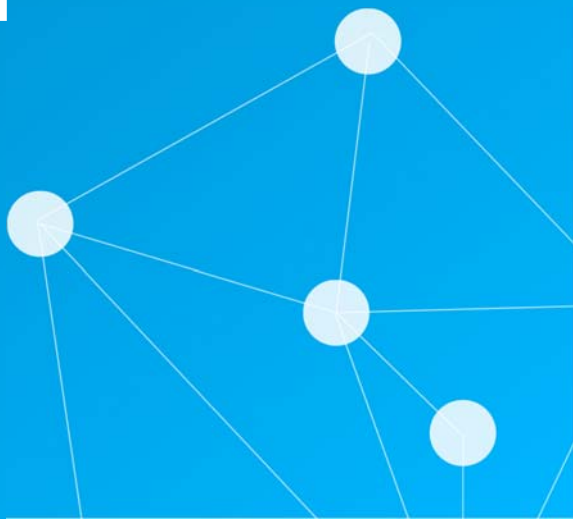
DATA
STRUCTURES
USING C

- 큐의 개념과 추상 자료형을 이해한다.
- 배열을 이용한 큐의 구현 방법을 이해한다.
- 덱의 개념과 구현 방법을 이해한다.
- 큐를 이용하여 프로그래밍 할 수 있는 능력을 키운다.

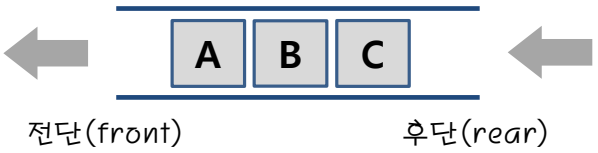
CHAPTER 04

큐



큐(QUEUE)

- 큐: 먼저 들어온 데이터가 먼저 나가는 자료구조
- **선입선출(FIFO: First-In First-Out)**
  - **(예)**매표소의 대기열



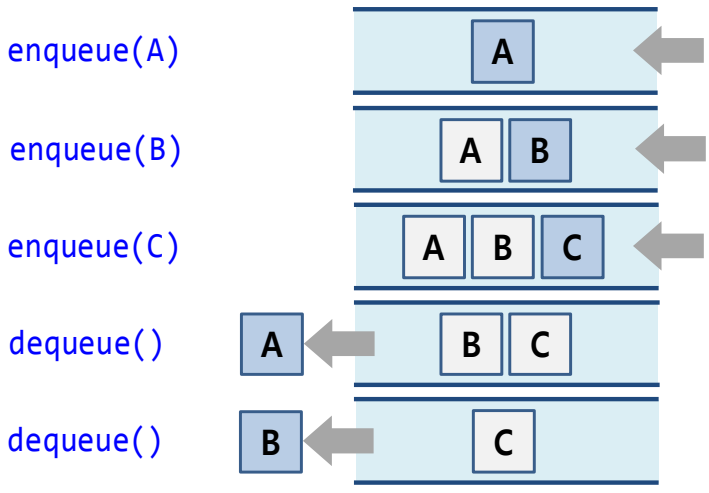
큐 ADT

- 삽입과 삭제는 FIFO(First In First Out)순서를 따른다.
- 삽입은 큐의 후단rear에서, 삭제는 전단front에서 이루어진다.

데이터: 선입선출(FIFO)의 접근 방법을 유지하는 요소들의 모음

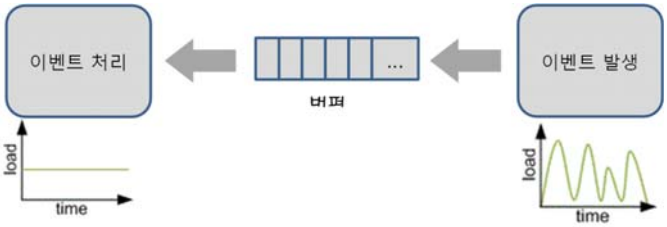
연산:
- init(): 큐를 초기화한다.
- enqueue(e): 주어진 요소 e를 큐의 맨 뒤에 추가한다.
- dequeue(): 큐가 비어있지 않으면 맨 앞 요소를 삭제하고 반환한다.
- is_empty(): 큐가 비어있으면 true를 아니면 false를 반환한다.
- peek(): 큐가 비어있지 않으면 맨 앞 요소를 삭제하지 않고 반환한다.
- is_full(): 큐가 가득 차 있으면 true을 아니면 false을 반환한다.
- size(): 큐의 모든 요소들의 개수를 반환한다.

큐의 연산



큐의 응용

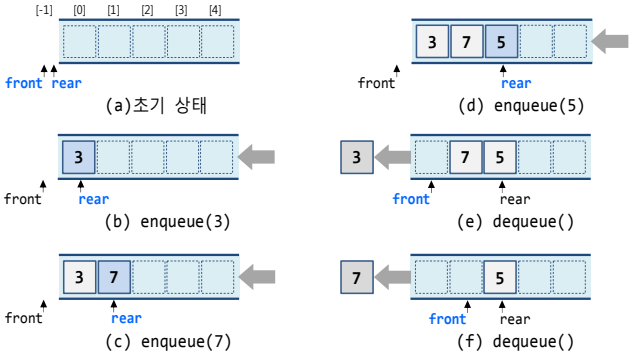
- 직접적인 응용
  - 시뮬레이션의 대기열(공항의 비행기들, 은행에서의 대기열)
  - 통신에서의 데이터 패킷들의 모델링에 이용
  - 프린터와 컴퓨터 사이의 버퍼링



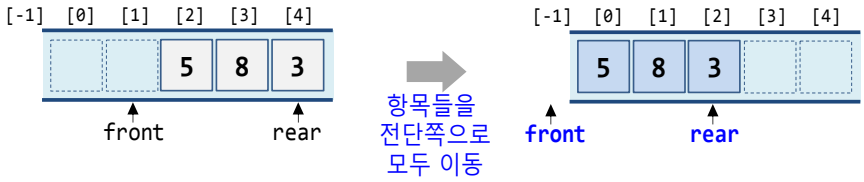
- 간접적인 응용
  - 스택과 마찬가지로 프로그래머의 도구
  - 많은 알고리즘에서 사용됨

4.2 배열을 이용한 큐 : 선형 큐

- 배열을 선형으로 사용하여 큐를 구현

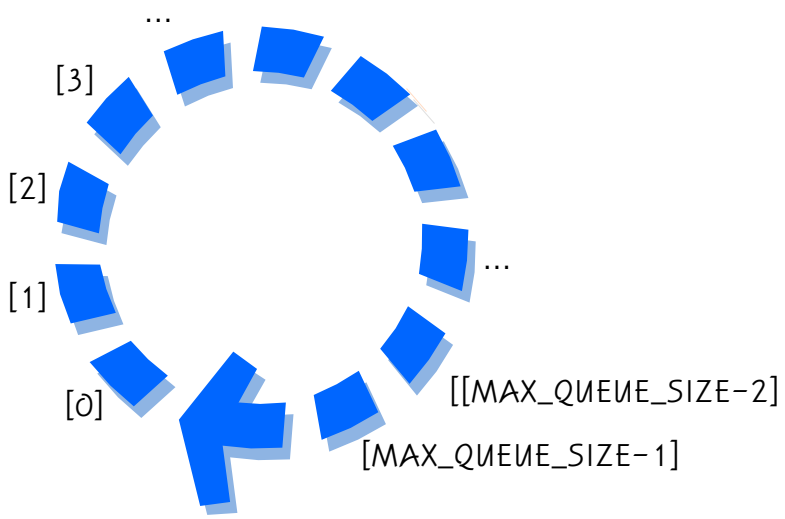


- 삽입을 계속하기 위해서는 요소들을 이동시켜야 함



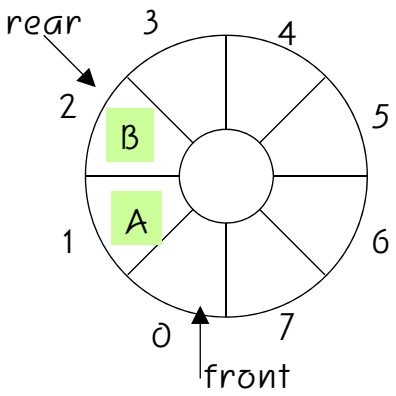
해결방법 → 원형큐

- 원형큐: 배열을 원형으로 사용하여 큐를 구현

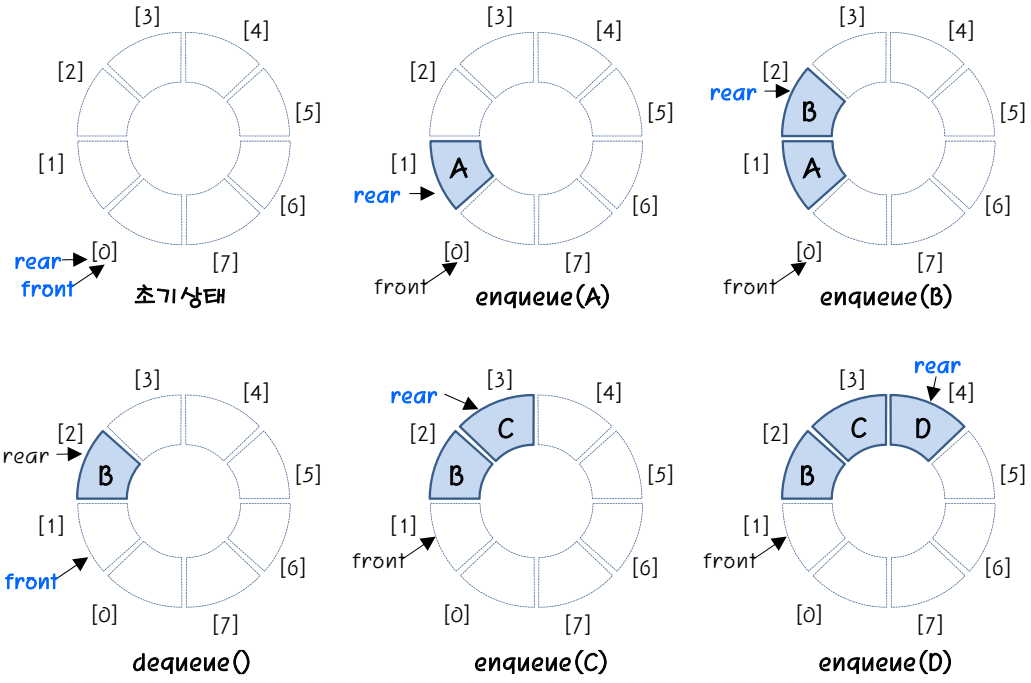


원형큐의 구조

- 전단과 후단을 관리하기 위한 2개의 변수 필요
  - front: 첫번째 요소 하나 앞의 인덱스
  - rear: 마지막 요소의 인덱스

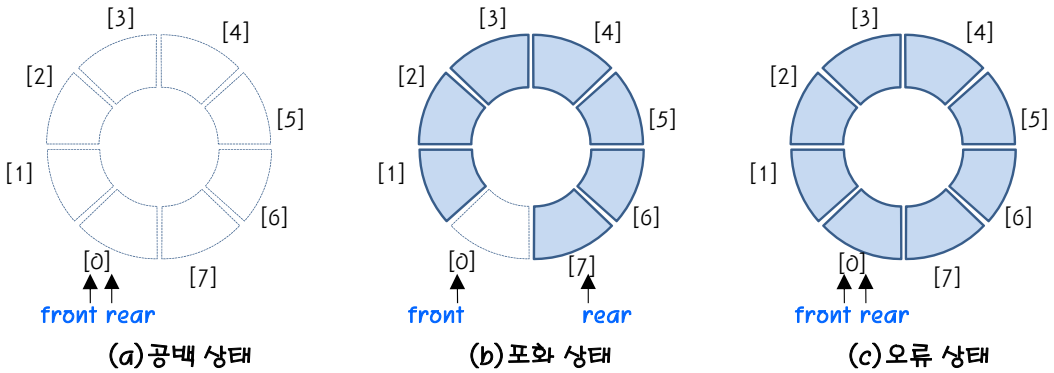


원형큐의 삽입과 삭제연산 예



공백상태, 포화상태

- 공백상태: `front == rear`
- 포화상태: `front % M==(rear+1) % M`
- 공백상태와 포화상태를 구별 방법은?
  - 하나의 공간은 항상 비워둠



큐의 연산

- 나머지(modulo) 연산을 사용하여 인덱스를 원형으로 회전시킨다.
- 삽입 연산

```
enqueue(x)
rear ← (rear+1) mod MAX_QUEUE_SIZE;
data[rear] ← x;
```

- 삭제 연산

```
dequeue()
front ← (front+1) mod MAX_QUEUE_SIZE;
return data[front];
```


- 선형 큐의 문제점은 무엇인가?
- 원형 큐에서 front와 rear가 가리키는 것은 무엇인가?
- 크기가 10인 원형 큐에서 front와 rear가 모두 0으로 초기화되었다고 가정하고 다음과 같은 연산 후에 front와 rear의 값을 말하라.

  enqueue(a), enqueue(b), enqueue(c), dequeue(), enqueue(d)

- 원형 큐의 front와 rear의 값이 각각 7과 2일 때, 이 원형 큐가 가지고 있는 데이터의 개수는? (단, MAX_QUEUE_SIZE는 12이고, front와 rear의 초기값은 0이다.)

4.3 원형 큐의 구현

- 원형 큐를 위한 데이터
  - int 큐에서 Element는 int로 지정
  - 전역변수 사용

```
#define MAX_QUEUE_SIZE   100
#define Element  int

Element data[MAX_QUEUE_SIZE];
int     front;
int     rear;
```

- 원형 큐의 단순한 연산들

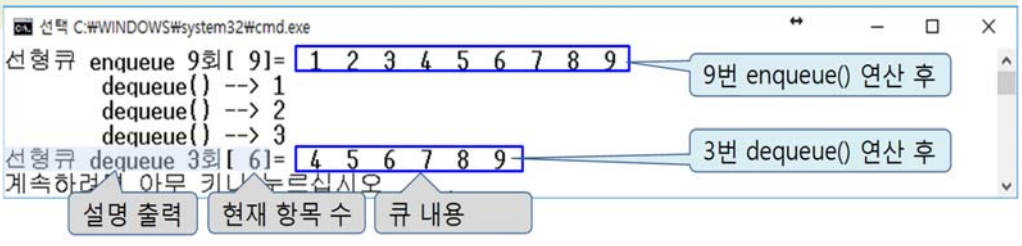
```
void init_queue() { front = rear = 0; ; }
int is_empty() { return front == rear;; }
int is_full() { return (rear+1)%MAX_QUEUE_SIZE == front; }
int size() { return (rear-front+MAX_QUEUE_SIZE)%MAX_QUEUE_SIZE; }
```

원형 큐의 구현

```
void enqueue( Element val )
{
        if( is_full() )
                error(" 큐 포화 에러");
        rear = (rear+1) % MAX_QUEUE_SIZE;
        data[rear] = val;
}
Element dequeue( )
{
        if( is_empty() )
                error(" 큐 공백 에러");
        front = (front+1) % MAX_QUEUE_SIZE;
        return data[front];
}
Element peek( )
{
        if( is_empty() )
                error(" 큐 공백 에러");
        return data[(front+1) % MAX_QUEUE_SIZE];
}
```


사용방법

```
void main()
{
        int i;
        init_queue( );
        for( i=1 ; i<10 ; i++ )
         enqueue( i );
        print_queue("선형큐 enqueue 9회");
        printf("\tdequeue() --> %d\n", dequeue());
        printf("\tdequeue() --> %d\n", dequeue());
        printf("\tdequeue() --> %d\n", dequeue());
        print_queue("선형큐 dequeue 3회");
}
```



실습

- 원형 큐에서는 공백 상태와 포화 상태를 구분하기 위하여 필수적으로 하나의 빈 공간이 필요하다. 이것은 하나의 변수를 큐에 추가함으로써 모든 배열 공간을 사용할 수도 있다. lastOp라고 하는 변수는 가장 최근에 큐에 행해진 연산을 기억하고 있다. 만약 최근에 행해진 연산이 삽입 연산이었다면 큐는 절대로 공백 상태가 아닐 것이다. 마찬가지로 만약 최근에 행해진 연산이 삭제 연산이었다면 절대로 포화 상태는 아닐 것이다. 따라서 lastOp 변수는 front == rear 일 때 공백 상태와 포화 상태를 구분하는데 이용될 수 있다. 이것을 구현하여 테스트한 예를 보여라.

실습

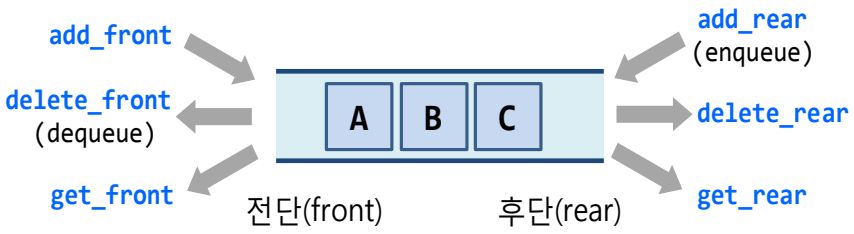
- 피보나치 수열을 효과적으로 계산하기 위하여 큐를 이용할 수 있다. 만일 피보나치 수열을 순환에 의하여 계산하게 되면 경우에 따라서는 많은 순환 함수의 호출에 의해 비효율적일 수 있다. 이를 개선하기 위하여 큐를 사용하는데 큐에는 처음에는 F(0)와 F(1)의 값이 들어가 있어 다음에 F(2)를 계산할 때 F(0)를 큐에서 제거한다. 그 다음에 계산된 F(2)를 다시 큐에 넣는다. 피보나치 수열은 다음과 같이 정의된다. 큐를 이용하여 피보나치 수열을 계산하는 프로그램을 작성하라.

  $F(0) = 0, F(1) = 1$

  $F(n) = F(n-1) + F(n-2)$

4.4 덱(deque)

- 덱(deque)은 double-ended queue의 줄임말
  - 전단(front)와 후단(rear)에서 모두 삽입과 삭제가 가능한 큐



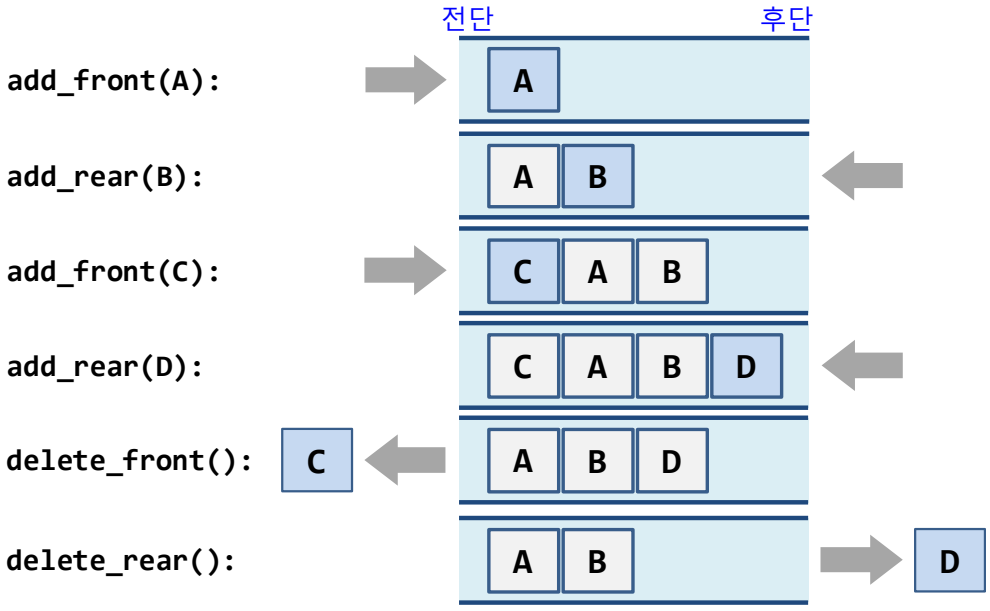
덱 ADT

- 큐와 데이터는 동일
  - 연산은 추가됨

데이터: 전단과 후단을 통한 접근을 허용하는 요소들의 모음

연산:
- init(): 덱을 초기화한다.
- add_front(e): 주어진 요소 e를 덱의 맨 앞에 추가한다.
- delete_front(): 전단 요소를 삭제하고 반환한다.
- add_rear(e): 주어진 요소 e를 덱의 맨 뒤에 추가한다.
- delete_rear(): 후단 요소를 삭제하고 반환한다.
- is_empty(): 공백 상태이면 TRUE를 아니면 FALSE를 반환한다.
- get_front(): 전단 요소를 삭제하지 않고 반환한다.
- get_rear(): 후단 요소를 삭제하지 않고 반환한다.
- is_full(): 덱이 가득 차 있으면 TRUE를 아니면 FALSE를 반환한다.
- size(): 덱 내의 모든 요소들의 개수를 반환한다.

덱의 연산



원형 덱의 연산

- 큐와 데이터는 동일함
- 연산은 유사함.

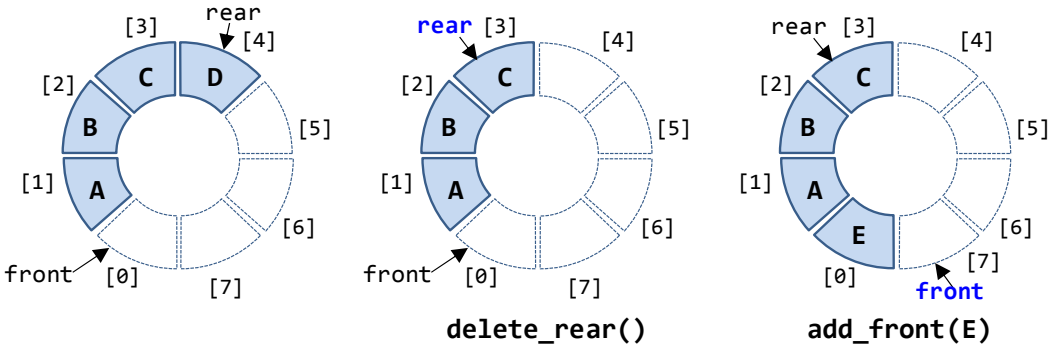
- 큐와 알고리즘이 동일한 연산
  - init_deque() : 원형큐의 init_queue()
  - print_deque(): 원형큐의 print_queue()
  - add_rear(): 원형큐의 enqueue()
  - delete_front(): 원형큐의 dequeue()
  - get_front(): 원형큐의 peek()

원형 덱의 연산

- 덱에서 추가된 연산
  - delete_rear(), add_front(), get_rear()
  - 반대방향의 회전이 필요: delete_rear, add_front의 경우

```
front ← (front-1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
rear ← (rear-1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
```

  - 예:



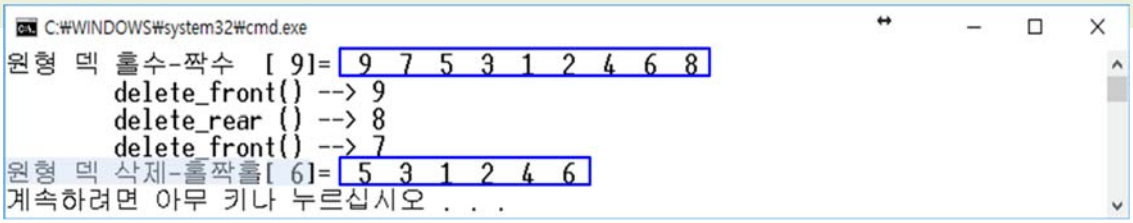
4.5 원형 덱의 구현

```
void init_deque( ) { init_queue( ); }
void add_rear(Element val) { enqueue( val);}
Element delete_front( ) { return dequeue(); }
Element get_front( ) { return peek(); }
void print_deque(char msg[]) { print_queue(msg); }

void add_front( Element val ) {
        if( is_full() )
                error(" 덱 포화 에러");
        data[front] = val;
        front = (front-1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
}
Element delete_rear( ) {
        Element ret;
        if( is_empty() )
                error(" 덱 공백 에러");
        ret = data[rear];
        rear = (rear-1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
        return ret;
}
Element get_rear( ){
        if( is_empty() )
                error(" 덱 공백 에러");
        return data[rear];
}
```

원형 덱 테스트 프로그램

```
void main()
{
        int i;
        init_deque( );
        for( i=1 ; i<10 ; i++ ) {
         if( i % 2 ) add_front( i );
         else add_rear( i );
        }
        print_queue("원형 덱 홀수-짝수 ");
        printf("\tdelete_front() --> %d\n", delete_front());
        printf("\tdelete_rear () --> %d\n", delete_rear ());
        printf("\tdelete_front() --> %d\n", delete_front());
        print_queue("원형 덱 삭제-홀짝홀");
}
```

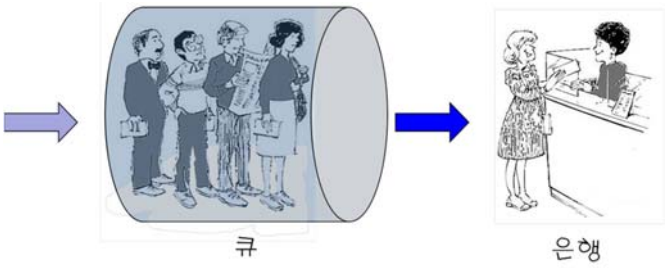


실습

- 회문(palindrome)이란 앞뒤 어느 쪽에서 읽어도 똑같은 단어나 문장을 의미한다. 예를 들면 "eye", "Madam, I'm Adam", "radar" 등이다. 여기서 물론 구두점이나 스페이스, 대소문자 등은 무시하여야 한다. 덱을 이용하여 주어진 문자열이 회문인지 아닌지를 결정하는 프로그램을 작성하라. 덱을 하나만 사용한다.

4.6 큐의 응용 : 은행 시뮬레이션

- 은행 시뮬레이션
  - 큐잉이론에 따라 시스템의 특성을 시뮬레이션하여 분석하는 데 이용
  - 큐잉모델은 고객에 대한 서비스를 수행하는 서버와 서비스를 받는 고객들로 이루어진다
  - 고객이 들어와서 서비스를 받고 나가는 과정을 시뮬레이션
    - 고객들이 기다리는 평균시간을 계산



큐의 응용 : 은행 시뮬레이션

- 입력:
  - 시뮬레이션 할 최대 시간 (예: 10 [단위시간])
  - 단위시간에 도착하는 고객 수 (예: 0.5 [고객수/단위시간])
  - 한 고객에 대한 최대 서비스 시간 (예: 5 [단위시간/고객])
- 출력:
  - 고객들의 평균 대기시간
- 서비스 인원(은행원): 1명
- 고객 정보:
  - 단위시간에 도착하는 고객 수를 바탕으로 무작위로 발생
  - 서비스 시간: 일정한 범위 내에서 무작위로 결정

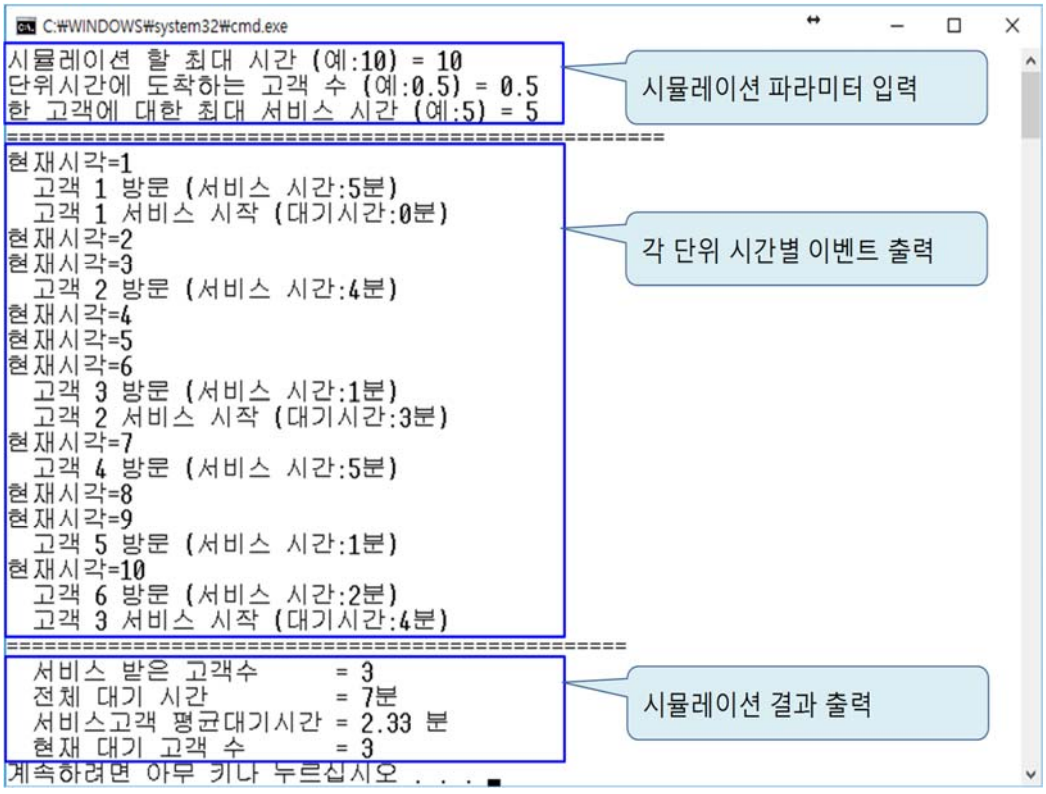
은행 시뮬레이션 테스트 프로그램

```
typedef struct BankCustomer {
        int     id;             // 고객 번호
        int     tArrival;       // 도착 시간
        int     tService;       // 서비스에 필요한 시간
} Customer;
typedef Customer Element;

int     nSimulation;
double  probArrival;
int     tMaxService;
int     totalWaitTime;
int     nCustomers;
int     nServedCustomers;
...

#include <time.h>
void main() {
        srand( (unsigned int)time(NULL) );
        read_sim_params( );
        run_simulation();
        print_result();
}
```


실행결과 예



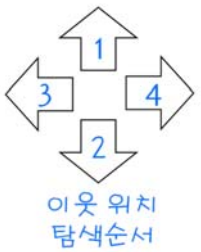
실습

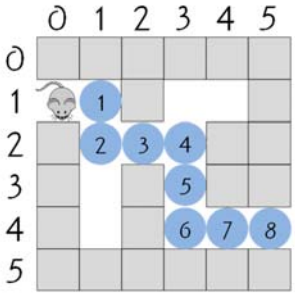
- 창구가 2 개일 때와 행원이 1명이지만 그 능력이 2배일 때의 두가지 경우에 대해서 고객들의 평균 대기 시간을 구하여 비교하도록 수정하라. 의미 있는 결과가 나오도록 시뮬레이션 시간을 충분히 크게 하라.

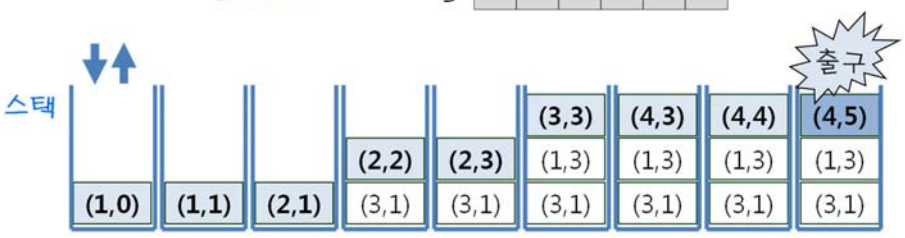
4.7 덱의 응용 : 미로 탐색 DFS

- 깊이 우선 탐색(DFS, Depth First Search)

```
typedef struct {
    int x;
    int y;
} Location2D;
```



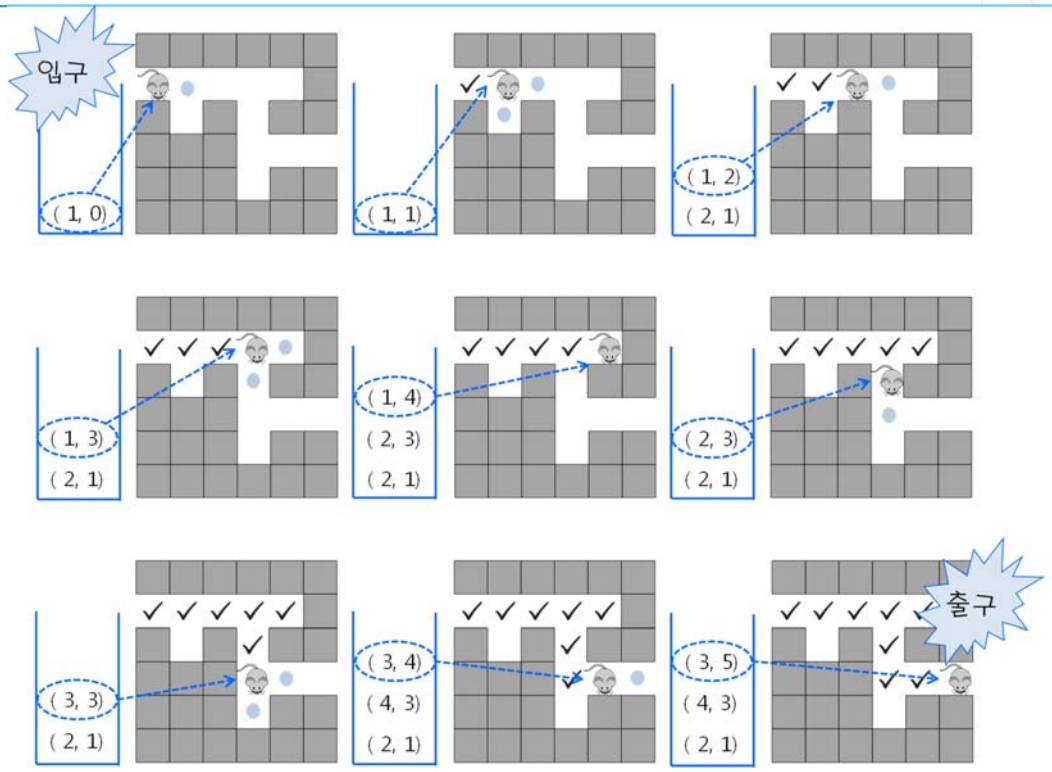




- depth_first_search()

  스택과 출구의 위치 x를 초기화
  스택에 입구의 위치를 삽입;
  while (is_empty() = false)
  do 현재위치 ← pop();
  if (현재위치가 출구이면)
  then 성공;
  if (현재위치의 위, 아래, 왼쪽, 오른쪽 방이 아직 방문
  되지 않았고 갈 수 있으면)
  then 그 위치들을 스택에 push();
  실패;

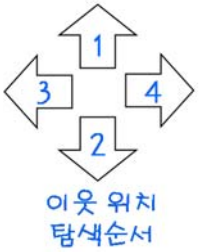
미로 탐색 DFS

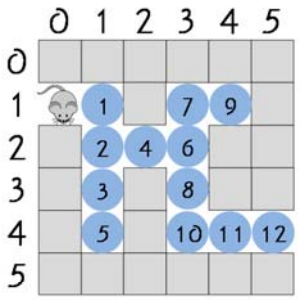


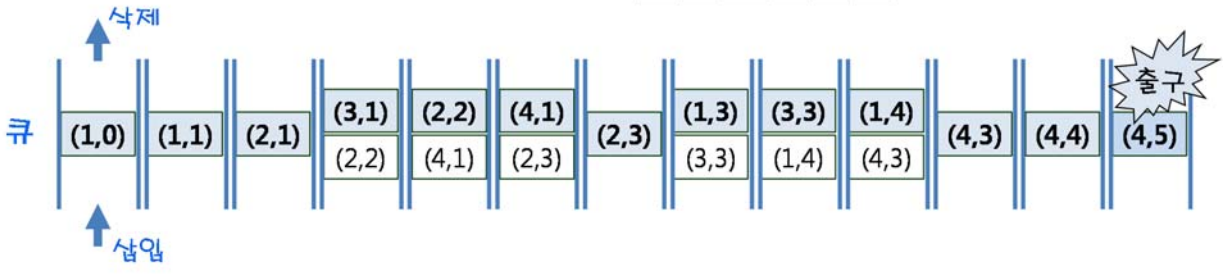
```
int DFS()
{
    int x, y;
    Location2D here;
    init_deque( );
    add_rear( getLocation2D(0,1) );
    printf("DFS: ");
    while ( is_empty() == 0 ) {
        here = delete_rear( );
        print_elem( here );
        x = here.x;
        y = here.y;
        if( map[x][y] == 'x' ) return 1;
        else {
            map[x][y] = '.';
            if( is_valid( x-1, y ) ) add_rear(getLocation2D(x-1,y));
            if( is_valid( x+1, y ) ) add_rear(getLocation2D(x+1,y));
            if( is_valid( x, y-1 ) ) add_rear(getLocation2D(x,y-1));
            if( is_valid( x, y+1 ) ) add_rear(getLocation2D(x,y+1));
        }
    }
    return 0;
}
```

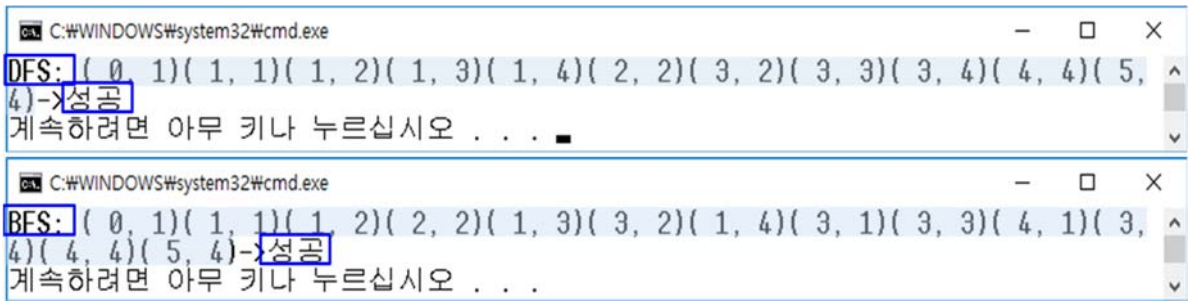

덱의 응용 : 미로 탐색 BFS

- 너비 우선 탐색(BFS, Breadth First Search)









4.8 전역 변수와 객체지향 프로그래밍

- 전역변수 + 전역함수
  - 좋지 않은 프로그래밍 습관?
  - 하나의 큐만 사용 가능

- **여러 개의 큐**가 필요한 경우 해결방안?
  ➔C++: 클래스로 전환 가능 ➔ 어렵지 않음!
  ➔C언어: 구조체 선언 및 매개변수 전달 ➔ 오히려 더 복잡함!

C++ 클래스로 전환

```
class Queue {
        Element   data[MAX_QUEUE_SIZE];       // 요소의 배열
        int       front, rear;
public:
        void init_queue( ) { front = rear = 0; ; }
        int is_empty( )    { return front == rear;; }
        ...
        void print_queue(char msg[]) ...
};
void main()
{
        int i;
        Queue q;
        q.init_queue( );
        for( i=1 ; i<10 ; i++ )
                q.enqueue( i );
        q.print_queue("원형큐 enqueue 9회");
        printf("\tdequeue() --> %d\n", q.dequeue());
        printf("\tdequeue() --> %d\n", q.dequeue());
        printf("\tdequeue() --> %d\n", q.dequeue());
        q.print_queue("원형큐 dequeue 3회");
}
```


C 구조체 선언 및 매개변수 전달

```
typedef int Element;
typedef struct CircularQueue {
    Element data[MAX_QUEUE_SIZE];
    int  front, rear;
} Queue;
```

```
void init_queue(Queue *q) { q->front = q->rear = 0; ; }
int is_empty(Queue *q) { return q->front == q->rear; }
...
void enqueue( Queue *q, Element val )
Element dequeue( Queue *q )
...
void main() {
        int i;
        Queue q;
        init_queue( &q );
        for( i=1 ; i<10 ; i++ )
         enqueue( &q, i );
        print_queue(&q, "선형큐 enqueue 9회");
        printf("\tdequeue() --> %d\n", dequeue(&q));
        printf("\tdequeue() --> %d\n", dequeue(&q));
        printf("\tdequeue() --> %d\n", dequeue(&q));
        print_queue( &q, "선형큐 dequeue 3회");
}
```