Stack

Data Structure

- **Dynamic Set**

  알고리즘에 의하여 다루어지는 데이터의 집합.
  집합의 크기 및 원소 값이 계속 변함.

  - Elements: data = key data+ satellite data
  - Operations:

    SEARCH(S,k), INSERT(S,x), DELETE(S,x)
    MINIMUM(S), MAXIMUM(S), SUCCESSOR(S,x), PREDECESSOR(S,x)

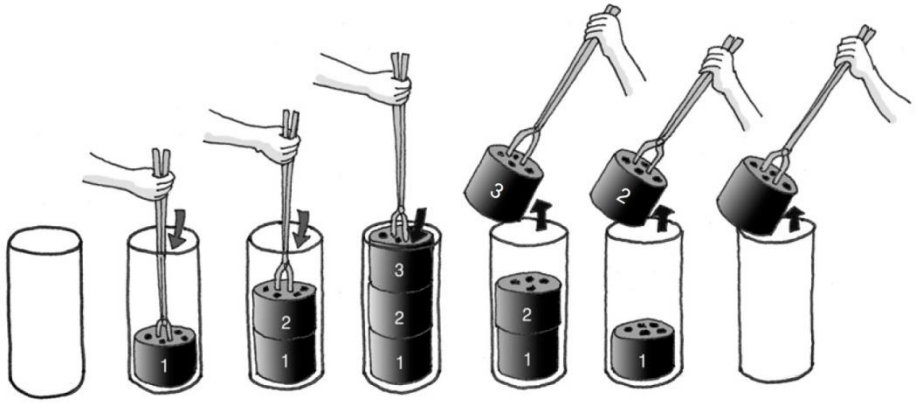
Data Structure

- **Data Structure**
  - Dynamic Set 를 표현하는 방법
  - Stack, Queue, Linked list, Hash table, Heap, Binary search tree, ….

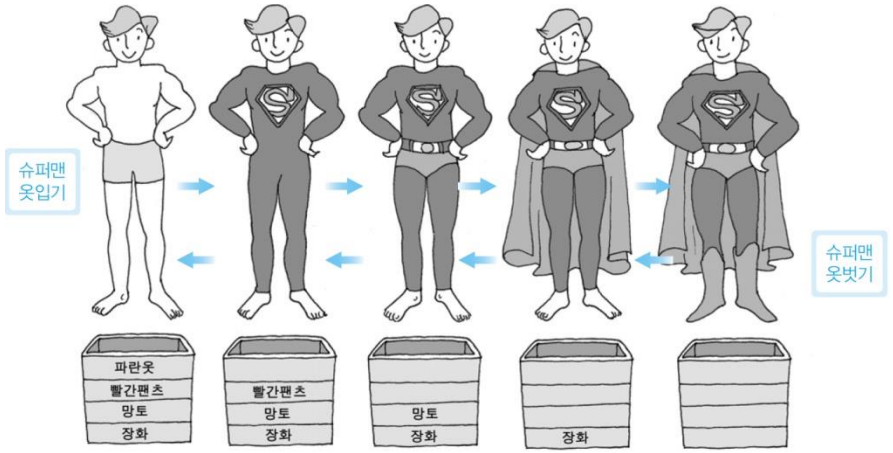
- **Data Structure vs. Class**
  - Data structure = elements + operations
  - Class = data member + member functions

Stack

- Definition
  - LIFO (last-in first-out, 후입선출) 로 element 를 삽입/삭제하는 dynamic set



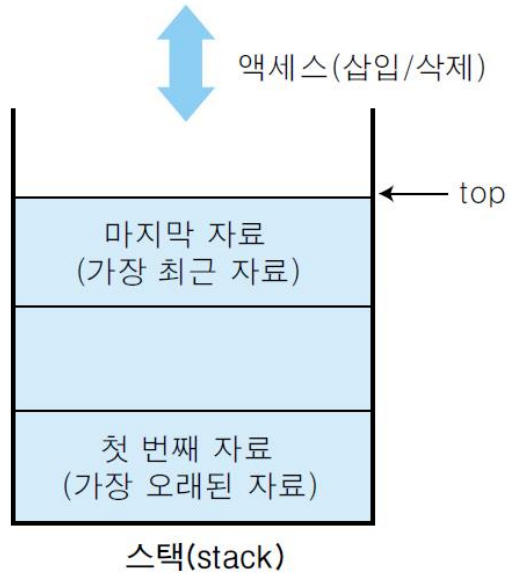
[그림 6-3] 스택 자료구조의 예 : 연탄아궁이



[그림 6-4] 스택 자료구조의 예 : 슈퍼맨 옷 입기

Stack

- Data Member
  - S [1 … n] : array
  - *top*[*S*] : S 에 마지막으로 삽입된 원소의 index



[그림 6-2] 스택의 구조

initial state: *top*[S] 0
stack empty: *top*[S] = 0
stack full: *top*[S] = n

스택에 저장된 원소는
top으로 정한 곳에서만 접근 가능

Stack

- Operations
  - STACK-EMPTY(S)
    : stack 이 비어있는 지 여부 체크
  - STACK-FULL(S)
    : stack 이 꽉차있는 지 여부 체크
  - PUSH(S, x)
    : stack S 에 원소 x 삽입
    : O(1)
  - POP(S)
    : stack S 에서 원소 삭제하고 값 리턴
    : O(1)

✓ Underflow: stack 이 empty 인데 Pop 수행
✓ Overflow: stack 이 full 인데 Push 수행

STACK-EMPTY(S)
```
if top[S] = 0
    then return TRUE
    else  return FALSE
```

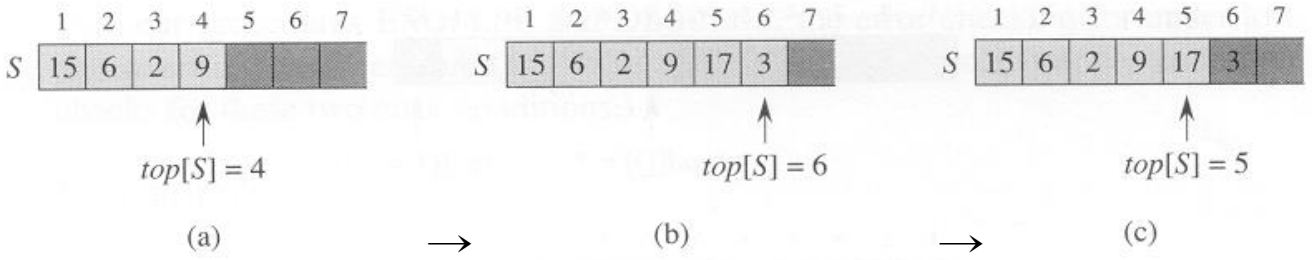
STACK-FULL(S)
```
if top[S] = n
    then return TRUE
    else  return FALSE
```

PUSH(S, x)
```
if  STACK-FULL(S)
     then error "overflow"
     else top[S] ← top[S]+1
          S[top[S]] ← x
```

POP(S)
```
if  STACK-EMPTY(S)
     then error "underflow"
     else top[S] ← top[S]-1
          return S[top[S]+1]
```

Stack

- Operation



PUSH(S,17), PUSH(S,3) POP(S)

(Q) S[1…6], top[S]=0

PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S,8), POP(S)

=> S , top[S]

Stack

- Program in C

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #define STACK_SIZE 100
04
05  typedef int element;       // int를 스택 element의 자료형으로 정의
06  element stack[STACK_SIZE];
07  int top= -1;               // 스택의 top의 초기값을-1로 설정
08
09  void Push(element item)
10  {
11          if(top >= STACK_SIZE-1) {    // 스택이 이미 Full인 경우
12             printf("₩n₩n Stack is FULL ! ₩n");
13             return;
14          }
15          else stack[++top]=item;
16   }
17
18  element Pop()
19  {
20          if(top==-1) {   // 현재 스택이 공백인 경우
21             printf("₩n₩n Stack is Empty!!₩n");
22             return 0;
23          }
24          else return stack[top--];
25  }
```

Stack

- Program in C++

```
template <class KeyType>
class Stack
{
private:
   KeyType *stack;       // KeyType 형의 1차원 array
   int MaxSize;          // stack size
   int top;              // stack 의 top을 가리키는 포인터 (초기치=-1)

public:
   Stack (int MaxStackSize = DefaultSize); // 최대 크기가 MaxStackSize인 공백 스택을 생성
   bool IsFull();
   bool IsEmpty();
   void Push(const KeyType& item);
   KeyType* Pop(KeyType&);};
};
```

Stack

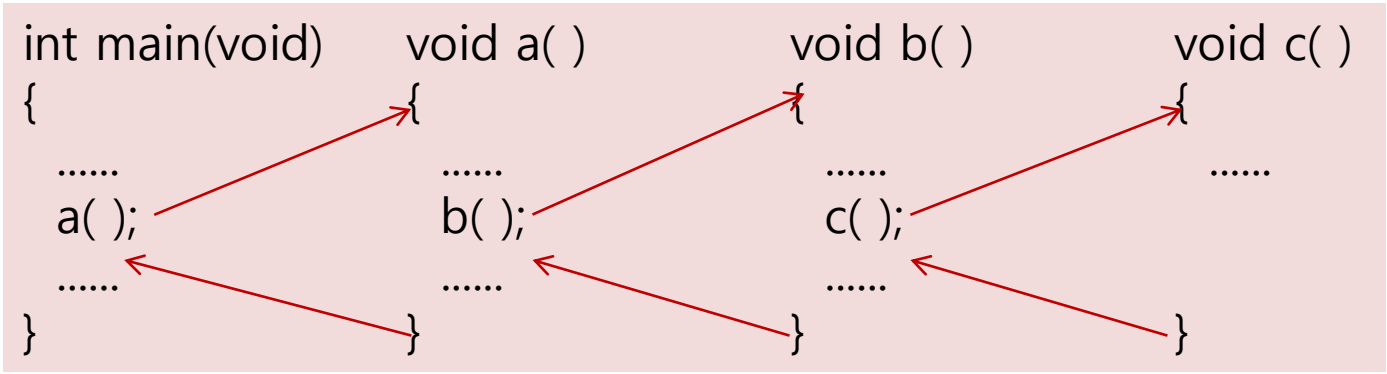
- Program in C++

```
template <class KeyType>
Stack<KeyType>::Stack (int MaxStackSize): MaxSize (MaxStackSize)
{
   stack = new KeyType[MaxSize];
   top = -1;
}

template <class KeyType>
bool Stack<KeyType>::IsFull()
{
   if (top == MaxSize-1) return TRUE;
   else return FALSE;
}

template <class KeyType>
bool Stack<KeyType>::IsEmpty()
{
   if (top == -1) return TRUE;
   else return FALSE;
}
```

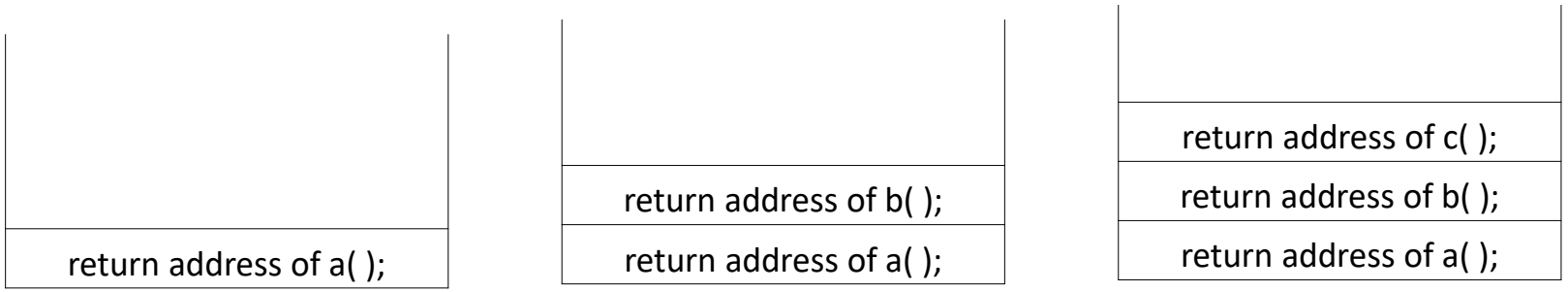
```
template <class KeyType>
void Stack<KeyType>::Push(const KeyType& x)
{
   if (IsFull()) StackFull();
   else stack[++top] = x;
}

template <class KeyType>
KeyType* Stack<KeyType>::Pop(KeyType& x)
{
   if (IsEmpty()) { StackEmpty(); return 0;}
   x = stack[top--];
   return &x;
}
```

System Stack

- System Stack
  - 프로그램 실행 시 함수 호출 및 반환을 처리



(Function Call 시) →



← (Function Return 시)

A Mazing Problem

- Maze
  - *m x p* 의 2차원 배열 (1 : 통로가 막혀 있음, 0 : 통과할 수 있음)

입구

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

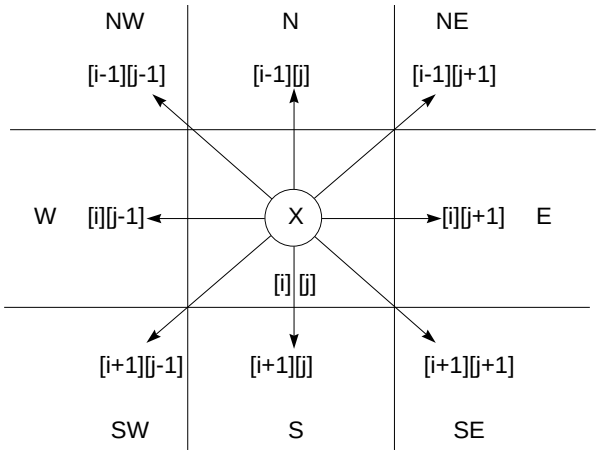
출구

  - Data

    maze[m+2][p+2]; (미로 좌우, 상하의 경계선도 데이터에 포함시킴)

A Mazing Problem

- Allowable moves




```
struct offsets
{
   int a, b;
};
enum directions{N, NE, E, SE, S, SW, W, NW};
offsets move[8];
```

q	move[q].a	move[q].b
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

(Ex) 위치 [i][j]에서 SW 방향으로 이동
⇒ 새 위치 [g][h] ?

```
g = i + move[SW].a;
h = j + move[SW].b;
```

A Mazing Problem

- Stack
  - 경로 탐색 시 분기점의 위치 및 다음 탐색할 방향을 stack 에 Push
  - 막다른 길 도착 시 다음 탐색할 위치 및 방향을 stack 에서 Pop

미로

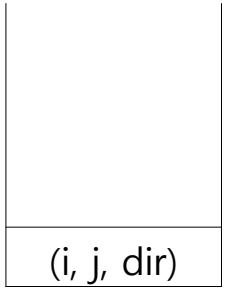
입구	0	0	0	0	0
	1	1	0	1	1
	1	1	0	1	1
	1	1	0	0	0

출구

maze[i][j]

i j	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	1	1	1	1	1	1	1
[1]	1	0	0	0	0	0	1
[2]	1	1	1	0	1	1	1
[3]	1	1	1	0	1	1	1
[4]	1	1	1	0	0	0	1
[5]	1	1	1	1	1	1	1

i j	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	0	0	0	0	0	0	0
[1]	0	0	0	0	0	0	0
[2]	0	0	0	0	0	0	0
[3]	0	0	0	0	0	0	0
[4]	0	0	0	0	0	0	0
[5]	0	0	0	0	0	0	0



stack

mark[i][j] = 1 : (i, j) 를 통과한 경우

A Mazing Problem

- Algorithm

Step1) 현 위치 (i, j)에서 시계 방향으로 (d=N, ..., NW) 인접 위치 탐색
g = i + move[d].a; h= j + move[d].b;
if (maze[g][h]==0) 이고 (mark[g][h]==0) // 이동가능하고 처음 오는 길일 때
{
mark[g][h] = 1;
stack 에 (i, j, d+1) push
i = g; j = h; d = N; // 새로운 위치(g,h) 에 대하여
go to Step1; // 최초방향 (N) 부터 반복
}
else
go to Step 1; // 다음 방향에 대하여

Step 2) Step 1에서 새로운 이동점을 못 찾는 경우 (막다른 길)
stack 에서 (i, j, d+1) pop
go to Step 1

* (g == m) && (h == n) 이 될 때까지 Step 1, 2 반복 진행

A Mazing Problem

- Algorithm

maze

i j	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	1	1	1	1	1	1	1
[1]	1	0	0	0	0	0	1
[2]	1	1	1	0	1	1	1
[3]	1	1	1	0	1	1	1
[4]	1	1	1	0	0	0	1
[5]	1	1	1	1	1	1	1

mark

i j	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	0	0	0	0	0	0	0
[1]	0	0	0	0	0	0	0
[2]	0	0	0	0	0	0	0
[3]	0	0	0	0	0	0	0
[4]	0	0	0	0	0	0	0
[5]	0	0	0	0	0	0	0

- Running time : O(mp)

(i, j)	dir	(g, h)	Stack
(1,1)	E	(1,2)	push (1,1,SE)
(1,2)	E	(1,3)	push (1,2,SE)
(1,3)	E	(1,4)	push (1,3,SE)
(1,4)	E	(1,5)	push (1,4,SE)
(1,5)	?	?	pop (1,4,SE)
(1,4)	SW	(2,3)	push (1,4,W)
(2,3)	S	(3,3)	push (2,3,SW)
		. . . . .	

A Mazing Problem

- Program

```
struct items
{
   int x, y, dir;
};

void path(int m, int p)
{
    mark[1][1]=1;                                  // (1, 1)에서 시작
   stack<items> stack(m*p);
   items temp;
   temp.x = 1; temp.y = 1; temp.dir = E;
   stack.Push(temp);
   while (!stack.IsEmpty())                        // 스택이 공백이 아님
   {
      temp = *stack.Pop(temp);                     // 스택에서 삭제
      while (d < 8) // 앞으로 이동
      {
          int g = i + move[d].a; int h = j + move[d].b;
          if ((g == m) && (h == p)) {              // 출구 도착
          cout << stack;// 경로 출력
          cout << i << " " << j << endl;           // 경로상의 마지막 두 위치
          cout << m << " " << p << endl;
          return;
      }
      if ((!maze[g][h]) && (!mark[g][h])) {        // 새로운 위치
      mark[g][h] = 1;
      temp.x = i; temp.y = j; temp.dir = d+1;
      stack.Push(temp);                            // 스택에 삽입
      i = g; j = h; d = N;                         // (g, h)로 이동
   }
   else d++;                                       // 다음 방향으로 시도
}
cout << "no path in maze" << endl;
```

Infix to Postfix

- 수식의 표기법
  - 전위표기법(prefix notation)
    - 연산자를 피연산자를 앞에 표기하는 방법

    예) +AB

  - 중위표기법(infix notation)
    - 연산자를 피연산자의 가운데 표기하는 방법
    - 사람 연산에 편리

    예) A+B

  - 후위표기법(postfix notation)
    - 연산자를 피연산자 뒤에 표기하는 방법
    - 컴퓨터 연산에 편리

    예) AB+

Infix to Postfix

– 중위표기식의 후위표기식 변환 방법

① 수식의 각 연산자에 대해서 우선순위에 따라 괄호를 사용하여 다시 표현한다.

② **각 연산자를 그에 대응하는 오른쪽괄호의 뒤로 이동시킨다.**

③ 괄호를 제거한다.

- 예) A*B-C/D
- 1단계: ((A*B) - (C/D))
- 2단계:

  ⇒ ((A B)* (C D)/)-
- 3단계: AB*CD/-

Infix to Postfix

- Stack 을 사용한 변환방법

(1) 왼쪽괄호를 만나면 무시하고 다음 문자를 읽는다.

(2) **피연산자**를 만나면 **출력**한다.

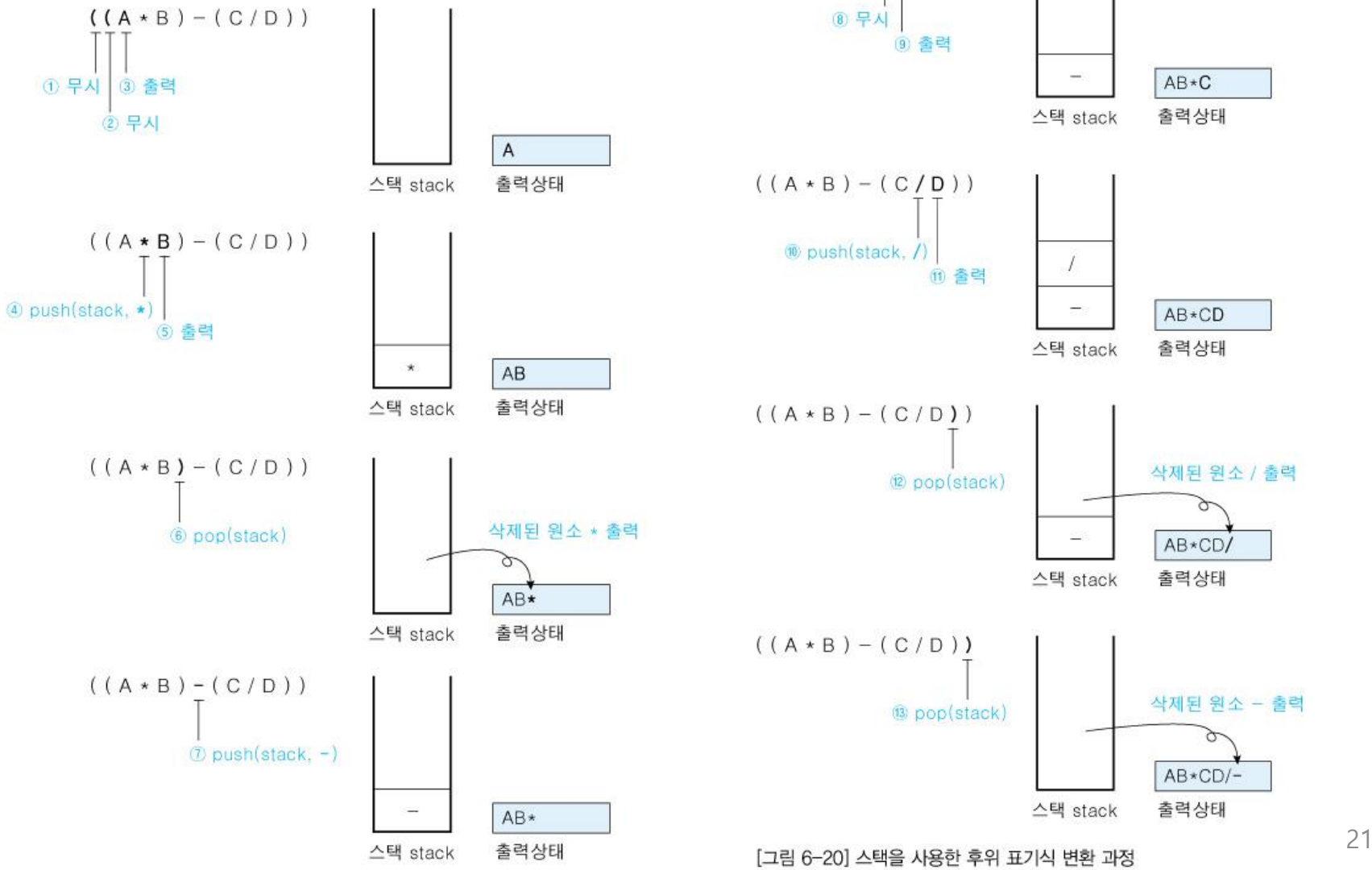
(3) **연산자**를 만나면 스택에 **push**한다.

(4) **오른쪽괄호**를 만나면 스택을 **pop하여 출력**한다.

(5) 수식이 끝나면, 스택이 공백이 될 때까지 **pop하여 출력**한다.

Infix to Prefix

(Ex) ((A*B)-(C/D))



[그림 6-20] 스택을 사용한 후위 표기식 변환 과정

Infix to Prefix

- Algorithm

알고리즘 6-4 후위 표기법 변환 알고리즘

```
infix_to_postfix(exp)
    while(true) do {
        symbol ← getSymbol(exp);
        case {
            symbol = operand :          // 피연산자 처리
                print(symbol);
            symbol = operator :         // 연산자 처리
                push(stack, symbol);
            symbol = ")" :              // 오른쪽 괄호 처리
                print(pop(stack));
            symbol = null :             // 수식의 끝 처리
                while(top > -1) do
                    print(pop(stack));
            else :
        }
    }
end  infix_to_postfix( )
```

Evaluating Postfix Expression

- 연산 방법

> ⑴ **피연산자**를 만나면 스택에 **push** 한다.
>
> ⑵ **연산자**를 만나면 필요한 만큼의 피연산자를 스택에서 **pop**하여 연산하고, **연산결과**를 다시 스택에 **push** 한다.
>
> ⑶ 수식이 끝나면, 마지막으로 스택을 **pop**하여 출력한다.

  - 수식이 끝나고 스택에 마지막으로 남아있는 원소는 전체 수식의 연산결과 값이 된다.

Evaluating Postfix Expression

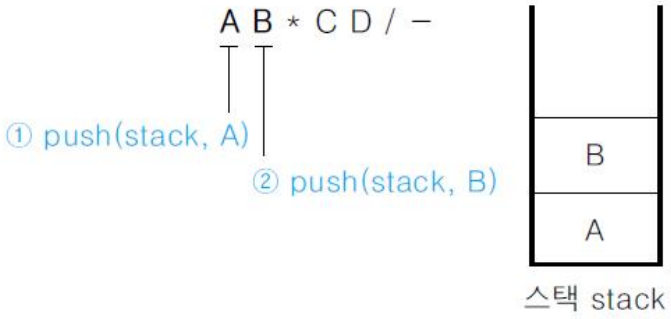
- Algorithm

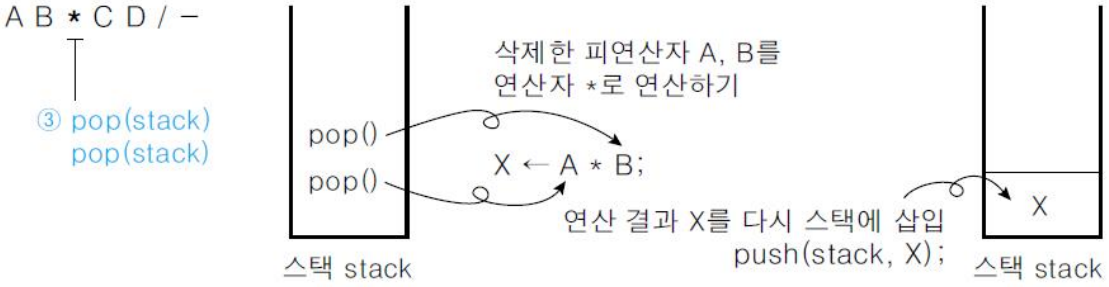
알고리즘 6-5 후위 표기 수식의 연산 알고리즘

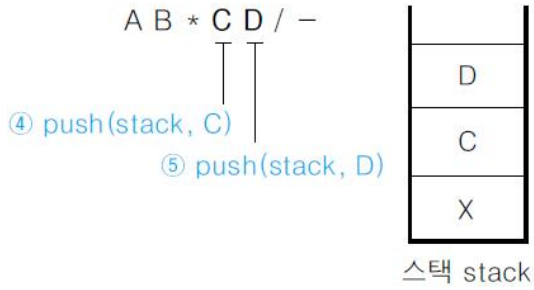
```
evalPostfix(exp)
    while (true) do {
        symbol ← getSymbol(exp);
        case {
            symbol = operand :          // 피연산자 처리
                push(Stack, symbol);
            symbol = operator :         // 연산자 처리
                opr2 ← pop(Stack);
                opr1 ← pop(Stack);
                result ← opr1 op(symbol) opr2;
                // 스택에서 꺼낸 피연산자들을 연산자로 연산
                push(Stack, result);
            symbol = null :             // 후위 수식의 끝
                print(pop(Stack));
        }
    }
end evalPostfix()
```

Evaluating Postfix Expression

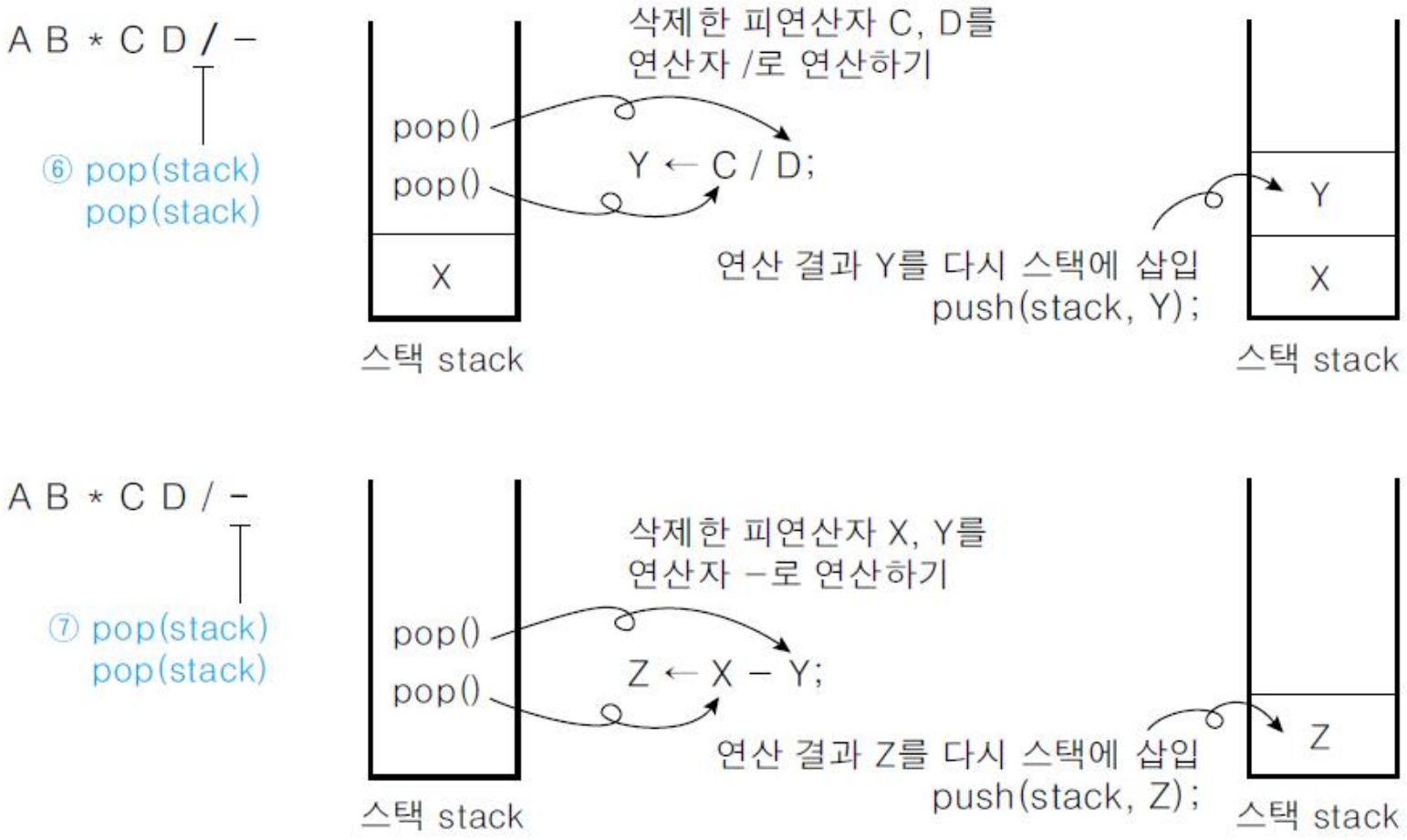
(Ex) AB*CD/-







Evaluating Postfix Expression



[그림 6-22] 스택을 사용한 후위 표기법 연산 과정