

Tabelas de dispersão

Estruturas de Dados

Marco A L Barbosa

malbarbo.pro.br

Departamento de Informática

Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-Compartilhualgal 4.0 Internacional.

<http://github.com/malbarbo/na-ed>

Vimos que algumas vezes informações sobre características da entrada nos permitem desenvolver algoritmos e estrutura de dados mais eficientes. Por exemplo:

- Se um arranjo está ordenado, então podemos fazer uma busca binária ao invés de uma busca linear;
- Se as chaves para inserção em uma ABB são uniformemente distribuídas, então a altura da árvore será pequena e as operações serão eficientes.

Além disso, quando a entrada não tem aquelas características, ainda podemos tomar algumas providências:

- Modificar a entrada, como por exemplo, ordenar os valores para fazer a busca binária;
- Fazer o rebalanceamento em ABB, se as chaves não são uniformemente distribuídas, para manter a altura da árvore pequena.

Vimos que podemos implementar as operações de busca, inserção e remoção do TAD Dicionario usando árvores AVL com tempo $O(\lg n)$. Será que podemos fazer melhor se soubermos algo sobre as entradas (chaves)?

Na definição do TAD as chaves eram strings, e se as chaves fossem inteiros?

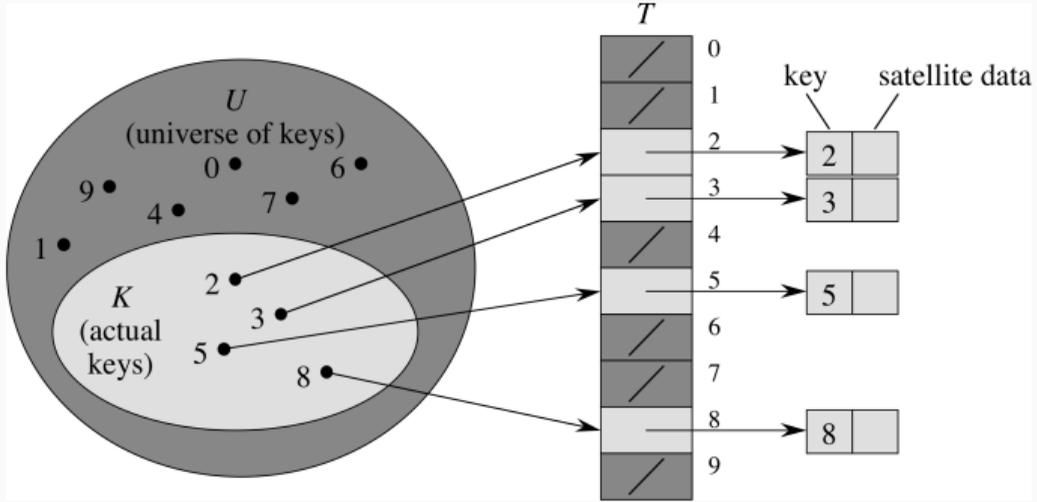
E se as chaves estivessem em um intervalo específico, entre 0 e 10.000?

Como tirar proveito desse conhecimento sobre as chaves para fazer uma implementação eficiente?

Podemos criar um arranjo com uma posição para cada possível chave da entrada, armazenamos na posição o valor associada com a chave, se houver tal valor, ou **None** caso contrário.

Essa estratégia é chamada de **endereçamento direto**.

De forma geral, para chaves no intervalo de 0 a $M - 1$, alocamos um arranjo com M posições.



Endereçamento direto

Implemente um dicionário que associa números naturais a strings usando endereçamento direto.

```
# As restrições (assert) sobre as
# chaves foram omitidas.
class Dicionario:
    valores: list[str | None]

    def __init__(self, m: int) -> None:
        assert 0 <= m
        self.valores = [None] * m

    def busca(self, chave: int) -> str | None:
        return self.valores[chave]

    def associa(self, chave: int, valor: str):
        self.valores[chave] = valor

    def remove(self, chave: int):
        self.valores[chave] = None
```

Qual é a complexidade de tempo das operações de dicionário usando endereçamento direto? $O(1)$.

Quais as desvantagens/limitações dessa estratégia?

- Se a quantidade de chaves é muito menor que M , então existe um desperdício muito grande de memória. Além disso, a quantidade de memória disponível pode não ser suficiente.
- As chaves precisam ser maiores que 0.
- As chaves são restritas a inteiros.

Vamos isolar e tentar lidar com cada uma dessas questões separadamente.

Intervalos quaisquer

O que podemos fazer se as chaves estão em um intervalo de $[A, B)$ qualquer?

Podemos *mapear* cada chave para um valor distinto no intervalo $[0, B - A)$:

$A \rightarrow 0, (A+1) \rightarrow 1, \dots, (B-1) \rightarrow (B-A-1)$.

Dessa forma podemos usar o resultado do mapeamento da chave como índice em um arranjo de tamanho $B - A$.

Projete uma função que faça o mapeamento de uma chave usando esse esquema.

```
def mapeia(chave: int, A: int, B: int) -> int:
    '''
    Mapeia uma chave no intervalo de
    *A* a *B* para um índice de um arranjo
    com *(B - A)* elementos.
    Requer que A <= chave < B.

    Exemplos
    >>> mapeia(-5, -5, 20)
    0
    >>> mapeia(19, -5, 20)
    24
    >>> mapeia(6, -5, 20)
    11
    '''
    assert A <= chave < B
    return chave - A
```

O que fazer quando a quantidade de chaves armazenadas n é muito menor que a quantidade de chaves possíveis U ?

Usar um arranjo de tamanho $m = O(n)$ e *mapear* as chaves para índices no intervalo de $[0, m)$ (índice válido para o arranjo).

Como mapear uma chave no intervalo de $[0, U)$ para posições no intervalo $[0, m)$?

Algumas opções (sendo k uma chave):

- Por divisão (resto): $k \bmod m$
- Por multiplicação: $\lfloor m \times (k \times A \bmod 1) \rfloor$, onde $A > 1$ é um valor real – note que $k \times A \bmod 1 < 1$.

Mapeie as chaves 734, 141, 8, 230, 554, 650 usando o esquema de divisão com $m = 20$.

734 \rightarrow 14, 141 \rightarrow 1, 8 \rightarrow 8, 230 \rightarrow 10, 554 \rightarrow 14, 650 \rightarrow 10.

Considerando que o resultado do mapeamento de uma chave é uma posição em um arranjo (onde o valor associado com a chave será armazenado), qual o problema que temos? Colisões!

Uma **colisão** ocorre quando duas chaves são mapeadas para a mesma posição do arranjo.

Podemos evitar as colisões? Sem conhecer as chaves, não.

Podemos lidar com as colisões? Sim, de algumas maneiras que vamos ver depois.

Considerando o mapeamento por divisão, qual a característica das chaves para que as colisões sejam mais raras? Elas devem estar uniformemente distribuídas.

E para que as colisões sejam frequentes? Que tenham o mesmo resto quando divididas por m .

O que fazer quando as chaves não são inteiros?

Mapear as chaves para valores inteiros e depois para posições. (Ou diretamente para posições)

Como mapear uma string para um inteiro?

Cada elemento em uma posição de uma string é internamente representado por um número (*code point*), esse número pode ser obtido com a função `ord`.

```
>>> ord('a')           >>> ord('z')           >>> ord('7')           >>> ord('%')
97                     122                     55                     37
```

Então, podemos, por exemplo, mapear uma string para o *code point* do seu primeiro caractere, ou zero se a string for vazia. Ou ainda, somar todos os *code point* de todos os caracteres.

Qual o problema dessas formas de mapeamento? Geram muitas colisões!

Mas não vamos nos preocupar com isso por enquanto, basta sabermos que é possível mapear *qualquer* tipo de valor para um número inteiro.

Vimos que, se as chaves são inteiros em um intervalo de 0 a $M - 1$, então podemos implementar um dicionário usando endereçamento direto, onde as operações de busca, inserção e remoção tem complexidade de tempo de $O(1)$.

Vimos as limitações do endereçamento direto e discutimos estratégias de como superá-las. O que essas estratégias tinham em comum?

- O mapeamento da chave para uma posição de um arranjo com tamanho conhecido

Em outras palavras, a estratégia é a mesma!

Chamamos a função que mapeia as chaves para posições de um arranjo de **função de dispersão** ou **função *hash***.

Uma **tabela de dispersão** ou **tabela *hash*** é uma estrutura de dados que usa uma função de dispersão para calcular índices em um arranjo que fornece uma forma de armazenar pares de chave-valor.

Existem dois desafios no projeto e implementação de uma tabela de dispersão:

- A função de dispersão
- O tratamento de colisões

Função de dispersão

Criar uma boa função de dispersão, isto é, uma função que gere poucas colisões, requer conhecimentos avançados de probabilidade e estatística, por isso não vamos tratar desse assunto.

Como função de dispersão, vamos combinar a função `hash`, pré-definida em Python, com o resto da divisão. Para uma chave k , vamos representar o resultado da função de dispersão por $h(k)$.

```
>>> hash('casa')
-3155579165809741514
>>> hash('arroz')
-5974344979373615551
>>> hash(7)
7
>>> hash(-17)
-17
>>> hash(2 ** 100)
549755813888
```

```
>>> hash('casa') % 20
6
>>> hash('arroz') % 20
9
>>> hash(7) % 20
7
>>> hash(-17) % 20
3
>>> hash(2 ** 100) % 20
8
```

Considere os pares de chave-valor: (734, 'maça'), (141, 'mamão'), (84, 'banana'), (236, 'goiaba'), (554, 'ameixa'), (1, 'laranja').

Considerando um tabela (arranjo) de $m = 10$ posições, calcule $h(k) = k \bmod m$ para cada chave k listada anteriormente.

Proponha uma forma de lidar com as colisões, isto é, uma maneira de armazenar, busca, inserir e remover os pares chave-valor na tabela.

Podemos armazenar todas os pares chave-valor cuja a chave gerou o mesmo índice em um coleção:

- Arranjo dinâmico
- Lista encadeada
- Árvore AVL
- Outra tabela de dispersão!

Quando usamos uma lista encadeada em cada posição, chamamos a estratégia **encadeamento separado**.

Encadeamento separado

```

      +----+
(734, 'maça') -> 4  0 |  |
                    +----+ +-----+ +-----+
(141, 'mamão') -> 1  1 |  |-->| 141 mamão |-->| 1 laranja |
                    +----+ +-----+ +-----+
(84, 'banana') -> 4  2 |  |
                    +----+
(236, 'goiaba') -> 6  3 |  |
                    +----+ +-----+ +-----+ +-----+
(554, 'ameixa') -> 4  4 |  |-->| 734 maçã |-->| 84 banana |-->| 554 ameixa|
                    +----+ +-----+ +-----+ +-----+
(1, 'laranja') -> 1  5 |  |
                    +----+ +-----+
6 |  |-->| 236 goiaba|
                    +----+ +-----+
7 |  |
                    +----+
8 |  |
                    +----+
9 |  |
                    +----+
```

Como definir os tipos para implementar um dicionário usando uma tabela de dispersão com encadeamento?

```
@dataclass
class No:
    chave: str
    valor: int
    prox: No | None

class Dicionario:
    tabela: list[No | None]
    num_itens: int
```

Como implementar o método **busca**?

```
def busca(self, chave: str) -> int | None:
    # procurar por chave em self.tabela[h(chave)]
```

Como implementar **associa**?

```
def associa(self, chave: str, valor: int):
    # inserir (chave, valor) em self.tabela[h(chave)]
    # ou atualizar o valor associado com a chave
```

Como implementar **remove**?

```
def remove(self, chave: str):
    # remover (chave, valor) de self.tabela[h(chave)]
```

Qual é a complexidade de tempo de **busca**, **associa** e **remove**? Depende da quantidade de itens no encadeamento...

Para discutirmos a complexidade de tempo, precisamos de uma definição.

Chamamos de **fator de carga** α de uma tabela de dispersão T o valor n/m , onde m é a quantidade de posições na tabela e n é a quantidade de elementos em T .

Qual é o pior caso para as operações? Todos os n elementos estão na mesma posição da tabela. Nesse caso, o tempo das operações é $O(n)$.

E o caso médio? Qual o tamanho médio de cada lista encadeada? $n/m = \alpha$, ou seja, no caso médio, o tempo das operações é $O(1 + \alpha)$.

Se mantermos $n = O(m)$, então $\alpha = n/m = O(m)/m = O(1)$, e o tempo médio das operações fica $O(1)$.

Para manter o tempo médio em $O(1)$, temos que manter um fator de carga pequeno. Mas não muito pequeno para não desperdiçar memória.

Sedgewick recomenda um valor entre 5 e 10.

Então quanto α fica maior que 10, temos que alocar uma tabela *maior* e fazer a redistribuição das chaves.

Quanto α fica menor que 5, temos que alocar uma tabela *menor* e fazer a redistribuição das chaves.

Endereçamento aberto é uma técnica de resolução de colisão baseado em **sondagem**.

Nessa técnica todos os pares chave-valor são armazenados na própria tabela.

Quando um novo par chave-valor precisa ser inserido na tabela e a posição já está ocupada, outras posições são sondadas até que uma posição livre seja encontrada.

A busca e a remoção devem usar o mesmo processo de sondagem.

A forma mais simples de sondagem é a linear. Nesse esquema, quando há colisão na posição i , as posições são testadas na ordem $(i + j) \bmod m$ para $j = 1, 2, \dots, \dots m$.

A sondagem para quando uma posição que nunca foi ocupada é encontrada.

Veremos que para esse esquema funcionar, a remoção deve marcar de forma especial a posição.

Sondagem linear - inserção

Mostre passo a passo a inserção das chaves 734, 84, 236, 554, 141 em uma tabela com $m = 8$.

734 → 6

84 → 4

236 → 4, 5

554 → 2

31 → 7

141 → 5, 6, 7, 0

	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	0	0	0	0	0	0 141
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	1	1	1	1	1	1
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	2	2	2	2 554	2 554	2 554
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	3	3	3	3	3	3
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	4	4 84	4 84	4 84	4 84	4 84
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	5	5	5 236	5 236	5 236	5 236
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	6 734	6 734	6 734	6 734	6 734	6 734
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	7	7	7	7	7 31	7 31
	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+

Sondagem linear - remoção e busca

Mostre passo a passo para a remoção da chave 236, e a busca das chave 742 e 141?

	Remoção do		Busca do	Busca do
0 141	236 → 4, 5.	0 141	742 → 6, 7, 0, 1.	141 → 5, 6, 7, 0.
1	A posição de	1	Não encontrado.	Encontrado.
2 554	remoção deve ser	2 554		
3	marcada forma	3		
4 84	especial.	4 84		
5 236		5 \		
6 734		6 734		
7 31		7 31		

Sondagem linear

Defina os tipos de dados para um dicionário implementado usando tabelas de dispersão com sondagem.

```
@dataclass
class Removido:
    pass

@dataclass
class Presente:
    chave: str
    valor: int

class Dicionario:
    tabela: list[None | Removido | Presente]
    # Qtd de itens Presente na tabela.
    num_itens: int
    # Qtd de itens Removido na tabela.
    num_removidos: int
```

Implemente o método `Dicionario.busca`.

```
def busca(self, chave: str) -> None | int:
    p = hash(chave) % len(self.tabela)
    while self.tabela[p] is not None:
        if isinstance(self.tabela[p], Presente):
            if self.tabela[p].chave == chave:
                return self.tabela[p].valor
        p = (p + 1) % len(self.tabela)
    return None
```

Note que para a busca parar no caso que a chave não é encontrada é preciso que exista pelo menos uma posição com `None`, ou seja, a quantidade de itens presentes mais os removidos deve ser menor o tamanho da tabela.

Qual o tempo de execução das operações de busca, inserção e remoção?

No pior caso, $O(n)$.

Mas se o fator de carga for mantido menor do que 0.7, a complexidade de tempo no caso médio é $O(1)$.

Capítulo 11 - Seção Estratégias de hashing - Fundamentos de Python: Estruturas de dados.
Kenneth A. Lambert. (Disponível na Minha Biblioteca na UEM).

Capítulo 11 - Tabelas de Dispersão - Algoritmos: Teoria e Prática, 3a. edição, Cormen, T. et al.

Capítulo 5 - Hash Tables - [Open Data Structures](#).

Funções [hash](#) e [object.__hash__](#) do Python.