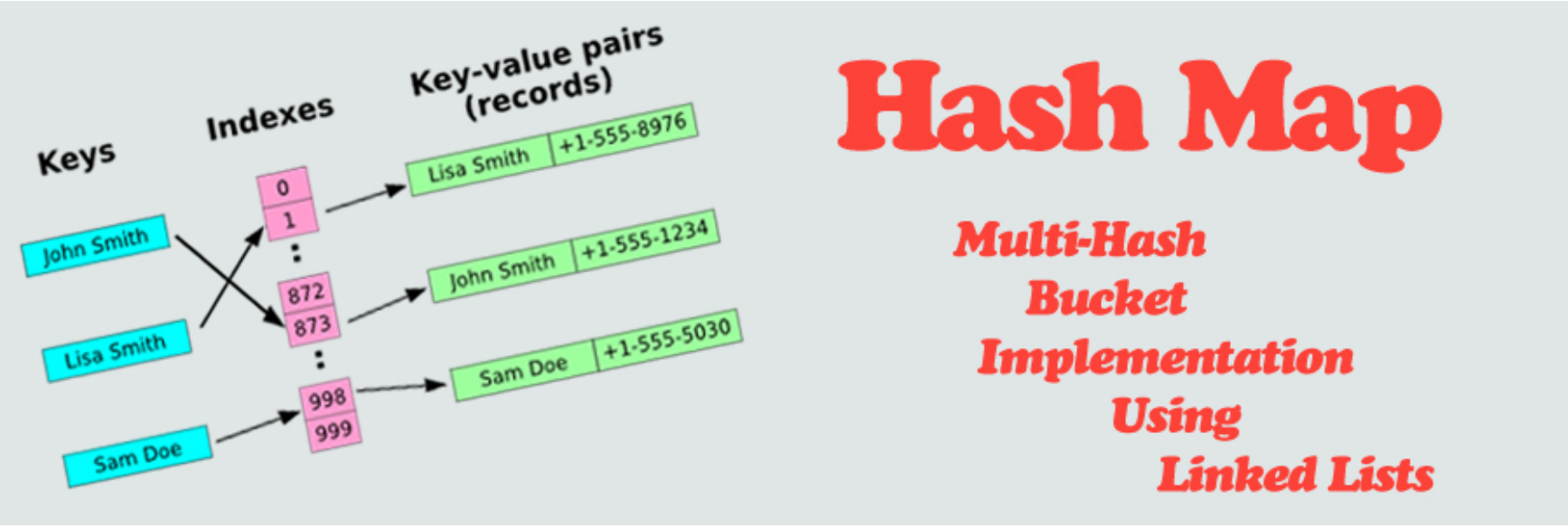
AULA 13

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;

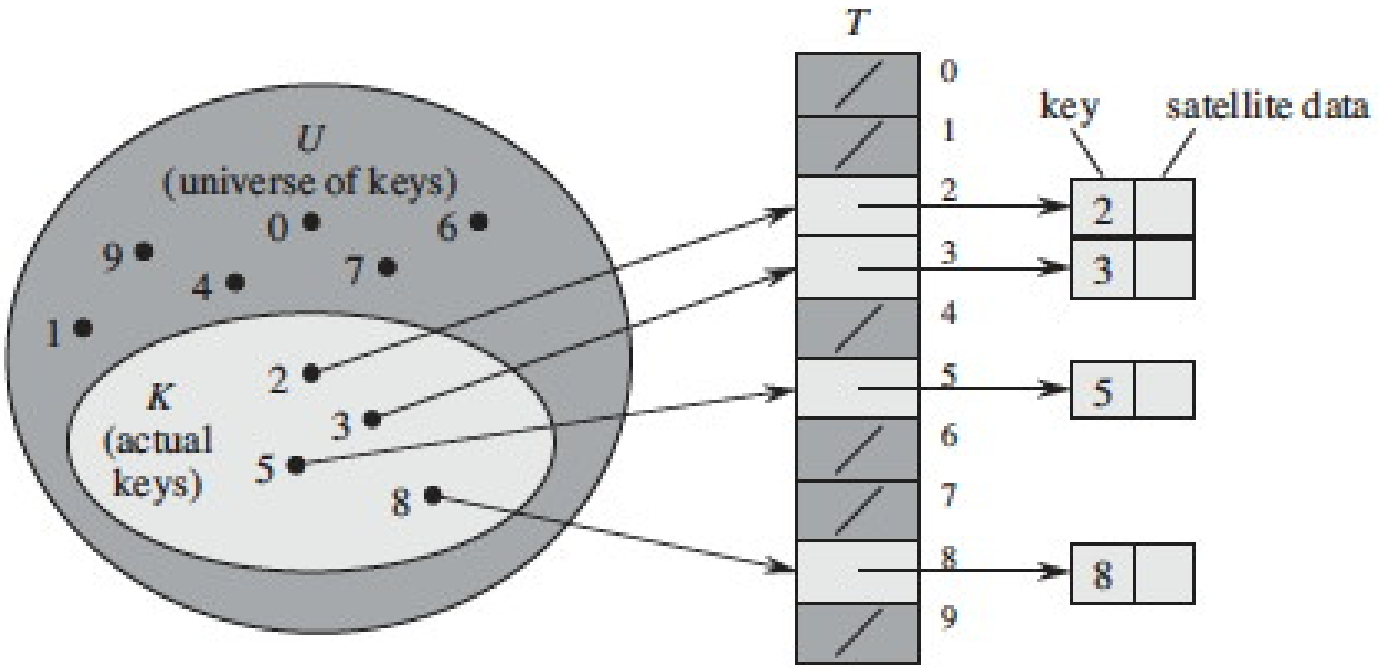
Endereçamento direto

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 arnazena 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 dispensão** (= *hash table*) é uma maneira de organizar uma tabela de símbolos.

Inventadas para funcionar bem (em $\mathrm{O}(1)$) 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...`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(key)` no intervalo 0 . . `m`-1.

O número `h(key)` é o **código de dispensão** (= *hash code*) da chave.

Queremos uma função de hashing que . . .

Queremos uma função de hashing que:

- possa ser calculada eficientemente (em $\mathrm{O}(1)$) e
- espalhe *bem* as chaves pelo intervalo $0, \ldots, \mathtt{m}-1$.

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 que 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 que 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 não sejam associadas a mesma posição é menor que 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 $\mathtt{m}$, por exemplo:

- se $\mathtt{m} = 2^{\mathtt{p}}$, então $\mathtt{h}(\mathtt{key})$ é os $\mathtt{p}$ bits menos significativos de $\mathtt{key}$.
- se string de caracteres é interpretado como números na base $2^{\mathtt{p}}$, então $\mathtt{m} = 2^{\mathtt{p}} - 1$ 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 $\mathtt{m}$.

Função Multiplicativa

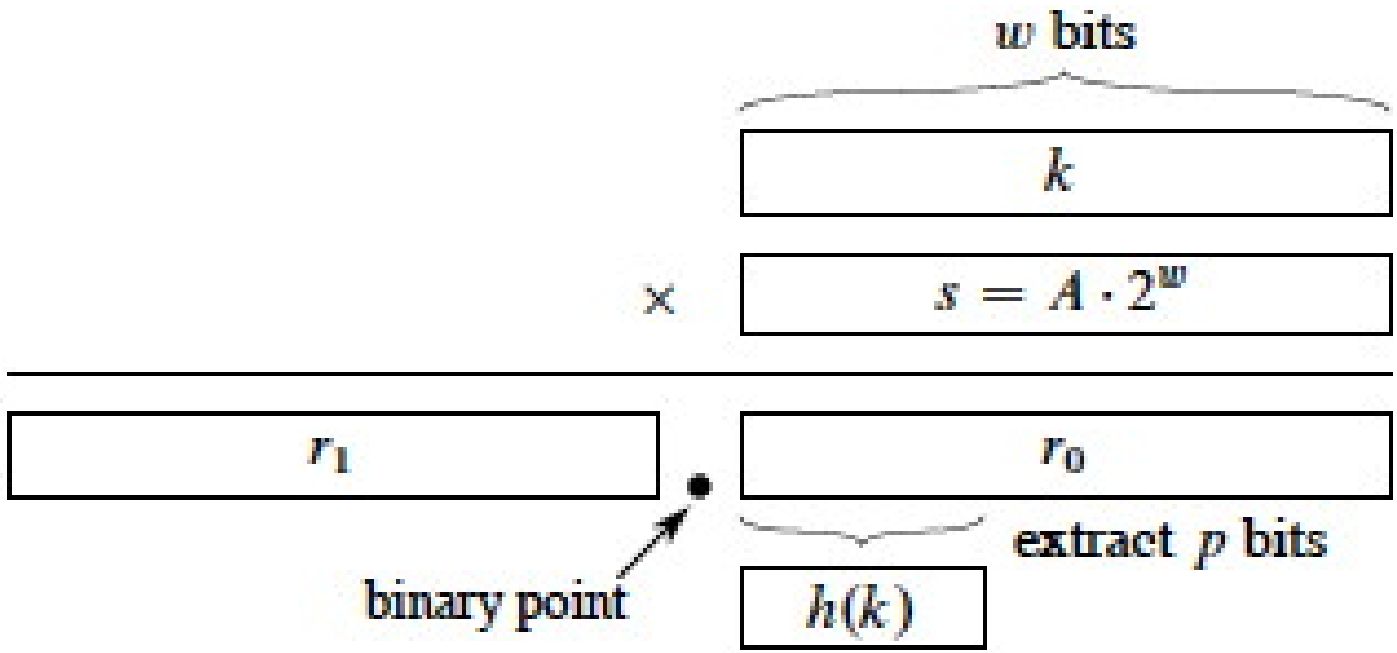
Método multiplicativo (*multiplicative method*):

- escolha uma constante `A` $0 < \texttt{A} < 1$;
- 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

Nesse caso `m` é uma potência de 2.

Assim, `h(key)` contém os bits iniciais da metade menos significa de `key` $\times$ `A`.



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 *ubur
8ff73f1b622276758475c3bd5190b382774626de5a82c50930519381f6c3a3f8 *ubur
eb921425349d2a51f90edc3977f83fb6fd8aed082b31515f1bde00d46b260492 *ubur
```

O que Java tem a dizer

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

Exemplo:

```
String s = StdIn.readInt();
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 sigificativo para evitar que % lide com números negativos e produza um resultado negativo:

```
private int hash(Keykey) {
   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

   public int hashCode() {
       return
   }
}
```

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 `Double`s 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

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

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*.

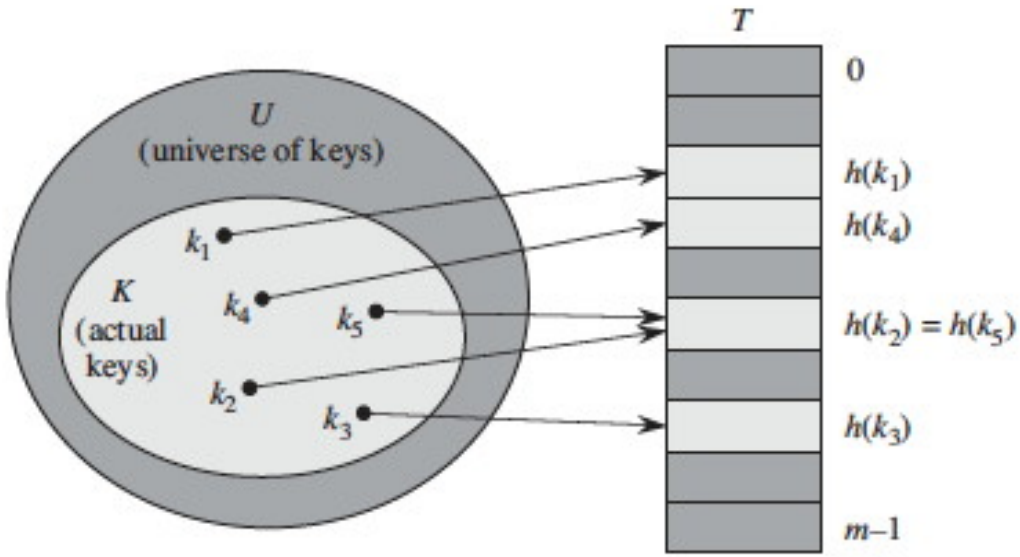
Por exemplos, se os dois últimos dígitos da chaves não variam muito, então "`key`% 100" é um péssima função de dispersão.

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)*;

Colisões por listas encadeadas

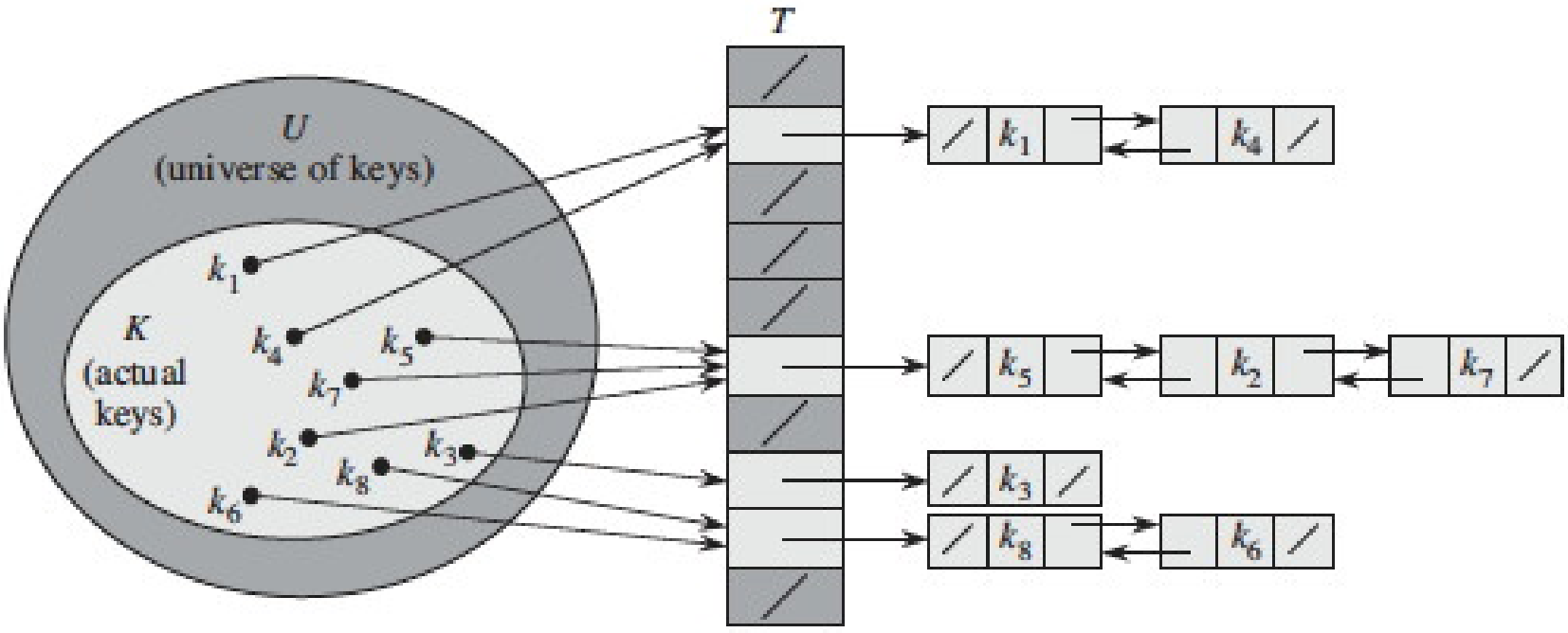
Uma solução popular para resolver colisões é conhecida como **separate chaining**:

> *para cada índice* h *da tabela há uma lista encadeada que armazena todos os objetos que a função de dispersão leva em* h.

Essa solução é muito boa se cada uma das "listas de colisão" resultar curta.

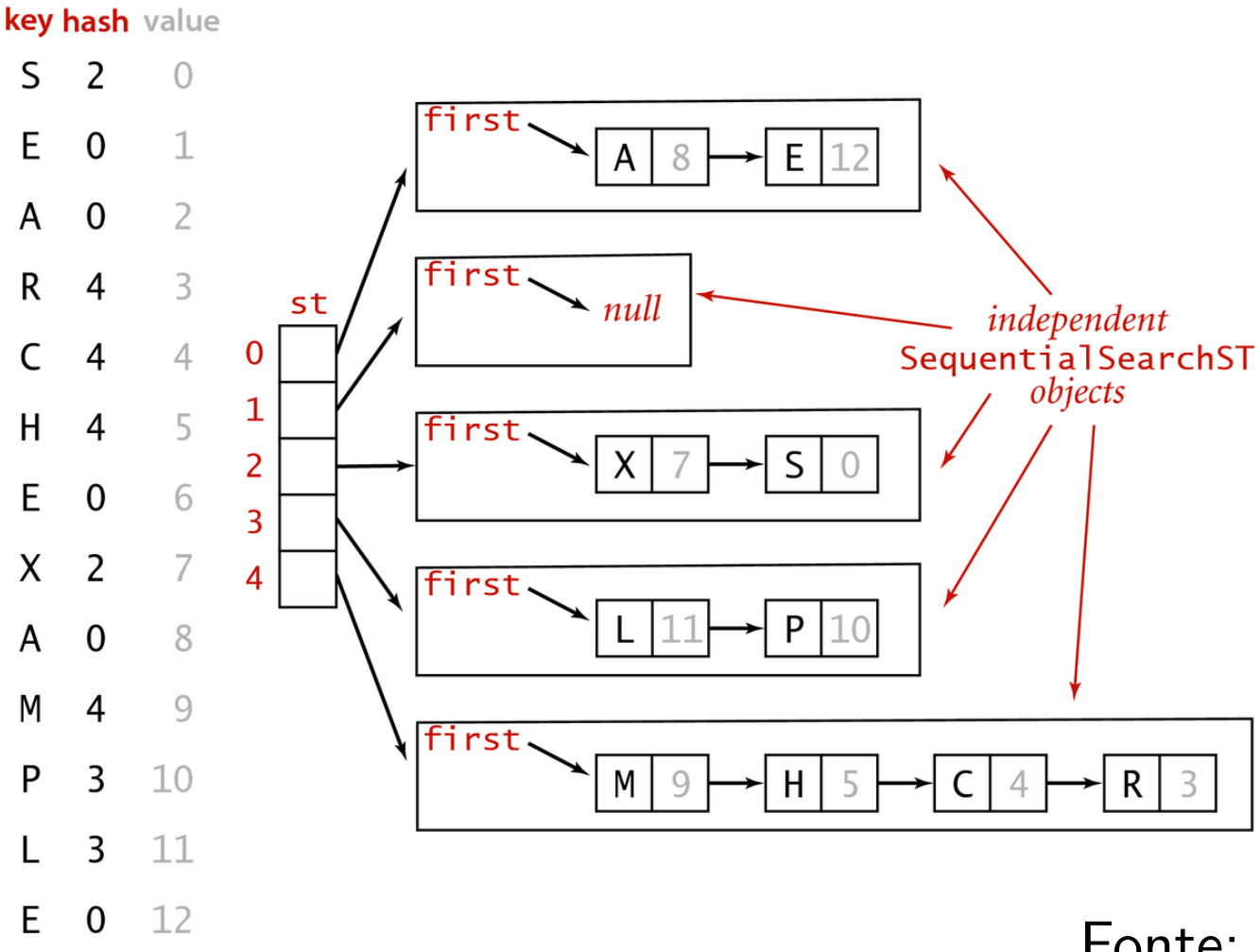
Se o número total de chaves usadas for $\mathtt{n}$, o comprimento de cada lista deveria, idealmente, estar próximo de $\alpha = \mathtt{n}/\mathtt{m}$.
O valor α é chamado de **fator de carga** ((= *load factor*) da tabela.

Colisões por listas encadeadas



Fonte: CLRS

Colisões por listas encadeadas



Fonte: algs4

Hashing with separate chaining for standard indexing client

Colisões por listas encadeadas

Classe SequentialSearchST: implementação de tabela de símbolos em uma lista ligada não ordenada

```
public class SeparateChainingHashST<Key, Value>{
   private int n; // número de chaves

   private int m; // tam da tabela de hash

   // vetor de TSs
   private SequentialSearchST<Key,Value>[] st;
```

Colisões por listas encadeadas

```
public SeparateChainingHashST(int m) {
   this.m = m;
   st = (SequentialSearchST<Key,Value>[])
            new SequentialSearchST[m];
   for (int i = 0; i < m; i++)
       st[i]=new SequentialSearchST<Key,Value>();
}

public SeparateChainingHashST() {
   this(997); // tamanho de tabela default
}
```

Colisões por listas encadeadas

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

public Value get(Key key) {
   int h = hash(key);
   return st[h].get(key);
}

public void put(Key key, Value val) {
   int h = hash(key);
   st[h].put(key, val);
}
```

Rehashing

A seguir, dobramos o tamanho da tabela se $\alpha \geq 10$.

```
public void put(Key key, Value val) {
   if (n >= 10*m) resize(2*m);
   int i = hash(key);
   if (!st[i].contains(key)) n++;
   st[i].put(key, val);
}
```

Rehashing

```
private void resize(int chains) {
   SeparateChainingHashST<Key, Value> t;
   t= new Sep...HashST<Key,Value>(chains);
   for (int i = 0; i < m; i++) {
       for (Key key:  st[i].keys()) {
           t.put(key, st[i].get(key));
       }
   }
   this.m = t.m;
   this.n = t.n;
   this.st = t.st;
}
```

Rehashing

Redimensiona a ST se $\alpha \leq 2$.

```
public void delete(Key key) {
   inti = hash(key);
   if (st[i].contains(key)) n--;
   st[i].delete(key);
   if (m > INIT_CAPACITY && n <= 2*m)
       resize(m/2);
}
```

Rehashing

Retorna as chaves da ST como iteráveis.

```
public Iterable<Key> keys() {
   Queue<Key> queue = new Queue<Key>();
   for (int i = 0; i < m; i++) {
       for (Key key:  st[i].keys())
           queue.enqueue(key);
   }
   return queue;
}
```