

데이터구조와 알고리즘 (Queue)

남춘성

Queue

- 큐(Queue)

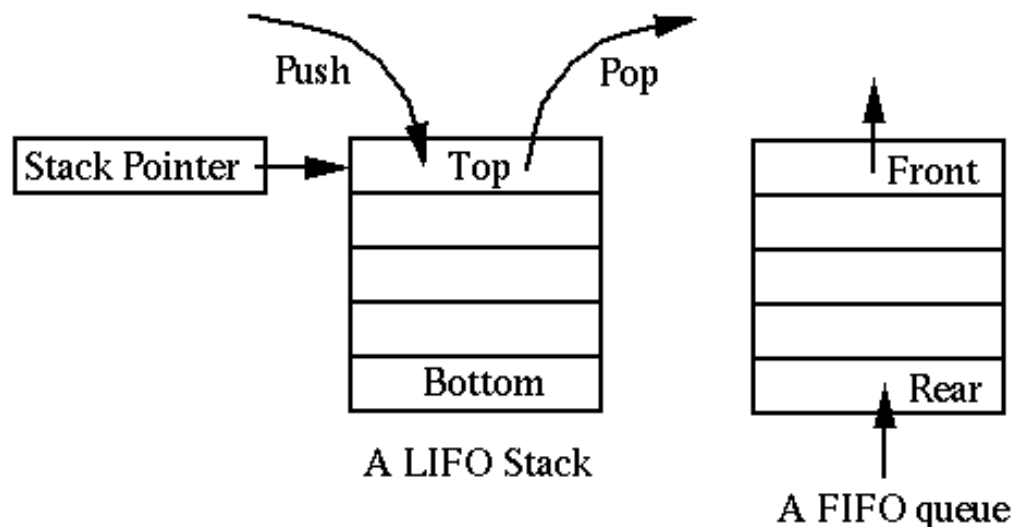
- 스택과 마찬가지로 삽입과 삭제의 위치가 제한되어있는 유한 순서 리스트
- 큐의 뒤에서는 삽입만 하고, 앞에서는 삭제만 할 수 있는 구조
 - 삽입한 순서대로 원소가 나열되어 가장 먼저 삽입(First-In)한 원소는 맨 앞에 있다가 가장 먼저 삭제(First-Out)된다.
- ✓ **선입선출 구조 (FIFO, First-In-First-Out)**
- 스택과 큐의 구조 비교



스택



큐



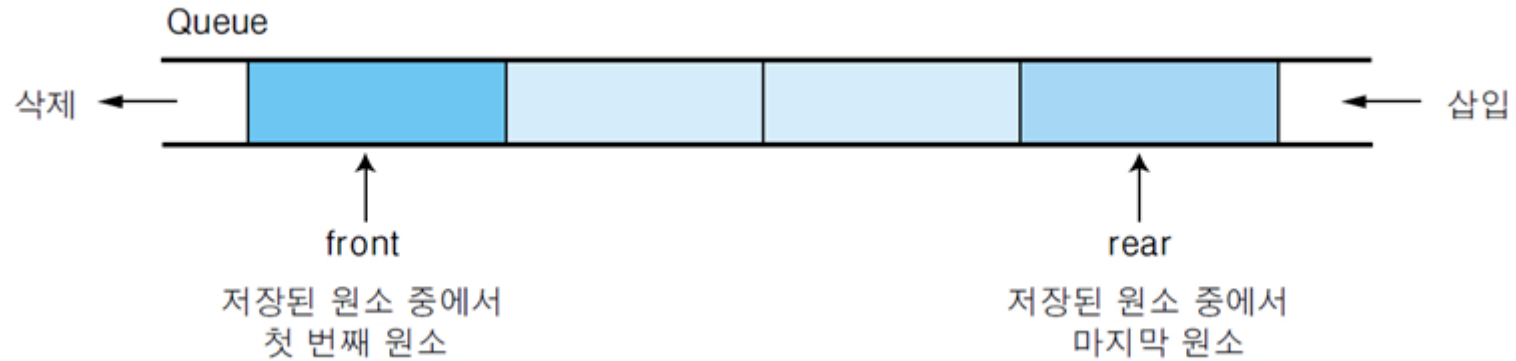
[그림 7-1] 스택과 큐의 구조 예

Queue

- 큐의 구조



[그림 7-2] 꼬리잡기 놀이의 머리와 꼬리



Queue

- 큐의 연산
 - 삽입 : **enqueue**
 - 삭제 : **dequeue**
- 스택과 큐의 연산 비교

스택과 큐에서의 삽입과 삭제 연산 비교

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

Queue

- 추상 자료형 Queue

ADT Queue

데이터 : 0개 이상의 원소를 가진 유한 순서 리스트

연산 :

$Q \in \text{Queue}; \text{item} \in \text{Element};$

createQueue() ::= create an empty Q;

// 공백 큐를 생성하는 연산

enqueue(Q, item) ::= insert item at the rear of Q;

// 큐의 rear에 item(원소)을 삽입하는 연산

isEmpty(Q) ::= if (Q is empty) then return true

else return false;

// 큐가 공백인지 아닌지를 확인하는 연산

dequeue(Q) ::= if (isEmpty(Q)) then return error

else { delete and return the front item of Q };

// 큐의 front에 있는 item(원소)을 큐에서 삭제하고 반환하는 연산

delete(Q) ::= if (isEmpty(Q)) then return error

else { delete the front item of Q };

// 큐의 front에 있는 item(원소)을 삭제하는 연산

peek(Q) ::= if (isEmpty(Q)) then return error

else { return the front item of the Q };

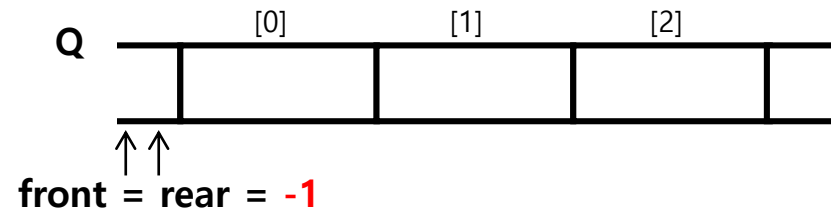
// 큐의 front에 있는 item(원소)을 반환하는 연산

End Queue

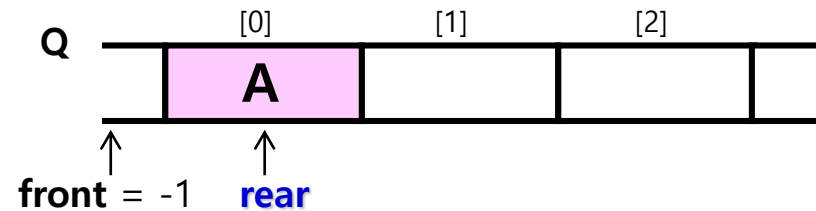
Queue-insert

- 큐의 연산 과정

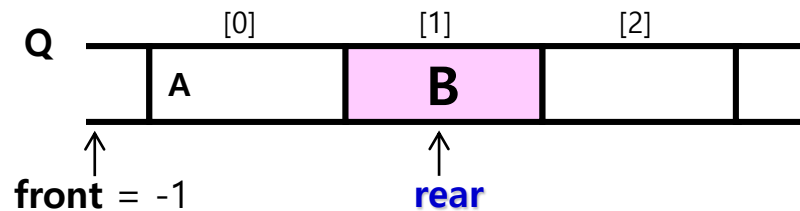
① 공백 큐 생성 : createQueue();



② 원소 A 삽입 : enqueue(Q, A);

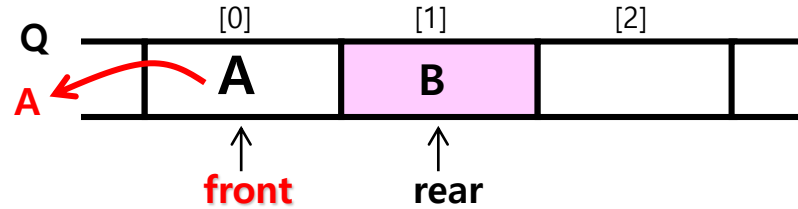


③ 원소 B 삽입 : enqueue(Q, B);

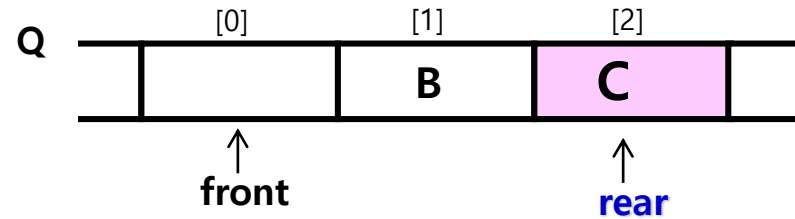


Queue-delete

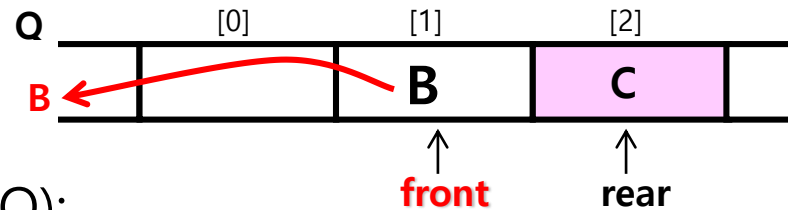
④ 원소 삭제 : `deQueue(Q);`



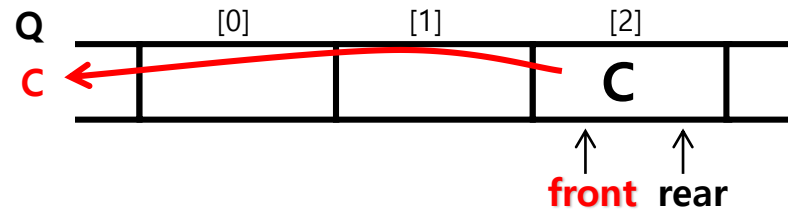
⑤ 원소 C 삽입 : `enqueue(Q, C);`



⑥ 원소 삭제 : `deQueue(Q);`



⑦ 원소 삭제 : `deQueue(Q);`



Queue

- 선형 큐

- 1차원 배열을 이용한 큐

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

- 상태 표현

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

Queue

– 초기 공백 큐 생성 알고리즘

- 크기가 n 인 1차원 배열 생성
- Front 와 rear를 -1로 초기화

```
createQueue() :  
    new_queue = []  
    front = -1  
    rear = -1  
    return new_queue
```

위와 같이 하면 될까? (Python에서?) – 함수를 알아야 함

Queue

- 공백 큐 검사 알고리즘과 포화상태 검사 알고리즘

- 공백 상태 : $\text{front} = \text{rear}$
- 포화 상태 : $\text{rear} = n-1$ (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()
```

Queue

- 큐의 삽입 알고리즘

```
enqueueer(Q, item)
    if(isFull(Q)) then Queue_Full();
    else {
        rear <- rear+1;
        Q[rear] <- item
    }
end enqueue()
```

- 마지막 원소의 뒤에 삽입해야 하므로
 - ① 마지막 원소의 인덱스를 저장한 **rear**의 값을 하나 증가시켜 삽입할 자리 준비
 - ② 그 인덱스에 해당하는 배열원소 Q[rear]에 item을 저장

Queue

- 큐의 삭제 알고리즘

```
dequeue(Q)
    if(isEmpty(Q)) then Queue_Empty();
    else {
        front <- front+1;
        return Q[front];
    }
end dequeue()
delete(Q)
    if(isEmpty(Q)) then Queue_Empty();
    else front <- front+1;
end delete()
```

- 가장 앞에 있는 원소를 삭제해야 하므로
 - ① **front**의 위치를 한자리 뒤로 이동하여 큐에 남아있는 첫 번째 원소의 위치로 이동하여 삭제할 자리 준비
 - ② 그 자리의 원소를 삭제하여 반환

Queue

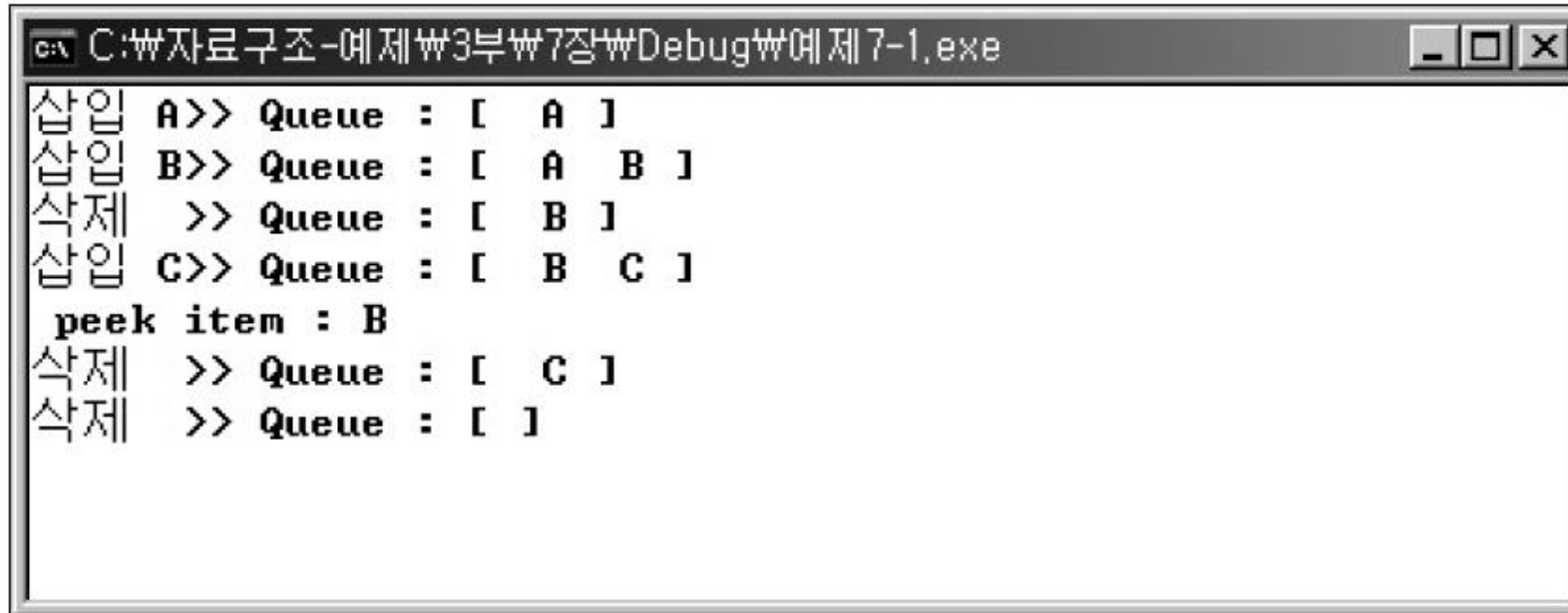
– 큐의 검색 알고리즘

```
peek(Q)
    if(isEmpty(Q)) then Queue_Empty();
    else return Q[front+1];
end peek()
```

- 가장 앞에 있는 원소를 검색하여 반환하는 연산
 - ① 현재 **front**의 한자리 뒤(front+1)에 있는 원소, 즉 큐에 있는 첫 번째 원소를 반환

Queue

- 실습 1



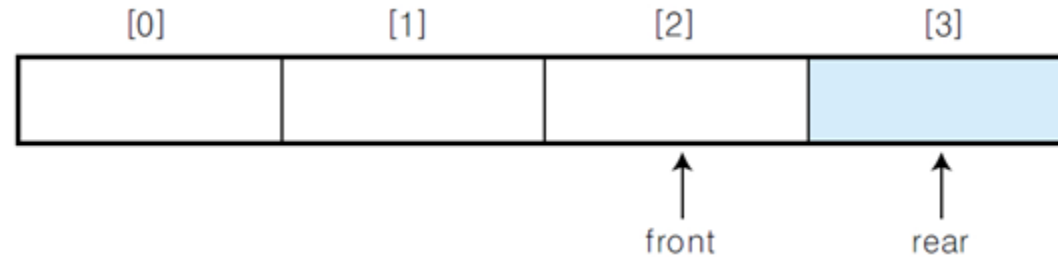
A screenshot of a Windows command prompt window. The title bar shows the path "C:\자료구조-예제\3부\7장\Debug\예제 7-1.exe". The command history shows a series of operations on a queue: adding 'A', adding 'B', removing 'A', adding 'C', peeking 'B', removing 'C', and finally removing 'B', leaving the queue empty.

```
C:\자료구조-예제\3부\7장\Debug\예제 7-1.exe
삽입 A>> Queue : [ A ]
삽입 B>> Queue : [ A B ]
삭제 >> Queue : [ B ]
삽입 C>> Queue : [ B C ]
peek item : B
삭제 >> Queue : [ C ]
삭제 >> Queue : [ ]
```

Queue

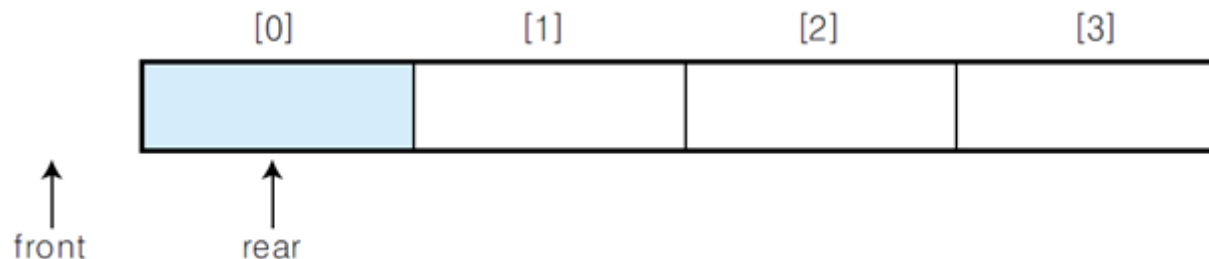
– 선형 큐의 잘못된 포화상태 인식

- 큐에서 삽입과 삭제를 반복하면서 아래와 같은 상태일 경우, 앞부분에 빈자리가 있지만 $rear=n-1$ 상태이므로 포화상태로 인식하고 더 이상의 삽입을 수행하지 않는다.



– 선형 큐의 잘못된 포화상태 인식의 해결 방법-1

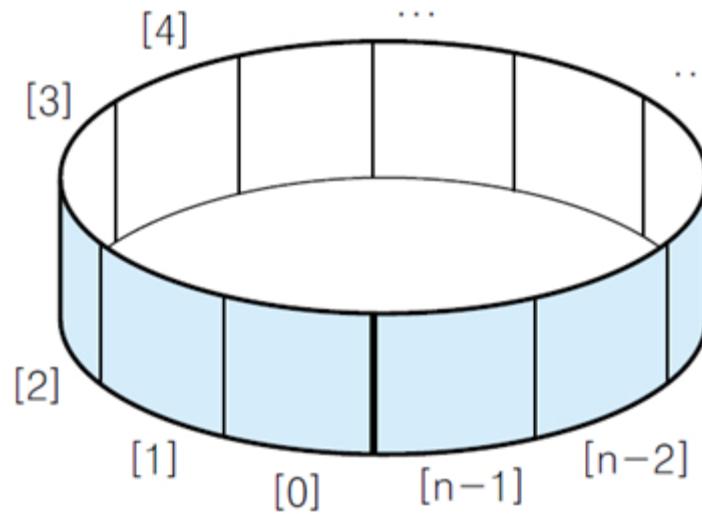
- 저장된 원소들을 배열의 앞부분으로 이동시키기
 - 순차자료에서의 이동 작업은 연산이 복잡하여 효율성이 떨어짐



Queue

- 선형 큐의 잘못된 포화상태 인식의 해결 방법-2

- 1차원 배열을 사용하면서 논리적으로 배열의 처음과 끝이 연결되어 있다고 가정하고 사용 \Rightarrow **원형큐**
- 원형 큐의 논리적 구조



Queue

- 원형 큐의 구조

- 초기 공백 상태 : $\text{front} = \text{rear} = 0$
- front 와 rear 의 위치가 배열의 마지막 인덱스 $n-1$ 에서 논리적인 다음 자리인 인덱스 0 번으로 이동하기 위해서 **나머지연산자 mod**를 사용
 - $3 \div 4 = 0 \dots 3$ (몫=0, 나머지=3)
 - $3 \bmod 4 = 3$

	삽입위치	삭제위치
선형큐	$\text{rear} = \text{rear} + 1$	$\text{front} = \text{front} + 1$
원형큐	$\text{rear} = (\text{rear} + 1) \bmod n$	$\text{front} = (\text{front} + 1) \bmod n$

- 사용조건) 공백 상태와 포화 상태 구분을 쉽게 하기 위해서 front 가 있는 자리는 사용하지 않고 항상 빈자리로 둔다.

Queue

– 초기 공백 원형 큐 생성 알고리즘

- 크기가 n 인 1차원 배열 생성
- Front 와 rear를 0으로 초기화

```
createQueue()  
    cQ = [ , , , , , .. , n]  
    front <- 0  
    rear <- 0  
end createQueue()
```

Queue

- 원형 큐의 공백상태 검사 알고리즘과 포화상태 검사 알고리즘
 - 공백 상태 : $\text{front} = \text{rear}$
 - 포화 상태 : 삽입할 rear의 다음 위치 = front의 현재 위치
 - $(\text{rear}+1) \bmod n = \text{front}$

```
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()
```

Queue

– 원형 큐의 삽입알고리즘

- rear 값을 조정하여 삽입할 자리를 준비 : $\text{rear} \leftarrow (\text{rear} + 1) \bmod n$
- 준비한 자리 $\text{cQ}[\text{rear}]$ 에 원소 item 삽입

```
enqueueer(cQ, item)
    if(isFull(cQ)) then Queue_Full();
    else {
        rear <- (rear+1) mod n;
        cQ[rear] <- item
    }
end enqueueer()
```

Queue

– 원형 큐의 삭제 알고리즘

- Front 값을 조정하여 삭제할 자리를 준비
- 준비한 자리에 있는 원소 $cQ[\text{front}]$ 를 삭제하여 반환

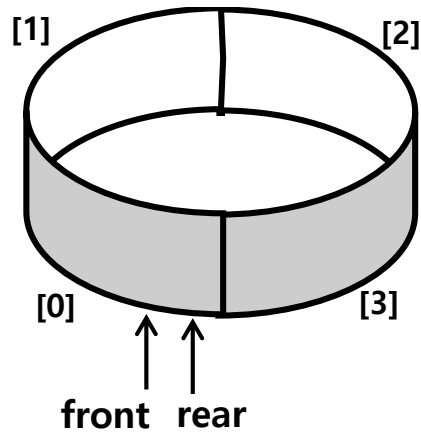
```
deQueue(cQ)
    if(isEmpty(cQ)) then Queue_Empty():
    else {
        front <- (front+1) mod n;
        return cQ[front];
    }
end deQueue()

delete(cQ)
    if(isEmpty(cQ)) then Queue_Exmpty();
    else front <- (front+1) mod n;
end delete
```

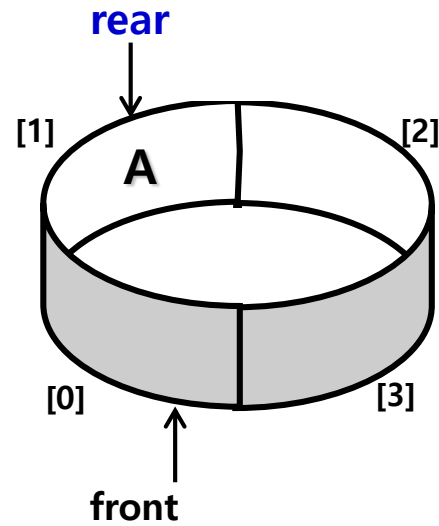
Queue

– 원형 큐에서의 연산 과정

① createQueue();

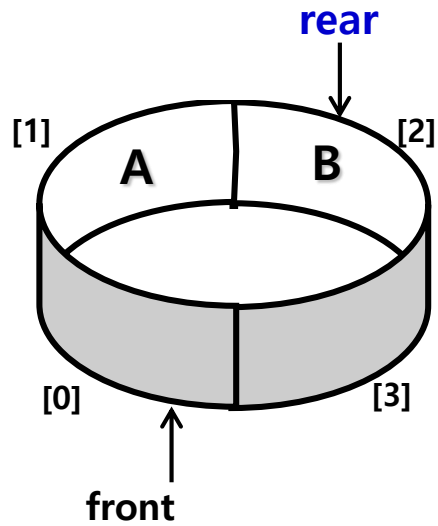


② enqueue(cQ, A);

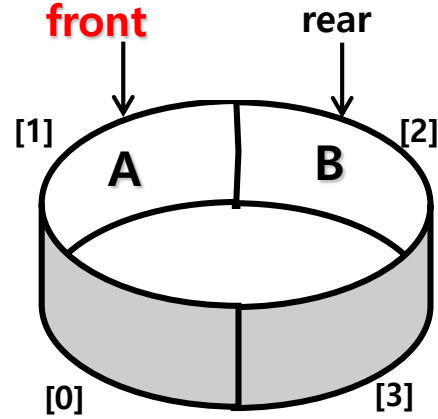


Queue

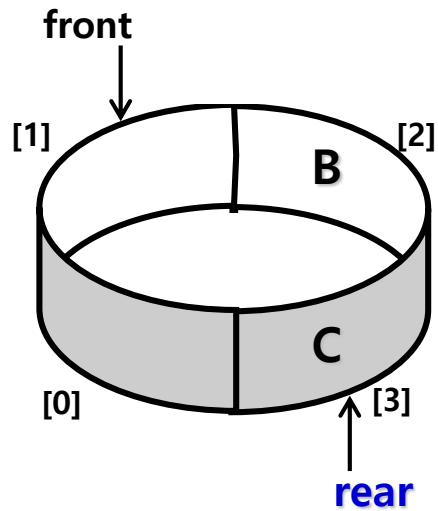
③ enQueue(cQ, B);



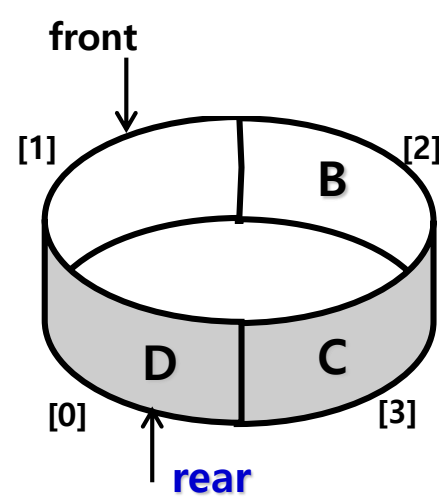
④ deQueue(cQ);



⑤ enQueue(cQ, C);

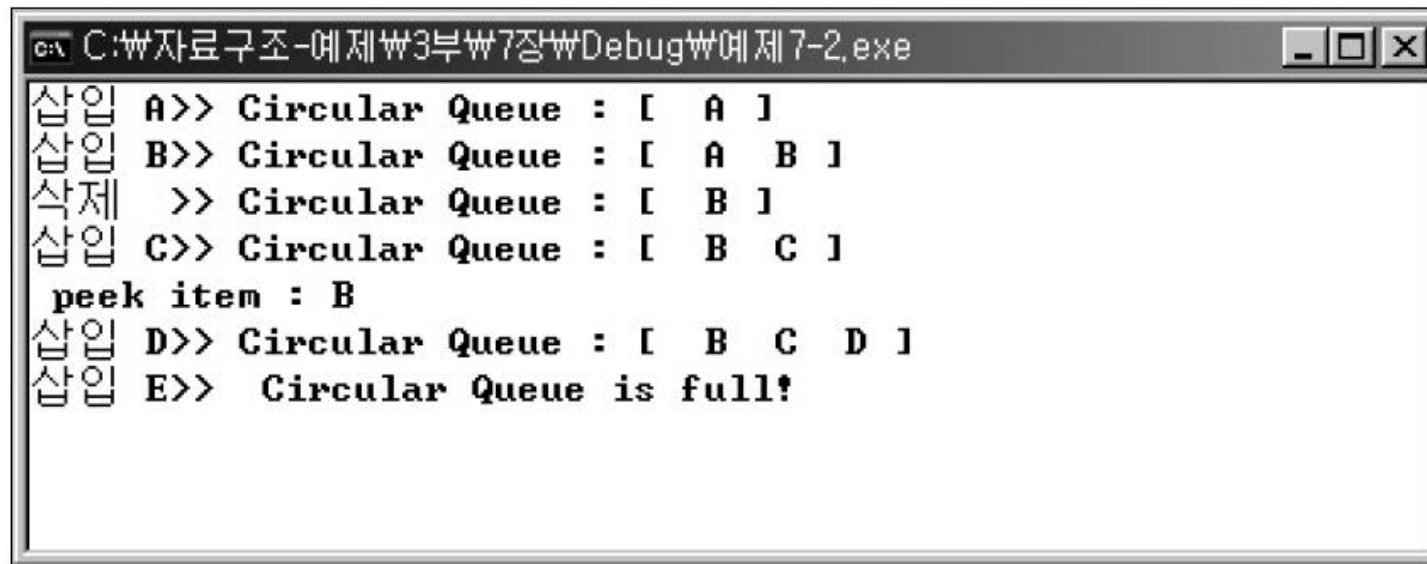


⑥ enQueue(cQ, D);



Queue

- 실습 II



```
C:\₩자료구조-예제₩3부₩7장₩Debug₩예제 7-2.exe
삽입 A>> Circular Queue : [ A ]
삽입 B>> Circular Queue : [ A B ]
삭제 >> Circular Queue : [ B ]
삽입 C>> Circular Queue : [ B C ]
peek item : B
삽입 D>> Circular Queue : [ B C D ]
삽입 E>> Circular Queue is full!
```


Queue

- 연결 큐

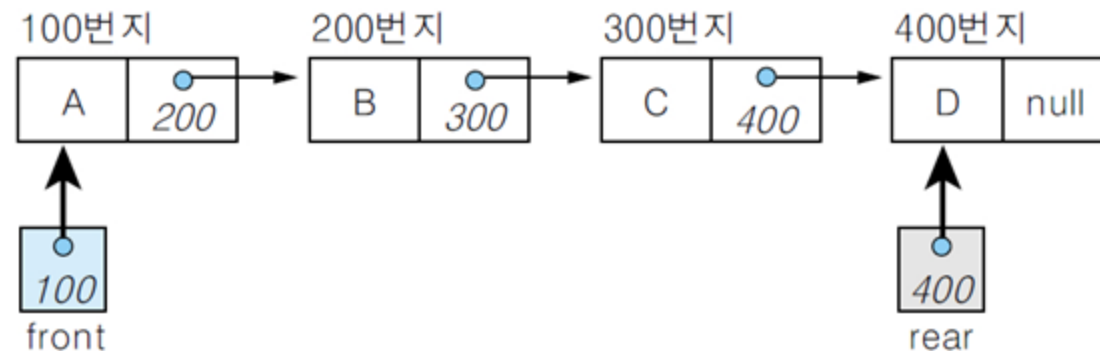
- 단순 연결 리스트를 이용한 큐

- 큐의 원소 : 단순 연결 리스트의 노드
 - 큐의 원소의 순서 : 노드의 링크 포인터로 연결
 - 변수 front : 첫 번째 노드를 가리키는 포인터 변수
 - 변수 rear : 마지막 노드를 가리키는 포인터 변수

- 상태 표현

- 초기 상태와 공백 상태 : front = rear = null

- 연결 큐의 구조

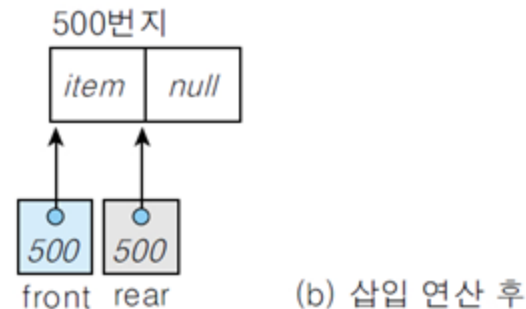
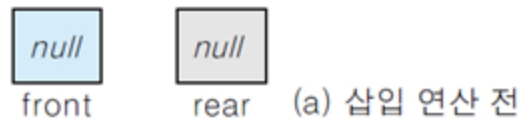


연결 큐 삽입

- ① 삽입할 새 노드를 생성하여 데이터 필드에 item을 저장한다. 삽입할 새 노드는 연결 큐의 마지막 노드가 되어야 하므로 링크 필드에 null을 저장한다.

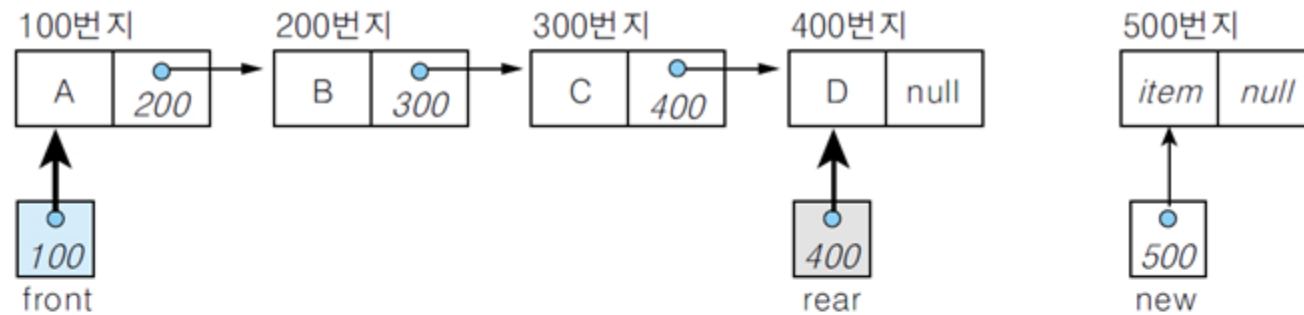


- ② 새 노드를 삽입하기 전에 연결 큐가 공백인지 아닌지를 검사한다. 연결 큐가 공백인 경우에는 삽입할 새 노드가 큐의 첫 번째 노드이자 마지막 노드이므로 포인터 front와 rear가 모두 새 노드를 가리키도록 설정한다.

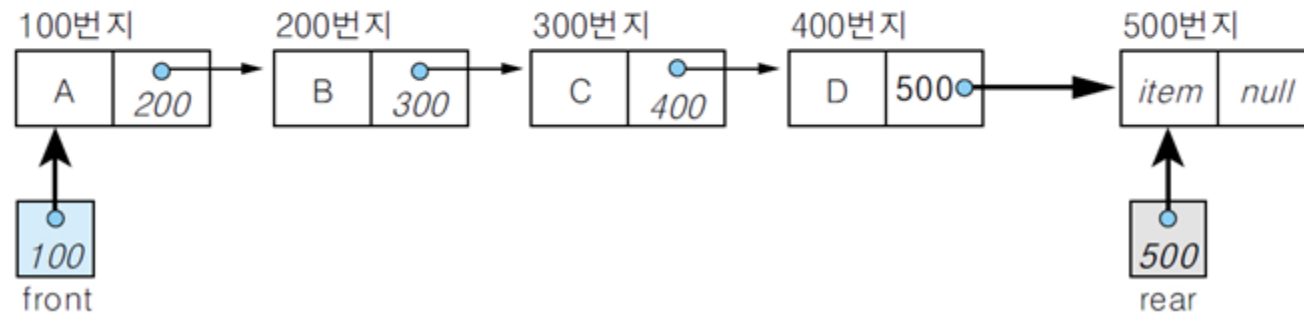


연결 큐 삽입

③ 큐가 공백이 아닌 경우, 즉 노드가 있는 경우에는 현재 큐의 마지막 노드의 뒤에 새 노드를 삽입하고 마지막 노드를 가리키는 rear가 삽입한 새 노드를 가리키도록 설정한다.



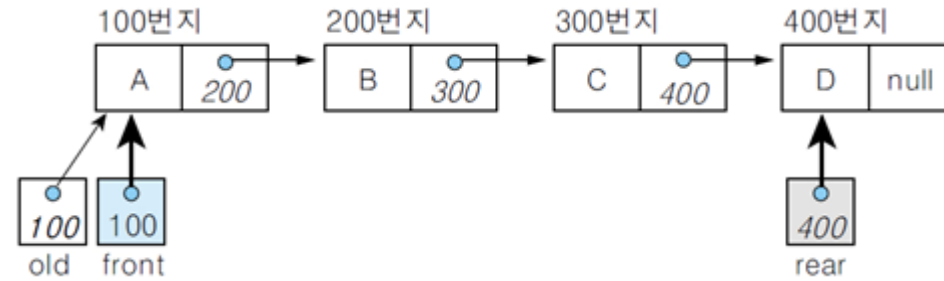
(a) 삽입 연산 전



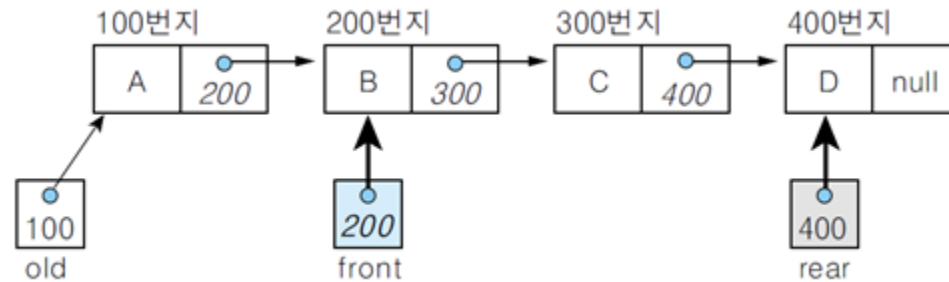
(b) 삽입 연산 후

연결 큐 삭제

- ① 삭제연산에서 삭제할 노드는 큐의 첫 번째 노드로서 포인터 front가 가리키고 있는 노드이다. front가 가리키는 노드를 포인터 old가 가리키게 하여 삭제할 노드를 지정한다.

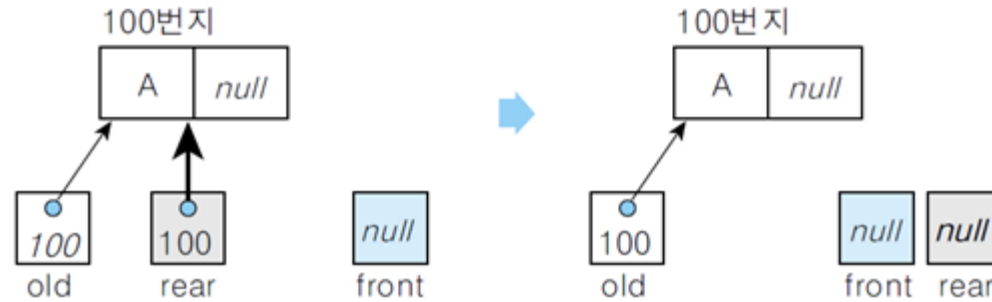


- ② 삭제연산 후에는 현재 front 노드의 다음 노드가 front 노드(첫번째 노드)가 되어야 하므로, 포인터 front를 재설정한다.

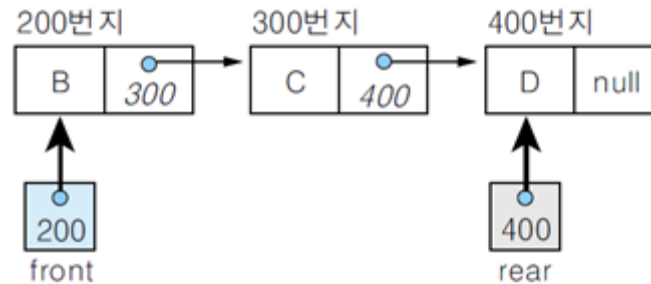


연결 큐 삭제

- ③ 현재 큐에 노드가 하나뿐이어서 삭제연산 후에 공백 큐가 되는 경우 :
☞ 큐의 마지막 노드가 없으므로 포인터 rear를 null로 설정한다.



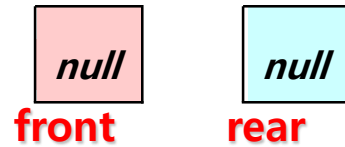
- ④ 포인터 old가 가리키고 있는 노드를 삭제하고, 메모리 공간을 시스템에 반환(returnNode())한다



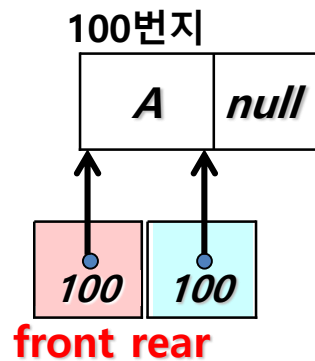
연결 큐

- 연결 큐에서의 연산 과정

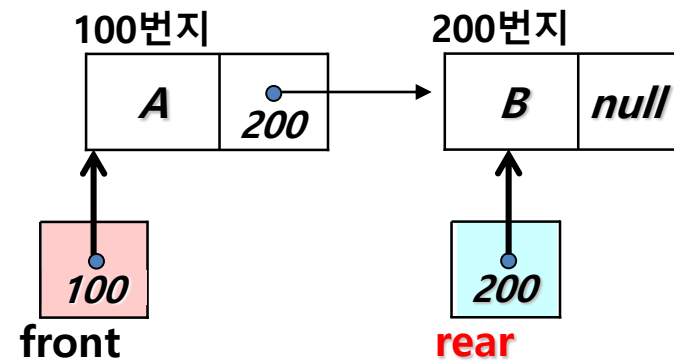
① 공백 큐 생성 : `createLinkedListQueue();`



② 원소 A 삽입 : `enqueue(LQ, A);`

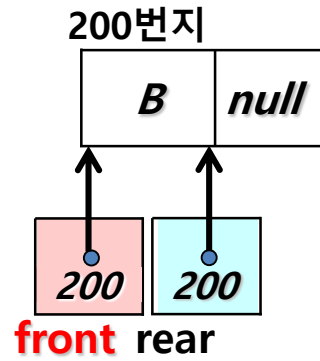


③ 원소 B 삽입 : `enqueue(LQ, B);`

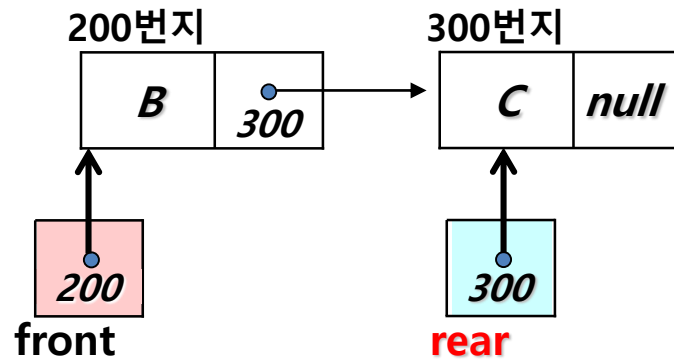


연결 큐

④ 원소 삭제 : `deQueue(LQ);`

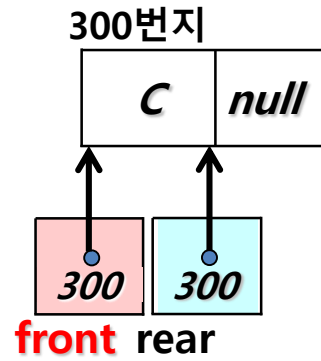


⑤ 원소 C 삽입 : `enQueue(LQ, C);`

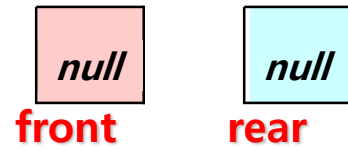


연결 큐

⑥ 원소 삭제 : `deQueue(LQ);`

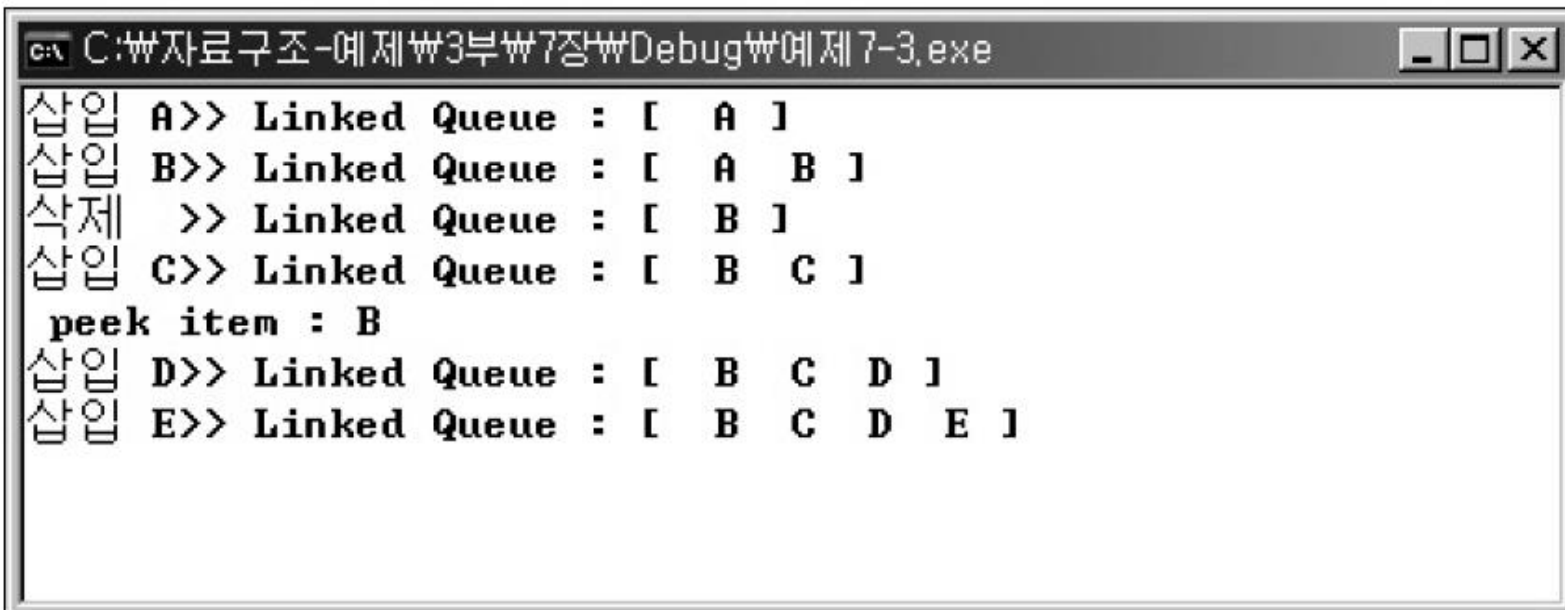


⑦ 원소 삭제 : `deQueue(LQ);`



연결 큐

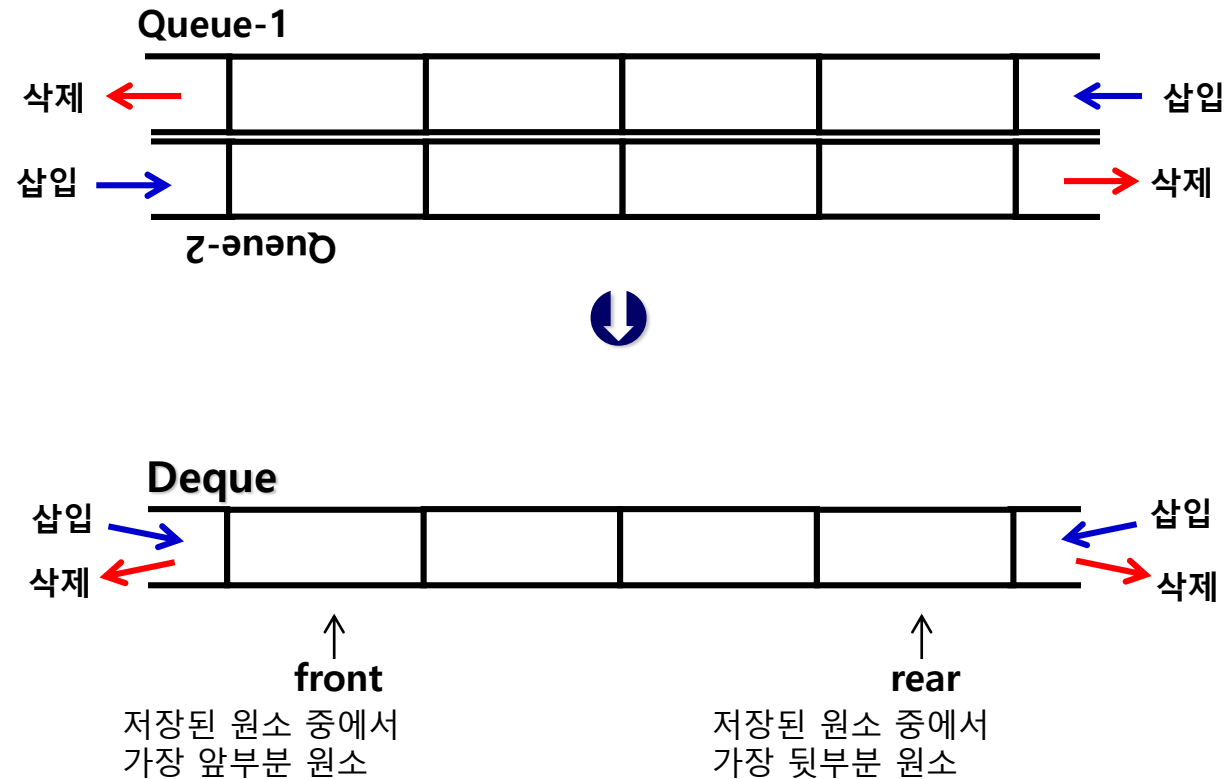
- 실습 III



```
c:\자료구조-예제\3부\7장\Debug\예제 7-3.exe
삽입 A>> Linked Queue : [ A ]
삽입 B>> Linked Queue : [ A B ]
삭제  >> Linked Queue : [ B ]
삽입 C>> Linked Queue : [ B C ]
peek item : B
삽입 D>> Linked Queue : [ B C D ]
삽입 E>> Linked Queue : [ B C D E ]
```

연결 큐

- 덱(Deque, double-ended queue)
 - 큐 2개를 반대로 붙여서 만든 자료구조



Queue

- 추상 자료형
deque

ADT deque

데이터 : 0개 이상의 원소를 가진 유한 순서 리스트

연산 :

$DQ \in \text{deque}$; $\text{item} \in \text{Element}$;

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

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

Queue

- 추상 자료형
deque

```
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
```

```
getFront(DQ) ::= if (isEmpty(DQ)) then return null  
                  else { return the front item of the DQ };  
                  // 덱의 front에 있는 item(원소)을 반환하는 연산
```

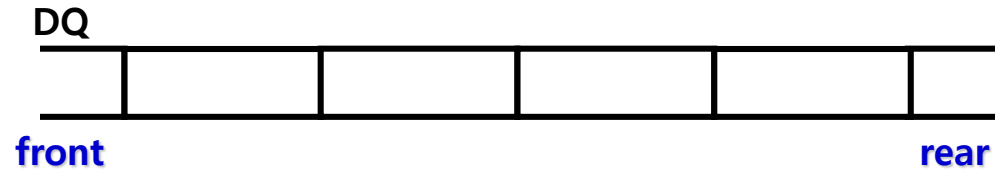
```
getRear(DQ) ::= if (isEmpty(DQ)) then return null  
                 else { return the rear item of the DQ };  
                 // 덱의 rear에 있는 item(원소)을 반환하는 연산
```

End deque

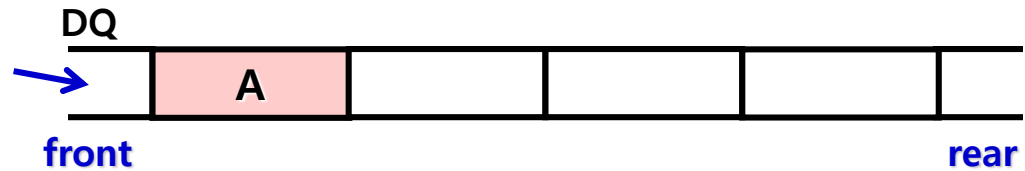
연결 큐

– 덱에서의 연산 과정

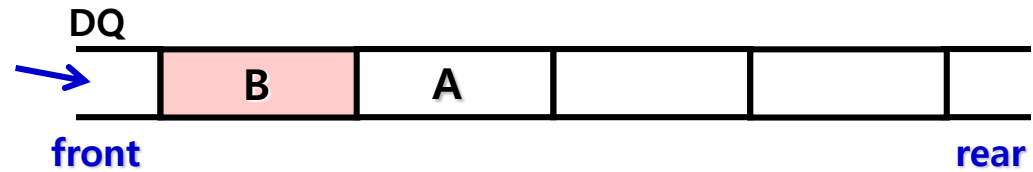
① createDeque();



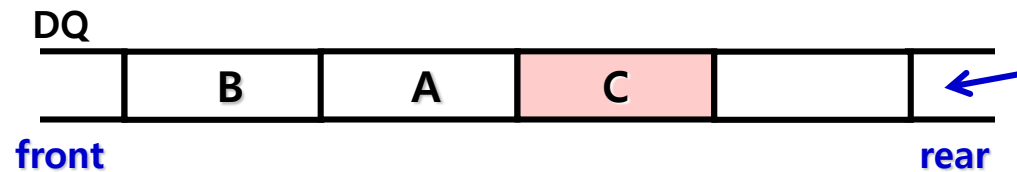
② insertFront(DQ, 'A');



③ insertFront(DQ, 'B');

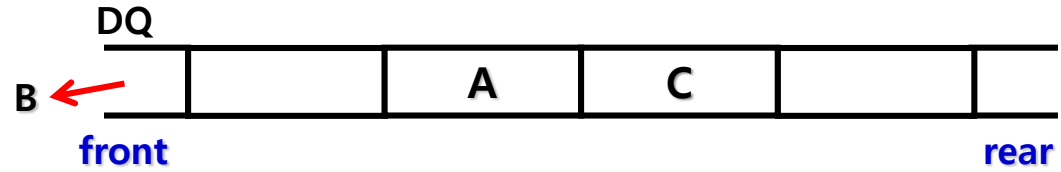


④ insertRear(DQ, 'C');

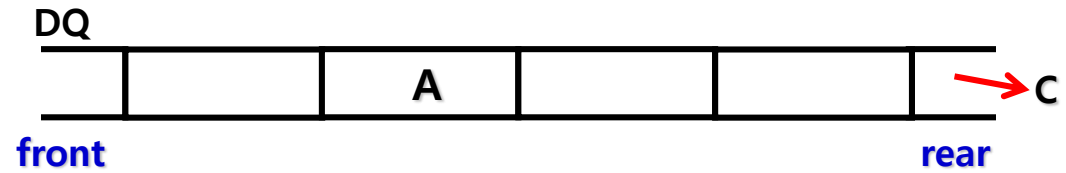


연결 큐

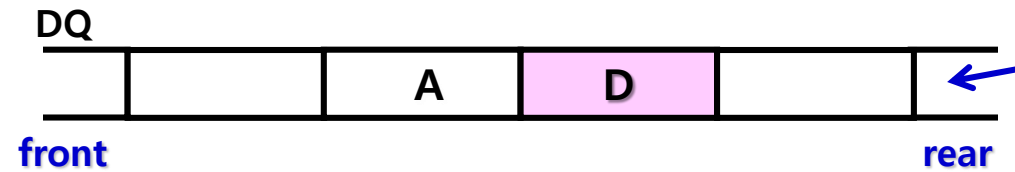
⑤ deleteFront(DQ);



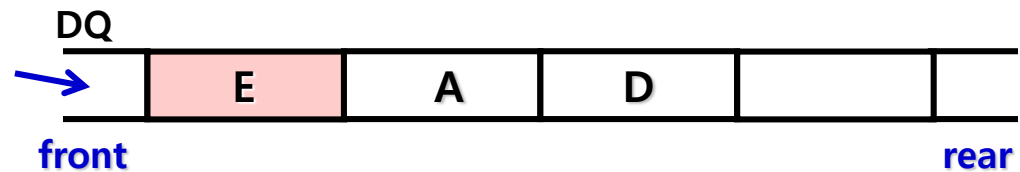
⑥ deleteRear(DQ);



⑦ insertRear(DQ, 'D');



⑧ insertFront(DQ, 'E');



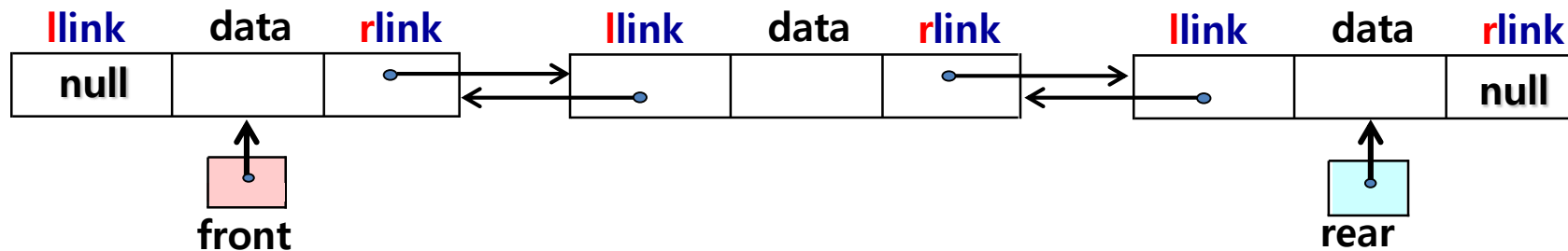
연결 큐

⑨ insertFront(DQ 'F');



- 덱의 구현

- 양쪽 끝에서 삽입/삭제 연산을 수행하면서 크기 변화와 저장된 원소의 순서 변화가 많으므로 순차 자료구조는 비효율적
- 양방향으로 연산이 가능한 이중 연결 리스트를 사용한다.



연결 큐

- 실습 IV

```
C:\자료구조-예제\3부\7장\Debug\예제 7-4.exe
front 삽입 A>> DeQue : [ A ]
front 삽입 B>> DeQue : [ B A ]
rear 삽입 C>> DeQue : [ B A C ]
front 삭제 >> DeQue : [ A C ]
rear 삭제 >> DeQue : [ A ]
rear 삽입 D>> DeQue : [ A D ]
front 삽입 E>> DeQue : [ E A D ]
front 삽입 F>> DeQue : [ F E A D ]
peek Front item : F
peek Rear item : D
```


큐 응용

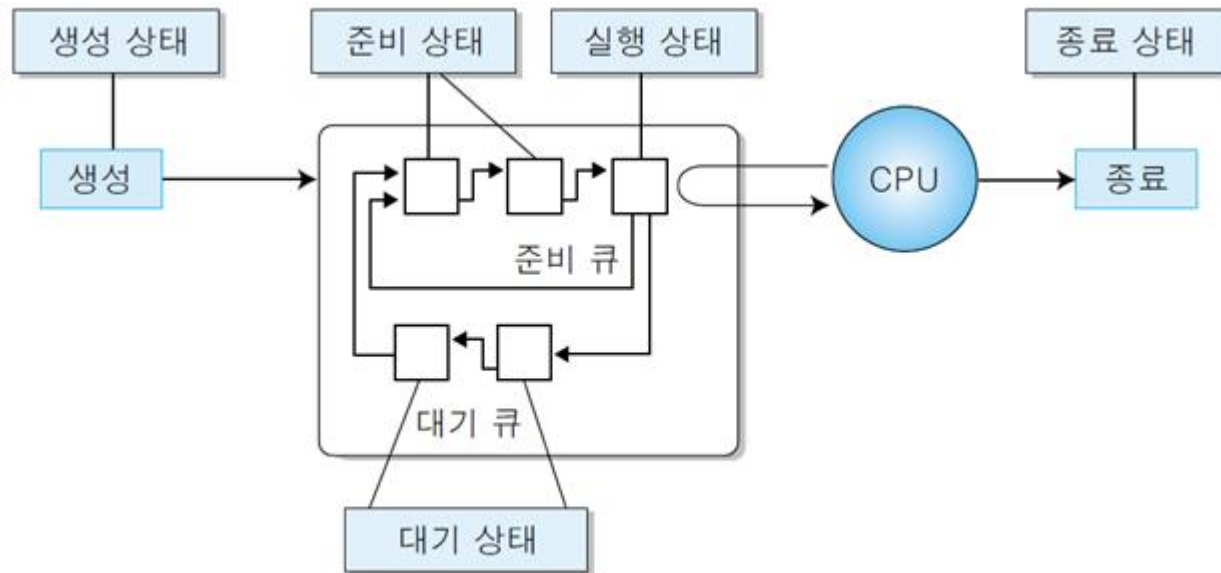
- 운영체제의 작업 큐

- 프린터 버퍼 큐

- CPU에서 프린터로 보낸 데이터 순서대로(선입선출) 프린터에서 출력하기 위해서 선입선출 구조의 큐 사용

- 스케줄링 큐

- CPU 사용을 요청한 프로세서들의 순서를 스케줄링하기 위해서 큐를 사용



큐 응용

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