2021-2 자료구조및실습 실습과제 04

학번	2020136129	이름	최수연

1) 각 문제에 대한 분석과 및 해결 방법

1. 자신만의 미로를 작성하고 다음의 세 가지 방법으로 미로찾기를 실행하는 프로그램을 작성하라.

(1) 미로의 크기는 10x10 이상이어야 한다.
(2) 스택을 이용한 미로탐색 코드를 작성하라. 이때 스택은 4장 131쪽의 스택 클래스를 이용하라.
(3) 큐를 이용한 미로탐색 코드를 작성하라. 이때 큐는 5.2절과 같이 원형큐 클래스를 작성하고 이를 이용하라. 교재 5.2~5.3절 참조.
(4) 정렬된 배열을 이용한 우선순위 큐 클래스를 구현하고, 이를 이용해 전략적 미로탐색 코드를 작성하라. 이때, 전략은 교재 5.7절에서와 같이 출구까지의 거리를 사용할 수도 있고, 자신만의 전략을 세우고 이를 이용해도 좋다. 사용한 전략이 반드시 다른 방법보다 빨리 출구를 찾을 필요는 없으며, "전략"만 의미가 있으면 된다.

[문제분석 및 해결방법]

(1) 미로의 크기를 배열로 10X10 크기로 만들어 시작을 (0, 5), 끝을 (9, 3)으로 하는 미로찾기를 만들었다.

(2) 기존 미로 탐색 파일 외 스택 클래스만 구현할 파일을 따로 생성하여 해당 파일에 스택 클래스를 생성한다. 교재 4장 131쪽에 있는 스택 클래스를 사용한다. 미로 탐색 파일에서는 스택 파일에서 스택 클래스를 사용하기 위한 모듈을 가져오고, 교재 148~151쪽의 코드를 참고하여 스택을 이용한 깊이 우선 미로 탐색 코드를 구현한다.

(3) 기존 미로 탐색 파일 외 큐 클래스를 구현한 파일을 따로 생성하여 해당 파일에 원형 큐를 클래스로 구현한다. 이때 교재 166~167쪽에 있는 원형 큐 클래스를 사용한다. 미로 탐색 파일에서는 큐 파일에서 원형 큐 스택 클래스를 사용하기 위해 모듈을 가져오고, 교재 170쪽의 코드를 참고하여 큐를 이용한 너비 우선 미로 탐색 코드를 구현한다.

(4) 정렬된 배열을 이용한 우선순위 큐 클래스는 위 (3)에서 생성한 큐 파일에 우선순위 큐 클래스를 새로 구현한다. 전략은 교재 5.7절과 같이 출구까지의 거리를 계산하는 함수를 사용하여 최소거리 전략의 미로 탐색을 하였다. 교재 180~181쪽의 정렬되지 않은 배열을 이용한 우선순위 큐의 구현과 달리 정렬된 배열을 이용한 우선순위 큐는 삽입 연산에서 입력한 순서대로 받아 해당 배열을 확인하여 오름차순에 따라 넣을 위치를 찾는 코드를 구현한다. 이렇게 하면 거리를 음수로 표현하고 있으므로 가장 큰 값을 반환하면 된다. 그러므로 삭제 연산과 peek 연산의 경우 마지막 인덱스의 값만 반환한다.

2) 자신이 구현한 주요 코드

```
class PriorityQueue:                    #정렬된 배열을 이용한 우선순위 큐 클래스
    def __init__( self ):               #생성자
        self.items = []                 #항목 저장을 위한 리스트

    def isEmpty( self ):                #공백 상태 검사
```

```
        return len( self.items ) == 0
    def size( self ): return len(self.items)              #전체 항목의 개수
    def clear( self ): self.items = []                    #초기화

    def enqueue( self, item ):                            #삽입 연산
        if self.isEmpty():                                #공백일 때 맨 뒤에 삽입
            self.items.append( item )
        else:
            self.items.append( item )                     #맨 뒤에 삽입 후 num을 –1로 선언
            num = -1
            for i in range(self.size() - 2, -1):          #맨 뒤에서 두 번째부터 0까지 i를 반복
                if self.items[i][2] > self.items[num][2]: #값을 비교하여 swap 사용해 오름차순 정렬
                    self.items[i][2], self.items[num][2] = self.items[num][2], self.items[i][2]
                    num = i

    def dequeue( self ):                                  #삭제 연산
        if not self.isEmpty():                            #공백이 아닐 때 마지막 값을 꺼내서 반환
            return self.items.pop(-1)

    def peek( self ):                                     #peek 연산
        if not self.isEmpty():                            #공백이 아닐 때 마지막 값을 꺼내지 않고 반환
            return self.items[-1]
```

3) 다양한 입력에 대한 테스트 결과

1. 자신만의 미로를 작성하고 다음의 세 가지 방법으로 미로찾기를 실행하는 프로그램을 작성하라.

```
DFS:
(0, 5)->(1, 5)->(1, 6)->(1, 7)->(1, 4)->(1, 3)->(2, 3)->(2, 2)->(3, 2)->(4, 2)->(4, 3)->(4, 4)->(4, 5)->(5, 5)->(6, 5)->(6, 6)->(6, 7)->(7, 7
)->(8, 7)->(8, 8)->(8, 6)->(8, 5)->(8, 4)->(8, 3)->(9, 3)-> --> 미로탐색 성공
BFS:
(0, 5)->(1, 5)->(1, 4)->(1, 6)->(1, 3)->(1, 7)->(2, 3)->(2, 2)->(3, 2)->(4, 2)->(4, 1)->(4, 3)->(4, 4)->(4, 5)->(5, 5)->(6, 5)->(6, 4)->(6, 6
)->(6, 7)->(7, 7)->(8, 7)->(8, 6)->(8, 8)->(8, 5)->(8, 4)->(8, 3)->(9, 3)-> --> 미로탐색 성공
PQueue:
(0, 5)-> 우선순위 큐:  [(1, 5, -8.246211251235321)]
(1, 5)-> 우선순위 큐:  [(1, 4, -8.06225774829855), (1, 6, -8.54400374531753)]
(1, 6)-> 우선순위 큐:  [(1, 4, -8.06225774829855), (1, 7, -8.94427190999916)]
(1, 7)-> 우선순위 큐:  [(1, 4, -8.06225774829855)]
(1, 4)-> 우선순위 큐:  [(1, 3, -8.0)]
(1, 3)-> 우선순위 큐:  [(2, 3, -7.0)]
(2, 3)-> 우선순위 큐:  [(2, 2, -7.0710678118654755)]
(2, 2)-> 우선순위 큐:  [(3, 2, -6.082762530298219)]
(3, 2)-> 우선순위 큐:  [(4, 2, -5.0990195135927845)]
(4, 2)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (4, 3, -5.0)]
(4, 3)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (4, 4, -5.0990195135927845)]
(4, 4)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (4, 5, -5.385164807134504)]
(4, 5)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (5, 5, -4.47213595499958)]
(5, 5)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 5, -3.605551275463989)]
(6, 5)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (6, 6, -4.242640687119285)]
(6, 6)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (6, 7, -5.0)]
(6, 7)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (7, 7, -4.47213595499958)]
(7, 7)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (8, 7, -4.123105625617661)]
(8, 7)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (8, 6, -3.1622776601683795), (8, 8, -5.0990195135927845)]
(8, 8)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (8, 6, -3.1622776601683795)]
(8, 6)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (8, 5, -2.23606797749979)]
(8, 5)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (8, 4, -1.4142135623730951)]
(8, 4)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (8, 3, -1.0)]
(8, 3)-> 우선순위 큐:  [(4, 1, -5.385164807134504), (6, 4, -3.1622776601683795), (9, 3, -0.0)]
(9, 3)-> --> 미로탐색 성공
```

4) 코드에 대한 설명 및 해당 문제에 대한 고찰

(1) 미로는 10X10으로 설정하였으며 출발 좌표 (0, 5)에 'e', 출구 좌표 (9, 3)에 'x'로 설정하고 MAZE_SIZE를 10으로 두어, 이를 사용하여 isValidPos() 함수를 통해 (x, y)가 갈 수 있는 방인지 검사하도록 한다. 이때 (x, y)가 미로 밖이면 갈 수 없고, '0' 또는 'x'이면 갈 수 있는 방으로 구현한다.

(2) 교재 131쪽의 코드를 참고하여 stackClass 파일의 클래스 Stack을 만들고, 해당 클래스 내에 생성자, isEmpty, size, clear, push, pop, peek 함수를 만든다. 그리고 미로 탐색 파일에서 스택을 이용한 미로 탐색 코드를 구현할 때 먼저 import를 통해 stackClass의 Stack 클래스를 가지고 온다. 그리고 DFS() 함수를 만들어 스택을 이용한 미로 탐색 코드를 작성한다. 해당 코드 또한 교재 149~150쪽을 참고하여 코드를 작성하였다. 교재의 코드에서 바뀔 점은 출발 지점을 해당 미로의 'e'의 좌표 (0, 5)로 수정해주면 된다. 미로를 탐색할 때 본 파일에는 스택, 큐, 우선순위 큐를 사용하여 미로 탐색을 총 3번 실행할 예정이므로 map을 깊은 복사를 통해 사용하도록 한다. 만약 DFS()가 True이면 미로 탐색 성공, False이면 미로 탐색 실패가 출력되도록 한다.

(3) 원형 큐를 이용한 미로 탐색의 경우, queueClass 파일에 CircularQueue 클래스를 생성하여 교재 166~167쪽을 참고하여 생성자, 공백, 포화 상태, 초기화, 삽입, 삭제, peek 등의 함수를 생성하도록 한다. 원형 큐에서 주의할 점은 꼭 원형 큐의 크기를 지정하고, 여러 함수를 생성하고 반환할 때 원형 큐의 크기를 사용하여 값이 음수가 되거나 하는 오류를 방지하도록 한다. 그리고 미로 탐색 파일에서는 큐를 이용한 미로 탐색 코드를 구현할 때, 스택과 마찬가지로 먼저 import를 통해 queueClass의 CircularQueue 클래스를 가지고 온다. 그리고 BFS() 함수를 만들어 큐를 이용한 미로 탐색 코드를 작성한다. 해당 코드 역시 교재 170쪽을 참고하여 작성하였다. 출발 위치는 (0, 5)로 수정해주었으며 큐가 공백 상태가 아닐 때 함수를 계속 반복하도록 한다. 또한 큐 미로도 마찬가지로 미로를 복사할 때 깊은 복사를 사용한다.

(4) 정렬된 배열을 사용한 우선순위 큐의 경우 queueClass 파일에 PriorityQueue 클래스를 생성하고, 해당 클래스를 **2) 자신이 구현한 주요 코드**와 같이 구현한다. 설명은 해당 2)의 주석을 참고한다. 전략적 미로 탐색은 유클리디언 거리를 통해 거리 계산 함수를 구현하여 거리를 우선순위 큐 클래스를 사용하여 비교하도록 하였다. 기본적으로 거리는 (-)로 설정하였으며, 출구로부터 거리가 가까운 위치부터 우선순위 출력이 되므로 오름차순으로 정렬된 배열의 경우 마지막으로 갈수록 거리의 숫자가 0에 가까워진다.

5) 이번 과제에 대한 느낀점

이번 과제의 경우 스택과 큐에 대해서 수업 시간에 배운 것을 응용하여 푸는 문제였는데, 사실 고등학교 때도 스택과 큐에 대한 것은 이론적으로 간단히는 알고 있었지만 이렇게 직접 파이썬을 사용하여 구현해보는 것은 처음이었다. 막상 구현해보니까 이론과 다를 바가 없긴 한데 생각보다 큐는 좀 더 종류가 다양하고 복잡한 감이 있어서 좀 헷갈리기도 했다.

6) 궁금한 점이나 건의사항

딱히 없습니다.

7) 자신이 구현한 전체 코드

#미로 탐색 파일

```
maze=[[ '1', '1', '1', '1', '1', '1', '1', '1', '1', '1' ],
      [ '1', '1', '1', '1', '0', '1', '1', '1', '1', '1' ],
      [ '1', '1', '0', '0', '0', '1', '1', '1', '1', '1' ],
      [ '1', '0', '0', '1', '0', '1', '1', '1', '0', 'x' ],
      [ '1', '0', '1', '1', '0', '1', '0', '1', '0', '1' ],
      [ 'e', '0', '1', '1', '0', '0', '0', '1', '0', '1' ],
      [ '1', '0', '1', '1', '1', '1', '0', '1', '0', '1' ],
      [ '1', '0', '1', '1', '1', '1', '0', '0', '0', '1' ],
      [ '1', '1', '1', '1', '1', '1', '1', '1', '0', '1' ],
      [ '1', '1', '1', '1', '1', '1', '1', '1', '1', '1' ]]
MAZE_SIZE = 10

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'

######################################################################
#스택을 이용한 미로탐색 코드
from stackClass import Stack

def DFS():
    stack = Stack()
    stack.push((0,5))
    print('DFS: ')

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


import copy
```

```
map = copy.deepcopy(maze)
result = DFS()
if result: print(' --> 미로탐색 성공')
else: print(' --> 미로탐색 실패')

#####################################################################
#큐를 이용한 미로탐색 코드
from queueClass import CircularQueue

def BFS():
    que = CircularQueue()
    que.enqueue((0,5))
    print('BFS: ')

    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


map = copy.deepcopy(maze)
result = BFS()
if result: print(' --> 미로탐색 성공')
else: print(' --> 미로탐색 실패')

#####################################################################
#정렬된 배열을 이용한 우선순위 큐를 이용한 전략적 미로탐색 코드
from queueClass import PriorityQueue
import math

(ox, oy) = (9, 3)
def dist(x,y):
    (dx, dy) = (ox-x, oy-y)
    return math.sqrt(dx*dx + dy*dy)

def MYSmartSearch():
```

```
    q = PriorityQueue()
    q.enqueue((0, 5, -dist(0,5)))
    print('PQueue: ')

    while not q.isEmpty():
        here = q.dequeue()
        print(here[0:2], end='->')
        x,y,_ = here
        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(' 우선순위 큐: ', q.items)
    return False

map = copy.deepcopy(maze)
result = MYSmartSearch()
if result: print(' --> 미로탐색 성공')
else: print(' --> 미로탐색 실패')
```

#stackClass 파일

```
class Stack:
    def __init__( self ):
        self.top = []

    def isEmpty( self ): return len(self.top) == 0
    def size( self ): return len(self.top)
    def clear( self ): self.top = []

    def push( self, item ):
        self.top.append(item)

    def pop( self ):
        if not self.isEmpty():
            return self.top.pop(-1)

    def peek( self ):
        if not self.isEmpty():
            return self.top[-1]
```

#queueClass 파일

```
MAX_QSIZE = 10

class CircularQueue:
    def __init__( self ):
        self.front = 0
        self.rear = 0
        self.items = [None] * MAX_QSIZE

    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

    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

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

class PriorityQueue:
    def __init__( self ):
        self.items = []

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

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

def enqueue( self, item ):
    if self.isEmpty():
        self.items.append( item )
    else:
        self.items.append( item )
        num = -1
        for i in range(self.size() - 2, -1):
            if self.items[i][2] > self.items[num][2]:
                self.items[i][2], self.items[num][2] = self.items[num][2], self.items[i][2]
                num = i

def dequeue( self ):
    if not self.isEmpty():
        return self.items.pop(-1)

def peek( self ):
    if not self.isEmpty():
        return self.items[-1]
```