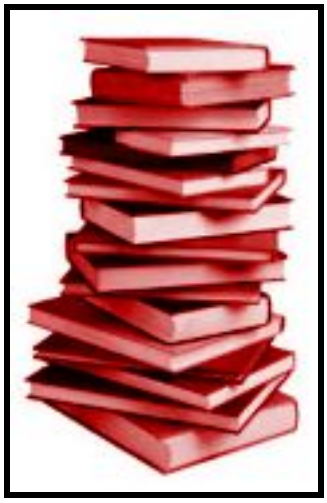
Stacks

Introduction

- Stacks are simple data structures that allow us to store and retrieve data sequentially.
- A stack is a linear data structure like arrays and linked lists.
- It is an abstract data type**(ADT)**.
- In a stack, the order in which the data arrives is essential. It follows the LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out.**
- Consider the example of a pile of books:

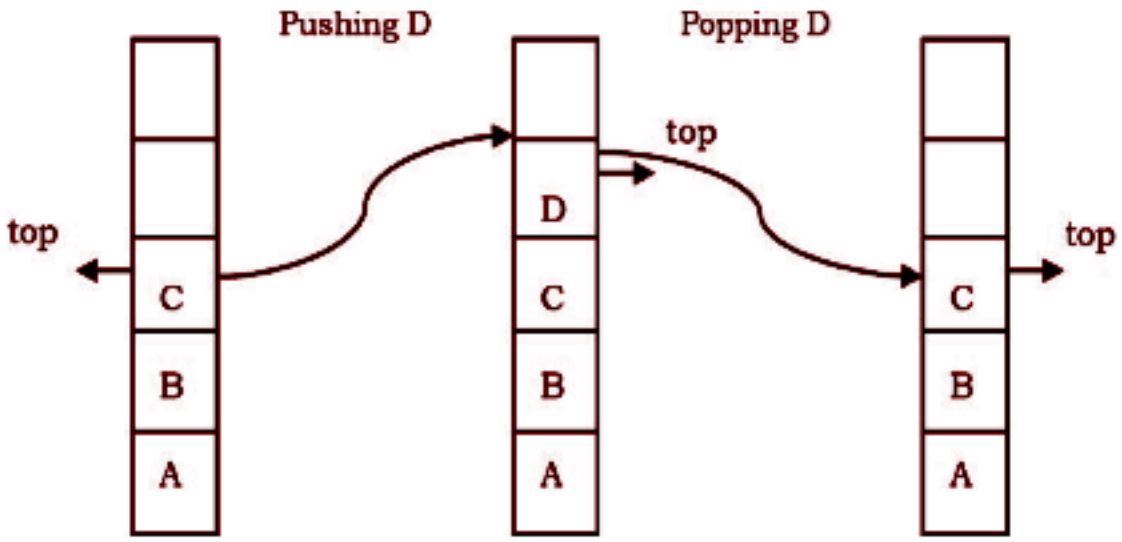


Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

Like deletion, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as **LIFO.**

Operations on the stack:

- In a stack, insertion and deletion are done at one end, called **top**.
- **Insertion**: This is known as a **push** operation.
- **Deletion**: This is known as a **pop** operation.



Main stack operations

• `Push (int data)`: Insert data onto the stack.

• `int Pop()`: Removes and returns the last inserted element from the stack.

Auxiliary stack operations

• `int Top()`: Returns the last inserted element without removing it.

• `int Size()`: Returns the number of elements stored in the stack.

• `int IsEmptyStack()`: Indicates whether any elements are stored in the stack or not.

• `int IsFullStack()`: Indicates whether the stack is full or not.

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Exceptions

- Attempting the execution of an operation may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- Attempting the execution of pop() on an empty stack throws an exception called **Stack Underflow**.
- Trying to push an element in a full-stack throws an exception called **Stack Overflow**.

Implementing stack- Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the **top** element.

Consider the given implementation in Python for more understanding:

```
class Stack:
    #Constructor
    def __init__(self):
        self.stack = list()
        self.maxSize = 8 #Maximum size of the list
        self.top = 0 #Top element

    #Adds element to the Stack
    def push(self,data):
        if self.top>=self.maxSize:#Stack Overflow
            return ("Stack Full!")
        self.stack[top]= data #Assign the new element at 'top'
        self.top += 1 #Increment top
        return True

    #Removes element from the stack
    def pop(self):
        if self.top<=0:#Stack Underflow
            return ("Stack Empty!")
        item = self.stack[top-1]#Topmost element
        self.top -= 1 #Simply decrementing top
        return item #Returning the removed element

    #Size of the stack
    def size(self):
        return self.top
```

Limitations of Simple Array Implementation

In other programming languages like C++, Java, etc, the maximum size of the an array must first be defined i.e. it is fixed and it cannot be changed. However, in Python, arrays are resizable by nature. This means that even though Python internally handles the resizing of arrays, it is still very expensive.

Stack using Linked Lists

Till now we have learned how to implement a stack using arrays, but as discussed earlier, we can also create a stack with the help of linked lists. All the five functions that stacks can perform could be made using linked lists:

```
class Node:#Node of a Linked List
    def __init__(self, data):
       self.data = data
       self.next = None

class Stack:#Stack Implementation using LL
    def __init__(self):
        self.head = None

    def push(self, data):
        if self.head is None:
            self.head = Node(data)
        else:
            new_node = Node(data)
            new_node.next = self.head
            self.head = new_node

    def pop(self):
        if self.head is None:
            return None
        else:
            popped = self.head.data
            self.head = self.head.next
            return popped
```

Inbuilt Stack in Python

- The **queue** module also has a **LIFO Queue**, which is basically a **Stack**.

There are various functions available in this module:

- **maxsize** – Returns the maximum number of items allowed in the stack.
- **empty()** – Returns **True** if the stack is empty, otherwise it returns **False**.
- **get()** – Remove and return an item from the stack.
- **put(item)** – Put an item into the stack.
- **qsize()** – Return the number of items currently present in the stack.

```
from queue import Queue

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

print(q.maxsize())# Maximum size of the stack

stack.put('14') # Adding elements to the stack
stack.put('28')
stack.put('36')

print("\nisFull: ", stack.full()) # Check if the stack is full

print("\nElement dequeued from the stack: ")
print(stack.get()) # Removing an element from stack
```

We get the following output:

```
3
isFull:  True
Element dequeued from the stack:
36 #Stack follows LIFO
```

Problem Statement- Balanced Parenthesis

For a given string expression containing only round brackets or parentheses, check if they are balanced or not. Brackets are said to be balanced if the bracket which opens last, closes first. You need to return a boolean value indicating whether the expression is balanced or not.

Approach:

- We will use stacks.
- Each time, when an open parenthesis is encountered push it in the stack, and when closed parenthesis is encountered, match it with the top of the stack and pop it.
- If the stack is empty at the end, return Balanced otherwise, Unbalanced.

Python Code:

```
open = ["[","{","("]
close = ["]","}",")"]

# Function to check parentheses
def checkBalanced(inputStr):
    s = [] #The stack
    for i in inputStr:
        if i in open:
            s.append(i)
        elif i in close:
            position = close.index(i)
            if ((len(s)>0) and (open[position]==s[len(s)-1])):
                s.pop()
            else:
                return "Unbalanced"
    if len(s) == 0:
        return "Balanced"
    else:
        return "Unbalanced"
```

Practice Problems:

- https://www.hackerrank.com/challenges/equal-stacks/problem
- https://www.hackerrank.com/challenges/simple-text-editor/problem
- https://www.techiedelight.com/design-a-stack-which-returns-minimum-element-without-using-auxiliary-stack/
- https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/practice-problems/algorithm/monk-and-order-of-phoenix/