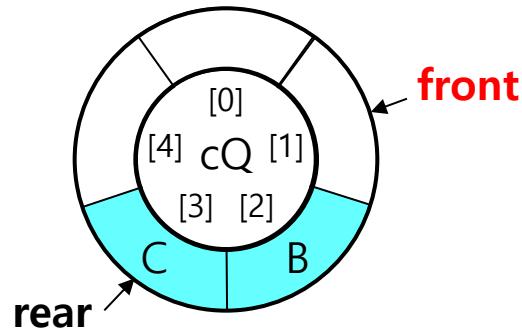
③ enqueue(cQ, 'B')



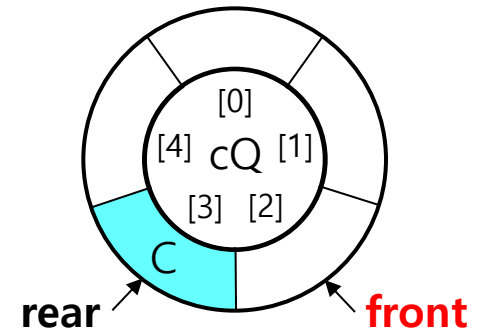
④ enqueue(cQ, 'C')



⑤ dequeue(cQ)

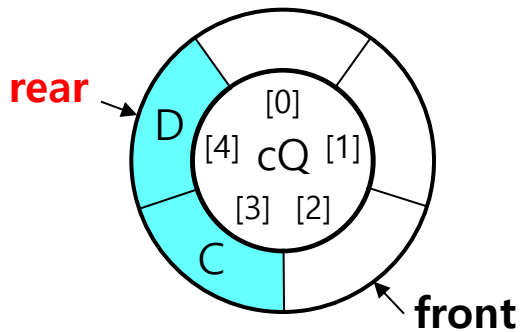


⑥ dequeue(cQ)

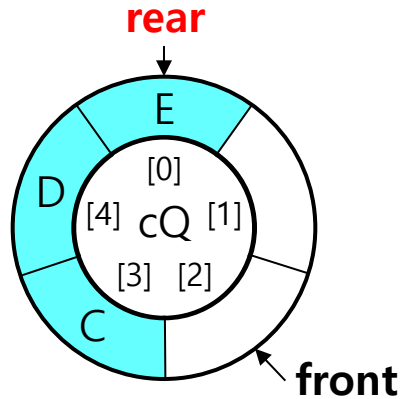


원형 큐의 연산 과정 [2/2]

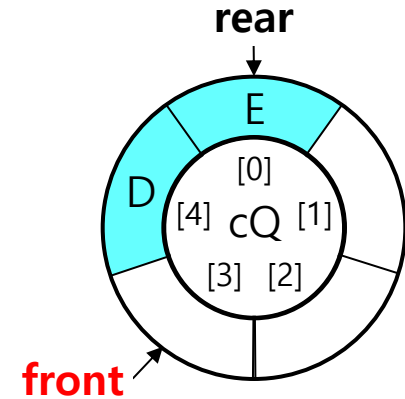
⑦ enqueue(cQ, 'D')



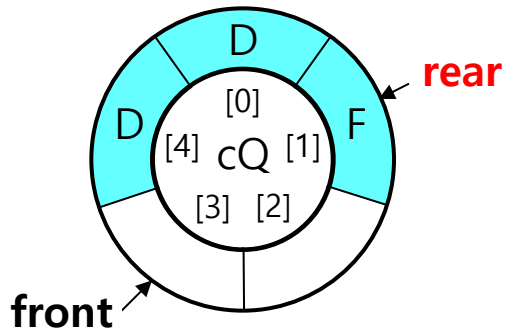
⑧ enqueue(cQ, 'E')



⑨ dequeue(cQ)



⑩ enqueue(cQ, 'F')



큐 구현 2: 원형 큐 [1/5]

- 초기 공백 원형 큐 생성 알고리즘
 - 크기가 n 인 1차원 배열 생성
 - front와 rear를 0 으로 초기화

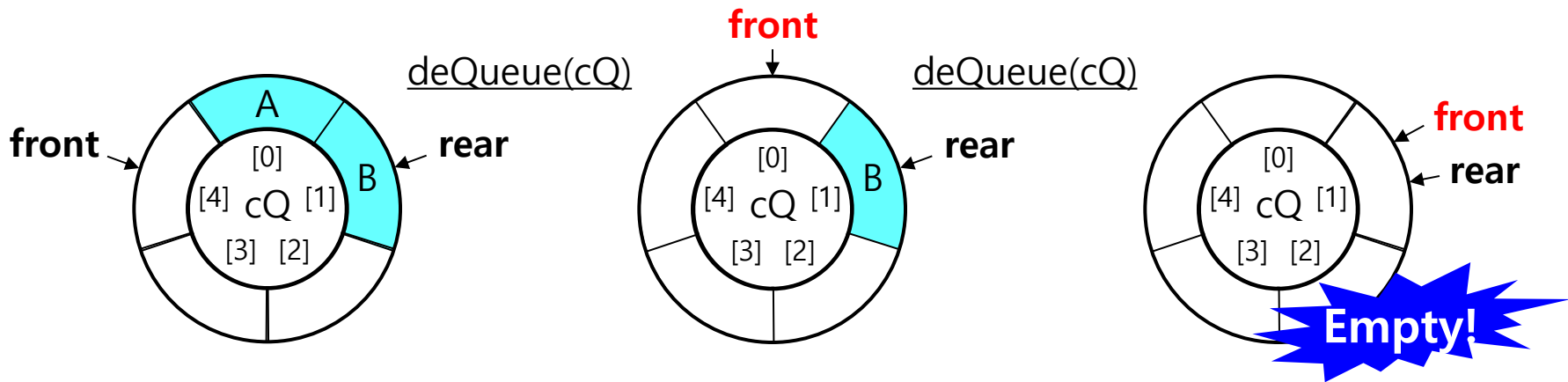
```
createQueue( )  
    cQ[n];  
    front  $\leftarrow$  0;  
    rear  $\leftarrow$  0;  
end createQueue( )
```

큐 구현 2: 원형 큐 [2/5]

● 원형 큐의 공백상태 검사 알고리즘

■ 공백상태 : $\text{front} = \text{rear}$

```
isEmpty(cQ)
  if (front = rear) then return true;
  else return false;
end isEmpty( )
```



큐 구현 2: 원형 큐 [3/5]

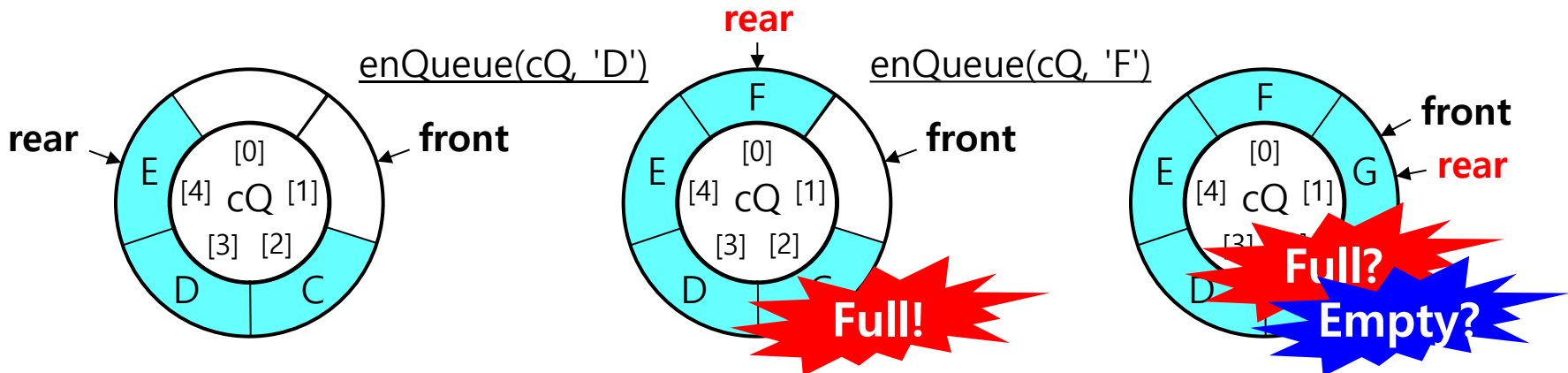
● 원형 큐의 포화상태 검사 알고리즘

■ 포화상태 :

isFull(cQ)

if $((rear + 1) \bmod n) = front$ **then return true;**
else return false;

end isFull()



공백상태와 포화상태를 쉽게 구분하기 위해 front는 항상 빈자리로 둔다.

큐 구현 2: 원형 큐 [4/5]

- 원형 큐의 삽입 알고리즘
 - 큐가 포화상태라면 큐 풀 오류 발생
 - 큐가 포화상태가 아니라면,
rear를 다음 위치로 옮기고 $Q[\text{rear}]$ 에 item을 저장

```
enqueue(cQ, item)  
  if (isFull(cQ)) then queue_full_error;  
  else {  
    rear  $\leftarrow$  (rear + 1) mod n;  
    cQ[rear]  $\leftarrow$  item;  
  }  
end enqueue()
```

큐 구현 2: 원형 큐 [5/5]

● 원형 큐의 인출 알고리즘

- 큐가 공백상태라면 큐 공백 오류 발생
- 큐가 공백상태가 아니라면,
front를 다음 위치로 옮기고 그 자리의 원소를 삭제하여 반환

```
deQueue(cQ)
```

```
  if (isEmpty(cQ)) then queue_empty_error;
```

```
  else {
```

```
    front  $\leftarrow$  (front + 1) mod n;
```

```
    return cQ[front];
```

```
  }
```

```
end deQueue()
```

큐의 응용

● 큐잉 시스템

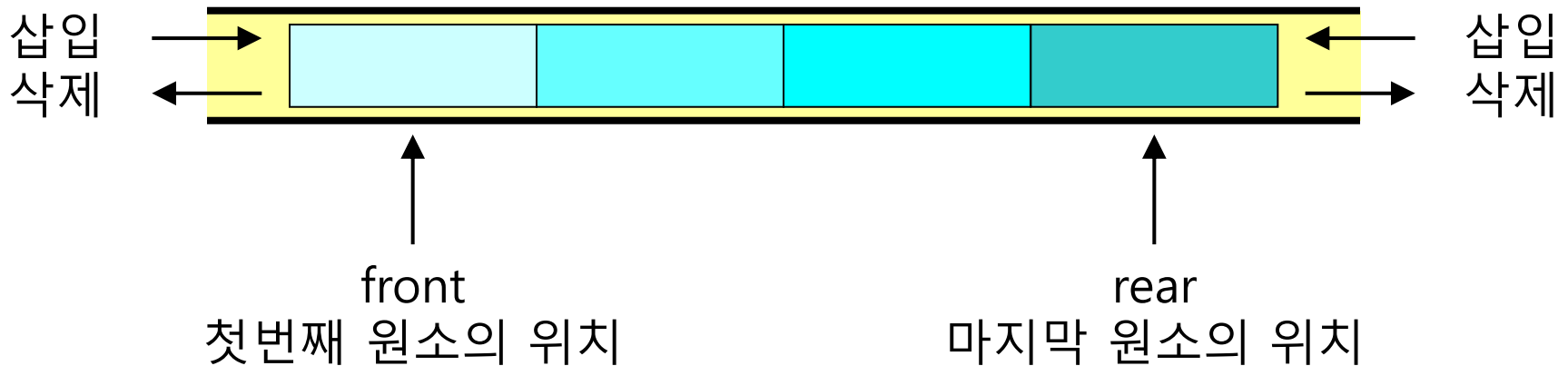
- 컴퓨터 시스템이 처리할 수 있는 것 이상으로 데이터가 전달되면, 나중에 처리하기 위해 잠시 보관할 때 큐를 사용한다.

● 큐의 응용

- 스케줄링 큐
- 프린터 버퍼 큐
- 네트워크 TX 큐

덱(Deque, Double-Ended Queue)

- 큐의 양쪽에서 모두 삽입, 삭제할 수 있는 자료구조
 - 스택 두 개를 붙여놓은 것과 같은 특징
 - 양쪽에서 삽입, 삭제를 할 수 있다.
- 덱의 구조



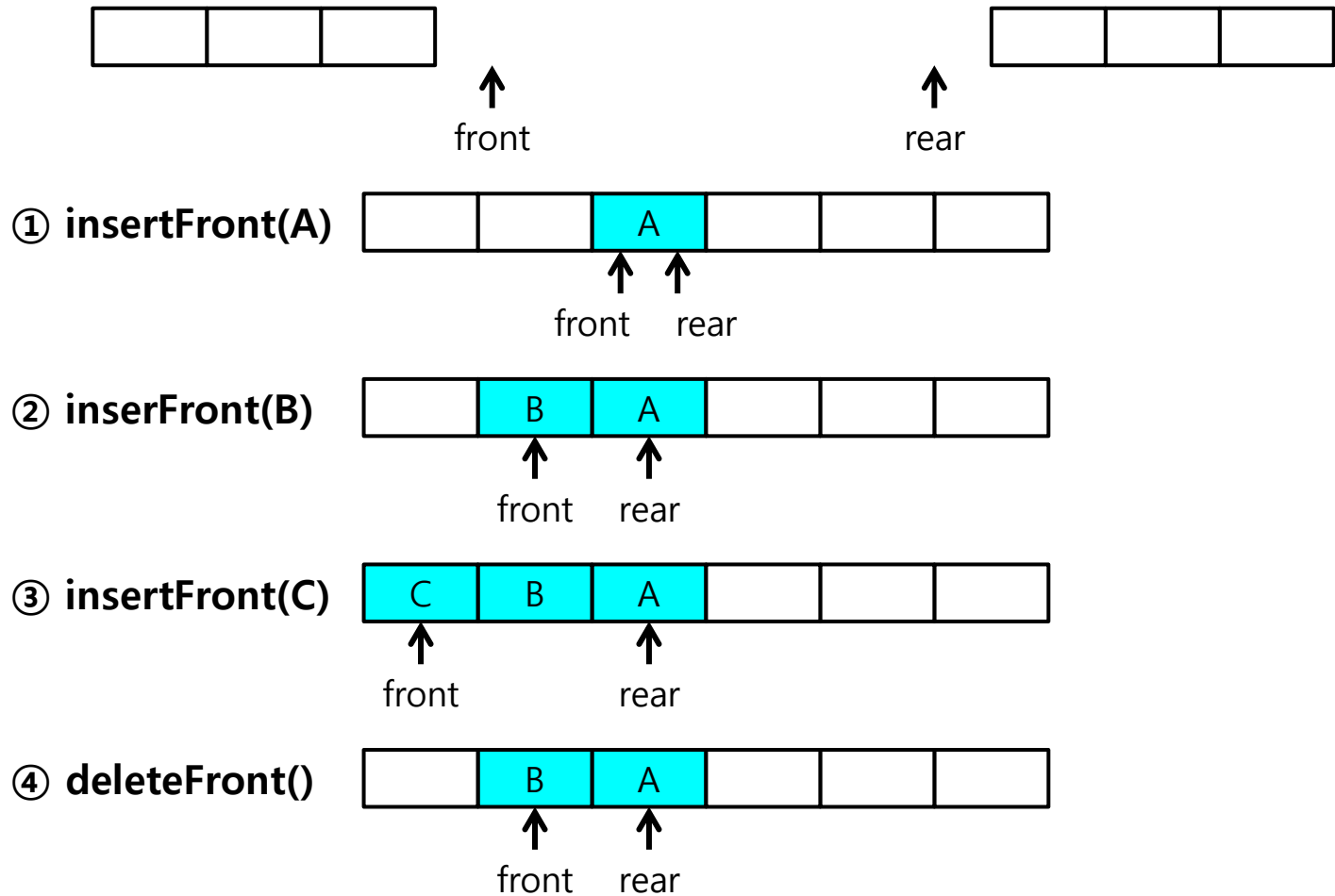
덱 ADT

이 름 : Deque

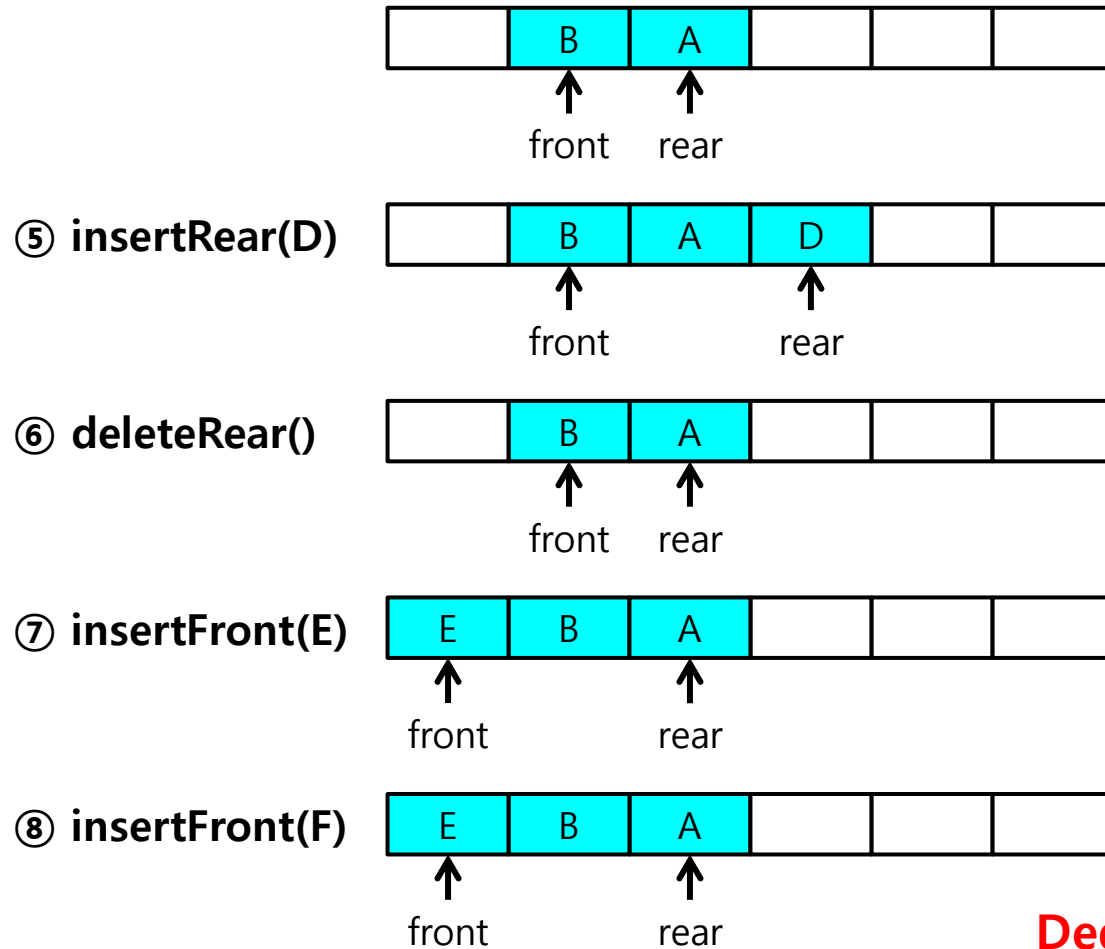
데이터 : 0개의 이상의 원소를 갖는 유한 순서 리스트

연 산 : initDeque(DQ)	::= initialize DQ;
isEmpty(DQ)	::= if (DQ is empty) then return true else return false;
isFullFront(DQ)	::= if (front-side of DQ is full) then return true else return false;
isFullRear(DQ)	::= if (rear-side of DQ is full) then return true else return false;
insertFront(DQ, item)	::= if (isFull(DQ)) then deque full error else { insert item at the front of DQ; }
insertRear(DQ, item)	::= if (isFull(DQ)) then deque full error else { insert item at the rear of DQ; }
deleteFront(DQ)	::= if (isEmpty(DQ)) then deque empty error else { delete and return the front item of DQ ;}
deleteRear(DQ)	::= if (isEmpty(DQ)) then deque empty error else { delete and return the rear item of DQ ;}

배열을 이용한 덱의 구현 [2/3]

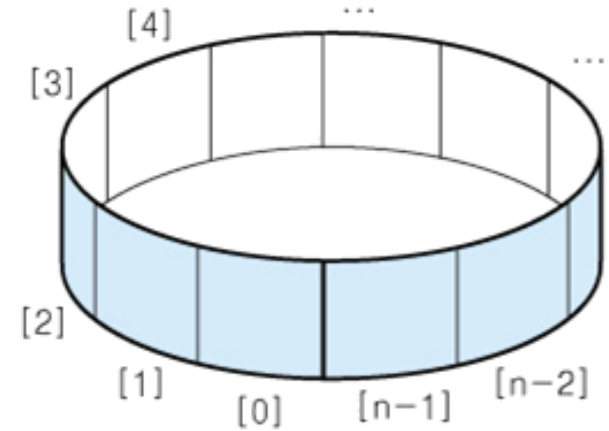


배열을 이용한 덱의 구현 [3/3]



원형 덱

- 잘못된 포화상태 문제
 - 반대쪽에 공간이 있어도 Deque-full error 발생
- 원형 덱
 - 1차원 배열을 사용하면서 논리적으로 배열의 처음과 끝을 연결하여 사용
 - 원형 큐와 같은 원리



요약

- 큐: 선입선출(FIFO, First-In-First-Out)의 선형 자료구조
 - enqueue: 큐의 rear에 새로운 원소를 추가한다.
만일, 큐가 가득 찬 상태라면 오류 발생
 - dequeue: 큐의 front부터 원소를 삭제하고, 그 값을 알려준다.
만일, 큐에 원소가 하나도 없으면 오류 발생
- 원형 큐: 논리적으로 양쪽 끝을 연결한 큐
 - 공백상태: $front = rear$
 - 포화상태: $(rear + 1) \bmod n = front$
 - 공백상태와 포화상태를 구분하기 위해 front는 비워둔다.