Dauphine | PSL
UNIVERSITÉ PARIS

→ COMPUTER SCIENCE, DECISION-MAKING, AND DATA

# Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Remy Belmonte remy.belmonte@dauphine.eu

Masterclass 2

Electronic copy available at: https://ssrn.com/abstract=3953786

# Reminder of the objective of this course

- People often learn about data structures out of context
- But in this course you will learn foundational concepts by building a real application with python and Flask (we will start in session 3)!

- To learn the ins and outs of the essential data structure, experiencing in practice has proved to be a much more powerful way to learn data structures


Electronic copy available at: https://ssrn.com/abstract=3953786

# Outline

1. Stack
   a) Concepts
   b) Implementation choice
   c) Corresponding codes
2. Queue
   a) Concepts
   b) Implementation choice
   c) Corresponding codes


Electronic copy available at: https://ssrn.com/abstract=3953786

# What is a stack

A stack is a simple data structure used for storing data (similar to Linked Lists). In a stack, the order in which the data arrives is important. A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and they are placed on the top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

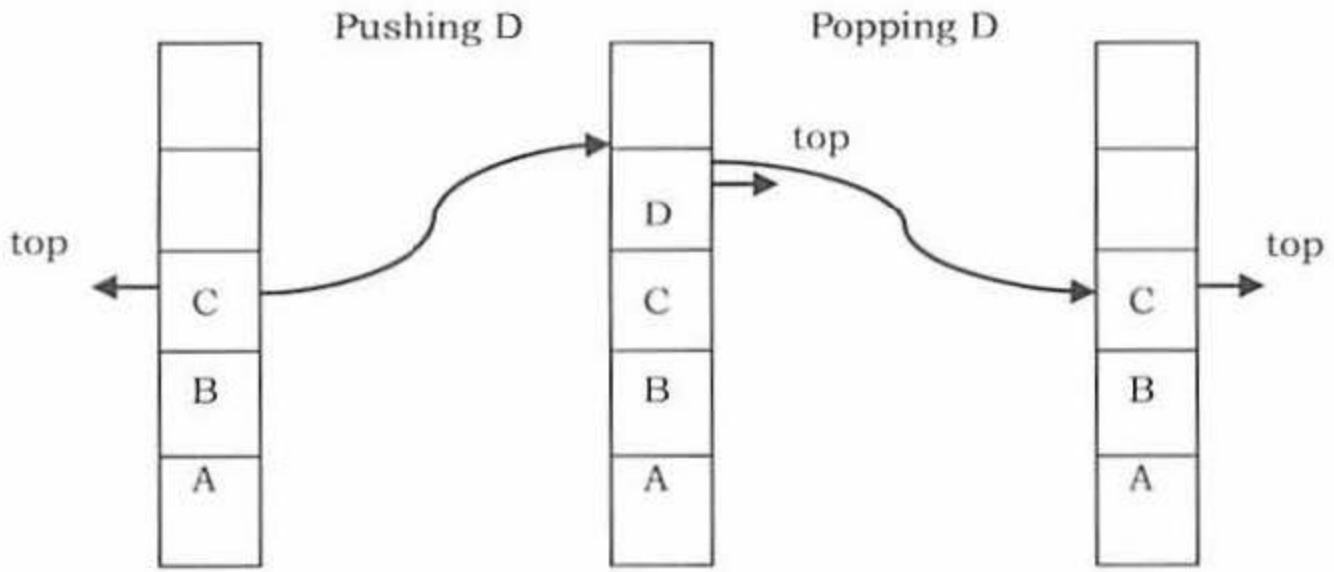
**Definition:** A *stack* is an ordered list in which insertion and deletion are done at one end, called *top*. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.



Electronic copy available at: https://ssrn.com/abstract=3953786

# Special names

Special names are given to the two changes that can be made to a stack. When an element is inserted in a stack, the concept is called *push*, and when an element is removed from the stack, the concept is called *pop*. Trying to pop out an empty stack is called *underflow* and trying to push an element in a full stack is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshots of the stack.





Electronic copy available at: https://ssrn.com/abstract=3953786

# How stack are used?

Consider a working day in the office. Let us assume a developer is working on a long-term project. The manager then gives the developer a new task which is more important. The developer puts the long-term project aside and begins work on the new task. The phone rings, and this is the highest priority as it must be answered immediately. The developer pushes the present task into the pending tray and answers the phone.

When the call is complete the task that was abandoned to answer the phone is retrieved from the pending tray and work progresses. To take another call, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.


Electronic copy available at: https://ssrn.com/abstract=3953786

# Stack advanced data structure

## 4.3 Stack ADT

The following operations make a stack an ADT. For simplicity, assume the data is an integer type.

### Main stack operations

- Push (int data): Inserts *data* onto 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.


Electronic copy available at: https://ssrn.com/abstract=3953786

# Direct applications

## 4.4 Applications

Following are some of the applications in which stacks play an important role.

### Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer to *Problems* section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML


Electronic copy available at: https://ssrn.com/abstract=3953786

# Implementation

## 4.5 Implementation

There are many ways of implementing stack ADT; below are the commonly used methods.

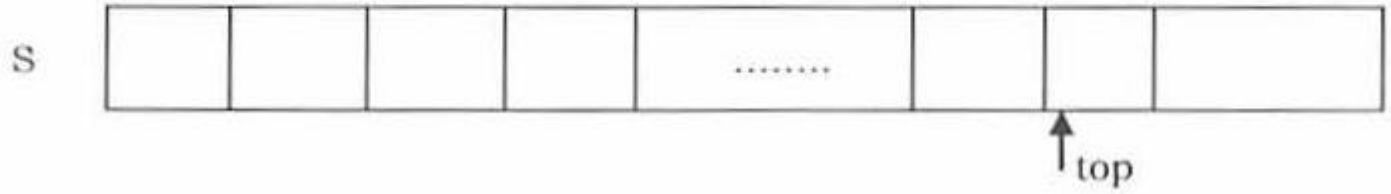
- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation



Electronic copy available at: https://ssrn.com/abstract=3953786

# 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.





Electronic copy available at: https://ssrn.com/abstract=3953786

The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from an empty stack it will throw *stack empty exception*.

```
class Stack(object):
 def __init__(self, limit = 10):
        self.stk = []
        self.limit = limit

 def isEmpty(self):
        return len(self.stk) <= 0

 def push(self, item):
        if len(self.stk) >= self.limit:
                print 'Stack Overflow!'
        else:
                self.stk.append(item)
        print 'Stack after Push',self.stk

 def pop(self):
        if len(self.stk) <= 0:
                print 'Stack Underflow!'
                return 0
        else:
                return self.stk.pop()

 def peek(self):
        if len(self.stk) <= 0:
                print 'Stack Underflow!'
                return 0
        else:
                return self.stk[-1]

 def size(self):
        return len(self.stk)
```



Electronic copy available at: https://ssrn.com/abstract=3953786

# Performance and limitations?

## Performance & Limitations

**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 IsFullStack() | $O(1)$ |
| Time Complexity of DeleteStack() | $O(1)$ |


Electronic copy available at: https://ssrn.com/abstract=3953786

# Limitations?

**Limitations**

The maximum size of the stack must first be defined and it cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.



Electronic copy available at: https://ssrn.com/abstract=3953786

## Dynamic Array Implementation

First, let's consider how we implemented a simple array based stack. We took one index variable *top* which points to the index of the most recently inserted element in the stack. To insert (or push) an element, we increment *top* index and then place the new element at that index.

Similarly, to delete (or pop) an element we take the element at *top* index and then decrement the *top* index. We represent an empty queue with *top* value equal to −1. The issue that still needs to be resolved is what we do when all the slots in the fixed size array stack are occupied?

**First try:** What if we increment the size of the array by 1 every time the stack is full?

- Push(): increase size of S|| by 1
- Pop(): decrease size of S|| by 1



Electronic copy available at: https://ssrn.com/abstract=3953786

### Problems with this approach?

This way of incrementing the array size is too expensive. Let us see the reason for this. For example, at $n = 1$, to push an element create a new array of size 2 and copy all the old array elements to the new array, and at the end add the new element. At $n = 2$, to push an element create a new array of size 3 and copy all the old array elements to the new array, and at the end add the new element.

Similarly, at $n = n - 1$, if we want to push an element create a new array of size $n$ and copy all the old array elements to the new array and at the end add the new element. After $n$ push operations the total time $T(n)$ (number of copy operations) is proportional to $1 + 2 + \ldots + n \approx O(n^2)$.

### Alternative Approach: Repeated Doubling

Let us improve the complexity by using the array *doubling* technique. If the array is full, create a new array of twice the size, and copy the items. With this approach, pushing $n$ items takes time proportional to $n$ (not $n^2$).

For simplicity, let us assume that initially we started with $n = 1$ and moved up to $n = 32$. That means, we do the doubling at $1, 2, 4, 8, 16$. The other way of analyzing the same approach is: at $n = 1$, if we want to add (push) an element, double the current size of the array and copy all the elements of the old array to the new array.

At $n = 1$, we do 1 copy operation, at $n = 2$, we do 2 copy operations, and at $n = 4$, we do 4 copy operations and so on. By the time we reach $n = 32$, the total number of copy operations is $1 + 2 + 4 + 8 + 16 = 31$ which is approximately equal to $2n$ value (32). If we observe carefully, we are doing the doubling operation $logn$ times.

Now, let us generalize the discussion. For $n$ push operations we double the array size $logn$ times. That means, we will have $logn$ terms in the expression below. The total time $T(n)$ of a series of $n$ push operations is proportional to


Electronic copy available at: https://ssrn.com/abstract=3953786

# Question compute the complexity?

$$1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n = \ ?$$



Electronic copy available at: https://ssrn.com/abstract=3953786

# Solution

$$
\begin{aligned}
1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n &= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1 \\
&= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right) \\
&= n(2) \approx 2n = O(n)
\end{aligned}
$$



Electronic copy available at: https://ssrn.com/abstract=3953786

# Implementation

```
class Stack(object):
 def __init__(self, limit = 10):
        self.stk = limit*[None]
        self.limit = limit

 def isEmpty(self):
        return len(self.stk) <= 0

 def push(self, item):
        if len(self.stk) >= self.limit:
                self.resize()
        self.stk.append(item)
        print 'Stack after Push',self.stk

 def pop(self):
        if len(self.stk) <= 0:
                print 'Stack Underflow!'
                return 0
        else:
                return self.stk.pop()
```


Electronic copy available at: https://ssrn.com/abstract=3953786

```
    def peek(self):
        if len(self.stk) <= 0:
            print 'Stack Underflow!'
            return 0
        else:
            return self.stk[-1]
    def size(self):
        return len(self.stk)
    def resize(self):
        newStk = list(self.stk)
        self.limit = 2*self.limit
        self.stk = newStk
our_stack = Stack(5)
our_stack.push("1")
our_stack.push("21")
our_stack.push("14")
our_stack.push("11")
our_stack.push("31")
our_stack.push("14")
our_stack.push("15")
our_stack.push("19")
our_stack.push("3")
our_stack.push("99")
our_stack.push("9")
print our_stack.peek()
print our_stack.pop()
print our_stack.peek()
print our_stack.pop()
```



Electronic copy available at: https://ssrn.com/abstract=3953786

# Performance

## Performance

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

| | |
|---|---|
| Space Complexity (for $n$ push operations) | $O(n)$ |
| Time Complexity of CreateStack() | $O(1)$ |
| Time Complexity of Push() | $O(1)$ (Average) |
| Time Complexity of Pop() | $O(1)$ |
| Time Complexity of Top() | $O(1)$ |
| Time Complexity of IsEmptyStack() | $O(1)$) |
| Time Complexity of IsFullStack() | $O(1)$ |
| Time Complexity of DeleteStack() | $O(1)$ |

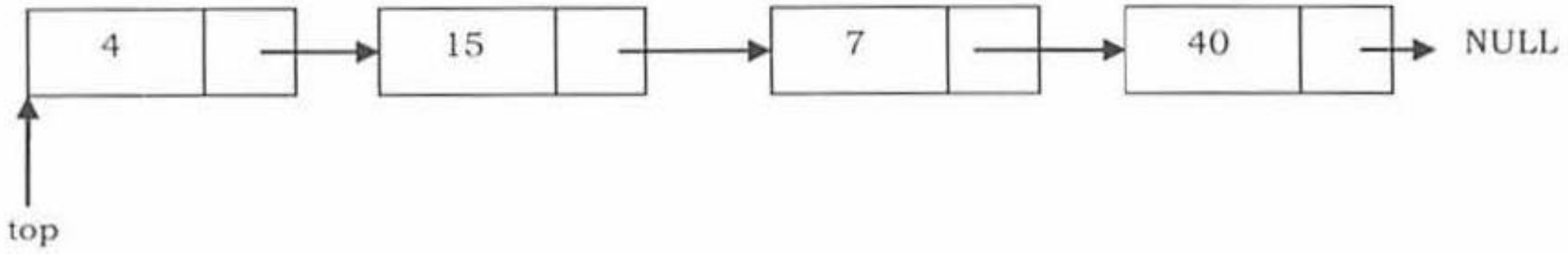
**Note:** Too many doublings may cause memory overflow exception.



Electronic copy available at: https://ssrn.com/abstract=3953786

## Linked List Implementation

The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).





Electronic copy available at: https://ssrn.com/abstract=3953786

```
#Node of a Singly Linked List
class Node:
    #constructor
    def __init__(self):
        self.data = None
        self.next = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
        return self.next != None
```


Electronic copy available at: https://ssrn.com/abstract=3953786

```
        return self.next != None
class Stack(object):
    def __init__(self, data=None):
        self.head = None
        if data:
            for data in data:
                self.push(data)
    def push(self, data):
        temp = Node()
        temp.setData(data)
        temp.setNext(self.head)
        self.head = temp
    def pop(self):
        if self.head is None:
            raise IndexError
        temp = self.head.getData()
        self.head = self.head.getNext()
        return temp
    def peek(self):
        if self.head is None:
            raise IndexError
        return self.head.getData()
our_list = ["first", "second", "third", "fourth"]
our_stack = Stack(our_list)
print our_stack.pop()
print our_stack.pop()
```


Electronic copy available at: https://ssrn.com/abstract=3953786

## Performance

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

| | |
|---|---|
| Space Complexity (for $n$ push operations) | $O(n)$ |
| Time Complexity of CreateStack() | $O(1)$ |
| Time Complexity of Push() | $O(1)$ (Average) |
| Time Complexity of Pop() | $O(1)$ |
| Time Complexity of Top() | $O(1)$ |
| Time Complexity of IsEmptyStack() | $O(1)$ |
| Time Complexity of DeleteStack() | $O(n)$ |



Electronic copy available at: https://ssrn.com/abstract=3953786

# 4.6 Comparison of Implementations

## Comparing Incremental Strategy and Doubling Strategy

We compare the incremental strategy and doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ push operations. We start with an empty stack represented by an array of size 1.

---

We call *amortized* time of a push operation is the average time taken by a push over the series of operations, that is, $T(n)/n$.

**Incremental Strategy**

The amortized time (average time per operation) of a push operation is $O(n)$ $[O(n^2)/n]$.

**Doubling Strategy**

In this method, the amortized time of a push operation is $O(1)$ $[O(n)/n]$.

**Note**: For analysis, refer to the *Implementation* section.



Electronic copy available at: https://ssrn.com/abstract=3953786

## Comparing Array Implementation and Linked List Implementation

**Array Implementation**

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of $n$ operations (starting from empty stack) – "*amortized*" bound takes time proportional to $n$.

**Linked List Implementation**

- Grows and shrinks gracefully.
- Every operation takes constant time O(1).
- Every operation uses extra space and time to deal with references.


Electronic copy available at: https://ssrn.com/abstract=3953786

# Queue

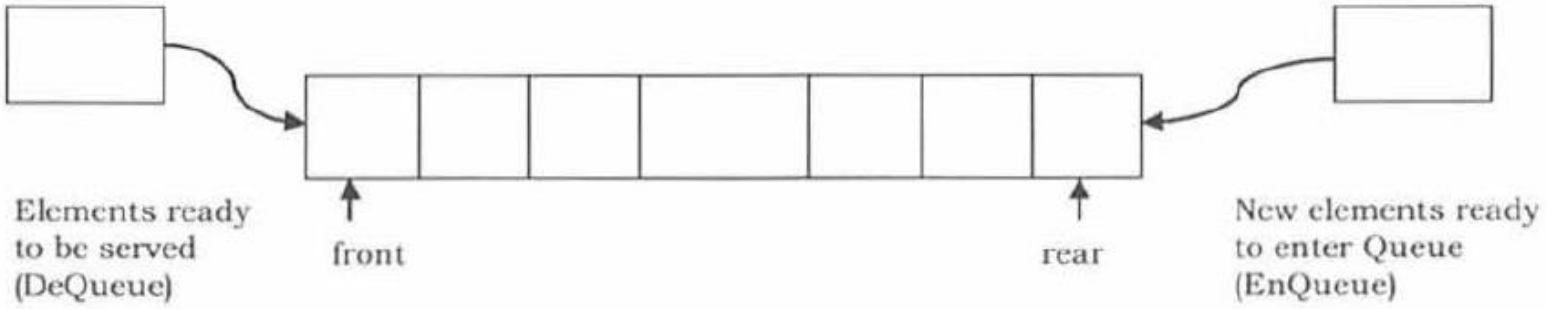
## 5.1 What is a Queue?

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

**Definition:** A *queue* is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.

Similar to *Stacks*, special names are given to the two changes that can be made to a queue. When an element is inserted in a queue, the concept is called *EnQueue*, and when an element is removed from the queue, the concept is called *DeQueue*.

*DeQueueing* an empty queue is called *underflow* and *EnQueuing* an element in a full queue is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of the queue.





Electronic copy available at: https://ssrn.com/abstract=3953786

## 5.2 How are Queues Used

The concept of a queue can be explained by observing a line at a reservation counter. When we enter the line we stand at the end of the line and the person who is at the front of the line is the one who will be served next. He will exit the queue and be served.

As this happens, the next person will come at the head of the line, will exit the queue and will be served. As each person at the head of the line keeps exiting the queue, we move towards the head of the line. Finally we will reach the head of the line and we will exit the queue and be served. This behavior is very useful in cases where there is a need to maintain the order of arrival.

## 5.3 Queue ADT

The following operations make a queue an ADT. Insertions and deletions in the queue must follow the FIFO scheme. For simplicity we assume the elements are integers.


Electronic copy available at: https://ssrn.com/abstract=3953786

**Main Queue Operations**

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

**Auxiliary Queue Operations**

- int Front(): Returns the element at the front without removing it
- int QueueSize(): Returns the number of elements stored in the queue
- int IsEmptyQueue(): Indicates whether no elements are stored in the queue or not

## 5.4 Exceptions

Similar to other ADTs, executing *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and executing *EnQueue* on a full queue throws a "*Full Queue Exception*".



Electronic copy available at: https://ssrn.com/abstract=3953786

## 5.5 Applications

Following are the some of the applications that use queues.

**Direct Applications**

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

**Indirect Applications**

- Auxiliary data structure for algorithms
- Component of other data structures

## 5.6 Implementation

There are many ways (similar to Stacks) of implementing queue operations and some of the commonly used methods are listed below.

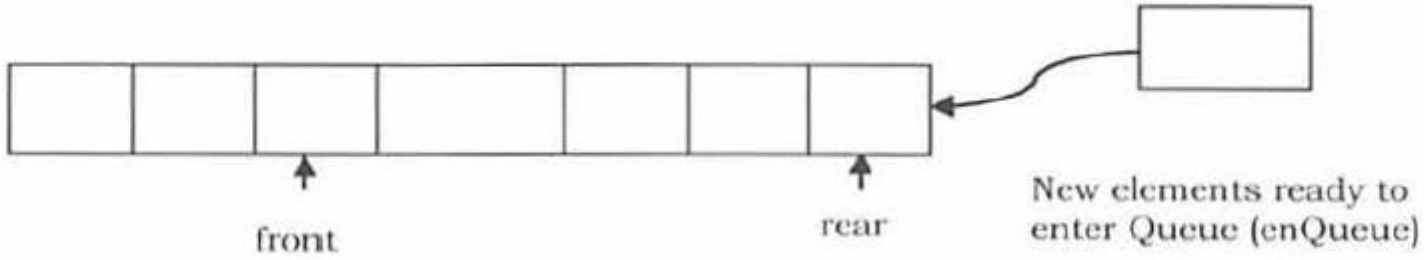
- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked list implementation


Electronic copy available at: https://ssrn.com/abstract=3953786

## Why Circular Arrays?

First, let us see whether we can use simple arrays for implementing queues as we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at the other end. After performing some insertions and deletions the process becomes easy to understand.

In the example shown below, it can be seen clearly that the initial slots of the array are getting wasted. So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat the last element and the first array elements as contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slot.

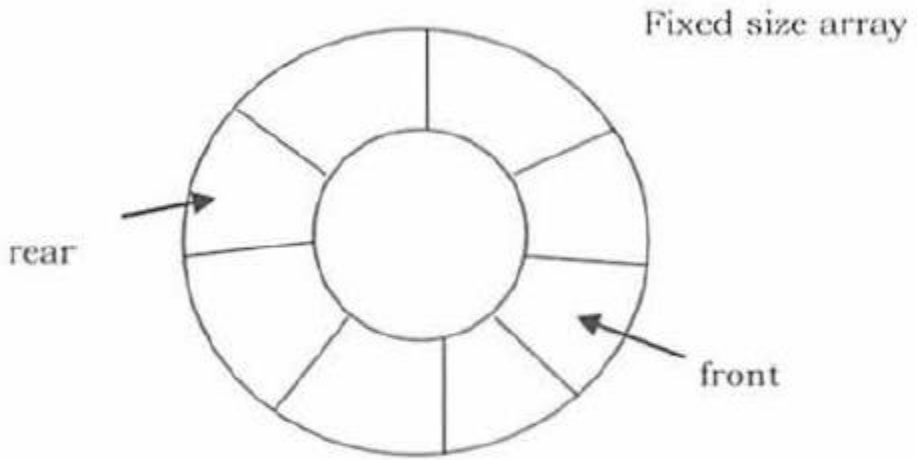




Electronic copy available at: https://ssrn.com/abstract=3953786

**Note:** The simple circular array and dynamic circular array implementations are very similar to stack array implementations. Refer to *Stacks* chapter for analysis of these implementations.

## Simple Circular Array Implementation




This simple implementation of Queue ADT uses an array. In the array, we add elements circularly and use two variables to keep track of the start element and end element. Generally, *front* is used to indicate the start element and *rear* is used to indicate the end element in the queue.

The array storing the queue elements may become full. An *EnQueue* operation will then throw a *full queue exception*. Similarly, if we try deleting an element from an empty queue it will throw *empty queue exception*.

**Note:** Initially, both front and rear points to -1 which indicates that the queue is empty.


Electronic copy available at: https://ssrn.com/abstract=3953786

```
class Queue(object):
 def __init__(self, limit = 5):
        self.que = []
        self.limit = limit
        self.front = None
        self.rear = None
        self.size = 0
 def isEmpty(self):
        return self.size <= 0
 def enQueue(self, item):
        if self.size >= self.limit:
                print 'Queue Overflow!'
                return
        else:
                self.que.append(item)

        if self.front is None:
                self.front = self.rear = 0
        else:
                self.rear = self.size
        self.size += 1
        print 'Queue after enQueue',self.que
 def deQueue(self):
        if self.size <= 0:
                print 'Queue Underflow!'
                return 0
        else:
                self.que.pop(0)
                self.size -= 1
                if self.size == 0:
                        self.front = self.rear = None
                else:
                        self.rear = self.size-1
                print 'Queue after deQueue',self.que
```



Electronic copy available at: https://ssrn.com/abstract=3953786

```
                print 'Queue after deQueue',self.que
    def queueRear(self):
        if self.rear is None:
                print "Sorry, the queue is empty!"
```

```
                raise IndexError
        return self.que[self.rear]
    def queueFront(self):
        if self.front is None:
                print "Sorry, the queue is empty!"
                raise IndexError
        return self.que[self.front]
    def size(self):
        return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.deQueue()
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
```



Electronic copy available at: https://ssrn.com/abstract=3953786

## Performance and Limitations

**Performance:** Let $n$ be the number of elements in the queue:

| | |
|---|---|
| Space Complexity (for $n$ EnQueue operations) | $O(n)$ |
| Time Complexity of EnQueue() | $O(1)$ |
| Time Complexity of DeQueue() | $O(1)$ |
| Time Complexity of IsEmptyQueue() | $O(1)$ |
| Time Complexity of IsFullQueue() | $O(1)$ |
| Time Complexity of QueueSize() | $O(1)$ |
| Time Complexity of DeleteQueue() | $O(1)$ |

**Limitations:** The maximum size of the queue must be defined as prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.



Electronic copy available at: https://ssrn.com/abstract=3953786

## Dynamic Circular Array Implementation

```
class Queue(object):
 def __init__(self, limit = 5):
        self.que = []
        self.limit = limit
        self.front = None
        self.rear = None
        self.size = 0

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

 def enQueue(self, item):
        if self.size >= self.limit:
                self.resize()

        self.que.append(item)

        if self.front is None:
                self.front = self.rear = 0
        else:
                self.rear = self.size
        self.size += 1
```



Electronic copy available at: https://ssrn.com/abstract=3953786

```
        print 'Queue after enQueue',self.que
    def deQueue(self):
        if self.size <= 0:
            print 'Queue Underflow!'
            return 0
        else:
            self.que.pop(0)
            self.size -= 1
            if self.size == 0:
                self.front = self.rear = None
            else:
                self.rear = self.size-1
            print 'Queue after deQueue',self.que
    def queueRear(self):
        if self.rear is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.que[self.rear]
    def queueFront(self):
        if self.front is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.que[self.front]
    def size(self):
        return self.size
    def resize(self):
        newQue = list(self.que)
        self.limit = 2*self.limit
        self.que = newQue
```



Electronic copy available at: https://ssrn.com/abstract=3953786

Let $n$ be the number of elements in the queue.

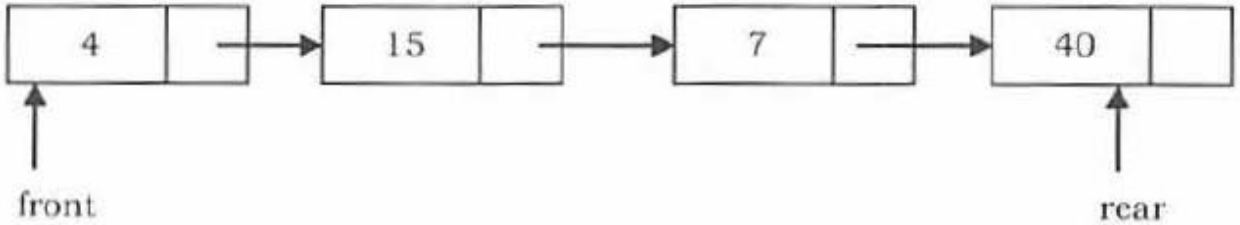
| | |
|---|---|
| Space Complexity (for $n$ EnQueue operations) | $O(n)$ |
| Time Complexity of EnQueue() | O(1) (Average) |
| Time Complexity of DeQueue() | O(1) |
| Time Complexity of QueueSize() | O(1) |
| Time Complexity of IsEmptyQueue() | O(1) |
| Time Complexity of IsFullQueue() | O(1) |
| Time Complexity of QueueSize() | O(1) |
| Time Complexity of DeleteQueue() | O(1) |



Electronic copy available at: https://ssrn.com/abstract=3953786

## Linked List Implementation

Another way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting an element at the end of the list. *DeQueue* operation is implemented by deleting an element from the beginning of the list.




```
#Node of a Singly Linked List
class Node:
 #constructor
 def __init__(self, data=None, next=None):
        self.data = data
        self.last = None
        self.next = next
 #method for setting the data field of the node
 def setData(self,data):
        self.data = data
 #method for getting the data field of the node
 def getData(self):
        return self.data
 #method for setting the next field of the node
 def setNext(self,next):
        self.next = next
 #method for getting the next field of the node
```


Electronic copy available at: https://ssrn.com/abstract=3953786

```
    def getNext(self):
            return self.next
    #method for setting the last field of the node
    def setLast(self,last):
            self.last = last
    #method for getting the last field of the node
    def getLast(self):
            return self.last
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
class Queue(object):
    def __init__(self, data=None):
            self.front = None
            self.rear = None
            self.size = 0
    def enQueue(self, data):
            self.lastNode = self.front
            self.front = Node(data, self.front)
            if self.lastNode:
                    self.lastNode.setLast(self.front)
            if self.rear is None:
                    self.rear = self.front
```



Electronic copy available at: https://ssrn.com/abstract=3953786

```
        self.size += 1
    def queueRear(self):
        if self.rear is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.rear.getData()
    def queueFront(self):
        if self.front is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        return self.front.getData()
    def deQueue(self):
        if self.rear is None:
            print "Sorry, the queue is empty!"
            raise IndexError
        result = self.rear.getData()
        self.rear = self.rear.last
        self.size -= 1
        return result
    def size(self):
        return self.size
que = Queue()
que.enQueue("first")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("second")
print "Front: "+que.queueFront()
print "Rear: "+que.queueRear()
que.enQueue("third")
```



Electronic copy available at: https://ssrn.com/abstract=3953786

## Performance

Let $n$ be the number of elements in the queue, then

| | |
|---|---|
| Space Complexity (for $n$ EnQueue operations) | $O(n)$ |
| Time Complexity of EnQueue() | O(1) (Average) |
| Time Complexity of DeQueue() | O(1) |
| Time Complexity of IsEmptyQueue() | O(1) |
| Time Complexity of DeleteQueue() | O(1) |

## Comparison of Implementations

**Note**: Comparison is very similar to stack implementations and *Stacks* chapter.



Electronic copy available at: https://ssrn.com/abstract=3953786

# In Lab session

- You will play with the concepts and starts getting more and more familiar with how this works in real life
- This will be useful for your project

- Lab is done by Remy Belmonte


Electronic copy available at: https://ssrn.com/abstract=3953786