Professor Luiz Thadeu Grizendi
UniAcademia

Capítulo

3 Estrutura de dados básicas em Python

Professor Luiz Thadeu Grizendi
UniAcademia

Objetivos

- Entender os tipos de dados abstratos pilha (*stack*), fila (*queue*), deque (*deque*) e lista (*list*).
- Ser capaz de implementar os TADs pilha, fila e deque usando listas de Python.
- Compreender o desempenho das implementações estruturas lineares básicas.
- Entender expressões em notação prefixa, infixa e posfixa.
- Usar pilhas para calcular o valor de expressões posfixas.
- Usar pilhas para converter uma expressão em notação infixa para posfixa.
- Usar filas para simulações básicas.
- Ser capaz de reconhecer situações onde pilhas, filas e deques podem ser utilizadas.
- Ser capaz de implementar um tipo abstrato de dados lista como uma lista ligada usando nós e padrões de referência.
- Ser capaz de comparar o desempenho de listas ligadas implementadas com listas nativas de Python

Professor Luiz Thadeu Grizendi
UniAcademia

O que são estruturas lineares

Começaremos nosso estudo de estruturas de dados considerando quatro estruturas simples, mas muito uteis. *Pilhas*, *filas*, *deques* e *listas* são exemplos de coleções de dados cujos itens são ordenados de acordo com ordem que são inseridos ou removidos da estrutura. Uma vez que um item é inserido, fica em uma mesma posição em relação aos demais itens que foram inseridos antes ou que serão inseridos depois. Coleções como essas são frequentemente chamadas de **estruturas de dados lineares**.

Estruturas lineares podem ser consideradas como tendo duas extremidades. Às vezes essas extremidades são chamadas de *esquerda* e *direita* ou, em alguns casos, de *frente* e *traseira*. Você também pode chamá-las de *topo* e *base*. Os nomes dados às extremidades não são relevantes. O que distingue uma estrutura linear de outra é a maneira em que itens são inseridos e removidos, em particular a extremidade onde estes inserções e remoções ocorrem. Por exemplo, uma estrutura pode permitir que novos itens sejam inseridos em apenas uma das extremidades (pilhas e filas). Algumas estruturas podem permitir que itens sejam removidos de ambas as extremidades (deques).

Essas variações dão origem a algumas das estruturas de dados mais utilizadas Ciência da Computação. Elas aparecem em muitos algoritmos utilizados diariamente e podem ser usadas para resolver uma variedade de problemas importantes.

Professor Luiz Thadeu Grizendi
UniAcademia

A TAD lista

Ao longo da discussão das estruturas básicas de dados, usamos as listas do Python para implementar os tipos abstratos de dados apresentados. A lista (list) é um mecanismo de coleção de itens poderoso, porém simples, que fornece ao programador com uma ampla variedade de operações. No entanto, nem toda linguagem de programação incluem uma coleção de listas nativa. Nestes casos, a noção de lista deve ser implementada pelo programador.

Uma **lista** é uma coleção de itens em que cada item tem uma posição relativa em relação aos outros. Mais especificamente, nos referiremos este tipo de lista como uma lista desordenada. Podemos considerar que a lista possui um primeiro item, um segundo item, um terceiro item e assim por diante. Nós também podemos no referirmos ao início da lista (o primeiro item) ou ao final da lista (o último item). Por simplicidade, vamos supor que as listas não podem conter itens duplicados.

Por exemplo, a coleção dos inteiros 54, 26, 93, 17, 77 e 31 pode representar uma lista simples e desordenada de pontuações de um exame. Note que nós os escrevemos como valores delimitados por vírgula, uma maneira comum de mostrar uma estrutura lista. Claro, o Python mostraria essa lista como :

myList = [54, 26, 93, 17, 77, 31]

A classe list() do Python

O Python já disponibiliza a estrutura lista, denominada de list,

Criando uma lista

Primeira opção chamando o constructor da classe list, por exemplo:

```
myList = list()
```

Segunda opção chamando o constructor da classe list, passando como parâmetro uma lista já formada, por exemplo:

```
myList = list([1,2])
```

Os elementos da lista devem estar entre colchetes e, separadas por vírgula. A última forma é explicitar a lista, da seguinte forma.

```
myList = [1,2]
```

Obtendo o tamanho da lista.

A função **len()** retorna o comprimento de uma lista passada como parâmetro. Por exemplo:

```
len(myList)
```

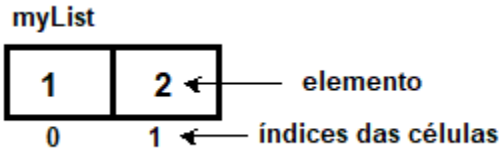
Nesse exemplo a função retornará 2

Obtendo índice de um determinado elemento da lista.

A função **index()** retorna o índice de um elemento da lista passado como parâmetro. Por exemplo:

```
myList.index(1)
```

Nesse exemplo a função retornará 0, que é o endereço do elemento "1" na lista. Fisicamente a estrutura list, é uma estrutura do tipo vetor, os elementos estão armazenados em posições adjacentes, por contiguidade física. A lista anterior poderia ser representada fisicamente por:

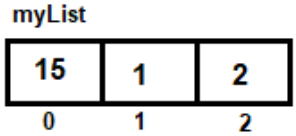


Inserindo um elemento numa lista

Podemos inserir de duas formas, inserção sempre no final da lista, utilizando a função **append(elemento)** **ou,** inserção sempre na posição informada como parâmetro, utilizando a função **insert(index, elemento).** Por exemplo:

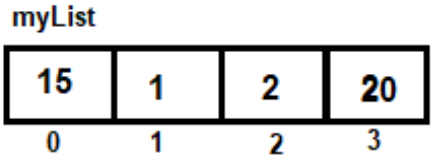
```
myList.insert(0,15)
```

Nesse exemplo a função vai inserir o elemento "15" no início da lista, posição "0". A lista ficará [15, 1, 2]. Esquematicamente, podemos exemplificar da seguinte forma



```
myList.append(20)
```

Nesse exemplo a função vai inserir o elemento “20” no final da lista. A lista ficará [15, 1, 2, 20]. Esquematicamente, podemos exemplificar da seguinte forma

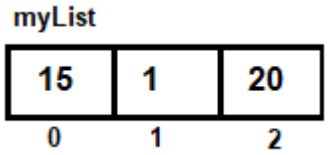


Removendo um elemento numa lista

Podemos remover um determinado elemento da lista, utilizando a função **remove(elemento)**. **O elemento deverá estar na lista**. Por exemplo, considere a lista [15, 1, 2, 20] e o comando a seguir

```
myList.remove(2)
```

Nesse exemplo a função vai excluir o elemento “2” da lista. A lista ficará [15, 1, 20]. Esquematicamente, podemos exemplificar da seguinte forma



Podemos também remover um elemento de uma determinado posição específica da lista, utilizando a função **pop()**. A sequência de comandos ilustra a utilização do método pop().

```
>>> myList
[10, 20, 30, 20, 15, 20, 1, 3]
>>> myList.pop()  # retorna o elemento que está na última posição
3
>>> myList.pop(0) # retorna o elemento que está na posição 0
10
>>> myList.pop(-1) # retorna o primeiro elemento na ordem inversa
1
>>> myList.pop(20)  # retorna o elemento que está na posição 20
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    myList.pop(20)
IndexError: pop index out of range
```

O primeiro caso, o método pop(), sem parâmetros, elimina o último elemento da lista. O método pop(0), elimina o primeiro elemento da lista, isto é, de endereço 0. O método pop(-1), elimina o primeiro elemento considerando a lista na ordem inversa, isto é, elimina-se o último elemento. Finalmente, o método pop(20), elimina o elemento da posição 20, nesse caso, retorna erro pois não existe essa posição na lista.

Verificando se um elemento existe na lista

Podemos verificar se um determinado elemento está numa lista usando o comando **in** da seguinte forma:

```
1 in myList
```

Considerando a lista [15, 1, 20], o exemplo anterior estamos verificando se o elemento "1" está na lista. Essa expressão retornará True, pois o elemento está na lista. Se a expressão fosse

```
2 in myList
```

teria como resultado false, pois o elemento "2" não faz mais parte da lista.

Concatenação e multiplicação

É possível concatenar listas por meio do operador de adição + . Também podemos utilizar o operado * e multiplicá-las por um número inteiro n, o que gerará n cópias dos seus itens.

Exemplo: Supondo que a lista myList seja a lista [15, 1, 20].

O comando

```
myList+myList
```

retornará

```
[15, 1, 20, 15, 1, 20]
```

O comando

```
myList * 3
```

retornará

```
[15, 1, 20, 15, 1, 20, 15, 1, 20]
```

Extraindo parte de uma lista

É possível extrair partes de uma determinada lista utilizando o comando de iteração, por exemplo:

Supondo que a lista myList seja a lista [15, 1, 20, 30, 50].

O comando

```
myList[:2]
```

Retornará a lista da posição inicial até a posição 2, exclusive a posição 2.

```
[15, 1]
```

O comando

```
myList[1:]
```

Retornará a lista da posição 1 até o final.

```
[1, 20, 30, 50]
```

O comando

```
myList[1:2]
```

Retornará a lista da posição 1 até a posição 2, exclusive a posição 2..

```
[1]
```

Reverso de uma lista

É possível inverter a ordem dos elementos de uma lista de várias formas descritas a seguir.

O comando

```
myList[::-1]
```

Retornará a lista reversa, isto é, as posições vão estar em ordem inversa

```
[50, 30, 20, 1, 15]
```

Uma outra forma está em aplicar o método reverse() à lista que se queira obter o reverso, por exemplo

```
myList = [70, 60, 50, 66, 35, 30, 10]
myList.reverse()
```

O valor de myList é alterado para a seguinte lista

```
[10, 30, 35, 66, 50, 60, 70]
```

É importante frisar que myList.reverse() transforma a lista original na lista em reverso, mas não deve ser usada na forma de atribuição.

Iteração em uma lista

Varrer os elementos de uma lista de forma iterativa é feita usando a função interna iterator (__iter__()) do Python. Por exemplo, imprimir todos os elementos da minha lista, basta fazer:

```
for element in myList:
    print(element)
```

Se desejarmos, imprimir, a sequência inversa, podemos utilizar o comando reversed(), que retornará um iterator do reverso da lista. Neste caso podemos escrever

```
for element in reversed(myList):
    print(element)
```

Outra forma de imprimir, a sequência inversa, basta aplicar a ideia vista anteriormente, da seguinte forma:

```
for element in myList[::-1]:
    print(element)
```

Cópia de uma lista

É possível copiar uma lista e armazená-la em outra lista. A sequência de comandos ilustra a obtenção da cópia de uma lista

```
>>> myList = [1,2,3]
>>> otherList = myList.copy()
>>> otherList
[1, 2, 3]
```

Limpar os elementos de uma lista

É possível limpar os elementos de uma lista. O método responsável em limpar uma lista é o método clear() aplicado na lista que se deseja limpar. A sequência de comandos ilustra a limpeza de uma lista

```
>>> myList.clear()
>>> myList
[]
```

Verificando a quantidade de vezes em que um elemento faz parte de uma lista

Para verificarmos a quantidade de vezes em que um determinado elemento ocorre na lista devemos aplicar o método count(). A sequência de comandos ilustra a obtenção do número de vezes em que um determinado elemento se repete em uma lista

```
>>> myList = [10, 20, 30, 20, 15, 20 ]
>>> myList.count(20)
3
```

Acrescentar uma outra lista no final de uma lista

Para anexarmos uma lista dada em uma outra lista já existente devemos utilizar o método extend. A sequência de comandos ilustra a extensão de uma lista, anexando uma outra lista

```
>>> myList.extend([1,3])
>>> myList
[10, 20, 30, 20, 15, 20, 1, 3]
```

Ordenando os elementos de uma lista

Para ordenarmos uma lista dada devemos utilizar o método sort.. A sequência de comandos ilustra a ordenação de uma lista

```
>>> myList
[20, 30, 20, 15, 20]
>>> myList.sort()
>>> myList
[15, 20, 20, 20, 30]
```

Podemos também ordenar uma lista sem alterar a lista original. Para isso devemos utilizar o método ordered(iterable) e passar a lista que devemos ordenar como parâmetro no lugar do iterable.

```
>>> myList = [20, 30, 10, 15, 50]
>>> otherList = sorted(myList)
>>> otherList
[10, 15, 20, 30, 50]
```

Finalmente, podemos acrescentar a opção reverse = True, para ordenarmos em ordem inversa ou reverse = False, ordenação na ordem sequencial. Se omitirmos a opção reverse, é o mesmo que reverse = False.

```
>>> otherList = sorted(myList, reverse = True)
>>> otherList
[50, 30, 20, 15, 10]
```

Professor Luiz Thadeu Grizendi
UniAcademia

Exercícios

1. Considere a rotina a seguir que utiliza as rotinas append(e), que serve para ensirer um elemento no final da lista, insert(0,e), que serve para inserir um elemento na posição 0 da lista e pop() para remover e retornar o elemento do final da fila.

```
l.clear()
for i in range(5):
    l.append(i)
    l.insert(0,2*i)
    print(l.pop())
```

   Dar a sequência que será impressa.

2. Complete a tabela a seguir considerando o TAD list (Lista) já inicializada com os valores 10,33,5,44.

OPERAÇÃO	SAÍDA	CONTEÚDO DO PILHA
len(l)	4	(10,33,5,44)
33 in l		
l.append(5)		
l.pop()		
l.insert(1,20)		
l.remove(10)		

   Onde:

append(e)	Insere o elemento “e”, no final da lista
insert(index,e)	Insere o elemento “e”, na posição index da lista
pop()	Remove o elemento do final da lista
remove(e)	Remove o elemento informado da lista
e in l	Retorna **True** se o elemento “e” existe na lista e **False** caso contrário..
len(l)	Retorna o tamanho da lista.

3. A letra significa append e um asterisco significa pop na seguinte seqüência. Dar a seqüência de valores devolvidos pelas operações pop() quando esta seqüência de operações é realizada em uma fila inicialmente vazia.

E A S * Y * Q U E *** S T *** I O * N ***

Professor Luiz Thadeu Grizendi
UniAcademia

Pilha

A **pilha** é uma coleção de objetos cujo princípio de inserção e remoção é o ***last-in-first-out*** (LIFO). Este princípio consiste no procedimento de que o último elemento a entrar na estrutura (***last-in***), é o primeiro a sair (***first-out***).

É interessante frisar que o nome da estrutura **pilha** veio de uma metáfora de uma pilha qualquer de objetos. Como exemplo de estruturas que utilizam o princípio LIFO de inserção e remoção de seus objetos, podem-se destacar a pilha de livros e a pilha de pratos, ilustrados na Figura 5.1.

Figura 5.1 – Pilha de Livros e a pilha de Pratos.

As operações básicas na estrutura **pilha** são "*empilhar*" (**push)**, e "*desempilhar*" (**pop**). A figura 5.2 ilustra a visão esquemática da estrutura pilha onde se pode observar que a operação **push** coloca o elemento no topo da **pilha** e a operação **pop** retira o elemento do topo da **pilha**.

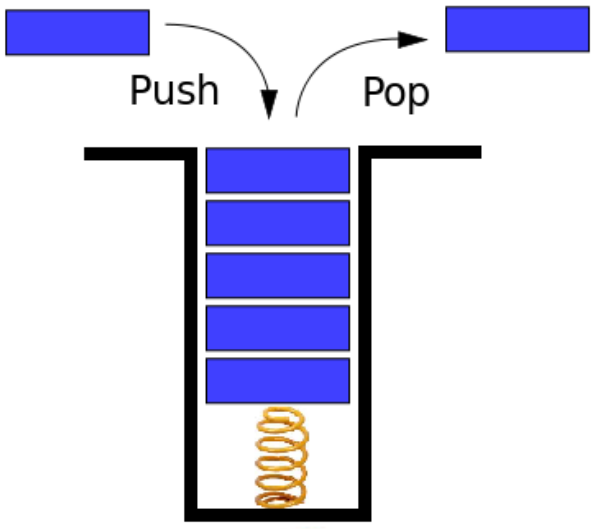


Figura 5.2– Visão esquemática da estrutura pilha, com as operações push (empilhar) e pop (desempilhar).

A ordem que eles são removidos é exatamente a inversa da ordem em que foram colocados. As pilhas são fundamentalmente importantes, pois elas podem ser usadas para reverter a ordem dos itens. A ordem de inserção é a inversa da ordem de remoção. A Figura 5.3 ilustra a pilha de itens como ela foi criada e, novamente, como os itens são removidos. Observa-se que a ordem de inserção é diferente da ordem de remoção. O primeiro inserido será o último removido.

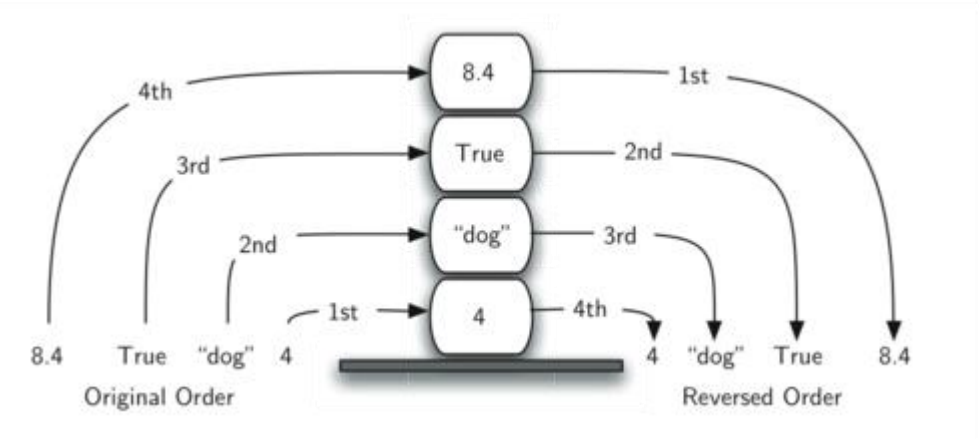


Figura 5.3 – Visão esquemática de inserção e remoção na pilha.

A figura 5.3 ilustra as operações de empilhamento e desempilhamento em uma estrutura **pilha.** Pode-se observar que o último elemento a ser inserido, marcado com uma seta, denominado de "*topo*", será o primeiro elemento a ser removido.

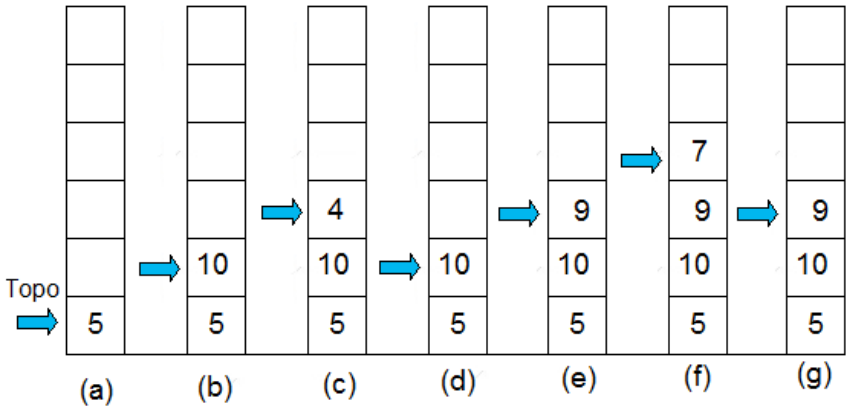


Figura 5.3 – Sequência de inserção e remoção numa Pilha: (a) Empilha 5, (b) Empilha 10, (c) Empilha 4, (d) Desempilha, (e) Empilha 9, (f) Empilha 7, (g) Desempilha.

Professor Luiz Thadeu Grizendi
UniAcademia

A Pilha como Tipo Abstrato de Dados

A **pilha** além de ser a estrutura mais simples é uma das mais utilizadas. Por exemplo, num editor de texto, a operação ***undo***, reverte às operações mais recentes executadas. Cada operação é armazenada numa **pilha** e, se o usuário desejar, ele pode reverter estas operações desempilhando cada operação. Outro exemplo que se pode destacar está na utilização de um *browser* qualquer de navegação no ambiente *web*. Neste programa o usuário pode ver o histórico de todas as suas visitas utilizando os botões ***back*** que implementa a função desempilha. Os botões undo e redo de um editor de texto, e os botões backward e o forward de um browser, são exemplos de aplicações da estrutura pilha. A figura 5.4 ilustra os botões undo , redo, back e forward.

(a) (b)

Figura 5.4 – (a) botões undo e redo, (b) botões backward e o forward.

Existem alguns métodos básicos para o funcionamento de uma pilha, os métodos **push(e)** para empilha**r,** o método **pop()** para desempilhar**,** e o método **peek()** para consultar o elemento do topo da **pilha**, descritos a seguir.

s.push(e):	Adiciona o elemento e, no topo da **pilha s**.
s.pop():	Remove o elemento que está no topo da **pilha s**. Este método retorna o elemento removido. Caso a pilha esteja vazia este método chama a diretiva de tratamento de erro.
s.peek()	Retorna o elemento que está no topo da **pilha s** sem excluí-lo da estrutura. Caso a pilha esteja vazia este método chama a diretiva de tratamento de erro.
s.top()	O memo que s.peek()
s.clear()	Inicializa a **pilha s** como vazia.
s.isEmpty()	Retorna True se a pilha s estiver vazia e, False caso contrário.
s.extend(iterator)	Adiciona todos os elementos do iterator na pilha s.

Existem alguns métodos auxiliares para o funcionamento de uma pilha s. A seguir descrevem-se esses métodos.

__contains(e)__:	Retorna True se o elemento **e** estiver na **pilha s** e, False caso contrário. Utilizado na sintaxe **e** in **s**
__repr(s)__:	Retorna a representação da pilha s. Utilizado na sintaxe repr(**s)**
__str(s)__	Retorna um string com a representação da pilha s. Utilizado na sintaxe str(**s)**
__len(s)__	Retorna o tamanho da pilha **s**. Utilizado na sintaxe len(**s)**
__iter(s)__	Retorna a pilha **s** na forma de um *iterator*. Utilizado na sintaxe iter(**s)** e for element in **s**
__next(s)__	Retorna o próximo elemento da pilha em uma iteração. Utilizado indiretamente na sintaxe iter(**s)** ou diretamente next(**s)**

A tabela *5.1* exemplifica estes métodos sendo executados e, para cada operação, pode-se observar o parâmetro de retorno de cada método e, a variação do conteúdo da **pilha**. O primeiro elemento da sequência de conteúdo da pilha corresponde ao topo da **pilha**.

Tabela 5.1 – Sequência de operações numa Pilha.

OPERAÇÃO	SAÍDA	CONTEÚDO DA PILHA
s.push(5)	-	5
s.push(10)	-	10 5
s.push(4)	-	4 10 5
s.peek()	4	4 10 5
len(s)	3	4 10 5
s.isEmpty	False	4 10 5
s.pop()	4	10 5
push(9)	-	9 10 5
10 in s	True	9 10 5
s.clear()	-	
s.extend([10,5])		5 10

Implementação da estrutura Pilha

Uma forma simples de implementação da estrutura pilha em Python, é considerar a pilha como uma estrutura lista(list) e, gerenciarmos a inserção e remoção dos elementos sempre no mesmo lado, isto é, a inserção e remoção é feita sempre no final. Também poderíamos optar em inserir e remover no início.

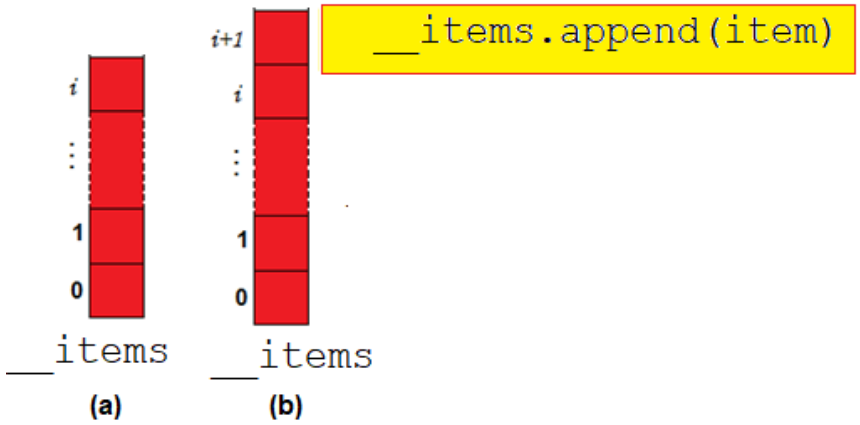
O Fragmento de código 5.1 ilustra a implementação do constructor da classe pilha. O parâmetro iterator é opcional e contém um iterator com itens a serem incluídos na pilha. Esse parâmetro é inicializado com None para o caso em que não seja informado.

```
def __init__(self,iterator=None):
    self.__items = []
    if iterator:
        for item in iterator:
            self.push(item)
```

Fragmento de Código 5.1. Implementação em Python do constructor da classe Stack.

Inserção (push)

O método que implementa a inserção na pilha é o método push(element). Genericamente, pode-se descrever este método, como sendo: inserir o elemento no final da lista e. A Figura 5.5 ilustra este procedimento. Observa-se o uso do comando append da estrutura list correspondendo à inserção do item no final da pilha.



(a) (b)

Figura 5.5. Visão esquemática da inserção em uma pilha. A letra (a) ilustra a pilha antes da inserção e a letra (b) após a inserção.

O fragmento de código 5.2 ilustra a implementação do procedimento push(item).

```
def push(self, item):
    self.__items.append(item)
```

Fragmento de Código 5.2. Implementação em Python do método push(item) da classe pilha.

Remoção (pop)

O método que implementa a remoção na pilha é o método pop(). Genericamente, pode-se descrever este método, como sendo: remover o elemento que está no topo da pilha. Como a inserção é feita sempre no final pois foi utilizado o comando append, a remoção tem que ser no mesmo lado, isto é remover no final, utilizando o comando pop(). A figura 5.6 ilustra este procedimento. Observa-se o uso do comando pop da estrutura list correspondendo à remoção do item no final da pilha.

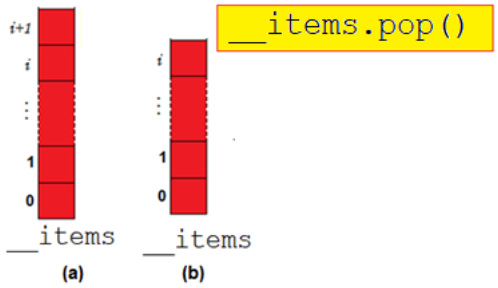


Figura 5.6. Visão esquemática da remoção em uma pilha, método pop() (desempilha). A letra (a) ilustra a pilha antes da remoção e, a letra (b) após a remoção.

O fragmento de código 5.3 ilustra a implementação do procedimento pop(). Pode-se observar, o teste verificando se a pilha está ou não vazia. No caso da pilha vazia é ativada a diretiva de erro, e, caso contrário é feita a exclusão do elemento no final da lista.

```
def pop(self):
    if len(self) == 0:
        raise IndexError('pop from empty stack')
    return self.__items.pop()
```

Fragmento de Código 5.3. Implementação em Python do método pop da classe pilha.

Consulta (peek)

Para consultarmos o elemento que está no topo da pilha, sem removê-lo utilizaremos a função peek(), que retorna, sem excluir, o elemento que está no topo da pilha. Como a inserção é sempre feita no final da lista, o topo, será o último elemento do vetor.

O fragmento de código 5.4 ilustra a implementação do procedimento peek(). Pode-se observar, o teste verificando se a pilha está ou não vazia. No caso da pilha vazia é ativada a diretiva de erro, e, caso contrário retorna-se o elemento no final da lista.

```
def peek(self):
    if len(self) == 0:
         raise IndexError('top from empty stack')
    return self.__items[-1]
```

Fragmento de Código 5.4. Implementação em Python do método peek da classe pilha.

Professor Luiz Thadeu Grizendi
UniAcademia

Parênteses Balanceados

Agora voltamos nossa atenção para o uso de pilhas para resolver verdadeiros problemas de ciência da computação. Você certamente deve já deve ter escrito expressões aritméticas, como

$$(5+6)*(7+8)/(4+3)$$

onde parênteses são usados para determinar a ordem em que as operações são executadas.

Em ambos os exemplos, os parênteses devem aparecer de maneira *balanceada*. **Parênteses balanceados** significa que na expressão para cada abre parênteses há um correspondente fecha parênteses e os pares de parênteses estão aninhados. Considere as seguintes strings de parênteses corretamente balanceadas:

(()()()())

((((()))

(()((())()))

Compare essas strings com as seguintes, que não são balanceadas:

(((((((()

())))

(()()(()

A capacidade de diferenciar entre sequências de parênteses corretamente balanceadas daquelas que estão desbalanceadas é um componente importante no reconhecimento estruturas em muitas linguagens de programação.

O desafio então é escrever um algoritmo que leia uma string de parênteses da esquerda para a direita e decida se os parênteses estão balanceados. Para resolver este problema, precisamos fazer uma observação importante. Ao examinar da esquerda para a direita os símbolos na string, cada fecha parêntese deve ser associado ao abre parêntese que foi examinado mais recentemente e ainda não foi associado a um fecha parêntese. A Figura 5.7 ilustra esse procedimento. Além disso, o primeiro abre parêntese examinado pode ter que esperar até o último símbolo da string para encontrar o seu fecha parêntese. Fecha parênteses são associados a abre parênteses na ordem inversa que foram examinados; eles são emparelhados de “dentro para fora”. Este é um indício de que pilhas podem ser usadas para resolver problema.

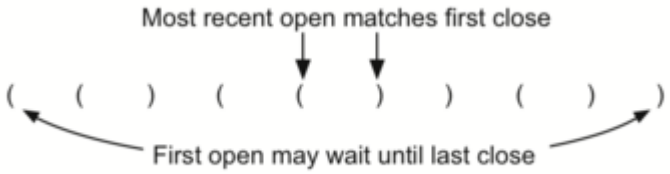


Figura 5.5. Emparelhamento de Parênteses.

Uma vez que você concorda que uma pilha é a estrutura de dados apropriada para mantermos os parênteses, a descrição do algoritmo é clara. Começando com uma pilha vazia, processe os parênteses na string da esquerda para a direita. Se um símbolo é um abre parêntese, insira-o (push()) na pilha para indicar que o fecha parêntese correspondente precisa aparecer mais tarde. Se, por outro lado, um símbolo é um fecha parêntese, remova (pop()) um abre da pilha. Contanto que seja possível realizarmos um pop() na pilha para corresponder a cada fecha parêntese encontrado, os parênteses estarão balanceados. Se em qualquer momento não houver um abre parêntese na pilha para corresponder a um fecha parêntese, a string não está balanceada. No final da string, quando todos os símbolos tiverem sido processados, a pilha deve estar vazia. O Fragmento de código 5.5 implementa esse procediemnto.

```
from Stack import Stack

def parChecker(symbolString):
        s = Stack()
        balanced = True
        index = 0
        while index < len(symbolString) and balanced:
            symbol = symbolString[index]
            if symbol == "(":
                s.push(symbol)
            else:
                if s.isEmpty():
                    balanced = False
                else:
                    s.pop()

            index = index + 1

        if balanced and s.isEmpty():
            return True
        else:
            return False
```

Fragmento de Código 5.5. Implementação em Python da função parChecker.

A função, parChecker(), supõe que uma classe Stack está disponível e retorna um resultado booleano (bool) após decidir se a string está ou não balanceada. Note que a variável booleana balanced é inicializado com True porque inicialmente não há razão para supor o contrário. Se o símbolo atual é (, então ele é inserido na pilha (linhas 9-10). Note também na linha 15 que pop() simplesmente remove um símbolo da pilha. O valor retornado não é usado, pois sabemos que é um abre parêntese visto anteriormente. No final (linhas 19 a 22), desde que os fecha parênteses tenha encontrado os correspondentes abre e a pilha tenha sido completamente limpa, a string representa uma sequência de parênteses balanceada.

Considere a sequência de comandos a seguir:

```
print('parChecker')
print(parChecker('((()))'))
print(parChecker('(()'))
```

Ao executarmos a sequência anterior o programa retornaria

```
parChecker
True
False
```

Professor Luiz Thadeu Grizendi
UniAcademia

Símbolos Balanceados

O problema dos parênteses balanceados mostrado anteriormente é um caso particular de uma situação mais geral que surge em muitas linguagens de programação. O problema geral de balancear e aninhar tipos diferentes símbolos de abertura e fechamento ocorrem com frequência.

Por exemplo, em Python os símbolos colchetes, [e], são usados para listas; chaves, { e }, são usados para dicionários; e parênteses, (e), são usados para tuplas e expressões aritméticas. É possível misturar símbolos desde que mantenham sua própria relação de abertura e fechamento.

Strings de símbolos tais como

```
{{([][])}()}

[[{{(())}}]]

[][][](){}
```

são devidamente balanceadas já que não só cada símbolo de abertura corresponde a um símbolo de fechamento, mas também os tipos de símbolos correspondem.

Compare essas strings com as seguintes strings que não são balanceadas:

```
([)]

(((()]))

[{()]
```

O verificador de parênteses da seção anterior pode ser facilmente estendido para lidar com esses novos tipos de símbolos. Lembre-se de que cada símbolo de abertura é simplesmente inserido (push()) na pilha para aguardar o correspondente símbolo de fechamento que deve aparecer mais tarde na sequência. Quando um símbolo de fechamento aparece, a única diferença é que devemos verificar se o símbolo de abertura que está no topo da tinha é do tipo apropriado. Se os dois símbolos não se casam, a string não está balanceada. Mais uma vez, se a string inteira é examinada e nenhum símbolo é deixado na pilha, a string está corretamente balanceada.

O fragmento de código 5.6 implementa a ideia descrita anteriormente. A única mudança aparece na linha 16 onde nós chamamos uma função auxiliar, matches(), para ajudar com a correspondência de símbolos. Cada símbolo que é removido da pilha deve ser examinado para nos assegurarmos que corresponde ao símbolo de fechamento atual. Se ocorre uma incompatibilidade, a variável booleana balanced recebe o valor False.

```
def symbolChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                       balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

def matches(open,close):
    opens = "([{"
    closers = ")]}"
    return opens.index(open) == closers.index(close)
```

Fragmento de Código 5.6. Implementação em Python das funções SimbolChecker e matches.

Considere a sequência de comandos a seguir:

```
print('symbolChecker')
print(symbolChecker('{{([][])}()}'))
print(symbolChecker('[{()]'))
```

Ao executarmos a sequência anterior o programa retornaria

```
symbolChecker
True
False
```

Professor Luiz Thadeu Grizendi
UniAcademia

Conversão decimal binário

O Fragmento de código 5.7 ilustra a conversão de um número inteiro decimal em binário.

```
def decToBin(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ""
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString
```

Fragmento de Código 5.5. Implementação em Python da função decToBin.

Exercícios

4. Considere a rotina a seguir que utiliza as rotinas push(e), que serve para empilhar um elemento na pilha, pop() para desempilhar e peek() para retornar o elemento do topo da pilha.

```
S.clear();
for i in range(1,5):
   S.push(i)
   S.push(2*i)
   S.pop()
   print(S.peek()+',')
```

   Dar a sequência que será impressa.

5. Complete a tabela a seguir considerando o TAD Stack (PILHA) já inicializada com os valores 10,33,5,44. Considere o topo da pilha o elemento 10 e assim sucessivamente.

OPERAÇÃO	SAÍDA	CONTEÚDO DO PILHA
size()	4	(10,33,5,44)
contains(33)		
push(5)		
pop()		
peek()		
size()		

   Onde:

   push(e): Insere o elemento “e”, na Pilha.
   pop() Remove o topo da Pilha.
   peek() Retorna o elemento do topo da Pilha.
   contains(e): Retorna **true** se o elemento “e” existe na Pilha e **false** caso contrário..
   size() Retorna o tamanho da Pilha.

6. A letra significa empilhar e um asterisco significa desempilhar na seguinte seqüência. Dar a seqüência de valores devolvidos pelas operações desempilar (pop()) quando esta seqüência de operações é realizada em uma pilha inicialmente vazia.

E A S * Y * Q U E *** S T *** I O * N ***

7. Suponha que uma sequência misturada de operações empilha e desempilha são executadas. Os empilhamentos são números inteiros de 0 a 9 em ordem; A seguir descrevem-se sequências de valores de retorno do desempilhamentos. Quais das seguintes sequências não poderia ocorrer?

(a)	4	3	2	1	0	9	8	7	6	5
(b)	4	5	6	7	8	9	0	2	3	1
(c)	2	4	5	6	7	8	9	0	1	3
(d)	4	3	2	1	0	5	6	7	8	9

Gabarito: A

8. Assume que x, y, z sejam variáveis inteiras e que a pilha é do tipo inteiro, dê o estado de saída de cada programa.

a)

```
x = 3
y = 5
z = 2
s.push(x)
s.push(4)
z=s.pop()
s.push(y)
s.push(3)
s.push(z)
x=s.pop()
s.push(2)
s.push(x)
while not s.isEmpty():
      x=s.pop()
      print(str(x)+',')
```

b)

```
y = 1
s.push(5)
s.push(7)
x=s.pop()
x += y
s.push(x)
s.push(y)
s.push(2)
y=s.pop()
x=s.pop()
while not s.isEmpty():
     y=s.pop()
     print(str(y)+',')
print("x = " + str(x))
print("y = " + str(y))
```

9. Mostre o que será impresso em cada sequência de fragmento de programa, onde *s1* e *s2* são pilhas de inteiros e *j, k e m* são variáveis inteiras.

a)

```
for j in range(0,9):
   s1.push(j)
while not s1.isEmpty():
   j=s1.pop()
   if (j % 2 == 0):
      s2.push(j)

while not s2.isEmpty():
   j=s2.pop()
   print(str(j)+',')
```

b)

```
j = 1
s1.clear()
s2.clear()
while (j * j < 50):
    k = j * j
    s1.push(k)
    j += 1

for j in range(1,6):
    k=s1.pop()
    s2.push(k)

j=s1.pop()
for k  in range(1,j+1):
    m=s2.pop()
    s1.push(m)

while not s1.isEmpty():
    j=s1.pop()
    print(str(j)+',')
```

c)

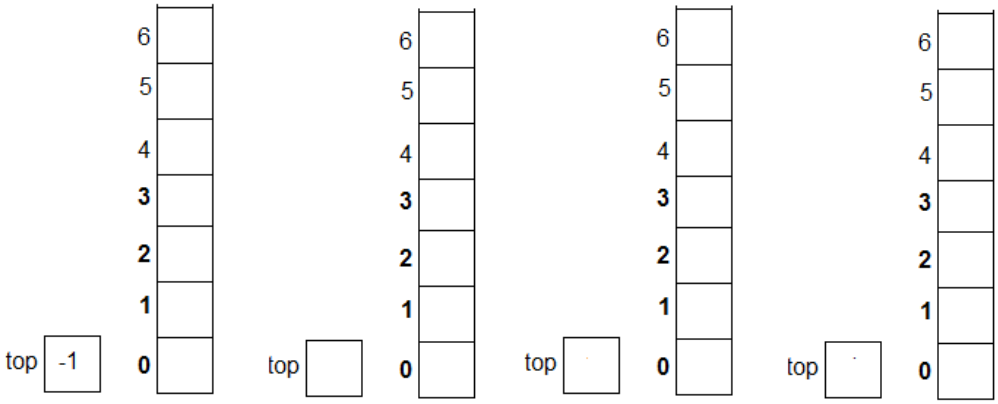
```
S.clear()
x = 3
y = 5
z = y % 2
S.push(x)
S.push(x)
x=S.pop()
S.push(x + z)
S.push(y)
while not S.isEmpty():
    y=S.pop()
    print(str(y)+',')
print(z + y + x)
```

d)

```
S.clear()
T.clear()
for index in range(0,20):
     S.push(index)
while not S.isEmpty():
   x=S.pop()
   if (x % 4 == 0):
      T.push(x)

while not T.isEmpty():
   x=T.pop ()
   print(str(x)+',')
```

10. Considere a lista pilha por contiguidade física inicializada a seguir;



Efetue as seguintes operações na ordem pré-determinada

```
push('B'); push('E'); push(pop()), push('F'), pop()
```

Professor Luiz Thadeu Grizendi
UniAcademia

Fila

A **fila** é uma coleção de objetos cujo princípio de inserção e remoção é o ***first-in- first-out*** (FIFO). Este princípio consiste no procedimento de que o primeiro elemento a entrar na estrutura (***first-in***) é o primeiro a sair (***first-out***). É interessante frisar que o nome da estrutura **fila** veio de uma metáfora da **fila** de pessoas para serem atendidas durante um procedimento qualquer, como por exemplo, a fila de atendimento de um banco.

Como exemplo de estruturas que utilizam o princípio FIFO de inserção e remoção de seus objetos, podem-se destacar o dispensador de copos e fila de pessoas para atendimento em caixa de um banco, ilustrados na Figura 5.1.

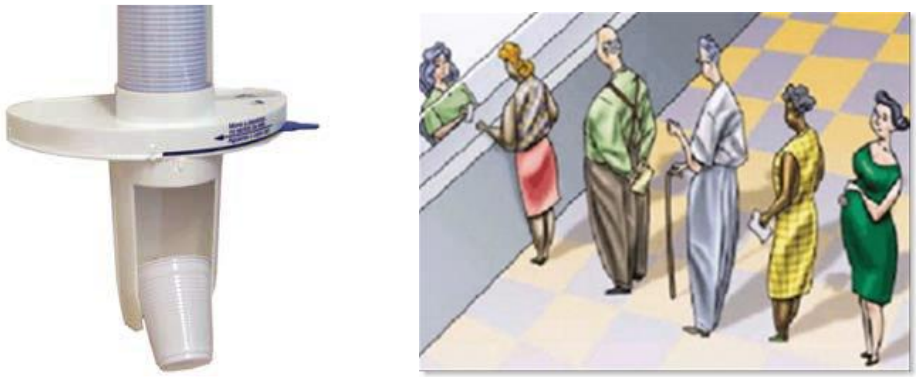


Figura 5.1 – Exemplos de filas: o dispensador de copos e a fila de uma caixa em um banco.

As operações básicas na estrutura **fila** são inserir um elemento na fila (enqueue), e remover o elemento da fila (dequeue). A figura 5.2 ilustra a visão esquemática da estrutura fila onde pode-se observar a operação enqueue para inserir um elemento no final da **fila** e a operação dequeue para remover o elemento do início da **fila**.

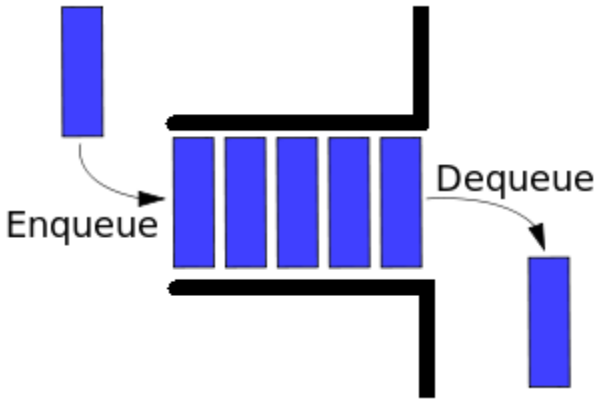


Figura 5.2 – Visão esquemática da estrutura fila, com as operações inserir na fila (enqueue) e remover da fila (dequeue).

O exemplo mais simples de uma fila é a fila típica que todos nós participarmos de tempos em tempos. Nós esperamos em uma fila do cinema, esperamos na fila do supermercado, e esperamos na fila da lanchonete (para pegarmos uma bandeja da pilha de bandejas). Filas bem-comportadas são muito restritivas em que elas têm apenas uma maneira de entrarmos e apenas uma maneira de sairmos. Não é permitido furar a fila e sair antes de você ter esperado o tempo necessário para chegar ao início. A Figura 5.3 ilustra uma fila simples de itens de dados.

Figura 5.3 – Visão esquemática da estrutura fila, com os elementos que estão na frente (front) e na cauda (rear).

A Fila como Tipo Abstrato de Dados

A **fila** é uma estrutura muito utilizada em computação como por exemplo:

- Gerência dos arquivos para impressão
- Gerência dos processos nas requisições do sistema operacional
- Buffer para gravações de mídia
- Processos de comunicação de redes

Existem dois métodos básicos para o funcionamento de uma **fila**, o método **enqueue(element)** e o método **dequeue()** descritos a seguir:

q.enqueue(e):	Adiciona o elemento e, na cauda da fila q.
q.dequeue():	Remove o elemento que está na frente da fila q. Este método retorna o elemento removido. Caso a fila esteja vazia este método chama a diretiva de tratamento de erro.
q.peek()	Retorna o elemento que está na frente da fila q sem excluí-lo da estrutura. Caso a fila esteja vazia este método chama a diretiva de tratamento de erro.
q.top()	O memo que s.peek()
q.clear()	Inicializa a fila q como vazia.
q.isEmpty()	Retorna True se a fila q estiver vazia e, False caso contrário.
q.extend(iterator)	Adiciona todos os elementos do iterator na fila q.

Existem alguns métodos auxiliares para o funcionamento de uma pilha s. A seguir descrevem-se esses métodos.

__contains(e)__	Retorna True se o elemento **e** estiver na fila q, False caso contrário. Utilizado na sintaxe **e** in **q**
__repr(q)__	Retorna a representação da fila q. Utilizado na sintaxe repr(q**)**
__str(q)__	Retorna um string com a representação da fila q. Utilizado na sintaxe str(q**)**
__len(q)__	Retorna o tamanho da fila q. Utilizado na sintaxe len(q**)**
__iter(q)__	Retorna a fila **s** na forma de um *iterator*. Utilizado na sintaxe iter(q**)** e for element in **q**

A tabela *5.1* exemplifica estes métodos sendo executados e, para cada operação, pode-se observar o retorno de cada método e a variação do conteúdo da **fila.**

Tabela 5.1 – Sequência de inserção e remoção numa fila.

OPERAÇÃO	SAÍDA	CONTEÚDO DA FILA
q.enqueue(5)	-	(5)
q.enqueue(10)	-	(5,10)
q.enqueue(4)	-	(5,10,4)
len(q**)**	3	(5,10,4)
q.dequeue()	5	(10,4)
q.enqueue(9)	-	(10,4,9)
q.enqueue(7)	-	(10,4,9,7)
q.peek()	10	(10,4,9,7)
q.dequeue()	10	(4,9,7)
10 in q	False	(4,9,7)
q.dequeue()	4	(9,7)
q.dequeue()	9	(7)
q.dequeue()	7	()
q.dequeue()	**Error**	()
q.isEmpty	True	-
q.extend([10,20])	-	(10,20)
q.clear()	-	()

Implementação da estrutura Fila

Uma forma simples de implementação da estrutura fila em Python, é considerar a fila como uma estrutura lista(list) e, gerenciarmos a inserção e remoção dos elementos sempre em lados opostos, isto é, inserirmos no final, e removemos no início.

O Fragmento de código 5.1 ilustra a implementação do constructor da classe fila. O parâmetro iterator é opcional e contém um iterator com itens a serem incluídos na fila. Esse parâmetro é inicializado com None para o caso em que não seja informado.

```
def __init__(self,iterator=None):
    self.__items = []
    if iterator:
        for item in iterator:
            self.enqueue(item)
```

Fragmento de Código 5.1. Implementação em Python do constructor da classe Queue.

Inserção (emqueue/peek)

O método que implementa a inserção na pilha é o método enqueue(element). Genericamente, pode-se descrever este método, como sendo: inserir o elemento no final da lista. A figura 5.5 ilustra este procedimento. Observa-se o uso do comando append da estrutura list correspondendo à inserção do item no final da pilha.

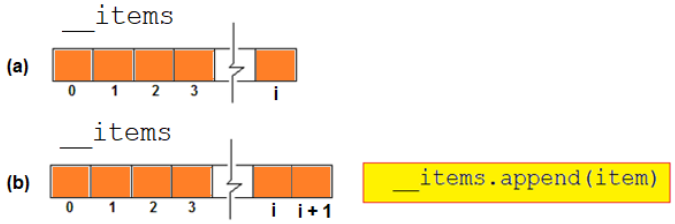


Figura 5.5. Visão esquemática da inserção em uma fila. A letra (a) ilustra a fila antes da inserção e a letra (b) após a inserção.

O fragmento de código 5.2 ilustra a implementação do procedimento enqueue(item).

```
def enqueue(self, item):
    self.__items.append(item)
```

Fragmento de Código 5.2. Implementação em Python do método enqueue(item) da classe fila.

Remoção (dequeue/poll)

O método que implementa a remoção na fila é o método dequeue(). Genericamente, pode-se descrever este método, como sendo: remover o elemento que está na frente da fila. Como a inserção é feita sempre no final pois foi utilizado o comando append, a remoção tem que ser no lado oposto, isto é, remover no início, utilizando o comando pop(0). A figura 5.6 ilustra este procedimento. Observa-se o uso do comando pop da estrutura list correspondendo à remoção do item no início da fila.

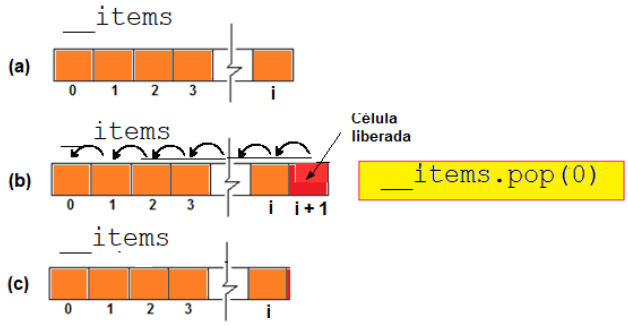


Figura 5.6. Visão esquemática da remoção em uma fila, método dequeue(). A letra (a) ilustra a fila antes da remoção e, a letra (b) durante a exclusão e, a letra (c) após a remoção.

O fragmento de código 5.3 ilustra a implementação do procedimento dequeue(). Pode-se observar, o teste verificando se a fila está ou não vazia. No caso da fila vazia é ativada a diretiva de erro, e, caso contrário é feita a exclusão do elemento no final início.

```
def dequeue(self):
    if len(self) == 0:
        raise IndexError('dequeue/poll from empty queue')
    return self.__items.pop(0)
```

Fragmento de Código 5.3. Implementação em Python do método dequeue da classefila.

Consulta (peek)

Para consultarmos o elemento que está no frente da fila, sem removê-lo utilizaremos a função peek(), que retorna, sem excluir, o elemento que está na frente da fila. Como a inserção é sempre feita no final, o elemento da frente, será o primeiro elemento do vetor.

O fragmento de código 5.4 ilustra a implementação do procedimento peek(). Pode-se observar, o teste verificando se a fila está ou não vazia. No caso da fila vazia é ativada a diretiva de erro, e, caso contrário retorna-se o elemento no início do elemento __items.

```
def peek(self):
     if len(self) == 0:
         raise IndexError('peek/front from empty queue')
     return self.__items[0]
```

Fragmento de Código 5.4. Implementação em Python do método peek da classe fila.

Professor Luiz Thadeu Grizendi
UniAcademia

Simulação: Batata Quente

Uma das aplicações típicas para mostrar uma fila em ação é simular uma situação real que requer que os dados sejam gerenciados em de um maneira FIFO. Para começar, vamos considerar o jogo infantil Batata Quente (*Hot Potato*). Nesse jogo (ver Figura 2) crianças se alinham em um círculo e passam um item (a batata quente) de vizinho para vizinho o mais rápido que puderem. A um certo ponto no jogo, a ação é interrompida e a criança que tem o item (a batata) é removida do círculo. O jogo continua até que resta apenas um criança.

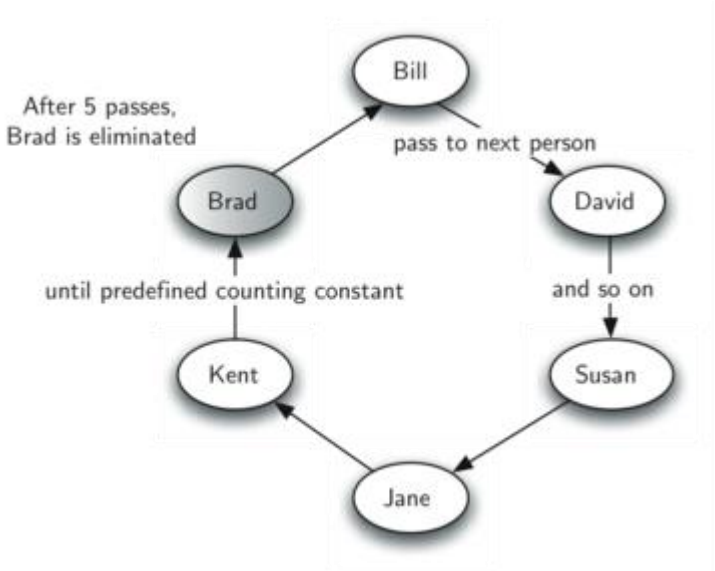


Figura 2: Um jogo de batata quente com seis pessoas

Este jogo é um equivalente moderno do famoso problema de Josephus. Baseado em uma lenda sobre o famoso historiador do século I, Flavius Josephus. A história diz que na revolta judaica contra Roma, Josephus e 39 de seus companheiros resistiram aos romanos em uma caverna. Com a derrota iminente, eles decidiram que prefeririam morrer a serem escravos dos romanos. Eles se organizaram em um círculo. Um homem foi designado como número um, e percorrendo o círculo no setindo e contando eles matavam todo o sétimo homem. Josephus, de acordo com a lenda, era entre outras coisas um matemático realizado. Ele instantaneamente descobriu onde deveria sentar-se para ser o último a ser executado. Quando chegou a hora, em vez de matar-se, ele se juntou ao lado romano. Você pode encontrar muitos versão diferentes desta história. Algumas versões matam cada terceiro homem e outras permitem que o último homem fuja em um cavalo. Em qualquer caso, a ideia é a mesma.

Vamos implementar uma **simulação** geral de Batata Quente. Nosso programa irá utilizar uma lista de nomes e uma constante “num” utilizada para a contagem. Ele retornará o nome da última pessoa restante depois contagem repetitiva por `num`. O que acontece nesse momento é com você.

Para simular o círculo, usaremos uma fila (veja:ref:*Figura 3 <fig_qupotatoqueue>*). Suponha que a criança segurando a batata esta no início da fila. Ao passar a batata, a simulação vai simplesmente remover essa criança da fila (`dequeue()`) e em seguida inseri-la no final da fila (`enqueue()`). Ela então esperará até

que todos os outros tenham passado pelo início antes que seja sua vez novamente. Depois de `num` operações de remoção/inserção, a criança no início será removida permanentemente e outro ciclo começará. Este processo continuará até que apenas um nome permaneça (o tamanho da fila seja 1).

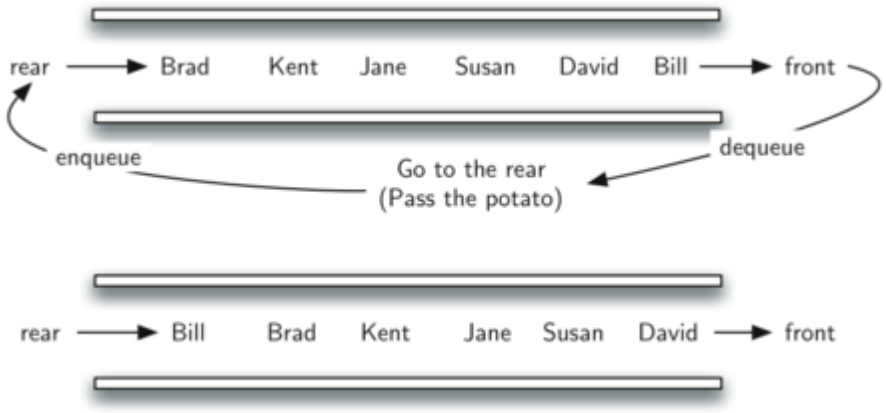


Figura 3: Uma Implementação de Batata Quente

O programa é mostrado em ActiveCode 1. Uma chamada de `hotPotato()` usando 7 como a constante para a contagem returno `Susan`.

```
from Queue import Queue
def hotPotato(namelist, num):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())

        simqueue.dequeue()

    return simqueue.dequeue()

print(hotPotato(["Bill","David","Susan","Jane","Kent","Brad"],7))
```

Note que neste exemplo o valor da constante de contagem é maior do que o número de nomes na lista. Isto não é um problema já que o fila age como um círculo e a contagem continua no início até que o valor seja atingido. Além disso, observe que a lista representa a fila de tal forma que o primeiro nome da lista será o do início da a fila. `Bill` neste caso é o primeiro item da lista e portanto, se move para o início fila. Uma variação dessa implementação, descrita nos exercícios, permite um contador aleatório.

Professor Luiz Thadeu Grizendi
UniAcademia

Simulação: Tarefas de Impressão

Uma simulação mais interessante nos permite estudar o comportamento de uma fila de impressão descrita anteriormente nesta seção. Lembre-se que como as alunas e os alunos enviam tarefas de impressão para uma impressora compartilhada, as tarefas são colocadas em uma fila para ser processada de maneira que a primeiro a chegar seja a primeiro a ser atendida. Muitas perguntas surgem com esta configuração. A mais importante delas pode ser se a impressora é capaz de lidar com uma certa quantidade de trabalhos. Se não for possível, as alunas e os alunos aguardarão muito tempo pela impressão e pode perder a próxima aula.

Considere a seguinte situação em um laboratório de ciência da computação. Em qualquer dia em média cerca de 10 alunas ou alunos estão trabalhando no laboratório em uma dada hora qualquer. Essas alunas e alunos costumam imprimir até duas vezes durante esse período e o tamanho dessas tarefas varia de 1 a 20 páginas. A impressora no laboratório é antiga, capaz de processar 10 páginas por minuto em qualidade de rascunho. A impressora pode ser trocada para oferecer melhor qualidade, mas produziria apenas cinco páginas por minuto. A velocidade de impressão mais lenta poderia fazer as alunas e os alunos esperarem demais. Qual frequência de impressão deve ser usada?

Poderíamos decidir em construir uma simulação que modela o laboratório. Nós precisará construir representações para as alunas e alunos, para as tarefas de impressão e para a impressora (Figura 4). À medida que as alunas e alunos enviam tarefas de impressão, vamos adicioná-las a uma lista de espera, uma fila de tarefas de impressão associada a impressora. Quando a impressora concluir uma tarefa, ela examinará a fila para ver se há alguma tarefa restante a ser processada. De interesse para nós é a quantidade média de tempo que as alunas e alunos vão esperar para que seus trabalhos sejam impressos. Isso é igual à quantidade média de tempo que uma tarefa aguarda na fila.

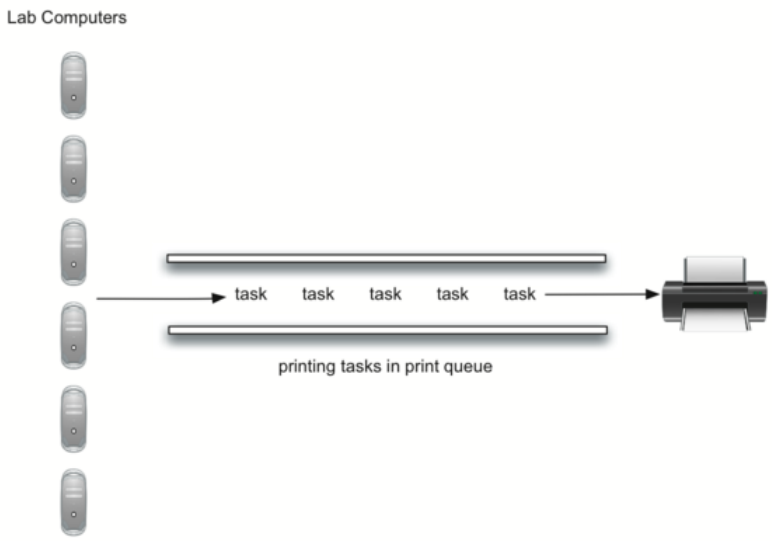


Figure 4: Fila de Impressão do Laboratório de Ciência da Computação

Para modelar essa situação, precisamos usar algumas probabilidades. Por exemplo, as alunas e os alunos podem imprimir um documento de 1 a 20 páginas. Se cada comprimento de 1 a 20 é igualmente provável, a duração de uma tarefa de impressão pode ser simulada usando um número aleatório entre 1 e 20 inclusive. Isto significa que há chance igual de qualquer comprimento de impressão de 1 a 20 entrar na fila.

Se houver 10 alunas e alunos no laboratório e cada uma delas e deles imprimir duas vezes, haverá 20 tarefas de impressão por hora, em média. Qual é a probabilidade de que a qualquer dado segundo, uma tarefa de impressão será criada? A maneira de responder isso é considerar a razão de tarefas para o tempo. 20 tarefas por hora significam que, em média, haverá uma tarefa a cada 180 segundos:

$$\frac{20\ tarefas}{1\ hora} \times \frac{1\ hora}{60\ minutos} \times \frac{1\ minuto}{60\ segundos} = \frac{1\ task}{180\ segundos}$$

Para cada segundo podemos simular a chance de que uma tarefa de impressão ocorra gerando um número aleatório entre 1 e 180 inclusive. Se o número for 180, dizemos que uma tarefa foi criada. Note que é possível que muitas tarefas sejam criadas em seguida ou podemos esperar um bom tempo para uma nova tarefa aparecer. Essa é a natureza da simulação. Você quer simular a situação real, tanto quanto possível, dado que você sabe alguns parâmetros gerais.

Principais Passos da Simulação

Aqui está a simulação principal.

1. Crie uma fila de tarefas de impressão. Cada tarefa será associada ao instante que chegou na fila (*timestamp*). Inicialmente a fila está vazia.
2. Para cada segundo (`currentSecond`):
    - **Uma nova tarefa de impressão é criada? Em caso afirmativo, adicione-a à fila com** `currentSecond` como o instante associado.
    - Se a impressora não estiver ocupada e se uma tarefa estiver aguardando,
        - Remova a próxima tarefa da fila de impressão e atribua-a a impressora.
        - **Subtrai o instante associado a tarefa do** `currentSecond` **para computar** o tempo de espera para essa tarefa.
        - **Anexe o tempo de espera dessa tarefa a uma lista para mais tarde ser** processada.
        - Com base no número de páginas da tarefa de impressão, calcule o tempo será necessário para imprimi-la.
    - **A impressora agora faz um segundo de impressão, se necessário. Também é** subtraido um segundo do tempo necessário para essa tarefa.

- **Se a tarefa foi concluída, em outras palavras, o tempo necessário** chegou a zero, a impressora passa a estaar desocupada.

3. Após a conclusão da simulação, calcule o tempo médio de espera da lista de tempos de espera gerados.

Implementação em Python

Para projetar esta simulação, criaremos classes para os três objetos do mundo real descritos acima: `Printer` (Impressora), `Task` (Tarefa) e `PrintQueue` (Fila de impressão).

A classe `Printer` precisará rverificar se há uma tarefa a ser impressa. Em caso afirmativo, ela está ocupado (linhas 13 a 17) e o tempo necessário pode ser calculado a partir do número de páginas da tarefa. O construtor também permitirá que a configuração de páginas por minuto seja inicializada. O método `tick` diminui o representa o cronômetro interno e determina que a impressora fique inativa (linha 11) se a tarefa tiver sido concluída.

```
class Printer:
    def __init__(self, ppm):
        self.pagerate = ppm
        self.currentTask = None
        self.timeRemaining = 0

    def tick(self):
        if self.currentTask != None:
            self.timeRemaining = self.timeRemaining - 1
            if self.timeRemaining <= 0:
                self.currentTask = None

    def busy(self):
        if self.currentTask != None:
            return True
        else:
            return False

    def startNext(self,newtask):
        self.currentTask = newtask
        self.timeRemaining = newtask.getPages() * 60/self.pagerate
```

A classe Task representará uma única tarefa de impressão. Quando a tarefa é criada, um gerador de números aleatórios fornecerá um comprimento de 1 a 20 páginas. Nós escolhemos usar para isso a função `randrange()` do módulo `random`.

Cada tarefa também precisará manter o instante em que foi criada (*timestamp*) a ser usado para computar o tempo de espera. Esse registro de data e hora representará o tempo em que a tarefa foi criada e colocada na fila da impressora. O método `waitTime()` pode então ser usado para recuperar a quantidade de tempo gasto na fila antes a impressão começar.

```
import random

class Task:
    def __init__(self,time):
        self.timestamp = time
        self.pages = random.randrange(1,21)

    def getStamp(self):
        return self.timestamp

    def getPages(self):
        return self.pages

    def waitTime(self, currenttime):
        return currenttime - self.timestamp
```

A simulação principal implementa o algoritmo descrito acima. O objeto `printQueue` é uma instância da nossa TDA fila (`queue`) existente. Uma função auxiliar booleana, `newPrintTask()`, decide se uma nova tarefa de impressão foi criada. Nós decidimos usar novamente a função `randrange()` do módulo `random` para retornar um inteiro aleatório entre 1 e 180. As tarefas de impressão chegam uma vez a cada 180 segundos. Ao escolher arbitrariamente 180 do intervalo de inteiros aleatórios (linha 28), podemos simular esse evento aleatório. A função de simulação nos permite definir o tempo total e as páginas por minuto para a impressora.

```
def simulation(numSeconds, pagesPerMinute):
    labprinter = Printer(pagesPerMinute)
    printQueue = Queue()
    waitingtimes = []

    for currentSecond in range(numSeconds):

      if newPrintTask():
         task = Task(currentSecond)
         printQueue.enqueue(task)

      if (not labprinter.busy()) and (not printQueue.isEmpty()):
        nexttask = printQueue.dequeue()
        waitingtimes.append(nexttask.waitTime(currentSecond))
        labprinter.startNext(nexttask)

      labprinter.tick()

    averageWait=sum(waitingtimes)/len(waitingtimes)
    print("Average Wait %6.2f secs %3d tasks remaining."
          %(averageWait,printQueue.size()))

def newPrintTask():
    num = random.randrange(1,181)
    if num == 180:
        return True
    else:
        return False
```

Quando executamos a simulação, não devemos nos preocupar que o os resultados são diferentes a cada vez. Isto é devido à natureza probabilística dos números aleatórios. Estamos interessados nas tendências que podem estar ocorrendo como resultado dos ajustes dos parâmetros para a simulação. Aqui estão alguns resultados.

Primeiro, executaremos a simulação por um período de 60 minutos (3.600 segundos) usando uma taxa de impressão de cinco páginas por minuto. Além disso, nós iremos executar 10 testes independentes. Lembre-se que como a simulação trabalha com números aleatórios cada execução retornará resultados diferentes.

```
>>> for i in range(10):

      simulation(3600,5)



Average Wait 165.38 secs 2 tasks remaining.
Average Wait  95.07 secs 1 tasks remaining.
Average Wait  65.05 secs 2 tasks remaining.
Average Wait  99.74 secs 1 tasks remaining.
Average Wait  15.27 secs 0 tasks remaining.
Average Wait 239.61 secs 5 tasks remaining.
Average Wait  75.11 secs 1 tasks remaining.
Average Wait  45.33 secs 0 tasks remaining.
Average Wait  39.31 secs 3 tasks remaining.
Average Wait 376.05 secs 1 tasks remaining.
```

Depois de executar nossos 10 testes, podemos ver que o tempo médio de espera (`Average Wait`) é 122,09 segundos. Você também pode ver que há uma grande variação no tempo médio de espera com um mínimo de 17,27 segundos e um máximo de 376,05 segundos. Você também pode notar que em apenas dois dos casos foram todas as tarefas concluídas.

Agora, ajustaremos a taxa de páginas para 10 páginas por minuto e executaremos as 10 testes novamente, com uma frequência maior de páginas por segundo, nossa esperança seria que mais tarefas seria concluídas no período de uma hora.

```
>>>for i in range(10):

      simulation(3600,10)

Average Wait   1.29 secs 0 tasks remaining.
Average Wait   5.00 secs 0 tasks remaining.
Average Wait  25.96 secs 1 tasks remaining.
Average Wait  13.55 secs 0 tasks remaining.
Average Wait  12.67 secs 0 tasks remaining.
Average Wait   6.46 secs 0 tasks remaining.
Average Wait  22.33 secs 0 tasks remaining.
Average Wait  12.39 secs 0 tasks remaining.
Average Wait   5.27 secs 0 tasks remaining.
Average Wait  15.17 secs 0 tasks remaining.
```

Professor Luiz Thadeu Grizendi
UniAcademia

Exercícios de Fila

1. Considere a rotina a seguir que utiliza as rotinas offer(e), que serve para inserir um elemento na fila, poll() para remover e peek() para retornar o 1º elemento da fila.

```
q.clear();
for  i in range(0, 5):
     q.offer(i)
     q.offer(2*i)
     q.poll()
     print(str(q.peek())+',')
```

   Dar a sequência que será impressa.

2. Complete a tabela a seguir considerando o TAD Queue (FILA) já inicializada com os valores 10,33,5,44. Considere o 1º elemento da fila o elemento 10 e assim sucessivamente.

OPERAÇÃO	SAÍDA	CONTEÚDO DA FILA
size()	4	(10,33,5,44)
contains(33)		
offer (5)		
poll()		
peek()		
size()		

   Onde:

   offer(e): Insere o elemento "e", na fila.
   poll() Remove o 1º elemento da fila.
   peek() Retorna o 1º elemento da fila.

3. A letra significa inserir um elemento na fila e um asterisco significa remover o elemento da fila na seguinte seqüência. Dar a seqüência de valores devolvidos pelas operações poll() quando esta seqüência de operações é realizada em uma fila inicialmente vazia.

E A S * Y * Q U E *** S T *** I O * N ***

4. Suponha que uma seqüência misturada de operações inserir na fila (offer) e remover o elemento da fila (poll) são executadas. Os elementos inseridos são números inteiros de 0 a 9 em ordem. A seguir descrevem-se sequências de valores de retorno do desempilhamentos. Qual das seguintes seqüências não poderia ocorrer?
   (a) 0 1 2 3 4 5 6 7 8 9
   (b) 4 6 8 7 5 3 2 9 0 1
   (c) 2 5 6 7 4 8 9 3 1 0
   (d) 4 3 2 1 0 5 6 7 8 9

5. Assume que x, y, z sejam variáveis inteiras e que a fila é do tipo inteiros, dê o estado de saída de cada programa.
   a)

```
x = 3
y = 5
z = 2
q.clear()
q.offer(x)
q.offer(4)
z=q.poll()
q.offer(y)
q.offer(3)
q.offer(z)
x=q.poll()
q.offer (2)
q.offer (x)
while not q.isEmpty():
     x=q.poll()
     print(str(x)+',')
```

b) ,

```
y = 1
q.clear()
q.offer(5)
q.offer(7)
x=q.poll()
x += y
q.offer(x)
q.offer(y)
q.offer(2)
q.poll()
x=q.poll()
while not q.isEmpty():
        y=q.poll()
        print(str(y)+',')

print('x = ' + str(x))
print('y = ' + str(y))
```

6. Mostre o que será impresso em cada sequência de fragmento de programa, onde *s1* e *s2* são filas de inteiros e *j, k e m* são variáveis inteiras.

a) .

```
q1.clear()
q2.clear()
for j in range(0,9):
        q1.offer(j)
while not q1.isEmpty():
       j=q1.poll()
       if(j % 2 == 0):
            q2.offer(j)

while not q2.isEmpty():
         j=q2.poll()
         print(str(j)+',')
```

b)

```
j = 1
q1.clear()
q2.clear()
while ( j * j < 50 ):
     k = j * j
     q1.offer(k)
     j += 1

for j  in range(1, 6):
     k=q1.poll()
     q2.offer(k)

j=q1.poll()
for  k  in range(1,j):
        m=q2.poll()
        q1.offer(m)

while not q1.isEmpty():
        j=q1.poll()
        print(str(j)+',')
```

Professor Luiz Thadeu Grizendi
UniAcademia

O que é uma Deque?

Uma **deque**, também é conhecida como fila de duas extremidades, é uma coleção ordenada de itens semelhantes à fila. Tem duas extremidades, uma é o início ou frente (*front*) e a outra, o fim ou retarguada (*rear*), e os itens permanecem posicionados na coleção. O que faz um deque diferente é a natureza não-restritiva de adicionar e remover itens. Novos itens podem ser adicionados no início ou no fim. Da mesma forma, itens existentes podem ser removidos de qualquer uma das extremidades. De certa forma, esta estrutura híbrida fornece todas as capacidades de pilhas e filas em uma única estrutura de dados Figura 1 mostra um deque de objetos de dados do Python.

É importante notar que, embora a deque possa assumir muitas das características de pilhas e filas, não requer a ordenação LIFO nem a FIFO que são impostas para essas estruturas de dados. A responsabilidade é do programador fazer uso consistente das operações de inserção e remoção.

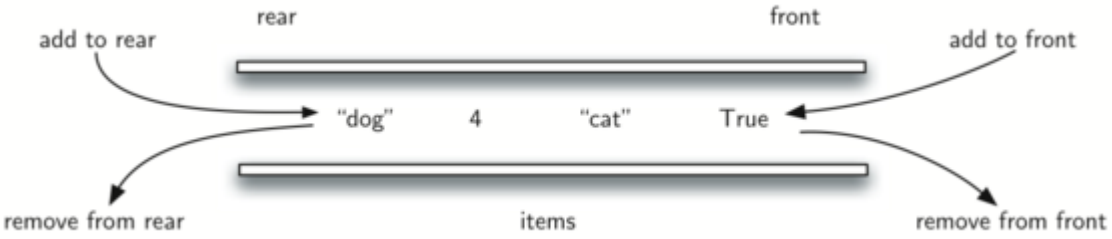


Figura 1: Uma Deque de Objetos do Python

O Tipo Abstrato de Dados Deque

O tipo de dado abstrato de deque (*fila dupla*) é definido pela seguinte estrutura e operações. Um deque é estruturado, como descrito anteriormente, como um coleção de itens em que os itens são inseridos e removidos de qualquer extremidade início ou fim. As operações deque são dadas abaixo.

- `Deque ()` **cria uma nova deque vazia. Não precisa de parâmetros**
  e retorna um deque vazia.
- `addFront(item)` insere um novo item no início da deque. Isto precisa do item como parâmetro e não retorna nada.
- `addRear(item)` insere um novo item no fim da deque. Precisa do item e não retorna nada.
- `removeFront()` remove o item do início da deque. Não precisa de parâmetros e retorna o item. A deque é modificada.
- `removeRear()` remove o item do fim da deque. Não precisa de parâmetros e retorna o item. A deque é modificada.
- `isEmpty()` testa se a deque está vazia. Não precisa de parâmetros e retorna um valor booleano (`bool`); `True` se a fila está vazia e `False` em caso contrário.
- `size()` retorna o número de itens na deque. Não precisa de parâmetros e retorna um inteiro (`int`).

Por exemplo, se supormos que `d` é uma deque que foi criada e está atualmente vazia, então a tabela 1 mostra os resultados de uma sequência de operações sobre a deque. Note que o conteúdo do início é listado à direita. É muito importante acompanhar o início e o fim à medida que itens são inseridos e removidos da deque já que as coisas podem ficar um pouco confusas.

Tabela 1: Exemplos de Operações sobre Deques

Operação	Conteúdo da Deque	Valor Retornado
`d.isEmpty()`	`[]`	`True`
`d.addRear(4)`	`[4]`	
`d.addRear('dog')`	`['dog',4]`	
`d.addFront('cat')`	`['dog',4,'cat']`	
`d.addFront(True)`	`['dog',4,'cat',True]`	
`d.size()`	`['dog',4,'cat',True]`	`4`
`d.isEmpty()`	`['dog',4,'cat',True]`	`False`
`d.addRear(5.4)`	`[5.4,'dog',4,'cat',True]`	
`d.removeRear()`	`['dog',4,'cat',True]`	`5.4`
`d.removeFront()`	`['dog',4,'cat']`	`True`

Professor Luiz Thadeu Grizendi
UniAcademia

Implementação de uma Deque in Python

Como fizemos nas seções anteriores, criaremos uma classe para a implementação do tipo de dados abstrato deque. Novamente, a lista do Python irá fornecer um conjunto muito bom de métodos sobre os quais construir os detalhes do deque. Nossa implementação assumirá que o fim da deque está na posição 0 na lista.

```
class Deque:
        def __init__(self):
            self.items = []

        def isEmpty(self):
            return self.size() == 0

        def addFront(self, item):
            self.items.insert(0,item)

        def addRear(self, item):
            self.items.append(item)

        def removeFront(self):
            return self.items.pop(0)

        def removeRear(self):
            return self.items.pop()

        def rear(self):
            return self.items[self.size()-1]

        def front(self):
            return self.items[0]
```

Em `removeFront()` usamos o método `pop(0)` para remover o primeiro elemento da lista. No entanto, em `removeRear()`, o método `pop()` deve remover o último elemento da lista. Da mesma forma, precisamos usar o método `insert()` em `addFront()` , inserir na posição inicial da lista e, o método `append()` em `addRear(),` que pressupõe a adição de um novo elemento ao final da lista.

Professor Luiz Thadeu Grizendi
UniAcademia

Verificação de Palíndromos

Um problema interessante que pode ser facilmente resolvido usando a estrutura de dados deque (*fila dupla*) é o problema clássico do palíndromo. Um **palíndromo** é uma string que é lida da mesma forma do início para o fim e do fim para o início, por exemplo, *radar*, *toot* e *aba* são palíndromos. Gostaríamos de construir um algoritmo para inserir se uma string é um palíndromo.

A solução para este problema usará um deque para armazenar os caracteres da string. Vamos processar a string da esquerda para a direita e inserir cada caractere no fim do deque. Neste ponto, o deque atuará agindo de maneira muito parecida com uma fila normal. No entanto, agora podemos fazer uso da a dupla funcionalidade do deque. O início da deque armazenará o primeiro caractere da string e o fim da deque manterá o último caractere (veja Figura 2).

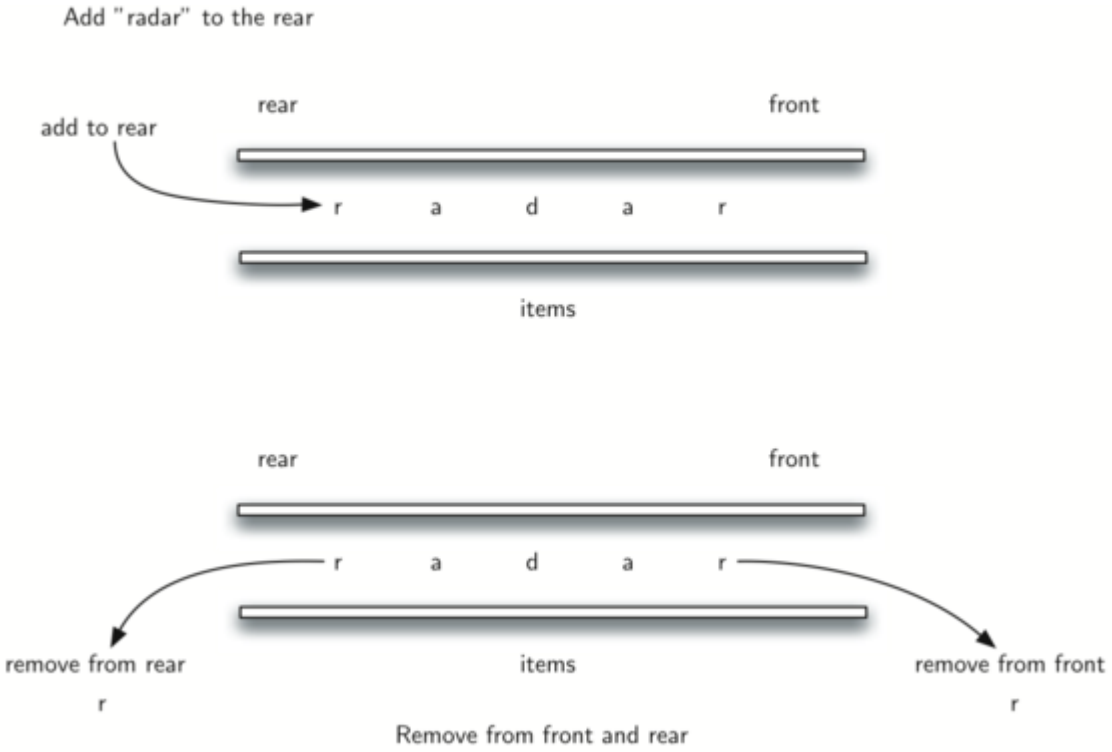


Figure 2: Uma Deque

Como podemos remover os ambos diretamente, podemos compará-los e continuar apenas se esses caracteres forem iguais. Se pudermos manter a correspondência entre os itens do início e do fim, ao final ficaremos sem caracteres na deque ou ficar com uma deque de tamanho 1, dependendo se o comprimento da string original era par ou ímpar. Em ambos os casos, a string deve ser um palíndromo. a função completa para verificação de palíndromos pode ser vista no código a seguir.

```
from Deque import Deque

def isPal(aString):
    chardeque = Deque()

    for ch in aString:
        chardeque.addRear(ch)

    stillEqual = True

    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False

    return stillEqual
```

Observação

O Python já implementa a estrutura deque, que pode ser utilizada para pilha, fila ou até uma lista linear mesmo.

Primeiramente devemos buscar a estrutura deque que está no repositório collections. A chamada é feita da seguinte forma:

```
from collections import deque
```

Inicializando a deque vazia

```
q = deque()
```

Inicializando a deque passando uma lista de elementos (A ordem de inserção será a mesma da lista de entrada)

```
q = deque([10,30])
```

Esta estrutura não implementa os métodos front e rear da deque, logo, deve-se ficar ao cargo do programador

Obtenção do elemento na frente da deque (front)

```
deque([10, 30])
>>> q[0]
10
```

Obtenção do elemento na retaguarda da deque (rear)

```
>>> q[len(q)-1]
30
```

Para adicionarmos por exemplo o elemento 5, na frente da deque (addFront), à esquerda, devemos proceder da seguinte forma:

```
q.appendleft(5)
```

Para adicionarmos por exemplo o elemento 50, na retaguarda da deque (addRear), à direita, devemos proceder da seguinte forma:

```
q.append(50)
```

Supondo a deque inicialmente com os elementos

```
deque([5, 10, 30, 50])
```

Para removermos o elemento da frente da deque (removeFront), à esquerda, devemos proceder da seguinte forma:

```
>>> q.popleft()
5
```

Para removermos o elemento da retaguarda da deque (removeRear), à direita, devemos proceder da seguinte forma:

```
>>> q.pop()
50
```

Para verificarmos se a deque está vazia (isEmpty) devemos verificar se o tamanho do deque é igual a zero, da seguinte forma:

```
>>> len(q)==0
False
```