

Software Development: Data Structures

(H16Y 35)

Dynamic Data Structures

& Collections

Objectives:

* To be able to form linked data structures using references, self-referential classes and recursion
* To be able to create and manipulate dynamic data structures such as linked lists, queues, stacks and binary trees
* To understand various important applications of linked data structures
* To understand how to create reusable data structures with classes, inheritance and composition

**A direct extract from Chapter 23,”C# How To Program”, Deitel (ISBN 0-13-062221-4)**

Contents Page

Introduction Page 1

Self-Referential Classes Page 2

Sample self-referential Node class definition Page 4

Linked Lists Page 6

Fig. 23.4

Definitions of classes ListNode, List and EmptyListException Page 9

Fig. 23.5 Demonstrating the linked list Page 13

Fig. 23.4 and Fig. 23.5 code walkthroughs Page 15

Stacks Page 20

Fig. 23.10 StackInheritance extends class List Page 21

Fig. 23.11 Using class StackInheritance Page 22

Fig. 23.12

StackComposition class encapsulates functionality of class List Page 23

Fig. 23.10 and Fig. 23.11 code walkthroughs Page 24

Queues Page 27

Fig. 23.13 QueueInheritance extends class List Page 28

Fig. 23.14 Using inheritance to create a queue Page 29

Fig. 23.13 and Fig. 23.14 code walkthroughs Page 30

Binary Trees Page 32

Fig. 23.17

Definitions of TreeNode and Tree for a binary search tree Page 34

Fig. 23.18 Creating and traversing a binary tree Page 38

Fig. 23.17 and Fig. 23.18 code walkthroughs Page 39

Binary Search Tree of IComparable Objects Page 42

Fig. 23.20

Definitions of class TreeNode and Tree

for manipulating IComparable objects Page 43

Fig. 23.21 Demonstrating class Tree with IComparable objects Page 47

Fig. 23.20 and Fig. 23.21 code walkthroughs Page 49

Collection Classes Page 51

Class Array Page 52

Fig. 23.22 Demonstrating the Array class Page 53

Fig. 23.22 code walkthrough Page 55

Class ArrayList Page 56

Fig. 23.23 Demonstrating the ArrayList class Page 57

Fig. 23.24 (code required within Fig. 23.23) Page 60

Fig. 23.23 code walkthrough Page 65

Class Stack Page 66

Fig. 23.25 Using the Stack class Page 67

Fig. 23.25 code walkthrough Page 70

Class Hashtable Page 71

Fig. 23.26 Using the Hashtable class Page 73

Fig. 23.26 code walkthrough Page 77

Dynamic Data Structures & Collections Summary Page 79

Self-Review Exercises Page 83

An Introduction to Dynamic Data Structures and Collections

The data structures that we have studied so far have had fixed size, such as single-and double-subscripted arrays

We now introduce dynamic data structures that grow and shrink at execution time

**Linked Lists** are collections of data items “lined up in a row” - users can make insertions and deletions anywhere in a linked list

**Stacks** are important in compilers and operating systems because insertions and deletions are made at only one end - its top

**Queues** represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue, and deletions 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

These data structures have many other interesting applications as well

We will discuss each of the major types of data structures and implement programs that create and manipulate them

We use classes, inheritance and composition to create and package these data structures for reusability and maintainability

The booklet examples are practical programs that will be useful in more advanced courses and in industrial applications

The programs devote special attention to and focus on reference manipulation

Self-Referential Classes

A self-referential class contains a reference member that refers to an object of the same class type

For example, the class definition shown (see Sample self-referential Node class definition on page 4) defines a type, Node

This type has two private instance variables—integer data and Node reference next

Member next references an object of type Node, an object of the same type as the one being declared here—hence, the term “self-referential class

”Member next is referred to as a link (i.e., next can be used to “tie” an object of type Node to another object of the same type)

Class Node also has two properties: One for variable data (named Data), and another for variable next (named Next)

Self-referential objects can be linked together to form useful data structures, such as lists, queues, stacks and trees

The following illustrates two self-referential objects linked together to form a list

[Figure 23.2](javascript:void(0))

**Two self-referential class objects linked together**

A backslash (representing a null reference) is placed in the link member of the second self-referential object to indicate that the link does not refer to another object

The slash is for illustration purposes; it does not correspond to the backslash character in C#

A null reference normally indicates the end of a data structure

Creating and maintaining dynamic data structures requires dynamic memory allocation—a program’s ability to obtain more memory space at execution time to hold new nodes and to release space no longer needed

As we have already learned, C# programs do not explicitly release dynamically allocated memory

Rather, C# performs automatic garbage collection

The limit for dynamic memory allocation can be as large as the amount of available disk space in a virtual-memory system

Often, the limits are much smaller, because the computer’s available memory must be shared among many users

Operator new is essential to dynamic memory allocation

Operator **new** takes as an operand the type of the object being dynamically allocated and returns a reference to a newly created object of that type

For example, the statement –

**Node nodeToAdd = new Node(10);**

allocates the appropriate amount of memory to store a Node and stores a reference to this object in nodeToAdd

If no memory is available, new throws an OutOfMemoryException

The 10 is the Node object’s data

The following sections discuss lists, stacks, queues and trees

These data structures are created and maintained with dynamic memory allocation and self-referential classes

Sample self-referential Node class definition

class Node   
{   
 private int data;  
 private Node next;

public Node( int d )  
 {  
 // constructor body  
 }

public int Data  
 {  
 get  
 {  
 // get body  
 }

set  
 {  
 // set body  
 }  
 }

public Node Next  
 {  
 get  
 {  
 // get body  
 }

set  
 {  
 // set body  
 }  
 }  
}

Linked Lists

A linked list is a linear collection (i.e., a sequence) of self-referential class objects, called nodes, connected by reference links - hence, the term “linked” list

A program accesses a linked list via a reference to the first node of the list

Each subsequent node is accessed via the link-reference member stored in the previous node

By convention, the link reference in the last node of a list is set to null to mark the end of the list

Data is stored in a linked list dynamically—that is, each node is created as necessary

A node can contain data of any type, including objects of other classes

Stacks and queues are also linear data structures, and they are constrained versions of linked lists

Trees are nonlinear data structures

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 in the data structure is unpredictable

Unlike a linked list, the size of a conventional C# array cannot be altered, because the array size is fixed at creation time

Conventional arrays can become full, but linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests

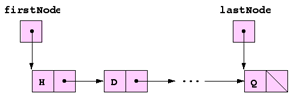
Programmers can maintain linked lists in sorted order simply by inserting each new element at the proper point in the list (locating the proper insertion point does take time)

They do not need to move existing list elements

Memory does not normally store linked list nodes contiguously

Rather, the nodes are logically contiguous

The figure below illustrates a linked list with several nodes

[](javascript:void(0))

**A graphical representation of a linked list**

Program Fig. 23.4 plus Fig. 23.5 demonstrate the use of an object of class List to manipulate a list of miscellaneous object types

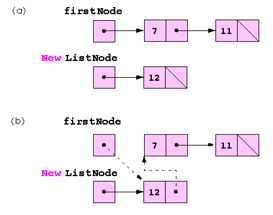
The Main method of class ListTest (Fig. 23.5) creates a list of objects, inserts objects at the beginning of the list using List method InsertAtFront,

inserts objects at the end of the list using List method InsertAtBack,

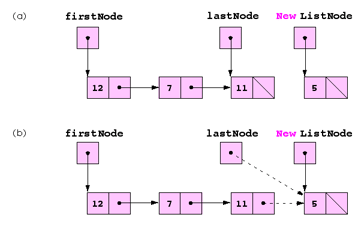
deletes objects from the front of the list using List method RemoveFromFront

and deletes objects from the end of the list using List method RemoveFromBack

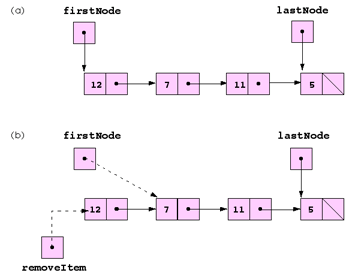
Graphical representations of these List methods are shown below:

[](javascript:void(0))

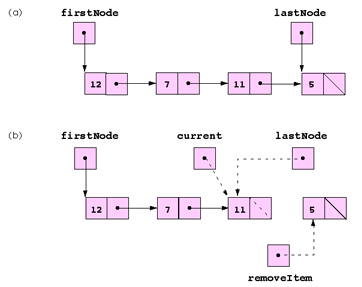
**Operation 1 - InsertAtFront of the list**

[](javascript:void(0))

**Operation 2 - InsertAtBack of the list**

[](javascript:void(0))

**Operation 3 - RemoveFromFront of the list**

[](javascript:void(0))

**Operation 4 - RemoveFromBack of the list**

Each insertion and deletion operation invokes List method Print to display the current list contents

If an attempt is made to remove an item from an empty list, an EmptyListException occurs

Fig. 23.4 Definitions of classes ListNode, List and EmptyListException

1 **// Fig. 23.4: LinkedListLibrary.cs**  
 2 **// Class ListNode and class List definitions.**  
 3  
 4 **using System;**  
 5  
 6 **namespace LinkedListLibrary**  
 7 **{**  
 8 **// class to represent one node in a list**  
 9 **class ListNode**  
 10 **{**  
 11 **private object data;**  
 12 **private ListNode next;**  
 13  
 14 **// constructor to create ListNode that refers to dataValue**  
 15 **// and is last node in list**  
 16 **public ListNode( object dataValue )**   
 17 **: this( dataValue, null )**  
 18 **{**  
 19 **}**  
 20  
 21 **// constructor to create ListNode that refers to dataValue**  
 22 **// and refers to next ListNode in List**  
 23 **public ListNode( object dataValue, ListNode nextNode )**  
 24 **{**  
 25 **data = dataValue;**   
 26 **next = nextNode;**   
 27 **}**  
 28  
 29 **// property Next**  
 30 **public ListNode Next**  
 31 **{**  
 32 **get**  
 33 **{**  
 34 **return next;**  
 35 **}**  
 36  
 37 **set**  
 38 **{**  
 39 **next = value;**  
 40 **}**  
 41 **}**  
 42  
 43 **// property Data**  
 44 **public object Data**  
 45 **{**  
 46 **get**  
 47 **{**  
 48 **return data;**  
 49 **}**  
 50 **}**   
 51  
 52 **} // end class ListNode**  
 53  
 54 **// class List definition**  
 55 **public class List**  
 56 **{**  
 57 **private ListNode firstNode;**  
 58 **private ListNode lastNode;**  
 59 **private string name; // string like "list" to display**  
 60  
 61 **// construct empty List with specified name**  
 62 **public List( string listName )**  
 63 **{**  
 64 **name = listName;**  
 65 **firstNode = lastNode = null;**  
 66 **}**  
 67  
 68 **// construct empty List with "list" as its name**  
 69 **public List() : this( "list" )**  
 70 **{**   
 71 **}**  
 72  
 73 **// Insert object at front of List. If List is empty,**   
 74 **// firstNode and lastNode will refer to same object.**  
 75 **// Otherwise, firstNode refers to new node.**  
 76 **public void InsertAtFront( object insertItem )**  
 77 **{**  
 78 **lock ( this )**  
 79 **{**  
 80 **if ( IsEmpty() )**  
 81 **firstNode = lastNode =**   
 82 **new ListNode( insertItem );**  
 83 **else**  
 84 **firstNode =**   
 85 **new ListNode( insertItem, firstNode );**  
 86 **}**  
 87 **}**  
 88  
 89 **// Insert object at end of List. If List is empty,**   
 90 **// firstNode and lastNode will refer to same object.**  
 91 **// Otherwise, lastNode's Next property refers to new node.**  
 92 **public void InsertAtBack( object insertItem )**  
 93 **{**  
 94 **lock ( this )**  
 95 **{**  
 96 **if ( IsEmpty() )**  
 97 **firstNode = lastNode =**   
 98 **new ListNode( insertItem );**  
 99  
 100 **else**  
 101 **lastNode = lastNode.Next =**   
 102 **new ListNode( insertItem );**  
 103 **}**  
 104 **}**  
 105  
 106 **// remove first node from List**  
 107 **public object RemoveFromFront()**  
 108 **{**  
 109 **lock ( this )**  
 110 **{**  
 111 **if ( IsEmpty() )**  
 112 **throw new EmptyListException( name );**  
 113  
 114 **object removeItem = firstNode.Data; // retrieve data**  
 115  
 116 **// reset firstNode and lastNode references**  
 117 **if ( firstNode == lastNode )**  
 118 **firstNode = lastNode = null;**  
 119  
 120 **else**   
 121 **firstNode = firstNode.Next;**  
 122  
 123 **return removeItem; // return removed data**  
 124 **}**  
 125 **}**  
 126  
 127 **// remove last node from List**  
 128 **public object RemoveFromBack()**  
 129 **{**  
 130 **lock ( this )**  
 131 **{**  
 132 **if ( IsEmpty() )**  
 133 **throw new EmptyListException( name );**  
 134  
 135 **object removeItem = lastNode.Data; // retrieve data**  
 136  
 137 **// reset firstNode and lastNode references**  
 138 **if ( firstNode == lastNode )**  
 139 **firstNode = lastNode = null;**  
 140  
 141 **else**   
 142 **{**  
 143 **ListNode current = firstNode;**  
 144  
 145 **// loop while current node is not lastNode**  
 146 **while ( current.Next != lastNode )**   
 147 **current = current.Next; // move to next node**  
 148  
 149 **// current is new lastNode**  
 150 **lastNode = current;**  
 151 **current.Next = null;**  
 152 **}**  
 153  
 154 **return removeItem; // return removed data**  
 155 **}**  
 156 **}**  
 157  
 158 **// return true if List is empty**  
 159 **public bool IsEmpty()**  
 160 **{**  
 161 **lock ( this )**  
 162 **{**  
 163 **return firstNode == null;**  
 164 **}**  
 165 **}**  
 166  
 167 **// output List contents**  
 168 **virtual public void Print()**  
 169 **{**  
 170 **lock ( this )**  
 171 **{**  
 172 **if ( IsEmpty() )**   
 173 **{**  
 174 **Console.WriteLine( "Empty " + name );**  
 175 **return;**  
 176 **}**  
 177  
 178 **Console.Write( "The " + name + " is: " );**  
 179  
 180 **ListNode current = firstNode;**  
 181  
 182 **// output current node data while not at end of list**  
 183 **while ( current != null )**   
 184 **{**  
 185 **Console.Write( current.Data + " " );**  
 186 **current = current.Next;**  
 187 **}**  
 188  
 189 **Console.WriteLine( "\n" );**  
 190 **}**  
 191 **}**  
 192  
 193 **} // end class List**  
 194  
 195 **// class EmptyListException definition**  
 196 **public class EmptyListException : ApplicationException**  
 197 **{**  
 198 **public EmptyListException( string name )**  
 199 **: base( "The " + name + " is empty" )**  
 200 **{**  
 201 **}**  
 202  
 203 **} // end class EmptyListException**  
 204  
 205 **} // end namespace LinkedListLibrary**

Fig. 23.5 Demonstrating the linked list

1 **// Fig 23.5: ListTest.cs**   
 2 **// Testing class List.**  
 3  
 4 **using System;**  
 5 **using LinkedListLibrary;**  
 6  
 7 **namespace ListTest**  
 8 **{**  
 9 **// class to test List class functionality**  
 10 **class ListTest**  
 11 **{**  
 12 **static void Main( string[] args )**  
 13 **{**   
 14 **List list = new List(); // create List container**  
 15  
 16 **// create data to store in List**  
 17 **bool aBoolean = true;**  
 18 **char aCharacter = '$';**  
 19 **int anInteger = 34567;**  
 20 **string aString = "hello";**  
 21  
 22 **// use List insert methods**  
 23 **list.InsertAtFront( aBoolean );**  
 24 **list.Print();**  
 25 **list.InsertAtFront( aCharacter );**  
 26 **list.Print();**  
 27 **list.InsertAtBack( anInteger );**  
 28 **list.Print();**  
 29 **list.InsertAtBack( aString );**  
 30 **list.Print();**  
 31  
 32 **// use List remove methods**  
 33 **object removedObject;**  
 34  
 35 **// remove data from list and print after each removal**  
 36 **try**   
 37 **{**  
 38 **removedObject = list.RemoveFromFront();**  
 39 **Console.WriteLine( removedObject + " removed" );**  
 40 **list.Print();**  
 41  
 42 **removedObject = list.RemoveFromFront();**  
 43 **Console.WriteLine( removedObject + " removed" );**  
 44 **list.Print();**  
 45  
 46 **removedObject = list.RemoveFromBack();**  
 47 **Console.WriteLine( removedObject + " removed" );**  
 48 **list.Print();**  
 49  
 50 **removedObject = list.RemoveFromBack();**  
 51 **Console.WriteLine( removedObject + " removed" );**  
 52 **list.Print();**  
 53 **}**  
 54  
 55 **// process exception if list empty when attempt is**   
 56 **// made to remove item**  
 57 **catch ( EmptyListException emptyListException )**   
 58 **{**  
 59 **Console.Error.WriteLine( "\n" + emptyListException );**  
 60 **}**   
 61 **Console.ReadKey();** 62 **} // end method Main**  
 63  
 64 **} // end class ListTest**  
 65 **}**

Fig. 23.4 and Fig. 23.5 code walkthroughs

The program consists of four classes –

ListNode (Fig. 23.4, lines 9–52),

List (Fig. 23.4, lines 55–193),

EmptyListException (Fig. 23.4, lines 196 - 203) and

ListTest (Fig. 23.5)

The classes in Fig. 23.4 create a linked-list library (defined in namespace LinkedListLibrary) that can be reused throughout this booklet

Encapsulated in each List object is a linked list of ListNode objects

Class ListNode (Fig. 23.4, lines 9 - 52) consists of two member variables - data and next

Member data can refer to any object

Member next stores a reference to the next ListNode object in the linked list

A List accesses the ListNode member variables via the properties Data (lines 44 - 50) and Next (lines 30 - 41), respectively

Class List contains private members firstNode (a reference to the first ListNode in a List) and lastNode (a reference to the last ListNode in a List)

The constructors (lines 62 - 66 and 69 - 71) initialize both references to null

InsertAtFront (lines 76 - 87), InsertAtBack (lines 92 - 104), RemoveFromFront (lines 107 - 125) and RemoveFromBack (lines 128 - 156) are the primary methods of class List

Each method uses a lock block to ensure that List objects are multithread safe when used in a multithreaded program

i.e. if one thread is modifying the contents of a List object, no other thread can modify the same List object at the same time

Method IsEmpty (lines 159 - 165) is a predicate method that determines whether the list is empty (i.e. if the reference to the first node of the list is null)

Predicate methods typically test a condition and do not modify the object on which they are called

If the list is empty, method IsEmpty returns true; otherwise, it returns false

Method Print (lines 168 - 191) displays the list’s contents

Both IsEmpty and Print also use lock blocks so that the state of the list does not change while those methods are performing their tasks

Class EmptyListException (lines 196 - 203) defines an exception class to handle illegal operations on an empty List

Class ListTest (Fig. 23.5) uses the linked-list library to create and manipulate a linked list

Line 14 creates a new instance of type List named list

Lines 17 - 20 create data to add to the list

Lines 23 - 30 use List insertion methods to insert these objects and use List method Print to output the contents of list after each insertion

The code inside the try block (lines 36–53) removes objects via List deletion methods, outputs the object removed and outputs list after every deletion

If there is an attempt to remove an object from an empty list, this try block catches the EmptyListException

Note that class ListTest uses namespace LinkedListLibrary (Fig. 23.4); thus, the solution for class ListTest must have a reference to the LinkedListLibrary class library

Over the next several pages, we will discuss each of the methods of class List in detail

Method InsertAtFront (Fig. 23.4, lines 76 - 87) places a new node at the front of the list

The method consists of the following three steps

(as illustrated in diagram ‘Operation 1’ above):

1. Call IsEmpty to determine whether the list is empty (line 80)
2. If the list is empty, set both firstNode and lastNode to refer to a new ListNode initialized with insertItem (lines 81 - 82)

The ListNode constructor at lines 16 - 19 (Fig. 23.4) calls the ListNode constructor at lines 23–27 (Fig. 23.4) to set instance variable data to refer to the object passed as the first argument and sets the next reference to null

1. If the list is not empty, the new node is “threaded” (not to be confused with multithreading) into the list by setting firstNode to refer to a new ListNode object initialized with insertItem and firstNode (lines 84–85)

When the ListNode constructor (lines 23–27 of Fig. 23.4) executes, it sets instance variable data to refer to the object passed as the first argument and performs the insertion by setting the next reference to the ListNode passed as the second argument

Diagram ‘Operation 1’ illustrates method InsertAtFront

Part (a) of the figure shows the list and the new node during the InsertAtFront operation and before the threading of the new node into the list

The dotted arrows in part (b) illustrate step 3 of the InsertAtFront operation, which enables the node containing 12 to become the new list front

Method InsertAtBack (Fig. 23.4, lines 92 - 104) places a new node at the back of the list

The method consists of the following three steps

(as illustrated in diagram ‘Operation 2’ above):

1. Call IsEmpty to determine whether the list is empty (line 96)
2. If the list is empty, set both firstNode and lastNode to refer to a new ListNode initialized with insertItem (lines 97 - 98)

The ListNode constructor at lines 16 - 19 (Fig. 23.4) calls the ListNode constructor at lines 23 - 27 (Fig. 23.4) to set instance variable data to refer to the object passed as the first argument and sets the next reference to null

1. If the list is not empty, thread the new node into the list by setting lastNode and lastNode.next to refer to a new ListNode object initialized with insertItem (lines 101 - 102)

When the ListNode constructor (lines 16 - 19 of Fig. 23.4) executes, it sets instance variable data to refer to the object passed as an argument and sets the next reference to null

Diagram ‘Operation 2’ illustrates an InsertAtBack operation

Part (a) of the figure shows the list and the new node during the InsertAtBack operation and before the new node has been threaded into the list

The dotted arrows in part (b) illustrate the steps of method InsertAtBack that enable a new node to be added to the end of a list that is not empty

Method RemoveFromFront ( Fig. 23.4, lines 107 - 127) removes the front node of the list and returns a reference to the removed data

The method throws an EmptyListException (line 114) if the programmer tries to remove a node from an empty list

Otherwise, the method returns a reference to the removed data

The method consists of the following four steps

(as illustrated in diagram ‘Operation 3’ above):

1. Assign firstNode.Data (the data being removed from the list) to reference removeItem (line 116)
2. If the objects to which firstNode and lastNode refer are the same object, the list has only one element prior to the removal attempt

In this case, the method sets firstNode and lastNode to null (line 120) to “dethread” (remove) the node from the list (leaving the list empty)

1. If the list has more than one node prior to removal, then the method leaves reference lastNode as is and simply assigns firstNode.Next to reference firstNode (line 123)

Thus, firstNode references the node that was the second node prior to the RemoveFromFront call

Diagram ‘Operation 3’ illustrates method RemoveFromFront

Part (a) illustrates the list before the removal operation

Part (b) shows actual reference manipulations

Method RemoveFromBack (Fig. 23.4, lines 130 - 160) removes the last node of a list and returns a reference to the removed data

The method throws an EmptyListException (line 137) if the program attempts to remove a node from an empty list

The method consists of several steps

(as illustrated in diagram ‘Operation 4’ above):

1. Assign lastNode.Data (the data being removed from the list) to reference removeItem (line 139)
2. If the objects to which firstNode and lastNode refer are the same object (line 142), the list has only one element prior to the removal attempt

In this case, the method sets firstNode and lastNode to null (line 143) to ‘dethread’ (remove) that node from the list (leaving the list empty)

1. If the list has more than one node prior to removal, create the ListNode reference current and assign it firstNode (line 147)
2. Now “walk the list” with current until it references the node before the last node

The while loop (lines 150–151) assigns current.Next to reference current as long as current.Next is not equal to lastNode

1. After locating the second-to-last node, assign current to lastNode (line 154) to dethread the last node from the list
2. Set current.Next to null (line 155) in the new last node of the list to ensure proper list termination
3. Return the removeItem reference (line 140)

Diagram ‘Operation 4’ illustrates method RemoveFromBack

Part (a) illustrates the list before the removal operation

Part (b) shows the actual reference manipulations

Method Print (Fig.23.4, lines 172 - 195) first determines whether the list is empty (line 176)

If so, Print displays a string consisting of the string "Empty" and the list’s name, then returns control to the calling method

Otherwise, Print outputs the data in the list

The method prints a string consisting of the string "The", the name and the string

" is:"

Then, line 184 creates ListNode reference current and initializes it with firstNode

While current is not null, there are more items in the list

Therefore, the method prints current.Data (line 189), then assigns current.Next to current (line 190) to move to the next node in the list

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

Stacks

A stack is a constrained version of a linked list - a stack takes new nodes and releases nodes only at the top

For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure

The link member in the bottom (i.e., last) node of the stack is set to null to indicate the bottom of the stack

The primary operations to manipulate a stack are push and pop

Operation push adds a new node to the top of the stack

Operation pop removes a node from the top of the stack and returns the item from the popped node

Stacks have many interesting applications

For example, when a program calls a method, the called method must know how to return to its caller, so the return address is pushed onto the program execution stack

If a series of method calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each method can return to its caller

Stacks support recursive method calls in the same manner that they do conventional non-recursive method calls

The program-execution stack contains the space created for local variables on each invocation of a method during a program’s execution

When the method returns to its caller, the space for that method's local variables is popped off the stack, and those variables are no longer known to the program

The System.Collections namespace contains class Stack for implementing and manipulating stacks that can grow and shrink during program execution

The “Collection Classes” section of this booklet will discuss the class Stack

We take advantage of the close relationship between lists and stacks to implement a stack class by reusing a list class

We demonstrate two different forms of reusability

First, we implement the stack class by inheriting from class List of Fig. 23.4

Then, we implement an identically performing stack class through composition by including a List object as a private member of a stack class

This booklet implements list, stack and queue data structures to store object references to encourage further reusability

Thus, any object type can be stored in a list, stack or queue

Fig. 23.10 StackInheritance extends class List

1 **// Fig. 23.10: StackInheritanceLibrary.cs**  
 2 **// Implementing a stack by inheriting from class List.**  
 3  
 4 **using System;**  
 5 **using LinkedListLibrary;**  
 6  
 7 **namespace StackInheritanceLibrary**  
 8 **{**  
 9 **// class StackInheritance inherits class List's capabilities**  
 10 **public class StackInheritance : List**   
 11 **{**  
 12 **// pass name "stack" to List constructor**  
 13 **public StackInheritance() : base( "stack" )**  
 14 **{**  
 15 **}**  
 16  
 17 **// place dataValue at top of stack by inserting**   
 18 **// dataValue at front of linked list**  
 19 **public void Push( object dataValue )**   
 20 **{**   
 21 **InsertAtFront( dataValue );**   
 22 **}**  
 23  
 24 **// remove item from top of stack by removing**  
 25 **// item at front of linked list**  
 26 **public object Pop()**  
 27 **{**   
 28 **return RemoveFromFront();**   
 29 **}**  
 30  
 31 **} // end class StackInheritance**  
 32 **}**

Fig. 23.11 Using class StackInheritance

1 **// Fig. 23.11: StackInheritanceTest.cs**  
 2 **// Testing class StackInheritance.**  
 3  
 4 **using System;**  
 5 **using StackInheritanceLibrary;**  
 6 **using LinkedListLibrary;**  
 7  
 8 **namespace StackInheritanceTest**  
 9 **{**  
 10 **// demonstrate functionality of class StackInheritance**  
 11 **class StackInheritanceTest**  
 12 **{**  
 13 **static void Main( string[] args )**  
 14 **{**  
 15 **StackInheritance stack = new StackInheritance();**  
 16  
 17 **// create objects to store in the stack**  
 18 **bool aBoolean = true;**  
 19 **char aCharacter = '$';**  
 20 **int anInteger = 34567;**  
 21 **string aString = "hello";**  
 22  
 23 **// use method Push to add items to stack**  
 24 **stack.Push( aBoolean );**  
 25 **stack.Print();**  
 26 **stack.Push( aCharacter );**  
 27 **stack.Print();**  
 28 **stack.Push( anInteger );**  
 29 **stack.Print();**  
 30 **stack.Push( aString );**  
 31 **stack.Print();**  
 32  
 33 **// use method Pop to remove items from stack**  
 34 **try**   
 35 **{**   
 36 **while ( true )**   
 37 **{**  
 38 **object removedObject = stack.Pop();**  
 39 **Console.WriteLine( removedObject + " popped" );**  
 40 **stack.Print();**  
 41 **}**  
 42 **}**  
 43  
 44 **// if exception occurs, print stack trace**  
 45 **catch ( EmptyListException emptyListException )**   
 46 **{**  
 47 **Console.Error.WriteLine(**   
 48 **emptyListException.StackTrace );**  
 49 **}**  
 50  
 51 **} // end method Main**  
 52  
 53 **} // end class StackInheritanceTest**  
 54 **}**

Fig. 23.12 StackComposition class encapsulates functionality of class List

1 **// Fig. 23.12: StackCompositionLibrary.cs**   
 2 **// StackComposition definition with composed List object.**  
 3  
 4 **using System;**  
 5 **using LinkedListLibrary;**  
 6  
 7 **namespace StackCompositionLibrary**  
 8 **{**  
 9 **// class StackComposition encapsulates List's capabilities**  
 10 **public class StackComposition**  
 11 **{**  
 12 **private List stack;**  
 13  
 14 **// construct empty stack**  
 15 **public StackComposition()**  
 16 **{**  
 17 **stack = new List( "stack" );**  
 18 **}**  
 19  
 20 **// add object to stack**  
 21 **public void Push( object dataValue )**   
 22 **{**   
 23 **stack.InsertAtFront( dataValue );**   
 24 **}**  
 25  
 26 **// remove object from stack**  
 27 **public object Pop()**  
 28 **{**   
 29 **return stack.RemoveFromFront();**   
 30 **}**  
 31  
 32 **// determine whether stack is empty**  
 33 **public bool IsEmpty()**  
 34 **{**  
 35 **return stack.IsEmpty();**  
 36 **}**  
 37  
 38 **// output stack contents**  
 39 **public void Print()**  
 40 **{**  
 41 **stack.Print();**  
 42 **}**  
 43  
 44 **} // end class StackComposition**   
 45 **}**

Fig. 23.10 and Fig. 23.11 code walkthroughs

The program of Fig. 23.10 and Fig. 23.11 creates a stack class by inheriting from class List of Fig. 23.4

We want the stack to have methods Push, Pop, IsEmpty and Print

Essentially, these are the methods InsertAtFront, RemoveFromFront, IsEmpty and Print of class List

Of course, class List contains other methods (such as InsertAtBack and RemoveFromBack) that we would rather not make accessible through the public interface of the stack

It is important to remember that all methods in the public interface of class List are also public methods of the derived class StackInheritance (Fig. 23.10)

When we implement the stack’s methods, we have each StackInheritance method call the appropriate List method - method Push calls InsertAtFront, method Pop calls RemoveFromFront

Class StackInheritance does not define methods IsEmpty and Print, because StackInheritance inherits these methods from class List into StackInheritance’s public interface

The methods in class StackInheritance do not use lock statements

Each of the methods in this class calls a method from class List that uses lock

If two threads call Push on the same stack object, only one of the threads at a time will be able to call List method InsertAtFront

Note that class StackInheritance uses namespace LinkedListLibrary (Fig. 23.4) therefore the solution for the class library that defines StackInheritance must have a reference to the LinkedListLibrary class library

StackInheritanceTest’s Main method (Fig. 23.11) uses class StackInheritance to instantiate a stack of objects called stack

Lines 18 - 21 define four objects that will be pushed onto the stack and popped off the stack

The program pushes onto the stack (lines 24, 26, 28 and 30) a bool containing true, a char containing $, an int containing 34567 and a string containing hello

An infinite while loop (lines 36 - 41) pops the elements from the stack

When there are no objects left to pop, method Pop throws an EmptyListException and the program displays the exception’s stack trace, which shows the program execution stack at the time the exception occurred

The program uses method Print (inherited from class List) to output the contents of the stack after each operation

Note that class StackInheritanceTest uses namespace LinkedListLibrary ( Fig. 23.4) and namespace StackInheritanceLibrary ( Fig. 23.10) and therefore, the solution for class StackInheritanceTest must have references to both class libraries

Another way to implement a stack class is by reusing a list class through composition

The class in Fig. 23.12 uses a private object of class List (line 12) in the definition of class StackComposition

Composition enables us to hide the methods of class List that should not be in our stack’s public interface by providing public interface methods only to the required List methods

This class implements each stack method by delegating its work to an appropriate List method

In particular, StackComposition calls List methods InsertAtFront, RemoveFromFront, IsEmpty and Print

A class StackCompositionTest has not been provided here; there is no need because the only difference in this example is that we change the type of the stack from StackInheritance to StackComposition

The output would be identical

Queues

The queue is another common dynamic data structure

A queue is similar to a checkout line in a supermarket - the first person in line is served first; customers enter the line only at the end, and they wait to be served

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 a first-in, first-out (FIFO) data structure

The insert and remove operations are known as ‘enqueue’ and ‘dequeue’

Queues have many applications in computer systems

Most computers have only a single processor, so they can only serve one process / task at a time

Entries for the other processes / tasks are placed in a queue

The entry at the front of the queue receives the first available service

Each entry gradually advances to the front of the queue as processes / tasks receive service

Another application - queues also support print spooling

A multiuser environment may have only one printer

Several users may send output to the printer. If the printer is busy, users may still generate other outputs, which are “spooled” to disk (much as thread is wound onto a spool), where they wait in a queue until the printer becomes available

Yet another application - information packets also wait in queues in computer networks

Each time a packet arrives at a network node, the routing node must route it 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

Fig. 23.13 QueueInheritance extends class List

1 **// Fig. 23.13: QueueInheritanceLibrary.cs**  
 2 **// Implementing a queue by inheriting from class List.**  
 3  
 4 **using System;**  
 5 **using LinkedListLibrary;**  
 6  
 7 **namespace QueueInheritanceLibrary**  
 8 **{**  
 9 **// class QueueInheritance inherits List's capabilities**  
 10 **public class QueueInheritance : List**   
 11 **{**  
 12 **// pass name "queue" to List constructor**  
 13 **public QueueInheritance() : base( "queue" )**  
 14 **{**  
 15 **}**  
 16  
 17 **// place dataValue at end of queue by inserting**   
 18 **// dataValue at end of linked list**  
 19 **public void Enqueue( object dataValue )**  
 20 **{**  
 21 **InsertAtBack( dataValue );**  
 22 **}**  
 23  
 24 **// remove item from front of queue by removing**  
 25 **// item at front of linked list**  
 26 **public object Dequeue( )**  
 27 **{**   
 28 **return RemoveFromFront();**   
 29 **}**  
 30  
 31 **} // end of QueueInheritance**  
 32 **}**

Fig. 23.14 Using inheritance to create a queue

1 **// Fig. 23.14: QueueTest.cs**  
 2 **// Testing class QueueInheritance.**  
 3  
 4 **using System;**  
 5 **using QueueInheritanceLibrary;**  
 6 **using LinkedListLibrary;**  
 7  
 8 **namespace QueueTest**  
 9 **{**  
 10 **// demonstrate functionality of class QueueInheritance**  
 11 **class QueueTest**  
 12 **{**  
 13 **static void Main( string[] args )**  
 14 **{**  
 15 **QueueInheritance queue = new QueueInheritance();**  
 16  
 17 **// create objects to store in the stack**  
 18 **bool aBoolean = true;**  
 19 **char aCharacter = '$';**  
 20 **int anInteger = 34567;**  
 21 **string aString = "hello";**  
 22  
 23 **// use method Enqueue to add items to queue**  
 24 **queue.Enqueue( aBoolean );**  
 25 **queue.Print();**  
 26 **queue.Enqueue( aCharacter );**  
 27 **queue.Print();**  
 28 **queue.Enqueue( anInteger );**  
 29 **queue.Print();**  
 30 **queue.Enqueue( aString );**  
 31 **queue.Print();**  
 32  
 33 **// use method Dequeue to remove items from queue**  
 34 **object removedObject = null;**  
 35  
 36 **// remove items from queue**  
 37 **try**   
 38 **{**  
 39 **while ( true )**   
 40 **{**  
 41 **removedObject = queue.Dequeue();**  
 42 **Console.WriteLine( removedObject + " dequeue" );**  
 43 **queue.Print();**  
 44 **}**  
 45 **}**  
 46  
 47 **// if exception occurs, print stack trace**  
 48 **catch ( EmptyListException emptyListException )**   
 49 **{**  
 50 **Console.Error.WriteLine(**   
 51 **emptyListException.StackTrace );**  
 52 **}**  
 53  
 54 **} // end method Main**  
 55  
 56 **} // end class QueueTest**  
 57 **}**

Fig. 23.13 and Fig. 23.14 code walkthroughs

The program of Fig. 23.13 and Fig. 23.14 creates a queue class through inheritance from a list class

We want the QueueInheritance class (Fig. 23.13) to have methods Enqueue, Dequeue, IsEmpty and Print

Note that these methods are essentially the InsertAtBack, RemoveFromFront, IsEmpty and Print methods of class List

Of course, the list class contains other methods (such as InsertAtFront and RemoveFromBack) that we would rather not make accessible through the public interface to the queue class

Remember that all methods in the public interface of the List class are also public methods of the derived class QueueInheritance

When we implement the queue’s methods, we have each QueueInheritance method call the appropriate List method - method Enqueue calls InsertAtBack, method Dequeue calls RemoveFromFront, and IsEmpty and Print calls invoke their base-class versions

Class QueueInheritance does not need to define methods IsEmpty and Print, because QueueInheritance inherits these methods from class List into QueueInheritance’s public interface

Also, the methods in class QueueInheritance do not use lock statements

Each of the methods in this class calls a method from class List that uses lock

Note that class QueueInheritance uses namespace LinkedListLibrary (Fig. 23.4) therefore the solution for the class library that defines QueueInheritance must have a reference to the LinkedListLibrary class library

Class QueueInheritanceTest’s Main method (Fig. 23.14) uses class QueueInheritance to instantiate a queue of objects called queue

Lines 18–21 define four objects that will be enqueued and dequeued

The program enqueues (lines 24, 26, 28 and 30) a bool containing true, a char containing $, an int containing 34567 and a string containing hello

An infinite while loop (lines 39 - 44) dequeues the elements from the queue in FIFO order

When there are no objects left to dequeue, method Dequeue throws an EmptyListException and the program displays the exception’s stack trace, which shows the program execution stack at the time the exception occurred

The program uses method Print (inherited from class List) to output the contents of the queue after each operation

Note that class QueueInheritanceTest uses namespace LinkedListLibrary (Fig. 23.4) and namespace QueueInheritanceLibrary ( Fig. 23.13)therefore the solution for class QueueInheritanceTest must have references to both class libraries

Binary Trees

Linked lists, stacks and queues are linear data structures (i.e., sequences)

A tree is a nonlinear, two-dimensional data structure with special properties

Tree nodes contain two or more links

This section discusses ‘binary trees’ – tree structures whose nodes all contain two links (none, one or both of which may be null)

The root node is the first node in a tree

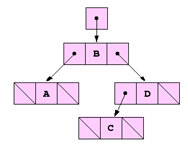
Each link in the root node refers to a child

The left child is the first node in the left subtree, and the right child is the first node in the right subtree

The children of a specific node are called siblings

A node with no children is called a leaf node

Computer scientists normally draw trees from the root node down—exactly the opposite of the way most trees grow in nature

[](javascript:void(0))

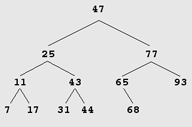
**A graphical representation of a binary tree**

In our binary tree example, we create a special binary tree called a binary search tree

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 the subtree’s parent node, and the values in any right subtree are greater than the value in the subtree’s parent node

The diagram below illustrates a binary search tree with 12 integer values

Note that the shape of the binary search tree that corresponds to a set of data can depend on the order in which the values are inserted into the tree

[](javascript:void(0))

**A binary search tree containing 12 values**

Binary Search Tree of Integer Values

The following application of Fig. 23.17 plus Fig. 23.18 creates a binary search tree of integers and traverses it (i.e., walks through all its nodes) in three ways - using recursive inorder, preorder and postorder **‘traversals’**

The program generates 10 random numbers and inserts each into the tree

Fig. 23.17’s code defines class Tree in namespace BinaryTreeLibrary for reuse purposes

Fig. 23.18’s code defines class TreeTest to demonstrate class Tree’s functionality

Method Main of class TreeTest instantiates an empty Tree object, then randomly generates 10 integers and inserts each value in the binary tree by calling Tree method InsertNode

The program then performs preorder, inorder and postorder traversals of the tree

We will discuss these traversals later

Fig. 23.17 Definitions of TreeNode and Tree for a binary search tree

1 **// Fig. 23.17: BinaryTreeLibrary.cs**  
 2 **// Definition of class TreeNode and class Tree.**  
 3  
 4 **using System;**  
 5  
 6 **namespace BinaryTreeLibrary**  
 7 **{**  
 8 **// class TreeNode definition**  
 9 **class TreeNode**   
 10 **{**  
 11 **private TreeNode leftNode;**   
 12 **private int data;**   
 13 **private TreeNode rightNode;**   
 14  
 15 **// initialize data and make this a leaf node**  
 16 **public TreeNode( int nodeData )**  
 17 **{**   
 18 **data = nodeData;**   
 19 **leftNode = rightNode = null; // node has no children**  
 20 **}**  
 21  
 22 **// LeftNode property**  
 23 **public TreeNode LeftNode**   
 24 **{**  
 25 **get**   
 26 **{**  
 27 **return leftNode;**  
 28 **}**  
 29  
 30 **set**  
 31 **{**  
 32 **leftNode = value;**  
 33 **}**  
 34 **}**  
 35  
 36 **// Data property**  
 37 **public int Data**  
 38 **{**  
 39 **get**   
 40 **{**  
 41 **return data;**  
 42 **}**  
 43  
 44 **set**   
 45 **{**  
 46 **data = value;**  
 47 **}**  
 48 **}**  
 49  
 50 **// RightNode property**  
 51 **public TreeNode RightNode**   
 52 **{**  
 53 **get**   
 54 **{**  
 55 **return rightNode;**  
 56 **}**  
 57  
 58 **set**  
 59 **{**  
 60 **rightNode = value;**  
 61 **}**  
 62 **}**  
 63  
 64  
 65 **// insert TreeNode into Tree that contains nodes;**  
 66 **// ignore duplicate values**  
 67 **public void Insert( int insertValue )**  
 68 **{**  
 69 **// insert in left subtree**  
 70 **if ( insertValue < data )**   
 71 **{**  
 72 **// insert new TreeNode**  
 73 **if ( leftNode == null )**  
 74 **leftNode = new TreeNode( insertValue );**  
 75  
 76 **// continue traversing left subtree**  
 77 **else**  
 78 **leftNode.Insert( insertValue );**  
 79 **}**  
 80  
 81 **// insert in right subtree**  
 82 **else if ( insertValue > data )**   
 83 **{**  
 84 **// insert new TreeNode**  
 85 **if ( rightNode == null )**  
 86 **rightNode = new TreeNode( insertValue );**  
 87  
 88 **// continue traversing right subtree**  
 89 **else**  
 90 **rightNode.Insert( insertValue );**  
 91 **}**  
 92  
 93 **} // end method Insert**  
 94  
 95 **} // end class TreeNode**  
 96  
 97 **// class Tree definition**  
 98 **public class Tree**   
 99 **{**  
 100 **private TreeNode root;**  
 101  
 102 **// construct an empty Tree of integers**  
 103 **public Tree()**   
 104 **{**   
 105 **root = null;**   
 106 **}**  
 107  
 108 **// Insert a new node in the binary search tree.**  
 109 **// If the root node is null, create the root node here.**  
 110 **// Otherwise, call the insert method of class TreeNode.**  
 111 **public void InsertNode( int insertValue )**  
 112 **{**  
 113 **lock ( this )**   
 114 **{**   
 115 **if ( root == null )**  
 116 **root = new TreeNode( insertValue );**  
 117  
 118 **else**  
 119 **root.Insert( insertValue );**  
 120 **}**  
 121 **}**  
 122  
 123 **// begin preorder traversal**  
 124 **public void PreorderTraversal()**  
 125 **{**   
 126 **lock ( this )**   
 127 **{**  
 128 **PreorderHelper( root );**   
 129 **}**  
 130 **}**  
 131  
 132 **// recursive method to perform preorder traversal**  
 133 **private void PreorderHelper( TreeNode node )**  
 134 **{**  
 135 **if ( node == null )**  
 136 **return;**  
 137  
 138 **// output node data**  
 139 **Console.Write( node.Data + " " );**   
 140  
 141 **// traverse left subtree**  
 142 **PreorderHelper( node.LeftNode );**   
 143  
 144 **// traverse right subtree**  
 145 **PreorderHelper( node.RightNode );**   
 146 **}**  
 147  
 148 **// begin inorder traversal**  
 149 **public void InorderTraversal()**  
 150 **{**   
 151 **lock ( this )**   
 152 **{**  
 153 **InorderHelper( root );**   
 154 **}**  
 155 **}**  
 156  
 157 **// recursive method to perform inorder traversal**  
 158 **private void InorderHelper( TreeNode node )**  
 159 **{**  
 160 **if ( node == null )**  
 161 **return;**  
 162  
 163 **// traverse left subtree**  
 164 **InorderHelper( node.LeftNode );**  
 165  
 166 **// output node data**  
 167 **Console.Write( node.Data + " " );**  
 168  
 169 **// traverse right subtree**  
 170 **InorderHelper( node.RightNode );**  
 171 **}**  
 172  
 173 **// begin postorder traversal**  
 174 **public void PostorderTraversal()**  
 175 **{**   
 176 **lock ( this )**   
 177 **{**  
 178 **PostorderHelper( root );**   
 179 **}**  
 180 **}**  
 181  
 182 **// recursive method to perform postorder traversal**  
 183 **private void PostorderHelper( TreeNode node )**  
 184 **{**  
 185 **if ( node == null )**  
 186 **return;**  
 187  
 188 **// traverse left subtree**  
 189 **PostorderHelper( node.LeftNode );**  
 190  
 191 **// traverse right subtree**  
 192 **PostorderHelper( node.RightNode );**  
 193  
 194 **// output node data**  
 195 **Console.Write( node.Data + " " );**  
 196 **}**  
 197  
 198 **} // end class Tree**  
 199 **}**

Fig. 23.18 Creating and traversing a binary tree

1 **// Fig. 23.18: TreeTest.cs**  
 2 **// This program tests class Tree.**  
 3  
 4 **using System;**  
 5 **using BinaryTreeLibrary;**  
 6  
 7 **namespace TreeTest**  
 8 **{**  
 9 **// class TreeTest definition**  
 10 **public class TreeTest**   
 11 **{**  
 12 **// test class Tree**  
 13 **static void Main( string[] args )**  
 14 **{**  
 15 **Tree tree = new Tree();**  
 16 **int insertValue;**  
 17  
 18 **Console.WriteLine( "Inserting values: " );**  
 19 **Random random = new Random();**  
 20  
 21 **// insert 10 random integers from 0-99 in tree**   
 22 **for ( int i = 1; i <= 10; i++ )**   
 23 **{**  
 24 **insertValue = random.Next( 100 );**  
 25 **Console.Write( insertValue + " " );**  
 26  
 27 **tree.InsertNode( insertValue );**  
 28 **}**  
 29  
 30 **// perform preorder traversal of tree**  
 31 **Console.WriteLine( "\n\nPreorder traversal" );**  
 32 **tree.PreorderTraversal();**  
 33  
 34 **// perform inorder traversal of tree**  
 35 **Console.WriteLine( "\n\nInorder traversal" );**  
 36 **tree.InorderTraversal();**  
 37  
 38 **// perform postorder traversal of tree**  
 39 **Console.WriteLine( "\n\nPostorder traversal" );**  
 40 **tree.PostorderTraversal();**  
 41 **Console.WriteLine();**  
 42 **}**  
 43  
 44 **} // end class TreeTest**  
 45 **}**

The creation then traversal through a binary search tree of integers

Fig. 23.17 and Fig. 23.18 code walkthroughs

Class TreeNode (lines 9 - 95 of Fig. 23.17) is a self-referential class containing three private data members - leftNode and rightNode, of type TreeNode, and data, of type int

Initially, every TreeNode is a leaf node, so the constructor (lines 16 - 20) initializes references leftNode and rightNode to null

Properties LeftNode (lines 23 - 34), Data (lines 37 - 48) and RightNode (lines 51 - 62) provide access to a ListNode’s private data members

We will discuss TreeNode method Insert (lines 67 - 93) shortly

Class Tree (lines 98 - 198 of Fig. 23.17) manipulates objects of class TreeNode

Class Tree has as private data root (line 100) - a reference to the root node of the tree

The class contains public method InsertNode (lines 111 - 121) to insert a new node in the tree and public methods PreorderTraversal (lines 124–130), InorderTraversal (lines 149–155) and PostorderTraversal (lines 174–180) to begin traversals of the tree

Each of these methods calls a separate recursive utility method to perform the traversal operations on the internal representation of the tree

The Tree constructor (lines 103 - 106) initializes root to null to indicate that the tree initially is empty

The Tree class’s method InsertNode (lines 111 - 121) first locks the Tree object for thread safety, then determines whether the tree is empty

If the tree is empty, line 116 allocates a new TreeNode, initializes the node with the integer being inserted in the tree and assigns the new node to root

If the tree is not empty, InsertNode calls TreeNode method Insert (lines 67 - 93), which recursively determines the location for the new node in the tree and inserts the node at that location

A node can be inserted only as a leaf node in a binary search tree

The TreeNode method Insert compares the value to insert with the data value in the root node

If the insert value is less than the root-node data, the program determines whether the left subtree is empty (line 73)

If empty, line 74 allocates a new TreeNode, initializes it with the integer being inserted and assigns the new node to reference leftNode

Otherwise, line 78 recursively calls Insert for the left subtree to insert the value into the left subtree

If the insert value is greater than the root-node data, the program determines whether the right subtree is empty (line 85)

If empty, line 86 allocates a new TreeNode, initializes it with the integer being inserted and assigns the new node to reference rightNode

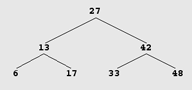
Otherwise, line 90 recursively calls Insert for the right subtree to insert the value in the right subtree

Methods InorderTraversal, PreorderTraversal and PostorderTraversal call helper methods InorderHelper (lines 158 - 171), PreorderHelper (lines 133 - 146) and PostorderHelper (lines 183 - 196), respectively, to traverse the tree and print the node values

The purpose of the helper methods in class Tree is to allow the programmer to start a traversal without the need to obtain a reference to the root node first, then call the recursive method with that reference

Methods InorderTraversal, PreorderTraversal and PostorderTraversal simply take the private reference root and pass it to the appropriate helper method to initiate a traversal of the tree

For the following discussion, we will use the binary search tree shown:

[](javascript:void(0))

**A binary search tree**

Method InorderHelper (lines 158 - 171) defines the steps for an **inorder** traversal

Those steps are as follows:

1. If the argument is null, return immediately
2. Traverse the left subtree with a call to InorderHelper (line 164)
3. Process the value in the node (line 167)
4. Traverse the right subtree with a call to InorderHelper (line 170)

The inorder traversal does not process the value in a node until the values in that node’s left subtree are processed

The **inorder** traversal of the tree illustrated above 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 - so this process is called the binary tree sort

Method PreorderHelper (lines 133 - 146) defines the steps for a **preorder** traversal

Those steps are as follows:

1. If the argument is null, return immediately
2. Process the value in the node (line 139)
3. Traverse the left subtree with a call to PreorderHelper (line 142)
4. Traverse the right subtree with a call to PreorderHelper (line 145)

The preorder traversal processes the value in each node as the node is visited

After processing the value in a given node, the preorder traversal processes the values in the left subtree, then the values in the right subtree

The **preorder** traversal of the tree illustrated above is

**27 13 6 17 42 33 48**

Method PostorderHelper (lines 183–198) defines the steps for a **postorder** traversal

Those steps are as follows:

1. If the argument is null, return immediately
2. Traverse the left subtree with a call to PostorderHelper (line 189)
3. Traverse the right subtree with a call to PostorderHelper (line 192)
4. Process the value in the node (line 195)

The postorder traversal processes the value in each node after the values of all that node’s children are processed

The **postorder** traversal of the tree illustrated above is

**6 17 13 33 48 42 27**

The binary search tree facilitates duplicate elimination

While building a tree, the insertion operation recognizes attempts to insert a duplicate value, because a duplicate follows the same “go left” or “go right” decisions on each comparison as the original value did

Thus, the insertion operation eventually compares the duplicate with a node containing the same value

At this point, the insertion operation might simply discard the duplicate value

Searching a binary tree for a value that matches a key value is fast, especially for tightly packed trees

In a tightly packed tree, each level contains about twice as many elements as the previous level

The tree illustrated above is a tightly packed binary tree

A binary search tree with n elements has a minimum of log2n levels

Thus, at most log2n comparisons are required either to find a match or to determine that no match exists

Searching a (tightly packed) 1000 - element binary search tree requires at most 10 comparisons, because 210 > 1000

Searching a (tightly packed) 1,000,000 - element binary search tree requires at most 20 comparisons, because 220 > 1,000,000

Binary Search Tree of IComparable Objects

The binary tree example shown above works nicely when all the data is of type int

Suppose that you want to manipulate a binary tree of double values

You could rewrite the TreeNode and Tree classes with different names and customize the classes to manipulate double values

Similarly, for each data type you could create customized versions of classes TreeNode and Tree

This results in a proliferation of code, which can become difficult to manage and maintain

The C++ programming language provides a technology called ‘templates’ that enables us to write a class definition once, then have the compiler generate new versions of the class for any data type we choose

Ideally, we would like to define the functionality of a binary tree once and reuse that functionality for many data types

Languages like Java™ and C# provide **polymorphic** capabilities that enable all objects to be manipulated in a uniform manner

Using such capabilities enables us to design a more flexible data structure

In the example that follows, we take advantage of C#’s polymorphic capabilities by implementing TreeNode and Tree classes that manipulate objects of any type that implements interface IComparable (namespace System)

It is imperative that we be able to compare objects stored in a binary search, so we can determine the path to the insertion point of a new node

Classes that implement IComparable define method CompareTo, which compares the object that invokes the method with the object that the method receives as an argument

The method returns an int value less than zero if the calling object is less than the argument object, zero if the objects are equal and a positive value if the calling object is greater than the argument object

In addition, note that both the calling and argument objects must be of the same data type; otherwise, the method throws an ArgumentException

Fig. 23.20 Definitions of class TreeNode and Tree for manipulating IComparable objects

1 **// Fig. 23.20: BinaryTreeLibrary2.cs**  
 2 **// Definition of class TreeNode and class Tree for IComparable**  
 3 **// objects.**  
 4  
 5 **using System;**  
 6  
 7 **namespace BinaryTreeLibrary2**  
 8 **{**  
 9 **// class TreeNode definition**  
 10 **class TreeNode**   
 11 **{**  
 12 **private TreeNode leftNode;**   
 13 **private IComparable data;**   
 14 **private TreeNode rightNode;**   
 15  
 16 **// initialize data and make this a leaf node**  
 17 **public TreeNode( IComparable nodeData )**  
 18 **{**   
 19 **data = nodeData;**   
 20 **leftNode = rightNode = null; // node has no children**  
 21 **}**  
 22  
 23 **// LeftNode property**  
 24 **public TreeNode LeftNode**   
 25 **{**  
 26 **get**   
 27 **{**  
 28 **return leftNode;**  
 29 **}**  
 30  
 31 **set**  
 32 **{**  
 33 **leftNode = value;**  
 34 **}**  
 35 **}**  
 36  
 37 **// Data property**  
 38 **public IComparable Data**  
 39 **{**  
 40 **get**   
 41 **{**  
 42 **return data;**  
 43 **}**  
 44  
 45 **set**   
 46 **{**  
 47 **data = value;**  
 48 **}**  
 49 **}**  
 50  
 51 **// RightNode property**  
 52 **public TreeNode RightNode**   
 53 **{**  
 54 **get**   
 55 **{**  
 56 **return rightNode;**  
 57 **}**  
 58  
 59 **set**  
 60 **{**  
 61 **rightNode = value;**  
 62 **}**  
 63 **}**  
 64  
 65 **// insert TreeNode into Tree that contains nodes;**  
 66 **// ignore duplicate values**  
 67 **public void Insert( IComparable insertValue )**  
 68 **{**  
 69 **// insert in left subtree**  
 70 **if ( insertValue.CompareTo( data ) < 0 )**   
 71 **{**  
 72 **// insert new TreeNode**  
 73 **if ( leftNode == null )**  
 74 **leftNode = new TreeNode( insertValue );**  
 75  
 76 **// continue traversing left subtree**  
 77 **else**  
 78 **leftNode.Insert( insertValue );**  
 79 **}**  
 80  
 81 **// insert in right subtree**  
 82 **else if ( insertValue.CompareTo( data ) > 0 )**   
 83 **{**  
 84 **// insert new TreeNode**  
 85 **if ( rightNode == null )**  
 86 **rightNode = new TreeNode( insertValue );**  
 87  
 88 **// continue traversing right subtree**  
 89 **else**  
 90 **rightNode.Insert( insertValue );**  
 91 **}**  
 92  
 93 **} // end method Insert**  
 94  
 95 **} // end class TreeNode**  
 96  
 97 **// class Tree definition**  
 98 **public class Tree**   
 99 **{**  
 100 **private TreeNode root;**  
 101  
 102 **// construct an empty Tree of integers**  
 103 **public Tree()**   
 104 **{**   
 105 **root = null;**   
 106 **}**  
 107  
 108 **// Insert a new node in the binary search tree.**  
 109 **// If the root node is null, create the root node here.**  
 110 **// Otherwise, call the insert method of class TreeNode.**  
 111 **public void InsertNode( IComparable insertValue )**  
 112 **{**  
 113 **lock ( this )**   
 114 **{**   
 115 **if ( root == null )**  
 116 **root = new TreeNode( insertValue );**  
 117  
 118 **else**  
 119 **root.Insert( insertValue );**  
 120 **}**  
 121 **}**  
 122  
 123 **// begin preorder traversal**  
 124 **public void PreorderTraversal()**  
 125 **{**   
 126 **lock ( this )**   
 127 **{**  
 128 **PreorderHelper( root );**   
 129 **}**  
 130 **}**  
 131  
 132 **// recursive method to perform preorder traversal**  
 133 **private void PreorderHelper( TreeNode node )**  
 134 **{**  
 135 **if ( node == null )**  
 136 **return;**  
 137  
 138 **// output node data**  
 139 **Console.Write( node.Data + " " );**   
 140  
 141 **// traverse left subtree**  
 142 **PreorderHelper( node.LeftNode );**   
 143  
 144 **// traverse right subtree**  
 145 **PreorderHelper( node.RightNode );**   
 146 **}**  
 147  
 148 **// begin inorder traversal**  
 149 **public void InorderTraversal()**  
 150 **{**   
 151 **lock ( this )**   
 152 **{**  
 153 **InorderHelper( root );**   
 154 **}**  
 155 **}**  
 156  
 157 **// recursive method to perform inorder traversal**  
 158 **private void InorderHelper( TreeNode node )**  
 159 **{**  
 160 **if ( node == null )**  
 161 **return;**  
 162  
 163 **// traverse left subtree**  
 164 **InorderHelper( node.LeftNode );**  
 165  
 166 **// output node data**  
 167 **Console.Write( node.Data + " " );**  
 168  
 169 **// traverse right subtree**  
 170 **InorderHelper( node.RightNode );**  
 171 **}**  
 172  
 173 **// begin postorder traversal**  
 174 **public void PostorderTraversal()**  
 175 **{**   
 176 **lock ( this )**   
 177 **{**  
 178 **PostorderHelper( root );**   
 179 **}**  
 180 **}**  
 181  
 182 **// recursive method to perform postorder traversal**  
 183 **private void PostorderHelper( TreeNode node )**  
 184 **{**  
 185 **if ( node == null )**  
 186 **return;**  
 187  
 188 **// traverse left subtree**  
 189 **PostorderHelper( node.LeftNode );**  
 190  
 191 **// traverse right subtree**  
 192 **PostorderHelper( node.RightNode );**  
 193  
 194 **// output node data**  
 195 **Console.Write( node.Data + " " );**  
 196 **}**  
 197  
 198 **} // end class Tree**  
 199 **}**

Fig. 23.21 Demonstrating class Tree with IComparable objects

1 **// Fig. 23.21: TreeTest.cs**  
 2 **// This program tests class Tree.**  
 3  
 4 **using System;**  
 5 **using BinaryTreeLibrary2;**  
 6  
 7 **namespace TreeTest**  
 8 **{**  
 9 **// class TreeTest definition**  
 10 **public class TreeTest**   
 11 **{**  
 12 **// test class Tree**  
 13 **static void Main( string[] args )**  
 14 **{**  
 15 **int[] intArray = { 8, 2, 4, 3, 1, 7, 5, 6 };**  
 16 **double[] doubleArray =**   
 17 **{ 8.8, 2.2, 4.4, 3.3, 1.1, 7.7, 5.5, 6.6 };**  
 18 **string[] stringArray = { "eight", "two", "four",**   
 19 **"three", "one", "seven", "five", "six" };**  
 20  
 21 **// create int Tree**  
 22 **Tree intTree = new Tree();**  
 23 **populateTree( intArray, intTree, "intTree" );**  
 24 **traverseTree( intTree, "intTree" );**  
 25  
 26 **// create double Tree**  
 27 **Tree doubleTree = new Tree();**  
 28 **populateTree( doubleArray, doubleTree, "doubleTree" );**  
 29 **traverseTree( doubleTree, "doubleTree" );**  
 30  
 31 **// create string Tree**  
 32 **Tree stringTree = new Tree();**  
 33 **populateTree( stringArray, stringTree, "stringTree" );**  
 34 **traverseTree( stringTree, "stringTree" );**  
 35 **}**  
 36  
 37 **// populate Tree with array elements**  
 38 **static void populateTree(**   
 39 **Array array, Tree tree, string name )**  
 40 **{**  
 41 **Console.WriteLine( "\nInserting into " + name + ":" );**  
 42  
 43 **foreach ( IComparable data in array )**   
 44 **{**  
 45 **Console.Write( data + " " );**  
 46 **tree.InsertNode( data );**  
 47 **}**  
 48 **}**  
 49  
 50 **// insert perform traversals**  
 51 **static void traverseTree( Tree tree, string treeType )**  
 52 **{**  
 53 **// perform preorder traversal of tree**  
 54 **Console.WriteLine(**   
 55 **"\n\nPreorder traversal of " + treeType );**  
 56 **tree.PreorderTraversal();**  
 57  
 58 **// perform inorder traversal of tree**  
 59 **Console.WriteLine(**   
 60 **"\n\nInorder traversal of " + treeType );**  
 61 **tree.InorderTraversal();**  
 62  
 63 **// perform postorder traversal of tree**  
 64 **Console.WriteLine(**   
 65 **"\n\nPostorder traversal of " + treeType );**  
 66 **tree.PostorderTraversal();**  
 67 **Console.WriteLine( "\n" );**  
 68 **}**  
 69  
 70 **} // end class TreeTest**  
 71 **}**

Adopting C#’s polymorphic capabilities to manipulate objects of any type that implements interface IComparable

Fig. 23.20 and Fig. 23.21 code walkthroughs

The Fig. 23.20 plus Fig. 23.21 application enhances the previous program (Fig. 23.17 plus Fig. 23.18) to manipulate IComparable objects

One restriction on the new versions of classes TreeNode and Tree in Fig. 23.20 is that each Tree object can contain objects of only one data type (e.g. all strings or all doubles)

If a program attempts to insert multiple data types in the same Tree object, ArgumentExceptions will occur

We have modified only six lines of code in class TreeNode (lines 13, 17, 38, 67, 70 and 82) and one line of code in class Tree (line 111) to enable processing of IComparable objects

With the exception of lines 70 and 82, all other changes simply replaced the type int with the type IComparable

Lines 70 and 82 previously used the < and > operators to compare the value being inserted with the value in a given node

These lines now compare IComparable objects via the interface’s method CompareTo, then test the method’s return value to determine whether it is less than zero (the calling object is less than the argument object) or greater than zero (the calling object is greater than the argument object), respectively

Class TreeTest (Fig. 23.21) creates three Tree objects to store int, double and string values, all of which the .NET Framework defines as IComparable types

The program populates the trees with the values in arrays intArray (line 15), doubleArray (lines 16 - 17) and stringArray (lines 18–19), respectively

Method populateTree (lines 38 - 48) receives an Array containing the initializer values for the Tree, a Tree into which the array elements will be placed and a string representing the Tree name as arguments, then inserts each Array element in the Tree

Method traverseTree (lines 51 - 68) receives a Tree and a string representing the Tree name as arguments, then outputs the preorder, inorder and postorder traversals of the Tree

Note that the inorder traversal of each Tree outputs the data in sorted order regardless of the data type stored in the Tree

Our polymorphic implementation of class Tree invokes the appropriate data type’s CompareTo method to determine the path to each value’s insertion point by using the standard binary search tree insertion rules

Notice also that the Tree of strings appears in alphabetical order

Collection Classes

Previous sections of this booklet discussed how to create and manipulate data structures

The discussion was “low level,” in the sense that we painstakingly created each element of each data structure dynamically with the new keyword and modified the data structures by directly manipulating their elements and references to their elements

In this section, we consider the **pre-packaged data-structure classes** provided by the .NET Framework

These classes are known as **collection classes**—they store collections of data

Each instance of one of these classes is known as a collection, which is a set of items

With collection classes, instead of creating data structures, the programmer simply uses existing data structures, without concern for how the data structures are implemented

This methodology is a marvellous example of code reuse

Programmers can code faster and can expect excellent performance, maximizing execution speed and minimizing memory consumption

Some examples of collections are the cards you hold in a card game, your favourite songs stored in your computer and the real-estate records in your local registry of deeds (which map book numbers and page numbers to property owners)

The .NET Framework provides several collections

We shall demonstrate four collection classes –

* Array
* ArrayList
* Stack
* Hashtable
* all from namespace System.Collections, plus built-in array capabilities

Note that namespace System.Collections provides several other additional data structures, including BitArray (a collection of true/false values), Queue and SortedList (a collection of key/value pairs that are sorted by key and can be accessed either by key or by index)

The .NET Framework provides ready-to-go, reusable components; you do not need to write your own collection classes

The collections are standardized so applications can share them easily, without having to be concerned with the details of their implementation

These collections are written for broad reuse

They are tuned for rapid execution and for efficient use of memory

As new data structures and algorithms are developed that fit this framework, a large base of programmers already will be familiar with the interfaces and algorithms implemented by those data structures

Class Array

All arrays defined in C# inherit from class Array (namespace System) which defines a Length property that specifies the number of elements in an array

Class Array also provides static methods that perform basic array-processing capabilities and algorithms for processing arrays

Typically, class Array overloads these methods to provide multiple options for performing algorithms

For example, Array method Reverse can reverse the order of the elements in an entire array or can reverse the elements in a specified range of elements in an array

For a complete list of class Array’s static methods and their overloaded versions, see the online documentation for the class

The following code listing demonstrates some of the static methods of class Array

Fig. 23.22 Program that demonstrates several static methods of class Array

1 **// Fig. 23.22: UsingArray.cs**  
 2 **// Using Array class to perform common array manipulations.**  
 3  
 4 **using System;**  
 5 **using System.Windows.Forms;**  
 6 **using System.Collections;**  
 7  
 8 **namespace UsingArray**  
 9 **{**  
 10 **// demonstrate algorithms of class Array**  
 11 **class UsingArray**  
 12 **{**  
 13 **private int[] intValues = { 1, 2, 3, 4, 5, 6 };**  
 14 **private double[] doubleValues =**   
 15 **{ 8.4, 9.3, 0.2, 7.9, 3.4 };**  
 16 **private int[] intValuesCopy;**  
 17 **private string output;**   
 18  
 19 **// method to build and display program output**  
 20 **public void Start()**  
 21 **{**  
 22 **intValuesCopy = new int[ intValues.Length ];**  
 23  
 24 **output = "Initial array values:\n";**  
 25 **PrintArray(); // output initial array contents**  
 26  
 27 **// sort doubleValues**  
 28 **Array.Sort( doubleValues );**  
 29  
 30 **// copy intValues into intValuesCopy**  
 31 **Array.Copy( intValues, intValuesCopy,**   
 32 **intValues.Length );**  
 33  
 34 **output += "\nArray values after Sort and Copy:\n";**  
 35 **PrintArray(); // output array contents**  
 36 **output += "\n";**  
 37  
 38 **// search for 5 in intValues**  
 39 **int result = Array.BinarySearch( intValues, 5 );**  
 40 **output +=**   
 41 **( result >= 0 ? "5 found at element " + result :**  
 42 **"5 not found" ) + " in intValues\n";**  
 43  
 44 **// search for 8763 in intValues**  
 45 **result = Array.BinarySearch( intValues, 8763 );**  
 46 **output +=**   
 47 **( result >= 0 ? "8763 found at element " + result :**  
 48 **"8763 not found" ) + " in intValues";**  
 49  
 50 **MessageBox.Show( output, "Using Class Array",**  
 51 **MessageBoxButtons.OK, MessageBoxIcon.Information );**  
 52 **}**  
 53  
 54 **// append array content to output string**  
 55 **private void PrintArray()**  
 56 **{**  
 57 **output += "doubleValues: ";**  
 58  
 59 **foreach ( double element in doubleValues )**  
 60 **output += element + " ";**  
 61  
 62 **output += "\nintValues: ";**  
 63  
 64 **foreach ( int element in intValues )**  
 65 **output += element + " ";**  
 66  
 67 **output += "\nintValuesCopy: ";**  
 68  
 69 **foreach ( int element in intValuesCopy )**  
 70 **output += element + " ";**  
 71  
 72 **output += "\n";**  
 73 **}**  
 74  
 75 **// main entry point for application**  
 76 **static void Main( string[] args )**  
 77 **{**  
 78 **UsingArray application = new UsingArray();**  
 79  
 80 **application.Start();**  
 81 **}**  
 82  
 83 **} // end class UsingArray**  
 84 **}**

Demonstrating a sample of class Array’s static methods

Fig. 23.22 code walkthrough

Line 28 uses static Array method Sort to sort an array of double values

When this method returns, the array contains its original elements sorted in ascending order

Lines 31 - 32 uses static Array method Copy to copy elements from array intArray into array intArrayCopy

The first argument is the array to copy (intValues), the second argument is the destination array (intValuesCopy) and the third argument is an integer representing the number of elements to copy (in this case, intValues.Length specifies all elements)

Lines 39 and 45 invoke static Array method BinarySearch to perform binary searches on array intValues

Method BinarySearch receives the sorted array in which to search and the key for which to search. The method returns the index in the array at which it finds the key (but a negative number if the key was not found)

Other static Array methods include:

Clear (to set a range of elements to 0 or null),

CreateInstance (to create a new array of a specified data type),

IndexOf (to locate the first occurrence of an object in an array or portion of an array),

LastIndexOf (to locate the last occurrence of an object in an array or portion of an array)

and

Reverse (to reverse the contents of an array or portion of an array)

Class ArrayList

In most programming languages, conventional arrays have a fixed size - they cannot be changed dynamically to conform to an application’s execution-time memory requirements

In some applications, this fixed-size limitation presents a problem for programmers

They need to choose between using fixed-size arrays that are large enough to store the maximum number of elements the program may require and using dynamic data structures that can grow and shrink the amount of memory required to store data in response to the changing requirements of a program at execution time

The .NET Framework’s class ArrayList collection mimics the functionality of conventional arrays and provides dynamic resizing of the collection through the class’s methods

At any time an ArrayList contains a certain number of elements less than or equal to its capacity - the number of elements currently reserved for an ArrayList

A program can manipulate the capacity with ArrayList property Capacity

If an ArrayList needs to grow, it (by default) doubles its current Capacity

ArrayLists store references to objects

All classes derive from class Object, so an ArrayList can contain objects of any type

The table below lists some useful methods of class ArrayList

|  |  |
| --- | --- |
| Method | Description |
| Add | Adds an object to the ArrayList. Returns an int specifying the index at which the object was added |
| Clear | Removes all the elements from the ArrayList |
| Contains | Returns true if the specified object is in the ArrayList; otherwise, returns false |
| IndexOf | Returns the index of the first occurrence of the specified object in the ArrayList |
| Insert | Inserts an object at the specified index |
| Remove | Removes the first occurrence of the specified object |
| RemoveAt | Removes an object at the specified index |
| RemoveRange | Removes a specified number of elements starting at a specified index in the ArrayList |
| Sort | Sorts the ArrayList |
| TrimToSize | Sets the Capacity of the ArrayList to be the number of elements the ArrayList currently contains |

Fig. 23.23 Demonstrating the ArrayList class

1 **// Fig. 23.24: ArrayListTest.cs**  
 2 **// Using class ArrayList.**  
 3  
 4 **using System;**  
 5 **using System.Drawing;**  
 6 **using System.Collections;**  
 7 **using System.ComponentModel;**  
 8 **using System.Windows.Forms;**  
 9 **using System.Data;**  
 10 **using System.Text;**  
 11  
 12 **namespace ArrayListTest**  
 13 **{**  
 14 **// demonstrating ArrayList functionality**  
 15 **public class ArrayListTest : System.Windows.Forms.Form**  
 16 **{**  
 17 **private System.Windows.Forms.Button addButton;**  
 18 **private System.Windows.Forms.TextBox inputTextBox;**  
 19 **private System.Windows.Forms.Label inputLabel;**  
 20 **private System.Windows.Forms.Button removeButton;**  
 21 **private System.Windows.Forms.Button firstButton;**  
 22 **private System.Windows.Forms.Button lastButton;**  
 23 **private System.Windows.Forms.Button isEmptyButton;**  
 24 **private System.Windows.Forms.Button containsButton;**  
 25 **private System.Windows.Forms.Button locationButton;**  
 26 **private System.Windows.Forms.Button trimButton;**  
 27 **private System.Windows.Forms.Button statisticsButton;**  
 28 **private System.Windows.Forms.Button displayButton;**  
 29  
 30 **// Required designer variable.**  
 31 **private System.ComponentModel.Container components = null;**  
 32 **private System.Windows.Forms.TextBox consoleTextBox;**  
 33  
 34 **// ArrayList for manipulating strings**  
 35 **private ArrayList arrayList = new ArrayList( 1 );**  
 36  
 37 **public ArrayListTest()**  
 38 **{**  
 39 **// Required for Windows Form Designer support**  
 40 **InitializeComponent();**  
 41 **}**  
 42  
 43 **// \*\*\*\*\* INSERT Fig. 23.24 CODE HERE \*\*\*\*\***

**(provided below)**

44  
 45 **// main entry point for the application**  
 46 **[STAThread]**  
 47 **static void Main()**   
 48 **{**  
 49 **Application.Run( new ArrayListTest() );**  
 50 **}**  
 51  
 52 **// add item to end of arrayList**  
 53 **private void addButton\_Click(**  
 54 **object sender, System.EventArgs e )**  
 55 **{**  
 56 **arrayList.Add( inputTextBox.Text );**  
 57 **consoleTextBox.Text =**   
 58 **"Added to end: " + inputTextBox.Text;**  
 59 **inputTextBox.Clear();**  
 60 **}**  
 61  
 62 **// remove specified item from arrayList**  
 63 **private void removeButton\_Click(**  
 64 **object sender, System.EventArgs e )**  
 65 **{**  
 66 **arrayList.Remove( inputTextBox.Text );**  
 67 **consoleTextBox.Text = "Removed: " + inputTextBox.Text;**  
 68 **inputTextBox.Clear();**  
 69 **}**  
 70  
 71 **// display first element**  
 72 **private void firstButton\_Click(**  
 73 **object sender, System.EventArgs e )**  
 74 **{**  
 75 **// get first element**  
 76 **try**  
 77 **{**  
 78 **consoleTextBox.Text =**   
 79 **"First element: " + arrayList[ 0 ];**  
 80 **}**  
 81  
 82 **// show exception if no elements in arrrayList**  
 83 **catch ( ArgumentOutOfRangeException outOfRange )**  
 84 **{**  
 85 **consoleTextBox.Text = outOfRange.ToString();**  
 86 **}**  
 87 **}**  
 88  
 89 **// display last element**  
 90 **private void lastButton\_Click(**  
 91 **object sender, System.EventArgs e )**  
 92 **{**  
 93 **// get last element**  
 94 **try**  
 95 **{**  
 96 **consoleTextBox.Text = "Last element: " +**   
 97 **arrayList[ arrayList.Count - 1 ];**  
 98 **}**  
 99  
 100 **// show exception if no elements in arrrayList**  
 101 **catch ( ArgumentOutOfRangeException outOfRange )**  
 102 **{**  
 103 **consoleTextBox.Text = outOfRange.ToString();**  
 104 **}**  
 105 **}**  
 106  
 107 **// determine whether arrayList is empty**  
 108 **private void isEmptyButton\_Click(**  
 109 **object sender, System.EventArgs e )**  
 110 **{**  
 111 **consoleTextBox.Text = ( arrayList.Count == 0 ?**   
 112 **"arrayList is empty" : "arrayList is not empty" );**  
 113 **}**  
 114  
 115 **// determine whether arrayList contains specified object**  
 116 **private void containsButton\_Click(**  
 117 **object sender, System.EventArgs e )**  
 118 **{**  
 119 **if ( arrayList.Contains( inputTextBox.Text ) )**  
 120 **consoleTextBox.Text = "arrayList contains " +**  
 121 **inputTextBox.Text;**  
 122 **else**  
 123 **consoleTextBox.Text = inputTextBox.Text +**   
 124 **" not found";**  
 125 **}**  
 126  
 127 **// determine location of specified object**  
 128 **private void locationButton\_Click(**  
 129 **object sender, System.EventArgs e )**  
 130 **{**  
 131 **consoleTextBox.Text = "Element is at location " +**  
 132 **arrayList.IndexOf( inputTextBox.Text );**  
 133 **}**  
 134  
 135 **// trim arrayList to current size**  
 136 **private void trimButton\_Click(**  
 137 **object sender, System.EventArgs e )**  
 138 **{**  
 139 **arrayList.TrimToSize();**  
 140 **consoleTextBox.Text = "Vector trimmed to size";**  
 141 **}**  
 142  
 143 **// show arrayList current size and capacity**  
 144 **private void statisticsButton\_Click(**  
 145 **object sender, System.EventArgs e )**  
 146 **{**  
 147 **consoleTextBox.Text = "Size = " + arrayList.Count +**   
 148 **"; capacity = " + arrayList.Capacity;**  
 149 **}**  
 150  
 151 **// display contents of arrayList**  
 152 **private void displayButton\_Click(**  
 153 **object sender, System.EventArgs e )**  
 154 **{**  
 155 **IEnumerator enumerator = arrayList.GetEnumerator();**  
 156 **StringBuilder buffer = new StringBuilder();**  
 157  
 158 **while ( enumerator.MoveNext() )**  
 159 **buffer.Append( enumerator.Current + " " );**  
 160  
 161 **consoleTextBox.Text = buffer.ToString();**  
 162 **}**  
 163 **}**  
 164 **}**

Fig. 23.24 Windows Form Designer code

Required by the above Fig. 23.23 code that demonstrates

the Array collection (built-in) class

**/// <summary>**

**/// Clean up any resources being used**

**/// </summary>**

**protected override void Dispose( bool disposing )**

**{**

**if ( disposing )**

**{**

**if ( components != null )**

**{**

**components.Dispose();**

**}**

**}**

**base.Dispose( disposing );**

**}**

**#region Windows Form Designer generated code**

**/// <summary>**

**/// Required method for Designer support - do not modify**

**/// the contents of this method with the code editor.**

**/// </summary>**

**private void InitializeComponent()**

**{**

**this.addButton = new System.Windows.Forms.Button();**

**this.removeButton = new System.Windows.Forms.Button();**

**this.firstButton = new System.Windows.Forms.Button();**

**this.lastButton = new System.Windows.Forms.Button();**

**this.isEmptyButton = new System.Windows.Forms.Button();**

**this.containsButton = new System.Windows.Forms.Button();**

**this.locationButton = new System.Windows.Forms.Button();**

**this.trimButton = new System.Windows.Forms.Button();**

**this.statisticsButton = new System.Windows.Forms.Button();**

**this.displayButton = new System.Windows.Forms.Button();**

**this.inputTextBox = new System.Windows.Forms.TextBox();**

**this.consoleTextBox = new System.Windows.Forms.TextBox();**

**this.inputLabel = new System.Windows.Forms.Label();**

**this.SuspendLayout();**

**//**

**// addButton**

**//**

**this.addButton.Location = new System.Drawing.Point(200, 8);**

**this.addButton.Name = "addButton";**

**this.addButton.Size = new System.Drawing.Size(80, 23);**

**this.addButton.TabIndex = 0;**

**this.addButton.Text = "Add";**

**this.addButton.Click += new System.EventHandler(this.addButton\_Click);**

**//**

**// removeButton**

**//**

**this.removeButton.Location = new System.Drawing.Point(8, 56);**

**this.removeButton.Name = "removeButton";**

**this.removeButton.Size = new System.Drawing.Size(80, 23);**

**this.removeButton.TabIndex = 1;**

**this.removeButton.Text = "Remove";**

**this.removeButton.Click += new System.EventHandler(this.removeButton\_Click);**

**//**

**// firstButton**

**//**

**this.firstButton.Location = new System.Drawing.Point(104, 56);**

**this.firstButton.Name = "firstButton";**

**this.firstButton.Size = new System.Drawing.Size(80, 23);**

**this.firstButton.TabIndex = 2;**

**this.firstButton.Text = "First";**

**this.firstButton.Click += new System.EventHandler(this.firstButton\_Click);**

**//**

**// lastButton**

**//**

**this.lastButton.Location = new System.Drawing.Point(200, 56);**

**this.lastButton.Name = "lastButton";**

**this.lastButton.Size = new System.Drawing.Size(80, 23);**

**this.lastButton.TabIndex = 3;**

**this.lastButton.Text = "Last";**

**this.lastButton.Click += new System.EventHandler(this.lastButton\_Click);**

**//**

**// isEmptyButton**

**//**

**this.isEmptyButton.Location = new System.Drawing.Point(8, 104);**

**this.isEmptyButton.Name = "isEmptyButton";**

**this.isEmptyButton.Size = new System.Drawing.Size(80, 23);**

**this.isEmptyButton.TabIndex = 4;**

**this.isEmptyButton.Text = "Is Empty?";**

**this.isEmptyButton.Click += new System.EventHandler(this.isEmptyButton\_Click);**

**//**

**// containsButton**

**//**

**this.containsButton.Location = new System.Drawing.Point(104, 104);**

**this.containsButton.Name = "containsButton";**

**this.containsButton.Size = new System.Drawing.Size(80, 23);**

**this.containsButton.TabIndex = 5;**

**this.containsButton.Text = "Contains";**

**this.containsButton.Click += new System.EventHandler(this.containsButton\_Click);**

**//**

**// locationButton**

**//**

**this.locationButton.Location = new System.Drawing.Point(200, 104);**

**this.locationButton.Name = "locationButton";**

**this.locationButton.Size = new System.Drawing.Size(80, 24);**

**this.locationButton.TabIndex = 6;**

**this.locationButton.Text = "Location";**

**this.locationButton.Click += new System.EventHandler(this.locationButton\_Click);**

**//**

**// trimButton**

**//**

**this.trimButton.Location = new System.Drawing.Point(8, 152);**

**this.trimButton.Name = "trimButton";**

**this.trimButton.Size = new System.Drawing.Size(80, 24);**

**this.trimButton.TabIndex = 7;**

**this.trimButton.Text = "Trim";**

**this.trimButton.Click += new System.EventHandler(this.trimButton\_Click);**

**//**

**// statisticsButton**

**//**

**this.statisticsButton.Location = new System.Drawing.Point(104, 152);**

**this.statisticsButton.Name = "statisticsButton";**

**this.statisticsButton.Size = new System.Drawing.Size(80, 23);**

**this.statisticsButton.TabIndex = 8;**

**this.statisticsButton.Text = "Statistics";**

**this.statisticsButton.Click += new System.EventHandler(this.statisticsButton\_Click);**

**//**

**// displayButton**

**//**

**this.displayButton.Location = new System.Drawing.Point(200, 152);**

**this.displayButton.Name = "displayButton";**

**this.displayButton.Size = new System.Drawing.Size(80, 23);**

**this.displayButton.TabIndex = 9;**

**this.displayButton.Text = "Display";**

**this.displayButton.Click += new System.EventHandler(this.displayButton\_Click);**

**//**

**// inputTextBox**

**//**

**this.inputTextBox.Location = new System.Drawing.Point(88, 8);**

**this.inputTextBox.Name = "inputTextBox";**

**this.inputTextBox.TabIndex = 10;**

**this.inputTextBox.Text = "";**

**//**

**// consoleTextBox**

**//**

**this.consoleTextBox.Location = new System.Drawing.Point(8, 200);**

**this.consoleTextBox.Multiline = true;**

**this.consoleTextBox.Name = "consoleTextBox";**

**this.consoleTextBox.ReadOnly = true;**

**this.consoleTextBox.Size = new System.Drawing.Size(272, 64);**

**this.consoleTextBox.TabIndex = 11;**

**this.consoleTextBox.Text = "";**

**//**

**// inputLabel**

**//**

**this.inputLabel.Location = new System.Drawing.Point(8, 16);**

**this.inputLabel.Name = "inputLabel";**

**this.inputLabel.Size = new System.Drawing.Size(72, 24);**

**this.inputLabel.TabIndex = 12;**

**this.inputLabel.Text = "Enter a string";**

**//**

**// ArrayListTest**

**//**

**this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);**

**this.ClientSize = new System.Drawing.Size(292, 273);**

**this.Controls.AddRange(new System.Windows.Forms.Control[]**

**(**

**this.inputLabel,**

**this.consoleTextBox,**

**this.inputTextBox,**

**this.displayButton,**

**this.statisticsButton,**

**this.trimButton,**

**this.locationButton,**

**this.containsButton,**

**this.isEmptyButton,**

**this.lastButton,**

**this.firstButton,**

**this.removeButton,**

**this.addButton**

**});**

**this.Name = "ArrayListTest";**

**this.Text = "ArrayListTest";**

**this.ResumeLayout(false);**

**}**

**#endregion**

Use of the ArrayList class

Fig. 23.23 code walkthrough

This code listing demonstrates class ArrayList and several of its methods

Users can type a string into the user interface’s TextBox, then press a button representing an ArrayList method to see that method’s functionality

A TextBox displays messages indicating each operation’s results

The ArrayList in this example stores strings that users input in the TextBox

Line 35 creates an ArrayList with an initial capacity of one element

This ArrayList will double in size each time the user fills the array and attempts to add another element

ArrayList method Add appends a new element at the end of an ArrayList

When the user clicks Add, event handler addButton\_Click (lines 53 - 60) invokes method Add (line 56) to append the string in the inputTextBox to the ArrayList

ArrayList method Remove deletes a specified item from an ArrayList

When the user clicks Remove, event handler removeButton\_Click (line 63 - 69) invokes Remove (line 66) to remove the string specified in the inputTextBox from the ArrayList

If the object passed to Remove is in the ArrayList, the first occurrence of that object is removed, and all subsequent elements shift toward the beginning of the ArrayList to fill the empty position

A program can access ArrayList elements like conventional array elements by following the ArrayList reference name with the array subscript operator ([]) and the desired index of the element

Event handlers firstButton\_Click (lines 72 - 87) and lastButton\_Click (lines 90 - 105) use the ArrayList subscript operator to retrieve the first element (line 79) and last element (line 97), respectively

An ArgumentOutOfRangeException occurs if the specified index is not both greater than 0 and less than the number of elements currently stored in the ArrayList

Event handler isEmptyButton\_Click (lines 108 - 113) uses ArrayList property Count (line 111) to determine whether the ArrayList is empty

Event handler containsButton\_Click (lines 116 - 125) uses ArrayList method Contains (line 119) to determine whether the the given object is currently in the ArrayList

If so, the method returns true; otherwise, it returns false

When the user clicks Location, event handler locationButton\_Click (lines 128 - 133) invokes ArrayList method IndexOf (line 132) to determine the index of a particular object in the ArrayList

IndexOf returns -1 if the element is not found

When the user clicks Trim, event handler trimButton\_Click (lines 136 - 141) invokes method TrimToSize (line 139) to set the Capacity property to equal the Count property

This reduces the storage capacity of the ArrayList to the exact number of elements currently in the ArrayList

When the user clicks Statistics, statisticsButton\_Click (lines 144 - 149) uses the Count and Capacity properties to display the current number of elements in the ArrayList and the maximum number of elements that can be stored without allocating more memory to the ArrayList

When user clicks Display, displayButton\_Click (lines 152 - 162) outputs the contents of the ArrayList

This event handler uses an IEnumerator (sometimes called an enumerator or an iterator) to traverse the elements of an ArrayList one element at a time

Interface IEnumerator defines methods MoveNext and Reset and property Current

MoveNext moves the enumerator to the next element in the ArrayList

The first call to MoveNext positions the enumerator at the first element of the ArrayList

MoveNext returns true if there is at least one more element in the ArrayList; otherwise, the method returns false

Method Reset positions the enumerator before the first element of the ArrayList

Methods MoveNext and Reset throw an InvalidOperationException if the contents of the collection are modified in any way after the enumerator’s creation

Property Current returns the object at the current location in the ArrayList

Line 155 creates an IEnumerator called enumerator and assigns it the result of calling ArrayList method GetEnumerator

Lines 158 - 159 iterate while MoveNext returns true, retrieve the current item via property Count and append it to buffer

When the loop terminates, line 161 displays the contents of buffer

Class Stack

The Stack class, as its name implies, implements a stack data structure

This class provides much of the functionality that we defined in our implementation in the preceding ‘Stacks’ section of this booklet

(refer back to that section for a discussion of stack data structure concepts)

Fig. 23.25 Using the Stack class

1 **// Fig. 23.25: StackTest.cs**  
 2 **// Demonstrates class Stack of namespace System.Collections.**  
 3  
 4 **using System;**  
 5 **using System.Drawing;**  
 6 **using System.Collections;**  
 7 **using System.ComponentModel;**  
 8 **using System.Windows.Forms;**  
 9 **using System.Data;**  
 10 **using System.Text;**  
 11  
 12 **namespace StackTest**  
 13 **{**  
 14 **// demonstrate Stack collection**  
 15 **public class StackTest : System.Windows.Forms.Form**  
 16 **{**  
 17 **private System.Windows.Forms.Label inputLabel;**  
 18 **private System.Windows.Forms.TextBox inputTextBox;**  
 19 **private System.Windows.Forms.Button pushButton;**  
 20 **private System.Windows.Forms.Button popButton;**  
 21 **private System.Windows.Forms.Button peekButton;**  
 22 **private System.Windows.Forms.Button isEmptyButton;**  
 23 **private System.Windows.Forms.Button searchButton;**  
 24 **private System.Windows.Forms.Button displayButton;**  
 25 **private System.Windows.Forms.Label statusLabel;**  
 26  
 27 **// Required designer variable.**  
 28 **private System.ComponentModel.Container components = null;**  
 29  
 30 **private Stack stack;**  
 31  
 32 **public StackTest()**  
 33 **{**  
 34 **// Required for Windows Form Designer support**  
 35 **InitializeComponent();**  
 36  
 37 **// create Stack**  
 38 **stack = new Stack();**  
 39 **}**  
 40  
 41 **// Visual Studio.NET generated code**  
 42  
 43 **// main entry point for the application**  
 44 **[STAThread]**  
 45 **static void Main()**   
 46 **{**  
 47 **Application.Run( new StackTest() );**  
 48 **}**  
 49  
 50 **// push element onto stack**  
 51 **private void pushButton\_Click(**  
 52 **object sender, System.EventArgs e )**  
 53 **{**  
 54 **stack.Push( inputTextBox.Text );**   
 55 **statusLabel.Text = "Pushed: " + inputTextBox.Text;**  
 56 **}**  
 57  
 58 **// pop element from stack**  
 59 **private void popButton\_Click(**  
 60 **object sender, System.EventArgs e )**  
 61 **{**  
 62 **// pop element**  
 63 **try**  
 64 **{**   
 65 **statusLabel.Text = "Popped: " + stack.Pop();**  
 66 **}**  
 67  
 68 **// print message if stack is empty**  
 69 **catch ( InvalidOperationException invalidOperation )**  
 70 **{**  
 71 **statusLabel.Text = invalidOperation.ToString();**  
 72 **}**  
 73 **}**  
 74  
 75 **// peek at top element of stack**  
 76 **private void peekButton\_Click(**  
 77 **object sender, System.EventArgs e )**  
 78 **{**  
 79 **// view top element**  
 80 **try**  
 81 **{**   
 82 **statusLabel.Text = "Top: " + stack.Peek();**  
 83 **}**  
 84  
 85 **// print message if stack is empty**  
 86 **catch ( InvalidOperationException invalidOperation )**  
 87 **{**  
 88 **statusLabel.Text = invalidOperation.ToString();**  
 89 **}**  
 90 **}**  
 91  
 92 **// determine whether stack is empty**  
 93 **private void isEmptyButton\_Click(**  
 94 **object sender, System.EventArgs e )**  
 95 **{**  
 96 **statusLabel.Text = ( stack.Count == 0 ?**   
 97 **"Stack is empty" : "Stack is not empty" );**  
 98 **}**  
 99  
 100 **// determine whether specified element is on stack**  
 101 **private void searchButton\_Click(**  
 102 **object sender, System.EventArgs e )**  
 103 **{**  
 104 **string result = stack.Contains( inputTextBox.Text ) ?**   
 105 **" found" : " not found";**  
 106  
 107 **statusLabel.Text = inputTextBox.Text + result;**  
 108 **}**  
 109  
 110 **// display stack contents**  
 111 **private void displayButton\_Click(**  
 112 **object sender, System.EventArgs e )**  
 113 **{**  
 114 **IEnumerator enumerator = stack.GetEnumerator();**  
 115 **StringBuilder buffer = new StringBuilder();**  
 116  
 117 **// while the enumerator can move on to the next element**  
 118 **// print that element out.**  
 119 **while ( enumerator.MoveNext() )**  
 120 **buffer.Append( enumerator.Current + " " );**  
 121  
 122 **statusLabel.Text = buffer.ToString();**  
 123 **}**  
 124 **}**  
 125 **}**

Using the Stack (collection) class

Fig. 23.25 code walkthrough

This application provides a GUI that enables the user to test many stack methods

Line 38 of the StackTest constructor creates a stack with the default initial capacity (10 elements)

As one might expect, class Stack has methods Push and Pop to perform the basic stack operations

Method Push takes an object as an argument and adds it to the top of the stack

If the number of items on the stack (the Count property) is equal to the capacity at the time of the Push operation, the stack grows to accommodate more objects

Event handler pushButton\_Click (lines 51 - 56) uses method Push to add a user-specified string to the stack (line 54)

Method Pop takes no arguments

This method removes and returns the object currently on top of the stack

Event handler popButton\_Click (lines 59 - 73) calls method Pop (line 57) to remove an object from the stack

An InvalidOperationException occurs if the stack is empty when the program calls Pop

Method Peek returns the value of the top stack element, but does not remove the element from the stack

We demonstrate Peek at line 82 in event handler peekButton\_Click (lines 76 - 90) to view the object on top of the stack

As with Pop, an InvalidOperationException occurs if the stack is empty when the program calls Peek

Event handler isEmptyButton\_Click (lines 93 - 98) determines whether the stack is empty by comparing Stack’s Count property to 0

If it is 0, the stack is empty; otherwise, it is not

Event handler searchButton\_Click (lines 101 - 108) uses Stack method Contains (lines 104 - 105) to determine whether the stack contains the object specified as its argument

Contains returns true if the stack contains the specified object, false otherwise

Event handler isEmptyButton\_Click (lines 111 - 123) uses an IEnumerator to traverse the Stack and display its contents

Class Hashtable

Object-oriented programming languages facilitate creating new types

When a program creates objects of new or existing types, it then needs to manage those objects efficiently

This includes sorting and retrieving objects

Sorting and retrieving information with arrays is efficient if some aspect of your data directly matches the key value and if those keys are unique and tightly packed

If you have 100 employees with nine-digit Social Security numbers and you want to store and retrieve employee data by using the Social Security number as a key, it would nominally require an array with 999,999,999 elements, because there are 999,999,999 unique nine-digit numbers

This is impractical for virtually all applications that key on Social Security numbers

If you could have an array that large, you could get very high performance storing and retrieving employee records by simply using the Social Security number as the array index

A large variety of applications have this problem—namely, that either the keys are of the wrong type (i.e., not non-negative integers), or they are of the right type, but they are sparsely spread over a large range

What is needed is a high-speed scheme for converting keys such as Social Security numbers and inventory part numbers into unique array subscripts

Then, when an application needs to store something, the scheme could convert the application key rapidly into a subscript and the record of information could be stored at that location in the array

Retrieval occurs the same way - once the application has a key for which it wants to retrieve the data record, the application simply applies the conversion to the key, which produces the array subscript where the data resides in the array and retrieves the data

The scheme we describe here is the basis of a technique called **hashing**

Why the name hashing ?

Because, when we convert a key into an array subscript, we literally scramble the bits, forming a kind of “mishmash” number

The number actually has no real significance beyond its usefulness in storing and retrieving this particular data record

A glitch in this scheme occurs when collisions occur

i.e. when two different keys “hash into” the same cell (or element) in the array

Since we cannot sort two different data records into the same space, we need to find an alternative home for all records beyond the first that hash to a particular array subscript

Many schemes exist for doing this. One is to “hash again” i.e. to reapply the hashing transformation to the key to provide a next candidate cell in the array

The hashing process is designed to be quite random, so the assumption is that with just a few hashes, an available cell will be found

Another scheme uses one hash to locate the first candidate cell

If the cell is occupied, successive cells are searched linearly until an available cell is found

Retrieval works the same way - the key is hashed once, the resulting cell is checked to determine whether it contains the desired data

If it does, the search is complete

If it does not, successive cells are searched linearly until the desired data is found

The most popular solution to hash-table collisions is to have each cell of the table be a hash “bucket,” typically a linked list of all the key/value pairs that hash to that cell

This is the solution that the .NET Framework’s Hashtable class implements

The load factor is one factor that affects the performance of hashing schemes

The load factor is the ratio of the number of occupied cells in the hash table to the size of the hash table

The closer the ratio gets to 1.0, the greater the chance of collisions

Programming hash tables properly is too complex for most casual programmers

Computer science students study hashing schemes thoroughly in courses called “Data Structures” and “Algorithms”

Recognizing the value of hashing, C# provides class Hashtable and some related features to enable programmers to take advantage of hashing without the complex details

**The preceding sentence is profoundly important in our study of object-oriented programming**

Classes encapsulate and hide complexity (i.e. implementation details) and offer user-friendly interfaces

Crafting classes to do this properly is one of the most valued skills in the field of object-oriented programming

A hash function performs a calculation that determines where to place data in the hashtable

The hash function is applied to the key in a key/value pair of objects

Class Hashtable can accept any object as a key

For this reason, class Object defines method GetHashCode, which all objects in C# inherit

Most classes that are candidates to be used as keys in a hash table override this method to provide one that performs efficient hashcode calculations for a specific data type

For example, a string has a hashcode calculation that is based on the contents of the string

The following code listing illustrates several methods of class Hashtable

Fig. 23.26 Using the Hashtable class

1 **// Fig. 23.26: HashtableTest.cs**  
 2 **// Demonstrate class Hashtable of namespace System.Collections.**  
 3  
 4 **using System;**  
 5 **using System.Drawing;**  
 6 **using System.Collections;**  
 7 **using System.ComponentModel;**  
 8 **using System.Windows.Forms;**  
 9 **using System.Data;**  
 10 **using System.Text;**  
 11  
 12 **namespace HashTableTest**  
 13 **{**  
 14 **// demonstrate Hashtable functionality**  
 15 **public class HashTableTest : System.Windows.Forms.Form**  
 16 **{**  
 17 **private System.Windows.Forms.Label firstNameLabel;**  
 18 **private System.Windows.Forms.Label lastNameLabel;**  
 19 **private System.Windows.Forms.Button addButton;**  
 20 **private System.Windows.Forms.TextBox lastNameTextBox;**  
 21 **private System.Windows.Forms.TextBox consoleTextBox;**  
 22 **private System.Windows.Forms.TextBox firstNameTextBox;**  
 23 **private System.Windows.Forms.Button getButton;**  
 24 **private System.Windows.Forms.Button removeButton;**  
 25 **private System.Windows.Forms.Button emptyButton;**  
 26 **private System.Windows.Forms.Button containsKeyButton;**  
 27 **private System.Windows.Forms.Button clearTableButton;**  
 28 **private System.Windows.Forms.Button listObjectsButton;**  
 29 **private System.Windows.Forms.Button listKeysButton;**  
 30 **private System.Windows.Forms.Label statusLabel;**  
 31  
 32 **// Required designer variable.**  
 33 **private System.ComponentModel.Container components = null;**  
 34  
 35 **// Hashtable to demonstrate functionality**  
 36 **private Hashtable table;**  
 37  
 38 **public HashTableTest()**  
 39 **{**  
 40 **// Required for Windows Form Designer support**  
 41 **InitializeComponent();**  
 42  
 43 **// create Hashtable object**  
 44 **table = new Hashtable();**  
 45 **}**  
 46  
 47 **// Visual Studio.NET generated code**  
 48  
 49 **// main entry point for the application**  
 50 **[STAThread]**  
 51 **static void Main()**   
 52 **{**  
 53 **Application.Run( new HashTableTest() );**  
 54 **}**  
 55  
 56 **// add last name and Employee object to table**  
 57 **private void addButton\_Click(**  
 58 **object sender, System.EventArgs e )**  
 59 **{**  
 60 **Employee employee = new Employee( firstNameTextBox.Text,**  
 61 **lastNameTextBox.Text );**  
 62  
 63 **// add new key/value pair**  
 64 **try**  
 65 **{**  
 66 **table.Add( lastNameTextBox.Text, employee );**  
 67 **statusLabel.Text = "Put: " + employee.ToString();**  
 68 **}**  
 69  
 70 **// if key is null or already in table, output message**  
 71 **catch ( ArgumentException argumentException )**  
 72 **{**  
 73 **statusLabel.Text = argumentException.ToString();**  
 74 **}**  
 75 **}**  
 76  
 77 **// get object for given key**  
 78 **private void getButton\_Click(**  
 79 **object sender, System.EventArgs e )**  
 80 **{**  
 81 **object result = table[ lastNameTextBox.Text ];**  
 82  
 83 **if ( result != null )**  
 84 **statusLabel.Text = "Get: " + result.ToString();**  
 85 **else**  
 86 **statusLabel.Text = "Get: " + lastNameTextBox.Text +**  
 87 **" not in table";**  
 88 **}**  
 89  
 90 **// remove key/value pair from table**  
 91 **private void removeButton\_Click(**  
 92 **object sender, System.EventArgs e )**  
 93 **{**  
 94 **table.Remove( lastNameTextBox.Text );**  
 95 **statusLabel.Text = "Object Removed";**  
 96 **}**  
 97  
 98 **// determine whether table is empty**  
 99 **private void emptyButton\_Click(**  
 100 **object sender, System.EventArgs e )**  
 101 **{**  
 102 **statusLabel.Text = "Table is " + (**  
 103 **table.Count == 0 ? "empty" : "not empty" );**  
 104 **}**  
 105  
 106 **// determine whether table contains specified key**  
 107 **private void containsKeyButton\_Click(**  
 108 **object sender, System.EventArgs e )**  
 109 **{**  
 110 **statusLabel.Text = "Contains key: " +**   
 111 **table.ContainsKey( lastNameTextBox.Text );**  
 112 **}**  
 113  
 114 **// discard all table contents**  
 115 **private void clearTableButton\_Click(**  
 116 **object sender, System.EventArgs e )**  
 117 **{**  
 118 **table.Clear();**  
 119 **statusLabel.Text = "Clear: Table is now empty";**  
 120 **}**  
 121  
 122 **// display list of objects in table**  
 123 **private void listObjectsButton\_Click(**  
 124 **object sender, System.EventArgs e )**  
 125 **{**  
 126 **IDictionaryEnumerator enumerator =**   
 127 **table.GetEnumerator();**  
 128 **StringBuilder buffer = new StringBuilder();**  
 129  
 130 **while ( enumerator.MoveNext() )**  
 131 **buffer.Append( enumerator.Value + "\r\n" );**  
 132  
 133 **consoleTextBox.Text = buffer.ToString();**  
 134 **}**  
 135  
 136 **// display list of keys in table**  
 137 **private void listKeysButton\_Click(**  
 138 **object sender, System.EventArgs e )**  
 139 **{**  
 140 **IDictionaryEnumerator enumerator =**   
 141 **table.GetEnumerator();**  
 142 **StringBuilder buffer = new StringBuilder();**  
 143  
 144 **while ( enumerator.MoveNext() )**  
 145 **buffer.Append( enumerator.Key + "\r\n" );**  
 146  
 147 **consoleTextBox.Text = buffer.ToString();**  
 148 **}**  
 149  
 150 **} // end class HashtableTest**  
 151  
 152 **// class Employee for use with HashtableTest**  
 153 **class Employee**  
 154 **{**  
 155 **private string first, last;**  
 156  
 157 **// constructor**  
 158 **public Employee( string fName, string lName )**  
 159 **{**  
 160 **first = fName;**  
 161 **last = lName;**  
 162 **}**  
 163  
 164 **// return Employee first and last names as string**  
 165 **public override string ToString()**  
 166 **{**  
 167 **return first + " " + last;**  
 168 **}**  
 169  
 170 **} // end class Employee**  
 171 **}**

Use of the Hashtable class

Fig. 23.26 code walkthrough

Event handler addButton\_Click (lines 57 - 75) reads the first name and last name of an employee from the user interface, creates an object of class Employee (defined at lines 153 - 170) and adds that Employee to the Hashtable with method Add (line 66)

This method receives two arguments - a key object, and a value object

In this example, the key is the last name of the Employee (a string), and the value is the corresponding Employee object

An ArgumentException occurs if the Hashtable already contains the key or if the key is null

Event handler getButton\_Click (lines 78 - 88) retrieves the object associated with a specific key, using the Hashtable’s subscript operator as shown on line 81

The expression in square brackets is the key for which the Hashtable should return the corresponding object

If the key is not found, the result is null

Event handler removeButton\_Click (lines 91 - 96) invokes Hashtable method Remove to delete a key and its associated object from the Hashtable

If the key does not exist in the table, nothing happens

Event handler emptyButton\_Click (lines 99 - 104) uses Hashtable property Count to determine whether the Hashtable is empty (i.e. Count is 0)

Event handler containsKeyButton\_Click (lines 107 - 112) invokes Hashtable method ContainsKey to determine whether the Hashtable contains the specified key

If so, the method returns true; otherwise, it returns false

Event handler clearTableButton\_Click (lines 115 - 120) invokes Hashtable method Clear to delete all Hashtable entries

Class Hashtable provides method GetEnumerator that returns an enumerator of type IDictionaryEnumerator, which derives from IEnumerator

Such enumerators provide properties Key and Value to access the information for a key/value pair

The event handler at lines 123 - 134 (listObjectsButton\_click) uses the Value property of the enumerator to output the objects in the Hashtable

The event handler at lines 137 - 148 (listKeysButton\_click) uses the Key property of the enumerator to output the keys in the Hashtable

Booklet Summary

Dynamic Data Structures & Collections – A summary

* Dynamic data structures can grow and shrink at execution time
* Creating and maintaining dynamic data structures requires dynamic memory allocation - the ability for a program to obtain more memory at execution time

(to hold new nodes) and to release memory no longer needed

* The limit for dynamic memory allocation can be as large as the available physical memory in the computer or the amount of available disk space in a virtual-memory system
* Operator new takes as an operand the type of the object being dynamically allocated and returns a reference to a newly created object of that type

If no memory is available, new throws an OutOfMemoryException

* A self-referential class contains a data member that refers to an object of the same class type. Self-referential objects can be linked to form useful data structures such as lists, queues, stacks and trees
* A linked list is a linear collection (i.e., a sequence) of self-referential class objects called nodes, connected by reference links
* A node can contain data of any type, including objects of other classes
* A linked list is accessed via a reference to the first node of the list

Each subsequent node is accessed via the link-reference member stored in the previous node

* By convention, the link reference in the last node of a list is set to null to mark the end of the list
* Stacks are important in compilers and operating systems
* A stack is a constrained version of a linked list—new nodes can be added to a stack and removed from a stack only at the top. A stack is referred to as a last-in, first-out (LIFO) data structure
* The primary stack operations are push and pop

Operation push adds a new node to the top of the stack

Operation pop removes a node from the top of the stack and returns the data object from the popped node

* Queues represent waiting lines

Insertions occur at the back (also referred to as the tail) of a queue and deletions occur from the front (also referred to as the head) of a queue

* A queue is similar to a checkout line in a supermarket

The first person in line is served first; other customers enter the line only at the end and wait to be served

* 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 for a queue are known as enqueue and dequeue
* Binary trees facilitate high-speed searching and sorting of data
* Tree nodes contain two or more links
* A binary tree is a tree whose nodes all contain two links

The root node is the first node in a tree

* Each link in the root node refers to a child

The left child is the first node in the left subtree and the right child is the first node in the right subtree

* The children of a node are called siblings

A node with no children is called a leaf node

* A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the values that subtree’s parent node and the values in any right subtree are greater than the values in that subtree’s parent node
* A node can be inserted only as a leaf node in a binary search tree
* An inorder traversal of a binary search tree processes the node values in ascending order
* The process of creating a binary search tree actually sorts the data—hence, the term “binary tree sort”
* In a preorder traversal, 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

* In a postorder traversal, the value in each node is processed after the node’s left and right subtrees are processed
* The binary search tree facilitates duplicate elimination

As the tree is created, attempts to insert a duplicate value are recognized because a duplicate follows the same “go left” or “go right” decisions on each comparison as the original value did

Thus, the duplicate eventually is compared with a node containing the same value. The duplicate value may simply be discarded at this point

* Class ArrayList can be used as a dynamically growing array
* ArrayList method Add adds an object to the ArrayList
* ArrayList method Remove removes the first occurrence of the specified object from the ArrayList
* The ArrayList subscript operator accesses elements of an ArrayList as if it were an array
* Class Stack is provided in the System.Collections namespace
* Stack method Push performs the push operation on the Stack
* Stack method Pop performs the pop operation on the Stack
* Class Hashtable is provided in the System.Collections namespace
* Hashtable method Add adds a key/value pair to the Hashtable
* Any class that implements the IEnumerator interface must define methods MoveNext and Reset and the Current property
* Method MoveNext must be called before the Current property is accessed for the first time
* Methods MoveNext and Reset throw an InvalidOperationException if the contents of the collection were modified in any way after the enumerator’s creation

Self-Review Exercises

Self-Review Exercises

Part 1

State whether each of the following is **true** or **false**

If **false**, explain why

1. In a queue, the first item to be added, is the last item to be removed
2. Trees can have no more than two child nodes per node
3. A tree node with no children is called a leaf node
4. Class Stack is in the System.Collections namespace
5. A class implementing interface IEnumerator must define only methods MoveNext and Reset
6. A hashtable stores key/value pairs
7. Linked list nodes are stored contiguously in memory
8. The primary operations of the stack data structure are enqueue and dequeue
9. Lists, stacks and queues are linear data structures

Part 2

Fill in the blanks for each of the following statements

1. A **\_\_\_\_\_\_\_\_** class is used to define nodes that form dynamic data structures, which can grow and shrink at execution time
2. Operator **\_\_\_\_\_\_\_\_** allocates memory dynamically; this operator returns a reference to the allocated memory
3. A **\_\_\_\_\_\_\_\_** is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list; this data structure returns node values in last-in, first-out order
4. A queue is a **\_\_\_\_\_\_\_\_** data structure, because the first nodes inserted are the first nodes removed
5. A **\_\_\_\_\_\_\_\_** is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list
6. A **\_\_\_\_\_\_\_\_** is a nonlinear, two-dimensional data structure that contains nodes with two or more links
7. The nodes of a \_\_\_\_\_\_\_\_ tree contain two link members
8. IEnumerator method \_\_\_\_\_\_\_\_ advances the enumerator to the next item
9. The tree-traversal algorithm that processes the node and then processes all the nodes to its left followed by all the nodes to its right is called \_\_\_\_\_\_\_\_
10. If the collection it references was altered after the enumerator’s creation, calling method Reset will cause an \_\_\_\_\_\_\_\_