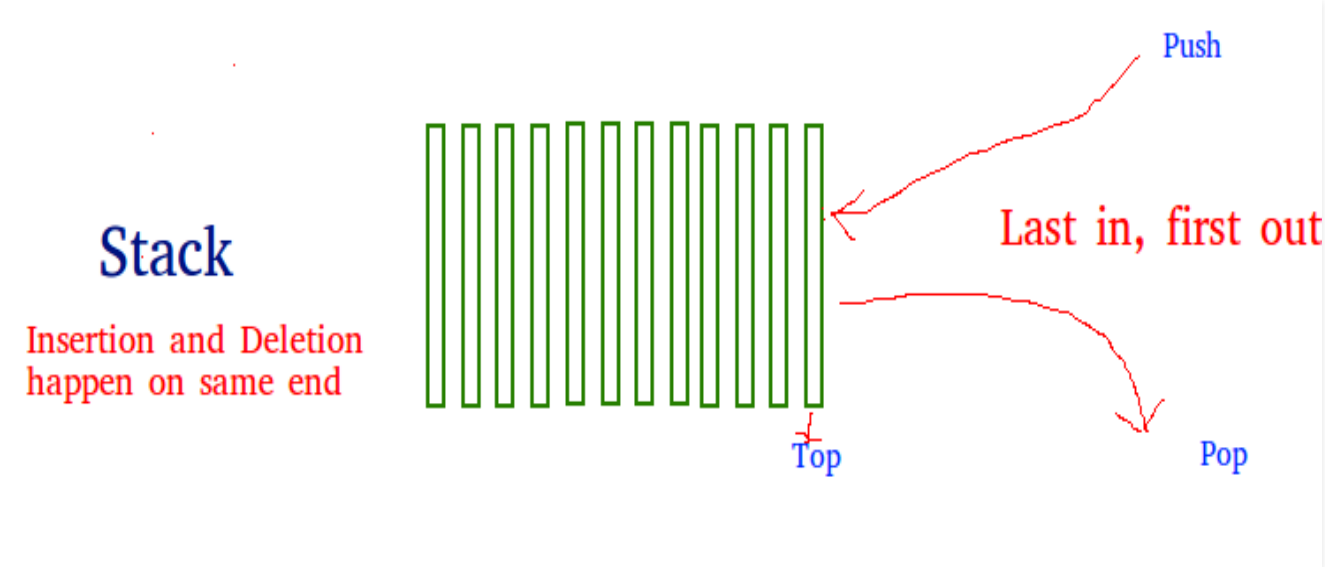
Stack in Python

Difficulty Level : Easy • Last Updated : 14 Oct, 2020

A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.



The functions associated with stack are:

- **empty()** – Returns whether the stack is empty – Time Complexity : O(1)
- **size()** – Returns the size of the stack – Time Complexity : O(1)
- **top()** – Returns a reference to the top most element of the stack – Time Complexity : O(1)
- **push(g)** – Adds the element 'g' at the top of the stack – Time Complexity : O(1)
- **pop()** – Deletes the top most element of the stack – Time Complexity : O(1)

Implementation

There are various ways from which a stack can be implemented in Python. This article

library.

Stack in Python can be implemented using following ways:

- list
- collections.deque
- queue.LifoQueue

Implementation using list:

Python's buil-in data structure list can be used as a stack. Instead of push(), append() is used to add elements to the top of stack while pop() removes the element in LIFO order. Unfortunately, list has a few shortcomings. The biggest issue is that it can run into speed issue as it grows. The items in list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently hold it, then Python needs to do some memory allocations. This can lead to some append() calls taking much longer than other ones.

Python3

```
# Python program to
# demonstrate stack implementation
# using list


stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')
```

```
# pop() fucntion to pop
# element from stack in
# LIFO order
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are poped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

Output:

```
Initial stack
['a', 'b', 'c']

Elements poped from stack:
c
b
a
```

Related Articles

```
Traceback (most recent call last):
  File "/home/2426bc32be6a59881fde0eec91247623.py", line 25, in <module>
    print(stack.pop())
IndexError: pop from empty list
```

Implementation using collections.deque:

Python stack can be implemented using deque class from collections module. Deque is preferred over list in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity.
The same methods on deque as seen in list are used, append() and pop().

Python3

```
# Python program to
# demonstrate stack implementation
```

```
from collections import deque

stack = deque()

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack:')
print(stack)

# pop() fucntion to pop
# element from stack in
# LIFO order
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are poped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

Output:

```
Initial stack:
deque(['a', 'b', 'c'])

Elements poped from stack:
c
b
a

Stack after elements are poped:
deque([])

Traceback (most recent call last):
  File "/home/97171a8f6fead6988ea96f86e4b01c32.py", line 29, in <module>
    print(stack.pop())
IndexError: pop from an empty deque
```

Implementation using queue module

There are various functions available in this module:

- **maxsize** – Number of items allowed in the queue.
- **empty()** – Return True if the queue is empty, False otherwise.
- **full()** – Return True if there are maxsize items in the queue. If the queue was initialized with maxsize=0 (the default), then full() never returns True.
- **get()** – Remove and return an item from the queue. If queue is empty, wait until an item is available.
- **get_nowait()** – Return an item if one is immediately available, else raise QueueEmpty.
- **put(item)** – Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.
- **put_nowait(item)** – Put an item into the queue without blocking.
- **qsize()** – Return the number of items in the queue. If no free slot is immediately available, raise QueueFull.

Python3

```
# Python program to
# demonstrate stack implementation
# using queue module


from queue import LifoQueue

# Initializing a stack
stack = LifoQueue(maxsize = 3)

# qsize() show the number of elements
# in the stack
print(stack.qsize())

# put() function to push
# element in the stack
stack.put('a')
stack.put('b')
stack.put('c')

print("Full: ", stack.full())
print("Size: ", stack.qsize())

# get() fucntion to pop
# element from stack in
# LIFO order
print('\nElements poped from the stack')
print(stack.get())
print(stack.get())
print(stack.get())
```

Output:

```
0
Full:  True
Size:  3

Elements poped from the stack
c
b
a

Empty:  True
```

Implementation using singly linked list

The linked list has two methods addHead(item) and removeHead() that run in constant time. These two methods are suitable to implement a stack.

- **getSize()** – Get the number of items in the stack.
- **isEmpty()** – Return True if the stack is empty, False otherwise.
- **peek()** – Return the top item in the stack. If the stack is empty, raise an exception.
- **push(value)** – Push a value into the head of the stack.
- **pop()** – Remove and return a value in the head of the stack. If the stack is empty, raise an exception.

Below is the implementation of the above approach:

Python3

```
# Python program to demonstrate
# stack implementation using a linked list.
# node class
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Stack:

    # Initializing a stack.
    # Use a dummy node, which is
    # easier for handling edge cases.
    def __init__(self):
        self.head = Node("head")
        self.size = 0

    # String representation of the stack
```

```
        while cur:
            out += str(cur.value) + "->"
            cur = cur.next
        return out[:-3]

    # Get the current size of the stack
    def getSize(self):
        return self.size

    # Check if the stack is empty
    def isEmpty(self):
        return self.size == 0

    # Get the top item of the stack
    def peek(self):

        # Sanitary check to see if we
        # are peeking an empty stack.
        if self.isEmpty():
            raise Exception("Peeking from an empty stack")
        return self.head.next.value

    # Push a value into the stack.
    def push(self, value):
        node = Node(value)
        node.next = self.head.next
        self.head.next = node
        self.size += 1

    # Remove a value from the stack and return.
    def pop(self):
        if self.isEmpty():
            raise Exception("Popping from an empty stack")
        remove = self.head.next
        self.head.next = self.head.next.next
        self.size -= 1
        return remove.value

# Driver Code
if __name__ == "__main__":
    stack = Stack()
    for i in range(1, 11):
        stack.push(i)
    print(f"Stack: {stack}")

    for _ in range(1, 6):
        remove = stack.pop()
        print(f"Pop: {remove}")
    print(f"Stack: {stack}")
```

Output:

```
Stack: 10 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1
```

```
Pop: 9
Pop: 8
Pop: 7
Pop: 6

Stack: 5 -> 4 -> 3 -> 2 -> 1
```

nikhilaggarwal3
@nikhilaggarwal3

Vote for difficulty

Current difficulty : Easy

Easy | Normal | Medium | Hard | Expert

Improved By : nidhi_biet, duybao200300

Article Tags : Python, Stack

Practice Tags : Stack

Improve Article | Report Issue

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments