9주차 2차시 큐의 구현1

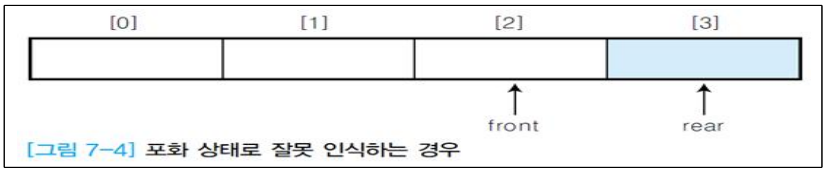
【학습목표】

1. 연결 표현 방법을 이용한 큐를 구현할 수 있다.
2. 원형 큐의 구현을 설명할 수 있다.

학습내용1 : 원형 큐의 이해

1. 선형 큐의 잘못된 포화상태 인식

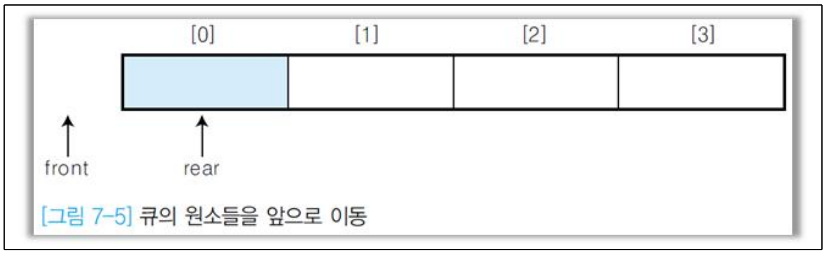
- 큐에서 삽입과 삭제를 반복하면서 아래와 같은 상태일 경우, 앞부분에 빈자리가 있지만 rear=n-1 상태이므로 포화상태로 인식하고 더 이상의 삽입을 수행하지 않는다



[그림 7-4] 포화 상태로 잘못 인식하는 경우

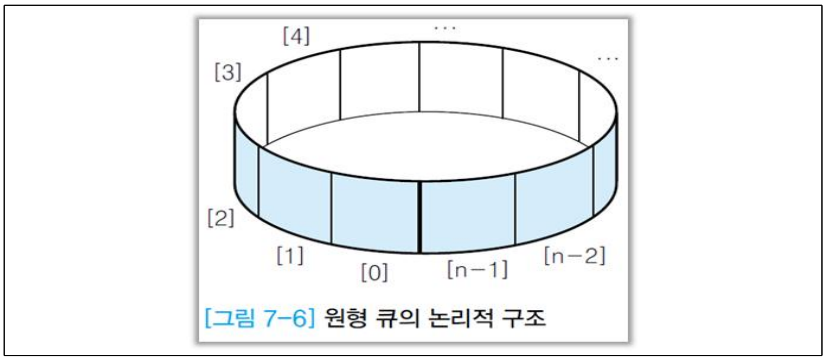
2. 선형 큐의 잘못된 포화상태 인식의 해결 방법

① 저장된 원소들을 배열의 앞부분으로 이동시키기
- 순차자료에서의 이동 작업은 연산이 복잡하여 효율성이 떨어짐



[그림 7-5] 큐의 원소들을 앞으로 이동

② 1차원 배열을 사용하면서 논리적으로 배열의 처음과 끝이 연결되어 있다고 가정하고 사용 ⇒ 원형 큐
- 원형 큐의 논리적 구조



[그림 7-6] 원형 큐의 논리적 구조

학습내용2 : 원형 큐의 구현

1. 원형 큐의 구조

- 초기 공백 상태 : front = rear = 0
- front와 rear의 위치가 배열의 마지막 인덱스 n-1에서 논리적인 다음 자리인 인덱스 0번으로 이동하기 위해서 나머지연산자 mod를 사용
- 공백 상태와 포화 상태 구분을 쉽게 하기 위해서 front가 있는 자리는 사용하지 않고 항상 빈자리로 둔다.

	삽입위치	삭제위치
선형큐	rear = rear + 1	front = front + 1
원형큐	rear = (rear+1) **mod** n	front = (front+1) + **mod** n

2. 초기 공백 원형 큐 생성 알고리즘

- 크기가 n인 1차원 배열 생성
- front와 rear을 0으로 초기화

알고리즘 7-6 초기 공백 원형 큐의 생성 알고리즘

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

3. 원형 큐의 공백 상태 검사 알고리즘과 포화상태 알고리즘

- 공백 상태 : front = rear
- 포화 상태 : 삽입할 rear의 다음 위치 = front의 현재 위치
    - (rear+1) mod n = front

4. 원형 큐의 삽입 알고리즘

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

5. 원형 큐의 삭제 알고리즘

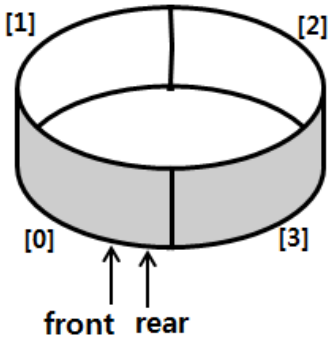
- front의 값을 조정하여 삭제할 자리를 준비
- 준비한 자리에 있는 원소 cQ[front]를 삭제하여 반환

알고리즘 7-7 원형 큐의 공백 상태와 포화 상태 검사 알고리즘

```
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()
```

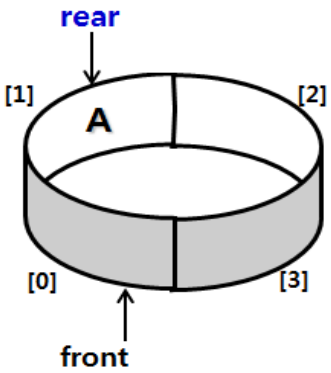
6. 원형 큐에서의 연산 과정

① createQueue();



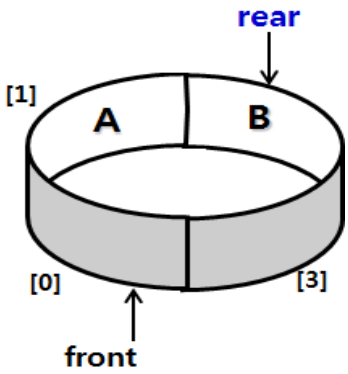
front ← 0;
rear ← 0;

② enQueue(cQ, A);



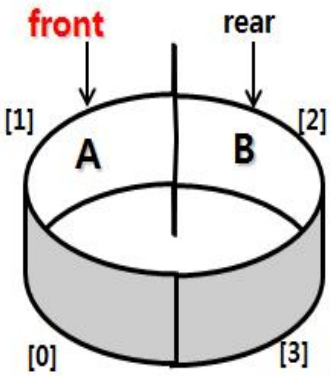
rear ← (0+1) mod 4;
cQ[1] ← A

③ enQueue(cQ, B);



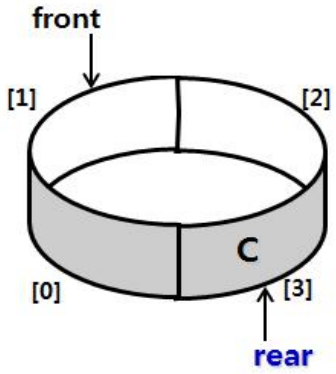
rear ← (1+1) mod 4;
cQ[2] ← B

④ deQueue(cQ);



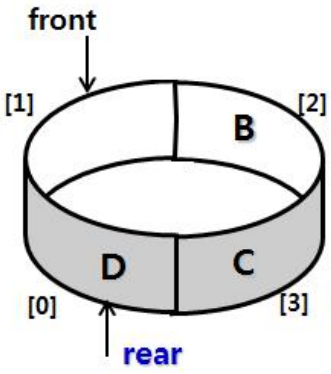
front ← (0+1) mod 4;
retrun cQ[1];

⑤ enQueue(cQ, C);



rear ← (2+1) mod 4;
cQ[3] ← C

⑥ enQueue(cQ, D);



rear ← (3+1) mod 4'
cQ[4] ← D

학습내용3 : 연결 큐의 이해와 구현

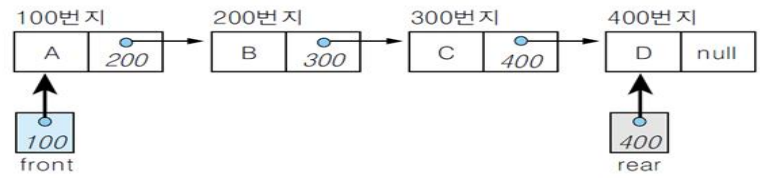
1. 연결 큐

* 순차 자료구조를 이용한 큐의 문제점
- 사용 크기가 제한됨
- 원소가 없는 경우에도 고정된 크기를 유지하고 있어 메모리 낭비

* 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

* 초기 상태와 공백 상태 : frotn = rear = null

* 연결 큐의 구조



[그림 7-8] 연결 큐의 구조

* 초기 공백 연결 큐 생성 알고리즘
- 초기화 : front = rear = null

알고리즘 7-10 초기 공백 연결 큐 생성 알고리즘

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

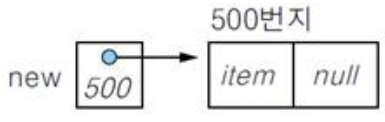
* 공백 연결 큐 검사 알고리즘
- 공백 상태 : front = rear = null

알고리즘 7-11 공백 연결 큐 검사 알고리즘

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

* 연결 큐의 삽입 알고리즘

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

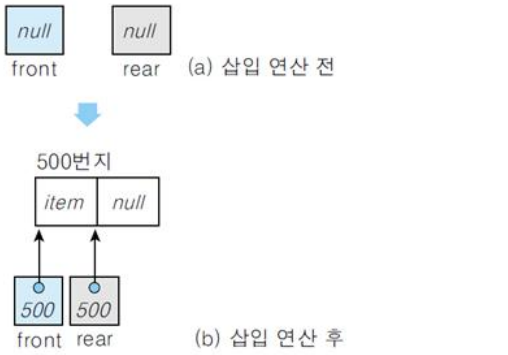


[그림 7-9] 연결 큐의 삽입 연산 과정 : 알고리즘 설명 ❶

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

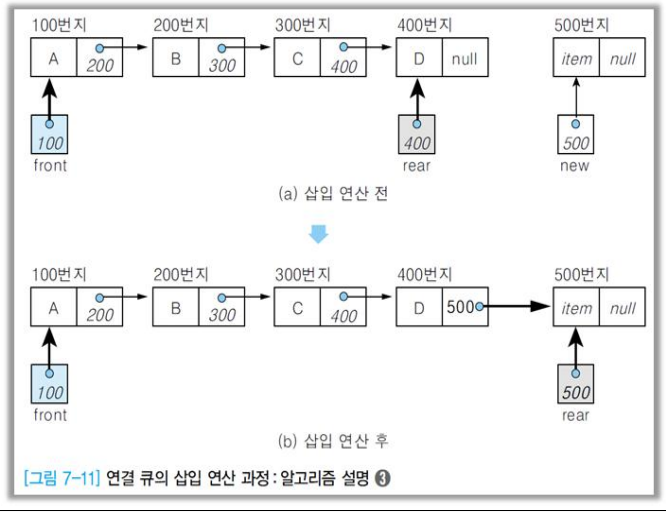
알고리즘 7-12 연결 큐의 삽입 알고리즘

```
enQueue(LQ, item)
    new ← getNode();
    new.data ← item;                    // ❶
    new.link ← null;
    if (front=null) then {              // ❷
        rear ← new;
        front ← new;
    }
    else {                              // ❸
        rear.link ← new;
        rear ← new;
    }
end enQueue()
```



[그림 7-10] 연결 큐의 삽입 연산 과정 : 알고리즘 설명 ❷

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



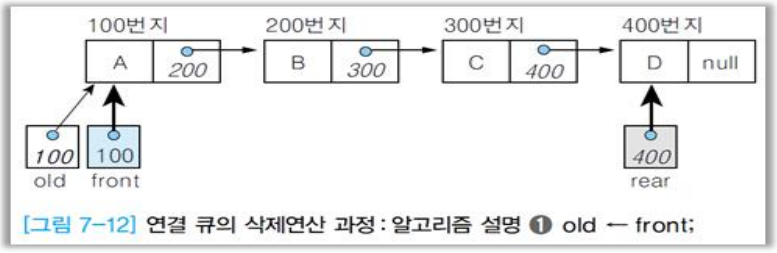
[그림 7-11] 연결 큐의 삽입 연산 과정 : 알고리즘 설명 ❸

* 연결 큐의 삭제 알고리즘

알고리즘 7-13 연결 큐의 삭제 알고리즘

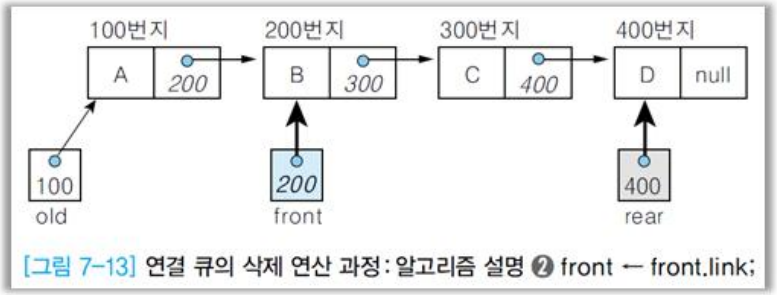
```
deQueue(LQ)
    if(isEmpty(LQ)) then Queue_Empty();
    else {
        old ← front;                          // ❶
        item ← front.data;
        front ← front.link;                   // ❷
        if (isEmpty(LQ)) then rear ← null;    // ❸
        returnNode(old);                      // ❹
        return item;
    }
end deQueue()

delete(LQ)
    if(isEmpty(LQ)) then Queue_Empty();
    else {
        old ← front;
        front ← front.link;
        if(isEmpty(LQ)) then rear ← null;
        returnNode(old);
    }
end delete()
```

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



[그림 7-12] 연결 큐의 삭제연산 과정 : 알고리즘 설명 ❶ old ← front;

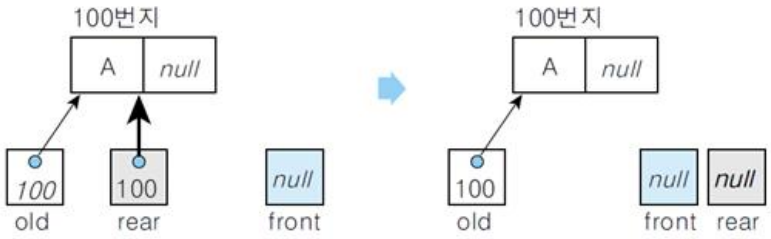
② 삭제연산 후에는 현재 front 노드의 다음 노드가 front 노드(첫번째 노드)가 되어야 하므로, 포인터 front를 재설정한다



[그림 7-13] 연결 큐의 삭제 연산 과정 : 알고리즘 설명 ❷ front ← front.link;

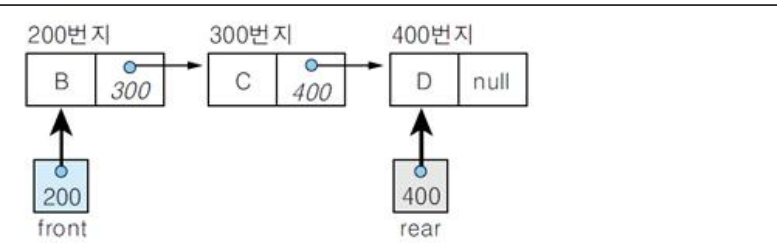
③ 현재 큐에 노드가 하나뿐이어서 삭제연산 후에 공백 큐가 되는 경우 :

☞ 큐의 마지막 노드가 없어지므로 포인터 rear를 null로 설정한다.



[그림 7-14] 연결 큐의 삭제 연산 과정 : 알고리즘 설명 ❸

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



[그림 7-15] 연결 큐의 삭제 연산 과정 : 알고리즘 설명 ❹ returnNode(old);

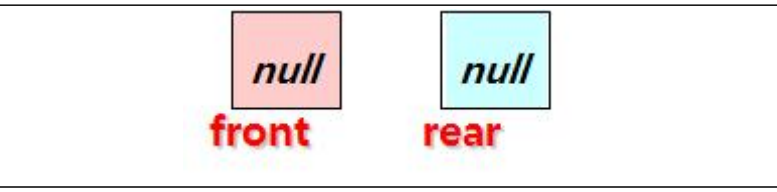
* 연결 큐의 검색 알고리즘

- 연결 큐의 첫 번째 노드, 즉 frotn 노드의 데이터 필드 값을 반환

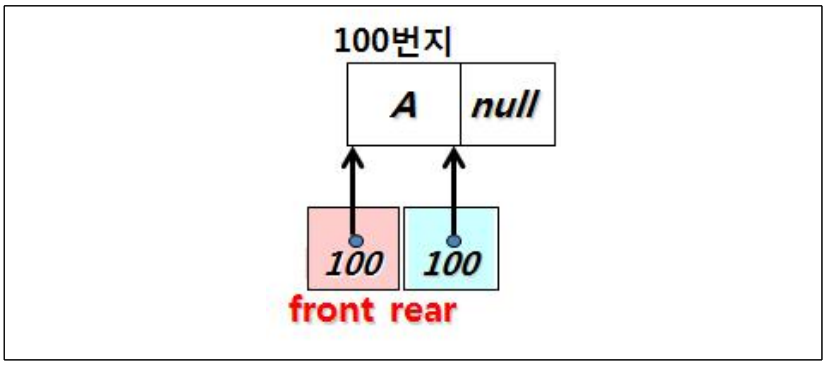
알고리즘 7-14 연결 큐의 검색 알고리즘

```
peek(LQ)
    if(isEmpty(LQ)) then Queue_Empty()
    else return (front.data);
end peek()
```

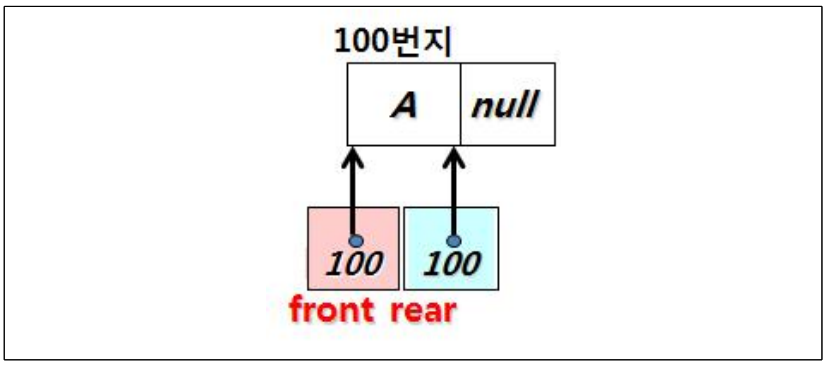
① 공백 큐 생성 : createLinkedQueue();

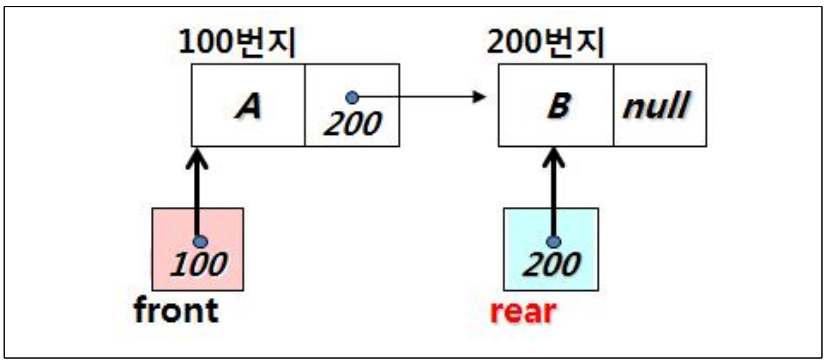


② 원소 A 삽입 : enQueue(LQ, A);

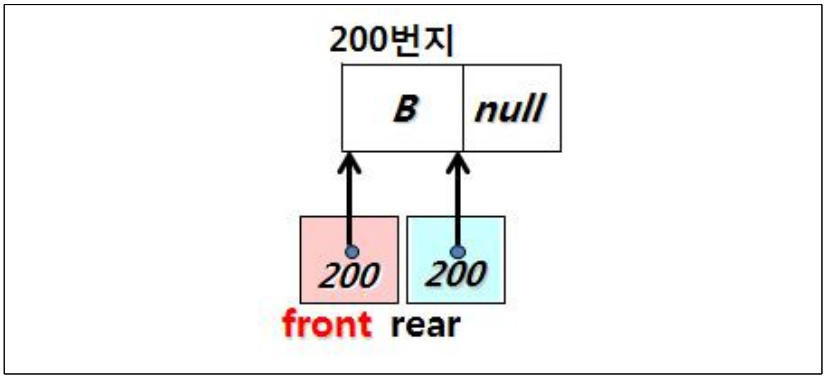


③ 원소 B 삽입 : enQueue(LQ, B);

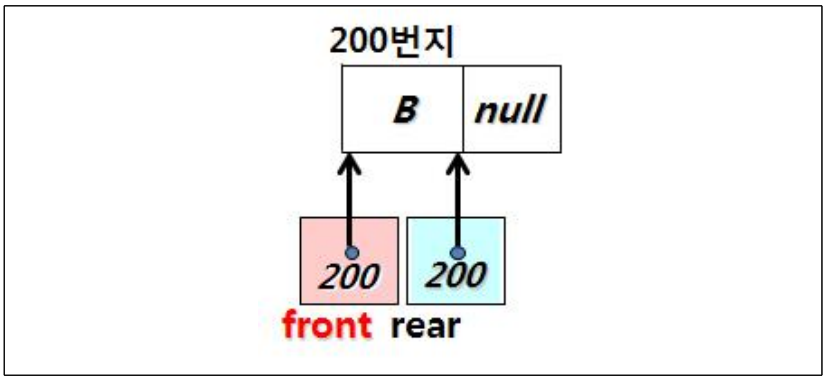


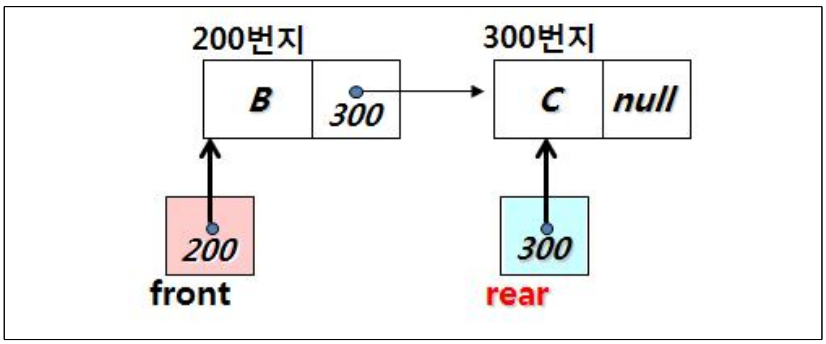


④ 원소 삭제 ¦ : deQueue(LQ);

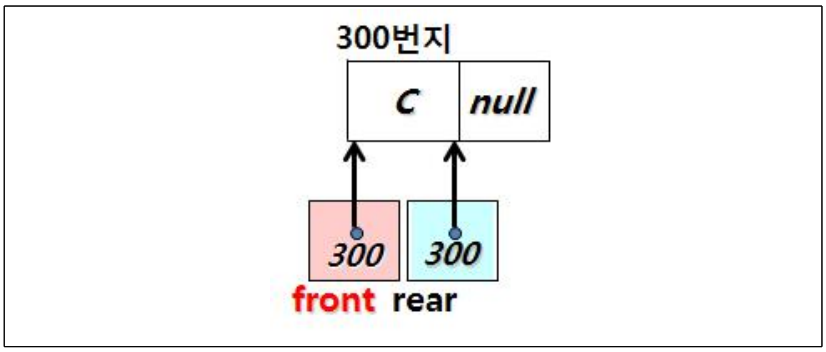


⑤ 원소 C 삽입 : enQueue(LQ, C);

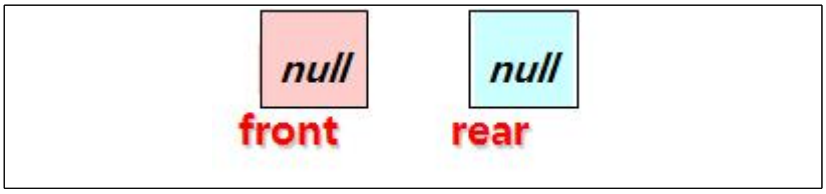




⑥ 원소 삭제 : deQueue(LQ);



⑦ 원소 삭제 : deQueue(LQ);



【학습정리】

1. 원형 큐는 1차원 배열의 처음과 끝을 논리적으로 연결하여 만든 큐로서 선형 큐의 잘못된 포화 상태 문제를 해결한다.

2. 연결 자료구조로 구현한 연결 큐는 크기에 제한이 없으며, 데이터 필드와 링크 필드를 가진 노드를 사용하여 큐의 원소를 표현하여 첫 번째 노드를 가리키는 포인터 front와 마지막 노드를 가리키는 포인터 rear를 사용한다.