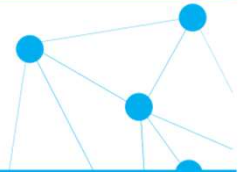
7주차 큐(QUEUE)

큐(QUEUE)란?

- 큐는 선입선출(First-In First Out: FIFO)의 자료구조
- 일상생활의 관공서, 은행, 우체국, 병원 등에서 번호표를 이용한 줄서기

- 큐의 구조

큐 ADT



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

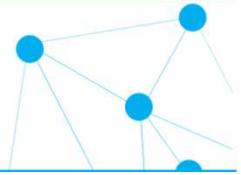
정의 5.1 Queue ADT

데이터: 선입선출(FIFO)의 접근 방법을 유지하는 항목들의 모음

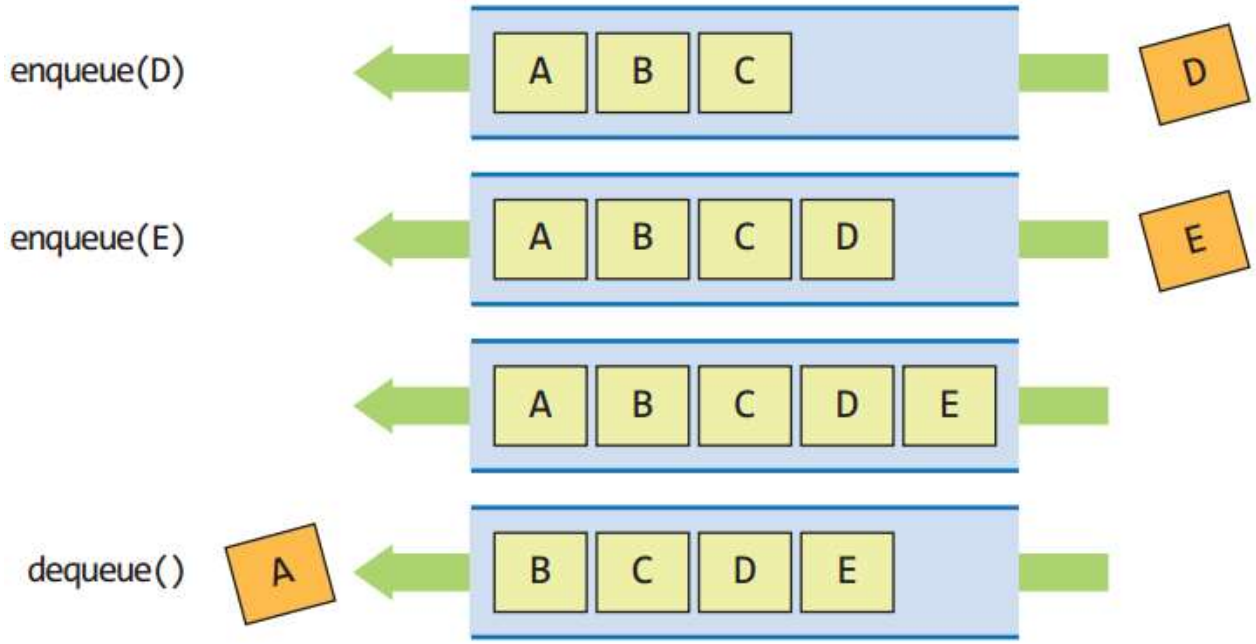
연산

- 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.

큐의 연산

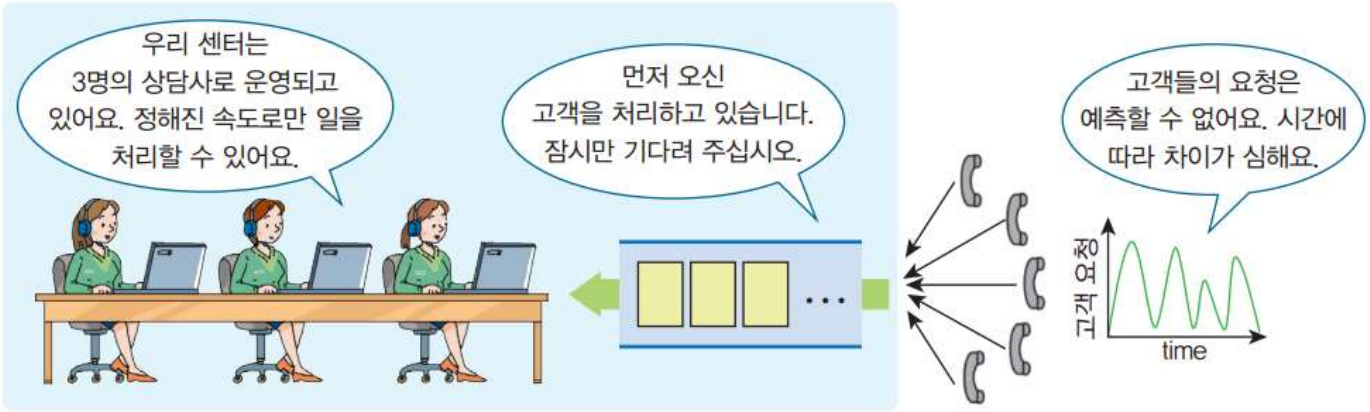


- 삽입: enqueue()
- 삭제: dequeue()



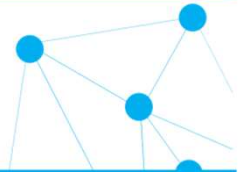
큐의 응용

- 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

큐의 구현

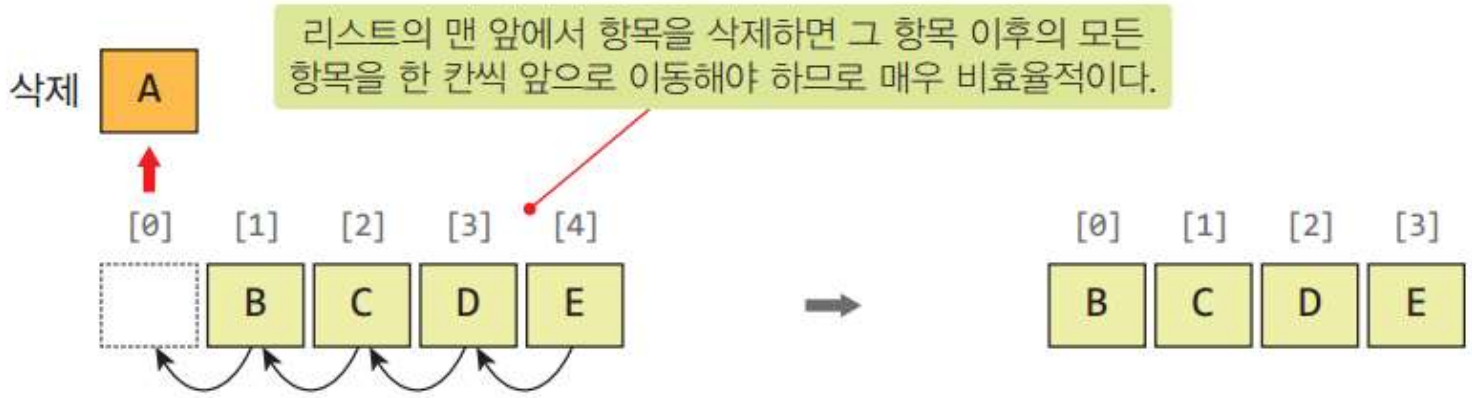


- 선형큐는 비효율적이다.
  - enqueue(item): 삽입 연산 → $O(1)$

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

  - dequeue(): 삭제 연산 → $O(n)$ *why?*

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



파이썬 리스트로 구현한 큐

```
01 def add(item): # 삽입 연산
02     q.append(item)
03
04 def remove(): # 삭제 연산
05     if len(q) != 0:
06         item = q.pop(0)
07         return item
08
09 def print_q(): # 큐 출력
10     print('front ->  ', end='')
11     for i in range(len(q)):
12         print('{!s:<8}'.format(q[i]), end='')
13     print('  <- rear')
```

맨 뒤에 새 항목 삽입

맨 앞의 항목 삭제

맨 앞부터 항목들을 차례로 출력

```
14 q = []
15 add('apple')
16 add('orange')
17 add('cherry')
18 add('pear')
19 print('사과, 오렌지, 체리, 배 삽입 후: \t', end='')
20 print_q()
21 remove()
22 print('remove한 후:\t\t', end='')
23 print_q()
24 remove()
25 print('remove한 후:\t\t', end='')
26 print_q()
27 add('grape')
28 print('포도 삽입 후:\t\t', end='')
29 print_q()
```

리스트 선언

일련의 큐 연산과 출력

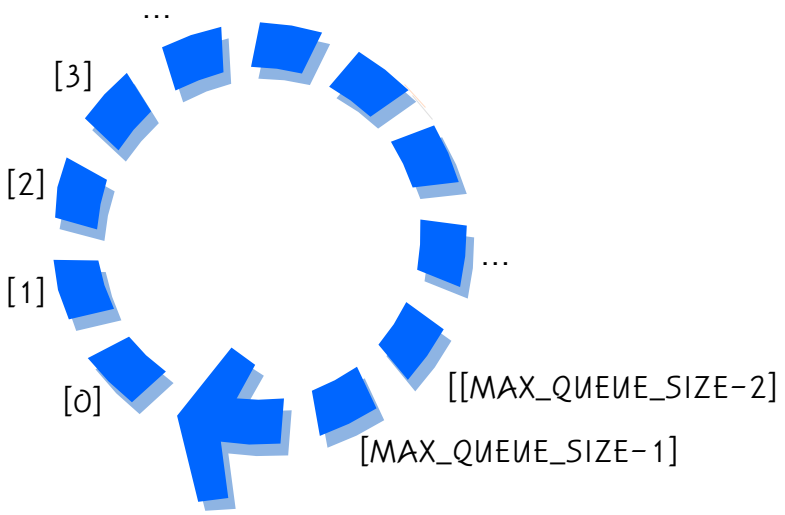
[프로그램 3-3]

Console PyUnit

```
<terminated> listqueue.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32₩pyt
사과, 오렌지, 체리, 배 삽입 후:  front -> apple  orange  cherry  pear      <- rear
remove한 후:                     front -> orange  cherry  pear      <- rear
remove한 후:                     front -> cherry  pear      <- rear
포도 삽입 후:                    front -> cherry  pear    grape     <- rear
```

원형 큐가 훨씬 효율적이다.

- 원형큐
  - 배열을 원형으로 사용

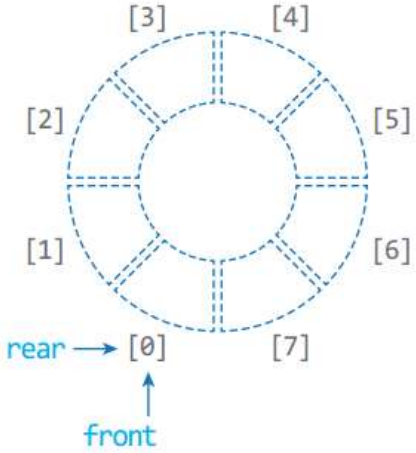


- 전단과 후단을 위한 2개의 변수
  - front: 첫번째 요소 하나 앞의 인덱스
  - rear: 마지막 요소의 인덱스

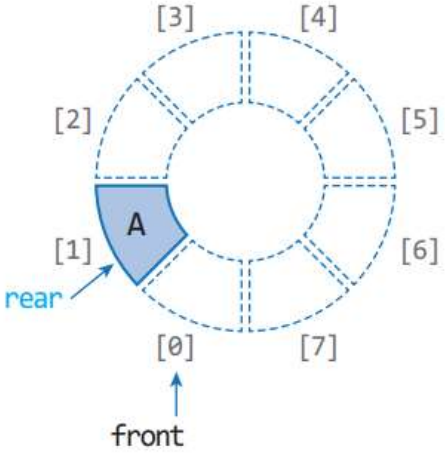
- 회전(시계방향) 방법

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

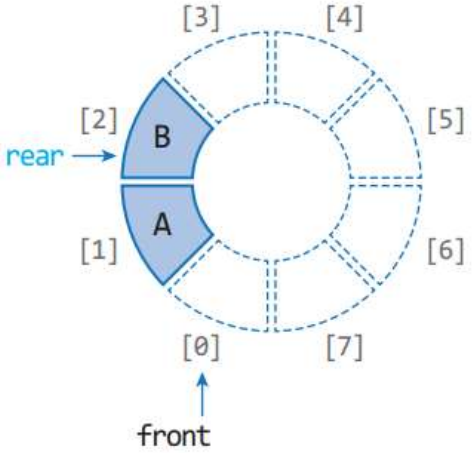
원형 큐의 삽입과 삭제 과정



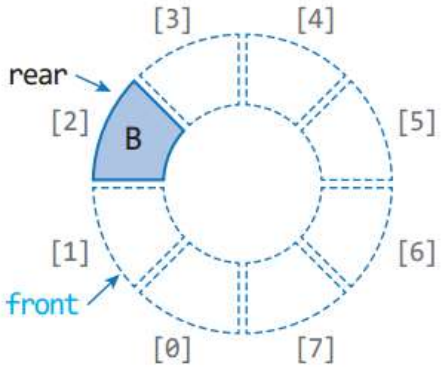
초기상태



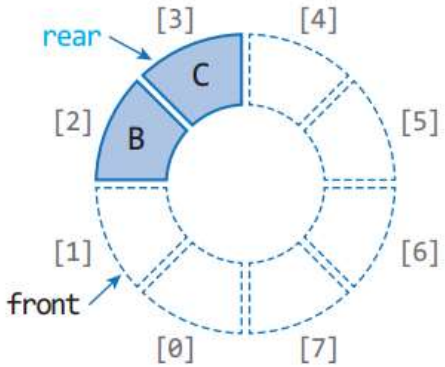
enqueue(A)



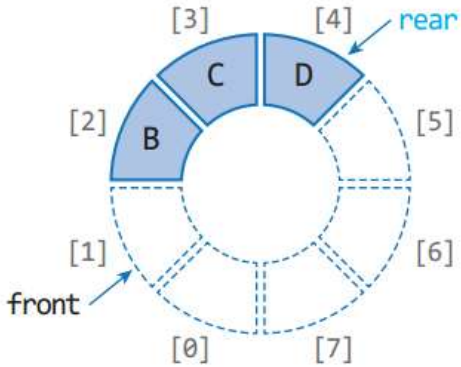
enqueue(B)



dequeue()



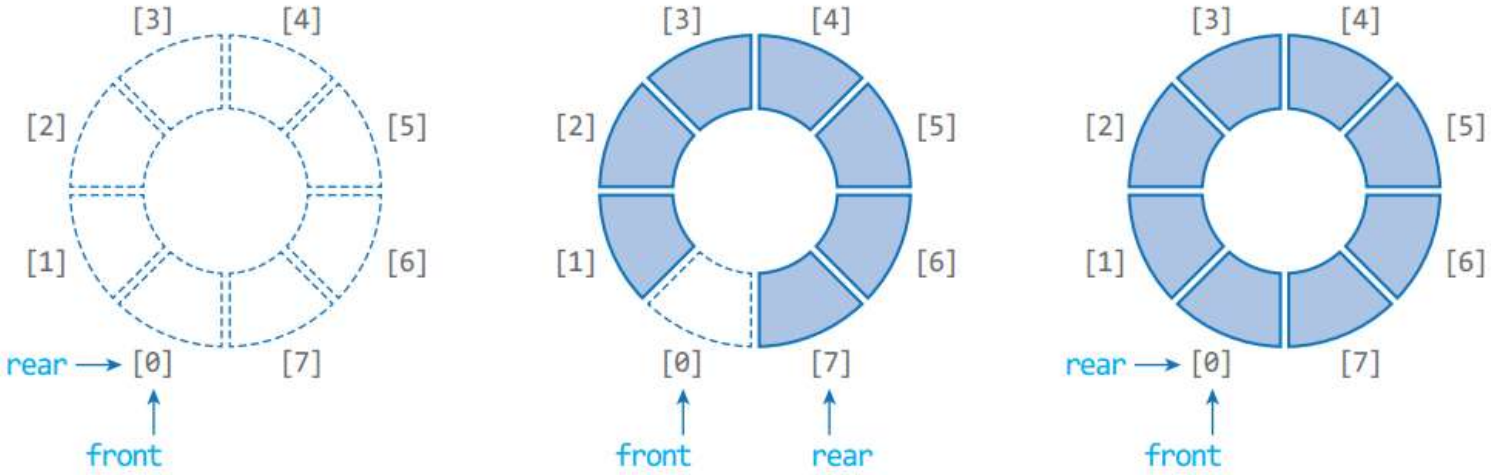
enqueue(C)



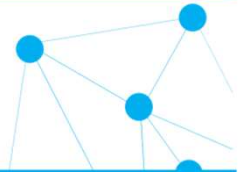
enqueue(D)

공백상태와 포화상태

- 공백상태: `front == rear`
- 포화상태: `front ==(rear+1) % MAX_QSIZE`
- 공백상태와 포화상태를 구별 방법은?
  - 하나의 공간은 항상 비워둠



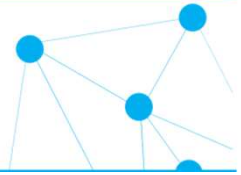
원형 큐의 구현



- 파이썬 리스트 사용
- 리스트의 크기가 미리 결정되어야 함.➔ 포화상태 있음
- 원형 큐 클래스

```
MAX_QSIZE = 10                          # 원형 큐의 크기
class CircularQueue :
    def __init__( self ) :              # CircularQueue 생성자
        self.front = 0                  # 큐의 전단 위치
        self.rear = 0                   # 큐의 후단 위치
        self.items = [None] * MAX_QSIZE # 항목 저장용 리스트 [None,None,...]
    def isEmpty( self ) : return self.front == self.rear
    def isFull( self ) : return self.front == (self.rear+1)%MAX_QSIZE
```

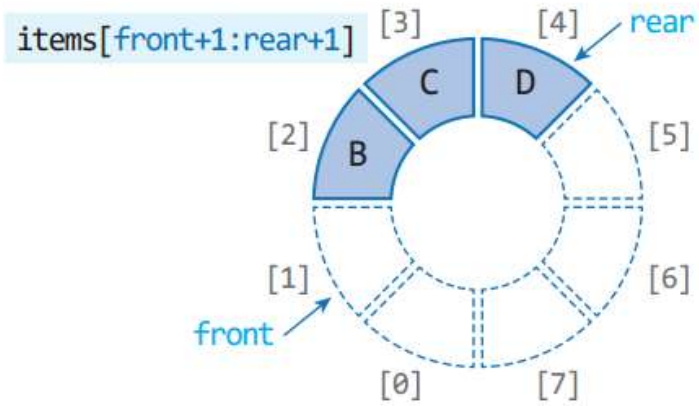
원형 큐의 연산들(삽입/삭제)

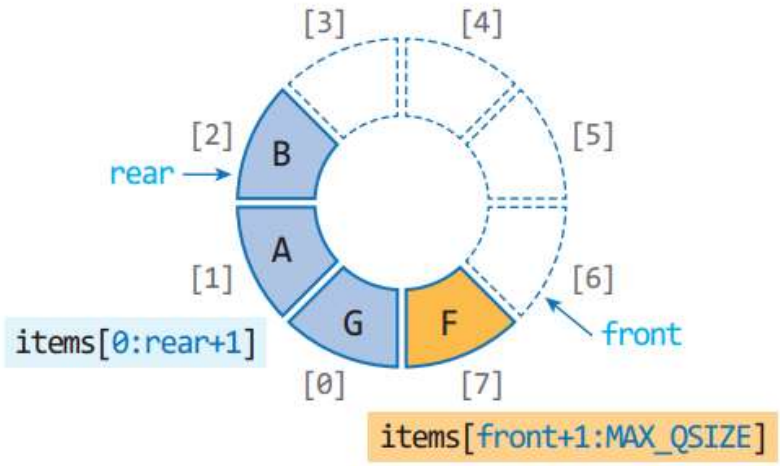


```
def enqueue( self, item ):
    if not self.isFull():                    # 포화상태가 아니면
        self.rear = (self.rear+1)% MAX_QSIZE # rear 회전
        self.items[self.rear] = item         # rear 위치에 삽입

def dequeue( self ):
    if not self.isEmpty():                   # 공백상태가 아니면
        self.front = (self.front+1)% MAX_QSIZE # front 회전
        return self.items[self.front]        # front위치의 항목 반환

def peek( self ):
    if not self.isEmpty():
        return self.items[(self.front + 1) % MAX_QSIZE]
def size( self ) :
    return (self.rear - self.front + MAX_QSIZE) % MAX_QSIZE
```

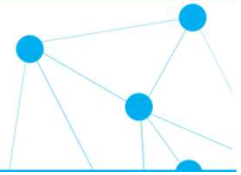
원형 큐의 연산들(출력)

```
def display( self ):
    out = []
    if self.front < self.rear :
        out = self.items[self.front+1:self.rear+1]          # 슬라이싱
    else:
        out = self.items[self.front+1:MAX_QSIZE]            # 다음 줄에 계속...
            + self.items[0:self.rear+1]                     # 슬라이싱
    print( "[f=%d,r=%d] ==> "%(self.front, self.rear), out)
```





테스트 프로그램

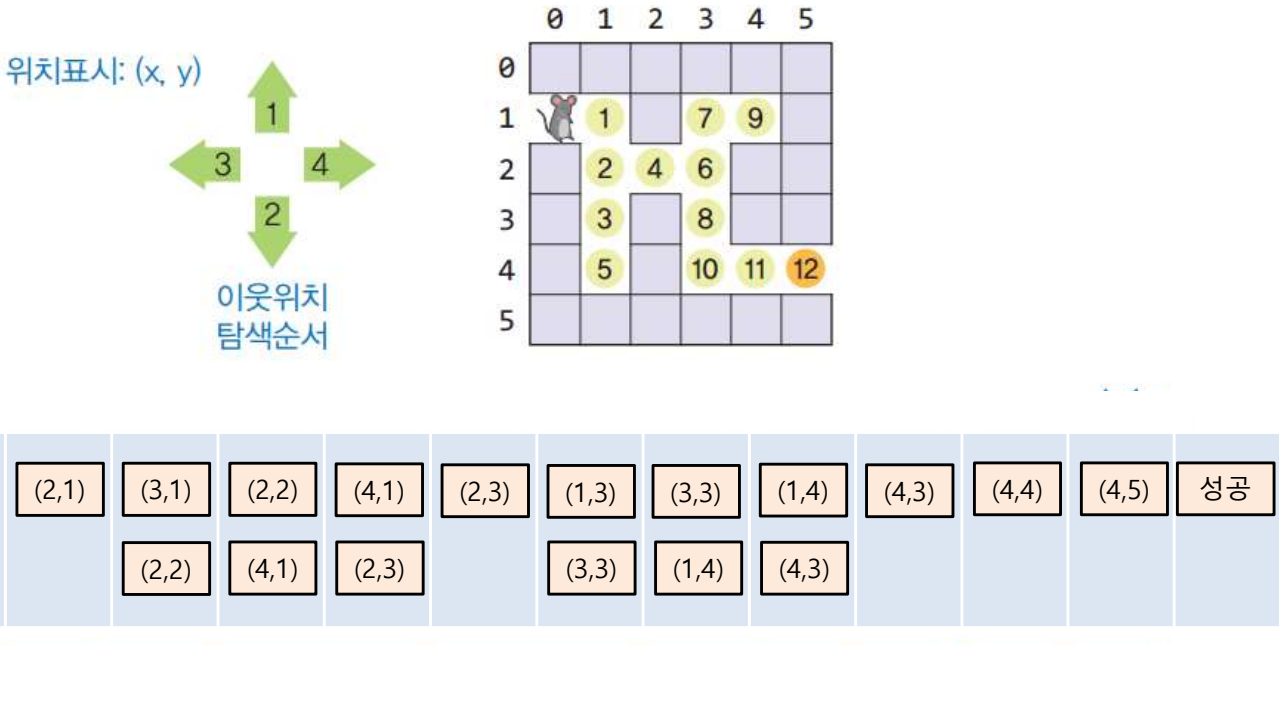


```
q = CircularQueue()                    # 원형큐 만들기 (MAX_QSIZE=10)
for i in range(8): q.enqueue(i)        # 0, 1, ... 7 삽입(f=0, r=8)
q.display()                            # 원형 큐에서 구현한 print()호출
for i in range(5): q.dequeue();        # 5번 삭제 (f=5, r=8)
q.display()
for i in range(8,14): q.enqueue(i)     # 8, 9, ... 13 삽입 (f=5, r=4)
q.display()
```

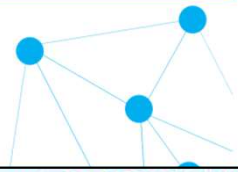
```
C:₩WINDOWS₩system32₩cmd.exe
[f=0,r=8] ==>  [0, 1, 2, 3, 4, 5, 6, 7]
[f=5,r=8] ==>  [5, 6, 7]
[f=5,r=4] ==>  [5, 6, 7, 8, 9, 10, 11, 12, 13]
```

큐의 응용: 너비우선탐색

- 수업에서 큐 응용
  - 이진트리의 레벨 순회
  - 기수정렬에서 레코드의 정렬을 위해 사용
  - 그래프의 탐색에서 너비우선탐색
  - 미로 탐색: 너비우선탐색



너비우선탐색 알고리즘

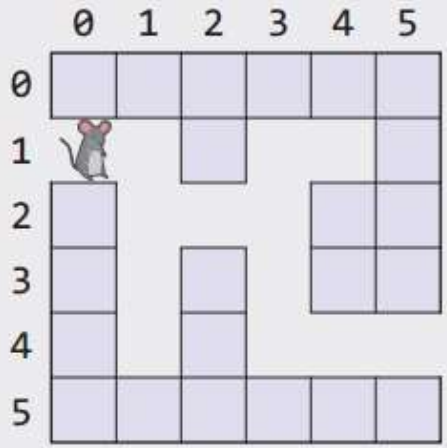


```
def BFS() :                          # 너비우선탐색 함수
    que = CircularQueue()
    que.enqueue((0,1))
    print('BFS: ')                   # 출력을 'BFS'로 변경

    while not que.isEmpty():
        here = que.dequeue()
        print(here, end='->')
        x,y = here
        if (map[y][x] == 'x') : return True
        else :
            map[y][x] = '.'
            if isValidPos(x, y - 1) : que.enqueue((x, y - 1))
            if isValidPos(x, y + 1) : que.enqueue((x, y + 1))
            if isValidPos(x - 1, y) : que.enqueue((x - 1, y))
            if isValidPos(x + 1, y) : que.enqueue((x + 1, y))
    return False
```

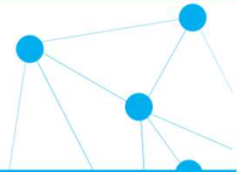
```
def DFS() :                          # 깊이우선탐색 함수
    stack = Stack()                  # 사용할 스택 객체를
    stack.push( (0,1) )              # 시작위치 삽입. (0,
    print('DFS: ')

    while not stack.isEmpty():       # 공백이 아닐 동안
        here = stack.pop()           # 항목을 꺼냄(pop)
        print(here, end='->')
        (x, y) = here                # 스택에 저장된 튜플
        if (map[y][x] == 'x') :      # 출구이면 탐색 성공
            return True
        else :
            map[y][x] = '.'          # 현재위치를 지나왔
            # 4방향의 이웃을 검사해 갈 수 있으면 스
            if isValidPos(x, y - 1): stack.push((x,
            if isValidPos(x, y + 1): stack.push((x,
            if isValidPos(x - 1, y): stack.push((x -
            if isValidPos(x + 1, y): stack.push((x +
        print(' 현재 스택: ', stack)  # 현재 스택 내
    return False                     # 탐색 실패. False
```

테스트 프로그램

```
map = [  [ '1', '1', '1', '1', '1', '1' ],
         [ 'e', '0', '1', '0', '0', '1' ],
         [ '1', '0', '0', '0', '1', '1' ],
         [ '1', '0', '1', '0', '1', '1' ],
         [ '1', '0', '1', '0', '0', 'x' ],
         [ '1', '1', '1', '1', '1', '1' ]]
MAZE_SIZE = 6
result = BFS()
if result : print(' --> 미로탐색 성공')
else : print(' --> 미로탐색 실패')
```



파이썬의 queue모듈



- 큐(Queue)와 스택(LifoQueue) 클래스를 제공
- 사용하기 위해서는 먼저 queue 모듈을 import해야 함

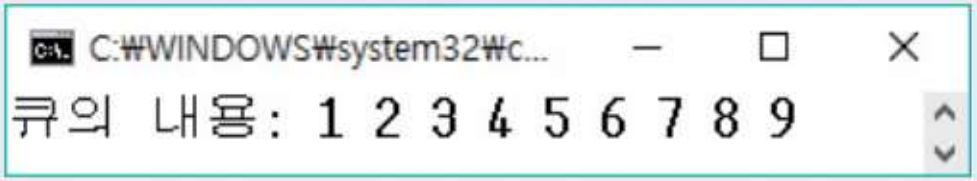
```
import queue                # 파이썬의 큐 모듈 포함
```

- 큐 객체 생성

```
Q = queue.Queue(maxsize=20)     # 큐 객체 생성(최대크기 20)
```

- 함수 이름 변경: 삽입은 put(), 삭제는 get()

```
for v in range(1, 10) :
    Q.put(v)
print("큐의 내용: ", end='')
for _ in range(1, 10) :
    print(Q.get(), end=' ')
print()
```



단순연결리스트로 구현한 큐

```
01 class Node:
02     def __init__(self, item, n):
03         self.item = item
04         self.next = n
05 def add(item): # 삽입 연산
06     global size
07     global front
08     global rear
09     new_node = Node(item, None)
10     if size == 0:
11         front = new_node
12     else:
13         rear.next = new_node
14     rear = new_node
15     size += 1
```

노드 생성자
항목과 다음 노드 레퍼런스

전역 변수

새 노드 객체를 생성

연결리스트의 맨 뒤에 삽입

```
16 def remove(): # 삭제 연산
17     global size
18     global front
19     global rear
20     if size != 0:
21         fitem = front.item
22         front = front.next
23         size -= 1
24         if size == 0:
25             rear = None
26         return fitem
```

전역 변수

연결리스트에서 front가 참조하던 노드 분리시킴

제거된 맨 앞의 항목 리턴

```
27 def print_q(): # 큐 출력
28     p = front
29     print('front: ', end='')
30     while p:
31         if p.next != None:
32             print(p.item, '->   ', end='')
33         else:
34             print(p.item, end = '')
35         p = p.next
36     print('  : rear')
37 front = None
38 rear = None
39 size = 0
40
:
54
```

단순연결리스트(스택)의 항목을 차례로 출력

초기화

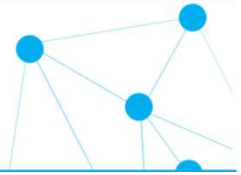
[프로그램 3-3]의 line 15~29와 동일

[프로그램 3-4]

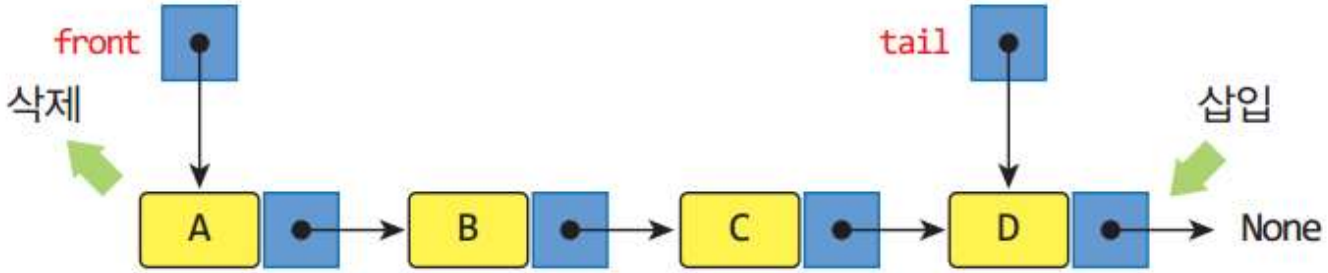
Console PyUnit

```
<terminated> linkedqueue.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32₩pyth
사과, 오렌지, 체리, 배 삽입 후: front: apple ->   orange ->   cherry ->   pear   : rear
remove한 후:                  front: orange ->   cherry ->   pear   : rear
remove한 후:                  front: cherry ->   pear   : rear
포도 삽입 후:                 front: cherry ->   pear ->   grape   : rear
```

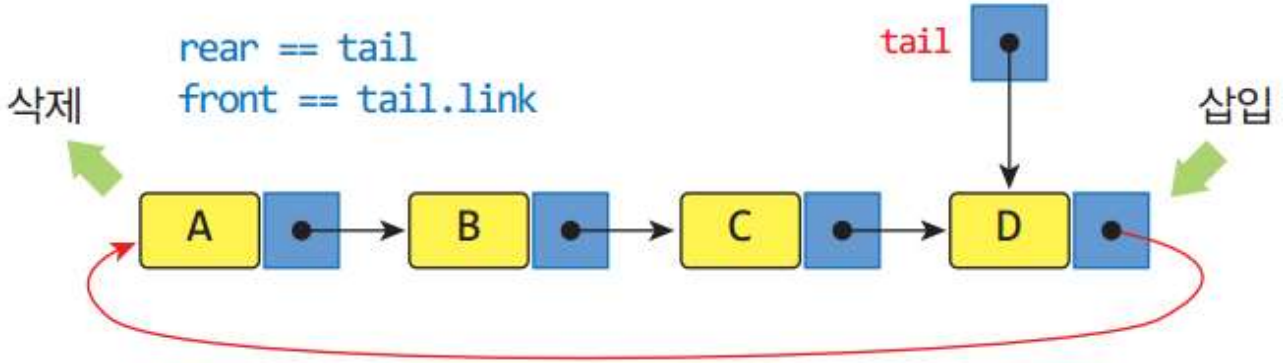
원형연결리스트의 응용: 연결된 큐



- 단순연결리스트로 구현한 큐

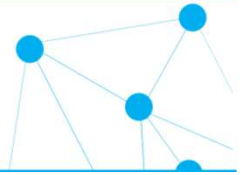


- 원형연결리스트로 구현한 큐



  - tail을 사용하는 것이 rear 와 front에 바로 접근할 수 있다는 점에서 훨씬 효율적

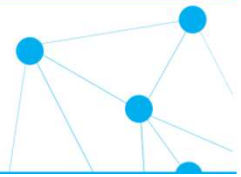
연결된 큐 클래스



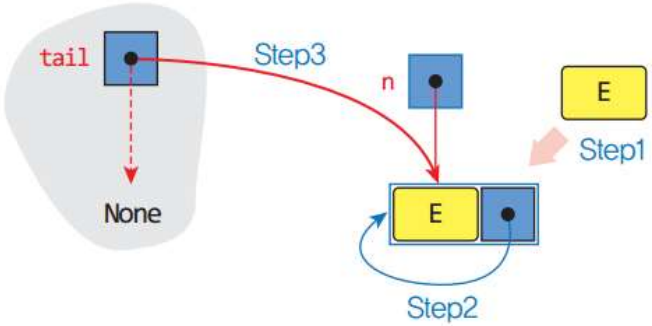
```
class CircularLinkedQueue:
    def __init__( self ):                          # 생성자 함수
        self.tail = None                           # tail: 유일한 데이터

    def isEmpty( self ): return self.tail == None  # 공백상태 검사
    def clear( self ): self.tail = None            # 큐 초기화
    def peek( self ):                              # peek 연산
        if not self.isEmpty():                     # 공백이 아니면
            return self.tail.link.data             # front의 data를 반환
```

```
class Node:                          #노드 생성
    def __init__ (self, elem, link=None):
        self.data = elem
        self.link = link
```

삽입 연산: enqueue()

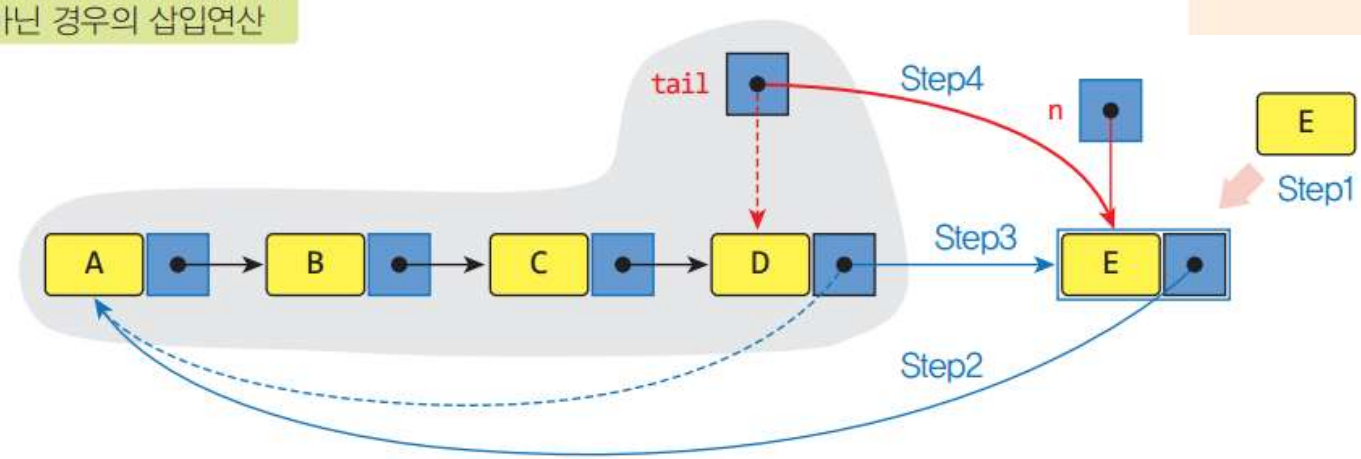


Case1: 큐가 공백상태인 경우의 삽입연산

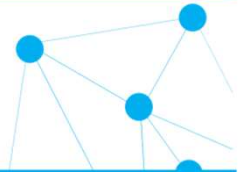



```
def enqueue( self, item ):
    node = Node(item, None)
    if self.isEmpty() :
        node.link = node
        self.tail = node
    else :
        node.link = self.tail.link
        self.tail.link = node
        self.tail = node
```

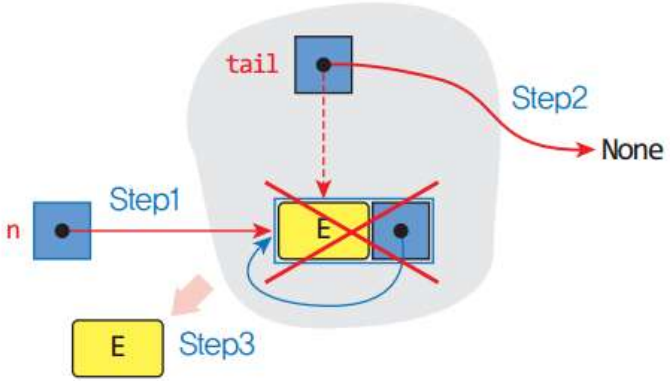
Case2: 큐가 공백상태가 아닌 경우의 삽입연산



삭제 연산: dequeue()

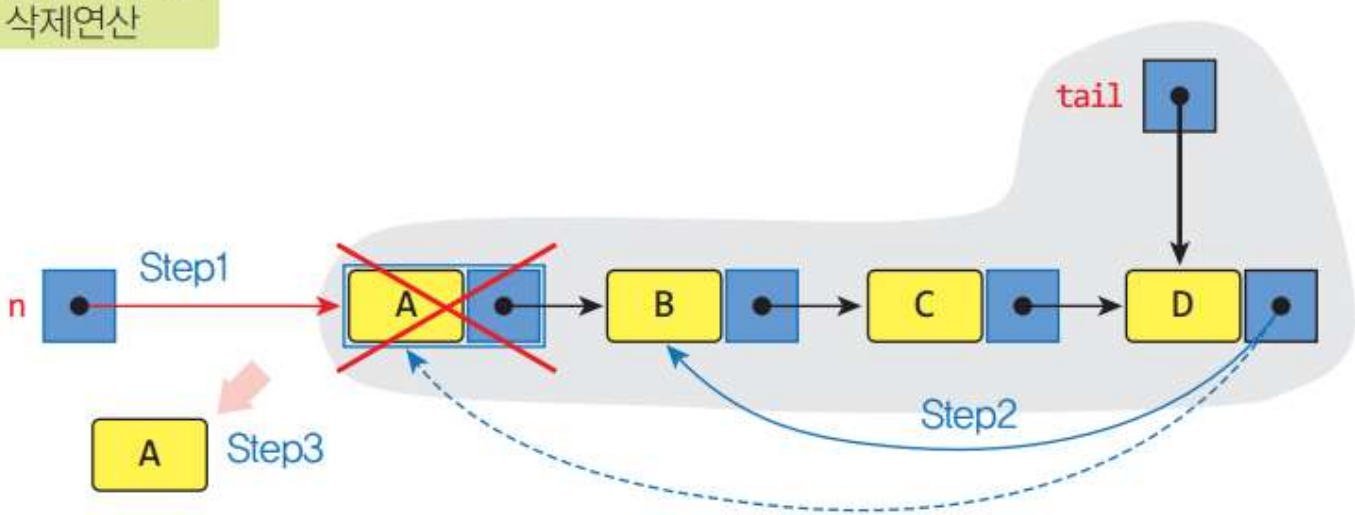


Case1: 큐가 하나의 항목을 갖는 경우의 삭제연산

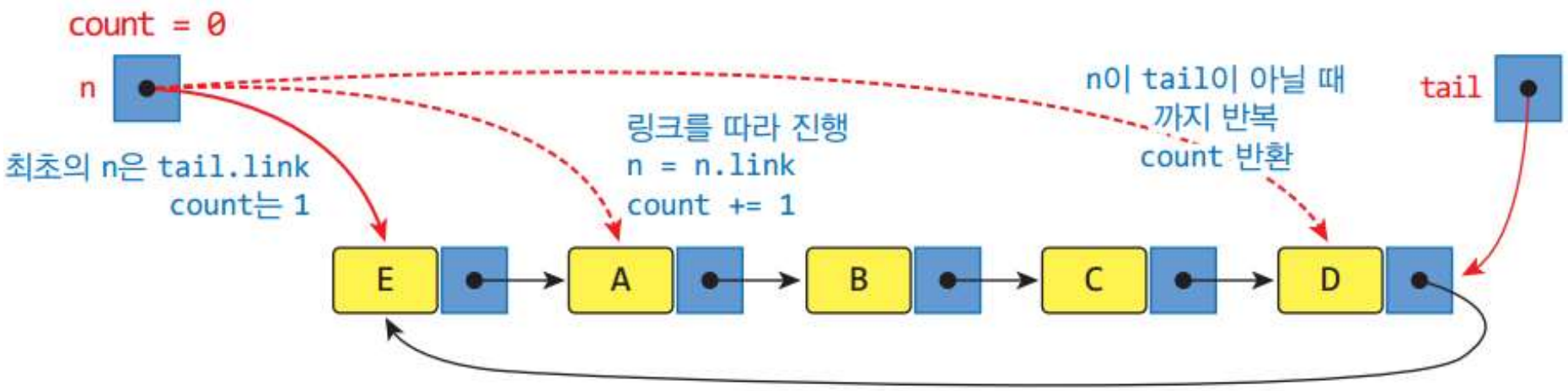



```
def dequeue( self ):
    if not self.isEmpty():
        data = self.tail.link.data
        if self.tail.link == self.tail :
            self.tail = None
        else:
            self.tail.link = self.tail.link.link
        return data
```

Case2: 큐가 여러 개의 항목을 갖는 경우의 삭제연산

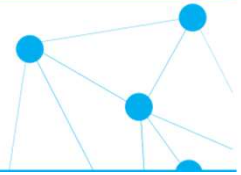


전체 노드의 방문




```
def size( self ):
    if self.isEmpty() : return 0          # 공백: 0반환
    else :                                # 공백이 아니면
        count = 1                         # count는 최소 1
        node = self.tail.link             # node는 front부터 출발
        while not node == self.tail:      # node가 rear가 아닌 동안
            node = node.link              # 이동
            count += 1                    # count 증가
        return count                      # 최종 count 반환
```

테스트 프로그램



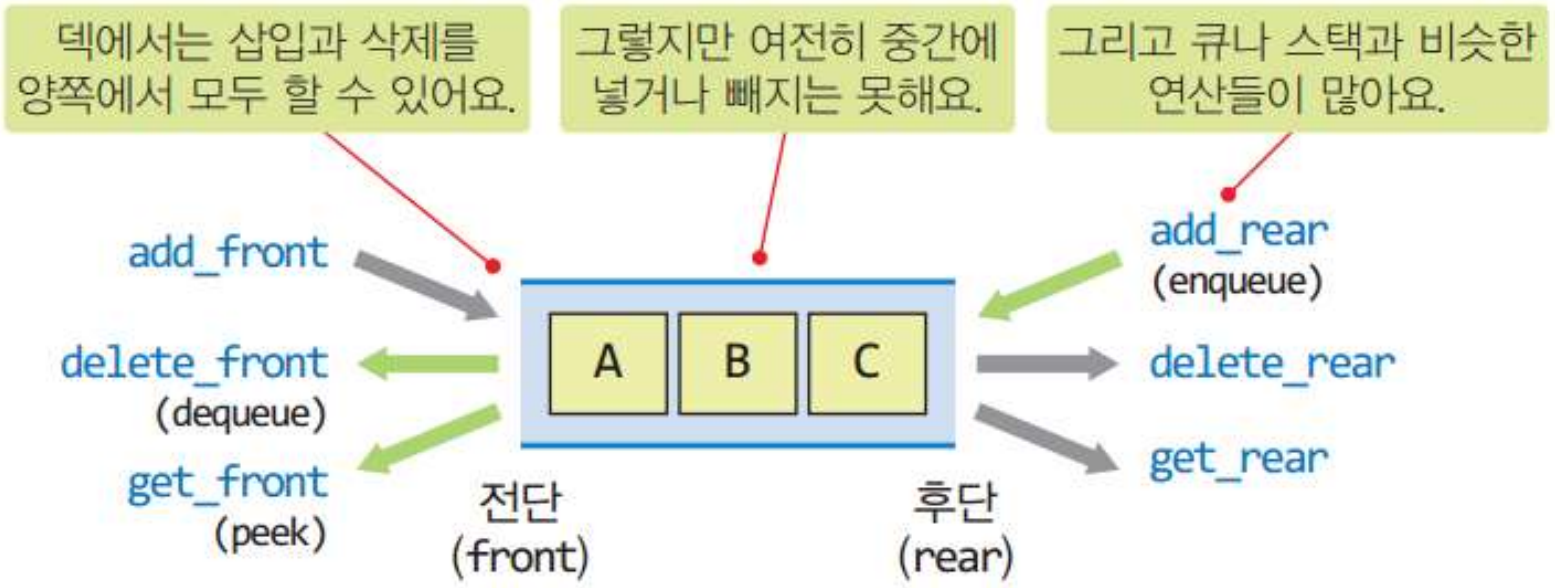
- 원형 큐 테스트 코드와 동일 (객체 생성만 다름)

```
q = CircularLinkedQueue()        # 연결된 큐 만들기
```

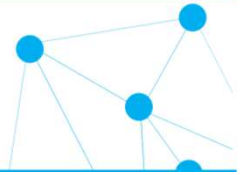
- 용량 제한이 없고, 삽입/삭제가 모두 $O(1)$

덱이란?

- 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



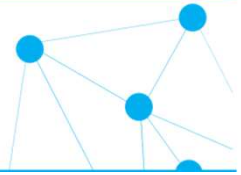
정의 5.2 Deque ADT

데이터: 전단과 후단을 통한 접근을 허용하는 항목들의 모음

연산

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

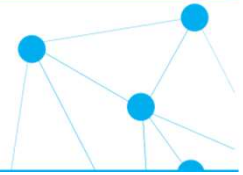
원형 덱의 연산



- 큐와 데이터는 동일함
- 연산은 유사함.

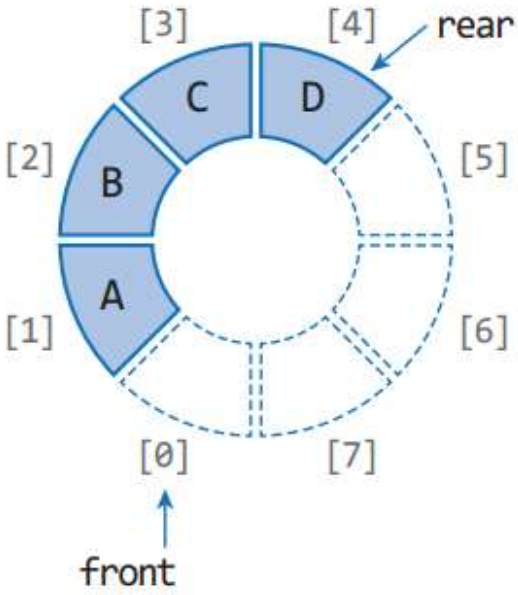
- 큐와 알고리즘이 동일한 연산
  - addRear(), deleteFront(), getFront()
    - 큐의 enqueue, dequeue, peek 연산과 동일
  - 덱의 후단(rear)을 스택의 상단(top)으로 사용하면 addRear(), deleteRear(), getRear() 연산은
    - 스택의 push, pop, peek 연산과 정확히 동일하다.

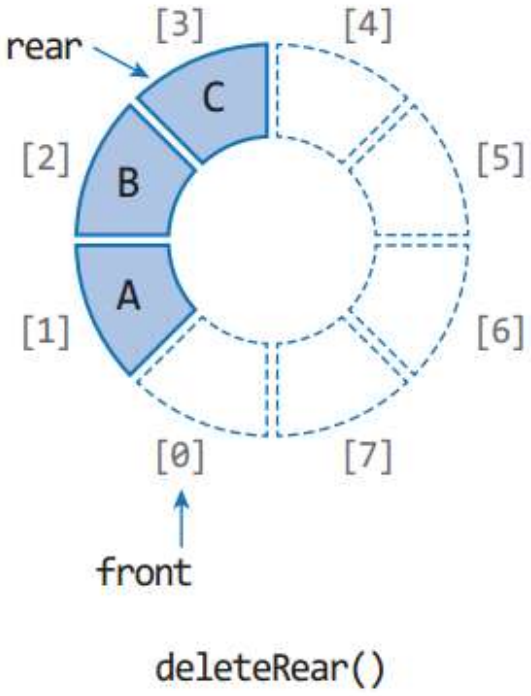
원형 큐에서 추가된 연산

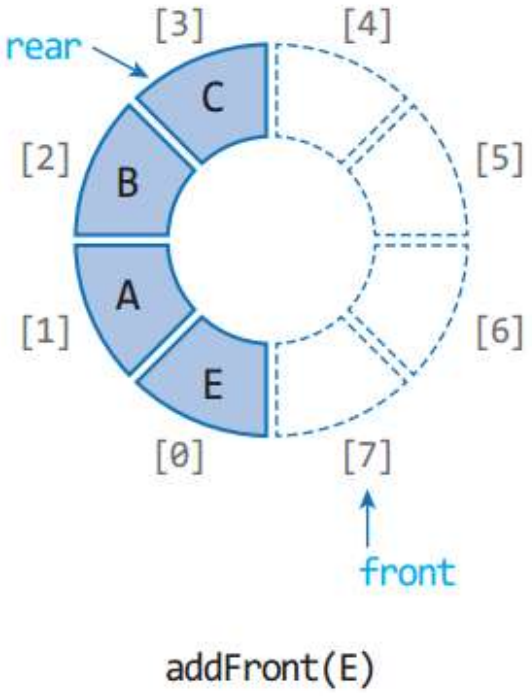


- delete_rear(), add_front(), get_rear()
  - 반 시계방향 회전 필요

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







- 파이썬에는 데크가 Collections 패키지에 정의되어 있음
- 삽입, 삭제 등의 연산은 파이썬의 리스트의 연산들과 매우 유사

```
01 from collections import deque
02 dq = deque('data')
03 for elem in dq:
04     print(elem.upper(), end='')
05 print()
06 dq.append('r')
07 dq.appendleft('k')
08 print(dq)
09 dq.pop()
10 dq.popleft()
11 print(dq[-1])
12 print('x' in dq)
13 dq.extend('structure')
14 dq.extendleft(reversed('python'))
15 print(dq)
```

새 데크 객체를 생성

맨 뒤와 맨 앞에 항목 삽입

맨 뒤와 맨 앞의 항목 삭제

맨 뒤의 항목 출력

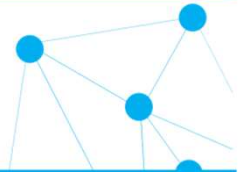
맨 뒤와 맨 앞에 여러 항목 삽입

Console PyUnit

<terminated> deque.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32₩python.exe]

```
DATA
deque(['k', 'd', 'a', 't', 'a', 'r'])
a
False
deque(['p', 'y', 't', 'h', 'o', 'n', 'd', 'a', 't', 'a', 's', 't', 'r', 'u', 'c', 't', 'u', 'r', 'e'])
```

덱의 구현



- 원형 큐를 상속하여 원형 덱 클래스를 구현

```
class CircularDeque(CircularQueue) :      # CircularQueue에서 상속
```

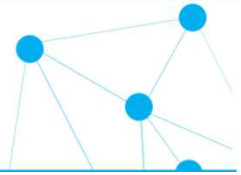
- 덱의 생성자 (상속되지 않음)

```
def __init__( self ) :              # CircularDeque의 생성자
    super().__init__()              # 부모 클래스의 생성자를 호출함
```

  - front, rear, items와 같은 멤버 변수는 추가로 선언하지 않음
  - 자식클래스에서 부모를 부르는 함수가 super()
- 재 사용 멤버들: isEmpty, isFull, size, clear
- 인터페이스 변경 멤버들

```
def addRear( self, item ): self.enqueue(item )     # enqueue 호출
def deleteFront( self ): return self.dequeue()     # 반환에 주의
def getFront( self ): return self.peek()           # 반환에 주의
```

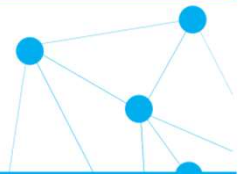
원형 덱의 구현



- 추가로 구현할 메소드

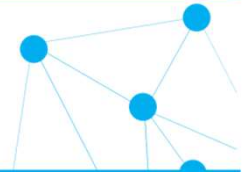
```
def addFront( self, item ):                    # 새로운 기능: 전단 삽입
    if not self.isFull():
        self.items[self.front] = item          # 항목 저장
        self.front = self.front - 1            # 반시계 방향으로 회전
        if self.front < 0 : self.front = MAX_QSIZE - 1

def deleteRear( self ):                        # 새로운 기능: 후단 삭제
    if not self.isEmpty():
        item = self.items[self.rear];          # 항목 복사
        self.rear = self.rear - 1              # 반시계 방향으로 회전
        if self.rear < 0 : self.rear = MAX_QSIZE - 1
        return item                            # 항목 반환

def getRear( self ):                           # 새로운 기능: 후단 peek
    return self.items[self.rear]
```

테스트 프로그램



```
dq = CircularDeque()                    # 덱 객체 생성. f=r=0
for i in range(9):                      # i : 0, 1, 2, ... 8
      if i%2==0 : dq.addRear(i)         # 짝수는 후단에 삽입:
      else : dq.addFront(i)             # 홀수는 전단에 삽입
dq.display()                            # front=6,rear=5
for i in range(2): dq.deleteFront()     # 전단에서 두 번의 삭제: f=8
for i in range(3): dq.deleteRear()      # 후단에서 세 번의 삭제: r=2
dq.display()
for i in range(9,14): dq.addFront(i)    # i : 9, 10, ... 13 : f=3
dq.display()
```

```
C:₩WINDOWS₩system32₩cmd.exe
[f=6,r=5] ==>  [7, 5, 3, 1, 0, 2, 4, 6, 8]
[f=8,r=2] ==>  [3, 1, 0, 2]
[f=3,r=2] ==>  [13, 12, 11, 10, 9, 3, 1, 0, 2]
```



감사합니다