

# 5장: 스택

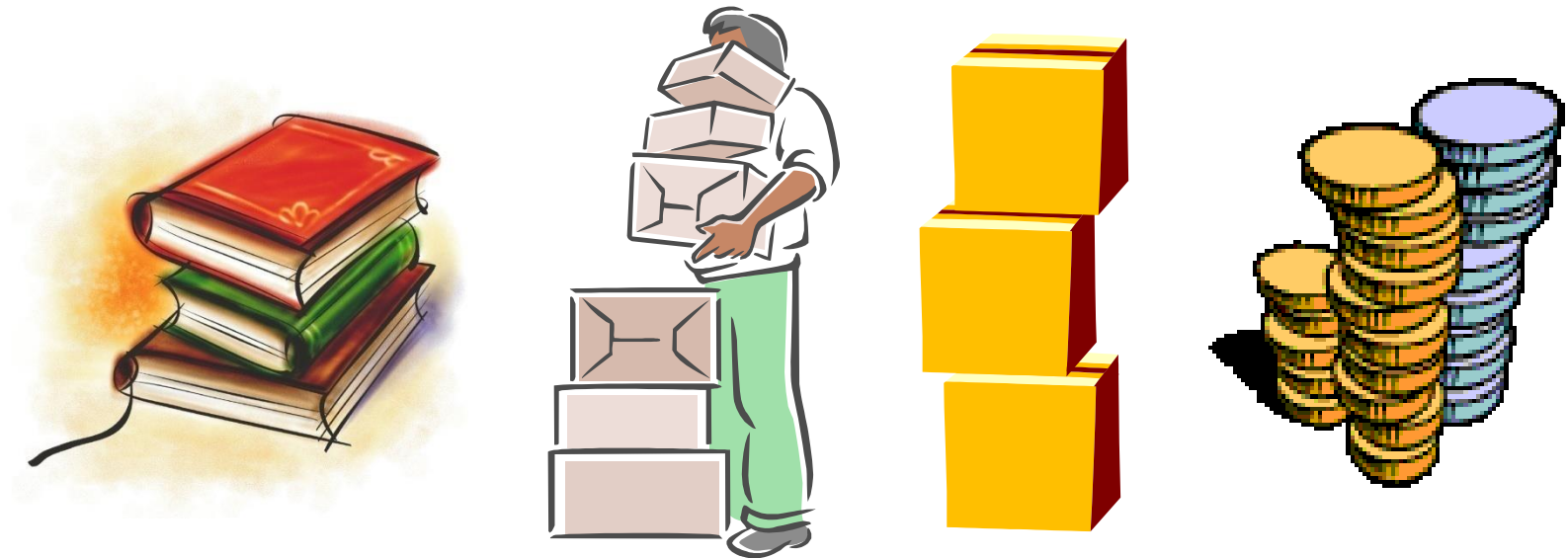
---

- 목차
  - 1. 스택이란?
  - 2. 배열로 구현한 스택
  - 3. 연결 리스트로 구현한 스택
  - 4. 괄호 검사
  - 5. 수식의 계산
  - 6. 미로 탐색 문제

# 1. 스택이란?

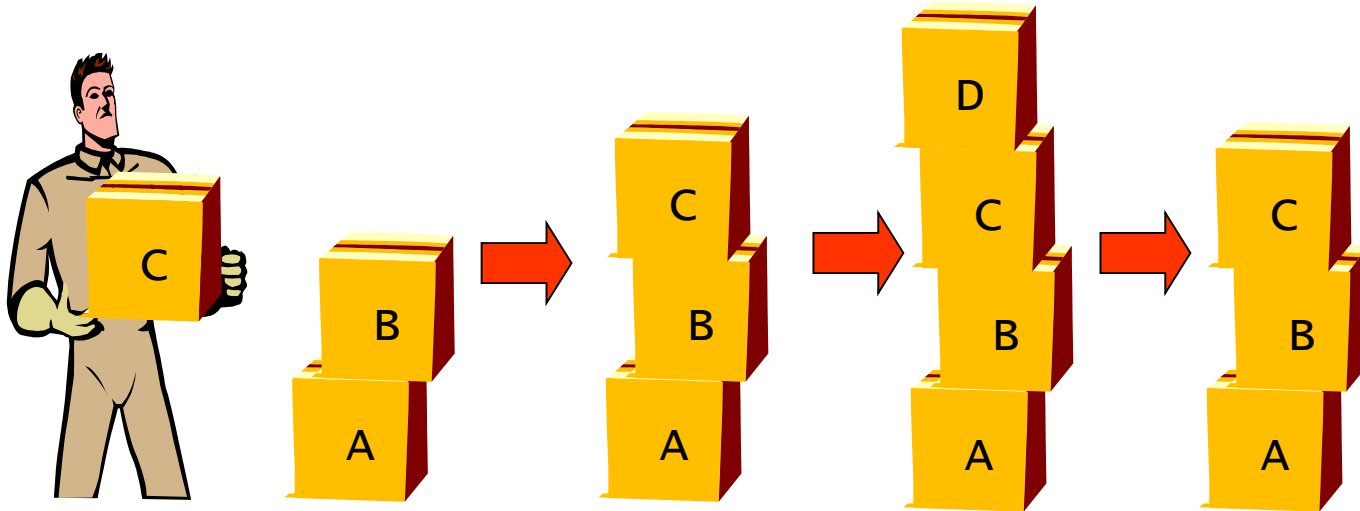
---

- 스택(stack): 쌓아놓은 더미

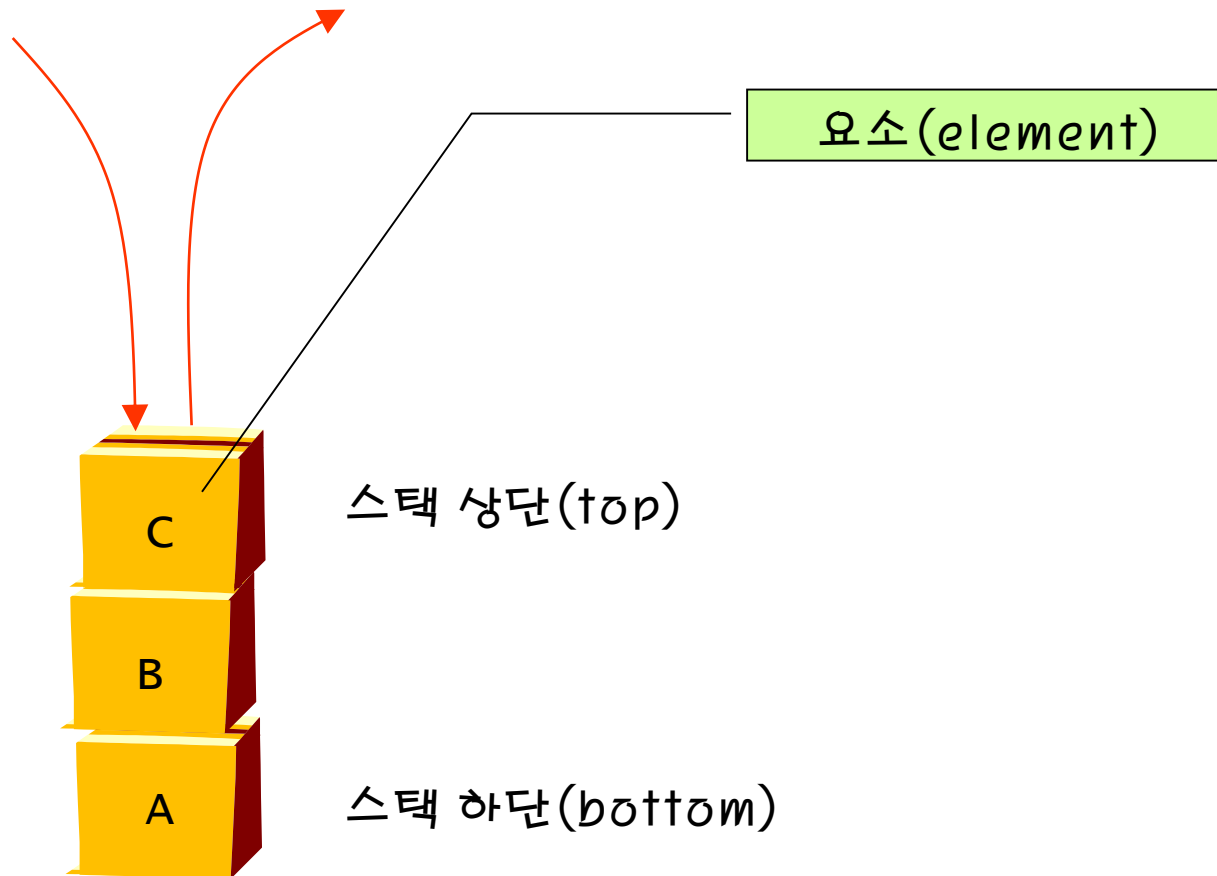


# 스택의 특징

- 후입선출(LIFO: Last-In First-Out): 가장 최근에 들어온 데이터가 가장 먼저 나감.



# 스택의 구조



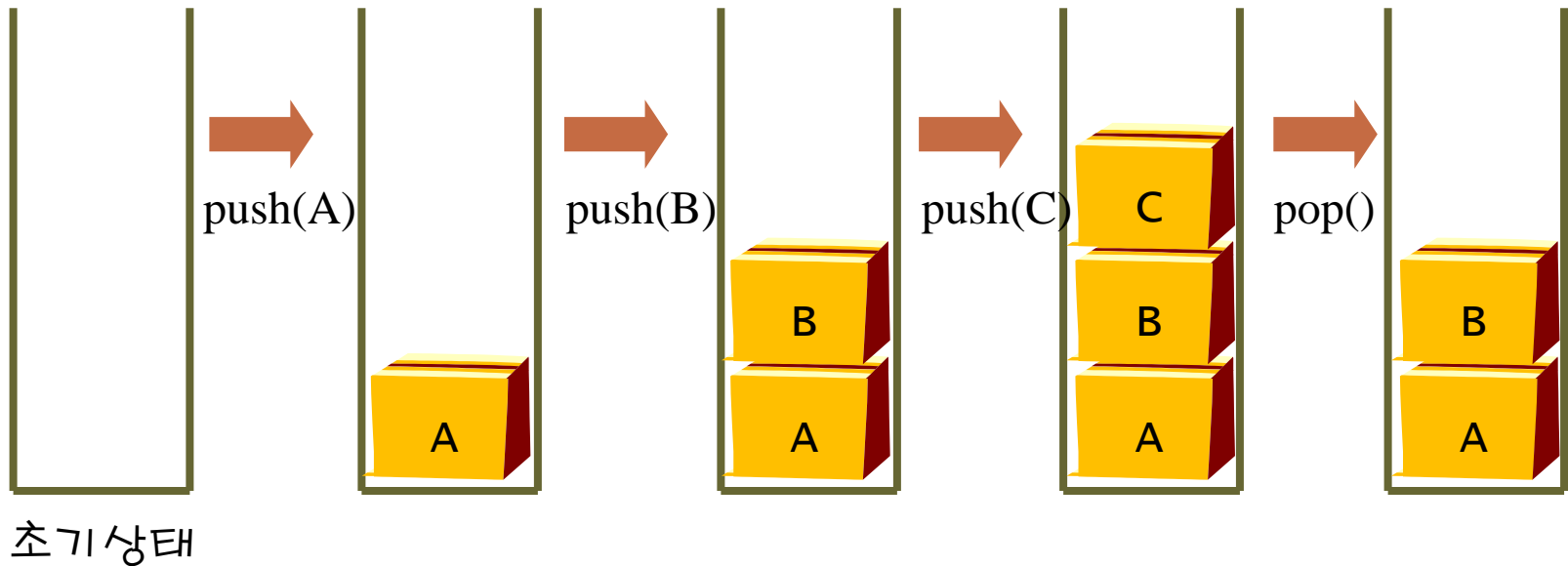
# 스택 추상데이터타입(ADT)

---

- 객체:  $n$ 개의 element형의 요소들의 선형 리스트
- 연산:
  - `create()` ::= 스택을 생성한다.
  - `is_empty(s)` ::= 스택이 비어있는지를 검사한다.
  - `is_full(s)` ::= 스택이 가득 찼는가를 검사한다.
  - `push(s, e)` ::= 스택의 맨 위에 요소  $e$ 를 추가한다.
  - `pop(s)` ::= 스택의 맨 위에 있는 요소를 삭제한다.
  - `peek(s)` ::= 스택의 맨 위에 있는 요소를 삭제하지 않고 반환한다.

# 스택의 연산

- 삽입(push), 삭제(pop)



# 스택의 연산

---

- `is_empty(s)`: 스택이 공백상태인지 검사
- `is_full(s)`: 스택이 포화상태인지 검사
- `create()`: 스택을 생성
- `peek(s)`: 요소를 스택에서 삭제하지 않고 보기만 하는 연산
  - (참고)`pop` 연산은 요소를 스택에서 완전히 삭제하면서 가져온다.

# 스택의 용도

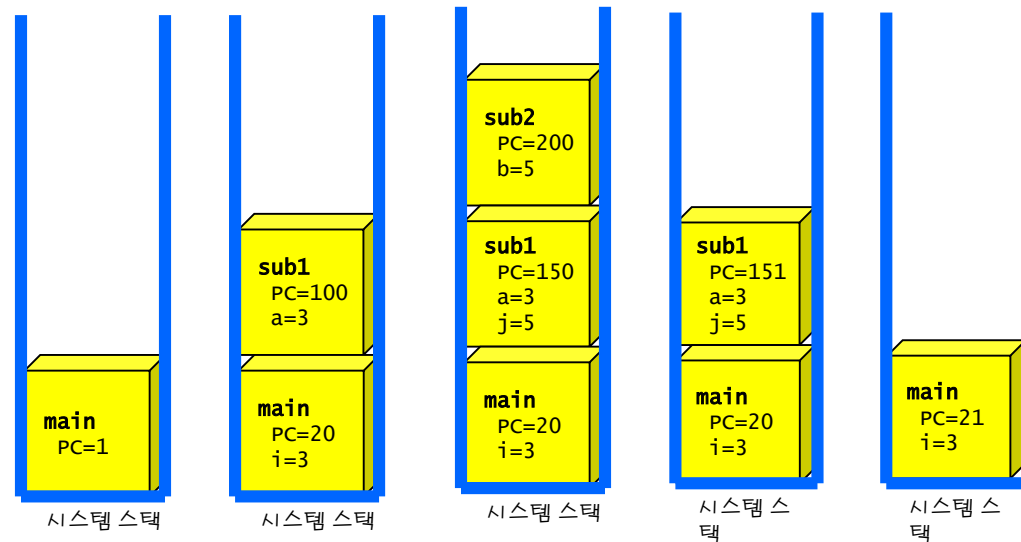
- 입력과 역순의 출력이 필요한 경우
  - 에디터에서 되돌리기(undo) 기능
  - 함수호출에서 복귀주소 기억

```

1  int main()
   {
   int i=3;
20  sub1(i);
   ...
   }

100 int sub1(int a)
   {
   int j=5;
150  sub2(j);
   ...
   }

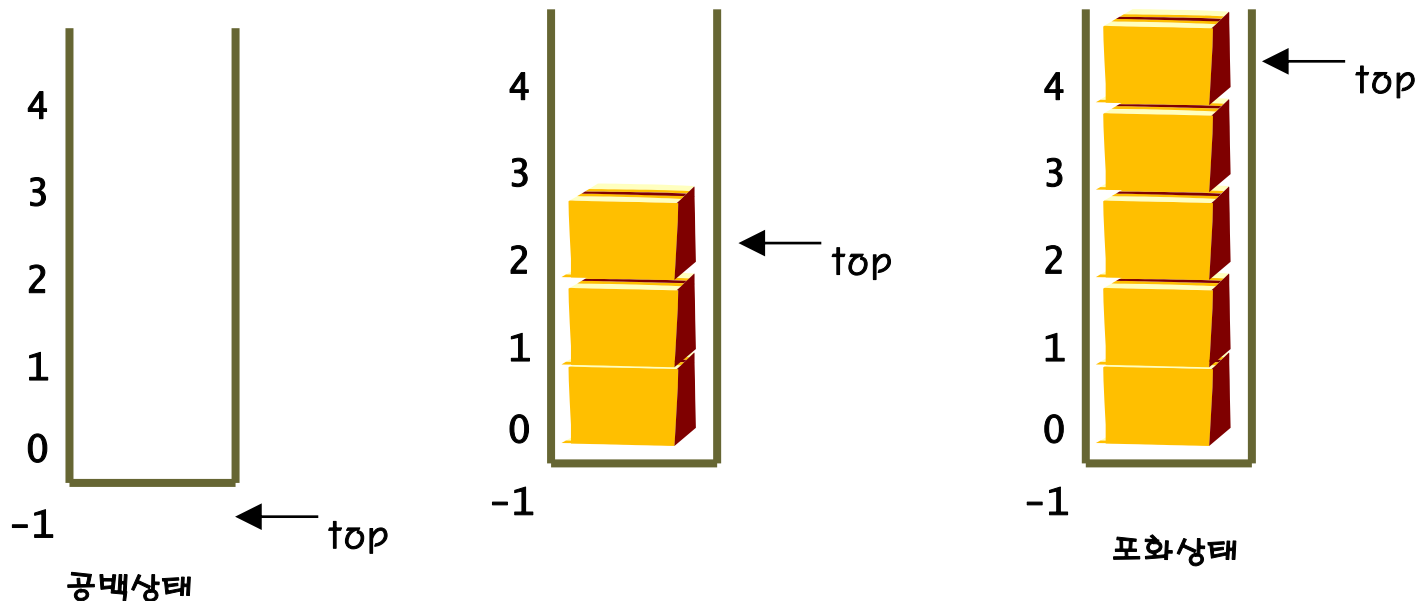
200 void sub2(int b)
   {
   ...
   }
    
```





## 2. 배열로 구현한 스택

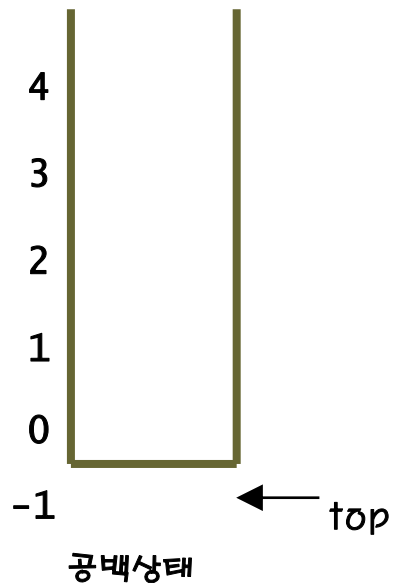
- 1차원 배열 `stack[ ]`
- 스택에서 가장 최근에 입력되었던 자료를 가리키는 `top` 변수
- 가장 먼저 들어온 요소는 `stack[0]`에, 가장 최근에 들어온 요소는 `stack[top]`에 저장
- 스택이 공백상태이면 `top`은 -1



# is\_empty, is\_full 연산의 구현

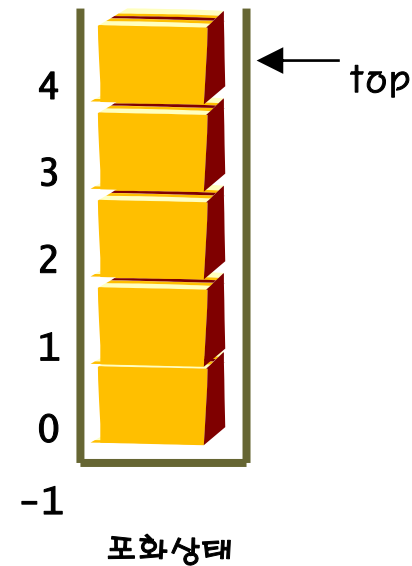
is\_empty(*s*)

```
if top == -1
  then return TRUE
  else return FALSE
```



is\_full(*s*)

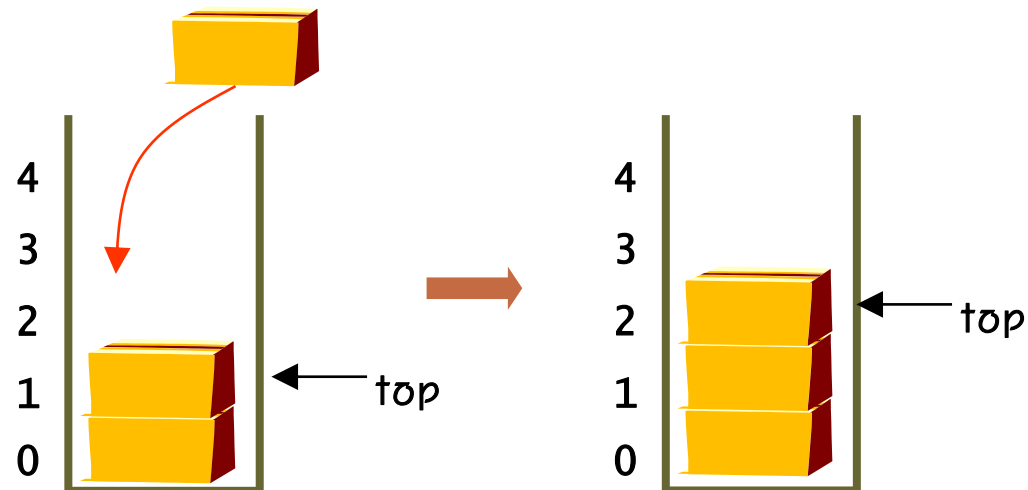
```
if top == (MAX_STACK_SIZE-1)
  then return TRUE
  else return FALSE
```



# push 연산

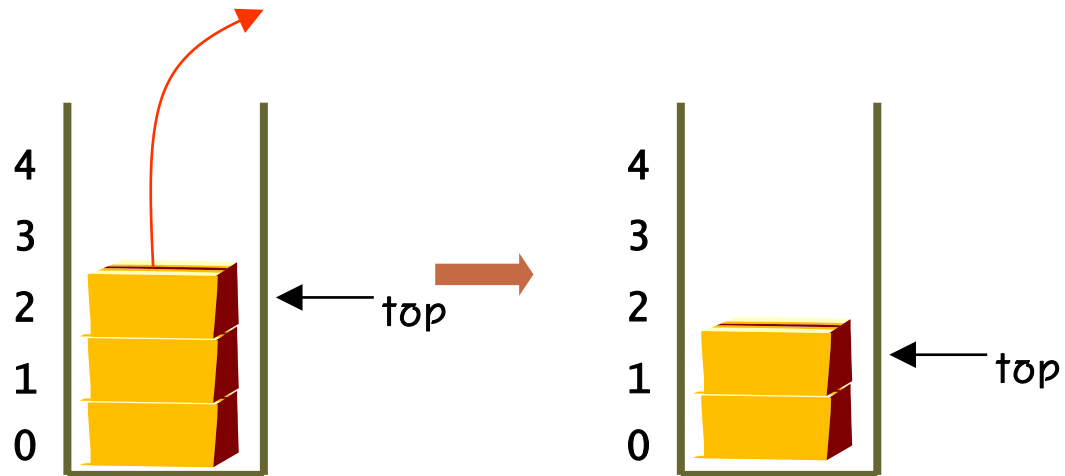
```
push(S, x)
```

```
if is_full(S)  
    then error "overflow"  
    else top ← top + 1  
        stack[top] ← x
```



# pop 연산

```
pop(S, x)  
  
if is_empty(S)  
    then error "underflow"  
    else e ← stack[top]  
        top ← top - 1  
        return e
```



# C언어 구현

```
typedef int element;
typedef struct {
    element stack[MAX_STACK_SIZE];
    int top;
} StackType;

// 스택 초기화 함수
void init(StackType *s)
{
    s->top = -1;
}

// 공백 상태 검출 함수
int is_empty(StackType *s)
{
    return (s->top == -1);
}

// 포화 상태 검출 함수
int is_full(StackType *s)
{
    return (s->top == (MAX_STACK_SIZE-1));
}
```

배열의 요소는 element 타입  
으로 선언

관련 데이터를 구조체로 묶어  
서 함수의 파라미터로 전달

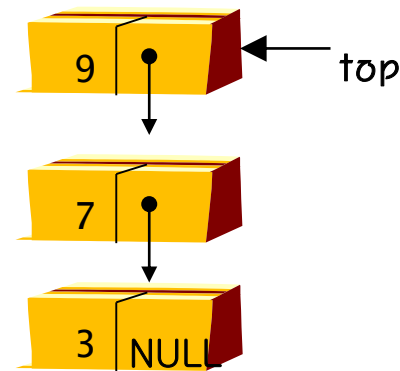
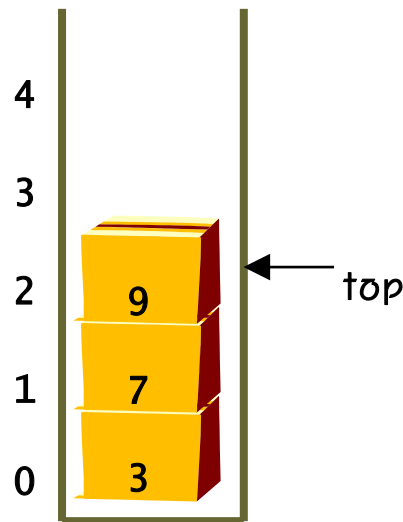
```
// 삽입함수
void push(StackType *s, element item)
{
    if( is_full(s) ) {
        fprintf(stderr, "스택 포화 에러\n");
        return;
    }
    else s->stack[++(s->top)] = item;
}

// 삭제함수
element pop(StackType *s)
{
    if( is_empty(s) ) {
        fprintf(stderr, "스택 공백 에러\n");
        exit(1);
    }
    else return s->stack[(s->top)--];
}

// 피크함수
element peek(StackType *s)
{
    if( is_empty(s) ) {
        fprintf(stderr, "스택 공백 에러\n");
        exit(1);
    }
    else return s->stack[s->top];
}
```

### 3. 연결 리스트로 구현한 스택

- 연결된 스택(linked stack): 연결리스트를 이용하여 구현한 스택
- 장점: 크기가 제한되지 않음
- 단점: 구현이 복잡하고 삽입이나 삭제 시간이 오래 걸린다.



# 연결된 스택 정의

```
typedef int element;
```

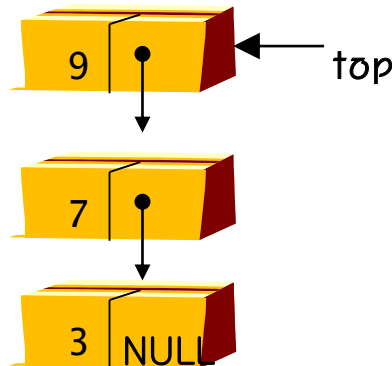
요소의 타입

```
typedef struct StackNode {
    element item;
    struct StackNode *link;
} StackNode;
```

노드의 타입

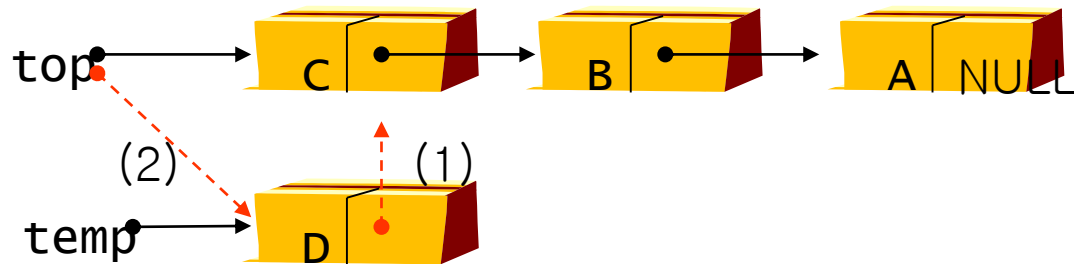
```
typedef struct {
    StackNode *top;
} LinkedStackType;
```

연결된 스택의 관련 데이터



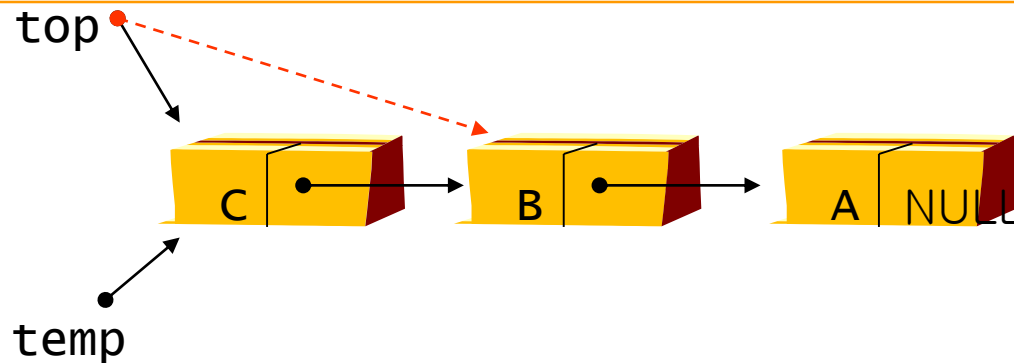


# 연결된 스택에서 push 연산



```
// 삽입 함수
void push(LinkedStackType *s, element item)
{
    StackNode *temp=(StackNode *)malloc(sizeof(StackNode));
    if( temp == NULL ){
        fprintf(stderr, "메모리 할당에러\n");
        return;
    }
    else{
        temp->item = item;
        temp->link = s->top;
        s->top = temp;
    }
}
```

# 연결된 스택에서 pop 연산



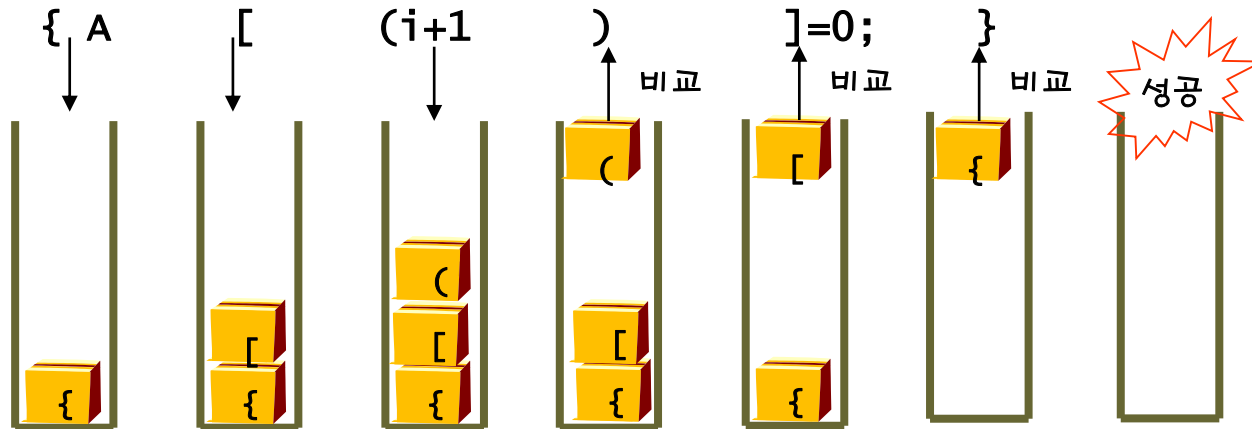
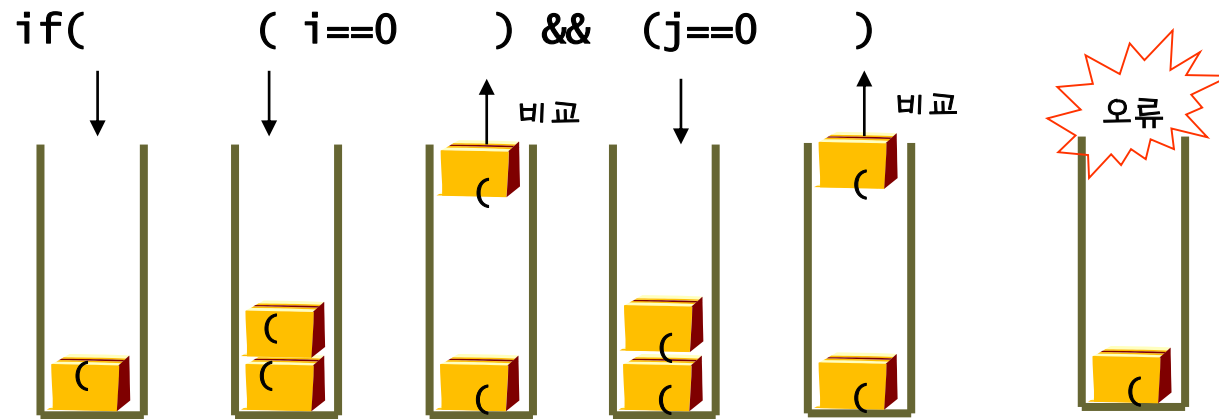
```
// 삭제 함수
element pop(LinkedStackType *s)
{
    if( is_empty(s) ) {
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else{
        StackNode *temp=s->top;
        int item = temp->item;
        s->top = s->top->link;
        free(temp);
        return item;
    }
}
```

## 4. 괄호 검사

---

- 괄호의 종류: 대괄호 ('[', ']'), 중괄호 ('{', '}'), 소괄호 ('(', ')')
- 조건
  1. 왼쪽 괄호의 개수와 오른쪽 괄호의 개수가 같아야 한다.
  2. 같은 괄호에서 왼쪽 괄호는 오른쪽 괄호보다 먼저 나와야 한다.
  3. 서로 다른 타입의 왼쪽괄호와 오른쪽 괄호 쌍은 서로를 교차하면 안된다.
- 잘못된 괄호 사용의 예
  - $a(b)$  : 1번 조건 위반
  - $a(b)c$  : 1,2번 조건 위반
  - $a\{b(c[d]e)f\}$  : 3번 조건 위반

# 스택을 이용한 괄호 검사



# 알고리즘

---

- 알고리즘의 개요

- 문자열에 있는 괄호를 차례대로 조사하면서 왼쪽 괄호를 만나면 스택에 삽입하고, 오른쪽 괄호를 만나면 스택에서 **top** 괄호를 삭제한 후 오른쪽 괄호와 짝이 맞는지를 검사한다.
- 이 때, 스택이 비어 있으면 조건 1 또는 조건 2 등을 위배하게 되고 괄호의 짝이 맞지 않으면 조건 3 등에 위배된다.
- 마지막 괄호까지를 조사한 후에도 스택에 괄호가 남아 있으면 조건 1에 위배되므로 0(거짓)을 반환하고, 그렇지 않으면 1(참)을 반환한다.

# 괄호 검사 알고리즘

*check\_matching(expr)*

while (입력 expr의 끝이 아니면)

ch ← expr의 다음 글자

switch(ch)

case '(': case '[': case '{':

ch를 스택에 삽입

break

case ')': case ']': case '}':

if ( 스택이 비어 있으면 )

then 오류

else 스택에서 open\_ch를 꺼낸다

if (ch 와 open\_ch가 같은 짝이 아니면)

then 오류 보고

break

if( 스택이 비어 있지 않으면 )

then 오류

왼쪽 괄호이면 스택에 삽입

오른쪽 괄호이면 스택에서 삭제비교

## 5. 수식의 계산

- 수식의 표기방법: 전위(prefix), 중위(infix), 후위(postfix)

중위 표기법	전위 표기법	후위 표기법
$2+3*4$	$+2*34$	$234*+$
$a*b+5$	$+5*ab$	$ab*5+$
$(1+2)+7$	$+7+12$	$12+7+$

- 컴퓨터에서의 수식 계산순서
  - 중위표기식-> 후위표기식->계산
  - $2+3*4 \rightarrow 234*+ \rightarrow 14$
  - 모두 스택을 사용
  - 먼저 후위표기식의 계산법을 알아보자

# 후위 표기식의 계산

- 장점: 괄호를 사용하지 않고도 계산가능. 이미 식 자체에 우선순위가 표현되어 있음. 수식을 읽으면서 바로 계산이 가능
- 수식을 왼쪽에서 오른쪽으로 스캔하여 피연산자이면 스택에 저장하고 연산자이면 필요한 수만큼의 피연산자를 스택에서 꺼내 연산을 실행하고 연산의 결과를 다시 스택에 저장
- (예)  $82/3-32^{*}+$

토큰	스택						
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	8						
2	8	2					
/	4						
3	4	3					
-	1						
3	1	3					
2	1	3	2				
*	1	6					
+	7						

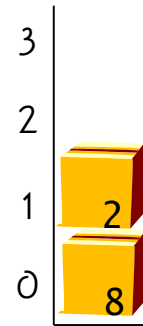


↓  
8 2 / 3 -



피연산자-> 삽입

↓  
8 2 / 3 -



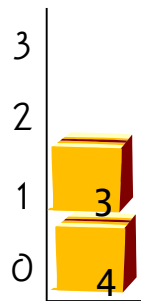
피연산자-> 삽입

↓  
8 2 / 3 -



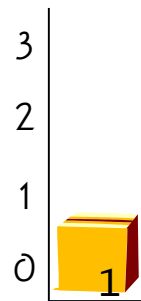
연산자->  $8/2=4$  삽입

↓  
8 2 / 3 -



피연산자-> 삽입

↓  
8 2 / 3 -



연산자->  $4-1=1$  삽입

8 2 / 3 -



종료->전체 연산 결과=1

# 후위 표기식 계산 알고리즘

스택  $s$ 를 생성하고 초기화한다.

for 항목 in 후위표기식

do if (항목이 피연산자이면)

push( $s$ , item)

if (항목이 연산자  $op$ 이면)

then second  $\leftarrow$  pop( $s$ )

first  $\leftarrow$  pop( $s$ )

result  $\leftarrow$  first  $op$  second //  $op$  는  $+ - * /$  중의 하나

push( $s$ , result)

final\_result  $\leftarrow$  pop( $s$ );

```
// 후위 표기 수식 계산 함수
eval(char exp[])
{
    int op1, op2, value, i=0;
    int len = strlen(exp);
    char ch;
    StackType s;

    init(&s);
    for( i=0; i<len; i++){
        ch = exp[i];
        if( ch != '+' && ch != '-' && ch != '*' && ch != '/' ){
            value = ch - '0';      // 입력이 피연산자이면
            push(&s, value);
        }
        else{ //연산자이면 피연산자를 스택에서 제거
            op2 = pop(&s);
            op1 = pop(&s);
            switch(ch){ //연산을 수행하고 스택에 저장
                case '+': push(&s, op1+op2); break;
                case '-': push(&s, op1-op2); break;
                case '*': push(&s, op1*op2); break;
                case '/': push(&s, op1/op2); break;
            }
        }
    }
    return pop(&s);
}
```

# 중위표기식->후위표기식

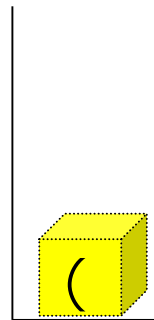
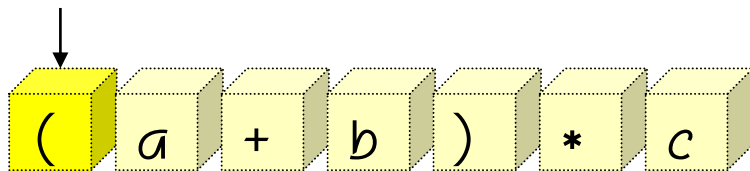
---

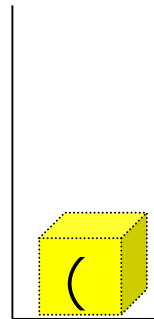
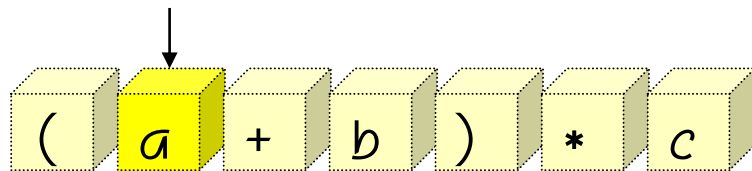
- 중위표기와 후위표기

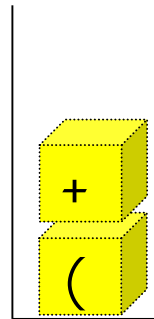
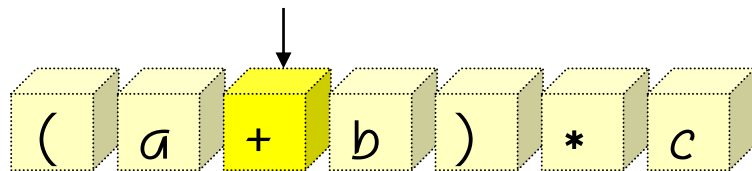
- 중위 표기법과 후위 표기법의 공통점은 피연산자의 순서는 동일
- 연산자들의 순서만 다름(우선순위순서)
  - >연산자만 스택에 저장했다가 출력하면 된다.
- $2+3*4 \rightarrow 234*+$

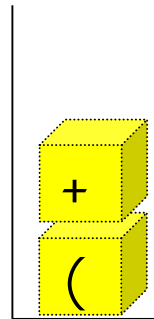
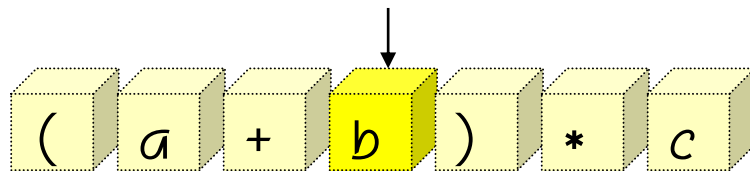
- 알고리즘

- 피연산자를 만나면 그대로 출력
- 연산자를 만나면 스택에 저장했다가 스택보다 우선 순위가 낮은 연산자가 나오면 그때 출력
- 왼쪽 괄호는 우선순위가 가장 낮은 연산자로 취급
- 오른쪽 괄호가 나오면 스택에서 왼쪽 괄호위에 쌓여있는 모든 연산자를 출력

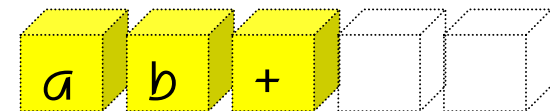
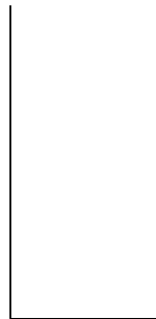
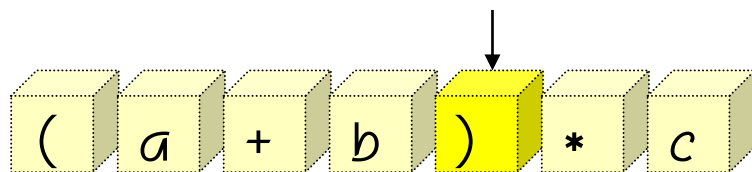


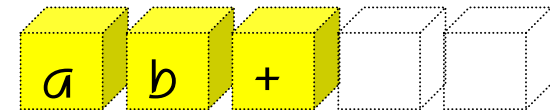
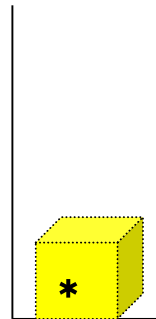
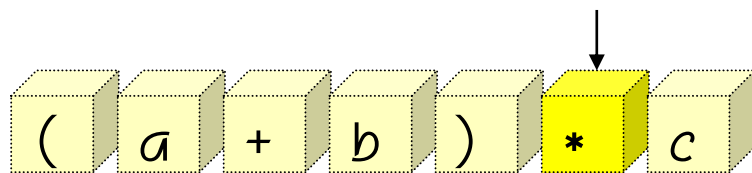


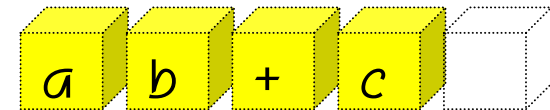
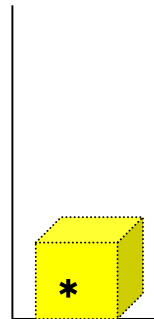
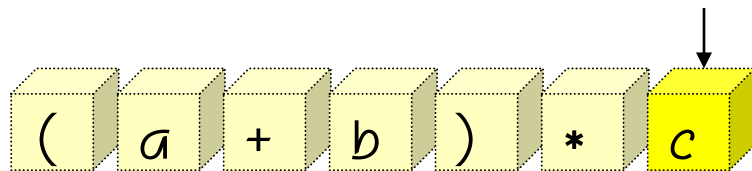


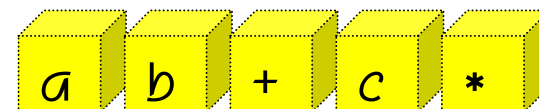
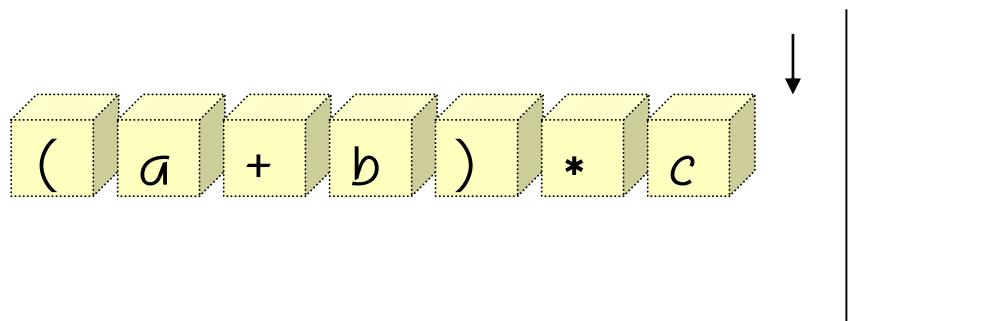












```
infix_to_postfix(exp)
```

스택 s를 생성하고 초기화

```
while (exp에 처리할 문자가 남아 있으면)
```

```
    ch ← 다음에 처리할 문자
```

```
    switch (ch)
```

```
        case 연산자:
```

```
            while ( peek(s)의 우선순위 ≥ ch의 우선순위 )
```

```
                do e ← pop(s)
```

```
                e를 출력
```

```
            push(s, ch);
```

```
            break;
```

```
        case 왼쪽 괄호:
```

```
            push(s, ch);
```

```
            break;
```

```
        case 오른쪽 괄호:
```

```
            e ← pop(s);
```

```
            while( e ≠ 왼쪽괄호 )
```

```
                do e를 출력
```

```
                e ← pop(s)
```

```
            break;
```

```
        case 피연산자:
```

```
            ch를 출력
```

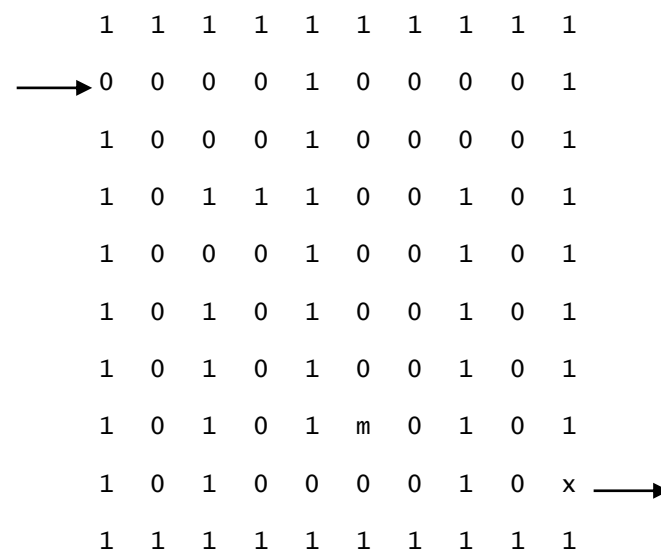
```
            break;
```

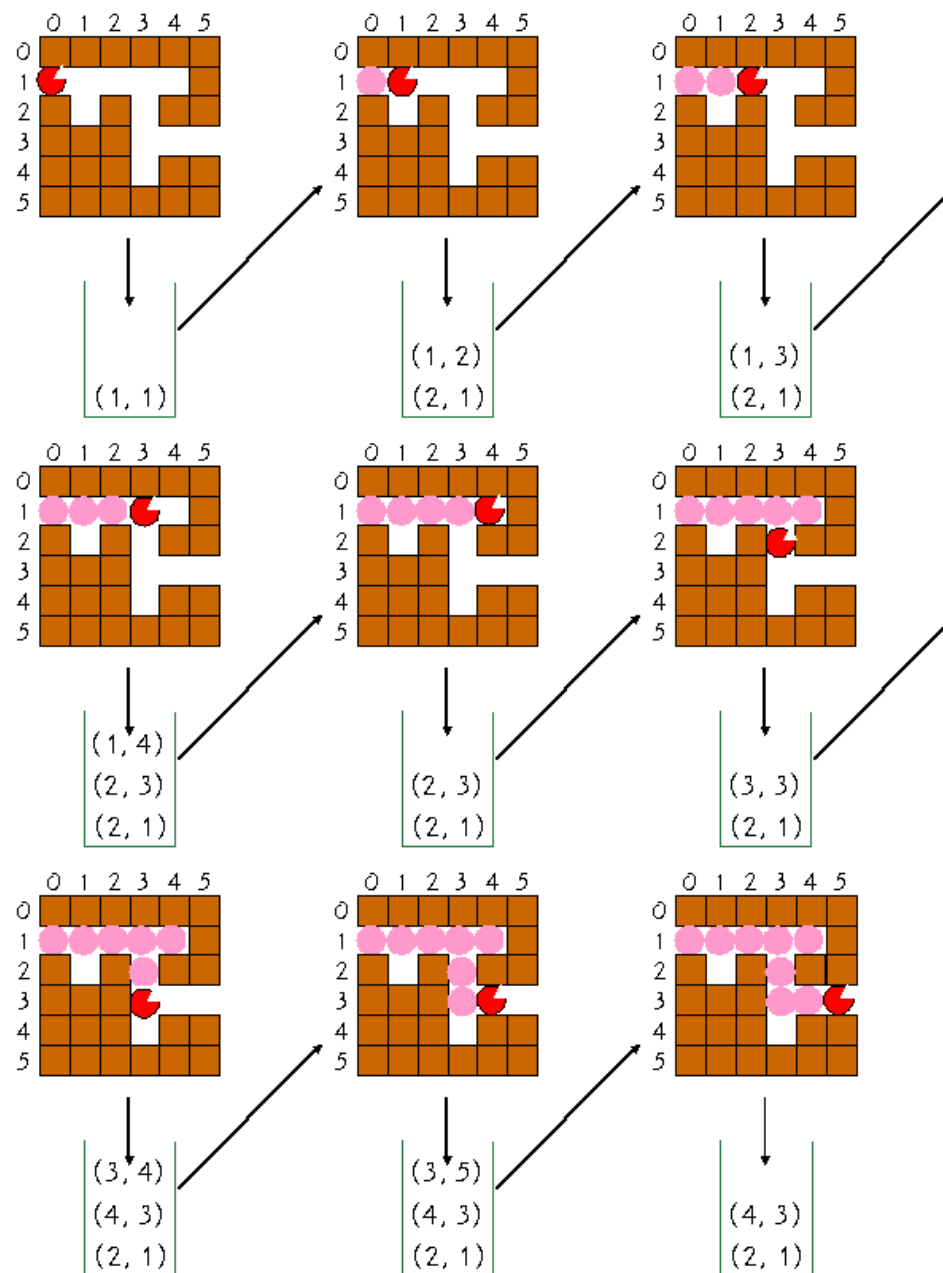
```
while( not is_empty(s) )
```

```
    do e ← pop(s)
```

```
    e를 출력 4
```

- 체계적인 방법 필요
- 현재의 위치에서 가능한 방향을 스택에 저장해놓았다가 막다른 길을 만나면 스택에서 다음 탐색 위치를 꺼낸다.





# 미로탐색 알고리즘

```
스택 s과 출구의 위치 x, 현재 생쥐의 위치를 초기화
while( 현재의 위치가 출구가 아니면 )
do  현재위치를 방문한 것으로 표기
    if( 현재위치의 위, 아래, 왼쪽, 오른쪽 위치가 아직 방문되지 않았고 갈수 있으면 )
        then 그 위치들을 스택에 push
    if( is_empty(s) )
        then 실패
        else 스택에서 하나의 위치를 꺼내어 현재 위치로 만든다;
성공;
```