

Compacto dos melhores momentos

AULA 13

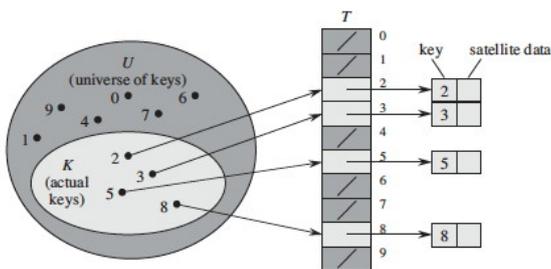
Endereçamento direto

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

Consumo de tempo

Em uma **ST** 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

Hash tables

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

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

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

Hash functions

A **função de dispersão** (= *hash function*) 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:

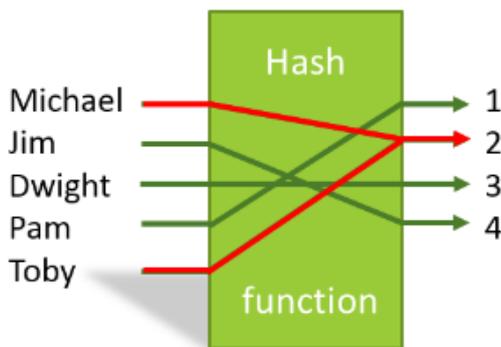
- ▶ possa ser **calculada** em $O(1)$ e
- ▶ **espalhe bem as chaves** pelo intervalo $0, \dots, m-1$.

Perfeição é difícil...

Perfect hashing: funções que associam **chaves diferentes** a inteiros diferentes são difíceis de se encontrar mesmo que **conhecendo as chaves de antemão!**

O **paradoxo do aniversário** nos diz se seleccionarmos 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**.

Conviver com colisões...



Fonte: <https://stackoverflow.com/>

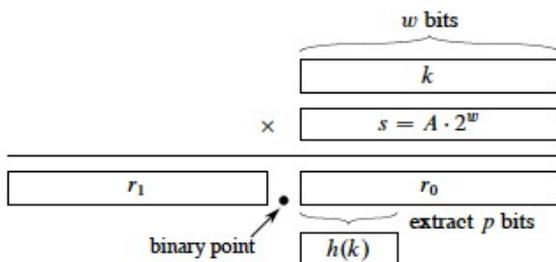
Hashing modular e multiplicativo

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):

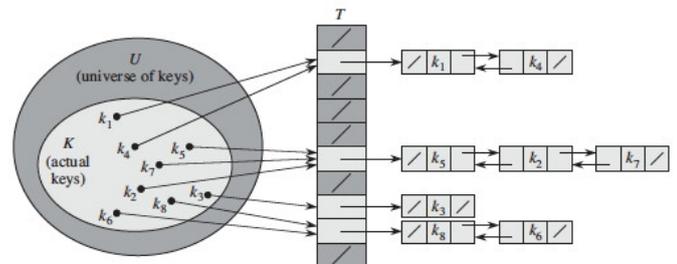
```
private int hash(int key) {
    return key % m;
}
```

Hashing modular e multiplicativo

$h(\text{key})$ contém os bits iniciais da metade menos significativa de $\text{key} \times A$ para uma constante $0 < A < 1$.

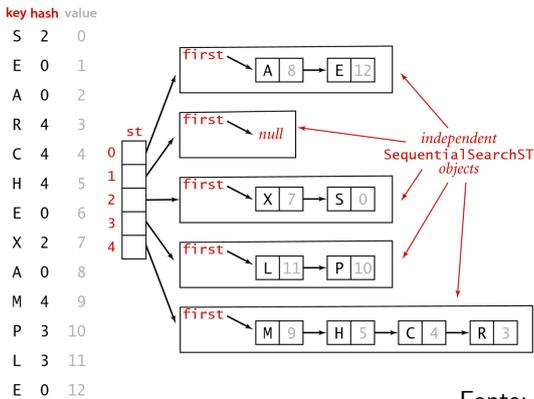


Separate chaining



Fonte: CLRS

Separate chaining



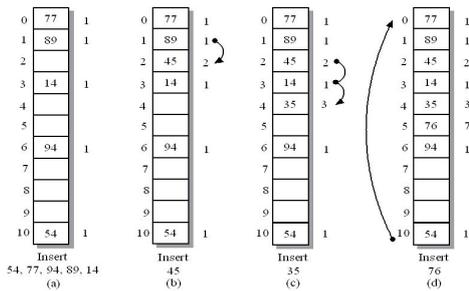
Fonte: [algs4](#)

Hashing with separate chaining for standard indexing client

AULA 14

Mais Hashing

Hash Table Using Linear Probing – Open Addressing



Fonte: CSE Lecture 16

HashMap<Key, Value>

```
public class HashMap<Key, Value>:
https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html
```

“... *Hash table* based implementation of the *Map* interface.

This implementation provides *constant-time performance* for the basic operations (*get* and *put*)

assuming the hash function disperses the elements properly among the buckets....”

HashMap<Key, Value>

```
public class HashMap<Key, Value>:
https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html
```

An instance of *HashMap* has two parameters that affect its performance: *initial capacity* and *load factor*.

... *initial capacity* is simply the capacity at the time the *hash table* is created.

The *load factor* is a measure of how full the *hash table* is allowed to get before its capacity is *automatically increased*.

HashMap<Key, Value>

```
public class HashMap<Key, Value>:
https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html
```

When the *number of entries* in the *hash table* exceeds the product of the *load factor* and the current capacity, the *hash table* is *reshaped* ...

so that the *hash table* has approximately *twice the number of buckets*.

Hipótese do Hashing Uniforme

O comprimento médio das listas é $\alpha = n/m$.

Poderíamos ter uma lista muito longa e todas as demais muito curtas...

Para eliminar essa possibilidade, precisamos saber ou supor algo sobre os dados.

Hipótese do Hashing Uniforme: Vamos supor que nossas funções de hashing distribuem as chaves pelo intervalo de inteiros $0..m$ de maneira uniforme (todos os valores hash igualmente prováveis) e independente.

Navigation icons

Hipótese do Hashing Uniforme

Isso significa que se as cada chave key é escolhida de um universo U de acordo com uma distribuição de probabilidade Pr ; ou seja, $Pr(key)$ é a probabilidade de key ser escolhida. Então a hipótese do hashing uniforme nos diz que

$$\sum_{key:h(key)=j} Pr(key) = \frac{1}{m}$$

para $j = 0, 1, 2, \dots, m - 1$.

Navigation icons

Análise separate chaining

Qual é o consumo de tempo de $get(key)$?

Análise é em termos do fator de carga $\alpha = n/m$ onde n é o número de itens na tabela e m é o número de listas.

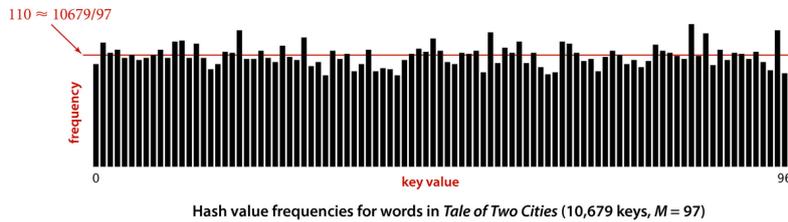
O fator de carga α é o número médio de itens por lista.

O pior caso ocorre quando todas as n chaves vão para mesma lista.

Consumo de tempo médio depende de quão bem a função de hash $h()$ distribui as chaves.

Navigation icons

Hipótese do Hashing Uniforme



Fonte:algs4

Navigation icons

Hipótese do Hashing Uniforme

Proposição: Em uma tabela de hash encadeada com m listas e n chaves, se vale a hipótese do hashing uniforme, a probabilidade de que o número de chaves em cada lista não passa de $\alpha = n/m$ multiplicado por uma pequena constante é muito próxima de 1.

Exemplo: Se $n/m = 10$, a probabilidade de que uma lista tenha comprimento maior que 20 é inferior a 0,8%.

Navigation icons

Consumo de tempo médio

A análise do consumo de tempo se apoia em uma suposição de uniform hashing.

Para $j = 0, \dots, m-1$ seja n_j o comprimento da lista $st[j]$.

Logo, $n = n_1 + n_2 + \dots + n_{m-1}$.

O valor esperado de n_j é $E[n_j] = \alpha$.

Supondo que $h(key)$ é computada em tempo $O(1)$, o tempo gasto por $get(key)$ depende do comprimento da lista $st[h(key)]$.

Navigation icons

Busca malsucedida

Considere dois casos: **malsucedida** (= **key** não está na **ST**) e busca **bem-sucedida** (= **key** está na **ST**).

Na **busca malsucedida** percorremos a lista $st[h[key]]$ até o final.

Hash uniforme nos diz que $\Pr(h(key) = j) = 1/m$.

O comprimento esperado da lista $st[h(key)]$ é α .

Logo, o **consumo de tempo médio** de uma busca de uma chave **key** que não está em $st[]$ é $O(1 + \alpha)$.

O termo "1" é devido ao consumo de tempo de $h(key)$.

◀ ▶ ⏪ ⏩ 🔍 ↺

Busca bem-sucedida

Para $j = 1, \dots, n$ seja **key_j** a **j**-ésima chave inserida na **ST**.

Para todo **i** e **j** defina a variável aleatória indicadora:

$$X_{ij} = I_{i,j} = \begin{cases} 1, & \text{se } h(\text{key}_i) = h(\text{key}_j) \\ 0, & \text{caso contrário} \end{cases}$$

Hash uniforme: $\Pr(h(\text{key}_i) = h(\text{key}_j)) = 1/m$.
Por quê?

Portanto, $E[X_{ij}] = 1/m$.

◀ ▶ ⏪ ⏩ 🔍 ↺

Busca bem-sucedida

Pela linearidade da esperança e ... a média é

$$\begin{aligned} &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{ij}] \right) \\ &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m} \right) \\ &= 1 + \frac{1}{nm} \sum_{i=1}^n (n-i) \\ &= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right) \end{aligned}$$

◀ ▶ ⏪ ⏩ 🔍 ↺

Busca bem-sucedida

Supremos que o elemento **key** procurado é igualmente provável de ser qualquer elemento na **ST**.

O **número de chaves examinadas** por $get(key)$ é 1 mais o número de elementos na lista $st[h(key)]$ **antes de key**.

Todos esses elementos foram inseridos na **ST** depois de **key**. Por quê?

Precisamos encontrar, para cada **key** na **ST**, o **número médio** de elementos inseridos em $st[h(key)]$ depois de **key**.

Esse é um trabalho para variáveis indicadoras!

◀ ▶ ⏪ ⏩ 🔍 ↺

Busca bem-sucedida

O número esperado de chaves examinadas em uma **busca com sucesso** é o número médio de chaves **key_j** inseridas depois de **key_i** e tal que $X_{ij} = 1$.

$$E \left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{ij} \right) \right]$$

- ▶ o **somatório interno** conta as chaves **key_j** inseridas depois de **key_i** e que tem o mesmo valor de hash de **key_i**
- ▶ o "1" é pelo custo de examinar **key_i**
- ▶ o **somatório mais externo** faz a soma sobre todas as chaves
- ▶ $1/n$ é para a **média**

◀ ▶ ⏪ ⏩ 🔍 ↺

Busca bem-sucedida

Continuando ...

$$\begin{aligned} &= 1 + \frac{n-1}{2m} \\ &= 1 + \frac{n}{2m} - \frac{n}{2mn} \end{aligned}$$

Substituindo n/m pelo fator de carga α obtemos

$$\begin{aligned} &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \\ &= O(1 + \alpha) \end{aligned}$$

◀ ▶ ⏪ ⏩ 🔍 ↺

Consumo de tempo

Seja n é o número de **chaves** e m é o tamanho da tabela.

Supondo que a função **hash** distribua as chaves uniformemente em $[0..m-1]$, em uma **tabela de distribuição** com **listas encadeadas** o consumo de tempo de **get()**, **put()** e **delete()** é $O(n/m)$.

Consumo de tempo

Supondo a função **hash** distribua as chaves uniformemente em $[0..m-1]$, em uma **tabela de distribuição** com **listas encadeadas** o consumo de tempo de **get()**, **put()** e **delete()** é $O(\alpha)$.

Se $n \leq c m$ para alguma contante c , ou seja, $n = O(m)$, então α é $O(1)$.

Open addressing

Open addressing procura **evitar** o espaço extra usado por listas ligadas colocando todas as chaves na tabela **st []**.

O **fator de carga** $\alpha = n/m$ da tabela é **menor do que 1**.

Examinar uma posição é chamado de **sondagem** (= *probe*).

Estendemos a **função de hash** para ter o **número da sondagem** como segundo parâmetro.

A sequência de sondagens

$$h(\text{key}, 0), h(\text{key}, 1), \dots, h(\text{key}, m-1)$$

deve ser uma **permutação** de $1, \dots, m-1$.

Colisões por sondagem linear

O métodos de resolução de **colisões** por *open addressing* mais simples é conhecido como **sondagem linear** (= *linear probing*).

Todos os itens são armazenados em um vetor **st[0..m-1]**.

Quando ocorre uma **colisão**, procuramos a **próxima posição vaga** do vetor.

Se **h()** é a função de hash então a sequência de sondagens é

$$h(\text{key}) \% m, h(\text{key}) + 1 \% m, \dots, h(\text{key}) + m - 1 \% m$$

Colisões por sondagem linear

Quanto maior o **fator de carga**, mais tempo as funções de busca e inserção vão consumir.

Durante a busca há três possibilidades:

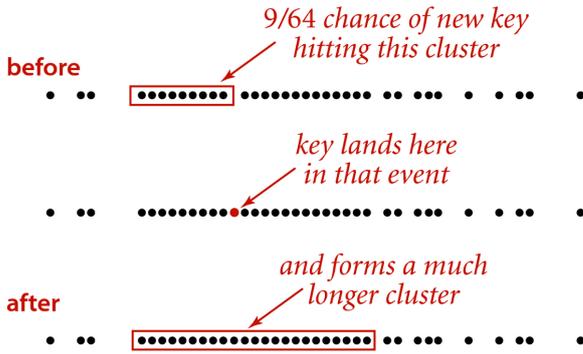
- ▶ **encontramos a chave**, paramos a busca;
- ▶ **posição não-ocupada**, paramos a busca;
- ▶ **posição está ocupada** e não é a chave, vamos para a próxima posição

LinearProbingHashST

```
private Key[] keys;  
private Value[] vals;
```

```
keys      cluster 0          cluster 1          cluster 2  
+-----+-----+-----+-----+-----+-----+-----+-----+  
+>| K | K | K | K | null| null| K | K | null| null| K | K | K | null| K |  
+-----+-----+-----+-----+-----+-----+-----+-----+  
      0  1  2  3  4  5  ...          ...          m-2  m-1  
  
vals  
+-----+-----+-----+-----+-----+-----+-----+-----+  
+>| V | V | V | V | null| null| V | V | null| null| V | V | V | null| V |  
+-----+-----+-----+-----+-----+-----+-----+-----+  
      0  1  2  3  4  5  ...          ...          m-2  m-1
```

Clusters



Clustering in linear probing ($M = 64$)

Fonte: [algs4](#)

Colisões por sondagem linear

```
public class Lin...HashST<Key, Value>{
    private int n;
    private int m = 16;
    private Key[] keys;
    private Value[] vals;
```

```
public Value get(Key key) {
    for (int i=hash(key); keys[i]!=null; i=(i+1)%m)
        if (keys[i].equals(key))
            return vals[i];
    return null;
}
```

Colisões por sondagem linear

Clusters

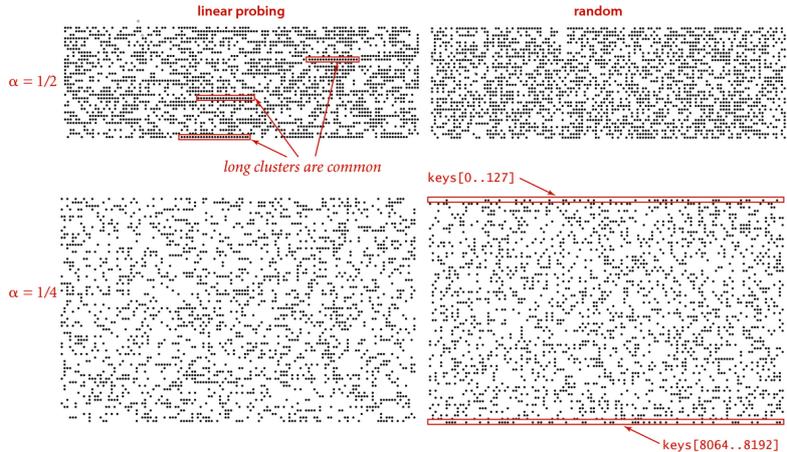


Table occupancy patterns (2,048 keys, tables laid out in 128-position rows)

Colisões por sondagem linear

```
public LinearProbingHashST() {
    keys = (Key[]) new Object[m];
    vals = (Value[]) new Object[m];
}
public LinearProbingHashST(int cap) {
    m = cap;
    keys = (Key[]) new Object[m];
    vals = (Value[]) new Object[m];
}
```

```
public void put(Key key, Value val) {
    int i;
    for (i=hash(key); keys[i]!=null; i=(i+1)%m)
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
    this.keys[i] = key;
    this.vals[i] = val;
    this.n++;
}
```

Colisões por sondagem linear

Análise

Proposição: Supondo que vale a hipótese do hashing uniforme, e que α está entre 0 e 1 mas não muito perto de 1, o número médio de sondagens em buscas bem-sucedidas é aproximadamente

$$\frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

e o número médio de sondagens em buscas malsucedidas (ou inserções) é aproximadamente

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

◀ ▶ ⏪ ⏩ 🔍 🔄

Memória

| método | espaço usado para n itens |
|---------------------|--------------------------------------|
| separating chaining | $\approx 48n + 64m$ |
| linear probing | entre $\approx 32n$ e $\approx 128n$ |
| BSTs | $\approx 56n$ |

◀ ▶ ⏪ ⏩ 🔍 🔄

Colisões por Double Hashing

Para uma dada chave key sondaremos

- ▶ primeiro a posição $h(key, 0) = h1(key) \% m$
- ▶ depois $h(key, 1) = (h1(key) + h2(key)) \% m$
- ▶ em seguida
 $h(key, 2) = (h1(key) + 2 \times h2(key)) \% m$
- ▶ em seguida
 $h(key, 3) = (h1(key) + 3 \times h2(key)) \% m$
- ▶ ...

◀ ▶ ⏪ ⏩ 🔍 🔄

Análise

Exemplo: quando $\alpha = 0.5$, temos aproximadamente 1,5 sondagens por busca bem-sucedida e aproximadamente 2,5 sondagens por busca malsucedida.

Exemplo: quando $\alpha = 0.25$, temos aproximadamente 1,16 sondagens por busca bem-sucedida e aproximadamente 1,39 por busca malsucedida.

◀ ▶ ⏪ ⏩ 🔍 🔄

Colisões por Double Hashing

Um outra estratégia é usarmos duas funções de hash $h1$ e $h2$, onde

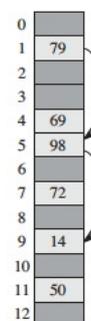
- ▶ $h1()$ fornece a posição inicial da sondagem e
- ▶ $h2()$ é responsável pelas demais sondagens.

As posições a serem sondadas serão dadas pela função

$$h(key, i) = (h1(key) + i \times h2(key)) \% m.$$

◀ ▶ ⏪ ⏩ 🔍 🔄

Colisões por Double Hashing



Fonte: CLRS

◀ ▶ ⏪ ⏩ 🔍 🔄

