

Algoritmos e estruturas de Dados

Tabelas Hash

Busca

Problema:

Encontrar dados em uma estrutura (ex. um vetor)

O valor a ser buscado na estrutura será chamado **chave**

Exemplo:

Seja a struct *aluno*.

Verificar as notas da aluna *Carol* no vetor *v* abaixo (cada item do vetor é uma *struct*).

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

A *chave* a ser buscada é **Carol**

V =	Bob	Ana	Carol	Tom
	7.5; 8.5; 9.0	8.5; 8.0; 9.5	8.5; 8.0; 9.5	8.0; 7.0; 8.5

Busca

Problema: Verificar as notas da aluna *Carol* no vetor *v* abaixo (cada item do vetor é uma *struct*):

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

	Bob	Ana	Carol	Tom
V =	7.5; 8.5; 9.0	8.5; 8.0; 9.5	8.5; 8.0; 9.5	8.0; 7.0; 8.5

Solução 1:

- Percorrer todo o vetor e comparar o valor procurado com cada um dos itens.
- Custo: $O(n)$.

Solução 2:

- Ordenar o vetor e, depois disso, fazer busca binária.
- Custo: $O(n \log n)$
 - Ordenação: $O(n \log n)$ (merge sort)
 - Busca binária: $O(\log n)$

Busca

Métodos de buscas usam o mesmo princípio:

Procurar a informação desejada com base na comparação das chaves:

Examinar cada posição do vetor até encontrar a chave procurada

Problema:

- Elementos de forma ordenada;
- Custo da ordenação: $O(N \log N)$;
- Custo da busca: $O(\log N)$

Busca

Solução ideal:

Acesso direto ao elemento procurado
sem necessidade de comparação: $O(1)$

Exemplo:

Identificar que *Carol* ocupa a 3ª posição
no vetor *V* sem percorrê-lo.

(isto é como acessar “v[Carol]” em vez de v[2])

V =	Bob	Ana	Carol	Tom
	7.5; 8.5; 9.0	8.5; 8.0; 9.5	8.5; 8.0; 9.5	8.0; 7.0; 8.5

Busca

- **Vetores** permitem acesso direto à informação, com custo $O(1)$.

Ex. $v[2] = \{\text{Carol}, 8.5, 8.0, 9.5\}$

5001	5002	5003	5004	5005	5006
5003		Bob 7.5; 8.5; 9.0	Ana 8.5; 8.0; 9.5	Carol 8.5; 8.0; 9.5	Tom 8.0; 7.0; 8.5
v		v[0] ou v+0	v[1] ou v+1	v[2] ou v+2	v[3] ou v+3

- Mas encontrar a posição de uma determinada informação não é $O(1)$.

- Ex.: Qual posição armazena os dados de Carol?

V =	Bob	Ana	Carol	Tom
	7.5; 8.5; 9.0	8.5; 8.0; 9.5	8.5; 8.0; 9.5	8.0; 7.0; 8.5

TABELA HASH

- Tabelas de **indexação** ou de **espalhamento**
- Utiliza uma função para **espalhar** os elementos na tabela
- Utilizam a mesma função para acessar diretamente os elementos ($O(1)$)
- Elementos ficam dispersos de forma não ordenada dentro de um vetor.
- Como espalhar os elementos?

TABELA HASH

- Função Hashing
 - Função que espalha os elementos na tabela

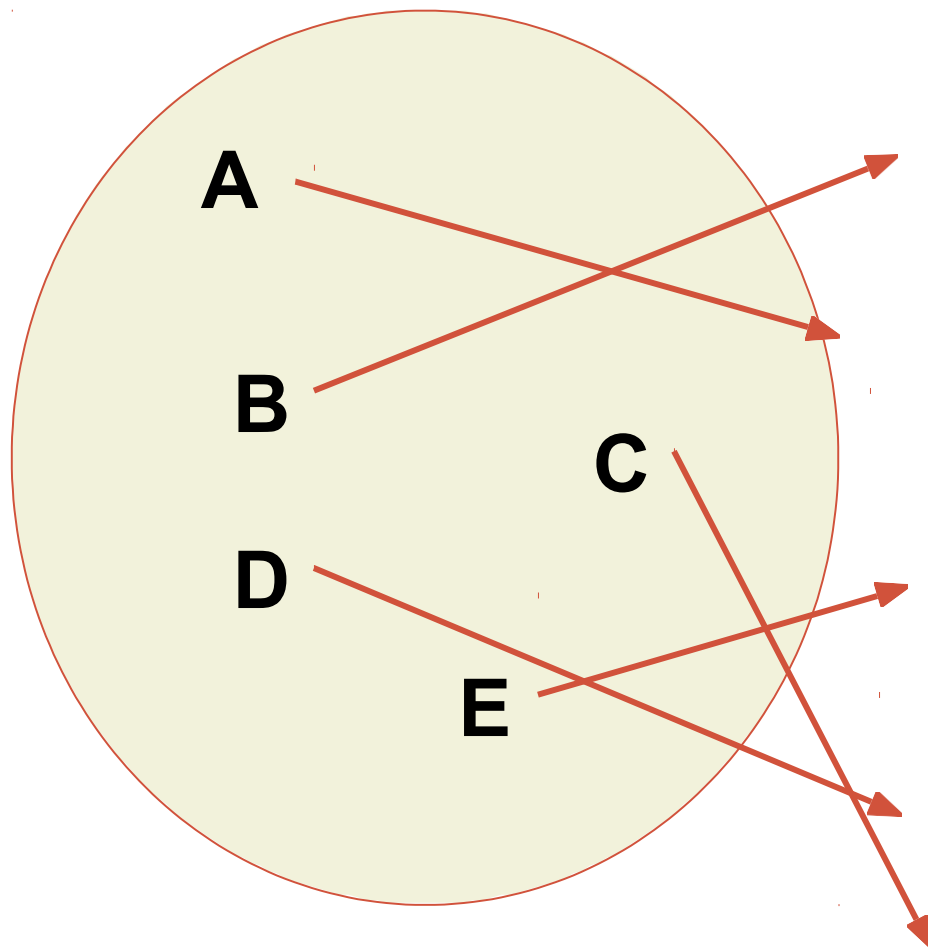


TABELA HASH

NULL	0
B	1
NULL	
A	
NULL	
E	
NULL	
D	
C	
NULL	N - 1

TABELA HASH

■ Associa valores a chaves

- **Chave:** Parte da **informação** que compõe o **elemento** a ser inserido ou localizado na tabela (ex. *Carol*)
- Valor: **Índice** onde o elemento se encontra no **vetor** que define a tabela (ex. *Carol* → 2)

A partir da chave é possível acessar um determinado elemento no vetor

(ex. *Carol* → 2)

5001	5002	5003	5004	5005	5006
5003		Bob 7.5; 8.5; 9.0	Ana 8.5; 8.0; 9.5	Carol 8.5; 8.0; 9.5	Tom 8.0; 7.0; 8.5
v		v[0] ou v+0	v[1] ou v+1	v[2] ou v+2	v[3] ou v+3

Vantagens e Desvantagens

- Vantagens
 - Melhor performance na busca
 - Implementação Simples
- Desvantagens
 - Alto custo para recuperar elementos da tabela ordenada pela chave, porque é necessário ordenar a tabela.
 - Alto número de colisões

IMPLEMENTAÇÃO

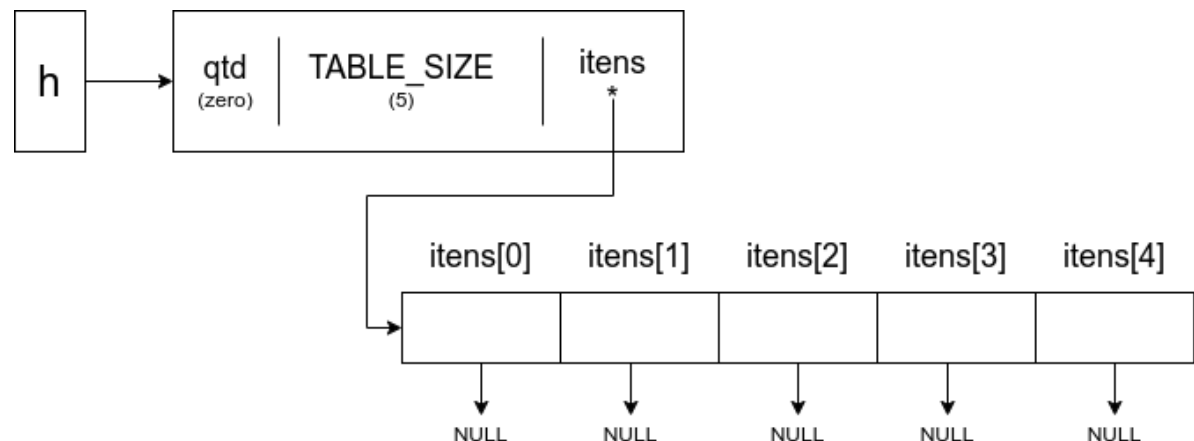
- Implementação usando uma estrutura Sequencial Estática
- Um vetor irá armazenar os elementos

```
struct{  
    char nome[30];  
    float n1, n2, n3;  
} typedef Aluno;
```

```
struct{  
    int qtd, TABLE_SIZE;  
    Aluno **itens;  
} typedef Hash;
```



```
Hash *criaHash(int TABLE_SIZE) {  
    Hash *h = malloc(sizeof(Hash));  
    int i;  
    h->TABLE_SIZE = TABLE_SIZE;  
    h->itens = malloc(TABLE_SIZE*sizeof(Aluno));  
    h->qtd = 0;  
    for (i=0; i<h->TABLE_SIZE; i++) {  
        h->itens[i]=NULL;  
    }  
    return h;  
}
```



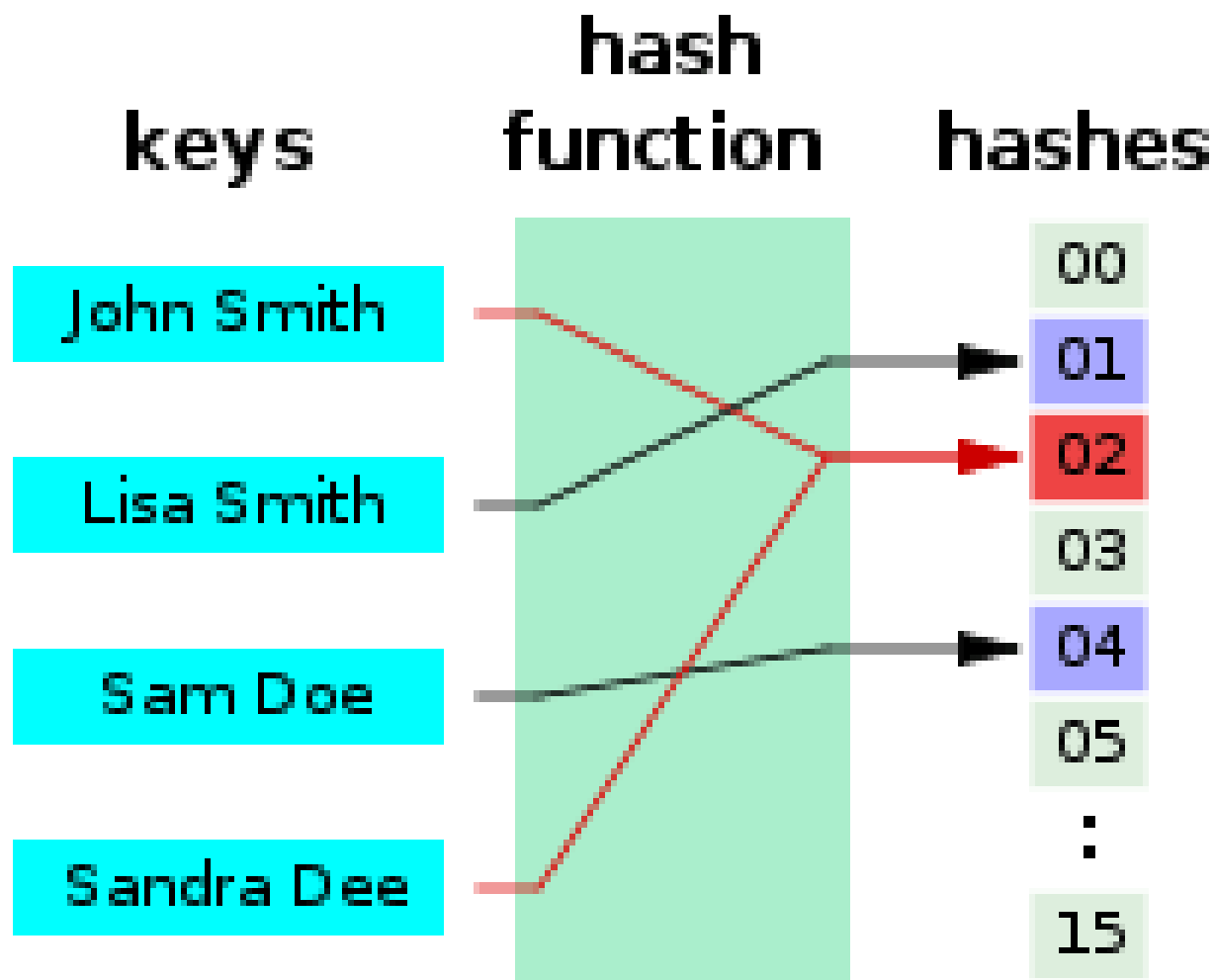
```
void liberaHash(Hash *h){  
    if(h!=NULL){  
        int i;  
        for(i=0;i<h->TABLE_SIZE;i++){  
            if(h->itens[i]!=NULL)  
                free(h->itens[i]);  
        }  
        free(h->itens);  
        free(h);  
    }  
}
```

Calculando a posição da chave

- Sempre que for **inserir** ou **buscar** um elemento é necessário **calcular** a **posição** dos dados dentro da tabela.
- Implementando a **Função Hashing**
 - Permite calcular uma posição a partir de uma **chave**
 - Responsável por distribuir as informações de forma equilibrada para minimizar as **colisões**.

Colisões

Colisões acontecem quando 2 elementos tentam ocupar o mesmo lugar da tabela **hash**



Função Hash - Requisitos

- Ser simples
- Valores diferentes produzem códigos *hash* diferentes (e mapeiam para posições diferentes)
- Distribuição Equilibrada (minimizar/evitar colisões)
- Conhecimento prévio e domínio da chave a ser utilizada

Função Hash - Requisitos

Seja U o universo de todas as chaves possíveis.

Ex.: U = conjunto de todas as palavras com 4 letras:

$$|U| = 26^1 + 26^2 + 26^3 + 26^4 = 475.254 \text{ chaves possíveis}$$

Em geral, a quantidade de possíveis chaves é menor que $|U|$

Seja $m \leq |U|$ a quantidade de chaves possíveis. Pode-se armazenar todos os dados em um vetor de tamanho m , sendo que cada elemento com chave k será armazenado na posição $h(k)$, tal que $h: U \longrightarrow \{0, 1, \dots, m-1\}$

Função Hash

Método da Divisão

- Calcular resto da divisão do valor inteiro que representa o elemento pelo tamanho da tabela "TABLE_SIZE"

```
int chaveDivisao ( int chave, int TABLE_SIZE) {  
    return chave % TABLE_SIZE;  
}
```

Ponto importante – evitar colisões

- Para definição do tamanho da tabela **Hash**, sugere-se utilizar um **número primo**
- Números primos evitam colisões

Cada chave k que compartilha um fator com m será mapeada uma posição múltipla desse fator

Função Hash

Sabe-se que, para dois inteiros $i=f.v$ e $j=f.w$:

$$i \bmod j = f.v \bmod f.w = f.(v \bmod w)$$

Seja uma tabela hash de tamanho m e uma chave k tal que m e k compartilham um fator f . Isto é:

$$m=f.x \text{ e } k=f.y$$

para algum x e algum y

Assim, $k \bmod m = f.(y \bmod x)$.

Ou seja, o resto da divisão de k por m será um múltiplo de f .

Inserção na tabela Hash

- Calcular a **posição** da **chave**
- Alocar espaço
- Inserir os dados

Inserção na tabela Hash

```
void insere(Hash *h, Aluno *a){  
    if(h==NULL){  
        printf("Não será possível inserir");  
    }  
    else{  
        int posicao = gera_hash(a->nome,h->TABLE_SIZE);  
        h->itens[posicao] = a;  
    }  
}
```

Busca na tabela Hash

- Calcular a **posição** da **chave**
- Verificar se há dados na **posição** calculada
- Retornar os dados

Busca na tabela Hash

```
Aluno *busca(Hash *h, char *nome){  
    int posicao = gera_hash(nome,h->TABLE_SIZE);  
    return h -> itens[posicao];  
}
```

Resolução colisões por encadeamento

- Todos os elementos que efetuam hash para uma mesma posição são armazenados em uma lista encadeada.
- A tabela hash de tamanho n é um vetor de listas encadeadas.
- Pior caso: todos os n elementos fazem hash para mesma posição.
 - Busca: $O(n)$
 - Inserção: $O(n)$

Resolução colisões por encadeamento

Exemplo – L U N E S:

$$h(L)=h(12)=5$$

$$h(U)=h(21)=0$$

$$h(N)=h(14)=0$$

$$h(E)=h(5)=5$$

$$h(S)=h(19)=5$$

Considerando:

$$h(k) = k \bmod n$$

e

k = inteiro correspondente ao caractere

$$k(A)=1, k(B)=2, k(C)=3, \dots$$

0	1	2	3	4	5	6
U					L	
N					E	
					S	

Resolução colisões por sondagem linear

(linear probing)

- Aplicável quando sabe-se a quantidade máxima m de elementos da tabela de tamanho n e $m \leq n$.
- Elemento de chave k é colocado na posição obtida pelo cálculo do *hash* (i.e. $h(k)$)
- No caso de colisão, calcula-se novo hash:
 - $h(k) = (h(k) + j) \bmod n$
- onde j indica a quantidade de tentativas de executar o hash (s.t. $1 \leq j \leq n-1$)

- Exemplo – L U N E S:

- $h(L) = h(12) = 5$
- $h(U) = h(21) = 0$
- $h(N) = h(14) = 0$
- $h(E) = h(5) = 5$
- $h(S) = h(19) = 5$

0	1	2	3	4	5	6
U	N	S			L	E