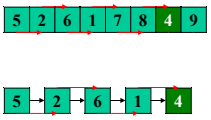
Tabelas Hash

Motivação

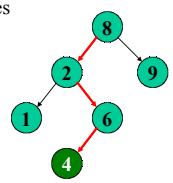
•Dada uma tabela com uma chave e vários valores por linha, quero rapidamente procurar, inserir e apagar registros baseados nas suas chaves

•Estruturas de busca sequencial/binária levam tempo até encontrar o elemento desejado.

Ex: Arrays e listas



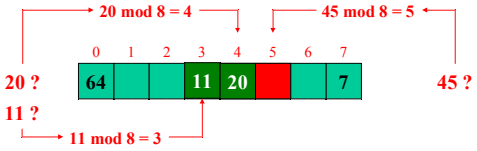
Ex: Árvores



Motivação

•Em algumas aplicações, é necessário obter o valor com poucas comparações, logo, é preciso saber a posição em que o elemento se encontra, sem precisar varrer todas as chaves.

•A estrutura com tal propriedade é chamada de **tabela *hash***.



Funções *Hashing*

- Os registros com as chaves são armazenados nessa tabela, e endereçados a partir de uma função de transformação sobre a chave de pesquisa
- A essa função dá-se o nome de Função *HASHING*
- Seja *M* o tamanho da tabela:
  - **A função de hashing mapeia as chaves de entrada em inteiros dentro do intervalo [1..*M*]**
- Formalmente:
  - **A função de hashing $h(k_j) \rightarrow [1,M]$ recebe uma chave $k_j \in \{k_0,..,k_m\}$ e retorna um número *i*, que é o índice do subconjunto $m_i \in [1,M]$ onde o elemento que possui essa chave vai ser manipulado**

Funções *Hashing*

- Método pelo qual:
  - As chaves de pesquisa são transformadas em endereços para a tabela (função de transformação);
  - Obtém-se valor do endereço da chave na tabela HASH
- Tal função deve ser fácil de se computar e fazer uma distribuição equiprovável das chaves na tabela
- Existem várias funções *Hashing*, dentre as quais:
  - Resto da Divisão
  - Meio do Quadrado
  - Método da Dobra
  - Método da Multiplicação
  - Hashing Universal

Resto da Divisão

- Forma mais simples e mais utilizada
  - Nesse tipo de função, a chave é interpretada como um valor numérico que é dividido por um valor
- O endereço de um elemento na tabela é dado simplesmente pelo resto da divisão da sua chave por *M* ($F_h(x) = x \bmod M$), onde M é o tamanho da tabela e x é um inteiro correspondendo à chave
- 0 <= F(x) <= M

Resto da Divisão

- **Ex: M=1001 e a seqüência de chaves: 1030, 839, 10054 e 2030**

Chave	Endereço
1030	29
10054	53
839	838
2030	29

Resto da Divisão – Desvantagens

- Função extremamente dependente do valor de M escolhido
  - M deve ser um número primo
  - Valores recomendáveis de M devem ser >20

Funções Hash

- Seja qual for a função, na prática existem **sinônimos** – chaves distintas que resultam em um mesmo valor de *hashing*.
- Quando duas ou mais chaves sinônimas são mapeadas para a mesma posição da tabela, diz-se que ocorre uma **colisão**.

Tratamento de Colisões

Algumas soluções conhecidas

Tabelas Hash- Colisões

- Qualquer que seja a função de transformação, existe a possibilidade de **colisões**, que devem ser resolvidas, mesmo que se obtenha uma distribuição de registros de forma uniforme;
- Tais colisões devem ser corrigidas de alguma forma;
- O ideal seria uma função HASH tal que, dada uma chave 1 <= I <= 26, a probabilidade da função me retornar a chave x seja PROB(F_h(x)= I) = 1/26, ou seja, não tenha colisões, mas tal função é difícil, se não impossível

Resto da Divisão - Colisão

- **No exemplo dado, M=1001 e a seqüência de chaves: 1030, 839, 10054 e 2031**

O valor de *h*(*k*) é o mesmo para 1030 e 2030: colisão

Chave	Endereço
1030	29
10054	53
839	838
2030	29

Tratamento de Colisões

- Alguns dos algoritmos de Tratamento de Colisões são:
  - Endereçamento Fechado
  - Endereçamento Aberto
    - Hashing Linear
    - Hashing Duplo

Endereçamento Fechado

- Também chamado de *Overflow* Progressivo Encadeado
- Algoritmo: usar uma lista encadeada para cada endereço da tabela
- Vantagem: só sinônimos são acessados em uma busca. Processo simples.
  - Desvantagens:
    - É necessário um campo extra para os ponteiros de ligação.
    - Tratamento especial das chaves: as que estão com endereço base e as que estão encadeadas

Endereçamento Fechado

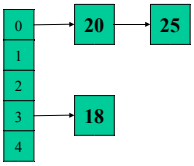
No **endereçamento fechado**, a posição de inserção não muda. Todos devem ser inseridos na mesma posição, através de uma **lista ligada** em cada uma.

20 mod 5 = 0

18 mod 5 = 3

25 mod 5 = 0

colisão com 20



Endereçamento Fechado

```
Program TabelaHash;
Const n = 50;
Type Ponteiro = ^no;
Type no = record
   item: integer;
   prox: Ponteiro;
End;
Var posicoes: array [1..n] of Integer;
```

A tabela *hash*, neste caso, contém um array de listas ligadas

```
Procedure Inicia_Hash;
Var i: integer;
Begin
   for i:=1 to n do
      posicoes[i] := nil;
End;
```

Endereçamento Fechado

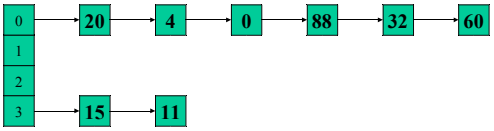
Quando uma chave for inserida, a função *hash* é aplicada, e ela é acrescentada à lista adequada

```
Procedure inserir(chave: integer);
Var i: integer; aux: ponteiro;
Begin
  i := hashCode(chave);        -> Assume-se que a função hash está implementada
  aux := lista[i];
  if aux = nil then begin      -> /*Caso em que não existe o registro na posição ainda*/
   new[lista[i]];
   lista[i] := chave;
   lista[i].^.prox := nil;
  end else begin
  while aux^.prox <> nil do
     aux := aux^.prox;
     new(aux^.prox);
     aux := aux^.prox;
     aux^.prox := nil;
     aux^.item := chave;
   End;
End;
```

Endereçamento Fechado

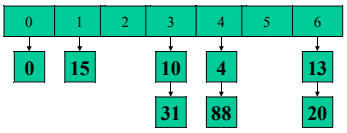
A busca é feita do mesmo modo: calcula-se o valor da função *hash* para a chave, e a busca é feita na lista correspondente.

Se o tamanho das listas variar muito, a busca pode se tornar ineficiente, pois a busca nas listas se torna seqüencial



Endereçamento Fechado

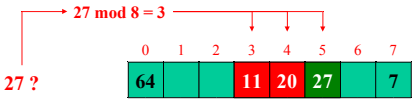
É obrigação da função HASH distribuir as chaves entre as posições de maneira **uniforme**



Hashing Linear

- Também conhecido como *Overflow* Progressivo
- Consiste em procurar a próxima posição vazia depois do endereço-base da chave
- Vantagem: simplicidade
- Desvantagem: se ocorrerem muitas colisões, pode ocorrer um *clustering* (agrupamento) de chaves em uma certa área. Isso pode fazer com que sejam necessários muitos acessos para recuperar um certo registro. O problema vai ser agravado se a densidade de ocupação para o arquivo for alta

Hashing Linear



Hashing Linear - Implementação

A tabela *hash*, neste caso, contém um array de objetos, e posições vazias são indicadas por **-1**. Neste caso, os objetos serão do tipo **Integer**:

```
Program TabelaHash;
Const n = 50;
Var posicoes: array [1..n] of Integer;
```

```
Procedure Inicia_Hash;
Var i: integer;
Begin
   for i:=1 to n do
      posicoes[i] := -1;
End;
```

Hashing Linear - Implementação

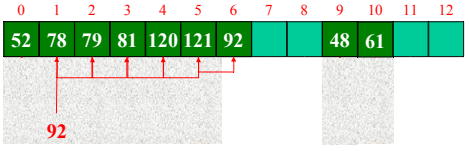
Na inserção, a função *hash* é calculada, e a posição incrementada, até que uma posição esteja livre

```
Procedure inserir(chave: integer;) {
Var i: integer;
Begin
   i := FuncaoHash(chave);
   while (posicoes[i] <> -1)
      i := (i + 1) mod n;
   posicoes[i] := chave;
End;
```

Também assume-se que a função *hash* está implementada

Hashing Linear

Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
Função: hash(k) = k mod 13
Tamanho da tabela: 13



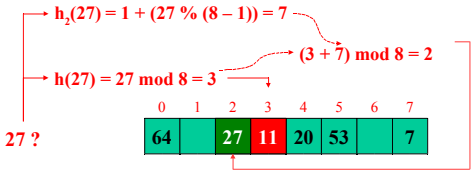
Hashing Duplo

- Também chamado de *re-hash*
- Ao invés de incrementar a posição de 1, uma função *hash* auxiliar é utilizada para calcular o incremento. Esta função também leva em conta o valor da chave.
  - Vantagem: tende a espalhar melhor as chaves pelos endereços.
  - Desvantagem: os endereços podem estar muito distantes um do outro (o princípio da localidade é violado), provocando *seekings* adicionais

Hashing Duplo

- Para o primeiro cálculo:
  - h(k) = k mod M
- Caso haja colisão, inicialmente calculamos $h_2(k)$, que pode ser definido como:
  - $h_2(k) = 1 + (k \bmod (M-1))$
- Em seguida calculamos a função re-hashing como sendo:
  - $rh(i,k) = (i + h_2(k)) \bmod M$

Hashing Duplo



Hashing Duplo - Implementação

A estrutura é semelhante ao Hashing Linear: um vetor de chaves

```
Program TabelaHash;
Const n = 50;
Var posicoes: array [1..n] of Integer;
Procedure Inicia_Hash;
Var i: integer;
Begin
   for i:=1 to n do
      posicoes[i] := -1;
End;
```

Hashing Duplo - Implementação

Na inserção, a função *hash* é calculada. Caso exista conflito, chama-se uma função *hash* alternativa até que uma posição esteja livre

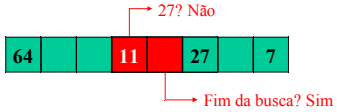
```
Procedure inserir(chave: integer);
Var i: integer;
Begin
   i := FuncaoHash(chave);
   while (posicoes[i] <> -1)
      i := FunçãoHash_Alternativa(chave);
   posicoes[i] := chave;
End;
```

Assume-se que essas funções estão implementadas

Endereçamento Aberto – Remoção

Para fazer uma busca com endereçamento aberto, basta aplicar a função *hash*, e a função de incremento até que o elemento ou uma posição vazia sejam encontrados.

Porém, quando um elemento é removido, a posição vazia pode ser encontrada antes, mesmo que o elemento pertença a tabela:



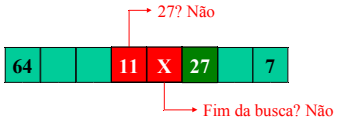
Inserção do 27

Remoção do 20

Busca pelo 27

Endereçamento Aberto: Remoção

Para contornar esta situação, mantemos um bit (ou um campo **booleano**) para indicar que um elemento foi removido daquela posição:



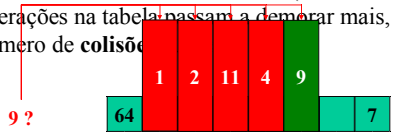
Esta posição estaria livre para uma nova **inserção**, mas não seria tratada como vazia numa **busca**.

Tabelas HASH Dinâmica

Endereçamento Aberto – Expansão

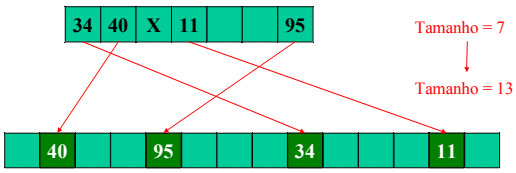
Na política de *hashing*, há que chamamos de **fator de carga** (*load factor*). Ele indica a porcentagem de células da tabela *hash* que estão ocupadas, incluindo as que foram removidas.

Quando este fator fica muito alto (ex: excede 50%), as operações na tabela passam a demorar mais, pois o número de **colisõ**



Endereçamento Aberto – Expansão

Quando isto ocorre, é necessário **expandir** o array que constitui a tabela, e reorganizar os elementos na nova tabela. Como podemos ver, o tamanho atual da tabela passa a ser um parâmetro da função *hash*.



Endereçamento Aberto – Expansão

- O problema é: Quando expandir a tabela?
- O momento de expandir a tabela pode variar
  - Quando não for possível inserir um elemento
  - Quando metade da tabela estiver ocupada
  - Quando o *load factor* atingir um valor escolhido
- A terceira opção é a mais comum, pois é um meio termo entre as outras duas.

Quando não usar Hashing?

Muitas **colisões** diminuem muito o tempo de acesso e modificação de uma tabela *hash*. Para isso é necessário escolher bem:

- a função *hash*
- o algoritmo de tratamento de colisões
- o tamanho da tabela

Quando não for possível definir parâmetros eficientes, pode ser melhor utilizar árvores balanceadas (como AVL), em vez de tabelas *hash*.

FIM