MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 15

Hash tables

Inventadas para funcionar em $\mathrm{O}(1)$ . . . 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.

Hash functions

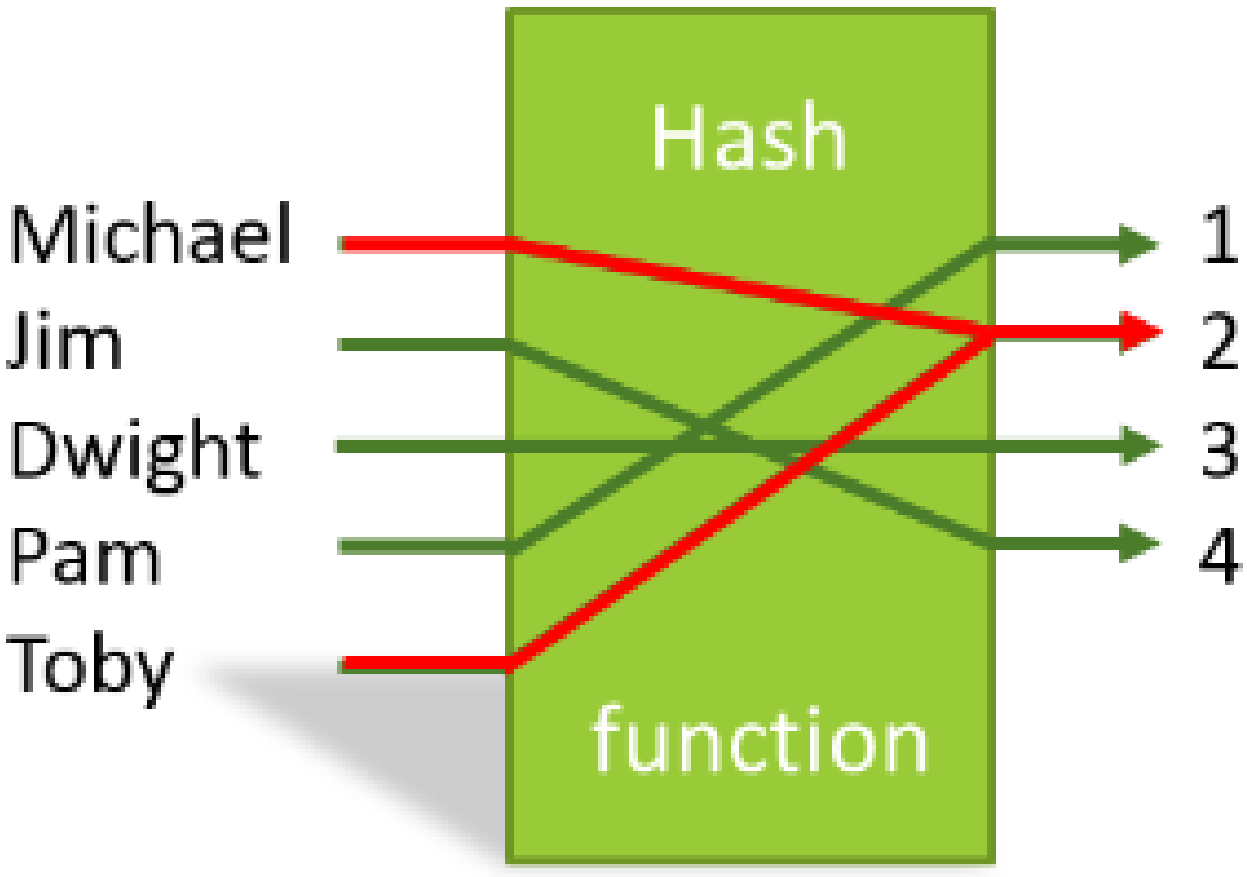
A função de dispersão (= *hash function*)
recebe uma chave `key` e retorna
um número inteiro `h(key)` no intervalo 0 . . `m`-1.

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

Queremos uma função de hashing que:

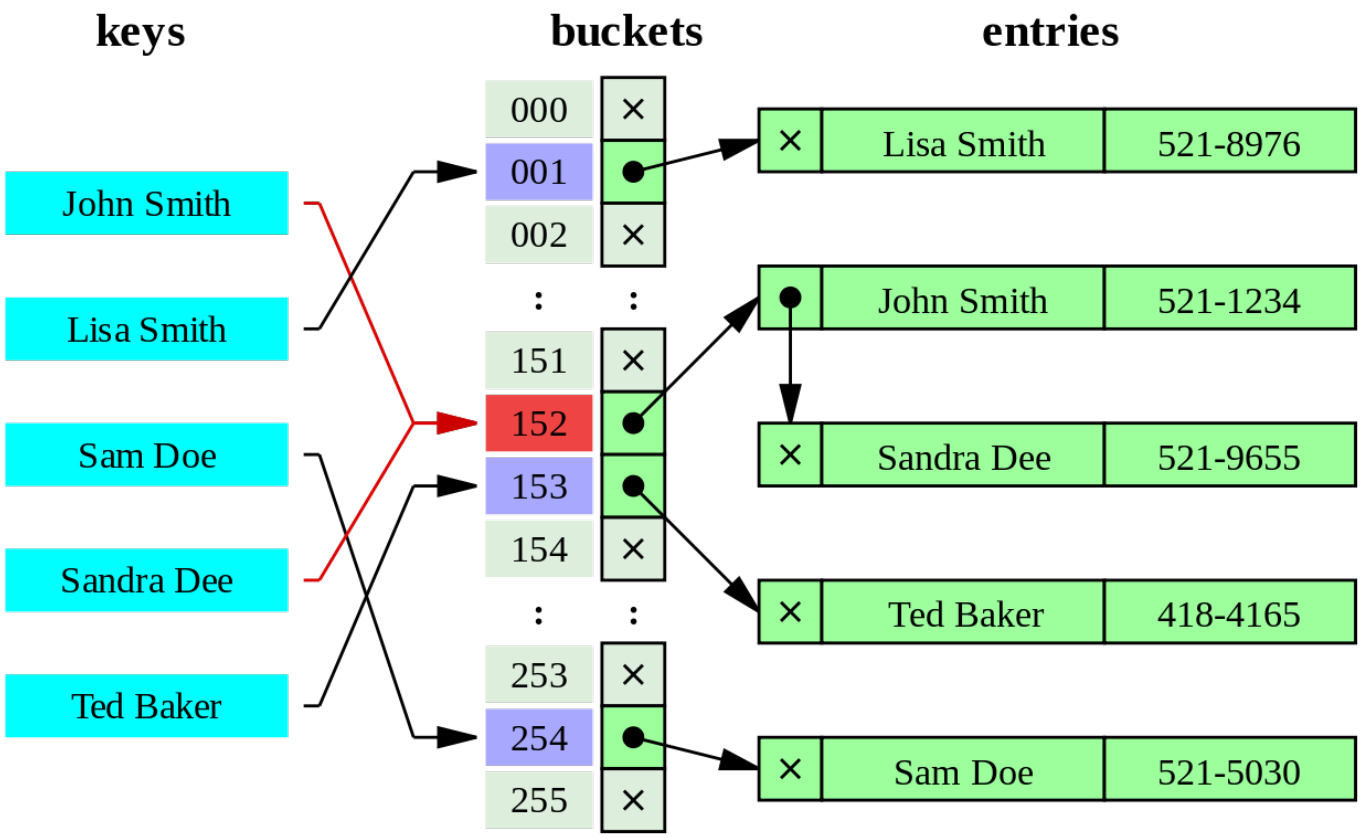
- possa ser calculada em $\mathrm{O}(1)$ e
- espalhe *bem* as chaves pelo intervalo $0 \,..\, \texttt{m}-1$.

Conviver com colisões...



Fonte: https://stackoverflow.com/

Separate chaining



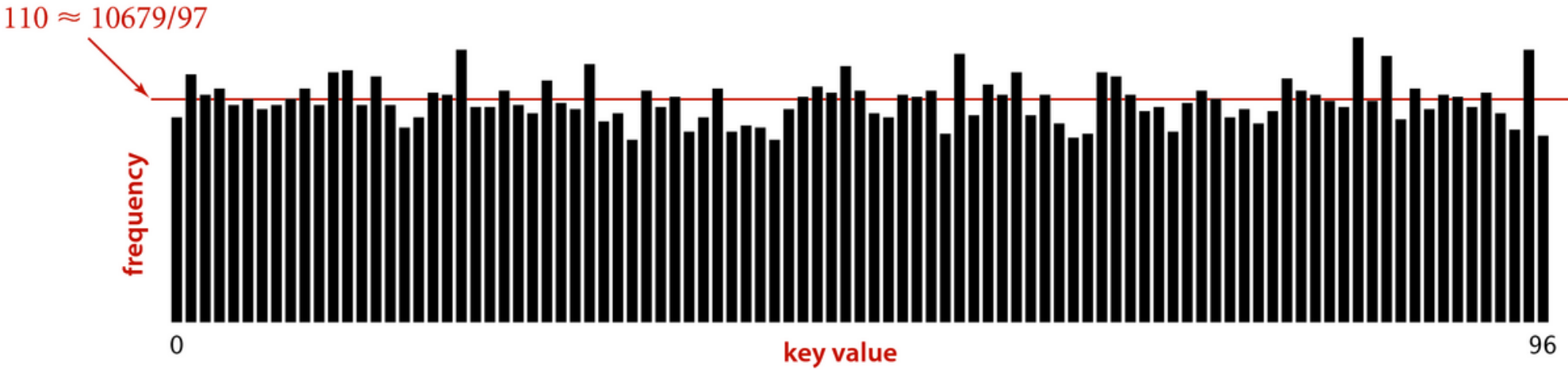
Fonte: `por Jorge Stolfi`

Hipótese do Hashing Uniforme

Hipótese do Hashing Uniforme:

Vamos supor que nossas funções de hashing distribuem as chaves pelo intervalo de inteiros 0 . . `m`-1 de maneira uniforme (todos os valores hash igualmente prováveis) e independente.

Hipótese do Hashing Uniforme



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)

Fonte:algs4

Hipótese do Hashing Uniforme

Isso significa que, se 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

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

para $\texttt{j} = 0, 1, 2, \ldots, \texttt{m} - 1$.

Consumo de tempo

Supondo que a função `hash` distribua as chaves uniformemente em [0 . . `m`-1], em uma tabela de dispersão 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 experimentos ainda

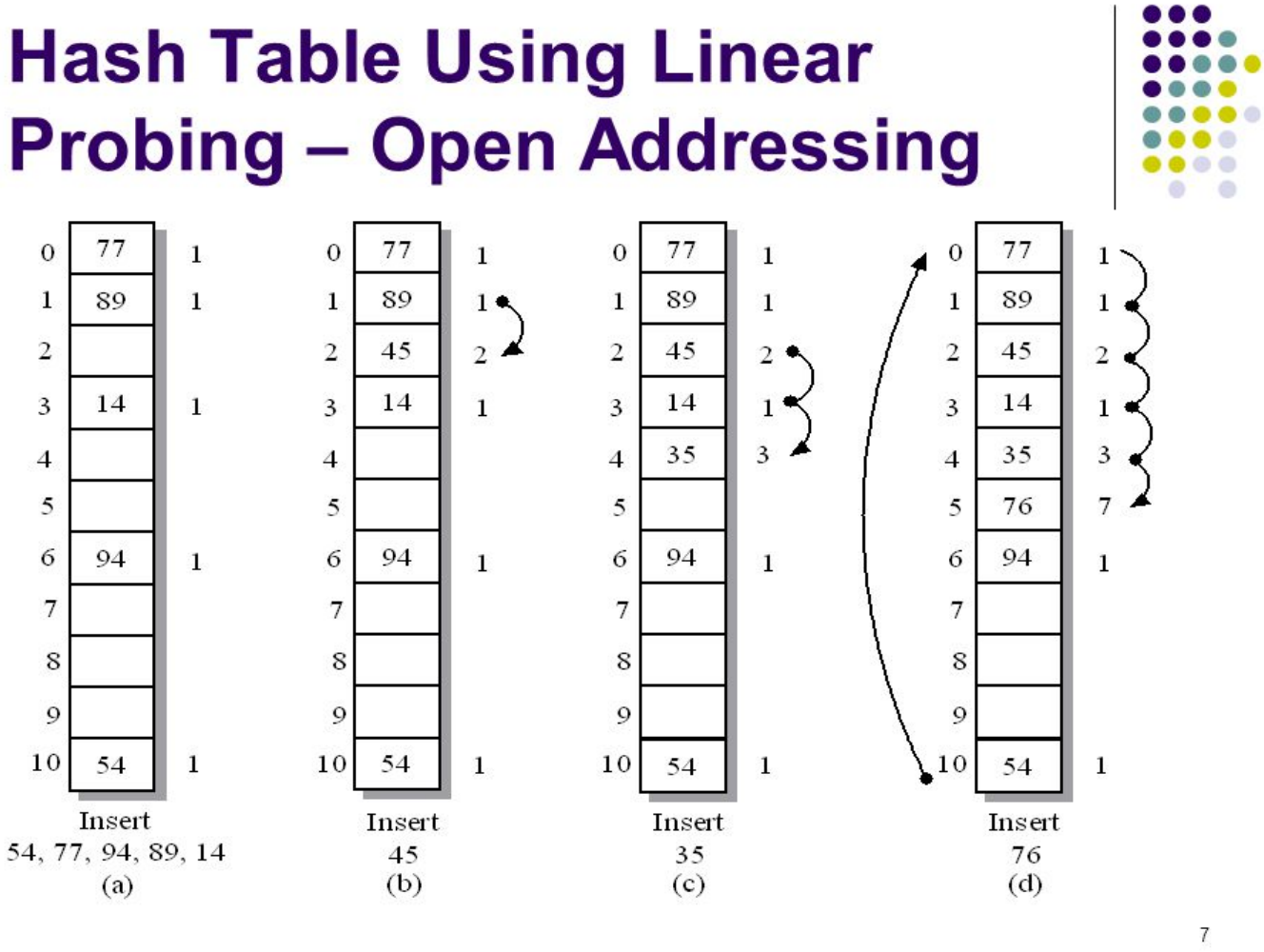
Consumo de tempo para se criar uma `ST` em que as chaves são as palavras em `les_miserables.txt` e os valores o número de ocorrências.

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 16

Colisões por *open addressing*



Fonte: CSE Lecture 16

Java: 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...***

Java: 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.*

Java: 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 $\alpha = \mathtt{n}/\mathtt{m}$ da tabela é menor do que 1.

Open addressing

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

O fator de carga $\alpha = \mathtt{n}/\mathtt{m}$ 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

$$\mathtt{h}(\mathtt{key}, 0), \mathtt{h}(\mathtt{key}, 1), \ldots, \mathtt{h}(\mathtt{key}, \mathtt{m}-1)$$

deve ser uma permutação de 0,. . . ,m-1.

Colisões por sondagem linear

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

Todos os itens são armazenados no vetor `tab`[0 . . `m`-1].

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

Colisões por sondagem linear

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

Todos os itens são armazenados no vetor `tab`[0 . . `m`-1].

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

Se `h()` é a função de dispersão,
então a sequência de sondagens é

$$\texttt{h(key)\%m},\ \texttt{(h(key)+1)\%m},\ \ldots,\ \texttt{(h(key)+m-1)\%m}$$

Colisões por sondagem linear

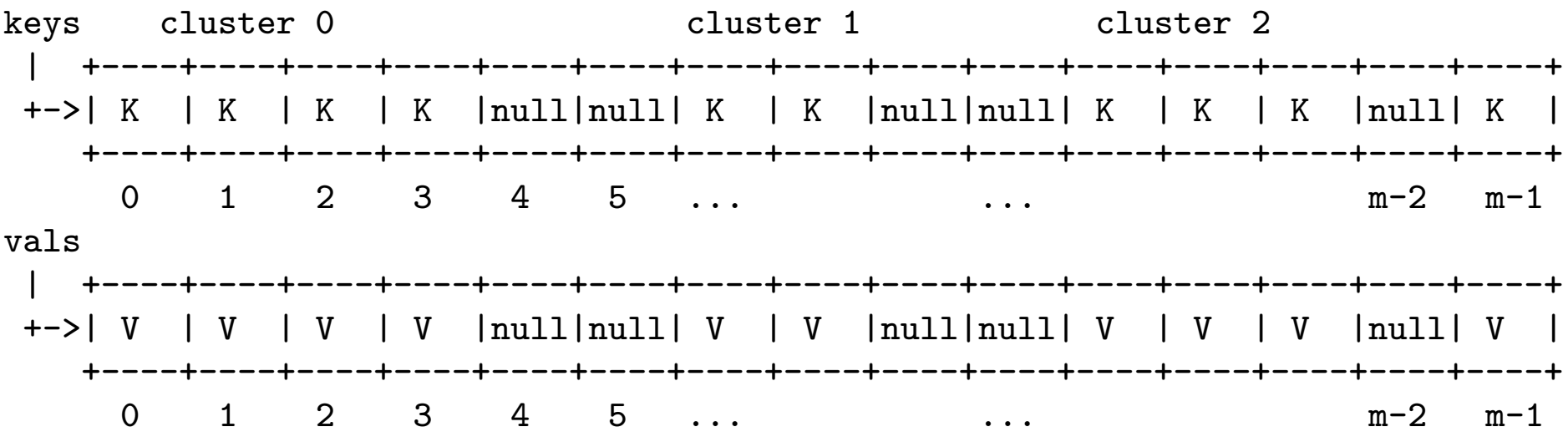
Quanto maior o fator de carga, mais tempo as funções de busca e inserção vão consumir.

Durante a busca, há três possibildades:

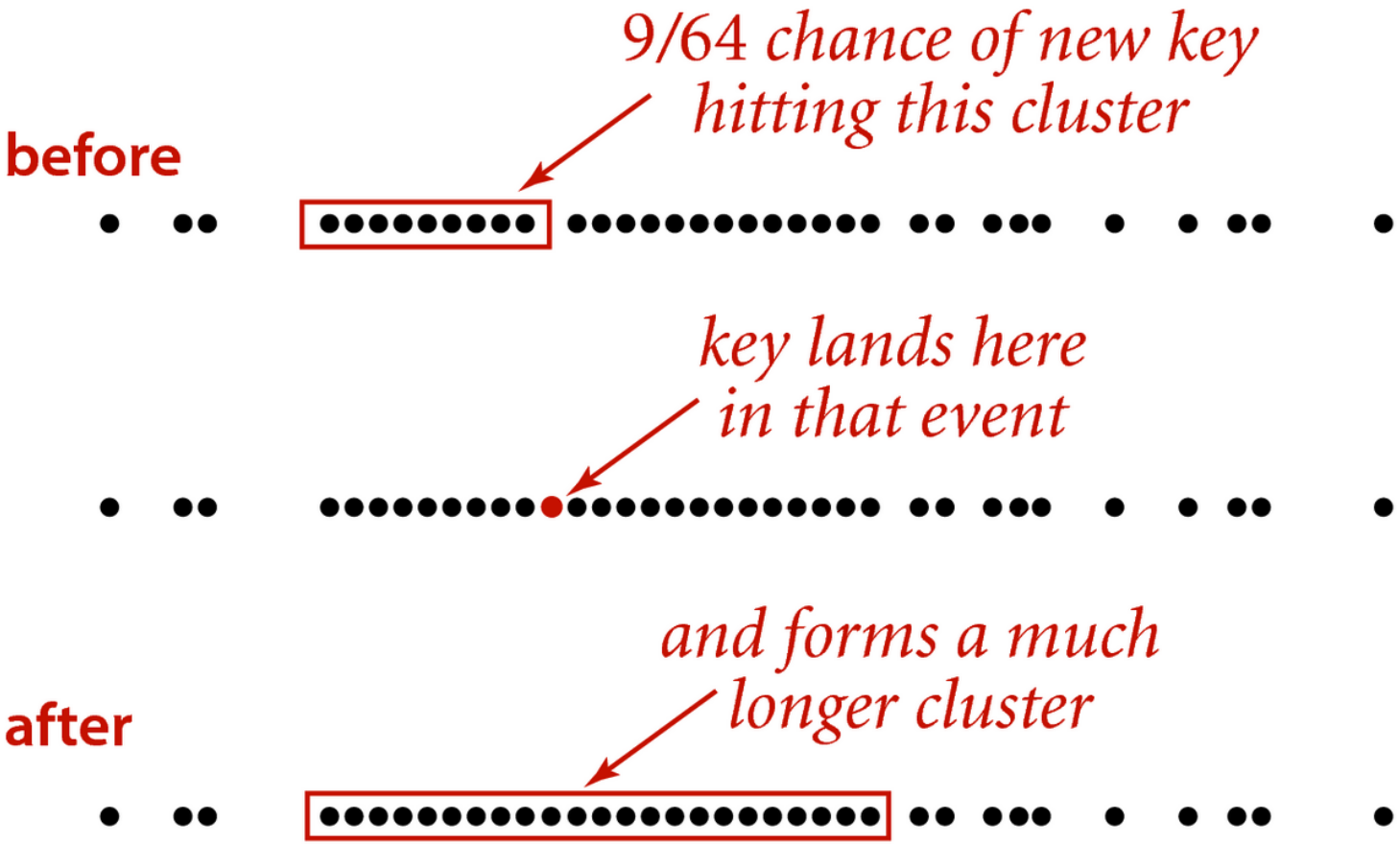
- encontramos a chave, paramos a busca;
- posição não-ocupada, paramos a busca;
- posição está ocupada e não é a chave, vamos para a próxima posição.

LinearProbingHashST

```
static Key   *keys;
static Value *vals;
```



Clusters



Clustering in linear probing ($M = 64$)

Fonte: `algs4`

Clusters

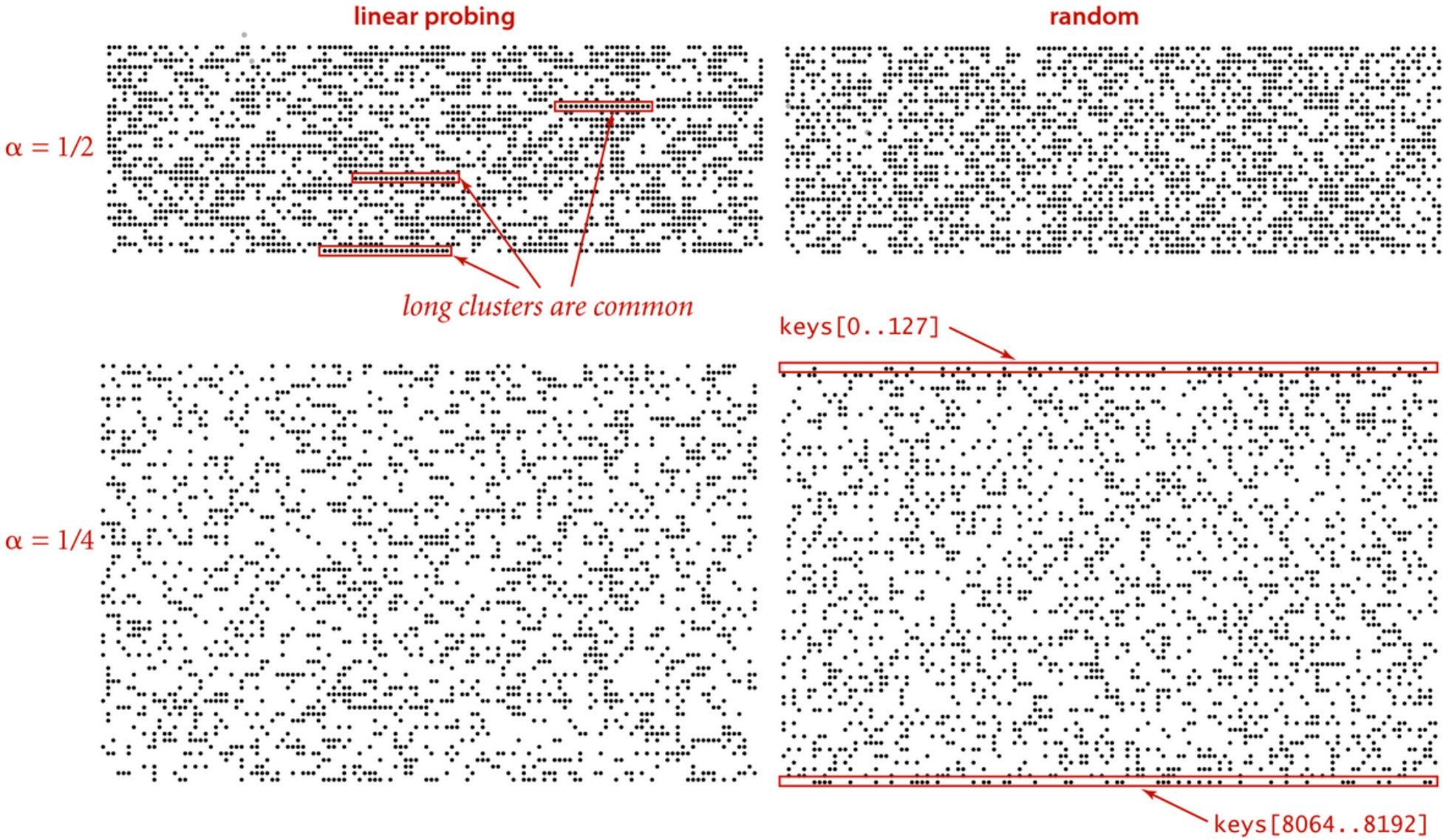


Table occupancy patterns (2,048 keys, tables laid out in 128-position rows)

Colisões por sondagem linear

```
static int n;
static int m;
static Key   *keys;
static Value *vals;
```

Colisões por sondagem linear

```
static int n;
static int m;
static Key   *keys;
static Value *vals;

void LinearProbingHashSTInit(int cap) {
   m = cap;
   keys = mallocSafe(m * sizeof(Key));
   vals = mallocSafe(m * sizeof(Value));
   for (int h = 0; h < m; h++)
      keys[h] = vals[h] = NULL;
   n = 0;
}
```

Colisões por sondagem linear

```
Value getST(Key key) {
   int h;
   for (h=hash(key); keys[h]!=NULL; h=(h+1)%m)
      if (compare(keys[h], key) == 0)
         return vals[h];
   return NULL;
}
```

Colisões por sondagem linear

```
void putST(Key key, Value val) {
   int h;
   for (h=hash(key); keys[h]!=NULL; h=(h+1)%m)
      if (compare(keys[h], key) == 0) {
         vals[h] = val;
         return;
      }
   keys[h] = key;
   vals[h] = val;
   n++;
}
```

Colisões por sondagem linear

Retorna uma fila com todas as chaves na ST.

```
Queue keys() {
   Queue queue = queueInit(n);
   for (int h = 0; h < m; h++)
      if (keys[h] != NULL)
         enqueue(queue, keys[h]);
   return queue;
}
```

Rehashing

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

Convém manter alpha $\leq 1/2$ (ou seja, $\texttt{n} \leq \texttt{m}/2$).

Para manter α sob controle,
a tabela de dispersão deve ser redimensionada,
quando necessário, no início de `putST()`.

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

Rehashing

```
static void resize(int cap) {
   Key *k = keys; Value *v = vals;
   int h, aux = m;
   m = cap;
   keys = mallocSafe(m * sizeof(Key));
   vals = mallocSafe(m * sizeof(Value));
   for (int h = 0; h < m; h++)
      keys[h] = vals[h] = NULL;
   for (h = 0; h < aux; h++)
      if (k[h] != NULL)
         insertST(k[h], v[h]);
   free(k);   free(v);
}
```

Rehashing

```
void delete(Key key) {
   if (!contains(key)) return;

   /* procure key */
   int h = hash(key);
   while (compare(key, keys[h]) != 0)
      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--;
       putST(keyToRehash, valToRehash);
       h = (h + 1) % M;
    }
    /* resize se alfa <= 0.125 */
    if (n > 0 && 8*n <= m) resize(m/2);
}
```

Consumo de tempo

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

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 mal sucedidas (ou inserções) é aproximadamente

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

Consumo de tempo

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

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

Memória

Estimativa do número de bytes usados considerando que fossem armazenados apontadores para `key` e `val`.

Um apontador: 8 bytes (em máquinas de 64 bits)

método	espaço usado para `n` itens
separating chaining	$\approx 8\mathtt{m} + 24\mathtt{n}$
linear probing	entre $\approx 32\mathtt{n}$ e $\approx 128\mathtt{n}$
BSTs	$\approx 32\mathtt{n}$

Colisões por *Double Hashing*

Uma 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

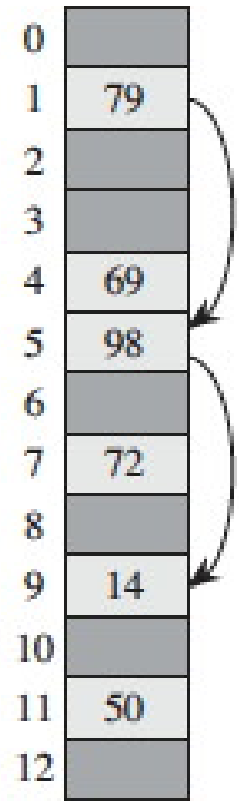
$$\texttt{h}(\texttt{key}, \texttt{i}) = (\texttt{h1}(\texttt{key}) + \texttt{i} \times \texttt{h2}(\texttt{key}))\%\texttt{m}.$$

Colisões por *Double Hashing*

Para uma dada chave $\texttt{key}$, sondaremos

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

Colisões por *Double Hashing*



Fonte: CLRS

Colisões por *Double Hashing*

O valor de `h2(key)` e `m` devem ser relativamente primos para garantir que a sequência de sondagens é uma permutação de 0,1,...,`m`-1.

Dois métodos utilizados:

- tome `m` como uma potência de 2 e `h2()` que sempre forneça inteiros ímpares;
- selecione um primo `m` e `h2()` que retorne valores no intervalo 2..`m`-1.

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

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.

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!

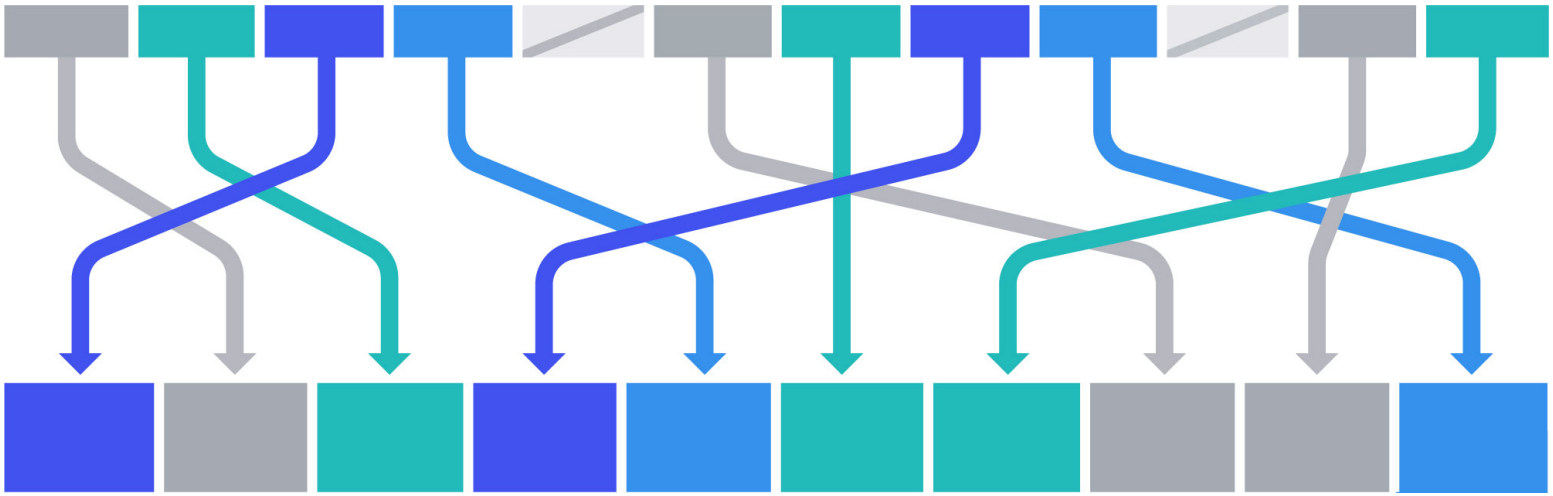
Mais experimentos ainda

Consumo de tempo para se criar uma ST em que as 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

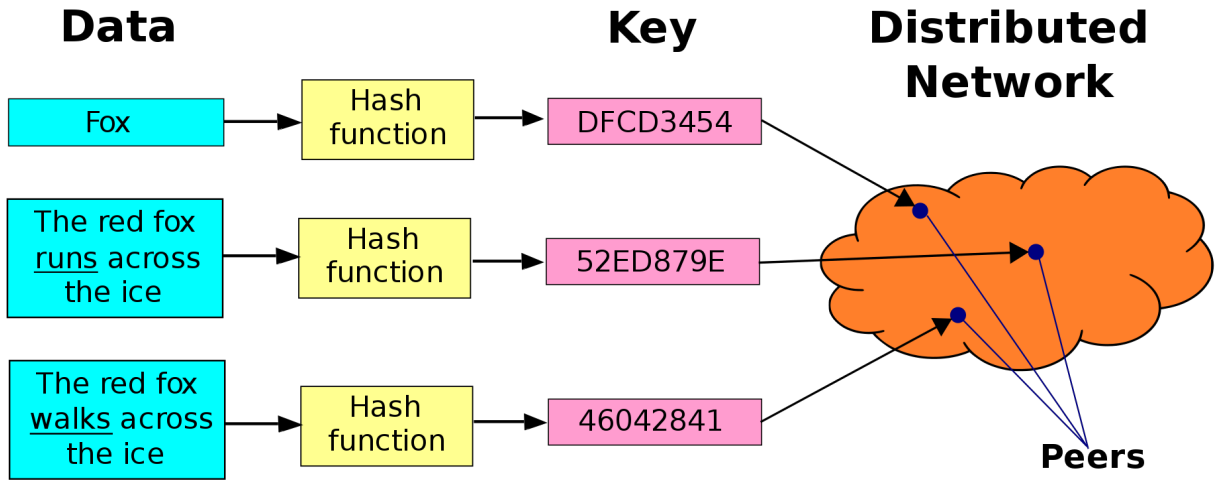


Fonte: `Open-sourcing F14 for faster, more memory-efficient hash tables`

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.

Mais comentários: *Distributed hash table*



Fonte: `Distributed hash table (wikipedia)`

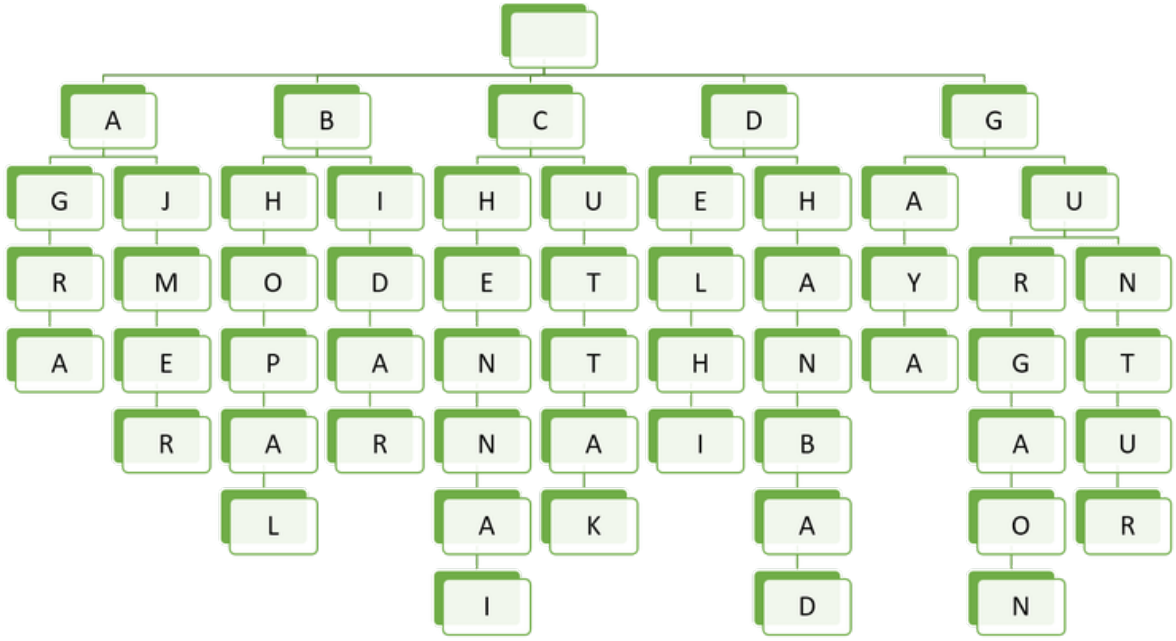
Classe de sistemas distribuídos descentralizados que fornece um serviço de consulta semelhante a uma tabela hash: pares `key`-`val` são armazenados em um DHT, e qualquer nó participante pode recuperar com eficiência `val` associado a uma dada `key`.

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
- ...

Próxima aula: Tries (árvores digitais)



Fonte: `Building an autocomplete system using Trie`

Referências: Tries (árvores digitais) (PF); Tries (S&W); slides (S&W); Vídeo (S&W); TAOCP, vol 3, cap. 6.3

R-way tries

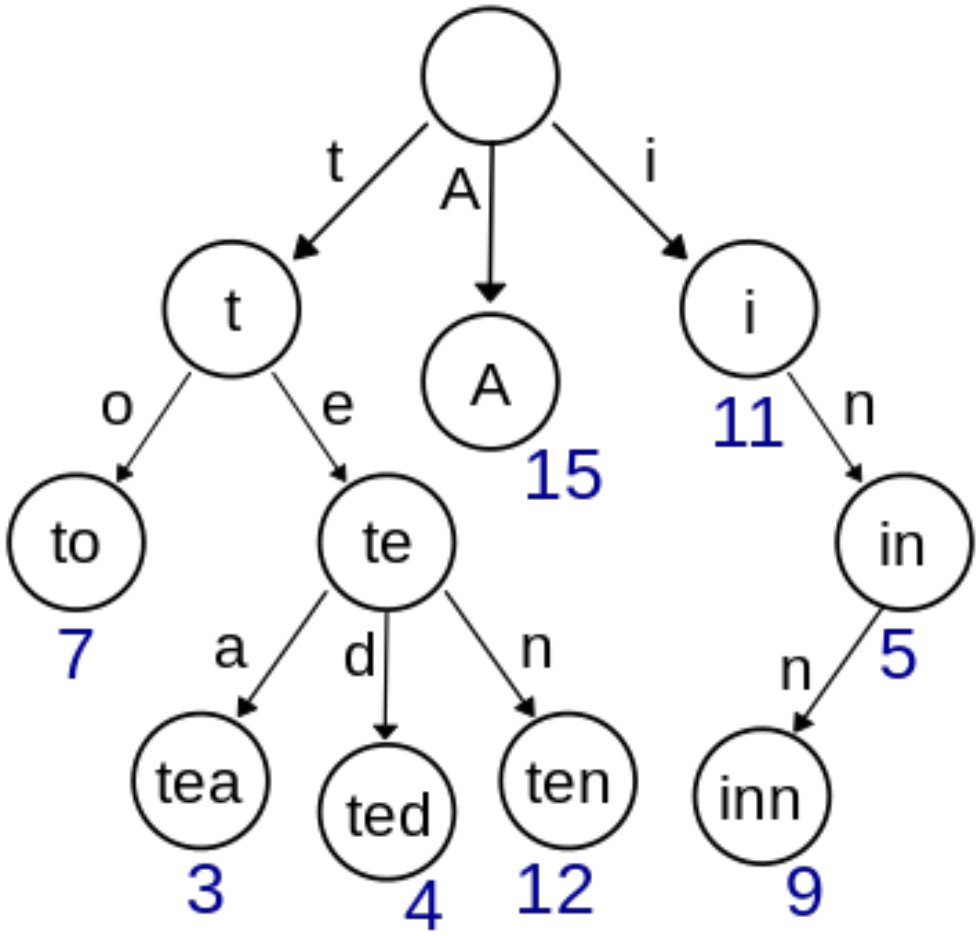
Uma **trie** (=*R-way trie*) é um tipo de árvore usado para implementar STs de strings sobre um alfabeto com R símbolos.

Tries também são conhecidas como árvores digitais e como árvores de prefixos.

Com Tries, em vez do método de busca ser baseado em comparações entre chaves, é utilizada a representação das chaves como caracteres de um alfabeto.

Considere, por exemplo,
a busca de uma palavra no dicionário:
a primeira letra indica as páginas que devemos olhar;
a segunda letra restringe o espaço de busca; . . .

Ilustração



Fonte: Wikipedia