# Queue

≔ 태그



## 큐

- 스택 처럼, 삽입과 삭제의 위치가 제한적이다.
    - 스택과 반대로 뒤에서는 삽입, 앞에서는 삭제만 이루어진다

  ⇒ 선입선출의 구조다(FIFO) ⇒ 점심시간의 배식과 같다.

- Front, Rear

### 큐의 선입선출 구조


### 큐의 기본 연산

- 삽입 : enQueue
- 삭제 : deQueue

- 연산 과정

1. 공백 큐 생성 : createQueue()
2. 원소 A 삽입 : enQueue(A) ⇒ `rear += 1 , Q[rear] = A`
3. 원소 반환 / 삭제 : deQueue() ⇒ `front += 1, Q[front]`
4. 새로운 원소 C 삽입 : enQueue(C) ⇒ `rear += 1, Q[rear] = C`
5. A 삭제 ⇒ `front += 1, Q[front]`
6. C도 삭제 ⇒ `front += 1, Q[front]`
7. front와 rear가 같아진다 ⇒ 큐가 비어있는 상태가 된다. ⇒ 큐가 비어있는지 확인하려면, front와 rear가 비어있는지를 확인하면 된다.

## 선형큐

- 1차원 배열을 이용한 큐
  - 큐의 크기 = 배열의 크기
  - front : 마지막으로 삭제된 인덱스
  - rear : 저장된 마지막 원소의 인덱스
  - 초기 상태 : front = rear = -1 / 공백 상태 : front == rear / rear == n-1

- 구현
  - 큐 생성 : 크기 n인 1차원 배열, front = rear = -1
  - enQueue(item) ⇒ rear += 1, Q[rear] = item
  - deQueue(item) ⇒ front += 1, return Q[front]
  - isEmpty() ⇒ return front == rear (공백 확인)
  - isFull() ⇒ return rear == len(Q) - 1 (포화 확인)
  - Qpeek() ⇒ 가장 앞의 원소를 검색해서 반환 return Q[front + 1]
    (다음번에 꺼내는 값을 확인하는 것)

- 메서드로 구현

```
queue = []
queue.append(1)
queue.append(2)
queue.append(3)
# print(queue.pop(0), queue)
while queue:
    print(queue.pop(0))
```

- 코드로 구현

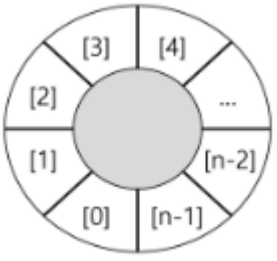
```
# 큐 생성
N = 10
q = [0] * 10
front = rear = -1

rear += 1 # euqueue(1)
q[rear] = 1
rear += 1 # euqueue(2)
```

```
q[rear] = 2
rear += 1 # euqueue(3)
q[rear] = 3
while front != rear: # 큐가 비어있지 않으면 # dequeue()
     front += 1
     print(q[front])
```

## 원형큐

- 선형 큐의 문제점
  - 잘못된 포화상태 인식 ⇒ 배열의 앞부분에 활용할 공간이 있어도, rear = n-1 즉 포화상태로 인식하여 더이상의 삽입을 수행하지 않게 된다.
  - 해결방법

  1. 저장된 원소들을 앞부분으로 모두 이동시키는 방법 → 그러면 효율성이 너무 떨어진다.
  2. 1차원 배열을 사용하되, 논리적으로 배열의 처음과 끝이 연결시켜 원형을 이룬다고 가정




- 구조
  - 초기 공백 상태 : front = rear = 0
  - index의 순환 : front와 rear의 위치가 배열의 마지막 인덱스인 n-1을 가리키고, 논리적 순환을 이루어 배열의 처음 인덱스인 0으로 이동해야 한다.
  - 나머지 연산자 mod를 사용한다.
  - front가 있는 자리는 사용하지 않고 항상 빈자리

- 삽입과 삭제 : mod(%) 사용 ⇒ 앞쪽으로 인덱스를 되돌리게 되는 것이다.

### 삽입 위치 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 |

### 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
```

- 삽입과 삭제

### 삽입 : enQueue(item)

- 마지막 원소 뒤에 새로운 원소를 삽입하기 위해

1) rear 값을 조정하여 새로운 원소를 삽입할 자리를 마련함 :
   rear <- (rear + 1) mod n;

2) 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(), delete()

- 가장 앞에 있는 원소를 삭제하기 위해

1) front 값을 조정하여 삭제할 자리를 준비함

2) 새로운 front 원소를 리턴 함으로써 삭제와 동일한 기능함

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

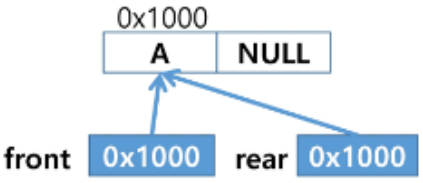
## 연결 큐

- 단순 연결 리스트(linked list)를 이용한 큐
  - 큐의 원소 : 단순 연결 리스트의 노드
  - 원소 순서 : 노드의 연결 순서 ⇒ 링크로 연결되어 있음
  - front : 첫 노드를 가리키는 링크 / rear : 마지막 노드를 가리키는 링크

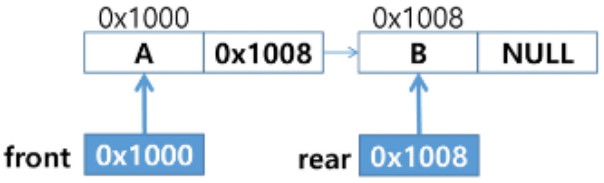
- 상태 표현 ⇒ 초기 상태 : front = rear = null / 공백 상태 : front = rear = null

1) 공백 큐 생성 : createLinkedQueue();

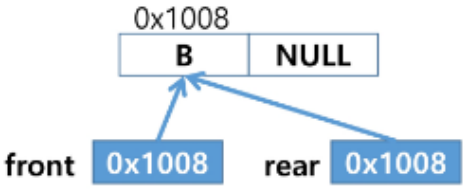

2) 원소 A 삽입 : enQueue(A);



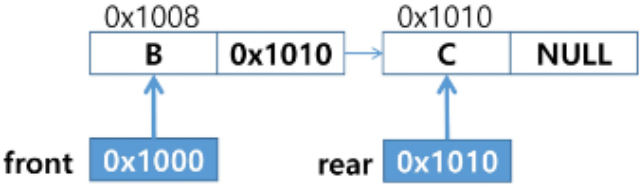

3) 원소 B 삽입 : enQueue(B);



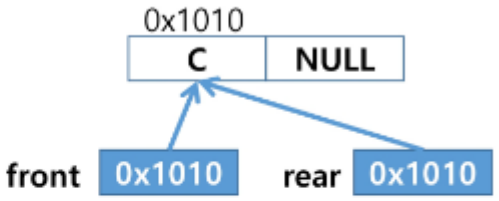

4) 원소 삭제 : deQueue();




5) 원소 C 삽입 : enQueue(C);



6) 원소 삭제 : deQueue();




7) 원소 삭제 : deQueue();


## 덱

- 양쪽 끝에서 빠르게 추가와 삭제를 할 수 있는 리스트류 컨테이너
- 연산
  - append(x) : 오른쪽에 x 추가
  - popleft() : 왼쪽에서 요소를 제거하고 반환, 요소가 없으면 IndexError

```
from collections import deque

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

- 코드

```
from collections import deque

q = deque()
```

```
q.append(1)
q.append(2)
print(q.popleft())
print(q.popleft())
```

- deque와 리스트의 비교

```
from collections import deque

q = []
for i in range(100000):
    q.append(i)
print('append')
while q:
    q.pop(0)
print('end')

dq = deque()
for i in range(1000000):
    dq.append(i)
print('append')
while dq:
    dq.popleft()
print('end')

## deque 가 훨씬 빠르다
```

## 우선순위 큐

- 우선순위를 가진 항목들을 저장하는 큐다.
- FIFO의 순서가 아니라, 우선순위가 높은 순서대로 먼저 나가게 된다.

⇒ 적용 분야 : 시뮬레이션 시스템, 네트워크 트래픽 제어, 운영체제의 테스크 스케줄링

- 구현 : 배열 or 리스트를 이용
- 가장 앞에 최고 우선순위 원소가 위치하게 된다. ⇒ 삽입, 삭제가 발생할 때, 원소 재배치 발생
    - 소요되는 시간이나, 메모리 낭비가 크다

# 큐의 활용

## 버퍼

- 데이터를 한 곳에서 다른 한 곳으로 전송하는 동안, 일시적으로 그 데이터를 보관하는 영역
- 버퍼를 활용하는 방식 or 버퍼를 채우는 동작을 의미한다.

- 버퍼의 자료 구조
    - 입출력 및, 네트워크와 관련된 기능에서 이용된다.
    - 입력/출력/전달 되어야 하므로 FIFO 방식의 자료구조인 큐가 활용된다.

# 오프라인

## 클래스를 이용한 큐의 구현

```
class Queue:
    def __init__(self,maxsize):
        self.size = maxsize
        self.items = [None] * maxsize
        self.rear = self.size
        self.front = 0
```

```
    def enQueue(self, item):

        if self.isFull():
            print('Queue is Full!!')
        else:
            self.rear = (self.rear + 1) % self.size
            self.items[self.rear] = item

    def deQueue(self):
        self.front += 1
        return self.items[self.front]

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

q = Queue(3)
q.enQueue('a')
q.enQueue('b')
q.enQueue('c')
q.enQueue('d')
print(q.items)
print(q.deQueue())
print(q.deQueue())
print(q.deQueue())
print(q.items)
```

## heapq

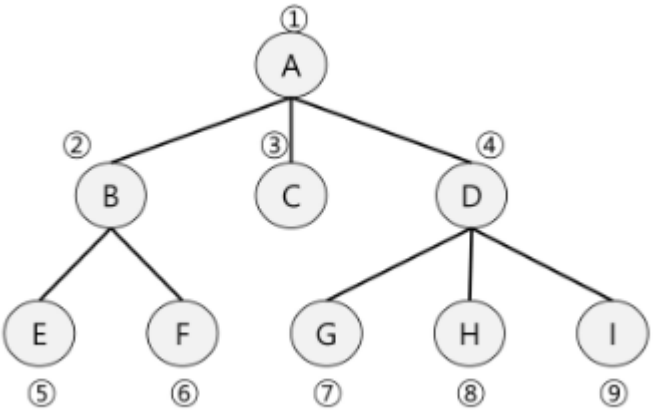
```
import heapq

# q = PriorityQueue()
# q.put((45, 'z'))
# q.put((17, 'x'))
# print(q.queue)
```

```
arr = [(45, 'z'), (17, 'x'), (6, 'a'), (100,'b')]
heapq.heapify(arr)
print(arr)
'''
        6
    17  45
100
완전 이진 트리
'''
heapq.heappush(arr, (4,'t'))
print(arr)
heapq.heappush(arr, (30,'f'))
print(arr)
print(heapq.heappop(arr))
print(arr)

[(6, 'a'), (17, 'x'), (45, 'z'), (100, 'b')]
[(4, 't'), (6, 'a'), (45, 'z'), (100, 'b'), (17, 'x')]
[(4, 't'), (6, 'a'), (30, 'f'), (100, 'b'), (17, 'x'), (45, '
(4, 't')
[(6, 'a'), (17, 'x'), (30, 'f'), (100, 'b'), (45, 'z')]
```

# BFS

- 깊이 우선 탐색(DFS) 와 반대로 너비 우선 탐색이다.
  - 인접한 정점들을 먼저 모두 차례로 방문한 후에, 정점을 시작점으로, 다시 인접한 정점 방문
  - 차례로 너비우선탐색 진행 ⇒ 선입선출 형태의 자료구조인 큐를 활용해야 한다.

- 탐색 순서




A - B - C - D - E - F - G - H - I 순으로 간다.

- 슈도코드

```
def BFS(G,v):
    visited = [0] * (n+1) # 정점의 개수
    queue   = [] # 큐 생성
    queue.append(v) # 시작점 v를 큐에 삽입
    while queue:
        t = queue.pop(0)
        if not visited[t]:
            visited[t] = True
            visit(t)
            for i in G[t]:
                if not visited[i]:
                    queue.append(i)
```

- visited를 나중에 칠하기

```
def BFS(G, v, n): # 그래프 G, 시작점 v
    visited = [0] * (n+1)
```

```
    queue   = []
    queue.append(v)
    visited[v] = 1
    while queue:
        t = queue.pop(0)
        visited(t)
        for i in G[t]:
            if not visited[i]:
                queue.append(i)
                visited[i] = visited[t] + 1
```

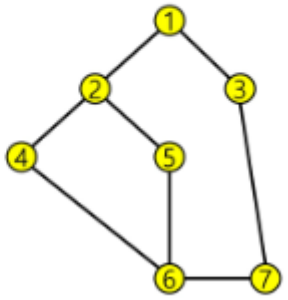
- 실습 문제

다음은 연결되어 있는 두 개의 정점 사이의 간선을 순서대로 나열 해 놓은 것이다. 모든 정점을 너비우선탐색 하여 경로를 출력하시오. 시작 정점을 1로 시작하시오.

- 1, 2, 1, 3, 2, 4, 2, 5, 4, 6, 5, 6, 6, 7, 3, 7

- 출력 결과의 예는 다음과 같다.

1 – 2 – 3 – 4 – 5 – 7 – 6




- 코드

```
import sys
sys.stdin = open('input.txt')

'''
V E : V 1~V번까지 V개의 정점, E개의 간선
E개의 간선정보
7 8
```

```
1 2 1 3 2 4 2 5 4 6 5 6 6 7 3 7
'''
def bfs(s, N): #시작 정점s, 노드개수 N
    q = [] # 큐
    visited = [0] * (N+1) # visited
    q.append(s)           # 시작점 인큐
    visited[s] = 1        # 시작점 방문표시
    while q: #큐가 비워질때까지...(남은 정점이 있으면)
        t = q.pop(0)
        # t에서 할일...
        print(t)
        for i in adjl[t]: # t에 인접인 정점 i
            if visited[i] == 0: # 방문한적 없으면
                q.append(i) # 인큐
                visited[i] = 1 + visited[t] # 방문표시 (하나 더
    print(visited)

V, E = map(int, input().split())
arr = list(map(int, input().split()))

# 인접리스트라는 형태로 저장
adjl = [[] for _ in range(V+1)]
for i in range(E):
    n1, n2 = arr[i*2], arr[i*2+1]
    adjl[n1].append(n2)
    adjl[n2].append(n1) # 무향 그래프

bfs(1, V)
```