# Programação 2 _ T10

Tabelas de Dispersão

Rui Camacho
(slides por Luís Teixeira)
**MIEEC 2020/2021**

# TABELAS DE DISPERSÃO

## Definição:

- Tabela de **tamanho fixo** em que os elementos são colocados na posição determinada por uma função denominada **função de dispersão** (*hash function*).

## A função de dispersão deve:

- ser fácil de calcular
- distribuir os objectos **uniformemente** pela tabela

## Vantagem do uso de tabelas de dispersão:

- Asseguram **tempo médio constante** para inserção, remoção e pesquisa

# FUNÇÃO DE DISPERSÃO

Exemplo de Função de Dispersão:

F(x) = comprimento(x)%10

| *Nome* | *F(Nome)* |
|---|---|
| Carlos | 6 |
| Rodrigo | 7 |
| Artur | 5 |
| Ana | 3 |
| Miguel | 6 |
| Clementina | 0 |
| Aristófanes | 1 |

| | |
|---|---|
| 0 | Clementina |
| 1 | Aristófanes |
| 2 | |
| 3 | Ana |
| 4 | |
| 5 | Artur |
| 6 | Carlos / Miguel |
| 7 | Rodrigo |
| 8 | |
| 9 | |

É uma má função de dispersão, porque tem elevada probabilidade de levar a muitas colisões

# FUNÇÃO DE DISPERSÃO

A função de dispersão envolve o comprimento da tabela para que todos os resultados estejam dentro da gama pretendida

```
unsigned int hash(char *key, unsigned int tableSize) {
    unsigned int hashVal = 0;

    while( *key != '\0' )
        hashVal += *key++;

    return hashVal % tableSize;
}
```

```
unsigned int hash(int key, unsigned int tableSize) {
    if (key<0)
        key = -key;
    return key % tableSize;
}
```

A qualidade da função de dispersão depende do tamanho da tabela

- tamanhos primos são os melhores.

# COLISÕES

Se a função de dispersão não for injectiva, podem ocorrer situações de **colisão:**

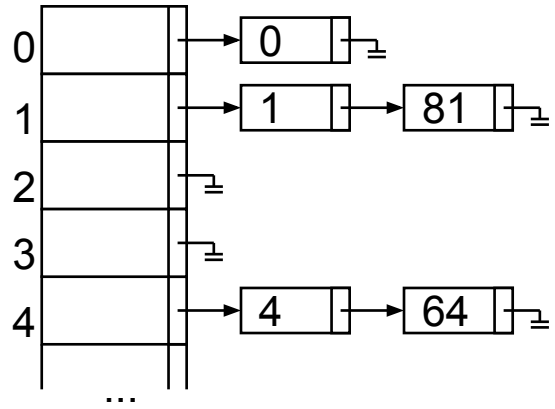
- tentativa de inserir mais do que um elemento na mesma posição da tabela de dispersão.

O comportamento de uma tabela de dispersão é caracterizado por:

- função de dispersão
- técnica de resolução de colisões

# TÉCNICAS DE RESOLUÇÃO DE COLISÕES

## Encadeamento / Endereçamento Fechado



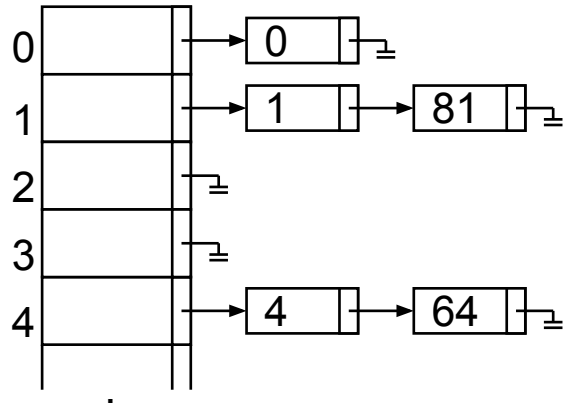

- Os elementos para os quais a função de dispersão devolve uma mesma posição P são colocados numa **lista** situada na posição P

## Endereçamento Aberto

- Cada posição da tabela guarda no **máximo 1 elemento**.
- Em caso de colisão, recorre-se à **sondagem**.
  - Casos típicos:
    - Sondagem linear
    - Sondagem quadrática

# ENCADEAMENTO / ENDEREÇAMENTO FECHADO

Desempenho pode ser medido pelo número de sondagens efetuadas, que depende do **fator de carga (λ)**




$$\lambda = \frac{nelementos}{tamanho}$$

Comprimento médio de cada lista: λ

Tempo médio de pesquisa (em número de sondagens)

- Pesquisa **sem** sucesso: λ
- Pesquisa **com** sucesso: 1 + λ/2

# ENCADEAMENTO – IMPLEMENTAÇÃO EM C

```
typedef struct list_node *node_ptr;

struct list_node
{
    element_type element;  /* element_type a definir */
    node_ptr next;
};

struct hash_tbl
{
    int Size;
    node_ptr *Lists;/* vetor de listas alocado mais tarde */
};

typedef struct hash_tbl *HASH_TABLE;
```

# ENCADEAMENTO – IMPLEMENTAÇÃO EM C

```
/* Cria tabela de dispersao, inicializando as listas a NULL */

HASH_TABLE create_HT(unsigned int table_size) {
    int i;
    HASH_TABLE HT = malloc(sizeof(struct hash_tbl));
    if(HT == NULL)
        return NULL;
    HT->Size = next_prime( table_size );
    HT->Lists = malloc(sizeof(node_ptr) * HT->Size);
    if(HT->Lists == NULL ){
      free(HT);
        return NULL; /* "Out of space!!! */
   }
    for(i=0; i<HT->Size; i++ )
        HT->Lists[i] = NULL;

    return HT;
}
```

# ENCADEAMENTO – IMPLEMENTAÇÃO EM C

```
/* Procura um elemento na tabela, devolvendo o endereco do no' da
respetiva lista ligada onde e' encontrado (ou NULL caso nao seja
encontrado) */

node_ptr search_HT( element_type elem, HASH_TABLE HT )
{
    unsigned int position;
    node_ptr elemAddr;

    position = hash(elem, HT->Size);
    elemAddr = HT->Lists[position];

    while(elemAddr!=NULL && elemAddr->element!=elem) /* ou strcmp */
         elemAddr = elemAddr->next;

    return elemAddr;
}
```

# ENCADEAMENTO – IMPLEMENTAÇÃO EM C

```
/* Insere elemento na tabela, caso dela ainda nao conste */

void insert_in_HT( element_type elem, HASH_TABLE HT ) {
    unsigned int position;
    node_ptr new_node, elemAddr = search_HT(elem, HT);

    if(elemAddr == NULL) /* se elem ainda nao consta da tabela... */
    {
        position = hash(elem, HT->Size);
        new_node = (node_ptr) malloc(sizeof(struct list_node));
        if(new_node == NULL)
            return; /* "Out of space!!! */

        new_node->element = elem; /* ou strcpy */
        new_node->next = HT->Lists[position];
        HT->Lists[position] = new_node;
    }
}
```

# ENDEREÇAMENTO ABERTO

Quando ocorre colisão, procura-se uma posição alternativa:

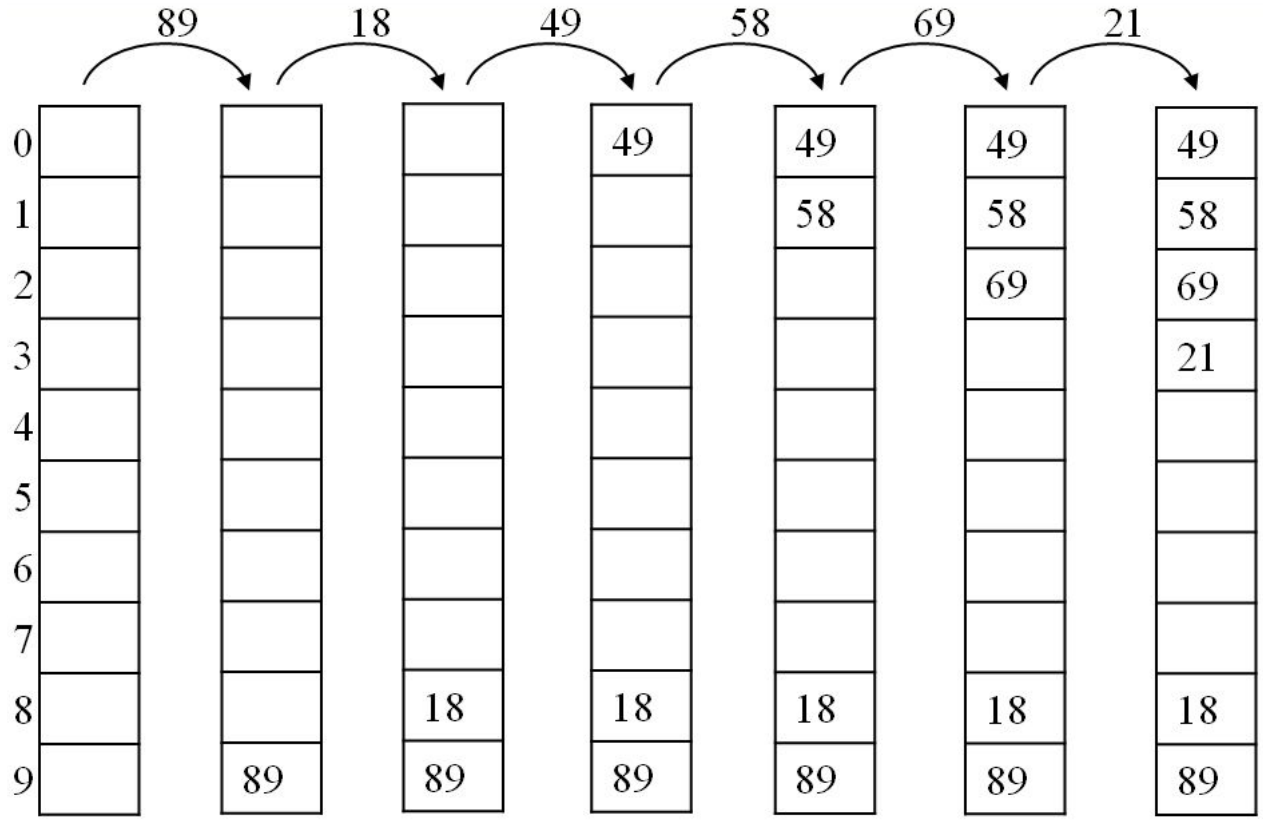
- Sonda-se **sequencialmente** as posições $H_1(x)$, $H_2(x)$, etc., até se encontrar uma posição livre, sendo:

$$\mathbf{H_i(x) = (hash(x) + f(i)) \ \% \ TamanhoTabela}$$

- **Sondagem linear** : f(i) = c*i
  - Se a constante c for um número primo, garante a utilização completa da tabela.
- **Sondagem quadrática** : $f(i) = a*i^2 + b*i$
  - Pode ser impossível inserir um elemento numa tabela com espaço
- **Dispersão dupla** : $f(i) = i*hash_2(x)$

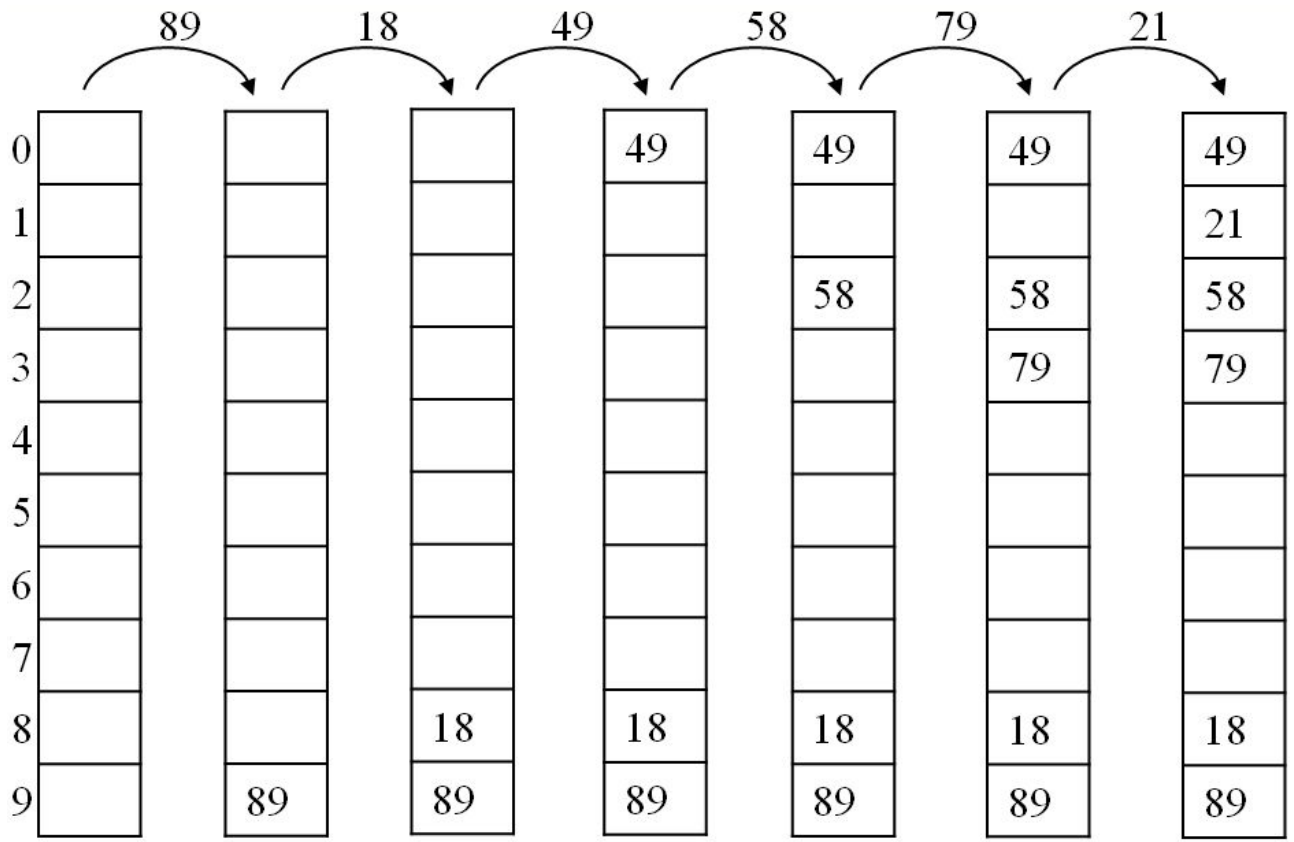
# SONDAGEM LINEAR - EXEMPLO

| hash(x) = x % 10 | $H_i(x) = ( hash(x) + i ) \% 10$ |
|---|---|



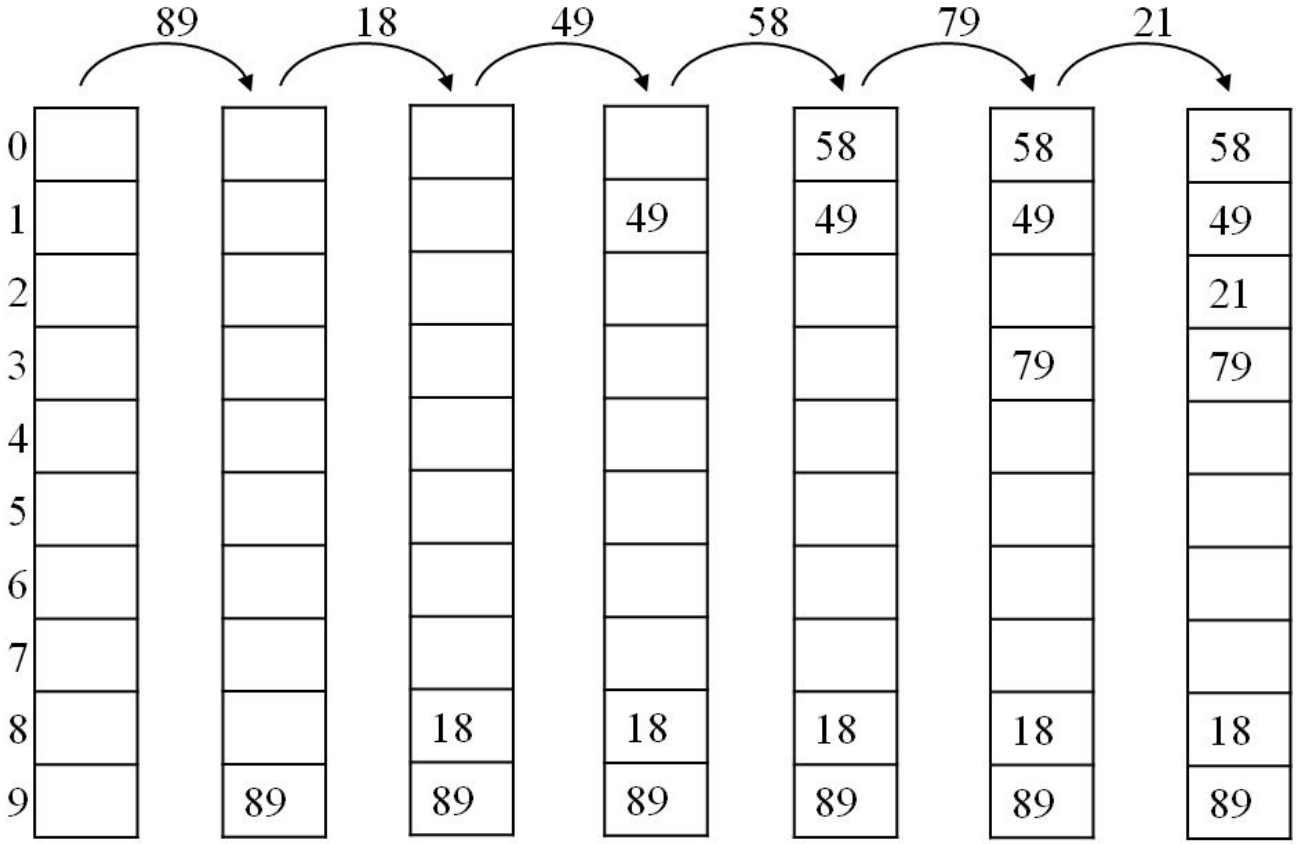
# SONDAGEM QUADRÁTICA - EXEMPLO

| $hash(x) = x \% 10$ | $H_i(x) = ( hash(x) + i^2 ) \% 10$ |
|---|---|



# DISPERSÃO DUPLA - EXEMPLO

| $hash(x) = x \% 10$ | $H_i(x) = ( hash(x) + i*hash_2(x) ) \% 10$ | $hash_2(x) = x\%3 + 1$ |
|---|---|---|



# ENDEREÇAMENTO ABERTO – IMPLEMENTAÇÃO EM C

```
enum kind_of_entry { legitimate, empty, deleted };

typedef struct hash_entry
{
    element_type element;  /* element_type a definir */
    enum kind_of_entry info;
} cell;

struct hash_tbl
{
    unsigned int Size;
    cell *Cells;    * vetor de celulas alocado mais tarde */
};

typedef struct hash_tbl *HASH_TABLE;
```

# ENDEREÇAMENTO ABERTO – IMPLEMENTAÇÃO EM C

```
/* Cria tabela de dispersao, sinalizando celulas como empty */

HASH_TABLE create_HT(unsigned int table_size) {
    int i;
    HASH_TABLE HT = malloc(sizeof(struct hash_tbl));
    if( HT == NULL )
        return NULL; /* "Out of space!!! */
    HT->Size = next_prime( table_size );
    HT->Cells = (cell*) malloc( sizeof(cell) * HT->Size );
    if( HT->Cells == NULL ) {
      free(HT);
        return NULL; /* "Out of space!!! */
   }
    for(i=0; i<HT->Size; i++ )
        HT->Cells[i].info = empty;

    return HT;
}
```

# ENDEREÇAMENTO ABERTO – IMPLEMENTAÇÃO EM C

```
/* Procura um elemento na tabela, devolvendo a posicao onde e' encontrado (ou onde inserir, caso nao seja encontrado) */

unsigned int search_HT( element_type elem, HASH_TABLE HT )
{
    unsigned int pos, i=0;
    pos = hash(elem, HT->Size );

    while((HT->Cells[pos].element != elem )    /* ou strcmp */
        && (HT->Cells[pos].info != empty ) )
    {
        pos += 2*(++i) - 1;

        if(pos >= HT->Size)
            pos -= HT->Size;
    }
    return pos;
}
```

# ENDEREÇAMENTO ABERTO – IMPLEMENTAÇÃO EM C

```
/* Insere elemento na tabela, caso dela ainda nao conste */

void insert_in_HT( element_type elem, HASH_TABLE HT )
{
    unsigned int pos;

    pos = search_HT( elem, HT );

    if( HT->Cells[pos].info != legitimate )
    {
        HT->Cells[pos].info = legitimate;
        HT->Cells[pos].element = elem;  /* ou strcpy */
    }
}
```