스택에서의 push vs pop 처럼
큐에서 enqueue vs dequeue

<5장, 큐 (QUEUES), 김성현, 201910783>

5.1 큐 추상 데이터 타입	5.2 선형큐	5.3 원형큐	5.4 큐의 응용 : 버퍼
- 큐(Queue)의 추상 데이터 타입(ADT) └ 스택과 반대 개념으로 접근	- 배열을 이용한 큐의 구현 └ 실제로 잘 쓰이진 않지만 원형 큐 보다 쉬움	- 원형 큐 (circular queue) 구조 └ 배열로 구현하되 반드시 원형/연결로 선언!	

5.5 덱이란? (double ended queue의 줄임말)	5.6 큐의 응용 : 시뮬레이션
- 덱(Deque)의 추상 데이터 타입(ADT) - 배열을 이용한 원형 덱의 구현 └ 배열 그대로 가지고 보다 원형으로 구현	

큐에서 좀 더 확장된 개념

5.1 큐 추상 데이터 타입

□ 큐(Queue)의 추상 데이터 타입(ADT).

p.146 ~ 149.

p.146
<큐 : 선입선출>

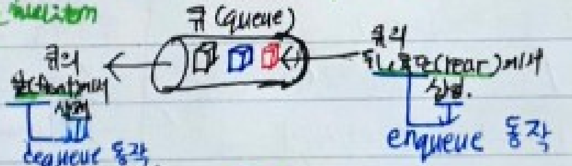
ex) 매표소에서 표를 사기 위해 늘어선 줄.

• 큐(Queue). — 먼저 들어가고 먼저 나간다. 먼저 들어오면 먼저 나간다.

□ 선입선출(FIFO : First-In First-Out)의 특징을 가지는 자료구조.

□ 큐에서 요소(element)의 삽입은 큐의 뒤(후단)(rear)에서 이루어지고,
요소의 삭제는 앞(전단)(front)에서 이루어진다.

[스택은 Top (+bottom)
[큐는 앞 / 뒤



삭제(delete) <FIFO : First-In-First-Out> 삽입(entry)

※ 자료가 입력되는 장소와 (rear) 삭제되는 장소가 다르다. (front)

스택은 Top이라는 변수 하나면 됐지만, 큐는 앞(front), 뒤(rear)라는 2개의 변수가 필요

p.147.

ADT : 객체(object) + 연산(operation)으로 구성된다!!

• 큐 ADT — 추상적이어서 코드 X !
(0개 이상의 요소들로 구성된 선형 리스트)

· 객체 : n개의 element 형으로 구성된 요소들의 순서 있는 모임.

· 연산 :

■ create() ::= 큐를 생성한다. → "생성하면서 초기화한다."

■ init(q) ::= 큐를 초기화한다.

■ is_empty(q) ::= 큐가 비어있는지(empty) 검사한다.

■ is_full(q) ::= 큐가 가득 찼는지(full) 검사한다.

■ enqueue(q, e) ::= 큐의 뒤(rear)에 요소를 삽입한다. — 스택의 push와 유사한 개념

■ dequeue(q) ::= 큐의 앞(front)에 있는 요소를 삭제한 후 반환한다. — 스택의 pop과 유사, 개념

■ peek(q) ::= 큐의 앞(front)에서 요소를 삭제하지 않고 반환한다.
— 꺼내지 않고 한번 살펴 본다.

```
if (size == 0) return TRUE;
else return FALSE;
if (size == max_size) return TRUE;
else return FALSE
```

→ 자료구조가 꽉 찼나, 비었나를 꼭 체크할 것. (컴퓨터의 자원(resources)은 무한하지 않음)
full/empty 체크!! 자료를 넣거나 뺄 때 항상 체크!!!

p.148

• 큐는 스택과 마찬가지로 "프로그래머의 도구"
⇒ 컴퓨터로 이용해 현실세계의 실제상황을 시뮬레이션 (★대기열 대상★) / 운영체제 (컴퓨터 내 작업처리)
→ 대기열, 비행기 등, 데이터 패킷에 사용

5.2, 선형큐

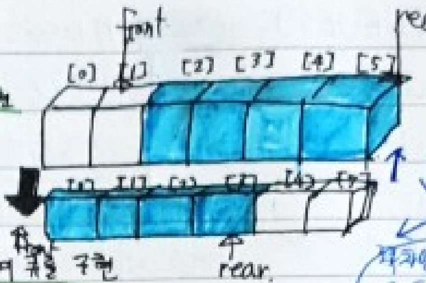
p.149~153 • 배열을 이용한 큐 구현

사용X 소멸함.

□ 선형 큐 (linear queue): 배열을 선형으로 사용하여 큐를 구현
- □ 삽입 시 요소들을 이동시켜야 할 때가 있음
- □ 실용적이지 못함

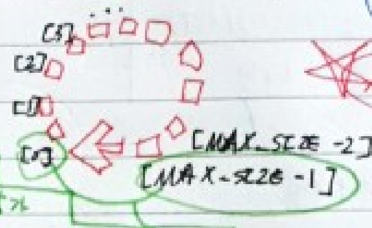
□ 원형 큐 (Circular queue): 배열을 원형으로 사용하여 큐를 구현
- □ 삽입 시 요소들을 이동할 필요가 없음.
- □ 실용적이어서 많이 사용됨.

매우 중요!!



쭉 가면 실제로 다 찬 거 아니어도 element를 줄여야 함

시간 + 효율 떨어짐



"배열의 가장 마지막 element가 배열의 첫 번째 element와 붙어 있다, 연결되어 있다." 라고 설명할 수 있는 것.

실제로 붙어 있는 것이 아닌, 그렇다고 상상하는 것일

실제로 array는 원형 모양이 아니지만, 우리가 상상해 보는 것이다.
n-1 번째 index와 (item) 0번째 index가 (item) 연결되어 있다고!!
< n-1 까지 꽉 차면 다시 연결된 0번째로 가서 비었으면 값을 넣는 형식>

p.150
- front는 항상 삭제하고 싶은 걸 가리키니 (앞의 요소) 있는 element index의 하나 앞에 위치 (빈 공간)
- rear는 뒤의 요소 (새로 item 직접 가리킴)
- front와 rear의 초기값은 -1 (처음엔 비어 있으니까)
- 데이터 추가 → rear + 1 / 2 위치에 데이터 저장 <증가 후 저장>
- 삭제할 때도 → front + 1 / 2 위치 데이터 삭제 <증가 후 삭제>.

(p.150에 선형 큐의 코드가 있으니 궁금하면 계속 보자)

p.152~153 ✱ 선형 큐의 응용: 작업 스케줄링
- 큐는 운영 체제에서도 사용
  - ↳ 많은 작업들 동시 실행 → if (cpu가 하나 & 모든 작업들의 우선순위 X) ⇒ 작업은 운영체제에 들어간 순서대로 처리.

이럴 때 큐를 사용하여 작업 처리

↓ 여기 해결하는 선형 선택.

큐 ⇒ 사용 예시 / 운영체제 (cpu)

⇓ 이러면 운영체제는, 기본적으로 주기적으로 이동을 시켜줘야 되서 시간 + 효율의 문제점이 있다.

⟶ 이것의 해결책은 <원형 큐>!!

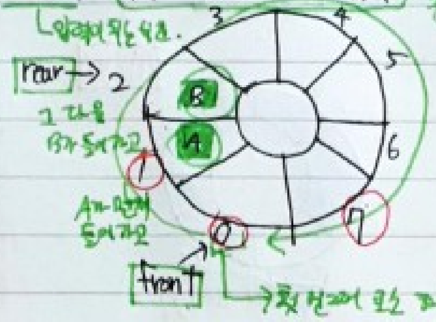
5.3 원형 큐

p.153 ~ 160

• 원형 큐 (Circular queue)의 구조.

* 원형 큐는 선형과 다르게 front, rear이 배열의 첫 index인 0에서 시작!! (합치서 rear+1 => 첫번째 공간이 공백(비어)으로 남아있다)

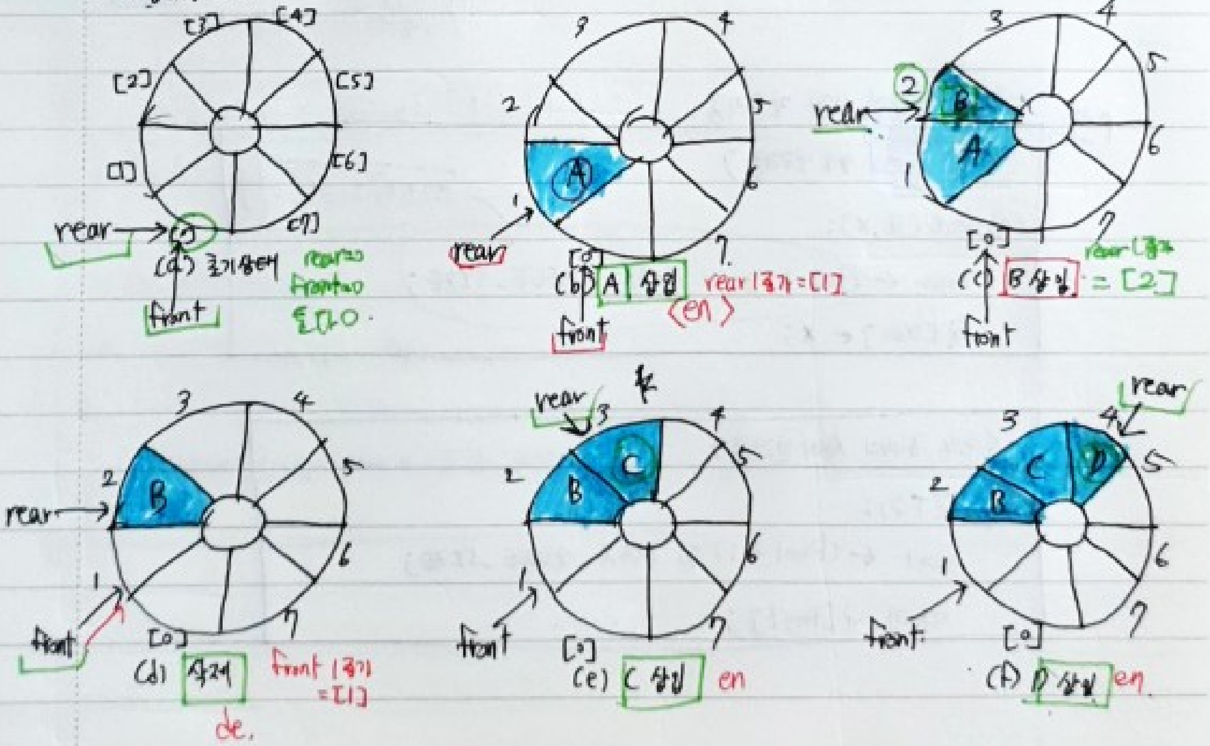
- 큐의 앞과 뒤의 위치를 관리하기 위한 [2개의 변수] 필요.
  - 앞(front) : 큐의 첫번째 요소 하나 (이전의 위치) ☆-1☆ ← 제거가 되는 방향
  - 뒤(rear) : 큐의 마지막 요소의 위치 → 가장 최근에 넣은 data 그 자체 위치 ☆ ← 삽입이 되는 방향.



↳ 왜 이렇게 하냐고? 왜 이게 편하냐고?
(이전에 있어야지 이게 시작점, 연결되는걸 알 수 있으므로 넣어가서 삭제할 수 있으니까!)
+ 포화 / 공백 상태 구분하기 위해

☆ <공백 상태 표시를 다른 식으로 하는 경우도 있다> ☆

• 원형 큐의 동작.



p.int ☆ [삽입을 하면 rear 1증가 / 삭제를 하면 front 1증가] => ☆ rear, front 모두 '증가'를 한다!!! ☆ (끝까지 가면 다시 0으로 가고)

<스택과는 다르다!!>

en → rear 변화(1증가), 전체 element 1증가

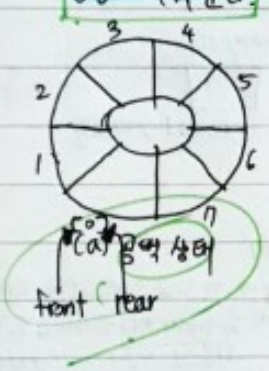
de → front 변화(1증가), 전체 element 1감소

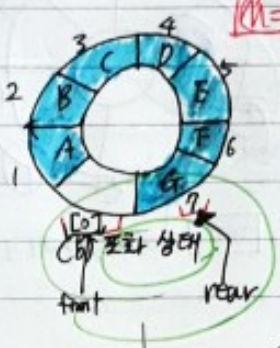
둘 다 (증가)!!

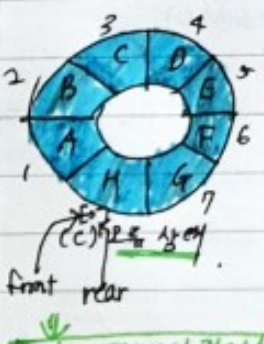
반드시 헷갈리지마!!

p-155 • 원형 큐의 공백 및 포화 상태 표현

- 공백(empty) 상태: front == rear
- 포화(full) 상태: front == (rear + 1) % M (여기서 M은 큐의 크기) M: 큐의 크기.
  - 주의: front 및 rear를 증가하는 경우 꼭 modulus(%) 계산이 필요.
- 공백 상태와 포화 상태를 구별하기 위해 하나의 보조가 저장될 공간은 항상 비워 둔다.







p-156 ~157 • 원형 큐의 삽입, 삭제 알고리즘.

<원형 큐에서의 삽입 알고리즘>

```
enqueue(Q, x):
    rear ← (rear+1) % MAX_QUEUE_SIZE;
    Q[rear] ← x;
```

<원형 큐에서의 삭제 알고리즘>

```
dequeue(Q):
    front ← (front + 1) % MAX_QUEUE_SIZE;
    return Q[front];
```

• 큐 프로그램 (원형 큐의 구현)

p.157~p.160. (1/2)

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_QUEUE_SIZE 5

typedef int element;   편의상 새로 만듦.
typedef struct {  // 큐 타입
        element data[MAX_QUEUE_SIZE];   새로운 type 큐를 만듦.
        int front, rear;   2개 필요
} QueueType;

// 공백 상태 검출 함수
int is_empty(QueueType *q)
{
        return (q->front == q->rear);
}

// 포화 상태 검출 함수
int is_full(QueueType *q)
{                                          큐의 크기=M
        return ((q->rear + 1) % MAX_QUEUE_SIZE == q->front);
}
```

• 큐 프로그램 (2/2)

//삽입함수

void enqueue(QueueType *q, element item)

{

// full체크

if (is_full(q))

error("큐가 포화상태입니다");

q->rear = (q->rear+1) % MAX_QUEUE_SIZE; ① 증가

q->data[q->rear] = item; ② 삽입

}

① rear 증가 먼저 할 것!!

② item 삽입!!

//삭제함수 return value형 element(int)가 올 것임.

element dequeue(QueueType *q)

{

// empty체크

if (is_empty(q))

error("큐가 공백상태입니다.");

q->front = (q->front+1) % MAX_QUEUE_SIZE;

return q->data[q->front];

① front 증가

② 증가된 front에 있는 item 삭제!

5.4 큐의 응용 : 버퍼

p.160~162

• 큐는 어디에 사용될까?

→ 큐는 서로 다른 속도로 실행되는 두 프로세스 간의 상호 작용을 조화시키는 버퍼 역할

↓

(ex) CPU와 프린터 사이의 프린팅 버퍼/CPU와 키보드 사이의 키보드 버퍼 등

〈대표적인 큐의 응용 분야〉

- 생산자 - 소비자 프로세스 : 큐를 버퍼로 사용.
- 교통 관리 시스템 : 컴퓨터로 제어되는 신호등에는 신호 등을 순차적으로 제어하는데 원형 큐가 사용
- CPU 스케줄링 : 운영체제는 실행 가능한 프로세스들을 저장하거나 이벤트를 기다리는 프로세스들을 저장하기 위해 몇 개의 큐도 사용.

(궁금하면 p.161~162의 〈큐 응용 프로그램〉을 보세요.)

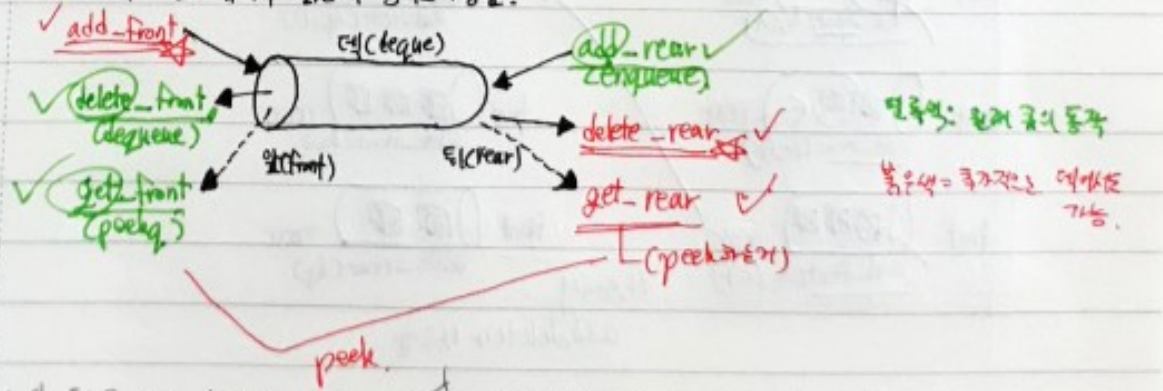
5.5 덱이란?

p.162~167

• 덱 (Deque).

▫ 덱(deque)은 double-ended queue의 준말로서 큐의 앞(front)과 뒤(rear)에서 삽입과 삭제가 모두 가능한 큐.

▫ 아래 그림의 큐에서 붉은 색 동작도 가능함.



※ 앞, 뒤 모두에서의 삽입/삭제가 가능하다. but 여전히 중간에 삽입, 삭제는 허용 X.

p.163

• 덱 ADT

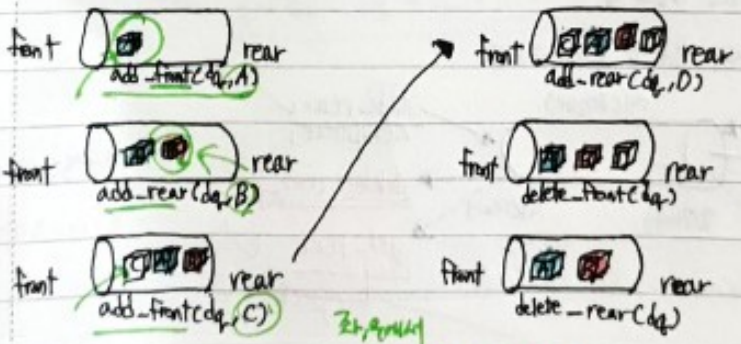
· 객체: n개의 element형으로 구성된 요소들의 순서있는 모임.

· 연산:

- create() ::= 덱을 생성한다.
- init(dq) ::= 덱을 초기화한다.
- is_empty(dq) ::= 덱이 공백상태인지를 검사한다.
- is_full(dq) ::= 덱이 포화상태인지를 검사한다.
- add_front(dq, e) ::= 덱의 앞에 요소를 추가한다.
- add_rear(dq, e) ::= 덱의 뒤에 요소를 추가한다.
- delete_front(dq) ::= 덱의 앞에 있는 요소를 반환한 다음 삭제한다.
- delete_rear(dq) ::= 덱의 뒤에 있는 요소를 반환한 다음 삭제한다.
- get_front(q) ::= 덱의 앞에서 삭제하지 않고 요소를 반환한다.
- get_rear(q) ::= 덱의 뒤에서 삭제하지 않고 요소를 반환한다.

※ 공간 표현을 위해 이들 덱에 앞 front와 뒤 rear를 전부 표기.

• 덱의 동작 보기.



조작위치에 add, delete가 반응함.

p.164 • 배열을 이용한 원형 덱의 구현

☐ 배열을 사용한 원형 큐의 구현과 유사함.

☐ delete_rear() 및 add_front() 동작 구현 시, rear 혹은 front를 감소시키는데,

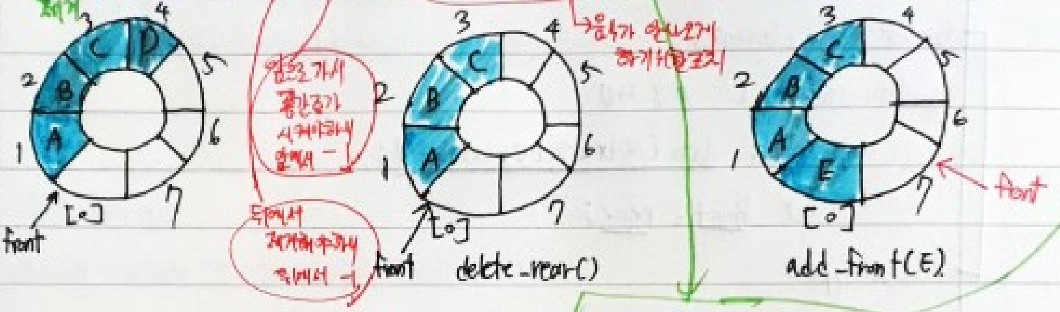
이 때 결과 값이 음수가 되지 않게 하기 위하여 주의가 필요함.

(★ 큐에서는 증가만, 덱에서는 증가 감소 둘 다 해야하니 주의해야 한다)

앞 공간 추가 ☐ front <- ((front - 1) + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE

뒤 공간 제거 ☐ rear <- ((rear - 1) + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE.

(음수가 안 나오게 하기 위한 조치)



음수가 안 나오게 하기 위해 감소 때 -1 한 것에 Arraysize만큼 더한 값을 그 전체를 Array size로 Modulo 연산을 해줘야 한다!!

• 위에 것을 제외하고는

큐의 함수들을 그대로 똑같이

사용하면 된다.

(원형)

• 덱 프로그램 (1/3)

p465 ~467

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_QUEUE_SIZE 5.


typedef int element;
typedef struct {  // 큐 타입
        element data[MAX_QUEUE_SIZE];
        int front, rear;
} DequeType;

// 공백 상태 검출 함수
int is_empty(DequeType *q)
{
        return (q->front == q->rear);
}

// 포화 상태 검출 함수
int is_full(DequeType *q)
{
        return ((q->rear+1) % MAX_QUEUE_SIZE == q->front);
}
```

메인에서 큐나 덱을 변수로 만들

원형

• 덱 프로그램 (2/3)

```
void add_front(DequeType *q, element val)
{
    if(is_full(q))                    add시에 항상 full인지 체크!!
        error("큐가 포화상태입니다.")
    q->data[q->front] = val;          ① 먼저 값이 들어감!!
    q->front = (q->front - 1 + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
}                                     ② 줄어든다!!

//front 삭제 함수
element delete_front(DequeType *q)
{
    if(is_empty(q))
        error("큐가 공백상태입니다")
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}
```

원형

• 덱 프로그램 (3/3)

```
void add_rear(DequeType *q, element item)
{
    if(is_full(q))
        error("큐가 포화상태입니다.");
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}

//rear 삭제 함수
element delete_rear(DequeType *q)
{
    int prev = q->rear;
    if(is_empty(q))
        error("큐가 공백상태입니다");
    q->rear = (q->rear - 1 + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;   ① 줄어들고
    return q->data[prev];                                        ② 없어진다, 삭제한다!
}
```

• 스택이나 큐같이 덱도 연결 리스트 구현 가능 (코드 복잡하니까 패스)

5.6 큐의 응용: 시뮬레이션

p.168~170 • 큐 → 주로 컴퓨터로 큐잉이론에 따라 시스템의 특성을 "시뮬레이션"하여 분석하는 데 이용.

ㄴ여기서 큐잉 모델은 고객에 대한 서비스를 수행하는 서버와 서비스를 받는 고객들로 이루어진다.

ㄴ제한된 수의 서버 때문에 고객들이 대기행렬 발생 (대기행렬 = 큐)

ㄴ이 대기행렬이 큐로 구현

• 은행에서 고객이 들어와서 서비스를 받고 나가는 과정을 '시뮬레이션'해보자.

-우리에게 필요한 것은 "고객들이 기다리는 평균시간이 얼마나 되느냐" [너무 길면 창구 더 두기 ↓ 대기시간 줄이기

<은행에서의 서비스 큐>

• 시뮬레이션은 하나의 반복 루프로 이루어진다.

(1) 먼저 현재시각을 나타내는 clock이라는 변수를 하나 증가.

(2) [0, 10] 사이의 난수를 생성하여 3보다 작으면 새로운 고객이 들어왔다고 판단한다.

새로운 고객이 들어오면 구조체를 생성하고 여기에 고객의 아이디, 도착시간, 서비스 시간 등의 정보를 복사한다.

여기서 고객이 필요로 하는 서비스 시간도 역시 난수로 생성한다.

이 구조체를 enqueue()를 호출하여 큐에 추가.

전역변수인 service_time에 현재 처리 중인 고객의 서비스 시간을 저장.

(3) service_time이 0인지 아닌지를 살펴본다.

만약 service_time이 0이 아니면 어떤 고객이 지금 서비스를 받고 있는 중임을 의미.

clock이 하나 증가했으므로 service_time을 하나 감소시킨다.

만약 service_time이 0이면 현재 서비스 받는 고객이 없다는 것을 의미.

따라서 큐에서 고객 구조체를 하나 꺼내어 서비스를 시작.

즉, 서비스를 시작한다는 의미는 전역 변수 service_time에 고객의 서비스 시간을 저장한다는 것.

보다 복잡한 거래를 시뮬레이션하려면 코드를 추가.

(4) 60분의 시간이 지나면 고객들의 기다린 시간을 전부 합하여 화면에 출력.

(코드 보고 싶으면 p.169~170 봐)

여기에 관한 내용임.

큐(덱) → 시뮬레이션 // 운영체제