Busca

Estruturas de Dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Os TAD's Pilha, Fila e FilaDupla, permitem o armazenamento e recuperação de itens independente do conteúdo.

O TAD Lista tem apenas uma operação que é dependente do conteúdo: `remove_item`.

Vamos estudar um TAD em que a maioria das operações depende do conteúdo dos itens armazenados.

Dicionário

Um **dicionário**, também chamado de arranjo associativo ou mapa, é um tipo abstrato de dados que representa uma coleção de associações chave-valor, onde cada chave é única.

As operações comuns em um dicionário são a associação de uma chave com um valor, a busca do valor associado com uma chave e a exclusão de uma chave e o valor associado.

Dicionário

```python
class Dicionario:
  '''Uma coleção de associações chave-valor, onde
  cada chave é única.'''

  def num_itens(self) -> int:
    '''Devolve a quantidade de chaves no dicionário.'''

  def associa(self, chave: str, valor: int):
    '''Associa a *chave* com o *valor* no dicionário.
    Se *chave* já está associada com um valor, ele
    é sustituído por *valor*.'''

  def busca(self, chave: str) -> int | None:
    '''Devolve o valor associado com *chave* no dicio-
    nário ou None se a chave não está no dicionário.'''

  def remove(self, chave: str):
    '''Remove a *chave* e o valor associado com ela do
    dicionário. Não faz nada se a *chave* não está no
    dicionário.'''
```

```
>>> d = Dicionario()
>>> d.num_itens()
0
>>> d.associa('Jorge', 25)
>>> d.associa('Bia', 40)
>>> d.num_itens()
2
>>> d.busca('Jorge')
25
>>> d.busca('Bia')
40
>>> d.busca('Andre') is None
True
>>> d.associa('Bia', 50)
>>> d.busca('Bia')
50
>>> d.remove('Jorge')
>>> d.busca('Jorge') is None
True
>>> d.remove('Ana')
```

Dicionário - Implementação com arranjo

Como podemos implementar o TAD Dicionário utilizando arranjo?

- Armazenamos um par chave-valor em cada posição do arranjo.
- Busca: busca por todos os itens, se a chave está presente, devolve o valor associado, senão devolve `None`.
- Associação: *busca* por todos os itens, se a chave está presente, atualiza o valor, senão adiciona a nova associação chave-valor no final.
- Remoção: *busca* por todos os itens, se a chave está presente, troca pelo último item e remove o último.

```python
@dataclass class Item:
    chave: str
    valor: int

class Dicionario:
    itens: list[Item]

    def __init__(self) -> None:
        self.itens = []

    def num_itens(self) -> int:
        return len(self.itens)

    def __busca(self, chave: str) -> int | None:
        '''Devolve a posição da *chave* ou
        None se a *chave* não está presente.'''
        for i in range(len(self.itens)):
            if self.itens[i].chave == chave:
                return i
        return None
```

Dicionário - Implementação com arranjo

```
class Dicionario:
    def associa(self, chave: str, valor: int):
        i = self.__busca(chave)
        if i is not None:
            self.itens[i].valor = valor
        else:
            self.itens.append(Item(chave, valor))
    def busca(self, chave: str) -> int | None:
        i = self.__busca(chave)
        if i is not None:
            return self.itens[i].valor
        else:
            return None
    def remove(self, chave: str):
        i = self.__busca(chave)
        if i is not None:
            self.itens[i], self.itens[-1] = \
                self.itens[-1], self.itens[i]
            self.itens.pop()
```

Qual a complexidade de tempo das operações?

Todas têm tempo de execução $O(n)$ pois requerem uma busca que pode analisar todos os itens.

Será que podemos fazer melhor usando encadeamento linear?

Não... A busca ainda precisaria analisar todos os elementos no pior caso.

Podemos fazer melhor? As operações dependem do conteúdo do item mas não estamos usando o conteúdo para organizar os itens.

Busca eficiente

Como organizar uma coleção de cartas Pokémon em um monte de maneira que seja possível encontrar uma carta rapidamente, isso é, sem precisar olhar todas elas?

Se as cartas estiverem em ordem alfabética, dividimos o monte aproximadamente ao meio e olhamos para a carta que está no topo da segunda metade. Se for a carta que estamos procurando, ótimo, terminamos! Caso contrário:

- Se a carta que estamos procurando vem antes, em ordem alfabética, repetidos o processo para a primeira metade;
- Se a carta vem depois, repetimos o processo para a segunda metade descartando a carta que já verificamos;
- Se o monte ficar vazio, concluímos que a carta não está presente.

Este algoritmo é chamado de **busca binária**.

Busca binária em arranjo

Como podemos fazer uma busca binária em um arranjo?

Mantemos duas variáveis, `ini` e `fim`, que indicam respectivamente o início e o fim do intervalo do arranjo onde estamos fazendo a busca.

Se o intervalo é vazio, finalizamos a busca.

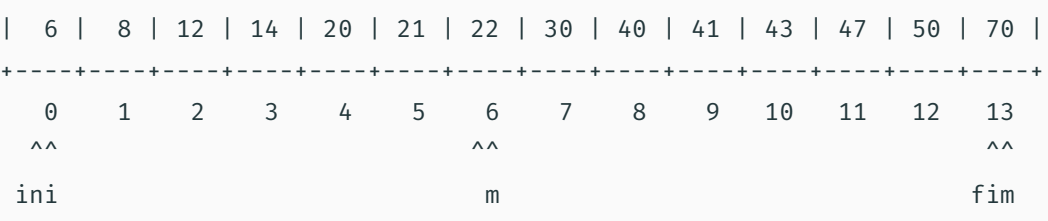
Senão, verificamos se o elemento que estamos buscando está no meio (`(ini + fim) // 2`).

Se estiver, encontramos o elemento e finalizamos a busca.

Senão, atualizamos o intervalo e fazemos a busca novamente.

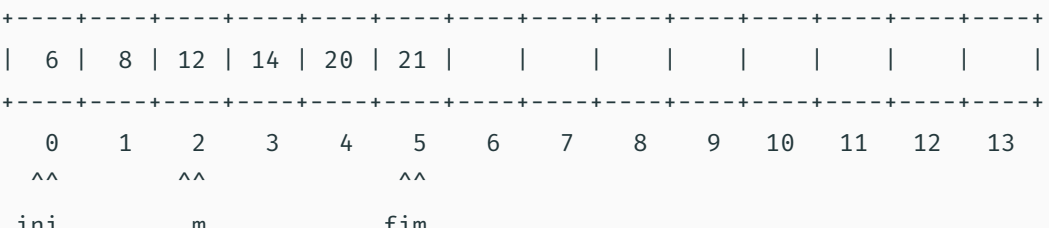
Exemplo - pesquisa pelo 20

Busca pelo 20. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o "meio".



Exemplo - pesquisa pelo 20

Busca pelo 20. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o "meio".

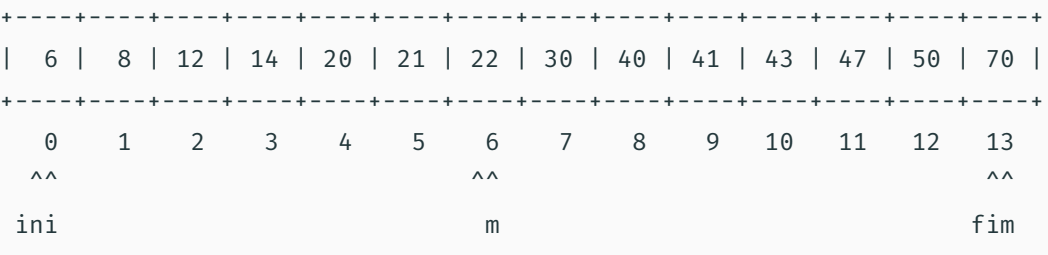


Exemplo - pesquisa pelo 20

Busca pelo 20. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o “meio”.

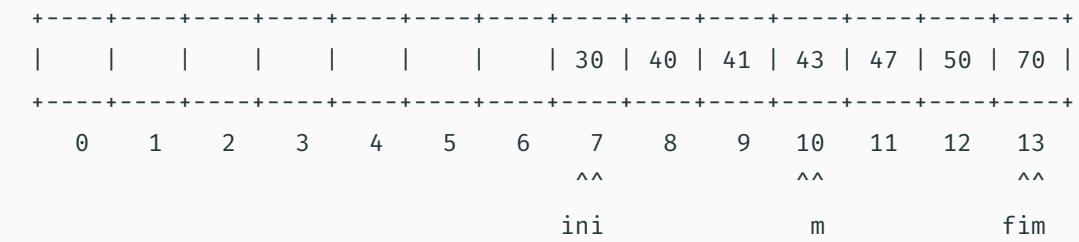
Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o "meio".



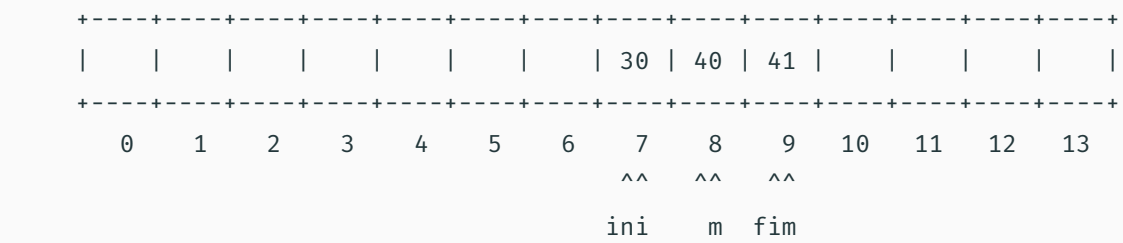
Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o "meio".



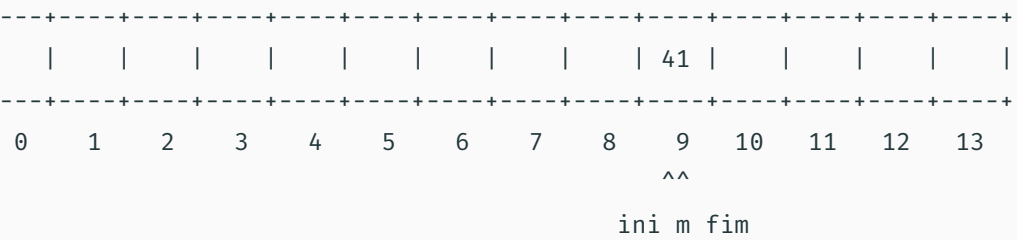
Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o “meio”.



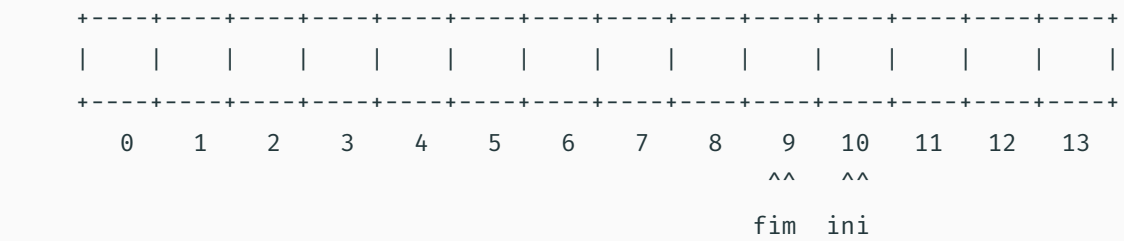
Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o “meio”.



Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` indicam o intervalo e `m = (ini + fim) // 2` é o "meio".



Quantas comparações no máximo são feitas entre a chave e um valor do arranjo? Quantas divisões sucessivas por 2 são necessárias para que um valor n chegue em 1?

Supondo que n seja uma potência de dois, e sendo i a quantidade de divisões, temos

$$\frac{n}{2^i} = 1 \rightarrow n = 2^i$$

Aplicando $\log_2$ obtemos

$$\log_2 n = \log_2 2^i \rightarrow i = \lg n$$

Portanto, a complexidade de tempo da busca binária é $O(\lg n)$.

Como as complexidades de tempo da busca linear e binária se comparam?

n	Busca linear	Busca binária
10^1	10	≈ 4
10^2	100	≈ 7
10^3	1.000	≈ 10
10^6	1.000.000	≈ 20
10^9	1.000.000.000	≈ 30

Implementação da busca binária

Existem várias formas de implementar a busca binária (veja a lista de exercícios!).

A seguir, mostramos uma implementação iterativa que devolve um índice onde a chave está na lista ou onde ela deveria estar. Isto é útil, pois permite usar esse índice para inserir a chave caso ela não esteja presente.

Implementação da busca binária

```
def busca_binaria(valores: list[int], chave: int) -> int:
    '''
    Se *chave* está presente em *valores*, devolve o
    índice i tal que *valores[i] == chave*. Senão,
    devolve o índice i tal que a inserção de *chave*
    na posição *i* de *valores* mantém *valores* em
    ordem não decrescente.

    Requer que *valores* esteja em ordem não decrescente.

    Exemplos
    >>> busca_binaria([6, 8, 10, 12, 20], 7)
    1
    >>> busca_binaria([6, 8, 10, 12, 20], 20)
    4
    '''
    ini = 0
    fim = len(valores) - 1
    while ini <= fim:
        m = (ini + fim) // 2
        if chave == valores[m]:
            return m
        elif chave < valores[m]:
            fim = m - 1
        else: # chave > valores[m]
            ini = m + 1
    return ini
```

O que é preciso para podermos utilizar a busca binária na implementação do TAD dicionário?

Manter as associações chave-valor ordenadas pela chave.

A implementação fica como exercício.

Qual é a complexidade de tempo de `busca`? $O(\lg n)$.

E a complexidade de tempo `associa` e `remove`? Continua sendo $O(n)$!

Quando a implementação de dicionário utilizando arranjo ordenado e busca binária é adequada?

Quando a quantidade de consultas for muito maior que a quantidade de alterações.

E a implementação usando arranjo com busca linear?

Pode ser adequada se a quantidade de elementos for pequena.

E para o caso geral, podemos fazer uma implementação mais adequada?

Veremos a seguir.

Referências

Artigo Binary_search da Wikipédia.

Capítulo 3 - Pesquisa, ordenação e análise de complexidade - Fundamentos de Python: Estruturas de dados. Kenneth A. Lambert. (Disponível na Minha Biblioteca da UEM)

- Algoritmos de pesquisa / Pesquisa binária em uma lista ordenada

Capítulo 11 - Conjuntos e dicionários - Fundamentos de Python: Estruturas de dados. Kenneth A. Lambert. (Disponível na Minha Biblioteca da UEM)