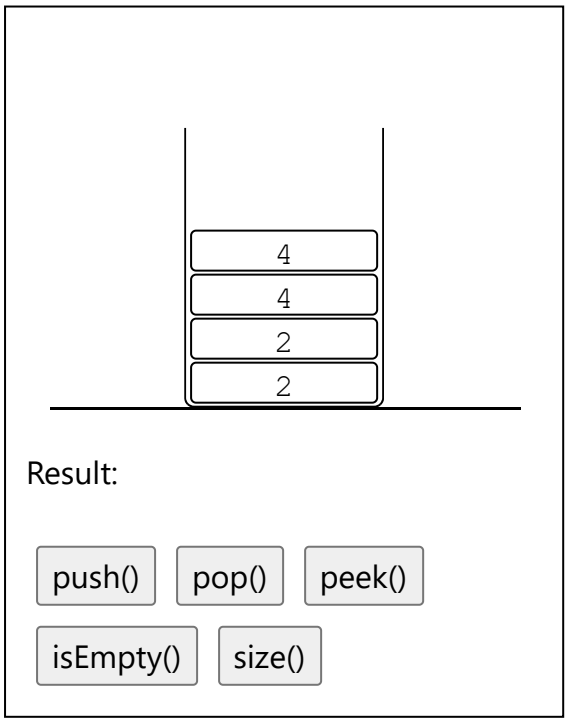
DSA Stacks

‹ Previous Next ›

Stacks

A stack is a data structure that can hold many elements.



Think of a stack like a pile of pancakes.

Basic operations we can do on a stack are:

- **Push:** Adds a new element on the stack.
- **Pop:** Removes and returns the top element from the stack.
- **Peek:** Returns the top element on the stack.
- **isEmpty:** Checks if the stack is empty.
- **Size:** Finds the number of elements in the stack.

Experiment with these basic operations in the stack animation above.

Stacks can be implemented by using arrays or linked lists.

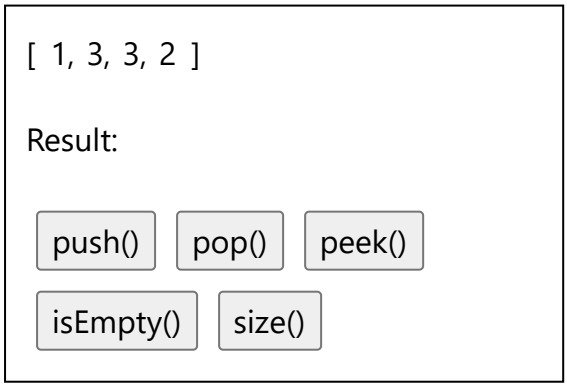
Stacks can be used to implement undo mechanisms, to revert to previous states, to create algorithms for depth-first search in graphs, or for backtracking.

Stacks are often mentioned together with Queues, which is a similar data structure described on the next page.

Stack Implementation using Arrays

To better understand the benefits with using arrays or linked lists to implement stacks, you should check out this page that explains how arrays and linked lists are stored in memory.

This is how it looks like when we use an array as a stack:



Reasons to implement stacks using arrays:

- **Memory Efficient:** Array elements do not hold the next elements address like linked list nodes do.
- **Easier to implement and understand:** Using arrays to implement stacks require less code than using linked lists, and for this reason it is typically easier to understand as

- **Fixed size:** An array occupies a fixed part of the memory. This means that it could take up more memory than needed, or if the array fills up, it cannot hold more elements.

Note: When using arrays in Python for this tutorial, we are really using the Python 'list' data type, but for the scope of this tutorial the 'list' data type can be used in the same way as an array. Learn more about Python lists here.

Since Python lists has good support for functionality needed to implement stacks, we start with creating a stack and do stack operations with just a few lines like this:

Example

Python:

```
stack = []

# Push
stack.append('A')
stack.append('B')
stack.append('C')
print("Stack: ", stack)

# Pop
element = stack.pop()
print("Pop: ", element)

# Peek
topElement = stack[-1]
print("Peek: ", topElement)

# isEmpty
isEmpty = not bool(stack)
print("isEmpty: ", isEmpty)
```

But to explicitly create a data structure for stacks, with basic operations, we should create a stack class instead. This way of creating stacks in Python is also more similar to how stacks can be created in other programming languages like C and Java.

Example

Python:

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, element):
        self.stack.append(element)

    def pop(self):
        if self.isEmpty():
            return "Stack is empty"
        return self.stack.pop()

    def peek(self):
        if self.isEmpty():
            return "Stack is empty"
        return self.stack[-1]

    def isEmpty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

# Create a stack
myStack = Stack()

myStack.push('A')
```

```
print("Pop: ", myStack.pop())

print("Peek: ", myStack.peek())

print("isEmpty: ", myStack.isEmpty())

print("Size: ", myStack.size())
```

Try it Yourself »

Stack Implementation using Linked Lists

A reason for using linked lists to implement stacks:

- **Dynamic size:** The stack can grow and shrink dynamically, unlike with arrays.

Reasons for **not** using linked lists to implement stacks:

- **Extra memory:** Each stack element must contain the address to the next element (the next linked list node).
- **Readability:** The code might be harder to read and write for some because it is longer and more complex.

This is how a stack can be implemented using a linked list.

Example

Python:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Stack:
```

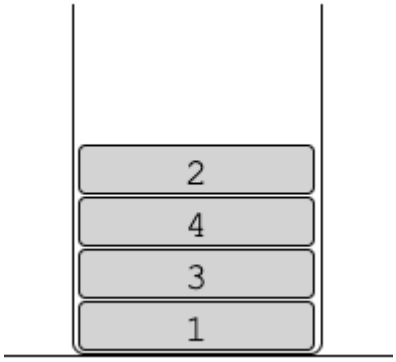
```
    def push(self, value):
        new_node = Node(value)
        if self.head:
            new_node.next = self.head
        self.head = new_node
        self.size += 1

    def pop(self):
        if self.isEmpty():
            return "Stack is empty"
        popped_node = self.head
        self.head = self.head.next
        self.size -= 1
        return popped_node.value

    def peek(self):
        if self.isEmpty():
            return "Stack is empty"
        return self.head.value

    def isEmpty(self):
        return self.size == 0

    def stackSize(self):
        return self.size

myStack = Stack()
myStack.push('A')
myStack.push('B')
myStack.push('C')

print("Pop: ", myStack.pop())
print("Peek: ", myStack.peek())
print("isEmpty: ", myStack.isEmpty())
print("Size: ", myStack.stackSize())
```

Try it Yourself »

Test Yourself With Exercises

Exercise:

The image below represents a "Stack" data structure.

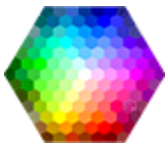


Running the `peek()` method on the Stack above, what is returned?

Submit Answer »

Start the Exercise

COLOR PICKER



Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate