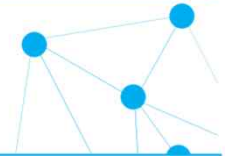


8주차 온라인 수업

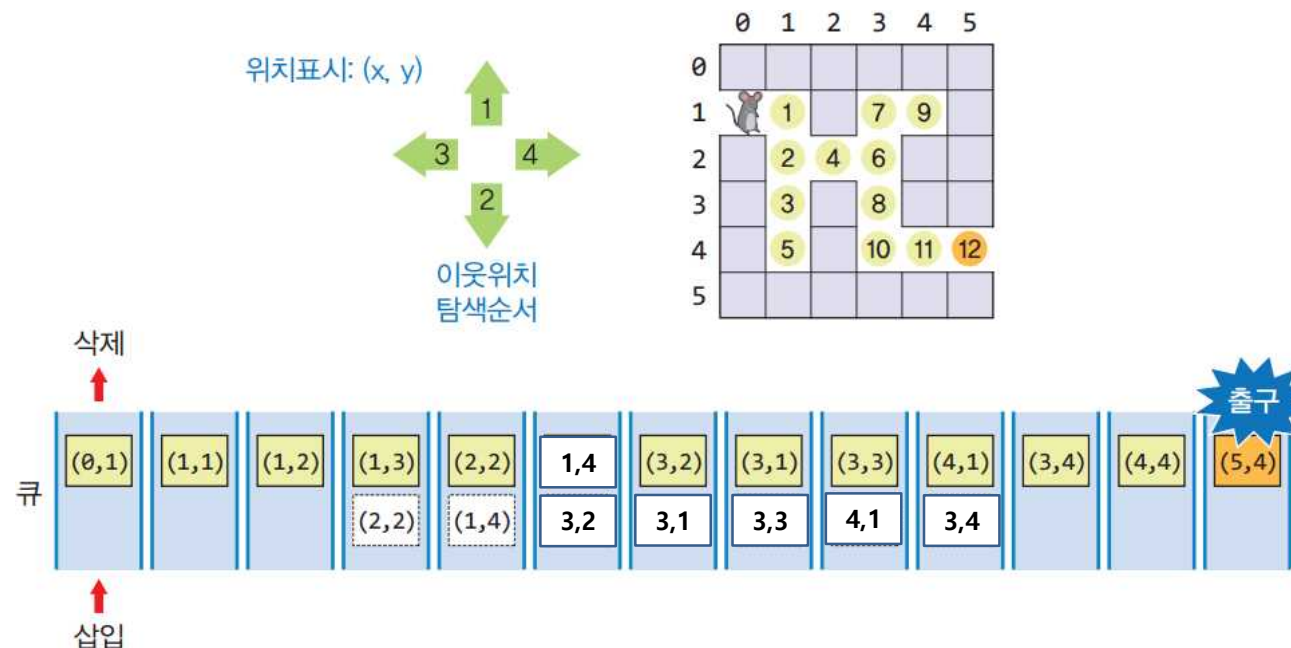
자료구조/알고리즘

큐 (QUEUE)

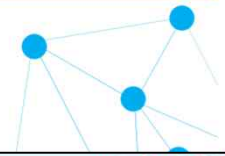
큐의 응용: 너비우선탐색



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



너비우선탐색 알고리즘



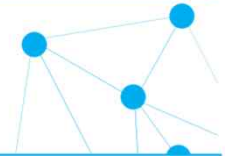
```
def BFS() :                                # 너비우선탐색 함수
    que = CircularQueue()
    que.enqueue((0,1))
    print('BFS: ')                          # 출력을 'BFS'로 변경

    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
    stack = Stack()                         # 사용할 스택
    stack.push( (0,1) )                    # 시작위치 삽입
    print('DFS: ')

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

테스트 프로그램



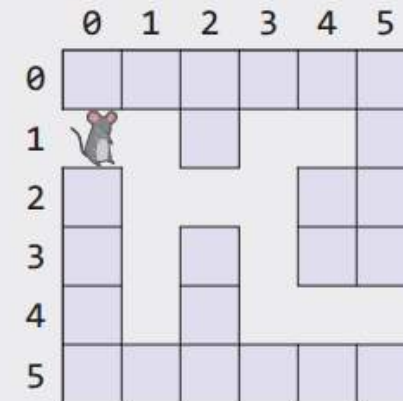
```
map = [ [ '1', '1', '1', '1', '1', '1' ],  
        [ 'e', '0', '1', '0', '0', '1' ],  
        [ '1', '0', '0', '0', '1', '1' ],  
        [ '1', '0', '1', '0', '1', '1' ],  
        [ '1', '0', '1', '0', '0', 'x' ],  
        [ '1', '1', '1', '1', '1', '1' ] ]
```

```
MAZE_SIZE = 6
```

```
result = BFS()
```

```
if result : print(' --> 미로탐색 성공')
```

```
else : print(' --> 미로탐색 실패')
```



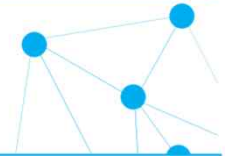
C:\WINDOWS\system32\cmd.exe

너비우선탐색 과정

BFS:

(0, 1) -> (1, 1) -> (1, 2) -> (1, 3) -> (2, 2) -> (1, 4) -> (3, 2) -> (3, 1) -> (3, 3) -> (4, 1) -> (3, 4) -> (4, 4) -> (5, 4) -> --> 미로탐색 성공

파이썬의 queue 모듈



- 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()
```

```
C:\WINDOWS\system32\cmd.exe  
큐의 내용: 1 2 3 4 5 6 7 8 9
```

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\wsbyang\AppData\Local\Programs\Python\Python36-32\python

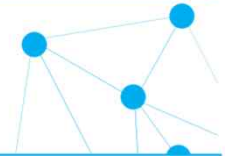
사과, 오렌지, 체리, 배 삽입 후: front: apple -> orange -> cherry -> pear : rear

remove한 후: front: orange -> cherry -> pear : rear

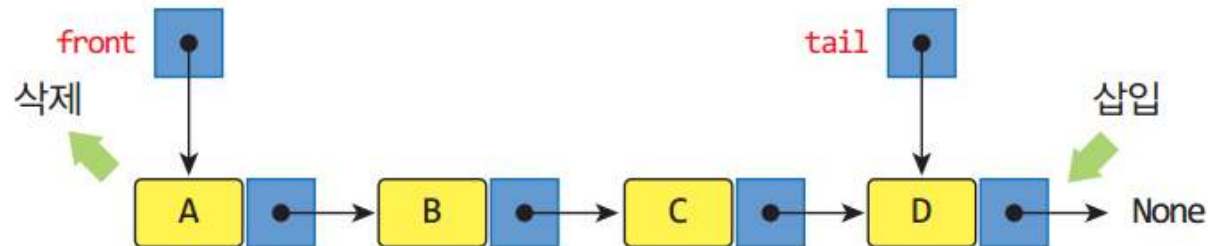
remove한 후: front: cherry -> pear : rear

포도 삽입 후: front: cherry -> pear -> grape : rear

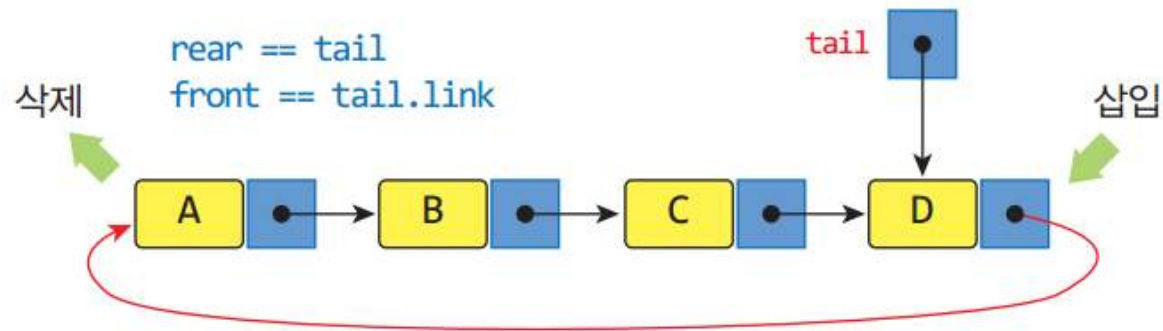
원형 연결리스트의 응용: 연결된 큐



- 단순 연결리스트로 구현한 큐

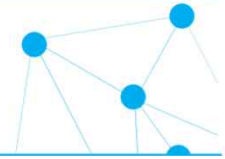


- 원형 연결리스트로 구현한 큐



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

연결된 큐 클래스



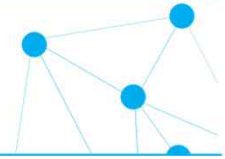
```
class CircularLinkedList:
    def __init__( self ):
        self.tail = None

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

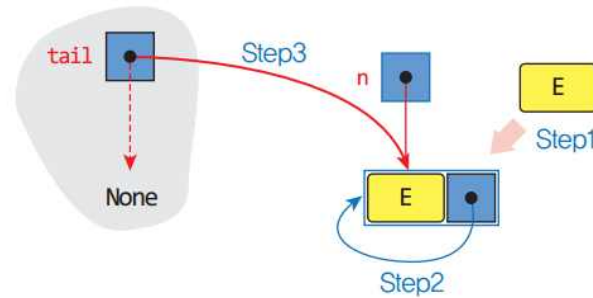
생성자 함수
tail: 유일한 데이터

공백상태 검사
큐 초기화
peek 연산
공백이 아니면
front의 data를 반환

삽입 연산: enqueue()

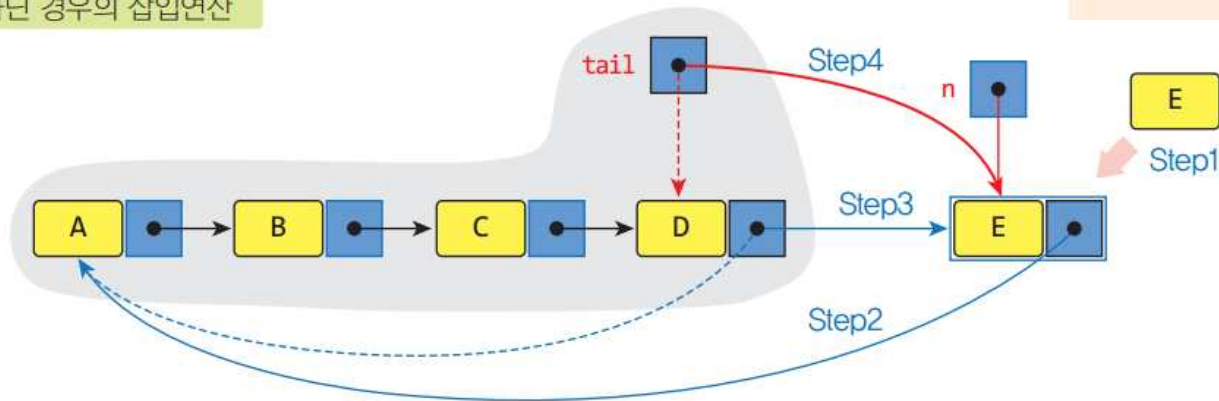


Case1: 큐가 공백상태인
경우의 삽입연산

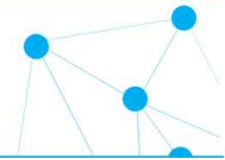


```
def enqueue( self, item ):  
    node = Node(item, None)  
    if self.isEmpty() :  
        node.link = node  
        self.tail = node  
    else :  
        node.link = self.tail.link  
        self.tail.link = node  
        self.tail = node
```

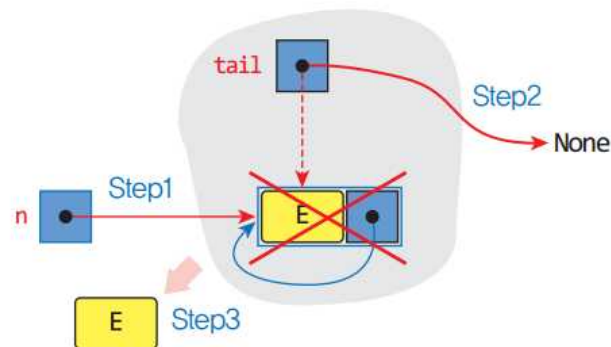
Case2: 큐가 공백상태가
아닌 경우의 삽입연산



삭제 연산: dequeue()

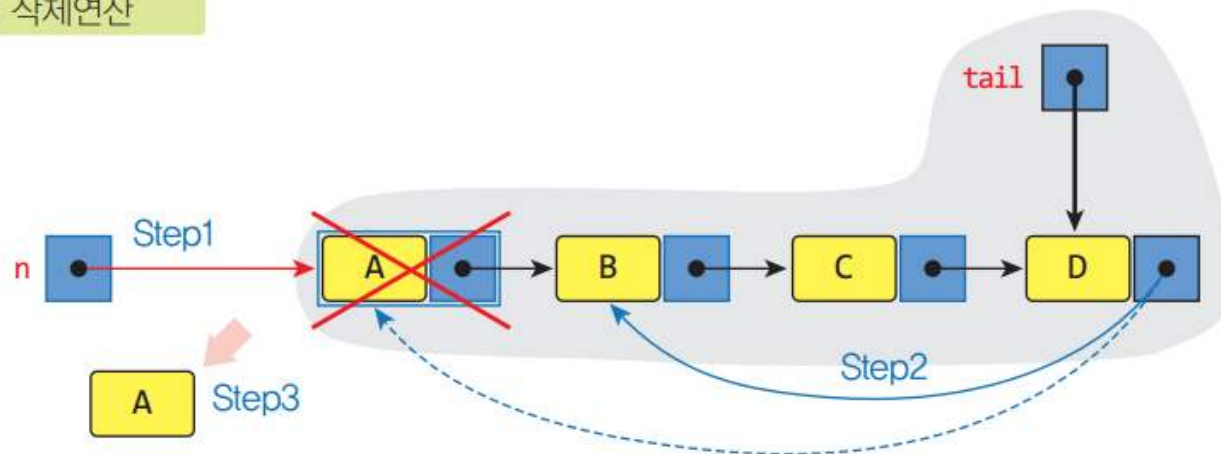


Case1: 큐가 하나의 항목을
갖는 경우의 삭제연산

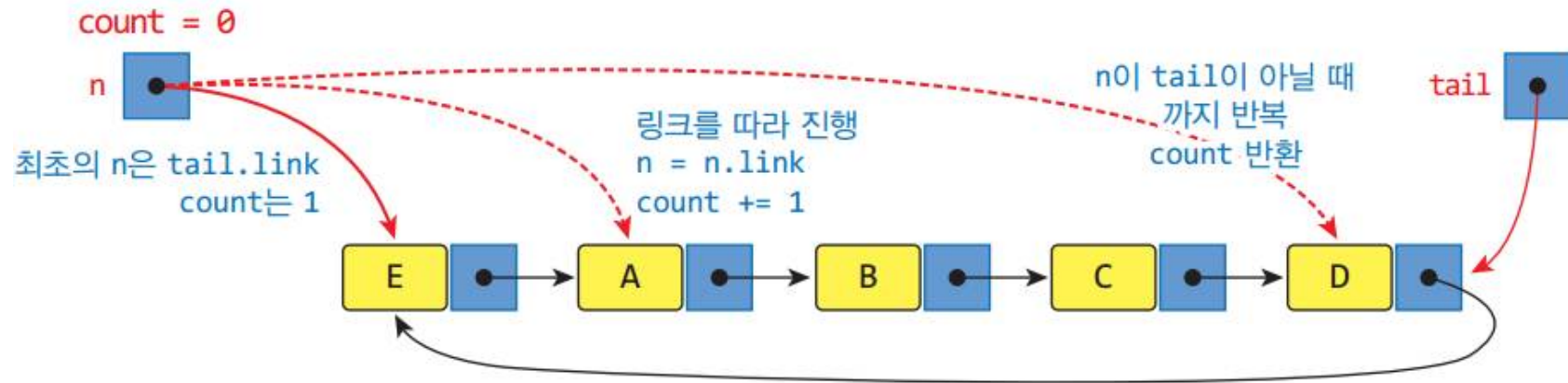


```
def dequeue( self ):  
    if not self.isEmpty():  
        data = self.tail.link.data  
        if self.tail.link == self.tail :  
            self.tail = None  
        else:  
            self.tail.link = self.tail.link.link  
    return data
```

Case2: 큐가 여러 개의 항목을
갖는 경우의 삭제연산



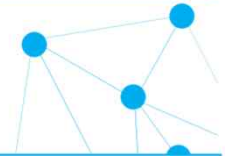
전체 노드의 방문



```
def size( self ):  
    if self.isEmpty() : return 0  
    else :  
        count = 1  
        node = self.tail.link  
        while not node == self.tail:  
            node = node.link  
            count += 1  
        return count
```

공백: 0반환
공백이 아니면
count는 최소 1
node는 front부터 출발
node가 rear가 아닌 동안
이동
count 증가
최종 count 반환

테스트 프로그램



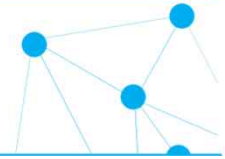
- 원형 큐 테스트 코드와 동일 (객체 생성만 다름)

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

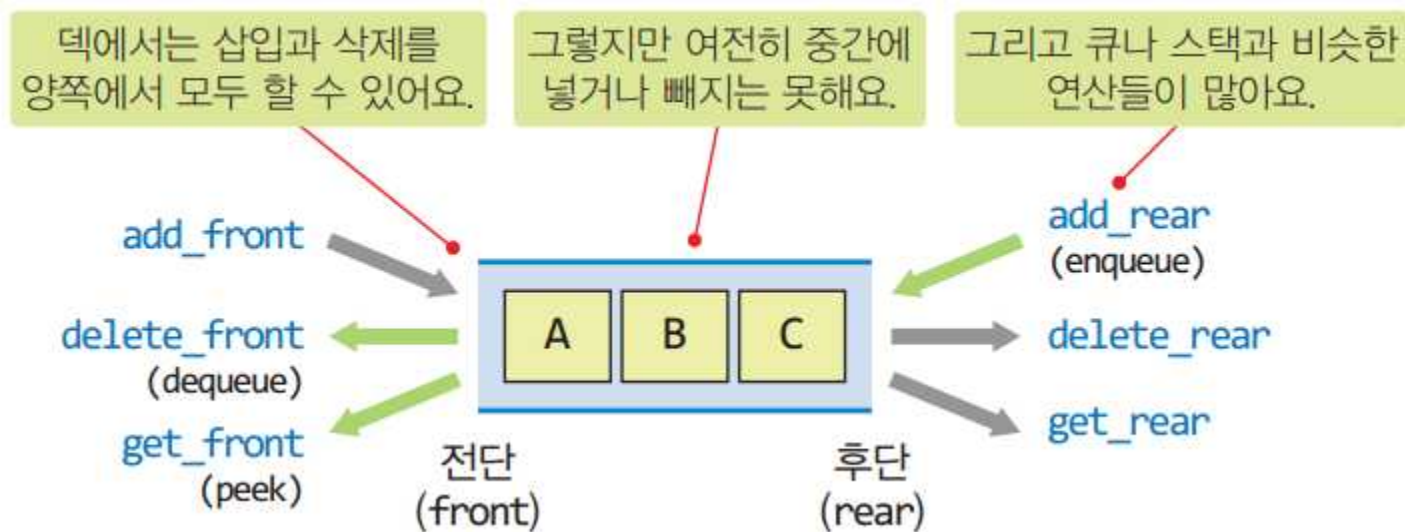
```
C:\WINDOWS\system32\cmd.exe
CircularLinkedList:0 1 2 3 4 5 6 7
CircularLinkedList:5 6 7
CircularLinkedList:5 6 7 8 9 10 11 12 13
```

- 용량 제한이 없고, 삽입/삭제가 모두 $O(1)$

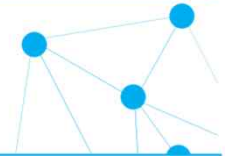
덱이란?



- 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

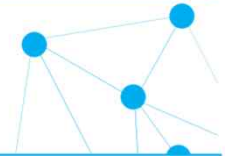


정의 5.2 Deque ADT

데이터: 전단과 후단을 통한 접근을 허용하는 항목들의 모임
연산

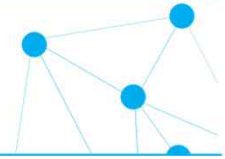
- `Deque()`: 비어 있는 새로운 덱을 만든다.
- `isEmpty()`: 덱이 비어있으면 `True`를 아니면 `False`를 반환한다.
- `addFront(x)`: 항목 `x`를 덱의 맨 앞에 추가한다.
- `deleteFront()`: 맨 앞의 항목을 꺼내서 반환한다.
- `getFront()`: 맨 앞의 항목을 꺼내지 않고 반환한다.
- `addRear(x)`: 항목 `x`를 덱의 맨 뒤에 추가한다.
- `deleteRear()`: 맨 뒤의 항목을 꺼내서 반환한다.
- `getRear()`: 맨 뒤의 항목을 꺼내지 않고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

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.

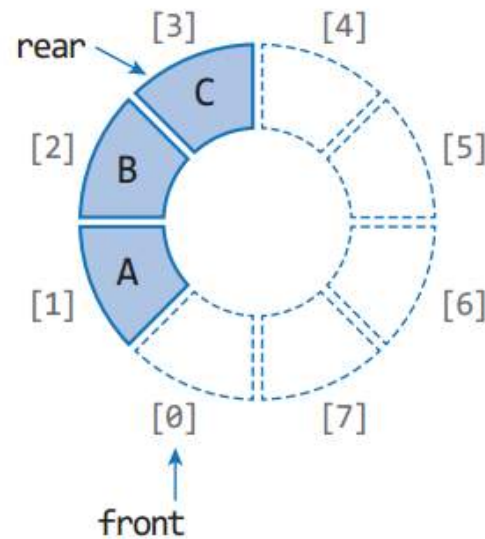
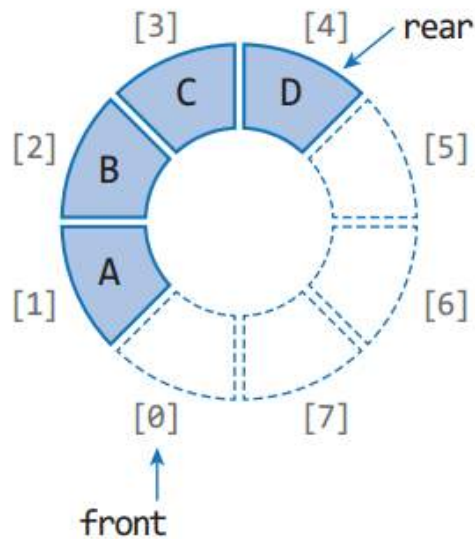
원형 큐에서 추가된 연산



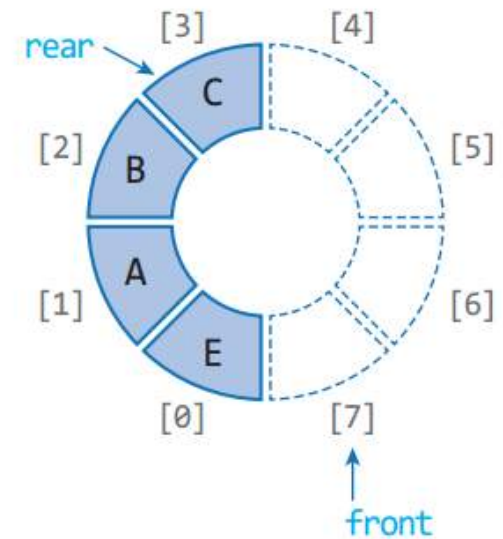
- delete_rear(), add_front(), get_rear()
 - 반 시계방향 회전 필요

$\text{front} \leftarrow (\text{front} - 1 + \text{MAX_QSIZE}) \% \text{MAX_QSIZE}$

$\text{rear} \leftarrow (\text{rear} + 1 + \text{MAX_QSIZE}) \% \text{MAX_QSIZE}$



deleteRear()



addFront(E)

- 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\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

DATA

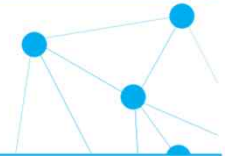
deque(['k', 'd', 'a', 't', 'a', 'r'])

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
```

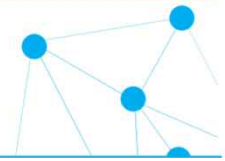
- 덱의 생성자 (상속되지 않음)

```
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



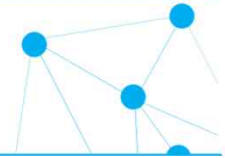
- 추가로 구현할 메소드

```
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



```

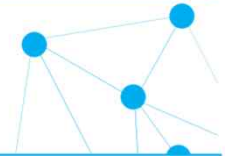
dq = CircularDeque()
for i in range(9):
    if i%2==0 : dq.addRear(i)
    else : dq.addFront(i)
dq.display()
for i in range(2): dq.deleteFront()
for i in range(3): dq.deleteRear()
dq.display()
for i in range(9,14): dq.addFront(i)
dq.display()
```

덱 객체 생성. f=r=0
i : 0, 1, 2, ... 8
짝수는 후단에 삽입:
홀수는 전단에 삽입
front=6, rear=5
전단에서 두 번의 삭제: f=8
후단에서 세 번의 삭제: r=2
i : 9, 10, ... 13 : f=3

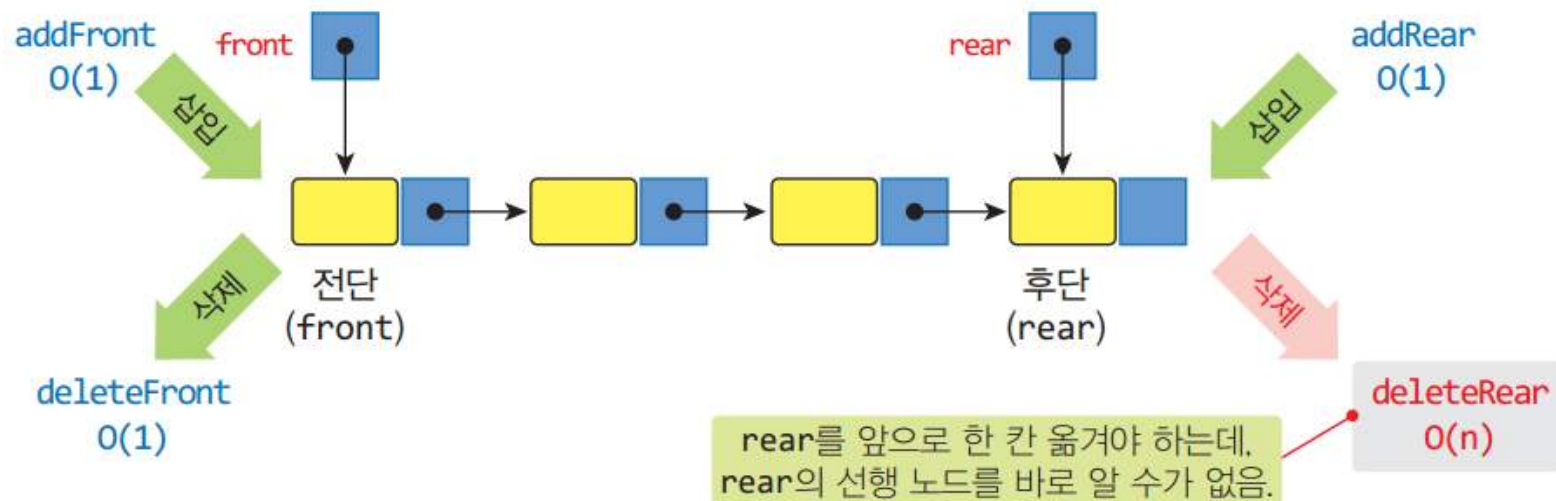
```

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]
```

이중연결리스트의 응용: 연결된 덱

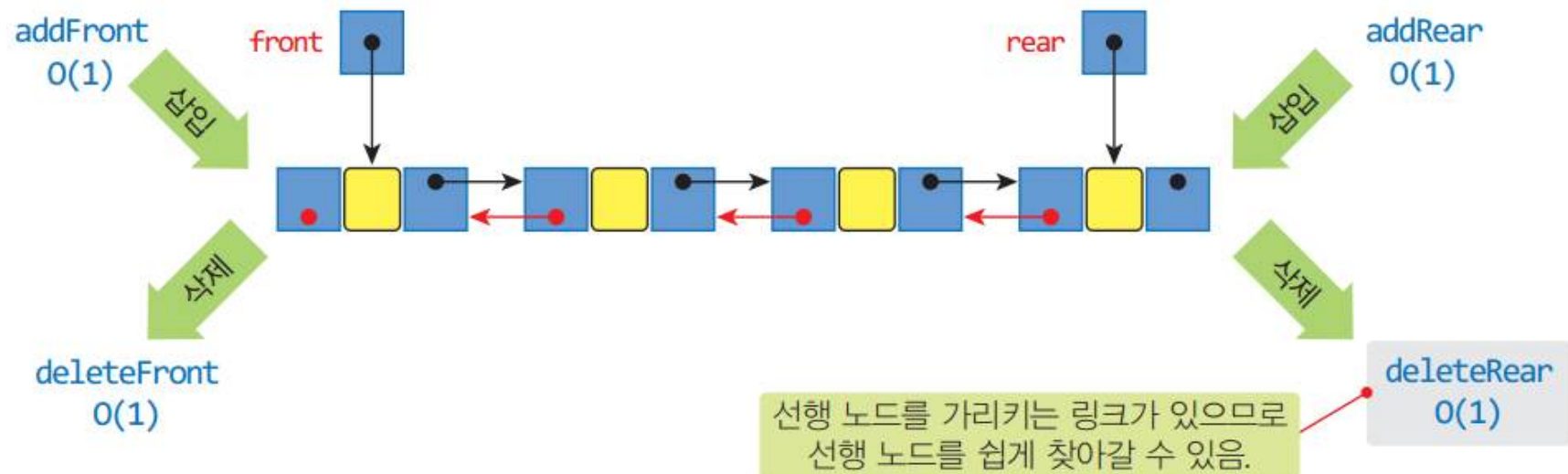
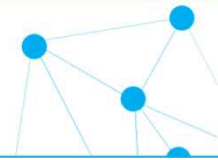


- 단순연결리스트로 구현한 덱



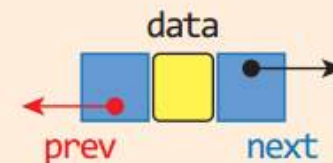
- 해결 방안은?
 - 이중연결리스트 사용

이중연결리스트로 구현한 덱

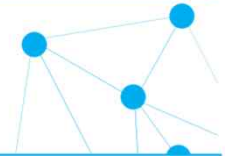


- 이중연결리스트를 위한 노드

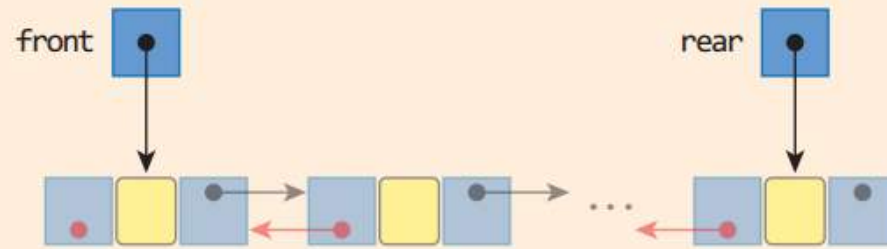
```
class DNode:                                # 이중연결리스트를 위한 노드
    def __init__(self, elem, prev = None, next = None):
        self.data = elem
        self.prev = prev
        self.next = next
```



연결된 덱 클래스

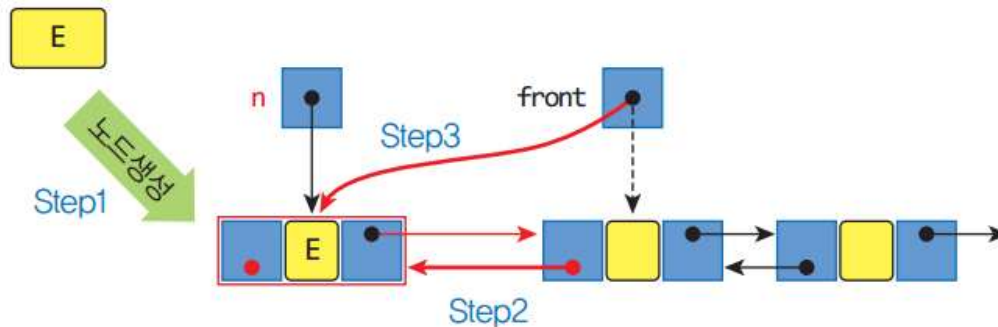


```
class DoublyLinkedDeque:  
    def __init__( self ):  
        self.front = None  
        self.rear = None
```



```
def isEmpty( self ) : return self.front == None           # 공백상태 검사  
def clear( self ) : self.front = self.rear = None        # 초기화  
def size( self ) : ...                                    # self.top->self.front로 수정. 코드 동일  
def display(self, msg):...                               # self.top->self.front로 수정. 코드 동일
```

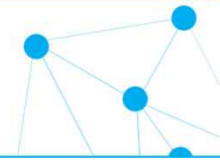
addFront(), addRear()



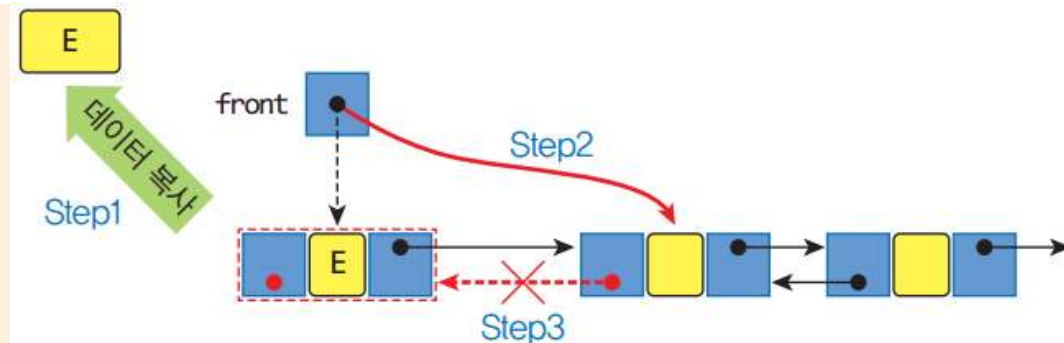
```
def addFront( self, item ):
    node = DNode(item, None, self.front)      # Step1
    if( self.isEmpty()):                      # 공백이면
        self.front = self.rear = node         # front와 rear가 모두 node
    else :                                    # 공백이 아니면
        self.front.prev = node                # Step2
        self.front = node                    # Step3
```

```
def addRear( self, item ):
    node = DNode(item, self.rear, None)       # Step1
    if( self.isEmpty()):                     # 공백이면
        self.front = self.rear = node         # front와 rear가 모두 node
    else :                                    # 공백이 아니면
        self.rear.next = node                 # Step2
        self.rear = node                     # Step3
```


deleteFront(), deleteRear()



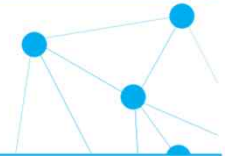
```
def deleteFront( self ):
    if not self.isEmpty():
        data = self.front.data
        self.front = self.front.next
    if self.front==None :
        self.rear = None
    else:
        self.front.prev = None
    return data
```



```
def deleteRear( self ):
    if not self.isEmpty():
        data = self.rear.data
        self.rear = self.rear.prev
    if self.rear==None :
        self.front = None
    else:
        self.rear.next = None
    return data
```

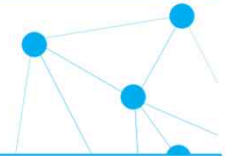
Step1
Step2
노드가 하나 뿐이면
front도 None으로 설정
Step3
Step4

테스트 프로그램



```
dq = DoublyLinkedDeque()
```

```
# 연결된 덱 만들기
```



감사합니다