Illustrations List   *(Main Page)*

Fig. 15.1   Two self-referential class objects linked together.

Fig. 15.2   A graphical representation of a list.

```
1  // Fig. 15.3: listnd.h
2  // ListNode template definition
3  #ifndef LISTND_H
4  #define LISTND_H
```

Fig. 15.3   Manipulating a linked list (part 1 of 8).

```
5
6  template< class NODETYPE > class List;  // forward declaration
7
8  template<class NODETYPE>
9  class ListNode {
10    friend class List< NODETYPE >; // make List a friend
11 public:
12    ListNode( const NODETYPE & );  // constructor
13    NODETYPE getData() const;      // return data in the node
14 private:
15    NODETYPE data;                 // data
16    ListNode< NODETYPE > *nextPtr; // next node in the list
17 };
18
19 // Constructor
20 template<class NODETYPE>
21 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
22    : data( info ), nextPtr( 0 ) { }
23
24 // Return a copy of the data in the node
25 template< class NODETYPE >
26 NODETYPE ListNode< NODETYPE >::getData() const { return data; }
27
28 #endif
```

Fig. 15.3   Manipulating a linked list (part 2 of 8).

```
29 // Fig. 15.3: list.h
30 // Template List class definition
31 #ifndef LIST_H
32 #define LIST_H
33
34 #include <iostream.h>
35 #include <assert.h>
36 #include "listnd.h"
37
38 template< class NODETYPE >
39 class List {
```

```
40   public:
41      List();        // constructor
42      ~List();       // destructor
43      void insertAtFront( const NODETYPE & );
44      void insertAtBack( const NODETYPE & );
45      bool removeFromFront( NODETYPE & );
46      bool removeFromBack( NODETYPE & );
47      bool isEmpty() const;
48      void print() const;
49   private:
50      ListNode< NODETYPE > *firstPtr;  // pointer to first node
51      ListNode< NODETYPE > *lastPtr;   // pointer to last node
```

Fig. 15.3   Manipulating a linked list (part 3 of 8).

```
52
53      // Utility function to allocate a new node
54      ListNode< NODETYPE > *getNewNode( const NODETYPE & );
55   };
56
57   // Default constructor
58   template< class NODETYPE >
59   List< NODETYPE >::List() : firstPtr( 0 ), lastPtr( 0 ) { }
60
61   // Destructor
62   template< class NODETYPE >
63   List< NODETYPE >::~List()
64   {
65      if ( !isEmpty() ) {     // List is not empty
66         cout << "Destroying nodes ...\n";
67
68         ListNode< NODETYPE > *currentPtr = firstPtr, *tempPtr;
69
70         while ( currentPtr != 0 ) {  // delete remaining nodes
71            tempPtr = currentPtr;
72            cout << tempPtr->data << '\n';
73            currentPtr = currentPtr->nextPtr;
74            delete tempPtr;
75         }
76      }
77
78      cout << "All nodes destroyed\n\n";
79   }
80
81   // Insert a node at the front of the list
82   template< class NODETYPE >
83   void List< NODETYPE >::insertAtFront( const NODETYPE &value )
84   {
85      ListNode< NODETYPE > *newPtr = getNewNode( value );
86
87      if ( isEmpty() )  // List is empty
88         firstPtr = lastPtr = newPtr;
89      else {            // List is not empty
90         newPtr->nextPtr = firstPtr;
91         firstPtr = newPtr;
92      }
93   }
94
95   // Insert a node at the back of the list
96   template< class NODETYPE >
97   void List< NODETYPE >::insertAtBack( const NODETYPE &value )
98   {
99      ListNode< NODETYPE > *newPtr = getNewNode( value );
100
```

```
101     if ( isEmpty() )  // List is empty
102         firstPtr = lastPtr = newPtr;
```

Fig. 15.3   Manipulating a linked list (part 4 of 8).

```
103     else {              // List is not empty
104         lastPtr->nextPtr = newPtr;
105         lastPtr = newPtr;
106     }
107  }
108
109  // Delete a node from the front of the list
110  template< class NODETYPE >
111  bool List< NODETYPE >::removeFromFront( NODETYPE &value )
112  {
113     if ( isEmpty() )                // List is empty
114         return false;              // delete unsuccessful
115     else {
116         ListNode< NODETYPE > *tempPtr = firstPtr;
117
118         if ( firstPtr == lastPtr )
119             firstPtr = lastPtr = 0;
120         else
121             firstPtr = firstPtr->nextPtr;
122
123         value = tempPtr->data;  // data being removed
124         delete tempPtr;
125         return true;               // delete successful
126     }
127  }
128
129  // Delete a node from the back of the list
130  template< class NODETYPE >
131  bool List< NODETYPE >::removeFromBack( NODETYPE &value )
132  {
133     if ( isEmpty() )
134         return false;    // delete unsuccessful
135     else {
136         ListNode< NODETYPE > *tempPtr = lastPtr;
137
138         if ( firstPtr == lastPtr )
139             firstPtr = lastPtr = 0;
140         else {
141             ListNode< NODETYPE > *currentPtr = firstPtr;
142
143             while ( currentPtr->nextPtr != lastPtr )
144                 currentPtr = currentPtr->nextPtr;
145
146             lastPtr = currentPtr;
147             currentPtr->nextPtr = 0;
148         }
149
150         value = tempPtr->data;
151         delete tempPtr;
```

Fig. 15.3   Manipulating a linked list (part 5 of 8).

```
152         return true;    // delete successful
153     }
154  }
155
156  // Is the List empty?
```

```
157   template< class NODETYPE >
158   bool List< NODETYPE >::isEmpty() const
159      { return firstPtr == 0; }
160
161   // Return a pointer to a newly allocated node
162   template< class NODETYPE >
163   ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
164                                         const NODETYPE &value )
165   {
166      ListNode< NODETYPE > *ptr =
167         new ListNode< NODETYPE >( value );
168      assert( ptr != 0 );
169      return ptr;
170   }
171
172   // Display the contents of the List
173   template< class NODETYPE >
174   void List< NODETYPE >::print() const
175   {
176      if ( isEmpty() ) {
177         cout << "The list is empty\n\n";
178         return;
179      }
180
181      ListNode< NODETYPE > *currentPtr = firstPtr;
182
183      cout << "The list is: ";
184
185      while ( currentPtr != 0 ) {
186         cout << currentPtr->data << ' ';
187         currentPtr = currentPtr->nextPtr;
188      }
189
190      cout << "\n\n";
191   }
192
193   #endif
```

Fig. 15.3   Manipulating a linked list (part 6 of 8).

```
194   // Fig. 15.3: fig15_03.cpp
195   // List class test
196   #include <iostream.h>
197   #include "list.h"
198
199   // Function to test an integer List
200   template< class T >
201   void testList( List< T > &listObject, const char *type )
202   {
203      cout << "Testing a List of " << type << " values\n";
204
205      instructions();
206      int choice;
207      T value;
208
209      do {
210         cout << "? ";
211         cin >> choice;
212
213         switch ( choice ) {
214            case 1:
215               cout << "Enter " << type << ": ";
216               cin >> value;
217               listObject.insertAtFront( value );
```

```
218                    listObject.print();
219                    break;
220                case 2:
221                    cout << "Enter " << type << ": ";
222                    cin >> value;
223                    listObject.insertAtBack( value );
224                    listObject.print();
225                    break;
226                case 3:
227                    if ( listObject.removeFromFront( value ) )
228                        cout << value << " removed from list\n";
229
230                    listObject.print();
231                    break;
232                case 4:
233                    if ( listObject.removeFromBack( value ) )
234                        cout << value << " removed from list\n";
235
236                    listObject.print();
237                    break;
238            }
239        } while ( choice != 5 );
240
241        cout << "End list test\n\n";
242 }
243
```

Fig. 15.3   Manipulating a linked list (part 7 of 8).

```
244 void instructions()
245 {
246    cout << "Enter one of the following:\n"
247         << "  1 to insert at beginning of list\n"
248         << "  2 to insert at end of list\n"
249         << "  3 to delete from beginning of list\n"
250         << "  4 to delete from end of list\n"
251         << "  5 to end list processing\n";
252 }
253
254 int main()
255 {
256    List< int > integerList;
257    testList( integerList, "integer" ); // test integerList
258
259    List< float > floatList;
260    testList( floatList, "float" );     // test integerList
261
262    return 0;
263 }
```

Fig. 15.3   Manipulating a linked list (part 8 of 8).

```
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
```

Fig. 15.4   Sample output for the program of Fig. 15.3 (part 1 of 2).

```
Testing a List of float 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 float: 1.1
The list is: 1.1

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

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

? 2
Enter float: 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
```
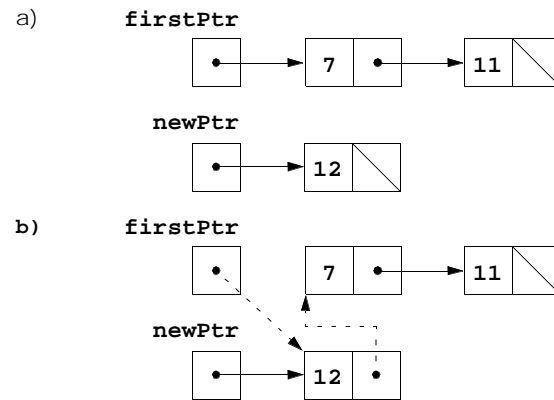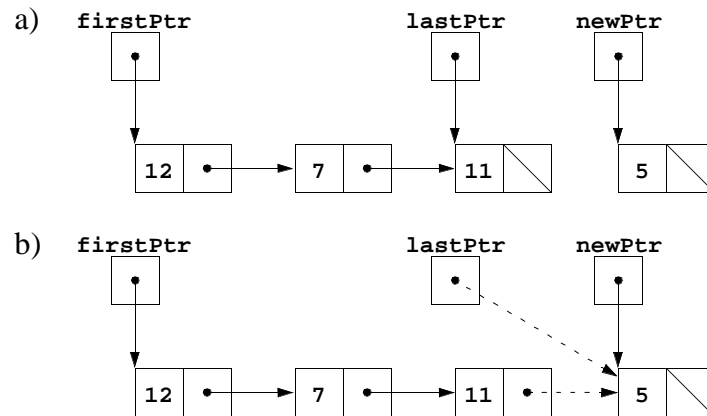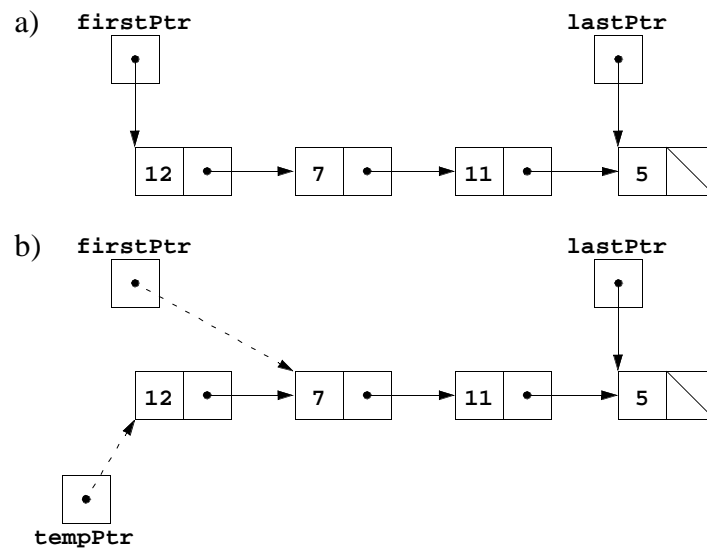
Fig. 15.4   Sample output for the program of Fig. 15.3 (part 2 of 2).

a)

Fig. 15.5   The **insertAtFront** operation.

Fig. 15.6   A graphical representation of the **insertAtBack** operation.

Fig. 15.7   A graphical representation of the **removeFromFront** operation.
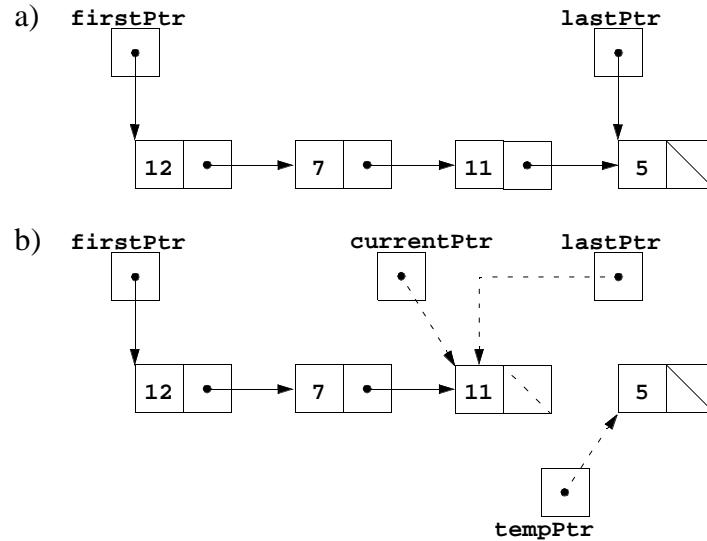
a)

b)

Fig. 15.8   A graphical representation of the **removeFromBack** operation.

```
1   // Fig. 15.9: stack.h
2   // Stack class template definition
3   // Derived from class List
4   #ifndef STACK_H
5   #define STACK_H
6
7   #include "list.h"
8
9   template< class STACKTYPE >
10  class Stack : private List< STACKTYPE > {
```

Fig. 15.9   A simple stack program (part 1 of 3).

```
11  public:
12     void push( const STACKTYPE &d ) { insertAtFront( d ); }
13     bool pop( STACKTYPE &d ) { return removeFromFront( d ); }
14     bool isStackEmpty() const { return isEmpty(); }
15     void printStack() const { print(); }
16  };
17
18  #endif
```

Fig. 15.9   A simple stack program (part 2 of 3).

```
19  // Fig. 15.9: fig15_09.cpp
20  // Driver to test the template Stack class
21  #include <iostream.h>
22  #include "stack.h"
23
24  int main()
25  {
26     Stack< int > intStack;
27     int popInteger;
28     cout << "processing an integer Stack" << endl;
29
30     for ( int i = 0; i < 4; i++ ) {
31        intStack.push( i );
```

```
32              intStack.printStack();
33         }
34
35         while ( !intStack.isStackEmpty() ) {
36              intStack.pop( popInteger );
37              cout << popInteger << " popped from stack" << endl;
38              intStack.printStack();
39         }
40
41         Stack< double > doubleStack;
42         double val = 1.1, popdouble;
43         cout << "processing a double Stack" << endl;
44
45         for ( i = 0; i < 4; i++ ) {
46              doubleStack.push( val );
47              doubleStack.printStack();
48              val += 1.1;
49         }
50
51         while ( !doubleStack.isStackEmpty() ) {
52              doubleStack.pop( popdouble );
53              cout << popdouble << " popped from stack" << endl;
54              doubleStack.printStack();
55         }
56         return 0;
57     }
```

Fig. 15.9   A simple stack program (part 3 of 3).

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

The list is: 1 0

The list is: 2 1 0

The list is: 3 2 1 0

3 popped from stack
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

The list is: 4.4 3.3 2.2 1.1

4.4 popped from stack
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
```

Fig. 15.10   Sample output from the program of Fig. 15.9.

```
1   // Fig. 15.11: stack_c.h
2   // Definition of Stack class composed of List object
3   #ifndef STACK_C
4   #define STACK_C
5   #include "list.h"
6
7   template< class STACKTYPE >
8   class Stack {
9   public:
10      // no constructor; List constructor does initialization
```

```
11        void push( const STACKTYPE &d ) { s.insertAtFront( d ); }
12        bool pop( STACKTYPE &d ) { return s.removeFromFront( d ); }
13        bool isStackEmpty() const { return s.isEmpty(); }
14        void printStack() const { s.print(); }
15   private:
16        List< STACKTYPE > s;
17   };
18
19   #endif
```

Fig. 15.11   A simple stack program using composition.

```
 1   // Fig. 15.12: queue.h
 2   // Queue class template definition
 3   // Derived from class List
 4   #ifndef QUEUE_H
 5   #define QUEUE_H
 6
 7   #include "list.h"
 8
 9   template< class QUEUETYPE >
10   class Queue: private List< QUEUETYPE > {
11   public:
12        void enqueue( const QUEUETYPE &d ) { insertAtBack( d ); }
13        bool dequeue( QUEUETYPE &d )
14            { return removeFromFront( d ); }
15        bool isQueueEmpty() const { return isEmpty(); }
16        void printQueue() const { print(); }
17   };
18
19   #endif
```

Fig. 15.12   Processing a queue (part 1 of 2).

```
20   // Fig. 15.12: fig15_12.cpp
21   // Driver to test the template Queue class
22   #include <iostream.h>
23   #include "queue.h"
24
25   int main()
26   {
27        Queue< int > intQueue;
28        int dequeueInteger;
29        cout << "processing an integer Queue" << endl;
30
31        for ( int i = 0; i < 4; i++ ) {
32            intQueue.enqueue( i );
33            intQueue.printQueue();
34        }
35
36        while ( !intQueue.isQueueEmpty() ) {
37            intQueue.dequeue( dequeueInteger );
38            cout << dequeueInteger << " dequeued" << endl;
39            intQueue.printQueue();
40        }
41
42        Queue< double > doubleQueue;
43        double val = 1.1, dequeuedouble;
44
45        cout << "processing a double Queue" << endl;
46
```

```
47      for ( i = 0; i < 4; i++ ) {
48         doubleQueue.enqueue( val );
49         doubleQueue.printQueue();
50         val += 1.1;
51      }
52
53      while ( !doubleQueue.isQueueEmpty() ) {
54         doubleQueue.dequeue( dequeuedouble );
55         cout << dequeuedouble << " dequeued" << endl;
56         doubleQueue.printQueue();
57      }
58
59      return 0;
60   }
```

Fig. 15.12   Processing a queue (part 2 of 2).

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

The list is: 0 1

The list is: 0 1 2

The list is: 0 1 2 3

0 dequeued
The list is: 1 2 3

1 dequeued
The list is: 2 3

2 dequeued
The list is: 3

3 dequeued
The list is empty

processing a float Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

The list is: 1.1 2.2 3.3 4.4

1.1 dequeued
The list is: 2.2 3.3 4.4

2.2 dequeued
The list is: 3.3 4.4

3.3 dequeued
The list is: 4.4

4.4 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

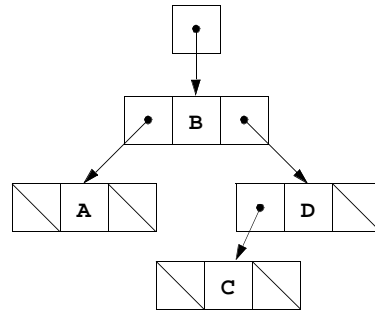**Fig. 15.13**   Sample output from the program in Fig. 15.12.

Fig. 15.14   A graphical representation of a binary tree.
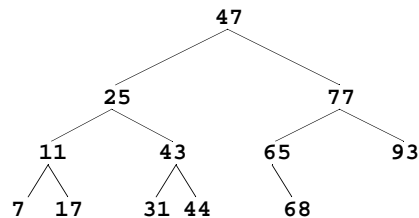

Fig. 15.15   A binary search tree.

```
1    // Fig. 15.16: treenode.h
2    // Definition of class TreeNode
3    #ifndef TREENODE_H
4    #define TREENODE_H
5
6    template< class NODETYPE > class Tree;  // forward declaration
7
8    template< class NODETYPE >
9    class TreeNode {
10      friend class Tree< NODETYPE >;
11   public:
12      TreeNode( const NODETYPE &d )
13         : leftPtr( 0 ), data( d ), rightPtr( 0 ) { }
14      NODETYPE getData() const { return data; }
15   private:
16      TreeNode< NODETYPE > *leftPtr;  // pointer to left subtree
17      NODETYPE data;
18      TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
19   };
20
21   #endif
```

Fig. 15.16   Creating and traversing a binary tree (part 1 of 6).

```
22   // Fig. 15.16: fig15_16.cpp
23   // Definition of template class Tree
24   #ifndef TREE_H
25   #define TREE_H
26
27   #include <iostream.h>
28   #include <assert.h>
29   #include "treenode.h"
```

```
30
31   template< class NODETYPE >
32   class Tree {
33   public:
34      Tree();
35      void insertNode( const NODETYPE & );
36      void preOrderTraversal() const;
37      void inOrderTraversal() const;
38      void postOrderTraversal() const;
39   private:
40      TreeNode< NODETYPE > *rootPtr;
41
42      // utility functions
43      void insertNodeHelper(
44              TreeNode< NODETYPE > **, const NODETYPE & );
45      void preOrderHelper( TreeNode< NODETYPE > * ) const;
46      void inOrderHelper( TreeNode< NODETYPE > * ) const;
47      void postOrderHelper( TreeNode< NODETYPE > * ) const;
48   };
49
50   template< class NODETYPE >
51   Tree< NODETYPE >::Tree() { rootPtr = 0; }
52
53   template< class NODETYPE >
54   void Tree< NODETYPE >::insertNode( const NODETYPE &value )
55      { insertNodeHelper( &rootPtr, value ); }
```

Fig. 15.16   Creating and traversing a binary tree (part 2 of 6).

```
56
57   // This function receives a pointer to a pointer so the
58   // pointer can be modified.
59   template< class NODETYPE >
60   void Tree< NODETYPE >::insertNodeHelper(
61          TreeNode< NODETYPE > **ptr, const NODETYPE &value )
62   {
63      if ( *ptr == 0 ) {                        // tree is empty
64         *ptr = new TreeNode< NODETYPE >( value );
65         assert( *ptr != 0 );
66      }
67      else                               // tree is not empty
68         if ( value < ( *ptr )->data )
69            insertNodeHelper( &( ( *ptr )->leftPtr ), value );
70         else
71            if ( value > ( *ptr )->data )
72               insertNodeHelper( &( ( *ptr )->rightPtr ), value );
73            else
74               cout << value << " dup" << endl;
75   }
76
77   template< class NODETYPE >
78   void Tree< NODETYPE >::preOrderTraversal() const
79      { preOrderHelper( rootPtr ); }
80
81   template< class NODETYPE >
82   void Tree< NODETYPE >::preOrderHelper(
83                          TreeNode< NODETYPE > *ptr ) const
84   {
85      if ( ptr != 0 ) {
86         cout << ptr->data << ' ';
87         preOrderHelper( ptr->leftPtr );
88         preOrderHelper( ptr->rightPtr );
89      }
90   }
```

```
91
92   template< class NODETYPE >
93   void Tree< NODETYPE >::inOrderTraversal() const
94      { inOrderHelper( rootPtr ); }
95
96   template< class NODETYPE >
97   void Tree< NODETYPE >::inOrderHelper(
98                             TreeNode< NODETYPE > *ptr ) const
99   {
100     if ( ptr != 0 ) {
101        inOrderHelper( ptr->leftPtr );
102        cout << ptr->data << ' ';
103        inOrderHelper( ptr->rightPtr );
104     }
105  }
```

Fig. 15.16   Creating and traversing a binary tree (part 3 of 6).

```
106
107  template< class NODETYPE >
108  void Tree< NODETYPE >::postOrderTraversal() const
109     { postOrderHelper( rootPtr ); }
110
111  template< class NODETYPE >
112  void Tree< NODETYPE >::postOrderHelper(
113                             TreeNode< NODETYPE > *ptr ) const
114  {
115     if ( ptr != 0 ) {
116        postOrderHelper( ptr->leftPtr );
117        postOrderHelper( ptr->rightPtr );
118        cout << ptr->data << ' ';
119     }
120  }
121
122  #endif
```

Fig. 15.16   Creating and traversing a binary tree (part 4 of 6).

```
123  // Fig. 15.16: fig15_16.cpp
124  // Driver to test class Tree
125  #include <iostream.h>
126  #include <iomanip.h>
127  #include "tree.h"
128
129  int main()
130  {
131     Tree< int > intTree;
132     int intVal;
133
134     cout << "Enter 10 integer values:\n";
135     for( int i = 0; i < 10; i++ ) {
136        cin >> intVal;
137        intTree.insertNode( intVal );
138     }
139
140     cout << "\nPreorder traversal\n";
141     intTree.preOrderTraversal();
142
143     cout << "\nInorder traversal\n";
144     intTree.inOrderTraversal();
145
146     cout << "\nPostorder traversal\n";
```

```
147     intTree.postOrderTraversal();
148
149     Tree< double > doubleTree;
150     double doubleVal;
151
```

Fig. 15.16   Creating and traversing a binary tree (part 5 of 6).

```
152     cout << "\n\n\nEnter 10 double values:\n"
153          << setiosflags( ios::fixed | ios::showpoint )
154          << setprecision( 1 );
155     for ( i = 0; i < 10; i++ ) {
156        cin >> doubleVal;
157        doubleTree.insertNode( doubleVal );
158     }
159
160     cout << "\nPreorder traversal\n";
161     doubleTree.preOrderTraversal();
162
163     cout << "\nInorder traversal\n";
164     doubleTree.inOrderTraversal();
165
166     cout << "\nPostorder traversal\n";
167     doubleTree.postOrderTraversal();
168
169     return 0;
170  }
```

Fig. 15.16   Creating and traversing a binary tree (part 6 of 6).

```
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
```

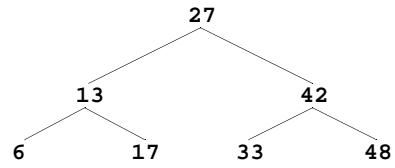**Fig. 15.17**   Sample output from the program of Fig. 15.16.

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

Fig. 15.18   A binary search tree.