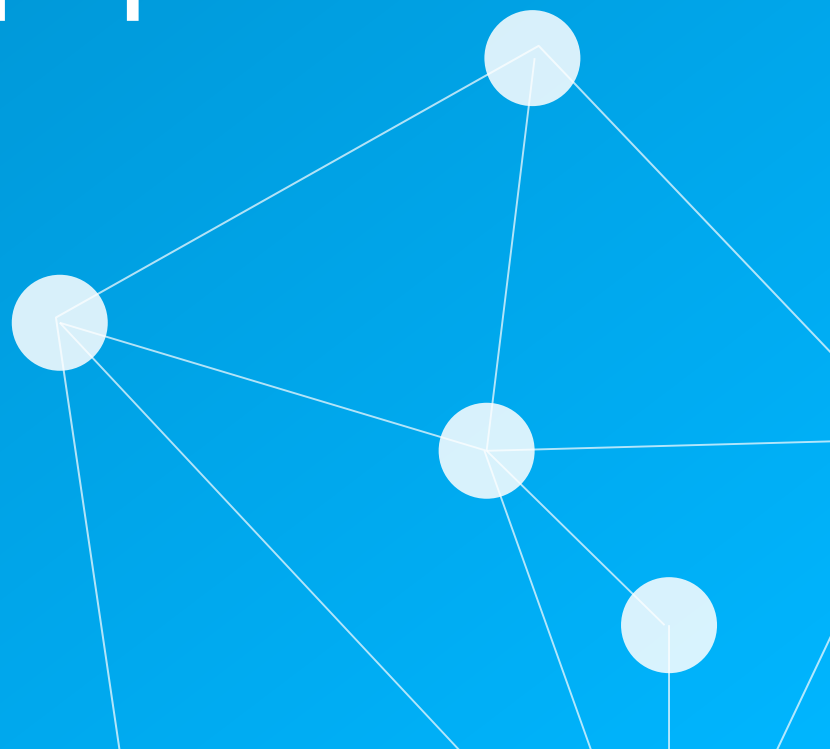
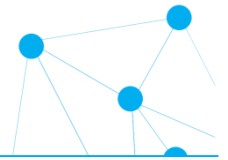

파이썬
자료구조

05^{CHAPTER}

큐와 덱

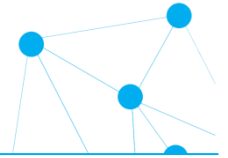


5장. 학습 목표



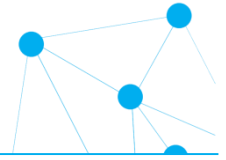
- 선형 큐의 문제와 원형 큐의 구조와 동작 원리를 이해한다.
- 덱과 우선순위 큐의 개념과 동작 원리를 이해한다.
- 파이썬 리스트를 이용한 큐, 덱, 우선순위 큐의 구현 방법을 이해한다.
- 상속을 이용하여 새로운 클래스를 만들고 사용하는 방법을 이해한다.
- 우선순위 큐를 이용한 전략적 미로 탐색 방법을 이해한다.

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

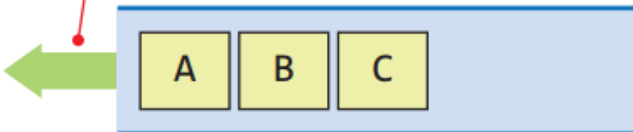


- 큐의 구조

삭제는 이쪽으로만 할 수 있어요.

삽입은 이쪽으로만 해야 합니다.

전단
(front)



후단
(rear)

큐 ADT



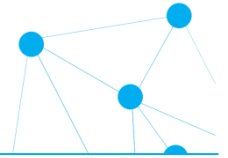
- 삽입과 삭제는 FIFO순서를 따른다.
- 삽입은 큐의 후단에서, 삭제는 전단에서 이루어진다.

정의 5.1 Queue ADT

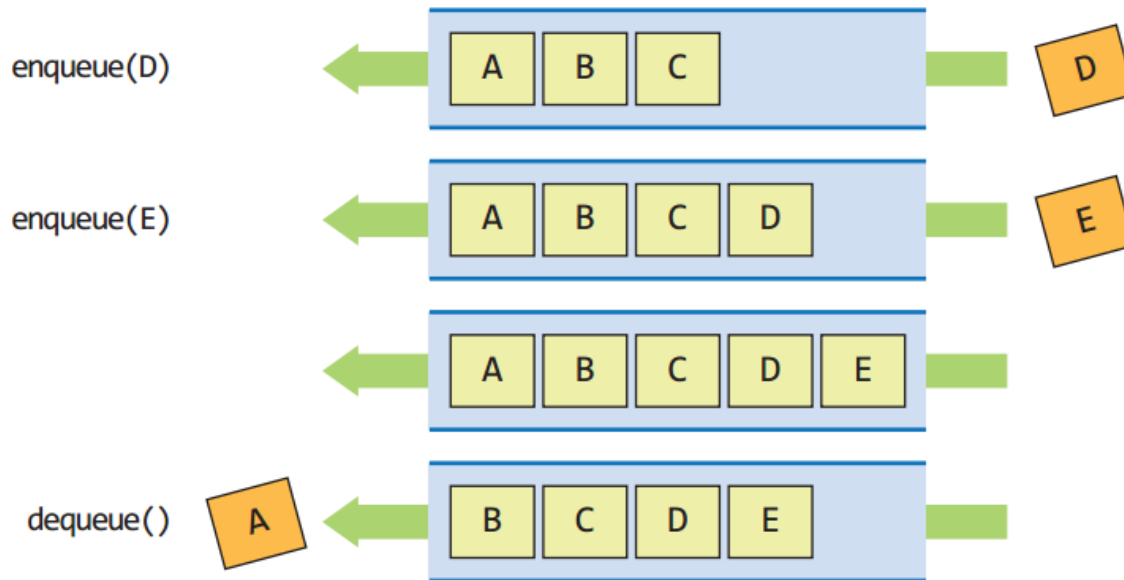
데이터: 선입선출(FIFO)의 접근 방법을 유지하는 항목들의 모임
연산

- `Queue()`: 비어 있는 새로운 큐를 만든다.
- `isEmpty()`: 큐가 비어있으면 `True`를 아니면 `False`를 반환한다.
- `enqueue(x)`: 항목 `x`를 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

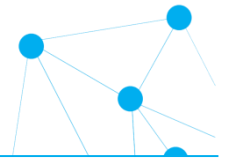


• 예) 서비스센터의 콜 큐



- 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

선형큐의 문제점

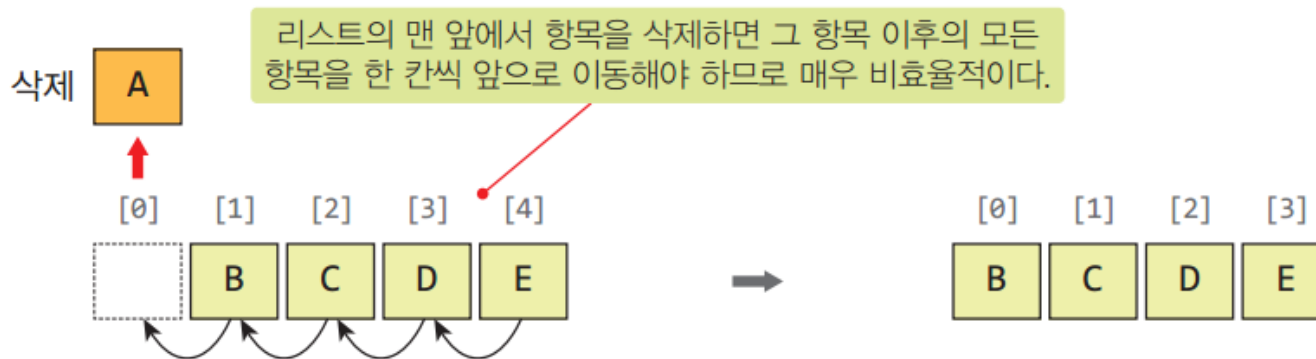


- 선형큐는 비효율적이다.
 - enqueue(item): 삽입 연산 → $O(1)$

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

- dequeue(): 삭제 연산 → $O(n)$ why?

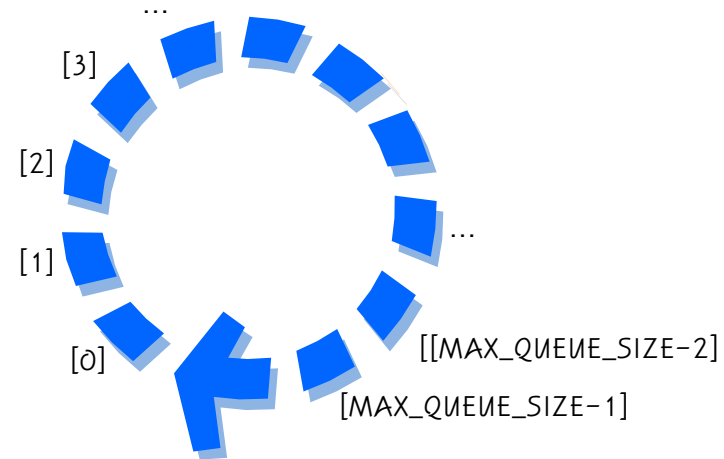
```
def dequeue():  
    if not isEmpty():           # 공백상태가 아니면  
        return items.pop(0)     # 맨 앞 항목을 꺼내서 반환
```



원형 큐가 훨씬 효율적이다.



- 원형큐
 - 배열을 원형으로 사용

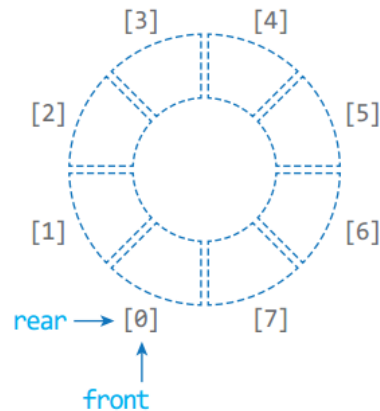
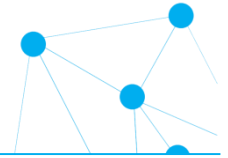


- 전단과 후단을 위한 2개의 변수
 - front: 첫번째 요소 하나 앞의 인덱스
 - rear: 마지막 요소의 인덱스

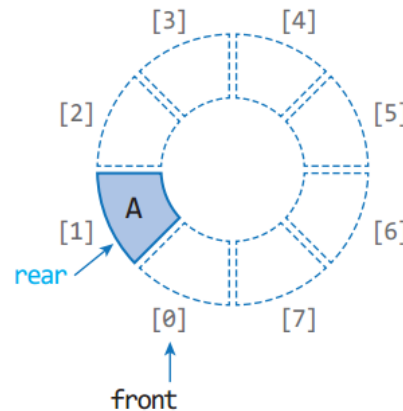
- 회전(시계방향) 방법

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

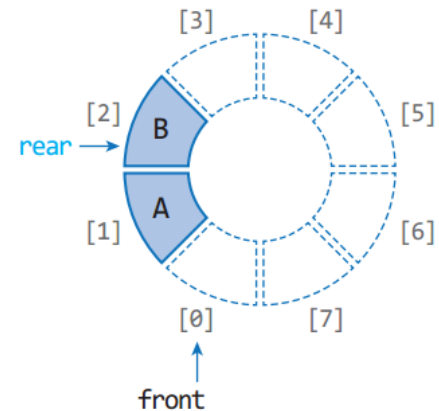
원형 큐의 삽입과 삭제 과정



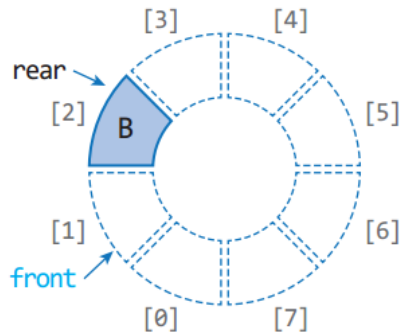
초기상태



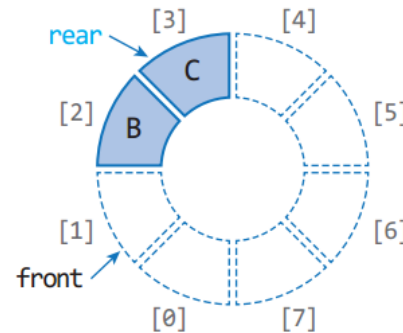
enqueue(A)



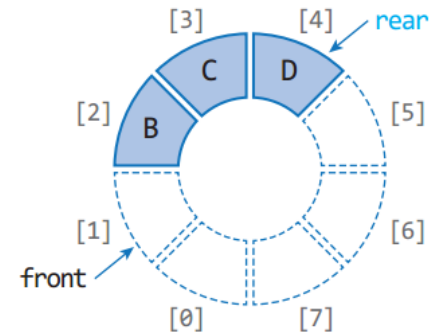
enqueue(B)



dequeue()



enqueue(C)



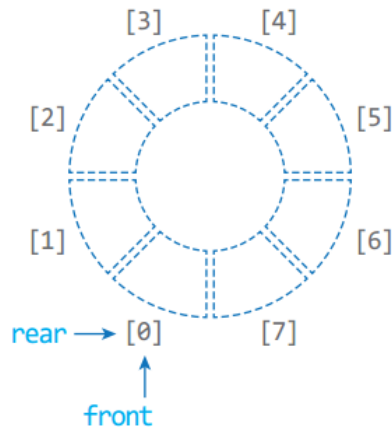
enqueue(D)

삽입: rear를 하나 증가시킨후 그 위치에 삽입
삭제: front를 하나 증가시킨후 그 위치 항목 반환

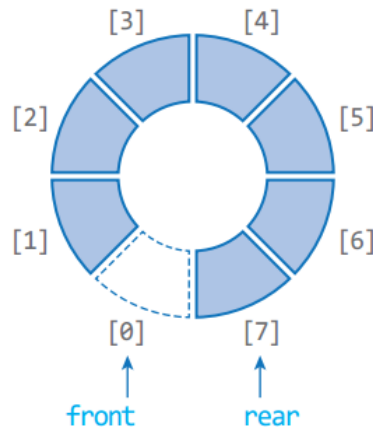
공백상태와 **포화**상태



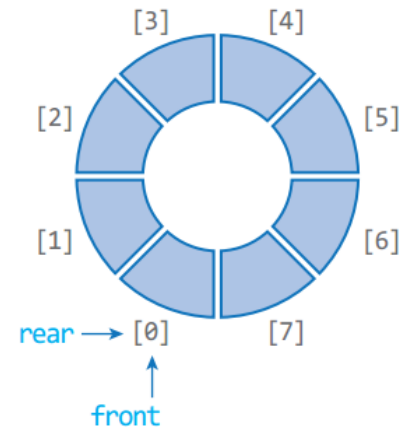
- 공백상태: $front == rear$
- 포화상태: $front == (rear+1) \% M$
- 공백상태와 포화상태를 구별 방법은?
 - **하나의 공간은 항상 비워둠**



(a) 공백 상태



(c) 포화 상태



(b) 오류 상태

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
    def clear( self ) : self.front = self.rear
```

원형 큐의 연산들(삽입/삭제)



```
def enqueue( self, item ):
    if not self.isFull():
        self.rear = (self.rear+1)% MAX_QSIZE
        self.items[self.rear] = item

def dequeue( self ):
    if not self.isEmpty():
        self.front = (self.front+1)% MAX_QSIZE
        return self.items[self.front]

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
```

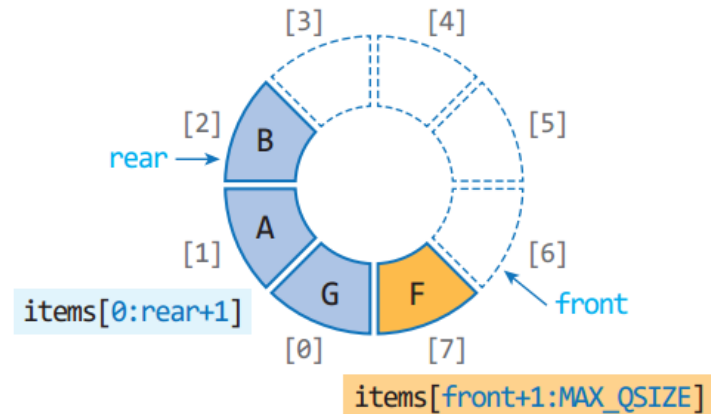
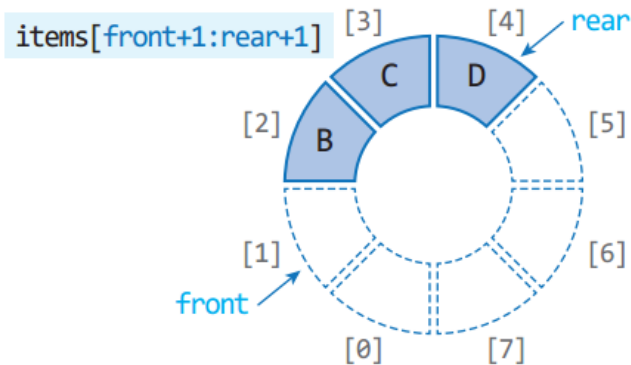
포화상태가 아니면
rear 회전
rear 위치에 삽입

공백상태가 아니면
front 회전
front위치의 항목 반환

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] W     # 다음 줄에 계속...
               + self.items[0:self.rear+1]             # 슬라이싱
    print("[f=%s,r=%d] ==>"%(self.front, self.rear), out)
```



테스트 프로그램

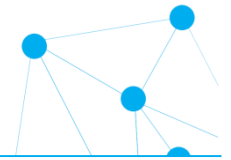


```
q = CircularQueue()
for i in range(8): q.enqueue(i)
q.display()
for i in range(5): q.dequeue();
q.display()
for i in range(8,14): q.enqueue(i)
q.display()
```

원형큐 만들기 (MAX_QSIZE=10)
0, 1, ... 7 삽입(f=0, r=8)
원형 큐에서 구현한 print()호출
5번 삭제 (f=5, r=8)
8, 9, ... 13 삽입 (f=5, r=4)

```
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]
```


5.3 큐의 응용: 너비우선탐색



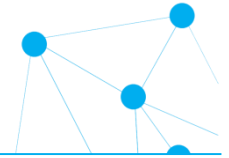
- 큐를 이용한 너비우선탐색
- 파이썬의 queue모듈은 큐와 스택 클래스를 제공한다.

BFS(Breadth First Search): 출발점에서부터 인접 위치들을 먼저 방문한 다음, 방문한 위치들에 인접한 위치들을 순서대로 찾아가는 방법

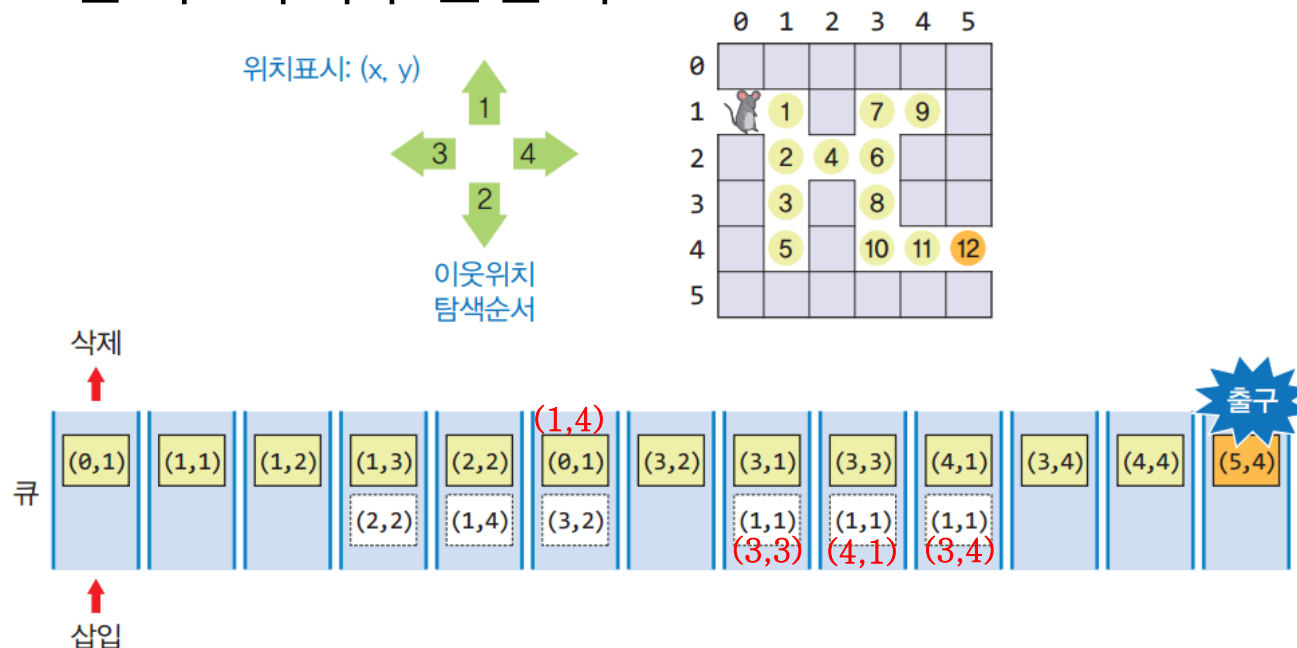
깊이우선탐색: 하나의 경로를 선택해 끝까지 가보고 막히면 가장 최근 경로 시도(스택)

너비우선탐색: 가까운 위치부터 차근차근 찾아가는 전략(큐)

큐의 응용: 너비우선탐색



- 이 책에서의 큐 응용
 - 이진트리의 레벨 순회 (8장)
 - 기수정렬에서 레코드의 정렬을 위해 사용 (12장)
 - 그래프의 탐색에서 너비우선탐색 (10장)
- 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))    # 상
            if isValidPos(x, y + 1) : que.enqueue((x, y + 1))    # 하
            if isValidPos(x - 1, y) : que.enqueue((x - 1, y))    # 좌
            if isValidPos(x + 1, y) : que.enqueue((x + 1, y))    # 우

    return False
```

테스트 프로그램



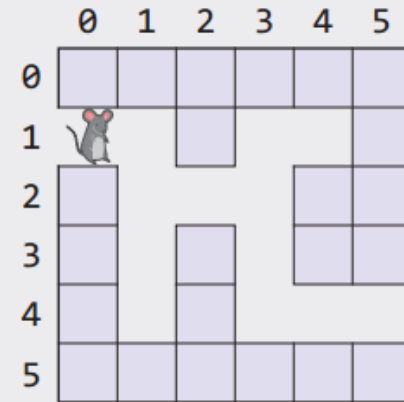
```
map = [ [ '1', '1', '1', '1', '1', '1' ],  
        [ 'e', '0', '1', '0', '0', '1' ],  
        [ '1', '0', '0', '0', '1', '1' ],  
        [ '1', '0', '1', '0', '1', '1' ],  
        [ '1', '0', '1', '0', '0', 'x' ],  
        [ '1', '1', '1', '1', '1', '1' ] ]
```

```
MAZE_SIZE = 6
```

```
result = BFS()
```

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

```
else : print(' --> 미로탐색 실패')
```



```
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 queue모듈



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

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

- 큐 객체 생성

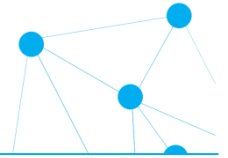
```
Q = queue.Queue(maxsize=20)  # 큐 객체 생성(최대크기 20)
```

- 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()
```

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

5.4 덱이란?

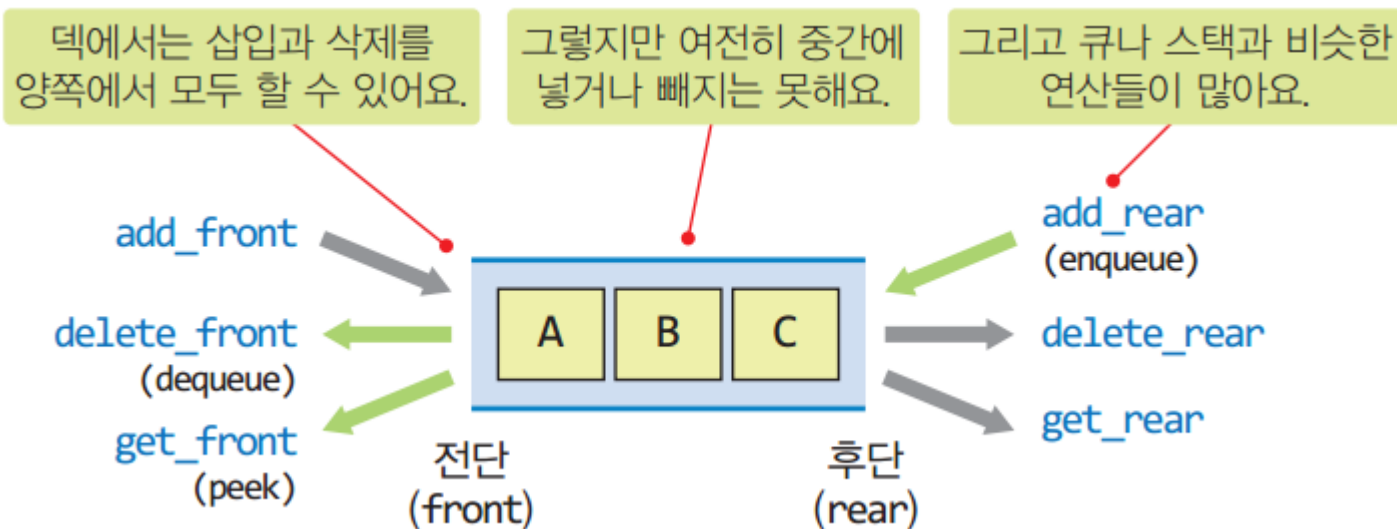


- 덱은 스택이나 큐보다는 입출력이 자유로운 자료구조이다.
 - 덱의 구조
 - 덱 ADT
 - 덱의 연산
- 덱을 스택이나 큐로 사용할 수 있다.

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

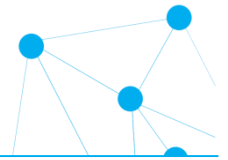


정의 5.2 Deque ADT

데이터: 전단과 후단을 통한 접근을 허용하는 항목들의 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 덱터는 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.

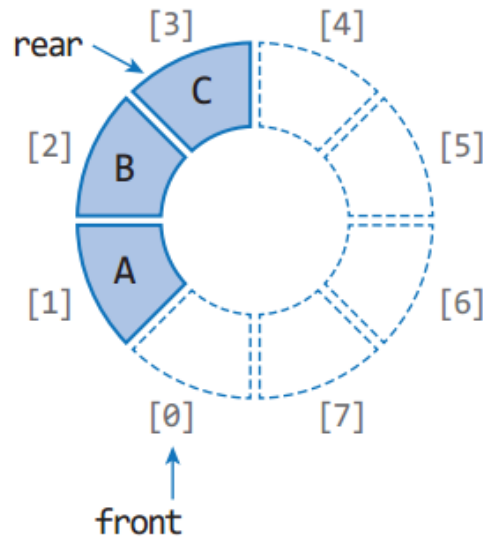
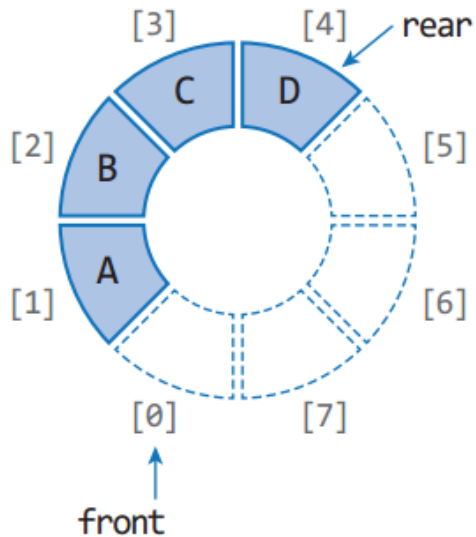
원형 큐에서 추가된 연산



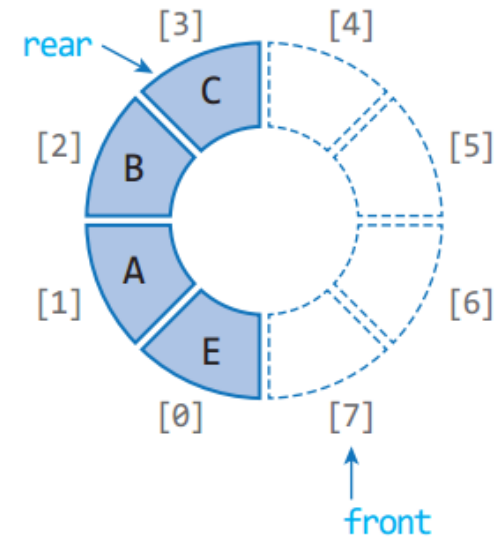
- delete_rear(), add_front(), get_rear()
 - 반 시계방향 회전 필요

$front \leftarrow (front - 1 + MAX_QSIZE) \% MAX_QSIZE$

$rear \leftarrow (rear + 1 + MAX_QSIZE) \% MAX_QSIZE$



deleteRear()



addFront(E)

5.5 덱의 구현



- 원형 큐를 상속하여 원형 덱 클래스를 구현하자.

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



```

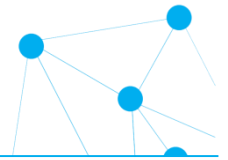
dq = CircularDeque()
for i in range(9):
    if i%2==0 : dq.addRear(i)
    else : dq.addFront(i)
dq.display()
for i in range(2): dq.deleteFront()
for i in range(3): dq.deleteRear()
dq.display()
for i in range(9,14): dq.addFront(i)
dq.display()
```

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]
```

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 스케줄링 등

우선순위 큐 ADT



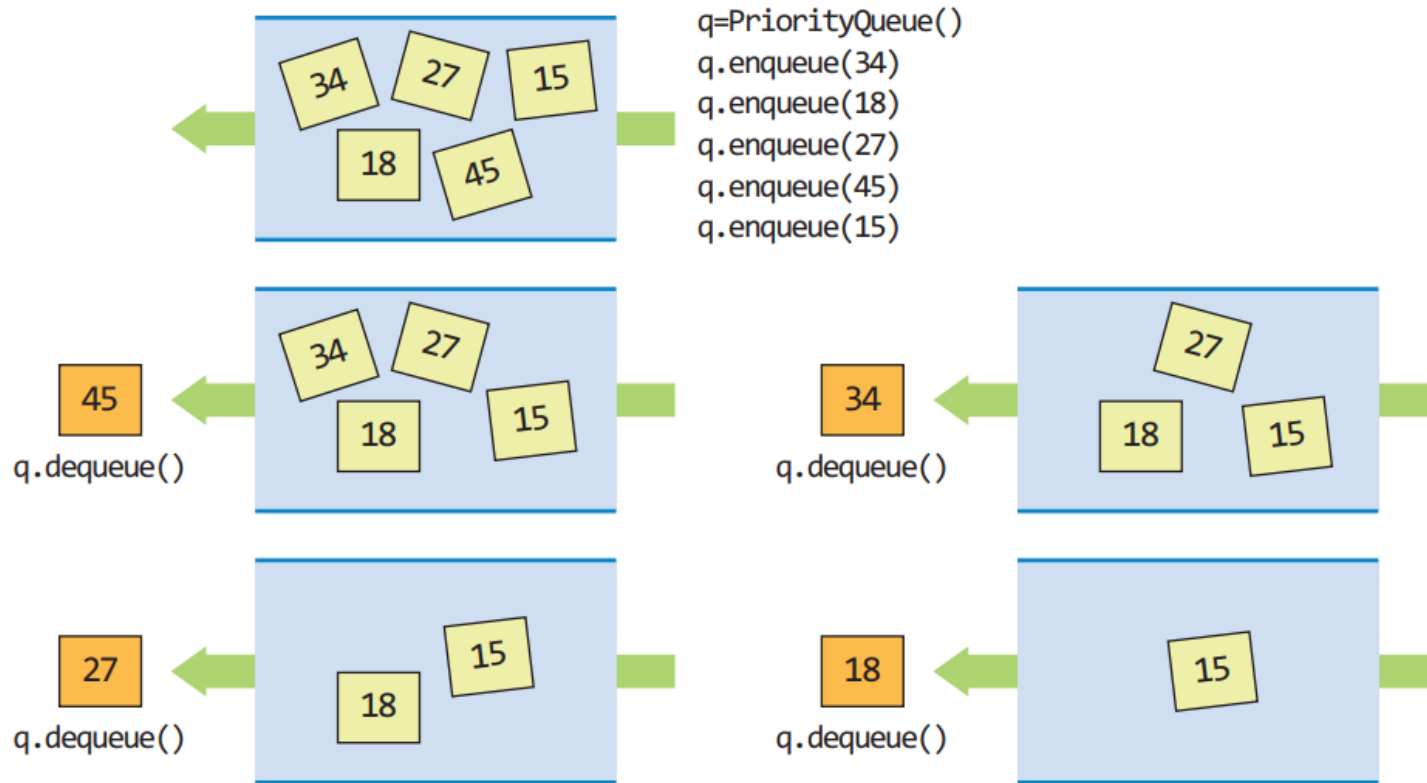
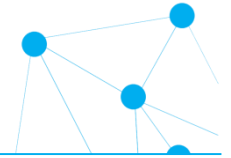
정의 5.3 Priority Queue ADT

데이터: 우선순위를 가진 요소들의 모임

연산

- `PriorityQueue()`: 비어 있는 우선순위 큐를 만든다.
- `isEmpty()`: 우선순위 큐가 공백상태인지를 검사한다.
- `enqueue(e)`: 우선순위를 가진 항목 `e`를 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 ):
```

```
        self.items = []
```

생성자

항목 저장을 위한 리스트

```
    def isEmpty( self ):
```

```
        return len( self.items ) == 0
```

공백상태 검사

```
    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 ):
    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

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

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

최대 우선순위 항목의 인덱스 반환

0번을 최대라고 하고

모든 항목에 대해

최고 우선순위 인덱스 갱신

최고 우선순위 인덱스 반환

삭제 연산

우선순위가 가장 높은 항목

리스트에서 꺼내서 반환

peek 연산

우선순위가 가장 높은 항목

꺼내지 않고 반환

테스트 프로그램



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

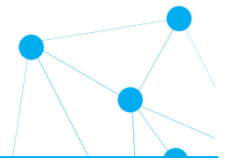
print("PQueue:", q.items)
while not q.isEmpty() :
    print("Max Priority = ", q.dequeue() )
```

```
C:\WINDOWS\system32\cmd.exe
PQueue: [34, 18, 27, 45, 15]
Max Priority = 45
Max Priority = 34
Max Priority = 27
Max Priority = 18
Max Priority = 15
```

입력한 순서 대로 우선순위 큐에 들어감

리스트에서 가장 큰 값을 반환

시간 복잡도



- 정렬되지 않은 리스트 사용
 - enqueue(): 대부분의 경우 $O(1)$
 - findMaxIndex(): $O(n)$
 - dequeue(), peek() : $O(n)$
- 정렬된 리스트 사용
 - enqueue(): $O(n)$
 - dequeue(), peek() : $O(1)$ 우선순위 높은 항목 위치가 정해져~
- 힙 트리(8장)
 - enqueue(), dequeue(): $O(\log n)$
 - peek() : $O(1)$

5.7 우선순위 큐의 응용: 전략적인 미로 탐색



- 전략적 미로 탐색
- 전략적 탐색 알고리즘

전략적 미로 탐색

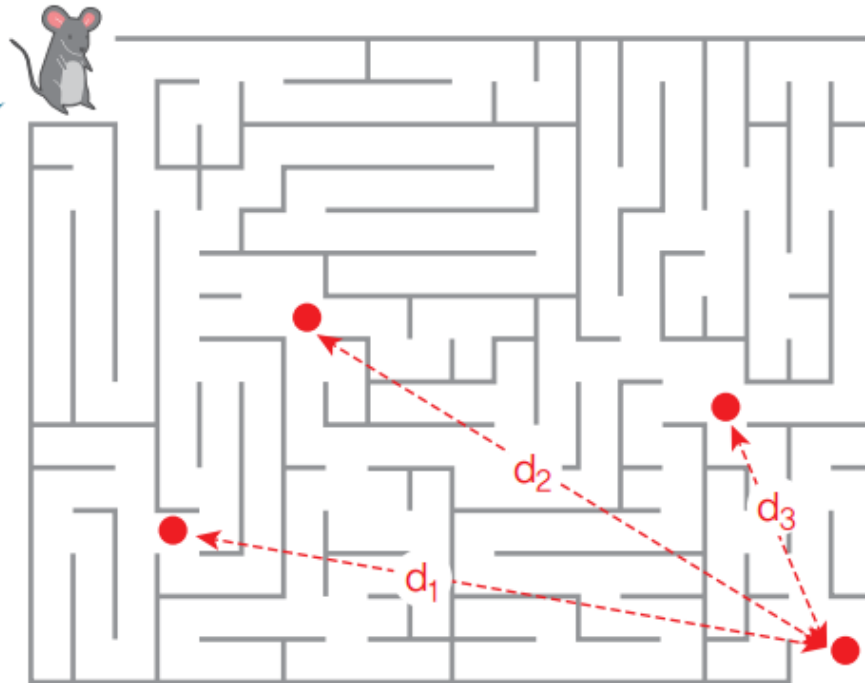


- 전략 (출구의 위치를 알고 있다고 가정 함)
 - 가능한 한 출구와의 가까운 방향을 먼저 선택하자.

Eucledian Distance

개념이 있는 쥐

어디서 다시 시작할까?
아무래도 출구와 가까운 쪽이
유리하겠지?



최대 우선순위 항목 선택



- 큐에 저장되는 항목: $(x, y, -d)$ 형태의 튜플

```
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][2] > self.items[highest][2] :
                highest = i               # 최고 우선순위 인덱스 갱신
        return highest                   # 최고 우선순위 인덱스 반환
```

#항목이 $(x, y, -d)$ 의 튜플의 마지막 요소(2) 이용해 우선순위항목 찾기

전략적 탐색 알고리즘



```
def MySmartSearch() :
```

```
    q = PriorityQueue()
```

```
    q.enqueue((0,1,-dist(0,1)))
```

```
    print('PQueue: ')
```

최소거리 전략의 미로탐색

우선순위 큐 객체 생성

튜플에 거리정보 추가

-d 는 거리를 음수로 가까울수록 더 큰값이 되어야 하기 때문

```
while not q.isEmpty():
```

```
    here = q.dequeue()
```

```
    print(here[0:2], end='->')
```

(x,y,-d)에서 (x,y)만 출력

```
    x,y,_ = here
```

(x,y,-d)에서->(x,y,_)

```
    if (map[y][x] == 'x') : return True
```

```
    else :
```

```
        map[y][x] = '.'
```

```
        if isValidPos(x, y - 1) : q.enqueue((x,y-1, -dist(x,y-1)))
```

```
        if isValidPos(x, y + 1) : q.enqueue((x,y+1, -dist(x,y+1)))
```

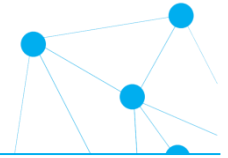
```
        if isValidPos(x - 1, y) : q.enqueue((x-1,y, -dist(x-1,y)))
```

```
        if isValidPos(x + 1, y) : q.enqueue((x+1,y, -dist(x+1,y)))
```

```
    print('우선순위큐: ', q.items)
```

```
return False
```

실행 결과 및 우선순위 큐 응용



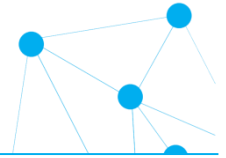
PQueue:

```
(0, 1)→우선순위큐: [(1, 1, -5.0)]
(1, 1)→우선순위큐: [(1, 2, -4.47213595499958), (2, 1, -4.242640687119285)]
(2, 1)→우선순위큐: [(1, 2, -4.47213595499958), (3, 1, -3.605551275463989)]
(3, 1)→우선순위큐: [(1, 2, -4.47213595499958), (3, 2, -2.8284271247461903), (4, 1, -3.1622776601683795)]
(3, 2)→우선순위큐: [(1, 2, -4.47213595499958), (4, 1, -3.1622776601683795), (3, 3, -2.23606797749979)]
(3, 3)→우선순위큐: [(1, 2, -4.47213595499958), (4, 1, -3.1622776601683795), (3, 4, -2.0), (4, 3, -1.4142135623730951)]
(4, 3)→우선순위큐: [(1, 2, -4.47213595499958), (4, 1, -3.1622776601683795), (3, 4, -2.0), (5, 3, -1.0)]
(5, 3)→ → 미로탐색 성공
```

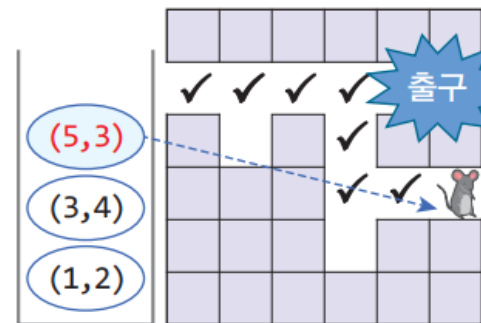
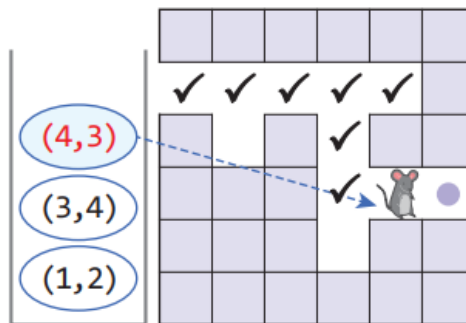
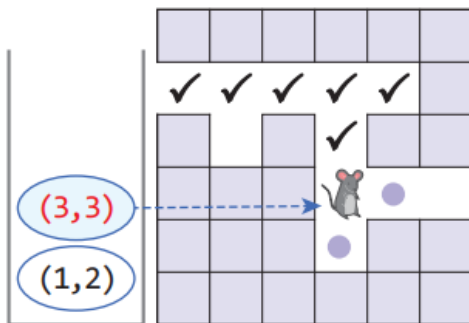
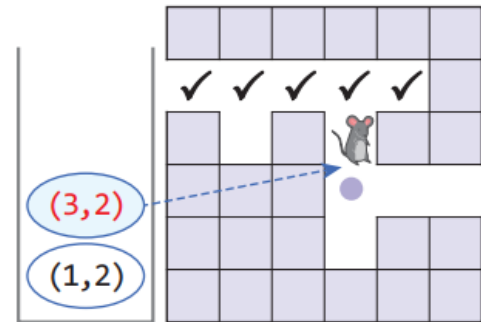
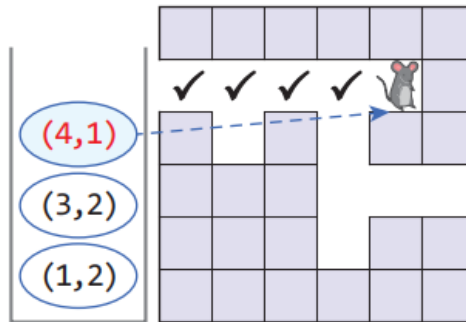
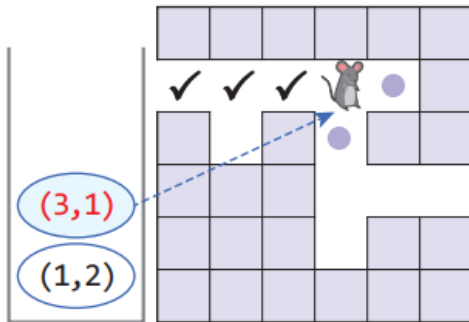
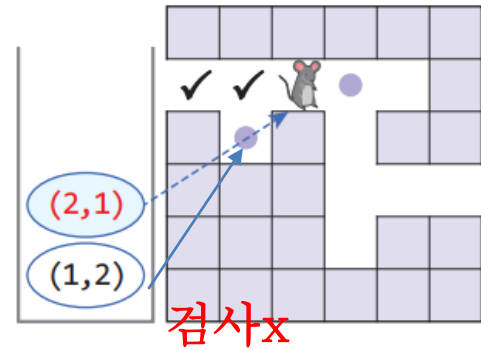
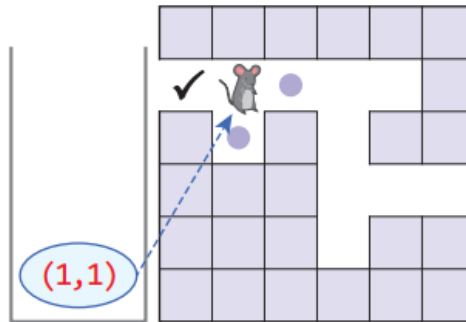
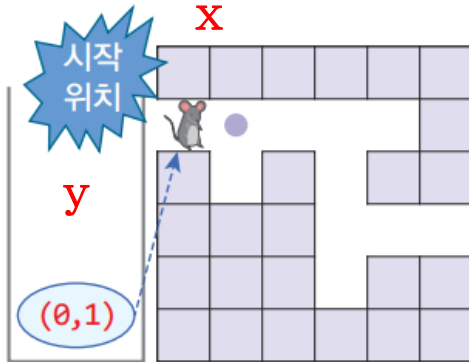
• 우선순위 큐의 주요 응용

- 허프만 코딩 트리: 빈도가 작은 두 노드를 선택 (8.6절)
- Kruskal의 MST 알고리즘: MST에 포함되지 않은 간선 중에 최소 가중치 간선을 선택 (11.3절)
- Dijkstra의 최단거리 알고리즘: 최단거리가 찾아지 지 않은 정점들 중에서 가장 거리가 가까운 정점 선택 (11.4절)
- 인공지능의 A* 알고리즘: 상태 공간트리(state space tree)에서 가장 가능성이 높은 (promising) 경로를 먼저 선택하여 시도

깊이우선탐색: 스택 사용



(x,y)



출구검사

테스트 프로그램



```
result = DFS()
if result : print(' --> 미로탐색 성공')
else : print(' --> 미로탐색 실패')
```

C:\WINDOWS\system32\cmd.exe

DFS:

최종 탐색 순서

가장 최근에 삽입된 항목이 먼저 출력되도록 함

```
(0, 1) -> 현재 스택: [(1, 1)]
(1, 1) -> 현재 스택: [(2, 1), (1, 2)]
(2, 1) -> 현재 스택: [(3, 1), (1, 2)]
(3, 1) -> 현재 스택: [(4, 1), (3, 2), (1, 2)]
(4, 1) -> 현재 스택: [(3, 2), (1, 2)]
(3, 2) -> 현재 스택: [(3, 3), (1, 2)]
(3, 3) -> 현재 스택: [(4, 3), (3, 4), (1, 2)]
(4, 3) -> 현재 스택: [(5, 3), (3, 4), (1, 2)]
(5, 3) -> --> 미로탐색 성공
```

테스트 프로그램



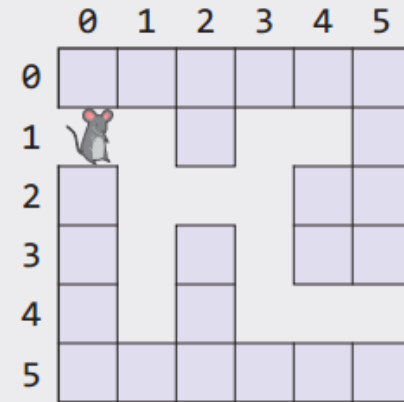
```
map = [ [ '1', '1', '1', '1', '1', '1' ],  
        [ 'e', '0', '1', '0', '0', '1' ],  
        [ '1', '0', '0', '0', '1', '1' ],  
        [ '1', '0', '1', '0', '1', '1' ],  
        [ '1', '0', '1', '0', '0', 'x' ],  
        [ '1', '1', '1', '1', '1', '1' ] ]
```

```
MAZE_SIZE = 6
```

```
result = BFS()
```

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

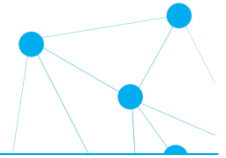
```
else : print(' --> 미로탐색 실패')
```



```
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
```

5장 연습문제, 실습문제





감사합니다!