Capítulo 12

Tabelas de Dispersão

" Não existe um conjunto de regras para a concepsão de programas fáceis de ler, inteligíveis e capazes de serem completamente eficazes. O que existe são linhas gerais. Entretanto, o estilo do programador, a clareza do seu raciocínio, a sua criatividade, irão contribuir para a qualidade do seu trabalho "

- Peter J. Denning -

Sumário:

12.1- Introdução

Os algoritmos de busca que vimos no capítulo anterior, baseam-se numa pesquisa exaustiva por uma determinada chave. Agora, o nosso objectivo é tentar desenvolver um método, que prescinda dessa pesquisa e aceda ao elemento que contêm essa chave de forma directa ou quase indirecta.

Se conseguirmos definir uma função, que relacione cada chave que pretendemos inserir, à um índice no vector, então o nosso problema estará resolvido. Teremos com essa função, um método que permite determinar com antecedência à localização dos elementos que inserimos. Este princípio de organização de dados é denominado por **tabela de dispersão** e é conhecido na literatura da ciência de computação por função de Hash (**Hash function**).

12.2- Conceitos

A transformação de uma chave de busca em um endereço numa tabela (hashing) é uma técnica de pesquisa que tem por objectivo diminuir o tempo médio de busca em tabelas.

Essa técnica baseia-se em utilizar um vector, e calcular através de uma função, denominada por **função de Hash**, um índice que faz referência a posição onde essa chave deverá estar armazenada.

A situação ideal consiste em ter funções Hash injectoras, ou seja, funções, que transformem cada chave numa e apenas uma posição na tabela. Essas funções são muito raras, e isso leva-nos a pensar que poderemos ter duas ou mais chaves cuja função de Hash faz corresponder à mesma posição na tabela. Nesse caso, temos uma colisão de chaves, e necessitamos de conhecer um método para resolver o problema da colisão.

Vamos clarificar estes conceitos com um exemplo. Suponhamos que a função de Hash H(x) recebe como parâmetro um nome e calcula a posição de inserção com base na primeira letra, ou seja, 'A' será armazenado na posição 0, 'B' na posição 1, ..., etc. A seguir, descrevemos uma tabela que indica os resultados da inserção de vários nomes. Atenção, foram ignorados os acentos.

Endereço	Nome
0	Adriano
1	
2	Carlos
3	Danilo
. .	
25	Zidane

Dependendo da localização dos espaços livres, poderemos inserir novos nomes. Por exemplo, Bernardo será armazenado na posição 1, enquanto António não poderá ser inserido porque esse endereço já está ocupado. Neste caso temos uma **colisão de chaves**.

Dadas duas chaves c1 e c2 tais que c1 ≠ c2, dizemos que temos uma colisão quando H(c1) = H(c2).

Para utilizarmos de forma eficiente uma tabela de dispersão temos de resolver dois problemas: a escolha da função de transformação (função de Hash) e o tratamento das colisões.

12.3- Função de Hash

Uma função de hash caracteriza-se por transformar uma chave de pesquisa num índice de uma tabela, que denominamos por **endereço de base**. Esta função deve satisfazer as seguintes propriedades: ser fácil de calcular e distribuir as chaves de forma uniforme, minimizando desse modo, à probabilidade de colisão.

Com essas propriedades esperamos que os valores devolvidos por essa função, tenham um comportamento aparentemente aleactório, diminuindo desse modo à probabilidade de conflitos enquanto houver espaço disponível.

Existe uma grande variedade de funções de Hash, algumas são tão eficientes, que são segredos das fábricas de software. As funções de hash mais difundidas na literatura da Ciência de Computação, são o Método da Divisão e o Método da Multiplicação.

O **Método da Divisão** caracteriza-se por obter o resto da divisão do valor da chave pela dimensão da tabela.

$$h(x) = x \% M$$

Enquanto o **Método da Multiplicação**, caracteriza-se pela multiplicação do valor da chave por um certo valor real A, Knuth recomenda que se utilize como valor da A a razão áurea A =(√5−1)/2

$$h(x) = \lfloor M\ (A \cdot x\ \%1) \rfloor$$

Vimos que as chaves de pesquisa, podem ser valores numéricos ou valores alfanuméricos. Se as chaves forem valores numéricos do tipo inteiro, utilizaremos o método o resto da divisão inteira de x por M, ou seja, a função recebe como parâmetro um número inteiro e retorna um índice no intervalo [0,..,M-1]. Exemplo:

```
int hash ( int x)
 {
   return x % M;
 }
```

Mas, as chaves também podem ser valores do tipo alfanuméricos (cadeias de caracteres). Para esse caso, devemos tomar algum cuidado. Vamos clarificar essa recomendação com um exemplo. Suponhamos que pretendemos implementar uma tabela de Hash que recebe um nome e devolve um endereço na tabela com base na seguinte correspondência: a = 0, b = 1, c = 2, etc. Uma função de dispersão muito simples, consiste em utilizar a primeira letra de um nome. Logo, se M = 26, temos:

$$h(\text{anabela}) = 0\ ,\quad h(\text{leando}) = 11\ ,\ h(\text{telmo}) = 19.$$

É evidentemente esta escolha muito ingênua e pobre. Como temos maior probabilidade de termos nomes que começam com “m” ou “a” do que com “w” e “k”, a distribuição de chaves não é uniforme e não minimizar à probabilidade de colisão.

Uma ideia mais sofisticada e inteligente, consiste em interpretar o nome como um número em uma certa base e usar o método do resto da divisão. Por exemplo, podemos pensar que cada carácter é um número na base 26, então,

$$carlos = 1 \times 26^5 + 0 \times 26^4 + 17 \times 26^3 + 11 \times 26^2 + 14 \times 26^1 + 18 \times 26^0$$
$$= 24.069.362.$$

Se M for igual a 51 então: f(carlos) = 24.069.362 % 51 = 14.

Este exemplo, é uma boa escolha, e sua estratégia, consiste em tratar cada carácter como um valor inteiro, somá-los e dividir o resultado dessa soma pela dimensão da tabela.

```
int hash (char st[ ])
 {
   int k, i, h = 0;
   k = strlen(st);
   for ( i = 0 ; i < k ; i++)   h = h + i * abs(st[i]);
   return  h % M;
 }
```

Mas, pela tabela ASCII, cada caracter é um número entre 0 a 255. Então uma cadeia de caracteres pode ser interpretada como a representação de um número na base 256. Por exemplo, se a cadeia de caracteres possui dois elementos, então, o correspondente número seria determinado por:

```
st[0]*256 + st[1]
```

e uma boa função de Hash pode ser descrita pela seguinte função:

```
int hash (char st[ ])
 {
   nt i, h = st[0];
   for ( i = 0 ; v[i] != '\0' ; i++)   h = h*256 +st[i];
   return h % M;
 }
```

Para evitar que a instrução h = h*256 + st[i] provoque um **overflow** de memória e com isso gerar um valor negativo, deveremos utilizar o número primo mais próximo de 256, ou seja:

```
int hash (char st[ ])
{
    int i, h = st[0];
    for ( i = 0 ; st[i] != '\0' ; ++i)   h = (h*251 + st[i]) /M;
    return h % M;
}
```

12.4- Métodos para Determinar o Tamanho da Tabela

Para minimizar o número de colisões, e permitir que a função de hash, faça uma dispersão uniforme, a dimensão da tabela deve ser maior do que o número de elementos a inserir. Como determinar esse tamanho?

Se a dimensão da tabela for um número par, então h(x) é par todas as chaves múltiplas de dois, irão possuir o mesmo endereço de base. Como consequência, essas chaves ficarão agrupadas numa parte da tabela. Este fenómeno é responsável de diminuir drasticamente a eficiência dos algoritmos inserção e recebe o nome de **aglomeração de elementos**.

Para evitar a aglomeração de elementos e permitir que a dispersão seja uniforme, deve-se escolher para M um número primo que não esteja muito próximo de uma potência de 2.

Também aconselha-se que o **factor de carga** seja menor ou igual a 0.5. Por definição, o factor de carga é um indicador que mede a percentagem de ocupação da tabela, e é definido por:

$$\alpha = n / M$$

onde

n é o número de elementos inseridos;
M é a dimensão da tabela.

Para resolver o problema das colisões, temos dois técnicas: o **Hash fechad**o que utiliza uma estrutura de dados estática, o vector. Esta técnica é utilizada quando o número de elementos a ser armazenado for relativamente pequeno e a dimensão do vector puder ser estimada com antecedência. O **Hash aberto** que utiliza uma estrutura de dados dinâmica, a lista ligada. Está técnica é utilizado quando o número de elementos for é muito grande e não cabe na memória principal.

12.5- Endereçamento Aberto

A técnica de endereçamento aberto (**Open Adressing**) enquadra-se nas estratégias de **Hash fechado** e caracteriza-se por: ao ocorrer uma colisão, efectua-se uma pesquisa sequencial pela próxima posição livre e procede-se a inserção do registro nessa posição. Mas, para efectuar essa operação, necessitamos de definir uma estrutura de dados.

12.5.1- Estrutura de Dados

Na estratégia de Hash fechado utiliza uma estrutura de dados em armazenamento sequencial. Para a técnica de Endereço Aberto, vamos definir um vector de estruturas com M elementos. Cada elemento desse vector é uma estrutura que possui pelo menos dois campos: a chave e o campo estado. O campo chave é do tipo inteiro e contêm as chaves que pretendemos inserir, enquanto o campo estado é do tipo booleano e contêm os valores 0 e 1 que

fazem referencia a situação de LIVRE e OCUPADO. Associado a esse vector temos uma variável do tipo inteiro, denominada por nElem, que dar-nos-á em qualquer instante o número de elementos inseridos. Para além disso, vamos declarar um tipo de dados enumerado, denominado por Boolean que emula os valores lógicos da álgebra boolena.

12.5.2- Inicialização da Tabela

A operação para inicializar uma tabela de dispersão consiste em atribuir o valor zero à variável que determina o número de elementos inseridos, e o valor verdadeiro ao campo livre de todos os elementos do vector.

```
void inicializarHash ( THash *tab )
 {
   for (int i = 0 ; i <= M – 1 ; i++ ) tab->item[i].estado = LIVRE ;
       tab->nElem = 0 ;
 }
```

A operação para verificar se uma tabela de dispersão está vazia é muito simples, por esse facto será omitida.

A operação para verificar se uma tabela de dispersão está cheia, apesar de ser muito simples, iremos implementá-la.

```
Boolean  cheia ( TTable Tab )
 {
   return ( tab.nElem === M-1) ;
 }
```

12.5.3- Inserção de uma Chave

A operação para inserir uma chave só faz sentido se a tabela não estiver cheia. O algoritmo calcula através da função Hash(x) à posição de inserção na tabela. Se o registro que ocupa essa posição estiver livre, inserimos a informação, alteramos o campo estado para OCUPADO, e actualizamos o número de elementos inseridos. No caso contrário, tentamos inserir o elemento na posição Hash(x) + 1. Se esta estiver ocupada, tentamos a posição Hash(x) + 2 ,…, até encontrarmos um posição livre. Como esta estratégia utiliza a aritmética modular, o processo termina quando inserimos o elemento ou quanto percorremos a tabela na totalidade. A essa estratégia recebe o nome de **Hashing Linear**, ou de **Sondagem Linear,** e com base nessa descrição, implementamos a seguinte função:

```
void insereHashLinear( THash *tab , int x , int *nElem )
 {
    int pos;
    pos  = hash(x) ;
    while ( tab->item[pos].livre == OCUPADO ) pos = (pos+1) % M ;
    tab->item[pos].livre = OCUPADO ;
```

```
    tab->item[pos].chave = x ;
    tab->nElem++ ;
}
```

12.5.4- Remoção de uma Chave

A operação para remoção de uma chave é mais complexa. Ela só faz sentido se a tabela possuir pelo menos um elemento. O algoritmo calcula através da função Hash(x) a posição de remoção. Se o registro que ocupa essa posição estiver ocupado vamos verificar se o campo chave é igual ao valor que procuramos. Se essa comparação for verdadeira, guardamos o registro para ser devolvido como parâmetro; subtraímos uma unidade ao número de elementos inseridos, e alteramos o estado do campo estado para LIVRE. Se essa comparação for falsa, vamos tentar a remoção na posição Hash(x) + 1, se a comparação for falsa, posição Hash(x) + 2 e assim por diante. A busca termina quando encontramos a chave ou quando a tabela for percorrida na totalidade. Com base nessa descrição, implementamos a seguinte função:

```
void removerHashLinear ( THash *tab , int  x , TItem * y )
  {
    int cont = 0; pos = hash(x);
    while (cont <= M-1))
     {
       if ( (tab->item[pos].livre == OCUPADO ) &&
            (tab->item[pos].chave == x) )
           {
             *y = tab->item[pos] ;
             tab->item[pos].livre = LIVRE ;
             tab->nElem -- ;
           }
       else
           {
             pos = (pos + 1) % M ;
             cont ++ ;
           }
     }
  }
```

A estratégia de Hashing Linear é muito fácil de ser implementada, mas tem como principal deficiência o facto de concentrar as chaves numa determinada parte da tabela, tornando desse modo, as operações sobre essa estrutura de dados mais lentas. Uma alternativa consiste em utilizar o Hash duplo.

O **Hash duplo** é um dos melhores métodos para o Endereço Aberto, porque evita a concentração das chaves numa determinada parte da tabela e consiste em utilizar duas funções de Hash. A primeira H1(x) que será responsável por determinar o primeiro endereço de base na tabela, e a segunda H2(x) que determinará o processo de pesquisa pela chave.

Para além disso, o método tende a distribuir as chaves de forma uniforme. Se x e y são duas chaves diferentes, tais que H1(x) = H1(y) as duas primeiras tentativas obtidas pelo método anterior teríamos índices iguais, o que provocaria uma concentração de elementos numa parte da lista. No presente método os índices devolvidos por essa função só serão iguais se H2(x) for igual a H2(y).

Existem vários métodos para assegurar essa relação. O mais simples consiste em escolher para M um número primo e garantir que a segunda função de Hash devolva um valor inteiro k menor que M. Dessa forma M e K serão primos entre si, e a expressão (j*k) % M irá gerar números inteiros entre 0 a M -1.

Para consolidar a matéria, veremos apenas a operação para inserir uma chave nessa tabela, deixando as restantes como exercício.

```
void InsereHashDuplo ( int x, THash *tab)
 {
   int pos, desloc ;
   pos = Hash1(x);
   desloc = Hash2(x) ;
   while (tab->item[pos].livre == OCUPADO ) pos = (pos + desloc) % M ;
   tab->item[pos].livre = OCUPADO ;
   tab->item[pos].chave = x ;
   tab->nElem++ ;
 }
```

O duplo Hash espalha os elementos de forma uniforme na tabela, mas no pior caso será necessário percorrer a tabela inteira e o algoritmo continua a ter um desempenho O(N).

12.6- Encadeamento Interno

A técnica de Encadeamento Interno faz parte da estratégia de Hash Fechado e caracteriza-se por dividir a tabela de dispersão em duas partes consecutivas. Uma denominamos por **tabela primária** que varia de 0 a p e outra denominada por **tabela de secundária** ou **tabela de sinónimos** que varia de p+1 à m. As inserções são feitas na tabela primária, mas se houver alguma colisão a chave que pretendemos inserir vai ser armazenada na tabela secundária.

Para que a tabela de dispersão tenha uma distribuição uniforme, recomenda-se que o número p seja primo.

12.6.1- Estrutura de Dados

Vimos que na estratégia de Hash Fechado utiliza-se uma estrutura de dados em organização sequencial. Para a técnica de Encadeamento Interno, vamos definir um vector de estruturas com M elementos. Cada elemento desse vector é uma estrutura que possui pelo menos três campos: a chave, o campo estado e um o campo prox. O campo chave é do tipo inteiro e contêm as chaves que pretendemos inserir; o campo estado é do tipo booleano e contêm os valores 0

e 1 que fazem referencia a situação de LIVRE e OCUPADO; o campo prox é do tipo inteiro e serve para encontrar as chaves que colidiram e foram armazenadas na tabela secundária. Vamos supor que pretendemos processar 100 registros. Associado a esse vector, temos duas variáveis do tipo inteiro, uma denominada por totPrimario e outra denominada por totSecundario que dar-nos-ão em qualquer instante o número de elementos inseridos na tabela primária e na tabela secundária. Para além disso, vamos declarar um tipo de dados enumerado, denominado por Boolean que emula os valores lógicos.

12.6.2- Operações

Devido a analogia com a estrutura de dados para a estratégia de Endereço Aberto, acreditamos que o leitor não terá qualquer dificuldade em implementar a operação para inicializar uma tabela de hash. Acreditamos também que as operações para verificar se a tabela primária está cheia, a tabela secundária está cheia não constituem qualquer problema. O mesmo se passa para verificar se a tabela primária está vazia e a tabela secundária está vazia.

As restantes operações são mais complexas, mas descreveremos apenas o funcionamento da operação de inserção de uma chave. A sua implementação deixamos como exercício.

Seja x uma chave qualquer. Converta em primeiro lugar essa chave para um valor do tipo inteiro e obtenha com base na função de Hash uma posição de inserção na tabela primária. Seja r essa posição. Para inserir essa chave temos os casos possíveis:

O primeiro caso é o mais simples, ele acontece quando à posição está livre, e a sua solução é trivial. Basta inserir a chave nesse registro, alterar o estado do campo livre para FALSE e colocar no campo prox o valor zero. O valor zero no campo prox denota que essa chave não teve qualquer colisão.

Os restantes casos acontecem quando à posição está ocupada. Então, elas contêm uma chave y e temos uma colisão com a chave x. Para resolver o problema da colisão temos duas possibilidades: O campo prox contêm o valor zero ou um índice c maior do que zeros.

Se o campo prox contêm o valor zero, estamos em presença da primeira colisão. Então vamos procurar por um registro livre da tabela de secundária. Seja c um índice que faz referencia a esse registro. Inserimos nesse registro a chave, alteramos o campo livre para FALSE e colocamos no campo prox o valor zero. Para além disso, armazenamos no campo prox do registro onde houve a colisão o valor do índice c. Com essa acção criamos uma ligação entre o registro onde houve a colisão na tabela primária e o registro c da tabela de secundária. A está ligação dá-se o nome de **cadeia de colisões**.

Se o campo prox contêm um valor maior do que zeros, estamos em presença de um registro onde ocorreram várias colisões. Percorremos na tabela secundária a cadeia de colisões associada a esse registro até chegarmos à um registro cujo

campo prox é igual a 0. Repetimos para esse registro o procedimento anteriormente.

Contudo a tabela de espalhamento com tratamento por Encadeamento Interior possui o problema do falso overflow. O falso overflow consiste em tentar incluir uma chave num registro ocupado cuja cadeia de colisões ocupa a totalidade da tabela secundária. Esse registro não pode ser incluído, mas podemos ter espaço livre na tabela primária.

Uma das técnicas para resolver o problema do falso overflow, consiste em aumentar o tamanho da tabela de colisões, diminuindo desse modo o tamanho da tabela primária e, como consequência, tornando os algoritmos de manipulação da tabela de espalhamento pouco eficientes.

12.7 - Recomendações Bibliográficas

Para o leitor aprofundar os seus conhecimentos sobre métodos de ordenação interna, recomendamos os seguintes livros:

Gonnet G. H., Baeza-Yates R.;- *Handbook of Algorithms and Data Structures*, Addison Weley, Reading, Mass. 1991

Knuth D. E.;- *The Art of Computer Programming Vol 3: Sorting and Searching*, 3th Edition, Addison-Wesley, Reading, Mass, 1998.

Sedgewick R. ;- Algoritms in C, Massachusetts, Addison-Wesley, Reading, Massachusetts,1990.

Szwarcfiter L. J. , Markenson L.;- *Estruturas de Dados e seus Algoritmos*, 2ª edição, Editora LTC, Rio de Janeiro, Brasil, 1994

Wirt N;- *Algoritms and Data Structures*, oberon version: August 2004, disponível em: www.ethoberon.etz.ch/withPubli/AD.pdf

Ziviani N.;- *Projeto de Algoritmos Com implementações em Pascal e C*, 4ª Edição, Editora Pioneira, São Paulo, 1999

12.8 - Exercícios

10.9.1- Considere uma tabela de Hash de tamanho m = 1000 e a função de Hash H(x) = (x * A % 1) * M, com A=(√5 -1)/2. Calcule os valores de Hash para as seguintes chaves 61, 62, 63, 64 e 65.
10.9.2- Dada uma tabela de dispersão. Desenvolva um algoritmo recursivo com o paradigma de decrementar para conquistar, para inserir uma determinada chave nessa tabela, utilizando a técnica de hashing linear.

10.9.3- Dada uma tabela de dispersão. Desenvolva um algoritmo com a técnica de Hash duplo para remover uma chave da tabela.

10.9.4- Desenhe a tabela de Hash que resulta da inserção das chaves 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 e 5, utilizando uma função de Hash H(k) = (2k + 5)%11 e supondo que as colisões sejam tratadas com sondagem linear.

10.9.5- Dada uma tabela de dispersão. Desenvolva um algoritmo recursivo com o paradigma de Decrementar para Conquistar, para remover uma determinada chave com a técnica de Hash linear.

10.9.6- Dada uma tabela de dispersão. Desenvolva um algoritmo iterativo com a técnica Hash linear para procurar uma determinada chave.

10.9.8- Dada uma tabela de dispersão. Desenvolva um algoritmo recursivo com o paradigma de Decrementar para Conquistar para procurar uma determinada chave na tabela, utilizando a técnica de Hash linear.

10.9.8- Dada uma tabela de dispersão. Desenvolva um algoritmo com a técnica Hashing linear para alterar o conteúdo de um determinado registro.

10.9.9- Dado um conjunto de sequências não ordenadas de números naturais diferentes. Por exemplo: [2,7,8], [1,5,9], [3,4,1], [3,7,10], [7,10,3], [9,5,1] etc. Desenvolva um algoritmo que processe essas sequências, uma por uma, e retorne o número de sequências que foram processadas quando todas as 6 permutações possíveis (para um conjunto qualquer) já tiverem sido processadas durante a varredura do conjunto. Por exemplo: o algoritmo deve parar quando tiver lido as sequências [2,7,8], [1,5,9], [3,4,1], [3,7,10], [7,10,3], [9,5,1], [5,1,9], [2,17,3], [5,9,1], [9,5,1], [10,5,2], [1,9,5]. Nessa altura deve devolver o valor 12, porque a décima-segunda sequência desse conjunto, [1,9,5], completa todas as seis permutações possíveis dos números 1, 5 e 9.

10.9.10- Dado uma tabela de dispersão com encadeamento interno. Desenvolva os algoritmos de inicialização, inserção, remoção e busca

10.9.11- Suponha que na tabela de Hash o campo estado possuía os seguintes valores: LIVRE quando o registro não recebeu nenhum valor, OCUPADO quando o registro recebeu um valor e REMOVIDO quando o registro receu um valor e este foi removido. Desenvolva as seguintes operações: inserir, alterar, remover, buscar e recuperar.