

# 자료구조론

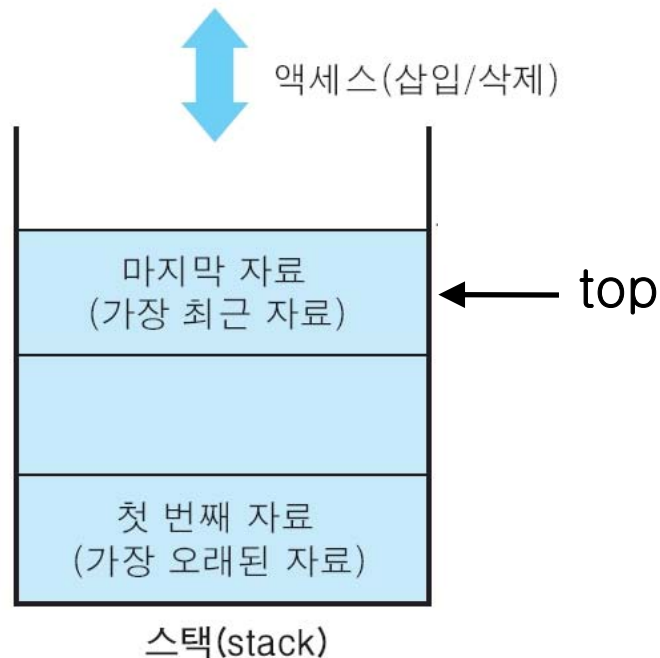
## 7장 스택(stack)

## □ 이 장에서 다룰 내용

- ❖ 스택
- ❖ 스택의 추상 자료형
- ❖ 스택의 구현
- ❖ 스택의 응용

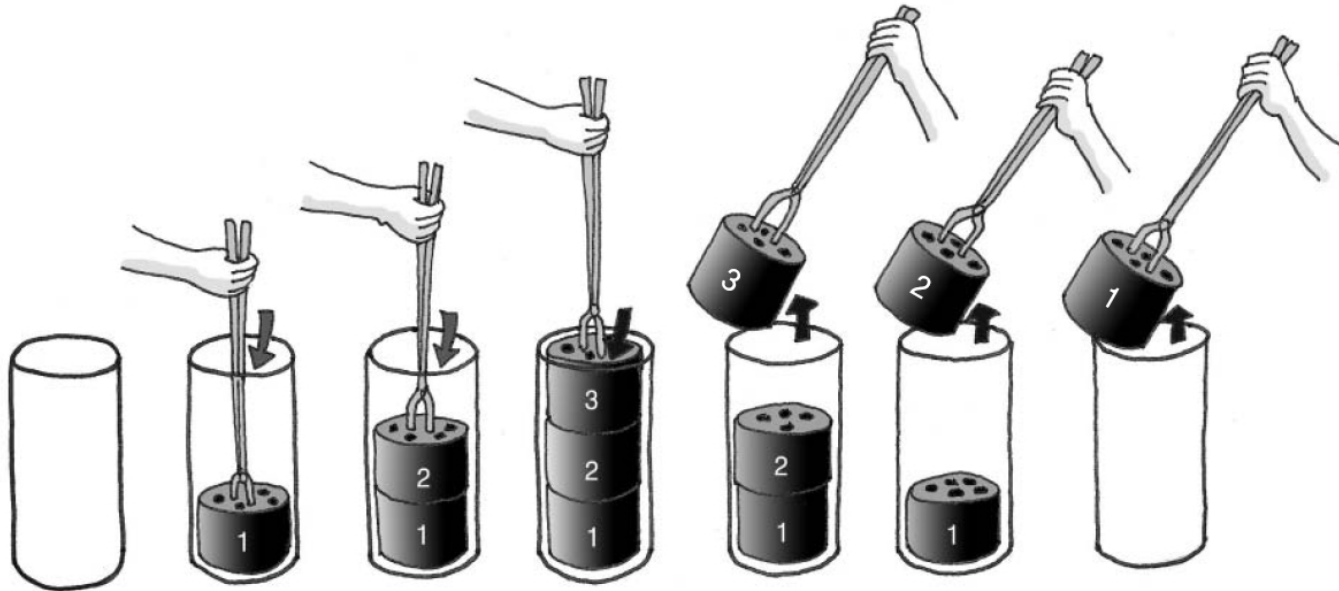
## ❖ 스택(stack)

- 접시를 쌓듯이 자료를 차곡차곡 쌓아 올린 형태의 자료구조
- 스택에 저장된 원소는 top이라고 부르는 한 곳에서만 접근 가능
- 후입선출 구조 (LIFO: Last-In-First-Out)
  - 마지막에 삽입한(Last-In) 원소는 맨 위에 쌓여 있다가 가장 먼저 삭제된다(First-Out).



## ■ LIFO 구조의 예 : 연탄 아궁이

- 연탄을 하나씩 쌓으면서 아궁이에 넣으므로 마지막에 넣은 3번 연탄이 가장 위에 쌓여 있다.
- 연탄을 아궁이에서 꺼낼 때에는 위에서부터 하나씩 꺼내야 하므로 마지막에 넣은 3번 연탄을 가장 먼저 꺼내게 된다.



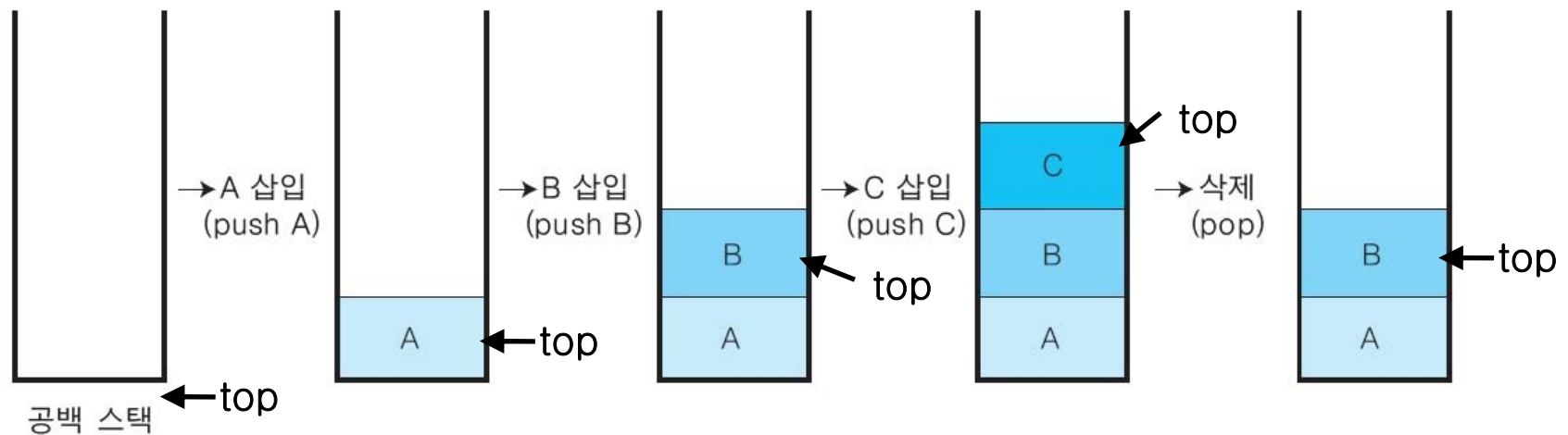
## □ 스택

### ❖ 스택의 연산

- 삽입 연산 : push
- 삭제 연산 : pop

### ❖ 예) 스택의 원소 삽입/삭제

- 공백 스택에 원소 A, B, C를 순서대로 삽입하고 한번 삭제해보자.
  - top은 스택의 가장 위에 놓인 원소를 가리킨다.



## □ 스택의 추상 자료형

### ADT Stack

데이터 : 0개 이상의 원소를 가진 유한 순서 리스트

연산 :  $S \in \text{Stack}; \text{item} \in \text{Element};$

$\text{createStack()} ::= \text{create an empty Stack};$

// 공백 스택을 생성하는 연산

$\text{isEmpty}(S) ::= \text{if } (S \text{ is empty}) \text{ then return true else return false};$

// 스택 S가 공백인지 아닌지를 확인하는 연산

$\text{push}(S, \text{item}) ::= \text{insert item onto the top of } S;$

// 스택 S의 top에 item(원소)을 삽입하는 연산

$\text{pop}(S) ::= \text{if } (\text{isEmpty}(S)) \text{ then return error}$

**else** { delete and return the top item of S };

// 스택 S의 top에 있는 item(원소)을 삭제하여 리턴하는 연산

$\text{peek}(S) ::= \text{if } (\text{isEmpty}(S)) \text{ then return error}$

**else** return (the top item of the S);

// 스택 S의 top에 있는 item을 삭제하지않고 리턴하는 연산

End Stack

## □ 스택의 구현 - 순차 자료구조

### ❖ 스택의 구현

- 순차 자료구조
- 연결 자료구조

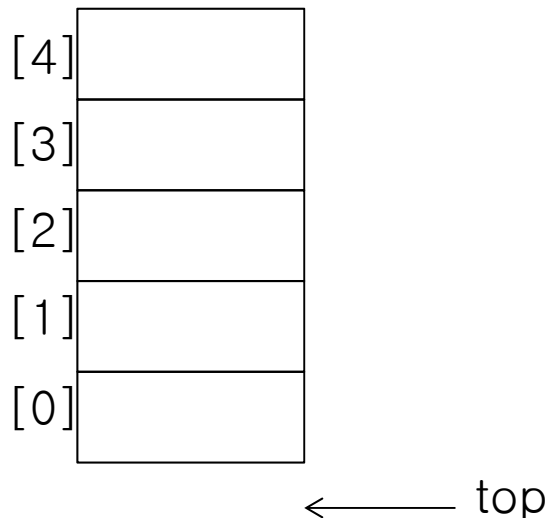
### ❖ 순차 자료구조를 이용한 스택의 구현

- 변수 **top** : 스택의 가장 위에 놓인 원소의 인덱스를 저장하는 변수

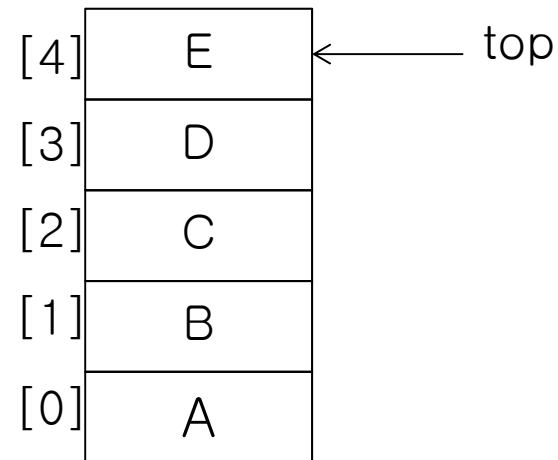
예) 스택의 크기  $n$ 이 5인 경우

empty 상태 :  $\text{top} == -1$

(초기 상태)



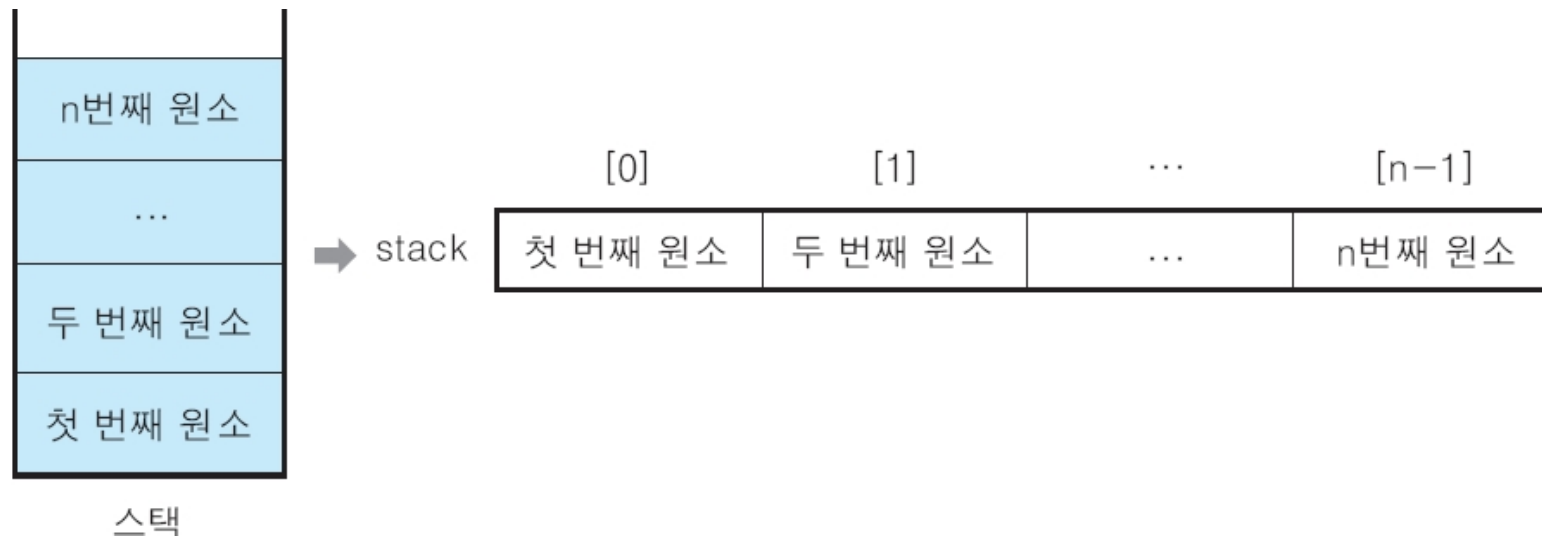
full 상태 :  $\text{top} == n-1$



## □ 스택의 구현 - 순차 자료구조

### ❖ 순차 자료구조를 이용한 스택의 구현

- 1차원 배열을 이용하여 스택을 구현할 수 있다.
- 스택 크기 = 배열 크기 → 배열 원소 수 만큼의 자료 저장 가능
- 스택에 저장된 원소의 순서 = 배열 원소의 인덱스
  - 스택의 첫번째 원소 = 인덱스 0
  - 스택의 n번째 원소 = 인덱스 n-1

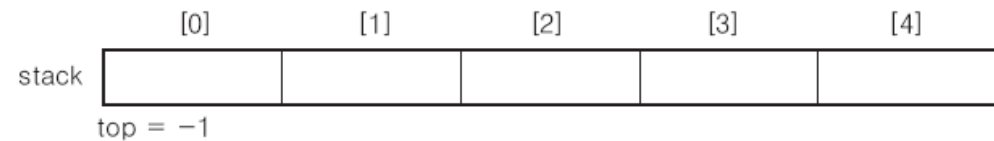




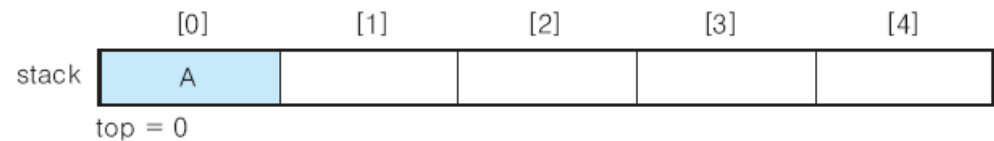
## □ 스택의 구현 - 순차 자료구조

- 크기가 5인 1차원 배열의 스택에서 [그림 7-6]의 연산 수행과정

① 공백 스택 생성 : `create(stack, 5);`



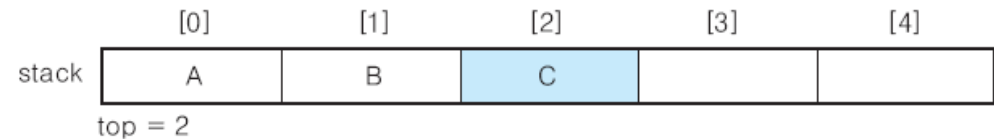
② 원소 A 삽입 : `push(stack, A);`



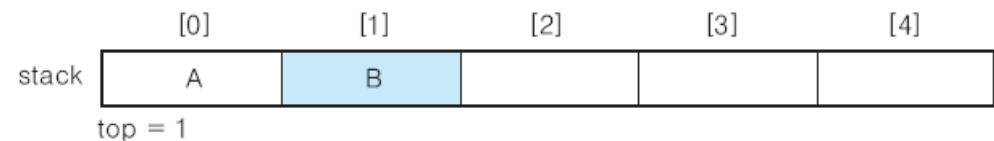
③ 원소 B 삽입 : `push(stack, B);`



④ 원소 C 삽입 : `push(stack, C);`



⑤ 원소 삭제 : `pop(stack);`



## □ 스택의 구현 - 순차 자료구조

### ❖ 스택의 삽입 연산 : push 알고리즘

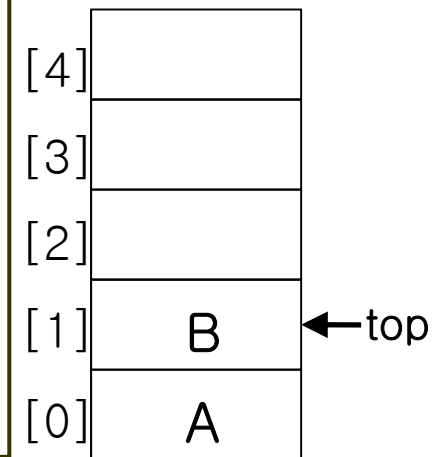
#### ① 스택이 가득 찬 경우가 아니면

- top이 가장 위의 자료를 가리키고 있으므로 그 위에 자료를 삽입하려면 먼저 top의 위치를 하나 증가
- $top \leftarrow top+1;$

#### ② 스택의 top이 가리키는 새로운 위치에 x 삽입

- $S[top] \leftarrow x;$

```
push(S, x)
  if (top = stack_SIZE-1) then overflow;
  else {
    top ← top+1;
    S[top] ← x;
  }
end push( )
```

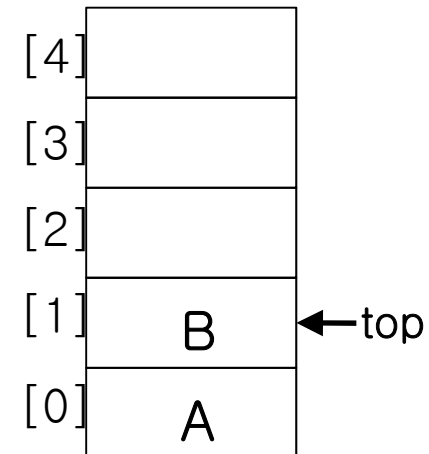


## □ 스택의 구현 - 순차 자료구조

### ❖ 스택의 삭제 연산 : **pop** 알고리즘

- ① 스택이 공백 스택이 아니라면 top이 가리키는 원소를 반환
- ② 스택의 top 원소를 삭제하였으므로 top의 위치를 그 아래의 원소로 변경해야 함 → top의 위치를 하나 감소

```
pop(S)
  if (top = -1) then error;
  else {
    temp ← S[top];
    top ← top-1;
    return temp;
  }
end pop()
```



## □ 스택의 구현 - 순차 자료구조

- 배열로 구현한 문자 스택

```
public class ArrayStack {  
    private int top;  
    private int stackSize;  
    private char[] itemArray;  
  
    public ArrayStack(int stackSize){  
        top = -1;  
        this.stackSize = stackSize;  
        itemArray = new char[stackSize];  
    }  
  
    public boolean isEmpty(){  
        return (top == -1);  
    }  
  
    private boolean isFull(){  
        return (top == stackSize-1);  
    }  
}
```

// 다음 슬라이드에 계속

## □ 스택의 구현 - 순차 자료구조

```
public void push(char item) {
    if(isFull())
        System.out.println("Inserting fail! Array Stack is full!!");
        // ... 배열을 확장하든지, 에러를 발생시키든지 적절한 처리 필요
    else
        itemArray[++top] = item;
}

public char pop() {
    if(isEmpty()) {
        System.out.println("Deleting fail! Array Stack is empty!!");
        예외 발생;
    }
    else
        return itemArray[top--];
}
// 다음 슬라이드에 계속
```

## □ 스택의 구현 - 순차 자료구조

```
public char peek(){  
    ...  
    스택이 비어 있으면 예외 발생;  
}  
  
public String toString(){  
    String str = "Stack>> ";  
    for(int i=0; i<=top; i++)  
        str += itemArray[i] + " ";  
    return str;  
}  
}
```

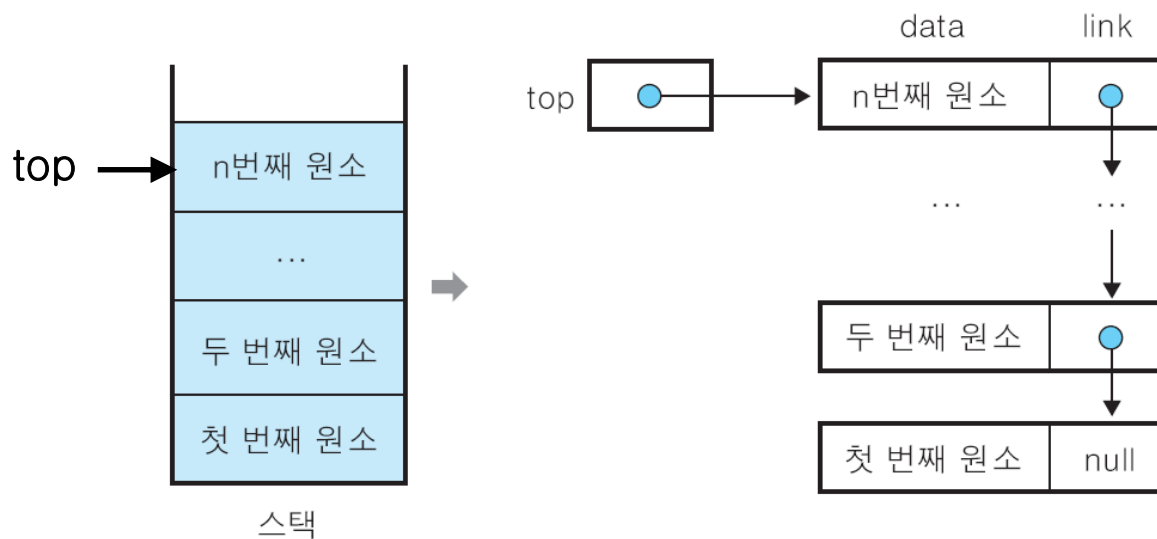
## □ 스택의 구현 - 순차 자료구조

- 순차 자료구조로 구현한 스택의 장점
    - 순차 자료구조인 1차원 배열을 사용하여 쉽게 구현
  - 순차 자료구조로 구현한 스택의 단점
    - 크기가 고정된 배열을 사용하므로 비어있는 공간으로 기억장소가 낭비될 수 있으며, 스택의 크기 변경이 비효율적임
    - 즉, 순차 자료구조의 단점을 그대로 가지고 있다.
- ➔ 연결 자료구조를 이용하여 스택을 구현하면 이러한 단점들을 해결할 수 있다.

## □ 스택의 구현 - 연결 자료구조

### ❖ 연결 자료구조를 이용한 스택의 구현

- 단순 연결 리스트를 이용하여 구현할 수 있다.
- 스택의 원소 하나가 단순 연결 리스트의 노드 하나로 표현된다.
  - push : 리스트의 가장 앞에 노드 삽입
  - pop : 리스트의 가장 앞 노드 삭제
- 변수 top : 단순 연결 리스트의 가장 앞 노드를 가리키는 변수
  - empty 상태 : top == null (초기 상태)





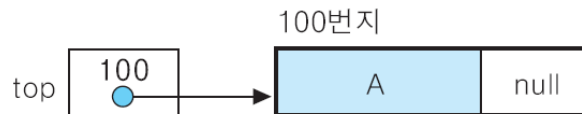
## □ 스택의 구현 - 연결 자료구조

- 단순 연결 리스트 스택에서  
[그림 7-6]의 연산 수행과정

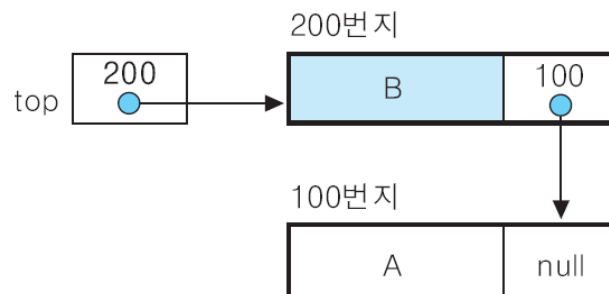
① 공백 스택 생성 : `create(stack);`



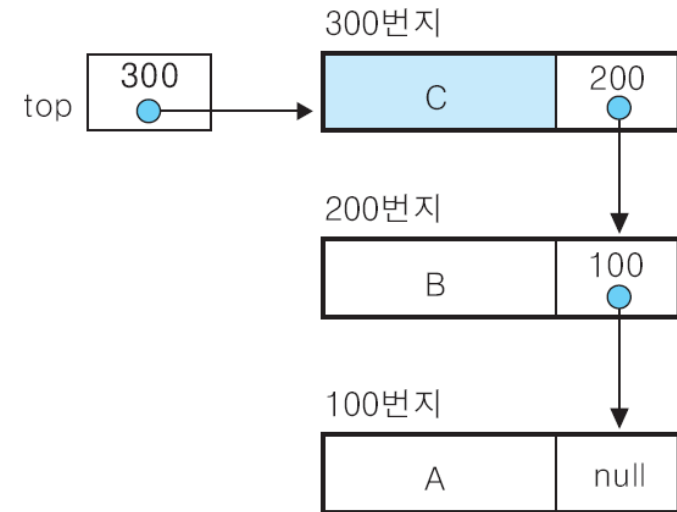
② 원소 A 삽입 : `push(stack, A);`



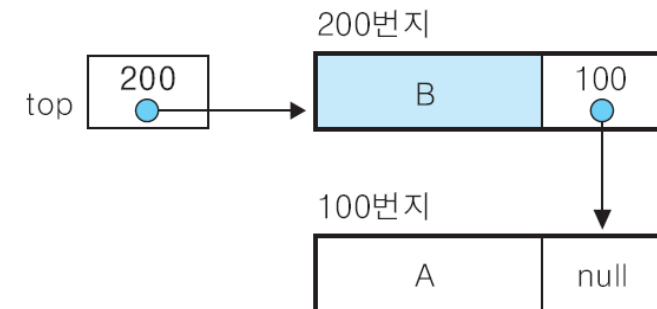
③ 원소 B 삽입 : `push(stack, B);`



④ 원소 C 삽입 : `push(stack, C);`



⑤ 원소 삭제 : `pop(stack);`



## □ 스택의 구현 - 연결 자료구조

- 연결 자료구조 방식을 이용하여 구현한 문자 스택

```
public class LinkedStack {  
    private Node top = null;  
  
    public boolean isEmpty(){  
        return (top == null);  
    }  
  
    public void push(char item){  
        Node newNode = new Node();  
        newNode.data = item;  
        newNode.link = top;  
        top = newNode;  
    }  
    // 다음 슬라이드에 계속
```

## □ 스택의 구현 - 연결 자료구조

```
public char pop() {  
    if(isEmpty()) {  
        System.out.println("Deleting fail! Linked Stack is empty!!");  
        예외 발생;  
    }  
    else {  
        char item = top.data;  
        top = top.link;  
        return item;  
    }  
}  
  
public char peek() {  
    ...  
    스택이 비어 있으면 예외 발생;  
}  
// 다음 슬라이드에 계속
```

## □ 스택의 구현 – 연결 자료구조

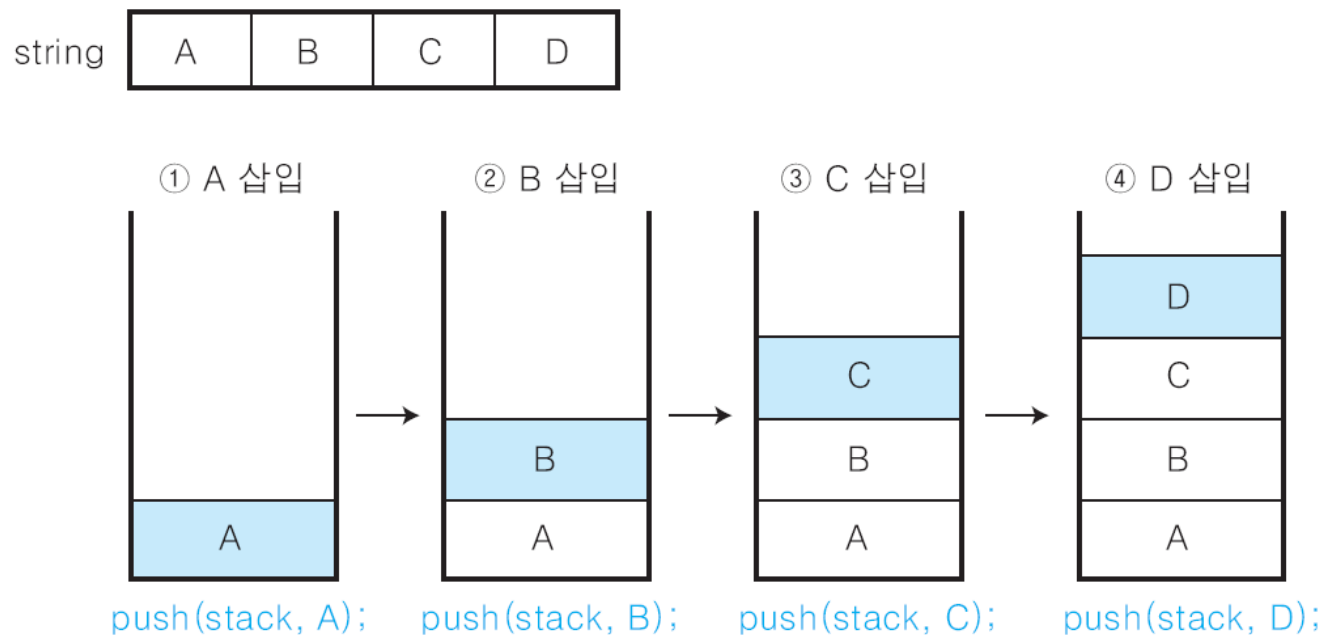
```
public String toString(){  
    String str = "Stack>> ";  
    ...  
    return str;  
}  
  
private class Node{  
    char data;  
    Node link;  
}  
}
```

## □ 스택의 응용 - (1) 역순 문자열 만들기

### ❖ 역순 문자열 만들기

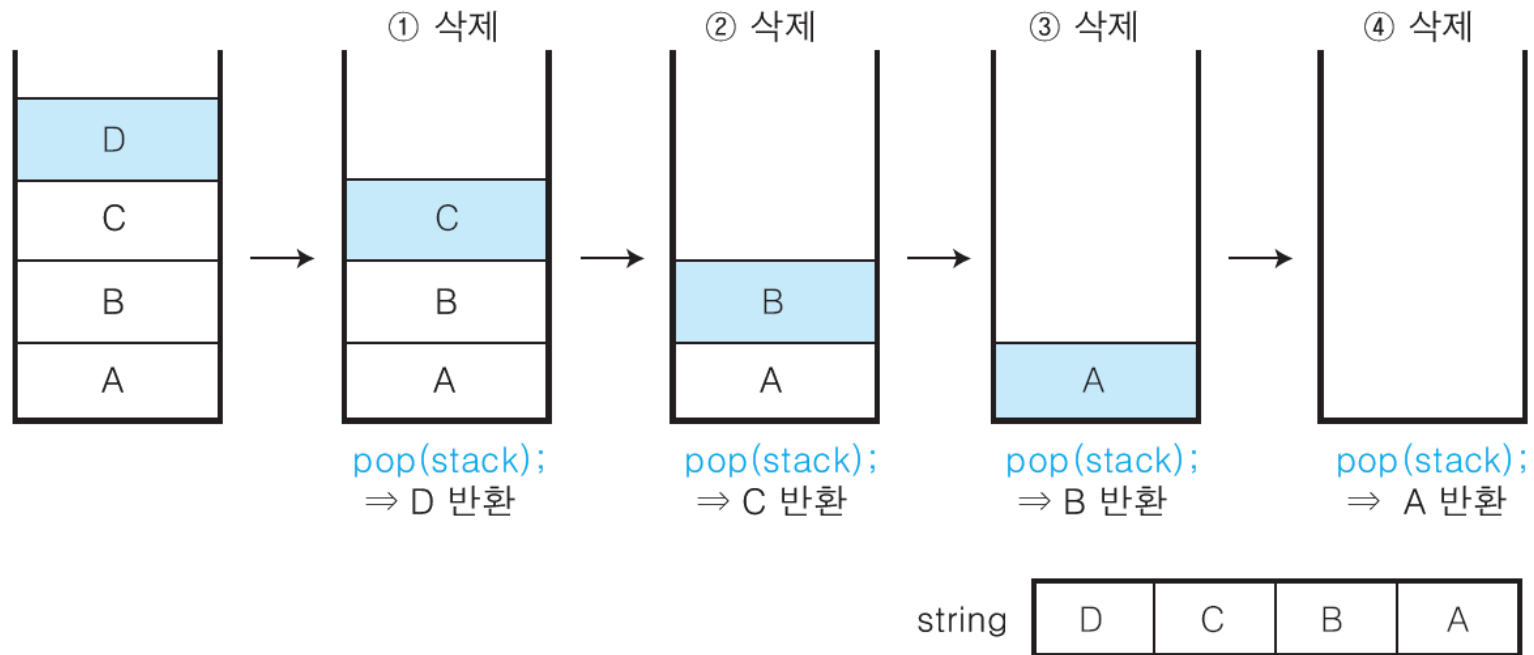
- 스택의 후입선출(LIFO) 성질을 이용

① 문자열의 문자들을 순서대로 스택에 push 하기



## □ 스택의 응용 - (1) 역순 문자열 만들기

② 스택을 pop하여 나오는 순서대로 문자들을 문자열에 저장하기

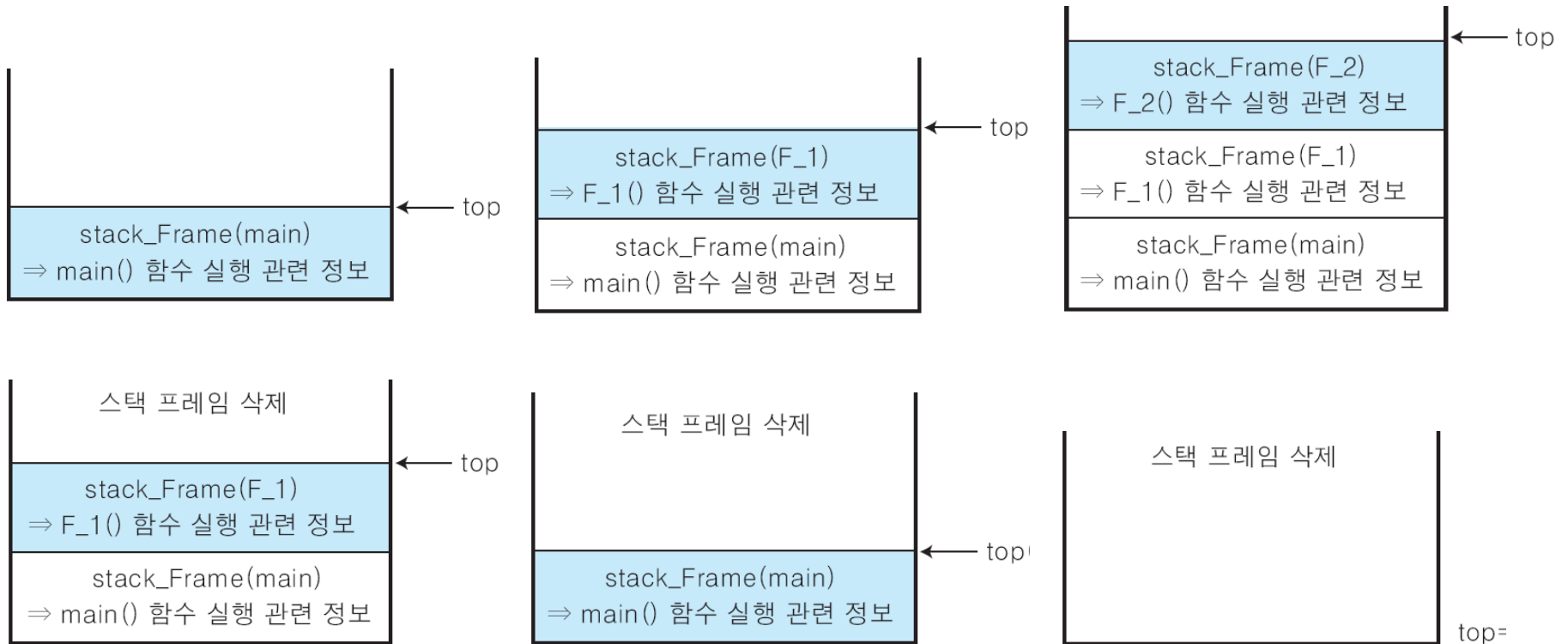
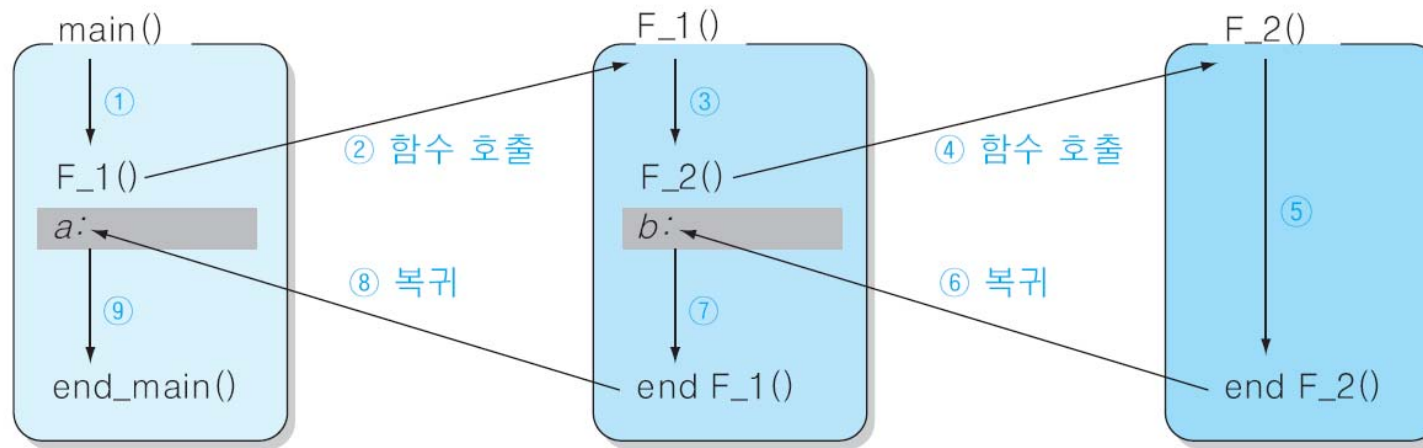


## □ 스택의 응용 - (2) 시스템 스택

### ❖ 시스템 스택

- 프로그램에서의 호출과 복귀에 따른 수행 순서를 관리
  - 가장 마지막에 호출된 함수가 가장 먼저 실행을 완료하고 복귀하는 후입선출 구조이므로, 후입선출 구조의 스택을 이용하여 수행 순서 관리
  - 함수 호출이 발생하면 호출한 함수 수행에 필요한 지역변수, 매개변수 및 수행 후 복귀할 주소 등의 정보를 스택 프레임(stack frame)에 저장하여 시스템 스택에 삽입
  - 함수의 실행이 끝나면 시스템 스택의 top 원소(스택 프레임)를 삭제(pop)하면서 프레임에 저장되어있던 복귀주소를 확인하고 복귀
  - 함수 호출과 복귀에 따라 이 과정을 반복하여 전체 프로그램 수행이 종료되면 시스템 스택은 공백이 됨

## □ 스택의 응용 - (2) 시스템 스택





## □ 스택의 응용 - [3] 수식의 괄호 검사

### ❖ 수식의 괄호 검사

- 수식에 포함되어있는 괄호는 가장 마지막에 열린 괄호를 가장 먼저 닫아 주어야 하는 후입선출 구조로 구성

예)  $[ \{ a * (b + c) \} ]$

➔ 후입선출 구조의 스택을 이용하여 괄호를 검사

- 괄호 검사 방법

- 수식을 왼쪽에서 오른쪽으로 하나씩 읽으면서 검사
- 왼쪽 괄호를 읽으면 스택에 push
- 오른쪽 괄호를 읽으면 스택을 pop하여 방금 읽은 오른쪽 괄호와 같은 종류인지를 확인

➤ 같은 종류의 괄호가 아니면 잘못된 수식임

예)  $[ ( ) )$

➤ 스택이 비어있으면 잘못된 수식임

예)  $( ) )$

- 수식에 대한 검사가 모두 끝났을 때 스택은 공백

➤ 스택이 공백이 아니면 잘못된 수식임

예)  $( ( ) )$

## □ 스택의 응용 - [3] 수식의 괄호 검사

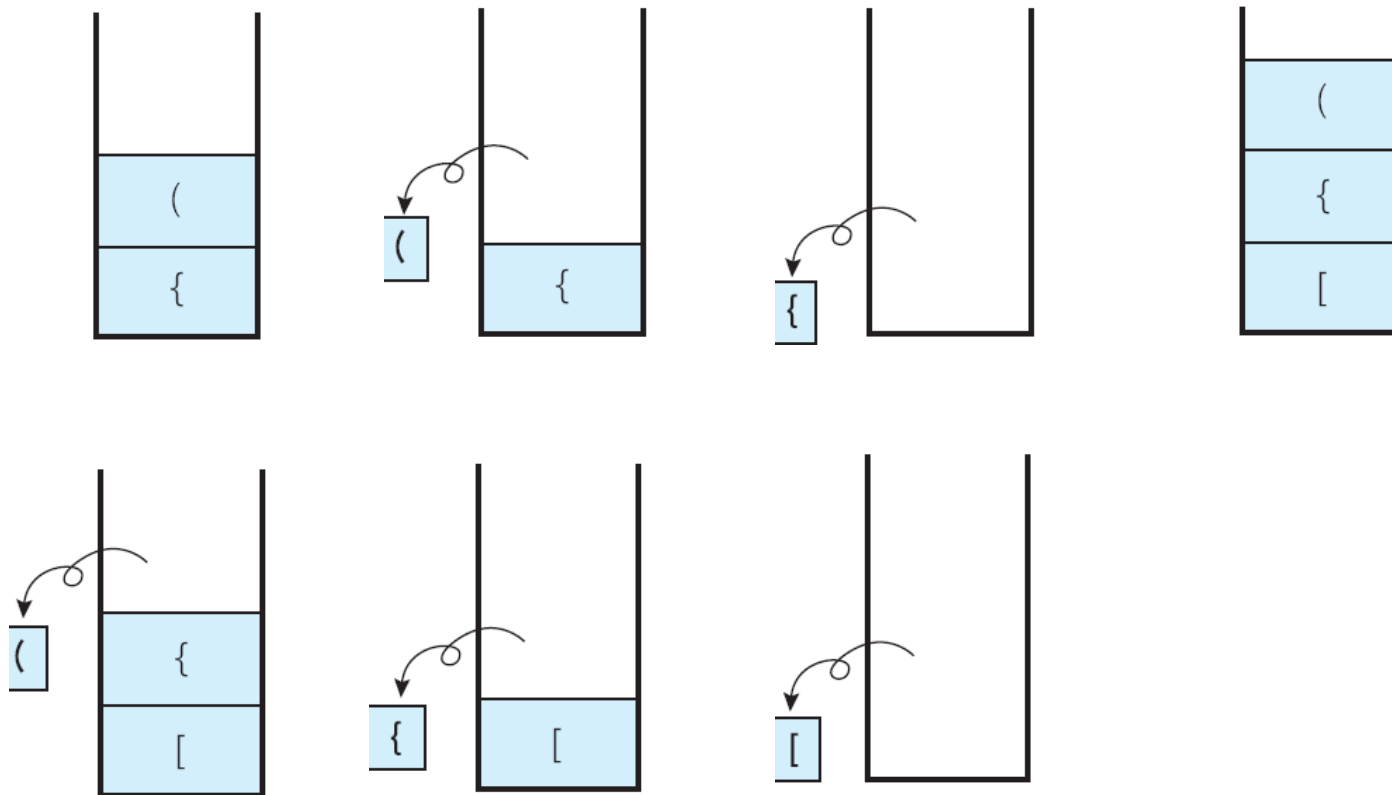
### ■ 수식의 괄호 검사 알고리즘

```
testPair( )
  exp ← Expression;      Stack ← null;
  while (true) do {
    symbol ← getSymbol(exp);
    case {
      symbol = "(" or "[" or "{" :
        push(Stack, symbol);
      symbol = ")" :
        open_pair ← pop(Stack);
        if (open_pair ≠ "(") then return false;
      symbol = "]" :
        open_pair ← pop(Stack);
        if (open_pair ≠ "[") then return false;
      symbol = "}" :
        open_pair ← pop(Stack);
        if (open_pair ≠ "{") then return false;
      symbol = null :
        if (isEmpty(Stack)) then return true;
        else return false;
    }
  }
end testPair( )
```

## □ 스택의 응용 - [3] 수식의 괄호 검사

- ## ■ 수식의 괄호 검사 예

$$\{ (A+B) - 3 \} * 5 + [ \{ \cos(x+y) + 7 \} - 1 ] * 4$$



>> 계속

## □ 스택의 응용 - [4] 수식의 후위표기법 변환

### ❖ 수식의 표기법

#### ■ 중위표기법(infix notation)

- 연산자를 피연산자의 가운데 표기하는 방법
- 연산자 우선순위를 표현하기 위해 괄호를 사용
  - 예)  $A+B$                        $A*(B+C)$

#### ■ 후위표기법(postfix notation)

- 연산자를 피연산자 뒤에 표기하는 방법
- 괄호가 필요 없음
  - 예)  $AB+$                        $ABC+*$

#### ■ 전위표기법(prefix notation)

- 연산자를 앞에 표기하고 그 뒤에 피연산자를 표기하는 방법
- 괄호가 필요 없음
  - 예)  $+AB$                        $*+ABC$

## □ 스택의 응용 - [4] 수식의 후위표기법 변환

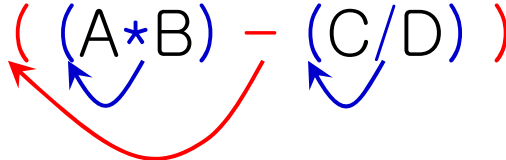
### ■ 중위표기식 to 전위표기식 변환 방법

- ① 연산자 우선순위에 따라 괄호를 사용하여 수식을 다시 표현한다.
- ② 각 연산자를 그에 대응하는 왼쪽 괄호의 앞으로 이동한다.
- ③ 괄호를 제거한다.

• 예)  $A * B - C / D$

1단계:  $( (A * B) - (C / D) )$

2단계:  $( (A * B) - (C / D) )$



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

3단계:  $- * A B / C D$

## □ 스택의 응용 - [4] 수식의 후위표기법 변환


### ■ 중위표기식 to 후위표기식 변환 방법

- ① 연산자 우선순위에 따라 괄호를 사용하여 수식을 다시 표현한다.
- ② 각 연산자를 그에 대응하는 오른쪽 괄호의 앞으로 이동한다.
- ③ 괄호를 제거한다.

• 예)  $A * B - C / D$

1단계:  $( (A * B) - (C / D) )$

2단계:  $( (A * B) - (C / D) )$



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

3단계:  $A B * C D / -$

## □ 스택의 응용 - [4] 수식의 후위표기법 변환

### ❖ 스택을 사용한 중위표기식 to 후위표기식 변환

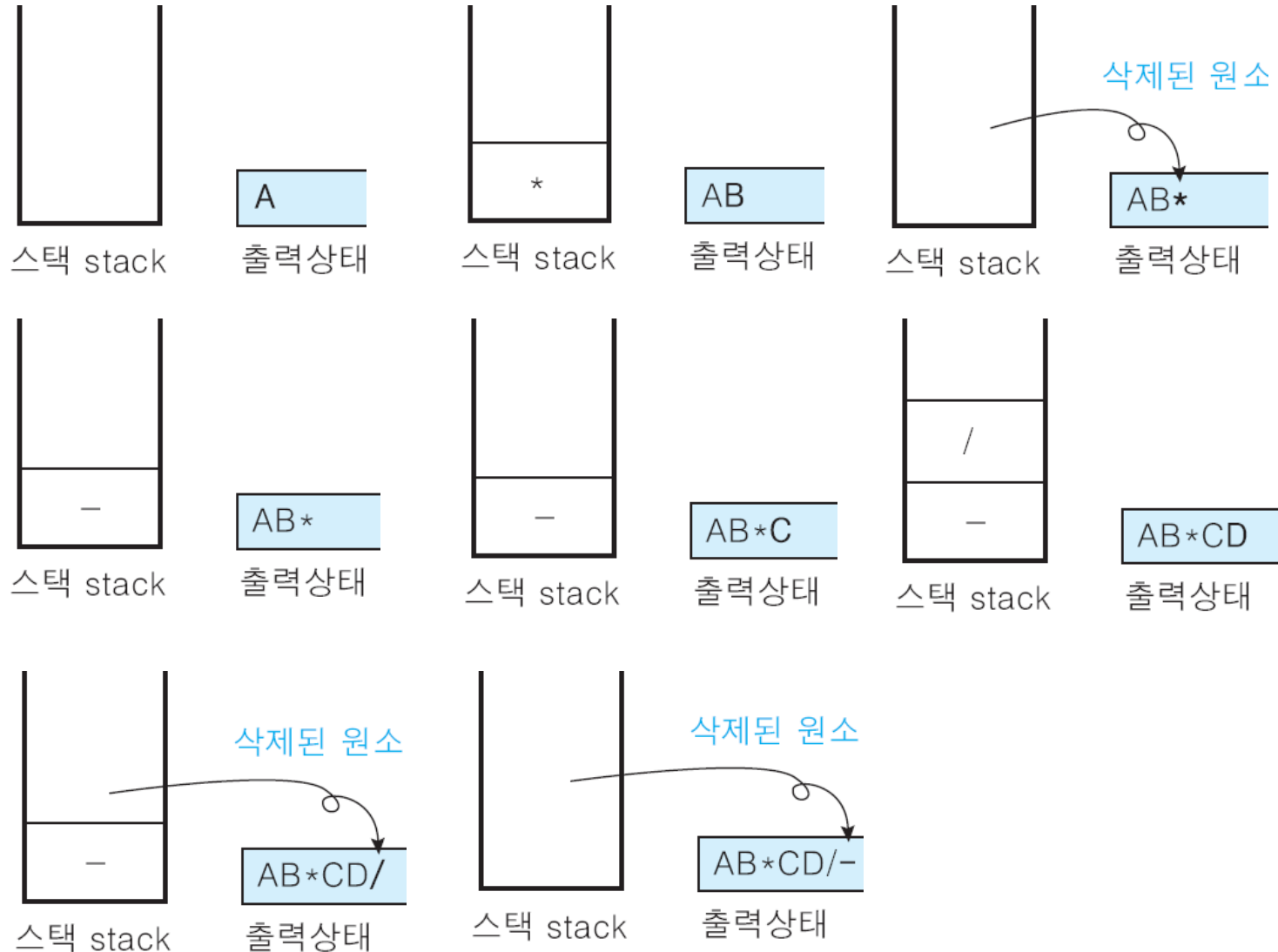
- 변환 방법 : 입력은 모든 연산에 대해 괄호가 있다고 가정.  
예를 들어  $(3+(4*5))$

- ① 왼쪽 괄호를 만나면 무시하고 다음 문자를 읽는다.
- ② 피연산자를 만나면 출력한다.
- ③ 연산자를 만나면 스택에 push한다.
- ④ 오른쪽괄호를 만나면 스택을 pop하여 출력한다.
- ⑤ 수식이 끝나면, 스택이 공백이 될 때까지 pop하여 출력한다.

✓ 연산자 스택 사용

## □ 스택의 응용 - [4] 수식의 후위표기법 변환

■ 예)  $((A*B)-(C/D))$





## □ 스택의 응용 - [4] 수식의 후위표기법 변환

- 중위 표기법 → 후위 표기법 변환 알고리즘

```
infix_to_postfix(exp)
  while(중위 수식 exp의 끝을 만나기 전) 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 :       // 중위 수식의 끝
        while(!isEmpty(stack)) do
          print(pop(stack));
    }
  }
end infix_to_postfix( )
```

## □ 스택의 응용 - [4] 수식의 후위표기법 변환

- 후위 표기 변환 프로그램

```
public class Ex7_3
{
    public static void main(String [] args)
    {
        // 피연산자는 한자리 정수라고 가정
        String exp;
        String postfix;

        exp = "(3*5)-(6/2)";
        postfix = OptExp.toPostfix(exp);

        System.out.println(exp);
        System.out.println("후위표기식 : ");
        System.out.println(postfix);
    }
}
```

## □ 스택의 응용 - [4] 수식의 후위표기법 변환

```
public class OptExp {
    ...
    public static String toPostfix(String infix) {
        char testCh;
        String postfix = "";
        LinkedStack s = new LinkedStack();
        for(int i=0; i< infix.length(); i++) {
            testCh = infix.charAt(i);
            switch(testCh) {
                case '0': case '1': case '2': case '3': case '4': // 피연산자
                case '5': case '6': case '7': case '8': case '9':
                    postfix += testCh;
                    break;
                case '+': case '-': case '*': case '/':           // 연산자
                    s.push(testCh);
                    break;
                case ')':
                    postfix += s.pop(); break;
            }
        }
        while(!s.isEmpty()) postfix += s.pop();
        return postfix;
    }
}
```

## □ 스택의 응용 - [5] 후위표기수식의 연산

❖ 스택을 사용하여 후위표기식을 계산할 수 있다.

■ 계산 방법

- ① 피연산자를 만나면 스택에 **push** 한다.
- ② 연산자를 만나면 필요한 만큼의 피연산자를 스택에서 **pop**하여 연산하고, **연산결과**를 다시 스택에 **push** 한다.
- ③ 수식이 끝나면, 마지막으로 스택을 **pop**하여 출력한다.

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

✓ 피연산자 스택 사용

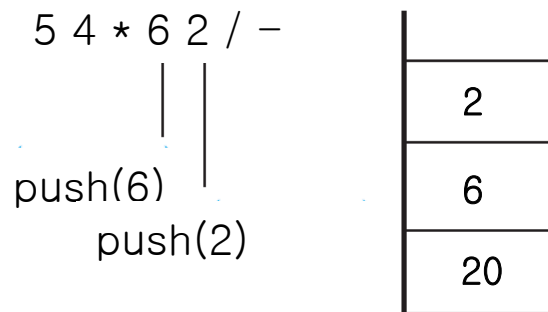
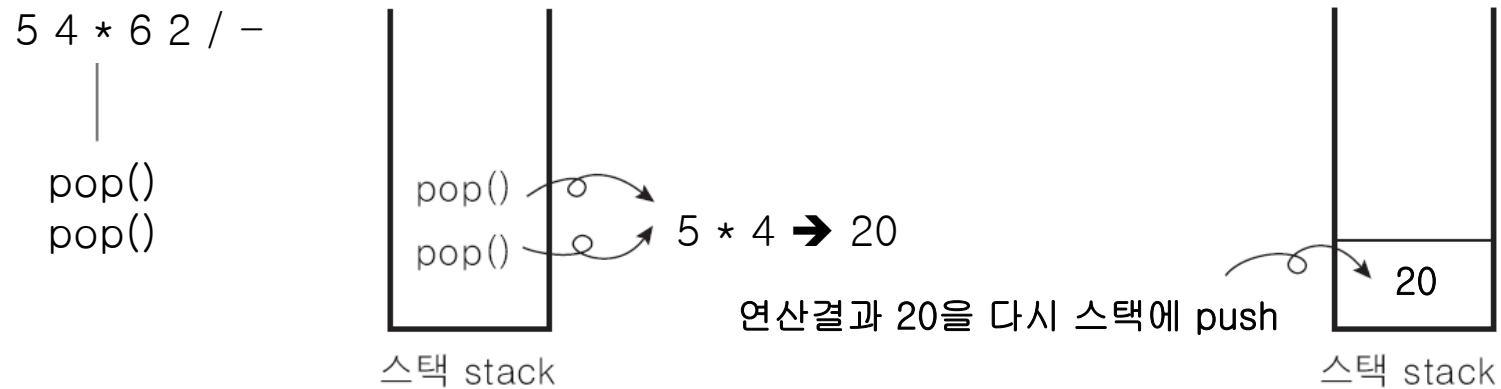
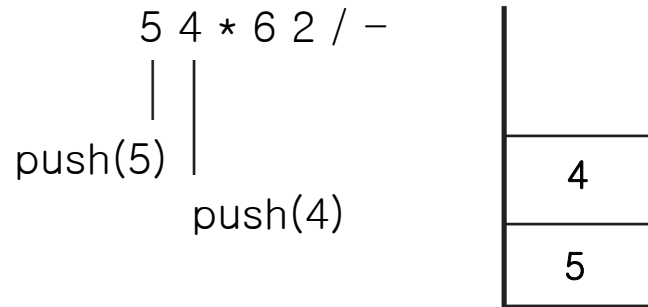
## □ 스택의 응용 - [5] 후위표기수식의 연산

- 후위 표기 수식 계산 알고리즘

```
evalPostfix(exp) // 이진 연산자(binary operator)만 사용한다고 가정
  while (후위 수식 exp의 끝을 만나기 전) 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 계산
        push(stack, result);
      symbol = null : // 후위 수식의 끝
        print(pop(stack));
    }
  }
end evalPostfix()
```

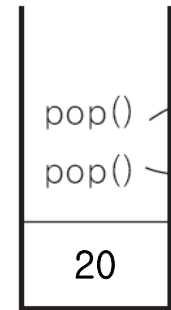
## □ 스택의 응용 - [5] 후위표기수식의 연산

■ 예) 5 4 \* 6 2 / -



## □ 스택의 응용 - [5] 후위표기수식의 연산

5 4 \* 6 2 / -  
 |  
 pop()  
 pop()

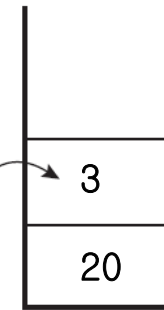


스택 stack

pop()  
 pop()

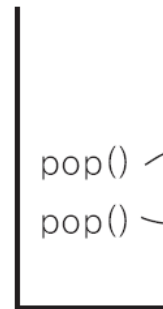
$$6 / 2 \rightarrow 3$$

연산 결과 3을 다시 스택에 삽입



스택 stack

5 4 \* 6 2 / -  
 |  
 pop()  
 pop()

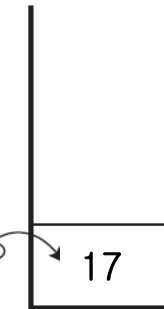


스택 stack

pop()  
 pop()

$$20 - 3 \rightarrow 17$$

연산 결과 17을 다시 스택에 삽입



스택 stack

## ❏ *java.util.Stack* 클래스

❖ java.util 패키지의 Stack 클래스

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<Integer> s = new Stack<Integer>(); // Integer 스택 생성
        s.push(3);
        int n = s.pop();
        ...
    }
}
```