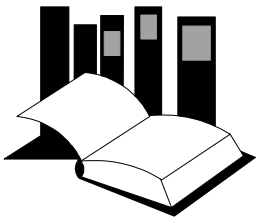
Tabelas Hash



- Nesta aula são discutidos modos de armazenar informações em um vetor, e depois procurar por uma informação
- Tabelas Hash constituem uma abordagem comum para o problema de armazenar e procurar dados
- Esta apresentação introduz a tabela hash

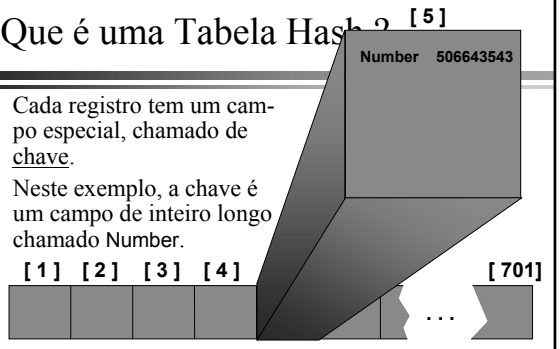
O Que é uma Tabela Hash ?

- A forma mais simples de tabelas hash é um vetor de registros.
- Este exemplo tem 701 registros.

Um vetor de registros

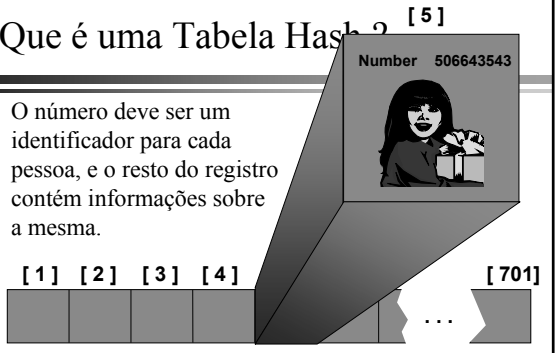
O Que é uma Tabela Hash ?

- Cada registro tem um campo especial, chamado de chave.
- Neste exemplo, a chave é um campo de inteiro longo chamado Number.



O Que é uma Tabela Hash ?

- O número deve ser um identificador para cada pessoa, e o resto do registro contém informações sobre a mesma.



O Que é uma Tabela Hash ?

- Quando uma tabela hash está sendo usada, algumas células contêm registros válidos, e outras estão "vazias".

Inserindo um Novo Registro

- Para inserir um novo registro, a **chave** deve, de alguma maneira, ser **convertida** em um **índice** de vetor.
- O índice é chamado de **valor hash** da chave.

Inserindo um Novo Registro

- Maneira típica de se criar um valor hash:

  (Number % 701) + 1

Quanto é (580625685 % 701) + 1 ?

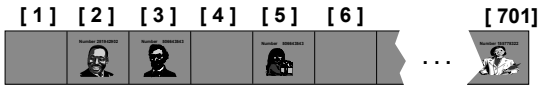
Inserindo um Novo Registro

- Maneira típica de se criar um valor hash:

  (Number % 701) + 1

Quanto é (580625685 % 701) + 1 ?

4



Inserindo um Novo Registro

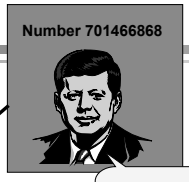
- O valor hash é usado para a localização do novo registro.

4

Inserindo um Novo Registro

- O valor hash é usado para a localização do novo registro.

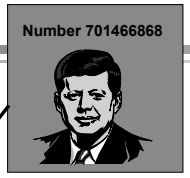
Colisões



- Aqui temos um novo registro a ser inserido, com um valor hash igual a 3.

Meu valor hash é [3].

Colisões



- Isto é chamado de **colisão**, porque já há outro registro válido em [3].

Quando ocorrer uma colisão, mova-se pelo vetor até encontrar uma célula vazia.

Colisões

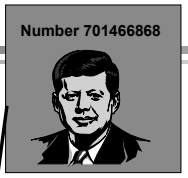
- Isto é chamado de **colisão**, porque já há outro registro válido em [3].

Quando ocorrer uma colisão, mova-se pelo vetor até encontrar uma célula vazia.

Colisões

- Isto é chamado de **colisão**, porque já há outro registro válido em [3].

Quando ocorrer uma colisão, mova-se pelo vetor até encontrar uma célula vazia.



Colisões

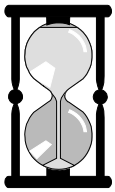
- Isto é chamado de **colisão**, porque já há outro registro válido em [3].

O novo registro vai para a célula vazia.

Questão

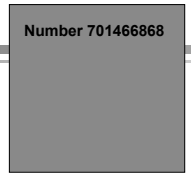
Onde você estaria nesta tabela, se não 'houver colisões?

Use o número de seu R.G. ou um outro número de sua preferência.



Procurando uma Chave

- Os dados associados a uma chave podem ser localizados fácil e rapidamente.



Procurando uma Chave

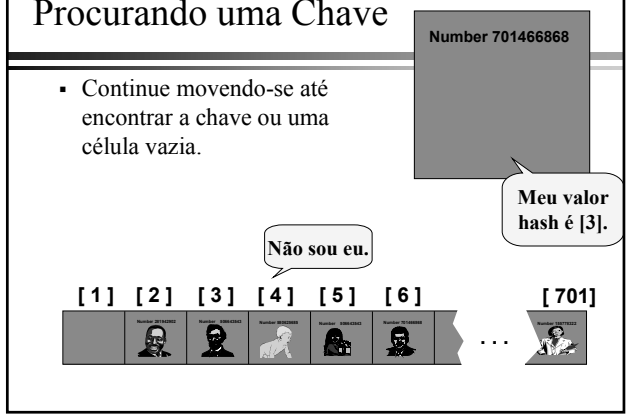
- Calcule o valor hash.
- Confira a chave do elemento do vetor com a chave procurada.

Meu valor hash é [3].

Não sou eu.

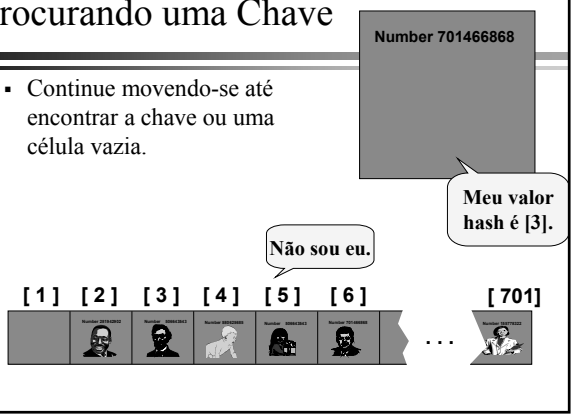
Procurando uma Chave

- Continue movendo-se até encontrar a chave ou uma célula vazia.



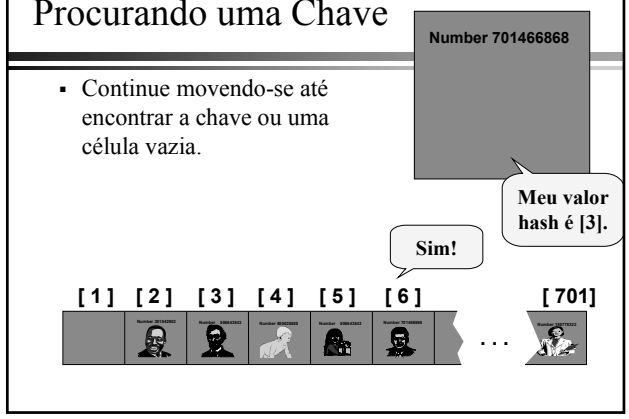
Procurando uma Chave

- Continue movendo-se até encontrar a chave ou uma célula vazia.



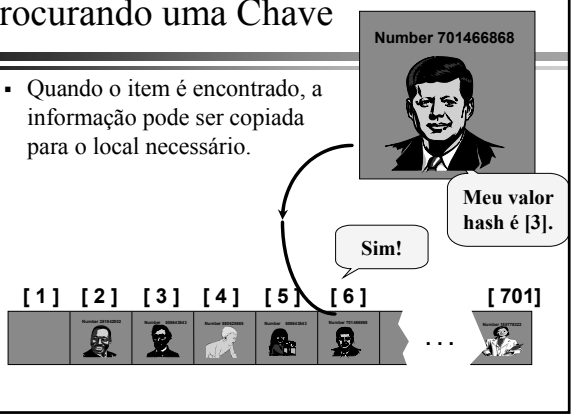
Procurando uma Chave

- Continue movendo-se até encontrar a chave ou uma célula vazia.



Procurando uma Chave

- Quando o item é encontrado, a informação pode ser copiada para o local necessário.

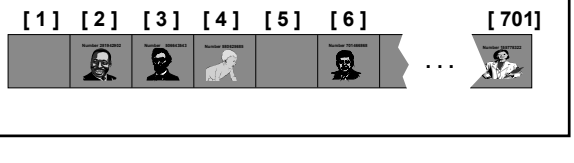


Removendo um Registro

- Registros podem também ser removidos da tabela hash.

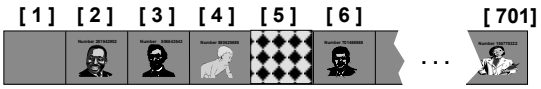
Removendo um Registro

- Registros podem também ser removidos da tabela hash.
- Mas o local não pode ser deixado como uma “célula vazia” ordinária, pois pode interferir nas buscas.



Deletando um Registro

- Registros podem também ser removidos da tabela hash.
- Mas o local não pode ser deixado como uma "célula vazia" ordinária, pois pode interferir nas buscas.
- O local deve ser marcado de alguma maneira especial para que na busca possa-se saber que havia algo lá.



Pseudo-Código

```
int Hash-Insert(T,k)
// pré: tabela hash T[1..P], chave de busca k
// pós: insere k em T, retornando posição de inserção
  // H(·,·) função de mapeamento
  i = 0;
  do
  { h = H(k,i);
    if (T[h] está livre)
    { T[h] = k;
      return h;
    }
    else
      i = i + 1;
  } while (i != P);
  cerr << "Error: hash table overflow";
```

Pseudo-Código

```
int Hash-Search(T,k)
// pré: tabela hash T[1..P], chave de busca k
// pós: retorna posição onde k foi achada ou zero
  c.c.
  // H(·,·) função de mapeamento
  i = 0;
  do
  { h = H(k,i);
    if (T[h].key == k.key)
      return h;
    else
      i = i + 1;
  } while (i != P && T[h] não está livre);
  return 0;    // não encontrado
```

Fator de Carga

- Usualmente indicado por α (alpha)
- Definição: O número de elementos ocupados em uma tabela hash (**n**) dividido pelo número total de elementos disponíveis (**P**)
- Quanto maior o fator de carga, mais lento é o processo de recuperação
- Com endereçamento aberto, $0 <= \alpha <= 1$
- Com endereçamento encadeado, freqüentemente $\alpha > 1$

Funções de Mapeamento

- Uma função hash de boa qualidade satisfaz (aproximadamente) à hipótese do hash uniforme simples: cada chave tem igual probabilidade de efetuar o mapeamento para qualquer uma das **P** posições da tabela, não importando a posição para onde foi feito o hash de qualquer outra chave
- Todavia, é raro conhecer a distribuição de probabilidade segundo a qual as chaves são obtidas
- Na prática podem ser usadas heurísticas para criar uma função hash que provavelmente terá um bom desempenho
- Endereçamento
  - Aberto: método de solucionar colisões no qual todos os elementos são armazenados dentro da própria tabela hash
  - Encadeado: colisões são mantidas em uma estrutura de dados separada (por exemplo, uma lista linear)

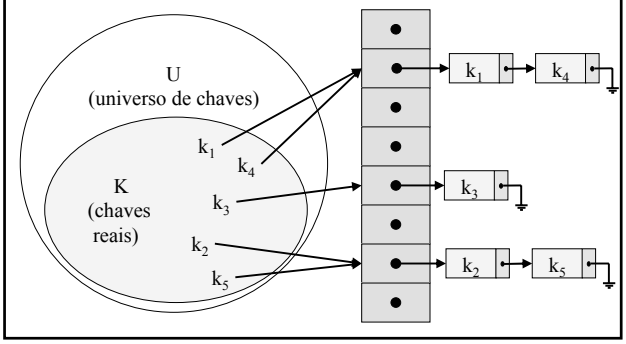
Funções de Mapeamento Endereçamento Aberto

- P é um número primo, não próximo a uma potência de 2
- Mapeamento simples
  - $H(k) = k \% P$ (intervalo 0 até P-1)
  - $H(k) = k \% P + 1$ (intervalo 1 até P)
- Sondagem Linear ($0 <= i <= P-1$)
  - $H(k,i) = (H(k) + i) \% P$
- Sondagem Quadrática
  - $H(k,i) = (H(k) + i^2) \% P$
  - $H(k,i) = (H(k) + c_1*i + c_2*i^2) \% P$ (c_1, c_2 constantes)
- Hash duplo
  - $H(k,i) = (H_1(k) + i*H_2(k)) \% P$

Endereçamento Aberto

- Vimos que $\alpha = n/P$, e $0 <= \alpha <= 1$ neste caso
- Quando **n** e **P** tendem a infinito, o número médio de sondagens em uma busca malsucedida é no máximo $1/(1-\alpha)$, para hash uniforme
- Idem para inserção

Endereçamento Encadeado



Endereçamento Encadeado

- No encadeamento, todos os elementos que efetuam hash para um mesmo valor são colocados em uma lista linear
- A posição T[**h**] contém um ponteiro para o início da lista de todos os elementos armazenados que efetuam hash para **h**
- Se não houver nenhum desses elementos, a posição T[**h**] conterá NULL (ou NIL ou qualquer valor que indique término da lista)

Endereçamento Encadeado

- Tempos no pior caso
  - Inserção: O(1)
  - Busca: tempo proporcional ao tamanho da lista
  - Remoção: O(1) se as listas forem duplamente encadeadas; se não for, tempo proporcional ao tamanho da lista
- Busca no caso médio: $O(1+\alpha)$, sob a hipótese de hash uniforme simples

Interpretação de Chaves como Números Naturais

- A maioria das funções hash assume que o universo de chaves é o conjunto **N** = {0, 1, 2, ...} de números naturais
- Se as chaves não são números naturais, deve-se encontrar um modo de interpretá-las como números naturais em notação de base apropriada
- Por exemplo "pt" poderia ser interpretado como o par de inteiros (112, 116), pois 'p'=112 e 't'=116 no conjunto de caracteres ASCII
- Expresso como um inteiro de base 128, "pt" se torna (112*128)+116 = 14452

Exemplo de Transformação de Chave String para Inteiro

```
int Transform(string s)
// pré: s chave a ser transformada
// pós: retorna s convertida para um número
//      natural
// BASE = 128 (ASCII) ou 256 (ASCII Estendido)

  k = 0;
  for(i=0; i<s.length(); i++)
    k = (k * BASE + s[i]) % P;
  return k;
```

Especificação para uma Tabela Hash

Operações:

- Criação
- Destruição
- Status
- Operações Básicas
- Outras Operações

Operações de Criação/Destruição

- Criação
  - `void Create(HashTable &T)`
  - pré-condição: nenhuma
  - pós-condição: tabela hash T é criada e iniciada como vazia
- Destruição
  - `void Destroy(HashTable &T)`
  - pré-condição: tabela hash T já tenha sido criada
  - pós-condição: tabela T é destruída

Operações de Status

- Vazia
  - `bool Empty(HashTable &T)`
  - pré-condição: tabela hash já tenha sido criada
  - pós-condição: função retorna true se a tabela hash está vazia; false caso contrário
- Cheia
  - `bool Full(HashTable &T)`
  - pré-condição: tabela hash já tenha sido criada
  - pós-condição: função retorna true se a tabela hash está cheia; false caso contrário
- Fator de Carga
  - `float LoadFactor(HashTable &T)`
  - pré-condição: tabela hash já tenha sido criada
  - pós-condição: função retorna o fator de carga da tabela hash

Operações Básicas

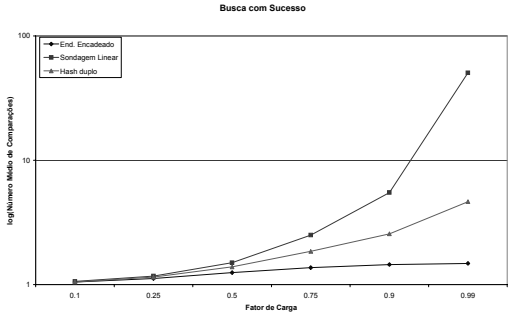
O tipo **HashEntry** depende da aplicação e pode variar desde um simples caracter ou número até uma **struct** ou **class** com muitos campos.

- Inserção
  - `int Insert(HashTable &T, HashEntry k)`
  - pré-condição: tabela T já tenha sido criada e não está cheia
  - pós-condição: O item k é inserido na tabela, retornando a posição onde k foi inserido
- Busca
  - `int Search(HashTable &T, HashEntry k)`
  - pré-condição: tabela T já tenha sido criada
  - pós-condição: retorna a posição onde k se encontra ou zero caso contrário
- Remoção
  - `bool Delete(HashTable &T, HashEntry k)`
  - pré-condição: tabela T já tenha sido criada
  - pós-condição: O item k é removido, retornando true se k estava na tabela e false caso contrário

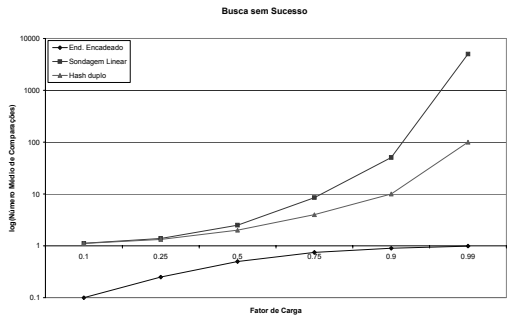
Outras Operações

- Limpeza
  - `void Clear(HashTable &T)`
  - pré-condição: tabela já tenha sido criada
  - pós-condição: todos os itens da tabela são descartados e ela torna-se uma tabela hash vazia
- Tamanho
  - `int Size(HashTable &T)`
  - pré-condição: tabela já tenha sido criada
  - pós-condição: função retorna o número de elementos em uso na tabela hash

Desempenho



Desempenho



Desempenho

- Por exemplo, se há 4096 nós em uma ABBB, toma-se em média 12.25 comparações para completar uma busca com sucesso
- Usando uma tabela hash, com $\alpha < 0.5$, serão necessárias 1.39 comparações em média

	Busca	Inserção	Remoção
Vetor Ordenado	O(log n)	O(n)	O(n)
Árvore B.B.B.	O(log n)	O(n log n)	O(log n)
Tabela Hash	O(1)	O(1)	O(1)

Sumário

- Tabelas hash armazenam uma coleção de registros com chaves
- O local de um novo registro depende do valor hash de sua chave
- Quando ocorre uma colisão, é usado o próximo local disponível
- Procurar por uma chave é, geralmente, rápido.
- Quando um item é removido, o local deve ser marcado de maneira especial, para que na busca se saiba que a célula já foi utilizada