Fundamentos de Programação @ LEIC/LETI

Semana 12

Estruturas lineares

Pilhas e Filas. Pilhas. Operações básicas, axiomatização e representação de pilhas/filas. Exemplos

Alberto Abad, Tagus Park, IST, 2018

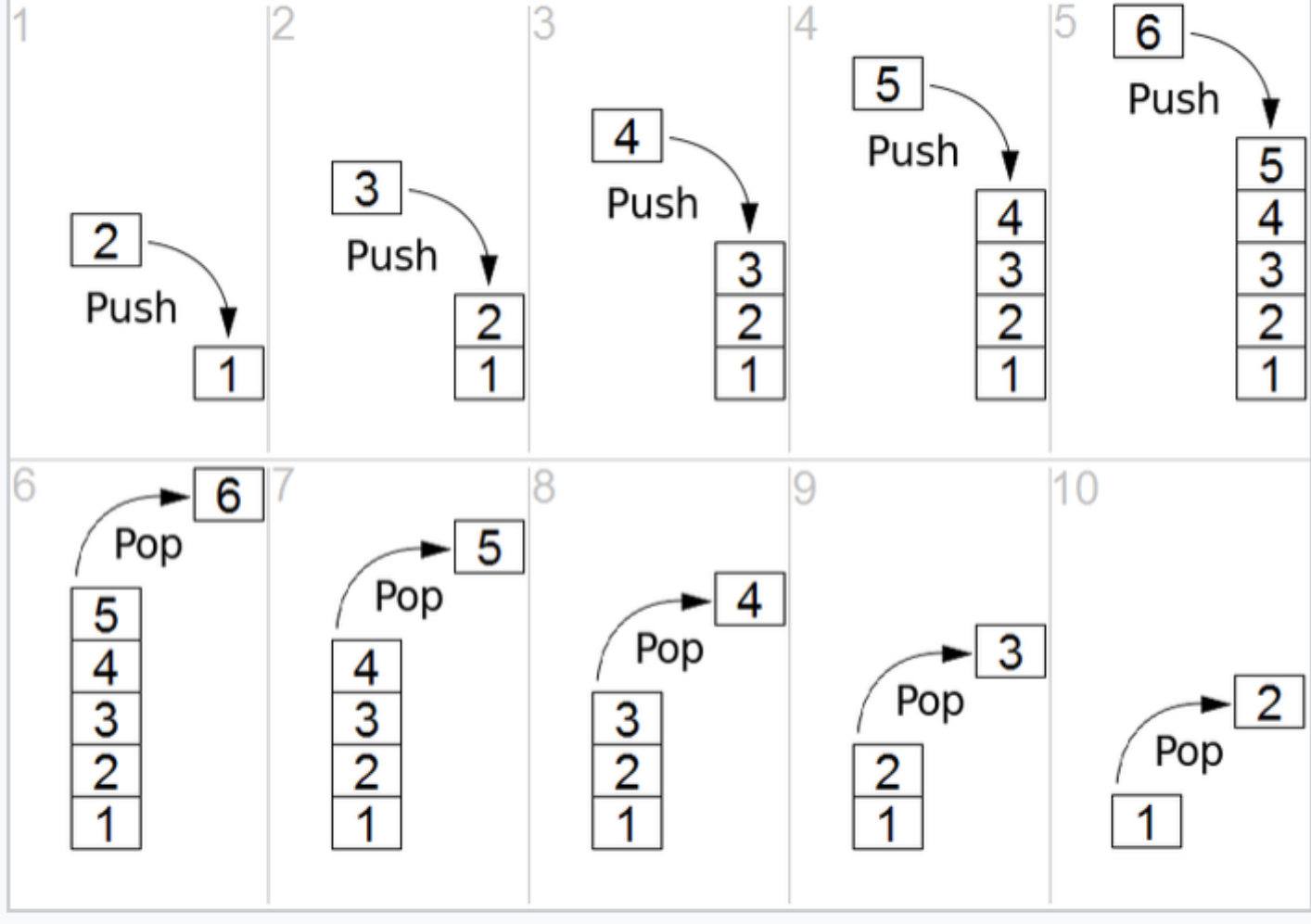
Estruturas lineares

Introdução

- Um dos tipos de estruturas de dados mais usados são aquilos que têm os seus elementos organizados em sequência, sendo por isso conhecidas por **estruturas de dados lineares**.
- Em Python, conhecemos as listas e os tuplos que nos servem como abstração para modelar vetores, sequencias, series e outras estrutaras do mundo real.
- Nesta aula, iremos ver dos tipos de estruturas lineares muito parecidas: as **pilhas** e as **filas**.
- As **pilhas** e **filas** são estruturas de dados constituídas por sequência de elementos, em que em geral não podemos aceder ou inspeccionar elementos arbitrários. Diferencias:
    - Nas **pilhas** os elementos podem ser removidos pela ordem inversa da inserção, i.e., LIFO (last in first out) e temos acesso ao elemento do topo.
    - Nas **filas** os elementos podem ser removidos pela ordem de inserção, i.e., FIFO (first in first out) e temos acesso ao primeiro elemento.

Pilha (Stack)

- Sequência do tipo **LIFO**:



Simple representation of a stack runtime with *push* and *pop* operations.

TAD Pilha (mutável) - Definição das operações básicas

Vejamos então como definir o TAD pilha e as suas operações básicas.

- Construtores:
  - pilha_nova: {} --> pilha, permite criar uma pilha vazia;
- Reconhecedores:
  - e_pilha: universal --> booleano, dada um qualquer valor passado por argumento decide se este é ou não uma pilha;
  - pilha_vazia: pilha --> booleano, dada uma pilha decide se a mesma está ou não vazia.
- Selectores:
  - pilha_topo: pilha --> elemento, dada uma pilha retorna o elemento que está no topo, se a pilha estiver vazia o comportamento é indefinido (aka top).
- Modificadores (caso em que a pilha é um tipo **mutável**):
  - pilha_retira: pilha --> pilha, dada uma pilha retira o elemento que está no topo e retorna a pilha, se a pilha estiver vazia o comportamento é indefinido (aka *pop*);
  - pilha_empurra: pilha x elemento --> pilha, dada uma pilha e um elemento coloca esse elemento, empurra para, o topo da pilha, e retorna a pilha (aka *push*);
- Testes:
  - pilha_igual: pilha x pilha --> booleano, dadas duas pilhas toma decide se as mesmas são ou não iguais.

TAD Pilha (mutável) - Axiomatização

```
e_pilha(pilha_nova())
e_pilha(pilha_empurra(p,e))              # para quaisquer pilha p e elemento e
e_pilha(pilha_retira(p))                 # para qualquer pilha p não vazia
pilha_vazia(pilha_nova())
not(pilha_vazia(pilha_empurra(p,e)))     # para quaisquer pilha p e elemento e
pilha_topo(pilha_empurra(p,e)) == e      # para quaisquer pilha p e elemento e

pilha_retira(pilha_empurra(p,e)) == p    # para quaisquer pilha p e elemento e
pilha_igual(pilha_nova(),pilha_nova())
pilha_igual(p, q)
    if not(pilha_vazia(p)) and not(pilha_vazia(q))
        and pilha_topo(p) == pilha_topo(q)
        and pilha_igual(pilha_retira(p), pilha_retira(q)) # para quaisquer pilhas
 p e q
```

TAD Pilha (mutável) - Exercícios

Exercício 1: Realização das operações básicas do TAD pilha, incluindo testes relativos à axiomatização recorrendo ao módulo `doctest` link (https://docs.python.org/3/library/doctest.html). Representar como listas, em que o topo da pilha é o primeiro elemento da lista.

Exercício 2: Comparar a eficiência das operações pilha_empurra e pilha_retira tendo em conta a representação interna anterior e a representação interna alternativa, também sobre listas, mas em que o topo da pilha é o último elemento da lista.

Exercício 3: Implementar o TAD pilha recorrendo à programação com objectos.

Exercício 4: Recorrendo ao TAD pilha, implemente um verificador de balancamento de parêntesis

Exercício 5: Recorrendo ao TAD pilha, implemente uma calculadora de pilha (https://en.wikipedia.org/wiki/Reverse_Polish_notation) (left-to-right processing).

TAD Pilha (mutável) - Exercício 1

Exercício 1: Realização das operações básicas do TAD pilha, incluindo testes relativos à axiomatização recorrendo ao módulo `doctest`. Representar como listas, em que o topo da pilha é o primeiro elemento da lista.

In [36]:
```
import doctest
"""
We can use the docstrings to show some samples of the module and
the doctest module to automatically validate them:

>>> e_pilha(pilha_nova())
True
>>> p = pilha_nova()
>>> e = 4
>>> e_pilha(pilha_empurra(p,e))              # para quaisquer pilha p e elemento e
True
>>> e_pilha(pilha_retira(p))                 # para qualquer pilha p não vazia
True
>>> pilha_vazia(pilha_nova())
True
>>> not(pilha_vazia(pilha_empurra(p,e)))     # para quaisquer pilha p e elemento e
True
>>> pilha_topo(pilha_empurra(p,e)) == e      # para quaisquer pilha p e elemento e
True
>>> pilha_retira(pilha_empurra(p,e)) == p    # para quaisquer pilha p e elemento e
True
>>> pilha_igual(pilha_nova(),pilha_nova())
True
"""

def pilha_nova():
    return []
def e_pilha(u):
    return isinstance(u, list)

def pilha_vazia(p):
    return e_pilha(p) and p == []

def pilha_topo(p):
    if pilha_vazia(p):
        raise ValueError("")
```

```
    return p[0]

def pilha_retira(p):
    if pilha_vazia(p):
        raise ValueError("")
    del p[0]
    return p

def pilha_empurra(p,e):
    p.insert(0,e)
    return p

def pilha_igual(p1, p2):
    return e_pilha(p1) and e_pilha(p2) and p1 == p2

doctest.testmod()
```

Out[36]: TestResults(failed=0, attempted=10)

TAD Pilha (mutável) - Exercício 2

Exercício 2: Comparar a eficiência das operações pilha_empurra e pilha_retira tendo em conta a representação interna anterior e a representação interna alternativa, também sobre listas, mas em que o topo da pilha é o último elemento da lista.

In [38]:
```
"""
We can use the docstrings to show some samples of the module:

>>> e_pilha(pilha_nova())
True

>>> p = pilha_nova()

>>> e_pilha(pilha_empurra(p,1)) # para quaisquer pilha p e elemento e
True

>>> e_pilha(pilha_retira(p)) # para qualquer pilha p não vazia
True

>>> pilha_vazia(pilha_nova())
True

>>> not(pilha_vazia(pilha_empurra(p,2))) # para quaisquer pilha p e elemento e
True

>>> pilha_topo(pilha_empurra(p,3)) == 3 # para quaisquer pilha p e elemento e
True

>>> pilha_retira(pilha_empurra(p,4)) == p # para quaisquer pilha p e elemento e
True

>>> pilha_igual(pilha_nova(),pilha_nova())
True
"""

def pilha_nova():
    return []

def e_pilha(u):
    return isinstance(u, list)
```

```
def pilha_vazia(p):
    return p == []

def pilha_topo(p):
    return p[-1]

def pilha_retira(p):
    del p[-1]
    return p

def pilha_empurra(p,e):
    p.append(e)
    return p

def pilha_igual(p1, p2):
    return p1 == p2

doctest.testmod()
```

Out[38]: TestResults(failed=0, attempted=9)

TAD Pilha (mutável) - Exercício 2

Exercício 2: Comparar a eficiência das operações pilha_empurra e pilha_retira tendo em conta a representação interna anterior e a representação interna alternativa, também sobre listas, mas em que o topo da pilha é o último elemento da lista.

In [37]:
```
# Rerun the original stack cell
def teste_func(p):
    for n in range(10000):
        if n%2 != 0:
            p = pilha_retira(p)
        else:
            p = pilha_empurra(p, n)


p = pilha_nova()
for n in range(1000):
    p = pilha_empurra(p, n)
%timeit teste_func(p)
```

```
5.11 ms ± 155 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [39]:

```
# Rerun the modified stack cell
def teste_func(p):
    for n in range(10000):
        if n%2 != 0:
            p = pilha_retira(p)
        else:
            p = pilha_empurra(p, n)


p = pilha_nova()
for n in range(1000):
    p = pilha_empurra(p, n)
%timeit teste_func(p)
```

```
1.62 ms ± 65 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

TAD Pilha (mutável) - Exercício 3

Exercício 3: Implementar o TAD pilha recorrendo à programação com objectos.

In [2]:

```
class pilha:

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

    def empty(self):
        return self.p == []

    def top(self):
        if self.empty():
            raise ValueError("pilha: top: pilha vazia")
        return self.p[-1]

    def pop(self):
        if self.empty():
            raise ValueError("pilha: pop: pilha vazia")
        self.p = self.p[:-1]
        return self

    def push(self, e):
        self.p.append(e)
        return self

    def __repr__(self):
        return "".join(" " + str(e) + " \n" for e in self.p[-1::-1]) + "---"

p = pilha()
p.push(2).push(4)
print(p)
print(p.top())
print(p)
p.pop()
p
```

```
 4
 2
---
4
 2
---
```

TAD Pilha (mutável) - Exercício 4

Exercício 4: Recorrendo ao TAD pilha, implemente um verificador de balancamento de parêntesis

```
>>> balanceado('[([{}()])()]')
True
>>> balanceado('[([{}()])()')
False
>>> balanceado('[([{}()])()}')
False
```

In [51]:
```
def balanceado(instring):
    s = pilha()
    for c in instring:
        if c in ('(','{','['):
            s.push(c)
        elif c in ')}]':
            if s.empty():
                return False
            top = s.top()
            if top == '(' and c == ')' or \
                top == '[' and c == ']' or \
                top == '{' and c == '}':
                s.pop()
            else:
                return False
    return s.empty()









balanceado('(')
```

Out[51]:
```
False
```

TAD Pilha (mutável) - Exercício 5

Exercício 5: Recorrendo ao TAD pilha, implemente uma calculadora de pilha (https://en.wikipedia.org/wiki/Reverse_Polish_notation) (left-to-right processing).

- infix: ((15 ÷ (7 − (1 + 1))) × 3) − (2 + (1 + 1))
- postfix: 15 7 1 1 + − ÷ 3 × 2 1 1 + + −

Algoritmo

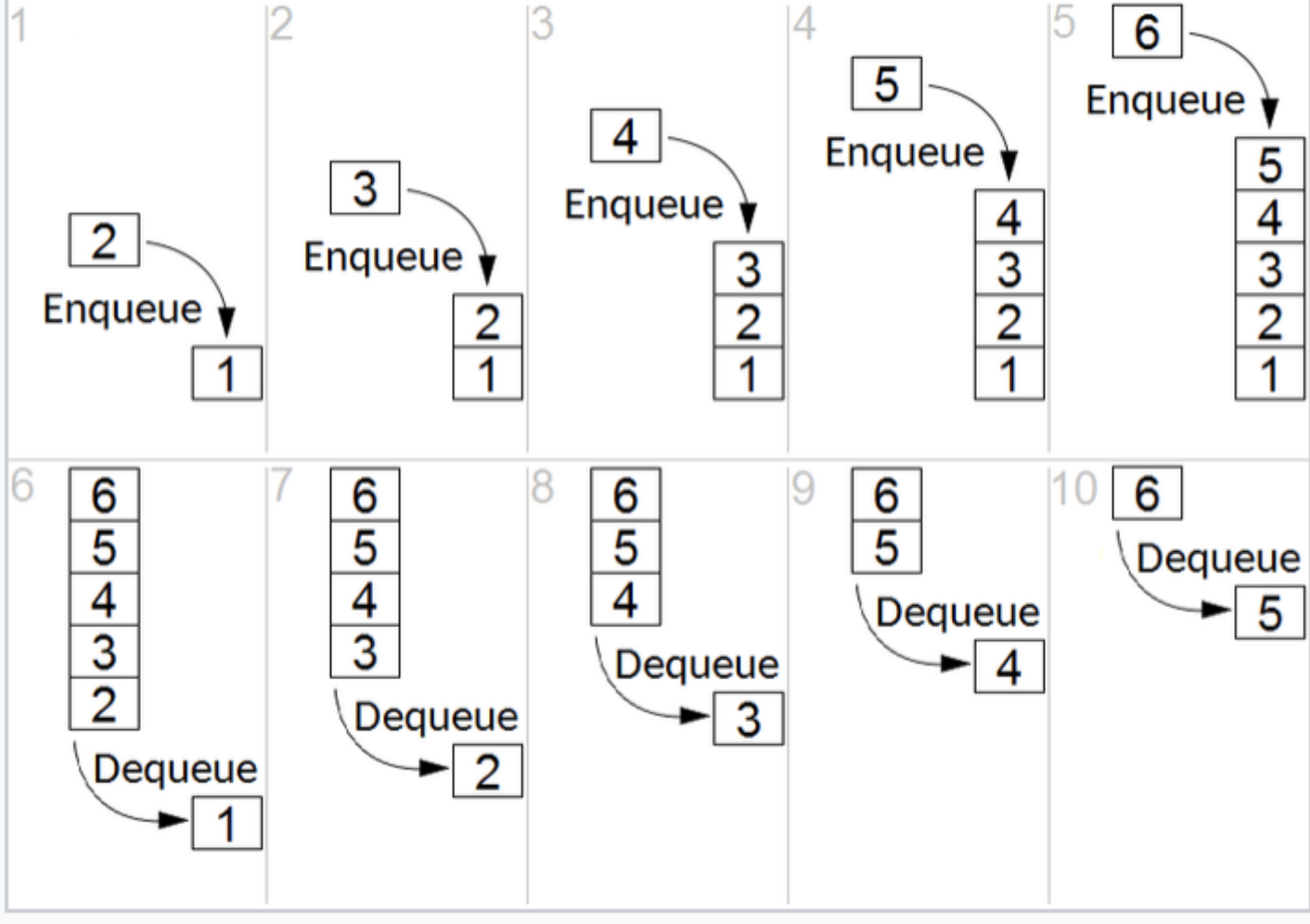
```
for each token in the postfix expression:
  if token is an operator:
    operand_2 ← pop from the stack
    operand_1 ← pop from the stack
    result ← evaluate token with operand_1 and operand_2
    push result back onto the stack
  else if token is an operand:
    push token onto the stack
result ← pop from the stack
```

In [3]:
```
def postfix_calculator(postfix):
    """
    >>> postfix_calculator('15 7 1 1 + - / 3 * 2 1 1 + + -')
    5
    """
    s = pilha()
    for token in postfix.split():
        if token in '+-/*':
            op2 = s.top()
            s.pop()
            op1 = s.top()
            s.pop()
            res = eval(op1 + token + op2)
            s.push(str(res))
        else:
            s.push(token)

    return eval(s.top())


postfix_calculator('15 7 1 1 + - / 3 * 2 1 1 + + -')
```

Out[3]:
```
5.0
```

Fila (Queue)

- Sequência do tipo **FIFO**:



Representation of a FIFO (queue) with *enqueue* and *dequeue* operations.

TAD Fila (mutável) - Definiçaõ das operações básicas

- Construtores:
  - fila_nova: {} --> fila, permite criar uma fila vazia, i.e., sem elementos;
- Reconhecedores:
  - e_fila: universal --> booleano, dada um qualquer valor passado por argumento decide se este é ou não uma fila;
  - fila_vazia: fila --> booleano, dada uma fila decide se a mesma está ou não vazia.
- Selectores:
  - fila_inicio: fila --> elemento, dada uma fila retorna o elemento que está no início, se a fila estiver vazia o comportamento é indefinido (aka first);
  - fila_comprimento: fila --> int, dada uma fila retorna o número de elementos na mesma.
- Modificadores (caso em que a fila é um tipo mutável):
  - fila_retira: fila --> fila, dada uma fila retira o elemento que está no início e retorna a fila, se a fila estiver vazia o comportamento é indefinido (aka dequeue);
  - fila_coloca: fila x elemento --> fila, dada uma fila e um elemento coloca esse elemento no fim da fila, e retorna a fila (aka queue);
- Testes:
  - fila_igual: fila x fila --> booleano, dadas duas filas toma decide se as mesmas são ou não iguais.
- Transformadores:
  - fila_para_lista: fila --> lista, dada uma fila retorna uma lista com os elementos na fila, pela mesma ordem.

TAD Fila (mutável) - Axiomatização

```
e_fila(fila_nova())
e_fila(fila_coloca(f,e))              # para quaisquer fila f e elemento e
e_fila(fila_retira(f))                # para qualquer fila f não vazia
fila_vazia(fila_nova())
not(fila_vazia(fila_coloca(f,e)))     # para quaisquer fila f e elemento e
fila_inicio(fila_coloca(f,e)) == e    # para quaisquer fila f vazia e elemento e
fila_inicio(fila_coloca(f,e)) == fila_inicio(f) # para quaisquer fila f nao vazia
 e elemento e

fila_retira(fila_coloca(f,e)) == fila_nova() # para quaisquer fila f vazia e elem
ento e
fila_retira(fila_coloca(f,e)) == fila_coloca(fila_retira(f), e) # para quaisquer
 fila f nao vazia e elemento e
fila_comprimento(fila_nova()) == 0
fila_comprimento(fila_coloca(f,e)) == 1 + fila_comprimento(f) # para quaisquer fi
la f e elemento e
fila_igual(fila_nova(),fila_nova())
fila_igual(f, g)
    if not(fila_vazia(f)) and not(fila_vazia(g))\
        and fila_topo(f) == fila_inicio(g)\
        and fila_igual(fila_retira(f), fila_retira(g)) # para quaisquer filas f e
 g
```

TAD Fila (mutável) - Exercícios

Exercício 1: Realização das operações básicas do TAD fila, incluindo testes relativos à axiomatização recorrendo ao módulo doctest. Podemos escolher para representação interna uma lista, em que o elemento no início da fila está na primeira posição da lista e o elemento no final da fila está na última posição.

Exercício 2: Assumindo que sabemos o número máximo de elementos a guardar na fila, parâmetro dado ao construtor, propor uma realização do TAD fila que recorra a um *array circular* como representação interna.

Exercício 3: Implementar o TAD fila recorrendo à programação com objectos.

Exercício 4: Proponha um TAD que suporte simultaneamente o comportamento de uma fila e o comportamento de uma pilha.

Exercício 5 Recorrendo ao TAD anterior, implemente um detetor de palíndromos.

Estruturas lineares

TAD Fila (mutável) - Exercício 1

Exercício 1: Realização das operações básicas do TAD fila, incluindo testes relativos à axiomatização recorrendo ao módulo doctest. Podemos escolher para representação interna uma lista, em que o elemento no início da fila está na primeira posição da lista e o elemento no final da fila está na última posição.

In [ ]:

```
import doctest

"""
We can use the docstrings to show some samples of the module:

>>> e_fila(fila_nova())
True

"""

def fila_nova():
    return []

def e_fila(u):
    return isinstance(u, list)

def fila_vazia(p):
    return p == []

def fila_inicio(p):
    return p[0]

def fila_cumprimento(p):
    return len(p)

def fila_retira(p):
    if fila_cumprimento(p)== 0:
        raise ValueError("fila: fila_retira: fila vazia")
    del p[0]
    return p

def fila_coloca(p,e):
    p.append(e)
    return p

def fila_igual(p1, p2):
```

```
    return p1 == p2


def fila_para_lista(p):
    return p
```

TAD Fila (mutável) - Exercício2

Exercício 2: Assumindo que sabemos o número máximo de elementos a guardar na fila, parâmetro dado ao construtor, proponha uma realização do TAD fila que recorra a um array circular como representação interna. Quais as vantagens em termos de eficiência em relação à representação interna considerada anteriormente?

In [ ]:

```
import doctest

"""
We can use the docstrings to show some samples of the module:

>>> e_fila(fila_nova(50))
True

"""

def fila_nova(size):
    return [0, 0, [0]*size]

def e_fila(u):
    return isinstance(u, list) and len(u) == 3 and \
            isinstance(u[0], int) and \
            isinstance(u[1], int) and \
            isinstance(u[2], list) and \
            0 <= u[0] < len(u) and \
            0 <= u[1] < len(u)

def fila_vazia(p):
    return p[0] == p[1]

def fila_inicio(p):
    return p[2][p[0]]

def fila_cumprimento(p):
    l = p[1] - p[0]
    return l if l >= 0 else l + len(p[2])

def fila_retira(p):
    if fila_cumprimento(p)== 0:
        raise ValueError("fila: fila_retira: fila vazia")
    p[0] = (p[0] +1)%len(p[2])
    return p
```

```
def fila_coloca(p,e):
    if fila_cumprimento(p)== len(p[2]) - 1:
        raise ValueError("fila: fila_coloca: fila cheia")
    p[2][p[1]] = e
    p[1] = (p[1]+1)%len(p[2])
    return p

def fila_igual(p1, p2):
    if fila_vazia(p1):
        return fila_vazia(p2)
    elif fila_vazia(p2):
        return False
    else:
        return fila_inicio(p1) == fila_inicio(p2) and  fila_igual(fila_retira(p1),
 fila_retira(p2))

def fila_para_lista(p):
    if p[0] < p[1]:
        return p[2][p[0]:p[1]]
    else:
        return p[2][p[0]:] + p[2][:p[1]]


def fila_para_str(p):
    return "<< " + " ".join(str(n) for n in fila_para_lista(p)) + " <<"


q = fila_nova(5)
fila_coloca(q,1)
fila_coloca(q,2)
fila_coloca(q,3)
fila_coloca(q,4)
print(fila_para_str(q))
fila_retira(q)
fila_retira(q)
fila_coloca(q,5)
```

```
fila_coloca(q,6)
print(fila_para_str(q))
```

In [ ]:

```
class cqueu:
    def __init__(self, size):
        self.nextread = 0
        self.nextwrite = 0
        self.maxsize = size
        self.values = [0]*self.maxsize

    def empty(self):
        return self.nextread == self.nextwrite

    def first(self):
        if self.empty():
            raise ValueErro("first: empty queu")
        return self.values[self.nextread]

    def len(self):
        l = self.nextwrite - self.nextread
        return l if l >= 0 else l + self.maxsize

    def dequeu(self):
        if self.empty():
            raise ValueErro("dequeu: empty queu")
        self.nextread = (self.nextread+1)%self.maxsize
        return self

    def queu(self, e):
        if self.len() == self.maxsize - 1:
            raise ValueErro("queu: full queu")
        self.values[self.nextwrite] = e
        self.nextwrite = (self.nextwrite+1)%self.maxsize
        return self

    def __eq__(self, other):
        if self.empty():
            return other.empty()
        elif other.empty():
```

```
            return False
        else:
            return self.first() == other.first() and self.dequeu() == other.dequeu
()

    def to_list(self):
        if self.nextread <= self.nextwrite:
            return self.values[self.nextread:self.nextwrite]
        else:
            return self.values[self.nextread:] + self.values[:self.nextwrite]

    def __repr__(self):
        return "<< " + " ".join(str(n) for n in self.to_list()) + " <<"

q = cqueu(5)
q.queu(1).queu(2).queu(3).queu(4)
print(q)
q.dequeu()
q.dequeu()
q.queu(5).queu(6)
print(q)
```

TAD Fila (mutável) - Exercício 3

Exercício 3: Implementar o TAD fila recorrendo à programação com objectos.

TAD Fila (mutável) - Exercício 4

Exercício 4: Proponha um TAD que suporte simultaneamente o comportamento de uma fila e o comportamento de uma pilha:

- Os novos elementos são acrescentados no fim da lista/topo da pilha
- Elementos do ínicio e do topo podem ser inspecionados/retirados

In [ ]:
```
class squeu:
    def __init__(self):
        self.v = []

    def add(self, v):
        self.v.append(v)

    def first(self):
        return self.v[0]

    def top(self):
        return self.v[-1]

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

    def empty(self):
        return self.v == []

    def pop(self):
```

```
        del self.v[-1]
        return self

    def dequeu(self):
        del self.v[0]
        return self
```

TAD Fila (mutável) - Exercício 5

Exercício 5 Recorrendo ao TAD anterior, implemente um detetor de palíndromos:

```
>>> palindromo('amor a roma')
True
>>> palindromo('sopapos')
True
>>> palindromo('abcdedcba')
True
```

In [ ]:
```
import doctest

def palindromo(string):
    """
    >>> palindromo('amor a roma')
    True
    >>> palindromo('sopapos')
    True
    >>> palindromo('abcdedcba')
    True
    """
```

In [ ]:

```
import doctest

def palindromo(string):
    """
    >>> palindromo('amor a roma')
    True
    >>> palindromo('sopapos')
    True
    >>> palindromo('abcdedcba')
    True
    """
    sq = squeu()
    for c in string:
        sq.add(c)

    while sq.size() > 1:
        if sq.first() != sq.top():
```

```
                return False
            sq.pop().dequeu()

    return True

doctest.testmod()
```

Sobre pilhas e filas em Python

- Existem implementações no módulos `queue` e `collections`:

In [ ]:
```
from queue import Queue
q = Queue(maxsize=0)
q.put(10)
print(q.empty())
```

In [ ]:
```
from collections import deque
s = deque()
s.append(10)
print(s)
s.append(2)
print(s)
print(s.pop())
print(s)
```

Representação gráfica

Existem várias bibliotecas disponíveis para desenvolver GUIs. Entre as mais comuns estão:

- pygame (http://pygame.org/ (http://pygame.org/))
  - Um tutorial em português: https://old.gustavobarbieri.com.br/jogos/jogo/doc/ (https://old.gustavobarbieri.com.br/jogos/jogo/doc/)
- Módulo `graphics` de John Zellee (http://mcsp.wartburg.edu/zelle/python/graphics.py (http://mcsp.wartburg.edu/zelle/python/graphics.py) and http://mcsp.wartburg.edu/zelle/python/graphics/graphics/index.html

Próxima semana:

- Discussão do projeto, incluindo:
    - Apresentação de soluções (voluntarios?)
    - Apresentação versão com GUI (voluntarios?)
- Apresentar alguns tópicos mais avançados ou que ficaram incompletos (voluntarios!?):