

# UE5 Fundamentals of Algorithms

## Lecture 3: Stacks and Queues

Ecole Centrale de Lyon, Bachelor of Science in Data Science for Responsible Business

Romain Vuillemot



**CENTRALE  
LYON**

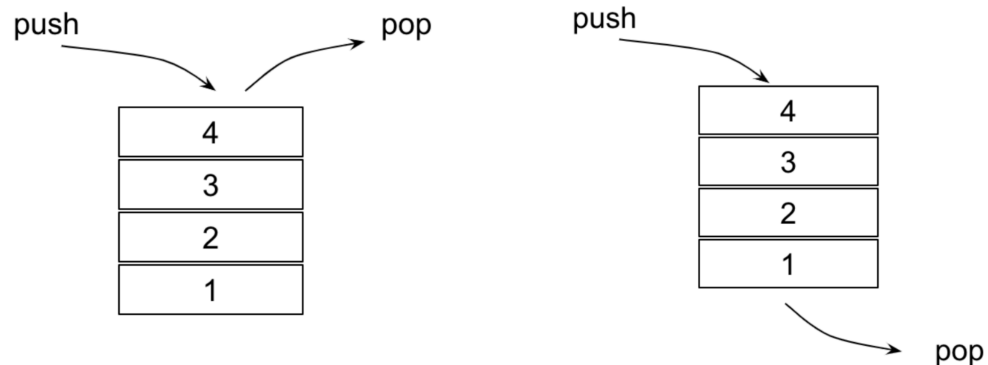


# Outline

- Definitions
- Stacks
- Queues
- Priority queues

# Defintions

*Stacks and queues allow the manipulation of values (or objects) sequentially. They have many operations, the main ones are: addition (push) and removal (pop), but with different order strategies:*



- **stacks** follow the Last-In, First-Out (LIFO) principle
- **queues** follows the First-In, First-Out (FIFO) principle

Note that stacks and queues define the operations and their results, but not their implementation.

# Operations

- `empty()` : Checks for emptiness.
- `full()` : Checks if it's full (if a maximum size was provided during creation).
- `get()` : Returns (and removes) an element.
- `push()` : Adds an element.
- `size()` : Returns the size of the list.
- `reverse()` : Reverses the order of elements.
- `peek()` : Returns an element (without removing it).

More operations can be included.

# Stacks

*A stack is an abstract data type that follows the Last-In, First-Out (LIFO) principle*

- It supports operations on a collection of elements.
- The element inserted last is at the *head*.
- Easily achievable with a simple list! See this [Python tutorial](#)

## Stacks (using lists)

```
In [3]: stack = [3, 4, 5]
stack.append(6) # push
stack.append(7) # push
```

```
In [4]: print(stack)
stack.pop() # get
print(stack)
stack.pop()
stack.pop()
print(stack)
print(stack[-1]) # peek
```

```
[3, 4, 5, 6, 7]
```

```
[3, 4, 5, 6]
```

```
[3, 4]
```

```
4
```

## Stacks (using modules)

<https://docs.python.org/3/library/queue.html>

```
In [5]: import queue
        pile = queue.LifoQueue()

        for i in [3, 4, 5]: pile.put(i)

        pile.put(6)
        pile.put(7)

        while not pile.empty():
            print(pile.get(), end=" ")
```

7 6 5 4 3

# Stacks (using OOP)

Internally, it be based on an `List` structure.



# Stacks (using OOP)

Internally, it be based on an `List` structure.

```
In [6]: class Stack():
        def __init__(self, values = []):
            self.__values = []
            for v in values:
                self.push(v)

        def push(self, v):
            self.__values.append(v)
            return v

        def get(self):
            v = self.__values.pop()
            return v

        def display(self):
            for v in self.__values:
                print(v)

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

## Stacks (using OOP)

```
In [7]: s = Stack()  
        for d in [3, 4, 5]:  
            s.push(d)  
            e = s.get()  
            print(e)
```

```
3  
4  
5
```

# Queues

*A queue is an abstract data type that follows the Last-In, First-Out (LIFO) principle*

- Similar to a Stack
- But the returned element is the first one inserted

# Queues (list)

```
In [8]: queue = [3, 4, 5]
        queue.append(6)
        queue.append(7) # push
```

```
In [9]: print(queue)
        queue.pop(0) # get
        print(queue)
        queue.pop(0)
        queue.pop(0)
        print(queue)
        print(queue[0]) # peek
```

```
[3, 4, 5, 6, 7]
```

```
[4, 5, 6, 7]
```

```
[6, 7]
```

```
6
```

# Queues (module)

```
In [11]: import queue

q = queue.Queue()

for i in [3, 4, 5]: q.put(i)

while not q.empty():
    print(q.get(), end=" ")
```

3 4 5

# Priority queues

*A **priority queue** is a queue (or stack or list) that returns an element based on the characteristics of a variable (priority).*

- For a quantitative variable, it's the minimum or maximum of the queue. For other types of variables (e.g., categories), any order relation is valid.
- Queues can exhibit the same behavior but have a different internal state: either constantly updated or updated after reads/writes.
- The internal state can be preserved with a sorting function, thus optimizing the complexity of the data structure.

# Priority queues (module)

```
In [12]: from heapq import heapify, heappush, heappop  
heap = [10, 8, 1, 2, 4, 9, 3, 4, 7]  
heapify(heap)
```

```
In [13]: heap
```

```
Out[13]: [1, 2, 3, 4, 4, 9, 10, 8, 7]
```

```
In [14]: heappop(heap)
```

```
Out[14]: 1
```

```
In [15]: heap
```

```
Out[15]: [2, 4, 3, 4, 7, 9, 10, 8]
```

```
In [16]: heappush(heap, 5)
```

```
In [17]: heap
```

```
Out[17]: [2, 4, 3, 4, 7, 9, 10, 8, 5]
```

# Priority queues (using OOP)

```
In [18]: class PriorityQueue(object):
    def __init__(self):
        self.__queue = []

    def __str__(self):
        return ' '.join([str(i) for i in self.__queue])

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

    def insert(self, data):
        self.__queue.append(data)

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

    def delete(self):
        min = 0
        for i in range(0, len(self.__queue)):
            if self.__queue[i][2] < self.__queue[min][2]:
                min = i
        item = self.__queue[min]
        del self.__queue[min]
        return item
```



```
In [19]: import queue

myQueue = queue.PriorityQueue()

# insert
myQueue.put(12)
myQueue.put(1)
myQueue.put(14)
myQueue.put(7)

# print
while not myQueue.empty():
    print(myQueue.get())
```

```
1
7
12
14
```

# Queue as linked list

In [21]:

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

class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0

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

    def push(self, data):
        new_node = Node(data)
        if self.rear is None:
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
        self.size += 1

    def get(self):
        if self.is_empty():
            raise IndexError("queue is empty")
```

```
temp = self.front
self.front = self.front.next

if self.front is None:
    self.rear = None

self.size -= 1
return temp.data

def peek(self):
    if self.is_empty():
        raise IndexError("empty queue")
    return self.front.data

def get_size(self):
    return self.size
```

In [22]:

```
queue = Queue()
queue.push(1)
queue.push(2)
queue.push(3)

print("first item peek:", queue.peek())
print("size:", queue.get_size())

print("get:", queue.get())
print("peek second item:", queue.peek())
print("size:", queue.get_size())
```

```
first item peek: 1
size: 3
get: 1
peek second item: 2
size: 2
```

In [ ]:

