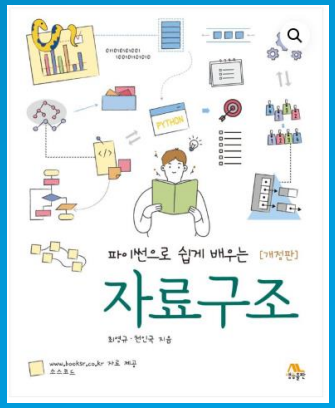



CHAPTER 05

# 큐와 덱

# 5장. 큐와 덱

5.1 큐란?

5.2 큐의 구현

5.3 큐의 응용: 너비우선탐색

5.4 덱이란?

5.5 덱의 구현

5.6 우선순위 큐

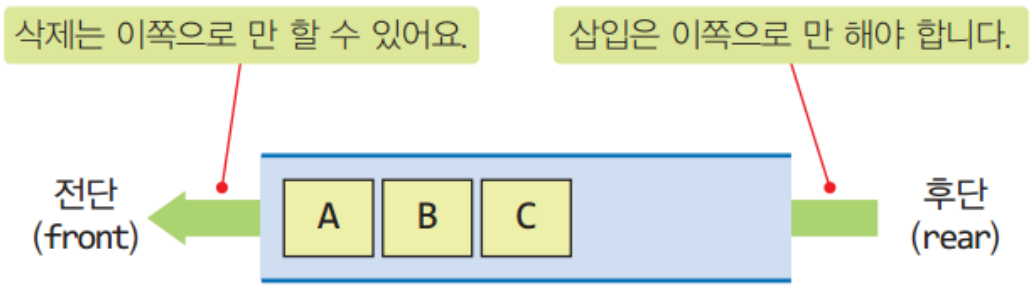
5.7 우선순위 큐의 응용: 전략적인 미로 탐색

# 5.1 큐란?

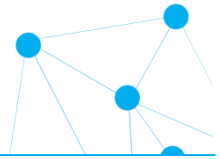
- 선입선출(First-In First Out: FIFO)의 자료구조
  - 먼저 들어온 데이터가 먼저 처리됨
  - 예: 줄서기 대기열

- 큐의 구조
  - 전단: 삭제 통로
  - 후단: 삽입 통로

리스트: 가장 자유로운 형태의 선형 자료구조 (삽입, 삭제 자유로움)
큐:



# 큐의 추상 자료형



데이터: 선입선출(FIFO)의 접근 방법을 유지하는 요소들의 모음

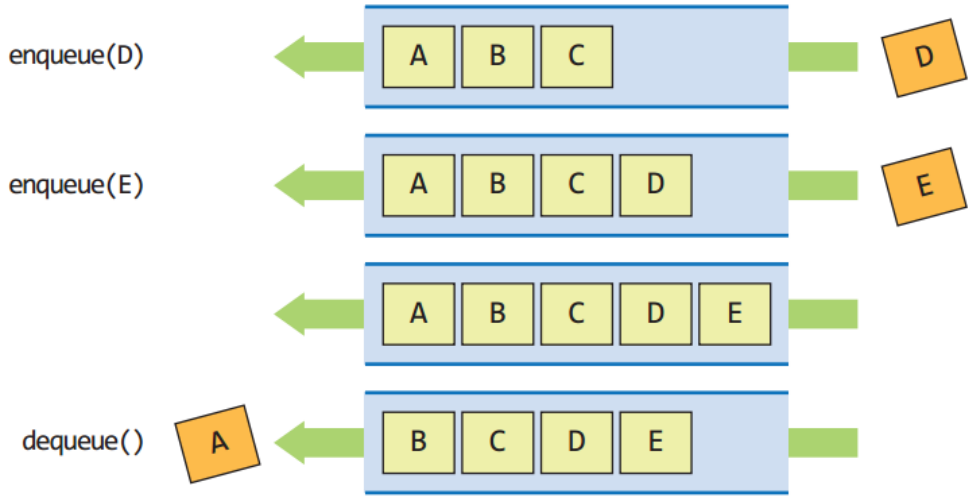
연산

- enqueue(e): 요소 e를 큐의 맨 뒤에 추가한다.
- dequeue(): 큐의 맨 앞에 있는 요소를 꺼내 반환한다.
- isEmpty(): 큐가 비어있으면 True를 아니면 False를 반환한다.
- isFull(): 큐가 가득 차 있으면 True를 아니면 False를 반환한다.
- peek(): 큐의 맨 앞에 있는 요소를 삭제하지 않고 반환한다.

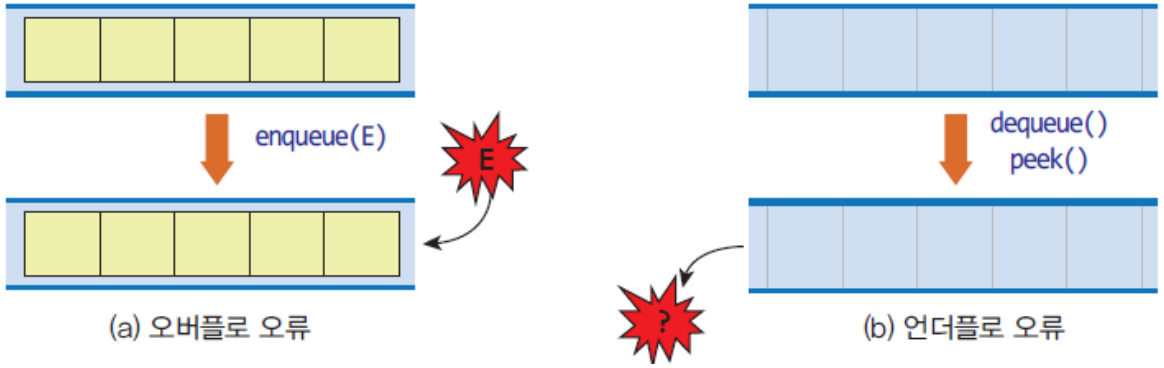
스택과 큐는 중간에 있는 자료에 접근할 수 없음

# 큐의 연산

- 삽입/삭제 연산

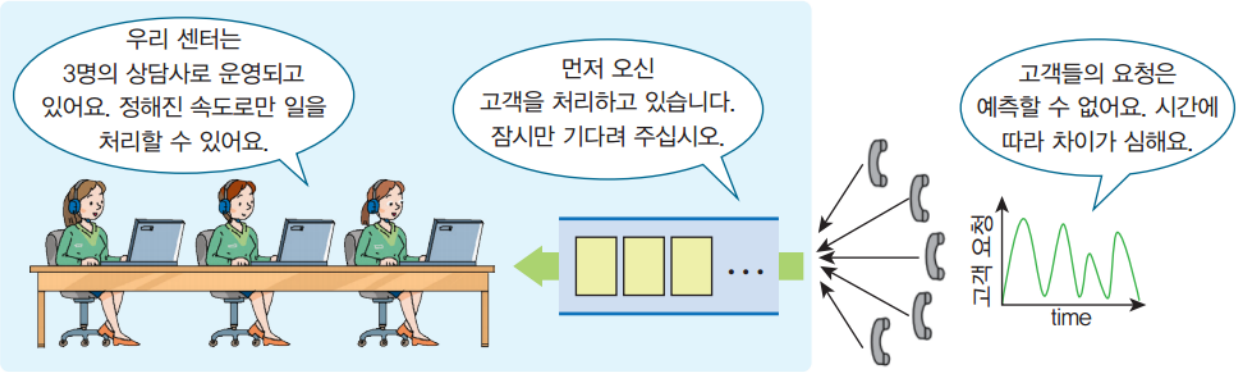



- 두 가지 오류 상황



# 큐의 응용

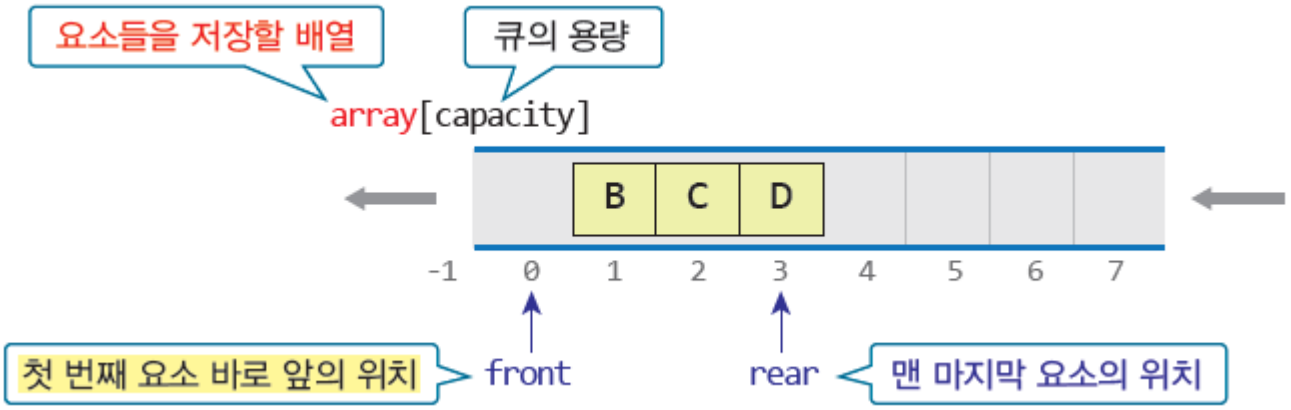
- 예) 서비스센터의 콜 큐




- 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

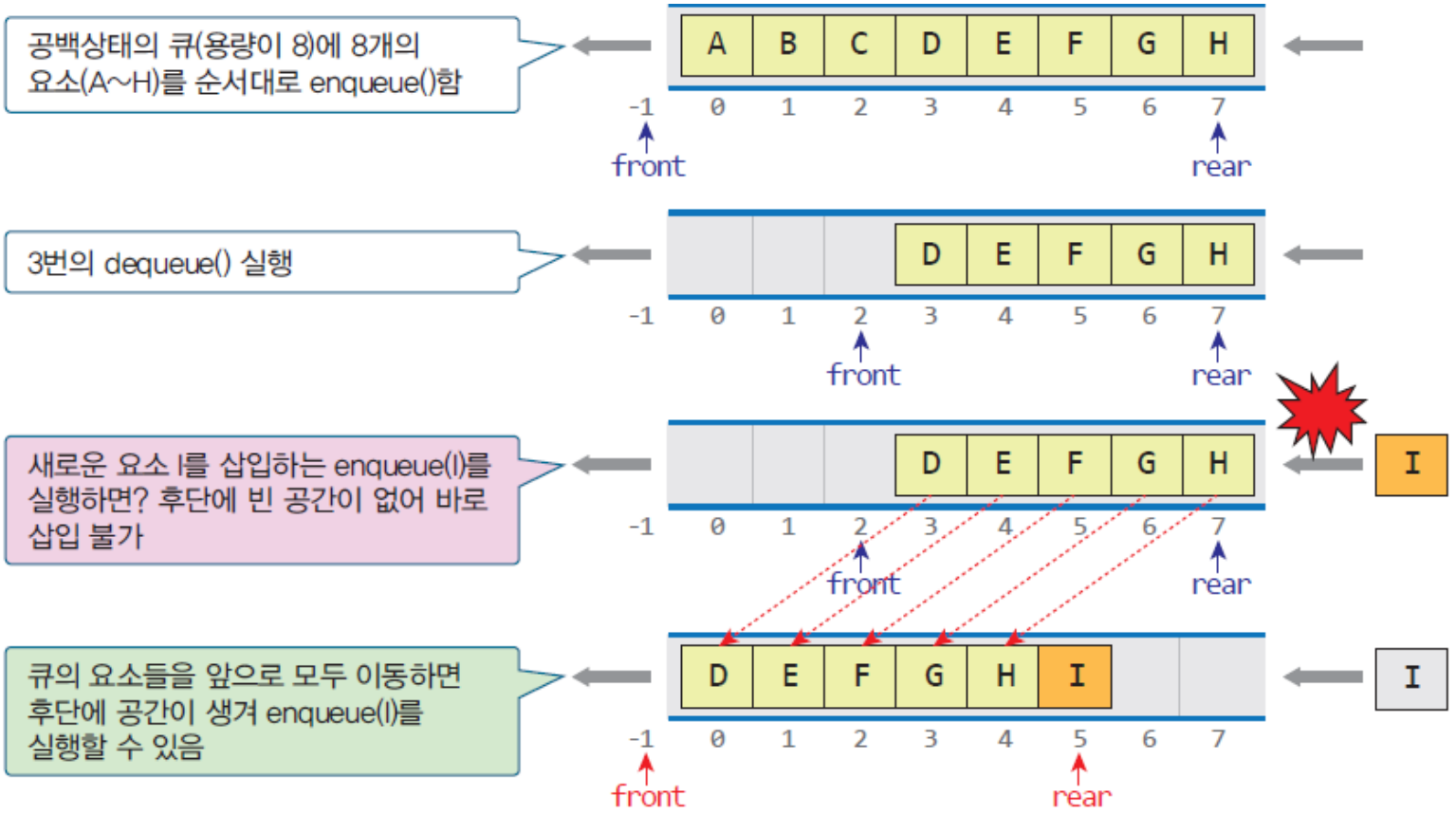
- 배열을 이용한 큐의 구조
  - 용량이 고정된 큐




  - front: 전단 요소 바로 앞의 위치(인덱스)
  - rear: 맨 마지막 요소의 위치(인덱스)

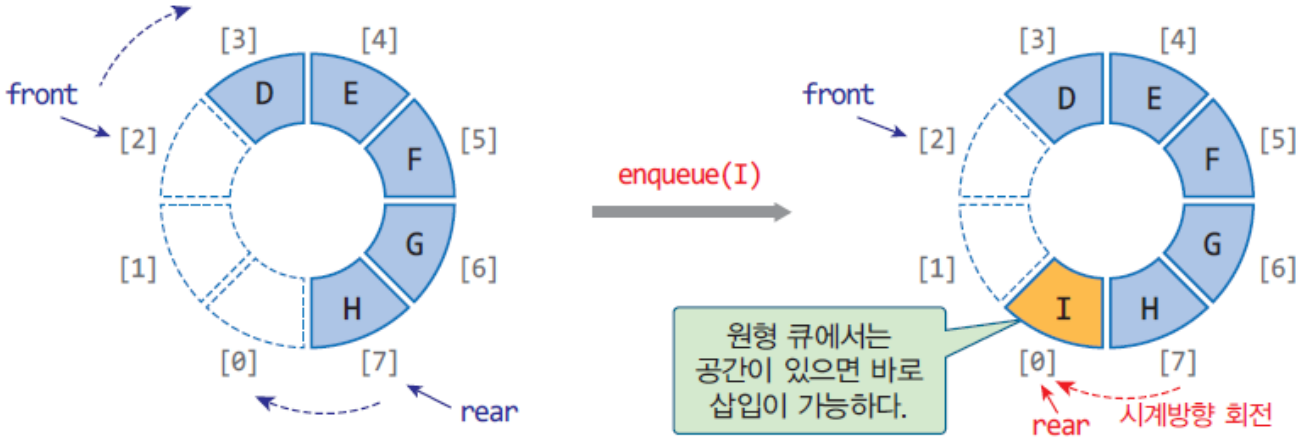
# 선형큐의 문제점

- 많은 이동이 필요한 경우 발생



# 원형 큐가 훨씬 효율적이다.

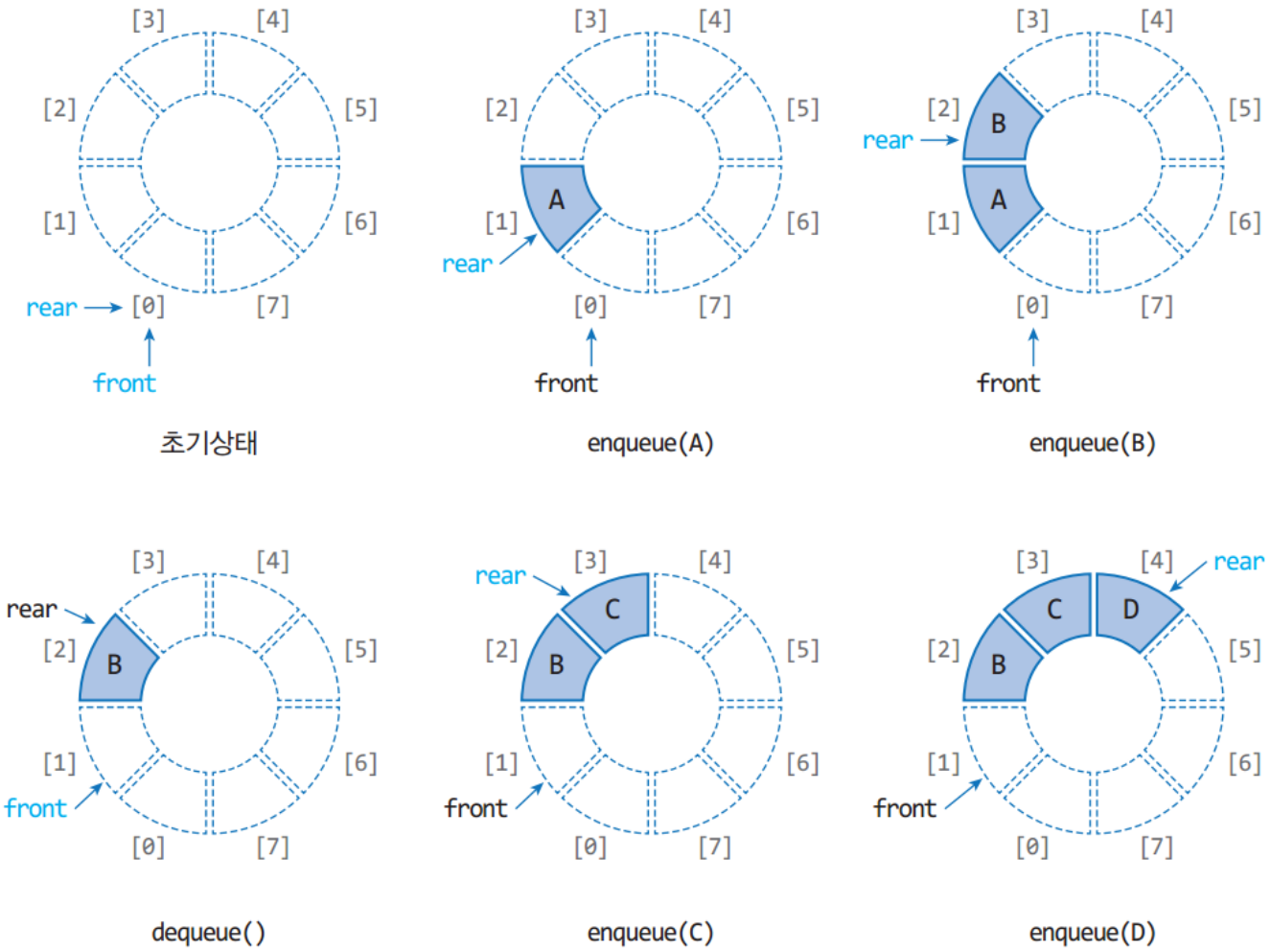
- 원형큐
  - 배열을 원형으로 사용 → 인덱스를 회전시키는 개념
    - front: 첫번째 요소 하나 앞의 인덱스
    - rear: 마지막 요소의 인덱스




  - 인덱스 회전(시계방향) 방법

- 전단 회전 : front ← (front+1) % capacity
- 후단 회전 : rear ← (rear+1) % capacity

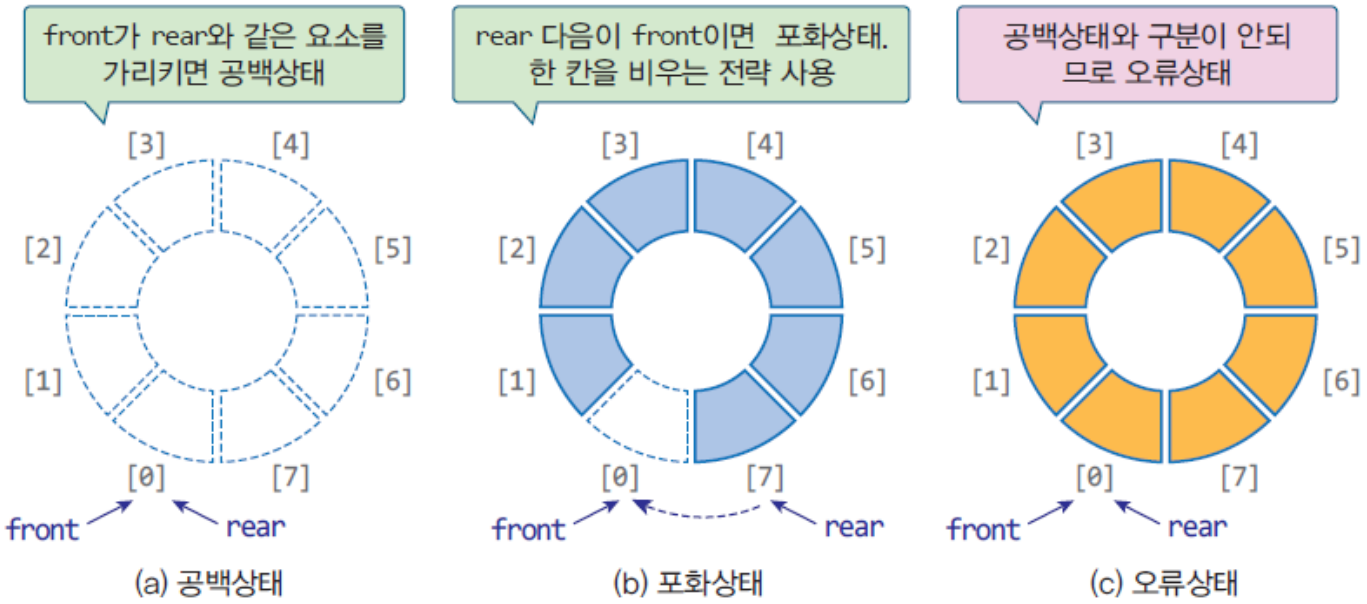
# 원형 큐의 삽입과 삭제 과정



# 원형 큐 클래스(CircularQueue)

- 큐의 생성을 담당하는 생성자 __init_ _()
  - capacity, array, front, rear 초기화

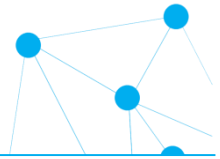
- 공백상태와 포화상태를 검사하는 isEmpty()와 isFull()




front==rear

front == (rear +1)% capacity

# CircularQueue



- 새로운 요소 e를 삽입하는 enqueue(e) 연산
  ① 후단 rear를 먼저 시계방향으로 한 칸 회전
  ② 그 위치에 새로운 요소 e를 복사

```
def enqueue( self, item ):
    if not self.isFull():
        self.rear = (self.rear + 1) % self.capacity
        self.array[self.rear] = item
    else: pass       # 오버플로 예외. 처리 않음
```

- 맨 앞의 요소를 삭제하는 dequeue() 연산
  ① front를 시계방향으로 한 칸 회전
  ② 그 위치의 요소를 반환

```
def dequeue( self ):
    if not self.isEmpty():
        self.front = (self.front + 1) % self.capacity
        return self.array[self.front]
    else: pass       # 언더플로 예외. 처리 않음
```

# CircularQueue

- 맨 앞의 요소를 들여다보는 peek() 연산
  - front의 다음 위치(시계방향으로 회전한)의 요소 반환
- 전체 요소의 수: 코드 5.2
  - (rear - front + capacity) % capacity
- 화면 출력을 위한 요소의 범위: 코드 5.3

# 테스트 프로그램

```
q = CircularQueue(8)
q.enqueue('A')
...
q.enqueue('F')
print('A B C D E F 삽입: ', q)
print('삭제 -->', q.dequeue())
print('삭제 -->', q.dequeue())
print('삭제 -->', q.dequeue())
print('      3번의 삭제: ', q)
q.enqueue('G')
q.enqueue('H')
q.enqueue('I')
print('      G H I 삽입: ', q)
```

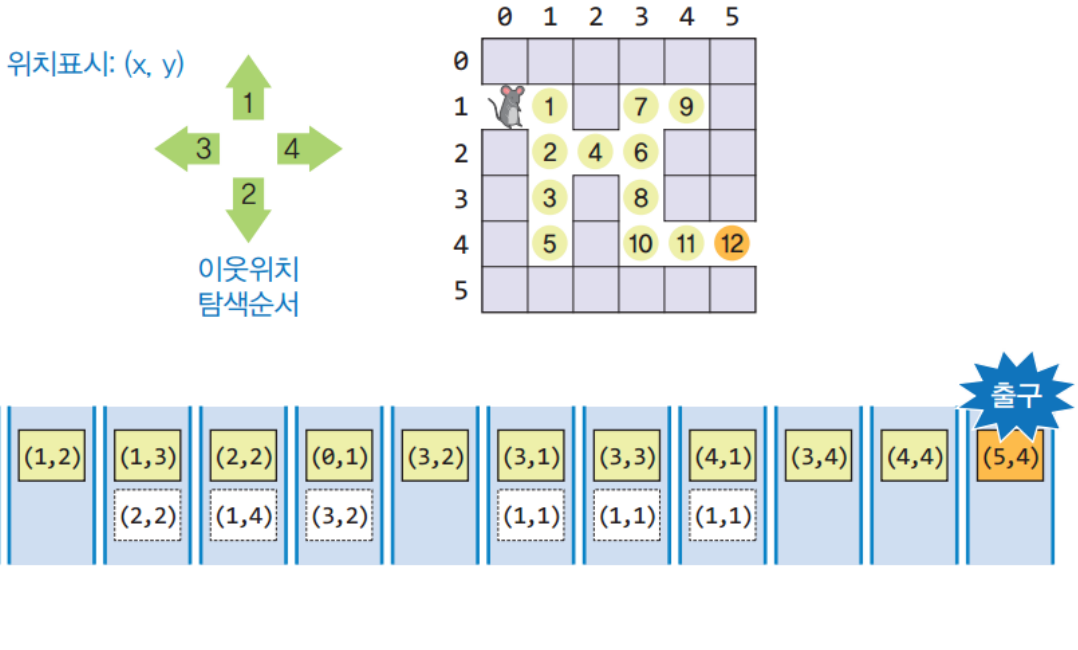
capacity가 8인 원형 큐 객체 생성. 실제 최대 용량은 8-1이 됨

A부터 F까지 7개의 요소를 삽입

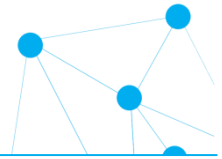
```
C:₩WINDOWS₩system32₩cmd.exe
A B C D E F 삽입:  ['A', 'B', 'C', 'D', 'E', 'F']
삭제 --> A
삭제 --> B
삭제 --> C
      3번의 삭제:  ['D', 'E', 'F']
      G H I 삽입:  ['D', 'E', 'F', 'G', 'H', 'I']
```

# 5.3 큐의 응용: 너비우선탐색

- 큐의 응용 예
  - 이진트리의 레벨 순회 (8장)
  - 기수정렬에서 레코드의 정렬을 위해 사용 (12장)
  - 그래프의 탐색에서 너비우선탐색 (10장)
- 미로 탐색: 너비우선탐색



# 너비우선탐색 알고리즘

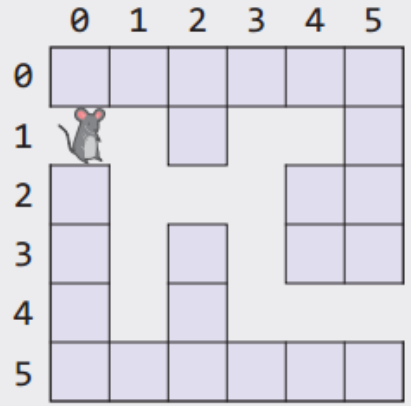


```
def BFS() :                          # 너비우선탐색 함수
    que = CircularQueue()
    que.enqueue((0,1))
    print('BFS: ')                   # 출력을 'BFS'로 변경

    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

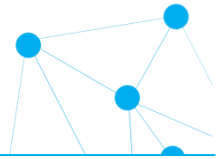
```
map = [  [ '1', '1', '1', '1', '1', '1' ],
         [ 'e', '0', '1', '0', '0', '1' ],
         [ '1', '0', '0', '0', '1', '1' ],
         [ '1', '0', '1', '0', '1', '1' ],
         [ '1', '0', '1', '0', '0', 'x' ],
         [ '1', '1', '1', '1', '1', '1' ]]
MAZE_SIZE = 6
result = BFS()
if result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




너비우선탐색 과정

```
C:₩WINDOWS₩system32₩cmd.exe
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)
```

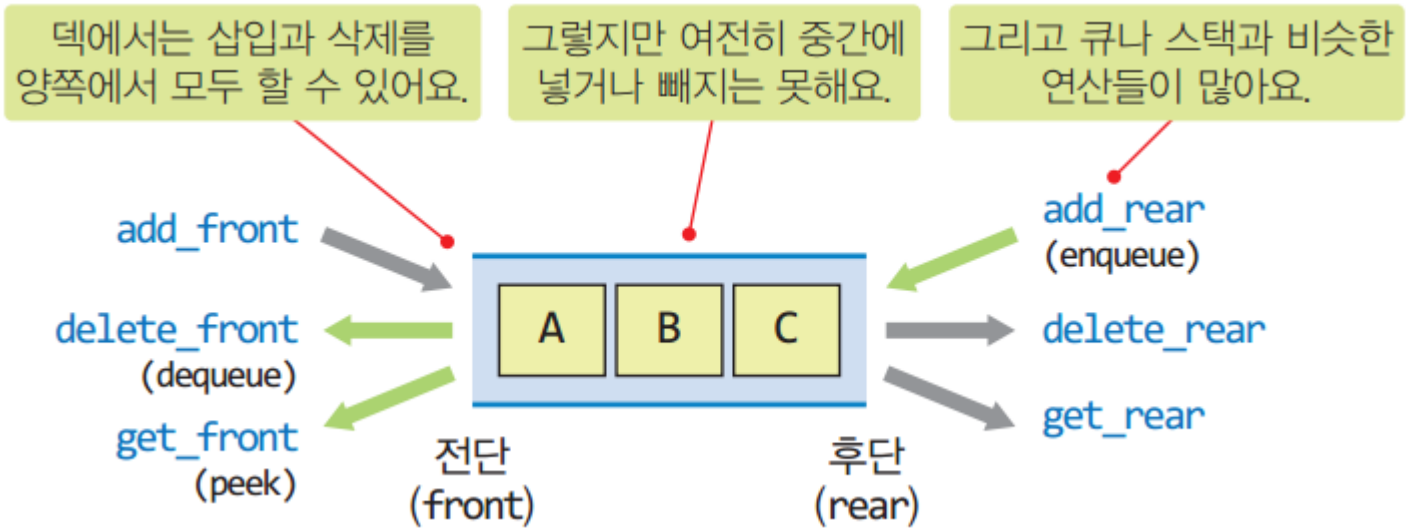
- 함수 이름 변경: 삽입은 put(), 삭제는 get()

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

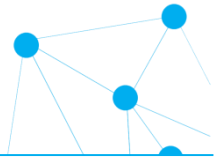
```
C:₩WINDOWS₩system32₩c...
큐의 내용: 1 2 3 4 5 6 7 8 9
```

# 5.4 덱이란?

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



# 덱의 추상 자료형

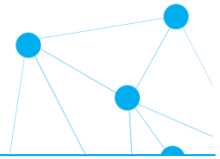


데이터: 전단과 후단을 통한 접근을 허용하는 요소들의 모음

연산

- `isEmpty()`: 덱이 비어 있으면 `True`를 아니면 `False`를 반환한다.
- `isFull()`: 덱이 가득 차 있으면 `True`를 아니면 `False`를 반환한다.
- `addFront(e)`: 맨 앞(전단)에 새로운 요소 `e`를 추가한다.
- `deleteFront()`: 맨 앞(전단)의 요소를 꺼내서 반환한다.
- `getFront()`: 맨 앞(전단)의 요소를 꺼내지 않고 반환한다.
- `addRear(e)`: 맨 뒤(후단)에 새로운 요소 `e`를 추가한다.
- `deleteRear()`: 맨 뒤(후단)의 요소를 꺼내서 반환한다.
- `getRear()`: 맨 뒤(후단)의 요소를 꺼내지 않고 반환한다.

# 덱과 큐의 연산 비교



- 덱은 스택과 큐의 연산들을 모두 가짐
  - addRear, deleteFront, getFront → enqueue, dequeue, peek
  - addRear, deleteRear, getRear → 스택의 push, pop, peek

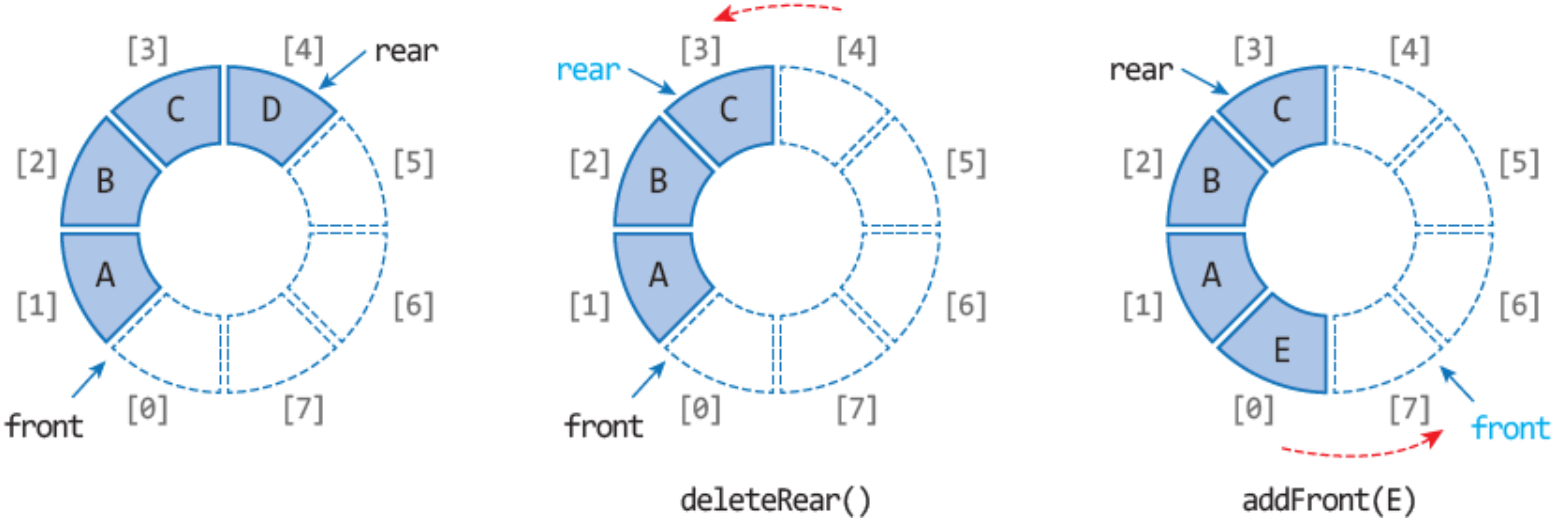
- 덱은 구조상 큐와 더 비슷함
  - front, rear, 원형 회전 등
  - 원형 큐에 추가되어야 하는 연산
    - delete_rear(), add_front(), get_rear()
    - 인덱스를 반시계 방향으로 회전

- 전단 반시계방향 회전 : front ← (front-1+capacity) % capacity
- 후단 반시계방향 회전 : rear ← (rear-1+capacity) % capacity

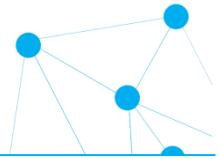
# 원형 큐에서 추가된 연산

- addFront(), deleteRear(), getRear()
  - 반 시계방향 회전 필요

- 전단 반시계방향 회전 : front ← (front-1+capacity) % capacity
- 후단 반시계방향 회전 : rear ← (rear-1+capacity) % capacity



# 5.5 덱의 구현



- 원형 큐를 상속하여 원형 덱 클래스를 구현

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

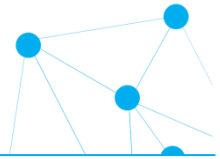
- 덱의 생성자 (상속되지 않음)

```
    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, __str__
- 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
```

# CircularDeque



- 추가할 메소드
  - addFront(), deleteRear(), getRear()

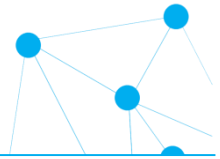
```
# 새로 구현이 필요한 연산들
def addFront( self, item ):
    if not self.isFull():
        self.array[self.front] = item
        self.front = (self.front - 1 + self.capacity) % self.capacity
    else: pass


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

전단 삽입은 덱이 포화상태가 아닌 경우 처리 가능. 현재의 front에 새로운 요소를 저장한 다음, front를 반시계 방향으로 한 칸 회전

후단 삭제는 공백이 아닌 경우 처리가능. 현재의 rear 요소를 저장한 후, rear를 반시계 방향으로 한 칸 회전, 마지막으로 저장했던 요소를 반환함

# 테스트 프로그램



```
dq = CircularDeque()

for i in range(9):
    if i%2==0 : dq.addRear(i)
    else : dq.addFront(i)
print("홀수->전단, 짝수->후단:", dq)

for i in range(2): dq.deleteFront()
for i in range(3): dq.deleteRear()
print(" 전단삭제x2 후단삭제x3:", dq)

for i in range(9,14): dq.addFront(i)
print("   전단삽입 9,10,...13:", dq)
```

짝수는 후단 삽입
홀수는 전단 삽입

전단 삭제 2번
후단 삭제 3번

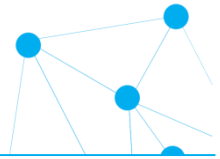
홀수는 전단으로 삽입

짝수는 후단으로 삽입

```
C:\WINDOWS\system32\cmd.exe
홀수->전단, 짝수->후단: [7, 5, 3, 1, 0, 2, 4, 6, 8]
 전단삭제x2 후단삭제x3: [3, 1, 0, 2]
   전단삽입 9,10,...13: [13, 12, 11, 10, 9, 3, 1, 0, 2]
```

9, 10, 11, 12, 13을 순서적으로 전단에 삽입

# 5.6 우선순위 큐

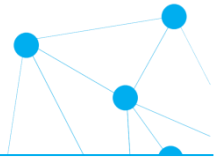


- 실생활에서의 우선순위
  - 도로에서의 자동차 우선순위

- 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 큐의 추상 자료형

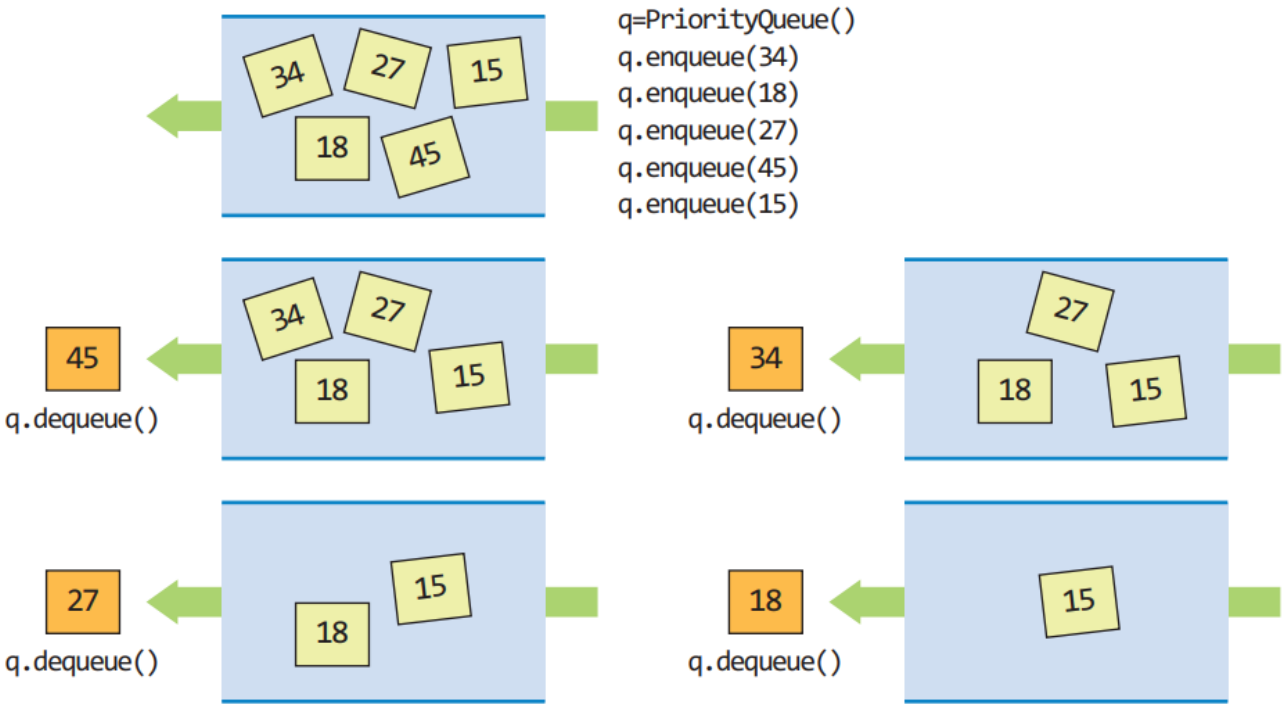


데이터: 우선순위를 가진 요소들의 모음

연산

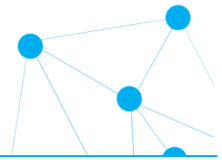
- `isEmpty()`: 우선순위큐가 비어있으면 `True`를 아니면 `False`를 반환한다.
- `isFull()`: 우선순위큐가 가득 차 있으면 `True`를 아니면 `False`를 반환한다.
- `enqueue(e)`: 우선순위를 가진 요소 `e`를 삽입한다.
- `dequeue()`: 가장 우선순위가 높은 요소를 꺼내서 반환한다.
- `peek()`: 가장 우선순위가 높은 요소를 삭제하지 않고 반환한다.

# 우선순위 큐의 삽입과 삭제 연산




- 구현 방법
  - 배열 구조, 연결된 구조, 힙 트리 등

# 우선순위 큐의 구현



- 정렬되지 않은 배열을 이용한 우선순위 큐
  - 용량이 고정된 우선순위 큐
  - 코드 5.10

```
class PriorityQueue :
    def __init__( self, capacity = 10 ) :
        self.capacity = capacity
        self.array = [None] * capacity
        self.size = 0          # 요소의 수
```

```
    def enqueue( self, e ):
        if not self.isFull():
            self.array[self.size] = e
            self.size += 1
```

삽입: 맨 뒤에 추가

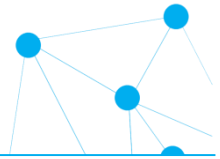
```
    def dequeue( self ):
        highest = self.findMaxIndex()
        if highest != -1 :
            self.size -= 1
            self.array[highest], self.array[self.size] = \
                self.array[self.size], self.array[highest]
            return self.array[self.size]
```

삭제: 최고 우선순위 요소 찾기
→ 맨 뒤 요소와 교체 → 삭제

```
    def peek( self ):
        highest = self.findMaxIndex()
        if highest != -1 :
            return self.array[height]
```

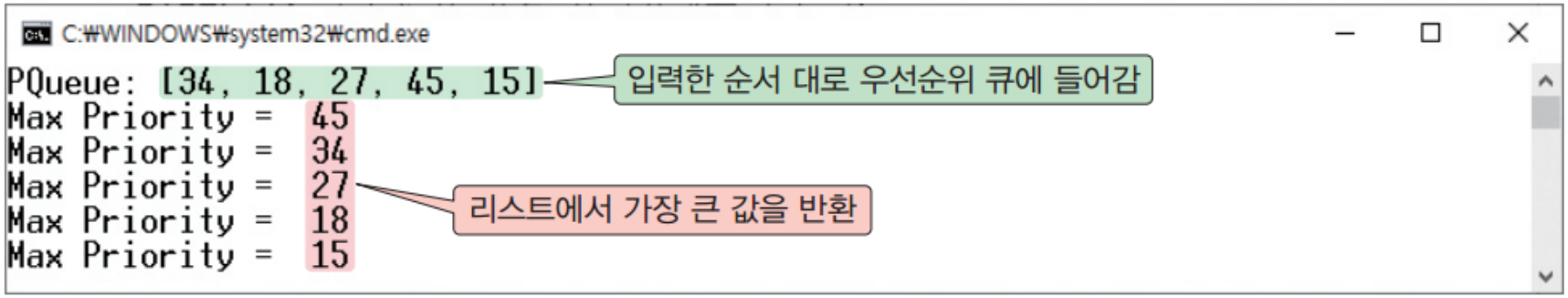
탐색: 최고 우선순위 요소 찾기
→ 반환

# 테스트 프로그램

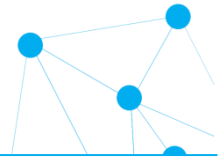


```
q = PriorityQueue()
q.enqueue( 34 )
q.enqueue( 18 )
q.enqueue( 27 )
q.enqueue( 45 )
q.enqueue( 15 )

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



# 시간 복잡도



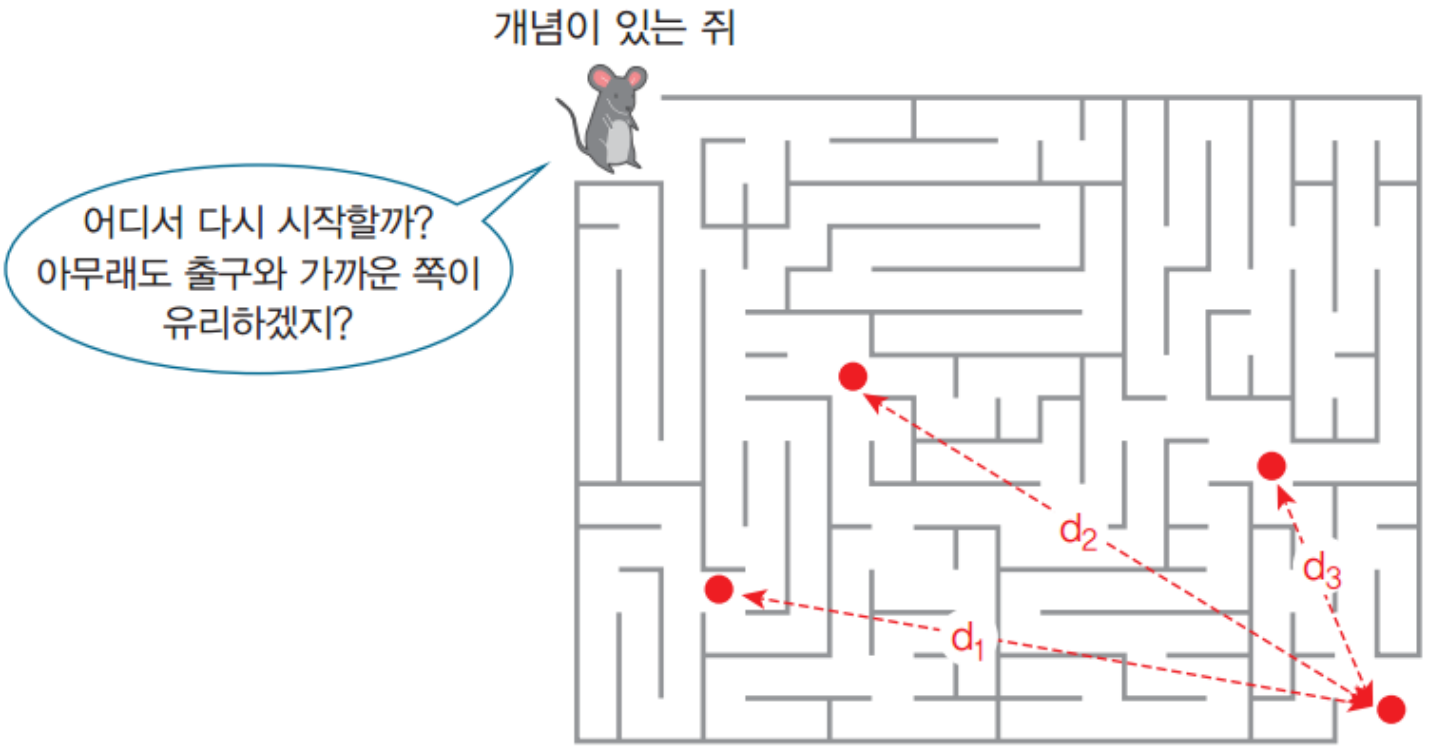
- 정렬되지 않은 리스트 사용
  - enqueue(): 대부분의 경우 $O(1)$
  - findMaxIndex(): $O(n)$
  - dequeue(), peek() : $O(n)$

- 정렬된 리스트 사용
  - enqueue(): $O(n)$
  - dequeue(), peek() : $O(1)$

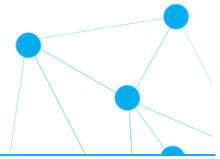
- 힙 트리(8장)
  - enqueue(), dequeue(): $O(logn)$
  - peek() : $O(1)$

# 5.7 전략적인 미로 탐색

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



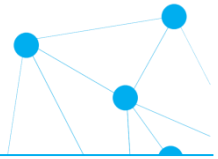
# 최대 우선순위 항목 선택



- 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
```

# 전략적 미로 탐색 알고리즘



```
def MySmartSearch() :                          # 최소거리 전략의 미로탐색
    q = PriorityQueue()                        # 우선순위 큐 객체 생성
    q.enqueue((0,1,-dist(0,1)))                # 튜플에 거리정보 추가
    print('PQueue: ')

    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


- 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

# 감사합니다!