19

Ordenação e Pesquisa

No mundo da computação, talvez as tarefas mais fundamentais e extensivamente analisadas sejam ordenação e pesquisa. Essas rotinas são utilizadas em praticamente todos os programas de banco de dados, bem como em compiladores, interpretadores e sistemas operacionais. Este capítulo introduz os conceitos básicos de ordenação e pesquisa. Como você verá, ordenar e pesquisar ilustram diversas técnicas de programação em C.

Como o objetivo de ordenar os dados geralmente é facilitar e acelerar o processo de pesquisa nesses dados, discutimos primeiro a ordenação.

Ordenação

Ordenação é o processo de arranjar um conjunto de informações semelhantes numa ordem crescente ou decrescente. Especificamente, dada uma lista ordenada i de n elementos, então

$$i_1 <= i_2 <= ... <= i_n$$

Muito embora a maioria dos compiladores forneça a função **qsort()** como parte da biblioteca padrão, você deve entender a ordenação por três razões. Primeiro, você não pode aplicar uma função generalizada como **qsort()** a todas as situações. Segundo, pelo fato de **qsort()** ser parametrizada para operar em uma variedade de dados, ela roda mais lentamente que uma ordenação semelhante que opera sobre apenas um tipo de dado. (Generalização aumenta inerentemente o tempo de execução devido ao tempo de processamento extra necessário para

manipular os diversos tipos de dados.) Finalmente, como você verá, embora o algoritmo quicksort (usado por **qsort()**) seja muito eficiente no caso geral, ele pode não ser a melhor ordenação para situações especiais.

Existem duas categorias gerais de algoritmos de ordenação: algoritmos que ordenam matrizes (tanto na memória como em arquivos de acesso aleatório em disco) e algoritmos que ordenam arquivos seqüenciais em disco ou fita. Este capítulo enfoca apenas a primeira categoria, por ser mais relevante à maioria dos programadores.

Geralmente, quando a informação é ordenada, apenas uma porção dessa informação é usada como *chave* da ordenação. Essa chave é utilizada nas comparações, mas, quando uma troca se torna necessária, toda a estrutura de dados é transferida. Por exemplo, em uma lista postal, o campo de código de área (CEP) poderia ser usado como chave, mas o nome e o endereço acompanham o CEP quando uma troca é feita. Com o objetivo de simplificar, os exemplos ordenarão matrizes de caracteres enquanto você aprende os diversos métodos de ordenação. Mais tarde você aprenderá a adaptar esses métodos a qualquer tipo de estrutura de dados.

Tipos de Algoritmos de Ordenação

Existem três métodos gerais para ordenar matrizes:

- por troca
- por seleção
- por inserção

Para entender esses três métodos, imagine as cartas de um baralho. Para ordenar as cartas, utilizando *troca*, espalhe-as, voltadas para cima, numa mesa, então troque as cartas fora de ordem até que todo o baralho esteja ordenado. Utilizando *seleção*, espalhe as cartas na mesa, selecione a carta de menor valor, retire-a do baralho e segure-a em sua mão. Esse processo continua até que todas as cartas estejam em sua mão. As cartas em sua mão estarão ordenadas quando o processo tiver terminado. Para ordenar as cartas por *inserção*, segure todas as cartas em sua mão. Ponha uma carta por vez na mesa, sempre inserindo-a na posição correta. O maço estará ordenado quando não restarem mais cartas em sua mão.

Uma Avaliação dos Algoritmos de Ordenação

Existem muitos algoritmos diferentes para cada método de ordenação. Cada um deles tem seus méritos, mas os critérios gerais para avaliação de um algoritmo são:

- Em que velocidade ele pode ordenar as informações no caso médio?
- Qual a velocidade do seu melhor e pior casos?
- Esse algoritmo apresenta um comportamento natural ou não-natural?
- Ele rearranja elementos com chaves iguais?

Olhe, agora, atentamente para esses critérios. Evidentemente a velocidade em que um algoritmo particular ordena é de grande importância. A velocidade em que uma matriz pode ser classificada está diretamente relacionada com o número de comparações e o número de trocas que ocorrem, com as trocas exigindo mais tempo. Uma *comparação* ocorre quando um elemento da matriz é comparado a outro; uma *troca* ocorre quando dois elementos na matriz ocupam um o lugar do outro. Como você verá em breve, algumas ordenações variam o tempo de ordenação de um elemento de forma exponencial e outras de forma logarítmica.

Os tempos de processamento para o pior e melhor casos são importantes se você espera, freqüentemente, encontrar uma dessas situações. Normalmente, uma ordenação tem um bom caso médio, mas um terrível pior caso.

Diz-se que uma ordenação tem um comportamento *natural* se ela trabalha o mínimo quando a lista já está ordenada, trabalha mais quanto mais desordenada estiver a lista e o maior tempo quando a lista está em ordem inversa. A determinação do quanto uma ordenação trabalha é baseada no número de comparações e trocas que ela deve executar.

Para entender por que rearranjar elementos com chaves iguais pode ser importante, imagine um banco de dados como uma lista postal, que é ordenada de acordo com uma chave principal e uma subchave. A chave principal é o CEP e, dentro dos códigos de CEP, o sobrenome é a subchave. Quando um novo endereço for acrescentado à lista e esta for reordenada, as subchaves (isto é, os sobrenomes com os mesmos códigos de CEP) não devem ser arranjadas. Para garantir que isso não aconteça, uma ordenação não deve trocar as chaves principais de mesmo valor.

A discussão que se segue examina, primeiro, as ordenações representativas de cada categoria e, então, analisa a eficiência de cada uma. Mais adiante, você aprenderá métodos mais aperfeiçoados de ordenação.

A Ordenação Bolha — O Demônio das Trocas

A ordenação mais conhecida (e mais difamada) é a *ordenação bolha*. Sua popularidade vem do seu nome fácil e de sua simplicidade. Porém, é uma das piores ordenações já concebidas.

A ordenação bolha é uma ordenação por trocas. Ela envolve repetidas comparações e, se necessário, a troca de dois elementos adjacentes. Os elementos são como bolhas em um tanque de água — cada uma procura o seu próprio nível. A forma mais simples da ordenação bolha é mostrada aqui:

```
/* A ordenação bolha. */
void bubble(char *item, int count)
{
  register int a, b;
  register char t;

  for(a=1; a<count; ++a)
    for(b=count-1; b>=a; --b) {
      if(item[b-1] > item[b]) {
        /* troca os elementos */
        t = item[b-1];
        item[b-1] = item[b];
        item[b] = t;
      }
    }
}
```

No código anterior, **item** é um ponteiro para uma matriz de caracteres a ser ordenada e **count** é o número de elementos da matriz. A ordenação bolha é dirigida por dois laços. Dado que existem **count** elementos na matriz, o laço mais externo faz a matriz ser varrida **count**-1 vezes. Isso garante que, na pior hipótese, todo elemento estará na posição correta quando a função terminar. O laço mais interno faz as comparações e as trocas. (Uma versão ligeiramente melhorada da ordenação bolha termina se não ocorre nenhuma troca, mas isso acrescenta uma outra comparação a cada passagem pelo laço interno.)

Essa versão da ordenação bolha pode ser utilizada para ordenar uma matriz de caracteres em ordem ascendente. Por exemplo, o programa seguinte ordena uma string digitada no teclado.

```
/*Sort Driver*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void bubble(char *item, int count);
```

```
void main(void)
{
  char s[80];

  printf("Digite uma string:");
  gets(s);
  bubble(s, strlen(s));
  printf("A string ordenada é: %s.\n", s);
}
```

Para ver como funciona a ordenação bolha, assuma que a matriz a ser ordenada contenha **dcab**. Cada passo é mostrado aqui:

inicial	d c a b
passo 1	a d c b
passo 2	a b d c
passo 3	a b c d

Ao analisar qualquer ordenação, você deve determinar quantas comparações e trocas serão realizadas para o menor, médio e pior casos. Com a ordenação bolha, o número de comparações é sempre o mesmo, porque os dois laços **for** repetem o número especificado de vezes, estando a lista inicialmente ordenada ou não. Isso significa que a ordenação bolha sempre executa

$$^1/_2(n^2 - n)$$

comparações, onde n é o número de elementos a ser ordenado. Essa fórmula deriva do fato de que o laço mais externo executa $n-1$ vezes e o laço mais interno $n/2$ vezes. Multiplicando-se um pelo outro obtemos a fórmula anterior.

O número de trocas é zero, para o melhor caso, em uma lista já ordenada. O número de trocas para o caso médio e o pior caso são

médio	$^3/_4(n^2 - n)$
pior	$^3/_2(n^2 - n)$

Está fora do escopo deste livro explicar a origem das fórmulas anteriores, mas você pode observar que, à medida que a lista se torna menos ordenada, o número de elementos fora de ordem se aproxima do número de comparações. (Lembre-se de que, na ordenação bolha, existem três trocas para cada elemento fora de ordem.)

Essa é uma ordenação *n-quadrado*, pois seu tempo de execução é um múltiplo do quadrado do número de elementos. Esse tipo de algoritmo é muito ineficiente quando aplicado a um grande número de elementos, porque o tempo de execução está diretamente relacionado com o número de comparações e trocas. Por exemplo, ignorando o tempo que leva para trocar qualquer elemento fora

da posição, assuma que cada comparação leva 0,001 segundos. Para ordenar 10 elementos, são gastos 0,05 segundos, para ordenar 100 elementos, serão gastos 5 segundos, e ordenar 1.000 elementos, tomará 500 segundos. Uma ordenação de 100.000 elementos, o tamanho de uma pequena lista telefônica, levaria em torno de 5.000.000 segundos ou 1.400 horas ou por volta de dois meses de ordenação contínua! A Figura 19.1 mostra como o tempo de execução aumenta com relação ao tamanho da matriz.

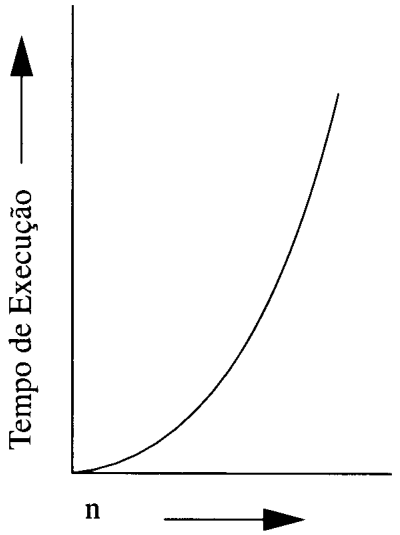

Figura 19.1 Tempo de execução de uma ordenação n^2 em relação ao tamanho da matriz.

Você pode fazer ligeiras melhorias na ordenação bolha para que ela fique mais rápida. Por exemplo, a ordenação bolha tem uma peculiaridade: um elemento fora de ordem na "extremidade grande" (como o "a" no exemplo **dcab**) irá para a sua posição correta em um passo, mas um elemento desordenado na "extremidade pequena" (como o "d") subirá vagarosamente para seu lugar apropriado. Isso sugere uma melhoria na ordenação bolha. Em vez de sempre ler a matriz na mesma direção, pode-se inverter a direção entre passos subseqüentes. Dessa forma, elementos muito fora do lugar irão mais rapidamente para suas posições corretas. Essa versão da ordenação bolha é chamada de *ordenação oscilante*, devido ao seu movimento de vaivém sobre a matriz.

```
/* A ordenação oscilante. */
void shaker(char *item, int count)
```

```
{
  register int a;
  int exchange;
  char t;

  do {
    exchange = 0;
    for(a=count-1; a>0; --a) {
      if(item[a-1]>item[a]) {
        t = item[a-1];
        item[a-1] = item[a];
        item[a] = t;
        exchange = 1;
      }
    }

    for(a=1; a<count; ++a) {
      if(item[a-1]>item[a]) {
        t = item[a-1];
        item[a-1] = item[a];
        item[a] = t;
        exchange = 1;
      }
    }
  } while(exchange); /*ordena até que não existam mais trocas*/
}
```

Embora a ordenação oscilante seja uma melhoria da ordenação bolha, ela ainda é executada na ordem de um algoritmo *n-quadrado*, porque o número de comparações não foi alterado e o número de trocas foi reduzido de uma constante relativamente pequena. A ordenação oscilante é melhor que a ordenação bolha, mas existem ordenações ainda melhores.

Ordenação por Seleção

A ordenação por seleção seleciona o elemento de menor valor e troca-o pelo primeiro elemento. Então, para os n-1 elementos restantes, é encontrado o elemento de menor chave, trocado pelo segundo elemento e assim por diante. As trocas continuam até os dois últimos elementos. Por exemplo, se o método de seleção fosse utilizado na matriz **bdac**, cada passo se apresentaria como:

inicial	b	d	a	c
passo 1	a	d	b	c
passo 2	a	b	d	c
passo 3	a	b	c	d