제4장

스택 (Stack)

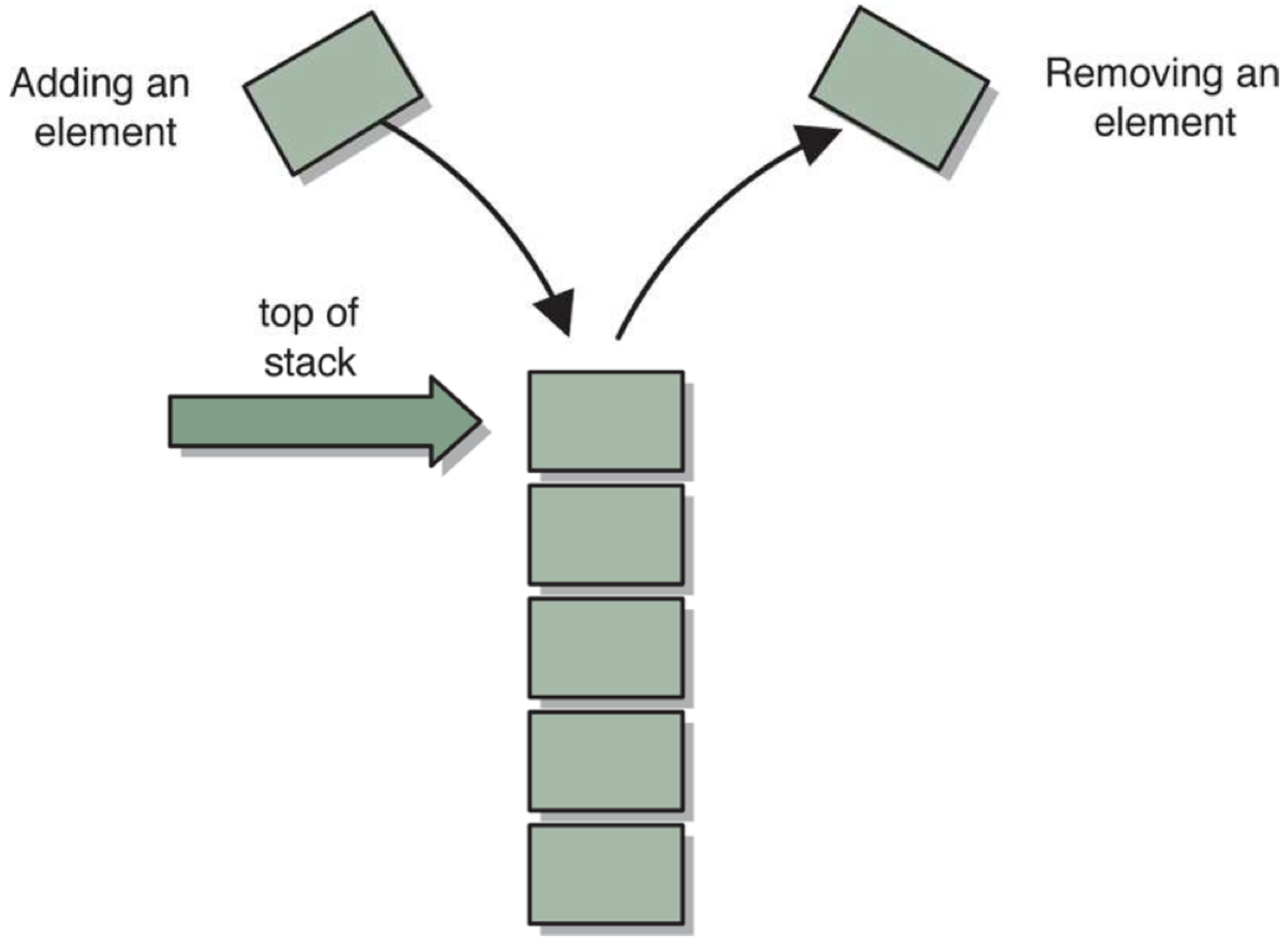
스택

개념과 응용

스택

- 스택은 일종의 리스트
- 단, 데이터의 삽입과 삭제가 한쪽 끝에서만 이루어짐
- **LIFO (Last-In, First-Out)**
- 삽입/삭제가 일어나는 쪽을 스택의 **top**이라고 부름

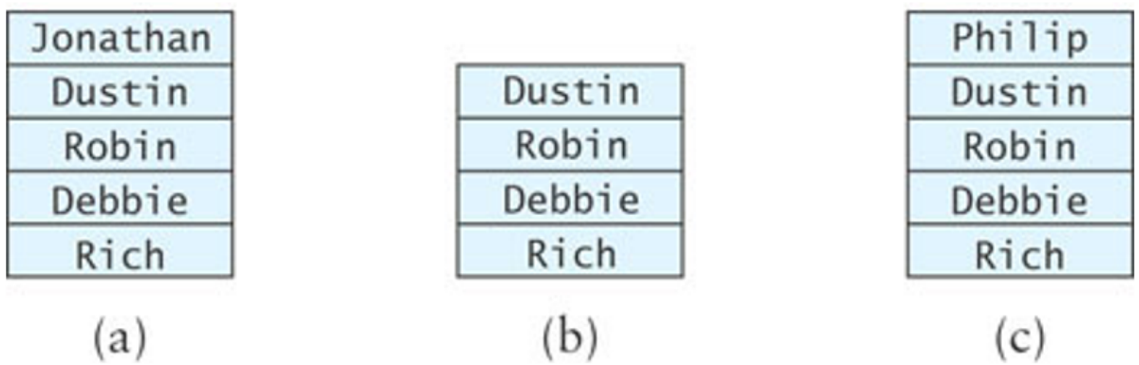
스택



스택이 지원하는 연산

- **push:** 스택에 새로운 원소를 삽입하는 연산
- **pop:** 스택의 **top**에 있는 원소를 스택에서 제거하고 반환
- **peek:** 스택 **top**의 원소를 제거하지 않고 반환
- **is_empty:** 스택이 비었는지 검사

스택의 연산의 예




```
push("Rich");
push("Debbie");
push("Robin");
push("Dustin");
push("Jonathan");                  // 스택의 상태는 (a)가 됨

char *str1 = peek();               // str1은 "Jonathan"이 됨
char *str2 = pop();                // str2는 "Jonathan"이 되고 스택의 상태는 (b)로 바뀜
push("Philip");                    //  스택의 상태가 (c)로 바뀜
```

스택 응용 예: 괄호 검사 문제

- 입력 수식의 괄호가 올바른지 검사

  예: [a + b * { c / (d - e) }] + (d / e)

- 단순히 여는 괄호와 닫는 괄호의 개수 비교 만으로는 부족
- 스택을 이용하여 검사
  - 여는 괄호는 스택에 push
  - 닫는 괄호가 나오면 스택에서 pop한 후, 두 괄호가 같은 유형이어야
  - 마지막에 스택에 남는 괄호가 없어야

paren_checker.c

```c
#include <stdio.h>
#include <string.h>
#include "stack.h"

#define MAX_LENGTH 100

char OPEN[] = "([{";
char CLOSE[] = ")]}";

int is_balanced(char *expr);
int is_open(char ch);
int is_close(char ch);

int main()
{
    char expr[MAX_LENGTH];
    scanf("%s", expr);
    if (is_balanced(expr))
        printf("%s: balanced.\n", expr);
    else
        printf("%s: unbalanced.\n", expr);
}
```

문자(char)들을 저장하는 스택이
stack.c 파일에 구현되어 있다고 가정한다.

입력은 괄호만으로 이루어진 하나의 문자열이다.

paren_checker.c

```
int is_balanced(char *expr)  {
    int balanced = 1;
    int index = 0;
    while (balanced && index < strlen(expr)) {
        char ch = expr[index];
        if (is_open(ch)>-1)
            push(ch);
        else if (is_close(ch)>-1) {
            if (is_empty()) {
                balanced = 0;
                break;
            }
            char top_ch = pop();
            if (is_open(top_ch) != is_close(ch)) {
                balanced = 0;
            }
        }
        index++;
    }
    return (balanced == 1 && is_empty() == 1);
}
```

여는 괄호는 스택에 push한다.

닫는 괄호가 나오면 스택에서 pop하여

같은 유형의 괄호인지 검사한다.

paren_checker.c

```
int is_close(char ch) {
    for (int i=0; i<strlen(CLOSE); i++) {
        if (CLOSE[i] == ch)
            return i;
    }
    return -1;
}

int is_open(char ch) {
    for (int i=0; i<strlen(OPEN); i++)
        if (OPEN[i] == ch)
            return i;
    return -1;
}
```

소괄호는 0, 대괄호는 1, 중괄호는 2를 반환하고

여는 괄호가 아니면 -1을 반환한다.

스택의 구현

배열 혹은 연결리스트를 이용한

배열을 이용한 구현

```
#include "stack.h"
#define MAX_CAPACITY 100

char stack[MAX_CAPACITY];
int top = -1;       // index of the top element

void push(char ch) {
    if (is_full())
        return;
    top++;
    stack[top] = ch;
}

char pop() {
    char tmp = stack[top];
    top--;
    return tmp;
}
```

스택에 저장되는 데이터의 타입을 문자(char)라고 가정하자.

스택이 가득차면 더이상 push할 수 없다.

pop을 호출하기 전에 먼저 empty인지 검사해야 한다.

배열을 이용한 구현

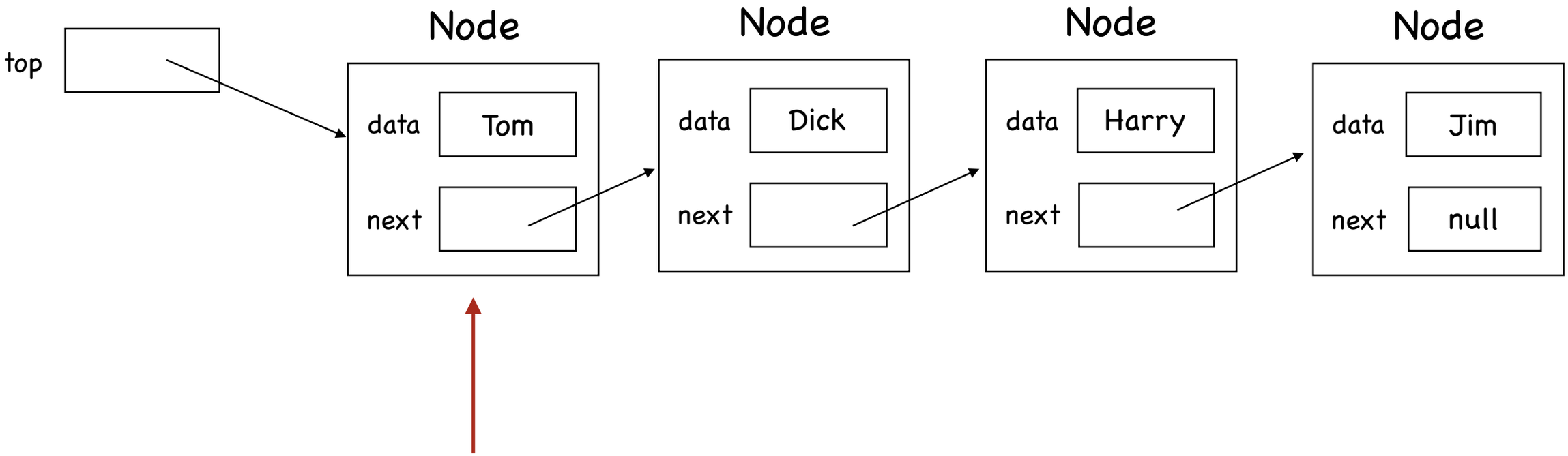
```
char peek() {
    return stack[top];
}
```

peek을 호출하기 전에 먼저 empty인지 검사해야 한다.

```
int is_empty() {
    return top==-1;
}

int is_full() {
    return top == MAX_CAPACITY-1;
}
```

연결리스트로 구현



연결리스트의 맨 앞이 노드를 삽입/삭제하기에 편하다.
따라서 여기를 stack의 top으로 한다.

연결리스트로 구현

```
struct node  {
    char *data;
    struct node *next;
}
typedef struct node Node;

Node *top = NULL;
```

문자열들을 저장하는 스택이라고 가정하자.

연결리스트로 구현

```
void push(char *item) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->data = item;
    p->next = top;
    top = p;
}
```

```
char *pop()  {
    if (is_empty())
        return NULL;
    char *result = top->data;
    top = top->next;
    return result;
}
```

연결리스트로 구현

```
char *peek() {
    if (is_empty())
        return NULL;
    return top->data;
}

int is_empty() {
    return top == NULL;
}
```

문제점

- 만약 스택이 동시에 2개 이상 필요하다면?
- 서로 다른 타입의 데이터를 저장할 스택이 필요하다면?

stackADT.h

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef int Item;

typedef struct stack_type *Stack;

Stack create();
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
void push(Stack s, Item i);
Item pop(Stack s);
Item peek(Stack s);

#endif
```

배열로 구현: stackADT.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

#define INIT_CAPACITY 100

struct stack_type {
    Item *contents;      /* 배열 */
    int top;
    int size;
};

static void terminate(const char *message)
{
    printf("%s\n", message);
    exit(1);
}
```

배열로 구현: stackADT.c

```
Stack create()
{
    Stack s = (Stack)malloc(sizeof(struct stack_type));
    if (s == NULL)
        terminate("Error in create: stack could not be created.");
    s->contents = (Item *)malloc(INIT_CAPACITY * sizeof(Item));
    if (s->contents == NULL) {
        free(s);
        terminate("Error in create: stack could not be created.");
    }
    s->top = -1;
    s->size = INIT_CAPACITY;
    return s;
}

void destroy(Stack s)
{
    free(s->contents);
    free(s);
}
```

배열로 구현: stackADT.c

```
void make_empty(Stack s)
{
    s->top = -1;
}
```

```
bool is_empty(Stack s)
{
    return s->top == -1;
}
```

배열로 구현: stackADT.c

```
void push(Stack s, Item i)
{
    if (is_full(s))
        reallocate(s);
    s->top++;
    s->contents[s->top] = i;
}

Item pop(Stack s)
{
    if (is_empty(s))
        terminate("Error in pop: stack is empty.");
    s->top--;
    return s->contents[s->top+1];
}

Item peek(Stack s)
{
    if (is_empty(s))
        terminate("Error in peek: stack is empty.");
    return s->contents[s->top];
}
```

배열로 구현: stackADT.c

```
void reallocate(Stack s)
{
    Item *tmp = (Item *)malloc(2 * s->size * sizeof(Item));
    if (tmp == NULL) {
        terminate("Error in create: stack could not be created.");
    }
    for (int i=0; i<s->size; i++)
        tmp[i] = s->contents[i];
    free(s->contents);
    s->contents = tmp;
    s->size *= 2;
}
```

사용 예

```
#include “stackADT.h”

int main()
{
    Stack s1 = create();
    Stack s2 = create();

    push(s1, 12);
    push(s2, 9);
    …

}
```

연결리스트로 구현: stackADT.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

struct node {
    Item data;
    struct node *next;
};

struct stack_type {
    struct node *top;
};

static void terminate(const char *message)
{
    printf("%s\n", message);
    exit(EXIT_FAILURE);
}
```

연결리스트로 구현: stackADT.c

```
Stack create()
{
    Stack s = malloc(sizeof(struct stack_type));
    if (s == NULL)
        terminate("Error in create: stack could not be created.");
    s->top = NULL;
    return s;
}
```

```
void destroy(Stack s)
{
    make_empty(s);
    free(s);
}

void make_empty(Stack s)
{
    while (!is_empty(s))
        pop(s);
}
```

연결리스트로 구현: stackADT.c

```
bool is_empty(Stack s)
{
    return s->top == NULL;
}

void push(Stack s, Item i)
{
    struct node *new_node = malloc(sizeof(struct node));
    if (new_node == NULL)
        terminate("Error in push: stack is full.");

    new_node->data = i;
    new_node->next = s->top;
    s->top = new_node;
}
```

연결리스트로 구현: stackADT.c

```
Item pop(Stack s)
{
    struct node *old_top;
    Item i;

    if (is_empty(s))
        terminate("Error in pop: stack is empty.");

    old_top = s->top;
    i = old_top->data;
    s->top = old_top->next;
    free(old_top);
    return i;
}

Item peek(Stack s)
{
    if (is_empty(s))
        terminate("Error in peek: stack is empty.");
    return s->top->data;
}
```

Generic ADTs

- **1가지 타입의 데이터 만을 저장할 수 있음**
  - 데이터 타입이 달라지면 Item 타입 정의를 수정해야 함
    - 서로 다른 타입의 데이터를 저장하는 2개의 스택을 만들 수 없음
- **`void *` 타입을 이용하여 generic한 스택을 구현할 수 있으나 단점이 있음**
- **객체지향 프로그래밍 언어**

스택 응용: 후위표기식(Post-Fix Expression)

후위표기식

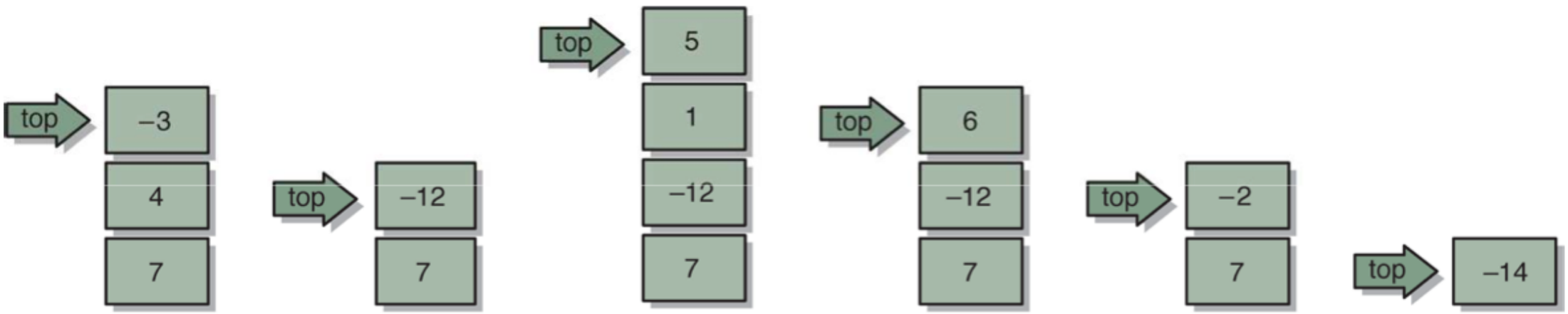
- **일반적으로 사용하는 수식의 표기법을 중위(infix) 표기식이라고 부름**
  - 연산자가 피연산자의 사이에 들어감
- **후위표기식(postfix expression)**
  - 연산자가 피연산자 뒤에 나옴

Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

후위표기식 계산

```
1.  create an empty stack of integers;
2.  while there remains a token   // 여기서 token은 연산자 혹은 피연산자이다.
3.      get the next token;
4.      if the token is an operand
5.          push the operand on the stack;
6.      else if the token is an operator
7.          pop the right operand off the stack;
8.          pop the left operand off the stack;
9.          evaluate the operation;
10.         push the result onto the stack;
11. pop the stack and return the result;
```

후위표기식 계산




```
7   4   -3   *   1   5   +   /   *
```

후위표기식 계산

```
static char OPERATORS[] = "+-*/";
```

이 프로그램이 지원하는 연산자들을 하나의 String에 모아두었다.

```
Stack operand_stack;
```

피연산자들을 저장할 스택이다.

```
int is_operator(char ch) {
    for (int i=0; i<strlen(OPERATORS); i++)
        if (OPERATORS[i] == ch)
            return i;
    return -1;
}
```

문자 ch가 연산자인지 검사한다. 아니면 -1을 반환한다.

후위표기식 계산

```
int eval(char *expr) {
    operand_stack = create();
    char *token = strtok(expr, " ");
    while(token != NULL) {
        if (token[0] >= '0' && token[0] <= '9') {    /* 피연산자 */
            int value = atoi(token);
            push(operand_stack, value);
        }
        else if (is_operator(token[0]) > -1) {          /* 연산자 */
            int result = eval_op(token[0]);
            push(operand_stack, result);
        }
        else {
            handle_exception("Syntax Error: invalid character encountered.");
        }
        token = strtok(NULL, " ");
}
```

모든 연산자와 피연산자가 공백문자로 구분되어 있다고 가정한다.

후위표기식 계산

```
if (is_empty(operand_stack))
    handle_exception("Syntax Error: Stack empty in eval_op.");
int answer = pop(operand_stack);
if (is_empty(operand_stack))
    return answer;
else {
    handle_exception("Syntax Error: Stack should be empty.");
        return -1;
    }
}

void handle_exception(const char *err_msg) {
    printf("%s\n", err_msg);
    exit(1);
}
```

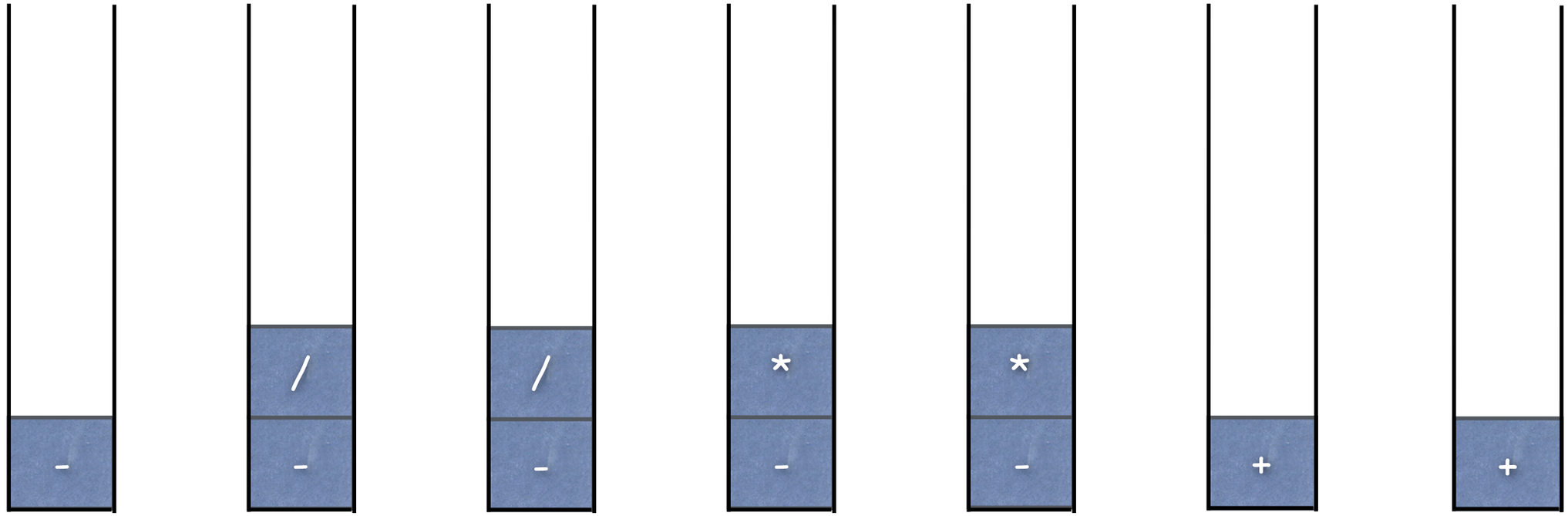
후위표기식 계산

```
int eval_op(char op) {
    if (is_empty(operand_stack))
        handle_exception("Syntax Error: Stack empty in eval_op.");
    int rhs = pop(operand_stack);
    if (is_empty(operand_stack))
        handle_exception("Syntax Error: Stack empty in eval_op.");
    int lhs = pop(operand_stack);
    int result = 0;
    switch (op) {
        case '+': result = lhs + rhs; break;
        case '-': result = lhs - rhs; break;
        case '*': result = lhs * rhs; break;
        case '/': result = lhs / rhs; break;
    }
    return result;
}
```

후위표기식으로 변환 - 괄호없는 경우

2 - 10 / 5 * 6 + 4

output 2 push -	output 10 push /	output 5	pop / push *	output 6	pop * pop - push +	output 4 pop +



output: 2 | 2 10 | 2 10 5 | 2 10 5 / | 2 10 5 / 6 | 2 10 5 / 6 * -

2 10 5 / 6 * - 4 +

후위표기식으로 변환 - 괄호없는 경우

- 중위 표기식을 처음부터 순서대로 읽으면서
- 피연산자는 즉시 출력한다.
- 모든 연산자는 일단 스택에 **push**한다.
- 단, 이때 스택 내에 우선순위가 자신보다 더 높은 연산자가 있으면 **pop**하여 출력한 후에 **push**한다.
- 수식이 종료되면 스택에 있는 모든 연산자를 **pop**하여 출력한다.

후위표기식으로 변환 - 괄호없는 경우

1. Initialise **`postfix`** to an empty string;
2. Create the operator stack;
3. while there are more tokens in the infix expression
4. Get the next token;
5. if the next token is an operand
6. Append it to **`postfix`**;
7. else if the next token is an operator
8. Call **`process_operator`** to process the operator;
9. else
10. Indicate a syntax error.
11. Pop the remaining operators off the operator stack and append them to **`postfix`**;

후위표기식으로 변환 - 괄호없는 경우

Algorithm for `process_operator`

1. **if** the operator stack is empty
2. Push the current operator onto the stack;
3. **else**
4. Peek the operator stack and let **`top_op`** be the top operator;
5. **if** the precedence of the current operator is greater than that of **`top_op`**
6. Push the current operator onto the stack;
7. **else**
8. **while** the stack is not empty and the precedence of the current operator is less than or equal to the precedence of **`top_op`**
9. Pop **`top_op`** off the stack and append it to **`postfix`**;
10. **if** the operator stack is not empty
11. Peek the operator stack and let **`top_op`** be the top operator;
12. Push the current operator onto the stack;

후위표기식으로 변환 - 괄호없는 경우

```
static char OPERATORS[] = "+-*/";
static int PRECEDENCE[] = {1, 1, 2, 2};
```

연산자 +, -, *, /의 우선순위를 순서대로 1, 1, 2, 2로 정의하였다. 큰 값이 더 높은 우선순위를 의미한다.

```
Stack operator_stack;
```

operator stack에는 char들이 저장된다.

후위표기식으로 변환 - 괄호없는 경우

```
char *convert(char *infix) {
    operator_stack = create();

    char *postfix = (char *)malloc(strlen(infix)+1);
    char *pos = postfix;

    char *token = strtok(infix, " ");
    while (token != NULL) {
        if (token[0] >= '0' && token[0] <= '9') {    // operand
            sprintf(pos, "%s ", token);
            pos += (strlen(token) + 1);
        }
        else if (is_operator(token[0]) > -1) {  // operator
            pos = process_op(token[0], pos);
        }
        else {
            handle_exception("Syntax Error: invalid character encountered.");
        }
        token = strtok(NULL, " ");
    }
```

변환된 postfix expression이 저장될 문자배열

모든 연산자와 피연산자가 공백문자로 구분되어 있다고 가정한다.

sprintf를 이용하여 문자열에 append한다.

process_op 함수가 연산자를 append한 후 문자열의 끝 주소를 반환한다.

후위표기식으로 변환 - 괄호없는 경우

```
while(!is_empty(operator_stack)) {
    char op = (char)pop(operator_stack);
    sprintf(pos, "%c ", op);
    pos += 2;
}
*pos = '\0';
return postfix;
}
```

마지막에 '\0'를 추가해서 문자열의 끝을 표시한다.

후위표기식으로 변환 - 괄호없는 경우

```
char *process_op(char op, char *pos) {
    if (is_empty(operator_stack))
        push(operator_stack, op);
    else {
        char top_op = peek(operator_stack);
        if (precedence(op) > precedence(top_op))
            push(operator_stack, op);
        else {
            while(!is_empty(operator_stack) && precedence(op) <=
                                                precedence(top_op)) {
                pop(operator_stack);  /* discard since it's top_op */
                sprintf(pos, "%c ", top_op);
                pos += 2;
                if (!is_empty(operator_stack))
                    top_op = (char)peek(operator_stack);
            }
            push(operator_stack, op);
        }
    }
    return pos;
}
```

후위표기식으로 변환 - 괄호없는 경우

```
int precedence(char op) {
    return PRECEDENCE[is_operator(op)];
}

int is_operator(char ch) {
    for (int i=0; i<strlen(OPERATORS); i++)
        if (OPERATORS[i] == ch)
            return i;
    return -1;
}

void handle_exception(const char *err_msg) {
    printf("%s\n", err_msg);
    exit(1);
}
```

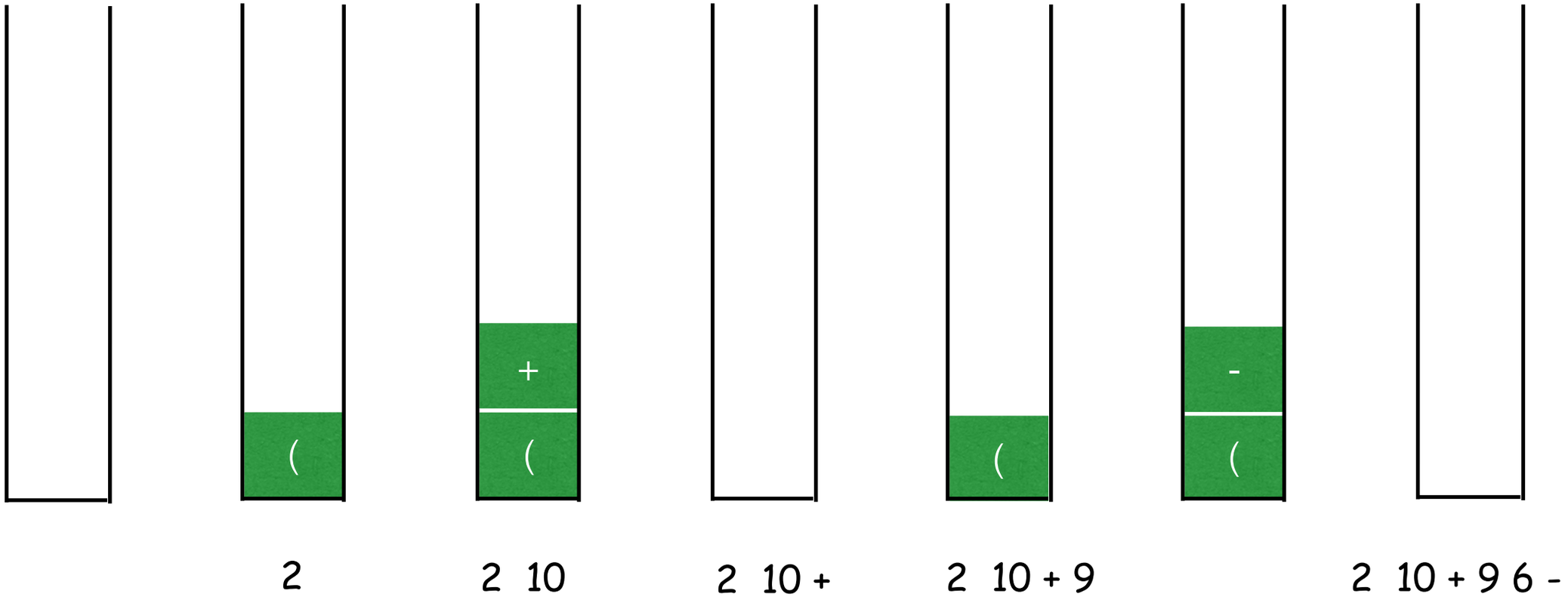
후위표기식으로 변환 - 괄호있는 경우

1. 여는 괄호는 무조건 스택에 **push**한다. 이때 스택 내의 어떤 연산자도 **pop**하지 않는다.
2. 어떤 연산자를 스택에 **push**할 때 스택의 **top**에 여는 괄호가 있으면 아무도 **pop**하지 않고 그냥 **push**한다.
3. 입력에 닫는 괄호가 나오면 스택에서 여는 괄호가 나올 때 까지 **pop**하여 출력한다. 닫는 괄호는 스택에 **push**하지 않는다.

후위표기식으로 변환

(2+10)/(9-6)

push (	push +	pop until (	push (	push -	output 6
output 2	output 10		output 9		pop until (



2 2 10 2 10 + 2 10 + 9 2 10 + 9 6 -

후위표기식으로 변환 - 괄호있는 경우

```
char OPERATORS[] = "+-*/()";
int PRECEDENCE[] = {1, 1, 2, 2, -1, -1};

Stack operator_stack;
```

여는 괄호의 우선순위를 -1로 정의하였다.
이렇게 하면 2번 규칙을 예외로 처리할 필요가 없다.

후위표기식으로 변환 - 괄호있는 경우

```
char *convert(char *infix) {
    operator_stack = create(100);
    char *postfix = (char *)malloc(strlen(infix)+1);
    char *pos = postfix;

    char *token = strtok(infix, " ");
    while (token != NULL) {

        /* 괄호가 없는 경우와 동일하다. */

    }

    while(!is_empty(operator_stack)) {
        char op = (char)pop(operator_stack);
        if (op == '(')
            handle_exception("Unmatched parenthesis.");
        sprintf(pos, "%c ", op);
        pos += 2;
    }
    *pos = '\0';
    return postfix;
}
```

모든 연산자와 피연산자가 공백문자로 구분되어 있다고 가정한다.

스택에 여는 괄호가 남아있어서는 안된다.

후위표기식으로 변환 - 괄호있는 경우

```
char *process_op(char op, char *pos) {
    if (is_empty(operator_stack) || op == '(')
        push(operator_stack, op);
    else {
        char top_op = peek(operator_stack);
        if (precedence(op) > precedence(top_op))
            push(operator_stack, op);
        else {
            while(!is_empty(operator_stack) &&
                                precedence(op) <= precedence(top_op)) {
                pop(operator_stack);
                if (top_op == '(')
                    break;
                sprintf(pos, "%c ", top_op);
                pos += 2;
                if (!is_empty(operator_stack))
                    top_op = (char)peek(operator_stack);
            }
            if (op != ')')
                push(operator_stack, op);
        }
    }
    return pos;
}
```

여는 괄호는 그냥 스택에 push한다.

op의 우선순위가 top_op보다 낮거나 같은데 top_op가 닫는 괄호이면 op는 닫는 괄호라는 의미이다.

닫는 괄호는 스택에 push하지 않는다.

main

```c
#define MAX_LENGTH 100

int main()
{
    char infix[MAX_LENGTH];
    read_line(stdin, infix, MAX_LENGTH);

    printf(" %s := ", infix);
    char *postfix = convert(infix);
    printf("%d\n", eval(postfix));
}
```

infix expression에 공백문자들이 포함 되어 있으므로 scanf로 읽을 수 없다.

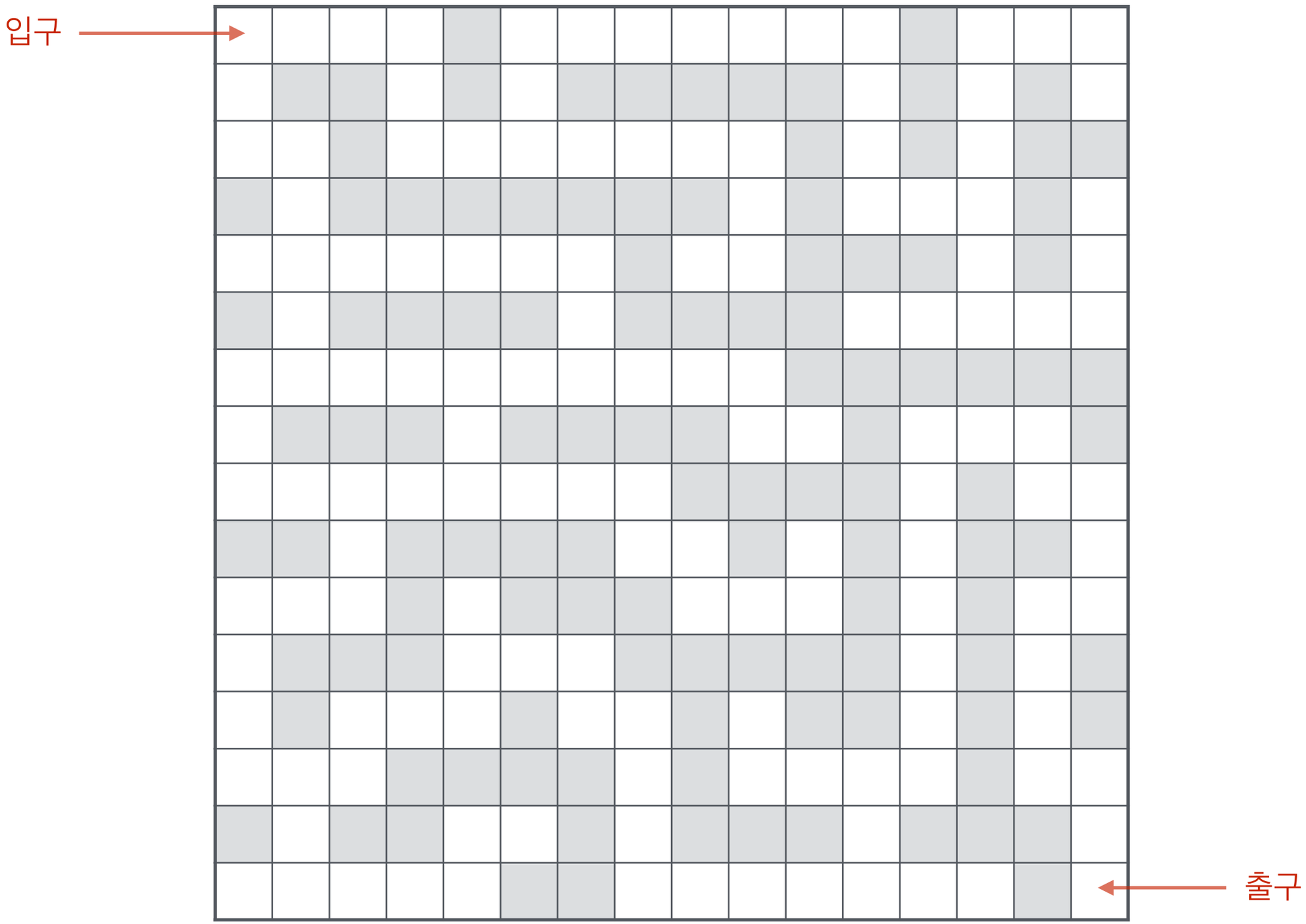
미흡한 점

- 피연산자는 양의 정수만 허용: 음수나 실수로 확장
- 모든 토큰들이 공백문자로 구분되어 있어야 함
- 일진(unary) 연산자의 처리: -(-2)
- right associativity를 가지는 연산자의 처리: 2^3^4
- 후위표기식으로 변환하는 일과 후위표기식을 계산하는 일을 하나로 합치기

미로찾기

Maze

미로찾기

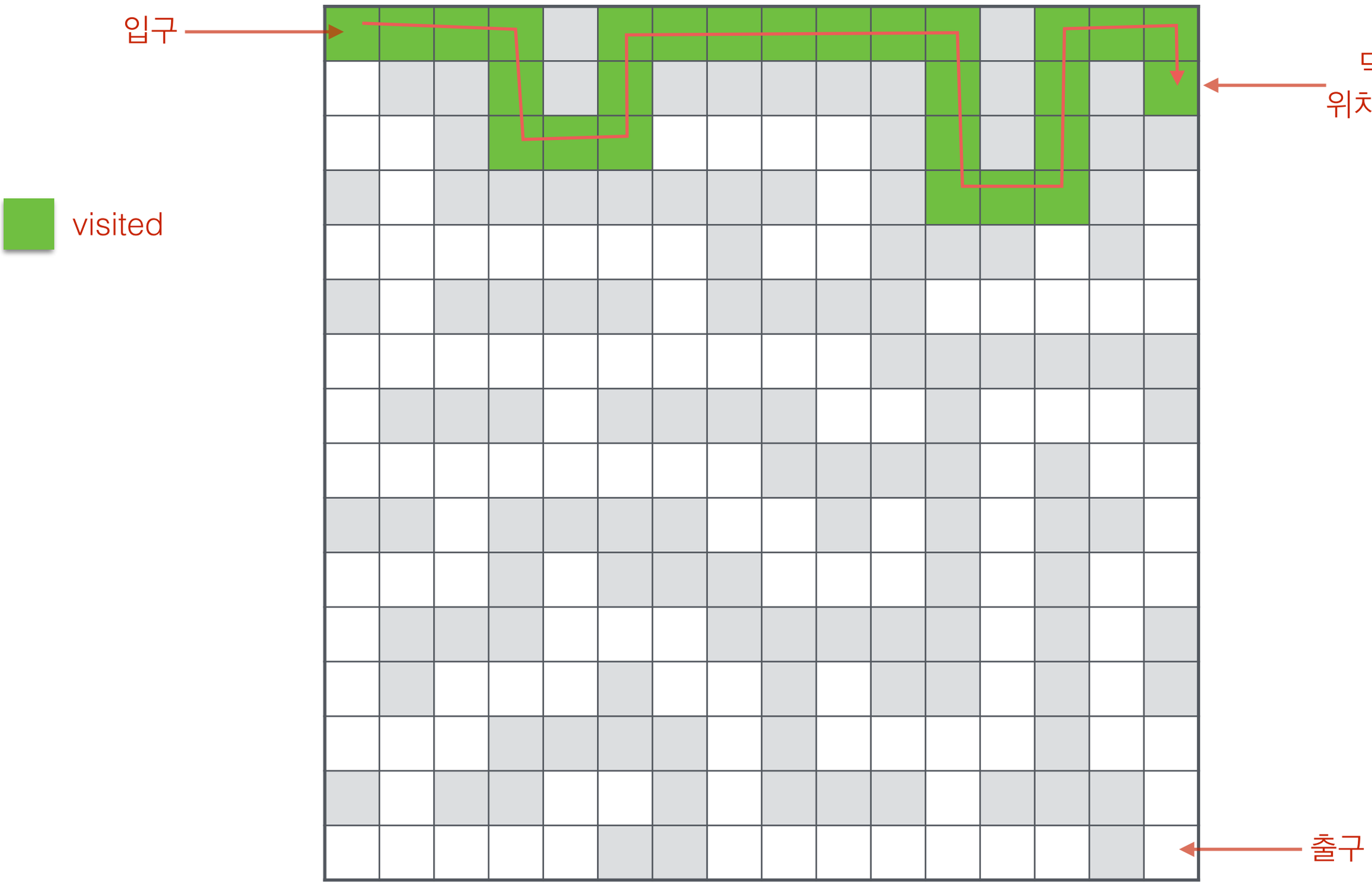


미로찾기

- **이미 방문한 위치에는 표시를 해서 무한루프를 방지한다.**
- **현재 위치에서 일정한 규칙으로 다음 위치로 이동한다.**
  - 북, 동, 남, 서의 순서로 검사하여,
  - 그 방향으로 갈 수 있으면, 즉 아직 안 가본 위치면서 벽이 아니면 그 방향으로 간다.
- **아무 방향으로도 갈 수 없으면 그 위치에 오기 직전 위치로 되돌아 간다.**

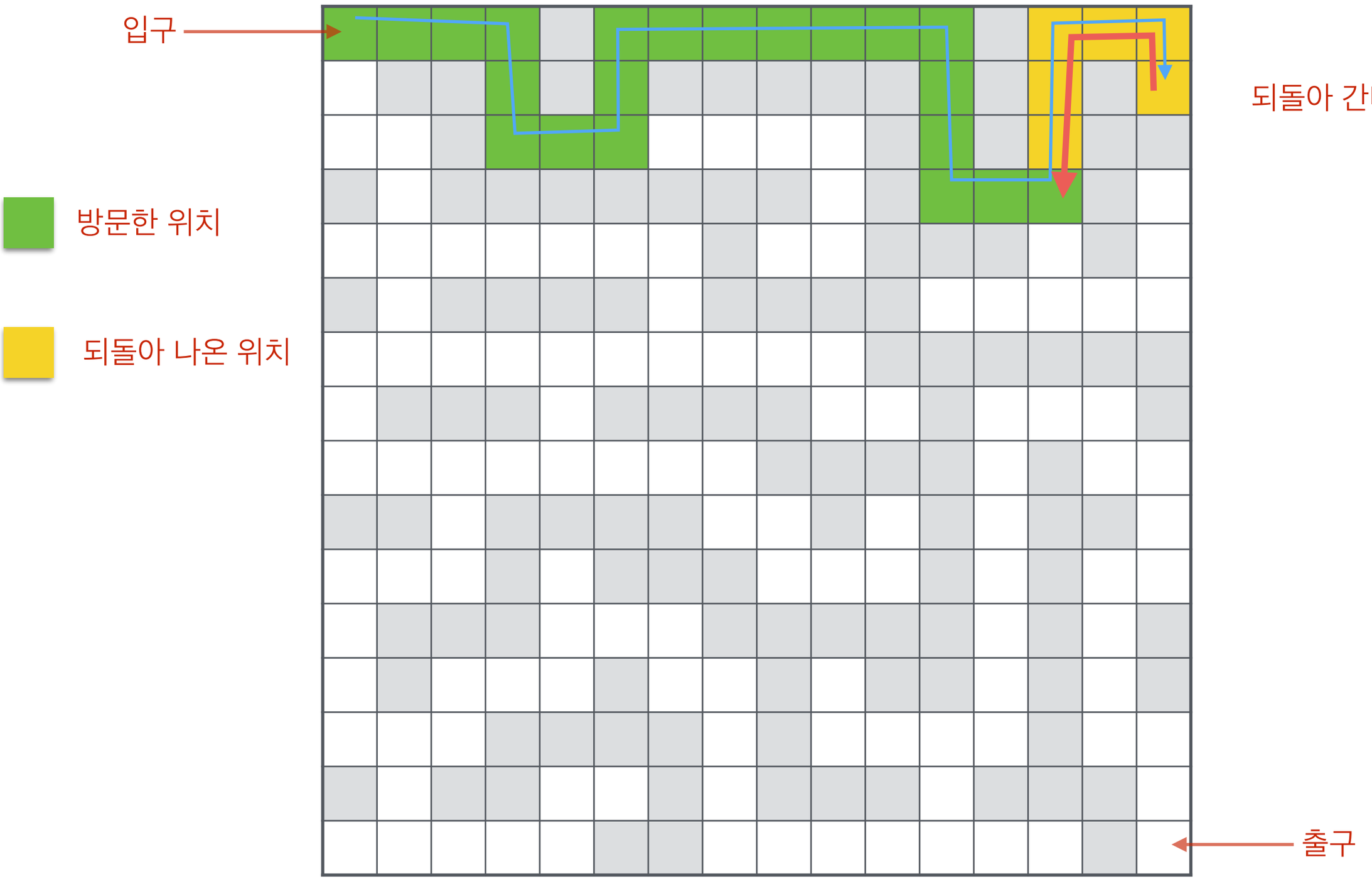
미로찾기

현재 위치에서 북,동,남,서의 순서로
인접한 셀을 검사하여 갈 수 있으면 간다.

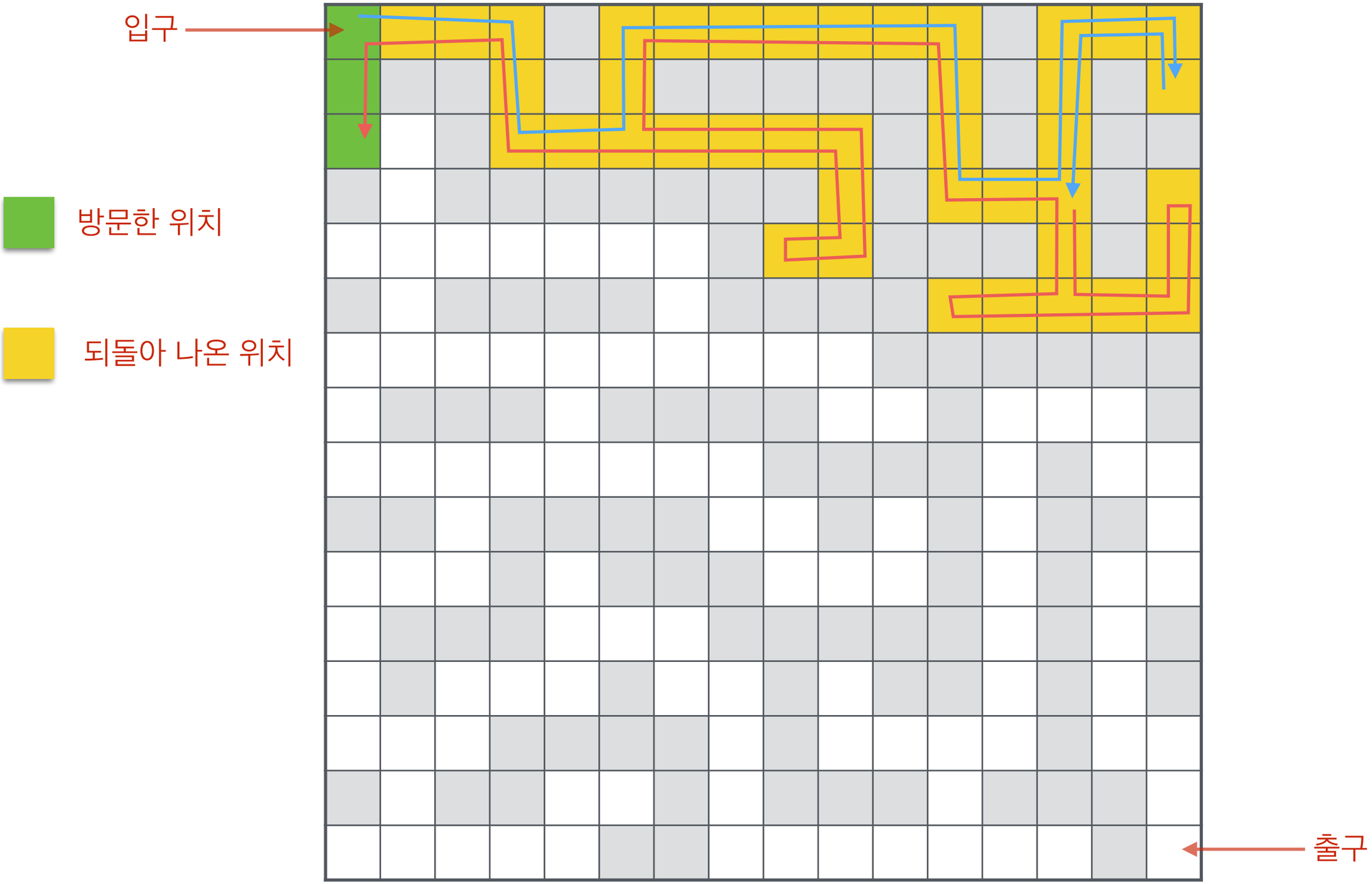


미로찾기

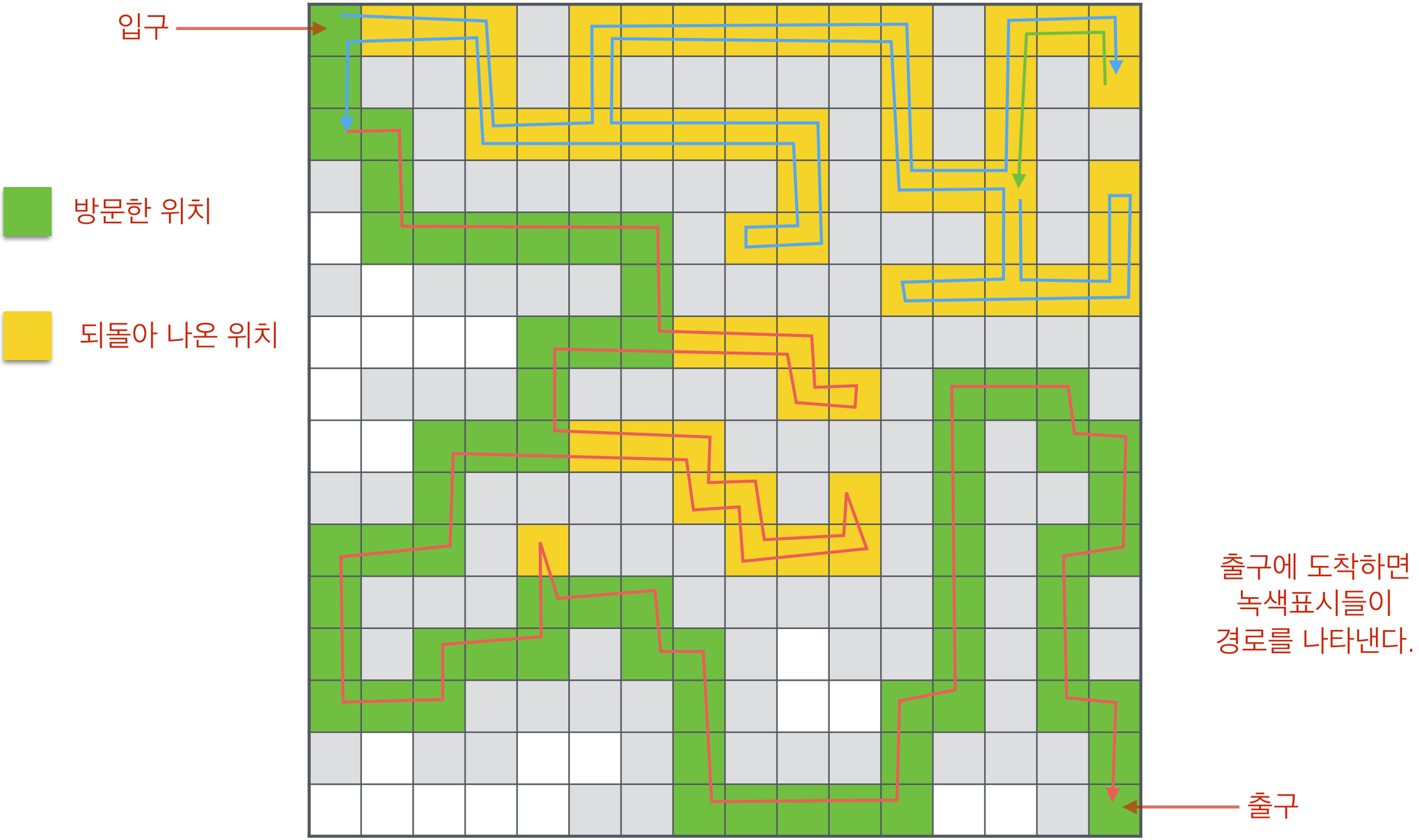
갈 곳이 없으면 되돌아 나간다.



미로찾기



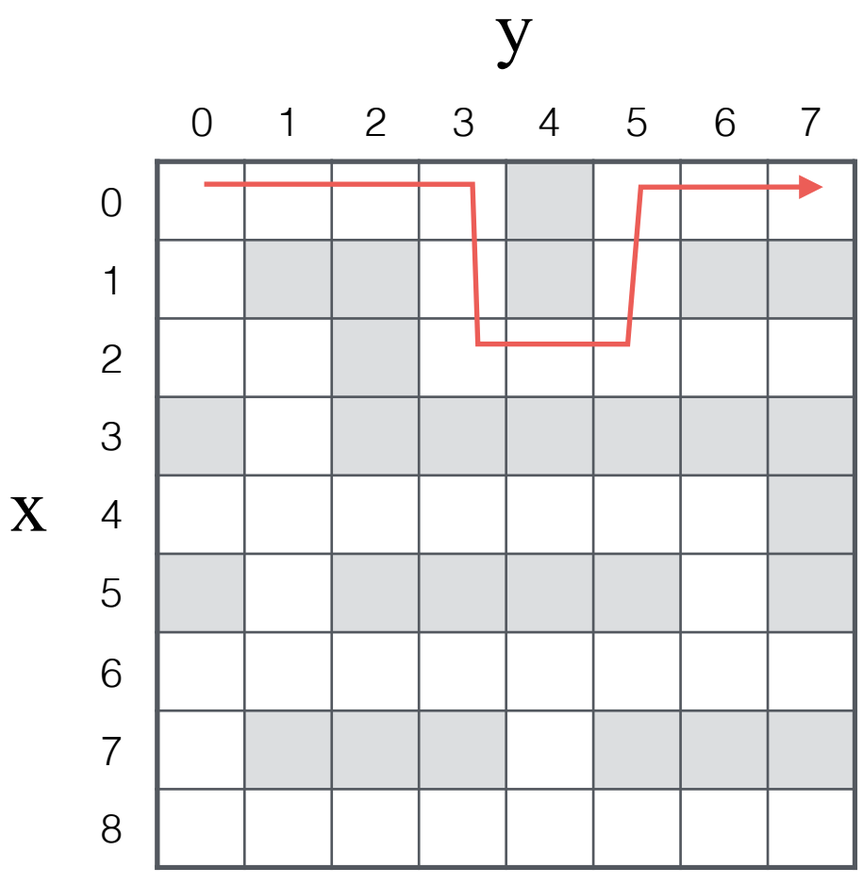
미로찾기



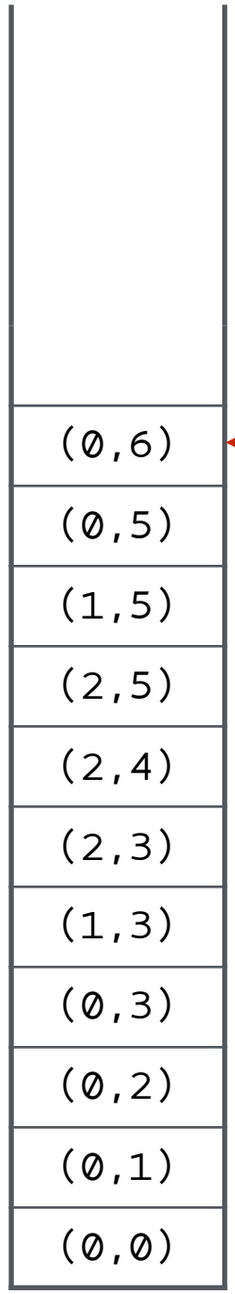
1. **현재위치는 출발점 (0,0)이다.**
2. **다음을 반복한다.**
    1. 현재 위치에 방문했다는 표시를 한다.
    2. 현재 위치가 출구라면 종료한다.
    3. 현재 위치에서 북, 동, 남, 서 4방향에 대해서 순서대로
        1) 그 방향으로 이동할 수 있는지(즉 벽이 아니고, 미로의 외부도 아니고, 이미 방문한 위치도 아닌지) 검사한다.
        2) 만약 갈 수 있으면 그 방향으로 이동한다.
    4. 만약 3번에서 4방향 중 어느 쪽으로도 가지 못했다면 **현재 위치에 도달하기 직전 위치로 돌아간다.** 만약 돌아갈 위치가 없다면 원래 길이 없는 미로이다.

1. **현재위치는 출발점 (0,0)이다.**
2. **다음을 반복한다.**
    1. 현재 위치에 방문했다는 표시를 한다.
    2. 현재 위치가 출구라면 종료한다.
    3. 현재 위치에서 북, 동, 남, 서 4방향에 대해서 순서대로
        1) 그 방향으로 이동할 수 있는지(즉 벽이 아니고, 미로의 외부도 아니고, 이미 방문한 위치도 아닌지) 검사한다.
        2) 만약 갈 수 있으면 **현재 위치를 스택에 push하고** 그 방향으로 이동한다.
    4. 만약 3번에서 4방향 중 어느 쪽으로도 갈 수 없었다면 **스택에서 pop한 후 그 위치로 돌아간다.** 만약 돌아갈 위치가 없다면 원래 길이 없는 미로이다.

스택의 사용

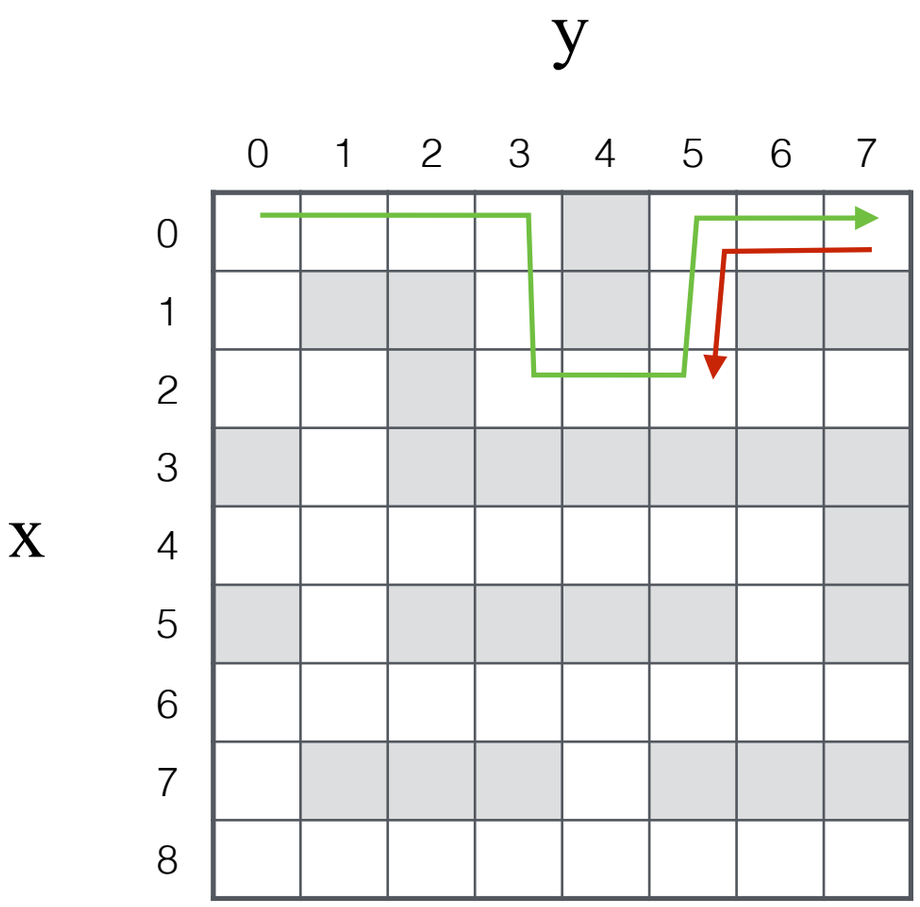


현재 위치를
스택에 push하고
다음 위치로 이동

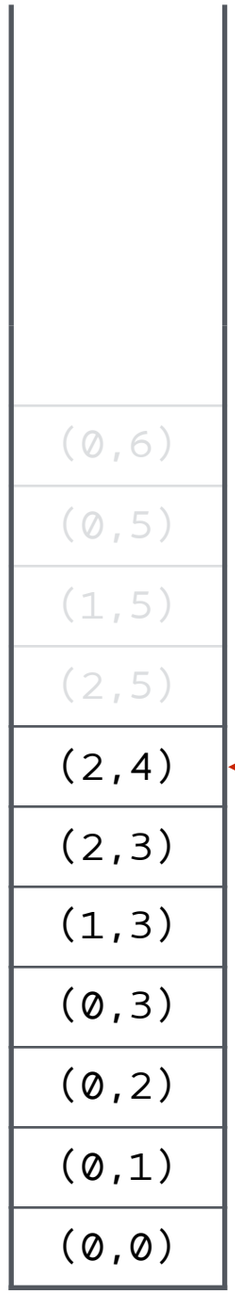


스택의 top에는
항상 직전 위치가
저장되어 있음

스택의 사용



스택에서 pop하여
이전 위치로
되돌아감



스택의 top에는
항상 직전 위치가
저장되어 있음

maze.c

```c
#include <stdio.h>
#include "stack.h"
#include "pos.h"

#define MAX 100
#define PATH 0          /* 지나갈 수 있는 위치 */
#define WALL 1          /* 지나갈 수 없는 위치 */
#define VISITED 2       /* 이미 방문한 위치 */
#define BACKTRACKED 3   /* 방문했다가 되돌아 나온 위치 */

int maze[MAX][MAX];
int n;                  /* 미로의 크기 */

void read_maze();
void print_maze();
bool movable(POS pos, int dir);
```

maze.c

```
int main()
{
    read_maze();                /* maze.txt파일로부터 미로의 모양을 배열 maze로 입력받는다. */

    Stack s = create();         /* 위치를 저장할 스택 */
    Position cur;               /* 항상 현재 위치를 표현 */
    cur.x = 0;
    cur.y = 0;

    while(1) {
       maze[cur.x][cur.y] = VISITED;          /* 현재 위치를 방문했다고 표시한다. */
       if (cur.x == n-1 && cur.y == n-1) {    /* 현재 위치가 출구라면. */
          printf("Found the path.\n");
          break;
       }

       bool forwarded = false;      /* 4방향 중 한 곳으로 전진하는데 성공했는지를 표시한다.*/
       for (int dir = 0; dir<4; dir++) {     /* 0:N, 1:E, 2:S, 3:W */
          if (movable(cur, dir)) {     /* dir 방향으로 이동할 수 있는지 검사 */
             push(s, cur);             /* 현재 위치를 stack에 push하고  */
             cur = move_to(cur, dir); /* dir 방향으로 한 칸 이동한 위치를 새로운 cur라고 한다.*/
             forwarded = true;
             break;
          }
       }
```

```
        if (!forwarded) {     /* 4방향중 어느 곳으로도 가지 못했다면 */
            maze[cur.x][cur.y] = BACKTRACKED;  /* 왔다가 되돌아간 위치임을 표시 */
            if (is_empty(s)) {    /* 되돌아갈 위치가 없다면 원래 길이 없는 미로 */
                printf("No path exists.\n");
                break;
            }
            cur = pop(s);  /* 스택에서 pop한 위치가 새로운 현재위치(cur)가 된다. */
        }
    }

    print_maze();
}
```

pos.h

```c
#ifndef POS_H
#define POS_H

typedef struct pos {
    int x, y;
} Position;

Position move_to(Position pos, int dir);

#endif POS_H
```

pos.c

```c
#include "pos.h"

int offset[4][2] = { {-1, 0},
                     {0, 1},
                     {1, 0},
                     {0, -1} };
```

북, 동, 남, 서 방향으로 한 칸 이동할 때
x-좌표와 y-좌표의 변화량

```c
Position move_to(Position pos, int dir) {
    Position next;
    next.x = pos.x + offset[dir][0];
    next.y = pos.y + offset[dir][1];
    return next;
}
```

알고리즘의 개선

```
int main()
{
    read_maze();

    Stack s = create();    /* 이번에는 Position대신 정수들을 저장하는 스택을 사용한다. */
    Position cur;
    cur.x = 0;
    cur.y = 0;

    int init_dir = 0;  /* 어떤 위치에 도착했을 때 처음으로 시도해 볼 이동 방향 */

    while(1) {
        maze[cur.x][cur.y] = VISITED;  /* visited */
        if (cur.x == n-1 && cur.y == n-1) {
            printf("Found the path.\n");
            break;
        }

        bool forwarded = false;
        for (int dir = init_dir; dir<4; dir++) {        /* 0:N, 1:E, 2:S, 3:W */
            if (movable(cur, dir)) {
                push(s, dir);    /* 스택에는 현재 위치 대신 이동하는 방향을 push */
                cur = move_to(cur, dir);
                forwarded = true;
                init_dir = 0;    /* 처음 방문하는 위치에서는 항상 0번 방향부터 시도해 본다. */
                break;
            }
        }
```

개선

```
if (!forwarded) {     /* nowhere to go forward */
    maze[cur.x][cur.y] = BACKTRACKED;  /* backtracked */
    if (is_empty(s)) {
        printf("No path exists.\n");
        break;
    }
    int d = pop(s);
    cur = move_to(cur, (d+2)%4);

    init_dir = d+1;
    }
}
print_maze();
}
```

이전 위치에서 지금 위치에 올 때 d방향으로 이동했었다면 되돌아 가려면 (d+2)%4번 방향으로 이동하면 된다.

되돌아 간 위치에서는 d+1번 방향부터 시도해보면 된다.