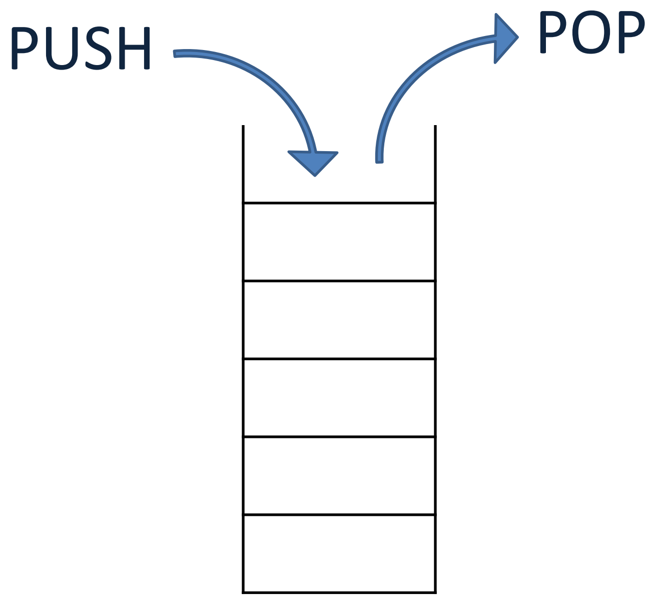
STACK

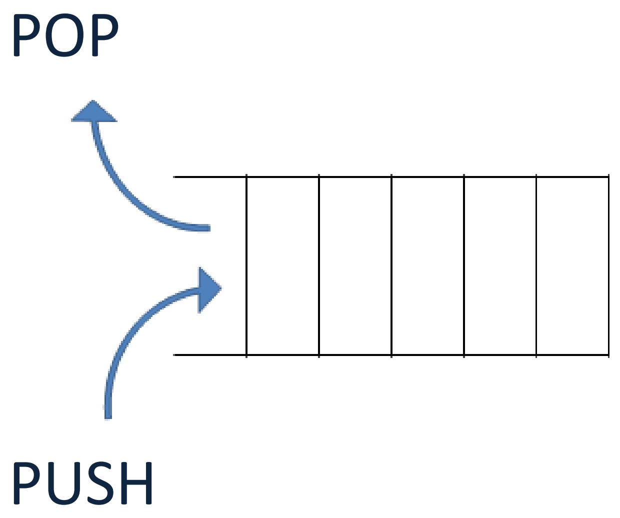
STACK

- Linear Data Structure
- Homogenous Data Structure
- ADT i.e. Abstract Data Type
- Elements are inserted and deleted only from one end, called top of stack.
- Stack is collection of elements that follows the LIFO order. **LIFO stands for Last In First Out**

STACK

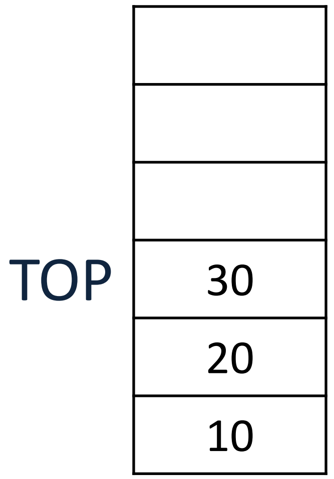
- In computer's memory stacks can be represented as a linear array.
- Insertion of element is called **PUSH** and deletion is called **POP**.
- These operations can be done from only one end of stack and we call that position as **top**.





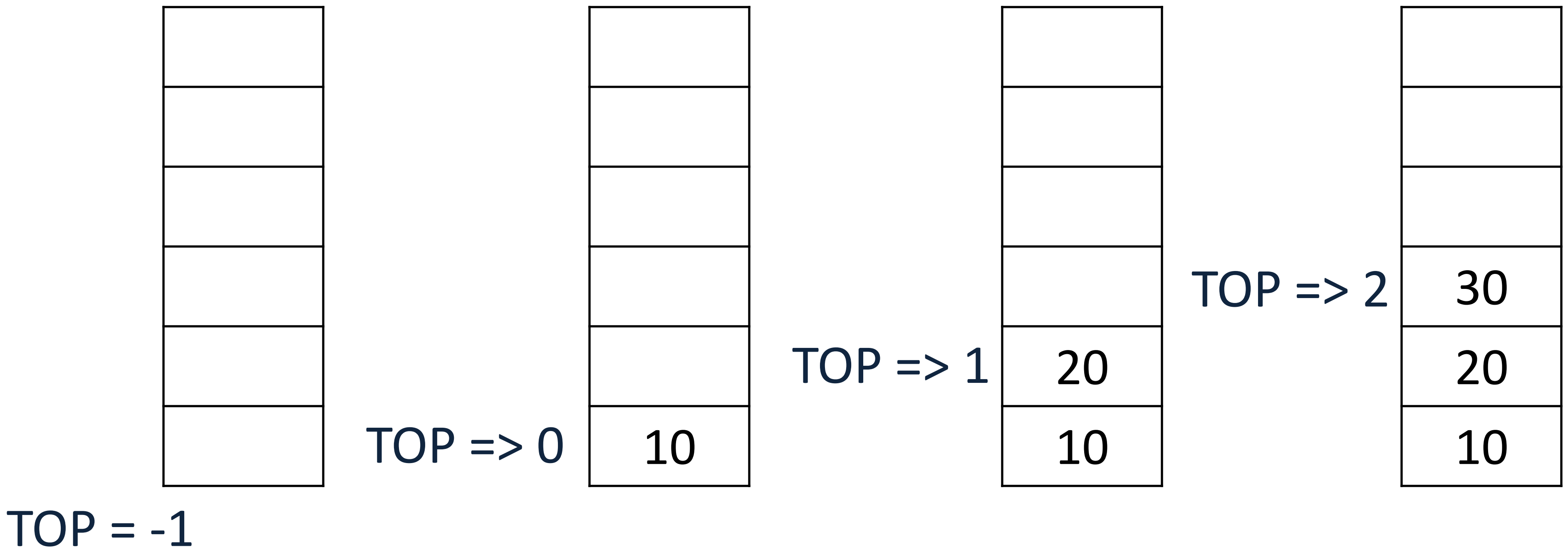
ARRAY REPRESENTATION OF STACK

- Every stack has a variable TOP associated with it.
- TOP is used to point to the top most element of the stack. It is the position from where the element will be added or deleted.
- Variable MAX will be used to store the maximum number of elements that the stack can hold.
- Initial value of TOP is -1 which indicates that stack is empty.
- TOP = MAX-1 indicates that stack is full.



PUSH OPERATION

- Used to insert an element into the stack.
- The new element is added at the topmost position of the stack.



TAKE CARE:

- **TOP = TOP + 1**
- **Check for TOP = MAX – 1 (Stack is full or not)**

ALGORITHM FOR PUSH OPERATION

Algorithm:

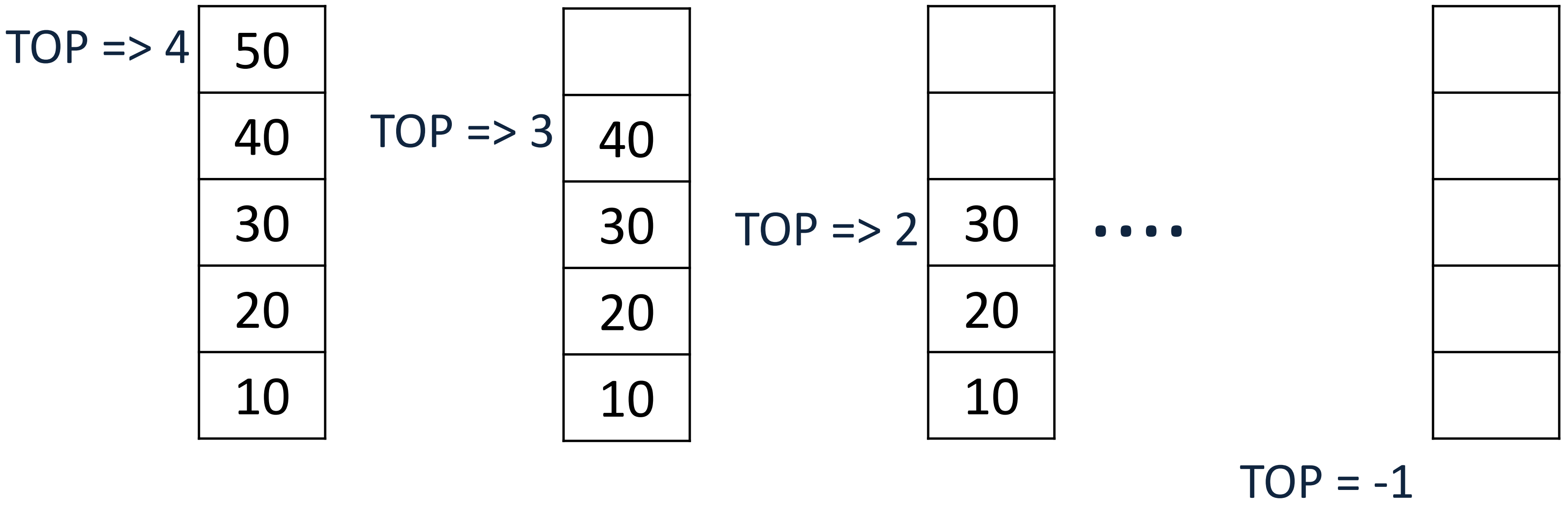
PUSH (VAL): This algorithm inserts an element to the top of the stack.

- S is stack which contains MAX elements.
- TOP is a pointer which points to the top element of the Stack.

```
Step 1) [Check for stack overflow]
        IF TOP = MAX - 1 then
        Write "OVERFLOW"
        Return
Step 2) [Increment TOP]
        TOP = TOP + 1
Step 3) [Insert Element]
        S[TOP] = VALUE
Step 4) [Finished]
        Return
```

POP OPERATION

- Used to delete an element from the stack.
- The element is deleted from the topmost position of the stack.



TAKE CARE:

- **Access element at top of the stack**
- **TOP = TOP - 1**
- **Check for TOP = - 1 (Stack is empty or not)**

ALGORITHM FOR POP OPERATION

Algorithm:

POP (): This algorithm deletes an element at top of the stack.

- S is stack which contains MAX elements.
- TOP is a pointer which points to the top element of the Stack.

```
Step 1) [Check for stack underflow]
            IF TOP = - 1 then
            Write "UNDERFLOW"
            Return
Step 2) [Access top most Element]
            VALUE = S[TOP]
Step 3) [Decrement TOP]
            TOP = TOP - 1
Step 4) [Finished]
            Return VALUE
```

APPLICATIONS OF STACK

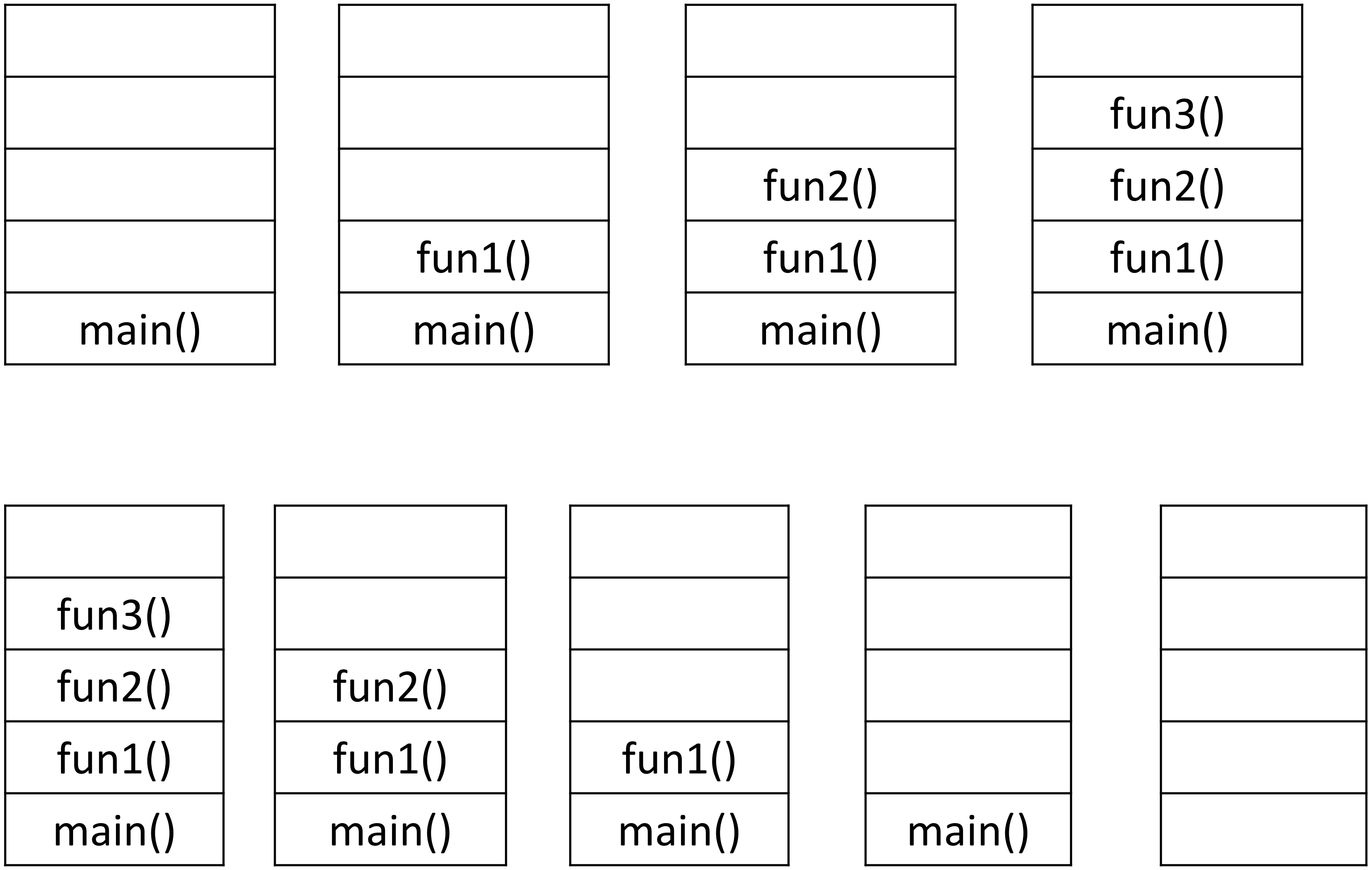
- Storing function calls
- Undo functionality
- Parentheses Checker
- Expression conversion
- Expression evaluation
- Recursion
- Tower of Hanoi

STORING FUNCTION CALLS

Example:

```
int main(){
    ....
    fun1();
    ....
}
void fun1(){
    ....
    fun2();
    ....
}
void fun2(){
    ....
    fun3();
    ....
}
void fun3(){
    ....
    fun4();
    ....
}
void fun4(){
    //code
}
```

STORING FUNCTION CALLS

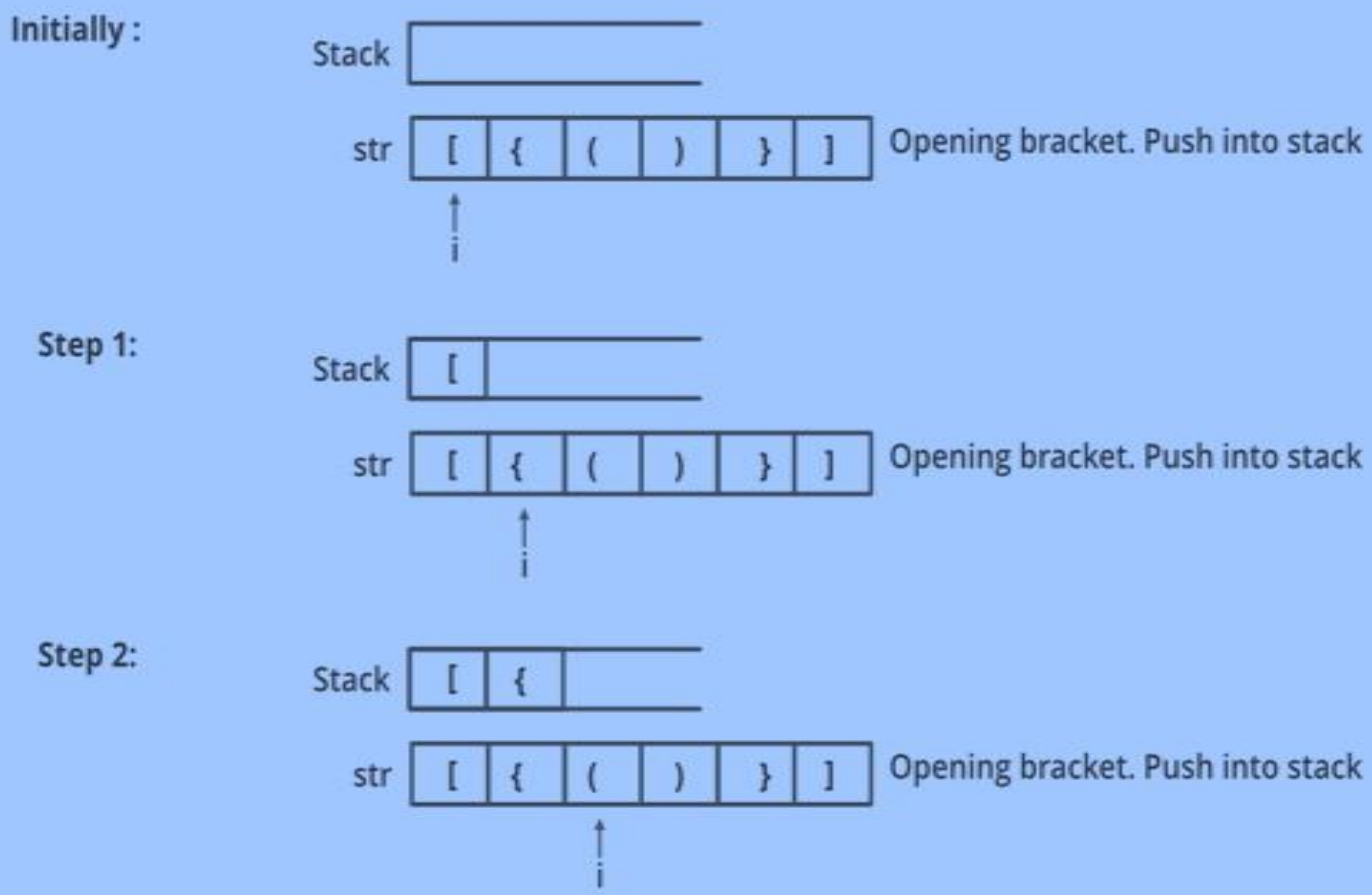


EXERCISE

- Describe the output of the following series of stack operations:

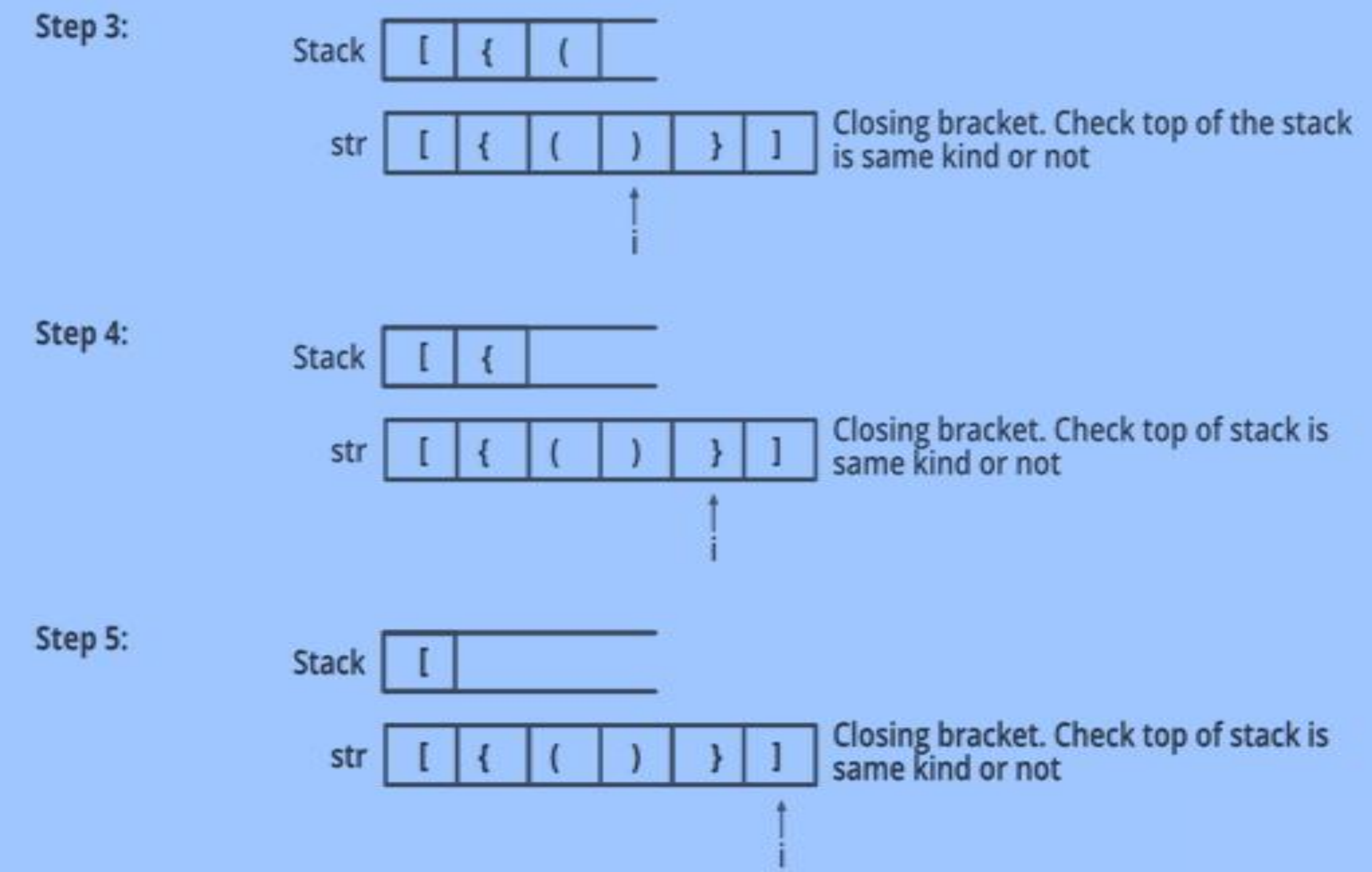
  Push(8), Push(3), Pop(), Push(2), Push(5), Pop(), Pop(), Push(9), Push(1)

PARENTHESES CHECKER



Initially :
Stack
str [{ ( ) }]
Opening bracket. Push into stack
i
Step 1:
Stack [
str [{ ( ) }]
Opening bracket. Push into stack
i
Step 2:
Stack [{
str [{ ( ) }]
Opening bracket. Push into stack
i

PARENTHESES CHECKER



Step 3:

Closing bracket. Check top of the stack is same kind or not

Step 4:

Closing bracket. Check top of stack is same kind or not

Step 5:

Closing bracket. Check top of stack is same kind or not

UNDO FUNCTIONALITY

- Maintain two stack say UNDO stack and REDO stack.
- Use UNDO stack to store all the operations that have been processed in the text editor.
- whenever an user encounter undo operation, pop the top of the element from UNDO stack and push it to REDO stack.
- Then, if the user want a redo operation pop the top of the element of REDO stack and push it to UNDO,
- If user performs a new operation then make the stack REDO empty.

EXPRESSION

- An expression is a collection of operators and operands that represents a specific value.
- Operator is a symbol which performs a particular task.
- Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable.
- Based on the operator position, expressions are divided into THREE types. They are as follows...
  1. Infix Notation
  2. Postfix Notation
  3. Prefix Notation

INFIX NOTATION

- In expression using infix notation, the operator is placed in between the operands.
- **Structure: Operand1 Operator Operand2**
- **Example: A+B**

- For computers parsing of infix expression is difficult as lot of information is required to evaluate the expression.
- Information is needed about operator precedence and associativity rules, and brackets which override these rules.
- Computers work more efficiently with expressions written using prefix and postfix notations.

PREFIX NOTATION

- In prefix notation, the operator is placed before the operands.
- Also known as Polish Notation.
- **Structure: Operator Operand1 Operand2**
- **Example: +AB**

- **Evaluation of Prefix Expression:**

  Operator is applied to the operands that are present immediately on the right of the operator.

  Example: *+ABC

POSTFIX NOTATION

- In postfix notation, the operator is placed after the operands.
- Also known as Reverse Polish Notation(RPN).
- **Structure: Operand1 Operand2 Operator**
- **Example: AB+**

- **Evaluation of Prefix Expression:**

  Operator is applied to the operands that are present immediately on the left of the operator.

  Example: AB+C*

- Prefix and postfix expressions are evaluated from left to right.
- No need to follow operator precedence rules and associativity.
- Prefix and postfix notations are developed with aim to create parenthesis free expressions.

EXPRESSION CONVERSION

- Following conversion can be possible:

  1. Infix to Postfix
  2. Infix to Prefix
  3. Prefix to Postfix
  4. Postfix to Prefix
  5. Prefix to Infix
  6. Postfix to Infix

- We consider five binary operators:

OPERATOR	**PRECEDENCE**	**ASSOCIATIVITY**
Exponentiation ($ or ↑ or ^)	Highest	Right to Left
* , /	Next Highest	Left to Right
+, -	Lowest	Left to Right

EXPRESSION CONVERSION

- Process for conversion:

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

- ***INFIX to POSTFIX:***

  A + B * C

  ✓ Here Operators are: **+ , ***

  ✓ Order of Operators according to their preference : *** , +**

  A + BC *

  ABC*+

EXPRESSION CONVERSION

- ***INFIX to PREFIX:***

  A + B * C

  ✓ Here Operators are: **+ , ***

  ✓ Order of Operators according to their preference : *** , +**

  A + *BC

  +A*BC

  Another Example:

  (A + B) * C

  What is the postfix and prefix expression???

  Postfix: AB+C*

  Prefix: *+ABC

MORE EXAMPLES

1. A + B * C + D
2. (A + B) * (C + D)
3. P * Q + R / S
4. (J – K / L) * (M / N – O)
5. (A – B) * (C + D)
6. (P + Q) / (R + S) – (T * U)
7. 14 / 7 * 3 – 4 + 9 / 2

INFIX TO POSTFIX USING STACK

1. Scan the infix expression from left to right.
2. If the scanned character is an **operand**, output it.
3. If the scanned character is an **operator**,

   3.1 If the **precedence of the scanned operator** is **greater** than the **precedence of the operator in the stack** (or the stack is empty or the stack contains a '('), **push** it.

   3.2 Else, **Pop all the operators** from the stack which are **greater than or equal to in precedence** than that of the scanned operator. After doing that **Push the scanned operator** to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an **'('**, **push** it to the stack.
5. If the scanned character is an **')'**, **pop** the stack and output it **until a '(' is encountered**, and discard both the parenthesis.

INFIX TO POSTFIX USING STACK

6. Repeat steps 2-5 until infix expression is scanned.
7. **Pop** from the stack **and output it** until stack is not empty.
8. Print the output.

EXAMPLE

1. **A * B + C**

SCANNED CHARACTER	OPERATOR STACK	POSTFIX STRING (OUTPUT)
A		A
*	*	A
B	*	A B
+	+	A B * **{POP * before pushing the +}**
C	+	A B * C
		A B * C +

EXAMPLE

2. A * (B + C)

SCANNED CHARACTER	OPERATOR STACK	POSTFIX STRING (OUTPUT)
A		A
*	*	A
(	* (	A
B	* (	A B
+	* (+	A B **{PUSH +}**
C	* (+	A B C
)	*	A B C + **{POP FROM STACK UNTIL) ENCOUNTERED}**
		A B C + *

EXAMPLE

3. A * (B + C * D) + E

SCANNED CHARACTER	OPERATOR STACK	POSTFIX STRING (OUTPUT)
A		A
*	*	A
(	* (	A
B	* (	A B
+	* (+	A B
C	* (+	A B C
*	* (+ *	A B C
D	* (+ *	A B C D
)	*	A B C D * +
+	+	A B C D * + *
E	+	A B C D * + * E
		A B C D * + * E +

MORE EXAMPLES

1. 3 + 4 * 5 / 6
2. ((A + B) — C * (D / E)) + F
3. (300 + 23) * (43 – 21) / (84 + 7)
4. x + y * z / w - v
5. (4 + 8) * (6 – 5) / ((3 – 2) * (2 + 2))

EXPRESSION EVALUATION

- EVALUATION OF INFIX EXPRESSION:

  a + b / c –d * e   where a=10, b=6, c=2, d=8, e=13

  10 + 6 / 2 – 8 * 13

  10 + 3 – 8 * 13

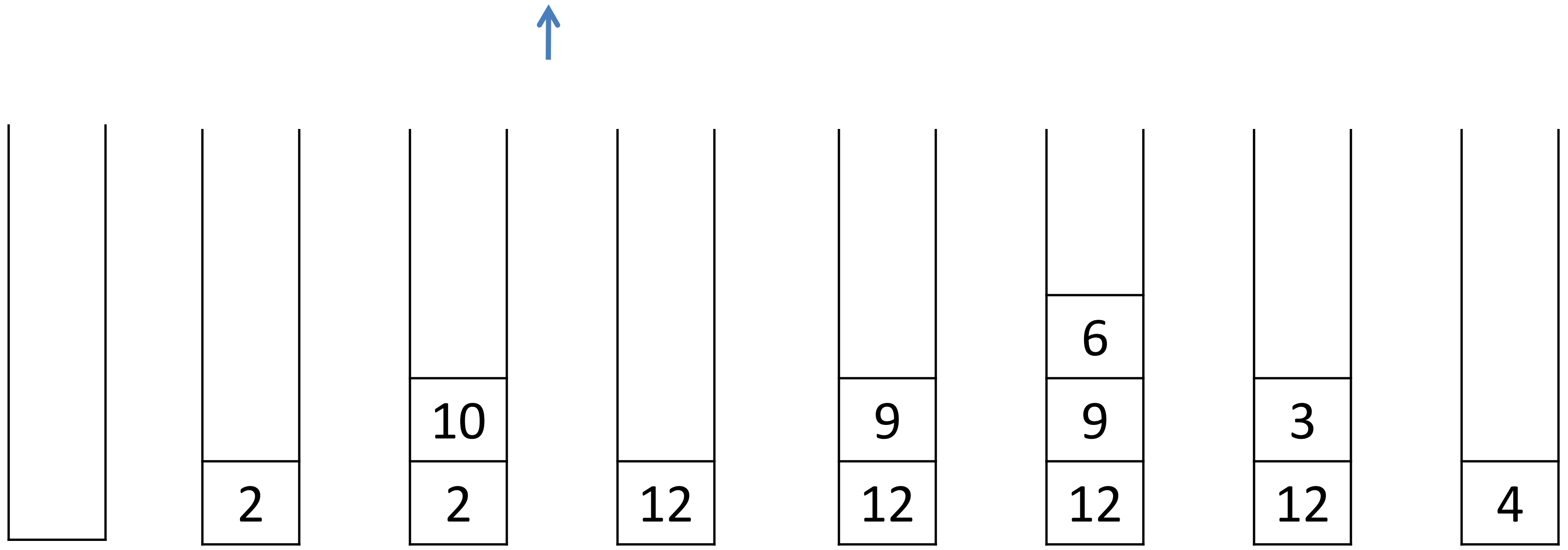
  10 + 3 - 104

  13 - 104

  - 91

POSTFIX EXPRESSION EVALUATION

- Postfix expression: 2 10 + 9 6 - /



POSTFIX EXPRESSION EVALUATION

1. Create a stack to store operands (or values) and scan the postfix expression from left to right.
2. If the scanned element is an operand, push it into the stack.
3. If the scanned element is an operator(O),
    a) Pop two operands from stack as A and B.
    b) Evaluate B O A where A is top most element and B is element below the A
    c) Push the result back to the stack.
4. When the expression is ended, the number in the stack is the final answer.

POSTFIX EXPRESSION EVALUATION

EVALUATE POSTFIX EXPRESSION: 2 7 * 18 - 6 +

SCANNED ELEMENT	STACK	CALCULATION
2	2	
7	2, 7	
*	14	2 * 7
18	14, 18	
-	-4	14 - 18
6	-4, 6	
+	2	-4 + 6

POSTFIX EXPRESSION EVALUATION

EVALUATE POSTFIX EXPRESSION: 30, 23, +, 43, 21, -, *, 6, 5, +, /

SCANNED ELEMENT	STACK	CALCULATION
30	30	
23	30, 23	
+	53	30 + 23
43	53, 43	
21	53, 43, 21	
-	53, 22	43 - 21
*	1166	53 * 22
6	1166, 6	
5	1166, 6, 5	
+	1166, 11	6 + 5
/	106	1166 / 11

RECURSION

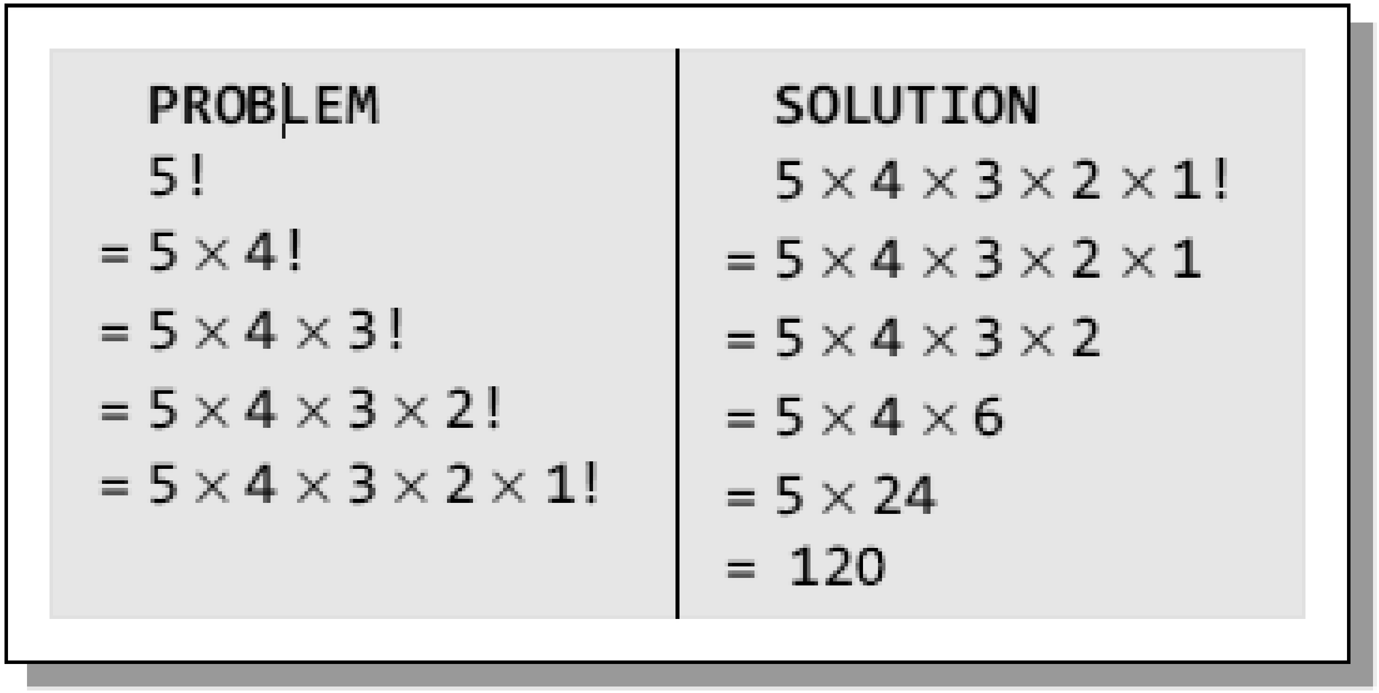
- A recursive function is defined as a function that calls itself.
- Final call does not require to call itself.
- Recursive function makes use of stack to temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases.

  1. Base case: no further calls to function itself.

  2. Recursive case: Problem is divided into subparts, function calls itself and result is obtained by combining the solutions of subparts.

RECURSION

- Example: Factorial of number (n)
- n! = n * (n–1)!
- Let us say we need to find the value of 5!



PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

RECURSION

- **Base case: n = 1**, because if n = 1, the result will be 1 as 1! = 1.
- **Recursive case:** factorial function will call itself but with a smaller value of n, i.e.

  factorial(n) = n × factorial (n–1)

THANK YOU