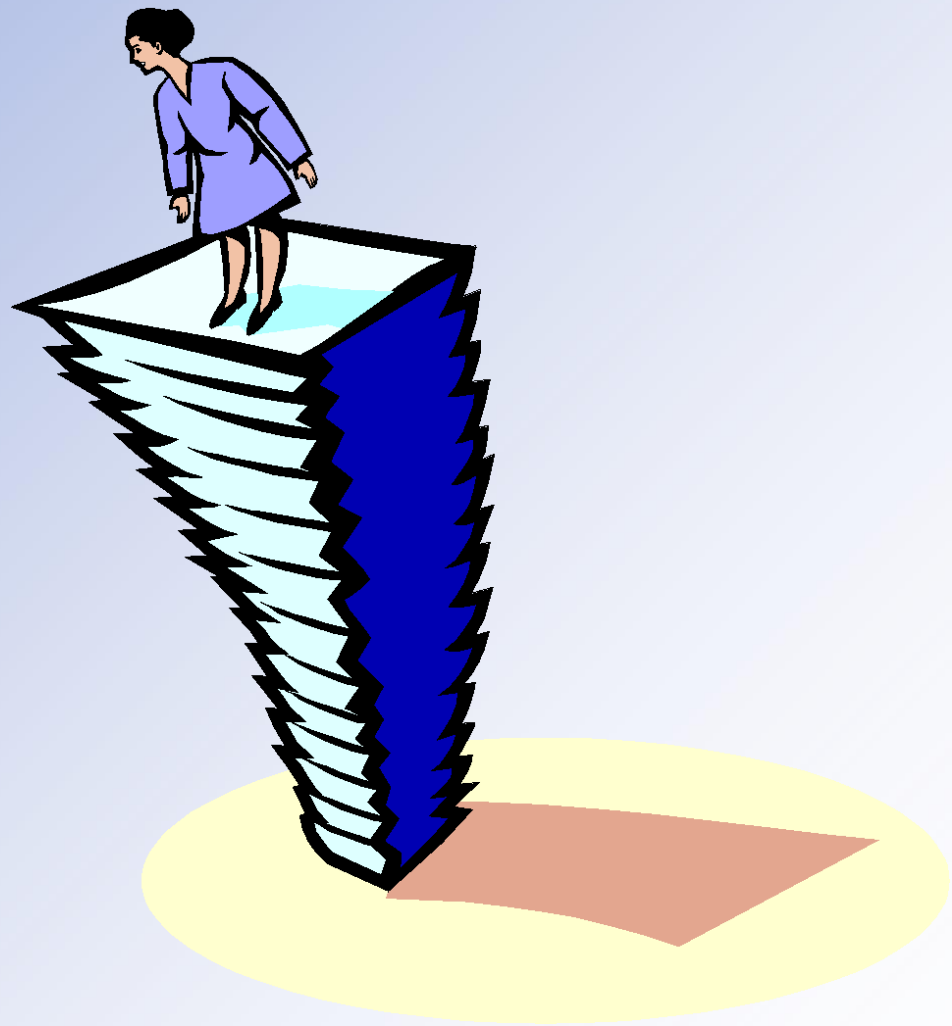
Stacks



Introduction to Stacks

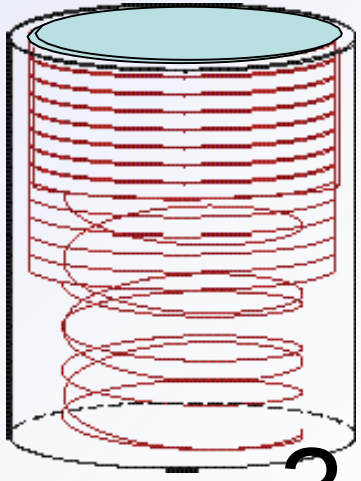
- Consider a card game with a discard pile
  - Discards always placed on the top of the pile
  - Players may retrieve a card only from the top

What other examples can you think of that are modeled by a stack?

- We seek a way to represent and manipulate this in a computer program
- This is a stack

Introduction to Stacks

- A stack is a last-in-first-out (LIFO) data structure
- Adding an item
  - Referred to as pushing it onto the stack
- Removing an item
  - Referred to as popping it from the stack



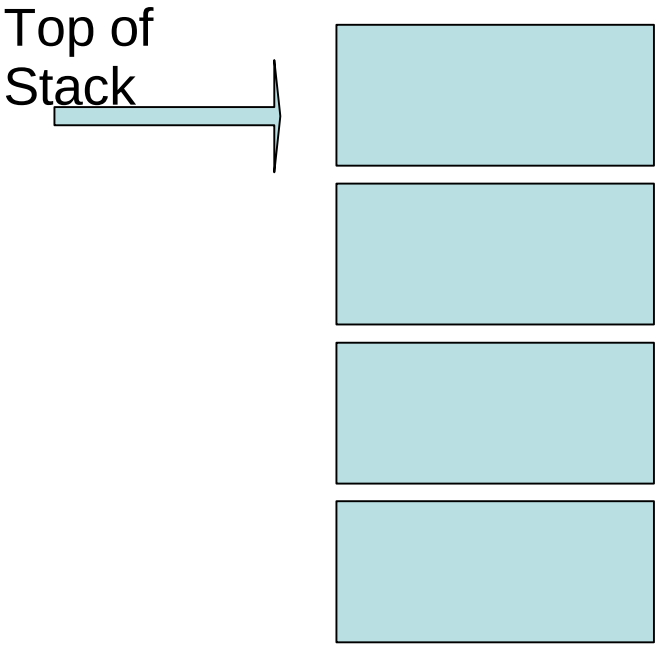
A Stack

- Definition:
  - An ordered collection of data items
  - Can be accessed at only one end (the top)
- Operations:
  - construct a stack (usually empty)
  - check if it is empty
  - Push: add an element to the top
  - Top: retrieve the top element
  - Pop: remove the top element

A Conceptual View of a Stack

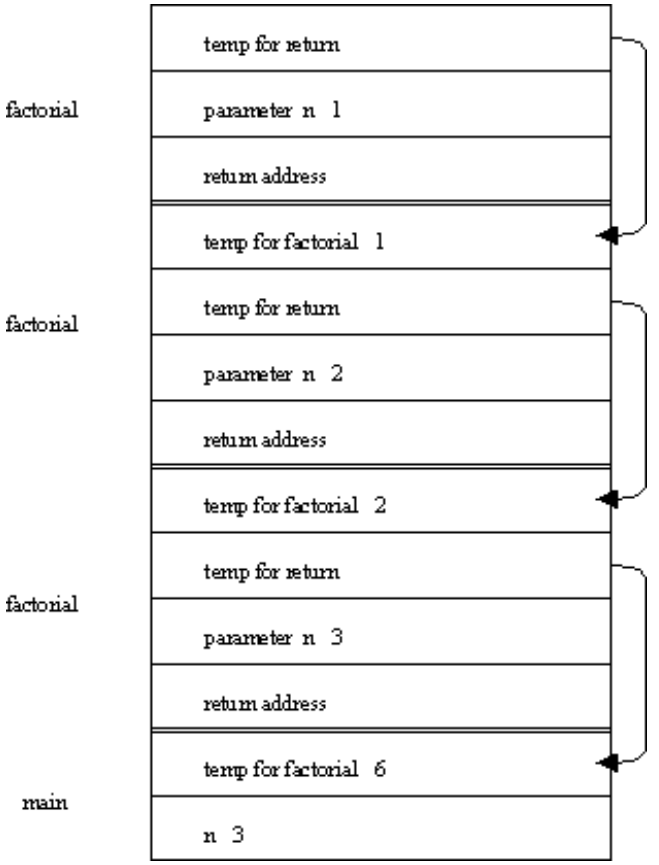
Adding an Element

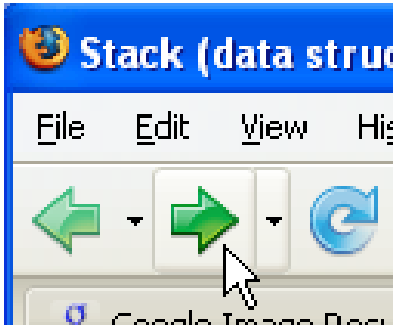
Removing an Element

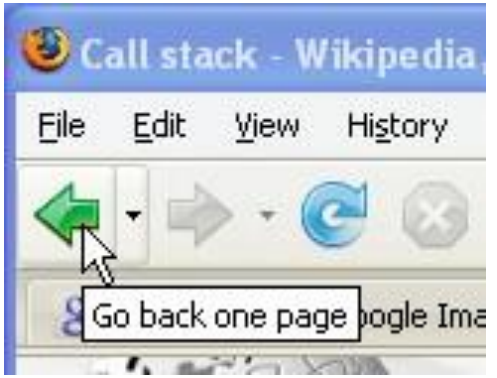


Uses of Stacks

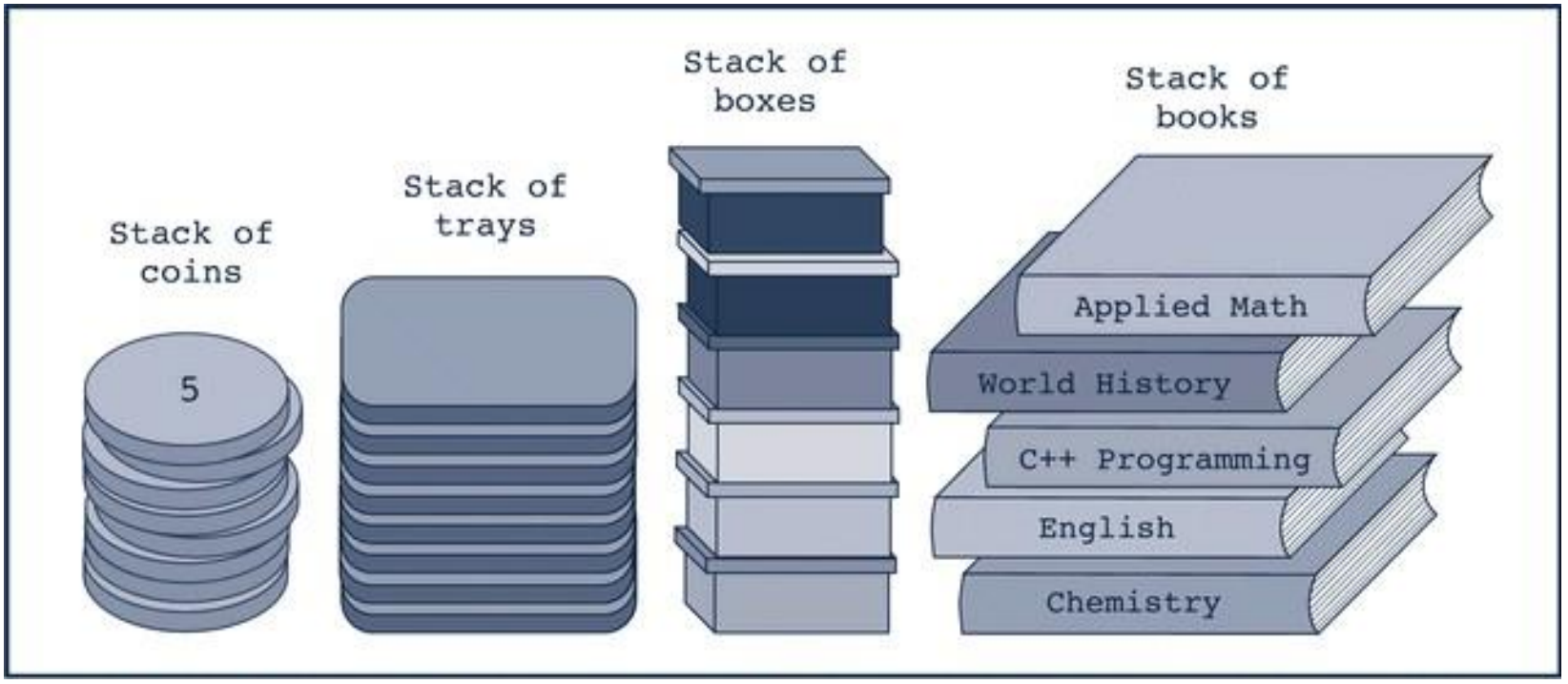
- The runtime stack used by a process (running program) to keep track of methods in progress
- Search problems
- Undo, redo, back, forward







Examples


Stack of coins
5
Stack of trays
Stack of boxes
Stack of books
Applied Math
World History
C++ Programming
English
Chemistry

Basic Operations on a Stack

isFullStack: Checks whether the stack is full. If full, it returns true; otherwise, it returns false

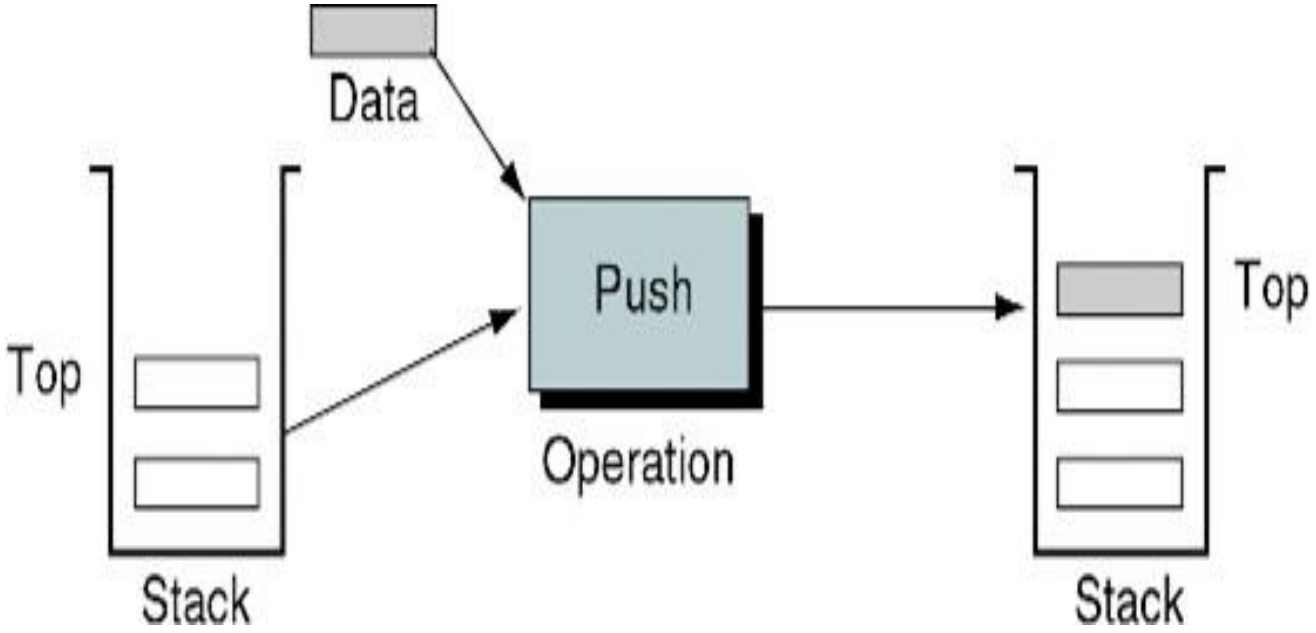
push:

Add new element to the top of the stack

The input consists of the stack and the new element.

Prior to this operation, the stack must exist and must not be full

Stack Push Operation

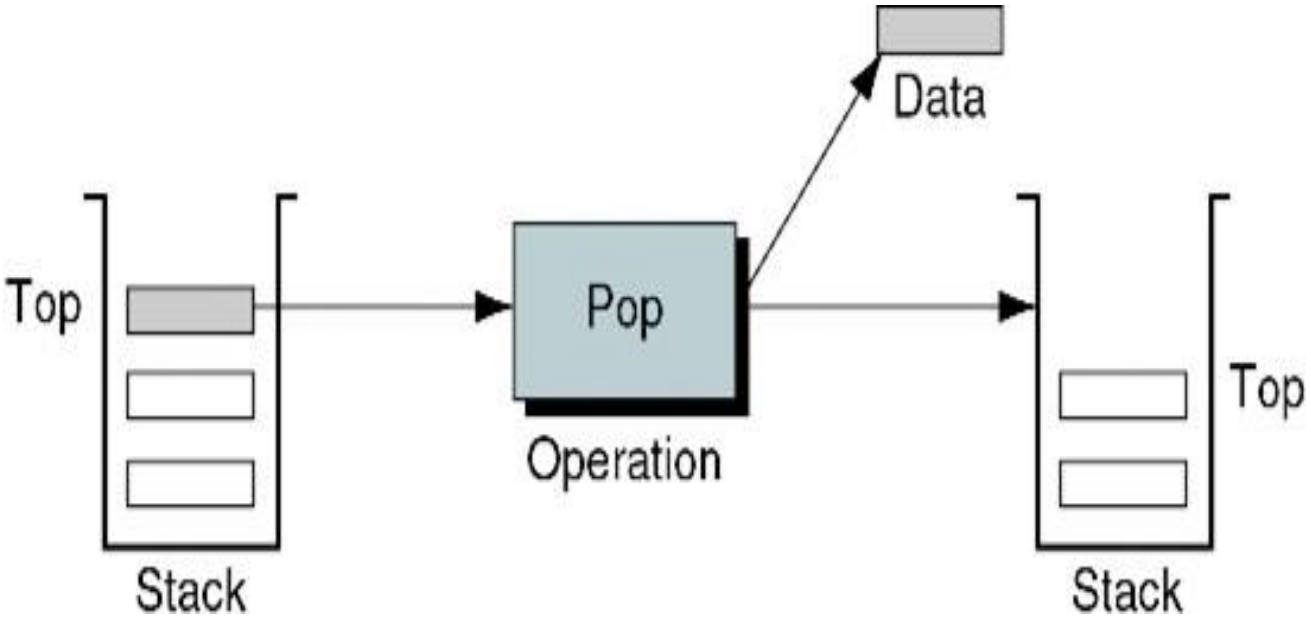


Basic Operations on a Stack

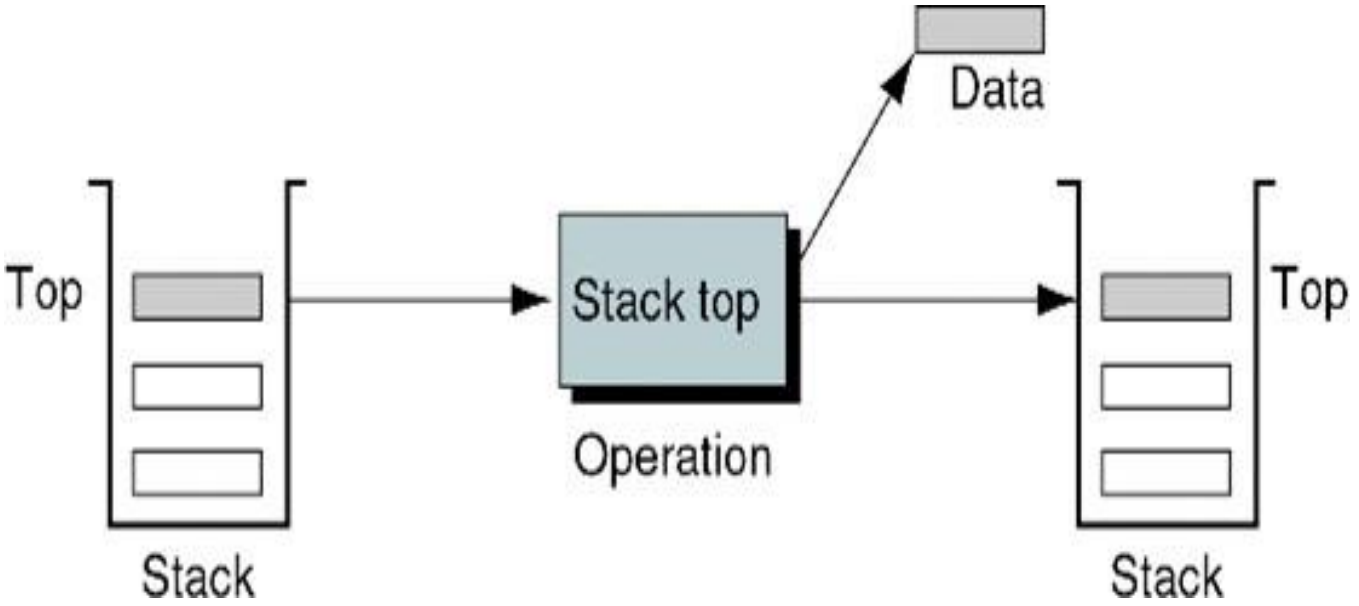
top: Returns the top element of the stack. Prior to this operation, the stack must exist and must not be empty.

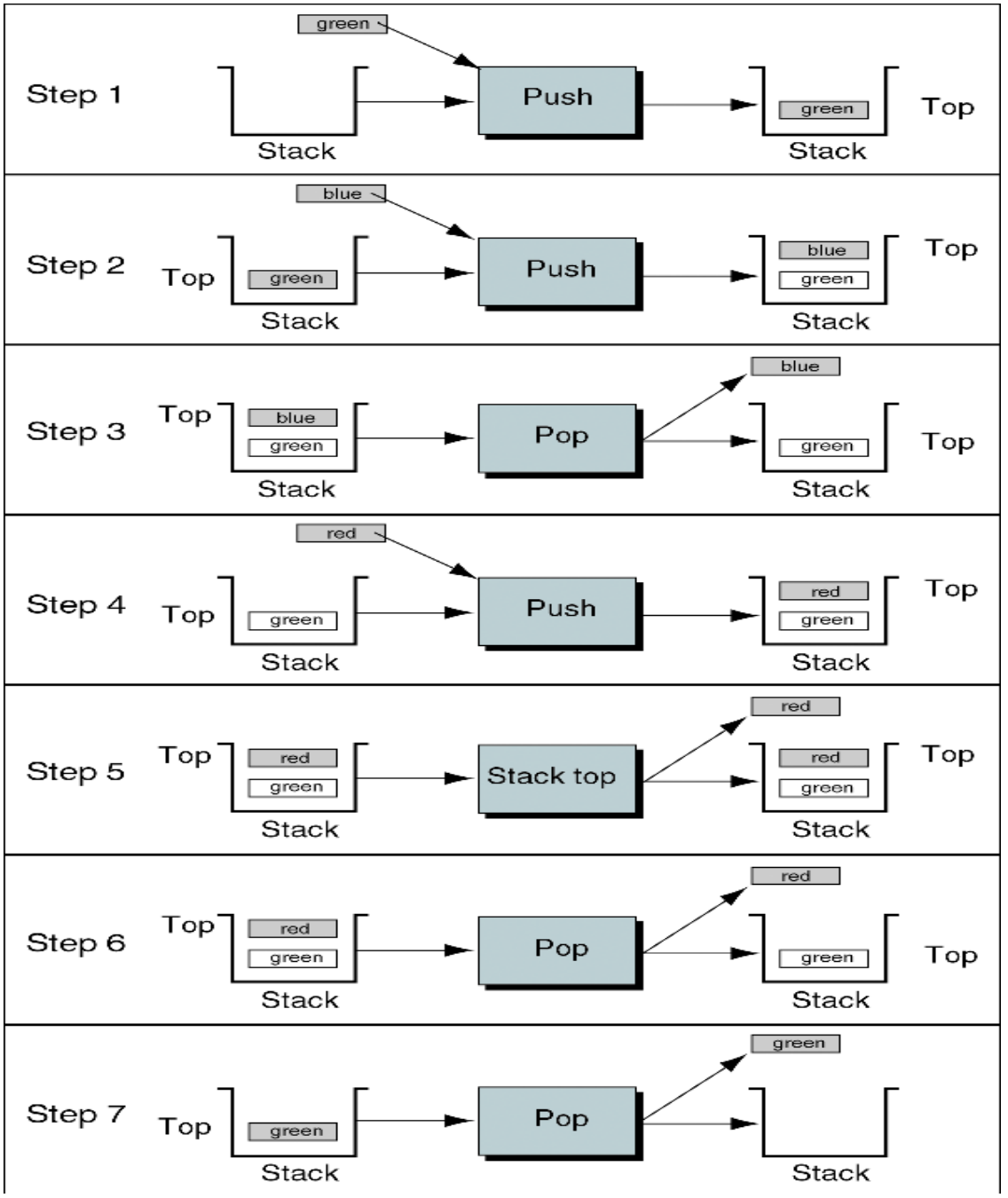
pop: Removes the top element of the stack. Prior to this operation, the stack must exist and must not be empty.

Stack Pop Operation



Stack Top Operation




Step 1
green
Push
Stack
green
Stack
Top
Step 2
blue
Top
green
Push
Stack
blue
green
Stack
Top
Step 3
Top
blue
green
Pop
Stack
blue
green
Stack
Top
Step 4
red
Top
green
Push
Stack
red
green
Stack
Top
Step 5
Top
red
green
Stack top
Stack
red
red
green
Stack
Top
Step 6
Top
red
green
Pop
Stack
red
green
Stack
Top
Step 7
Top
green
Pop
Stack
green
Stack

Representing Stacks in C

```
#define stacksize 100
struct stack
{
int top;
int items[stacksize];
};
```

Applications of Stack

1. Polish Notation
2. Recursion
3. Reversing Data
4. Backtracking

Polish Notation

Operators are either before, between or after their operands:

before → prefix

after → postfix

Note:

between → infix

Examples

a × b

prefix → × a b

postfix → a b ×

infix → a × b

a + b × c

prefix → + a × b c

postfix → a b c × +

Note:
Prefix and Postfix are not mirror to each other

Prefix – Polish notation
Postfix – Reverse polish notation

- Change the following expression to
  a) Reverse Polish notation
  b) Polish notations

3 + (4 + 6 × 2) × ((8 – 3) × (2 - 5) + 4) – 2 × 6

a) Reverse Polish Notation:

3 4 6 2 × + 8 3 – 2 5 – × 4 + × + 2 6 × –

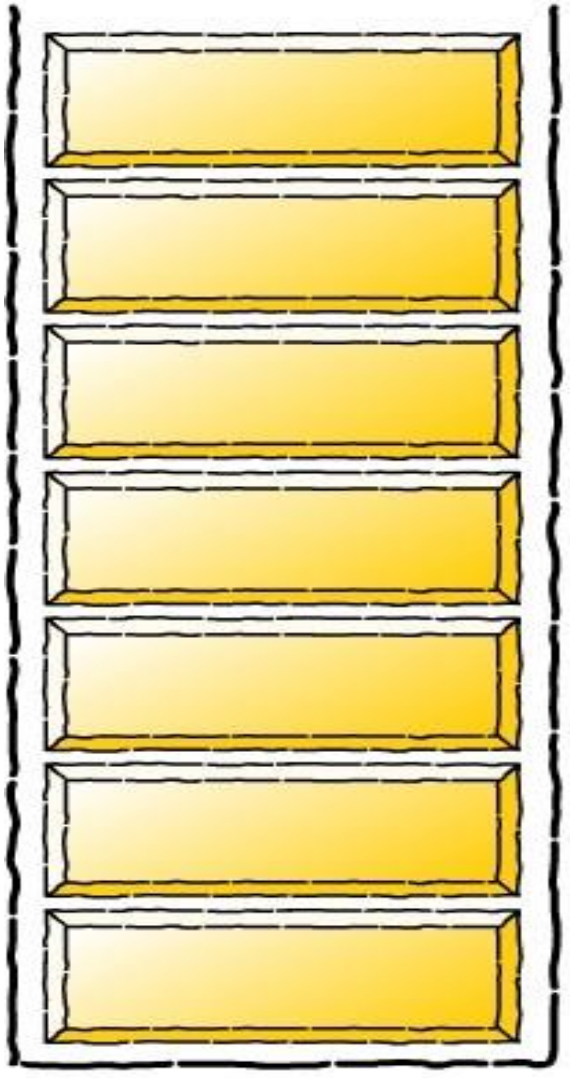
b) Polish Notation:

– + 3 × + 4 × 6 2 + × – 8 3 – 2 5 4 × 2 6

Converting Infix to Postfix with Stack

- Read expression from Left-to-Right and
  - if an operand is read copy it to the output,
  - if operator is **'('** then **push** it into the stack,
  - If operator is ')' then **pop** the stack until '(' is not found. When that occurs, both parentheses are discarded,
  - if an operator is read and has a higher precedence than the operator at the top of the stack, the operator being read is pushed onto the stack,
  - while the precedence of the operator being read is lower than or equal to the precedence of the operator at the top of the stack, the operator at the top of the stack is **popped** and copied to the output,
  - when reached the end of the expression, the remaining operators in the stack are popped and copied to the output.

Infix to postfix conversion



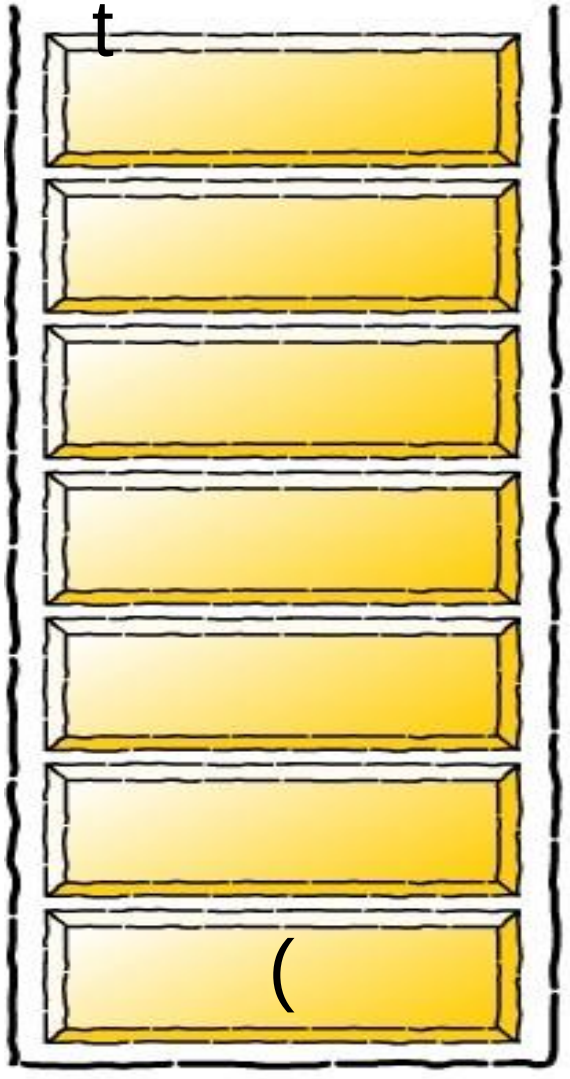
infixVec
t

(a + b - c) * d – (e + f
)

postfixVec
t

Infix to postfix conversion

stackVec
t



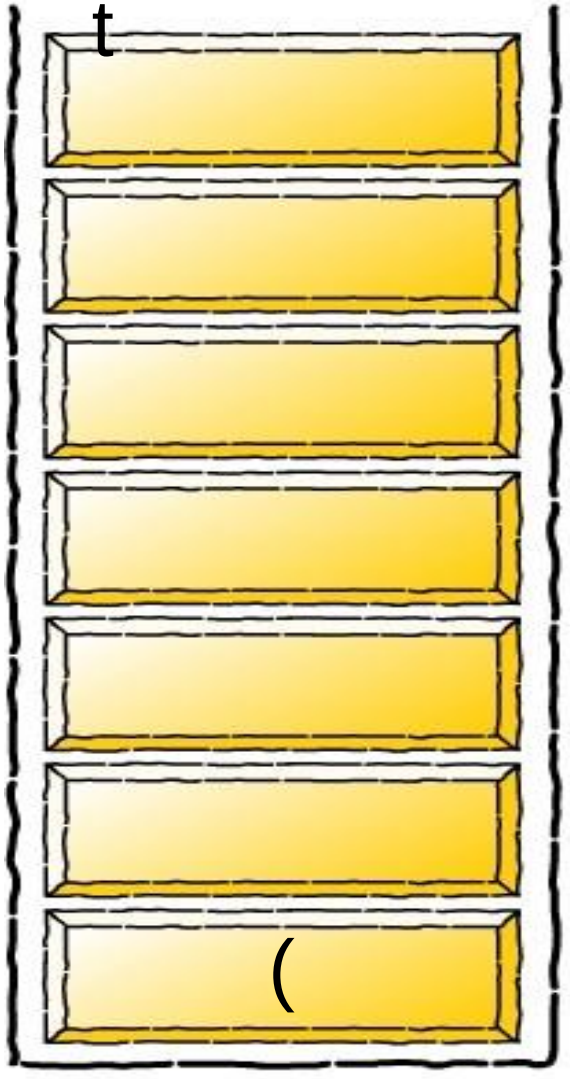
infixVec
t

a + b - c) * d – (e + f
)

postfixVec
t

Infix to postfix conversion

stackVec
t



infixVec
t

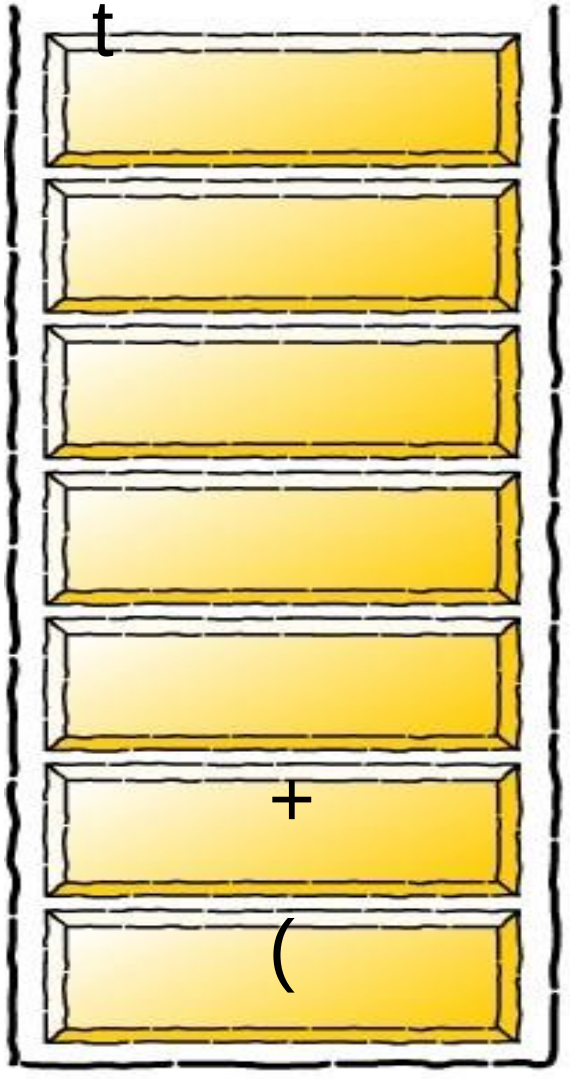
+ b - c) * d – (e + f
)

postfixVec
t

a

Infix to postfix conversion

stackVec
t



infixVec
t

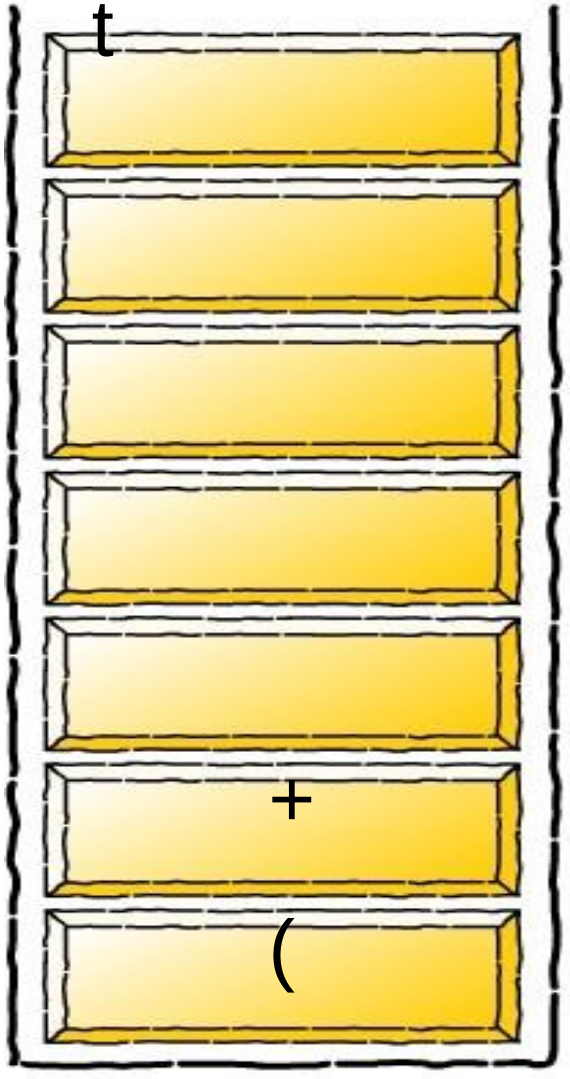
b - c) * d – (e + f
)

postfixVec
t

a

Infix to postfix conversion

stackVec
t



infixVec
t

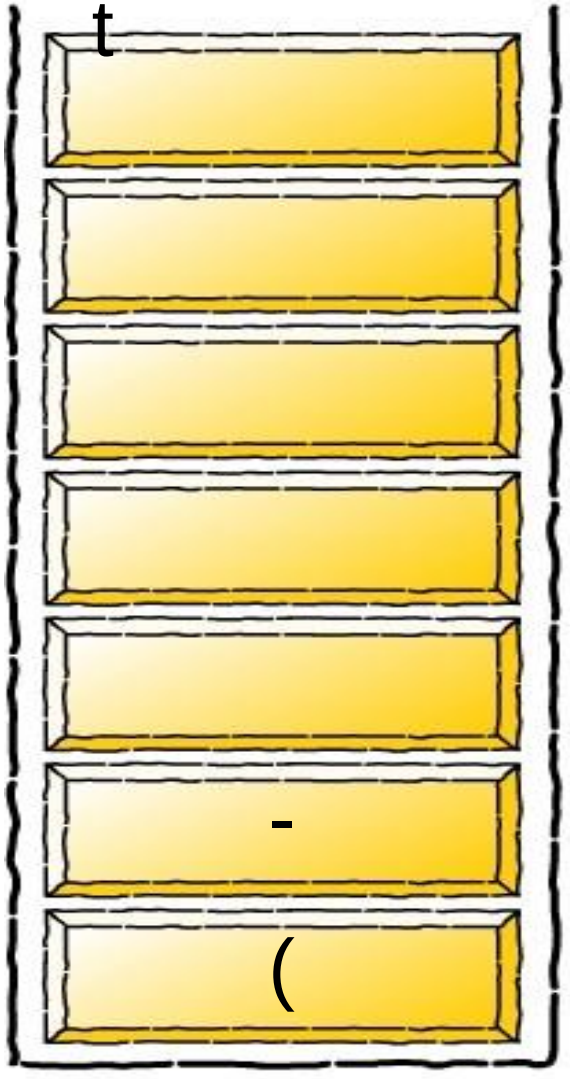
- c) * d – (e + f
)

postfixVec
t

a b

Infix to postfix conversion

stackVec
t



infixVec
t

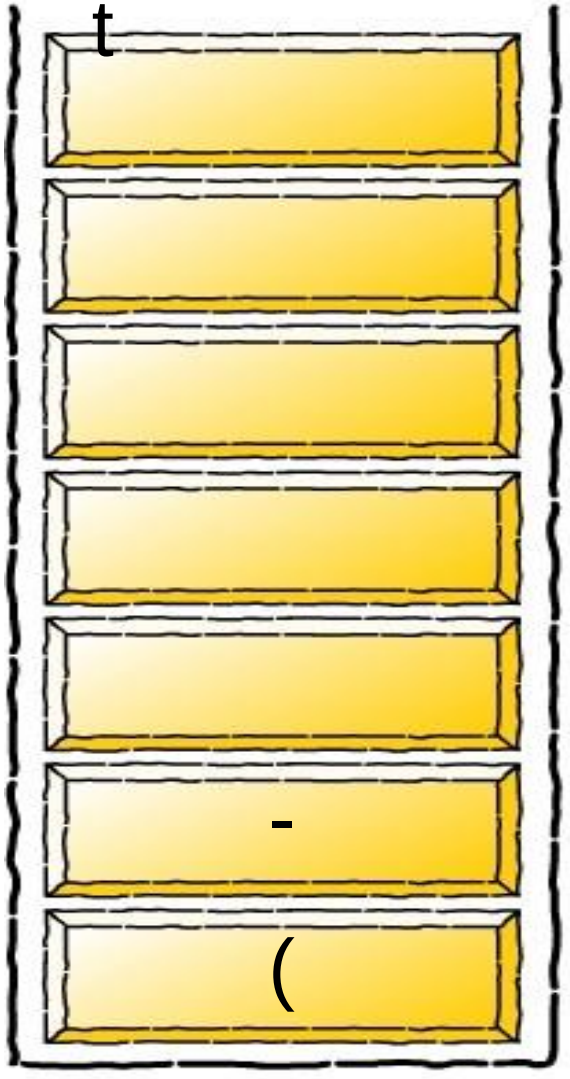
c) * d – (e + f
)

postfixVec
t

a b +

Infix to postfix conversion

stackVec
t



infixVec
t

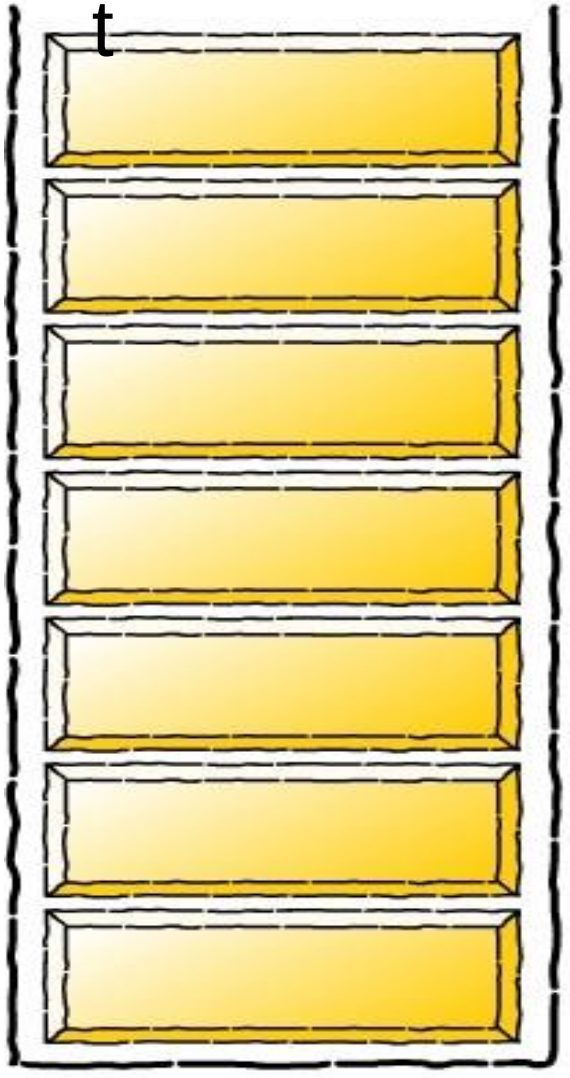
) * d – (e + f
)

postfixVec
t

a b + c

Infix to postfix conversion

stackVec
t



infixVec
t

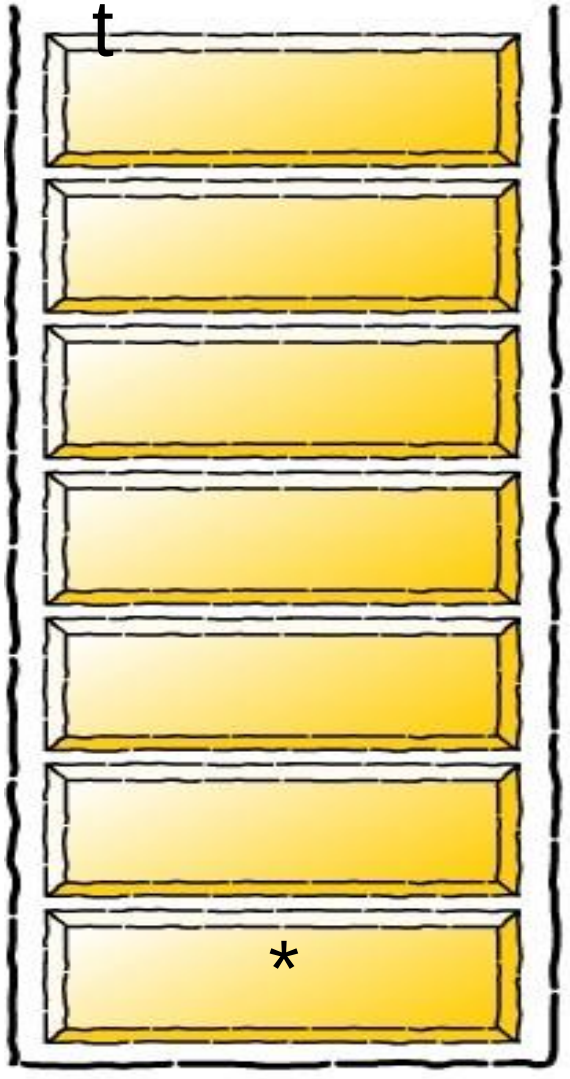
* d – (e + f
)

postfixVec
t

a b + c -

Infix to postfix conversion

stackVec
t



infixVec
t

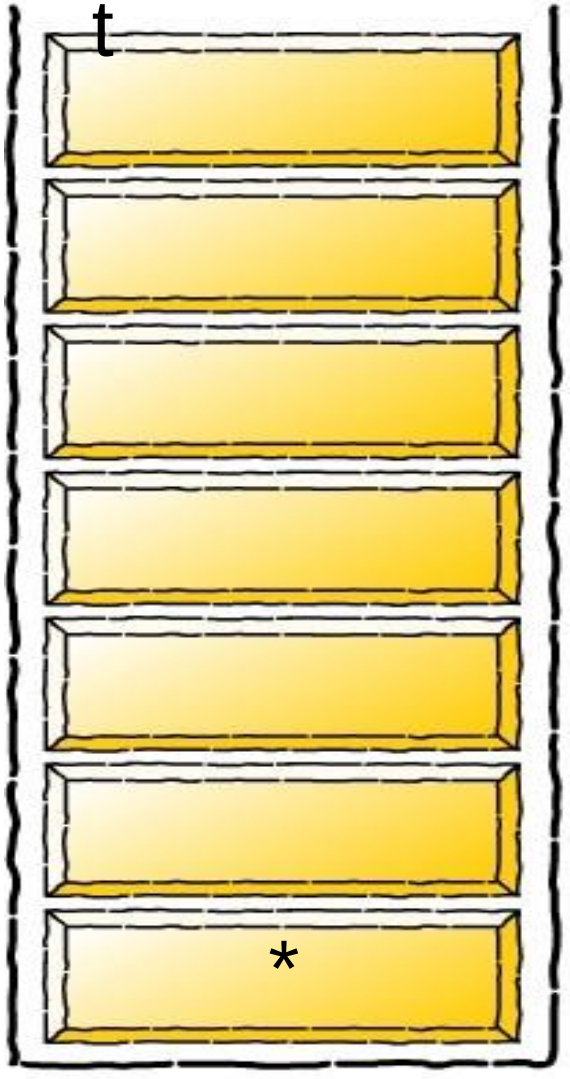
d – (e + f
)

postfixVec
t

a b + c -

Infix to postfix conversion

stackVec
t



infixVec
t

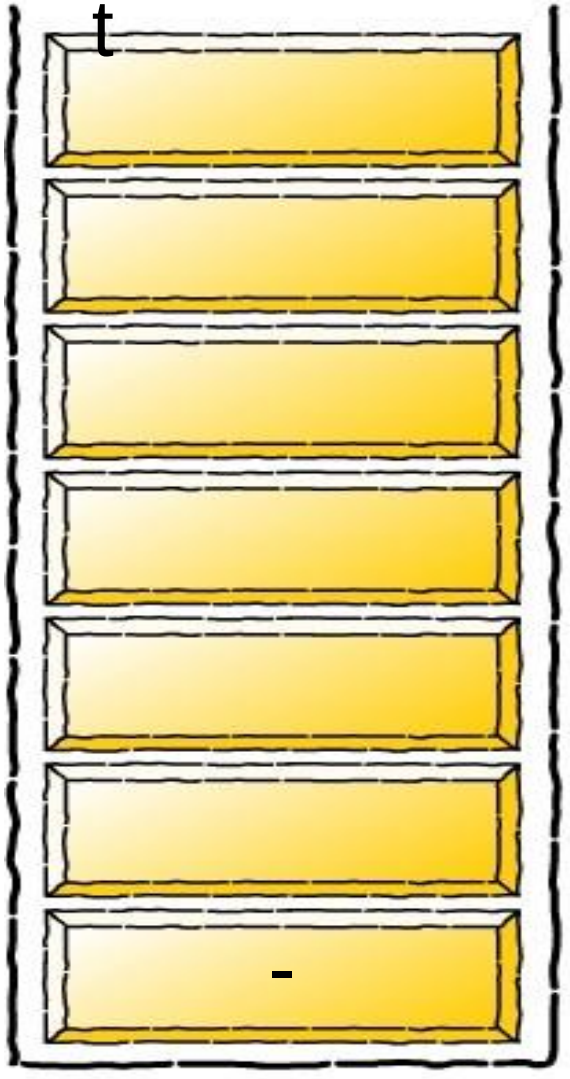
– (e + f
)

postfixVec
t

a b + c - d

Infix to postfix conversion

stackVec
t



infixVec
t

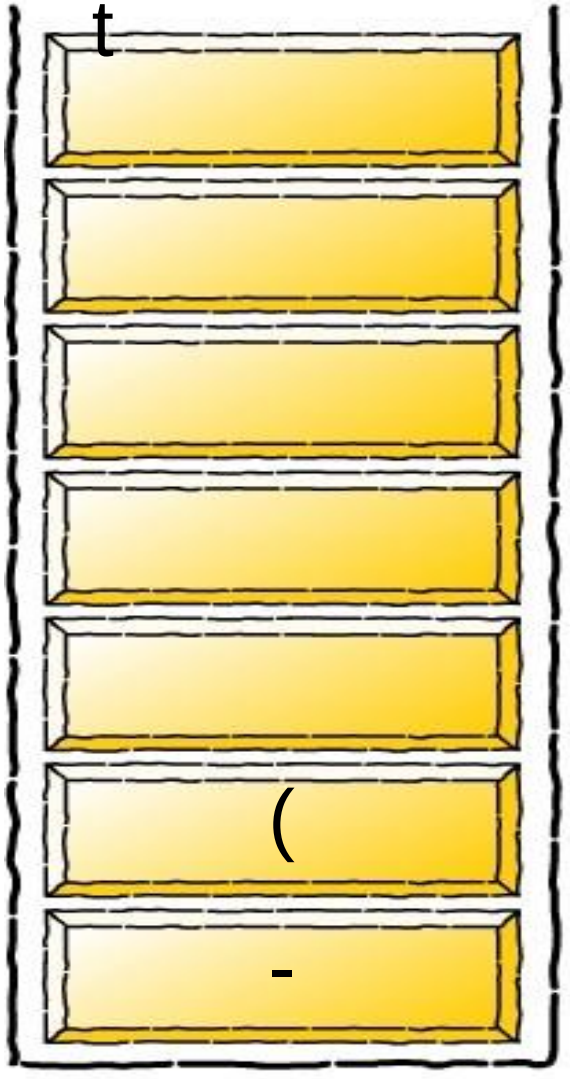
(e + f
)

postfixVec
t

a b + c – d *

Infix to postfix conversion

stackVec
t



infixVec
t

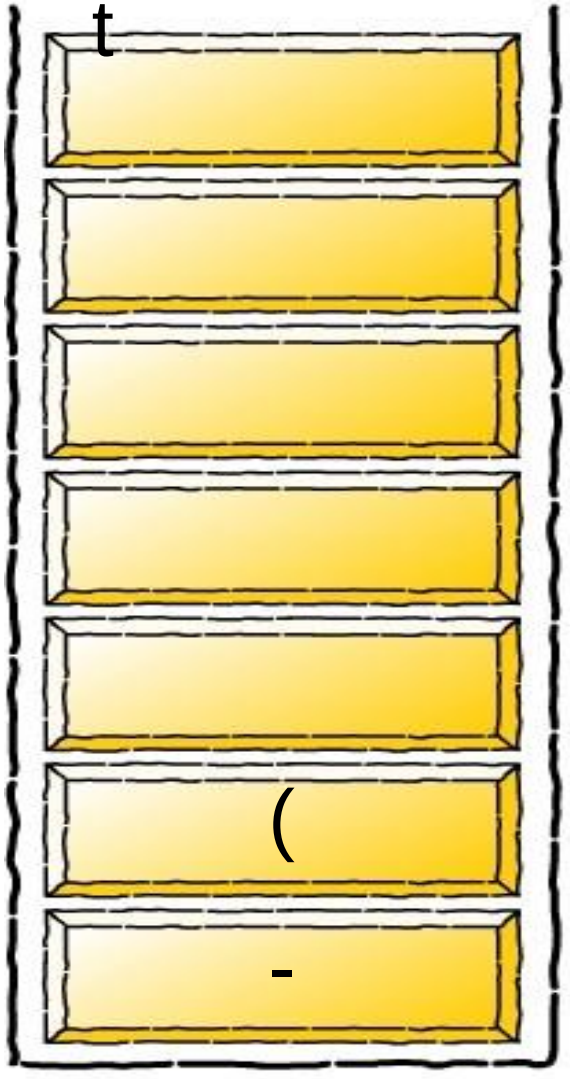
e + f
)

postfixVec
t

a b + c – d *

Infix to postfix conversion

stackVec
t



infixVec
t

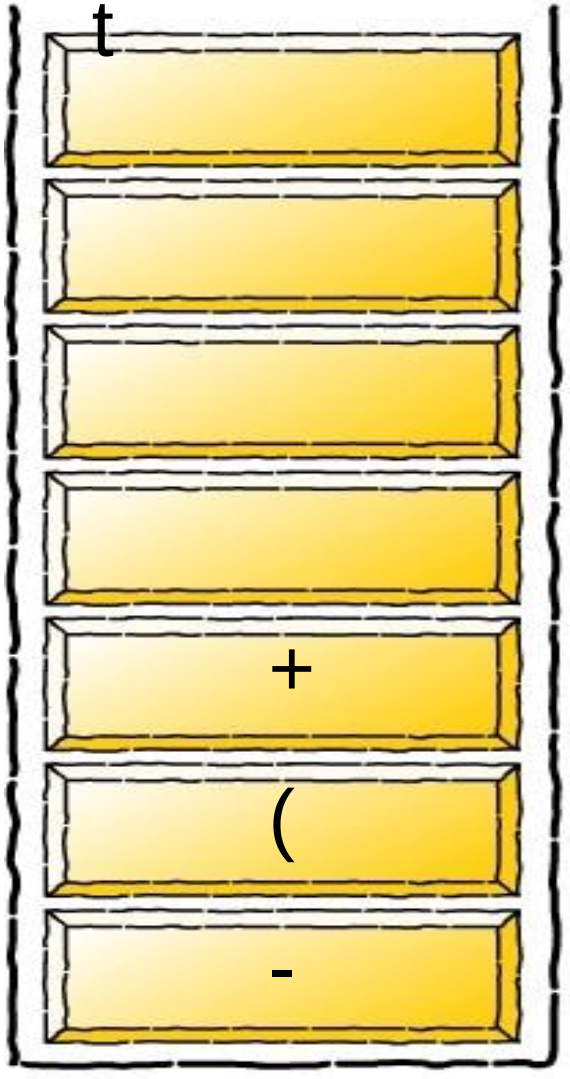
+ f
)

postfixVec
t

a b + c – d * e

Infix to postfix conversion

stackVec
t



infixVec
t

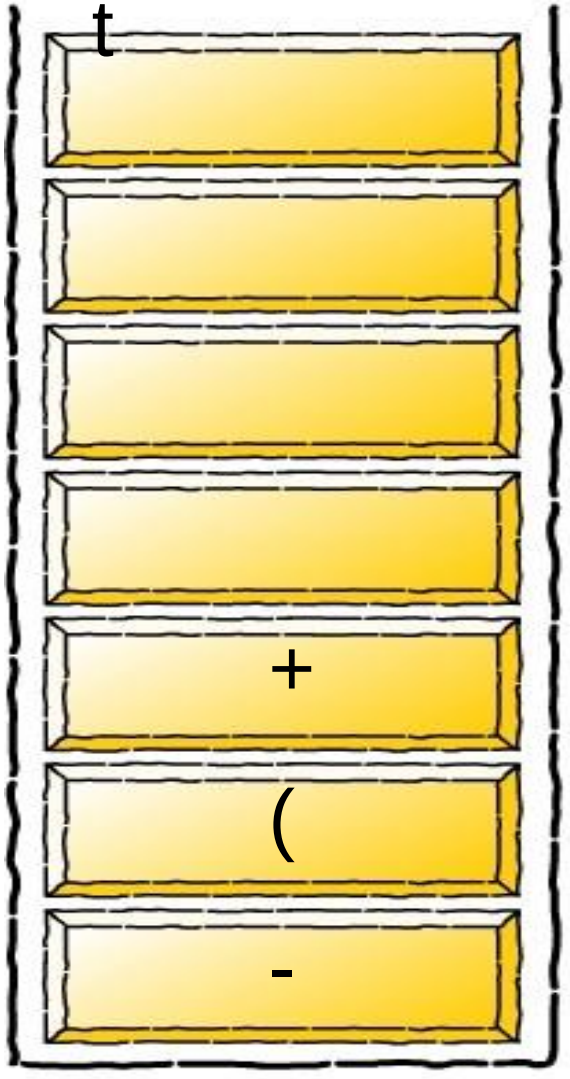
f
)

postfixVec
t

a b + c – d * e

Infix to postfix conversion

stackVec
t



infixVec
t

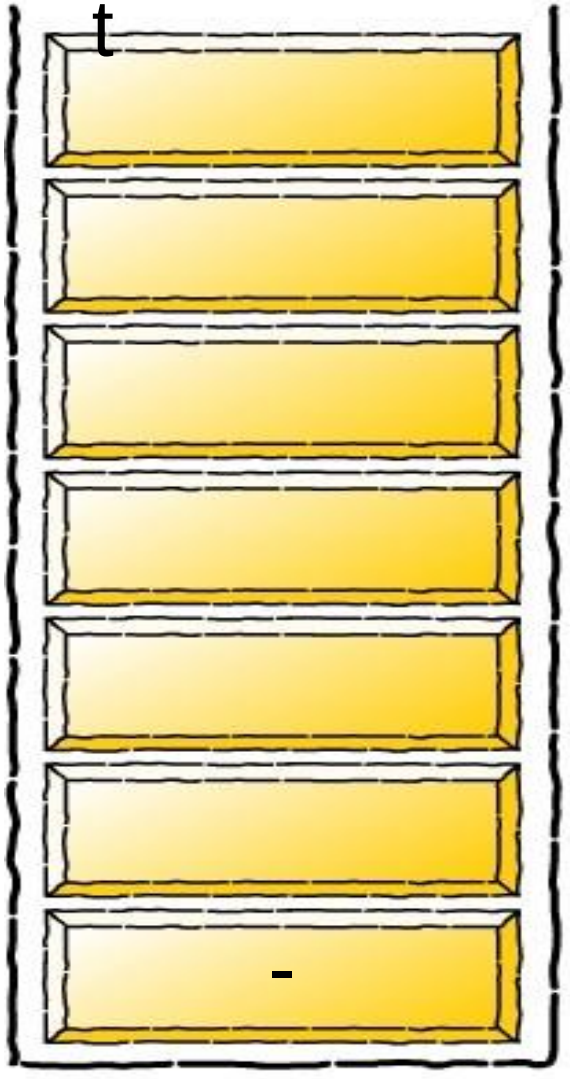
)

postfixVec
t

a b + c – d * e f

Infix to postfix conversion

stackVec
t



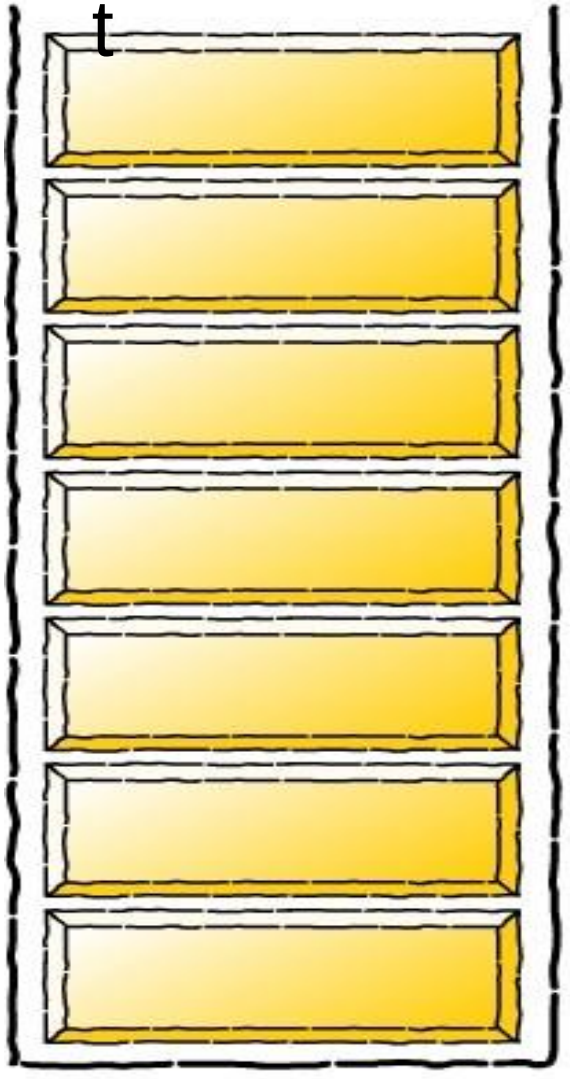
infixVec
t

postfixVec
t

a b + c – d * e f +

Infix to postfix conversion

stackVec
t



infixVec
t

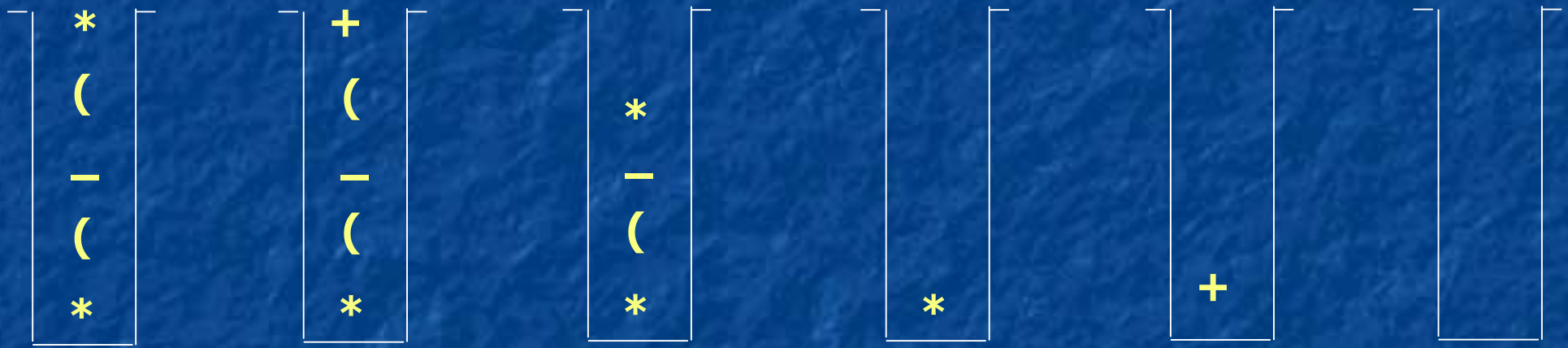
postfixVec
t

a b + c – d * e f + -

Example- Infix to Postfix

Input: 4 * (2 – (6 * 3 + 4) * 2) +1

Output: 4 2 6 3 * 4 + 2 * - * 1 +



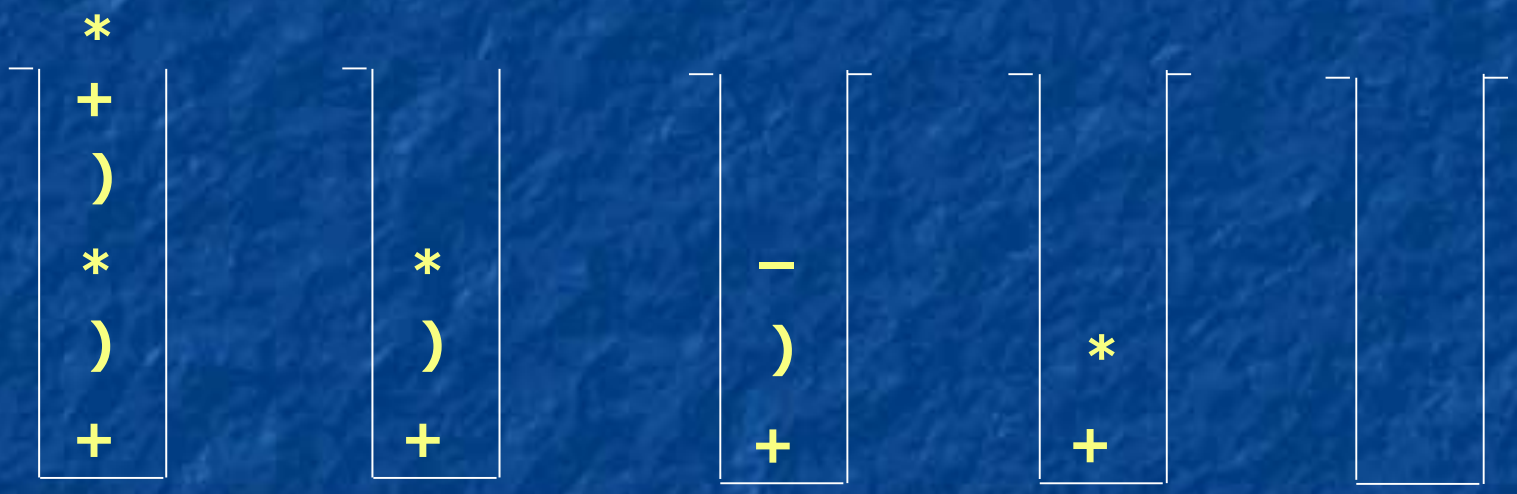
Converting Infix to Prefix with Stack

- **Read expression from Right-to-Left and**
  - if an operand is read copy it to the LEFT of the output,
  - if a right parenthesis is read push it into the stack,
  - when a left parenthesis is encountered, the operator at the top of the stack is popped off the stack and copied to the LEFT of the output ***until*** the symbol at the top of the stack is a right parenthesis. When that occurs, both parentheses are discarded,
  - if an operator is scanned and has a higher or equal precedence than the operator at the top of the stack, the operator being scanned is pushed onto the stack,
  - while the precedence of the operator being scanned is lower than to the precedence of the operator at the top of the stack, the operator at the top of the stack is popped and copied to the LEFT of the output,
  - when the end of the expression is reached on the input scan, the remaining operators in the stack are popped and copied to the LEFT of the output.

Example

Input: 4 * (2 – (6 * 3 + 4) * 2) +

Output: + * 4 – 2 * + * 6 3 4 2 1



Exercises

- Using stack diagrams convert the following expressions into postfix and prefix forms of polish notation:

a) $8 - 3 \times 4 + 2$

b) $8 - 3 \times (4 + 2)$

c) $(8 - 3) \times (4 + 2)$

d) $(8 - 3) \times 4 + 2$

e) $(a + b) \times (c + a) - 5$

Evaluation of Reverse Polish Expressions

Most compilers use the polish form to translate expressions into machine language.

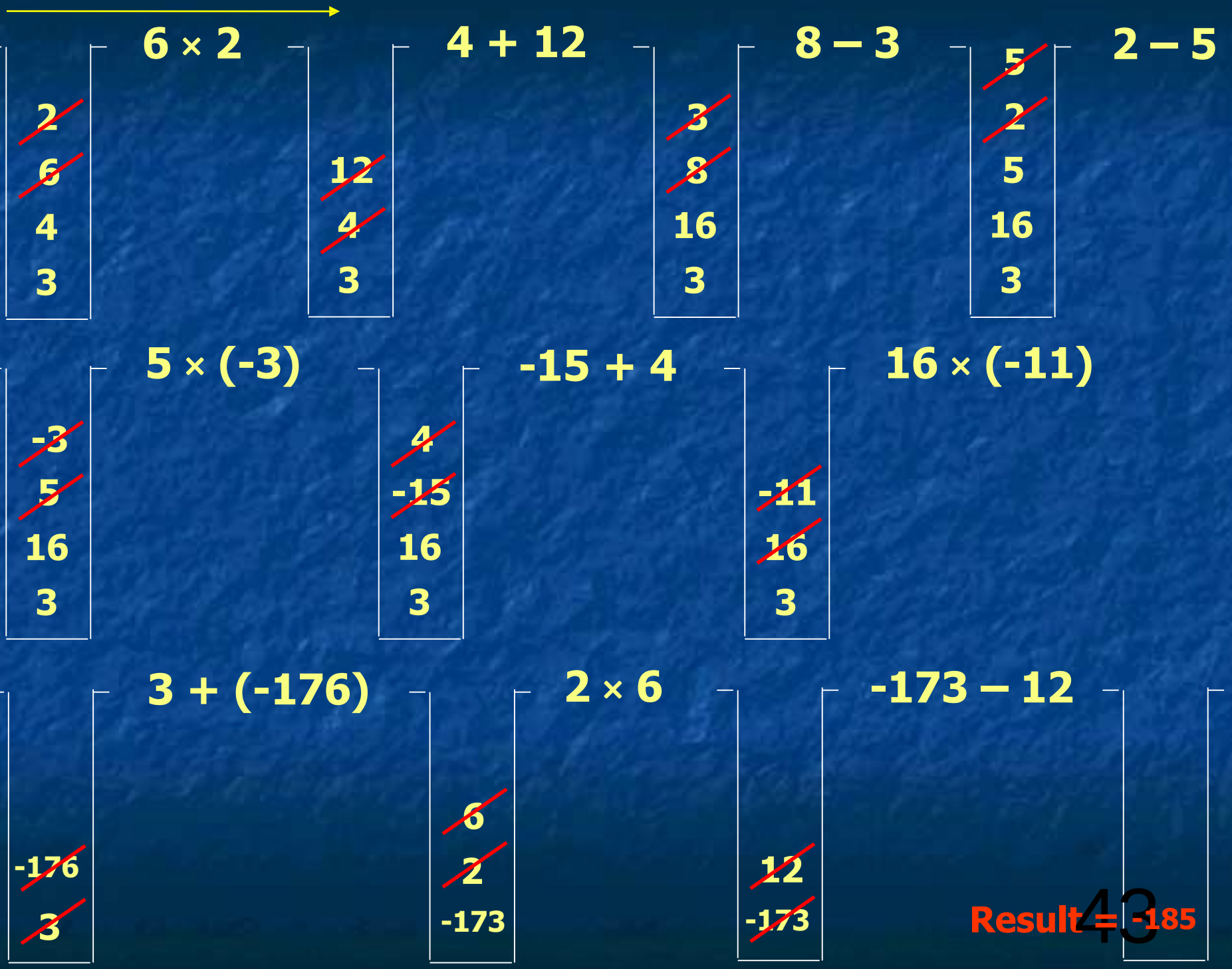
Evaluation is done using a **stack** data-structure

Read expression from **left to right** and build the stack of numbers (operands).

When an operator is read two operands are **popped** out of the stack they are evaluated with the operator and the result is **pushed** into the stack.

At the end of the expression there must be only one operand into the stack (the solution) otherwise ERROR.

3 4 6 2 × + 8 3 – 2 5 – × 4 + × + 2 6 × –



Evaluation of Polish/Prefix Expressions

Evaluation is done using a **stack** data-structure

Read expression from **right to left** and build the stack of numbers (operands).

When an operator is read two operands are **popped** out of the stack they are evaluated with the operator and the result is **pushed** into the stack.

At the end of the expression there must be only one operand into the stack (the solution) otherwise ERROR.

– × 3 – 8 × 3 2 – ~ 4 – 6 2

Recursion

What's behind this function ?

```
public int f(int a){
        if (a==1)
           return(1);
        else
           return(a * f( a-1));
}
```

It computes f! (factorial)

Factorial

Factorial:

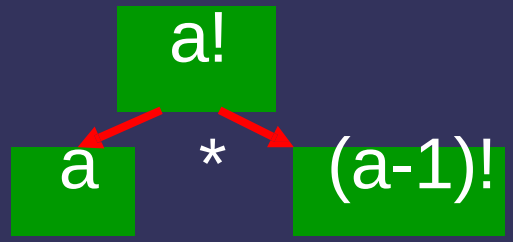
$$a! = 1 * 2 * 3 * \ldots * (a-1) * a$$

Note:

$$a! = a * (a-1)!$$

remember:

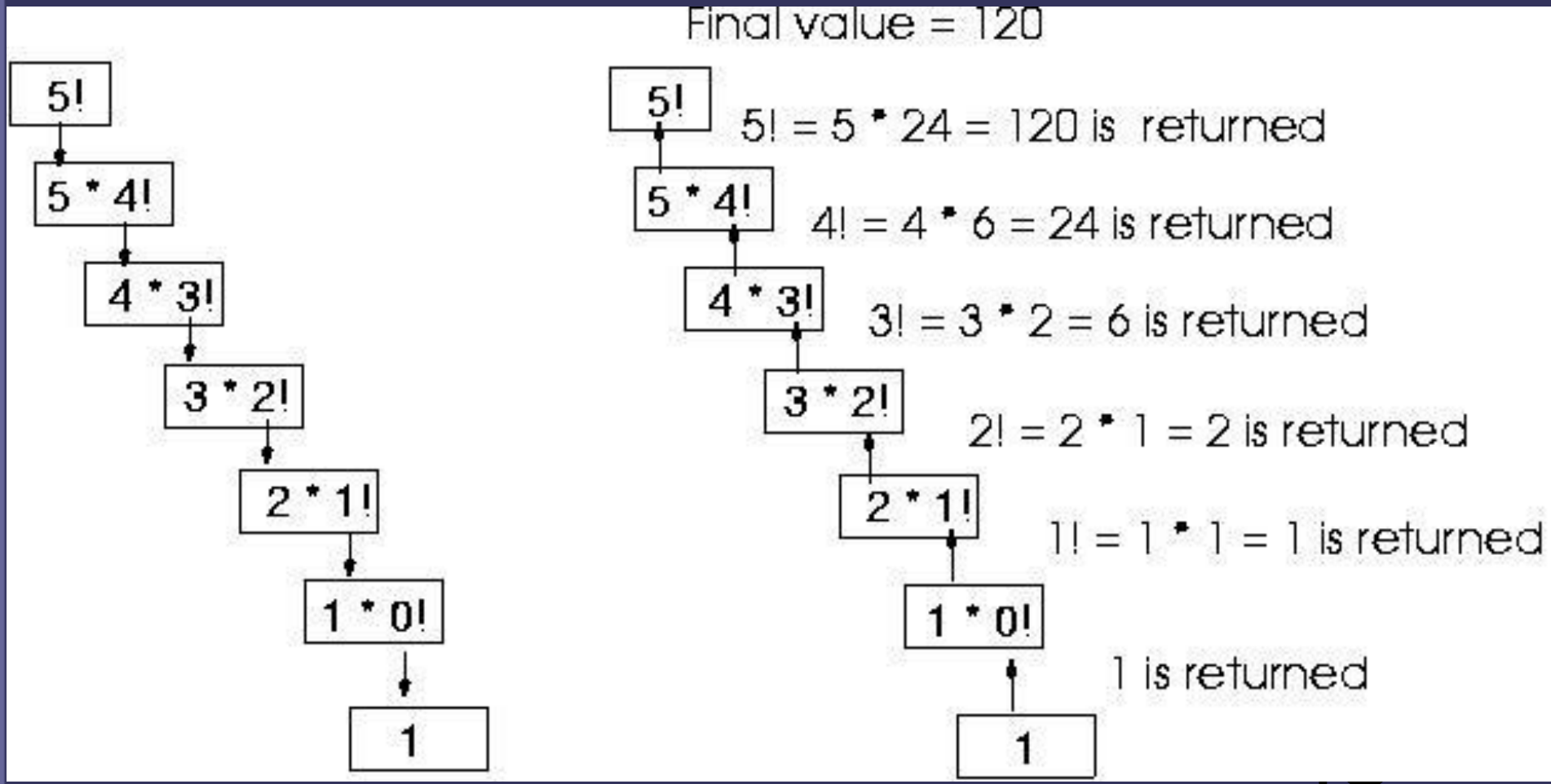
...splitting up the problem into a smaller problem of the same type...



Tracing the example

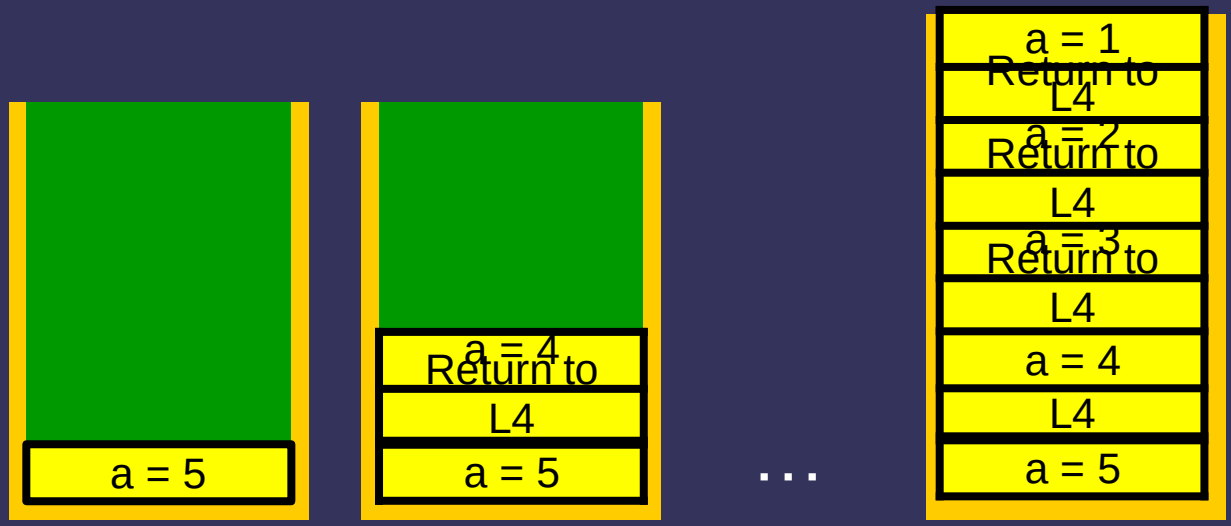
```
public int factorial(int a){
        if (a==0)
           return(1);
        else
           return(a * factorial( a-1));
}
```

RECURSION !



Watching the Stack

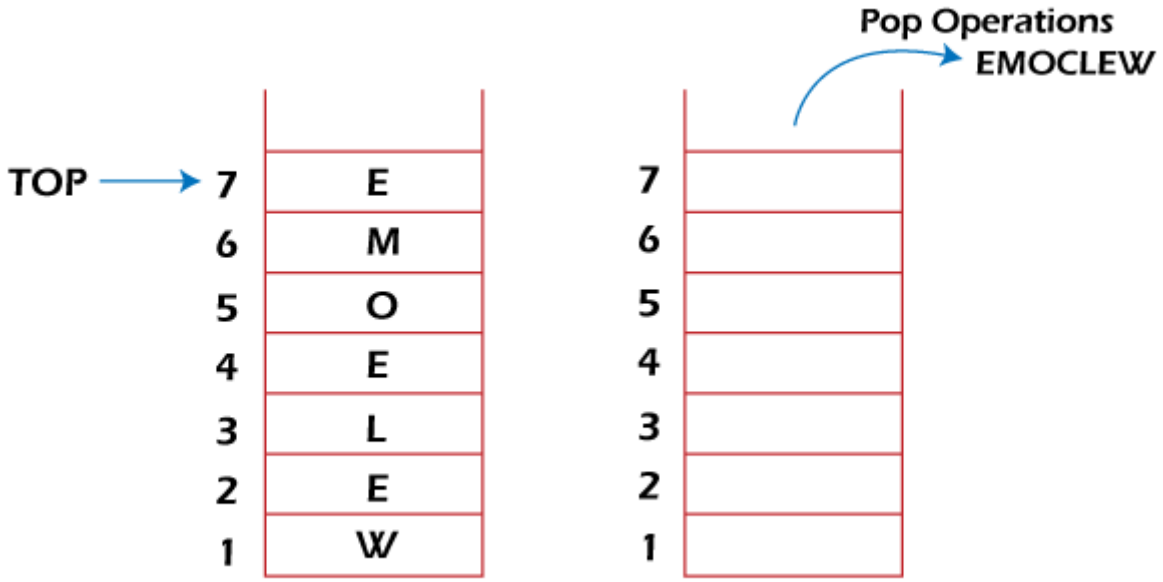
```
public int factorial(int a){
        if (a==1)
           return(1);
        else
           return(a * factorial( a-1));
}
```



Every call to the method creates a new set of local variables !

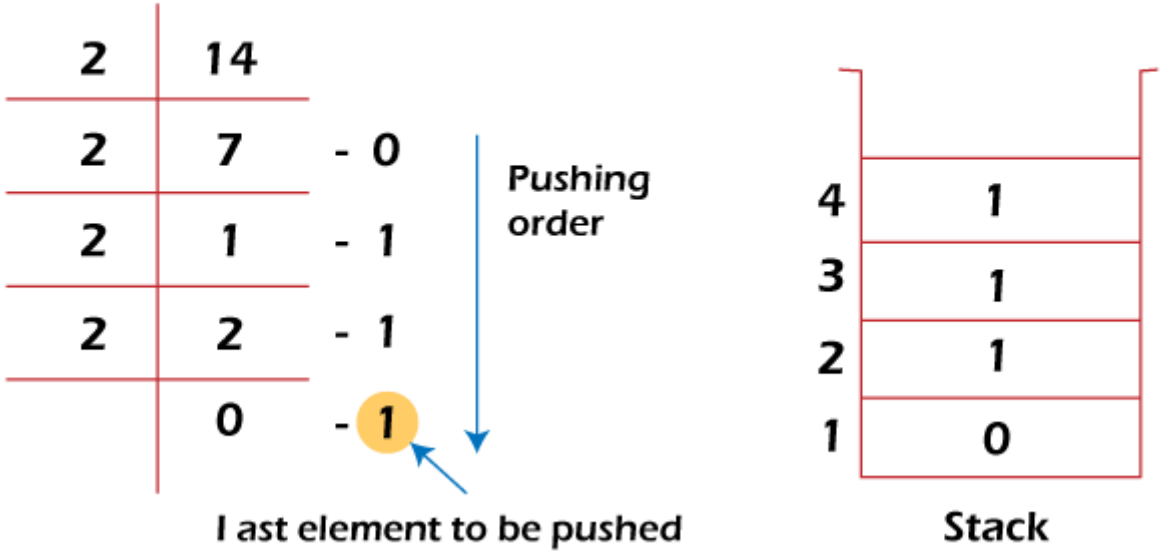
Watching the Stack

```
public int factorial(int a){
        if (a==1)
           return(1);
        else
           return(a * factorial( a-1));
}
```



Reverse string

Example: Converting 14 number Decimal to Binary:



Hexadecimal 14 to binary 1110

Parenthesis Matching

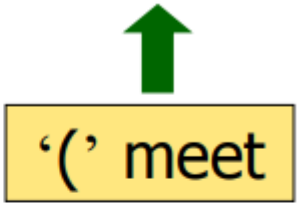
- Print out matching of the left and right parentheses in a character string
  - (a*(b+c)+d) : output ➔ (0,10), (3,7) match
  - (a+b))(
    - Output ➔ (0, 4) match
    - Output ➔ 5, 6 have no matching parentheses

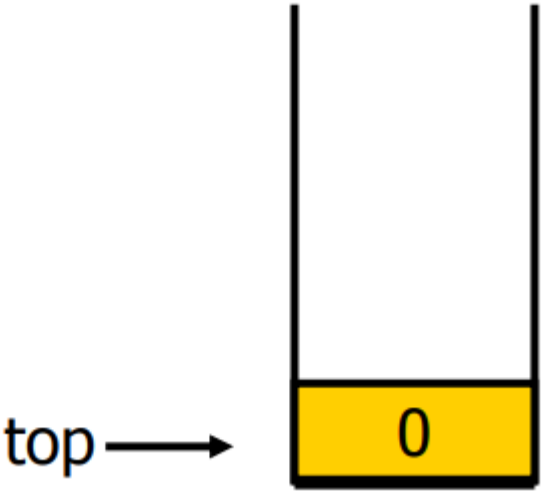
- Solution steps
  - Scan the input expression from left to right
  - ` ( ` is encountered, add its position to the stack
  - ` ) ` is encountered, remove matching position from stack

Example: Parenthesis Matching (1)

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

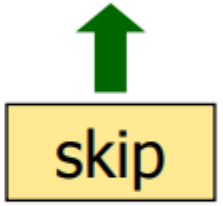
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

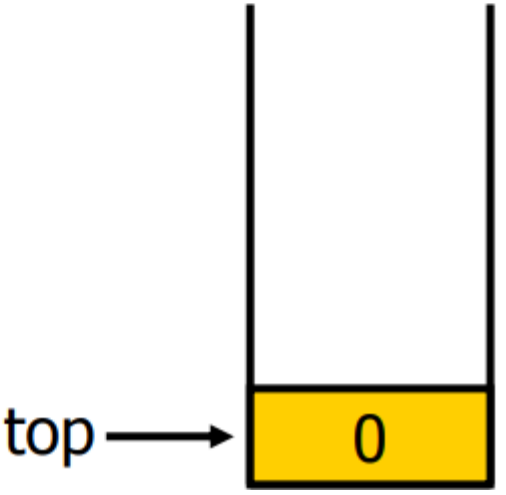




- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

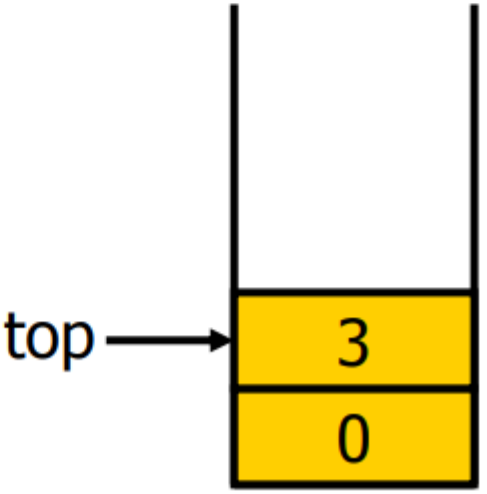




- (a*(b+c)+d)

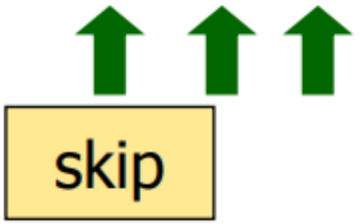
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

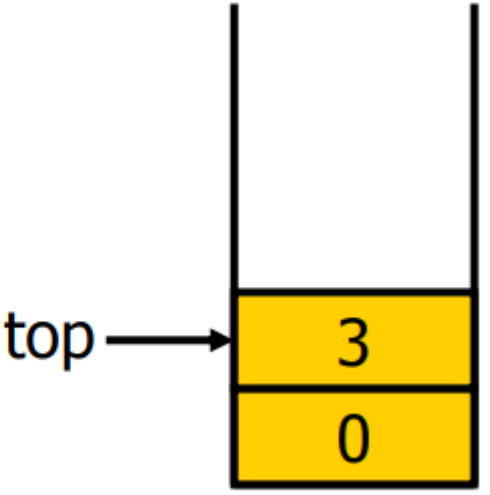
‘(‘ meet



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

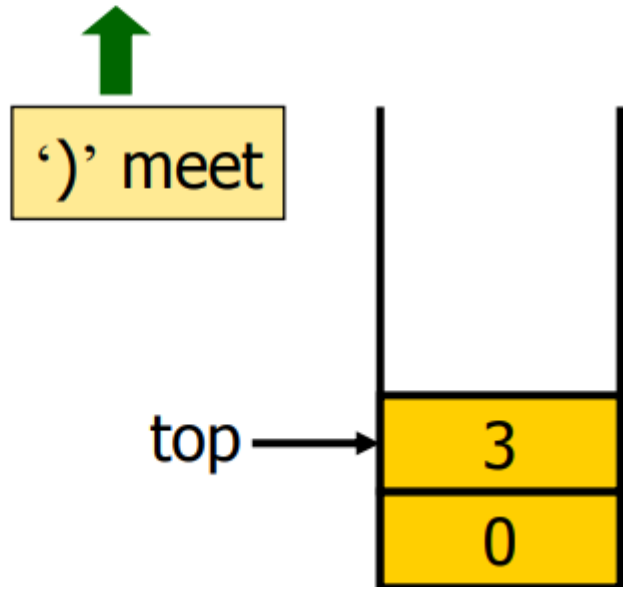
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)





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

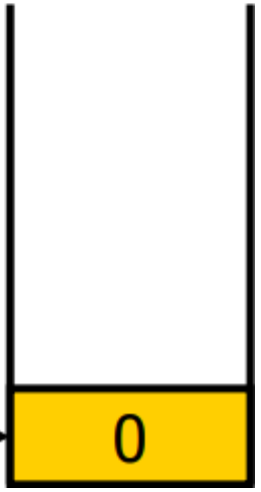
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



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

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

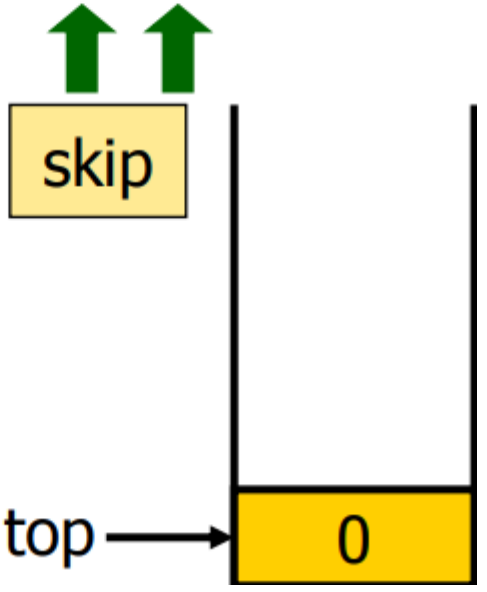
‘)’ meet



Output : (3, 7)

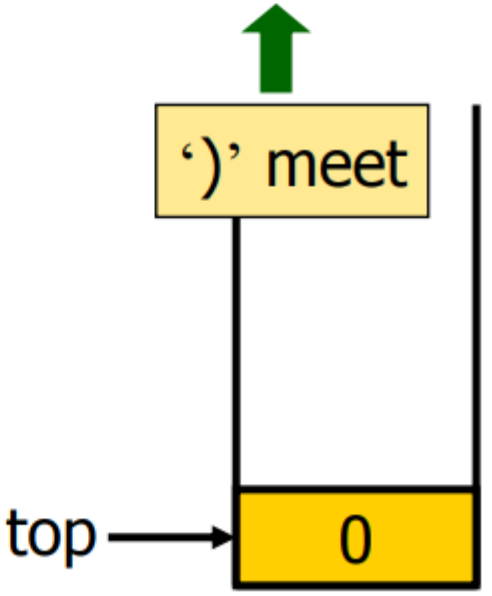
- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



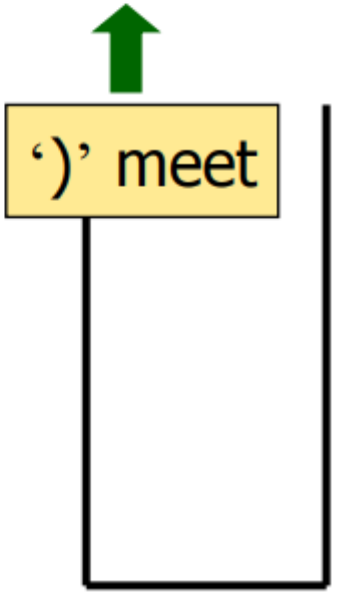
(a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



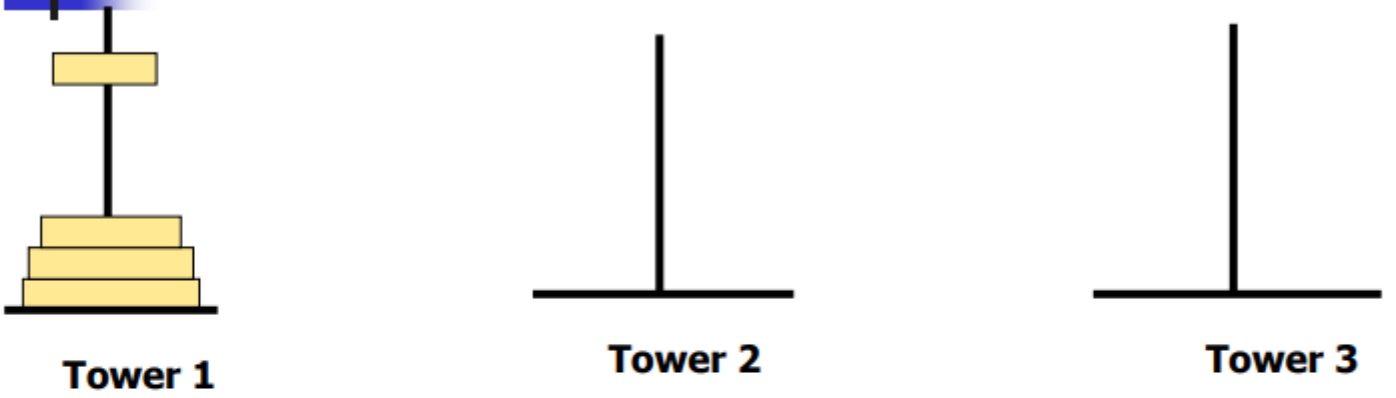
- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



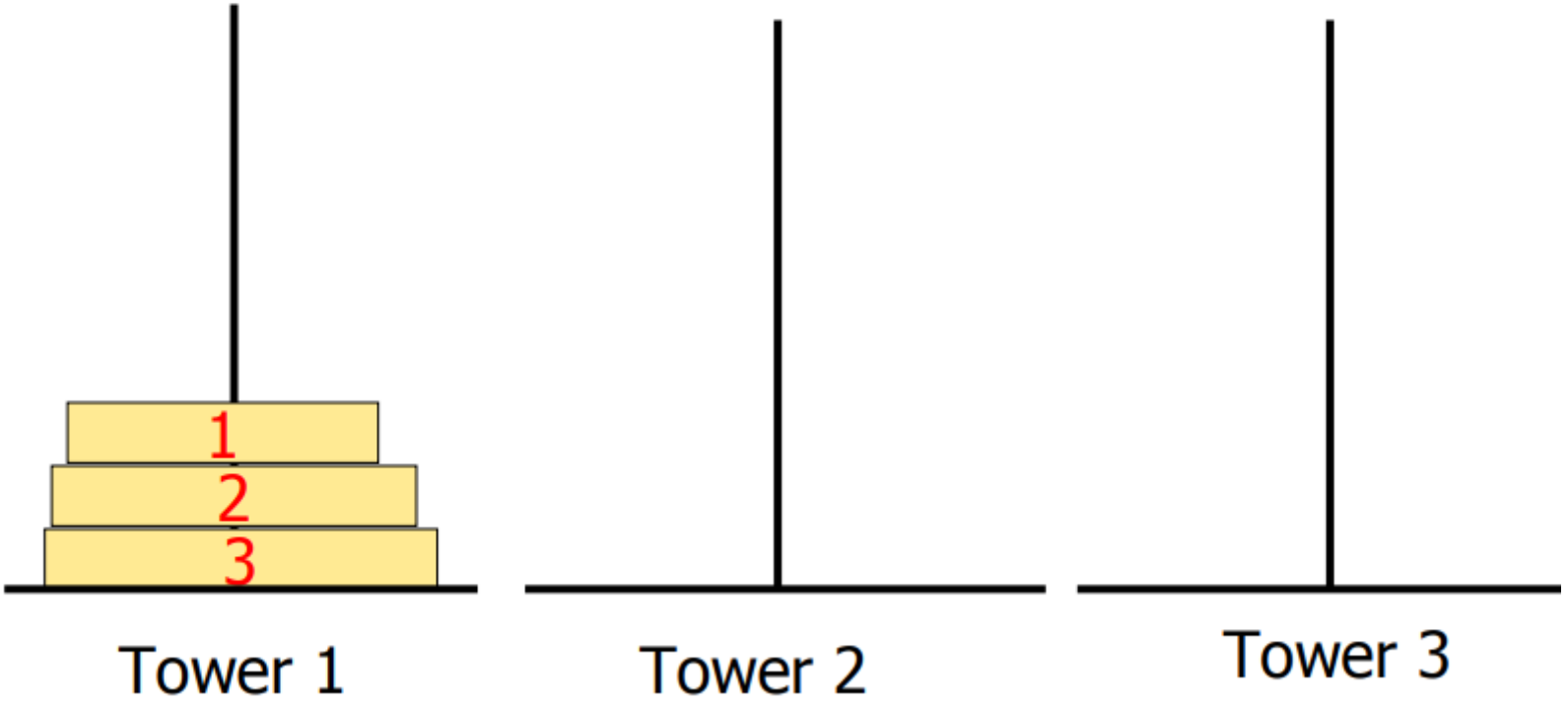
Output : (3, 7), (0, 10)

Towers of Hanoi

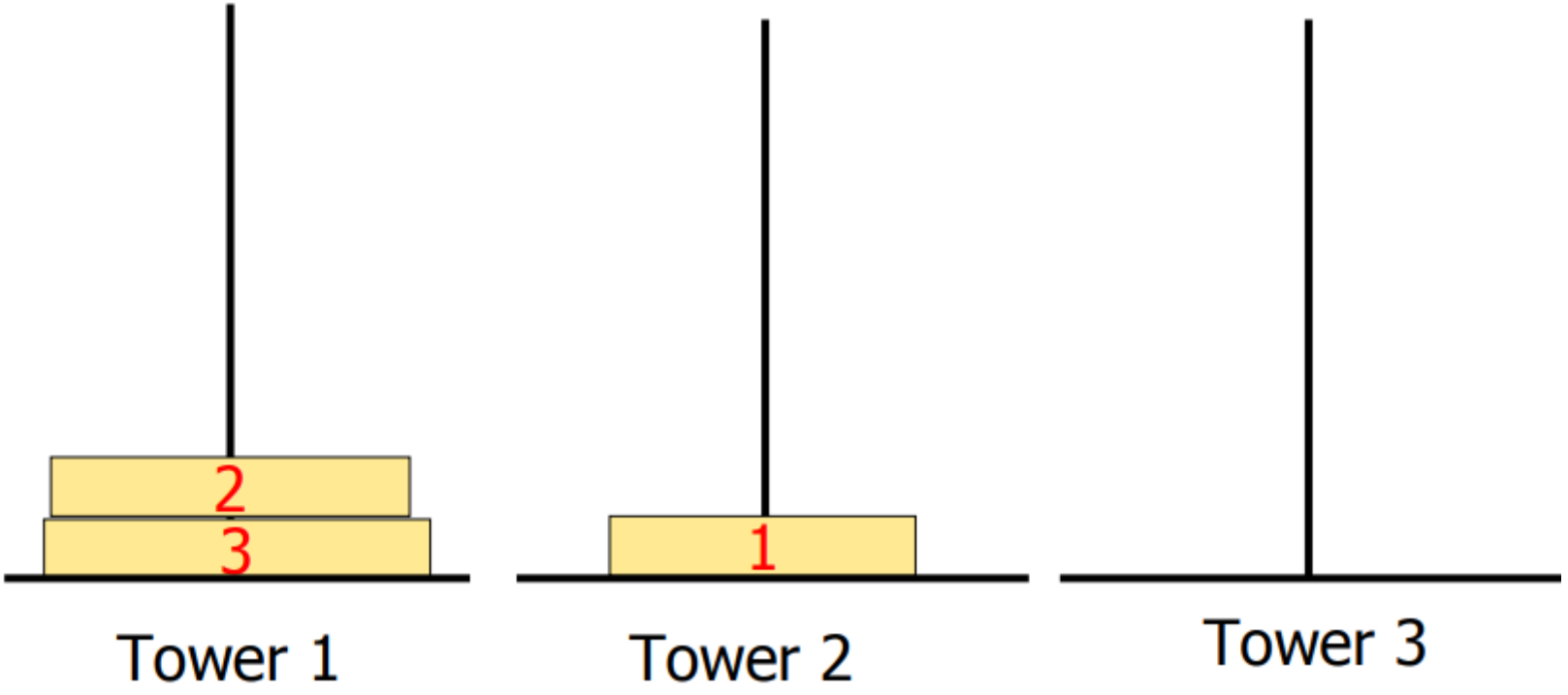


- Mission: Move the disks from tower1 to tower2
- Each tower operates as a stack
- Cannot place a big disk on top of a smaller one
  - Move n-1 disks to tower3 using tower2
  - Move the largest to tower2
  - Move the n-1 disks from tower3 to tower2 using tower1
- Use of Recursion

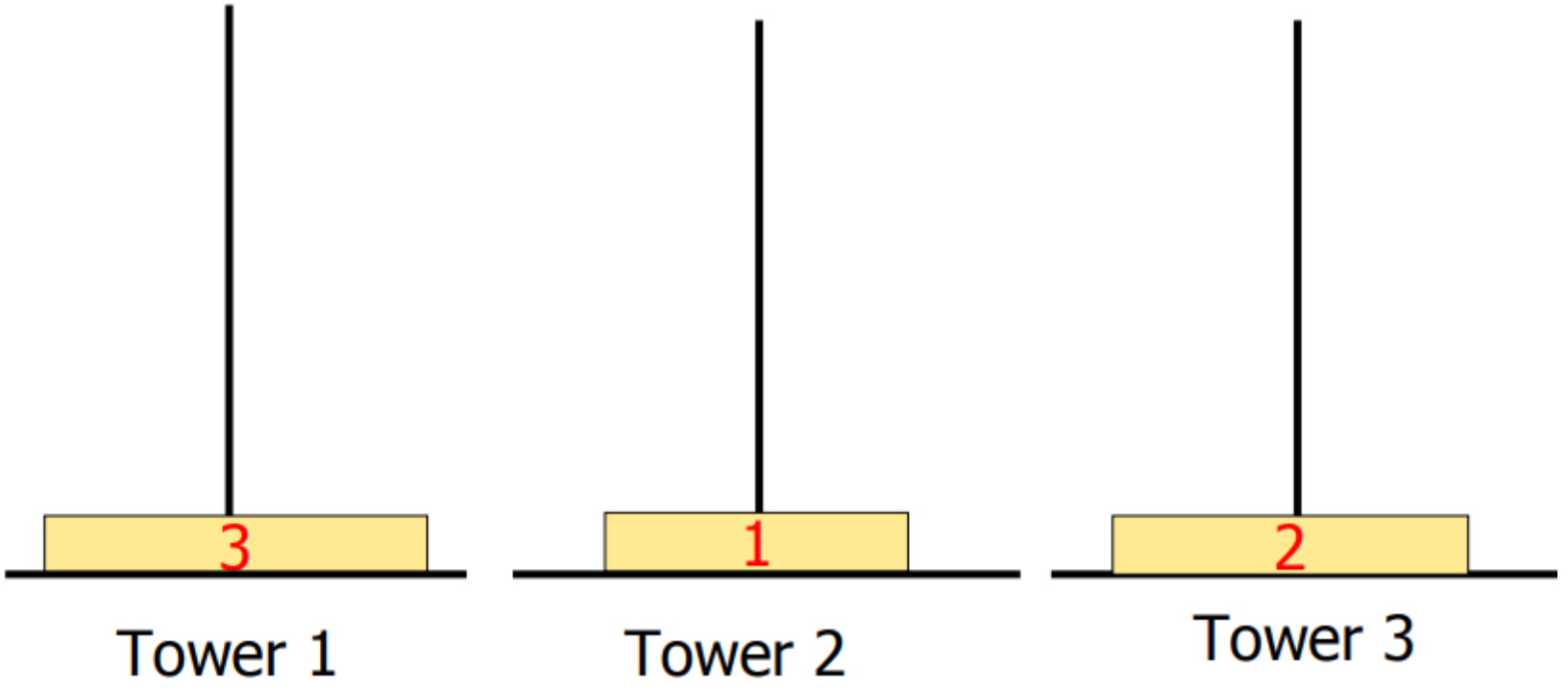
- Mission: Move the disks from tower1 to tower2



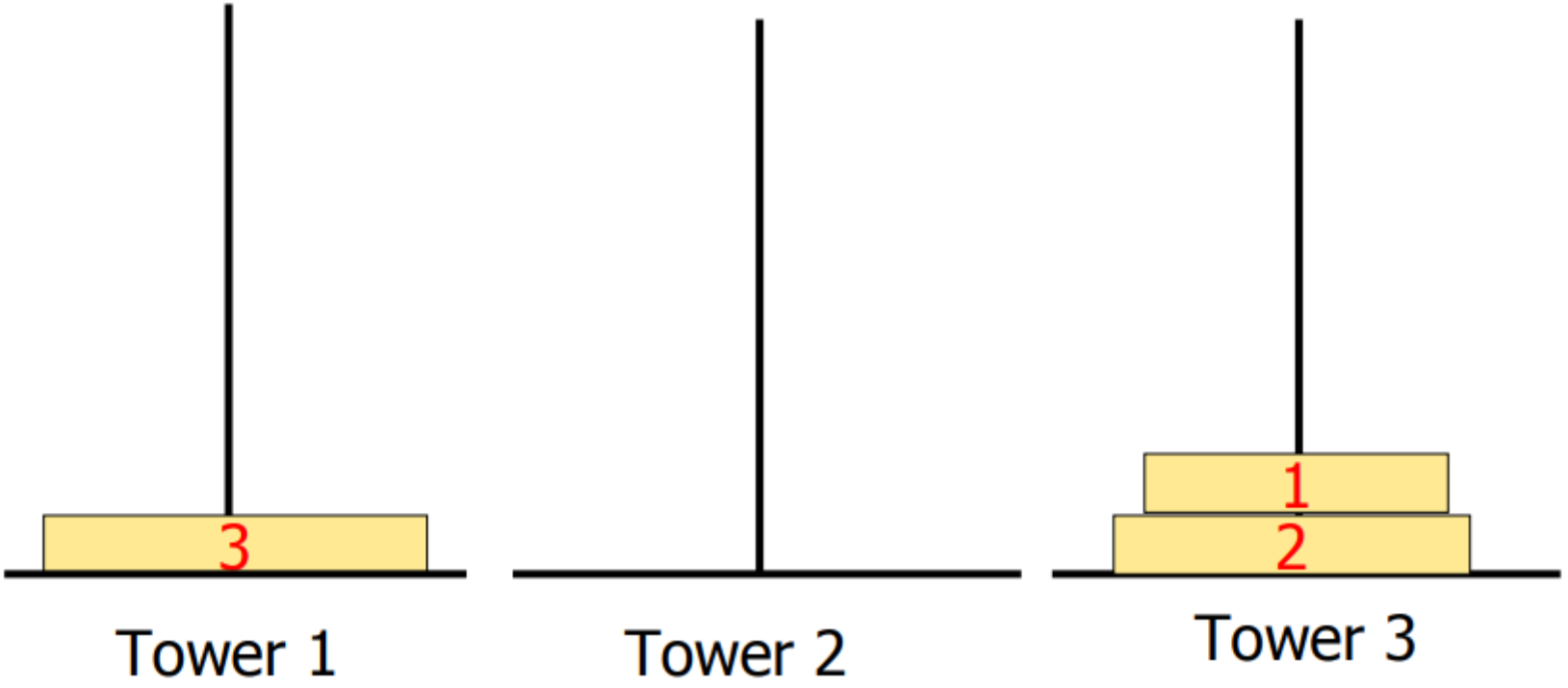
- Mission: Move the disks from tower1 to tower2



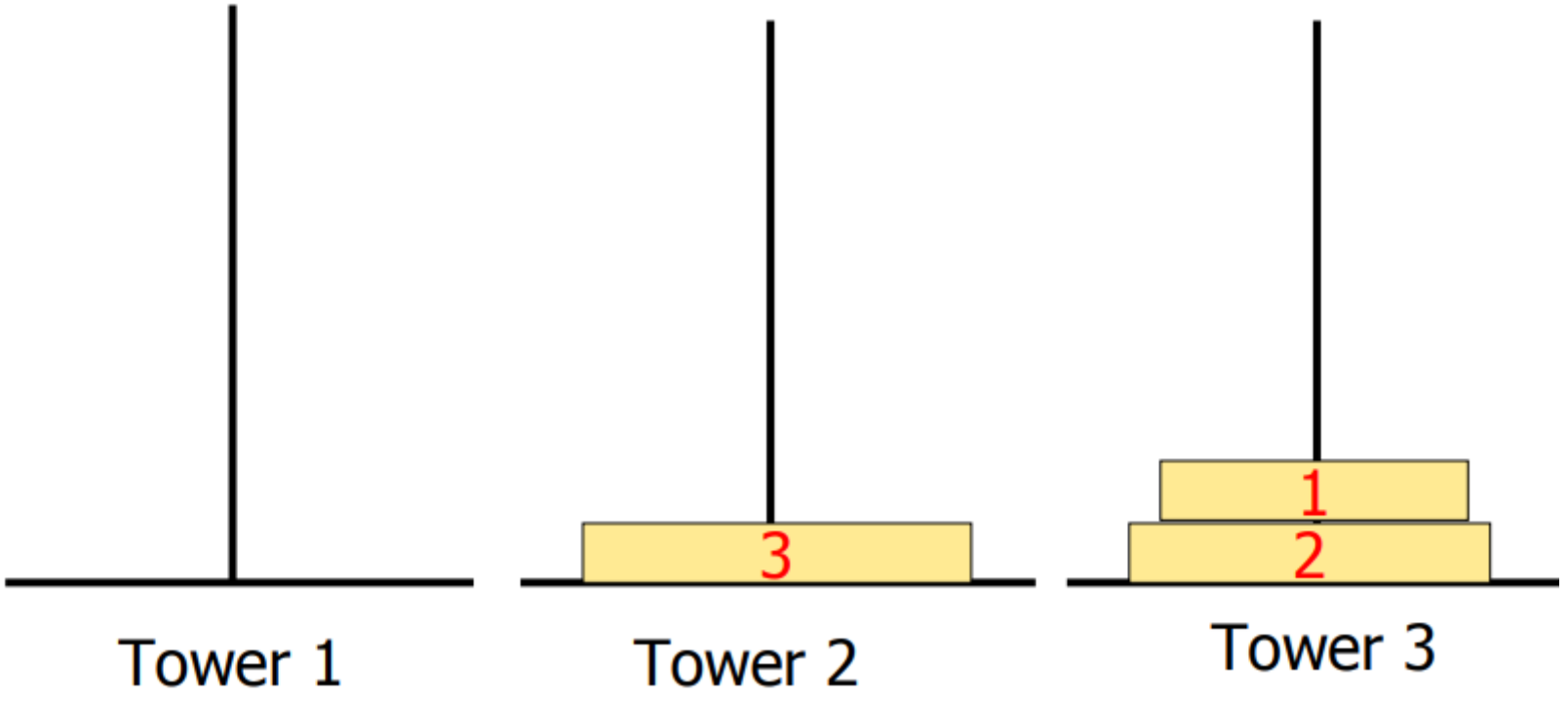
- Mission: Move the disks from tower1 to tower2



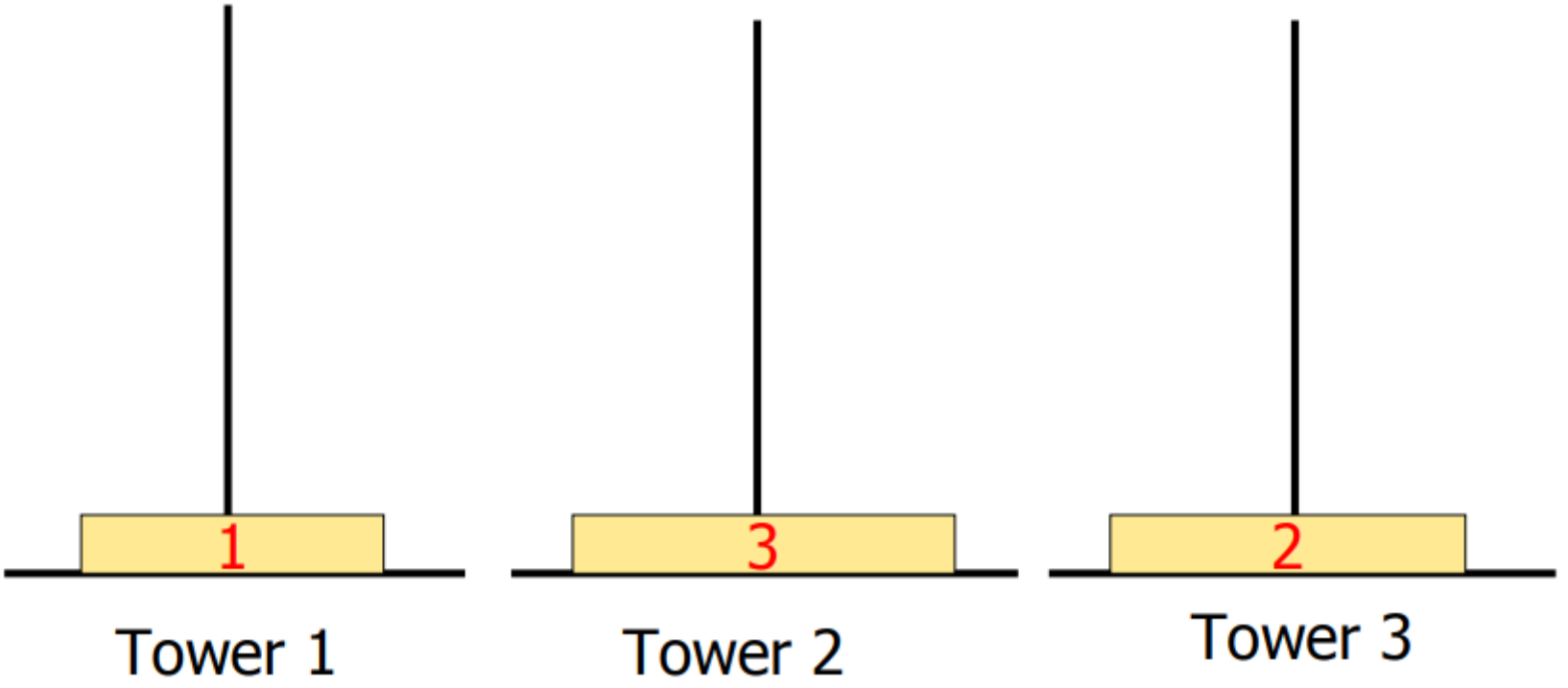
- Mission: Move the disks from tower1 to tower2



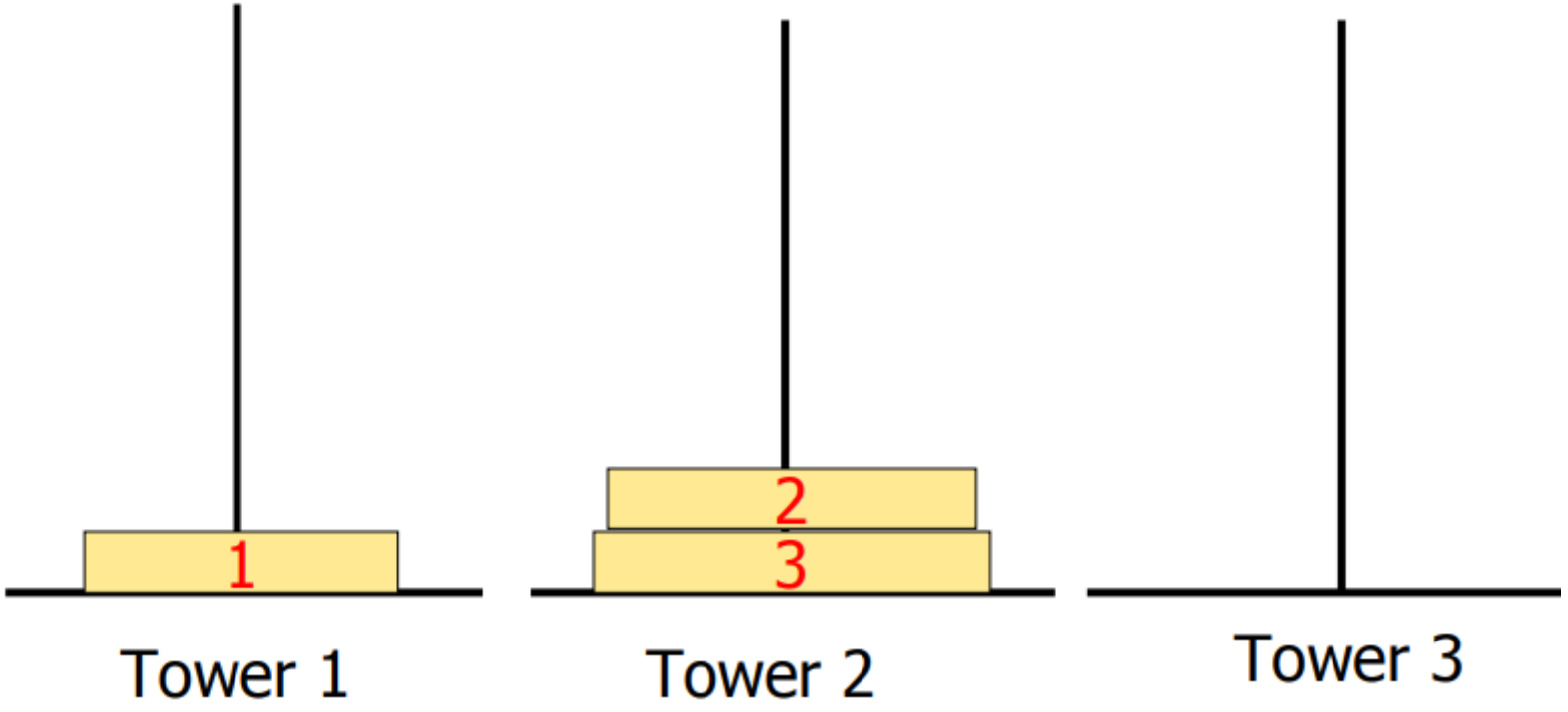
- Mission: Move the disks from tower1 to tower2



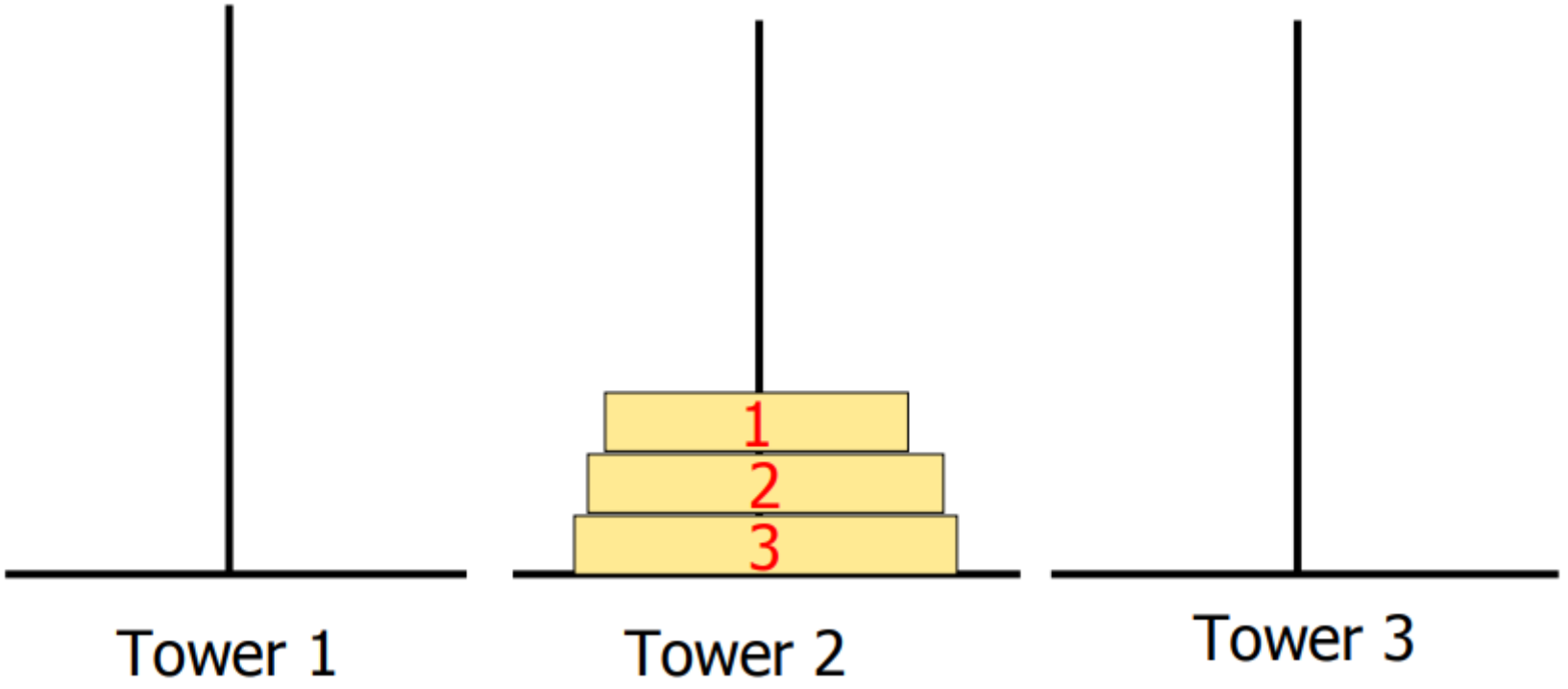
- Mission: Move the disks from tower1 to tower2



- Mission: Move the disks from tower1 to tower2



- Mission: Move the disks from tower1 to tower2



Formula: Function call= 2^(n+1) = value-1, Moves= (2^n)=value -1