Chapter 25 – Data Structures

Outline

Many slides modified by Prof. L. Lilien
(even many without an explicit message).

25.1 Introduction

- Dynamic data structures
  - Grow and shrink at execution time
  - Linked Lists:
    - "Lined up in a row"
    - Insertions and removals can occur anywhere in the list
  - Stacks:
    - Insertions and removals only at top
      - Push and pop
  - Queues:
    - Insertions made at back, removals from front
  - Trees, including binary trees:
    - Facilitate high-speed searching and sorting of data
    - Efficient elimination of duplicate items

25.2 Simple-Type `structs`, `Boxing` and `Unboxing`

- Data structures can store:
  - Simple-type values
  - Reference-type values
- There are mechanisms that enable manipulation of simple-type values as objects
  - Discussed below

Simple-Type `structs`

- Recall: Each simple type (int, char, …) has a corresponding struct in namespace `System` that declares this simple type

  We have `structs`:

  - `Boolean, Byte, Sbyte, Char, Decimal, Double, Single, Int32, UInt32, Int64, UInt64, Int16, UInt16`

- Simple types are just aliases for these corresponding `structs`
  - So a variable can be declared:

    EITHER using the keyword for that simple type

    OR the `struct` name
    - E.g., int (keyword) and Int32 (`struct` name) are interchangeable
  - Methods related to a simple type are located in the corresponding `struct`
    - E.g. method `Parse` –converting `string` to `int` value– is located in `struct In32`

Boxing Conversions

- All simple-type structs inherit from class ValueType in namespace System

  In turn, class ValueType inherits from class object

  => Any simple-type value can be assigned to an object variable

- Such an assignment is named a Boxing conversions
  - In Boxing conversion simple-type value is copied into an object

    => after Boxing conversion , a simple-type value can be manipulated as an object

Boxing Conversions

- Boxing conversions can be explicit or implicit

```
int i = 5; // create an int value
object obj1 = ( object) i;  // explicitly box the int value
object obj2 = i;  // implicitly box the int value
```

  - Now obj1 and obj2 refer to 2 different objects!
    - Both objects include a copy of the integer value from the `int` variable `i`

Unboxing Conversions

- Unboxing conversions — used for explicit conversion of an object reference to a simple value

```
int int1 = ( int ) obj1; // explicitly unbox the int value
                         // that was boxed within obj1
```

- Attempts to unbox an object reference that does not refer to a correct simple value causes InvalidCastException

- We'll see later(Ch. 27) how so called "generics" eliminate the overhead of boxing/unboxing

25.3 Self-Referential Classes

- Self-Referential Class
  - Contains a reference member to an object of the same class type
    - E.g.:
```
class Node
{
   private int data;
   private Node next;    // self-reference to node
   …
}
```
    - Reference can be used to link objects of the same type together
- Dynamic data structures require dynamic memory allocation
  - Ability to obtain memory when needed
  - Release memory when not needed any more
  - Uses **new** operator
    - Ex: Node nodeToAdd = **new** Node(10);

25.3 Self-Referential Class

```
1    class Node
2    {
3       private int data;
4       private Node next;   // self-reference
5
6       public Node( int d )
7       {
8          /* constructor body */
9       }
10
11      public int Data
12      {
13         get
14         {
15            /* get body */
16         }
17
18         set
19         {
20            /* set body */
21         }
22      }
23
24      public Node Next
25      {
26         get
27         {
28            /* get body */
29         }
30
31         set
32         {
33            /* set body */
34         }
35      }
36   }
```

Reference to object of same type

Fig. 23.1 Sample self-referential **Node** class definition (part 1)

25.3 Self-Referential Class

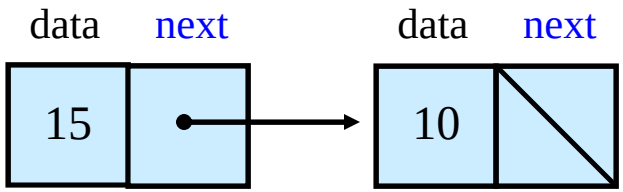
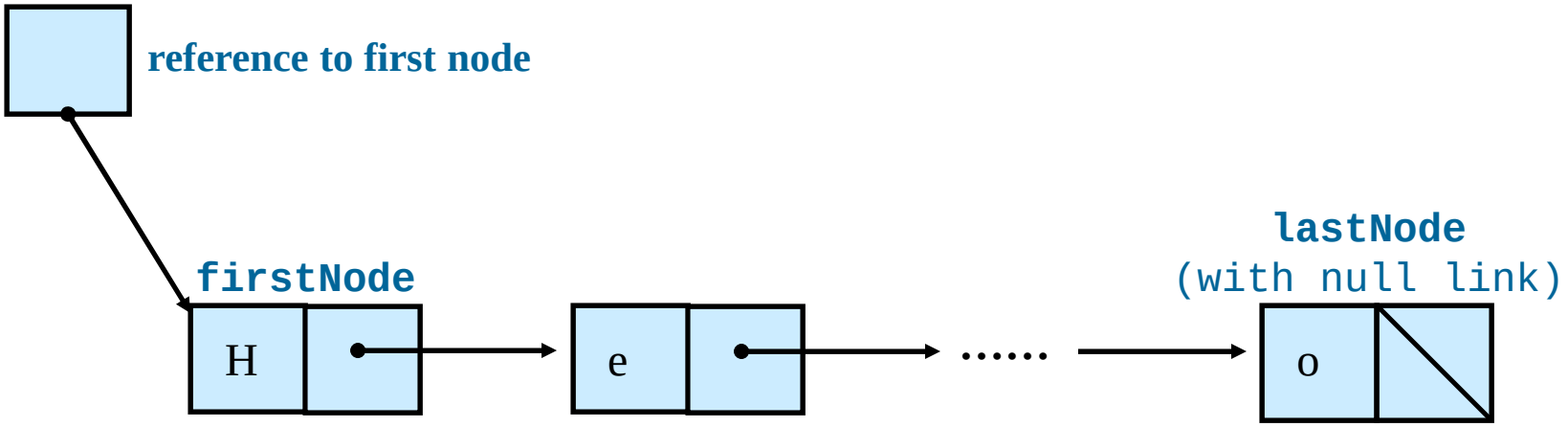


Fig. 23.2 Two self-referential class objects (nodes) linked together.

25.4 Linked Lists

- Linked List:
  - Linear collection of self-referential nodes connected by links
    - Nodes: class objects of linked-lists
    - Programs access linked lists through a reference to first node
      - Subsequent nodes accessed by link-reference members
      - Last node's link set to null to indicate end of list
    - Nodes can hold data of any type
    - Nodes created dynamically



25.4 Linked Lists

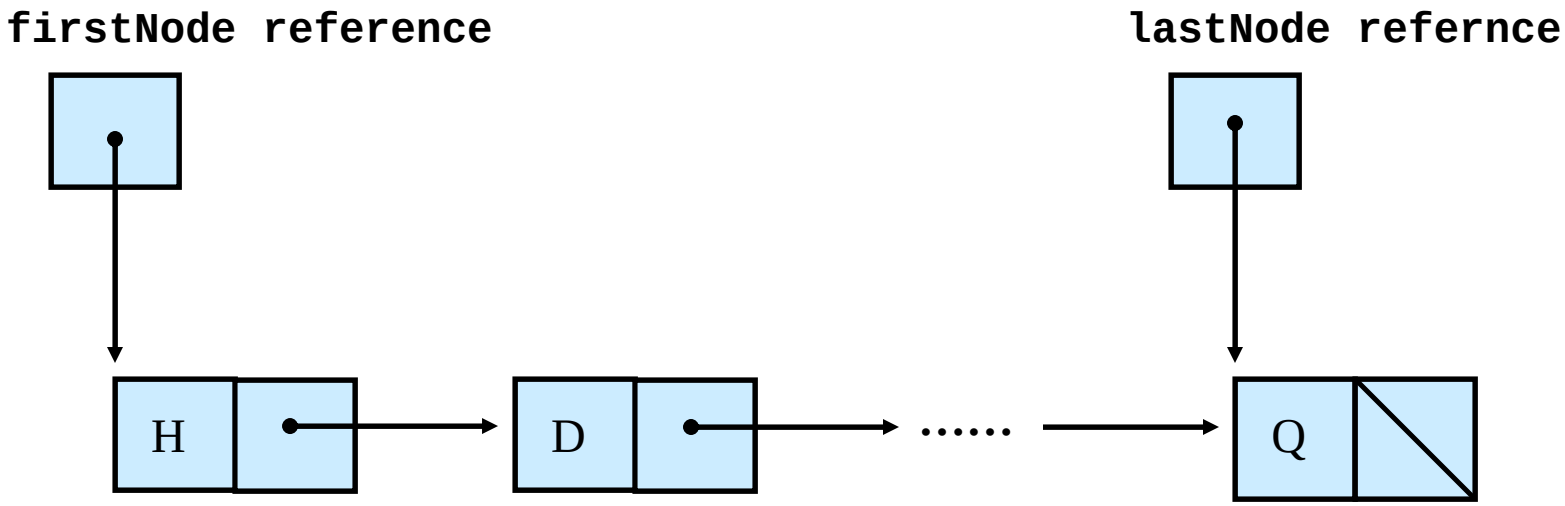


Fig. 23.3 A graphical representation of a linked list (with an optional reference to the last node).

25.4 Linked Lists

- Linked lists - similar to arrays

  However:

  - Arrays are a fixed size
  - Linked lists have no limit to size
    - More nodes can be added as program executes

25.4 Linked Lists - InsertAtFront

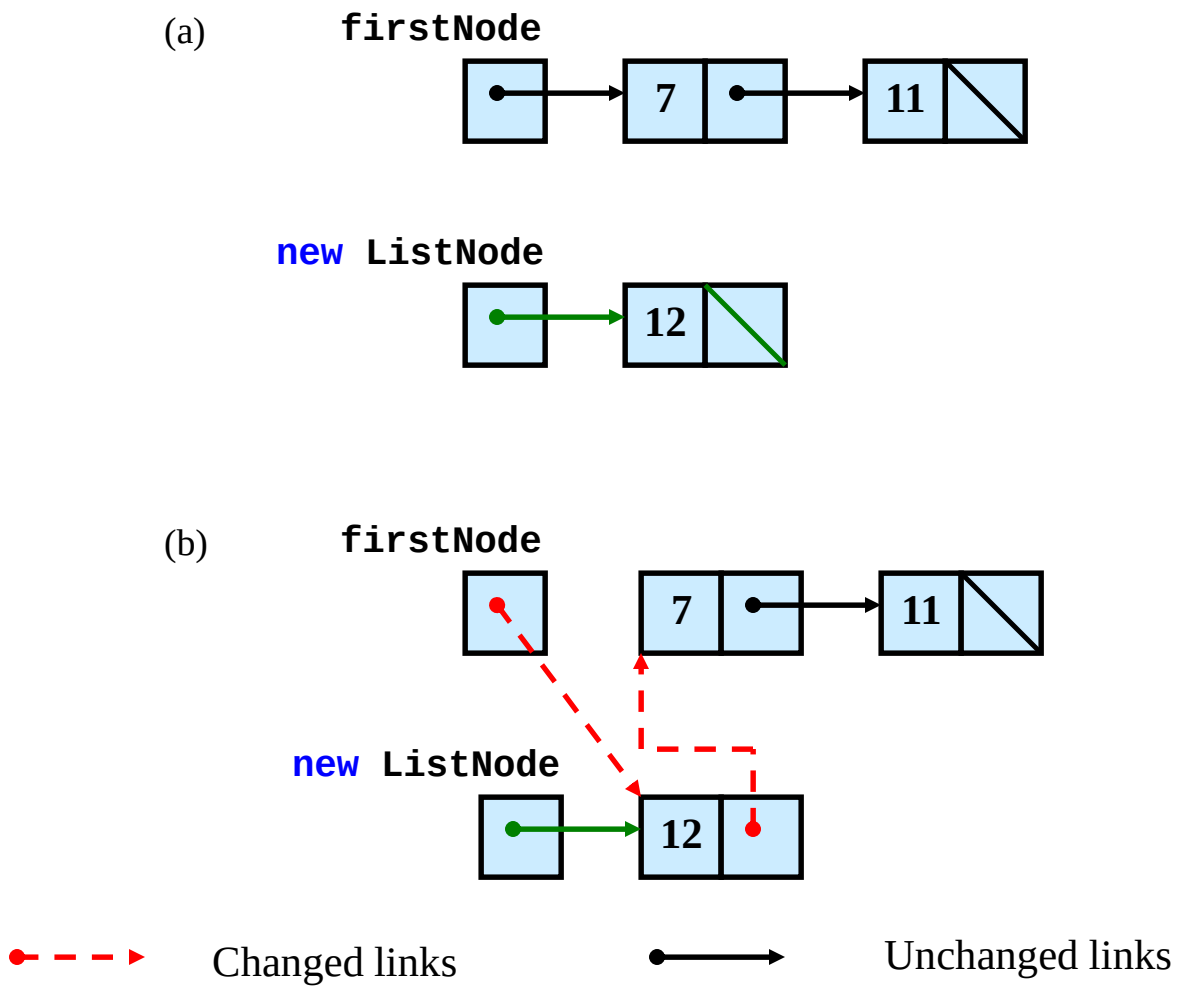


Fig. 23.6 A graphical representation of the `InsertAtFront` operation.

25.4 Linked Lists - InsertAtBack

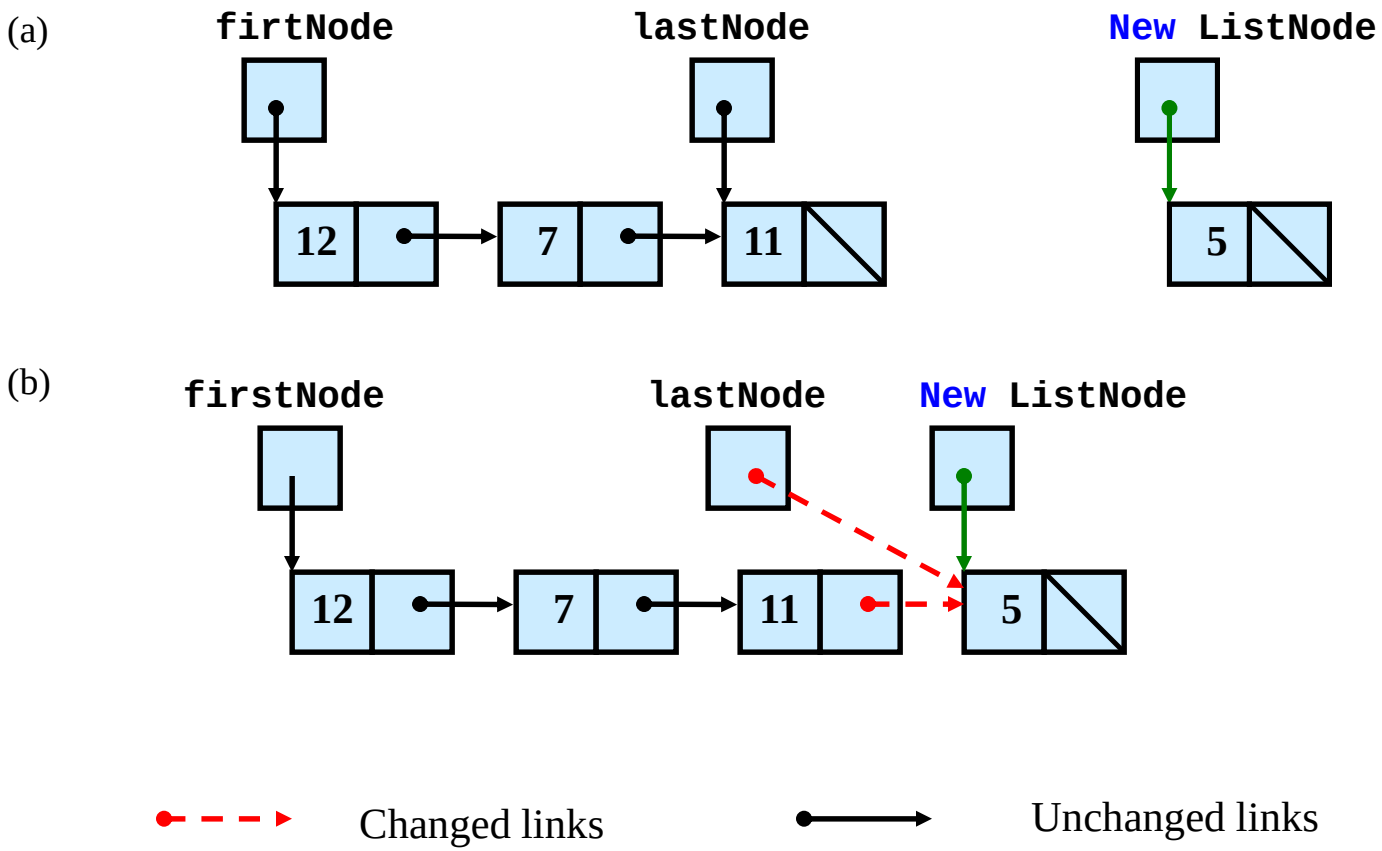


Fig. 23.7 A graphical representation of the `InsertAtBack` operation.

25.4 Linked Lists - RemoveFromFront

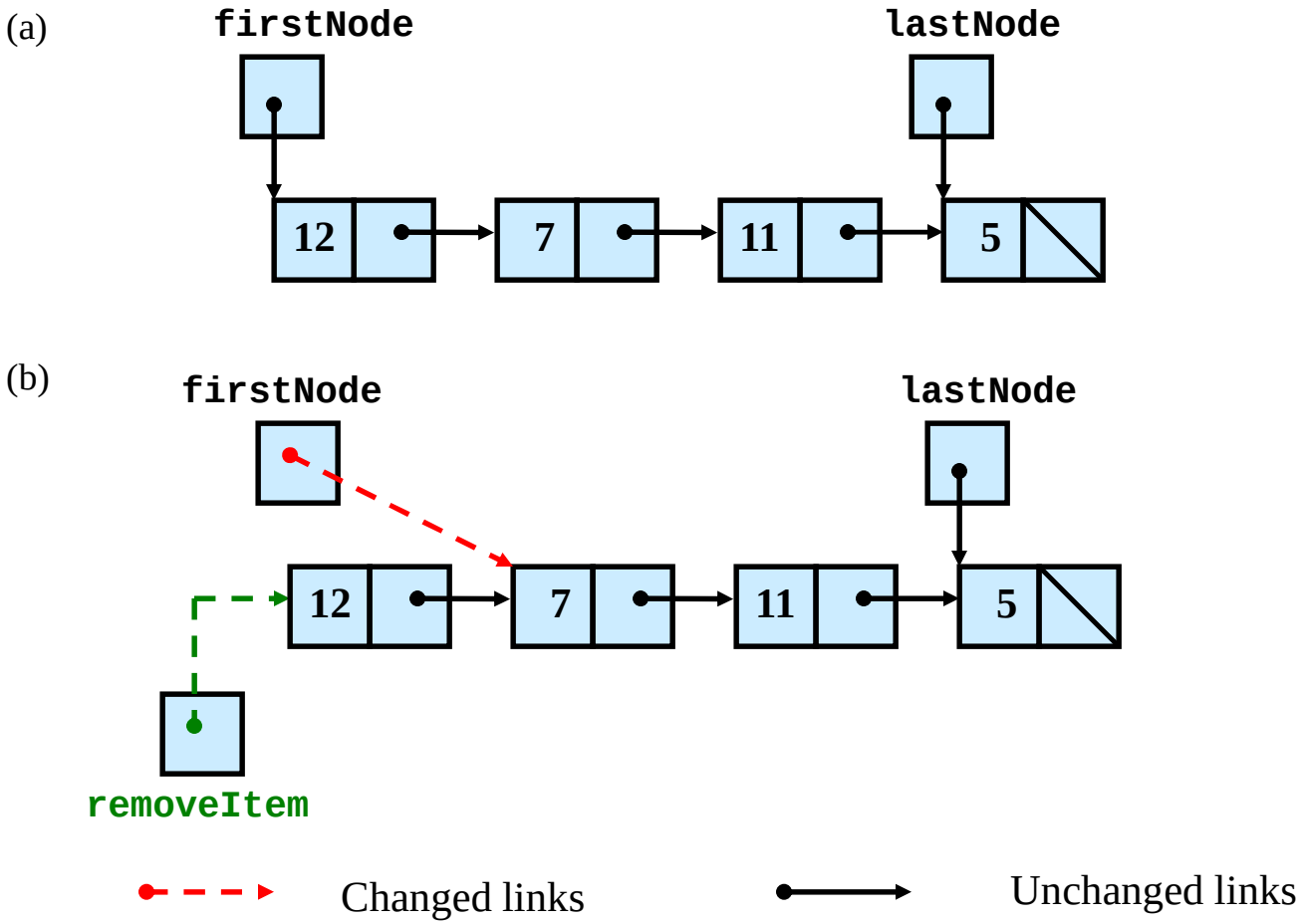


Fig. 23.8 A graphical representation of the `RemoveFromFront` operation.

25.4 Linked Lists - RemoveFromBack

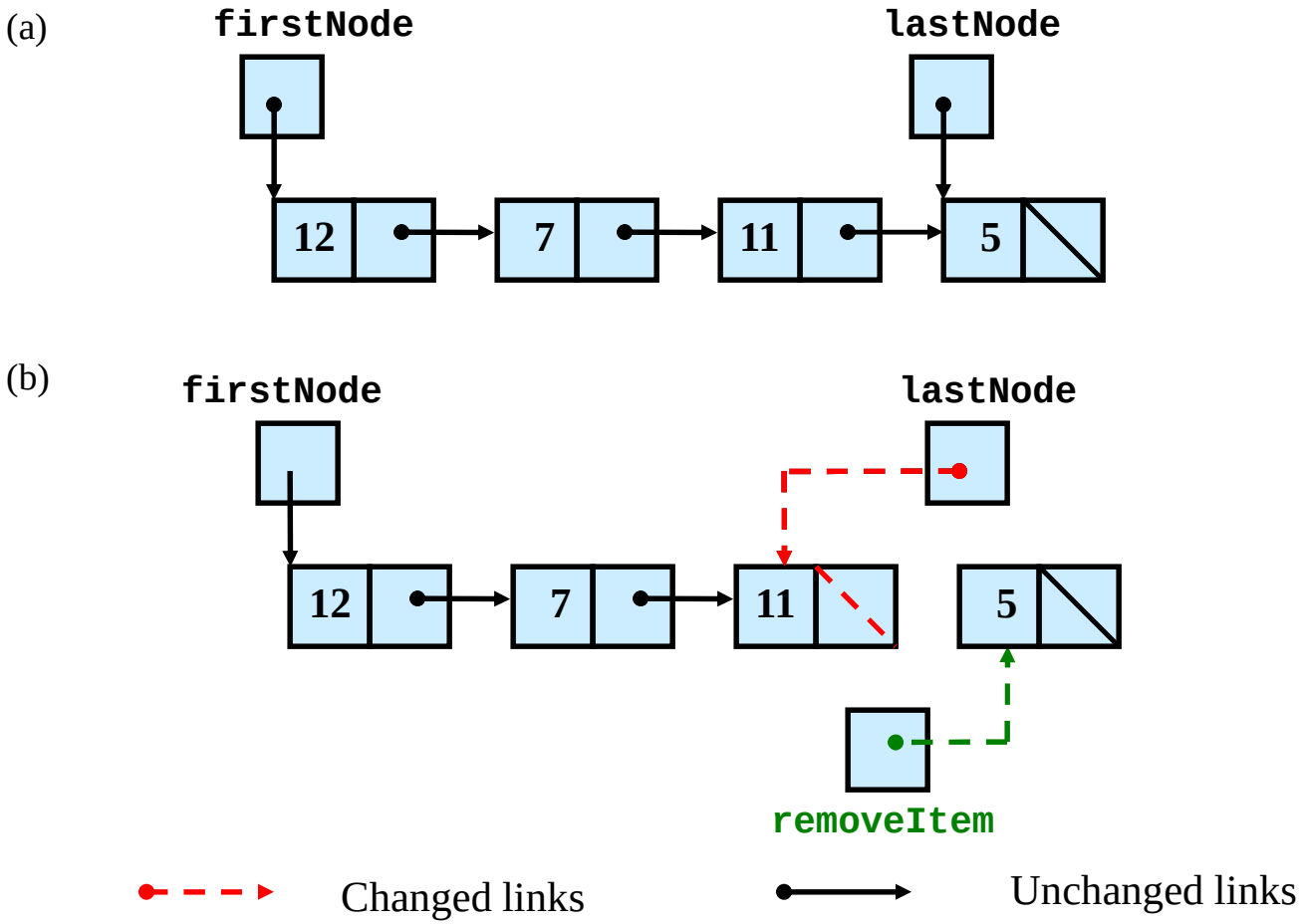
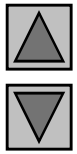


Fig. 23.9 A graphical representation of the `RemoveFromBack` operation.



Outline

LinkedListLibrary.cs

```
1    // Fig. 23.4: LinkedListLibrary.cs
2    // Class ListNode and class List definitions.
3
4    using System;
5
6    namespace LinkedListLibrary     // create library
7    {
8       // class to represent one node in a list
9       class ListNode
10      {
11         private object data;   // any o                st node
12         private ListNode next; // self                 able
13
14         // constructor to create ListNode that refers to dataValue
15         // and is last node in list (added to the end of the list)
16         public ListNode( object dataValue )
17            : this( dataValue, null ) // calls constructor below
18         {
19         }
20
21         // constructor to create ListNode that refers
22         // and refers to next ListNode in List (added
           // position other than the last one)
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            }
```

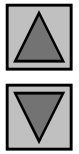
Reference to next ListNode in linked list

Constructor for first item in list

Invoke normal constructor, which will set data to dataValue and next to null

Constructor to set data and next values to parameter values

Accessor method so a List can access next member variable



Outline

LinkedListLibrary.cs

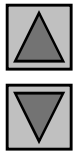
```
36
37          set
38          {
39             next = value;
40          }
41       }
42
43       // property Data
44       public object Data
45       {
46          get
47          {
48             return data;
49          }
50       }
51
52    } // end class ListNode
53
54    // class List definition
55    public class List
56    {
57       private ListNode firstNode;
58       private ListNode lastNode;
59       private string name;     // string li
60
61       // construct empty List with specified name
62       public List( string listName )
63       {
64          name = listName;
65          firstNode = lastNode = null;
66       }
67
```

Accessor method so a List can access data member variable

Reference to first node in list

Reference to last node in list

Set reference to first and last nodes to null



Outline

LinkedListLibrary.cs

```
68          // construct empty List with "list" as its name
69          public List() : this( "list" )  // uses previous constructor
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 ) // ignore-needed in multithreaded e
79             {
80                if ( IsEmpty() )  // defined in Line 105 here
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
92          public void InsertAtBack( object insertItem )
93          {
94             lock ( this ) // ignore-needed in multithreaded
95             {
96                if ( IsEmpty() )  // defined in L
97                   firstNode = lastNode =
98                      new ListNode( insertItem );
99
```

Method to insert an object at the front of the list

Get list lock

Test if list is empty

If list is empty, create new node and set firstNode and lastNode to refer to it

If list is not empty, insert object by setting its next reference to the first node

Method to insert object into back of list

Get list lock

Test if list is empty

If list is empty create a new node and set firstNode and lastNode to reference it

Outline

```
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 ) // ignore-needed in mult
110            {
111               object removeItem = null;
112
113               if ( IsEmpty() )  // defined in Line 163 b
114                  throw new EmptyListException( name );
115
116               removeItem = firstNode.Data;  // retrieve data
117
118               // reset firstNode and lastNode references
119               if ( firstNode == lastNode )
120                  firstNode = lastNode = null;
121
122               else
123                  firstNode = firstNode.Next;
124
125               return removeItem;  // return removed data
126            }
127         }
128
129         // remove last node from List
130         public object RemoveFromBack()
131         {
132            lock ( this ) // ignore-needed
133            {
134               object removeItem = null;
```

If list is not empty, create a new node and set the last node's next reference to the new node

Method to remove an object from the front of list

Get lock

Throw exception if list is empy

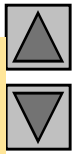
Set removeItem equal to data in first node

If there is only one node in the list, set firstNode and lastNode references to null

If there is more then one node, set firstNode to reference the second node

Return data stored in node

Method to remove an object from the back of the list



Outline

LinkedListLibrar

```
135
136            if ( IsEmpty() )
137               throw new EmptyListException( name );
138
139            removeItem = lastNode.Data;   // retrieve dat
140
141            // reset firstNode and lastNode references
142            if ( firstNode == lastNode ) // only one nod
143               firstNode = lastNode = null;
144
145            else
146            {  // walk the list until current precedes lastNode
147               ListNode current = firstNode;
148
149               // loop while current node is not lastNode
150               while ( current.Next != lastNode )
151                  current = current.Next;    // move to next n
152               // current is now node next to the last
153               // make current new lastNode
154               lastNode = current;
155               current.Next = null;
156            }
157
158            return removeItem; // return remo
159         }
160      }
161
162      // return true if List is empty
163      public bool IsEmpty()
164      {
165         lock ( this ) // ignore-needed in
166         {
167            return firstNode == null;  // checks if firstNode == null
168         }
169      }
```

Set removeItem equal to the data in the last node

If there is only one node, set firstNode and lastNode to refer to null

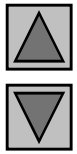
Loop until next to last node is reached

Set lastNode to refer to the next to last node

Set reference of new last node to null

Return data of old last node

Method to test if list is empty



Outline

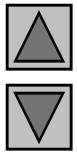
LinkedListLibrary.cs

```
170
171        // output List contents
172        virtual public void Print()
173        {
174           lock ( this ) // ignore-needed in multithreaded environ.
175           {
176              if ( IsEmpty() ) // defined in Line 163 above
177              {
178                 Console.WriteLine( "Empty " + name );
179                 return;
180              }
181
182              Console.Write( "The " + name + " is: " );
183
184              ListNode current = firstNode;
185
186              // output current node data while not at end of list
187              while ( current != null )
188              {
189                 Console.Write( current.Data + " " );
190                 current = current.Next;
191              }
192
193              Console.WriteLine( "\n" );
194           }
195        }
196
197     } // end class List
198
```

Method to output the list

Tell user if list is empty

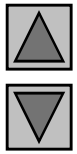
Output data in list



Outline

```
199     // class EmptyListException definition
200     public class EmptyListException : ApplicationException
201     {
202        public EmptyListException( string name )
203           : base( "The " + name + " is empty" )
204        {
205        }
206
207     } // end class EmptyListException
208
209  } // end namespace LinkedListLibrary
```

Handles illegal operations on an empty list



Outline

ListTest.cs

```
1    // Fig 23.5: ListTest.cs
2    // Testing class List.
3
4    using System;
5    using LinkedListLibrary;      // library we created
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
```

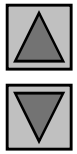
Create list of objects

Create data to put in list

Insert objects at beginning of listinto list using InsertAtFront method

Insert objects at end of listinto list using InsertAtBack method

Print the list



Outline

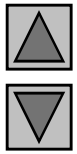
```
35            // remove data from list and print after each removal
36            try
37            {
38               removedObject = list.RemoveFromFront();
39               Console.WriteLine( removedObject + " removed" );
40               list.Print();
41
42               removedObject = list.RemoveFromFront();
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
62         } // end method Main
63
64      } // end class ListTest
65   }
```

Remove objects from front of list using method RemoveFromFront

Print the list after each remove

Remove objects from back of list using method RemoveFromBack

If remove is called on an empty list tell user



Outline

ListTest.cs

Program Output

```
The list is: True

The list is: $ True

The list is: $ True 34567

The list is: $ True 34567 hello

$ removed
The list is: True 34567 hello

True removed
The list is: 34567 hello

hello removed
The list is: 34567

34567 removed
Empty list
```

We have already seen this and next 3 slides - Review yourself if needed

25.4 Linked Lists - InsertAtFront

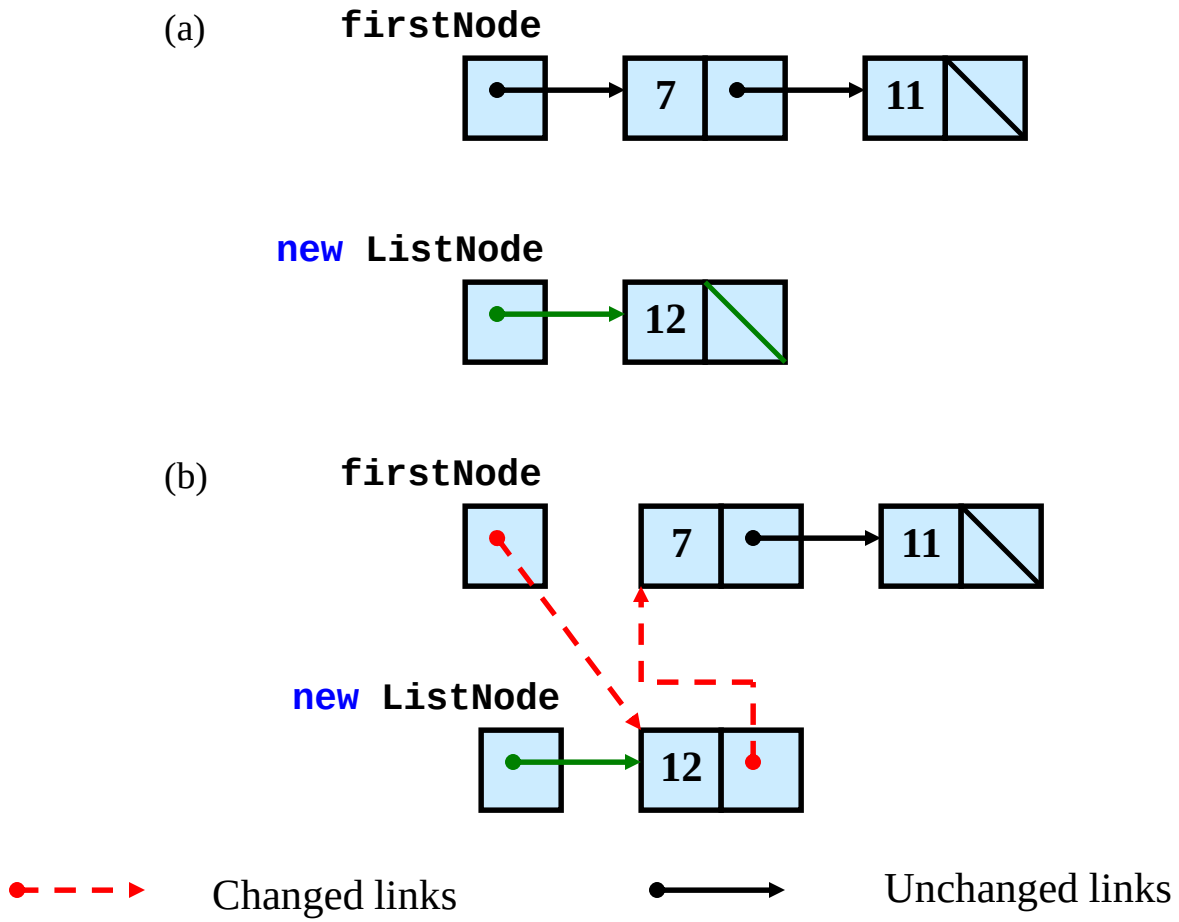


Fig. 23.6 A graphical representation of the `InsertAtFront` operation.

25.4 Linked Lists - InsertAtBack

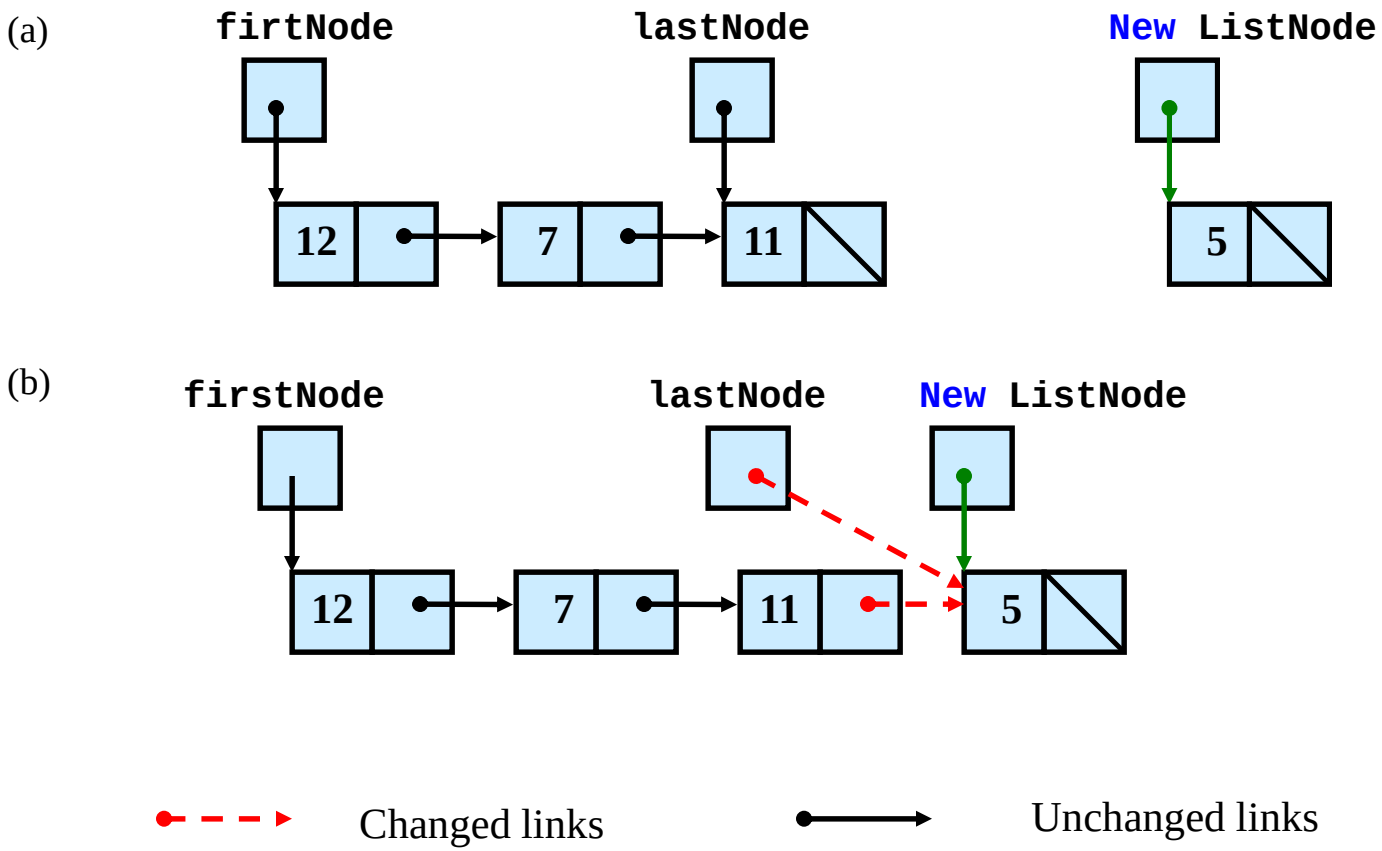


Fig. 23.7 A graphical representation of the `InsertAtBack` operation.

25.4 Linked Lists - RemoveFromFront

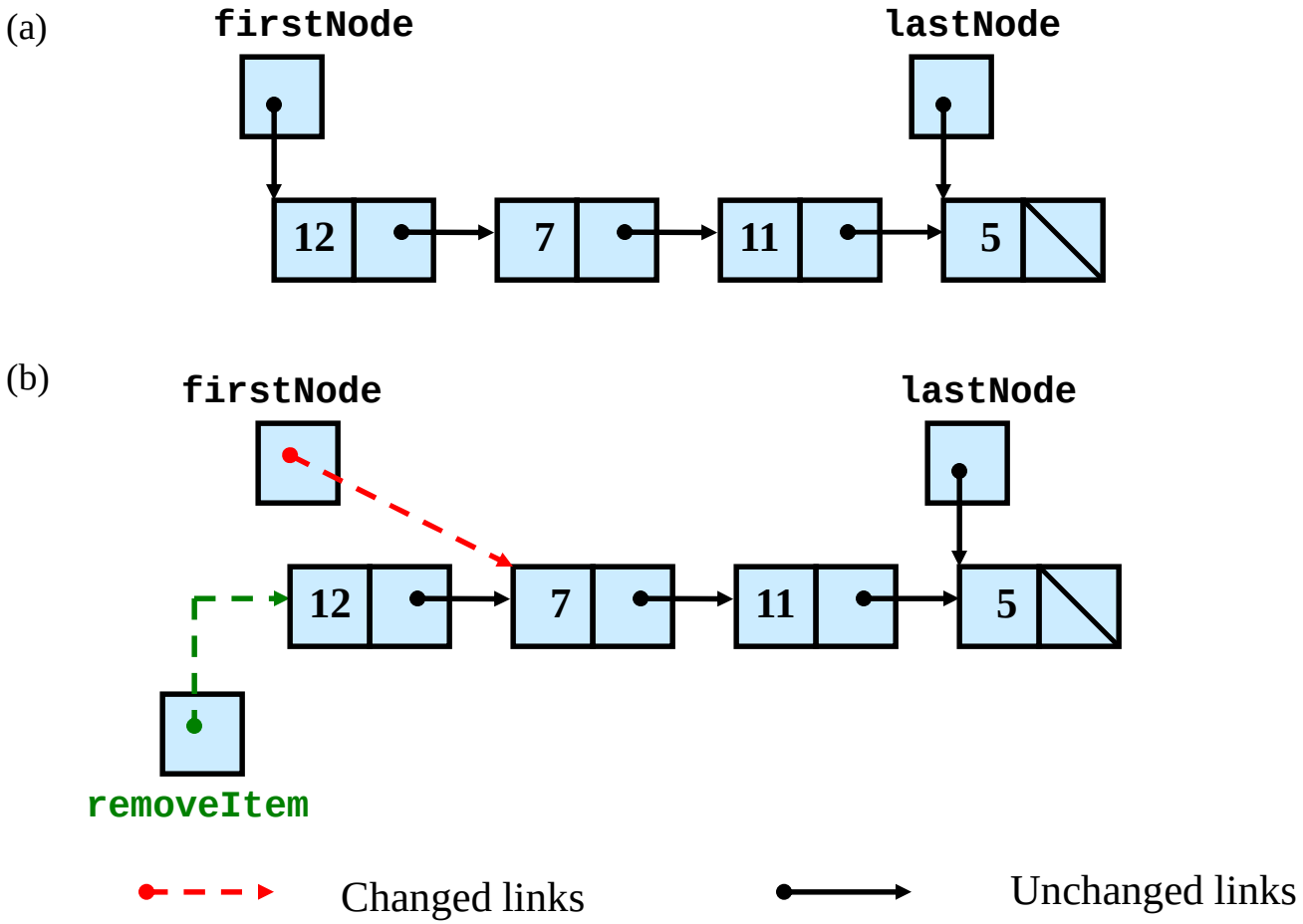


Fig. 23.8 A graphical representation of the `RemoveFromFront` operation.

25.4 Linked Lists - RemoveFromBack

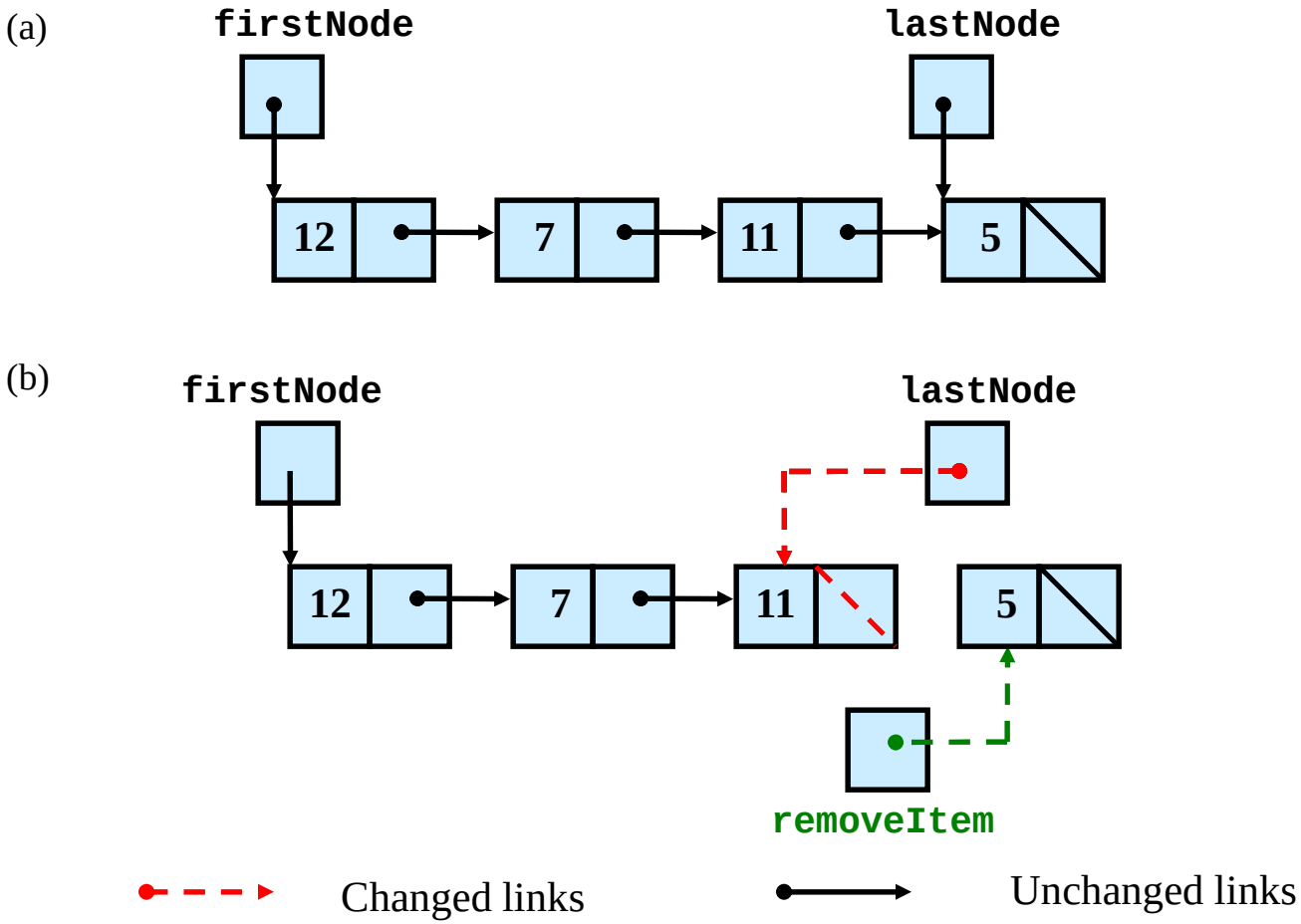
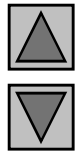


Fig. 23.9 A graphical representation of the `RemoveFromBack` operation.

25.5 Stacks

- Stack – a special version of a linked list:
  - Last-in, first-out (LIFO) data structure:
    - Takes and releases new nodes only at top
- Operations:
  - Push: adds new entry (node) to top of stack
  - Pop: removes top entry (node) from stack
- Can be used for:
  - Storing return addresses
  - Storing local variables
  - …



Outline

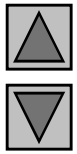
StackInheritance Library.cs

```
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  }
```

StackInheritance class is derived from List class

Call InsertAtFront method of List class to push objects

Call RemoveFromFront method of List class to pop objects



Outline

StackInheritanceTest.cs

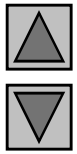
```
1  // Fig. 23.11: StackInheritanceTest.cs
2  // Testing class StackInheritance.
3
4  using System;
5  using StackInheritanceLibrary;
6  using LinkedListLibrary;
7
8  namespace StackInheritanceTest
9  {
10    // demonstrate functionality of class StackInheritance
11    class StackInheritanceTest
12    {
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          object removedObject = null;
35
```

Create stack

Create objects to store in stack

Push objects onto the stack

Print stack after each push



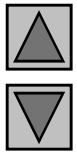
Outline

StackInheritance

```
36            // remove items from stack
37            try
38            {
39               while ( true )
40               {
41                  removedObject = stack.Pop();
42                  Console.WriteLine( removedObject + " popped" );
43                  stack.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 StackInheritanceTest
57   }
```

Remove objects from stack

Empty stack exception



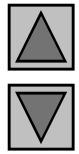
Outline

StackInheritanceTest.cs

Program Output

```
The stack is: True

The stack is: $ True

The stack is: 34567 $ True

The stack is: hello 34567 $ True

hello popped
The stack is: 34567 $ True

34567 popped
The stack is: $ True

$ popped
The stack is: True

True popped
Empty stack
   at LinkedListLibrary.List.RemoveFromFront()
      in z:\ch24\linkedlistlibrary\linkedlistlibrary.cs:line 114
   at StackInheritanceLibrary.StackInheritance.Pop()
      in z:\ch24\stackinheritancelibrary\
      stackinheritancelibrary.cs:line 28
   at StackInheritanceTest.StackInheritanceTest.Main(String[] args
      in z:\ch24\fig24_11\stackinheritancetest.cs:line 41
```

Line 144 - Slide 16



Outline

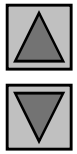
StackComposition Library.cs

```
1    // Fig. 23.12: StackCompositionLibrary.cs
2    // StackComposition definition with composed List object.
3                    // (no inheritance here!)
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
```

Create List object

Call method InsertAtFront to push

Use method RemoveFromFront to pop



Outline

StackComposition Library.cs

```
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   }
```

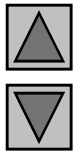
Call is empty to see if list has nodes

Call method Print for output

25.6 Queues

- Queue:

  First-in, first-out (FIFO) data structure
  - Nodes added to tail, removed from head
- Operations:
  - Enqueue: insert node at the end
  - Dequeue: remove node from the front
- Many computer applications:
  - Printer spooling
  - Information packets on networks
  - …



Outline

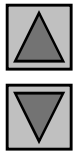
QueueInheritanceLibrary.cs

```
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  }
```

Class QueueInheritance derives from class List

Call InsertAtBack to enqueue

Call RemoveFromFront to dequeue



Outline

QueueTest.cs

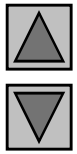
```
1    // Fig. 23.14: QueueTest.cs
2    // Testing class QueueInheritance.
3
4    using System;
5    using QueueInheritanceLibrary;
6    using LinkedListLibrary;
7
8    namespace QueueTest
9    {
10      // demonstrate functionality of class QueueInheritance
11      class QueueTest
12      {
13         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
```

Create queue

Create objects to be inserted into queue

Enqueue objects

Print queue after each enqueue



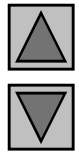
Outline

QueueTest.cs

```
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   }
```

Dequeue objects

Print queue after each enqueue



Outline

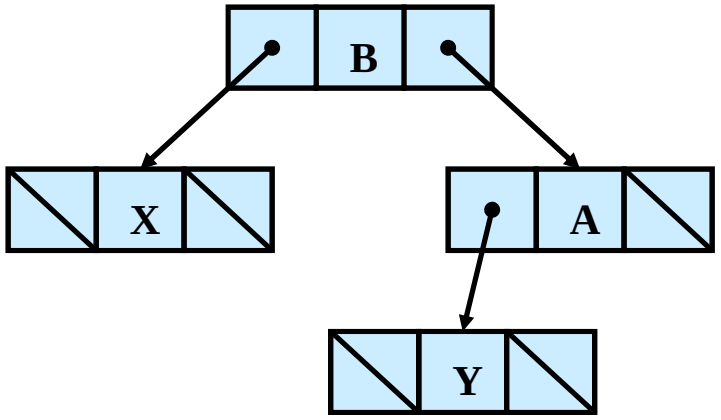
QueueTest.cs

Program Output

```
The queue is: True

The queue is: True $

The queue is: True $ 34567

The queue is: True $ 34567 hello

True dequeue
The queue is: $ 34567 hello

$ dequeue
The queue is: 34567 hello

34567 dequeue
The queue is: hello

hello dequeue
Empty queue
   at LinkedListLibrary.List.RemoveFromFront()
      in z:\ch24\linkedlistlibrary\linkedlistlibrary.cs:line 114
   at QueueInheritanceLibrary.QueueInheritance.Dequeue()
      in z:\ch24\queueinheritancelibrary\
      queueinheritancelibrary.cs:line 28
   at QueueTest.QueueTest.Main(String[] args)
      in z:\ch24\fig24_13\queuetest.cs:line 41
```

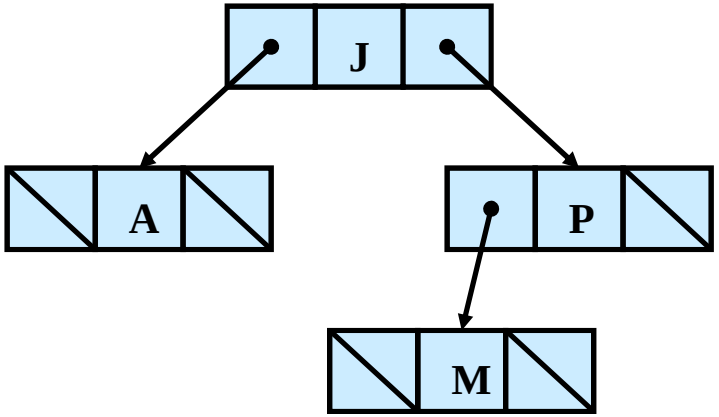
25.7 Trees

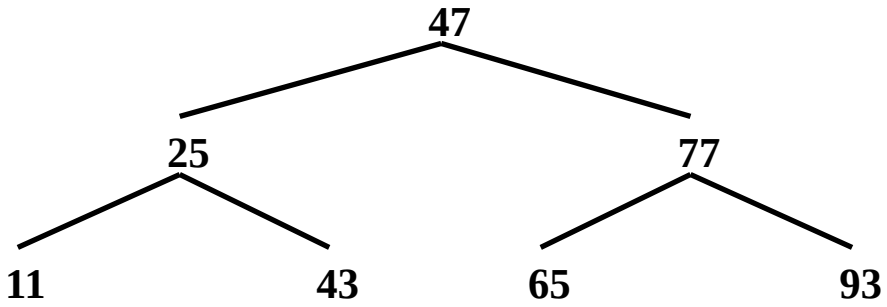
- Tree: non-linear, two-dimensional data structure
- Binary tree:
  - Each node contains data & two links (hence "binary in the name): left link and right link
  - Root node: "top" node in a tree
    - Links refer to child nodes
  - Leaf node: node with no children



25.7 Trees

- Binary search tree:
  - Values in left subtree are less than the value of the subtree's parent
  - Values in right subtree are greater than the value of the subtree's parent



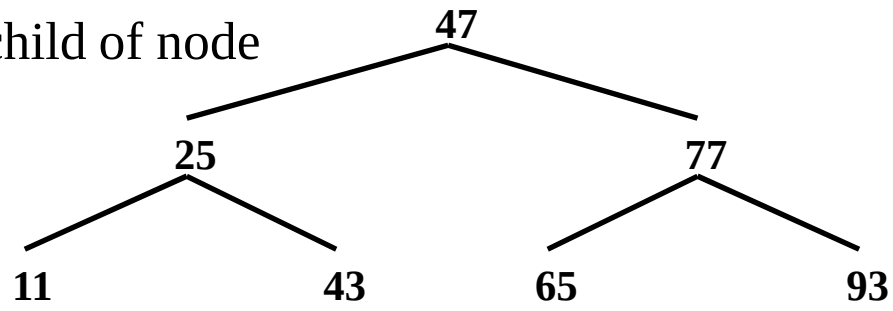


25.7.1 Binary Search Tree of Integer Values

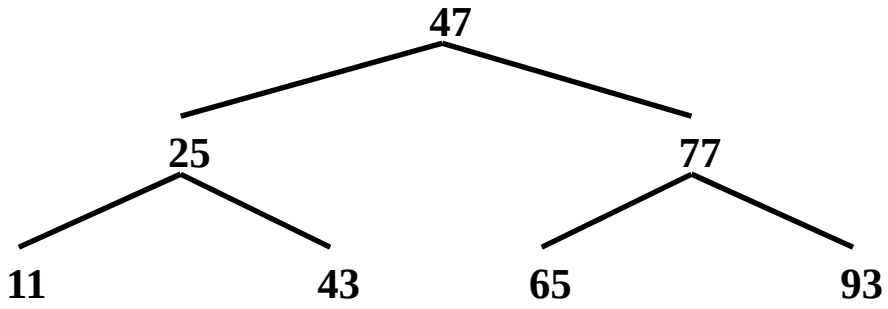
- Traversal: method of retrieving data from a tree
  - In these methods if there is a subtree, recursively the traversal is called recursively
- Kinds of traversals:
  - Inorder traversal:
    - Get data from left subtree/child of node
    - Get data from node
    - Get data from right subtree/child of node
- Example (figure)

  Inorder traversal:

  11 25 43 47 65 77 93



25.7.1 Binary Search Tree of Integer Values



- Preorder traversal:
  - Get data from node
  - Get data from left subtree/child of node
  - Get data from right subtree/child of node
  - Example: 47 25 11 43 77 65 93
- Postorder traversal
  - Get data from left subtree/child of node
  - Get data from right subtree/child of node
  - Get data from node
  - Example: 11 43 25 65 93 77 47
- Level-order traversal
  - Visit nodes of tree row by row, from left to right
  - Example: 47 25 77 11 43 65 93

25.7 Trees

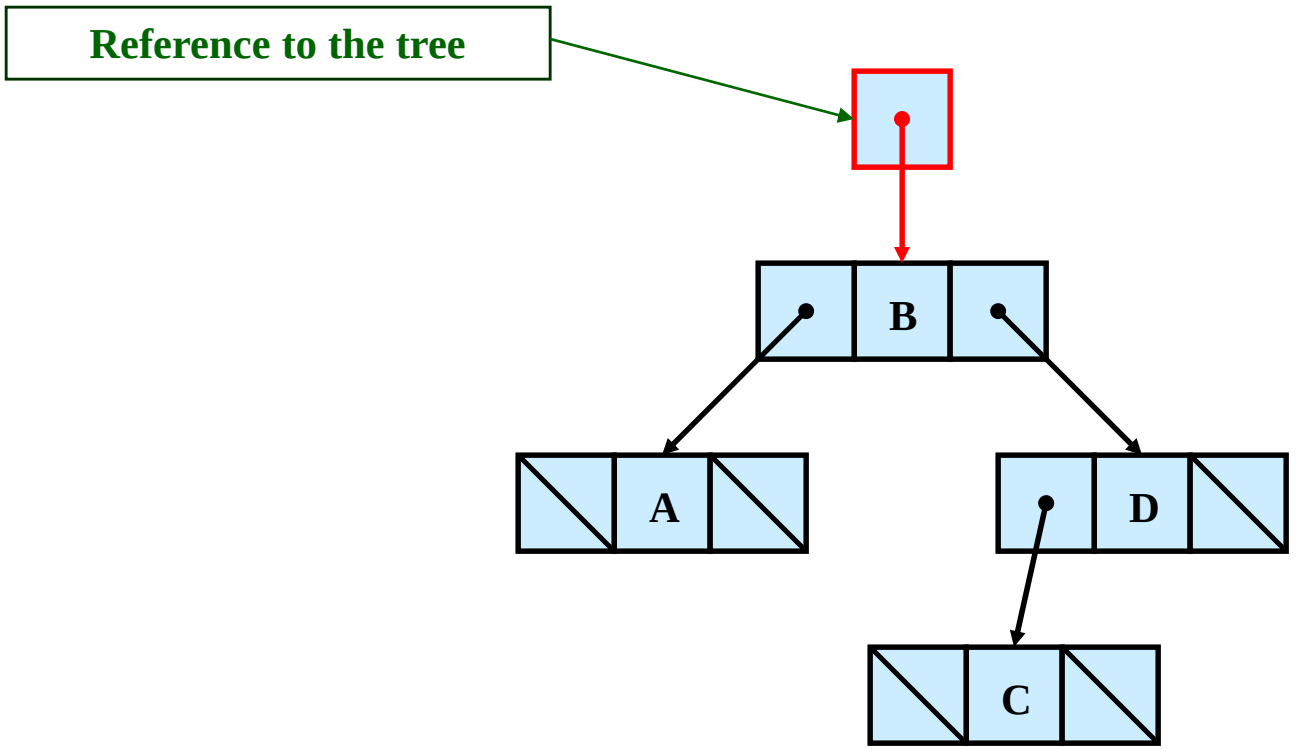


Fig. 23.15 A graphical representation of a binary tree.

25.7 Trees

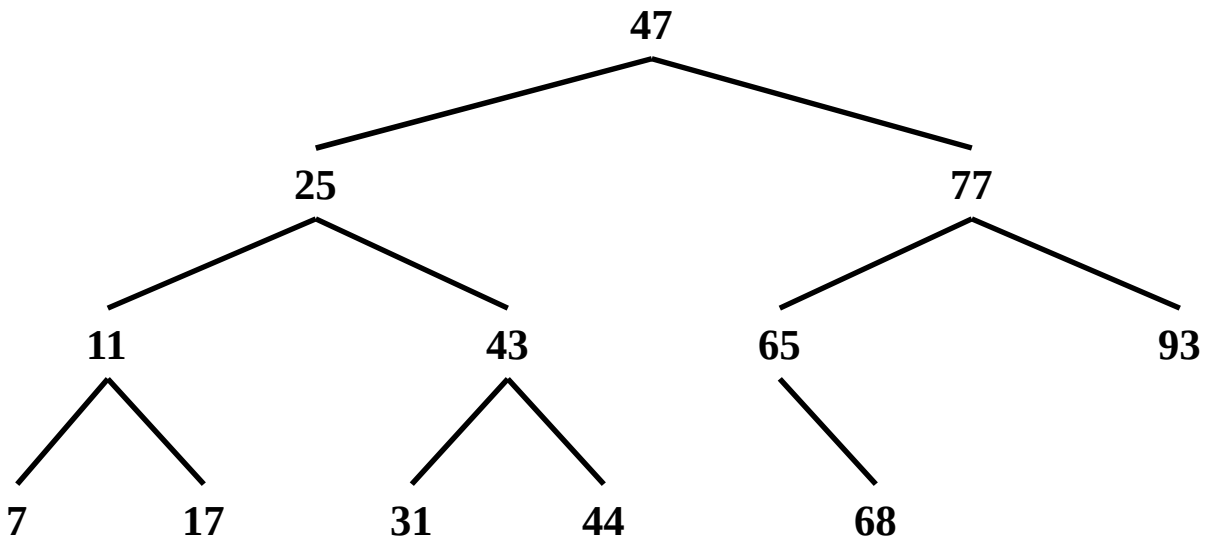
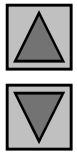


Fig. 23.16 A binary search tree containing 12 values.



Outline

BinaryTreeLibrary.cs

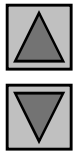
```
1    // Fig. 23.17: BinaryTreeLibrary.cs
2    // Definition of class TreeNode and class Tree.
3
4    using System;
5
6    namespace BinaryTreeLibrary
7    {
8       // class TreeNode definition
9       class TreeNode
10      {
11         private TreeNode leftNode;
12         private int data;
13         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
```

Left and right subtree references

Data stored in node

Since new nodes are leaf, set subtree references to null

Accessor methods for left subtree



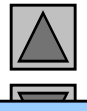
Outline

BinaryTreeLibrary.cs

```
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
```

Accessor methods for nodes data

Accessor methods for right subtree



Outline

TreeLibrary.cs

```
65          // insert TreeNode into Binary Search Tree containing 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             // ignore if insertValue == data (duplicate value)
93          }  // end method Insert
94
95       }  // end class TreeNode
96
```

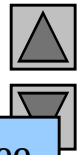
Method to determine location of new node

If value of new node is less than root, and the left subtree is empty, insert node as left child of root

If left subtree is not empty, recursively call Insert to determine location of new node in subtree

If value of new node is greater than root, and the right subtree is empty, insert node as right child of root

If right subtree is not empty, recursively call Insert to determine location of new node in subtree



Outline

BinaryTreeLibrary.cs

```
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 )   // ignore locks
114         {
115            if ( root == null )
116               root = new TreeNode( insertValue );
117
118            else
119               root.Insert( insertValue ); // s
120         }
121      }
122
123      // begin preorder traversal
124      public void PreorderTraversal()
125      {
126         lock ( this )
127         {
128            PreorderHelper( root );
129         }
130      }
131
```

Reference to root of tree

Set root to null when tree first created

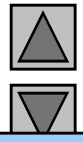
Method to insert a new node into tree

If tree is empty insert new node as root

If tree is not empty call Insert to determine location of new node

Perform preorder traversal

Call PreorderHelper to help perform traversal



Outline

```
132        // recursive method to perform preorder traversal
133        private void PreorderHelper( TreeNode node )
134        {
135           if ( node == null )
136              return;
137
138           // output data from this node
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
```

Method to help with preorder traversal

Display node data

Call PreorderHelper recursively on left subtree

Call PreorderHelper recursively on right subtree

Perform inorder traversal

Call InorderHelper to help with traversal

Method to help with inorder traversal

Call InorderHelper recursively on left subtree

Outline

```
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 }
```

Display node data

Call InorderHelper recursively on right subtree

Perform postorder traversal

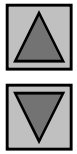
Call PostorderHelper to help with traversal

Method to help with postorder traversal

Call PostorderHelper recursively on left subtree

Call PostorderHelper recursively on right subtree

Display node data



Outline

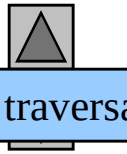
TreeTest.cs

```
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 traveral of tree
31            Console.WriteLine( "\n\nPreorder traversal" );
32            tree.PreorderTraversal();
33
```

Create a tree

Insert ten nodes in tree

Call preorder traversal



Outline

```
34            // perform inorder traveral of tree
35            Console.WriteLine( "\n\nInorder traversal" );
36            tree.InorderTraversal();
37
38            // perform postorder traveral of tree
39            Console.WriteLine( "\n\nPostorder traversal" );
40            tree.PostorderTraversal();
41            Console.WriteLine();
42         }
43
44      }  // end class TreeTest
45   }
```

Call inorder traversal

Call postorder traversal

TreeTest.cs

```
Inserting values:
39 69 94 47 50 72 55 41 97 73

Preorder traversal
39 69 47 41 50 55 94 72 73 97

Inorder traversal
39 41 47 50 55 69 72 73 94 97

Postorder traversal
41 55 50 47 73 72 97 94 69 39
```

Program Output

25.7 Binary Search Tree of Integer Values

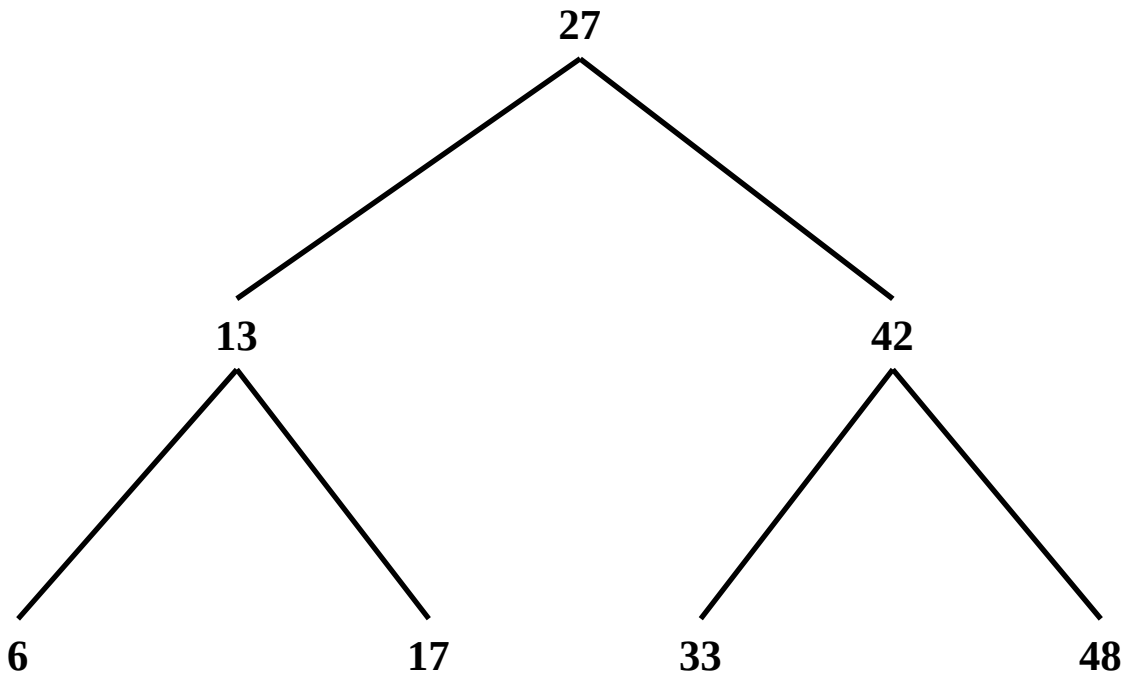
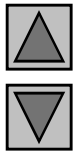


Fig. 23.19 A binary search tree.

25.7.2 Binary Search Tree of `IComparable` Objects

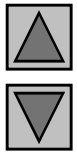
- Can use polymorphism to manipulate objects of different types in uniform ways
  - Binary search trees can be implemented to manipulate data of any object that implements the **`IComparable`** interface
- Implementation of **`IComparable`** defines:
  - CompareTo method
    - E.g.:
```
public void Insert( IComparable insertValue )
{
   ...
   if ( insertValue.CompareTo( data ) < 0 )
      ...
}
```
  - insertValue.CompareTo(data) returns value:
    - < 0 if insertValue < data
    - = 0 if they are equal
    - > 0 if insertValue > data



Outline

BinaryTreeLibrary2.cs

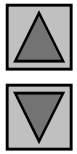
```
1    // Fig. 23.20: BinaryTreeLibrary2.cs
2    // Definition of class TreeNode and class Tree for IComparable
3    // objects.
4
5    using System;
6
7    namespace BinaryTreeLibrary2
8    {
9       // class TreeNode definition
10      class TreeNode
11      {
12         private TreeNode leftNode;
13         private IComparable data;   // polymorphic data stored in node
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         }
```



Outline

BinaryTreeLibrary2.cs

```
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
```



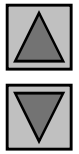
Outline

BinaryTreeLibrar

```
65          // insert TreeNode into Tree that contains nodes;
66          // ignore duplicate values
67          public void Insert( IComparable insertValue )
68          {  // insertValue of polymorphic-enabling type IComparable
69             // insert in left subtree
70             if ( insertValue.CompareTo( data ) < 0 ) //
71             {  // 'insertValue < data' not sufficient –
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 );  // recursive
79             }
80
81             // insert in right subtree
82             else if ( insertValue.CompareTo( data ) > 0 )
83             {// 'insertValue > data' not sufficient – cf. sl.47
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 );  // recursive
91             }
92    // ignore if insertValue.CompareTo( data ) == 0 (duplicate value)
93          }  // end method Insert
94
95       }  // end class TreeNode
96
```

Use Icomparable's method CompareTo to determine if new node is less than its parent

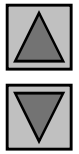
Use Icomparable's method CompareTo to determine if new node is greater than its parent



Outline

BinaryTreeLibrary2.cs

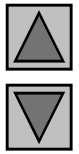
```
97    // class Tree definition
98    public class Tree  // differences w.r.t. Fig. 23.17 -  in red
99    {
100      private TreeNode root;
101
102      // construct an empty Tree of integers
103      public Tree()
104      {
105         root = null;
106      }
107
108      // Insert a new node in 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 ); // use Insert from
120         }    // previous slide to insert new node into tree
121      }       // rooted at 'root'
122
123      // begin preorder traversal
124      public void PreorderTraversal()
125      {
126         lock ( this )
127         {
128            PreorderHelper( root );
129         }
130      }
131
```



Outline

BinaryTreeLibrary2.cs

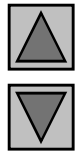
```
132        // recursive method to perform preorder traversal
133        private void PreorderHelper( TreeNode node )
134        {
135           if ( node == null )
136              return;
137
138           // output node data
139           Console.Write( node.Data + " " );
140
141           // traverse 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
```



Outline

BinaryTreeLibrary2.cs

```
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 }
```



Outline

TreeTest.cs

```
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
                  // - using: public public class Tree
22            Tree intTree = new Tree();   // empty tree
23            populateTree( intArray, intTree, "intTree" ); // next sl.
24            traverseTree( intTree, "intTree" );           // next sl.
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         }
```

Populate trees with int, double and string values

Outline

```
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 traveral of tree
54             Console.WriteLine(
55                "\n\nPreorder traversal of " + treeType );
56             tree.PreorderTraversal();
57
58             // perform inorder traveral of tree
59             Console.WriteLine(
60                "\n\nInorder traversal of " + treeType );
61             tree.InorderTraversal();
62
```

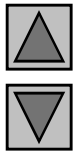
Method to add data from arrays to trees

Insert nodes into tree

Method to traverse tree

Perform preorder traversal

Perform inorder traversal



Outline

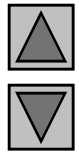
```
63            // perform postorder traveral of tree
64            Console.WriteLine(
65               "\n\nPostorder traversal of " + treeType );
66            tree.PostorderTraversal();
67            Console.WriteLine( "\n" );
68         }
69
70      }  // end class TreeTest
71   }
```

TreeTest.cs

Perform postorder traversal

```
Inserting into intTree:
8 2 4 3 1 7 5 6

Preorder traversal of intTree
8 2 1 4 3 7 5 6

Inorder traversal of intTree
1 2 3 4 5 6 7 8

Postorder traversal of intTree
1 3 6 5 7 4 2 8
```

Program Output



Outline

TreeTest.cs

Program Output

```
Inserting into doubleTree:
8.8 2.2 4.4 3.3 1.1 7.7 5.5 6.6

Preorder traversal of doubleTree
8.8 2.2 1.1 4.4 3.3 7.7 5.5 6.6

Inorder traversal of doubleTree
1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8

Postorder traversal of doubleTree
1.1 3.3 6.6 5.5 7.7 4.4 2.2 8.8


Inserting into stringTree:
eight two four three one seven five six

Preorder traversal of stringTree
eight two four five three one seven six

Inorder traversal of stringTree
eight five four one seven six three two

Postorder traversal of stringTree
five six seven one three four two eight
```

The End