큐

삽입이나 삭제 시 시간 복잡도 O(1)

데이터 탐색 시에는 찾을 때까지 수행하므로, O(n)

특성

- 삽입된 순서대로 원소가 저장되고,

  가장 먼저 들어온 원소가 우선적으로 나가는 선입선출 자료 구조

- 첫 번째 원소를 Front, 마지막 원소를 Rear 라고 한다

- 큐의 기본적인 연산 과정
  - 비어있는 큐를 처음 생성하면, Front와 Rear 모두 값이 없어서 -1
  - 원소 A가 삽입되면,

    Front는 여전히 -1이고 A가 들어간 위치가 Rear 로 지정된다
  - 위 과정을 반복해 큐에 여러 원소가 들어가게 된 상태에서,

    제일 앞 원소를 빼게 되면 그때 Front값이 +1 된다

- 주요 연산
  1. enQueue(item)

     : 큐의 Rear 다음에 원소를 삽입하는 연산
  2. deQueue( )

     : 큐의 Front 원소를 삭제하고 반환하는 연산
  3. isEmpty( )

     : 큐가 공백상태인지 확인하는 연산
  4. Qpeek( )

     : 큐의 Front 를 삭제하지는 않고, 반환하는 연산

선형 큐

- 제일 간단하고 기본적인 큐 형태로, 1차원 리스트 사용
- 큐의 크기는 곧 리스트의 크기

- 초기 상태는 Front = Rear = -1

  공백 상태는 Front = Rear

  포화 상태는 Rear = Queue_list[-1] 로 표현된다

- 선형 큐의 구현
  1. 초기 큐 생성 시 자동으로 빈 리스트 생성

```
class Queue():
  def __init__(self):
```

```
    self.queue = []
```

2. enqueue - 들어온 순서대로 쌓이니, append 이용

```
def enqueue(self, element):
  self.queue.append(element)
```

3. dequeue - 들어온 순서대로 나가야 하니, pop 에 값 지정해 사용

```
def dequeue(self):
  dequeue_object = None
  if self.isEmpty():
    print('Empty Queue')
  else:
    dequeue_object = self.queue.pop(0)

  return dequeue_object
```

4. Qpeek - 리스트의 첫 번째 원소 반환

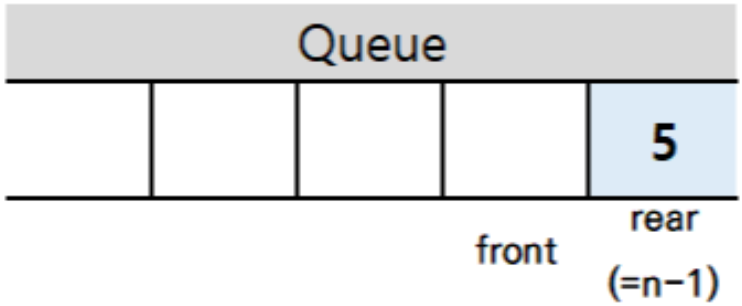
```
def Qpeek(self):
  Qpeek_object = None
  if self.isEmpty():
    print('Empty Queue')
  else:
    Qpeek_object = self.queue[0]

  return Qpeek_object
```

5. isEmpty - 리스트의 길이를 통해 판단

```
def isEmpty(self):
  is_empty = False
```

```
    if len(self.queue) == 0:
      is_empty = True

    return is_empty
```

- 선형 큐에서 삽입과 삭제를 반복하면, 결국 Rear는 마지막 인덱스를 가리키게 되고, 이렇게 되면 앞이 비어있더라도 포화라고 인식

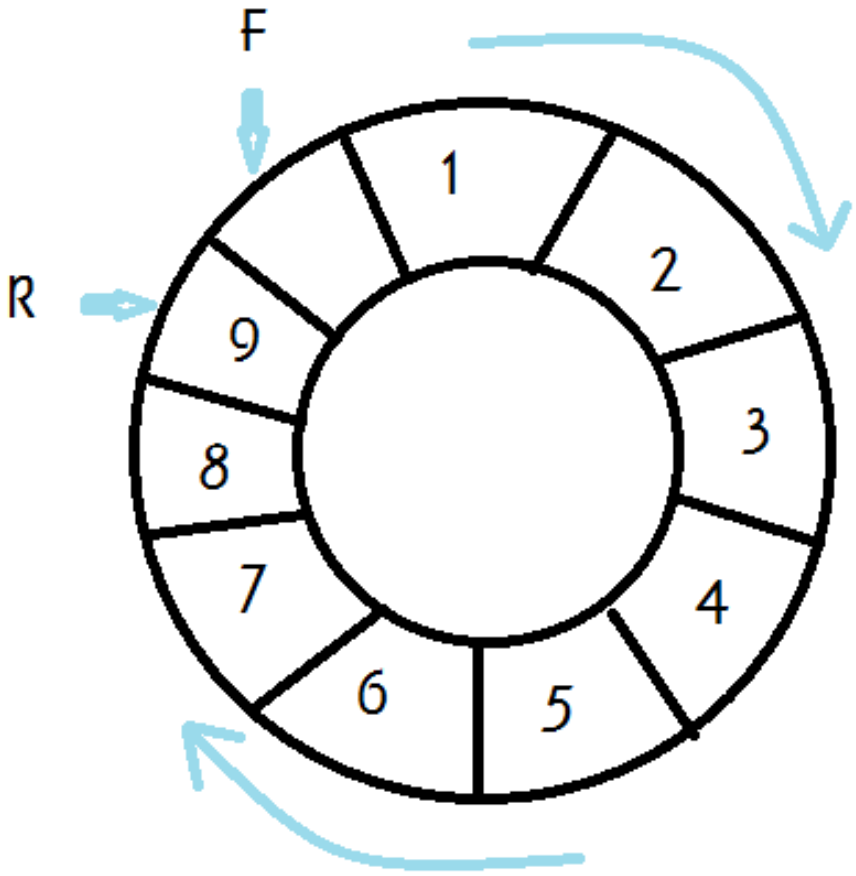


앞이 삭제되더라도 실제 데이터가 앞으로 이동하지 않고
인덱스 단위로 연산하며 Front와 Rear만 수정되었기 때문

원형 큐

- 선형 큐와 같이 1차원 리스트를 사용하는 것은 동일하지만,
  리스트의 처음과 끝이 연결되어 원형을 이룬다고 가정하고 사용

    - Front는 첫 원소 직전에, Rear는 마지막 원소가 있는 곳을 가리킨다



- 크기가 정해져있기 때문에, 선형 큐의 연산에 더해

  큐에 데이터가 꽉 차있는지를 판단하는 isFull 연산이 추가된다

- 원형 큐의 구현

  1. 초기 생성 시 큐의 크기와 Front / Rear 값, 현재 큐에 들어간 데이터 수를 의미할 count 변수 지정

```
class CircularQueue:
 def __init__(self, n):
   self.queue = [None] * n
   self.front = -1
   self.rear = -1
   self.count = 0
   self.maxcount = n
```

2. enqueue - 원소를 집어넣어주고, count와 rear에 1을 더해 큐 정보 수정

```
def enqueue(self, element):
  if self.isFull():
    print('Full Queue')
  else:
    # % 연산자를 통해 rear가 큐의 최대 크기를 넘어갈 경우 큐의 앞으로
    self.rear = (self.rear + 1) % self.maxcount
    self.queue[self.rear] = element
    self.count += 1
```

3. dequeue - 앞부분의 원소를 제거하고, count에서는 1을 빼주고
   front를 한칸 밀어 빈 칸을 다시 사용할 수 있도록 한다

```
def dequeue(self):
  if self.isEmpty():
    print('Empty Queue')
  else:
    # enqueue 때와 같이, front 값을 다시 큐의 앞으로 보내기 위한 것
    self.front = (self.front + 1) % self.maxcount
    self.queue[self.front] = element
    self.count -= 1
    return element
```

4. Qpeek - 첫 원소는 Front + 1 에 위치하므로, 이를 반환

```
def Qpeek(self):
  if self.isEmpty():
    print('Empty Queue')
  else:
    top = self.queue[(self.front + 1) % self.maxcount]
    return top
```

5. isEmpty - Front와 Rear가 같은 위치를 가리키면 비어있는 큐임을 이용

```
def isEmpty(self):
  return self.front == self.rear

# count 변수를 이용중이라면
def isEmpty(self):
  return self.count == 0
```

6. isFull - Rear + 1이 Front를 가리키면 가득 찬 큐임을 이용

```
def isFull(self):
  return self.front == (self.rear + 1) % self.maxcount

# count 변수를 이용중이라면
def isFull(self):
  return self.count == self.maxcount
```

- 일반적인 배열로 구현된 큐에서 dequeue를 수행할 때,
  제거와 더불어 남은 원소들을 앞으로 미는 작업 때문에 오래 걸릴 수 있다

  원형 큐는 크기는 정해져있고 Front와 Rear로 큐를 돌려가며 사용하기 때문에 dequeue를 수행할 때도 비교적 빠르게 수행될 수 있다

우선순위 큐

put, get 메서드 모두 O(log n)

- 선입선출 방식이 아니라, 원소들의 우선순위에 따라 순서를 정해 제거되는 큐

- 파이썬에서는 queue 내장 모듈의 PriorityQueue 클래스를 이용해 사용 가능

- 모듈을 이용한 우선순위 큐

  1. 우선순위 큐 생성

```
from queue import PriorityQueue

que = PriorityQueue([maxsize = n])

# maxsize 지정하지 않으면 최대 크기로 생성
```

  2. put 메서드를 사용해 우선순위 큐에 원소 추가

```
que.put(element1)
que.put(element3)
que.put(element2)
```

  3. get 메서드를 사용해 우선순위 큐에서 원소 제거

```
print(que.get())  # element1
print(que.get())  # element2  -  3이 먼저 들어갔으나, 2가 먼저 제거
```

  - 정렬 기준 변경

    단순 오름차순이 아니라, 우선순위를 이용하고 싶다면

    (우선순위, 값) 의 튜플 형태로 데이터를 추가하고 삭제

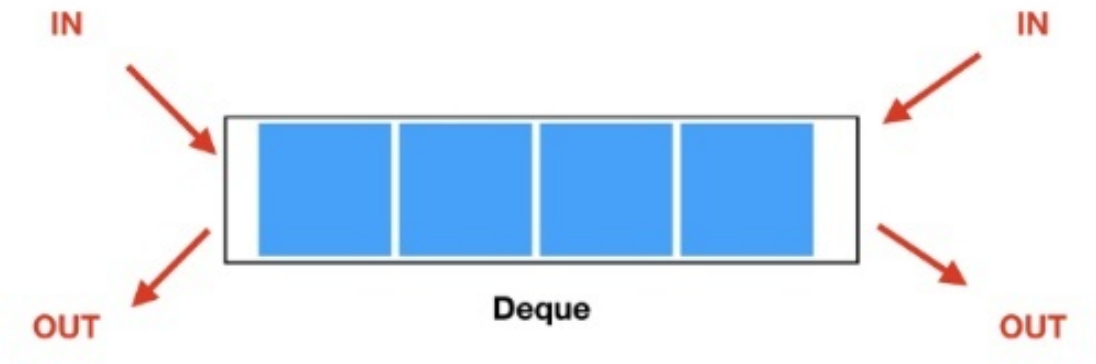
```
que.put((3, element2))
que.put((1, element1))
que.put((2, element3))

print(que.get()[1])    # element1
```

```
print(que.get()[1])    # element3  -  오름차순이었다면 2 였지만
                       #               우선순위 때문에 3이 먼저 제거
```

덱

append, pop 연산 모두 O(1)

- **D**ouble-**E**nded **QUE**ue
- 삽입과 제거가 큐 앞 / 뒤 모두에서 가능한 자료 구조
- 모든 특성을 가지고 있어 스택과 큐의 연산 모두 구현 가능



- 원형 큐에서 앞에서 더하고 뒤에서 빼는 기능이 추가로 구현
  - 앞에서 더할 때

    : Front에 요소를 넣고, front == 0 일때 나머지가 -1이 나오기 때문에

    self.front = (self.front -1 + self.maxcount) % self.maxcount 사용
  - 뒤에서 뺄 때

    : rear 자리의 요소를 빼고, rear == 0 일때도 나머지가 -1 이기 때문에

    self.rear = (self.rear - 1 + self.maxcount) % self.maxcount 사용

- 위처럼 덱을 원형 큐를 기반으로 구현할 수도 있지만, 내장 모듈도 존재한다

  1. 덱 생성

```
from collections import deque

deque([iterable [, maxlen]])

# iterable 데이터를 지정해주지 않으면 빈 덱 생성
# maxlen이 지정되지 않거나 None 이면, 덱은 임의의 길이로 커지기 가능
```

  2. append(element), appendleft(element)

     덱의 오른쪽에 요소 추가, appendleft는 덱의 왼쪽에 추가

     - 크기 제한을 둔 덱이 가득 차면, 새 항목 추가될 시 반대쪽 요소 빠짐

       꽉 찼을 때 append 하면 deque[0]이, appendleft 하면 deque[-1] 빠짐

  3. pop( ), popleft( )

     덱의 제일 오른쪽 요소 제거, popleft는 왼쪽 요소 제거

  4. insert(idx, element)

     입력한 인덱스에 요소를 추가

     만약 이로 인해 최대 크기를 초과하게 되면 IndexError

  5. extend(iterable), extendleft(iterable)

     주어진 iterable 데이터를 덱의 오른쪽을 확장하며 추가

     extendleft 시, 왼쪽에 추가되는 iterable 는 역순으로 삽입

     - extendleft([1, 2, 3]) → [3, 2, 1] 을 왼쪽에 삽입

6. index(element [, start [, stop]])

   덱 안에 있는 element의 위치 반환

   start와 stop에 인덱스를 넣어 탐색 범위 지정도 가능

   값이 없을 시 ValueError

7. rotate(n)

   덱을 n 만큼 오른쪽으로 회전, n이 음수면 왼쪽으로 회전

   ‘밀어낸다’ 라고 생각해도 될 것 같음

8. clear( )

   덱의 모든 요소를 제거해 길이가 0 인 상태로 만든다

9. remove(element)

   첫 번째로 나오는 element 제거, 못 찾으면 ValueError

10. reverse( )

    덱의 요소들을 순서를 뒤집으며 None 을 반환

11. count(element)

    덱 안의 element 의 수를 반환

12. maxlen

    덱의 최대 크기를 반환, 제한이 없다면 None 반환