큐(Queue)

❖ 큐(Queue)

- 큐는 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)됨
  - ☞ 선입선출 구조 (FIFO, First-In-First-Out)

큐(Queue)

❖ 큐의 구조

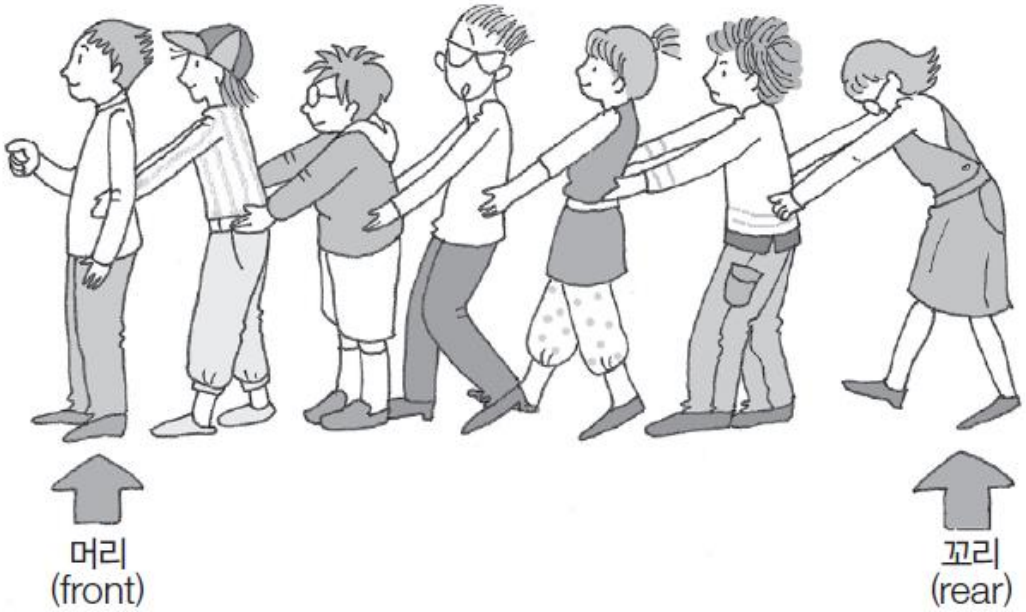


그림 6-2 FIFO 구조의 예 : 꼬리잡기 놀이의 머리와 꼬리

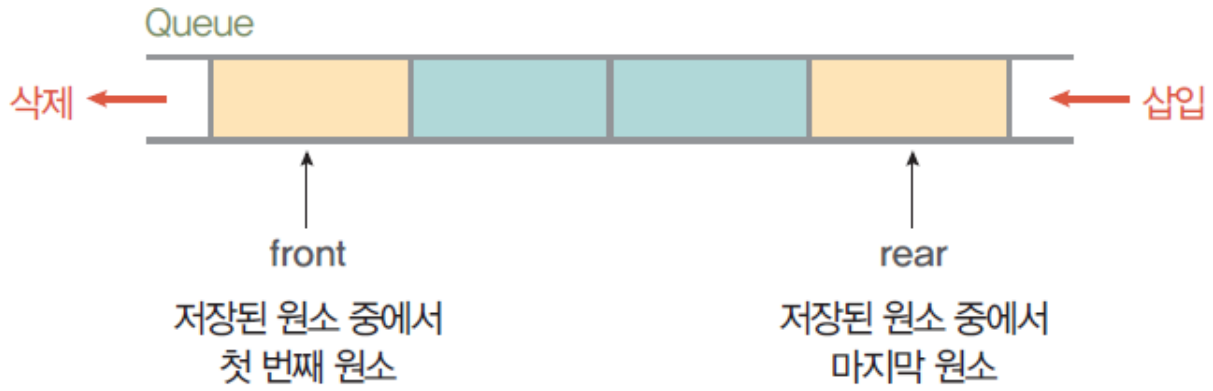


그림 6-3 큐의 FIFO 구조

큐(Queue)

❖ 큐의 연산

- ㅇ 삽입 : enQueue
- ㅇ 삭제 : deQueue

❖ 스택과 큐의 연산 비교

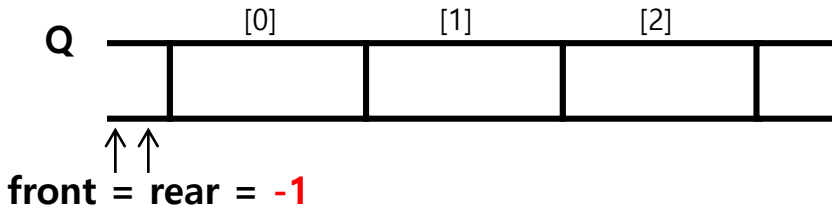
표 6-1 스택과 큐에서의 삽입과 삭제 연산 비교

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

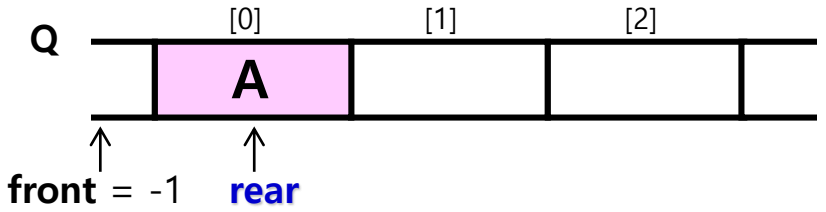
큐(Queue)

❖ 큐의 연산 과정

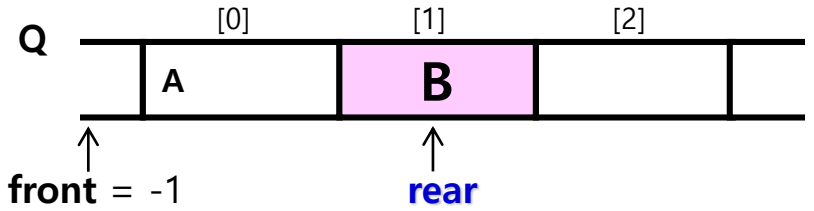
① 공백 큐 생성 : createQueue();



② 원소 A 삽입 : enQueue(Q, A);

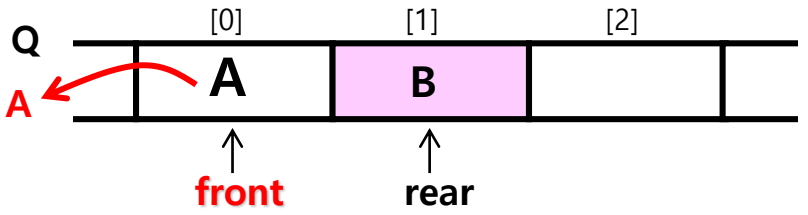


③ 원소 B 삽입 : enQueue(Q, B);

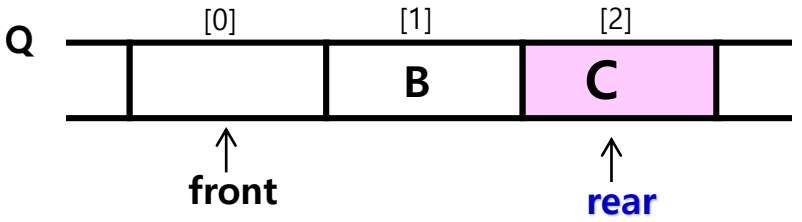


큐(Queue)

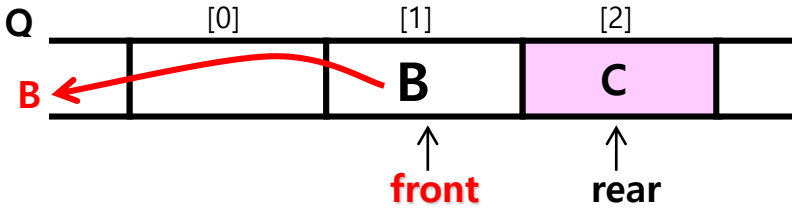
④ 원소 삭제 : deQueue(Q);



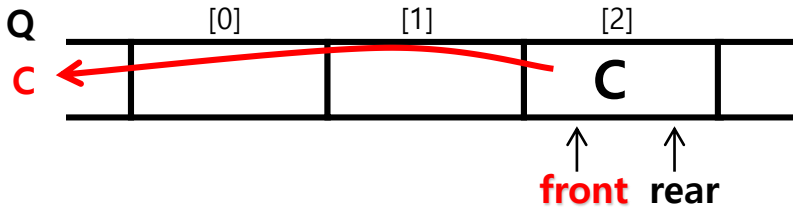
⑤ 원소 C 삽입 : enQueue(Q, C);



⑥ 원소 삭제 : deQueue(Q);



⑦ 원소 삭제 : deQueue(Q);



큐(Queue)

❖ 큐의 구현

- 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)

큐(Queue)

❖ 큐의 구현

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

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

큐(Queue)

- 큐의 삽입 알고리즘
  - 마지막 원소의 뒤에 삽입해야 하므로

  ① 마지막 원소의 인덱스를 저장한 rear의 값을 하나 증가시켜 삽입할 자리 준비

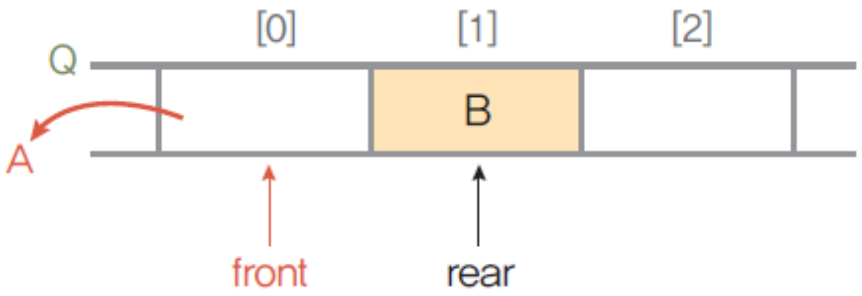
  ② 수정한 rear값에 해당하는 배열원소 Q[rear]에 item을 저장

큐(Queue)

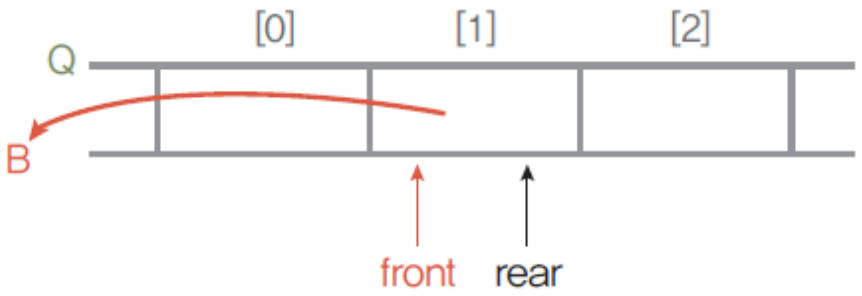
- 큐의 삭제 알고리즘
  - 가장 앞에 있는 원소를 삭제해야 하므로

  ① front의 위치를 한자리 뒤로 이동하여 큐에 남아있는 첫 번째 원소의 위치로 이동하여 삭제할 자리 준비

  ② front 자리의 원소를 삭제하여 반환



(a) 첫 번째 deQueue( ) 연산 후 상태



(b) 두 번째 deQueue( ) 연산 후 상태

그림 6-4 deQueue( ) 연산 후 상태

큐(Queue)

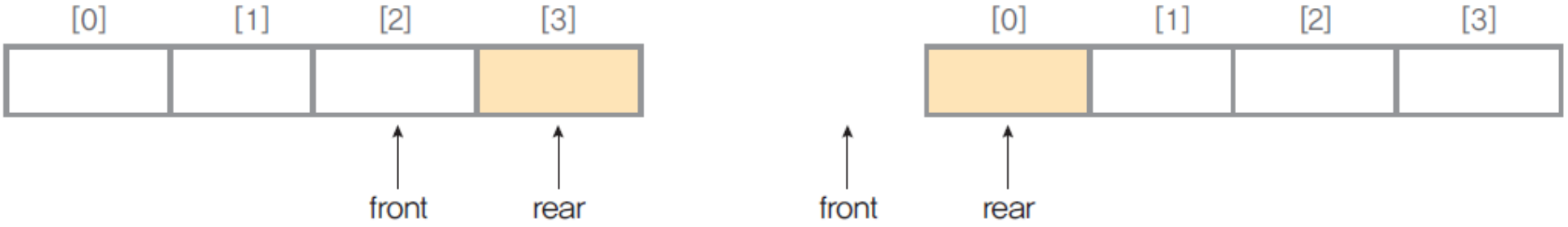
- 큐의 검색 알고리즘
  - 가장 앞에 있는 원소를 검색하여 반환하는 연산

  ① 현재 front의 한자리 뒤(front+1)에 있는 원소, 즉 큐에 있는 첫 번째 원소를 반환

❖ 배열(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

❖ 순차 큐의 잘못된 포화상태 인식의 해결 방법-2

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

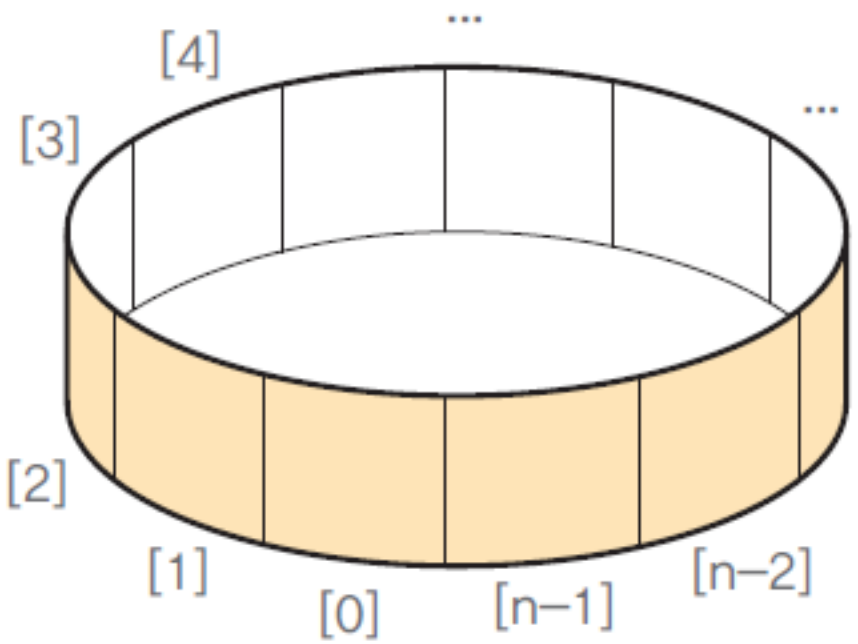


그림 6-6 원형 큐의 논리적 구조

❖ 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
  - 3 ÷ 4 = 0 …3 (몫=0, 나머지=3)
  - 3 mod 4 = 3

표 6-2 순차 큐와 원형 큐의 비교

종류	삽입 위치	삭제 위치
순차 큐	rear = rear + 1	front = front +1
원형 큐	rear = (rear+1) mod n	front = (front+1) mod n

  - 사용조건) 공백 상태와 포화 상태 구분을 쉽게 하기 위해서 front가 있는 자리는 사용하지 않고 항상 빈자리로 둠

큐(Queue)

❖ 초기 공백 원형 큐 생성 알고리즘

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

❖ 원형 큐의 공백상태 검사 알고리즘과 포화상태 검사 알고리즘

표 6-3 원형 큐의 상태에 따른 front와 rear의 관계

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

❖ 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

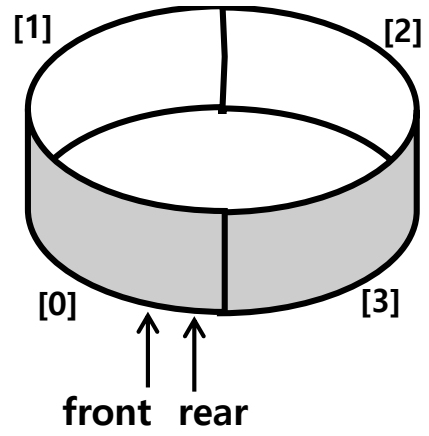
❖ 원형 큐의 삭제 알고리즘

① front의 값을 조정하여 삭제할 자리를 준비

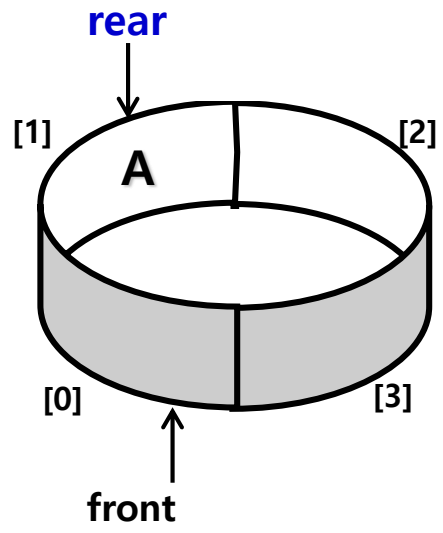
② 준비한 자리에 있는 원소 cQ[front]를 삭제하여 반환

❖ 크기가 4인 원형 큐에서 큐를 생성하고 삽입 · 삭제하는 연산 과정

① 공백 원형 큐 생성 : createQueue();

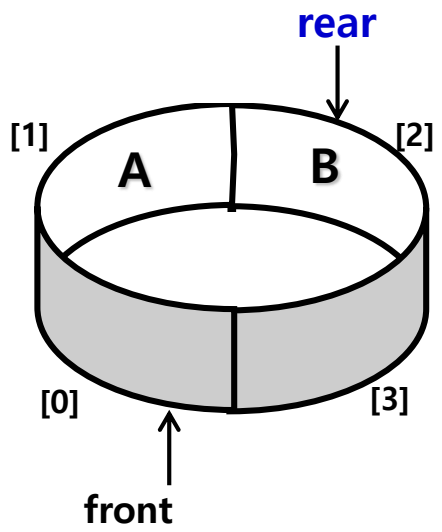


② 원소 A 삽입 : enQueue(cQ, A);

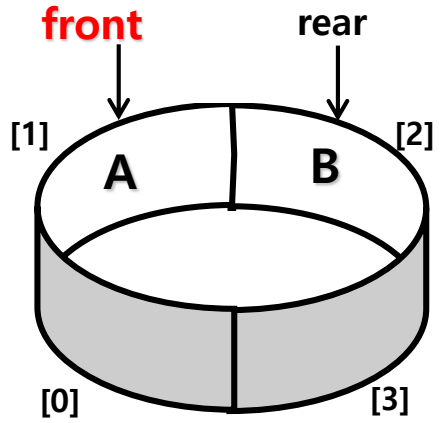


큐(Queue)

③ 원소 B 삽입 : enQueue(cQ, B);

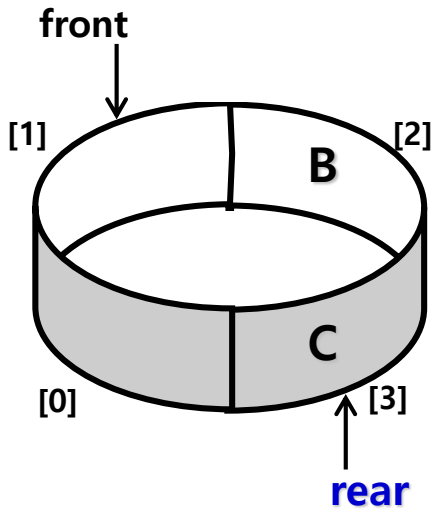


④ 원소 삭제 : deQueue(cQ);
(삭제 데이터 : A)



큐(Queue)

⑤ 원소 C 삽입 : enQueue(cQ, C);



⑥ 원소 D 삽입 : enQueue(cQ, D);