MAKRON
Books

Parte 3

Algoritmos e Aplicações

O propósito da Parte 3 é mostrar formas pelas quais C pode ser aplicada em uma série de tarefas de programação. Neste processo, ela apresenta muitos algoritmos comuns e úteis e aplicações que ilustram o uso da linguagem C. Muitos dos exemplos contidos nesta Parte 3 podem ser úteis como pontos de partida para os seus próprios projetos em C.

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.

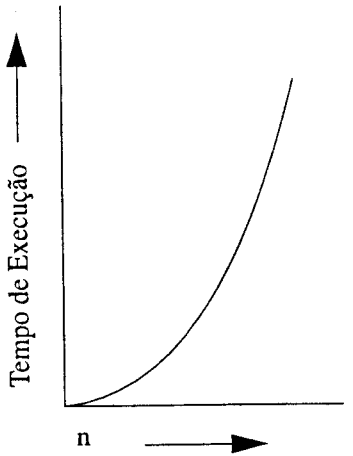


Figura 19.1 Tempo de execução de uma ordenação n^2 em relação ao tamanho da matriz.

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

O código a seguir mostra uma ordenação por seleção simples.

```
/* A ordenação por seleção. */
void select(char *item, int count)
{
  register int a, b, c;
  int exchange;
  char t;

  for(a=0; a<count-1; ++a) {
    exchange = 0;
    c = a;
    t = item[a];
    for(b=a+1; b<count; ++b) {
      if(item[b]<t) {
        c = b;
        t = item[b];
        exchange = 1;
      }
    }
    if(exchange) {
      item[c] = item[a];
      item[a] = t;
    }
  }
}
```

Infelizmente, como na ordenação bolha, o laço mais externo é executado n-1 vezes e o laço interno $1/2(n)$ vezes. Como resultado, a ordenação por seleção requer

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

comparações, que a tornam muito lenta para um número grande de itens. O número de trocas para o melhor e o pior casos são

melhor	$3(n\text{-}1)$
pior	$n^2/4 + 3(n-1)$

Para o melhor caso, quando a lista está inicialmente ordenada, apenas n-1 elementos precisam ser movimentados e cada movimento requer três trocas. O pior caso aproxima-se do número de comparações. O caso médio é difícil de ser determinado e seu desenvolvimento está além do escopo deste livro. No entanto, é igual a

$$n(\log n + y)$$

onde y é a constante de Euler, aproximadamente 0,577216.

Embora o número de comparações para a ordenação bolha e para a ordenação por seleção seja o mesmo, o número de trocas, no caso médio, é muito menor para a ordenação por seleção. Contudo, existem ordenações ainda melhores.

Ordenação por Inserção

A ordenação por *inserção* é o terceiro e último dos algoritmos simples de ordenação. Inicialmente, ela ordena os dois primeiros membros da matriz. Em seguida, o algoritmo insere o terceiro membro na sua posição ordenada com relação aos dois primeiros membros. Então, insere o quarto elemento na lista dos três elementos. O processo continua até que todos os elementos tenham sido ordenados. Por exemplo, dada a matriz **dcab**, cada passo da ordenação por inserção é mostrado aqui:

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

O código para uma versão da ordenação por inserção é dado a seguir:

```
/* A ordenação por inserção. */
void insert(char *item, int count)
{

  register int a, b;
  char t;
  for(a=1; a<count; ++a) {
    t = item[a];
    for(b=a-1; b>=0 && t<item[b]; b--)
      item[b+1] = item[b];
    item[b+1] = t;
  }
}
```

Ao contrário da ordenação bolha e da ordenação por seleção, o número de comparações que ocorrem durante a ordenação por inserção depende de como a lista está inicialmente ordenada. Se a lista estiver em ordem, o número de comparações será n-1. Se estiver fora de ordem, o número de comparações será

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

O caso médio é

$^1/_4(n^2 - n)$

O número de troca para cada caso é o seguinte

melhor	$2(n - 1)$
médio	$^1/_4(n^2 - n)$
pior	$^1/_2(n^2 + n)$

Portanto, para o pior caso, a ordenação por inserção é tão ruim quanto a ordenação bolha e a ordenação por seleção e, para o caso médio, é somente um pouco melhor. No entanto, a ordenação por inserção tem duas vantagens. Primeiro, ela se comporta naturalmente, isto é, trabalha menos quando a matriz já está ordenada e o máximo quando a matriz está ordenada no sentido inverso. Isso torna a ordenação por inserção excelente para listas que estão quase em ordem. A segunda vantagem é que ela não rearranja elementos de mesma chave. Isso significa que uma lista que é ordenada por duas chaves permanece ordenada para ambas as chaves após uma ordenação por inserção.

Muito embora o número de comparações possa ser razoavelmente baixo para certos conjuntos de dados, a matriz precisa ser deslocada cada vez que um elemento é colocado na sua posição correta. Como resultado, o número de movimentações pode ser significativo. Contudo, existem ordenações ainda melhores.

Ordenações Melhores

Todos os algoritmos da seção anterior tinham o grave defeito de processarem em um tempo *n*-quadrado. Para grandes quantidades de dados, isso torna a ordenação muito lenta. De fato, em algum ponto, as ordenações seriam lentas demais para serem usadas. Infelizmente, casos de "ordenação que durou três dias" são freqüentemente verdadeiros. Quando uma ordenação demora tanto, normalmente é uma falha do algoritmo básico. Porém, a primeira reação geralmente é "vamos escrevê-lo em código assembly". A linguagem assembly aumenta, de fato, a velocidade de uma rotina de um fator constante. Se o algoritmo básico for ineficiente, a ordenação será lenta, não importando quão bom seja o código. Lembre-se: quando uma rotina processa relativamente a *n*, aumentar a velocidade do código do computador provocará apenas uma pequena melhora, porque a razão em que o tempo de execução aumenta é exponencial. (Em essência, a curva da Figura 19.1 é deslocada levemente para a direita, mas a curva continua a mesma.) A norma prática é que, se uma rotina não é suficientemente rápida quando escrita em C, ela não será rápida o bastante em linguagem assembly. A solução é usar um melhor algoritmo de ordenação.

Esta seção descreve duas excelentes ordenações. A primeira é a ordenação *Shell*. A segunda, a *quicksort*, normalmente é considerada a melhor rotina de ordenação. Essas ordenações rodam tão rápido que, se você piscar, você as perde de vista!

Ordenação Shell

A ordenação Shell é assim chamada devido ao seu inventor, D. L. Shell. Porém, o nome provavelmente pegou porque seu método de operação é freqüentemente descrito como conchas do mar empilhadas umas sobre as outras.

O método geral é derivado da ordenação por inserção e é baseado na diminuição dos incrementos. Considere o diagrama da Figura 19.2. Primeiro, todos os elementos que estão três posições afastados um do outro são ordenados. Em seguida, todos os elementos que estão duas posições afastados são ordenados. Finalmente, todos os elementos adjacentes são ordenados.

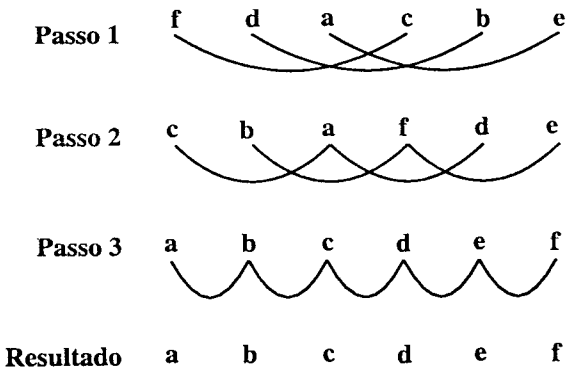


Figura 19.2 A ordenação Shell.

Não é fácil perceber que esse método conduz a bons resultados ou mesmo que ordene a matriz. Mas ele executa ambas as funções. Cada passo da ordenação envolve relativamente poucos elementos ou elementos que já estão razoavelmente em ordem, logo, a ordenação Shell é eficiente e cada passo aumenta a ordenação dos dados.

A seqüência exata para os incrementos pode mudar. A única regra é que o último incremento deve ser 1. Por exemplo, a seqüência

9, 5, 3, 2, 1

funciona bem e é usada na ordenação Shell mostrada aqui. Evite seqüências que são potências de 2 — por razões matemáticas complexas, elas reduzem a eficiência do algoritmo de ordenação (mas a ordenação ainda funciona).

```
/* A ordenação Shell. */
void shell(char *item, int count)
{

  register int i, j, gap, k;
  char x, a[5];

  a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1;

  for(k=0; k<5; k++) {
    gap = a[k];
    for(i=gap; i<count; ++i) {
      x = item[i];
      for(j=i-gap; x<item[j] && j>=0; j=j-gap)
        item[j+gap] = item[j];
      item[j+gap] = x;
    }
  }
}
```

Você deve ter observado que o laço **for** mais interno tem duas condições de teste. A comparação **x<item[j]** é obviamente necessária para o processo de ordenação. O teste **j>=0** evita que os limites da matriz **item** sejam ultrapassados. Essas verificações extras degenerarão até certo ponto o desempenho da ordenação Shell.

Versões um pouco diferentes da ordenação Shell empregam elementos especiais de matriz, chamados sentinelas, que não fazem parte realmente da matriz a ser ordenada. *Sentinelas* guardam valores especiais de terminação, que indicam o menor e o maior elemento possível. Dessa forma, as verificações dos limites são desnecessárias. No entanto, usar sentinelas requer um conhecimento específico dos dados, o que limita a generalização da função de ordenação.

A análise da ordenação Shell apresenta alguns problemas matemáticos que estão além do objetivo desta discussão. O tempo de execução é proporcional a

$$n^{1.2}$$

para se ordenar n elementos. Essa é uma redução significativa com relação às ordenações n-quadrado. Para entender o quanto essa ordenação é melhor, observe a Figura 19.3, que mostra os gráficos das ordenações n^2 e $n^{1.2}$. Porém, antes de se decidir pela ordenação Shell, você deve saber que a ordenação quicksort é ainda melhor.

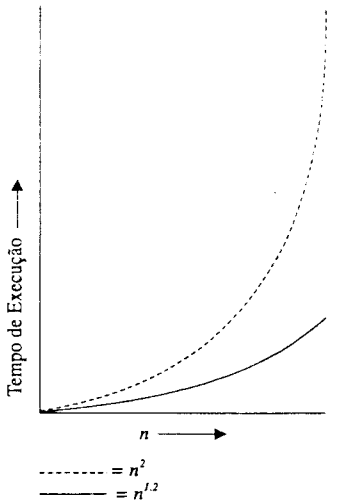


Figura 19.3 As curvas n^2 e $n^{1.2}$.

Quicksort

A quicksort, inventada e denominada por C.A.R. Hoare, é superior a todas as outras ordenações deste livro, e geralmente é considerada o melhor algoritmo de ordenação de propósito geral atualmente disponível. É baseada no método de ordenação por trocas. Isso é surpreendente, quando se considera o terrível desempenho da ordenação bolha!

A quicksort é baseada na idéia de partições. O procedimento geral é selecionar um valor, chamado de *comparando*, e, então, fazer a partição da matriz em duas seções, com todos os elementos maiores ou iguais ao valor da partição de um lado e os menores do outro. Esse processo é repetido para cada seção restante até que a matriz esteja ordenada. Por exemplo, dada a matriz **fedacb** e usando o valor **d** para a partição, o primeiro passo da quicksort rearranja a matriz como segue:

início	f e d a c b
passo1	b c a d e f

Esse processo é, então, repetido para cada seção — isto é, **bca** e **def**. Como você pode ver, o processo é essencialmente recursivo por natureza e, certamente, as implementações mais claras da quicksort são algoritmos recursivos.

Você pode selecionar o valor do comparando intermediário de duas formas. Você pode escolhê-lo aleatoriamente ou selecioná-lo fazendo a média de um pequeno conjunto de valores retirado da matriz. Para uma ordenação ótima, você deveria selecionar um valor que estivesse precisamente no centro da faixa de valores. Porém, isso não é fácil para a maioria dos conjuntos de dados. No pior caso, o valor escolhido está em uma extremidade e, mesmo nesse caso, quicksort ainda tem um bom rendimento. A versão seguinte da quicksort seleciona o elemento intermediário da matriz. Embora isso nem sempre resulte em uma boa escolha, a ordenação ainda é efetuada corretamente.

```
/* Função de inicialização da Quicksort. */
void quick(char *item, int count)
{
  qs(item, 0, count-1)
}

/* A Quicksort. */
void qs(char *item, int left, int right)
{
  register int i, j;
  char x, y;

  i = left; j = right;
  x = item[(left+right)/2];

  do {
    while(item[i]<x && i<right) i++;
    while(x<item[j] && j>left) j--;

    if(i<=j) {
      y = item[i];
      item[i] = item[j];
      item[j] = y;
      i++; j--;
    }
  } while(i<=j);

  if(left<j)  qs(item, left, j);
  if(i<right) qs(item, i, right);
}
```

Nessa versão, a função **quick()** executa a chamada à função da ordenação principal **qs()**. Isso permite manter a interface comum com **item** e **count**, mas não é essencial porque **qs()** poderia ter sido chamada diretamente, usando-se três argumentos.

A dedução do número de comparações e de trocas que quicksort realiza requer uma matemática fora do âmbito deste livo. Porém, o número médio de comparações é

$n \log n$

e o número médio de trocas é aproximadamente

$n/6 \log n$

Esses números são significativamente menores do que aqueles vistos até agora para qualquer ordenação.

No entanto, existe um aspecto particularmente problemático de quicksort sobre o qual você deve ser advertido. Se o valor do comparando, para cada partição, for o maior valor, então a quicksort se degenerará em uma ordenação lenta com um tempo de procesamento n. Geralmente, porém, isso não acontece.

Você deve escolher com cuidado um método para definir o valor do comparando. O método é freqüentemente determinado pelo tipo de dado que está sendo ordenado. Em listas postais muito grandes, onde a ordenação é freqüentemente feita por meio do código CEP, a seleção é simples, porque os códigos são razoavelmente distribuídos — e uma simples função algébrica pode determinar um comparando adequado. Porém, em certos bancos de dados, as chaves podem ser iguais ou muito próximas em valor e uma seleção randômica é freqüentemente a melhor. Um método comum e satisfatório é tomar uma amostra com três elementos de uma partição e utilizar o valor médio.

Escolhendo uma Ordenação

Geralmente, quicksort é a melhor ordenação em virtude da sua velocidade. Porém, quando apenas listas muito pequenas de dados devem ser ordenadas (menos que 100), o tempo extra criado pelas chamadas recursivas do quicksort pode compensar os benefícios de um algoritmo superior. Em casos muito raros como esse, uma das ordenações mais simples — talvez até mesmo a ordenação bolha — pode ser mais rápida.

Ordenando Outras Estruturas de Dados

Até agora, apenas matrizes de caracteres foram ordenadas. Obviamente, matrizes de quaisquer tipos de dados intrínsecos podem ser ordenadas simplesmente trocando-se os tipos de dados dos parâmetros e as variáveis da função de ordenação. Geralmente, porém, são os tipos de dados complexos, como strings, ou agrupamentos de informações, como estruturas, que precisam ser ordenados. A maioria das ordenações envolve uma chave e informação correlacionada com essa chave. Para mudar o algoritmo e, assim, acomodar uma chave, você precisa alterar a seção de comparação, a seção das trocas, ou ambas. O algoritmo permanece inalterado.

A ordenação quicksort será usada nos exemplos seguintes por ser uma das melhores rotinas de uso geral disponíveis até o momento. No entanto, as mesmas técnicas se aplicam a qualquer uma das ordenações descritas anteriormente.

Ordenação de Strings

A ordenação de strings é uma tarefa comum na programação. De longe, é mais fácil ordenar strings quando estão contidas numa tabela de strings. Uma tabela de strings é simplesmente uma matriz de strings. E, em matriz de strings é uma matriz bidimensional de caracteres na qual o número de strings na tabela é determinado pelo tamanho da dimensão esquerda e o comprimento máximo de cada string é determinado pelo tamanho da dimensão direita. (Veja o Capítulo 4 para mais informações sobre matrizes de strings). A versão para string da quicksort a seguir aceita uma matriz de strings, cada uma de até 10 caracteres de comprimento. (Você pode modificar esse comprimento, se assim desejar.) Essa versão ordena strings em ordem alfabética.

```
/* Uma quicksort para strings. */
void quick_string(char item[][10], int count)
{
  qs_string(item, 0, count-1)
}

void qs_string(char item[][10], int left, int right)
{
  register int i, j;
  char *x;
  char temp[10];

  i = left; j = right;
  x = item[(left+right)/2];
```



```
  do {
    while(strcmp(item[i],x)<0 && i<right) i++;
    while(strcmp(item[j],x)>0 && j>left) j--;
    if(i<=j) {
      strcpy(temp, item[i]);
      strcpy(item[i], item[j]);
      strcpy(item[j], temp);
      i++; j--;
    }
  } while(i<=j);

  if(left<j)  qs_string(item, left, j);
  if(i<right) qs_string(item, i, right);
}
```

Observe que o passo da comparação foi alterado para usar a função **strcmp()**. A função devolve um número negativo se a primeira string é lexicograficamente menor que a segunda, zero se as strings são iguais e um número positivo se a primeira string é lexicograficamente maior que a segunda. Note que para trocar duas strings, são necessárias três chamadas a **strcpy()**.

O uso da função **strcmp()** diminui a velocidade da ordenação por duas razões. Primeiro, ela envolve uma chamada a uma função, que sempre toma tempo. Segundo, **strcmp()** realiza diversas comparações para determinar a relação entre as duas strings. No primeiro caso, se a velocidade de execução for realmente crítica, pode-se colocar o código para **strcmp()** em linha, dentro da rotina, duplicando seu código. No segundo caso, não há maneira de evitar a comparação entre as strings, visto que, por definição, essa é a tarefa que tem de ser executada. O mesmo raciocínio se aplica a função **strcpy ()**. O uso de **strcpy()** para trocar duas strings consiste em uma chamada de função e uma troca caractere a caractere das duas strings — e ambas gastam tempo. O tempo da chamada de função poderia ser eliminado pelo uso de código em linha. No entanto, não podemos modificar o fato de que a troca de strings significa trocar seus caracteres (um a um).

Ordenação de Estruturas

Muitos dos programas aplicativos que requerem uma ordenação precisam ter um agrupamento de dados ordenados. Uma lista postal é um excelente exemplo, porque o nome, a rua, a cidade, o estado e o CEP estão todos relacionados. Quando essa unidade conglomerada de dados é ordenada, uma chave de ordenação é usada, mas toda a estrutura é trocada. Para entender como isso é feito, primeiro criemos uma estrutura.

Para ver um exemplo de ordenação de estruturas, considere uma estrutura, chamada **address**, que seja capaz de conter uma lista de endereços. Uma estrutura como esta poderia ser usada por um programa de mala direta. A estrutura **address** é mostrada aqui.

```
struct address {
  char name[40];
  char street[40];
  char city[20];
  char state[3];
  char zip[10];
};
```

Como é razoável que uma lista postal possa ser arranjada como uma matriz de estruturas, assuma, para esse exemplo, que a rotina ordenará uma matriz de estruturas do tipo **address**. A rotina é mostrada aqui. Ela ordena os endereços pelo código de endereçamento postal (CEP).

```
/* Uma quicksort para estruturas do address. */
void quick_struct(struct address item[], int count)
{
  qs_struct(item,0,count-1)
}

void qs_struct(struct address item[], int left, int right)
{

  register int i, j;
  char *x;
  struct address temp;

  i = left; j = right;
  x = item[(left+right)/2].zip;

  do {
    while(strcmp(item[i].zip,x)<0 && i<right) i++;
    while(strcmp(item[j].zip,x)>0 && j>left) j--;
    if(i<=j) {
      temp = item[i];
      item[i] = item[j];
      item[j] = temp;
      i++; j--;
    }
```



```
  } while(i<=j);
  if(left<j)  qs_struct(item, left, j);
  if(i<right) qs_struct(item, i, right);
}
```

Ordenando Arquivos de Acesso Aleatório em Disco

Existem dois tipos de arquivos em disco: *seqüenciais* e de *acesso aleatório*. Se qualquer tipo de arquivo em disco for pequeno o bastante, ele pode ser lido para a memória e as rotinas de ordenação de matrizes apresentadas anteriormente podem ser utilizadas para ordená-lo. Porém, muitos arquivos em disco são muito grandes para serem ordenados facilmente na memória e exigem técnicas especiais. Esta seção mostra uma maneira pela qual arquivos em disco de acesso aleatório podem ser ordenados.

Arquivos em disco de acesso aleatório têm duas vantagens principais sobre arquivos seqüenciais em disco. Primeiro, eles são mais fáceis de manter. Você pode atualizar informações sem precisar copiar toda a lista. Segundo, eles podem ser tratados como uma matriz muito grande em disco, o que simplifica enormemente a ordenação.

Tratar um arquivo de acesso aleatório como uma matriz significa que você pode usar a base da quicksort com poucas modificações. Em lugar de indexar uma matriz, a versão em disco da quicksort deve usar **fseek()** para pesquisar os registros apropriados no disco.

Cada situação de ordenação difere da estrutura exata de dados que é ordenada e da chave que é usada. Todavia, a idéia geral de ordenação de arquivos de acesso aleatório em disco pode ser compreendida desenvolvendo-se um programa de ordenação para estruturas do tipo **address**, a estrutura para a lista postal definida anteriormente. A quantidade de endereços a ordenar é especificada por **NUM_ELEMENTS** (que é 4 para este programa). Em aplicações reais, porém, uma contagem de registros precisaria ser mantida dinamicamente. Você deve experimentar este programa por conta própria, tentando usar diferentes tipos de estruturas, contendo diferentes tipos de dados.

```
/* Ordenação em disco para estruturas do tipo address. */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#define NUM_ELEMENTS 4 /* Esse é um número arbitrário
                          que deve ser determinado
                          dinamicamente para cada lista.*/
struct address {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  char zip[11];
}ainfo;

struct address addrs [NUM_ELEMENTS] = {
  "A. Alexander", "101 1st St", "Olney", "Ga", "55555",
  "B. Bertrand", "22 2nd Ave", "Oakland", "Pa", "34232",
  "C. Carlisle", "33 3rd Blvd", "Ava", "Or", "92000",
  "D. Dodger", "4 Fourth Dr", "Fresno", "Mi", "45678"
};

void quick_disk(FILE *fp, int count);
void qs_disk(FILE *fp, int left, int right);
void swap_all_fields(FILE *fp, long i, long j);
char *get_zip(FILE *fp, long rec);

void main(void)
{
  FILE *fp;

  /* primeiro, crie um arquivo para ser ordenado */
  if((fp=fopen ("mlist", "wb"))==NULL) {
    printf ("O arquivo não pode ser aberto para escrita.\n");
    exit (1);
  }
  printf ("Gravando dados não ordenados para disco. \n");
  fwrite (addrs, sizeof (addrs), 1, fp);
  fclose (fp);

  /* agora, ordene o arquivo */
  if((fp=fopen("mlist", "rb+"))==NULL) {
    printf("O arquivo não pode ser aberto para leitura/escrita.
    \n");
    exit(1);
  }
  /printf ("Ordenando arquivo de disco. \n");
```



```
  quick_disk(fp, NUM_ELEMENTS);
  fclose(fp);
  printf("Lista ordenada.\n");
}

/* Um quicksort para arquivos. */
void quick_disk(FILE *fp, int count)
{
  qs_disk(fp, 0, count-1);
}

void qs_disk(FILE *fp, int left, int right)
{
  long int i, j;
  char x[100];

  i = left; j = right;

  strcpy(x, get_zip(fp,(long)(i+j)/2));  /* obtém o CEP inter-
                                            mediário */

  do {
    while(strcmp(get_zip(fp,i),x)<0 && i<right) i++;
    while(strcmp(get_zip(fp,j),x)>0 && j>left) j--;

    if(i<=j) {
      swap_all_fields(fp, i, j);
      i++; j--;
    }
  } while(i<=j);

  if(left<j)  qs_disk(fp, left, (int) j);
  if(i<right) qs_disk(fp, (int) i, right);
}

void swap_all_fields(FILE *fp, long i, long j)
{
  char a[sizeof(ainfo)], b[sizeof(ainfo)];

  /* primeiro lê os registros i e j */
  fseek(fp, sizeof(ainfo)*i, SEEK_SET);
  fread(a, sizeof(ainfo), 1, fp);

  fseek(fp, sizeof(ainfo)*j, SEEK_SET);
  fread(b, sizeof(ainfo), 1, fp);
```

```
  /* em seguida escreve-os de volta em posições diferentes */
  fseek(fp, sizeof(ainfo)*j, SEEK_SET);
  fwrite(a, sizeof(ainfo), 1, fp);

  fseek(fp, sizeof(ainfo)*i, SEEK_SET);
  fwrite(b, sizeof(ainfo), 1, fp);
}

/* Devolve um ponteiro para o código cep */
char *get_zip(FILE *fp, long rec)
{
  struct address *p;

  p = &ainfo;

  fseek(fp, rec*sizeof(ainfo), SEEK_SET);
  fread(p, sizeof(ainfo), 1, fp);

  return ainfo.zip;
}
```

Como você pode ver, foi necessário escrever diversas funções de suporte para ordenar os registros com endereços. Na seção de comparação da ordenação, a função **get_zip()** foi usada para retornar um ponteiro para o CEP do comparando e do registro sendo verificado. A função **swap_all_fields()** efetua a troca real dos dados. Note que, sob a maioria dos sistemas operacionais, a ordem das leituras e gravações tem um grande impacto sobre a velocidade desta ordenação. Quando ocorre uma troca, o código exibido força um acesso ao registro **i**, depois ao **j**. Enquanto a cabeça da unidade de disco ainda estiver posicionada em **j**, os dados de **j** são gravados. Isto significa que a cabeça não precisa movimentar-se uma grande distância. Se o código tivesse sido escrito para gravar os dados de **i** primeiro, teria sido necessário um acesso adicional.

Pesquisa

Bancos de dados existem para que, de tempos em tempos, um usuário possa localizar o dado de um registro, simplesmente digitando sua chave. Há apenas um método para encontrar informações em um arquivo (matriz) desordenado e um outro para um arquivo (matriz) ordenado. Muitos compiladores fornecem funções de pesquisa como parte da biblioteca padrão. Porém, analogamente à ordenação, rotinas de uso geral algumas vezes são simplesmente ineficientes em situações mais exigentes devido ao tempo extra provocado pela sua generalização.

Métodos de Pesquisa

Encontrar informações em uma matriz desordenada requer uma pesquisa seqüencial, começando no primeiro elemento e parando quando o elemento procurado ou o final da matriz é encontrado. Esse método deve ser usado em dados desordenados, mas também pode ser aplicado a dados ordenados. Se os dados foram ordenados, você pode usar uma pesquisa binária, o que ajuda a localizar o dado mais rapidamente.

A Pesquisa Seqüencial

A pesquisa seqüencial é fácil de ser codificada. A função a seguir faz uma pesquisa em uma matriz de caracteres de comprimento conhecido até que seja encontrado, a partir de uma chave específica, o elemento procurado:

```
sequential_search(char *item, int count, char key)
{
  register int t;

  for(t=0; t<count; ++t)
    if(key==item[t]) return t;
  return -1; /* não encontrou */
}
```

Essa função devolve o índice da entrada encontrada se existir alguma; caso contrário, ela devolve -1.

É fácil ver que uma pesquisa seqüencial testará em média $1/2n$ elementos. No melhor caso, testará somente um elemento e, no pior caso, n elementos. Se a informação está armazenada em disco, o tempo de procura pode ser muito longo. Mas se os dados estiverem desordenados, a pesquisa seqüencial é o único método disponível.

Pesquisa Binária

Se o dado a ser encontrado se apresentar de forma ordenada, você poderá usar um método muito superior para encontrar o elemento procurado. Esse método é a *pesquisa binária*, que utiliza a abordagem "dividir e conquistar". Ele primeiro verifica o elemento central. Se esse elemento é maior que a chave, ele testa o elemento central da primeira metade; caso contrário, ele testa o elemento central da segunda metade. Esse procedimento é repetido até que o elemento seja encontrado ou que não haja mais elementos a testar.

Por exemplo, para encontrar o número 4 na matriz

1 2 3 4 5 6 7 8 9

uma pesquisa binária primeiro testa o elemento médio, nesse caso 5. Visto que ele é maior que 4, a pesquisa continua com a primeira metade, ou

1 2 3 4 5

O elemento central agora é 3, que é menor que 4, então, a primeira metade é descartada. A pesquisa continua com

4 5

Nesse momento o elemento é encontrado.

Em uma pesquisa binária, o número de comparações, no pior caso, é

$\log_2 n$

No caso médio, o número é um pouco menor e, no melhor caso, o número de comparações é um.

A seguir, é mostrada uma pesquisa binária para matrizes de caracteres. Você pode fazer com que essa pesquisa encontre qualquer estrutura de dados arbitrária, trocando a parte de comparação da rotina:

```
/* A pesquisa binária. */
binary(char *item, int count, char key)
{
  int low, high, mid;

  low = 0; high = count-1;
  while(low<=high) {
    mid = (low+high)/2;
    if(key<item[mid]) high = mid-1;
    else if(key>item[mid]) low = mid+1;
    else return mid;  /* encontrou */
  }
  return -1;
}
```

20

Filas, Pilhas, Listas Encadeadas e Árvores Binárias

Programas consistem em duas coisas: algoritmos e estruturas de dados. Um bom programa é uma combinação de ambos. A escolha e a implementação de uma estrutura de dados são tão importantes quanto as rotinas que manipulam os dados. A forma como a informação é organizada e acessada é normalmente determinada pela natureza do problema de programação. Por essa razão, é importante ter o método certo de armazenamento e recuperação, para uma variedade de situações, na sua bagagem de conhecimentos.

A separação que existe entre o conceito lógico de um item de dados e sua representação de máquina tem correlação inversa com sua abstração. Ou seja, conforme os tipos de dados se tornam mais complexos, a forma como o programador os imagina apresenta uma semelhança cada vez menor com a forma na qual eles são, na realidade, representados na memória. Por exemplo, tipos simples como **char** e **int** estão intimamente ligados à sua representação de máquina; e o valor que um inteiro tem na sua representação de máquina aproxima-se daquilo que o programador imagina que ele tem. Matrizes simples, que são coleções organizadas de tipos de dados simples, não são tão intimamente ligadas como tipos simples, porque uma matriz pode não aparecer na memória da forma que o programador supõe que apareça.

Os números em ponto flutuante estão menos relacionados ainda. A representação interna na máquina real tem pouca semelhança com a imagem que alguns programadores fazem de um número em ponto flutuante. A estrutura, que é um tipo de dado conglomerado acessado sobre apenas um nome, é ainda mais distinta da representação da máquina. O último nível de abstração transcende os meros aspectos físicos dos dados e, em contrapartida, concentra-se na

seqüência pela qual os dados serão acessados — isso é, armazenados e recuperados. Em essência, os dados físicos estão unidos a um *mecanismo de dados*, que controla a forma como as informações podem ser acessadas pelo seu programa.

Existem basicamente quatro desses mecanismos:

- fila
- pilha
- lista encadeada
- árvore binária

Cada um desses métodos fornece uma solução para uma classe de problemas. Esses métodos são essencialmente dispositivos que executam uma operação específica de armazenamento e recuperação da informação dada, de acordo com o que lhes foi requisitado. Todos eles armazenam um item e recuperam um item, onde um item é uma unidade de informação. O restante deste capítulo mostra como você pode construir estas estruturas de dados usando a linguagem C. Nesse processo são ilustradas diversas técnicas comuns de programação em C, incluindo a alocação dinâmica e a manipulação de ponteiros.

Filas

Uma *fila* é simplesmente uma lista linear de informações, que é acessada na ordem *primeiro a entrar, primeiro a sair*, sendo chamada, algumas vezes, de FIFO (First In, First Out). Isto é, o primeiro item colocado na fila é o primeiro a ser retirado, o segundo item colocado é o segundo a ser recuperado e assim por diante. Essa é a única forma de armazenar e recuperar em uma fila; não é permitido acesso randômico a nenhum item específico.

Filas são muito comuns no nosso dia-a-dia. Por exemplo, filas (físicas) em um banco ou lanchonete são filas lógicas (exceto quando maus fregueses tentam furá-la!). Para visualizar como uma fila opera, considere duas funções: **qstore()** e **qretrieve()**. A função **qstore()** coloca um item no final da fila e **qretrieve()** remove o primeiro item da fila e devolve seu valor. A Tabela 20.1 mostra o efeito de uma série dessas operações.

Tenha em mente que uma operação de recuperação remove um item da fila e o destrói, se não for armazenado em algum outro lugar. Assim, uma fila pode ficar vazia, porque todos os itens foram removidos, muito embora o programa ainda esteja em atividade.

Tabela 20.1 Uma fila em ação.

Ação	Conteúdo da fila
qstore(A)	A
qstore(B)	A B
qstore(C)	A B C
qretrieve() devolve A	B C
qstore(D)	B C D
qretrieve() devolve B	C D
qretrieve() devolve C	D

Filas são usadas em muitas situações de programação. Uma das mais comuns é em simulações. Filas também são usadas em distribuição de eventos, como nos diagramas PERT ou Gantt, e em bufferização de E/S.

Por exemplo, considere um programa simples de Agenda de compromissos. Este programa permite que você digite um número de eventos; então, conforme cada evento é executado, ele é tirado da lista e a descrição do próximo evento é apresentada. Para simplificar, será usada uma matriz de ponteiros para as strings de eventos e cada descrição de evento será limitada a 255 caracteres. O número de eventos é arbitrariamente limitado a 100.

Primeiro, as funções **qstore()** e **qretrieve()**, mostradas aqui, são necessárias ao programa de agenda. Elas armazenarão ponteiros para as strings que descrevem os eventos.

```
#define MAX 100

char *p[MAX];
int spos = 0;
int rpos = 0;

/* Armazena um evento. */
void qstore(char *q)
{
  if(spos==MAX) {
    printf("Lista cheia\n");
    return;
  }
  p[spos] = q;
  spos++;
}

/* Recupera um evento. */
char *qretrieve()
```

```
{
  if(rpos==spos) {
    printf("Sem eventos.\n");
    return NULL;
  }
  rpos++;
  return p[rpos-1];
}
```

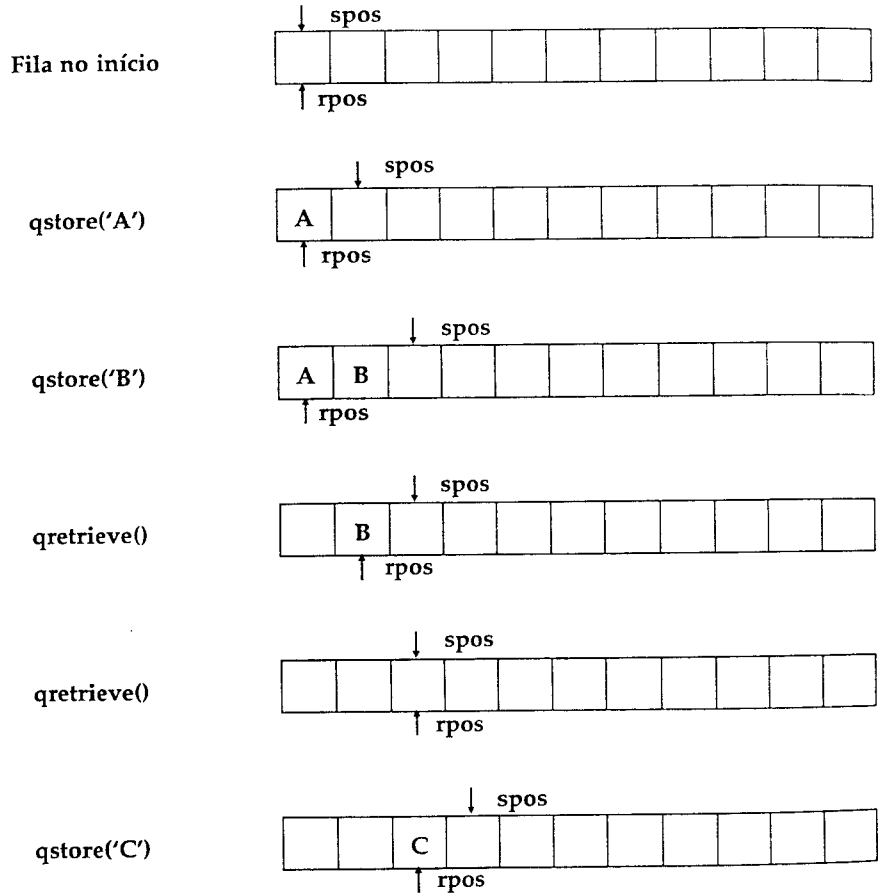


Figura 20.1 O índice de recuperação persegue o índice de armazenamento.

Observe que essas funções requerem duas variáveis globais: **spos** (que contém o índice da próxima posição de armazenamento livre) e **rpos** (que contém o índice do próximo item a ser recuperado). Essas funções podem ser usadas para manter uma fila de outros tipos de dados simplesmente mudando o tipo de base da matriz em que operam.

A função **qstore()** coloca um ponteiro para um novo evento ao final da lista e verifica se a lista está cheia. A função **qretrieve()** tira os eventos da fila enquanto há eventos a executar. Quando um novo evento é escalado, **spos** é incrementado e, quando um evento é completado, **rpos** é incrementado. Em síntese, **rpos** "persegue" **spos** através da fila. Veja na Figura 20.1 como isso aparece na memória enquanto o programa é executado. Se **rpos** e **spos** são iguais, não há eventos a executar. Muito embora a informação armazenada na fila não seja realmente destruída por **qretrieve()**, efetivamente ela é perdida, pois não pode ser acessada novamente.

O programa completo para esse distribuidor simples de eventos é mostrado aqui. Você pode melhorar esse programa para seu próprio uso.

```
/* Miniprograma de agenda */

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

#define MAX 100

char *p[MAX], *qretrieve(void);
int spos = 0;
int rpos = 0;
void enter(void), qstore(char *q), review(void), delete(void);

void main(void)
{
  char s[80];
  register int t;

  for(t=0; t<MAX; ++t) p[t] = NULL; /* inicializa a matriz com
                                       nulos */
  for(;;){
    printf("Inserir, Listar, Remover, Sair: ");
    gets(s);
    *s = toupper(*s);
```

```
    switch(*s) {
      case 'E':
        enter();
        break;
      case 'L':
        review();
        break;
      case 'R':
        delete();
        break;
      case 'Q':
        exit(0);
    }
  }
}

/* Insere um evento na fila. */
void enter(void)
{
  char s[256]; *p;

  do {
    printf("Insira o evento %d: ", spos+1);
    gets(s);
    if(*s==0) break; /* nenhuma entrada */
    p = malloc(strlen(s)+1);
    if(!p) {
      printf("Sem memória.\n");
      return;
    }
    strcpy(p, s);
    if(*s) qstore(p);
  }while(*s);
}

/* Vê o que há na fila. */
void review(void)
{
  register int t;

  for(t=rpos; t<spos; ++t)
    printf("%d. %s\n", t+1, p[t]);
}

/* Apaga um evento da fila. */
void delete(void)
```



```
{
  char *p;

  if((p=qretrieve())==NULL) return;
  printf("%s\n", p);
}

/* Armazena um evento. */
void qstore(char *q)
{

  if(spos==MAX) {
    printf("Lista cheia\n");
    return;
  }
  p[spos] = q;
  spos++;
}

/* Recupera um evento. */
char *qretrieve(void)
{
  if(rpos==spos) {
    printf("Sem eventos.\n");
    return NULL;
  }
  rpos++;
  return p[rpos-1];
}
```

A Fila Circular

Ao estudar a seção anterior, você pode ter pensado em um melhoramento para o programa de agenda. Em lugar de ter uma parada do programa, quando o limite da matriz usada para armazenar a fila é atingido, você poderia ter o índice de armazenamento (**spos**) e o índice de recuperação (**rpos**) retornando ao início da matriz. Dessa forma, qualquer número de itens poderia ser colocado na fila, contanto que itens também estivessem sendo tirados. Essa implementação de fila é chamada *fila circular*, porque usa sua matriz de armazenamento como se fosse um círculo em vez de uma lista linear.

Para criar uma fila circular, para ser usada no programa distribuidor de eventos, as funções **qstore()** e **qretrieve()** precisam ser alteradas como mostrado aqui:

```
void qstore(char *q)
{
  /* A fila está cheia se spos tem um a menos que rpos ou se
     spos está no início da matriz da fila e rpos está no final.
  */
  if(spos+1==rpos || (spos+1==MAX && !rpos)) {
    printf("Lista cheia\n");
    return;
  }

  p[spos] = q;
  spos++;
  if(spos==MAX) spos = 0; /* reinicia */
}

char *qretrieve(void)
{
  if(rpos==MAX) rpos = 0;  /* reinicia */
  if(rpos==spos) {
    printf("Sem eventos para recuperar.\n");
    return NULL;
  }
  rpos++;
  return p[rpos-1];
}
```

Em essência, a fila só está cheia quando os índices de armazenamento e recuperação são iguais; caso contrário, há espaço na fila para outro evento. Conceitualmente, a matriz usada para a versão circular do programa distribuidor de eventos é vista como na Figura 20.2.

Talvez o uso mais comum de fila circular seja em sistemas operacionais, onde a fila circular contém as informações lidas dos arquivos em disco ou do console. Filas circulares também são usadas em programas aplicativos de tempo real, que devem continuar a processar informações enquanto põem as requisições de E/S em um buffer. Muitos editores de texto fazem isso enquanto reformatam um parágrafo ou justificam uma linha. O que está sendo digitado não é mostrado até que o outro processo termine. Para conseguir isso, o programa aplicativo deve verificar a entrada do teclado durante a execução do outro processo. Se uma tecla foi pressionada, ela é rapidamente colocada em uma fila e o processo continua. Uma vez que o processo tenha terminado, os caracteres são recuperados da fila.

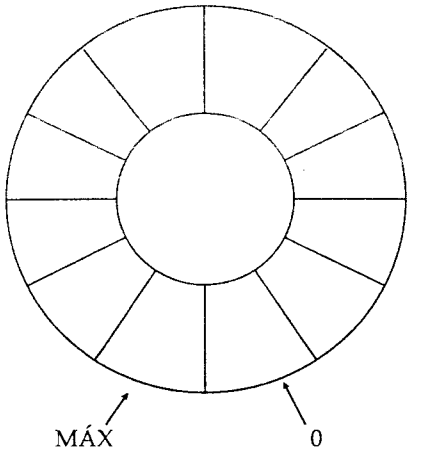


Figura 20.2 A matriz na versão circular do programa de agenda.

Por exemplo, considere um programa simples que contém dois processos. O primeiro processo do programa imprime os números de 1 a 32.000 na tela. O segundo coloca os caracteres em uma fila circular conforme são digitados, sem ecoá-los na tela, até que seja pressionado ENTER. Os caracteres digitados não são mostrados porque o primeiro processo tem prioridade sobre a tela. Após ter sido pressionado ENTER, os caracteres na fila são recuperados e apresentados.

Para que o programa funcione como foi descrito, ele deve usar duas funções não definidas pelo padrão C ANSI: **kbhit()** e **getch()**. A função **kbhit()** devolve verdadeiro se uma tecla, foi pressionada; caso contrário, devolve falso. A função **getch()** lê uma tecla, mas não ecoa o caractere na tela. O padrão C ANSI não define as funções de biblioteca que verificam o estado do teclado ou lêem caracteres do teclado sem ecoá-las na tela porque essas funções são altamente dependentes do sistema operacional. Não obstante, a maioria dos compiladores fornece rotinas que fazem essas coisas. O programa mostrado aqui funciona com a maioria dos compiladores; porém, as duas funções não-padrões podem ter nomes diferentes.

```
/* Um exemplo de fila circular usando um buffer para teclado. */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
```

```
#define MAX 80

char buf[MAX+1];
int spos=0;
int rpos=0;

void qstore(char q);
void qretrieve(void);

void main(void)
{
  register char ch;
  int t;

  buf[80] = NULL;

  /* Insere caracteres até que um CR seja digitado. */
  for(ch=' ', t=0; t<32000 && ch!='\r'; ++t) {
    if(kbhit()) {
      ch = getch();
      qstore(ch);
    }
    printf("%d ", t);
    if(ch=='\r') {
      /* Mostra e esvazia o buffer de teclas. */
      printf("\n");
      while((ch=qretrieve())!=NULL) printf("%c", ch);
      printf("\n");
    }
  }
}

/* Armazena caracteres na fila. */
void qstore(char q)
{
  if(spos+1==rpos || (spos+1==MAX && !rpos)) {
    printf("Lista cheia\n");
    return;
  }
  buf[spos] = q;
  spos++;
  if(spos==MAX) spos = 0; /* reinicia */
}

/* Recupera um caractere. */
char qretrieve(void)
```



```
{
  if(rpos==MAX) rpos = 0;  /* reinicia */
  if(rpos==spos) return NULL;

  rpos++;
  return buf[rpos-1];
}
```

Pilhas

Uma *pilha* é o inverso de uma fila porque usa o acesso *último a entrar, primeiro a sair*, o que algumas vezes é chamado de LIFO (Last In, First Out). Para visualizar uma pilha, imagine uma pilha de pratos. O primeiro prato na mesa é o último a ser usado e o último prato colocado na pilha é o primeiro a ser usado. Pilhas são usadas em grande quantidade em software de sistema, incluindo compiladores e interpretadores. De fato, a maioria dos compiladores C usa uma pilha quando passa argumentos para funções.

As duas operações básicas — armazenar e recuperar — são tradicionalmente chamadas de *push* e *pop*, respectivamente. Assim, para implementar uma pilha, você precisa de duas funções: **push()** (que coloca um valor na pilha) e **pop()** (que recupera um valor da pilha). Você também precisa de uma região de memória para usar como pilha. Você pode usar uma matriz para esse propósito ou alocar uma região de memória, usando as funções de alocação dinâmica de C. Como na fila, a função de recuperação retira um valor da pilha e o destrói se ele não for armazenado em algum outro lugar. As formas gerais de **push()** e **pop()** que usam uma matriz de inteiros são mostradas a seguir. Você pode manter pilhas de outros tipos de dados modificando o tipo de base da matriz em que **push()** e **pop()** operam.

```
int stack[MAX];
int tos=0; /* topo da pilha */

/* Põe um elemento na pilha. */
void push(int i)
{

  if(tos>=MAX) {
    printf("Pilha cheia\n");
    return;
  }
  stack[tos] = i;
  tos++;
```

```
}

/* Recupera o elemento do topo da pilha. */
pop(void)
{
  tos--;
  if(tos<0) {
    printf("Pilha vazia\n");
    return 0;
  }
  return stack[tos];
}
```

A variável **tos** é o índice da próxima posição livre da pilha. Ao implementar essas funções, você deve lembrar-se de evitar o armazenamento da pilha, ultrapassando seus limites (*overflow*), e a tentativa de recuperar dados depois de ter esvaziado a pilha (*underflow*). Nessas rotinas, uma pilha vazia é indicada por **tos** com o valor zero e uma pilha cheia é indicada por **tos** com um valor maior que a última posição de armazenamento. Para ver como uma pilha funciona, veja a Tabela 20.2.

Tabela 20.2 Uma pilha em ação

Ação	Conteúdo da pilha
push(A)	A
push(B)	B A
push(C)	C B A
pop() recupera C	B A
push(F)	F B A
pop() recupera F	B A
pop() recupera B	A
pop() recupera A	*vazia*

Um excelente exemplo do uso de pilha é uma calculadora de quatro funções. A maioria das calculadoras de hoje aceita a forma padrão de expressões chamada de *notação infix*, que toma a forma geral *operando-operador-operando*. Por exemplo, para somar 200 a 100, entre **100**, pressione a tecla de adição (+), digite **200** e, então pressione a tecla de IGUAL (=). Porém, muitas calculadoras antigas usavam uma forma de avaliação de expressão chamada de *notação postfix* (pós-fixada), na qual os dois operandos são inseridos primeiro e, e em seguida, o operador é inserido. Por exemplo, para somar 200 a 100, usando notação pós-fixada, você insere **100**, em seguida **200** e então pressiona a tecla de adição. Conforme os operandos são inseridos, são colocados em uma pilha. Cada vez que um operador é inserido, dois operandos são removidos da pilha e o resultado é colocado de volta na pilha. A vantagem da forma pós-fixada é que expressões muito complexas podem ser facilmente digitadas pelo usuário.

O próximo exemplo demonstra uma pilha implementando uma calculadora posfixa para expressões inteiras. De início, as funções **push()** e **pop()** devem ser modificadas como mostrado a seguir. Elas também usarão memória alocada dinamicamente (em vez de um vetor de tamanho fixo) para a pilha. Embora o uso de memória dinâmica não seja necessário para este exemplo simples, ele ilustra como a memória alocada dinamicamente pode ser usada para implementar uma pilha.

```
int *p;   /* apontará para uma região de memória livre */
int *tos; /* aponta para o topo da pilha */
int *bos; /* aponta para o final da pilha */

/* Armazena um elemento na pilha. */
void push(int i)
{
  if(p>bos) {
    printf("Pilha cheia\n");
    return;
  }
  *p = i;
  p++;
}

/* Recupera o elemento do topo da pilha. */
pop(void)
{
  p--;
  if(p<tos) {
    printf("Pilha vazia\n");
    return 0;
  }
  return *p;
}
```

Antes que essas funções possam ser usadas, uma região de memória livre deve ser alocada com **malloc()**, o endereço do início dessa região deve ser atribuído a **tos**, e o endereço do final, atribuído a **bos**.

O programa completo da calculadora é visto aqui:

```
/* Uma calculadora simples de quatro funções. */

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

#define MAX 100

int *p;   /* apontará para uma região de memória livre */
int *tos; /* aponta para o topo da pilha */
int *bos; /* aponta para o final da pilha */

void push(int i);
int pop(void);

void main(void)
{
  int a, b;
  char s[80];

  p = (int *) malloc(MAX*sizeof(int));/*obtém memória da pilha*/
  if(!p) {
    printf("Falha de alocação\n");
    exit(1);
  }
  tos = p;
  bos = p+MAX-1;

  printf("Calculadora de quatro funções\n");
  printf("Digite 's' para sair\n")

  do {
    printf(": ");
    gets(s);
    switch(*s) {
      case '+':
        a = pop();
        b = pop();
        printf("%d\n", a+b);
        push(a+b);
        break;
      case '-':
        a = pop();
        b = pop();
        printf("%d\n", b-a);
        push(b-a);
```



```
        break;
      case '*':
        a = pop();
        b = pop();
        printf("%d\n", b*a);
        push(b*a);
        break;
      case '/':
        a = pop();
        b = pop();
        if(a==0) {
          printf("divisão por 0\n");
          break;
        }
        printf("%d\n", b/a);
        push(b/a);
        break;
      case '.': /* mostra o conteúdo do topo da pilha */
        a = pop();
        push(a);
        printf("O valor atual da pilha é: %d\n", a);
        break;
      default:
        push(atoi(s));
    }
  } while(*s!='s');
}

/* Põe um elemento na pilha. */
void push(int i)
{
  if(p>bos) {
    printf("Pilha cheia\n");
    return;
  }
  *p = i;
  p++;
}

/* Recupera o elemento do topo da pilha. */
pop(void)
{
  p--;
  if(p<tos) {
```

```
    printf("Pilha vazia\n");
    return 0;
  }
  return *p;
}
```

Listas Encadeadas

Filas e pilhas compartilham duas características comuns: ambas têm regras muito rigorosas para acessar os dados armazenados nelas e as operações de recuperação são, por natureza, destrutivas. Em outras palavras, acessar um item em uma pilha requer a sua remoção e, a menos que o item seja armazenado em outro lugar, ele é destruído. Além disso, tanto pilhas como filas usam uma região contígua de memória. Diferente de uma pilha ou de uma fila, uma lista encadeada pode acessar seu armazenamento de forma randômica, porque cada porção de informação carrega consigo um elo ao próximo item de dados na corrente. Ou seja, uma lista encadeada exige uma estrutura complexa de dados — em oposição à pilha ou fila, que pode operar em itens de dados simples e complexos. Além disso, uma operação de recuperação em uma lista encadeada não remove e destrói um item da lista. Na verdade, você precisa acrescentar uma operação de exclusão específica para isso.

Listas encadeadas podem ser singularmente ou duplamente encadeadas. Uma lista singularmente encadeada contém um elo com o próximo item de dado. Uma lista duplamente encadeada contém elos tanto com o elemento anterior quanto com o próximo elemento da lista. O tipo de lista encadeada utilizada depende de sua aplicação.

Listas Singularmente Encadeadas

Uma lista singularmente encadeada requer que cada item de informação contenha um elo com o próximo elemento da lista. Cada item de dado geralmente consiste em uma estrutura que inclui campos de informação e ponteiro de enlace. Conceitualmente, uma lista singularmente encadeada é vista como mostrado na Figura 20.3.

Basicamente, existem duas maneiras de construir uma lista singularmente encadeada. A primeira é simplesmente colocar cada novo item no início ou final da lista. A outra é acrescentar itens em posições específicas na lista — em ordem crescente, por exemplo. A forma como a lista é construída determina como a função de armazenamento é codificada. Veja o caso mais simples de se criar uma lista encadeada acrescentando itens ao final da lista.

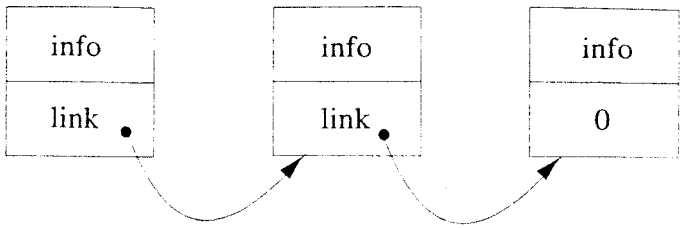


Figura 20.3 Uma lista encadeada na memória.

Antes, você precisa definir uma estrutura de dados que contenha a informação e os elos. Consideremos, neste exemplo, uma lista postal. A estrutura de dados para cada elemento da lista postal é definida aqui:

```
struct address {
  char name[40];
  char street[40];
  char city[20];
  char state[3];
  char zip[11];
  struct address *next;
}info;
```

A função **slstore()** constrói uma lista singularmente encadeada colocando cada novo item no final. Deve ser passado um ponteiro para uma estrutura do tipo **address** e um ponteiro para o último elemento, como mostrado aqui:

```
void slstore(struct address *i,
             struct address **last)
{
  if(!*last) *last = i;  /* primeiro item na lista */
  else (*last)->next = i;
  i->next = NULL;
  *last = i;
}
```

Embora você possa ordenar a lista criada com a função **slstore()** como uma operação separada, é mais fácil ordenar a lista enquanto ela é construída, inserindo cada novo item na seqüência apropriada. Além disso, se a lista já está ordenada, é vantajoso mantê-la assim, inserindo-se os novos itens em suas posições apropriadas. Isso é feito varrendo-se seqüencialmente a lista até que a posição apropriada seja encontrada, inserindo o novo endereço nesse ponto e rearrajando os elos, se necessário.

Três situações possíveis podem ocorrer quando você insere um item em uma lista singularmente encadeada. Primeiro, ele pode tornar-se o novo primeiro item; segundo, ele pode ser inserido entre dois outros itens; terceiro, ele pode tornar-se o último elemento. A Figura 20.4 mostra como os elos são alterados em cada caso.

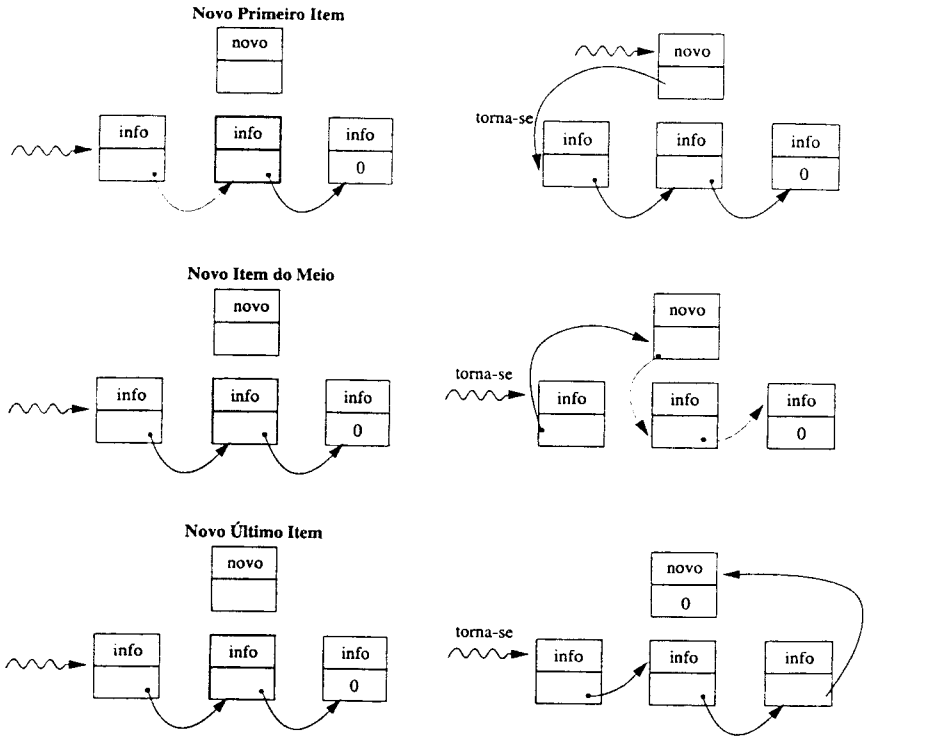


Figura 20.4 Inserindo um item em uma lista singularmente encadeada.

Lembre-se de que, se você alterar o primeiro item da lista, precisará atualizar o ponto de entrada para a lista em outro ponto do seu programa. Para evitar isso, você pode usar uma sentinela como primeiro item. Nesse caso, um valor especial é escolhido, de forma que ele sempre será o primeiro da lista, mantendo, assim, o ponto de entrada. Esse método tem a desvantagem de usar uma posição extra de armazenamento para guardar a sentinela.

A função mostrada a seguir, **sls_store()**, insere endereços na lista postal em ordem crescente, baseando-se no campo **name**. Deve ser passado um ponteiro para o ponteiro do primeiro e do último elemento na lista junto com um ponteiro para a informação a ser armazenada. Como o primeiro e o último elementos podem mudar, **sls_store()** automaticamente atualiza os ponteiros para o início e o final da lista, se eles mudarem. Na primeira vez em que seu programa chama **sls_store()**, **first** e **last** devem apontar para null.

```
/* Armazena em ordem. */
void sls_store(struct address *i, /*novo elemento a armazenar*/
               struct address **start, /*início da lista */
               struct address **last) /*final da lista */
{
  struct address *old, *p;

  p = *start;

  if(!*last) {  /* primeiro elemento da lista */
    i->next = NULL;
    *last = i;
    *start = i;
    return;
  }

  old = NULL;
  while(p) {
    if(strcmp(p->name, i->name)<0) {
      old = p;
      p = p->next;
    }
    else {
      if(old) { /* fica no meio */
        old->next = i;
        i->next = p;
        return;
      }
      i->next = p; /* novo primeiro elemento */
      *start = i;
      return;
    }
  }
  (*last)->next = i; /* coloca no final */
  i->next = NULL;
  *last = i;
}
```

Uma lista encadeada raramente tem uma função dedicada ao processo de recuperação — isto é, para devolver um item após outro na ordem da lista. Normalmente esse código é tão pequeno que é simplesmente colocado dentro de outra rotina como uma função de busca, exclusão ou visualização. Por exemplo, a rotina mostrada aqui apresenta todos os nomes de uma lista postal:

```
void display(struct address *start)
{
  while(start) {
    printf("%s\n", start->name);
    start = start->next;
  }
}
```

Quando **display()** é chamada, **start** deve ser o ponteiro para a primeira estrutura da lista.

Retirar itens de uma lista é tão simples quanto seguir uma seqüência. A rotina de pesquisa baseada no campo **name** poderia ser escrita desta forma:

```
struct address *search(struct address *start, char *n)
{
  while(start) {
    if(!strcmp(n, start->name)) return start;
    start = start->next;
  }
  return NULL; /* não encontrou */
}
```

Como **search()** retorna um ponteiro para um item da lista que coincide com o nome pesquisado, ela deve ser declarada como devolvendo um ponteiro de estrutura do tipo **address**. Se o elemento procurado não for encontrado, é devolvido um nulo.

Apagar um item de uma lista singularmente encadeada é muito simples. Como com a inserção, existem três casos: apagar o primeiro item, apagar um item intermediário e apagar o último item. A Figura 20.5 mostra cada uma dessas operações.

A função a seguir excluirá um item dado de uma lista de estruturas do tipo **address**:

```
void sldelete(
     struct address *p,      /* item anterior */
     struct address *i,      /* item a apagar */
     struct address **start, /* início da lista */
     struct address **last)  /* final da lista */
{
  if(p) p->next = i->next;
  else  *start = i->next;

  if(i==*last && p) *last = p;
}
```

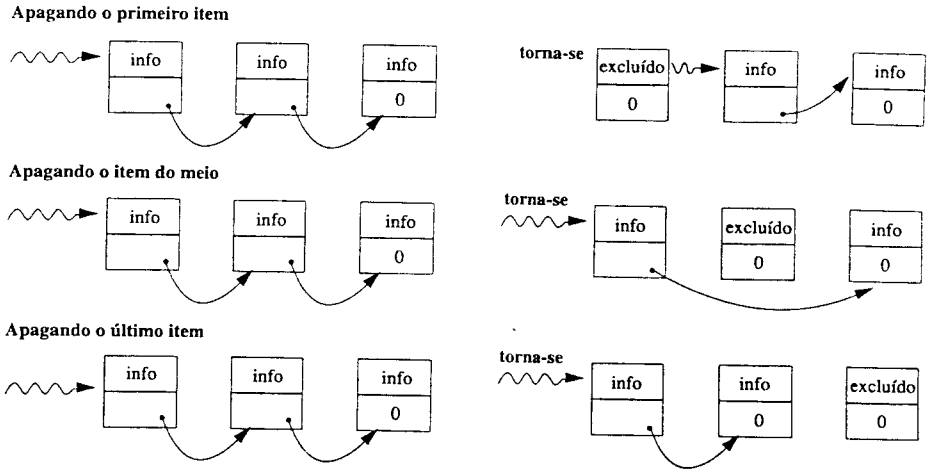



Figura 20.5 Excluindo um item de uma lista singularmente encadeada.

sldelete() deve enviar ponteiros ao item a ser apagado, ao item anterior a esse na seqüência, e ao primeiro e ao último item da lista. Se o primeiro item deve ser removido, o ponteiro anterior deve ser nulo. A função atualiza automaticamente **start** e **last** no caso de um desses pontos ser apagado.

Listas singularmente encadeadas têm uma desvantagem principal que impede seu uso extensivo: a lista não pode ser lida em ordem inversa. Por essa razão, listas duplamente encadeadas são geralmente utilizadas.

Listas Duplamente Encadeadas

Listas duplamente encadeadas consistem em dados e elos para o próximo item e para o item precedente. A Figura 20.6 mostra como esses elos são arranjados.

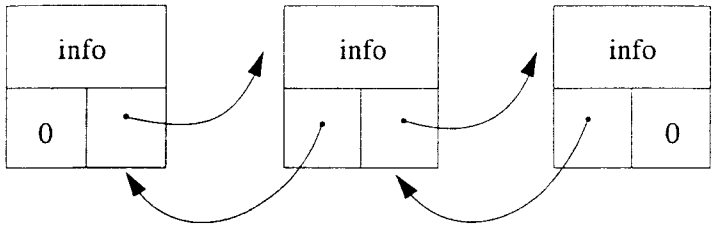


Figura 20.6 Uma lista duplamente encadeada.

O fato de ter dois elos em lugar de um tem duas vantagens principais. Primeiro, a lista pode ser lida em ambas as direções. Isso não apenas simplifica o gerenciamento da lista como também, no caso de um banco de dados, permite ao usuário varrer a lista em ambas as direções. A segunda vantagem só tem sentido no caso de falha do equipamento. Tanto o elo de avanço quanto o de retrocesso podem ler a lista inteira; assim, se um dos elos tornar-se inválido, a lista pode ser reconstruída usando-se o outro.

Um novo elemento pode ser inserido em uma lista duplamente encadeada de três maneiras: inserir um novo primeiro elemento, inserir um elemento intermediário ou inserir um novo último elemento. Essas operações estão ilustradas na Figura 20.7.

A construção de uma lista duplamente encadeada é semelhante à de uma lista singularmente encadeada, exceto pelo fato de que dois elos devem ser mantidos. Assim, a estrutura deve ter espaço para os dois elos. Usando novamente uma lista postal, você pode modificar a estrutura **address**, como mostrado aqui, para acomodar os dois elos:

```
struct address {
  char name[40] ;
  char street[40];
  char city[20];
  char state[3];
  char zip[11];
  struct address *next;
  struct address *prior;
}info;
```

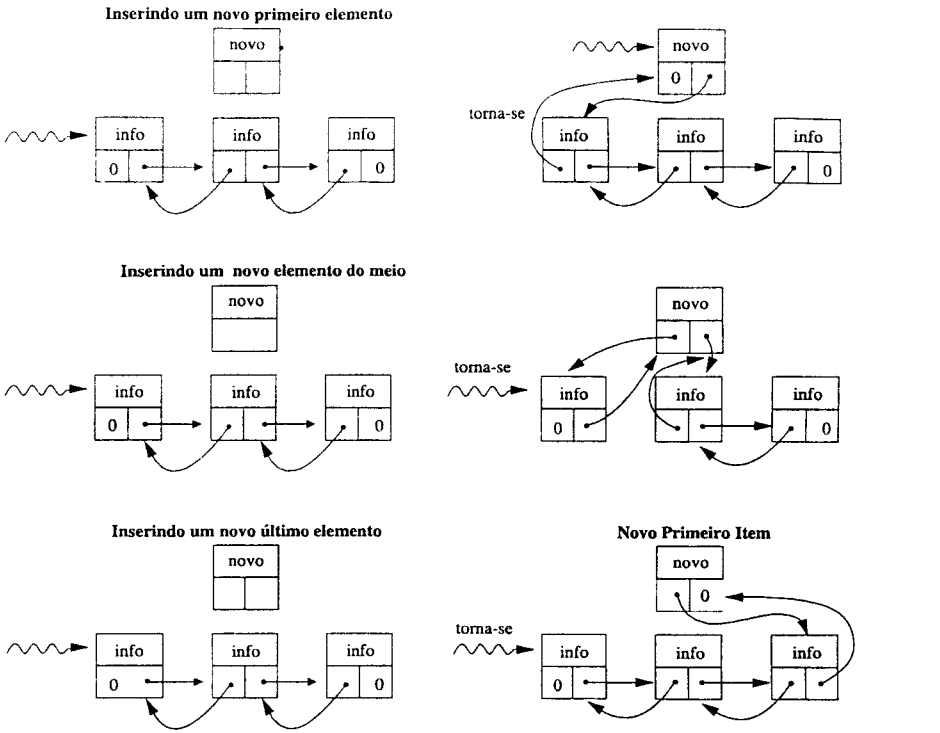



Figura 20.7 Operações em uma lista duplamente encadeada.

Usando **address** como o item de dado básico, a função seguinte, **dlstore()**, constrói uma lista duplamente encadeada:

```
void dlstore(struct adress *i, struct address **last)
{

  if(!*last) *last =i; /* é o primeiro item da lista */
  else (*last)->next = i;
  i->next = NULL;
  i->prior = *last;
  *last = i;
}
```

A função **dlstore()** coloca cada nova entrada no final da lista. Ela deve ser chamada com um ponteiro para os dados a ser armazenado e um ponteiro para o final da lista, que deve ser um nulo na primeira chamada.

Como listas singularmente encadeadas, uma lista duplamente encadeada pode ter uma função que armazena cada novo item em uma posição específica na lista em vez de sempre colocá-lo no final. A função mostrada aqui, **dls_store()**, cria uma lista que é classificada em ordem crescente.

```
/* Cria uma lista duplamente encadeada ordenada. */
void dls_store(
  struct address *i,      /* novo elemento */
  struct address **start, /* primeiro elemento da lista */
  struct address **last   /* último elemento da lista */
)
{
  struct address *old, *p;

  if(*last==NULL) {  /* primeiro elemento da lista */
    i->next = NULL;
    i->prior = NULL;
    *last = i;
    *start = i;
    return;
  }

  p = *start; /* começa no topo da lista */

  old = NULL;
  while(p) {
    if(strcmp(p->name, i->name)<0) {
      old = p;
      p = p->next;
    }
    else {
      if(p->prior) {
        p->prior->next = i;
        i->next = p;
        i->prior = p->prior;
        p->prior= i;
        return;
      }
      i->next = p; /* novo primeiro elemento */
      i->prior = NULL;
```



```
      p->prior = i;
      *start = i;
      return;
    }
  }
  old->next = i; /* coloca no final */
  i->next = NULL;
  i->prior = old;
  *last = i;
}
```

Como o primeiro e o último elemento da lista podem ser alterados, a função **dls_store()** atualiza automaticamente os ponteiros para os elementos inicial e final da lista por meio dos parâmetros **start** e **last**. Você deve chamar a função com um ponteiro para o dado a ser armazenado e um ponteiro para ponteiros para o primeiro e o último elemento da lista. Quando chamados pela primeira vez, os objetos apontados por **first** e **last** devem ser null.

Como nas listas singularmente encadeadas, a recuperação do item de dado específico em uma lista duplamente encadeada é simplesmente o processo de seguir os elos até que o elemento apropriado seja encontrado.

Há três casos a considerar ao excluir um elemento de uma lista duplamente encadeada: excluir o primeiro item, excluir um item intermediário ou excluir o último item. A Figura 20.8 mostra como os elos são rearranjados. A função **dldelete()**, mostrada aqui, exclui um item de uma lista duplamente encadeada.

```
void dldelete(
  struct address *i, /* item a apagar */
  struct address **start, /* primeiro item */
  struct address **last) /* último item */
{
  if(i->prior) i->prior->next = i->next;
  else { /* novo primeiro item */
    *start = i->next;
    if(start) start->prior = NULL;
  }

  if(i->next) i->next->prior = i->prior;
  else  /* apaga o último elemento */
    *last = i->prior;
}
```

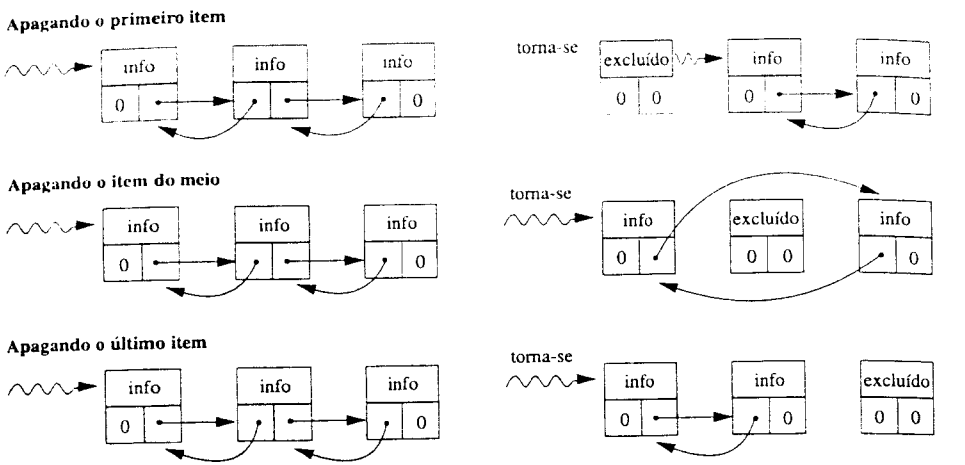


Figura 20.8 Exclusão de uma lista duplamente encadeada.

Como o primeiro ou último elemento na lista podem ser apagados, a função **dldelete()** automaticamente atualiza os ponteiros para os elementos inicial e final da lista por meio dos parâmetros **start** e **last**. Você deve chamar a função com um ponteiro para o dado a ser apagado e um ponteiro para ponteiros para o primeiro e último itens da lista.

Um Exemplo de Lista Postal

Para finalizar a discussão de listas duplamente encadeadas, esta seção apresenta um programa simples, porém completo, de lista postal. Toda a lista é mantida na memória enquanto é usada. No entanto, ela pode ser armazenada em um arquivo em disco e carregada para uso posterior.

```
/* Um programa simples de lista postal que ilustra o
   uso e a manutenção de listas duplamente encadeadas.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct address {
```



```
  char name[30] ;
  char street[40];
  char city[20];
  char state[3];
  char zip[11];
  struct address *next; /* ponteiro para a próxima entrada */
  struct address *prior; /* ponteiro para o registro anterior */
} list_entry;

struct address *start;  /* ponteiro para a primeira entrada
                           da lista */
struct address *last;  /* ponteiro para a última entrada */
struct address *find(char *);

void enter(void), search(void), save(void);
void load(void), list(void);
void delete(struct address **, struct address **);
void dls_store (struct address *i, struct address **start,
                struct address **last);
void inputs(char *, char *, int), display(struct address *);
int menu_select(void);

void main(void)
{
  start = last = NULL;  /* inicializa os ponteiros de início
                           e fim */
  for(;;) {
    switch(menu_select()) {
      case 1: enter();
        break;
      case 2: delete(&start, &last);
        break;
      case 3: list();
        break;
      case 4: search(); /* encontra uma rua */
        break;
      case 5: save(); /* grava a lista no disco */
        break;
      case 6: load(); /* lê do disco */
        break;
      case 7: exit(0);
    }
  }
}
```

```
/* Seleciona uma operação. */
menu_select(void)
{
  char s[80];
  int c;

  printf("1. Inserir um nome\n");
  printf("2. Deletar um nome\n");
  printf("3. Listar o arquivo\n");
  printf("4. Pesquisar\n");
  printf("5. Gravar o arquivo\n");
  printf("6. Carregar o arquivo\n");
  printf("7. Sair\n");
  do {
    printf("\nInsira sua escolha: ");
    gets(s);
    c =atoi(s);
  } while(c<0 || c>7);
  return c;
}

/* Insere nomes e endereços. */
void enter(void)
{
  struct address *info;

  for(;;) {
    info = (struct address *)malloc(sizeof(list_entry));
    if(!info) {
      printf("\nsem memória");
      return;
    }

    inputs("Insira o nome: ", info->name, 30);
    if(!info->name[0]) break; /* não efetua a inserção */
    inputs("Insira a rua: ", info->street, 40);
    inputs("Insira a cidade: ", info->city, 20);
    inputs("Insira o estado: ", info->state, 3);
    inputs("Insira o cep: ", info->zip, 10);

    dls_store(info, &start, &last);
  } /* laço de entrada */
}
```



```
/* Essa função lê uma string de comprimento máximo count e
   evita que a string seja ultrapassada. Ela também apresenta
   uma mensagem de aviso. */
void inputs(char *prompt, char *s, int count)
{
  char p[255];

  do {
    printf(prompt);
    gets(p);
    if(strlen(p)>count) printf("\nmuito longo\n");
  } while(strlen(p)>count);
  strcpy(s, p);
}

/* Cria uma lista duplamente encadeada ordenada. */
void dls_store(
  struct address *i, /* novo elemento */
  struct address **start, /* primeiro elemento da lista */
  struct address **last  /* último elemento da lista */
)

{
  struct address *old, *p;

  if(*last==NULL) {  /* primeiro elemento da lista */
    i->next = NULL;
    i->prior = NULL;
    *last = i;
    *start = i;
    return;
  }
  p = *start; /* começa no topo da lista */

  old = NULL;
  while(p) {
    if(strcmp(p->name, i->name)<0) {
      old = p;
      p = p->next;
    }
    else {
      if(p->prior) {
        p->prior->next = i;
        i->next = p;
        i->prior = p->prior;
```

```
        p->prior= i;
        return;
      }
      i->next = p; /* novo primeiro elemento */
      i->prior = NULL;
      p->prior = i;
      *start = i;
      return;
    }
  }
  old->next = i; /* coloca no final */
  i->next = NULL;
  i->prior = old;
  *last = i;
}

/* Remove um elemento da lista. */
void delete(struct address **start, struct address **last)
{
  struct address *info, *find();
  char s[80];

  printf("insira o nome: ", s, 30);
  info = find(s);
  if(info) {
    if(*start==info) {
      *start=info->next;
      if(*start) (*start)->prior = NULL;
      else *last = NULL;
    }
    else {
      info->prior->next = info->next;
      if(info!=*last)
          info->next->prior = info->prior;
      else
        *last = info->prior;
    }
    free(info); /* devolve memória para o sistema */
  }
}

/* Encontra um endereço. */
struct address *find(char *name)
{
  struct address *info;
```



```
  info = start;
  while(info) {
    if(!strcmp(name, info->name)) return info;
    info = info->next;  /* obtém novo endereço */
  }
  printf("Nome não encontrado.\n");
  return NULL; /* não encontrou */
}

/* Mostra a lista completa. */
void list(void)
{
  struct address *info;

  info = start;
  while(info) {
    display(info);
    info = info->next;   /* obtém próximo endereço */
  }
  printf("\n\n");
}

/* Essa função imprime os campos de cada endereço.*/
void display(struct address *info)
{
  printf("%s\n", info->name);
  printf("%s\n", info->street);
  printf("%s\n", info->city);
  printf("%s\n", info->state);
  printf("%s\n", info->zip);
  printf("\n\n");
}

/* Procura por um nome na lista.*/
void search(void)
{
  char name[40];
  struct address *info, *find();

  printf("Insira o nome a procurar: ");
  gets(name);
  info = find(name);
  if(!info) printf("Não encontrado\n");
  else display(info);
}
```

```
/* Salva o arquivo em disco. */
void save(void)
{
  struct address *info;

  FILE *fp;

  fp = fopen("mlist", "wb");
  if(!fp) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }
  printf("\nsalvando arquivo\n");

  info = start;
  while(info) {
    fwrite(info, sizeof(struct address), 1, fp);
    info = info->next; /* obtém próximo endereço */
  }
  fclose(fp);
}

/* Carrega o arquivo de endereço. */
void load()
{
  struct address *info;
  FILE *fp;

  fp = fopen("mlist", "rb");
  if(!fp) {
    printf("O arquivo não pode ser aberto.\n");
    exit(1);
  }

  /* libera qualquer memória previamente alocada */
  while(start) {
    info = start->next;
    free(info);
    start = info;
  }

  /* reinicializa os ponteiros de início e fim */
  start = last = NULL;

  printf("\nloading file\n");
  while(!eof(fp)) {
    info = (struct address *) malloc(sizeof(struct address));
    if(!info) {
      printf("sem memória");
      return;
    }
    if(1!=fread(info, sizeof(struct address), 1, fp)) break;
    dls_store(info, &start, &last);
  }
  fclose(fp);
}
```


Árvores Binárias

A última estrutura de dados a ser examinada é a *árvore binária*. Embora possa haver muitos tipos diferentes de árvores, árvores binárias são especiais porque, quando ordenadas, elas conduzem a pesquisas, inserções e exclusões rápidas. Cada item em uma árvore consiste em informação juntamente com um elo ao membro esquerdo e um elo ao membro direito. A Figura 20.9 mostra uma pequena árvore.

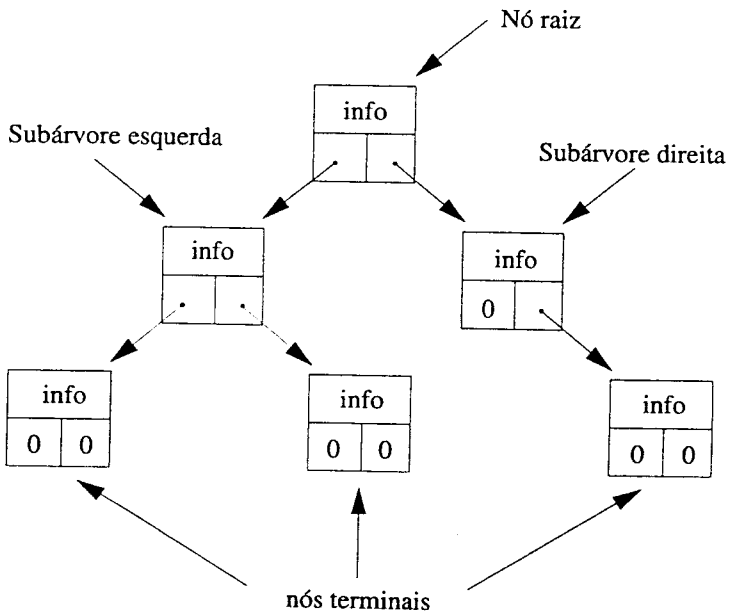


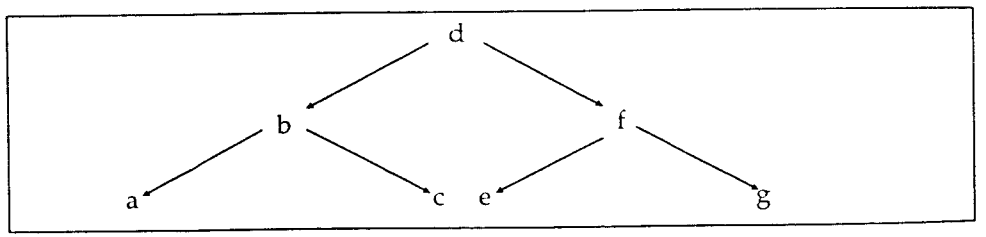
Figura 20.9 Uma amostra de árvore binária de altura três

Uma terminologia especial é necessária quando se discutem árvores. Cientistas de computadores não são conhecidos por sua gramática e a terminologia para árvores é um caso clássico de metáforas confusas. A *raiz* é o primeiro item em uma árvore. Cada item de dado é chamado de *nó* (ou, às vezes, de folha) da árvore e qualquer parte da árvore é chamada de uma *subárvore*. Um nó que não tem subárvores ligadas a ele é chamado de *nó terminal*. A *altura* da árvore é igual ao número de camadas que as raízes atingem. Imagine uma árvore binária como elas aparecem no papel. Lembre-se, porém, de que uma árvore é apenas uma maneira de estruturar dados na memória e que a memória tem sempre formato linear.

Em certo sentido, a árvore binária é uma forma especial de lista encadeada. Pode-se inserir, excluir e acessar itens em qualquer ordem. Além disso, a operação de recuperação é não-destrutiva. Embora árvores sejam fáceis de visualizar, elas apresentam alguns difíceis problemas de programação, que esta seção apenas introduzirá.

A maioria das funções que usam árvores é recursiva porque a própria árvore é uma estrutura de dados recursiva. Isto é, cada subárvore é ela própria uma árvore. Assim, as rotinas desenvolvidas nessa discussão serão recursivas. Existem versões não-recursivas dessas funções, mas seu código é muito mais difícil de entender.

A forma como uma árvore é ordenada depende de como ela será referenciada. O procedimento de acesso a cada nó na árvore é chamado de *transversalização da árvore*. Considere a seguinte árvore:



Existem três maneiras de percorrer uma árvore: de forma *ordenada*, *preordenada* e *pós-ordenada*. Usando a forma ordenada, você visita a subárvore da esquerda, a raiz e, em seguida, a subárvore da direita. Na maneira preordenada, você visita a raiz, a subárvore da esquerda e, em seguida, a subárvore da direita. Com a pós-ordenada, você visita a subárvore da esquerda, a subárvore da direita e, depois, a raiz. A ordem de acesso à árvore usando cada método é

ordenada	a b c d e f g
preordenada	d b a c f e g
pós-ordenada	a c b e g f d

Embora uma árvore não precise ser ordenada, a maioria dos usos exige isso. Obviamente, o que constitui uma árvore binária depende de como a árvore será transversalizada. O restante deste capítulo assume a forma ordenada. Portanto, uma árvore binária ordenada é aquela em que a subárvore da esquerda contém nós que são menores ou iguais à raiz e os da direita são maiores que a raiz.

A função seguinte, **stree()**, constrói uma árvore binária ordenada:

```
struct tree {
  char info;
  struct tree *left;
  struct tree *right;
};

struct tree *stree {
  struct tree *root,
  struct tree *r,
  char info)
{
  if(!r) {
    r = (struct tree *) malloc(sizeof(struct tree));
    if(!r) {
      printf("sem memória\n");
      exit(0);
    }
    r->left = NULL;
    r->right = NULL;
    r->info = info;
    if(!root) return r;  /* primeira entrada */
    if(info<root->info) root->left = r;
    else root->right = r;
    return r;
  }
  if(info<r->info) stree(r, r->left, info);
  else
    stree(r, r->right, info);
}
```

O algoritmo anterior simplesmente segue os elos através da árvore indo para a esquerda ou para a direita baseado no campo **info**. Para utilizar essa função, você precisa de uma variável global que contenha a raiz da árvore. Essa variável deve ser inicializada com **NULL** e um ponteiro à raiz será definido na primeira chamada a **stree()**. Chamadas subseqüentes a **stree()** não necessitam redefinir a raiz. Assumindo que o nome dessa variável global seja **rt**, para chamar a função **stree()** você usaria:

```
/* chama stree() */
if(!rt) rt = stree(rt, rt, info);
else stree(rt, rt, info);
```

Dessa forma, tanto o primeiro quanto os elementos subseqüentes podem ser inseridos corretamente.

A função **stree()** é um algoritmo recursivo, como a maioria das rotinas de árvore. A mesma rotina seria várias vezes maior se fossem empregados métodos puramente iterativos. A função deve ser chamada com um ponteiro para a raiz, o nó esquerdo ou direito e a informação. Para simplificar, apenas um caractere é usado como informação. Porém, você poderia substituir por qualquer tipo de dado, simples ou complexo.

Para percorrer a árvore construída por **stree()** de forma ordenada e imprimir o campo **info** de cada nó, você poderia usar a função **inorder()**, mostrada aqui:

```
void inorder(struct tree *root)
{
  if(!root) return;

  inorder(root->left);
  if(root->info) printf("%c ", root->info);
  inorder(root->right);
}
```

Essa função recursiva retorna quando um nó terminal (um ponteiro nulo) é encontrado.

As funções que percorrem a árvore de forma preordenada e pós-ordenada são mostradas na listagem seguinte.

```
void preorder(struct tree *root)
{
  if(!root) return;

  if (root->info) printf("%c ", root->info);
  preorder(root->left);
  preorder(root->right);
}

void postorder(struct tree *root)
{
  if(!root) return;
```



```
  postorder(root->left);
  postorder(root->right);
  if(root->info) printf("%c ", root->info);
}
```

Neste momento, considere um programa simples, porém interessante, para construir uma árvore binária e imprimir a árvore lateralmente na tela. O programa requer apenas uma pequena modificação na função **inorder()**. Essa nova função é chamada de **print_tree()**, como mostrado aqui:

```
void print_tree(struct tree *r, int l)
{
  int i;
  if(r==NULL) return;

  print_tree(r->right, l+1);
  for(i=0; i<l; ++i) printf("   ");
  printf("%c\n", r->info);
  print_tree(r->left, l+1);
}
```

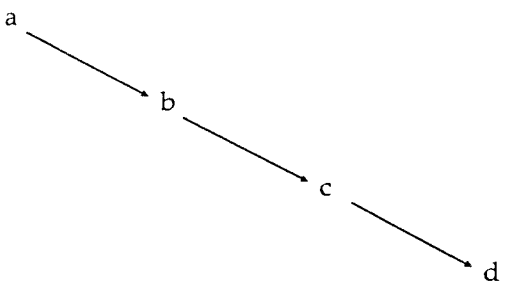
O programa completo de impressão de árvores é visto a seguir. Tente inserir várias árvores para ver como cada uma é construída.

```
/* Este programa mostra uma árvore binária. */

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

struct tree {
  char info;
  struct tree *left;
  struct tree *right;
};

struct tree *root; /* primeiro nó da árvore */
struct tree *stree(struct tree *root,
                   struct tree *r, char info);
void print_tree(struct tree *root, int l);

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

  root = NULL; /* inicializa a raiz */
```

```
  do {
    printf("insira uma letra: ");
    gets(s);
    if(!root) root = stree(root, root, *s);
    else stree(root, root, *s);
  } while(*s);
  print_tree(root, NULL);
}

struct tree *stree(
  struct tree *root,
  struct tree *r,
  char info)
{

  if(!r) {
    r = (struct tree *) malloc(sizeof(struct tree));
    if(!r) {
      printf("Sem memória\n");
      exit(0);
    }
    r->left = NULL;
    r->right = NULL;
    r->info = info;
    if(!root) return r; /* primeira entrada */
    if(info<root->info) root->left = r;
    else root->right = r;
    return r;
  }

  if(info<r->info) stree(r, r->left, info);
  else
    stree(r, r->right, info);
}

void print_tree(struct tree *r, int l)
{
  int i;

  if(!r) return;

  print_tree(r->right, l+1);
  for(i=0; i<l; ++i) printf("  ");
  printf("%c\n", r->info);
  print_tree(r->left, l+1);
}
```


Este programa, na verdade, ordena a informação que lhe é dada. Ele é essencialmente uma variação da ordenação por inserção, que foi vista no capítulo anterior. No caso médio, seu desempenho pode ser muito bom, mas o quicksort ainda é uma melhor ordenação de uso geral, porque usa menos memória e tem um tempo de processamento menor. Porém, se você precisa construir uma árvore desde o princípio ou manter uma árvore já ordenada, deverá sempre inserir novas entradas ordenadamente, usando a função **stree()**.

Se você executou o programa de impressão de árvore, provavelmente notou que algumas árvores são *balanceadas* — isto é, cada subárvore tem a mesma ou quase a mesma altura que qualquer outra — e que outras não são balanceadas. De fato, se você inserisse a árvore **abcd**, ela seria criada desta forma:



Não haveria subárvore da direita. Essa situação é chamada de *árvore degenerada*, porque ela se degenerou em uma lista linear. Em geral, se os dados que você está usando como entrada, para construir uma árvore binária, são razoavelmente aleatórios, a árvore produzida se aproxima de uma árvore balanceada. Porém, se a informação já está ordenada, o resultado é uma árvore degenerada. (É possível reajustar a árvore a cada inserção para manter a árvore em equilíbrio, mas os algoritmos são um tanto complexos. Os leitores interessados devem procurar livros sobre algoritmos avançados de programação.)

Funções de pesquisa para árvores binárias são fáceis de implementar. A função seguinte devolve um ponteiro para o nó da árvore que coincide com a chave; caso contrário, ela devolve um nulo.

```
struct tree *search_tree(struct tree *root, char key)
{
  if (!root) return root; /* not found*/
  if(!root) return root;  /* árvore vazia */
  while(root->info!=key) {
```

```
    if(key<root->info) root = root->left;
    else root = root->right;
    if(root==NULL) break;
  }
  return root;
}
```

Infelizmente, apagar um nó de uma árvore não é tão simples quanto fazer buscas na árvore. O nó excluído pode ser uma raiz, um nó esquerdo ou um nó direito. Além disso, o nó pode ter de zero a duas subárvores ligadas a ele. O processo de rearranjar os ponteiros nos conduz a um algoritmo recursivo, que é mostrado aqui:

```
struct tree *dtree(struct tree *root, char key)
{
  struct tree *p, *p2;

  if(!root) return root; /* não encontrado */

  if(root->info==key) { /* apagar a raiz */
    /* isso significa uma árvore vazia */
    if(root->left==root->right) {
      free(root);
      return NULL;
    }
    /* ou se uma subárvore é nula */
    else if(root->left==NULL) {
      p = root->right;
      free(root);
      return p;
    }
    else if(root->right==NULL) {
      p = root->left;
      free(root);
      return p;
    }
    /* ou as duas subárvores estão presentes */
    else {
      p2 = root->right;
      p = root->right;
      while(p->left) p = p->left;
      p->left = root->left;
      free(root);
```



```
      return p2;
    }
  }
  if(root->info<key) root->right = dtree(root->right, key);
  else root->left = dtree(root->left, key);
  return root;
}
```

Lembre-se de atualizar o ponteiro para a raiz no resto do código do seu programa, porque o nó apagado pode ser a raiz da árvore. A melhor maneira de fazer isto é atribuindo o valor de retorno de **dtree()** à variável em seu programa que aponta para a raiz, usando uma chamada similar à seguinte:

```
root = dtree(root, key);
```

Árvores binárias oferecem grande poder, flexibilidade e eficiência quando usadas em programas de gerenciamento de banco de dados. Isso ocorre porque a informação para esses bancos de dados deve residir em disco e os tempos de acesso são importantes. Como uma árvore balanceada tem, no pior caso, $\log_2 n$ comparações em uma pesquisa, ela se comporta melhor que uma lista encadeada, que depende de uma busca seqüencial.

21

Matrizes Esparsas

Um dos problemas mais intrigantes em programação é a implementação de uma matriz esparsa. Uma *matriz esparsa* é aquela em que nem todos os elementos estão realmente presentes ou são necessários. As matrizes esparsas são valiosas quando as duas condições seguintes são atendidas: as dimensões de uma aplicação são relativamente grandes (possivelmente excedendo a memória disponível), e quando nem todas as posições da matriz serão usadas. Lembre-se de que matrizes — especialmente matrizes multidimensionais — podem consumir enormes quantidades de memória, porque suas necessidades de armazenamento estão exponencialmente relacionadas com seu tamanho. Por exemplo, uma matriz de caracteres de 10 por 10 precisa de apenas 100 bytes de memória, uma matriz 100 por 100 necessita de 10.000, mas uma matriz de 1.000 por 1.000 necessita de 1.000.000 de bytes de memória — nitidamente um número muito grande para a maioria dos computadores.

Há numerosos exemplos de aplicações que exigem o processamento de matrizes esparsas. Muitas se aplicam a problemas científicos ou de engenharia que só são facilmente entendidos por peritos. No entanto, há uma aplicação muito familiar que usa matriz esparsa: um programa de planilha. Muito embora a matriz de uma planilha comum seja muito grande, digamos 999 por 999, apenas uma porção da matriz pode realmente estar sendo usada em um dado momento. Planilhas usam a matriz para guardar fórmulas, valores e strings associados a cada posição. Em uma matriz esparsa, o armazenamento para cada elemento é alocado de um espaço de memória livre conforme se torne necessário. Embora apenas uma pequena porção dos elementos esteja realmente sendo usada, a matriz pode parecer muito grande — maior do que o que normalmente caberia na memória do computador.

O restante dessa discussão usa os termos *matriz lógica* e *matriz física*. A *matriz lógica* é a matriz que você pensa que existe no sistema. Por exemplo, se uma matriz em uma planilha de cálculo tem dimensões de 1000 x 1000, então a matriz lógica que suporta a matriz também tem dimensões de 1000 x 1000, embora esta matriz não exista fisicamente dentro do computador. A matriz física é a matriz que realmente existe dentro do computador. Assim, se somente 100 elementos da matriz da planilha estão em uso, a matriz física estará usando espaço apenas para esses 100 elementos. As técnicas de matriz esparsa desenvolvidas neste capítulo oferecem a ligação entre as matrizes lógicas e físicas.

Este capítulo examina quatro técnicas distintas para criar uma matriz esparsa: a lista encadeada, a árvore binária, uma matriz de ponteiros e fragmentação. Embora nenhum programa de planilha seja desenvolvido, todos os exemplos dizem respeito a uma matriz de planilha que é organizada como mostrado na Figura 21.1. Nela, o X está localizado na célula B2.

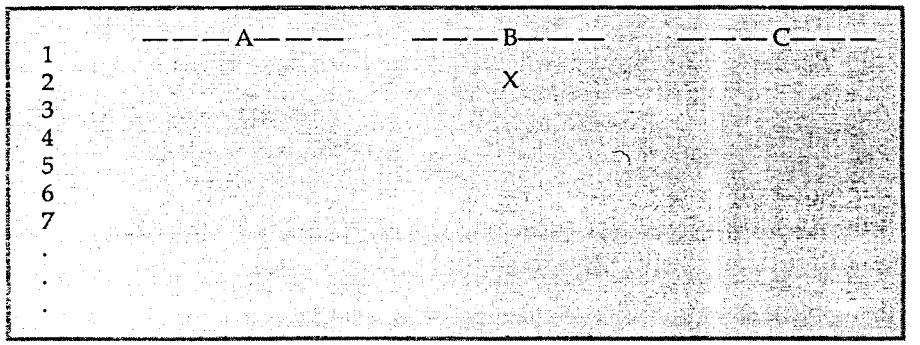


Figura 21.1 A organização de uma planilha.

A Matriz Esparsa com Lista Encadeada

Quando você implementa uma matriz esparsa usando uma lista encadeada, você deve criar uma estrutura que contenha os seguintes itens:

- Os dados que estão sendo armazenados
- Sua posição lógica dentro da matriz

■ Ligações com o elemento anterior e próximo

Cada estrutura é colocada na lista com os elementos inseridos de forma ordenada, baseada no índice da matriz. A matriz é acessada seguindo-se os elos.

Por exemplo, você pode usar a seguinte estrutura como base de uma matriz esparsa para ser usada em um programa de planilha:

```
struct cell {
  char cell_name[9];  /* nome da célula p.e. A1, B34 */
  char formula[128];  /* p.e. 10/B2 */
  struct cell *next;  /* ponteiro para a próxima entrada */
  struct cell *prior; /* ponteiro para o registro anterior */
} list_entry;
```

O campo **cell_name** possui uma string que contém o nome da célula como A1, B34, Z19 etc. A string **formula** contém a fórmula que é atribuída a cada posição da planilha.

Um programa de planilha completo seria muito grande para usar como exemplo. Por isso, este capítulo examina as funções básicas que suportam a matriz esparsa com lista encadeada. Lembre-se de que há muitas maneiras de implementar um programa de planilha. A estrutura de dados e as rotinas apresentadas aqui são apenas exemplos de técnicas de matrizes esparsas.

As seguintes variáveis globais apontam para o início e o final da matriz com lista encadeada.

```
struct cell *start;  /* primeiro elemento da lista */
struct cell *last;   /* último elemento da lista */
```

Na maioria das planilhas, ao inserir uma fórmula em uma célula, você está, na verdade, criando um novo elemento da matriz esparsa. Se a planilha usa uma lista encadeada, cada nova célula é inserida via uma função semelhante a **dls_store()**, que foi desenvolvida no Capítulo 20. Lembre-se de que a lista é ordenada de acordo com o nome da célula; ou seja, A12 precede A13 e assim por diante.

```
/* Armazena células de forma ordenada. */
void dls_store(struct cell *i,
               struct cell **start,
               struct cell **last)

{
  struct cell *old, *p;
```



```
  if(!*last) {  /* primeiro elemento da lista */
    i->next = NULL;
    i->prior = NULL;
    *last = i;
    *start = i;
    return;
  }

  p = *start;  /* começa do topo da lista */

  old = NULL;
  while(p) {
    if(strcmp(p->cell_name, i->cell_name)<0) {
      old = p;
      p = p->next;
    }
    else {
      if(p->prior) { /* é um elemento intermediário */
        p->prior->next = i;
        i->next = p;
        i->prior = p->prior;
        i->prior = i;
        return;
      }
      i->next = p;  /* novo primeiro elemento */
      i->prior = NULL;
      p->prior = i;
      *start = i;
      return;
    }
  }
  old->next = i;  /* põe no final */
  i->next = NULL;
  i->prior = old;
  *last = i;
  return;
}
```

A função **delete()**, mostrada a seguir, remove da lista a célula cujo nome é um argumento para a função:

```
void delete(char *cell_name,
            struct cell **start,
            struct cell **last)
{
```

```
    struct cell *info;

    info = find(cell_name, *start);
    if(info) {
      if(*start==info) {
        *start = info->next;
        if(*start) (*start)->prior = NULL;
        else *last = NULL;
      }
      else {
        if(info->prior) info->prior->next = info->next;
        if(info!=*last)
            info->next->prior = info->prior;
        else
          *last = info->prior;
      }
      free(info); /* devolve memória ao sistema */
    }
}
```

A última função de que você precisa para suportar uma matriz esparsa com lista encadeada é **find()**, que localiza uma célula específica. A função requer uma pesquisa linear para localizar cada item e, como foi visto no Capítulo 19, o número médio de comparações em uma pesquisa linear é $n/2$, onde n é o número de elementos na lista. **find()** é mostrada aqui:

```
struct cell *find(char *cell_name, struct cell *start)
{
  struct cell *info;

  info = start;
  while(info) {
    if(!strcmp(cell_name, info->cell_name)) return info;
    info = info->next;  /* obtém a próxima célula */
  }
  printf("Célula não encontrada.\n");
  return NULL;  /* não encontrou */
}
```

Análise da Abordagem com Lista Encadeada

A principal vantagem do método da lista encadeada é que ele faz um uso eficiente da memória. No entanto, há uma grande desvantagem: ele precisa usar uma pesquisa linear para acessar as células na lista. Sem o uso de informação adicional, que exige memória adicional, não há como realizar uma pesquisa binária para localizar uma célula. Além disso, a rotina de armazenamento usa uma pesquisa linear para encontrar o lugar apropriado para inserir a nova célula na lista. Esses problemas podem ser solucionados usando-se uma árvore binária para operar a matriz esparsa.

A Abordagem com Árvore Binária para Matriz Esparsa

Em essência, a árvore binária é simplesmente um lista duplamente encadeada modificada. Sua principal vantagem sobre uma lista é que ela pode ser varrida rapidamente, o que significa que as inserções e consultas podem ser muito rápidas. Em aplicações nas quais você deseja uma estrutura de lista encadeada, mas precisa de pesquisas rápidas, a árvore binária é perfeita.

Para usar uma árvore binária na operação do exemplo da planilha, você deve modificar a estrutura **cell**, como mostrado no código a seguir:

```
struct cell {
  char cell_name[9];  /* nome da célula p.e. A1, B34 */
  char formula[128];  /* p.e. 10/B2 */
  struct cell *left;  /* ponteiro para a subárvore esquerda */
  struct cell *right; /* ponteiro para a subárvore direita */
} list_entry;
```

Você pode modificar a função **stree()**, do Capítulo 20, de forma que ela construa uma árvore baseada no nome da célula. Observe que ela assume que o parâmetro **new** é um ponteiro para uma nova entrada da árvore.

```
struct cell *stree(
       struct cell *root,
       struct cell *r,
       struct cell *new)
{
  if(!r) { /* primeiro nó em uma subárvore */
    new->left = NULL;
    new->right = NULL;
    if(!root) return new; /* primeira entrada na árvore */
    if(strcmp(new->cell_name, root->cell_name)<0)
      root->left = new;
```

```
    else
      root->right = new;
    return new;
  }

  if(strcmp(r->cell_name, new->cell_name)<=0)
    stree(r, r->right, new);
  else
    stree(r, r->left, new);

  return root;
}
```

A função **stree()** deve ser chamada com um ponteiro para o nó raiz, para os dois primeiros parâmetros, e um ponteiro para a nova célula no terceiro. Ela devolve um ponteiro para a raiz.

Para apagar uma célula da planilha, modifique a função **dtree()**, como mostrado aqui, para aceitar o nome de uma célula como uma chave:

```
struct cell *dtree(
        struct cell *root,
        char *key)

{
  struct cell *p, *p2;

  if (!root) return root; /* item não encontrado */

  if(!strcmp(root->cell_name,key)) { /* apagar a raiz */
    /* isto significa uma árvore vazia */
    if(root->left==root->right) {
      free(root);
      return NULL;
    }

    /* ou se uma subárvore é nula */
    else if(root->left==NULL) {
      p = root->right;
      free(root);
      return p;
    }
    else if(root->right==NULL) {
      p = root->left;
      free(root);
      return p;
    }
```



```
    /* ou as duas subárvores estão presentes */
    else {
      p2 = root->right;
      p = root->right;
      while(p->left) p = p->left;
      p->left = root->left;
      free(root);
      return p2;
    }
  }
  if(strcmp(root->cell_name, key)<=0)
    root->right = dtree(root->right, key);
  else root->left = dtree(root->left, key);
  return root;
}
```

Finalmente, você pode usar uma versão modificada de **search()** para localizar rapidamente qualquer célula na planilha, especificando o nome da célula.

```
struct cell *search_tree(
        struct cell *root,
        char *key)
{
  if(!root) return root;  /* árvore vazia */
  while(strcmp(root->cell_name,key)) {
    if(strcmp(root->cell_name, key)<=0)
      root = root->right;
    else root = root->left;
    if(root==NULL) break;
  }
  return root;
}
```

Análise da Abordagem com Árvores Binárias

Uma árvore binária resulta em tempos de pesquisa muito mais rápidos que uma lista encadeada. Lembre-se de que uma pesquisa seqüencial requer, em média, $n/2$ comparações, onde n é o número de elementos na lista, ao passo que uma árvore binária requer apenas $\log_2 n$ comparações. Além disso, a árvore binária usa a memória tão eficientemente quanto uma lista duplamente encadeada. No entanto, em algumas situações existem alternativas ainda melhores.

A Abordagem com Matriz de Ponteiros para Matriz Esparsa

Suponha que sua planilha tenha as dimensões 26 por 100 (A1 até Z100) ou um total de 2600 elementos. Teoricamente, você poderia usar a seguinte matriz de estruturas para armazenar as entradas da planilha:

```
struct cell {
  char cell_name[9];
  char formula[128];
} list_entry[2600];  /* 2.600 células */
```

Porém, 2.600 multiplicado por 137 (o tamanho da estrutura — algumas máquinas exigem que os dados sejam alinhados em posições pares, aumentando, assim, o número de bytes realmente necessários) resulta em 356.200 bytes de memória. Esse é um número muito grande para muitos sistemas. Além disso, em processadores que usam arquitetura segmentada, tais como a família 8086, o acesso à memória de matrizes tão grandes é muito lento, porque tornam-se necessários ponteiros de 32 bits. Portanto, essa abordagem geralmente não é prática. No entanto, você pode criar uma matriz de ponteiros a estruturas do tipo **Cell**. Essa matriz de ponteiros exigiria muito menos armazenamento permanente que uma matriz inteira. Toda vez que se atribuem dados a uma localização de matriz, uma memória seria alocada para esses dados e o ponteiro apropriado na matriz de ponteiro seria definido para apontar para os dados. Esse esquema oferece um desempenho superior em relação aos métodos com árvore binária e lista encadeada. A declaração que cria essa matriz de ponteiros é

```
struct cell {
  char cell_name[9];
  char formula[128];
} list_entry;

struct cell *sheet[2600];  /* matriz de 2.600 ponteiros */
```

Você pode usar essa matriz menor para armazenar ponteiros para a informação que é inserida pelo usuário da planilha. Conforme cada item é inserido, um ponteiro para a informação contida na célula é armazenado na posição apropriada na matriz. A Figura 21.2 mostra como isso apareceria na memória, com a matriz de ponteiros fornecendo suporte para a matriz esparsa.

Antes que uma matriz de ponteiros possa ser usada, cada elemento deve ser inicializado com nulo, o que indica que não há entrada nessa posição.

A função que executa isso é a mostrada depois da Figura 21.2.

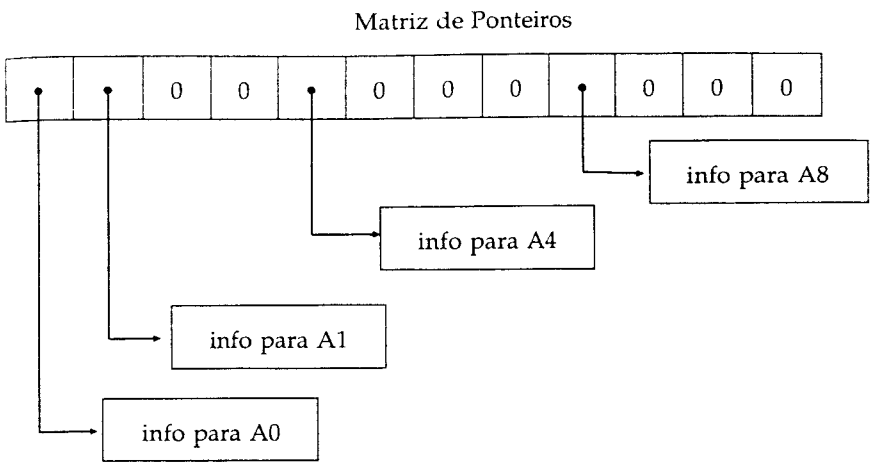


Figura 21.2 Uma matriz de ponteiros como suporte para uma matriz esparsa.

```
void init_sheet(void)
{
  register int t;

  for(t=0; t<2600; ++t) sheet[t]= NULL;
}
```

Quando o usuário insere uma fórmula para uma célula, a posição da célula (que é definida por seu nome) é usada para produzir um índice para a matriz de ponteiros **sheet**. O índice é obtido do nome da célula, convertendo-se o nome em um número, como mostrado na listagem seguinte.

```
void store(struct cell *i)
{
  int loc;
  char *p;

  /* calcula a posição dado o nome do ponto */
  loc = *(i->cell_name)- 'A';
  p = &(i->cell_name[1]);
  loc += (atoi(p)-1) * 26;  /* linhas * colunas */

  if(loc>=2600) {
    printf("Célula fora dos limites.\n");
    return;
```

```
    }
    sheet[loc] = i;  /* coloca o ponteiro na matriz */
}
```

Para calcular o índice, **store()** assume que todos os nomes das células começam com uma letra maiúscula e são seguidos por um número inteiro — por exemplo, B34, C19, e assim por diante. Assim, usando a fórmula armazenada em **store()**, o nome da célula A1 gera um índice zero, B1 gera um índice 1, A2 gera um índice 26, e assim por diante. Como cada nome de célula é único, cada índice também é único e o ponteiro para cada entrada é armazenado no elemento apropriado da matriz. Se você comparar essa rotina com a versão para lista encadeada ou árvore binária, você verá o quanto ela é mais curta e simples.

A função **delete()** também fica bem reduzida. Quando chamada com o nome da célula a remover, ela simplesmente zera o ponteiro para o elemento e devolve a memória ao sistema.

```
void delete(struct cell *i)
{
  int loc;
  char *p;

  /* calcula a posição dado o nome do ponto */
  loc = *(i->cell_name)-'A';
  p = &(i->cell_name[1]);
  loc += (atoi(p)-1) * 26;  /* linhas * colunas */

  if(loc>=2600) {
    printf("Célula fora dos limites.\n");
    return;
  }
  if(!sheet[loc]) return;  /* não libera um ponteiro nulo */

  free(sheet[loc]); /* devolve memória ao sistema */
  sheet[loc] = NULL;
}
```

Novamente, esse código é muito mais rápido e simples que a versão para lista encadeada.

O processo de localização de uma célula, dado seu nome, é simples, porque o nome em si produz diretamente o índice da matriz. Assim, a função **find()** torna-se

```
struct cell *find(char *cell_name)
{
```



```
  int loc;
  char *p;

  /* calcula a posição dado o nome do ponto */
  loc = *(cell_name) -'A';
  p = &(cell_name[1]);
  loc += (atoi(p)-1) * 26;  /* linhas * colunas */

  if(loc>=2600 || !sheet[loc]) { /* nenhuma entrada na célula */
    printf("Célula não encontrada.\n");
    return NULL; /* não encontrada */
  }
  else return sheet[loc];
}
```

Análise da Abordagem com Matriz de Ponteiros

O método de matriz de ponteiros para manipulação de matrizes esparsas fornece um acesso muito mais rápido aos elementos da matriz que os métodos de lista encadeada ou árvore binária. A menos que a matriz seja muito grande, a memória usada pela matriz de ponteiros normalmente não consome, significativamente, a memória livre do sistema. No entanto, a matriz de ponteiros, por si só, utiliza alguma memória para toda posição — estejam os ponteiros sendo utilizados ou não. Isso pode ser uma séria limitação para certas aplicações, embora, em geral, não seja um problema.

Hashing

Hashing é o processo de extrair o índice de um elemento de matriz diretamente da informação que deve ser armazenada. O índice gerado é chamado *hash*. Tradicionalmente, hashing tem sido aplicado a arquivos em disco como uma forma de diminuir o tempo de acesso. Porém, você pode usar os mesmos métodos gerais para implementar matrizes esparsas. O exemplo anterior de matriz de ponteiros usou uma forma especial de hashing chamada *indexação direta*, onde cada chave é mapeada em uma e apenas uma posição na matriz. Isto é, cada índice fragmentado é único. (O método com matriz de ponteiros não requer um hashing com indexação direta — isso foi apenas uma abordagem óbvia dada ao problema da planilha.) Na prática, existem poucos esquemas de hashing direto e um método mais flexível torna-se necessário. Esta seção mostra como hashing pode ser generalizado para permitir maior poder e flexibilidade.

O exemplo de planilha torna claro que, mesmo nos mais rigorosos ambientes, nem toda célula da planilha será usada. Suponha que, para virtualmente todos os casos, no máximo 10% das posições potenciais são ocupadas por entradas reais. Isto é, se a planilha tem dimensões 26x100 (2.600 posições), apenas cerca de 260 são realmente usadas em um dado momento e a maior matriz necessária para conter todas as entradas consistirá normalmente em apenas 260 elementos. Mas como as posições da matriz lógica são mapeadas e acessadas nessa matriz física menor? O que acontece se essa matriz está cheia?

Quando uma fórmula para uma célula é inserida pelo usuário da planilha, que é a matriz lógica, a posição da célula, definida por seu nome, é usada para produzir um índice (um *hash*) na matriz física menor, algumas vezes chamada de *matriz primária*. O índice é obtido do nome da célula, que é convertido em um número, como no exemplo da matriz de ponteiros. Porém, o número é, em seguida, dividido por 10 para produzir um ponto de entrada inicial na matriz. (Lembre-se de que, nesse exemplo, o tamanho da matriz física é apenas 10% da matriz lógica.) Se a posição referenciada pelo índice está livre, o índice lógico e o valor são armazenados nessa posição. No entanto, como 10 posições lógicas são mapeadas em uma única posição física, podem ocorrer colisões de hash. Quando isso acontece, uma lista encadeada, algumas vezes chamada de *lista de colisão*, é usada para conter as entradas. Uma lista de colisão separada é mantida para cada entrada na matriz primária. Obviamente, essas listas têm comprimento zero até que ocorra uma colisão, como ilustrado na Figura 21.3.

Suponha que você deseje encontrar um elemento na matriz física, dado seu índice na matriz lógica. Primeiro, deve-se transformar o índice lógico em seu valor de hash e verificar na matriz física, no índice gerado pelo hash, para ver se o índice lógico armazenado nessa posição coincide com aquele sendo pesquisado. Se coincide, a informação é devolvida. Caso contrário, a lista de colisão é seguida até que o índice apropriado seja encontrado ou até que seja atingido o fim da seqüência.

Antes de poder entender como esse procedimento se aplica ao programa de planilha, você precisa definir uma matriz de estrutura chamada **primary**, como mostrado a seguir:

```
#define MAX 260

struct htype {

  int index;  /* índice real */
  int val;  /* valor real do elemento da matriz */.
  struct htype *next;  /* ponteiro para o próximo valor com
                          mesmo fragmento*/
} primary[MAX];
```

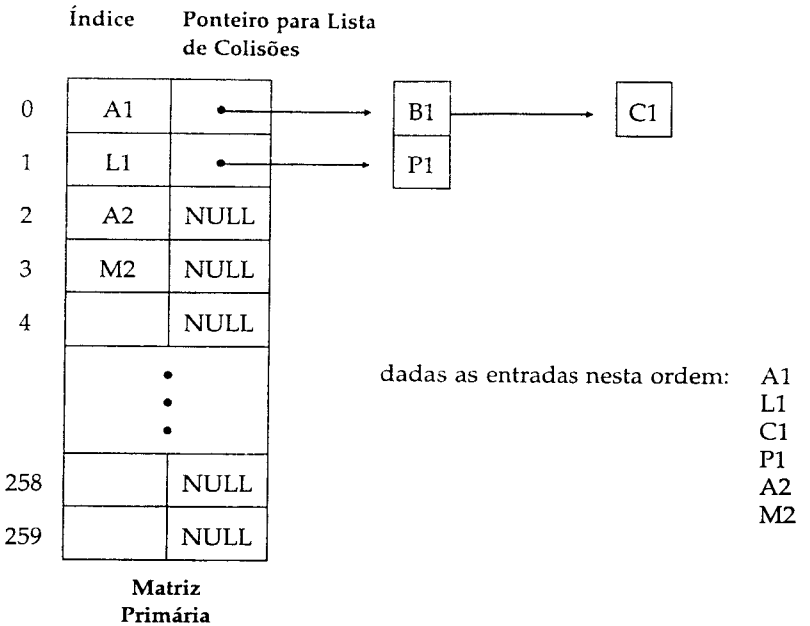



Figura 21.3 Um exemplo de hashing.

Antes que essa matriz possa ser usada, ela deve ser inicializada. A função seguinte inicializa o campo **index** para -1 (um valor que, por definição, não pode ser gerado) para indicar um elemento vazio. Um **NULL** no campo **next** indica o final da seqüência de hash.

```
/* Inicializa a matriz de fragmentos. */
void init(void)
{
  register int i;

  for(i=0; i<MAX; i++) {
    primary[i].index = -1;
    primary[i].next = NULL;  /* seqüência nula */
    primary[i].val = 0;
  }
}
```

A função **store()** converte o nome de uma célula em um hash para a matriz **primary**. Se a posição diretamente apontada pelo valor está ocupada, a função acrescenta automaticamente a entrada na lista de colisão, usando uma versão modificada de **slstore()** desenvolvida no capítulo anterior. O índice lógico deve ser armazenado, pois ele é necessário quando esse elemento for acessado novamente. Essas funções são mostradas aqui:

```
/* Calcula o fragmento e armazena o valor. */
void store(char *cell_name, int v)
{
  int h, loc;
  struct htype *p;

  /* produz o valor do hash */
  loc = *cell_name - 'A';
  loc += (atoi(&cell_name[1])-1) * 26; /* linha * colunas */
  h = loc/10;

  /* armazena a posição a menos que esteja cheia ou
     armazena se os índices são iguais - p.e., atualiza.
  */
  if(primary[h].index = -1 || primary[h].index==loc) {
    primary[h].index = loc;
    primary[h].val = v;
    return;
  }

  /* caso contrário, cria ou adiciona a uma lista de colisão */
  p = malloc(sizeof(struct htype));
  if(!p) {
    printf("Sem memória\n");
    return;
  }
  p->index = loc;
  p->val = v;
  slstore(p, &primary[h]);
}

/* Acrescenta elementos à lista de colisão. */
void slstore(struct htype *i,
             struct htype *start)
{
  struct htype *old, *p;
  old = start;
```



```
  /* encontra o final da lista */
  while(start) {
    old = start;
    start = start->next;
  }
  /* une a nova entrada */
  old->next = i;
  i->next = NULL;
}
```

Antes de encontrar o valor de um elemento, seu programa deve, primeiro, calcular o hash e, em seguida, verificar se o índice lógico armazenado na matriz física coincide com o índice da matriz lógica solicitado. Em caso afirmativo, esse valor é devolvido; caso contrário, a lista de colisão é varrida. A função **find()**, que executa essas tarefas, é mostrada aqui:

```
/* Calcula o hash e devolve o valor. */
int find(char *cell_name)
{
  int h, loc;
  struct htype *p;

  /* produz o valor do hash */
  loc = *cell_name - 'A';
  loc += (atoi(&cell_name[1])-1) * 26; /* linha * colunas */
  h = loc/10;

  /* devolve o valor se encontrou */
  if(primary[h].index==loc) return(primary[h].val);
  else { /* procura na lista de colisão */
    p = primary[h].next;
    while(p) {
      if(p->next == loc) return p->val;
      p = p->next;
    }
    printf("Não está na matriz\n");
    return -1;
  }
}
```

A função de exclusão é deixada para você como exercício. (Sugestão: apenas inverta o processo de inserção).

Tenha em mente que o algoritmo de hashing anterior é muito simples. Geralmente, você usaria um método mais complexo para conseguir uma distribuição uniforme de índices na matriz primária, evitando, assim, seqüências de hashing muito longas. No entanto, o princípio básico é o mesmo.

Análise de Hashing

No melhor caso (muito raro), todo índice físico criado pelo hashing é único e o tempo de acesso aproxima-se do da indexação direta. Isso significa que nenhuma lista de colisão é criada e todas as consultas são essencialmente acessos diretos. No entanto, esse raramente é o caso, porque exige que os índices lógicos estejam uniformemente distribuídos ao longo do espaço dos índices lógicos. No pior caso (também raro), um esquema de hashing se degenera em uma lista encadeada. Isso pode acontecer quando os hash dos índices lógicos são iguais. No caso médio (e o mais provável), o método de hashing pode acessar qualquer elemento específico na quantidade de tempo levada para usar um índice direto dividido por alguma constante que é proporcional ao comprimento médio das cadeias de colisão. O fator mais crítico no uso de hashing, para operar uma matriz esparsa, é assegurar que o algoritmo distribua os índices físicos uniformemente de forma a evitar longas listas de colisão.

Escolhendo uma Abordagem

Você deve considerar a velocidade e a eficiência no uso da memória ao decidir se usará uma lista encadeada, uma árvore binária, uma matriz de ponteiros ou um método de hashing para implementar uma matriz esparsa.

Quando a matriz é muito esparsa, as abordagens que utilizam mais eficientemente a memória são as listas encadeadas e as árvores binárias, porque apenas os elementos da matriz que estão realmente sendo usados têm memória alocada para eles. Os elos requerem pouquíssima memória adicional e, geralmente, têm um efeito desprezível. O modelo com matriz de ponteiros requer que toda a matriz de ponteiros exista, mesmo que alguns dos seus elementos não sejam usados. Não apenas a matriz inteira deve ser acomodada na memória como deve existir memória suficiente para o aplicativo usar. Isso pode ser um sério problema para certas aplicações, e não ser para outras. Normalmente você calcula a quantidade aproximada de memória livre e determina se é suficiente para seu programa. O método de hashing está situado em algum lugar entre as abordagens com matriz de ponteiros e árvore binária/lista encadeada. Embora ele exija que a matriz física exista com todos seus elementos, mesmo que nem

todos sejam usados, ela deve ser menor que uma matriz de ponteiros, que necessita de pelo menos um ponteiro para toda posição da matriz lógica.

No entanto, quando a matriz está razoavelmente cheia, a matriz de ponteiros faz o melhor uso da memória. Isso ocorre porque as implementações com árvore binária e lista encadeada usam dois ponteiros para cada elemento, ao passo que a matriz de ponteiros tem apenas um. Por exemplo, suponha que uma matriz de 1.000 elementos esteja cheia e os ponteiros tenham dois bytes de comprimento. Tanto a árvore binária quanto a lista encadeada usariam 4.000 bytes para os ponteiros, mas a matriz de ponteiros precisaria de apenas 2.000 — uma economia de 2.000 bytes. No método de hashing, mais memória ainda seria gasta para suportar a matriz.

A abordagem mais rápida, em termos de velocidade de execução, é a matriz de ponteiros. Como no exemplo da planilha, sempre existe um método fácil para indexar uma matriz de ponteiros e conectá-la aos elementos da matriz esparsa. Isso torna o acesso aos elementos da matriz esparsa tão rápido quanto o acesso a uma matriz normal. A versão com lista encadeada é muito lenta, porque usa uma pesquisa linear para localizar cada elemento. Mesmo que fossem acrescentadas informações extras à lista encadeada, para proporcionar o acesso mais rápido aos elementos, ela ainda seria mais lenta que a capacidade de acesso direto da matriz de ponteiros. A árvore binária certamente acelera o tempo de pesquisa, mas é vagarosa comparada com a capacidade de indexação direta da matriz de ponteiros. Se você escolher o algoritmo de hashing, esse método pode superar a velocidade da árvore binária, mas nunca será mais rápido que a abordagem com matriz de ponteiros.

A regra prática é usar uma implementação com matriz de ponteiros, quando possível, porque esse método é o mais rápido. Porém, se o uso da memória é crítico, você deve usar a abordagem com lista encadeada ou árvore binária.

MAKRON Books

22

Análise de Expressões e Avaliação

Como escrever um programa que recebe como entrada uma string contendo um expressão numérica, como (10 - 5) * 3, e calcular a resposta apropriada? Se ainda há um "alto clero" entre os programadores, dele devem fazer parte os poucos que sabem como isso pode ser feito. Muitos dos que usam computador mistificam a maneira pela qual uma linguagem de alto nível converte expressões complexas, como 10 * 3 - (4 + count)/12, em instruções que um computador pode executar. Este procedimento é chamado de *análise de expressões*, e é a espinha dorsal de todos os compiladores e interpretadores de linguagens, programas de planilha de cálculo e qualquer outra coisa que necessite converter expressões numéricas em uma forma que o computador possa usar. Análise de expressão é, geralmente, considerada fora dos limites, exceto para aqueles poucos iluminados, mas esse não é o caso.

Embora misteriosa, análise de expressões é, na realidade, muito simples, e, de certa forma, é ainda mais simples que outras tarefas de programação. A razão disso é que a tarefa é bem-definida e funciona de acordo com as regras rígidas da álgebra. Esse capítulo desenvolve o que é normalmente chamado de *analisador recursivo descendente* e todas as rotinas de suporte necessárias para avaliar expressões numéricas complexas. Uma vez que você tenha dominado a operação do analisador, pode facilmente aperfeiçoá-lo e modificá-lo para se adaptar às suas necessidades. De mais a mais, os outros programadores pensarão que você entrou para o "alto clero"!

NOTA: O interpretador C apresentado na Parte 5 deste livro usa versão melhorada do analisador desenvolvido aqui. Se você pretende explorar o interpretador C, achará o material deste capítulo especialmente útil.

Expressões

Embora expressões possam ser feitas com todo tipo de informação, este capítulo trata apenas de expressões numéricas. Para nossos propósitos, *expressões numéricas* podem ser formadas com os seguintes itens:

- Números
- Os operadores +, -, /, *, ^, %, =
- Parênteses
- Variáveis

O operador ^ indica exponenciação, como em BASIC, e = é o operador de atribuição. Esses itens podem ser combinados em expressões de acordo com as regras de álgebra. Aqui estão alguns exemplos:

```
10 - 8
(100 - 5) * 14 / 6
a + b - c
10 ^ 5
a = 10 - b
```

Assuma a seguinte precedência para cada operador:

maior	+, - unários
	^
	*, /, %
	+, -
menor	=

Operadores de igual precedência são avaliados da esquerda para a direita.

Nos exemplos deste capítulo, todas as variáveis são formadas por uma única letra (em outras palavras, estão disponíveis 26 variáveis, de **A** a **Z**). As variáveis não são diferenciadas com relação a minúsculas e maiúsculas (**a** e **A** são tratadas da mesma forma). Todo número é um **double**, embora você possa facilmente escrever rotinas para manipular outros tipos de números. Finalmente, para manter a lógica clara e fácil de entender, apenas uma quantidade mínima de verificação de erros está incluída nas rotinas.

Caso você nunca tenha pensado sobre análise de expressão, tente avaliar esta expressão:

```
10 - 2 * 3
```

Essa expressão tem o valor 4. Embora você possa facilmente criar um programa que calcule essa expressão específica, a questão é como criar um pro-

grama que forneça a resposta correta para qualquer expressão arbitrária. A princípio, você poderia pensar em uma rotina semelhante a esta:

```
a = pega o primeiro operando
while(operandos presentes) {
        op = pega operador
        b  = pega segundo operando
        a  = a op b
}
```

Essa rotina apanha o primeiro operando, o operador, e o segundo operando, para executar a primeira operação, e, em seguida, pega o próximo operador e operando — se houver — para executar a próxima operação, e assim por diante. No entanto, se você usar essa abordagem básica, a expressão 10 - 2 * 3 será avaliada como 24 (isto é, 8 * 3) em vez de 4, porque esse procedimento despreza a precedência dos operadores. Você não pode simplesmente tomar os operandos e operadores em ordem, da esquerda para a direita, porque a multiplicação deve ser feita antes da subtração. Alguns principiantes pensam que isso pode ser facilmente resolvido e algumas vezes pode — em casos muito restritos. Mas o problema só piora quando são acrescentados parênteses, exponenciação, variáveis, chamadas a funções e coisas do gênero.

Embora existam umas poucas maneiras de escrever uma rotina que avalie expressões desse tipo, a desenvolvida aqui é a de mais fácil escrita e também a mais simples. (Alguns dos outros métodos usados para escrever analisadores empregam tabelas complexas que devem ser geradas por outro programa de computador. Esses métodos são, às vezes, chamados de *analisadores dirigidos por tabelas*.) O método usado aqui é chamado de *analisador recursivo descendente* e, no decorrer deste capítulo, você verá por que ele recebeu esse nome.

Dissecando uma Expressão

Antes que você possa desenvolver um analisador para avaliar expressões, precisa ser capaz de dividir uma expressão em seus componentes. Por exemplo, a expressão

```
A * B - (W + 10)
```

contém os componentes A, *, B, -, (, W, +, 10, e). Cada componente representa uma unidade indivisível da expressão. Em geral, você precisa de uma rotina que devolva cada item de uma expressão individualmente. A rotina também deve ser capaz de ignorar espaços e tabulações e detectar o final da expressão.

Cada componente de uma expressão é chamado de *token*. Assim, a função que devolve o próximo token da expressão é, geralmente, chamada de **get_token()**. Um ponteiro global para caractere é necessário para armazenar a string da expressão. Na versão de **get_token()** mostrada aqui, esse ponteiro é chamado de **prog**. A variável **prog** é global porque ela deve manter seu valor entre as chamadas a **get_token()** e permitir que outras funções a utilizem. Além de receber um token de **get_token()**, você precisa saber que tipo de token está sendo devolvido. Para o analisador desenvolvido neste capítulo, você só precisa de três tipos: **VARIAVEL**, **NUMERO** e **DELIMITADOR**. (**DELIMITADOR** é usado tanto para operador como para parênteses.) Aqui está **get_token()** com suas variáveis globais, **#defines** e funções de suporte necessárias:

```
#define DELIMITADOR 1
#define VARIAVEL 2
#define NUMERO 3

extern char *prog; /* contém a expressão a ser analisada */
char token[80];
char tok_type;

/* Devolve o próximo token. */
void get_token(void)
{
  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*prog) return; /* final da expressão */
  while(isspace(*prog)) ++prog; /* ignora espaços em branco */

  if(strchr("+-*/%^=()", *prog)) {
    tok_type = DELIMITADOR;
    /* avança para o próximo char */
    *temp++ = *prog++;
  }
  else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = VARIAVEL;
  }
  else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
```

```
    tok_type = NUMERO;
  }

  *temp = '\0';
}

/* Devolve verdadeiro se c é um delimitador. */
isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

Olhe atentamente para as funções anteriores. Após algumas poucas inicializações, **get_token()** verifica se a terminação com **NULL** da expressão foi encontrada. Em seguida, os espaços iniciais são ignorados. Depois que os espaços são ignorados, **prog** está apontando para um número, uma variável, um operador ou — se a expressão termina com espaços — um nulo. Se o próximo caractere é um operador, ele é devolvido como uma string na variável global **token** e o **DELIMITADOR** é colocado em **tok_type**. Se o próximo caractere é uma letra, ela é assumida como sendo uma das variáveis, devolvida como uma string e o valor **VARIAVEL** é atribuído a **tok_type**. Se o próximo caractere é um dígito, o número inteiro é lido e colocado na string **token** como tipo **NUMERO**. Finalmente, se o próximo caractere não é nenhum desses, é assumido que o final da expressão foi atingido. Nesse caso, **token** é nulo, o que significa o final da expressão.

Como dito anteriormente, para manter claro o código desta função, várias verificações de erro foram omitidas e algumas suposições foram feitas. Por exemplo, qualquer caractere não reconhecido pode terminar a expressão. Além disso, nessa versão, as variáveis podem ter qualquer extensão, mas apenas a primeira letra é significativa. Você pode adicionar uma maior verificação de erros e outros detalhes de acordo com sua aplicação específica. Você pode facilmente modificar ou melhorar **get_token()** para permitir strings, outros tipos de números ou qualquer coisa que seja devolvida como um token por vez de uma string de entrada.

Para entender melhor como **get_token()** opera, estude o que ela devolve para cada token da seguinte expressão:

A + 100 - (B * C)/2

Token	Tipo do token
A	VARIAVEL
+	DELIMITADOR
100	NUMERO
-	DELIMITADOR
(	DELIMITADOR
B	VARIAVEL
*	DELIMITADOR
C	VARIAVEL
)	DELIMITADOR
/	DELIMITADOR
2	NUMERO
nulo	Null

Não se esqueça de que **token** sempre contém uma string terminada com um nulo, mesmo que ela tenha apenas um único caractere.

Análise de Expressão

Há muitas maneiras de analisar e avaliar uma expressão. Para usar um analisador recursivo descendente, imagine as expressões como sendo *estruturas de dados recursivas* — isto é, expressões que são definidas em termos delas mesmas. Se, para o momento, você restringir as expressões a usar para apenas +, -, *, / e parênteses, todas as expressões podem ser definidas com as seguintes regras:

expressão → termo[+termo][-termo]
termo → fator [*fator][/fator]
fator → variável, número ou (expressão)

Os colchetes designam um elemento opcional e → significa "produz". As regras são normalmente chamadas de *regras de produção* da expressão. Assim, você poderia ler a definição de *termo* como: "Termo produz fator vezes fator ou fator dividido por fator". Note que a precedência dos operadores está implícita na maneira como uma expressão é definida.

A expressão

10 + 5 * B

tem dois termos: 10 e 5 * B. O segundo termo tem 2 fatores: 5 e B. Esses fatores consistem em um número e uma variável.

Por outro lado, a expressão

14 * (7 - C)

tem dois fatores: 14 e (7 - C). Os fatores consistem em um número e uma expressão entre parênteses. A expressão entre parênteses contém dois termos: um número e uma variável.

Esse processo forma a base de um *analisador recursivo descendente*, que é basicamente um conjunto de funções mutuamente recursivas que opera de forma encadeada. A cada passo, o analisador executa as operações especificadas na seqüência algebricamente correta. Para ver como esse processo funciona, analise a expressão e execute as operações aritméticas no momento apropriado:

9/3 - (100 + 56)

Se você analisou a expressão corretamente, seguirá estes passos:

1. Pegue o primeiro termo, 9/3.
2. Pegue cada fator e divida os inteiros. O valor do resultado é 3.
3. Pegue o segundo termo, (100 + 56). Neste ponto, comece a analisar recursivamente a segunda subexpressão.
4. Pegue cada fator e some. O valor do resultado é 156.
5. Retorne da chamada recursiva e subtraia 156 de 3. A resposta é -153.

Se, neste ponto, você está um pouco confuso, não se preocupe. Esse é um conceito razoavelmente complexo que precisa ser aplicado. Há dois pontos básicos a serem lembrados sobre essa visão recursiva das expressões. Primeiro, a precedência dos operadores está implícita na maneira como as regras de produção são definidas. Segundo, esse método de análise e avaliação de expressões é muito semelhante à forma como os humanos avaliam expressões matemáticas.

Um Analisador Simples de Expressões

O restante deste capítulo desenvolve dois analisadores. O primeiro analisa e avalia apenas expressões constantes — isto é, expressões sem variáveis. Esse exemplo mostra o analisador na sua forma mais simples. O segundo analisador inclui as 26 variáveis de **A** a **Z**.

Aqui está a versão completa do analisador recursivo descendente simples para expressões em ponto flutuante:

```
/* Este módulo contém um analisador de expressões simples que
   não reconhece variáveis.
*/

#include <stdlib.h>
```



```
#include <ctype.h>
#include <stdio.h>
#include <string.h>

#define DELIMITADOR  1
#define VARIAVEL     2
#define NUMERO       3

extern char *prog; /* contém a expressão a ser analisada */
char token[80];
char tok_type;

void eval_exp(double *answer), eval_exp2(double *answer);
void eval_exp3(double *answer), eval_exp4(double *answer);
void eval_exp5(double *answer);
void eval_exp6(double *answer), atom(double *answer);
void get_token(void), putback(void);
void serror(int error);
int isdelim(char c);

/* Ponto de entrada do analisador. */
void eval_exp(double *answer)
{
  get_token();
  if(!*token) {
    serror(2);
    return;
  }
  eval_exp2(answer);
  if (*token) serror (0); /* último token deve ser null */
}

/* Soma ou subtrai dois termos. */
void eval_exp2(double *answer)
{
  register char op;
  double temp;

  eval_exp3(answer);
  while((op = *token) == '+' || op == '-') {
    get_token():
    eval_exp3(&temp);
    switch(op) {
      case '-':
        *answer = *answer - temp;
```

```
        break;
      case '+':
        *answer = *answer + temp;
        break;
    }
  }
}

/* Multiplica ou divide dois fatores. */
void eval_exp3(double *answer)
{
  register char op;
  double temp;

  eval_exp4(answer);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(&temp);
    switch(op) {
      case '*':
        *answer = *answer * temp;
        break;
      case '/':
        *answer = *answer / temp;
        break;
      case '%':
        *answer = (int) *answer % (int) temp;
        break;
    }
  }
}

/* Processa um expoente */
void eval_exp4(double *answer)
{
  double temp, ex;
  register int t;

  eval_exp5(answer);
  if(*token== '^') {
    get_token();
    eval_exp4(&temp);
    ex = *answer;
    if(temp==0.0) {
      *answer = 1.0;
```



```
      return;
    }
    for(t=temp-1; t>0; --t) *answer = (*answer) * (double)ex;
  }
}

/* Avalia um + ou - unário. */
void eval_exp5(double *answer)
{
  register char op;

  op = 0;
  if((tok_type == DELIMITADOR) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(answer);
  if(op=='-') *answer = -(*answer);
}

/* Processa uma expressão entre parênteses. */
void eval_exp6(double *answer)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(answer);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else
    atom(answer);
}

/* Obtém o valor real de um número. */
void atom(double *answer)
{
  if(tok_type==NUMERO) {
    *answer = atof(token);
    get_token();
    return;
  }
  serror(0); /* caso contrário, erro de sintaxe na expressão */
}
```

```
/* Devolve um token à stream de entrada. */
void putback(void)
{
  char *t;

  t = token;
  for(; *t; t++) prog--;
}

/* Apresenta um erro de sintaxe. */
void serror(int error)
{
  static char *e[] = {
      "Erro de sintaxe",
      "Falta parênteses",
      "Nenhuma expressão presente"
  };
  printf("%s\n", e[error]);
}

/* Devolve o próximo token. */
void get_token(void)
{

  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*prog) return; /* final da expressão */
  while(isspace(*prog)) ++prog; /* ignora espaços em branco */

  if(strchr("+-*/%^=()", *prog)) {
    tok_type = DELIMITADOR;
    /* avança para o próximo char */
    *temp++ = *prog++;
  }
  else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = VARIAVEL;
  }
  else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = NUMERO;
```



```
  }

  *temp = '\0';
}

/* Devolve verdadeiro se c é um delimitador. */
isdelim(char c)
{

  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}
```

O analisador, como mostrado, pode manipular os seguintes operadores: +, -, *, /, %, assim como exponenciação inteira (^) e o menos unário. O analisador também pode trabalhar corretamente com parênteses. Observe que ele tem seis níveis e a função **atom()**, que devolve o valor de um número. Como discutido, as duas variáveis globais, **token** e **tok_type**, retornam da string de expressão o próximo token e seu tipo. O ponteiro **prog** aponta para a string que contém a expressão.

A função **main()** a seguir demonstra o uso do analisador:

```
/* Programa de demonstração do analisador. */
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

char *prog;
void eval_exp(double *answer);

void main(void)
{
  double answer;
  char *p;

  p = malloc(100);
  if(!p) {
    printf("Falha na alocação.\n");
    exit(1);
  }
```

```
  /* Processa expressões até que uma linha em branco seja
     digitada.
  */
  do {
    prog = p;
    printf ("Digite a expressão: ");
    gets(prog);
    if(!*prog) break;
    eval_exp(&answer);
    printf("A resposta é %.2f\n", answer);
  } while(*p);
}
```

Para entender exatamente como o analisador avalia uma expressão, trabalhe sobre a seguinte expressão, apontada por **prog**:

10 - 3 * 2

Quando **eval_exp()**, o ponto de entrada do analisador, é chamada, ela pega o primeiro token. Se o token é nulo, a rotina escreve a mensagem "nenhuma expressão presente" e retorna. Nesse ponto, o token contém o número 10. Se o token não é nulo, **eval_exp2()** é chamada. (**eval_exp1()** é usada quando o operador de atribuição é utilizado, não sendo necessário aqui.) Como resultado, **eval_exp2()** chama **eval_exp3()** e **eval_exp3()** chama **eval_exp4()**, que, por sua vez, chama **eval_exp5()**. Em seguida, **eval_exp5()** verifica se o token é um mais ou um menos unário, que, nesse caso, não é; então, **eval_exp6()** é chamada. Nesse ponto, **eval_exp6()** chama **eval_exp2()** (no caso de expressões entre parênteses) ou **atom()** para encontrar o valor do número. Finalmente, **atom()** é executado e ***answer** contém o número 10. Outro token é retirado, e as funções começam a retornar ao início da seqüência. O token é agora o operador -, e as funções retornam até **eval_exp2()**.

O que acontece nesse ponto é muito importante. Como o token é -, ele é salvo em **op**. O analisador então obtém o novo token 3 e reinicia a descida na seqüência. Novamente **atom()** é chamada, o valor devolvido em ***answer** é 3 e o token * é lido. Isso provoca um retorno na seqüência até **eval_exp3()**, onde o token final é lido. Nesse ponto, ocorre a primeira operação aritmética com a multiplicação de 2 e 3. O resultado é devolvido a **eval_exp2()** e a subtração é executada. A subtração fornece 4 como resposta. Embora esse processo possa, a princípio, parecer complicado, trabalhe com outros exemplos e verifique que esse método sempre funciona corretamente.

Esse analisador poderia ser adequado a uma calculadora de mesa, como ilustrado no programa anterior. Ele também poderia ser usado em um banco de dados limitado. Antes que pudesse ser usado em uma linguagem de computador ou em uma calculadora sofisticada, seria necessária a habilidade de manipular variáveis. Esse é o assunto da próxima seção.

Acrescentando Variáveis ao Analisador

Todas as linguagens de programação, muitas calculadoras e planilhas de cálculo usam variáveis para armazenar valores para uso posterior. O analisador simples da seção anterior precisa ser expandido para incluir variáveis antes de ser capaz de armazenar valores. Para incluir variáveis, você precisa acrescentar diversos itens ao analisador. Primeiro, obviamente, as próprias variáveis. Como dito anteriormente, o analisador reconhece apenas as variáveis de **A** a **Z** (embora isso possa ser expandido, se você quiser). Cada variável usa uma posição em uma matriz de 26 elementos **double**s. Assim, acrescente as seguintes linhas ao analisador:

```
double vars[26]= {  /* 26 variáveis do usuário, A-Z */
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0
};
```

Como você pode ver, as variáveis são inicializadas com 0, como cortesia para o usuário.

Você também precisa de uma rotina para ler o valor de uma variável dada. Como as variáveis têm nomes de **A** a **Z**, elas podem facilmente ser usadas para indexar a matriz **vars**, subtraindo o valor de ASCII para **A** do nome da variável. A função **find_var()** é mostrada aqui:

```
/* Devolve o valor de uma variável. */
double find_var(char *s);
{
  if(!isalpha(*s)) {
    serror(1);
    return 0;
  }
  return vars[toupper(*token)-'A'];
}
```

Como está escrita, a função aceitará nomes longos de variáveis, mas apenas a primeira letra é significativa. Isso pode ser modificado para se adaptar às suas necessidades.

A função **atom()** também deve ser modificada para manipular números e variáveis. A nova versão é mostrada aqui:

```
/* Obtém o valor de um número ou uma variável. */
void atom(double *answer)
{
  switch(tok_type) {
    case VARIAVEL:
      *answer = find_var(token);
      get_token();
      return;
    case NUMERO;
      *answer = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}
```

Tecnicamente, isso é tudo que precisa ser acrescentado para o analisador utilizar variáveis corretamente; porém, não há como atribuir um valor a essas variáveis. Normalmente isso é feito fora do analisador, mas o sinal de igual pode ser tratado como um operador de atribuição e tornar-se parte do analisador. Existem várias maneiras de fazer isso. Um método é adicionar **eval_exp1()** ao analisador, como mostrado aqui:

```
/* Processa uma atribuição. */
void eval_exp1(double *result)
{
  int slot, ttok_type;
  char temp_token[80];

  if(tok_type==VARIAVEL) {
    /* salva token antigo */
    strcpy(temp_token, token);
    ttok_type = tok_type;

    /* calcula o índice da variável */
    slot = toupper(*token)-'A';

    get_token();
    if(*token != '=') {
      putback(); /* devolve token atual */
      /* restaura token antigo - nenhuma atribuição */
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); /* pega próxima parte da expressão */
      eval_exp2(result);
      vars[slot] = *result;
      return;
    }
  }

  eval_exp2(result);
}
```


Como você pode ver, a função precisa olhar à frente para determinar se uma atribuição está realmente sendo feita. Isso ocorre porque o nome da variável precede uma atribuição, mas um nome de variável sozinho não garante que uma expressão de atribuição venha a seguir. Isto é, o analisador aceitará A = 100 como uma atribuição, mas ele é inteligente o bastante para saber que A/10 é uma expressão. Para realizar isso, **eval_exp1()** lê o próximo token da entrada. Se ele não for o sinal de igual, o token será devolvido à entrada, para uso posterior, com uma chamada a **putback()**, mostrada aqui:

```
/* Devolve um token à stream de entrada. */
void putback(void)
{
  char *t;

  t = token;
  for(; *t; t++) prog--;
}
```

Aqui está o analisador melhorado completo:

```
/* Este módulo contém um analisador recursivo descendente
   que reconhece variáveis. */

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

#define DELIMITADOR  1
#define VARIAVEL 2
```

```
#define NUMERO  3

extern char *prog; /* contém a expressão a ser analisada */
char token[80];
char tok_type;

double vars[26]= {  /* 26 variáveis do usuário, A-Z */
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0
};

void eval_exp(double *answer), eval_exp2(double *answer);
void eval_exp1(double *result);
void eval_exp3(double *answer), eval_exp4(double *answer);
void eval_exp5(double *answer);
void eval_exp6(double *answer), atom(double *answer);
void get_token(void), putback(void);
void serror(int error);
double find_var(char *s);
int isdelim(char c);

/* Ponto de entrada do analisador. */
void eval_exp(double *answer)
{
  get_token();
  if(!*token) {
    serror(2);
    return;
  }
  eval_exp1(answer);
  if (*token) serror(0); /* o último token deve ser null */
}

/* Processa uma atribuição. */
void eval_exp1(double *answer)
{
  int slot
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIAVEL) {
    /* salva token antigo */
    strcpy(temp_token, token);
    ttok_type = tok_type;
```



```
     /* calcula o índice da variável */
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); /* devolve token atual */
      /* restaura token antigo - nenhuma atribuição */
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); /* pega a próxima parte da expressão */
      eval_exp2(answer);
      vars[slot] = *answer;
      return;
    }
  }
  eval_exp2(answer);
}

/* Soma ou subtrai dois termos. */
void eval_exp2(double *answer)
{
  register char op;
  double temp;

  eval_exp3(answer);
  while((op = *token) == '+' || op == '-') {
    get_token();
    eval_exp3(&temp);
    switch(op) {
      case '-':
        *answer = *answer - temp;
        break;
      case '+':
        *answer = *answer +  temp;
        break;
    }
  }
}

/* Multiplica ou divide dois fatores. */
void eval_exp3(double *answer)
{
```

```
  register char op;
  double temp;

  eval_exp4(answer);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(&temp);
    switch(op) {
      case '*':
        *answer = *answer * temp;
        break;
      case '/':
        *answer = *answer / temp;
        break;
      case '%':
        *answer = (int) *answer % (int) temp;
        break;
    }
  }
}

/* Processa um expoente. */
void eval_exp4(double *answer)
{
  double temp, ex;
  register int t;

  eval_exp5(answer);
  if(*token=='^') {
    get_token();
    eval_exp4(&temp);
    ex = *answer;
    if(temp==0.0) {
      *answer = 1.0;
      return;
    }
    for(t=temp-1; t>0; --t) *answer = (*answer) * (double)ex;
  }
}

/* Avalia um + ou - unário. */
void eval_exp5(double *answer)
{
  register char op;
```



```
  op = 0;
  if((tok_type == DELIMITADOR) && *token=='+' || *token == '-') {
    op = *token;
    get_token();
  }
  eval_exp6(answer);
  if(op=='-')*answer = -(*answer);
}

/* Processa uma expressão entre parênteses. */
void eval_exp6(double *answer)
{
  if((*token == '(')) {
    get_token();
    eval_exp2(answer);
    if(*token != ')')
      serror(1);
    get_token();
  }
  else atom(answer);
}

/* Obtém o valor de um número ou uma variável. */
void atom(double *answer)
{
  switch(tok_type) {
    case VARIAVEL:
      *answer = find_var(token);
      get_token();
      return;
    case NUMERO;
      *answer = atof(token);
      get_token();
      return;
    default:
      serror(0);
  }
}

/* Devolve um token à stream de entrada. */
void putback(void)
{
  char *t;

  t = token;
```

```
  for(; *t; t++) prog--;
}

/* Apresenta um erro de sintaxe. */
void serror(int error)
{
  static char *e[] = {
       "Erro de sintaxe",
       "Falta parênteses",
       "Nenhuma expressão presente"

  };
  printf("%s\n", e[error]);
}

/* Devolve o próximo token. */
void get_token(void)
{

  register char *temp;

  tok_type = 0;
  temp = token;
  *temp = '\0';

  if(!*prog) return; /* final da expressão */

  while(isspace(*prog)) ++prog; /* ignora espaços em branco */

  if(strchr("+-*/%^=()", *prog)) {
    tok_type = DELIMITADOR;
    /* avança para o próximo char */
    *temp++ = *prog++;
  }
  else if(isalpha(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = VARIAVEL;
  }
  else if(isdigit(*prog)) {
    while(!isdelim(*prog)) *temp++ = *prog++;
    tok_type = NUMERO;
  }

  *temp = '\0';
}
```



```
/* Devolve verdadeiro se c é um delimitador. */
isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

/* Devolve o valor de uma variável. */
double find_var(char *s);
{
  if(!isalpha(*s)) {
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

A mesma função **main()** usada com o analisador simples ainda pode ser usada. Com o analisador melhorado, você pode, agora, inserir expressões como

A = 10/4
A - B
C = A * (F - 21)

Verificação de Sintaxe em um Analisador Recursivo Descendente

Em análise de expressões, um erro de sintaxe é simplesmente uma situação em que a expressão de entrada não se encaixa nas regras rígidas exigidas pelo analisador. Na maioria das vezes, isso é provocado por erro humano — normalmente erros de digitação. Por exemplo, as expressões seguintes não são válidas para os analisadores deste capítulo:

10**8
(10 - 5)*9)
/8

A primeira contém dois operadores seguidos, a segunda tem um parênteses a mais e a última, um sinal de divisão no começo de uma expressão. Nenhuma dessas condições é permitida pelos analisadores deste capítulo. Como os erros de sintaxe podem fazer com que o analisador forneça resultados errados, você precisa prevenir-se contra eles.

Enquanto você estudava o código dos analisadores, provavelmente observou a função **serror()**, que é chamada sob certas situações. Ao contrário de muitos outros analisadores, o método recursivo descendente torna fácil a verificação de sintaxe, porque, na maioria das vezes, ela ocorre em **atom()**, **find_var()** ou **eval_exp6()**, onde são verificados os parênteses. O único problema com a verificação, como se apresenta agora, é que o analisador não é interrompido caso ocorra um erro de sintaxe. Isso pode levar a múltiplas mensagens de erro.

A melhor maneira de implementar a rotina **serror()** é tê-la executando uma rotina de reinicialização. Os compiladores que seguem o padrão ANSI vêm com um par de funções associadas, chamadas de **setjmp()** e **longjmp()**. Essas duas funções permitem que um programa desvie para uma função diferente. Portanto, em **serror()**, execute um **longjmp()** para algum lugar seguro fora do analisador.

Se o código for deixado da maneira como está, múltiplas mensagens de erros podem ser mostradas. Isso pode ser incômodo em algumas situações, mas um benefício em outros casos, porque mais de um erro pode ser encontrado. Geralmente, porém, a verificação de sintaxe deve ser melhorada antes de se usar esse código em programas comerciais.

MAKRON Books

23

Solução de Problemas de Inteligência Artificial

O campo da inteligência artificial (AI — Artificial Intelligence) é composto de diversos aspectos interessantes. Contudo, a solução de problemas é fundamental para a maioria das aplicações de inteligêngia artificial.

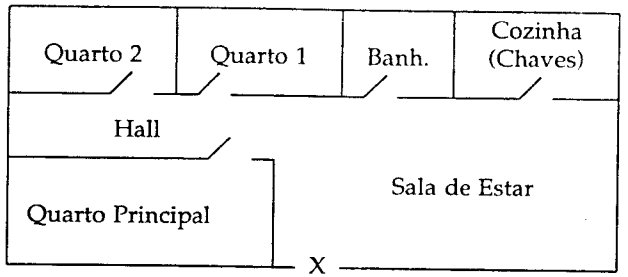
Basicamente, existem dois tipos de problema. O primeiro pode ser resolvido utilizando-se algum tipo de procedimento determinístico, com sucesso garantido — em outras palavras, uma *computação*. Os métodos utilizados para resolver esse tipo de problema são, geralmente, facilmente traduzidos em um algoritmo que um computador pode executar. Porém, poucos problemas reais se prestam a soluções computacionais. Na realidade, a maioria dos problemas é não-computacional. Esses problemas são resolvidos *através* de uma busca da solução — o método de solução de problemas com que inteligêngia artificial se preocupa.

Um dos sonhos da pesquisa de inteligêngia artificial é o *solucionador genérico de problemas*. Um solucionador genérico de problemas é um programa que pode produzir uma solução para todos os tipos de problemas diferentes sobre os quais ele não tem nenhum conhecimento específico. Este capítulo mostra por que o sonho é tão tentador quanto difícil de realizar.

Em investigações anteriores sobre inteligêngia artificial, desenvolver bons métodos de busca era o principal objetivo. Há duas razões para isso: necessidade e desejo. Um dos mais difíceis obstáculos quando se aplicam as técnicas de inteligêngia artificial aos problemas do mundo real, é a magnitude e complexidade da maioria das situações. Resolver esses problemas requer boas técnicas de busca. Além disso, os pesquisadores acreditavam, como ainda acreditam, que a busca constitui a mola mestra da solução de problemas, que é o ingrediente crucial da inteligência.

Representação e Terminologia

Imagine que você perdeu as chaves do seu carro. Você sabe que elas estão em algum lugar dentro de sua casa, que tem a seguinte planta baixa:



Você está de pé na porta da frente (onde está o X). Você começa a sua busca pela sala de estar. Em seguida, você passa pelo hall até o primeiro quarto, para o hall e para o segundo quarto, de volta ao hall e daí ao quarto principal. Não tendo encontrado as chaves, você volta à sala de estar. Você encontra as chaves na cozinha. Essa situação é facilmente representada por um diagrama, como mostrado na Figura 23.1.

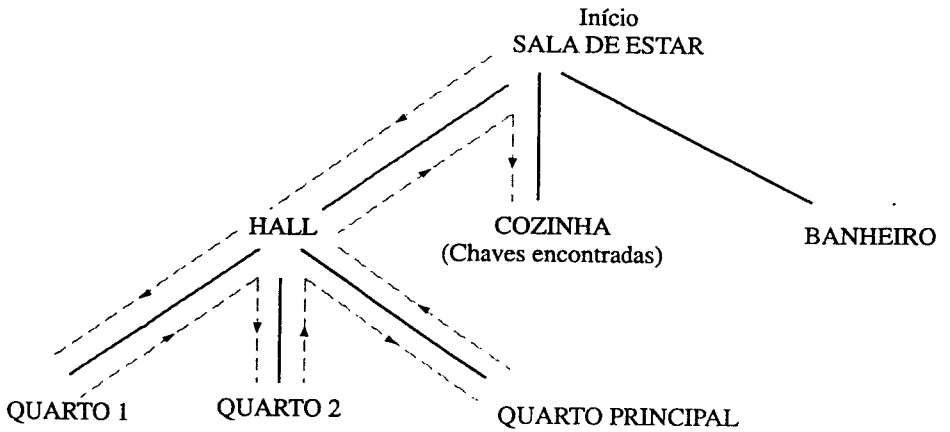


Figura 23.1 O percurso solução para encontrar as chaves perdidas.

O fato de os problemas serem representados por diagramas é importante, porque um diagrama fornece uma forma de visualizar como as diferentes técnicas de busca operam. (Além disso, a capacidade de representar problemas por meio deles permite ao pesquisador aplicar vários teoremas da teoria dos diagramas. No entanto, esses teoremas estão além do escopo deste livro.) Com isso em mente, estude as seguintes definições:

Nó	Um ponto discreto e uma possível meta
Nó terminal	Um nó que termina um percurso
Espaço de busca	O conjunto de todos os nós
Meta	O nó que é o objeto da busca
Heurística	Informação sobre se algum nó específico é uma escolha melhor que outra
Percurso solução	Um diagrama com os nós visitados na rota até a solução

No exemplo das chaves perdidas, cada cômodo da casa é um nó; a casa inteira é o espaço de busca; a meta, quando alcançada, é a cozinha; e o percurso solução é mostrado na Figura 23.1. Os quartos e o banheiro são nós terminais porque não levam a lugar nenhum. Esse exemplo não usa heurística, que será vista mais adiante neste capítulo.

Explosões Combinatórias

Nesse ponto, você pode estar pensando que a busca a uma solução é fácil — você parte do início e explora seu caminho até a conclusão. No caso extremamente simples das chaves perdidas, esse é um método eficaz. Mas, na maioria dos problemas que um computador é chamado a resolver, a situação é bem diferente. Em geral, utiliza-se um computador para resolver problemas onde o número de nós no espaço de busca é muito grande e, conforme o espaço de busca cresce, também o faz o número de diferentes percursos possíveis até a meta. O problema é que cada nó adicionado ao espaço de pesquisa acrescenta mais de um percurso. Isto é, o número de caminhos até a meta aumenta mais rapidamente a cada nó acrescido.

Por exemplo, considere o número de formas em que três objetos — A, B e C — podem ser arranjados sobre uma mesa. Os seis arranjos possíveis são

A	B	C
A	C	B
B	C	A
B	A	C
C	B	A
C	A	B

Você pode rapidamente provar a si mesmo que essas são as seis únicas formas de arranjar A, B e C. Porém, o mesmo número pode ser deduzido usando um teorema do ramo da matemática chamado de *análise combinatória* — o estudo de como as coisas podem ser combinadas. De acordo com o teorema, o número de maneiras em que N objetos podem ser permutados é igual a *N*! (N fatorial). O fatorial de um número é o produto de todos os números inteiros iguais ou menores que ele até 1. Assim, 3! é 3x2x1, ou 6. Se você tem quatro objetos para dispor, então há 4! ou 24 combinações. Com cinco objetos, o número é 120 e com seis, 720. Com 1000 objetos o número de combinações possíveis é enorme! O gráfico da Figura 23.2 lhe dá uma idéia visual do que os pesquisadores de inteligência artificial costumam chamar de *explosão combinatória*. Assim, quando há mais de um punhado de possibilidades, rapidamente torna-se impossível examinar (ou mesmo enumerar) todas as combinações.

Em outras palavras, cada nó adicional no espaço de pesquisa aumenta o número de soluções possíveis em um número muito maior que um. Logo, em algum ponto haverá possibilidades demais para se trabalhar. Como o número de possibilidades cresce tão rápido, apenas os mais simples dos problemas se prestam a pesquisas exaustivas. Uma pesquisa exaustiva é aquela que examina todos os nós — algo como uma técnica da "força bruta". Força bruta sempre funciona, mas normalmente não é prática, pois consome muito tempo, muitos recursos computacionais, ou ambos. Por essa razão, outras técnicas de pesquisa foram desenvolvidas pelos pesquisadores.

Técnicas de Pesquisa

Existem diversas maneiras de pesquisar uma possível solução. As mais comuns e mais importantes são

- Pesquisas de profundidade primeiro
- Pesquisas de extensão primeiro
- Pesquisas de escalada da montanha
- Pesquisas de menor custo

Esse capítulo examina cada uma dessas pesquisas.

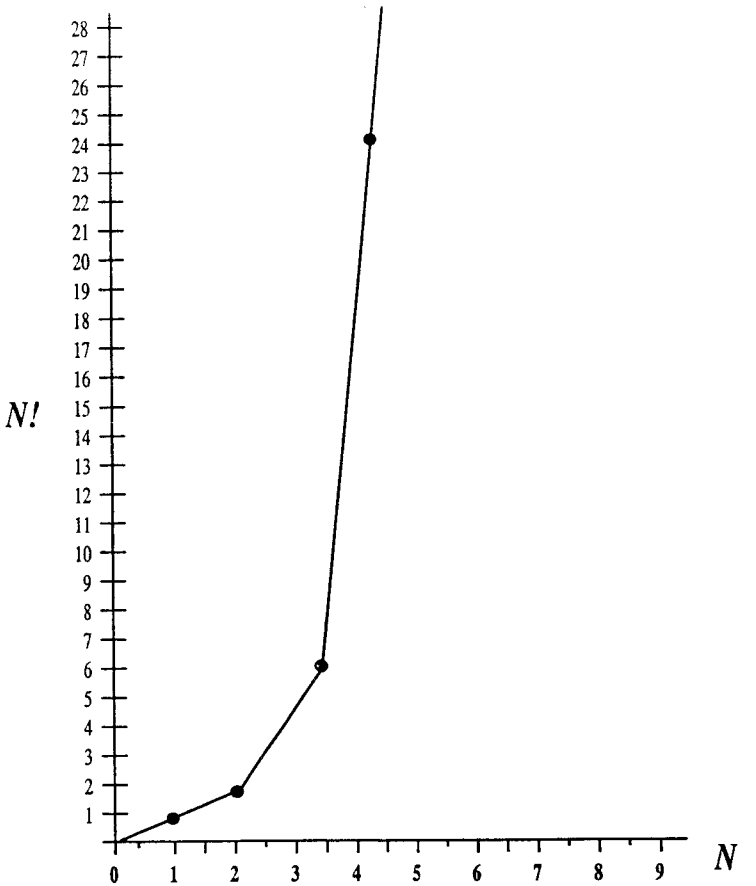


Figura 23.2 Uma explosão combinatória com fatoriais.

Avaliação das Pesquisas

Avaliar o desempenho de uma técnica de pesquisa pode ser bastante complicado. Na verdade, a avaliação das pesquisas forma uma grande parte da inteligência artificial. No entanto, para os nossos propósitos, são apenas duas as medidas mais importantes:

- A velocidade em que a pesquisa encontra a solução
- Quão boa é a solução encontrada

Existem diversos tipos de problema em que tudo o que importa é que uma solução, qualquer solução, seja encontrada, com o mínimo de esforço. Para esses problemas, a primeira medida é importante. Porém, em outras situações, a solução deve ser boa, talvez mesmo ótima.

A velocidade de uma pesquisa é determinada pelo comprimento do percurso da solução e pelo número de nós atravessados. Lembre-se de que retornar de nós sem saída é essencialmente esforço desperdiçado, portanto é desejável uma pesquisa que raramente tenha de retornar.

É necessário entender que existe uma diferença entre encontrar uma solução ótima e encontrar uma boa solução. Encontrar uma solução ótima normalmente está vinculado a uma pesquisa exaustiva, porque essa é a única forma de saber se a melhor solução foi encontrada. Encontrar uma boa solução, por outro lado, significa encontrar uma solução que está submetida a um conjunto de limitações — não importando se existe uma melhor.

Como você poderá observar, todas as técnicas de pesquisa descritas neste capítulo funcionam melhor em certas situações do que em outras. Logo, é difícil dizer se um método de pesquisa é *sempre* superior a outro. Mas algumas técnicas de pesquisa têm uma maior probabilidade de ser melhores no caso médio. Além disso, o modo como o problema é definido pode, algumas vezes, ajudar a escolher um método de pesquisa apropriado.

Primeiro, imagine um problema em que utilizaremos diversos métodos para resolver. Imagine que você seja um agente de viagens e que um cliente um tanto mal-humorado deseje comprar uma passagem de um vôo de New York a Los Angeles na companhia aérea XYZ. Você tenta dizer ao cliente que a XYZ não mantém um vôo direto de New York a Los Angeles, mas o cliente insiste em que a XYZ é a única empresa aérea em que ele viajará. A XYZ escalou os vôos da seguinte forma:

New York a Chicago	1000	milhas
Chicago a Denver	1000	milhas
New York a Toronto	800	milhas
New York a Denver	1900	milhas
Toronto a Calgary	1500	milhas
Toronto a Los Angeles	1800	milhas
Toronto a Chicago	500	milhas
Denver a Urbana	1000	milhas
Denver a Houston	1500	milhas

Houston a Los Angeles	1500	milhas
Denver a Los Angeles	1000	milhas

Você rapidamente vê que há uma maneira de viajar de New York a Los Angeles pela XYZ, utilizando vôos de escala. Então, você vende ao cliente seus vôos.

Sua tarefa é escrever um programa em C que faça a mesma coisa ainda melhor.

Uma Representação Gráfica

As informações de vôos da XYZ podem ser traduzidas para o diagrama direcionado mostrado na Figura 23.3. Um *diagrama direcionado* é simplesmente aquele em que as linhas que conectam cada nó incluem uma seta para indicar a direção do movimento. Em um diagrama direcionado, você não pode viajar na direção contrária à seta.

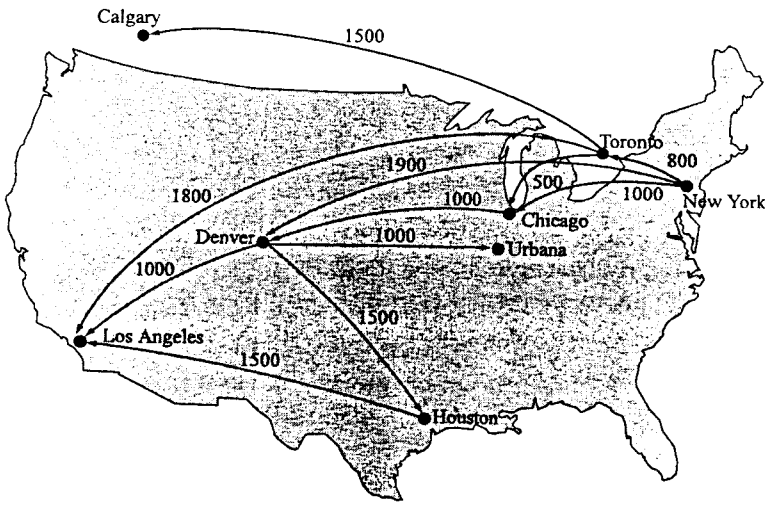


Figura 23.3 Um diagrama direcionado dos vôos da XYZ.

Para tornar as coisas mais fáceis de entender, esse diagrama é redesenhado como árvore na Figura 23.4. Essa versão é usada no restante desta discussão. A meta, Los Angeles, é envolvida por um círculo. Note, também, que várias cidades aparecem mais de uma vez para simplificar a construção do diagrama.

Agora você está pronto para desenvolver os diversos programas de pesquisa para encontrar os caminhos de New York a Los Angeles.

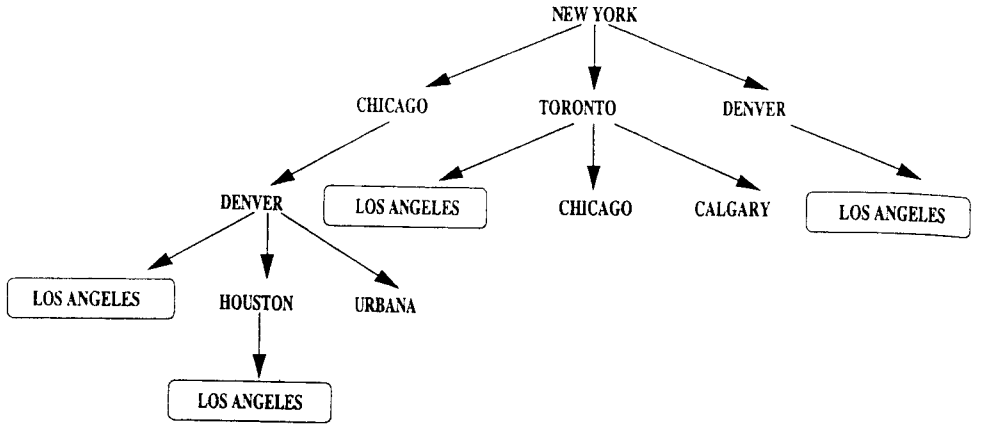
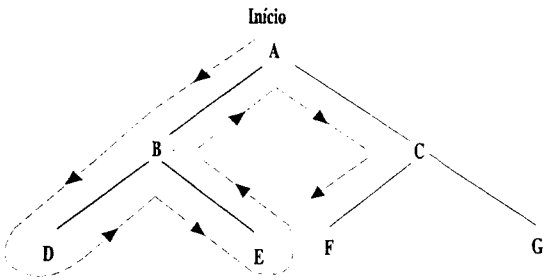


Figura 23.4 Uma versão em árvore dos vôos da XYZ.

A Pesquisa de Profundidade Primeiro

A *pesquisa de profundidade primeiro* explora cada caminho possível até a conclusão (ou meta) antes que outro caminho seja tentado. Para entender exatamente como isso funciona, considere a árvore a seguir. F é a meta.



Uma pesquisa de profundidade primeiro transversaliza o diagrama na seguinte ordem: ABDBEBACF. Se você está familiarizado com árvores, então

reconhece que esse tipo de pesquisa é uma transversalização da árvore de forma ordenada. Isto é, o percurso vai pela esquerda até que um nó terminal seja encontrado ou que a meta seja encontrada. Se um nó terminal é alcançado, o percurso volta um nível, vai à direita, em seguida à esquerda e continua até que a meta ou um nó terminal seja encontrado. Esse procedimento é repetido até que a meta seja encontrada ou até que o último nó do espaço de pesquisa tenha sido examinado.

Como você pode ver, uma pesquisa de profundidade primeiro certamente encontra a meta, porque, no pior caso, ela se degenera em uma pesquisa exaustiva. Nesse exemplo, aconteceria uma pesquisa exaustiva se G fosse a meta.

Escrever um programa em C, para encontrar uma rota de New York a Los Angeles, requer um banco de dados que contenha as informações sobre os vôos da XYZ. Cada entrada no banco de dados deve conter as cidades de origem e de destino, a distância entre elas e um indicador para ajudar no retorno (como você verá em breve). A estrutura seguinte contém estas informações:

```
#define MAX 100

/* estrutura do banco de dados sobre os vôos */
struct FL {
  char from[20]; /* de */
  char to[20]; /* para */
  int distance;
  char skip; /* usado no retorno */
};

struct FL flight[MAX]; /* matriz de estruturas do bd */

int f_pos=0; /* número de entradas do bd dos vôos */
int find_pos=0; /* índice de pesquisa no bd dos vôos */
```

As entradas são colocadas no banco de dados usando a função **assert_fli-ght()**, e **setup()** inicializa a informação. A variável global **f_pos** contém o índice do último item no banco de dados. Essas rotinas são mostradas aqui:

```
void setup(void)
{
  assert_flight("New York", "Chicago", 1000);
  assert_flight("Chicago", "Denver", 1000);
  assert_flight("New York", "Toronto",800);
  assert_flight("New York", "Denver", 1900);
  assert_flight("Toronto", "Calgary", 1500);
```

```
  assert_flight("Toronto", "Los Angeles", 1800);
  assert_flight("Toronto", "Chicago", 500);
  assert_flight("Denver", "Urbana", 1000);
  assert_flight("Denver", "Houston", 1500);
  assert_flight("Houston", "Los Angeles", 1500);
  assert_flight("Denver", "Los Angeles", 1000);
}

/* Coloca os fatos no banco de dados. */
void assert_flight(char *from, char *to, int dist)
{
  if(f_pos<MAX) {
    strcpy(flight[f_pos].from, from);
    strcpy(flight[f_pos].to, to);
    flight[f_pos].distance = dist;
    flight[f_pos].skip = 0;
    f_pos++;
  }
  else printf("Banco de dados de vôo cheio.\n");
}
```

Mantendo o espírito da inteligência artificial, imagine o banco de dados como contendo fatos. O programa a ser desenvolvido usará esses fatos para chegar a uma solução. Por essa razão, muitos pesquisadores de inteligência artificial referem-se ao banco de dados como um *banco de conhecimento*. Este capítulo usa os dois termos sem distinção.

Antes que possa escrever o código real para encontrar uma rota entre New York e Los Angeles, você precisa de diversas funções de suporte. Primeiro, precisa de uma rotina que determine se há um vôo entre as duas cidades. Essa função é chamada de **match()** e devolve zero se não existe o vôo ou devolve a distância entre as duas cidades se há um vôo. Essa rotina é mostrada aqui:

```
/* Se há o vôo entre from e to, então devolve a distância do
   vôo; caso contrário, devolve 0. */
match(char *from, char *to)
{
  register int t;

  for(t=f_pos-1; t>-1; t--)
    if(!strcmp(flight[t].from, from) &&
       !strcmp(flight[t].to, to)) return flight[t].distance;

  return 0; /* não encontrou */
}
```


Outra rotina necessária é **find()**. Dada uma cidade, **find()** pesquisa no banco de dados qualquer conexão. Se uma conexão é encontrada, o nome da cidade de destino e sua distância são devolvidos; caso contrário, zero é devolvido. A rotina **find()** é mostrada a seguir:

```
/* Dado from, encontre anywhere. */
find(char *from, char *anywhere)
{
  find_pos = 0;
  while(find_pos<f_pos) {
    if(!strcmp(flight[find_pos].from, from) &&
     !flight[find_pos].skip) {
      strcpy(anywhere, flight[find_pos].to);
      flight[find_pos].skip = 1; /* torna ativo */
      return flight[find_pos].distance;
    }
    find_pos++;
  }
  return 0;
}
```

Como você pode observar, as cidades que têm o campo **skip** com 1 não são conexões válidas. Além disso, se uma conexão é encontrada, seu campo **skip** é marcado como ativo — isso controla o retorno de caminhos sem saída.

O retorno é um ingrediente crucial em muitas técnicas de inteligência artificial. O retorno é efetuado pelo uso de rotinas recursivas e de uma pilha de retorno. Quase todas as situações de retorno têm operação tipo pilha — isto é, elas são primeira a entrar, última a sair. Conforme um percurso é explorado, nós são colocados na pilha à medida que são encontrados. A cada ponto sem saída, o último nó é retirado da pilha e um novo percurso, a partir desse ponto, é tentado. Esse processo continua até que a meta seja alcançada ou todos os percursos tenham se exaurido. As funções **push()** e **pop()**, que gerenciam a pilha de retorno, são mostradas a seguir. Elas usam as variáveis globais **tos** e **bt_stack** para guardar o apontador ao topo da pilha e a matriz que contém a pilha, respectivamente.

```
/* Rotinas de pilha */
void push(char *from, char *to, int dist)
{
  if(tos<MAX) {
    strcpy(bt_stack[tos].from, from);
    strcpy(bt_stack[tos].to, to);
    bt_stack[tos].dist = dist);
```

```
    tos++;
  }
  else printf("Pilha cheia.\n");
}

void pop(char *from, char *to, int *dist)
{
  if(tos>0) {
    tos--;
    strcpy(from, bt_stack[tos].from);
    strcpy(to, bt_stack[tos].to);
    *dist = bt_stack[tos].dist;
  }
  else printf("Pilha vazia.\n");
}
```

Agora que as rotinas de suporte já foram desenvolvidas, considere o código a seguir. É **isflight()**, a rotina principal para encontrar a rota entre New York e Los Angeles.

```
/* Determina se há uma rota entre from e to. */
void isflight(char *from, char *to)
{
  int d, dist;
  char anywhere[20];

  /* vê no destino */
  if(d=match(from, to)) {
    push(from, to, d);
    return;
  }

  /* tenta outra conexão */
  if(dist=find(from, anywhere)) {
    push(from, to, dist);
    isflight(anywhere, to);
  }
  else if(tos>0) {
    /* retorna */
    pop(from, to, &dist);
    isflight(from, to);
  }
}
```


A rotina opera da seguinte forma. Em primeiro lugar, o banco de dados é verificado por **match()** para ver se existe um vôo entre **from** e **to**. Se existe, a meta já foi encontrada — a conexão é colocada na pilha e a função retorna. Caso contrário, **find()** verifica se há alguma conexão entre **from** e algum outro lugar. Se houver, essa conexão é colocada na pilha e **isflight()** é chamada recursivamente. Esse processo continua até que a meta seja encontrada. O campo **skip** é necessário no retorno para evitar que as mesmas conexões sejam tentadas repetidamente.

Assim, se chamada com Denver e Houston, a primeira parte da rotina obteria sucesso e **isflight()** terminaria. Imagine, porém, que **isflight()** tenha sido chamada com Chicago e Houston. Nesse caso, a primeira parte falharia, porque não há nenhum vôo direto conectando essas duas cidades. A segunda parte seria tentada no intuito de encontrar uma conexão entre a cidade de origem e qualquer outra cidade. Nesse caso, Chicago tem uma conexão com Denver, portanto, **isflight()** é chamada recursivamente com Denver e Houston. Mais uma vez a primeira condição é testada. Nesse momento, é encontrada uma conexão. Finalmente, as chamadas recursivas desenredam-se e **isflight()** termina. Verifique que **isflight()**, como apresentada aqui, realiza uma pesquisa de profundidade primeiro no banco de dados.

É importante observar que **isflight()**, na verdade, não *devolve* a solução — ela a *gera*. Ao sair, **isflight()** deixa na pilha de retorno a rota entre Chicago e Houston — que é a solução. O estado da pilha determina se **isflight()** obteve sucesso ou não. Uma pilha vazia indica falha; de outra forma, a pilha contém a solução. Assim, você precisa de mais uma função para completar o programa. A função é chamada **route()** e ela escreve o percurso a seguir e a distância total. A função **route()** é mostrada aqui:

```
/* Mostra a rota e a distância total. */
void route(char *to)
{
  int dist, t;

  dist = 0;
  t = 0;
  while(t<tos) {
    printf("%s para ", bt_stack[t].from);
    dist += bt_stack[t].dist;
    t++;
  }
  printf("%s\n", to);
  printf("A distância é %d.\n", dist);
}
```

O programa completo de pesquisa de profundidade primeiro é mostrado a seguir. Digite, agora, este programa no seu computador.

```
/* Pesquisa de profundidade primeiro. */
#include <stdio.h>
#include <string.h>

#define MAX 100

/* estrutura do banco de dados sobre os vôos */
struct FL {
  char from[20]; /* de */
  char to[20]; /* para */
  int distance;
  char skip; /* usado no retorno */
};

struct FL flight[MAX]; /* matriz de estruturas do bd */

int f_pos=0; /* número de entradas do bd dos vôos */
int find_pos=0; /* índice de pesquisa no bd dos vôos */
int tos=0; /* topo da pilha */
struct stack {
  char from[20];
  char to[20];
  int dist;
};
struct stack bt_stack[MAX];  /* pilha de retorno */

void setup(void), route(char *to);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

void main(void)
{
  char from[20], to[20];

  setup();

  printf("De?");
  gets(from);
  printf("Para? ");
  gets(to);
```



```
  isflight(from, to);
  route(to);
}

/* Inicializa o banco de dados de vôos. */
void setup(void)
{
  assert_flight("New York", "Chicago", 1000);
  assert_flight("Chicago", "Denver", 1000);
  assert_flight("New York", "Toronto",800);
  assert_flight("New York", "Denver", 1900);
  assert_flight("Toronto", "Calgary", 1500);
  assert_flight("Toronto", "Los Angeles", 1800);
  assert_flight("Toronto", "Chicago", 500);
  assert_flight("Denver", "Urbana", 1000);
  assert_flight("Denver", "Houston", 1500);
  assert_flight("Houston", "Los Angeles", 1500);
  assert_flight("Denver", "Los Angeles", 1000);
}

/* Coloca os fatos no banco de dados */
void assert_flight(char *from, char *to, int dist)
{

  if(f_pos<MAX) {
    strcpy(flight[f_pos].from, from);
    strcpy(flight[f_pos].to, to);
    flight[f_pos].distance=dist);
    flight[f_pos].skip = 0;
    f_pos++;
  }
  else printf("Banco de dados de vôos cheio.\n");
}

/* Mostra a rota e a distância total. */
void route(char *to)

{
  int dist, t;

  dist = 0;
  t = 0;
  while(t<tos) {
    printf("%s para ", bt_stack[t].from);
    dist += bt_stack[t].dist.;
```

```
    t++;
  }
  printf("%s\n", to);
  printf("A distância é: %d.\n", dist);
}

/* Se há o vôo entre from e to, então devolve a distância do
   vôo; caso contrário, devolve 0. */
match(char *from, char *to)
{
  register int t;

  for(t=f_pos-1; t>-1; t--);
    if(!strcmp(flight[t].from, from) &&
       !strcmp(flight[t].to, to)) return flight[t].distance;

  return 0;/* não encontrou */
}

/* Dado from, encontre anywhere. */
find(char *from, char *anywhere)
{
  find_pos = 0;
  while(find_pos<f_pos) {
    if(!strcmp(flight[find_pos].from, from) &&
     !flight[find_pos].skip) {
       strcpy(anywhere, flight[find_pos].to);
       flight[find_pos].skip = 1; /* torna ativo */
       return flight[find_pos].distance;
     }
    find_pos++;
  }
  return 0;
}

/* Determina se há uma rota entre from e to. */
void isflight(char *from, char *to)
{
  int d, dist;
  char anywhere[20];

  /* vê se está no destino */
  if(d=match(from, to)) {
    push(from, to, d);
    return;
```



```
  }


  /* tenta outra conexão */
  if(dist=find(from, anywhere)) {
    push(from, to, dist);
    isflight(anywhere, to);
  }
  else if(tos>0) {
    /* retorna */
    pop(from, to, &dist);
    isflight(from, to);
  }
}

/* Rotinas de pilha */
void push(char *from, char *to, int dist)
{
  if(tos<MAX) {
    strcpy(bt_stack[tos].from, from);
    strcpy(bt_stack[tos].to,to);
    bt_stack[tos].dist=dist;
    tos++;
  }
  else printf("Pilha cheia.\n");
}

void pop(char *from, char *to, int *dist)
{
  if(tos>0) {
    tos--;
    strcpy(from,bt_stack[tos].from);
    strcpy(to,bt_stack[tos].to);
    *dist=bt_stack[tos].dist;
  }
  else printf("Pilha vazia.\n");
}
```

Note que **main()** pede tanto a cidade de origem como a de destino. Isso significa que você pode usar o programa para encontrar rotas entre duas cidades quaisquer. No entanto, o restante deste capítulo assume New York como origem e Los Angeles como destino.

Compile, agora, o programa. Para certos compiladores, incluindo Microsoft C, será necessário aumentar a quantidade de memória alocada para a pilha, porque, para certas soluções, as rotinas são altamente recursivas.

Quando executar com New York como origem e Los Angeles como destino, a solução será

New York para Chicago para Denver para Los Angeles
A distância é 3000.

A Figura 23.5 mostra o percurso da pesquisa.

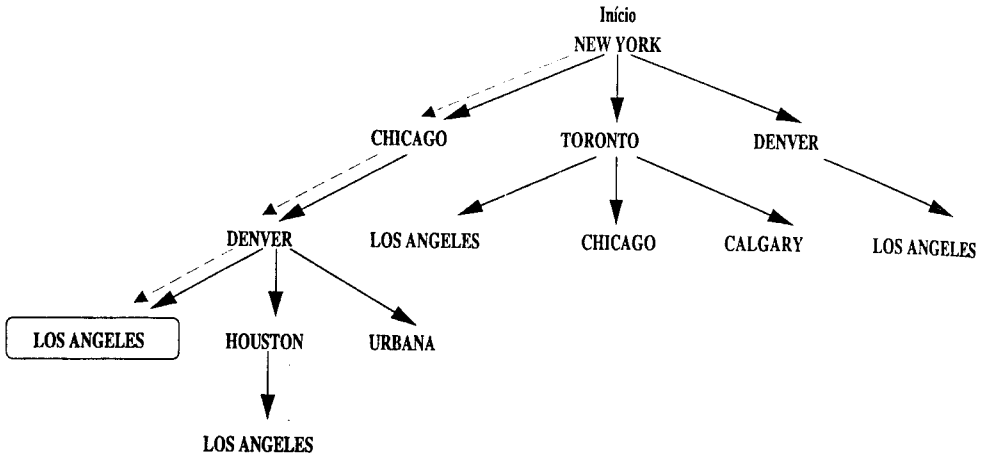


Figura 23.5 O percurso de profundidade primeiro para uma solução.

Se você se dirigir à Figura 23.5, verá que essa é certamente a primeira solução que seria encontrada por uma pesquisa de profundidade primeiro. Não há a solução ótima — que é New York para Denver para Los Angeles, com uma distância de 2600 milhas —, mas não é tão ruim.

Uma Análise da Pesquisa de Profundidade Primeiro

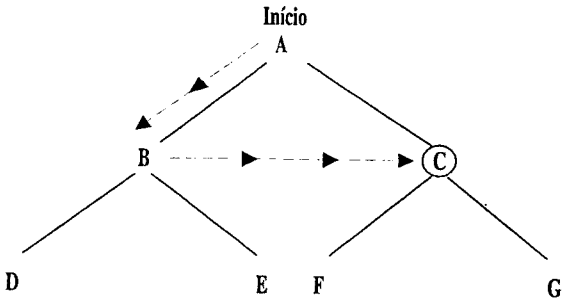
Como você pode observar, a abordagem profundidade primeiro encontrou uma solução razoavelmente boa. Além disso, em relação a esse problema específico, a pesquisa encontrou uma solução na sua primeira tentativa, sem nenhum retorno — isso é muito bom. Mas ela teria de passar por quase todos os nós para chegar à solução ótima — isso não é tão bom.

Note que o desempenho das pesquisas de profundidade primeiro pode ser muito pobre quando um ramo particularmente longo, sem nenhuma solução

no final, é explorado. Nesse caso, uma pesquisa de profundidade primeiro desperdiça um tempo considerável, não apenas explorando essa cadeia como também retornando para atingir a meta.

A Pesquisa de Extensão Primeiro

O oposto da pesquisa de profundidade primeiro é a *pesquisa de extensão primeiro*. Nesse método, cada nó pertencente ao mesmo nível é verificado antes que a pesquisa prossiga no próximo nível mais profundo. Esse método de transversalizar é mostrado aqui com C como meta:



Como você pode ver, os nós A, B e C são visitados. Como a pesquisa de profundidade primeiro, uma pesquisa de extensão primeiro garante uma solução, se existe alguma, porque eventualmente ela se degenera em uma pesquisa exaustiva.

Para fazer o programa que procura rotas executar uma pesquisa de extensão primeiro, é necessário apenas alterar a função **isflight()**, como mostrado aqui:

```
void isflight(char *from, char *to)
{
  int d, dist;
  char anywhere[20];

  while(dist=find(from, anywhere)) {
    /* modificação para extensão primeiro */
    if(d=match(anywhere, to)) {
      push(from, to, dist);
      push(anywhere, to, d);
```

```
        return;
      }
    }
    /* tenta outra conexão */
    if(dist=find(from, anywhere)) {
      push(from, to, dist);
      isflight(anywhere, to);
    }
    else if(tos>0) {
      pop(from, to, &dist);
      isflight(from, to);
    }
  }
```

Como você pode observar, apenas a primeira condição foi alterada. Agora, todas as cidades que se conectam à cidade de partida são verificadas para ver se elas se conectam à cidade de destino.

Substitua essa versão de **isflight()** no programa e execute-o. A solução é

New York para Toronto para Los Angeles
A distância é 2600.

A solução é ótima. A Figura 23.6 mostra o percurso de extensão primeiro até a solução.

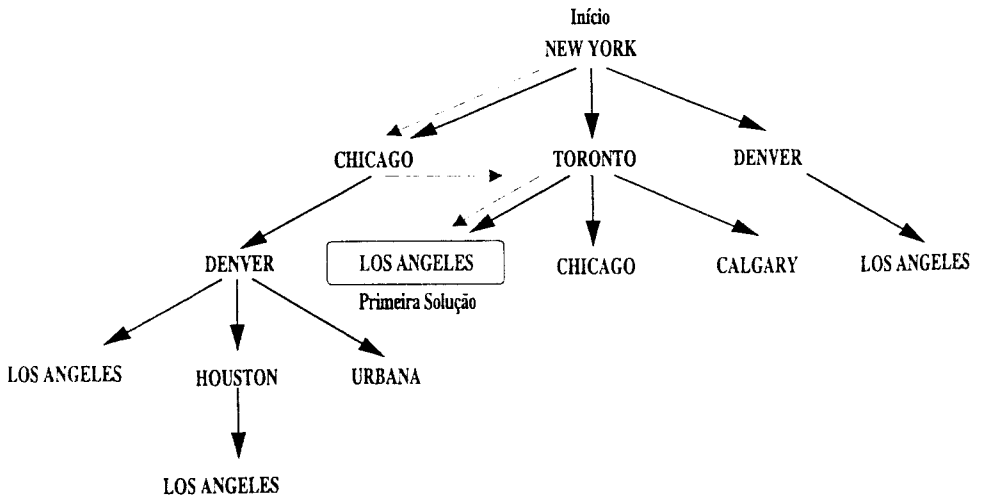


Figura 23.6 O percurso de extensão primeiro para uma solução.

Uma Análise da Pesquisa de Extensão Primeiro

Nesse exemplo, a pesquisa de extensão primeiro foi muito bem-sucedida, encontrando a primeira solução sem retornar, e o seu resultado foi a solução ótima. As três primeiras soluções que seriam encontradas são as três melhores rotas que existem. Porém, lembre-se de que esse resultado não se generaliza a outras situações, porque o percurso depende da organização física da informação e de como ela é armazenada no computador. O exemplo ilustra bem como as pesquisas de profundidade primeiro e de extensão primeiro diferem radicalmente.

Uma desvantagem da pesquisa por extensão primeiro torna-se evidente quando a meta está vários níveis abaixo. Nesse caso, uma pesquisa por extensão primeiro faz um esforço substancial para encontrar a meta. Em geral, a escolha entre uma pesquisa por profundidade primeiro ou por extensão primeiro é feita por meio de uma suposição da posição mais provável da meta.

Adicionando Heurísticas

Você provavelmente deve ter imaginado que as rotinas de pesquisa de profundidade primeiro e por extensão primeiro são cegas. Elas são métodos de procura de uma solução que se baseiam exclusivamente na movimentação de uma meta a outra sem nenhuma hipótese estudada pelo computador. Isso pode ser bom em certas situações controladas, onde se sabe que um método é melhor que outro. Porém, um programa de intelegência artificial generalizado precisa de um procedimento de pesquisa que tenha uma média superior a essas duas técnicas. A única maneira de conseguir essa pesquisa é adicionando heurísticas.

Lembre-se de que heurísticas são simplesmente regras que qualificam a possibilidade de uma pesquisa estar prosseguindo na direção correta. Por exemplo, imagine que você está perdido na selva e precisa de ajuda. A selva é tão densa que não se pode ver nada à frente e as árvores são altas demais para subir e dar uma olhada em volta. Porém, você sabe que rios, fontes e lagos são muito prováveis em vales; que animais freqüentemente fazem caminhos até seus bebedouros; que quando você está perto da água é possível percebê-la; e que se pode ouvir água corrente. Então, você começa movendo-se morro abaixo, porque é improvável que a água esteja morro acima. Logo você cruza com um rastro de veado que também está indo morro abaixo. Sabendo que isso pode levar à água, você segue o rastro. Começa, então, a escutar uma leve corrente à sua esquerda. Sabendo que isso pode ser água, cautelosamente se move naquela direção. Quando você se move, começa a detectar uma maior umidade do ar; você pode sentir a água. Finalmente, você encontra uma fonte e tem a sua água. Como você pode

observar, informação de heurística, embora não seja precisa nem segura, aumenta as chances de que um método de pesquisa encontre uma meta rapidamente, otimamente, ou ambos. Resumindo, ela aumenta as chances em favor de um rápido sucesso.

Você pode pensar que informação de heurística pode facilmente ser incluída em programas designados para aplicações específicas, mas que é impossível criar pesquisas com heurísticas generalizadas. Você verá mais adiante que isso não é verdade.

Na maioria das vezes, os métodos heurísticos de pesquisa são baseados na maximização ou minimização de algum aspecto do problema. As duas abordagens que veremos utilizam heurísticas opostas e levam a resultados diferentes. Ambas as pesquisas serão baseadas na pesquisa de profundidade primeiro.

A Pesquisa da Escalada da Montanha

No problema do vôo de New York a Los Angeles, há duas variáveis possíveis que um passageiro pode querer minimizar. A primeira é o número de escalas que deve ser feito. A segunda é a extensão da rota. A rota mais curta não implica necessariamente o mínimo de escalas. Um algoritmo de pesquisa que tente encontrar, como primeira solução, uma rota que minimiza o número de conexões utiliza uma heurística de que, quanto maior a distância do vôo, maior a probabilidade de o viajante estar mais perto do destino; portanto, o número de escalas é minimizado.

Na linguagem da inteligência artificial, isto é chamado de *escalada da montanha*. O algoritmo da escalada da montanha escolhe como próximo passo o nó que parece colocá-lo mais perto da meta (isso é, o mais longe possível da posição atual). Seu nome é obtido da analogia com um excursionista perdido na escuridão, a meio caminho de uma montanha. Assumindo que o seu acampamento está no topo da montanha, mesmo na escuridão o excursionista sabe que cada passo para cima é um passo na direção correta.

Trabalhando apenas com a informação contida no banco de dados das escalas dos vôos, para se incorporar a heurística da escalada da montanha no programa de rotas, deve-se escolher o vôo da escala que está o mais distante possível da posição atual na esperança de chegar o mais perto do destino. Para fazer isso, modifique a rotina **find()** como mostrado aqui:


```
/* Dado from, encontre o mais distante "anywhere". */
find(char *from, char *anywhere)
{
  int pos, dist;

  pos=dist = 0;
  find_pos = 0;

  while(find_pos<f_pos) {
    if(!strcmp(flight[find_pos].from, from) &&
      !flight[find_pos].skip) {
        if(flight[find_pos].distance>dist) {
        pos = find_pos;
        dist = flight[find_pos].distance;
       }
    }
    find_pos++;
  }
  if(pos) {
    strcpy(anywhere, flight[pos].to)
    flight[pos].skip = 1;
    return flight[pos].distance;
  }
  return 0;
}
```

A rotina **find()** faz agora uma pesquisa no banco de dados inteiro, procurando a escala que está mais distante da cidade de partida.

O programa completo da escalada da montanha é mostrado aqui. Digite agora este programa no seu computador:

```
/* Escalada da montanha */
#include <stdio.h>
#include <string.h>

#define MAX 100

/* estrutura do banco de dados sobre os vôos */
struct FL {
  char from[20]; /* de */
  char to[20]; /* para */
  int distance;
  char skip; /* usado no retorno */
};
```

```
struct FL flight[MAX]; /* matriz de estruturas do bd */

int f_pos=0; /* número de entradas do bd dos vôos */
int find_pos=0; /* índice de pesquisa no bd dos vôos */

int tos=0; /* topo da pilha */
struct stack {
  char from[20];
  char to[20];
  int dist;
};

struct stack bt_stack[MAX];  /* pilha de retorno */

void setup(void); void route(char *to);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

void main(void)
{
  char from[20], to[20];

  setup();

  printf("De? ");
  gets(from);
  printf("Para? ");
  gets(to);

  isflight(from, to);
  route(to);
}

/* Inicializa o banco de dados de vôos. */
void setup(void)
{
  assert_flight("New York", "Chicago", 1000);
  assert_flight("Chicago", "Denver", 1000);
  assert_flight("New York", "Toronto",800);
  assert_flight("New York", "Denver", 1900);
  assert_flight("Toronto", "Calgary", 1500);
```



```
  assert_flight("Toronto", "Los Angeles", 1800);
  assert_flight("Toronto", "Chicago", 500);
  assert_flight("Denver", "Urbana", 1000);
  assert_flight("Denver", "Houston", 1500);
  assert_flight("Houston", "Los Angeles", 1500);
  assert_flight("Denver", "Los Angeles", 1000);
}

/* Coloca os fatos no banco de dados. */
void assert_flight(char *from, char *to, int dist)
{

  if(f_pos<MAX) {
    strcpy(flight[f_pos].from, from);
    strcpy(flight[f_pos].to, to);
    flight[f_pos].distance = dist;
    flight[f_pos].skip = 0;
    f_pos++;
  }
  else printf("Banco de dados de vôos cheio.\n");
}

/* Mostra a rota e a distância total. */
void route(char *to)
{
  int dist, t;

  dist = 0;
  t = 0;
  while(t<tos) {
    printf("%s para ", bt_stack[t].from);
    dist += bt_stack[t].dist;
    t++;
  }
  printf("%s\n", to);
  printf("A distância é: %d.\n", dist);
}

/* Se há o vôo entre from e to, então devolve a distância do
   vôo; caso contrário, devolve 0. */
match(char *from, char *to)
{
  register int t;

  for(t=f_pos-1; t>-1; t--);
```

```
    if(!strcmp(flight[t].from, from) &&
      !strcmp(flight[t].to, to)) return flight[t].distance;
  return 0;          /* não encontrou */
}

/* Dado from, encontre o mais distante "anywhere". */
find(char *from, char *anywhere)
{
  int pos, dist;

  pos=dist = 0;
  find_pos = 0;

  while(find_pos<f_pos) {
    if(!strcmp(flight[find_pos].from, from) &&
      !flight[find_pos].skip) {
        if(flight[find_pos].distance>dist) {
          pos = find_pos;
          dist = flight[find_pos].distance;
        }
    }
    find_pos++;
  }
  if(pos) {
    strcpy(anywhere, flight[pos].to;
    flight[pos].skip = 1;
    return flight[pos].distance;
  }
  return 0;
}

/* Determina se há uma rota entre from e to. */
void isflight(char *from, char *to)
{
  int d, dist;
  char anywhere[20];

  if(d=match(from, to)) {
    /* é a meta */
    push(from, to, d);
    return;
  }

  /* tenta outra conexão */
  if(dist=find(from, anywhere)) {
```



```
    push(from, to, dist);
    isflight(anywhere, to);
  }
  else if(tos>0) {
    pop(from, to, &dist);
    isflight(from, to);
  }
}

/* Rotinas de pilha */
void push(char *from, char *to, int dist)
{
  if(tos<MAX) {
    strcpy(bt_stack[tos].from, from);
    strcpy(bt_stack[tos].to, to);
    bt_stack[tos].dist= dist;
    tos++;
  }
  else printf("Pilha cheia.\n");
}

void pop(char *from, char *to, int *dist)
{
  if(tos>0) {
    tos--;
    strcpy(from, bt_stack[tos].from);
    strcpy(to, bt_stack[tos].to);
    *dist = bt_stack[tos].dist;
  }
  else printf("Pilha vazia.\n");
}
```

Após a execução do programa, a solução é

New York para Denver para Los Angeles
A distância é 2900.

Isso é muito bom! A rota contém o número mínimo de paradas (apenas uma) e está realmente bem perto da rota mais curta. Além disso, o programa atingiu a solução sem desperdício de tempo ou esforço devido a retornos extensivos.

Porém, se a escala de Denver a Los Angeles não existisse, a solução não seria tão boa. O caminho percorrido seria New York para Denver para Houston para Los Angeles — uma distância de 4900 milhas! Essa solução escalou um "falso pico". Como pode ser facilmente observado, a rota a Houston não nos deixa mais perto da meta, que é Los Angeles. A Figura 23.7 mostra a primeira solução e também o percurso do falso pico.

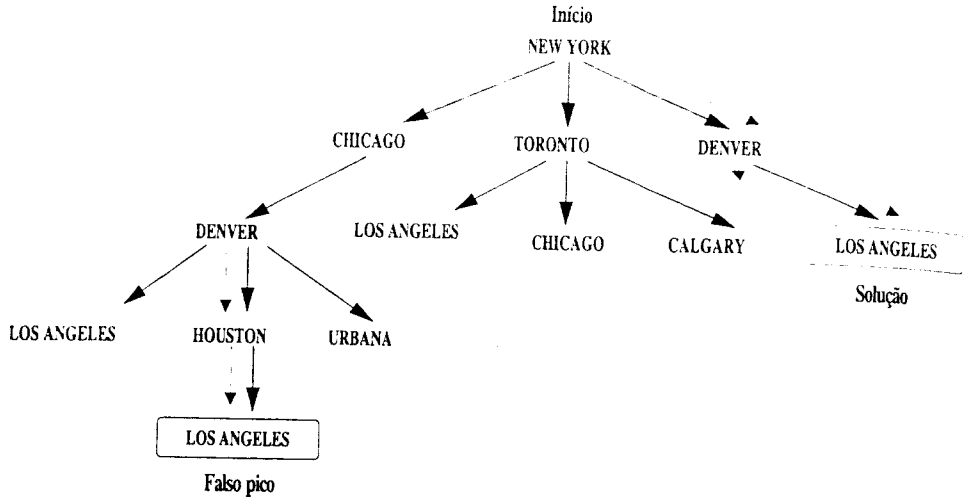


Figura 23.7 O percurso de escalada da montanha a uma solução e a um falso pico.

Análise da Escalada da Montanha

A escalada da montanha fornece resultados razoavelmente bons em muitas circunstâncias, porque ela tende a reduzir o número de nós que precisa ser visitado antes que seja alcançada uma solução. No entanto, ela sofre de três deficiências. Primeiro, há o problema dos falsos picos, como foi visto na segunda solução do exemplo. Nesse caso, tornam-se necessários extensos retornos para encontrar a solução. O segundo problema acontece quando se atinge um planalto, uma situação em que todos os passos seguintes parecem igualmente bons (ou ruins). Nesse caso, a escalada da montanha não é melhor do que a pesquisa de profundidade primeiro. O último problema é o de uma colina. Nesse caso, a escalada da montanha tem um péssimo desempenho, porque o algoritmo faz com que a colina seja atravessada diversas vezes quando ocorre retorno.

Apesar desses problemas potenciais, a escalada da montanha geralmente leva a soluções mais próximas da solução ótima do que qualquer um dos métodos que não utilizam heurística.

A Pesquisa por Menor Esforço

O oposto da pesquisa por escalada da montanha é a *pesquisa por menor esforço*. Essa estratégia é similar a estar no meio de uma grande ladeira usando patins de rodas; tem-se a exata sensação de que é muito mais fácil descer do que subir! Em outras palavras, uma pesquisa por menor esforço toma o caminho de menor resistência.

Aplicar a pesquisa por menor esforço ao problema dos vôos implica que o vôo de conexão mais curto é tomado em todos os casos, de forma que a rota encontrada tem uma boa chance de cobrir a menor distância. Ao contrário da escalada da montanha, que minimizava o número de escalas, uma pesquisa por menor esforço minimiza a distância entre a origem e o destino.

Para utilizar uma pesquisa por menor esforço, deve-se alterar **find()**, novamente como mostrado aqui:

```
/* Encontra o "anywhere" mais próximo. */
find(char *from, char *anywhere)
{
  int pos, dist;

  pos = 0;
  dist = 32000; /* maior que a maior rota */
  find_pos = 0;

  while(find_pos<f_pos) {
    if(!strcmp(flight[find_pos].from, from) &&
      !flight[find_pos].skip) {
        if(flight[find_pos].distance<dist) {
        pos = find_pos;
        dist = flight[find_pos].distance;
      }
    }
    find_pos++;
  }
  if(pos) {
    strcpy(anywhere, flight[pos].to);
    flight[pos].skip = 1;
    return flight[pos].distance;
  }
  return 0;
}
```

Utilizando essa versão de **find()**, a solução encontrada é

New York para Toronto para Los Angeles
A distância é 2600.

Como você pode observar, a pesquisa realmente encontrou a rota mais curta. A Figura 23.8 mostra o percurso de menor esforço até a meta.

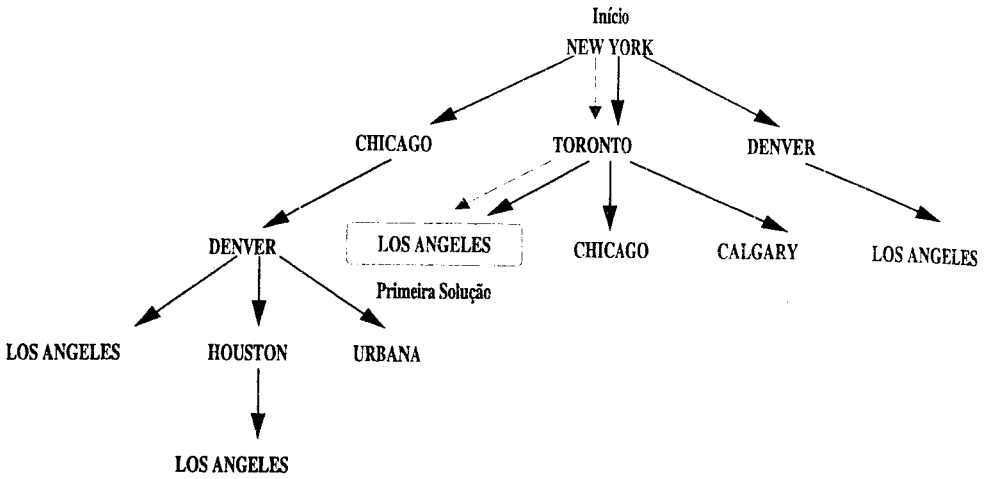


Figura 23.8 O percurso de menor esforço para uma solução.

Análise da Pesquisa por Menor Esforço

A pesquisa por menor esforço e a escalada da montanha têm as mesmas vantagens e desvantagens, porém inversas. Podem ocorrer vales falsos, baixadas e desfiladeiros, mas a pesquisa por menor esforço normalmente opera razoavelmente bem. No entanto, não se deve assumir que, apenas porque a pesquisa por menor esforço obteve melhor resultado do que a escalada da montanha nesse problema, ela seja melhor. Tudo o que se pode dizer é que, no caso médio, a pesquisa por menor esforço tem melhor desempenho do que uma pesquisa cega.

Escolhendo uma Técnica de Pesquisa

Como foi visto, as técnicas heurísticas, em média, operam melhor do que uma pesquisa cega. Porém, nem sempre é possível usar uma pesquisa heurística porque pode não haver informação suficiente para qualificar a probabilidade de o próximo passo estar no caminho para a meta. Portanto, as regras para escolher um método de pesquisa são separadas em duas categorias: uma para os problemas que podem utilizar uma pesquisa heurística e uma para aqueles que não podem.

Se você não pode aplicar heurística a um problema, a pesquisa por profundidade primeiro é normalmente a melhor abordagem. A única exceção advém de quando se sabe algo que indica que uma pesquisa por extensão primeiro será melhor.

A escolha entre a escalada da montanha e a pesquisa por menor esforço baseia-se em decidir que condição deve ser minimizada ou maximizada. Em geral, a escalada da montanha produz uma solução com o mínimo de nós visitados, mas a pesquisa por menor esforço encontra um percurso que requer o menor empenho.

Se você procura uma solução quase ótima, mas não pode aplicar uma pesquisa exaustiva pelas razões já expostas, um método efetivo é aplicar cada uma das quatro pesquisas e utilizar a melhor solução. Uma vez que todas as pesquisas operam de formas substancialmente diferentes, uma deve produzir um resultado melhor que as outras.

Encontrando Múltiplas Soluções

Algumas vezes é valioso encontrar diversas soluções para o mesmo problema. Isso não é o mesmo que encontrar todas as soluções como em uma pesquisa exaustiva. Por exemplo, pense no projeto da casa dos seus sonhos. Você precisa esboçar diversas plantas baixas para ajudá-lo a decidir o melhor projeto, mas você não pode esboçar todas as plantas possíveis. Em resumo, soluções múltiplas podem ajudá-lo a ver muitas maneiras diferentes de alcançar uma solução antes de implementá-la.

Existem diversas maneiras de gerar mais de uma solução, mas apenas duas são examinadas aqui. A primeira é a remoção de percurso e a segunda, a remoção de nó. Como seus nomes indicam, para gerar mais de uma solução, sem redundância, é necessário que as soluções já encontradas sejam removidas do sistema. Lembre-se de que esses métodos não tentam (nem podem ser usados para) encontrar todas as soluções. Encontrar todas as soluções é um problema diferente que normalmente não é tentado, porque implica uma pesquisa exaustiva.

Remoção de Percurso

O método de *remoção de percurso* para gerar mais de uma solução remove todos os nós que formam uma solução atual do banco de dados e, então, tenta encontrar outra solução. Em resumo, a remoção de percurso corta galhos da árvore.

Para encontrar múltiplas soluções, utilizando remoção de percurso, é necessário apenas alterar **main()** na pesquisa de profundidade primeiro, como mostrado aqui:

```
void main(void)
{
  char from[20], to[20];

  setup();

  printf("De? ");
  gets(from);
  printf("Para? ");
  gets(to);
  do {
    isflight(from, to);
    route(to);
    tos = 0; /* reinicializa a pilha de retorno */
  } while(getche()!='q');
}
```

Qualquer conexão que faça parte de uma solução terá seu campo **skip** marcado. Conseqüentemente, essa escala não pode mais ser encontrada por **find()** e todas as escalas em uma solução são removidas. É necessário apenas zerar **tos**, o que, efetivamente, limpa a pilha de retorno.

O método de remoção de percurso encontra as seguintes soluções:

New York para Chicago para Denver para Los Angeles
A distância é 3000.

New York para Toronto para Los Angeles
A distância é 2600.

New York para Denver para Los Angeles
A distância é 2900.

A pesquisa encontrou as três melhores soluções. Porém, esse resultado não pode ser generalizado, porque ele é baseado na forma em que os dados são colocados no banco de dados e a situação real sob estudo.

Remoção de Nó

A segunda maneira de forçar a produção de soluções adicionais, a *remoção do nó*, simplesmente remove o último nó do percurso solução atual e tenta novamente. Para fazer isso, a função **main()** deve retirar o último nó da pilha de retorno e removê-lo do banco de dados, utilizando uma nova função chamada **retract()**. Além disso, todos os campos **skip** devem ser zerados, utilizando-se **clearmarkers()**, e deve-se limpar a pilha de retorno. As funções **main()**, **clearmarkers()** e **retract()** são mostradas a seguir:

```
void main(void)
{
  char from[20], to[20], c1[20], c2[20];
  int d;

  setup();

  printf("De? ");
  gets(from);
  printf("Para? ");
  gets(to);
  do {
    isflight(from, to);
    route(to);
    clearmarkers(); /* reinicializa o banco de dados */
    if(tos>0) pop(c1, c2, &d);
    retract(c1, c2); /* remove o último nó do banco de dados */
    tos = 0; /* reinicializa a pilha de retorno */
  } while(getche()!='q');
}

/* Reinicializa o campo "skip" - isto é, reativa todos os nós, */
void clearmarkers()
{
  int t;

  for(t=0; t<f_pos; ++t) flight[t].skip = 0;
}

/* Remove uma entrada do banco de dados. */
void retract(char *from, char *to)

{
  int t;
```

```
  for(t=0; t<f_pos; t++)
    if(!strcmp(flight[t].from, from) &&
      !strcmp(flight[t].to, to)) {
        strcpy(flight[t].from, "");
        return;
    }
}
```

Como você pode observar, para retirar uma cidade, simplesmente utiliza-se uma string de comprimento zero para o nome da cidade. Para sua comodidade, o programa de remoção de nó completo é mostrado aqui:

```
/* Profundidade primeiro com multiplas soluções usando
   remoção de nó */
#include <stdio.h>
#include <string.h>
#include <conio.h>

#define MAX 100

/* estrutura do banco de dados sobre os vôos */
struct FL {
  char from[20]; /* de */
  char to[20]; /* para */
  int distance;
  char skip; /* usado no retorno */
};

struct FL flight[MAX]; /* matriz de estruturas do bd */

int f_pos=0; /* número de entradas do bd dos vôos */
int find_pos=0; /* índice de pesquisa no bd dos vôos */

int tos=0; /* topo da pilha */
struct stack {
  char from[20];
  char to[20];
  int dist;
};
struct stack bt_stack[MAX];  /* pilha de retorno */

void retract(char *from, char *to);
void clearmarkers(void);
```



```
void setup(void); route(char *to);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

void main(void)
{
  char from[20], to[20], c1[20], c2[20];
  int d;

  setup();

  printf("De? ");
  gets(from);
  printf("Para? ");
  gets(to);
  do {
    isflight(from, to);
    route(to);
    clearmarkers(); /* reinicializa o banco de dados */
    if(tos>0) pop(c1, c2, &d);
    retract(c1, c2); /* remove o último nó do banco de dados */
    tos = 0; /* reinicializa a pilha de retorno */
  } while(getche()!='q');
}

/* Inicializa o banco de dados de vôos. */
void setup(void)
{
  assert_flight("New York", "Chicago", 1000);
  assert_flight("Chicago", "Denver", 1000);
  assert_flight("New York", "Toronto",800);
  assert_flight("New York", "Denver", 1900);
  assert_flight("Toronto", "Calgary", 1500);
  assert_flight("Toronto", "Los Angeles", 1800);
  assert_flight("Toronto", "Chicago", 500);
  assert_flight("Denver", "Urbana", 1000);
  assert_flight("Denver", "Houston", 1500);
  assert_flight("Houston", "Los Angeles", 1500);
  assert_flight("Denver", "Los Angeles", 1000);
}
```

```
/* Coloca os fatos no banco de dados. */
void assert_flight(char *from, char *to, int dist)
{
  if(f_pos<MAX) {
    strcpy(flight[f_pos].from, from);
    strcpy(flight[f_pos].to, to);
    flight[f_pos].distance = dist;
    flight[f_pos].skip = 0;
    f_pos++;
  }
  else printf("Banco de dados de vôos cheio.\n");
}

/* Reinicializa o campo "skip" - isto é, reativa todos os nós. */
void clearmarkers()
{
  int t;

  for(t=0; t<f_pos; ++t) flight[t].skip = 0;
}

/* Remove uma entrada do banco de dados. */
void retract(char *from, char *to)
{
  int t;

  for(t=0; t<f_pos; t++)
    if(!strcmp(flight[t].from, from) &&
      !strcmp(flight[t].to, to)) {
          strcpy(flight[t].from,"");
          return;
    }
}

/* Mostra a rota e a distância total. */
void route(char *to)
{
  int dist, t;

  dist = 0;
  t = 0;
  while(t<tos) {
    printf("%s para ", bt_stack[t].from);
    dist += bt_stack[t].dist;
```



```
    t++;
  }
  printf("%s\n", to);
  printf("A distância é: %d.\n", dist);
}

/* Dado from, encontre anywhere. */
find(char *from, char *anywhere)
{
  find_pos = 0;
  while(find_pos<f_pos) {
    if(!strcmp(flight[find_pos].from, from) &&
     !flight[find_pos].skip) {
       strcpy(anywhere, flight[find_pos].to);
       flight[find_pos].skip = 1;
       return flight[find_pos].distance;
     }
    find_pos++;
  }
  return 0;
}

/* Se há o vôo entre from e to, então devolve a distância do
   vôo; caso contrário, devolve 0. */
match(char *from, char *to)
{
  register int t;

  for(t=f_pos-1; t>-1; t--)
    if(!strcmp(flight[t].from, from) &&
      !strcmp(flight[t].to, to)) return flight[t].distance;

  return 0; /* não encontrou */
}

/* Determina se há uma rota entre from e to. */
void isflight(char *from, char *to)
{
  int d, dist;
  char anywhere[20];

  if(d=match(from, to)) {
    push(from, to, d);/* distance */
    return;
  }
```

```
    if(dist=find(from, anywhere)) {
      push(from, to, dist);
      isflight(anywhere, to);
    }
    else if(tos>0) {
      pop(from, to, &dist);
      isflight(from, to);
    }
}

/* Rotinas de pilha */
void push(char *from, char *to, int dist)
{
  if(tos<MAX) {
    strcpy(bt_stack[tos].from, from);
    strcpy(bt_stack[tos].to, to);
    bt_stack[tos].dist= dist;
    tos++;
  }
  else printf("Pilha cheia.\n");
}

void pop(char *from, char *to, int *dist)
{
  if(tos>0) {
  tos--;
    strcpy(from, bt_stack[tos].from);
    strcpy(to, bt_stack[tos].to);
    *dist = bt_stack[tos].dist;
  }
  else printf("Pilha vazia.\n");
}
```

Utilizando esse método, são produzidas as seguintes soluções:

New York para Chicago para Denver para Los Angeles
A distância é 3000.

New York para Chicago para Denver para Houston para Los Angeles
A distância é 5000.

New York para Toronto para Los Angeles
A distância é 2600.

Nesse caso, a segunda solução é a pior rota possível, mas a solução ótima ainda foi encontrada. Porém, lembre-se de que não se pode generalizar esses resultados, porque eles são baseados tanto na organização física dos dados como na situação específica sob estudo.

Encontrando a Solução Ideal

Todas as técnicas anteriores de pesquisa estavam interessadas em encontrar uma solução. Como foi visto nas pesquisas com heurística, havia o esforço de aumentar a probabilidade de se encontrar uma boa (e, talvez, a ótima) solução. No entanto, há momentos em que apenas a solução ótima interessa. Nessa nossa discussão, "ótimo" significa simplesmente a melhor rota que pode ser encontrada, utilizando-se uma das diversas técnicas de geração de múltiplas soluções, e essa pode não ser realmente a melhor solução. (Encontrar a verdadeira solução ótima requer uma pesquisa exaustiva, que exige um gasto de tempo proibitivo.)

Antes de deixar o já bem explorado exemplo dos vôos, considere um programa que encontra o melhor roteiro de viagem com a condição de que a distância deve ser minimizada. Emprega-se, nesse programa, o método de remoção de percurso para a geração de múltiplas soluções e utiliza-se a pesquisa por menor esforço para minimizar a distância.

A maneira de encontrar o menor roteiro é armazenar um solução apenas se ela tiver uma distância menor do que a anterior. Assim, quando não há mais soluções a gerar, resta a solução ótima.

Para que isso seja realizado, deve-se fazer uma mudança na função **route()** e criar uma pilha extra. A nova pilha contém a solução atual, e, no final, a solução ótima. A nova pilha é chamada de **solution** e a função **route()** modificada é mostrada aqui:

```
/* Encontra a menor distância. */
route(void)
{
  int dist, t;
  static int old_dist=32000;

  if(!tos) return 0; /* feito */
  t = 0;
  dist = 0;
  while(t<tos) {
    dist += bt_stack[t].dist;
```

```
    t++;
  }

  /* se menor, então ache nova solução */
  if(dist<old_dist && dist) {
    t = 0;
    old_dist = dist;
    stos = 0; /* limpa a rota antiga da pilha de posição */
    while(t<tos) {
      spush(bt_stack[t].from, bt_stack[t].to, bt_stack[t].dist);
      t++;
    }
  }
  return dist;
}
```

O programa completo é mostrado a seguir. Observe as modificações em **main()** e o acréscimo de **spush()**, que coloca os novos nós da solução na pilha de solução.

```
/* Solução ótima usando menor esforço com remoção de
   percurso. */
#include <stdio.h>
#include <string.h>

#define MAX 100

/* estrutura do banco de dados sobre os vôos */
struct FL {
  char from[20]; /* de */
  char to[20];   /* para */
  int distance;
  char skip; /* usado no retorno */
};

struct FL flight[MAX]; /* matriz de estruturas do bd */

int f_pos=0; /* número de entradas do bd dos vôos */
int find_pos=0; /* índice de pesquisa no bd dos vôos */

int tos=0;  /* topo da pilha */
int stos=0; /* topo da pilha de solução */
```



```
struct stack {
  char from[20];
  char to[20];
  int dist;
};

struct stack bt_stack[MAX];  /* pilha de retorno */
struct stack solution[MAX];  /* guarda soluções temporárias */

void setup(void);
int route(void);
void assert_flight(char *from, char *to, int dist);
void push(char *from, char *to, int dist);
void pop(char *from, char *to, int *dist);
void isflight(char *from, char *to);
void spush(char *from, char *to, int dist);
int find(char *from, char *anywhere);
int match(char *from, char *to);

void main(void)
{
  char from[20], to[20];
  int t, d;

  setup();

  printf("De? ");
  gets(from);
  printf("Para? ");
  gets(to);
  do {
    isflight(from, to);
    d = route();
    tos = 0; /* reinicializa a pilha de retorno */
  } while(d!=0); /* enquanto estiver encontrando soluções */

  t = 0;
  printf("A solução ótima é: \n");
  while(t<stos) {
    printf("%s para", solution[t].from);
    d += solution[t].dist;
    t++;
  }
  printf("%s\n", to);
  printf("A distância é %d.\n", d);
```

```
}
/* Inicializa o banco de dados de vôos. */
void setup(void)
{
  assert_flight("New York", "Chicago", 1000);
  assert_flight("Chicago", "Denver", 1000);
  assert_flight("New York", "Toronto",800);
  assert_flight("New York", "Denver", 1900);
  assert_flight("Toronto", "Calgary", 1500);
  assert_flight("Toronto", "Los Angeles", 1800);
  assert_flight("Toronto", "Chicago", 500);
  assert_flight("Denver", "Urbana", 1000);
  assert_flight("Denver", "Houston", 1500);
  assert_flight("Houston", "Los Angeles", 1500);
  assert_flight("Denver", "Los Angeles", 1000);
}

/* Coloca os fatos no banco de dados. */
void assert_flight(char *from, char *to, int dist)
{
  if(f_pos<MAX) {
    strcpy(flight[f_pos].from, from);
    strcpy(flight[f_pos].to, to);
    flight[f_pos].distance = dist;
    flight[f_pos].skip = 0;
    f_pos++;
  }
  else printf("Banco de dados de vôos cheio.\n");
}

/* Encontra a menor distância. */
route(void)
{
  int dist, t;
  static int old_dist = 32000;

  if(!tos) return 0; /* feito */
  t = 0;
  dist = 0;
  while(t<tos) {
    dist +=bt_stack[t].dist;
    t++;
  }
  /* se menor, então ache nova solução */
```



```
  if(dis<old_dist && dist) {
    t = 0;
    old_dist = dist;
    stos = 0; /* limpa a rota antiga da pilha de posição */
    while(t<tos) {
      spush(bt_stack[t].from, bt_stack[t].to, bt_stack[t].dist);
      t++;
    }
  }
  return dist;
}

/* Se há o vôo entre from e to, então devolve a distância do
   vôo; caso contrário, devolve 0. */
match(char *from, char *to)
{
  register int t;

  for(t=f_pos-1; t>-1; t--);
    if(!strcmp(flight[t].from, from) &&
      !strcmp(flight[t].to, to)) return flight[t].distance;

  return 0; /* não encontrou */
}

/* Dado from, encontra anywhere. */
find(char *from, char *anywhere)
{
  find_pos=0;
  while(find_pos<f_pos) {
    if(!strcmp(flight[find_pos].from, from) &&
     !flight[find_pos].skip) {
       strcpy(anywhere, flight[find_pos].to);
       flight[find_pos].skip = 1;
       return flight[find_pos].distance;
    }
    find_pos++;
  }
  return 0;
}

/* Determina se há uma rota entre from e to. */
void isflight(char *from, char *to)
{
  int d, dist;
```

```
    char anychere [20];

    if(d=match(from, to)) {
      push(from, to, d);  /*distância*/
      return;
    }

    if(dist=find(from, anywhere)) {

      push(from, to, dist);
      isflight(anywhere, to);
    }
    else if(tos>0) {
      pop(from, to, &dist);
      isflight(from, to);
    }
}

/* Rotinas de pilha */
void push(char *from, char *to, int dist)
{
  if(tos<MAX) {
    strcpy(bt_stack[tos].from, from);
    strcpy(bt_stack[tos].to, to);
    bt_stack[tos].dist= dist;
    tos++;
  }
  else printf("Pilha cheia.\n");
}

void pop(char *from, char *to, int *dist)
{
  if(tos>0) {
    tos--;
    strcpy(from, bt_stack[tos].from);
    strcpy(to, bt_stack[tos].to);
    *dist = bt_stack[tos].dist;
  }
  else printf("Pilha vazia.\n");
}

/* Pilha de solução */
void spush(char *from, char *to, int dist)
{
  if(stos<MAX) {
```



```
    strcpy(solution[stos].from, from);
    strcpy(solution[stos].to, to);
    solution[stos].dist = dist;
    stos++;
  }
  else printf("Pilha de menor distância cheia.\n");
}
```

No método anterior, todos os percursos são seguidos até a sua conclusão. Um método mais aperfeiçoado pararia de seguir o percurso assim que a extensão se igualasse ou excedesse o mínimo atual. Talvez você queira modificar o programa acrescentando essa melhoria.

De Volta às Chaves Perdidas

Para concluir este capítulo sobre solução de problemas, parece muito apropriado apresentar um programa em C que encontre as chaves do carro, perdidas como descrito no primeiro exemplo. O código que o acompanha emprega as mesmas técnicas utilizadas na solução do problema de encontrar uma rota entre duas cidades. Agora, você deve ter um conhecimento razoável de como utilizar C para resolver problemas, de forma que esse programa é apresentado sem maiores explicações.

```
/* Encontra as chaves usando uma pesquisa de profundidade
   primeiro.*/
#include <stdio.h>
#include <string.h>

#define MAX 100

/* Estrutura  do banco de dados das chaves */
struct FL {
  char from[20];
  char to[20];
  char skip;
};

struct FL keys[MAX]; /* matriz de estruturas do banco de dados *

int f_pos=0; /* número de cômodos na casa */
int find_pos=0; /* índice do bd de busca */
```

```
int tos=0; /* topo da pilha */
struct stack {
  char from[20];
  char to[20];
};
struct stack bt_stack[MAX];  /* pilha de retorno */

void setup(void), route(void);
void assert_keys(char *from, char *to);
void push(char *from, char *to);
void pop(char *from, char *to,);
void iskeys(char *from, char *to);
int find(char *from, char *anywhere);
int match(char *from, char *to);

void main(void)
{
  setup();
  iskeys("front_door", "keys");
  route();
}

/* Inicializa o banco de dados. */
void setup(void)
{
  assert_keys("front_door", "lr");
  assert_keys("lr", "banheiro");
  assert_keys("lr", "hall");
  assert_keys("hall", "bd1");
  assert_keys("hall", "bd2");
  assert_keys("hall", "mb");
  assert_keys("lr", "cozinha");
  assert_keys("cozinha", "keys");
}

/* Coloca os fatos no banco de dados. */
void assert_keys(char *from, char *to)
{
  if(f_pos<MAX) {
    strcpy(keys[f_pos].from, from);
    strcpy(keys[f_pos].to, to);
    keys[f_pos].skip = 0;
    f_pos++;
  }
  else printf("Banco de dados das chaves cheio.\n");
}
```



```
}
/* Mostra a rota das chaves. */
void route(void)
{
  int t;

  t = 0;
  while(t<tos) {
    printf("%s", bt_stack[t].from);
    t++;
    if(t<tos) printf(" para ");
  }
  printf("\n");
}

/* Verifica se há uma coincidência. */
match(char *from, char *to)
{
  register int t;

  for(t=f_pos-1; t>-1; t--);
    if(!strcmp(keys[t].from, from) &&
       !strcmp(keys[t].to, to)) return 1;

  return 0; /* não encontrou */
}

/* Dado from, encontre anywhere. */
find(char *from, char *anywhere)
{
  find_pos = 0;
  while(find_pos<f_pos) {
    if(!strcmp(keys[find_pos].from, from) &&
     !keys[find_pos].skip) {
       strcpy(anywhere, keys[find_pos].to);
       keys[find_pos].skip = 1;
       return 1;
    }
    find_pos++;
  }
  return 0;
}

/* Determina se há uma rota entre from e to. */
void iskeys(char *from, char *to)
```

```
{
  char anywhere[20];

  if(match(from, to)) {
    push(from, to); /* distância */
    return;
  }

  if(find(from, anywhere)) {
    push(from, to);
    iskeys(anywhere, to);
  }
  else if(tos>0) {
    pop(from, to);
    iskeys(from, to);
  }
}

/* Rotinas de pilha */
void push(char *from, char *to)
{
  if(tos<MAX) {
    strcpy(bt_stack[tos].from, from);
    strcpy(bt_stack[tos].to, to);
    tos++;
  }
  else printf("Pilha cheia.\n");
}

void pop(char *from, char *to)
{
  if(tos>0) {
    tos--;
    strcpy(from, bt_stack[tos].from);
    strcpy(to, bt_stack[tos].to);
  }
  else printf("Pilha vazia.\n");
}
```

MAKRON Books

24

Construindo o Esqueleto de um Programa Windows 95

C é *a* linguagem de programação para Windows. Como tal, parece bastante apropriado incluir um exemplo de um programa Windows neste livro. No entanto, Windows é um ambiente muito grande e complexo de se programar. De fato, só a descrição do Windows exige quase 2.000 páginas de documentação! Embora não seja possível descrever todos os detalhes necessários para escrever uma aplicação Windows em um capítulo, é possível introduzir os elementos básicos comuns a todas as aplicações. Mais ainda, estes elementos básicos podem ser combinados em um esqueleto de uma aplicação Windows mínima que pode ser usado como os fundamentos de seus próprios programas Windows.

O Windows já teve diversas encarnações desde que foi lançado. No momento em que este texto estava sendo escrito para versão atual de Windows era o Windows 95. O material neste capítulo está ajustado especificamente a esta versão. No entanto, se você possui uma versão mais antiga ou mais nova, a maior parte da discussão será aplicável assim mesmo.

NOTA: Este capítulo foi adaptado do meu livro Programando em C e C++ com o Windows 95, *Makron Books, 1996. Se estiver interessado em aprender mais sobre a programação para Windows 95, você achará este livro especialmente útil. Você também achará útil a Osborne Windows Programming Series, volumes 1, 2 e 3, de Schildt, Pappas e Murray Berkeley, CA: Osborne/McGraw-Hill, 1994.*

Para começar, este capítulo apresenta a perspectiva da programação Windows 95.

A Perspectiva da Programação Windows 95

O objetivo do Windows 95 (e do Windows em geral) é o de permitir a uma pessoa que tenha uma familiaridade básica com o sistema sentar-se e executar virtualmente qualquer aplicação sem a necessidade de um treinamento prévio. Para atingir este objetivo, Windows apresenta uma interface de usuário consistente. Na teoria, se você sabe usar um programa Windows, sabe usar todos eles. Claro que na realidade a maioria dos programas úteis ainda exigirá algum treinamento para ser usada de forma efetiva, mas ao menos esta instrução pode ser reduzida ao *que* o programa *faz*, não a *como* o usuário *interage* com ele. De fato, uma grande parte do código de uma aplicação Windows existe apenas para suportar a interface de usuário.

Antes de continuar, deve ser dito que nem todo programa que execute sob Windows 95 deve necessariamente apresentar ao usuário uma interface estilo Windows. É possível escrever aplicações Windows que não tirem proveito dos elementos de interface de usuário do Windows. Para criar um programa estilo Windows, você deve fazer isto propositalmente. Somente aqueles programas escritos para tirar vantagem do Windows aparecerão como os programas Windows. Embora você possa substituir esta filosofia básica de projeto do Windows, é bom que tenha um bom motivo para tanto, porque os usuários do seu programa muito provavelmente serão perturbados por este fato. Em geral, qualquer aplicação que você esteja escrevendo para o Windows 95 deve utilizar a interface Windows normal e seguir as práticas de projeto padronizadas pelo Windows.

O Windows 95 é gráfico, o que significa que provê uma interface gráfica de usuário (GUI — Graphical User Interface). Embora o hardware gráfico e os modos de vídeo sejam bastante diversificados, muitas das diferenças são tratadas pelo Windows. Isto significa que, na maior parte, seu programa não precisa preocupar-se sobre que tipo de hardware para gráficos ou modo de vídeo está sendo usado. No entanto, por causa da orientação gráfica, você como programador tem responsabilidade adicional ao criar uma aplicação Windows.

Vamos dar uma olhada em algumas das funções mais importantes do Windows 95.

O Modelo da Mesa de Trabalho

Com poucas exceções, o ponto de vista de uma interface de usuário baseada em janelas é o de fornecer o equivalente a uma *mesa de trabalho* (*desktop*, em inglês) na tela. Sobre uma mesa você poderá encontrar diversos papéis, um sobre o outro, sendo que freqüentemente partes de páginas embaixo da primeira estão visíveis. O equivalente do desktop no Windows é a tela. O equivalente a pedaços de papel é representado pelas *janelas* na tela. Em uma mesa, você pode mover os pedaços

de papel, trocar o papel que está em cima ou controlar o quanto dos demais papéis permanece visível. O Windows permite o mesmo tipo de operações sobre as suas janelas. Ao selecionar uma janela, você a torna *ativa*, o que significa que ela fica por cima de todas as demais janelas abertas. Você pode aumentar ou encolher uma janela, ou movê-la pela tela. Em resumo, o Windows permite que você controle a superfície da tela da maneira que você controla os itens em sua mesa.

O Mouse

Assim como as versões anteriores do Windows, o Windows 95 permite o uso do mouse para quase todas as operações de controle, seleção e desenho. De fato, dizer que ele *permite* o uso do mouse é uma afirmação errada. O fato é que a interface do Windows 95 foi *projetada para o mouse* — ela *permite* o uso do teclado! Embora seja possível a um programa aplicativo ignorar o mouse, ele estaria violando um dos princípios básicos do projeto do Windows.

Ícones e Mapas de Bits

O Windows 95 recomenda o uso de ícones e mapas de bits (imagens gráficas). A teoria por trás deste uso de ícones e mapas de bits é o velho ditado segundo o qual "uma imagem vale mais que mil palavras".

Um ícone é um pequeno símbolo que é usado para representar uma operação ou programa. Geralmente, a operação ou programa podem ser ativados selecionando o ícone. Um mapa de bits é usado freqüentemente para exibir informação de maneira rápida e simples para o usuário. No entanto, mapas de bits também podem ser usados como elementos de menu.

Menus, Barras de Ferramentas, Barras de Status e Caixas de Diálogo

À parte das janelas padrões, o Windows 95 também fornece diversas janelas de propósito especial. As mais comuns destas são o menu, a barra de ferramentas, a barra de status e a caixa de diálogo.

Um *menu* é, como seria de esperar, uma janela especial que contém somente um menu a partir do qual o usuário faz uma seleção. No entanto, em vez de ter de fornecer as suas próprias funções de seleção de itens do menu, você simplesmente cria um menu padrão usando funções de seleção de menu embutidas no Windows.

Uma *barra de ferramentas* é essencialmente um tipo especial de menu que exibe suas opções usando pequenas imagens gráficas (ícones). O usuário seleciona um objeto dando um clique na imagem desejada. Uma *barra de status* é uma barra localizada na parte inferior da janela que exibe informação relacionada com o estado da aplicação. Tanto as barras de ferramentas quanto as barras de status são inovações no Windows 95. Elas não existiam nas versões anteriores de Windows como elementos padrões.

Uma *caixa de diálogo* é uma janela especial que permite uma interação mais complexa com a aplicação do que a permitida em um menu ou barra de ferramentas. Por exemplo, sua aplicação pode usar uma caixa de diálogo para solicitar um nome de arquivo. Com poucas exceções, as entradas que não vêm do menu são efetuadas via caixas de diálogo.

Como Windows 95 e Seu Programa Interagem

Quando você escreve um programa para muitos sistemas operacionais, é seu programa que inicia a interação com o sistema operacional. Por exemplo, em um programa DOS, é o programa que solicita coisas como entradas e saídas. Dito de maneira diferente, programas escritos na "maneira tradicional" chamam o sistema operacional. O sistema operacional não chama seu programa. No entanto, o Windows 95 geralmente funciona da maneira oposta. É o Windows 95 que chama seu programa. O processo funciona assim: seu programa espera até que lhe seja enviada uma *mensagem* pelo Windows. A mensagem é passada a seu programa mediante uma função especial que é chamada pelo Windows. Uma vez que a mensagem tiver sido recebida, espera-se que seu programa tome uma ação apropriada. Embora seu programa possa chamar uma ou mais funções da API do Windows 95 enquanto responde à mensagem, não deixa de ser o Windows 95 que inicia a atividade. Mais que qualquer outra coisa, é a interação baseada em mensagens com o Windows 95 que dita a forma geral de todos os programas Windows 95.

Há muitos tipos diferentes de mensagens que o Windows 95 pode enviar a seu programa. Por exemplo, cada vez que se dá um clique com o mouse sobre uma janela pertencente a seu programa, uma mensagem que recebeu o clique será enviada para seu programa. Outro tipo de mensagem é enviado cada vez que uma janela pertencente a seu programa precisa ser redesenhada. Ainda outra mensagem é enviada cada vez que o usuário pressiona uma tecla quando seu programa é o foco da entrada. Lembre-se bem deste fato: do ponto de vista do seu programa, as mensagens chegam de forma aleatória. Isto explica por que

programas Windows 95 se assemelham a programas gerenciados por interrupções. Você não tem como saber qual será a próxima mensagem.

Um ponto final: mensagens enviadas para seu programa são armazenadas em uma *fila de mensagens* associada com seu programa. Portanto, nenhuma mensagem será perdida porque seu programa está ocupado processando outra mensagem. A mensagem simplesmente esperará na fila até que seu programa esteja pronto para ela.

Windows 95 Usa Multitarefa Preemptiva

Desde o começo, o Windows sempre foi um sistema operacional multitarefa. Isto significa que ele pode executar dois ou mais programas simultaneamente. O Windows 95 usa multitarefa preemptiva. Com este enfoque, cada programa ativo recebe uma fatia do tempo da CPU. Durante sua fatia de tempo é que cada aplicação de fato é executada. Quando a fatia de tempo da aplicação se esgota, a próxima aplicação começa a executar. (A aplicação executada anteriormente entra em um estado de suspensão, aguardando a sua próxima fatia de tempo.) Desta maneira, cada aplicação no sistema recebe uma parte do tempo da CPU. Embora o esqueleto de aplicação que desenvolveremos neste capítulo não lide com os aspectos de multitarefa do Windows 95, eles serão uma parte importante de qualquer aplicação que você criar.

NOTA: Versões mais antigas de Windows usam uma forma de multitarefa chamada de não-preemptiva. Com este enfoque, uma aplicação mantinha o controle da CPU até liberá-la explicitamente. Isto permitia que as aplicações monopolizassem a CPU e deixassem outros programas "de fora". A multitarefa preemptiva elimina este problema.

A API Win32: A API de Windows 95

Em geral, o ambiente Windows é acessado por meio de uma interface baseada em chamadas denominada *Application Program Interface* (API, ou Interface de Programas Aplicativos). A API consiste em algumas centenas de funções que seu programa chama à medida que for necessário. As funções da API provêm todos os recursos de sistema executados pelo Windows 95. Existe um subconjunto da API denominado Graphics Device Interface (GDI), que é a parte do Windows que provê suporte a gráficos independentemente dos dispositivos físicos. São as funções da GDI que tornam possível que um programa Windows execute em uma variedade de hardwares gráficos.

Os programas Windows 95 usam a API Win32. Na sua maior parte, Win32 é um superconjunto da API mais antiga do Windows 3.1 (Win16). De

fato, na sua maior parte, as funções são chamadas pelo mesmo nome e são usadas da mesma maneira. No entanto, embora similar até em espírito e propósito, as duas APIs diferem porque Win32 suporta endereçamento de 32 bits, enquanto Win16 suporta somente o modelo de memória segmentada de 16 bits. Por causa desta diferença, muitas das funções mais antigas da API tiveram de ser expandidas de forma a aceitar argumentos de 32 bits e a retornar valores de 32 bits. Além disso, algumas poucas funções tiveram de ser alteradas para suportar a arquitetura de 32 bits. Também foram adicionadas algumas funções à API para suportar o novo enfoque da multitarefa, os novos elementos da interface de usuário e os demais recursos melhorados no Windows 95. Se você é novo na programação Windows, estas mudanças não o afetarão de maneira significativa. No entanto, se você estiver portando código do Windows 3.1 para o Windows 95, então precisará examinar com cuidado cada um dos argumentos que passar a cada função da API.

Como o Windows 95 suporta o endereçamento de 32 bits, faz sentido que os inteiros também sejam de 32 bits. Isto significa que os tipos **int** e **unsigned** têm 32 bits de comprimento, em vez de 16 bits, como é o caso no Windows 3.1. Se você deseja usar um inteiro de 16 bits, ele deve ser declarado como **short**. (Como você verá em breve, o Windows 95 fornece nomes portáteis definidos usando **typedef**.) Portanto, se você estiver portando código do ambiente de 16 bits, precisará verificar o uso dos inteiros porque eles serão expandidos de 16 para 32 bits, o que pode incorrer em efeitos colaterais indesejados.

Outro efeito do endereçamento de 32 bits é que os ponteiros não precisam mais ser declarados como **near** ou **far**. Qualquer ponteiro pode acessar qualquer parte da memória. No Windows 95, tanto **far** como **near** são definidos como nada. Isto significa que você pode manter **near** e **far** em seus programas quando os carregar para o Windows 95, mas eles não terão nenhum efeito.

Os Componentes de uma Janela

Antes de passar a aspectos específicos da programação Windows 95, alguns termos importantes precisam ser definidos. A Figura 24.1 exibe uma janela padrão com cada um dos seus elementos em destaque.

Todas as janelas têm uma borda que define os limites da janela; as bordas também são usadas quando uma janela muda de tamanho. No topo da janela encontramos diversos itens. No canto esquerdo encontramos o ícone do menu de sistema (também chamado de ícone da barra de título). Dando um clique sobre esta caixa, é exibido o menu de sistema. À direita do ícone do menu do sistema encontramos o título da janela. No canto direito encontramos as caixas de minimizar, maximizar e fechar. (Versões anteriores do Windows não tinham

uma caixa de fechar. Esta é uma inovação do Windows 95.) A área cliente é parte da janela na qual ocorre a atividade específica de seu programa. A maioria das janelas também possui barras de deslocamento vertical e horizontal, usadas para mover a informação através da janela.

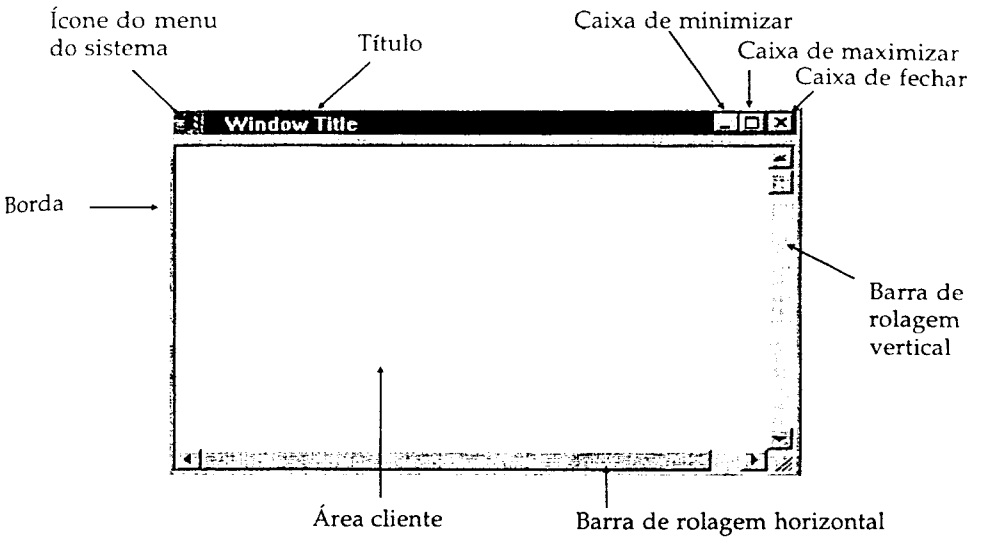


Figura 24.1 Os elementos de uma janela padrão.

Noções Básicas sobre Aplicações Windows 95

Antes de desenvolver o esqueleto de uma aplicação Windows 95, alguns conceitos básicos comuns a todos os programas Windows 95 precisam ser explicados.

WinMain()

Todos os programas Windows 95 iniciam a sua execução com uma chamada da função **WinMain()**. (Programas Windows não possuem uma função **main()**.) **WinMain()** tem algumas propriedades especiais que a diferenciam das demais funções em sua aplicação. Primeiramente, ela deve ser compilada usando a convenção de chamada **WINAPI**. (Você verá **APIENTRY** sendo usada também. Ambas significam a mesma coisa.) Normalmente, as funções de seu programa C

usam a convenção de chamada de C. No entanto, é possível compilar uma função de forma a usar uma convenção de chamada diferente; Pascal é uma alternativa comum. Por várias razões técnicas, a convenção de chamada usada pelo Windows 95 para chamar **WinMain()** é **WINAPI**. O tipo de retorno de **WinMain()** deve ser **int**.

A Função de Janela

Todos os programas Windows 95 devem conter uma função especial que *não* é chamada por seu programa, mas é chamada pelo Windows 95. Esta função é geralmente chamada de *função de janela* ou *procedimento de janela*. A função de janela é chamada pelo Windows 95 quando ele precisa passar uma mensagem a seu programa. É por meio desta função que o Windows 95 se comunica com seu programa. A função de janela recebe uma mensagem em seus parâmetros. Todas as funções de janela devem ser declaradas como retornando o tipo **LRESULT CALLBACK**. O tipo **LRESULT** é um **typedef** que, no momento em que este livro está sendo escrito, é outro nome para um inteiro longo. A convenção de chamada **CALLBACK** é utilizada com aquelas funções que serão chamadas pelo Windows 95. Na terminologia Windows, qualquer função que é chamada pelo Windows é referenciada como uma função de *callback*.

Além de receber as mensagens enviadas pelo Windows 95, a função de janela deve iniciar quaisquer ações indicadas por uma mensagem. Tipicamente, o corpo de uma função de janela consiste em um comando **switch** que liga uma resposta específica com cada mensagem para a qual o programa possui uma resposta. Seu programa não precisa responder a cada mensagem enviada pelo Windows 95. As mensagens de que seu programa não trata, você pode deixar que sejam tratadas pelo próprio Windows 95. Como existem centenas de mensagens que o Windows 95 pode gerar, é comum que a maioria das mensagens seja tratada pelo Windows 95, e não pelo seu programa.

Todas as mensagens são números inteiros de 32 bits. Além disso, todas as mensagens estão ligadas com qualquer informação adicional necessária para a mensagem.

Classes de Janelas

Quando seu programa Windows 95 inicia a sua execução, precisará definir e registrar uma *classe de janela*. Ao registrar uma classe de janela, você está dizendo ao Windows 95 qual a forma e função da sua janela. No entanto, o ato de registrar uma classe de janela não cria nenhuma janela. Para criar realmente uma janela são necessários passos adicionais.

A Repetição de Mensagens

Como explicado antes, o Windows 95 se comunica com seu programa enviando-lhe mensagens. Todas as aplicações Windows 95 devem criar uma *repetição de mensagens* dentro de **WinMain()**. Esta repetição lê qualquer mensagem pendente da fila de mensagens e a despacha para que o Windows 95 a envie, como parâmetro, à função de janela. Este pode parecer um método muito complexo de passar mensagens, mas é, mesmo assim, a maneira pela qual todos os programas Windows devem funcionar. (Parte do motivo para este esquema é o de retornar o controle ao Windows 95 para que o scheduler possa alocar a CPU como convier, em vez de aguardar o fim da fatia de tempo da sua aplicação.)

Os Tipos de Dados Windows

Como você verá em breve, os programas Windows 95 não fazem uso intensivo dos tipos de dados padrão de C, tais como **int** ou **char ***. Em vez disso, todos os tipos de dados usados pelo Windows 95 foram convertidos em **typedef** dentro do arquivo WINDOWS.H e/ou outros arquivos relacionados. O arquivo WINDOWS.H é fornecido pelo seu compilador compatível com Windows e deve ser incluído em todos os programas Windows 95. Alguns dos tipos mais comuns são **HANDLE, HWND, BYTE, WORD, DWORD, UINT, LONG, BOOL, LPSTR** e **LPCSTR**. **HANDLE** é um inteiro de 32 bits usado como handle. Como você verá, existe uma variedade de tipos de handles, mas todos são do mesmo tamanho que o tipo **HANDLE**. Um *handle* é simplesmente um valor que identifica um recurso. Ainda, todos os tipos de handle começam com **H**. Por exemplo, **HWND** é um inteiro de 32 bits usado como handle de janela. **BYTE** é um inteiro de 8 bits sem sinal. **WORD** é um inteiro sem sinal de 16 bits. **DWORD** é um inteiro longo sem sinal. **UINT** é um inteiro sem sinal de 32 bits. **LONG** é outro nome para **long**. **BOOL** é um inteiro. Este tipo é usado para indicar valores que são ou verdadeiros ou falsos. **LPSTR** é um ponteiro para uma string e **LPCSTR** é um ponteiro **const** para uma string.

Além dos tipos básicos descritos, o Windows 95 define uma variedade de estruturas. As duas que são necessárias no programa esqueleto são **MSG** e **WNDCLASS**. A estrutura **MSG** contém uma mensagem do Windows 95, enquanto **WNDCLASS** é uma estrutura que define uma classe de janela. Estas estruturas serão discutidas mais adiante neste capítulo.

Um Esqueleto Windows 95

Agora que abordamos a informação básica necessária, está na hora de desenvolver uma aplicação Windows 95 mínima. Como citado, todos os programas Windows 95 possuem algumas coisas em comum. Esta seção desenvolve um esqueleto Windows 95 que fornece estes recursos necessários. No mundo da programação Windows, esqueletos de programa são usados com freqüência porque existe uma "tarifa de admissão" elevada ao criar um programa Windows. Ao contrário de programas DOS que você possa ter escrito, nos quais um programa mínimo tem umas 5 linhas, um programa mínimo Windows tem aproximadamente 50 linhas de comprimento.

Um programa Windows 95 mínimo contém duas funções: **WinMain()** e a função de janela. A função **WinMain()** deve executar os seguintes passos genéricos:

1. Definir uma classe de janela.
2. Registrar essa classe no Windows 95.
3. Criar uma janela dessa classe.
4. Exibir a janela.
5. Iniciar a execução da repetição de mensagens.

A janela de função deve responder a todas as mensagens relevantes. Como o programa esqueleto não faz nada além de exibir a janela, a única mensagem que precisa responder é aquela que indica ao programa que o usuário o está encerrando.

Antes de entrar em detalhes específicos, examine o programa seguinte, que é um esqueleto mínimo Windows 95. Ele cria uma janela padrão contendo um título. A janela também contém o menu de sistema e, portanto, é capaz de ser minimizada, maximizada, movida, redimensionada e fechada. Ela também contém as caixas padrões de minimizar, maximizar e fechar.

```
/* Um esqueleto mínimo Windows 95. */

#include <windows.h>

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

char szWinName[] = "MinhaJan"; /* Nome da classe de janela */

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR lpszArgs, int nWinMode)
{
```



```
HWND hwnd;
MSG msg;
WNDCLASS wcl;

/* Define uma classe de janela. */
wcl.hInstance = hThisInst; /* handle desta instância */
wcl.lpszClassName = szWinName; /* nome da classe de janela */
wcl.lpfnWndProc = WindowFunc; /* função de janela */
wcl.style = 0; /* estilo padrão */

wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* estilo de
                                                ícone */
wcl.hCursor = LoadCursor(NULL, IDC_ARROW); /* estilo de
                                              cursor */
wcl.lpszMenuName = NULL; /* sem menu */

wcl.cbClsExtra = 0; /* nenhuma informação */
wcl.cbWndExtra = 0; /* extra é necessária */

/* Faz o fundo da janela ser branco. */
wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

/* Registra a classe de janela. */
if (!RegisterClass(&wcl)) return 0;

/* Agora que uma classe de janela foi registrada,
   pode ser criada uma janela. */
hwnd = CreateWindow(
  szWinName, /* nome da classe da janela */
  "Esqueleto Windows 95", /* título */
  WS_OVERLAPPEDWINDOW, /* estilo da janela - normal */
  CW_USEDEFAULT, /* coordenada X - deixe Windows decidir */
  CW_USEDEFAULT, /* coordenada Y - deixe Windows decidir */
  CW_USEDEFAULT, /* largura - deixe Windows decidir */
  CW_USEDEFAULT, /* altura - deixe Windows decidir */
  HWND_DESKTOP, /* nenhuma janela-pai */
  NULL, /* sem menu */
  hThisInst, /* handle desta instância do programa */
  NULL /* nenhum argumento adicional */
);

/* Exibe a janela. */
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);
```

```
  /* Cria a repetição de mensagens. */
  while (GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg); /* permite uso do teclado */
    DispatchMessage(&msg); /* retorna o controle ao Windows */
  }
  return msg.wParam;
}

/* Esta função é chamada pelo Windows 95 e
   recebe mensagens da fila de mensagens.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    case WM_DESTROY: /* encerra o programa */
      PostQuitMessage(0);
      break;
    default:
      /* Deixe o Windows 95 processar quaisquer mensagens
         não especificadas no comando switch acima. */
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}
```

A janela produzida por este programa é exibida na Figura 24.2. Agora vamos analisar este programa passo a passo.

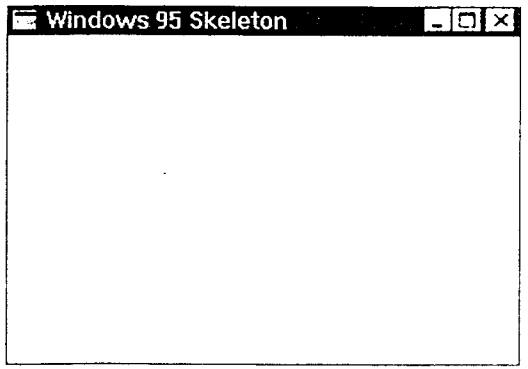


Figura 24.2 A janela produzida pelo esqueleto Windows 95.

Primeiro, todos os programas Windows 95 devem incluir o arquivo de cabeçalho WINDOWS.H. Como já citado, este arquivo (junto com seus arquivos de suporte) contém os protótipos das funções da API e diversos tipos, macros e definições usados pelo Windows 95. Por exemplo, os tipos de dados **HWND** e **WNDCLASS** são definidos em WINDOWS.H.

A função de janela usada pelo programa é chamada **WindowFunc()**. Ela é declarada como uma função de callback, porque é esta a função chamada por Windows 95 para se comunicar com o programa.

A execução do programa inicia com **WinMain()**, a qual recebe quatro parâmetros. **hThisInst** e **hPrevInst** são handles. **hThisInst** se refere a instância atual do programa. Lembre-se, o Windows 95 é um sistema operacional multitarefa, de forma que mais de uma instância de seu programa podem estar sendo executadas ao mesmo tempo. **hPrevInst** sempre é **NULL**. (Em programas Windows 3.1, **hPrevInst** era não zero se havia outras instâncias do programa corrente sendo executadas, mas isto não se aplica no Windows 95.) O parâmetro **lpszArgs** é um ponteiro para uma string que contém quaisquer argumentos de linha de comando especificados quando a aplicação foi iniciada. O parâmetro **nWinMode** contém um valor que determina como a janela será exibida quando seu programa inicia a execução.

Dentro da função, são criadas três variáveis. A variável **hwnd** conterá o handle da janela do programa. A variável do tipo estrutura **msg** conterá mensagens de janela, enquanto a variável do tipo estrutura **wcl** será usada para definir a classe de janela.

Definindo a Classe de Janela

As duas primeiras ações tomadas por **WinMain()** são a de definir uma classe de janela e depois registrá-la. Uma classe de janela é definida preenchendo os campos definidos pela estrutura **WNDCLASS**. Seus campos são mostrados aqui:

```
UINT style; /* tipo da janela */
WNDPROC lpfnWndProc; /* endereço da função de janela */
int cbClsExtra; /* informação extra da classe */
int cbWndExtra; /* informação extra da janela */
HINSTANCE hInstance; /* handle desta instância */
HICON hIcon; /* handle do ícone minimizado */
HCURSOR hCursor; /* handle do cursor do mouse */
HBRUSH hbrBackground; /* cor de fundo */
LPCSTR lpszMenuName; /* nome do menu principal */
LPCSTR lpszClassName; /* nome da classe de janela */
```

Como você pode ver neste programa, o campo **hInstance** recebe o handle da instância atual como especificado por **hThisInst**. O nome da classe de janela é apontado por **lpszClassName**, que aponta para a string "MinhaJan", neste caso. O endereço da função de janela é atribuído a **lpfnWndProc**. Nenhum estilo padrão é especificado, e nenhuma informação adicional é necessária.

Todas as aplicações Windows precisam definir uma forma padrão para o cursor do mouse e para o ícone da aplicação. Uma aplicação pode definir suas próprias versões especializadas destes recursos, ou pode usar um dos estilos embutidos, como o esqueleto faz. O estilo do ícone é carregado pela função **LoadIcon()** da API, cujo protótipo é mostrado aqui:

HICON LoadIcon(HINSTANCE *hInst*, LPCSTR *lpszName*);

Esta função retorna um handle para um ícone. Aqui, *hInst* especifica o handle do módulo que contém o ícone, e o nome do ícone é especificado em *lpszName*. No entanto, para usar um dos ícones embutidos, você deve passar **NULL** como o primeiro parâmetro e especificar uma das seguintes macros para o segundo.

Macro de ícone	**Forma**
IDI_APPLICATION	Ícone default
IDI_ASTERISK	Ícone de informação
IDI_EXCLAMATION	Ícone de ponto de exclamação
IDI_HAND	Símbolo de "Pare"
IDI_QUESTION	Ícone de ponto de interrogação

Para carregar o cursor do mouse, use a função **LoadCursor()** da API. Esta função tem o seguinte protótipo:

HCURSOR LoadCursor(HINSTANCE *hInst*, LPCSTR *lpszName*);

Esta função retorna um handle para um recurso de cursor. Aqui, *hInst especifica* o handle do módulo que contém o cursor do mouse, e o nome do cursor do mouse é especificado em *lpszName*. No entanto, para usar um dos ícones embutidos, você deve passar **NULL** como o primeiro parâmetro e especificar um dos cursores embutidos, usando sua macro, como o segundo parâmetro. Alguns dos cursores embutidos mais comuns são exibidos aqui.

Macro de cursor	**Forma**
IDC_ARROW	Ponteiro de seta padrão
IDC_CROSS	Cruz
IDC_IBEAM	I vertical
IDC_WAIT	Ampulheta

A cor de fundo da janela criada pelo esqueleto é especificada como branco, e um handle para este *pincel* é obtido usando a função **GetStockObject()** da API. Um pincel é um recurso que pinta a tela usando uma determinada cor, tamanho e padrão. A função **GetStockObject()** é usada para obter um handle

para uma quantidade de objetos padrões de exibição, incluindo pincéis, canetas (que desenham linhas) e fontes de caracteres. Ela tem este protótipo:

HGDIOBJ GetStockObject(int *object*);

A função retorna um handle para o objeto especificado em *object*. (O tipo **HGDIOBJ** é um handle da GDI.) Aqui estão alguns dos pincéis embutidos disponíveis para seu programa:

Macro de pincel	**Tipo de fundo**
BLACK_BRUSH	Preto
DKGRAY_BRUSH	Cinza-escuro
HOLLOW_BRUSH	Janela transparente
LTGRAY_BRUSH	Cinza-claro
WHITE_BRUSH	Branco

Você pode usar essas macros como parâmetros para **GetStockObject()** para obter um pincel.

Uma vez que a classe de janela tenha sido especificada por completo, ela é registrada junto ao Windows 95 usando a função **RegisterClass()** da API, cujo protótipo é mostrado aqui:

ATOM RegisterClass(CONST WNDCLASS **lpWClass*);

A função retorna um valor que identifica a classe de janela. **ATOM** é um **typedef** que significa **WORD**. Cada classe de janela recebe um valor único. *lpWClass* tem de ser o endereço da estrutura **WNDCLASS**.

Criando uma Janela

Uma vez que a janela foi definida e registrada, sua aplicação pode realmente criar uma janela dessa classe usando a função **CreateWindow()** da API, cujo protótipo é mostrado aqui.

```
HWND CreateWindow(
  LPCSTR lpClassName, /* nome da classe de janela */
  LPCSTR  lpWinName, /* título da janela */
  DWORD dwStyle, /* tipo da janela */
  int X, int Y, /* coordenadas do canto superior esquerdo */
  int Width, int Height, /* tamanho da janela */
  HWND hParent, /* handle da janela-mãe */
  HMENU hMenu, /* handle do menu principal */
  HINSTANCE hThisInst, /* handle da instância criadora */
  LPVOID lpszAdditional /* ponteiro para informação adicional */
);
```

Como você pode ver no programa esqueleto, muitos dos parâmetros de **CreateWindow()** podem usar valores padrões ou ser simplesmente **NULL**. De fato, a maioria das vezes os parâmetros *X*, *Y*, *Width* e *Height* simplesmente usarão a macro **CW_USEDEFAULT**, que indica ao Windows 95 para selecionar um tamanho e posição apropriados para a janela. Se a janela não possui mãe, que é o caso do esqueleto, então *hParent* deve ser especificado como **HWND_DESKTOP**. (Você também pode usar **NULL** para este parâmetro.) Se a janela não contém um menu principal, então *hMenu* deve ser **NULL**. Além disso, se nenhuma informação adicional for necessária, como é o caso com freqüência, então *lpszAdditional* deverá ser **NULL**. (O tipo **LPVOID** é tornado **typedef** como **void** *. Historicamente, **LPVOID** indica "ponteiro longo para **void**".)

Os quatro parâmetros restantes têm de ser definidos explicitamente por seu programa. Primeiro, *lpszClassName* deve apontar para o nome da classe de janela. (Este é o nome que você lhe deu quando ela foi registrada.) O título da janela é uma string apontada por *lpszWinName*. Isto pode ser uma string nula, mas usualmente a janela receberá um título. O estilo (ou tipo) da janela realmente criada é determinado pelo valor de *dwStyle*. A macro **WS_OVERLAPPEDWINDOW** especifica uma janela padrão que tem um menu de sistema, uma borda e caixas de minimizar, maximizar e fechar. Embora este estilo de janela seja o mais comum, você pode construir um de acordo com as suas especificações. Para tanto, você simplesmente junta com OR as várias macros de estilo que quer. Alguns outros estilos comuns são mostrados aqui:

Macros de estilo	Recurso na janela
WS_OVERLAPPED	Janela sobreposta com borda
WS_MAXIMIZEBOX	Caixa de maximizar
WS_MINIMIZEBOX	Caixa de minimizar
WS_SYSMENU	Menu de sistema
WS_HSCROLL	Barra de rolagem horizontal
WS_VSCROLL	Barra de rolagem vertical

O parâmetro *hThisInst* deve conter o handle da instância corrente da aplicação.

A função **CreateWindow()** retorna o handle da janela que ela cria, ou **NULL** se a janela não pode ser criada.

Uma vez que a janela tiver sido criada, ela ainda não é exibida na tela. Para fazer com que a janela seja exibida, chame a função **ShowWindow()** da API. Esta função tem o seguinte protótipo:

BOOL ShowWindow(HWND *hwnd*, int *nHow*);

O handle da janela a ser exibida deve ser especificado em *hwnd*. O modo de exibição é especificado em *nHow*. A primeira vez que a janela for exibida, você quererá passar o parâmetro **nWinMode** de **WinMain()** como o parâmetro *nHow*.

Lembre-se, o valor de **nWinMode** determina como a janela será exibida quando o programa inicia sua execução. Chamadas subseqüentes podem exibir (ou remover) a janela conforme necessário. Alguns valores comuns para *nHow* são mostrados aqui:

Macros de exibição	Efeito
SW_HIDE	Remove a janela
SW_MINIMIZE	Minimiza a janela para um ícone
SW_MAXIMIZE	Maximiza a janela
SW_RESTORE	Retorna uma janela para seu tamanho normal

A função **ShowWindow()** retorna o estado anterior de exibição da janela. Se a janela estava sendo exibida, esse valor é diferente de zero. Se a janela não estava sendo exibida, esse valor é zero.

Embora não seja tecnicamente necessária no esqueleto, a chamada da função **UpdateWindow()** é incluída porque ela é necessária para virtualmente qualquer aplicação Windows 95 que você crie. Ela essencialmente indica ao Windows 95 para enviar uma mensagem para sua aplicação indicando que a janela principal precisa ser atualizada.

A Repetição de Mensagens

A parte final da **WinMain()** do esqueleto é a *repetição de mensagens*. A repetição de mensagens é uma parte de todas as aplicações Windows. Seu objetivo é o de receber e processar mensagens enviadas pelo Windows 95. Quando uma aplicação está executando, são enviadas mensagens para ela de forma quase contínua. Essas mensagens são armazenadas na fila de mensagens da aplicação até que possam ser lidas e processadas. Cada vez que seu programa estiver pronto para ler outra mensagem, ele deve chamar a função **GetMessage()** da API, que tem o protótipo:

BOOL GetMessage(LPMSG *msg*, HWND *hwnd*, UINT *min*, UNIT *max*);

A mensagem será recebida pela estrutura apontada por *msg*. Todas as mensagens do Windows são do tipo de estrutura **MSG**, exibido aqui.

```
/* Estrutura de mensagem */
typedef struct tagMSG
{
  HWND hwnd; /* janela à qual se destina a mensagem */
  UINT message; /* mensagem */
  WPARAM wParam; /* informação dependente da mensagem */
  LPARAM lParam; /* mais informação dependente da mensagem */
  DWORD time; /* momento em que a mensagem foi enfileirada */
  POINT pt; /* coordenadas X,Y do mouse */
} MSG;
```

Em **MSG**, o handle da janela à qual se destina a mensagem está contido em **hwnd**. Todas as mensagens Win32 são inteiros de 32 bits, e a mensagem é contida em **message**. Informação adicional referente a cada mensagem é passada em **wParam** e **lParam**. O tipo **WPARAM** é um **typedef** para **UINT**, enquanto **LPARAM** é um **typedef** para **LONG**.

O momento em que a mensagem foi enviada é especificado em milissegundos no campo **time**.

O membro **pt** contém as coordenadas do mouse no momento em que a mensagem foi enviada. As coordenadas são mantidas em uma estrutura **POINT**, definida como segue:

```
typedef struct tagPOINT {
  LONG x, y;
} POINT;
```

Se não houver mensagens na fila de mensagens da aplicação, então uma chamada a **GetMessage()** passará o controle de volta ao Windows 95.

O parâmetro *hwnd* de **GetMessage()** especifica a janela para a qual as mensagens serão obtidas. É possível e até provável que uma aplicação contenha diversas janelas, mas você só irá querer receber mensagens para uma janela específica. Se você deseja receber todas as mensagens destinadas a sua aplicação, este parâmetro deve ser **NULL**.

Os dois parâmetros restantes de **GetMessage()** especificam uma faixa de mensagens que será recebida. Geralmente, você deseja que sua aplicação receba todas as mensagens. Para conseguir isto, especifique tanto *min* como *max* como 0, como o esqueleto faz.

GetMessage() retorna zero quando o usuário encerra o programa, fazendo com que a repetição de mensagens termine. Em qualquer outro caso, ela retorna um valor diferente de zero.

Dentro da repetição de mensagens, são chamadas duas funções. A primeira é a função **TranslateMessage()** da API. Esta função traduz códigos de teclas virtuais gerados pelo Windows 95 em mensagens de caracteres. (Teclas virtuais incluem as teclas de função, as teclas de direção etc.) Embora não seja necessário para todas as aplicações, a maioria das aplicações chama **TranslateMessage()** porque ela é necessária para permitir a total integração do teclado com seu programa aplicativo.

Uma vez que a mensagem foi lida e traduzida, ela é enviada de volta para o Windows 95 usando a função **DispatchMessage()** da API. O Windows 95 então mantém a mensagem até que ela possa ser passada à função de janela do programa.

Uma vez que a repetição de mensagens termina, a função **WinMain()** termina retornando o valor de **msg.wParam** para Windows 95. Este valor contém o código de retorno gerado quando seu programa termina.

A Função de Janela

A segunda função no esqueleto de aplicação é a sua função de janela. Neste caso a função é chamada **WindowFunc()**, mas ela poderia ter qualquer nome de que você gostasse. A função de janela recebe os primeiros quatro membros da estrutura **MSG** como parâmetros. Para o esqueleto, o único parâmetro usado é a mensagem propriamente dita. No entanto, aplicações reais usarão os outros parâmetros desta função.

A função de janela do esqueleto responde a uma única mensagem explicitamente: **WM_DESTROY**. Esta mensagem é enviada quando o usuário encerra o programa. Quando essa mensagem é recebida, seu programa deve executar uma chamada à função **PostQuitMessage()** da API. O argumento desta função é o código de encerramento que é retornado em **msg.wParam** dentro de **WinMain()**. Chamar **PostQuitMessage()** faz com que uma mensagem **WM_QUIT** seja enviada para sua aplicação, fazendo com que **GetMessage()** retorne falso, e portanto parando o programa.

Quaisquer outras mensagens recebidas por **WindowFunc()** são passadas para o Windows 95 por meio de uma chamada a **DefWindowProc()** para receberem o tratamento padrão. Este passo é necessário porque todas as mensagens devem ser tratadas de alguma forma.

Usando um Arquivo de Definição

Se você estiver familiarizado com a programação Windows 3.1, então já usou *arquivos de definição*. No Windows 3.1, todos os programas precisam de um arquivo de definição associado a eles. Um arquivo de definição é simplesmente um arquivo-texto que especifica certas informações e opções necessárias para seu programa Window 3.1. No entanto, por causa da arquitetura de 32 bits de Windows 95 (e outras melhorias), os arquivos de definição não são mais necessários.

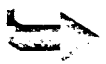
Mesmo assim, não faz mal nenhum fornecer um arquivo de definição, e se você deseja fornecer um para manter a compatibilidade com o Windows 3.1, é livre para fazê-lo.

Se você é novo na programação Windows em geral e não sabe o que é um arquivo de definição, a discussão seguinte dá um breve resumo.

Todos os arquivos de definição usam a extensão .DEF. Por exemplo, o arquivo de definição para o programa esqueleto poderia ser chamado SKEL.DEF. Eis um arquivo de definição que você poderia fornecer para manter a compatibilidade com Windows 3.1:

```
DESCRIPTION 'Programa Esqueleto'
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
HEAPSIZE 8192
STACKSIZE 8192
EXPORTS WindowFunc
```

Este arquivo especifica o nome do programa e sua descrição, ambos opcionais. Ele também determina que o arquivo executável será compatível com o Windows (em vez do DOS, por exemplo). O comando **CODE** indica ao Windows 95 para carregar todo o programa no início (**PRELOAD**), que o código pode ser movimentado na memória (**MOVEABLE**) e que o código pode ser eliminado da memória e recarregado se (e quando) necessário (**DISCARDABLE**). O arquivo especifica que os dados do seu programa devem ser carregados no início da execução e podem ser movidos na memória. Ele também determina que cada instância do programa terá seus próprios dados (**MULTIPLE**). A seguir, são especificados o tamanho do heap e da pilha do programa. Finalmente, o nome da função de janela é exportado. A exportação permite que o Windows 3.1 chame a função.



NOTA: Arquivos de definição não são necessários ao programar para Windows 95. No entanto, eles não causam problemas e podem ser incluídos para manter a compatibilidade com o Windows 3.1. (Pode ser necessário aumentar o tamanho do heap e da pilha para aplicações reais.)

Convenções sobre Nomes

Antes de concluir este capítulo, é necessário fazer um breve comentário sobre os nomes das funções e variáveis. Se você é novo na programação Windows, di-

versos dos nomes de variáveis e parâmetros no programa esqueleto e sua descrição devem ter parecido estranhos. Isto se deve ao fato de que seguem uma convenção de nomes inventada pela Microsoft para a programação Windows. Para funções, o nome consiste em um verbo seguido de um substantivo. A primeira letra do verbo e do substantivo são indicadas em maiúsculas.

Para nomes de variáveis, a Microsoft escolheu um sistema bastante complexo de incluir o tipo de dados no nome. Para conseguir isto, um prefixo em letras minúsculas é adicionado no início do nome da variável. O nome propriamente dito começa com uma letra maiúscula. Os prefixos de tipo são exibidos na Tabela 24.1. Francamente, o uso de prefixos de tipo é controvertido e não é suportado universalmente. Muitos programadores Windows usam este método, mas muitos também não o usam. Você é livre para usar as convenções sobre nomes que desejar.

Tabela 24.1 Caracteres para prefixo de tipo nas variáveis.

Prefixo	Tipo de dados
b	Boolean (um byte)
c	Caractere (um byte)
dw	Inteiro longo sem sinal
f	Flags (campo de 16 bits)
fn	Função
h	Handle
l	Inteiro longo
lp	Ponteiro longo
n	Inteiro curto
p	Ponteiro
pt	Inteiro longo contendo coordenadas de tela
w	Inteiro curto sem sinal
sz	Ponteiro para string terminada em zero
lpsz	Ponteiro longo para string terminada em zero
rgb	Inteiro longo contendo valores de cor RGB