

자료구조

7장 스택(stack)

□ 7장 스택

❖ 스택

❖ 스택의 추상 자료형

❖ 스택의 구현

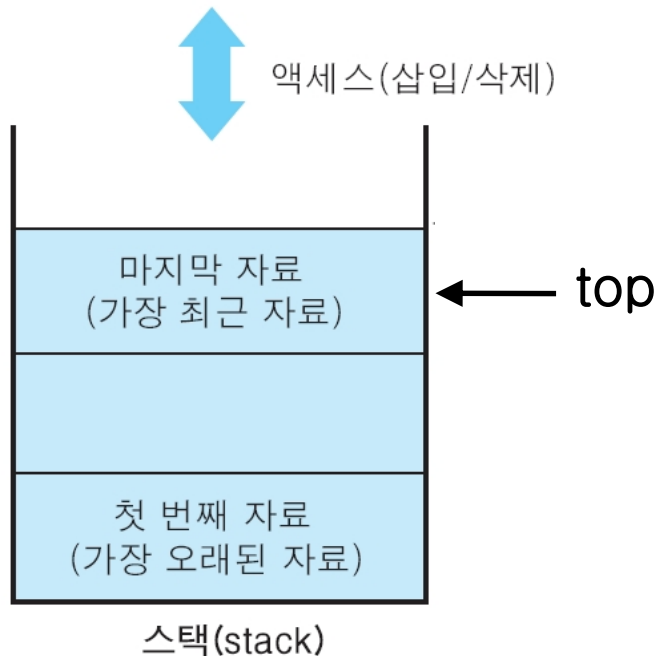
- 순차 자료구조
- 연결 자료구조

❖ 스택의 응용

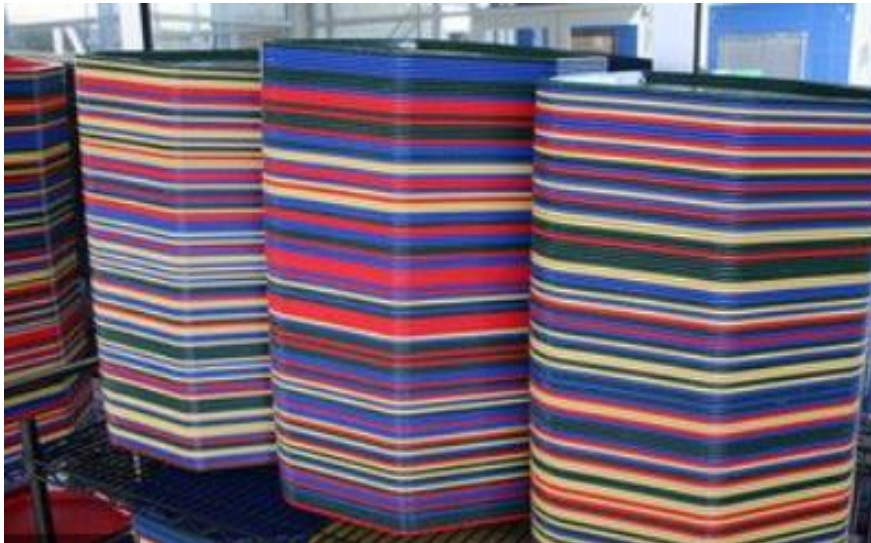
- 역순 문자열 만들기
- 시스템 스택
- 수식의 괄호 검사
- 수식의 후위표기법 변환
- 후위표기수식 계산

❖ 스택(stack)

- 접시 쌓아 둘 때, 맨 위에 있는 접시를 먼저 들어내 사용하고 새 접시는 맨 위에 쌓는다. ➔ 이와 같은 방식의 자료구조가 스택이다.
- 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).

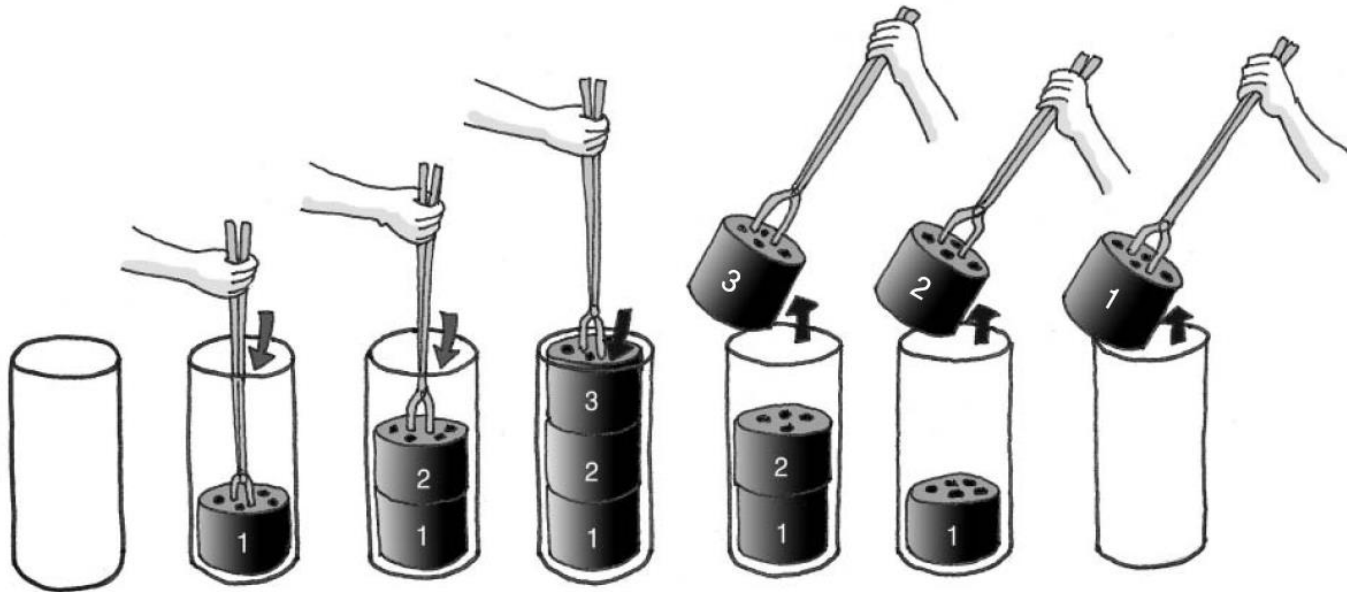


- LIFO 구조의 예 : 식판의 스택
 - 최근에 쌓은 식판을 먼저 꺼낸다.



■ 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.

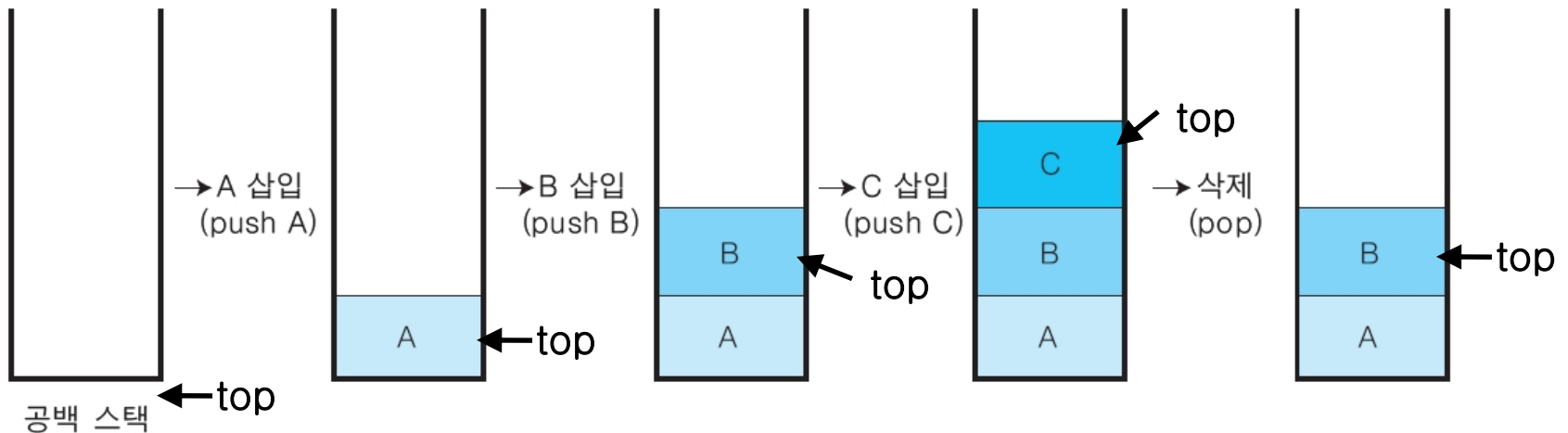


❖ 스택의 연산

- 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

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

$\text{else } \{ \text{delete and return the top item of } S \};$

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

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

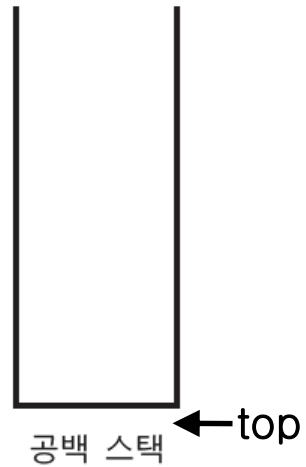
$\text{else return (the top item of the } S);$

// 스택 S의 top에 있는 item을 삭제하지않고 리턴하는 연산

End Stack

❖ 예) 공백 스택에 다음 연산을 모두 수행한 후 스택의 상태는?

push A
push B
pop
push C



□ 7장 스택

❖ 스택

❖ 스택의 추상 자료형

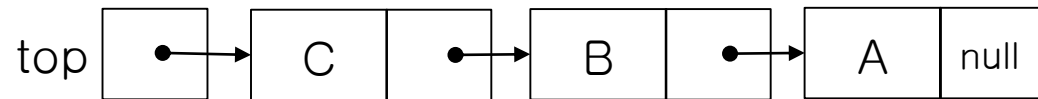
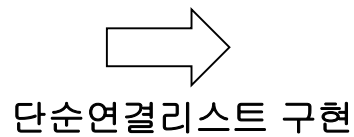
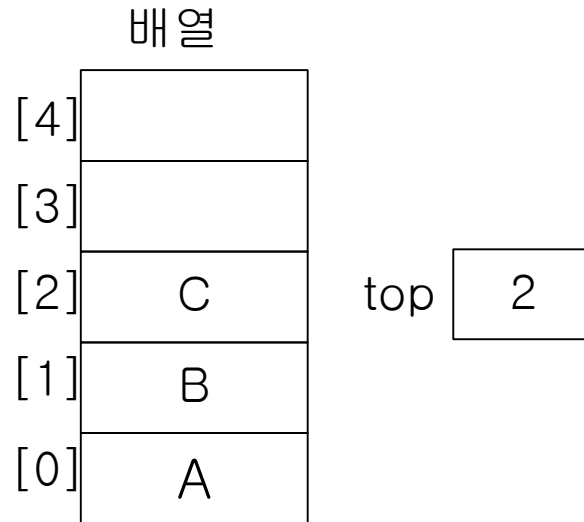
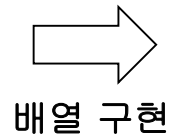
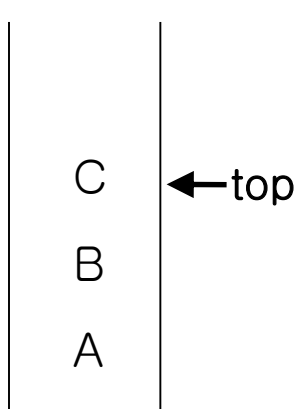
❖ 스택의 구현

❖ 스택의 응용

□ 스택의 구현

❖ 스택의 구현

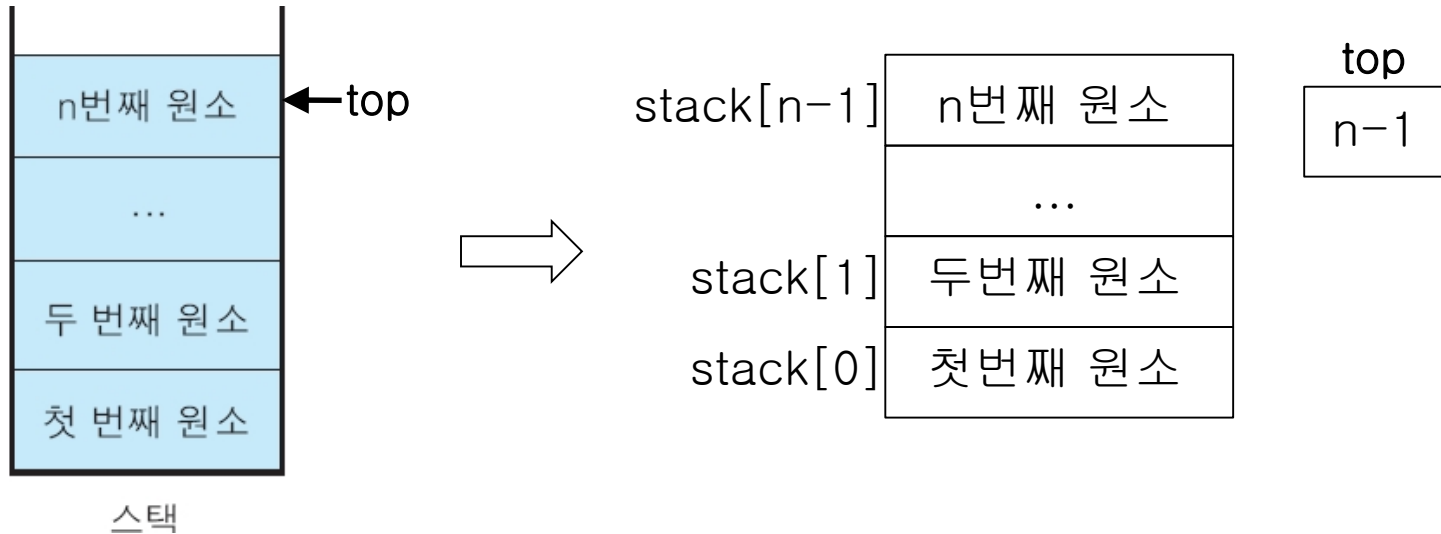
- 순차 자료구조
- 연결 자료구조



□ 스택의 구현 - 순차 자료구조

❖ 순차 자료구조를 이용한 스택의 구현

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



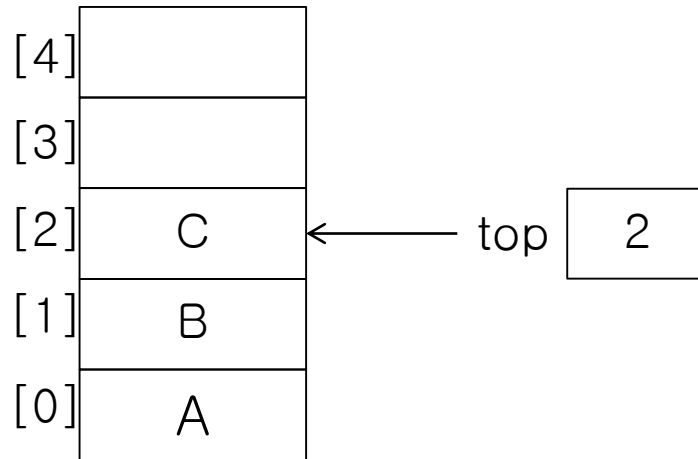
□ 스택의 구현 - 순차 자료구조

❖ 순차 자료구조를 이용한 스택의 구현

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

예) 스택의 크기 `stackSize`가 5인 경우

A, B, C를 차례대로 삽입한 후 스택의 상태



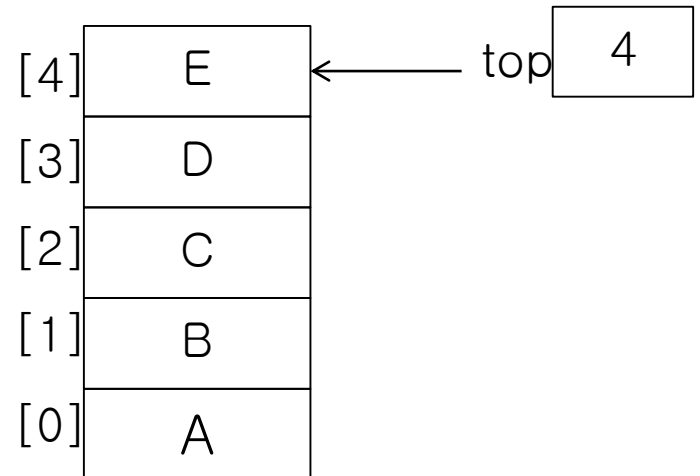
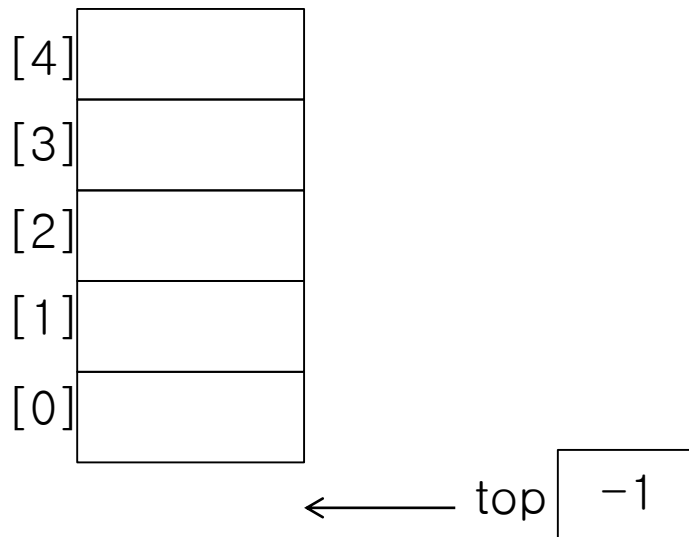
□ 스택의 구현 - 순차 자료구조

- 변수 top의 값으로 스택의 empty 상태와 full 상태를 표시함

empty 상태 : $\text{top} == -1$
(초기 상태)

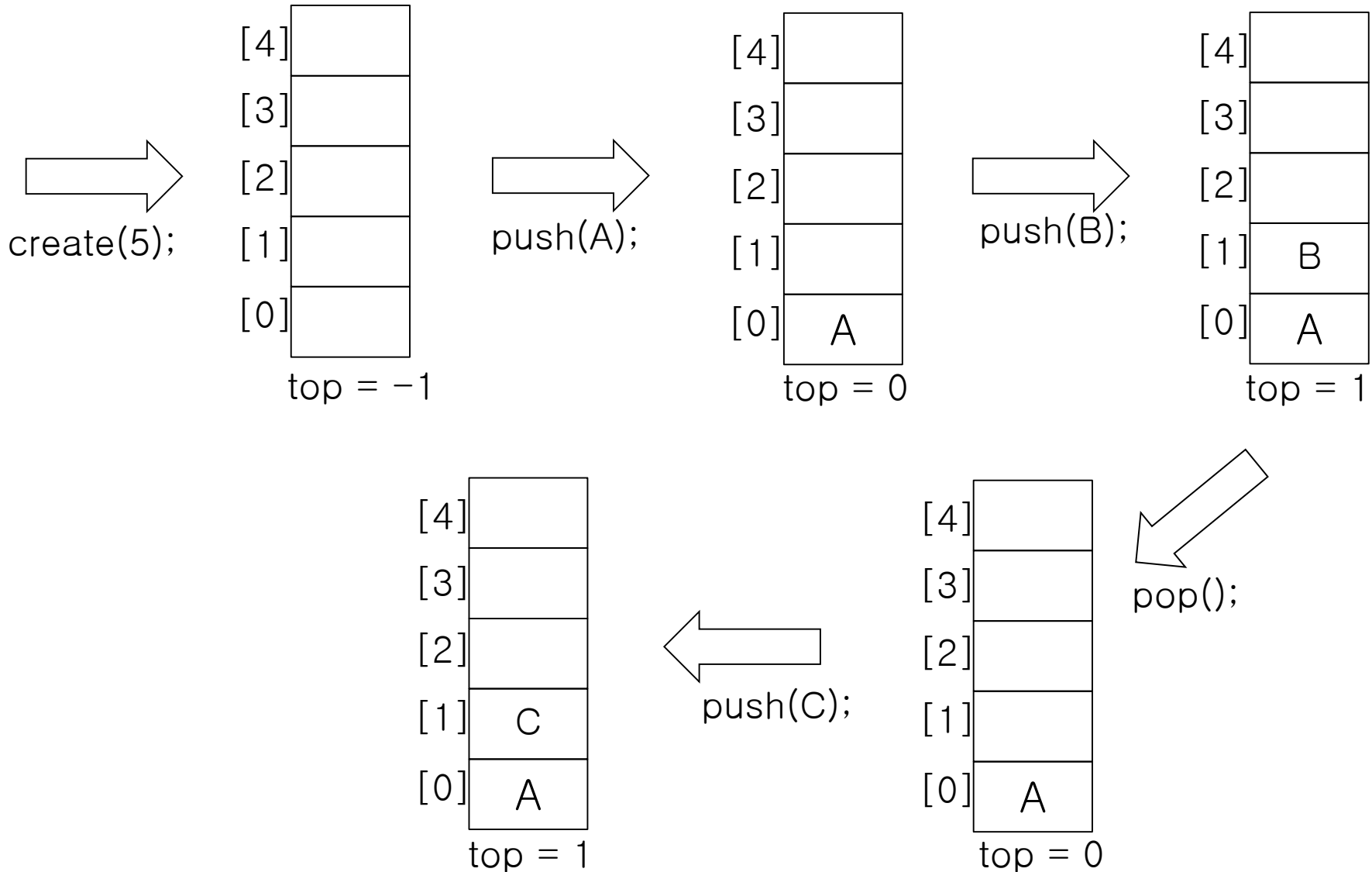
full 상태 : $\text{top} == \text{stackSize} - 1$

예) 스택의 크기 stackSize가 5인 경우



□ 스택의 구현 - 순차 자료구조

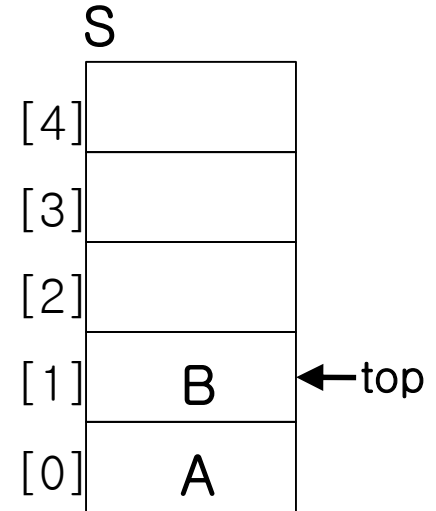
- 크기가 5인 1차원 배열 스택에서 연산 수행과정 예



□ 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
 - $S[top] \leftarrow x;$



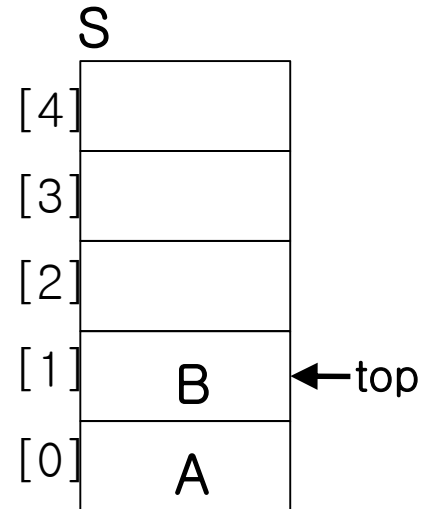
```
push(x)
  if (top = STACK_SIZE-1) then overflow; // stack full
  else {
    top ← top+1;
    S[top] ← x;
  }
end push( )
```

□ 스택의 구현 - 순차 자료구조

❖ 스택의 삭제 연산 : **pop** 알고리즘

- ① 스택이 공백 스택이 아니면 top이 가리키는 원소를 반환
- ② 스택의 top 원소를 삭제. 즉, top이 현재보다 한칸 아래 원소를 가리키도록 변경

$top \leftarrow top - 1;$



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


□ 스택의 구현 - 순차 자료구조

- 자바 구현 - 배열로 구현한 문자 스택

```
public class MyArrayStack {  
    private int top;  
    private char[] array;  
  
    public MyArrayStack() { // 크기 10인 공백 스택 생성  
        top = -1;  
        array = new char[10];  
    }  
  
    public MyArrayStack(int capacity) { // 크기 capacity인 공백 스택 생성  
        top = -1;  
        array = new char[capacity];  
    }  
  
    public boolean isEmpty() { // 스택이 비었는지를 검사  
        return (top == -1);  
    }  
  
    public boolean isFull() { // 스택이 가득찬는지를 검사  
        return (top == array.length - 1);  
    }  
}
```

stack empty 상태 자체는 에러 상태는 아님. stack empty일 때 pop을 수행하는 것이 에러임

// 다음 슬라이드에 계속

□ 스택의 구현 - 순차 자료구조

```
public void push(char item) { // 스택에 item을 삽입
    if(isFull()) {
        System.out.println("Stack full");
    }
    else {
        array[++top] = item;
    }
}
```

스택이 가득찼을 때 오류를 발생시키는 대신에, 스택의 용량을 2배로 늘린 후에 삽입 연산을 수행하도록 구현할 수도 있다.

```
public char pop() { // 스택 꼭대기 원소를 삭제하여 리턴
    if(isEmpty()) {
        throw new EmptyStackException(); // 예외 발생
    }
    else {
        return array[top--];
    }
}
```

// 다음 슬라이드에 계속

□ 스택의 구현 - 순차 자료구조

```
public char peek() { // 스택 꼭대기 원소를 리턴
    if(isEmpty()) {
        throw new EmptyStackException(); // 예외 발생
    }
    else {
        return array[top]; // 스택의 꼭대기 원소를 리턴(삭제하지 않음)
    }
}

} // MyArrayStack 정의 끝
```

□ 스택의 구현 - 순차 자료구조

- 스택을 순차 자료구조로 구현하는 경우 장점
 - 순차 자료구조인 1차원 배열을 사용하여 쉽게 구현
 - 연산의 구현도 쉽고, 빠르게 수행됨
 - Q: 배열로 스택을 구현한 경우, 삽입/삭제시 자료 이동 오버헤드 문제가 발생하는가?
 - 스택을 순차 자료구조로 구현하는 경우 단점
 - 크기가 고정된 배열을 사용하므로 비어있는 공간으로 기억장소가 낭비될 수 있으며,
 - 스택의 크기 변경이 비효율적임
- ➔ **연결 자료구조**를 이용하여 스택을 구현하면 이러한 단점들을 해결할 수 있다.

□ 스택의 구현

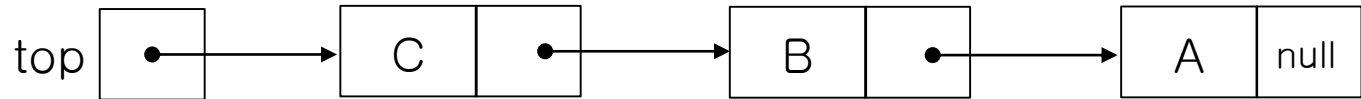
❖ 스택의 구현

- 순차 자료구조
- 연결 자료구조

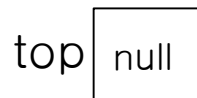
□ 스택의 구현 - 연결 자료구조

❖ 연결 자료구조를 이용한 스택의 구현

- 단순 연결 리스트를 이용하여 구현하면 된다.
 - 스택의 원소 하나를 단순 연결 리스트의 노드 하나로 표현



- 필요한 만큼 노드를 할당 받으면 되므로
 - 스택의 크기 변경 문제가 없고,
 - 비어있는 공간으로 인한 메모리 낭비 문제가 없다.
- 변수 top : 단순 연결 리스트의 첫번째 노드를 가리키는 변수
 - empty 상태 : $top == null$ (초기 상태)

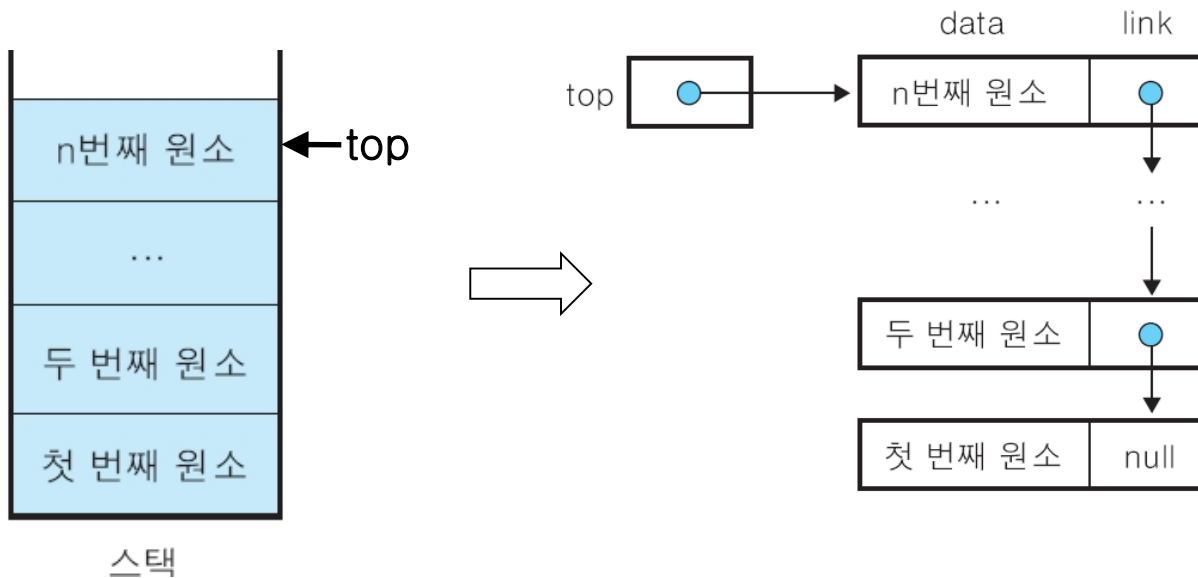


- full 상태 : 정의되지 않음

□ 스택의 구현 - 연결 자료구조

❖ 연결 자료구조를 이용한 스택의 구현

- 삽입/삭제 연산
 - push : 단순 연결 리스트의 가장 앞에 노드 삽입
 - pop : 단순 연결 리스트의 첫번째 노드 삭제



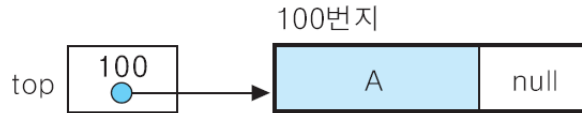
□ 스택의 구현 - 연결 자료구조

- 단순 연결 리스트 스택에서 연산 수행과정 예

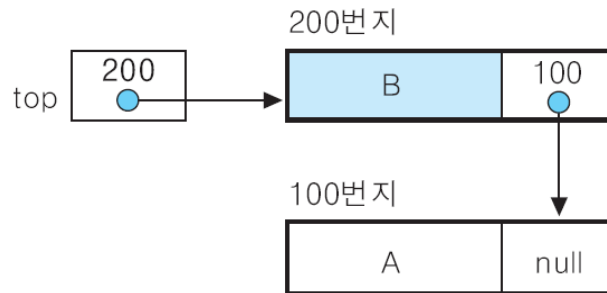
① 공백 스택 생성 : `create();`



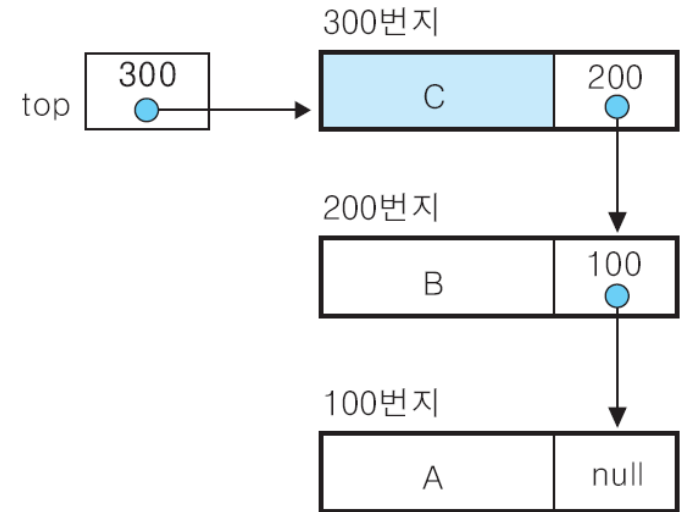
② 원소 A 삽입 : `push(A);`



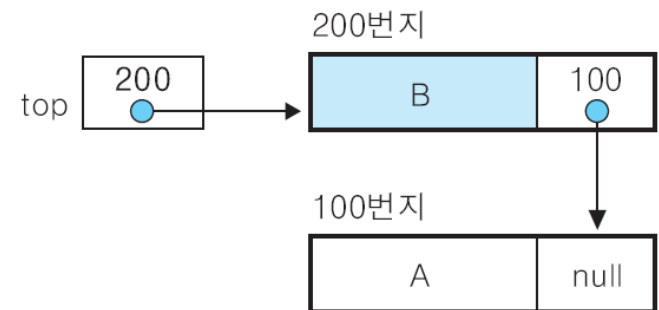
③ 원소 B 삽입 : `push(B);`



④ 원소 C 삽입 : `push(C);`



⑤ 원소 삭제 : `pop();`





참고

- 자바에서 제공하는 문자 Stack 클래스 스택 사용

```
public class Main {  
    public static void main(String[] args) {  
        Stack<Character> s = new Stack<Character>();  
        s.push('A');  
        char ch = s.pop();  
    }  
}
```

- 자바 구현 - 배열로 구현한 문자 스택 사용

```
public class Main {  
    public static void main(String[] args) {  
        MyArrayStack s = new MyArrayStack();  
        s.push('A');  
        char ch = s.pop();  
    }  
}
```

```
public class MyArrayStack {  
    private int top;  
    private char[] array;  
  
    public MyArrayStack() { // 크기 10인 공백 스택 생성  
        top = -1;  
        array = new char[10];  
    }  
  
    public MyArrayStack(int capacity) { // 크기 capacity인 공백 스택 생성  
        top = -1;  
        array = new char[capacity];  
    }  
  
    public boolean isEmpty() { // 스택이 비었는지를 검사  
        return (top == -1);  
    }  
  
    public boolean isFull() { // 스택이 가득찼는지를 검사  
        return (top == array.length - 1);  
    }  
}
```

// 다음 슬라이드에 계속

□ 참고

- 자바 구현 - 단순 연결 리스트로 구현한 문자 스택 사용

```
public class Main {  
    public static void main(String[] args) {  
        MyLinkedStack s = new MyLinkedStack();  
        s.push('A');  
        char ch = s.pop();  
    }  
}
```

```
public class MyLinkedStack {  
    private class Node { // 노드 구조  
        char data;  
        Node link;  
    }  
  
    private Node top = null; // 첫번째 노드를 가리키는 변수  
  
    public boolean isEmpty() { // 스택이 비었는지를 검사  
        return (top == null);  
    }  
  
    public void push(char item) { // 스택에 item을 삽입  
        Node newNode = new Node();  
        newNode.data = item;  
        newNode.link = top;  
        top = newNode;  
    }  
}
```

// 다음 슬라이드에 계속

□ 스택의 구현 - 연결 자료구조

- 자바 구현 - 단순 연결 리스트로 구현한 문자 스택

```
public class MyLinkedStack {  
    private class Node { // 노드 구조  
        char data;  
        Node link;  
    }  
  
    private Node top = null; // 첫번째 노드를 가리키는 변수  
  
    public boolean isEmpty() { // 스택이 비었는지를 검사  
        return (top == null);  
    }  
  
    public void push(char item) { // 스택에 item을 삽입  
        Node newNode = new Node();  
        newNode.data = item;  
        newNode.link = top;  
        top = newNode;  
    }  
}
```

// 다음 슬라이드에 계속

□ 스택의 구현 – 연결 자료구조

```
public char pop() {    // 스택의 꼭대기 원소를 삭제하여 리턴
    if(isEmpty()) {
        throw new EmptyStackException(); // 예외 발생
    }
    else {
        char item = top.data;
        top = top.link;
        return item;
    }
}
```

// 다음 슬라이드에 계속

□ 스택의 구현 - 연결 자료구조

```
public char peek() { // 스택의 꼭대기 원소를 리턴
    if(isEmpty()) {
        throw new EmptyStackException(); // 예외 발생
    }
    else {
        return top.data; // 스택의 꼭대기 원소를 리턴(삭제하지 않음)
    }
}
```

```
} // MyLinkedStack 정의 끝
```

□ 7장 스택

❖ 스택

❖ 스택의 추상 자료형

❖ 스택의 구현

❖ 스택의 응용

❖ 스택의 응용

[1] 역순 문자열 만들기

[2] 시스템 스택

[3] 수식의 괄호 검사

[4] 수식의 후위표기법 변환

[5] 후위표기수식 계산

□ 스택의 응용 - 역순 문자열 만들기

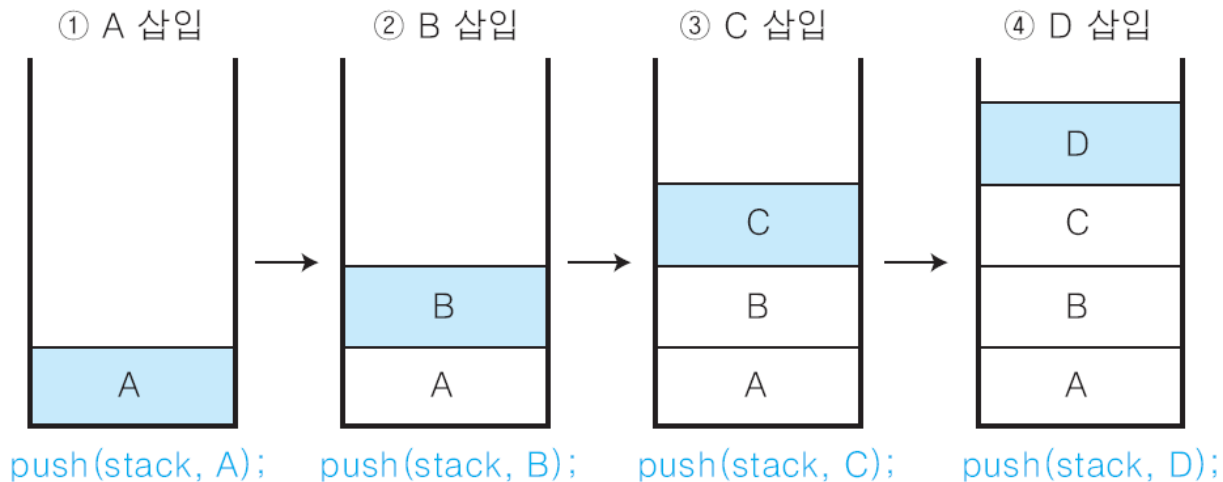
❖ 역순 문자열 만들기

- 스택의 후입선출(LIFO) 성질을 이용

① 문자열의 문자들을 순서대로 스택에 push 하기

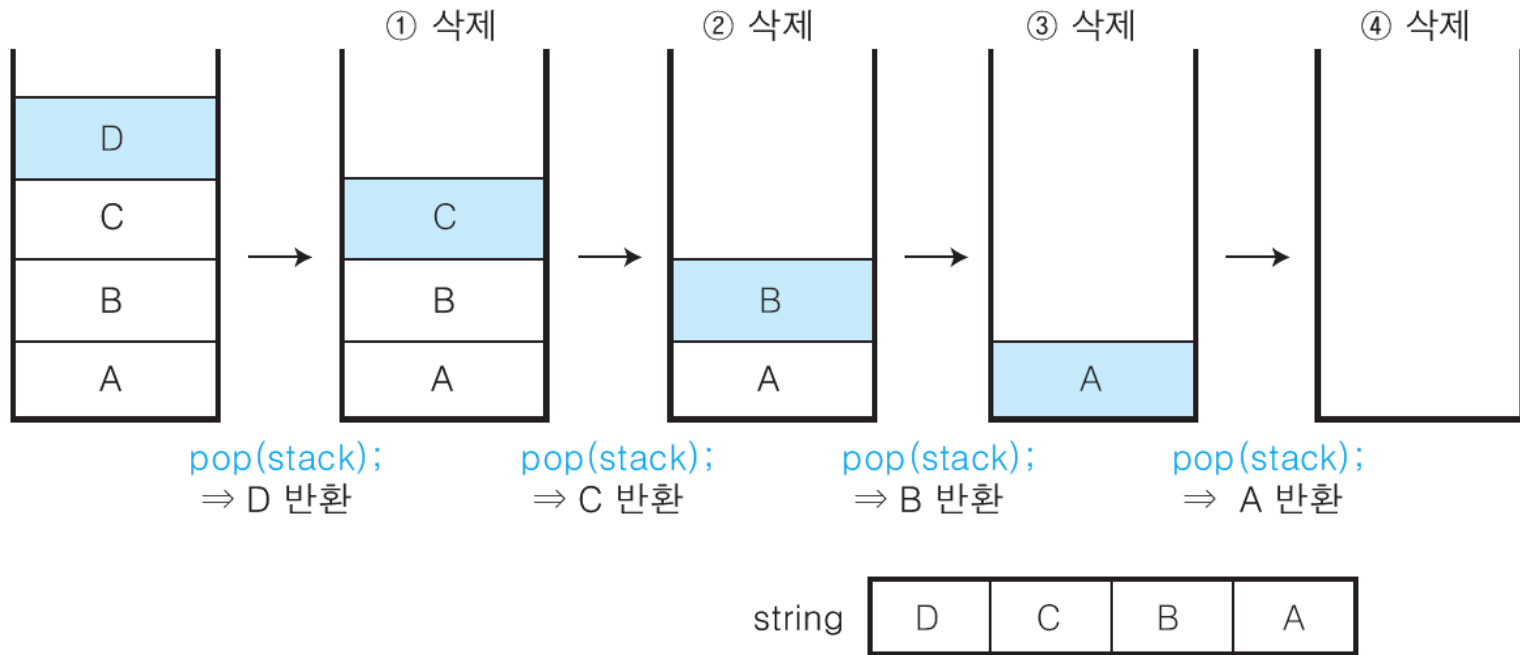
string

A	B	C	D
---	---	---	---



□ 스택의 응용 - 역순 문자열 만들기

② 스택을 pop하여 나오는 순서대로 문자들을 문자열에 저장하기



❖ 스택의 응용

[1] 역순 문자열 만들기

[2] 시스템 스택

[3] 수식의 괄호 검사

[4] 수식의 후위표기법 변환

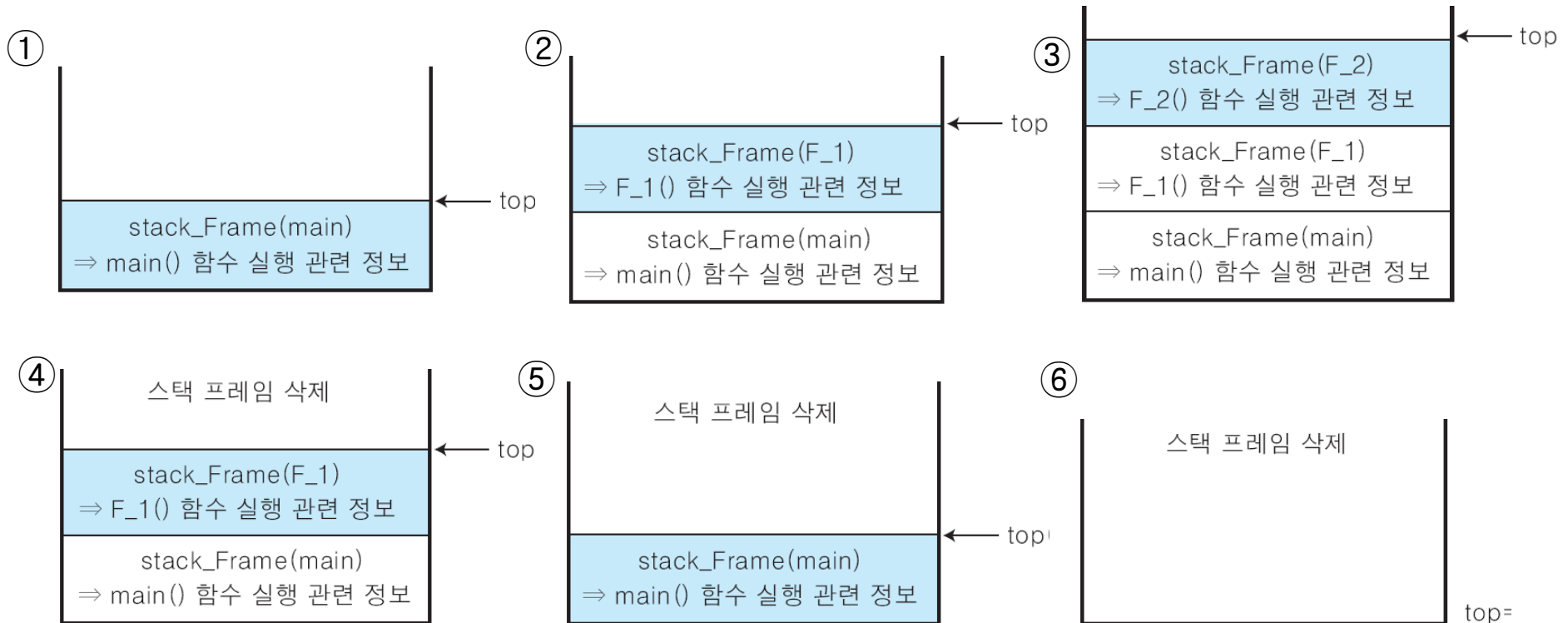
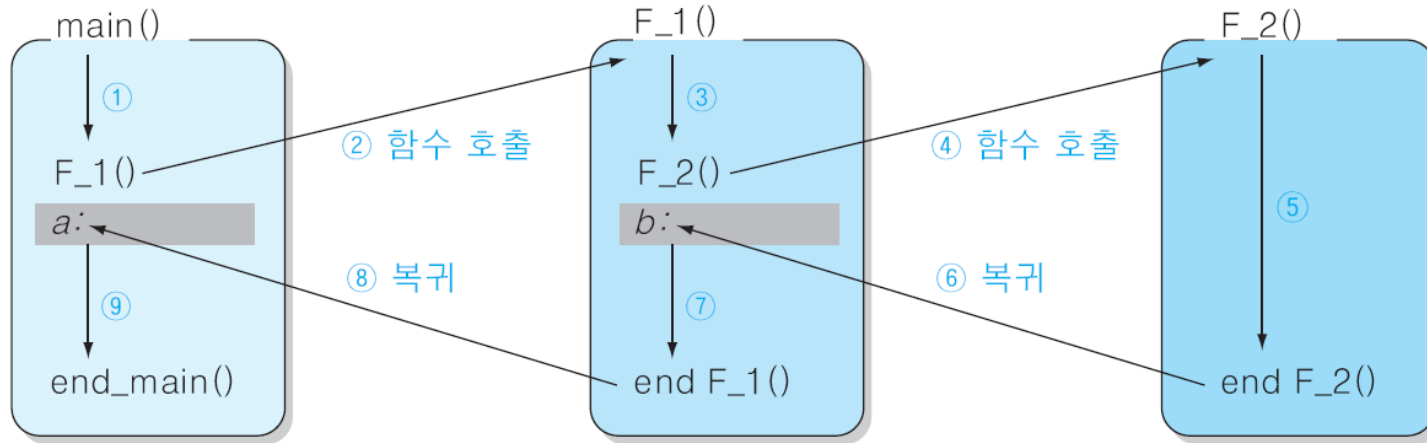
[5] 후위표기수식 계산

□ 스택의 응용 - 시스템 스택

❖ 시스템 스택

- 프로그램에서의 호출과 복귀에 따른 수행 순서를 관리
 - 가장 마지막에 호출된 함수가 가장 먼저 실행을 완료하고 복귀하는 후입선출 구조이므로, 후입선출 자료구조인 스택을 이용하여 수행순서 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 - 시스템 스택



❖ 스택의 응용

[1] 역순 문자열 만들기

[2] 시스템 스택

[3] 수식의 괄호 검사

[4] 수식의 후위표기법 변환

[5] 후위표기수식 계산

□ 스택의 응용 - 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 검사할 수 있다.

□ 스택의 응용 - 수식의 괄호 검사

■ 괄호 검사 방법

[$a * (b + c)$]

- 수식을 왼쪽에서 오른쪽으로 하나씩 읽으면서 검사
- 왼쪽 괄호(여는 괄호)를 읽으면 스택에 push
- 오른쪽 괄호(닫는 괄호)를 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 공백이어야 함
 - 스택이 공백이 아니면 잘못된 수식임
예) (() ()

□ 스택의 응용 - 수식의 괄호 검사

■ 수식의 괄호 검사 알고리즘

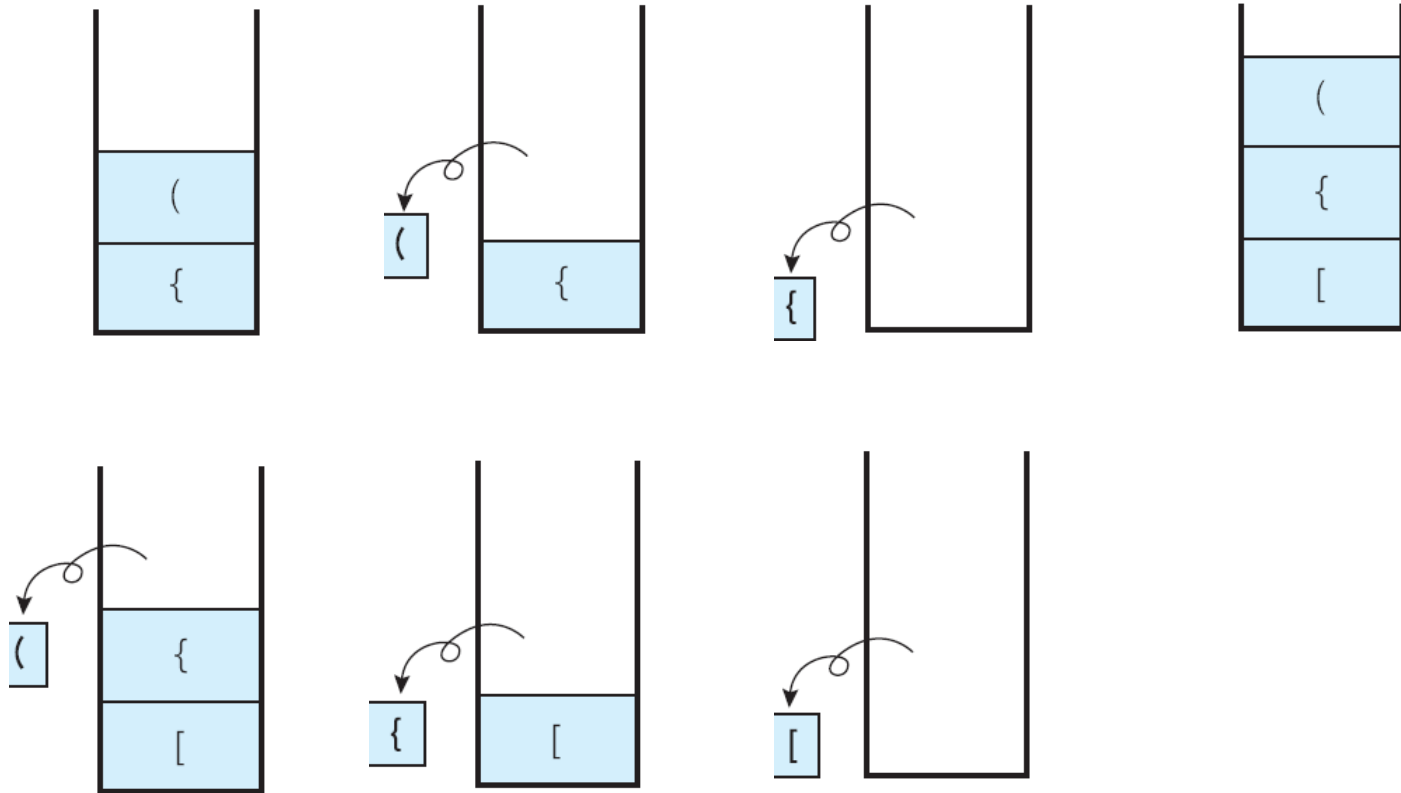
```
testPair( )  
  exp ← Expression;      Stack ← null;  
  while (true) do {  
    symbol ← getSymbol(exp); // 수식에서 심볼 하나를 읽어 들임  
    case {  
      symbol = "(" or "[" or "{" : // 왼쪽 괄호이면 스택에 삽입  
        push(Stack, symbol);  
      symbol = ")" : // 오른쪽 괄호이면  
        open_pair ← pop(Stack); // 스택에서 삭제한 심볼과 짝이 맞는지 검사  
        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 : // 더 이상 읽을 심볼이 없는 경우  
        if (isEmpty(Stack)) then return true; // 스택이 비어야 함  
        else return false;  
    }  
  }  
end testPair( )
```

스택에 더 이상 삭제할 심볼이 없으면 return false;

□ 스택의 응용 - 수식의 괄호 검사

- 수식의 괄호 검사 예

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



❖ 스택의 응용

[1] 역순 문자열 만들기

[2] 시스템 스택

[3] 수식의 괄호 검사

[4] 수식의 후위표기법 변환

[5] 후위표기수식 계산

□ 스택의 응용 - 수식의 후위표기법 변환

❖ 수식의 표기법

■ 중위표기법(infix notation)

- 연산자를 피연산자의 가운데 표기하는 방법
- 연산 순서는 연산자 우선순위를 고려해야 하며, 특정 연산을 먼저 수행하려면 괄호를 사용

➤ 예) $A + B$ $A + B * C$ $(A + B) * C$

■ 후위표기법(postfix notation)

- 연산자를 피연산자 뒤에 표기하는 방법
- 연산자 우선순위가 이미 반영된 수식이므로 우선순위를 고려하지 않고 차례대로 연산을 수행하며, 괄호가 없음

➤ 예) $A B +$ $A B C * +$ $A B + C *$

■ 전위표기법(prefix notation)

- 연산자를 앞에 표기하고 그 뒤에 피연산자를 표기하는 방법
- 연산자 우선순위가 이미 반영된 수식이므로 우선순위를 고려하지 않고 차례대로 연산을 수행하며, 괄호가 없음

➤ 예) $+ A B$ $+ A * B C$ $* + A B C$

□ 스택의 응용 - 수식의 후위표기법 변환

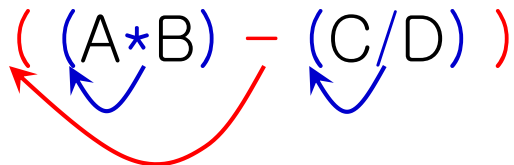
■ 중위표기 to 전위표기 변환 방법

- 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 - 수식의 후위표기법 변환


■ 중위표기 to 후위표기 변환 방법

- ① 연산자 우선순위에 따라 괄호를 사용하여 수식을 다시 표현한다.
- ② 각 연산자를 그에 대응하는 오른쪽 괄호의 뒤로 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 - 수식의 후위표기법 변환

❖ 중위표기와 후위표기

- 실생활에서 일반적으로 사용하는 표기법은 중위표기법
- 컴퓨터가 수식을 계산하기에 가장 효율적인 표기법은 후위표기법
 - 후위표기법은 괄호나 연산자 우선순위를 고려하지 않고 왼쪽에서 오른쪽으로 읽으면서 순서대로 처리할 수 있다.
- 스택을 이용한 컴퓨터의 수식 처리

중위 표기 수식 → 후위 표기 수식 → 수식 계산
 스택 이용 스택 이용

□ 스택의 응용 - 수식의 후위표기법 변환

❖ 스택을 사용한 중위표기 to 후위표기 변환

- 교재에서는 연산자 우선순위(precedence)의 개념은 다루지 않는다.
- 따라서 모든 연산자가 우선순위가 동일하다고 가정하고, 중위표기 수식에서 연산의 순서는 괄호로 표현한다.

$(3 + (4 * 5)) \rightarrow 3\ 4\ 5\ *\ +$

$((3 + 4) * 5) \rightarrow 3\ 4\ +\ 5\ *$

- 변환 결과인 후위표기 수식은 괄호 없이 연산의 순서를 표현함

□ 스택의 응용 - 수식의 후위표기법 변환

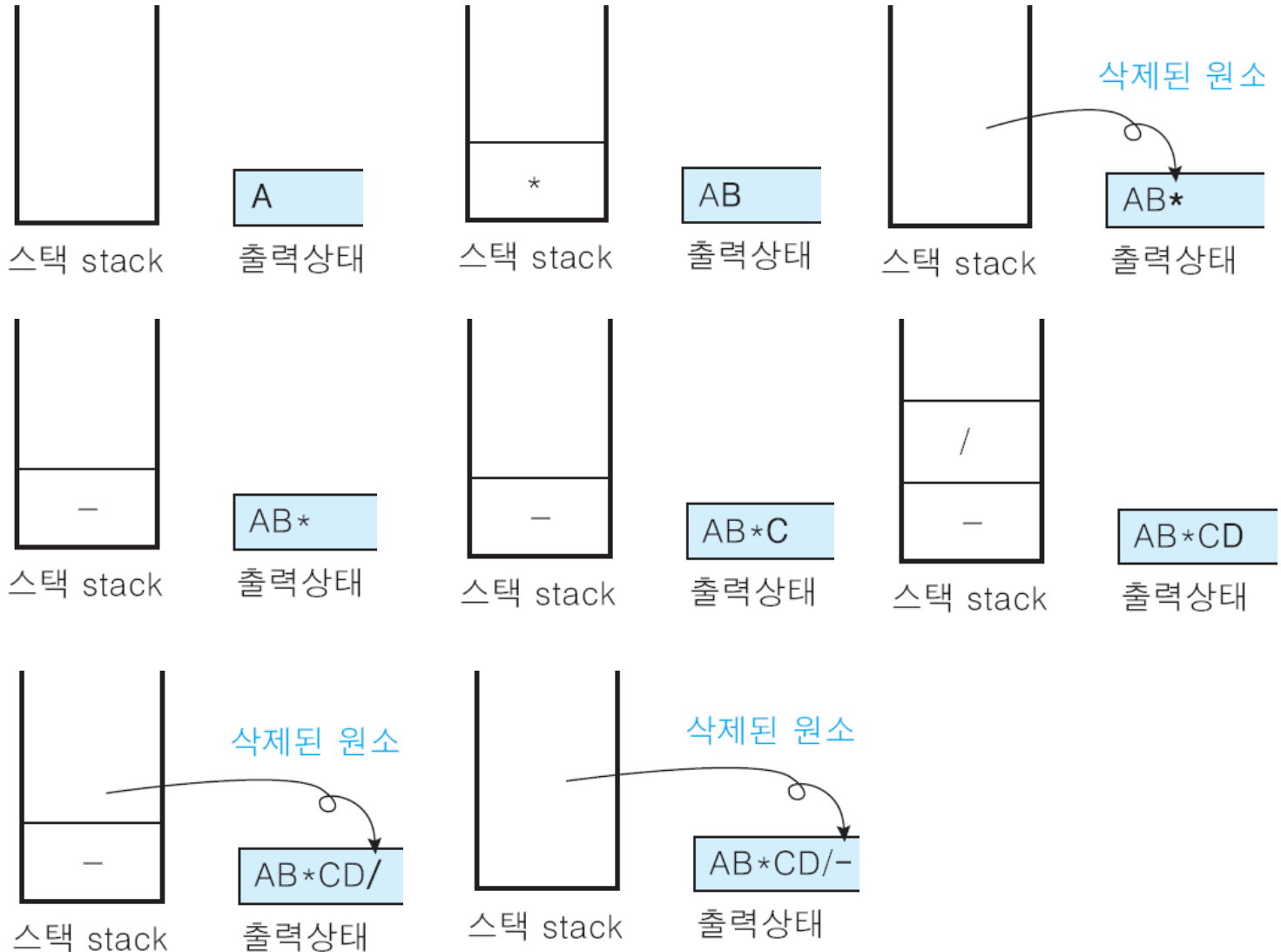
- 중위 표기 → 후위 표기 변환 방법 :
 - 입력(중위수식)은 모든 연산에 대해 괄호가 있다고 가정한다.
예를 들어 (3 + (4 * 5))

- ① 왼쪽 괄호를 만나면 무시하고 다음 문자를 읽는다.
- ② 피연산자를 만나면 **출력**한다.
- ③ 연산자를 만나면 스택에 **push**한다.
- ④ 오른쪽 괄호를 만나면 스택을 **pop**하여 **출력**한다.
- ⑤ (수식이 끝나면, 스택이 공백이 될 때까지 pop하여 출력한다.)

✓ 연산자 스택 사용

□ 스택의 응용 – 수식의 후위표기법 변환

■ 예) $((A * B) - (C / D))$



□ 스택의 응용 – 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 - 수식의 후위표기법 변환

- 후위 표기 변환 Java 프로그램

```
public class PostfixProgram {  
    public static void main(String [] args) {  
  
        // 가정: 피연산자는 한자리 정수, 연산자는 +, -, *, /  
        String infix = "((3*5)-(6/2))";  
        String postfix = toPostfix(infix);  
  
        System.out.println(infix);  
        System.out.print("후위표기 수식 = ");  
        System.out.println(postfix);  
    }  
    // 다음 슬라이드에 계속
```

실행 결과

((3*5)-(6/2))

후위표기 수식 = 35*62/-

□ 스택의 응용 - 수식의 후위표기법 변환

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

□ 스택의 응용 - 수식의 후위표기법 변환

❖ 참고: 스택을 사용한 중위표기 to 후위표기 변환

- 연산자(이진 연산자) 우선순위를 반영한 경우

심볼	In-Stack Priority	In-Coming Priority
)	-	-
*, /	2	2
+, -	1	1
(0	4
$-\infty$	-1	-

- ① $-\infty$ 를 push한다.
- ② 피연산자를 만나면, 출력한다.
- ③)를 만나면, (가 나올 때까지 스택에서 pop하여 출력한다.
- ④)가 아닌 심볼 x를 만나면, ISP(스택 top의 심볼) \geq ICP(x) 인 동안 스택에서 pop하여 출력한 후, 심볼 x를 push한다.
- ⑤ 수식이 끝나면, 스택이 공백이 될 때 까지 pop하여 $-\infty$ 를 제외하고 출력한다.

❖ 스택의 응용

- [1] 역순 문자열 만들기
- [2] 시스템 스택
- [3] 수식의 괄호 검사
- [4] 수식의 후위표기법 변환
- [5] 후위표기수식 계산

□ 스택의 응용 - 후위표기수식 계산

❖ 스택을 사용한 후위표기 수식 계산

■ 계산 방법

✓ 피연산자 스택 사용

- 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 남아있는 원소는 전체 수식의 연산 결과값이며, 이 마지막 원소를 pop하여 결과값을 얻은 후에는 스택이 비어있어야 한다. 예) 3 4 5 * +
- 단계 3 수행 후 스택이 비어있지 않으면 수식에 오류가 있는 것임.
오류 예) 5 4 3 *
- 단계 2에서 스택 pop 오류가 발생하는 경우에도 수식에 오류가 있는 것임.
오류 예) 5 4 * +

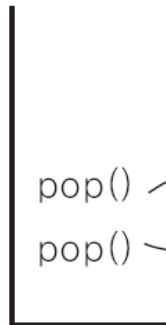
□ 스택의 응용 - 후위표기수식 계산

■ 예) 5 4 * 6 2 / -

5 4 * 6 2 / -
| |
push(5) |
 push(4)

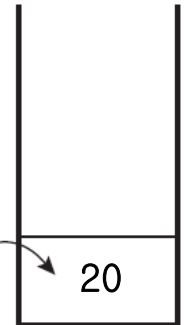


5 4 * 6 2 / -
|
pop()
pop()

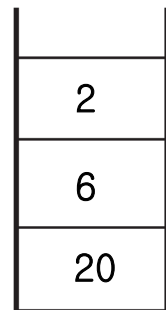


5 * 4 → 20

연산결과 20을 다시 스택에 push



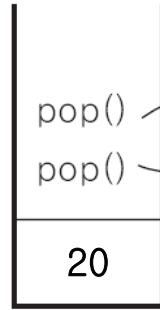
5 4 * 6 2 / -
 | |
push(6) |
 push(2)



□ 스택의 응용 - 후위표기수식 계산

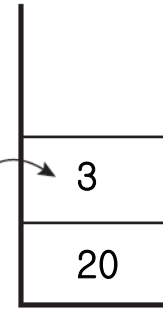
5 4 * 6 2 / -

pop()
pop()



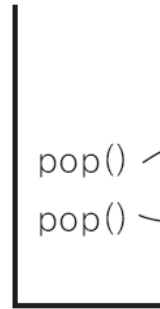
$6 / 2 \rightarrow 3$

연산 결과 3을 다시 스택에 push



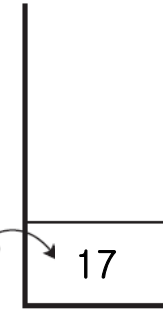
5 4 * 6 2 / -

pop()
pop()



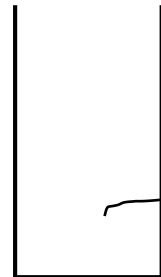
$20 - 3 \rightarrow 17$

연산 결과 17을 다시 스택에 push



5 4 * 6 2 / -

pop()



17

□ 스택의 응용 - 후위표기수식 계산

- 후위표기 수식 계산 알고리즘

```
evalPostfix(postfix) // 이진 연산자 +, -, *, /만 사용한다고 가정
while (후위 수식 postfix의 끝을 만나기 전) do {
    symbol ← getSymbol(postfix);
    if (symbol이 operand이면) { // 피연산자는 스택에 삽입
        push(stack, symbol);
    }
    else { // symbol이 operator이면 계산하여 결과를 스택에 삽입
        opr2 ← pop(stack);
        opr1 ← pop(stack);
        result ← opr1 op(symbol) opr2;
        // 스택에서 꺼낸 피연산자들을 symbol연산자로 계산
        // 즉, symbol이 '+'이면 opr1+opr2, '-'이면 opr1-opr2
        // '*'이면 opr1*opr2, '/'이면 opr1/opr2
        push(stack, result);
    }
}
return pop(stack); // 스택에 마지막으로 꺼낸 값이 결과값임
end evalPostfix()
```

□ 7장 스택

❖ 참고 - java.util 패키지의 Stack 클래스 이용

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // 문자를 저장할 Character형 스택을 생성하고 이용
        Stack<Character> stack = new Stack<>();
        stack.push('a');
        char ch = stack.pop();
        ...
    }
}
```

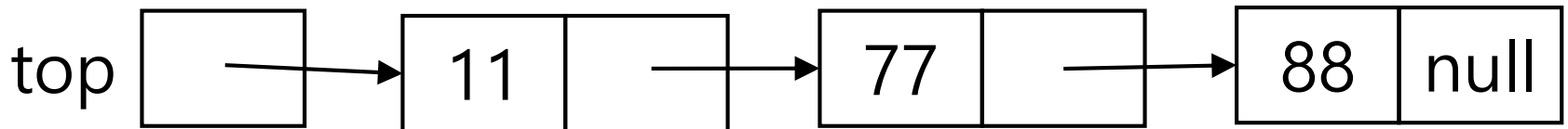
7장 문제 1

- ❖ 그림과 같이 연결리스트로 구현한 스택(stack)에 다음 연산을 모두 수행한 후 상태는?

`stack.pop();`

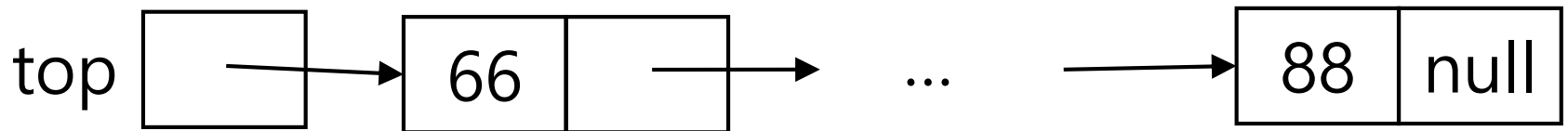
`stack.push(66);`

`stack.push(22);`



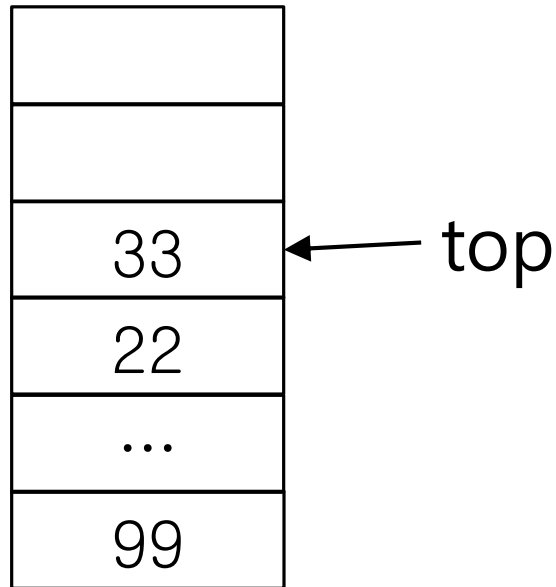
7장 문제 2

- ❖ 연결리스트로 구현한 스택의 push/pop 시간 복잡도는? 단, 스택의 원소 수는 n



7장 문제 3

- ❖ 배열로 구현한 스택의 push/pop 시간 복잡도는? 단, 스택의 원소 수는 n



7장 문제 4

- ❖ infix notation으로 표현된 다음 수식을 postfix notation과 prefix notation으로 변환하세요.

$$a + b * (c - d)$$

7장 문제 5

- ❖ 스택을 이용하여 postfix notation으로 표현된 다음 수식을 계산하세요.

2 3 * 4 + 9 7 - /