Stack

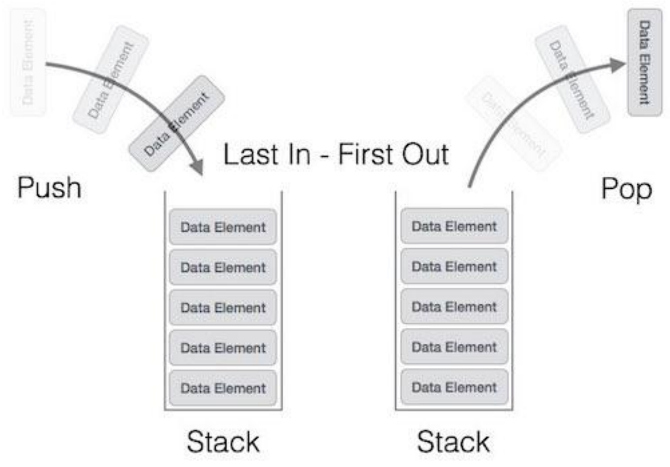
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- push() − Pushing (storing) an element on the stack.
- pop() − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- peek() − get the top data element of the stack, without removing it.
- isFull() − check if stack is full.
- isEmpty() − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

Stack Operation using Python List

In a stack the element inserted last in sequence will come out first as we can remove only from the top of the stack. Such feature is known as Last in First Out(LIFO) feature. The operations of adding and removing the elements is known as PUSH and POP. In the following program we implement it as add and and remove functions. We dclare an empty list and use the append() and pop() methods to add and remove the data elements.

PUSH into a Stack

```
class Stack:

   def __init__(self):
      self.stack = []

   def add(self, dataval):
# Use list append method to add element
      if dataval not in self.stack:
         self.stack.append(dataval)
         return True
      else:
         return False
# Use peek to look at the top of the stack

   def peek(self):
            return self.stack[-1]

AStack = Stack()
AStack.add("Mon")
AStack.add("Tue")
AStack.peek()
print(AStack.peek())
AStack.add("Wed")
AStack.add("Thu")
print(AStack.peek())
```

When the above code is executed, it produces the following result:

```
Tue
Thu
```

POP from a Stack

As we know we can remove only the top most data element from the stack, we implement a python program which does that. The remove function in the following program returns the top most element. we check the top element by calculating the size of the stack first and then use the in-built pop() method to find out the top most element.

```
class Stack:

   def __init__(self):
      self.stack = []

   def add(self, dataval):
# Use list append method to add element
      if dataval not in self.stack:
         self.stack.append(dataval)
         return True
      else:
         return False

# Use list pop method to remove element
   def remove(self):
      if len(self.stack) <= 0:
         return ("No element in the Stack")
      else:
         return self.stack.pop()

AStack = Stack()
AStack.add("Mon")
AStack.add("Tue")
AStack.add("Wed")
AStack.add("Thu")
print(AStack.remove())
print(AStack.remove())
```

Another programming example

```
# Code to demonstrate Implementation of   # stack using list
x = ["Python", "C", "Android"]
x.push("Java")
x.push("C++")
print(x)
print(x.pop())
print(x)
print(x.pop())
```

9. print(x)

Another Example of Stack

```
 letters = []
# Let's push some letters into our list
letters.append('c')
letters.append('a')
letters.append('t')
letters.append('g')
# Now let's pop letters, we should get 'g'
last_item = letters.pop()
print(last_item)
# If we pop again we'll get 't'
last_item = letters.pop()
print(last_item)
# 'c' and 'a' remain
print(letters) # ['c', 'a']
```

Queue

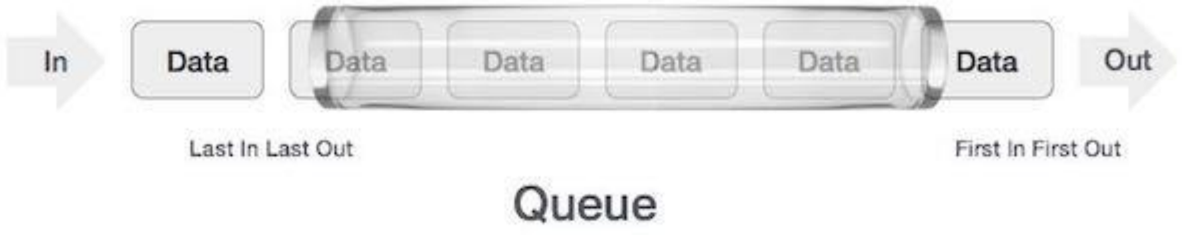
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

The queue data structure also means the same where the data elements are arranged in a queue. The uniqueness of queue lies in the way items are added and removed. The items are allowed at on end but removed form the other end. So it is a First-in-First out method. An queue can be implemented using python list where we can use the insert() and pop() methods to add and remove elements. Their is no insertion as data elements are always added at the end of the queue.

Adding Elements to a Queue

In the below example we create a queue class where we implement the First-in-First-Out method. We use the in-built insert method for adding data elements.

```
class Queue:

  def __init__(self):
      self.queue = list()

  def addtoq(self,dataval):
# Insert method to add element
      if dataval not in self.queue:
          self.queue.insert(0,dataval)
          return True
      return False

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

TheQueue = Queue()
TheQueue.addtoq("Mon")
TheQueue.addtoq("Tue")
TheQueue.addtoq("Wed")
print(TheQueue.size())
```

Removing Element from a Queue

In the below example we create a queue class where we insert the data and then remove the data using the in-built pop method.

.

```
class Queue:

  def __init__(self):
      self.queue = list()

  def addtoq(self,dataval):
# Insert method to add element
      if dataval not in self.queue:
          self.queue.insert(0,dataval)
          return True
      return False
# Pop method to remove element
  def removefromq(self):
      if len(self.queue)>0:
          return self.queue.pop()
      return ("No elements in Queue!")

TheQueue = Queue()
TheQueue.addtoq("Mon")
TheQueue.addtoq("Tue")
TheQueue.addtoq("Wed")
print(TheQueue.removefromq())
print(TheQueue.removefromq())
```

Another programming example of Queue using Queue library

```
import queue
# Queue is created as an object 'L'
L = queue.Queue(maxsize=10)

# Data is inserted in 'L' at the end using put()
L.put(9)
L.put(6)
L.put(7)
L.put(4)
# get() takes data from
# from the head
# of the Queue
print(L.get())
```

14. print(L.get())
15. print(L.get())
16. print(L.get())

Another Example of Queue

```
 fruits = []
# Let's enqueue some fruits into our list
fruits.append('banana')
fruits.append('grapes')
fruits.append('mango')
fruits.append('orange')
# Now let's dequeue our fruits, we should get 'banana'
first_item = fruits.pop(0)
print(first_item)
# If we dequeue again we'll get 'grapes'
first_item = fruits.pop(0)
print(first_item)
# 'mango' and 'orange' remain
print(fruits) # ['c', 'a']
```

****