UE5 Fundamentals of Algorithms

Lecture 7: Stacks and queues

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

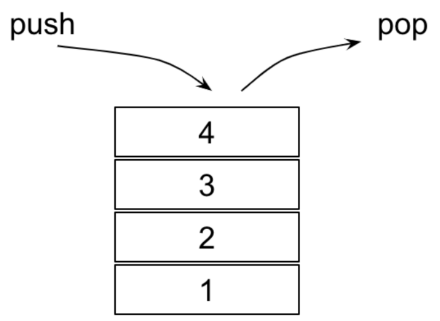
Romain Vuillemot

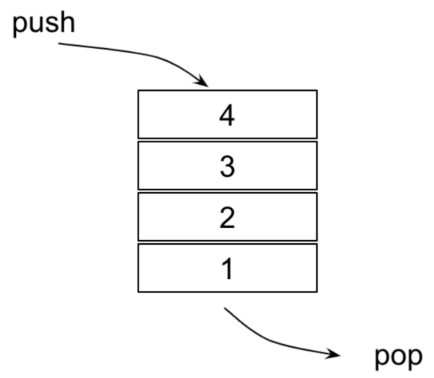
Outline

- 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).

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 [39]:
stack = [3, 4, 5]
stack.append(6) # push
stack.append(7)
```

```
In [40]:
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 [41]: import queue
         pile = queue.LifoQueue()

         for i in range(5): pile.put(i)

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

```
4 3 2 1 0
```

Stacks (using OOP)

Internally, will be based on an `Array` structure.

Stacks (using OOP)

Internally, will be based on an `Array` structure.

In [17]:
```
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 [42]:
data = ["A", "B", "C"]

s = Stack()
for d in data:
    s.push(d)
    e = s.pop()
    print(e)
```

```
A
B
C
```

Queues

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

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

Queues (list)

In [44]:
```
queue = [3, 4, 5]
queue.append(6)
queue.append(7) # push
```

In [45]:
```
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 [33]:
import queue

q = queue.Queue()

for i in range (5): q.put(i)

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

```
0 1 2 3 4
```

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

In [47]: heap

Out[47]: [1, 2, 3, 4, 4, 9, 10, 8, 7]

In [48]: heappop(heap)

Out[48]: 1

In [49]: heap

Out[49]: [2, 4, 3, 4, 7, 9, 10, 8]

In [50]: heappush(heap, 5)

In [51]: heap

Out[51]: [2, 4, 3, 4, 7, 9, 10, 8, 5]
```

Priority queues (using OOP)

```
In [56]:
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 [60]: import queue

         myQueue = queue.PriorityQueue()

         # Insert elements into the priority queue
         myQueue.put(12)
         myQueue.put(1)
         myQueue.put(14)
         myQueue.put(7)

         # Print the contents of the priority queue
         while not myQueue.empty():
             print(myQueue.get())
```

```
1
7
12
14
```

Improvements

- Handle empty lists

In [ ]:
```
def dequeue(self):
    if not self.is_empty():
        return self.items.pop(0)
    else:
        raise IndexError("Queue is empty")
```

In [ ]:
```
import queue
q = queue.Queue(5000)
# q.put([1,2,3,4])
q.put([2,3,4,5])
try:
    [x1, x2, x3, x4] = q.get(block=True)
except queue.Empty:
    print("Problème : aucun élément dans la file")
else:
    print([x1, x2, x3, x4])
```

In [ ]:
```
```