대구대학교 DAEGU UNIVERSITY

일상에서 찾아보는

자료구조의 원리

04주차 순서대로 처리하기 - 큐

2차시

메모리를 효율적으로 사용하는 원형 큐

학습목표

- FIFO(First-in First-out)의 원리를 원형 구조로 처리하는 방법을 설명할 수 있다.
- 원형 큐의 구현 방법을 설명할 수 있다.

학습내용

- 일상에서 자원 공유의 예
- 원형 큐의 원리
- 원형 큐의 상태
- 원형 큐의 동작
- 원형 큐의 구현

일상에서 자원 공유의 예

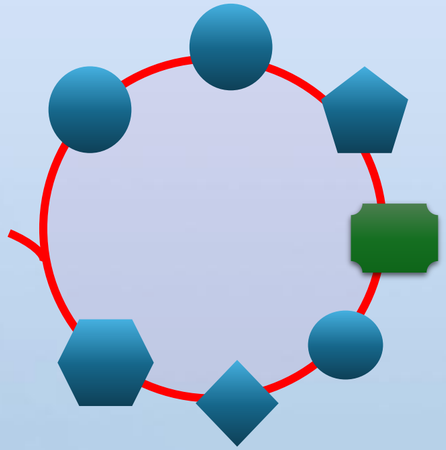
» 공용 자전거

- 한정된 공간을 공유
- 자원을 순환하며 사용하는 시스템
- 각 자전거가 사용되고 다시 돌아오면 처음 위치에서 다시 대기하여 다른 사용자가 사용할 수 있도록 함

원형 큐의 원리

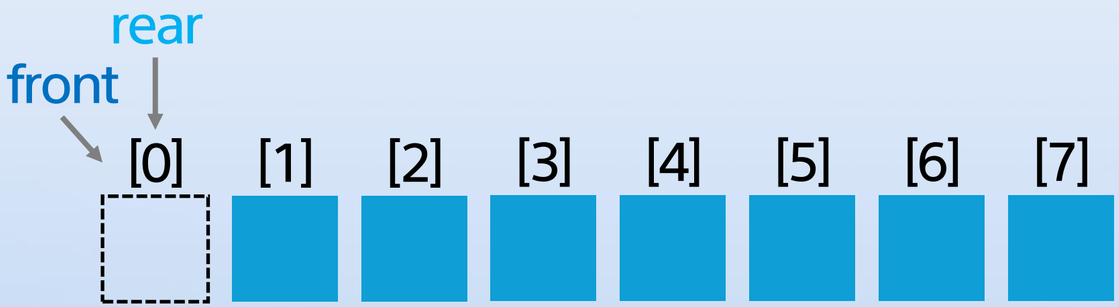
- 큐(Queue)의 일종
- 처음과 끝을 연결한 원형 구조
- 마지막 위치의 뒤에 첫 번째 위치가 이어지는 구조
- 자료를 순환하며 사용하는 구조

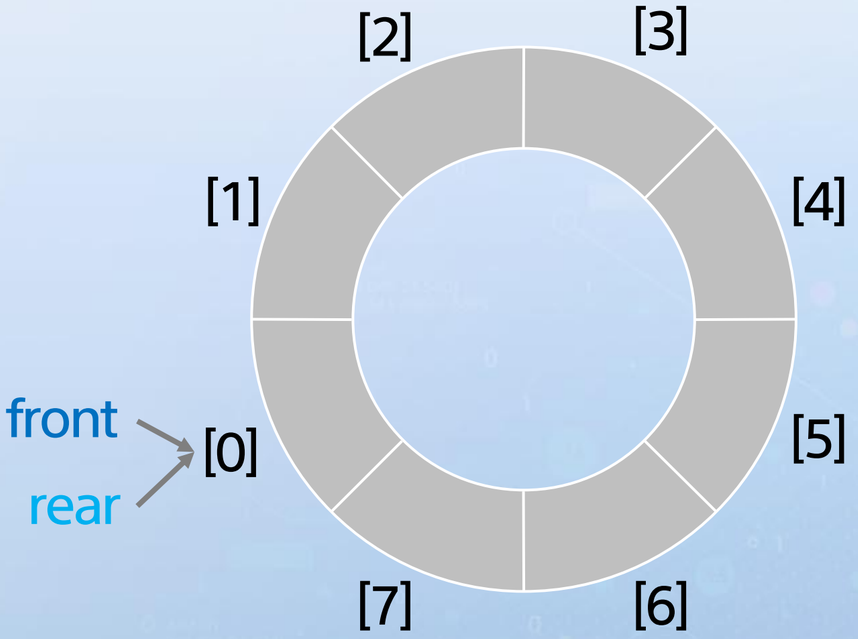
고정된 공간에 자료를
순환시켜 사용하는 원리



원형 큐의 원리

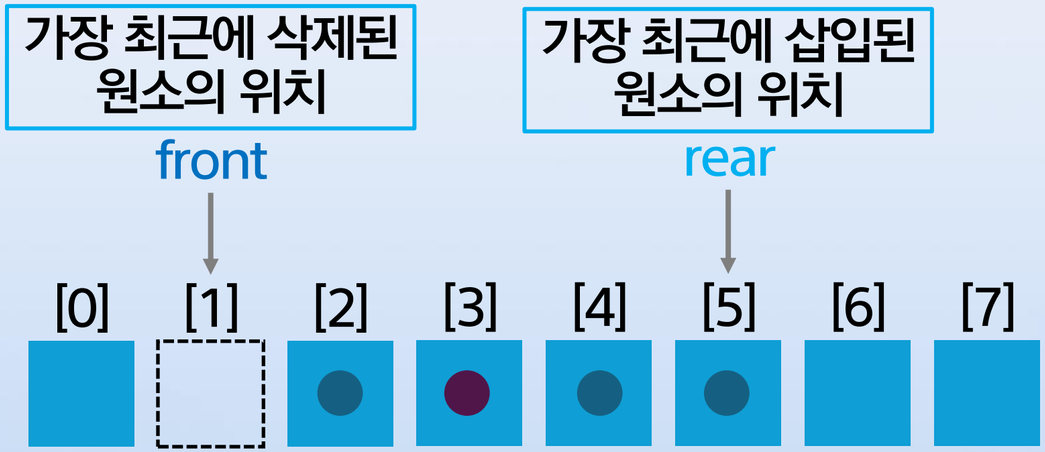
- 선형 리스트를 논리적으로 원형으로 생각함
- 리스트의 처음과 끝이 연결되어 있다고 가정함

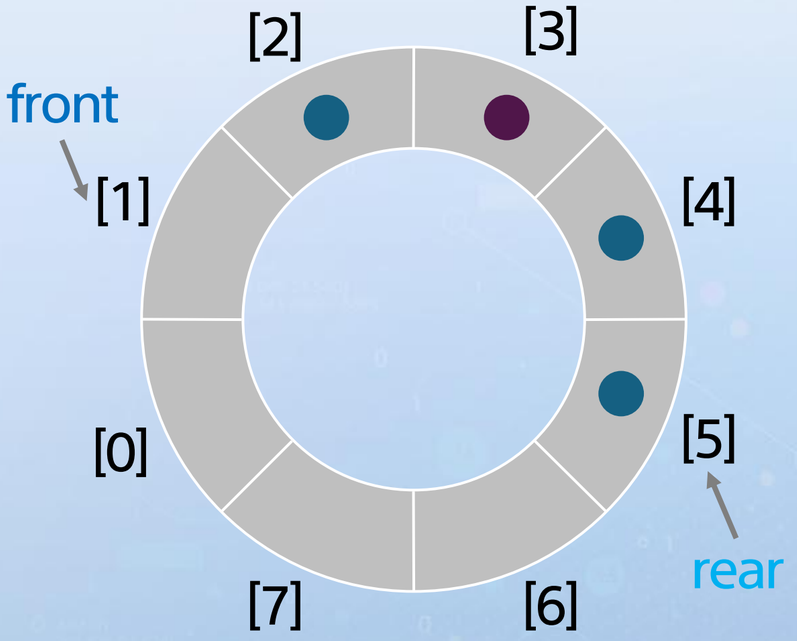




원형 큐의 원리

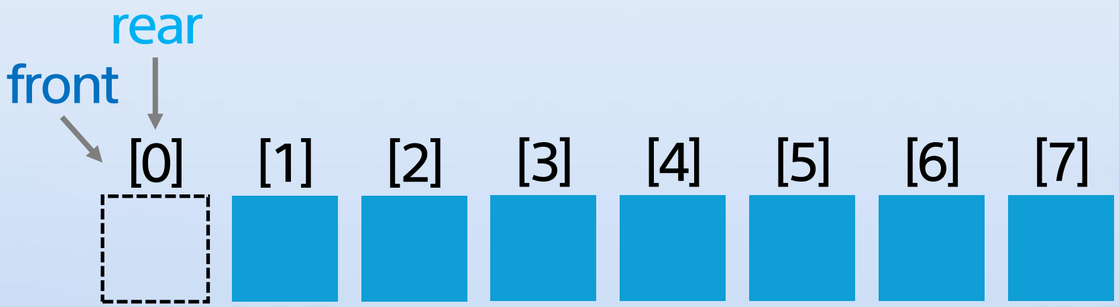
- 리스트의 크기는 변화되지 않고 원소의 삽입, 삭제 위치를 원형을 따라 움직이도록 함

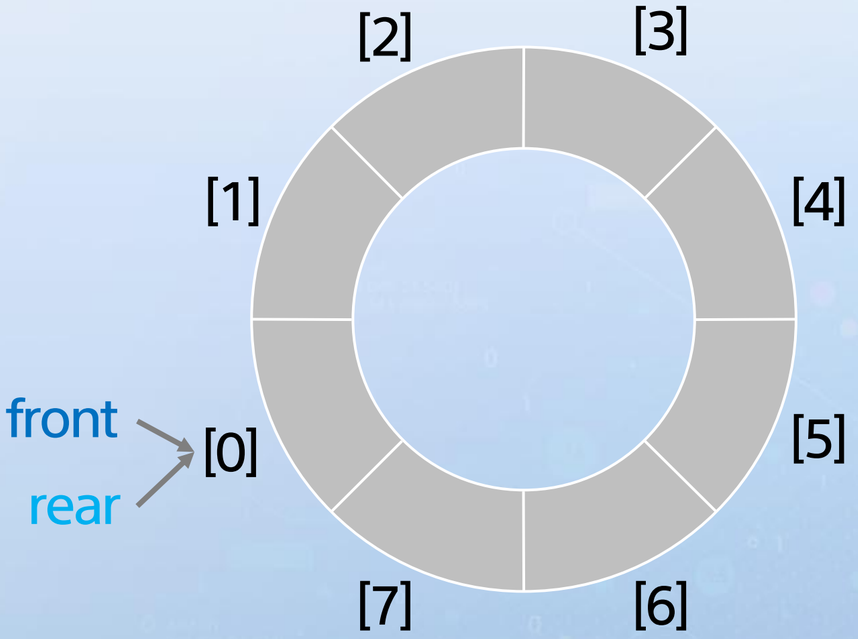




원형 큐의 원리

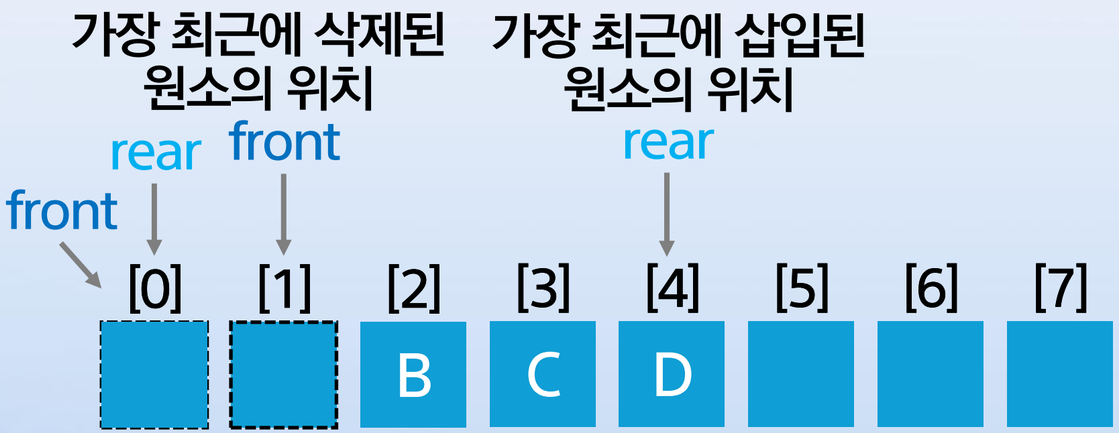
- front에는 값을 채우지 않음
  - 큐의 상태를 관리하고 효율적으로 작동하도록 하기 위함
  - 큐가 비어 있는 상태와 가득 찬 상태를 구분하기 위해 필요함

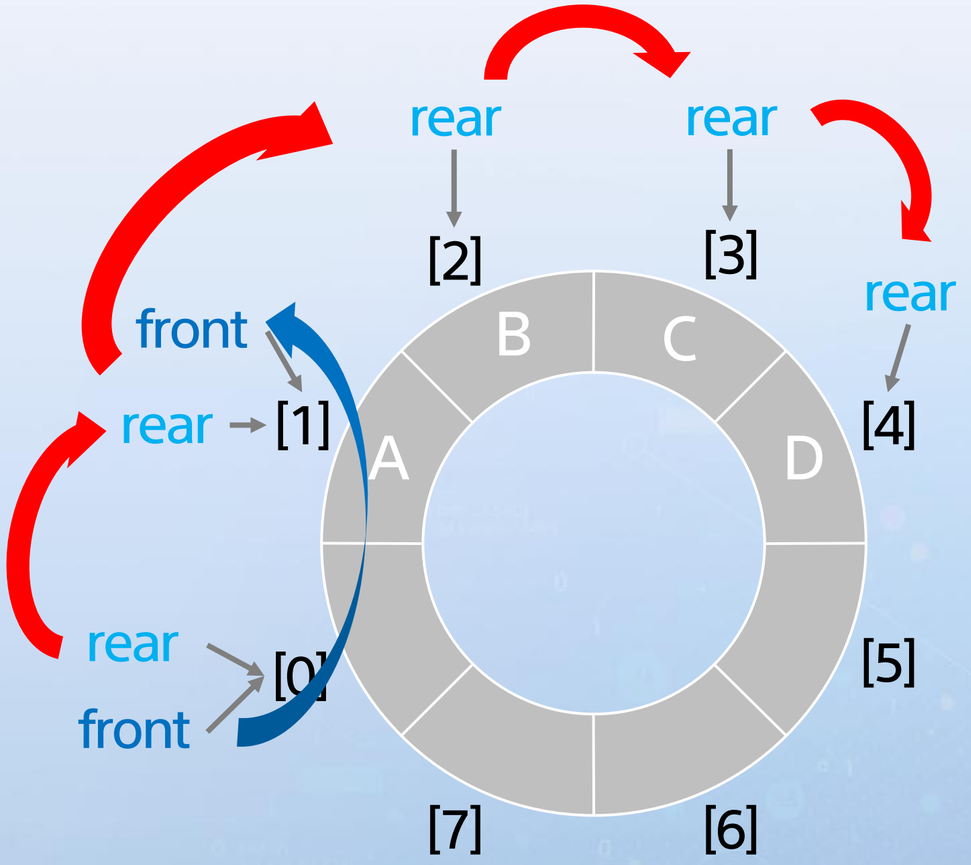




원형 큐의 원리

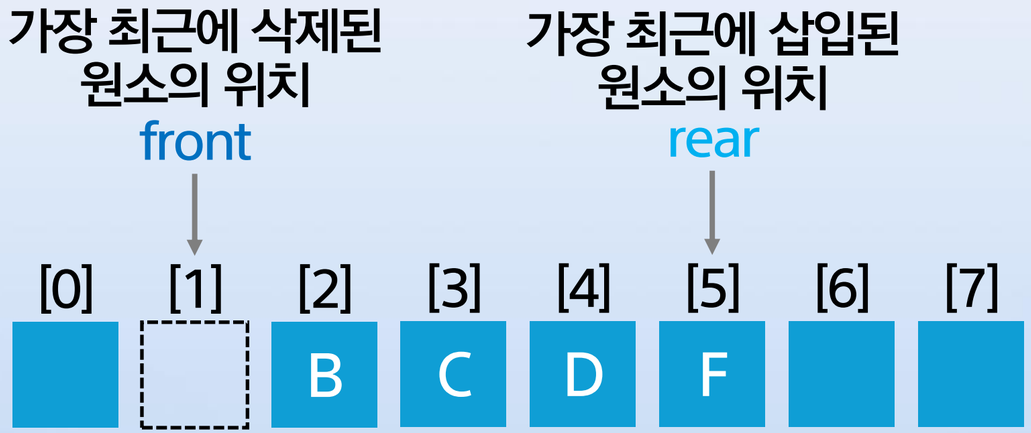
- A → B → C → D 삽입하고 한 번 삭제함

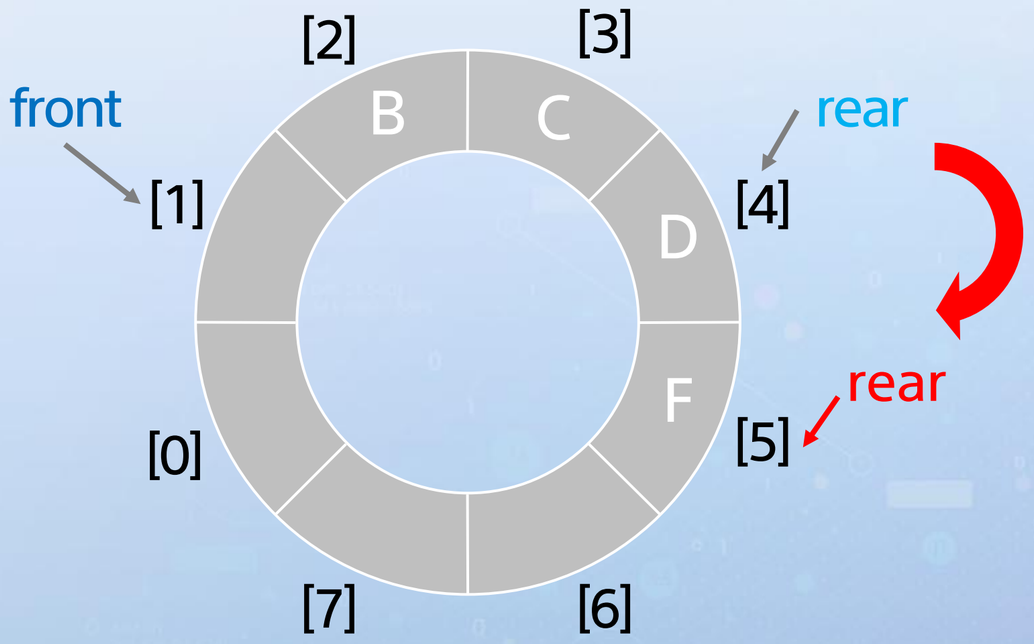




원형 큐의 원리

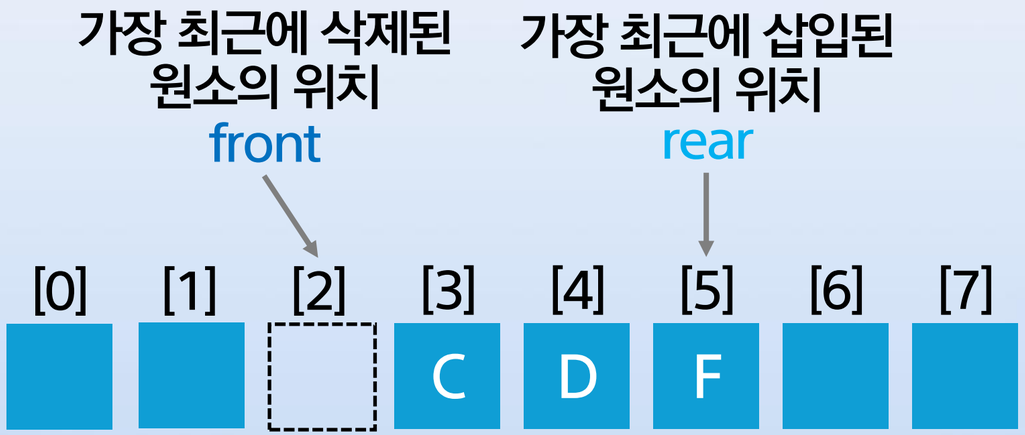
- 새로운 원소를 삽입하려면 rear가 한 칸 이동함
- rear의 위치에 삽입

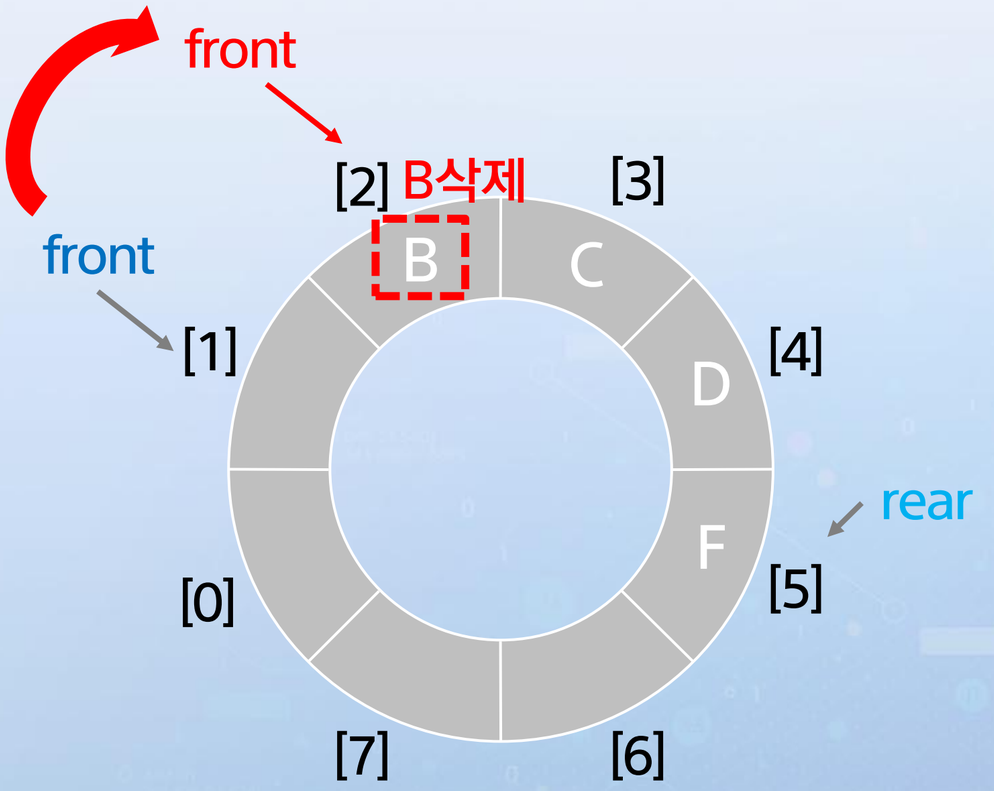




원형 큐의 원리

- 원소를 제거하려면 front가 한 칸 이동함
- front의 위치에서 삭제

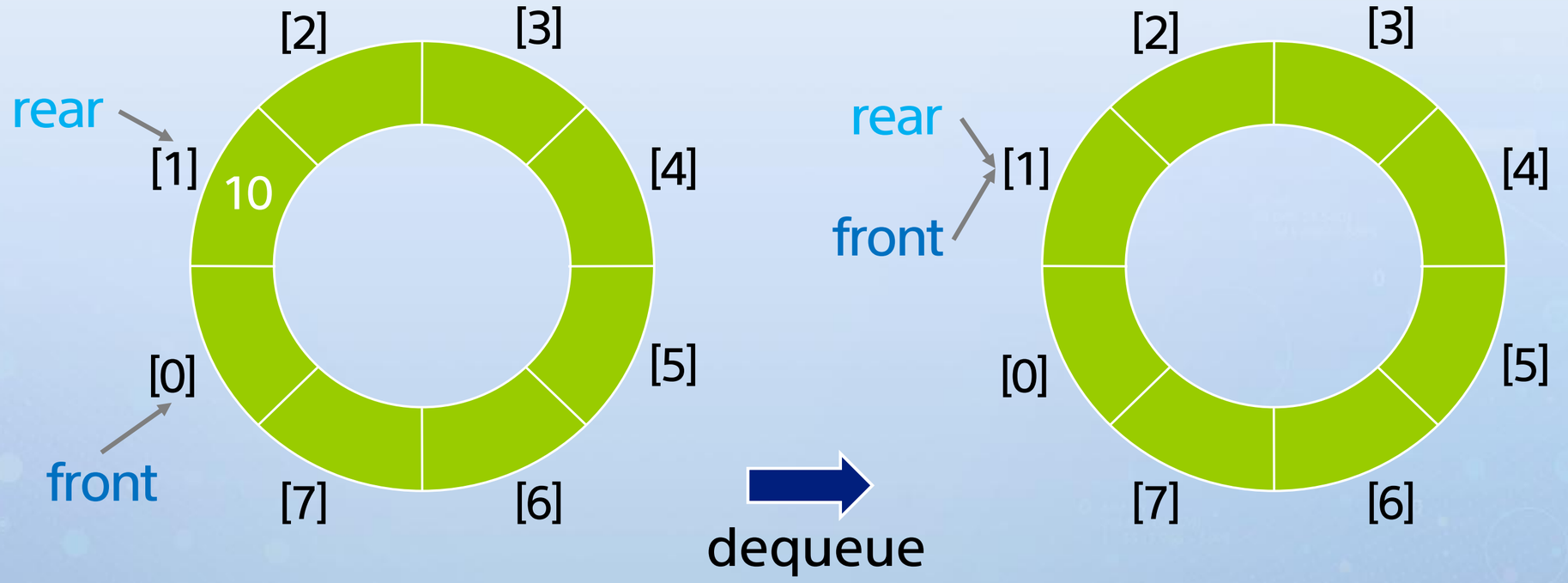




원형 큐의 상태

» 비어 있는 상태

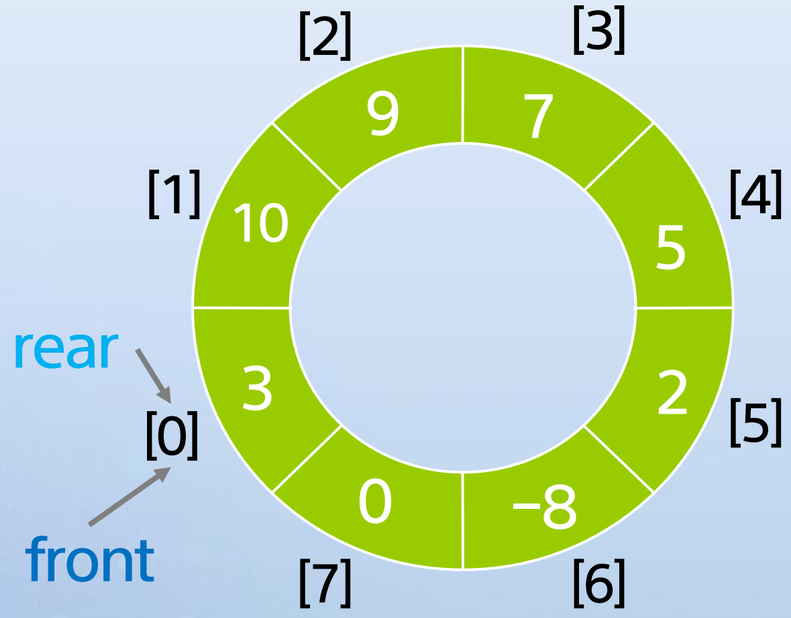
- front 와 rear가 같은 곳을 가리키면 비어 있음
- 큐가 비어 있으면 자료가 없기 때문에 dequeue를 할 수 없음



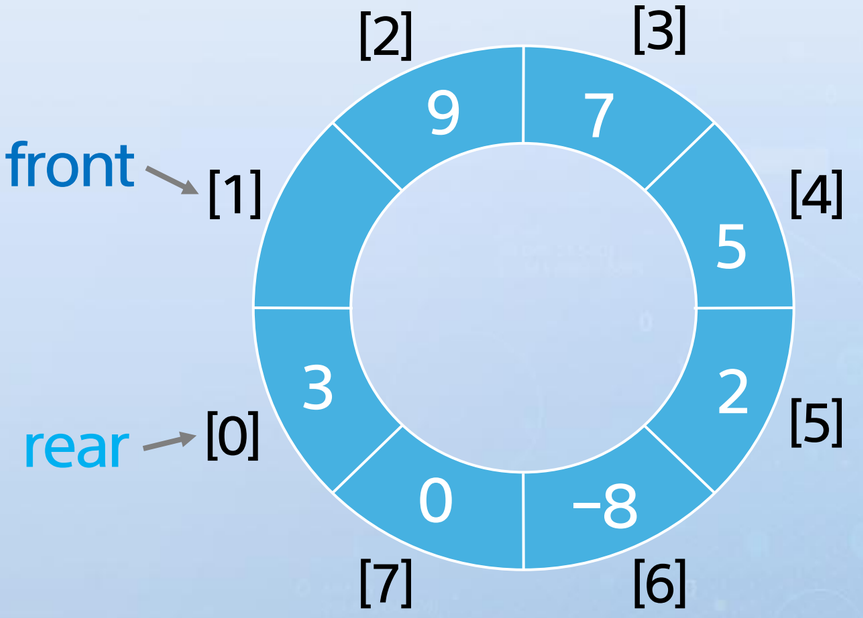
원형 큐의 상태

» 포화 상태

- front가 rear보다 한자리 앞에 있음
- 가득 찬 상태이면 더 이상 enqueue를 할 수 없음



오류

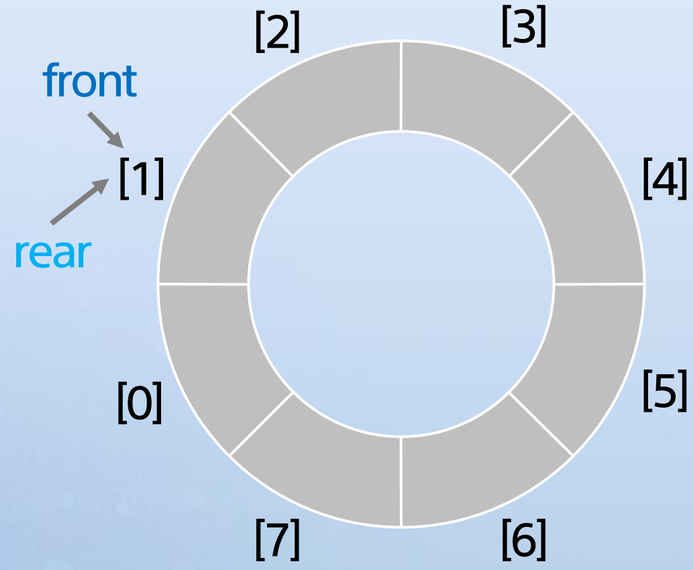


가득 찬 상태

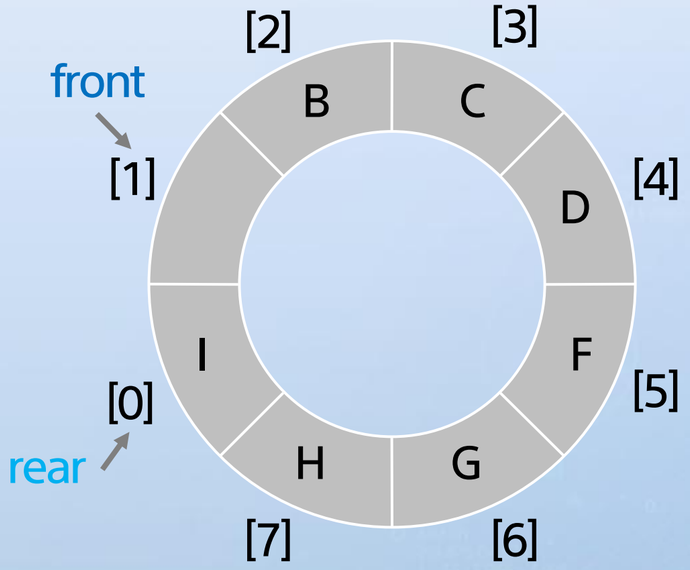
원형 큐의 상태

- front의 자리는 항상 비어 있음
- front 다음부터 rear까지 자료가 채워져 있음

비어 있는 상태:
front == rear



가득 찬 상태 :
front가 rear보다 한 자리 앞



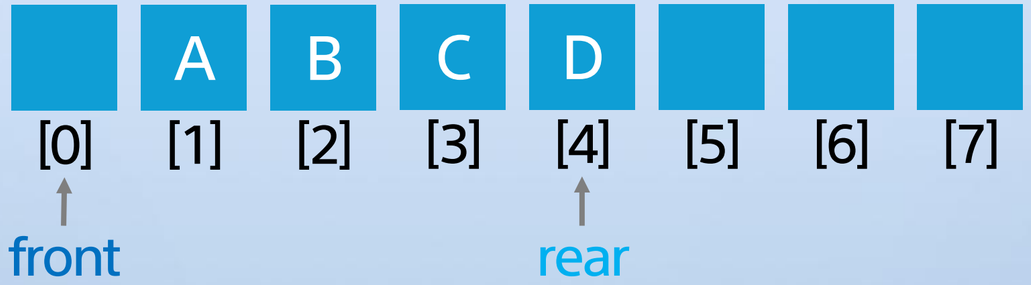
원형 큐의 동작

삽입

- rear를 1 증가하고 그 위치에 삽입
- rear가 가리키는 위치에 가장 최근에 삽입한 원소가 있음

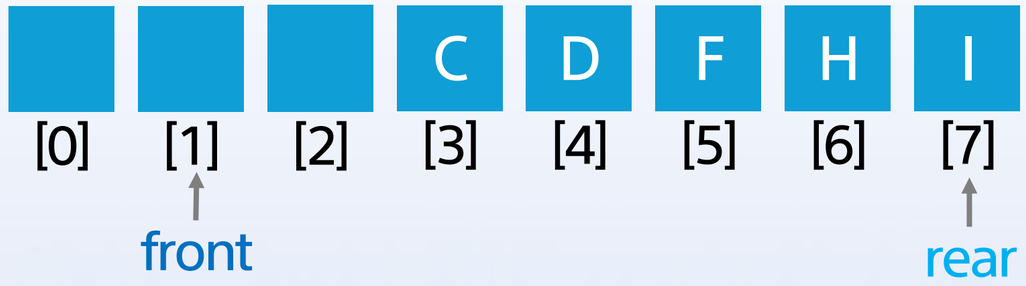
삭제

- front를 1 증가하고 그 위치에서 삭제
- front가 가리키는 위치는 가장 최근에 삭제된 위치

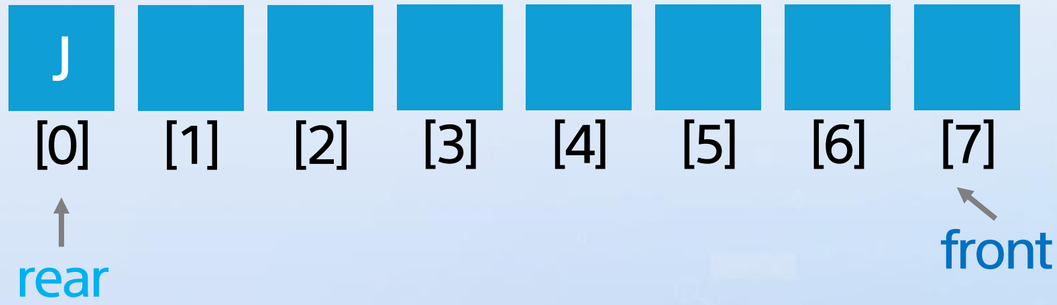


원형 큐의 동작

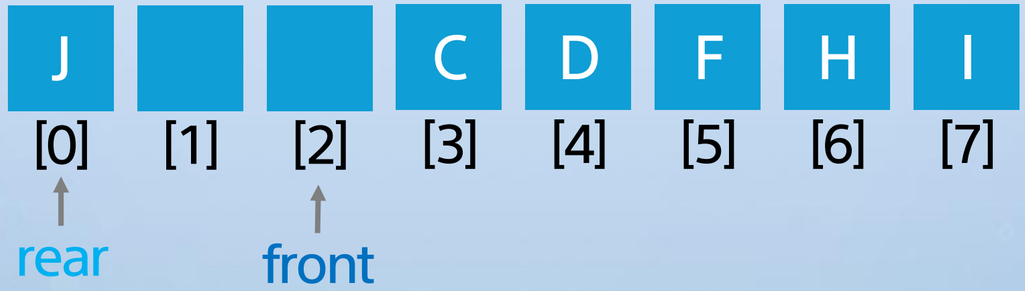
- 시계방향 회전
  - 리스트의 크기를 QSIZE로 정의



- front = (front+1)%QSIZE (7+1)%8
- rear = (rear+1)%QSIZE



  - 0부터 증가하다가 QSIZE-1 다음은 다시 0부터 시작함



원형 큐의 구현

» 원형 큐 클래스

```
QSIZE = 8
class CircularQueue :
    def__init__(self) :
        self.front = 0
        self.rear = 0
        self.q_list = [None] * QSIZE
```

원형 큐의 구현

» 원형 큐 클래스의 삽입

```
def enqueue(self, data):
        if not self.isFull() :                  # 포화상태가 아니면
            self.rear = (self.rear+1)%QSIZE     # rear 회전
            self.q_list[self.rear] = data       # rear 위치에 삽입
```

원형 큐의 구현

» 원형 큐 클래스의 삭제 메소드

```
def dequeue(self) :
        if not self.isEmpty() :                    # 빈 상태가 아니면
            self.front = (self.front+1)%QSIZE      # front 회전
            data = self.q_list[self.front]
            self.q_list[self.front] = None         # 비움
            return data                            # front 위치의 원소 반환
```

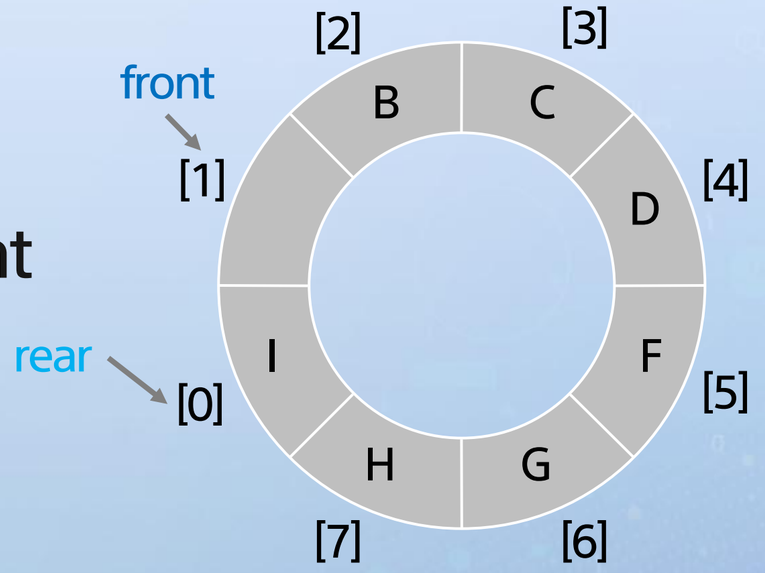
원형 큐의 구현

» 원형 큐의 상태 확인

```
def isEmpty(self) : return self.front == self.rear
def isFull(self) : return self.front == (self.rear+1)%QSIZE
def clear(self) : self.front = self.rear
def display(self):
        print("Queue state:", self.q_list)
```

- 가득 찬 상태
  - front가 rear보다 한 자리 앞
  - (rear + 1) % 큐의 크기 == front

(0+1)%8 = 0



원형 큐의 구현

» 테스트

```
cq = CircularQueue()
cq.enqueue(1)
cq.enqueue(2)
cq.enqueue(3)
cq.display()  # Queue state: [None, 1, 2, 3, None, None, None, None]
print(cq.dequeue())  # 1
cq.display()  # Queue state: [None, None, 2, 3, None, None, None, None]
```

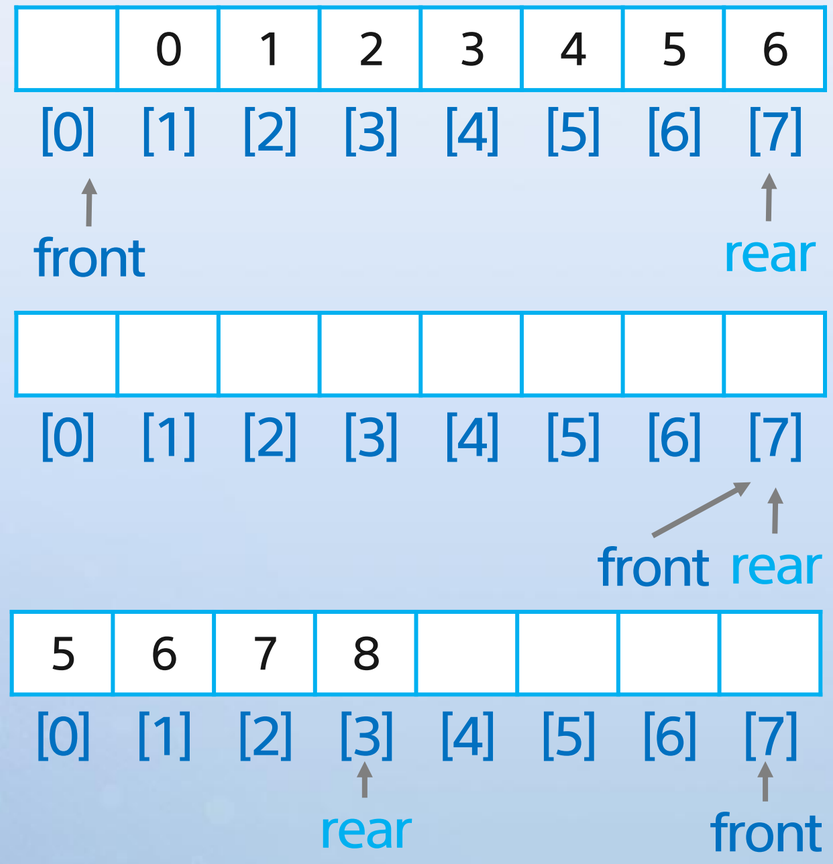
원형 큐의 구현

» 테스트

```
cq = CircularQueue()
for i in range(QSIZE):
        cq.enqueue(i)
for i in range(QSIZE):
        print(cq.dequeue(), " ", end='')
        print("front: ", cq.front, "rear : ", cq.rear)
print()
print("4회 삽입")
for i in range(5, 9):
        cq.enqueue(i)
print("front: ", cq.front, "rear : ", cq.rear)
```

원형 큐의 구현

» 테스트


0 1 2 3 4 5 6
[0] [1] [2] [3] [4] [5] [6] [7]
front
rear
[0] [1] [2] [3] [4] [5] [6] [7]
front rear
5 6 7 8
[0] [1] [2] [3] [4] [5] [6] [7]
rear
front

학습정리

원형 큐

- 리스트의 처음과 끝이 연결된 원형의 구조
- 선형 큐에 비해서 상태 관리가 복잡함
- 메모리 공간 등 고정된 자원을 순환하여 사용함
  - 네트워크 데이터 패킷 관리
  - 프로세서의 작업 스케줄링