< 4장. 스택(STACKS), 김성현, 201910783 >

<순서>

4.1 스택이란?	4.2 스택의 구현	4.3 동적 배열 스택	4.4 스택의 응용: 괄호검사문제	4.5 스택의 응용: 후위 표기 수식의 계산
- 스택(stack) 소개	- '배열'을 사용한 스택의 구현			- 수식 표기 변환 및 계산
				4.6 스택의 응용: 미로 문제
				- 미로 탐색 문제

자료구조 첫번째, 컴퓨터에서 뺄 수 없을 정도로 많이 사용되는 자료구조임.

4.1. 스택이란?

p.102 - 스택(stack) 소개.

- ✓ • 스택(stack)이란?
- ✓ • 스택의 구조 → 뭐가 있고, 뭐가 있다. → 뭐가 있어야 한다! (component로 보도 집중해 볼 것!)
- ✓ • 함수 호출과 스택 → function → call / return 할 때 생각해야 한다. (여기 기준에 스택 사용!!)
- ✓ • 스택 추상 데이터 타입(ADT)
  - 스택을 이해했다면, 이 스택 ADT를 알 수 있어야 한다!

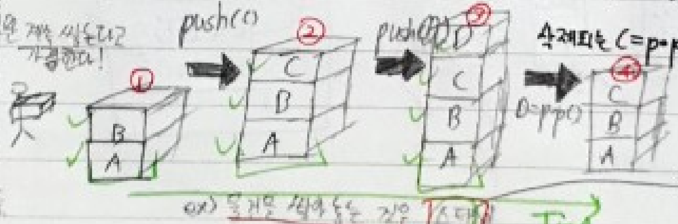
"공학에서는 이 '구조'라는 것이 굉장히 중요함. 시스템이 복잡해지면 이게 굉장히 중요"

"어떤 개념이 앞으로 나오면, 그 개념을 구성하는 구조, 즉 그 특징(properties)을 파악해놓으면 나중에 굉장히 편하게 시스템들을 파악할 수 있게 된다!

p.102~103 • 스택(stack)이란?

▫ 후입선출(LIFO: Last-in-First-Out): 가장 나중에 삽입된(=push) 데이터가 가장 먼저 삭제되는(=pop) 자료구조.

데이터 쌓는 것 ← • 우리 주변의 스택 예: 접시더미, 책상에 쌓여있는 책, 창고에 쌓여있는 상자들



ex) 물건을 쌓아 놓는 것은 (스택)! → 스택 → 쌓아 놓는다 → 차곡차곡 쌓는다는 의미

Time.

★ <stack 동작>

★ push() → 쌓는다
★ pop() → 삭제한다, 제거한다

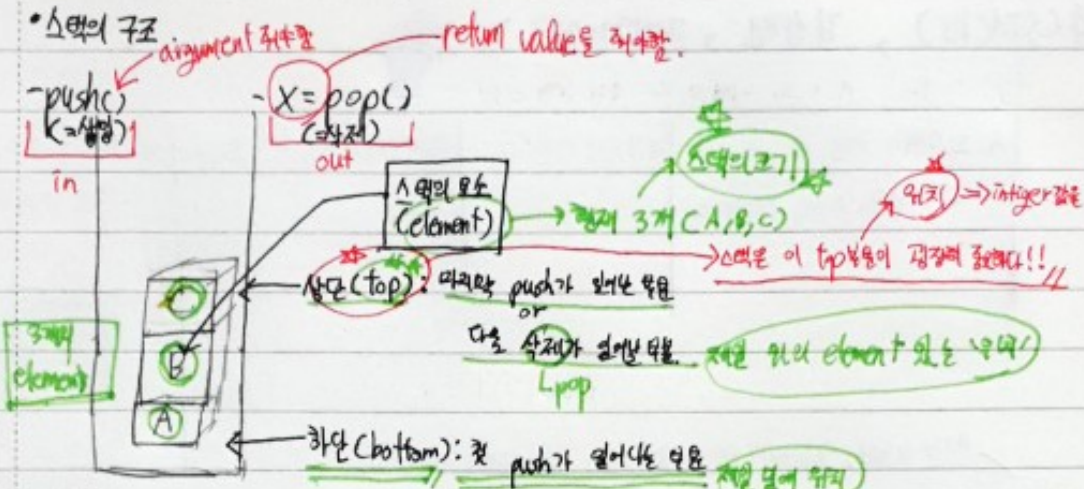
push() → Argument O
pop() → Argument X

• push(C) → 삽입되는 물건은 Argument로 주게됨.
• push(D)

D = pop() → 가지고 왔더니 D다 → 처음엔 뭘 가지고 올지 모른다. (그냥 위에 있는 걸 제거하는 거임) 전달할 값은, Argument가 없다 ★ 그러니 결정할 수 없으니 → return value가 굉장히 중요!

p.103

• 스택의 구조



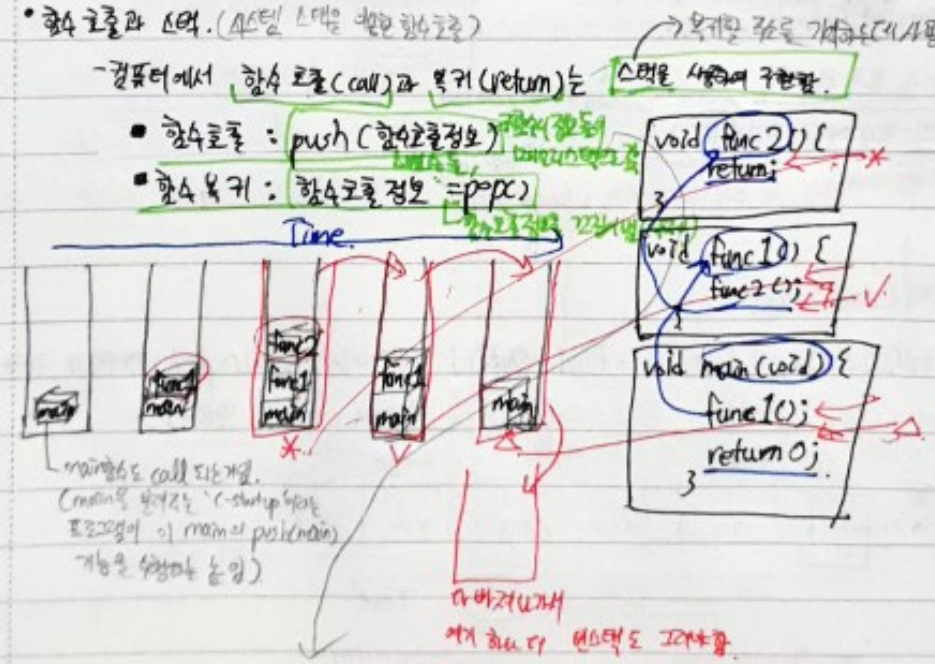
• 스택에 요소가 하나도 없을 때

=> 공백스택(empty stack)

• 스택 → 자료의 출력순서가 입력순서의 역순으로 이루어져야 할 경우에 주로 사용.

p.103~105

• 함수 호출과 스택. (시스템 스택을 이용한 함수호출)



p.103 • 시스템 스택 => 함수가 호출될 때마다 활성 레코드(activation record)가 스택 안에서 생성되고 함수가 종료.

그리고 매개변수를 통한 이유나 함수호출시 메모리 할당 등에서 선언된 지역 변수가 같이 생성됨.

p.104 스택에서 활성 레코드가 언제 생기고 없어지게 됨.

p.104~106

• 두가지 기본연산
- 삽입 (push)
- 삭제 (pop)

• 스택 추상 데이터 타입 (ADT)

- 객체: 0개 이상의 원소를 가지는 유한 선형 리스트.
- 연산:
  - create (size) ::= 최대 크기가 size인 공백 스택을 생성한다.
    - └ empty, 비어있는, 텅 빈 스택 ⇒ 공백 스택이라 한다.
  - is_full(s) ::= if (스택의 원소 수 == size) return TRUE; else return FALSE;
    - is_full(s): 스택이 꽉 찼나? / TRUE: 꽉 찼다. / FALSE: 꽉 안찼다.
  - is_empty(s) ::= if (스택의 원소 수 == 0) return TRUE; else return FALSE;
    - is_empty(s): 스택이 element가 하나도 없나? / TRUE: 텅 비었다 / FALSE: 하나 이상 있다.
  - push(s, item) ::= if(is_full(s)) return ERROR_STACKFULL; else 스택의 맨 위에 item을 추가;
    - ERROR_STACKFULL → push이 안되고 에러 발생! / └→ 정상 실행되므로 push를 한다는 뜻
  - pop(s) ::= if(is_empty(s)) return ERROR_STACKEMPTY; else 스택의 맨 위의 원소를 제거해서 반환;
    - 서로 비슷하지만 다름
  - peek(s) ::= if (is_empty(s)) return ERROR_STACKEMPTY; else 스택의 맨 위의 원소를 제거하지 않고 반환;
    - └ pop과 다르게 갯수가 줄어들지는 않음!! 무엇이 있는지 보기만 한다!!!

(여백 메모)
- 이 두개 중요!
- 이거 구현에 사용됨!!
- 이거 두개 또 사용됨으로 중요!!!
- 이렇게 ADT에선 그 밑의 구현에 필요한 함수 위에 기본적으로 구현된다
- 이 두개는 가장 중요!!!
- 유한한 개수를 갖는다!!

p.105 ⑤
- • peek 연산 → 스택에서 요소를 스택에서 삭제하지 않고 보기만 하는 연산.
- • pop 연산 → 스택에서 요소를 완전히 삭제하면서 가져온다.

4.2 스택의 구현.

p.101 • 스택을 구현하는 방법 ┌ 배열을 이용하는 방법 → 구현 간단, 성능우수 (단점) 스택의 크기가 고정되는 약점 有 (기존에 제일 많이 쓰던 거 이거!!)
└ 연결 리스트를 이용하는 방법 → 구현이 약간 복잡, (장점) 스택의 크기를 필요에 따라 가변적이게 장점 有

• 배열을 사용한 스택의 구현

□ 배열을 사용한 스택

□ 전역변수로 구현 → 간단할 때 쓰지만 잘 안씀

□ 구조체 형으로 구현. → 이걸 가장 많이 사용 ☆☆

p.107 시에 • 배열을 사용한 스택

(element 갯수, 1보다는 과정 중요한 것은 개념.)

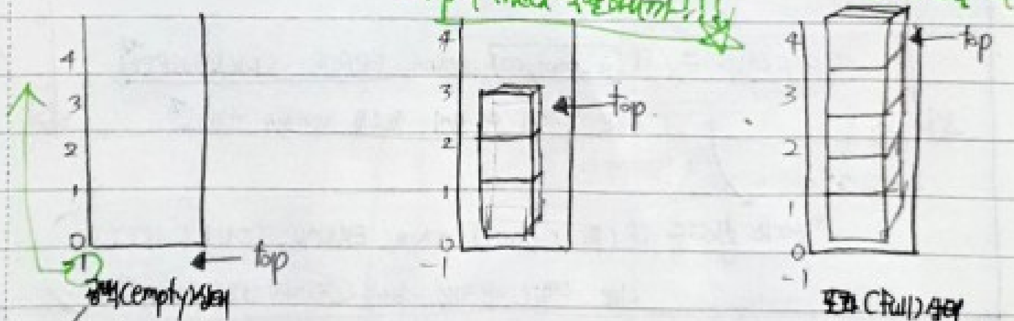
□ 1차원 배열 stack[MAX_STACK_SIZE]과 top = -1를 사용.

□ 스택이 공백(empty) 상태이면 (top == -1), 스택 포화(full) 상태이면 (top == MAX_STACK_SIZE - 1)

→ Array의 index로 사용함. (index가 0부터 시작하니 최대 element는 Array(스택)의 배열 원소보다 -1 이니는 이나로 열의 값을 크기보다 크 것임)

□ 가장 최근에 push한 요소는 stack[top]에 저장.

└ top이 index 역할하니까!!!

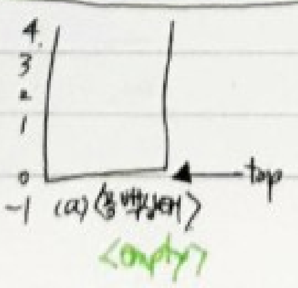


p.106 ~107 top변수 → 스택이 비어 있으면 -1의 값을 가짐

└→ top의 값이 0이면 배열의 인덱스 0에 데이터가 있다는 것을 의미하기 때문

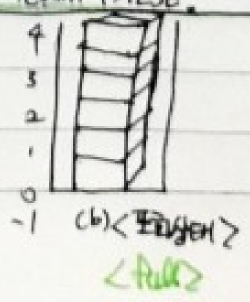
• is_empty(), is_full() 연산

is_empty(S):	is_full(S):
if top == -1 then return TRUE else return FALSE	if top == (MAX_STACK_SIZE - 1) then return TRUE. else return FALSE.



(a) <공백상태>

<empty>



(b) <포화상태>

<full>

p.108.

push() 연산

S: 스택, X: 넣을 element

is_full로 통해 먼저 꽉 차 있는지 체크

```
push(S, X):

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

① 제일 먼저 스택이 full 인지를 체크

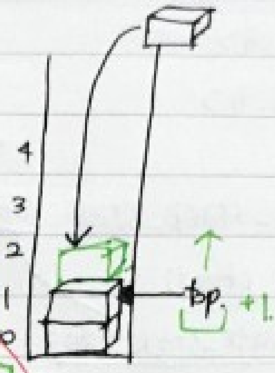
full 일 때 → then

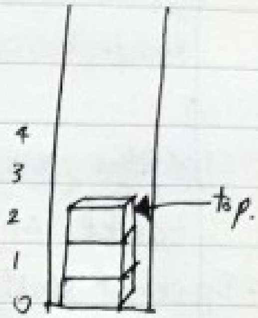
full 아닐 때 → else

→ top을 1 증가시키고 → top을 index로 넣어야 함

먼저 top+1 하고 그래서 공간을 넓힌 후 X를 집어넣어야 한다!!!

순서 중요!! (top이 가리키는 위치는 마지막으로 삽입되었던 요소라고 top을 증가시키지 않고 삽입하면 마지막 요소가 지워지게 된다.)





pop() 연산

is_empty로 통해 빈 것인지 확인

e: 스택에 있는 element

```
pop(S, X):

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

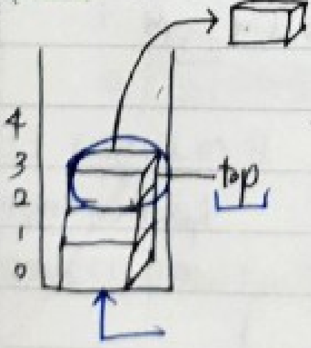
① 먼저 empty 인지 아닌지를 체크

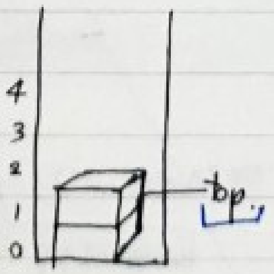
empty 일 때 → then

empty가 아닐 때 → else

1. 먼저 top에 있는 걸 끄집어내고
2. 그 다음 top을 1 줄여야 함. (감소시킴)

순서 중요!!!





• push
1. top을 1 증가시키고 (공간 확보!)
2. top을 index로 하여 값을 넣음 (추가!!) (element)

• pop
1. 먼저 element를 제거하고 (제거!! 끄집어내기)
2. top을 1 감소시킴 (빼고, 제거하고 남은 공간 없애기)

순서 반드시 알고 있어야 한다!!

p.168.

S : 스택, x : 넣을 element

is_full을 통해 먼저 꽉차있는지를 체크

• push() 연산

```
push(s, x):

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

① 제일 먼저 스택이 full 인지를 체크

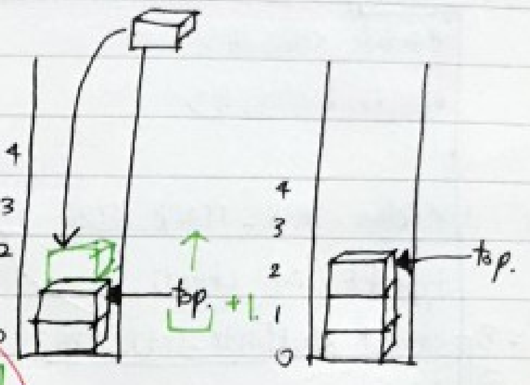
full 일때 → then error "overflow"

full 아닐때 → else

→ top을 1증가시키고

→ top을 index로 해서 넣어줌

먼저 top+1하고 그래서 공간을 넓힌 후 X를 집어넣어야 한다!!!



☆ 순서 중요!! ☆ (top이 가리키는 위치는 마지막으로 삽입되었던 요소임으로 top을 증가시키지 않고 삽입하면 마지막 요소가 지워지게 된다.)

• pop() 연산

is_empty를 통해 빈 공간인지 확인

e : 스택에 있는 element

```
pop(s, x):

if is_empty(s)
   then error "underflow"
   else [e ← stack[top]
        [top ← top-1
   return e.
```

① 먼저 empty인지 아닌지를 체크

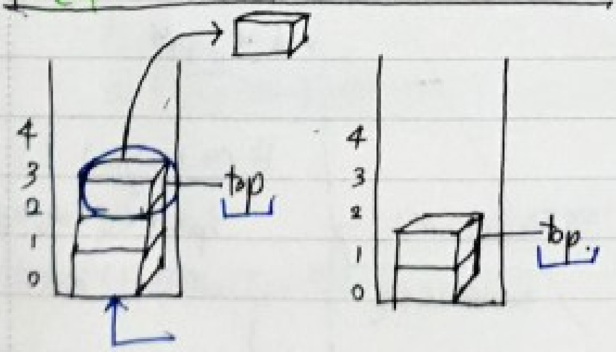
empty 일때 → then error "underflow"

empty가 아닐때 → else

→ 1. 먼저 top에 있는 걸 끄집어내고 ☆

→ 2. 그 다음 top을 1 줄여야 함. ☆ (감소시킴)

순서 중요!!! ☆



• push

1. top을 1증가시키고 (공간확보!)
2. top을 index로 하여 값을 넣음 (추가!!) (element)

• pop

1. 먼저 element를 제거하고 (제거!!, 끄집어내기)
2. top을 1 감소시킴 (빼고, 제거하고 남은 공간 없애기)

※순서 반드시 알고있어야 한다!!

구현방법 : ① 전역변수 / ② 구조체

① 전역변수로 구현 (전역변수로 구현하는 방법)

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

#define MAX_STACK_SIZE 100      // 스택의 최대크기
typedef int element;            // 데이터의 자료형
element stack[MAX_STACK_SIZE];  // 스택
int top = -1;                   // top.
```

(1/3)

```
// 공백 상태 검출함수
int is_empty()
{
    return (top == -1);
}
// 포화상태 검출함수
int is_full()
{
    return (top == (MAX_STACK_SIZE-1));
}
```

(2/3)

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

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

(3/3)

```
int main(void)
{
    push(1);
    push(2);
    push(3);
    printf("%d\n", pop());
    printf("%d\n", pop());
    printf("%d\n", pop());
    return 0;
}
```

출력 : 3 2 1

☆ 스택에 구조체 저장 (stack 구조체를 선언)
→ 스택에 저장되는 값의 제한 없음, 구조체 안에 필요한 모든 정보 넣기. (결국 계속 구조체형으로 저어 나가라!)
활용성 설계가 될 수 있음.

② • 구조체 형으로 구현 (스택의 요소를 구조체로 하기)

p.111~113

(1/3)

```c
typedef int element;
typedef struct {
    element stack[MAX_STACK_SIZE];   // (스택 배열)
    int top;                          // (top)
} StackType;

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

StackType의 pointer인 's'를 매개변수로 사용함

```c
// 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));
}
```

p.113~116.
• top과 stack배열을 하나의 구조체
=> 이 구조체의 포인터를 함수로 전달
└ call-by-reference

(2/3)

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

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

이건 프로그램의 코드 한 줄을 줄이기 위해 이렇게 바꾼 거지 이게 꼭 필요한 건 아니다. 결국 같은 코드로 실행된다.

동적 할당 대신 변수 선언 (전역변수)
구조체 + 구조체 포인터 가 (스택을 동적 메모리 할당으로 생성) 결합된 것임!!

(3/3)

```c
int main(void)
{
    StackType s;      // 기존 방식 (malloc 사용 시 → StackType *s; s = malloc(sizeof(StackType));)
    init_stack(&s);
    push(&s, 1);
    push(&s, 2);
    push(&s, 3);
    printf("%d \n", pop(&s));
    printf("%d \n", pop(&s));
    printf("%d \n", pop(&s));
}
```

malloc으로 생성 시 &는 필요 없음

3
2
1

free() 함수?

4.3. 동적 배열 스택.

"정적선언 → 동적선언"

p.117~119. • 앞에서는 모두 컴파일 시간에 크기가 결정되는 1차원 배열을 사용

⇒ 컴파일 시간에 필요한 스택의 크기를 알아야 하는 데 실제로는 아주 어려움

• C언어에는 malloc()을 호출하여 실행 시간에 메모리를 할당받을 수 있음 〈동적 할당!〉

↳ 필요할 때마다 스택의 크기를 동적으로 늘릴 수 있다!

```
typedef int element;
typedef struct {
        element *data;        // data는 포인터로 정의된다.
        int capacity;         // 현재 크기
        int top;
} StackType;
```

• 스택이 만들어질 때에, 1개의 요소를 저장할 수 있는 공간을 일단 확보.

```
// 스택 생성 함수
void init_stack(StackType *s)
{
    s->top = -1;
    s->capacity = 1;
    s->data = (element *)malloc(s->capacity * sizeof(element));
}
// 스택 삭제 함수
void delete(StackType *s)
{
    free(s);
}
```

• 가장 큰 변화가 있는 함수는 push()이다. 공간이 부족하면 메모리를 2배로 더 확보.

```
void push(StackType *s, element item)
{
    if (is_full(s)) {
        s->capacity *= 2;
        s->data = (element *) realloc(s->data, s->capacity * sizeof(element));
    }
    s->data[++(s->top)] = item;
}
```

realloc()
: 동적 메모리의 크기를 변경하는 함수,
현재 내용은 유지하면서 주어진 크기로 동적 메모리를 다시 할당

4.4 스택의 응용 : 괄호 검사 문제.

p.119 ~ 123

- 스택의 응용 : 괄호 검사.
  - 괄호의 종류: 대괄호(Square brackets) [,] / 중괄호(braces) { , } / 소괄호(parenthesis) : (,)
  - 매칭 (matching) 괄호의 조건
    1. 왼쪽 괄호의 개수와 오른쪽 괄호의 개수가 같아야 한다.
    2. 같은 종류의 괄호에서 왼쪽 괄호가 오른쪽 괄호보다 먼저 나와야 한다.
    3. 서로 다른 종류의 왼쪽 괄호와 오른쪽 괄호 쌍은 서로 교차하면 안된다.
  - 괄호 검사 보기.

```
{A[(i+1)]=0;}                    오류없음.
if((i==0)&&(j==0))?              조건 1 위반 L/R의 개수가 다름.
A[10] = B]10]                    조건 2 위반 (같은 또 같음)(열려야 되는데 닫힘)
A[(i+1])=0;                      조건 3 위반.
```

↳ 이렇게 맺어버리면, 교차해버리면 안됨.

➡ 괄호 검사의 오류를 검사하는데 스택을 사용

- 괄호 검사 알고리즘. (1/2).
  - 알고리즘 개요. 입력 → 논리적으로 괄호를 실제적으로 검사.
    - 입력에 있는 모든 괄호에 대하여 순서적으로
      1. 괄호가 왼쪽 괄호이면 스택에 push함.
      2. 괄호가 오른쪽 괄호이면 스택에서 괄호를 pop한 후 짝이 맞는 괄호인지 점검함.
         - pop할 때 스택이 비어 있으면 조건 1 과 2 위반
         - 짝이 맞지 않는 괄호이면 조건 3 위반
    - 스택에 괄호가 남아 있으면 조건 1 위반
      ↳ 왼쪽 괄호가 오른쪽보다 더 많다 (왼쪽 괄호 > 오른쪽 괄호)

왼쪽 괄호가 나오면 스택에 push하고

오른쪽 괄호가 나오면 짝이 맞는지 검사

↓

스택이 비어있지 않으면 문제 X,

스택이 비어 있으면 문제O,

+

짝이 안맞아도 문제O.

p.121

• 괄호 검사 알고리즘 (2/2)

```
check_matching(expr):

while (입력 expr의 끝이 아니면)
    ch ← expr의 다음 글자
    switch (ch)
        case '{': case '[': case '(':
            push(ch)
            break
        case '}': case ']': case ')':
            if (스택이 비어 있으면)
                then 오류
            else open_ch = pop()
                if (ch와 open_ch가 같은 짝이 아니면)
                    then 오류
    break
if (스택이 비어 있지 않으면)
    then 오류
```

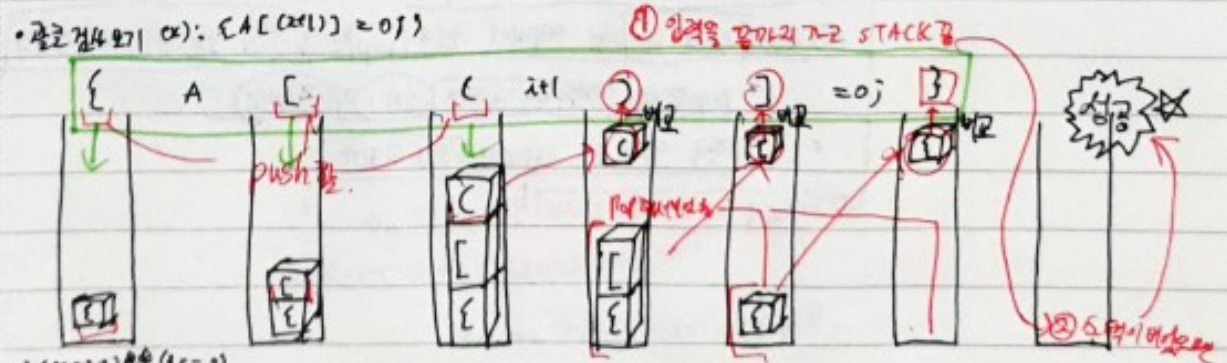
완전 괄호이면 스택에 push.

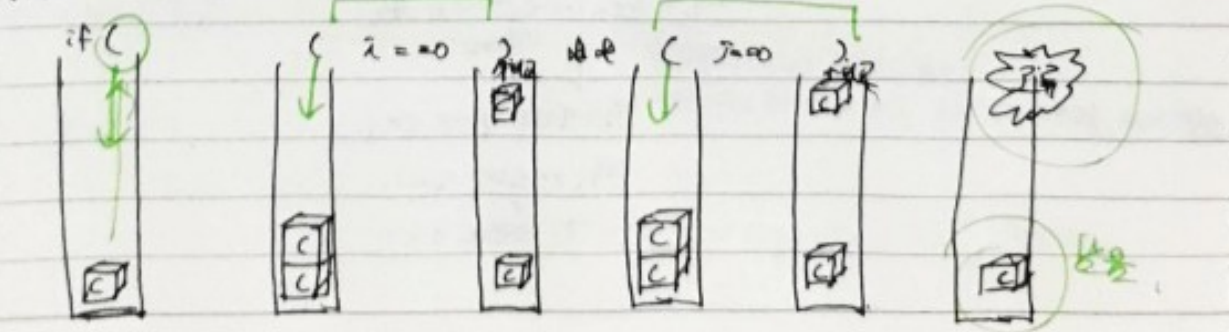
오른쪽 괄호이면 스택에 pop한 것과 비교

• 장점 3단

교재 p.122 참고

• 괄호 검사 예) : {A[(i+1)] = 0;}





p.124~134 . 4.5 스택의 응용 : 후위 표기수식의 계산

p.124~128 • 스택의 응용 : 수식 표기변환 및 계산

- operation, 식 — operator (연산자) / operand (피연산자)
- 2 + 3 = 5

- □ 수식 표기 방법 (3가지 방법 / <전위> <중위> <후위>)
- □ ① 중위 —> 후위 표기식 변환 → 후위로 바꾸면 계산과정이 쉬워짐 (괄호없이)
- □ ② 후위 (postfix) 표기식의 계산.
  - stack을 사용

사람이 가장 많이 쓰는 표기법

• 수식 표기방법.

□ 수식 표기 방법은 연산자 (operator) 및 operand (피연산자)의 위치에 따라 아래와 같이 3가지 종류가 있음.

□ 중위 (infix), 전위 (prefix), 후위 (postfix) 표기법. (연산자 위치에 따른 기준)

중위 표기법	전위 표기법 → 앞쪽에	후위 표기법 → 뒤쪽에 연산자 위치
2+3*4	+2*34	234*+
a*b+5	+*ab5	ab*5+
(1+2)*7	*+127	12+7*

(add (2,3) 같은 형태; 연산자의 위치)

이렇게 숫자 2개 연속으로 붙은 거에 연산자 붙은 거는 우선순위가 높다!!

□ 컴퓨터에서의 중위 표기 수식의 계산 순서

□ 중위 표기식 <변환> 후위 표기식 <연산> 계산.

ex) 2+3*4 → 234*+ → 14

□ 위 표기식 변환 및 계산 (연산) 과정에 스택을 사용.

p.124 • 컴파일러에서 후위 표기방법을 선호하는 이유?

⇒ 후위 표기 방식은 괄호가 필요 X ⊕ 괄호의 사용 없이도 우선순위 표현가능 때문에 단순하고 빠르게 계산됨

└ 수식을 읽으면서 바로 계산할 수 있는 방법

p-128

• 중위 → 후위 표기식 변환

□ 중위 표기식과 후위 표기식의 관계

a 피연산자(operand)의 순서는 동일

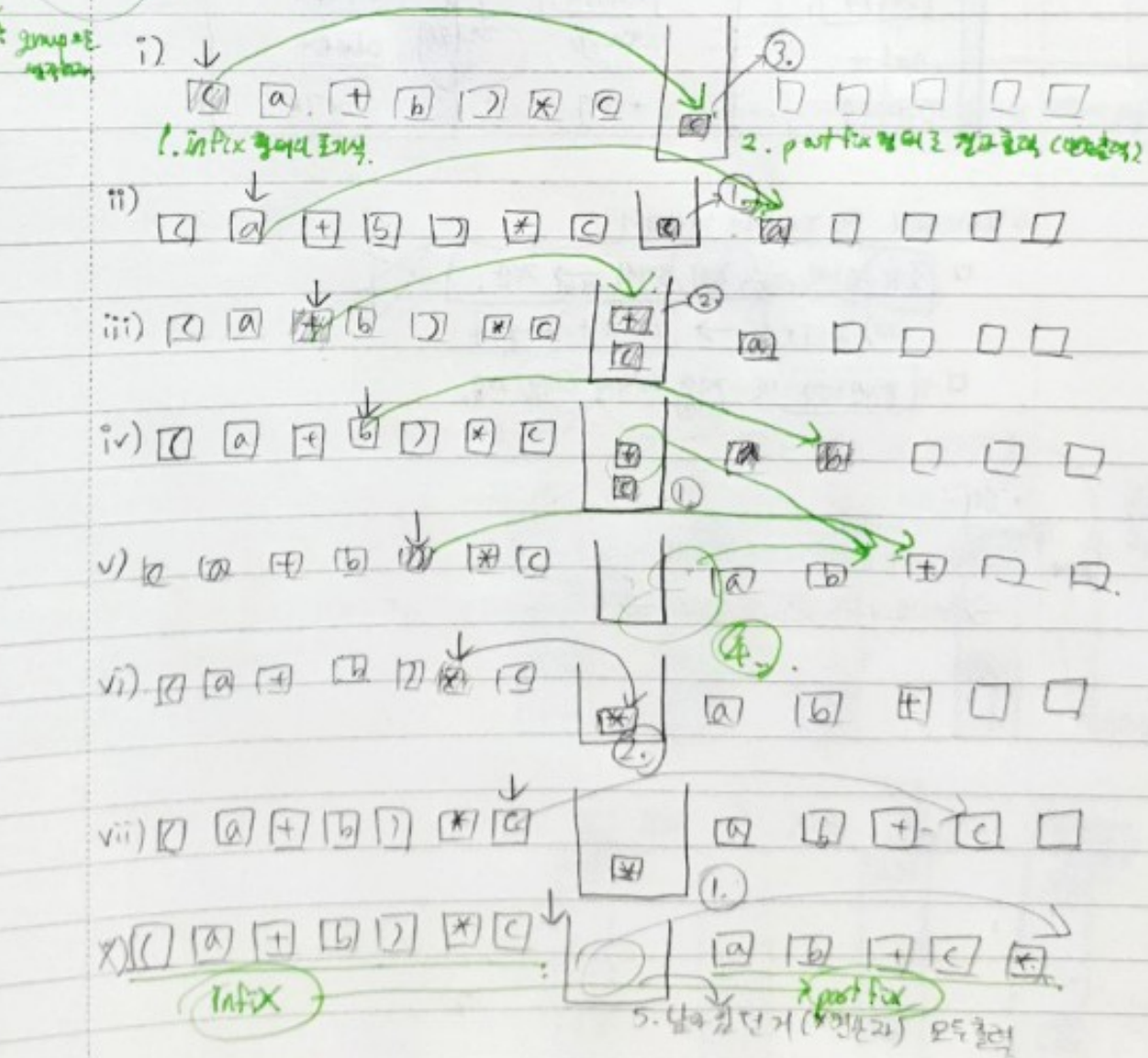
b 연산자(operator)들의 순서만 다름 (역 순서 + 우선 순위 순서)

보기: 2+3*4 ⟶ 234*+

개발자로 역할로 나오면서 우선순위(괄호, 곱하기, 더하기)를 지켜주는 함수.

□ 중위 → 후위 표기식 변환 알고리즘

1. 피연산자를 만나면 그대로 출력.
2. 연산자 ch를 만나면 스택에서 우선 순위가 같거나 높은 연산자를 모두 pop하여 출력한 후, 연산자 ch를 push.
3. 왼쪽 괄호는 스택에 저장
4. 오른쪽 괄호가 나오면 왼쪽 괄호가 나올 때까지 연산자를 pop하여 출력
5. 식이 끝까지 간 후 스택에 남아있는 모든 연산자는 pop하여 출력.



(중위→후위변환 알고리즘)

```
infix_to_postfix(exp):

  스택 s를 생성하고 초기화
  while (exp에 처리할 문자가 남아 있으면)
      ch ← 다음에 처리할 문자.
      switch (ch)
        case 피연산자:                                   // 1 피연산자
            ch를 출력
            break;
        case 연산자:                                     // 2 연산자
            while (peek(s)의 우선순위 ≥ ch의 우선순위)
                do e ← pop(s)      → 스택 s안에 있는 연산자들을 인식해주는 함수 ← (peek(s))
                   e를 출력
            push(s, ch);           ← 높지않으면/같지않으면 ch를 그냥 push.
            break;
        case 왼쪽 괄호:                                  // 3. 왼쪽 괄호
            push(s, ch);           ← 그냥 푸시.
            break;
        case 오른쪽 괄호:                                // 4. 오른쪽 괄호
            e ← pop(s);
            while (e ≠ 왼쪽 괄호)    왼쪽 괄호 나올 때까지
                do e를 출력
                   e ← pop(s);
            break;
  while (not is_empty(s))                              // 5. 스택에 남아있는 연산자 →
      do e ← pop(s)
         e를 출력.
```

· 실제코드는
교재 p.128~133까지
코드 참고

2+3 → 23+ → pop / 연산 / push

p.124 ~128.

• 후위 표기식의 계산

□ 수식을 왼쪽에서 오른쪽으로 스캔하면서

① 피연산자(operand)이면 스택에 push.

② 연산자(operator)이면 연산에 필요한 2개의 피연산자(operand)를 스택에서 pop한 후, 연산하고, 연산 결과를 push.

• 후위 표기식의 계산 알고리즘

스택을 생성하고 초기화한다.

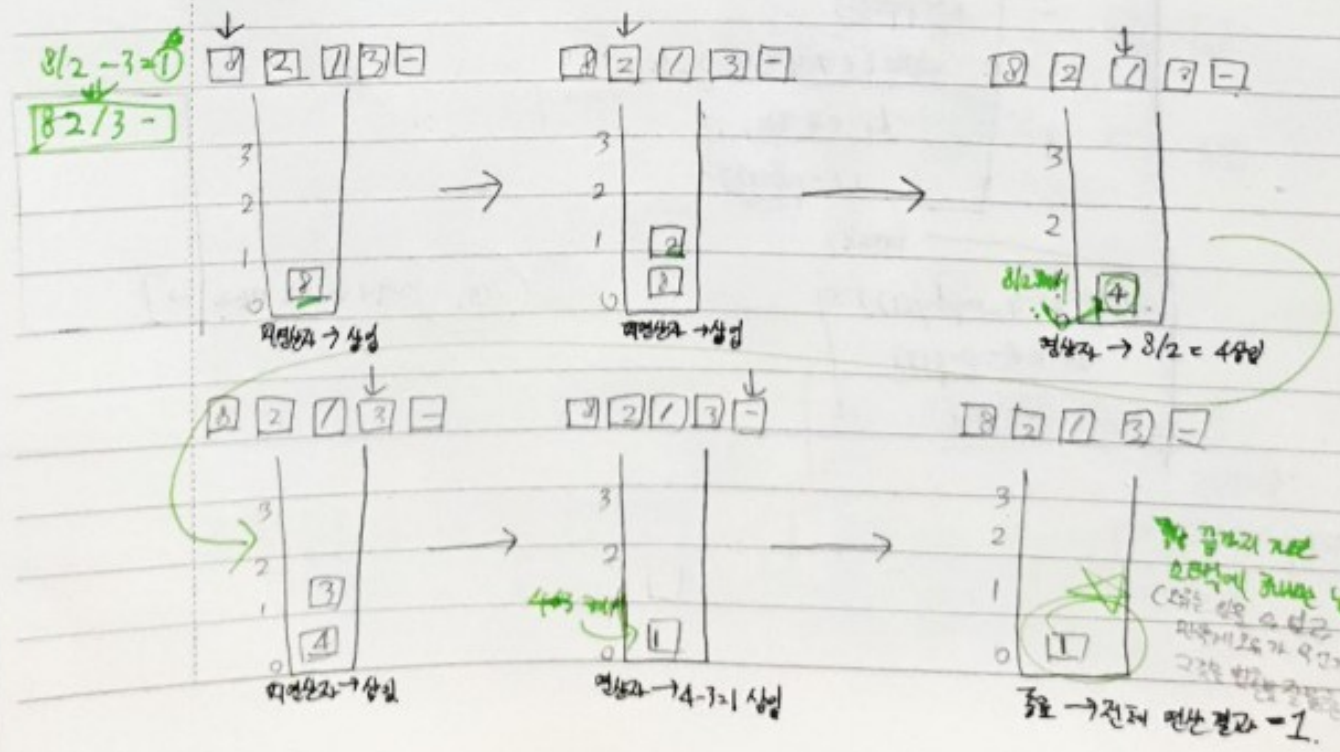
```
for 항목 in 후위 표기식
  do
    if (항목이 피연산자이면)
      then push(item)
    if (항목이 연산자 op이면)
      then second ← pop()
           first ← pop()
           result ← (first op second)
           push(result)
final_result ← pop(s))
```

(8 1 2 - 3) → 8 2 1 3 -

들어올	스택			
8	8			
2	8	2		
1	4			
3	4	3		
-	1			
3	1	3		
2	1	3	2	
+	1	6		
+	7			

8 1 2 - 3 + 3 * 2

8 2 1 3 -



4.6 스택의 응용: 미로문제

p.135~141

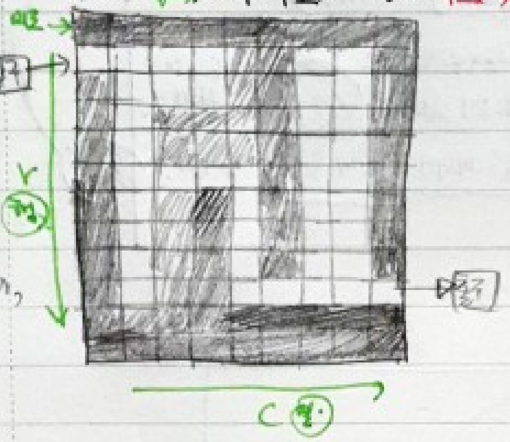
• 스택의 응용 : 미로 탐색 문제

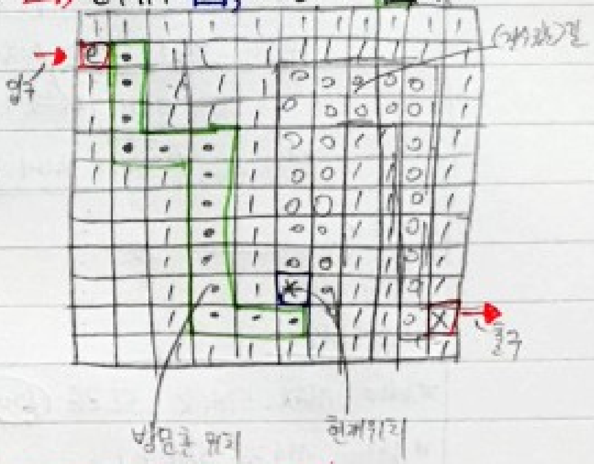
★ r: row / c: column
행(가로)　열(세로).

- 미로 탐색 문제는 미로의 입구에서 출구까지 가는 길을 찾는 문제
- 현재의 위치에서 갈 수 있는 위치(r,c)를 스택에 push해 놓았다가 더 이상 갈 곳이 없으면 스택에서 pop하여 다음 위치를 가져옴.
- 미로는 2차원 배열로 표현 → 갈 수 있는 곳.

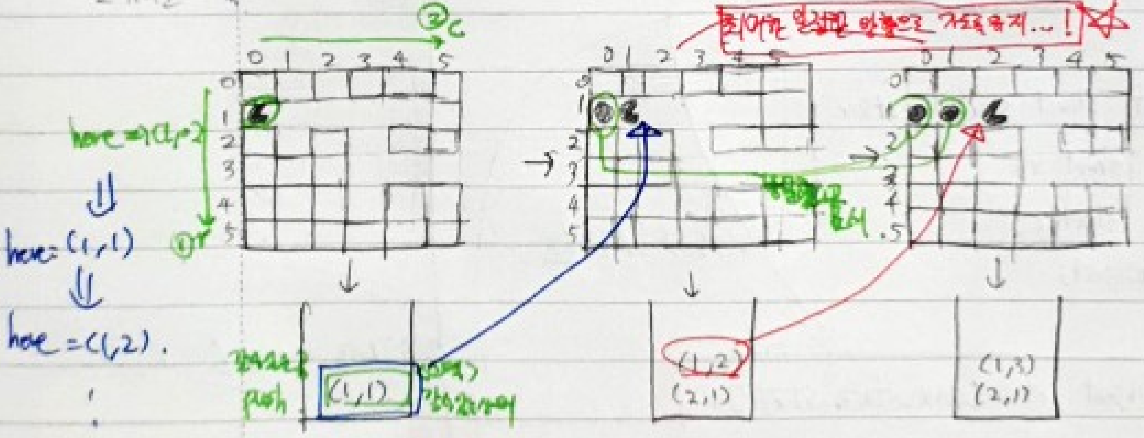
★(길 = 0, 벽 = 1　입구 = e, 출구 = x, 현재 위치 = *, 방문한 위치 = .)★

• 미로를 탈출하는 방법
⇒ 기본적으로 시행착오 방법
↳ 하나의 경로를 선택하여 한 번 시도해보고 안되면 다시 다른 경로를 시도하는 것





• 현재의 위치에서 갈 수 있는 위치를 스택에 push해놨다가, 더 이상 갈 수 없으면 스택에서 pop해서 다음 위치를 가져옴



★ 갈 수 있는 모든 방향(좌우상하, 위아래, 左,右)를 모두 탐색해야한다!!

이런 식으로 순차적으로 가보고 막히면 돌아오는 등을 반복하면서 미로의 탈출구를 찾아낸다.

p.137 • 미로 탐색 알고리즘

```
스택 S와 출구의 위치 x, 현재의 위치를 초기화
while (현재 위치가 출구가 아니면)
    do 현재 위치를 방문한 것으로 표기.                              //1
       if (현재 위치의 위, 아래, 왼쪽, 오른쪽 위치가 아직 방문되지 않았고,
           갈 수 있으면)
           then 그 위치들을 스택에 push.                            //2.
       if (is_empty(s))
           then 실패                                                //3.
           else 스택에서 하나의 위치를 꺼내어 현재 위치로 만든다.       //4.
성공;
```

1, 2, 3의 순서 (case 매칭!! 오호!!!)

• 미로 프로그램

(1/3)

```
#define MAX_STACK_SIZE 100
#define MAZE_SIZE 6

typedef struct StackObjectRec {
        short r;
        short c;
} StackObject;

StackObject stack[MAX_STACK_SIZE];
int top = -1;
StackObject here = {1,0}, entry = {1,0};

char maze[MAZE_SIZE][MAZE_SIZE] = {
        {'1', '1', '1', '1', '1', '1'},
        {'e', '0', '1', '0', '0', '1'},
        {'1', '0', '0', '0', '1', '1'},
        {'1', '0', '1', '0', '1', '1'},
        {'1', '0', '1', '0', '0', 'x'},
        {'1', '1', '1', '1', '1', '1'},
};
```

r, c를 struct로 만듦.

가로 세로 길이가 6

— stack 구현에 필요한 부분 (...top = -1로 시작)

p.140~141

// 위치를 스택에 삽입

(2/3)

```
void push_loc(StackType *s, int r, int c)
{
        if (r < 0 || c < 0) return;
        if (maze[r][c] != '1' && maze[r][c] != '.') {
                element tmp;
                tmp.r = r;
                tmp.c = c;
                push(s, tmp);
        }
}
```

이 조건을 만족해야 한다!!
1이고, . 이면 안된다. <도달하면 안되니까요>

r,c가 벽이 아니고 and 방문했던 곳이 아니면!!

두 값을 우리는 스택에 push한다!

// 미로를 화면에 출력한다.

```
void print_maze(char maze[MAZE_SIZE][MAZE_SIZE]) { ... }
```

r, c for 중첩문으로 출력

// 스택을 화면에 출력한다.

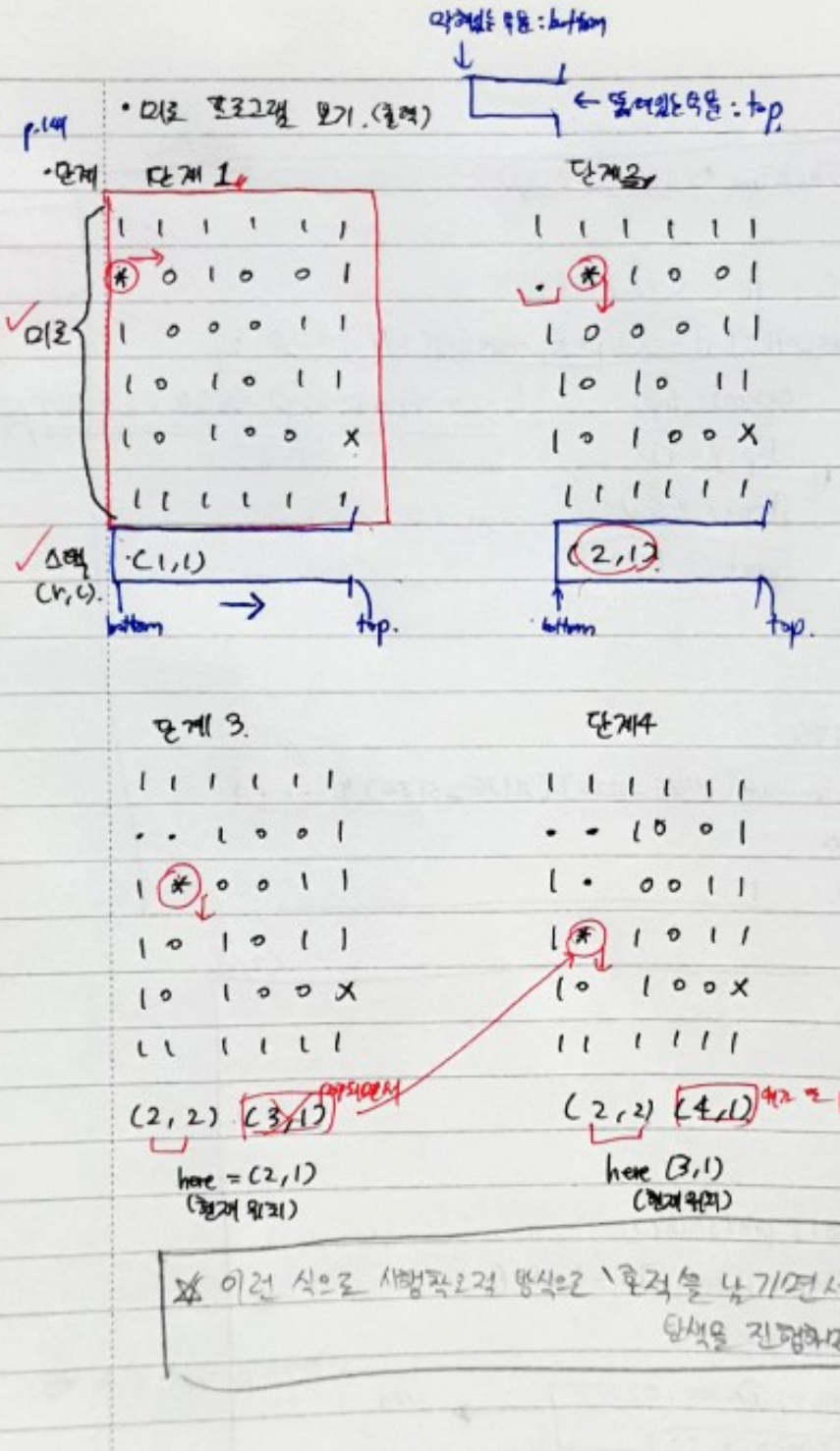
```
void print_stack() { ... }
```

(3/3)

maze → 미로

```
void main()
{
    int r, c;
    here = entry;
    printMaze(maze); printStack();
    while (maze[here.r][here.c] != 'x'){
        printMaze(maze);
        r = here.r; c = here.c;        //1
        maze[r][c] = '.';
        pushLoc(r-1, c); pushLoc(r+1, c);        //2
        pushLoc(r, c-1); pushLoc(r, c+1);
        printStack(); // 스택 출력
        if(isEmpty) {        //3
            printf("실패\n"); return;
        }
        else        //4.
            here = pop();
    }
    printf("성공\n");
}
```

//1 현재위치를 방문한 곳으로 표등.
방문한 위치 표시.
//2 → 다음으로 갈 수 있는 위치를 스택에 넣기 (위, 아래, 왼쪽 오른쪽 4방향)
실패 → 길이 4개 통과하지 않아서 못찾는다.
//3, //4 → 같은 거 → 다른 거 다고 통 ~~
pop 해서 다음 위치로 이동.


마지막 위치 : bottom
← 돌아왔던 위치 : top
p.144
• 미로 프로그램 보기 (절차)
• 단계
단계 1
단계 2
미로
스택
(1,1)
(2,1)
bottom
top
단계 3
단계 4
(2,2)
(3,1)
(4,1)
here = (2,1)
(현재 위치)
here (3,1)
(현재 위치)
※ 이런 식으로 시행착오적 방식으로 좌표를 넣기/빼면서
탐색을 진행하면 된다.