자료구조

과제 4. 스택/큐

과목명 : 자료구조
담당교수 : 김승태
소속학과 : 응용통계학과
학번 : 20193011
이름 : 유승희

1. 미로 탐색
최적의 미로를 찾는 프로그램 작성

1.1 스택 구성

미로를 탐색하고 미로의 길을 저장하는 알고리즘을 수행하기 위한 스택을 구성하였다. 스택의 기능을 수행하는 함수들은 다음과 같다.

<구성한 스택의 함수들>
- 데이터 멤버를 선언하고 초기화 하는 생성자
- 데이터를 스택에 넣는 push
- 제일 마지막에 들어간 데이터를 꺼내고 삭제하는 pop, 단 스택에 데이터가 없다면 빈스택이라고 알려준다.
- 가장 마지막 데이터를 표시만 하는 top
- 스택의 길이를 나타내는 len
- 스택이 비어있으면 True를 반환하는 isEmpty
- 스택에 들어있는 데이터들을 반환하는 str

위 함수들로 구성하여 스택 구조체를 작성하였다.

- 코드

```
# stack
class Stack:
    def __init__(self):
        self.items = []

    def push(self, val):
        self.items.append(val)

    def pop(self):
        try:
            return self.items.pop()
        except IndexError:
            print("Stack is empty")

    def top(self):
        try:
            return self.items[-1]
        except IndexError:
            print("Stack is empty")

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

    def isEmpty(self):
        return self.__len__() == 0

    def __str__(self):
        return str(self.items[::-1])
```

1.2 미로 불러오기 및 미로 변환

미로 불러오기
txt 파일을 불러오기 위해 반복문을 통해 한줄씩 읽고 파일내에서 줄바꿈이 생길때까지 한 str로 append 해주고 줄바꿈이 생기면 data 리스트에 append한 후 반복해서 data를 만들어 주었다. 파일 내용이 담긴 data는 1차원 리스트가 된다.

- 코드

```
# 파일에서 불러 오는 부분
data = []
with open('미로\maze1.txt') as f:
    size = f.readline()[:-1]
    while True:
        line = f.readline()
        if not line: break
        if '\n' in line:
            if '\n' != line:
                data.append(line[:-1])
        else:
            data.append(line)
```

- maze1.txt를 불러오기

```
data
```

```
['+-------+-------+-+',
 '|       |       | |',
 '| +---+ +---+ + | |',
 '|     | |     | | |',
 '+---+ | + +-+ | | |',
 '|     |   |   | | |',
 '| +-+ +-+-+ +-+ + |',
 '| | | | | | |     |',
 '| | +-+ | +-+ +-+ |',
 '| | |   |     |   |',
 '| + | + +---+-+ + |',
 '|   | |     |   | |',
 '| +-+ | +---+---+ |',
 '|     | |         |',
 '+-+---+ | +---+ + |',
 '| |     | |     | |',
 '| | +---+ | +---+-+',
 '| |       |       |',
 '| +---+ +-+-+ +-+ |',
 '|     |   | | | | |',
 '| +-+ | + | +-+ | |',
 '|   | | | |     | |',
 '+-+ | + | + +---+ |',
 '|   |   |   |     |',
 '+---+---+---+-----+']
```

미로 변환

미로의 길 찾기에 사용하기 편하도록 미로 데이터를 변환해주었다. 미로에서 움직일 때 길 위치 기준으로 동,서,남,북 방향으로 움직일 수 있다. 이 정보를 이용하여 길 위치를 변환해주었다.

움직일 수 있는 방향 정보를 의미하는 길이가 4인 move 리스트를 만들어준다. move의 인덱스는 순서대로 동,서,남,북을 의미하고 해당 방향이 뚫려있으면 (즉, 미로 데이터에서 ' '이면) 1, 벽으로 막혀있으면 (즉, 미로 데이터에서 ' '이 아니면) 0을 대입한다.
(move[0]은 동쪽, move[1]은 서쪽, move[2]는 남쪽 , move[3]은 북쪽 방향을 의미한다.

위에서 불러온 미로 data의 0,2,4,...인 짝수번째 행은 벽이 들어올 수 있는 위치이므로 길의 정보와는 무관하여 변환할 때 1,3,5.. 번째인 홀수번째 행만 바꾸어주었다. 동일한 이유로 짝수번째 열은 벽의 정보이므로 홀수번째 열만 변환해주었다.

- 코드

```
# maze 방향에 따라 표시
def newmaze(data):
    new_maze = []
    for row_idx in range(1, len(data), 2 ):
        row = []
        for col_idx in range(1, len(data[0]), 2):
            move = [0,0,0,0]
            if data[row_idx][col_idx+1] == ' ' : #동
                move[0] = 1
            if data[row_idx][col_idx-1] == ' ' : #서
                move[1] = 1
            if data[row_idx+1][col_idx] == ' ' : #남
                move[2] = 1
            if data[row_idx-1][col_idx] == ' ' : #북
                move[3] = 1
            row.append(move)
        new_maze.append(row)
    return new_maze
```

1.3 미로 길 찾기

미로의 길을 찾기 위해 도움이 될 변수와 스택을 다음과 같이 정의해주었다.

<사용한 변수와 스택과 목적>
- row_idx : 현재 위치의 행을 나타내는 변수
- col_idx : 현재 위치의 열을 나타내는 변수
- end_row_idx : 출구의 행 좌표
- end_col_idx : 출구의 열 좌표
- path : 지나온 좌표([row_idx, col_idx])의 정보들을 저장하는 스택, 즉 경로 스택
- stack : 길 탐색을 하다 갈림길(갈 수 있는 방향이 2개 이상인 경우) 해당 위치 좌표와 가지 않은 방향(move)의 정보를 저장하는 스택, 즉 갈림길 스택
- save_path : 출구까지 도달했을 때의 path를 저장하는 리스트, 이는 최종 경로를 출력할 때, 경로의 개수를 세는 곳에 이용된다.

위의 변수와 스택을 가지고 미로 길 탐색 알고리즘을 구성하였다. 길 탐색 규칙은 다음과 같다.

<길 탐색 규칙>
1. 이전에 위치에서 온 방향으로 다시 되돌아가면 안된다.
2. 뚫려있는 방향으로만 좌표 업데이트 하여 이동한다.
3. 갈림길이 나오면 동,서,남,북 순으로 가고 그 위치의 좌표와 가지 않은 방향(move)를 갈림길 스택에 저장한다.
4. 길이 막히면 가장 최근 갈림길로 돌아와 가지 않은 방향으로 이동한다.

1. 만약 이전 위치에서 동쪽으로 한칸 움직였다면 현재 위치 기준으로는 서쪽으로부터 움직였기 때문에 서쪽으로 움직이는 일은 막아야 한다. 이 작업을 하기 위해서는 이전 위치의 좌표가 필요한데 이 정보는 경로 스택인 path에 저장되어 있다. path의 top 매서드를 이용하여 바로 직전 위치와 현재 위치를 비교하여 지나온 방향을 막아주었다.
   열의 증감은 동쪽과 서쪽 방향으로의 움직임을 의미한다. 이전 위치 좌표의 col_idx가 현재 좌표의 col_idx보다 크면 서쪽 방향으로 움직인 것을 의미하므로 현재 위치 기준에서는 동쪽 방향을 막아준다. 즉 move[0]을 0으로 바꾸어준다. 반대로 이전의 col_idx가 현재의 col_idx보다 작으면 서쪽 방향을 막아준다.
   행의 증감은 북쪽과 남쪽 방향으로의 이동을 뜻하므로 이전 위치 좌표의 row_idx가 현재 좌표의 row_idx보다 크면 북쪽으로 움직인 것을 의미하므로 현재 위치 기준에서 남쪽 방향을 막아준다. 즉 move[2]를 0으로 바꾸어준다. 반대로 이전의 row_idx가 현재의 row_idx보다 작으면 북쪽 방향을 막아준다.

- 코드

```
# 어디서 왔는지 확인하고 지나온 길은 못가게 막는다.
if path.isEmpty() == False:
    if path.top()[1] > col_idx : move[0] = 0
    if path.top()[1] < col_idx : move[1] = 0
    if path.top()[0] > row_idx : move[2] = 0
    if path.top()[0] < row_idx : move[3] = 0
```

2. 뚫려 있는 방향으로만 좌표를 한칸씩 증가하여 이동하기 위해서는 우선 현재 위치에서 뚫려있는 방향의 개수를 확인해준다. 뚫려있는 방향의 개수가 한 개라면 갈 수 있는 방향은 한 곳이므로 간단해진다. 뚫려있는 방향의 개수가 2개 이상이라면 갈림길을 의미하므로 이는 3.에서 다루도록한다.
뚫려있는 방향의 개수를 확인하기 위해서는 move의 합을 이용하여 알 수 있다. 막혀있는 방향은 0, 뚫려있는 방향은 1을 이용하여 move를 구성했기 때문에 sum(move) 가 1이라면 갈 수 있는 방향을 한 곳 이여서 해당 방향에 맞는 행 또는 열의 위치를 1칸 증가시키면 된다.

- 코드

```
if sum(move) == 1: # 가는 길이 하나일 때
    if move[0] == 1 : col_idx += 1
    if move[1] == 1 : col_idx -= 1
    if move[2] == 1 : row_idx += 1
    if move[3] == 1 : row_idx -= 1
```

3. 갈림길이 나오면 동,서,남,북 순으로 가고 그 위치의 좌표와 가지 않은 방향(move)를 갈림길 스택에 저장한다. 갈림길은 2.에서 언급한 바와 같이 뚫려있는 방향이 2개 이상인 것이다. 즉, move의 합계가 1을 초과한다는 것을 의미한다. 갈림길이 나오면 동,서,남,북 순으로 길을 탐색하게 한다. 그리고 갈림길 스택에는 갈림길이 존재하는 해당 좌표와 가지 않은 방향을 저장해주어야 하는데, 만약 동쪽과 북쪽으로 나뉘는 갈림길이라면 우선 동쪽으로 먼저 탐색하기 때문에 스택에는 동쪽방향은 막힌, 북쪽 방향만 뚫려있는 move가 저장된다. 따라서 동쪽에서 길 탐색을 하다가 막혀서 되돌아오는 경우 북쪽으로 길 탐색을 실시하게 된다.

- 코드

```
elif sum(move) > 1 : # 가는 길이 여러개 일때 동,서,남,북 순으로 이동 한후 지나왔던 방향은 막기
    print(f"PUSH({row_idx},{col_idx})")
    if move[0] == 1 :
        move[0] = 0
        stack.push([row_idx, col_idx, move])
        col_idx += 1
    elif move[1] == 1:
        move[1] = 0
        stack.push([row_idx, col_idx, move])
        col_idx -= 1
    elif move[2] == 1:
        move[2] = 0
        stack.push([row_idx, col_idx, move])
        row_idx += 1
    elif move[3] == 1:
        move[3] = 0
        stack.push([row_idx, col_idx, move])
        row_idx -= 1
```

4. 길이 막히면 가장 최근 갈림길로 돌아와 가지 않은 방향으로 길 탐색을 실시한다. 길이 막히는 경우는 현재 위치의 move에서 뚫려있는 방향이 한 개도 존재하지 않는 것을 의미하므로 sum(move)가 0일때이다. 길이 막히면 갈림길 스택의 최근 정보를 pop해서 되돌아 간다. 또한 다시 갈림길 좌표로 돌아갔기 때문에 이동할때마다의 좌표를 저장하던 경로 스택에 잘못된 경로 정보가 들어가있다. 따라서 갈림길 위치까지 반복문을 통해 갈림길로부터 막다른 길까지의 경로 좌표를 다 지워준다.

-코드

```
elif sum(move) == 0 : # 모든 방향이 막혔을때

    if stack.isEmpty(): # 들아갈 갈림길이 없으면 반복문 중단 및 현재까지 찾은 경로개수 출력
        print(f"모두 {len(save_path)}개의 길을 찾았습니다.")
        break

    poped = True  # 갈림길로 들아갔기 때문에 pop변수 true로 변환
    row_idx, col_idx, move = stack.pop()
    print(f"POP({row_idx},{col_idx})")
    while 1: # 갈림길로 부터 막다른 길까지의 경로 다 지우기
        if path.top()[0] == row_idx and path.top()[1] == col_idx: # 갈림길 위치까지 반복문을 통해 좌표 지우기
            break
        path.pop()
```

위 규칙들을 한번에 모아 반복문을 돌리면서 최종경로를 저장하는 방법과 반복문을 끝내는 경우를 추가해주었다.

우선 지나온 방향과 갈림길에 따라 move의 상태 자주 변하는데 이렇게 되면 초기의 move가 바뀌어 잘못된 경로를 찾을 수 있다. 그렇기 때문에 new_maze[row_idx][col_idx].copy()를 해주어 초기의 move가 바뀌지 않게 해준다. 하지만 얕은 복사로 인해 갈림길 stack에 저장한 move가 아닌 초기의 move가 들어가는 문제를 해결하기 위해 poped 변수를 이용해 갈림길에서 나온 move면 stack의 move를 그렇지 않으면 기존의 move를 사용하게 해주었다.

출구 좌표 row_idx는 (변환한 미로의 길이 -1) , col_idx는 (변환한 미로의 행의 길이 -1)로 지정해두어 출구에 도달하면 현재까지 저장되어있는 경로를 save_path 리스트에 넣어준다.

또한 길을 탐색하다 보면 다시 시작점(0,0)으로 돌아가 무한반복하게 되는 문제점이 생길 수가 있는데 이는 시작점의 전 위치에서 시작점으로 움직일 수 있는 방향을 막아준다.

또한 길 탐색을 반복하다가 막다른 길에 도달했는데 돌아갈 스택이 없다면 이 때까지 저장해둔 save_path의 길이를 통해 최종 경로 개수를 확인하며 반복문을 빠져나온다.

- 코드

```
path = Stack() # 지나온 경로
stack = Stack() # 갈림길에서 좌표랑 방향
save_path = [] # 출구까지 도달했을때의 path 저장하는 리스트

#시작 좌표는 0,0이다.
row_idx = 0
col_idx = 0

# 출구 좌표 설정하기
end_row_idx = len(new_maze)-1
end_col_idx = len(new_maze[0])-1

poped = False
while 1: #

    if poped == False:
        move = new_maze[row_idx][col_idx].copy()
    poped = False

    # 어디서 왔는지 확인하고 지나온 길은 못가게 막는다.
    if path.isEmpty() == False:
        if path.top()[1] > col_idx : move[0] = 0
        if path.top()[1] < col_idx : move[1] = 0
        if path.top()[0] > row_idx : move[2] = 0
        if path.top()[0] < row_idx : move[3] = 0

    # 다시 시작점(0,0)으로 들아가 무한반복 막는다.
    if row_idx == 0 and col_idx == 1 :
        move[1] = 0
    if row_idx == 1 and col_idx == 0 :
        move[3] = 0

    # 현재 위치를 경로 스택에 넣기
    path.push([row_idx, col_idx])

    # 현재 위치가 출구 좌표라면 현재 현재 까지의 경로 하나를 save_path 리스트에 넣어준다.
    if row_idx == end_row_idx and col_idx == end_col_idx:
        save_path.append(path.items.copy())

    # 가는거
    if sum(move) == 1: # 가는 길이 하나일 때
        if move[0] == 1 : col_idx += 1
        if move[1] == 1 : col_idx -= 1
        if move[2] == 1 : row_idx += 1
        if move[3] == 1 : row_idx -= 1

    elif sum(move) > 1 : # 가는 길이 여러개 일때 동,서,남,북 순으로 이동 한후 지나왔던 방향은 막기
        print(f"PUSH({row_idx},{col_idx})")
        if move[0] == 1 :
            move[0] = 0
            stack.push([row_idx, col_idx, move])
            col_idx += 1
        elif move[1] == 1:
            move[1] = 0
            stack.push([row_idx, col_idx, move])
            col_idx -= 1
        elif move[2] == 1:
            move[2] = 0
            stack.push([row_idx, col_idx, move])
            row_idx += 1
        elif move[3] == 1:
            move[3] = 0
            stack.push([row_idx, col_idx, move])
            row_idx -= 1

    elif sum(move) == 0 : # 모든 방향이 막혔을때

        if stack.isEmpty(): # 들아갈 갈림길이 없으면 반복문 중단 및 현재까지 찾은 경로개수 출력
            print(f"모두 {len(save_path)}개의 길을 찾았습니다.")
            break

        poped = True  # 갈림길로 들아갔기 때문에 pop변수 true로 변환
        row_idx, col_idx, move = stack.pop()
        print(f"POP({row_idx},{col_idx})")
        while 1: # 갈림길로 부터 막다른 길까지의 경로 다 지우기
            if path.top()[0] == row_idx and path.top()[1] == col_idx: # 갈림길 위치까지 반복문을 통해 좌표 지우기
                break
            path.pop()
```

1.4 미로 경로 표시

경로를 변환되기 전 미로 데이터에 표시하기 위해서는 경로 스택의 정보를 이용해 순서대로 좌표를 미로에 그려준다. 인덱스 설정만 주의하여 지나온 경로를 “O”로 표시해준다. path에 들어있는 좌표가 (0,0) 이고 (0,1) 이라면 변환되기 전 미로의 좌표에서는 (1,1) 과 (1,3) 이 된다. 이를 함수로 만들어주고 경로가 2개 이상인 경우도 있기 때문에 save_path를 이용하여 반복문을 이용해 가능한 경로를 모두 표시해준다.

- 코드

```
# 경로를 미로에  그리는 함수
def print_maze(data, path):
    copy_data = data.copy()
    for p in path:
        row = list(copy_data[p[0]*2+1])
        row[p[1]*2+1] = "O"
        copy_data[p[0]*2+1]= ''.join(row)

    for i in range(len(copy_data)):
        print(copy_data[i])
for path in save_path:
    print(f"경로 {save_path.index(path)+1}/{len(save_path)}")
    print_maze(data, path)
    print("====================")
```

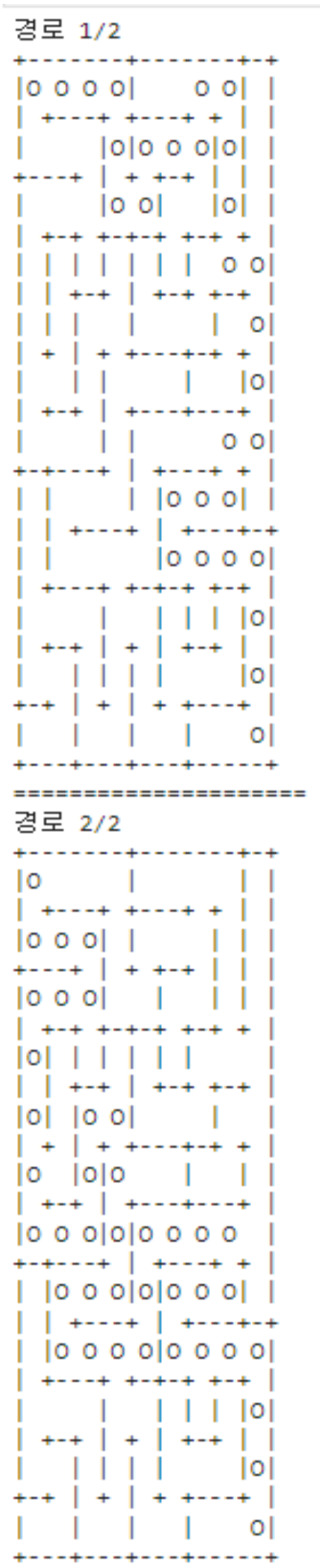
1.5 최종 결과 (maze1.txt 기준)

경로 탐색

```
PUSH(0,0)
PUSH(1,6)
POP(1,6)
PUSH(0,6)
PUSH(3,7)
PUSH(3,8)
PUSH(4,8)
POP(4,8)
PUSH(6,8)
PUSH(6,7)
PUSH(8,3)
PUSH(5,3)
POP(5,3)
PUSH(4,3)
PUSH(5,0)
POP(5,0)
PUSH(2,2)
POP(2,2)
POP(4,3)
POP(8,3)
PUSH(9,3)
PUSH(10,5)
POP(10,5)
POP(9,3)
PUSH(9,0)
POP(9,0)
POP(6,7)
PUSH(8,6)
POP(8,6)
POP(6,8)
POP(3,8)
POP(3,7)
POP(0,6)
POP(0,0)
PUSH(2,2)
PUSH(5,0)
POP(5,0)
PUSH(4,3)
PUSH(5,3)
POP(5,3)
PUSH(8,3)
PUSH(6,7)
PUSH(6,8)
```

```
POP(6,8)
PUSH(4,8)
POP(4,8)
PUSH(3,8)
PUSH(3,7)
POP(3,7)
PUSH(0,6)
POP(0,6)
PUSH(1,6)
POP(1,6)
POP(3,8)
POP(6,7)
PUSH(8,6)
POP(8,6)
POP(8,3)
PUSH(9,3)
PUSH(10,5)
POP(10,5)
POP(9,3)
PUSH(9,0)
POP(9,0)
POP(4,3)
POP(2,2)
모두 2개의 길을 찾았습니다.
```

경로 표시



2. 큐 운영하기

문자가 입력되어 큐에 쌓이는 시스템에서 필요한 수만큼 문자를 가져오는 프로그램

1.1 원형 큐 구성하기

문제를 해결하고 출력하기 위해 원형큐를 구성하였다. 원형큐의 기능을 수행하는 함수들은 다음과 같다. 원형큐의 최대 사이즈는 20으로 지정해준다.

<구성한 원형큐의 함수들>

- 데이터 멤버를 None값으로 최대 사이즈 만큼 초기화 하는 생성자, 이때 front와 rear는 0이다.
- front와 rear가 같으면 큐가 비어있는 의미로 큐가 비어있으면 True를 반환하는 isEmpty
- front의 위치와 rear의 다음 위치가 같으면 가득찬 큐의 의미로 큐가 가득차있으면 True를 반환하는 isFull
- 큐를 공백상태로 만드는 clear
- 데이터를 큐의 맨 뒤에 추가하는 enqueue, 이는 큐가 가득차있지 않을 때 추가되며 rear를 한칸 증가한 후 그 위치에 데이터를 추가한다. 또한 결과를 출력하기 위해 현재 작업 상태와 front, rear 값을 출력한다.
- 큐의 맨 앞에 있는 항목을 꺼내 반환하는 dequeue, 이는 큐가 비어있지 않을 때 반환되며 front의 위치를 한칸 증가한 후 해당 위치의 데이터를 반환한다. enqueue처럼 현재 상태를 출력하고 큐가 비어있으면 dequeue 실행 실패한 것을 출력한다.
- 큐의 맨 앞에 있는 항목을 삭제하지 않고 반환하는 peek
- 큐의 현재 상태를 나타내는 display. 이는 front가 rear보다 작으면 front다음 위치부터 rear까지의 데이터를 표시해주고, front가 rear보다 크면 front 다음부터 마지막까지 그리고 0부터 rear까지의 데이터를 표시해준다. 출력을 위해서는 들어있는 큐의 데이터와 길이를 표시해준다.

위 함수들로 구성하여 원형큐 구조체를 작성하였다.

- 코드

```
MAX_SIZE = 20

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

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

    def isFull(self):
        return self.front == (self.rear+1)%MAX_SIZE

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

    def enqueue( self, item) :
        if not self.isFull():
            self.rear = (self.rear+1)%MAX_SIZE
            self.items[self.rear] = item
            print(f"(SYSTEM) ADDQUEUE({item}) F={self.front}, R={self.rear}")

    def dequeue(self):
        if not self.isEmpty():
            self.front = (self.front+1)%MAX_SIZE
            print(f"DELETEQUEUE() = {self.items[self.front]} ,F={self.front}, R={self.rear}")
            return self.items[self.front]

        else:
            print("DELETEQUEUE( ) FAIL. QueueEmpty")


    def peek(self):
        if not self.isEmpty():
            return self.items[(self.front+1)%MAX_SIZE]

    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_SIZE]+self.items[0:self.rear+1]
        if not self.isEmpty():
            print(f"QUEUE = {''.join(out)}({len(out)})")
        else :
            print("QUEUE = (0)")
```

1.2 큐 입력 및 출력 하기

새로운 큐를 선언해주고 사용자가 입력한 것을 변수 x에 저장한다. 입력받은 x가 '0'과 '9' 사이라면 사용자가 숫자를 입력한 의미이므로 정수형으로 바꾸어주고 0이라면 현재 큐의 상태를 표시해준다. 그렇지 않으면 사용자가 입력한 숫자만큼 원형큐 dequeue를 실행한다. 사용자가 문자를 입력했다면 이를 리스트로 변환하여 원형큐에 enqueue를 해준다. 무한 반복문으로 실시하였고 데이터가 큐에 들어갔다가 empty가 되면 반복문을 빠져나오도록 설정해주었다.

- 코드

```
q = CircularQueue()
count =0
while 1:
    x = input()
    if(x >= '0' and x <= '9'):
        x = int(x)
        if x == 0:
            q.display()
        else:
            for i in range(x):
                q.dequeue()
    else:
        x = list(x)
        for i in x:
            q.enqueue(i)
            count += 1

    if q.isEmpty() == True and count != 0:
        break
```

1.3 최종 결과

```
 0
QUEUE = (0)
 ABC
(SYSTEM) ADDQUEUE(A) F=0, R=1
(SYSTEM) ADDQUEUE(B) F=0, R=2
(SYSTEM) ADDQUEUE(C) F=0, R=3
 DE
(SYSTEM) ADDQUEUE(D) F=0, R=4
(SYSTEM) ADDQUEUE(E) F=0, R=5
 3
DELETEQUEUE() = A ,F=1, R=5
DELETEQUEUE() = B ,F=2, R=5
DELETEQUEUE() = C ,F=3, R=5
 0
QUEUE = DE(2)
 FG
(SYSTEM) ADDQUEUE(F) F=3, R=6
(SYSTEM) ADDQUEUE(G) F=3, R=7
 5
DELETEQUEUE() = D ,F=4, R=7
DELETEQUEUE() = E ,F=5, R=7
DELETEQUEUE() = F ,F=6, R=7
DELETEQUEUE() = G ,F=7, R=7
DELETEQUEUE( ) FAIL. QueueEmpty
```