

5장 Stack

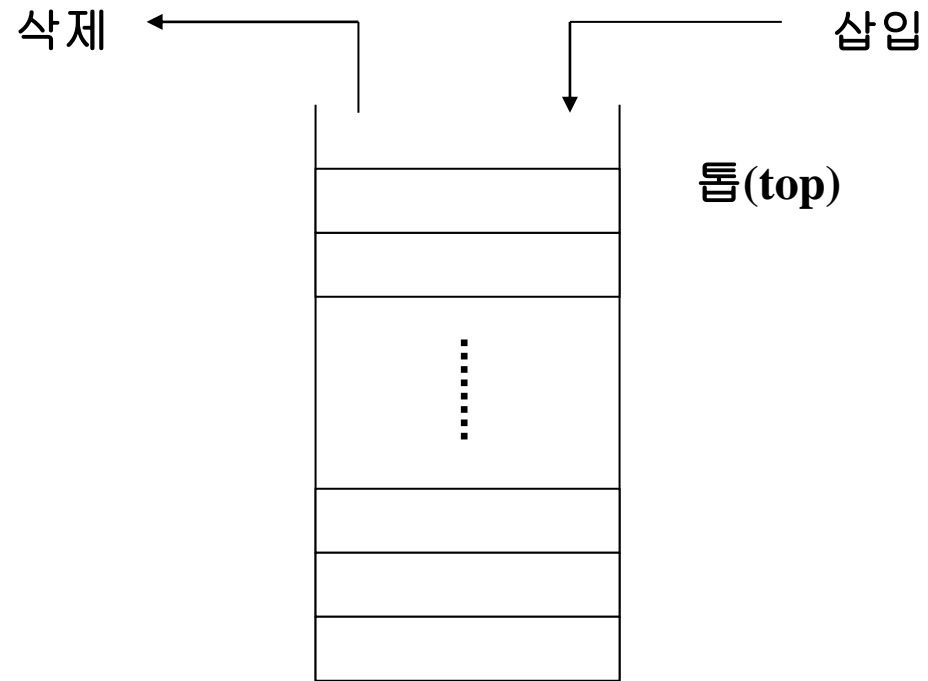
순서

- 5.1 스택 추상 데이터 타입
- 5.2 스택의 순차 표현
- 5.3 배열을 이용한 스택의 구현
- 5.4 복수 스택의 순차 표현
- 5.5 스택의 연결 표현
- 5.6 리스트를 이용한 스택 구현
- 5.7 수식의 괄호쌍 검사
- 5.8 스택을 이용한 수식의 계산
- 5.9 미로문제

5.1 스택 추상 데이터 타입

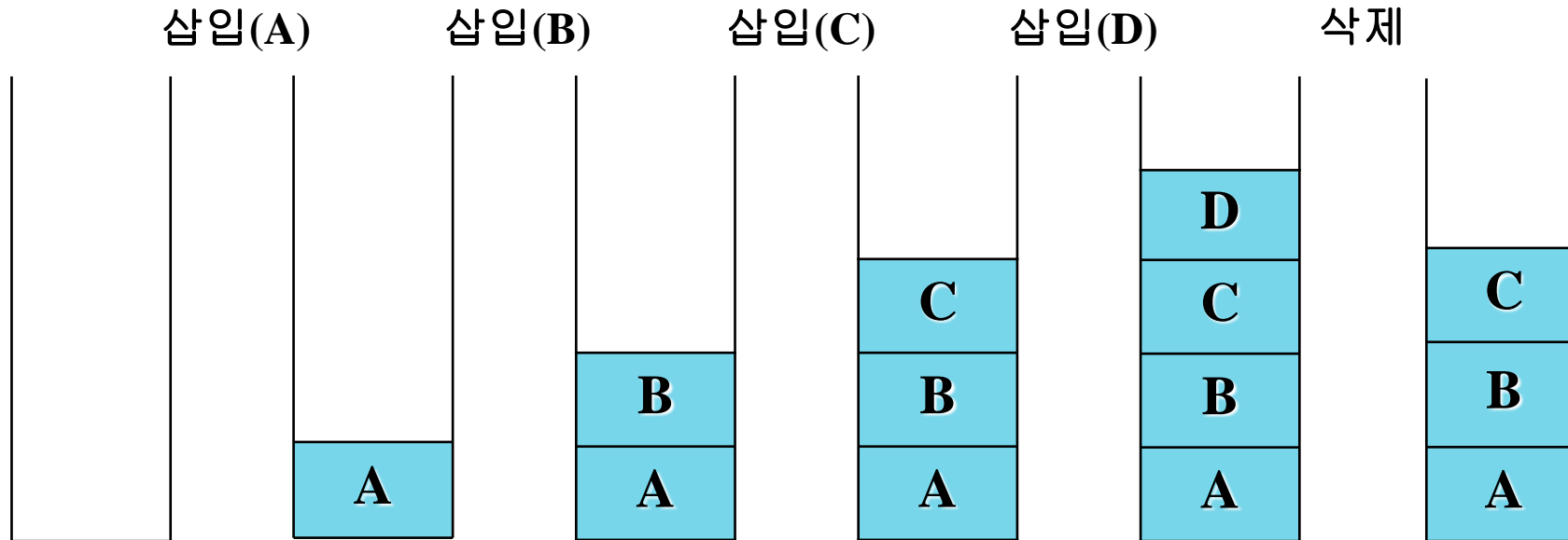
스택(stack)의 정의

- ▶ 삽입과 삭제과 한쪽 끝, 톱(top)에서만 이루어지는 유한 순서 리스트 (finite ordered list)



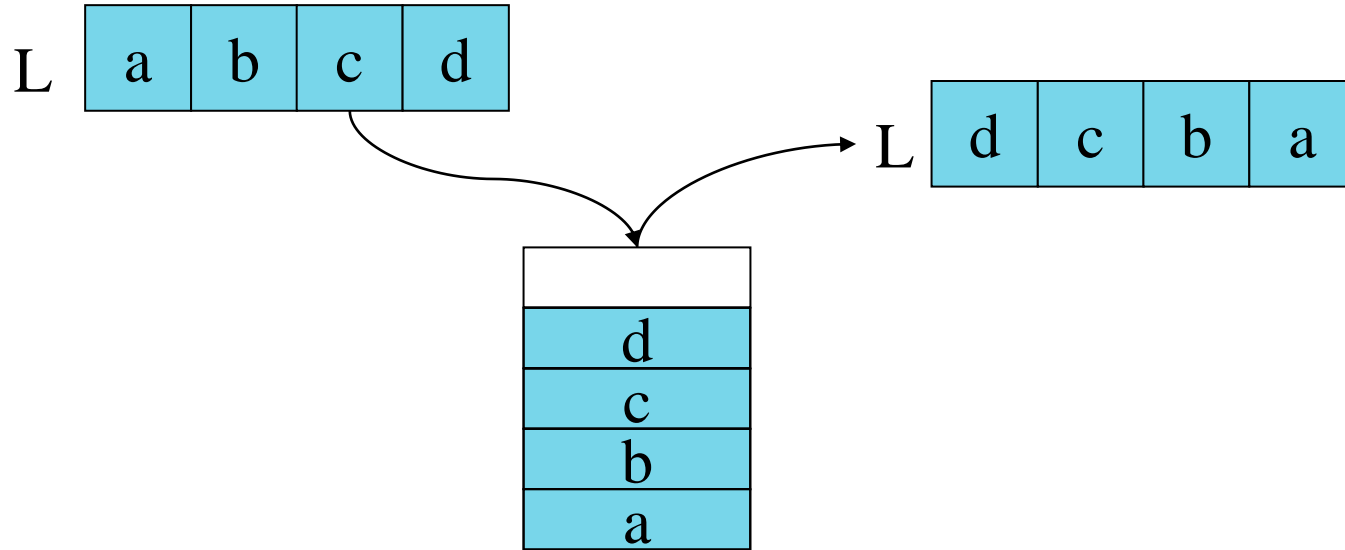
후입 선출 (Last-In-First-Out (LIFO)) 리스트

- ▶ 삽입 : push, 삭제 : pop
- ▶ 스택을 pushdown 리스트라고도 함



스택의 응용

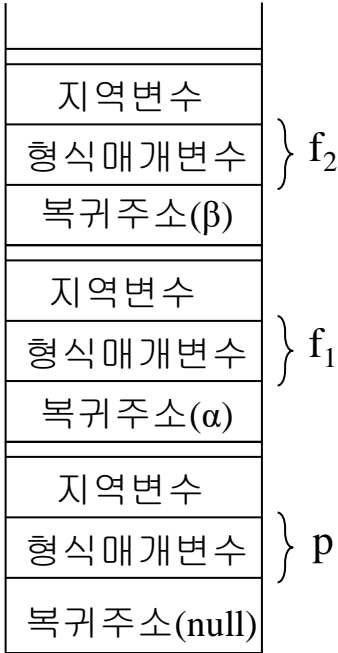
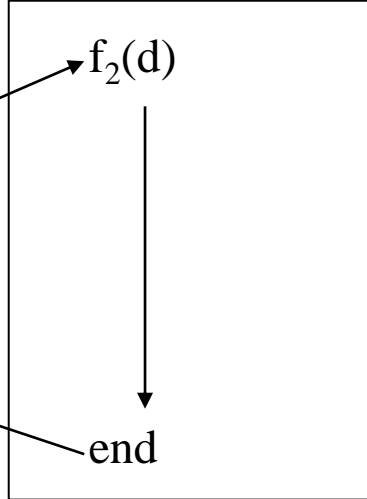
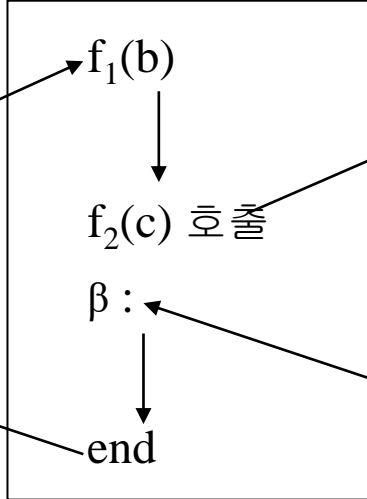
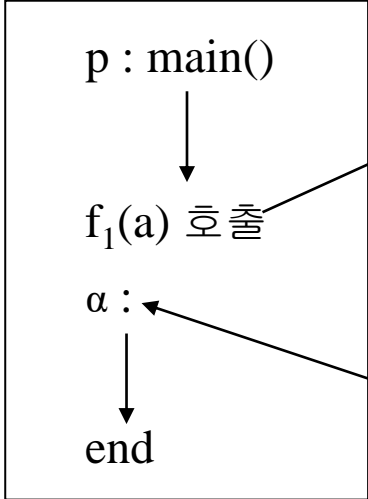
- ▶ 리스트의 순서를 역순으로 만드는 데 유용



- ▶ 컴퓨터 시스템이 사용: 시스템 스택(system stack) 또는 실행 시간 스택(runtime stack)

시스템 스택(system stack)

- ▶ 프로그램 간의 호출과 복귀에 따른 실행 순서를 관리
- ▶ 활성화 레코드(activation record)를 만들어 스택에 삽입
 - 복귀 주소, 형식 매개 변수, 지역 변수들을 포함
 - 스택의 톱에는 항상 현재 실행되는 함수의 활성화 레코드가 존재



Recursive Call (순환 호출)

- ▶ 순환 호출 (recursive call)
 - 순환 호출이 일어날 때마다 활성화 레코드가 만들어져 시스템 스택에 삽입됨
 - 가능한 순환 호출의 횟수는 활성화 레코드의 최대 개수를 얼마로 정하느냐에 따라 결정
- ▶ 순환 프로그램의 실행이 느린 이유
 - 활성화 레코드들의 생성과 필요한 정보 설정 등의 실행 환경 구성에 많은 시간이 소요

스택 추상 데이터 타입

```
package hufs.dislab.util;

public interface Stack<E> {
    public boolean empty();
    public E peek();
    public E pop();
    public E push(E item);
}
```

<<interface>>
Stack<E>

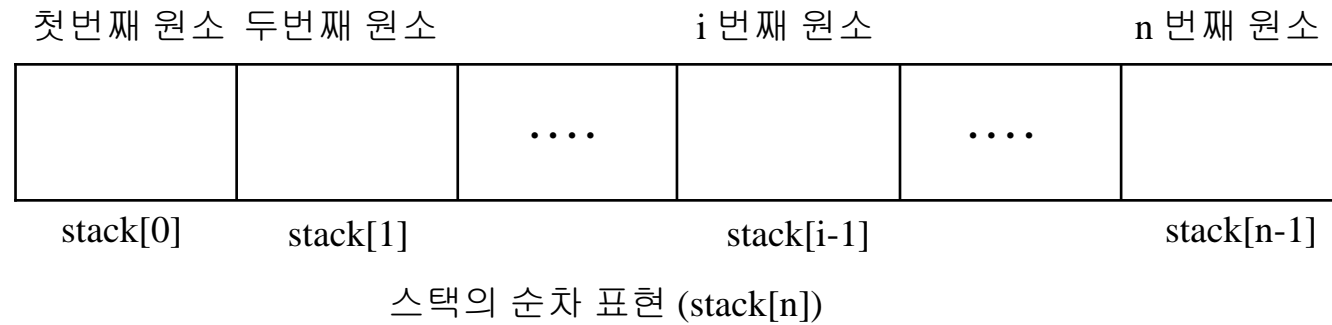
+push(item:E) : E
+empty() : boolean
+pop() : E
+peek() : E

5.2 스택의 순차 표현

5.3 배열을 이용한 스택의 구현

스택의 순차 표현

- ▶ 1차원 배열, `stack[n]`을 이용한 순차 표현
 - 스택을 표현하는 가장 간단한 방법
 - `n`은 스택에 저장할 수 있는 최대 원소 수
 - 스택의 `i` 번째 원소는 `stack[i-1]`에 저장
 - 변수 `top`은 스택의 톱 원소를 가리킴
 - 초기에는 `top = -1`로 설정하여 공백 스택(empty stack)을 표시



배열을 이용한 스택 구현

```
package hufs.dislab.util;

import java.util.EmptyStackException;

public class ArrayStack<E> implements Stack<E> {
    private int top = -1;
    private Object[] arr;

    public ArrayStack() {
        arr = new Object[20];
    }

    @Override
    public boolean empty() {
        return top < 0;
    }
}
```

```
@SuppressWarnings("unchecked")
@Override
public E peek() {
    if (top < 0)
        throw new EmptyStackException();
    return (E)arr[top];
}

@SuppressWarnings("unchecked")
@Override
public E pop() {
    if (top < 0)
        throw new EmptyStackException();
    return (E)arr[top--];
}

@Override
public E push(E item) {
    arr[++top] = item;
    return item;
}
}
```

배열을 이용한 스택 예제

```
package hufs.dislab.util;

import java.util.Date;

public class TestArrayStack {

    public static void main(String[] args) {
        Stack<Object> stack = new ArrayStack<Object>();

        stack.push("Sample");
        stack.push(new Date());

        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}
```

Mon Jan 30 04:28:18 KST 2023
Sample

5.4 복수 스택의 순차 표현

복수 스택의 순차 표현

- ▶ 복수 개의 스택 객체를 생성하여 이용

```
Stack s1, s2, s3;
```

```
s1 = new ArrayStack();
```

```
s2 = new ArrayStack();
```

```
s3 = new ArrayStack();
```

```
s1.push("Sample");
```

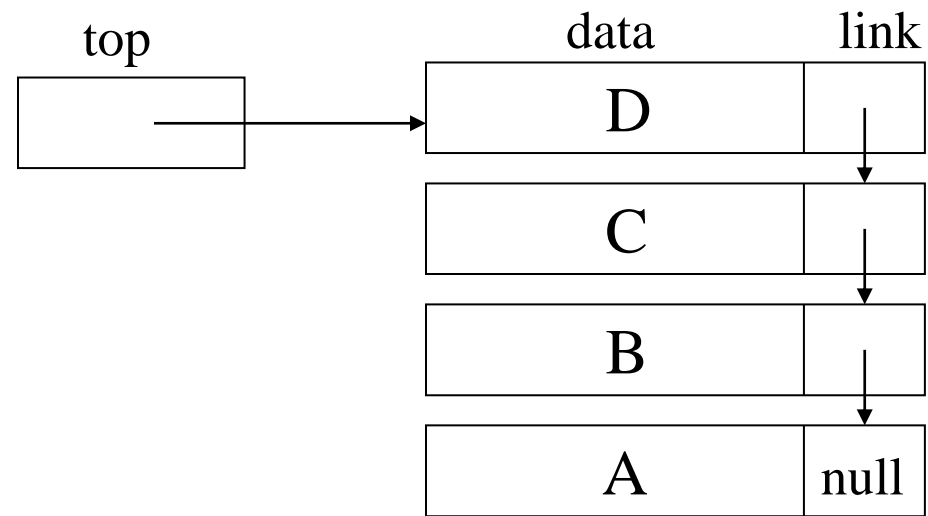
```
s2.push(3);
```

```
s3.push(new Date());
```


5.3 스택의 연결 표현

스택의 연결 표현

- ▶ 연결 리스트로 표현된 연결 스택(linked stack)
 - 삽입, 삭제
 - pop



5.6 리스트를 이용한 스택 구현

연결리스트를 이용한 스택 구현

```
package hufs.dislab.util;

import java.util.EmptyStackException;

public class ListStack<E> implements Stack<E> {

    private LinkedList<E> list = new LinkedList<E>();

    @Override
    public boolean empty() {
        return list.isEmpty();
    }
}
```

```
    @Override
    public E peek() {
        if (list.isEmpty())
            throw new EmptyStackException();
        return list.getFirst();
    }

    @Override
    public E pop() {
        if (list.isEmpty())
            throw new EmptyStackException();
        return list.removeFirst();
    }

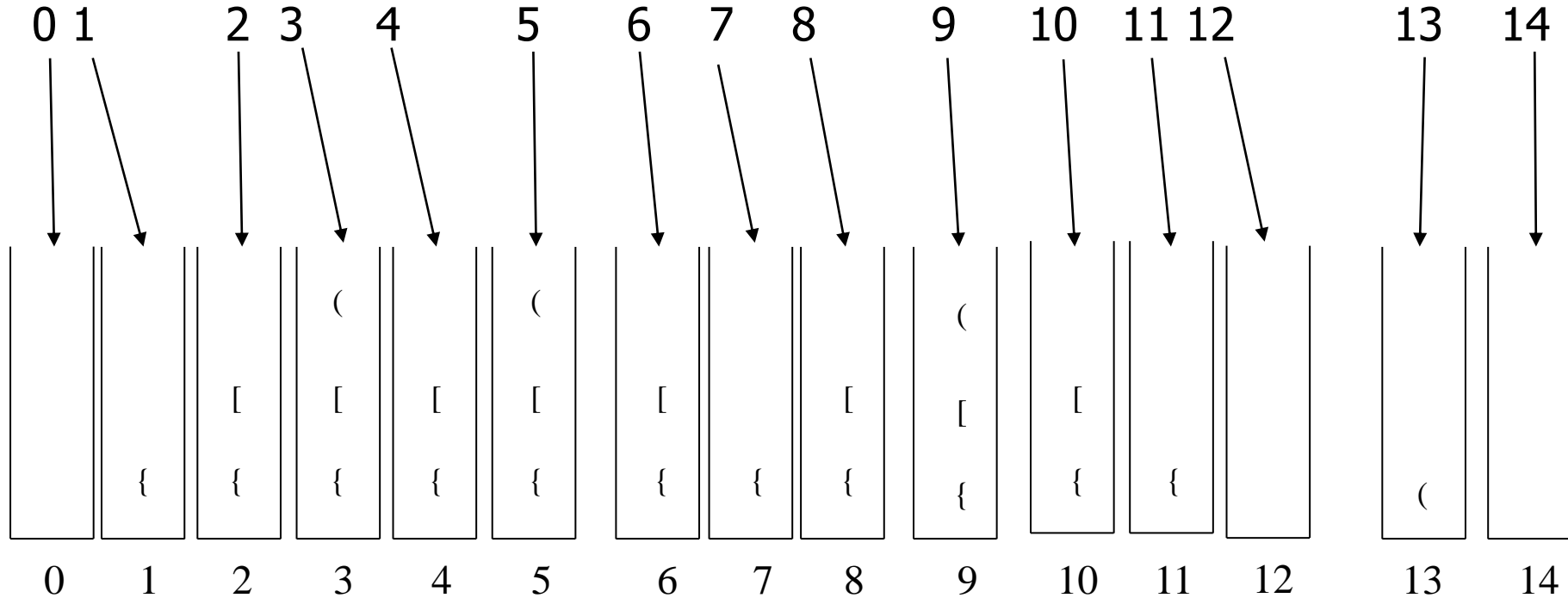
    @Override
    public E push(E item) {
        list.addFirst(item);
        return item;
    }
}
```

5.7 수식의 괄호쌍 검사

수식의 괄호 쌍 검사

- ▶ 수식 : 대괄호, 중괄호, 소괄호를 포함

$$\{ a^2 - [(b+c)^2 - (d+e)^2] * [\sin (x - y)] \} - \cos (x+y)$$



수식의 괄호 쌍 검사

```
package hufs.dislab.util;

import java.util.StringTokenizer;

public class Calculator {

    public boolean parenTest(String exp) {
        StringTokenizer tokenizer =
            new StringTokenizer(exp, "[({})]+-*/ ", true);
        Stack<String> stack = new ListStack<String>();

        while(tokenizer.hasMoreElements()) {
            String token = tokenizer.nextToken();

            switch(token) {
                case "(" :
                case "[" :
                case "{" :
                    stack.push(token);
                    break;
            }
        }
    }
}
```

```
        case ")" : {
            if (stack.empty())
                return false;
            String left = stack.pop();
            if (!"(".equals(left))
                return false;
            break;
        }
        case "]" : {
            String left = stack.pop();
            if (!"[".equals(left))
                return false;
            break;
        }
        case "}" : {
            String left = stack.pop();
            if (!"{".equals(left))
                return false;
            break;
        }
    }
}
return stack.empty();
}
```

괄호 쌍 검사 테스트 프로그램

```
package hufs.dislab.util;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class TestParenTest {
    public static void main(String[] args) throws IOException {
        BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
        String str = r.readLine();

        Calculator calc = new Calculator();
        System.out.println(calc.parenTest(str));
    }
}
```


5.8 스택을 이용한 수식의 계산

스택을 이용한 수식의 연산

- ▶ 수식(expression)
 - 연산자와 피연산자로 구성
 - 피연산자(operand)
 - 변수나 상수
 - 피연산자의 값들은 그 위에 동작하는 연산자와 일치해야 함
 - 연산자(operator)
 - 연산자들 간에는 우선 순위가 존재
 - 값의 타입에 따른 연산자의 분류
 - 기본 산술 연산자 : +, -, *, /
 - 비교 연산자 : <, <=, >, >=, =, !=
 - 논리 연산자 : and, or, not 등
 - 예
 - $A+B*C-D/E$

수식의 표기법

- ▶ 중위 표기법(infix notation)
 - 연산자가 피연산자 가운데 위치
 - 예) $A+B$
- ▶ 전위 표기법(prefix notation)
 - 연산자가 피연산자 앞에 위치
 - 예) $+AB$
- ▶ 후위 표기법(postfix notation)
 - 연산자가 피연산자 뒤에 위치
 - 폴리쉬 표기법(polish notation)
 - 예) $AB+$
 - 장점
 - 연산 순서가 간단 - 왼쪽에서 오른쪽으로 연산자 순서대로 계산
 - 괄호가 불필요

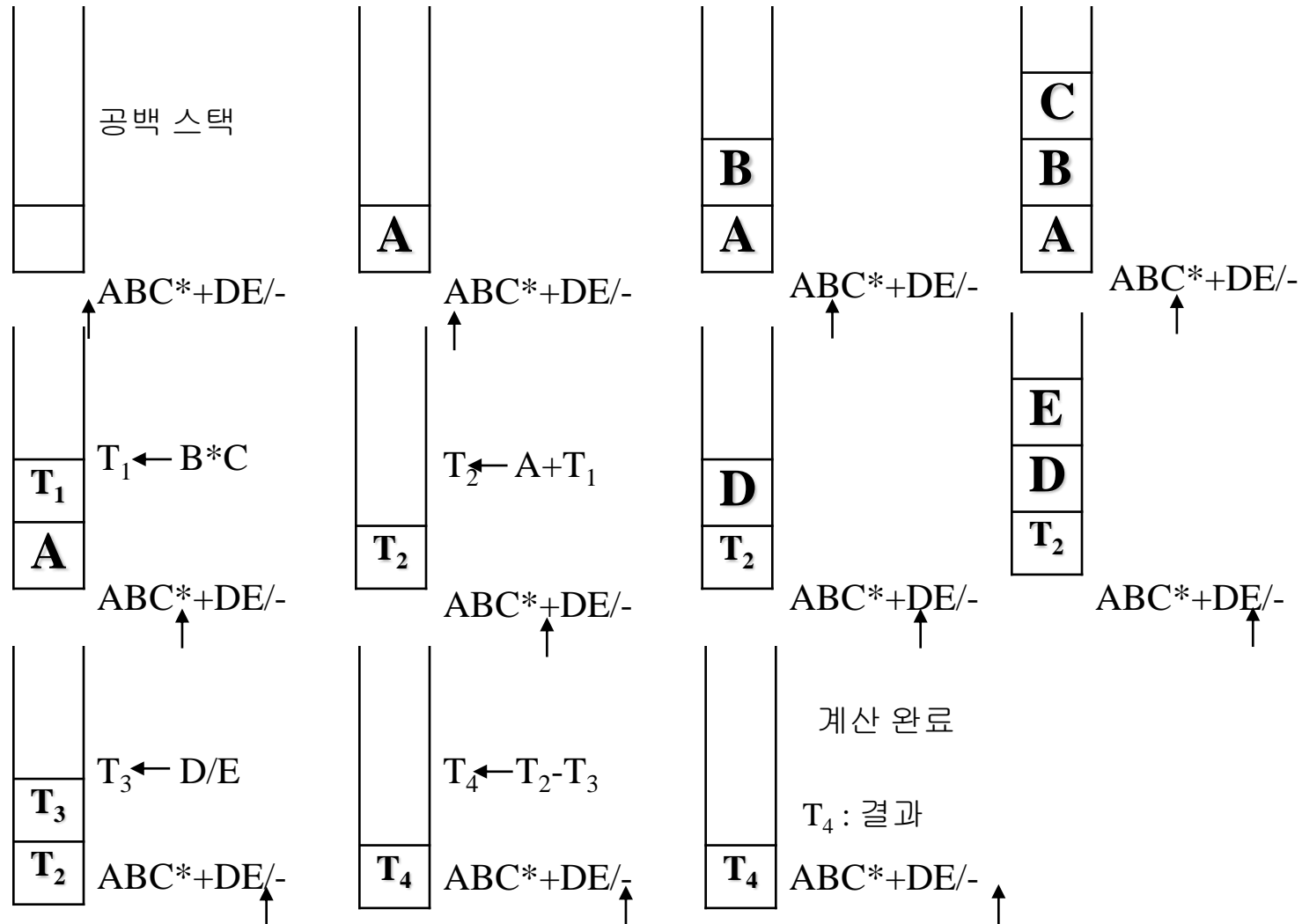
후위 표기식의 연산

- ▶ 후위 표기식: $(ABC^*+DE/-)$ 의 연산

후위 표기식	연산
$ABC^*+DE/-$	$T_1 \leftarrow B * C$
$AT_1+DE/-$	$T_2 \leftarrow A + T_1$
$T_2DE/-$	$T_3 \leftarrow D / E$
T_2T_3-	$T_4 \leftarrow T_2 - T_3$
T_4	

스택을 이용한 후위 표기식의 연산

후위 표기식 : $ABC^*+DE/-$



후위 표기식 연산 알고리즘

```
package hufs.dislab.util;

import java.util.StringTokenizer;

public class Calculator {

    public double evalPostfix(String exp) {
        StringTokenizer tokenizer =
            new StringTokenizer(exp, "[({})]+-*/ ", true);
        Stack<String> stack = new ListStack<String>();

        while(tokenizer.hasMoreElements()) {
            String token = tokenizer.nextToken();

            switch(token) {
                case "+" : {
                    double p2 = Double.parseDouble(stack.pop());
                    double p1 = Double.parseDouble(stack.pop());
                    stack.push(String.valueOf(p1 + p2));
                    break;
                }
                case "-" : {
                    double p2 = Double.parseDouble(stack.pop());
                    double p1 = Double.parseDouble(stack.pop());
                    stack.push(String.valueOf(p1 - p2));
                    break;
                }
            }
        }
    }
}
```

```
        case "*" : {
            double p2 = Double.parseDouble(stack.pop());
            double p1 = Double.parseDouble(stack.pop());
            stack.push(String.valueOf(p1 * p2));
            break;
        }
        case "/" : {
            double p2 = Double.parseDouble(stack.pop());
            double p1 = Double.parseDouble(stack.pop());
            stack.push(String.valueOf(p1 / p2));
            break;
        }
        case " " :
            break;
        default :
            stack.push(token);
            break;
    }
}


return Double.parseDouble(stack.pop());
}
```

중위 표기식을 후위 표기식으로의 변환

▶ 수동 변환 방법

1. 중위 표기식을 완전하게 괄호로 묶는다.
2. 각 연산자를 묶고 있는 괄호의 오른쪽 괄호로 연산자를 이동시킨다.
3. 괄호를 모두 제거한다.

- 피연산자의 순서는 불변
- 연산자의 순서는 우선순위를 반영
- 예

$$((A + (B * C)) - (D / E)) \longrightarrow ABC^* + DE / -$$


스택을 이용한 후위 표기식으로 변환 예

입력(중위 표기식)	토큰	스택	출력(후위 표기식)
$\uparrow A+B*C \infty$			
$A+B*C \infty$	A		A
$\uparrow A+B*C \infty$	+	+	A
$A+B*C \infty$	B	+	AB
$\uparrow A+B*C \infty$	*	+	AB
$A+B*C \infty$	C	+	ABC
$A+B*C \infty$	∞		ABC*+

괄호 처리의 예 (1/2)

입력(중위 표기식)	토큰	스택	출력(후위 표기식)
$A^*(B+C)/D \infty$			
$A^*(B+C)/D \infty$	A		A
$A^*(B+C)/D \infty$	*	*	A
$A^*(B+C)/D \infty$	((*	A
$A^*(B+C)/D \infty$	B	(* B	AB
$A^*(B+C)/D \infty$	+	+ (* B	AB

괄호 처리의 예 (2/2)

입력(중위 표기식)	토큰	스택	출력(후위 표기식)
$A*(B+C)/D \infty$	C	<div> <div></div> <div>+</div> <div>(</div> <div>*</div> </div>	ABC
$A*(B+C)/D \infty$)	<div> <div></div> <div>*</div> </div>	ABC+
$A*(B+C)/D \infty$	/	<div> <div></div> <div>/</div> </div>	ABC+*
$A*(B+C)/D \infty$	D	<div> <div></div> <div>/</div> </div>	ABC+*D
$A*(B+C)/D \infty$	∞	<div> <div></div> </div>	ABC+*D/

후위 표기식으로의 변환 알고리즘

```
package hufs.dislab.util;

import java.util.HashMap;
import java.util.Map;
import java.util.StringTokenizer;

public class Calculator {

    static final Map<String, Integer> PIS;
    static final Map<String, Integer> PIE;

    static {
        PIS = new HashMap<String, Integer>();
        PIS.put("^", 3);
        PIS.put("*", 2);
        PIS.put("/", 2);
        PIS.put("+", 1);
        PIS.put("-", 1);
        PIS.put("(", 0);

        PIE = new HashMap<String, Integer>();
        PIE.put("^", 3);
        PIE.put("*", 2);
        PIE.put("/", 2);
        PIE.put("+", 1);
        PIE.put("-", 1);
        PIE.put("(", 4);
    }
}
```

연산자	PIS	PIE
)	-	-
^	3	3
*, /	2	2
+, -	1	1
(0	4

```
public String postfix(String exp) {
    StringTokenizer tokenizer = new StringTokenizer(exp, "[({})]+-*/ ", true);
    Stack<String> stack = new ListStack<String>();
    StringBuffer str = new StringBuffer();

    while(tokenizer.hasMoreElements()) {
        String token = tokenizer.nextToken();

        switch(token) {
            case ")" : {
                while(!stack.empty() && !stack.peek().equals("("))
                    str.append(stack.pop()).append(" ");
                stack.pop();
                break;
            }
            case "(" :
            case "+" :
            case "-" :
            case "*" :
            case "/" :
            case "^" : {
                while(!stack.empty() && PIS.get(stack.peek()) >= PIE.get(token))
                    str.append(stack.pop()).append(" ");
                stack.push(token);
                break;
            }
            case " " :
                break;
            default :
                str.append(token).append(" ");
                break;
        }
    }

    while(!stack.empty())
        str.append(stack.pop()).append(" ");
    return str.toString();
}
```

중위식 계산 예제

```
package hufs.dislab.util;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class TestCalculator {
    public static void main(String[] args) throws IOException {
        BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
        String str = r.readLine();

        Calculator calc = new Calculator();
        String exp = calc.postfix(str);
        System.out.println(exp);

        double result = calc.evalPostfix(exp);
        System.out.println(result);
    }

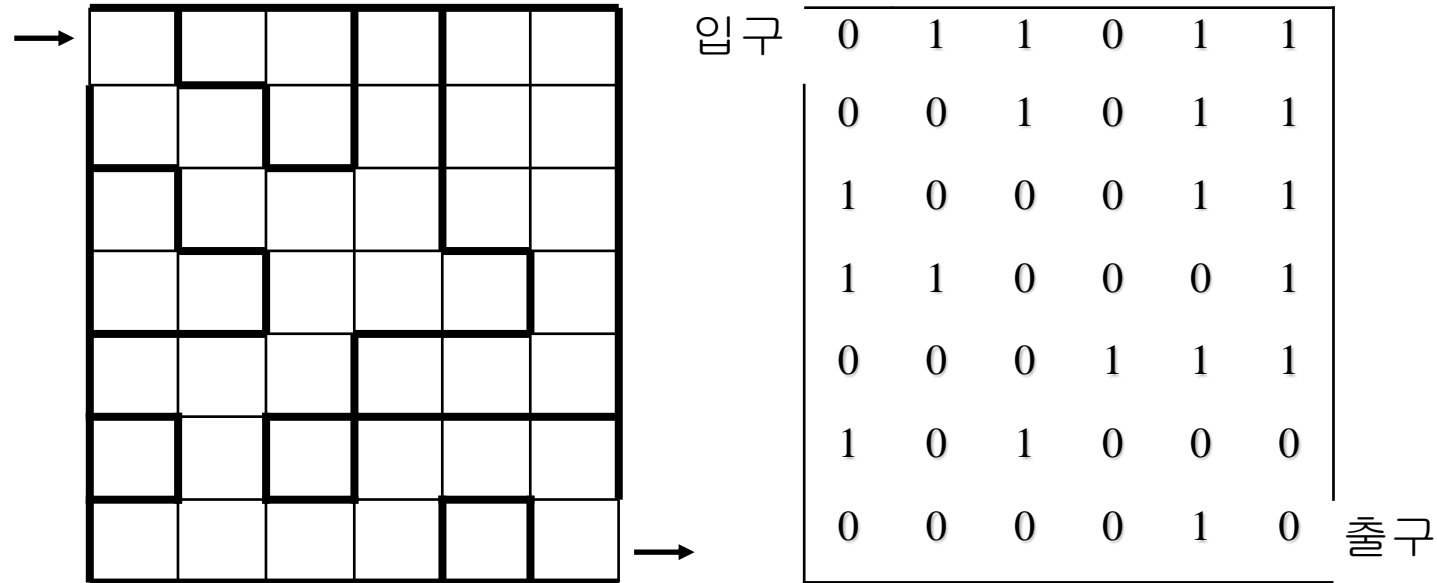
    ...
}
```

5.9 미로 문제

미로 문제 (1)

▶ 미로

◦ 예

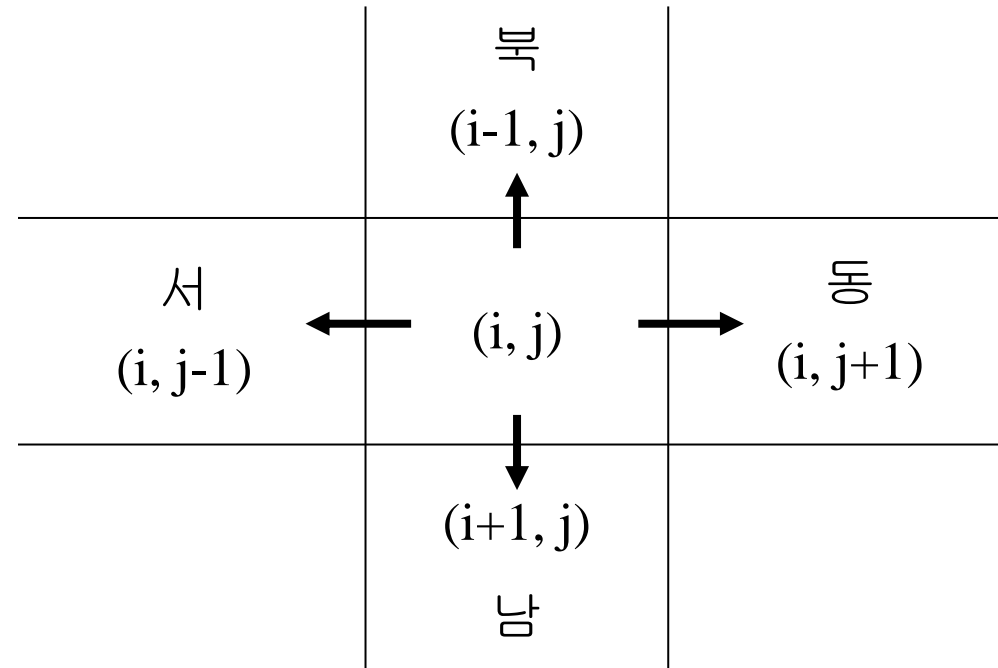


▶ $m \times n$ 미로를 $\text{maze}(m+2, n+2)$ 배열로 표현

- 원소의 값은 0과 1: 0은 이동 가능, 1은 막혀있는 벽
- 사방을 1로 둘러싸서 경계 위치에 있을 때의 예외성(두 방향만 존재)을 제거

미로 문제 (2)

- ▶ 현재 위치 : $\text{maze}[i, j]$
- ▶ 이동 방향
 - 북, 동, 남, 서 순서 (시계 방향)



미로문제 (3)

- ▶ 이동 방향 배열 : `move[4, 2]`

(dir)	i [0]	j [1]
북[0]	-1	0
동[1]	0	1
남[2]	1	0
서[3]	0	-1

- ▶ 다음 위치계산 : `maze[next_i,next_j]`
 - $\text{next_i} \leftarrow i + \text{move}[\text{dir}, 0]$
 - $\text{next_j} \leftarrow j + \text{move}[\text{dir}, 1]$
- ▶ 방문한 경로를 `mark[m+2, n+2]`에 표시
 - 한 번 시도했던 위치로는 다시 이동하지 않음
 - 초기에는 0, 방문한 위치는 1
- ▶ 지나온 경로 $\langle i, j, \text{dir} \rangle$ 을 스택에 저장
 - 스택의 최대 크기 : $m * n$

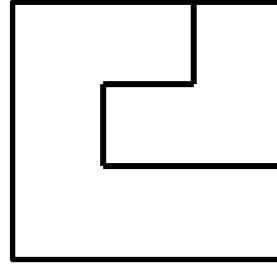

```

mazePath( )
  maze[m + 2, n + 2];
  mark[m + 2, n + 2];
  stack = new Stack(m×n);    // top ← -1;
  maze[1,1] ← 1;
  stack.push(<1, 1, 1>);
  while (not isEmpty(stack)) do {
    <i, j, dir> ← stack.pop();

    while (dir ≤ 3) do {
      next_i ← i + move[dir, 0];
      next_j ← j + move[dir, 1];
      if (next_i = m and next_j = n) then {
        print("The path is as follows.");
        print(path in stack);
        print(i, j);
        print(m, n);
        return;
      }

      if (maze[next_i, next_j] = 0 and mark[next_i, next_j] = 0) then {
        mark[next_i, next_j] ← 1;
        stack.push(<i, j, dir>);
        <i, j, dir> ← <next_i, next_j, 0>;
      }
      else dir ← dir + 1;
    }
  }
  print("There is no path");
end mazePath()

```



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

maze

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

mark

(dir)	[0]	[1]
북[0]	-1	0
동[1]	0	1
남[2]	1	0
서[3]	0	-1