알고리즘_11~12일차 (2/15~2/16)

목차

큐 (Queue)

0. 큐 (Queue)

0-1. 큐 (Queue)

> 스택과 마찬가지로 삽입과 삭제의 위치가 제한적인 자료구조
> 큐의 뒤에서는 삽입, 앞에서는 삭제만 이루어지는 구조

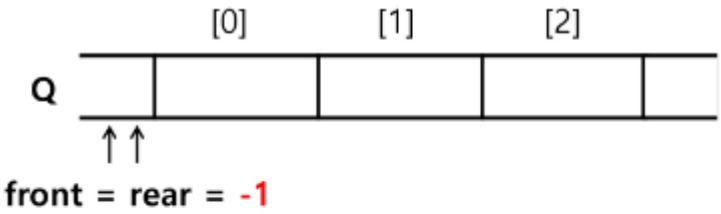
> 선입선출구조 (FIFO : First In First Out)
> 큐에 삽입한 순서대로 원소가 저장되어, 가장 먼저 삽입된 원소는 가장 먼저 삭제된다.

큐의 기본연산

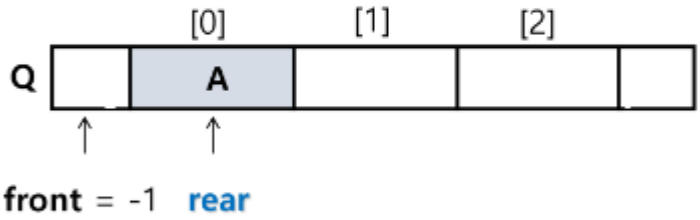
> 삽입 : enQueue
> 삭제 : deQueue

연산	기능
enQueue(item)	큐의 뒤쪽(rear 다음)에 원소를 삽입하는 연산
deQueue( )	큐의 앞쪽(front)에서 원소를 삭제하고 반환하는 연산
createQueue( )	공백상태의 큐를 생성하는 연산
isEmpty( )	큐가 공백상태인지 확인하는 연산
isFull( )	큐가 포화상태인지 확인하는 연산
Qpeek( )	큐의 앞쪽(front)에서 원소를 삭제 없이 반환하는 연산

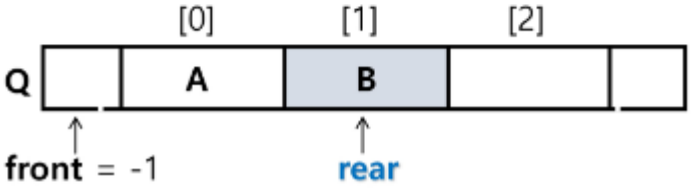
1) 공백 큐 생성 : createQueue();



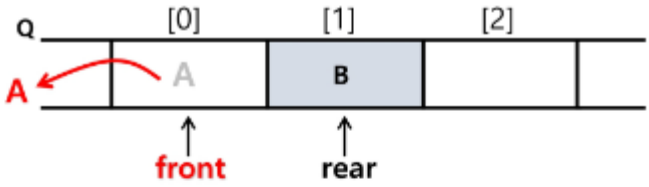
2) 원소 A 삽입 : enQueue(A);



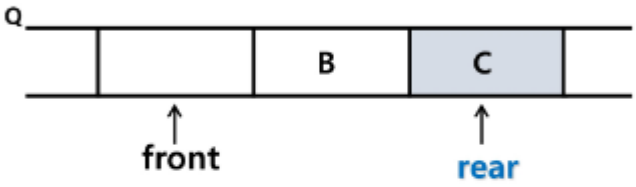
3) 원소 B 삽입 : enQueue(B);



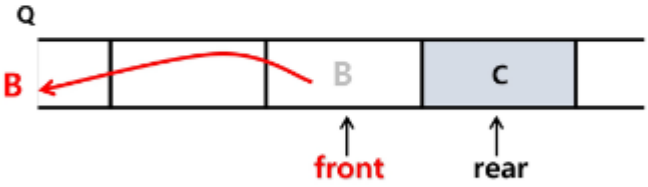
4) 원소 반환/삭제 : deQueue();



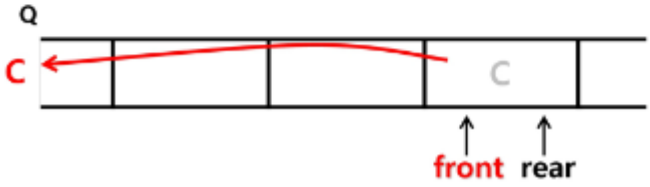
5) 원소 C 삽입 : enQueue(C);



6) 원소 반환/삭제 : deQueue();



7) 원소 반환/삭제 : deQueue();



---

1. 선형큐

> 1차원 배열을 이용한 큐
> 큐의 크기 = 배열의 크기
> front : 저장된 첫 번째 원소의 인덱스
> rear : 저장된 마지막 원소의 인덱스
> 초기 상태 : front = rear = -1
> 공백 상태 : front == rear
> 포화 상태 : rear == n-1
> (n: 배열의 크기, n-1: 배열의 마지막 인덱스)

1-1. 큐의 구현

초기 공백 큐 생성

> 크기 n인 1차원 배열 생성
> front와 rear를 -1로 초기화

삽입 : enQueue(item)

> 마지막 원소 뒤에 새로운 원소를 삽입하기 위해
> rear 값을 하나 증가시켜 새로운 원소를 삽입할 자리를 마련
> 그 인덱스에 해당하는 배열원소 Q[rear]에 item을 저장

```
def enQueue(item):
    global rear
    if isFull():
        print("Queue_Full")
    else:
        rear == rear + 1
        Q[rear] == item
```

삭제 : deQueue()

> 가장 앞에 있는 원소를 삭제하기 위해
> front 값을 하나 증가시켜 큐에 남아있는 첫 번째 원소로 이동
> 새로운 첫 번째 원소를 리턴함으로써 삭제와 동일한 기능

```
deQueue()
    if(isEmpty()) them Queue_Empty()
    else{
        front <- front + 1
        return Q[front]
    }
```

공백상태 및 포화상태 검사 : isEmpty(), isFull()

> 공백상태 : front == rear
> 포화상태 : rear == n-1
> (n: 배열의 크기, n-1: 배열의 마지막 인덱스)

```
def isEmpty():
    return front == rear

def isFull():
    return rear == len(Q) - 1
```

검색 : Qpeek()

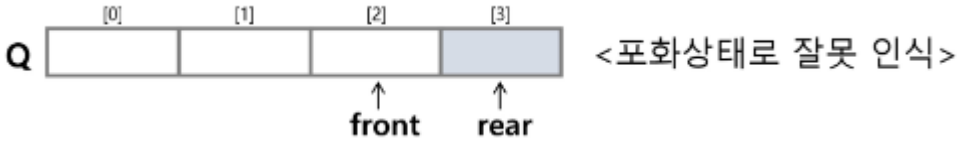
> 가장 앞에 있는 원소를 검색하여 반환하는 연산
> 현재 front의 한자리 뒤(front+1)에 있는 원소, 즉 큐의 첫 번째에 있는 원소를 반환

```
def Qpeek():
    if isEmpty():
        print("Queue_Empty")
    else:
        return Q[front+1]
```

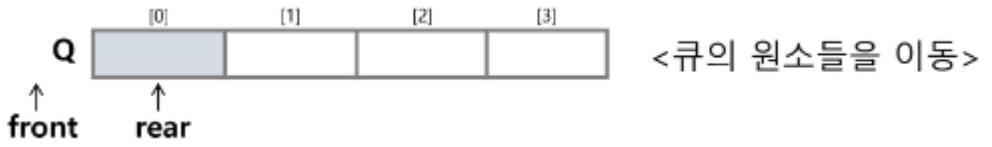
1-2. 선형큐 이용시 문제점

잘못된 포화상태 인식

> 선형큐를 이용하여 원소의 삽입과 삭제를 계속할 경우, 배열의 앞부분에 활용할 수 있는 공간이 있음에도 불구하고 rear = n-1인 상태, 포화상태로 인식하여 더이상의 삽입을 수행하지 않게 됨



> 해결방법1
> 매 연산이 이루어질 때마다 저장된 원소들을 배열의 앞부분으로 모두 이동시킴
> -> 원소 이동에 많은 시간이 소요되어 큐의 효율성 급감



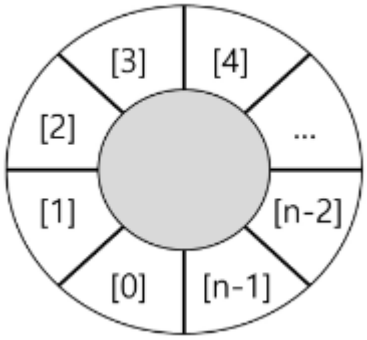
> 해결방법2
> 1차원 배열을 사용하되, 논리적으로는 배열의 처음과 끝이 연결된 원형형태의 큐를 이룬다고 가정하고 사용
> -> 원형큐

---

2. 원형큐

2-1. 원형큐의 구조

> 1차원 배열을 사용하되, 논리적으로는 배열의 처음과 끝이 연결된 원형형태의 큐를 이룬다고 가정하고 사용



초기 공백 상태

> front = rear = 0

인덱스의 순환

> front와 rear의 위치가 배열의 마지막 인덱스인 n-1을 가리킨 후, 그 다음에는 논리적 순환을 이루어 배열의 처음 인덱스인 0으로 이동해야 함
> 이를 위해 나머지 연산자 mod를 사용

front 변수

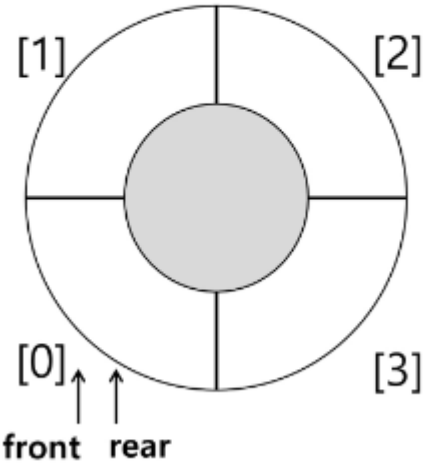
> 공백 상태와 포화 상태의 구분을 쉽게하기 위해 front가 있는 자리는 사용하지 않고 항상 빈자리로 둠

삽입 및 삭제 위치

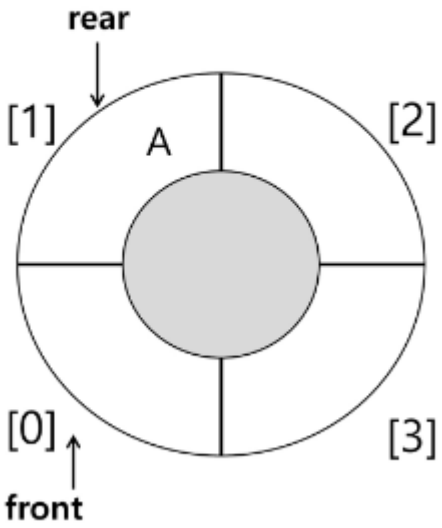
	삽입 위치	삭제 위치
선형큐	rear = rear + 1	front = front + 1
원형큐	rear = (rear+1) mod n	front = (front+1) mod n

2-2. 원형큐의 연산과정

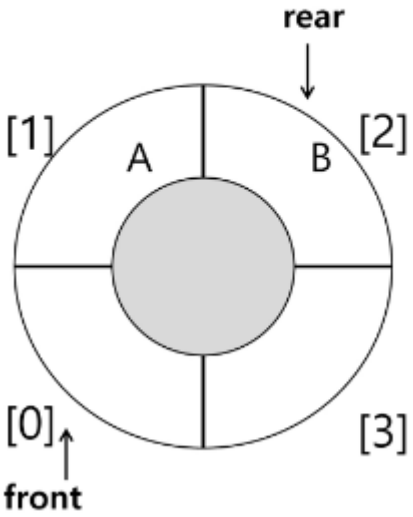
1) create Queue



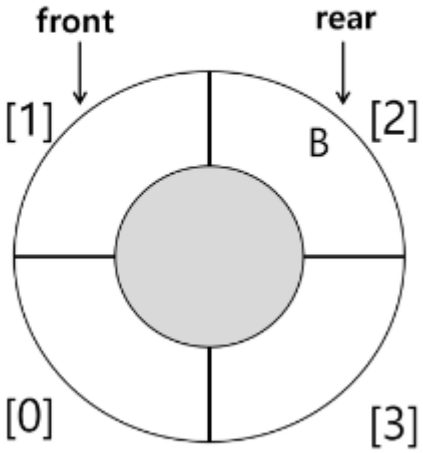
2) enQueue(A);



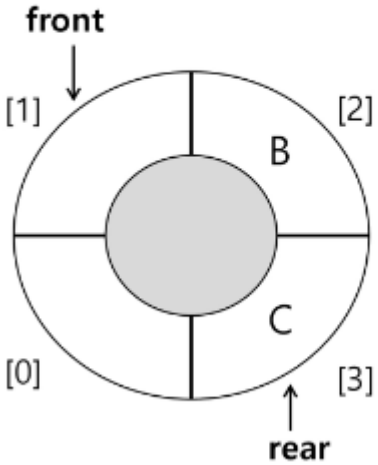
3) enQueue(B);



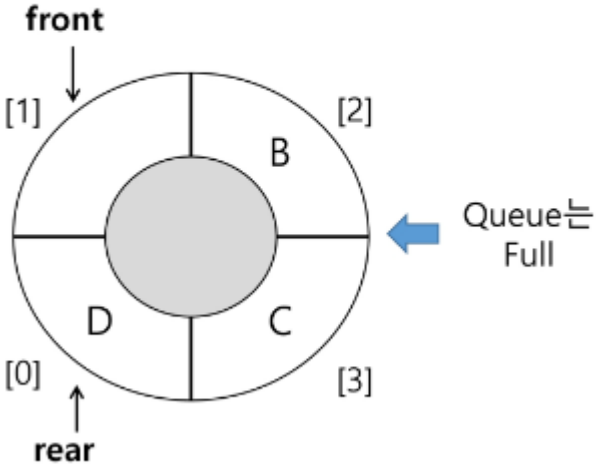
4) deQueue();



5) enQueue(C);



6) enQueue(D);



2-3. 원형큐의 구현

초기 공백 큐 생성

> 크기 n인 1차원 배열 생성
> front와 rear를 0으로 초기화

공백상태 및 포화상태 검사 : isEmpty(), isFull()

> 공백상태 : front == rear
> 포화상태 : 삽입할 rear의 다음 위치 == 현재 front
> (rear+1) mod n == front

```
def isEmpty():
    return front == rear

def isFull():
    return (rear+1) % len(cQ) == front
```

삽입 : enQueue(item)

> 마지막 원소 뒤에 새로운 원소를 삽입하기 위해
> rear값을 조정하여 새로운 원소를 삽입할 자리를 마련 : rear <- (rear+1) mod n
> 그 인덱스에 해당하는 배열원소 cQ[rear]에 item을 저장

```
def enQueue(item):
    global rear
    if isFull():
        print("Queue_Full")
    else:
        rear = (rear+1) % len(cQ)
        cQ[rear] = item
```

삭제 : deQueue()

> 가장 앞에 있는 원소를 삭제하기 위해
> front 값을 조정하여 삭제할 자리를 준비
> 새로운 front 원소를 리턴함으로써 삭제와 동일한 기능

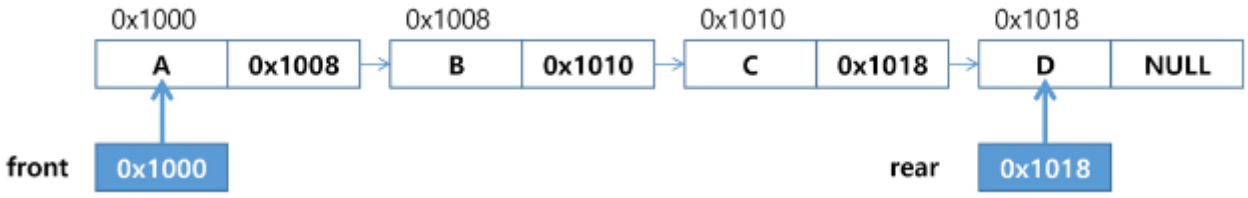
```
def deQueue():
    global front
    if isEmpty():
        print("Queue_Empty")
    else:
        front = (front+1) % len(cQ)
        return cQ[front]
```

3. 연결큐

> 단순 연결 리스트(Linked List)를 이용한 큐

3-1. 연결큐의 구조

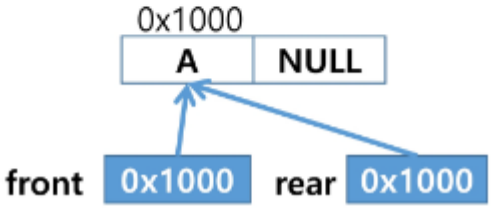
> 큐의 원소 : 단순 연결 리스트의 노드
> 큐의 원소 순서 : 노드의 연결 순서 (링크로 연결되어 있음)
> front : 첫 번째 노드를 가리키는 링크
> rear : 마지막 노드를 가리키는 링크
> 초기 상태 : front = rear = null 공백 상태 : front = rear = null



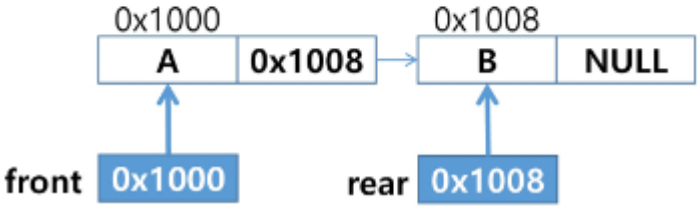
3-2. 연결큐의 연산 과정

1) 공백 큐 생성 : createLinkedQueue();

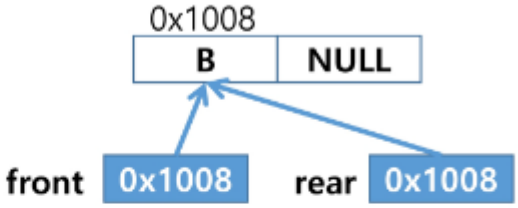
2) 원소 A 삽입 : enQueue(A);



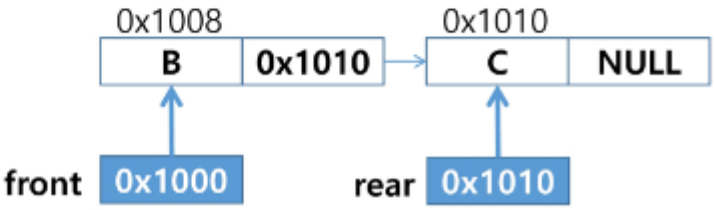
3) 원소 B 삽입 : enQueue(B);



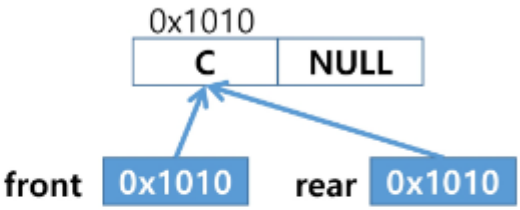
4) 원소 삭제 : deQueue();



5) 원소 C 삽입 : enQueue(C);



6) 원소 삭제 : deQueue();



7) 원소 삭제 : deQueue();

[참고] deque(덱)

> 컨테이너 자료형 중 하나
> deque 객체 : 양쪽 끝에서 빠르게 추가와 삭제를 할 수 있는 리스트류 컨테이너

연산

> append(x) : 오른쪽에 x 추가
> popleft() : 왼쪽에서 요소 제거 후 반환
> (요소가 없으면 IndexError)

```
from collections import deque
q = deque()
q.append(1)     # enqueue()
t = q.popleft() # dequeue()
```

---

4. 우선순위 큐 (Priority Queue)

특성

> 우선순위를 가진 항목들을 저장하는 큐
> FIFO 순서가 아니라 우선순위가 높은 순서대로 먼저 나간다.

적용 분야

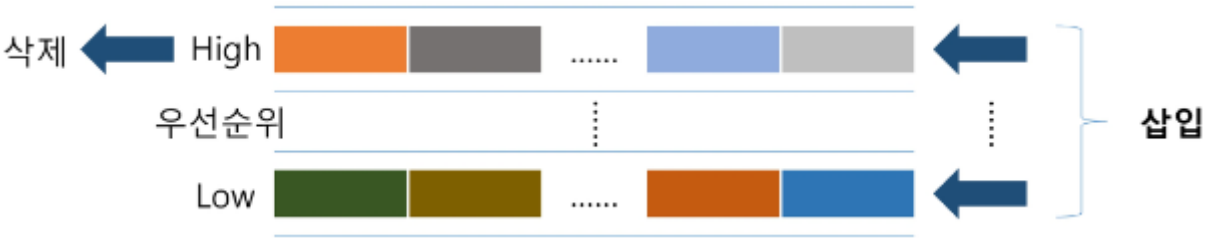
> 시뮬레이션 시스템, 네트워크 트래픽 제어, 운영체제의 테스크 스케줄링

우선순위 큐의 구현

> 배열을 이용한 우선순위 큐
> 리스트를 이용한 우선순위 큐

우선순위 큐의 기본 연산

> 삽입 : enQueue
> 삭제 : deQueue



배열을 이용한 우선순위 큐 구현

> 배열을 이용하여 자료 저장
> 원소를 삽입하는 과정에서 우선순위를 비교하여 적절한 위치에 삽입하는 구조

> 가장 앞에 최고 우선순위의 원소가 위치하게 됨

문제점

> 배열을 사용하므로, 삽입 및 삭제 연산이 일어날 때 원소의 재배치가 발생함
> -> 이에 소요되는 시간 및 메모리 낭비가 큼

---

5. 큐의 활용 : 버퍼 (Buffer)

> 데이터를 한 곳에서 다른 한 곳으로 전송하는 동안 일시적으로 그 데이터를 보관하는 메모리의 영역
> 버퍼링 : 버퍼를 활용하는 방식 또는 버퍼를 채우는 동작

버퍼의 자료 구조

> 버퍼는 일반적으로 입출력 및 네트워크와 관련된 기능에서 이용된다.
> 순서대로 입력-출력-전달되어야 하므로 FIFO 방식의 자료구조인 큐가 활용된다.

---

6. BFS (Breadth First Search)

> 너비 우선 탐색 (BFS)
> 그래프를 탐색하는 방법 중 하나
> (나머지는 깊이 우선 탐색(DFS))

6-1. BFS의 구조

> 탐색 시작점의 인접한 정점들을 먼저 모두 차례로 방문한 후에, 방문했던 정점을 시작점으로 하여 다시 인접한 정점들을 방문하는 방식

> 인접한 정점들에 대해 탐색을 한 후, 차례로 다시 너비우선탐색을 진행해야 하므로 선입선출 형태의 자료구조인 큐를 활용

6-2. BFS 알고리즘

```
# 그래프 G, 탐색 시작점 V
def bfs(G, v):
    visited = [0] * (n+1)           # n : 정점의 개수
    queue = []                      # 큐 생성
    queue.append(v)                 # 시작점 v를 큐에 삽입
    while queue:                    # 큐가 비어있지 않은 경우
        t = queue.pop(0)            # 큐의 첫번째 원소 반환
        if not visited[t]:          # 방문되지 않은 곳이면
            visited[t] = True       # 방문으로 표시
            visit(t)                # 정점 t에서 할 일
            for i in G[t]:          # t와 연결된 모든 정점에 대해
                if not visited[i]:  # 방문되지 않은 곳이라면
                    queue.append(i) # 큐에 넣기
```

BFS 예제

```
# 그래프 G, 탐색 시작점 V
def bfs(G, v, n):
    visited = [0] * (n+1)       # n : 정점의 개수
    queue = []                  # 큐 생성
    queue.append(v)             # 시작점 v를 큐에 삽입
    visited[v] = 1
    while queue:                # 큐가 비어있지 않은 경우
        t = queue.pop(0)        # 큐의 첫번째 원소 반환
        visit(t)
        for i in G[t]:          # t와 연결된 모든 정점에 대해
            if not visited[i]:  # 방문되지 않은 곳이라면
                queue.append(i) # 큐에 넣기
                visited[i] = visited[t] + 1
                # n으로부터 1만큼 이동
```