

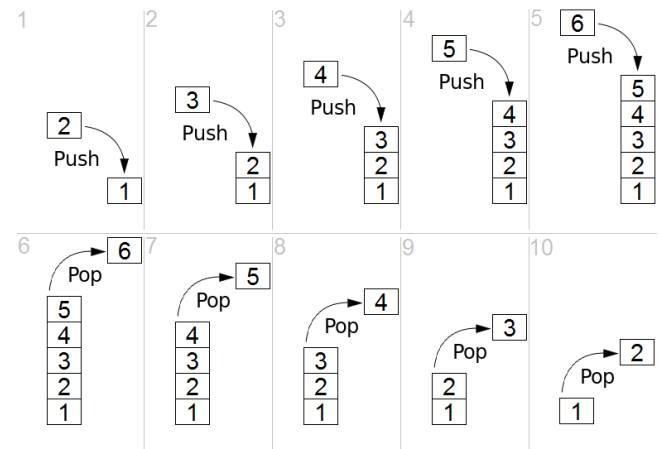
# 스택과 큐 (Stacks and Queues)

소프트웨어학과  
이 의 종



# 스택 (Stacks)

Stacks entered the computer science literature in 1946, when Alan M. Turing used the terms "bury" and "unbury" as a means of calling and returning from subroutines. Subroutines had already been implemented in Konrad Zuse's Z4 in 1945.

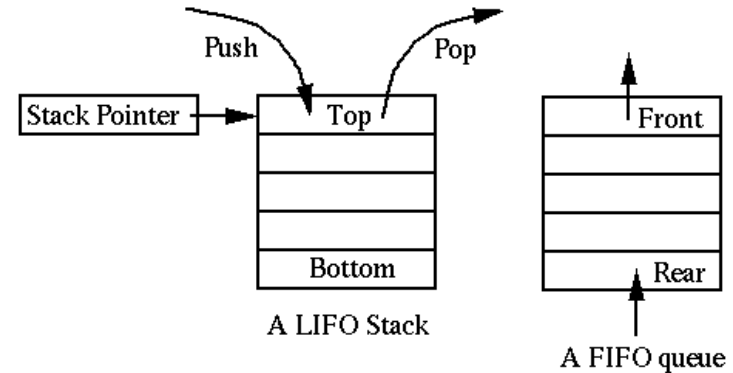


# 스택(1)

- 스택(stack)과 큐(queue)

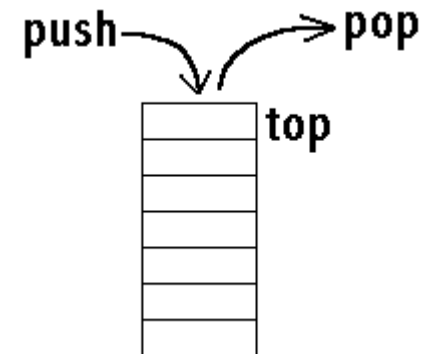
- 순서 리스트(ordered list)의 특별한 경우
  - 순서 리스트:  $A = a_0, a_1, \dots, a_{n-1}, n \geq 0$
  - $a_i$  : 원자(atom) 또는 원소(element)
- 널 또는 공백 리스트 :  $n = 0$ 인 리스트

배열 = 순서리스트



- 스택(stack)

- 탑(top)이라고 하는 한쪽 끝에서 삽입(push)과 삭제(pop)가 일어남
  - 스택  $S = (a_0, \dots, a_{n-1})$ :
  - $a_0$ 는 bottom,  $a_{n-1}$ 은 top의 원소
  - $a_i$ 는 원소  $a_{i-1} (0 < i < n)$ 의 위에 있음
- 후입선출 (LIFO, Last-In-First-Out) 리스트



# 스택(2)

- 원소 A,B,C,D,E를 삽입하고 한 원소를 삭제하는 예

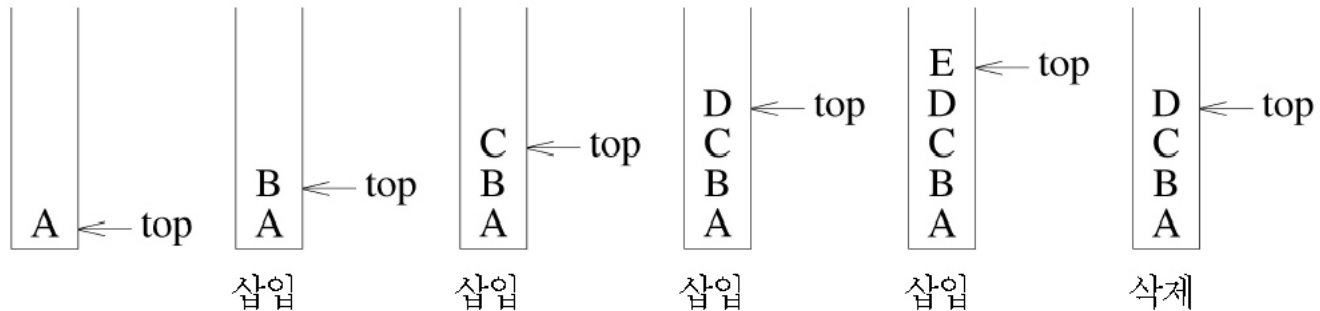
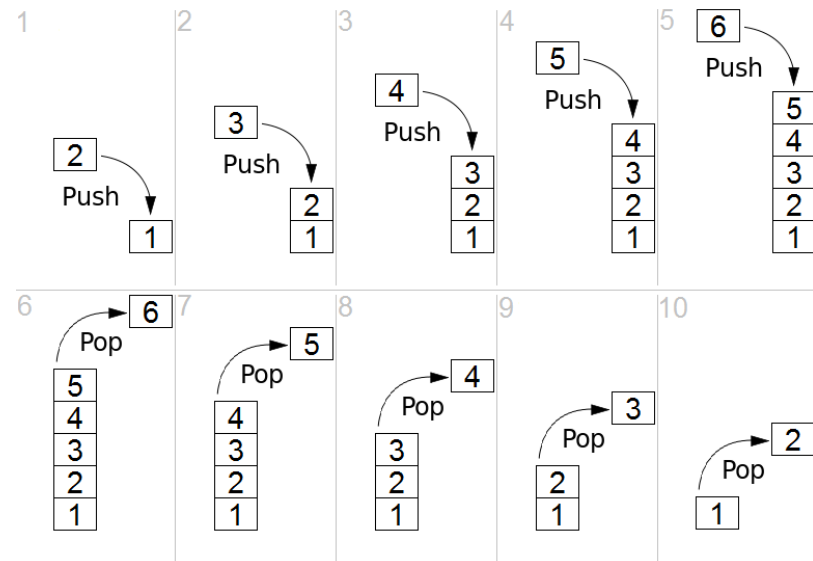


그림 3.1 스택에서의 원소 삽입과 삭제



# 스택(3)

- 시스템 스택(system stack)

- 프로그램 실행 시 함수 호출을 처리
- 함수 호출 시 활성 레코드 (activation record) 또는 스택프레임(stack frame) 이라는 구조를 생성하여 시스템 스택의 탑에 둔다.
  - 이전의 스택프레임(호출한 함수의 스택프레임)에 대한 포인터
  - 복귀 주소 (함수가 종료된 후 실행되어야 할 명령문 위치)
  - 지역 변수 (static 변수 제외)      static → 지역변수지만 한번만 초기화되고 함수 호출 시 글로벌 변수 처럼 사용됨
  - 호출한 함수의 매개 변수
- 함수가 자기자신을 호출하는 순환 호출도 마찬가지로 처리
  - 순환 호출 시마다 새로운 스택 프레임 생성
  - 최악의 경우 가용 메모리 전부 소모

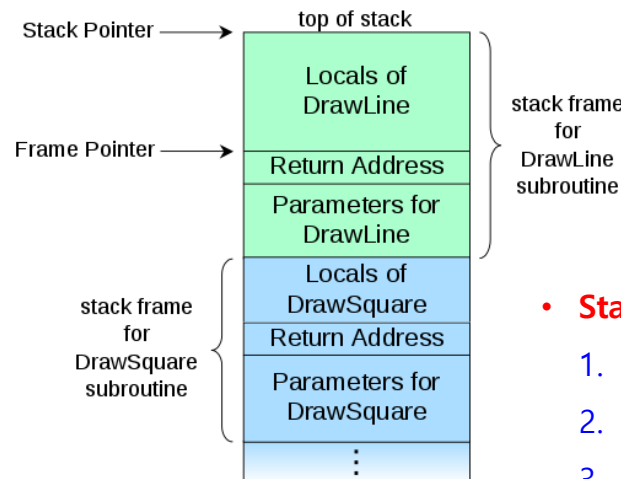
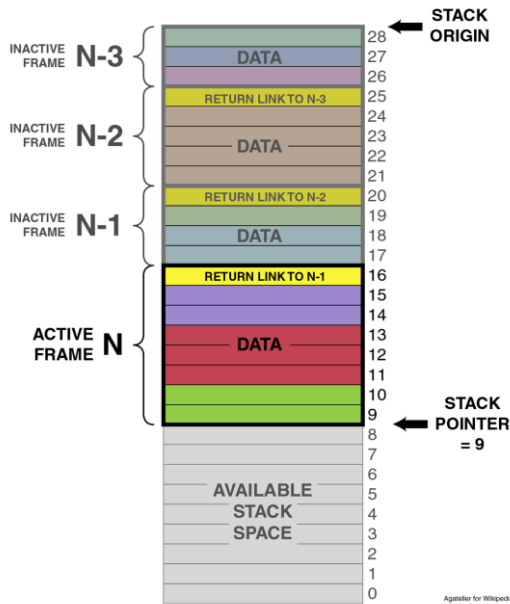
# 스택(3)

- 시스템 스택(system stack)

- 함수 호출 시 스택프레임(stack frame) 생성

- 이전의 스택프레임(호출한 함수의 스택프레임)에 대한 포인터
    - 복귀 주소 (함수가 종료된 후 실행되어야 할 명령문 위치)
    - 지역 변수 (static 변수 제외)
    - 호출한 함수의 매개 변수

- 각각 함수(subroutine)에 대한 스택프레임 생성
  - 프레임포인터(frame pointer)는 현재의 스택프레임에 대한 포인터
  - 스택포인터(stack pointer)는 시스템스택의 Top



stack pointer vs. frame pointer  
stack frame의 크기가 동일하지 않음

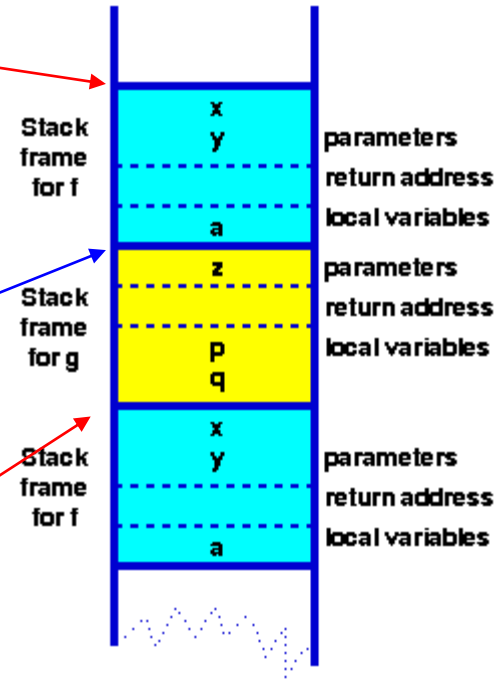
- Stack Frame이 포함하는 것**

- Arguments (parameter values)
- Previous stack frame address
- Return address back to the routine's caller
- Space for local variables of the routine

# 스택(4)

```
function f(int x, int y)
{
    int a;
    if ( term_cond )
        return ...;
    a = .....;
    return g(a);
}
```

```
function g(int z)
{
    int p,q;
    p = ...;
    q = ...;
    return f(p,q);
}
```



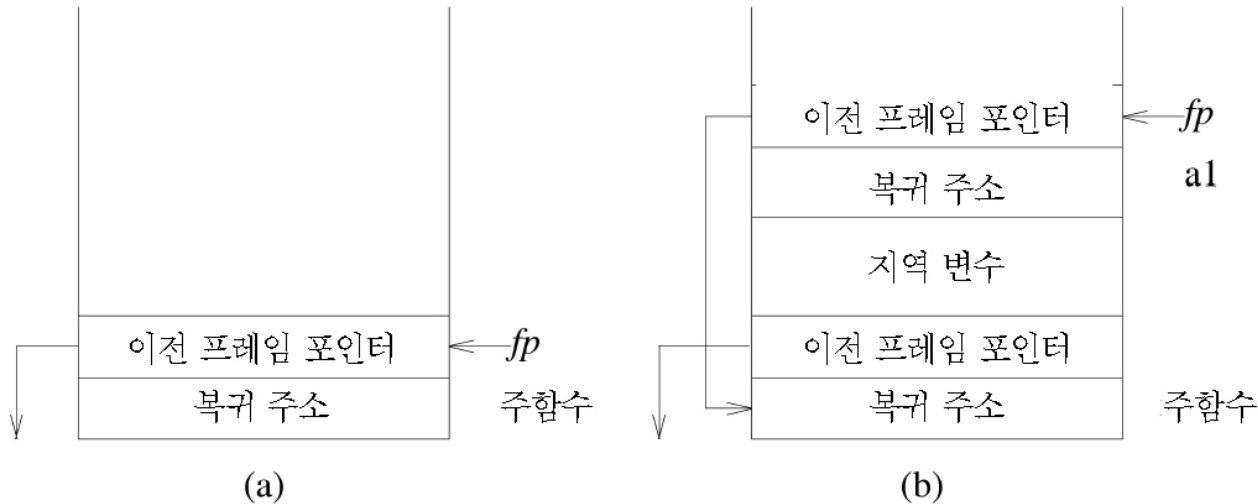
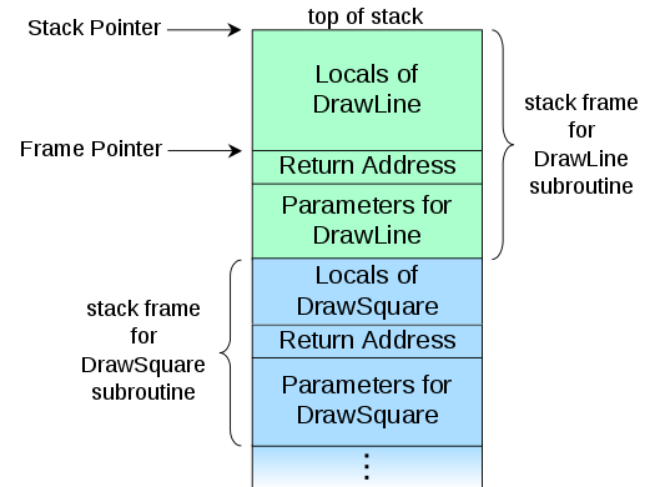
return address → 함수가 종료된 후 돌아가야 할 주소

# 스택(5)

## • 주 함수가 함수 a1을 호출하는 예

- (a) 함수 a1이 호출되기 전의 시스템 스택
- (b) 함수 a1이 호출된 후의 시스템 스택
  - fp : 현재 스택 프레임에 대한 포인터
  - 스택 포인터 sp는 별도로 유지

stack pointer vs. frame pointer  
stack frame의 크기가 동일하지 않음



함수 호출 뒤의 시스템 스택

# 스택(6)

ADT =

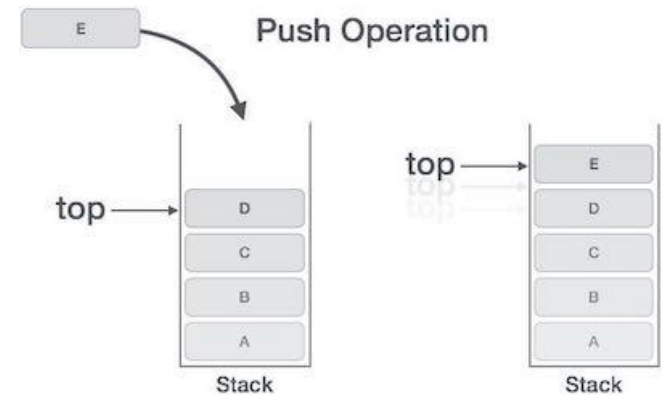
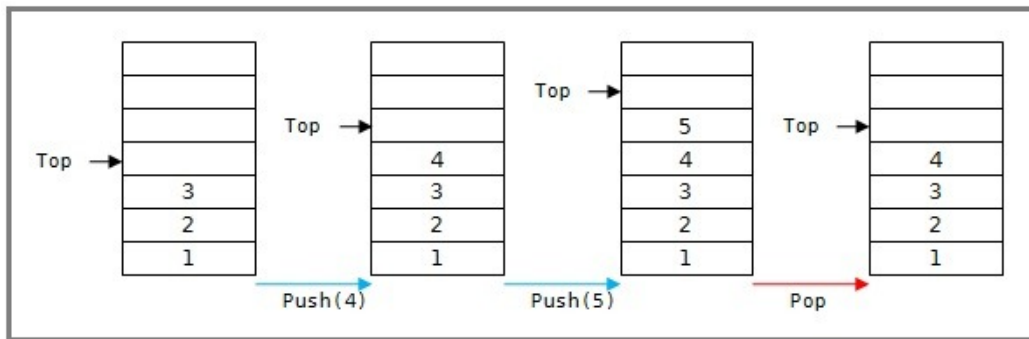
- 1) 객체정의 (데이터표현 대상)
- 2) 객체에 대한 함수 정의

## • 스택 ADT 구현

- 일차원 배열 `stack[MAX_STACK_SIZE]` 사용
- $i$ 번째 원소는 `stack[i-1]`에 저장
- 변수 `top`은 스택의 최상위 원소를 가리킴

(초기 : `top = -1`, 공백 스택을 의미함)

`top`을 어떻게 정의할 것인가?  
→ 구현이 달라짐



# 추상 데이터 타입(1)

ADT =

- 1) 객체정의 (데이터표현 대상)
- 2) 객체에 대한 함수 정의

ADT Stack

objects: 0개 이상의 원소를 가진 유한 순서 리스트

functions:

모든  $stack \in \text{Stack}$ ,  $item \in \text{element}$ ,  $maxStackSize \in \text{positive integer}$

Stack **CreateS**(maxStackSize) ::= 최대 크기가 maxStackSize인 공백 스택을 생성

Boolean **IsFull**(stack, maxStackSize) ::=

if (stack의 원소수 == maxStackSize)  
return TRUE  
else return FALSE

Stack **Push**(stack, item) ::=

if (IsFull(stack)) stackFull  
else stack의 톱에 item을 삽입하고 return

Boolean **IsEmpty**(stack) ::=

if (stack == CreateS(maxStackSize))  
return TRUE  
else return FALSE      stack의 원소 수==0?

Element **Pop**(stack) ::=

if (IsEmpty(stack)) return  
else 스택 톱의 item을 제거해서 반환

# 추상 데이터 타입(2)

- CreateS와 stackFull의 구현

```
Stack CreateS(maxStackSize) ::=
    #define MAX_STACK_SIZE 100 /* 스택의 최대 크기 */
    typedef struct {
        int key;
        /* 다른 필드 */
    } element;
    element stack[MAX_STACK_SIZE];
    int top = -1;

Boolean IsEmpty(Stack) ::= top < 0;
Boolean IsFull(Stack)  ::= top >= MAX_STACK_SIZE-1;
```

```
void stackFull()
{
    fprintf(stderr, "Stack is full, cannot add element");
    exit(EXIT_FAILURE);
}
```

# 추상 데이터 타입(3)

- push와 pop 연산 구현

스택에 원소 삽입

```
void push(element item)
{
    /* 전역 stack에 item을 삽입 */
    if (top >= MAX_STACK_SIZE-1)
        stackFull();
    stack[++top] = item;
}
```

더 이상 추가할 수 없을 때  
← exit() ????

스택으로부터 삭제

```
element pop()
{
    /* stack의 최상위 원소를 삭제하고 반환 */
    if (top == -1)
        return stackEmpty(); /* returns an error key */
    return stack[top--];
}
```

```
void stackFull()
{
    fprintf(stderr, "Stack is full, cannot add element");
    exit(EXIT_FAILURE);
}
```

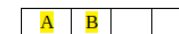
# 동적 배열을 사용하는 스택

## (Stacks using Dynamic Arrays)

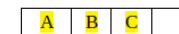
### TIGHT STRATEGY



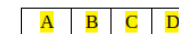
Push(A)



Push(B)



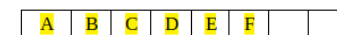
Push(C)



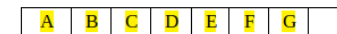
Push(D) (stack is full)



Create new stack  
Push(E)



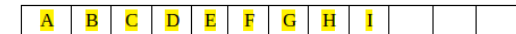
Push(F)



Push(G)



Push(H) (stack is full)



Create new stack  
Push(I)

# 동적 배열을 사용하는 스택 (1)

- 동적배열 이용

- 스택의 범위(MAX\_STACK\_SIZE)를 컴파일 시간에 알아야 하는 단점 극복
- 원소를 위해 동적으로 할당된 배열을 이용
- 필요 시 배열의 크기를 증대시킴

```
Stack CreateS() ::= typedef struct {  
                        int key;  
                        /* 다른 필드 */  
                        } element;  
                        element *stack;  
                        MALLOC(stack, sizeof(*stack));  
                        int capacity = 1;  
                        int top = -1;
```

```
Boolean IsEmpty(Stack) ::= top < 0;
```

```
Boolean isFull(Stack) ::= top >= capacity-1;
```

초기 크기가 1인 동적할당  
배열 stack을 사용



# 동적 배열을 사용하는 스택 (2)

- 스택 만원(stack full)에 대한 새로운 검사 이용
  - MAX\_STACK\_SIZE를 capacity로 대체
  - 함수 push 변경
  - 함수 stackFull 변경
    - 배열 stack의 크기를 확장시켜서 스택에 원소를 추가로 삽입할 수 있게 함
    - 배열 배가(array doubling): 배열의 크기를 늘릴 필요가 있을 시 항상 배열의 크기를 두 배로 만든다.

REALLOC MACRO (Page 62)

```
void stackFull()
{
    REALLOC(stack, 2*capacity*sizeof(*stack))
    capacity *= 2;
}
```

배열 배가를 사용하는 stackFull

# 동적 배열을 사용하는 스택 (3)

Memory 주소 확인

```
void stackFull()
{
    REALLOC(stack, 2*capacity*sizeof(*stack))
    capacity *= 2;
}
```

\*realloc() 함수는 값을 복사하지는 않음

- Time Complexity :  $O(n)$

- $\text{realloc}() \rightarrow 2 \times \text{capacity} \times \text{sizeof}(*\text{stack})$  바이트의 메모리 할당  $O(1)$
- $\text{capacity} \times \text{sizeof}(*\text{stack})$  바이트를 새로운 배열로 복사  $O(\text{capacity})$
- time complexity of array doubling =  $O(\text{capacity})$
- $\text{capacity} = 1 \rightarrow 2^k$  ( $k > 0$ )  $\leftarrow$  array doubling에 의해
- $\text{capacity} = 2^k \rightarrow$  array doubling time complexity =  
 $O(\sum_{i=1}^k 2^i) = O(2^{k+1}) = O(2^k)$

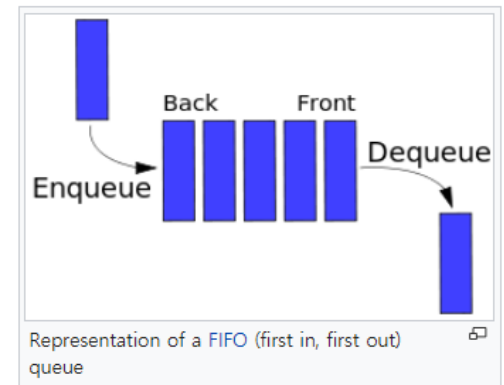
**capacity  $\rightarrow 2^k$**

**전체의 삽입의 수(n)  $> 2^{k-1}$**

# 큐(Queues)

In computer science, a queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and removal from the other end of the sequence. By convention, the end of the sequence at which elements are added is called the back, tail, or rear of the queue and the end at which elements are removed is called the head or front of the queue, analogously to the words used when people line up to wait for goods or services.

Queue		
Time complexity in big O notation		
Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert	$O(1)$	$O(1)$
Delete	$O(1)$	$O(1)$



# 큐(Queue)의 개념

- 한쪽 끝에서 삽입이 일어나고 그 반대쪽 끝에서 삭제가 일어남
- 순서 리스트
  - 새로운 원소가 삽입되는 끝 - rear
  - 원소가 삭제되는 끝 - front
- 선입선출(FIFO, First-In-First-Out) 리스트
- 원소 A, B, C, D, E를 순서대로 큐에 삽입하고 한 원소를 삭제하는 예

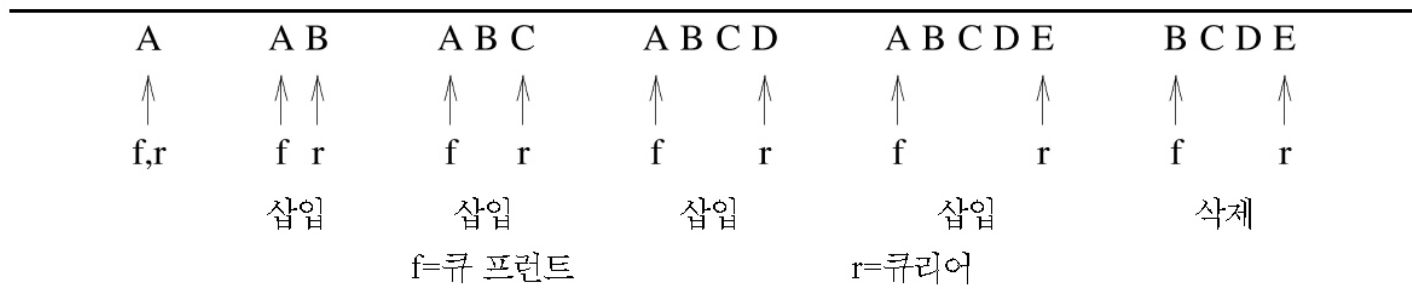
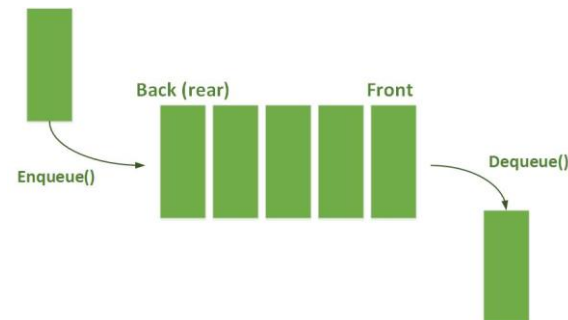


그림 3.4 큐에서 원소의 삽입과 삭제



# 추상 데이터 타입 (1)

>- `return_test.c`

ADT Queue

objects: 0개 이상의 원소를 가진 유한 순서 리스트

functions:

모든 queue Queue, item element, maxQueueSize positive integer

Queue **CreateQ**(maxQueueSize) ::= 최대 크기가 maxQueueSize인 공백 큐를 생성

Boolean **IsFullQ**(queue, maxQueueSize ) ::=  
if (queue의 원소 수 == maxQueueSize) return TRUE  
else return FALSE

Queue **AddQ**(queue, item) ::= if (IsFull(queue)) queueFull  
else queue의 뒤에 item을 삽입하고 이 queue를 반환

Boolean **IsEmptyQ**(queue) ::= if (queue == CreateQ(maxQueueSize)) return TRUE  
else return FALSE      **queue의 원소의 수 == 0**

Element **DeleteQ**(queue) ::= if (IsEmpty(queue)) return  
else queue의 앞에 있는 item을 제거해서 반환

# 추상 데이터 타입 (2)

- 큐를 순차 기억 장소로 표현

- 1차원 배열과 두 변수 front, rear 필요

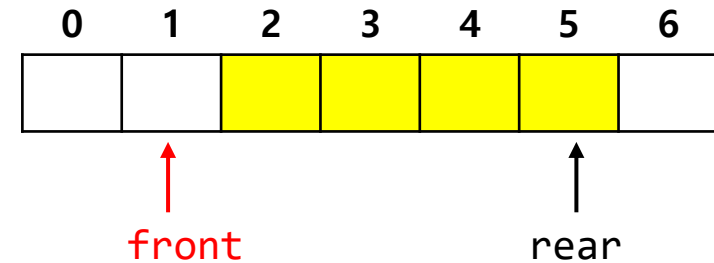
- CreateQ, IsEmptyQ, IsFullQ의 구현

- **CreateQ, IsEmptyQ, IsFullQ**

```
Queue CreateQ(maxQueueSize) ::=
    #define MAX_QUEUE_SIZE 100 /* 큐의 최대 크기 */
    typedef struct {
        int key;
        /* 다른 필드 */
    } element;
    element queue[MAX_QUEUE_SIZE];
    int rear = -1;
    int front = -1;

Boolean IsEmptyQ(queue) ::= front == rear

Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1
```



rear가 앞으로 갈 방법이  
있어야 함

# 추상 데이터 타입 (3)

- **addq와 deleteq 연산의 구현**
  - 스택의 push, pop과 유사
  - 스택에서는 push와 pop 모두 top 변수를 사용
  - 큐에서는 addq는 rear를 증가, deleteq는 front를 증가

- **addq, deleteq**

큐에서의 삽입

```
void addq(element item)
{
    /* queue에 item을 삽입 */
    if (rear == MAX_QUEUE_SIZE-1)
        queueFull();
    queue[++rear] = item;
}
```

큐에서의 삭제

```
element deleteq()
{
    /* queue의 앞에 있는 원소를 삭제 */
    if (front == rear)
        return queueEmpty(); /* returns an error key */
    return queue[++front];
}
```

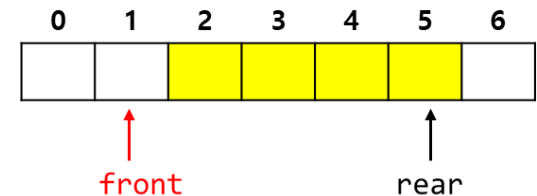
# 큐의 응용 분야

## • 작업 스케줄링 (job scheduling)

- 운영체제에 의한 작업 큐 (job queue) 생성      ← Queue의 대표적 예
- 시스템에 들어간 순서대로 처리      ← priority를 사용하지 않을 경우
- 큐를 순차 표현으로 구현할 때의 문제점

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				job 1 is added
-1	1	J1	J2			job 2 is added
-1	2	J1	J2	J3		job 3 is added
0	2		J2	J3		job 1 is deleted
1	2			J3		job 2 is deleted

← 순차 큐에서의  
삽입과 삭제

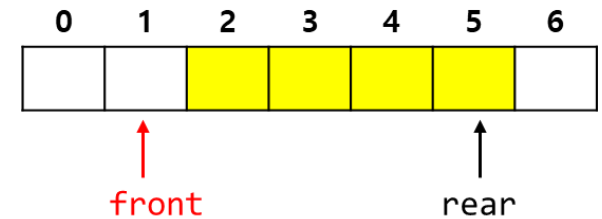


- 큐는 점차 오른쪽으로 이동
- 결국, rear 값이 MAX\_QUEUE\_SIZE-1과 같아짐 (queue full)

# 순차 큐의 문제점

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				job 1 is added
-1	1	J1	J2			job 2 is added
-1	2	J1	J2	J3		job 3 is added
0	2		J2	J3		job 1 is deleted
1	2			J3		job 2 is deleted

← 순차 큐에서의  
삽입과 삭제



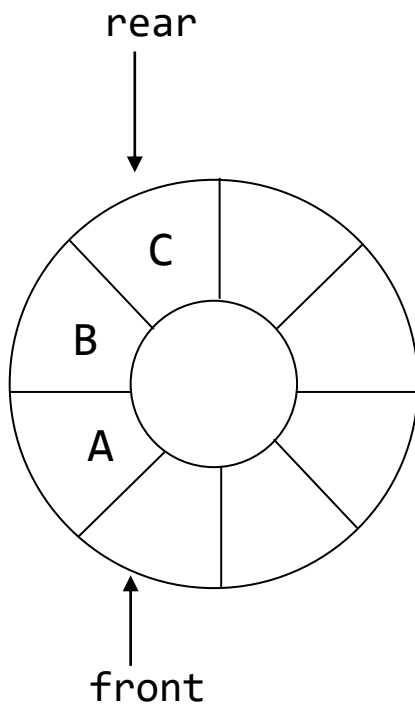
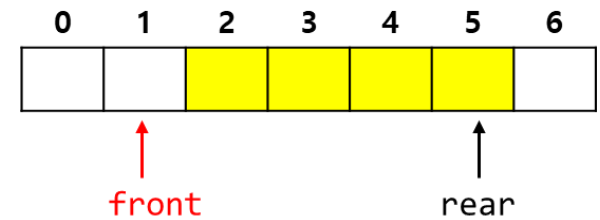
## • 문제점 해결 → 빈공간 찾아 이동 → 비용이 많이 발생

- queueFull은 전체 큐를 왼쪽으로 이동시켜야 함
- 이동 시 첫 번째 원소가 q[0]에 오도록 조정
- front = -1로 조정, rear도 다시 조정
- 배열 이동 시간이 많이 든다 → 최악의 경우 복잡도 :  
 $O(\text{MAX\_QUEUE\_SIZE})$

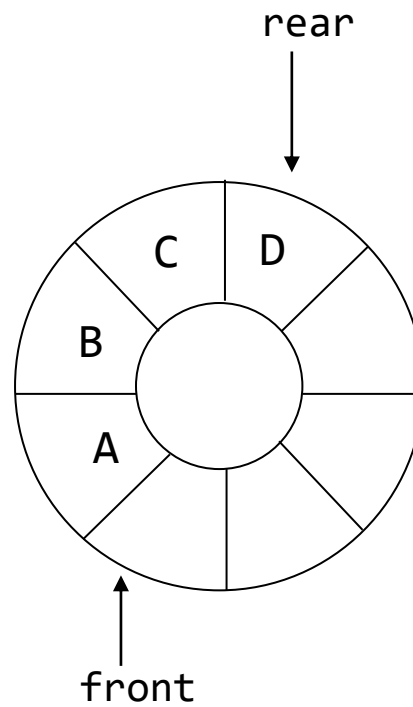
Queue를 원형으로 본다면, 효율성을 높일 수 있음

# 원형 큐(Circular Queue)-(1)

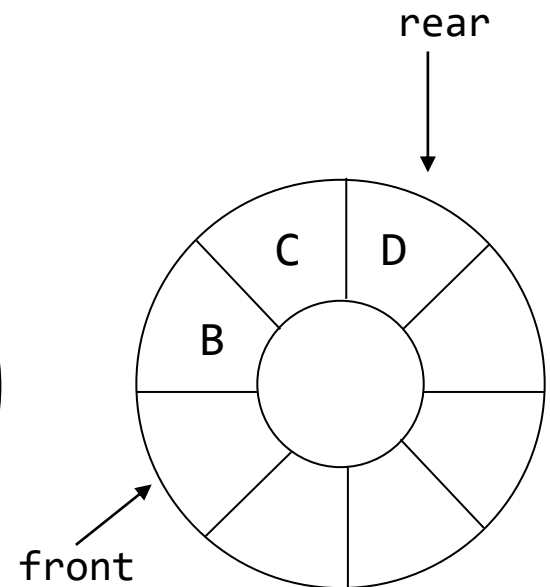
- 큐가 배열의 끝을 둘러싸도록 함
- 변수 front는 큐에서 첫 원소의 위치로부터 시계 시계방향으로 하나 앞 위치를 가리킴



(a) 초기



(b) 삽입



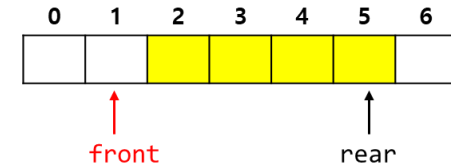
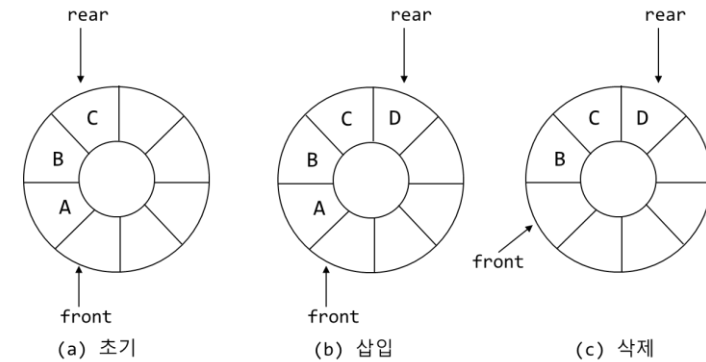
(c) 삭제

# 원형 큐(Circular Queue)-(2)

## - 위치

- $\text{MAX\_QUEUE\_SIZE}-1$ 의 다음 위치 = 0
- 0의 뒤 위치 =  $\text{MAX\_QUEUE\_SIZE}-1$

## - 원형 큐의 동작을 위해 front와 rear를 다음위치(시계 방향) 으로 이동



```
if(rear == MAX_QUEUE_SIZE-1) rear = 0;      ← modulo 연산
else rear++; /* (rear+1) % MAX_QUEUE_SIZE와 동등*/
```

## - 큐의 공백 조건 : $\text{front} == \text{rear}$

## - 큐의 포화(full)와 공백(empty)을 구분하기 위해 만원이 되기 전에 큐의 크기를 증가시킨다.

- $\text{front} == \text{rear}$ 가 포화인지 공백인지 구분하기 위해, 최대 원소 수를  $\text{MAX\_QUEUE\_SIZE}$ 가 아니라  $\text{MAX\_QUEUE\_SIZE} - 1$ 로 한다

queue가 empty일 때와 full 일 때 구별해야 함 → 공간 하나를 비워 둠

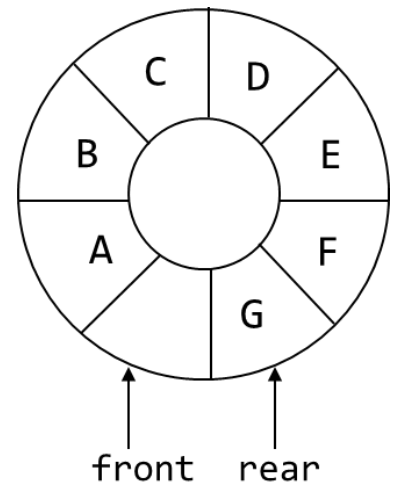
# 원형 큐(Circular Queue)-(3)

- 멤버 함수 addq와 deleteq의 구현

```
void addq(element item)
{ /* queue에 item을 삽입 */
    rear = (rear+1)%MAX_QUEUE_SIZE;           ← modulo 연산 = 나머지 값
    if (front == rear)
        queueFull(); /* error를 프린트하고 끝낸다 */
    queue[rear] = item;                       ← rear 값은 이미 증가됨
}
```

```
element deleteq()
{ /* queue의 앞 원소를 삭제 */
    element item;
    if (front == rear)
        return queueEmpty(); /* 에러 키를 반환한다 */
    front = (front+1) % MAX_QUEUE_SIZE;
    return queue[front];
}
```

실제 값에 대한 삭제 x



- addq()와 delete() 함수에서 full과 empty를 검사하는 로직이 같음(front==rear)
- queue가 full - 1일 경우, 다음 addq() 호출 시 front==rear가 됨

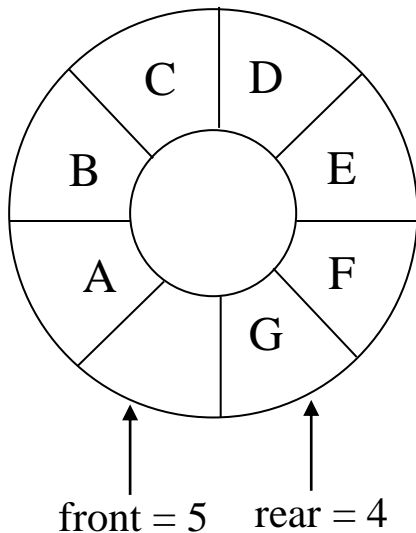
# **동적할당 배열을 이용하는 원형 큐**

**(Circular Queues using Dynamically Allocated Arrays)**

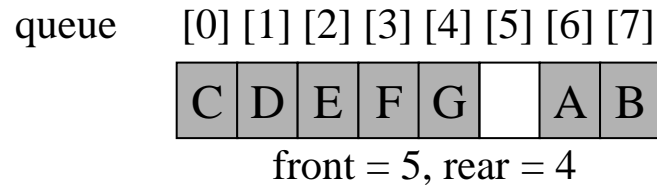
# 동적 할당 배열을 이용하는 원형 큐(1)

## • 동적 할당 배열 사용

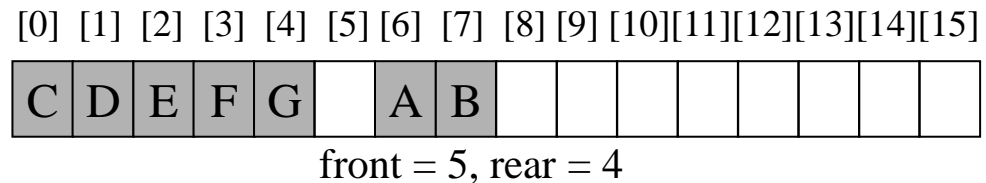
- capacity : 배열 queue의 위치 번호
- 원소 삽입 시, 배열 크기 확장해야 함
  - 함수 realloc
  - 배열 배가 방법 사용 (array doubling)



(a) 만원이 된 원형 큐



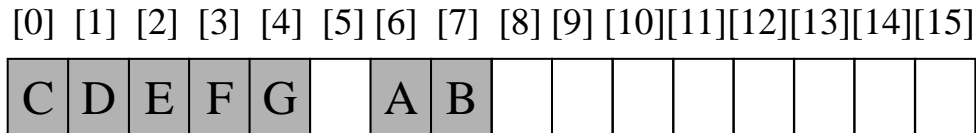
(b) 만원이 된 원형 큐를 펼쳐 놓은 모양



(c) 배열을 두 배로 확장한 뒤의 모양  
(realloc을 이용하여 확장)

(c)는 원형큐에 맞게 재구성되어야 함

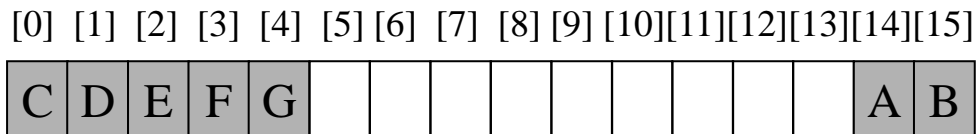
# 동적 할당 배열을 이용하는 원형 큐(2)



front = 5, rear = 4

(c) 배열을 두 배로 확장한 뒤의 모양  
(realloc을 이용하여 확장)

원형 queue를 얻기  
위해 A,B를 이동

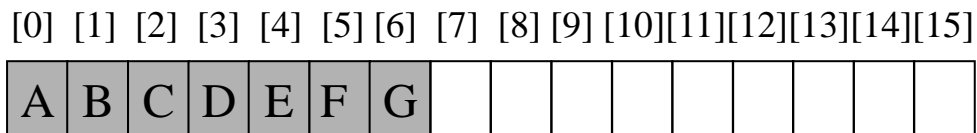
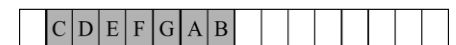


front = 13, rear = 4

(d) 세그먼트를 오른쪽으로 이동한 뒤의 모양

복사한 후 이동

← 최대 복사 원소 수:  
 $2 * \text{capacity} - 2$



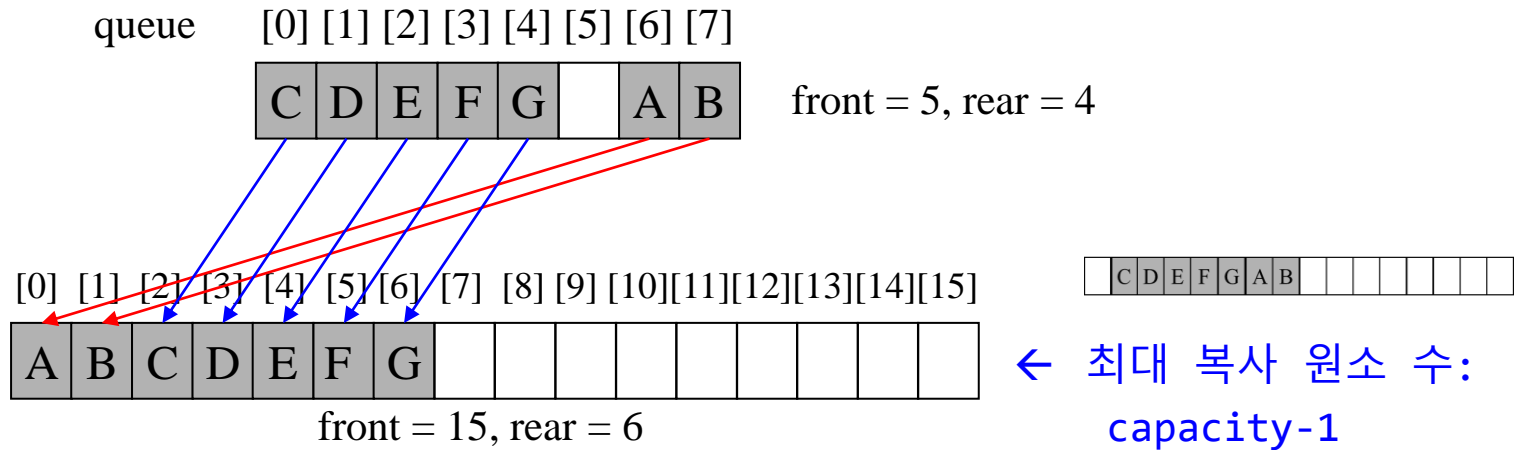
front = 15, rear = 6

(e) 같은 내용의 다른 모양

복사 전 이동

← 최대 복사 원소 수:  
 $\text{capacity} - 1$

# 동적 할당 배열을 이용하는 원형 큐(3)



(e) 같은 내용의 다른 모양

## • 최대 복사 원소 수: $capacity - 1$

- 1) 크기가 두 배 되는 새로운 배열 `newQueue`를 생성한다.
- 2) 두 번째 부분(즉, `queue[front+1]`과 `queue[capacity-1]` 사이에 있는 원소들)을 `newQueue`의 0에서부터 복사해 넣는다.
- 3) 첫 번째 부분(즉 `queue[0]`와 `queue[rear]`사이에 있는 원소들)을 `newQueue`의 `capacity-front-1`에서부터 복사해 넣는다.

# 동적 할당 배열을 이용하는 원형 큐(4)

- addq(), queueFull()

```
void addq(element item)
{
    /* queue에 item을 삽입 */
    rear = (rear + 1) % MAX_QUEUE_SIZE;    ← rear = rear + 1
    if (front == rear)
        queueFull();    /* double capacity */
    queue[rear] = item;
}
```

동적 할당 배열을 이용하는 원형 큐에 삽입

# 동적 할당 배열을 이용하는 원형 큐(5)

```
void queueFull()
```

```
{
```

```
    /* 2배 크기의 배열을 할당 */
```

```
    element* newQueue;
```

```
    MALLOC(newQueue, 2*capacity*sizeof(*queue));
```

```
    /* queue로부터 newQueue로 복사 */
```

```
    int start = (front+1) % capacity;
```

```
    if (start < 2)
```

```
        /* 둘러싸지 않음 */
```

```
        copy(queue+start, queue+start+capacity-1, newQueue);
```

```
    else
```

```
        /* 큐가 주변을 둘러쌘 */
```

```
        copy(queue+start, queue+capacity, newQueue);
```

```
        copy(queue, queue+rear+1, newQueue+capacity-start);
```

```
    }
```

```
    /* newQueue로 전환 */
```

```
    front = 2*capacity-1;
```

```
    rear = capacity-2;
```

```
    capacity *= 2;
```

```
    free(queue);
```

```
    queue = newQueue;
```

```
}
```

큐의 크기를 2배로 확장

← start = 실제 값이 시작하는 곳

← rear > front 경우

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	D	E	F	G	C	A	B

← rear < front 경우

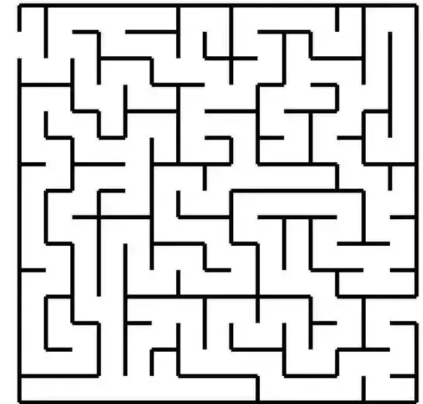
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
D		E	F	G	C	A	B

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
C	D	E	F	G		A	B

front = 0 경우

copy(a,b,c) = 위치 a에서 b-1까지 원소를  
c에서 시작하는 위치로 복사

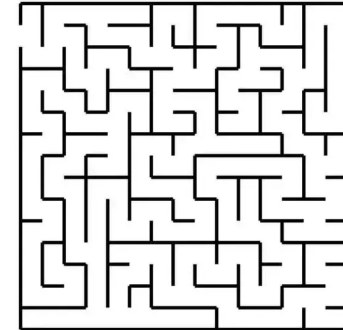
# 미로문제 (A Mazing Problem)



# 미로 문제(1)

- 미로(maze)는  $1 \leq i \leq m$ 이고  $1 \leq j \leq p$ 인 이차원 배열  $\text{maze}[m][p]$ 로 표현

- 1 : 통로가 막힌 길
- 0 : 통과할 수 있는 길



입구  
 $\text{maze}[1][1]$

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

출구  
 $\text{maze}[m][p]$

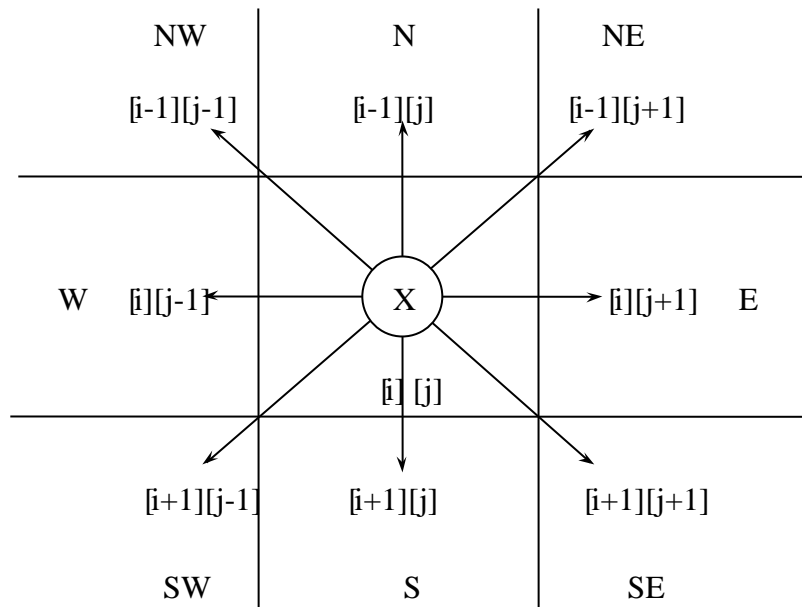
미로속의 주의 위치  $x = \text{maze}[\text{row}][\text{col}]$

경계조건을 어떻게 찾을 것인가?

예제 미로 (경로를 찾을 수 있는가?)

# 미로 문제(2)

- 현재의 위치  $x$  :  $\text{maze}[i][j]$



가능한 이동

모든 위치가 8개의 인접위치를 갖는 것이 아님

- $i=1$ 이거나  $m$  또는  $j=1$ 이거나  $p$ 인 경계선에 있는 경우 가능한 방향은 8방향이 아니라 3방향만 존재
- 경계 조건을 매번 검사하는 것을 피하기 위해 미로의 주위를 1로 둘러쌘
- 배열은  $\text{maza}[m+2][p+2]$ 로 선언되어야 함

배열(maze)의 크기를 늘리는 대신 이동할 수 있는 방향에 대한 정보를 이용

# 미로 문제(3)

- 이동할 수 있는 방향들을 배열 move에 미리 정의
  - 여덟 개의 가능한 이동 방향 : 0 ~ 7의 숫자로 표시

```
typedef struct {  
    short int vert;  
    short int horiz;  
} offsets;  
  
offsets move[8]; /* 각 방향에 대한 이동 배열 */
```

북쪽(N)으로 이동할 경우,  
현재위치의 vert값을 1감  
소 시킴

Name	Dir	move[dir].vert	move[dir].horiz
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

- 현재 위치 : maze[row][col]
- 다음 이동할 위치 :  
    maze[nextRow][nextCol]

- nextRow = row + move[dir].vert;
- nextCol = col + move[dir].horiz;

# 미로 문제(4)

- 길을 찾아가는 방법

- 미로 이동 시, 현 위치 저장 후 방향 선택
- 잘못된 길을 갔을 때 되돌아와서 다른 방향을 시도할 수 있게 함
- 시도했던 길을 2차원 배열 mark에 기록해서 유지
  - 한번 시도한 경로의 재시도 방지 위해 사용
  - `maze[row][col]` 방문  $\rightarrow$  `mark[row][col] = 1`

이동할 때마다 현재 위치를 스택에 저장(push)  $\rightarrow$   
잘못된 길을 만날 때 Roll back (pop)

# 미로 문제(5)

## 개념적인 미로 알고리즘

```
initialize a stack to the maze's entrance coordinates and direction to north;
while (stack is not empty) {
    /* 스택의 탑에 있는 위치로 이동*/
    <row, col, dir> = pop from top of stack;    ← 실패해서 이전 위치로 되돌아옴

    while (there are more moves from current position) {
        <nextRow, nextCol> = coordinate of next move;    ← 다음 가능한 위치 가져옴
        dir = direction of move;                        ← 북쪽부터 시계방향

        if ((nextRow == EXIT_ROW) && (nextCol == EXIT_COL))    ← 성공
            success;

        if (maze[nextRow][nextCol] == 0) && mark[nextRow][nextCol] == 0) {
            /* 가능하지만 아직 이동해보지 않은 이동 방향 */    ← 방문하지 않았고, 이동도 가능
            mark[nextRow][nextCol] = 1;

            /* 현재의 위치와 방향을 저장 */
            push<row, col, dir> to the top of the stack;
            row = nextRow;
            col = nextCol;
            dir = north;    ← default 값
        }
    }
}
printf("No path found\n");
```

# 미로 문제(6)

- 초기 알고리즘에 추가적으로 필요한 것
  - 스택에 들어갈 원소(item)은 현재위치 + 방향정보가 포함
  - 스택이 full 일 때 확장 (array doubling)
    - 미로의 각 위치는 기껏해야 한번씩 방문 → 스택의 크기는 미로의 0의 개수만큼이면 충분
    - $m \times p$  크기의 미로라고 할 때, 스택의 최대크기는  $m \times p$

element 재정의

```
typedef struct {  
    short int row;  
    short int col;  
    short int dir;  
}element;
```

0	0	0	0	0	1
1	1	1	1	1	0
1	0	0	0	0	1
0	1	1	1	1	1
1	0	0	0	0	1
1	1	1	1	1	0
1	0	0	0	0	1
0	1	1	1	1	1
1	0	0	0	0	0

긴 경로를 가진 미로 예

# 미로 문제(7)

Time Complexity:  $O(mp)$

```
void path(void)
{
    /* 미로를 통과하는 경로가 있으면 그 경로를 출력한다. */
    int i, row, col, nextRow, nextCol, dir, found=FALSE;
    element position;
    mark[1][1]=1; top=0;
    stack[0].row=1; stack[0].col=1; stack[0].dir=1;
    while (top > -1 && !found) {
        position = pop(&top);          ← 이전 위치
        row = position.row;           col = position.col;   dir = position.dir;

        while (dir < 8 && !found) {
            /* dir 방향으로 이동 */
            nextRow = row + move[dir].vert;
            nextCol = col + move[dir].horiz;
            if (nextRow==EXIT_ROW && nextCol==EXIT_COL)
                found = TRUE;
            else if ( !maze[nextRow][nextCol] && !mark[nextRow][nextCol]) {
                mark[nextRow][nextCol] = 1;
                position.row = row; position.col = col;
                position.dir = ++dir;
                // 이전에 방문하지 않았고, 갈수 있다
                add(&top, position);          ← 현재위치 저장
                row = next.row; col = next.col; dir = 0; ← 새로운 곳에서는 북쪽(N)부터
            }
            else ++dir;
        }
    }
    /* 경로를 찾았을 때 경로를 프린트하는 코드 생략 */
}
```

# 수식의 계산

## (Evaluation of Expressions)

$$+ \ 3 \ 4$$

Postfix notation  
("Reverse Polish")

Infix notation

**Prefix notation**  
("Polish")

V • T • E

$$3 \ + \ 4$$

Postfix notation  
("Reverse Polish")

**Infix notation**

Prefix notation  
("Polish")

V • T • E

$$3 \ 4 \ +$$

**Postfix notation**  
("Reverse Polish")

Infix notation

Prefix notation  
("Polish")

V • T • E

# 수식(Expressions)

- 복잡한 수식

- $(\text{rear}+1==\text{front}) || ((\text{rear}==\text{MAX\_QUEUE\_SIZE}-1) \&\& !\text{front})) \text{ -- (1)}$

- 복잡한 지정문

- $x = a/b - c + d*e - a*c \text{ -- (2)}$

- 식 (1)은 피연산자(rear, front), 연산자(==, ||) 및 괄호로 구성

- 수식이나 명령문의 의미를 이해하기 위해 연산의 수행 순서를 파악

- $a=4, b=c=2, d=e=3$ 일 때 명령문 (2)에서  $x$ 값을 계산

- $((4/2)-2)+(3*3)-(4*2) = 0+9-8 = 1 \quad (o)$

- $(4/(2-2+3))*(3-4)*2 = (4/3)*(-1)*2 = -2.66666\dots \quad (x)$

- 연산자의 연산 순서를 결정하는 우선순위 계층(precedence hierarchy)  
를 가지고 있음

# 우선순위

## C 언어의 우선순위 계층

Programming

`c = a+++b;`

`>_ precedence.c`

토큰	연산자	우선순위	결합성
() [] →	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement	16	left-to-right
-- ++	decrement, increment	15	right-to-left
! - - + & * sizeof	logical not one's complement unary minus or plus address or indirection size (in bytes)		
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^=  =	assignment	2	right-to-left
,	comma	1	left-to-right

# 후위 표기식의 연산(1)

- 수식을 표기하는 표준방식 → 중위표기법(infix notation)
  - 두피연산자(operand) 사이에 이항 연산자(binary operator)가 위치
  - 인간에게는 보편적 방식, 하지만 컴퓨터는 그렇지 못함
- 후위 표기법(postfix notation)
  - 연산자(operator)가 피연산자(operand) 이후에 위치
  - 괄호를 사용하지 않음

3 + 4

Postfix notation  
("Reverse Polish")

Infix notation

Prefix notation  
("Polish")

V • T • E

중위 표기	후위 표기
2+3*4	2 3 4*+
a*b+5	ab*5+
(1+2)*7	1 2+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

Operand1    Operand2    Operator

ab+

Prefix: + a b

Postfix: a b +

# 후위 표기식의 연산(2)

## • 후위 표기식의 연산:

- 수식을 왼쪽에서 오른쪽으로 스캔
- 연산자를 만날 때까지 피연산자를 스택에 저장
- 연산자를 만나면 연산에 필요한 만큼의 피연산자를 스택에서 pop
- 가져온(pop) 피연산자를 연산
- 연산 결과를 다시 스택에 저장(push)
- 식의 끝에 도달할 때까지 위의 과정 반복
- 스택의 톱에서 해답을 가져온다

6/2-3+4\*2



62/3-42\*+

postfix

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

stack의 가장 아래에  
결과가 저장되어 있음

# 후위 표기식의 연산(3)

## • Example:

- +, -, \*, /, % 이항연산자
- 한자리 정수로 된 피연산자 → 숫자(입력은 문자, eg: 7)를 ASCII 문자 값 48을 빼면, 숫자를 얻을 수 있음
- enum 타입으로 연산에 대한 우선순위를 정의

DEC	HEX	OCT	CHAR
0	0x00	000	NUL(null)
1	0x01	001	SOH(start of heading)
2	0x02	002	STX(start of text)
3	0x03	003	ETX(end of text)
4	0x04	004	EOI(end of transmission)
5	0x05	005	ENQ(enquiry)
6	0x06	006	ACK(acknowledge)
7	0x07	007	BEL(bell)
8	0x08	010	BS(backspace)
9	0x09	011	HT(horizontal tab)
10	0x0A	012	LF(LF, line feed, new line)
11	0x0B	013	VT(vertical tab)
12	0x0C	014	FF(NP from feed, new page)
13	0x0D	015	CR(carrriage return)
14	0x0E	016	SO(shift out)
15	0x0F	017	SI(shift in)
16	0x10	020	DLE(data link escape)
17	0x11	021	DC1(device control 1)
18	0x12	022	DC2(device control 2)
19	0x13	023	DC3(device control 3)
20	0x14	024	DC4(device control 4)
21	0x15	025	NAK(negative acknowledge)
22	0x16	026	SYN(synchronous idle)
23	0x17	027	ETB(end of trans. block)
24	0x18	030	CAN(cancel)
25	0x19	031	EM(end of medium)
26	0x1A	032	SUB(substitute)
27	0x1B	033	ESC(escape)
28	0x1C	034	FS(file separator)
29	0x1D	035	GS(group separator)
30	0x1E	036	RS(record separator)
31	0x1F	037	US(unit separator)

32	0x20	040	SP(space)
33	0x21	041	!
34	0x22	042	"
35	0x23	043	#
36	0x24	044	\$
37	0x25	045	%
38	0x26	046	&
39	0x27	047	'
40	0x28	050	(
41	0x29	051	)
42	0x2A	052	*
43	0x2B	053	+
44	0x2C	054	,
45	0x2D	055	-
46	0x2E	056	.
47	0x2F	057	/
48	0x30	060	0
49	0x31	061	1
50	0x32	062	2
51	0x33	063	3
52	0x34	064	4
53	0x35	065	5
54	0x36	066	6
55	0x37	067	7
56	0x38	070	8
57	0x39	071	9

```
scanf("%c", &n);
n = '7' (0x37) = 55
```

정수 n = 55 - 48

```
typedef enum {
    lparen, rparen,
    plus, minus, times,
    divide, mod, eos,
    operand} precedence;
```

ASCII (American Standard Code for Information Interchange)  
ANSI 라는 미국 표준 협회에서 제정한 문자 표현 방식

```

int eval(void)
{
    /* 전역 변수로 되어있는 후위 표기식 expr을 연산한다. '\0'은 수식의 끝을 나타낸다.
    stack과 top은 전역 변수이다. 함수 getToken은 토큰의 타입과 문자 심벌을 반환한다.
    피연산자는 한 문자로 된 숫자임을 가정한다. */
    precedence token;
    char symbol;
    int op1,op2;
    int n = 0; /* 수식 스트링을 위한 카운터 */ ← index
    int top = -1;
    token = getToken(&symbol, &n); ← 문자와, 인덱스
    while (token != eos) {
        if (token == operand) ← 피연산자, 즉 숫자
            push(symbol-'0'); / *스택 삽입 */ ← 문자 - 48
        else {
            /* 두 피연산자를 삭제하여 연산을 수행한 후, 그 결과를 스택에 삽입함 */
            op2 = pop(); /* 스택 삭제 */
            op1 = pop();
            switch(token) {
                case plus: push(op1+op2); break;
                case minus: push(op1-op2); break;
                case times: push(op1*op2); break;
                case divide: push(op1/op2); break;
                case mod: push(op1%op2);
            }
        }
        token = getToken(&symbol, &n);
    }
    return pop(); /* 결과를 반환 */
}

```

return 값  
precedence

n	Token	[0]	Stack [1]	[2]	Top
0→	6	6			0
1→	2	6	2		1
2→	/	6/2			0
3→	3	6/2	3		1
-	4	6/2-3			0
2	4	6/2-3	4		1
*	2	6/2-3	4	2	2
...	*	6/2-3	4*2		1
	+	6/2-3+4*2			0

# 후위 표기식의 연산(5)

- 보조 함수 getToken

- 수식 스트링으로부터 토큰을 얻기 위해 사용

```
precedence getToken(char *symbol, int *n)
{
    /* 다음 토큰을 취한다. symbol은 문자 표현이며,
       token은 그것의 열거된 값으로 표현되고, 명칭으로 반환된다. */

    *symbol = expr[(*n)++];      ← 스택이 배열로 구현되어 있음

    switch (*symbol) {
        case '(' : return lparen;
        case ')' : return rparen;
        case '+' : return plus;
        case '-' : return minus;
        case '/' : return divide;
        case '*' : return times;
        case '%' : return mod;
        case ' ' : return eos;
        default  : return operand; /* 에러 검사는 하지 않고 기본 값은 피연산자 */
    }
}
```

입력 스트링으로부터 토큰을 가져오는 함수

# 중위 표기에서 후위 표기로의 변환(1)

- 중위 표기식을 후위 표기식으로 변환하는 알고리즘
  - 식을 모두 괄호로 묶는다.
  - 이항 연산자들을 모두 그들 오른쪽 괄호와 대체시킨다.
  - 모든 괄호를 삭제한다.

2 pass algorithm

1. 괄호 묶기
2. 괄호와 연산자 대체

$a/b - c + d * e - a * c$

$((((a/b) - c) + (d * e)) - (a * c))$

연산자의 출력은  
우선순위에 의해  
결정 되어야 함


$((((a/b) - c) + (d * e)) - (a * c))$

$ab/c - de * + ac * -$

Note : 피연산자의 순서는 불변

# 중위 표기에서 후위 표기로의 변환(2)

- 예제 3.3:  $a+b*c$ 로부터  $abc*+$ 를 생성
  - 우선순위가 높은 연산자는 낮은 연산자보다 먼저 출력
  - 입력 연산자가 스택의 탑에 있는 연산자보다 우선순위가 높으면 스택에 삽입



Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

# 중위 표기에서 후위 표기로의 변환(3)

- 예제 3.4: [괄호가 있는 수식]

$a*(b+c)*d$  (infix)  $\rightarrow$   $abc+*d*$  (postfix)

	Token	Stack			Top	Output
		[0]	[1]	[2]		
	a				-1	a
	*	*			0	a
	(	*	(		1	a
	b	*	(		1	ab
	+	*	(	+	2	ab
	c	*	(	+	2	abc
	)	*			0	abc+
	*	*			0	abc+*
	d	*			0	abc+*d
	eos	*			0	abc+*d*

즉시 출력

push

오른쪽 괄호가 나오면,  
왼쪽 괄호가 나올 때까지 pop  
왼쪽 괄호는 스택에서 삭제

연산자의 우선순위가 동등  $\rightarrow$   
pop & push

# 중위 표기에서 후위 표기로의 변환(4)

- 두 예제는 스택킹(stack) 언스택킹(unstack) 수행
  - 우선 순위를 기반으로 하는 기법
  - ‘(’ 괄호 때문에 복잡해짐
    - 스택 속에 있을 때는 낮은 우선순위의 연산자처럼 동작
    - 그 외의 경우 높은 우선순위의 연산자처럼 동작
  - 해결 방법 : 두 가지 종류의 우선순위 사용
    - isp(in-stack precedence)와 icp(incoming precedence) 사용
    - 스택에서 연산자 삭제:  $isp \geq icp$

```
precedence stack[MAX_STACK_SIZE];

/*isp와 icp 배열 - 인덱스는 연산자 lparen, rparen, plus,
  minus, times, divide, mod, eos의 우선순위 값 */

static int isp[] = {0,19,12,12,13,13,13,0};
static int icp[] = {20,19,12,12,13,13,13,0};
```

```
typedef enum {
    lparen, rparen,
    plus, minus, times,
    divide, mod, eos,
    operand} precedence;
```

↑  
니모닉(mnemonic) 사용  
특정문자를 알기 쉽게 표현

$isp[plus] \rightarrow isp[2]$  52

# 중위 표기에서 후위 표기로의 변환(5)

- isp, icp를 이용하는 방식

$a*(b+c)*d$  (infix)  $\rightarrow$   $abc+*d*$  (postfix)

	Token	Stack			Top	Output
		[0]	[1]	[2]		
	a				-1	a
	*	*			0	a
	(	*	(		1	a
	b	*	(		1	ab
	+	*	(	+	2	ab
	c	*	(	+	2	abc
	)	*			0	abc+
	*	*			0	abc+*
	d	*			0	abc+*d
	eos	*			0	abc+*d*

즉시 출력

push

오른쪽 괄호가 나오면,  
왼쪽 괄호가 나올 때까지 pop  
왼쪽 괄호는 스택에서 삭제

연산자의 우선순위가 동등  $\rightarrow$   
pop & push

```
static int isp[] = {0,19,12,12,13,13,13,0};
static int icp[] = {20,19,12,12,13,13,13,0};
```

# 중위 표기에서 후위 표기로의 변환(6)

- mnemonic

Binary	Mnemonic	Name	Action(s)
0000XXXXXXXXXXXXX	LODD	Load Direct	AC ← M[X]
0001XXXXXXXXXXXXX	STOD	Store Direct	M[X] ← AC
0010XXXXXXXXXXXXX	ADDD	Add Direct	AC ← AC + M[X]
0011XXXXXXXXXXXXX	SUBD	Subtract Direct	AC ← AC - M[X]
0100XXXXXXXXXXXXX	JPOS	Jump on pos.	IF AC ≥ 0 : PC ← X
0101XXXXXXXXXXXXX	JZER	Jump on zero	IF AC = 0 : PC ← X
0110XXXXXXXXXXXXX	JUMP	Jump uncond.	PC ← X
0111CCCCCCCCCCCCC	LOCO	Load constant	AC ← C

니모닉(mnemonic) 사용  
특정문자를 알기 쉽게 표현

```

void postfix(void)
{
    /* 수식을 후위 표기식으로 출력한다. 수식 스트링, 스택, top은 전역적이다. */
    char symbol;
    precedence token;
    int n = 0;
    int top = 0; /* eos를 스택에 삽입한다. */
    stack[0] = eos;
    for (token==getToken(&symbol,&n); token!=eos; token==getToken(&symbol,&n)) {
        if (token == operand)
            printf("%c", symbol);
        else if (token == rparen) {
            /* 왼쪽 괄호가 나올 때까지 토큰들을 제거해서 출력시킴 */
            while (stack[top] != lparen)
                printToken(pop(&top));
            pop(&top); /* 왼쪽 괄호를 버린다 */
        } else
            /* symbol의 isp가 token의 icp보다 크거나 같으면 symbol을 제거, 출력 */
            while (isp[stack[top]] >= icp[token])
                printToken(pop(&top));
            add(&top, token);
    }
    while ((token=pop(&top)) != eos)
        printToken(token);
    printf("\n");
}

```

plus, minus → if (token == operand)

token에 저장된 문자 (p143, 그림 3.16)  
token index = 0 → int top = 0; /\* eos를 스택에 삽입한다. \*/

← 피연산자 바로 출력  
← 오른쪽 괄호면, 왼쪽 괄호가 나올 때 까지 pop

Time Complexity =  $O(n)$

전체 for loop =  $O(n)$   
2개 while loop는 각각 token에 대해서 동작하는 것이 아니며,  
stack push, pop이 max n개를 넘지 못함

# 전위 표기 (prefix)

- 연습문제 6번 확인

A+B

↑  
Infix

+AB

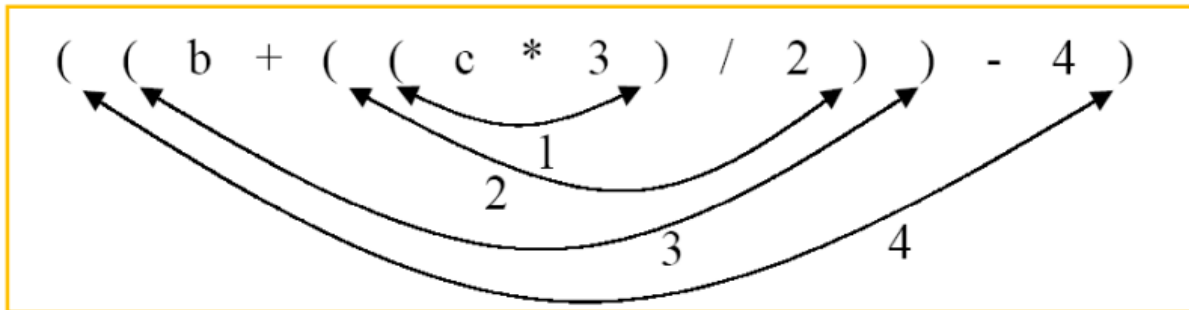
↑  
Prefix

AB+

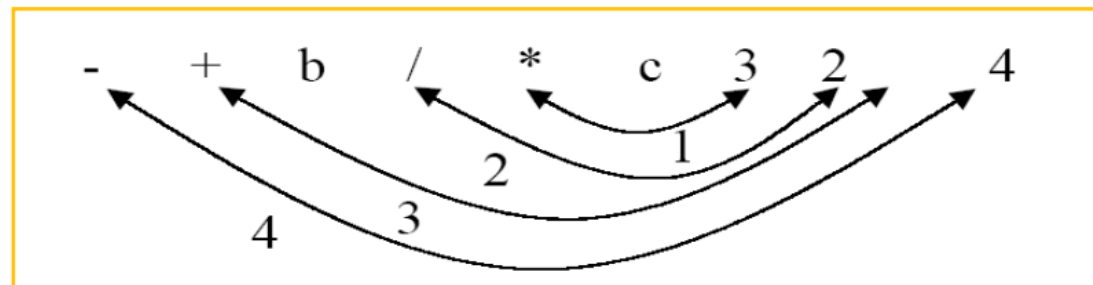
↑  
Postfix

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + ( b * c )	+ a * b c	a b c * +
( a + b ) * c	* + a b c	a b + c *

**b + c \* 3 / 2 - 4**



← 우선순위 결정



# **다중 스택과 큐**

## **(Multiple Stacks and Queues)**

# 다중 스택과 큐(1)

- 같은 배열 내에 두 개 이상의 스택을 구현
  - 각 스택에 대한 영역이 명시되어야 함
- n개의 스택
  - 기억 장소를 n개의 세그먼트로 나눔
    - 스택의 예상 크기를 아는 경우 그 크기에 따라 나눔
    - 스택의 예상 크기를 모를 경우 똑같은 크기로 나눔

```
#define MEMORY_SIZE 100 /* memory의 크기 */
#define MAX_STACKS 10 /* (가능한 스택의 최대 수)+1 */
/* 전역적 메모리 선언 */
element memory[MEMORY_SIZE];

int top[MAX_STACKS];           ← 스택의 개수만큼 top
int boundary[MAX_STACKS];
int n; /* 사용자가 입력한 스택의 수 */
```

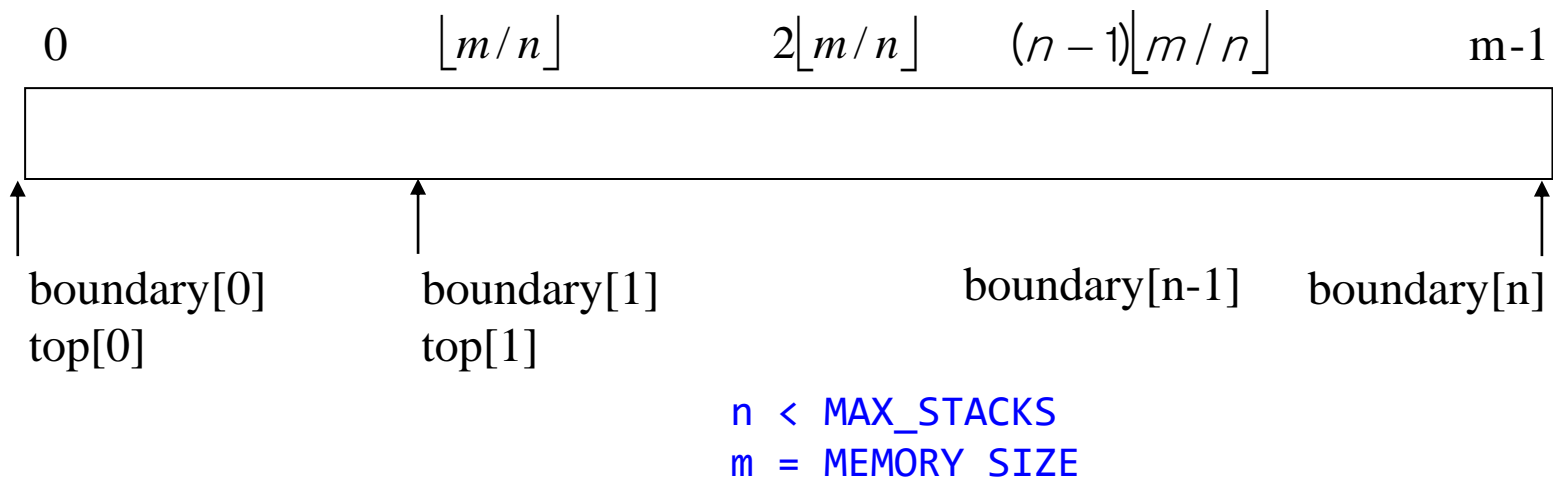
# 다중 스택과 큐(2)

- 배열을 거의 같은 크기의 세그먼트로 나누는 코드

```
top[0] = boundary[0] = -1;
for (i=1; i<n; i++)
    top[i] = boundary[i] = (MEMORY_SIZE/n)*i;

boundary[n] = MEMORY_SIZE-1;
```

- memory[m]에 있는 n 스택에 대한 초기 구성



# 다중 스택과 큐(3)

```
void push(int i, element item)
{ /* item을 i번째 스택에 삽입 */
    if (top[i] == boundary[i+1])
        stackFull(i);
    memory[++top[i]] = item;
}
```

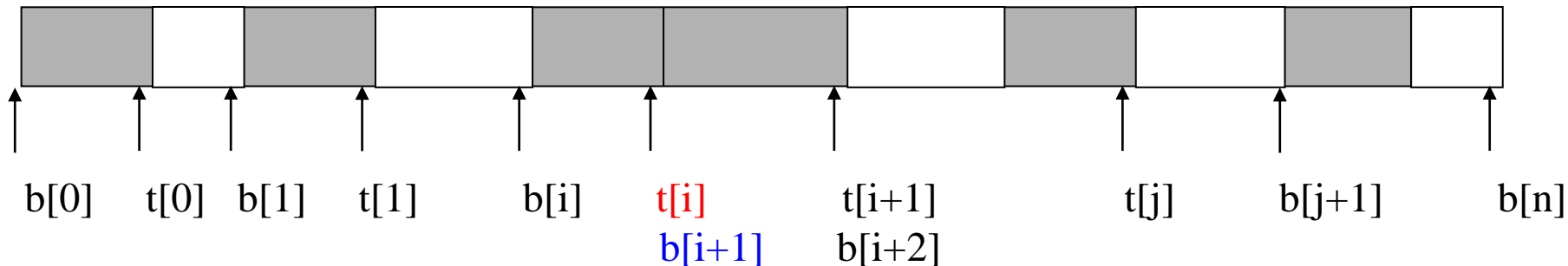
$\text{top}[i]$ 값 = memory index

스택 i에 item 삽입

```
element pop(int i)
{ /* i번째 스택에서 톱 원소를 제거 */
    if (top[i] == boundary[i])
        return stackEmpty(i);
    return memory[top[i]--];
}
```

스택 i에서 item 삭제

- 스택 i와 스택 i+1이 만났지만 메모리는 만원이 아닌 상태



b = boundary, t = top

# 다중 스택과 큐(4)

- 배열이 full이 될 때 삽입할 수 있는 stackFull 함수
  - 메모리 내의 빈 공간이 있는 가를 결정
  - 사용할 수 있는 자유 공간이 있다면 만원인 스택에 그 자유 공간을 할당하도록 다른 스택들을 이동시킴
- 1) 공간 찾기 (stack i를 기준) ← i를 기준으로 뒤쪽 스택을 오른쪽으로 이동
  - 스택 j와 j+1사이에 빈 공간이 있는, 즉  $top[j] < boundary[j+1]$  ( $i < j < n$ )을 만족하는 최소의 j를 찾음
  - 그러한 j가 있다면 스택 i+1, i+2, ..., j를 오른쪽으로 한 자리씩 이동시켜 스택 i와 i+1 사이에 하나의 공간을 만든다.
- 2) 1)에서 j를 찾지 못하면, 스택 i의 왼쪽을 조사
  - 스택 j와 스택 j+1 사이에 빈 공간이 있는, 즉  $top[j] < boundary[j+1]$  ( $1 < j < i$ )를 만족하는 최대 j를 찾음
  - 그러한 j가 있다면 스택 j+1, j+2, ..., i를 왼쪽으로 한자리씩 이동시켜 스택 i와 i+1사이에 하나의 빈 공간을 만든다.
- 3) 1), 2)에서 j를 못 찾으면 빈 공간은 없음.
  - 에러 메시지와 함께 종료