Algoritmos e estruturas de Dados

Tabelas Hash

Prof. Maiquel de Brito

Busca

Problema: Verificar as notas da aluna *Carol* vetor v abaixo (cada item do vetor é uma *struct*)*:*

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

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

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.

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.

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:

- É possível acessar facilmente a informação em uma determinada posição – *O(1)*
  - Ex. v[2] = {Carol, 8.5, 8.0, 9.5}
- Mas calcular a posição de uma determinada informação não é *O(1).*
  - Ex.: Qual posição armazena os dados de Carol*?*

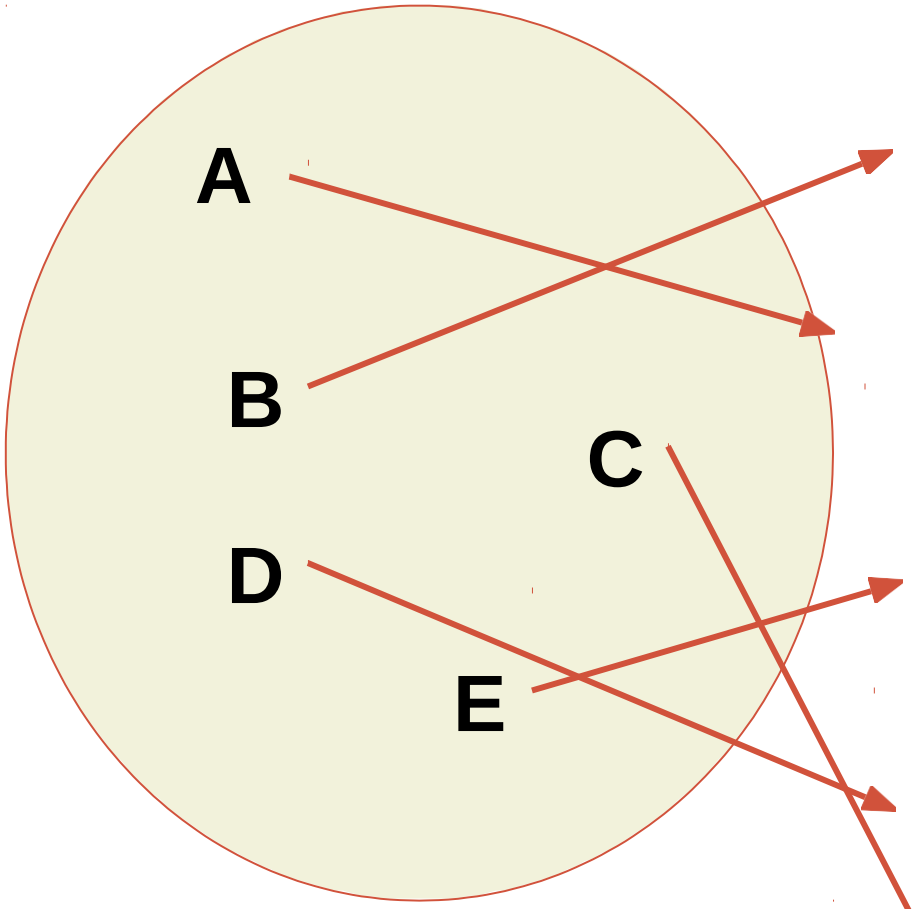
Bob	Ana	Carol	Tom
7.5; 8.5; 9.0	8.5; 8.0; 9.5	8.5; 8.0; 9.5	8.0; 7.0; 8.5

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



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

TABELA HASH

TABELA HASH

- Porque o espalhamento?
- Associa **valores** a **chaves**
- **Chave:** Parte da **informação** que compõe o **elemento** a ser inserido ou localizado na tabela
- Valor: **Índice** onde o elemento se encontra no **vetor** que define a tabela
- A partir da chave é possível acessar um determinado elemento no vetor.

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;
}
```

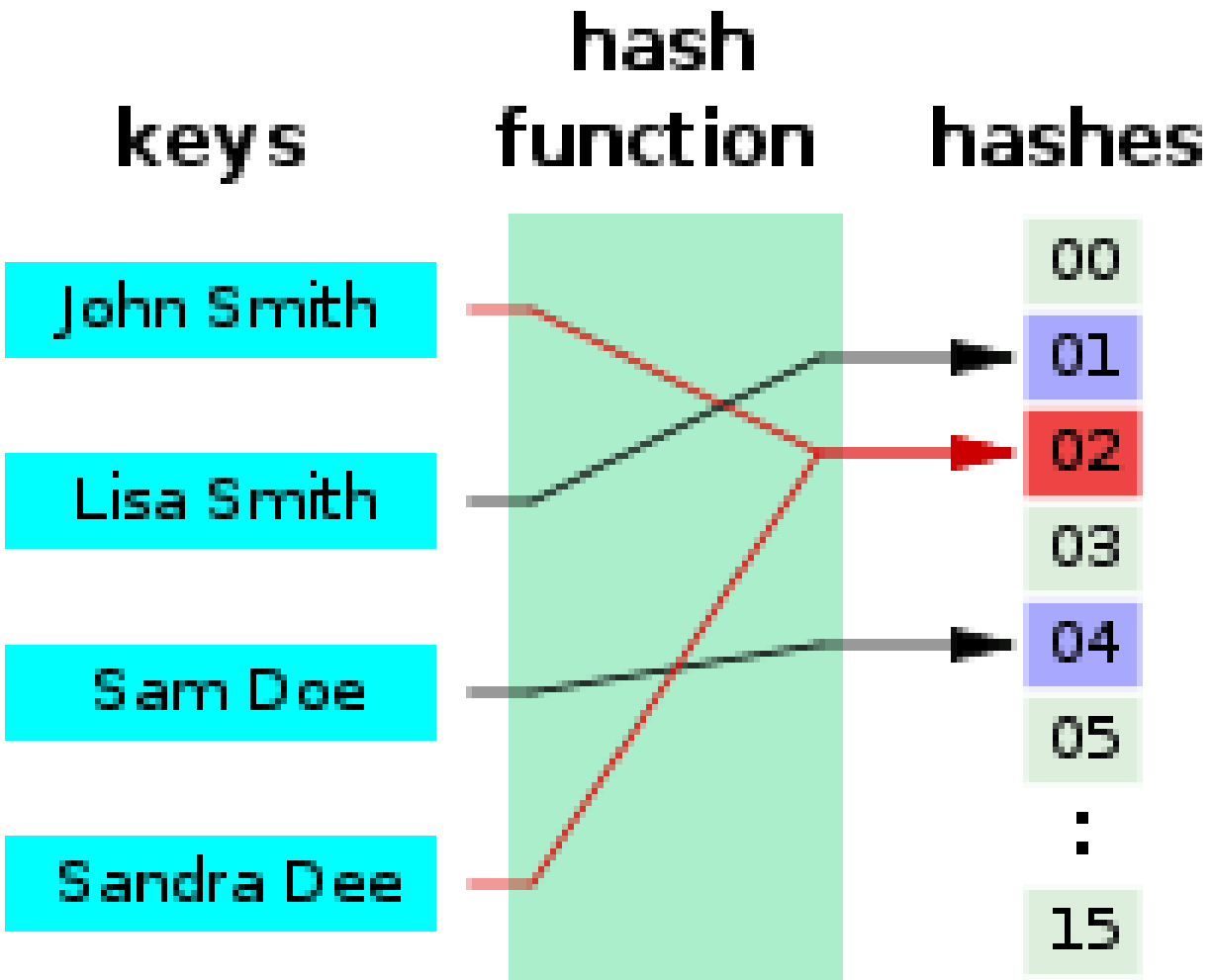
```
void liberaHash(Hash *h){
   if(h!=NULL){
      int i;
      for(i=0;i<h->TABLE_SIZE;i++){
         if(h->itens[i]!=NULL)
            free(h->itens[i]);
      }
      free(h->itens);
      free(h);
   }
}
```

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**


keys
hash function
hashes
John Smith
Lisa Smith
Sam Doe
Sandra Dee
00
01
02
03
04
05
:
15

Função Hash - Requisitos

- Ser simples
- Valores diferentes produz 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 $m <= |U|$ 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* $h: U \longrightarrow \{0, 1, \ldots, m-1\}$

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

Inserção na tabela Hash

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

Inserção na tabela Hash

```
void insere(Hash *h, Aluno *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){

}
```

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 é um vetor de listas encadeadas.
- Pior caso: todos os *n* elementos fazem hash para mesma posição.
    - Busca: *O(n)*
    - Inserção: *O(n)*

Tratando colisões

- Chaves diferentes com posições iguais
- Usando **sondagem Linear**
  - Espalha os elementos de forma sequencial a partir da posição calculada
  - Primeiro elemento é colocado na posição (pos) obtida na **função hashing**
  - Segundo elemento (colisão) é colocado na posição **pos+1**