# Tabelas *hash*

Prof. Renato Pimentel

2025/2



## Sumário

1. Tabelas *hash*
   - Definições básicas e aplicações
   - Funções de *hashing*
     - Método da divisão
     - Método da multiplicação
     - Método da dobra
     - Cadeias de caracteres como chaves
   - Um tipo abstrato de dados (TAD) para tabela *hash*
   - Inserção e busca sem tratamento de colisões
   - *Hashing* universal, perfeito e imperfeito
   - Tratamento de colisões
   - Exercícios

# Motivação: busca

- Princípio de funcionamento dos métodos de busca:
  - Procurar a informação desejada com base na comparação de suas **chaves**, i.e. com base em algum valor que a compõe.
- **Problema**: algoritmos eficientes necessitam que os elementos estejam armazenados de forma **ordenada**:
  - Custo ordenação melhor caso é $O(n \log n)$;
  - Custo da busca melhor caso é $O(\log n)$.
    - ★ Árvores binárias balanceadas: $O(\log_2 n)$.



# Busca ideal: $O(1)$

- Custo da comparação de chaves é alto.
- O que seria uma operação de busca ideal?
  - Seria aquela que permitisse o acesso direto ao elemento procurado, sem **nenhuma** etapa de comparação de chaves.
  - Nesse caso, teríamos um custo $O(1)$ – tempo sempre **constante** de acesso.

- Uma saída é usar *arrays*:
  - Estruturas que utilizam **índices** para armazenar informações;
  - Permitem acesso a qualquer posição com custo $O(1)$.

**Problema**

*Arrays* não possuem nenhum mecanismo que permita calcular a posição onde uma informação está armazenada – a operação de busca **não é** $O(1)$.



- Precisamos do tempo de acesso do *array*, em conjunto com a capacidade de busca qualquer elemento em tempo constante.
- **Solução**: usar uma tabela *hash*.

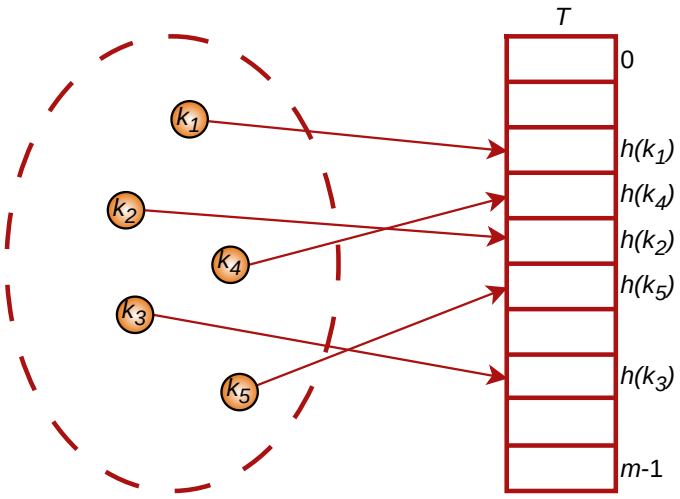
# Tabelas *hash*

- Também conhecidas como **tabelas de indexação** ou **de espalhamento**[1].
  - São generalizações do conceito de *array*.
  - EDs especializadas nas **operações de dicionário** (busca, inserção e remoção) de maneira eficiente.
- Ideia central:
  - Utilizar uma função, chamada de **função de *hashing***, para **espalhar** os elementos que se deseja armazenar na tabela.
  - Tal espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do *array* que define a tabela.

[1] Em inglês: *hashing*



# Exemplo



## Benefícios do espalhamento

- A tabela permite a associar valores a chaves:
  - **Chave**: parte da informação que compõe o elemento a ser inserido ou buscado na tabela.
    - ★ Ex.: Num cadastro de alunos, a **matrícula** pode ser chave – dois alunos não tem a mesma matrícula.
    - ★ As demais informações de cada aluno, como nome, etc., são os **dados satélites**.
  - **Valor**: posição (índice) onde o elemento se encontra no *array* que define a tabela.
- Dadas $n$ chaves, e uma tabela de tamanho $m$ – com $m = \Theta(n)$ – as posições da tabela se situam no intervalo $[0, m-1]$: cada posição pode armazenar um ou mais elementos.
- Logo, a partir de uma chave pode-se acessar de forma rápida – em geral, $O(1)$ – uma determinada posição do *array*.



## Vantagens

- Alta eficiência na operação de busca
  - Caso médio é $O(1)$, enquanto o da busca linear é $O(n)$ e da busca binária é $O(log_2 n)$.
- Tempo de busca é praticamente independente do número de chaves armazenadas na tabela.
- Implementação simples.

# Desvantagens

- Alto custo para recuperar os elementos da tabela ordenados pela chave.
  - Nesse caso, é preciso ordenar a tabela;
  - O pior caso é $O(m)$, sendo $m$ o tamanho da tabela.
- **Colisões**.

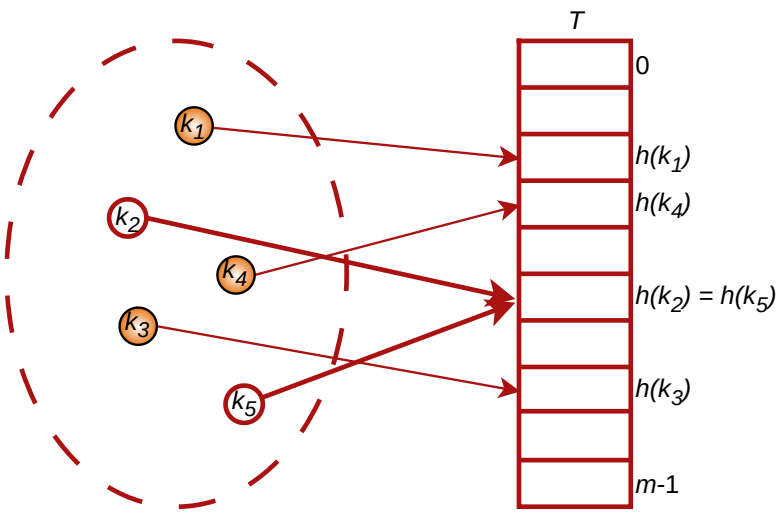
**Colisão**

Quando duas (ou mais) chaves diferentes tentam ocupar a mesma posição na tabela *hash*.

- Inevitável, mas indesejável, pois diminui o desempenho do sistema.



# Exemplo de colisão



## Principais aplicações

- Busca de elementos em bases de dados. Ex.: estruturas de dados em memória, bancos de dados e mecanismos de busca na Internet;
- verificação de integridade de dados e autenticação de mensagens;
  - os dados são enviados juntamente com o resultado da função de *hashing*;
  - O receptor dos dados recalcula a função de *hashing* usando os dados recebidos, e compara o resultado obtido com o que ele recebeu;
  - Resultados diferentes: **erro** de transmissão.
- Tabelas de símbolos em compiladores;
- Compressão de dados – para evitar armazenamentos duplicados;
- NAT (*Network Address Translation*) em redes de computadores;
- Criptografia – armazenamento de senhas com segurança.
  - MD5 (*Message-Digest Algorithm* 5);
  - SHA-1 (*Secure Hash Algorithm*).



## Sumário

1. Tabelas *hash*
   - Definições básicas e aplicações
   - Funções de *hashing*
     - Método da divisão
     - Método da multiplicação
     - Método da dobra
     - Cadeias de caracteres como chaves
   - Um tipo abstrato de dados (TAD) para tabela *hash*
   - Inserção e busca sem tratamento de colisões
   - *Hashing* universal, perfeito e imperfeito
   - Tratamento de colisões
   - Exercícios

# Funções de *hashing*

- Operações de dicionário: necessário calcular a **posição** (valor) dos dados dentro da tabela.
- A **função de *hashing*** $h$ calcula a posição $h(k)$, $0 \leq h(k) < m$, para uma determinada chave $k$ a partir dos dados.
  - Extremamente importante para o bom desempenho da tabela.
  - Responsável por distribuir as informações de forma equilibrada na ED – **espalhamento**.

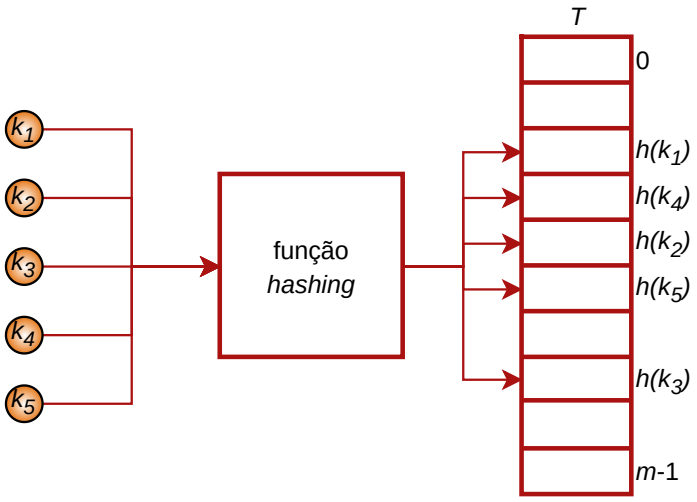
# Requisitos

- Idealmente fácil de computar e **uniforme** (distribuição equilibrada dos dados).
  - Cada posição da tabela tem a mesma chance de receber uma chave (máximo espalhamento).
- O ideal é que a função forneça índices únicos para o conjunto das chaves de entrada possíveis;
- Eventualmente, duas chaves diferentes podem ser mapeadas em valores iguais (colisão): $x \neq y$ tal que $h(x) = h(y)$.

- A implementação depende do conhecimento prévio da natureza e domínio da chave a ser utilizada.
- **Exemplo**: utilizar apenas três dígitos do número de telefone de uma pessoa para armazená-lo na tabela.
  - Neste caso, seria melhor usar os três últimos dígitos ao invés dos três primeiros, pois estes costumam se repetir com maior frequência, gerando posições iguais na tabela.
- Assim, o ideal é usar um cálculo diferente de espalhamento para cada tipo de chave.



# Esquema de funcionamento



## Funções de *hashing* comumente utilizadas

- Método da divisão;
- Método da multiplicação;
- Método da dobra.



## Método da divisão

- Também conhecido como **método da congruência linear**.
- Consiste em calcular o **resto da divisão** do valor inteiro que representa o elemento pelo tamanho da tabela, $m$:

$$h(k) = k \bmod m\,.$$

- Simples e direta;
- **Implementação em C**: A **operação de E bit-a-bit** (&) com o valor 0x7FFFFFFF elimina o **bit de sinal** e evita o risco de ocorrer um *overflow* e consequente obtenção de um número negativo.

TabelaHash.c: método da divisão

```
52 int chaveDivisao(int chave, int TABLE_SIZE) {
53   return (chave & 0x7FFFFFFF) % TABLE_SIZE;
54 }
```

## Limitações

- **Resto da divisão**: valores diferentes podem resultar na mesma posição.
  - Ex.: O resto da divisão de 11 por 10 e de 21 por 10 são o mesmo valor de posição: 1
- Uma maneira de reduzir esse tipo de problema é utilizar como tamanho da tabela, $m$, um **número primo**.



## Método da multiplicação

- Também conhecido como **método da congruência linear multiplicativo**.
- Usa uma **constante fracionária** $A$, $0 < A < 1$, para multiplicar o valor da chave que representa o elemento;
- Em seguida, a **parte fracionária** resultante é multiplicada pelo tamanho da tabela para calcular a posição do elemento:

$$h(k) = \lfloor m \cdot [(k \cdot A) \bmod 1] \rfloor$$

TabelaHash.c: método da multiplicação

```
71 int chaveMultiplicacao(int chave, int TABLE_SIZE) {
72   double A = 0.6180339887; // constante: 0 < A < 1
73   double val = chave * A;
74   val = val - (int) val;
75   return (int) (TABLE_SIZE * val);
76 }
```

## Aspectos

- Escolha de $m$ não é mais crítica;
- Porém, escolha de $A$ depende dos dados.
  - Knuth sugere o negativo do **conjugado da razão áurea**:

$$A = -\frac{1-\sqrt{5}}{2} \approx 0,6180339887$$



## Exemplo – com tabela de tamanho $m = 1024$

- Chave $k = 2112$ e $A = 0,8660254038$:
- Multiplicação: $k \cdot A = 1829,0456528256$;
- Parte fracionária: $(k \cdot A) \bmod 1 = 0,0456528256$;
- Produto da parte fracionária por $m$:
  $m \cdot [(k \cdot A) \bmod 1] = 1024 \cdot 0,0456528256 = 46,7484934144$;
- A parte inteira (piso) é a **posição** na tabela: $h(k) = \lfloor 46,74\ldots \rfloor = 46$.

- Repetindo-se o exemplo anterior para $A = 0,6180339887$ (Knuth): $h(k) = 294$.



## Método da dobra

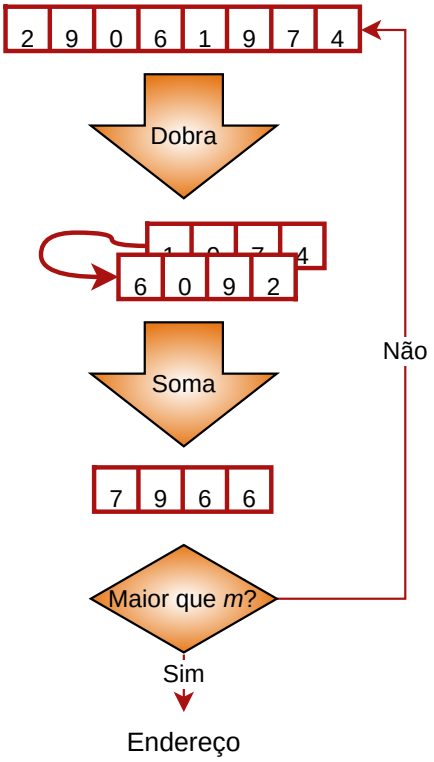
- Utiliza um esquema de **dobrar e somar** os dígitos da chave para calcular a sua posição.
- Considera o valor inteiro que representa o elemento como uma sequência de dígitos escritos num pedaço de papel.
  - Enquanto esse valor for maior que o tamanho da tabela, o papel é dobrado e os dígitos sobrepostos são somados, **desconsiderando-se as dezenas**.
- O processo se repete até que os dígitos formem um número menor que $m$, o tamanho da tabela.

# Exemplo – com tabela de tamanho $m = 64$

$k = 29061974$

1974
<u>6092</u>
7966

66
<u>97</u>
**53**






# Implementação com operações bit-a-bit

- O método da dobra pode ser usado com **operações bit-a-bit**:
  - Utiliza a operação de **OU exclusivo** (XOR);
  - Não se usa as operações de **E** e **OU** bit-a-bit, pois estas produzem resultados menores e maiores, respectivamente, que os operandos envolvidos.

TabelaHash.c: método da dobra

```
int chaveDobra(int chave, int TABLE_SIZE) {
  int num_bits = 10;
  int parte1 = chave >> num_bits;
  int parte2 = chave & (TABLE_SIZE-1);
  return (parte1 ^ parte2);
}
```

- A dobra neste caso pode ser realizada de $b$ em $b$ bits, o que resulta em $f(k) \in [0, 2^b)$.
- Ex.: queremos calcular a posição do valor 71 (0001000111 em binário), usando $b = 5$:

$$\begin{aligned} \text{Posição} &= 00010 \text{ “OU exclusivo” } 00111 \\ &= 00101 \\ &= \mathbf{5}\,. \end{aligned}$$

## Cadeias de caracteres como chaves

- Quando a chave é **alfanumérica**, i.e. uma *string*, pode-se optar por calcular um valor numérico a partir de seu conteúdo:
  - Esse valor pode ser facilmente calculado, por exemplo, somando-se os valores ASCII dos caracteres que compõem a *string* – porém, isto não é bom.
- O resultado pode então ser usado como parâmetro numa função de *hashing*.

TabelaHash.c: String como chave

```
41 int valorString(char *str) {
42   int i, valor = 7;
43   int tam = strlen(str);
44   for (i=0; i < tam; i++)
45     valor = 31 * valor + (int) str[i];
46   return (valor & 0x7FFFFFF);
47 }
```

- Por que não devemos simplesmente somar os valores ASCII dos caracteres da *string*?
  - Porque palavras que correspondem a permutações uma da outra produzirão o mesmo valor e, consequentemente, uma **colisão**.
  - Exemplo: **rato** e **ator**.

$$\begin{aligned} \text{rato:}\ & 114 + 97 + 116 + 111 = 438 \\ \text{ator:}\ & 97 + 116 + 111 + 114 = 438\,. \end{aligned}$$

- A posição da letra **deve** ser considerada.
- No trecho de código, cada novo caractere é somado ao produto da soma dos caracteres anteriores pelo número 31 (número primo, assim reduzindo a probabilidade de se produzir valores iguais).



# Sumário

1. Tabelas *hash*
   - Definições básicas e aplicações
   - Funções de *hashing*
     - Método da divisão
     - Método da multiplicação
     - Método da dobra
     - Cadeias de caracteres como chaves
   - Um tipo abstrato de dados (TAD) para tabela *hash*
   - Inserção e busca sem tratamento de colisões
   - *Hashing* universal, perfeito e imperfeito
   - Tratamento de colisões
   - Exercícios

# Um tipo abstrato de dados (TAD) para tabela *hash*

- Consideramos um tipo abstrato de dados **opaco** em Linguagem C

**Relembrando: Tipo opaco**

- A ED será referenciada mediante um **ponteiro** para a mesma, ao invés de uma variável do tipo "tabela *hash*" diretamente.
- Isto permite ocultar a implementação, permitindo sua alteração sem interferir no funcionamento de qualquer código que o utilize – desde que a **interface** seja respeitada.



- Modularização do TAD:
  - **TabelaHash.h**: a **interface** em si: definição do tipo opaco; funções disponíveis para as operações na tabela (protótipos).
  - **TabelaHash.c**: o que deve ficar oculto do usuário, além da implementação das funções definidas pela interface. Inclui a definição do **tipo *hash***.
- **Tipo *hash***: estrutura que contém:
  - `TABLE_SIZE`: tamanho da tabela ($m$, a quantidade de entradas disponíveis) ;
  - `qtd`: quantidade de entradas ocupadas da tabela;
  - `**itens`: a ED em si, i.e. um *array* de endereços dos dados.

# Observação

- Por questões de desempenho e de bom uso da memória, a tabela irá armazenar apenas o endereço para a estrutura que contém os dados, e não os dados em si.
- A alocação é feita à medida que os elementos são inseridos.



# Interface: TabelaHash.h

TabelaHash.h

```
struct aluno{
  int matricula;
  char nome[30];
  float n1,n2,n3;
};

typedef struct hash Hash;

Hash* criaHash(int TABLE_SIZE);
void liberaHash(Hash* ha);
int valorString(char *str);
int insereHash_SemColisao(Hash* ha, struct aluno al);
int buscaHash_SemColisao(Hash* ha, int mat, struct
    aluno* al);
int insereHash_EnderAberto(Hash* ha, struct aluno al);
int buscaHash_EnderAberto(Hash* ha, int mat, struct
    aluno* al);
```

# Definição da tabela: TabelaHash.c

TabelaHash.c: bibliotecas usadas e definição do tipo

```c
#include <stdlib.h>
#include <string.h>
#include "TabelaHash.h" //inclui os protótipos

//Definição do tipo Hash
struct hash {
  int qtd, TABLE_SIZE;
  struct aluno **itens;
};
```



# Criando uma tabela *hash* I

main.c – Exemplo de uso

```c
Hash *tabela = criaHash(tamanho);
```

TabelaHash.c: criação da tabela hash

```c
Hash* criaHash(int TABLE_SIZE) {
  Hash* ha = (Hash*) malloc(sizeof(Hash));
  if (ha != NULL) {
    int i;
    ha->TABLE_SIZE = TABLE_SIZE;
    ha->itens = (struct aluno**) malloc(TABLE_SIZE *
   sizeof(struct aluno*));
    if (ha->itens == NULL) {
      free(ha);
      return NULL;
    }
    ha->qtd = 0;
```

# Criando uma tabela *hash* II

```
22      for (i=0; i < ha->TABLE_SIZE; i++)
23        ha->itens[i] = NULL;
24    }
25    return ha;
26  }
```



# Liberando a tabela *hash* da memória I

main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
liberaHash(tabela);
```

TabelaHash.c: liberando a tabela

```
28 void liberaHash(Hash* ha) {
29   if (ha != NULL) {
30     int i;
31     for (i=0; i < ha->TABLE_SIZE; i++) {
32         if (ha->itens[i] != NULL)
33             free(ha->itens[i]);
34     }
35     free(ha->itens);
36     free(ha);
```

# Liberando a tabela *hash* da memória II

```
37     }
38 }
```

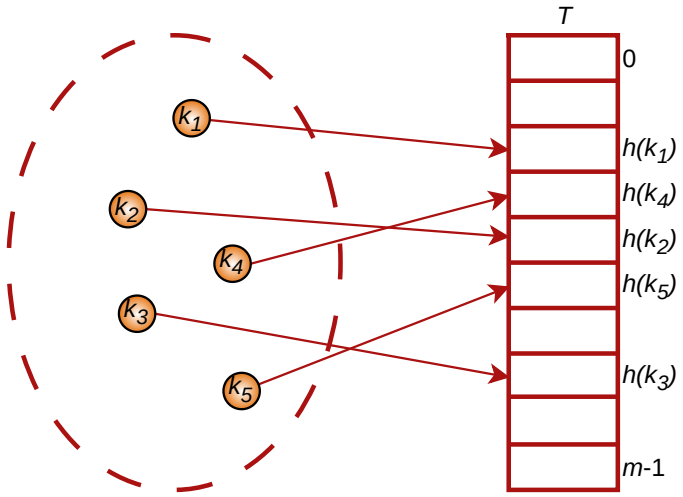


# Sumário

1 Tabelas *hash*
- Definições básicas e aplicações
- Funções de *hashing*
  - Método da divisão
  - Método da multiplicação
  - Método da dobra
  - Cadeias de caracteres como chaves
- Um tipo abstrato de dados (TAD) para tabela *hash*
- Inserção e busca sem tratamento de colisões
- *Hashing* universal, perfeito e imperfeito
- Tratamento de colisões
- Exercícios

# Inserção

- Calcular a posição da chave (valor) no *array*.
- Alocar espaço para os dados.
- Armazenar os dados na posição calculada.






main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos",
    8.8, 7.4, 7.7};
insereHash_SemColisao(tabela,alu);
```

TabelaHash.c: inserção na tabela hash

```
81 int insereHash_SemColisao(Hash* ha, struct aluno al) {
82   if (ha == NULL || ha->qtd == ha->TABLE_SIZE)
83     return 0;
84
85   int chave = al.matricula;
86   //int chave = valorString(al.nome);
87
88   int pos = chaveDivisao(chave,ha->TABLE_SIZE);
89   struct aluno* novo;
```

```
90    novo = (struct aluno*) malloc(sizeof(struct aluno));
91    if (novo == NULL)
92      return 0;
93    *novo = al;
94    ha->itens[pos] = novo;
95    ha->qtd++;
96    return 1;
97  }
```



# Busca

- Calcular a posição da chave (valor) no *array*.
- Verificar se há dados na posição calculada.
- Retornar os dados.

## main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos",
    8.8, 7.4, 7.7};
insereHash_SemColisao(tabela,alu);
.
.
buscaHash_SemColisao(tabela, 1000, &al);
printf("%s, %d\n",al.nome,al.matricula);
```



## TabelaHash.c: busca na tabela hash

```
int buscaHash_SemColisao(Hash* ha, int mat, struct
    aluno* al) {
  if (ha == NULL)
    return 0;

  int pos = chaveDivisao(mat,ha->TABLE_SIZE);
  if (ha->itens[pos] == NULL)
    return 0;
  *al = *(ha->itens[pos]);
  return 1;
}
```

# Sumário


1. Tabelas *hash*
   - Definições básicas e aplicações
   - Funções de *hashing*
     - Método da divisão
     - Método da multiplicação
     - Método da dobra
     - Cadeias de caracteres como chaves
   - Um tipo abstrato de dados (TAD) para tabela *hash*
   - Inserção e busca sem tratamento de colisões
   - *Hashing* universal, perfeito e imperfeito
   - Tratamento de colisões
   - Exercícios




# *Hashing* universal

- Função de *hashing* está sujeita ao problema de se gerar **colisões** – posições iguais para chaves diferentes;
- Por se tratar de uma função determinística, pode ser manipulada de forma indesejada.
  - Conhecendo-se a função de *hashing*, pode-se escolher as chaves de entrada de modo que todas colidam, diminuindo o desempenho da tabela na busca para $O(n)$.

- O ***hashing* universal** é uma estratégia que busca minimizar o problema de colisões.
- Escolhe-se aleatoriamente, **em tempo de execução**, a função de *hashing* que será utilizada.
- Para isso, constrói-se um conjunto (ou família) de funções de *hashing*.



## Família de funções

- Existem diversas formas de se construir uma família de funções de *hashing*.
- Ex.: Uma família de funções pode ser facilmente obtida da seguinte forma:
  1. Escolhe-se um número primo $p$ que seja maior do que qualquer chave $k$ a ser inserida.
  2. $p$ também deve ser maior do que o tamanho da tabela, $m$.
  3. Então, escolhe-se, aleatoriamente, dois números inteiros, $a$ e $b$, tais que $0 < a < p$ e $0 \leq b < p$.
  4. Dados $p$, $a$ e $b$, define-se a **função de *hashing* universal** como sendo:

$$h_{ab}(k) = [(a \cdot k + b) \bmod p] \bmod m$$

- A função de *hashing* universal dada permite o tamanho da tabela, $m$, possa não ser necessariamente primo.
- Ainda, como existem ao todo $p - 1$ valores possíveis para $a$ e $p$ para $b$, é possível gerar $p(p - 1)$ funções de *hashing* diferentes.



## *Hashing* imperfeito $\times$ *hashing* perfeito

- Dependendo do tamanho da tabela, $m$, e dos valores inseridos, uma função de *hashing* pode ser definida como:
  - *Hashing* imperfeito;
  - *Hashing* perfeito.

## *Hashing* imperfeito

- Para duas chaves diferentes $k_1$ e $k_2$, a saída da função de *hashing* é a mesma, i.e. $h(k_1) = h(k_2)$: **colisão**.
- Como já explicado, algo indesejável, pois diminui o desempenho.
- De modo geral, muitas tabelas *hash* fazem uso de alguma outra ED para lidar com tal problema, como será visto adiante.



## *Hashing* perfeito

- **Nunca** ocorre colisão:
  - Chaves diferentes sempre produzem posições diferentes.
- No pior caso, as operações são sempre executadas em tempo constante, $O(1)$.
- É utilizado onde a colisão não é tolerável.
- Trata-se de um tipo de aplicação muito específica, por exemplo, o conjunto de palavras reservadas de uma linguagem de programação. Nesse caso, se conhece previamente o conteúdo a ser armazenado.

# Sumário


1. Tabelas *hash*
   - Definições básicas e aplicações
   - Funções de *hashing*
     - Método da divisão
     - Método da multiplicação
     - Método da dobra
     - Cadeias de caracteres como chaves
   - Um tipo abstrato de dados (TAD) para tabela *hash*
   - Inserção e busca sem tratamento de colisões
   - *Hashing* universal, perfeito e imperfeito
   - Tratamento de colisões
   - Exercícios




# Mundo ideal × mundo real

- No mundo ideal:
  - *Hashing* perfeito – a função de *hashing* irá sempre fornecer posições diferentes para cada uma das chaves inseridas.
- No mundo real, contudo:
  - Independente da função de *hashing* utilizada, a mesma fatalmente poderá retornar a mesma posição para duas chaves diferentes: **colisão**!

## Técnicas para tratamento de colisões

- Infelizmente, não se pode garantir que as funções de *hashing* possuam um bom potencial de espalhamento por que as colisões – teoricamente inevitáveis – também são uniformemente distribuídas.
- Por isso, é preciso ter uma abordagem para tratá-las.
- Duas técnicas muito comuns:
  - **Endereçamento aberto**;
  - **Encadeamento separado**.

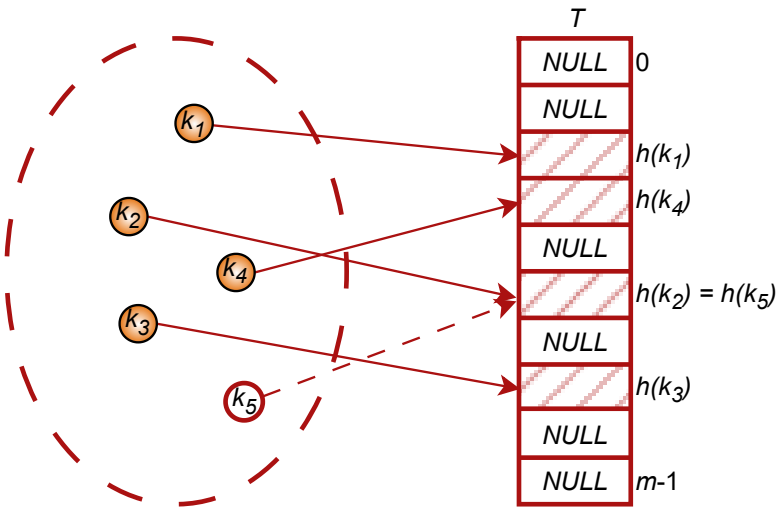


## Endereçamento aberto

- Em inglês, *open addressing*. Também conhecido por **rehash**.
- Em caso de colisão, percorre-se a tabela *hash* buscando por uma posição ainda não ocupada.
- Os elementos são armazenados na própria tabela.
  - Desta forma, evita-se o uso de listas encadeadas.

# Exemplo

- $k_5$ é tal que $h(k_5) = h(k_2)$, com $k_2$ já inserido na tabela:
  - Percorre-se a tabela até encontrar uma posição vaga (NULL).






# Vantagens

- Maior número de posições na tabela para a mesma quantidade de memória usada no **encadeamento separado**;
  - A memória utilizada para armazenar os ponteiros da lista encadeada no encadeamento separado pode ser aqui usada para aumentar o tamanho da tabela, diminuindo o número de colisões.
- Busca é realizada dentro da própria tabela – recuperação mais rápida de elementos;
- Voltada para aplicações com restrições de memória;
- Ao invés de acessar ponteiros extras, calcula-se a sequência de posições a serem armazenadas.

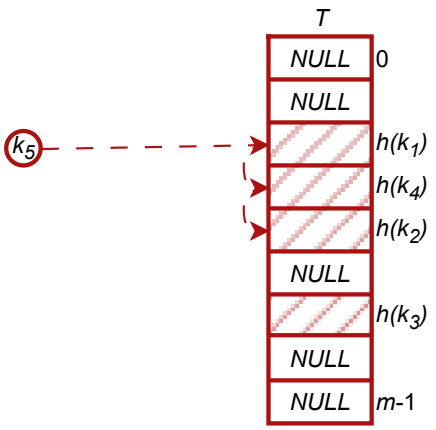
# Desvantagens

- Custo maior no processamento no cálculo das posições
  - Se deve ao fato de que, quando uma colisão ocorre, deve-se calcular uma nova posição da tabela.
- Possibilidade de colisões sucessivas.



# Colisões sucessivas: exemplo

- Ao simplesmente se percorrer a tabela, haverá colisões sucessivas.



# Estratégias de endereçamento aberto

- Para a realização do cálculo da nova posição após a colisão, existem três estratégias muito utilizadas:
  - Sondagem linear;
  - Sondagem quadrática;
  - Duplo *hash*.



# Sondagem linear

- Também conhecida como **tentativa linear**, **espalhamento linear** ou ***rehash* linear**.
- Tenta espalhar os elementos de forma sequencial, a partir da posição calculada utilizando a função de **hashing**:
  - Se já existe um elemento na posição, tenta-se o próximo, até que se encontre uma posição vazia, de forma circular.

- Funcionamento:
  - Primeiro elemento ($i = 0$) é colocado na posição obtida pela função de *hashing*: pos;
  - Segundo elemento (colisão) é colocado na posição pos + 1 – caso a mesma esteja livre;
  - Terceiro elemento (nova colisão) é colocado na posição pos + 2 – caso a mesma esteja livre, etc.;

TabelaHash.c: sondagem linear

```c
int sondagemLinear(int pos, int i, int TABLE_SIZE) {
  return ((pos + i) & 0x7FFFFFFF) % TABLE_SIZE;
}
```

## Exemplo

| Chave | $h(k_i)$ | Inserção |
|---|---|---|
| $k_1$ | 2 | Posição 2 vazia: insere o elemento. |
| $k_2$ | 6 | Posição 6 vazia: insere o elemento. |
| $k_3$ | 2 | Posição 2 ocupada, procura na próxima posição: 3. Posição 3 vazia. Insere o elemento. |
| $k_4$ | 9 | Posição 9 vazia: insere o elemento. |
| $k_5$ | 9 | Posição 9 ocupada, procura na próxima posição. Como 9 é a última, volta para o início: 0. Posição 0 vazia. Insere elemento. |

| | |
|---|---|
| $k_5$ | 0 |
| NULL | 1 |
| $k_1$ | 2 |
| $k_3$ | 3 |
| NULL | 4 |
| NULL | 5 |
| $k_2$ | 6 |
| NULL | 7 |
| NULL | 8 |
| $k_4$ | 9 |

## Desvantagem: agrupamento primário

- À medida que a tabela *hash* fica mais cheia, o tempo para incluir ou buscar um elemento aumenta;
- À medida que os elementos são inseridos, surgem longas sequências de posições ocupadas;
- A ocorrência desses agrupamentos aumenta o tempo de pesquisa, diminuindo o desempenho.



## Sondagem quadrática

- Também conhecida como **tentativa quadrática**, **espalhamento quadrático** ou ***rehash* quadrático**;
- Tenta espalhar os elementos com base numa equação do segundo grau.
- Exemplo: $pos + (c_1 \cdot i) + (c_2 \cdot i^2)$.
  - $pos$ é a posição obtida pela função de *hashing*;
  - $i$ é tentativa atual;
  - $c_1$ e $c_2$ são os coeficientes da equação.

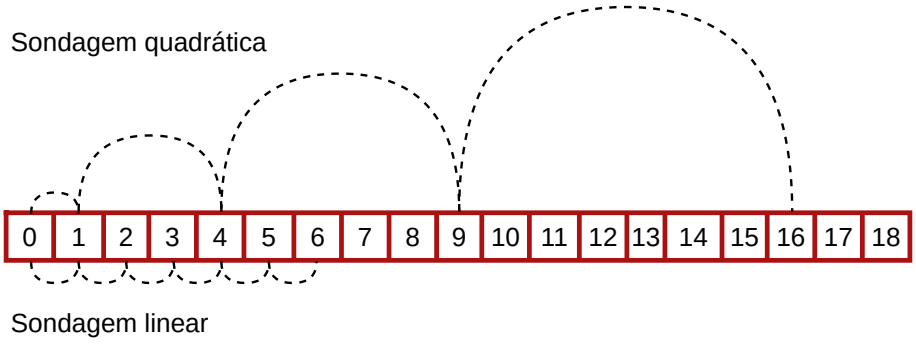
- Funcionamento:
  - Primeiro elemento ($i = 0$) é colocado na posição obtida pela função de *hashing*: pos;
  - Segundo elemento (colisão) é colocado na posição pos + (c1*1) + (c2*1) – caso a mesma esteja livre;
  - Terceiro elemento (nova colisão) é colocado na posição pos + (c1*2) + (c2*2*2) – caso a mesma esteja livre, etc.;

TabelaHash.c: sondagem quadrática

```
int sondagemQuadratica(int pos, int i, int TABLE_SIZE) {
  pos = pos + 2*i + 5*i*i; // hash + (c1 * i) + (c2 * i ^2)
  return (pos & 0x7FFFFFFF) % TABLE_SIZE;
}
```

# Comparativo





## Aspectos

- Resolve o problema de agrupamento primário;
- Porém, gera outro problema, conhecido como **agrupamento secundário**:
  - Todas as chaves que produzirem a mesma posição inicial também produzirão as mesmas posições na sondagem quadrática.
- Felizmente, a degradação produzida pelos agrupamentos secundários é menor que a produzida pelos agrupamentos primários.



## Duplo *hash*

- Também conhecida como **espalhamento duplo**;
- Tenta espalhar os elementos utilizando **duas** funções de *hashing*, necessariamente diferentes entre si:
  - a primeira função de *hashing*, $h_1$, é utilizada para calcular a **posição inicial** do elemento;
  - a segunda função de *hashing*, $h_2$, é utilizada para calcular os deslocamentos em relação à posição inicial, em caso de colisão.
- A posição de um novo elemento na tabela *hash* é obtida como sendo

$$h_1 + i \cdot h_2,$$

  onde $i$ é tentativa atual de inserção do elemento.
- A segunda função de *hashing* não pode resultar em **zero**, pois neste caso não haveria deslocamento.

- Funcionamento:
  - Primeiro elemento ($i = 0$) é colocado na posição obtida pela função de *hashing* $h_1$;
  - Segundo elemento (colisão) é colocado na posição dada por $h_1 + h_2$ – caso a mesma esteja livre;
  - Terceiro elemento (nova colisão) é colocado na posição $h_1 + 2h_2$ – caso a mesma esteja livre, etc.;

TabelaHash.c: duplo hash

```
int duploHash(int H1, int chave, int i, int TABLE_SIZE)
    {
  int H2 = chaveDivisao(chave,TABLE_SIZE-1) + 1;
  return ((H1 + i*H2) & 0x7FFFFFFF) % TABLE_SIZE;
}
```

## Inserção com tratamento de colisões

- Calcular a posição da chave (valor) no *array*.
- Recalcular a posição enquanto houver colisão (limitar o número de tentativas).
- Alocar espaço para os dados.
- Armazenar os dados na posição calculada.

main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos",
    8.8, 7.4, 7.7};
insereHash_EnderAberto(tabela,alu);
```

TabelaHash.c: inserção com tratamento de colisão

```
132 int insereHash_EnderAberto(Hash* ha, struct aluno al) {
133   if (ha == NULL || ha->qtd == ha->TABLE_SIZE)
134     return 0;
135
136   int chave = al.matricula;
137   //int chave = valorString(al.nome);
138
139   int i, pos, newPos;
140   pos = chaveDivisao(chave,ha->TABLE_SIZE);
```



```
141   for (i=0; i < ha->TABLE_SIZE; i++) {
142     newPos = sondagemLinear(pos,i,ha->TABLE_SIZE);
143     //newPos = sondagemQuadratica(pos,i,ha->TABLE_SIZE)
    ;
144     //newPos = duploHash(pos,chave,i,ha->TABLE_SIZE);
145     if (ha->itens[newPos] == NULL) {
146       struct aluno* novo;
147       novo = (struct aluno*) malloc(sizeof(struct aluno
    ));
148       if (novo == NULL)
149           return 0;
150       *novo = al;
151       ha->itens[newPos] = novo;
152       ha->qtd++;
153       return 1;
154     }
155   }
156   return 0;
157 }
```

# Busca com tratamento de colisões

- Calcular a posição da chave (valor) no *array*.
- Verificar se há dados na posição calculada, e se tais dados correspondem à chave.
- Recalcular a posição enquanto os dados forem diferentes da chave.
- Retornar os dados.



main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos",
    8.8, 7.4, 7.7};
insereHash_EnderAberto(tabela,alu);
.
.
int x = buscaHash_EnderAberto(tabela, 1000, &al);
```

TabelaHash.c: busca com tratamento de colisão

```
159 int buscaHash_EnderAberto(Hash* ha, int mat, struct
        aluno* al) {
160   if (ha == NULL)
161     return 0;
162
163   int i, pos, newPos;
```

```
  pos = chaveDivisao(mat,ha->TABLE_SIZE);
  for (i=0; i < ha->TABLE_SIZE; i++) {
    newPos = sondagemLinear(pos,i,ha->TABLE_SIZE);
    //newPos = sondagemQuadratica(pos,i,ha->TABLE_SIZE);
    //newPos = duploHash(pos,mat,i,ha->TABLE_SIZE);
    if (ha->itens[newPos] == NULL)
      return 0;

    if (ha->itens[newPos]->matricula == mat) {
      *al = *(ha->itens[newPos]);
      return 1;
    }
  }
  return 0;
}
```

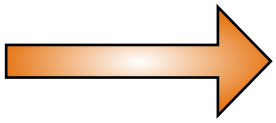


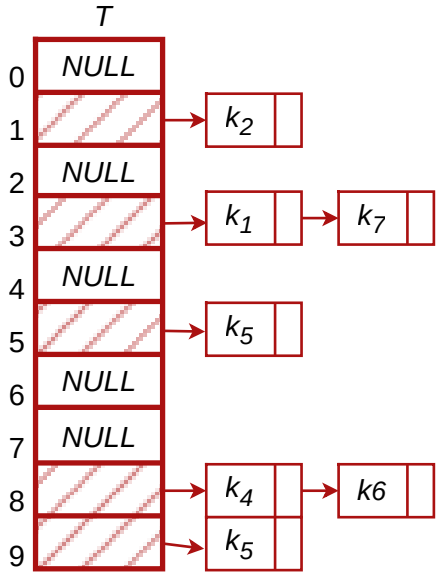
# Encadeamento separado

- Em inglês, *separate chaining*.
- Não procura por posições vagas (valor `NULL`) dentro do *array* que define a tabela.
- Armazena em cada posição do *array* o início de uma **lista dinâmica encadeada**.
  - É dentro dessa lista que serão armazenadas as colisões (elementos com chaves iguais) para aquela posição do *array*.

# Exemplo

| chave | posição |
|---|---|
| $k_1$ | 3 |
| $k_2$ | 1 |
| $k_3$ | 9 |
| $k_4$ | 8 |
| $k_5$ | 5 |
| $k_6$ | 8 |
| $k_7$ | 3 |








# Características

- A lista dinâmica encadeada mantida em cada posição da tabela pode ser ordenada ou não. Lista não ordenada
  - Inserção tem complexidade $O(1)$ no pior caso: basta inserir o elemento no **início da lista**.
  - Busca tem complexidade $O(m)$ no pior caso: **busca linear**.
- **Desvantagem**: quantidade de memória consumida – gasta-se mais memória para manter os ponteiros que ligam os diferentes elementos dentro de cada lista.

# Sumário


1. Tabelas *hash*
   - Definições básicas e aplicações
   - Funções de *hashing*
     - Método da divisão
     - Método da multiplicação
     - Método da dobra
     - Cadeias de caracteres como chaves
   - Um tipo abstrato de dados (TAD) para tabela *hash*
   - Inserção e busca sem tratamento de colisões
   - *Hashing* universal, perfeito e imperfeito
   - Tratamento de colisões
   - Exercícios




# Exercícios I

1. Descreva de que forma podemos detectar que todas as posições possíveis vagas da tabela *hash* foram acessadas durante o reespalhamento.
2. Considere uma tabela de *hash* de tamanho $m = 1000$ e a função de *hashing* usando o método da multiplicação com $A = (\sqrt{5} - 1)/2$. Calcule os valores de *hash* das chaves 61, 62, 63, 64 e 65.
3. Suponha um conjunto de $n$ chaves, formado pelos $n$ primeiros múltiplos do número 7. Quantas colisões seriam obtidas mediante a aplicação das funções de *hashing* seguintes?
   1. $h(k) = k \bmod 7$
   2. $h(k) = k \bmod 14$
   3. $h(k) = k \bmod 5$

# Exercícios II

4. Suponha uma tabela *hash* de tamanho $m = 10$ com endereçamento aberto para armazenar chaves no intervalo $[1, 999]$. Insira as seguintes chaves na tabela: 371, 173, 203, 11, 24, nesta ordem, considerando diferentes métodos de resolução de colisões:
   1. Sondagem linear, função de *hashing* $h(k) = (k + i) \bmod m$.
   2. Sondagem quadrática, função de *hashing* $h(k) = (k + i^2) \bmod m$.
   3. Sondagem quadrática, função de *hashing* $h(k) = (k + 2i + i^2) \bmod m$.
   4. *Hash* duplo, funções de *hashing*: $h_1(k) = k \bmod m$ e $h_2(k) = 7 - (k \bmod 7)$.
5. (CORMEN, 2024) Considere uma tabela de espalhamento com 9 posições inicialmente vazia e a função de *hash* $h(k) = k \bmod 9$. Demonstre o que acontece quando inserimos as chaves 5, 28, 19, 15, 20, 33, 12, 17, 10 com as colisões resolvidas por encadeamento.



# Referências


1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.
2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.


Os materiais de parte desta seção foram gentilmente cedidos por Ana Cláudia Martinez (FACOM/UFU) e André R. Backes (UFSCar).
TAD tabela *hash*: disponível em: http://www.facom.ufu.br/~backes/wordpress/ProjTabelaHash.zip. Acesso em: 31/mar/2025.
Adaptações: Renato Pimentel, FACOM/UFU