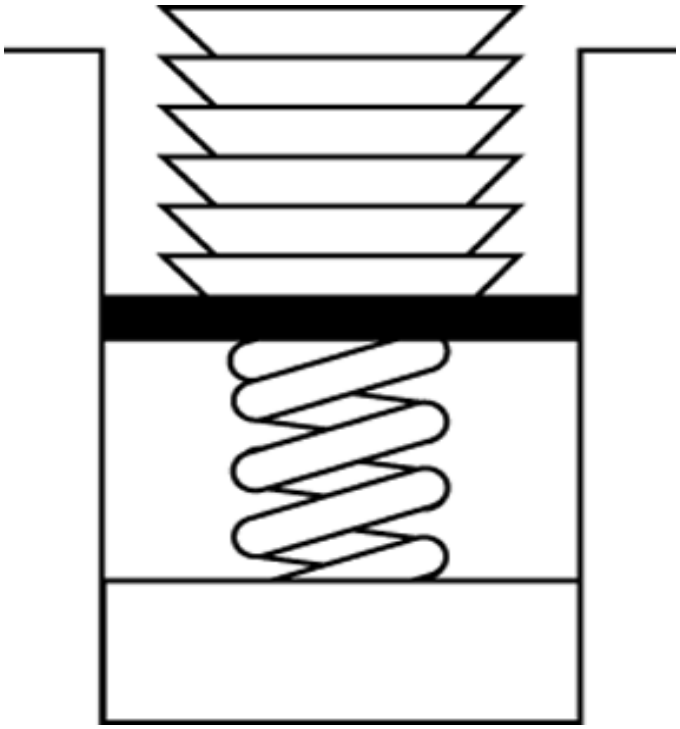
# Ch. 6 Stack

- 도입을 위한 예
  - "←" in keyboard input line
  - E.g., abcd←←efgh←←←ij←km←
    - ➢ abeik

- 한 character를 읽어 '←' 이 아니면 저장하고 '←' 이면 **최근에 저장된** character를 제거한다.

# A Stack Example



최근에 쌓은 접시를 꺼낸다

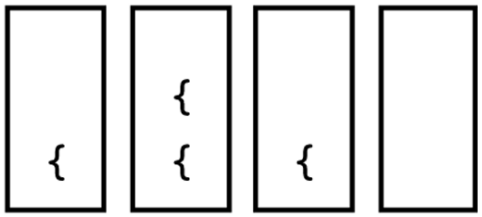
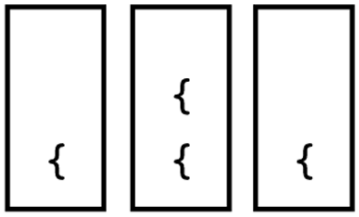
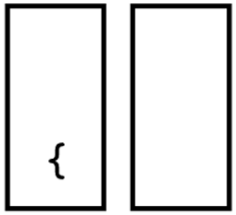
Stack of cafeteria dishes

# ADT *Stack* Operations

- Create an empty stack
- Determine whether a stack is empty
- Add a new item on top of the stack
- Remove from the stack the most-recently added item
- Retrieve from the stack the most-recently added item
- Remove all the items from the stack

- Notable fact
  - The only access to the stack is the most-recently added item

# Example: Checking Balance of Braces

| Input string | Stack as algorithm executes | |
|---|---|---|
| | 1. 2. 3. 4. | |
| {a{b}c} |  | 1. push "{"<br>2. push "{"<br>3. pop<br>4. pop<br>Stack empty ⟹ balanced |
| {a{bc} |  | 1. push "{"<br>2. push "{"<br>3. pop<br>Stack not empty ⟹ not balanced |
| {ab}c} |  | 1. push "{"<br>2. pop<br>Stack empty when last "}" encountered ⟹ not balanced |

# Example: Checking Symmetric Language

$L = \{w\$w' : w$ is a possibly empty string w/o \$, $w'$ = reverse($w$) }

```
do {
        ch = string.getNextCharacter( );
        stack.push(ch);
} while (ch is not '$')
stack.pop( );
do {
        if (string.noMoreCharacter( )) {
                if (stack.isEmpty( )) return true;
                else return false;
        } else {
                if (stack.isEmpty( )) return false;
                stackTop = stack.pop( );
                ch = string.getNextCharacter( );
        }
} while (ch == stackTop)
return false;
```

} 설명의 편의상 error 처리 생략

← 의미는 느낌으로 받아들일 것

# Stack Implementation

```
public interface StackInterface {

        public boolean isEmpty( );
        public void push(Object newItem);
        public Object pop( );
        public void popAll ( );
        public Object peek( );
}
```

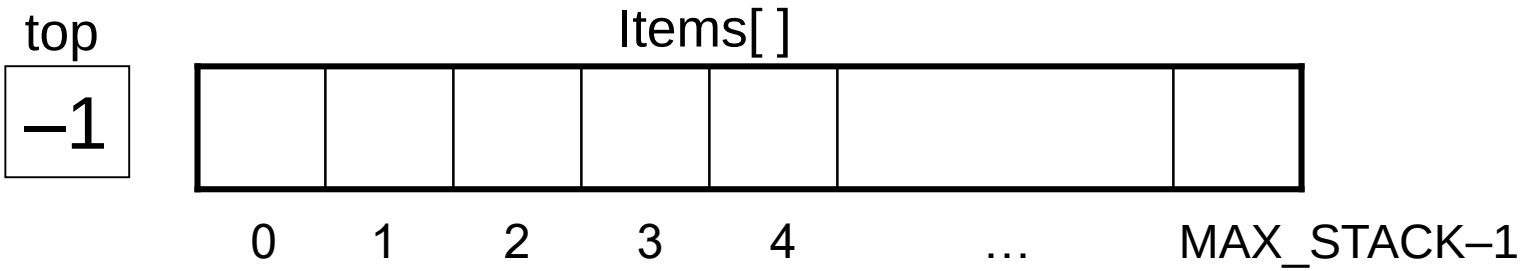
# 세가지 구현

- Array Based
- Reference Based
- Reusing List

✓ Interface는 전혀 변하지 않는다

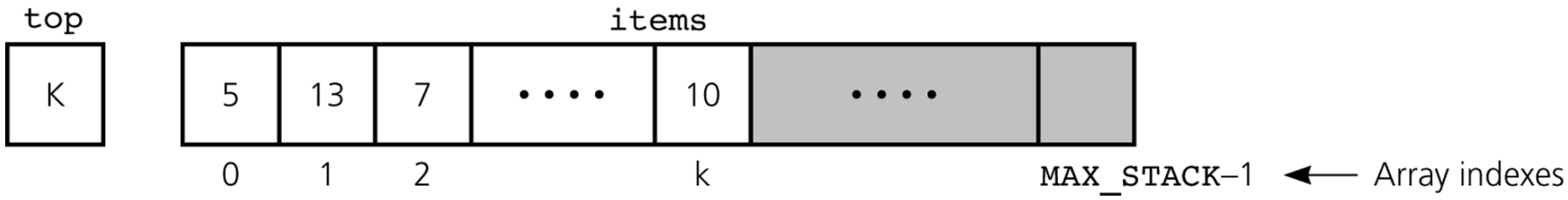
```
public interface StackInterface {
        public boolean isEmpty( );
        public void push(Object newItem);
        public Object pop( );
        public void popAll ( );
        public Object peek( );
}
```

# Array-Based Implementation

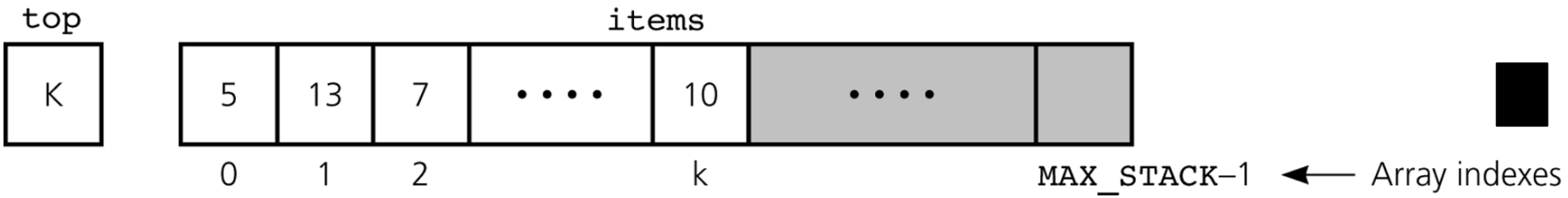



```
public class StackArrayBased implements StackInterface {

        final int MAX_STACK = 50;
        private Object items[ ];
        private int top;  // index for stack top

        public StackArrayBased( ) {
                items = new Object[MAX_STACK];
                top = –1;
        }
```

```
public boolean isEmpty( ) {
        return (top < 0);
}
public boolean isFull ( ) {
        return (top ==  MAX_STACK–1);
}
public void push(Object newItem) {
        if (!isFull( )) items[++top] = newItem;
        else {exception 처리}
}
```

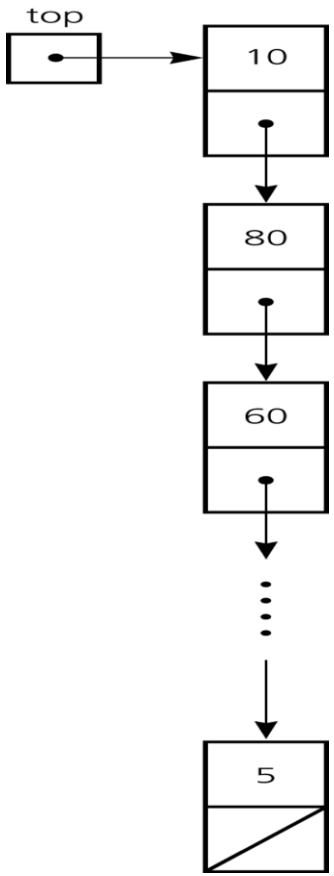


```
        public Object pop( ) {
                if (!isEmpty( )) return items[top--];
                else {exception 처리}
        }
        public void popAll( ) {
                items = new Object[MAX_STACK];
                top = –1;
        }
        public Object peek( ) {
                if (!isEmpty( )) return items[top];
                else {exception 처리}
        }
} // end class  StackArrayBased
```



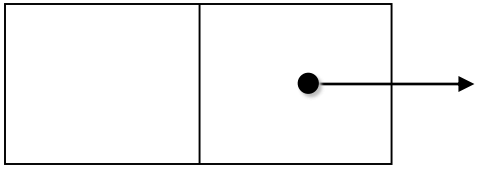
# Reference-Based Implementation

```
public class StackReferenceBased implements StackInterface{
        private Node top;

        public StackReferenceBased( ) {
                top = null;
        }
```



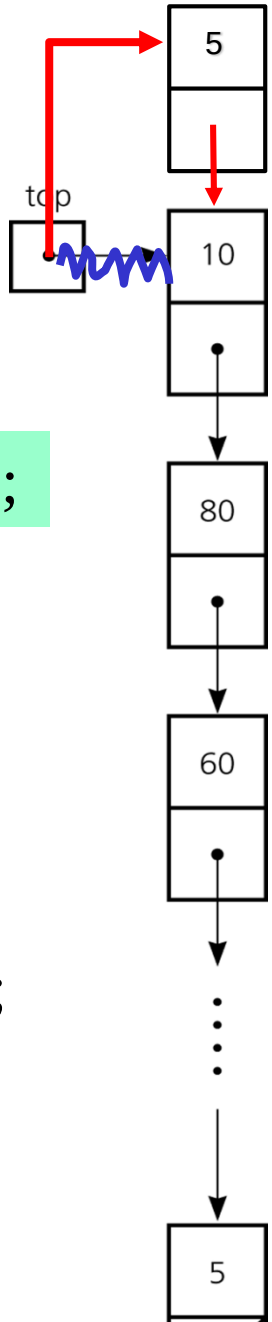

✓ Class *Node*: Ch.4, 강의 노트 P.8 참조

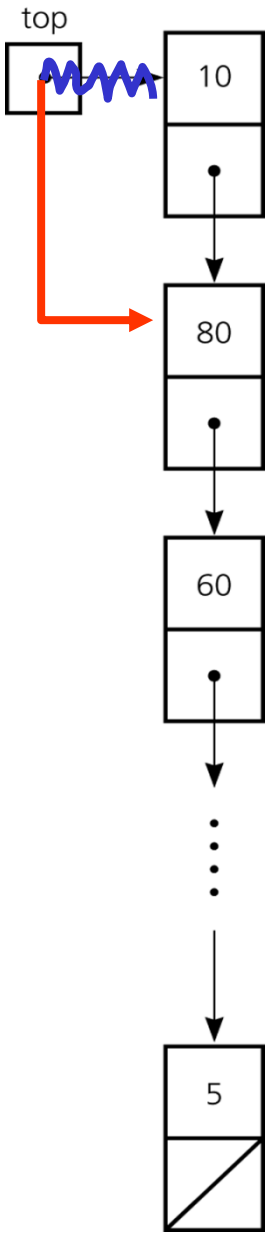
# Remind: class Node (Ch. 4)



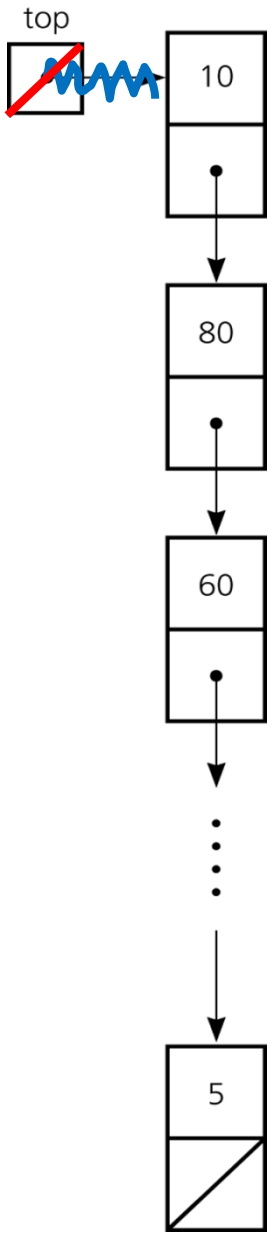
```
public class Node {
        private Object item;
        private Node next;
        public Node(Object newItem) {
                item = newItem;
                next = null;
        }
        public Node(Object newItem, Node nextNode) {
                item = newItem;
                next = nextNode;
        }
        public Object getItem( ) {
                return item;
        }
        // setItem, setNext, getNext
        …
}
```

```
public boolean isEmpty( ) {
        return (top == null);
}
public void push(Object newItem) {
        top = new Node(newItem, top);
}
public Object pop( ) {
        if (!isEmpty( )) {
                Node temp = top;
                top = top.getNext( );
                return temp.getItem( );
        } else {exception 처리}
}
```



```
        public void popAll ( ) {
                top = null;
        }
        public Object peek( ) {
                if (!isEmpty( )) return top.getItem( );
                else {exception 처리}
        }
} // end class  StackReferenceBased
```



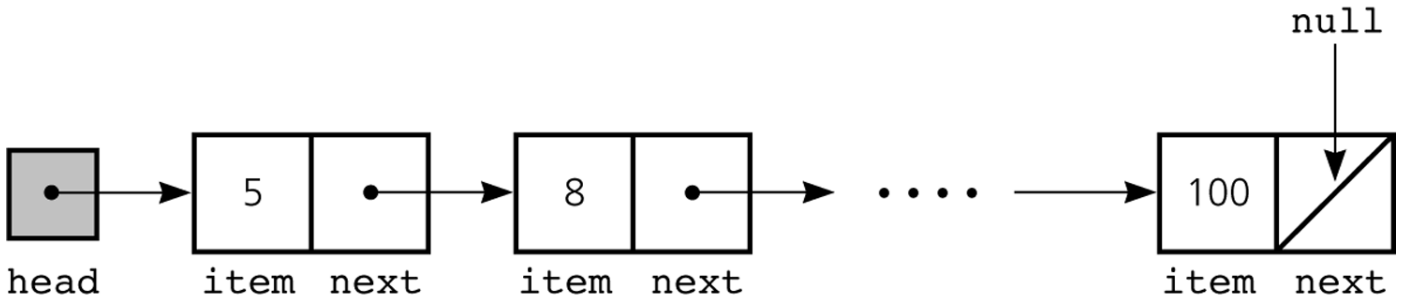
# ADT List-Based Implementation

```
public class StackListBased implements StackInterface {
        private ListInterface list;

        public StackListBased( ) {
                list = new ListReferenceBased( );
        }
```

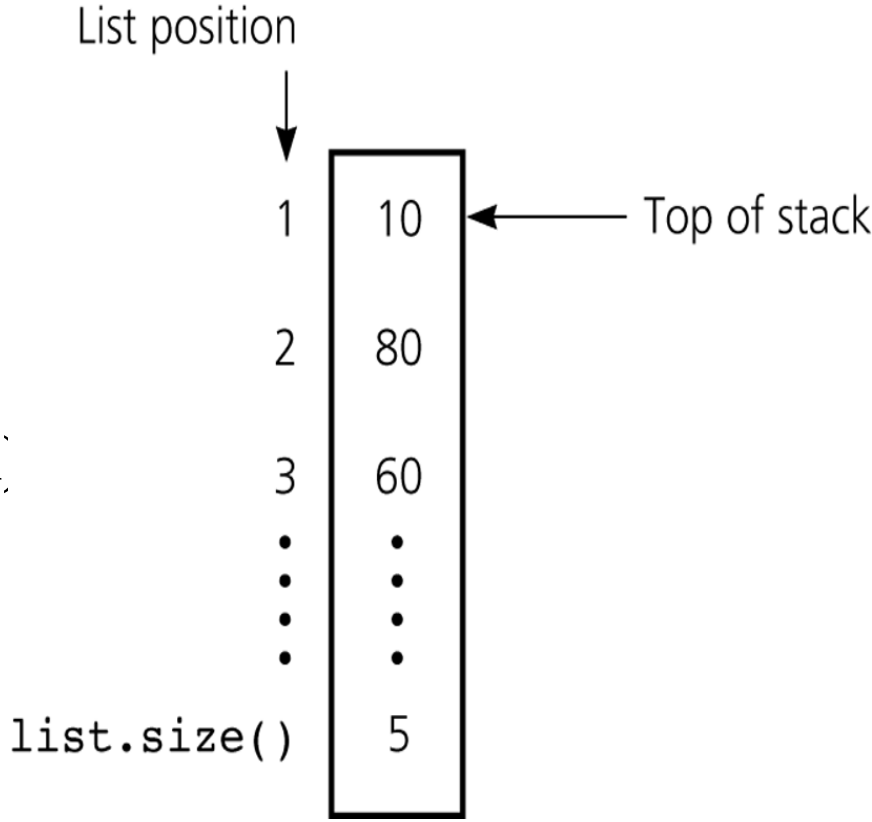
✓Class *ListReferenceBased*: Ch.4, 강의 노트 P.13 참조

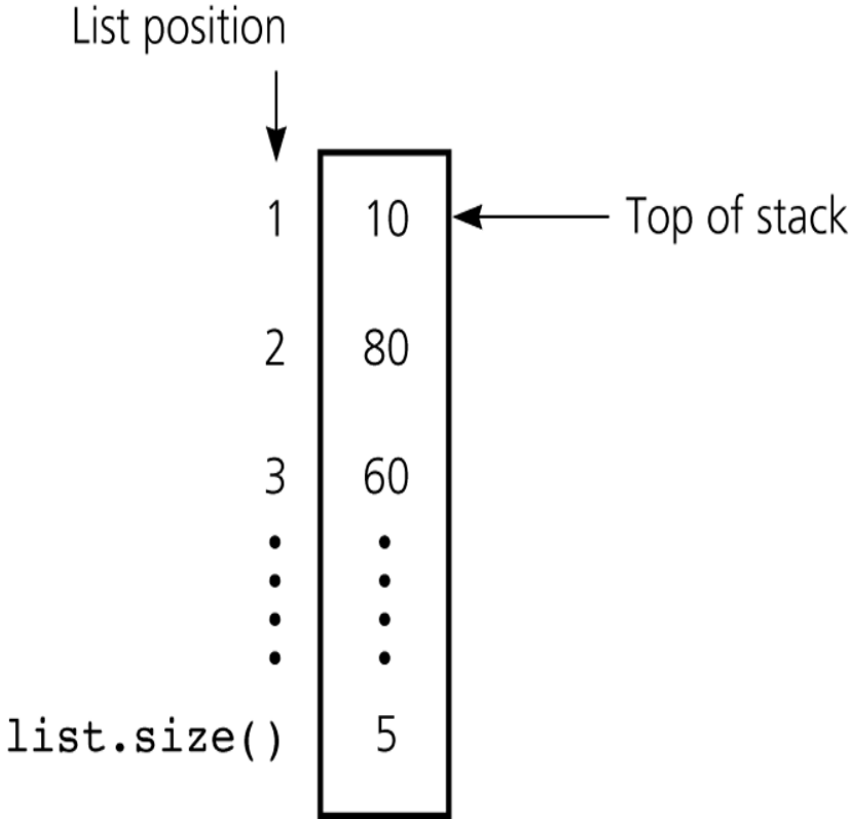
# Remind: class ListReferenceBased (Ch. 4)

```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        // constructor
        public ListReferenceBased( ) {
                numItems = 0;
                head =  null;
        }
        // operations
         public Object get(int index) {
        …
```



```
public boolean isEmpty( ) {
        return list.isEmpty( );
}
public void push(Object newItem)
        list.add(1, newItem);
}
public Object pop( ) {
        if (!list.isEmpty( )) {
                Object temp = list.get(1);
                list.remove(1);
                return temp;
        } else {exception 처리}
}
```






```
        public void popAll ( ) {
                list.removeAll( );
        }
        public Object peek( ) {
                if (!isEmpty( )) return list.get(1);
                else {exception 처리}
        }
} // end class  StackListBased
```

- ✓ “ADT list”-based version
  reuses the class ListReferenceBased.
- ✓ The reuse made the code simple.

- ✓ Reuse 비교

isEmpty( );
push(Object newItem);
Object pop( );
popAll ( );
peek( );

```
public Object pop( ) {
        if (!list.isEmpty( )) {
                Object temp = list.get(1);
                list.remove(1);
                return temp;
        } else {exception 처리}
}
```

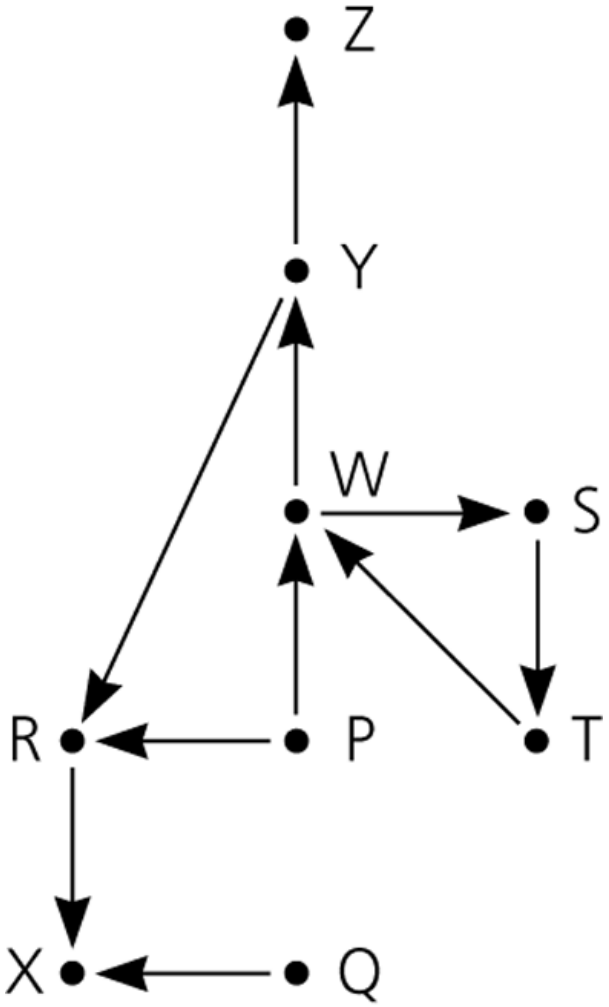
# Evaluating Postfix Expressions

Postfix expression: 2 3 4 + *

| Key entered | Calculator action | | Stack (bottom to top) |
|---|---|---|---|
| 2 | push 2 | | 2 |
| 3 | push 3 | | 2 3 |
| 4 | push 4 | | 2 3 4 |
| + | operand2 = pop stack | (4) | 2 3 |
| | operand1 = pop stack | (3) | 2 |
| | result = operand1 + operand2 | (7) | 2 |
| | push result | | 2 7 |
| * | operand2 = pop stack | (7) | 2 |
| | operand1 = pop stack | (2) | |
| | result = operand1 * operand2 | (14) | |
| | push result | | 14 |

```
for (each character ch in the expression) {
        if (ch is an operand)
                stack.push(ch);
        else { // ch is an operator named op
                operand2 = stack.pop( );
                operand1 = stack.pop( );
                result = operarand1 op operand2;
                stack.push(result);
        }
}
```

# A Search Problem in Flight Map




↑ : direct flight

Is there a path
from city A to city B?

```
searchS(originCity, destinationCity)
{       // All cities are marked UNVISITED in the beginning
        stack.push(originCity);
        mark[originCity] = VISITED;

        while (!stack.isEmpty( ) && destinationCity is not at the top of stack) {
                if (no flight exists from the stack-top city to unvisited cities)
                        temp ← stack.pop( );  // backtracking
                else {
                        Select an unvisited city C that is directly reachable
                                                        from the stack-top city;
                        stack.push(C);
                        mark[C] ← VISITED;
                }
        }

        if (stack.isEmpty( ))  // There is no path
                return false;
        else // A path found
                return true;
}
```

## Equivalently

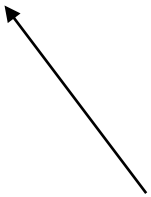
```
searchS(originCity, destinationCity)
{       // All cities are marked UNVISITED in the beginning
        stack.push(originCity);
        mark[originCity] = VISITED;

        while (!stack.isEmpty( )) {
                if (no flight exists from the stack-top city to unvisited cities)
                        temp ← stack.pop( );  // backtracking
                else {
                        Select an unvisited city C that is directly reachable
                                                        from the stack-top city;
                        if (C = destinationCity) return true;
                        else {
                                stack.push(C);
                                mark[C] ← VISITED;
                        }
                }
        }

        // There is no path
        return false;
}
```

# Recursive version

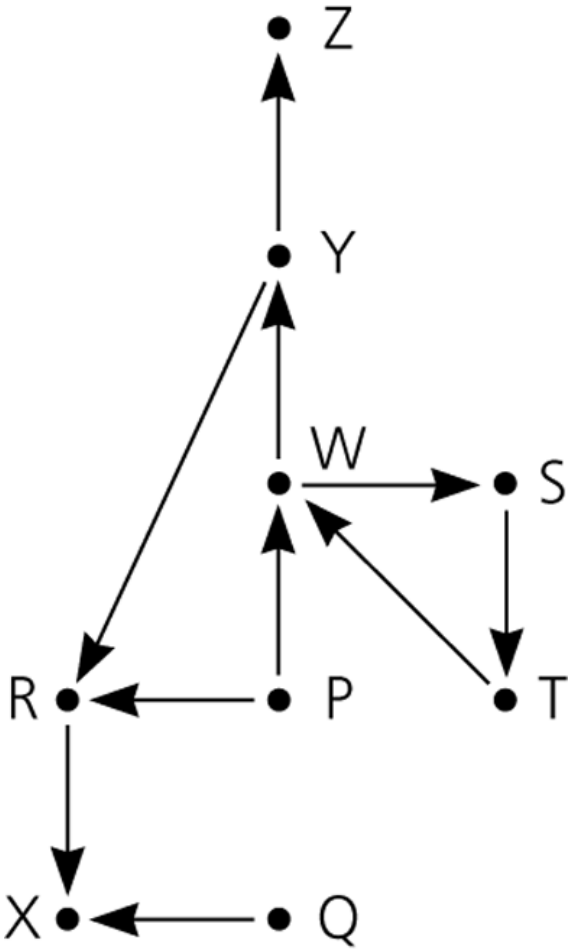
```
searchS(originCity, destinationCity)
{
    if (originCity = destinationCity) return true;
    else {
        mark[originCity] ← VISITED;
        arrived ← false;
        for each x ∈ L(originCity)      ▷ L(originCity) : originCity에 인접한 도시들
            if (mark[x] = UNVISITED) {
                arrived ← searchS(x , destinationCity);
                if (arrived= true) return true;
            }
        return false;
    }
}
```

# Equivalently

```
searchS(originCity, destinationCity)
{
    if (originCity = destinationCity) return true
    else {
        mark[originCity] ← VISITED;
        for each x ∈ L(originCity) {   ▷ L(originCity) : originCity에 인접한 도시들
            if (mark[x] = UNVISITED and searchS(x , destinationCity) = true)
                return true;
        }
        return false;
    }
}
```

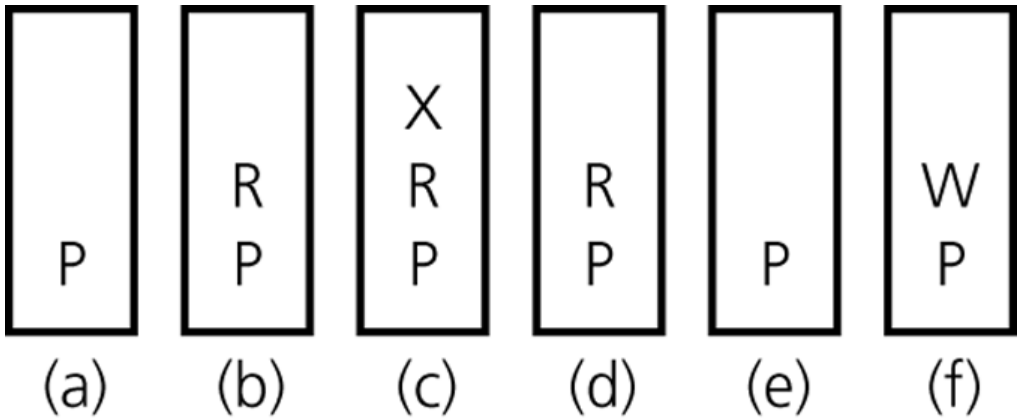


나중에 배울 DFS( )와 유사



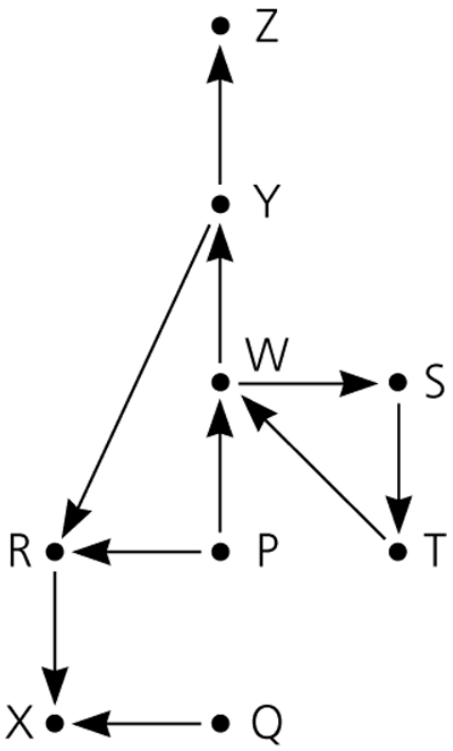

**The stack of cities as you travel**

a) from *P*
b) to *R*
c) to *X*
d) back to *R*
e) Back to *P*
f) to *W*

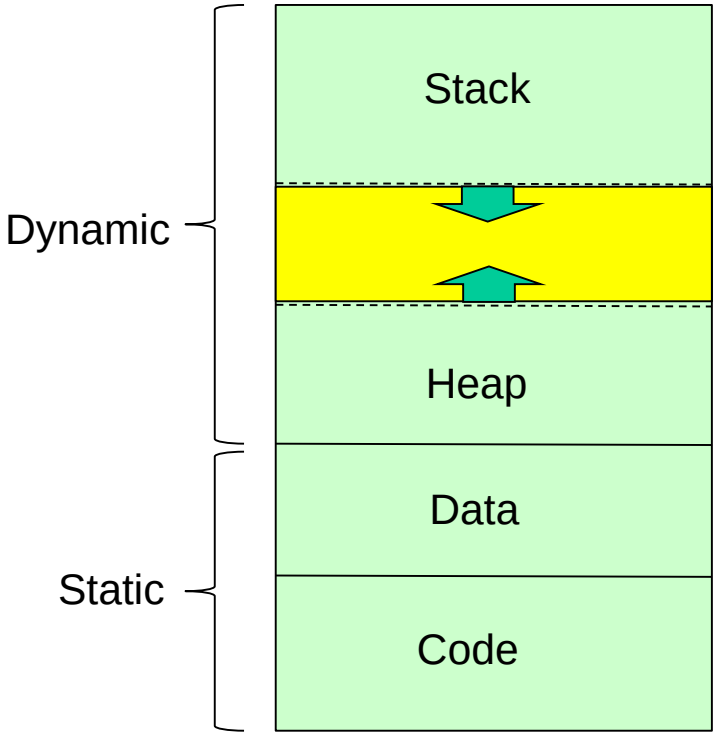


# A trace of the search algorithm, given the flight map

| Action | Reason | Contents of stack (bottom to top) |
|---|---|---|
| Push P | Initialize | P |
| Push R | Next unvisited adjacent city | P R |
| Push X | Next unvisited adjacent city | P R X |
| Pop X | No unvisited adjacent city | P R |
| Pop R | No unvisited adjacent city | P |
| Push W | Next unvisited adjacent city | P W |
| Push S | Next unvisited adjacent city | P W S |
| Push T | Next unvisited adjacent city | P W S T |
| Pop T | No unvisited adjacent city | P W S |
| Pop S | No unvisited adjacent city | P W |
| Push Y | Next unvisited adjacent city | P W Y |
| Push Z | Next unvisited adjacent city | P W Y Z |



# Memory 분할과 Stack




active 상태의 함수 관련 parameter와 variable들

수행중에 사용자가 할당받는 메모리.
예; Java의 new, C의 malloc()

program이 끝날 때까지 없어지지 않는 데이터.
Global, static variables

Program

Virtual Memory
(OS라는 벽을 통해 보는 메모리)