

Data Structure

자료구조



다양한 큐



한국기술교육대학교
온라인평생교육원

학습내용

- 덱의 개념과 동작 원리
- 상속을 이용한 원형 덱의 구현
- 우선 순위 큐

학습목표

- 덱의 개념과 동작 원리를 설명할 수 있다.
- 상속을 이용해 원형 덱을 구현할 수 있다.
- 우선 순위 큐의 개념과 동작을 설명할 수 있다.

덱의 개념과 동작 원리



다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

덱(Deque)

- Double-ended queue의 줄임말
- 전단(front)과 후단(rear)에서 모두 삽입과 삭제가 가능한 큐
- 스택이나 큐보다 입출력이 자유로운 자료구조
(덱을 스택이나 큐로 사용할 수 있음)
- 연산의 이름은 달라짐

다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

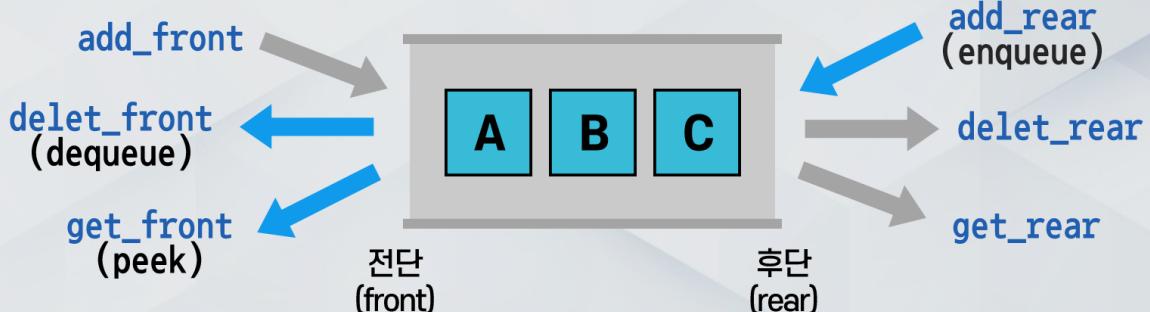
◉ 덱이란?

▶ 덱의 구조

덱에서는 삽입과 삭제를 양쪽에서 모두 할 수 있음

중간에 넣거나 빼는 것은 못 함

큐, 스택과 비슷한 연산이 많음





덱의 개념과 동작 원리

다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

◉ 달라지는 연산의 이름

스택의 push(), 큐의 enqueue()

→ add rear()

스택의 pop()

→ delete rear()

큐의 dequeue()

→ delete front()

스택의 peek()

→ get rear()

큐의 peek()

→ get front()

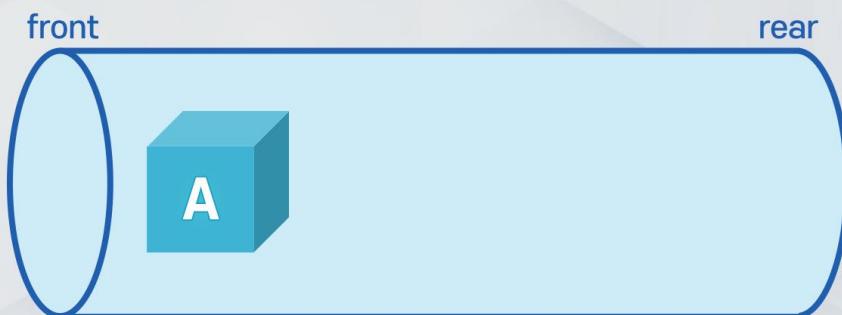
다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

◉ 덱의 연산

▶ 덱의 일련의 연산



add_front(A)



덱의 개념과 동작 원리

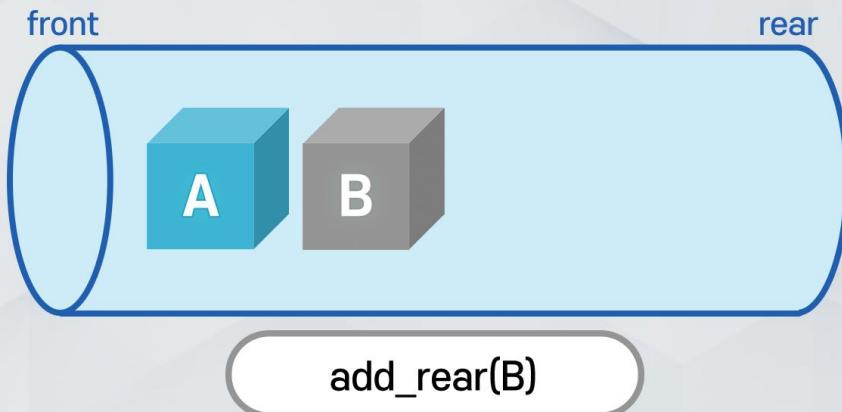
다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

◉ 덱의 연산

▶ 덱의 일련의 연산



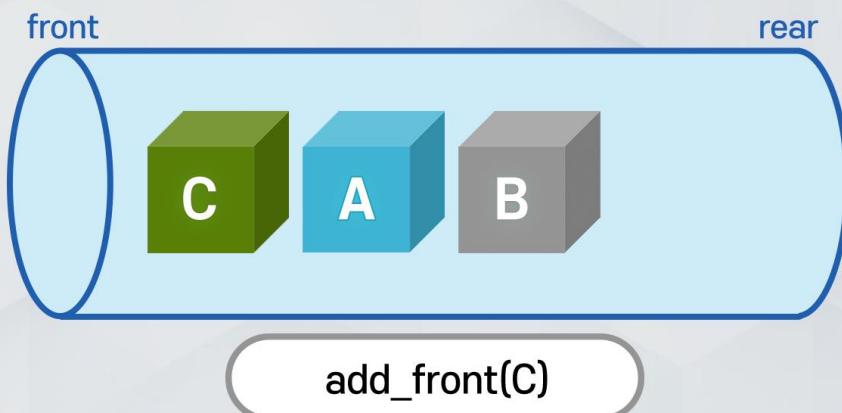
다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

◉ 덱의 연산

▶ 덱의 일련의 연산





덱의 개념과 동작 원리

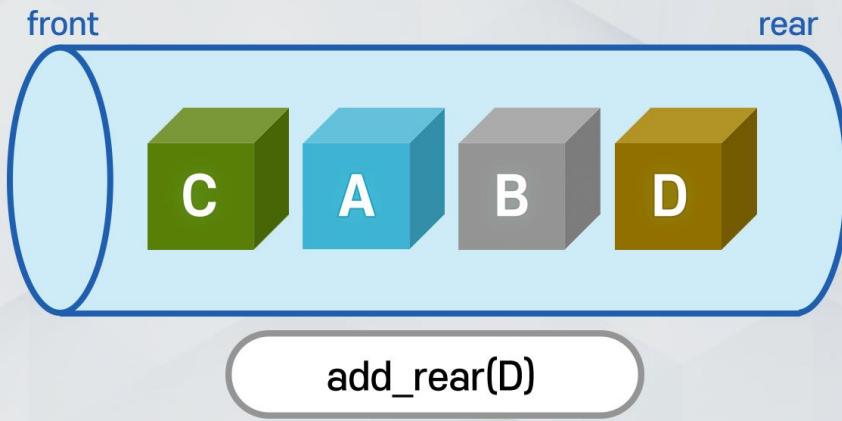
다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

◉ 덱의 연산

▶ 덱의 일련의 연산



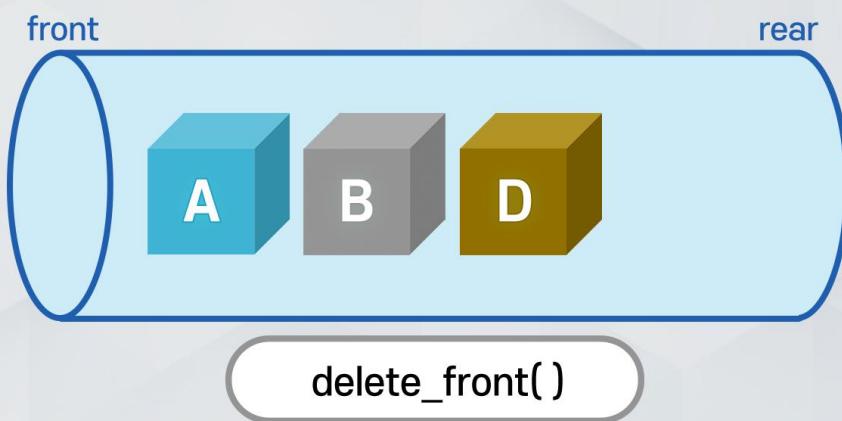
다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

◉ 덱의 연산

▶ 덱의 일련의 연산





덱의 개념과 동작 원리

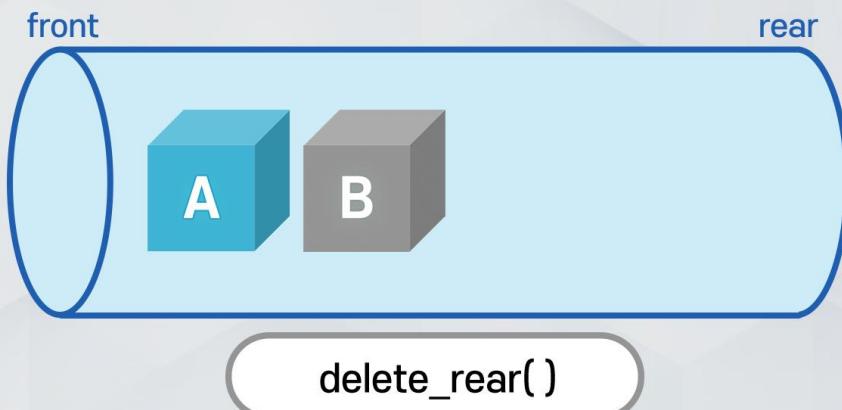
다양한 큐

덱의 개념과 동작 원리

1 덱의 개념과 구조

● 덱의 연산

▶ 덱의 일련의 연산



다양한 큐

덱의 개념과 동작 원리

2 덱의 추상 자료형

데이터

전단과 후단의 접근을 허용하는 항목들의 모음

▶ 연산

Deque()	비어 있는 새로운 덱을 만듦
isEmpty()	덱이 비어있으면 True를 아니면 False를 반환
isFull()	덱이 포화상태이면 True를 아니면 False를 반환
add_front(x)	항목 X를 덱의 맨 앞에 추가
delete_front()	맨 앞에 있는 항목을 꺼내 반환



덱의 개념과 동작 원리

다양한 큐

덱의 개념과 동작 원리

2 덱의 추상 자료형

▶ 연산

`add_rear(x)`

항목 X를 덱의 맨 뒤에 추가

`delete_rear()`

맨 뒤의 항목을 꺼내서 반환

다양한 큐

덱의 개념과 동작 원리

2 덱의 추상 자료형

▶ 연산

`get_front()`

전단 항목을 꺼내지 않고 반환

`get_rear()`

후단 항목을 꺼내지 않고 반환

`size()`

모든 항목들의 개수를 반환

`clear()`

덱을 공백상태로 만듦

`display()`

덱을 화면에 보기 좋게 출력

덱의 개념과 동작 원리



다양한 큐

덱의 개념과 동작 원리

2 덱의 추상 자료형

● 배열 구조의 덱

원형 덱

- 원형 큐와 마찬가지로 배열을 원형으로 사용
- 큐와 데이터는 동일
- 연산들도 기능은 유사, 이름은 다름

덱과 큐에서 동일한 연산

- 덱의 addRear(), deleteFront(), getFront()
- 큐의 enqueue, dequeue, peek 연산과 동일

다양한 큐

덱의 개념과 동작 원리

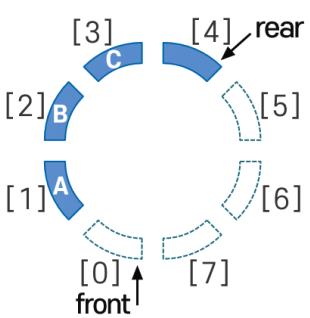
2 덱의 추상 자료형

● 원형 큐에서 추가되어야 할 연산

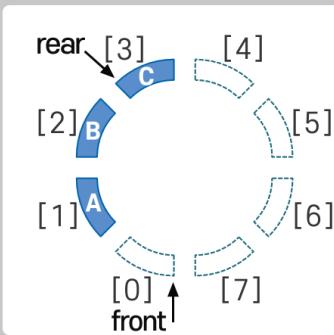
`delete_rear()`, `add_front()`, `get_rear()`

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

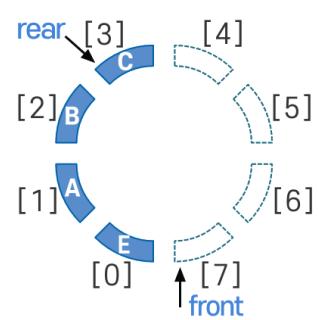
▶ 반 시계방향 회전 필요



`deleteRear()`



`addFront(E)`





상속을 이용한 원형 덱의 구현

다양한 큐

상속을 이용한 원형 덱의 구현

1 상속을 이용한 원형 덱

덱은 큐와 많은 부분이 유사

▶ 데이터

- 항목 저장을 위한 배열 items
- 전단, 후단: front, rear

▶ 원형 큐와 비슷한 방법으로 덱 클래스 구현 가능

더 좋은 방법: **클래스의 상속 기능** 이용

▶ 원형 큐 클래스 상속 → 원형 덱 클래스 구현

다양한 큐

상속을 이용한 원형 덱의 구현

1 상속을 이용한 원형 덱

데이터 멤버(front,rear,items)

- 추가 멤버가 필요 없음 (부모 클래스에 이미 정의)

멤버 함수

- 재사용 멤버 함수들
- 인터페이스 변경 함수들
 - `addRear(), deleteFront(), getFront()`
- 새로 구현할 함수
 - `addFront(), deleteRear(), getRear()`



상속을 이용한 원형 덱의 구현

다양한 큐

상속을 이용한 원형 덱의 구현

2 원형 덱의 구현

멤버 변수

정의하지 않음(상속)

▶ 부모 클래스의 멤버

Items[]

큐 항목들을 저장하는 배열

capacity

큐의 최대 용량

front

전단, 가장 최근에 삭제된 위치

rear

후단, 가장 최근에 삽입된 위치

다양한 큐

상속을 이용한 원형 덱의 구현

2 원형 덱의 구현

◎ 생성자

생성자

▪ 상속되지 않음

- 자식 클래스에서 다시 만들어야 함

▪ 부모 클래스의 생성자를 호출하여 데이터 멤버를 초기화

- 자식 클래스에서 부모를 호출하는 함수: super()



상속을 이용한 원형 덱의 구현

다양한 큐

상속을 이용한 원형 덱의 구현

2 원형 덱의 구현

● 생성자

▶ 클래스 정의와 생성자

```
class CircularDeque(CircularQueue) :           # CircularQueue를 상속함
    def __init__( self, capacity=10 ) :          # 생성자
        super().__init__(capacity)                # 부모 클래스의 생성자 호출
```

다양한 큐

상속을 이용한 원형 덱의 구현

2 원형 덱의 구현

● 재사용 연산들

▶ 추가할 필요 없는 클래스

공백 상태	isEmpty()	→ front == rear
포화 상태	isFull()	→ front == (rear+1) % MAX_QSIZE
항목의 수	size()	
초기화	clear()	
화면 출력	display()	



상속을 이용한 원형 덱의 구현

다양한 큐

상속을 이용한 원형 덱의 구현

2 원형 덱의 구현

- 인터페이스 변경 연산들 ▶ 후단 삽입: addRear() == enqueue()

후단 삽입

addRear() == enqueue()

```
def addRear( self, item ):
    self.enqueue( item )
```

전단 삭제

deleteFront() == dequeue()

```
def deleteFront( self ):
    return self.dequeue()
```

전단 peek()

getFront() == peek()

```
def getFront( self ):
    return self.peek()
```

다양한 큐

상속을 이용한 원형 덱의 구현

2 원형 덱의 구현

- 추가로 구현할 연산들

▶ 전단 삽입: 반 시계 방향의 회전 필요

```
def addFront( self, item ):
    if not self.isFull():
        self.items[self.front] = item
        self.front = (self.front - 1 + self.capacity) % self.capacity
```



상속을 이용한 원형 덱의 구현

다양한 큐

상속을 이용한 원형 덱의 구현

2 원형 덱의 구현

● 추가로 구현할 연산들

▶ 후단 삭제: 반 시계 방향의 회전 필요

```
def deleteRear( self ):
    if not self.isEmpty():
        item = self.items[self.rear];
        self.rear = (self.rear - 1 + self.capacity) % self.capacity
    return item
```

▶ 후단 peek()

```
def getRear( self ):
    return self.items[self.rear]
```



상속을 이용한 원형 덱의 구현

```

파일(F) 편집(V) 보기(V) 프로젝트(P) 디버그(D) 테스트(S) 도구(T) 확장(X) 찾(W) 도움말(H) 검색(Ctrl+Q) Python 3.6 (64-bit) 교재-파이썬
MazeQueue.py QueueCircular.py
DequeCircular.py MazeQueue.py QueueCircular.py
CircularDeque
4 ##### 덱 ADT의 구현: 원형 큐 클래스를 상속하여 구현 #####
5 # 덱 ADT의 구현: 원형 큐 클래스를 상속하여 구현
6 #####
7 from QueueCircular import CircularQueue
8
9 class CircularDeque(CircularQueue):
10     def __init__(self, capacity=10):          # 생성자
11         super().__init__(capacity)           # 부모 클래스의 생성자 호출
12
13     def addRear(self, item):                 # 삽입(원형 큐 뒤)
14         self.enqueue(item)
15
16     def deleteFront(self):                  # 삭제(원형 큐 앞)
17         return self.dequeue()
18
19     def getFront(self):                    # 첫 원소 확인
20         return self.peek()
21
22     def addFront(self, item):              # 삽입(원형 큐 앞)
23         if not self.isEmpty():
24             self.items[self.front] = item
25             self.front = (self.front - 1 + self.capacity) % self.capacity

```

실습 단계
자식 클래스: <u>CircularDeque</u> , 부모 클래스: <u>CircularQueue</u>
<u>CircularQueue</u> 를 사용하기 위해 <u>QueueCircular.py</u> 를 import
부모 클래스(<u>CircularQueue</u>)에 구현되어 있는 연산은 자식 클래스(<u>CircularDeque</u>)에서는 추가하지 않음(size(), clear() 등)
인터페이스 변경 연산
새롭게 추가한 연산
실행 결과 확인
파이썬의 <u>collections</u> 모듈에서 <u>deque</u> 클래스를 제공(코드에 import collections 추가)
후단 삽입/삭제, 전단 삽입/삭제 연산의 이름이 다름
peek 연산 제공하지 않음
실행 결과 확인

우선 순위 큐



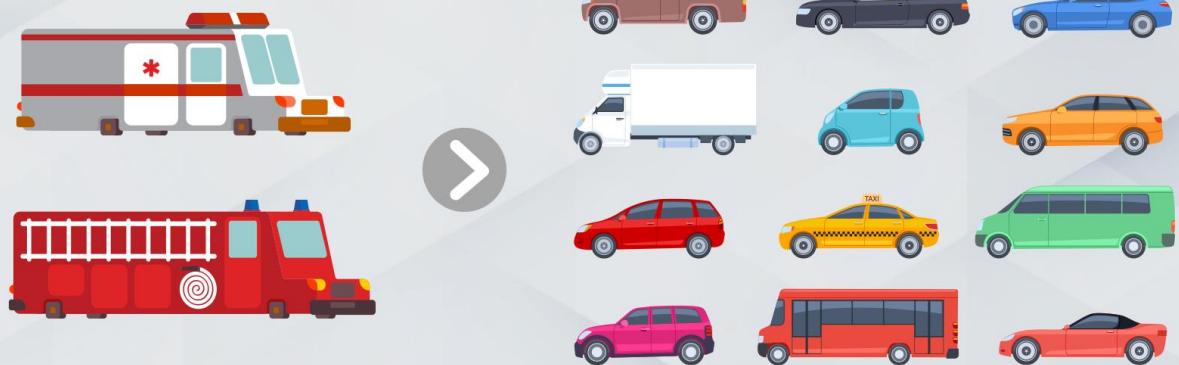
다양한 큐

우선 순위 큐

1 우선 순위 큐의 개념

● 실생활에서의 우선 순위

예 도로에서의 자동차 우선 순위



다양한 큐

우선 순위 큐

1 우선 순위 큐의 개념

우선 순위 큐 (Priority queue)

- 우선 순위의 개념을 큐에 도입한 자료구조
- 모든 데이터가 우선 순위를 가짐
- 입력 순서와 상관없이 우선 순위가 높은 데이터가 먼저 출력
- 가장 일반적인 큐로 볼 수 있음
 - 우선 순위를 어떻게 정하느냐에 따라 스택, 큐 또는 다른 형태의 큐를 만들 수 있기 때문

우선 순위 큐



다양한 큐

우선 순위 큐

1 우선 순위 큐의 개념

우선 순위 큐 (Priority queue)

- 선형 자료형으로 볼 수 없음

- 큐에 들어가 있는 항목들 중 우선 순위가 높은 항목을 뽑기만 하면 됨
- 다른 항목들 사이에서 순서가 지켜질 필요 없음

응용분야

- 시뮬레이션
- 네트워크 트래픽 제어
- OS 작업 스케줄링 등

다양한 큐

우선 순위 큐

1 우선 순위 큐의 개념

● 우선 순위 큐의 추상 자료형

데이터

우선 순위를 가진 요소들의 모음

▶ 연산

PriorityQueue()

비어 있는 우선 순위 큐를 만듦

isEmpty()

우선 순위 큐가 공백 상태인지를 검사

isFull()

우선 순위 큐가 포화 상태인지를 검사

enqueue(e)

우선 순위를 가진 항목 e를 추가

dequeue()

가장 우선 순위가 높은 항목을 꺼내서 반환



우선 순위 큐

다양한 큐 우선 순위 큐

1 우선 순위 큐의 개념

● 우선 순위 큐의 추상 자료형

▶ 연산

peek()	가장 우선 순위가 높은 요소를 삭제하지 않고 반환
size()	우선순위 큐의 모든 항목들의 개수를 반환
clear()	우선 순위 큐를 공백 상태로 만듦
display()	우선 순위 큐를 화면에 보기 좋게 출력

다양한 큐 우선 순위 큐

1 우선 순위 큐의 개념

● 우선 순위 큐의 일련의 연산



```

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

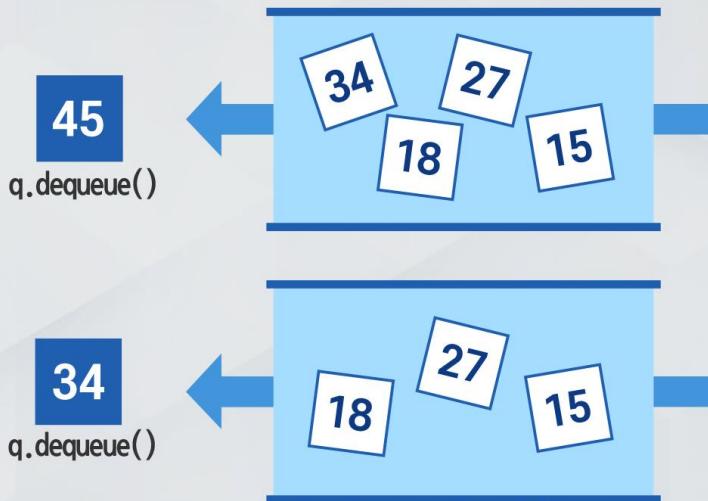


우선 순위 큐

다양한 큐 우선 순위 큐

1 우선 순위 큐의 개념

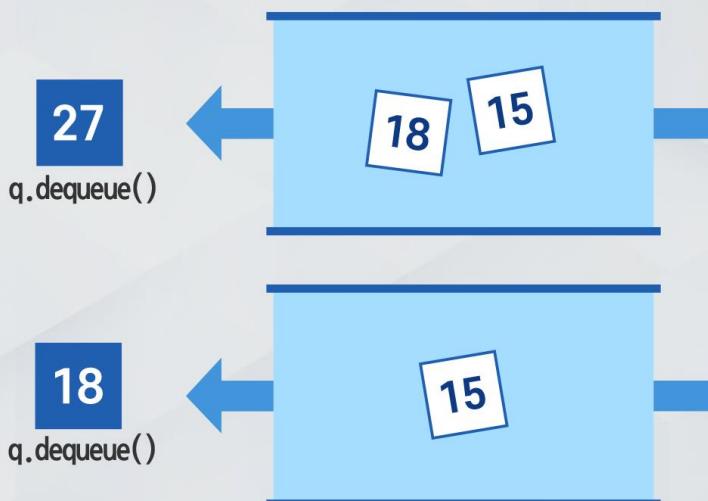
- 우선 순위 큐의 일련의 연산



다양한 큐 우선 순위 큐

1 우선 순위 큐의 개념

- 우선 순위 큐의 일련의 연산





우선 순위 큐

다양한 큐

우선 순위 큐

1 우선 순위 큐의 개념

● 우선 순위 큐의 구현 방법

선형으로 구현하는 방법

- 배열 구조 / 연결된 구조
- 항목들을 정렬하여 저장 / 정렬하지 않고 저장하는 방법

비선형으로 구현하는 방법

▪ 힙 트리

- 가장 효율적인 방법
- 삽입과 삭제 연산의 복잡도 동일: $O(\log n)$

다양한 큐

우선 순위 큐

1 우선 순위 큐의 개념

● 우선 순위 큐의 응용 분야

주요 응용 분야

▪ 허프만 코딩 트리

- 빈도가 작은 두 노드 선택할 때

▪ Kruskal의 MST 알고리즘

- MST에 포함되지 않은 간선 중, 최소 가중치 간선 선택할 때



우선 순위 큐

다양한 큐

우선 순위 큐

1 우선 순위 큐의 개념

◉ 우선 순위 큐의 응용 분야

주요 응용 분야

- Dijkstra의 최단 거리 알고리즘
 - 최단 거리가 찾아지지 않은 정점들 중, 가장 거리가 가까운 정점 선택
- 인공지능의 A* 알고리즘
 - 공간 트리(state space tree)에서 가장 가능성 높은(promising) 경로 먼저 선택하여 시도

우선 순위 큐의 가장 효율적인 구현 방법

힙 트리

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

가장 간단한 방법으로 우선순위 큐 구현

- ▶ 정렬되지 않은 배열을 이용한 구현
- ▶ “배열” ↗ 파이썬의 리스트 이용
- ▶ 용량(capacity)을 사용하지 않고, 파이썬 리스트의 스마트한
동적 배열 기능을 그대로 활용해서 구현
 - ➡ 파이썬 리스트의 `append()`, `pop()` 등의 연산 이용



우선 순위 큐

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

● 클래스 정의와 멤버 변수

```
class PriorityQueue :
    def __init__( self ):
        self.items = []
```

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

● 기본적인 연산들

▶ 항목의 수: 파이썬의 len() 함수 이용

```
def size( self ):
    return len(self.items)
```

▶ 초기화

```
def clear( self ):
    self.items = []
```

▶ 화면 출력

```
def display(self, msg='PQueue:'):
    print(msg, self.items)
```



우선 순위 큐

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

● 공백 상태, 포화 상태

▶▶ 공백 상태 검사

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

▶▶ 포화 상태 검사: 의미 없음 ➡ 항상 False

```
def isFull( self ):
    return False
```

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

● 삽입 연산

삽입 연산

- 항목을 정렬하지 않으므로 단순히 리스트에 삽입
- 삽입 위치: **리스트의 후단**이 가장 효율적

```
def enqueue( self, item ):
    self.items.append( item )
```



우선 순위 큐

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

◎ 삭제 연산

삭제 연산

- 우선 순위가 가장 높은 항목을 먼저 찾아야 함

```
def findMaxIndex( self ):
    if self.isEmpty(): return None
    else:
        highest = 0
        for i in range(1, self.size()):
            if self.items[i] > self.items[highest]:
                highest = i
    return highest
```

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

◎ 삭제 연산

삭제 연산

- 찾은 위치의 항목을 리스트에서 꺼내 반환

```
def dequeue( self ):
    highest = self.findMaxIndex()
    if highest is not None:
        return self.items.pop(highest)
```

우선 순위 큐



다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

◎ peek 연산

peek 연산

- 삭제 연산과 유사
- 우선 순위가 가장 높은 항목을 배열에서 꺼내지 않고 반환만 함

```
def peek( self ):
    highest = findMaxIndex()
    if highest is not None :
        return self.items[highest]
```

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

◎ 테스트 프로그램

```
if __name__ == "__main__":
    print('〈우선 순위 큐(정렬되지 않은 리스트로 구현) 테스트〉')
    pq = PriorityQueue()
    pq.enqueue( 34 )
    pq.enqueue( 18 )
    pq.enqueue( 27 )
    pq.enqueue( 45 )
    pq.enqueue( 15 )

    pq.display()
    while not pq.isEmpty() :
        print("Max Priority = ", pq.dequeue() )
```

C:\WINDOWS\system32\cmd.exe

〈우선순위 큐(정렬되지 않은 리스트로 구현) 테스트〉

PQueue: [34, 18, 27, 45, 15]

Max Priority = 45

Max Priority = 34

Max Priority = 27

Max Priority = 18

Max Priority = 15



우선 순위 큐

다양한 큐

우선 순위 큐

2 정렬되지 않은 배열을 이용한 구현

● 시간 복잡도

정렬되지 않은 리스트 사용	정렬된 리스트 사용	힙 트리
<ul style="list-style-type: none">enqueue(): 대부분의 경우 $O(1)$findMaxIndex: $O(n)$dequeue(), peek(): $O(n)$	<ul style="list-style-type: none">enqueue(): $O(n)$dequeue(), peek(): $O(1)$	<ul style="list-style-type: none">dequeue(), peek(): $O(\log n)$peek(): $O(1)$