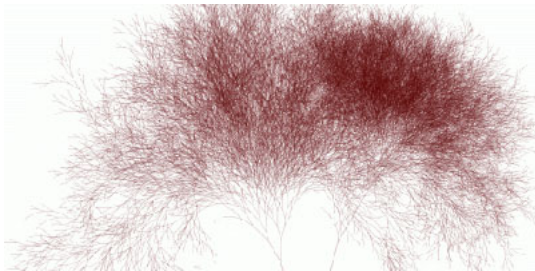




Fonte: ash.atozviews.com

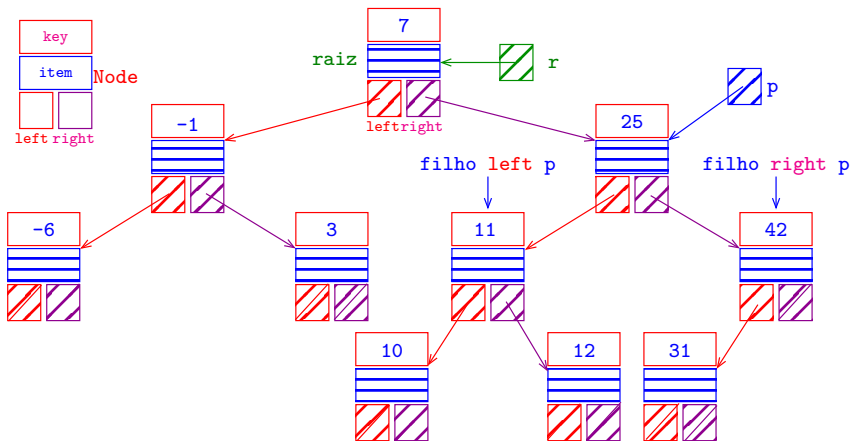
Compacto dos melhores momentos das últimas aulas

Árvores binárias de busca



Fonte: <http://infosthetics.com/archives/>

Árvore binárias de busca



in-ordem (e-r-d): -6 -1 3 7 10 11 12 25 31 42

Consumo de tempo

O consumo de tempo das operações `get()`, `put()` e `delete()` é, no **pio**r caso, proporcional à **altu**ra da **árvore**.

Consumo de tempo no pior caso

No pior caso a altura de um BST é proporcional ao número n de nós BST.

Conclusão:

O consumo de tempo das operações `get()`, `put()` e `delete()` em uma BST é, no pior caso, proporcional ao número n de nós.

Consumo de tempo **esperado**

A **altura esperada** de **BST aleatória** é aproximadamente $2 \lg n$.

Conclusão:

O consumo de tempo **esperado** das operações **get()**, **put()** e **delete()** em uma **BST aleatória** é proporcional $\lg n$, onde **n** é o número de nós.

Árvores binárias de busca ótima

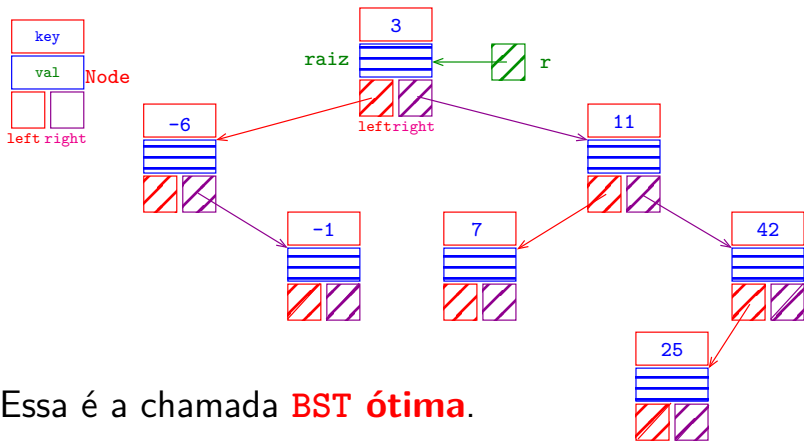
Devemos realizar um série de operação `get()` em uma **BST** com chaves

$$\text{key}[0] < \text{key}[1] < \text{key}[2] < \dots < \text{key}[n-1].$$

Busca bem-sucedida: suponha que $p[i]$ é a probabilidade de `key[i]` ser argumento de `get()`.

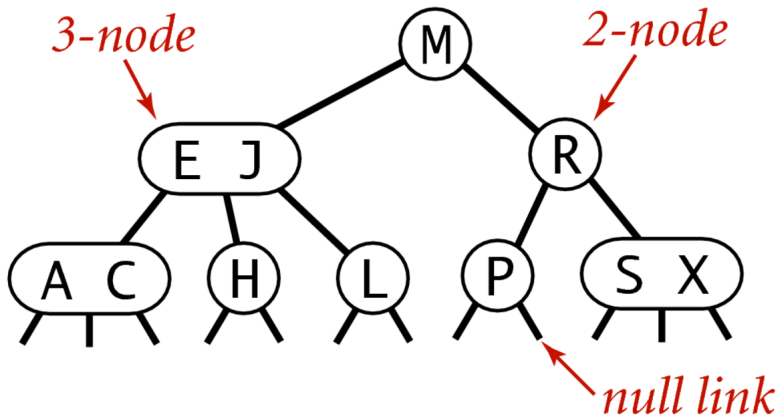
Problema. construir uma **BST** que **minimize** o número esperado de comparações.

BST ótima



$$0.2 + 2(0.22 + 0.25) + 3(0.18 + 0.05 + 0.08) + 4(0.02) = 2.15 \text{ comparações}$$

Árvore 2-3 de busca



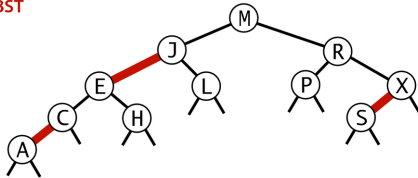
Anatomy of a 2-3 search tree

Consumo de tempo

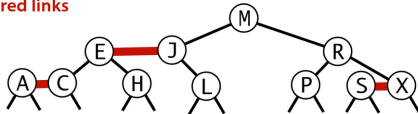
Numa **árvore 2-3** com **n** nós, **busca** e **inserção** nunca visitam mais que $\lg(n+1)$. Cada visita faz no **máximo 2 comparações** de chaves.

BST rubro-negra

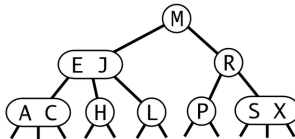
red-black BST



horizontal red links



2-3 tree

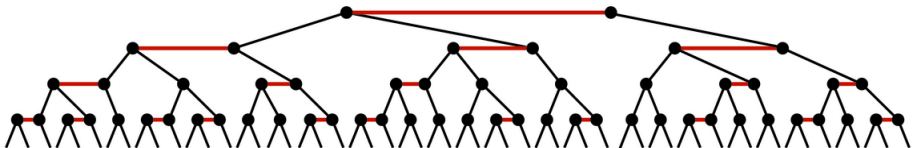


Fonte: [algs4](#)

1-1 correspondence between red-black BSTs and 2-3 trees

Árvore 2-3 para rubro-negra

Se os **links rubros** forem desenhados horizontalmente e depois contraídos, teremos uma **árvore 2-3**:



A red-black tree with horizontal red links is a 2-3 tree

Fonte: [algs4](#)

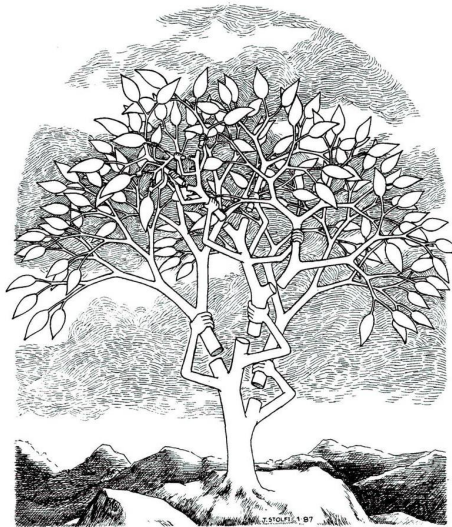
Consumo de tempo

A altura esperada de **BST rubro-negra** é aproximadamente $\leq 2 \lg n$.

Conclusão:

O consumo de tempo das operações **get()**, **put()** e **delete()** em uma **BST rubro-negra** é $O(\lg n)$.

Self-adjusting BSTs



Fonte: [Jorge Stolfi](#)

Self-adjusting BSTs

Uma **BST** é de **auto balanceamento/ajuste** (*self-balancing/self-adjusting*) se **automaticamente** mantém a sua altura pequena diante de uma sequência de operações `put()`, `get()`, ...

Árvores **rubro-negras** são **BSTs** de **auto balanceamento**.

Splay trees

Uma **splay tree** é uma **BST** de **auto-balanceamento** com a **propriedade adicional** que os elementos **acessados recentemente** são **rapidamente acessados**.

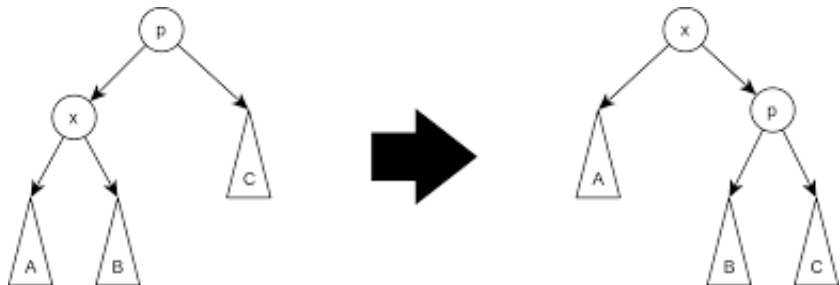
Splay trees implementam em **BSTs** a política *move to front*.

Splay trees

Uma **splay tree** é uma **BST** de **auto-balanceamento** com a **propriedade adicional** que os elementos **acessados recentemente** são **rapidamente acessados**.

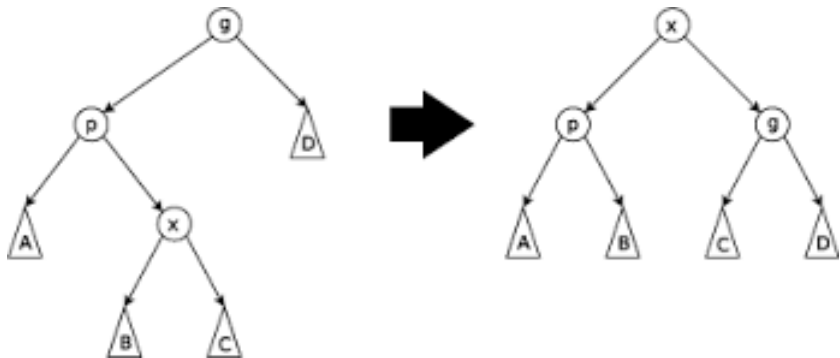
Splay trees implementam em **BSTs** a política *move to front*.

Splaying: zig



Fonte: [Wikipedia](#)

Splaying: zig-zag



Fonte: [Wikipedia](#)

Resumo

estrutura	consumo de tempo <code>get()</code> , <code>put()</code> , ...	observação
Skip list	$O(\lg n)$	esperado
BST	$O(n)$	pior caso
BST-aleatória	$O(\lg n)$	esperado
2-3 ST	$O(\lg n)$	pior caso
RedBlack BST	$O(\lg n)$	pior caso
Splay BST	$O(\lg n)$	amortizado
Treap BST	$O(\lg n)$	esperado

n = número de nós na estrutura

Mais experimentos ainda

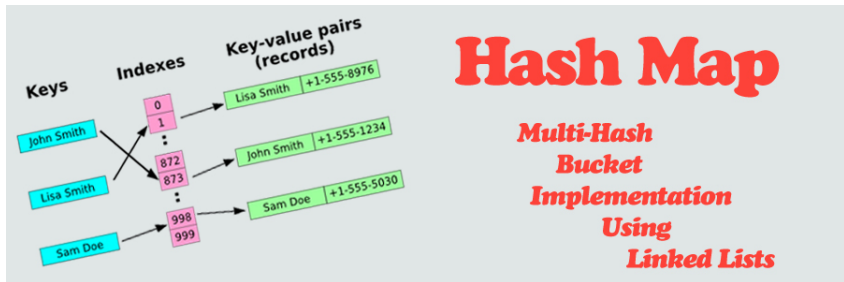
Consumo de tempo para se criar um **ST** em que a **chaves** são as palavras em `les_miserables.txt` e os **valores** o número de ocorrências.

estrutura	ST	tempo
vetor MTF	não-ordenada	7.6
vetor	ordenada	1.5
lista ligada MTF	não-ordenada	15.3
skiplist	ordenada	1.1
árvore binária de busca	ordenada	0.72
árvore rubro-negra	ordenada	0.76
splay tree	ordenada	0.68

Tempos em **segundos** obtidos com **StopWatch**.

AULA 13

Hashing



Fonte: <http://programmingnotes.freeweq.com>

Referências: Hashing (PF); Hash Tables (S&W); slides (S&W); Hashing Functions (S&W); CLRS, cap 12; TAOP, vol 3, cap. 6.4;

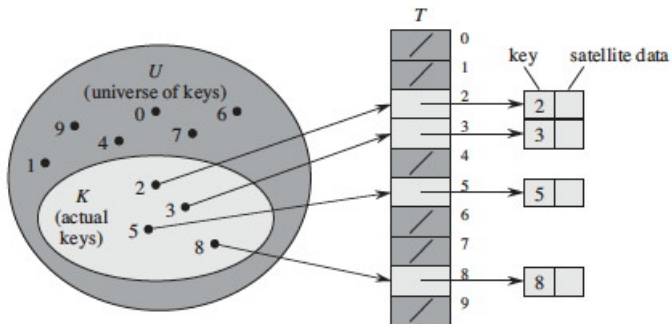
Endereçamento direto

Endereçamento direto (*directed-address*) é uma técnica que funciona bem quando o universo de chaves é **razoavelmente pequeno**.

Tabela indexada pelas **chaves**, uma posição para cada possível **índice**.

Cada posição armazena o **valor** correspondente a uma dada **chave**.

Endereçamento direto



Fonte: CLRS

Endereçamento direto

```
public class DirectAddressST<Value> {  
    private Value[] vals;  
  
    public DirectAddressST(int m) {  
        vals = (Value[]) new Object[m];  
    }  
}
```

Endereçamento direto

```
public Value get(Key key) {  
    return vals[key];  
}
```

```
public void put(Key key, Value val) {  
    vals[key] = val;  
}
```

```
public void delete(Key key) {  
    vals[key] = null;  
}
```

```
}
```

Consumo de tempo

Em uma **tabela se símbolos** com **endereçamento direto** o consumo de tempo de **get()**, **put()** e **delete()** é **$O(1)$** .

Maiores defeitos

Os **maiores defeitos** dessa implementação são:

- ▶ Em geral, as **chaves não são inteiros** não-negativos pequenos. . .
- ▶ **desperdício de espaço**: é possível que a maior parte da tabela fique vazia

Tabelas de dispersão (*hash tables*)

Uma **tabela de dispersão** (= *hash table*) é uma maneira de organizar uma tabela de símbolos.

Inventadas para funcionar bem (em $O(1)$) em média.

universo de chaves = conjunto de **todas** as possíveis **chaves**

chaves realmente usadas são, em geral, uma **parte pequena** do **universo**.

A tabela terá a forma $st[0 \dots m-1]$, onde m é o tamanho da tabela.

Funções de dispersão

Uma **função de dispersão** (= *hash function*) é uma maneira de mapear o **universo de chaves** no conjunto de **índices** da tabela.

A **função de dispersão** recebe uma **chave** `key` e retorna um número inteiro $h(\text{key})$ no intervalo $0 \dots m-1$.

O número $h(\text{key})$ é o **código de dispersão** (= *hash code*) da chave.

Queremos uma função de hashing que ...

Queremos uma **função de hashing** que:

- ▶ possa ser **calculada eficientemente** (em $O(1)$) e
- ▶ **espalhe bem as chaves** pelo intervalo $0, \dots, m-1$.

Knuth, TAOC, pg. 514:

*“The verb ‘**to hash**’ means to chop something up to make a mess out of it; the idea in **hashing** is to scramble some aspects of the key and to use this partial information as basis for searching... ”*

Funções injetoras...

Funções que associam **chaves diferentes** a inteiros diferentes são difíceis de se encontrar.

Mesmo se **conhecêssemos as chaves de antemão!**

Exemplo:

Existem $41^{31} \equiv 10^{50}$ funções de **31** elementos em **41** elementos e somente $41!/10! \equiv 10^{43}$ são **injetoras**: uma em cada 10 milhões!

Funções injetoras...

Funções que associam **chaves diferentes** a inteiros diferentes são difíceis de se encontrar.

Mesmo se **conhecêssemos as chaves de antemão!**

Mesmo se o **tamanho da tabela** for **razoavelmente maior** que o **número de chaves**.

O **paradoxo do aniversário** nos diz se selecionarmos uniformemente ao acaso uma função que leva **23 chaves** em uma tabela de **tamanho 365**, a probabilidade de que duas chaves sejam associadas a uma mesma posição é **maior 0,5**.

Conclusão: temos que conviver com **colisões**.

Função de hashing modular

Método da divisão (*division method*) ou hash modular: supondo que as **chaves são inteiros positivos**, podemos usar a função modular (resto da divisão por m):

```
private int hash(int key) {  
    return key % m;  
}
```

Função de hashing modular

Exemplos com $m = 100$ e com $m = 97$:

key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25

Fonte: [algs4](#)

Função de hashing modular

No caso de **Strings**, podemos iterar **hashing modular** sobre os caracteres da string:

```
private int hash(String key) {  
    int h = 0;  
    for (int i = 0; i < key.length(); i++)  
        h = (31 * h + key.charAt(i)) % m;  
    return h;  
}
```

Função de hashing modular

Vantagens: rápida, faz apenas uma divisão.

Desvantagem: devemos evitar certos valores para m , por exemplo:

- ▶ se $m = 2^p$, então $h(\text{key})$ é os p bits menos significativos de key .
- ▶ se string de caracteres é interpretado como números na base 2^p , então $m = 2^p - 1$ é uma má escolha: permutações de caracteres são levadas ao mesmo valor de hash.

Um **primo** não “muito perto” de um potência de 2 parece ser uma boa escolha para m .

Função Multiplicativa

Método multiplicativo (*multiplicative method*):

- ▶ escolha uma constante A , $0 < A < 1$;
- ▶ multiplique key por A ;
- ▶ extraia a parte fracional de $key \times A$
- ▶ multiplique a parte fracionária por m
- ▶ o valor de hash é o chão dessa multiplicação

Função Multiplicativa

Desvantagem: mais lenta que o hash modular

Vantagem: o valor de m não é crucial

O que Ubuntu tem a dizer...

<http://releases.ubuntu.com/17.10/>

MD5SUMS	2018-01-12	05:38	198
MD5SUMS-metalink	2018-01-12	05:38	213
MD5SUMS-metalink.gpg	2018-01-12	05:38	916
MD5SUMS.gpg	2018-01-12	05:38	916
SHA1SUMS	2018-01-12	05:38	222
SHA1SUMS.gpg	2018-01-12	05:38	916
SHA256SUMS	2018-01-12	05:38	294
SHA256SUMS.gpg	2018-01-12	05:38	916

O que Ubuntu tem a dizer...

<https://en.wikipedia.org/wiki/MD5>

```
773c839d24cf91c394aca6f1b9cd40da *ubuntu-17.10.1-desktop-amd64.iso  
7fe25fa47bebc40f4e5007aa182eb627 *ubuntu-17.10.1-server-amd64.iso  
f713724032a1b0fdbf3ebd90d2eec8d8 *ubuntu-17.10.1-server-i386.iso
```

O que Ubuntu tem a dizer...

<https://en.wikipedia.org/wiki/SHA-2>

1a3d2d32ada795e5df47293745a7479bcb3e4e29d8ee1eaa114350b691cf38d3 *ubun
8ff73f1b622276758475c3bd5190b382774626de5a82c50930519381f6c3a3f8 *ubun
eb921425349d2a51f90edc3977f83fb6fd8aed082b31515f1bde00d46b260492 *ubun

O que Java tem a dizer

Em Java, **toda classe** tem um método padrão `hashCode()` que produz um inteiro entre -2^{31} e $2^{31} - 1$,

Exemplo:

```
String s = StdIn.readString();  
int h = s.hashCode();
```

O que Java tem a dizer

Outro exemplo:

```
public class Teste {  
    private int val;  
  
    public Teste(int val) {  
        this.val = val;  
    }  
}
```

O que Java tem a dizer

Welcome to DrJava.

```
> Teste t = new Teste(5)
```

```
> t.hashCode()
```

```
22767675
```

```
> Teste t = new Teste(6)
```

```
> t.hashCode()
```

```
27103358
```

```
> Teste r = new Teste(27)
```

```
> r.hashCode()
```

```
5836093
```

O que Java tem a dizer

`hashCode()` deve ser consistente com `equals()`:

se `a.equals(b) == true`, então

`a.hashCode() == b.hashCode()`

Para converter o `hashCode()` em um número entre 0 e $m-1$, tome o resto da divisão por m .

Antes, é melhor desprezar o bit mais significativo para evitar que % lide com números negativos e produza um resultado negativo:

```
private int hash(Key key) {  
    return(key.hashCode() & 0x7fffffff)% m;  
}
```


O que Java tem a dizer

Chaves `Integer`

Em Java o valor de hash de um `int` é ele mesmo:

```
public class Integer {  
    private final int val;  
    public int hashCode() {  
        return val;  
    }  
}
```

O que Java tem a dizer

Welcome to DrJava.

```
> Integer i = 15
```

```
> i.hashCode()
```

```
15
```

```
> i = -15
```

```
-15
```

```
> i.hashCode()
```

```
-15
```

O que Java tem a dizer

Chaves `Double`

Para `Doubles` o Java usa hashing modular na representação binária do `double`.

```
public class Double {  
    private final double value;  
    public int hashCode() {  
        long bits=doubleToLongBits(value);  
        return (int)(bits^(bits >> 32));  
    }  
}
```

O que Java tem a dizer

Chaves **Boolean**

```
public class Boolean {  
    private final boolean val;  
    public int hashCode() {  
        if (val) return 1231;  
        return 1237;  
    }  
}
```

1231 e 1237 são primos.

O que Java tem a dizer

Chaves `String`

Consideramos o string como um número muito grande na base $R (= 31)$.

Se s é um `String` de comprimento k , `s.hashCode()` é o valor

$$s[0] \times 31^{k-1} + s[1] \times 31^{k-2} + \dots + s[k-2] \times 31 + s[k-1].$$

O método `hashCode()` utiliza o **método de Horn** para calcular esse valor.

O que Java tem a dizer

```
int h = 0;
for (int i = 0; i < s.length(); i++)
    h = (31 * h + s.charAt(i)) % m;
```

No lugar do **multiplicador 31**, poderia usar qualquer outro inteiro **R**, de preferência primo, mas suficientemente pequeno para que os cálculos não **produzam overflow**.

O que Java tem a dizer

Welcome to DrJava.

```
> "a".hashCode()
```

97

```
> "Como é bom estudar MAC0323!".hashCode()
```

-638314223

```
> ("Como é bom estudar  
MAC0323!".hashCode())&0x7fffffff
```

1509169425

```
> Float x = (float)3.14
```

```
> x.hashCode()
```

1078523331

Boas e más funções de dispersão

Uma função só é **eficiente** se **espalha** as chaves pelo intervalo de índices de maneira *razoavelmente uniforme*.

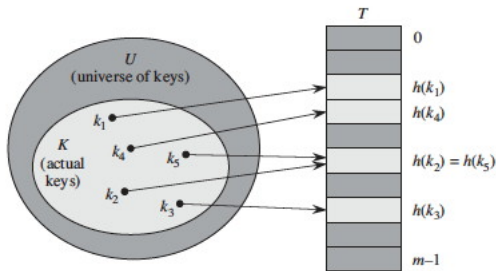
Por exemplos, se os **dois últimos dígitos** da chaves não variam muito, então “**key** % 100” é um **péssima** função de dispersão.

Em geral é recomendável que **m** seja um número **primo**.

Escolha de funções de dispersão é uma **combinação** de **estatística**, **probabilidade**, **teoria dos números (primalidade)**, . . . ,

Colisões

Como o número de chaves é em geral maior que m , é inevitável que a função de dispersão leve várias chaves diferentes no mesmo índice.



Fonte: CLRS

Colisões

Dizemos que há uma **colisão** quando duas **chaves diferentes** são levadas no **mesmo índice**.

Algumas maneiras de tratar colisões:

- ▶ **lista encadeadas** (= *separating chaining*);
- ▶ **sondagem linear** (= *linear probing (open addressing)*);
- ▶ **double hashing** (*open addressing*);