Compacto dos melhores momentos

AULA 13

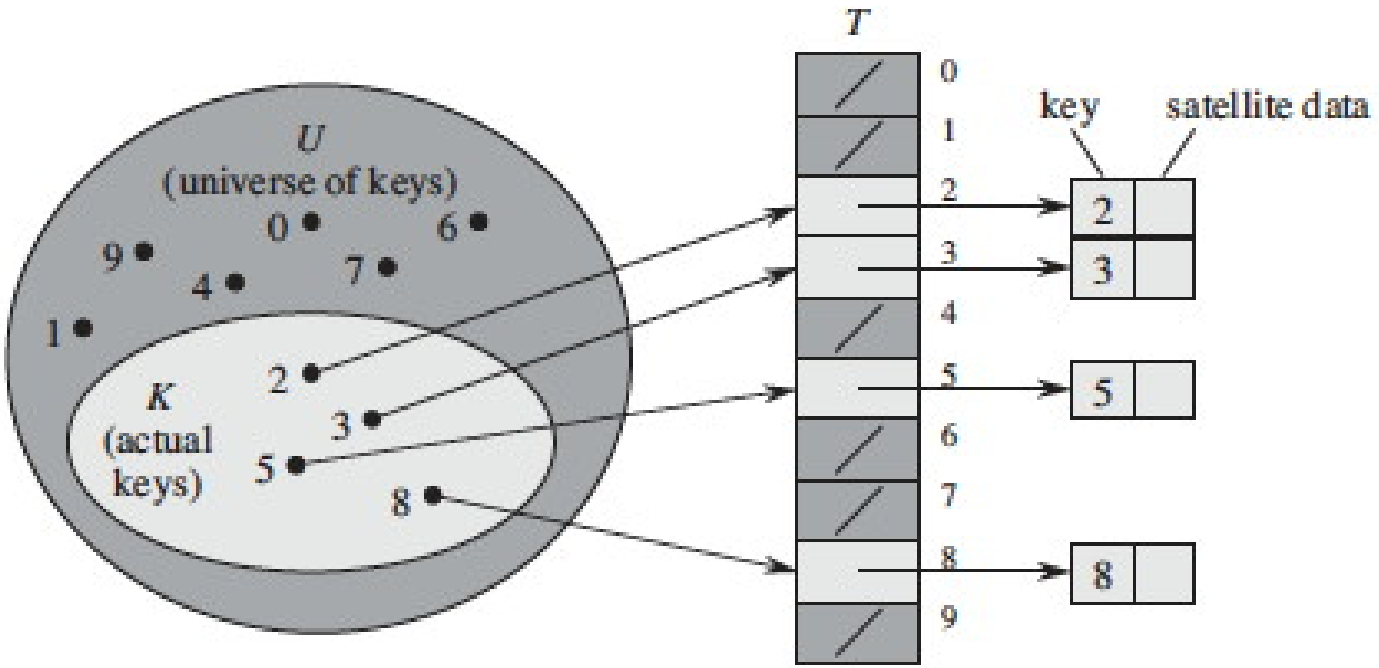
Endereçamento direto

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 arnazena 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 $\mathrm{O}(1)$ . . . em média.

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

A tabela terá a forma `st`[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:

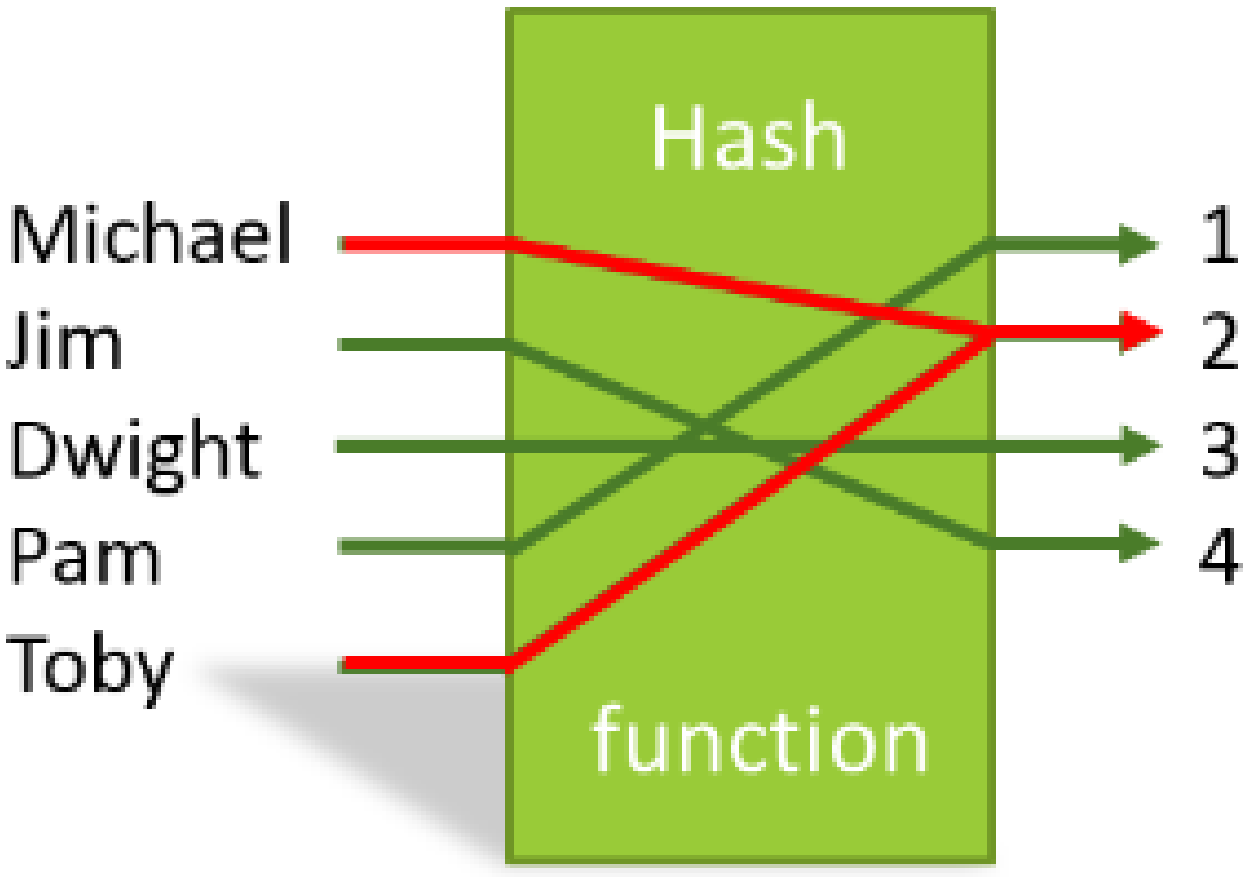
- possa ser calculada em $O(1)$ e
- espalhe *bem* as chaves pelo intervalo $0, \ldots, \texttt{m}-1$.

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 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.

Conviver com colisões...



Fonte: https://stackoverflow.com/

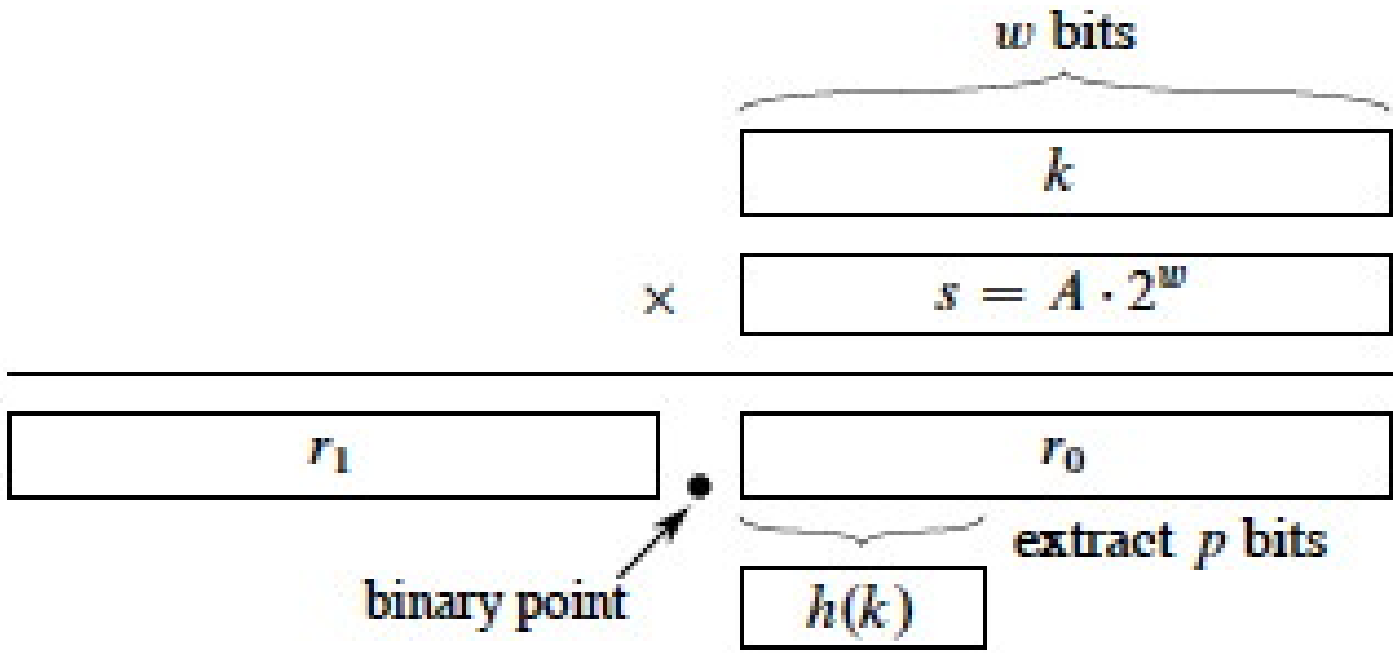
Hashing modular e multiplicativo

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`):

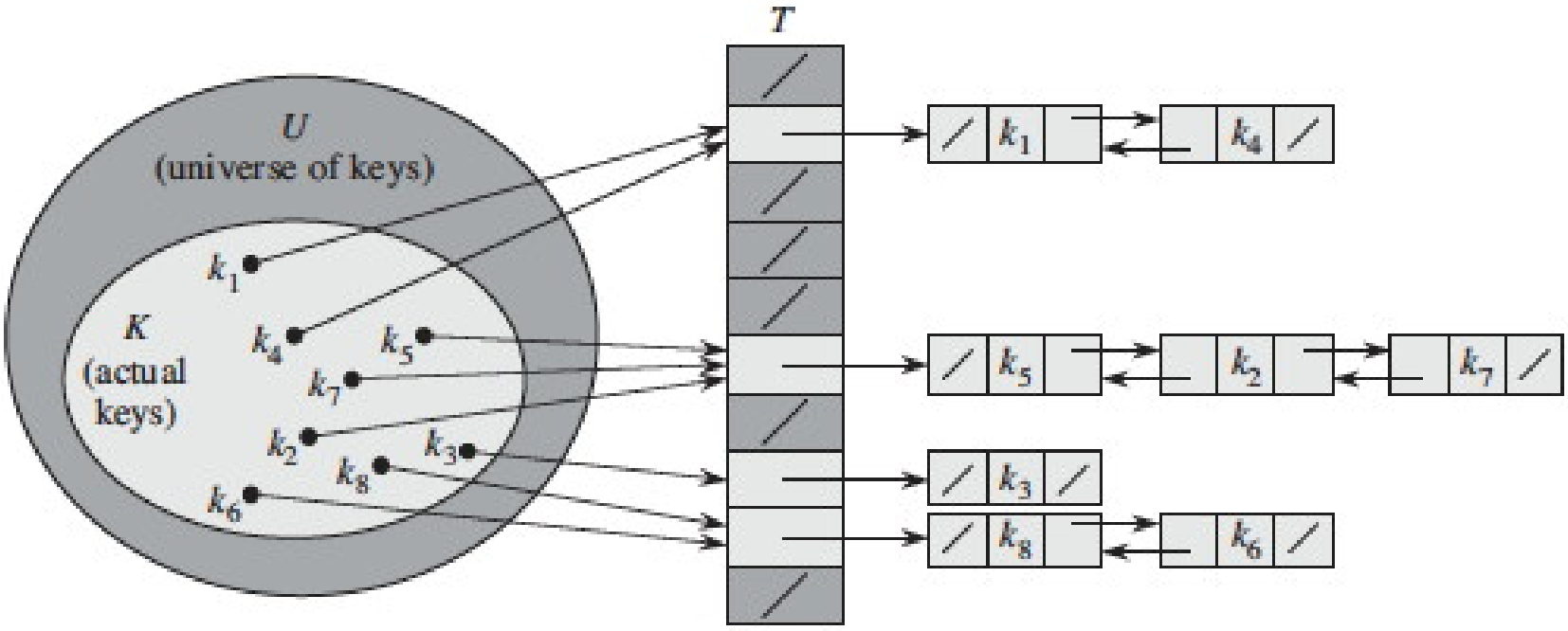
```
private int hash(int key) {
   return key % m;
}
```

Hashing modular e multiplicativo

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

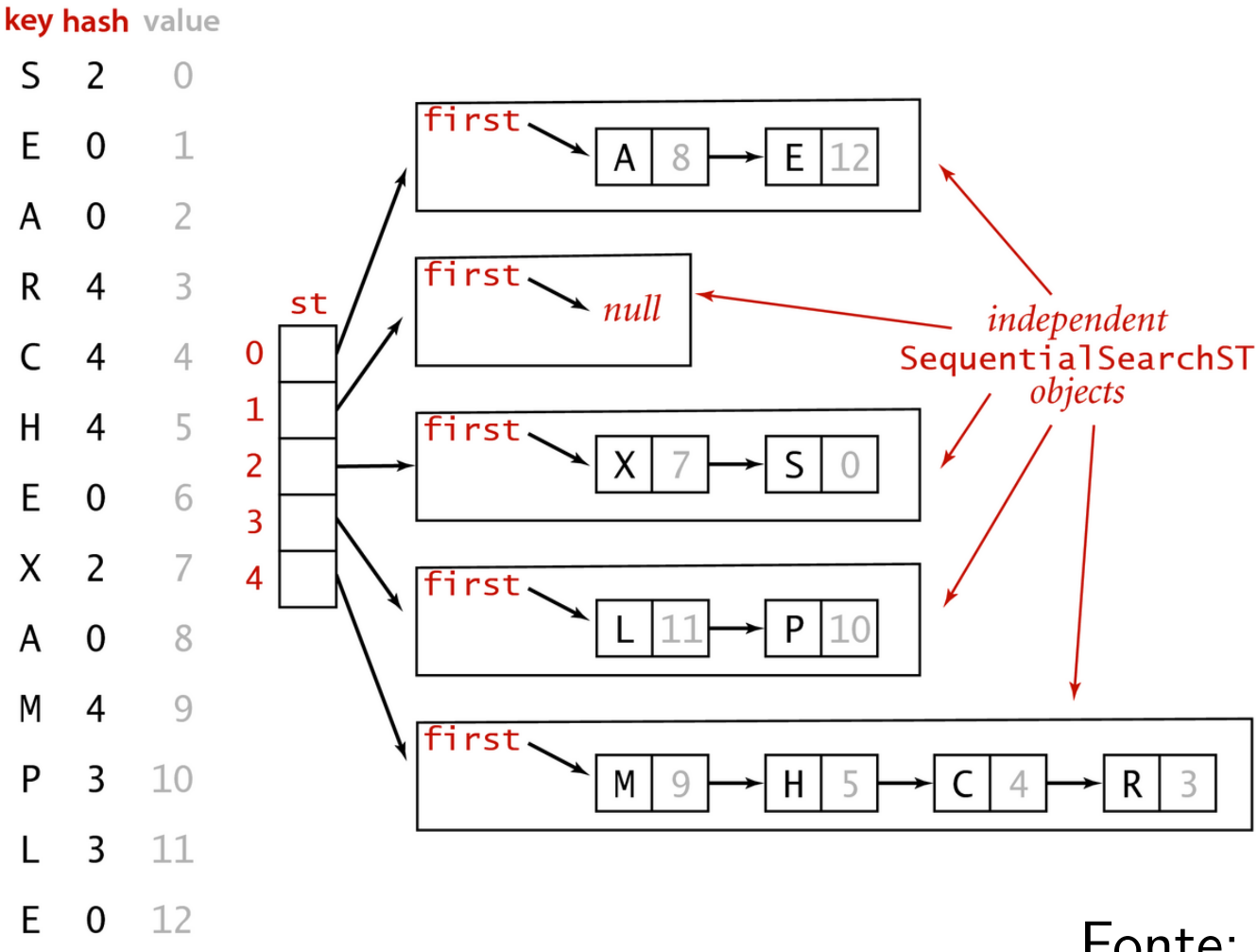


Separate chaining



Fonte: CLRS

Separate chaining



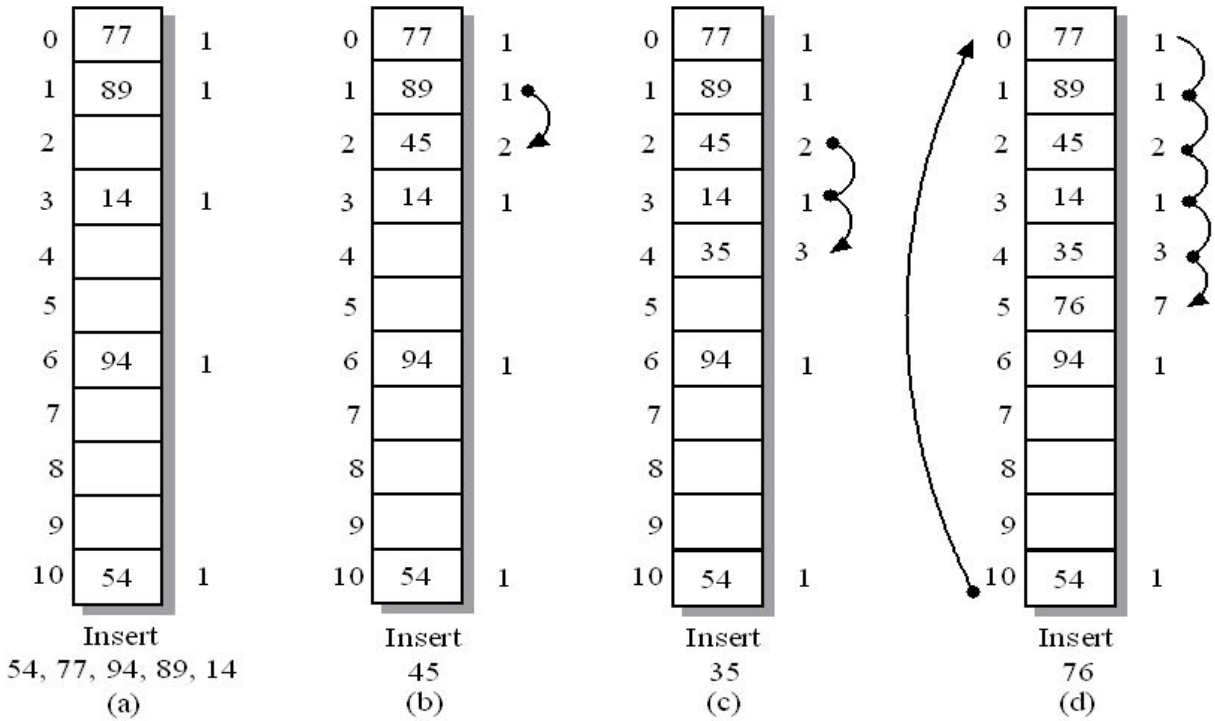
Fonte: algs4

Hashing with separate chaining for standard indexing client

AULA 14

Mais Hashing

Hash Table Using Linear Probing – Open Addressing



7

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.*

Hipótese do Hashing Uniforme

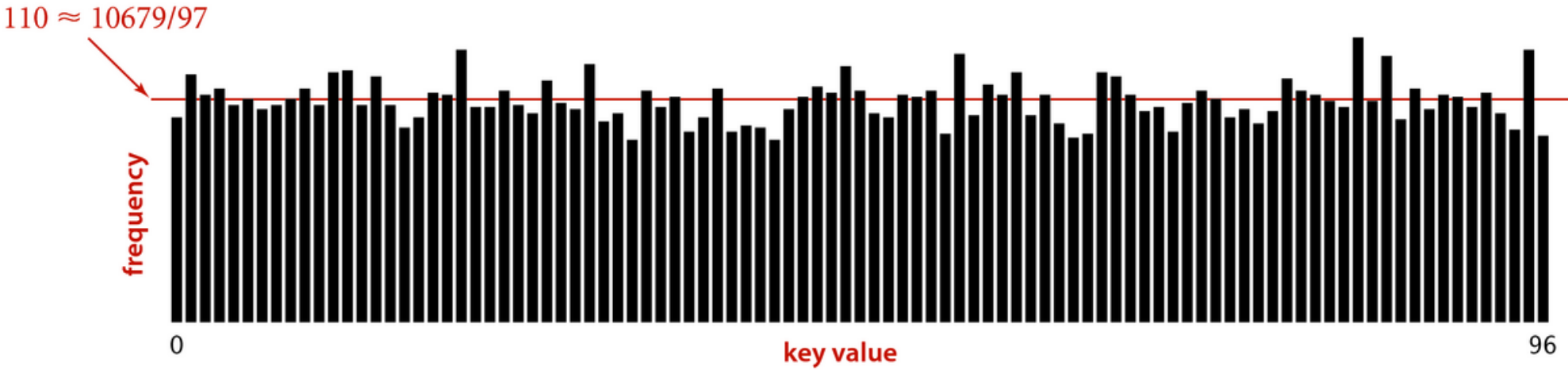
O comprimento médio das listas é $\alpha = \mathtt{n}/\mathtt{m}$.

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 $0\,..\,\mathtt{m}$ 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 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

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

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

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%.

Análise separate chaining

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

Análise é em termos do fator de carga $\alpha = \mathtt{n}/\mathtt{m}$ 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 o corre 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.

Consumo de tempo médio

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

Para $\texttt{j} = 0, \ldots, \texttt{m}-1$ seja $\texttt{n}_\texttt{j}$ o comprimento da lista `st[j]`.

Logo, $\texttt{n} = \texttt{n}_1 + \texttt{n}_2 + \cdots + \texttt{n}_{\texttt{m}-1}$.

O valor esperado de $\texttt{n}_\texttt{j}$ é $\texttt{E}[\texttt{n}_\texttt{j}] = \alpha$.

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

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 $\Pr(\texttt{h}(\texttt{key}) = \texttt{j}) = 1/\texttt{m}$.

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

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

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

Busca bem-sucedida

Suporemos 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

Para $\mathtt{j} = 1, \ldots, \mathtt{n}$ seja $\mathtt{key}_\mathtt{j}$ a $\mathtt{j}$-ésima chave inserida na $\mathtt{ST}$.

Para todo $\mathtt{i}$ e $\mathtt{j}$ defina a variável aleatória indicadora:

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

Hash uniforme: $\Pr(\mathtt{h}(\mathtt{key}_\mathtt{i}) = \mathtt{h}(\mathtt{key}_\mathtt{j})) = 1/\mathtt{m}$.
Por quê?

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

Busca bem-sucedida

O número esperado de chaves examinadas em uma busca com sucesso é o número médio de chaves $\texttt{key}_\texttt{j}$ inseridas depois de $\texttt{key}_\texttt{i}$ **e** tal que $X_{\texttt{ij}} = 1$.

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

- o somatório interno conta as chaves $\texttt{key}_\texttt{j}$ inseridas depois de $\texttt{key}_\texttt{i}$ e que tem o mesmo valor de hash de $\texttt{key}_\texttt{i}$
- o "1" é pelo custo de examinar $\texttt{key}_\texttt{i}$
- o somatório mais externo faz a soma sobre todas a chaves
- $1/\texttt{n}$ é para a média

Busca bem-sucedida

Pela linearidade da esperança e . . . a média é

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

Busca bem-sucedida

Continuando . . .

$$= \quad 1 + \frac{\mathtt{n} - 1}{2\,\mathtt{m}}$$

$$= \quad 1 + \frac{\mathtt{n}}{2\,\mathtt{m}} - \frac{\mathtt{n}}{2\,\mathtt{m}\,\mathtt{n}}$$

Substituindo $\mathtt{n}/\mathtt{m}$ pelo fator de carga α obtemos

$$= \quad 1 + \frac{\alpha}{2} - \frac{\alpha}{2\mathtt{n}}$$

$$= \quad \mathrm{O}(1 + \alpha)$$

Consumo de tempo

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

Supondo que a função `hash` distribuia 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` distribuia 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 $\alpha = \mathtt{n}/\mathtt{m}$ 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 1,. . . ,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

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

Colisões por sondagem linear

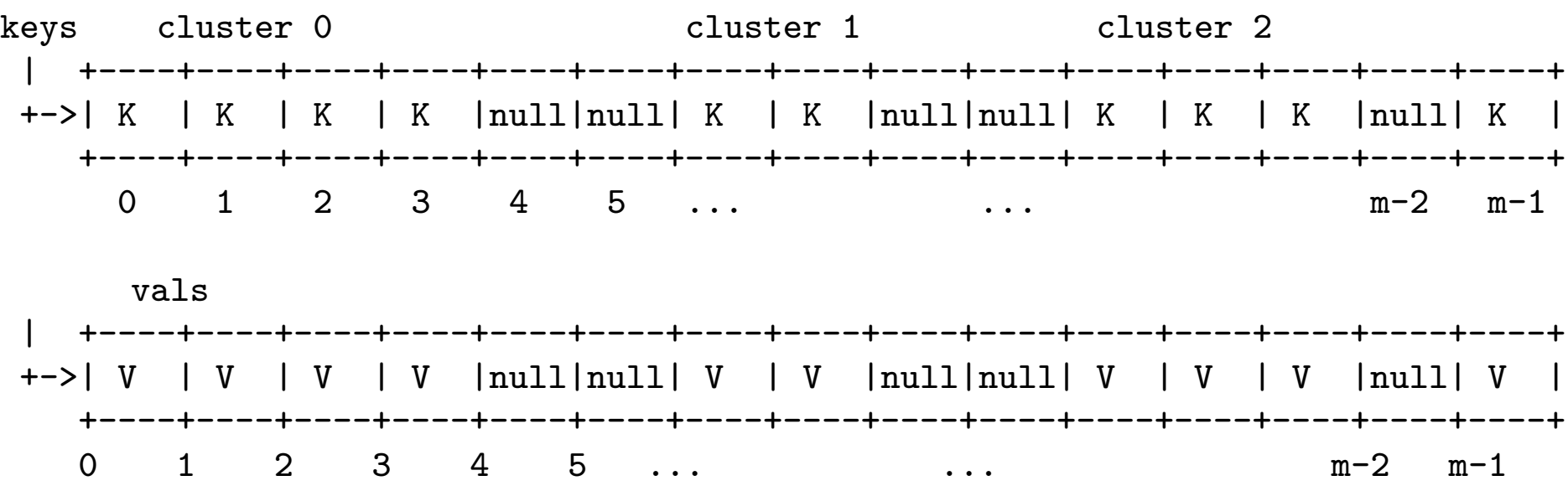
Quanto maior o fator de carga, mais tempo as funções de busca e inserção vão consumir.

Durante a busca há três possibildades:

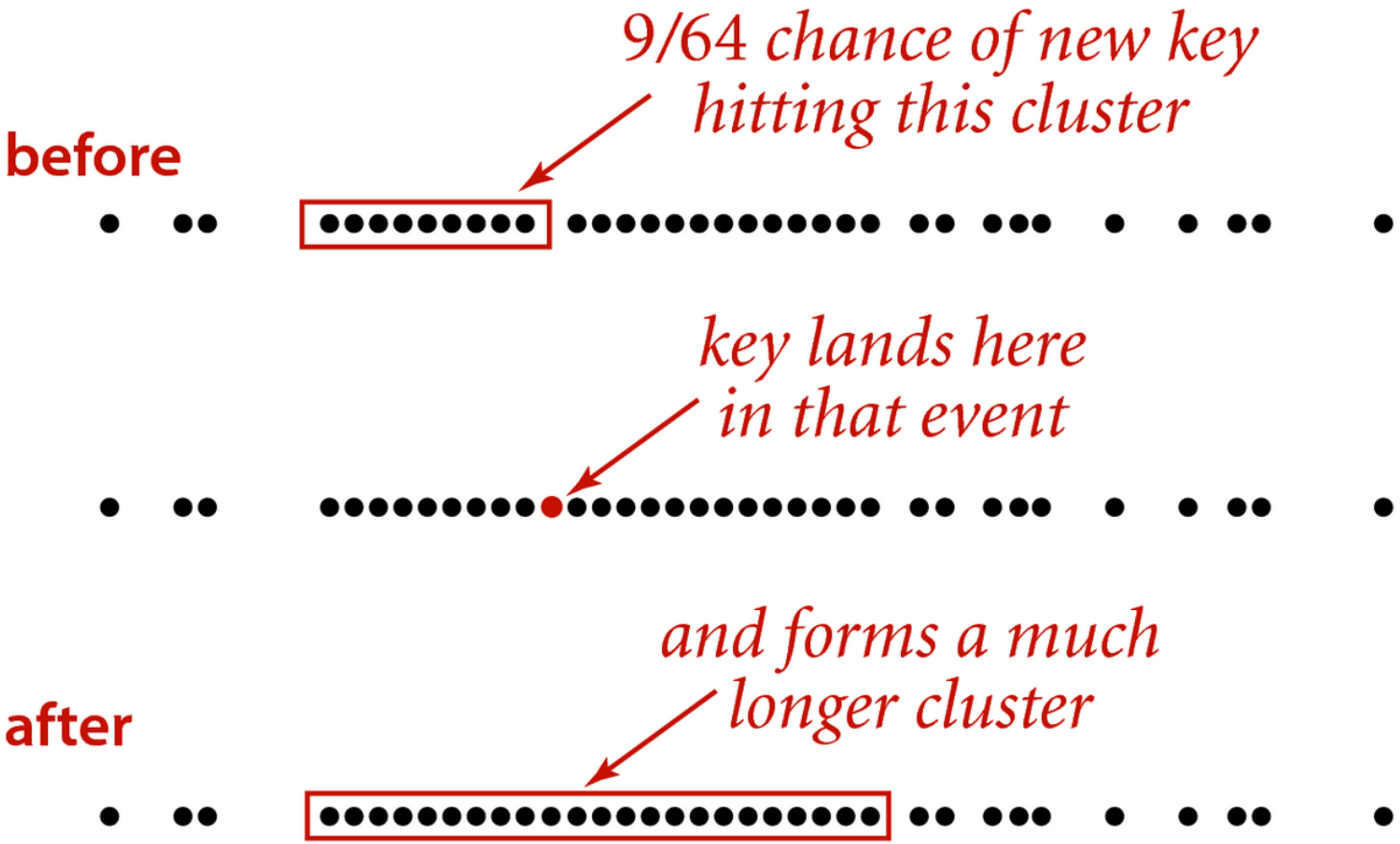
- encontramos a chave, paramos a busca;
- posição não-ocupada, paramos a busca;
- posição está ocupada e não é a chave, vamos para a próxima posição

LinearProbingHashST

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



Clusters



Clustering in linear probing ($M = 64$)

Fonte:algs4

Clusters

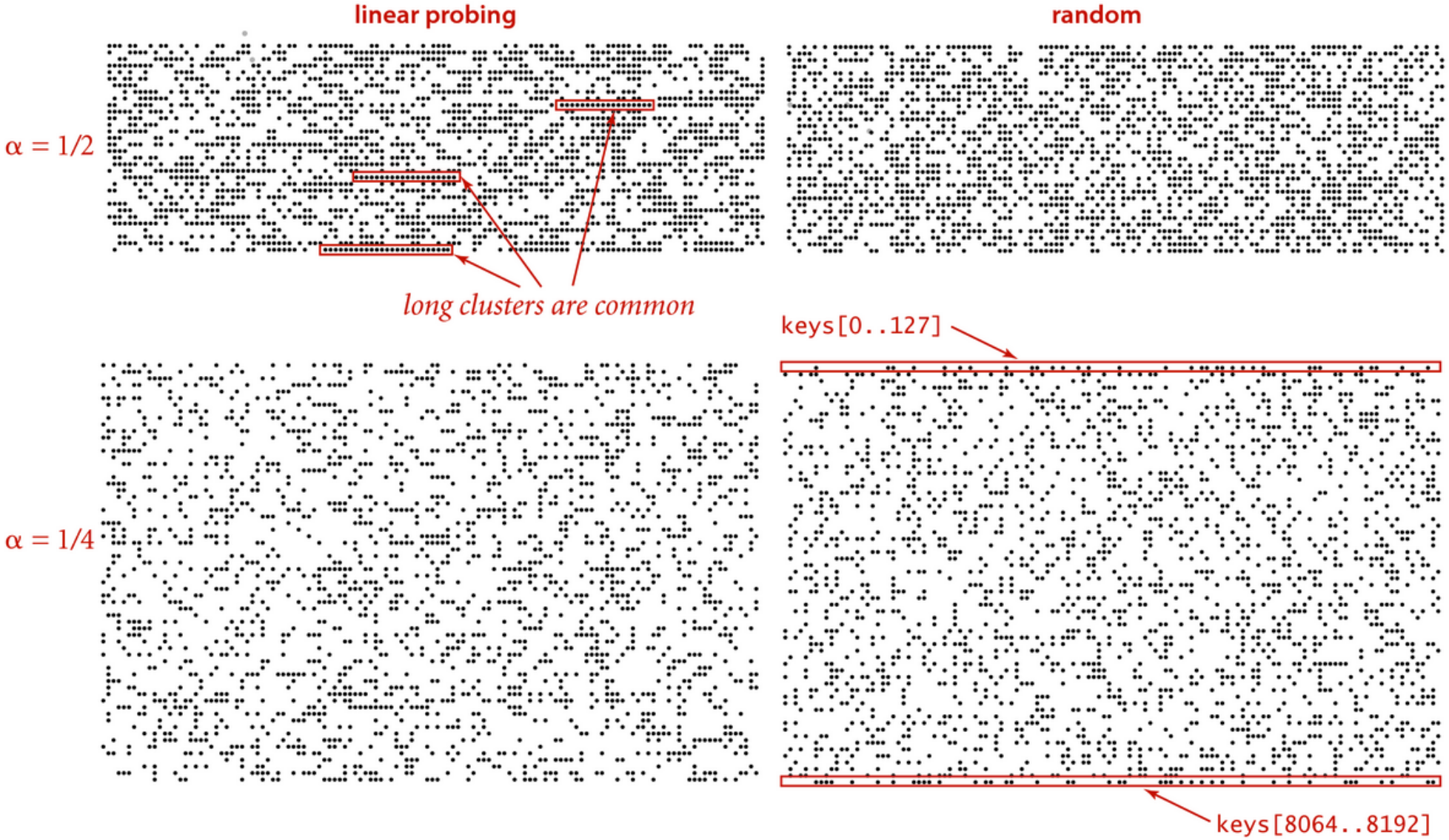


Table occupancy patterns (2,048 keys, tables laid out in 128-position rows)

Colisões por sondagem linear

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

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];
}
```

Colisões por sondagem linear

```
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

```
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

Retorna todas as chaves na ST como iterável.

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

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

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

Rehashing

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

   // encontre key
   int i = hash(key);
   while (!key.equals(keys[i]))
       i = (i + 1) % m;

   // remova key
   keys[i] = null;
   vals[i] = null;
   n--;
```

Rehashing

```
    // rehash todas as chaves no cluster
    i = (i + 1) % m;
    while (keys[i] != null) {
        Key keyToRehash = keys[i];
        Value valToRehash = vals[i];
        keys[i] = null;
        vals[i] = null;
        n--;
        put(keyToRehash, valToRehash);
        i = (i + 1) % m;
    }
    // resize se estiver alfa <= 1.2.5
    if (n > 0 && 8*n<= m) resize(m/2);
}
```

Análise

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).

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

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

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.

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*

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

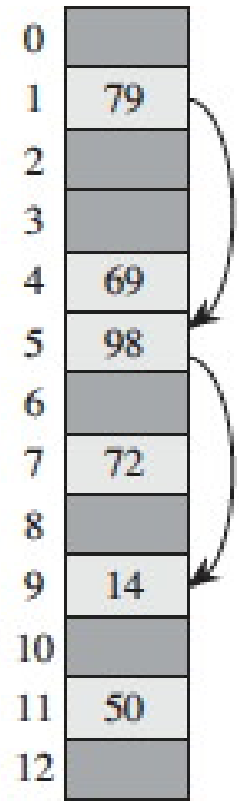
$$\texttt{h}(\texttt{key}, \texttt{i}) = (\texttt{h1}(\texttt{key}) + \texttt{i} \times \texttt{h2}(\texttt{key}))\%\texttt{m}.$$

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.

Duas métodos utilizados:

- tome `m` como um potência de 2 e `h2()` que sempre forneça inteiros impares;
- 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 que 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!