Capítulo 11

Pesquisa em Memória Interna

" O estudo da ciência da computação não consegue transformar qualquer um em um excelente programador, da mesma forma que o estudo de tintas e pinceis não transforma qualquer um em um excelente pintor."

- Eric S. Raymond -

Sumário:

11.1- Conceitos

Uma das tarefas mais comuns no dia-adia é a pesquisa de informação. Pesquisar uma sequência de dados, à procura da localização de um valor específico é uma tarefa muito frequente e muito simples. Mas se os algoritmos não forem eficientes, essa tarefa torna-se muito dispendiosa em termos de processamento automático de informação.

Neste capítulo discutiremos as diferentes estratégias de efectuar essa pesquisa, e veremos os conceitos necessários, para realizarmos essa operação.

Entendemos por **tabela** um conjunto finito de elementos, denominados por registros. Em cada registro temos um campo que goza da seguinte propriedade:

se i ≠ j então campo(i) ≠ campo(j) para qualquer que seja i, j

Esse campo, denominado por **chave**, permite identificar de forma individual cada registro na tabela. Por exemplo, para o conjunto de estudantes de uma universidade, o número do estudante é uma chave desse conjunto.

Os algoritmos de pesquisa (busca) podem ser classificados como **pesquisa interna**, quando a tabela é pequena e cabe na memória principal do computador,

e **pesquisa externa** quando a tabela é muito grande e tem de ser armazenada em dispositivos de memória externa.

O clássico problema de pesquisa "**Searching"**, consiste em dado uma tabela com n elementos e um determinado valor, que denominamos por chave de pesquisa, localizar o índice da primeira ocorrência dessa chave.

Nestas notas iremos estudar alguns métodos de pesquisa interna, organizados numa tabela representada por um vector.

Na linguagem C, as funções que implementam os algoritmos de pesquisa, retornam o valor -1 se o elemento não está contido no vector, ou o índice da primeira ocorrência, no caso contrário.

11.2- Estrutura de Dados

Como no capítulo anterior, vamos utilizar um vector de estruturas com MAX elementos. Cada elemento desse vector, é uma estrutura possui os seguintes campos. Uma chave do tipo inteiro e um valor do tipo real. Associado a esse vector temos uma variável, 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 emulada os valores lógicos da álgebra booleana.

Em termos da linguagem C. Os elementos (itens) desse vector são declarados por:

```
typedef
 {
    int chave;
    float valor;
 }TItem;
```

O vector que contêm esses elementos e a variável que dar-nos-á o número de elementos inseridos no vector, pode ser declarado por:

```
typedef
 {
   TItem item[MAX];
    int nElem;
 } TTable;
```

onde

```
define MAX   1000
```

e

$0 \leq nElem \leq MAX$

Para terminar, o tipo enumerado que emula os valores lógicos pode ser declarado por:

```
typedey enum {FALSE = 0, TRUE = 1} Boolean;
```

11.3- Pesquisa Sequencial em tabelas não Ordenadas

A pesquisa sequencial *(**sequencial search**)* é a forma mais simples de pesquisar uma informação num vector. Ela consiste em percorrer o vector, elemento por elemento. Se o elemento que procuramos não está contido no vector, essa pesquisa será feita até ao fim dos elementos inseridos. Mas se o elemento está contido no vector, essa pesquisa termina quando foi encontrada a primeira ocorrência. Esta estratégia pode ser descrita pela seguinte função.

```
int pesquisaSequencial (TTable v , int x)
{
  int i;
  for (i = 0 ; i < v.nElem ; i ++)
     if ( v.item[i].chave == x )  return i;  /* Encontrou o elemento */
  return -1;                                 /* Não encontrou o elemento */
}
```

Esta função retorna o resultado que esperamos? Para responder a essa questão veremos o comportamento da função para os seguintes casos:

Se o vector estiver vazio, a função retorna o valor -1. Mas esse valor indica que o elemento que procuramos não está contido no vector, e não a existência de um vector vazio. Para contornar esse erro, basta desenvolver uma função para testar a condição de vector vazio. Se essa condição for falsa, invocar a função pesquisaSequencial(), no caso contrário emitir uma mensagem de erro.

Se o vector não estiver vazio, a função percorre todos os seus elementos até a posição que antecede o número de elementos inseridos. Se a chave for encontrada, a condição v.item[i].chave == x é verdadeira, e a função retorna essa posição. Mas se todos os elementos foram percorridos, então a chave não foi encontrada e a função retorna o valor -1.

Vejamos a seguir uma variante dessa implementação, muito elegante e muito eficiente.

```
int pesquisaSequencial (TTable v , int x)
{
   int i;
   for ( i = v.nElem ; i > = 0 && v.item[i].chave != x ; i--);
   return i;
}
```

Em contrapartida, mostramos a seguir uma função muito popular, cujo código é pouco elegante.

```
int pesquisaSequencial (TTable v , int x)
 {
    int codret = -1, i = 0;
    while (i <= v.nElem && v.item[i].chave != x ) i++;
     if (v.item[i].chave == x)  codret = i;
     return codret;
 }
```

Vejamos a seguir a versão recursiva da pesquisa sequencial com o paradigma de decrementar e conquistar. O algoritmo é extremamente simples: se o índice de pesquisa for igual ao número de elementos inseridos, então o elemento que procuramos não está no vector. Se o conteúdo da chave for igual ao elemento que procuramos então o elemento foi encontrado, no caso contrário devemos efectuar a chamada recursiva da função para o próximo elemento.

```
int pesquisaSequencialRec (TTable v , int x , int i)
 {
    if (i == v.nElem)  return -1;
    if ( v.item[i].chave == x ) return i;
    return pesquisaSequencialRec (v , x , i+1);
 }
```

O algoritmo de pesquisa sequencial só deve ser utilizado em tabelas com poucos elementos. Quando a tabela é muito grande o custo em termos de tempo de processamento é inviável. Por exemplo, suponhamos que a tabela possui 1.000.000 de elementos. Se cada repetição demorar cerca de 100 microssegundos, a algoritmo de pesquisa sequencial levará quase 2 minutos para achar uma solução se o elemento estiver no fim da tabela. Isso não é aceitável em computação

11.4- Pesquisa Sequencial com Sentinela

O algoritmo de pesquisa sequencial que vimos na secção anterior realiza duas comparações em cada iteração. Para minimizar esse número de comparações e melhorar o seu desempenho, aconselhamos a utilizar o algoritmo de pesquisa sequencial com sentinela.

A pesquisa sequencial com sentinela consiste em inserir depois da posição do último elemento um registro cuja chave é igual ao valor que procuramos. Com esse sentinela temos as garantias que o elemento que procuramos está contido no vector. Então o processo de pesquisa termina quando encontramos o elemento na posição que faz referencia ao número de elementos inseridos, e nesse caso retornamos o valor -1; ou, o elemento que procuramos está numa posição entre zero e a posição que antecede o número de elementos inseridos, e nesse caso, retornamos a sua posição desse elemento. Esta estratégia pode ser implementada pela seguinte função:

```
int pesquisaSequencialSentinela (TTable v , int x)
 {
    int i;
    v.item[v.nElem].chave = x;
    for (i = 0 ; v.item[i].chave != x  ; i++);
     if ( i == v.nElem) return -1;
    return i;
 }
```

Se o elemento não estiver contido no vector, essa versão melhorada da pesquisa sequencial apresenta um desempenho melhor, mas continua a ter uma complexidade linear, ou seja, O(n).

11.5- Pesquisa Sequencial em tabelas Ordenadas

Suponhamos sem perda da generalidade, que as chaves estão ordenadas em ordem crescente. A estratégia para resolver esse problema consiste em percorrer o vector no sentido crescente. Para cada elemento, comparar o conteúdo do campo chave com o valor que procuramos. Se o conteúdo for menor, incrementar o índice; se for igual suspender o processo de pesquisa e retornar o valor dessa posição; se for maior suspender o processo de pesquisa e retornar o valor -1. Esta estratégia pode ser implementada pela seguinte função:

```
int pesquisaSequencialOrdenada (TTable v , int x)
 {
    for ( int i = 0 ; i < v.nElem ; i++ )
       {
          if ( v.item[i].chave == x ) return i;
          if ( v.item[i].chave > x ) return -1;
        }
    return -1;
 }
```

de outra forma:

```
int pesquisaSequencialOrdenada (TTable v , int x)
{
   int i = 0;
   while ( i < v.nElem  &&  v.item[i].chave < x ) i++;
   if ( i == v.nElem )   return -1;
   if ( v.item[i].chave == x ) return i;
   return -1;
}
```

Vamos escrever em seguida, uma versão recursiva, utilizando o paradigma de decrementar para conquistar.

```
int pesquisaSequencialOrdenadaRec (TTable v , int x , int i)
{
   if ( i == v.nElem) return -1;
   if ( v.item[i].chave > x) return -1;
   if ( v.item[i].chave == x) return i;
   return pesquisaSequencialOrdenadaRec ( v , x, i+1);
 }
```

Apesar de ser mais rápido do que a pesquisa sequencial, este método continua a ter uma complexidade linear. Devido a essa complexidade o método torna-se inviável para tabelas com muitos registros. Para suprir essa deficiência, apresentaremos a seguir uma estratégia mais forte.

11.6- Pesquisa Binária

A pesquisa binária (***binary search***) é uma forma de acelerar o processo de pesquisa sequencial em tabelas ordendas, e consiste em fragmentar o vector em duas partes. A pesquisa começa por selecionar uma posição k no interior do vector. Se o elemento encontrado for menor do que a chave que procuramos, devemos continuar a pesquisa na parte do vector à direita de k, mas se o elemento encontrado for maior do que a chave que procuramos devemos continuar a pesquisa na parte do vector à esquerda de k. Finalmente se o elemento encontrado for igual a chave que procuramos a pesquisa termina com sucesso.

Para fragmentarmos o vector em duas partes, devemos dividir o intervalo ao meio em cada iteração. O processo de repetição termina quando encontramos o elemento ou geramos um subvector vazio.

```
int pesquisaBinaria (TTable v , int x)
{
   int inicio = 0, fim = v.nElem-1 , meio;
   while (inicio <= fim)
     {
        meio = (inicio+fim)/2;
        if (v.item[meio].chave == x)  return meio;
        if (v.item[meio].chave < x)   inicio = meio +1
        else   fim = meio – 1;
      }
    return -1;
 }
```

Vamos analisar em seguida a versão recursiva deste método com a estratégia divisão e conquista.

Seja T = (t_0,t_1,…,t_{n-1}) um conjunto de elementos armazenados num vector T, ordenados em ordem crescente tal que n-1 é menor do que a dimensão do vector. A resolução do problema consiste em calcular em primeiro em lugar um índice meio tal que 0 ≤ meio ≤ n-1. Comparar em seguida a chave do registro

que ocupa essa posição com a chave que procuramos. Se a chave que procuramos for menor do que essa chave efetuamos o processo de pesquisa no subconjunto T1=(t_0,t_1,…,t_{meio}), no caso contrário efectuamos o processo de pesquisa no subconjunto T2= (t_{meio+1},t_{meio+2},…,t_{n-1}). Observe que em cada processo de partição do intervalo, concentramos a nossa pesquisa num subconjunto do vector que contêm apenas metade dos seus elementos. Torna-se evidente que esse processo de partição é finito, e termina quando chegarmos a um subconjunto unitário ou a um subconjunto vazio cuja solução é trivial

```
int pesquisaBinariaRec (int inicio, int fim , TTable v , int x)
{
  int meio;
  if ( inicio > fim )  return -1;
  meio = ( inicio+ fim )/2;
  if ( v.item[meio].chave == x ) return meio;
  if ( x < v.item[meio].chave)
     return PesquisaBinariaRec (inicio , meio-1 , v , x);
  return PesquisaBinariaRec  (meio+1 , fim , v , x);
}
```

11.7 - Recomendações Bibliográficas

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

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

Feofiloff P.;- *Algoritmos em linguagem C*, Editora Campus, São Paulo, Brasil, 2009.

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

Robert E. S.;- Programming Abstraction in C: a Second Course in Computer Science, Addison.Wesley, 1998.

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

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

11.8 - Exercícios

11.8.1- A busca ternária aplica-se a tabelas ordenadas e caracteriza-se por: de forma análoga a busca binária, fragmentar o vector em três subvectores. A busca começa por selecionar duas posições k1 e k2 no interior do vector, que denominamos por pivô. O valor que procuramos é comparado aos pivôs de tal forma que é possível escolher um dos subvectores, para continuar o processo de busca. Para realizar essa fragmentação, atribuimos ao primeiro pivô um terço do intervalo, enquanto ao segundo pivô dois terços desse intervalo. Este processo termina quando encontramos o elemento ou um subvector vazio. Desenvolva uma versão iterativa e uma versão recursiva deste método com o paradigma de decrementar para dividir

11.8.2- Diga o que acontece se substituirmos na função de busca binária a expressão meio = inicio +1 por meio = inicio e meio = fim-1 ou meio = fim?

11.8.3- Desenvolva uma função iterativa com estratégia de divisão e conquista, que recebe como argumento uma lista sequencial ordenada em ordem decrescente e uma determinada chave. Verificar se essa chave está na lista.

11.8.4-Desenvolva uma função recursiva com o paradigma de decrementar para conquistar para encontrar e remover uma determinada elemento numa tabela. Desenvolva em seguida a correspondente versão iterativa.

11.8.5- A função que descrevemos em seguida, tem por finalidade encontrar uma determinada chave vector cujos elementos estão em ordem crescente com a estratégia de busca binária. Prove que essa versão funciona e compare com a estudada neste capítulo. Suponha que o vector não esteja vazio.

```
int busca(TTable v , int x)
 {
    int inicio= 0, fim = n-1 , meio;
    while (inicio < fim)
      {
          meio = (inicio+fim)/2;
          if (v.item[meio].chave < x)  inicio = meio +1;
          else  fim = meio -1 ;
      }
   if (v.item[inicio] == x)  return inicio;
   else  return -1;
 }
```

11.8.6- Desenvolva uma função iterativa e a correspondente versão recursiva com o paradigma de decrementar para conquistar que recebe como parâmetro um vector, uma determina chave e um registro com uma determinada informação. Actualizar o registro que contêm essa chave.

11.8.7- Dado o conjunto:

S = {3,7,20,25,26,28,30,34,42,44,50,60,68,75, 86,145,209,250}

analisar o funcionamento da busca binária para x = 34 ou x = 40.

11.8.8- Desenvolva uma versão recursiva com o paradigma de decrementar para conquistar para implementar o método de busca sequencial com sentinela.

11.8.9- A função que descrevemos em seguida, tem por finalidade encontrar uma determinada chave num vector ordenado em ordem crescente com a estratégia de busca binária. Prove que essa versão funciona e compare essa versão com a estudada neste capítulo. Suponha que o vector não esteja vazio.

```
int busca(TTable v , int x)
{
    int med, len;
    med = (n-1)/2;
    len = (n-2)/2;
   while (x != v.item[med].chave])
      {
        if  (x < v.item[meio].chave) med -= len/2;
        else  med += len/2;
        len /= 2;
       }
    return med;
 }
```