21

Estruturas de dados

Muito do que eu prendi, não pude libertar; muito do que libertei voltou para mim.
Lee Wilson Dodd

'Não dá pra ir mais rápido?', disse a enchova para o caracol. 'Há um delfim atrás de mim, e ele está me empurrando.'
Lewis Carroll

Há sempre espaço no ponto mais alto.
Daniel Webster

Prossiga — continue andando.
Thomas Morton

Virarei uma nova folha.
Miguel de Cervantes

OBJETIVOS

Neste capítulo, você aprenderá:

- A formar estruturas de dados encadeadas utilizando ponteiros, classes auto-referenciais e recursão.
- Como criar e manipular estruturas de dados dinâmicas, como listas vinculadas, filas, pilhas e árvores binárias.
- A utilizar árvores de pesquisa binária para pesquisa e classificação de alta velocidade.
- Como são várias aplicações importantes de estruturas de dados vinculadas.
- Como criar estruturas de dados reutilizáveis com templates de classe, herança e composição.

Sumário

21.1 Introdução

Estudamos as **estruturas de dados** de tamanho fixo como arrays unidimensionais, arrays bidimensionais e `structs`. Este capítulo introduz as **estruturas de dados dinâmicas** que crescem e encolhem durante a execução. As **listas vinculadas** são coleções de itens de dados 'vinculados em uma cadeia' — as inserções e as exclusões são feitas em qualquer lugar de uma lista vinculada. As **pilhas** são importantes em compiladores e sistemas operacionais: as inserções e exclusões são feitas somente em uma extremidade de uma pilha — sua **parte superior**. **Filas** (***queues***) representam seqüências de espera; as inserções são feitas na parte de trás (também referida como **cauda**) de uma fila e as remoções são feitas a partir da parte da frente (também referida como **cabeça**) de uma fila. As **árvores binárias** facilitam a pesquisa e a classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistema de arquivos e a compilação de expressões em linguagem de máquina. Essas estruturas de dados têm muitas outras aplicações interessantes.

Discutimos várias estruturas de dados populares e importantes e implementamos programas que as criam e as manipulam. Utilizamos classes, templates de classe, herança e composição para criar e empacotar essas estruturas de dados para facilitar o reúso e a manutenção.

Estudar este capítulo é uma sólida preparação para o Capítulo 23, "Standard Template Library (STL)". A STL é uma parte importante da Standard Library C++. A STL fornece contêineres, iteradores para percorrer esses contêineres e algoritmos para processar os elementos desses contêineres. Você verá que a STL empacotou cada uma das estruturas de dados que discutimos neste capítulo em classes de template. O código STL é cuidadosamente escrito para ser portável, eficiente e extensível. Uma vez que entender os princípios e a construção de estruturas de dados da maneira apresentada neste capítulo, você será capaz de fazer o melhor uso de estruturas de dados pré-empacotadas, iteradores e algoritmos na STL, um conjunto de primeira classe de componentes reutilizáveis.

Os exemplos do capítulo são programas práticos que você será capaz de utilizar em cursos mais avançados e em aplicações da indústria. Os programas empregam extensa manipulação de ponteiros. Os exercícios incluem uma rica coleção de aplicativos úteis.

Encorajamos você a tentar o projeto principal descrito na seção especial "Construindo seu próprio compilador". Você vem utilizando um compilador C++ para converter programas em linguagem de máquina a fim de poder executar esses programas no computador. Nesse projeto, você realmente construirá seu próprio compilador. Ele lerá um arquivo de instruções escrito em uma linguagem simples, mas poderosa e de alto nível, semelhante às primeiras versões da popular linguagem BASIC. Seu compilador traduzirá essas instruções em um arquivo de instruções de Simpletron Machine Language (SML) — SML é a linguagem que você aprendeu na seção especial do Capítulo 8, "Construindo seu próprio computador". Seu programa Simpletron Simulator então executará o programa SML produzido por seu compilador! A implementação desse projeto utilizando uma abordagem orientada a objetos lhe fornecerá uma maravilhosa oportunidade de praticar grande parte do que você aprendeu neste livro. A seção especial orienta você cuidadosamente nas especificações da linguagem de alto nível e descreve os algoritmos que você precisará para converter cada tipo de instrução de linguagem de alto nível em instruções de linguagem de máquina. Se gosta de desafios, você pode tentar os muitos aprimoramentos no compilador e no Simpletron Simulator sugeridos nos exercícios deste capítulo.

21.2 Classes auto-referenciais

Uma **classe auto-referencial** contém um membro ponteiro que aponta para um objeto de classe do mesmo tipo de classe. Por exemplo, a definição

```
class Node
{
public:
   Node( int ); // construtor
   void setData( int ); // configura membro de dados
   int getData() const; // obtém membro de dados
   void setNextPtr( Node * ); // configura ponteiro como próximo Node
```

```
   Node *getNextPtr() const; // obtém ponteiro para próximo Node
private:
   int data; // dados armazenados neste Node
   Node *nextPtr; // ponteiro para outro objeto do mesmo tipo
}; // fim da classe Node
```

define um tipo, `Node`. O tipo `Node` tem dois membros de dados `private` — o membro do tipo inteiro `data` e membro ponteiro `nextPtr`. O membro `nextPtr` aponta para um objeto do tipo `Node` — um objeto do mesmo tipo que aquele sendo declarado aqui, daí o termo 'classe auto-referencial'. O membro `nextPtr` é referido como um **vínculo** (ou ***link***) — isto é, `nextPtr` pode 'vincular' um objeto de tipo `Node` a outro objeto do mesmo tipo. O tipo `Node` também tem cinco funções-membro — um construtor que recebe um inteiro para inicializar o membro `data`, uma função `setData` para configurar o valor do membro `data`, uma função `getData` para retornar o valor do membro `data`, uma função `setNextPtr` para configurar o valor do membro `nextPtr` e uma função `getNextPtr` para retornar o valor do membro `nextPtr`.

Os objetos de classe auto-referencial podem ser vinculados para formar estruturas de dados úteis como listas, filas, pilhas e árvores. A Figura 21.1 ilustra dois objetos de classe auto-referencial vinculados entre si para formar uma lista. Observe que uma barra — representando um ponteiro nulo (`0`) — é colocada no membro de vínculo do segundo objeto de classe auto-referencial para indicar que o vínculo não aponta para outro objeto. A barra serve apenas para propósitos de ilustração; ela não corresponde ao caractere de barra invertida em C++. Um ponteiro nulo normalmente indica o fim de uma estrutura de dados, assim como o caractere nulo (`'\0'`) indica o fim de uma string.

Erro comum de programação 21.1

Não configurar o vínculo no último nó de uma estrutura de dados vinculada como nulo (`0`) é um erro de lógica (possivelmente fatal).

21.3 Alocação de memória e estruturas de dados dinâmicas

A criação e a manutenção de estruturas de dados dinâmicas requerem alocação dinâmica de memória, o que permite a um programa obter mais memória em tempo de execução para armazenar novos nós. Quando o programa não precisa mais dessa memória, esta pode ser liberada a fim de poder ser reutilizada para alocar outros objetos no futuro. O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador ou a quantidade de memória virtual disponível em um sistema de memória virtual. Freqüentemente, os limites são muito menores, porque a memória disponível deve ser compartilhada entre muitos programas.

O operador `new` aceita como um argumento o tipo do objeto dinamicamente sendo alocado e retorna um ponteiro para um objeto desse tipo. Por exemplo, a instrução

```
Node *newPtr = new Node( 10 ); // cria Node com o valor de 10
```

aloca os bytes `sizeof( Node )`, executa o construtor `Node` e atribui o endereço do novo objeto `Node` a `newPtr`. Se nenhuma memória estiver disponível, `new` lança uma exceção `bad_alloc`. O valor `10` é passado para o construtor `Node`, que inicializa o membro `data` do `Node` como 10.

O operador `delete` executa o destrutor `Node` e desaloca a memória alocada com `new` — a memória é retornada ao sistema para que ela possa ser futuramente realocada. Para liberar memória dinamicamente alocada pelo `new` anterior, utilize a instrução

```
delete newPtr;
```

Observe que `newPtr` em si não é excluído; em vez disso, é o espaço para o qual `newPtr` aponta que é excluído. Se ponteiro `newPtr` tem o valor de ponteiro nulo `0`, a instrução precedente não tem nenhum efeito. Não é um erro excluir (`delete`) um ponteiro nulo.

As seções a seguir discutem listas, pilhas, filas e árvores. As estruturas de dados apresentadas neste capítulo são criadas e mantidas com alocação dinâmica de memória, classes auto-referenciais, templates de classe e templates de função.

21.4 Listas vinculadas

Uma lista vinculada é uma coleção linear de objetos auto-referenciais de classe, chamados **nós**, conectados por **vínculos de ponteiro** — daí o termo lista 'vinculada'. Uma lista vinculada é acessada via ponteiro ao primeiro nó da lista. Cada nó subseqüente é acessado via

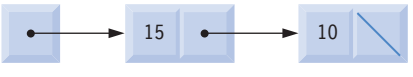

Figura 21.1 Dois objetos de classe auto-referencial vinculados entre si.

o membro ponteiro de vínculo armazenado no nó anterior. Por convenção, o ponteiro de vínculo no último nó de uma lista é configurado como nulo (0) para marcar o final da lista. Os dados são armazenados em uma lista vinculada dinamicamente — cada nó é criado conforme necessário. Um nó pode conter dados de qualquer tipo, incluindo objetos de outras classes. Se os nós contiverem ponteiros de classe básica para objetos de classe básica e classe derivada relacionados por herança, podemos ter uma lista vinculada desses nós e utilizar chamadas de função `virtual` para processar esses objetos polimorficamente. Pilhas e filas também são **estruturas de dados lineares** e, como veremos, podem ser visualizadas como versões restritas de listas vinculadas. As árvores são **estruturas de dados não-lineares**.

As listas de dados podem ser armazenadas em arrays, mas as listas vinculadas fornecem várias vantagens. Uma lista vinculada é apropriada quando o número de elementos de dados a serem representados em qualquer dado momento é imprevisível. As listas vinculadas são dinâmicas, portanto o comprimento de uma lista pode aumentar ou diminuir conforme necessário. O tamanho de um array C++ 'convencional', porém, não pode ser alterado, porque o tamanho do array é fixado em tempo de compilação. Os arrays 'convencionais' podem tornar-se cheios. As listas vinculadas tornam-se cheias apenas quando o sistema tem memória insuficiente para satisfazer solicitações de alocação de armazenamento dinâmico.

Dica de desempenho 21.1

Um array pode ser declarado para conter mais elementos do que o número de itens esperado, mas isso pode desperdiçar memória. As listas vinculadas podem fornecer melhor utilização de memória nessas situações. As listas vinculadas permitem ao programa se adaptar em tempo de execução. Observe que o template da classe `vector` (introduzido na Seção 7.11) implementa uma estrutura de dados baseada em array dinamicamente redimensionável.

As listas vinculadas podem ser mantidas em ordem classificada inserindo cada novo elemento no ponto adequado na lista. Os elementos existentes da lista não precisam ser movidos.

Dica de desempenho 21.2

A inserção e a exclusão em um array classificado podem consumir muito tempo — todos os elementos que se seguem ao elemento inserido ou excluído devem ser deslocados apropriadamente. Uma lista vinculada permite operações de inserção eficientes em qualquer lugar da lista.

Dica de desempenho 21.3

Os elementos de um array são armazenados contiguamente na memória. Isso permite acesso imediato a qualquer elemento do array, porque o endereço de qualquer elemento pode ser calculado diretamente com base em sua posição em relação ao começo do array. As listas vinculadas não têm recursos para suportar esse 'acesso direto' imediato aos seus elementos. Então, acessar elementos individuais em uma lista vinculada pode ser consideravelmente mais caro que acessar elementos individuais em um array. A seleção de uma estrutura de dados é, em geral, baseada no desempenho de operações específicas utilizadas por um programa e na ordem em que os itens de dados são mantidos na estrutura de dados. Por exemplo, normalmente é mais eficiente inserir um item em uma lista vinculada classificada do que em um array classificado.

Nós de lista vinculada normalmente não são armazenados contiguamente na memória. Logicamente, porém, os nós de uma lista vinculada parecem ser contíguos. A Figura 21.2 ilustra uma lista vinculada com vários nós.

Dica de desempenho 21.4

Utilizar alocação dinâmica de memória (em vez de arrays de tamanho fixo) para estruturas de dados que crescem e encolhem em tempo de execução pode poupar memória. Tenha em mente, porém, que os ponteiros ocupam espaço e que a alocação dinâmica de memória incorre no overhead de chamadas de função.

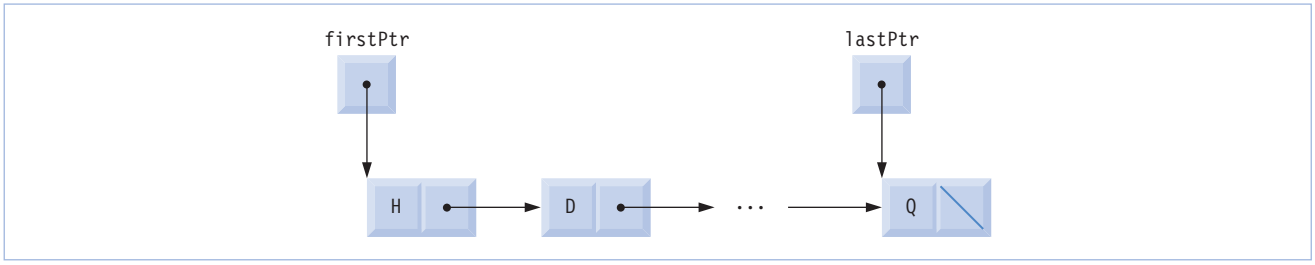

Figura 21.2 Representação gráfica de uma lista.

Implementação de lista vinculada

O programa das figuras 21.3–21.5 utiliza um template de classe `List` (ver Capítulo 14 para obter informações sobre templates de classe) para manipular uma lista de valores de inteiro e uma lista de valores de ponto flutuante. O programa de driver (Figura 21.5) fornece cinco opções: 1) inserir um valor no começo da lista, 2) inserir um valor no final da lista, 3) excluir um valor do início da lista, 4) excluir um valor do fim da lista e 5) terminar o processamento da lista. Uma discussão detalhada do programa se segue. O Exercício 21.20 pede para você implementar uma função recursiva que imprime uma lista vinculada de trás para a frente, e o Exercício 21.21 pede para você implementar uma função recursiva que pesquisa em uma lista vinculada um item de dados particular.

O programa utiliza os templates de classe `ListNode` (Figura 21.3) e `List` (Figura 21.4). Encapsulada em cada objeto `List` está uma lista vinculada de objetos `ListNode`. O template de classe `ListNode` (Figura 21.3) contém os membros `private data` e `nextPtr` (linhas 19–20), um construtor para inicializar esses membros e a função `getData` para retornar os dados em um nó. O membro `data` armazena um valor de tipo `NODETYPE`, o parâmetro de tipo passado ao template de classe. O membro `nextPtr` armazena um ponteiro para o próximo objeto `ListNode` na lista vinculada. Observe que a linha 13 da definição de template de classe `ListNode` declara a classe `List< NODETYPE >` como um `friend`. Isso torna todas as funções-membro de uma dada especialização de template da classe `List` amigas (friends) da especialização correspondente do template de classe `ListNode`, então elas podem acessar os membros `private` de objetos `ListNode` desse tipo. Como o parâmetro `NODETYPE` do template `ListNode` é utilizado como o argumento de template para `List`

```
1  // Figura 21.3: Listnode.h
2  // Definição do template de classe ListNode.
3  #ifndef LISTNODE_H
4  #define LISTNODE_H
5
6  // declaração antecipada da classe List necessária para anunciar essa classe
7  // List existe, portanto pode ser utilizada na declaração friend na linha 13
8  template< typename NODETYPE > class List;
9
10 template< typename NODETYPE>
11 class ListNode
12 {
13    friend class List< NODETYPE >; // torna List uma amiga (friend)
14
15 public:
16    ListNode( const NODETYPE & ); // construtor
17    NODETYPE getData() const; // retorna dados no nó
18 private:
19    NODETYPE data; // dados
20    ListNode< NODETYPE > *nextPtr; // próximo nó na lista
21 }; // fim da classe ListNode
22
23 // construtor
24 template< typename NODETYPE>
25 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
26    : data( info ), nextPtr( 0 )
27 {
28    // corpo vazio
29 } // fim do construtor ListNode
30
31 // retorna cópia de dados no nó
32 template< typename NODETYPE >
33 NODETYPE ListNode< NODETYPE >::getData() const
34 {
35    return data;
36 } // fim da função getData
37
38 #endif
```

Figura 21.3 Definição do template de classe `ListNode`.

na declaração `friend`, os `ListNode`s especializados com um tipo particular podem ser processados somente por uma `List` especializada com o mesmo tipo (por exemplo, uma `List` de valores `int` gerencia os objetos `ListNode` que armazenam os valores `int`).

As linhas 24–25 do template de classe `List` (Figura 21.4) declaram os membros de dados `private` `firstPtr` (um ponteiro para o primeiro `ListNode` em uma `List`) e `lastPtr` (um ponteiro para o último `ListNode` em uma `List`). O construtor-padrão (linhas 32–37) inicializa ambos os ponteiros como `0` (nulo). O destrutor (linhas 40–60) assegura que todos os objetos `ListNode` em um objeto `List` são destruídos quando esse objeto `List` é destruído. As principais funções `List` são `insertAtFront` (linhas 63–75), `insertAtBack` (linhas 78–90), `removeFromFront` (linhas 93–111) e `removeFromBack` (linhas 114–141).

A função `isEmpty` (linhas 144–148) é chamada de função de predicado — não altera a `List`; em vez disso, determina se a `List` está vazia (isto é, o ponteiro para o primeiro nó da `List` é nulo). Se a `List` estiver vazia, `true` é retornado; caso contrário, `false` é retornado. A função `print` (linhas 159–179) exibe o conteúdo da `List`. A função utilitária `getNewNode` (linhas 151–156) retorna um objeto `ListNode` alocado dinamicamente. Essa função é chamada a partir das funções `insertAtFront` e `insertAtBack`.

```
1  // Figura 21.4: List.h
2  // Definição do template de classe List.
3  #ifndef LIST_H
4  #define LIST_H
5
6  #include <iostream>
7  using std::cout;
8
9  #include "listnode.h" // definição da classe ListNode
10
11 template< typename NODETYPE >
12 class List
13 {
14 public:
15    List(); // construtor
16    ~List(); // destrutor
17    void insertAtFront( const NODETYPE & );
18    void insertAtBack( const NODETYPE & );
19    bool removeFromFront( NODETYPE & );
20    bool removeFromBack( NODETYPE & );
21    bool isEmpty() const;
22    void print() const;
23 private:
24    ListNode< NODETYPE > *firstPtr; // ponteiro para o primeiro nó
25    ListNode< NODETYPE > *lastPtr; // ponteiro para o último nó
26
27    // função utilitária para alocar novo nó
28    ListNode< NODETYPE > *getNewNode( const NODETYPE & );
29 }; // fim da classe List
30
31 // construtor-padrão
32 template< typename NODETYPE >
33 List< NODETYPE >::List()
34    : firstPtr( 0 ), lastPtr( 0 )
35 {
36    // corpo vazio
37 } // fim do construtor List
38
39 // destrutor
40 template< typename NODETYPE >
41 List< NODETYPE >::~List()
42 {
43    if ( !isEmpty() ) // List não está vazia
```

Figura 21.4 Definição do template de classe `List`. (continua)

```
   {
      cout << "Destroying nodes ...\n";

      ListNode< NODETYPE > *currentPtr = firstPtr;
      ListNode< NODETYPE > *tempPtr;

      while ( currentPtr != 0 ) // exclui nós restantes
      {
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      } // fim do while
   } // fim do if

   cout << "All nodes destroyed\n\n";
} // fim do destrutor List

// insere nó na frente da lista
template< typename NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // novo nó

   if ( isEmpty() ) // List está vazia
      firstPtr = lastPtr = newPtr; // nova lista tem apenas um nó
   else // List não está vazia
   {
      newPtr->nextPtr = firstPtr; // aponta novo nó para o primeiro nó anterior
      firstPtr = newPtr; // aponta firstPtr para o novo nó
   } // fim do else
} // fim da função insertAtFront

// insere nó no fim da lista
template< typename NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // novo nó

   if ( isEmpty() ) // List está vazia
      firstPtr = lastPtr = newPtr; // nova lista tem apenas um nó
   else // List não está vazia
   {
      lastPtr->nextPtr = newPtr; // atualiza o último nó anterior
      lastPtr = newPtr; // novo último nó
   } // fim do else
} // fim da função insertAtBack

// exclui nó da frente da lista
template< typename NODETYPE >
bool List< NODETYPE >::removeFromFront( NODETYPE &value )
{
   if ( isEmpty() ) // List está vazia
      return false; // exclusão malsucedida
   else
   {
```

Figura 21.4 Definição do template de classe `List`. (continua)

```
100        ListNode< NODETYPE > *tempPtr = firstPtr; // armazena tempPtr para excluir
101
102        if ( firstPtr == lastPtr )
103           firstPtr = lastPtr = 0; // nenhum nó permanece depois da exclusão
104        else
105           firstPtr = firstPtr->nextPtr; // aponta para segundo nó anterior
106
107        value = tempPtr->data; // retorna os dados sendo removidos
108        delete tempPtr; // reivindica nó frontal anterior
109        return true; // exclusão bem-sucedida
110     } // fim do else
111  } // fim da função removeFromFront
112
113  // exclui nó do fim da lista
114  template< typename NODETYPE >
115  bool List< NODETYPE >::removeFromBack( NODETYPE &value )
116  {
117     if ( isEmpty() ) // List está vazia
118        return false; // exclusão malsucedida
119     else
120     {
121        ListNode< NODETYPE > *tempPtr = lastPtr; // armazena tempPtr para excluir
122
123        if ( firstPtr == lastPtr ) // List tem um elemento
124           firstPtr = lastPtr = 0; // nenhum nó permanece depois da exclusão
125        else
126        {
127           ListNode< NODETYPE > *currentPtr = firstPtr;
128
129           // localiza do segundo ao último elemento
130           while ( currentPtr->nextPtr != lastPtr )
131              currentPtr = currentPtr->nextPtr; // move para próximo nó
132
133           lastPtr = currentPtr; // remove último nó
134           currentPtr->nextPtr = 0; // esse é agora o último nó
135        } // fim do else
136
137        value = tempPtr->data; // retorna valor do último nó antigo
138        delete tempPtr; // reivindica o primeiro último nó
139        return true; // exclusão bem-sucedida
140     } // fim do else
141  } // fim da função removeFromBack
142
143  // List está vazia?
144  template< typename NODETYPE >
145  bool List< NODETYPE >::isEmpty() const
146  {
147     return firstPtr == 0;
148  } // fim da função isEmpty
149
150  // retorna ponteiro para nó recentemente alocado
151  template< typename NODETYPE >
152  ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
153     const NODETYPE &value )
154  {
155     return new ListNode< NODETYPE >( value );
```

Figura 21.4 Definição do template de classe `List`. (continua)

```
156 } // fim da função getNewNode
157
158 // exibe o conteúdo de List
159 template< typename NODETYPE >
160 void List< NODETYPE >::print() const
161 {
162    if ( isEmpty() ) // List está vazia
163    {
164       cout << "The list is empty\n\n";
165       return;
166    } // fim do if
167
168    ListNode< NODETYPE > *currentPtr = firstPtr;
169
170    cout << "The list is: ";
171
172    while ( currentPtr != 0 ) // obtém dados de elemento
173    {
174       cout << currentPtr->data << ' ';
175       currentPtr = currentPtr->nextPtr;
176    } // fim do while
177
178    cout << "\n\n";
179 } // fim da função print
180
181 #endif
```

Figura 21.4 Definição do template de classe `List`. (continuação)

Dica de prevenção de erro 21.1

Atribua nulo (0) ao membro de vínculo de um novo nó. Os ponteiros devem ser inicializados antes de ser utilizados.

O programa de driver (Figura 21.5) utiliza o template de função `testList` para permitir ao usuário manipular objetos da classe `List`. As linhas 74 e 78 criam objetos `List` para tipos `int` e `double`, respectivamente. As linhas 75 e 79 invocam o template de função `testList` com esses objetos `List`.

```
1  // Figura 21.5: Fig21_05.cpp
2  // Programa de teste da classe List.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  #include <string>
9  using std::string;
10
11 #include "List.h" // definição da classe List
12
13 // função para testar um List
14 template< typename T >
15 void testList( List< T > &listObject, const string &typeName )
16 {
```

Figura 21.5 Manipulando uma lista vinculada. (continua)

```
17     cout << "Testing a List of " << typeName << " values\n";
18     instructions(); // exibe instruções
19
20     int choice; // armazena a escolha do usuário
21     T value; // armazena valor de entrada
22
23     do // realiza ações selecionadas pelo usuário
24     {
25        cout << "? ";
26        cin >> choice;
27
28        switch ( choice )
29        {
30           case 1: // insere no início
31              cout << "Enter " << typeName << ": ";
32              cin >> value;
33              listObject.insertAtFront( value );
34              listObject.print();
35              break;
36           case 2: // insere no final
37              cout << "Enter " << typeName << ": ";
38              cin >> value;
39              listObject.insertAtBack( value );
40              listObject.print();
41              break;
42           case 3: // remove do início
43              if ( listObject.removeFromFront( value ) )
44                 cout << value << " removed from list\n";
45
46              listObject.print();
47              break;
48           case 4: // remove do final
49              if ( listObject.removeFromBack( value ) )
50                 cout << value << " removed from list\n";
51
52              listObject.print();
53              break;
54        } // fim do switch
55     } while ( choice != 5 ); // fim da instrução do...while
56
57     cout << "End list test\n\n";
58  } // fim da função testList
59
60  // exibe instruções de programa para o usuário
61  void instructions()
62  {
63     cout << "Enter one of the following:\n"
64        << "  1 to insert at beginning of list\n"
65        << "  2 to insert at end of list\n"
66        << "  3 to delete from beginning of list\n"
67        << "  4 to delete from end of list\n"
68        << "  5 to end list processing\n";
69  } // fim da função instructions
70
71  int main()
72  {
```

Figura 21.5 Manipulando uma lista vinculada. (continua)

```
   // testa List de valores int
   List< int > integerList;
   testList( integerList, "integer" );

   // testa List de valores double
   List< double > doubleList;
   testList( doubleList, "double" );
   return 0;
} // fim do main
```

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
```

Figura 21.5 Manipulando uma lista vinculada. (continua)

```
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Figura 21.5 Manipulando uma lista vinculada. (continuação)

Função-membro ***insertAtFront***

Nas várias páginas a seguir, discutimos cada uma das funções-membro da classe `List` em detalhes. A função `insertAtFront` (Figura 21.4, linhas 63–75) coloca um novo nó na frente da lista. A função consiste em vários passos:

1. Chame a função `getNewNode` (linha 66), passando para ela o `value`, que é uma referência constante ao valor do nó a ser inserido.
2. A função `getNewNode` (linhas 151–156) utiliza o operador `new` para criar um novo nó de lista e retorna um ponteiro para esse nó recém-alocado, que é atribuído a `newPtr` em `insertAtFront` (linha 66).
3. Se a lista estiver vazia (linha 68), então tanto `firstPtr` como `lastPtr` são configurados como `newPtr` (linha 69).
4. Se a lista não estiver vazia (linha 70), então o nó apontado por `newPtr` é encadeado na lista copiando `firstPtr` em `newPtr->nextPtr` (linha 72), para que o novo nó aponte ao que costumava ser o primeiro nó da lista e copie `newPtr` em `firstPtr` (linha 73), para que `firstPtr` agora aponte ao novo primeiro nó da lista.

A Figura 21.6 ilustra a função `insertAtFront`. A parte (a) da figura mostra a lista e o novo nó antes da operação `insertAtFront`. As setas tracejadas na parte (b) ilustram o Passo 4 da operação `insertAtFront` que permite ao nó contendo `12` tornar-se a nova frente da lista.

Função-membro `insertAtBack`

A função `insertAtBack` (Figura 21.4, linhas 78–90) coloca um novo nó no fim da lista. A função consiste em vários passos:

1. Chame a função `getNewNode` (linha 81), passando para ela o `value`, que é uma referência constante ao valor do nó a ser inserido.
2. A função `getNewNode` (linhas 151–156) utiliza o operador `new` para criar um novo nó de lista e retorna um ponteiro para esse nó recém-alocado, que é atribuído a `newPtr` em `insertAtBack` (linha 81).
3. Se a lista estiver vazia (linha 83), então tanto `firstPtr` como `lastPtr` são configurados como `newPtr` (linha 84).
4. Se a lista não estiver vazia (linha 85), então o nó apontado por `newPtr` é encadeado na lista copiando `newPtr` em `lastPtr->nextPtr` (linha 87), para que o novo nó seja apontado pelo que costumava ser o último nó da lista e copiando `newPtr` em `lastPtr` (linha 88), para que `lastPtr` agora aponte ao novo último nó da lista.

A Figura 21.7 ilustra uma operação `insertAtBack`. A parte (a) da figura mostra a lista e o novo nó antes da operação. As setas tracejadas na parte (b) ilustram o Passo 4 da função `insertAtBack` que permite a um novo nó ser adicionado ao final de uma lista que não está vazia.

Função-membro `removeFromFront`

A função `removeFromFront` (Figura 21.4, linhas 93–111) remove o nó frontal da lista e copia o valor de nó no parâmetro de referência. A função retorna `false` se uma tentativa de remover um nó de uma lista vazia for feita (linhas 96–97) e retorna `true` se a remoção for bem-sucedida. A função consiste em vários passos:

1. Atribua a `tempPtr` o endereço para qual `firstPtr` aponta (linha 100). Por fim, `tempPtr` será utilizado para excluir o nó sendo removido.
2. Se `firstPtr` for igual a `lastPtr` (linha 102), isto é, se a lista tiver apenas um elemento antes da tentativa de remoção, então configure `firstPtr` e `lastPtr` como zero (linha 103) para desencadear esse nó da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nó antes da remoção, então deixe `lastPtr` como está e configure `firstPtr` como `firstPtr->nextPtr` (linha 105); isto é, modifique `firstPtr` para apontar para o que era o segundo nó antes da remoção (e agora é o novo primeiro nó).
4. Depois que todas essas manipulações de ponteiro estiverem completas, copie para o parâmetro de referência `value` o membro `data` do nó sendo removido (linha 107).
5. Agora exclua (`delete`) o nó apontado por `tempPtr` (linha 108).
6. Retorne `true`, que indica a remoção bem-sucedida (linha 109).

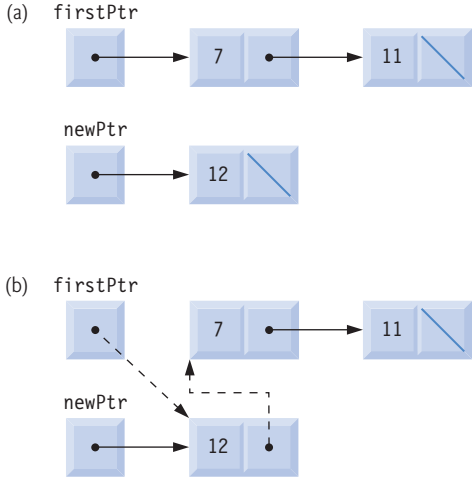

Figura 21.6 A operação `insertAtFront` representada graficamente.

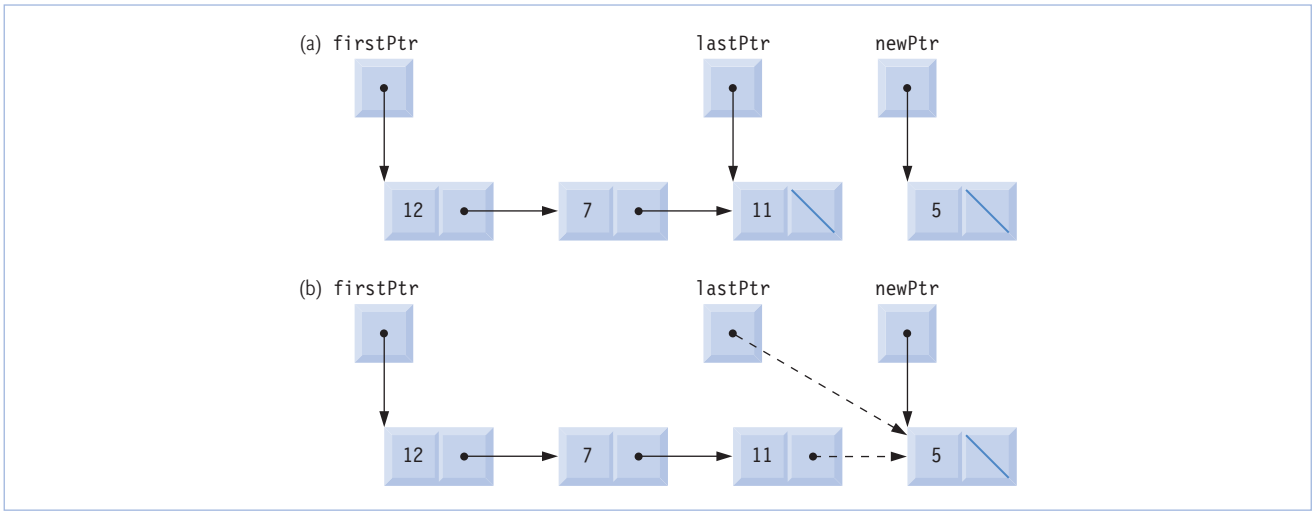

Figura 21.7 A operação `insertAtBack` representada graficamente.

A Figura 21.8 ilustra a função `removeFromFront`. A parte (a) ilustra a lista antes da operação de remoção. A parte (b) mostra as manipulações reais de ponteiro para remover o nó frontal de uma lista não vazia.

Função-membro `removeFromBack`

A função `removeFromBack` (Figura 21.4, linhas 114–141) remove o nó posterior da lista e copia o valor do nó para o parâmetro de referência. A função retorna `false` se uma tentativa de remover um nó de uma lista vazia for feita (linhas 117–118) e retorna `true` se a remoção for bem-sucedida. A função consiste em vários passos:

1. Atribua a `tempPtr` o endereço para o qual `lastPtr` aponta (linha 121). Por fim, `tempPtr` será utilizado para excluir o nó sendo removido.
2. Se `firstPtr` for igual a `lastPtr` (linha 123), isto é, se a lista tiver apenas um elemento antes da tentativa de remoção, então configure `firstPtr` e `lastPtr` como zero (linha 124) para desencadear esse nó da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nó antes da remoção, então atribua a `currentPtr` o endereço para o qual `firstPtr` aponta (linha 127) para 'percorrer a lista'.

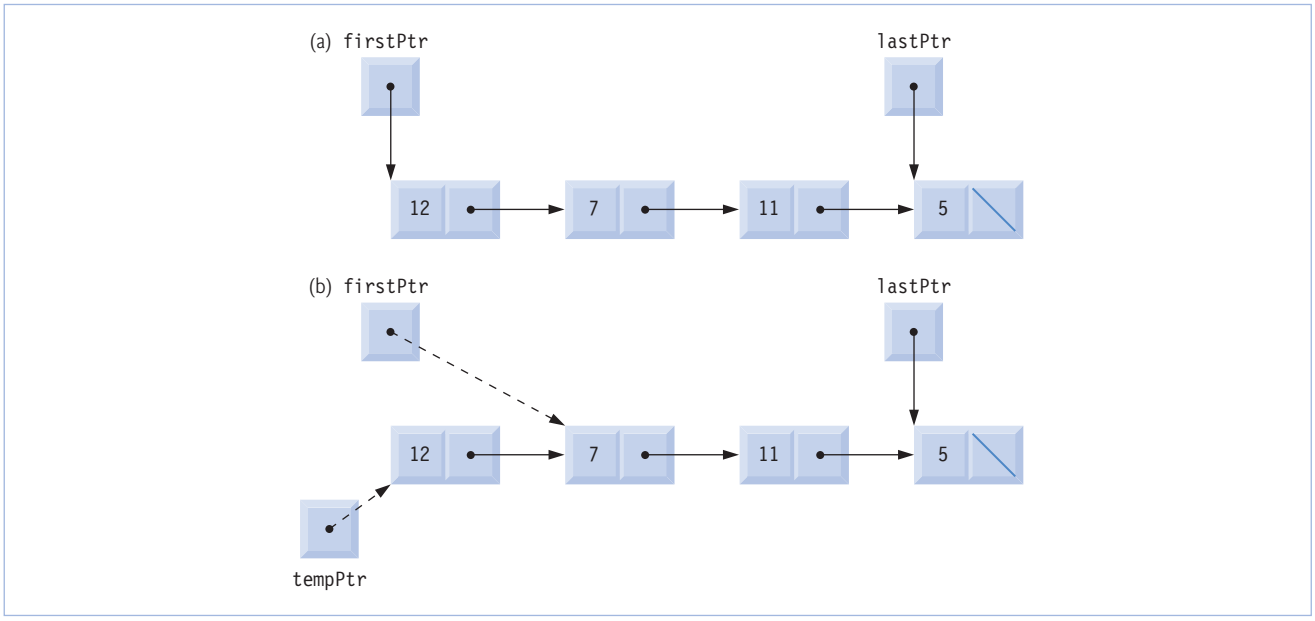

Figura 21.8 A operação `removeFromFront` representada graficamente.

4. Agora 'percorra lista' com `currentPtr` até ela apontar para o nó antes do último nó. Esse nó se tornará o último nó depois que a operação de remoção estiver completa. Isso é feito com um loop `while` (linhas 130–131) que continua substituindo `currentPtr` por `currentPtr->nextPtr`, enquanto `currentPtr->nextPtr` não é `lastPtr`.
5. Atribua `lastPtr` ao endereço para o qual `currentPtr` aponta (linha 133) para desencadear o nó posterior da lista.
6. Configure `currentPtr->nextPtr` como zero (linha 134) no novo último nó da lista.
7. Depois que todas as manipulações de ponteiro forem concluídas, copie para o parâmetro de referência `value` o membro `data` do nó sendo removido (linha 137).
8. Agora exclua (`delete`) o nó apontado por `tempPtr` (linha 138).
9. Retorne `true` (linha 139), que indica que a remoção foi bem-sucedida.

A Figura 21.9 ilustra `removeFromBack`. A parte (a) da figura ilustra a lista antes da operação de remoção. A parte (b) da figura mostra as manipulações de ponteiro reais.

Função-membro ***print***

A função `print` (linhas 159–179) primeiro determina se a lista está vazia (linha 162). Se estiver, imprime `"The list is empty"` e retorna (linhas 164–165). Caso contrário, itera pela lista e gera saída do valor em cada nó. A função inicializa `currentPtr` como uma cópia de `firstPtr` (linha 168), então imprime a string `"The list is: "` (linha 170). Enquanto `currentPtr` não for nulo (linha 172), `currentPtr->data` é impresso (linha 174) e `currentPtr` recebe o valor de `currentPtr->nextPtr` (linha 175). Observe que, se o vínculo no último nó da lista não for nulo, o algoritmo de impressão imprimirá erroneamente antes do final da lista. O algoritmo de impressão é idêntico para listas vinculadas, pilhas e filas (porque baseamos cada uma dessas estruturas de dados na mesma infra-estrutura de lista vinculada).

Listas lineares e circulares simples e duplamente vinculadas

O tipo de lista vinculada que discutimos é uma **lista simplesmente vinculada** — a lista inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó 'na seqüência'. Essa lista termina com um nó cujo membro ponteiro tem o valor 0. Uma lista simplesmente vinculada pode ser percorrida em apenas uma direção.

Uma **lista circular simplesmente vinculada** (Figura 21.10) inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó. O 'último nó' não contém um ponteiro 0; em vez disso, o ponteiro no último nó aponta de volta para o primeiro nó, fechando, assim, o 'círculo'.

Uma **lista duplamente vinculada** (Figura 21.11) permite percorrer para a frente e para trás. Tal lista é freqüentemente implementada com dois 'ponteiros iniciais' — um que aponta para o primeiro elemento da lista para permitir o percurso da frente para trás da lista e um que aponta para o último elemento para permitir o percurso de trás para a frente. Cada nó tem tanto um ponteiro para a frente para o próximo nó na lista na direção para a frente como um ponteiro para trás para o próximo nó na lista na direção para trás. Se sua lista

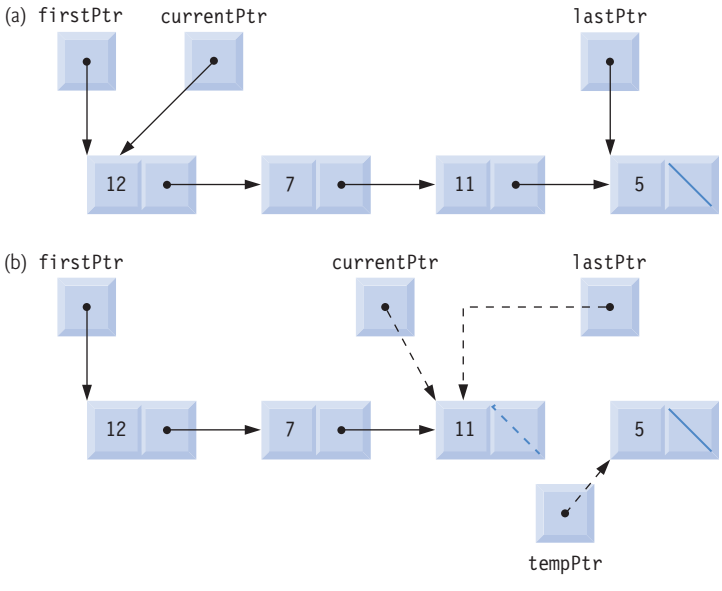

Figura 21.9 A operação `removeFromBack` representada graficamente.

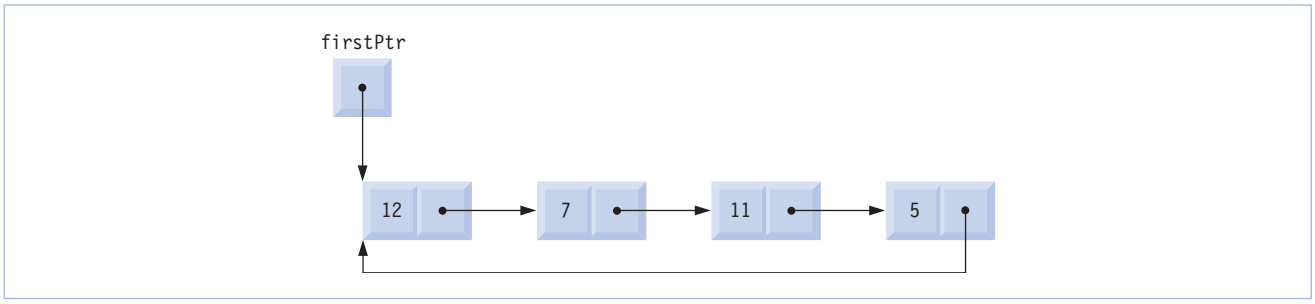

Figura 21.10 Lista circular simplesmente vinculada.

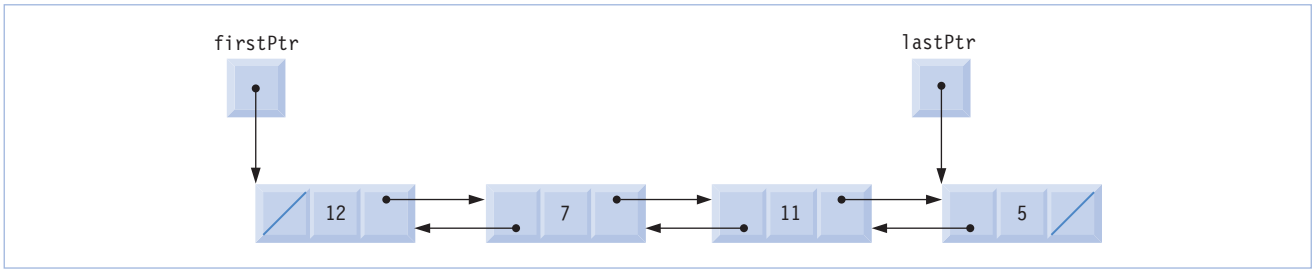

Figura 21.11 Lista duplamente vinculada.

contém um diretório de telefones indexados em ordem alfabética, por exemplo, uma pesquisa por alguém cujo nome inicia com uma letra próxima do início do alfabeto poderia começar a partir do topo da lista. Procurar alguém cujo nome inicia com uma letra próxima do fim do alfabeto poderia começar a partir do final da lista.

Em uma **lista circular duplamente vinculada** (Figura 21.12), o ponteiro para a frente da lista do último nó aponta para o primeiro nó e o ponteiro para trás do primeiro nó aponta para o último nó, fechando assim o 'círculo'.

21.5 Pilhas

No Capítulo 14, "Templates", explicamos a noção de um template de classe de pilhas com uma implementação de array subjacente. Nesta seção, utilizamos uma implementação de lista vinculada baseada em ponteiro subjacente. Discutimos também as pilhas no Capítulo 23, "Standard Template Library (STL)".

Uma estrutura de dados de pilha permite que nós sejam adicionados à pilha e removidos da pilha somente na parte superior. Por essa razão, uma pilha é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*last-in, first-out* – LIFO). Uma maneira de implementar uma pilha é como uma versão restrita de uma lista vinculada. Em tal implementação, o membro de vínculo no último nó da pilha é configurado como nulo (zero) para indicar a parte inferior da pilha.

As principais funções-membro utilizadas para manipular uma pilha são `push` e `pop`. A função `push` insere um novo nó na parte superior da pilha. A função `pop` remove um nó da parte superior da pilha, armazena o valor removido em uma variável de referência que é passada para função chamadora e retorna `true` se a operação `pop` foi bem-sucedida (`false`, caso contrário).

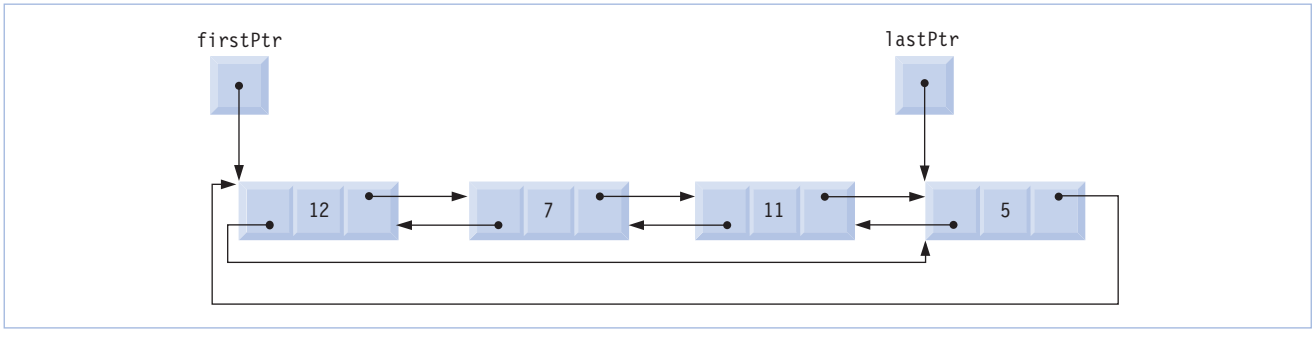

Figura 21.12 Lista circular duplamente vinculada.

As pilhas têm muitas aplicações interessantes. Por exemplo, quando uma chamada de função é feita, a função chamada deve saber retornar para seu chamador, portanto o endereço de retorno é inserido em uma pilha. Se uma série de chamadas de função ocorre, os valores sucessivos de retorno são inseridos na pilha na ordem primeiro a entrar, primeiro a sair, de modo que cada função possa retornar para seu chamador. As pilhas suportam chamadas de função recursiva da mesma maneira que as chamadas não recursivas convencionais. A Seção 6.11 discute a pilha de chamadas de função em detalhes.

As pilhas fornecem a memória para, e armazenam os valores de, variáveis automáticas em cada invocação de uma função. Quando a função retorna para seu chamador ou lança uma exceção, o destrutor (se houver algum) de cada objeto local é chamado, o espaço para as variáveis automáticas dessa função é removido da pilha e essas variáveis não são mais conhecidas pelo programa.

As pilhas são utilizadas por compiladores no processo de avaliar expressões e gerar código de linguagem de máquina. Os exercícios exploram várias aplicações de pilhas, inclusive utilizá-las para desenvolver seu próprio compilador funcional completo.

Tiramos proveito do íntimo relacionamento entre listas e pilhas para implementar uma classe de pilha principalmente reutilizando uma classe de lista. Primeiro, implementamos a classe de pilha por herança `private` da classe `List`. Então implementamos uma classe de pilha de idêntico desempenho por meio de composição incluindo um objeto lista como um membro `private` de uma classe de pilhas. Naturalmente, todas as estruturas de dados deste capítulo, incluindo essas duas classes de pilha, são implementadas como templates para encorajar mais reusabilidade.

O programa das figuras 21.13–21.14 cria um template de classe `Stack` (Figura 21.13) principalmente por meio da herança `private` (linha 9) do template de classe `List` da Figura 21.4. Queremos que a `Stack` tenha funções-membro `push` (linhas 13–16), `pop` (linhas

```
1  // Figura 21.13: Stack.h
2  // Definição do template de classe Stack derivada da classe List.
3  #ifndef STACK_H
4  #define STACK_H
5
6  #include "List.h" // definição da classe List
7
8  template< typename STACKTYPE >
9  class Stack : private List< STACKTYPE >
10 {
11 public:
12    // push chama a função List insertAtFront
13    void push( const STACKTYPE &data )
14    {
15       insertAtFront( data );
16    } // fim da função push
17
18    // pop chama a função List removeFromFront
19    bool pop( STACKTYPE &data )
20    {
21       return removeFromFront( data );
22    } // fim da função pop
23
24    // isStackEmpty chama a função List isEmpty
25    bool isStackEmpty() const
26    {
27       return isEmpty();
28    } // fim da função isStackEmpty
29
30    // printStack chama a função List print
31    void printStack() const
32    {
33       print();
34    } // fim da função print
35 }; // fim da classe Stack
36
37 #endif
```

Figura 21.13 Definição do template de classe `Stack`.

19–22), `isStackEmpty` (linhas 25–28) e `printStack` (linhas 31–34). Observe que essas são essencialmente as funções `insertAtFront`, `removeFromFront`, `isEmpty` e `print` do template da classe `List`. Naturalmente, o template de classe `List` contém outras funções-membro (isto é, `insertAtBack` e `removeFromBack`) que não gostaríamos de tornar acessível pela interface `public` para a classe `Stack`. Então, quando indicamos que o template de classe `Stack` deve herdar do template de classe `List`, especificamos a herança `private`. Isso torna `private` todas as funções-membro do template de classe `List` no template de classe `Stack`. Ao implementarmos as funções-membro de `Stack`, fazemos então cada uma dessas funções chamar a função-membro apropriada da classe `List` — `push` chama `insertAtFront` (linha 15), `pop` chama `removeFromFront` (linha 21), `isStackEmpty` chama `isEmpty` (linha 27) e `printStack` chama `print` (linha 33) — isso é referido como **delegação**.

O template de classe de pilhas é utilizado em `main` (Figura 21.14) para instanciar a pilha de inteiros `intStack` do tipo `Stack< int >` (linha 11). Os inteiros de 0 a 2 são inseridos em `intStack` (linhas 16–20) e, então, removidos de `intStack` (linhas 25–30). O programa utiliza o template da classe `Stack` para criar `doubleStack` do tipo `Stack< double >` (linha 32). Os valores 1.1, 2.2 e 3.3 são inseridos em `doubleStack` (linhas 38–43) e, então, removidos de `doubleStack` (linhas 48–53).

Outra maneira de implementar um template de classe `Stack` é reutilizando o template de classe `List` por meio da composição. A Figura 21.15 é uma nova implementação do template de classe `Stack` que contém um objeto `List< STACKTYPE >` chamado `stackList` (linha 38). Essa versão do template de classe `Stack` utiliza a classe `List` da Figura 21.4. Para testar essa classe, use o programa de driver da Figura 21.14, mas inclua o novo arquivo de cabeçalho — `Stackcomposition.h` na linha 6 desse arquivo. A saída do programa é idêntica para ambas as versões da classe `Stack`.

```
1  // Figura 21.14: Fig21_14.cpp
2  // Programa de teste do template de classe Stack.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Stack.h" // definição da classe Stack
8
9  int main()
10 {
11    Stack< int > intStack; // cria Stack de ints
12
13    cout << "processing an integer Stack" << endl;
14
15    // insere inteiros em intStack
16    for ( int i = 0; i < 3; i++ )
17    {
18       intStack.push( i );
19       intStack.printStack();
20    } // fim do for
21
22    int popInteger; // armazena o int removido da pilha
23
24    // remove os inteiros de intStack
25    while ( !intStack.isStackEmpty() )
26    {
27       intStack.pop( popInteger );
28       cout << popInteger << " popped from stack" << endl;
29       intStack.printStack();
30    } // fim do while
31
32    Stack< double > doubleStack; // cria Stack de doubles
33    double value = 1.1;
34
35    cout << "processing a double Stack" << endl;
36
37    // insere valores de ponto flutuante em doubleStack
```

Figura 21.14 Um programa simples de pilha. (continua)

```
38      for ( int j = 0; j < 3; j++ )
39      {
40         doubleStack.push( value );
41         doubleStack.printStack();
42         value += 1.1;
43      } // fim do for
44
45      double popDouble; // armazena o double removido da pilha
46
47      // remove os valores de ponto flutuante de doubleStack
48      while ( !doubleStack.isStackEmpty() )
49      {
50         doubleStack.pop( popDouble );
51         cout << popDouble << " popped from stack" << endl;
52         doubleStack.printStack();
53      } // fim do while
54
55      return 0;
56   } // fim do main
```

```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty

processing a double Stack
The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

Figura 21.14 Um programa simples de pilha. (continuação)

```
1  // Figura 21.15: Stackcomposition.h
2  // Definição do template de classe Stack com objeto List composto.
3  #ifndef STACKCOMPOSITION_H
4  #define STACKCOMPOSITION_H
5
6  #include "List.h" // definição da classe List
7
8  template< typename STACKTYPE >
9  class Stack
10 {
11 public:
12    // nenhum construtor; construtor List faz inicialização
13
14    // push chama a função-membro insertAtFront do objeto stackList
15    void push( const STACKTYPE &data )
16    {
17       stackList.insertAtFront( data );
18    } // fim da função push
19
20    // pop chama a função-membro removeFromFront do objeto stackList
21    bool pop( STACKTYPE &data )
22    {
23       return stackList.removeFromFront( data );
24    } // fim da função pop
25
26    // isStackEmpty chama a função-membro isEmpty do objeto stackList
27    bool isStackEmpty() const
28    {
29       return stackList.isEmpty();
30    } // fim da função isStackEmpty
31
32    // printStack chama a função-membro print do objeto stackList
33    void printStack() const
34    {
35       stackList.print();
36    } // fim da função printStack
37 private:
38    List< STACKTYPE > stackList; // objeto List composto
39 }; // fim da classe Stack
40
41 #endif
```

Figura 21.15 O template de classe `Stack` com um objeto `List` composto.

21.6 Filas

Uma **fila** (*queue*) é semelhante a uma fila de pessoas no caixa de um supermercado — a primeira pessoa na fila é atendida em primeiro lugar, e outros clientes entram no fim da fila e esperam pelo atendimento. Os nós da fila são removidos apenas do início (ou cabeça) da fila e são inseridos somente no final (ou cauda) da fila. Por essa razão, uma fila é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). As operações de inserção e remoção são conhecidas como **enqueue** e **dequeue**.

As filas têm muitas aplicações em sistemas de computador. Os computadores que têm um único processador podem atender apenas um usuário por vez. As entradas para os outros usuários são colocadas em uma fila. Cada entrada avança gradualmente para a frente da fila quando os usuários recebem o serviço. A entrada na frente da fila é a próxima a receber o serviço.

As filas também são utilizadas para suportar **spooling de impressão**. Por exemplo, uma única impressora talvez seja compartilhada por todos os usuários de uma rede. Muitos usuários podem enviar trabalhos de impressão à impressora, mesmo quando a impressora já estiver ocupada. Esses trabalhos de impressão são colocados em uma fila até a impressora ficar disponível. Um programa chamado **spooler** gerencia a fila para assegurar que, à medida que cada trabalho de impressão é concluído, o próximo trabalho de impressão é enviado à impressora.

Os pacotes de informação também esperam em filas em redes de computadores. Toda vez que um pacote chega a um nó de rede, ele deve ser roteado para o próximo nó na rede ao longo do caminho até o destino final do pacote. O nó de roteamento roteia um pacote por vez, então pacotes adicionais são enfileirados até o roteador conseguir roteá-los.

Um servidor de arquivos em uma rede de computadores trata as solicitações de acesso de arquivo de muitos clientes por toda a rede. Os servidores têm uma capacidade limitada para servir solicitações de clientes. Quando essa capacidade é excedida, as solicitações dos clientes esperam em filas.

O programa das figuras 21.16–21.17 cria um template de classe `Queue` (Figura 21.16) pela herança `private` (linha 9) do template de classe `List` da Figura 21.4. Queremos que a `Queue` tenha funções-membro `enqueue` (linhas 13–16), `dequeue` (linhas 19–22), `isQueueEmpty` (linhas 25–28) e `printQueue` (linhas 31–34). Observe que essas são essencialmente as funções `insertAtBack`, `removeFromFront`, `isEmpty` e `print` do template da classe `List`. Naturalmente, o template de classe `List` contém outras funções-membro (isto é, `insertAtFront` e `removeFromBack`) que não gostaríamos de tornar acessíveis pela interface `public` para a classe `Queue`. Então, quando indicamos que o template de classe `Queue` deve herdar o template de classe `List`, especificamos a herança `private`. Isso torna `private` todas as funções-membro do template de classe `List` no template de classe `Queue`. Quando implementamos as funções-membro de `Queue`, fazemos cada uma dessas funções chamar a função-membro apropriada da classe de listas — `enqueue` chama `insertAtBack` (linha 15), `dequeue` chama `removeFromFront` (linha 21), `isQueueEmpty` chama `isEmpty` (linha 27) e `printQueue` chama `print` (linha 33). Novamente, isso é chamado delegação.

```
1  // Figura 21.16: Queue.h
2  // Definição do template de classe Queue derivada da classe List.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "List.h" // definição da classe List
7
8  template< typename QUEUETYPE >
9  class Queue : private List< QUEUETYPE >
10 {
11 public:
12    // enqueue chama função-membro List insertAtBack
13    void enqueue( const QUEUETYPE &data )
14    {
15       insertAtBack( data );
16    } // fim da função enqueue
17
18    // denqueue chama a função-membro List removeFromFront
19    bool dequeue( QUEUETYPE &data )
20    {
21       return removeFromFront( data );
22    } // fim da função dequeue
23
24    // isQueueEmpty chama a função-membro List isEmpty
25    bool isQueueEmpty() const
26    {
27       return isEmpty();
28    } // fim da função isQueueEmpty
29
30    // printQueue chama função-membro List print
31    void printQueue() const
32    {
33       print();
34    } // fim da função printQueue
35 }; // fim da classe Queue
36
37 #endif
```

Figura 21.16 Definição do template de classe `Queue`.

A Figura 21.17 utiliza o template de classe `Queue` para instanciar a fila de inteiros `intQueue` do tipo `Queue< int >` (linha 11). Os inteiros de 0 a 2 são enfileirados para `intQueue` (linhas 16–20) e, então, desenfileirados de `intQueue` na ordem primeiro a entrar, primeiro a sair (linhas 25–30). Em seguida, o programa instancia a fila `doubleQueue` do tipo `Queue< double >` (linha 32). Os valores 1.1, 2.2 e 3.3 são enfileirados para `doubleQueue` (linhas 38–43) e, então, desenfileirados de `doubleQueue` na ordem primeiro a entrar, primeiro a sair (linhas 48–53).

```
1  // Figura 21.17: Fig21_17.cpp
2  // Programa de teste do template de classe Queue.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Queue.h" // definição da classe Queue
8
9  int main()
10 {
11    Queue< int > intQueue; // cria Queue de inteiros
12
13    cout << "processing an integer Queue" << endl;
14
15    // enfileira inteiros em intQueue
16    for ( int i = 0; i < 3; i++ )
17    {
18       intQueue.enqueue( i );
19       intQueue.printQueue();
20    } // fim do for
21
22    int dequeueInteger; // armazena inteiro desenfileirado
23
24    // desenfileira inteiros de intQueue
25    while ( !intQueue.isQueueEmpty() )
26    {
27       intQueue.dequeue( dequeueInteger );
28       cout << dequeueInteger << " dequeued" << endl;
29       intQueue.printQueue();
30    } // fim do while
31
32    Queue< double > doubleQueue; // criar Queue de doubles
33    double value = 1.1;
34
35    cout << "processing a double Queue" << endl;
36
37    // enfileira valores de ponto flutuante em doubleQueue
38    for ( int j = 0; j < 3; j++ )
39    {
40       doubleQueue.enqueue( value );
41       doubleQueue.printQueue();
42       value += 1.1;
43    } // fim do for
44
45    double dequeueDouble; // armazena double desenfileirado
46
47    // desenfileira valores de ponto flutuante de doubleQueue
48    while ( !doubleQueue.isQueueEmpty() )
49    {
```

Figura 21.17 Programa de processamento de fila. (continua)

```
50          doubleQueue.dequeue( dequeueDouble );
51          cout << dequeueDouble << " dequeued" << endl;
52          doubleQueue.printQueue();
53       } // fim do while
54
55       return 0;
56    } // fim do main
```

```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

0 dequeued
The list is: 1 2

1 dequeued
The list is: 2

2 dequeued
The list is empty

processing a double Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

1.1 dequeued
The list is: 2.2 3.3

2.2 dequeued
The list is: 3.3

3.3 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

Figura 21.17 Programa de processamento de fila. (continuação)

21.7 Árvores

As listas vinculadas, pilhas e filas são estruturas de dados lineares. Uma árvore é uma estrutura de dados bidimensional não-linear. Os nós da árvore contêm dois ou mais vínculos. Esta seção discute as **árvores binárias** (Figura 21.18) — árvores cujos nós contêm, cada um, dois vínculos (nenhum, um ou ambos os quais podem ser nulos).

Terminologia básica

Para os propósitos desta discussão, consulte os nós A, B, C e D na Figura 21.18. O **nó-raiz** (nó `b`) é o primeiro nó em uma árvore. Cada vínculo no nó-raiz referencia um **filho** (nós `A` e `D`). O **filho esquerdo** (nó `a`) é o nó-raiz da **subárvore esquerda** (que contém apenas o nó `a`), e o **filho direito** (nó `d`) é o nó-raiz da **subárvore direita** (que contém os nós `D` e `C`). Os filhos de um único nó são chamados **irmãos** (por exemplo, nós `A` e `D` são irmãos). Um nó sem filhos é chamado **nó-folha** (por exemplo, nós `A` e `C` são nós-folha). Em geral,

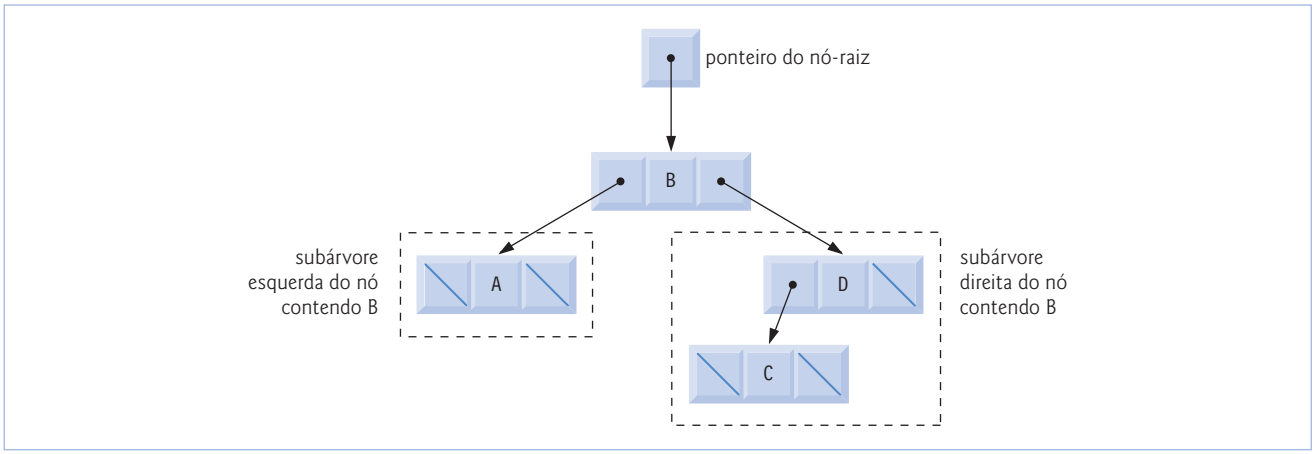

Figura 21.18 Uma representação gráfica de uma árvore binária.

os cientistas da computação normalmente desenham árvores a partir do nó-raiz para baixo — exatamente o oposto de como as árvores crescem na natureza.

Árvores de pesquisa binária

Esta seção discute uma árvore binária especial chamada de **árvore de pesquisa binária**. Uma árvore de pesquisa binária (sem valores de nó duplicados) tem a característica de que os valores em qualquer subárvore esquerda são menores que o valor em seu **nó-pai**, e os valores em qualquer subárvore direita são maiores que o valor em seu nó-pai. A Figura 21.19 ilustra uma árvore de pesquisa binária com 9 valores. Observe que a forma da árvore de pesquisa binária que corresponde a um conjunto de dados pode variar, dependendo da ordem em que os valores são inseridos na árvore.

Implementando o programa de árvore de pesquisa binária

O programa das figuras 21.20–21.22 cria uma árvore de pesquisa binária e a percorre (isto é, passa por todos os seus nós) de três maneiras — utilizando os percursos recursivos **na ordem**, **pré-ordem** e **pós-ordem**. Explicamos esses algoritmos de percurso em breve.

Começamos nossa discussão com o programa de driver (Figura 21.22), então continuamos com as implementações de classes `TreeNode` (Figura 21.20) e `Tree` (Figura 21.21). A função `main` (Figura 21.22) começa instanciando a árvore de inteiro `intTree` do tipo `Tree< int >` (linha 15). O programa solicita 10 inteiros, cada um dos quais é inserido na árvore binária chamando `insertNode` (linha 24). O programa então realiza os percursos pré-ordem, na ordem e pós-ordem (esses serão explicados em breve) da `intTree` (linhas 28, 31 e 34, respectivamente). O programa então instancia a árvore de ponto flutuante `doubleTree` do tipo `Tree< double >` (linha 36). O programa solicita 10 valores `double`, cada um dos quais é inserido na árvore binária chamando `insertNode` (linha 46). Em seguida, o programa realiza os percursos pré-ordem, na ordem e pós-ordem de `doubleTree` (linhas 50, 53 e 56, respectivamente).

Agora discutiremos as definições do template de classe. Iniciamos com a definição de template de classe `TreeNode` (Figura 21.20) que declara `Tree< NODETYPE >` como seu `friend` (linha 13). Isso torna todas as funções-membro de uma dada especialização de template da classe `Tree` (Figura 21.21) amigas da especialização correspondente do template de classe `TreeNode`. Desse modo elas podem acessar os membros `private` dos objetos `TreeNode` desse tipo. Como o parâmetro `NODETYPE` do template `TreeNode` é utilizado como o argumento de template para `Tree` na declaração `friend`, os `TreeNodes` especializados com um tipo particular podem ser processados somente por uma `Tree` especializada com o mesmo tipo (por exemplo, uma `Tree` de valores `int` gerencia os objetos `TreeNode` que armazenam os valores `int`).

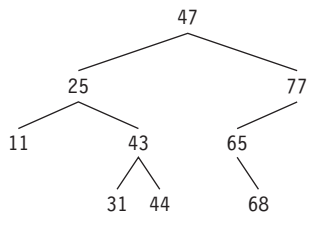

Figura 21.19 Uma árvore de pesquisa binária.

```
1  // Figura 21.20: Treenode.h
2  // Definição do template de classe TreeNode.
3  #ifndef TREENODE_H
4  #define TREENODE_H
5
6  // encaminha a declaração da classe Tree
7  template< typename NODETYPE > class Tree;
8
9  // definição do template de classe TreeNode
10 template< typename NODETYPE >
11 class TreeNode
12 {
13    friend class Tree< NODETYPE >;
14 public:
15    // construtor
16    TreeNode( const NODETYPE &d )
17       : leftPtr( 0 ), // ponteiro para subárvore esquerda
18         data( d ), // dados de nó de árvore
19         rightPtr( 0 ) // ponteiro para subárvore direita
20    {
21       // corpo vazio
22    } // fim do construtor TreeNode
23
24    // retorna a cópia de dados do nó
25    NODETYPE getData() const
26    {
27       return data;
28    } // fim da função getData
29 private:
30    TreeNode< NODETYPE > *leftPtr; // ponteiro para subárvore esquerda
31    NODETYPE data;
32    TreeNode< NODETYPE > *rightPtr; // ponteiro para subárvore direita
33 }; // fim da classe TreeNode
34
35 #endif
```

Figura 21.20 Definição do template de classe `TreeNode`.

```
1  // Figura 21.21: Tree.h
2  // Definição do template de classe Tree.
3  #ifndef TREE_H
4  #define TREE_H
5
6  #include <iostream>
7  using std::cout;
8  using std::endl;
9
10 #include "Treenode.h"
11
12 // definição do template de classe Tree
13 template< typename NODETYPE > class Tree
14 {
15 public:
16    Tree(); // construtor
```

Figura 21.21 Definição do template de classe `Tree`. (continua)

```
17    void insertNode( const NODETYPE & );
18    void preOrderTraversal() const;
19    void inOrderTraversal() const;
20    void postOrderTraversal() const;
21 private:
22    TreeNode< NODETYPE > *rootPtr;
23
24    // funções utilitárias
25    void insertNodeHelper( TreeNode< NODETYPE > **, const NODETYPE & );
26    void preOrderHelper( TreeNode< NODETYPE > * ) const;
27    void inOrderHelper( TreeNode< NODETYPE > * ) const;
28    void postOrderHelper( TreeNode< NODETYPE > * ) const;
29 }; // fim da classe Tree
30
31 // construtor
32 template< typename NODETYPE >
33 Tree< NODETYPE >::Tree()
34 {
35    rootPtr = 0; // indica que a árvore está inicialmente vazia
36 } // fim do construtor Tree
37
38 // insere nó em Tree
39 template< typename NODETYPE >
40 void Tree< NODETYPE >::insertNode( const NODETYPE &value )
41 {
42    insertNodeHelper( &rootPtr, value );
43 } // fim da função insertNode
44
45 // função utilitária chamada por insertNode; recebe um ponteiro
46 // para que a função possa modificar o valor do ponteiro
47 template< typename NODETYPE >
48 void Tree< NODETYPE >::insertNodeHelper(
49    TreeNode< NODETYPE > **ptr, const NODETYPE &value )
50 {
51    // subárvore está vazia; cria novo TreeNode contendo o valor
52    if ( *ptr == 0 )
53       *ptr = new TreeNode< NODETYPE >( value );
54    else // subárvore não está vazia
55    {
56       // os dados a inserir são menores que os dados no nó atual
57       if ( value < ( *ptr )->data )
58          insertNodeHelper( &( ( *ptr )->leftPtr ), value );
59       else
60       {
61          // os dados a inserir são maiores que os dados no nó atual
62          if ( value > ( *ptr )->data )
63             insertNodeHelper( &( ( *ptr )->rightPtr ), value );
64          else // duplica valor dos dados ignorados
65             cout << value << " dup" << endl;
66       } // fim do else
67    } // fim do else
68 } // fim da função insertNodeHelper
69
70 // inicia o percurso na pré-ordem de Tree
71 template< typename NODETYPE >
72 void Tree< NODETYPE >::preOrderTraversal() const
```

Figura 21.21 Definição do template de classe `Tree`. (continua)

```
{
   preOrderHelper( rootPtr );
} // fim da função preOrderTraversal

// função utilitária para executar o percurso na pré-ordem de Tree
template< typename NODETYPE >
void Tree< NODETYPE >::preOrderHelper( TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      cout << ptr->data << ' '; // processa nó
      preOrderHelper( ptr->leftPtr ); // percorre subárvore esquerda
      preOrderHelper( ptr->rightPtr ); // percorre subárvore direita
   } // fim do if
} // fim da função preOrderHelper

// inicia percurso na ordem de Tree
template< typename NODETYPE >
void Tree< NODETYPE >::inOrderTraversal() const
{
   inOrderHelper( rootPtr );
} // fim da função inOrderTraversal

// função utilitária para realizar o percurso na ordem de Tree
template< typename NODETYPE >
void Tree< NODETYPE >::inOrderHelper( TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      inOrderHelper( ptr->leftPtr ); // percorre subárvore esquerda
      cout << ptr->data << ' '; // processa nó
      inOrderHelper( ptr->rightPtr ); // percorre subárvore direita
   } // fim do if
} // fim da função inOrderHelper

// inicia o percurso na pós-ordem de Tree
template< typename NODETYPE >
void Tree< NODETYPE >::postOrderTraversal() const
{
   postOrderHelper( rootPtr );
} // fim da função postOrderTraversal

// função utilitária para realizar o percurso de pós-ordem de Tree
template< typename NODETYPE >
void Tree< NODETYPE >::postOrderHelper(
   TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      postOrderHelper( ptr->leftPtr ); // percorre subárvore esquerda
      postOrderHelper( ptr->rightPtr ); // percorre subárvore direita
      cout << ptr->data << ' '; // processa nó
   } // fim do if
} // fim da função postOrderHelper

#endif
```

Figura 21.21 Definição do template de classe `Tree`. (continuação)

As linhas 30–32 declaram os dados `private` de um `TreeNode` — o valor `data` do nó, e os ponteiros `leftPtr` (para a subárvore esquerda do nó) e `rightPtr` (para a subárvore direita do nó). O construtor (linhas 16–22) configura `data` como o valor fornecido como um argumento de construtor e configura os ponteiros `leftPtr` e `rightPtr` como zero (inicializando, assim, esse nó como um nó-folha). A função-membro `getData` (linhas 25–28) retorna o valor `data`.

O template de classe `Tree` (Figura 21.21) tem como dados `private rootPtr` (linha 22), um ponteiro para o nó-raiz da árvore. As linhas 17–20 do template de classe declaram as funções-membro `public insertNode` (que insere um novo nó na árvore) e `preOrderTraversal`, `inOrderTraversal` e `postOrderTraversal`, cada uma das quais percorre a árvore da maneira designada. Cada uma dessas funções-membro chama sua própria função utilitária recursiva separada para realizar as operações apropriadas na representação interna da árvore, assim o programa não precisa acessar os dados `private` subjacentes para realizar essas funções. Lembre-se de que a recursão requer que passemos um ponteiro que representa a próxima subárvore a ser processada. O construtor `Tree` inicializa `rootPtr` como zero para indicar que a árvore está inicialmente vazia.

A função utilitária `insertNodeHelper` (linhas 47–68) da classe `Tree` é chamada por `insertNode` (linhas 39–43) para inserir recursivamente um nó na árvore. *Um nó pode ser inserido somente como um nó-folha em uma árvore de pesquisa binária.* Se a árvore está vazia, um novo `TreeNode` é criado, inicializado e inserido na árvore (linhas 52–53).

Se a árvore não está vazia, o programa compara o valor a ser inserido com o valor `data` no nó-raiz. Se o valor de inserção é menor (linha 57), o programa chama `insertNodeHelper` (linha 58) recursivamente para inserir o valor na subárvore esquerda. Se o valor de inserção é maior (linha 62), o programa chama `insertNodeHelper` (linha 63) recursivamente para inserir o valor na subárvore direita. Se o valor a ser inserido é idêntico ao valor dos dados no nó-raiz, o programa imprime a mensagem `"dup"` (linha 65) e retorna sem inserir o valor duplicado na árvore. Observe que `insertNode` passa o endereço de `rootPtr` para `insertNodeHelper` (linha 42). Desse modo, ele pode modificar o valor armazenado em `rootPtr` (isto é, o endereço do nó-raiz). Para receber um ponteiro para `rootPtr` (que é também um ponteiro), o primeiro argumento do `insertNodeHelper` é declarado como um ponteiro para um ponteiro para um `TreeNode`.

Cada uma das funções-membro `inOrderTraversal` (linhas 90–94), `preOrderTraversal` (linhas 71–75) e `postOrderTraversal` (linhas 109–113) percorre a árvore e imprime os valores do nó. Para o propósito da próxima discussão, utilizamos a árvore de pesquisa binária da Figura 21.23.

```
1  // Figura 21.22: Fig21_22.cpp
2  // Programa de teste de classe Tree.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include "Tree.h" // definição da classe Tree
12
13 int main()
14 {
15    Tree< int > intTree; // cria Tree de valores int
16    int intValue;
17
18    cout << "Enter 10 integer values:\n";
19
20    // insere 10 inteiros em intTree
21    for ( int i = 0; i < 10; i++ )
22    {
23       cin >> intValue;
24       intTree.insertNode( intValue );
25    } // fim do for
26
27    cout << "\nPreorder traversal\n";
28    intTree.preOrderTraversal();
29
30    cout << "\nInorder traversal\n";
```

Figura 21.22 Criando e percorrendo uma árvore binária. (continua)

```
31      intTree.inOrderTraversal();
32
33      cout << "\nPostorder traversal\n";
34      intTree.postOrderTraversal();
35
36      Tree< double > doubleTree; // cria Tree de valores double
37      double doubleValue;
38
39      cout << fixed << setprecision( 1 )
40         << "\n\n\nEnter 10 double values:\n";
41
42      // insere 10 doubles em doubleTree
43      for ( int j = 0; j < 10; j++ )
44      {
45         cin >> doubleValue;
46         doubleTree.insertNode( doubleValue );
47      } // fim do for
48
49      cout << "\nPreorder traversal\n";
50      doubleTree.preOrderTraversal();
51
52      cout << "\nInorder traversal\n";
53      doubleTree.inOrderTraversal();
54
55      cout << "\nPostorder traversal\n";
56      doubleTree.postOrderTraversal();
57
58      cout << endl;
59      return 0;
60   } // fim do main
```

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50


Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```

Figura 21.22 Criando e percorrendo uma árvore binária. (continuação)

Algoritmo do percurso na ordem

A função `inOrderTraversal` invoca a função utilitária `inOrderHelper` para realizar a percurso na ordem da árvore binária. Os passos de um percurso na ordem são:

1. Percorrer a subárvore esquerda com um percurso na ordem. (Isso é realizado pela chamada a `inOrderHelper` na linha 102.)
2. Processar o valor no nó — isto é, imprime o valor de nó (linha 103).
3. Percorrer a subárvore direita com um percurso na ordem. (Isso é realizado pela chamada a `inOrderHelper` na linha 104.)

O valor em um nó não é processado até que os valores em sua subárvore esquerda sejam processados, porque cada chamada para `inOrderHelper` chama imediatamente de novo `inOrderHelper` com o ponteiro para a subárvore esquerda. O percurso na ordem da árvore na Figura 21.23 é

```
6 13 17 27 33 42 48
```

Observe que o percurso na ordem de uma árvore de pesquisa binária imprime os valores de nó na ordem crescente. O processo de criar uma árvore de pesquisa binária realmente classifica os dados — portanto, esse processo é chamado **classificação de árvore binária**.

Algoritmo de percurso pré-ordem

A função `preOrderTraversal` invoca a função utilitária `preOrderHelper` para realizar o percurso pré-ordem da árvore binária. Os passos de um percurso pré-ordem são:

1. Processar o valor no nó (linha 83).
2. Percorrer a subárvore esquerda com um percurso pré-ordem. (Isso é realizado pela chamada a `preOrderHelper` na linha 84.)
3. Percorrer a subárvore direita com um percurso pré-ordem. (Isso é realizado pela chamada a `preOrderHelper` na linha 85.)

O valor em cada nó é processado quando o nó é visitado. Depois que o valor em um dado nó é processado, os valores na subárvore esquerda são processados. Então os valores na subárvore direita são processados. O percurso pré-ordem da árvore na Figura 21.23 é

```
27 13 6 17 42 33 48
```

Algoritmo de percurso pós-ordem

A função `postOrderTraversal` invoca a função utilitária `postOrderHelper` para realizar a percurso pós-ordem da árvore binária. Os passos de um percurso pós-ordem são:

1. Percorrer a subárvore esquerda com um percurso pós-ordem. (Isso é realizado pela chamada a `postOrderHelper` na linha 122.)
2. Percorrer a subárvore direita com um percurso pós-ordem. (Isso é realizado pela chamada a `postOrderHelper` na linha 123.)
3. Processar o valor no nó (linha 124).

O valor em cada nó não é impresso até os valores de seus filhos serem impressos. A `postOrderTraversal` da árvore na Figura 21.23 é

```
6 17 13 33 48 42 27
```

Eliminação de duplicatas

A árvore de pesquisa binária facilita a **eliminação de duplicatas**. À medida que a árvore é criada, uma tentativa de inserir um valor duplicado será reconhecida, porque uma duplicata seguirá as mesmas decisões 'vá para a esquerda' ou 'vá para a direita' em cada comparação da mesma forma que o valor original faz ao ser inserido na árvore. Portanto, a duplicata será finalmente comparada com um nó que contém o mesmo valor. O valor duplicado pode ser descartado nesse ponto.

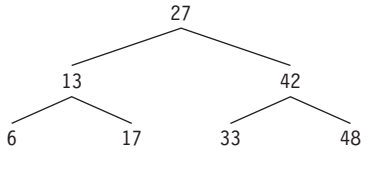

Figura 21.23 Uma árvore de pesquisa binária.

Procurar em uma árvore binária um valor que corresponda a um valor-chave também é rápido. Se a árvore é equilibrada, então cada desvio contém cerca da metade do número de nós na árvore. Cada comparação de um nó com a chave de pesquisa elimina metade dos nós. Isso é chamado de algoritmo $O(\log n)$ (a notação O é discutida no Capítulo 20). Portanto, uma árvore de pesquisa binária com n elementos requereria um máximo de $\log_2 n$ comparações para localizar uma correspondência ou determinar que não há nenhuma correspondência. Isso significa, por exemplo, que, ao pesquisar em uma árvore (equilibrada) de pesquisa binária de 1.000 elementos, não é preciso fazer mais que 10 comparações, porque $2^{10} > 1.000$. Ao pesquisar em uma árvore (equilibrada) de pesquisa binária de 1.000.000 elementos, não mais que 20 comparações são necessárias, porque $2^{20} > 1.000.000$.

Visão geral dos exercícios de árvore binária

Nos exercícios, os algoritmos são apresentados para várias outras operações de árvore binária, como excluir um item de uma árvore binária, imprimir uma árvore binária em um formato de árvore bidimensional e realizar um percurso na ordem de nível de uma árvore binária. O percurso na ordem de nível de uma árvore binária percorre os nós da árvore linha por linha iniciando no nível do nó-raiz. Em cada nível da árvore, os nós são percorridos da esquerda para a direita. Outros exercícios de árvore binária incluem permitir que uma árvore de pesquisa binária contenha valores duplicados, insira valores de string em uma árvore binária e determine quantos níveis estão contidos em uma árvore binária.

21.8 Síntese

Neste capítulo, você aprendeu que as listas vinculadas são coleções de itens de dados ‘vinculados em uma cadeia’. Também aprendeu que um programa pode realizar inserções e exclusões em qualquer lugar de uma lista vinculada (embora nossa implementação tenha executado somente inserções e exclusões nos finais da lista). Demonstramos que as estruturas de dados da pilha e de fila são versões restritas de listas. Quanto às pilhas, vimos que as inserções e exclusões só podem ser feitas no topo. Quanto às filas que representam seqüências de espera, você viu que inserções são feitas na cauda (na parte de trás) e as exclusões são feitas na cabeça (na parte da frente). Apresentamos também a estrutura de dados de árvore binária. Você viu uma árvore de pesquisa binária que facilitou a pesquisa e classificação de dados em alta velocidade e a eliminação eficiente de duplicatas. Por todo o capítulo, você aprendeu a criar essas estruturas de dados para facilitar o reúso (como templates) e a manutenção. No próximo capítulo, introduziremos as `structs`, que são semelhantes às classes, e discutiremos a manipulação de bits, caracteres e strings no estilo do C.

Resumo

- As estruturas de dados dinâmicas crescem e encolhem durante a execução.
- As listas vinculadas são coleções de itens de dados ‘vinculados em uma cadeia’ — inserções e exclusões são feitas em qualquer lugar de uma lista vinculada.
- As pilhas são importantes em compiladores e sistemas operacionais: as inserções e exclusões são feitas somente no final de uma pilha — sua parte superior.
- As filas representam filas de espera; as inserções são feitas na parte de trás (também referida como cauda) de uma fila e as remoções são feitas na parte da frente da fila (também referida como cabeça).
- As árvores binárias facilitam a pesquisa e a classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistema de arquivos e a compilação de expressões em linguagem de máquina.
- Uma classe auto-referencial contém um membro ponteiro que aponta para um objeto de classe do mesmo tipo de classe.
- Os objetos de classe auto-referencial podem ser vinculados para formar estruturas de dados úteis como listas, filas, pilhas e árvores.
- O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador ou a quantidade de memória virtual disponível em um sistema de memória virtual.
- Uma lista vinculada é uma coleção linear de objetos de classe auto-referencial, chamados nós, conectados por vínculos de ponteiro — daí, o termo lista ‘vinculada’.
- Uma lista vinculada é acessada via ponteiro ao primeiro nó da lista. Cada nó subseqüente é acessado via membro ponteiro de vínculo armazenado no nó anterior.
- As listas vinculadas, pilhas e filas são estruturas de dados lineares. As árvores são estruturas de dados não-lineares.
- Uma lista vinculada é apropriada quando o número de elementos de dados a ser representado em um tempo é imprevisível.
- As listas vinculadas são dinâmicas, portanto o comprimento de uma lista pode aumentar ou diminuir conforme necessário.
- Uma lista simplesmente vinculada inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó ‘na seqüência’.
- Uma lista circular simplesmente vinculada inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó. O ‘último nó’ não contém um ponteiro nulo; em vez disso, o ponteiro no último nó aponta de volta para o primeiro nó, fechando, assim, o ‘círculo’.

- Uma lista duplamente vinculada permite tanto percursos para a frente como para trás.
- Uma lista duplamente vinculada é freqüentemente implementada com dois 'ponteiros iniciais' — um que aponta para o primeiro elemento da lista a fim de permitir percurso de frente para trás na lista, e outro que aponta para o último elemento a fim de permitir percurso de trás para a frente. Cada nó tem um ponteiro para a frente para o próximo nó na lista na direção para a frente, e um ponteiro para trás para o próximo nó na direção para trás.
- Em uma lista circular duplamente vinculada, o ponteiro para a frente do último nó aponta para o primeiro nó, e o ponteiro para trás do primeiro nó aponta para o último nó, fechando, assim, o 'círculo'.
- Uma estrutura de dados de pilha permite que nós sejam adicionados à pilha e removidos da pilha somente na parte superior.
- Uma pilha é referida como uma estrutura de dados último a entrar, primeiro a sair (*last-in, first-out* – LIFO).
- As principais funções-membro utilizadas para manipular uma pilha são `push` e `pop`. A função `push` insere um novo nó na parte superior da pilha. A função `pop` remove um nó da parte superior da pilha.
- Uma fila é semelhante a uma fila de caixa em um supermercado — a primeira pessoa na fila é atendida primeiro, e outros clientes entram no fim da fila e esperam o atendimento.
- Os nós da fila são removidos apenas do início (ou cabeça) da fila e são inseridos somente no final (ou cauda) da fila.
- Uma fila é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). As operações de inserção e remoção são conhecidas como `enqueue` e `dequeue`.
- As árvores binárias são árvores cujos nós contêm dois vínculos (nenhum, um ou ambos podem ser nulos).
- O nó-raiz é o primeiro nó em uma árvore.
- Cada vínculo no nó-raiz referencia um filho. O filho esquerdo é o nó-raiz da subárvore esquerda e o filho direito é o nó-raiz da subárvore direita.
- Os filhos de um nó individual são chamados irmãos. Um nó sem filhos é chamado de nó-folha.
- Uma árvore de pesquisa binária (sem valores duplicados de nó) tem a característica de que os valores em qualquer subárvore esquerda são menores que o valor em seu nó-pai e os valores em qualquer subárvore direita são maiores que o valor em seu nó-pai.
- Um nó pode apenas ser inserido como um nó-folha em uma árvore de pesquisa binária.
- Um percurso na ordem de uma árvore binária percorre a subárvore esquerda na ordem, processa o valor no nó-raiz e, então, percorre a subárvore direita na ordem. O valor em um nó não é processado até os valores em sua subárvore esquerda serem processados.
- Um percurso pré-ordem processa o valor no nó-raiz, percorre a subárvore esquerda na pré-ordem e, então, percorre a subárvore direita na pré-ordem. O valor em cada nó é processado quando o nó é encontrado.
- Um percurso pós-ordem percorre a subárvore esquerda na pós-ordem, percorre a subárvore direita na pós-ordem e, então, processa o valor no nó-raiz. O valor em cada nó não é processado até que os valores em suas duas subárvores sejam processados.
- A árvore de pesquisa binária facilita a eliminação de duplicatas. Durante a criação da árvore, uma tentativa de inserir um valor duplicado será reconhecida e o valor duplicado pode ser descartado.
- O percurso na ordem de nível de uma árvore binária percorre os nós da árvore linha por linha iniciando no nível do nó-raiz. Em cada nível da árvore, os nós são percorridos da esquerda para a direita.

Terminologia

árvore binária
árvore de pesquisa binária
cabeça de uma fila
classificação árvore binária
delegação
`dequeue`
eliminação de duplicatas
`enqueue`
estrutura auto-referencial
estrutura de dados
estrutura de dados linear
estrutura de dados não-linear
estruturas de dados dinâmicas
fila
filho direito
filho esquerdo
final de uma fila
inserir um nó
irmãos
lista circular duplamente vinculada
lista circular simplesmente vinculada
lista duplamente vinculada
lista simplesmente vinculada
lista vinculada
nó
nó-filho
nó-folha
nó-pai
nó-raiz
parte superior de uma pilha
percurso na ordem de uma árvore binária
percurso na ordem do nível
percurso pós-ordem de uma árvore binária
percurso pré-ordem de uma árvore binária
pilha
`pop`
primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO)
`push`
spooler
spooling de impressão
subárvore direita
subárvore esquerda
último a entrar, primeiro a sair (*last-in, first-out* – LIFO)
vínculo (*link*)
vínculo de ponteiro

Exercícios de revisão

21.1 Preencha as lacunas em cada uma das seguintes sentenças:

a) Uma classe auto-_____ é utilizada para formar estruturas de dados dinâmicas que podem crescer e encolher em tempo de execução.
b) O operador _____ é utilizado para alocar memória dinamicamente e construir um objeto; esse operador retorna um ponteiro para o objeto.
c) Um(a) _____ é uma versão restrita de uma lista vinculada em que os nós podem ser inseridos e excluídos somente a partir do início da lista e valores de nó são retornados na ordem último a entrar, primeiro a sair.
d) Uma função que não altera uma lista vinculada, mas examina a lista para determinar se ela está vazia, é um exemplo de uma função _____.
e) Uma fila é referida como uma estrutura de dados _____ porque os primeiros nós inseridos são os primeiros nós removidos.
f) O ponteiro para o próximo nó em uma lista vinculada é referido como _____.
g) O operador _____ é utilizado para destruir um objeto e liberar memória dinamicamente alocada.
h) Um(a) _____ é uma versão restrita de uma lista vinculada em que os nós podem ser inseridos apenas no final da lista e excluídos apenas do início da lista.
i) Um(a) _____ é uma estrutura de dados bidimensional não-linear que contém nós com dois ou mais vínculos.
j) Uma pilha é referida como uma estrutura de dados _____, porque o último nó inserido é o primeiro nó removido.
k) Os nós de uma árvore _____ contém dois membros de vínculo.
l) O primeiro nó de uma árvore é o nó- _____.
m) Cada vínculo em um nó de árvore aponta para um _____ ou _____ desse nó.
n) Um nó de árvore que não tem filhos é chamado de nó- _____.
o) Os quatro algoritmos de percursos que mencionamos no texto para árvores de pesquisa binária são _____, _____, _____ e _____.

21.2 Quais são as diferenças entre uma lista vinculada e uma pilha?

21.3 Quais são as diferenças entre uma pilha e uma fila?

21.4 Talvez um título mais apropriado para este capítulo fosse "Estruturas de dados reutilizáveis". Comente como cada uma das seguintes entidades ou conceitos contribuem para a capacidade de reutilização das estruturas de dados:

a) classes
b) templates de classe
c) herança
d) herança `private`
e) composição

21.5 Forneça manualmente os percursos na ordem, pré-ordem e pós-ordem da árvore de pesquisa binária da Figura 21.24.

Respostas dos exercícios de revisão

21.1 a) referencial. b) `new`. c) pilha. d) predicado. e) primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). f) vínculo (*link*). g) `delete`. h) fila. i) árvore. j) último a entrar, primeiro a sair (*last-in, first-out* – LIFO). k) binária. l) raiz. m) filho ou subárvore. n) folha. o) na ordem, pré-ordem, pós-ordem e na ordem do nível.

21.2 É possível inserir um nó em qualquer lugar em uma lista vinculada e remover um nó de qualquer lugar em uma lista vinculada. Os nós em uma pilha só podem ser inseridos na parte superior da pilha e só podem ser removidos da parte superior de uma pilha.

21.3 Uma estrutura de dados de fila permite que os nós sejam removidos somente da parte superior da fila e inseridos somente na parte inferior da fila. Uma fila é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). Uma estrutura de

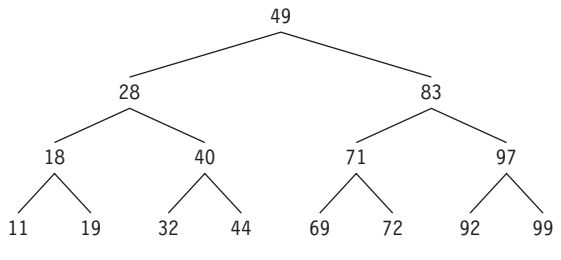

Figura 21.24 Um árvore de pesquisa binária de 15 nós.

dados de pilha permite que nós sejam adicionados à pilha e removidos da pilha somente na parte superior. Uma pilha é referida como uma estrutura de dados último a entrar, primeiro a sair (*last-in, first-out* – LIFO).

21.4 a) As classes nos permitem instanciar quantos objetos de estrutura de dados de certo tipo (isto é, classe) quisermos.
b) Os templates de classe nos permitem instanciar classes relacionadas, cada uma com base em diferentes parâmetros de tipo — então podemos gerar quantos objetos de cada classe de template quisermos.
c) A herança nos permite reutilizar o código de uma classe básica em uma classe derivada, de modo que a estrutura de dados de classe derivada também seja uma estrutura de dados de classe básica (isto é, com a herança `public`).
d) Herança `private` permite reutilizar partes do código de uma classe básica para formar uma estrutura de dados de classe derivada; como a herança é `private`, todas as funções-membro `public` de classe básica tornam-se `private` na classe derivada. Isso nos permite impedir que clientes da estrutura de dados de classe derivada acessem as funções-membro da classe básica que não se aplicam à classe derivada.
e) A composição nos permite reutilizar código tornando uma estrutura de dados de um objeto de classe um membro de uma classe composta; se tornarmos o objeto de classe um membro `private` da classe composta, então as funções-membro públicas do objeto de classe não são disponíveis por meio da interface do objeto composto.

21.5 O percurso na ordem é

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

O percurso pré-ordem é

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

O percurso pós-ordem é

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Exercícios

21.6 Escreva um programa que concatena dois objetos de lista vinculada de caracteres. O programa deve incluir a função `concatenate`, que aceita referências a ambos os objetos da lista como argumentos e concatena a segunda com a primeira lista.

21.7 Escreva um programa que mescla dois objetos ordenados de lista de inteiros em um único objeto ordenado de lista de inteiros. A função `merge` deve receber referências a cada um dos objetos de lista a ser mesclado e referenciar o objeto lista em que os elementos mesclados serão colocados.

21.8 Escreva um programa que insere 25 inteiros aleatórios de 0 a 100 na ordem em um objeto lista vinculada. O programa deve calcular a soma dos elementos e a média de ponto flutuante dos elementos.

21.9 Escreva um programa que cria um objeto lista vinculada de 10 caracteres e cria um segundo objeto lista contendo uma cópia da primeira lista, mas na ordem inversa.

21.10 Escreva um programa que insere uma linha de texto e utiliza um objeto pilha para imprimir a linha invertida.

21.11 Escreva um programa que utiliza um objeto pilha para determinar se uma string é um palíndromo (isto é, a string é grafada identicamente da esquerda para a direita e da direita para a esquerda). O programa deve ignorar espaços e pontuação.

21.12 As pilhas são utilizadas por compiladores para ajudar no processo de avaliação de expressões e na geração de código de linguagem de máquina. Neste e no próximo exercício, investigamos como os compiladores avaliam expressões aritméticas que consistem apenas de constantes, operadores e parênteses.

Humanos geralmente escrevem expressões como `3 + 4` e `7 / 9` em que o operador (`+` ou `/` aqui) é escrito entre seus operandos — isso é chamado **notação infixa**. Os computadores 'preferem' **notação pós-fixa**, em que o operador é escrito à direita de seus dois operandos. As expressões infixas precedentes apareceriam na notação pós-fixa como `3 4 +` e `7 9 /`, respectivamente.

Para avaliar uma expressão infixa complexa, um compilador primeiro converteria a expressão em notação pós-fixa e então avaliaria a versão da expressão pós-fixa. Cada um desses algoritmos requer apenas uma única passagem da esquerda para a direita pela expressão. Cada algoritmo utiliza um objeto pilha em suporte à sua operação, e em cada algoritmo a pilha é utilizada para um propósito diferente.

Neste exercício, você escreverá uma versão C++ do algoritmo de conversão de infixo para pós-fixo. No próximo exercício, você escreverá uma versão C++ do algoritmo de avaliação de expressão pós-fixa. Mais adiante no capítulo, você descobrirá que o código que você escreve neste exercício pode ajudá-lo a implementar um compilador funcional completo.

Escreva um programa que converte uma expressão aritmética infixa comum (suponha que uma expressão válida é inserida) com inteiros de único dígito como

```
(6 + 2) * 5 - 8 / 4
```

para uma expressão pós-fixa. A versão pós-fixa da expressão infixa precedente é

```
6 2 + 5 * 8 4 / -
```

O programa deve ler a expressão no array de caracteres `infix` e utilizar as versões modificadas das funções de pilha implementadas neste capítulo para ajudar a criar a expressão pós-fixa no array de caracteres `postfix`. O algoritmo para criar uma expressão pós-fixa é o seguinte:

1) Insira um parêntese esquerdo `'('` na pilha.
2) Acrescente um parêntese direito `')'` ao final de `infix`.
3) Enquanto a pilha não estiver vazia, leia `infix` da esquerda para a direita e faça o seguinte:

   Se o caractere atual no `infix` for um dígito, copie-o para o próximo elemento de `postfix`.
   Se o caractere atual em `infix` for um parêntese esquerdo, adicione-o à pilha.
   Se o caractere atual no `infix` for um operador,
      Remova os operadores (se houver algum) a partir da parte superior da pilha, embora eles tenham precedência igual ou maior que a do operador atual e insira os operadores removidos em `postfix`.
      Adicione o caractere atual a `infix` na pilha.
   Se o caractere atual no `infix` for um parêntese direito,
      Remova os operadores a partir da parte superior da pilha e os insira em `postfix` até que um parêntese esquerdo esteja na parte superior da pilha.
      Remova (e descarte) o parêntese esquerdo da pilha.

As seguintes operações aritméticas são permitidas em uma expressão:

`+` adição
`-` subtração
`*` multiplicação
`/` divisão
`^` exponenciação
`%` módulo

[*Nota:* Para o propósito deste exercício, supomos a associatividade da esquerda para a direita para todos os operadores.] A pilha deve ser mantida com os nós de pilha, cada nó contendo um membro de dados e um ponteiro para o próximo nó de pilha.

Algumas capacidades funcionais que você pode querer fornecer são:

a) A função `convertToPostfix`, que converte a expressão infixa em notação pós-fixa.
b) A função `isOperator`, que determina se `c` é um operador.
c) A função `precedence`, que determina se a precedência de `operator1` é menor que, igual a ou maior que a precedência de `operator2` (a função retorna –1, 0 e 1, respectivamente).
d) A função `push`, que insere um valor na pilha.
e) A função `pop`, que remove um valor da pilha.
f) A função `stackTop`, que retorna o valor superior da pilha sem remover a pilha.
g) A função `isEmpty`, que determina se a pilha está vazia.
h) A função `printStack`, que imprime a pilha.

21.13 Escreva um programa que avalia uma expressão pós-fixa (pressuponha que ela seja válida) como

```
6 2 + 5 * 8 4 / -
```

O programa deve ler uma expressão pós-fixa consistindo em dígitos e operadores em um array de caracteres. Utilizando versões modificadas das funções de pilha implementadas anteriormente neste capítulo, o programa deve varrer a expressão e avaliá-la. O algoritmo é como segue:

1) Acrescente o caractere nulo (`'\0'`) ao fim da expressão pós-fixa. Quando o caractere nulo é encontrado, não é necessário mais nenhum processamento.
2) Enquanto `'\0'` não foi encontrado, leia a expressão da esquerda para a direita.

   Se o caractere atual for um dígito,
      Insira seu valor inteiro na pilha (o valor de inteiro de um caractere de dígito é seu valor no conjunto de caracteres do computador menos o valor de `'0'` no conjunto de caracteres do computador).
   Caso contrário, se o caractere atual for um *operador*,
      Remova os dois elementos superiores da pilha para variáveis `x` e `y`.
      Calcule `y` *operador* `x`.
      Adicione o resultado do cálculo à pilha.

3) Quando o caractere nulo for encontrado na expressão, remova o valor superior da pilha. Esse é o resultado da expressão pós-fixa.

[*Nota:* No Passo *2* citado, se o operador for `'/'`, a parte superior da pilha é `2` e o próximo elemento na pilha é `8`, então remova `2` em `x`, remova `8` em `y`, avalie `8 / 2` e insira o resultado, `4`, de volta na pilha. Essa nota também se aplica ao operador `'–'`.] As operações aritméticas permitidas em uma expressão são

+ adição
– subtração
* multiplicação
/ divisão
^ exponenciação
% módulo

[*Nota:* Para o propósito deste exercício, supomos a associatividade da esquerda para a direita para todos os operadores.] A pilha deve ser mantida com nós de pilha que contêm um membro de dados `int` e um ponteiro para o próximo nó da pilha. Você pode querer fornecer as seguintes capacidades funcionais:

a) A função `evaluatePostfixExpression`, que avalia a expressão pós-fixa.
b) A função `calculate`, que avalia a expressão `op1 operator op2`.
c) A função `push`, que insere um valor na pilha.
d) A função `pop`, que remove um valor da pilha.
e) A função `isEmpty`, que determina se a pilha está vazia.
f) A função `printStack`, que imprime a pilha.

21.14 Modifique o programa avaliador de pós-fixo do Exercício 21.13 de modo que ele possa processar os operandos de inteiro maiores que 9.

21.15 (*Simulação de supermercado*) Escreva um programa que simula uma fila de caixa em um supermercado. A fila é um objeto fila. Os clientes (isto é, objetos-cliente) chegam em intervalos de inteiro aleatório de 1–4 minutos. Além disso, cada cliente é atendido em intervalos de inteiro aleatório de 1–4 minutos. Obviamente, as taxas precisam ser equilibradas. Se a taxa média de chegada for maior que a taxa média de atendimento, a fila crescerá infinitamente. Mesmo com taxas 'equilibradas', a aleatoriedade ainda pode provocar filas longas. Execute a simulação de supermercado para um dia de 12 horas (720 minutos) utilizando o seguinte algoritmo:

1) Escolha um inteiro aleatório entre 1 e 4 para determinar o minuto em que o primeiro cliente chega.
2) Na hora de chegada do primeiro cliente:

    Determine o tempo de atendimento do serviço ao cliente (inteiro aleatório de 1 a 4).
    Comece atendendo o cliente.
    Agende a hora de chegada do próximo cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).

3) Para cada minuto do dia:

    Se o próximo cliente chegar,
        Expresse isso.
        Enfileire o cliente.
        Agende a hora de chegada do próximo cliente.
    Se o serviço foi completado para o último cliente,
        Expresse isso.
        Desenfileire o próximo cliente a ser atendido.
        Determine o tempo de conclusão do serviço de atendimento ao cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).

Agora execute sua simulação para 720 minutos e responda a cada uma das seguintes questões:

a) Qual é o número máximo de clientes na fila a qualquer hora?
b) Qual é a espera mais longa que qualquer cliente experimenta?
c) O que acontece se o intervalo de chegada é mudado de 1–4 minutos para 1–3 minutos?

21.16 Modifique o programa das figuras 21.20–21.22 para permitir que o objeto árvore binária contenha duplicatas.

21.17 Escreva um programa baseado nas figuras 21.20–21.22 que insira uma linha de texto, tokenize a frase em palavras separadas (você pode querer utilizar a função de biblioteca `strtok`), insira as palavras em uma árvore de pesquisa binária e imprima os percursos na ordem, pré-ordem e pós-ordem da árvore. Utilize uma abordagem OOP.

21.18 Neste capítulo, vimos que a eliminação de duplicatas é simples e direta quando se cria uma árvore de pesquisa binária. Descreva como você realizaria a eliminação de duplicatas utilizando apenas um array unidimensional. Compare o desempenho da eliminação de duplicatas baseada em array com o desempenho da eliminação de duplicatas baseada na pesquisa de árvore binária.

21.19 Escreva uma função `depth` que recebe uma árvore binária e determine quantos níveis ela tem.

21.20 (*Imprimir recursivamente uma lista de trás para a frente*) Escreva uma função-membro `printListBackward` que recursivamente gera saída dos itens em um objeto lista vinculada na ordem inversa. Escreva um programa de teste que cria uma lista classificada de inteiros e imprime a lista em ordem inversa.

21.21 (*Pesquisar recursivamente uma lista*) Escreva uma função-membro `searchList` que pesquisa recursivamente um valor especificado em um objeto lista vinculada. A função deve retornar um ponteiro para o valor se ele for localizado; caso contrário, nulo deve ser retornado.

Utilize sua função em um programa de teste que cria uma lista de inteiros. O programa deve solicitar ao usuário um valor para localizar na lista.

21.22 (*Exclusão de árvore binária*) Neste exercício, discutimos a exclusão de itens de árvores de pesquisa binária. O algoritmo de exclusão não é tão simples e direto quanto o algoritmo de inserção. Há três casos que são encontrados ao excluir um item — o item está contido em um nó-folha (isto é, não tem filhos), o item está contido em um nó que tem um filho ou o item está contido em um nó que tem dois filhos.

Se o item a ser excluído está contido em um nó-folha, o nó é excluído e o ponteiro no nó-pai é configurado como nulo.

Se o item a ser excluído está contido em um nó com um filho, o ponteiro no nó-pai é configurado para apontar para o nó-filho, e o nó contendo o item de dados é excluído. Isso faz com que o nó-filho tome o lugar do nó excluído na árvore.

O último caso é o mais difícil. Quando um nó com dois filhos é excluído, outro nó na árvore deve tomar seu lugar. Entretanto, o ponteiro no nó-pai não pode ser atribuído para apontar para um dos filhos do nó a ser excluído. Na maioria dos casos, a árvore de pesquisa binária resultante não obedeceria à seguinte característica das árvores de pesquisa binária (sem valores duplicados): *os valores em qualquer subárvore esquerda são menores que o valor no nó-pai, e os valores em qualquer subárvore direita são maiores que o valor no nó-pai*.

Qual é o nó utilizado como um *nó substituto* para manter essa característica? O nó contendo o maior valor na árvore menor que o valor no nó sendo excluído, ou o nó contendo o menor valor na árvore maior que o valor no nó sendo excluído? Vamos considerar o nó com o menor valor. Em uma árvore de pesquisa binária, o maior valor menor que um valor do pai encontra-se na subárvore esquerda do nó-pai e seguramente estará contido no nó mais à direita da subárvore. Esse nó é encontrado descendendo a subárvore esquerda pela direita até que o ponteiro para o filho direito do nó atual seja nulo. Agora estamos apontando para o nó substituto que é um nó-folha ou um nó com um filho à sua esquerda. Se o nó substituto for um nó-folha, os passos para realizar a exclusão são os seguintes:

1) Armazene o ponteiro para o nó a ser excluído em uma variável de ponteiro temporária (esse ponteiro é utilizado para excluir a memória dinamicamente alocada).
2) Configure o ponteiro no pai do nó sendo excluído para apontar para o nó substituto.
3) Configure o ponteiro no pai do nó substituto como nulo.
4) Configure o ponteiro como a subárvore direita no nó substituto para apontar para a subárvore direita do nó a ser excluído.
5) Exclua o nó para o qual a variável ponteiro temporária aponta.

Os passos de exclusão para um nó substituto com um filho esquerdo são semelhantes àqueles para um nó substituto sem filhos, mas o algoritmo também deve mover o filho para a posição do nó substituto na árvore. Se o nó substituto for um nó com um filho esquerdo, os passos para realizar a exclusão são como segue:

1) Armazene o ponteiro para o nó a ser excluído em uma variável ponteiro temporária.
2) Configure o ponteiro no pai do nó sendo excluído para apontar para o nó substituto.
3) Configure o ponteiro no pai do nó substituto para apontar para o filho esquerdo do nó substituto.
4) Configure o ponteiro como a subárvore direita no nó substituto para apontar para a subárvore direita do nó a ser excluído.
5) Exclua o nó para o qual a variável ponteiro temporária aponta.

Escreva a função-membro `deleteNode`, que aceita como seus argumentos um ponteiro para o nó-raiz do objeto-árvore e o valor a ser excluído. A função deve localizar na árvore o nó contendo o valor a ser excluído e utilizar os algoritmos discutidos aqui para excluir o nó. A função deve imprimir uma mensagem que indica se o valor foi excluído. Modifique o programa das figuras 21.20–21.22 para utilizar essa função. Depois de excluir um item, chame as funções de percurso `inOrder`, `preOrder` e `postOrder` para confirmar se a operação de exclusão foi realizada corretamente.

21.23 (*Pesquisa de árvore binária*) Escreva a função-membro `binaryTreeSearch`, que tenta localizar um valor especificado em um objeto-árvore de pesquisa binária. A função deve aceitar como argumentos um ponteiro para o nó-raiz da árvore binária e uma chave de pesquisa a ser localizada. Se o nó contendo a chave de pesquisa for localizado, a função deve retornar um ponteiro para aquele nó; caso contrário, a função deve retornar um ponteiro nulo.

21.24 (*Percurso de árvore binária na ordem de nível*) O programa das figuras 21.20–21.22 ilustrou três métodos recursivos de percorrer uma árvore binária — os percursos na ordem, pré-ordem e pós-ordem. Este exercício apresenta o percurso *na ordem de nível* de uma árvore binária, no qual os valores de nó são impressos nível por nível iniciando no nível do nó-raiz. Os nós em cada nível são impressos da esquerda para a direita. O percurso na ordem de nível não é um algoritmo recursivo. Ela utiliza um objeto fila para controlar a saída dos nós. O algoritmo é como segue:

1) Insira o nó-raiz na fila.
2) Enquanto houver nós esquerdos na fila,

    Obtenha o próximo nó na fila.
    Imprima o valor do nó.
    Se o ponteiro para o filho esquerdo do nó não for nulo,
        Insira o nó-filho esquerdo na fila.
    Se o ponteiro para o filho direito do nó não for nulo,
        Insira o nó-filho direito na fila.

Escreva a função-membro `levelOrder` para realizar um percurso na ordem de nível de um objeto árvore binária. Modifique o programa das figuras 21.20–21.22 para utilizar essa função. [*Nota:* Você também precisará modificar e incorporar as funções de processamento fila da Figura 21.16 neste programa.]

21.25 (*Imprimindo árvores*) Escreva uma função-membro recursiva `outputTree` para exibir um objeto árvore binária na tela. A função deve gerar saída da árvore linha por linha com a parte superior da árvore na parte esquerda da tela e a parte inferior da árvore em direção à parte direita da tela. Cada linha é enviada para a saída verticalmente. Por exemplo, a saída da árvore binária ilustrada na Figura 21.24 é realizada como mostrado a seguir:

```
                99
            97
                92
        83
                72
            71
                69
49
                44
            40
                32
        28
                19
            18
                11
```

Observe que o nó mais à direita da folha aparece na parte superior da saída na coluna mais à direita e o nó-raiz aparece à esquerda da saída. Cada coluna de saída inicia cinco espaços à direita da coluna anterior. A função `outputTree` deve receber um argumento `totalSpaces` representando o número de espaços que precedem o valor a ser enviado para a saída (essa variável deve iniciar em zero de modo que o nó-raiz seja enviado para a saída na parte esquerda da tela). A função utiliza um percurso na ordem modificado para gerar a saída da árvore — ela inicia no nó mais à direita na árvore e segue para a esquerda. O algoritmo é como segue:

Enquanto o ponteiro para o nó atual não é nulo,
  Chame recursivamente `outputTree` com a subárvore direita do nó atual e
    `totalSpaces` + 5
  Utilize uma estrutura `for` para contar de 1 a `totalSpaces` e gerar saída de espaços.
  Envie para a saída o valor no nó atual.
  Configure o ponteiro como o nó atual para apontar para a subárvore esquerda do nó atual.
  Incremente `totalSpaces` por 5.

Seção especial: construindo seu próprio compilador

Nos exercícios 8.18 e 8.19, introduzimos a Simpletron Machine Language (SML) e você implementou um simulador de computador Simpletron para executar programas escritos em SML. Nesta seção, construímos um compilador que converte programas escritos em uma linguagem de programação de alto nível em SML. Esta seção 'amarra' o processo de programação inteiro. Você irá escrever os programas nessa nova linguagem de alto nível, compilar esses programas no compilador construído e executá-los no simulador que construiu no Exercício 8.19. Você deve se esforçar o máximo possível para implementar seu compilador de uma maneira orientada a objetos.

21.26 (*A linguagem Simple*) Antes de iniciarmos a construção do compilador, discutimos uma linguagem simples, mas ainda poderosa e de alto nível, semelhante a versões anteriores da popular linguagem BASIC. Chamamos essa linguagem de Simple. Cada *instrução* (*statement*) Simple consiste em um *número de linha* e uma *instrução* (*instruction*) Simple propriamente dita. Os números da linha devem aparecer em ordem crescente. Cada instrução inicia com um dos seguintes *comandos* Simple: `rem`, `input`, `let`, `print`, `goto`, `if...goto` e `end` (ver Figura 21.25). Todos os comandos, exceto `end`, podem ser utilizados repetidamente. O Simple avalia apenas as expressões de inteiro que utilizam os operadores `+`, `-`, `*` e `/`. Esses operadores têm a mesma precedência em C++. Os parênteses podem ser utilizados para alterar a ordem de avaliação de uma expressão.

Nosso compilador de Simple reconhece apenas letras minúsculas. Todos os caracteres em um arquivo Simple devem estar em letras minúsculas (letras maiúsculas resultam em um erro de sintaxe, a menos que apareçam em uma instrução `rem`, caso em que são ignoradas). Um nome de variável tem uma única letra. A linguagem Simple não permite nomes de variáveis descritivos, portanto as variáveis devem ser explicadas em observações para indicar sua utilização em um programa. O Simple utiliza apenas variáveis de inteiro. O Simple não tem declarações de variável — a mera menção a um nome de variável em um programa faz com que a variável seja declarada e inicializada

Comando	Instrução de exemplo	Descrição
`rem`	`50 rem this is a remark`	O texto que se segue a `rem` é para fins de documentação e é ignorado pelo compilador.
`input`	`30 input x`	Exibe um ponto de interrogação para pedir que o usuário insira um inteiro. Lê esse inteiro a partir do teclado e o armazena em `x`.
`let`	`80 let u = 4 * (j - 56)`	Atribui a `u` o valor de `4 * (j - 56)`. Observe que uma expressão arbitrariamente complexa pode aparecer à direita do sinal de igual.
`print`	`10 print w`	Exibe o valor de `w`.
`goto`	`70 goto 45`	Transfere o controle do programa para a linha `45`.
`if...goto`	`35 if i == z goto 80`	Compara `i` e `z` quanto à igualdade e transfere o controle para a linha `80` se a condição for verdadeira; caso contrário, continua a execução com a próxima instrução.
`end`	`99 end`	Termina execução do programa.

Figura 21.25 Comandos do Simple.

como zero automaticamente. A sintaxe do Simple não permite manipulação de string (ler uma string, gravar uma string, comparar strings etc.) Se uma string for encontrada em um programa Simple (após qualquer outro comando que não `rem`), o compilador gera um erro de sintaxe. A primeira versão de nosso compilador supõe que os programas Simple são digitados corretamente. O Exercício 21.29 pede para o aluno modificar o compilador para realizar verificação de erros de sintaxe.

O Simple utiliza a instrução condicional `if...goto` e a instrução incondicional `goto` para alterar o fluxo de controle durante a execução do programa. Se a condição na instrução `if...goto` for verdadeira, o controle é transferido para uma linha específica do programa. Os seguintes operadores relacionais de igualdade são válidos em uma instrução `if...goto`: `<`, `>`, `<=`, `>=`, `==` e `!=`. A precedência desses operadores é a mesma que em C++.

Vamos agora considerar vários programas que demonstram recursos do Simple. O primeiro programa (Figura 21.26) lê dois inteiros do teclado, armazena os valores nas variáveis `a` e `b` e calcula e imprime sua soma (armazenada na variável `c`).

O programa da Figura 21.27 determina e imprime o maior de dois inteiros. Os inteiros são inseridos do teclado e armazenados em `s` e `t`. A instrução `if...goto` testa a condição `s >= t`. Se a condição for verdadeira, o controle é transferido para a linha `90`, e `s` é enviado para a saída; caso contrário, `t` é enviado para a saída, e o controle é transferido para a instrução `end` na linha `99`, onde o programa termina.

O Simple não fornece uma instrução de repetição (como `for`, `while` ou `do...while` do C++). Entretanto, o Simple pode simular cada uma das instruções de repetição do C++ utilizando as instruções `if...goto` e `goto`. A Figura 21.28 utiliza um loop controlado por sentinela para calcular os quadrados de vários inteiros. Cada inteiro é inserido do teclado e armazenado na variável `j`. Se o valor inserido é o valor de sentinela `-9999`, o controle é transferido para a linha `99`, onde o programa termina. Caso contrário, `k` é atribuído ao quadrado de `j`, `k` é enviado para a saída de tela e o controle é passado para a linha `20`, onde o próximo inteiro é inserido.

Utilizando os programas de exemplo das figuras 21.26, 21.27 e 21.28 como seu guia, escreva um programa Simple para realizar cada uma das seguintes instruções:

a) Insira três inteiros, determine sua média e imprima o resultado.
b) Utilize um loop controlado por sentinela para inserir 10 inteiros e calcular e imprimir sua soma.
c) Utilize um loop controlado por contador para inserir sete inteiros, alguns positivos e alguns negativos, e calcule e imprima sua média.
d) Insira uma série de inteiros e determine e imprima o maior. A primeira entrada de inteiro indica quantos números devem ser processados.
e) Insira 10 inteiros e imprima o menor.
f) Calcule e imprima a soma dos inteiros pares de 2 a 30.
g) Calcule e imprima o produto dos inteiros ímpares de 1 a 9.

21.27 (*Construindo um compilador; Pré-requisito: Completar os exercícios 8.18, 8.19, 21.12, 21.13 e 21.26*) Agora que a linguagem Simple foi apresentada (Exercício 21.26), discutiremos como construir um compilador de Simple. Primeiro, consideramos o processo pelo qual um programa Simple é convertido em SML e executado pelo Simpletron Simulator (ver Figura 21.29). Um arquivo contendo um programa Simple é lido pelo compilador e convertido em código SML. O código SML é enviado para um arquivo de saída em disco, no qual instruções de SML aparecem uma em cada linha. O arquivo SML então é carregado no Simpletron Simulator e os resultados são enviados para

```
 1  10 rem   determine and print the sum of two integers
 2  15 rem
 3  20 rem   input the two integers
 4  30 input a
 5  40 input b
 6  45 rem
 7  50 rem   add integers and store result in c
 8  60 let c = a + b
 9  65 rem
10  70 rem   print the result
11  80 print c
12  90 rem   terminate program execution
13  99 end
```

Figura 21.26 Programa em Simple que determina a soma de dois inteiros.

```
 1  10 rem   determine the larger of two integers
 2  20 input s
 3  30 input t
 4  32 rem
 5  35 rem   test if s >= t
 6  40 if s >= t goto 90
 7  45 rem
 8  50 rem   t is greater than s, so print t
 9  60 print t
10  70 goto 99
11  75 rem
12  80 rem   s is greater than or equal to t, so print s
13  90 print s
14  99 end
```

Figura 21.27 Programa em Simple que localiza o maior de dois inteiros.

```
 1  10 rem   calculate the squares of several integers
 2  20 input j
 3  23 rem
 4  25 rem   test for sentinel value
 5  30 if j == -9999 goto 99
 6  33 rem
 7  35 rem   calculate square of j and assign result to k
 8  40 let k = j * j
 9  50 print k
10  53 rem
11  55 rem   loop to get next j
12  60 goto 20
13  99 end
```

Figura 21.28 Calcule os quadrados de vários inteiros.

um arquivo em disco e para a tela. Observe que o programa Simpletron desenvolvido no Exercício 8.19 pegou sua entrada do teclado. Ele deve ser modificado para ler de um arquivo de modo que possa executar os programas produzidos pelo nosso compilador.

O compilador de Simple realiza duas *passagens* do programa Simple para convertê-lo em SML. A primeira passagem constrói uma *tabela de símbolos* (objeto) em que cada *número de linha* (objeto), *nome de variável* (objeto) e *constante* (objeto) do programa Simple é armazenado com seu tipo e posição correspondente no código SML final (a tabela de símbolos é discutida em detalhe a seguir). A primeira passagem também produz o(s) correspondente(s) objeto(s) de instrução (*instruction*) de SML para cada uma das instruções (*statements*) do Simple (objeto etc.). Como veremos, se o programa Simple contém instruções que transferem o controle posteriormente para uma linha no programa, a primeira passagem resulta em um programa SML contendo parte das instruções 'não concluídas'. A segunda passagem do compilador localiza e completa as instruções não finalizadas e envia o programa SML para um arquivo de saída.

Primeira passagem

O compilador começa a ler uma instrução (*statement*) do programa Simple na memória. A linha deve ser separada em seus *tokens* individuais (isto é, 'fragmentos' de uma instrução) para processamento e compilação (a função `strtok` da biblioteca-padrão pode ser utilizada para facilitar essa tarefa). Lembre-se de que cada instrução (*statement*) inicia com um número de linha seguido por um comando. Quando o compilador divide uma instrução (*statement*) em tokens, se o token é um número de linha, uma variável ou uma constante, ele é colocado na tabela de símbolos. Um número de linha é colocado na tabela de símbolos apenas se for o primeiro token em uma instrução. O objeto `symbolTable` é um array de objetos `tableEntry` representando cada símbolo no programa. Não há restrição quanto ao número de símbolos que podem aparecer no programa. Portanto, a `symbolTable` para um programa particular pode ser grande. Torne a `symbolTable` um array de 100 elementos por enquanto. Você pode aumentar ou diminuir seu tamanho depois que o programa estiver funcionando.

Cada objeto `tableEntry` contém três membros. O membro `symbol` é um inteiro contendo a representação ASCII de uma variável (lembre-se de que os nomes de variável são caracteres únicos), um número de linha ou uma constante. O membro `type` é um dos seguintes caracteres que indicam o tipo do símbolo: `'C'` para constante, `'L'` número de linha e `'V'` para variável. O membro `location` contém a posição da memória do Simpletron (`00` para `99`) que o símbolo referencia. A memória do Simpletron é um array de 100 inteiros em que as instruções de SML e os dados são armazenados. Para um número de linha, a posição é o elemento no array de memória do Simpletron em que as instruções de SML para a instrução Simple iniciam. Para uma variável ou constante, a posição é o elemento no array de memória de Simpletron em que a variável ou constante é armazenada. Variáveis e constantes são alocadas a partir do fim da memória do Simpletron para trás. A primeira variável ou constante é armazenada na posição em `99`, a próxima na posição em `98` etc.

A tabela de símbolos desempenha uma parte integrante na conversão de programas Simple em SML. Aprendemos no Capítulo 8 que uma instrução SML é um inteiro de quatro dígitos compostos de duas partes — o *código de operação* e o *operando*. O código de operação é determinado por comandos em Simple. Por exemplo, o comando `input` do Simple corresponde ao código de operação `10` (*read*) do SML e o comando `print` do Simple corresponde ao código de operação `11` (*write*) de SML. O operando é uma posição da memória contendo os dados em que o código de operação realiza sua tarefa (por exemplo, o código de operação `10` lê um valor do teclado e o armazena na posição da memória especificada pelo operando). O compilador pesquisa `symbolTable` para determinar a posição da memória no Simpletron para cada símbolo, de modo que a posição correspondente possa ser utilizada para completar as instruções de SML.

A compilação de cada instrução do Simple é baseada em seu comando. Por exemplo, depois que o número de linha em uma instrução `rem` é inserido na tabela de símbolos, o restante da instrução é ignorado pelo compilador porque uma observação serve apenas para propósitos de documentação. As instruções `input`, `print`, `goto` e `end` correspondem às instruções do SML *read*, *write*, *branch* ([desviar] para uma posição específica) e *halt*. As instruções contendo esses comandos do Simple são convertidas diretamente em SML (note que uma instrução `goto` pode conter uma referência não resolvida se o número de linha especificado referencia uma instrução mais adiante no arquivo do programa Simple; isso é às vezes chamado de referência antecipada).

Quando uma instrução `goto` é compilada com uma referência não resolvida, a instrução de SML deve ser *marcada com um flag* para indicar que a segunda passagem do compilador deve completar a instrução. Os flags são armazenados em `flags` de array de 100 elemen-

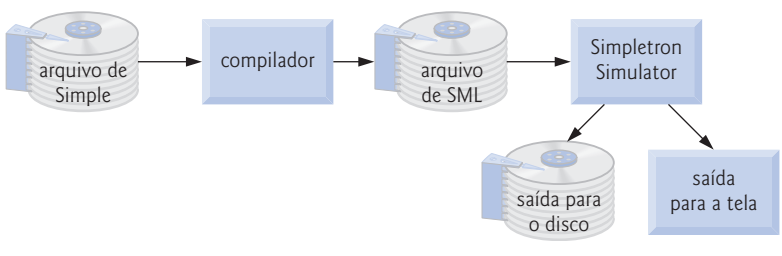

Figura 21.29 Gravando, compilando e executando um programa da linguagem Simple.

tos do tipo `int` em que cada elemento é inicializado como `-1`. Se a posição da memória a que um número de linha no programa Simple se refere ainda não é conhecida (isto é, não está na tabela de símbolos), o número de linha é armazenado em `flags` array no elemento com o mesmo subscrito que a instrução incompleta. O operando da instrução incompleta é configurado como `00` temporariamente. Por exemplo, uma instrução de desvio incondicional (*unconditional branch*, fazer uma referência antecipada) é deixada como `+4000` até a segunda passagem do compilador. A segunda passagem do compilador será descrita em breve.

A compilação das instruções `if...goto` e `let` é mais complicada que outras instruções — elas são as únicas instruções que produzem mais de uma instrução SML. Para um `if...goto`, o compilador produz código para testar a condição e desviar para outra linha se necessário. O resultado do desvio pode ser uma referência não resolvida. Cada um dos operadores relacionais de igualdade pode ser simulado utilizando instruções de *branch zero* ou *branch negative* do SML (ou uma combinação das duas).

Para uma instrução `let`, o compilador produz código para avaliar uma expressão aritmética arbitrariamente complexa consistindo em variáveis e/ou constantes de inteiro. As expressões devem separar cada operando e operador com espaços. Os exercícios 21.12 e 21.13 apresentaram o algoritmo de conversão de infixo em pós-fixo e o algoritmo de avaliação pós-fixa utilizado por compiladores para avaliar expressões. Antes de prosseguir com seu compilador, você deve completar cada um desses exercícios. Quando um compilador encontra uma expressão, ele converte a expressão de notação infixa em notação pós-fixa, e então avalia a expressão pós-fixa.

Como o compilador produz a linguagem de máquina para avaliar uma expressão contendo variáveis? O algoritmo de avaliação de pós-fixo contém um 'gancho' onde o compilador pode gerar instruções de SML em vez de realmente avaliar a expressão. Para ativar esse 'gancho' no compilador, o algoritmo de avaliação pós-fixo deve ser modificado para pesquisar a tabela de símbolos para cada símbolo que ele encontra (e possivelmente inseri-lo), determinar a posição da memória correspondente do símbolo e *adicionar a posição da memória na pilha* (*em vez do símbolo*). Quando um operador é encontrado na expressão pós-fixa, as duas posições da memória na parte superior da pilha são removidas e a linguagem de máquina para afetar a operação é produzida, utilizando as posições da memória como operandos. O resultado de cada subexpressão é armazenado em uma posição temporária na memória e inserido de volta na pilha de tal modo que a avaliação da expressão pós-fixa possa continuar. Quando a avaliação pós-fixa está completa, a posição da memória contendo o resultado é a única posição deixada na pilha. Essa é removida e as instruções de SML são geradas para atribuir o resultado à variável à esquerda da instrução `let`.

Segunda passagem

A segunda passagem do compilador realiza duas tarefas: resolver quaisquer referências não resolvidas e enviar o código de SML para um arquivo de saída. A solução de referências ocorre como segue:

a) Pesquise no array `flags` uma referência não resolvida (isto é, um elemento com um valor outro que `-1`).
b) Localize o objeto no array `symbolTable` contendo o símbolo armazenado no array `flags` (certificando-se de que o tipo do símbolo é `'L'` para o número de linha).
c) Insira a posição de memória do membro `location` na instrução com a referência não resolvida (lembre-se de que uma instrução contendo uma referência não resolvida tem operando `00`).
d) Repita os Passos 1, 2 e 3 até o final do array `flags` ser alcançado.

Depois que o processo de solução estiver completo, o array inteiro contendo o código de SML é enviado para um arquivo de saída em disco com uma instrução de SML por linha. Esse arquivo pode ser lido pelo Simpletron para execução (depois que o simulador é modificado para ler sua entrada de um arquivo). Compilar seu primeiro programa Simple em um arquivo de SML e então executar esse arquivo deve lhe dar uma verdadeira sensação de realização pessoal.

Um exemplo completo

O seguinte exemplo ilustra a conversão completa de um programa Simple em SML como ele será realizado pelo compilador de Simple. Considere um programa Simple que insere um inteiro e soma os valores de 1 até esse inteiro. O programa e as instruções de SML produzidas pela primeira passagem do compilador de Simple são ilustrados na Figura 21.30. A tabela de símbolos construída pela primeira passagem é mostrada na Figura 21.31.

A maioria das instruções do Simple é convertida diretamente em instruções únicas de SML. As exceções nesse programa são os comentários, a instrução `if...goto` na linha `20` e as instruções `let`. Os comentários não são traduzidos para a linguagem de máquina. Entretanto, o número de linha para uma observação é colocado na tabela de símbolos no caso de o número de linha ser referenciado em uma instrução `goto` ou em uma instrução `if...goto`. A linha `20` do programa especifica que, se a condição `y == x` for verdadeira, o controle do programa é transferido para a linha `60`. Como a linha `60` aparece mais adiante no programa, a primeira passagem do compilador ainda não colocou `60` na tabela de símbolos (os números de linha de instrução são colocados na tabela de símbolos apenas quando aparecem como o primeiro token em uma instrução). Portanto, não é possível nesse momento determinar o operando da instrução *branch zero* do SML na posição `03` no array de instruções de SML. O compilador coloca `60` na posição `03` do array `flags` para indicar que a segunda passagem completa essa instrução.

Devemos monitorar a próxima posição da instrução no array SML porque não há uma correspondência um para um entre as instruções de Simple e as instruções de SML. Por exemplo, a instrução `if...goto` da linha `20` é compilada em três instruções de SML. Toda vez que uma instrução é produzida, devemos incrementar o *contador de instrução* para a próxima posição no array de SML. Observe que o

Programa Simple	Posição & instrução SML	Descrição
`5 rem sum 1 to x`	*nenhuma*	`rem` ignorado
`10 input x`	00 +1099	lê `x` na posição 99
`15 rem check y == x`	*nenhuma*	`rem` ignorado
`20 if y == x goto 60`	01 +2098	carrega `y` (98) no acumulador
	02 +3199	subtrai `x` (99) a partir do acumulador
	03 +4200	desvia se zero para posição não resolvida
`25 rem increment y`	*nenhuma*	`rem` ignorado
`30 let y = y + 1`	04 +2098	carrega `y` no acumulador
	05 +3097	adiciona 1 (97) ao acumulador
	06 +2196	armazena na localização temporária 96
	07 +2096	carrega a partir da posição temporária 96
	08 +2198	armazena acumulador em `y`
`35 rem add y to total`	*nenhuma*	`rem` ignorado
`40 let t = t + y`	09 +2095	carrega `t` (95) no acumulador
	10 +3098	adiciona `y` ao acumulador
	11 +2194	armazena na localização temporária 94
	12 +2094	carrega a partir da posição temporária 94
	13 +2195	armazena acumulador em `t`
`45 rem loop y`	*nenhuma*	`rem` ignorado
`50 goto 20`	14 +4001	desvia para a posição 01
`55 rem output result`	*nenhuma*	`rem` ignorado
`60 print t`	15 +1195	saída `t` para a tela
`99 end`	16 +4300	termina a execução

Figura 21.30 Instruções de SML produzidas depois da primeira passagem do compilador.

Símbolo	Tipo	Posição	Símbolo	Tipo	Posição
5	L	00	35	L	09
10	L	00	40	L	09
'x'	V	99	't'	V	95
15	L	01	45	L	14
20	L	01	50	L	14
'y'	V	98	55	L	15
25	L	04	60	L	15
30	L	04	99	L	16
1	C	97			

Figura 21.31 A tabela de símbolos para programa da Figura 21.30.

tamanho da memória do Simpletron pode apresentar um problema para programas Simple com muitas instruções, variáveis e constantes. É concebível que o compilador fique sem memória. Para testar esse caso, seu programa deve conter um *contador de dados* para monitorar a posição em que a próxima variável ou constante será armazenada no array de SML. Se o valor do contador de instrução for maior que o valor do contador de dados, o array de SML está cheio. Nesse caso, o processo de compilação deve terminar e o compilador deve imprimir uma mensagem de erro indicando que ele ficou sem memória durante a compilação. Isso serve para enfatizar que, embora o programador seja liberado do peso de gerenciar a memória pelo compilador, o próprio compilador deve cuidadosamente determinar a colocação de instruções e dados na memória e verificar tais erros quando a memória se esgota durante o processo de compilação.

Uma visualização passo a passo do processo de compilação

Vamos agora percorrer o processo de compilação para o programa Simple na Figura 21.30. O compilador lê a primeira linha do programa

```
5 rem sum 1 to x
```

para a memória. O primeiro token na instrução (o número da linha) é determinado utilizando `strtok` (ver os capítulos 8 e 21 para uma discussão sobre as funções de manipulação de strings no estilo C do C++). O token retornado por `strtok` é convertido em um inteiro que utiliza `atoi`, portanto o símbolo `5` pode ser localizado na tabela de símbolos. Se o símbolo não for localizado, ele é inserido na tabela de símbolos. Já que estamos no começo do programa e essa é a primeira linha, nenhum símbolo ainda está na tabela. Então, `5` é inserido na tabela de símbolos como o tipo `L` (número da linha) e atribuído à primeira posição no array de SML (`00`). Embora essa linha seja uma observação, um espaço na tabela de símbolos ainda é alocado para o número de linha (no caso, ele é referenciado por um `goto` ou um `if...goto`). Nenhuma instrução de SML é gerada para uma instrução `rem`, então o contador de instrução não é incrementado.

A instrução

```
10 input x
```

é 'tokenizada' (separada em tokens, ou marcas) a seguir. O número de linha `10` é colocado na tabela de símbolos como o tipo `L` e atribuído à primeira posição no array de SML (`00`, porque um comentário foi iniciado no programa, então o contador de instrução é atualmente `00`). O comando `input` indica que o próximo token é uma variável (apenas uma variável pode aparecer em uma instrução `input`). Como `input` corresponde diretamente a um código de operação de SML, o compilador tem de determinar a posição de `x` no array de SML. O símbolo `x` não foi encontrado na tabela de símbolos, portanto ele é inserido na tabela de símbolos como a representação ASCII de `x`, recebe o tipo `V` e é atribuído à posição `99` no array SML (o armazenamento de dados começa em `99` e é alocado de trás para a frente). O código de SML agora pode ser gerado para essa instrução. O código de operação `10` (o código de operação de leitura do SML) é multiplicado por 100 e a posição de `x` (como determinado na tabela de símbolos) é adicionada para completar a instrução. A instrução então é armazenada no array de SML na posição `00`. O contador de instrução é incrementado por 1, porque uma única instrução de SML foi produzida.

A instrução

```
15 rem   check y == x
```

é 'tokenizada' (separada em tokens, ou marcas) a seguir. A tabela de símbolos é pesquisada por número de linha `15` (que não é localizado). O número da linha é inserido como o tipo `L` e recebe a próxima posição no array, `01` (lembre-se de que as instruções `rem` não produzem código, então o contador de instruções não é incrementado).

A instrução

```
20 if y == x goto 60
```

é 'tokenizada' (separada em tokens, ou marcas) a seguir. O número da linha `20` é inserido na tabela de símbolos e o tipo dado `L` com a próxima posição no array de SML `01`. O comando `if` indica que uma condição será avaliada. A variável `y` não é localizada na tabela de símbolos, então é inserida e lhe são atribuídos o tipo `V` e a posição de SML `98`. Em seguida, instruções de SML são geradas para avaliar a condição. Como não há equivalente direto em SML a `if...goto`, ele deve ser simulado realizando um cálculo com `x` e `y` e desviando com base no resultado. Se `y` for igual a `x`, o resultado de subtrair `x` de `y` é zero, então a instrução *branch zero* pode ser utilizada com o resultado do cálculo para simular a instrução `if...goto`. O primeiro passo requer que `y` seja carregado (da posição `98` do SML) no acumulador. Isso produz a instrução `01 +2098`. Em seguida, `x` é subtraído do acumulador. Isso produz a instrução `02 +3199`. O valor no acumulador pode ser zero, positivo ou negativo. Como o operador é `==`, queremos *branch zero*. Primeiro, a tabela de símbolos é pesquisada quanto à posição do desvio (`60` nesse caso), que não é localizada. Assim, `60` é colocado no array `flags` na posição `03`, e a instrução `03 +4200` é gerada (não podemos adicionar a posição de desvio porque ainda não atribuímos uma posição à linha `60` no array de SML). O contador de instrução é incrementado para `04`.

O compilador prossegue para instrução

```
25 rem   increment y
```

O número de linha `25` é inserido na tabela de símbolos como o tipo `L` e atribuído à posição `04` da SML. O contador de instrução não é incrementado.

Quando a instrução

```
30 let y = y + 1
```

é 'tokenizada' (dividida em tokens), o número de linha 30 é inserido na tabela de símbolos como o tipo L e é atribuído à posição 04 da SML. O comando let indica que a linha é uma instrução de atribuição. Primeiro, todos os símbolos na linha são inseridos na tabela de símbolos (se ainda não estiverem aí). O inteiro 1 é adicionado à tabela de símbolos como o tipo C e é atribuído à posição 97 da SML. Em seguida, o lado direito da atribuição é convertido de notação infixa em pós-fixa. Então a expressão pós-fixa (y 1 +) é avaliada. O símbolo y é localizado na tabela de símbolos e sua posição da memória correspondente é inserida na pilha. O símbolo 1 também é localizado na tabela de símbolos e sua correspondente posição na memória é inserida na pilha. Quando o operador + é encontrado, o avaliador de pós-fixo remove da pilha o operando direito do operador, remove da pilha novamente o operando esquerdo do operador e produz as instruções de SML

```
04 +2098    (load y)
05 +3097    (add 1)
```

O resultado da expressão é armazenado em uma posição temporária na memória (96) com a instrução

```
06 +2196    (store temporary)
```

e a posição temporária é adicionada à pilha. Agora que a expressão foi avaliada, o resultado deve ser armazenado em y (isto é, a variável no lado esquerdo de =). Então, a posição temporária é carregada no acumulador e o acumulador é armazenado em y com as instruções

```
07 +2096    (load temporary)
08 +2198    (store y)
```

O leitor imediatamente notará que as instruções de SML parecem ser redundantes. Discutiremos essa questão brevemente.

Quando a instrução

```
35 rem   add y to total
```

é tokenizada (dividida em tokens), o número de linha 35 é inserido na tabela de símbolos como o tipo L e atribuído à posição 09.

A instrução

```
40 let t = t + y
```

é semelhante à linha 30. A variável t é inserida na tabela de símbolos como o tipo V e atribuída à posição 95 da SML. As instruções seguem a mesma lógica e formato que a linha 30 e as instruções 09 +2095, 10 +3098, 11 +2194, 12 +2094 e 13 +2195 são geradas. Observe que o resultado de t + y é atribuído à posição temporária 94 antes de ser atribuído a t (95). Mais uma vez, o leitor notará que as instruções nas posições da memória 11 e 12 parecem ser redundantes. Novamente, discutiremos isso em breve.

A instrução

```
45 rem   loop y
```

é um comentário, então a linha 45 é adicionada à tabela de símbolos como o tipo L e atribuída à posição 14 da SML.

A instrução

```
50 goto 20
```

transfere o controle para a linha 20. O número de linha 50 é armazenado na tabela de símbolos como o tipo L e atribuído à posição 14 da SML. O equivalente de goto em SML é a instrução *unconditional branch* (40), que transfere o controle para uma posição específica de SML. O compilador pesquisa a tabela de símbolos para a linha 20 e acha que corresponde à posição 01 de SML. O código de operação (40) é multiplicado por 100 e a posição 01 é adicionada para produzir a instrução 14 +4001.

A instrução

```
55 rem   output result
```

é um comentário, então a linha 55 é inserida na tabela de símbolos como o tipo L e é atribuída à posição 15 da SML.

A instrução

```
60 print t
```

é uma instrução de saída. O número de linha 60 é armazenado na tabela de símbolos como o tipo L e atribuído à posição 15 da SML. O equivalente de print na SML é o código de operação 11 (*write*). A posição de t é determinada a partir da tabela de símbolos e adicionada ao resultado do código de operação multiplicado por 100.

A instrução

```
99 end
```

é a linha final do programa. O número de linha 99 é armazenado na tabela de símbolos como o tipo L e atribuído à posição 16 de SML. O comando end produz a instrução de SML +4300 (43 é *halt* na SML), que é escrita como a instrução final no array de memória de SML.

Isso completa a primeira passagem do compilador. Agora consideramos a segunda passagem. O array `flags` é pesquisado quanto a outros valores que não `-1`. A posição `03` contém `60`, então o compilador sabe que a instrução `03` está incompleta. O compilador completa a instrução pesquisando `60` na tabela de símbolos, determinando sua posição e adicionando a posição à instrução incompleta. Nesse caso, a pesquisa determina que a linha `60` corresponde à posição `15` de SML, então a instrução completada `03 +4215` é produzida, substituindo `03 +4200`. O programa Simple agora compilou com sucesso.

Para construir o compilador, você terá de realizar cada uma das seguintes tarefas:

a) Modifique o programa do Simpletron Simulator que você escreveu no Exercício 8.19 para pegar sua entrada de um arquivo especificado pelo usuário (veja o Capítulo 17). O simulador deve enviar seus resultados para um arquivo de saída em disco no mesmo formato que a saída em tela. Converta o simulador para ser um programa orientado a objetos. Em particular, torne cada parte do hardware um objeto. Organize os tipos de instrução em uma hierarquia de classes utilizando herança. Então execute o programa polimorficamente fazendo cada instrução executar a si própria com uma mensagem `executeInstruction`.

b) Modifique o algoritmo de conversão de infixo para pós-fixo do Exercício 21.12 para processar operandos de inteiro de múltiplos dígitos e operandos de nome de variável de uma única letra. [*Dica:* A função `strtok` da C++ Standard Library pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros utilizando a função da biblioteca-padrão `atoi` (`<csdtlib>`).] [*Nota:* A representação de dados da expressão pós-fixa deve ser alterada para suportar nomes de variáveis e constantes de inteiro.]

c) Modifique o algoritmo de avaliação pós-fixa para processar operandos de inteiro de múltiplos dígitos e operandos de nome de variável. Além disso, o algoritmo agora deve implementar o 'gancho' discutido anteriormente de modo que sejam produzidas instruções de SML em vez de diretamente avaliar a expressão. [*Dica:* A função da biblioteca-padrão `strtok` pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros utilizando a função da biblioteca-padrão `atoi`.] [*Nota:* A representação de dados da expressão pós-fixa deve ser alterada para suportar nomes de variáveis e constantes de inteiro.]

d) Construa o compilador. Incorpore as partes (b) e (c) para avaliar as expressões em instruções `let`. Seu programa deve conter uma função que realiza a primeira passagem do compilador e uma função que realiza a segunda passagem do compilador. Ambas as funções podem chamar outras funções para realizar suas tarefas. Torne seu compilador orientado a objeto o máximo possível.

21.28 (*Otimizando o compilador de Simple*) Quando um programa é compilado e convertido em SML, um conjunto de instruções é gerado. Certas combinações de instruções freqüentemente se repetem, normalmente em triplos chamados *produções*. Uma produção normalmente consiste em três instruções, como *load*, *add* e *store*. Por exemplo, a Figura 21.32 ilustra cinco das instruções da SML que foram produzidas na compilação do programa na Figura 21.30. As primeiras três instruções são a produção que adiciona `1` a `y`. Observe que as instruções `06` e `07` armazenam o valor do acumulador na posição temporária `96` e carregam o valor de volta no acumulador de modo que a instrução `08` possa armazenar o valor na posição `98`. Freqüentemente uma produção é seguida por uma instrução de carregar na mesma posição que acabou de ser armazenada. Esse código pode ser *otimizado* eliminando a instrução de armazenar e a instrução subseqüente de carregar que opera na mesma posição da memória, permitindo assim a Simpletron executar o programa mais rápido. A Figura 21.33 ilustra a SML otimizada para o programa da Figura 21.30. Observe que há menos quatro instruções no código otimizado — uma economia de 25% de espaço de memória.

Modifique o compilador para fornecer uma opção para otimizar o código que a Simpletron Machine Language produz. Compare manualmente o código não otimizado com o código otimizado e calcule a redução de porcentagem.

21.29 (*Modificações no compilador de Simple*) Realize as seguintes modificações no compilador de Simple. Algumas dessas modificações também podem exigir modificações no programa Simpletron Simulator escrito no Exercício 8.19.

a) Permita que o operador de módulo (`%`) seja utilizado em instruções `let`. A Simpletron Machine Language deve ser modificada para incluir uma instrução de módulo.

b) Permita exponenciação em uma instrução `let` utilizando `^` como o operador de exponenciação. A Simpletron Machine Language deve ser modificada para incluir uma instrução de exponenciação.

c) Permita que o compilador reconheça letras minúsculas e maiúsculas em instruções Simple (por exemplo, `'A'` é equivalente a `'a'`). Nenhuma modificação no Simulator é necessária.

d) Permita que as instruções `input` leiam os valores e os transfiram para múltiplas variáveis como `input x, y`. Nenhuma modificação no Simpletron Simulator é necessária.

```
1  04  +2098  (load)
2  05  +3097  (add)
3  06  +2196  (store)
4  07  +2096  (load)
5  08  +2198  (store)
```

Figura 21.32 Código não otimizado do programa da Figura 21.30.

Programa Simple	Posição & instrução SML	Descrição
`5 rem sum 1 to x`	*nenhuma*	`rem` ignorado
`10 input x`	00 +1099	lê `x` na posição `99`
`15 rem check y == x`	*nenhuma*	`rem` ignorado
`20 if y == x goto 60`	01 +2098	carrega `y` (`98`) no acumulador
	02 +3199	subtrai `x` (`99`) do acumulador
	03 +4211	desvia para a posição `11` se zero
`25 rem increment y`	*nenhuma*	`rem` ignorado
`30 let y = y + 1`	04 +2098	carrega `y` no acumulador
	05 +3097	adiciona `1` (`97`) ao acumulador
	06 +2198	armazena o acumulador em `y` (`98`)
`35 rem add y to total`	*nenhuma*	`rem` ignorado
`40 let t = t + y`	07 +2096	carrega `t` a partir da posição (`96`)
	08 +3098	adiciona `y` (`98`) ao acumulador
	09 +2196	armazena o acumulador em `t` (`96`)
`45 rem loop y`	*nenhuma*	`rem` ignorado
`50 goto 20`	10 +4001	desvia para a posição `01`
`55 rem output result`	*nenhuma*	`rem` ignorado
`60 print t`	11 +1196	gera saída de `t` (`96`) para tela
`99 end`	12 +4300	termina a execução

Figura 21.33 Código otimizado para o programa da Figura 21.30.

e) Permita que o compilador gere saída de múltiplos valores em uma única instrução `print`, como `print a, b, c`. Nenhuma modificação no Simpletron Simulator é necessária.

f) Adicione capacidades de verificação de sintaxe ao compilador de modo que as mensagens de erro sejam enviadas para a saída quando erros de sintaxe forem encontrados em um programa Simple. Nenhuma modificação no Simpletron Simulator é necessária.

g) Permita arrays de inteiros. Nenhuma modificação no Simpletron Simulator é necessária.

h) Permita sub-rotinas especificadas pelos comandos `gosub` e `return` do Simple. O comando `gosub` passa o controle do programa para uma sub-rotina e o comando `return` passa o controle de volta à instrução depois do `gosub`. Isso é semelhante a uma chamada de função em C++. A mesma sub-rotina pode ser chamada de muitos comandos `gosub` distribuídos por todo um programa. Nenhuma modificação no Simpletron Simulator é necessária.

i) Permita instruções de repetição da fórmula

```
for x = 2 to 10 step 2
   Simple statements
next
```

Essa instrução `for` faz loop de 2 a 10 com um incremento de 2. A linha `next` marca o fim do corpo do `for`. Nenhuma modificação no Simpletron Simulator é necessária.

j) Permita instruções de repetição da fórmula

```
for x = 2 to 10
   Simple statements
next
```

Essa instrução `for` faz loop de 2 a 10 com um incremento-padrão de 1. Nenhuma modificação no Simpletron Simulator é necessária.

k) Permita que o compilador processe entrada e saída de string. Isso requer que o Simpletron Simulator seja modificado para processar e armazenar valores de string. [*Dica:* Cada palavra do Simpletron pode ser dividida em dois grupos, cada uma armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente ASCII decimal de um caractere. Adicione uma instrução de linguagem de máquina que imprimirá uma string inicial em certa posição da memória de Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada meia palavra sucessiva contém um caractere ASCII expresso como dois dígitos decimais. A instrução de linguagem de máquina verifica o comprimento e imprime a string traduzindo cada número de dois dígitos em seu caractere equivalente.]

l) Permita que o compilador processe valores de ponto flutuante além de inteiros. O Simpletron Simulator também deve ser modificado para processar valores de ponto flutuante.

21.30 (*Um interpretador de Simple*) Um interpretador é um programa que lê uma instrução de programa de uma linguagem de alto nível, determina a operação a ser realizada pela instrução e executa a operação imediatamente. O programa de linguagem de alto nível não é convertido em linguagem de máquina primeiro. Os interpretadores executam lentamente porque cada instrução encontrada no programa deve ser decifrada primeiro. Se as instruções estão contidas em um loop, as instruções são decifradas toda vez que são encontradas no loop. As versões anteriores da linguagem de programação BASIC foram implementadas como interpretadores.

Escreva um interpretador para a linguagem Simple discutida no Exercício 21.26. O programa deve utilizar o conversor de infixo para pós-fixo desenvolvido no Exercício 21.12 e o avaliador de pós-fixo desenvolvido no Exercício 21.13 para avaliar expressões em uma instrução `let`. As mesmas restrições impostas à linguagem Simple do Exercício 21.26 devem ser obedecidas nesse programa. Teste o interpretador com os programas Simple escritos no Exercício 21.26. Compare os resultados de executar esses programas no interpretador com os resultados de compilar os programas Simple e executá-los no Simpletron Simulator construído no Exercício 8.19.

21.31 (*Inserção/exclusão em qualquer lugar em uma lista vinculada*) Nosso template de classe de listas vinculadas permitia inserções e exclusões no início e no fim da lista vinculada. Essas capacidades foram convenientes quando utilizamos a herança `private` e a composição para produzir um template de classe de pilhas e um template de classe de filas com uma quantidade mínima de código reutilizando o template de classe de lista. Na realidade, as listas vinculadas são mais gerais que aquelas que fornecemos. Modifique o template de classe de listas vinculadas que desenvolvemos neste capítulo para tratar inserções e exclusões em qualquer lugar na lista.

21.32 (*Lista e filas sem ponteiros de cauda*) Nossa implementação de uma lista vinculada (figuras 21.3–21.5) utilizou tanto um `firstPtr` como um `lastPtr`. O `lastPtr` foi útil para as funções-membro `insertAtBack` e `removeFromBack` da classe `List`. A função `insertAtBack` corresponde à função-membro `enqueue` da classe `Queue`. Reescreva a classe `List` de modo que ela não utilize um `lastPtr`. Portanto, quaisquer operações no fim de uma lista devem começar pesquisando no início da lista. Isso afeta nossa implementação da classe `Queue` (Figura 21.16)?

21.33 Utilize a versão composta do programa de pilha (Figura 21.15) para formar um programa de pilha funcional completo. Modifique esse programa para funções-membro `inline`. Compare as duas abordagens. Resuma as vantagens e desvantagens de colocar inline funções-membro.

21.34 (*Desempenho da classificação e da pesquisa de árvore binária*) Um problema com a classificação de árvore binária é que a ordem em que os dados são inseridos afeta a forma da árvore — para a mesma coleção de dados, diferentes ordens podem produzir árvores binárias de formas significativamente diferentes. O desempenho dos algoritmos de classificação e pesquisa de árvore binária é sensível à forma da árvore binária. Que forma teria uma árvore binária se seus dados fossem inseridos na ordem crescente? E na ordem decrescente? Que forma a árvore deveria ter para alcançar desempenho máximo de pesquisa?

21.35 (*Listas indexadas*) Como apresentado no texto, as listas vinculadas devem ser pesquisadas seqüencialmente. Para listas grandes, isso pode resultar em desempenho pobre. Uma técnica comum para aprimorar o desempenho de pesquisa de lista é criar e manter um índice para a lista. Um índice é um conjunto de ponteiros para vários lugares-chave na lista. Por exemplo, um aplicativo que pesquisa uma grande lista de nomes pode aprimorar seu desempenho criando um índice com 26 entradas — uma para cada letra do alfabeto. Uma operação de pesquisa de um sobrenome que inicia com 'Y' primeiro pesquisaria o índice para determinar onde as entradas 'Y' iniciam e então 'saltaria' na lista nesse ponto e pesquisaria linearmente até que o nome desejado fosse localizado. Isso seria muito mais rápido que pesquisar a lista vinculada desde o início. Utilize a classe `List` das figuras 21.3–21.5 como a base de uma classe `IndexedList`. Escreva um programa que demonstra a operação de listas indexadas. Certifique-se de incluir as funções-membro `insertInIndexedList`, `searchIndexedList` e `deleteFromIndexedList`.