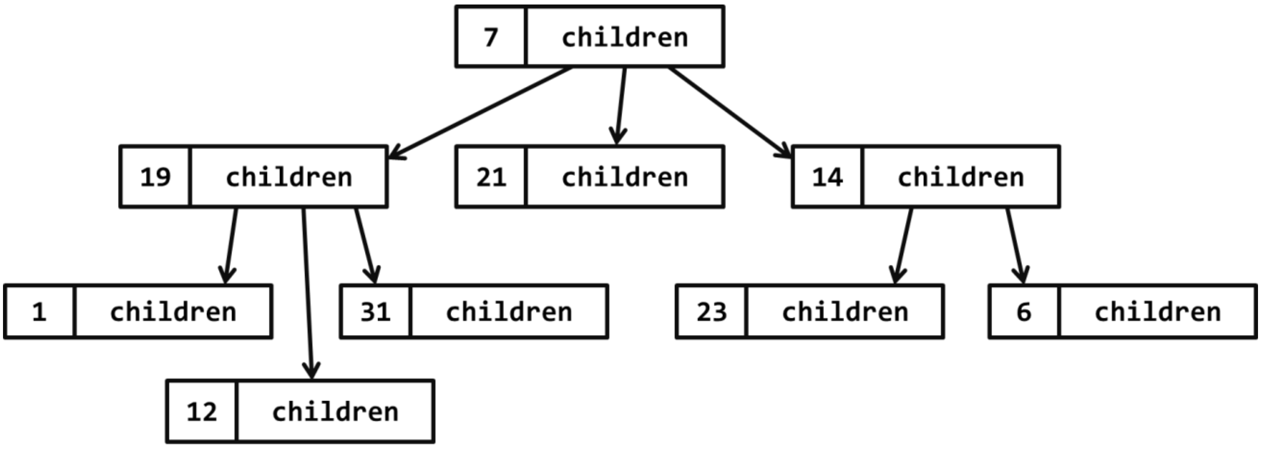
Exercises: Implement Trees and Traversals

This document defines the **in-class exercises** assignments for the ["Data Structures" course @ Software University](https://softuni.bg).

Part I – Implement a Tree

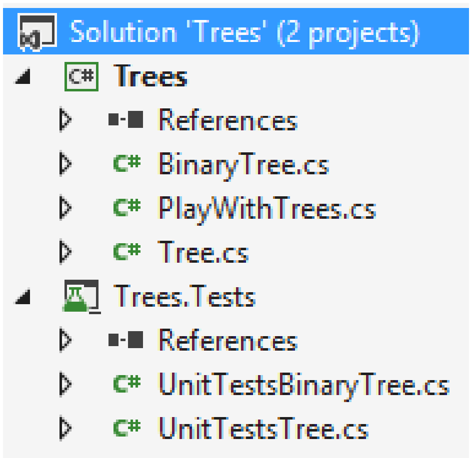
The first part of this lab aims to **implement a tree** (a node that holds a value and multiple child nodes) and **traverse it** recursively.



In the figure above, the **tree nodes** hold a **value** and a **list of descendent tree nodes**. It is recursive data structure.

Problem 1. Trees – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished **`Tree<T>`** and **`BnaryTree<T>`** classes and **unit tests** for their functionality. The project holds the following assets:



The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**.

The unfinished **`Tree<T>`** class stays in the file **`Tree.cs`**:

`Tree.cs`

`public class Tree<T>`
`{`
`    public Tree(T value, params Tree<T>[] children) { … }`
`    public void Print(int indent = 0) { … }`

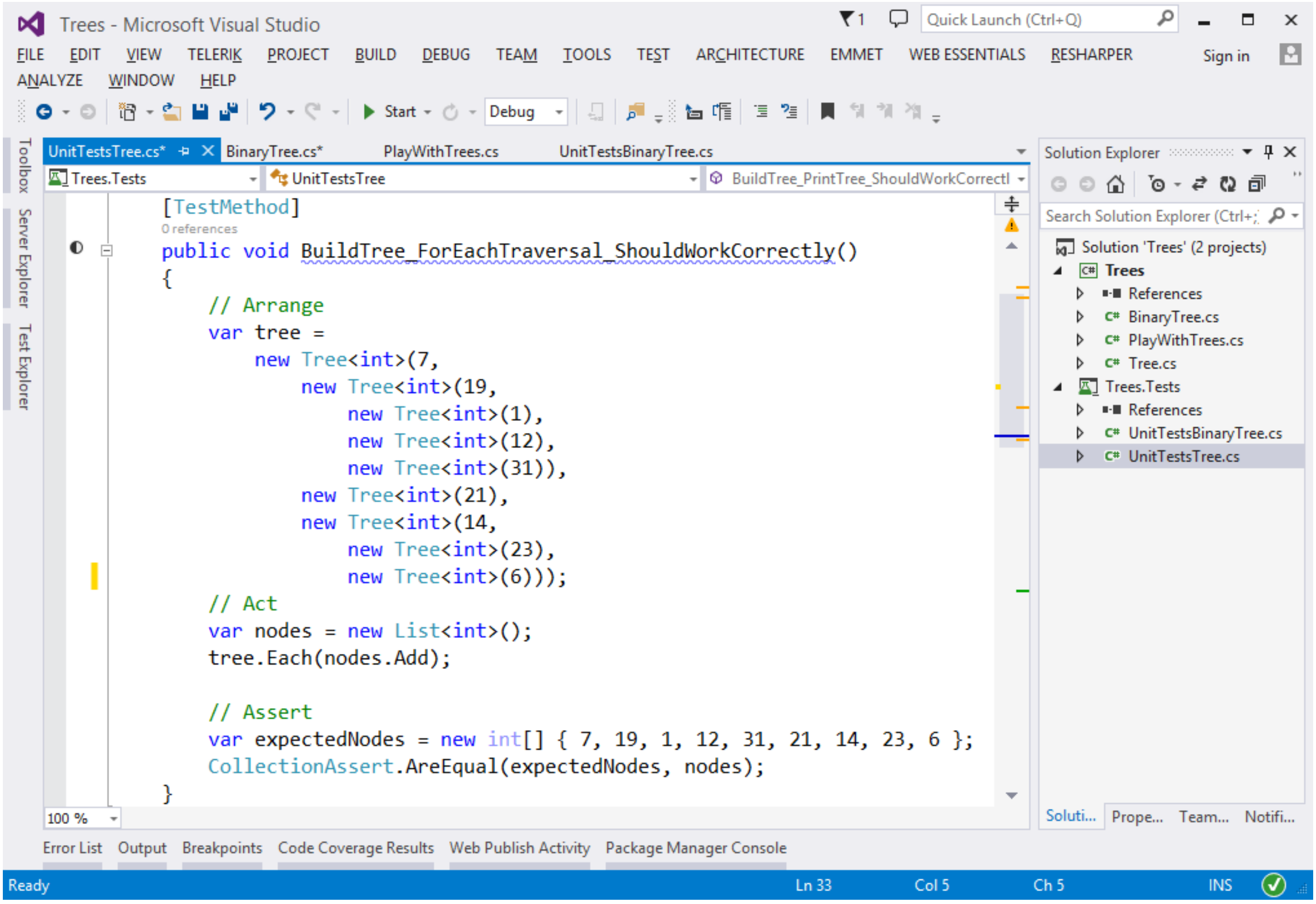
```
    public void Each(Action<T> action) { … }
}
```

The unfinished **BinaryTree<T>** class stays in the file **BinaryTree.cs**:

BinaryTree.cs

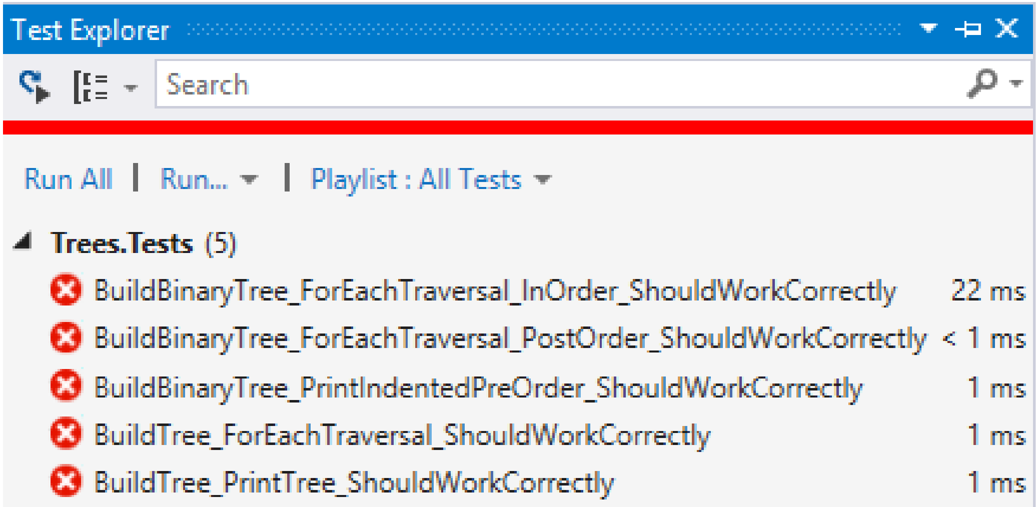
```
public class BinaryTree<T>
{
    public BinaryTree(T value, BinaryTree<T> leftChild = null,
        BinaryTree<T> rightChild = null) { … }
    public void PrintIndentedPreOrder(int indent = 0)  { … }
    public void EachInOrder(Action<T> action) { … }
    public void EachPostOrder(Action<T> action) { … }
}
```

The project comes with **unit tests** covering the entire functionality of the trees (see the files **UnitTestsTree.cs** and **UnitTestsBinaryTree.cs**):



Problem 2. Run the Unit Tests to Ensure All of Them Initially Fail

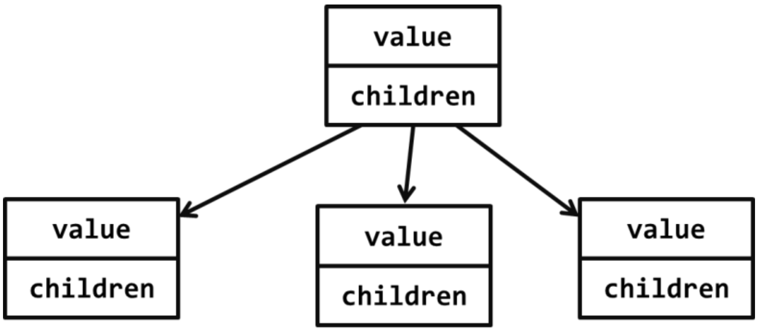
Run the unit tests from the **Trees.Tests** project. Open the "**Test Explorer**" window (Menu → Test → Windows → Test Explorer) and run all tests. The expected behavior is that all tests should fail:



This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

Problem 3. Define the Tree<T> Data Structure

The first step is to define the inner **data** hold **tree nodes**. It should hold the node **value** and a list of **child nodes**:



The source code might look like this:

```
public class Tree<T>
{
    0 references
    public T Value { get; set; }
    0 references
    public IList<Tree<T>> Children { get; private set; }
```

Problem 4. Define the Tree<T> Constructor

The next step is to define a **constructor** for the **`Tree<T>`** class to ensure you can create:

- **Leaf tree nodes** (holding a specified value) without child nodes, e.g.
  ```
  var node = new Tree<int>(5);
  ```
- **Internal tree nodes** (holding a specified value) with child nodes, e.g.
  ```
  var node = new Tree<int>(5,
      new Tree<int>(6),
      new Tree<int>(7));
  ```

You can use **optional parameters** for the child nodes to combine the above two constructors. A sample source code is shown below:

```
public Tree(T value, params Tree<T>[] children)
{
    this.Value = value;
    this.Children = new List<Tree<T>>();
    // Append the child nodes (children) to this.Children
    foreach (var child in children)
    {
        this.Children.Add(child);
    }
}
```

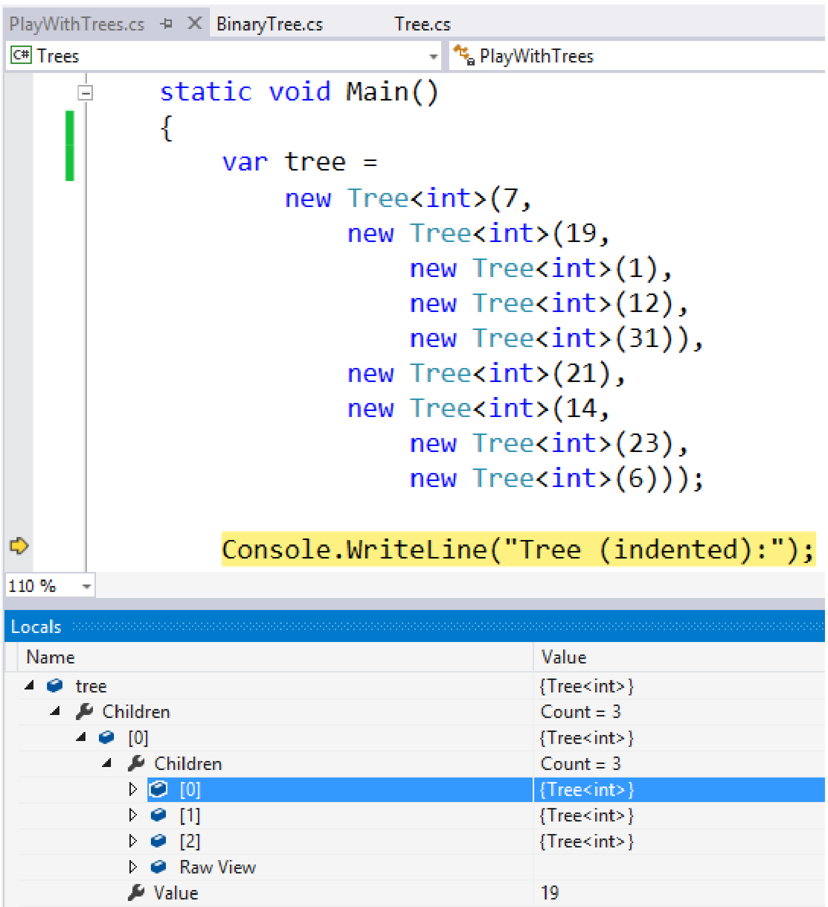
The parameter **`children`** is optional, so it could be passed or skipped. The goal is to allow creating trees by invoking **nested constructors** like these:

```
var tree =
    new Tree<int>(7,
        new Tree<int>(19,
            new Tree<int>(1),
            new Tree<int>(12),
        new Tree<int>(21)));
```

Problem 5. Test the Tree<T> Constructor

Now, test whether the **`Tree<T>`** and its constructor work as expected.

1. Use the debugger to set a breakpoint in the file **`PlayWithTrees.cs`** just after the tree construction.
2. Use the **`[Locals]`** debug window to browse the tree structure and the **child nodes** for each tree node.



Problem 6. Define the Print() Method

Now, we need to implement the **tree functionality**. First, implement the **`Print()`** method. It works recursively:

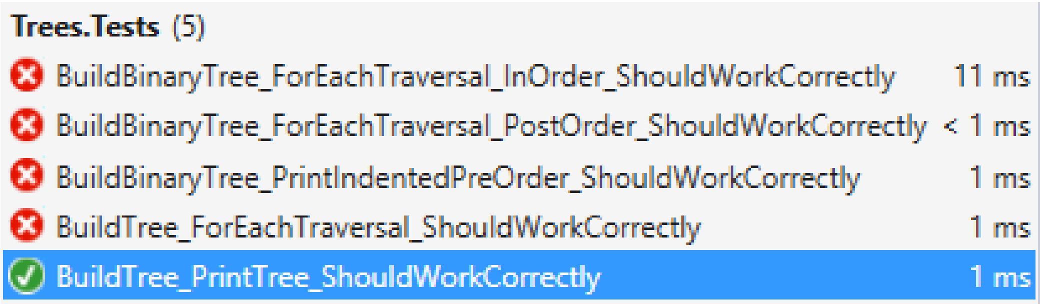
- Prints the current node **`value`** (indented a few spaces on the right).
- Calls the **`Print()`** method recursively to **print all child nodes** of the current node.

The code might look like this:

```
public void Print(int indent = 0)
{
    Console.Write(new string(' ', 2 * indent));
    Console.WriteLine(this.Value);
    foreach (var child in this.Children)
    {
        child.Print(indent + 1);
    }
}
```

Problem 7. Test the Print() Method

To test the **`Print()`** method, **run the unit tests**. Some of them should pass successfully:



Problem 8. Implement "For Each" Traversal

Now, implement the **`Each(Action<T>)`** method that **traverses the tree** recursively from its root to its leaves and invokes the provided **action function** for each visited tree node. It works as follows:

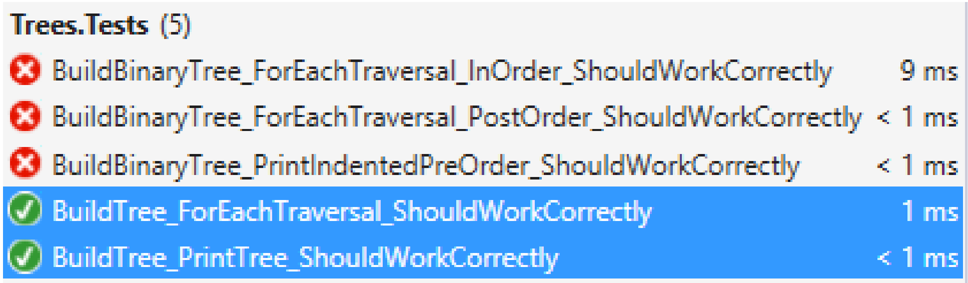
- Process the current node **`value`** (invokes the action function on it).
- Calls the **`Each()`** method recursively to **process all child nodes** of the current node.

The source code might look as follows:

```
public void Each(Action<T> action)
{
    action(this.Value);
    // For each child node invoke child.Each(action)
    foreach (var child in this.Children)
    {
        child.Each(action);
    }
}
```

Problem 9. Test the Each(Action<T>) Method

To test the **`Each(Action<T>)`** method, **run the unit tests**. All tests now should pass successfully:



The failed tests cover the **`BinaryTree<T>`** class, which is still not implemented.

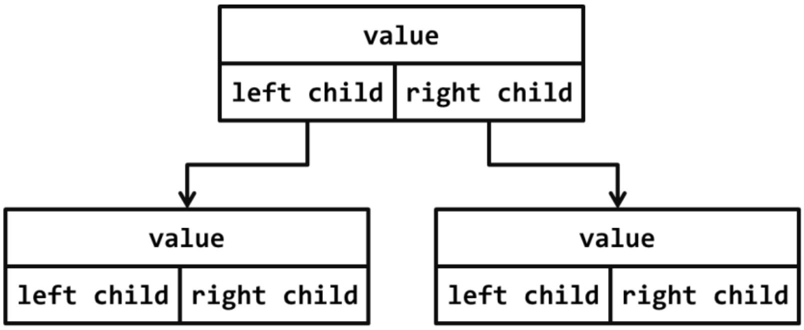
Congratulations! You have implemented your tree data structure.

Part II – Implement a Binary Tree

The second part of this lab aims to implement a **binary tree** (a node that holds a value + left and right child nodes).

Problem 10. Define the BinaryTree<T> Data Structure

The first step is to define the inner **data** hold **binary tree nodes**. It should hold the node **value** + **left** and **right child nodes** (both of them are optional and can be **`null`**):



The source code might look like this:

```
public class BinaryTree<T>
{
    0 references
    public T Value { get; set; }
    0 references
    public BinaryTree<T> LeftChild { get; set; }
    0 references
    public BinaryTree<T> RightChild { get; set; }
```

Problem 11. Define the BinaryTree<T> Constructor

The next step is to define a **constructor** for the **`BinaryTree<T>`** class to ensure you can create:

- **Leaf tree nodes** (holding a specified value) without child nodes, e.g.

  ```
  var node = new BinaryTree<string>("node");
  ```

- **Internal tree nodes** (holding a specified value) with left and right child nodes, e.g.

```
var binaryTree =
    new BinaryTree<string>("+",
        new BinaryTree<string>("3"),
        new BinaryTree<string>("2"));
```

You can use **optional parameters** (holding **null** by default) for the child nodes to combine the above two constructors. A sample source code is shown below:

```
public BinaryTree(T value,
    BinaryTree<T> leftChild = null,
    BinaryTree<T> rightChild = null)
{
    this.Value = value;
    this.LeftChild = leftChild;
    this.RightChild = rightChild;
}
```

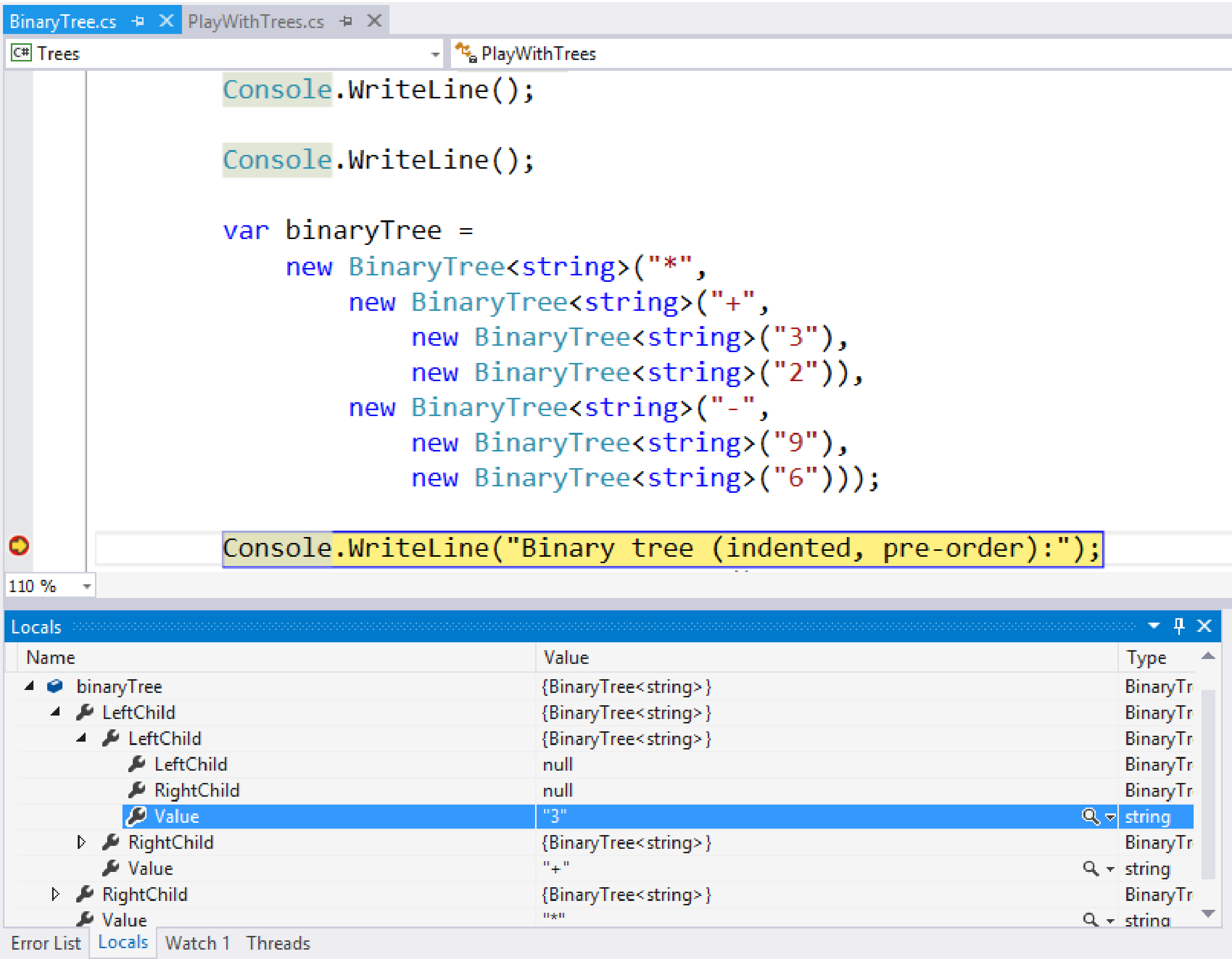
The parameters **leftChild** and **rightChild** are optional and can be passed or skipped. This will allow constructing binary tree like this:

```
var binaryTree =
    new BinaryTree<string>("*",
        new BinaryTree<string>("+",
            new BinaryTree<string>("3"),
            new BinaryTree<string>("2")),
        new BinaryTree<string>("-",
            new BinaryTree<string>("9"),
            new BinaryTree<string>("6")));
```

Problem 12. Test the BinaryTree<T> Constructor

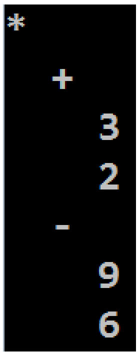
Now, test whether the **BinaryTree<T>** and its constructor work as expected.

3. Use the debugger to set a breakpoint in the file **PlayWithTrees.cs** just after the binary tree construction.
4. Use the **[Locals]** debug window to browse the binary tree structure and the **child nodes** for each tree node (left and right child).



Problem 13. Define the PrintIndentedPreOrder() Method

Now, we need to implement the **binary tree functionality**. First, implement the **`PrintIndentedPreOrder()`** method. It prints the tree in pre-order (root; left; right), indented visually like this:



The **`PrintIndentedPreOrder()`** method works recursively:

- Prints the current node **`value`** (indented a few spaces on the right).
- Calls the **`PrintIndentedPreOrder()`** method recursively to **print the left child** of the current node (when exists).
- Calls the **`PrintIndentedPreOrder()`** method recursively to **print the right child** of the current node (when exists).

The code might look like this:

```
public void PrintIndentedPreOrder(int indent = 0)
{
    // Pre-order == root node, left child, right child
    Console.Write(new string(' ', 2 * indent));
    Console.WriteLine(this.Value);
    if (this.LeftChild != null)
    {
        this.LeftChild.PrintIndentedPreOrder(indent + 1);
    }
    if (this.RightChild != null)
    {
        this.RightChild.PrintIndentedPreOrder(indent + 1);
    }
}
```

Problem 14. Test the PrintIndentedPreOrder() Method

To test the **`PrintIndentedPreOrder()`** method, **run the unit tests**. Some of them should pass successfully:

Trees.Tests (5)	
BuildBinaryTree_ForEachTraversal_InOrder_ShouldWorkCorrectly	12 ms
BuildBinaryTree_ForEachTraversal_PostOrder_ShouldWorkCorrectly	1 ms
BuildBinaryTree_PrintIndentedPreOrder_ShouldWorkCorrectly	1 ms
BuildTree_ForEachTraversal_ShouldWorkCorrectly	1 ms
BuildTree_PrintTree_ShouldWorkCorrectly	< 1 ms

Problem 15. Implement the EachInOrder(Action<T>) Method

Next, let's implement the **`EachInOrder(Action<T>)`** method that traverses the binary tree in **in-order** (left; root; right). It is again recursive, very similar to the previous method:

```
public void EachInOrder(Action<T> action)
{
    // In-order == left child, root node, right child
    // TODO: process the left child

    action(this.Value);

    // TODO: process the right child

}
```

Problem 16. Test the EachInOrder(Action<T>) Method

To test the **`EachInOrder(Action<T>)`** method, **run the unit tests**. One more test now should pass successfully:

Trees.Tests (5)	
✖ BuildBinaryTree_ForEachTraversal_PostOrder_ShouldWorkCorrectly	6 ms
✔ BuildBinaryTree_ForEachTraversal_InOrder_ShouldWorkCorrectly	51 ms
✔ BuildBinaryTree_PrintIndentedPreOrder_ShouldWorkCorrectly	< 1 ms
✔ BuildTree_ForEachTraversal_ShouldWorkCorrectly	1 ms
✔ BuildTree_PrintTree_ShouldWorkCorrectly	< 1 ms

Problem 17. Implement the EachPostOrder(Action<T>) Method

Next, let's implement the **`EachPostOrder(Action<T>)`** method that traverses the binary tree in **post-order** (left; right; root). It is again recursive, very, very similar to the previous method:

```
public void EachPostOrder(Action<T> action)
{
    // Post-order == left child, right child, root node
    if (this.LeftChild != null)
    {
        this.LeftChild.EachPostOrder(action);
    }
    if (this.RightChild != null)
    {
        this.RightChild.EachPostOrder(action);
    }
    action(this.Value);
}
```

Problem 18. Test the EachPostOrder(Action<T>) Method

To test the **`EachPostOrder(Action<T>)`** method, **run the unit tests**. All tests should now pass successfully:

Trees.Tests (5)	
✔ BuildBinaryTree_ForEachTraversal_InOrder_ShouldWorkCorrectly	6 ms
✔ BuildBinaryTree_ForEachTraversal_PostOrder_ShouldWorkCorrectly	< 1 ms
✔ BuildBinaryTree_PrintIndentedPreOrder_ShouldWorkCorrectly	< 1 ms
✔ BuildTree_ForEachTraversal_ShouldWorkCorrectly	1 ms
✔ BuildTree_PrintTree_ShouldWorkCorrectly	1 ms

Congratulations! You have implemented your binary tree data structure.