```
# 큐는 선입선출(FIFO)의 자료구조이다.
# 후단(rear)에서 삽입을 하며 전단(front)에서 삭제를 하는 구조이다.

# 큐의 ADT(추상자료형)
# Queue() : 비어있으면 True를 아니면 False를 반환한다.
# isEmpty() : 큐가 비어있었으면 T를 아니면 F를 반환한다.
# enqueue(x) : 항목 x를 큐의 맨 뒤에 추가한다.
# dequeue() : 규의 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 응용
# 서비스 센터의 콜 큐
# 인쇄작업에 버퍼링 큐
# 실시간 비디오 스트리밍에서의 버퍼링 큐
# 공항과 은행등에서의 대기열

# 선형큐는 비효율적이다.
# 삽입연산 : O(1)
# 삭제연산 : O(n) -> 맨 앞에 항목을 꺼내서 반환해야 하므로

# 원형 큐 : 배열을 원형으로 사용. 실제 원형이 생긴 것이 아니라 선형 리스트를 원형 큐처럼 사용하는 것
# 선형 큐보단 월씬 효율적이다.

# 시계방향 회전 방법 : front = (front+1)%MAX_QSIZE or rear = (rear+1)%MAX_QSIZE
# 반시계방향 회전 방법 : front = (front-1+MAX_QSIZE)%MAX_QSIZE or rear = (rear-1+MAX_QSIZE)%MAX_QSIZE

# 공백상태 : front == rear
# 포화상태 : front % MAX_QSIZE == (rear+1) % MAX_QSIZE




# 파이썬 리스트를 사용하여 원형 큐 클래스 구현
class CircularQueue :
    def __init__(self):
        self.front = 0
        self.rear = 0
        self.MAX_QSIZE = 20 # 디폴트 값으로 20 , 전역으로 선언하면 알고리즘 상 좋지않다.
        self.items = [None]*self.MAX_QSIZE # 항목 저장용 리스트 [None, None, ...]

    def isEmpty( self ): # 공백상태
        return self.front == self.rear

    def isFull( self ): # 포화상태
        return self.front == (self.rear + 1)%self.MAX_QSIZE

    def clear( self ): # 공백으로 만들기
        self.front = self.rear

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

```
    def dequeue( self ): # 전위에 값을 뽑고 삭제
        if not self.isEmpty():
            self.front = (self.front+1)%self.MAX_QSIZE
            return self.items[self.front] # 값이 자동으로 삭제된다. front가 변경될때 인덱스를 잃어버리며 링크가 잘리기 때문에

    def peek( self ): # 전위의 값을 복사하여 반환
        if not self.isEmpty():
            return self.items[(self.front + 1)%self.MAX_QSIZE]

    def size( self ): # 나머지의 성질을 이용하여 사이즈 계산
        return (self.rear - self.front + self.MAX_QSIZE) % self.MAX_QSIZE

    def display( self ): # front, rear, queue 순서로 출력
        out = []
        if self.front < self.rear :
            out = self.items[self.front+1:self.rear+1]
        else:
            out = self.items[self.front+1:self.MAX_QSIZE] + self.items[0:self.rear+1]
        print("[f=%s, r=%s] ==> "%(self.front, self.rear), out)

# 본문

if __name__ == "__main__":
    q = CircularQueue()
    q.MAX_QSIZE = 10 # 원형 큐의 크기 선언

    for i in range(8):
        q.enqueue(i) # 0, 1, ... , 7 대입

    q.display()

    for i in range(5):
        q.dequeue() # 전위부터 5개 반환하고 삭제

    q.display()

    for i in range(8,14):
        q.enqueue(i) # 8, 9, ... , 13 대입

    q.display()
'''
[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

# 미로탐색 : 너비우선탐색

# 큐를 이용한 미로 탐색 코드
from CircularQuenue import CircularQueue

def isValidPos(x, y): # 지금의 좌표가 유의한지 검사하는 함수
    if x<0 or y<0 or x >= MAZE_SIZE or y >= MAZE_SIZE: # 범위 밖이면 False
        return False
    else:
        return map[y][x] == '0' or map[y][x] == 'x' # 0, x만 참, 1은 거짓

def BFS(): # 너비 우선 탐색 함수
    que = CircularQueue() # 원형큐(리스트) 생성
    que.enqueue((0,1)) # 시작위치 삽입, (0,1)은 튜플
    print('BFS: ')

    while not que.isEmpty(): # 비어있지 않으면
        here = que.dequeue() # 맨위에 좌표 대입 후 삭제
        x, y = here # 맨 위에 있던 좌표를 각각 대입
        print(here, end="->")
        # 컴퓨터는 입력한 값이 열 중심으로 입력되기 때문에 y=-x 대칭으로 인식하여
        # map[x][y]가 아닌 map[y][x]로 해야 우리가 보는 모습과 같게 위치시킬수 있다.
        if (map[y][x] == 'x'): # 출구면 탐색 성공, T반환
            return True
        else:
            map[y][x] = '.' # 현재 위치를 지나왔다고 '.'표시, 그래야 isValidPos에서 범위 밖으로 판단하여 역주행을 안한다.
            # 컴터는 열을 기준으로 값을 대입하므로 눈에 보이는 지도와 인댁스가 y=-x 대칭이다.
            # 따라서 [x][y] 가 아닌 [y][x]로 생성
            # 4방향의 이웃을 검사해 갈 수 있으면 스택에 상하좌우를 우선순위로 삽입
            if isValidPos(x, y-1): # 움직인 점이 T이면(0 또는 x) 실행
                que.enqueue((x, y-1)) # 상, 만약 [x][y]이였다면, (x-1, y)
            if isValidPos(x, y+1): # 움직인 점이 T이면(0 또는 x) 실행
                que.enqueue((x, y+1)) # 하, 만약 [x][y]이였다면, (x+1, y)
            if isValidPos(x-1, y): # 움직인 점이 T이면(0 또는 x) 실행
                que.enqueue((x-1, y)) # 좌, 만약 [x][y]이였다면, (x, y-1)
            if isValidPos(x+1, y): # 움직인 점이 T이면(0 또는 x) 실행
                que.enqueue((x+1, y)) # 우, 만약 [x][y]이였다면, (x, y+1)
    return False

# 본문
if __name__ == "__main__":

    MAZE_SIZE = 6 # 정사각형의 미로의 사이즈 지정
    # 실제로 컴터가 인식하는 방향은 y=-x 대칭 모양이므로 x,y의 순서를 바꿔서 행렬을 계산한다.
    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'] ]

    result = BFS() # 결과 입력
    print('--> 미로탐색 성공') if result else print('--> 미로탐색 실패')
```

```
    # # 결과 출력
    # if result:
    #     print('--> 미로탐색 성공')
    # else:
    #     print('--> 미로탐색 실패')

'''
(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
'''
```

```
# 큐(Queue) 와 스택(stack)은 파이썬에서 모듈을 제공
# 스택(stack) -> class lifoQueue
# 큐(Queue) -> class queue

# 파이썬 모듈로 큐 연산하기
import queue

Q = queue.Queue(maxsize=20) # queue클래스에 Queue함수, 인수로 크기 입력(최대 20)

for v in range(1, 10):
    Q.put(v) # enqueue가 put으로 재정의

print("큐의 내용: ", end="")

for _ in range (1,10):
    print(Q.get(), end=" ") # dequeue는 get으로 재정의

print()

'''
큐의 내용: 1 2 3 4 5 6 7 8 9
'''
```

```
# 덱은 스택이나 큐보다는 입출력이 자유로운 자료구조이다.
# 덱(deque : double ended queue) : 전단과 후단에서 모두 삽입과 출력이 가능

# 덱의 ADT(추상자료형)
# Deque() : 비어 있는 새로운 덱을 만든다.
# isEmpty() : 덱이 비어있으면 T, 아니면 F를 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 T, 아니면 F를 반환한다.
# size() : 덱의 모든 항목들의 개수를 반환한다.
# clear() : 덱을 공백 상태로 만든다.

# 덱의 연산 : 스택과 큐의 연산과 같다.
#              add            delete          get
# Front    push(스택)    dequeue(큐)     peek(큐)
# Rear    enqueue(큐)    pop(츠택)       peek(스택)

# 시계방향 회전 방법 : front = (front+1)%MAX_QSIZE or rear = (rear+1)%MAX_QSIZE
# 반시계방향 회전 방법 : front = (front-1+MAX_QSIZE)%MAX_QSIZE or rear = (rear-1+MAX_QSIZE)%MAX_QSIZE

# 원형 덱 구현
from CircularQuenue import CircularQueue

class CircularDepue(CircularQueue): # CircularQueue에서 상속
    def __init__( self ):
        super().__init__() # 부모 클래스에서 생성자 호출

    # 전위 계산
    def addFront( self, item): # 값 추가
        if not self.isFull():
            self.items[self.front] = item
            self.front = (self.front - 1 + self.MAX_QSIZE) % self.MAX_QSIZE # 반시계 방향으로 회전

    def deleteFront( self ):
        return self.dequeue() # 부모 클래스(큐)에서 함수 호출, 시계 방향으로 회전

    def getFront( self ):
        return self.peek() # 부모 클래스(큐)에서 함수 호출

    # 후위 계산

    def addRear( self, item ):
        self.enqueue( item ) # 부모 클래스(큐)에서 함수 호출, 시계 방향으로 회전

    def deleteRear( self ): # 값을 리턴하고 삭제
        if not self.isEmpty():
            item = self.items[self.rear]; # 인덱스를 바꾸면서 값을 삭제
            self.rear = (self.rear - 1 + self.MAX_QSIZE) % self.MAX_QSIZE # 반시계 방향으로 회전
            return item

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

```
# 본문
if __name__ == "__main__":

    dq = CircularDepue() # 덱 선언
    dq.MAX_QSIZE = 10

    for i in range(9):
        if i%2 == 0: # 짝수는 뒤에서 입력
            dq.addRear(i)
        else: # 홀수는 앞에서 입력
            dq.addFront(i)

    dq.display()

    for i in range(2): # 전단에서 삭제
        dq.deleteFront()

    for i in range(3): # 후단에서 삭제
        dq.deleteRear()

    dq.display()

    for i in range(9,14):
        dq.addFront(i)

    dq.display()
'''
[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]
'''
```

```
# 우선순위 큐

# 응용분야
# 시뮬레이션, 네트워크 트래픽 제어, OS의 작업 스케쥴링 등

# 우선순의 큐의 ADT(추상자료형)
# PriorityQueue() : 비어있는 우선순위 큐를 만든다.
# isEmpty() : 우선순위 큐가 공백인지 검사한다.
# enqueue(e) :우선운위를 가진 항목 e를 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 공백 상태로 만든다.

# 일상의 흔한 데이터는 정렬이 되어있지 않다.
# 그리고 그중에서 우선순위를 가지고 출력을 해야하는 경우도 있다. 그래서 이를 구현 해본다.
# 우선 순위를 판별하는 함수를 새롭게 만들어서 사용한다.

# 정렬되지 않은 배열을 가지고 우선순위 큐 생성

# 정렬되지 않은 리스트 사용
# enqueue() : O(1)
# findMaxIndex() : O(n)
# dequeue() : O(n)
# peek() : O(n)

# 정렬된 리스크 사용
# enqueue() : O(n)
# dequeue() : O(1)
# peek() : O(1)

class PriorityQueue:
    def __init__( self ):
        self.items = [] # 항목을 저장하기 위한 리스트

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

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

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

    def enqueue( self, item ): # 추가
        # 순서는 다시 정렬할 것이므로 그냥 대입한다. 큐를 리스트로 정의 했으므로 list.append(item)을 사용
        self.items.append( item )

    def dequeue( self ): # 우선순위가 가장 높은 항목을 출력 후 삭제
        highest = self.findMaxIndex()
        if highest is not None:
            return self.items.pop(highest)
```

```
    def peek(self): # 우선순위가 가장 높은 항목을 출력
        highest = self.findMaxIndex()
        if highest is not None: # 값이 없지 않으면
            return self.items[highest] # List.pop(인덱스) : 인덱스 값 출력후 삭제

    def findMaxIndex( self ): # 우선순위 판단 함수
        if self.isEmpty():
            return None
        else:
            highest = 0
            for i in range( 1, self.size()):
                if self.items[i] > self.items[highest]: # 가장큰 값을 가지는 인덱스 반환
                    highest = i
        return highest

# 본문
if __name__ == "__main__":
    q = PriorityQueue() # 우선순위 큐 객체 생성, q.items = [] 생성
    q.enqueue(34)
    q.enqueue(18)
    q.enqueue(27)
    q.enqueue(45)
    q.enqueue(15) # 값 대입

    print("PQueue: ", q.items) # 정렬을 하지않은, 입력한 순서 그대로 출력

    while not q.isEmpty():
        print("Max Priority = ", q.dequeue(),", PQueue : ",q.items) # 배열 중 가장 큰 값을 반환

'''
PQueue:  [34, 18, 27, 45, 15]
Max Priority =  45 , PQueue :  [34, 18, 27, 15]
Max Priority =  34 , PQueue :  [18, 27, 15]
Max Priority =  27 , PQueue :  [18, 15]
Max Priority =  18 , PQueue :  [15]
Max Priority =  15 , PQueue :  []
'''
```

```
# 전략적 미로 탐색(알고리즘)

# 도착지점을 알 경우 점과 점사이의 거리를 사용하여 갈림길 중 좀더 가까운 쪽부터 먼저 계산하는 알고리즘
from PriorityQueue import PriorityQueue
from math import *

def dist(a, b): # 거리계산 함수
    return sqrt((5-a)*(5-a) + (4-b)*(4-b))*(-1)

def isValidPos(x, y):
    if x<0 or y<0 or x >= MAZE_SIZE or y >= MAZE_SIZE:
        return False
    else:
        return map[y][x] == '0' or map[y][x] == 'x'

def findMaxIndex( self ): # 부모의 함수를 재정의
    if self.isEmpty():
        return None
    else:
        highest = 0
        for i in range( 1, self.size()):
            if self.items[i][2] > self.items[highest][2]:
                highest = i
    return highest

def MySmartSearch() :
    q = PriorityQueue() # q.items = [] 생성
    q.enqueue((0,1,dist(0,1))) # 순서가 필요없으므로 차례대로 대입, dist는 도착점까지의 거리를 반환하는 함수이다.
    print('PQueue: ')

    while not q.isEmpty(): # 비어있지 않으면
        here = q.dequeue() # 재정의된 findMaxIndex를 통해 가장 위에 있는 좌표와 거리의 튜플를 대입 후 삭제
        x, y, _ = here # -를 사용하여 dist 값은 버린다.
        print(here[0:2], end="->") # 거리를 제외한 'X,Y' 만 출력하기 위해 0:2로 출력
        if (map[y][x] == 'x'):
            return True
        else:
            map[y][x] = '.' # 지나간 점 '.'으로 표기
            # 컴터는 열을 기준으로 값을 대입하므로 눈에 보이는 지도와 인댁스가 y=-x 대칭이다.
            # 따라서 [x][y] 가 아닌 [y][x]로 생성
            if isValidPos(x, y-1): # 움직인 점이 T이면(0 또는 x) 실행
                q.enqueue((x, y-1, dist(x, y-1))) # 상, 만약 [x][y]이였다면, (x-1, y)
            if isValidPos(x, y+1): # 움직인 점이 T이면(0 또는 x) 실행
                q.enqueue((x, y+1, dist(x, y+1))) # 하, 만약 [x][y]이였다면, (x+1, y)
            if isValidPos(x-1, y): # 움직인 점이 T이면(0 또는 x) 실행
                q.enqueue((x-1, y, dist(x-1, y))) # 좌, 만약 [x][y]이였다면, (x, y-1)
            if isValidPos(x+1, y): # 움직인 점이 T이면(0 또는 x) 실행
                q.enqueue((x+1, y, dist(x+1, y))) # 우, 만약 [x][y]이였다면, (x, y+1)
        # 갈림길에서 막히는 부분은 멈추고, 길이 있는 부분은 계속 갱신되는 방식으로 튜플이 연속해서 출력
        print('우선순위큐: ', q.items)
    return False
```

```
# 본문
if __name__ == "__main__":

    MAZE_SIZE = 6
    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'] ]

    # 경로 탐색
    result = MySmartSearch()

    # 결과 출력
    if result:
        print('--> 미로탐색 성공')
    else:
        print('--> 미로탐색 실패')
'''
PQueue:
(0, 1)->우선순위큐:  [(1, 1, -5.0)]
(1, 1)->우선순위큐:  [(1, 2, -4.47213595499958)]
(1, 2)->우선순위큐:  [(1, 3, -4.123105625617661), (2, 2, -3.605551275463989)]
(2, 2)->우선순위큐:  [(1, 3, -4.123105625617661), (3, 2, -2.8284271247461903)]
(3, 2)->우선순위큐:  [(1, 3, -4.123105625617661), (3, 1, -3.605551275463989), (3, 3, -2.23606797749979)]
(3, 3)->우선순위큐:  [(1, 3, -4.123105625617661), (3, 1, -3.605551275463989), (3, 4, -2.0)]
(3, 4)->우선순위큐:  [(1, 3, -4.123105625617661), (3, 1, -3.605551275463989), (4, 4, -1.0)]
(4, 4)->우선순위큐:  [(1, 3, -4.123105625617661), (3, 1, -3.605551275463989), (5, 4, -0.0)]
(5, 4)->--> 미로탐색 성공
'''
```

```
# 덱을 사용하여 회문인지 아닌지 확인하는 코드

# 함수 구현
from CircularQuenue import CircularQueue
from CircularDeque import CircularDepue

def palindrome(list):
    list = list.lower() # 대문자로 입력을 할 것을 대비하여 소문자로 변경
    # strs.MAX_QSIZE = len(list) + 1, 왜 안되는 것인가? input 문장 : madam, i'm Adam
    # 현재는 CircularQueue의 MAX_QSIZE값을 10 -> 20으로 하여 (11)자의 문장을 비교할 수 있다.
    for s in list: # 콤마, 마침표, 괄호, 특수기호, 숫자 등이 입력 되는 것을 막기 위해 아스키코드 값을 사용
        if s >="a" and s <="z":
            strs.addRear(s) # 후미대입 : 후미에 값을 계속 대입한다.

    num = 1 # 참일 경우 1을 유지
    strs.display()
    for i in range(strs.size() // 2): # 자리수를 2로 나눈 몫만큼 반복한다. 양쪽에서 비교하고 삭제하므로
        if strs.deleteFront() != strs.deleteRear(): # 양쪽에 값을 비교하고 삭제하며 회문을 검사
            num = 0 # 거짓일 경우 0으로 변경
    return num # 참:1 / 거짓:0 을 반환

def check_palindrome(num):
    if num == 1:
        print("회문입니다.")
    else:
        print("회문이 아닙니다.")

# 본문
if __name__ == "__main__":
    strs = CircularDepue() # 덱 객체 생성
    list = input("회문인지 확인할 출력할 문자열을 입력하세요 :  ")
    num = palindrome(list) # 사용자 지정 함수를 이용하여 회문 검사
    check_palindrome(num) # 결과 출력

'''
회문인지 확인할 출력할 문자열을 입력하세요 :  asDFg, f:. dS..  A
[f=0, r=9] ==>  ['a', 's', 'd', 'f', 'g', 'f', 'd', 's', 'a']
회문입니다.

회문인지 확인할 출력할 문자열을 입력하세요 :  madam, i'm Adam
[f=0, r=11] ==>  ['m', 'a', 'd', 'a', 'm', 'i', 'm', 'a', 'd', 'a', 'm']
회문입니다.

회문인지 확인할 출력할 문자열을 입력하세요 :  asD, 'gS.a
[f=0, r=6] ==>  ['a', 's', 'd', 'g', 's', 'a']
회문이 아닙니다.
'''
```