Exercises: QuadTree

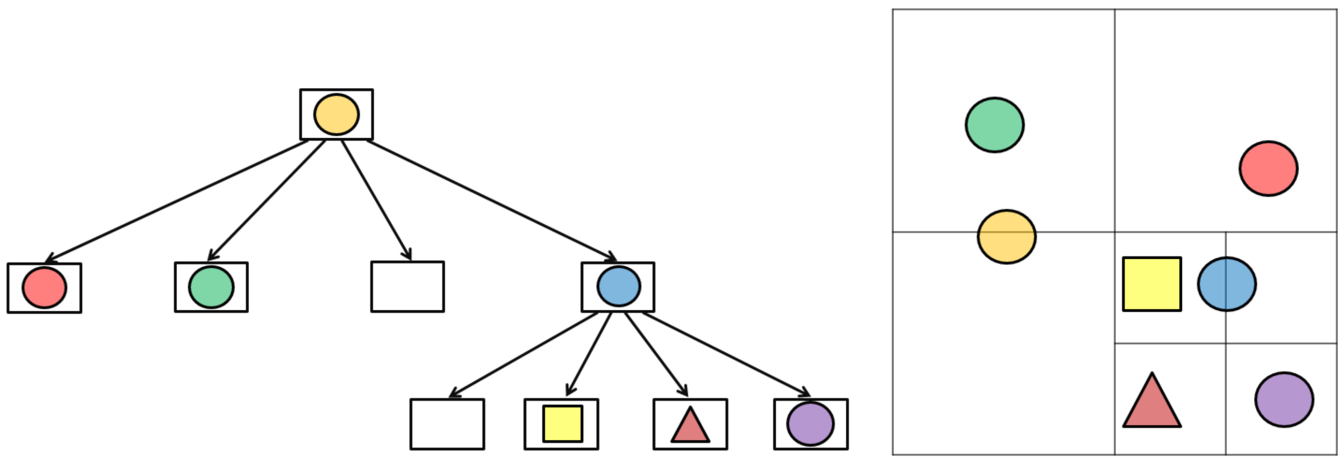
This document defines the **in-class exercises** assignments for the "Data Structures" course @ Software University.

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

```
namespace QuadTree.Core
{
    public interface IBoundable
    {
        Rectangle Bounds { get; set; }
    }
}
```

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

```
public class Rectangle
{
    public Rectangle(int x1, int y1, int width, int height)
    {
        this.X1 = x1;
        this.Y1 = y1;
        this.X2 = x1 + width;
        this.Y2 = y1 + height;
    }

    public int Y1 { get; set; }

    public int X1 { get; set; }

    public int Y2 { get; set; }

    public int X2 { get; set; }

    public int Width { get { return this.X2 - this.X1; } }

    public int Height { get { return this.Y2 - this.Y1; } }

    public int MidX { get { return this.X1 + this.Width / 2; } }

    public int MidY { get { return this.Y1 + this.Height / 2; } }

    public bool Intersects(Rectangle other)
    {
        return this.X1 <= other.X2 &&
                other.X1 <= this.X2 &&
                this.Y1 <= other.Y2 &&
                other.Y1 <= this.Y2;
    }

    public bool IsInside(Rectangle other)
    {
        return this.X2 <= other.X2 &&
               this.X1 >= other.X1 &&
               this.Y1 >= other.Y1 &&
               this.Y2 <= other.Y2;
    }
```

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

```
public class Node<T>
{
    public const int MaxItemCount = 4;

    public Node(int x, int y, int width, int height)
    {
        this.Bounds = new Rectangle(x, y, width, height);
        this.Items = new List<T>();
    }

    public Rectangle Bounds { get; set; }

    public List<T> Items { get; set; }

    public Node<T>[] Children { get; set; }

    public bool ShouldSplit
    {
        // TODO: Node should split when item count == MaxItemCount
        get { throw new NotImplementedException(); }
    }
```

Let's start!

Problem 1. 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.

```
public class QuadTree<T> where T : IBoundable
{
    public const int DefaultMaxDepth = 5;

    public readonly int MaxDepth;

    private Node<T> root;

    public QuadTree(int width, int height, int maxDepth = DefaultMaxDepth)
    {
        // TODO: Set root
        this.Bounds = this.root.Bounds;
        this.MaxDepth = maxDepth;
    }
```

Problem 2. 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.

```
public bool Insert(T item)
{
    // If item is outside quadtree bounds -> cannot add
    if (! item.Bounds.IsInside(this.Bounds))
    {
        return false;
    }

    int depth = 1;
    var currentNode = this.root;
    while (currentNode.Children != null)
    {
        var quadrant = GetQuadrant(currentNode, item.Bounds);
        // TODO: Traverse the tree and find where to insert item
    }

    currentNode.Items.Add(item);
    this.Split(currentNode, depth);
    this.Count++;

    return true;
}
```

Problem 3. Get Quadrant

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

```
private static int GetQuadrant(Node<T> node, Rectangle bounds)
{
    var verticalMidpoint = node.Bounds.MidX;
    var horizontalMidpoint = node.Bounds.MidY;

    var inTopQuadrant = node.Bounds.Y1 <= bounds.Y1 && bounds.Y2 <= horizontalMidpoint;
    var inBottomQuadrant = horizontalMidpoint <= bounds.Y1 && bounds.Y2 <= node.Bounds.Y2;
    var inLeftQuadrant = [redacted]
    var inRightQuadrant = [redacted]

    if (inLeftQuadrant)
    {
        if (inTopQuadrant)
            return [redacted];
        else if (inBottomQuadrant)
            return [redacted];
    }
    else if (inRightQuadrant)
        // TODO: Finish
```

Problem 4. 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**).

```
private void Split(Node<T> node, int nodeDepth)
{
    // If node does not need to split or we have reached max depth -> stop
    if (!(node.ShouldSplit && nodeDepth < MaxDepth))
    {
        return;
    }

    var leftWidth = node.Bounds.Width / 2;
    var rightWidth = node.Bounds.Width - leftWidth;
    var topHeight = node.Bounds.Height / 2;
    var bottomHeight = node.Bounds.Height - topHeight;

    node.Children = new Node<T>[4];
    node.Children[0] = new Node<T>(node.Bounds.MidX, node.Bounds.Y1, rightWidth, topHeight);
    node.Children[1] = new Node<T>(node.Bounds.X1, node.Bounds.Y1, leftWidth, topHeight);
    node.Children[2] = [redacted]
    node.Children[3] = [redacted]
```

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.

```
    // Transfer items from parent to new nodes
    for (int i = 0; i < node.Items.Count; i++)
    {
        var item = node.Items[i];
        var quandrant = GetQuadrant(node, item.Bounds);
        if (quandrant !=    )
        {
            // TODO: Remove item from parent and add to child
        }
    }

    // In case all items from parent go to the same node -> attemp to split recursively
    foreach (var child in node.Children)
    {
        Split(child, nodeDepth + 1);
    }
}
```

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**).

Problem 5. 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.

```
public List<T> Report(Rectangle bounds)
{
    var collisionCandidates = new List<T>();

    GetCollisionCandidates(this.root, bounds, collisionCandidates);

    return collisionCandidates;
}

private static void GetCollisionCandidates(Node<T> node, Rectangle bounds, List<T> results)
{
    var quadrant = GetQuadrant(node, bounds);
    if (quadrant == -1)
        // "bounds" does not fit in any sub-quadrant -> return all items in current subtree
        GetSubtreeContents(node, bounds, results);
    else
    {
        if (node.Children != null)
            // TODO: Call recursion for quadrant which contains "bounds"

        results.AddRange(node.Items);
    }
}
```

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.

```
// Post-Order DFS to retrieve all items from a given subtree
private static void GetSubtreeContents(Node<T> node, Rectangle bounds, List<T> results)
{
    if (node.Children != null)
    {
        foreach (var child in node.Children)
        {
            if (child.Bounds.Intersects(bounds))
            {
                GetSubtreeContents(child, bounds, results);
            }
        }
    }

    results.AddRange(node.Items);
}
```

Problem 6. 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.

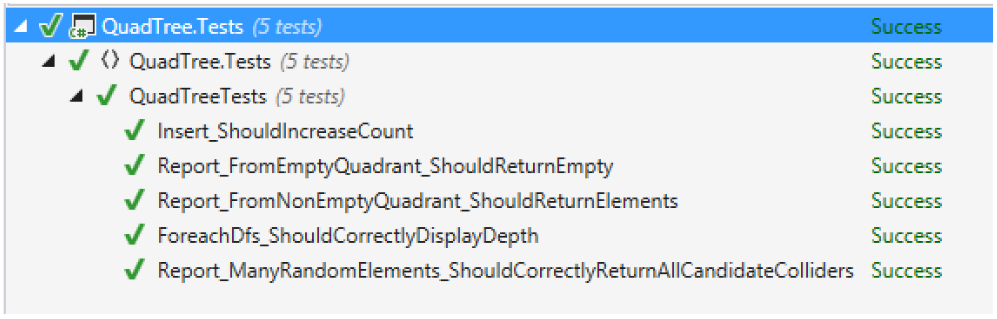
```csharp
public void ForEachDfs(Action<List<T>, int, int> action)
{
    this.ForEachDfs(this.root, action);
}

private void ForEachDfs(Node<T> node, Action<List<T>, int, int> action, int depth = 1, int
  quadrant = 0)
{
    if (node == null)
    {
        return;
    }

    if (node.Items.Any())
    {
        // Call action with arguments (items in current node + node depth + quadrant [0..3])
        action(node.Items, depth, quadrant);
    }

    if (node.Children != null)
    {
        // TODO: Traverse child nodes
    }
}
```

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



```
◢ ✓ QuadTree.Tests (5 tests)                                                  Success
  ◢ ✓ QuadTree.Tests (5 tests)                                                Success
    ◢ ✓ QuadTreeTests (5 tests)                                               Success
        ✓ Insert_ShouldIncreaseCount                                          Success
        ✓ Report_FromEmptyQuadrant_ShouldReturnEmpty                          Success
        ✓ Report_FromNonEmptyQuadrant_ShouldReturnElements                    Success
        ✓ ForeachDfs_ShouldCorrectlyDisplayDepth                              Success
        ✓ Report_ManyRandomElements_ShouldCorrectlyReturnAllCandidateColliders Success
```

Congratulations! You have implemented your QuadTree!