# 자료구조론

## 8장 큐(queue)

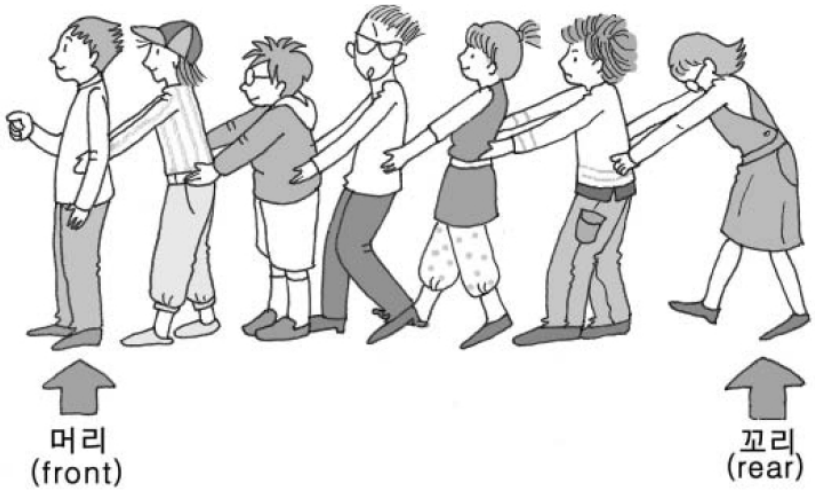
## ❑ 이 장에서 다룰 내용

- 큐
- 큐의 구현
- 큐의 응용

# ❑ 큐

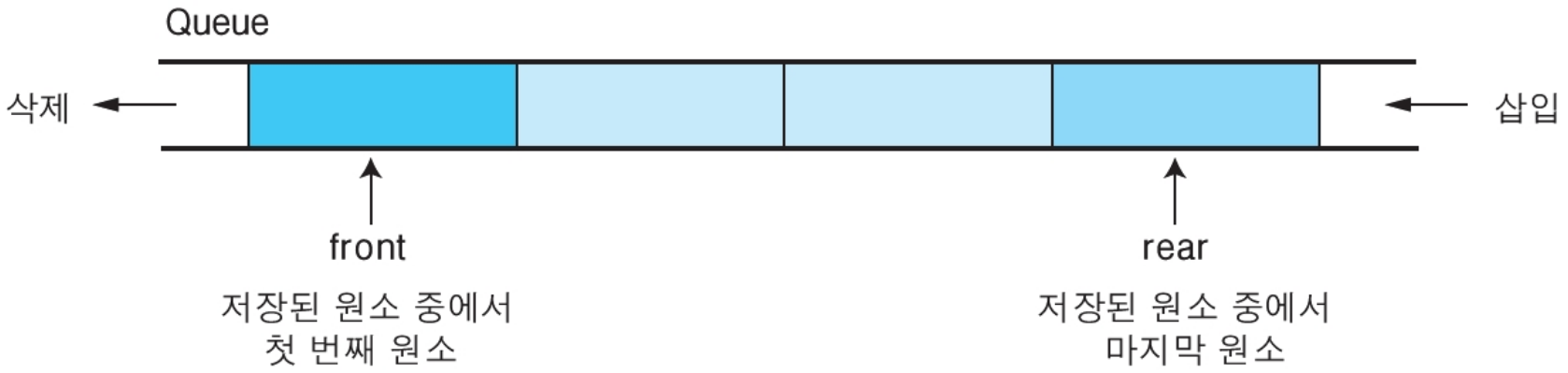
## ❖ 큐(queue)

- 삽입과 삭제의 위치가 다음과 같이 제한된 유한 순서 리스트
  - 삭제는 한쪽 끝(front 라고 부름)에서만 이루어지고,
  - 삽입은 다른 쪽 끝(rear 라고 부름)에서만 이루어진다.
- 선입선출 구조 (FIFO: First-In-First-Out)
  - 삽입한 순서대로 원소가 나열되어 가장 먼저 삽입(First-In)된 원소가 맨 앞에 있다가 가장 먼저 삭제(First-Out)된다.



# ❑ 큐

- 큐의 구조

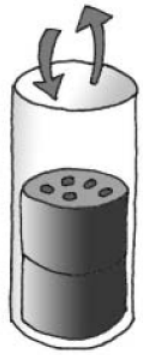



- 큐의 연산
  - 큐에서의 삽입 연산 : enQueue
  - 큐에서의 삭제 연산 : deQueue

# ❑ 큐

## ❖ 스택과 큐의 비교

- 구조



〈스택: 후입선출〉


〈큐: 선입선출〉

- 연산

| 항목 / 자료구조 | 삽입 연산 | | 삭제 연산 | |
|---|---|---|---|---|
| | 연산자 | 삽입 위치 | 연산자 | 삭제 위치 |
| 스택 | push | top | pop | top |
| 큐 | enQueue | rear | deQueue | front |

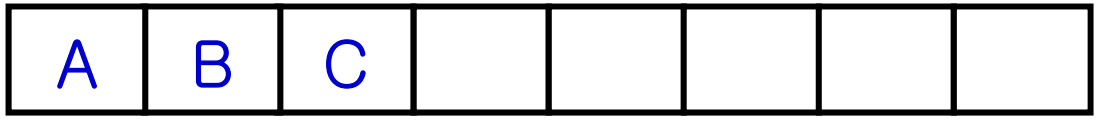
## ❑ 큐의 추상자료형

**ADT Queue**
데이터 : 0개 이상의 원소를 가진 유한 순서 리스트
연산 : Q∈Queue; item∈Element;
createQueue() ::= create an empty Q;
// 공백 큐를 생성하는 연산
isEmpty(Q) ::= **if** (Q is empty) **then** return true **else** return false;
// 큐가 공백인지 아닌지를 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 item(원소)을 삭제하여 리턴하는 연산
peek(Q) ::= **if** (isEmpty(Q)) **then** return error
**else** { return the front item of the Q };
// 큐의 front에 있는 item(원소)을 리턴하는 연산
**End Queue**

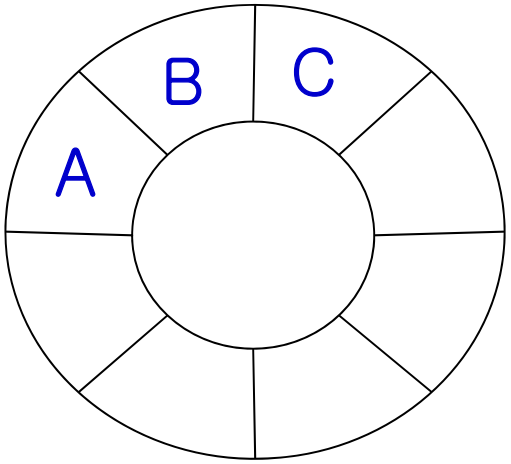
# ❑ 큐의 구현

## ❖ 큐의 구현

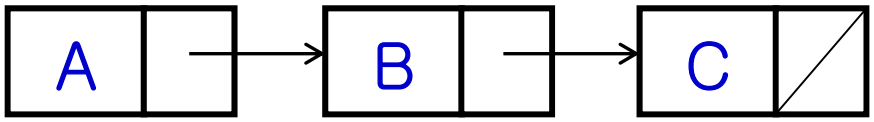
- 1차원 배열을 이용한 구현 (2가지 방법)
  - 선형 큐




  - 원형 큐



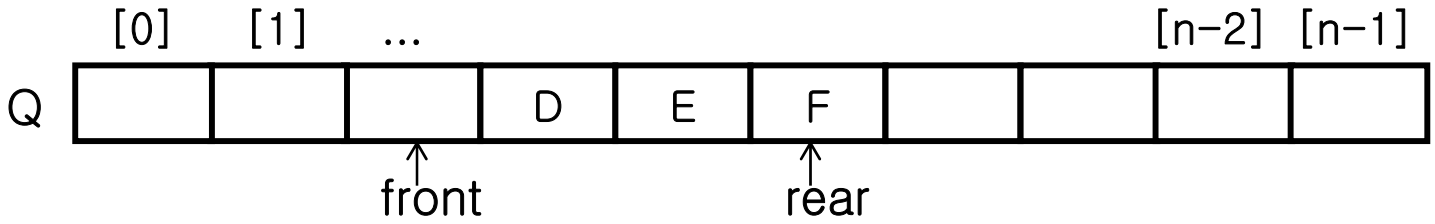

- 연결리스트를 이용한 구현



# ❑ 큐의 구현 – 배열 구현

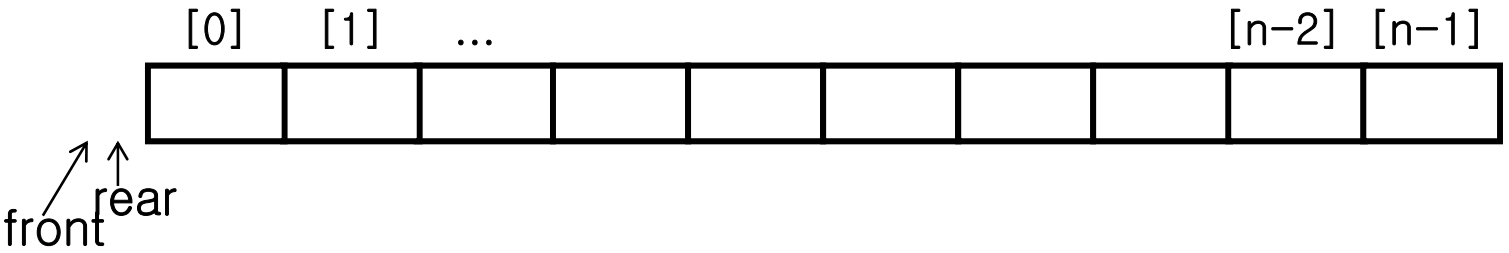
## ❖ 1차원 배열로 구현한 선형 큐

- 큐의 크기 = 배열의 크기
- front : 저장된 첫 번째 원소 바로 앞 인덱스
- rear : 저장된 마지막 원소의 인덱스

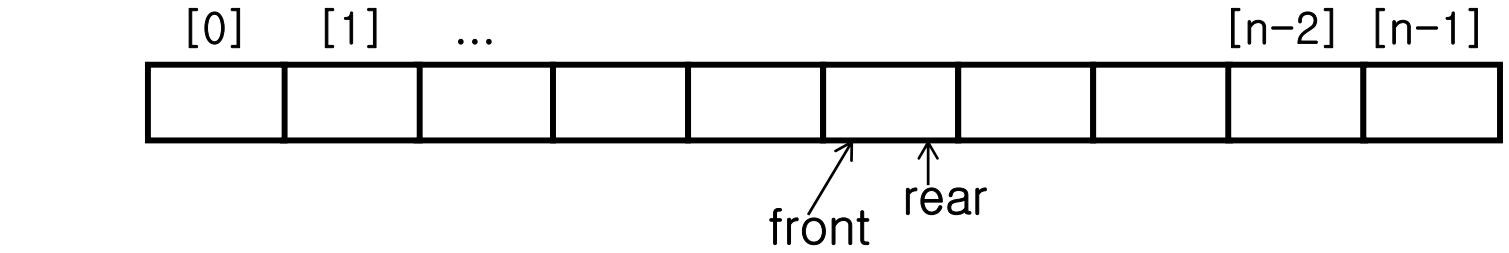


# ❑ 큐의 구현 – 배열 구현

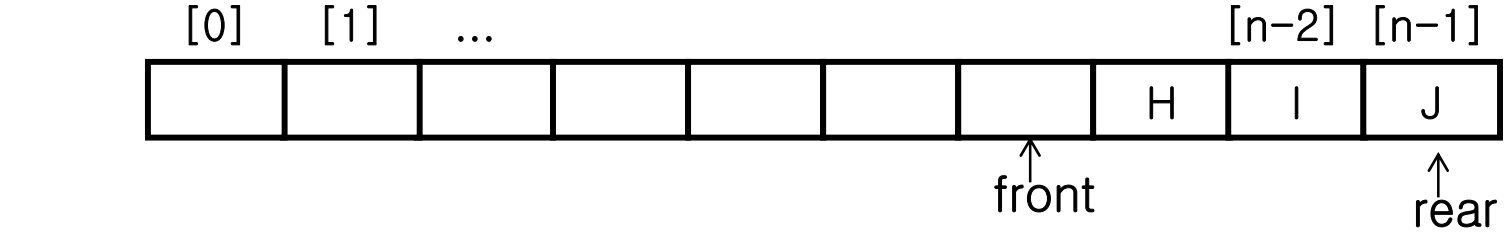
- 상태 표현 (n : 배열의 크기, n-1 : 배열의 마지막 인덱스)
  - 초기 상태 : front = rear = -1




  - 공백 상태 : front = rear



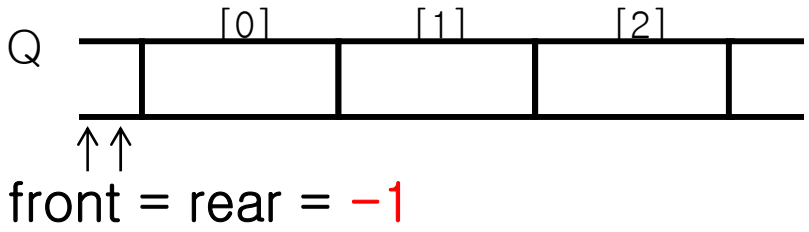

  - 포화 상태 : rear = n-1



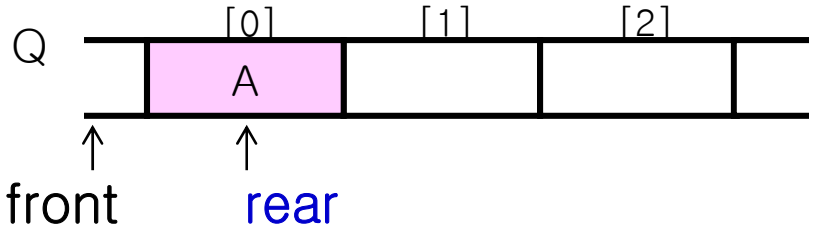
# ❑ 큐의 구현 – 배열 구현

- 큐의 연산 과정

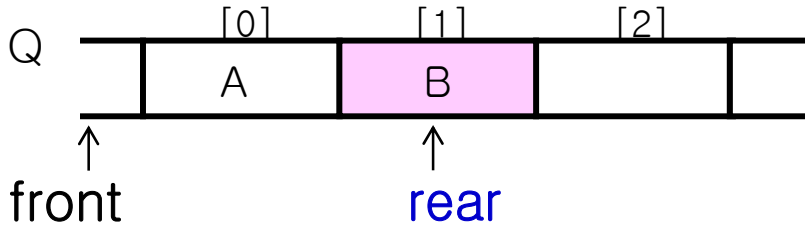
① 공백 큐 생성 : createQueue();




② 원소 A 삽입 : enQueue(Q, A);

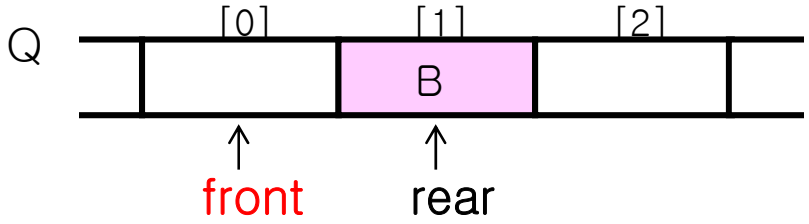



③ 원소 B 삽입 : enQueue(Q, B);

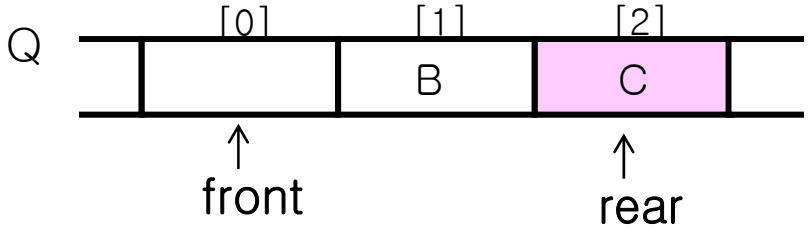


# ❑ 큐의 구현 – 배열 구현

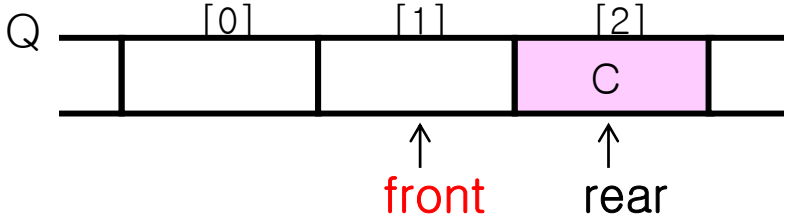
④ 원소 삭제 : deQueue(Q);



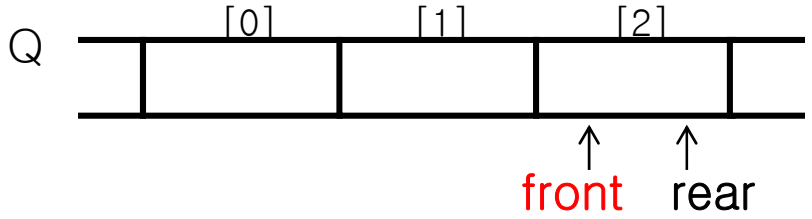

⑤ 원소 C 삽입 : enQueue(Q, C);




⑥ 원소 삭제 : deQueue(Q);




⑦ 원소 삭제 : deQueue(Q);



## ❑ 큐의 구현 – 배열 구현

- 큐 생성 알고리즘
  - 크기가 n인 1차원 배열 생성
  - front와 rear를 -1로 초기화

```
createQueue()  // 공백 큐 생성
    Q[n];
    front ← -1;
    rear ← -1;
end createQueue()
```

## ❑ 큐의 구현 – 배열 구현

- 공백상태/포화상태 검사 알고리즘
  - 공백 상태 : front = rear
  - 포화 상태 : rear = n-1
    (n: 배열의 크기, n-1: 배열의 마지막 인덱스)

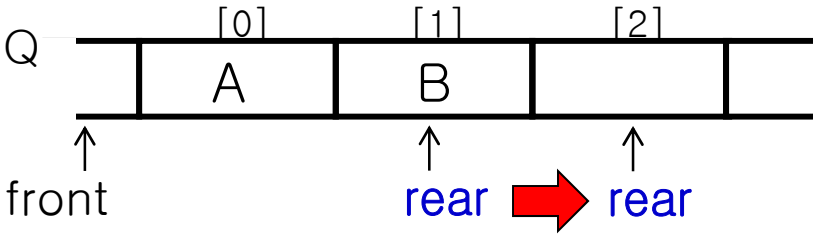
```
isEmpty(Q) // 큐가 공백인지 검사
    if(front=rear) then return true;
    else return false;
end isEmpty()

isFull(Q) // 큐가 포화상태인지 검사
    if(rear=n-1) then return true;
    else return false;
end isFull()
```

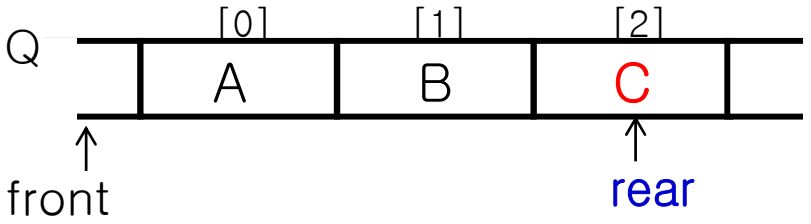
# ❑ 큐의 구현 – 배열 구현

- 큐의 삽입 알고리즘

```
enQueue(Q, item)   // 큐에 item을 삽입
    if(isFull(Q)) then Queue_Full();
    else {
        rear ← rear+1;      // ①
        Q[rear] ← item;     // ②
    }
end enQueue()
```

- 마지막 원소의 뒤에 삽입해야 하므로
  ① 마지막 원소의 인덱스를 저장한 rear의 값을 하나 증가시켜 삽입할 자리 준비
  ② rear 위치에 item을 저장



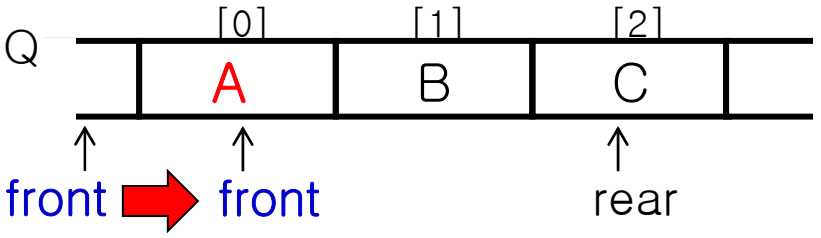



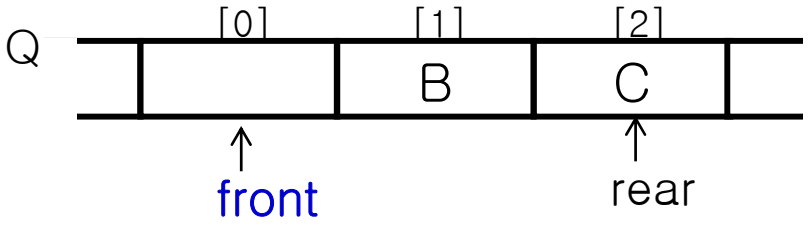
# ❑ 큐의 구현 – 배열 구현

- 큐의 삭제 알고리즘

```
deQueue(Q)  // 큐로부터 원소를 삭제하여 리턴
    if(isEmpty(Q)) then Queue_Empty();
    else {
        front ← front+1;  // ①
        return Q[front];  // ②
    }
end deQueue()
```

  - 가장 앞에 있는 원소를 삭제해야 하므로
    ① front의 위치를 한자리 뒤로 이동하여 큐에 남아있는 첫 번째 원소의 위치로 이동하여 <u>삭제할 자리 준비</u>
    ② 그 자리의 원소를 삭제하여 반환



## ❑ 큐의 구현 – 배열 구현

❖ 배열로 구현한 character 선형 큐 프로그램

```
public class Ex8_1 {
    public static void main(String[] args) {
        ArrayQueue q = new ArrayQueue(5);
        q.enQueue('A');
        q.enQueue('B');
        q.enQueue('C');
        System.out.println(q);

        while(!q.isEmpty())
            System.out.println("deleted item : " + q.deQueue());
        System.out.println(q);

        q.enQueue('D');
        q.enQueue('E');
        q.enQueue('F');
        System.out.println(q);
    }
}
```

## ❑ 큐의 구현 – 배열 구현

```
public class ArrayQueue {
    private int front;
    private int rear;
    private int queueSize;
    private char[] itemArray;

    public ArrayQueue(int queueSize) {
        front = -1;
        rear = -1;
        this.queueSize = queueSize;
        itemArray = new char[queueSize];
    }

    public boolean isEmpty() {
        return (front == rear);
    }

    public boolean isFull() {
        return (rear == queueSize-1);
    }
```

## ❑ 큐의 구현 – 배열 구현

```
public void enQueue(char item) {
    if(isFull())
        System.out.println("Inserting fail! Array Queue is full!!");
    else
        itemArray[++rear] = item;
}

public char deQueue() {
    if(isEmpty()) {
        System.out.println("Deleting fail! Array Queue is empty!!");
        예외 발생;
    }
    else{
        return itemArray[++front];
    }
}
// 다음 슬라이드에 계속
```

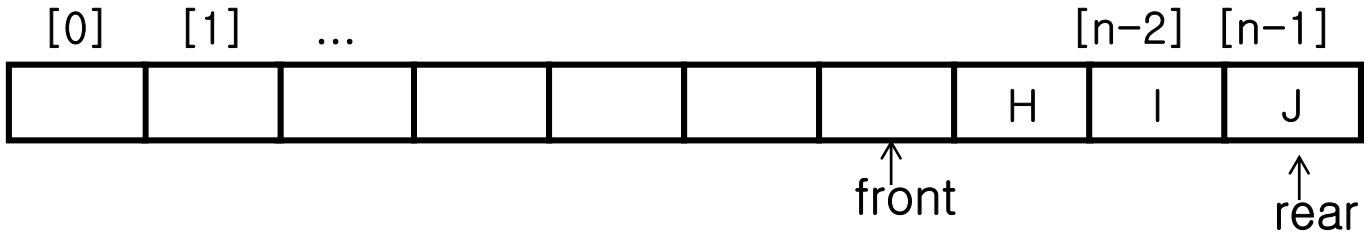
## ❑ 큐의 구현 – 배열 구현

```
    public char peek() {
        if(isEmpty()) {
            System.out.println("Peeking fail! Array Queue is empty!!");
            예외 발생;
        }
        else return itemArray[front+1];
    }

    public String toString() {
        String result = "Array Queue>> ";
        for(int i=front+1; i<=rear; i++)
            result += itemArray[i] + " ";
        return result;
    }
}
```

# ❑ 큐의 구현 – 원형 큐

❖ 선형 큐의 포화 상태 문제

- 삽입과 삭제를 반복하다 보면, 앞부분에 빈자리가 있지만 포화상태(rear=n-1)로 인식하여 더 이상의 삽입을 수행할 수 없는 경우가 생긴다.




❖ 선형 큐에서 위와 같은 잘못된 포화상태 인식 문제를 해결하는 방법 2가지

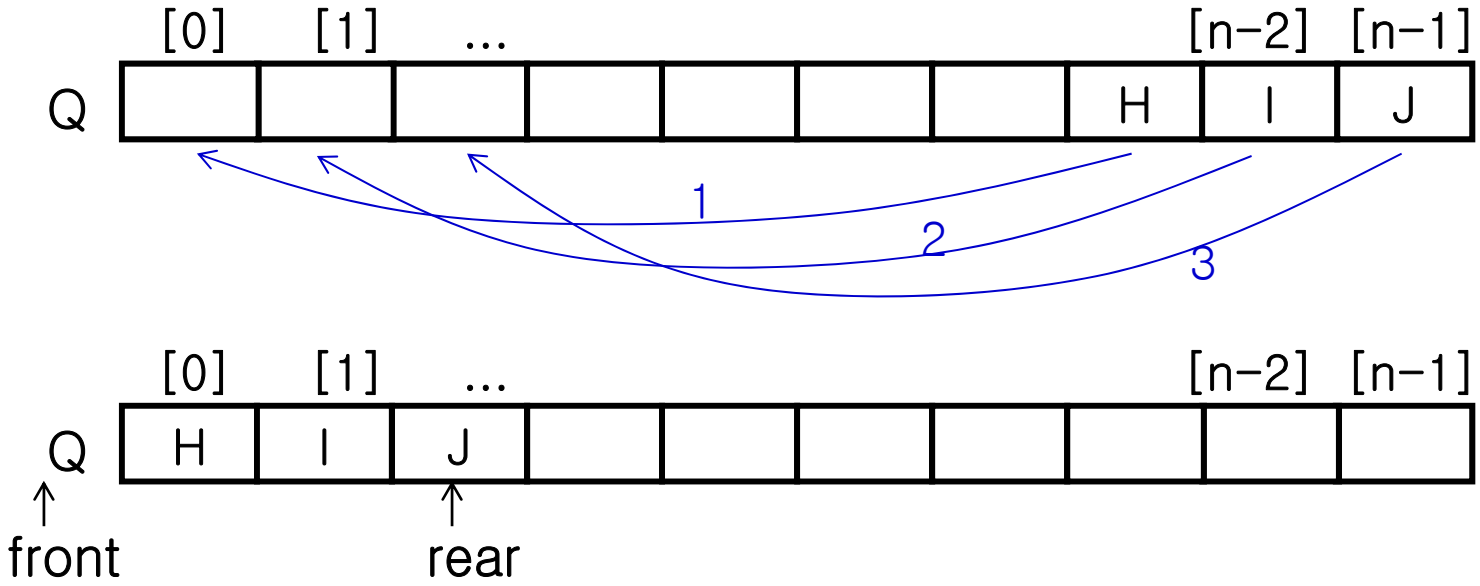
(1) 자료 이동

(2) 원형 큐로 구현

# ❑ 큐의 구현 – 원형 큐

❖ 해결 방법 (1)

- 저장된 원소들을 비어있는 배열 앞부분으로 이동시킴

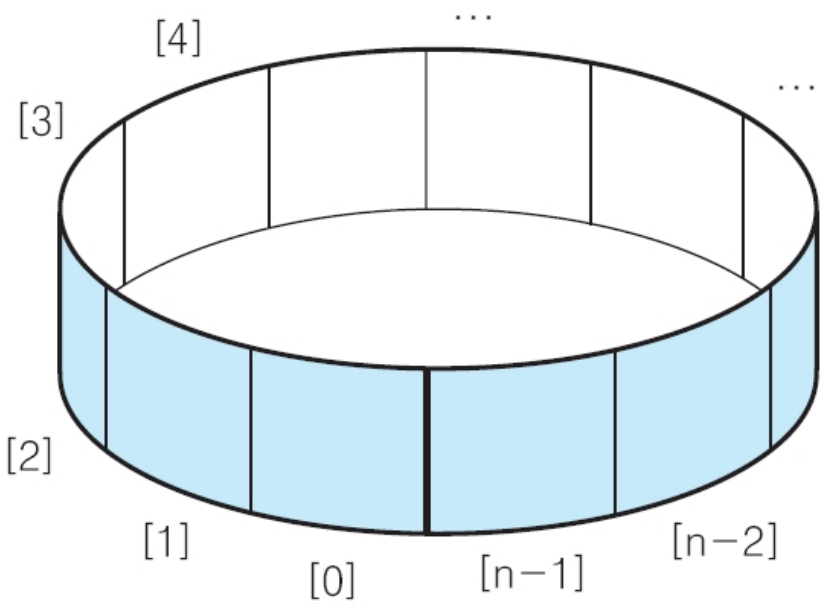



➔ 순차 자료구조에서의 자료 이동 작업으로 효율 떨어짐

# ❑ 큐의 구현 – 원형 큐

## ❖ 해결 방법 (2)

- 원형 큐(circular queue)
  - 1차원 배열을 사용하되 논리적으로 배열의 처음과 끝이 연결되어 있다고 가정하고 사용
  - 즉, Q[n-1] 다음 원소가 Q[0]
- 원형 큐의 논리적 구조



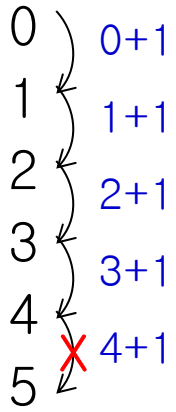
# ❑ 큐의 구현 – 원형 큐

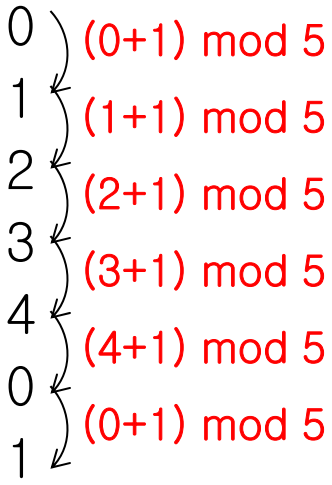
## ❖ 원형 큐의 구조

- 초기 공백 상태 : front = rear = 0
- 큐 연산시 front와 rear의 위치 이동 : 배열의 마지막 인덱스 n-1 다음 인덱스 0번으로 이동하기 위해서 나머지연산자 mod를 사용

| 선형 큐 | 원형 큐 |
|---|---|
| front = front + 1 | front = (front + 1) mod n |
| rear = rear + 1 | rear = (rear + 1) mod n |

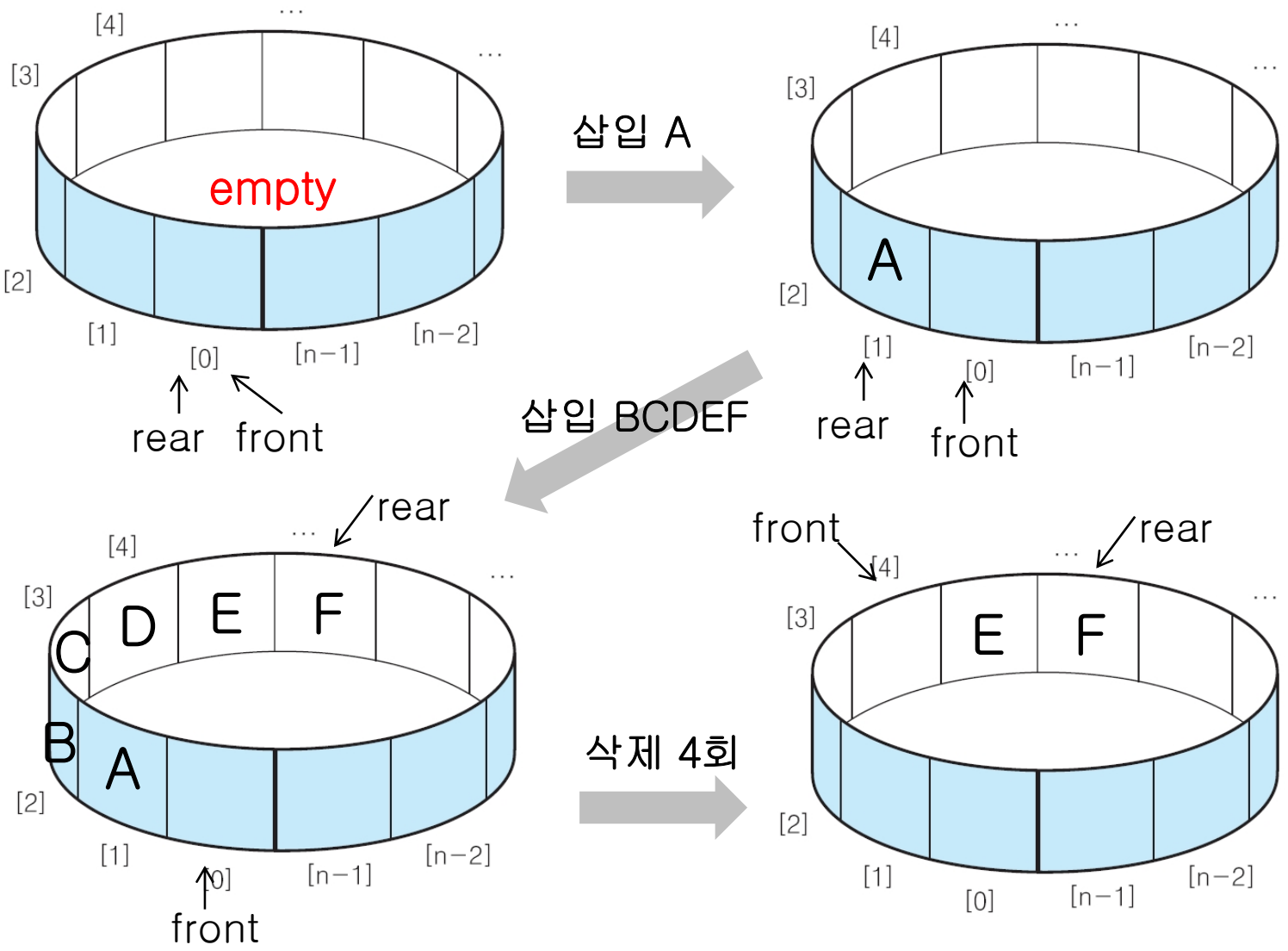
- 예를 들어 n=5인 경우



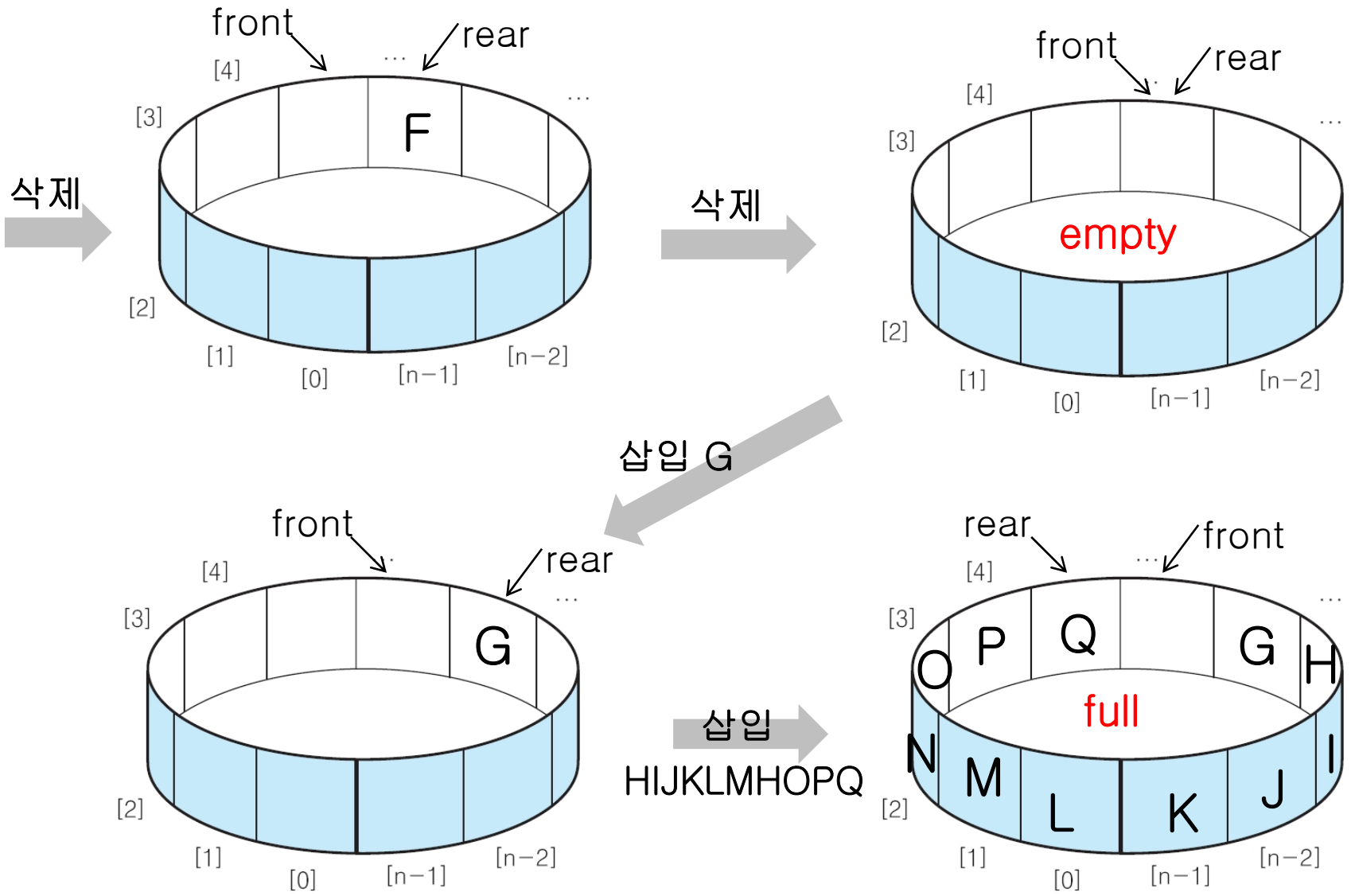



# ❑ 큐의 구현 – 원형 큐

- 공백 상태와 포화 상태 구분을 쉽게 하기 위해서 front가 있는 자리는 사용하지 않고 항상 빈자리로 둔다.

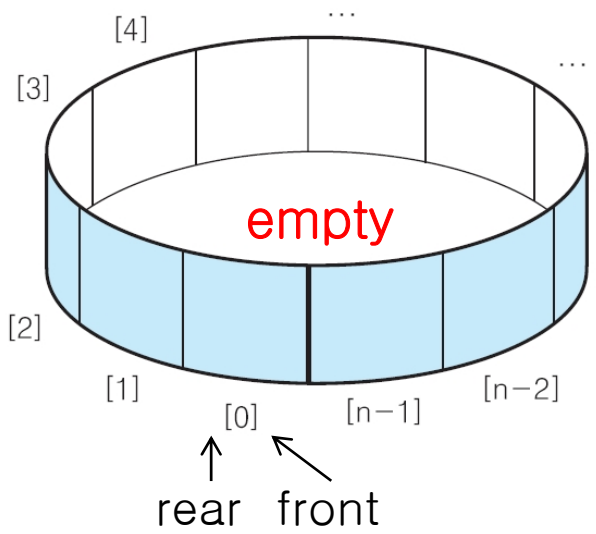


# ❑ 큐의 구현 – 원형 큐


front
rear
[4]
[3]
[2]
[1]
[0]
[n−1]
[n−2]
F
삭제
삭제
empty
삽입 G
G
삽입
HIJKLMHOPQ
O
P
Q
G
H
N
M
L
K
J
I
full

# ❑ 큐의 구현 – 원형 큐

- 원형 큐 생성 알고리즘
  - 크기가 n인 1차원 배열 생성
  - front와 rear를 0 으로 초기화




```
createQueue()  // 공백 원형 큐 생성
    cQ[n];
    front ← 0;
    rear ← 0;
end createQueue()
```

## ❑ 큐의 구현 – 원형 큐

- 원형 큐의 공백상태/포화상태 검사 알고리즘
  - 공백 상태
    front = rear
  - 포화 상태 : 삽입할 rear의 다음 위치 = front의 현재 위치
    (rear+1) mod n = front

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

isFull(cQ)
    if(((rear+1) mod n)=front) then return true;
    else return false;
end isFull()
```

◆ 큐의 현재 원소 수를 나타내는 변수를 사용하는 경우
큐의 한 칸을 비워둘 필요 없음
공백 상태, 포화 상태 조건은?

## ❑ 큐의 구현 – 원형 큐

- 원형 큐의 삽입 알고리즘
  ① rear의 값을 조정하여 삽입할 자리를 준비
  rear ← (rear+1) mod n;
  ② 준비한 자리 cQ[rear]에 원소 item을 삽입

```
enQueue(cQ, item)
    if(isFull(cQ)) then Queue_Full();
    else {
        rear ← (rear+1) mod n;   // ①
        cQ[rear] ← item;         // ②
    }
end enQueue()
```

## ❑ 큐의 구현 – 원형 큐

- 원형 큐의 삭제 알고리즘
  ① front의 값을 조정하여 삭제할 자리를 준비
  front ← (front+1) mod n;
  ② 준비한 자리에 있는 원소 cQ[front]를 삭제하여 반환

```
deQueue(cQ)
    if(isEmpty(cQ)) then Queue_Empty();
    else {
        front ← (front+1) mod n;    // ①
        return cQ[front];           // ②
    }
end deQueue()
```

## ❑ 큐의 구현 – 원형 큐

❖ 배열로 구현한 character 원형 큐

```
public class ArrayCQueue {
    private int front;
    private int rear;
    private int queueSize;
    private char[] itemArray;

    public ArrayCQueue(int queueSize) {
        front = 0;
        rear = 0;
        this.queueSize = queueSize;
        itemArray = new char[queueSize];
    }

    public boolean isEmpty() {
        return (front == rear);
    }

    public boolean isFull() {
        return (((rear+1) % queueSize) == front);
    }
    // 다음 슬라이드에 계속
```

## ❑ 큐의 구현 – 원형 큐

```
public void enQueue(char item) {
    if(isFull()) {
        System.out.println("Inserting fail!  Circular Queue is full!!");
    }
    else{
        rear = (rear+1) % queueSize;
        itemArray[rear] = item;
    }
}

public char deQueue() {
    if(isEmpty()) {
        System.out.println("Deleting fail! Circular Queue is empty!!");
        예외 발생;
    }
    else{
        front = (front+1) % queueSize;
        return itemArray[front];
    }
}
// 다음 슬라이드에 계속
```

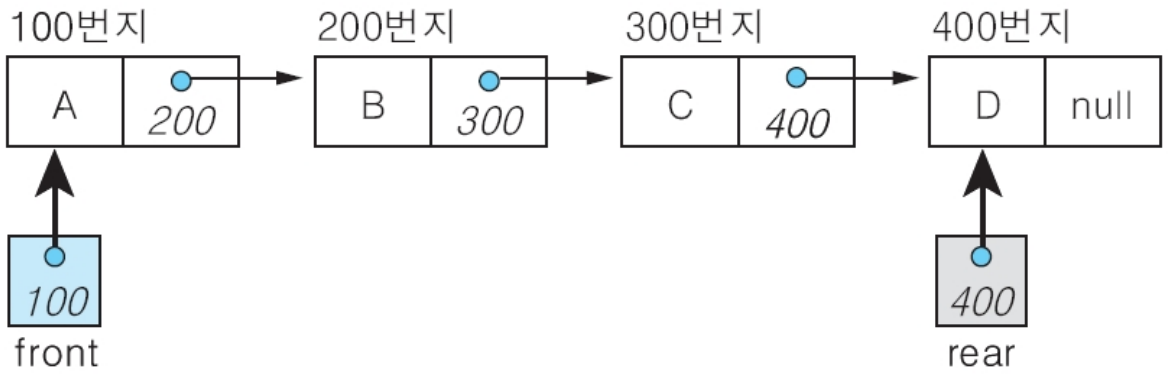
## ❑ 큐의 구현 – 원형 큐

```
    public char peek() {
        if(isEmpty()) {
            System.out.println("Peeking fail!  Circular Queue is empty!!");
            예외 발생;
        }
        else  {
            return itemArray[(front+1) % queueSize];
        }
    }

    public String toString() {
        String result = "Array Circular Queue>> ";
        for(int i=(front+1)%queueSize; i!=(rear+1)%queueSize;
                                              i=(i+1)%queueSize)
            result += itemArray[i]  + " ";
        return result;
    }
}
```

# ❑ 큐의 구현 – 연결리스트 구현

## ❖ 연결리스트로 구현한 큐

- 단순 연결리스트를 이용하여 큐를 구현할 수 있다.
  - front : 첫 번째 노드를 가리키는 포인터
  - rear : 마지막 노드를 가리키는 포인터
  - 큐 노드의 순서는 링크로 표현됨
- 상태 표현
  - 초기 상태 : front = rear = null
  - 공백 상태 : front = rear = null
- 큐의 구조
  - 예) A, B, C, D 순으로 큐에 삽입한 후



## ❑ 큐의 구현 – 연결리스트 구현

- 큐 생성 알고리즘
  - 초기화 : front = rear = null

```
createLinkedQueue()  // 공백 연결리스트 큐 생성
    front ← null;
    rear ← null;
end createLinkedQueue()
```

- 큐의 공백상태 검사 알고리즘
  - 공백 상태 : front = null

```
isEmpty(LQ)
    if(front=null) then return true;
    else return false;
end isEmpty()
```

## ❑ 큐의 구현 – 연결리스트 구현

- 큐 삽입 알고리즘

```
enQueue(LQ, item)
    newNode ← getNode();
    newNode.data ← item;
    newNode.link ← null;
    if (isEmpty(LQ)) then {  // 큐가 공백인 경우
        rear ← newNode;
        front ← newNode;
    }
    else {                          // 큐가 공백이 아닌 경우
        rear.link ← newNode;
        rear ← newNode;
    }
end enQueue()
```

## ❑ 큐의 구현 – 연결리스트 구현

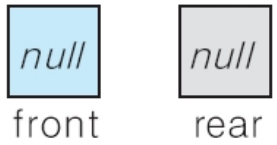
- 큐 삭제 알고리즘

```
deQueue(LQ)
    if(isEmpty(LQ)) then Queue_Empty();
    else {
        item ← front.data;
        front ← front.link;
        if (isEmpty(LQ)) then rear ← null;
        return item;
    }
end deQueue()
```

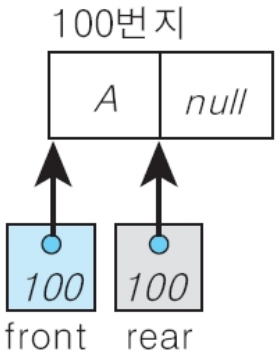
# ❑ 큐의 구현 – 연결리스트 구현

- 연산 과정

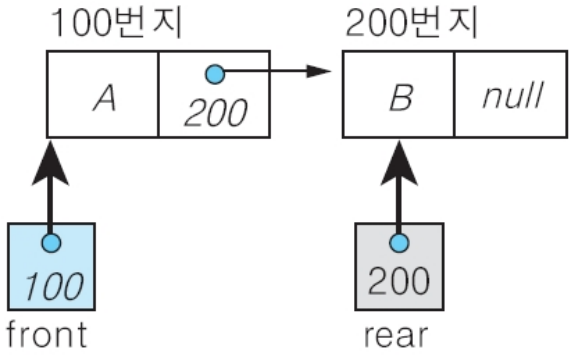
① createLinkedQueue();



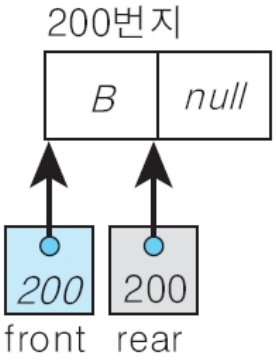

② enQueue(LQ, A);




③ enQueue(LQ, B);




④ deQueue(LQ);



## ❑ 큐의 구현 – 연결리스트 구현

❖ 연결 자료구조 방식을 이용하여 구현한 문자 큐

```
public class LinkedQueue {
    private Node front, rear;

    public LinkedQueue() {
        front = null;
        rear = null;
    }

    public void enQueue(char item) {
        Node newNode = new Node();
        newNode.data = item;
        newNode.link = null;
        if(isEmpty()) {
            front = newNode;
            rear = newNode;
        }
        else {
            rear.link = newNode;
            rear = newNode;
        }
    }                                   // 다음 슬라이드에 계속
```

## ❑ 큐의 구현 – 연결리스트 구현

```
public char deQueue() {
    if(isEmpty()) {
        System.out.println("Deleting fail! Linked Queue is empty!!");
        예외 발생;
    }
    else{
        char item = front.data;
        front = front.link;
        if(front == null)
            rear = null;
        return item;
    }
}

public boolean isEmpty() {
    return (front == null);
}
// 다음 슬라이드에 계속
```

## ❑ 큐의 구현 – 연결리스트 구현

```
    public String toString() {
        String result = "Linked Queue>> ";
        Node temp = front;
        while(temp != null){
            result += temp.data + " ";
            temp = temp.link;
        }
        return result;
    }

    private class Node {
        char data;
        Node link;
    }
}
```

# ❑ Deque

- 덱(Deque, double-ended queue)
  - 양쪽 끝에서 삽입과 삭제 연산이 모두 가능한 선형 자료구조

## Deque

- 덱에 대한 추상 자료형

**ADT deque**
데이터 : 0개 이상의 원소를 가진 유한 순서 리스트
연산 : DQ∈deque; item∈Element;
createDeque() ::= create an empty DQ;
// 공백 덱을 생성하는 연산
isEmpty(DQ) ::= if (DQ is empty) then return true
else return false;
// 덱이 공백인지 아닌지를 확인하는 연산
insertFront(DQ, item) ::= insert item at the front of DQ;
// 덱의 front 앞에 item(원소)을 삽입하는 연산
insertRear(DQ, item) ::= insert item at the rear of DQ;
// 덱의 rear 뒤에 item(원소)을 삽입하는 연산
deleteFront(DQ) ::= if (isEmpty(DQ)) then return null
else { delete and return the front item of DQ };
// 덱의 front에 있는 item(원소)을 덱에서 삭제하고 반환하는 연산
// 다음 슬라이드에 계속

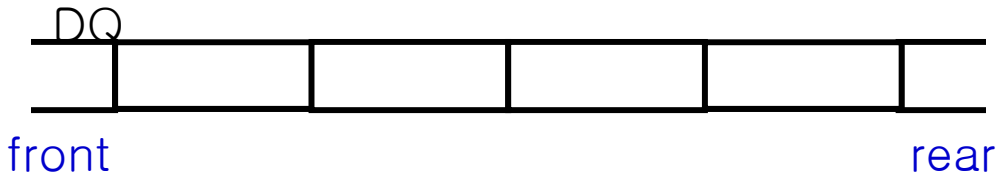
## ❑ Deque

```
    deleteRear(DQ) ::= if (isEmpty(DQ)) then return null
                       else { delete and return the rear item of DQ };
        // 덱의 rear에 있는 item(원소)을 덱에서 삭제하고 반환하는 연산
    removeFront(DQ) ::= if (isEmpty(DQ)) then return null
                       else { remove the front item of DQ };
        // 덱의 front에 있는 item(원소)을 삭제하는 연산
    removeRear(DQ) ::= if (isEmpty(DQ)) then return null
                       else { remove the rear item of DQ };
        // 덱의 rear에 있는 item(원소)을 삭제하는 연산
    peekFront(DQ) ::= if (isEmpty(DQ)) then return null
                       else { return the front item of the DQ };
        // 덱의 front에 있는 item(원소)을 반환하는 연산
    peekRear(DQ) ::= if (isEmpty(DQ)) then return null
                       else { return the rear item of the DQ };
        // 덱의 rear에 있는 item(원소)을 반환하는 연산
End deque
```

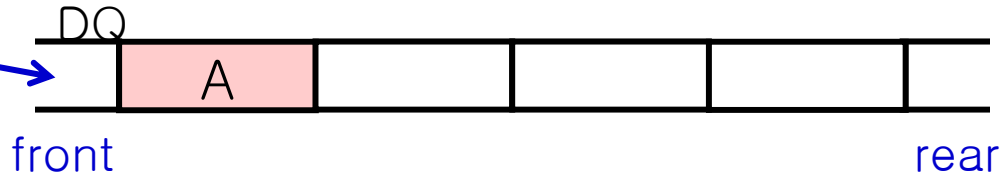
# Deque

- 덱의 연산 과정

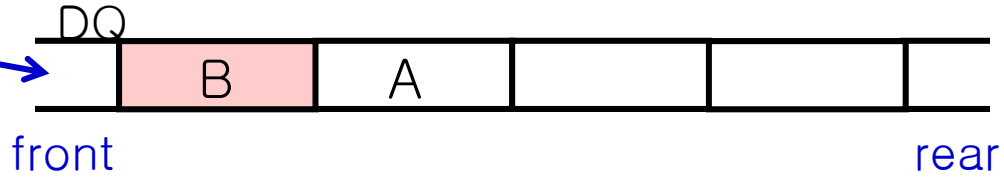
① createDeque();



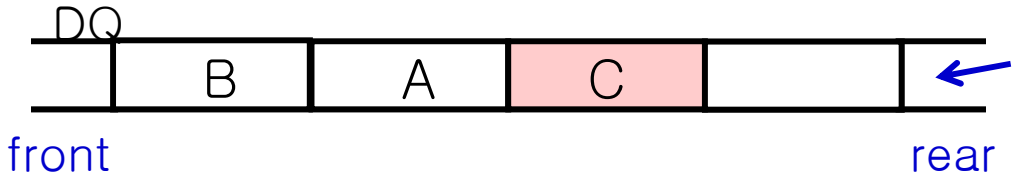

② insertFront(DQ, 'A');




③ insertFront(DQ, 'B');

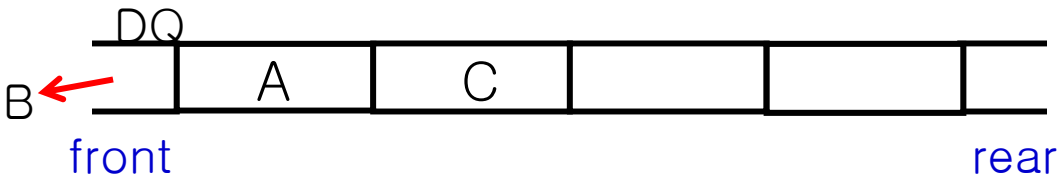



④ insertRear(DQ, 'C');

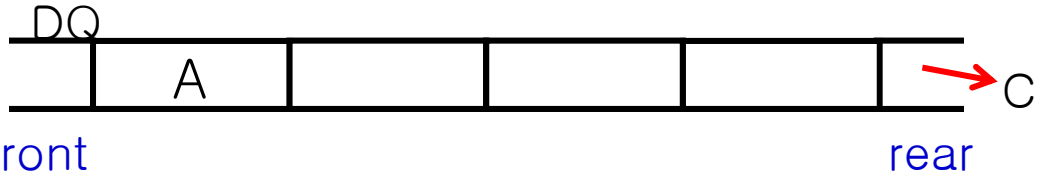


# ❑ Deque

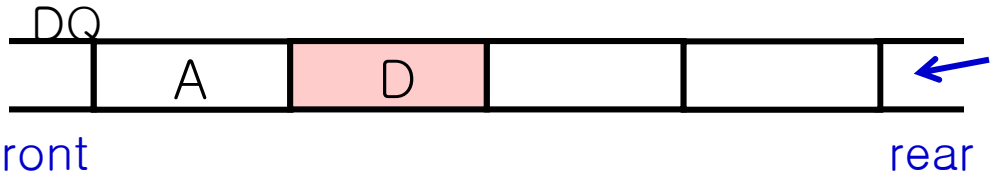
⑤ deleteFront(DQ);



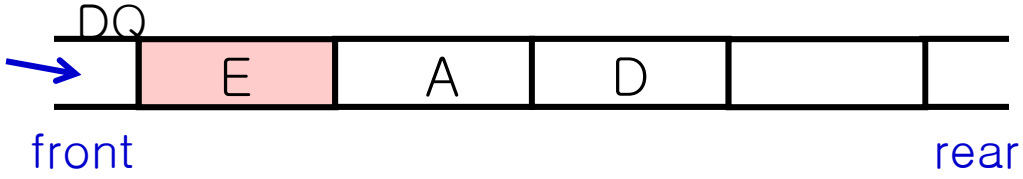

⑥ deleteRear(DQ);



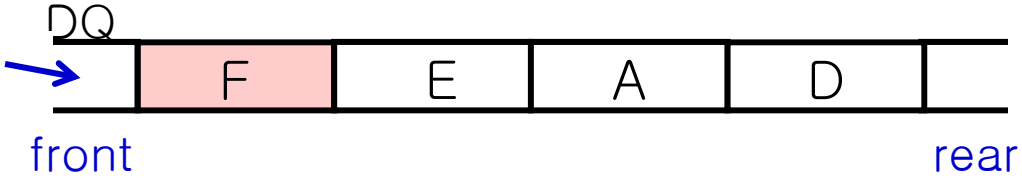

⑦ insertRear(DQ, 'D');




⑧ insertFront(DQ, 'E');

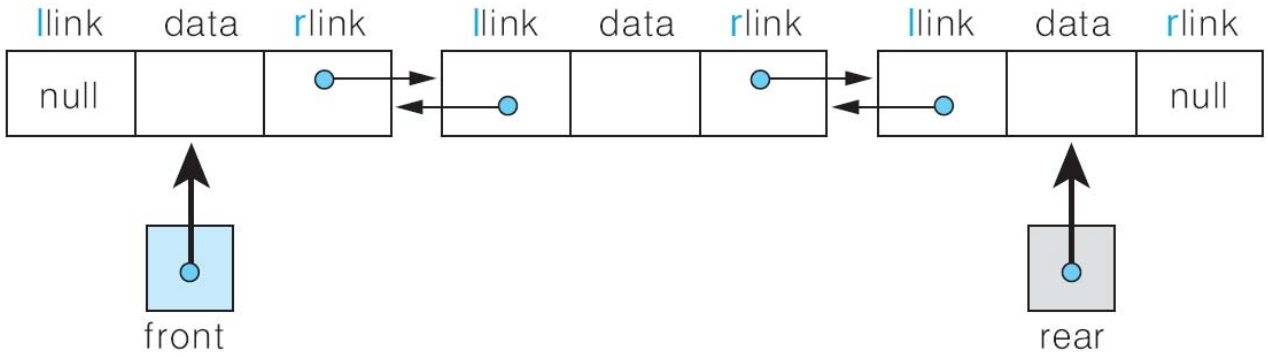



⑨ insertFront(DQ, 'F');



# ❑ 덱의 구현

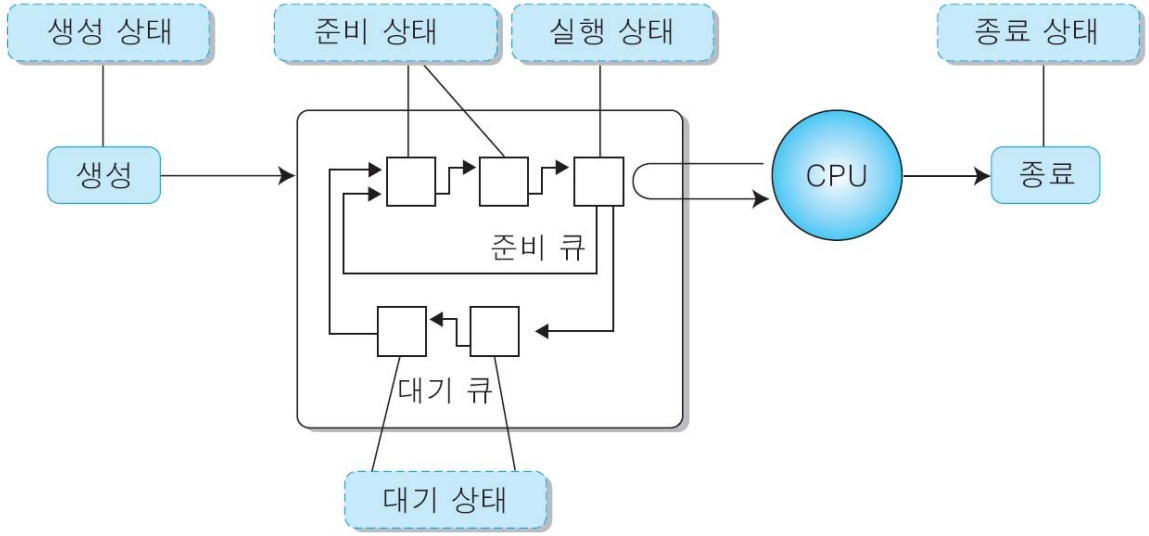
- 이중 연결리스트로 구현
  - 덱은 순차 자료구조(즉, 배열)로 구현하면 비효율적이다.
    - ➢ 양쪽 끝에서 삽입/삭제 연산을 수행하면서 크기 변화와 저장된 원소의 순서 변화가 많으므로
  - 양방향으로 연산이 가능하도록 이중 연결리스트로 구현한다.



# ❑ 큐의 응용

## ❖ 운영체제의 작업 큐

- 프린터 버퍼 큐
  - CPU에서 프린터로 보낸 데이터 순서대로(선입선출) 프린터에서 출력하기 위해서 선입선출 구조의 큐 사용
- 스케줄링 큐
  - CPU 사용을 요청한 프로세스들을 스케줄링하기 위해 큐를 사용




## ❖ 시뮬레이션에서의 큐잉 시스템

- 시뮬레이션을 위한 수학적 모델링에서 대기행렬과 대기시간 등을 모델링하기 위해서 큐잉 이론(Queue Theory) 사용