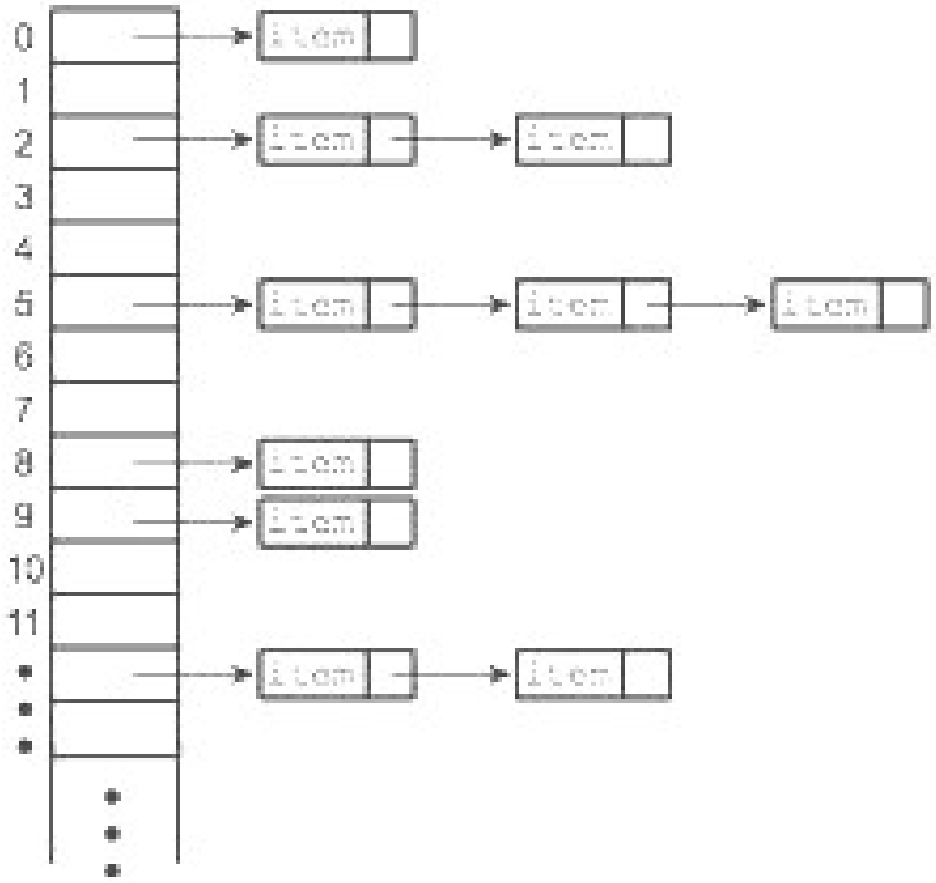
Tabelas Hash



Prof. Renê R. Veloso

Dicionários

Muitas aplicações exigem um conjunto dinâmico que admita apenas operações de ***inserção***, ***remoção*** e ***pesquisa***.

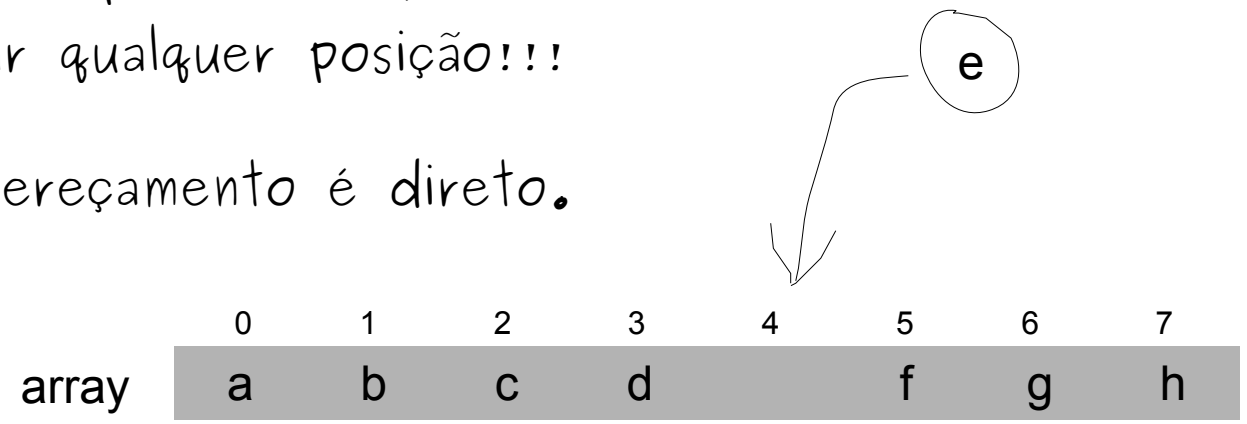
Outras funções como: sucessor, mínimo, máximo e etc, não são diretamente implementadas.

Tabela?

Podemos iniciar dizendo que a idéia de *tabela* vem dos arranjos simples...

É rápido para inserir, remover e acessar qualquer posição!!!

O endereçamento é direto.



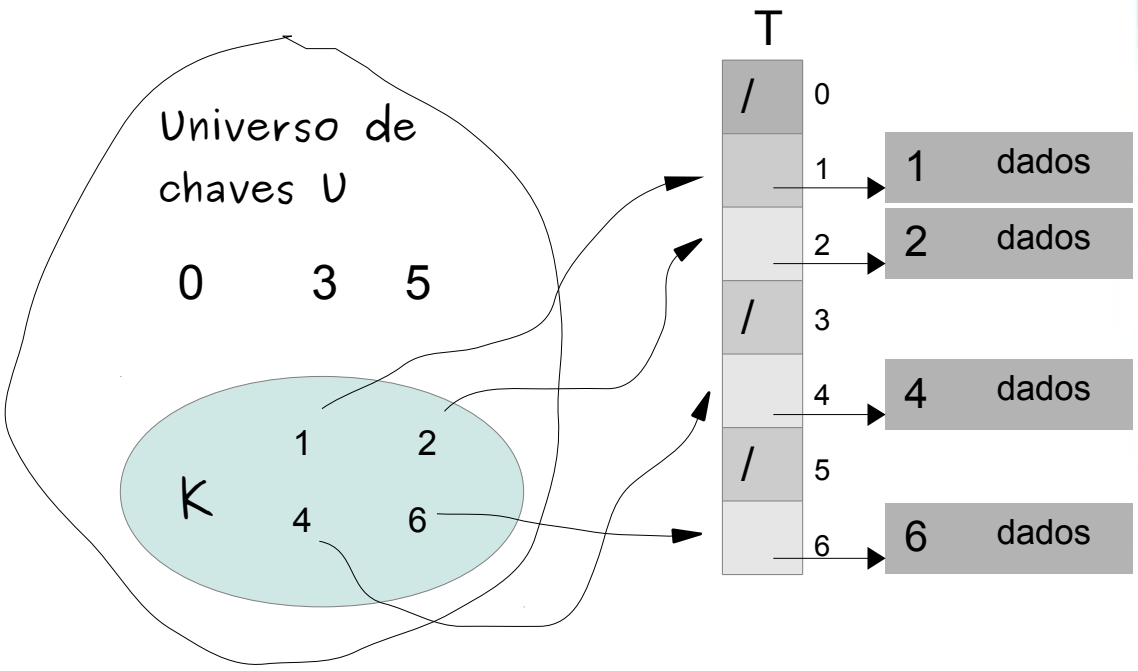
Tabelas de endereçamento direto

- Ideal para quando o universo (U) de chaves é pequeno.

```
Busca_endereço_direto(T,k)
    Return T[k]

Inserção_end_direto(T,x)
    T[chave[x]] ← x

Remoção_end_direto(T,x)
    T[chave[x]] ← NULL
```



Operações rápidas: tempo constante

O problema do endereçamento direto é óbvia.. é.. mas qual?

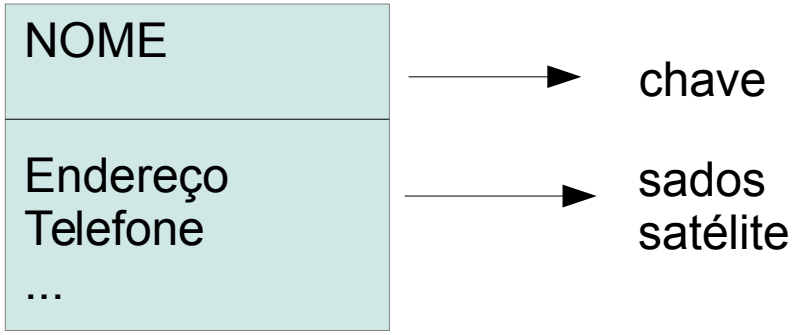
O problema do endereçamento direto é óbvia.. é.. mas qual?

Se o universo U de chaves é grande, o armazenamento de uma tabela T de tamanho |U| pode ser impraticável... ou mesmo impossível, por causa de limitações de memória.

Além disso, o conjunto K de chaves (realmente utilizadas) pode ser tão pequeno em relação a U que a maior parte do espaço alocado para T seria desperdiçada.

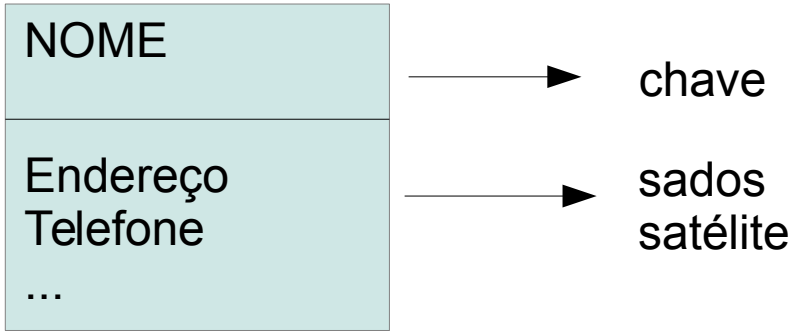
E se a chave for uma string ?

Por exemplo, se precisamos armazenar informações como nome e endereço, telefone de pessoas.. onde o nome é a chave.

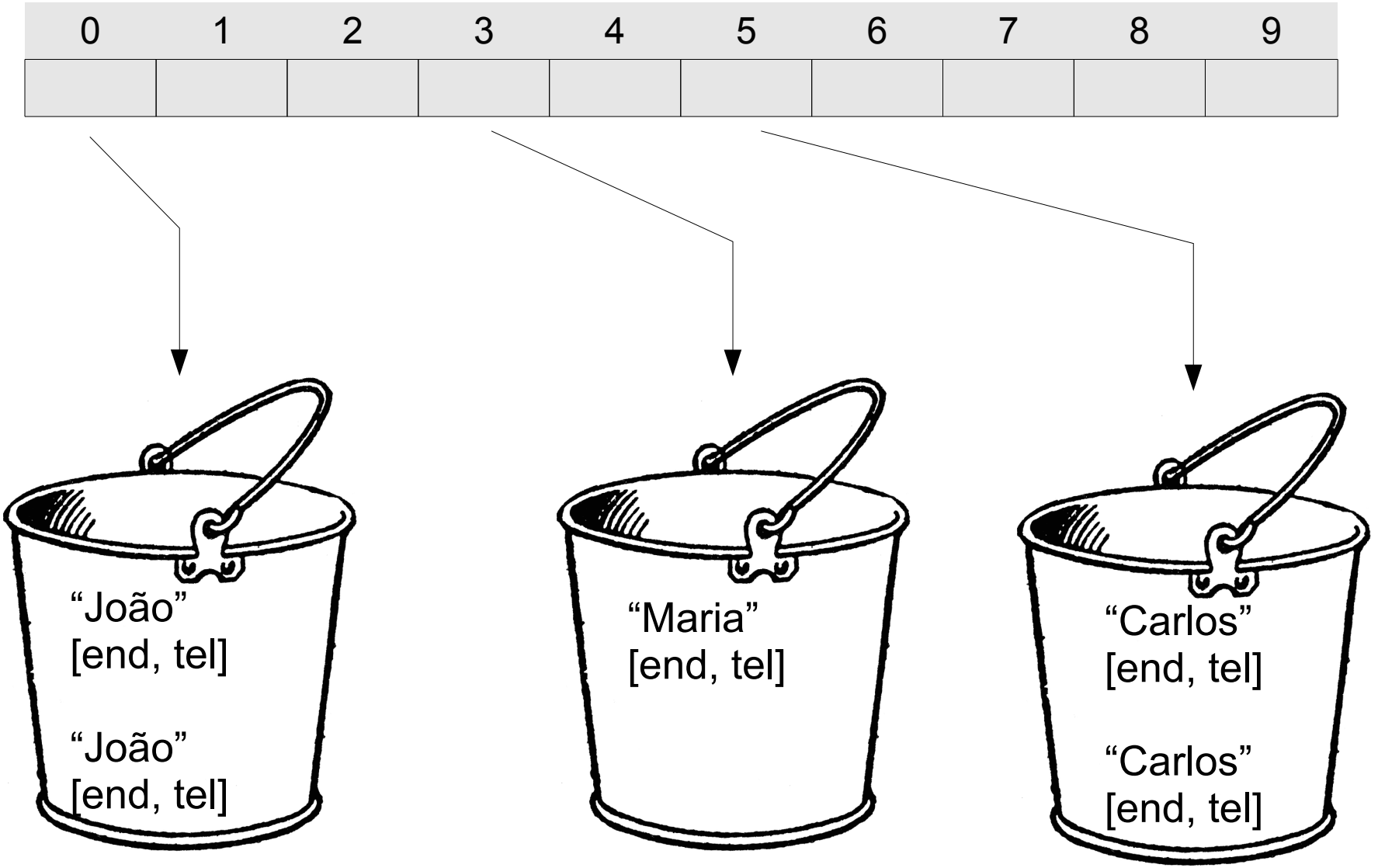


E se a chave for uma string ?

Por exemplo, se precisamos armazenar informações como nome e endereço, telefone de pessoas.. onde o nome é a chave.

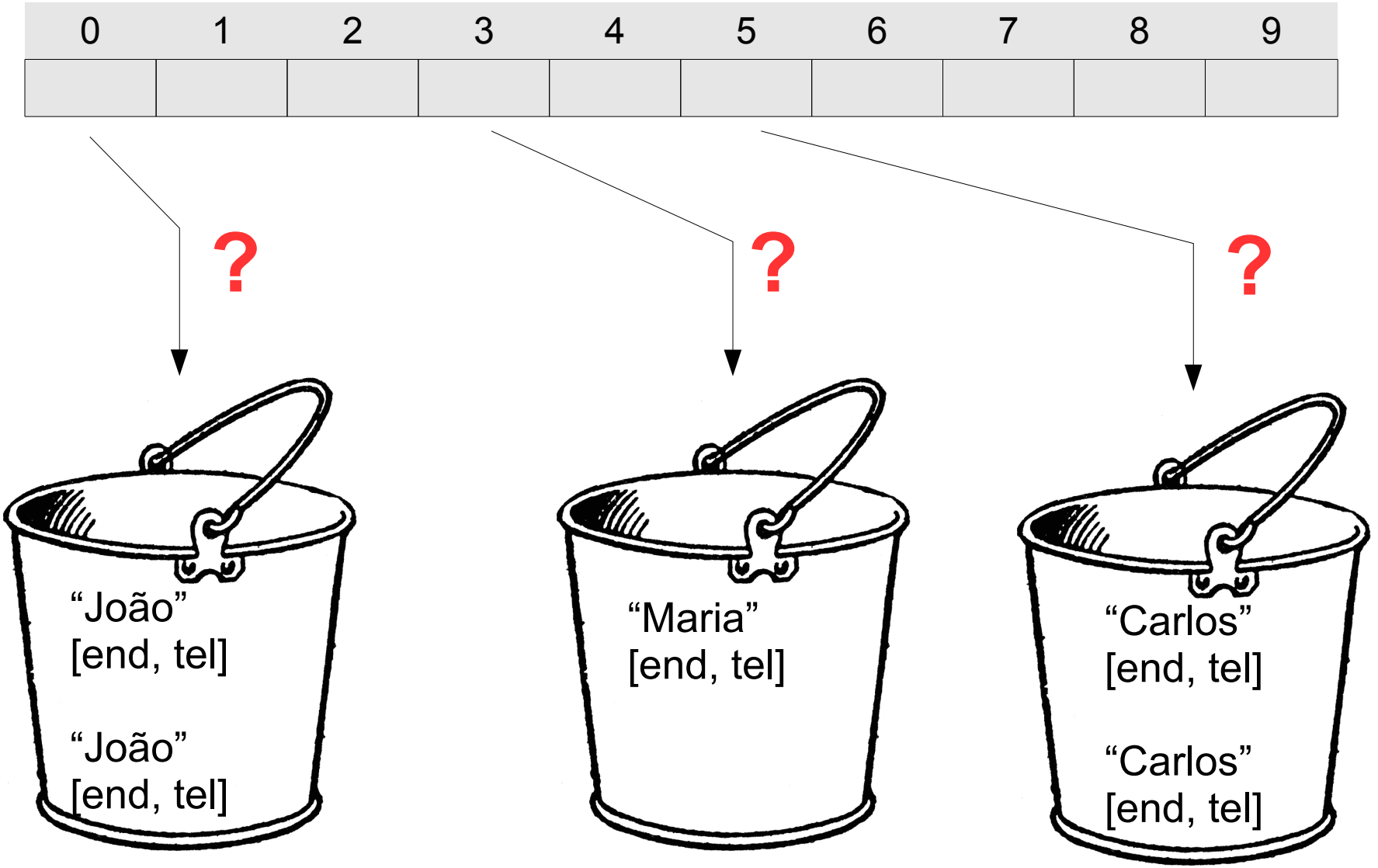


Podemos usar um ARRANJO DE BUCKETS:



ARRANJO DE BUCKETS:

Como mapear uma string em uma posição do vetor... ?



Tabelas hash

Quando o conjunto K de chaves armazenadas em um dicionário é muito menor que o universo U de todas as chaves possíveis, uma tabela hash exige muito menos espaço de armazenamento que uma tabela de endereçamento direto.

Os requisitos de armazenamento são reduzidos para, no máximo, $|K|$ embora a pesquisa ainda exija o *tempo médio* constante.

- Hash = dispersão, espalhamento

Tabelas hash

Quando o conjunto K de chaves armazenadas em um dicionário é muito menor que o universo U de todas as chaves possíveis, uma tabela hash exige muito menos espaço de armazenamento que uma tabela de endereçamento direto.

Os requisitos de armazenamento são reduzidos para, no máximo, $|K|$ embora a pesquisa ainda exija o *tempo médio* constante.

- Uma ***função hash*** *h* é utilizada para calcular uma posição a partir de uma chave *k*.
  - *h* mapeia o universo *U* de chaves nas posições de uma tabela hash *T* [0..n]

Termos importantes

- Hash table
  - Em termos mais técnicos: *array associativo*
    - Estrutura composta por pares (chave, valor).
- Chave *k* é ***mapeada*** para a posição *h(k)*
- *h(k)* é o ***valor hash*** da chave *k*

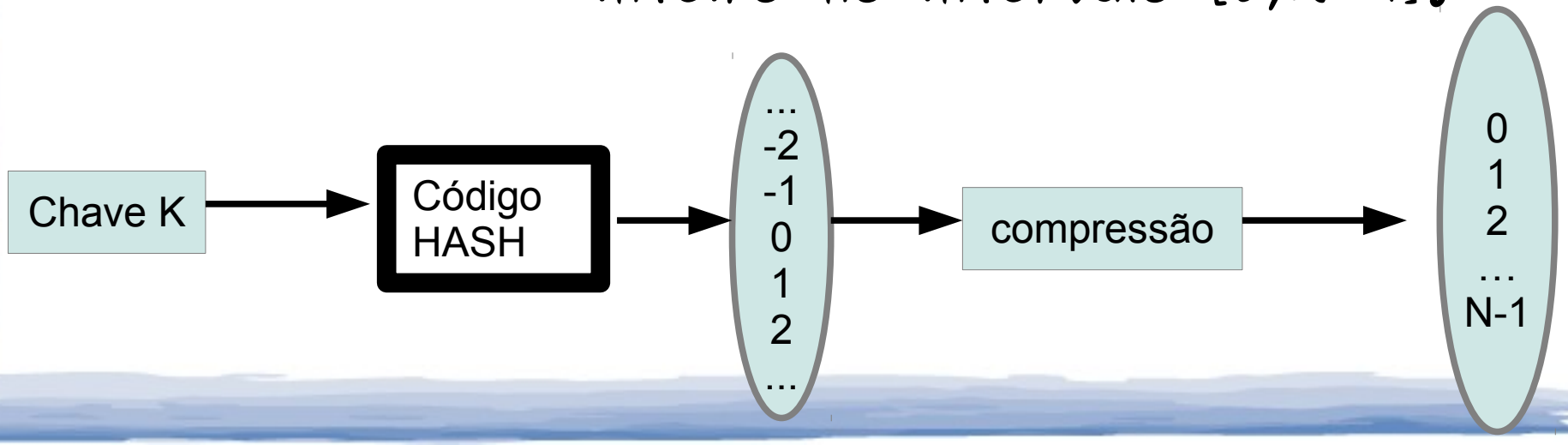
Funções Hash

Mapeia uma chave em um inteiro no intervalo [0,N−1]

Onde N é o tamanho do arranjo.

Duas partes:

- código hash: mapeia uma chave em um número inteiro qualquer.
- função de compressão: mapeia um inteiro em um inteiro no intervalo [0,N−1].



Código hash

- Inteiros longos para inteiros

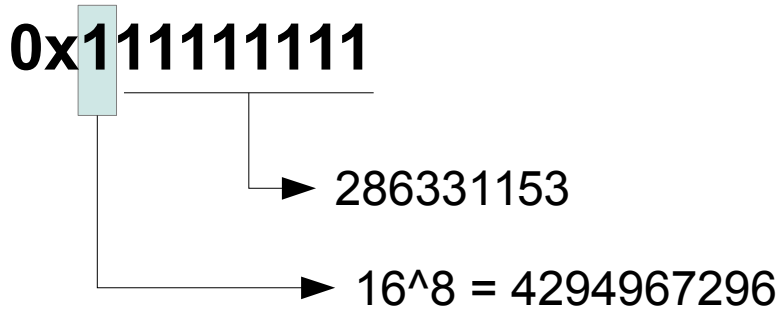
```
int hashcode(long x) {
   int p1, p2;
   p1 = (int) x;
   p2 = (int) (x>>32);
   return (p1+p2);
}
```

Exemplo:

x = 0x111111111 = 4581298449

P1 = 286331153
P2 = 1

hashcode(x) = 286331154



Código hash

- Strings para inteiros: *somar códigos ASCII*

```
int hashcode(char *s) {
   int p=0, i;
   for (i=0; s[i]!='\0'; i++)
      p=p+s[i];
   return p;
}
```

Exemplo:

S = “abc”

'a' = 97
'b' = 98
'c' = 99

= 294

Qual é o problema dessa abordagem?

Código hash

- Strings para inteiros: *somar códigos ASCII*

```
int hashcode(char *s) {
   int p=0, i;
   for (i=0; s[i]!='\0'; i++)
      p=p+s[i];
   return p;
}
```

Exemplo:

S = “abc”

'a' = 97
'b' = 98
'c' = 99

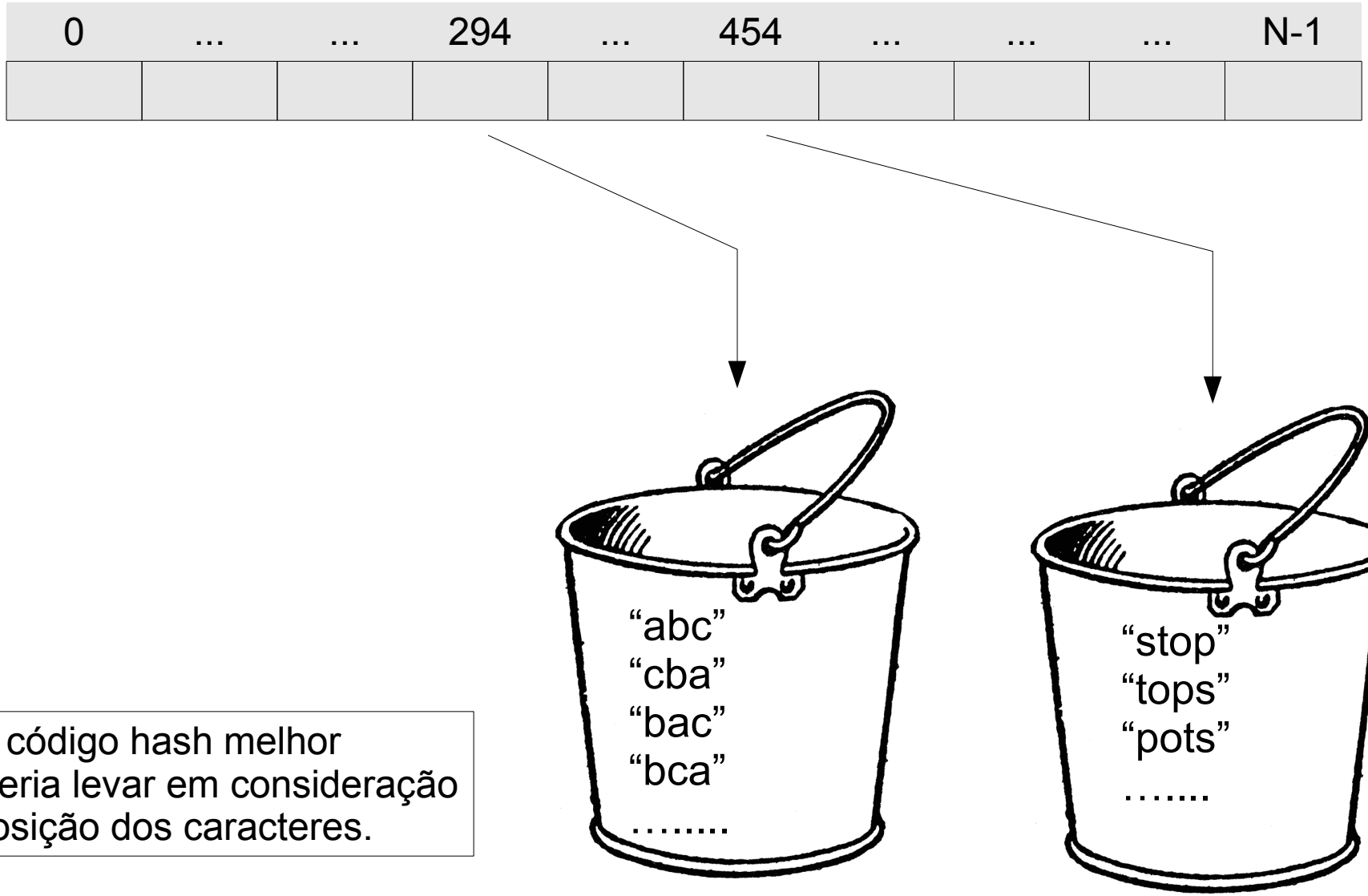
= 294

Qual é o problema dessa abordagem?

“abc” == “cba” == “bca” == “acb” = …

“stop” == “pots” == “tops” == ...

Facilita a ocorrência de colisões...



Um código hash melhor deveria levar em consideração a posição dos caracteres.

Código hash

- Strings para inteiros: *código polimonial*

– é como é feito na conversão de bases para a base 10:

$101_{(2)} = ?_{(10)} \rightarrow 1\times2^2+0\times2^1+1\times2^0 = 4+0+1 = 5_{(10)}$

$1234(10) = ?(10) \rightarrow 1\times10^3+2\times10^2+3\times10^1+4\times10^0 = 1000+200+30+4 = 1234_{(10)}$

– supondo que uma string está na base 127:

$\text{"abc"}_{(127)} = ?_{(10)} \rightarrow 97\times127^2+98\times127^1+99\times127^0 = 1564513 + 12446 + 99$

$= 1577058_{(10)}$

$\text{"abcd"}_{(127)} = 200286466_{(10)}$

Utilizar a base 127 torna o resultado muito grande, estudos sugerem que utilizar as bases 33, 37, 39 ou 41 são bons valores para a língua inglesa.

Código hash

- Strings para inteiros: *código polimonial*

Para um string $S = x_0 x_1 x_2 \ldots x_{n-1}$:

$$\text{hashcode}(S) = x_0 a^{n-1} + x_1 a^{n-2} + \ldots + x_{n-3} a^2 + x_{n-2} a^1 + x_{n-1} a^0$$

$$= x_{n-1} a^0 + x_{n-2} a^1 + x_{n-3} a^2 \ldots + x_1 a^{n-2} + x_0 a^{n-1}$$

$$= x_{n-1} + a(x_{n-2} + x_{n-3} a^1 + \ldots + x_1 a^{n-3} + x_0 a^{n-2})$$

$$= x_{n-1} + a(x_{n-2} + a(x_{n-3} + \ldots + x_1 a^{n-4} + x_0 a^{n-3})$$

$$= \ldots$$

$$= = x_{n-1} + a(x_{n-2} + a(x_{n-3} + a(\ldots + a(x_1 + a x_0) \ldots)$$

→ chamado de Regra de Horner.

Para uma base **a**, tem-se:

$$\text{"abcd"}_{(a)} = 100 + a(99 + a(98 + a(97)))$$

Código hash

- Strings para inteiros:
  - *código polimonial usando a regra de Horner*

```
int hashcode(char *s) {
   int r=0, i;
   for (i=0; s[i]!='\0'; i++)
      r=(int)s[i]+a*r;
   return r;
}
```

Código hash

- Strings para inteiros:
  - *Com shift*

Realiza a multiplicaçãodo código polimonial usando shift:

```
int hashcode(char *s) {
   int r=0, i;
   for (i=0; s[i]!='\0'; i++)
      r=(int)s[i]+(r<<1);
   return r;
}
```

Para a=2

Código hash

- Strings para inteiros:
  - *Com shift cíclico*

Somatório parcial de certo número de bits:

```
int hashcode(char *s) {
   int r=0, i;
   for (i=0; s[i]!='\0'; i++){
      r = (r<<5) | (r>>27);
      r=(int)s[i]+r;
   }
   return r;
}
```

Resultado de estudos.
Em uma lista de mais
de 25 mil palavras, esse
shift mostrou-se promissor.

Código hash

- Strings para inteiros: *Com shift cíclico*

```
...
   r = (r<<5) | (r>>27);
   r=(int)s[i]+r;
...
```

Ex.: “abcd”

```
  (shift) r      => 00000000 00000000 00000000 00000000
 + 'a' (97)      => 00000000 00000000 00000000 01100001
  (shift) r      => 00000000 00000000 00000000 01100001
                 => 00000000 00000000 00001100 00100000 (<< 5)
                 => 00000000 00000000 00000000 00000000 (>>27)
                 => 00000000 00000000 00001100 00100000 (or)
 + 'b' (98)      => 00000000 00000000 00001100 00100000 (r)
                 => 00000000 00000000 00000000 01100010 ('b' = 98)
                 => 00000000 00000000 00001100 10000010 (r+b = 3202)
...
          r     = 3282116
```

Código hash

- Strings para inteiros:
  - *XOR hash*

Produz valores "aparentemente" aleatórios pelo agrupamento dos bits

```
int hashcode(char *s) {
   int r=0, i;
   for (i=0; s[i]!='\0'; i++)
      r = r ^ (int)s[i];
   return r;
}
```

* (bom) melhor do que apenas somar os decimais.
* (ruim) não altera muito o estado interno :-(

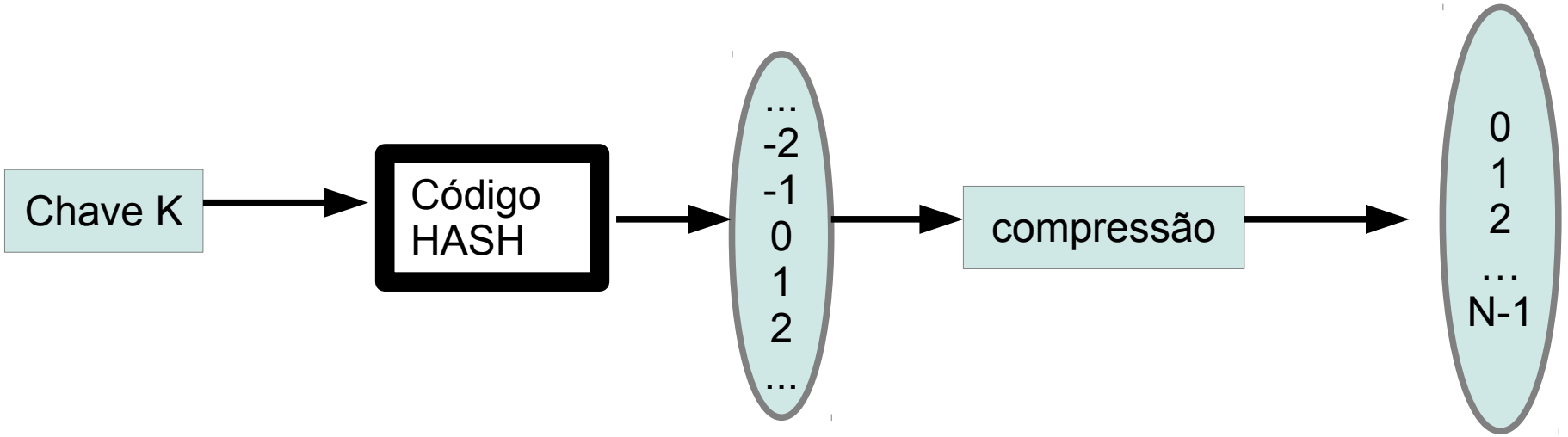
Código hash

- Strings para inteiros:
  - *Hash rotatório (aprimoramento do shift cíclico)*

```
int hashcode(char *s) {
   int r=0, i;
   for (i=0; s[i]!='\0'; i++){
      r = (r<<5) | (r>>27);
      r=(int)s[i]^r;
   }
   return r;
}
```

```
* (bom) resulta em melhor distribuição.
*          “mistura” melhor o código hash.
*          evita overflow   :-)
```

Funções de Compressão



a) Método da divisão: hashcode % N
   - onde N é o tamanho do arranjo de buckets.
   - a escolha de N como um nº primo ajuda a diminuir as colisões.

Ex.: distribuir os valores 200, 205, 210, 300, 305, 310 em um arranjo de tamanho 100 e 101.

Funções de Compressão

Ex.: distribuir os valores 200, 205, 210, 300, 305, 310 em um arranjo de tamanho 100 e 101.

```
200 % 100 = 0
205 % 100 = 5
210 % 100 = 10
300 % 100 = 0
305 % 100 = 5
310 % 100 = 10
```

Muitas colisões!

```
200 % 101 = 99
205 % 101 = 3
210 % 101 = 8
300 % 101 = 98
305 % 101 = 2
310 % 101 = 7
```

Nenhuma colisão!

O método da divisão: atenção!

Neste método, em geral se evitam certos valores de N.
Por exemplo, m não deve ser uma potência de 2 pois, se $N=2^p$ então $hash(k)$ será somente o grupo de p bits de mais baixa ordem de k.

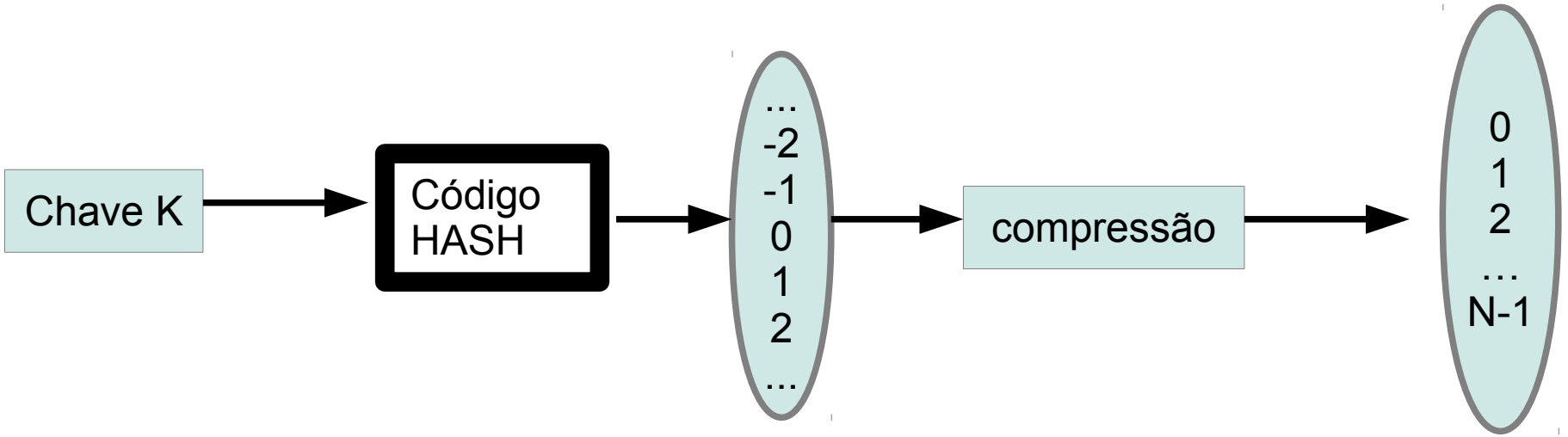
Um número primo não muito próximo a uma potência exata de 2 é uma melhor escolha para N.

Exemplo: para uma tabela que comporte $m=2000$ strings, onde um caractere tem 8 bits, aceitando até 3 elementos de colisão, alocamos uma tabela de tamanho $N=701$.

701 é primo próximo a 2000/3

Tratando cada chave k como um número inteiro, a função hash seria: $h(k) = k \% 701$

Funções de Compressão



b) Método MAD: (multiplicação, adição e divisão)

: ((a * hashcode + b) % p)%N

- onde;
  - N é o tamanho do arranjo de buckets.
  - a é um inteiro entre [1, p−1]
  - b é um inteiro entre [0, p−1]
  - p é um número primo

Funções de Compressão

b) Método MAD: (multiplicação, adição e divisão)

: ((a * hashcode + b) % p)%N

Ex.: hash(r) = ((3 * r + 1)%12)%10)

Hashcode r	=	0	1	2	3	4	5	6	7	8	9	10	11
hash(r)	=	1	4	7	0	1	4	7	0	1	4	7	0

repetição

Usamos p = 12, que não é primo.

Funções de Compressão

b) Método MAD: (multiplicação, adição e divisão)

: ((a * hashcode + b) % p)%N

Ex.: hash(r) = ((3 * r + 1)%13)%10)

Hashcode r	=	0	1	2	3	4	5	6	7	8	9	10	11
hash(r)	=	1	4	7	0	0	3	6	9	2	2	5	8

Funções de Compressão

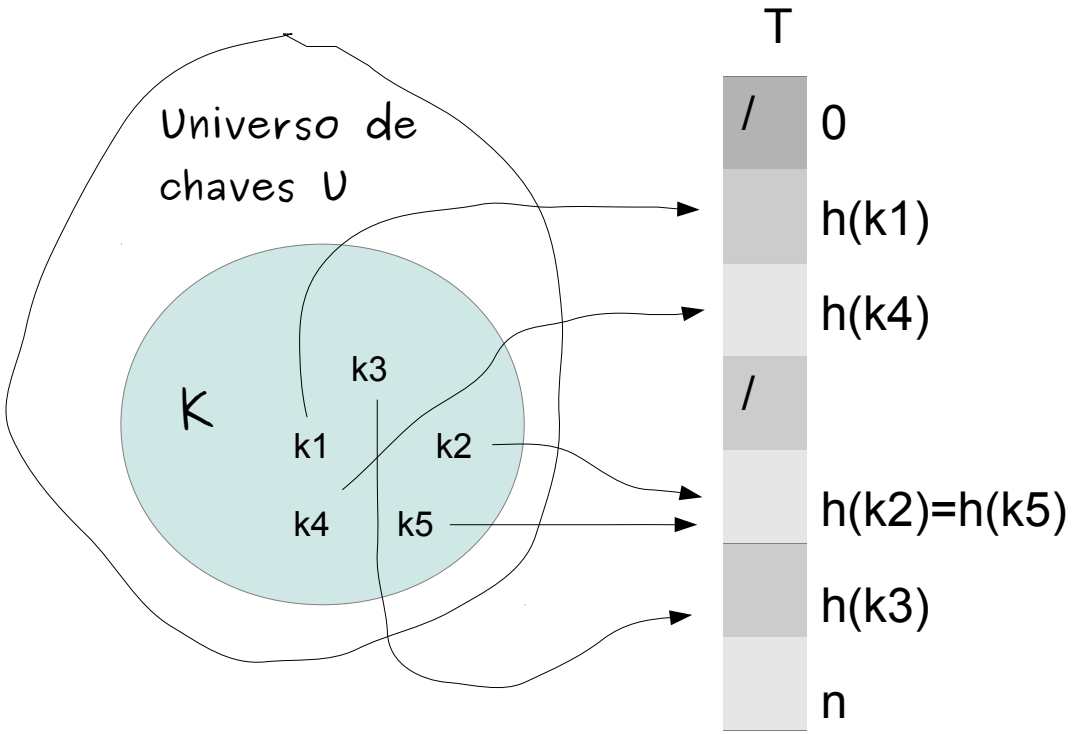
Outras funções hash:

- método da dobra
- método da multiplicação

Pesquise !!!

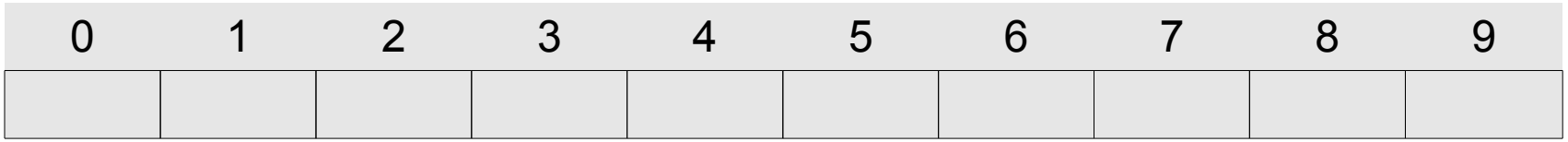
Tratamento de Colisões

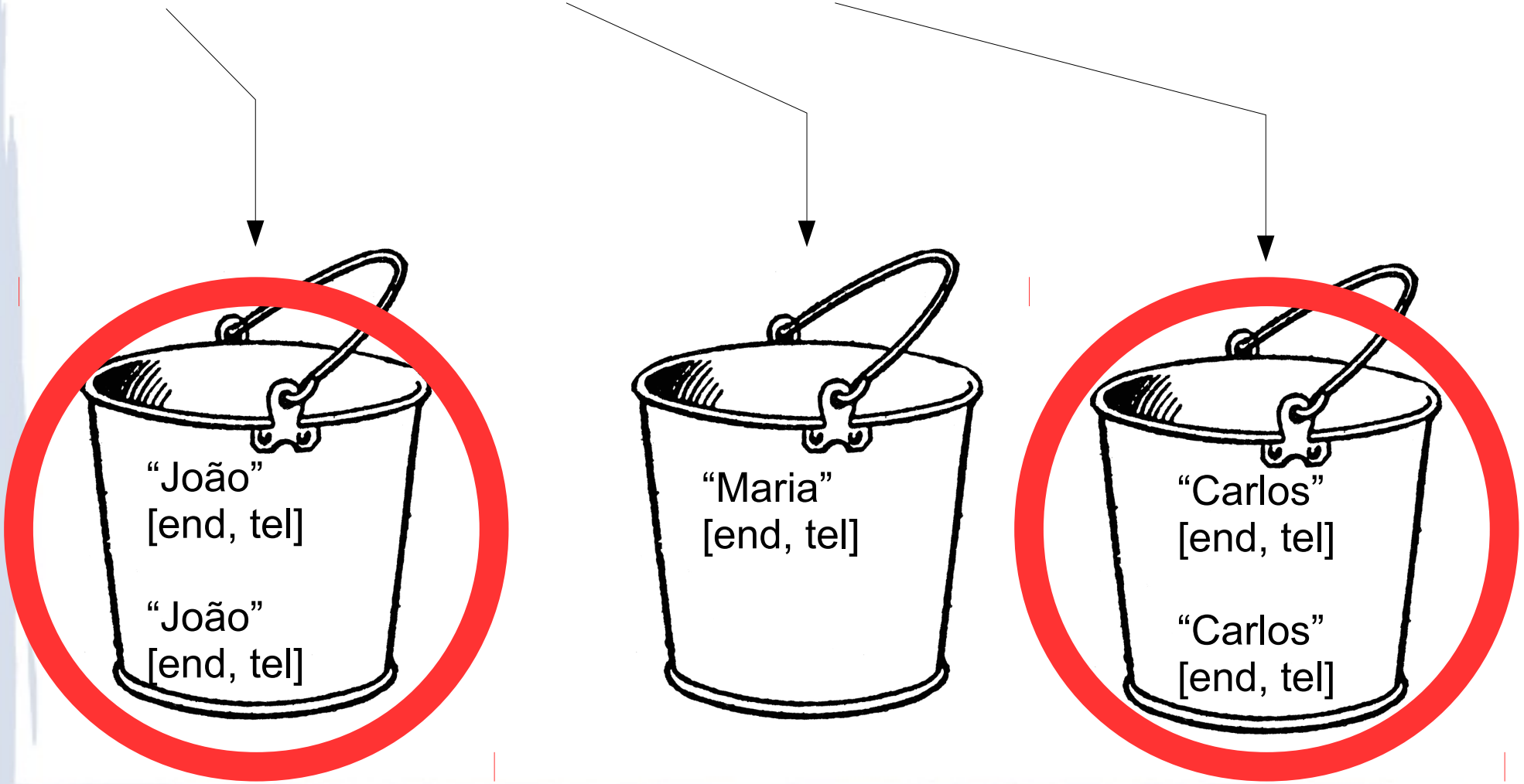
Colisões...



Duas (ou mais) chaves podem ter o hash na mesma posição...
chamamos essa situação de *colisão.*

Colisões !


0
1
2
3
4
5
6
7
8
9


“João”
[end, tel]
“João”
[end, tel]
“Maria”
[end, tel]
“Carlos”
[end, tel]
“Carlos”
[end, tel]

Colisões

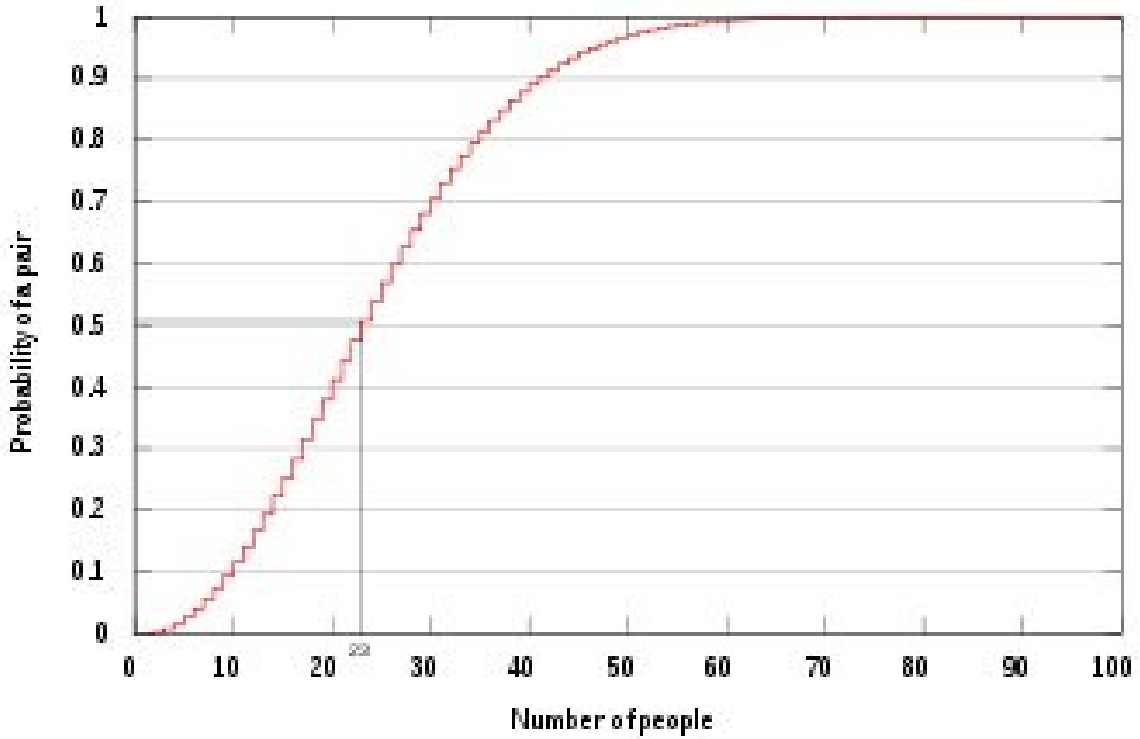
A idéia é evitar ao máximo as colisões... No entanto, a função hash deve ser determinista (dada uma entrada k, sempre deve produzir a mesma saída $h(k)$) e como $|U|$ é maior do que n, devem existir duas chaves com o mesmo valor hash.

Problema do aniversário :-(

Por exemplo: se 2500 chaves são mapeadas em 1 milhão de posições, mesmo com uma distribuição perfeitamente uniforme, de acordo com o ***problema do aniversário***
há uma chance de 95% de que pelo menos duas das chaves sejam mapeadas para a mesma posição.

Problema do aniversário

- Em teoria das probabilidades, o ***problema do aniversário*** afirma que dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas terão a mesma data de aniversário é de mais de 50%.



100% para mais de 367 pessoas.

Problema do aniversário

Para calcular aproximadamente a probabilidade de que em uma sala com n pessoas, pelo menos duas possuam o mesmo aniversário, p(n) desprezamos variações na distribuição, tais como anos bissextos, gêmeos, variações sazonais ou semanais, e assumimos que 365 possíveis aniversários são todos igualmente prováveis.

É mais fácil calcular a probabilidade $\overline{p}(n)$, para todos os aniversários sejam diferentes. Se n>365, pelo **Princípio da Casa dos Pombos** esta probabilidade é 1. Por outro lado, se $n \leq 365$, ele é dado por:

$$\overline{p}(n) = 1.(364/365).(363/365).(362/365)\ldots((365-(n-1))/365)$$
$$= 365!/365^n(365-n)!$$
$$p(n) = 1 - \overline{p}(n)$$

A segunda pessoa não pode ter nascido no mesmo dia da primeira

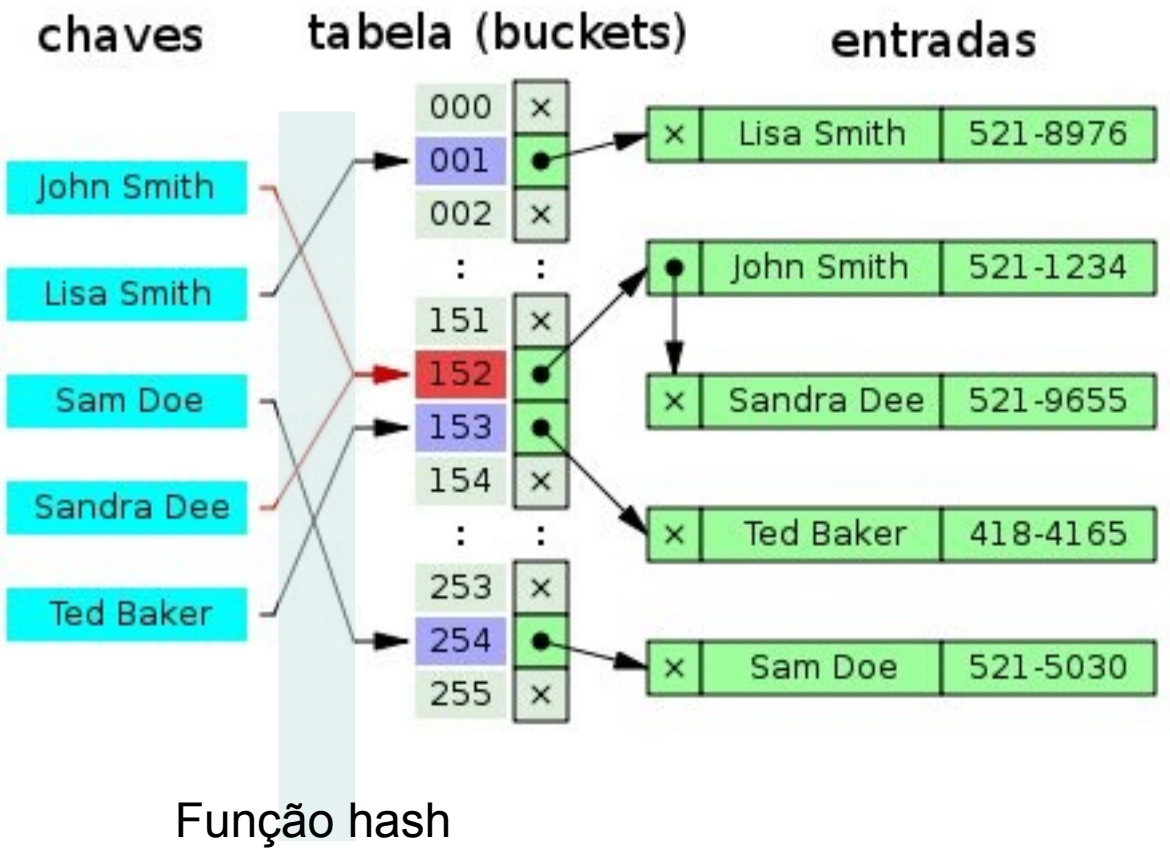
Resolução de colisões

- Toda tabela hash deve implementar algum método para tratar colisões. Dois método bem conhecidos são:
  - Resolução por encadeamento
  - Resolução por endereçamento aberto

Fator de carga: o desempenho da maioria dos métodos de resolução não depende diretamente do número m de chaves armazenadas, mas sim do *fator de carga da tabela*, que é a razão m/**N** entre **m** e o tamanho **N** da tabela (array). Que representa a porção dos N lugares na estrutura que são preenchidos com uma das m entradas armazenadas.

Estudos apontam que a probabilidade de colisão aumenta com cargas acima de 0.7 (2/3 cheia).

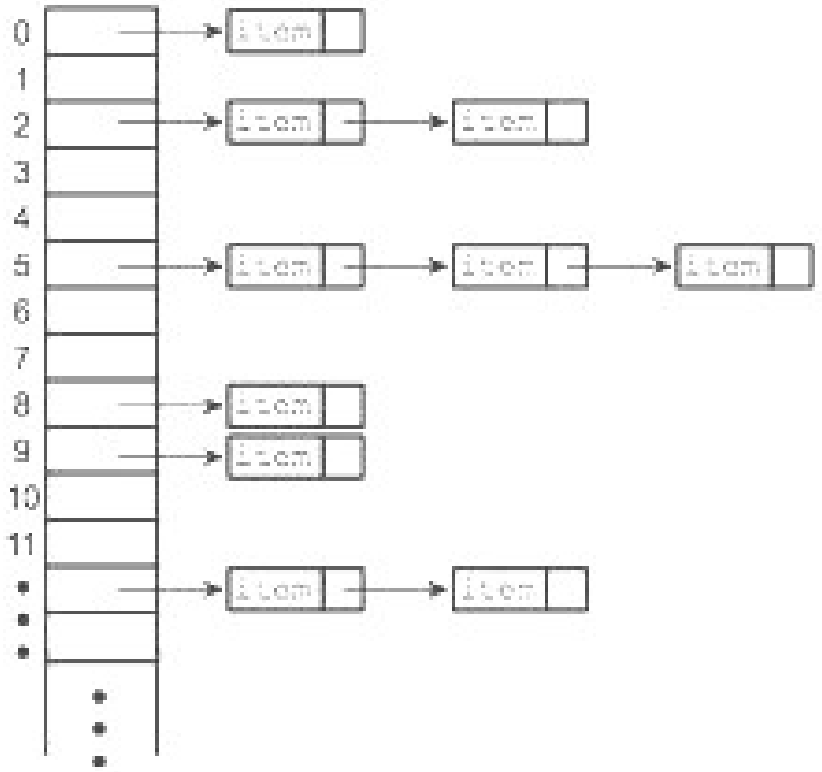
Resolução de colisão por encadeamento



Tempo de inserção no pior caso é constante.

Tempo de pesquisa e remoção é proporcional ao comprimento da lista

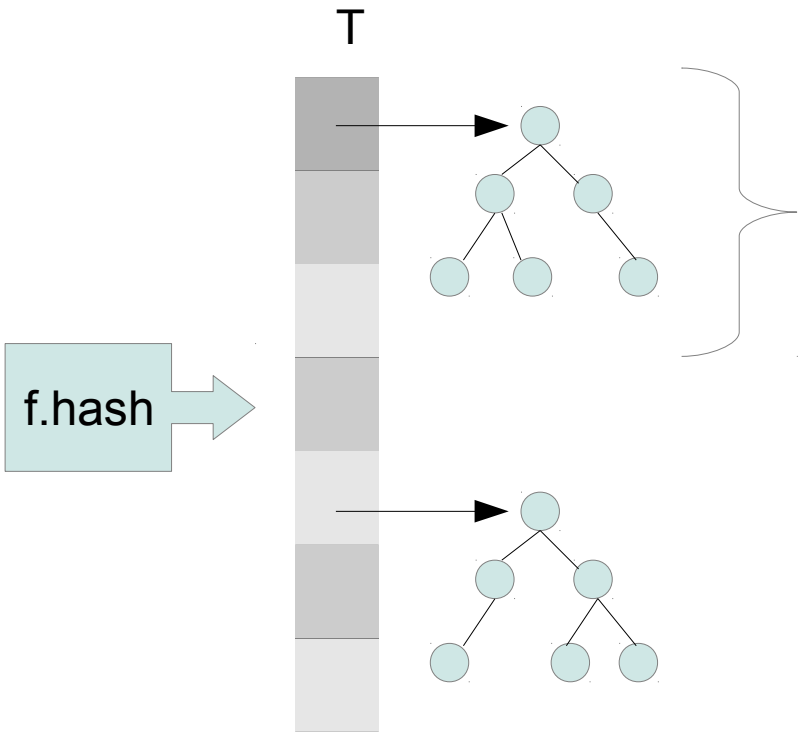
Resolução de colisão por encadeamento



O desempendo deste tipo de tratamento cai de forma suave com o fator de carga. Por exemplo, uma tabela com 1000 posições e 10000 chaves armazenadas (fator de carga 10) é de cinco a dez vezes mais lenta do que uma tabela com 10000 posições (fator 1). Mas ainda é 1000 vezes mais rápida do que uma lista simples e possivelmente mais rápida do que uma AVL.

Resolução de colisão por encadeamento

- É possível melhorar o pior caso:



Estruturas mais sofisticadas de dados, como árvores de busca balanceadas, são boas se o fator de carga for grande (cerca de 10 ou mais), ou se a distribuição de *hash* não for muito uniforme ou memso quando se quer garantir um bom desempenho no pior caso.

No entanto, usar uma tabela grande ou uma função hash melhor será sempre mais eficiente!!!

Trabalho hash (1)

- Construir um programa para:
  - Ler o conteúdo do arquivo “Memórias Póstumas de Brás Cubas” em formato txt.
  - Utilizar uma tabela hash para contar a quantidade de cada palavra encontrada no texto.
    - Ignorando coisas óbvias como pontuações, caracteres especiais ($,#,@,&, etc), ignorando a caixa (maiúsculas e minúsculas),...
  - Utilizando resolução de colisões por encadeamento.
  - Utilizando as funções de hashcode e compressão que preferir
  - O programa deve emitir um relatório, contendo:
    - Palavras encontradas e a quantidade de cada uma
    - Tempo de processamento total

Trabalho hash (1)

- Trabalho individual
- Valor: a ser definido no quadro.
- A ser apresentado em laboratório na data marcada no quadro.

Endereçamento aberto

Endereçamento aberto

- No endereçamento aberto todos os elementos são armazenados na própria tabela hash, isto é, não existem listas nem elementos armazenados fora da tabela, evitando assim o uso de ponteiros.
- A vantagem de se utilizar endereçamento aberto é que a quantidade de memória utilizada para armazenar ponteiros é utilizada para aumentar o tamanho da tabela, possibilitando menos colisões e aumentando a velocidade de recuperação das informações.
- Para inserir um novo elemento, examinamos sucessivamente a tabela até encontrarmos uma posição vazia onde possamos armazenar o elemento.
    - Um ponto importante é que não percorremos sempre a tabela inteira, isto é, a busca depende do elemento a ser inserido.
- É feita uma “sondagem” para cada chave, para determinar a posição de inserção. A função hash é associada a um número que, juntamente com a chave, determinam quais posições serão verificadas.

Endereçamento aberto

- No endereçamento aberto, exigimos que, para toda chave *k*, a sequência de sondagem <*h(k,0), h(k,1), ..., h(k,N-1)*> seja uma permutação de <0,1,2,...,*N*-1>, de forma que toda posição da tabela seja eventualmente considerada uma posição para uma nova chave, à medida que a tabela é preenchida.

```
int Hash_Insere(Elemento x) {
  int j,i=0;
  do{
    j = hash(chave[x],i);

    if (T[j] == VAZIO) {
      T[j] = x;
      return TRUE;
    }
    else i++;
  }while(i < tamanho[T]); //tamanho[T] == N
  return FALSE;
}
```

Exige que a tabela tenha indicações de espaços vazios

Endereçamento aberto

- A busca por algum elemento deve seguir o mesmo esquema de sondagem da inserção para uma chave *k*.
  - A eliminação de algum elemento não é tão simples.
  - Deve-se prestar atenção para que, ao remover um elemento, a posição não fique simplesmente nula. É importante sinalizar a posição como VAZIA de forma explícita, evitando assim, que elementos posteriores ao que foi removido não sejam perdidos.

Tipos de sondagem

- Sondagem linear
- Sondagem quadrática
- Hash duplo

Sondagem linear

- Utiliza uma função hash auxiliar comum

  hash': U → {0,1,2,...,m-1}

  Assim: ***hash(k,i) = (hash'(k) + i) % N***

- Tendo em vista que a posição inicial de sondagem determina toda a sequência de sondagem, só existem *N* sequências de sondagem distintas.
- Fácil de implementar, mas não é boa. Pois sofre de **agrupamento primário.** Longas sequências de posições agrupadas são construídas, aumentando o tempo médio de pesquisa.

Sondagem quadrática

- Utiliza uma função hash da forma:

$$hash(k,i) = (hash'(k) + c_1 i + c_2 i^2) \% N$$

- Onde *hash'* é uma função auxiliar, c_1 e c_2 são constantes diferentes de zero.
- As posições sondadas são deslocadas por quantidades que dependem de forma quadrática do número de sondagem *i*.
- Exemplo: h(k) = (h'(k) + 0.5*i + 0.5*i*i)%16
  - Para N = 16 e h'(k) = k % 16

Sondagem quadrática

- Utiliza uma função hash da forma:

  $$hash(k,i) = (hash'(k) + c_1 i + c_2 i^2)$$

- Para fazer uso de toda a tabela, os valores de c_1, c_2 e N devem ser limitados.
- Problema de agrupamento secundário
  - Se $hash(k1,0) == hash(k2,0)$ então

    $hash(k1,i) == hash(k2,i)$

Sondagem quadrática

- Se $hash(k,i) = (hash'(k) + i + i^2) \% N$, então a sequência de sondagem será $hash'(k)$, $hash(k)'+2$, $hash(k)'+6$,…
- Para $N = 2^m$, com m chaves, boas constantes são $c1 = c2 = 1/2$. Leva a uma sequência de sondagem $hash(k)'$, $hash(k)'+1$, $hash(k)'+3$, $hash(k)'+6$,…
- Para um primo $N > 2$, a maioria das escolhas para c1 e c2 irá tornar $hash(k,i)$ distinta para i em $[0,(N-1)/2]$. Tais escolhas incluem $c1 = c2 = 1/2$, $c1 = c2 = 1$, and $c1 = 0$, $c2 = 1$. Uma vez que existem somente algo em torno de $N/2$ sondagens distintas para um dado elemento, é difícil garantir que inserções terão sucesso quando o fator de carga é maior que 1/2.

Hash duplo

- Um dos melhores métodos para endereçamento aberto, porque as permutações escolhidas têm muitas características das permutações escolhidas aleatoriamente.

- $hash(k,i) = (h_1(k) + i*h_2(k)) \% N$
  - *Onde h_1 e h_2 são funções hash auxiliares.*

Hash duplo

- O valor $h_2(k)$ e o tamanho N da tabela hash devem ser primos entre si para que a tabela hash inteira possa ser pesquisada.
- Uma forma eficiente de assegurar essa condição é permitir que N seja uma potência de 2 e projetar h_2 de modo que ela sempre produza um número ímpar.
- Outra forma é permitir que N seja primo e projetar h_2 de forma que ela sempre retorne um inteiro positivo menor que N. Por exemplo:
  - $h_1(k)=k\%N$
  - $h_2(k)=1+(k\%(N-1))$

Hash duplo

- Por exemplo: $h(k,i) = (h_1(k) + i*h_2(k)) \% N$
  - $h_1(k)=k\%N$
  - $h_2(k)=1+(k\%(N-1))$

Se k=123456 e N=701, temos $h_1(k)=80$ e $h_2(k)=257$
Assim, a primeira sondagem ocorre na posição 80 e depois cada 257-ésima posição (módulo N) é examinada até a chave ser encontrada ou todas as posições serem examinadas.

O desempenho do hash duplo é muito próximo do desempenho de um hash "ideal" (uniforme).