PESQUISA DE DADOS

PESQUISA

A pesquisa de informações em uma estrutura de dados é uma das operações mais freqüentemente realizadas nos sistemas de computação.

Um método de pesquisa deve se basear na forma de representação dos dados e na organização das informações na estrutura de dados. A escolha de um método de pesquisa é fator determinante no desempenho de um programa.

A avaliação do desempenho do método de pesquisa deve considerar o número de comparações feitas para encontrar a informação, singularizada pela chave de pesquisa, na estrutura de dados ou para detectar a sua inexistência, tomando como base o número médio de comparações para cada localização possível e o número de comparações para o pior caso.

PESQUISA SEQÜENCIAL

O algoritmo mais óbvio e intuitivo de pesquisa é o chamado pesquisal seqüencial ou linear e consiste em percorrer a estrutura de dados a partir da posição que identifica o seu início até o final ou até a posição onde se encontra a chave procurada.

Considerando o vetor unidimensional A [1 .. M] e *k* a chave de pesquisa,

```
INÍCIO
   i ← 1
   ENQUANTO  i < M  E  A [i].chave ≠ k  FAÇA
      i ← i + 1
   FIM ENQUANTO
   SE  A [i].chave = k  ENTÃO
      "SUCESSO"
   SENÃO
      "FRACASSO"
   FIM SE
FIM
```

O número mínimo de comparações em uma pesquisa bem sucedida é 1 e o máximo M. Para o cálculo do número médio de comparações, o raciocínio é o seguinte : supondo que todas as chaves ocorram com a mesma freqüência na estrutura de dados, se o registro com a chave procurada estiver na primeira posição (A [1]), uma única comparação é feita; se estiver na segunda, serão duas comparações. Continuando o raciocínio até a M-ésima posição, o número médio de comparações é dado por

$$\overline{NC} = \frac{1+2+...+M}{M} = \frac{\frac{M(M+1)}{2}}{M} = \frac{M+1}{2}$$

Nas pesquisas sem sucesso, no entanto, a estrutura é totalmente percorrida e seu número máximo, mínimo e médio de comparações é M.

Apesar de sua simplicidade, este algoritmo pode ser acelerado às custas do acréscimo de um registro fictício A [M + 1]. No início do algoritmo A [M + 1].chave ← *k* e assim um teste no loop de pesquisa pode ser eliminado.

PESQUISA BINÁRIA

Este método de pesquisa pode ser aplicado na busca de uma chave em uma estrutura seqüencial ordenada e consiste na obtenção da chave do meio da estrutura e na comparação com a chave procurada. O resultado dessa comparação indica a ocorrência da chave ou as posições da estrutura de dados que deverá ser pesquisada.

Para uma estrutura ordenada crescentemente :

- A [meio].chave = *k*, o elemento pesquisado foi encontrado;
- *k* < A [meio].chave, a pesquisa é restringida da posição 1 a (meio – 1);
- *k* > A [meio].chave, a pesquisa é restringida da posição (meio + 1) a M.

Aplicando repetidamente este procedimento para a parte relevante da estrutura, chega-se a uma subestrutura com apenas um elemento e a pesquisa é, então, trivial.

O número máximo de comparações para este método é dado por $\log_2 M + 1$.

PESQUISA POR INTERPOLAÇÃO

A pesquisa por interpolação procura posições aproximadas da chave de pesquisa em uma estrutura de dados estática unidimensional ordenada.

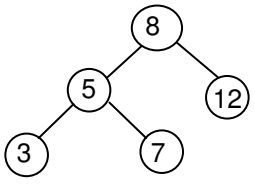
Esta técnica consiste em calcular a posição aproximada da chave de tal forma que a distância entre a menor chave e a chave de pesquisa seja proporcional à distância entre a menor chave e a maior chave do intervalo.

$$\frac{\text{índice chave pesquisada - índice chave mínima}}{\text{índice chave máxima - índice chave mínima}} = \frac{\text{chave pesquisada - chave mínima}}{\text{chave máxima - chave mínima}}$$

O desempenho do algoritmo depende da distribuição das chaves na estrutura de dados e será melhor quando a distribuição for uniforme, podendo ser, ainda, normal ou exponencial.

ÁRVORE BINÁRIA DE BUSCA

Em uma árvore binária de busca todo nodo da árvore possui chave maior que qualquer nodo da subárvore a sua esquerda e menor que qualquer nodo da subárvore a sua direita.



O pior caso de uma pesquisa em árvore binária de busca é equivalente a uma pesquisa linear e o melhor é equivalente a uma pesquisa binária.

HASHING

Um dos maiores problemas encontrados quando se estuda a alocação de estruturas de dados é o tempo de resposta da pesquisa de uma chave em um conjunto de elementos. Em se tratando de tabelas (associação = chave + informação) esse tempo de resposta corresponde ao tempo de recuperação de um determinado elemento.

O tempo requerido para pesquisas nos métodos estudados cresce com M, o tamanho da estrutura. No melhor desses métodos, a pesquisa binária, o tempo cresce com $\log_2 M$ e este é o melhor resultado possível para pesquisas através de comparações [KNUTH, 1973]. A pesquisa seqüencial oferece, no entanto, grandes vantagens sobre as demais técnicas quando existem poucos itens a examinar.

A idéia do *hashing* ou *espalhamento* é bastante simples. Projetar uma estrutura de dados para armazenar dados com chaves únicas numeradas de 1 a *n* é fácil : os dados podem ser representados em um vetor unidimensional de tamanho *n*, tal que a chave *i* é armazenada na localização *i*. Qualquer chave pode, então, ser acessada imediatamente. Se existirem *n* chaves únicas numeradas de 1 a 2*n*, por exemplo, então é ainda conveniente armazená-las em um vetor unidimensional de tamanho 2*n*, mesmo que a utilização das posições da estrutura de dados seja somente 50%. O acesso é tão eficiente que normalmente compensa o espaço desperdiçado. Entretanto, se as chaves forem inteiros, numerados de 1 a M, onde M corresponde ao maior inteiro que pode ser representado em uma linguagem de programação, não seria inteligente, e até impossível, alocar espaço de tamanho M. Por exemplo, se existirem 250 estudantes identificados pelo CPF, a alocação de um vetor unidimensional com 1 bilhão de posições desperdiçaria muita memória (há 1 bilhão de combinações para o CPF). Ao invés disso poderia-se utilizar os últimos três dígitos dos números e nesse caso seria necessário apenas um vetor de 1000 posições. No entanto, podem existir muitos estudantes com os mesmos últimos três dígitos; de fato, com 250 estudantes, a probabilidade de que isso ocorra é bastante alta. Poderia-se, então, utilizar os últimos quatro dígitos ou os três últimos dígitos e a primeira letra do nome do estudante para minimizar duplicidades. Entretanto, o uso de mais dígitos implica em uma tabela de maior tamanho e de menor utilização.

O objetivo da técnica de espalhamento é dividir os *n* elementos da tabela em M grupos, de tamanho médio *n* / M e pesquisar seqüencialmente um destes grupos. Deste modo, torna-se necessário um dispositivo que examine a chave e descubra a qual grupo ela tem possibilidade de pertencer para que se possa realizar a pesquisa através de comparações. A aplicação de uma função *hash* sobre o conjunto de chaves fornece o mapeamento das *n* chaves numeradas de acordo com algum critério, para o intervalo de 1 a M, permitindo, então, que seja feito o armazenamento de qualquer chave na estrutura de dados de tamanho M.

função *hash* (chave) $\rightarrow$ índice

Os valores retornados pela função *hash* são inteiros entre 1 e M que podem ser utilizados como índices de uma tabela auxiliar de M elementos. Os últimos três dígitos de um inteiro pode ser tal função.

Uma propriedade da função *hash* a ser salientada é que ela deve produzir o mesmo resultado para uma mesma chave.

O mecanismo da função *hash* é variável e depende do tipo de chave. Para 250 estudantes numerados de 1 a 250, a função *hash* corresponderia à identidade, ou seja, o mapeamento seria 1 : 1 e teria-se grupos formados por apenas um elemento. Para o mesmo número de estudantes, identificados pelo CPF, a função *hash* seria um pouco mais elaborada.

Considerando dois elementos por grupo, em média, a tabela de espalhamento terá 125 posições e poderá ser utilizada para indicar o início de listas encadeadas em memória que conteriam a tabela principal. Cada grupo de registros estaria, então, em uma única lista linear, de tamanho médio igual a dois. Uma possível função que, aplicada à chave, retorne um inteiro entre 1 e 125 é o resto da divisão da chave por 125, acrescido de 1. Assim, a chave 123456789 teria o endereço 40 na tabela de espalhamento (39 + 1, sendo 39 o resto da divisão por 125); à chave 917654321 corresponderia à posição 72.

A garantia de encontrar a chave de pesquisa está no fato de que a técnica empregada para gerar a tabela de espalhamento ser a mesma utilizada para realizar a busca.

A Figura 1 ilustra a montagem de uma tabela com 10 entradas utilizando uma tabela de espalhamento com 5 posições. O índice gerado pela função *hash* é chamado *endereço primário* e o verdadeiro endereço do registro é chamado *endereço efetivo*. Como o mapeamento é n : M, onde n > M, independente da função usada para realizar esse mapeamento, muitas chaves serão mapeadas para a mesma posição da tabela de espalhamento. Quando duas ou mais chaves receberem o mesmo mapeamento, ou seja, possuirem o mesmo endereço primário, diz-se que houve uma *colisão*. Chaves com as quais acontece isto são chamadas de *sinônimos*.

A tabela de espalhamento contém apenas apontadores; aumentando o seu tamanho, a eficiência da pesquisa melhora muito, sem grandes gastos de memória. Para os mesmos valores de entrada da Figura 1 utilizando uma tabela de espalhamento de tamanho 13 e, conseqüentemente, uma função *hash* = (chave MOD 13) + 1, obteria-se um *hashing* com o aspecto da Figura 2.

Ao custo de 8 palavras de memória, gastas com os apontadores extras, o número máximo de comparações reduziu de 3 para 2 e o número médio de comparações em pesquisas bem sucedidas de 1,6 para 1,1.

Desta forma, o desempenho da pesquisa em *hashing* depende :
i) do tamanho da tabela de espalhamento;
ii) da capacidade da função *hash* em distribuir uniformemente as chaves, minimizando colisões.

Os dois aspectos estão intimamente relacionados, pois o desempenho da função depende do tamanho da tabela de espalhamento.

Existem basicamente dois métodos para o tratamento de colisões :
i) encadeamento separado - todas as chaves que forem mapeadas para a mesma posição estarão juntas em uma lista encadeada;
ii) endereçamento aberto - ocorrendo a colisão, um novo índice é encontrado através da aplicação de uma outra função hash definida ou através da busca de um endereço livre.

O tempo gasto com pesquisas em uma tabela utilizando *hash* não depende do tamanho da tabela de espalhamento e aí reside sua grande vantagem. Não é difícil obter uma função *hash* para uma tabela de 1000 elementos que dê no máximo 3 colisões, o que é muito melhor do que o máximo de 10 comparações obtidas com pesquisa binária. Além disso, inserções são automáticas e não exigem o deslocamento de grandes massas de dados como ocorre em um vetor unidimensional ordenado, estrutura de dados da pesquisa binária.

FUNÇÃO HASH

Uma função *hash* ótima tem como característica a minimização do número de colisões e a rapidez no cálculo. A primeira característica é, de certa maneira, dependente dos dados e a segunda, dependente da máquina. Ao fazer a escolha de uma função o programador deve fazer as seguintes considerações : pode-se examinar os dados antes de escolher a função?; existem instruções de máquina que facilitam o cálculo do *hash*?; qual o tipo e o tamanho das chaves?; as chaves são tendenciosas?; pode-se especificar o tamanho da tabela?.

Métodos comumente utilizados para o cálculo de funções *hash* :

a) método randômico - a idéia é usar a chave como o número de partida, escolher o primeiro dos números gerados como endereço próprio e os outros, se necessário, para manipulação de colisões.

chave = 85
hash (chave) = 41

b) método da multiplicação - toma-se a chave e multiplica-se a chave por si mesma ou por alguma constante; do resultado, escolhe-se algumas posições e os valores ali contidos representarão o endereço próprio.

chave = 1234
$chave^2$ = 01522756
hash (chave) = 227

c) método do dobramento - uma maneira rápida de se obter um endereço de *k* dígitos de uma chave com *n* dígitos é dividir a chave em vários campos de *k* dígitos e somá-los, desprezando os “vai um”.

chave = 1234567890
n = 10
k = 4

1 2 | 3 4 5 6 | 7 8 9 0

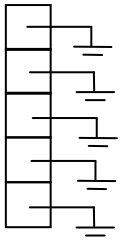
```
  3   4   5   6
+ 0 + 9 + 8 + 7
  3  13  13  13
+ 2 + 1
  5  14
```

hash (chave) = 5433

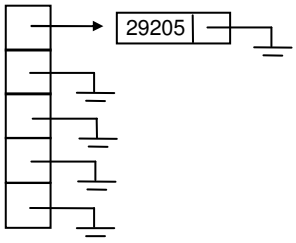
chave	*hash* (chave)
29205	1
69212	3
18250	1
69031	2
23058	4
00507	3
24179	5
39887	3
98858	4
56649	5

hash (chave) = chave MOD 5 + 1

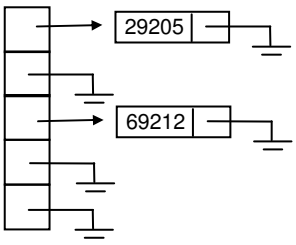
início



entra 29205



entra 69212



entra 18250

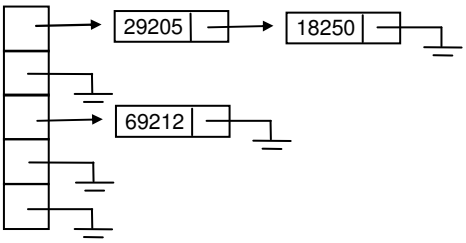
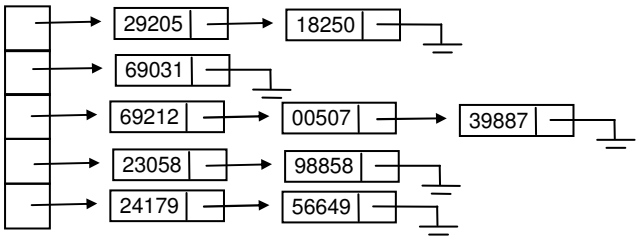


tabela completa



Sinônimos :

29205, 18250
69212, 00507, 39887
23058, 98858
24179, 56649

Total de colisões : 5
Número máximo de comparações : 3
Número médio de comparações em pesquisa bem sucedidas : 1,6

Figura 1

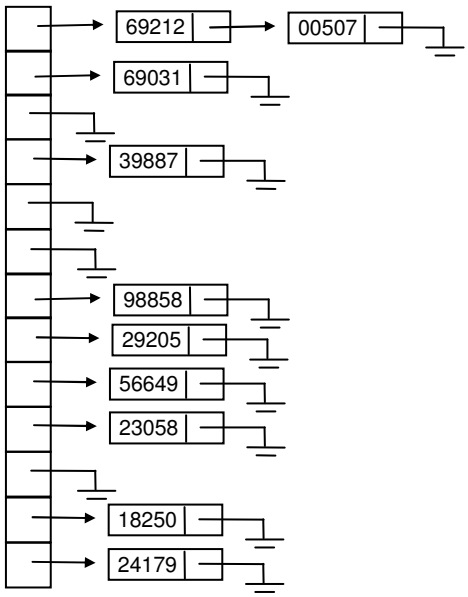

69212
00507
69031
39887
98858
29205
56649
23058
18250
24179

Figura 2

ENDEREÇAMENTO ABERTO

O tratamento de colisões pelo método de encadeamento separado utiliza listas encadeadas para armazenar chaves sinônimas. Essas listas utilizam ponteiros, o que consome espaço em memória. Com o intuito de melhor aproveitar o espaço em memória foi desenvolvido o método de *endereçamento aberto*. A idéia básica desse método é armazenar as associações contendo chaves sinônimas também na tabela; contudo, ao contrário do encadeamento separado, sem a utilização de apontadores. Nos casos de colisão, adota-se uma seqüência de posições na tabela com as quais se farão as comparações até se encontrar a chave procurada ou uma posição não ocupada. Existem várias formas para determinar a posição na tabela a ser examinada e a mais simples possui como seqüência de provas as posições consecutivas ao *endereço próprio (*aquele obtido pela aplicação da função *hash)*. Admitindo a tabela A [1..M] e supondo que *hash* (chave) = i, a chave pesquisada é comparada com A[i].chave, A[i+1].chave e assim sucessivamente. Para evitar problemas na extremidade da tabela, adota-se uma seqüência circular ... M-1, M, 1, 2, ...

Como uma pesquisa mal sucedida é detectada pelo encontro de uma posição vazia na tabela é conveniente ter sempre uma posição não ocupada na tabela de espalhamento e conhecer também o número de posições ocupadas na referida estrutura de dados.

chave	*hash* (chave)
JOÃO	6
MARIA	2
FERNANDO	6
CLÁUDIA	1
MÁRCIA	8
JORGE	7
PEDRO	2

1	CLÁUDIA
2	MARIA
3	JORGE
4	PEDRO
5	
6	JOÃO
7	FERNANDO
8	MÁRCIA

Figura 3

Um inconveniente desse método são as chamadas *pseudo-colisões* que ocorrem quando o endereço primário de uma nova associação está ocupado por uma associação que não é seu sinônimo. No exemplo da Figura 3, ocorreu uma pseudo-colisão entre Jorge e Fernando.

Esse fenômeno é, em geral, pouco prejudicial. Entretanto, tende a se agravar quando a densidade de ocupação da tabela aumenta. Uma medida da densidade de uma tabela de espalhamento é o *fator de carga*, α = N / M, onde M corresponde ao tamanho da tabela e N ao número de posições ocupadas na estrutura. Quando α se aproxima de 1, pseudo-colisões se tornam freqüentes.

A vantagem do método de endereçamento aberto está na simplicidade e o algoritmo funciona otimamente enquanto a tabela não estiver muito densa. Para um fator de carga próximo de 1 o método torna-se extremamente ineficiente e, portanto, não deve ser adotado.

```
INÍCIO          { pesquisa e inserção em endereçamento aberto }
    i ← hash ( chave )
    achouChave ← FALSO
    ENQUANTO  A [i].vazio = FALSO  E  achouChave = FALSO  FAÇA
        SE  A [i].chave = k  ENTÃO
            achouChave ← VERDADEIRO
        SENÃO
            SE  i  =  M+1  ENTÃO
                i ← 1
            SENÃO
                i ← i + 1
            FIM SE
        FIM SE
    FIM ENQUANTO
    SE  achouChave  ENTÃO
        "SUCESSO"
    SENÃO
        SE  N = M  ENTÃO        { N corresponde ao número de posições ocupadas }
            "OVERFLOW"
        SENÃO
            N ← N + 1
            A [i].vazio ← FALSO
            A [i].chave ← k
        FIM SE
    FIM SE
FIM
```