

Data Structure

자료구조



큐



한국기술교육대학교
온라인평생교육원

학습내용

- 큐의 개념과 동작 원리
- 선형 큐와 원형 큐
- 원형 큐의 구현
- 너비 우선 탐색

학습목표

- 큐의 개념과 동작 원리를 설명할 수 있다.
- 선형 큐와 원형 큐의 차이를 설명할 수 있다.
- 원형 큐를 구현할 수 있다.
- 너비 우선 탐색으로 미로의 출구를 찾을 수 있다.



큐의 개념과 동작 원리

큐

큐의 개념과 동작 원리

1 큐의 개념과 동작 원리에 대한 이해

큐(Queue)

- 선형 자료구조의 일종
- 자료의 입출력이 선입 선출(FIFO)로 일어남
 - 가장 먼저 들어온(First-In) 데이터가 가장 먼저 나감(First-Out)
 - First-In First-Out(FIFO)
- 입력과 같은 순서의 출력이 필요한 응용에 사용

큐

큐의 개념과 동작 원리

1 큐의 개념과 동작 원리에 대한 이해

예 공항 카운터의 대기열



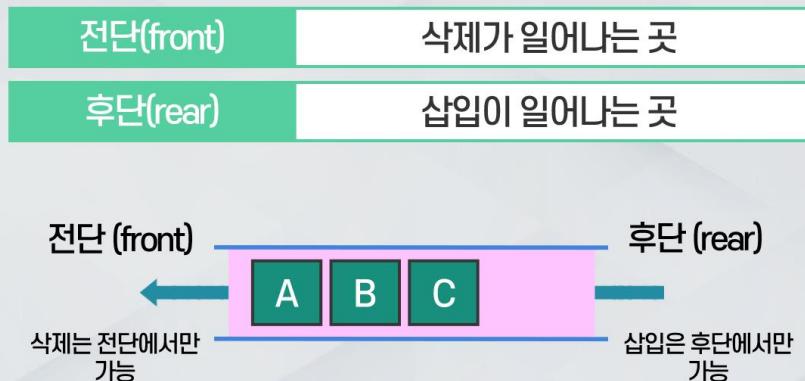
큐의 개념과 동작 원리

큐

큐의 개념과 동작 원리

1 큐의 개념과 동작 원리에 대한 이해

● 큐의 구조



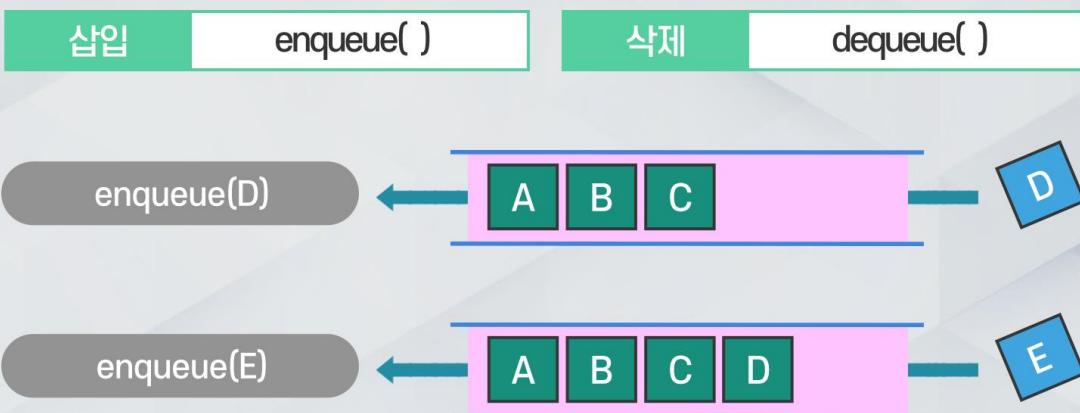
큐

큐의 개념과 동작 원리

1 큐의 개념과 동작 원리에 대한 이해

● 큐의 구조

▶ 큐의 일련의 검사





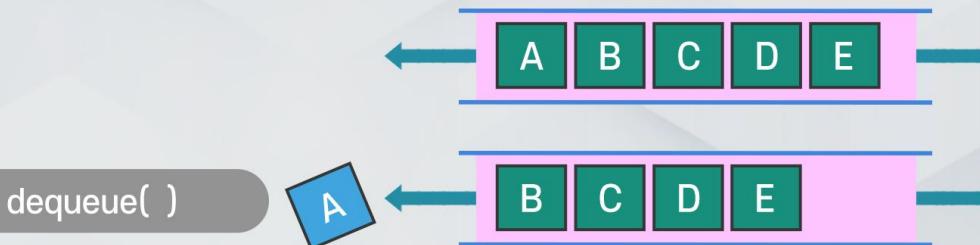
큐의 개념과 동작 원리

큐

큐의 개념과 동작 원리

1 큐의 개념과 동작 원리에 대한 이해

◎ 큐의 구조



큐

큐의 개념과 동작 원리

2 큐의 추상 자료형

데이터

선입 선출(FIFO)의 접근 방법을 유지하는 항목들의 모임

▶ 연산

Queue()	비어 있는 새로운 큐를 만듦
isEmpty()	큐가 공백상태이면 True를 아니면 False를 반환
isFull()	큐가 포화상태이면 True를 아니면 False를 반환
enqueue(x)	항목 x를 큐의 맨 위에 추가
dequeue()	큐의 맨 앞에 있는 항목을 꺼내 반환



큐의 개념과 동작 원리

큐

큐의 개념과 동작 원리

2 큐의 추상 자료형

▶ 연산

peek()	큐의 전단에 있는 항목을 삭제하지 않고 반환
size()	모든 항목들의 개수를 반환
clear()	큐를 공백상태로 만듦
display()	큐를 화면에 보기 좋게 출력

큐

큐의 개념과 동작 원리

3 큐의 용도

컴퓨터에서 버퍼의 용도로 매우 광범위하게 사용

버퍼(buffer)

- 컴퓨터에서 데이터를 주고 받을 때 각 주변 장치들 사이에 존재하는 속도 차이나 시간 차이를 극복하기 위한 임시 기억 장치



큐의 개념과 동작 원리

큐

큐의 개념과 동작 원리

3 큐의 용도

● 큐의 응용 분야

- 1 프린터와 컴퓨터 사이의 **인쇄 작업 큐** (**버퍼링**)
- 2 실시간 비디오 스트리밍에서의 **버퍼링**
- 3 시뮬레이션의 **대기열** (공항의 비행기, 은행에서 대기열)
- 4 통신에서의 **데이터 패킷들의 모델링**에 이용
- 5 서비스 센터의 **콜 큐** (클라이언트와 서버의 속도 차이 극복)



선형 큐와 원형 큐

큐

선형 큐와 원형 큐

1 큐의 구현 방법과 선형 큐

큐를 구현할 수 있는 두 가지 구조

- 배열 구조
- 연결된 구조

배열 구조의 큐 구현 방법

- 선형 큐: 배열을 선형으로 사용하는 방법
- 원형 큐: 배열을 원형으로 사용하는 방법

큐

선형 큐와 원형 큐

1 큐의 구현 방법과 선형 큐

배열 구조의 큐를 위한 변수

- front: 최근에 삭제가 일어난 위치
- rear: 최근에 삽입이 일어난 위치

선형 큐와 원형 큐

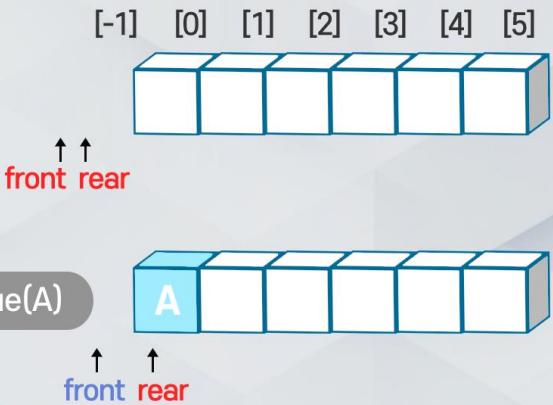
큐

선형 큐와 원형 큐

1 큐의 구현 방법과 선형 큐

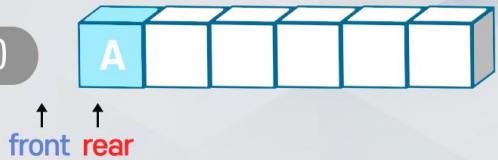
- 배열을 선형으로 사용하는 큐

최초에는 front와 rear 모두 -1



후단 삽입 → rear 증가

enqueue(A)



큐

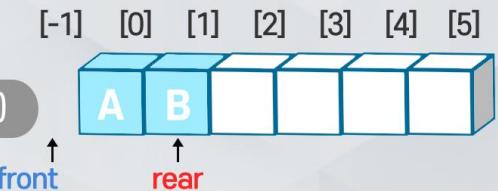
선형 큐와 원형 큐

1 큐의 구현 방법과 선형 큐

- 배열을 선형으로 사용하는 큐

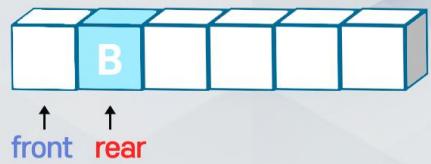
후단 삽입 → rear 증가

enqueue(B)



전단 삭제 → front 증가

dequeue()





선형 큐와 원형 큐

큐

선형 큐와 원형 큐

1 큐의 구현 방법과 선형 큐

● 선형 큐의 문제점

삽입 시 불필요한 항목의 이동이 필요한 경우 발생

- rear가 배열의 맨 끝을 가리키는 경우

예

- 큐의 용량: 6
- 6번의 삽입(enqueue), 2번의 삭제(dequeue) 연산이 수행
- 이 상황에서 추가적인 삽입 연산을 하는 경우

enqueue(A), enqueue(B), enqueue(C)
enqueue(D), enqueue(E), enqueue(F)
dequeue(), dequeue()

enqueue(G)

큐

선형 큐와 원형 큐

1 큐의 구현 방법과 선형 큐

● 선형 큐의 문제점

[-1] [0] [1] [2] [3] [4] [5]

O(1)

enqueue(A)
enqueue(B)
enqueue(C)
enqueue(D)
enqueue(E)
enqueue(F)
dequeue()
dequeue()



enqueue(G) ?



선형 큐와 원형 큐

큐

선형 큐와 원형 큐

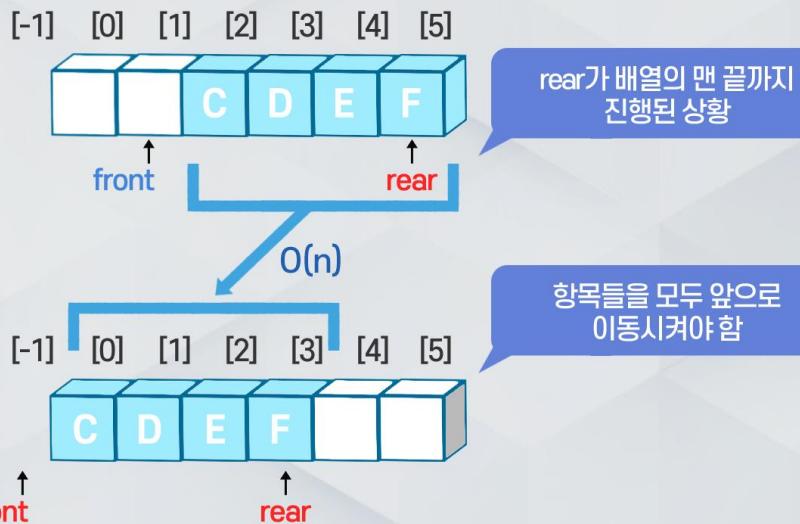
1 큐의 구현 방법과 선형 큐

● 선형 큐의 문제점

 $O(1)$

```
enqueue(A)
enqueue(B)
enqueue(C)
enqueue(D)
enqueue(E)
enqueue(F)
dequeue( )
dequeue( )
```

enqueue(G) ?



큐

선형 큐와 원형 큐

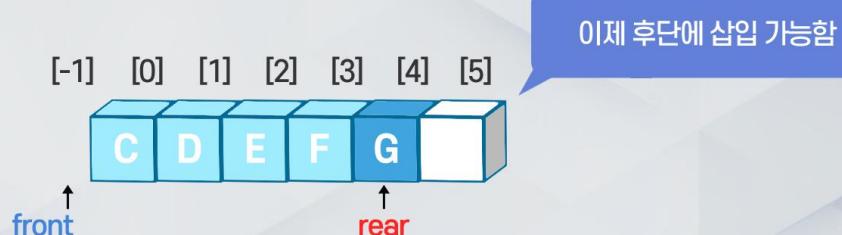
1 큐의 구현 방법과 선형 큐

● 선형 큐의 문제점

 $O(1)$

```
enqueue(A)
enqueue(B)
enqueue(C)
enqueue(D)
enqueue(E)
enqueue(F)
dequeue( )
dequeue( )
```

enqueue(G) ?





선형 큐와 원형 큐

큐

선형 큐와 원형 큐

2 원형 큐의 동작

선형 큐의 문제를 해결하는 방법?

- 배열을 원형으로 사용
- 인덱스만 회전시키면 됨

front와 rear의 시계방향 회전 code

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

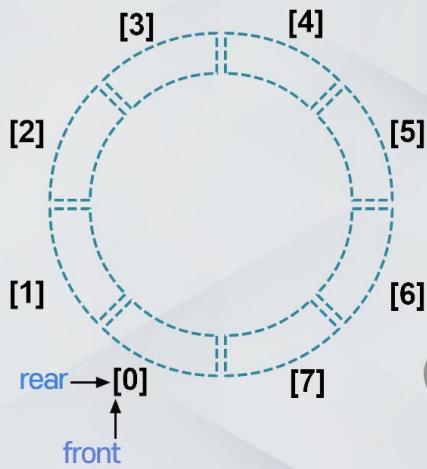


큐

선형 큐와 원형 큐

2 원형 큐의 동작

◎ 원형 큐의 연산 예 초기상태



초기 상태



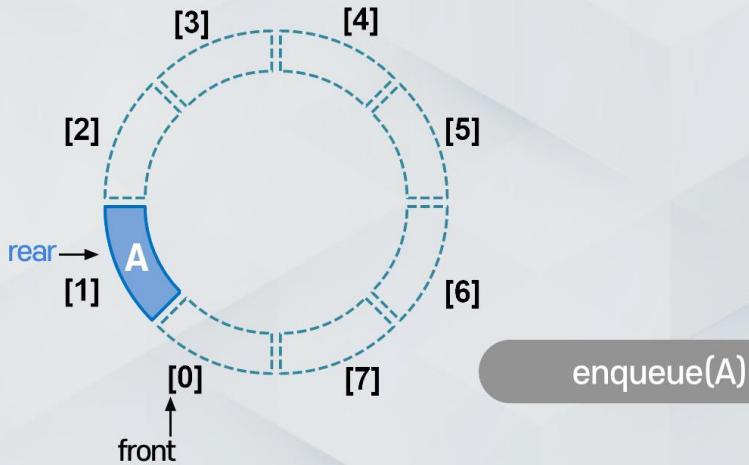
선형 큐와 원형 큐

큐

선형 큐와 원형 큐

2 원형 큐의 동작

- 원형 큐의 연산 예 enqueue(A)

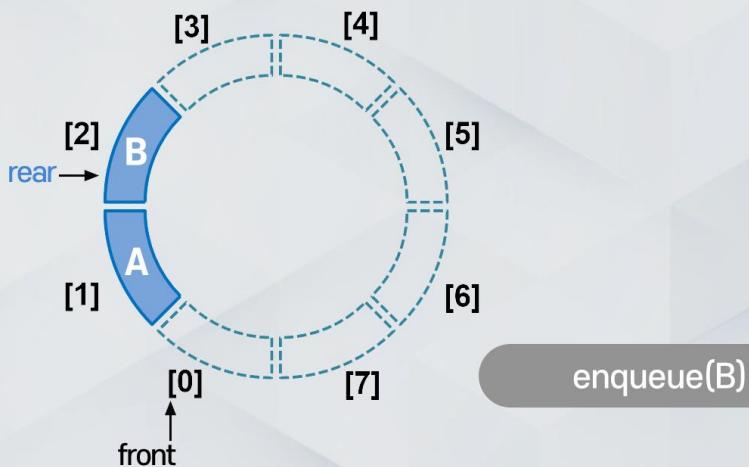


큐

선형 큐와 원형 큐

2 원형 큐의 동작

- 원형 큐의 연산 예 enqueue(B)





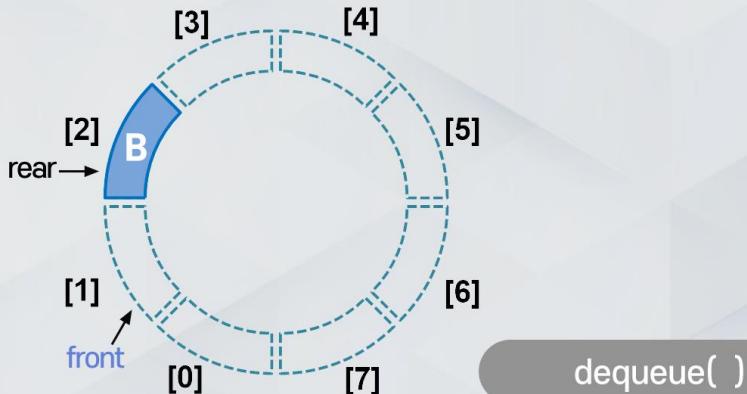
선형 큐와 원형 큐

큐

선형 큐와 원형 큐

2 원형 큐의 동작

- 원형 큐의 연산 예 `dequeue()`

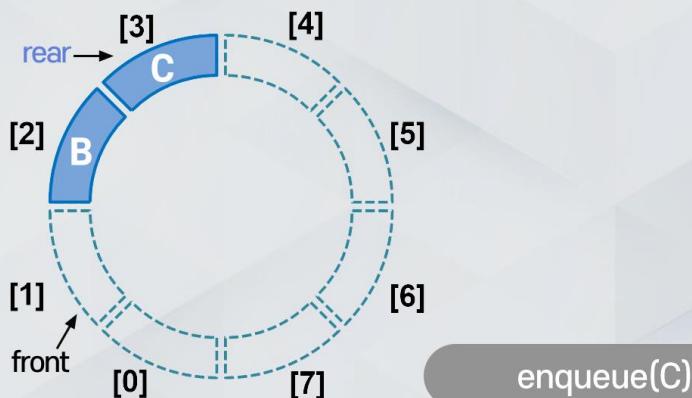


큐

선형 큐와 원형 큐

2 원형 큐의 동작

- 원형 큐의 연산 예 `enqueue(C)`



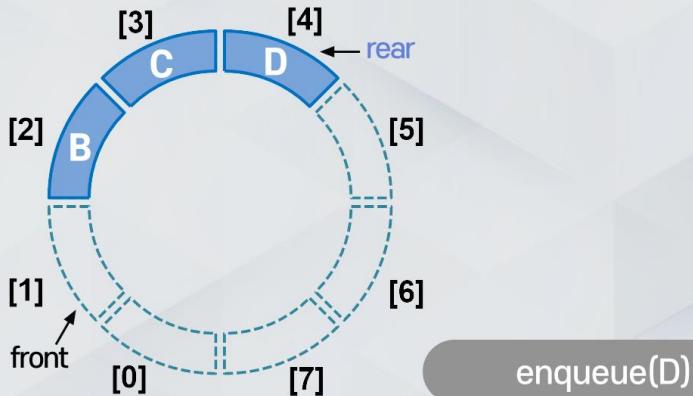
선형 큐와 원형 큐

큐

선형 큐와 원형 큐

2 원형 큐의 동작

- 원형 큐의 연산 예 enqueue(D)



enqueue(D)

큐

선형 큐와 원형 큐

2 원형 큐의 동작

- 공백 상태와 포화 상태

공백 상태

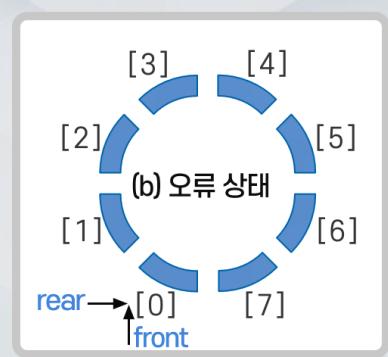
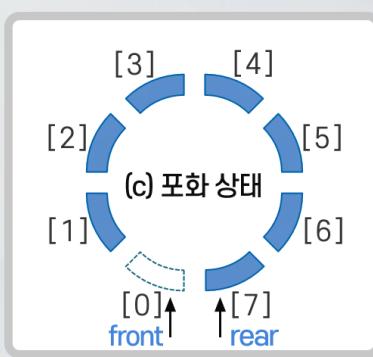
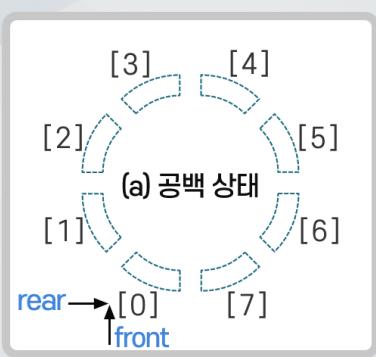
front == rear

포화 상태

front == (rear+1) % MAX_QSIZE

공백 상태와 포화 상태 구별 방법

하나의 공간은 항상 비워 둠



원형 큐의 구현

큐

원형 큐의 구현

1 원형 큐의 데이터

원형 큐를 클래스로 구현

- ‘배열’로 파이썬의 리스트 이용
- 배열의 크기(큐의 용량) 고정

데이터 멤버

- 항목 저장을 위한 배열
- 큐의 용량(상수)
- 전단(가장 최근에 삭제된 위치)을 위한 변수
- 후단(가장 최근에 삽입된 위치)을 위한 변수

큐

원형 큐의 구현

1 원형 큐의 데이터

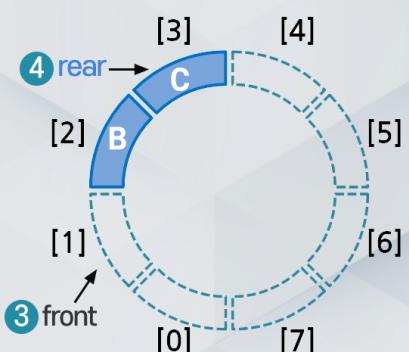
● 배열 구조의 원형 큐 설계

1 큐에 저장할 항목 보관
배열 구조(파이썬의 리스트)

2 큐의 용량
최대 저장 가능한 항목의 수

3 전단의 인덱스
가장 최근에 삭제된 위치, 시계 방향으로 회전

4 후단의 인덱스
가장 최근에 삽입된 위치, 시계 방향으로 회전



items[capacity]

1

2



원형 큐의 구현

큐

원형 큐의 구현

1 원형 큐의 데이터

● 클래스와 생성자

원형 큐의 데이터 멤버

- 1차원 배열에 저장 → **items**
 - 파이썬의 리스트 사용
 - 큐의 용량 → **capacity**
- 전단 인덱스를 위한 변수 → **front**
 - 다음 삭제될 항목: items [$(\text{front}+1) \% \text{capacity}$]
- 후단 인덱스를 위한 변수 → **rear**
 - 다음 삽입될 항목: items [$(\text{rear}+1) \% \text{capacity}$]

큐

원형 큐의 구현

1 원형 큐의 데이터

● 클래스와 생성자

▶ 원형 큐 클래스

```
class CircularQueue :
    def __init__( self, capacity = 10 ) :
        self.capacity = capacity
        self.front = 0
        self.rear = 0
        self.items = [None] * capacity
```

원형 큐의 구현

큐

원형 큐의 구현

2 원형 큐의 연산

기본 연산들

공백 상태와 포화 상태 검사

삽입 연산

삭제 연산

기타 연산들

큐

원형 큐의 구현

2 원형 큐의 연산

● 기본 연산들

기본 연산들

- 큐 초기화
- 큐의 전체 항목 수

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

```
def size( self ) :
    return (self.rear - self.front + self.capacity) % self.capacity
```



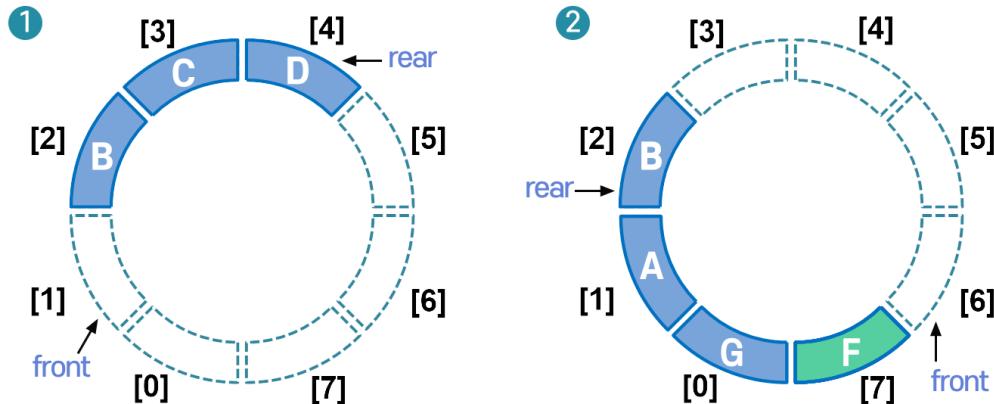
원형 큐의 구현

큐

원형 큐의 구현

2 원형 큐의 연산

● 큐의 전체 항목 수



큐

원형 큐의 구현

2 원형 큐의 연산

● 공백 상태와 포화상태

공백 상태와 포화 상태 검사

- 공백 상태: $\text{front} == \text{rear}$
- 포화 상태: $\text{front} == (\text{rear} + 1) \% \text{MAX_QSIZE}$

```

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

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



원형 큐의 구현

큐

원형 큐의 구현

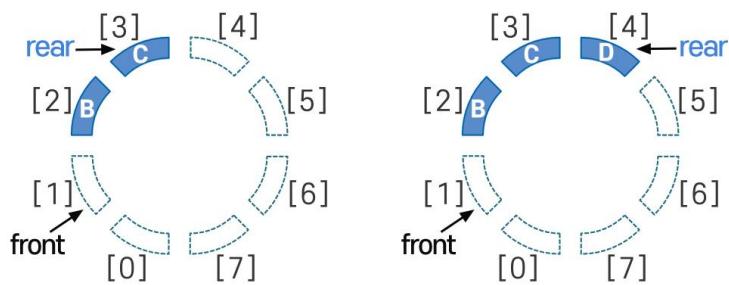
2 원형 큐의 연산

◎ 삽입 연산

enqueue()

- rear 가 회전
- 먼저 포화상태 검사 필요

예 enqueue(D)



큐

원형 큐의 구현

2 원형 큐의 연산

◎ 삽입 연산

enqueue()

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



원형 큐의 구현

큐

원형 큐의 구현

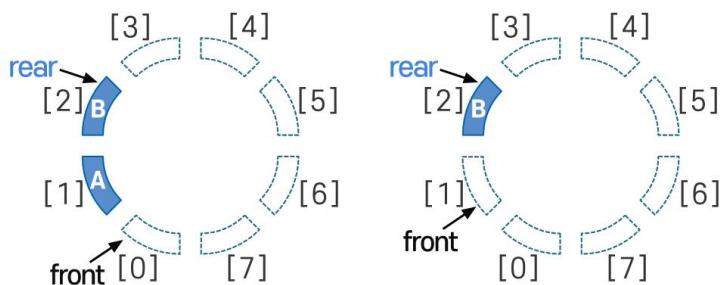
2 원형 큐의 연산

◎ 삭제 연산

dequeue()

- front가 회전
- 먼저 공백 상태 검사 필요

예 dequeue()



큐

원형 큐의 구현

2 원형 큐의 연산

◎ 삭제 연산

dequeue()

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



원형 큐의 구현

큐

원형 큐의 구현

2 원형 큐의 연산

◎ 기타 연산들

peek()

공백 상태 검사 필요

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

큐

원형 큐의 구현

2 원형 큐의 연산

◎ 기타 연산들

display()

front+1부터 rear까지 순서대로 출력

```
def display(self, msg='Queue:'):
    print(msg, end=' = [ ')
    count = self.size()
    for i in range(count):
        print(self.items[(self.front+1+i)%self.capacity], end=', ')
    print(" ]")
```

원형 큐의 구현

```

4 ##### 큐 ADT의 구현: 원형 큐의 구현 #####
5 # 큐 ADT의 구현: 원형 큐의 구현
6 #####
7 class CircularQueue :
8     def __init__( self, capacity = 10 ) :
9         self.capacity = capacity
10        self.front = 0
11        self.rear = 0
12        self.items = [None] * capacity
13
14    def isEmpty( self ) :
15        return self.front == self.rear
16
17    def isFull( self ) :
18        return self.front == (self.rear+1)%self.capacity
19
20    def clear( self ) :
21        self.front = self.rear
22
23    def size( self ) :
24        return (self.rear - self.front + self.capacity) % self.capacity
25

```

실습 단계

생성자 코드

size() 연산: self.rear에 capacity를 더한 후, front 뺀 전체 값을 나머지 처리

삽입 연산: 포화 상태 검사가 필요

삭제 연산: 공백 상태 검사가 필요

삭제 연산과 peek 연산의 비교 (self.front의 변경 여부)

테스트 코드

파이썬의 특징: 큐나 리스트에 꼭 같은 자료형의 항목만 넣을 필요는 없음

실행 결과 확인



원형 큐의 구현

큐

원형 큐의 구현

2 원형 큐의 연산

● 파이썬의 queue 모듈

사용하기 위해서 먼저 queue 모듈을 import 해야 함

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

스택과 큐를 모두 제공

- 스택 : LifoQueue
- 큐 : Queue

큐

원형 큐의 구현

2 원형 큐의 연산

● 파이썬의 queue 모듈

스택 객체 생성

예 S = queue.LifoQueue(maxsize = 100)

큐 객체 생성

예 Q = queue.Queue(maxsize = 100)



원형 큐의 구현

큐

원형 큐의 구현

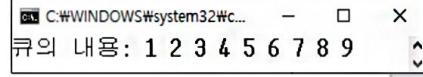
2 원형 큐의 연산

● 파이썬의 queue 모듈

연산의 이름 변경

- 삽입(push(), enqueue()) → put()
- 삭제(pop(), dequeue()) → get()

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



peek() 연산은 지원하지 않음

너비 우선 탐색

큐

너비 우선 탐색

1 큐의 활용 분야

자료구조에서의 큐 활용

- 이진 트리의 레벨 순회 알고리즘
- 그래프의 탐색에서 너비 우선 탐색 알고리즘
- 기수정렬에서 레코드의 정렬을 위해 사용

미로 탐색에서의 큐 활용

- 너비 우선 탐색(BFS, Breadth First Search) 방법 지원
- 스택: 깊이 우선 탐색(DFS, Depth First Search)

큐

너비 우선 탐색

1 큐의 활용 분야

미로 탐색에서의 전략

- 현재의 위치에서 가능한 방향을 **어딘가에 저장**
- 막다른 길을 만나면 **저장된 다음 위치**를 꺼냄

너비 우선 탐색(BFS, Breadth First Search)

- 큐에 가능한 다음 위치를 순서대로 저장
- 막다른 길을 만나면 가장 먼저 저장된 위치를 다시 시도
 - 입구에서 가까운 위치를 먼저 탐색

너비 우선 탐색

큐

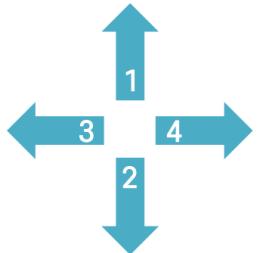
너비 우선 탐색

2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우

위치 표시: (x, y)



이웃 위치 탐색 순서

	0	1	2	3	4	5
0						
1		(1)			(7)	(9)
2		(2)	(4)	(6)		
3		(3)		(8)		
4		(5)		(10)	(11)	(12)
5						

큐

너비 우선 탐색

2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우

삭제



삽입



출구

<큐>

너비 우선 탐색

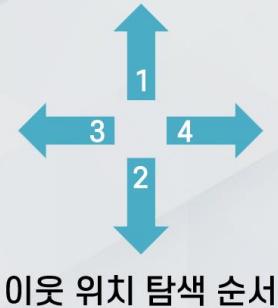
큐

너비 우선 탐색

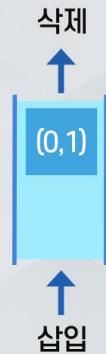
2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우



	0	1	2	3	4	5
0						
1	(1)			(7)	(9)	
2	(2)	(4)		(6)		
3	(3)		(8)			
4	(5)		(10)	(11)	(12)	
5						



큐

너비 우선 탐색

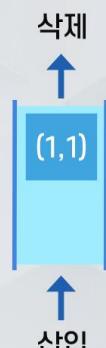
2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우



	0	1	2	3	4	5
0						
1	(1)			(7)	(9)	
2	(2)	(4)		(6)		
3	(3)		(8)			
4	(5)		(10)	(11)	(12)	
5						



너비 우선 탐색

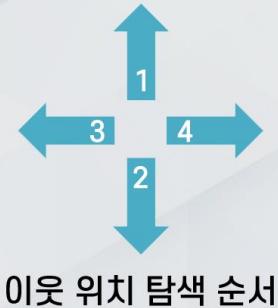
큐

너비 우선 탐색

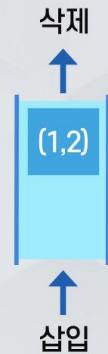
2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우



	0	1	2	3	4	5
0						
1		(1)		(7)	(9)	
2			(4)	(6)		
3		(3)		(8)		
4		(5)		(10)	(11)	(12)
5						



큐

너비 우선 탐색

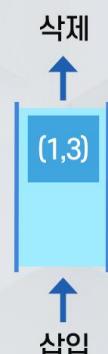
2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우



	0	1	2	3	4	5
0						
1		(1)		(7)	(9)	
2		(2)	(4)	(6)		
3		(5)		(8)		
4				(10)	(11)	(12)
5						



너비 우선 탐색

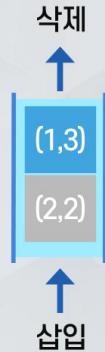
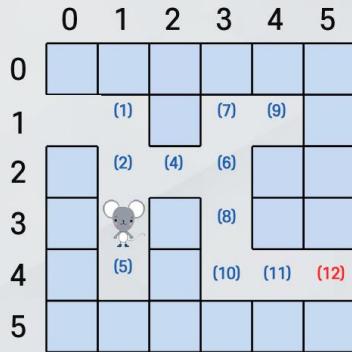
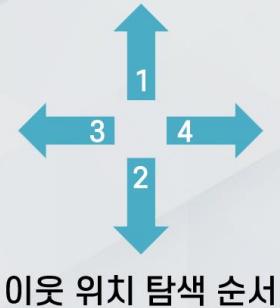
큐

너비 우선 탐색

2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우



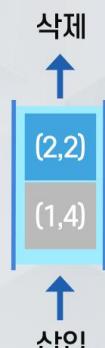
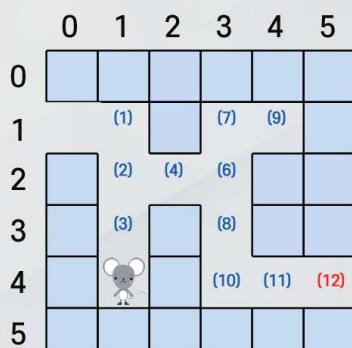
큐

너비 우선 탐색

2 미로의 출구 찾기

◎ 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우



너비 우선 탐색

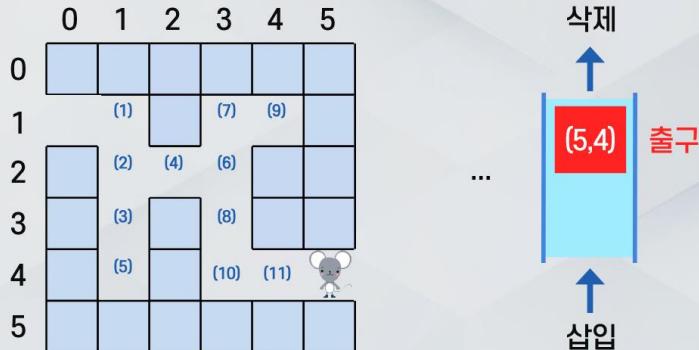
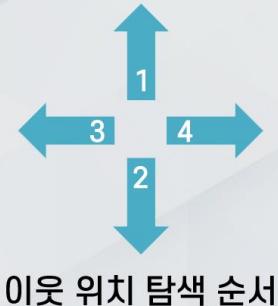
큐

너비 우선 탐색

2 미로의 출구 찾기

● 너비 우선 탐색

예 미로의 입구(0, 1)에서 출구(5, 4)를 찾는 경우



큐

너비 우선 탐색

2 미로의 출구 찾기

● 너비 우선 탐색 알고리즘

```
def BFS() :
    q = CircularQueue(100)
    q.enqueue((0, 1))
    print('BFS: ')
    while not q.isEmpty():
        here = q.dequeue()
        print(here, end='->')
        x = here[0];
        y = here[1];
        if (map[y][x] == 'x') : return True
        else :
            map[y][x] = '..'
            if isValidPos(x, y - 1) : q.enqueue((x, y - 1));
            if isValidPos(x, y + 1) : q.enqueue((x, y + 1));
            if isValidPos(x - 1, y) : q.enqueue((x - 1, y));
            if isValidPos(x + 1, y) : q.enqueue((x + 1, y));
    return False
```

깊이 우선 탐색 (스택)

- 스택 객체 생성
- 초기 위치를 push()로 스택에 삽입

너비 우선 탐색 (큐)

- 큐 생성 객체
- 초기 위치를 enqueue()로 큐에 삽입

너비 우선 탐색

The screenshot shows the PyCharm IDE interface with the following details:

- File Menu:** 파일(F), 편집(E), 보기(V), 프로젝트(P), 디버그(D), 테스팅(S), 봄(L), 도구(T), 허깅(X), 장(W), 도움말(H), 검색(Ctrl+Q).
- Toolbars:** 구성, 업데이트.
- Status Bar:** 파일 19, 문자 21, 툴팁.
- Code Editor:** The main window displays the `MazeQueue.py` file, which contains the following code:

```
from QueueCircular import CircularQueue

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 BFS():
    q = CircularQueue(100)
    q.enqueue((0,1))
    print('BFS: ')
    while not q.isEmpty():
        here = q.dequeue();
        print(here, end=' ->')
        x = here[0];
        y = here[1];
        if (map[y][x] == 'x') : return True
        else :
            map[y][x] = '.'

MAZE_SIZE = 10
map = [
    "0000000000",
    "0000000000",
    "0000000000",
    "0000000000",
    "0000000000",
    "0000000000",
    "0000000000",
    "0000000000",
    "0000000000",
    "0000000000"
]
```

The code implements a Breadth-First Search (BFS) algorithm to solve a maze. It uses a circular queue to store the coordinates of the maze cells to be visited. The algorithm starts at cell (0,1) and explores all adjacent cells (0,0), (0,2), (1,0), and (1,2). If it finds a 'x' (the goal), it returns True. Otherwise, it marks the cell as visited ('.') and continues the search.

실습 단계

스택 대신 → 큐 생성

`push()` 대신 → `enqueue()` 사용

`pop()` 대신 \rightarrow `dequeue()` 사용

큐를 사용하기 위해 QueueCircular.py를 import함

테스트 코드

실행 결과 확인