Lab 13: Building and Enumerating Custom Collection Classes

Exercise 1: Implementing the IList<TItem> Interface

Task 1: Open the CustomCollections solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 13\Ex1\Starter** folder, click **CustomCollections.sln**, and then click **Open**.

Task 2: Modify the Tree class to implement the IList<TItem> interface

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. In the task list, locate the **TODO - Implement the generic IList<TItem> interface** task, and then double-click this task.

   This task is located in the **Tree** class.
3. Remove the **TODO - Implement the generic IList<TItem> interface** comment, and then modify the class definition to implement the generic **IList**

interface. Specify the value **TItem** as the type parameter (this is the type parameter that the **Tree** class references).

Your code should resemble the following code example.

```
public class Tree<TItem> : IList<TItem>,
    IBinaryTree<TItem> where TItem : IComparable<TItem>
{
    ...
}
```

4. Add method and property stubs that implement the **IList** interface:
   - In the class definition, right-click **IList**, point to **Implement Interface**, and then click **Implement Interface**.

     Visual Studio generates method stubs for each method that is defined in the interface, and adds them to the end of the class file. You will add code to complete some of these methods later in this exercise.

Task 3: Add support for indexing items in the Tree class

1. In the task list, locate the **TODO - Add a member to define node position** task, and then double-click this task.
2. Remove the **TODO - Add a member to define node position** comment, and then add code to define a private integer member named **position**.

   Your code should resemble the following code example.

```
public class Tree<TItem> : IList<TItem>, IBinaryTree<TItem> where
TItem : IComparable<TItem>
{
    ...
    public Tree<TItem> RightTree { get; set; }

    // Add a private integer variable position to define
    // the node's position in the tree.
    private int position;
    ...
}
```

3. In the class constructor, add code to initialize the **position** member to −1.

Note: The **position** member is the index for items in the tree. When you add or remove items from the tree, you will invalidate the **position** member of any following elements

in the tree. By setting the **position** member to –1, you indicate to the tree that the index has become invalid. When the application attempts to use the index to perform an action, and encounters a negative value, the application can rebuild the index by invoking the **IndexTree** method that you will add later.

Your code should resemble the following code example.

```
public Tree(TItem nodeValue)
{
    ...
    this.position = -1;
}
```

4. At the beginning of the **Add** method, add code to set the **position** member to −1.

   Your code should resemble the following code example.

```
public void Add(TItem newItem)
{
    // If we're adding something, the position field will become
    // invalid. Reset position to -1.
    this.position = -1;
    ...
}
```

5. In the task list, locate the **TODO - Set the position member to -1** task, and then double-click this task.

   This task is located in the **Remove** method.

6. Remove the **TODO - Set the position member to -1** comment, and then add code to set the **position** member to −1.

   Your code should resemble the following code example.

```
public void Remove(TItem itemToRemove)
{
    ...
    // If we're deleting something, the position field will become
    // invalid. Reset position to -1
    this.position = -1;
    ...
}
```

7. In the task list, locate the **TODO - Add methods to enable indexing the tree** task, and then double-click this task.

This task is located at the end of the **Tree** class, in the **Utility methods** code region.

8. Delete the **TODO - Add methods to enable indexing the tree** comment, and then add a method named **IndexTree**. This method should accept an integer parameter named *index*, and return an integer value. Add code to the method to perform the following actions:

    a. If the **LeftTree** property is not **null**, call the **IndexTree** method of the **LeftTree** property and assign the result to the *index* parameter. Pass the current value of the index variable to the **IndexTree** method.

    b. Update the local **position** member with the value of the *index* parameter.

    c. Increment the *index* parameter.

    d. If the **RightTree** property is not **null**, call the **IndexTree** method of the **RightTree** property and assign the result to the *index* parameter. Pass the current value of the index variable to the **IndexTree** method.

    e. At the end of the method, return the value of the *index* parameter.

    Your code should resemble the following code example.

```
public class Tree<TItem> : IList<TItem>, IBinaryTree<TItem> where
TItem : IComparable<TItem>
{
    ...
    #region Utility methods
    ...
    private int IndexTree(int index)
    {
        if (this.LeftTree != null)
        {
            index = this.LeftTree.IndexTree(index);
        }
        this.position = index;

        index++;

        if (this.RightTree != null)
        {
            index = this.RightTree.IndexTree(index);
        }
        return index;
    }
    #endregion
}
```

9. After the **IndexTree** method, add a private method named **GetItemAtIndex**. This method should accept an integer parameter named *index*, and return a **Tree<TItem>** object. In the method, add code to perform the following actions:

   a. If the value of the **position** member is **-1**, call the local **IndexTree** method. Pass **0** as the parameter to the **IndexTree** method.

   b. If the value of the **position** member is greater than the value of the index variable, call the **GetItemAtIndex** method of the **LeftTree** property and return the value that is generated . Pass the value of the *index* parameter to the **GetItemAtIndex** method.

   c. If the value of the **position** member is less than the value of the index variable, call the **GetItemAtIndex** method of the **RightTree** property and return the value that is generated. Pass the value of the *index* parameter to the **GetItemAtIndex** method.

   d. At the end of the method, return a reference to the current object.

   Your code should resemble the following code example.

```
private Tree<TItem> GetItemAtIndex(int index)
{
    // Add the index values if they're not already there
    if (this.position == -1)
    {
        this.IndexTree(0);
    }

    if (this.position > index)
    {
        return this.LeftTree.GetItemAtIndex(index);
    }
    if (this.position < index)
    {
        return this.RightTree.GetItemAtIndex(index);
    }
    return this;
}
```

10. After the **GetItemAtIndex** method, add a private method named **GetCount**. This method should accept an integer parameter named *accumulator*, and return an integer value. Add code to the method to perform the following actions:

a. If the **LeftTree** property is not **null**, call the **GetCount** method of the **LeftTree** property and store the result in the accumulator variable. Pass the current value of the accumulator variable to the **GetCount** method.

b. Increment the value in the accumulator variable.

c. If the **RightTree** property is not **null**, call the **GetCount** method of the **RightTree** property and store the result in the accumulator variable. Pass the current value of the accumulator variable to the **GetCount** method.

d. At the end of the method, return the value of the accumulator variable.

Your code should resemble the following code example.

```
private int GetCount(int accumulator)
{
    if (this.LeftTree != null)
    {
        accumulator = LeftTree.GetCount(accumulator);
    }
    accumulator++;
    if (this.RightTree != null)
    {
        accumulator = RightTree.GetCount(accumulator);
    }
    return accumulator;
}
```

Task 4: Implement the IList<T> interface methods and properties

1. Locate the **IndexOf** method. This method accepts a **TItem** object named **item**, and returns an integer value.

   This method should iterate through the tree and return a value that indicates the index of the **TItem** object in the tree. The method currently throws a **NotImplementedException** exception.

2. Replace the code in the **IndexOf** method with code to perform the following actions:

   a. If the *item* parameter is **null**, return the value **−1**.

   b. If the value of the **position** member is **−1**, call the **IndexTree** method and pass the value **0** as a parameter to the **IndexTree** method.

   c. Compare the value of the *item* parameter to the local **NodeData** property value:

i. If the value of the *item* parameter is less than the value in the **NodeData** property, and if the *LeftTree* parameter is **null**, return **-1**. Otherwise, return the result of a recursive call to the **LeftTree.IndexOf** method, passing the **item** value to the **IndexOf** method.

ii. If the value of the *item* parameter is greater than the value in the **NodeData** property, and if the *RightTree* parameter is **null**, return **-1**. Otherwise, return the result of a recursive call to the **RightTree.IndexOf** method, passing the **item** value to the **IndexOf** method.

Hint: Use the **CompareTo** method to compare the value in the *item* parameter and the value in the **NodeData** property.

d. At the end of the method, return the value of the local **position** member.

Your code should resemble the following example.

```
public int IndexOf(TItem item)
{
    if (item == null) return -1;

    // Add the index values if they're not already there
    if (this.position == -1)
        this.IndexTree(0);

    // Find the item - searching the tree for a matching Node.
    if (item.CompareTo(this.NodeData) < 0)
    {
        if (this.LeftTree == null)
        {
            return -1;
        }
        return this.LeftTree.IndexOf(item);
    }
    if (item.CompareTo(this.NodeData) > 0)
    {
        if (this.RightTree == null)
        {
            return -1;
        }
        return this.RightTree.IndexOf(item);
    }
    return this.position;
}
```

3. Locate the **this** indexer.

   The **this** indexer should return the **TItem** object at the index that the *index* parameter specifies. Currently, both **get** and **set** accessors throw a **NotImplementedException** exception.

4. Replace the code in the **get** accessor with code to perform the following actions:

   a. If the value of the *index* parameter is less than zero, or greater than the value of the **Count** property, throw an **ArgumentOutOfRangeException** exception with the following parameters:

      i. A string value, "index".

      ii. The *index* parameter value.

      iii. A string value, "Indexer out of range".

   b. At the end of the **get** accessor, call the **GetItemAtIndex** method. Pass the value of the index variable to the **GetItemAtIndex** method. Return the value of the **NodeData** property from the item that is retrieved by calling the **GetItemAtIndex** method.

   Your code should resemble the following code example.

```
public TItem this[int index]
{
    get
    {
        if (index < 0 || index >= Count)
        {
            throw new ArgumentOutOfRangeException
                ("index", index, "Indexer out of range");
        }

        return GetItemAtIndex(index).NodeData;
    }
    set
    {
        throw new NotImplementedException();
    }
}
```

5. Locate the **Clear** method. This method accepts no parameters, and does not return a value.

   This method should clear the contents of the tree and return it to a default state. Currently, the method throws a **NotImplementedException** exception.

6. Replace the code in the **Clear** method with code to perform the following actions:

   a. Set the **LeftTree** property to **null**.

   b. Set the **RightTree** property to **null**.

   c. Set the **NodeData** property to the default value for a **TItem** object.

   Your code should resemble the following code example:

```
public void Clear()
{
    LeftTree = null;
    RightTree = null;
    NodeData = default(TItem);
}
```

7. Locate the **Contains** method. This method accepts a *TItem* parameter, *item*, and returns a Boolean value.

   This method should iterate through the tree and return a Boolean value that indicates whether a node that matches the value of the *item* parameter exists in the tree. Currently, the method throws a **NotImplementedException** exception.

8. Replace the code in the **Contains** method with code to perform the following actions:

   a. If the value of the **NodeData** property is the same as the value of the *item* parameter, return **true**.

   b. If the value of the **NodeData** property is greater than the value of the *item* parameter, and if the **LeftTree** property is not **null**, return the result of a recursive call to the **LeftTree**.**Contains** method, passing the *item* parameter to the **Contains** method.

   c. If the value of the **NodeData** property is less than the value of the *item* parameter, and if the **RightTree** property is not **null**, return the result of a recursive call to the **RightTree**.**Contains** method, passing the *item* parameter to the **Contains** method.

   d. At the end of the method, return **false**.

   Your code should resemble the following code example.

```
public bool Contains(TItem item)
{
    if (NodeData.CompareTo(item) == 0)
```

```
    {
        return true;
    }

    if (NodeData.CompareTo(item) > 0)
    {
        if (this.LeftTree != null)
            return this.LeftTree.Contains(item);
    }
    else
    {
        if (this.RightTree != null)
            return this.RightTree.Contains(item);
    }

    return false;
}
```

9. Locate the **Count** property.

   This property is read-only, and should return an integer that represents the total number of items in the tree. Currently, the **get** accessor throws a **NotImplementedException** exception.

10. Replace the code in the **get** accessor with code to invoke the **GetCount** method, by passing **0** to the method call. Return the value that the **GetCount** method calculates.

    Your code should resemble the following code example.

```
public int Count
{
    get
    {
        return this.GetCount(0);
    }
}
```

11. Locate the **IsReadOnly** property.

    This property should return a Boolean value that signifies whether the tree is read-only.

12. Replace the code in the **get** accessor with a statement that returns the Boolean value **false**.

    Your code should resemble the following code example.

```
public bool IsReadOnly
{
    get
    {
        return false;
    }
}
```

13. Locate the **ICollection<TItem>.Remove** method. This method accepts a *TItem* parameter named *item*, and returns a Boolean value.

    This method should check whether a node with a value that matches the *item* parameter exists in the tree, and if so, remove the item from the tree. If an item is removed, the method should return **true**; otherwise, the method should return **false**.

Note: This version of the **Remove** method is fully qualified with the name of the interface. This is to disambiguate it from the local **Remove** method that is defined elsewhere in the **Tree** class.

14. In the **ICollection<TItem>.Remove** method, replace the existing code with statements that perform the following actions:

    a. If the tree contains a node that matches the value in the *item* parameter, call the local **Remove** method, and then return **true**.

    b. At the end of the method, return **false**.

    Your code should resemble the following code example.

```
bool ICollection<TItem>.Remove(TItem item)
{
    if (this.Contains(item) == true)
    {
        this.Remove(item);
        return true;
    }
    return false;
}
```

15. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**.

Task 5: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file and examine the **Main** method.

   The BinaryTreeTestHarness project contains code that you will use to test the completed **BinaryTree** class. You will continue to extend the **BinaryTree** class in the following exercises, so the **BinaryTree** class is not currently complete. For this reason, this exercise does not use some methods in the test harness.

   The **Main** method contains method calls to each of the test methods that you are about to examine:

   - In Solution Explorer, in the BinaryTreeTestHarness project, double-click **Program.cs**.

2. Examine the **TestIntegerTree** method.

   The **TestIntegerTree** method tests the **Remove** and **Contains** methods, and the indexer functionality of the **BinaryTree** class. First, the method invokes the **CreateATreeOfIntegers** method to build a sample tree that contains 10 values. Then, the method invokes the **WalkTree** method, which prints each node value to the console in numerical order.

Note: The **CreateATreeOfIntegers** method creates a **Tree** object that contains the values 10, 5, 11, 5, –12, 15, 0, 14, –8, and 10 in the order that the method adds them.

   The method then invokes the **Count** method and prints the result to the console. The method casts the tree to an **ICollection** object, and then calls the **Remove** method to remove the value **11** from the tree. The method again prints the result of the **Count** method to the console to prove that an item has been removed.

Note: The **BinaryTree** method contains two **Remove** methods, and in this case, the test method should invoke the interface-defined **ICollection.Remove** method. To enable the test method to do this, it must cast the **Tree** object to an **ICollection** object.

   The method then tests the **Contains** method by invoking the **Contains** method with the value **11** (which has just been removed) and then −12 (which is known to exist in the list).

Finally, the method tests the tree indexer by first retrieving the index of the value **5** in the tree and printing the index to the console, and then using the same index to retrieve the value **5** from that position in the tree.

3. Examine the **TestDeleteRootNodeInteger** method.

   The **TestDeleteRootNodeInteger** method tests the functionality of the **Remove** method when it attempts to remove the tree root node. When the root node value is removed from the tree, the next available node should be copied into its place to enable the tree to continue to function.

   In this test, the root node has the value **10**. There is a second node with the value **10**, so the **Remove** method must be invoked twice to remove both values.

   The method first invokes the **CreateATreeOfIntegers** method to build a sample tree, and then prints the tree to the console by invoking the **WalkTree** method. The method then casts the **Tree** object to an **ICollection** object, and then invokes the **Remove** method twice to remove both values of **10**. Finally, the method again invokes the **WalkTree** method to verify that the tree still functions correctly.

4. Examine the **TestStringTree** method.

   This method uses similar logic to the **TestIntegerTree** method to test the **Count**, **Remove**, **Contains**, and indexer method functionality. This method uses a **BinaryTree** object that contains the string values "k203", "h624", "p936", "h624", "a279", "z837", "e762", "r483", "d776", and "k203". In this test, the **Remove** method is tested by using the "p936" string value, and the indexer is tested by using the "h624" string value.

5. Examine the **TestDeleteRootNodeString** method.

   This method uses similar logic to the **TestDeleteRootNodeInteger** method to test the **Remove** method functionality, using the same string-based tree as the **TestStringTree** method. In this test, the "k203" string value is removed twice to test root node removal.

6. Examine the **TestTestResultTree** method.

   This method uses similar logic to the **TestIntegerTree** and **TestStringTree** methods to test the **Count**, **Remove**, **Contains**, and indexer method functionality, but it uses a **BinaryTree** object based on the **TestResult** type.

Note: The **TestResult** class implements the **IComparable** interface, and uses the **Deflection** property to compare instances of the **TestResult** object. Therefore, items in this tree are indexed by their **Deflection** property value.

In this case, the **Remove** method is tested with the **TestResult** object that has a **Deflection** value of **226**. The indexer is tested with the **TestResult** object that has a **Deflection** value of **114**.

7. Examine the **TestDeleteRootNodeTestResult** method.

   This method uses similar logic to the **TestDeleteRootNodeInteger and TestDeleteRootNodeString** methods to test the **Remove** method functionality, using the same **TestResult**-based tree as the **TestTestResultTree** method. In this test, the **TestResult** object that has a **Deflection** value of **190** is removed twice to test root node removal.

8. Run the BinaryTreeTestHarness application:

   - On the **Debug** menu, click **Start Without Debugging**.

9. Verify that the output in the console window resembles the following code example.

```
TestIntegerTree()
WalkTree()
-12
-8
0
5
5
10
10
11
14
15

Count: 10
Remove(11)

Count: 9

Contains(11): False

Contains(-12): True
IndexOf(5): 3

tree[3]: 5
```

Note that:

a. The console shows the output of the **TestIntegerTree** method.

b. The tree is displayed in numerical order by the **WalkTree** method.

c. Initially, the list contains 10 items, and then after the **Remove** method is called, the tree contains nine items.

d. The **Remove** method removes the value **11**, so the result of the **Contains** method is **false**. Note also that the **Contains** method verifies the presence of the value **-12**.

e. The **IndexOf** method reports that the value **5** is in position 3 in the list. This is confirmed by retrieving the value in position 3, which is shown to be **5**.

10. Press ENTER, and then verify that the output in the console window resembles the following code example.

```
TestDeleteRootNodeInteger()
Before
-12
-8
0
5
5
10
10
11
14
15

Remove 10 twice

After
-12
-8
0
5
5
11
14
15
```

Note that the tree shows two instances of the value **10** in the first list. Then, after those values are removed, the list no longer contains them. Also note that, after removing the root node value, the tree retains the remaining values and continues to function as expected.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Count: 10

Remove("p936")

Count: 9

Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This is the same test as the one you performed in step 9, but it is performed by using string data. Items in the list are displayed in alphabetical order.

12. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
```

```
z837

Remove k203 twice

After
a279
d776
e762
h624
h624
p936
r483
z837
```

13. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Count: 10

Remove(def266)

Count: 9

Contains(def266): False

Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This test is the same as the one you performed in steps 9 and 11, but this test is based on **TestResult** objects. Items are displayed in numerical order based on the value of the **Deflection** property.

14. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

15. Press ENTER twice to return to Visual Studio.

Exercise 2: Implementing an Enumerator by Writing Code

Task 1: Open the CustomCollections solution

- Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex2\Starter folder:

Note: The CustomCollections solution in the Ex2 folder is functionally the same as the code that you completed in Exercise 1. However, it includes an updated task list and an updated test harness to enable you to complete this exercise.

a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 13\Ex2 \Starter** folder, click **CustomCollections.sln**, and then click **Open**.

Task 2: Create the TreeEnumerator class

- In the BinaryTree project, add a new class named **TreeEnumerator**. This class should implement the **IEnumerator** interface, and should take a type parameter, **TItem**, where the **TItem** type implements the **IComparable** interface:

  a. In Solution Explorer, right-click the **BinaryTree** project, point to **Add**, and then click **Class**.

  b. In the **Add New Item - BinaryTree** dialog box, in the **Name** box, type **TreeEnumerator** and then click **Add**.

  c. In the TreeEnumerator.cs file, modify the **TreeEnumerator** class definition to make the class generic, based on a type parameter called **TItem**. Specify that the class implements the **IEnumerator** generic interface, and that the **TItem** type parameter implements the **IComparable** generic interface.

  Your code should resemble the following code example.

```
class TreeEnumerator<TItem> : IEnumerator<TItem>
    where TItem : IComparable<TItem>
{

}
```

Task 3: Add class-level variables and a constructor

1. In the **TreeEnumerator** class, add the following members:

   a. A **Tree<TItem>** object named **currentData**, initialized to a **null** value.

   This member will store the initial **Tree** object data that is passed to the class when it is constructed, and will be used to populate the internal queue with data. The data is also stored to enable the internal queue to reset.

   b. A **TItem** object named **currentItem**, initialized to a default **TItem** object.

   This member will store the last item that is removed from the queue.

c. A private **Queue<TItem>** object named **enumData**, initialized to a **null** value.

   This member holds an internal queue of items that the enumerator will iterate over. You will populate this queue with the items in the **Tree** object.

Your code should resemble the following code example.

```
class TreeEnumerator<TItem> : IEnumerator<TItem>
    where TItem : IComparable<TItem>
{
    private Tree<TItem> currentData = null;

    private TItem currentItem = default(TItem);

    private Queue<TItem> enumData = null;
}
```

2. Add a constructor. The constructor should accept a *Tree<TItem>* parameter named *data*, and should initialize the **currentData** member with the value of this parameter.

   Your code should resemble the following code example.

```
class TreeEnumerator<TItem> : IEnumerator<TItem>
    where TItem : IComparable<TItem>
{
    ...
    public TreeEnumerator(Tree<TItem> data)
    {
        this.currentData = data;
    }
}
```

Task 4: Add a method to populate the queue

- Below the constructor, add a new private method named **Populate**. The method should accept a *Queue<TItem>* parameter named *enumQueue*, and a *Tree<TItem>* parameter named *tree*. It should not return a value. Add code to the method to perform the following actions:

  a. If the **LeftTree** property of the *tree* parameter is not **null**, recursively call the **Populate** method, passing the *enumQueue* parameter and the **tree**.**LeftTree** property as parameters to the method.

b. Add the **tree**.**NodeData** property value of the *tree* parameter to the **enumQueue** queue.

c. If the **RightTree** property of the *tree* parameter is not **null**, recursively call the **Populate** method, passing the *enumQueue* parameter and the **tree**.**RightTree** property as parameters to the method.

This code walks the tree and fills the queue with each item that is found, in order.

Your code should resemble the following code example.

```
private void Populate(Queue<TItem> enumQueue, Tree<TItem> tree)
{
    if (tree.LeftTree != null)
    {
        Populate(enumQueue, tree.LeftTree);
    }

    enumQueue.Enqueue(tree.NodeData);

    if (tree.RightTree != null)
    {
        Populate(enumQueue, tree.RightTree);
    }
}
```

Task 5: Implement the IEnumerator<T> and IEnumerator methods

1. In the class definition, right-click **IEnumerator**, point to **Implement Interface**, and then click **Implement Interface Explicitly**.

   Visual Studio will generate stubs for the methods and properties that the **IEnumerator<>**, **IEnumerator**, and **IDisposable** interfaces expose.

2. Locate the **Current** property.

   This property should return the last **TItem** object that was removed from the queue.

3. In the **get** accessor of the **Current** property, replace the existing code with code to perform the following actions:

   a. If the **enumData** member is **null**, throw a new **InvalidOperationException** exception with the message "Use MoveNext before calling Current".

   b. Return the value of the **currentItem** member.

Your code should resemble the following code example.

```
TItem IEnumerator<TItem>.Current
{
    get
    {
        if (this.enumData == null)
        {
            throw new InvalidOperationException("Use MoveNext before
calling Current");
        }
        return this.currentItem;
    }
}
```

4. Locate the **MoveNext** method. The method accepts no parameters and returns a Boolean value.

   The **MoveNext** method should ensure that the internal queue is initialized, retrieve the next item from the internal queue, and then store it in the **currentItem** property. If the operation succeeds, the method returns **true**, otherwise, it returns **false**.

5. In the **MoveNext** method, replace the existing code with code to perform the following actions:

   a. If the **enumData** object is **null**, create a new queue object, and then invoke the **Populate** method, passing the new **queue** object and the **currentData** member as parameters to the method call.

   b. If the **enumData** object contains any values, retrieve the first item in the queue, store it in the **currentItem** member, and then return the Boolean value **true**.

   c. At the end of the method, return the Boolean value **false**.

   Your code should resemble the following code example.

```
bool System.Collections.IEnumerator.MoveNext()
{
    if (this.enumData == null)
    {
        this.enumData = new Queue<TItem>();
        Populate(this.enumData, this.currentData);
    }

    if (this.enumData.Count > 0)
    {
        this.currentItem = this.enumData.Dequeue();
```

```
        return true;
    }
    return false;
}
```

6. Locate the **Reset** method. This method accepts no parameters, and does not return a value.

   This method should reset the enumerator to its initial state. You do this by repopulating the internal queue with the data from the **Tree** object.

7. In the **Reset** method, replace the existing code with code that invokes the **Populate** method, passing the **enumData** and **currentData** members as parameters to the method.

   Your code should resemble the following code example.

```
void System.Collections.IEnumerator.Reset()
{
    Populate(this.enumData, this.currentData);
}
```

8. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 6: Implement the IDisposable interface

1. In the **TreeEnumerator** class, locate the **Dispose** method. This method accepts no parameters and does not return a value.

   The method should dispose of the class, relinquishing any resources that may not be reclaimed if they are not disposed of explicitly, such as file streams and database connections.

Note: The **Queue** object does not implement the **IDisposable** interface, so you will use the **Dispose** method of the **TreeEnumerator** class to clear the queue of any data.

2. In the **Dispose** method, replace the existing code with code that clears the **enumQueue** queue object.

Hint: Use the **Clear** method of the **Queue** class to empty a **Queue** object.

Your code should resemble the following code example.

```
void IDisposable.Dispose()
{
    this.enumData.Clear();
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 7: Modify the Tree class to return a TreeEnumerator object

1. In the task list, locate the **TODO - Update the Tree class to return the TreeEnumerator class** task, and then double-click this task.

   This task is located in the **Tree** class.

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

   c. Double-click the **TODO - Update the Tree class to return the TreeEnumerator class** task.

2. Remove the comment. In the **GetEnumerator** method, replace the existing code with code that creates and initializes a new **TreeEnumerator** object. Specify the **TItem** type as the type parameter, and pass the current object as the parameter to the **TreeEnumerator** constructor. Return the **TreeEnumerator** object that is created.

   Your code should resemble the following code example.

```
public IEnumerator<TItem> GetEnumerator()
{
    return new TreeEnumerator<TItem>(this);
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 8: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file.

This version of the BinaryTreeTestHarness project contains the same code and performs the same tests as in Exercise 1. However, it has been updated to test the enumerator functionality that you just added:

- In Solution Explorer, in the BinaryTreeTestHarness project, double-click **Program.cs**.

2. Examine the **TestIteratorsIntegers** method.

   This method tests the iterator functionality that you just implemented, by using the same integer tree as in Exercise 1. The method builds the tree by invoking the **CreateATreeOfIntegers** method, and then uses a **foreach** statement to iterate through the list and print each value to the console. The method then attempts to iterate through the tree in reverse order, and print each item to the console.

Note: You will add the functionality to enable reverse iteration of the tree in the next exercise. It is expected that attempting to reverse the tree will throw a **NotImplementedException** exception. The **TestIteratorsIntegers** method will catch this exception when it occurs, and print a message to the console.

3. Examine the **TestIteratorsStrings** method.

   This method uses similar logic to the **TestIteratorsIntegers** method to test the iterator functionality of the **BinaryTree** object, but it uses the same string-based tree as the one you used in Exercise 1. The method uses the **CreateATreeOfStrings** method to build the tree, iterates through the tree, and then prints all items to the console. This method also attempts to display the data in the tree in reverse order, and will encounter a **NotImplementedException** exception (you will implement this feature in the next exercise).

4. Examine the **TestIteratorsTestResults** method.

   This method uses similar logic to the **TestIteratorsIntegers** and **TestIteratorsStrings** methods to test the iterator functionality of the **BinaryTree** object. It uses a **TestResult**-based tree by invoking the **CreateATreeOfTestResults** method as in Exercise 1.

5. Run the BinaryTreeTestHarness application:

   - On the **Debug** menu, click **Start Without Debugging**.

6. Verify that the output in the console window matches the following code example.

```
TestIntegerTree()

WalkTree()
-12
-8
0
5
5
10
10
11
14
15

Count: 10

Remove(11)

Count: 9

Contains(11): False

Contains(-12): True

IndexOf(5): 3

tree[3]: 5
```

This output matches the **TestIntegerTree** method output from Exercise 1, and confirms that you have not compromised existing functionality by adding the iterator functionality.

7. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeInteger()

Before
-12
-8
0
5
5
10
10
11
14
15
```

```
Remove 10 twice

After
-12
-8
0
5
5
11
14
15
```

This output matches the **TestDeleteRootNodeInteger** method output from Exercise 1, and again confirms that existing functionality works as expected.

8. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsIntegers()

In ascending order
-12
-8
0
5
5
10
10
11
14
15

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that the items in the list are displayed in numerical order, and note that the **Reverse** method displays a message that indicates that the **Reverse** functionality is not yet implemented.

9. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
```

```
h624
h624
k203
k203
p936
r483
z837

Count: 10

Remove("p936")

Count: 9

Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This output matches the **TestStringTree** method output from Exercise 1.

10. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Remove k203 twice

After
a279
d776
e762
h624
h624
```

```
p936
r483
z837
```

This output matches the **TestDeleteRootNodeString** method output from Exercise 1.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsStrings()

In ascending order
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that this represents the same test as you performed in step 8. It uses string data to verify the iterator functionality, and all items are displayed in alphabetical order.

12. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

```
Count: 10

Remove(def266)

Count: 9

Contains(def266): False

Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This output matches the **TestTestResultTree** method output from Exercise 1.

13. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

This output matches the **TestDeleteRootNodeTestResults** method output from Exercise 1.

14. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsTestResults()

In ascending order
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that this represents the same test as you performed in steps 8 and 11. It uses **TestResult** object data to verify the iterator functionality, and all items are displayed in numerical order based on the value of the **Deflection** property.

15. Press ENTER twice to return to Visual Studio.

Exercise 3: Implementing an Enumerator by Using an Iterator

Task 1: Open the CustomCollections solution

- Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex3\Starter folder:

Note: The CustomCollections solution in the Ex3 folder is functionally the same as the code that you completed in Exercise 2. However, it includes an updated task list and an updated test harness to enable you to complete this exercise.

a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the Open Project dialog box, move to the E:\Labfiles\Lab 13\Ex3 \Starter folder, click CustomCollections.sln, and then click Open.

Task 2: Add an enumerator to return an enumerator that iterates through data in reverse order

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. In the task list, locate the **TODO - Add a method to return the list in reverse order** task, and then double-click this task.

   This task is located at the end of the **Tree** class.
3. Remove the task comment, and then add a new public method named **Reverse**. The method should accept no parameters, and return an **IEnumerable** collection based on the **TItem** type parameter.

   Your code should resemble the following code example.

```
public IEnumerable<TItem> Reverse()
{

}
```

4. Add code to the method to perform the following actions:
   a. If the **RightTree** property is not **null**, iterate through the items that are returned by calling the **Reverse** method of the **RightTree** property, and then yield each item that is found.

Hint: The **yield** statement is used in an **iterator** block to return a value to the enumerator object, or to signal the end of an iteration.

   b. Yield the value in the **NodeData** property of the current item.
   c. If the **LeftTree** property is not **null**, iterate through the items that are returned by calling the **Reverse** method of the **LeftTree** property, and then yield each item that is found.

   Your code should resemble the following code example.

```
public IEnumerable<TItem> Reverse()
{
    if (this.RightTree != null)
    {
```

```
        foreach (TItem item in this.RightTree.Reverse())
        {
            yield return item;
        }
    }

    yield return this.NodeData;

    if (this.LeftTree != null)
    {
        foreach (TItem item in this.LeftTree.Reverse())
        {
            yield return item;
        }
    }

}
```

5. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 3: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file.

   This version of the BinaryTreeTestHarness project contains the same code and performs the same tests as in Exercise 2. Now that you have implemented the **Reverse** method in the **BinaryTree** object, the test application should not encounter the **NotImplementedException** exception in the **TestIteratorsIntegers**, **TestIteratorsStrings**, and **TestIteratorsTestResults** methods.

2. Run the BinaryTreeTestHarness application:
   - On the **Debug** menu, click **Start Without Debugging**.

3. Verify that the output in the console window matches the following code example.

```
TestIntegerTree()

WalkTree()
-12
-8
0
5
```

```
5
10
10
11
14
15

Count: 10

Remove(11)

Count: 9

Contains(11): False

Contains(-12): True

IndexOf(5): 3

tree[3]: 5
```

This output matches the **TestIntegerTree** method output from Exercises 1 and 2, and confirms that you have not compromised existing functionality by adding the reverse iterator functionality.

4. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeInteger()

Before
-12
-8
0
5
5
10
10
11
14
15

Remove 10 twice

After
-12
-8
0
5
```

```
5
11
14
15
```

This output matches the **TestDeleteRootNodeInteger** method output from Exercises 1 and 2, and again confirms that the existing functionality works as expected.

5. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsIntegers()

In ascending order
-12
-8
0
5
5
10
10
11
14
15

In descending order
15
14
11
10
10
5
5
0
-8
-12
```

This output is similar to the **TestIteratorsIntegers** method in Exercise 2, but the **Reverse** method is now implemented, so the tree is also displayed in descending numerical order.

6. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Count: 10

Remove("p936")

Count: 9

Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This output matches the **TestStringTree** method output from Exercises 1 and 2.

7. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Remove k203 twice
```

```
After
a279
d776
e762
h624
h624
p936
r483
z837
```

This output matches the **TestDeleteRootNodeString** method output from Exercises 1 and 2.

8. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsStrings()

In ascending order
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

In descending order
z837
r483
p936
k203
k203
h624
h624
e762
d776
a279
```

This test uses string data to verify the iterator functionality, and all items are displayed in alphabetical order, and then reverse alphabetical order.

9. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Count: 10

Remove(def266)

Count: 9

Contains(def266): False

Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This output matches the **TestTestResultTree** method output from Exercises 1 and 2.

10. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

```
Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

This output matches the **TestDeleteRootNodeTestResults** method output from Exercises 1 and 2.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsTestResults()

In ascending order
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

In descending order
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
```

This test uses **TestResult** object data to verify iterator functionality. Therefore, all items are displayed in numerical order based on the value of the **Deflection** property, and then the list is reversed to display data in descending numerical order based on the value of the **Deflection** property.

12. Press ENTER twice to return to Visual Studio.