PMR3201 Computação para Automação
Aula de Laboratório 7

Simulador de Sistema de Filas

Newton Maruyama
Thiago de Castro Martins
Marcos S. G. Tsuzuki
Rafael Traldi Moura
André Kubagawa Sato

23 de maio de 2020

PMR-EPUSP

Conteúdo

Filas FIFO,Filas de Prioridades

Filas FIFO

- Sistema onde o primeiro elemento a entrar é o primeiro a sair (First In First Out).

Implementação utilizando lista

- Usualmente a implementação mais trivial para uma fila é através da utilização de um array o que implica em uma fila com um tamanho limitado (**OBS:** Nesse caso a implementação mais eficiente é a utilização de um esquema de fila circular).
- Na linguagem Python não existe propriamente uma estrutura do tipo array.
- As filas são implementadas através de listas simples ou através de listas ligadas.

Exemplo

- Um exemplo de implementação de fila utilizando uma lista e suas operações básicas é apresentado abaixo.
- Operações básicas: `append()`, `pop()`
- Uma classe `Fila` é criada com operações para inserção, retirada e checagem de fila vazia.
- A implementação encontra-se no arquivo `teste_FilaListaSimples.py`

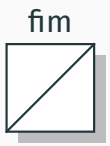
```
class Fila():
   def __init__(self):
      self.__fila = []
   def insereFila(self,dado):
      self.__fila.append(dado)
   def retiraFila(self): # retira o elemento da posicao 0 da lista
      return(self.__fila.pop(0))
   def FilaVazia(self): # verifica se a fila esta vazia
      if len(self.__fila) == 0:
         return(True)
      else:
         return(False)

if __name__== '__main__':
   f = Fila()
   f.insereFila("abacate")
   f.insereFila("banana")
   f.insereFila("kiwi")
   f.insereFila("amora")
   # Retirada dos elementos da FILA
   # e impressao
   while not f.FilaVazia():
      print(f.retiraFila())
```

Implementação de filas utilizando listas ligadas

- A utilização de listas ligadas para implementação de filas é bastante conveniente pois as operações de inserção e retirada são feitas nas extremidades da lista ligada o que permite complexidade constante.
- A implementação encontra-se no arquivo `teste_FilaListaLigada.py`.
- A classe `No` é um container de dados:

```
class No:    # Classe que armazena o conteudo
  def __init__(self, data):
  self.data = data
  sel .next = None
```

- No início os ponteiros `inicio` e `fim` são nulos.



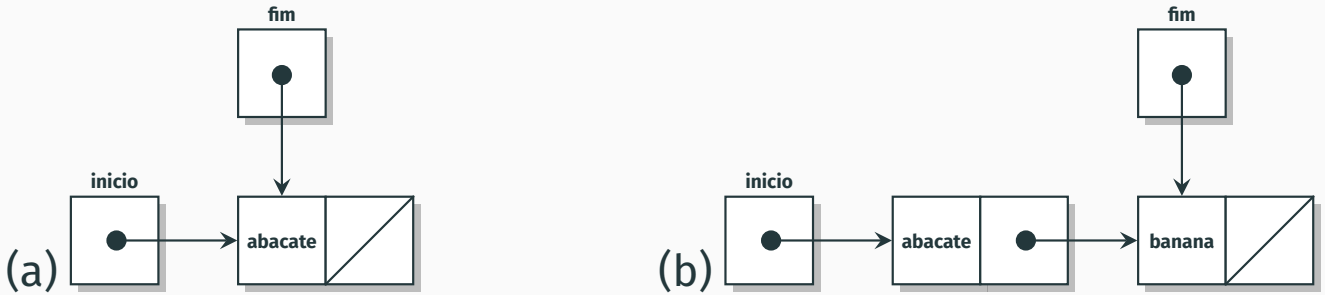

```
class Fila:
  def __init__(self): # Inicializacao da fila
     # Ponteiros inicio = fim = None
     self.inicio = None
     self.fim = None
```

Exemplo

- Define-se uma função para checar se a fila está vazia:

```
    def FilaVazia(self):      # Checa se a fila esta vazia
       return(self.inicio == None) # basta testar um ponteiro
```

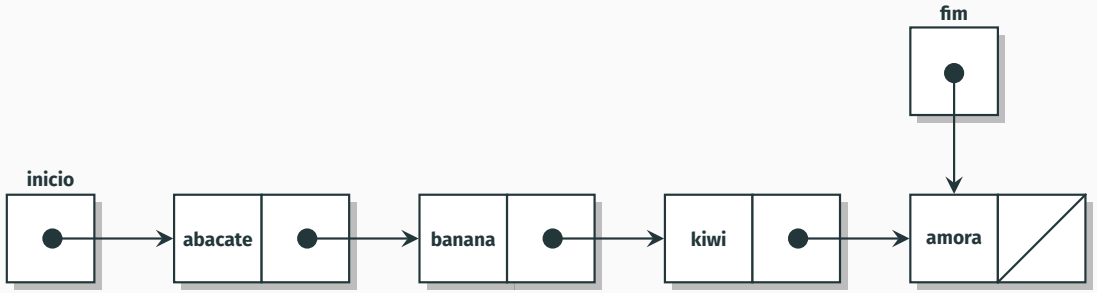
- Quando a fila está vazia a operação de inserção cria um novo objeto `No` com o conteúdo adequado e os dois ponteiros passam a apontar para esse objeto, figura (a), caso a fila não esteja vazia, o novo elemento deve ser conectado como ilustra a figura (b):



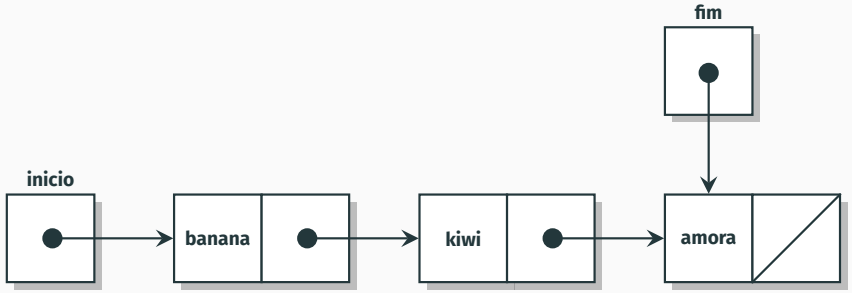

```
def InsereFila(self, item):
   temp = No(item)  # Cria um objeto No que passa a
   # armazenar item
   if self.fim == None: # Se a fila esta vazia entao
                        # os dois ponteiros apontam para o mesmo No
     self.inicio = temp
     self.fim = temp
   else: # Fila nao esta vazia
     # Novo objeto No conectado no final da fila
     self.fim.next = temp
     self.fim = temp
```

Exemplo

- Após 4 inserções obteríamos a seguinte configuração:



- A operação de remoção é feita modificando o conteúdo ponteiro `inicio` como ilustrado abaixo:




```
def RetiraFila(self):
    aux = None
    if not self.FilaVazia(): # Se a fila nao esta vazia
      temp = self.inicio
      aux = temp.data   # aux <-- data
      self.inicio = temp.next # movimenta o ponteiro inicio
    if self.inicio == None: # se a fila ficou vazia
      self.fim = None   # o ponteiro fim deve ser atualizado
    return(aux) # Se a fila estiver vazia retorna None
```

Exemplo

- O programa `main` que utilização a definição de fila é ilustrado abaixo:

```
if __name__== '__main__':
f = Fila()
f.InsereFila("abacate")
f.InsereFila("banana")
f.InsereFila("kiwi")
f.InsereFila("amora")
# Retirada dos elementos da FILA
# e impressao
while not f.FilaVazia():
    print(f.RetiraFila())
```

Fila de prioridades

- As filas de prioridades são filas onde elementos com maior prioridade são servidos antes dos elementos com menor prioridade.
- A maneira mais eficiente de implementação de uma lista de prioridades é através da utilização de árvores binárias que permite complexidade $O(\log n)$.
- A linguagem Python possui a biblioteca `heapq` que implementa uma árvore binária em listas simples.
- Um programa exemplo está implementado no arquivo `teste_FilaPrioridadeheap.py` e detalhado a seguir.
- Os elementos a serem inseridos são dicionários `(key,data)` onde `key` é uma chave de ordenação e `data` é o dado a ser armazenado:

```
import heapq
if __name__ == "__main__":
lista = []
a = (4,"abacate")
b = (1,"banana")
c = (7,"kiwi")
d = (3, "amora")
```

Fila de prioridades

- Inserções são realizadas através da utilização da função `heappush( )`:

```
heapq.heappush(lista,a)
heapq.heappush(lista,b)
heapq.heappush(lista,d)
heapq.heappush(lista,c)
```

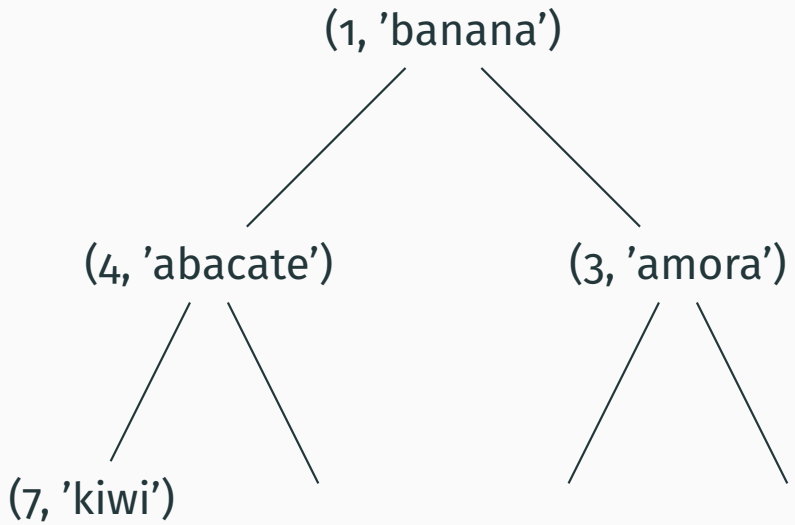
- Após as inserções podemos examinar o conteúdo de `lista`

```
print("Armazenamento interno")
print(lista)
```

cujo resultado é dado por:

```
Armazenamento interno
[(1, 'banana'), (4, 'abacate'), (3, 'amora'), (7, 'kiwi')]
```

- O armazenamento interno pode ser entendido como uma árvore binária:



Fila de prioridades

- As operações de retirada podem ser feitas através da função `pop(o)` que retira o elemento da posição o.
- Inserções e retiradas levam a reorganizações da árvore binária mas isso não é transparente para o usuário.
- As operações de retirada são feitas por ordem de prioridade como pode ser vista através do código a seguir:

```
# Retirada da lista na ordem de prioridades
print("\n Saida dos elementos por ordem de prioridade ->")
while lista:
  print(heapq.heappop(lista)) # pop(0) remove o primeiro elemento
```

O resultado é ilustrado a seguir:

```
Saida dos elementos por ordem de prioridade ->
(1, 'banana')
(3, 'amora')
(4, 'abacate')
(7, 'kiwi')
```

Sistemas de simulação de filas

Caracterização do sistema como um Sistema de Eventos Discretos

- Um sistema discreto possui variáveis que podem assumir apenas um número limitado de valores ou estados (On/Off por exemplo).
- Um sistema também pode ser determinístico ou estocástico dependendo da relação entre as suas entradas e saídas.
- Num **sistema determínistico**, é possível determinar o valor exato do estado final do sistema a partir do seu estado inicial.
- Num **sistema estocástico** não é possível determinar o valor exato de uma ou mais variáveis do sistema porque possuem um caráter intrinsicamente aleatório, por exemplo, o tempo de atendimento de um cliente no banco.
- A simulação de sistemas de eventos discretos é dirigida por eventos. O algoritmo deve examinar todos os eventos para descobrir qual será o próximo evento que vai ocorrer.

Caracterização do sistema de filas como um Sistema de Eventos Discretos

- Um dos modelos de sistemas de eventos discretos comumente utilizados são Sistemas de Filas (*Queueing System*, *Waiting Line*). Para tal, um corpo teórico, denominado Teoria de Filas, foi desenvolvido.
- A teoria de filas é utilizada em áreas diversas como: sistemas de atendimento a clientes (agências bancárias, supermercados, etc.), sistemas de manufatura e sistemas de computação (escalonamento e transmissão de dados em redes, escalonamento de processos em sistemas operacionais).
- Na Figura 1 apresenta-se um diagrama representando um modelo de filas de um sistema cliente-servidor. O fluxo de dados é regido por processos estocásticos.

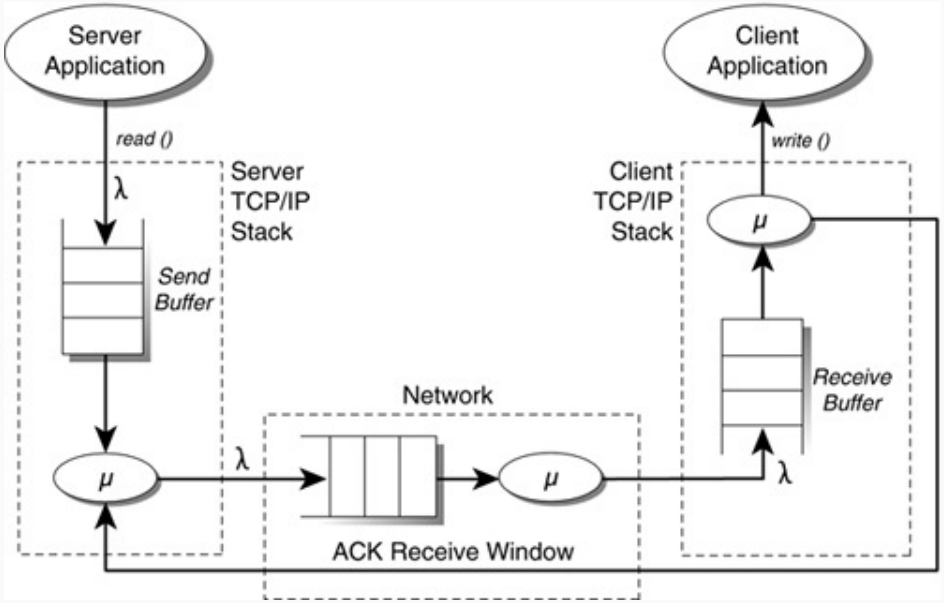


Figura 1: Modelo de filas de um sistema cliente-servidor.

Sistema de filas em agências bancárias

- O sistema de filas na agência bancária definido aqui possui um número de caixas dado por n_{caixas},
- O sistema é classificado como múltiplas filas múltiplos servidores (MFMS),
- Clientes chegam em intervalos de tempo que são caracterizados como uma **variável aleatória**,
- Uma vez escolhida a fila o cliente não pode mudar,
- o tempo de atendimento do caixa também é uma **variável aleatória** (OBS: na prática esse tempo depende do número de documentos que o cliente necessita processar e da eficiência do caixa).
- A figura abaixo ilustra um sistema desse tipo:

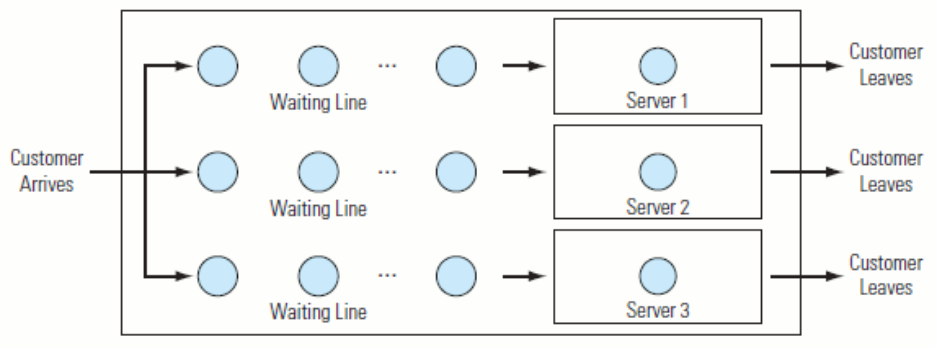


Figura 2: Sistema de fila múltiplas servidores múltiplos.

- A simulação é realizada por eventos. Os seguintes eventos são considerados: **Evento 1** - Chegada do cliente na agência, **Evento 2** - o caixa começa o atendimento do cliente, **Evento 3** - termina o atendimento do cliente.

Sequencia de ocorrência de eventos para um cliente

- O cliente chega na agência bancária no instante de tempo dado por t_{evento_1} (Evento1),
- O cliente verifica se existe algum caixa livre,
- Caso exista, ele escolhe um caixa e se dirige ao atendimento,
- Caso não exista caixa livre o cliente escolhe uma fila que tenha o menor comprimento e espera o atendimento,
- Quando começa o atendimento pelo caixa esse instante de tempo é denominado t_{evento_2} (Evento 2),
- Quando termina o atendimento esse instante de tempo é dado por t_{evento_3} (Evento 3).

Cálculo das grandezas que caracterizam o desempenho

- Através de simulações de sistemas de filas objetiva-se a estimativa de grandezas que caracterizam o desempenho do sistema.
- Em sistemas de atendimento ao cliente, por exemplo, procura-se estabelecer sistemas de filas que possam aumentar a eficiência do sistema (taxa de atendimento alta) o que consequentemente promove a satisfação dos clientes (tempo de espera na fila pequeno)
- Para cada cliente com atendimento finalizado podemos calcular:
  - **Tempo de espera na fila**: $\Delta t_{fila} = t_{evento_2} - t_{evento_1}$,
  - **Tempo de atendimento**: $\Delta t_{atend} = t_{evento_3} - t_{evento_2}$,
  - **Tempo total na agência bancária**: $\Delta t_{total} = t_{evento_3} - t_{evento_1}$
- Para um número de clientes $n_{clientes}$ podemos calcular a média e variância de cada uma dessas grandezas:
  - **tempo de espera na fila**: $\overline{\Delta t_{fila}}$, $Var_{\Delta t_{fila}}$
  - **tempo de atendimento**: $\overline{\Delta t_{atend}}$, $Var_{\Delta t_{atend}}$
  - **tempo total**: $\overline{\Delta t_{total}}$, $Var_{\Delta t_{total}}$

Cálculo das grandezas que caracterizam o desempenho

- Uma outra grandeza importante é o número médio de clientes na fila que pode ser calculado da seguinte forma:

$$\overline{n}_{CF} = \frac{\int_0^{T_{final}} n_{CF}(t)dt}{T_{final}}, \tag{1}$$

  onde T_{final} é o tempo quando o último cliente terminam de ser atendido.
- Como nessa simulação são utilizadas variáveis aleatórias sabemos que os resultados obtidos de uma única simulação não tem relevância estatística, dessa forma, devemos realizar um **número significativo de repetições** n_{rep}.

Distribuições de probabilidades em simulação de filas

Chegada de clientes

- Devemos determinar o instante de tempo absoluto $t_{evento_1}(i)$ de chegada dos clientes $i = 1 \ldots n_{clientes}$,
- Para tal inicialmente amostraremos uma função densidade de probabilidade $f(\lambda)$ que nos fornecerá valores aleatórios para o intervalos de tempo entre chagadas de clientes $\Delta t_{evento_1}(i-1)$,
- A partir de amostras de $\Delta t_{evento_1}(i-1)$ podemos facilmente obter os instantes de tempo absolutos.
- A figura abaixo ilustra o processo:

$$0 \xrightarrow{\Delta t_{evento_1}(0)} \mathbf{t_{evento_1}(1)} \xrightarrow{\Delta t_{evento_1}(1)} \mathbf{t_{evento_1}(2)} \xrightarrow{\Delta t_{evento_1}(2)} \cdots$$

$$\cdots \xrightarrow{\Delta t_{evento_1}(n_{clientes-1})} \mathbf{t_{evento_1}(n_{clientes})}$$

Chegada de clientes: Processos de Poisson

- Filas de eventos usualmente obedecem a Processos de Poisson.
- A chegada de clientes numa agência bancária por exemplo pode ser expressa através de uma distribuição de Poisson.
- A distribuição de Poisson descreve a probabilidade da chegada de n clientes durante um período de tempo T e é expressa por:

$$P_T(n) = \frac{(\lambda T)^n \exp^{-\lambda T}}{n!} \tag{2}$$

  onde λ é a média da taxa de chegada de clientes no tempo T.
- Por exemplo, se a taxa média de chegada de clientes é de 3 clientes por minuto ($\lambda = 3$) e deseja-se saber qual a probabilidade da chegada de 5 clientes por minuto ($n = 5, T = 1$), utilizando a equação acima obtemos:

$$P_1(5) = 0.101 \tag{3}$$

Chegada de clientes: Processos de Poisson

- A figura abaixo ilustra uma distribuição de Poisson para $\lambda = 3$. Tanto a média como a variância da distribuição de Poisson são iguais a λ.

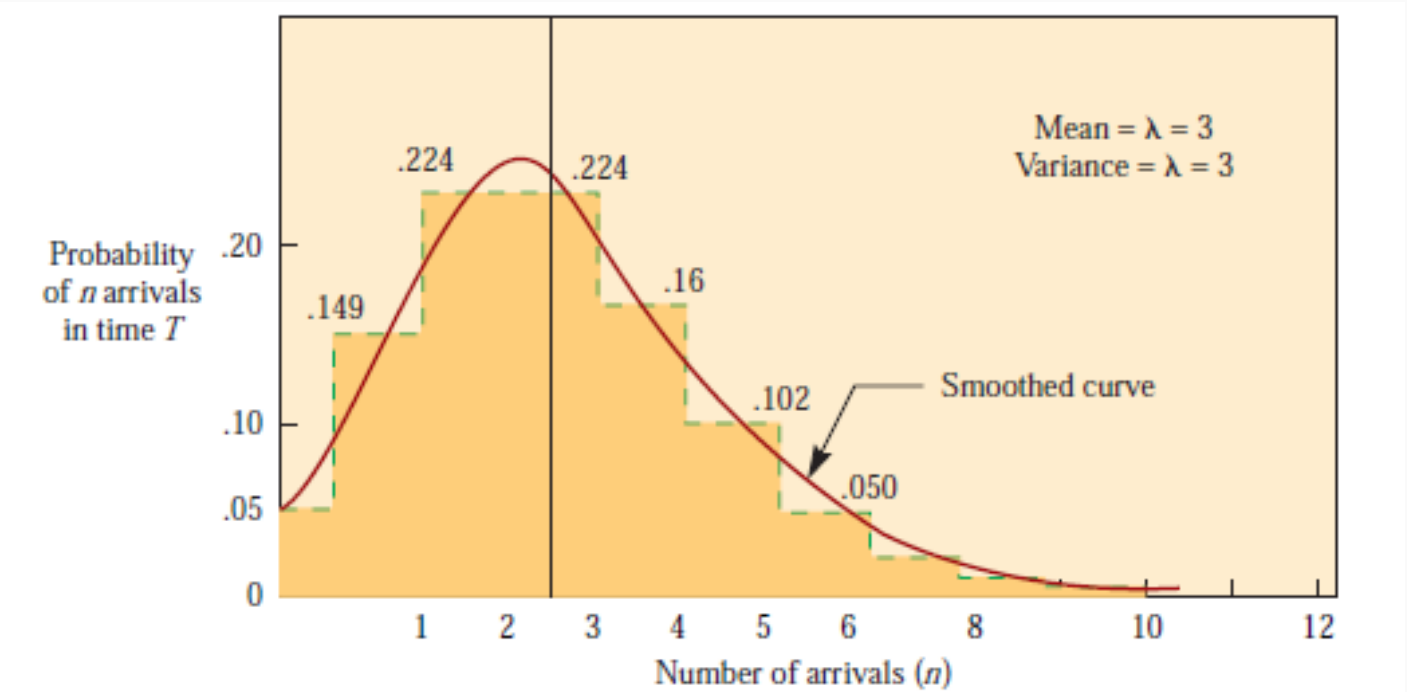


Figura 3: Distribuição de Poisson para $\lambda = 3$ (Fonte [2])

Distribuição exponencial

- Se a chegada de clientes segue um processo de Poisson então a função densidade de probabilidade exponencial modela a distribuição de intervalos de tempo entre eventos:

$$f(t) = \lambda \exp^{-\lambda t}, \tag{4}$$

  onde λ é o número médio de chegadas por unidade de tempo.
- Obviamente, λ da Equação 4 é o mesmo da Equação 2. A média da distribuição exponencial é $1/\lambda$ e sua variância $1/\lambda^2$. Se existe uma média de λ chegadas por unidade de tempo é natural imaginar que na média uma chegada demora $1/\lambda$.

- Obviamente o mesmo raciocínio vale para o modelo para a função densidade de probabilidade do intervalo de tempo que o servidor demora para finalizar uma tarefa (Taxa de serviço). Por exemplo, quanto tempo o caixa demora para atender um cliente.
- A Figura 4 ilustra uma função densidade de probabilidade exponencial.

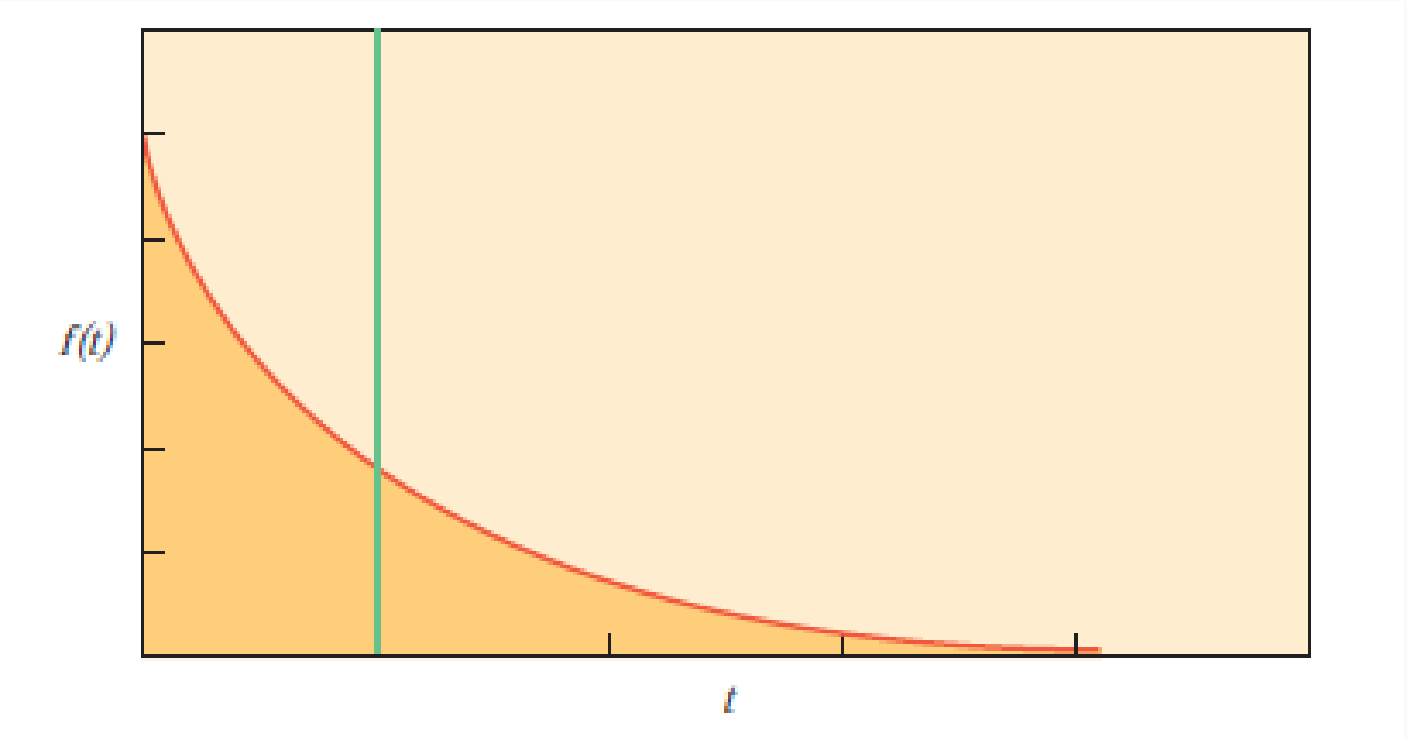


Figura 4: função densidade de probabilidade exponencial para $\lambda = 3$ (Fonte [2]).

Geração de intervalos de tempo entre chegadas de clientes

- Uma função denominada `fexp()` foi desenvolvida e se encontra no arquivo `AleatorioAlunos.py`.
- A função recebe como parâmetro de entrada a taxa de chegada de clientes em segundos e a duracao do expediente em segundos.
- A função retorna uma lista contendo os intervalos de tempo em segundos Δt_{evento_1} entre chegadas de clientes consecutivos.
- O número de amostras gerados é compatível com a duração do expediente. O último cliente não pode chegar após o término do horário de expediente.

```
def fexp(lambdacliente,duracaoexpediente):
x = 1000*[None]   # x e' alocado com 1000 posicoes
   somatoriadotempo = 0 # Somatoria dos intervalos de tempo Dtev1
   k = 0
   # Amostras sao geradas ate' que a somatoria dos intervalos de tempo
   # ultrapasse a duracao do expediente
   while somatoriadotempo <= duracaoexpediente:
     x[k] = rand.expovariate(lambdacliente)
     x[k] = int(x[k])
     somatoriadotempo = somatoriadotempo + x[k]
     k = k+1
   # retira-se o ultimo elemento que esta' alem do horario do expediente
   return x[0:k-1]
```

Arquivo teste: `teste_fexp.py`

- Um programa teste para ilustrar a utilização da função se encontra no arquivo `teste_fexp.py`.
- O código do programa é ilustrado a seguir:

```
import numpy as np
import random as rand
from matplotlib import pyplot as plt
from AleatorioAlunos import *

if __name__ == "__main__":
  lambdacliente = 40/3600 # Taxa de chegadas por seg
  duracaoexpediente = 6*3600; # Duracao do expediente em seg
  Dtev1 = fexp(lambdacliente,duracaoexpediente)
  print("Somatoria Dtev1 = ",sum(Dtev1)," Expediente 6h = ",\
  duracaoexpediente)
  # Histograma com 20 setores
  plt.hist(Dtev1,20)
  plt.show()
```

- Teste o programa e verifique o seu funcionamento

Geração da duração do tempo de atendimento no caixa

- A função `TempoDeAtendimentoCaixa()` gera a duração do tempo de atendimento no caixa utilizando a taxa de atendimento no caixa μ_{caixa} e a imposição de um tempo mínimo de atendimento.

```
def TempoDeAtendimentoCaixa(mucaixa,tempominimoatendimento):
    x = int(rand.expovariate(mucaixa))
    if x >= tempominimoatendimento:
      return(x)
    else:
      return(tempominimoatendimento)
```

- O código dessa função se encontra no arquivo `AleatorioAlunos.py`

Especificação do programa EP2

Especificação funcional

- O programa executar uma simulação estocástica de um sistema com múltiplas filas e múltiplos servidores mais especificamente um sistema de filas de atendimento numa agência bancária.
- O programa deve atender aos seguintes requisitos:
  - taxa de chegada de clientes no sistema,
  - taxa de atendimento individualizada para cada caixa,
  - tempo mínimo para um atendimento do caixa,
  - especificação do número de repetições da simulação,
  - análise desempenho através de cálculo de média e variância,
  - display gráfico do estado da fila para uma simulação específica (iteração).

Código a ser implementado no EP2: classes, funções e programa principal

- **Uma parte do código do sistema de simulação de filas é fornecida pronta**.
- A estrutura de arquivos proposta e as funções e classes que os compõem são apresentados a seguir. **As partes a serem desenvolvidas pelos alunos estão evidenciadas por retângulos**.
- Os seguintes arquivos devem compor o seu sistema:
  - `SimuladorDeFilas.py`:
    - Variáveis globais
    - Função `simulacao`
    - Programa principal `main()`
  - `FilasAlunos.py`:
    - Classe `Cliente`
    - Classe `FilaCaixa`
    - Classe `FilaEventos` (Nova implementação com estrutura dinâmica)
    - Função `verificaCaixaLivre(sinalCaixaLivre)`
    - Função `achaMenorFila(vetorFilaCaixa)`
    - Função `TamanhoDasFilas(vetorFilaCaixa)`
    - Classe `ListaClientesSaida`
  - `AleatorioAlunos.py`
    - Função `fexp(lambdacliente,duracaoexpediente)`
    - Função `TempoDeAtendimentoCaixa(mucaixa,tempominimoatendimento)`
    - Função `calculaEstatisticas(listasaida)`

Variáveis globais

- `lambdacliente_hour`: taxa de chegada de clientes na agência bancária por hora,
- `duracaoexpediente_hour`: duração do expediente em horas,
- `nCaixas`: número de caixas,
- `vetorFilaCaixa[]`: array onde cada posição armazena um objeto da classe `FilaCaixa`,
- `vetorTaxaDeAtendimentoCaixa[]`: array onde a posição i armazena a taxa de atendimento μ_i do caixa i. Obs: a taxa μ_i pode ser individualizada.
- `tempominimodeatendimento`: tempo mínimo que um atendimento é realizado em segundos,
- `sinalCaixaLivre[]`: array booleano `(True,False)` que indica se o caixa i está livre para atendimento,
- `FE`: objeto da classe `FilaEventos`,
- `saida`: objeto da classe `ListaClientesSaida`
- `ltemporaltamfila`: lista onde sub-listas se referem a um instante de tempo e o número de clientes em cada fila.
- `numeroderepeticoes`: número de repetições que uma simulação deve ser realizada.

OBS: Esse conjunto de variáveis é apenas uma proposta, o aluno não precisa necessariamente utilizar como especificado.

Classe Cliente: variáveis

- A classe denominada `Cliente` define objetos que armazenam todas as informações de ocorrências dos eventos para um cliente específico.
- Utiliza-se a terminologia *Inserir cliente na fila* ou *Inserir um Evento 1 na Fila de Eventos* se referindo ao mesmo objeto.
- As variáveis internas da classe são ilustradas a seguir:

```
class Cliente():
    def __init__(self,clienteid, tipoEvento,timeEvento1):
    self.__clienteid = clienteid
    self.__tipoEvento = tipoEvento # tipo do evento que sera' o
                                   # proximo a ocorrer
    self.__timeEvento1 = timeEvento1
    # instante de tempo de ocorrencia
    # do Evento 1
    self.__timeEvento2 = 0  # idem Evento 2
    self.__timeEvento3 = 0  # idem Evento 3
    self.__caixaid = 0      # identificacao do caixa e da fila
                            # em que foi colocado
```

- `__clienteid`: número inteiro sequencial que identifica o cliente.
- `__tipoEvento`: **identifica o tipo do evento que será o PRÓXIMO a a ocorrer**. Pode assumir os valores 1,2 e 3
- `_timeEvento1`,`__timeEvento2`,`__timeEvento3`: instantes de tempo absolutos em que ocorreram ou ocorrerão os eventos 1, 2 e 3. Unidade de tempo em segundos.
- `__caixaid`: número inteiro $(0, 1, 2, \ldots)$ que identica o caixa e a fila associada.

Classe Cliente: operações associadas

- Setter e getter para `__clienteid`:

```
def set_clienteid(self,clienteid):
    self.__clienteid = clienteid

def get_clienteid(self):
    return(self.__clienteid)
```

- Setter e getter para `__tipoEvento`:

```
def set_tipoEvento(self,tipo):
    if tipo in range(1,4):
      self.__tipoEvento = tipo
    else:
      print('Tipo de evento nao existente')

def get_tipoEvento(self):
    return(self.__tipoEvento)
```

Classe Cliente: operações associadas

- Setter e getter para `__timeEvento1,__timeEvento2,__timeEvento3`:

```
def set_timeEvento(self,time,tipoevento):
    if tipoevento == 1:
      self.__timeEvento1 = time
   else:
      if tipoevento == 2:
        self.__timeEvento2 = time
      else:
        if tipoevento == 3:
          self.__timeEvento3 = time
def get_timeEvento(self,tipoevento):
    if tipoevento == 1:
      return(self.__timeEvento1)
    else:
      if tipoevento == 2:
        return(self.__timeEvento2)
      else:
        if tipoevento == 3:
          return(self.__timeEvento3)
```

Classe Cliente: operações associadas

- Setter e getter para `__caixaid`:

```
def set_caixaid(self,caixaid):
    self.__caixaid = caixaid
def get_caixaid(self):
    x=self.__caixaid
    return(x)
```

- Definição da relação de ordem quando se compara dois objetos da classe Cliente:

```
def __lt__(self, other):
    return self.__clienteid < other.__clienteid
```

Necessário para implementação da fila de prioridades com `heapq`

- Definição da string de impressão (útil para Debug):

```
def __str__(self):
    x = "clienteid = " + str(self.__clienteid) + "\n" + \
    "tipoEvento = " + str(self.__tipoEvento) + "\n" + \
    "timeEvento1 = " + str(self.__timeEvento1) + "\n" + \
    "timeEvento2 = " + str(self.__timeEvento2) + "\n" + \
    "timeEvento3 = " + str(self.__timeEvento3) + "\n" + \
    "Caixa ID = " + str(self.__caixaid) + "\n\n"
    return(x)
```

Classe FilaEventos: variáveis e operações

- A única variável é a fila `__filae`

```
class FilaEventos():
    def __init__(self):
        self.__filae = []
```

- Insere e retira eventos da fila de eventos:

```
# x e' um dicionario (time,cliente)
# time: funciona como a chave de ordenacao da fila de
# prioridade
# cliente: objeto da classe Cliente()
def insereFilaEventos(self,x):
    heapq.heappush(self.__filae,x)

# Retira da fila de eventos
# retorna o objeto da classe Cliente()
def retiraFilaEventos(self):
    x = heapq.heappop(self.__filae)
    return(x[1])
```

- Verifica se a fila de eventos está vazia:

```
def FilaEventosVazia(self):
    if self.__filae == []:
        return(True)
    else:
        return(False)
```

- **Essa implementaçao é colocada aqui como referência, o aluno deve implementar uma solução com lista ligada.**

Classe ListaClientesSaida: variáveis e operações

- Essa classe contêm apenas a lista, `__filasaida`, onde objetos da classe Cliente são armazenados quando o atendimento é terminado.

```
class ListaClientesSaida():
     def __init__(self):
        self.__filasaida=[]
```

- Insere e retira da lista:

```
def insereListaSaida(self,cliente):
   self.__filasaida.append(cliente)

def retiraListaSaida(self):
   return(self.__filasaida.pop(0))
```

- Verifica se a lista está vazia:

```
def ListaSaidaVazia(self):
   if len(self.__filasaida) == 0:
      return(True)
   else:
      return(False)
```

- Calcula o tamanho da lista:

```
 def tamListaSaida(self):
    x = len(self.__filasaida)
    return(x)
```

Classe FilaCaixa

- A classe `FilaCaixa` implementa uma fila com comportamento FIFO (First-In First-Out) e deve ser implementado utilizando uma estrutura do tipo lista. Essa fila tem número de posições ilimitadas.
- As seguintes operações devem ser implementadas:
  - `insereFilaCaixa(cliente)`: insere na fila do caixa um objeto da classe `Cliente`,
  - `retiraFilaCaixa()`: retira um objeto da fila do caixa,
  - `FilaCaixaEstaVazia()`: verifica se a fila está vazia,
  - `tamFila()`: calcula o tamanho da fila (número de clientes).

Classe FilaEventos: implementação dinâmica com lista ligada

- **O aluno deve desenvolver essa classe de maneira que possa ter comportamento equivalente ao apresentado com** `heapq`.
- A classe `FilaEventos` implementa uma fila com prioridades o conteúdo da fila são objetos da classe `Cliente` e a chave de ordenação (*key*) corresponde ao instante de tempo absoluto em que ocorrerá o próximo evento.
- As seguintes operações devem ser implementadas:
  - `insereFilaEventos(x)`: `x` é um dicionário `x=(timeEvento,cliente)` onde `timeEvento` é o instante de tempo absoluto da ocorrência do próximo evento e `cliente` é um objeto da classe `Cliente`.
  - `retiraFilaEventos()`: devolve apenas o objeto da classe `Cliente`
  - `FilaEventosVazia()`: verifica se a fila de eventos está vazia.

Outras funções do arquivo `FilasAlunos.py`

- `verificaCaixaLivre(sinalCaixaLivre)`: verifica se existe um caixa livre. Devolve o dicionário `(sinal,caixaid)` onde sinal é uma variável booleana. Se `sinal=True` então `caixaid` deve conter um índice ($0, 1, 2, \ldots$) correspondente a um caixa livre. Função pode devolver índice referente a qualquer caixa livre.
- `achaMenorFila(vetorFilaCaixa)`: devolve o índice correspondente a menor fila de caixa. `vetorFilaCaixa` é um array que contêm em cada posição um objeto da classe `FilaCaixa`. Se houver mais de uma fila com o menor tamanho a função pode devolver o índice de qualquer uma dessas filas.
- `TamanhoDasFilas(vetorFilaCaixa)`: devolve uma lista contendo o tamanho de cada uma das filas.

Função `Simulacao`

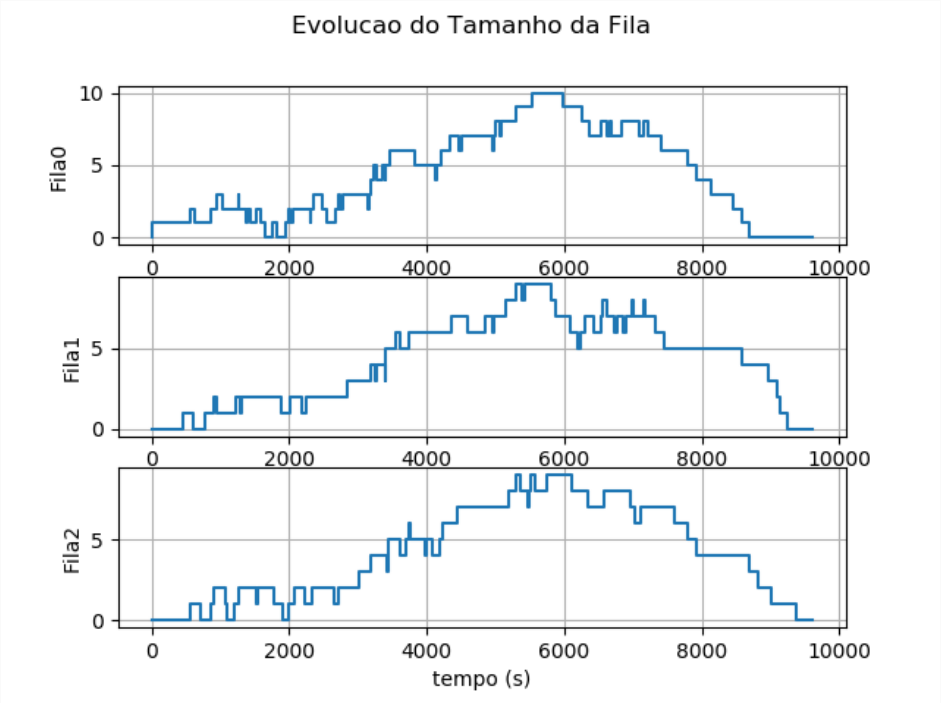
- A função `Simulacao` deve implementar um ciclo completo de simulação.
- As variáveis de retorno dessa função devem ser `saida` objeto da classe `ListaClientesSaida` que deve conter todos os objetos da classe `Cliente` na ordem em que o atendimento foi terminado e `logfilas` uma lista contendo instantes de tempo t em que ocorreram aumento ou dimimuição do tamanho de alguma fila e o tamanho de cada uma das filas:
- `logfilas` pode ser usado para plotar o número de clientes na fila a cada instante.
- Um possível algoritmo para referência de implementação é apresentado a seguir.

Função Simulacao

```
Criar <nc> eventos <Evento1> # criar objetos da classe <Cliente>
                             # todos os eventos   <Evento1> são criados no inicio
Inserir eventos <Evento1> na fila de eventos <FE>
Enquanto houver eventos em <FE> Faça
  {
  <evento> <-- retira evento de <FE>
  <clock>  <-- instante absoluto de ocorrencia de <evento>
  Se <evento> for <Evento1> então
     {
         Se existe(m) caixa(s) livre(s) Então
            {
            escolhe um caixa livre identificado como <caixaid>
            cria um Evento2 para o instante de tempo <clock> associado a <evento>
            Sinaliza o caixa identificado por <caixaid> como ocupado
            }
         Senão
           {
           Escolha uma fila de caixa mais curta
           Insere <evento> na fila de caixa
           Loga o estado das filas em <logfilas>
           }
    Senão Se <evento> for do tipo 2 Então
            {
            Dt_atend <-- amostra a funcao que gera o tempo de atendimento desse caixa
            Gera um <Evento3> para <evento> associado ao instante de tempo <clock>+<Dt_atend>
            Insere <evento> na fila de eventos <FE>
            }
         Senão # <Evento 3>
            {
            Coloca <evento> na lista de saida de clientes <saida>    # <Classe ListaSaidaCientes>
            O caixa identificado como <caixaid> associado a <evento> é sinalizado como livre
            Se existem clientes nessa fila Então
               {
               <proximoevento> <-- retira evento da fila do caixa
               Loga o estado das filas em <logfilas>
               Cria um <Evento2> aassociado a <proximovento> associado ao instante de tempo <clock>
               Insere <proximoevento> na fila de ventos <FE>
            }

     }
  }
```

Programa principal `main`

- O programa principal deve controlar a realização das várias iterações da função `Simulacao()`.
- O número de iterações é dado pela variável `numeroderepeticoes`
- Cálculos de médias e variâncias que caracterizam o desempenho do sistema devem ser realizados considerando todas as iterações.
- Deve ser implementado um gráfico tipo degrau (`step()`) que visualiza o estado da fila do caixa a cada instante.
- Deve ser permitido ao usuário escolher qual iteração específica ($i = 0, \ldots,$ `niteracoes-1`) deve ser visualizada.



Programa principal `main`

- Uma possível solução para o gráfico tipo degrau é ilustrado abaixo:

```
    # Plot do estado das filas ao longo do tempo
    # numero de subplots e' dependente de nCaixas
    # cada subplot e'incorporado sequencialmente
    # x = tempo da simulacao em segundos
    # y = tamanho de uma das filas
    #
    fig, ax = plt.subplots(nCaixas)
    fig.suptitle('Evolucao do Tamanho da Fila')
    nlines = len(ltemporaltamfila)
    # x recebe a coluna 0 da lista
    x = [row[0] for row in ltemporaltamfila]
    for k in range(1,nCaixas+1):
    # y recebe a coluna k da lista
       y = [row[k] for row in ltemporaltamfila]
       ax[k-1].step(x, y)
       ax[k-1].set_xlabel('tempo (s)')
       ax[k-1].set_ylabel('Fila'+str(k-1))
       ax[k-1].grid()
    fig.canvas.draw()
```

Para você fazer

Exercício para entrega

- Implementar a classe `FilaEventos` através de listas ligadas como especificado acima.
- Como um objeto da classe `FilaEventos` pode conter qualquer objeto, teste o seu programa com o seguinte programa teste:

```
if __name__ == "__main__":
x = FilaEventos()
a = (4,"abacate")
b = (1,"banana")
c = (7,"kiwi")
d = (3, "amora")

x.insereFilaEventos(a)
x.insereFilaEventos(b)
x.insereFilaEventos(c)
x.insereFilaEventos(d)

while not x.FilaEventosVazia()
  print(x.retiraFilaEventos())
```

- **Deadline: 29/05/2020**

Bibliografia

Referências

1. Christos G. Cassandras and Stephane Lafortune, *Introduction to Discrete Event Systems*,Springer, 2nd Edition, 2009.
2. F. Robert Jacobs and Richard B. Chase, *Operations and supply management: the core*, McGraw-Hill, 2012.
3. `http://www.promodel.com`
4. `http://www.arenasimulation`