Chapter 20
Custom Templatized Data Structures

C++ How to Program, 8/e

OBJECTIVES

In this chapter you'll learn:

- To form linked data structures using pointers, self-referential classes and recursion.
- To create and manipulate dynamic data structures such as linked lists, queues, stacks and binary trees.
- To use binary search trees for high-speed searching and sorting.
- To understand important applications of linked data structures.
- To understand how to create reusable data structures with class templates, inheritance and composition.

20.1 Introduction

- We’ve studied fixed-size data structures such as one-dimensional arrays and two-dimensional arrays.
- This chapter introduces dynamic data structures that grow and shrink during execution.
- Linked lists are collections of data items logically “lined up in a row”—insertions and removals are made anywhere in a linked list.
- Stacks are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and removals are made from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representation of file-system directories and compilation of expressions into machine language.

20.1 Introduction (cont.)

- We use classes, class templates, inheritance and composition to create and package these data structures for reusability and maintainability.
- This chapter is solid preparation for Chapter 22, Standard Template Library (STL).
  - The STL is a major portion of the C++ Standard Library.
  - The STL provides containers, iterators for traversing those containers and algorithms for processing the containers' elements.
  - The STL packages data structrues into templatized classes.
  - The STL code is carefully written to be portable, efficient and extensible.

20.2 Self-Referential Classes

- A self-referential class contains a pointer member that points to a class object of the same class type.
- Sample `Node` class definition:

```
class Node
{
public:
   Node( int ); // constructor
   void setData( int ); // set data member
   int getData() const; // get data member
   void setNextPtr( Node * ); // set pointer to next Node
   Node *getNextPtr() const; // get pointer to next Node
private:
   int data; // data stored in this Node
   Node *nextPtr; // pointer to another object of same type
}; // end class Node
```

20.2 Self-Referential Classes (cont.)

- Member `nextPtr` points to an object of type `Node`—another object of the same type as the one being declared here, hence the term "self-referential class."
- Member `nextPtr` is referred to as a link—i.e., `nextPtr` can "tie" an object of type `Node` to another object of the same type.
- Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees.
- Figure 20.1 illustrates two self-referential class objects linked together to form a list.
- Note that a slash—representing a null (`0`) pointer—is placed in the link member of the second self-referential class object to indicate that the link does not point to another object.
- The slash is only for illustration purposes; it does not correspond to the backslash character in C++.
- A null pointer normally indicates the end of a data structure just as the null character (`'\0'`) indicates the end of a string.

Common Programming Error 20.1

Not setting the link in the last node of a linked data structure to null (0) is a (possibly fatal) logic error.

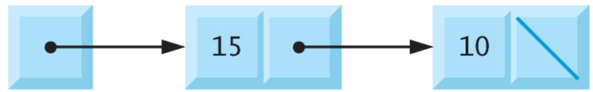

Fig. 20.1 | Two self-referential class objects linked together.

20.3 Dynamic Memory Allocation and Data Structures

- Creating and maintaining dynamic data structures requires dynamic memory allocation, which enables a program to obtain more memory at execution time to hold new nodes.
- When that memory is no longer needed by the program, the memory can be released so that it can be reused to allocate other objects in the future.
- The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system.
- Often, the limits are much smaller, because available memory must be shared among many programs.

20.3 Dynamic Memory Allocation and Data Structures (cont.)

- The `new` operator takes as an argument the type of the object being dynamically allocated and returns a pointer to an object of that type.
- For example, the following statement allocates `sizeof( Node )` bytes, runs the `Node` constructor and assigns the new `Node`'s address to `newPtr`.
  - ```
    // create Node with data 10
    Node *newPtr = new Node( 10 );
    ```
- If no memory is available, `new` throws a `bad_alloc` exception.
- The `delete` operator runs the `Node` destructor and deallocates memory allocated with `new`—the memory is returned to the system so that the memory can be reallocated in the future.
    ```

20.3 Dynamic Memory Allocation and Data Structures (cont.)

- To free memory dynamically allocated by the preceding `new`, use the statement
  - `delete newPtr;`
- Note that `newPtr` itself is not deleted; rather the desctructor of the `node` object that `newPtr` points to is called and the object's memory is freed.
- If pointer `newPtr` has the null pointer value `0`, the preceding statement has no effect.

20.4 Linked Lists

- A linked list is a linear collection of self-referential class objects, called nodes, connected by pointer links—hence, the term “linked” list.
- A linked list is accessed via a pointer to the list’s first node.
- Each subsequent node is accessed via the link-pointer member stored in the previous node.
- By convention, the link pointer in the last node of a list is set to null (0) to mark the end of the list.
- Data is stored in a linked list dynamically—each node is created as necessary.
- A node can contain data of any type, including objects of other classes.

20.4 Linked Lists (cont.)

- If nodes contain base-class pointers to base-class and derived-class objects related by inheritance, we can have a linked list of such nodes and process them polymorphically using `virtual` function calls.
- Stacks and queues are also linear data structures and, as we'll see, can be viewed as constrained versions of linked lists.
- Trees are nonlinear data structures.

20.4 Linked Lists (cont.)

- Lists of data can be stored in arrays, but linked lists provide several advantages.
- A linked list is appropriate when the number of data elements to be represented at one time is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- The size of a "conventional" C++ array, however, cannot be altered, because the array size is fixed at compile time.
- "Conventional" arrays can become full.
- Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Performance Tip 20.1

An array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime. Class template `vector` (Section 7.11) implements a dynamically resizable array-based data structure.

20.4 Linked Lists (cont.)

- Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.
- Existing list elements do not need to be moved.
- Pointers merely need to be updated to point to the correct node.

Performance Tip 20.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately. A linked list allows efficient insertion operations anywhere in the list.

Performance Tip 20.3

The elements of an array are stored contiguously in memory. This allows immediate access to any element, because an element's address can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate direct access to their elements. So accessing individual elements in a linked list can be considerably more expensive than accessing individual elements in an array. The selection of a data structure is typically based on the performance of specific operations used by a program and the order in which the data items are maintained in the data structure. For example, it's typically more efficient to insert an item in a sorted linked list than a sorted array.

20.4 Linked Lists (cont.)

- Linked-list nodes are not stored contiguously in memory, but logically they appear to be contiguous.
- Figure 20.2 illustrates a linked list with several nodes.

Performance Tip 20.4

Using dynamic memory allocation (instead of fixed-size arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that pointers occupy space and that dynamic memory allocation incurs the overhead of function calls.

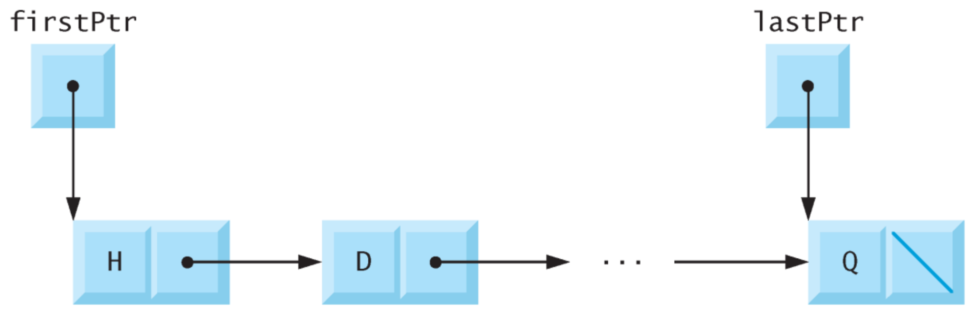

Fig. 20.2 | A graphical representation of a list.

20.4 Linked Lists (cont.)

- The program of Figs. 20.3–20.5 uses a `List` class template to manipulate a list of integer values and a list of floating-point values.
- The program uses class templates `ListNode` (Fig. 20.3) and `List` (Fig. 20.4).
- Encapsulated in each `List` object is a linked list of `ListNode` objects.

20.4 Linked Lists (cont.)

- Class template `ListNode` (Fig. 20.3) contains `private` members `data` and `nextPtr` (lines 19–20), a constructor to initialize these members and function `getData` to return the data in a node.
- Member `data` stores a value of type `NODETYPE`, the type parameter passed to the class template.
- Member `nextPtr` stores a pointer to the next `ListNode` object in the linked list.

20.4 Linked Lists (cont.)

- Line 13 of the `ListNode` class template definition declares class `List< NODETYPE >` as a `friend`.
- This makes all member functions of a given specialization of class template `List` friends of the corresponding specialization of class template `ListNode`, so they can access the `private` members of `ListNode` objects of that type.
- Because the `ListNode` template parameter `NODETYPE` is used as the template argument for `List` in the `friend` declaration, `ListNode`s specialized with a particular type can be processed only by a `List` specialized with the same type (e.g., a `List` of `int` values manages `ListNode` objects that store `int` values).

20.4 Linked Lists (cont.)

- Lines 23–24 of the `List` class template (Fig. 20.4) declare `private` data members `firstPtr` (a pointer to the first `ListNode` in a `List`) and `lastPtr` (a pointer to the last `ListNode` in a `List`).
- The default constructor (lines 31–36) initializes both pointers to `0` (null).
- The destructor (lines 39–59) ensures that all `ListNode` objects in a `List` object are destroyed when that `List` object is destroyed.

```
1 // Fig. 20.3: ListNode.h
2 // Template ListNode class definition.
3 #ifndef LISTNODE_H
4 #define LISTNODE_H
5
6 // forward declaration of class List required to announce that class
7 // List exists so it can be used in the friend declaration at line 13
8 template< typename NODETYPE > class List;
9
10 template< typename NODETYPE >
11 class ListNode
12 {
13    friend class List< NODETYPE >; // make List a friend
14
15 public:
16    ListNode( const NODETYPE & ); // constructor
17    NODETYPE getData() const; // return data in node
18 private:
19    NODETYPE data; // data
20    ListNode< NODETYPE > *nextPtr; // next node in list
21 }; // end class ListNode
22
```

Fig. 20.3 | ListNode class-template definition. (Part 1 of 2.)

```
// constructor
template< typename NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
   : data( info ), nextPtr( 0 )
{
   // empty body
} // end ListNode constructor

// return copy of data in node
template< typename NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const
{
   return data;
} // end function getData

#endif
```

Fig. 20.3 | ListNode class-template definition. (Part 2 of 2.)

20.4 Linked Lists (cont.)

- The primary `List` functions are `insertAtFront` (lines 62–74), `insertAtBack` (lines 77–89), `removeFromFront` (lines 92–110) and `removeFromBack` (lines 113–140).
- Function `isEmpty` (lines 143–147) is called a predicate function
  - it does not alter the `List`; rather, it determines whether the `List` is empty (i.e., the pointer to the first node of the `List` is null).
  - If the `List` is empty, `true` is returned; otherwise, `false` is returned.
- Function `print` (lines 158–178) displays the `List`'s contents.
- Utility function `getNewNode` (lines 150–155) returns a dynamically allocated `ListNode` object.
  - Called from functions `insertAtFront` and `insertAtBack`.

Error-Prevention Tip 20.1

Assign null (0) to the link member of a new node. Pointers must be initialized before they're used.

```
// Fig. 20.4: List.h
// Template List class definition.
#ifndef LIST_H
#define LIST_H

#include <iostream>
#include "ListNode.h" // ListNode class definition
using namespace std;

```

Fig. 20.4 | `List` class-template definition. (Part 1 of 10.)

```
template< typename 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

   // utility function to allocate new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
}; // end class List

```

Fig. 20.4 | List class-template definition. (Part 2 of 10.)

```
// default constructor
template< typename NODETYPE >
List< NODETYPE >::List()
   : firstPtr( 0 ), lastPtr( 0 )
{
   // empty body
} // end List constructor

```

Fig. 20.4 | `List` class-template definition. (Part 3 of 10.)

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

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

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

   cout << "All nodes destroyed\n\n";
} // end List destructor

```

Fig. 20.4 | List class-template definition. (Part 4 of 10.)

```
// insert node at front of list
template< typename NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      newPtr->nextPtr = firstPtr; // point new node to previous 1st node
      firstPtr = newPtr; // aim firstPtr at new node
   } // end else
} // end function insertAtFront

```

Fig. 20.4 | List class-template definition. (Part 5 of 10.)

```
// insert node at back of list
template< typename NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      lastPtr->nextPtr = newPtr; // update previous last node
      lastPtr = newPtr; // new last node
   } // end else
} // end function insertAtBack

```

Fig. 20.4 | List class-template definition. (Part 6 of 10.)

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

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
         firstPtr = firstPtr->nextPtr; // point to previous 2nd node

      value = tempPtr->data; // return data being removed
      delete tempPtr; // reclaim previous front node
      return true; // delete successful
   } // end else
} // end function removeFromFront

```

Fig. 20.4 | List class-template definition. (Part 7 of 10.)

```
// delete node from back of list
template< typename NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode< NODETYPE > *tempPtr = lastPtr; // hold tempPtr to delete

      if ( firstPtr == lastPtr ) // List has one element
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
      {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         // locate second-to-last element
         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr; // move to next node

         lastPtr = currentPtr; // remove last node
         currentPtr->nextPtr = 0; // this is now the last node
      } // end else

```

Fig. 20.4 | List class-template definition. (Part 8 of 10.)

```
      value = tempPtr->data; // return value from old last node
      delete tempPtr; // reclaim former last node
      return true; // delete successful
   } // end else
} // end function removeFromBack

// is List empty?
template< typename NODETYPE >
bool List< NODETYPE >::isEmpty() const
{
   return firstPtr == 0;
} // end function isEmpty

// return pointer to newly allocated node
template< typename NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
   const NODETYPE &value )
{
   return new ListNode< NODETYPE >( value );
} // end function getNewNode

```

Fig. 20.4 | List class-template definition. (Part 9 of 10.)

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

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

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

   cout << "\n\n";
} // end function print

#endif
```

Fig. 20.4 | List class-template definition. (Part 10 of 10.)

20.4 Linked Lists (cont.)

- In Fig. 20.5, Lines 69 and 73 create `List` objects for types `int` and `double`, respectively.
- Lines 70 and 74 invoke the `testList` function template to manipulate objects.

```cpp
// Fig. 20.5: Fig20_05.cpp
// List class test program.
#include <iostream>
#include <string>
#include "List.h" // List class definition
using namespace std;

// display program instructions to user
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";
} // end function instructions

```

Fig. 20.5 | Manipulating a linked list. (Part 1 of 8.)

```
// function to test a List
template< typename T >
void testList( List< T > &listObject, const string &typeName )
{
   cout << "Testing a List of " << typeName << " values\n";
   instructions(); // display instructions

   int choice; // store user choice
   T value; // store input value

   do // perform user-selected actions
   {
      cout << "? ";
      cin >> choice;

      switch ( choice )
      {
         case 1: // insert at beginning
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtFront( value );
            listObject.print();
            break;
```

Fig. 20.5 | Manipulating a linked list. (Part 2 of 8.)

```
         case 2: // insert at end
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtBack( value );
            listObject.print();
            break;
         case 3: // remove from beginning
            if ( listObject.removeFromFront( value ) )
               cout << value << " removed from list\n";

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

            listObject.print();
            break;
      } // end switch
   } while ( choice < 5 ); // end do...while

   cout << "End list test\n\n";
} // end function testList

```

Fig. 20.5 | Manipulating a linked list. (Part 3 of 8.)

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

   // test List of double values
   List< double > doubleList;
   testList( doubleList, "double" );
} // end main
```

Fig. 20.5 | Manipulating a linked list. (Part 4 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
```

Fig. 20.5 | Manipulating a linked list. (Part 5 of 8.)

```
? 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. 20.5 | Manipulating a linked list. (Part 6 of 8.)

```
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
  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
```

Fig. 20.5 | Manipulating a linked list. (Part 7 of 8.)

```
? 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. 20.5 | Manipulating a linked list. (Part 8 of 8.)

20.4 Linked Lists (cont.)

- Function `insertAtFront` (Fig. 20.4, lines 62–74) places a new node at the front of the list.
- The function consists of several steps:
  - Call function `getNewNode` (line 65), passing it `value`, which is a constant reference to the node value to be inserted.
  - Function `getNewNode` (lines 150–155) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtFront` (line 65).
  - If the list is empty (line 67), `firstPtr` and `lastPtr` are set to `newPtr` (line 68).
  - If the list is not empty (line 69), then the node pointed to by `newPtr` is threaded into the list by copying `firstPtr` to `newPtr->nextPtr` (line 71), so that the new node points to what used to be the first node of the list, and copying `newPtr` to `firstPtr` (line 72), so that `firstPtr` now points to the new first node of the list.

20.4 Linked Lists (cont.)

- Figure 20.6 illustrates function `insertAtFront`.
- Part (a) shows the list and the new node before calling `insertAtFront`.
- The dashed arrows in part (b) illustrate *Step* 4 of the `insertAtFront` operation that enables the node containing `12` to become the new list front.

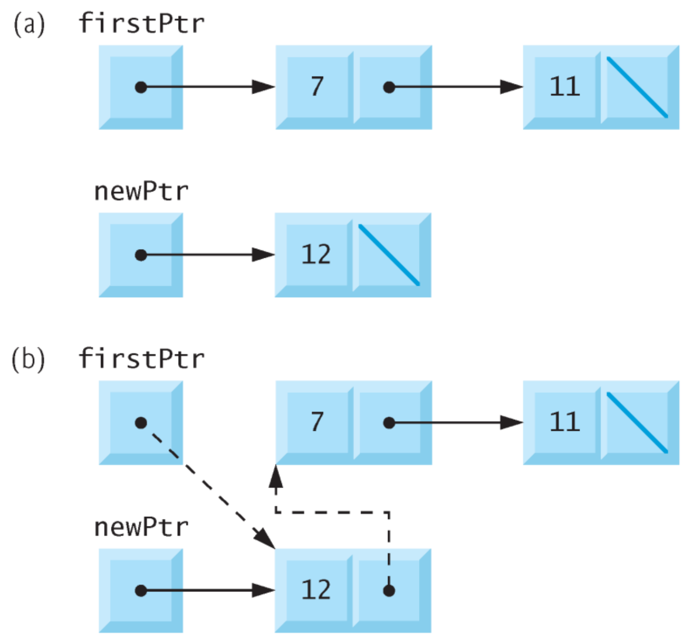

Fig. 20.6 | Operation `insertAtFront` represented graphically.

20.4 Linked Lists (cont.)

- Function `insertAtBack` (Fig. 20.4, lines 77–89) places a new node at the back of the list.
- The function consists of several steps:
  - Call function `getNewNode` (line 80), passing it `value`, which is a constant reference to the node value to be inserted.
  - Function `getNewNode` (lines 150–155) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtBack` (line 80).
  - If the list is empty (line 82), then both `firstPtr` and `lastPtr` are set to `newPtr` (line 83).
  - If the list is not empty (line 84), then the node pointed to by `newPtr` is threaded into the list by copying `newPtr` into `lastPtr->nextPtr` (line 86), so that the new node is pointed to by what used to be the last node of the list, and copying `newPtr` to `lastPtr` (line 87), so that `lastPtr` now points to the new last node of the list.

20.4 Linked Lists (cont.)

- Figure 20.7 illustrates an `insertAtBack` operation.
- Part (a) of the figure shows the list and the new node before the operation.
- The dashed arrows in part (b) illustrate *Step 4* of function `insertAtBack` that enables a new node to be added to the end of a list that is not empty.

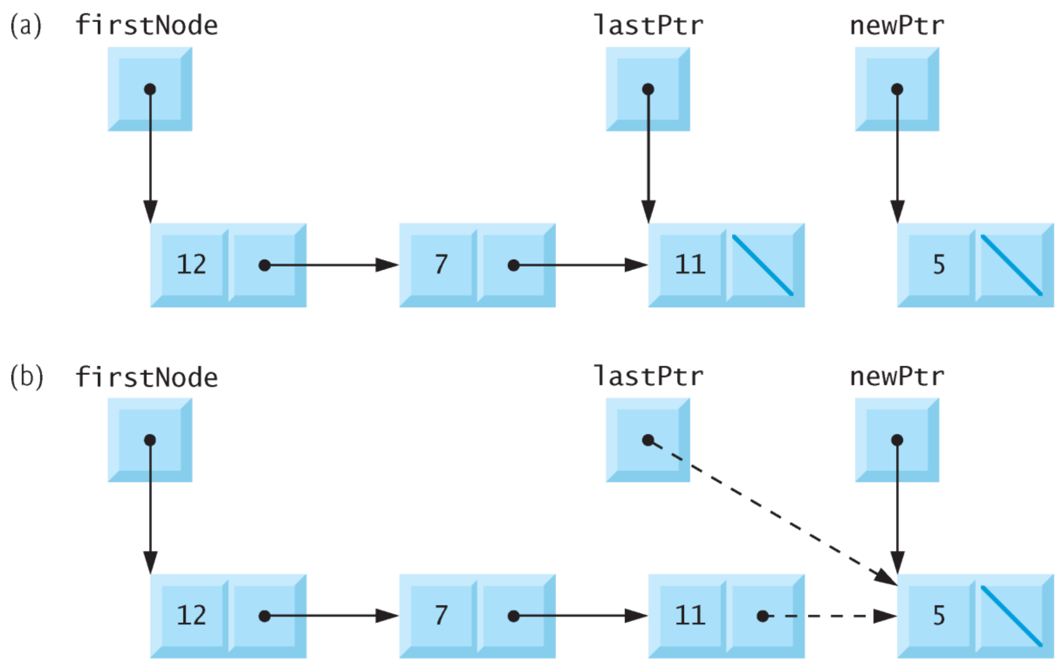

Fig. 20.7 | Operation `insertAtBack` represented graphically.

20.4 Linked Lists (cont.)

- Function `removeFromFront` (Fig. 20.4, lines 92–110) removes the front node of the list and copies the node value to the reference parameter.
- The function returns `false` if an attempt is made to remove a node from an empty list (lines 95–96) and returns `true` if the removal is successful.
- The function consists of several steps:
  - Assign `tempPtr` the address to which `firstPtr` points (line 99). Eventually, `tempPtr` will be used to delete the node being removed.
  - If `firstPtr` is equal to `lastPtr` (line 101), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to zero (line 102) to dethread that node from the list (leaving the list empty).

20.4 Linked Lists (cont.)

- Steps continued:
  - If the list has more than one node prior to removal, then leave `lastPtr` as is and set `firstPtr` to `firstPtr->nextPtr` (line 104); i.e., modify `firstPtr` to point to what was the second node prior to removal (and is now the new first node).
  - After all these pointer manipulations are complete, copy to reference parameter `value` the `data` member of the node being removed (line 106).
  - Now `delete` the node pointed to by `tempPtr` (line 107).
  - Return `true`, indicating successful removal (line 108).

20.4 Linked Lists (cont.)

- Figure 20.8 illustrates function `removeFromFront`.
- Part (a) illustrates the list before the removal operation.
- Part (b) shows the actual pointer manipulations for removing the front node from a nonempty list.

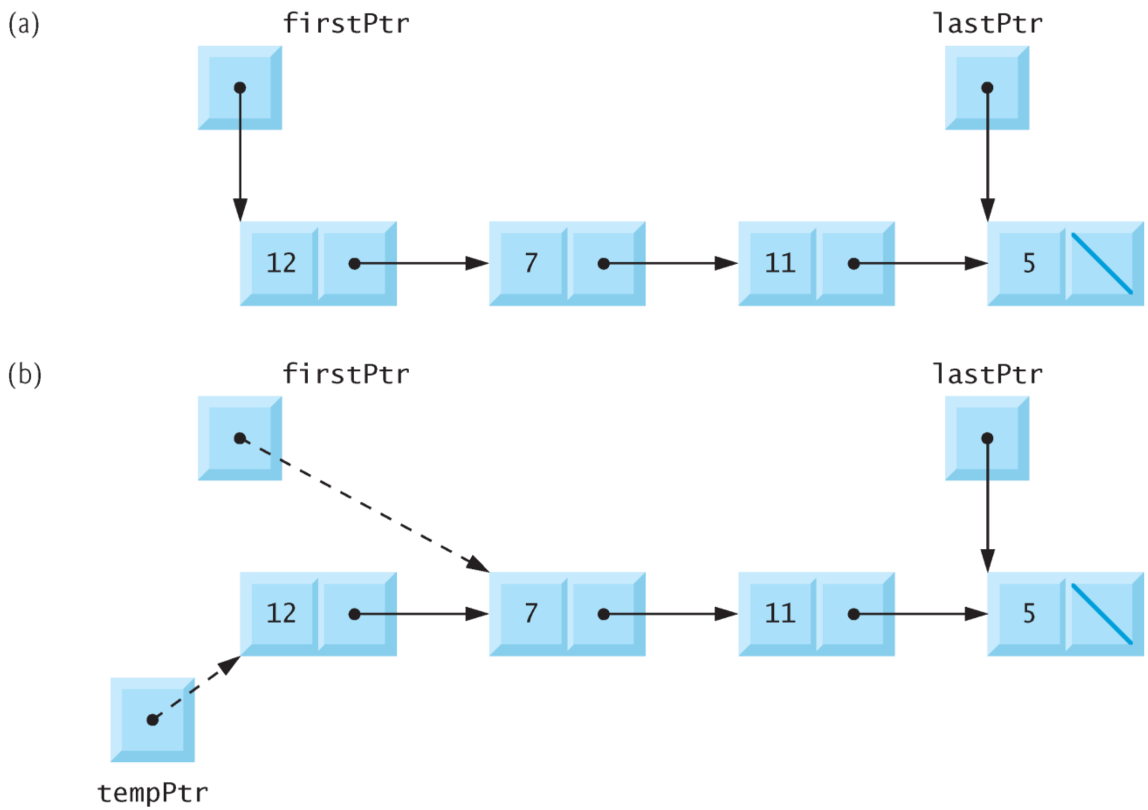

Fig. 20.8 | Operation `removeFromFront` represented graphically.

20.4 Linked Lists (cont.)

- Function `removeFromBack` (Fig. 20.4, lines 113–140) removes the back node of the list and copies the node value to the reference parameter.
- The function returns `false` if an attempt is made to remove a node from an empty list (lines 116–117) and returns `true` if the removal is successful.
- The function consists of several steps:
  - Assign to `tempPtr` the address to which `lastPtr` points (line 120). Eventually, `tempPtr` will be used to delete the node being removed.
  - If `firstPtr` is equal to `lastPtr` (line 122), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to zero (line 123) to dethread that node from the list (leaving the list empty).
  - If the list has more than one node prior to removal, then assign `currentPtr` the address to which `firstPtr` points (line 126) to prepare to "walk the list."

20.4 Linked Lists (cont.)

- Steps continued:
  - Now “walk the list” with `currentPtr` until it points to the node before the last node. This node will become the last node after the remove operation completes. This is done with a `while` loop (lines 129–130) that keeps replacing `currentPtr` by `currentPtr->nextPtr`, while `currentPtr->nextPtr` is not `lastPtr`.
  - Assign `lastPtr` to the address to which `currentPtr` points (line 132) to dethread the back node from the list.
  - Set `currentPtr->nextPtr` to zero (line 133) in the new last node of the list.
  - After all the pointer manipulations are complete, copy to reference parameter `value` the `data` member of the node being removed (line 136).
  - `delete` the node pointed to by `tempPtr` (line 137).
  - Return `true` (line 138), indicating successful removal.

20.4 Linked Lists (cont.)

- Figure 20.9 illustrates `removeFromBack`.
- Part (a) of the figure illustrates the list before the removal operation.
- Part (b) of the figure shows the actual pointer manipulations.

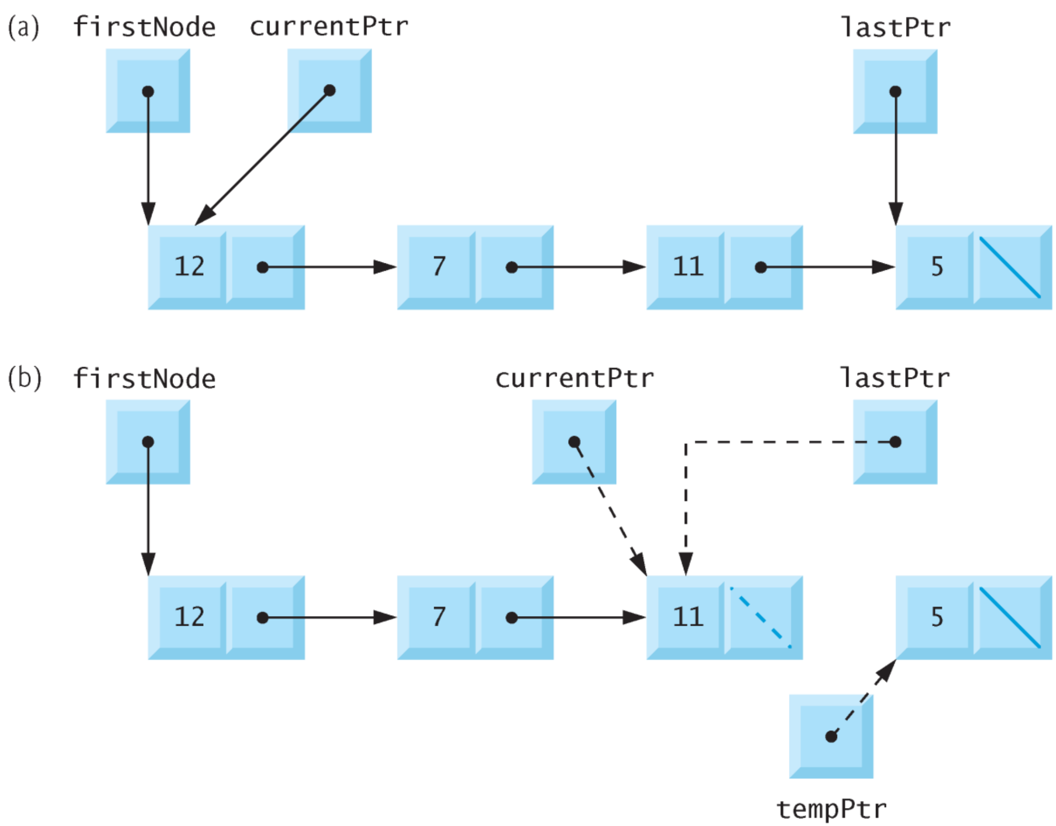

Fig. 20.9 | Operation removeFromBack represented graphically.

20.4 Linked Lists (cont.)

- Function `print` (lines 158–178) first determines whether the list is empty (line 161).
- If so, it prints `"The list is empty"` and returns (lines 163–164).
- Otherwise, it iterates through the list and outputs the value in each node.
- The function initializes `currentPtr` as a copy of `firstPtr` (line 167), then prints the string `"The list is: "` (line 169).
- While `currentPtr` is not null (line 171), `currentPtr->data` is printed (line 173) and `currentPtr` is assigned the value of `currentPtr->nextPtr` (line 174).
- Note that if the link in the last node of the list is not null, the printing algorithm will erroneously attempt to print past the end of the list.
- The printing algorithm is identical for linked lists, stacks and queues (because we base each of these data structures on the same linked list infrastructure).

20.4 Linked Lists (cont.)

- The kind of linked list we've been discussing is a singly linked list—the list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence."
- This list terminates with a node whose pointer member has the value 0.
- A singly linked list may be traversed in only one direction.
- A circular, singly linked list (Fig. 20.10) begins with a pointer to the first node, and each node contains a pointer to the next node.
- The "last node" does not contain a 0 pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."

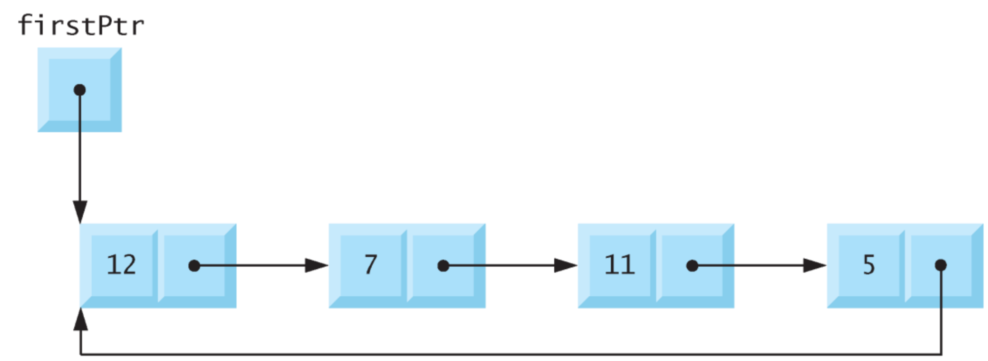

Fig. 20.10 | Circular, singly linked list.

20.4 Linked Lists (cont.)

- A doubly linked list (Fig. 20.11) allows traversals both forward and backward.
- Such a list is often implemented with two “start pointers”—one that points to the first element of the list to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal.
- Each node has both a forward pointer to the next node in the list in the forward direction and a backward pointer to the next node in the list in the backward direction.
- If your list contains an alphabetized telephone directory, for example, a search for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list.
- Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

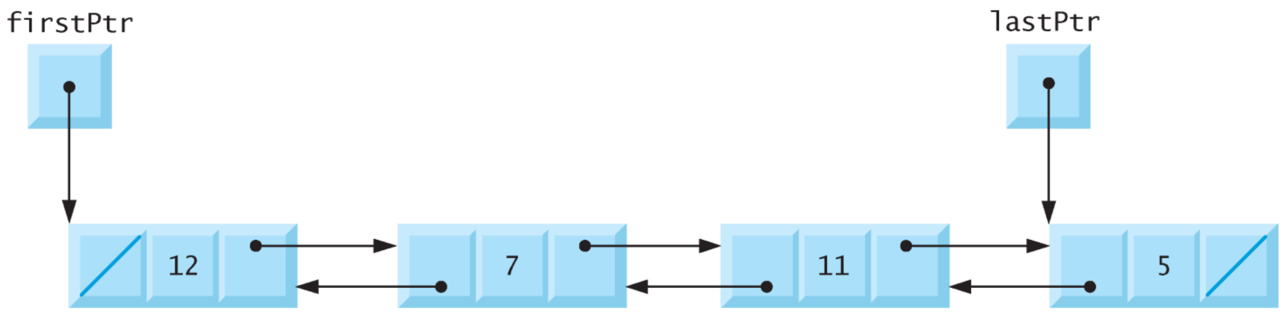

Fig. 20.11 | Doubly linked list.

20.4 Linked Lists (cont.)

- In a circular, doubly linked list (Fig. 20.12), the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the "circle."

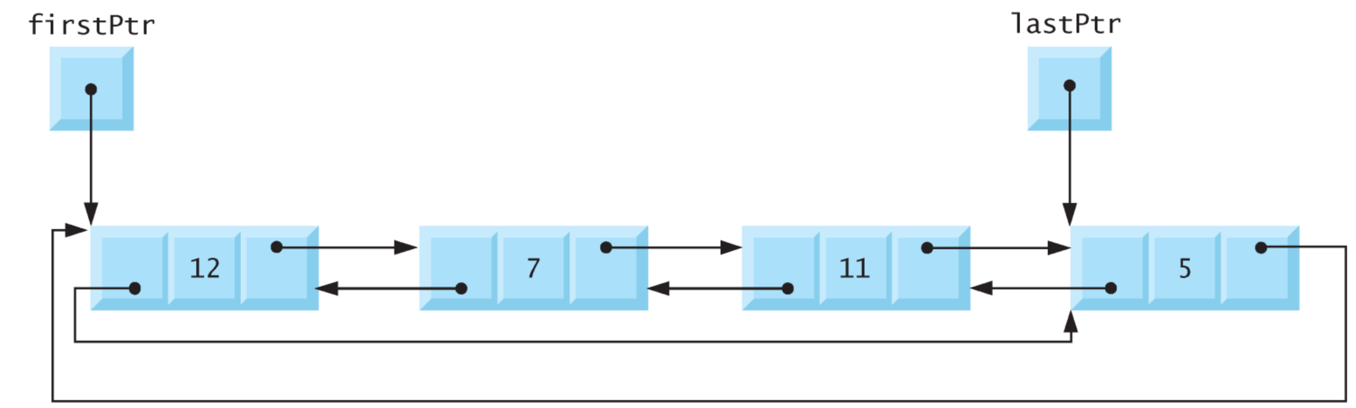

Fig. 20.12 | Circular, doubly linked list.

20.5 Stacks

- Chapter 14, Templates, explained the notion of a stack class template with an underlying array implementation.
- In this section, we use an underlying pointer-based linked-list implementation.
  - We also discuss stacks in Chapter 22, Standard Template Library (STL).
- A stack data structure allows nodes to be added to the stack and removed from the stack only at the top.
- For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.

20.5 Stacks (cont.)

- One way to implement a stack is as a constrained version of a linked list.
- In such an implementation, the link member in the last node of the stack is set to null (zero) to indicate the bottom of the stack.
- The primary member functions used to manipulate a stack are `push` and `pop`.
- Function `push` inserts a new node at the top of the stack.
- Function `pop` removes a node from the top of the stack, stores the popped value in a reference variable that is passed to the calling function and returns `true` if the `pop` operation was successful (`false` otherwise).

20.5 Stacks (cont.)

- Stacks have many interesting applications.
- For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack.
- If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its caller.
- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
- Section 6.11 discusses the function call stack in detail.
- Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function.
- When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's automatic variables is popped off the stack and those variables are no longer known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

20.5 Stacks (cont.)

- We’ll take advantage of the close relationship between lists and stacks to implement a stack class primarily by reusing a list class.
- First, we implement the stack class through `private` inheritance of the list class.
- Then we implement an identically performing stack class through composition by including a list object as a `private` member of a stack class.
- All of the data structures in this chapter, including these two stack classes, are implemented as templates to encourage further reusability.

20.5 Stacks (cont.)

- The program of Figs. 20.13–20.14 creates a `Stack` class template (Fig. 20.13) primarily through `private` inheritance (line 9) of the `List` class template of Fig. 20.4.
- We want the `Stack` to have member functions `push` (lines 13–16), `pop` (lines 19–22), `isStackEmpty` (lines 25–28) and `printStack` (lines 31–34).
  - These are essentially the `insertAtFront`, `removeFromFront`, `isEmpty` and `print` functions of the `List` class template.

20.5 Stacks (cont.)

- Of course, the `List` class template contains other member functions (i.e., `insertAtBack` and `removeFromBack`) that we would not want to make accessible through the `public` interface to the `Stack` class.
- So when we indicate that the `Stack` class template is to inherit from the `List` class template, we specify `private` inheritance.
- This makes all the `List` class template's member functions `private` in the `Stack` class template.
- When we implement the `Stack`'s member functions, we then have each of these call the appropriate member function of the `List` class—`push` calls `insertAtFront` (line 15), `pop` calls `removeFromFront` (line 21), `isStackEmpty` calls `isEmpty` (line 27) and `printStack` calls `print` (line 33)—this is referred to as delegation.

```
// Fig. 20.13: Stack.h
// Template Stack class definition derived from class List.
#ifndef STACK_H
#define STACK_H

#include "List.h" // List class definition

template< typename STACKTYPE >
class Stack : private List< STACKTYPE >
{
public:
   // push calls the List function insertAtFront
   void push( const STACKTYPE &data )
   {
      insertAtFront( data );
   } // end function push

   // pop calls the List function removeFromFront
   bool pop( STACKTYPE &data )
   {
      return removeFromFront( data );
   } // end function pop

```

Fig. 20.13 | Stack class-template definition. (Part 1 of 2.)

```
   // isStackEmpty calls the List function isEmpty
   bool isStackEmpty() const
   {
      return this->isEmpty();
   } // end function isStackEmpty

   // printStack calls the List function print
   void printStack() const
   {
      this->print();
   } // end function print
}; // end class Stack

#endif
```

Fig. 20.13 | Stack class-template definition. (Part 2 of 2.)

20.5 Stacks (cont.)

- The explicit use of `this` on lines 27 and 33 is required so the compiler can resolve identifiers in template definitions.
- A dependent name is an identifier that depends on a template parameter.
- For example, the call to `removeFromFront` (line 21) depends on the argument `data` which has a type that is dependent on the template parameter `STACKTYPE`.
- Resolution of dependent names occurs when the template is instantiated.

20.5 Stacks (cont.)

- In contrast, the identifier for a function that takes no arguments like `isEmpty` or `print` in the `List` superclass is a non-dependent name.
- Such identifiers are normally resolved at the point where the template is defined.
- If the template has not yet been instantiated, then the code for the function with the non-dependent name does not yet exist and some compilers will generate compilation errors.
- Adding the explicit use of `this->` in lines 27 and 33 makes the calls to the base class's member functions dependent on the template parameter and ensures that the code will compile properly.

20.5 Stacks (cont.)

- The stack class template is used in `main` (Fig. 20.14) to instantiate integer stack `intStack` of type `Stack< int >` (line 9).
- Integers 0 through 2 are pushed onto `intStack` (lines 14–18), then popped off `intStack` (lines 23–28).
- The program uses the `Stack` class template to create `doubleStack` of type `Stack< double >` (line 30).
- Values 1.1, 2.2 and 3.3 are pushed onto `doubleStack` (lines 36–41), then popped off `doubleStack` (lines 46–51).

```cpp
// Fig. 20.14: Fig20_14.cpp
// Template Stack class test program.
#include <iostream>
#include "Stack.h" // Stack class definition
using namespace std;

int main()
{
   Stack< int > intStack; // create Stack of ints

   cout << "processing an integer Stack" << endl;

   // push integers onto intStack
   for ( int i = 0; i < 3; ++i )
   {
      intStack.push( i );
      intStack.printStack();
   } // end for

   int popInteger; // store int popped from stack

```

Fig. 20.14 | A simple stack program. (Part 1 of 4.)

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

   Stack< double > doubleStack; // create Stack of doubles
   double value = 1.1;

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

   // push floating-point values onto doubleStack
   for ( int j = 0; j < 3; ++j )
   {
      doubleStack.push( value );
      doubleStack.printStack();
      value += 1.1;
   } // end for

   double popDouble; // store double popped from stack

```

Fig. 20.14 | A simple stack program. (Part 2 of 4.)

```
    // pop floating-point values from doubleStack
    while ( !doubleStack.isStackEmpty() )
    {
       doubleStack.pop( popDouble );
       cout << popDouble << " popped from stack" << endl;
       doubleStack.printStack();
    } // end while
} // end 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
```

Fig. 20.14 | A simple stack program. (Part 3 of 4.)

```
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
```

Fig. 20.14 | A simple stack program. (Part 4 of 4.)

20.5 Stacks (cont.)

- Another way to implement a `Stack` class template is by reusing the `List` class template through composition.
- Figure 20.15 is a new implementation of the `Stack` class template that contains a `List< STACKTYPE >` object called `stackList` (line 38).
- This version of the `Stack` class template uses class `List` from Fig. 20.4.
- To test this class, use the driver program in Fig. 20.14, but include the new header—`Stackcomposition.h` in line 6 of that file.
- The output of the program is identical for both versions of class `Stack`.

```
// Fig. 20.15: Stackcomposition.h
// Template Stack class definition with composed List object.
#ifndef STACKCOMPOSITION_H
#define STACKCOMPOSITION_H

#include "List.h" // List class definition

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

   // push calls stackList object's insertAtFront member function
   void push( const STACKTYPE &data )
   {
      stackList.insertAtFront( data );
   } // end function push

```

Fig. 20.15 | Stack class template with a composed `List` object. (Part 1 of 2.)

```
    // pop calls stackList object's removeFromFront member function
    bool pop( STACKTYPE &data )
    {
       return stackList.removeFromFront( data );
    } // end function pop

    // isStackEmpty calls stackList object's isEmpty member function
    bool isStackEmpty() const
    {
       return stackList.isEmpty();
    } // end function isStackEmpty

    // printStack calls stackList object's print member function
    void printStack() const
    {
       stackList.print();
    } // end function printStack
private:
    List< STACKTYPE > stackList; // composed List object
}; // end class Stack

#endif
```

Fig. 20.15 | Stack class template with a composed `List` object. (Part 2 of 2.)

20.6 Queues

- A queue is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced.
- Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue.
- For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.
- The insert and remove operations are known as enqueue and dequeue.

20.6 Queues (cont.)

- Queues have many applications in computer systems.
- Computers that have a single processor can service only one user at a time.
- Entries for the other users are placed in a queue.
- Each entry gradually advances to the front of the queue as users receive service.
- The entry at the front of the queue is the next to receive service.

20.6 Queues (cont.)

- Queues are also used to support print spooling.
- For example, a single printer might be shared by all users of a network.
- Many users can send print jobs to the printer, even when the printer is already busy.
- These print jobs are placed in a queue until the printer becomes available.
- A program called a spooler manages the queue to ensure that, as each print job completes, the next print job is sent to the printer.

20.6 Queues (cont.)

- Information packets also wait in queues in computer networks.
- Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet's final destination.
- The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.
- A file server in a computer network handles file access requests from many clients throughout the network.
- Servers have a limited capacity to service requests from clients.
- When that capacity is exceeded, client requests wait in queues.

20.6 Queues (cont.)

- The program of Figs. 20.16–20.17 creates a `Queue` class template (Fig. 20.16) through `private` inheritance (line 9) of the `List` class template (Fig. 20.4).
- The `Queue` has member functions `enqueue` (lines 13–16), `dequeue` (lines 19–22), `isQueueEmpty` (lines 25–28) and `printQueue` (lines 31–34).
- These are essentially the `insertAtBack`, `removeFromFront`, `isEmpty` and `print` functions of the `List` class template.

20.6 Queues (cont.)

- The `List` class template contains other member functions that we do not want to make accessible through the `public` interface to the `Queue` class.
- So when we indicate that the `Queue` class template is to inherit the `List` class template, we specify `private` inheritance.
- This makes all the `List` class template's member functions `private` in the `Queue` class template.
- When we implement the `Queue`'s member functions, we have each of these call the appropriate member function of the list class—`enqueue` calls `insertAtBack` (line 15), `dequeue` calls `removeFromFront` (line 21), `isQueueEmpty` calls `isEmpty` (line 27) and `printQueue` calls `print` (line 33).
- As with the `Stack` example in Fig. 20.13, this delegation requires explicit use of the `this` pointer in `isQueueEmpty` and `printQueue` to avoid compilation errors.

```
// Fig. 20.16: Queue.h
// Template Queue class definition derived from class List.
#ifndef QUEUE_H
#define QUEUE_H

#include "List.h" // List class definition

template< typename QUEUETYPE >
class Queue : private List< QUEUETYPE >
{
public:
   // enqueue calls List member function insertAtBack
   void enqueue( const QUEUETYPE &data )
   {
      insertAtBack( data );
   } // end function enqueue

   // dequeue calls List member function removeFromFront
   bool dequeue( QUEUETYPE &data )
   {
      return removeFromFront( data );
   } // end function dequeue

```

Fig. 20.16 | Queue class-template definition. (Part 1 of 2.)

```
   // isQueueEmpty calls List member function isEmpty
   bool isQueueEmpty() const
   {
      return this->isEmpty();
   } // end function isQueueEmpty

   // printQueue calls List member function print
   void printQueue() const
   {
      this->print();
   } // end function printQueue
}; // end class Queue

#endif
```

Fig. 20.16 | Queue class-template definition. (Part 2 of 2.)

20.6 Queues (cont.)

- Figure 20.17 uses the `Queue` class template to instantiate integer queue `intQueue` of type `Queue< int >` (line 9).
- Integers 0 through 2 are enqueued to `intQueue` (lines 14–18), then dequeued from `intQueue` in first-in, first-out order (lines 23–28).
- Next, the program instantiates queue `doubleQueue` of type `Queue< double >` (line 30).
- Values 1.1, 2.2 and 3.3 are enqueued to `doubleQueue` (lines 36–41), then dequeued from `doubleQueue` in first-in, first-out order (lines 46–51).

```cpp
// Fig. 20.17: Fig20_17.cpp
// Template Queue class test program.
#include <iostream>
#include "Queue.h" // Queue class definition
using namespace std;

int main()
{
   Queue< int > intQueue; // create Queue of integers

   cout << "processing an integer Queue" << endl;

   // enqueue integers onto intQueue
   for ( int i = 0; i < 3; ++i )
   {
      intQueue.enqueue( i );
      intQueue.printQueue();
   } // end for

   int dequeueInteger; // store dequeued integer

```

Fig. 20.17 | Queue-processing program. (Part 1 of 5.)

```
   // dequeue integers from intQueue
   while ( !intQueue.isQueueEmpty() )
   {
      intQueue.dequeue( dequeueInteger );
      cout << dequeueInteger << " dequeued" << endl;
      intQueue.printQueue();
   } // end while

   Queue< double > doubleQueue; // create Queue of doubles
   double value = 1.1;

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

   // enqueue floating-point values onto doubleQueue
   for ( int j = 0; j < 3; ++j )
   {
      doubleQueue.enqueue( value );
      doubleQueue.printQueue();
      value += 1.1;
   } // end for

   double dequeueDouble; // store dequeued double

```

Fig. 20.17 | Queue-processing program. (Part 2 of 5.)

```
   // dequeue floating-point values from doubleQueue
   while ( !doubleQueue.isQueueEmpty() )
   {
      doubleQueue.dequeue( dequeueDouble );
      cout << dequeueDouble << " dequeued" << endl;
      doubleQueue.printQueue();
   } // end while
} // end main
```

Fig. 20.17 | Queue-processing program. (Part 3 of 5.)

```
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
```

Fig. 20.17 | Queue-processing program. (Part 4 of 5.)

```
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
```

Fig. 20.17 | Queue-processing program. (Part 5 of 5.)

20.7 Trees

- Linked lists, stacks and queues are linear data structures.
- A tree is a nonlinear, two-dimensional data structure.
- Tree nodes contain two or more links.
- This section discusses binary trees (Fig. 20.18)—trees whose nodes all contain two links (none, one or both of which may be null).

20.7 Trees (cont.)

- For this discussion, refer to nodes A, B, C and D in Fig. 20.18.
- The root node (node B) is the first node in a tree.
- Each link in the root node refers to a child (nodes A and D).
- The left child (node A) is the root node of the left subtree (which contains only node A), and the right child (node D) is the root node of the right subtree (which contains nodes D and C).
- The children of a given node are called siblings (e.g., nodes A and D are siblings).
- A node with no children is a leaf node (e.g., nodes A and C are leaf nodes).
- Computer scientists normally draw trees from the root node down—the opposite of how trees grow in nature.

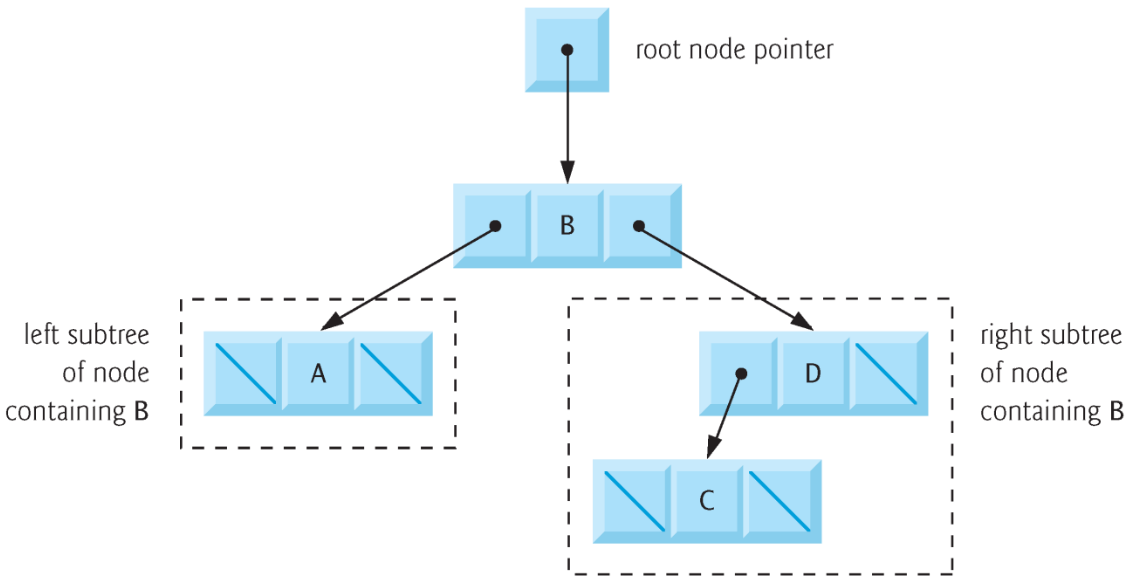

Fig. 20.18 | A graphical representation of a binary tree.

20.7 Trees (cont.)

- A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node.
- Figure 20.19 illustrates a binary search tree with 9 values.
- Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

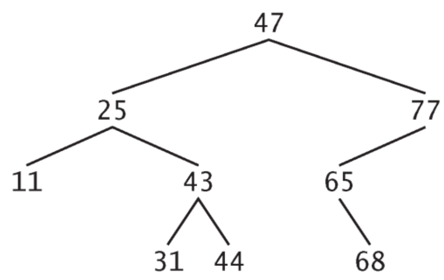

Fig. 20.19 | A binary search tree.

20.7 Trees (cont.)

- The program of Figs. 20.20–20.22 creates a binary search tree and traverses it (i.e., walks through all its nodes) three ways—using recursive inorder, preorder and postorder traversals.
- We explain these traversal algorithms shortly.
- We begin our discussion with the driver program (Fig. 20.22), then continue with the implementations of classes `TreeNode` (Fig. 20.20) and `Tree` (Fig. 20.21).
- Function `main` (Fig. 20.22) begins by instantiating integer tree `intTree` of type `Tree< int >` (line 10).
- The program prompts for 10 integers, each of which is inserted in the binary tree by calling `insertNode` (line 19).
- The program then performs preorder, inorder and postorder traversals (these are explained shortly) of `intTree` (lines 23, 26 and 29, respectively).

20.7 Trees (cont.)

- The program then instantiates floating-point tree `doubleTree` of type `Tree< double >` (line 31).
- The program prompts for 10 `double` values, each of which is inserted in the binary tree by calling `insertNode` (line 41).
- The program then performs preorder, inorder and postorder traversals of `doubleTree` (lines 45, 48 and 51, respectively).

```
// Fig. 20.20: TreeNode.h
// Template TreeNode class definition.
#ifndef TREENODE_H
#define TREENODE_H

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

// TreeNode class-template definition
template< typename NODETYPE >
class TreeNode
{
   friend class Tree< NODETYPE >;
public:
   // constructor
   TreeNode( const NODETYPE &d )
      : leftPtr( 0 ), // pointer to left subtree
        data( d ), // tree node data
        rightPtr( 0 ) // pointer to right substree
   {
      // empty body
   } // end TreeNode constructor

```

Fig. 20.20 | TreeNode class-template definition. (Part 1 of 2.)

```
   // return copy of node's data
   NODETYPE getData() const
   {
      return data;
   } // end getData function
private:
   TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
   NODETYPE data;
   TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
}; // end class TreeNode

#endif
```

Fig. 20.20 | TreeNode class-template definition. (Part 2 of 2.)

```
// Fig. 20.21: Tree.h
// Template Tree class definition.
#ifndef TREE_H
#define TREE_H

#include <iostream>
#include "TreeNode.h"
using namespace std;

```

Fig. 20.21 | Tree class-template definition. (Part 1 of 7.)

```
// Tree class-template definition
template< typename NODETYPE > class Tree
{
public:
   Tree(); // constructor
   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;
}; // end class Tree

```

Fig. 20.21 | Tree class-template definition. (Part 2 of 7.)

```
// constructor
template< typename NODETYPE >
Tree< NODETYPE >::Tree()
{
   rootPtr = 0; // indicate tree is initially empty
} // end Tree constructor

// insert node in Tree
template< typename NODETYPE >
void Tree< NODETYPE >::insertNode( const NODETYPE &value )
{
   insertNodeHelper( &rootPtr, value );
} // end function insertNode

```

Fig. 20.21 | Tree class-template definition. (Part 3 of 7.)

```
// utility function called by insertNode; receives a pointer
// to a pointer so that the function can modify pointer's value
template< typename NODETYPE >
void Tree< NODETYPE >::insertNodeHelper(
   TreeNode< NODETYPE > **ptr, const NODETYPE &value )
{
   // subtree is empty; create new TreeNode containing value
   if ( *ptr == 0 )
      *ptr = new TreeNode< NODETYPE >( value );
   else // subtree is not empty
   {
      // data to insert is less than data in current node
      if ( value < ( *ptr )->data )
         insertNodeHelper( &( ( *ptr )->leftPtr ), value );
      else
      {
         // data to insert is greater than data in current node
         if ( value > ( *ptr )->data )
            insertNodeHelper( &( ( *ptr )->rightPtr ), value );
         else // duplicate data value ignored
            cout << value << " dup" << endl;
      } // end else
   } // end else
} // end function insertNodeHelper
```

Fig. 20.21 | Tree class-template definition. (Part 4 of 7.)

```
// begin preorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::preOrderTraversal() const
{
   preOrderHelper( rootPtr );
} // end function preOrderTraversal

// utility function to perform preorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::preOrderHelper( TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      cout << ptr->data << ' '; // process node
      preOrderHelper( ptr->leftPtr ); // traverse left subtree
      preOrderHelper( ptr->rightPtr ); // traverse right subtree
   } // end if
} // end function preOrderHelper

```

Fig. 20.21 | Tree class-template definition. (Part 5 of 7.)

```
// begin inorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::inOrderTraversal() const
{
   inOrderHelper( rootPtr );
} // end function inOrderTraversal

// utility function to perform inorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::inOrderHelper( TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      inOrderHelper( ptr->leftPtr ); // traverse left subtree
      cout << ptr->data << ' '; // process node
      inOrderHelper( ptr->rightPtr ); // traverse right subtree
   } // end if
} // end function inOrderHelper

```

Fig. 20.21 | Tree class-template definition. (Part 6 of 7.)

```
// begin postorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::postOrderTraversal() const
{
   postOrderHelper( rootPtr );
} // end function postOrderTraversal

// utility function to perform postorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::postOrderHelper(
   TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      postOrderHelper( ptr->leftPtr ); // traverse left subtree
      postOrderHelper( ptr->rightPtr ); // traverse right subtree
      cout << ptr->data << ' '; // process node
   } // end if
} // end function postOrderHelper

#endif
```

Fig. 20.21 | Tree class-template definition. (Part 7 of 7.)

```
// Fig. 20.22: Fig20_22.cpp
// Tree class test program.
#include <iostream>
#include <iomanip>
#include "Tree.h" // Tree class definition
using namespace std;

int main()
{
   Tree< int > intTree; // create Tree of int values
   int intValue;

   cout << "Enter 10 integer values:\n";

   // insert 10 integers to intTree
   for ( int i = 0; i < 10; ++i )
   {
      cin >> intValue;
      intTree.insertNode( intValue );
   } // end for

```

Fig. 20.22 | Creating and traversing a binary tree. (Part 1 of 4.)

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

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

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

   Tree< double > doubleTree; // create Tree of double values
   double doubleValue;

   cout << fixed << setprecision( 1 )
      << "\n\n\nEnter 10 double values:\n";

   // insert 10 doubles to doubleTree
   for ( int j = 0; j < 10; ++j )
   {
      cin >> doubleValue;
      doubleTree.insertNode( doubleValue );
   } // end for

```

Fig. 20.22 | Creating and traversing a binary tree. (Part 2 of 4.)

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

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

    cout << "\nPostorder traversal\n";
    doubleTree.postOrderTraversal();
    cout << endl;
} // end main
```

Fig. 20.22 | Creating and traversing a binary tree. (Part 3 of 4.)

```
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. 20.22 | Creating and traversing a binary tree. (Part 4 of 4.)

20.7 Trees (cont.)

- The `TreeNode` class template (Fig. 20.20) definition declares `Tree<NODETYPE>` as its `friend` (line 13).
  - This makes all member functions of a given specialization of class template `Tree` (Fig. 20.21) friends of the corresponding specialization of class template `TreeNode`, so they can access the `private` members of `TreeNode` objects of that type.
  - Because the `TreeNode` template parameter `NODETYPE` is used as the template argument for `Tree` in the `friend` declaration, `TreeNode`s specialized with a particular type can be processed only by a `Tree` specialized with the same type (e.g., a `Tree` of `int` values manages `TreeNode` objects that store `int` values).

20.7 Trees (cont.)

- Lines 30–32 declare a `TreeNode`'s `private` data—the node's `data` value, and pointers `leftPtr` (to the node's left subtree) and `rightPtr` (to the node's right subtree).
- The constructor (lines 16–22) sets `data` to the value supplied as a constructor argument and sets pointers `leftPtr` and `rightPtr` to zero (thus initializing this node to be a leaf node).
- Member function `getData` (lines 25–28) returns the `data` value.

20.7 Trees (cont.)

- Class template `Tree` (Fig. 20.21) has as `private` data `rootPtr` (line 20), a pointer to the tree's root node.
- Lines 15–18 declare the `public` member functions `insertNode` (that inserts a new node in the tree) and `preOrderTraversal`, `inOrderTraversal` and `postOrderTraversal`, each of which walks the tree in the designated manner.
- Each of these member functions calls its own recursive utility function to perform the appropriate operations on the internal representation of the tree, so the program is not required to access the underlying `private` data to perform these functions.
- Remember that the recursion requires us to pass in a pointer that represents the next subtree to process.

20.7 Trees (cont.)

- The `Tree` constructor initializes `rootPtr` to zero to indicate that the tree is initially empty.
- The `Tree` class's utility function `insertNodeHelper` (lines 45–66) is called by `insertNode` (lines 37–41) to recursively insert a node into the tree.
- *A node can only be inserted as a leaf node in a binary search tree.*
- If the tree is empty, a new `TreeNode` is created, initialized and inserted in the tree (lines 51–52).
- If the tree is not empty, the program compares the value to be inserted with the `data` value in the root node.
- If the insert value is smaller (line 55), the program recursively calls `insertNodeHelper` (line 56) to insert the value in the left subtree.
- If the insert value is larger (line 60), the program recursively calls `insertNodeHelper` (line 61) to insert the value in the right subtree.

20.7 Trees (cont.)

- If the value to be inserted is identical to the data value in the root node, the program prints the message `" dup"` (line 63) and returns without inserting the duplicate value into the tree.
- `insertNode` passes the address of `rootPtr` to `insertNodeHelper` (line 40) so it can modify the value stored in `rootPtr` (i.e., the address of the root node).
- To receive a pointer to `rootPtr` (which is also a pointer), `insertNodeHelper`'s first argument is declared as a pointer to a pointer to a `TreeNode`.

20.7 Trees (cont.)

- Member functions `inOrderTraversal` (lines 88–92), `preOrderTraversal` (lines 69–73) and `postOrderTraversal` (lines 107–111) traverse the tree and print the node values.
- For the purpose of the following discussion, we use the binary search tree in Fig. 20.23.

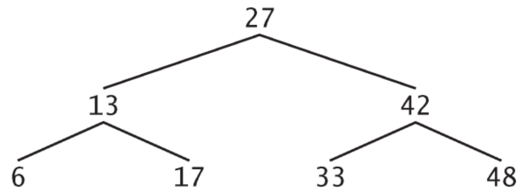

Fig. 20.23 | A binary search tree.

20.7 Trees (cont.)

- Function `inOrderTraversal` invokes utility function `inOrderHelper` to perform the inorder traversal of the binary tree.
- The steps for an inorder traversal are:
  - Traverse the left subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 100.)
  - Process the value in the node—i.e., print the node value (line 101).
  - Traverse the right subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 102.)
- The value in a node is not processed until the values in its left subtree are processed, because each call to `inOrderHelper` immediately calls `inOrderHelper` again with the pointer to the left subtree.

20.7 Trees (cont.)

- The inorder traversal of the tree in Fig. 20.23 is
  - `6 13 17 27 33 42 48`
- Note that the inorder traversal of a binary search tree prints the node values in ascending order.
- The process of creating a binary search tree actually sorts the data—thus, this process is called the binary tree sort.

20.7 Trees (cont.)

- Function `preOrderTraversal` invokes utility function `preOrderHelper` to perform the preorder traversal of the binary tree.
- The steps for an preorder traversal are:
  - Process the value in the node (line 81).
  - Traverse the left subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 82.)
  - Traverse the right subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 83.)
- The value in each node is processed as the node is visited.
- After the value in a given node is processed, the values in the left subtree are processed.
- Then the values in the right subtree are processed.
- The preorder traversal of the tree in Fig. 20.23 is
  - `27 13 6 17 42 33 48`

20.7 Trees (cont.)

- Function `postOrderTraversal` invokes utility function `postOrderHelper` to perform the postorder traversal of the binary tree.
- The steps for a postorder traversal are:
  - Traverse the left subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 120.)
  - Traverse the right subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 121.)
  - Process the value in the node (line 122).
- The value in each node is not printed until the values of its children are printed.
- The `postOrderTraversal` of the tree in Fig. 20.23 is
  - `6 17 13 33 48 42 27`

20.7 Trees (cont.)

- The binary search tree facilitates duplicate elimination.
- As the tree is being created, an attempt to insert a duplicate value will be recognized, because a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did when it was inserted in the tree.
- Thus, the duplicate will eventually be compared with a node containing the same value.
- The duplicate value may be discarded at this point.

20.7 Trees (cont.)

- Searching a binary tree for a value that matches a key value is also fast.
- If the tree is balanced, then each branch contains about half the number of nodes in the tree.
- Each comparison of a node to the search key eliminates half the nodes.
- This is called an O(log n) algorithm (Big O notation is discussed in Chapter 19).
- So a binary search tree with n elements would require a maximum of $\log_2 n$ comparisons either to find a match or to determine that no match exists.
- This means, for example, that when searching a (balanced) 1000-element binary search tree, no more than 10 comparisons need to be made, because $2^{10} > 1000$.
- When searching a (balanced) 1,000,000-element binary search tree, no more than 20 comparisons need to be made, because $2^{20} > 1,000,000$.

20.7 Trees (cont.)

- In the exercises, algorithms are presented for several other binary tree operations such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a level-order traversal of a binary tree.
- The level-order traversal of a binary tree visits the nodes of the tree row by row, starting at the root node level.
- On each level of the tree, the nodes are visited from left to right.
- Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.