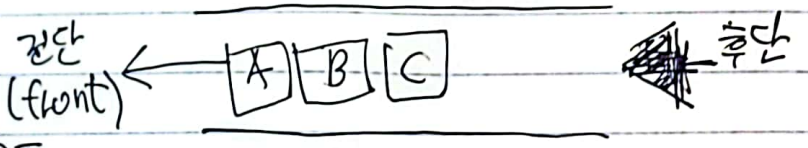
5. 큐와 덱

- 큐
· 삽입과 삭제가 양끝에서 각각 수행되는 자료구조
· 선입선출의 자료구조
✓먼저들어온 데이터가 먼저 나가는 구조
· 새로운 데이터가 추가되고 앞에서 데이터가 하나씩 삭제되는 구조
· 삽입이 일어난 곳을 후단이라 하고 삭제이 일어난 곳을 전단이라고 한다



큐 ADT
데이터: 선입선출 (FIFO)의 접근 방법을 유지하는 항목들의 모음
isEmpty(): 큐가 비어있으면 True 아니면 False를 반환한다
enqueue(x): 항목 x를 큐의 맨뒤에 추가
dequeue(): 큐의 맨앞에 있는 항목을 꺼내 반환한다
큐의 연산
· 삽입연산 enqueue: 삽입은 후단을 통해서만 가능
· 삭제연산 dequeue: 삭제는 전단을 통해서만 가능

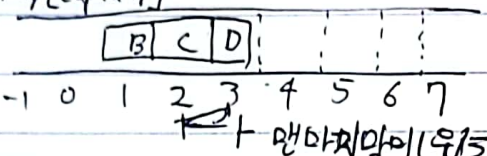
2가지오류
1. 오버플로 오류
2. 언더플로 오류

큐의사용
· 임시저장장치로 큐가 사용 이를 버퍼라고함
컴퓨터와 프린터의 차이 : 프린터는 CPU에 비해 상대적으로 속도가느림
✓프린터는 일정한속도로 인쇄작업큐에서 순서대로 데이터를가져와 인쇄함
실시간 비디오 스트리밍 : 다운로드된 데이터가 비디오를 재생하기에 충분하지 않을때 사용한다
✓큐에 순서대로 모아두었다가 충분한 양이되었을때 비디오를 복원해 재생하는데 이를 버퍼링이라함

배열을 이용한 큐의 구조

· 용량이 고정된 큐

요소저장 배열 array[capacity] 용량



front : 전단 요소 바로 앞에 위치 (인덱스)

rear : 맨 마지막 요소의 위치 (인덱스)

선형큐의 문제

· 많은 이동이 필요한 경우 발생

· 전단회전 : front ← (front+1) % capacity

· 후단회전 : rear ← (rear+1) % capacity

- 원형큐

- 선형큐의 비효율성 극복

. 리스트의 항상 고정된 크기 (MAX_SIZE) 가 있다고 생각

rear : 큐에 가장 최근의 삽입된 항목의 위치를 저장

front : 가장 최근에 삭제된 항목의 위치를 저장

원형큐 공백과 포화 (isEmpty() vs isFull())

공백상태 : front == rear

포화상태 : front == (rear+1) % ~~MAX_QSize~~ capacity

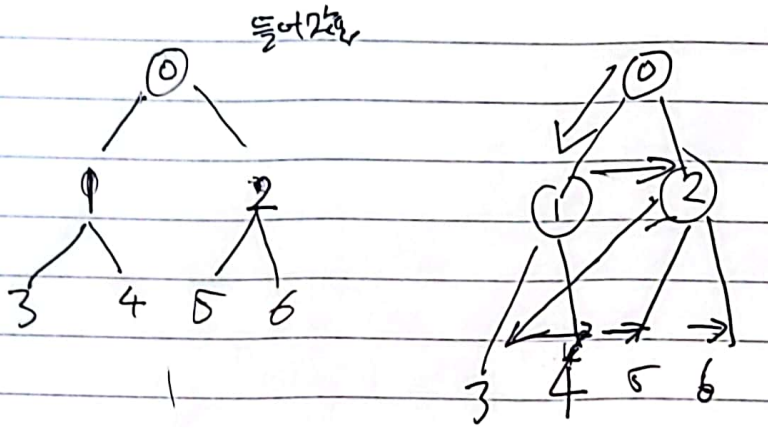
원형큐 구현 예제 1대P 코드 5.4

```
def enqueue(self, item):
    if not self.isFull():
        self.rear = (self.rear+1) % Max_QSize # rear 회전
        self.items[self.rear] = item # rear 위치에 삽입
def dequeue(self, item)
    self.front = (self.front+1) % self.capacity
    return self.array[self.front]
else: pass
```

큐의 응용

- 이진트리의 레벨 순회
- 기수정렬에서 버킷 레귤레이터 정렬을 위해 사용
- 그래프 탐색에서 너비 우선 탐색

- DFS VS BFS



파이썬의 queue 모듈

- 큐(Queue)와 스택(LifoQueue) 클래스로 제공
- 사용하기 위해서는 먼저 Queue 모델을 import 해야함
- 삽입(put()) 삭제(get()) 연산존재

```
import queue
Q = queue.Queue(maxsize=20)
for v in range(1,10):
    Q.put(v)
print("큐의내용:",end='')
for _ in range(1,10):
    print(Q.get(),end='')
print()
```

· 덱

· 덱은 Double-Ended Queue의 줄임말

· 스택이나 큐 보다는 입출력이 자유로운 편

· 양쪽 끝에서 삽입과 삭제가 모두 가능한 자료구조

  중간 삽입, 삭제 불가능

덱 ADT

데이터: 전단과 후단을 통한 접근을 허용하는 항목들의 모임

Deque(): 비어있는 새로운 덱을 만든다

isEmpty(): 덱이 비어있으면 True를 아니면 False를 반환한다

addFront(x): 항목 x를 덱의 맨앞에 추가한다

deleteFront(): 맨앞의 항목을

getFront()

addRear(x)

getRear()

덱의 연산

· 스택과 큐의 연산들과 더불어

  Rear과 front 반대방향 회전 연산 추가

Deque(): 비어있는 새로운 덱을 만든다

isEmpty(): 덱이 비어있으면 True 아님 False를 반환한다

addFront(x), addRear(X) 항목의 X를 덱의 맨앞 맨뒤에 추가

스택과 큐의 연산들과 유사함

Rear과 front 반대방향 회전 연산추가

덱의 구현

~~덱의 프로그램~~

- 원형 큐를 상속하여 원형덱 클래스를 구현
- 생성자는 상속되지 않음
- 자식 사용 멤버들 : isEmpty, isFull, size, clear

```
def addFront(self, item):
    if not self.isFull
        self.items[self.front]=item
        self.front = self.front-1
        if self.front<0 : self.front=MaxQueue-1
```

우선순위 큐

- 우선순위를 기준으로 ~~항목~~ 자료구조
- 모든 데이터가 우선순위를 가진다
- 입력 순서와 상관 없이 우선순위가 높은 데이터가 먼저 출력
- 가장 일반적인 큐를 볼 수 있음
- ✓ "우선순위" 정하는 방식에 따라 스택 or 큐로 사용가능

✓ 응용분야

✓ 시뮬레이션, 네트워크 트래픽 제어, OS의 작업 스케줄링 등

우선순위 큐

데이터 : 전단과 후단을 통한 접근을 허용하는 항목들의 모임

정렬되지 않은 리스트

- enqueue() : 대부분의 경우 O(1)
- findMaxIndex() : O(n)
- dequeue(), pick() : O(n)

정렬된 리스트 사용

- enqueue : O(n)
- dequeue, peek() : O(1)

힙트리

- enqueue(), dequeue() : O(log n)
- peek() : O(1)

4441P

정렬되지 않은 배열을 이용한 구현

LBT 5-10

MOP 전략적 미로탐색

- 쥐는 출구의 위치를 알고있다 가정
- 가능한 가까운 방향을 먼저 선택
- 큐에 저장되는 항목: (x, y, -d) 형태의 튜플
  ∨ 거리를 -d로 둔이유: 거리가 가까울수록 더 우선순위가 높아야함

```
def MySmartSearch():
    q=PriorityQueue()
    q.enqueue((0,1,-dist(0,1)))
    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( 생략
            if isValidPos(x,y+1): q.enqueue(
            if isValidPos(x-1,y): q.enqueue(
            if isValidPos(x+1,y): q.enqueue(
        print('우선순위큐:',q.items)
    return False

result=MySmartSearch()
if result: print('성공')
else: print('실패')
```