



첫 번째 강의

배열



목차

- 배열(**Array**)의 정의
- 배열에 적용가능한 연산
- 구조체(**Structure**)의 정의
- 실습
- 스스로 프로그래밍
- 침삭 지도



1. 배열의 정의

- C 언어에서 배열
 - 배열의 선언
 - `int list[5], *plist[5];`
 - 배열의 첨자(index)는 0부터 시작
 - 배열의 구현
 - `list[0]`의 주소 = α 라고 가정.
`list[1]`의 주소 = $\alpha + \text{sizeof}(\text{int})$
 - `list + i = &list[i]`, `*(list + i) = list[i]`
 - 배열을 인자로 전달: 배열의 시작주소가 복사
← Call-by Reference와 동일한 효과




배열의 동적 할당

- C 언어에서 1차원 배열의 동적 할당

```
int *A = (int *) malloc(sizeof(int) * 100);  
int *B = (int *) calloc(100, sizeof(int));  
  
A = (int *) realloc(A, sizeof(int) * 200);  
free(A);
```

- C 언어에서 2차원 배열의 동적 할당

```
int **C = (int **) malloc(sizeof(int *) * 10);  
for (int i = 0; i < 10; i++)  
    C[i] = (int *) malloc(sizeof(int) * 20);
```



2. 배열에 적용가능한 연산

- 뭐가 저장되어 있는가?
 - 배열의 내용 검사
- 저장된 내용을 내가 원하는 방식으로 바꾸자!
 - 배열의 내용 검사 + 원소 변경
- 배열을 저장 공간으로 사용하자!
 - 어떤 방식으로 저장할 것인가?
 - 배열의 어느 위치에 저장할 것인가?



2.1 배열의 내용 검사

- **Example**

- 제일 큰 원소/특정 값을 갖는 원소 찾기
- 모든 원소들의 합/평균 구하기
- 배열 내용을 출력하기



1차원 배열에서 내용 검사

```
int search(int A[], int n) {  
    int max = A[0];  
    for (int i = 1; i < n; i++)  
        if (A[i] > max)  
            max = A[i];  
  
    return max;  
}
```



2차원 배열에서 내용 검사

```
int search(int A[][m], int n) {  
    int max = A[0][0];  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < m; j++)  
            if (A[i][j] > max)  
                max = A[i][j];  
    return max;  
}
```




2.2 배열의 내용 바꾸기

- 가정

- 배열에 이미 데이터가 저장되어 있다
- 저장된 데이터를 내가 원하는 순서(또는 내용)로 바꾸자...

- **Example**

- 특정 조건을 만족하는 데이터를 다른 것으로 변경
- 배열의 내용을 섞기
- 정렬

- 알고리즘의 구성

- 배열의 모든 원소들을 검사 &
- 조건을 만족한 원소를 변경



1차원 배열에서 내용 변경

```
void convert(int A[], int n) {  
    for (int i = 0; i < n; i++)  
        if (A[i] < 0)  
            A[i] = -A[i];  
}
```

```
void shuffle(int A[], int n) {  
    int i, j, tmp;  
    for (int i = 0; i < n; i++) {  
        j = rand() % n;  
        tmp = A[i];  A[i] = A[j];  A[j] = tmp;  
    }  
}
```



2차원 배열에서 내용 변경

```
void convert(char A[][m], int n) {  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < m; j++)  
            if (islower(A[i][j]))  
                A[i][j] = toupper(A[i][j]);  
}
```

```
void shuffle(int A[][m], int n) {  
    int row, col, tmp;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < m; j++) {  
            row = rand() % n; col = rand() % m;  
            swap(A[i][j], A[row][col], tmp);  
        }  
}
```



1차원 배열의 정렬 - 선택 정렬

```
void sort(int list[], int n)
{
    int i, j, min, temp;
    for (i=0; i<n-1; i++) {
        min = i;
        for (j=i+1; j<n; j++)
            if (list[j]<list[min])
                min = j;
        SWAP(list[i], list[min], temp); // 최소값과 i의 내용을 교체
    }
}
```

// list[i]부터 list[n-1]까지 정렬.
// 최소값이 i에 있다고 일단 가정
// i 위치 다음의 모든 놈들에 대해
// 더 작은 것이 있으면
// 최소값을 이 놈으로...



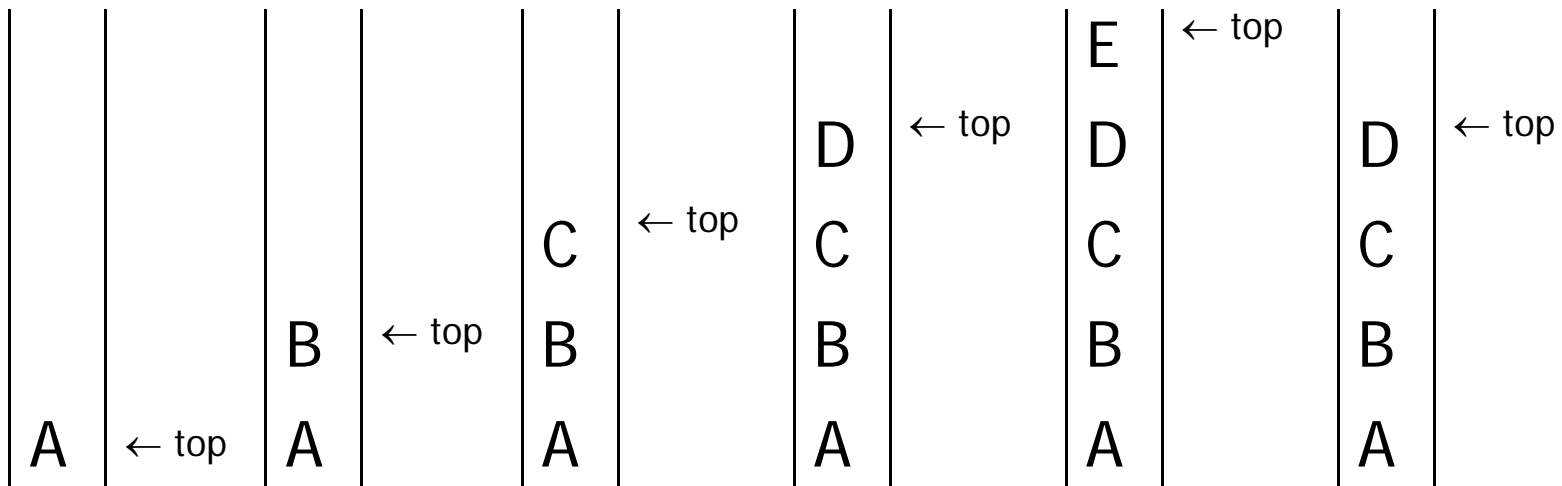
2.3 배열을 저장공간으로 사용

- 저장 방식에 따른 분류
 - 방식 1: 배열은 그냥 여러 개의 원소를 저장하는 공간
 - 각 원소마다 의미가 주어지지 않는 경우
 - 예: 스택, 큐
 - 방식 2: 각 원소가 별개의 counter 역할
 - 원소마다 의미가 할당
 - 예: A[0]에는 짝수의 수, A[1]에는 홀수의 수를 저장
- 저장 위치에 따른 분류
 - 데이터가 추가/삭제되는 위치가 고정되는 경우
 - 예: 스택(top), 큐(front, rear)
 - 배열의 모든 위치에서 데이터 추가/삭제가 발생
 - 예: `A[i%2]++;` `A[ch - '0']++;`

스택(Stack)

- 스택의 정의

- 삽입과 삭제가 "top"이라 불리는 한쪽 끝 지점에서 발생하는 순서화 리스트
- Last-In-First-Out (LIFO)



- 스택의 연산

- Push: top에 원소를 추가
- Pop: top에 있는 원소를 제거하고 반환



스택의 구현

- `int stack[100], top = -1;`

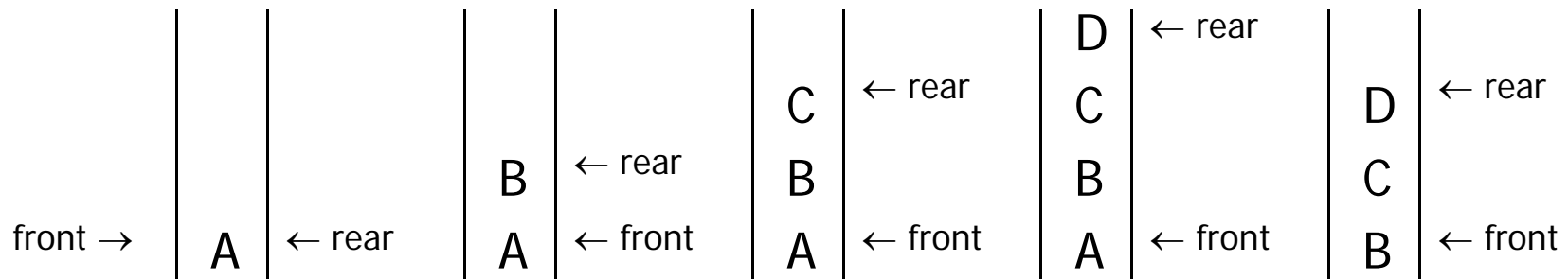
```
void push(int item)
{
    // 스택에 새로운 항목을 추가
    if (top >= 99) {
        stack_full();
        return;
    }
    stack[++top] = item;
}
```

```
int pop()
{
    // 스택 top의 항목을 return
    if (top == -1)
        return stack_empty();
    return stack[top--];
}
```

큐(Queue)


■ 큐의 정의

- 삽입과 삭제가 다른 쪽에서 발생하는 순서화 리스트
 - 삽입이 발생하는 위치: **rear**
 - 삭제가 발생하는 위치: **front**
- First-In-First-Out (FIFO)



■ 큐의 연산

- addq: rear 위치에 데이터를 추가
- deleteq: front 위치의 데이터를 제거하고 반환



큐의 구현

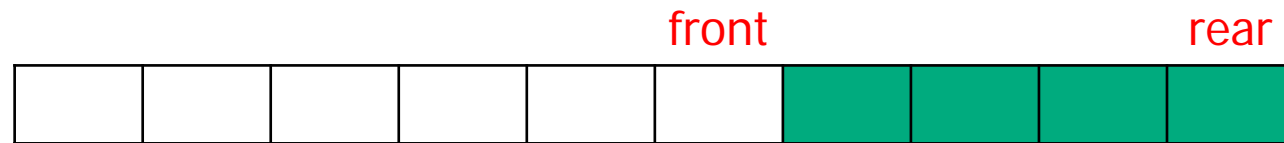
- `int queue[100], front = -1, rear = -1;`

```
void addq(int item)
{
    // Queue에 새로운 항목을 추가
    if (rear >= 99) {
        queue_full();
        return;
    }
    queue[++rear] = item;
}
```

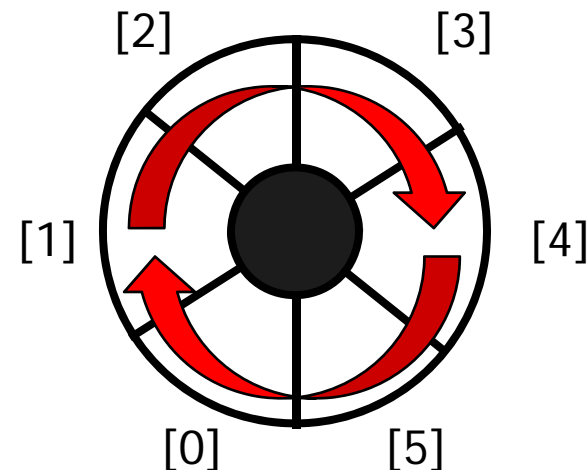
```
int deleteq()
{
    // Queue의 항목을 return
    if (front == rear) return queue_empty();
    return queue[++front];
}
```

원형 큐(Circular Queue)

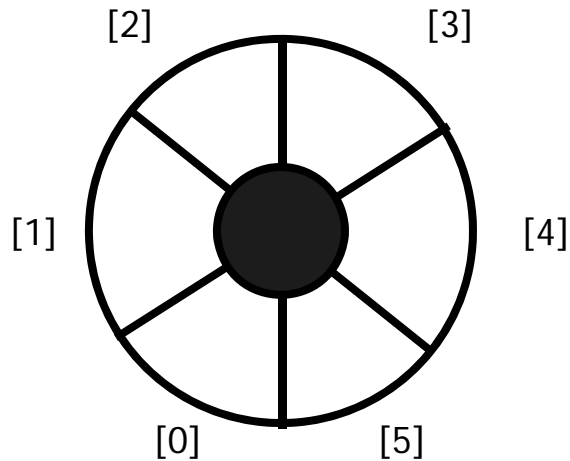
- 배열을 이용하여 큐를 구현할 때 발생하는 문제점
 - QueueFull의 조건: $\text{rear} == \text{Q_SIZE} - 1$
 - 문제점: **큐에 저장된 원소의 수 < Q_SIZE**
 - 큐의 모든 항목들을 왼쪽으로 이동
 - 최악의 성능: $O(\text{Q_SIZE})$



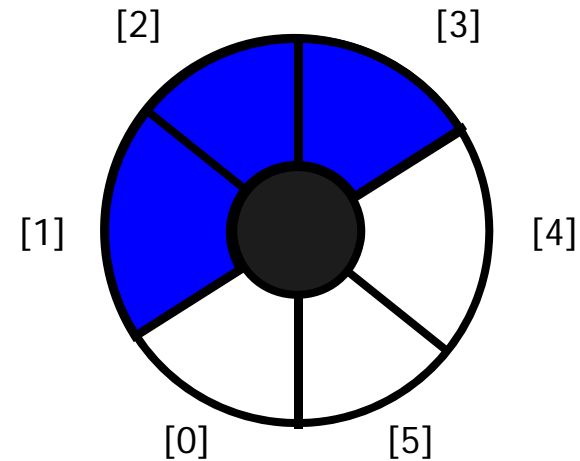
- 원형 큐의 개념
 - 큐의 처음과 마지막을 연결
 - 나머지(%) 연산자 이용



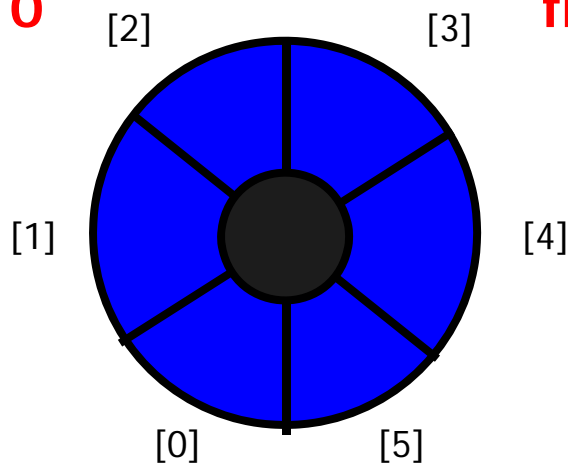
원형 큐(Circular Queue)



front = 0, rear = 0



front = 0, rear = 3



front = 0, rear = 0

최대 큐 이용률 =
 $\text{MAX_Q_SIZE} - 1$



원형 큐의 구현

```
void addq(int item)
{
    // 원형 큐에 새로운 항목을 추가
    rear = (rear + 1) % MAX_Q_SIZE;
    if (rear == front) {
        queue_full(); return;
    }
    queue[rear] = item;
}
```

```
int deleteq()
{
    // 원형 큐의 항목을 return
    if (front == rear)
        return queue_empty();
    front = (front + 1) % MAX_Q_SIZE;
    return queue[front];
}
```



3. 구조체

- 구조체

- 하나 이상의 기본 자료형을 기반으로 사용자 정의 자료형을 만들 수 있는 문법 요소
- 다양한 자료형을 포함 ↔ 배열: 동일한 자료형의 모음

```
struct humanBeing {  
    char    name[10];  
    int     age;  
    double  salary;  
};
```

```
typedef struct humanBeing human_being;
```


```
human_being person;  
strcpy(person.name, "홍길동");  
person.age = 21;  
printf("%f", person.salary);
```



Self-Referential Structure

- 자기 참조 구조체
 - 구조체의 속성중 하나가 스스로를 가리키는 구조체
- 예

```
struct list {  
    char  data;  
    struct list *link;  
};
```
- 연결 리스트의 구현에 많이 사용됨



배열 - 실습 1

- 2차원 배열 A[100][200] 에 rand() 함수를 이용하여 1부터 100사이의 정수를 무작위로 저장하라.
 - #include <stdlib.h>
 - rand()
- 위의 배열 A에 저장된 값에 대해 1부터 100까지 저장된 빈도수를 모두 출력하라.



배열 - 실습 2

- 2차원 배열 `A[100][200]` 에 `rand()` 함수를 이용하여 1부터 100사이의 정수를 무작위로 저장하라.
- 위의 배열 `A`에 저장된 값에 대해 가장 많이 나타난 상위 10개의 수와 빈도수를 출력하라.



배열 - 실습 3

- 2차원 배열 `A[100][200]` 에 `rand()` 함수를 이용하여 1부터 100사이의 정수를 무작위로 저장하라.
- 위의 배열 `A`에 저장된 값들을 오름차순으로 정렬하라.



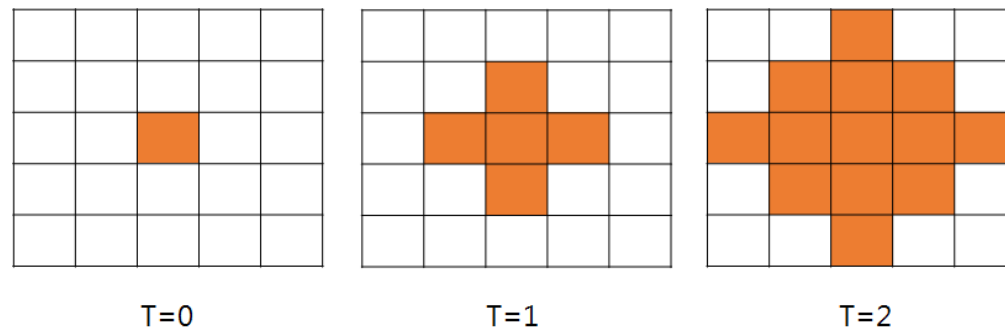
배열 - 문제 1 (Random Walk)

- 문제 설명
 - $n * m$ 행렬을 무작위로 순회하는데, 모든 원소를 적어도 한번 이상 방문하는 순간에 프로그램이 종료
 - 행렬의 크기 (n 과 m), 행렬 내에서 순회를 시작하는 좌표(row 와 col), 그리고 방문수의 한계값($limit$)이 입력으로 주어짐
 - 행렬은 이차원 배열로 구성하되, 동적으로 배열을 할당받도록 할 것
 - 배열의 모든 원소들은 0으로 초기화하며, 한 번씩 방문할 때마다 1씩 증가
 - 0인 원소가 존재하지 않거나, 방문수가 $limit$ 을 초과하면 프로그램 종료
- 나머지 내용들은 첨부한 문서 확인

배열 - 문제 2 (셀 활성화 Simulation)

■ 문제 설명

- 2차원 격자 셀이 있을 때, 일정 개수의 셀을 지정하면 시간이 지날 때마다 지정된 셀의 상하좌우 셀을 한칸씩 활성화 시킨다.



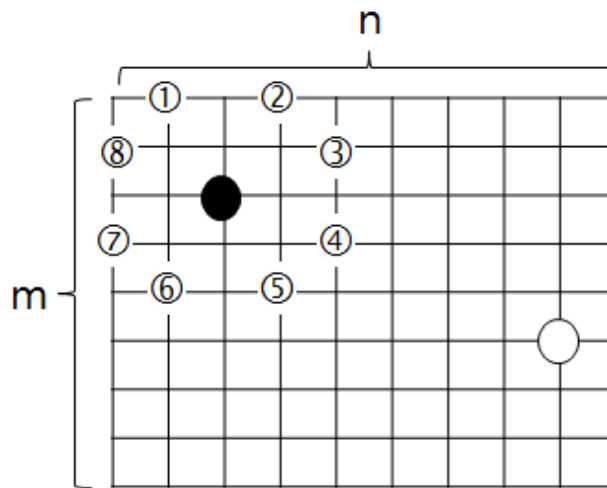
■ 입력 조건: 다음 4가지 사항 입력

- 2차원 행렬의 크기 M(행) 과 N(열)을 입력
- 초기 활성 셀의 개수를 입력
- 하나 이상의 서로 다른 활성 셀 좌표를 차례대로 입력
- 시뮬레이션을 수행할 time step (T)을 입력
- 나머지 내용들은 첨부한 문서 확인

배열 - 문제 3 (날일자(日) 행마)

■ 문제 설명

- $m * n$ 크기의 바둑판에서 백돌과 흑돌이 아래와 같이 놓여있다. 백돌의 위치는 고정이며, 흑돌은 날일자 행마로만 위치를 이동할 수 있다. 즉, 아래 그림에서 흑돌은 ①부터 ⑧까지 여덟 개의 위치 중 한 곳으로만 이동이 가능하다. 흑돌이 몇 번의 이동으로 백돌의 위치까지 갈 수 있는가? 최소한의 이동 횟수를 계산하라.



나머지 내용들은 첨부한 문서 확인