ESTRUTURA DE DADOS

Júlia Mara Colleoni Couto

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Implementação de deques em Python

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Reconhecer uma deque.
- Especificar operações de manipulação de deques.
- Identificar operações de acesso a uma deque.

Introdução

Uma deque é um tipo abstrato de dados, que generaliza uma fila, e a origem do nome deque vem do inglês *double ended queue*, ou fila duplamente terminada. Um dos aspectos que diferem deques de estruturas de dados, como filas e pilhas, é a sua flexibilidade no processo de adição e remoção de itens. Novos itens podem ser adicionados tanto no início quanto no fim de suas extremidades. O mesmo acontece com itens existentes, que podem ser removidos de qualquer uma das extremidades. Por esse motivo, pode-se dizer que uma deque é uma estrutura "híbrida" que reúne as características de pilhas e filas em uma única estrutura de dados.

Neste capítulo, você vai conhecer e trabalhar a estrutura de dados do tipo deque, aprender sobre as operações possíveis em deques e identificar as operações de acesso — tudo isso com exemplos práticos, implementados na linguagem de programação Python.

1 O que é uma estrutura do tipo deque?

Uma fila dupla, ou deque, suporta a adição e a remoção de elementos em ambas as extremidades, diferentemente das pilhas e filas, nas quais as entradas e as saídas dos elementos se restringem a uma única entrada/extremidade. Uma deque também é conhecida por fila de duas extremidades, sendo uma coleção ordenada de itens, semelhante a uma fila. Mesmo tendo duas extremidades, uma é o início (cabeça) e a outra é a traseira (cauda), e a manipulação pode ser feita exclusivamente por essas extremidades (EDELWEISS; GALANTE, 2009).

Em uma pilha, os itens são removidos e adicionados somente por uma extremidade, usando uma abordagem LIFO (*last in, first out*), na qual o primeiro a entrar é o último a sair. No entanto, em filas, a abordagem adotada é a FIFO (*first in, first out*), na qual o primeiro a entrar é o primeiro a sair. Por outro lado, em uma deque, os itens podem ser adicionados no início ou no fim, e os itens existentes podem ser removidos de qualquer uma das extremidades. Assim, essa estrutura híbrida une características de pilhas e filas em uma única estrutura de dados (GOODRICH; TAMASSIA, 2013). A Figura 1 mostra a representação gráfica de uma deque, com suas possibilidades de inserções, consultas e exclusões.

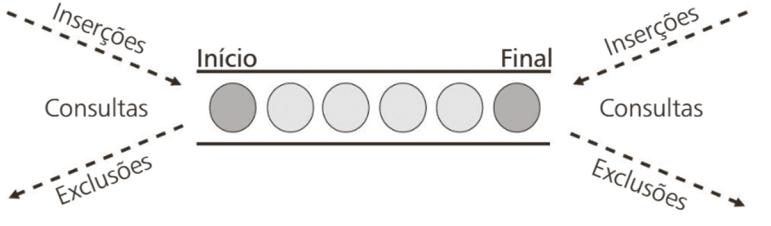

Figura 1. Representação de uma deque.
Fonte: Edelweiss e Galante (2009, p. 147).

Há duas formas possíveis para representar fisicamente uma deque: por contiguidade física e por encadeamento (EDELWEISS; GALANTE, 2009). A implementação de deques por contiguidade física é similar a uma fila simples, sendo recomendado que seja implementada com uma estratégia de ocupação circular do espaço disponível, para melhorar a otimização. Entretanto, na implementação de deques com o uso de encadeamento, assim como na contiguidade física, é importante utilizar um descritor que aponte para o endereço do primeiro e do último elemento, para que eles possam ser acessados diretamente.

2 Operações de manipulação sobre deques

As operações possíveis sobre deques envolvem adição e remoção de elementos tanto na cabeça (na frente) quando na cauda (atrás) da deque. As operações de manipulação alteram o estado da deque adicionando e removendo itens.

Você pode implementar uma deque do zero, levando em conta que deve desenvolver métodos, pelo menos, para estas operações básicas:

- `deque()` — define uma classe que cria uma nova deque que está vazia, não precisa de parâmetros e retorna uma deque vazia;
- `adicionaFrente(item)` — insere um novo item no início da deque;
- `adicionaAtras(item)` — insere um novo item no fim da deque;
- `removeFrente()` — remove o item do início da deque;
- `removeAtras()` — remove o item do fim da deque.

Exemplo

Um exemplo de contexto no qual encontramos a aplicação de deque inclui o histórico de um navegador da *web*, em que as URLs visitadas recentemente são adicionadas à frente da deque — as últimas URLs, na parte de trás da deque, são removidas após um número específico de inserções na frente. Segundo Edelweiss e Galante (2009), deques também podem ser utilizadas para implementar modelos de canais de navegação fluvial ou marítima e becos onde há possibilidade de circulação nos dois sentidos, mas que não comportam a circulação paralela (um carro deve passar de cada vez).

No código a seguir, apresenta-se um exemplo simples de como criar esses métodos:

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

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

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

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

    def removeAtras(self):
        return self.items.pop(0)
```

Utilizando a classe criada acima, vamos ver como funcionam os comandos na prática. O resultado apresentado após a execução de cada comando é apresentado após o caractere #.

Primeiro, criamos uma deque vazia:

```
d = Deque()
```

Então, adicionamos o "1" na frente e imprimimos a deque resultante da operação:

```
d.adicionaFrente(1)
print (list(d.items))
    # [1]
```

Na sequência, adicionamos o "7" na frente:

```
d.adicionaFrente(7)
print (list(d.items))
    # [1, 7]
```

Depois, adicionamos o "4" atrás:

```
d.adicionaAtras(4)
print (list(d.items))
     # [4, 1, 7]
```

Depois, adicionamos mais um "8" atrás:

```
d.adicionaAtras(8)
print (list(d.items))
     # [8, 4, 1, 7]
```

Agora, faremos uma remoção na frente:

```
d.removeFrente()
print (list(d.items))
     # [8, 4, 1]
```

Agora, faremos uma remoção atrás:

```
d.removeAtras()
print (list(d.items))
     # [4, 1]
```

Por fim, imprimimos a deque resultante:

```
print (list(d.items))
     # [4, 1]
```

Adicionalmente, existe a biblioteca *collections* em Python que automatiza a implementação desses métodos de manipulação, além de disponibilizar outras funcionalidades interessantes (PYTHON SOFTWARE FOUNDATION, 2020).

- `append(x)` — adiciona "x" ao lado direito da deque;
- `appendleft(x)` — adiciona "x" ao lado esquerdo da deque;
- `extend(iterable)` — estende o lado direito da deque, adicionando elementos a partir do argumento `iterable`;
- `extendleft(iterable)` — estende o lado esquerdo da deque, adicionando elementos a partir do argumento `iterable`;
- `insert(i, x)` — insere o elemento "x" dentro da deque, na posição "i";
- `pop()` — remove e retorna um elemento do lado direito da deque;

- `popleft()` — remove e retorna um elemento do lado esquerdo da deque;
- `remove(value)` — remove a primeira ocorrência do valor passado como parâmetro;
- `reverse()` — inverte localmente os elementos da deque e retorna *None*;
- `rotate(n=1)` — gira a deque *n* passos para a direita, e se *n* for negativo, gira para a esquerda;
- `Maxlen` — tamanho máximo da deque;
- `clear()` — deleta todos os elementos, deixando com tamanho igual a 0.

Para entender melhor esses métodos, vamos criar uma deque e aplicar as operações disponíveis por meio do uso da biblioteca *collections*.

```
from collections import deque # ou import collections
d = collections.deque()
print (d)
    # deque([])
```

Agora, vamos adicionar dois elementos na sequência, "a" e "b", que serão inseridos à direita da deque:

```
d.append('a')
d.append('b')
print (d)
    # deque(['a', 'b'])
```

Adicionaremos mais um elemento "b", agora à esquerda:

```
d.appendleft('b')
print (d)
    # deque(['b', 'a', 'b'])
```

Agora, vamos inserir à direita diversos elementos:

```
d.extend('cdef')
print (d)
    # deque(['b', 'a', 'b', 'c', 'd', 'e', 'f'])
```

Também vamos inserir mais de um elemento à esquerda:

```
d.extendleft('ghij')
print (d)
     #deque(['j', 'i', 'h', 'g', 'b', 'a', 'b', 'c', 'd', 'e', 'f'])
```

Inserimos o elemento "k" na posição 5:

```
d.insert(5, 'k')
print (d)
     #deque(['j', 'i', 'h', 'g', 'b', 'k', 'a', 'b', 'c', 'd', 'e', 'f'])
```

Removemos o elemento à direita, que no caso será o "f":

```
d.pop()
print (d)
     #deque(['j', 'i', 'h', 'g', 'b', 'k', 'a', 'b', 'c', 'd', 'e'])
```

Removemos o elemento à esquerda, que no caso será o "j":

```
d.popleft()
print (d)
     #deque(['i', 'h', 'g', 'b', 'k', 'a', 'b', 'c', 'd', 'e'])
```

Removemos, especificamente, a primeira ocorrência do elemento "b":

```
d.remove('b')
print (d)
     #deque(['i', 'h', 'g', 'k', 'a', 'b', 'c', 'd', 'e'])
```

Invertemos a deque utilizando o comando `reverse`:

```
d.reverse()
print (d)
     #deque(['e', 'd', 'c', 'b', 'a', 'k', 'g', 'h', 'i'])
```

Rotacionamos a deque com o comando `rotate` (traz o primeiro elemento para a frente da deque):

```
d.rotate()
print (d)
     #deque(['i', 'e', 'd', 'c', 'b', 'a', 'k', 'g', 'h'])
```

Limpamos a deque, excluindo todos os elementos:

```
d.clear()
print (d)
     # deque([])
```

Adicionalmente, também é possível criar uma deque com tamanho definido. No exemplo, limitamos o tamanho a 30 elementos:

```
d3 = collections.deque(maxlen=30)
print (d3)
     #deque([], maxlen=30)
```

Como você pode perceber, diferentemente de uma estrutura de dados da deque tradicional, na qual o acesso só pode ser feito exclusivamente pelas duas extremidades de início e fim, a biblioteca *collections* permite também acesso a elementos que se encontram em posições intermediárias, desde que seja passado pelo parâmetro qual o elemento ou qual o índice em que o elemento se encontra, ou onde ele será adicionado (MENEZES, 2019).

3 Operações de acesso sobre deques

As operações de acesso permitem os mais variados tipos de consultas, como verificar se está cheia ou vazia, e verificar o tamanho da deque. Quando implementamos uma deque do zero, precisamos pelo menos de duas operações de acesso:

- `estaVazio()` — testa se a deque está vazia;
- `tamanho()` — retorna o número de itens na deque.

Vamos adicioná-las à deque criada inicialmente e mostrar o resultado.

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

    def estaVazio(self):
        return self.items == []

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

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

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

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

    def tamanho(self):
        return len(self.items)
```

Utilizando a classe criada acima, vamos ver como funcionam os comandos na prática. O resultado apresentado após a execução de cada comando é exibido após o caractere #.

Primeiro, criamos uma deque vazia:

```
d = Deque()
```

Depois, usamos o método para confirmar se a deque está vazia, transformando em valor booleano para que tenhamos uma resposta `True` ou `False`:

```
print(bool(d.estaVazio))
    #True
```

Usando o método `tamanho`, verificamos quantos itens há na deque:

```
print(d.tamanho())
    #2
```

Na biblioteca *collections*, também há métodos de acesso (PYTHON SOFTWARE FOUNDATION, 2020).

- `copy()` — cria uma cópia da deque;
- `count(x)` — conta quantos elementos na deque são iguais a "x";
- `index(x[, start[, stop]])` — retorna a posição da primeira ocorrência de "x" na deque, a partir do índice `start` e antes do índice `stop`.

Vamos, novamente, criar uma deque e iniciá-la, utilizando o comando `extend` que vimos anteriormente:

```
from collections import deque # ou import collections
d = collections.deque()
d.extend('abcdefgh')
print (d)
    # deque(['a', 'b', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
```

Usando o comando `copy`, vamos fazer uma cópia da versão atual da deque. Assim, quando criamos uma cópia e continuamos manipulando a versão original, a cópia mantém o estado do momento em que foi duplicada.

```
d2 = d.copy()
print (d2)
    #deque(['a', 'b', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
```

Agora, vamos contar quantos elementos iguais a "b" há na deque:

```
print(d.count('b'))
    # 2
```

Agora, perguntamos à deque qual é o índice do elemento "a" (lembrando que o índice do primeiro elemento é igual a zero):

```
print(d.index('a'))
    # 0
```

Deques são estruturas muito úteis para a resolução de alguns problemas computacionais. Neste capítulo, você aprendeu a reconhecer uma deque, desenvolver uma deque e suas operações de acesso e manipulação, e também aprendeu sobre os métodos prontos disponibilizados na biblioteca *collections*.

Referências

EDELWEISS, N.; GALANTE, R. *Estruturas de dados*. Porto Alegre: Bookman, 2009.

GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados e algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.

MENEZES, N. N. C. *Introdução à programação com Python*: algoritmos e lógica de programação para iniciantes. 3. ed. São Paulo: Novatec, 2019.

PYTHON SOFTWARE FOUNDATION. *Collections*: container datatypes: deque objects. 2020. Disponível em: https://docs.python.org/3.9/library/collections.html#deque-objects. Acesso em: 25 fev. 2020.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS