Data Structures and Algorithms I SCS1201 - CS

Dr. Dinuni Fernando

Senior Lecturer

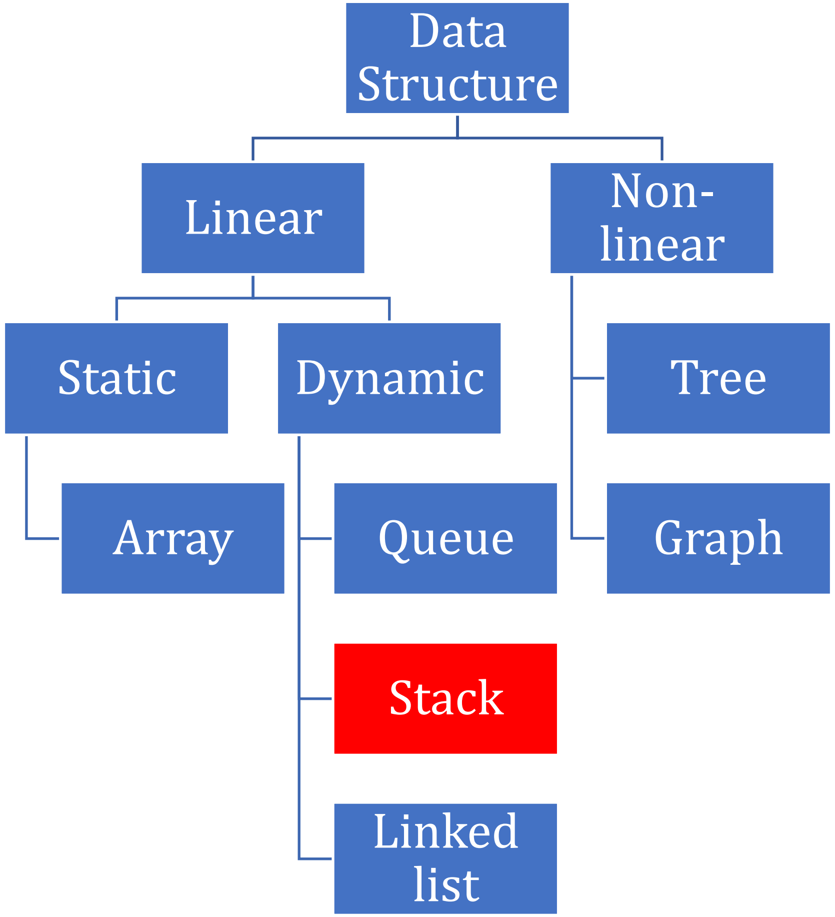
Lecture 2

Learning Objectives of the lesson

- Understand the What is stack data structure and how its being used.
- Understand and differentiate implementations of stack along with their applications

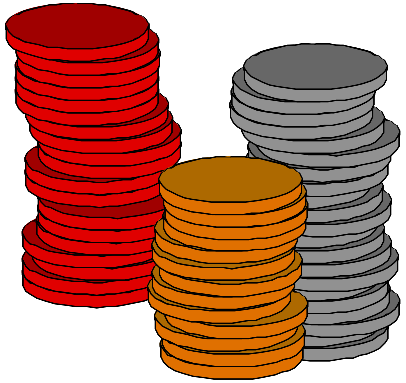
What is Stack?

Classification of Data Structure


Data Structure
Linear
Non-linear
Static
Dynamic
Tree
Array
Queue
Graph
Stack
Linked list

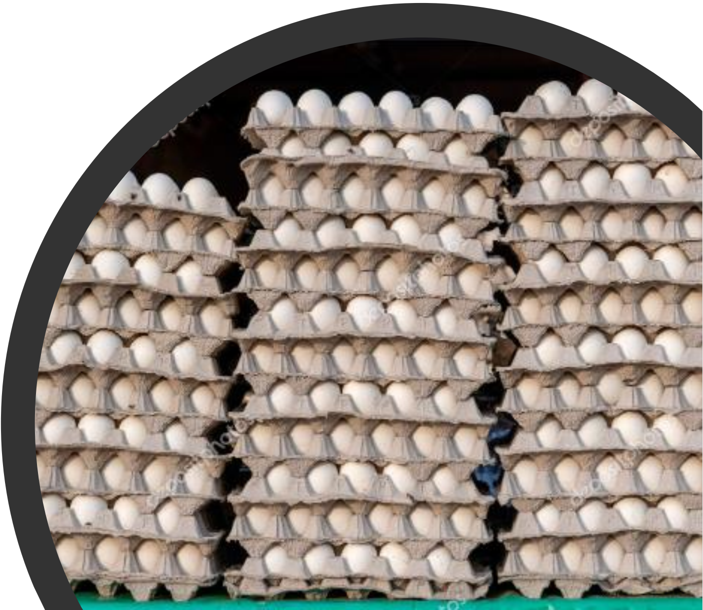
What is stack data structure ?

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle.
- It can be visualized as a stack of objects, where the last object placed on top is the first one to be removed.
- Stacks are used to store and retrieve data in a particular order.



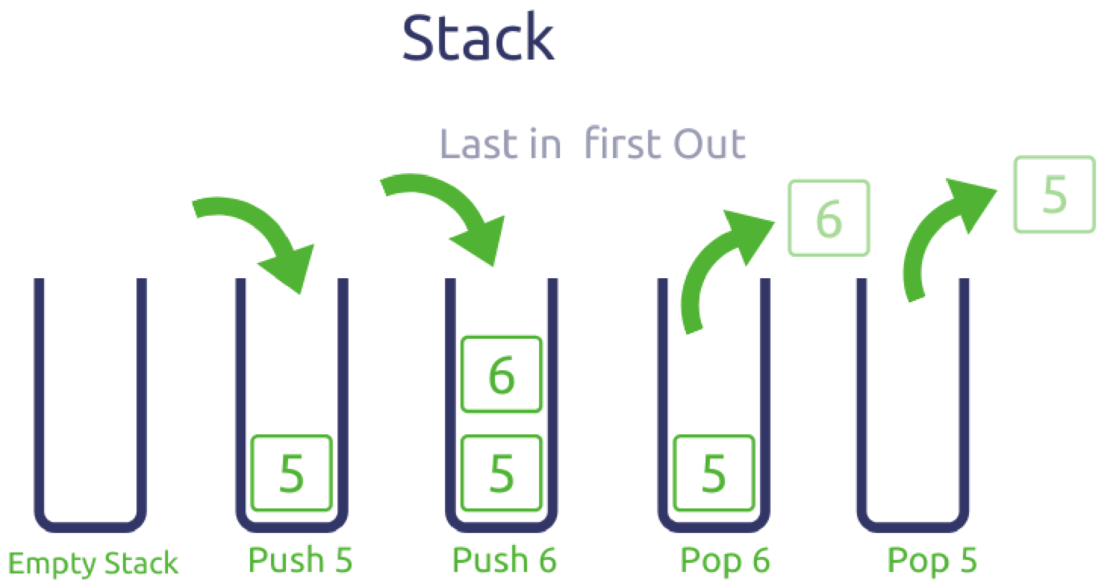
Stack : real world examples

- Stack of plates in a buffet table
- Stack of chairs
- The tennis balls in their container
- stack of trays in a cafeteria
- Stack of Books, Clothes piled



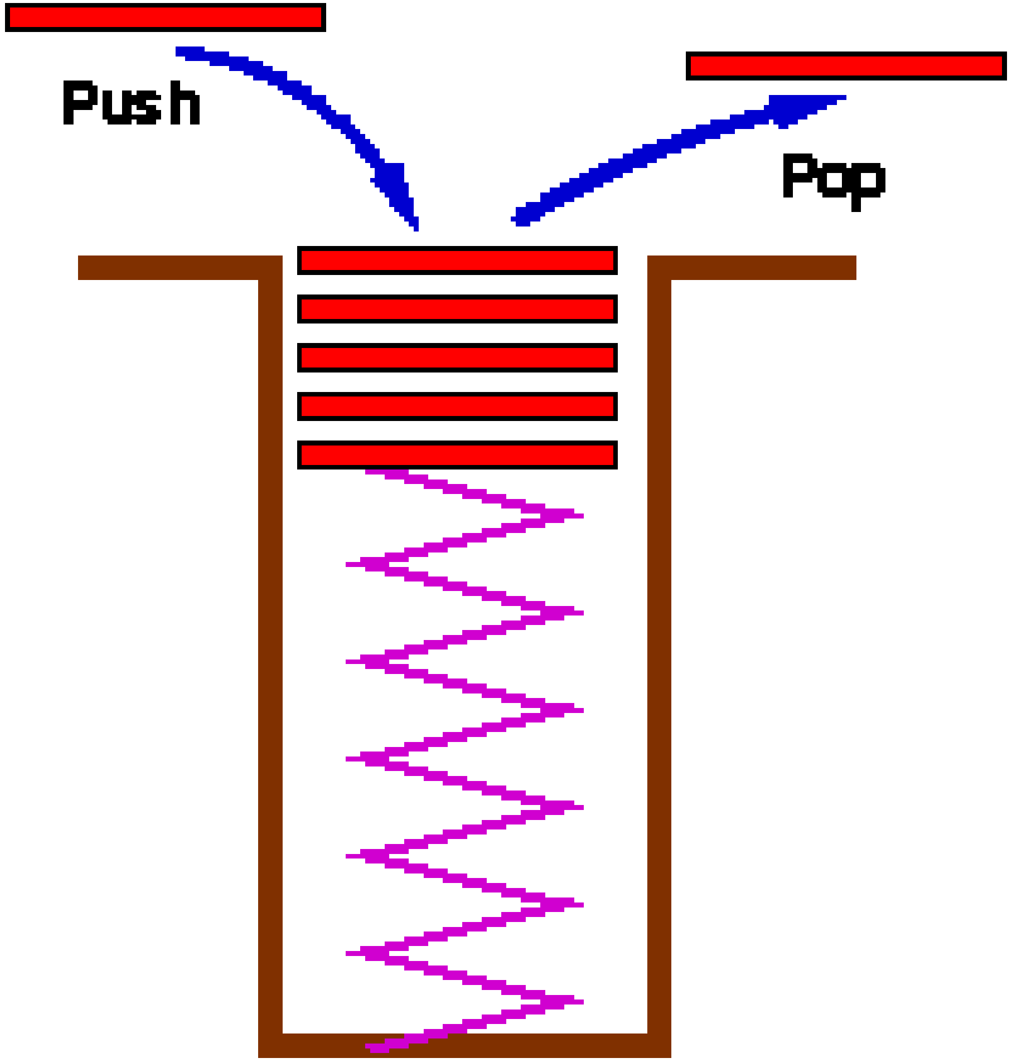
Stack

- Stores elements in an order
- Uses Last In First Out principle (LIFO)
  - Last element inserted is the first one to be remove.

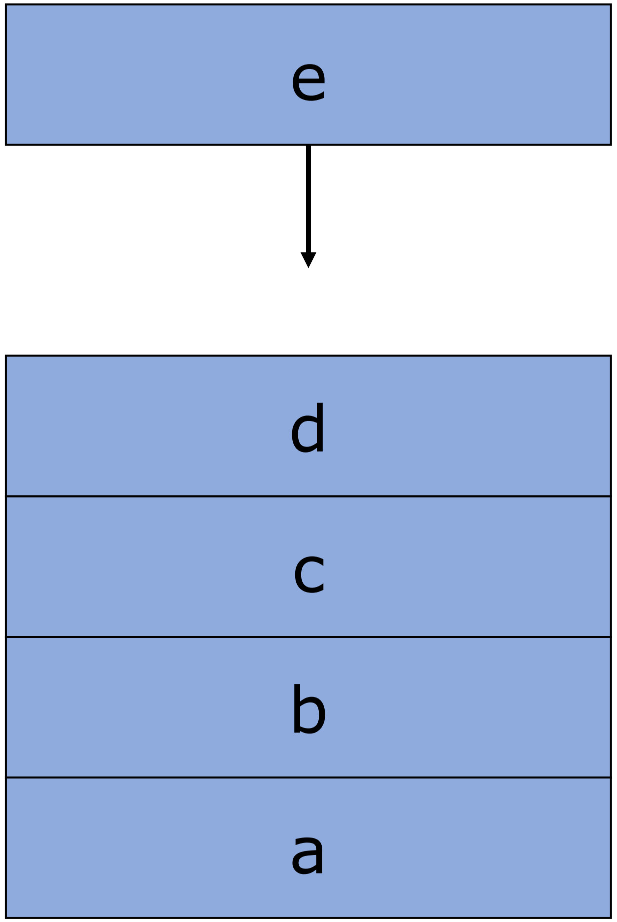


Example

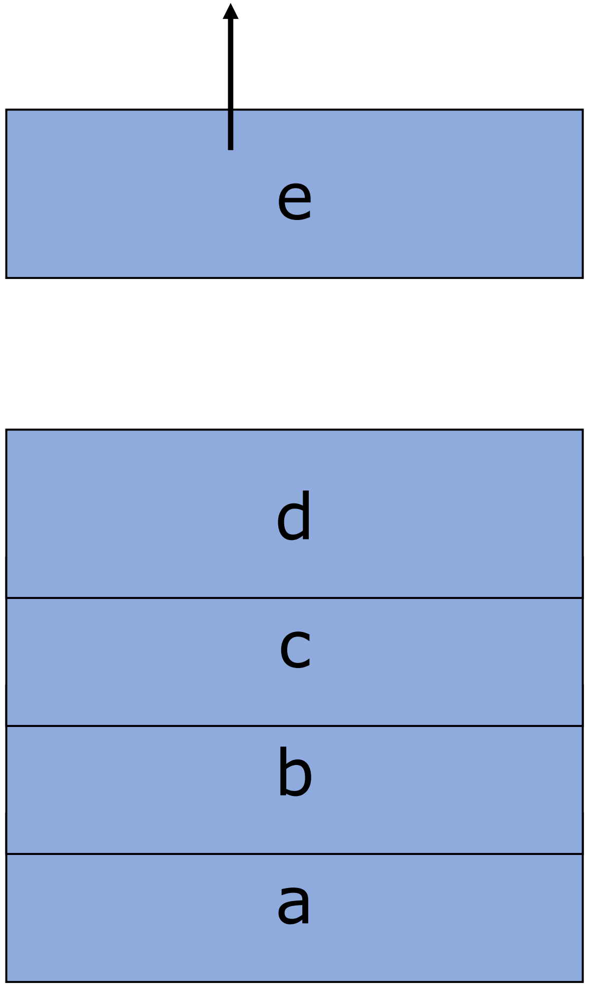
- It can be visualized as a stack of objects, where the last object placed on top is the first one to be removed.
- A common model of a stack is a plate or coin stacker. Plates are "pushed “onto to the top and "popped" off from the top.



Operations : Push



Operations : Pop



Last element that was pushed is the first to be popped.

Applications of Stack

- Back button of web browser

  History of visited web pages is pushed onto the stack and popped when “back” button is clicked.

- “Undo” functionality of a text editor.
- Reversing the order of elements in an array.
- Saving local variables when one function calls another, and this one calls another, and so on.
- Compilers use stacks to evaluate expressions.
- Stacks are great for reversing things, matching up related pairs of things, and backtracking algorithms

Applications of Stack

- Function Call/Execution: Stacks are used to manage function calls and executions. When a function is called, its return address and local variables are pushed onto the stack. Once the function completes, the stack is popped to resume execution from the previous context.

- Expression Evaluation: Stacks are utilized in evaluating arithmetic expressions, such as infix, postfix, and prefix expressions. They help in managing the order of operations and maintaining operand and operator precedence.

- Undo/Redo Functionality: Stacks can be used to implement undo and redo operations in applications. Each action is stored on the stack, allowing the user to revert or redo changes as needed.

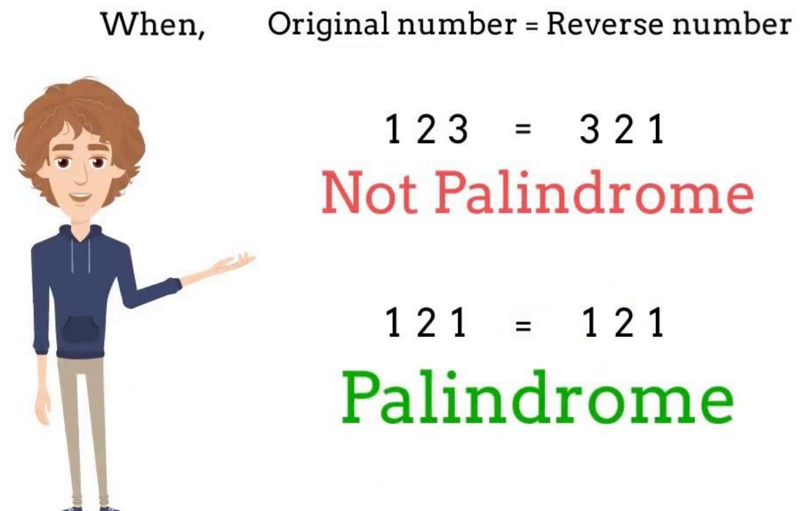
- Backtracking and Recursion: Stacks are crucial in backtracking algorithms and recursive function implementations. They help keep track of states and allow efficient exploration of paths.

- Browser History: Stacks can be employed to store the browsing history in web browsers. Each visited URL is pushed onto the stack, enabling navigation through previously visited pages.

Programming problems of Stack

- Reverse letters in a string, reverse words in a line, or reverse a list of numbers.
- Find out whether a string is a palindrome.

  Eg: madam, mom,wow,civic
- Examine a file to see if its braces { } and other operators' match.
- Convert infix expressions to postfix or prefix.



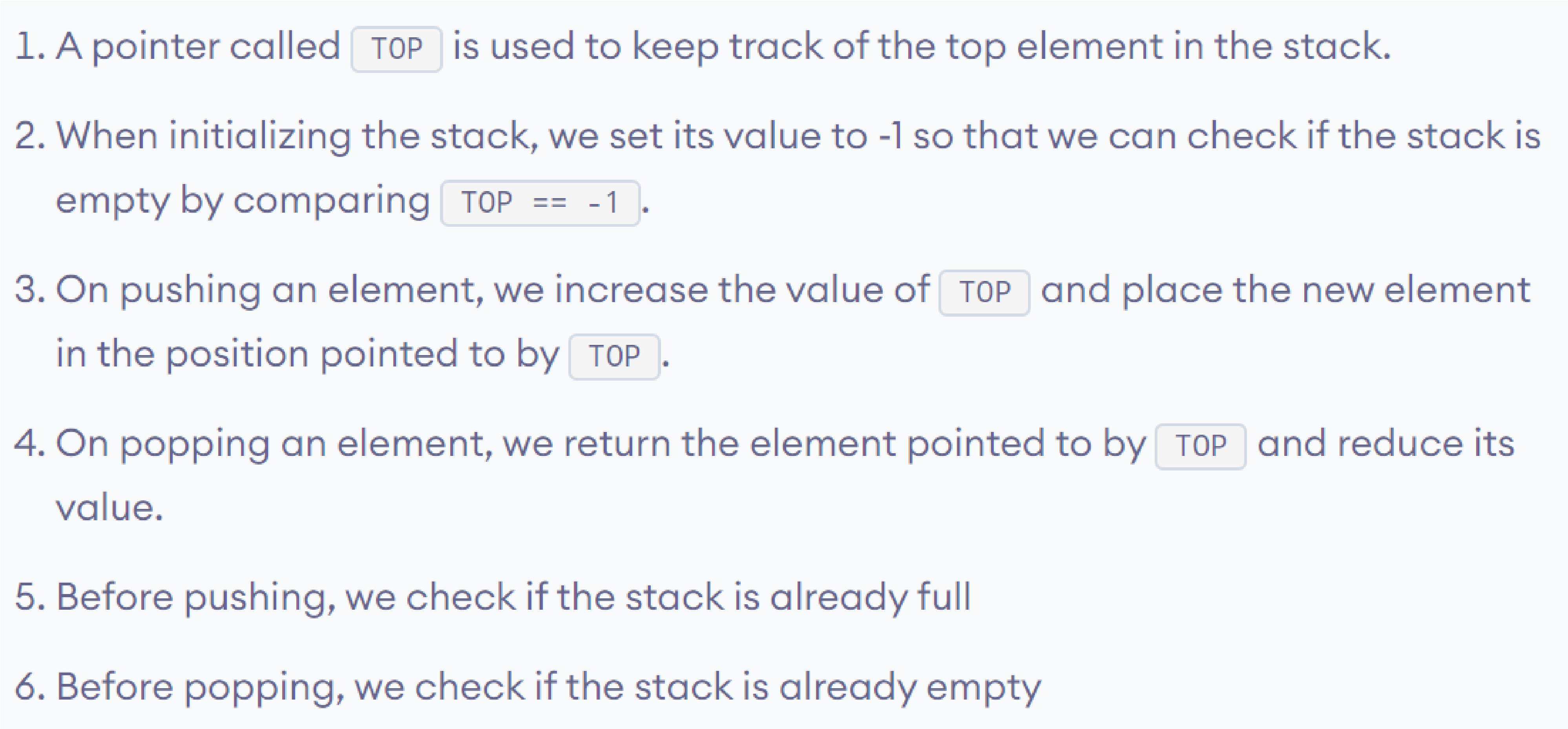
Stack : Implementation

- A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data.
- There are two ways of implementing a stack :
    - Array (Static)
    - Linked list (dynamic).

Basic operations on Stacks

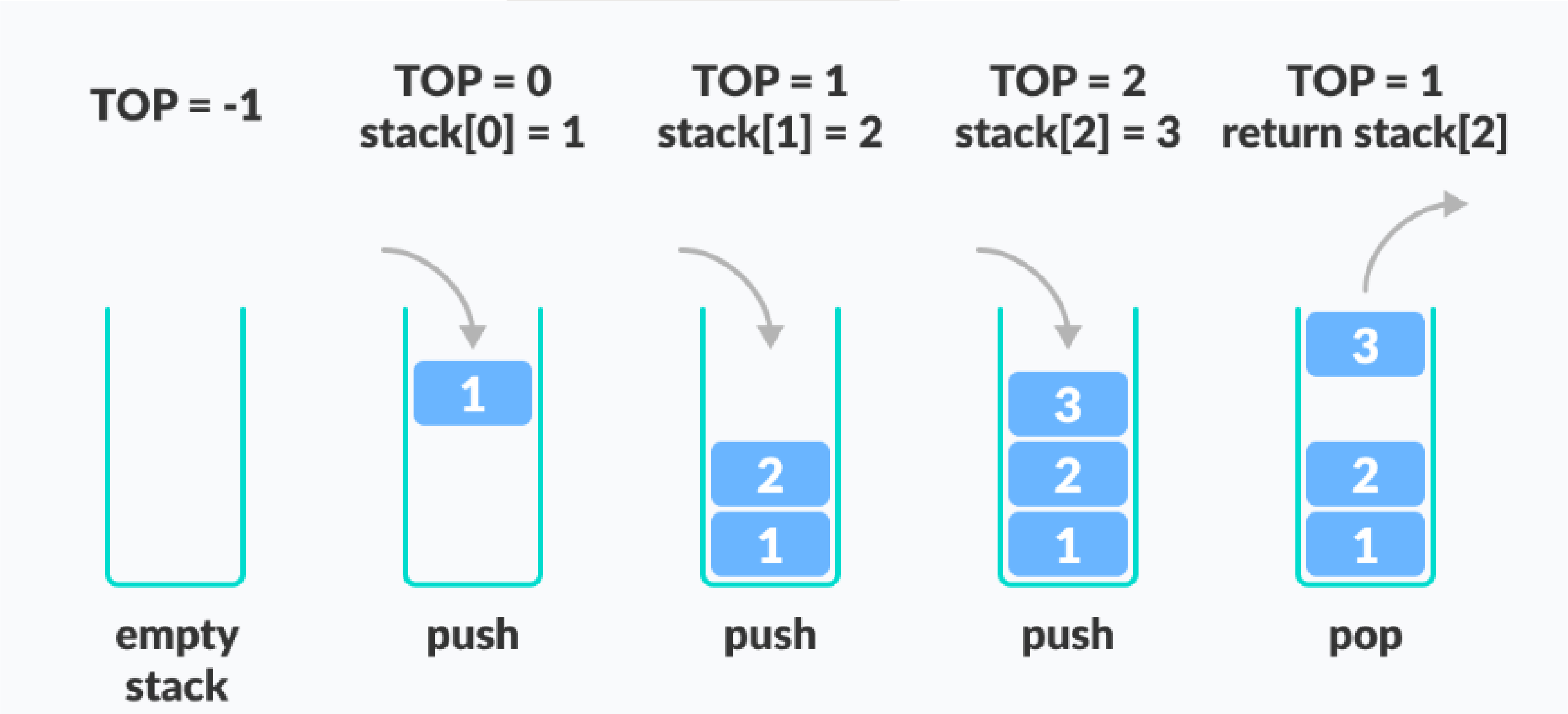
- Push : Add an element to the top of a stack.
- Pop : Remove an element from the top of a stack.
- IsEmpty : Check if the stack is empty.
- IsFull : Check if the stack is full.
- Peek : Get the value of the top element without removing it.

How a Stack Works



1. A pointer called `TOP` is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
3. On pushing an element, we increase the value of `TOP` and place the new element in the position pointed to by `TOP`.
4. On popping an element, we return the element pointed to by `TOP` and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty

How a Stack Works ?



Stack Implementation

- Implementation approaches
  - Static Implementation
  - Dynamic Implementation

- Static Implementation
  - Stacks have fixed size and are implemented as arrays.
  - Inefficient for memory utilization.

- Dynamic Implementation
    - Stack grow as needed and implemented as linked list.
    - Implementation done through pointers.
    - Memory is efficiently utilized with dynamic implementations.

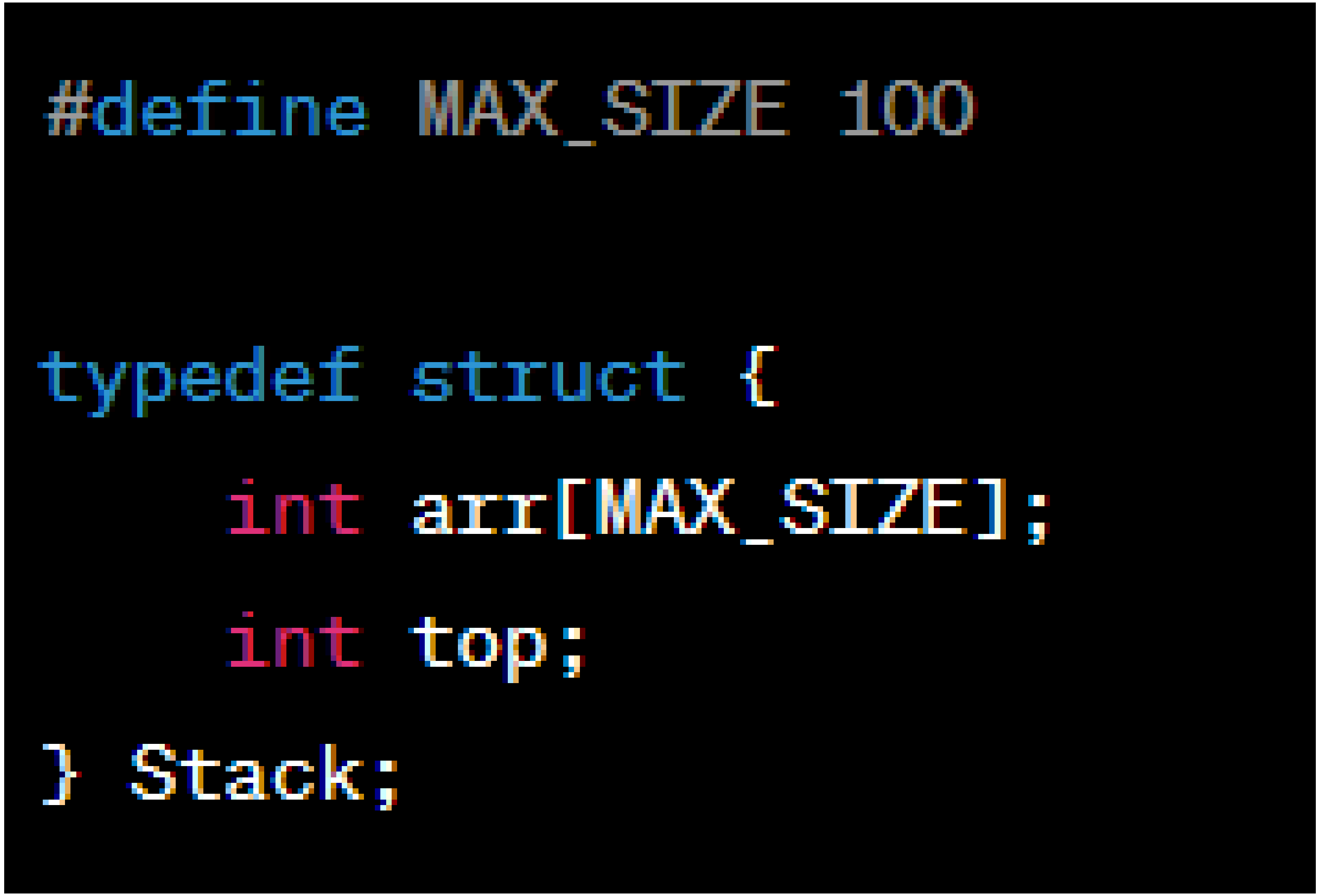
Terminology

- TOP : A pointer that points the top element in the stack.
- Stack Underflow : when there is no element in the stack, the status of stack is known as stack underflow.
- Stack Overflow : When the stack contains equal number of elements as per its capacity and no more elements can be added, the status of the stack is known as stack overflow.

Stack representation

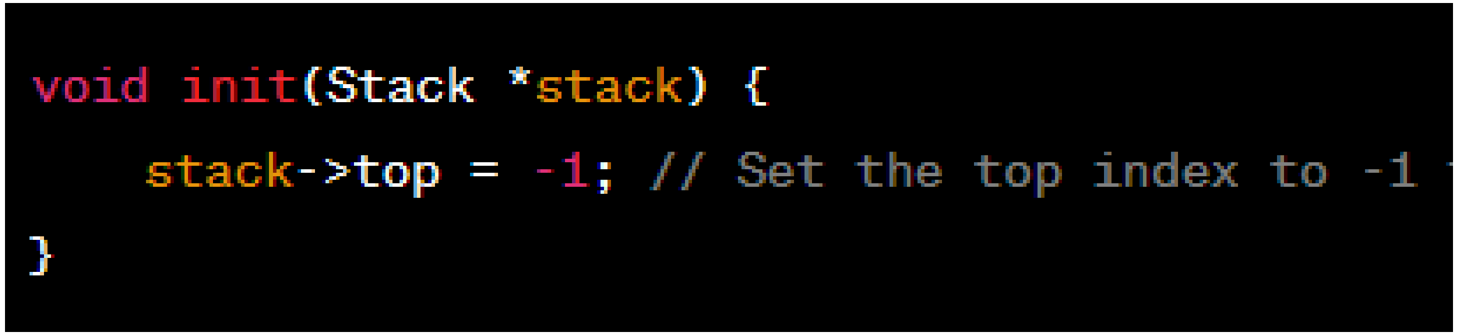
- Consider a stack with 6 elements capacity, This is called as the size of the stack.
- The number of elements to be added should not exceed the maximum size of the stack.
- If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition.
- Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition

Define the Stack structure



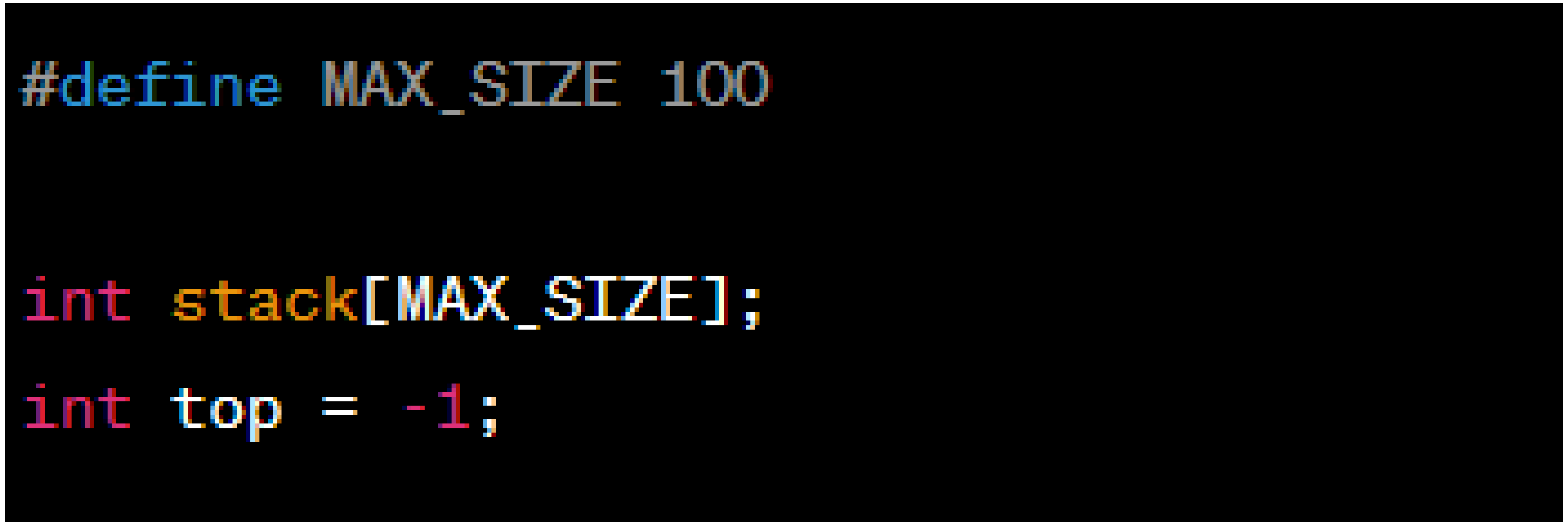
```
#define MAX_SIZE 100

typedef struct {
    int arr[MAX_SIZE];
    int top;
} Stack;
```

Initialize the Stack structure



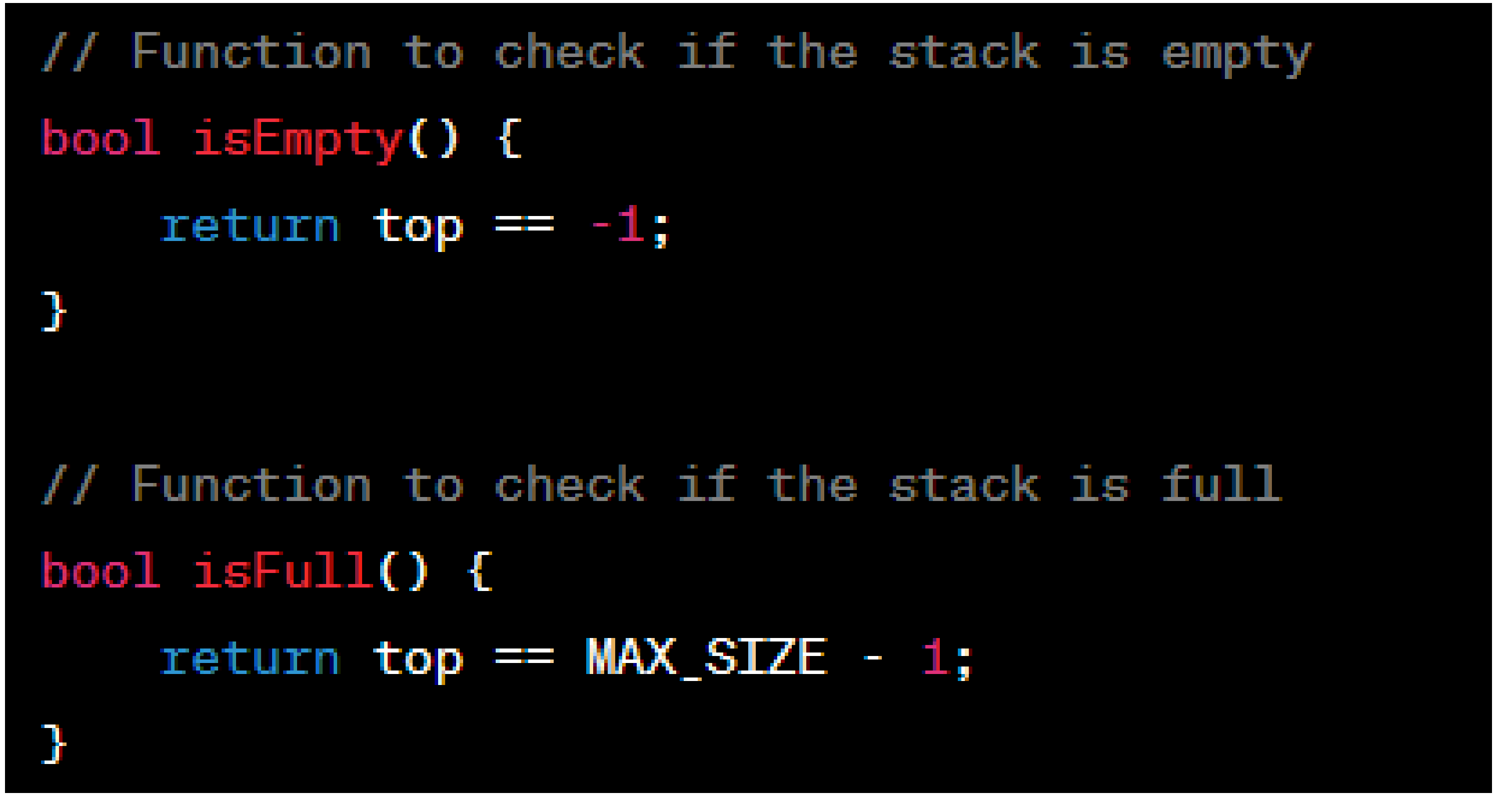
```
void init(Stack *stack) {
    stack->top = -1; // Set the top index to -1
}
```

Define the Stack using array



```c
#define MAX_SIZE 100


int stack[MAX_SIZE];
int top = -1;
```

Initialize the Stack structure

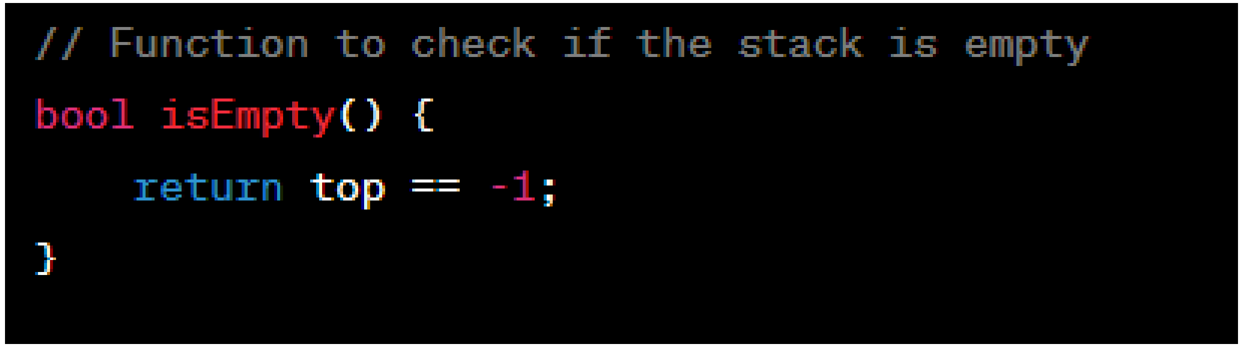


```
// Function to check if the stack is empty
bool isEmpty() {
    return top == -1;
}

// Function to check if the stack is full
bool isFull() {
    return top == MAX_SIZE - 1;
}
```

isEmpty()

```
Begin Procedure IsEmpty
        If top is less than 1
                return True
        else
                return False
        endif
End Procedure
```

Check if the stack is empty



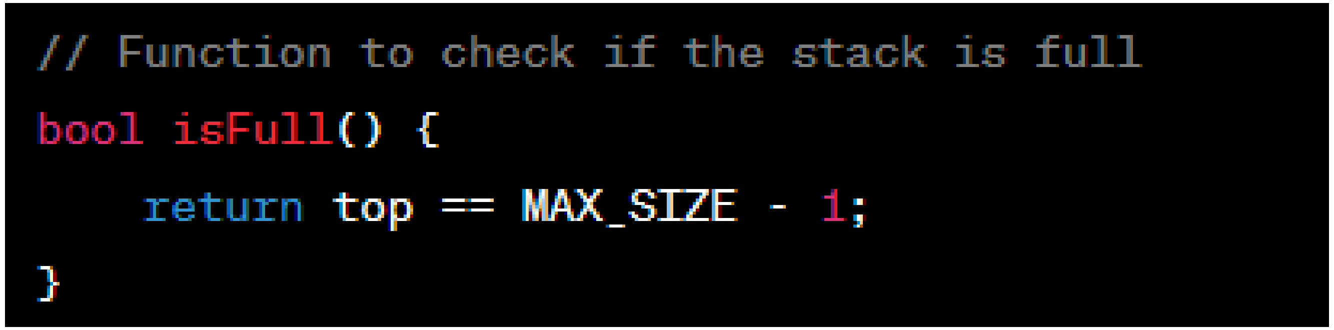
```
// Function to check if the stack is empty
bool isEmpty() {
    return top == -1;
}
```

```
Boolean Isempty()
{
        if (top==-1)
                return true;
        else
                return false;
}
```

isFull()

Check if the stack is empty

```
Begin Procedure IsFull
    If top is equal to MAXSIZE-1
        return True
    else
        return False
    endif
End procedure
```

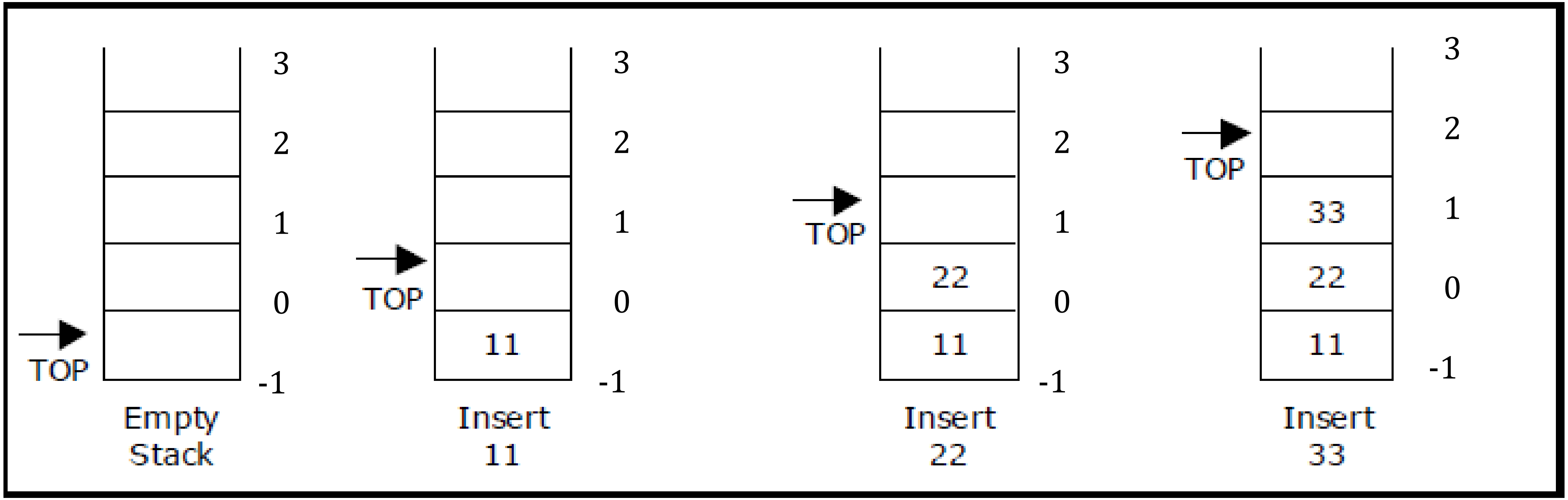


```
// Function to check if the stack is full
bool isFull() {
    return top == MAX_SIZE - 1;
}
```

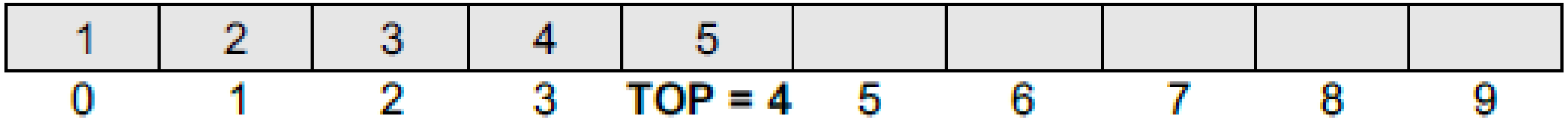
```
Boolean isfull()
{
    If (top=MAXSIZE-1)
        return true
    else
        return false
}
```

Example : Push() operation on stack

Push(11)
Push(22)
Push(33)

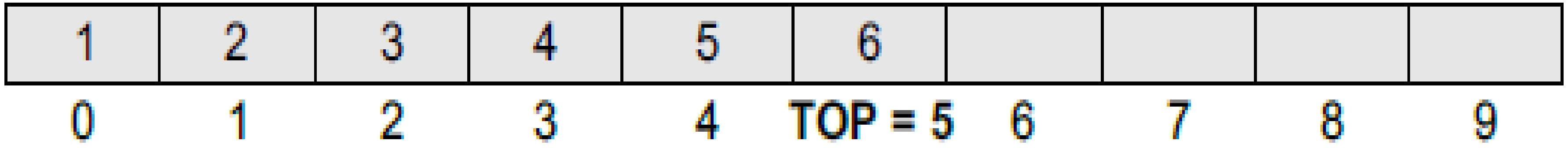


Insert an element in a stack : Example 1



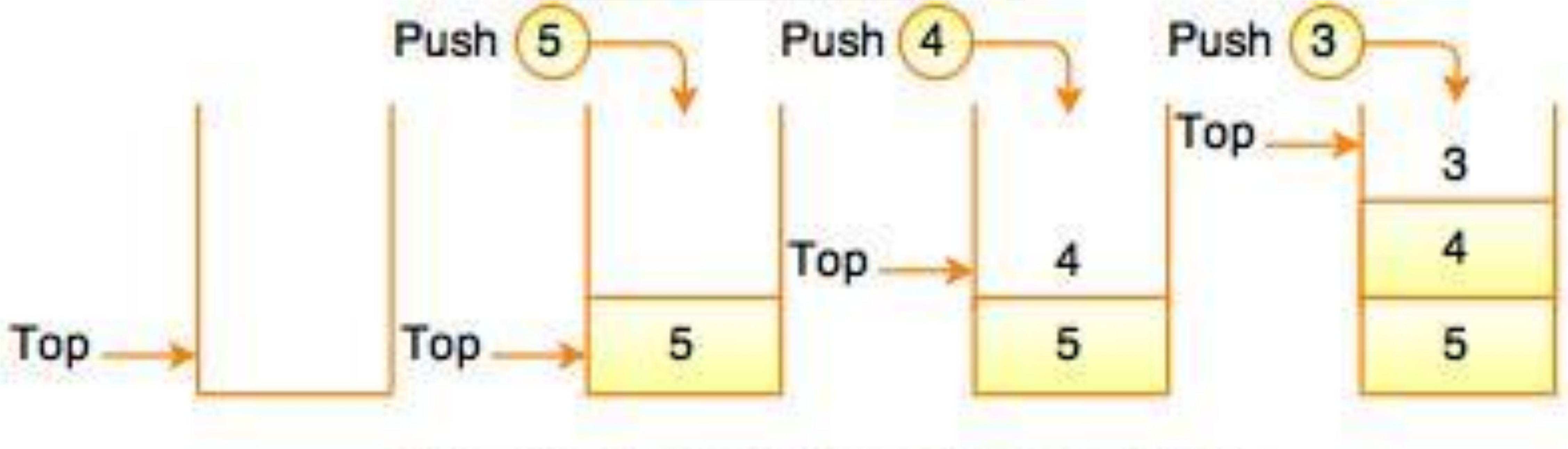
1	2	3	4	5					
0	1	2	3	TOP = 4	5	6	7	8	9

- To insert an element with value 6,
- we first check if TOP=MAX–1.
- If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].



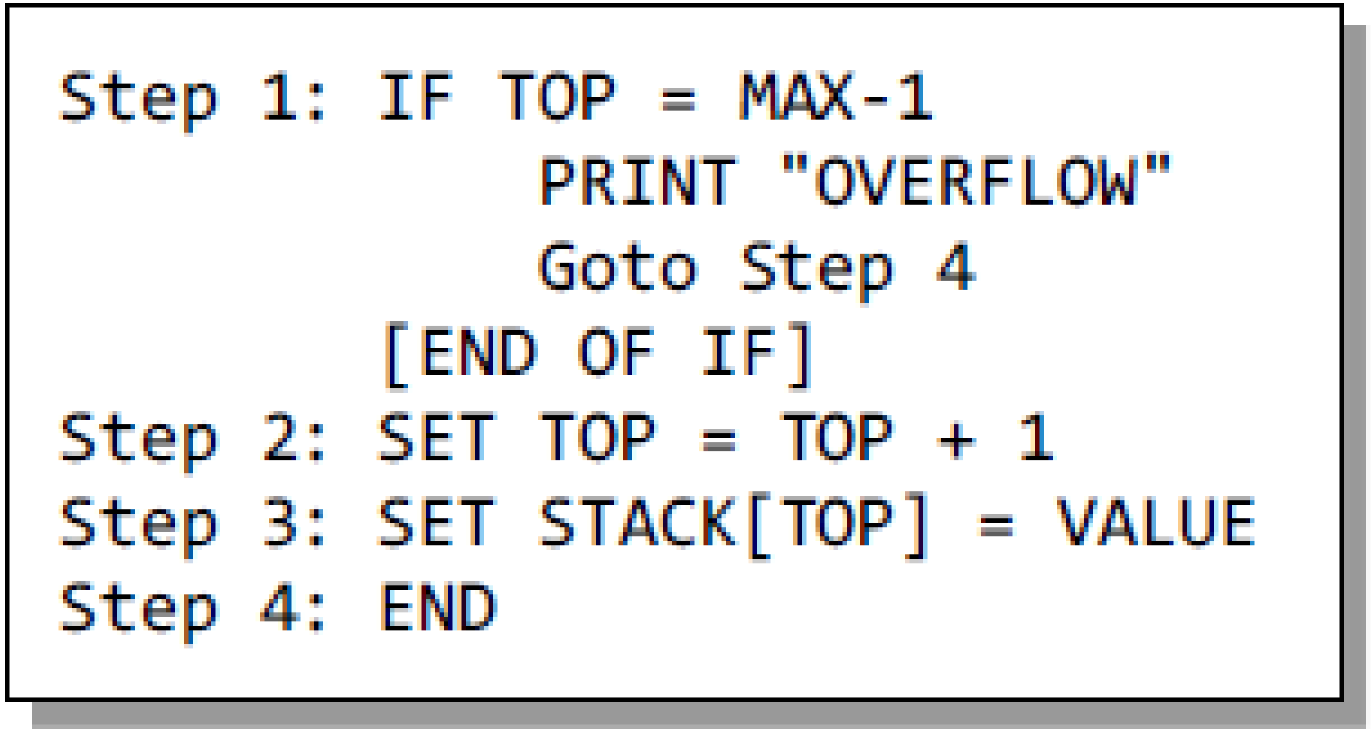
1	2	3	4	5	6				
0	1	2	3	4	TOP = 5	6	7	8	9

Insert an element in a stack : Example 2



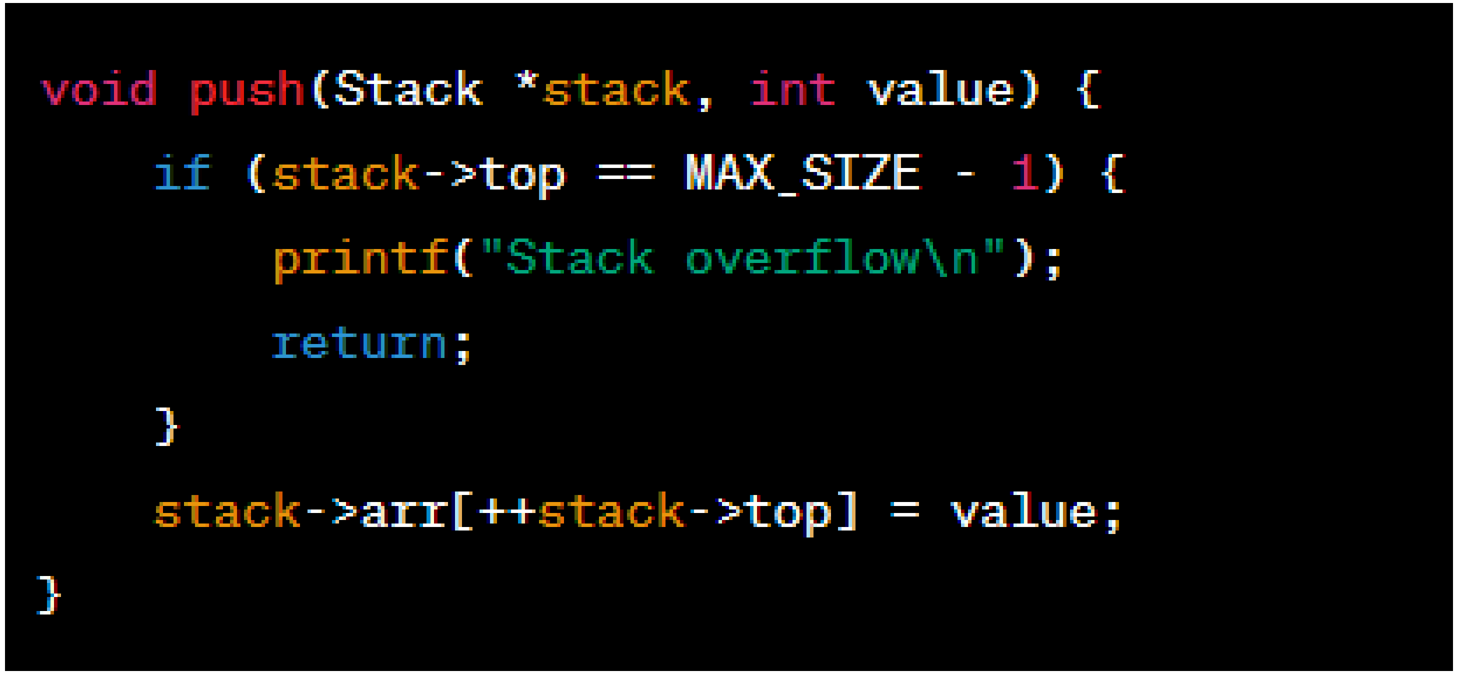
- To insert an element with value N,
- We first check if TOP=MAX–1. (MAX =3, TOP = 1)
- If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].

Pseudocode to insert an element in a stack



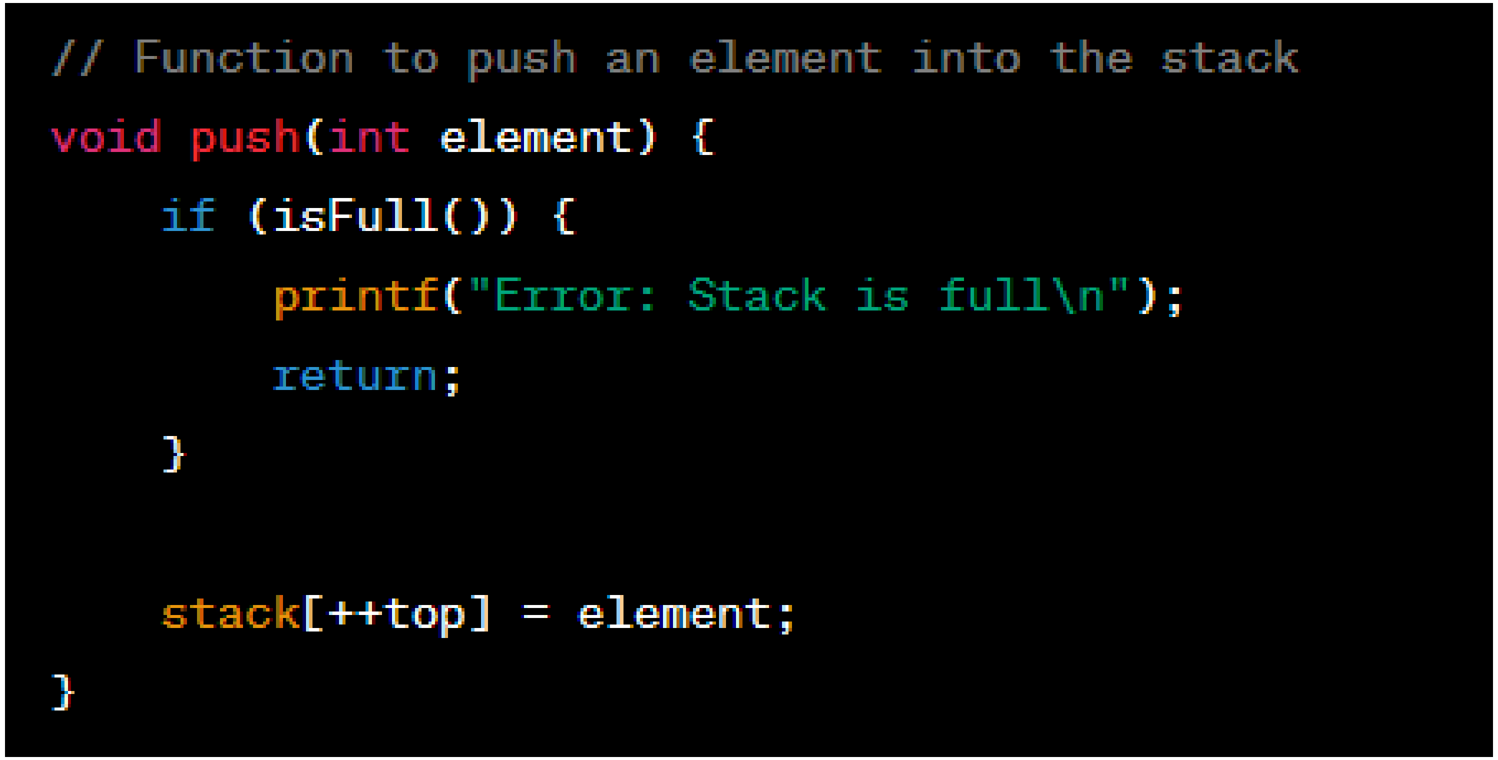
```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Push elements to Stack structure



```
void push(Stack *stack, int value) {
    if (stack->top == MAX_SIZE - 1) {
        printf("Stack overflow\n");
        return;
    }
    stack->arr[++stack->top] = value;
}
```

Push elements to Stack array

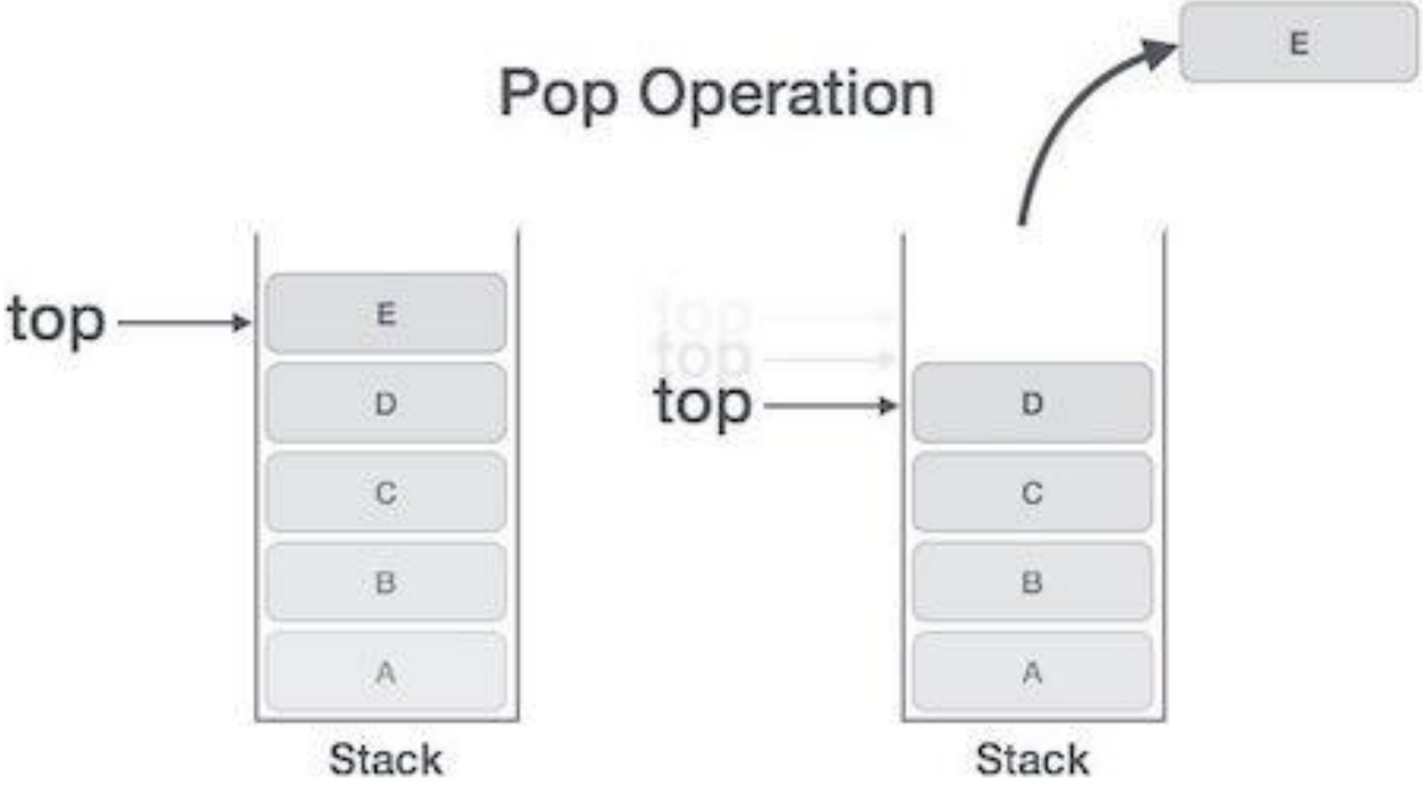


```
// Function to push an element into the stack
void push(int element) {
    if (isFull()) {
        printf("Error: Stack is full\n");
        return;
    }

    stack[++top] = element;
}
```

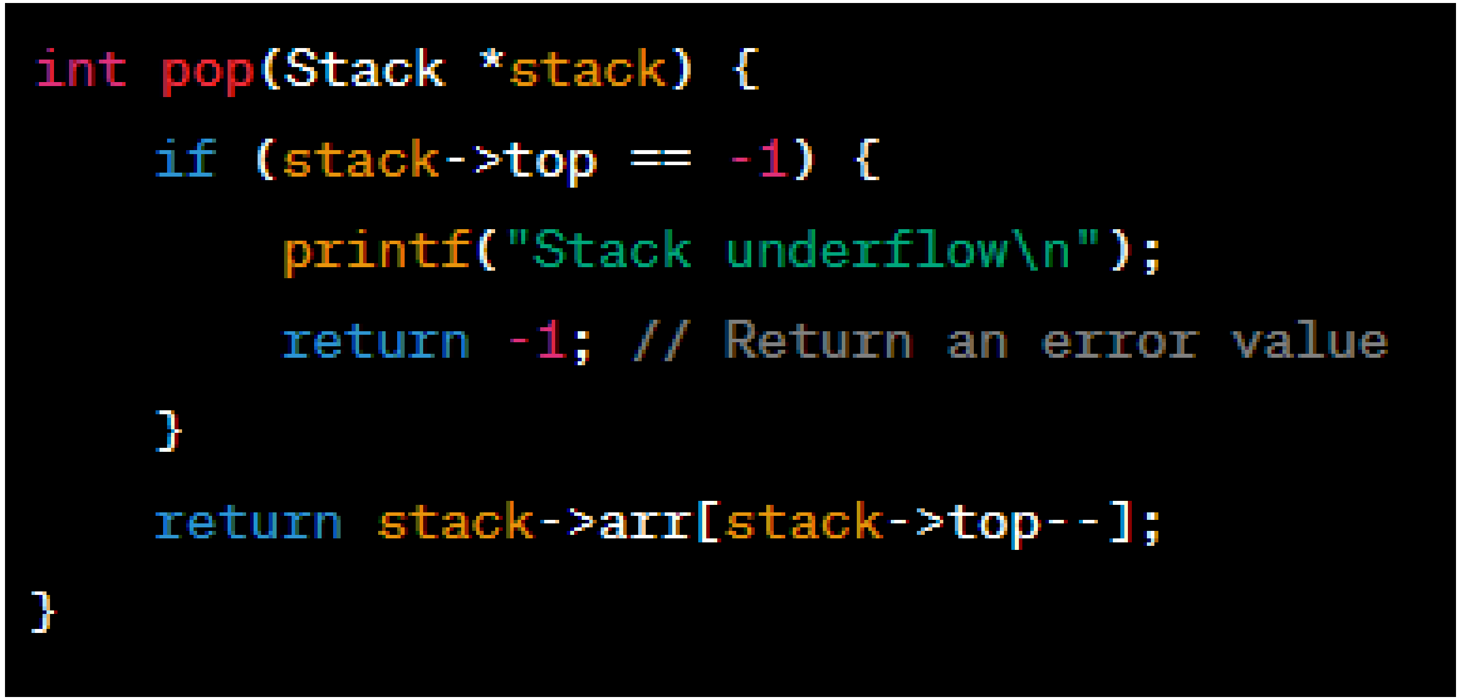
Pop Operation

- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=-1/ NULL because if that is the case, then it means the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

Pop Operation

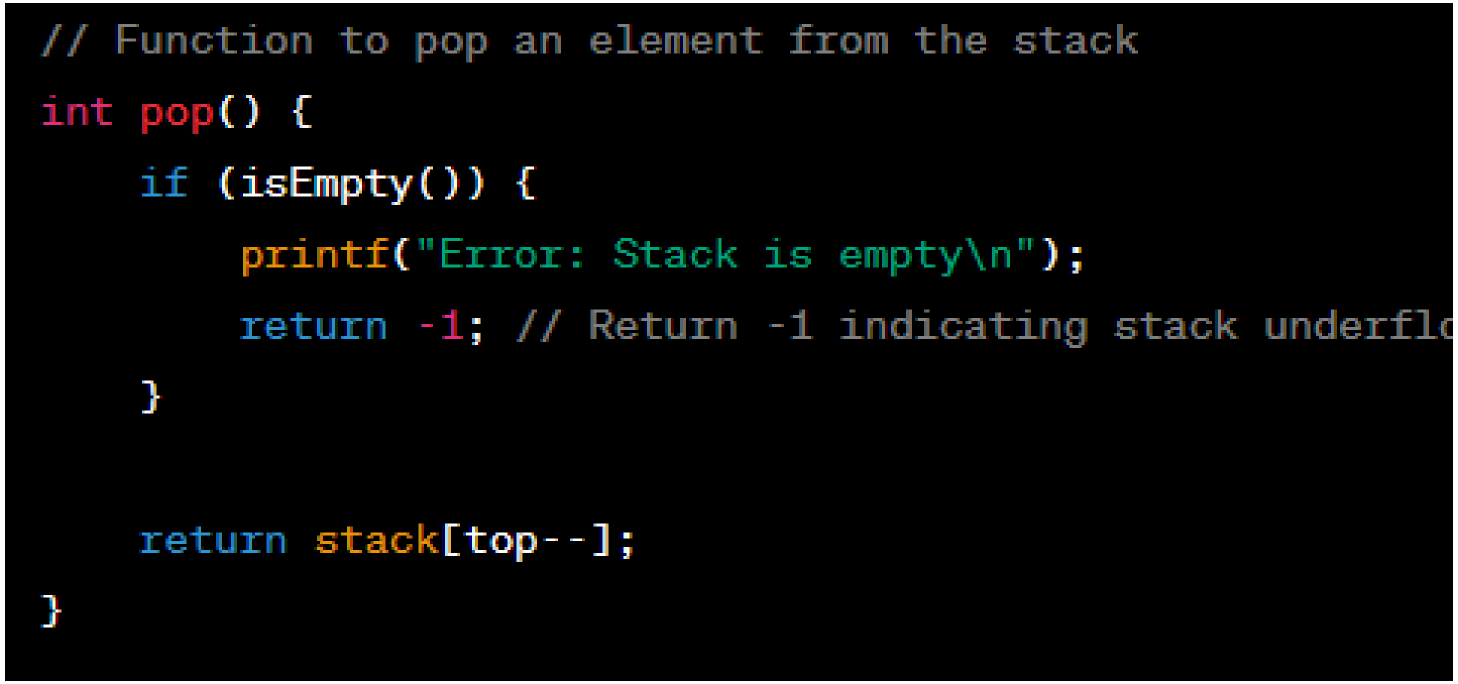


Pop elements from Stack structure



```
int pop(Stack *stack) {
    if (stack->top == -1) {
        printf("Stack underflow\n");
        return -1; // Return an error value
    }
    return stack->arr[stack->top--];
}
```

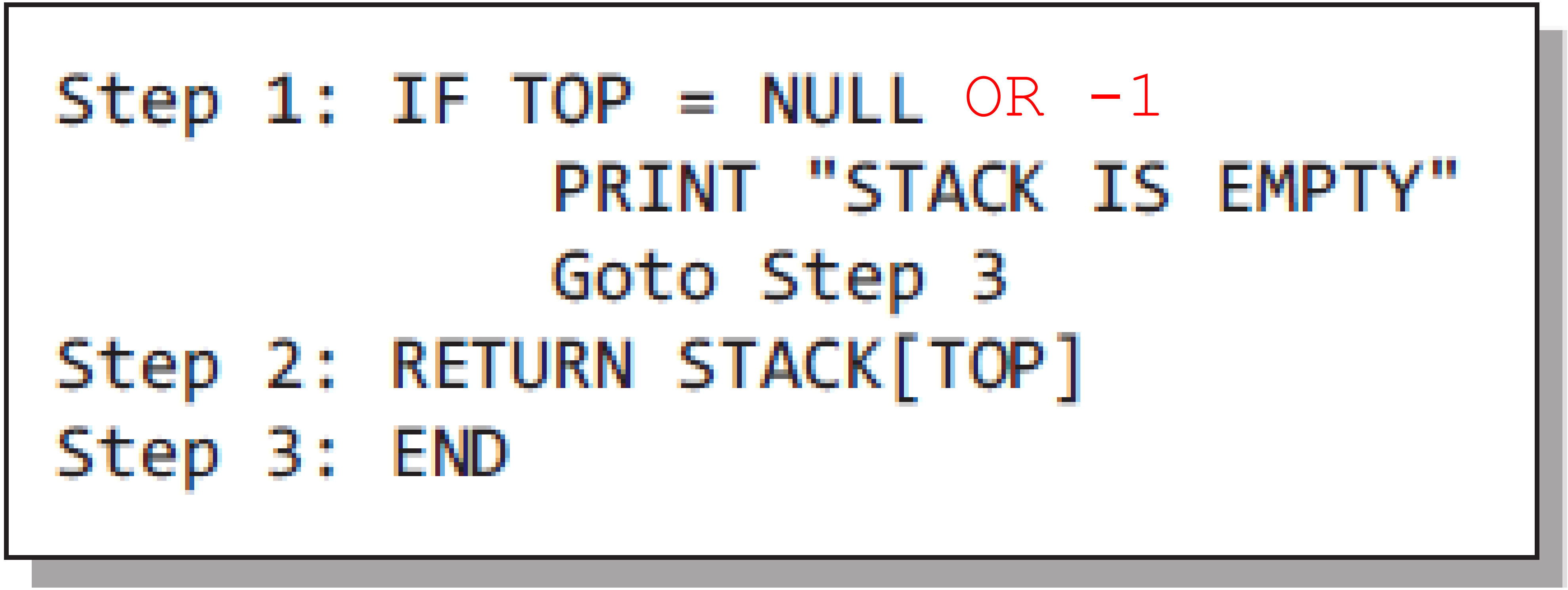
Pop elements from Stack array



```
// Function to pop an element from the stack
int pop() {
    if (isEmpty()) {
        printf("Error: Stack is empty\n");
        return -1; // Return -1 indicating stack underflo
    }

    return stack[top--];
}
```

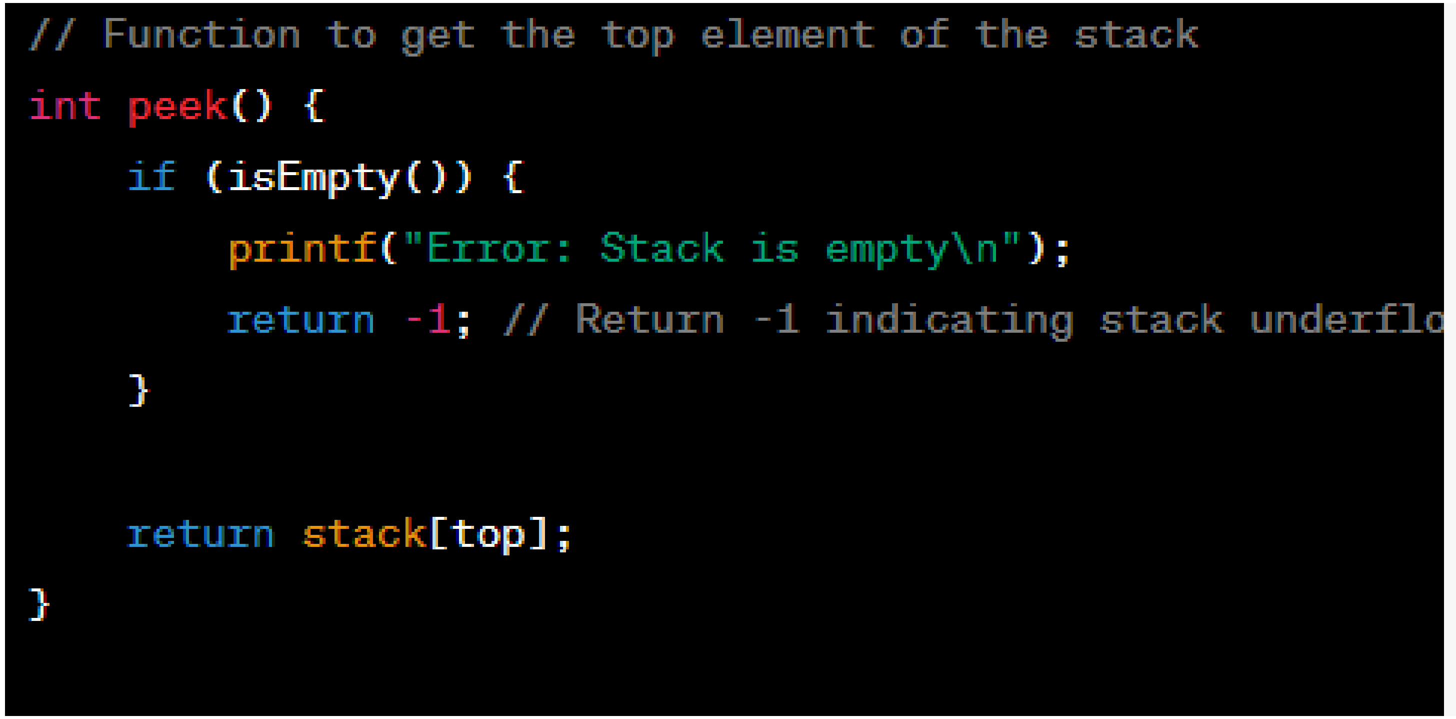
Peek Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack



```
Step 1: IF TOP = NULL OR -1
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Peek into Stack array



```
// Function to get the top element of the stack
int peek() {
    if (isEmpty()) {
        printf("Error: Stack is empty\n");
        return -1; // Return -1 indicating stack underflo
    }

    return stack[top];
}
```

- A stack can be implemented with an array and an integer. The integer tos (Top of stack) provides the array index of the top element of the stack. Thus if tos is –1, the stack is empty.

Stack implementation

Principal operations

Push :adds an item to a stack

Pop :extracts the most recently pushed item from the stack

Other methods such as

top: returns the item at the top *without removing it*

isempty :determines whether the stack has anything in it

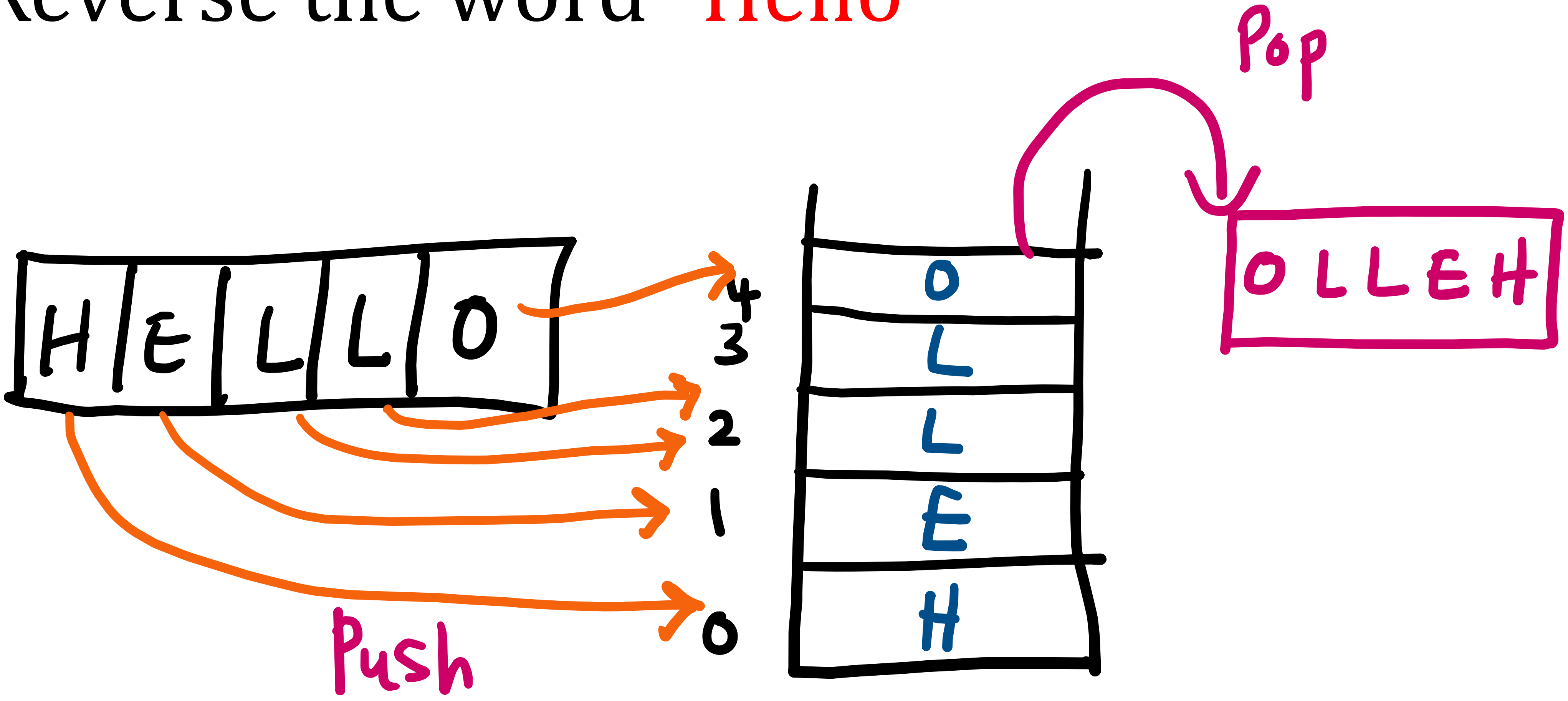
Applications of Stack

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

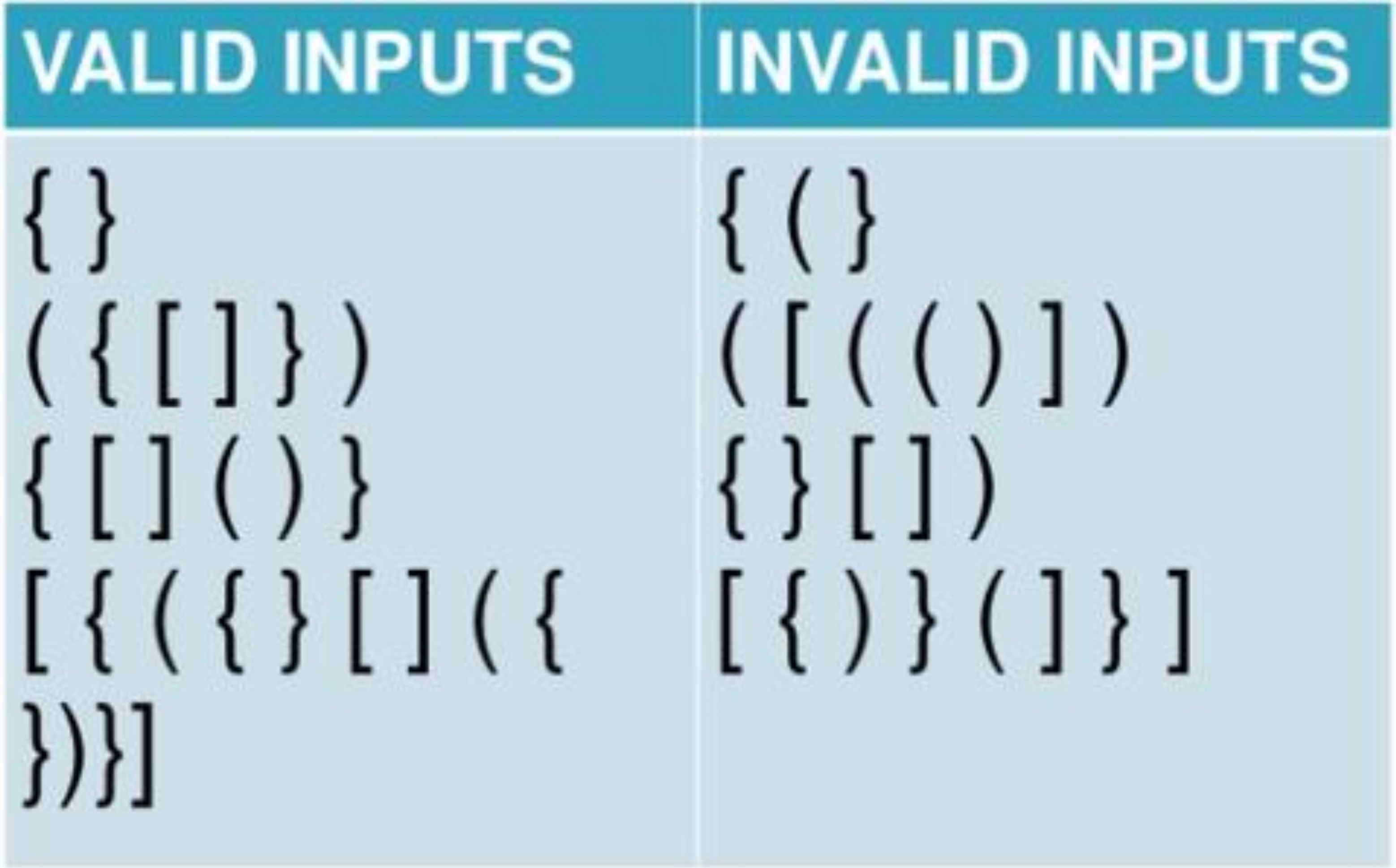
Applications : Reversing a List

- A list of characters can be reversed by reading each character from an array starting from the first index and pushing it on a stack.
- Once all the characters have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

Reverse the word “Hello”



Applications : Balanced parentheses

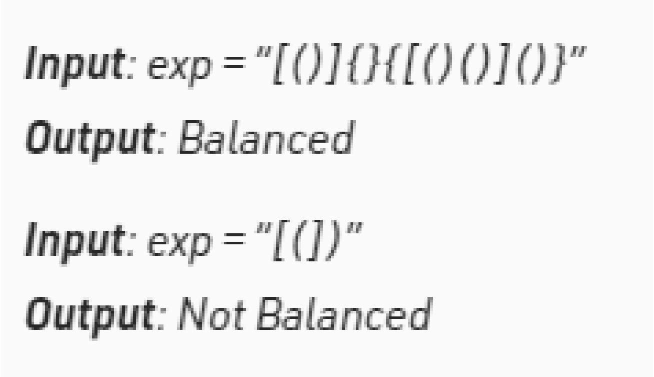


VALID INPUTS	INVALID INPUTS
{ }	{ (}
({ [ ] })	([(( )])
{ [ ] ( ) }	{ } [ ])
[{ ({ } [ ] ({ }))}]	[{) } ( ] }]

Checking balanced parenthesis

- Used to check whether a given arithmetic expressions containing nested parenthesis is properly parenthesized.
- Need to check in an expression, each left parenthesis, braces or bracket are with proper closing symbol, and they are properly nested.

Q: Given a string expression, write a program to examine whether the pairs and orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in expression .



Input: *exp = “[()]{}{[()()]()}”*

Output: *Balanced*

Input: *exp = “[(])”*

Output: *Not Balanced*

Balanced parenthesis

- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

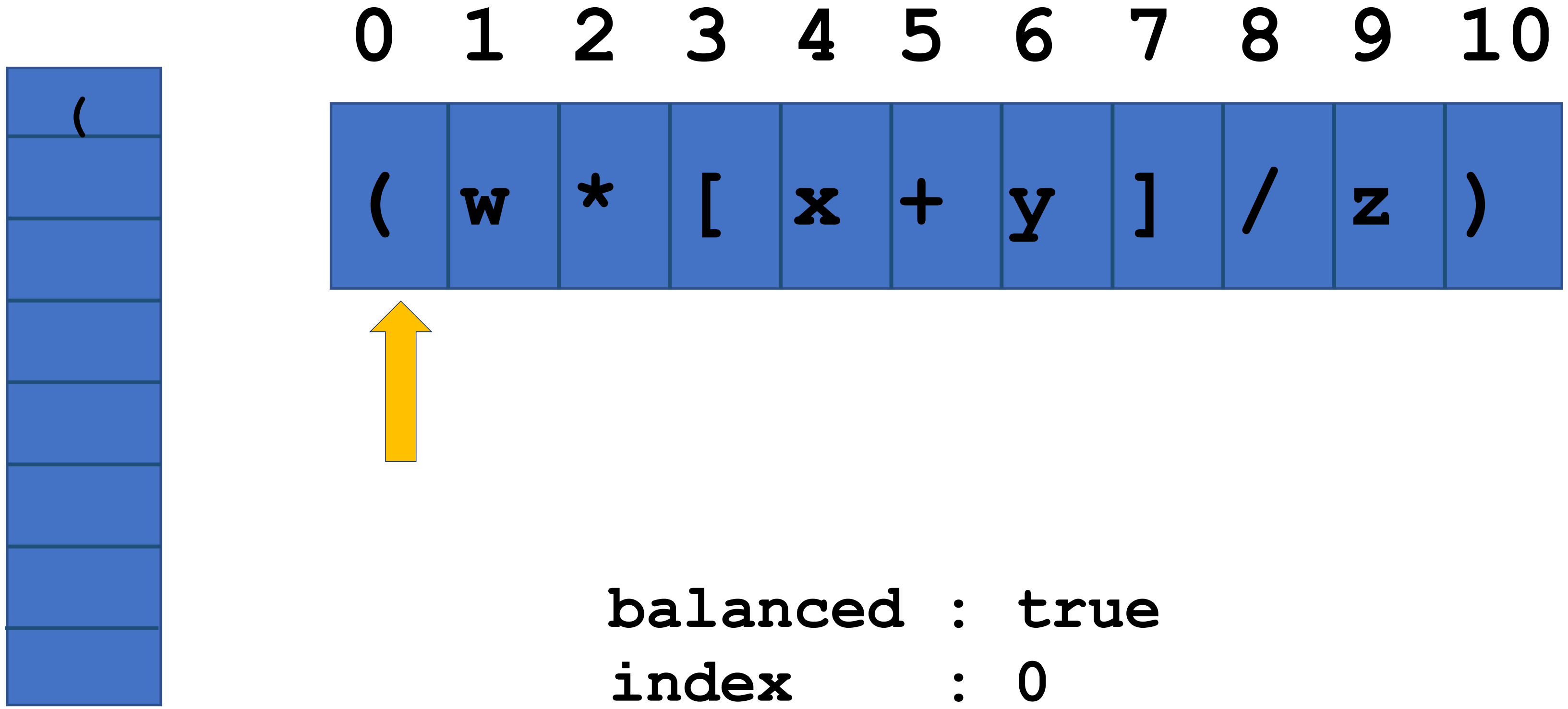
```
(a+b*(c/(d-e)))+(d/e)
```

- The problem is further complicated if braces or brackets are used in conjunction with parentheses
- The solution is to use stacks!

Balanced parenthesis

Expression **(w * [x + y] / z)**

:



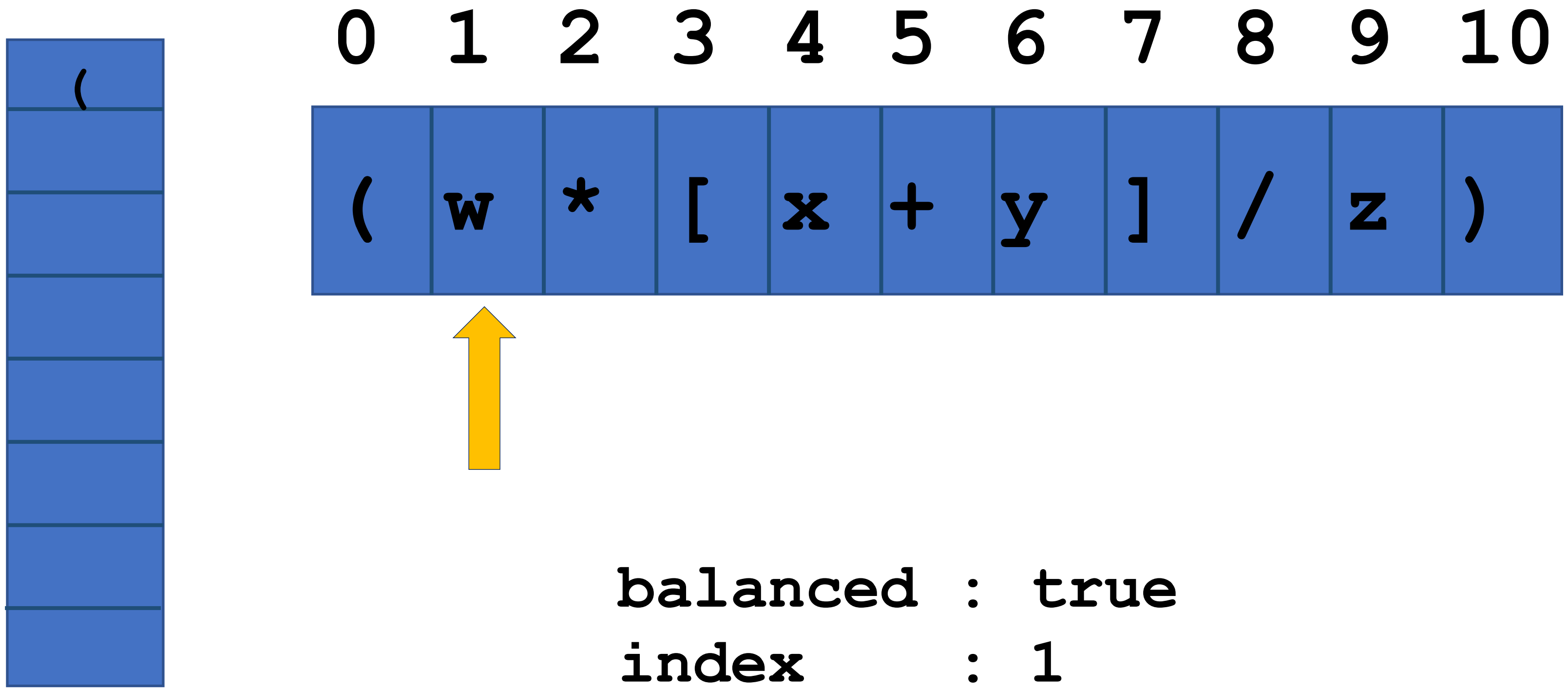
(

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 0
```

Balanced parenthesis

Expression **(w * [x + y] / z)**
:

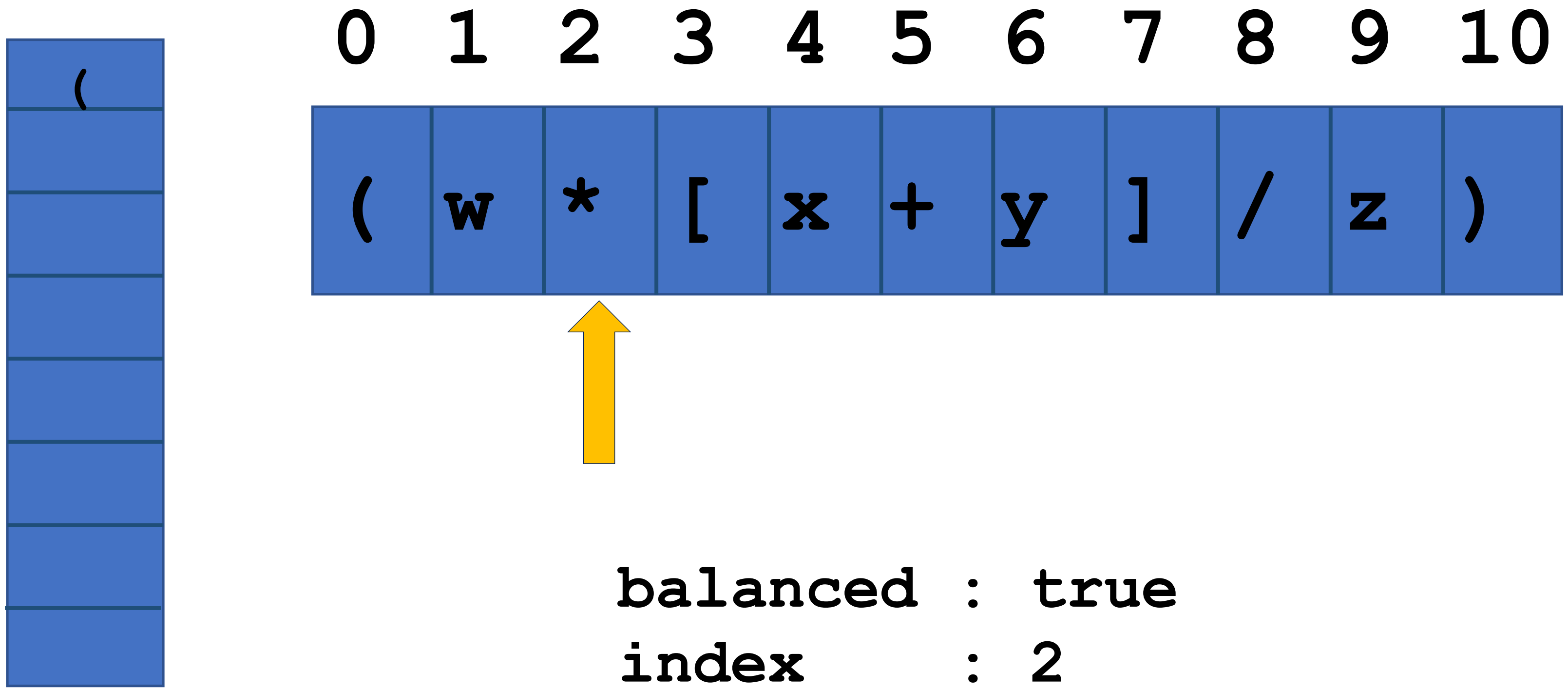


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 1
```

Balanced parenthesis

Expression: `(w * [x + y] / z)`

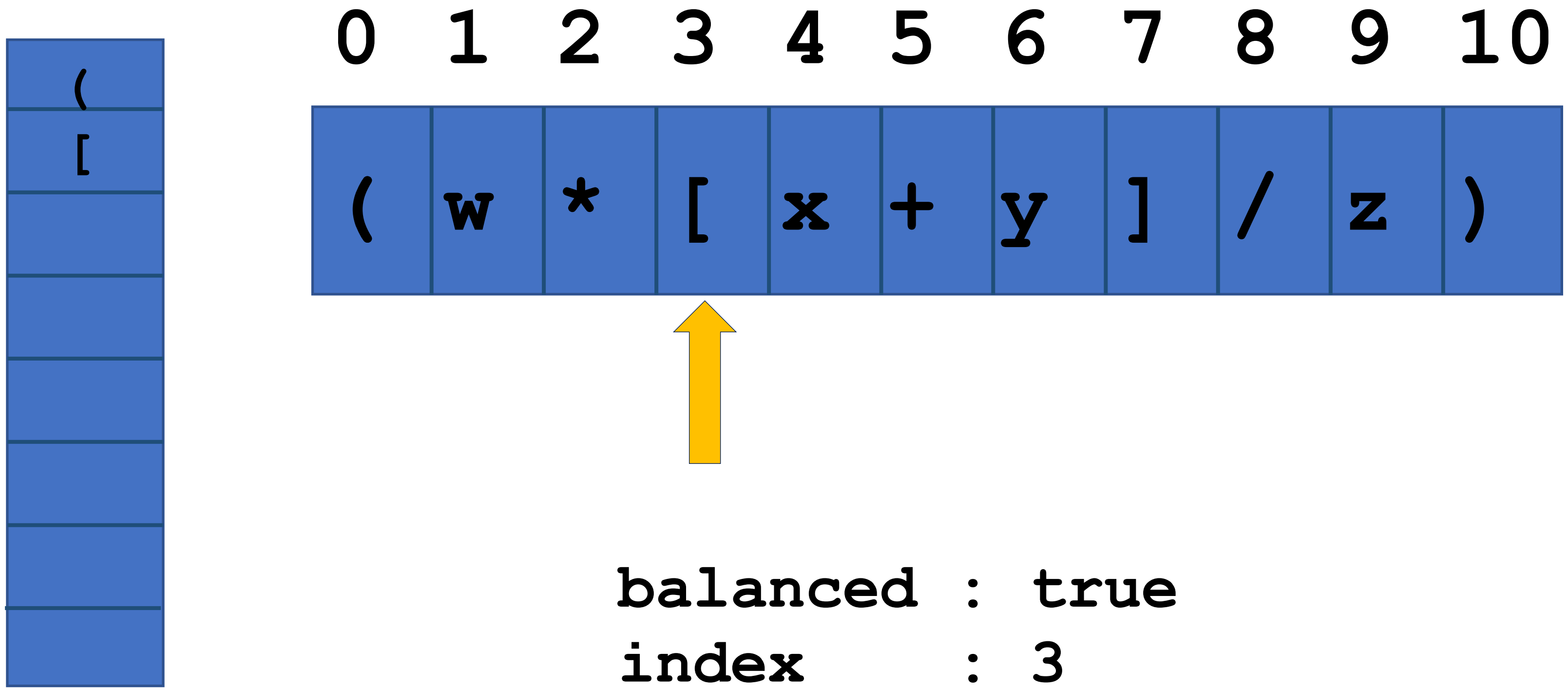


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 2
```

Balanced parenthesis

Expression (w * [x + y] / z)
:

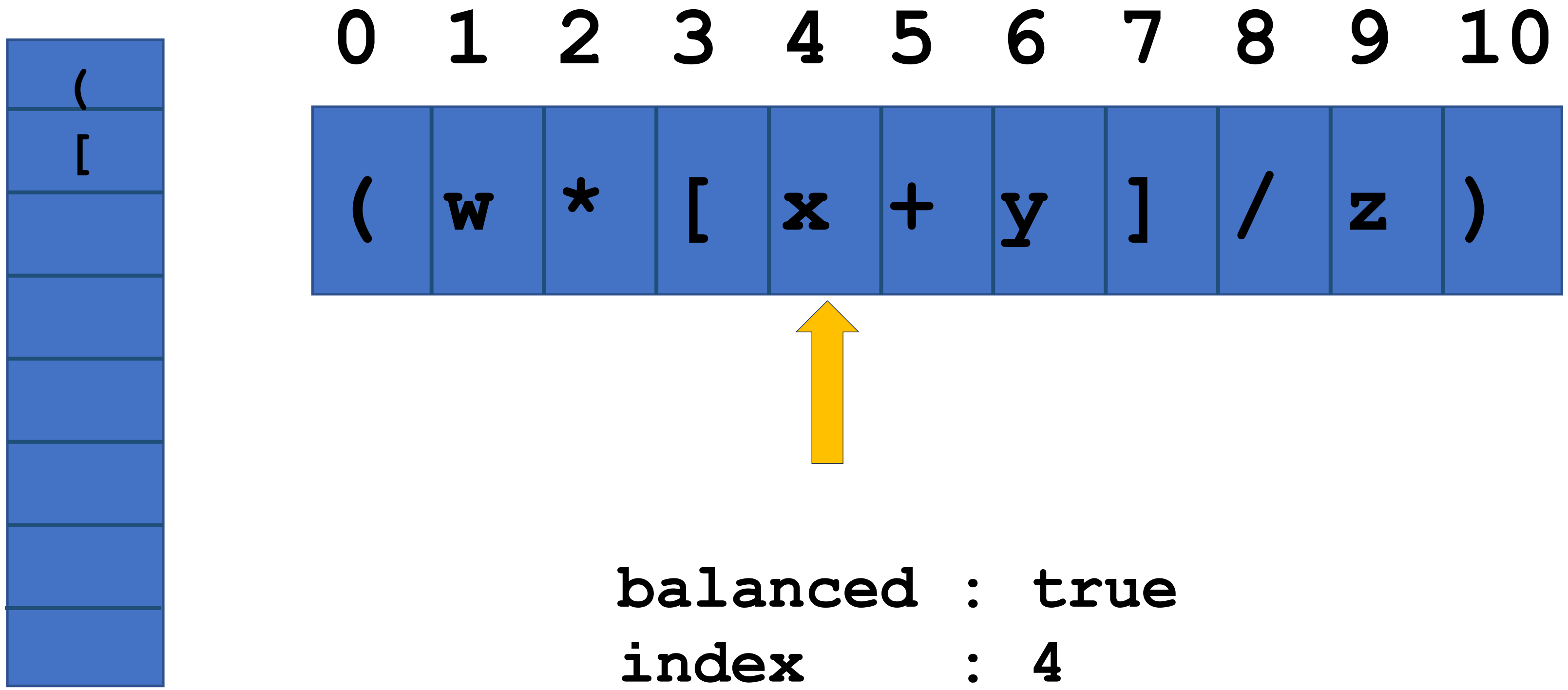


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 3
```

Balanced parenthesis

Expression `(w * [x + y] / z)`
:



(
[

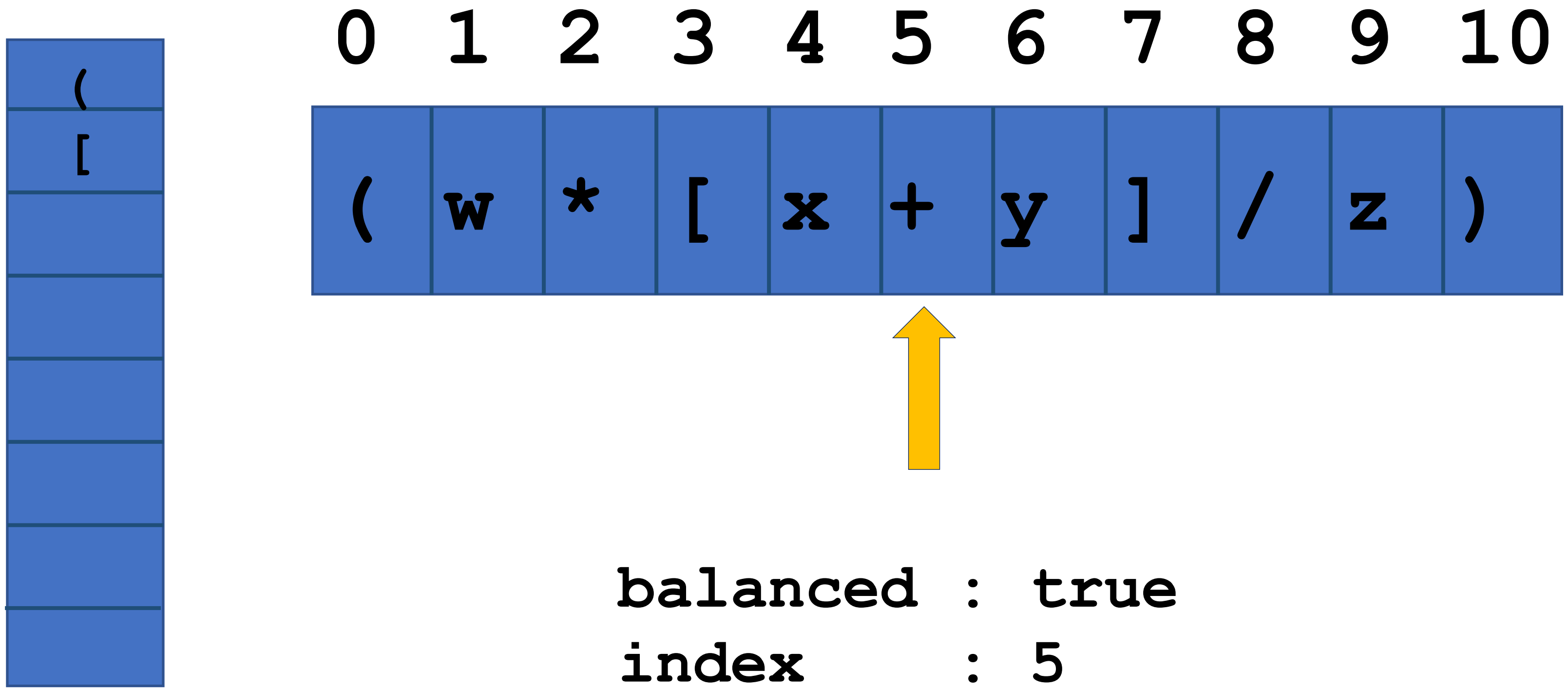
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 4
```

Balanced parenthesis

Expression **(w * [x + y] / z)**

:



(

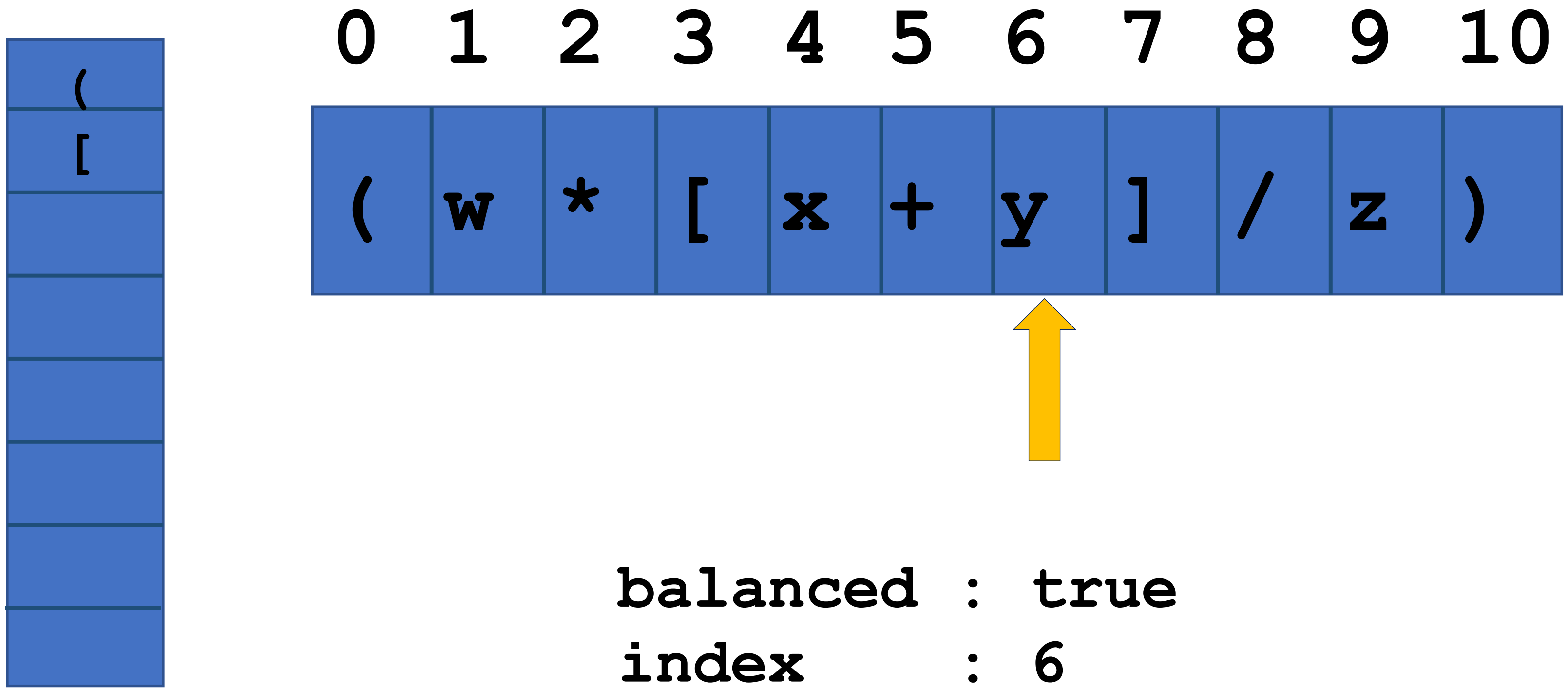
[

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 5
```

Balanced parenthesis

Expression : **`(w * [x + y] / z)`**

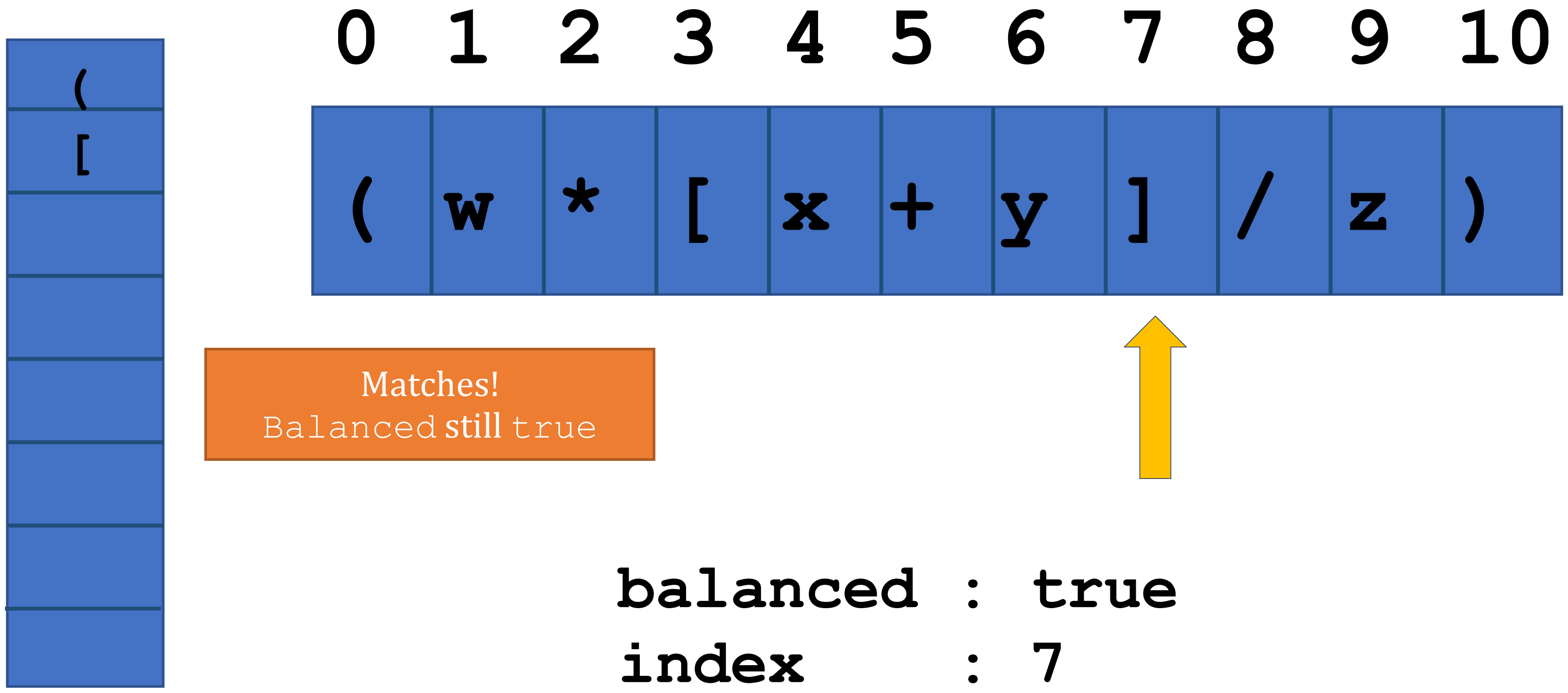


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 6
```

Balanced parenthesis

Expression **`(w * [x + y] / z)`**
:



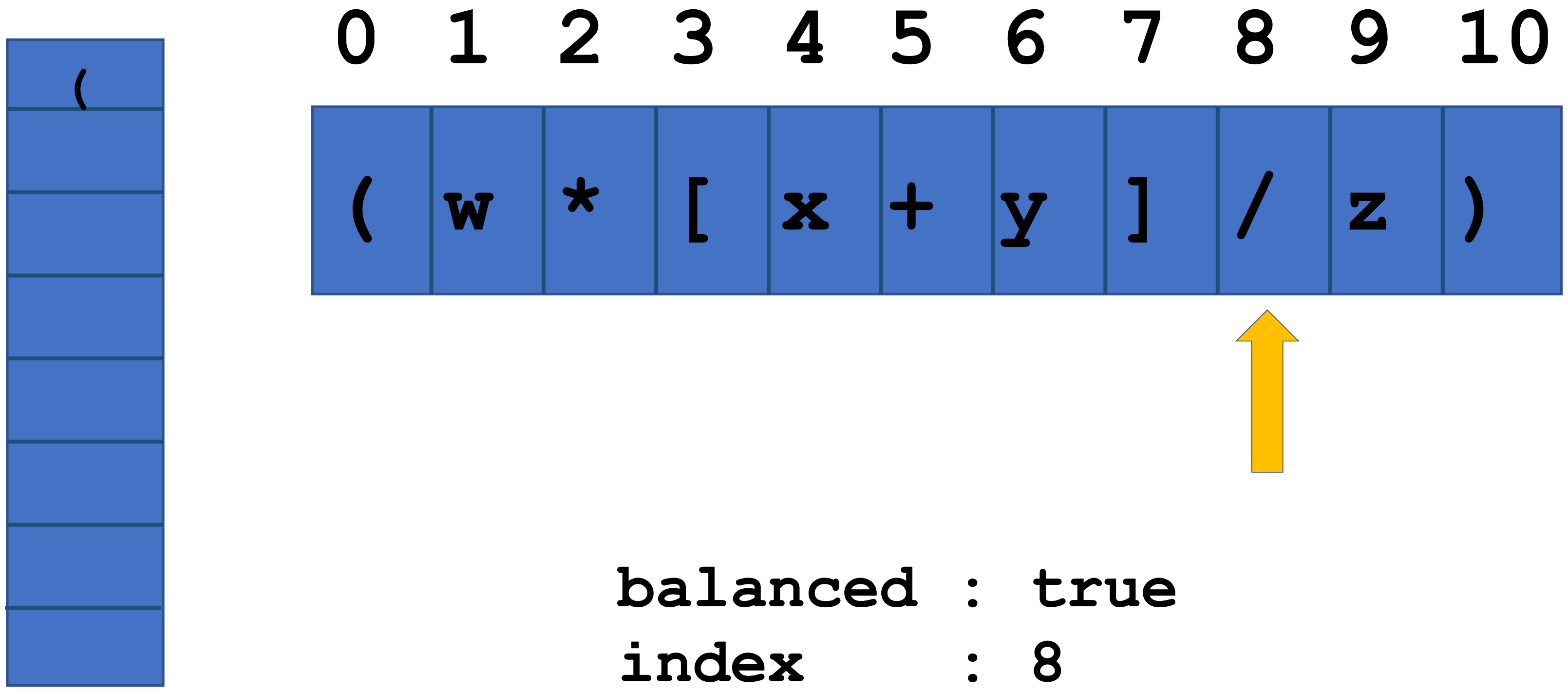
0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

Matches!
Balanced still true

```
balanced : true
index    : 7
```

Balanced parenthesis

Expression : **`(w * [x + y] / z)`**

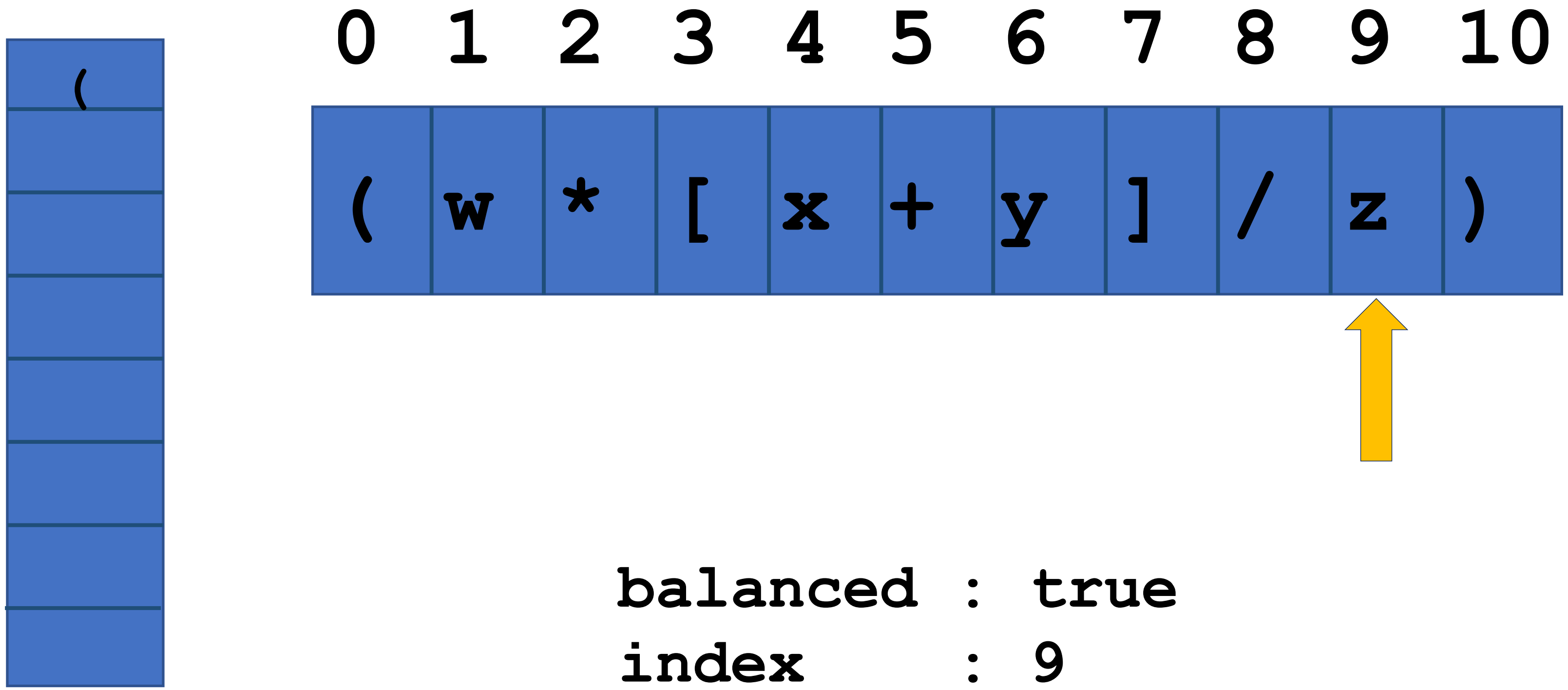


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 8
```

Balanced parenthesis

Expression : **(w * [x + y] / z)**

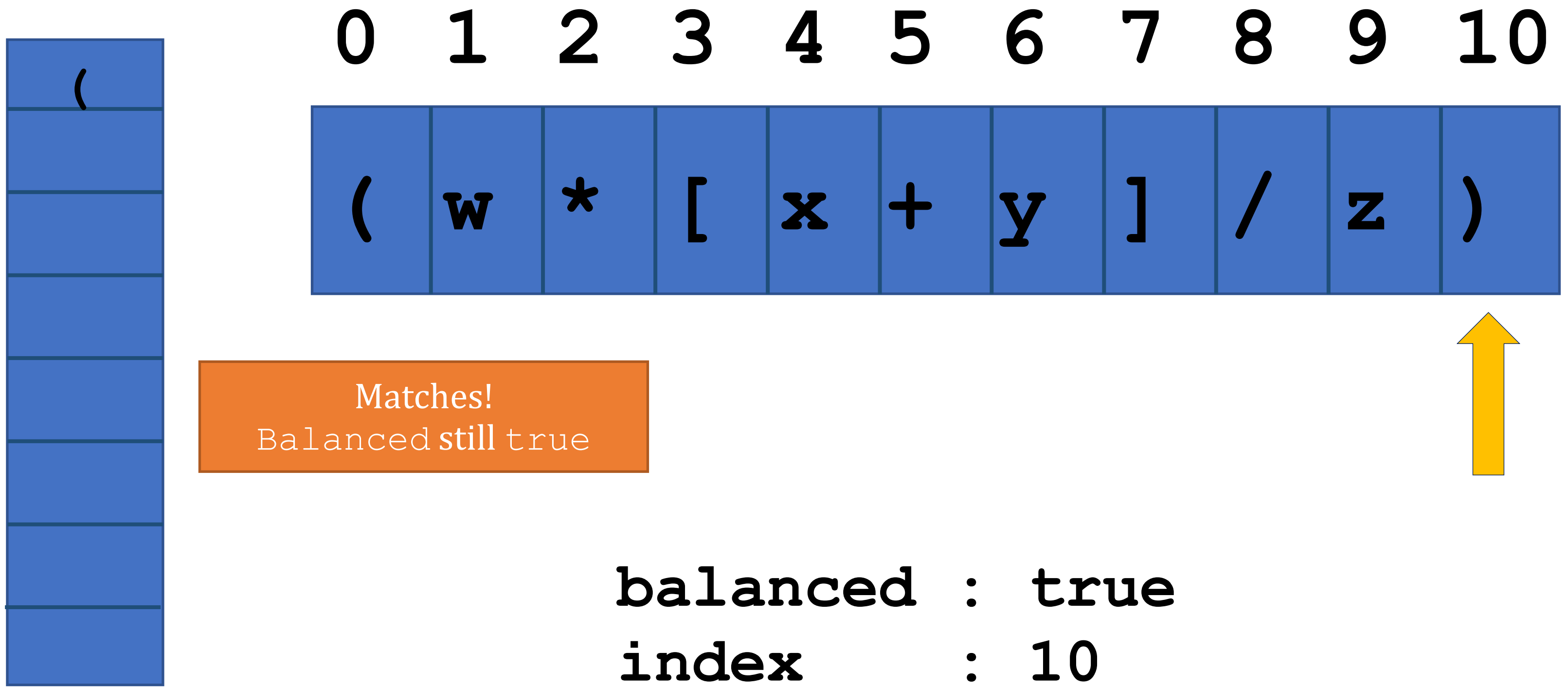


0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

```
balanced : true
index    : 9
```

Balanced parenthesis

Expression **`(w * [x + y] / z)`**
:



0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

Matches!
`Balanced` still `true`

```
balanced : true
index    : 10
```

Balanced parenthesis (Cont’d)

How to use a stack to check for balanced symbols?

Compilers check your program for syntax errors. However, frequently a lack of one symbol will cause the compiler to spill out a hundred lines of diagnostic without identifying the real error.

Parenthesis sequence [()] is legal, but [(]) is wrong.

1. Make an empty stack
2. Read symbols until the end of the file

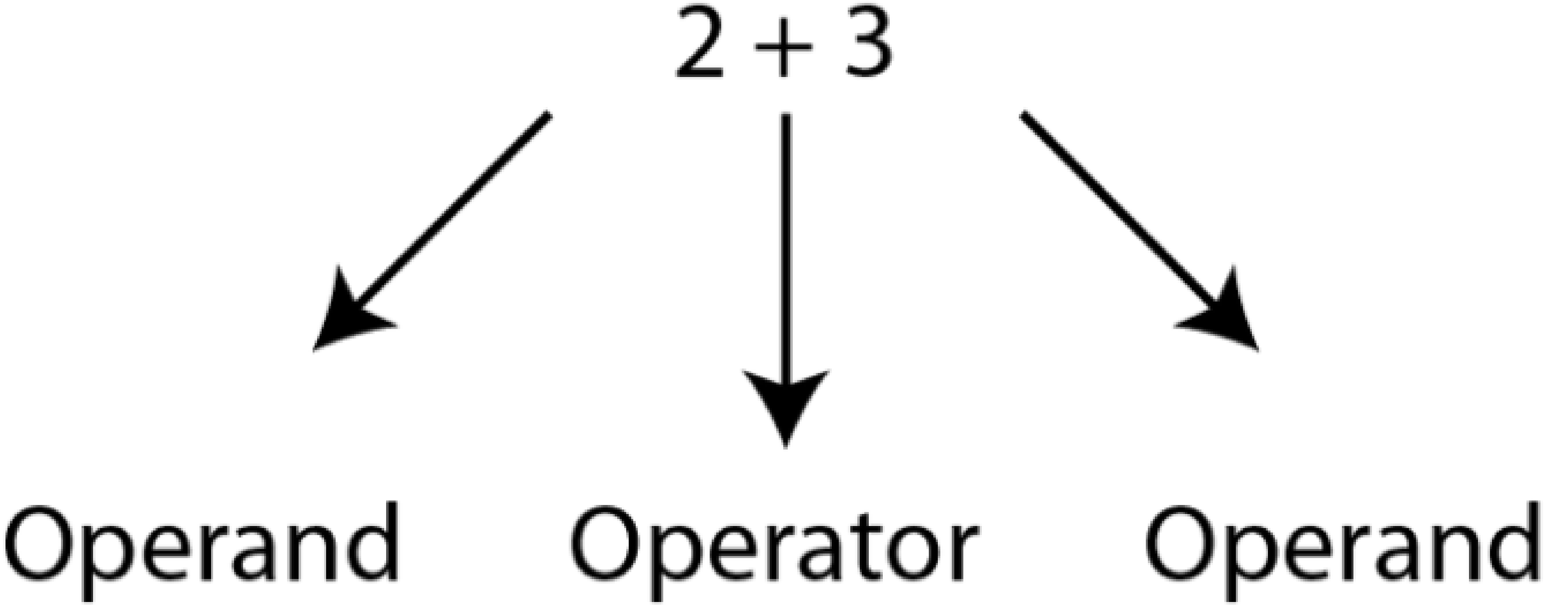
   a) If symbol is opening symbol, push it into the stack

   b) If it is closing symbol and if the stack is empty, then report an error.

   c) Otherwise, pop the stack, if the symbol popped is not in the corresponding opening symbol , then report an error
3. At the end of the file, if the stack is not empty, report an error.

Note : we should not consider a parenthesis as a symbol if it occurs inside a comment, string, constant or character string.

Evaluation of Arithmetic Expressions



Evaluation of Arithmetic Expressions

- Infix : A+B
- Prefix : +AB
- Postfix : AB+

Suppose that we would now like to rewrite A+B*C in postfix, Appling the rules precedence, we obtain,

A+(B*C) – Parentheses for emphasis

A+(BC*) – Convert the multiplication

A(BC*)+ – convert the addition

ABC*+ – postfix form

Eq: Infix – A+(B*C) = Postfix ABC*+

Infix – (A+B)*C = Postfix AB+C*

Question : How to evaluate a mathematical expression using a stack

Evaluating a postfix expression

- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Homework

1. Write a simple program to reverse a user given string.

2. Given a string expression, write a program to examine whether the pairs and orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in expression.

3. Write a simple program to evaluate postfix user given expression.

Thank you