Filas, Pilhas, Deques em Python

Filas

Até o momento, você lidou com estruturas de dados que permitiam inserção e remoção de nós em quaisquer posições. Entretanto, para algumas aplicações, queremos utilizar uma estrutura de dado que remova nós (também chamada de consumir os nós) na mesma ordem em que esses nós foram adicionados. Para isso, usamos a fila.

Conceito de fila

A estrutura de dados fila tem esse nome porque se assemelha conceitualmente às filas do mundo real. Nela, quem chegou primeiro está aguardando mais tempo e também é atendido primeiro. Você pode pensar na fila como uma fila única de um mercado ou banco.

Não importa quantos caixas existam, o primeiro da fila sempre será chamado (aquele que está esperando há mais tempo). Esse comportamento pode ser descrito como "primeiro a entrar, primeiro a sair" **(First In, First Out - FIFO)**, comumente usado para descrição das filas.

Implementação de fila

A fila pode ser implementada em alocação contígua ou encadeada, como as listas. De fato, a fila é uma lista especial na qual todos os nós são inseridos ao final da lista, e todos os nós são removidos apenas no início da lista.

Para uma alocação encadeada, você pode manter duas variáveis, uma para o início da fila, e outra para o final da fila. Se a fila estiver vazia, ambas as variáveis apontarão para o valor nulo. Se houver apenas um elemento, ambas apontarão para esse elemento. Com mais elementos, InícioFila apontará para o nó que pode ser removido, e FinalFila apontará para o último elemento a entrar, que apontará para o próximo nó a ser inserido. O código a seguir representa a criação da classe e seu construtor.

Para uma alocação contígua, você tem duas opções:

1. **Manter sempre o início da fila no endereço inicial da memória alocada, e ir inserindo ao final da fila, no próximo endereço vazio. Para facilitar esse processo, você pode manter uma variável guardando o final da fila. Entretanto, esse método tem a desvantagem de que, quando um nó é removido (do início), todo o restante da fila tem de ser ajustado para frente em memória.**

2. **Ir alocando os nós sequencialmente em memória, e manter uma variável para o início da fila, e uma para o final da fila. Ao remover um nó (do início), você apenas move a variável InícioFila um nó para frente. Ao inserir um novo nó, você move a variável FinalFila um nó para frente, para o novo nó. Esta é a opção mais comum em se tratando de alocação contígua.**

Quando atingir o final do espaço reservado em memória, você recomeça do início. Se em qualquer momento, ao tentar inserir um nó, o FinalFila fosse se mover e alcançar InicioFila, sua fila estará cheia e o novo nó não poderá ser inserido. O código a seguir inicializa uma fila com máximo de 10 valores, veja:

```
maxFila=10
fila=[None]*maxFila
inicioFila=None
finalFila=None
```

Operações em fila

Inserção em fila

A inserção em fila deve ocorrer sempre no final da fila.

Para o caso encadeado, você pode usar o seguinte código (novoNo deve estar alocado e com próximo apontando para o nulo):

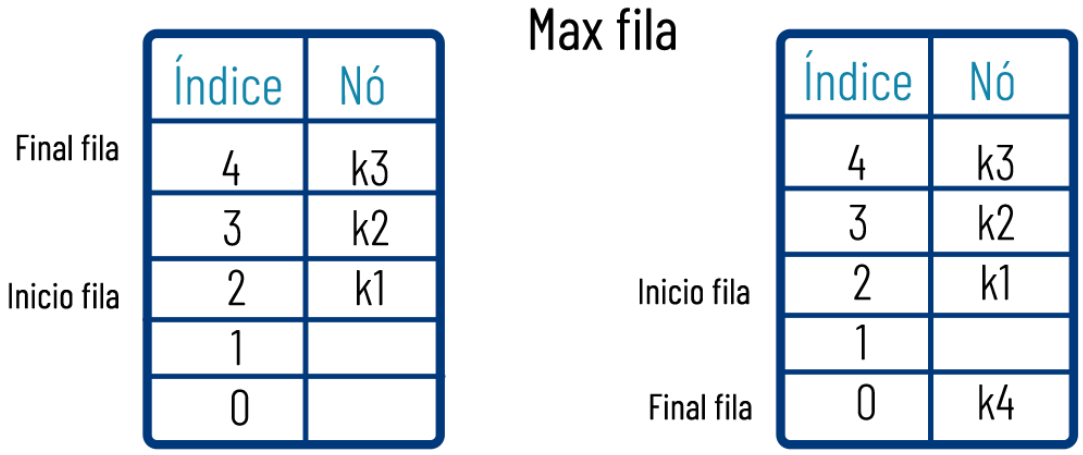
```
def insere(self, novoNo):
        if self.inicio==None:                    #fila vazia
            self.inicio=novoNo                   #atualiza o início da fila
        self.final=novoNo                        #atualiza o final da fila
    else:
        self.final.proximo=novoNo                #insere o nó
        self.final =novoNo                       #atualiza o final da
```

Para o caso contíguo, você possui 4 variáveis: Fila guarda o espaço de memória. MaxFila guarda o número máximo de elementos na fila, InicioFila e FinalFila guardam os índices de início e final da fila, respectivamente.

```
def insereFila(novoNo):
    global inicioFila                            #indica o acesso a variáveis globais
    global maxFila
    global finalFila
    global fila
if inicioFila== None:                            #fila vazia
        fila[0] = novoNo                         #insere o nó
    inicioFila=0                                 #atualiza o início da fila
```

```
                finalFila=0                                    #atualiza o
final da fila
            elif (finalFila+1) % maxFila ==inicioFila:        #fila cheia
                return -1                                      # -1 indica erro
de fila cheia
            else:
                finalFila=(finalFila+1) % maxFila             #atualiza o final da
fila
                fila[finalFila] = novoNo                      #insere o nó
            return finalFila                                   #saída normal
```

O detalhe nesse último código fica pelo uso da lógica modular representada pelo operador %. Ao somar 1 na variável FinalFila, se você atingir o valor de MaxFila, ao aplicar o módulo MaxFila, a variável FinalFila volta para 0. Veja essa lógica no esquema gráfico das filas a seguir (antes e após a inserção do nó k4):



Complexidade da inserção na fila

Analisando as funções de inserção na fila, e desconsiderando os tratamentos de erro de fila cheia, você pode verificar que a inserção consiste apenas em 2 ou 3 operações de atribuição, portanto, podemos dizer que a inserção em filas é O(1), ou constante.

Remoção da fila

A remoção da fila (às vezes chamado de consumir um nó) deve ocorrer sempre no início da fila. Para o caso encadeado, você pode usar este código:

```
def remove(self):
        if self.inicio==None:                          # erro -fila vazia
                return None                            # None indica erro fila vazia
else:
```

```
        k =self.inicio                          #salva o nó removido
        self.inicio=self.inicio.proximo       #remove o nó
return k                                      #retorne k=o nó consumido
```

Para o caso contíguo, você possui 4 variáveis: Fila guarda o espaço de memória, MaxFila guarda o número máximo de elementos na fila, InicioFila e FinalFila guardam os índices de início e final da fila, respectivamente.

```
def removeFila():
    global inicioFila            #indica o acesso a variáveis globais
    global maxFila
    global finalFila
    global fila
    if inicioFila== None:                #fila vazia
        return None                    #erro fila vazia
    k=fila[inicioFila]                 #salva o nó removido
    if finalFila==inicioFila:
        inicioFila=None                #fila vazia após remoção
    else:
        inicioFila=(inicioFila+1) % maxFila    #remove o nó
    return k                                # retorne k=o nó consumido
```

Para testar:

```
e0=No(0,'Joao')
fila=FilaEncadeada(e0)
k0=fila.busca(0)
print(k0.valor)
fila.print()
e1=No(1,'Maria')
fila.insere(e1)
fila.print()
e2=No(-1,'Ana')
fila.insere(e2)
fila.print()
e3=No(2,'Arthur')
```

```
fila.insere(e3)

fila.print()

k=fila.remove()

print("No removido: "+k.valor)

fila.print()
```

Já para testar sua fila em alocação contígua, você pode usar o seguinte código:

```
print(fila)
for i in range(10):
        insereFila(i)
        print(fila)
print(insereFila(11))
for i in range(10):
        print(removeFila())
print(removeFila())
```

Complexidade da remoção da fila

Analisando as funções de remoção da fila, e desconsiderando os tratamentos de erro de fila vazia, você pode verificar que a remoção consiste apenas em 2 operações de atribuição e, possivelmente, 2 comparações, portanto, podemos dizer que a remoção em filas também é O(1), ou constante.

Pilhas

Após aprender sobre as filas, você, agora, verá a estrutura de dados que permite colocar e remover nós apenas no início da lista: a pilha.

Conceito de pilha

A estrutura de dados pilha tem esse nome porque se assemelha conceitualmente às pilhas (de roupas, caixas, livros etc.) do mundo real. Nela, quem chegou primeiro fica embaixo de todos os outros. Ora, para você remover uma caixa que está abaixo de várias outras, você precisa remover primeiro as de cima.

Assim é a pilha: você só pode colocar um novo elemento no topo da pilha ou retirar o elemento do topo da pilha, os demais estão “debaixo”, então não podem ser movidos.

Essa estrutura faz com que o nó mais recente seja sempre o retirado da pilha. Essa política também é chamada de **Last In First Out**, com o acrônimo **LIFO**.

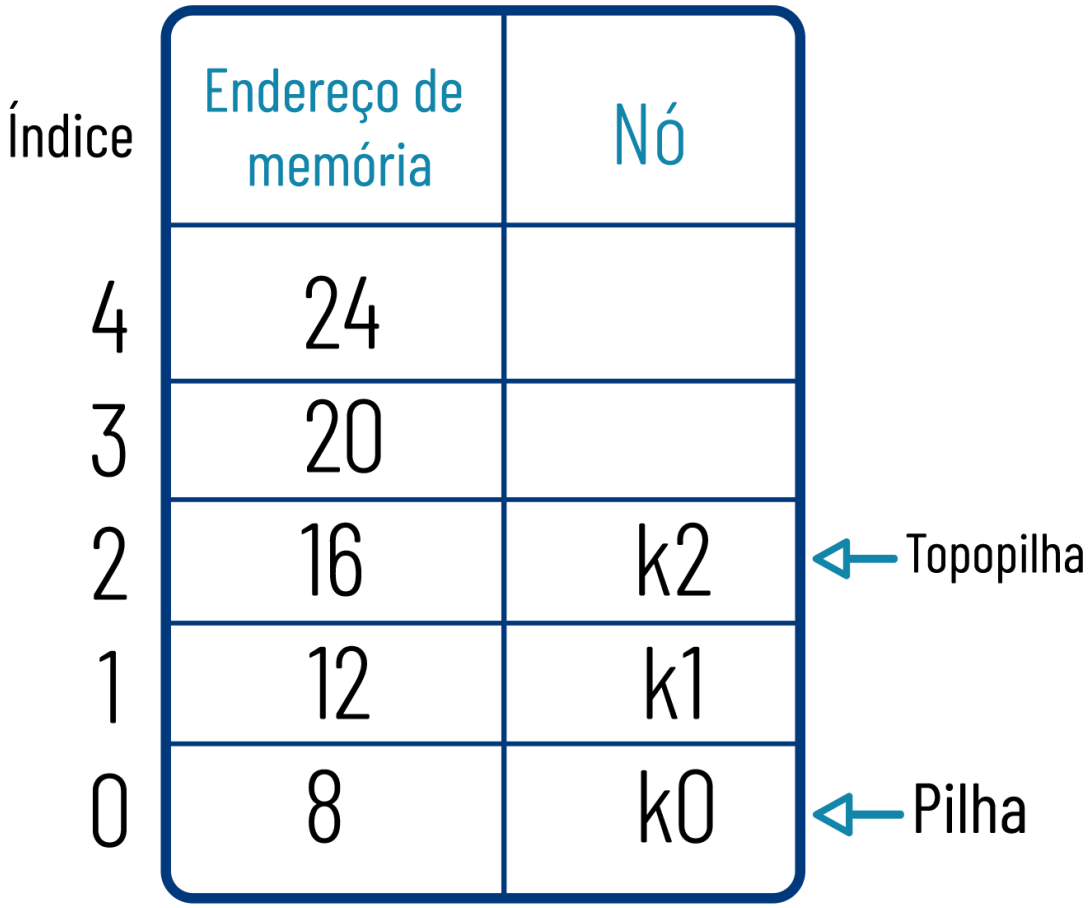
Implementação de pilha

A pilha pode ser implementada em alocação contígua ou encadeada, como as listas. De fato, a pilha é uma lista especial na qual todos os nós são inseridos e removidos apenas no início da lista.

Para uma alocação encadeada, você só precisa manter um ponteiro para o início (comumente chamado de topo) da pilha. Se a pilha estiver vazia, o TopoPilha apontará para o valor nulo. O código a seguir apresenta a classe Pilha e seu construtor.

```
class PilhaEncadeada:
        def __init__(self,topo=None):
                self.topo = topo
```

Para uma alocação contígua, você usará o espaço alocado de memória exatamente como uma pilha, enchendo da base (índice 0) para cima, e manterá uma variável TopoPilha que guarda o índice do topo da pilha. Deve apenas tomar cuidado para não armazenar mais nós do que o espaço reservado (que pode ser salvo em **MaxPilha**)



Nessa esquema, podemos ver que a variável Pilha aponta para o início do espaço alocado em memória. Pilha[0] contém o nó **k0**, e assim por diante. O topo da pilha está apontado por TopoPilha, que tem valor 2. O valor de MaxPilha é 5, ou seja, se topo pilha apontar para 4 e tentarmos inserir um nó, deve ocorrer um erro e a inserção não vai acontecer.

Operações em pilhas

Agora, você verá as operações de remoção e inserção em pilhas. A operação de busca é igual a de qualquer lista.

Inserção em pilha

A inserção na pilha deve ocorrer sempre no topo da pilha. Essa operação é comumente chamada de PUSH.

Para o caso encadeado, você pode usar o código abaixo (novoNo deve estar alocado e com o próximo apontando para o nulo):

```
def push(self,novoNo):
        novoNo.proximo=self.topo                    #insere o nó
        self.topo=novoNo                    #atualiza o topo da pilha
```

Para o caso contíguo, você possui 3 variáveis: Pilha guarda o espaço de memória, MaxPilha guarda o número máximo de elementos na pilha, TopoPilha guarda o índice do topo da pilha

```
def push(novoNo):
        global pilha
        global topoPilha
        global maxPilha
        if topoPilha== None:                                #pilha vazia
                pilha[0] = novoNo                       #insere o nó
                topoPilha=0                             #atualiza o topo da pilha
        elif (topoPilha=maxPilha-1):                    #pilha cheia
                 return -1                              # -1 → erro de pilha cheia
        else:
                topoPilha=topoPilha+1               #atualiza o topo da pilha
                 pilha[topoPilha] = novoNo          #insere o nó
        return  topoPilha                                   #saída normal
```

Complexidade da inserção na pilha

Analisando as funções de inserção na pilha, e desconsiderando os tratamentos de erro de pilha cheia, você pode verificar que a inserção consiste apenas em 2 operações de atribuição, portanto, podemos dizer que a inserção em pilhas é O(1), ou constante.

Remoção da pilha

A remoção da pilha (comumente chamado de POP) deve ocorrer sempre no topo da pilha. Para o caso encadeado, você pode usar este código:

```
def pop(self):
    if self.topo==None:
      return None                    #erro pilha vazia
    k=self.topo              #salva o nó removido
    self.topo=self.topo.proximo  #remove o nó
    return k                 #retorna o nó removido
```

Para o caso contíguo, você possui 3 variáveis: Pilha guarda o espaço de memória, MaxPilha guarda o número máximo de elementos na pilha, TopoPilha guarda o índice do topo da pilha. Experimente este código:

```
def pop():
      global pilha
      global topoPilha
      global maxPilha
      if topoPilha== None:                          # erro -pilha vazia
            return  None                      # None indica erro pilha vazia
      else:
            k =pilha[topoPilha]          # salva o nó removido
            if topoPilha==0:
                  topoPilha=None              #pilha vazia após remoção
            else:
                  topoPilha=topoPilha-1        #remove o nó
             return k                    #retorne k=o nó consumido
```

Agora, para testar suas funções de push e pop na pilha, que tal usar o código abaixo?

```
print(pilha)
for i in range(10):
            push(i)
            print(pilha)
print(push(11))
```

```
for i in range(10):
        print(pop())
print(pop())
```

Complexidade da remoção da pilha

Analisando as funções de remoção da pilha, e desconsiderando os tratamentos de erro de pilha vazia, você pode verificar que a remoção consiste apenas em 2 operações de atribuição e, possivelmente, 1 comparação, portanto, podemos dizer que a remoção em pilhas também é **O(1)**, ou constante.

Deques

Após aprender sobre as filas e pilhas, você verá a estrutura de dados que permite colocar e remover nós tanto no início como no fim da lista, chamada de deque (**Double Ended QUEue**, ou fila com duas pontas).

Conceito de deque

A estrutura de dados deque é uma generalização da fila e da pilha, essencialmente permitindo que se adicione ou remova nós do início ou do final da lista. Para evitar confusões, as operações são normalmente identificadas com qual lado da fila está sendo alterado. A remoção de um nó do início pode ser chamada de pop_front, enquanto a remoção de um nó ao final pode ser chamada de pop_back.

Implementação de deque

O módulo collections do Python já possui uma implementação de deque. Você pode acessá-la usando este comando:

```
from collections import deque
```

Veja exemplos de operações em deque, como inserção e remoção em ambas as pontas:

```
q=deque()                    #cria o deque
q.append('b')            #insere no final
q.append('c')
q.appendleft('a')
print(q)                     #imprime o deque
print(q.popleft())       #remove do inicio
print(q.pop())           #remove do final
```

Caso queira implementar seu próprio deque, ele pode ser implementado em alocação contígua ou encadeada, como as listas.

Para uma alocação encadeada, você precisa manter um ponteiro para o início e para o final do deque. Se o deque estiver vazio, ambos apontarão para o valor nulo. Usualmente, utilizamos um encadeamento duplo, que permite percorrer a lista em qualquer uma das direções e a partir de qualquer uma das pontas.

```
class No:
        def __init__(self,chave,valor):
                self.chave =chave
                        self.valor = valor
                self.proximo = None
                        self.anterior = None


class DequeEncadeada:
                def __init__(self,inicio=None):
                    self.inicio = inicio
                            self.final =       inicio
```

Para uma alocação contígua, você pode usar um vetor circular, como fizemos para a fila. As duas variáveis (InicioDeque e FinalDeque) indicam as pontas do deque e vão se deslocando conforme os nós são inseridos e removidos. Ao chegar ao fim do espaço alocado e ser incrementada, a variável faz a volta e o índice volta para zero.

De forma similar, se o valor estiver em 0 e for decrementado, passa para o maior valor possível. Normalmente, usamos a lógica modular para fazer esses incrementos/decrementos, assim como vimos na fila

```
maxDeque=10

deque=[None]*maxDeque

inicioDeque=None

finalDeque=None
```

Por fim, quando o espaço do vetor está cheio, temos um deque cheio que não poderá receber novos nós, a menos que o seu espaço seja aumentado dinamicamente.

Operações em deques

Nas operações de remoção e inserção em deques a operação de busca é igual a de qualquer lista.

Inserção em deques

A inserção no deque pode ocorrer no início ou final da estrutura. Quando ocorre no início da estrutura, seu funcionamento é idêntico à inserção na pilha, apenas

ajustando-se as variáveis **InicioDeque** e **FinalDeque**. Essa operação pode ser chamada **PUSH_front**.

Quando a inserção ocorre no final da estrutura, seu funcionamento é idêntico à inserção na fila. Essa operação também pode ser chamada **PUSH_back**.

Complexidade da inserção no deque

Considerando que as funções de inserção no deque são essencialmente as mesmas que as da fila e pilha, podemos afirmar que a inserção em deques é **O**(**1**), ou constante.

Remoção do deque

A remoção do deque pode ocorrer no início ou no final da estrutura. Quando ocorre no início, seu funcionamento é idêntico à remoção da pilha, apenas ajustando-se os ponteiros de início e final do deque (no caso encadeado) ou as variáveis **InicioDeque e FinalDeque**(no caso contíguo). Essa operação pode ser chamada **popFront** ou **popLeft**.

Quando a remoção ocorre no final da estrutura, seu funcionamento pode ser chamado **popBack** ou **popRight**. Para o caso encadeado, você pode usar o código abaixo:

```
def popBack(self):
        if self.inicio==None:                    # erro -deque vazia
            return None                          # None indica erro deque vazia
        else:
            k =self.final                        #salva o nó removido
            self.final=self.final.anterior       #remove o nó
            self.final.proximo=None    #aponta o próximo do final para λ
            return k                   #retorne k=o nó consumido
```

Para o caso contíguo, você possui 4 variáveis: **Deque** guarda o espaço de memória, **MaxDeque** guarda o número máximo de elementos do deque, **InicioDeque** e **FinalDeque** guardam o índice do início e final do deque. Você pode usar este código:

```
def popBack():
    global inicioDeque        #acesso a variavel global
    global maxDeque
    global finalDeque
    global deques
    if inicioDeque == None: #deque vazia retorna erro
        return (30*"-") + "\n Deque Vazio \n " + (30*"-")
```

```
k=deques[inicioDeque]
if finalDeque == inicioDeque:
    return  (30*"-") + "\n Deque Vazio \n " + (30*"-")
else:
  finalDeque=(finalDeque-1) % maxDeque #remove nó
return k          #retorne k= o nó consumido
```

Complexidade da remoção em deques

Analisando as funções de remoção do deque, e considerando que são similares às remoções de pilhas e filas, podemos dizer que a remoção em deques também é **O**(**1**), ou constante.