# Exercises: Traverse a Graph; Escape from Labyrinth

This document defines the **in-class exercises** assignments for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1147/Data-Structures-June-2015).

# Part I – Traverse a Graph to Find Its Connected Components

The first part of this lab aims to implement the **DFS algorithm** (Depth-First-Search) to **traverse a graph** and find its **connected components** (nodes connected to each other either directly, or through other nodes). The graph nodes are numbered from 0 to n-1. The graph comes from the console in the following format:

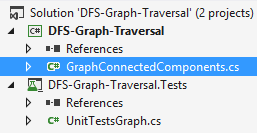
* First line: number of lines n
* Next n lines: list of child nodes for the nodes 0 … n-1 (separated by a space)

Print the connected components in the same format as in the examples below:

|  |  |  |
| --- | --- | --- |
| **Input** | **Graph** | **Output** |
| 9  3 6  3 4 5 6  8  0 1 5  1 6  1 3  0 1 4  2 |  | Connected component: 6 4 5 1 3 0  Connected component: 8 2  Connected component: 7 |
| 1  0 |  | Connected component: 0 |
| 0 | (empty graph) | Connected component: |
| 7  2 6  1  4  3  1 |  | Connected component: 0  Connected component: 2 6 1  Connected component: 4 3  Connected component: 5 |
| 4  1 2 3  0 1 2 3 3  0 1 3  0 1 1 2 |  | Connected component: 3 2 1 0 |

## Graph Traversal – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished class GraphConnectedComponents and **unit tests** for its functionality. The project holds the following assets:

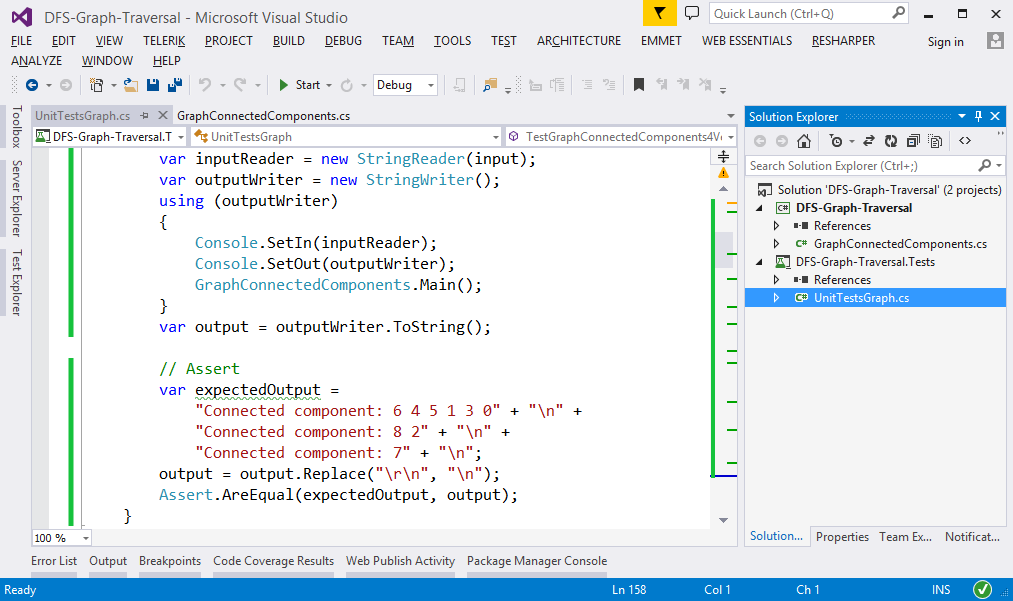


The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**.

The unfinished GraphConnectedComponents class stays in the file GraphConnectedComponents.cs:

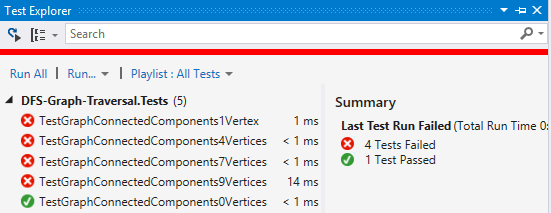
|  |
| --- |
| GraphConnectedComponents.cs |
| public class GraphConnectedComponents  {  public static void Main()  {  // **TODO: implement me**  }  } |

The project comes with **unit tests** covering the functionality of the GraphConnectedComponents class:



## Run the Unit Tests to Ensure They Initially Fail

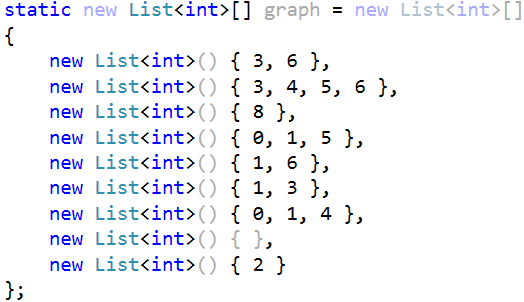
**Run the unit tests** from the DFS-Graph-Traversal.Tests project. Open the "**Test Explorer**" window (Menu 🡪 Test 🡪 Windows 🡪 Test Explorer) and run all tests. The expected behavior is that all tests should fail:



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

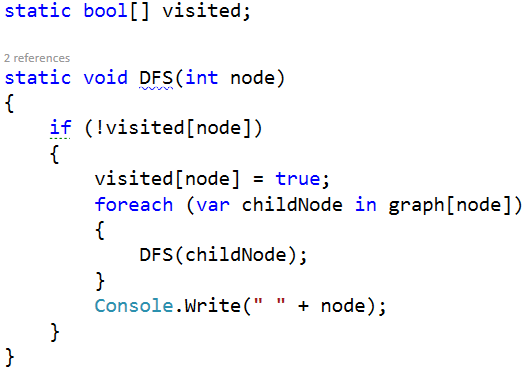
## Define a Sample Graph

The first step is to define a sample graph. It will be used to test the code during the development:



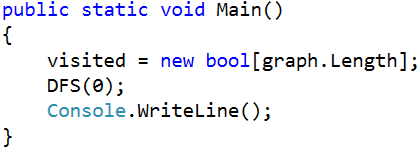
## Implement the DFS Algorithm

The next step is to implement the **DFS** (Depth-First-Search) algorithm to traverse recursively all connected nodes reachable from specified start node:

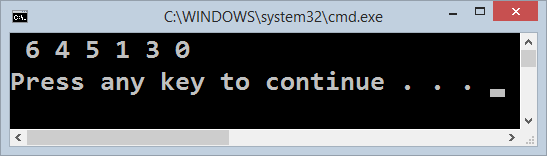


## Test the DFS Algorithm

Now, test whether the DFS algorithm implementation. Invoke it starting from node 0. It should print the connected component, holding the node 0:

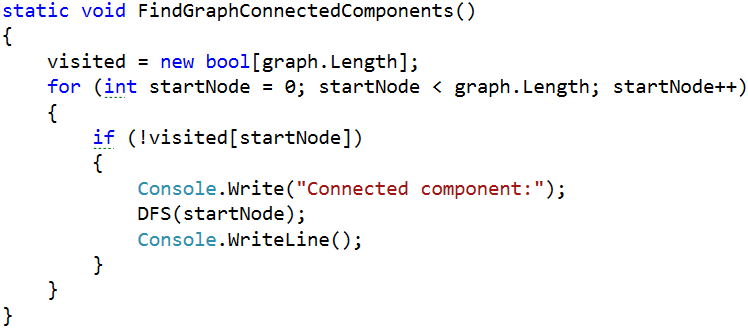


Now run the code above. It should find the first connected component in the graph, holding the node 0:

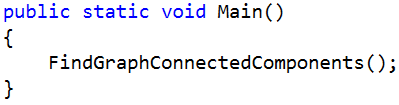


## Find All Connected Components

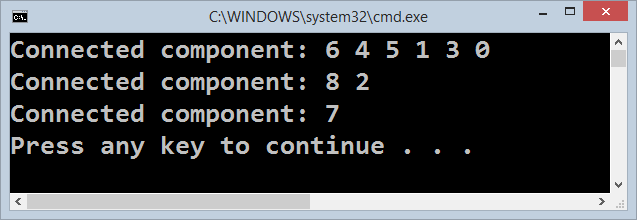
Now, we have DFS algorithm implemented, which finds the connected component holding all nodes reachable from given starting node. This is good, but we want to find all connected components. We can just run the DFS algorithm many times from each node (which was not visited already):



Now let's test the above code. Just call it from the main method:

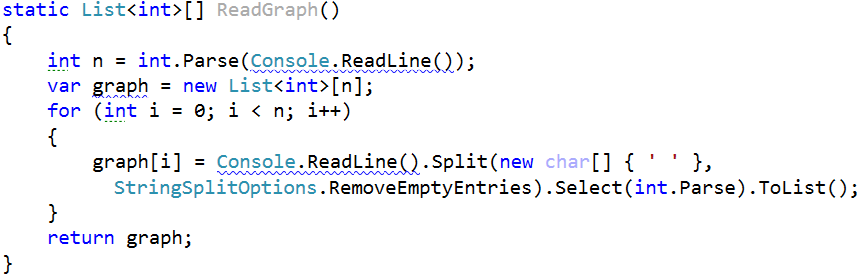


The output is as expected. It prints all connected components in the graph:

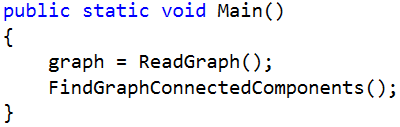


## Read the Input Data from the Console

Usually, when we solve problems, we work on hard-coded sample data (in our case the graph is hard-coded) and we write the code step by step, test it continuously and finally, when the code is ready and it works well, we change the hard-coded input data with a logic that reads it. Let's implement the data entry logic (read graph from the console):



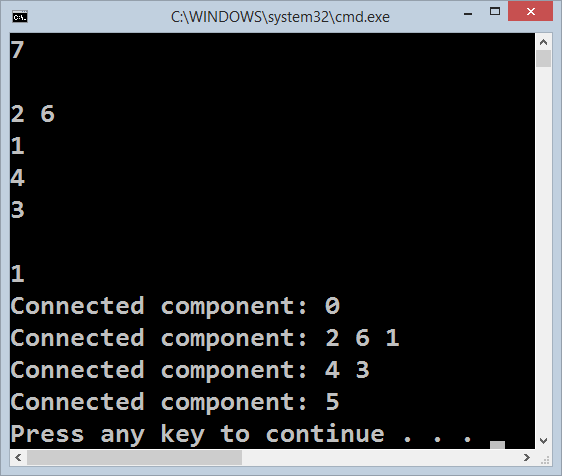
Modify the main method to read the graph from the console instead using the hard-coded graph:



Now test the program. Run it ([Ctrl] + [F5]). Enter a sample graph data and check the output:

|  |  |  |
| --- | --- | --- |
| **Input** | **Graph** | **Expected Output** |
| 7  2 6  1  4  3  1 |  | Connected component: 0  Connected component: 2 6 1  Connected component: 4 3  Connected component: 5 |

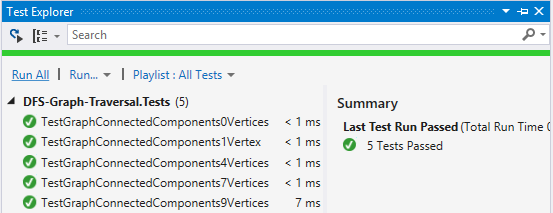
Seems like it runs correctly:



We are ready for the unit tests.

## Run the Unit Tests

Seems like we solved the graph problem. Let's run the unit tests that come with the program skeleton:

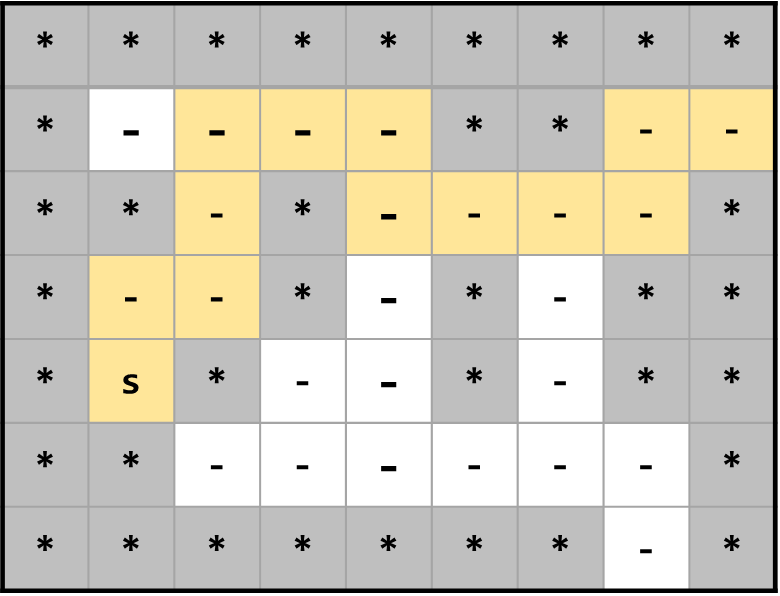


Congratulations! You have implemented the DFS algorithm to find all connected components in a graph.

# Part II – Find the Nearest Exit from a Labyrinth

The second part of this lab aims to implement the **Breadth-First-Search (BFS) algorithm** to find the nearest possible exit from a labyrinth. We are given a labyrinth. We start from a cell denoted by 's'. We can move **left**, **right**, **up** and **down**, through empty space cells '-'. We cannot pass through walls '\*'. An exit is found when a cell on a labyrinth side is reached.

For **example**, consider the labyrinth below. It has size **9 x 7**:



We start from cell {1, 4}, denoted by 's'. Тhe nearest exit is at the right side, the cell {8, 1}. The path to the nearest exit consists of **12** moves: URUURRDRRRUR (where 'U' means up, 'R' means right, 'D' means down and 'L' means left). There are two exits and several other paths to these exits, but the path URUURRDRRRUR is the shortest.

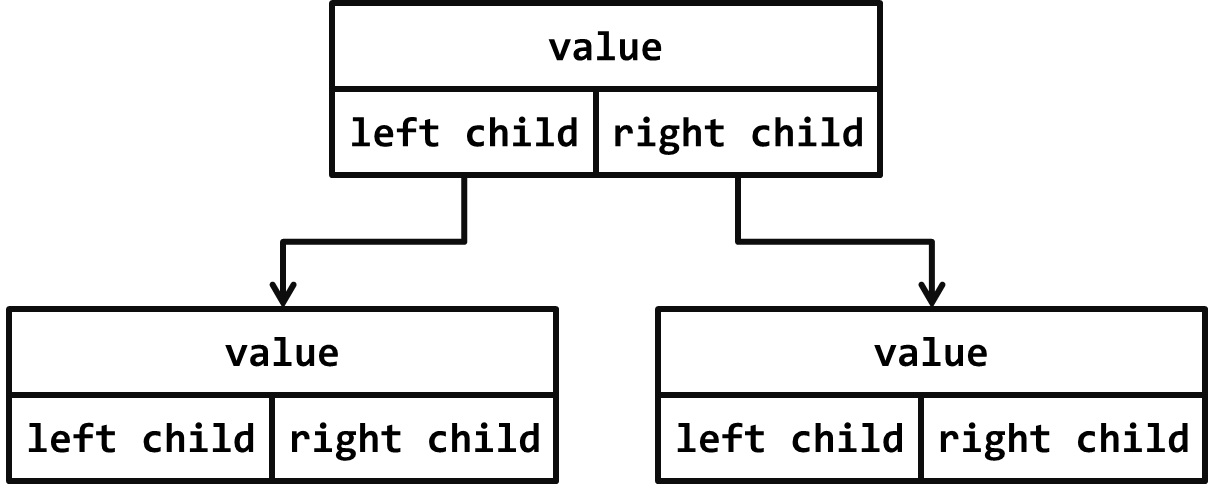
The input comes from the console. The first line holds the labyrinth width. The second line holds the labyrinth height. The next height lines hold the labyrinth cells – characters '\*' (wall), '-' (empty cell) or 's' (start cell).

Examples:

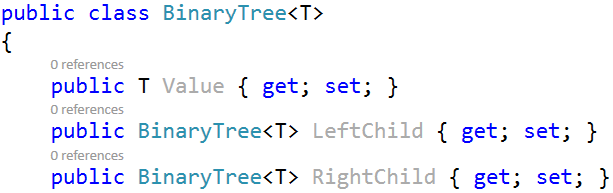
|  |  |  |
| --- | --- | --- |
| **Input** | **Labyrinth** | **Output** |
| 9  7  \*\*\*\*\*\*\*\*\*  \*----\*\*--  \*\*-\*----\*  \*--\*-\*-\*\*  \*s\*--\*-\*\*  \*\*------\*  \*\*\*\*\*\*\*-\* |  | Shortest exit: URUURRDRRRUR |
| 4  3  \*\*\*\*  \*-s\*  \*\*\*\* |  | No exit! |
| 4  2  \*\*\*\*  \*\*\*s |  | Start is at the exit. |
| 2  2  \*\*  \*\* |  | No exit! |

## 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):

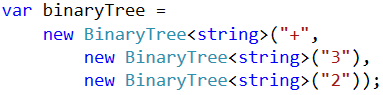


The source code might look like this:

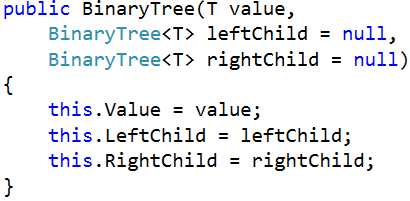


## Define the BinaryTree<T> Constructor

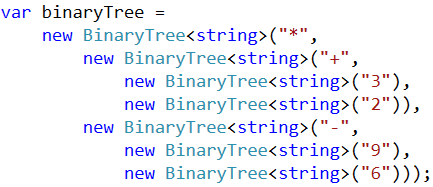
The next step is to define a **constructor** for the BinaryTree<T> class to ensure you can create:

* **Leaf tree nodes** (holding a specified value) without child nodes, e.g.  
  
* **Internal tree nodes** (holding a specified value) with left and right child nodes, e.g.  
  

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:



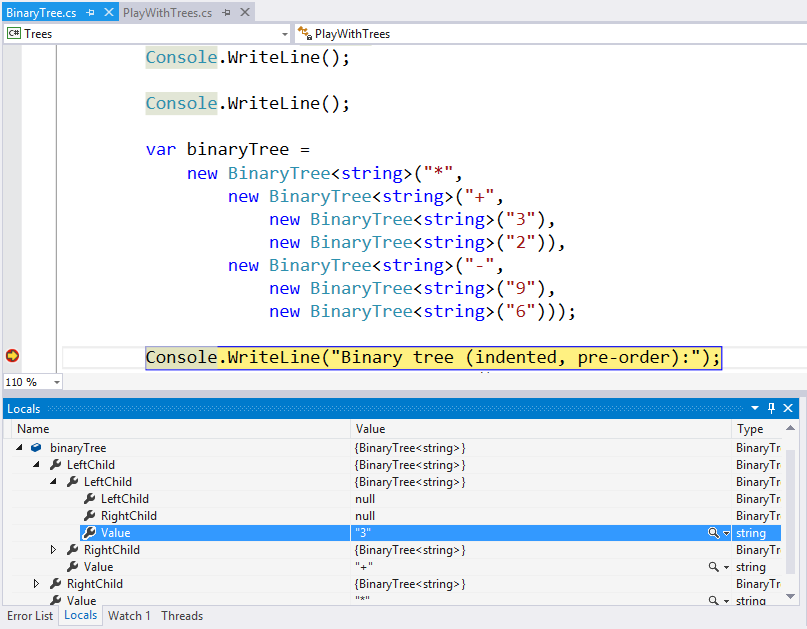
The parameters leftChild and rightChild are optional and can be passed or skipped. This will allow constructing binary tree like this:



## Test the BinaryTree<T> Constructor

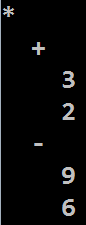
Now, test whether the BinaryTree<T> and its constructor work as expected.

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



## Define the PrintIndentedPreOrder() Method

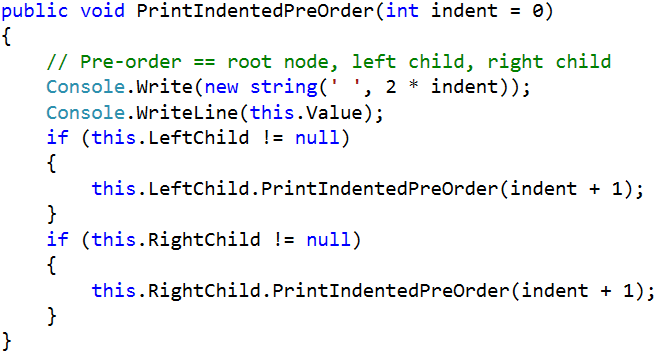
Now, we need to implement the **binary** **tree functionality**. First, implement the PrintIndentedPreOrder() method. It prints the tree in pre-order (root; left; right), indented visually like this:



The PrintIndentedPreOrder() method works recursively:

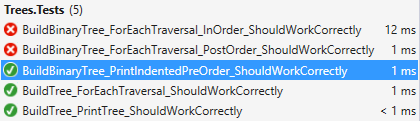
* Prints the current node value (indented a few spaces on the right).
* Calls the PrintIndentedPreOrder() method recursively to **print the left child** of the current node (when exists).
* Calls the PrintIndentedPreOrder() method recursively to **print the right child** of the current node (when exists).

The code might look like this:



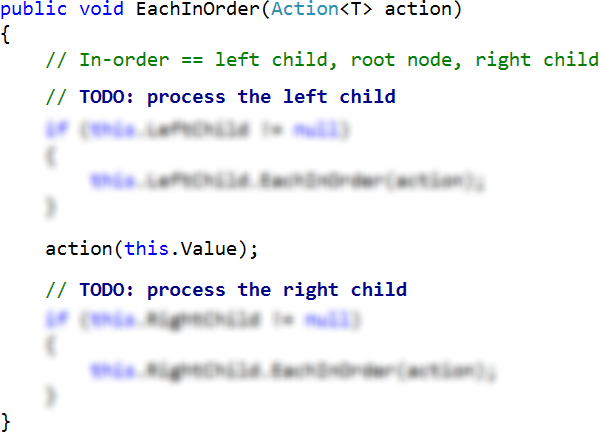
## Test the PrintIndentedPreOrder() Method

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



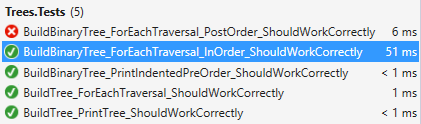
## Implement the EachInOrder(Action<T>) Method

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:



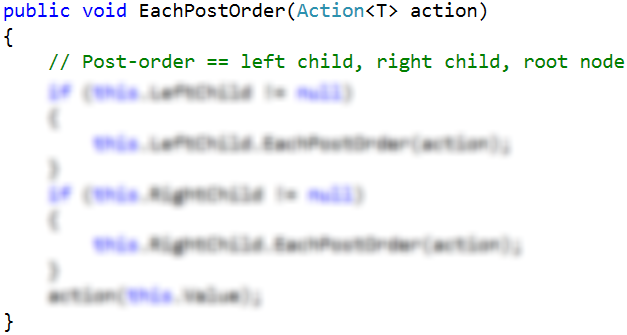
## Test the EachInOrder(Action<T>) Method

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



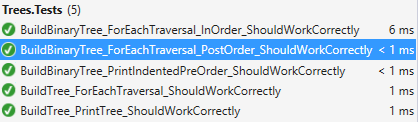
## Implement the EachPostOrder(Action<T>) Method

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:



## Test the EachPostOrder(Action<T>) Method

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



Congratulations! You have implemented your binary tree data structure.