Illustrations List ***(Main Page)***

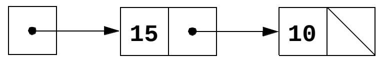

Fig. 15.1 Two self-referential class objects linked together.

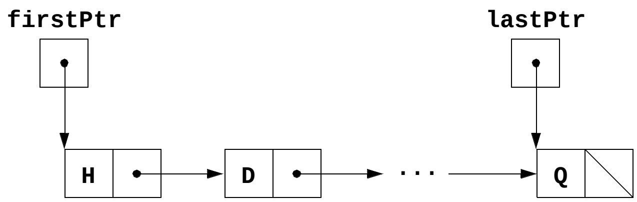

Fig. 15.2 A graphical representation of a list.

```
// Fig. 15.3: listnd.h
// ListNode template definition
#ifndef LISTND_H
#define LISTND_H
```

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

```

template< class NODETYPE > class List;  // forward declaration

template<class NODETYPE>
class ListNode {
   friend class List< NODETYPE >; // make List a friend
public:
   ListNode( const NODETYPE & );  // constructor
   NODETYPE getData() const;      // return data in the node
private:
   NODETYPE data;                 // data
   ListNode< NODETYPE > *nextPtr; // next node in the list
};

// Constructor
template<class NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
   : data( info ), nextPtr( 0 ) { }

// Return a copy of the data in the node
template< class NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const { return data; }

#endif
```

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

```
// Fig. 15.3: list.h
// Template List class definition
#ifndef LIST_H
#define LIST_H

#include <iostream.h>
#include <assert.h>
#include "listnd.h"

template< class NODETYPE >
class List {
```

```
public:
   List();      // constructor
   ~List();     // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool isEmpty() const;
   void print() const;
private:
   ListNode< NODETYPE > *firstPtr;  // pointer to first node
   ListNode< NODETYPE > *lastPtr;   // pointer to last node
```

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

```

   // Utility function to allocate a new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
};

// Default constructor
template< class NODETYPE >
List< NODETYPE >::List() : firstPtr( 0 ), lastPtr( 0 ) { }

// Destructor
template< class NODETYPE >
List< NODETYPE >::~List()
{
   if ( !isEmpty() ) {    // List is not empty
      cout << "Destroying nodes ...\n";

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

      while ( currentPtr != 0 ) {  // delete remaining nodes
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      }
   }

   cout << "All nodes destroyed\n\n";
}

// Insert a node at the front of the list
template< class NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

   if ( isEmpty() )  // List is empty
      firstPtr = lastPtr = newPtr;
   else {           // List is not empty
      newPtr->nextPtr = firstPtr;
      firstPtr = newPtr;
   }
}

// Insert a node at the back of the list
template< class NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value );

```

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

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

```
   else {           // List is not empty
      lastPtr->nextPtr = newPtr;
      lastPtr = newPtr;
   }
}

// Delete a node from the front of the list
template< class NODETYPE >
bool List< NODETYPE >::removeFromFront( NODETYPE &value )
{
   if ( isEmpty() )             // List is empty
      return false;             // delete unsuccessful
   else {
      ListNode< NODETYPE > *tempPtr = firstPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else
         firstPtr = firstPtr->nextPtr;

      value = tempPtr->data;  // data being removed
      delete tempPtr;
      return true;            // delete successful
   }
}

// Delete a node from the back of the list
template< class NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value )
{
   if ( isEmpty() )
      return false;   // delete unsuccessful
   else {
      ListNode< NODETYPE > *tempPtr = lastPtr;

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0;
      else {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr;

         lastPtr = currentPtr;
         currentPtr->nextPtr = 0;
      }

      value = tempPtr->data;
      delete tempPtr;
```

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

```
      return true;   // delete successful
   }
}

// Is the List empty?
```

```
template< class NODETYPE >
bool List< NODETYPE >::isEmpty() const
   { return firstPtr == 0; }

// Return a pointer to a newly allocated node
template< class NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
                                        const NODETYPE &value )
{
   ListNode< NODETYPE > *ptr =
      new ListNode< NODETYPE >( value );
   assert( ptr != 0 );
   return ptr;
}

// Display the contents of the List
template< class NODETYPE >
void List< NODETYPE >::print() const
{
   if ( isEmpty() ) {
      cout << "The list is empty\n\n";
      return;
   }

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

   while ( currentPtr != 0 ) {
      cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;
   }

   cout << "\n\n";
}

#endif
```

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

```
// Fig. 15.3: fig15_03.cpp
// List class test
#include <iostream.h>
#include "list.h"

// Function to test an integer List
template< class T >
void testList( List< T > &listObject, const char *type )
{
   cout << "Testing a List of " << type << " values\n";

   instructions();
   int choice;
   T value;

   do {
      cout << "? ";
      cin >> choice;

      switch ( choice ) {
         case 1:
            cout << "Enter " << type << ": ";
            cin >> value;
            listObject.insertAtFront( value );
```

```
            listObject.print();
            break;
         case 2:
            cout << "Enter " << type << ": ";
            cin >> value;
            listObject.insertAtBack( value );
            listObject.print();
            break;
         case 3:
            if ( listObject.removeFromFront( value ) )
               cout << value << " removed from list\n";

            listObject.print();
            break;
         case 4:
            if ( listObject.removeFromBack( value ) )
               cout << value << " removed from list\n";

            listObject.print();
            break;
      }
   } while ( choice != 5 );

   cout << "End list test\n\n";
}

```

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

```
void instructions()
{
   cout << "Enter one of the following:\n"
        << "  1 to insert at beginning of list\n"
        << "  2 to insert at end of list\n"
        << "  3 to delete from beginning of list\n"
        << "  4 to delete from end of list\n"
        << "  5 to end list processing\n";
}

int main()
{
   List< int > integerList;
   testList( integerList, "integer" ); // test integerList

   List< float > floatList;
   testList( floatList, "float" );     // test integerList

   return 0;
}
```

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).

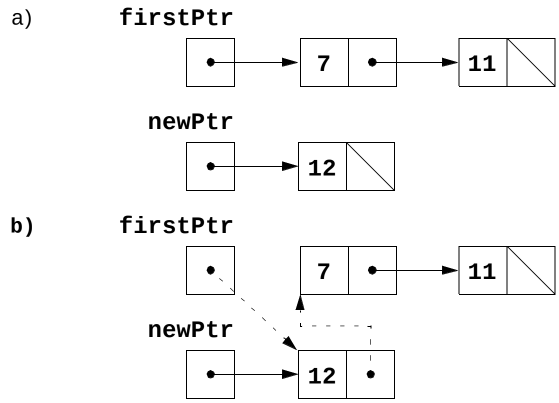

Fig. 15.5 The `insertAtFront` operation.

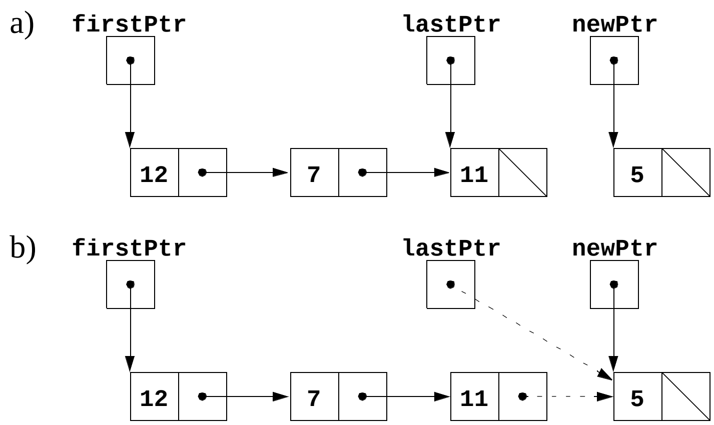

Fig. 15.6 A graphical representation of the `insertAtBack` operation.

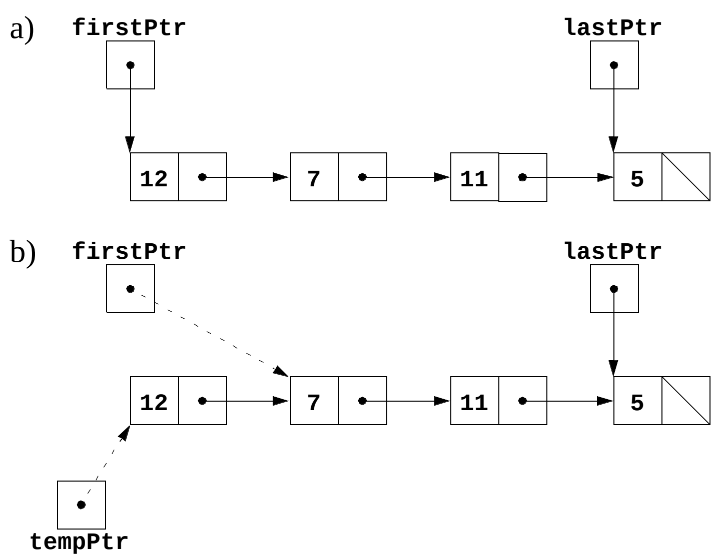

Fig. 15.7 A graphical representation of the `removeFromFront` operation.

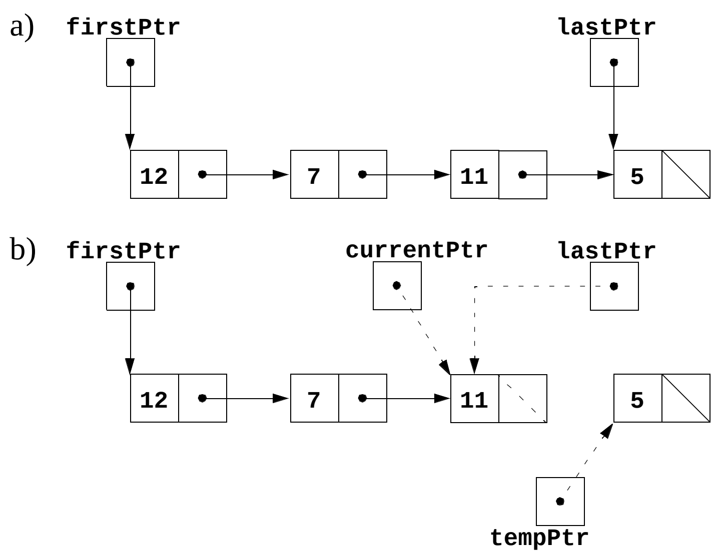

Fig. 15.8 A graphical representation of the `removeFromBack` operation.

```
// Fig. 15.9: stack.h
// Stack class template definition
// Derived from class List
#ifndef STACK_H
#define STACK_H

#include "list.h"

template< class STACKTYPE >
class Stack : private List< STACKTYPE > {
```

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

```
public:
   void push( const STACKTYPE &d ) { insertAtFront( d ); }
   bool pop( STACKTYPE &d ) { return removeFromFront( d ); }
   bool isStackEmpty() const { return isEmpty(); }
   void printStack() const { print(); }
};

#endif
```

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

```
// Fig. 15.9: fig15_09.cpp
// Driver to test the template Stack class
#include <iostream.h>
#include "stack.h"

int main()
{
   Stack< int > intStack;
   int popInteger;
   cout << "processing an integer Stack" << endl;

   for ( int i = 0; i < 4; i++ ) {
      intStack.push( i );
```

```
      intStack.printStack();
   }

   while ( !intStack.isStackEmpty() ) {
      intStack.pop( popInteger );
      cout << popInteger << " popped from stack" << endl;
      intStack.printStack();
   }

   Stack< double > doubleStack;
   double val = 1.1, popdouble;
   cout << "processing a double Stack" << endl;

   for ( i = 0; i < 4; i++ ) {
      doubleStack.push( val );
      doubleStack.printStack();
      val += 1.1;
   }

   while ( !doubleStack.isStackEmpty() ) {
      doubleStack.pop( popdouble );
      cout << popdouble << " popped from stack" << endl;
      doubleStack.printStack();
   }
   return 0;
}
```

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.

```
// Fig. 15.11: stack_c.h
// Definition of Stack class composed of List object
#ifndef STACK_C
#define STACK_C
#include "list.h"

template< class STACKTYPE >
class Stack {
public:
   // no constructor; List constructor does initialization
```

```
   void push( const STACKTYPE &d ) { s.insertAtFront( d ); }
   bool pop( STACKTYPE &d ) { return s.removeFromFront( d ); }
   bool isStackEmpty() const { return s.isEmpty(); }
   void printStack() const { s.print(); }
private:
   List< STACKTYPE > s;
};

#endif
```

Fig. 15.11 A simple stack program using composition.

```
// Fig. 15.12: queue.h
// Queue class template definition
// Derived from class List
#ifndef QUEUE_H
#define QUEUE_H

#include "list.h"

template< class QUEUETYPE >
class Queue: private List< QUEUETYPE > {
public:
   void enqueue( const QUEUETYPE &d ) { insertAtBack( d ); }
   bool dequeue( QUEUETYPE &d )
      { return removeFromFront( d ); }
   bool isQueueEmpty() const { return isEmpty(); }
   void printQueue() const { print(); }
};

#endif
```

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

```
// Fig. 15.12: fig15_12.cpp
// Driver to test the template Queue class
#include <iostream.h>
#include "queue.h"

int main()
{
   Queue< int > intQueue;
   int dequeueInteger;
   cout << "processing an integer Queue" << endl;

   for ( int i = 0; i < 4; i++ ) {
      intQueue.enqueue( i );
      intQueue.printQueue();
   }

   while ( !intQueue.isQueueEmpty() ) {
      intQueue.dequeue( dequeueInteger );
      cout << dequeueInteger << " dequeued" << endl;
      intQueue.printQueue();
   }

   Queue< double > doubleQueue;
   double val = 1.1, dequeuedouble;

   cout << "processing a double Queue" << endl;

```

```
   for ( i = 0; i < 4; i++ ) {
      doubleQueue.enqueue( val );
      doubleQueue.printQueue();
      val += 1.1;
   }

   while ( !doubleQueue.isQueueEmpty() ) {
      doubleQueue.dequeue( dequeuedouble );
      cout << dequeuedouble << " dequeued" << endl;
      doubleQueue.printQueue();
   }

   return 0;
}
```

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.

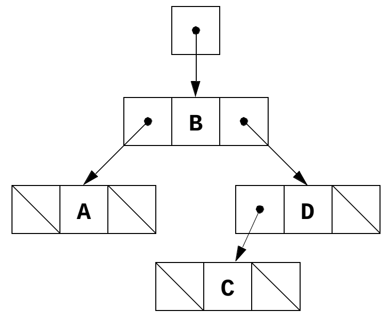

Fig. 15.14 A graphical representation of a binary tree.

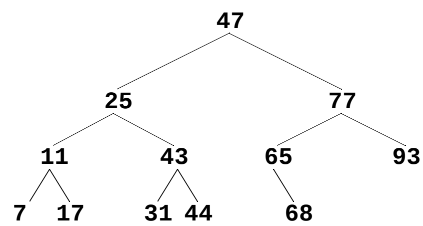

Fig. 15.15 A binary search tree.

```
// Fig. 15.16: treenode.h
// Definition of class TreeNode
#ifndef TREENODE_H
#define TREENODE_H

template< class NODETYPE > class Tree;  // forward declaration

template< class NODETYPE >
class TreeNode {
   friend class Tree< NODETYPE >;
public:
   TreeNode( const NODETYPE &d )
      : leftPtr( 0 ), data( d ), rightPtr( 0 ) { }
   NODETYPE getData() const { return data; }
private:
   TreeNode< NODETYPE > *leftPtr;  // pointer to left subtree
   NODETYPE data;
   TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
};

#endif
```

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

```
// Fig. 15.16: fig15_16.cpp
// Definition of template class Tree
#ifndef TREE_H
#define TREE_H

#include <iostream.h>
#include <assert.h>
#include "treenode.h"
```

```
template< class NODETYPE >
class Tree {
public:
   Tree();
   void insertNode( const NODETYPE & );
   void preOrderTraversal() const;
   void inOrderTraversal() const;
   void postOrderTraversal() const;
private:
   TreeNode< NODETYPE > *rootPtr;

   // utility functions
   void insertNodeHelper(
           TreeNode< NODETYPE > **, const NODETYPE & );
   void preOrderHelper( TreeNode< NODETYPE > * ) const;
   void inOrderHelper( TreeNode< NODETYPE > * ) const;
   void postOrderHelper( TreeNode< NODETYPE > * ) const;
};

template< class NODETYPE >
Tree< NODETYPE >::Tree() { rootPtr = 0; }

template< class NODETYPE >
void Tree< NODETYPE >::insertNode( const NODETYPE &value )
   { insertNodeHelper( &rootPtr, value ); }
```

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

```

// This function receives a pointer to a pointer so the
// pointer can be modified.
template< class NODETYPE >
void Tree< NODETYPE >::insertNodeHelper(
        TreeNode< NODETYPE > **ptr, const NODETYPE &value )
{
   if ( *ptr == 0 ) {                   // tree is empty
      *ptr = new TreeNode< NODETYPE >( value );
      assert( *ptr != 0 );
   }
   else                              // tree is not empty
      if ( value < ( *ptr )->data )
         insertNodeHelper( &( ( *ptr )->leftPtr ), value );
      else
         if ( value > ( *ptr )->data )
            insertNodeHelper( &( ( *ptr )->rightPtr ), value );
         else
            cout << value << " dup" << endl;
}

template< class NODETYPE >
void Tree< NODETYPE >::preOrderTraversal() const
   { preOrderHelper( rootPtr ); }

template< class NODETYPE >
void Tree< NODETYPE >::preOrderHelper(
                              TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 ) {
      cout << ptr->data << ' ';
      preOrderHelper( ptr->leftPtr );
      preOrderHelper( ptr->rightPtr );
   }
}
```

```
template< class NODETYPE >
void Tree< NODETYPE >::inOrderTraversal() const
   { inOrderHelper( rootPtr ); }

template< class NODETYPE >
void Tree< NODETYPE >::inOrderHelper( 
                          TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 ) {
      inOrderHelper( ptr->leftPtr );
      cout << ptr->data << ' ';
      inOrderHelper( ptr->rightPtr );
   }
}
```

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

```
template< class NODETYPE >
void Tree< NODETYPE >::postOrderTraversal() const
   { postOrderHelper( rootPtr ); }

template< class NODETYPE >
void Tree< NODETYPE >::postOrderHelper( 
                          TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 ) {
      postOrderHelper( ptr->leftPtr );
      postOrderHelper( ptr->rightPtr );
      cout << ptr->data << ' ';
   }
}

#endif
```

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

```
// Fig. 15.16: fig15_16.cpp
// Driver to test class Tree
#include <iostream.h>
#include <iomanip.h>
#include "tree.h"

int main()
{
   Tree< int > intTree;
   int intVal;

   cout << "Enter 10 integer values:\n";
   for( int i = 0; i < 10; i++ ) {
      cin >> intVal;
      intTree.insertNode( intVal );
   }

   cout << "\nPreorder traversal\n";
   intTree.preOrderTraversal();

   cout << "\nInorder traversal\n";
   intTree.inOrderTraversal();

   cout << "\nPostorder traversal\n";
```

```
   intTree.postOrderTraversal();

   Tree< double > doubleTree;
   double doubleVal;

```

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

```
   cout << "\n\n\nEnter 10 double values:\n"
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 1 );
   for ( i = 0; i < 10; i++ ) {
      cin >> doubleVal;
      doubleTree.insertNode( doubleVal );
   }

   cout << "\nPreorder traversal\n";
   doubleTree.preOrderTraversal();

   cout << "\nInorder traversal\n";
   doubleTree.inOrderTraversal();

   cout << "\nPostorder traversal\n";
   doubleTree.postOrderTraversal();

   return 0;
}
```

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.

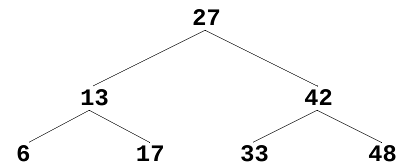

Fig. 15.18 A binary search tree.