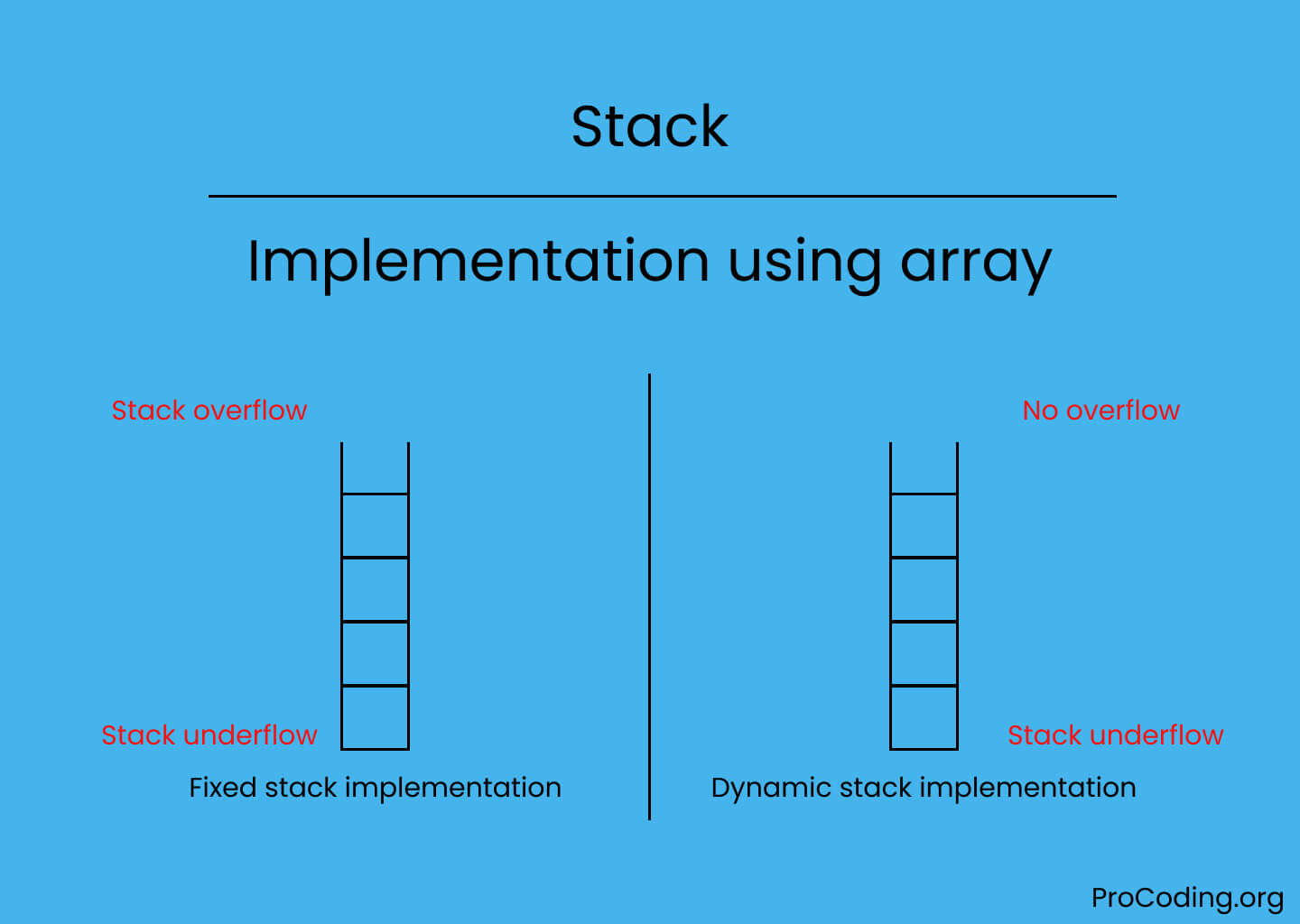
**Stack using Array**

Basically the size of the array in case fixed array implementation is always fixed but in case of dynamic array implementation size of the array may grow if stack overflow occurs.



Stack is a linear data structure which works in LIFO (Last In First Out) or FILO (First In Last Out) order. According to LIFO, the element inserted last will be removed first, and similarly, the element inserted first will be removed last.



LIFO

## **Introduction**

An array of one dimension can be used to build a stack data structure. However, a stack that is constructed using an array only stores a set number of data values. It's really easy to put this into practice. Simply build a one-dimensional array of a certain size, then add or remove values from that array using the LIFO principle with the use of a variable called **"top"**. The top is initially set at -1. Every time we wish to add a value to the stack, we first add one to the top value before inserting it. When a value has to be removed from the stack, the top value is removed, and the top value is reduced by one.



Working of Stack Data Structure

Four main operations on stack:

* **Push:** Adds new item on the top of stack.
* **Pop:** Removes an item from the top of stack.
* **Peek:** Returns top element of stack.
* **is\_empty:** Check if stack is empty or not.
* **Is\_full()**

### **push(value)**

This operation is used to insert values into the stack. It inserts elements on top of the stack.

**Algorithm**

* Check if the top is equal to the size of the array. If true, print "Stack is Full" and return.
* If false, increment the top by one.
* Assign an item to the top of the stack.

**def push(self, element):**

**# Check whether a stack is full or not**

**if self.isFull():**

**print("Stack is Full")**

**return**

**# Push the element into the stack**

**self.stack.append(element)**

### pop()

This operation is used to remove an element from the stack. It uses the LIFO property to remove elements from the stack, i.e., the element inserted last will be removed first from the stack. It returns the removed element from the stack.

**Algorithm**

* Check if the top is equal to -1. If true, then print "Stack is Empty" and return.
* If false, then store the top item in a variable.
* Decrement the top.
* Return the stored item through the variable.

def pop(self):

*# Check whether a stack is empty or not*

if not self.isEmpty():

item = self.stack[-1]

*# Pop the element from a stack*

del self.stack[-1]

return item

else:

return "Stack Already Empty"

### isEmpty()

This operation is used to check whether the stack is empty or not. It returns a boolean variable, True or False, indicating whether the stack is empty or has some elements in it.

**Algorithm:**

* Check if the top is equal to -1. If True, returns True.
* Else return false.

def isEmpty(self):

return self.stack == []

### isFull()

This operation is used to check whether the stack is full or not. It returns a boolean variable, True or False, indicating whether the stack is full or some more elements can still be inserted.

**Algorithm**

* Check if the top is equal to the size of the array, i.e., n, then return True.
* Otherwise, return False.

**def isFull(self):**

**return len(self.stack) == self.size**

### Peek()

This operation is used to see the top element of the stack. The Peek function will return the top element without removing it. It will work only when the stack is not empty.

**Algorithm:**

* Check if the stack top is equal to -1. If true, then print "Stack is Empty" and return.
* If false, store the top of the stack item in a variable.
* Return that variable.

**def peek(self):**

**if self.isEmpty():**

**return "Stack Empty"**

**return self.stack[-1]**

**CODING**

class Stack:

def \_\_init\_\_(self):

self.items = []

def is\_empty(self):

return len(self.items) == 0

def push(self, item):

self.items.append(item)

def pop(self):

if self.is\_empty():

return None

return self.items.pop()

def peek(self):

if self.is\_empty():

return None

return self.items[-1]

def size(self):

return len(self.items)

# Example usage

s = Stack()

s.push(1)

s.push(2)

s.push(3)

print(s.size()) # Output: 3

print(s.pop()) # Output: 3

print(s.pop()) # Output: 2

print(s.peek()) # Output: 1

## **Pros and Cons of a Stack Implementation using an Array**

### Pros

* The pointers in an array-based stack implementation can be stored without using any additional memory.
* It is more time-efficient than the stack implementation utilizing linked lists.
* The implementation of the stack using an array is more secure and reliable.
* Array implementation of Stack is highly recommended in allocation and deallocation.

### Cons

* Because the stack's size is constant, an array cannot be used to increase or decrease the stack's size.
* Given that the items are stored in sequential memory locations, insertion and deletion in an array are fairly challenging.
* There is a high possibility of Stack Overflow.
* While using an Array, random access to stack positions is not possible.

## **Applications of Stack Data Structure**

Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

* **To reverse a word** - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
* **In compilers** - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 \* (7 - 9) by converting the expression to prefix or postfix form.
* **In browsers** - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

## **Summary**

the implementation of stack using an array by mentioning some of the important points.

* A Stack is a linear data structure that uses LIFO functionality.
* A stack can be implemented using an array and supports functions like push, pop, peek, empty and full.
* The push and pop functions are used to insert and delete elements in the stack, respectively.
* In the stack implementation of the array, we maintain the top with a variable and the stack has a predefined size that cannot be increased later.
* Stack implementation using an array is much more time-efficient as compared to other methods.

Implementing a stack using a linked list in Python:

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class Stack:

def \_\_init\_\_(self):

self.top = None

def is\_empty(self):

return self.top is None

def push(self, item):

new\_node = Node(item)

new\_node.next = self.top

self.top = new\_node

def pop(self):

if self.is\_empty():

return None

item = self.top.data

self.top = self.top.next

return item

def peek(self):

if self.is\_empty():

return None

return self.top.data

def size(self):

current = self.top

count = 0

while current is not None:

count += 1

current = current.next

return count

# Example usage

s = Stack()

s.push(1)

s.push(2)

s.push(3)

print(s.size()) # Output: 3

print(s.pop()) # Output: 3

print(s.pop()) # Output: 2

print(s.peek()) # Output: 1

Implementation of stack using array

There are two ways to implement stack using arrays i.e.,

1. [Fixed array implementation](https://www.procoding.org/stack-using-array/#fixed-array-implementation)
2. [Dynamic array implementation](https://www.procoding.org/stack-using-array/#dynamic-array-implementation)

[**Fixed array implementation**](https://www.procoding.org/stack-using-array/#fixed-array-implementation)

In case of fixed array implementation of stack, the size of array is always fixed that's why when we try to push an element on stack if already the stack is full then **Stack overflow** exception will be raised and if we try to pop an element if stack is already empty then **Stack underflow** exception will be raised.

class Stack:

def \_\_init\_\_(self, size=10):

self.\_stack = []

self.\_size = size

def is\_empty(self):

return len(self.\_stack) <= 0

def push(self, data):

if len(self.\_stack) >= self.\_size:

raise Exception('Stack overflow')

else:

self.\_stack.append(data)

def pop(self):

if len(self.\_stack) <= 0:

raise Exception('Stack underflow')

else:

return self.\_stack.pop()

def peek(self):

if len(self.\_stack) <= 0:

raise Exception('Stack underflow')

else:

return self.\_stack[-1]

def length(self):

return len(self.\_stack)

stack = Stack(5)

stack.push(5)

stack.push(2)

stack.push(3)

stack.push(9)

stack.push(6)

print(stack.peek())

print(stack.pop())

print(stack.length())

print(stack.peek())

**output:**

**6**

**6**

**4**

**9**

#### [Dynamic array implementation](https://www.procoding.org/stack-using-array/#dynamic-array-implementation)

In case of dynamic array implementation of stack, the size of array is growable i.e., whenever stack reaches at the condition of overflow then the size of array will be increased that's why **Stack overflow exception will never raise** but **Stack underflow** may raise in case of stack is empty.

class Stack:

def \_\_init\_\_(self, size=10):

self.\_stack = []

self.\_size = size

def is\_empty(self):

return len(self.\_size) <= 0

def push(self, data):

if len(self.\_stack) >= self.\_size:

self.\_resize()

self.\_stack.append(data)

def pop(self):

if len(self.\_stack) <= 0:

raise Exception('Stack underflow')

else:

return self.\_stack.pop()

def peek(self):

if len(self.\_stack) <= 0:

raise Exception('Stack underflow')

else:

return self.\_stack[-1]

def length(self):

return len(self.\_size)

def \_resize(self):

new\_stack = [\*self.\_stack]

self.\_size = 2 \* self.\_size

self.\_stack = new\_stack

stack = Stack(2)

stack.push(5)

stack.push(4)

stack.push(3)

stack.push(9)

print(stack.peek())

print(stack.pop())

print(stack.peek())

**Output**

9

9

3