

# Tabela Hash

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados I (AE22CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Tabela *Hash*
- Funções *Hash*
- Implementações

- Vários métodos de busca realizam comparação de chaves durante o processamento
- Caso os dados estejam ordenados, a busca pode ter um custo de:
  - $O(n)$ : busca sequencial
  - $O(\log n)$ : busca binária
  - $O(\log(\log n))$ : busca por interpolação (se os dados estiverem uniformemente distribuídos)
  - $O(\max(m, n/m))$ : busca sequencial indexada
  - $O(\max(\log m, \log(n/m)))$ : busca binária indexada (busca sequencial indexada + busca binária)

- É possível encontrar uma chave sem a necessidade de compará-la com outras ou com custo constante ( $O(1)$ )?

- Sabe-se que em arranjos é possível acessar os dados de forma direta através de um índice, ou seja, com custo de  $O(1)$
- No entanto, como não é possível saber em qual índice a chave se encontra, geralmente teríamos que fazer uma busca, cuja abordagem depende de vários fatores:
  - Tamanho do arranjo
  - Distribuição dos dados
  - Ordenação
  - Etc

## Tabela *Hash*

- As tabelas *hash* (tabelas de espalhamento, tabelas de dispersão) são uma solução para este problema
- Uma tabela *hash* associa chaves e valores:
  - Chave: uma parte da informação que compõe o item a ser inserido ou buscado
  - Valor: posição onde o item deve estar no vetor

- *Hashing*: processamento de chave, cujo resultado é uma posição no arranjo
- A tabela *hash* usa uma função ***h***, onde:
  - A entrada é uma chave
  - A saída é a posição (endereço) onde essa chave deve ser inserida
- Com essa função, os dados podem não ser inseridos de forma ordenada
  - Por isso, o processo de aplicação de *hashing* é conhecido "como espalhamento"

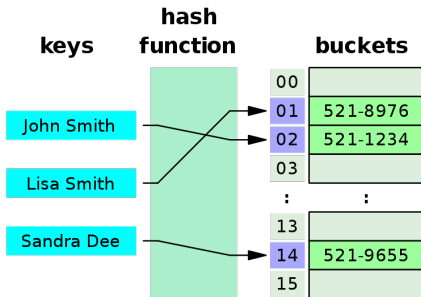


# Tabela *Hash*

- Ideia da tabela *hash*: particionar um conjunto de elementos (possivelmente infinito) em um número finito de classes:
  - $B$  classes (endereços), de 0 a  $B - 1$
  - Essas classes são chamadas de **buckets**

# Tabela Hash

- Conceitos relacionados:
  - A função  $h$  é chamada de função *hash*
  - $h(k)$  retorna o valor *hash* de  $k$
  - $k$  pertence ao *bucket*  $h(k)$



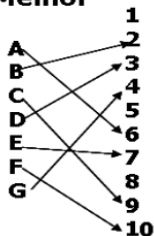
Fonte da figura:

<https://commons.wikimedia.org/w/index.php?curid=6471238>

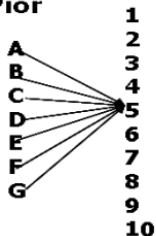
# Tabela Hash

- Colisão: ocorre quando a função *hash* produz o mesmo endereço para chaves diferentes

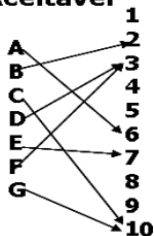
**Melhor**



**Pior**



**Aceitável**



- Distribuição uniforme é muito difícil
- Existe chance de alguns endereços serem gerados mais de uma vez e de outros nunca serem gerados
- Segredos para um bom *hashing*:
  - Escolher uma boa função *hash* (em função dos dados)
    - Distribui uniformemente os dados, na medida do possível
    - Evita colisões
    - Fácil implementação
    - Rápida ao ser computada
  - Estabelecer uma boa estratégia para tratamento de colisões

- Exemplo de função *hash* simples e muito utilizada que produz bons resultados:
  - Para chaves inteiras, calcular o resto da divisão  $k\%B$ , sendo que o resto indica a posição de armazenamento
  - Para chaves do tipo *string*, tratar cada caractere como um valor inteiro (ASCII), somá-los e obter o resto da divisão por  $B$
  - $B$  deve ser primo, preferencialmente

## Funções *Hash*

- A função *hashing* tem o objetivo de mapear o endereço (posição) de uma chave
- Existem diferentes possibilidades para implementar a função de *hashing*
  - Resto de Divisão
  - Multiplicação
  - Método da dobra

- Resto de Divisão: a posição da chave na tabela é dada pelo resto da divisão entre a mesma pelo tamanho da tabela
  - Exemplo: Seja  $B$  um arranjo de 7 elementos e deseja-se a inserção das chaves 1, 5, 10, 20, 25, 24

0	
1	
2	
3	
4	
5	
6	



- Resto de Divisão: a posição da chave na tabela é dada pelo resto da divisão entre a mesma pelo tamanho da tabela
  - Exemplo: Seja  $B$  um arranjo de 7 elementos e deseja-se a inserção das chaves 1, 5, 10, 20, 25, 24

0	
1	1
2	
3	
4	
5	
6	

$$1\%7 = 1$$

- Resto de Divisão: a posição da chave na tabela é dada pelo resto da divisão entre a mesma pelo tamanho da tabela
  - Exemplo: Seja  $B$  um arranjo de 7 elementos e deseja-se a inserção das chaves 1, 5, 10, 20, 25, 24

0	
1	1
2	
3	
4	
5	5
6	

$$5\%7 = 5$$

- Resto de Divisão: a posição da chave na tabela é dada pelo resto da divisão entre a mesma pelo tamanho da tabela
  - Exemplo: Seja  $B$  um arranjo de 7 elementos e deseja-se a inserção das chaves 1, 5, 10, 25, 20, 24

0	
1	1
2	
3	10
4	
5	5
6	

$$10 \% 7 = 3$$

- Resto de Divisão: a posição da chave na tabela é dada pelo resto da divisão entre a mesma pelo tamanho da tabela
  - Exemplo: Seja  $B$  um arranjo de 7 elementos e deseja-se a inserção das chaves 1, 5, 10, 20, 25, 24

0	
1	1
2	
3	10
4	
5	5
6	20

$$20\%7 = 6$$

- Resto de Divisão: a posição da chave na tabela é dada pelo resto da divisão entre a mesma pelo tamanho da tabela
  - Exemplo: Seja  $B$  um arranjo de 7 elementos e deseja-se a inserção das chaves 1, 5, 10, 20, 25, 24

0	
1	1
2	
3	10
4	25
5	5
6	20

$$25 \% 7 = 4$$

- Resto de Divisão: a posição da chave na tabela é dada pelo resto da divisão entre a mesma pelo tamanho da tabela
  - Exemplo: Seja  $B$  um arranjo de 7 elementos e deseja-se a inserção das chaves 1, 5, 10, 20, 25, 24

0	
1	1
2	
3	10, 24
4	25
5	5
6	20

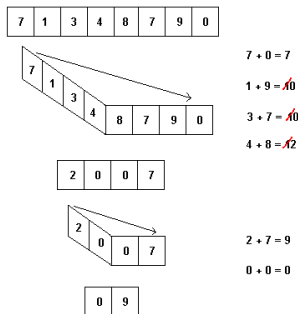
$$24 \% 7 = 3 \text{ (Colisão)}$$

- Multiplicação: a chave é multiplicada por uma constante  
 $0 < c < 1$ 
  - Seleção da parte fracionária da multiplicação da chave pela constante
  - Multiplicar essa parte pelo tamanho da tabela
  - A parte inteira dessa última multiplicação é usada como posição

- Multiplicação: a chave é multiplicada por uma constante  $0 < c < 1$ 
  - Exemplo:
    - Tamanho da tabela = 30, *chave* = 20,  $c = 0,26$
    - $chave * c = 5,2$
    - Parte fracionária = 0,2
    - Tamanho da tabela \* parte fracionária = 6,0
    - Parte inteira = 6 (o item de chave 20 será guardado em na posição 6)



- Método da dobra: a chave é interpretada como uma sequência de dígitos escrita em um papel
- Enquanto a chave for maior que o tamanho da tabela, o papel vai sendo dobrado ao meio e os dígitos que se sobrepõem são somados sem levar em consideração o "vai um"



- Colisões
  - Qualquer função *hashing* pode acarretar em colisões (chaves diferentes ocupam a mesma posição)
- Uma função *hashing* é perfeita quando chaves diferentes sempre são mapeadas para posições diferentes
- Uma função *hashing* é imperfeita quando podem acontecer colisões

- *Hash* universal:
  - A função *hash* é escolhida aleatoriamente no início de cada execução, de forma que minimize/evite tendências das chaves
  - Por exemplo,  $h(k) = (A * k + B) \% P$ 
    - $P$  é um número primo maior do que a maior chave  $k$
    - $A$  é uma constante escolhida aleatoriamente de um conjunto de constantes  $\{0, 1, 2, \dots, P - 1\}$  no início da execução
    - $B$  é uma constante escolhida aleatoriamente de um conjunto de constantes  $\{1, 2, \dots, P - 1\}$  no início da execução
  - Diz-se que  $h$  representa uma coleção de funções universais

## Implementação

- Na implementação a ser realizada nessa aula não tratará colisões
- O código será aprimorada na próxima aula para o tratamento de colisões
- Estrutura de dados simples para tabela *hash*

```
typedef struct{  
    int tamanho;  
    int *buckets;  
}HashTable;
```

- TAD simples para a tabela *hash*:

```
HashTable* gerarHT(unsigned int tam);
```

```
int procurar_chave(unsigned int chave, HashTable*  
t);
```

```
int inserir_chave(unsigned int chave, HashTable* t);
```

```
int remover_chave(unsigned int kchave, HashTable*  
t);
```

```
void imprimir_tabela(HashTable* t);
```

```
int liberar_tabela(HashTable* t);
```

- Código para arquivo .c

```
static int hash_code(int chave, int tam_tab){
    return chave % tam_tab;
}

HashTable* gerarHT(unsigned int tam){
    HashTable* t = malloc(sizeof(HashTable));
    int i;

    t->tamanho = tam;
    t->buckets = malloc(tam * sizeof(int));

    for (i = 0; i < tam; i++)
        t->buckets[i] = -1;

    return t;
}
```

- Código para arquivo .c

```
int procurar_chave(unsigned int chave, HashTable* t){
    int hc;

    if (t != NULL){
        hc = hash_code(chave, t->tamanho);

        if (t->buckets[hc] == chave)
            return hc;
    }

    return -1;
}
```



- Código para arquivo .c

```
int inserir_chave(unsigned int chave, HashTable* t){
    int hc;

    if (t != NULL){
        hc = hash_code(chave, t->tamanho);

        if (t->buckets[hc] < 0){
            t->buckets[hc] = chave;

            return 1;
        }

        printf("Houve colisao ao tentar inserir a chave %d!\n",
            chave);
    }

    return 0;
}
```

- Código para arquivo .c

```
int remover_chave(unsigned int chave, HashTable* t){
    int pos = procurar_chave(chave, t);

    if (pos >= 0){
        t->buckets[pos] = -1;

        return 1;
    }else
        return 0;
}

void imprimir_tabela(HashTable* t){
    int i;

    if (t != NULL)
        for (i = 0; i < t->tamanho; i++)
            if (t->buckets[i] > -1)
                printf("%d:  %d\n", i, t->buckets[i]);
}
```

- Código para arquivo .c

```
int liberar_tabela(HashTable* t){  
    int i;  
  
    if (t != NULL){  
        free(t->buckets);  
        free(t);  
  
        return 1;  
    }else  
        return 0;  
}
```



Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.  
*Introduction to Algorithms.*  
Third edition, The MIT Press, 2009.



Oliva, J. T.  
Tabela Hash. AE22CP – Algoritmos e Estrutura de Dados I.  
*Notas de Aula.* Engenharia de Computação.  
Dainf/UTFPR/Pato Branco, 2020.



Rosa, J. L. G.  
Métodos de Busca. SCE-181 – Introdução à Ciência da  
Computação II.  
*Slides.* Ciência de Computação. ICMC/USP, 2018.



Ziviani, N.  
*Projeto de Algoritmos - com implementações em Java e C++.*  
Thomson, 2007.