



Fonte: ash.atozviews.com

Hash tables

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

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

A tabela terá a forma $tab[0..M-1]$, onde **M** é o tamanho da tabela.

Compacto dos melhores momentos

AULA 14

Hash functions

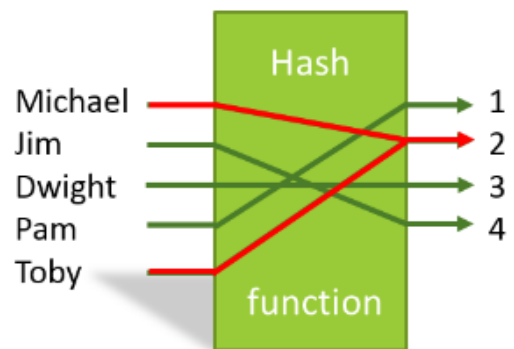
A função de dispersão (= *hash function*) recebe uma chave **key** e retorna um número inteiro $h(\text{key})$ no intervalo $0..M-1$.

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

Queremos uma função de hashing que:

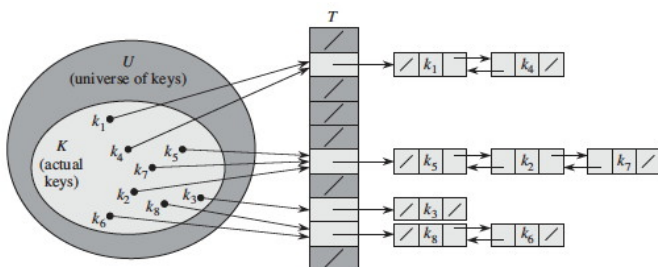
- ▶ possa ser calculada em $O(1)$ e
- ▶ espalhe bem as chaves pelo intervalo $0, \dots, m-1$.

Conviver com colisões...



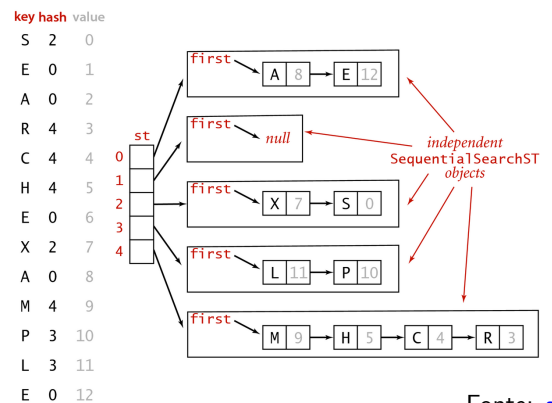
Fonte: <https://stackoverflow.com/>

Separate chaining



Fonte: CLRS

Separate chaining



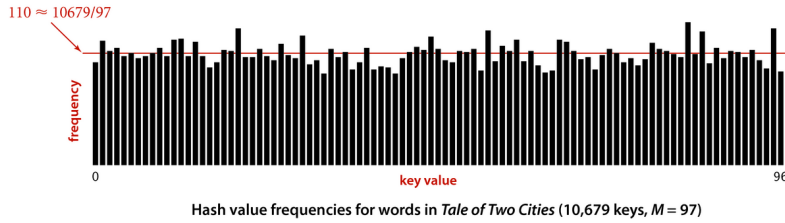
Fonte: algs4

Hashing with separate chaining for standard indexing client

Hipótese do Hashing Uniforme

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

Hipótese do Hashing Uniforme



Fonte:algs4

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, $\Pr(\text{key})$ é a probabilidade de **key** ser escolhida. Então a **hipótese do hashing uniforme** nos diz que

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

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

Consumo de tempo

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

Se $n \leq c m$ para alguma constante **c**, ou seja, $n = O(m)$, então α é $O(1)$ e portanto $O(1 + \alpha)$ é **constante**.

Mais 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.

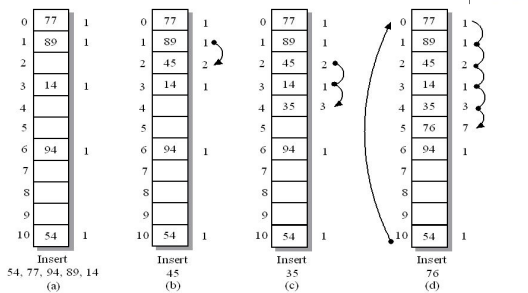
estrutura	ST	tempo
vetor	ordenada	1.5
skiplist	ordenada	1.1
árvore rubro-negra	ordenada	0.76
árvore binária de busca	ordenada	0.72
splay tree	ordenada	0.68
hash. encadeamento	não-ordenada	0.61
hash. encadeamento+MTF	não-ordenada	0.56

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

AULA 15

Colisões por open addressing

Hash Table Using Linear Probing – Open Addressing



Fonte: CSE Lecture 16

HashMap<Key, Value>

```
public class HashMap<Key, Value>:
https://docs.oracle
.../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
.../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
.../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 *rehashed* ...

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

Open addressing

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

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 $0, \dots, m-1$.

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 `tab[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

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

◀ ▶ ⏪ ⏩ 🔍

Colisões por sondagem linear

Retorna todas as chaves na ST como iterável.

```
public Iterable<Key> keys() {
    Queue<Key> queue = new Queue<Key>();
    for(int h = 0; h < m; h++)
        if (keys[h] != null)
            queue.enqueue(keys[h]);
    return queue;
}
```

◀ ▶ ⏪ ⏩ 🔍

Rehashing

```
private void resize(int cap) {
    LinearProbingHashST<Key, Value> t;
    t = new Line...HashST<Key, Value>(cap);
    for (int h = 0; h < m; h++) {
        if (keys[h] != null) {
            t.put(keys[h], vals[h]);
        }
    }
    keys = t.keys;
    vals = t.vals;
    m = t.m;
}
```

◀ ▶ ⏪ ⏩ 🔍

Colisões por sondagem linear

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

◀ ▶ ⏪ ⏩ 🔍

Rehashing

Na **sondagem linear**, é essencial que α fique bem abaixo de 1.

Convém manter $\alpha \leq 1/2$ (ou seja, $n \leq m/2$). Para manter α sob controle, a tabela de hash deve ser **redimensionada**, quando necessário, no início de `put()`.

```
public void put(Key key, Value val) {
    if (n >= m/2) resize(2*m);
    ...
}
```

◀ ▶ ⏪ ⏩ 🔍

Rehashing

```
public void delete(Key key) {
    if (!contains(key)) return;
    // encontre key
    int h = hash(key);
    while (!key.equals(keys[h]))
        h = (h + 1) % m;
    // remova key
    keys[h] = null;
    vals[h] = null;
    n--;
}
```

◀ ▶ ⏪ ⏩ 🔍

Rehashing

```
// rehash todas as chaves no cluster
h = (h + 1) % m;
while (keys[h] != null) {
    Key keyToRehash = keys[h];
    Value valToRehash = vals[h];
    keys[h] = null;
    vals[h] = null;
    n--;
    put(keyToRehash, valToRehash);
    h = (h + 1) % M;
}
// resize se estiver alfa <= 0.125
if (n > 0 && 8*n <= m) resize(m/2);
}
```

Consumo de tempo

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	$\approx 56n$

Consumo de tempo

O consumo de tempo de uma busca em tabelas de hash com sondagem linear depende, no pior caso, do tamanho do maior **cluster** (=fatia da tabela com chaves não nulas).

Consumo de tempo

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

$$h(\text{key}, h) = (h1(\text{key}) + h \times h2(\text{key})) \% m.$$

Colisões por Double Hashing

Para uma dada chave **key** sondaremos

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

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Colisões por Double Hashing

O valor de $h2(\text{key})$ e m devem ser relativamente primos. para garantir que a sequência de sondagens é uma permutação de $0, 1, \dots, m-1$.

Duas métodos utilizados:

- ▶ **tome** m como um potência de 2 e $h2()$ que sempre forneça inteiros ímpares;
- ▶ **selecione** um primo m e $h2()$ que retorne valores no intervalo $2 \dots m-1$.

O método nos fornece m^2 **sequências diferentes de sondagem**; sondagem linear que fornece apenas m .

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Universal hash functions

Um **adversário maldoso**, vendo a nossa função de hash, pode fornecer chaves que fazem com que as chaves sejam pessimamente distribuídas.

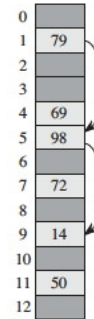
Hashing universal fornece um mecanismo aleatorizado que sorteia uma função de hash cada vez que construímos uma **ST**.

A função sorteada garante que o consumo de **tempo médio** das operações seja $O(1)$

Detalhes em MAC0338!

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Colisões por Double Hashing



Fonte: CLRS

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

Fator de carga

Note a diferença da interpretação do fator de carga. Em

- ▶ **separate chaining** o fator de carga α é o **número médio de itens** por lista: α pode ser maior que 1.
- ▶ **open addressing** o fator de carga α é a **fração da tabela** que está ocupada: α é menor que 1.

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ 🔍 ↺

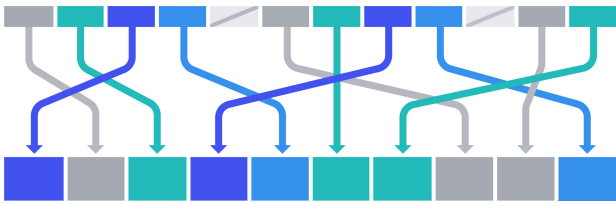
Mais mais mais experimentos ainda

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	ordenada	1.5
skiplist	ordenada	1.1
árvore rubro-negra	ordenada	0.76
árvore binária de busca	ordenada	0.72
splay tree	ordenada	0.68
hash. encadeamento	não-ordenada	0.61
hash. encadeamento+MTF	não-ordenada	0.56
hash. sondagem linear	não-ordenada	0.53

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

Comentários finais: Facebook F14



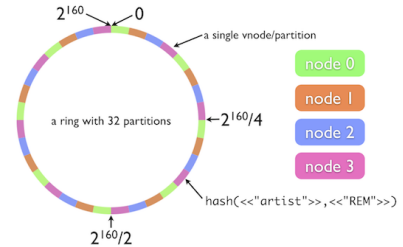
Fonte: [Open-sourcing F14 for faster, more memory-efficient hash tables](#)

Facebook F14 is a 14-way probing hash table that resolves collisions by double hashing.

Posted on apr 25, 2019 to [Developer Tools](#), [Open Source](#), no [github](#)

Navigation icons: back, forward, search, etc.

Mais comentários: Consistent hashing



Fonte: [The Simple Magic of Consistent Hashing](#)

Na média $\text{rehash } n/m$ chaves quando a tabela é redimensionada.

Navigation icons: back, forward, search, etc.

Mais comentários: banco de dados

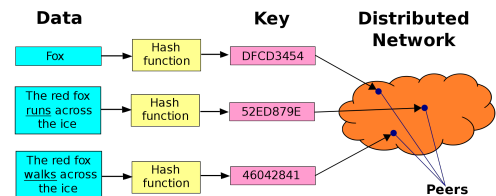


Fonte: [Cassandra](#)

Apache Cassandra é um banco de dados *open-source*, distribuído utiliza *consistent hash*.

Navigation icons: back, forward, search, etc.

Mais comentários: Distributed hash table



Fonte: [Distributed hash table \(wikipedia\)](#)

Classe sistemas distribuídos descentralizado que fornece um serviço de consulta semelhante a uma **tabela hash**: pares **key-val** são armazenados em um **DHT**, e qualquer nó participante pode recuperar com eficiência **val** associado a uma **dada key**.

Navigation icons: back, forward, search, etc.

Mais comentários: funções de hash

As aplicações de **funções de hash** vão muito além de **tabelas de hash**.

- ▶ consistência e arquivos: [Dropbox](#);
- ▶ criptografia: [Cryptographic hash function \(wikipedia\)](#)
- ▶ **hash'em all!**: sítio onde podemos obter o valor de hash de textos escolhendo dentre vários algoritmos;
- ▶ sistemas de senhas
- ▶ ...

Navigation icons: back, forward, search, etc.