Estruturas de Dados - T.332

Capítulo 9

Hashing

ou "Tabelas de Comprovação Complementar"
ou "Técnica de Transformação de Chaves"

9.1 Recapitulação

- Até agora vimos basicamente dois tipos de estruturas de dados para o armazenamento flexível de dados: **Listas e Árvores**.
  - Cada um desses grupos possui muitas variantes.
- As Listas são simples de se implementar, mas, com um tempo médio de acesso

  T = n/2, são impraticáveis para grandes quantidades de dados.
  - Em uma lista com 100.000 dados, para recuperar 3 elementos em seqüência faremos um número esperado de 150.000 acessos a nodos.
  - Listas são ótimas porém, para pequenas quantidades de dados.
- As Árvores são estruturas mais complexas, mas que possuem um tempo médio de acesso $T= \log_G n$, onde G = grau da árvore.
  - A organização de uma árvore depende da ordem de entrada dos dados.
  - Para evitar a deterioração, há modelos de árvores que se reorganizam sozinhas. As principais são AVL e Árvore B.

9.2 Hashing: Visão Geral

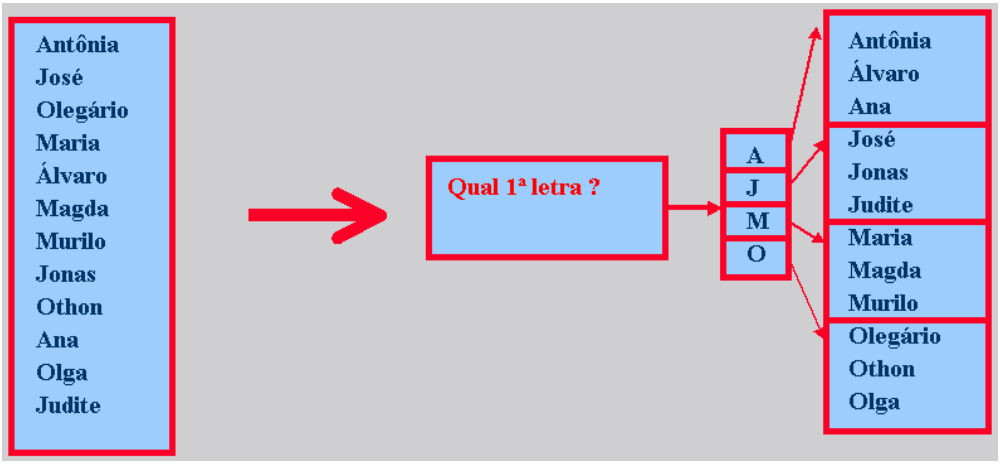
- É uma forma extremamente simples, fácil de se implementar e intuitiva de se organizar grandes quantidades de dados.
  - Permite armazenar e encontrar rapidamente dados por chave.
- Possui como idéia central a divisão de um universo de dados a ser organizado em subconjuntos mais gerenciáveis
- Possui dois conceitos centrais:
- **Tabela de Hashing:**Estrutura que permite o acesso aos subconjuntos.
- **Função de Hashing:** Função que realiza um mapeamento entre valores de chaves e entradas na tabela.
- Possui uma série de limitações em relação às árvores:
  - Não permite recuperar/imprimir todos os elementos em ordem de chave nem tampouco outras operações que exijam seqüência dos dados.
  - Não permite operações do tipo recuperar o elemento com a maior ou a menor chave.

9.2.1 Hashing: Introdução

- Idéia geral: Se eu possuo um universo de dados classificáveis por chave, posso:
  - Criar um critério simples para dividir este universo em subconjuntos com base em alguma qualidade do domínio das chaves.
  - Saber em qual subconjunto procurar e colocar uma chave.
  - Gerenciar estes subconjuntos bem menores por algum método simples.
- Para isso eu preciso:
  - Saber quantos subconjuntos eu quero e criar uma regra de cálculo que me diga, dada uma chave, em qual subconjunto devo procurar pelos dados com esta chave ou colocar este dado, caso seja um novo elemento.

    Isto é chamado de **função de hashing**.
  - Possuir um índice que me permita encontrar o início do subconjunto certo, depois de calcular o hashing. Isto é a **tabela de hashing**.
  - Possuir uma ou um conjunto de estruturas de dados para os subconjuntos. Existem duas filosofias: **hashing fechado** ou de***endereçamento aberto*** ou o **hashing aberto** ou***encadeado***.

9.2.1.1. Exemplo



9.2.1.2 Nomenclatura

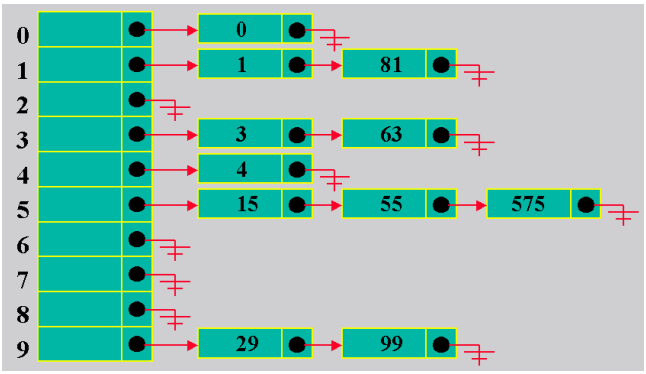
- ***n*: Tamanho do universo de dados.**
- ***b*: Número de subconjuntos em que dividimos os dados: *depósitos*.**
- ***s*: Capacidade de cada depósito (quando for limitada. Aplica-se somente a hashing fechado ou endereçamento aberto).**
- ***T*: Cardinalidade do domínio das chaves. Quantas chaves podem existir ?**
- ***n*/*T*: Densidade identificadora.**
- ***a* = *n*/(*b*.*s*)*: Densidade de carga. *b*.*s* fornece a capacidade máxima (quando existir explícitamente). *n*/(*b*.*s*) indica o fator de preenchimento. Aplica-se somente a hashing fechado (endereçamento aberto).**

9.3 Hashing Aberto ou de Encadeamento Separado (Separate Chaining Hashing)

- Forma mais intuitiva de se implementar o conceito de Hashing.
  - Intuitiva para nós, que usamos linguagens que lidam com ponteiros e estamos acostumados com a idéia. Na época do COBOL, era o modo menos comum.
- Utiliza a idéia de termos uma tabela com ***b*** entradas, cada uma como cabeça de lista para uma lista representando o conjunto b_i.
  - Calculamos a partir da chave qual entrada da tabela é a cabeça da lista que queremos.
  - Utilizamos uma técnica qualquer para pesquisa dentro de b_i. Tipicamente será a técnica de pesquisa seqüencial em lista encadeada.
  - Podemos utilizar qualquer outra coisa para representar os b_i. Uma árvore poderia ser uma opção.

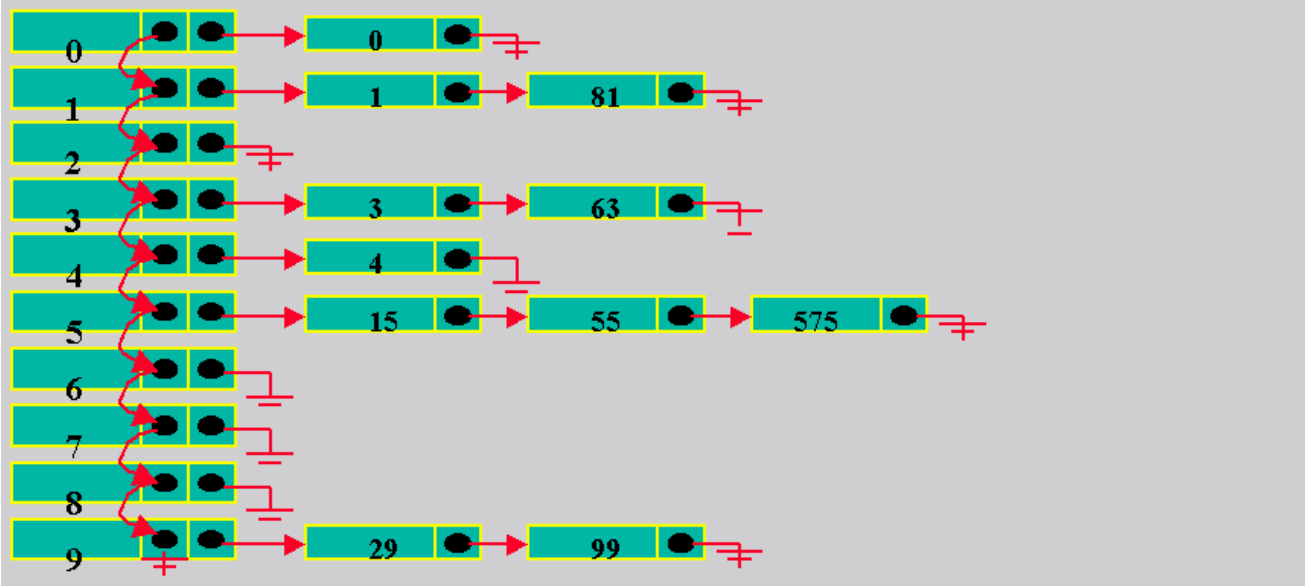
9.3.1Hashing Aberto ou de Encadeamento Separado

Implementação com a Tabela como Vetor



9.3.2 Hashing Aberto ou de Encadeamento Separado

Implementação como Multilista



9.4 Hashing Fechado ou de Endereçamento Aberto (Open Addressing Hashing)

- Utiliza somente uma estrutura de dados de tamanho

  fixo para implementar o hashing.
  - Não necessita de ponteiros para a sua implementação.
  - Forma muito utilizada "antigamente".
  - Adequada para implementação em disco (*ISAM*).
- Somente uma tabela dividida em partes iguais.
  - Áreas de tamanho fixo para cada b_i : ***Repositórios***.
  - Cada repositório é examinado seqüencialmente na busca

    por uma chave
  - Quando um repositório está cheio, há ***estouro***.
- Filosofias para tratamento de estouros
  - Utilização do primeiro espaço vazio
  - Busca quadrática

Implementação com busca seqüencial após estouro.
Suponha ***h(k)*** como ***mod(k, 10)***

- Manipulação de estouro:
  - Usamos o primeiro espaço livre.
  - No caso do 60, isto foi logo apos duas entradas em "1".
  - No caso dos valores com chave apontando

    para "3", foram incluídos dois valores antes mesmo de ser incluído algo com chave "hasheada" para "4".
- Critério de parada de busca:
  - Consideramos o arquivo inteiro como uma lista circular e

    paramos ao chegar de novo ao ponto de partida.
  - Somente aí consideramos uma chave como não existente.

9.5 Complexidade

- A ordem de complexidade de tempo de uma implementação de hashing é linear ***O(n)*** e ***T(n)*** compõe-se de três termos:

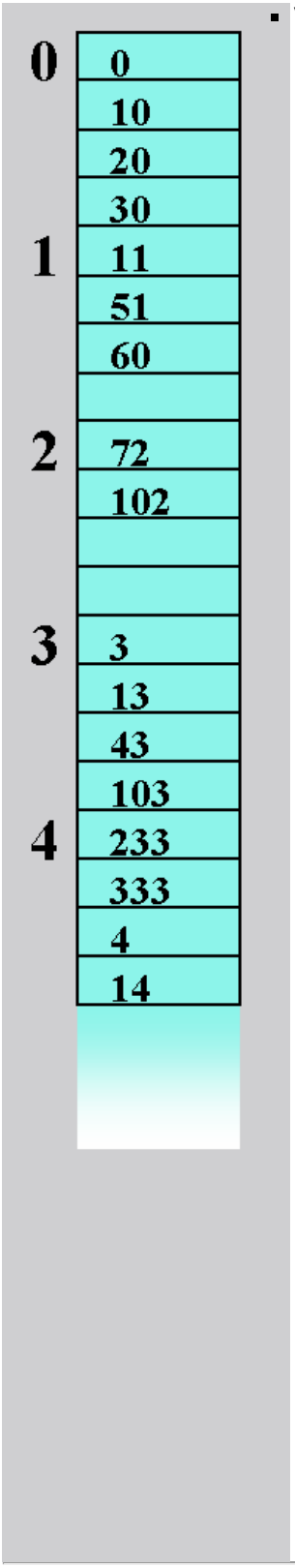
  1. O tempo para calcular a função de hashing

  2. O tempo para encontrar o início do depósito indicado pela função de hashing através da tabela de hashing

  3. O tempo para encontrar a chave procurada dentro do seu depósito.

- Estas complexidades são diferentes para as duas filosofias de implementação de hashing: aberto ou fechado.
  - ***T(n)* vai tender a ser algo em torno de *n*/*b* no caso ideal.**

- **Variam muito nos casos menos ideais.**



9.6 Vantagens

- Simplicidade
  - É muito fácil de imaginar 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.
- Aplicação imediata a arquivos
  - Os métodos de hashing, tanto de endereçamento aberto como fechado, podem ser utilizados praticamente sem nenhuma alteração em um ambiente de dados persistentes utilizando arquivos em disco.

9.7 Desvantagens

- Dependência da escolha de função de hashing
  - Para que o tempo de acesso médio ideal ***T(n) = c1 . (1/b).n + c2*** 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
  - A complexidade linear implica em um crescimento mais rápido em relação a ***n*** do que as árvores, p.ex.
- 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.

9.8 Função de 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 -> {1,..,b}***, onde ***K = {k_0,..,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_0,..,k_m}*** em ***b*** subconjuntos de mesmo tamanho.
- A probabilidade de uma chave ***k_j* pertencente a *K*** aleatória qualquer cair em um dos subconjuntos ***b_i: *i pertencente a* [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" do exemplo anterior é um exemplo de uma função ruim.
  - A letra do alfabeto com a qual um nome inicia não é distribuída uniformemememente. Quantos nomes começam com "X" ?

9.8.1 Funções de Hashing

- Para garantir a distribuição uniforme de um universo de chaves entre ***b*** conjuntos, foram desenvolvidas 4 técnicas principais.
- Lembre-se: a função de hashing ***h(k_j) -> [1,b]*** toma uma chave

  k_j Î{k_0,..,k_m} e devolve um número ***i***, que é o índice do subconjunto
  ***b_i : i Î*[1,*b*]** onde o elemento possuidor dessa chave vai ser colocado.
- 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.
- Principais Funções de Hashing:
  1. Divisão
  2. Meio do Quadrado
  3. Folding ou Desdobramento
  4. Análise de Dígitos

9.8.1.1 Divisão

- Forma mais simples e mais utilizada para função de hashing
- O endereço de um elemento na tabela de hashing é dado simplesmente pelo resto da divisão da sua chave por ***b***:

  h(k_j) = mod(k_j,b) + 1.
  - O termo "+ 1" é para numeração de ***b_i: i Î [1,b]*** a partir de 1.
- Funciona se a distribuição de valores de chave ***k_j*** for uniforme em seu domínio ***K***.
  - Se as chaves, mesmo sendo numéricas, possuem uma regra de formação que faz com que estejam irregularmente distribuídas em seu domínio, o resto da divisão também não gera distribuição uniforme. Ex.: CPF.
  - Uma ajuda sugerida é a utilização somente de números primos de valores altos (acima de 20) como valores para ***b***.
- Para chaves alfanuméricas pode-se utilizar a soma de todos os valores numéricos dos caracteres da chave como base.
  - Antigamente utilizava-se simplesmente a representação binária do string como "numero". É ruim porque para valores de ***b*** pequenos, toda a parte "mais alta" do string é ignorada.

- Exemplo:

  Suponha b=1000 e a sequinte seqüência de chaves [*Villas et.al*.]:
  1.030, 839, 10.054 e 2030.

  Teremos a seguinte distribuição:

Chave	**Endereço**
1030	**30**
10054	**54**
839	**839**
2030	**30**

- Observe que 1030 e 2030 geram o mesmo endereço.
  - A isto chama-se colisão.

9.8.1.2 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.

9.8.1.3 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.

Folding: Shift Folding

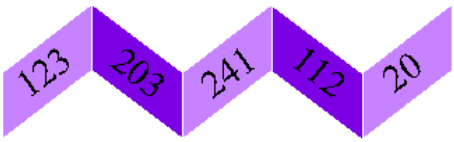
- Divido um string em pedaços, onde cada pedaço é representado numericamente e somo as partes.
- Exemplo mais simples: somar o valor ASCII de todos os caracteres.
- O resultado uso diretamente ou como chave para uma ***h´(k)***
- Exemplo:
- Suponha que os valores ASCII de um string sejam os seguintes:

  123, 203, 241, 112 e 20
- O folding será:

123
203
241
112
20
699

Folding: Limit Folding

- Uso a idéia da 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

9.8.1.4 Análise de Dígitos

- Pode ser utilizado quando eu conheço as distribuições dos dígitos ou caracteres das chaves
- É útil quando temos um domínio com poucas chaves ou com chaves com regra de formação bem conhecida.
- Escolhemos uma parte da chave para cálculo do Hashing
- Esta parte deverá ter uma distribuição conhecida
- Esta distribuição deverá ser a mais uniforme possível.
- Exemplo:
- Sabemos que o DDD e os 3 primeiros dígitos de um número telefônico não são distribuídos de forma uniforme: *não servem*
  - Diferentes estados possuem número diferente de telefones e a maioria das cidades começa seus números com X22.
- Podemos supor que os 4 últimos dígitos de um número telefônico são distribuídos mais ou menos uniformemente: *boa opção*.

9.9 Exercícios