

데이터 구조

큐(Queue)

Contents

❖ 학습목표

- 큐 자료구조의 개념을 스택과 비교하여 알아본다.
- 큐의 특징과 연산 방법을 알아본다.
- 순차 자료구조와 연결 자료구조를 이용해 큐를 구현해 본다.
- 큐를 확장한 자료구조인 데크의 특징과 연산 방법을 알아본다.
- 큐를 응용하는 방법을 알아본다.

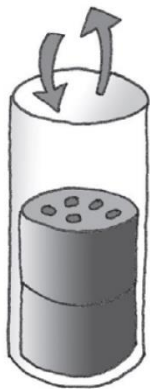
❖ 내용

- 01 큐의 이해
- 02 큐의 구현
- 03 데크
- 04 큐의 응용

1. 큐의 이해 : 큐의 개념과 구조

❖ 큐(Queue)

- 스택과 비슷한 삽입과 삭제의 위치가 제한되어있는 유한 순서 리스트
 - 큐는 뒤에서는 삽입만 하고, 앞에서는 삭제만 할 수 있는 구조
 - 삽입한 순서대로 원소가 나열되어 가장 먼저 삽입(**F**irst-**I**n)한 원소는 맨 앞에 있다가 가장 먼저 삭제(**F**irst-**O**ut)됨
- ☞ **선입선출 구조 (FIFO, First-In-First-Out)**



(a) 스택의 구조



(b) 큐의 구조

그림 6-1 스택과 큐의 구조 비교 예

1. 큐의 이해 : 큐의 개념과 구조

■ FIFO 구조의 예

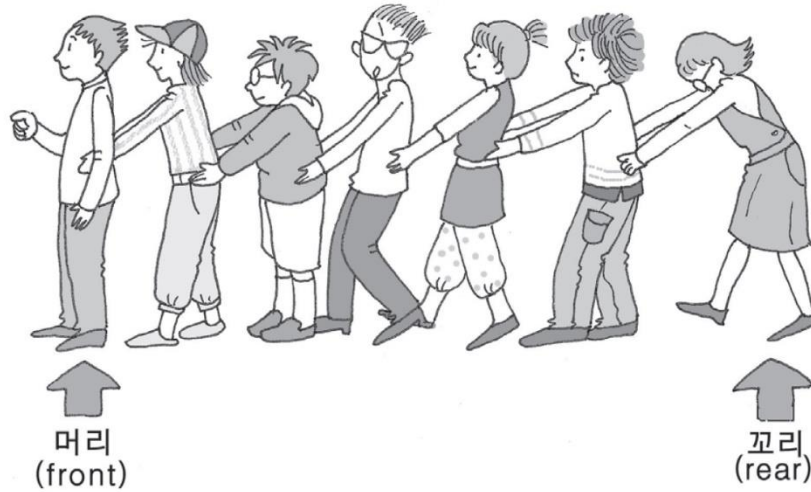


그림 6-2 FIFO 구조의 예 : 꼬리잡기 놀이의 머리와 꼬리

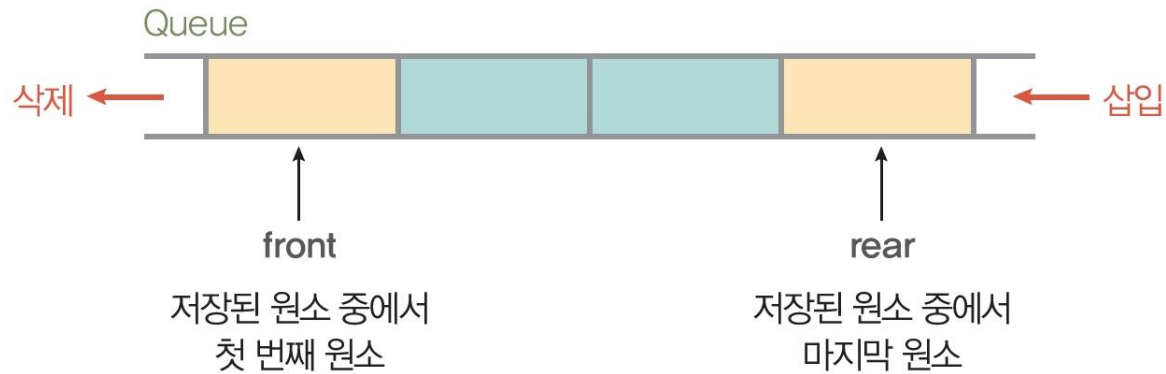


그림 6-3 큐의 FIFO 구조

1. 큐의 이해 : 큐의 개념과 구조

■ 큐의 연산

- 삽입 : **enQueue**
- 삭제 : **deQueue**

■ 스택과 큐의 연산 비교

표 6-1 스택과 큐에서의 삽입과 삭제 연산 비교

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

1. 큐의 이해 : 큐의 추상 자료형

ADT 6-1

큐의 추상 자료형

ADT Queue

데이터 : 0개 이상의 원소를 가진 유한 순서 리스트

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

// 공백 큐를 생성하는 연산

`createQueue()` ::= create an empty Q;

// 큐가 공백 상태인지 검사하는 연산

`isEmpty(Q)` ::= if (Q is empty) then return true
 else return false;

// 큐의 rear에 원소를 삽입하는 연산

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

// 큐의 front에 있는 원소를 삭제하는 연산

`dequeue(Q)` ::= if (`isEmpty(Q)`) then return error
 else { delete and return the front item Q };

// 큐의 front에 있는 원소를 반환하는 연산

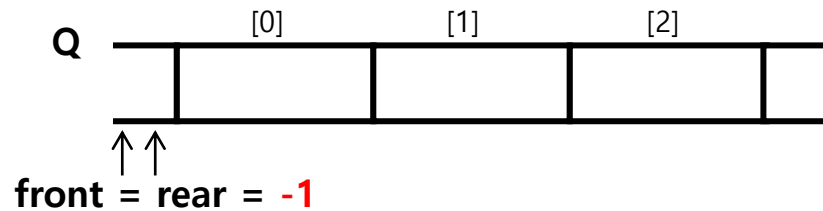
`peek(Q)` ::= if (`isEmpty(Q)`) then return error
 else { return the front item of the Q };

End Queue

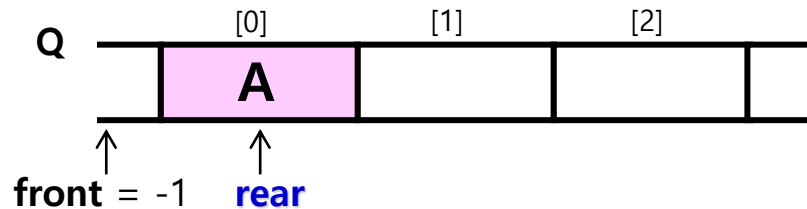
1. 큐의 이해

■ 큐의 연산 과정

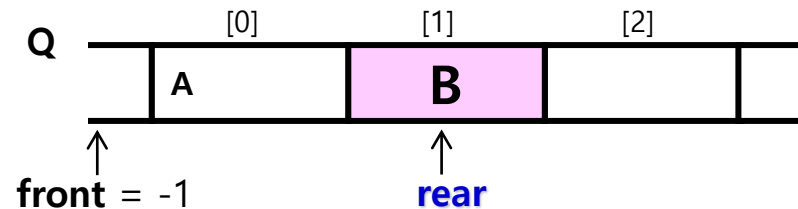
- ❶ 공백 큐 생성 : `createQueue();`



- ❷ 원소 A 삽입 : `enqueue(Q, A);`

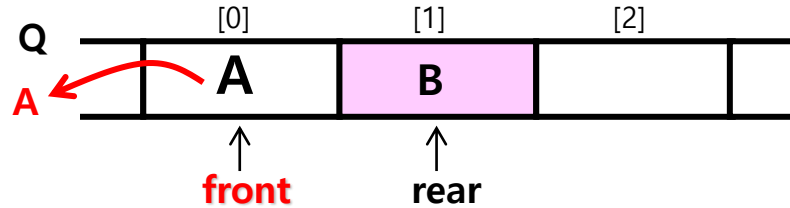


- ❸ 원소 B 삽입 : `enqueue(Q, B);`

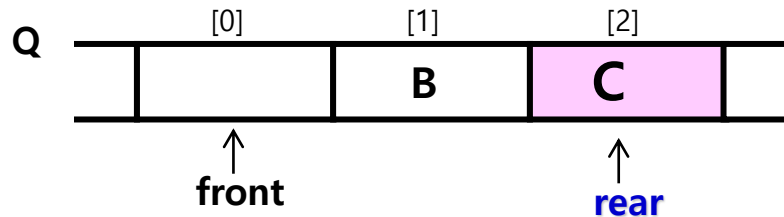


1. 큐의 이해

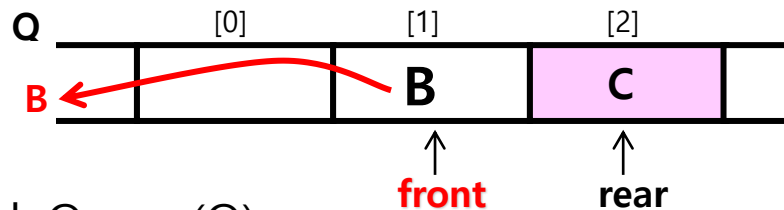
- ④ 원소 삭제 : `deQueue(Q);`



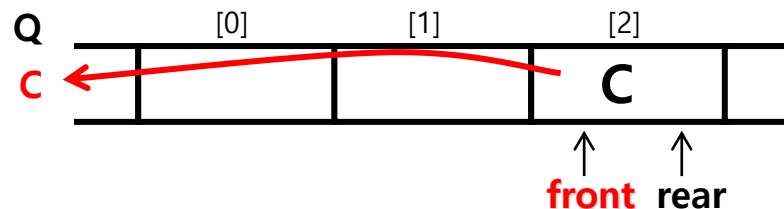
- ⑤ 원소 `C` 삽입 : `enqueue(Q, C);`



- ⑥ 원소 삭제 : `deQueue(Q);`



- ⑦ 원소 삭제 : `deQueue(Q);`



2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현

❖ 순차 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 : $\text{front} = \text{rear} = -1$
- 공백 상태 : $\text{front} = \text{rear}$
- 포화 상태 : $\text{rear} = n-1$ (n : 배열의 크기, $n-1$: 배열의 마지막 인덱스)

2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현

- 초기 공백 큐 생성 알고리즘
 - 크기가 n 인 1차원 배열 생성
 - front와 rear를 -1로 초기화

알고리즘 6-1 공백 순차 큐 생성

```
createQueue()  
  Q[n];  
  front  $\leftarrow$  -1;  
  rear  $\leftarrow$  -1;  
end createQueue()
```

2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현

- 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)

알고리즘 6-2 순차 큐의 공백 상태 검사

```
isQueueEmpty(Q)
  if (front == rear) then return true;
  else return false;
end isQueueEmpty()
```

알고리즘 6-3 순차 큐의 포화 상태 검사

```
isQueueFull(Q)
  if (rear == n - 1) then return true;
  else return false;
end isQueueFull()
```

2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현

■ 큐의 삽입 알고리즘

알고리즘 6-4 순차 큐의 원소 삽입

```
enqueue(Q, item)
  if (isQueueFull(Q)) then Queue_Full();    // 포화 상태이면 삽입 연산 중단
  else {
    ① rear ← rear + 1;
    ② Q[rear] ← item;
  }
end enqueue()
```

- 마지막 원소의 뒤에 삽입해야 하므로

- ① 마지막 원소의 인덱스를 저장한 **rear**의 값을 하나 증가시켜 삽입할 자리 준비
- ② 수정한 rear값에 해당하는 배열원소 Q[rear]에 item을 저장

2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현

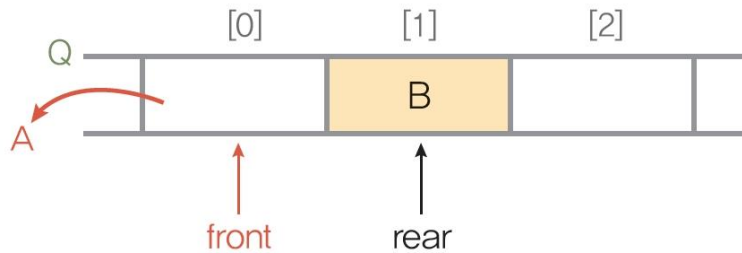
■ 큐의 삭제 알고리즘

알고리즘 6-5 순차 큐의 원소 삭제

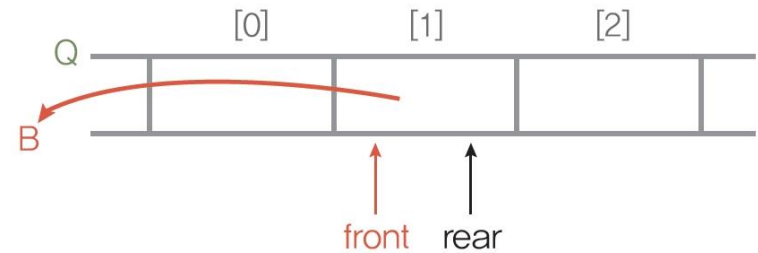
```
deQueue(Q)
  if (isEmpty(Q)) then Queue_Empty();      // 공백 상태이면 삭제 연산 중단
  else {
    ① front ← front + 1;
    ② return Q[front];
  }
end deQueue()
```

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

2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현



(a) 첫 번째 `deQueue()` 연산 후 상태



(b) 두 번째 `deQueue()` 연산 후 상태

그림 6-4 `deQueue()` 연산 후 상태

2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현

■ 큐의 검색 알고리즘

알고리즘 6-6 순차 큐의 원소 검색

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

- 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

2. 큐의 구현 : 순차 자료구조를 이용한 큐의 구현

- [예제 6-1] 순차 자료구조를 이용해 순차 큐 구현하기 : [교재 294p](#)
- 실행 결과

**** 순차 큐 연산 ****

삽입 A>> Queue : [A]

삽입 B>> Queue : [A B]

삽입 C>> Queue : [A B C] peek item : A

삭제 >> Queue : [B C] 삭제 데이터 : A

삭제 >> Queue : [C] 삭제 데이터 : B

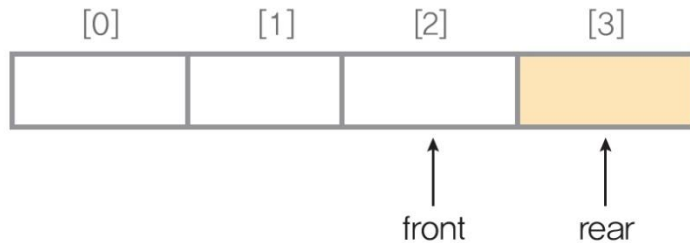
삭제 >> Queue : [] 삭제 데이터 : C

삽입 D>> Queue : [D]

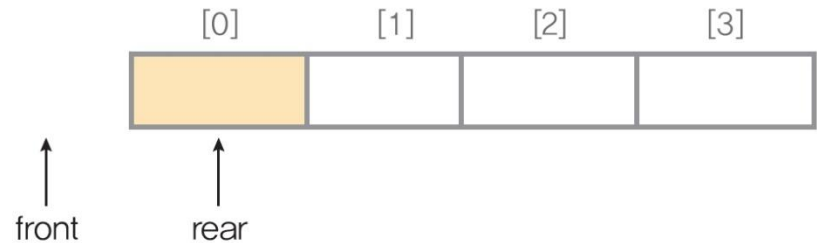
삽입 E>> Queue is full!
Queue : [D]

2. 큐의 구현 : 원형 큐의 구현

- 순차 큐의 잘못된 포화상태 인식
 - 큐에서 삽입과 삭제를 반복하면서 그림(a)와 같은 상태일 경우, 앞부분에 빈자리가 있지만 $rear=n-1$ 상태이므로 포화상태로 인식하고 더 이상의 삽입을 수행하지 않는다.
- 순차 큐의 잘못된 포화상태 인식의 해결 방법-1
 - 저장된 원소들을 배열의 앞부분으로 이동시키기
 - 순차자료에서의 이동 작업은 연산이 복잡하여 효율성이 떨어짐



(a) 포화 상태로 잘못 인식하는 경우



(b) 큐의 원소들을 앞으로 이동하여 해결

그림 6-5 순차 큐의 잘못된 포화 상태 문제와 해결 방법

2. 큐의 구현 : 원형 큐의 구현

- 순차 큐의 잘못된 포화상태 인식의 해결 방법-2
 - 1차원 배열을 사용하면서 논리적으로 배열의 처음과 끝이 연결되어 있다고 가정하고 사용 \Rightarrow **원형 큐**
 - 원형 큐의 논리적 구조

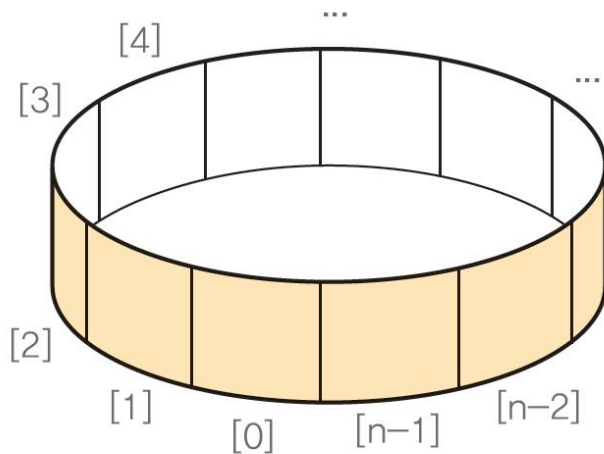


그림 6-6 원형 큐의 논리적 구조

2. 큐의 구현 : 원형 큐의 구현

■ 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 0번으로 이동하기 위해서 **나머지연산자 mod**를 사용
 - $3 \div 4 = 0 \dots 3$ (몫=0, 나머지=3)
 - $3 \bmod 4 = 3$

표 6-2 순차 큐와 원형 큐의 비교

종류	삽입 위치	삭제 위치
순차 큐	$\text{rear} = \text{rear} + 1$	$\text{front} = \text{front} + 1$
원형 큐	$\text{rear} = (\text{rear} + 1) \bmod n$	$\text{front} = (\text{front} + 1) \bmod n$

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

2. 큐의 구현 : 원형 큐의 구현

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

알고리즘 6-7 공백 원형 큐 생성

```
createQueue()  
  cQ[n];  
  front ← 0;  
  rear ← 0;  
end createQueue()
```

2. 큐의 구현 : 원형 큐의 구현

- 원형 큐의 공백상태 검사 알고리즘과 포화상태 검사 알고리즘

알고리즘 6-8 원형 큐의 공백 상태 검사

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

알고리즘 6-9 원형 큐의 포화 상태 검사

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

표 6-3 원형 큐의 상태에 따른 front와 rear의 관계

구분	조건
공백 상태	front == rear
포화 상태	(rear+1) mod n == front

2. 큐의 구현 : 원형 큐의 구현

■ 원형 큐의 삽입 알고리즘

- ① rear의 값을 조정하여 삽입할 자리를 준비 : $\text{rear} \leftarrow (\text{rear} + 1) \bmod n$;
- ② 준비한 자리 $\text{cQ}[\text{rear}]$ 에 원소 item 을 삽입

알고리즘 6-10 원형 큐의 원소 삽입

```
enCQueue(cQ, item)
  if (isCQueueFull(cQ)) then Queue_Full(); // 포화 상태이면 삽입 연산 중단
  else {
    ① rear  $\leftarrow$  (rear + 1) mod n;
    ② cQ[rear]  $\leftarrow$  item;
  }
end enCQueue()
```

2. 큐의 구현 : 원형 큐의 구현

■ 원형 큐의 삭제 알고리즘

- ① front의 값을 조정하여 삭제할 자리를 준비
- ② 준비한 자리에 있는 원소 cQ[front]를 삭제하여 반환

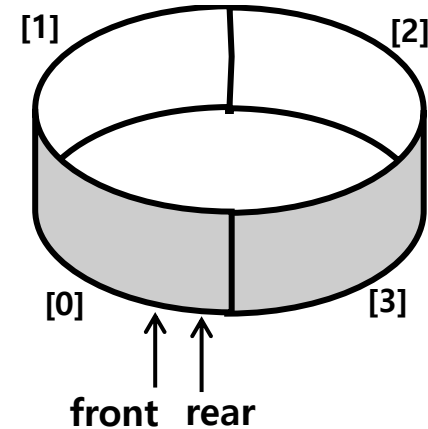
알고리즘 6-11 원형 큐의 원소 삭제

```
deCQueue(cQ)
  if (isCQueueEmpty(cQ)) then Queue_Empty(); // 공백 상태이면 삭제 연산 중단
  else {
    ① front ← (front + 1) mod n;
    ② return cQ[front];
  }
end deCQueue()
```

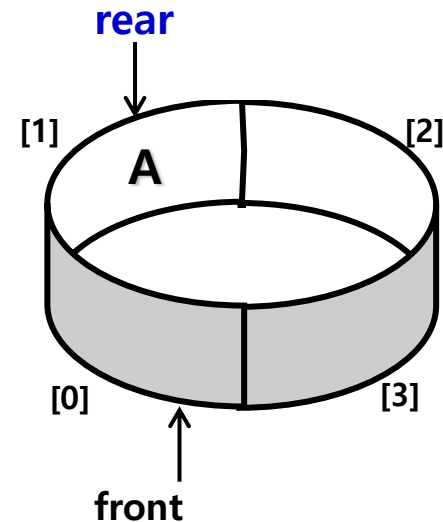
2. 큐의 구현 : 원형 큐의 구현

- 크기가 4인 원형 큐에서 큐를 생성하고 삽입·삭제하는 연산 과정

❶ 공백 원형 큐 생성 : `createQueue()`;

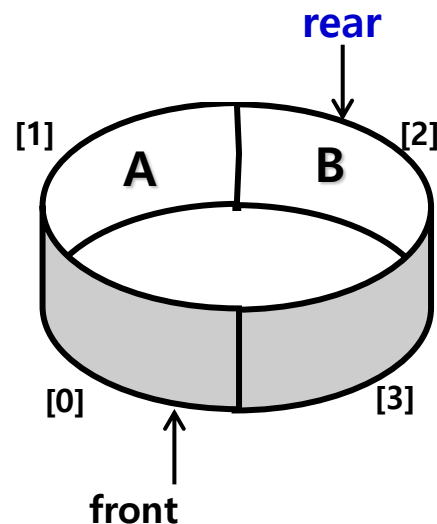


❷ 원소 A 삽입 : `enqueue(cQ, A)`;

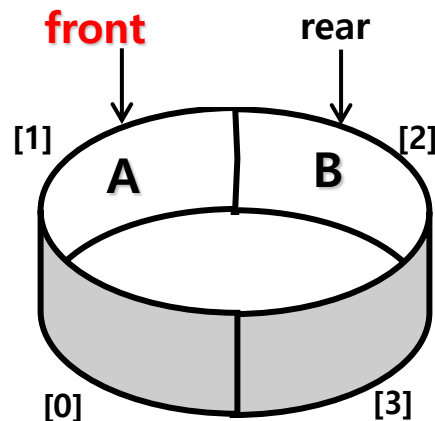


2. 큐의 구현 : 원형 큐의 구현

③ 원소 B 삽입 : `enqueue(cQ, B);`

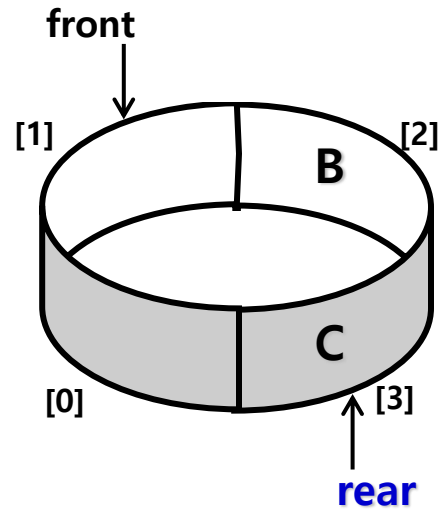


④ 원소 삭제 : `dequeue(cQ);`
(삭제 데이터 : A)

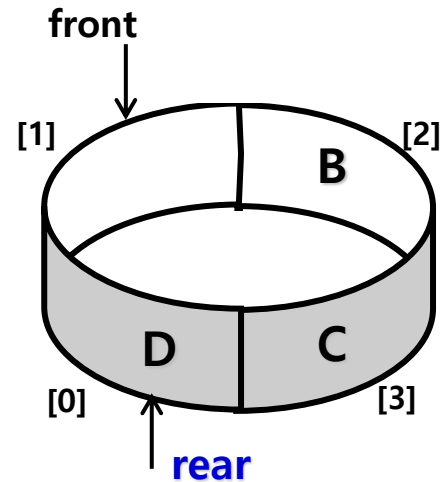


2. 큐의 구현 : 원형 큐의 구현

⑤ 원소 C 삽입 : `enqueue(cQ, C);`



⑥ 원소 D 삽입 : `enqueue(cQ, D);`



2. 큐의 구현 : 원형 큐의 구현

- [예제 6-2] 순차 자료구조를 이용해 원형 큐 구현하기 : [교재 301p](#)
- 실행 결과

***** 원형 큐 연산 *****

삽입 A>> Circular Queue : [A]

삽입 B>> Circular Queue : [A B]

삽입 C>> Circular Queue : [A B C] peek item : A

삭제 >> Circular Queue : [B C] 삭제 데이터 : A

삭제 >> Circular Queue : [C] 삭제 데이터 : B

삭제 >> Circular Queue : [] 삭제 데이터 : C

삽입 D>> Circular Queue : [D]

삽입 E>> Circular Queue : [D E]

2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

❖ 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

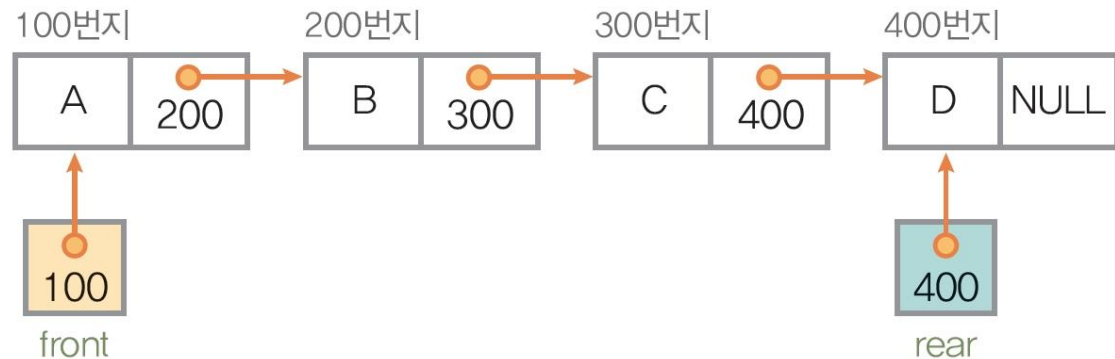


그림 6-7 연결 큐의 구조

2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

■ 공백 연결 큐 생성 알고리즘

- 초기화 : front = rear = null

알고리즘 6-12 공백 연결 큐 생성

```
createLinkedQueue()
  front ← NULL;
  rear ← NULL;
end createLinkedQueue()
```

■ 연결 큐의 공백 상태 검사 알고리즘

- 공백 상태 : front = rear = null

알고리즘 6-13 연결 큐의 공백 상태 검사

```
isLQEmpty(LQ)
  if (front == NULL) then return true;
  else return false;
end isLQEmpty()
```

2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

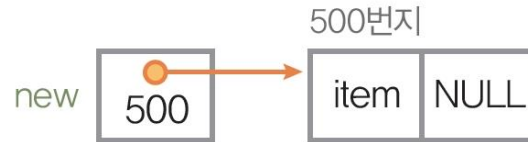
■ 연결 큐의 삽입 알고리즘

알고리즘 6-14 연결 큐의 원소 삽입

```
enLQueue(LQ, item)
    { new ← getNode();
    ① { new.data ← item;
        new.link ← NULL;
        { if (front == NULL) then {
        ② { rear ← new;
            front ← new;
            }
        }
        { else {
        ③ { rear.link ← new;
            rear ← new;
            }
        }
    }
end enLQueue()
```

2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

- ❶ 삽입할 새 노드를 생성하여 데이터 필드에 item을 저장. 삽입할 새 노드는 연결 큐의 마지막 노드가 되어야 하므로 링크 필드에 NULL을 저장

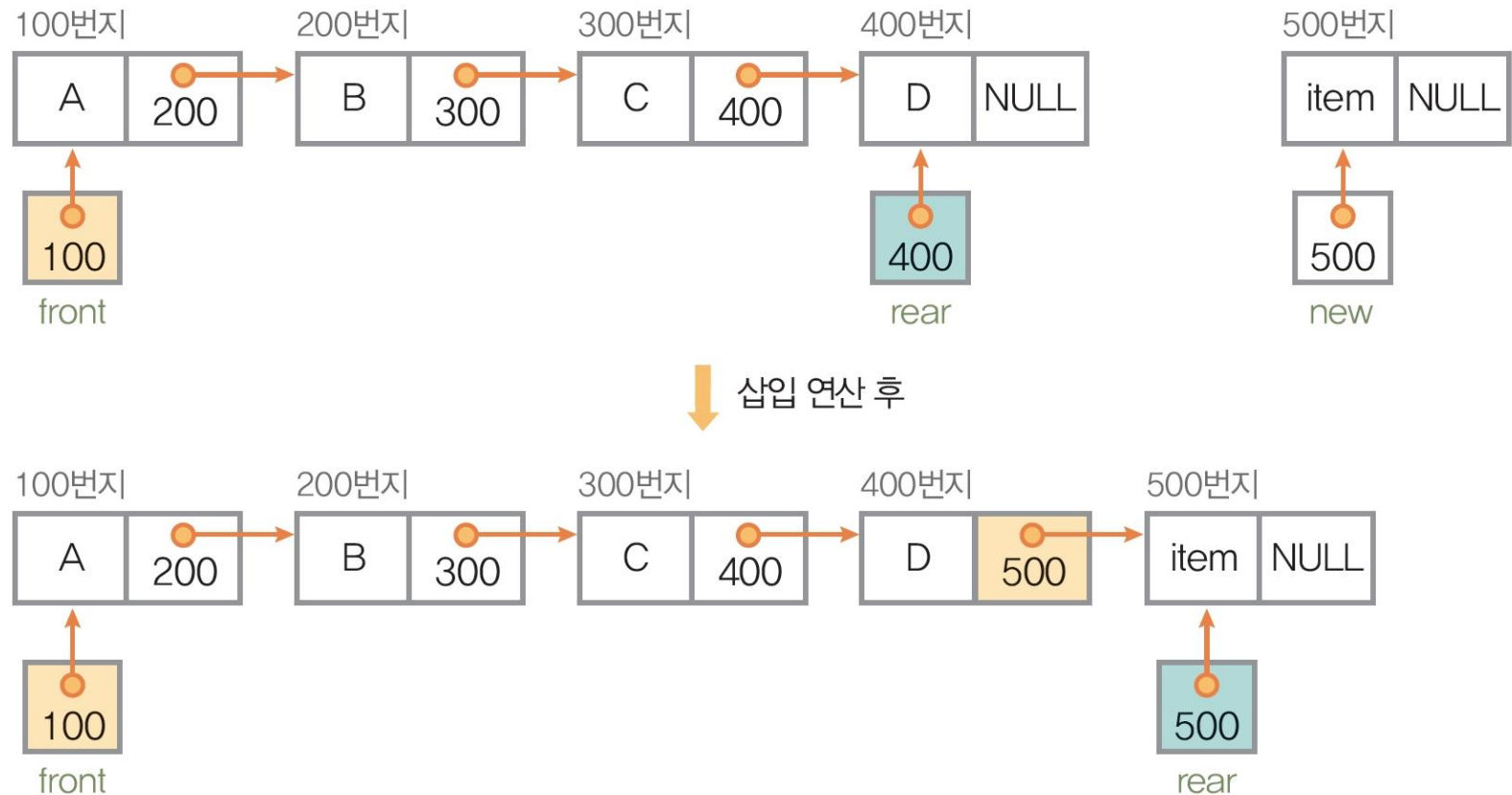


- ❷ 새 노드를 삽입하기 전에 연결 큐가 공백인지 아닌지를 검사. 연결 큐가 공백인 경우에는 삽입할 새 노드가 큐의 첫 번째 노드이자 마지막 노드이므로 포인터 front와 rear가 모두 새 노드를 가리키도록 설정



2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

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 설정



2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

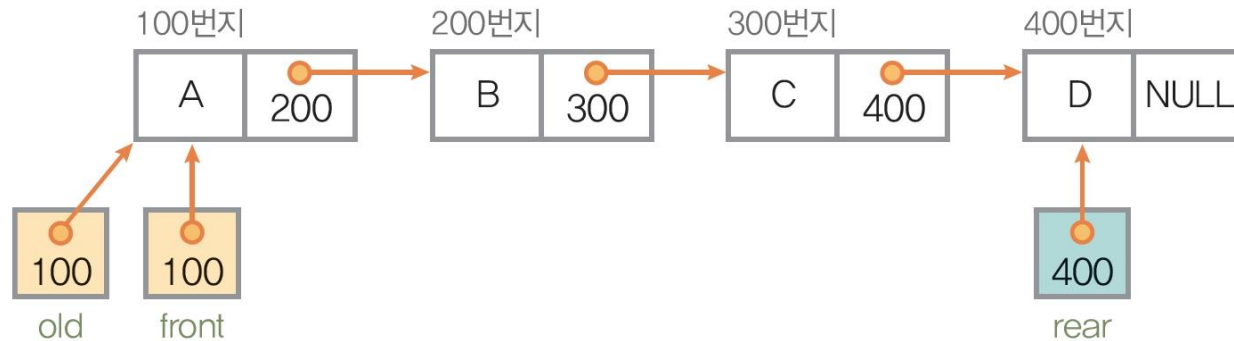
■ 연결 큐의 원소 삭제 알고리즘

알고리즘 6-15 연결 큐의 원소 삭제

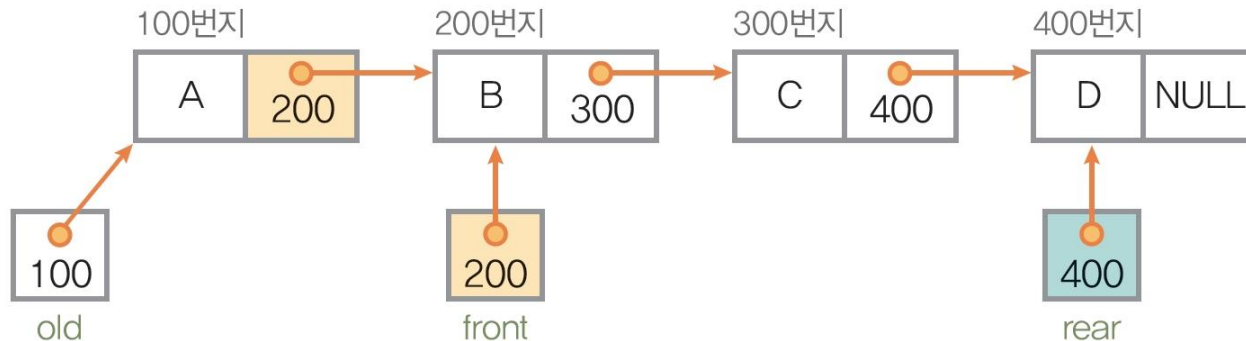
```
deLQueue(LQ)
  if (isLQEmpty(LQ)) then Queue_Empty();
  else {
    ❶ old ← front;
      item ← front.data;
    ❷ front ← front.link;
    ❸ if (isLQEmpty(LQ)) then rear ← NULL;
    ❹ returnNode(old);
      return item;
  }
end deLQueue()
```

2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

- ❶ 삭제 연산에서 삭제할 노드는 큐의 첫 번째 노드로, 포인터 front가 가리키고 있는 노드. Front가 가리키는 노드를 포인터 old가 가리키게 하여 삭제할 노드로 지정

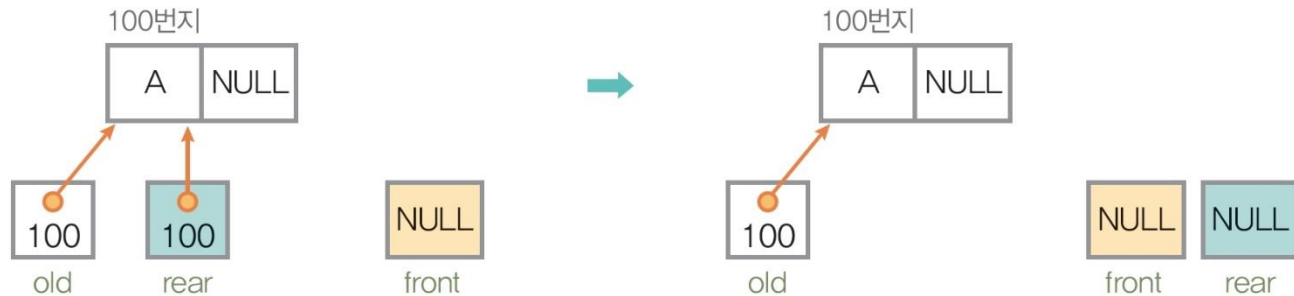


- ❷ 삭제 연산 후에는 현재 front 노드 다음 노드(front.link)가 front 노드가 되어야 하므로 포인터 front를 재설정

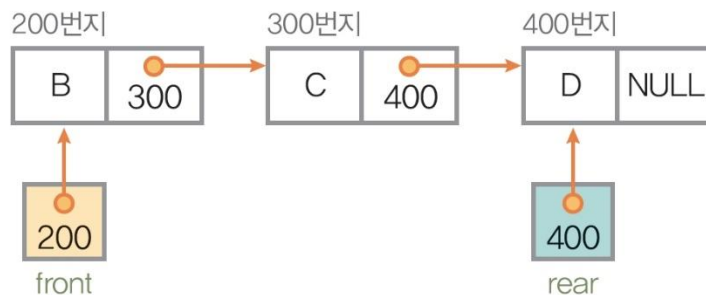


2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

- ③ 현재 큐에 노드가 하나뿐이어서 재설정된 front가 NULL이 되는 경우에는 삭제 연산 후에 공백 큐가 되므로 포인터 rear를 NULL로 설정



- ④ 포인터 old가 가리키고 있는 노드를 삭제하여 메모리 공간을 시스템에 반환(returnNode())



2. 큐의 구현 : 연결 자료구조를 이용한 큐의 구현

- 연결 큐의 원소 검색 알고리즘
 - 연결 큐의 첫 번째 노드, 즉 front 노드의 데이터 필드 값을 반환

알고리즘 6-16 연결 큐의 원소 검색

```
peekLQ(LQ)
  if (isLQEmpty(LQ)) then Queue_Empty()
  else return (front.data);
end peekLQ()
```

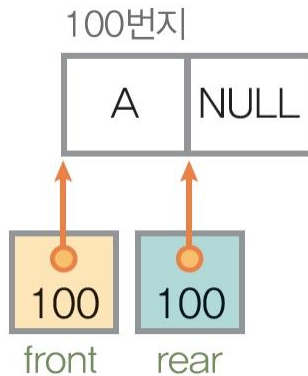
2. 큐의 구현 : 연결자료구조를 이용한 큐의 구현

■ 연결 큐에서의 연산 과정

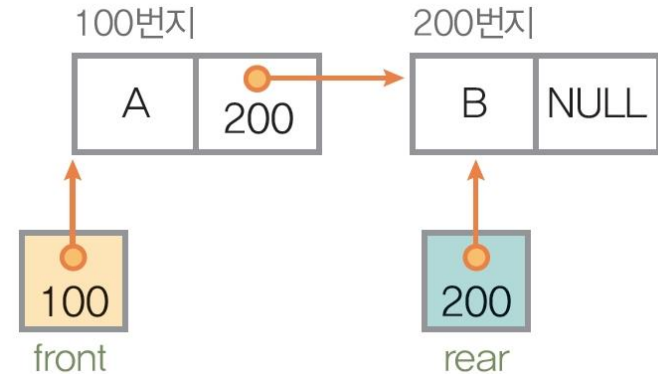
❶ 공백 연결 큐 생성 : `createLinkedListQueue()`;



❷ 원소 A 삽입 : `enLQueue(LQ, A)`;

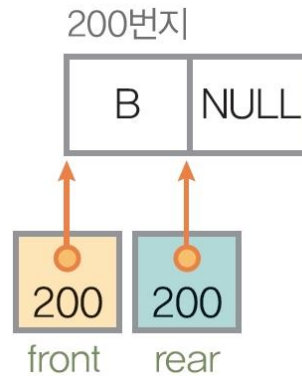


❸ 원소 B 삽입 : `enLQueue(LQ, B)`;

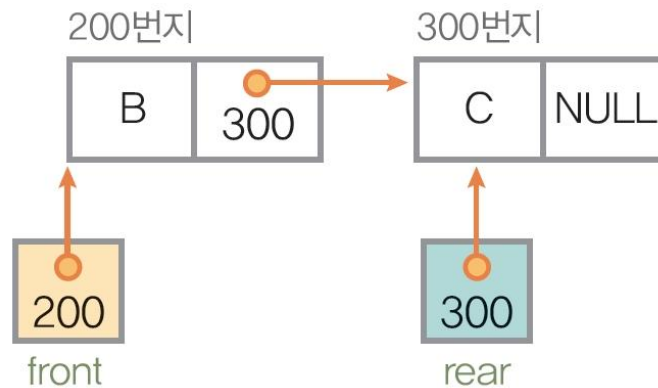


2. 큐의 구현 : 연결자료구조를 이용한 큐의 구현

④ 원소 삭제 : deLQueue(LQ);

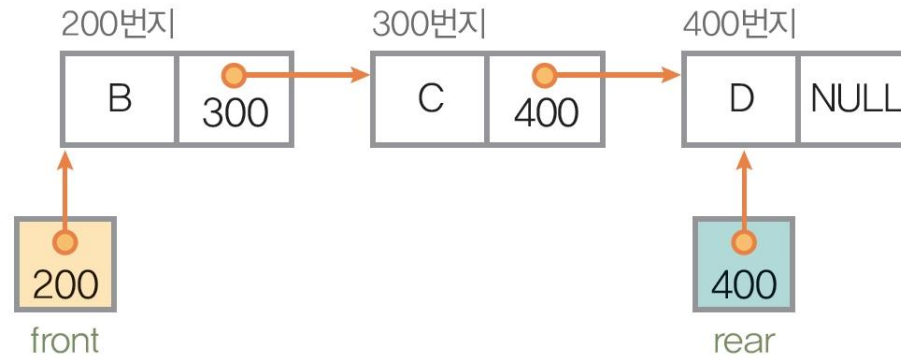


⑤ 원소 C 삽입 : enLQueue(LQ, C);

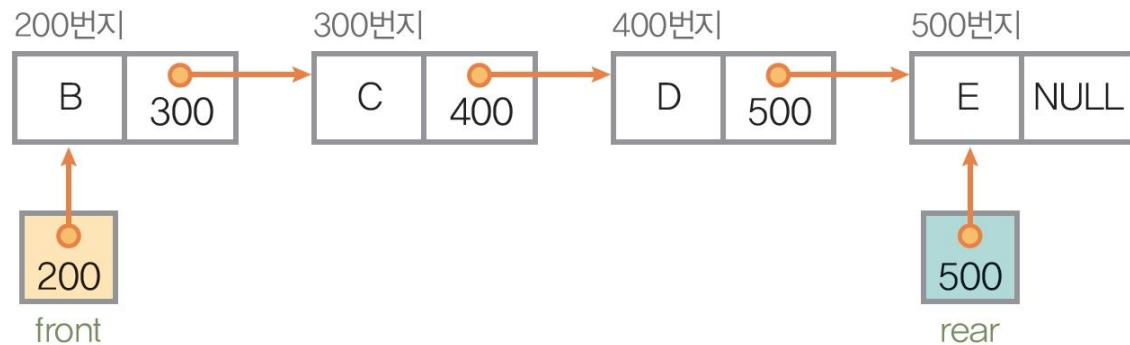


2. 큐의 구현 : 연결자료구조를 이용한 큐의 구현

⑥ 원소 D 삽입 : enLQueue(LQ, D);



⑦ 원소 E 삽입 : enLQueue(LQ, E);



2. 큐의 구현 : 연결자료구조를 이용한 큐의 구현

- [예제 6-3] 연결 자료구조를 이용해 연결 큐 구현하기 : [교재 309p](#)
- 실행 결과

***** 연결 큐 연산 *****

삽입 A>> Linked Queue : [A]

삽입 B>> Linked Queue : [A B]

삽입 C>> Linked Queue : [A B C] peek item : A

삭제 >> Linked Queue : [B C] 삭제 데이터 : A

삭제 >> Linked Queue : [C] 삭제 데이터 : B

삭제 >> Linked Queue : [] 삭제 데이터 : C

삽입 D>> Linked Queue : [D]

삽입 E>> Linked Queue : [D E]

3. 데크

❖ 데크 Deque : double-ended queue

- 큐 두 개 중 하나를 좌우로 뒤집어서 붙인 구조, 큐의 양쪽 끝에서 삽입 연산과 삭제 연산을 수행할 수 있도록 확장한 자료구조

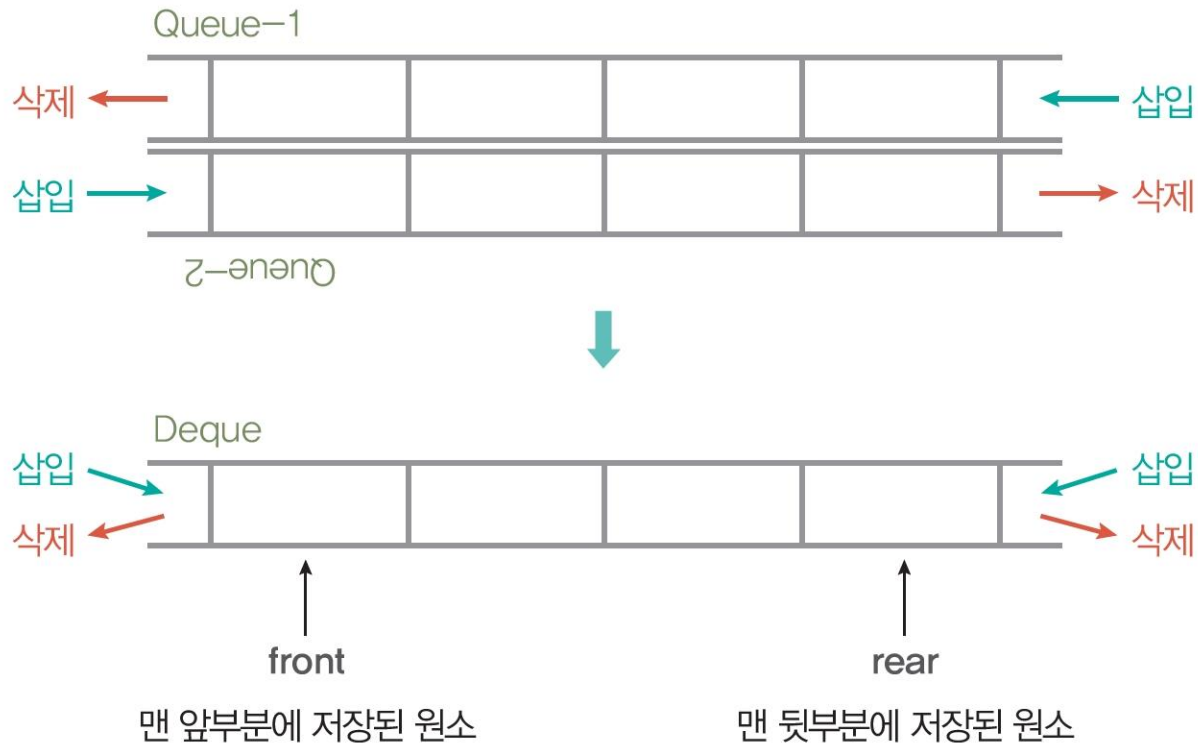


그림 6-8 데크의 구조

3. 데크

ADT 6-2 데크의 추상 자료형

ADT deque

데이터 : 0개 이상의 원소를 가진 유한 순서 리스트

연산 :

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

// 공백 데크를 생성하는 연산

$\text{createDeque}() ::= \text{create an empty DQ};$

// 데크가 공백 상태인지 검사하는 연산

$\text{isEmpty}(DQ) ::= \text{if } (DQ \text{ is empty}) \text{ then return true}$
 $\text{else return false};$

// 데크의 front 앞에 item(원소)을 삽입하는 연산

$\text{insertFront}(DQ, \text{item}) ::= \text{insert item at the front of DQ};$

// 데크의 rear 뒤에 item(원소)을 삽입하는 연산

$\text{insertRear}(DQ, \text{item}) ::= \text{insert item at the rear of DQ};$

// 데크의 front에 있는 item(원소)을 삭제하는 연산

$\text{deleteFront}(DQ) ::= \text{if } (\text{isEmpty}(DQ)) \text{ then return NULL}$
 $\text{else } \{ \text{delete and return the front item of DQ} \};$

// 데크의 rear에 있는 item(원소)을 삭제하는 연산

$\text{deleteRear}(DQ) ::= \text{if } (\text{isEmpty}(DQ)) \text{ then return NULL}$
 $\text{else } \{ \text{delete and return the rear item of DQ} \};$

// 데크의 front에 있는 item(원소)을 반환하는 연산

$\text{getFront}(DQ) ::= \text{if } (\text{isEmpty}(DQ)) \text{ then return NULL}$
 $\text{else } \{ \text{return the front item of the DQ} \};$

// 데크의 rear에 있는 item(원소)을 반환하는 연산

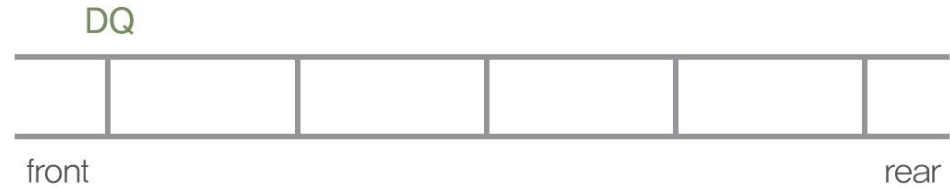
$\text{getRear}(DQ) ::= \text{if } (\text{isEmpty}(DQ)) \text{ then return NULL}$
 $\text{else } \{ \text{return the rear item of the DQ} \};$

End deque

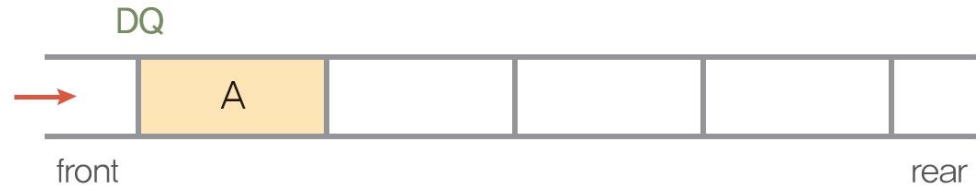
3. 덱

■ 덱의 연산 과정

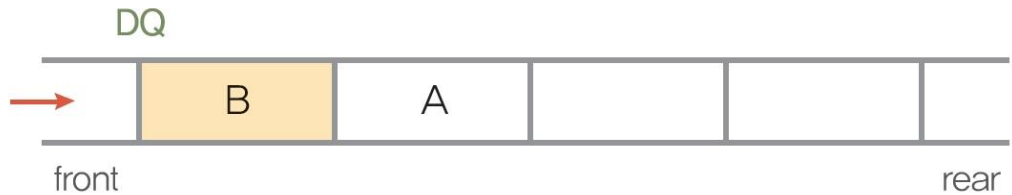
❶ createDeque();



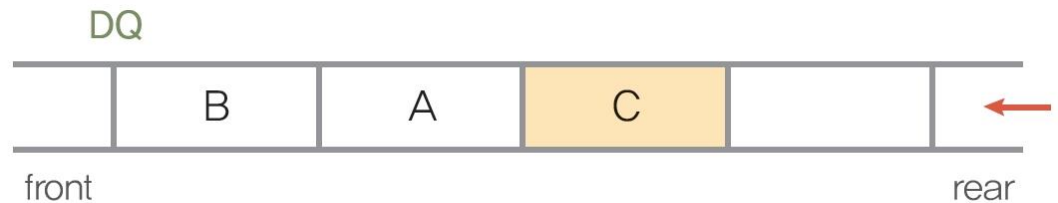
❷ insertFront(DQ, 'A');



❸ insertFront(DQ, 'B');

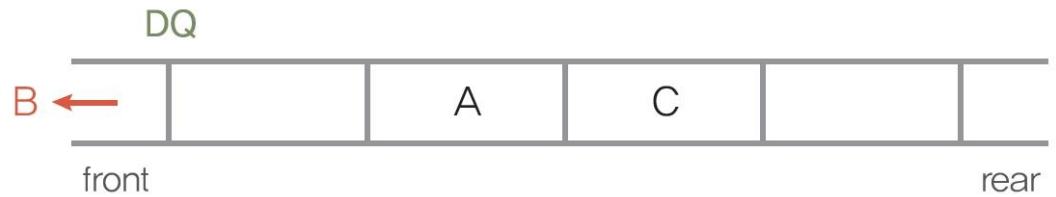


❹ insertRear(DQ, 'C');

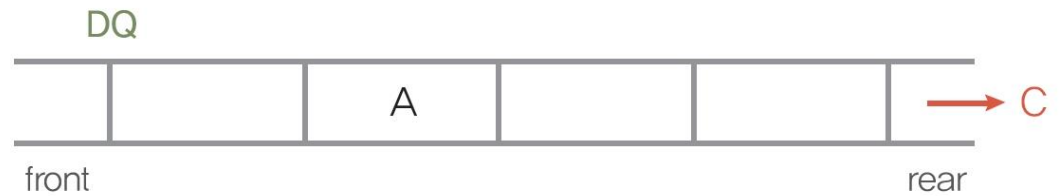


3. 데크

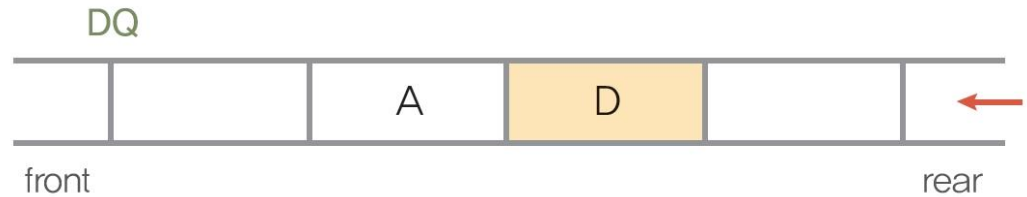
⑤ deleteFront(DQ);



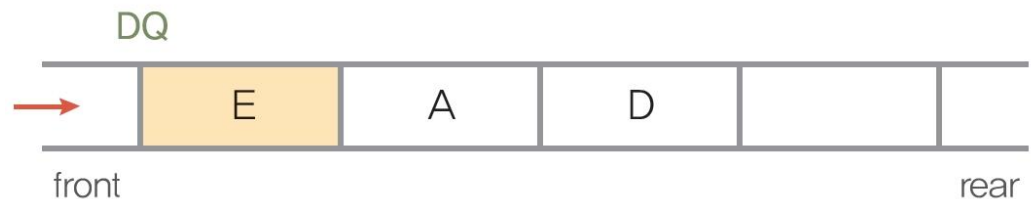
⑥ deleteRear(DQ);



⑦ insertRear(DQ, 'D');



⑧ insertFront(DQ, 'E');



3. 데크

⑨ insertFront(DQ, 'F');



■ 데크의 구현

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

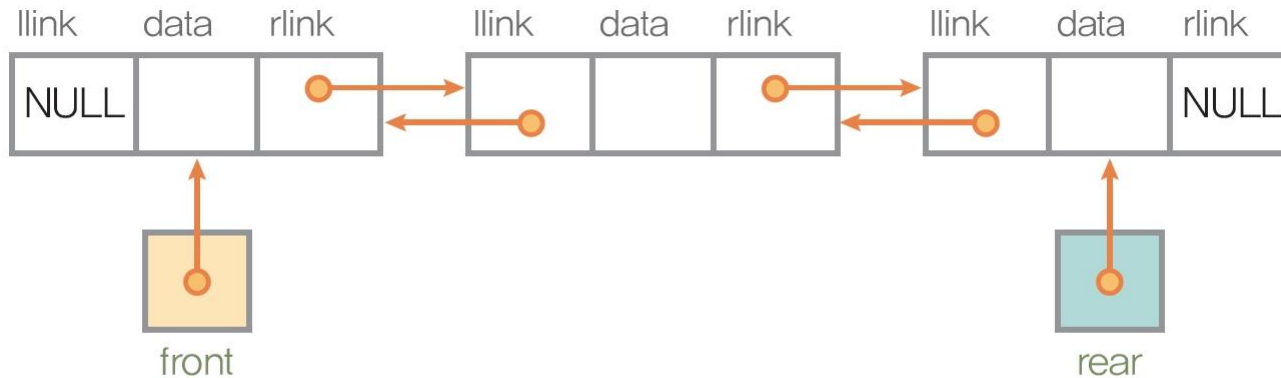


그림 6-9 데크의 이중 연결 리스트 구조

3. 데크

- [예제 6-4] 이중 연결 리스트를 이용해 데크 구현하기 : [교재 316p](#)
- 실행 결과

**** 데크 연산 ****

```
front 삽입 A>> DeQue : [ A ]  
front 삽입 B>> DeQue : [ B A ]  
rear 삽입 C>> DeQue : [ B A C ]  
front 삭제 >> DeQue : [ A C ]      삭제 데이터 : B  
rear 삭제 >> DeQue : [ A ]        삭제 데이터 : C  
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
```

4. 큐의 응용 : 운영체제의 작업 큐

❖ 운영체제의 작업 큐

■ 프린터 버퍼 큐(Printer Buffer Queue)

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

■ 스케줄링 큐(Scheduling Queue)

- CPU 사용을 요청한 프로세서들의 순서를 스케줄링 하기 위해서 큐를 사용

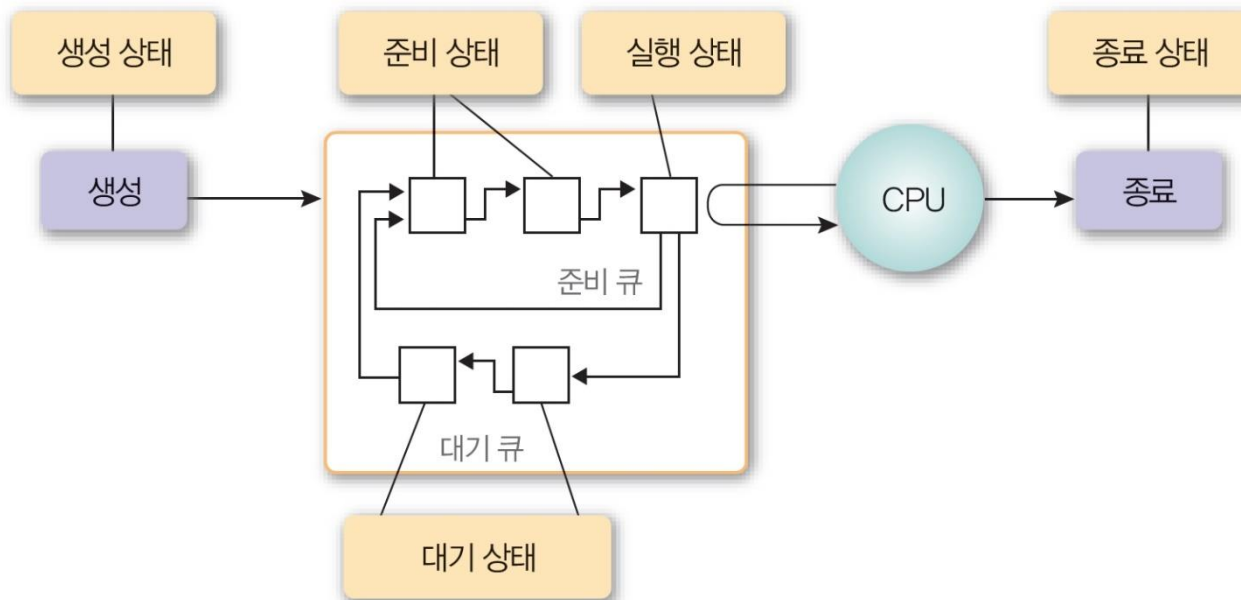


그림 6-10 프로세스 스케줄링 큐

4. 큐의 응용 : 시뮬레이션에서의 큐잉 시스템

❖ 시뮬레이션에서의 큐잉 시스템

- 시뮬레이션을 위한 수학적 모델링에서 대기행렬과 대기시간 등을 모델링 하기 위해서 큐잉 이론(Queueing theory) 사용