***Queue***

***2018 – 06 - 09***

**1, 선형 Queue**

* 1차원 배열을 이용
* Queue의 크기 = 배열의 크기
* Front : 저장된 첫 번째 원소의 인덱스
* Rear : 저장된 마지막 원소의 인덱스
* 상태표현
* 초기 상태 : front = rear = -1
* 공백 상태 : front = rear
* 포화 상태 : rear = n-1 ( n : 배열의 크기, n-1 : 배열의 마지막 인덱스 )

**알고리즘**

**\*초기화**

//크기 n인 1차원 배열 생성

//front, rear = -1 로 초기화

**\*삽입: enQueue()**

//마지막 원소 뒤에 새로운 원소를 삽입하기 위해

// 1, rear 값을 하나 증가시켜 새로운 원소를 삽입할 자리를 마련함

// 2, 그 인덱스에 해당하는 배열 원소 Q[rear]에서 item을 저장

enQueue(item)

if(isFull()) then Queue\_Full();

else {

rear <- rear +1;

Q[rear] <- item;

}

End enQueue()

**\*삭제: deQueue()**

//가장 앞에 있는 원소를 삭제하기 위해

//1, fornt 값을 하나 증가시켜 Queue에 남아있게 될 첫 번째 원소로 이동함

//2, 새로운 첫 번째 원소를 리턴함으로써 삭제와 동일한 기능을 함

deQueue()

if(isEmpty()) then Queue\_Empty();

else {

front <- front + 1;

return\_Q[front];

}

end deQueue()

**\*isEmpty() / isFull()**

isEmpty()

if(front=rear) then return true;

else return false;

end isEmpty()

isFull()

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

else return false;

end isFull()

**\*검색: Qpeek()**

//가장 앞에 있는 원소를 검색하여 반환하는 연산

//현재 Front의 한자리 뒤(front+1)에 있는 원소, 즉 Queue의 첫 번째에 있는 원소를 반환

Qpeek()

If(isEmpty()) then Queue\_Empty();

Else return Q[front+1];

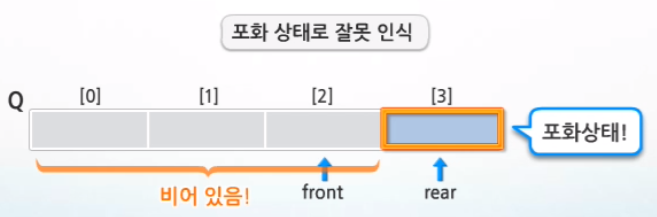
end Qpeek()

**\*선형 Queue의 문제점: 잘못된 포화 상태 인식**

1, 삽입, 삭제를 계속할 경우 배열의 앞부분에 활용할 수 있는 공간이 있음에도 불구하고,

Rear = n -1 인 상태 즉, 포화 상태로 인식

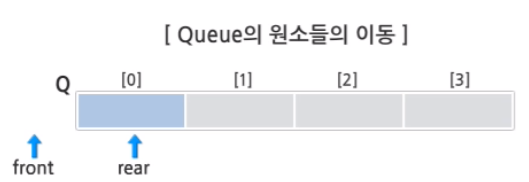
2, 더 이상의 삽입을 수행하지 않음



\*해결방법1

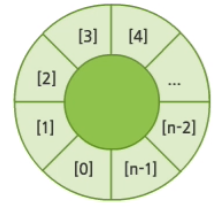
- 매 연산이 이루어질 때마다 저장된 원소들을 배열의 앞부분으로 모두 이동시킴

- 단점: 원소의 이동에 많은 시간이 소요되어 Queue의 효율성이 급격히 떨어진다.



\*해결방법2

-1차월 배열을 사용하되 논리적으로 배열의 처음과 끝이 연결되어 원형 형태의 Queue를 이룬다고 가정하고 사용함

- 원형의 Queue의 논리적 구조

**2, 원형 Queue**

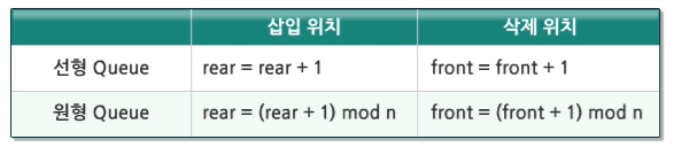
* 초기 공백 상태

front = rear = 0

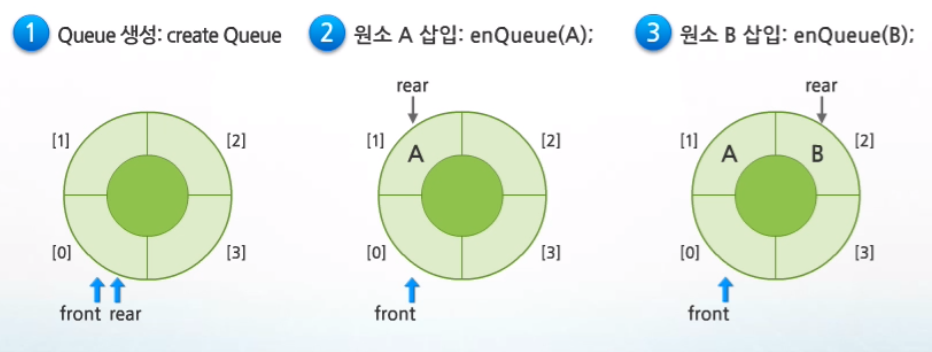
* Index의 순환
* front와 rear의 위치가 배열의 마지막 인덱스인 n-1를 가리킨 후, 논리적 순환을 이루어 배열의 처음 인덱스인 0으로 이동해야 함
* 이를 위해 나머지 연산자 mod를 사용
* front 변수

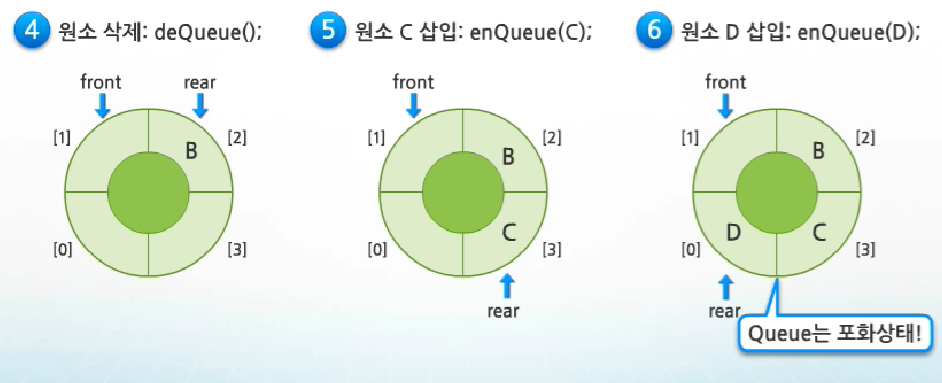
공백 상태와 포화 상태 구분을 쉽게 하기 위해 front가 있는 자리는 사용하지 않고 항상 빈자리로 둠

* 삽입 위치 및 삭제 위치



**\*원형 Queue의 기본 연산 과정**





**알고리즘**

\*초기 createQueue()

//크기 n인 1차원 배열 생성

//front, rear = 0으로 초기화

\* isEmpty(), isFull()

//공백상태 및 포화상태 검사

//공백상태: front = rear

//포화상태: 삽입할 rear의 다음 위치 = 현재 front

// (rear + 1) mod n = front

isEmpty()

if( front = rear ) then return true;

else return false;

end isEmpty()

isFull()

if( ( rear + 1 ) mod n = front ) then return true;

else return false;

end isFull()

\*삽입: enQueue(item)

//마지막 원소 뒤에 새로운 원소를 삽입하기 위해

//1, rear값을 조정하여 새로운 원소를 삽입할 자리를 마련함: rear <- (rear +1) mod n;

//2, 인덱스에 해당하는 배열원소 cQ[rear]에 item을 저장

enQueue(item)

if(isFull()) then Queue\_Full();

else {

rear <- ( rear + 1 ) mod n;

cQ[rear] <- item;

}

End enQueue()

\*삭제: deQueue(), delete()

가장 앞에 있는 원소를 삭제하기 위해

//1, front 값을 조정하여 삭제할 자리를 준비함

//2, 새로운 front 원소를 리턴함으로써 삭제와 동일한 기능을 함

deQueue()

if(isEmpty()) then Queue\_Empty();

else {

front <- ( front + 1 ) mod n;

return cQ[front];

}

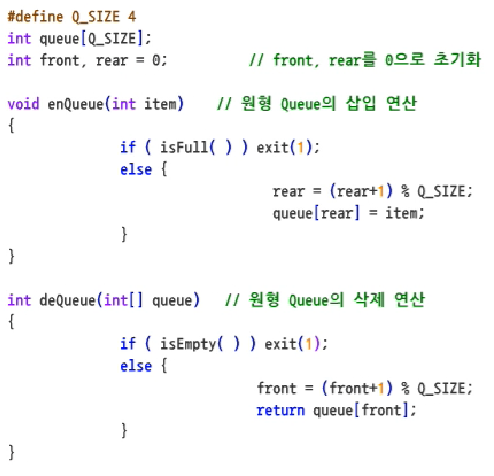
end deQueue()

delete()

if( isEmpty() ) then Queue\_Empty();

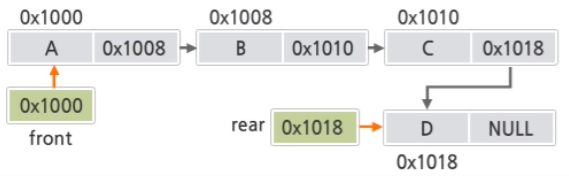
else front <- ( front + 1 ) mod n;

end delete()

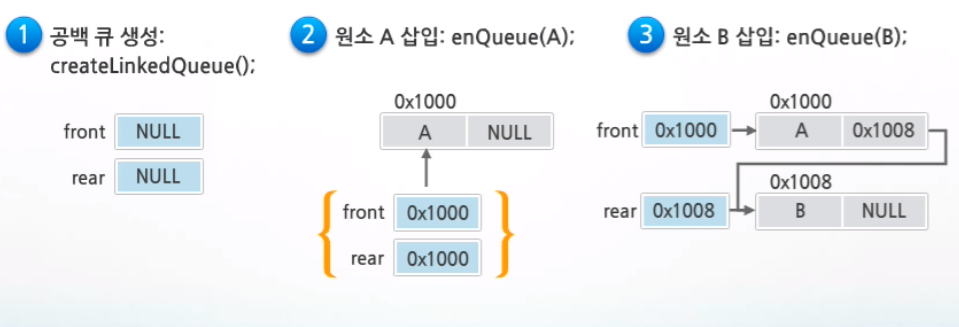


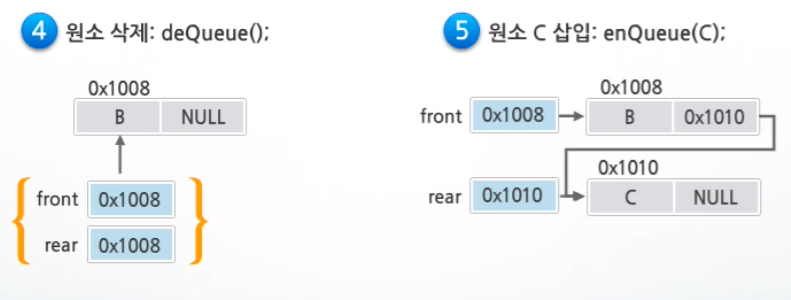
3, 연결 Queue

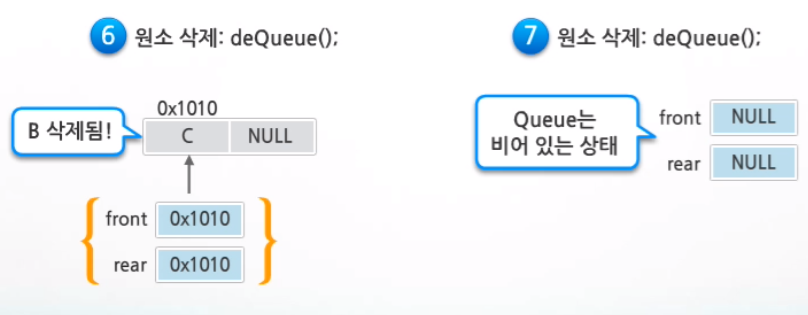
* 단순 연결 리스트(Linked List)를 이용한 Queue
* Queue의 원소 : 단순 연결 리스트의 노드
* Queue의 원소 순서 : 노드의 연결 순서, 링크로 연결되어 있음
* front : 첫 번째 노드를 가리키는 링크
* rear : 마지막 노드를 가리키는 링크
* 상태 표현
* 초기 상태 : front = rear = NULL
* 공백 상태 : front = rear = NULL



\*연결 Queue의 기본 연산 과정







알고리즘

\*creatLinkedQueue()

//리스트 노드 없이 포인터 변수만 생성함

//front와 rear를 NULL로 초기화

createLinkedQueue()

front <- NULL;

rear <- NULL;

end createLinkedQueue()

\*isEmpty()

// 공백상태 : front = rear = NULL

ifEmpty()

if(front=null) then return true;

else return false;

end isEmpty()

\*삽입 : enQueue(item)

//1, 새로운 노드 생성 후 데이터 필드에 item 저장

//2, 연결 Queue가 공백인 경우, 아닌 경우에 따라 front, rear변수 지정

enQueue(item)

new <- getNode(); //getNode(): 새로운 노드 할당 후 리턴

new.data <-item;

new.link <- NULL;

if ( front = NULL ) then {

rear <- new;

front <- new;

}

else {

rear.link <- new;

rear <- new;

}

end enQueue()

\*삭제: deQueue(), delete()

//1, old가 지울 노드를 가리키게 하고, front 재설정

//2, 삭제 후 공백 Queue가 되는 경우, rear도 NULL로 설정

//3, old가 가리키는 노드를 삭제하고 메모리 반환

deQueue()

if( isEmpty()) then Queue\_Empty();

else {

old <- front;

item <- front.data;

front <- front.link;

if ( isEmpty()) then rear <- NULL;

free(old);

return item;

}

end deQueue();

