

## Tabelas de hash fechada: Exercício

Implemente a função de remoção de uma chave na tabela hashing fechada em questão.

```
void remover(Tabela tabela, int n)
{
    int posicao = buscar(tabela, n);
    if (posicao < tam)
        tabela[posicao].livre = 'R';
    else
        printf ("\nElemento nao esta presente.");
}
```

## Tabelas de hash fechada: Exercício

```
int buscar(Tabela tabela, int n)
{
    int i=0;
    int pos=funcaoHashing(n);
    while(i < tam && tabela[(pos+i)%tam].livre != 'L'
    && tabela[(pos+i)%tam].chave != n)
        i = i+1;
    if (tabela[(pos+i)%tam].chave == n &&
    tabela[(pos+i)%tam].livre == 'O')
        return (pos+i)%tam;
    else
        return tam; // não encontrado
}
```

## Tabelas de hash fechada: Exercício

Para uma melhor fixação do tópico em estudo com base na definição do TAD abaixo implemente suas funções ainda não definidas.

```
#define tam 8
typedef struct
{
    int chave;
    char livre; // L = livre, O = ocupado, R = removido
}Hash;
typedef Hash TabelaHash [tam];
void inicializarHash(TabelaHash *);
int funcaoHashing(int);
void mostrarHash(TabelaHash);
void inserirChave(TabelaHash, int);
int buscarChave(TabelaHash, int);
void removerChave(TabelaHash, int);
```

## Tabelas de hash fechada – tentativa quadrática

Na tentativa linear as chaves que colidem geram os chamados agrupamentos primários, o que aumenta o tempo de busca.

Agora estudaremos a tentativa quadrática. Neste tipo de tentativa, outro tipo de agrupamento também é gerado, denominado agrupamento secundário, mas as degradações são reduzidas se comparadas com a tentativa linear.

## Tabelas de hash fechada – tentativa quadrática

Segundo Szwarcfiter (1994), para a aplicação deste método os endereços calculados pela função de hashing  $h'(x, k)$  devem corresponder à varredura de toda tabela, para  $k=0, \dots, m-1$ . As equações recorrentes abaixo fornecem uma maneira de calcular os endereços diretamente.

$$h'(x, 0) = h(x)$$

$$h'(x, k) = (h'(x, k-1) + k) \bmod m$$

para  $1 \leq k \leq m-1$ , sendo que  $h(x) = x \bmod m$ .

## Tabelas de hash fechada – tentativa quadrática

Para uma melhor compreensão analisaremos agora um exemplo, também nos utilizaremos de uma tabela com 8 entradas.

Inicialmente temos:

Índice	Situação	Chave
0	L	
1	L	
2	L	
3	L	
4	L	
5	L	
6	L	
7	L	

Inserir chave 12  
 $12\%8=4$

Índice	Situação	Chave
0	L	
1	L	
2	L	
3	L	
4	O	12
5	L	
6	L	
7	L	

## Tabelas de hash fechada – tentativa quadrática

Índice	Situação	Chave
0	L	
1	L	
2	L	
3	L	
4	O	12
5	L	
6	L	
7	L	

Inserir chave 20

$$20 \% 8 = 4$$

$$(4 + 1) \bmod 8 = 5$$

Índice	Situação	Chave
0	L	
1	L	
2	L	
3	L	
4	O	12
5	O	20
6	L	
7	L	

## Tabelas de hash fechada – tentativa quadrática

Índice	Situação	Chave
0	L	
1	L	
2	L	
3	L	
4	O	12
5	O	20
6	L	
7	L	

Inserir chave 28

$$28 \% 8 = 4$$

$$(4+1) \bmod 8 = 5$$

$$(5+2) \bmod 8 = 7$$

Índice	Situação	Chave
0	L	
1	L	
2	L	
3	L	
4	O	12
5	O	20
6	L	
7	O	28



## Tabelas de hash fechada – tentativa quadrática

Índice	Situação	Chave
0	L	
1	L	
2	L	
3	L	
4	O	12
5	O	20
6	L	
7	O	28

Inserir chave 36

$$36 \% 8 = 4$$

$$(4+1) \bmod 8 = 5$$

$$(5+2) \bmod 8 = 7$$

$$(7+3) \bmod 8 = 2$$

Índice	Situação	Chave
0	L	
1	L	
2	O	36
3	L	
4	O	12
5	O	20
6	L	
7	O	28

## Tabelas de hash fechada – tentativa quadrática

Índice	Situação	Chave
0	L	
1	L	
2	O	36
3	L	
4	O	12
5	O	20
6	L	
7	O	28

Remover chave 12

$$12 \% 8 = 4$$

Índice	Situação	Chave
0	L	
1	L	
2	O	36
3	L	
4	R	12
5	O	20
6	L	
7	O	28

## Tabelas de hash fechada – tentativa quadrática

Índice	Situação	Chave
0	L	
1	L	
2	O	36
3	L	
4	R	12
5	O	20
6	L	
7	O	28

Remover chave 20

$$20\%8=4$$

$$(4+1)\%8=5$$

Índice	Situação	Chave
0	L	
1	L	
2	O	36
3	L	
4	R	12
5	R	20
6	L	
7	O	28

## Tabelas de hash fechada – tentativa quadrática

Índice	Situação	Chave
0	L	
1	L	
2	O	36
3	L	
4	R	12
5	R	20
6	L	
7	O	28

Inserir chave 44

$$44 \% 8 = 4$$

Índice	Situação	Chave
0	L	
1	L	
2	O	36
3	L	
4	O	44
5	R	20
6	L	
7	O	28

## Tabelas de hash fechada: Exercício

Com base no que foi apresentado defina a(s) estrutura(s) de dados necessária(s) para a implementação de uma tabela hashing fechada com tentativa quadrática.

```
#define tam 8  
typedef struct  
{  
    int chave;  
    char livre; /* L = livre, O = ocupado, R =  
removido*/  
    }Hash;  
typedef Hash Tabela[tam];
```

## Tabelas de hash fechada: Exercício

Agora, implemente a função de inserção de uma chave na tabela hashing fechada em questão.

```
void inserirChave (Tabela tabela, int n)
{
    int pos = funcaoHashing(n);
    int k=1;
    while(k < tam && tabela[pos].livre != 'L' &&
tabela[pos].livre != 'R')
    {
        pos = (pos+k)%tam;
        k = k+1;
    }
}
```

## Tabelas de hash fechada: Exercício

```
if(k < tam)
{
    tabela[pos].chave = n;
    tabela[pos].livre = 'O';
}
else
    printf("\nTabela cheia ou em loop!");
}
```

```
int funcaoHashing(int num)
{
    return num % tam;
}
```

## Tabelas de hash fechada: Exercício

Implemente a função de remoção de uma chave na tabela hashing fechada em questão.

```
void removerChave(Tabela tabela, int n)
{
    int posicao = buscarChave (tabela, n);
    if (posicao < tam)
        tabela[posicao].livre = 'R';
    else
        printf("\nElemento nao estah presente.");
}
```



```
int buscarChave(Tabela tabela, int n)
{
    int pos = funcaoHashing(n);
    int k=1;
    while(k <= tam && tabela[pos].livre !='L'&&
tabela[pos].chave != n)
    {
        pos = (pos+k)%tam;
        k = k+1;
    }
    if(tabela[pos].chave == n && tabela[pos].livre =='O')
        return pos;
    else
        return tam;
}
```

## Tabelas de hash fechada: Exercício

Para uma melhor fixação do tópico em estudo com base na definição do TAD abaixo implemente suas funções ainda não definidas.

```
#define tam 8
typedef struct
{
    int chave;
    char livre; /* L = livre, O = ocupado, R = removido*/
}Hash;
typedef Hash Tabela[tam];
int funcaoHashing(int);
void inicializarTabela(Tabela);
void mostrarHash(Tabela);
void inserirChave (Tabela, int);
int buscarChave(Tabela, int);
void removerChave(Tabela, int);
```

## Função hashing

Antes de finalizarmos nosso estudo sobre tabelas hashing cabe destacar que uma função hashing:

- ▶ Possui o objetivo de transformar o valor de chave de um elemento de dados em uma posição para este elemento em um dos  **$b$**  subconjuntos definidos.
  - ▶ É um **mapeamento** de  **$K \rightarrow \{1, \dots, b\}$** , onde  **$K = \{k_1, \dots, k_m\}$**  é o conjunto de todos os valores de chave possíveis no universo de dados em questão.

- Deve dividir o universo de chaves  $K = \{k_1, \dots, k_m\}$  em  $b$  subconjuntos de mesmo tamanho.
  - A probabilidade de uma chave  $k_j \in K$  aleatória qualquer cair em um dos subconjuntos  $b_i : i \in [1, b]$  deve ser uniforme.
  - Se a função de Hashing não dividir  $K$  uniformemente entre os  $b_i$ , a tabela de hashing pode degenerar.
    - O pior caso de degeneração é aquele onde todas as chaves caem em um único conjunto  $b_i$ .
  - A função “primeira letra” vista em um exemplo trabalhado é um exemplo de uma função ruim.
    - A letra do alfabeto com a qual um nome inicia não é distribuída uniformemente. Quantos nomes começam com “X” ?

## Funções de Hashing

Até o momento utilizamos um exemplo simples porém, muito utilizado de função hashing. Agora veremos outras técnicas utilizadas para implementar funções hashing que visão garantir a distribuição uniforme de um universo de chaves entre  $b$  conjuntos.

- Lembre-se: a função de hashing  $h(k_j) \rightarrow [1, b]$  recebe uma chave  $k_j \in \{k_1, \dots, k_m\}$  e devolve um número  $i$ , que representa o índice do subconjunto  $b_i : i \in [1, b]$  onde o elemento possuidor dessa chave vai ser colocado.

## Funções de Hashing

- ✦ As funções de hashing abordadas adiante supõem sempre uma chave simples, um único dado, seja string ou número, sobre o qual será efetuado o cálculo.
- ✦ Hashing sobre mais de uma chave, p.ex. “Nome” E “CPF” também é possível, mas implica em funções mais complexas.

## Funções de Hashing: Meio do Quadrado

- Calculada em dois passos:
  - Eleva-se a chave ao quadrado
  - Utiliza-se um determinado número de dígitos ou bits do meio do resultado.
- Idéia geral:
  - A parte central de um número elevado ao quadrado depende dele como um todo.
- Quando utilizamos diretamente bits:
  - Se utilizarmos  $r$  bits do meio do quadrado, podemos utilizar o seu valor para acessar diretamente uma tabela de  $2^r$  entradas.

## Funções de Hashing: Folding ou Desdobramento

### ➤ Método para cadeias de caracteres

- Inspirado na idéia de se ter uma tira de papel e de se dobrar essa tira formando um “bolinho” ou “sanfona”.
- Baseia-se em uma representação numérica de uma cadeia de caracteres.
  - Pode ser binária ou ASCII.

### ➤ Dois tipos:

- Shift Folding e
- Limit Folding.



## Funções de Hashing: Shift Folding

➤ Dividi-se uma string em pedaços, onde cada pedaço é representado numericamente e soma-se as partes.

➤ Exemplo mais simples: somar o valor ASCII de todos os caracteres.

➤ O resultado é usado diretamente ou como chave para uma  $h'(k)$

### ➤ Exemplo:

➤ Suponha que os valores ASCII de uma string sejam os seguintes:

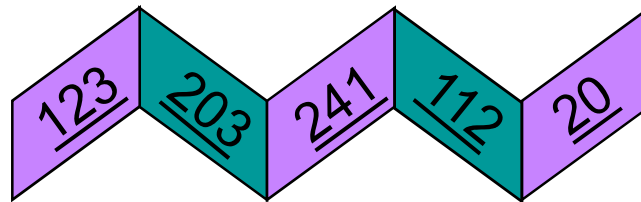
**123, 203, 241, 112 e 20**

➤ O folding será:

$$123 + 203 + 241 + 112 + 20 = 699$$

## Funções de Hashing: Limit Folding

- Usando a idéia de uma tira de papel como sanfona:
  - Exemplo mais simples: somar o valor ASCII de todos os caracteres, invertendo os dígitos a cada segundo caracter.



### ➤ Exemplo:

- Suponha que os valores ASCII de um string sejam os seguintes:

123, 203, 241, 112 e 20

- O folding será:

$$123 + 302 + 241 + 211 + 20 = 897$$

## Funções de Hashing: Análise de Frequência

- Se estabelece uma função escolhendo quais dígitos entrarão como argumentos da função com base na sua probabilidade de ocorrência
- Tabula-se a ocorrência de cada caractere em cada posição (a partir de uma amostra) e escolhe-se as k posições que apresentam valores mais uniformemente distribuídos
- Desvantagem: deve conhecer boa amostra das chaves

## Tabela hash: Vantagens

### ➤ Simplicidade

- É muito fácil de implementar um algoritmo para implementar hashing.

### ➤ Escalabilidade

- Podemos adequar o tamanho da tabela de hashing ao  $n$  esperado em nossa aplicação.

### ➤ Eficiência para $n$ grandes

- Para trabalharmos com problemas envolvendo  $n = 1.000.000$  de dados, podemos imaginar uma tabela de hashing com 2.000 entradas, onde temos uma divisão do espaço de busca da ordem de  $n/2.000$  de imediato.

## Tabela hash: Desvantagens

- Dependência da escolha de função de hashing
  - Para que o tempo de acesso médio ideal  $T(n)$  seja mantido, é necessário que a função de hashing divida o universo dos dados de entrada em  $b$  conjuntos de tamanho aproximadamente igual.
- Tempo médio de acesso é ótimo somente em uma faixa
  - Existe uma faixa de valores de  $n$ , determinada por  $b$ , onde o hashing será muito melhor do que uma árvore.
  - Fora dessa faixa é pior.