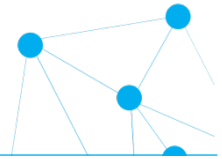

파이썬
자료구조

CHAPTER

큐와 덱

5.1 큐란?



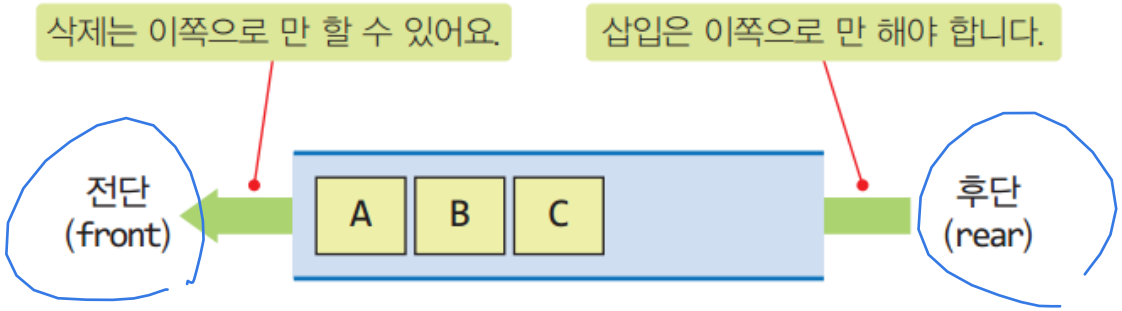
- 큐는 선입선출(First-In First Out: FIFO)의 자료구조이다.
  - 큐의 구조
  - 큐의 ADT
  - 큐의 연산

- 큐의 응용

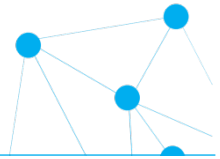
큐(QUEUE)란?

- 큐는 선입선출(First-In First Out: FIFO)의 자료구조

- 큐의 구조



큐 ADT



- 삽입과 삭제는 FIFO순서를 따른다.
- 삽입은 큐의 후단에서, 삭제는 전단에서 이루어진다.

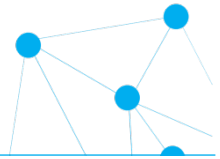
정의 5.1 Queue ADT

데이터: 선입선출(FIFO)의 접근 방법을 유지하는 항목들의 모음

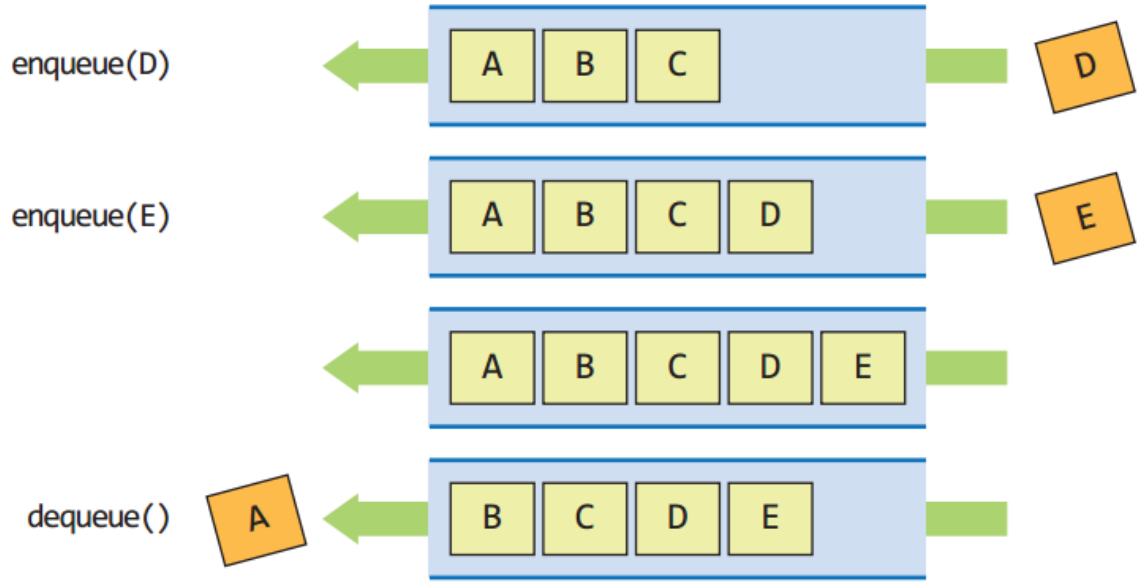
연산

- Queue(): 비어 있는 새로운 큐를 만든다.
- isEmpty(): 큐가 비어있으면 True를 아니면 False를 반환한다.
- enqueue(x): 항목 x를 큐의 맨 뒤에 추가한다.
- dequeue(): 큐의 맨 앞에 있는 항목을 꺼내 반환한다.
- peek(): 큐의 맨 앞에 있는 항목을 삭제하지 않고 반환한다.
- size(): 큐의 모든 항목들의 개수를 반환한다.
- clear(): 큐를 공백상태로 만든다.

큐의 연산

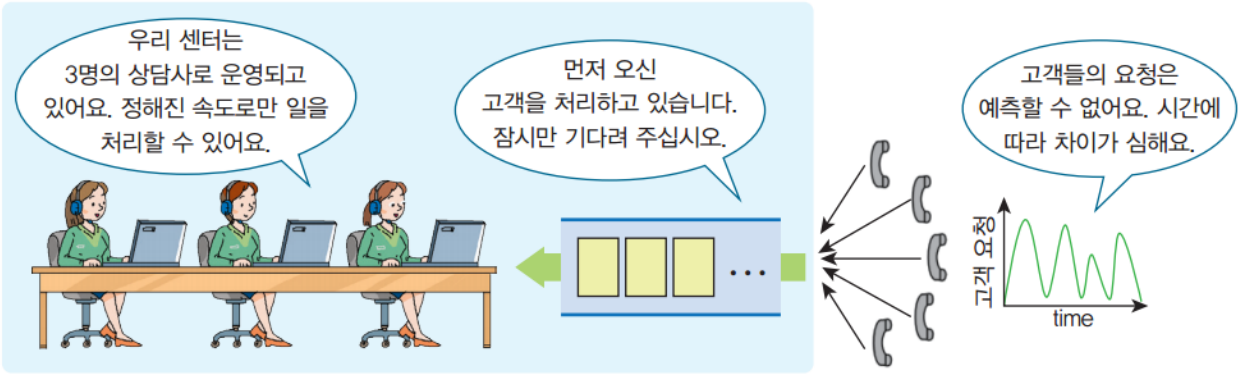


- 삽입: enqueue()
- 삭제: dequeue()



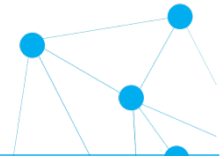
큐의 응용

- 예) 서비스센터의 콜 큐



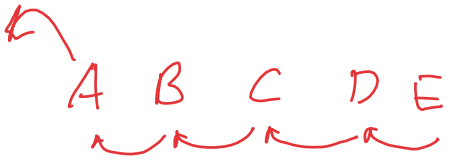
- 컴퓨터에서도 큐는 매우 광범위하게 사용
  - 프린터와 컴퓨터 사이의 인쇄 작업 큐 (버퍼링)
  - 실시간 비디오 스트리밍에서의 버퍼링
  - 시뮬레이션의 대기열(공항의 비행기들, 은행에서의 대기열)
  - 통신에서의 데이터 패킷들의 모델링에 이용

5.2 큐의 구현



- 선형 큐에는 어떤 문제가 있을까?
- 원형 큐가 훨씬 효율적이다.
- 원형 큐의 구현

선형큐

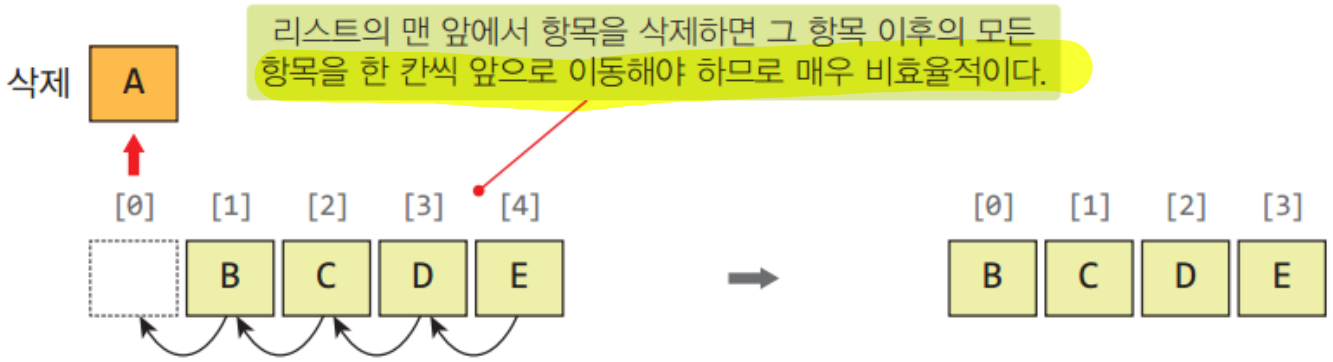


- 선형큐는 비효율적이다.
  - enqueue(item): 삽입 연산

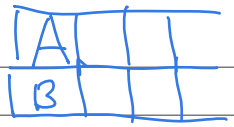
```
def enqueue(item):
    items.append(item)          # 리스트의 맨 뒤에 items 추가
```

  - dequeue(): 삭제 연산 why? ⇒ O(n)

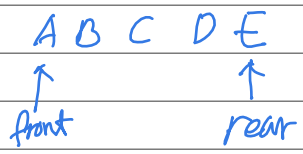
```
def dequeue():
    if not isEmpty():           # 공백상태가 아니면
        return items.pop(0)     # 맨 앞 항목을 꺼내서 반환
```



enque (A)
enque (B)

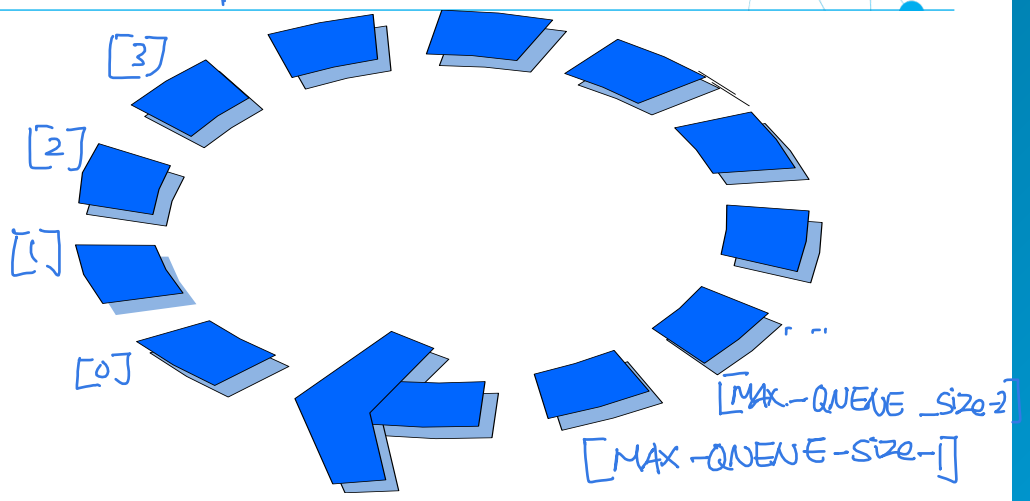


미로를 많이 사용하게 됨.



원형 큐가 훨씬 효율적이다. – 구현 1

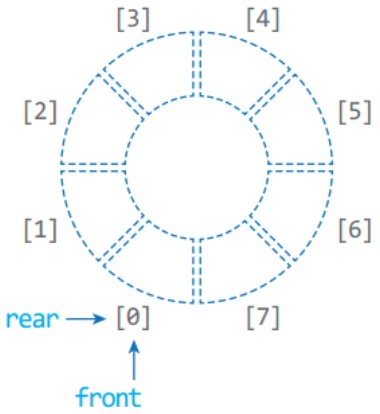
- 원형큐
  - 배열을 원형으로 사용



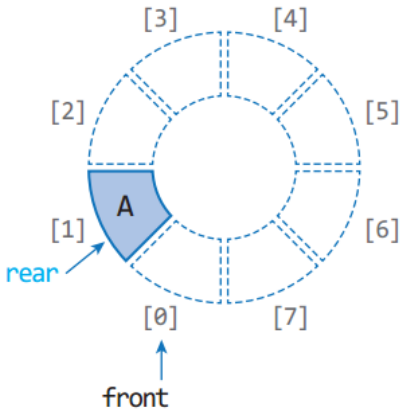
- 전단과 후단을 위한 2개의 변수
  - front: 큐의 첫번째 요소 바로 이전의 인덱스
  - Rear: 큐의 마지막 요소의 인덱스
- 회전(시계방향) 방법

```
front ← (front+1) % MAX_QSIZE
rear ← (rear +1) % MAX_QSIZE
```

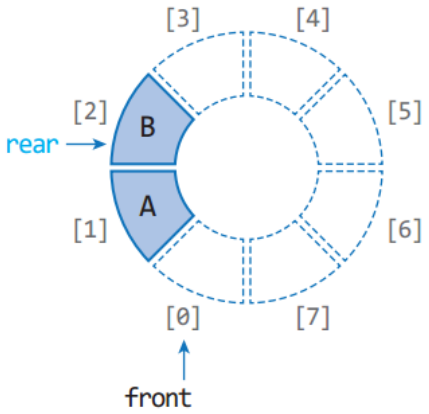
원형 큐의 삽입과 삭제 과정



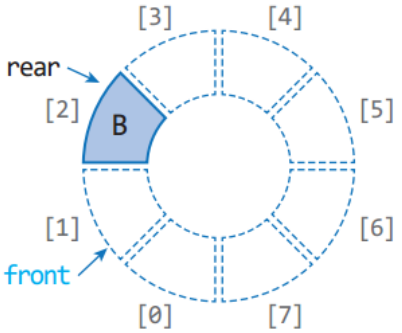
초기상태



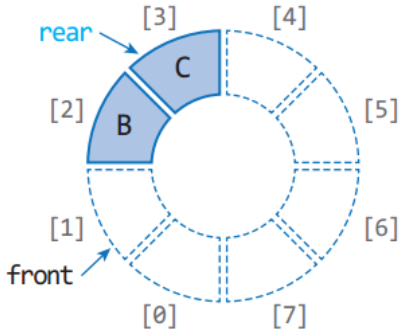
enqueue(A)



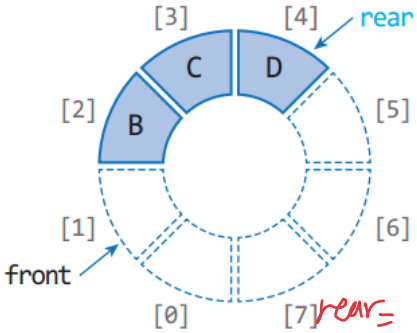
enqueue(B)



dequeue()



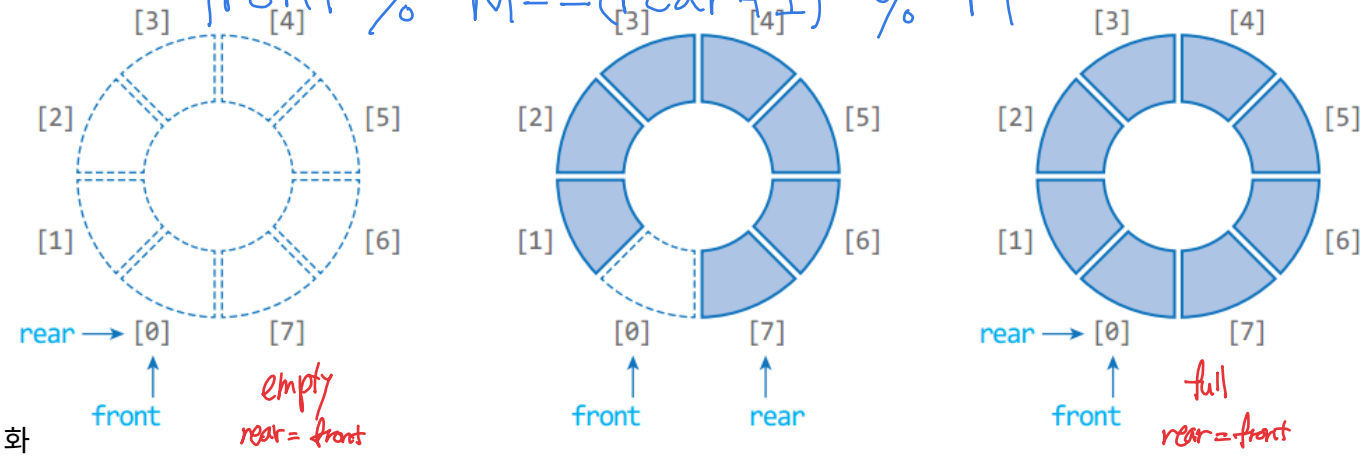
enqueue(C)



enqueue(D)

공백상태와 포화상태 – (1)

- 공백상태와 포화상태를 구별 방법 (1)
  - 하나의 공간은 항상 비워둠
- 공백상태: front == rear
- 포화상태: front % M==(rear+1) % M



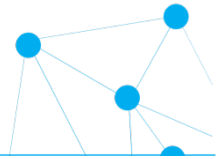
front, rear 초기화

front : 0
rear : 0

혹은

front : -1
rear : -1

공백상태와 포화상태 – (2)



- 공백상태와 포화상태 구별 방법 (2)

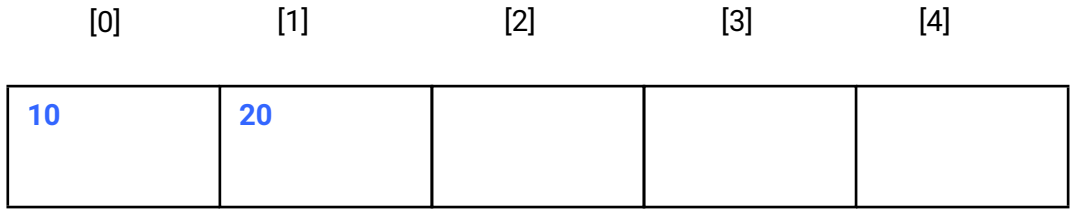
  멤버 변수 size를 추가:

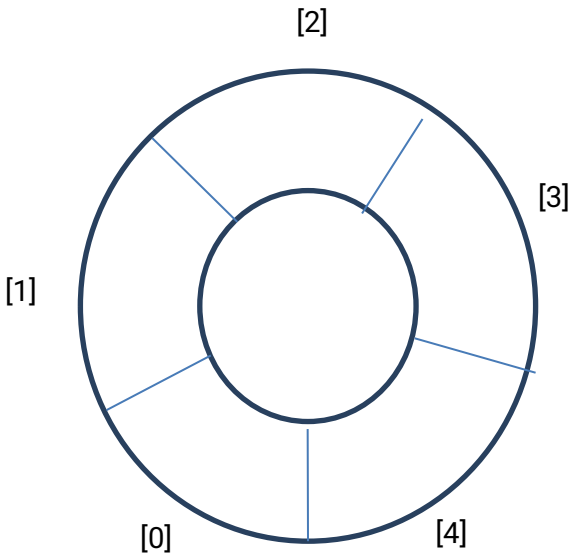
  size: 큐에 있는 요소의 개수

  공백상태: size = 0

  포화상태: size == 리스트(배열) 길이

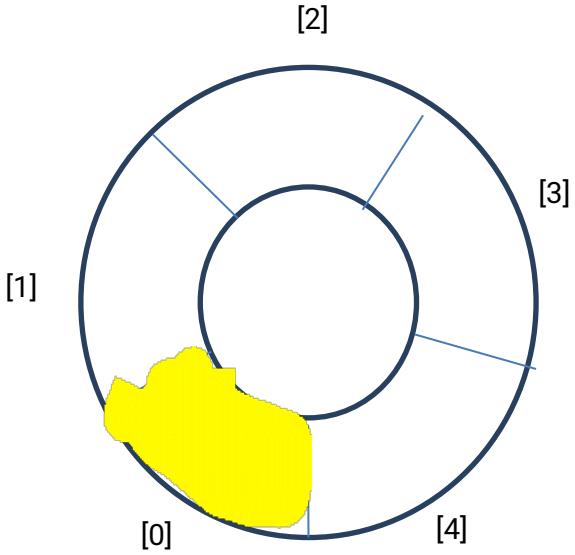
리스트(배열)
Items
길이 = 5





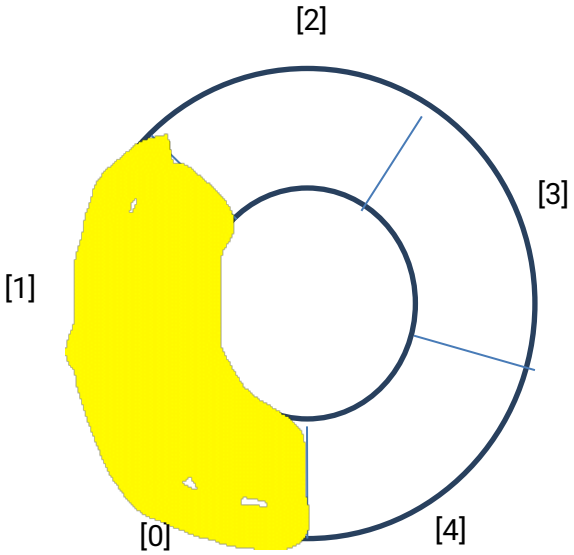
front : -1
rear : -1
size : 0

enqueue(10)



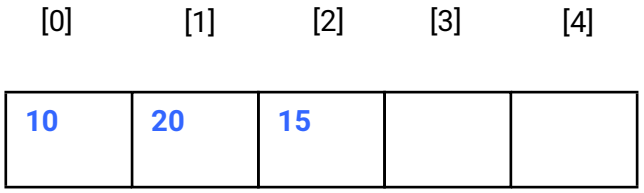
front : -1
rear : 0
size : 1

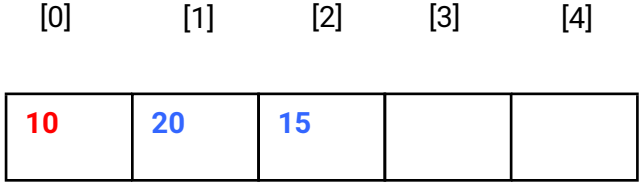
enqueue(20)

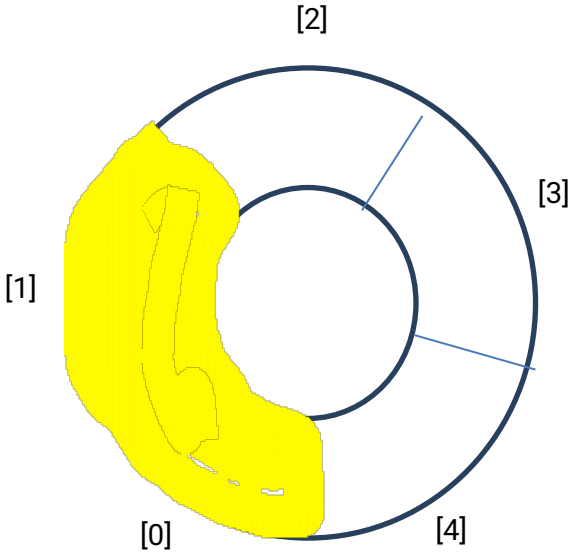


front : -1
rear : 1
size : 2

리스트(배열)
Items
길이 = 5

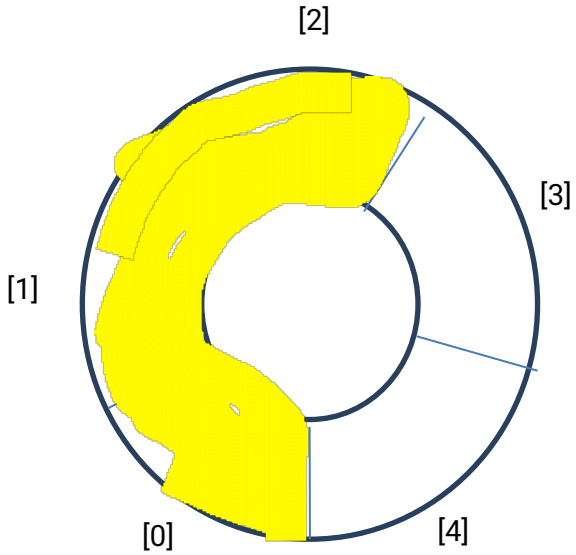






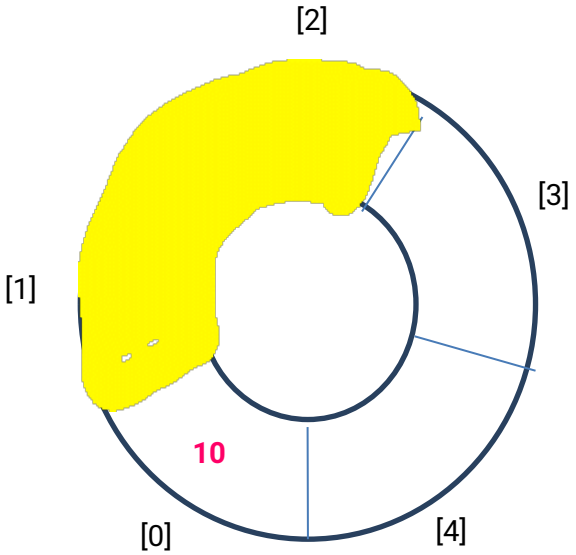
front : -1
rear : 1
size : 2

enqueue(15)



front : -1
rear : 2
size : 3

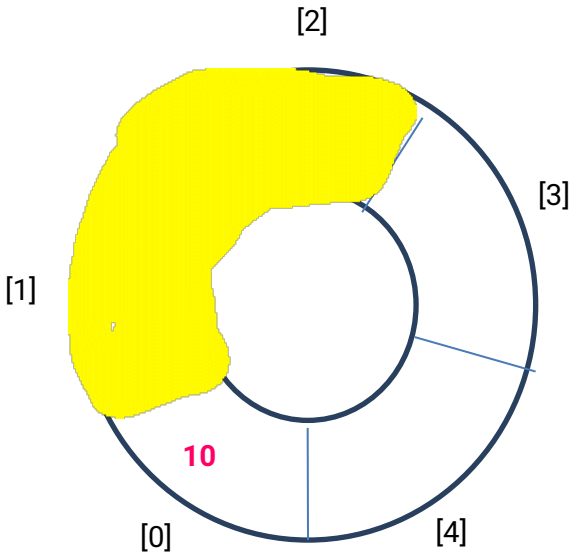
dequeue()
=> 10



front : 0
rear : 2
size : 2

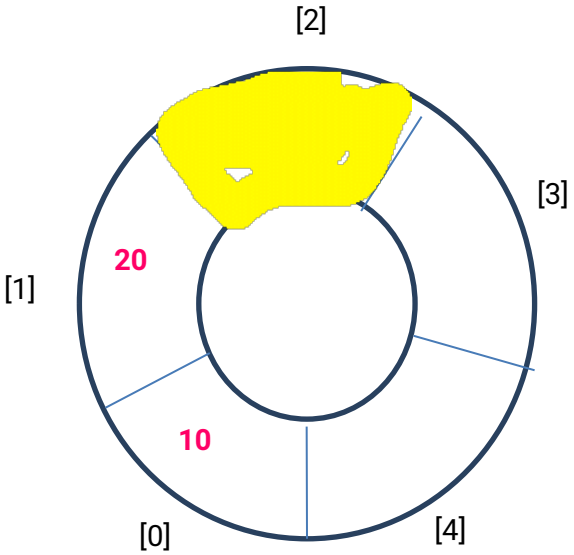
리스트(배열)
Items
길이 = 5

[0]	[1]	[2]	[3]	[4]
10	20	15		



front : 0
rear : 2
size : 2

dequeue()
=> 20

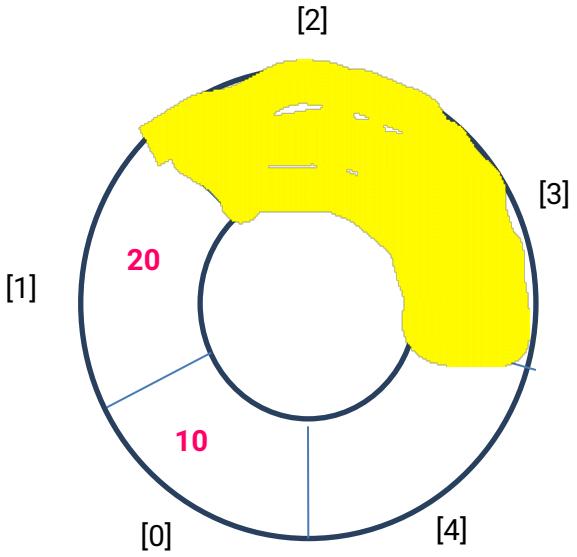


front : 1
rear : 2
size : 1

front 다음부터 원소 삽입을 시작함.

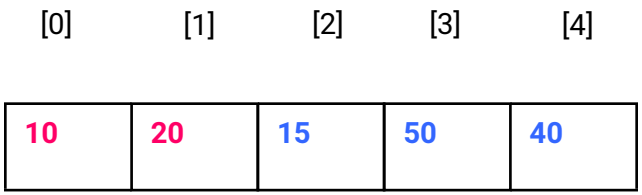
enqueue(50)

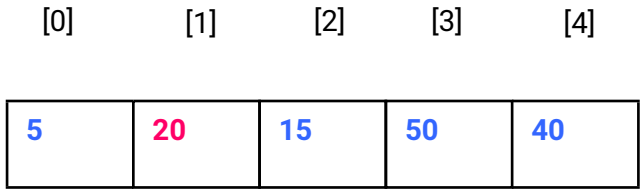
[0]	[1]	[2]	[3]	[4]
10	20	15	50	

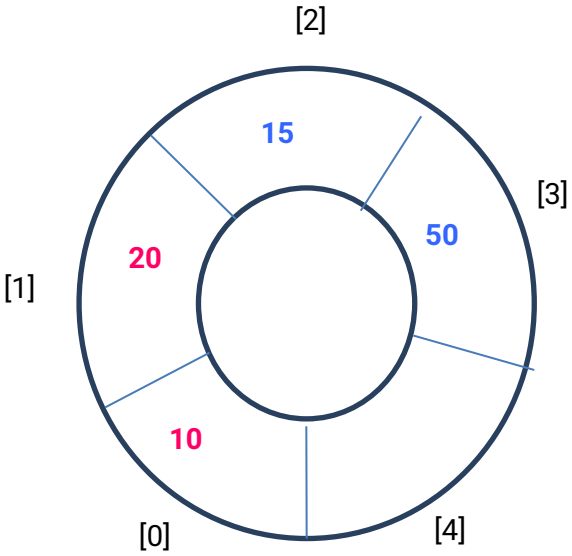


front : 1
rear : 3
size : 2

리스트(배열)
Items
길이 = 5

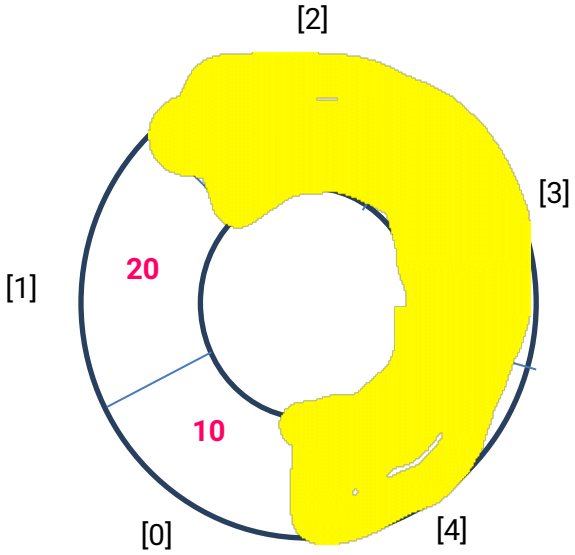






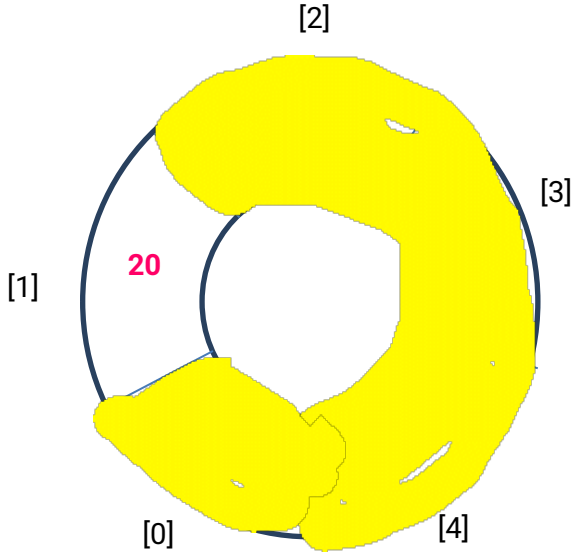
front = 1
rear = 3
size = 2

enqueue(40)



front = 1
rear =4
size = 3

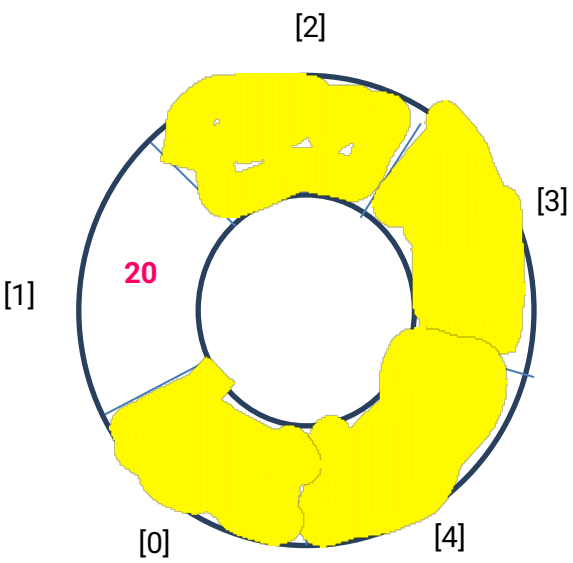
enqueue(5)



front = 1
rear = 0
size = 4

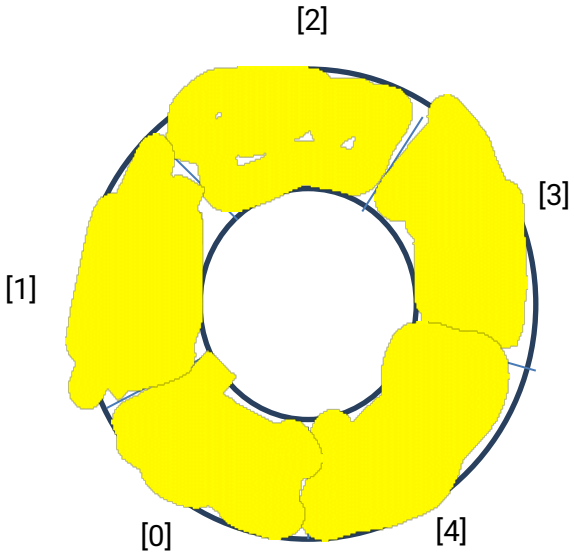
리스트(배열)
Items
길이 = 5

[0]	[1]	[2]	[3]	[4]
5	10	15	50	40



front = 1
rear = 0
size = 4

enqueue(10)



front = 1
rear = 1
size = 4

enqueue(30)
=> Queue Full

원형큐 구현(클래스)

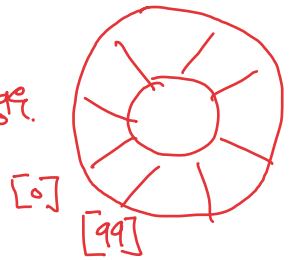
```
class Queue:
    MAX_QSIZE = 100   # 새로운 모든 큐에 대한 적당한 크기 설정
    def  __init__(self):
        self.items = [None]*Queue.MAX_QSIZE
        self.front = -1
        self.rear =-1
        self.size = 0

    def isEmpty(self):
        return self.size == 0
```

원형큐 구현(클래스)

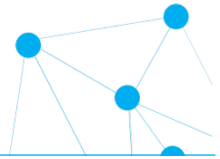
```
def isEmpty(self):
    return self.size == 0
```

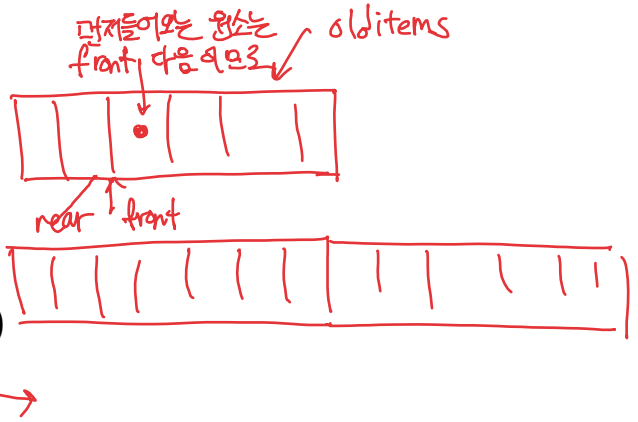
```
def dequeue(self):
    if self.isEmpty():
        print("Queue is empty")
    else:
        self.front = (self.front+1)%(len(self.items))
        e = self.items[self.front]
        self.size -= 1
        return e
```

```
def enqueue(self, e):
    if self.size == len(self.items):
        print("Queue is full")
#           self.resize(2*len(self.items))
    else:
        self.rear = (self.rear+1)%(len(self.items))
        self.items[self.rear] = e
        self.size += 1
```



원형큐 구현(클래스)



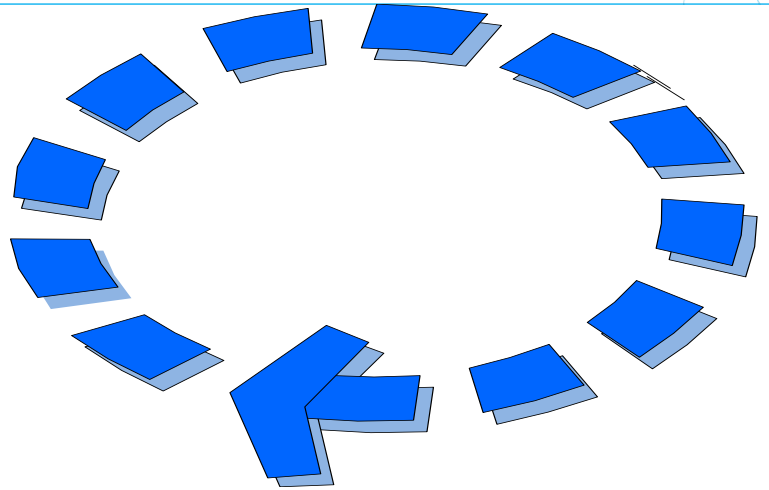



```
def resize(self, cap):
        olditems = self.items
        self.items = [None]*cap
        walk = self.front(self + 1) % len(olditems)
        for k in range(self.size):
            self.items[k] = olditems[walk]
            walk = (walk + 1)% len(olditems)
        self.front = -1
        self.rear = self.size - 1
```

~~원형 큐 – 구현 2~~

구현2 x 원형큐는 1로만 사용

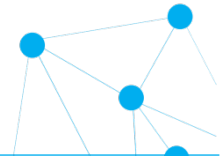
- 원형큐
  - 배열을 원형으로 사용



- 전단과 후단을 위한 2개의 변수
  - front: 큐에 삽입된 첫번째 요소의 인덱스
  - rear: 큐에 삽입된 마지막 요소 바로 다음의 인덱스
- 회전(시계방향) 방법

```
front ← (front+1) % MAX_QSIZE
rear ← (rear +1) % MAX_QSIZE
```

공백상태와 포화상태



- 공백상태와 포화상태 구별 방법

  멤버 변수 size를 추가:

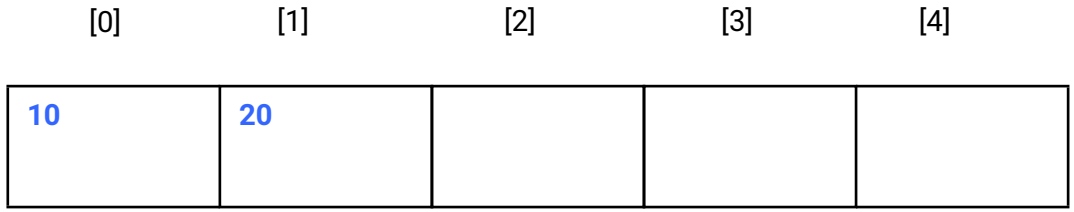
  size: 큐에 있는 요소의 개수

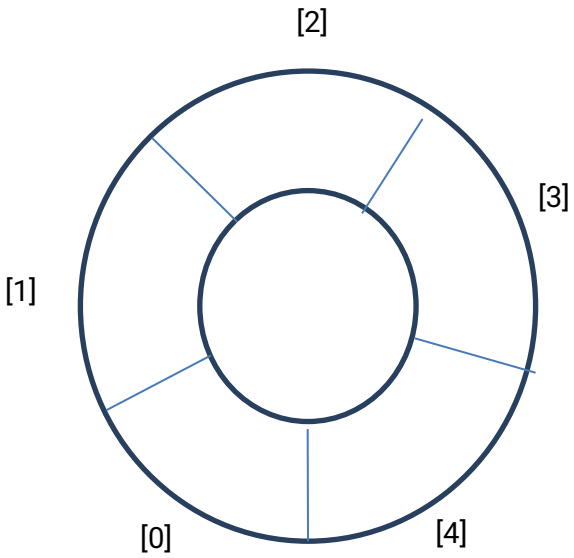
  공백상태: size = 0

  포화상태: size == 리스트(배열) 길이

0

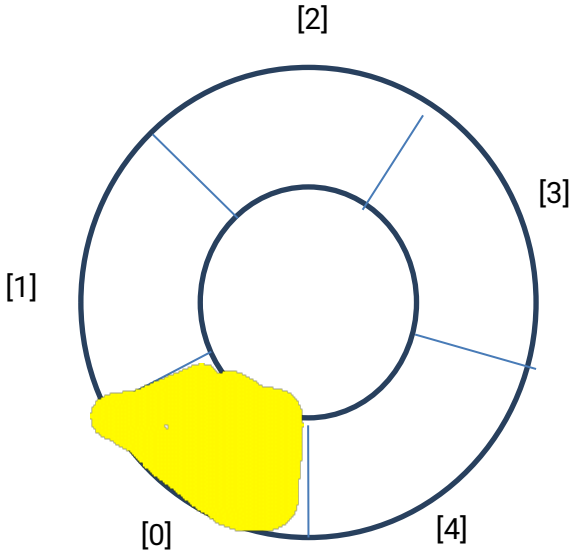
리스트(배열)
Items
길이 = 5





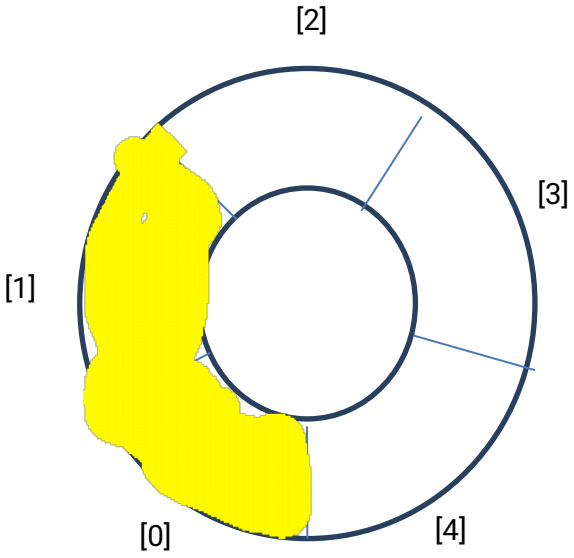
front : 0
rear : 0
size : 0

enqueue(10)



front : 0
rear : 1
size : 1

enqueue(20)

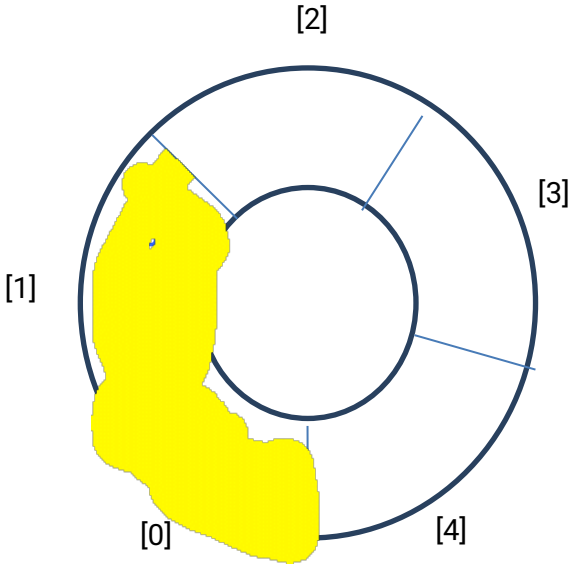


front : 0
rear : 2
size : 2

리스트(배열)
Items
길이 = 5

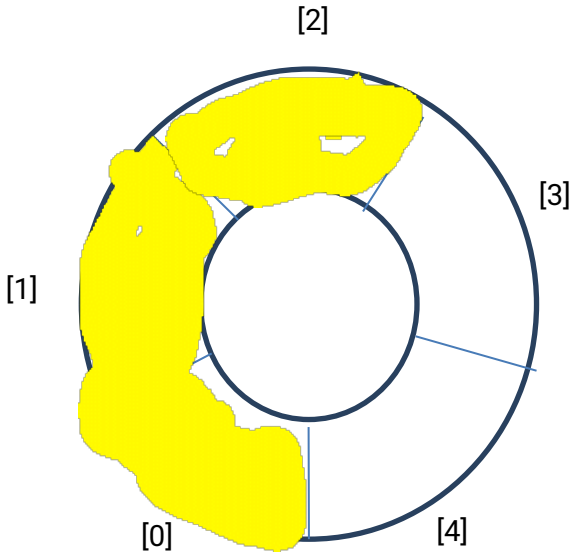
[0]	[1]	[2]	[3]	[4]
10	20	15		

[0]	[1]	[2]	[3]	[4]
10	20	15		



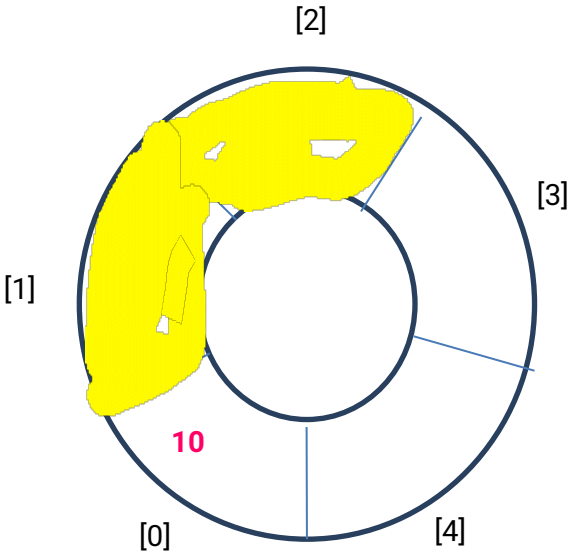
front : 0
rear : 2
size : 2

enqueue(15)



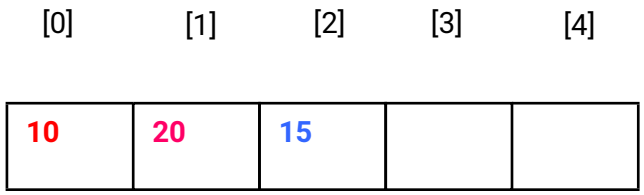
front : 0
rear : 3
size : 3

dequeue()
=> 10

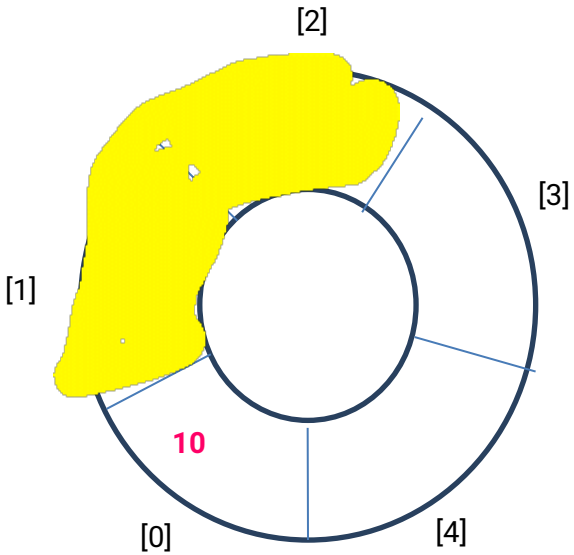


front : 1
rear : 3
size : 2

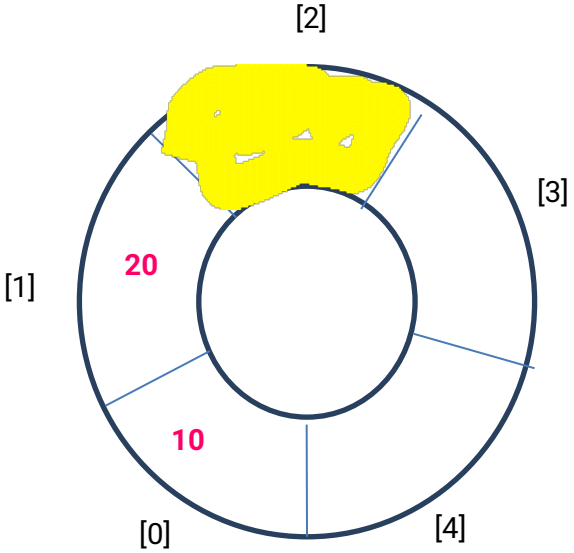
리스트(배열)
Items
길이 = 5



front : 1
rear : 3
size : 2

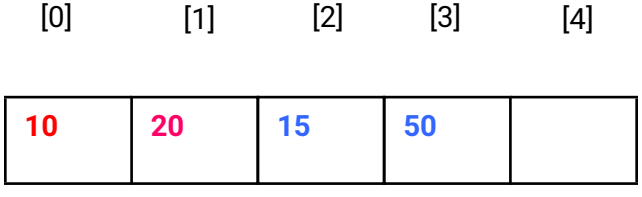


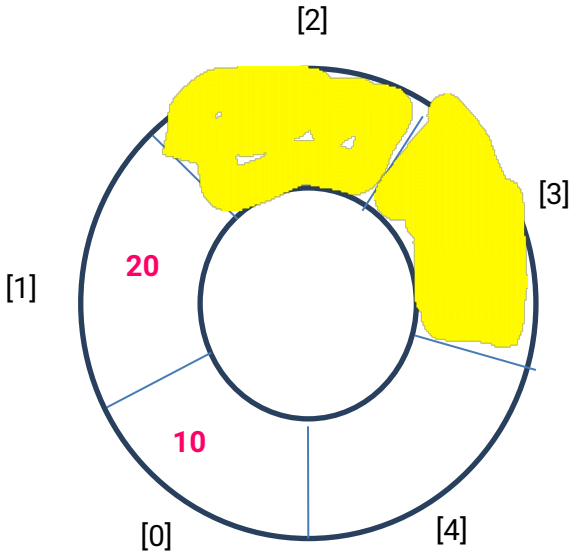
dequeue()
=> 20



front : 2
rear : 3
size : 1

enqueue(50)



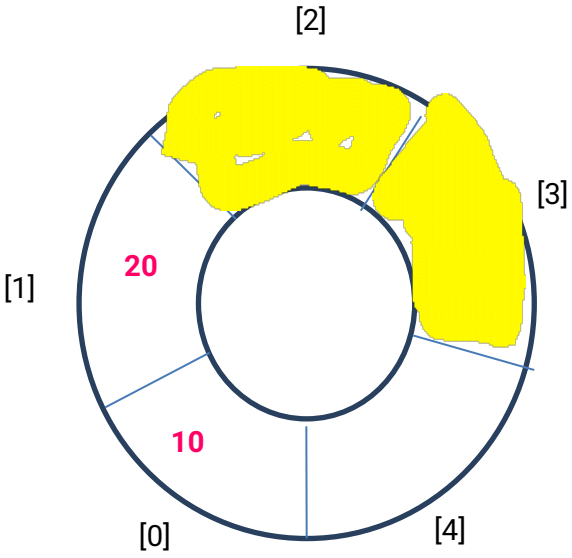


front := 2
rear : 4
size : 2

리스트(배열)
Items
길이 = 5

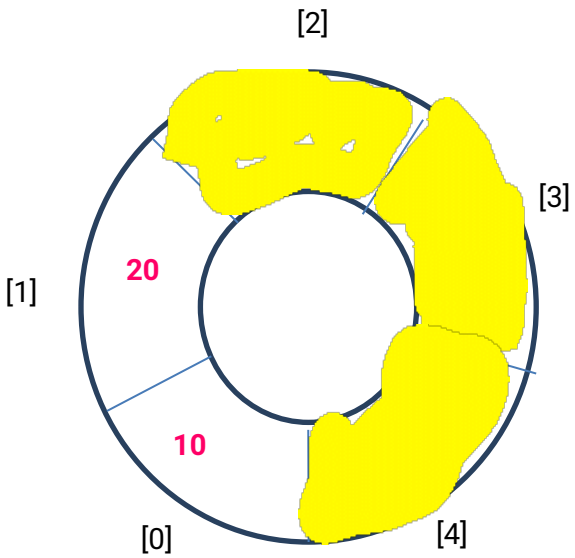
[0]	[1]	[2]	[3]	[4]
10	20	15	50	40

[0]	[1]	[2]	[3]	[4]
5	20	15	50	40



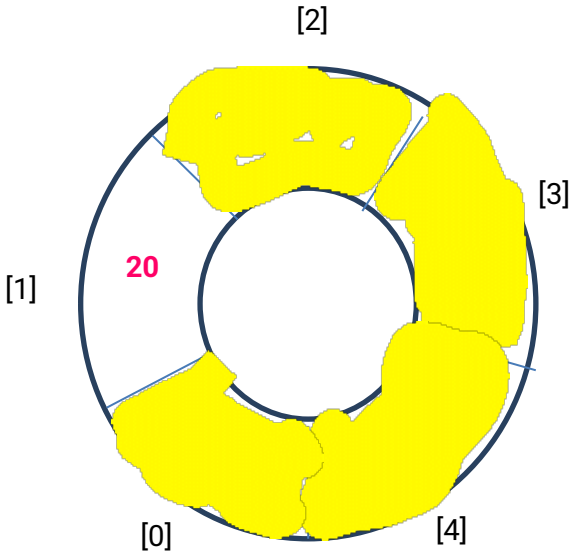
front : 2
rear : 4
size : 2

enqueue(40)



front : 2
rear : 0
size : 3

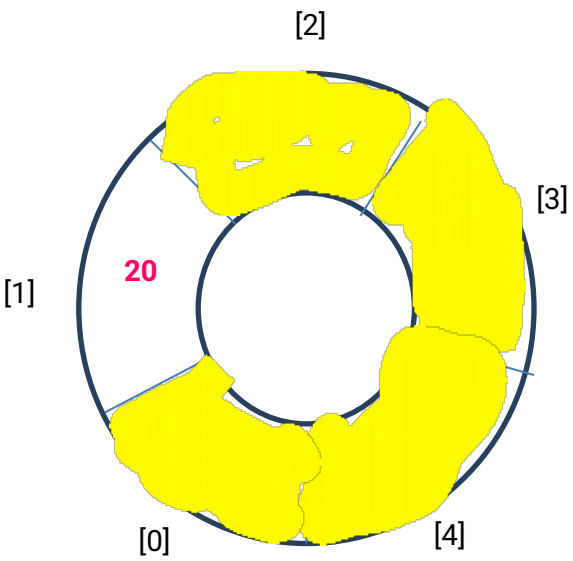
enqueue(5)



front : 2
rear : 1
size : 4

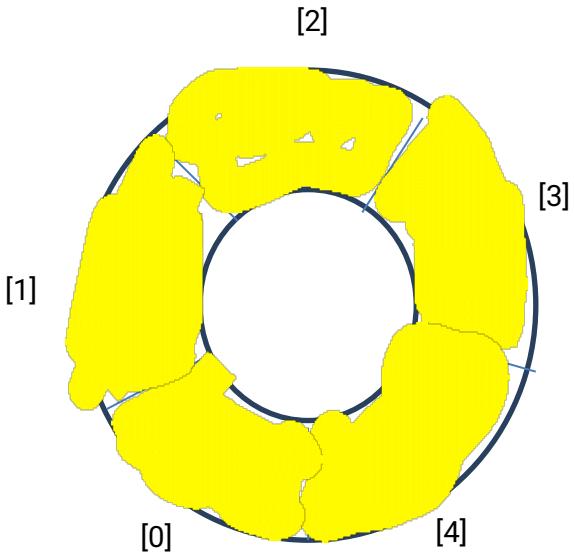
리스트(배열)
Items
길이 = 5

[0]	[1]	[2]	[3]	[4]
5	10	15	50	40



front : 2
rear : 1
size : 4

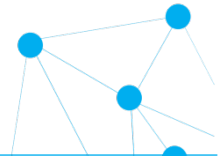
enqueue(10)



front : 2
rear : 2
size : 5

enqueue(30)
=> Queue Full

원형큐 구현(클래스) - 2



```
class Queue:
    MAX_QSIZE = 100
    def __init__(self):
        self.items = [None]*Queue.MAX_QSIZE
        self.front = 0
        self.rear = 0
        self.size = 0

    def isEmpty(self):
        return self.front(self.items) == 0
```

self.size 변수 없을경우 size가 있을 경우.

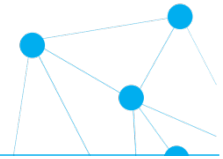
front rear에서 다른 인덱스를 가리켜야함. 배열의 크기만큼 넣을 수 있음.

하지만

self.size == 0

더이상 넣을 수 없는 상태 포화상태.

원형큐 구현(클래스)

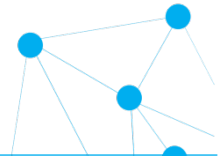


```
def isEmpty(self):
    return self.size == 0
```

```
def dequeue(self):
    if self.isEmpty():
        print("Queue is empty")
    else:
        e = self.items[self.front]
        self.front = (self.front+1)%(len(self.items))
        self.size -= 1
        return e
```

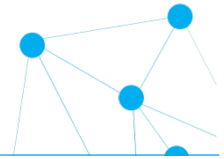
```
def enqueue(self, e):
    if self.size == len(self.items):
        print("Queue is full")
#           self.resize(2*len(self.items))
    else:
        self.items[self.rear] = e
        self.rear = (self.rear+1)%(len(self.items))
        self.size += 1
```

원형큐 구현(클래스)



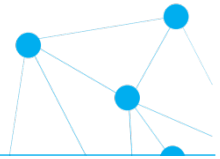
```
def resize(self, cap):
        olditems = self.items
        self.items = [None]*cap
        walk = self.front
        for k in range(self.size):
            self.items[k] = olditems[walk]
            walk = (walk + 1)% len(olditems)
        self.front = 0
        self.rear = self.size
```

테스트 프로그램



```
q = Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
e = q.dequque()
print(e)
q.enqueue(40)
```

5.3 파이썬의 queue 모듈



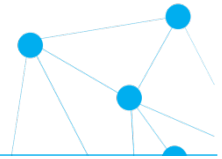
- 파이썬의 queue모듈은 큐 클래스를 제공한다.

- import queue

```
# methods:
      empty()
      put(e)    # enqueue
      get()     # dequeue
      qsize()
```

```
q = queue.Queue()   // 큐 객체 생성
q.put(20)     // 큐 원소 삽입 (enqueue)
e = q.get()   // 큐 원소 삭제 (dequeue)
```

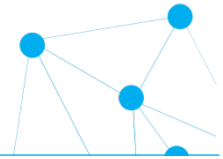
5.4 덱(deque)이란?



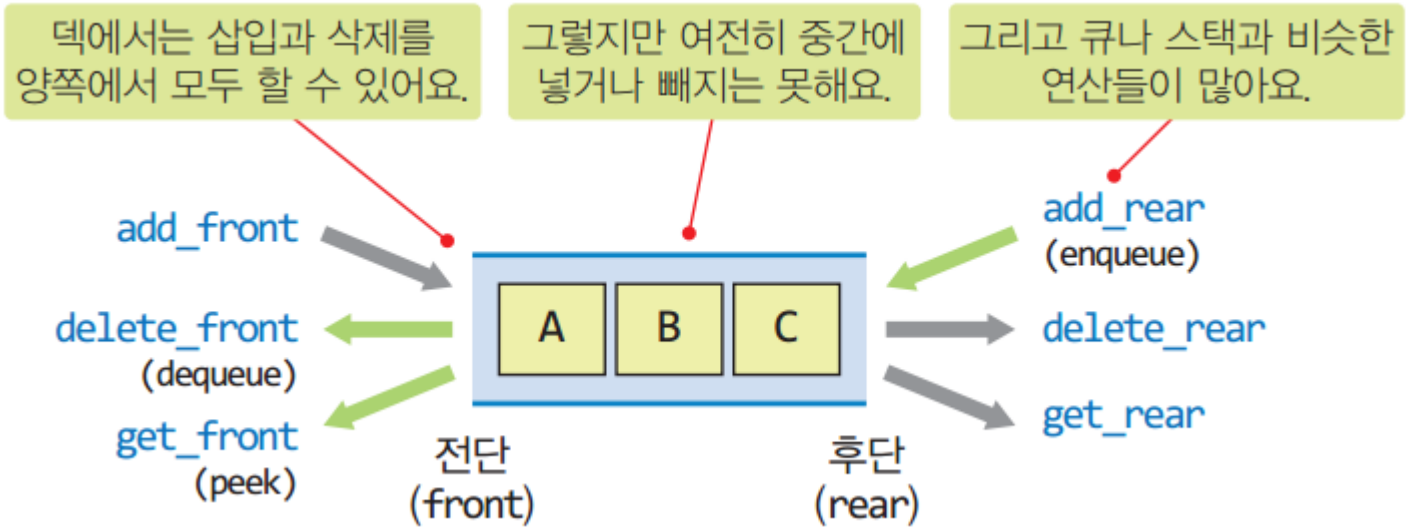
- 덱은 스택이나 큐보다는 입출력이 자유로운 자료구조이다.
  - 덱의 구조
  - 덱 ADT
  - 덱의 연산

- 덱을 스택이나 큐로 사용할 수 있다.

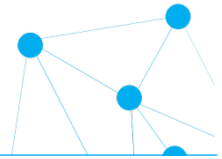
덱의 구조



- 스택이나 큐보다 입출력이 자유로운 자료구조
- 덱(deque)은 double-ended queue의 줄임말
  - 전단(front)와 후단(rear)에서 모두 삽입과 삭제가 가능한 큐



덱 ADT



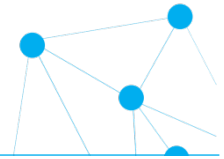
정의 5.2 Deque ADT

데이터: 전단과 후단을 통한 접근을 허용하는 항목들의 모음

연산

- `Deque()`: 비어 있는 새로운 덱을 만든다.
- `isEmpty()`: 덱이 비어있으면 `True`를 아니면 `False`를 반환한다.
- `addFront(x)`: 항목 `x`를 덱의 맨 앞에 추가한다.
- `deleteFront()`: 맨 앞의 항목을 꺼내서 반환한다.
- `getFront()`: 맨 앞의 항목을 꺼내지 않고 반환한다.
- `addRear(x)`: 항목 `x`를 덱의 맨 뒤에 추가한다.
- `deleteRear()`: 맨 뒤의 항목을 꺼내서 반환한다.
- `getRear()`: 맨 뒤의 항목을 꺼내지 않고 반환한다.
- `isFull()`: 덱이 가득 차 있으면 `True`를 아니면 `False`를 반환한다.
- `size()`: 덱의 모든 항목들의 개수를 반환한다.
- `clear()`: 덱을 공백상태로 만든다.

파이썬의 dequeue 모듈



- from collections import deque

- dq = deque() // deque 객체 생성
- Methods:

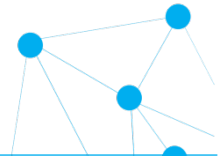
  append

  pop

  appendleft

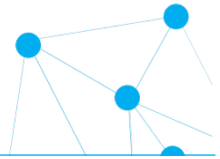
  popleft

5.6 우선순위 큐



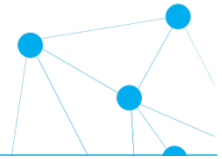
- 우선순위 큐란?
- 우선순위 큐 ADT
- 정렬되지 않은 배열을 이용한 우선순위 큐의 구현
- 시간 복잡도

우선순위 큐란?



- 실생활에서의 우선순위
  - 도로에서의 자동차 우선순위
- 우선순위 큐(priority queue)
  - 우선순위 의 개념을 큐에 도입한 자료구조
  - 모든 데이터가 우선순위를 가짐
  - 입력 순서와 상관없이 우선순위가 높은 데이터가 먼저 출력
  - 가장 일반적인 큐로 볼 수 있다. Why?
- 응용분야
  - 시뮬레이션, 네트워크 트래픽 제어, OS의 작업 스케쥴링 등

우선순위 큐 ADT



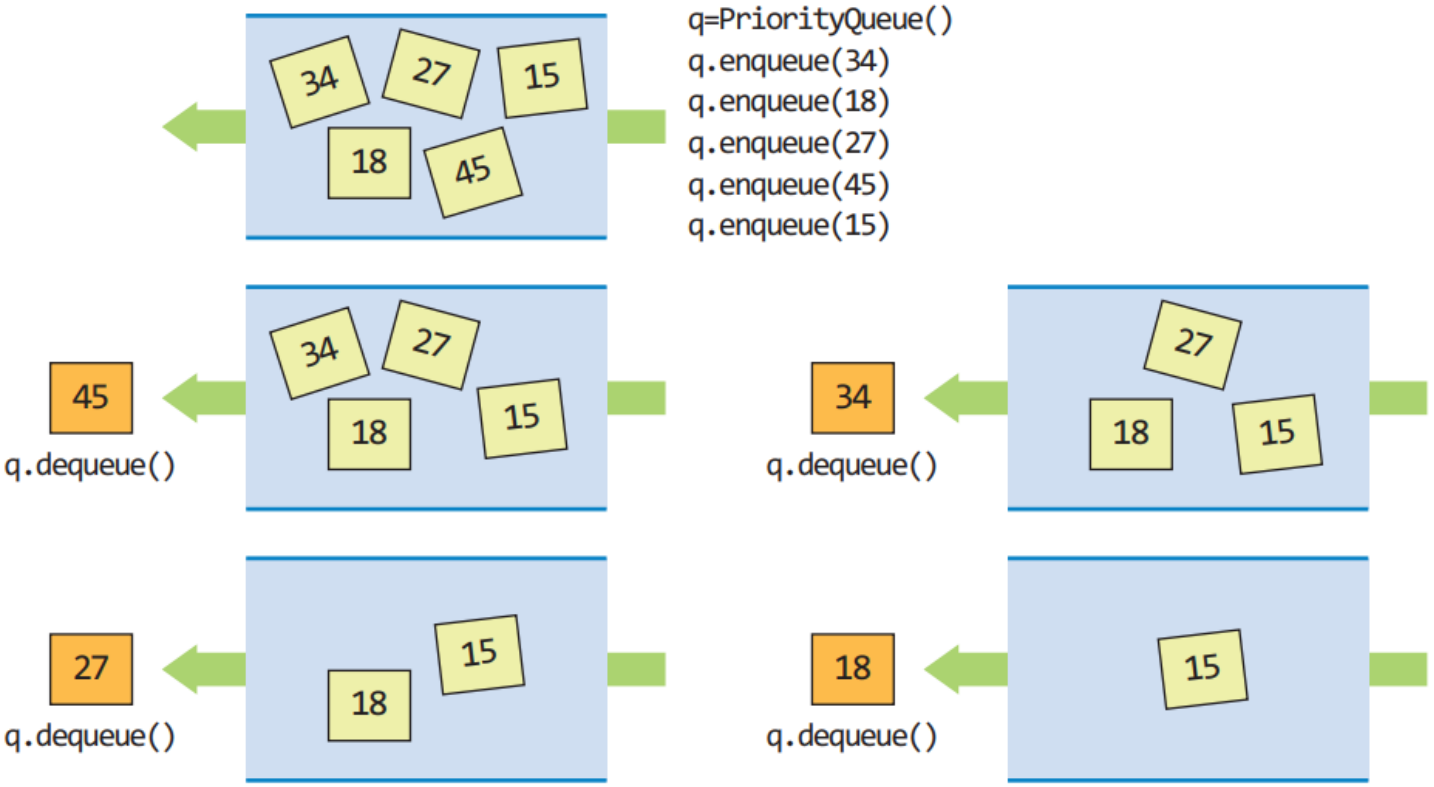
정의 5.3 Priority Queue ADT

데이터: 우선순위를 가진 요소들의 모음

연산

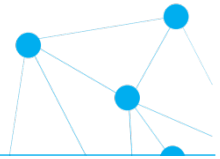
- `PriorityQueue()`: 비어 있는 우선순위 큐를 만든다.
- `isEmpty()`: 우선순위 큐가 공백상태인지를 검사한다.
- `enqueue(e)`: 우선순위를 가진 항목 e를 추가한다.
- `dequeue()`: 가장 우선순위가 높은 항목을 꺼내서 반환한다.
- `peek()`: 가장 우선순위가 높은 요소를 삭제하지 않고 반환한다.
- `size()`: 우선순위 큐의 모든 항목들의 개수를 반환한다.
- `clear()`: 우선순위 큐를 공백상태로 만든다.

우선순위 큐의 삽입과 삭제 연산



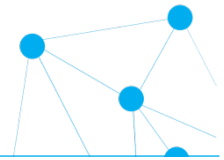
- 구현 방법: 배열 구조/연결된 구조/힙 트리

정렬되지 않은 배열을 이용한 구현



```
# Python list를 이용한 Priority Queue ADT 구현.
class PriorityQueue :
    def __init__( self ):                      # 생성자
        self.items = []                        # 항목 저장을 위한 리스트

    def isEmpty( self ):                       # 공백상태 검사
        return len( self.items ) == 0
    def size( self ): return len(self.items)   # 전체 항목의 개수
    def clear( self ): self.items = []         # 초기화

    def enqueue( self, item ):          # 삽입 연산
        self.items.append( item )       # 리스트의 맨 뒤에 삽입(O(1))
```

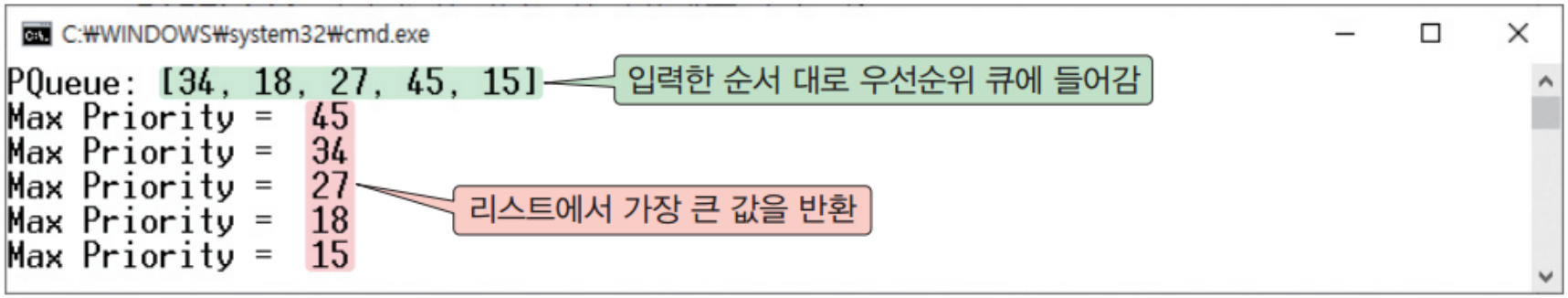
삭제 연산



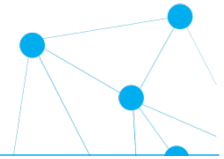
```
def findMaxIndex( self ):                    # 최대 우선순위 항목의 인덱스 반환
    if self.isEmpty(): return None
    else:
        highest = 0                          # 0번을 최대라고 하고
        for i in range(1, self.size()) :     # 모든 항목에 대해
            if self.items[i] > self.items[highest] :
                highest = i                  # 최고 우선순위 인덱스 갱신
        return highest                       # 최고 우선순위 인덱스 반환
def dequeue( self ):                         # 삭제 연산
    highest = self.findMaxIndex()            # 우선순위가 가장 높은 항목
    if highest is not None :
        return self.items.pop(highest)       # 리스트에서 꺼내서 반환

def peek( self ):                            # peek 연산
    highest = findMaxIndex()                 # 우선순위가 가장 높은 항목
    if highest is not None :
        return self.items[highest]           # 꺼내지 않고 반환
```

테스트 프로그램

```
q = PriorityQueue()
q.enqueue( 34 )
q.enqueue( 18 )
q.enqueue( 27 )
q.enqueue( 45 )
q.enqueue( 15 )

print("PQueue:", q.items)
while not q.isEmpty() :
    print("Max Priority = ", q.dequeue() )
```



시간 복잡도



- 정렬되지 않은 리스트 사용
  - enqueue(): 대부분의 경우
  - findMaxIndex():
  - dequeue(), peek() :
- 정렬된 리스트 사용
  - enqueue():
  - dequeue(), peek() :
- 힙 트리(8장)
  - enqueue(), dequeue():
  - peek() :