Alocação sequencial - filas

Filas

A estrutura de dados Fila também é bastante intuitiva. A analogia é com uma fila de pessoas aguardando para serem atendidas no guichê de um banco, ou aguardando o ônibus. Se houver respeito, todos obedecem a fila. Quando chega um novo elemento, este se posiciona no final da fila e quando libera um lugar quem vai é quem está no início da fila.

Assim, uma fila é caracterizada pela sequência:
O primeiro a entrar é o primeiro a sair.
Ou
O último a entrar é o último a sair.

As expressões em inglês que definem esta estrutura:
FIFO – First In First Out
LILO – Last In Last Out

Considere o exemplo abaixo que mostra a evolução de uma fila. Uma letra significa "adicione um elemento na fila" e um ponto significa "retire um elemento da fila".

Operação	Retirado	Fila
F		F
I		FI
L		FIL
.	F	IL
A		ILA
.	I	LA
E		LAE
X		LAEX
.	L	AEX
.	A	EX
M		EXM
P		EXMP
.	E	XMP
.	X	MP
.	M	P
.	P	
.		**<erro – vazia>**

Implementação de fila como uma lista – um ADT

A estrutura Fila, pode ser implementada em uma lista, como um ADT com os seguintes métodos:

Queue() – Construtor da classe (init()). Cria e retorna uma nova fila vazia
enqueue(item) – Adiciona novo item no final da fila.
dequeue() – Remove item do início da fila. Retorna esse item.
first() – Retorna o valor do item inicial da fila mas não o remove.
last() – Retorna o valor do item final da fila mas não o remove.
isEmpty() – Retorna True se a fila está vazia.
size() – Retorna o número de elementos da fila.

```
class Queue:
    # inicia a fila - lista vazia
    def __init__(self):
        self.items = []
    # devolve True se fila está vazia e False senão
    def is_empty(self):
        return self.items == []
    # enfila novo elemento no final da fila
    def enqueue(self,item):
        self.items.append(item)
    # retira da fila o primeiro elemento
    def dequeue(self):
        return self.items.pop(0)
    # devolve o valor do elemento do primeiro mas não remove
    def first(self):
        return self.items[0]
    # devolve o valor do elemento do último mas não remove
    def last(self):
        return self.items[-1]
    def size(self):
        return len(self.items)

# testes da classe Queue
f = Queue()
# enfila 1, 3 e 5 nesta ordem
f.enqueue(1)
f.enqueue(3)
f.enqueue(5)
# mostra início e fim da fila
print("inicio:", f.first(), "fim:", f.last())
# remove e mostra os dois primeiros 1 e 3
print("desenfila:", f.dequeue())
print("desenfila:", f.dequeue())
# mostra início e fim da fila
print("inicio:", f.first(), "fim:", f.last())
```

Saída:

```
inicio: 1 fim: 5
```

```
desenfila: 1
desenfila: 3
inicio: 5 fim: 5
```

Rigorosamente nas funções dequeue, first e last acima, temos que tratar o caso de fila vazia sinalizando exceção ou devolvendo None:

```
        if self.is_empty():
           raise Empty("Fila Vazia")
```

Ou:

```
        if self.is_empty(): return None
```

Outra forma de implementação de fila como uma lista

O uso das funções append() e pop() na implementação acima, embora bastante versátil, não é muito eficiente. Isso porque essas funções precisam fazer alocação dinâmica de memória para estender ou diminuir a tamanho da lista.

Outra solução possível seria:

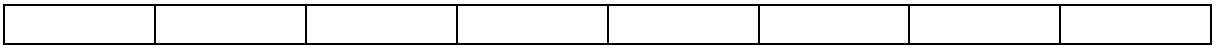
- Fixar um tamanho máximo para a fila
- Controlar o início e fim com índices ou apontadores para elementos da fila
- Tornar a lista circular para reaproveitar os elementos que forem sendo esvaziados.

O uso de uma fila restrita a um tamanho máximo e necessário em muitas aplicações para compatibilizar o crescimento da fila com a capacidade de processamento dos seus elementos.

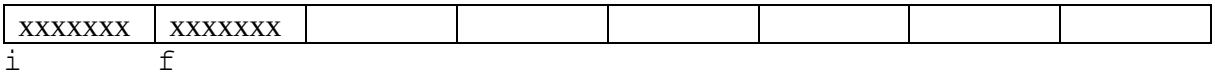
Vamos usar então 2 apontadores. Um indica o início da fila (primeiro elemento) e o outro o fim da fila (último elemento).

Veja abaixo uma fila implementada numa lista de tamanho máximo 8. (`v[0],v[1],....,v[7]`)

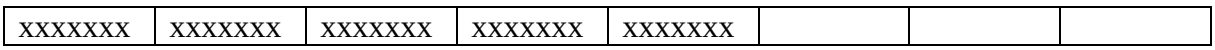
Fila vazia: `i=-1; f=-1;`



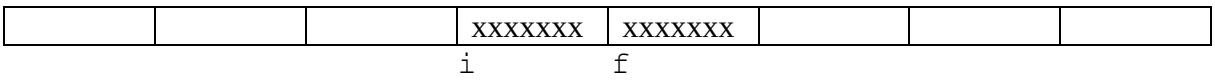
Inserir 2 elementos: `i=0; f=1;`



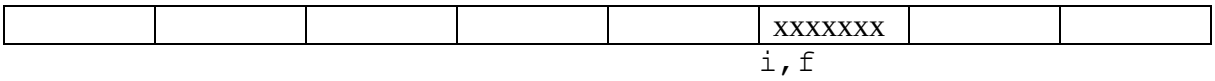
Inserir mais 3 elementos: `i=0; f=4;`



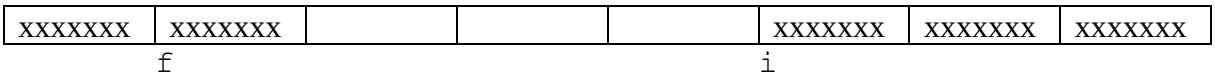
Remover 3 elementos: `i=3; f=4;`



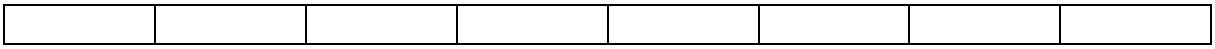
Inserir 1 elemento e remover 2: `i=5; f=5;`



Inserir 4 elementos: neste caso temos um problema, pois não cabem mais elementos no vetor e temos elementos vazios no início. A fila é circular, isto é, o último elemento do é seguido do primeiro. Ou seja, basta somar um (módulo o tamanho máximo). Neste exemplo teríamos: `i=5; f=1;`



Remover 5 elementos: neste caso também temos um problema, pois a fila ficaria vazia e qual seria o valor de `i` e `f`? Se deixarmos `i=f=2`, por exemplo, não conseguiremos distinguir fila vazia de fila com um só elemento. É melhor neste caso forçar `i=f=-1`.



Operações sobre uma fila F – um novo ADT

Vamos repetir as operações:

Adicionar um elemento x – enqueue(x)
Retirar um elemento – dequeue()

Consultar o valor do elemento do primeiro elemento – first()
Retornar True se a fila está vazia – is_empty()
Retornar o tamanho atual da fila – len()

Em especial a operação dequeue() e first() devem avisar se a fila está vazia e enqueue(x) se a fila está completamente cheia. Vamos manter também a quantidade corrente de elementos da fila.

Nessa implementação vamos precisar manter indicadores de inicio e fim, da quantidade atual e do número máximo de elementos. São vários atributos de um mesmo objeto fila.

```
class FilaLista:
    # MAXF define o tamanho máximo da fila.
    MAXF = 3 # teste com fila de 3 elementos
    def __init__(self):
        self._fila = [None] * FilaLista.MAXF
        self._size = 0
```

```
        self._front = -1
        self._end = -1

    # Adiciona um elemento no final da fila
    def enqueue(self, x):
        # verifica se a fila está cheia
        if self._size == FilaLista.MAXF: return False
        self._end = (self._end + 1) % FilaLista.MAXF
        self._fila[self._end] = x
        # se é o primeiro elemento altera também o inicio
        if self.is_empty(): self._front = 0
        self._size = self._size + 1
        return True

    # Remove um elemento do fim da fila
    def dequeue(self):
        if self.is_empty(): return None
        x = self._fila[self._front]
        self._fila[self._front] = None
        self._size = self._size - 1
        # verifica se a fila vai ficar vazia
        # neste caso, ambos os ponteiros ficam -1
        if self._front == self._end:
            self._front = self._end = -1
        else:
            self._front = (self._front + 1) % FilaLista.MAXF
        return x

    # Retorna True se a fila está vazia
    def is_empty(self):
        return self._size == 0

    # Devolve o primeiro elemento da fila sem removê-lo
    def first(self):
        if self.is_empty: return None
        return self._fila[self._front]

    # Retorna o tamanho atual da fila
    def __len__(self):
        return self._size

    # Mostra o conteúdo da fila - apenas para teste
    def print(self):
        print(self._fila)
```

Os comandos abaixo mostram um teste da classe:

```
# teste das funções - fila de 3 elementos
F = FilaLista()

# adicionar 4 elementos
for k in [32, 45, 12, 27]:
    if F.enqueue(k): print(k, " adicionado com sucesso")
    else: print(k, "não foi adicionado - fila cheia")
    F.print()

# retirar 2 elementos
print()
print(F.dequeue(), "foi removido")
F.print()
print(F.dequeue(), "foi removido")
F.print()

# primeiro elemento e tamanho
print()
print(F.first(), "é o primeiro elemento da fila")
print(len(F), "é o tamanho atual da fila")

# adicionar 1 elemento
print()
if F.enqueue(-1): print(-1, " adicionado com sucesso")
else: print(-1, "não foi adicionado - fila cheia")
F.print()

# retirar 3 elementos
print()
k = F.dequeue()
if k is not None: print(k, "foi removido")
else: print("fila vazia - nada foi removido")
F.print()

k = F.dequeue()
if k is not None: print(k, "foi removido")
else: print("fila vazia - nada foi removido")
F.print()

k = F.dequeue()
if k is not None: print(k, "foi removido")
else: print("fila vazia - nada foi removido")
F.print()
```

Será impresso:

```
32  adicionado com sucesso
```

```
[32, None, None]
45  adicionado com sucesso
[32, 45, None]
12  adicionado com sucesso
[32, 45, 12]
27 não foi adicionado - fila cheia
[32, 45, 12]

32 foi removido
[None, 45, 12]
45 foi removido
[None, None, 12]

12 é o primeiro elemento da fila
1 é o tamanho atual da fila

-1  adicionado com sucesso
[-1, None, 12]

12 foi removido
[-1, None, None]
-1 foi removido
[None, None, None]
fila vazia - nada foi removido
[None, None, None]
```

Uma simulação das funções de fila

Algoritmos sequenciais não precisam de filas. Por isso, os melhores exemplos de filas ocorrem quando há processos assíncronos (com velocidades diferentes) compartilhando dados. Por exemplo, um processo que cria mensagens e outro que processa essas mensagens, em velocidades diferentes. Há a necessidade de se manter uma fila com as mensagens para processá-las na mesma ordem em que foram criadas.

Vamos simular o comportamento de uma fila usando a classe FilaLista anterior. Para isso vamos jogar uma moeda, se der cara, adicionamos um elemento na fila e se der coroa removemos um elemento. Claro que usaremos o módulo random para fazer isso.

Vamos usar uma fila com 5 elementos. Para tanto basta modificar a atribuição inicial para:

```
MAXF = 5 # teste com fila de 5 elementos
```

Vamos melhorar a função print() para mostrar a fila e os ponteiros:

```
    # Mostra o conteúdo da fila
    def print(self):
        # mostra a fila
        print("* * * status da fila * * *")
        for k in range(FilaLista.MAXF):
```

```
            if self._fila[k] is None: print('Vazio', end = ' | ')
            else: print("%05d" %self._fila[k], end = ' | ')
        print()
        # mostra os índices
        for k in range(FilaLista.MAXF): print('%-8d' %k, end = '')
        print()
        # mostra o início e o fim
        print("i =", self._front, " f =", self._end)
        print()
```

Supondo agora que a classe FilaLista está dentro do arquivo ClassefilaLista.py, o programa abaixo realiza essa simulação. A cada lance (adição ou remoção) damos um enter para continuar.

```
from ClasseFilaLista import FilaLista
from random import randrange

# cria uma fila
NF = FilaLista()
# operações sobre a fila
while True:
    k = randrange(2)
    if k == 0:
        # adiciona elemento
        x = randrange(100000)
        if NF.enqueue(x):
            print(x, "adicionado a fila")
        else:
            print(x, "não foi adicionado - fila cheia")
    else:
        # remove elemento
        x = NF.dequeue()
        if x is not None:
            print(x, "- removido com sucesso")
        else:
            print("nada foi removido - fila vazia")
    # mostra a fila
    NF.print()
    # espera um tempo
    entrada = input("")
```

Um exemplo de execução desse programa abaixo:

```
nada foi removido - fila vazia
* * * status da fila * * *
Vazio | Vazio | Vazio | Vazio | Vazio |
0       1       2       3       4
i = -1  f = -1
```

```
42785 adicionado a fila
* * * status da fila * * *
42785 | Vazio | Vazio | Vazio | Vazio |
0       1       2       3       4
i = 0  f = 0


42785 - removido com sucesso
* * * status da fila * * *
Vazio | Vazio | Vazio | Vazio | Vazio |
0       1       2       3       4
i = -1  f = -1


37988 adicionado a fila
* * * status da fila * * *
37988 | Vazio | Vazio | Vazio | Vazio |
0       1       2       3       4
i = 0  f = 0


37988 - removido com sucesso
* * * status da fila * * *
Vazio | Vazio | Vazio | Vazio | Vazio |
0       1       2       3       4
i = -1  f = -1


41500 adicionado a fila
* * * status da fila * * *
41500 | Vazio | Vazio | Vazio | Vazio |
0       1       2       3       4
i = 0  f = 0


51891 adicionado a fila
* * * status da fila * * *
41500 | 51891 | Vazio | Vazio | Vazio |
0       1       2       3       4
i = 0  f = 1


41500 - removido com sucesso
* * * status da fila * * *
Vazio | 51891 | Vazio | Vazio | Vazio |
0       1       2       3       4
i = 1  f = 1
```

```
...
...
...
```

P6.1) Rodar o programa acima como vários valores de MAXF e verifique o comportamento da fila.

Outras formas de implementação

Certamente há outras maneiras de implementar as funções que mexem com filas. Não é necessário manter os indicadores de inicio e fim e também o tamanho da fila. Os indicadores inicio e fim ou inicio e tamanho seriam suficientes.

Não é necessário também que a implementação seja um ADT (Tipo de Dado Abstrato). Podemos implementar por funções independentes, usando mais parâmetros nestas funções ou usando variáreis globais. Considere a versão abaixo:

```
# Variáveis globais
iniciof = -1   # inicio da fila
fimf = -1      # fim da fila
sizeq = 0      # número de elementos na fila
MAXF = 10      # tamanho máximo da fila
Fila = MAXF * [None]     # a fila

# Adiciona um elemento no final da fila
def enqueue(x):
    global iniciof, fimf, sizeq, MAXF, Fila
    if sizeq == MAXF: return False
    fimf = (fimf + 1) % MAXF
    Fila[fimf] = x
    # se esse é o primeiro elemento altera também o iniciof
    if is_empty(): iniciof = 0
    sizeq = sizeq + 1
    return True

# Remove um elemento do fim da fila
def dequeue():
    ...

# Devolve o valor do primeiro elemento da fila sem removê-lo
def first():
    ...

# Retorna True se a fila está vazia
def is_empty():
    ...

# Retorna o tamanho atual da fila
```

```
def len():
    ...

# Mostra a fila
def print():
    ...

# variáveis globais da fila
print("inicio")
iniciof = fimf = -1
sizeq = 0
MAXF = 5
Fila = [None] * MAXF

# uso da fila
...
```

P6.2) Complete as funções acima da mesma forma que foi feito o enqueue()

Fila cheia e fila vazia

A situação de fila cheia ou fila vazia é um erro de programação. Em vez de retornar um aviso dessa situação a função poderia sinalizar uma exceção. Ficariam então:

```
if self.is_empty(): raise EmptyQueue("Fila vazia")

if self._size == FilaLista.MAXF: raise FullQueue("Fila cheia")
```

Para tanto será necessário a definição de duas novas sub-classes de exception.

```
class EmptyQueue(Exception):
    ''' Nova excessão - sub classe de Exception'''
    pass

class FullQueue(Exception):
    ''' Nova excessão - sub classe de Exception'''
    pass
```

Redimensionando o tamanho da fila

Outra forma de lidar com o problema de estouro do tamanho da fila, quando é necessário adicionar mais elementos, é redimensioná-la quando necessário. Por exemplo, quando ocorre a condição de fila cheia, redimensionamos para o dobro do tamanho.

```
if self._size == FilaLista.MAXF:
    self._resize(2*len(self._fila))
. . .
```

```
def _resize(self, newsize):
   fila_antiga[:] = self._fila[:] # guarda a fila antiga
   self._fila = [None] * newsize
   # reinsere os elementos na fila_antiga para self_fila
   . . .
```

O caso inverso ocorre quando removemos muitos elementos e a fila fica com um tamanho muito maior que a quantidade de elementos necessários. Basta neste caso realizar a operação inversa. Por exemplo, se a fila ficar ocupada com menos de ¼ do seu tamanho, reduzimos.

```
if 0 < self._size < len(self._fila) // 4:
   self._resize(len(self._fila) // 2)
```

Filas com dupla entrada e saida

Podemos pensar numa estrutura de fila em que é possível adicionar ou remover elementos tanto no início quanto no fim.

É uma fila de dupla entrada e saída e tem utilidade quando há a possiblidade do último elemento abandonar a fila, ou que o primeiro elemento resolve ceder o seu lugar para outro, mas tem a prioridade de voltar ao início da fila direto.

O que não dá para fazer facilmente é dar essa possibilidade a elementos do meio da fila.

Aplicações

Existem muitos algoritmos que precisam manipular estrutura de dados do tipo fila.
É usada principalmente para comunicação entre processos que possuem velocidades diferentes de geração de pedidos e atendimento de pedidos.
Por isso não é comum o uso desta estrutura na programação sequencial normal, pois se os pedidos são consumidos na mesma velocidade em que são gerados, não há necessidade de filas.
Esses algoritmos aparecem principalmente em comunicação entre processos e dentro do sistema operacional, que recebe requisições de vários programas ao mesmo tempo e tem que atendê-las sequencialmente.

Citamos alguns:

a) O controlador do disco é um processador dedicado para controlar o acesso ao disco. Vários usuários podem requisitar acesso a determinados setores do disco ao mesmo tempo. Como o atendimento a cada requisição demora alguns milisegundos, o controlador tem que gerenciar a fila de requisições dos usuários.

b) Um servidor de páginas web que atende um usuário de cada vez recebe ao mesmo tempo várias requisições de páginas de vários clientes da rede. Como só um cliente é atendido ao mesmo tempo, as várias requisições pendentes têm que permanecer numa fila. Neste caso, a fila é gerenciada pelo protocolo TCP que está em contato com a rede, recebendo as requisições dos usuários.

c) Uma parte importante do núcleo do sistema operacional é o escalonamento de processos. Os vários processos se alternam na execução, pois a CPU é uma só. O sistema operacional deve gerenciar a fila de processos prontos, esperando a disponibilidade da CPU. Neste caso, além de sua posição na fila, existe também a prioridade do processo. Isso é implementado pelo sistema operacional com uma fila de prioridades ou com várias filas, uma para cada prioridade.

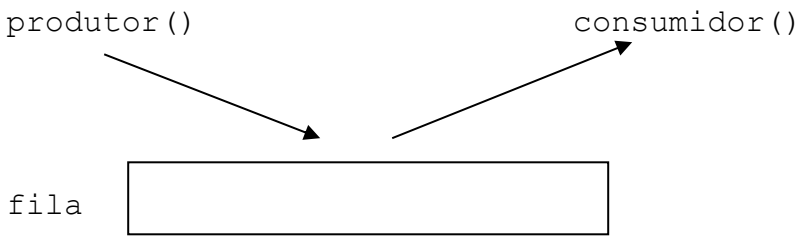
d) A fila da impressora. Num ambiente em rede, a impressora é usada por vários usuários. Num determinado instante, vários arquivos podem ser enviados por usuários diferentes à impressora para serem impressos. A impressora tem que colocar esses arquivos numa fila, pois a velocidade de impressão é muito menor que a velocidade de recepção dos arquivos.

Processos concorrentes

Os exemplos com filas são usados em geral quando o processamento é feito por processos assíncronos funcionando concorrentemente. A fila regula a comunicação entre os processos. Existem os processos "produtores" de elementos para a fila e os processos "consumidores" de elementos da fila. A produção e consumo de elementos da fila ocorre em velocidades diferentes. Daí a necessidade da fila.

```
# processo produtor
def Produtor():
  while True:
     produza (msg)
     # se a fila está cheia, fica esperando
     insere_msg_na_fila(msg);

# processo consumidor
def Consumidor():
  while True:
     # se fila está vazia fica esperando
     remove_msg_da_fila(msg)
     consuma(msg)
```



Neste exemplo, os processos Produtor e Consumidor são concorrentes.

Processos concorrentes em Python

Como exemplo, vamos implementar esses dois processos usando as ferramentas de sincronização de processos do Python. A sincronização de processos é um assunto bastante amplo de Sistemas Operacionais e vamos apenas mencionar os seus aspectos principais.

Uma **`Thread`** em Python é um processo em execução ou uma linha de execução de um programa. A classe que contém um programa concorrente é uma classe "filha" da classe **`Thread`** e precisa de uma função de nome **`run`**.

A classe **`Condition`** especifica os elementos de sincronização entre os processos concorrentes. O sincronismo é implementado através dos métodos:

`acquire()` e **`release()`**:
Bloqueia e desbloqueia um trecho de programa ou uma região crítica onde apenas um dos processos pode entrar por vez.

`wait()`:
Coloca o processo em espera de uma notificação. Só pode ser emitida se o processo está bloqueado. Libera o bloqueio e coloca o processo em espera da notificação. Quando recebe a notificação o bloqueio é retomado.

`notify()`:
Envia uma notificação a qualquer processo que esteja esperando esta notificação. Não libera o bloqueio imediatamente. O bloqueio só é liberado quando esse processo emitir um **`release()`**.

`<classe>().start()`:
Lança a **`Thread`** para execução (método **`run`** da **`Thread`**) e continua a execução normal do programa.

O exemplo abaixo lança duas Threads, protegendo ou bloqueando o contador e seu print. Sem esse bloqueio (tente rodar) os prints se confundem e também o incremento do contador.

```
from threading import Thread, Condition
import time

c = Condition() # para proteger a região crítica
cont = 0 # contador global aos dois programas abaixo

class Prog1(Thread):
    def run(self):
        global cont
        while True:
            c.acquire() # bloqueia região crítica
            cont += 1
            print("msg do Prog1 - ", cont)
            c.release() # libera região crítica
            time.sleep(0.1)
```

```
class Prog2(Thread):
    def run(self):
        global cont
        while True:
            c.acquire() # bloqueia região crítica
            cont += 1
            print("msg do Prog2 - ", cont)
            c.release() # libera região crítica
            time.sleep(0.1)

Prog1().start() # Lança o run do Prog1
Prog2().start() # Lança o run do Prog1
```

Exemplo: Fila compartilhada por 2 processos

No exemplo abaixo, uma fila de 10 elementos é manipulada por 2 processos. O processo Produtor, produz uma mensagem e a coloca na fila. O processo Consumidor retira uma mensagem da fila e a consome. As mensagens no exemplo são números aleatórios gerados mas poderiam ser qualquer tipo de mensagens.
Vamos usar uma lista simples do Python como fila (**FILA**):
Adicionar novo elemento: **FILA.append(msg)**
Remover elemento: **msg = FILA.pop(0)**

```
from threading import Thread, Condition
import time
import random

FILA = [] # fila inicialmente vazia
MAX_FILA = 10 # tamanho máximo da fila
cond = Condition()

class Produtor(Thread):
    def run(self):
        global FILA
        while True:
            # Produz uma nova mensagem
            msg = random.randrange(1000)
            cond.acquire() # Bloqueia e entra na região crítica
            if len(FILA) == MAX_FILA:
                print("<P>: Fila cheia - Produtor deve esperar notificação")
                cond.wait() # Desbloqueia a zona crítica e espera notificação
                print("<P>: Notificação recebida - Abriu espaço na fila")
            FILA.append(msg) # Coloque a msg na fila
            print("<P>: Mensagem colocada na fila:", msg)
            cond.notify() # Notifica que tem mais uma msg na fila
            cond.release() # Libera a região critica
            time.sleep(random.random()) # espera um tempo entre 0 e 1 segundo


class Consumidor(Thread):
    def run(self):
```

```
        global FILA
        while True:
            cond.acquire() # Bloqueia e entra na região crítica
            if FILA == []:
                print("<C>: Fila vazia - Consumidor deve esperar uma notificação")
                cond.wait()
                print("<C>: Notificação de nova msg na fila recebida")
            msg = FILA.pop(0) # retira msg da fila
            print("<C>: Consuma essa mensagem", msg)
            cond.notify() # Notifica que consumiu mais uma msg da fila
            cond.release() # Libera a região critica
            time.sleep(random.random()) # espera um tempo entre 0 e 1 segundo

# Lançamento dos 2 processos
Produtor().start()
Consumidor().start()
```

Filas com prioridades

Muitas vezes uma estrutura de fila simples não atende. Ocorre quando os elementos da fila têm prioridades diferentes para serem atendidos. Isso ocorre também no caso das filas da vida real. Veja o caso da fila dos clientes no banco, ou do supermercado. Tem que haver atendimento especial para idosos, gestantes, clientes preferenciais, etc.

Há duas soluções:

a) Várias filas. Uma para cada prioridade. Naturalmente ao retirar um elemento, sempre procuramos na fila de maior prioridade. Quando ela está vazia, tenta-se a fila com prioridade seguinte e assim por diante.
b) Uma só fila, só que quando se coloca um elemento na fila, não se coloca no final e sim no lugar correspondente à sua prioridade. Isso não é bem uma fila, pois não estamos usando o princípio de inserir no final e retirar do início.

Como já devem ter notado, essas soluções podem fazer com que elementos com baixa prioridade demorarem muito para serem atendidos e talvez nem o sejam. Portanto a administração de filas de prioridades requer algoritmos mais elaborados para um tratamento adequado.