# Exercises: QuadTree

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 - Implement a QuadTree

# Overview

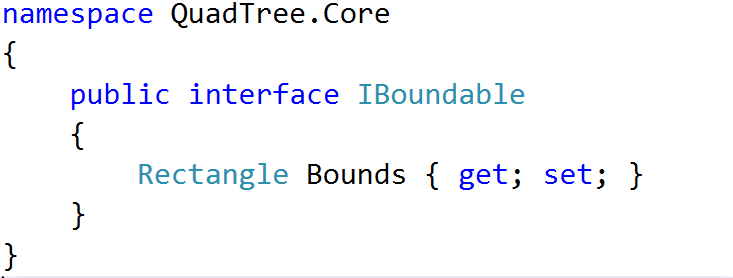
A **QuadTree** is a space-partitioning tree that divides 2D space into **quads** (regions of 2D space).

* The root represents the whole 2D world.
* A node has either 0 or 4 children.

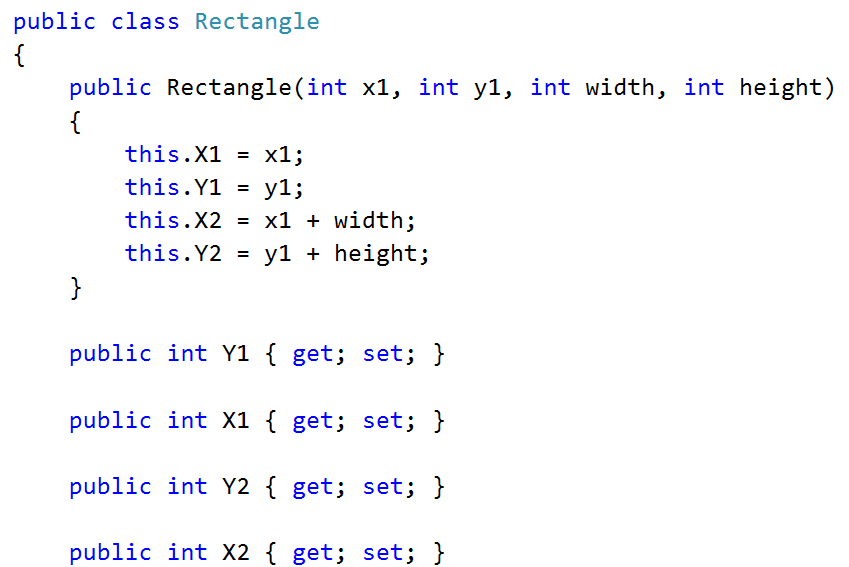
|  |
| --- |
|  |

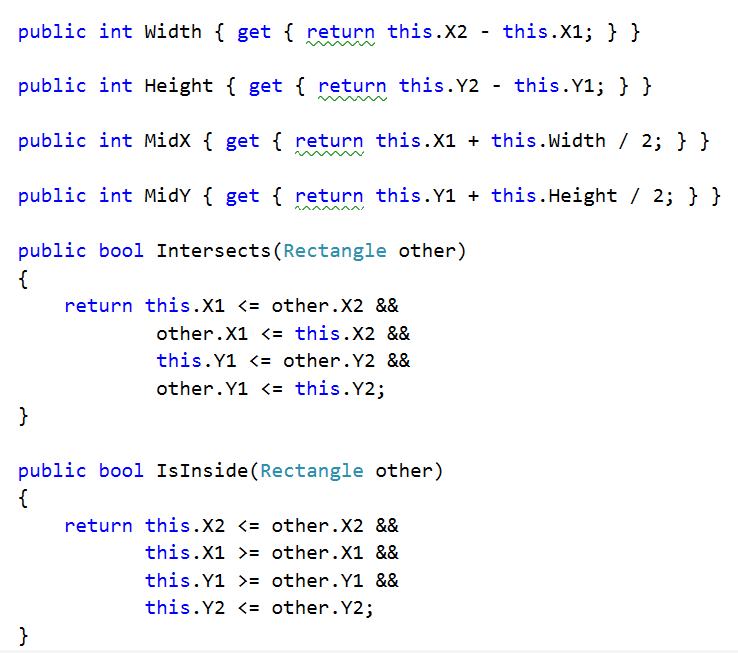
# Classes

* **IBoundable** interface - implemented by classes which can be stored in the QuadTree

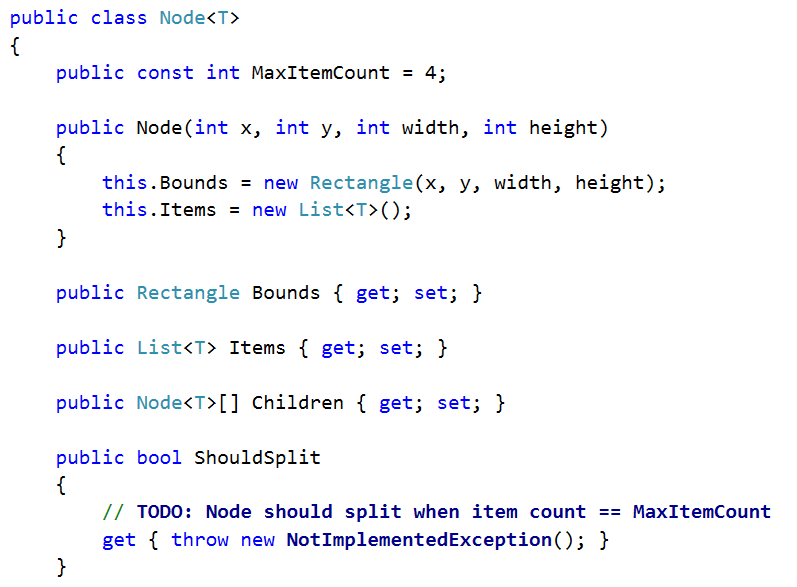


* **Rectangle** - holds information about a 2D object (coordinates, size and some helper methods)





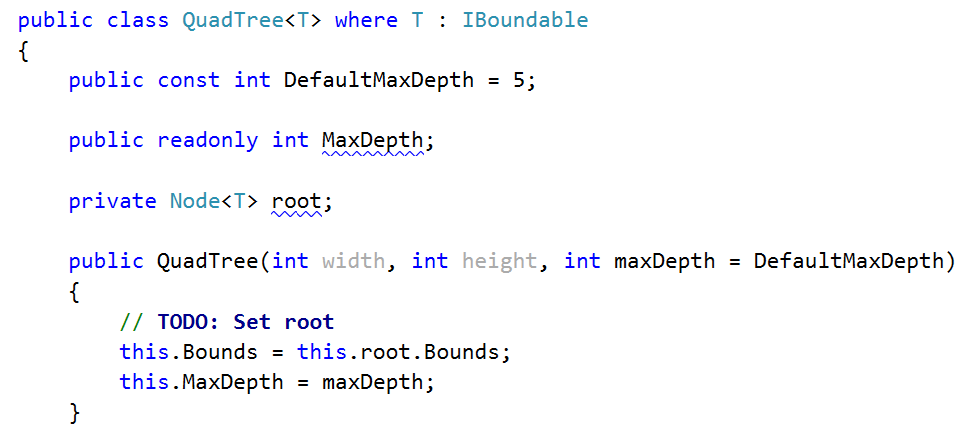
* **Node<T>** - node class for holding node data in our QuadTree



Let's start!

## Constructing the Tree

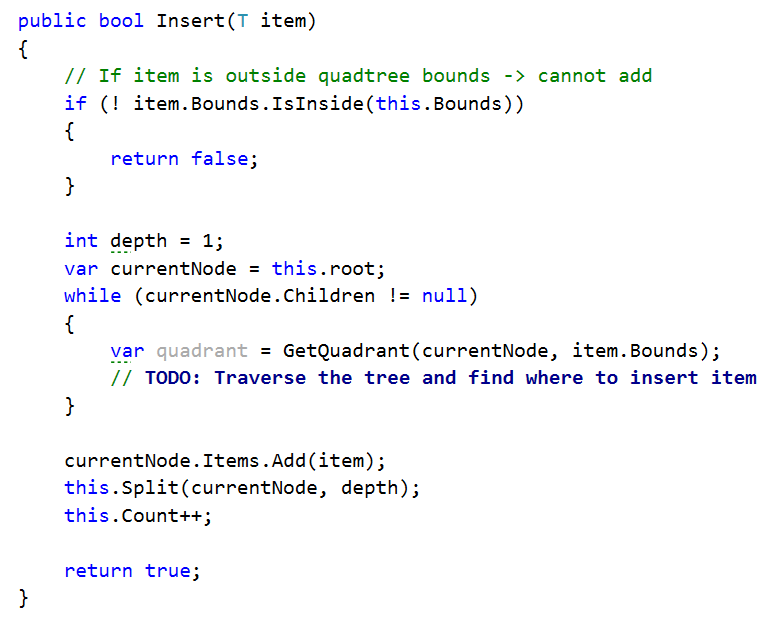
First, let's implement the constructor. Whenever we create a new QuadTree, we initialize a **root node** with   
**(X = 0, Y = 0)** and **Width** and **Height** as passed to the constructor.



## Item Insertion

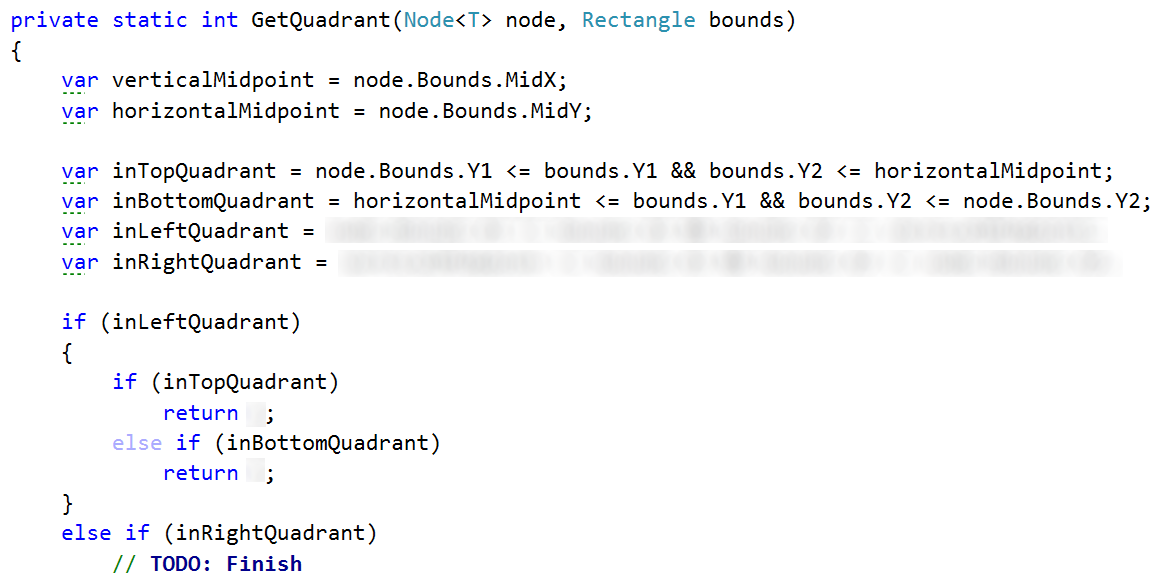
Inserting a node is quite simple.

1. If its bounds are **outside the bounds of the Quadtree**, we do **not insert it** (since it cannot be contained correctly). We use the **IsInside(Rectangle other)** helper method from the **Rectangle** class to determine this.
2. We start searching down the tree where to insert the item. The goal is to insert it at the **most lower possible node which can contain the entire item**.
   * Just like an insertion in an ordinary Binary Search Tree, we use a **while**-loop to traverse down the tree. How do we know which direction to go? We **check which quadrant of the current node** can hold the item.   
     We check that using the **GetQuadrant(Node<T> node, Rectangle bounds)** method (we will write that in a moment). It returns the respective quadrant (from 0 to 3).
   * If a **quadrant == -1** -> the item cannot fit in any quadrant, so we stop and insert it in the current node.
3. And finally, we call the **Split(Node<T> node, int depth)** method. It will **split the node into 4 quadrants** if the node has reached its max capacity for items.
4. Insertion successful -> increase the count by 1.



## Get Quadrant

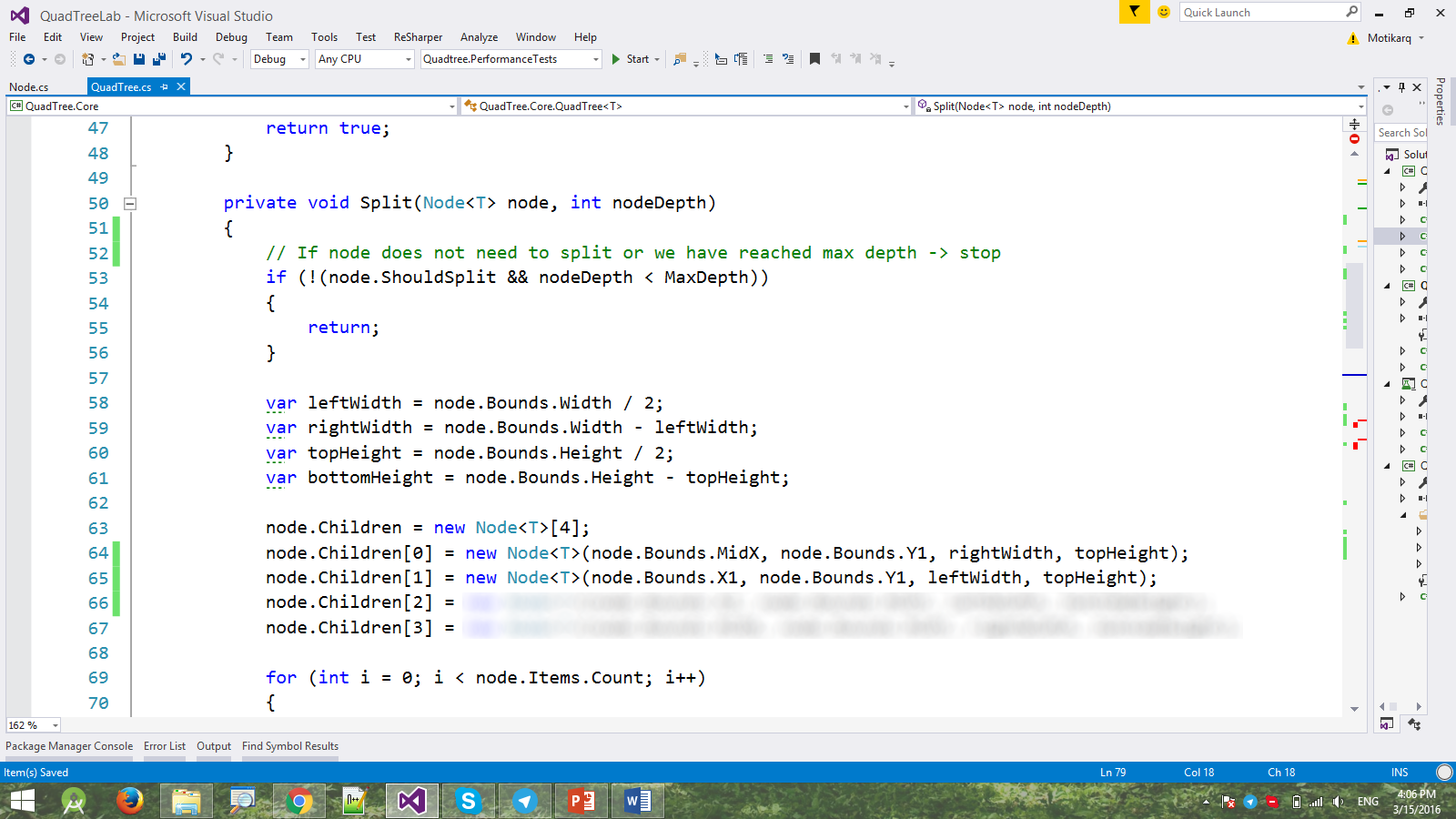
Before we go into splitting, let's write the code for determining **which quadrant** our search area is fully contained in.



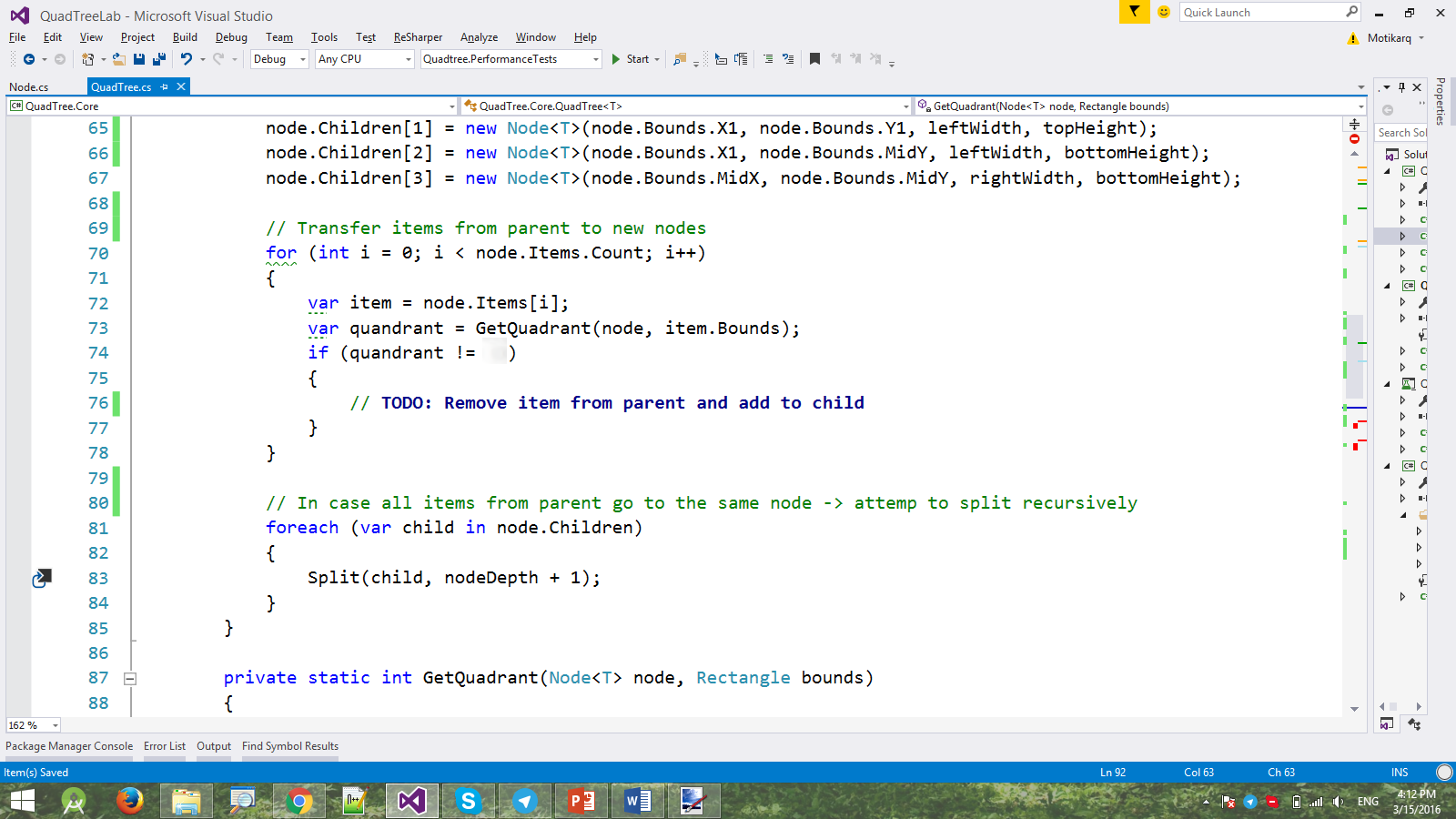
## Splitting

A node is split after adding if it holds **items == max capacity** and its **depth < MaxDepth**.

We then subdivide the current node into **4 sub-nodes** (each representing a **quadrant of its parent**).



After splitting, it's time to transfer all items which can fit in subtrees from the parent to the childen. How do we know which child we move an item to? By checking the **item** against each **quadrant,** if the **item** fits completely inside the **quadrant,** we insert it in the corresponding child. Otherwise, we leave it in the current node.

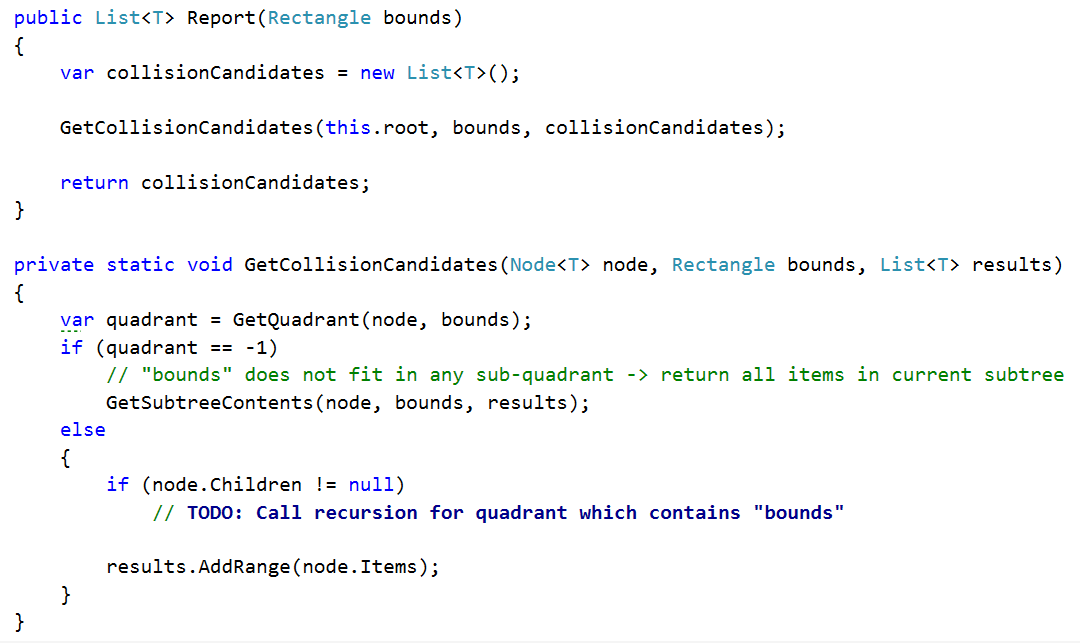


Finally, we **recursively** **call** **Split()** for each child (this is done because a child may have its **max capacity filled** and **should split as well**).

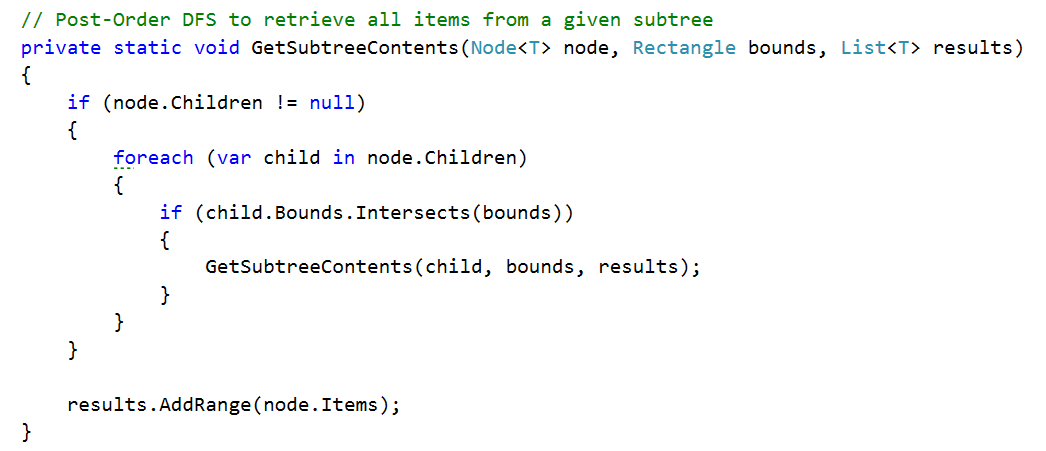
## Reporting Possible Collisions

The QuadTree's main strength is to retrieve only the items that might intersect a given bound.

* In order to find the intersected items we can narrow down the search by checking if the passed **bound** fits into a child **quadrant.** 
  + If it does, we recursively repeat the process for the child **quadrant** that contains the **bound.**
* On the backtrack of the recursion we add the current node's items to the list of results.



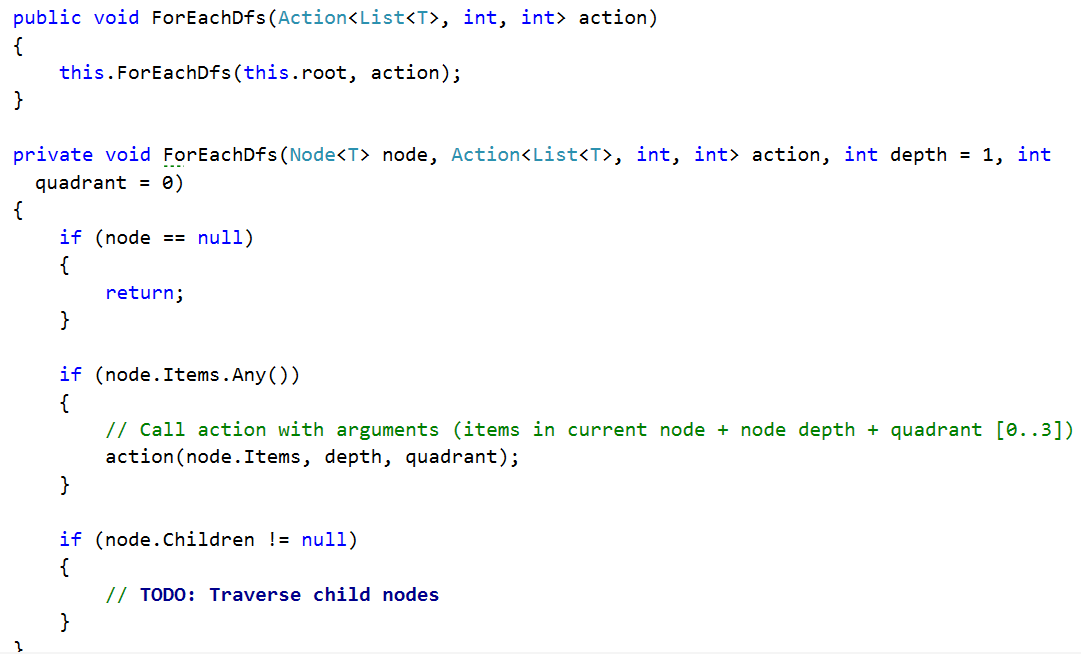
If we reach a point where the **bound** no longer fits into a child **quadrant** we recursively traverse the current subtree and add all intersecting items to the list of results.



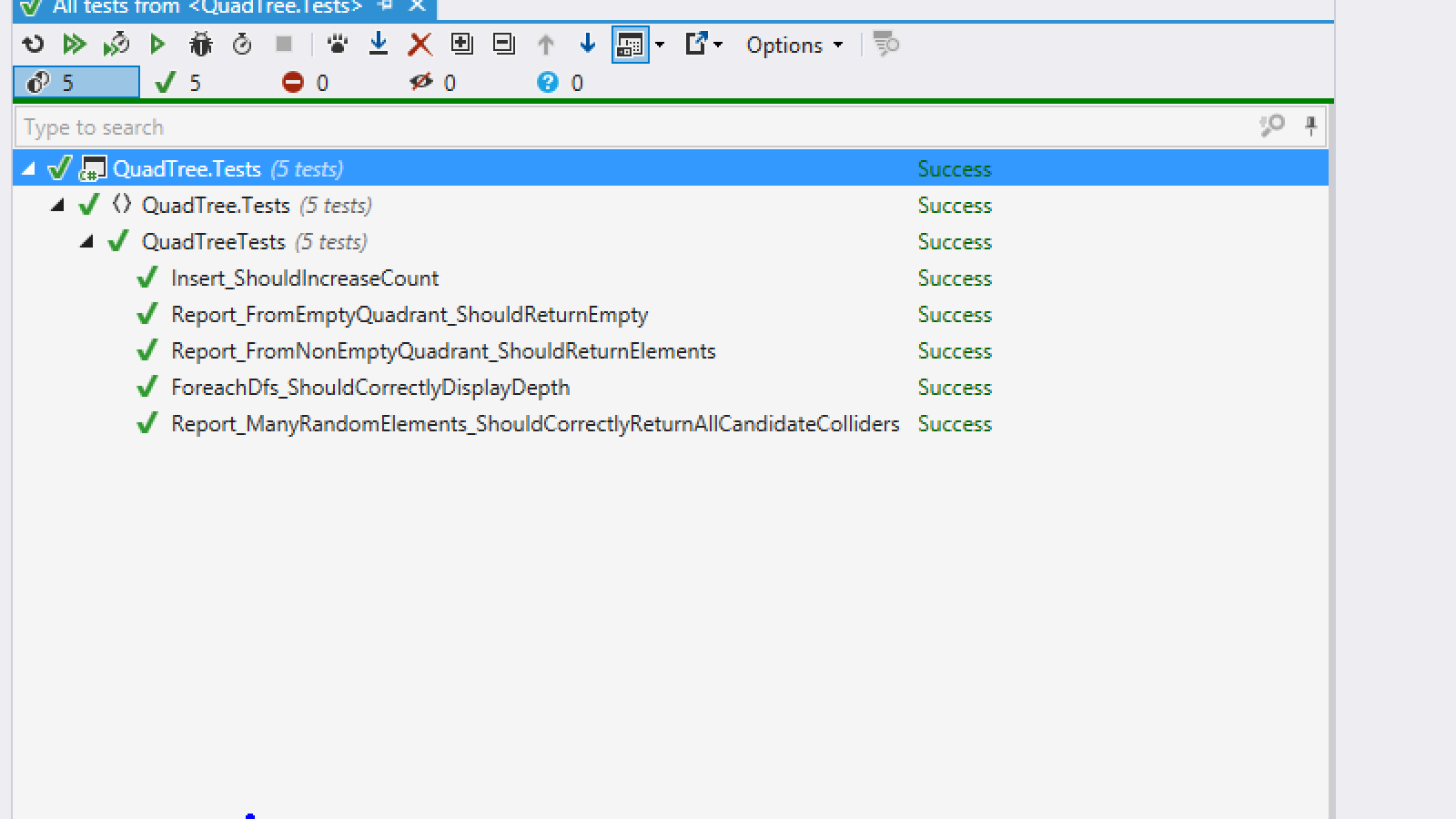
## DFS Foreach

Finally we implement a DFS. This will allow us to traverse the tree and check if the items have been correctly distributed.

We start at the **root** and if the current node contains **items** we call the passed action on each of the items. We then recursively repeat the same operation for all the children.



Run the unit tests. If all is correct, they should pass.



Congratulations! You have implemented your QuadTree!