Algoritmos e Estruturas de Dados II

Prof. Debora Medeiros

Hashing

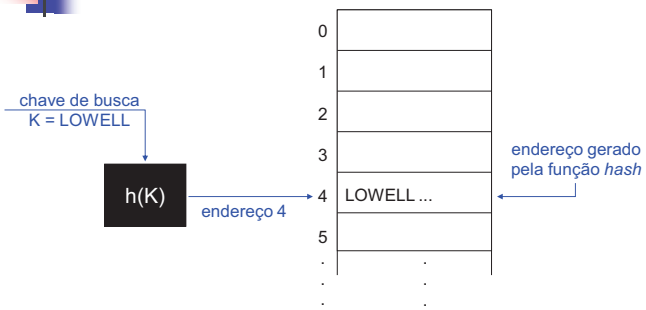
Adaptado dos Originais de:

Maria Cristina F. de Oliveira
Cristina Ciferri

Hashing convencional...

- Revisão...

Hashing



espaço de endereçamento: registros de tamanho fixo

Exemplo de espalhamento

TABLE 10.1 A simple hashing scheme

Name	ASCII Code for First Two Letters	Product	Home Address
BALL	66 65	66 × 65 = 4,290	290
LOWELL	76 79	76 × 79 = 6,004	004
TREE	84 82	84 × 82 = 6,888	888

Hashing

- Função *hash*
  - caixa preta que produz um endereço toda vez que uma chave de busca é passada como parâmetro
- Endereço resultante
  - usado para armazenamento e recuperação de registros no arquivo de dados
- Nomenclatura
  - h(K) → endereço
    - K: chave de busca

Hashing versus Indexação

- Semelhança
  - ambos envolvem associação de uma chave de busca a um endereço de registro
- Diferença (*hashing*)
  - endereço gerado é aleatório
    - não existe relação óbvia entre a chave e a localização do registro no arquivo de dados
  - duas chaves diferentes podem ser transformadas para o mesmo endereço — colisão

Exemplo de Colisão

nome	código ASC II 1ª e 2ª letras		produto	endereço gerado
BALL	66	65	66 x 65 = 4.290	290
LOWELL	76	79	76 x 79 = 6.004	004
TREE	84	82	84 x 82 = 6.888	888
OLIVER				

Exemplo de Colisão

nome	código ASC II 1ª e 2ª letras		produto	endereço gerado
BALL	66	65	66 x 65 = 4.290	290
LOWELL	76	79	76 x 79 = 6.004	004
TREE	84	82	84 x 82 = 6.888	888
OLIVER	79	76	79 x 76 = 6.004	004

chaves sinônimas: LOWELL e OLIVER

Colisão: Solução 1

- Encontrar um algoritmo de *hashing* perfeito que não produza colisões
- Cenário de uso
  - conjunto de dados pequenos e estáveis
- Limitação
  - abordagem não indicada para determinadas configurações de número de chaves e de dinâmica dos dados

perfect hashing algorithm

Colisão: Solução 2

- Encontrar um algoritmo de *hashing* que produza poucas colisões
- Objetivo
  - evitar o agrupamento de registros em certos endereços
- Funcionalidade
  - espalhar os registros aleatoriamente no espaço disponível para armazenamento
  - distribuir o mais uniformemente possível

Colisão: Solução 3

- Ajustar a forma de armazenamento dos registros
- Possibilidade 1: usar memória extra
  - aumentar o espaço de endereçamento, para um mesmo conjunto de registros
  - cenário de uso
    - poucos registros para serem distribuídos entre muitos endereços

Colisão: Solução 3

- Possibilidade 1: uso de memória extra
  - complexidade de espaço
    - perda de espaço de armazenamento
- Exemplo
  - registros: 75
  - espaço de endereçamento: 1.000
  - usado = 7,5%
  - não usado = 92,5%

Colisão: Solução 3

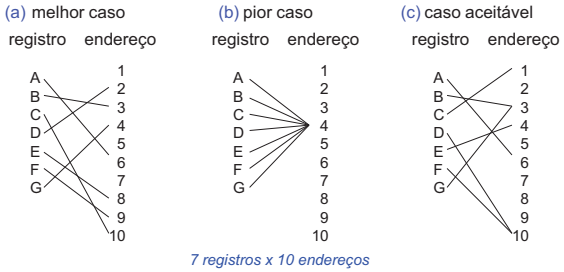
- Possibilidade 2: armazenar mais de um registro em um único endereço
  - uso de *buckets*
    - cada endereço é suficientemente grande para armazenar diversos registros
  - exemplo
    - registros de 80 bytes
    - *bucket* de 512 bytes

cada endereço pode armazenar até 6 registros!

  - complexidade de espaço
    - perda de espaço para registros sem sinônimos

Distribuição de Registros

- Como uma função *hash* distribui (espalha) os registros no espaço de endereços?



7 registros x 10 endereços

Distribuição Uniforme (a)

- Registros espalhados uniformemente entre os endereços
- Características
  - sem colisão
  - muito difícil de ser obtida

Distribuição Aleatória (c)

- Os registros espalhados no espaço de endereços com algumas colisões
- Propriedades (função randômica)
  - para uma certa chave, todos os endereços possuem a mesma probabilidade de serem escolhidos
  - a probabilidade de um endereço ser escolhido por uma outra chave não varia em função deste endereço já ter sido escolhido
  - na geração de um grande número de endereços, alguns endereços são gerados mais freqüentemente que outros

Outros Métodos de *Hash*

- Examinar as chaves em busca de um padrão
- Segmentar a chave em diversos pedaços e depois fundir os pedaços
- Dividir a chave por um número
- Elevar a chave ao quadrado e pegar o meio
- Transformar a base

Organização de índices hashing

- único arquivo
  - Os dados e o índice *hashing* ficam no mesmo arquivo
- dois arquivos
  - Os dados ficam em um arquivo e o índice hashing das chaves fica em outro

Categorias de Hashing

- Hashing estático: garante acesso O(1), para arquivos estáticos
- Hashing dinâmico: extensão do *hashing* estático para tratar arquivos dinâmicos

Hashing Extensível

- Espalhamento convencional: pouco adequado a arquivos dinâmicos, que crescem e diminuem com o tempo
- **Espalhamento Extensível** (*Extendible Hashing*): permite um auto-ajuste do espaço de endereçamento do espalhamento
- Idéia chave é combinar o espalhamento convencional com uma técnica de recuperação de informações denominada **trie**

Trie

- também conhecida como *radix searching tree*, ou árvore de busca digital
- árvore de busca na qual o fator de sub-divisão, ou número máximo de filhos por nó, é igual ao número de símbolos do alfabeto que compõe as chaves
- boa opção para manter chaves grandes e de tamanho variável...
- origem do nome: palavra reTRIEval

Exemplo Tries

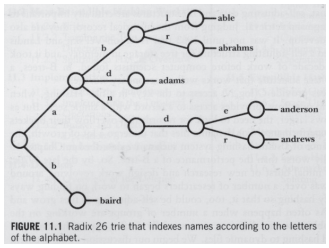


FIGURE 11.1 Radix 26 trie that indexes names according to the letters of the alphabet.

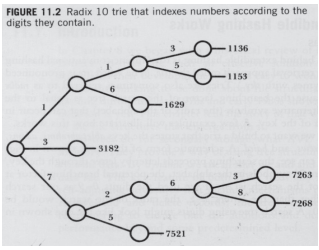


FIGURE 11.2 Radix 10 trie that indexes numbers according to the digits they contain.

Tries e espalhamento extensível

- Espalhamento extensível: tries de ordem 2
- Tabela de espalhamento indexa um conjunto de cestos (*buckets*)
- Conjunto de chaves (ou registros) armazenadas em cestos
- Busca por chave: análise bit-a-bit do valor de *key* ou do valor de *h(key)* permite localizar o seu cesto

Tries e espalhamento extensível

- Nível dos nós folha: cestos contendo várias chaves (ou registros)

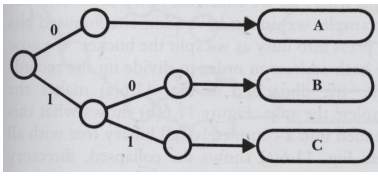


FIGURE 11.3 Radix 2 trie that provides an index to buckets.

Bucket (Cesto)

- Segmento físico útil de armazenamento externo
  - página, trilha ou segmento de trilha

Como representar a *trie*?

- Se for mantida como uma árvore, são necessárias várias comparações para descer ao longo de sua estrutura
- Solução mais eficiente: representá-la como um diretório de endereços

Transformando uma trie em um diretório

- A trie deve ser uma árvore binária completa
- Se não for, pode ser extendida

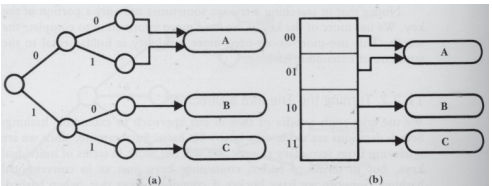


FIGURE 11.4 The trie from Fig. 11.3 transformed first into a complete binary tree, and then flattened into a directory to the buckets.

Transformando a trie em um diretório

- Uma vez "completa", trie pode ser representada por um vetor
- O vetor fornece acesso direto aos endereços
- Exemplo: endereço começando com os bits 10 pode ser encontrado a partir de um ponteiro na posição 10_2 do diretório

Diretório

- Profundidade do cesto: indicação do número de bits da chave necessários para determinar quais registros ele contém
- Informação mantida junto ao cesto
- Ex.
  - Cesto A: profundidade 1
  - Cestos B e C: profundidade 2

Diretório

- Inicialmente, a profundidade é a mesma para todos os cestos, e define a profundidade do diretório

Subdivisão para tratar overflow

- Se um registro precisa ser inserido e não há espaço no cesto associado ao seu endereço base, então o cesto é sub-dividido (*spliting*). Isso é feito adicionando mais um bit aos endereços
- se o novo espaço de endereçamento já estava previsto no diretório, nenhuma alteração é necessária
- senão, é necessário dobrar o espaço de endereçamento do diretório para acomodar o novo bit

Subdivisão para tratar overflow

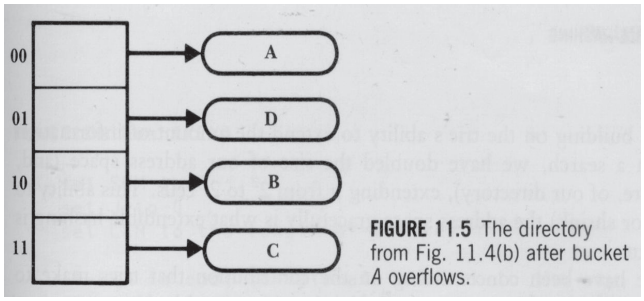
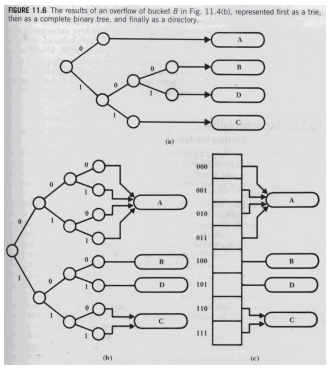


FIGURE 11.5 The directory from Fig. 11.4(b) after bucket *A* overflows.

Subdivisão para tratar overflow



Inserção de chave

- Seja *d* a profundidade do índice, dada pela maior profundidade dos cestos
- Localiza chave no diretório: seja *i* a profundidade do seu cesto
- Se a inserção da chave provoca a sub-divisão do cesto, existem 2 casos a considerar: i < d e i = d

Inserção de chave

- Se i < d
  - Remaneja os registros entre os 2 cestos
  - Insere nova chave no cesto adequado
  - Altera a profundidade de ambos os cestos
- Se i = d
  - é necessário dobrar o tamanho do diretório
  - Profundidade do índice passa a ser d + 1, assim como a dos cestos envolvidos na inserção
  - antigo conteúdo de todas as posições do índice copiado para o novo índice

Eliminação de chave

- Localiza chave no diretório
- Se encontrada, elimina a chave do seu cesto
- Verifica-se se o cesto possui um "buddy bucket"
  - Um par de cestos "buddy" é formado por dois cestos que são descendentes imediatos do mesmo nó na trie
- Se o "buddy bucket" existe, então verifica se é possível unir os cestos "buddy"
- Verifica se é possível diminuir (colapsar) o tamanho do diretório

Par de cestos "buddy"

- Se a profundidade do diretório é 0, então não existem "buddy"
- Se a profundidade do cesto for menor que a profundidade do diretório, tal cesto não tem "buddy"
- Caso o "buddy" exista, pode-se determinar o endereço do cesto "buddy" usando o cesto atual

Colapso de diretórios

- Se um par de cestos "buddy" é colapsado, pode acontecer que todo cesto tenha, no mínimo, um par de endereços referenciando-o
- Neste caso, o diretório pode ser colapsado, e seu tamanho reduzido pela metade

Bibliografia

- **M. J. Folk and B. Zoellick, *File Structures: A Conceptual Toolkit*, Addison Wesley, 1987.**