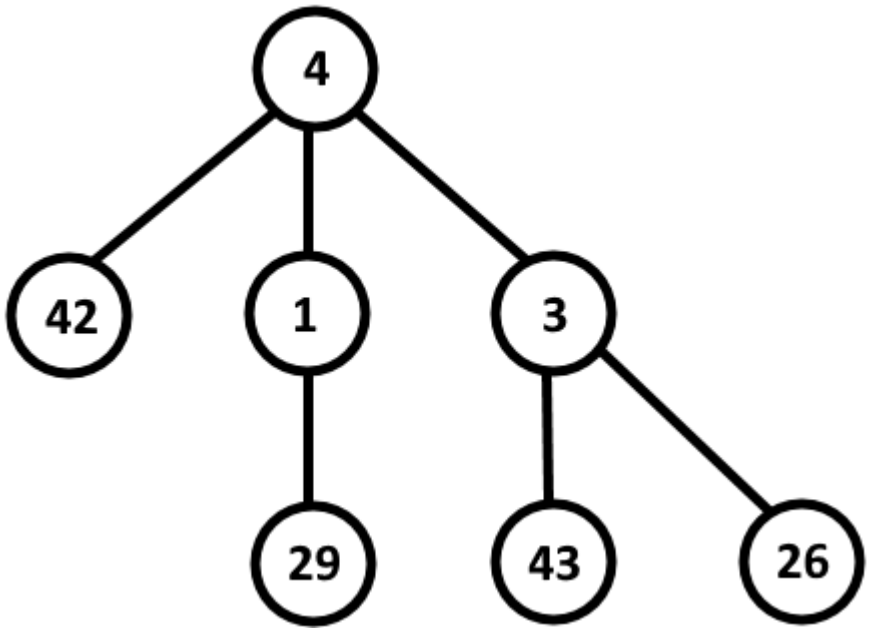
Trees Reading

Tim "Dr. T" Chamillard

What's a Tree?

A tree is a set of nodes connected by edges; the picture below shows one example.



Trees have a *root*, which is the node at the top of the tree; in the picture above the node containing 4 is the root of the tree. Every node in the tree has 0 or more children; in the picture above, node 4 has 3 children (nodes 42, 1, and 3), node 3 has 2 children (nodes 43 and 26), node 1 has 1 child (node 29), and nodes 42, 29, 43, and 26 don't have any children. A node with no children is called a *leaf node* and a node with one or more children is called a *branch node*. Each node in a tree (except for the root node) has exactly one parent. Finally, trees can't have cycles (for example, node 42 can't add node 4 as a child) because trees represent a hierarchical structure. We didn't discuss it this way when we learned about inheritance, but the parent-child relationship in an inheritance hierarchy are actually a tree.

There are actually lots of kinds of trees. For example, in a *binary tree* each node can only have 0, 1, or 2 children. A *binary search tree* is a binary tree where all the nodes in a subtree rooted in a left child of a node are less than the node value and all the nodes in a subtree rooted in a right child of a node are greater than the node value. This ordering supports very efficient searching, insertion, and deletion on average. You also may run into lots of other kinds of trees in your programming career, including *red-black trees*, *AVL trees*, *B-trees*, and others.

How are trees useful in game development? The most common place I've personally used trees is in turn-based games, where I need the AI to pick a reasonable move to

make by "looking forward" a number of moves to pick the best move (while also taking into account the current difficulty of the game). We'll see this when we explore *minimax trees* in one of the lectures and in a programming assignment.

Because there are so many different ways to implement a tree data structure that meets your specific needs, C# doesn't provide a `Tree` class. If you need one, you need to write one! That's okay; we'll do that now.

Tree Node

Let's start with the `TreeNode` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Trees
{
    /// <summary>
    /// A tree node
    /// </summary>
    /// <typeparam name="T">type of value stored in node</typeparam>
    class TreeNode<T>
    {
```

We're making our node a generic so it can contain any data type for the node value.

```
        #region Fields

        T value;
        TreeNode<T> parent;
        List<TreeNode<T>> children;

        #endregion
```

Each node contains a value, a reference to its parent, and a list of its child nodes.

```
        #region Constructors

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="value">value for the node</param>
        /// <param name="parent">parent for the node</param>
        public TreeNode(T value, TreeNode<T> parent)
        {
            this.value = value;
            this.parent = parent;
            children = new List<TreeNode<T>>();
        }
```

```
        #endregion
```

The constructor is pretty straightforward, initializing the value, setting the parent, and creating a new (empty) list of children.

```
        #region Properties

        /// <summary>
        /// Gets the value stored in the node
        /// </summary>
        public T Value
        {
            get { return value; }
        }
```

The `Value` property is read-only, so we can get the node value but we can't change it after the node has been created.

```
        /// <summary>
        /// Gets and sets the parent of the node
        /// </summary>
        public TreeNode<T> Parent
        {
            get { return parent; }
            set { parent = value; }
        }
```

The `Parent` property provides both read and write access. We need to be able to read the property when we're adding a node to a tree (to make sure its parent is in the tree) and we need to be able to write the property when we're making a node a child of another node.

```
        /// <summary>
        /// Gets a read-only list of the children of the node
        /// </summary>
        public IList<TreeNode<T>> Children
        {
            get { return children.AsReadOnly(); }
        }

        #endregion
```

The `Children` property provides read-only access to the node's children; this is just like the `Children` property in our `GraphNode` class.

```
        #region Methods

        /// <summary>
        /// Adds the given node as a child this node
        /// </summary>
        /// <param name="child">child to add</param>
        /// <returns>true if the child was added, false otherwise</returns>
```

```
public bool AddChild(TreeNode<T> child)
{
    // don't add duplicate children
    if (children.Contains(child))
    {
        return false;
    }
    else if (child == this)
    {
        // don't add self as child
        return false;
    }
    else
    {
        // add as child and add self as parent
        children.Add(child);
        child.Parent = this;
        return true;
    }
}
```

The `AddChild` method adds a node to the children for the node as long as that node isn't already a child and as long as we're not trying to make a node a child of itself.

The `List Contains` method performs an O(n) linear search, so the AddChild operation is O(n) where n is the number of children.

```
/// <summary>
/// Removes the given node as a child this node
/// </summary>
/// <param name="child">child to remove</param>
/// <returns>true if the child was removed, false otherwise</returns>
public bool RemoveChild(TreeNode<T> child)
{
    // only remove children in list
    if (children.Contains(child))
    {
        child.Parent = null;
        return children.Remove(child);
    }
    else
    {
        return false;
    }
}
```

We're taking advantage of the fact that the `List Remove` method returns false if the child we're trying to remove isn't in the list.

The `List Contains` and `Remove` methods both perform O(n) linear searches, so the RemoveChild operation is O(n) where n is the number of children.

```
/// <summary>
/// Removes all the children for the node
```

```
        /// </summary>
        /// <returns>true if the children were removed,
        ///     false otherwise</returns>
        public bool RemoveAllChildren()
        {
            for (int i = children.Count - 1; i >= 0; i--)
            {
                children[i].Parent = null;
                children.RemoveAt(i);
            }
            return true;
        }
```

The `RemoveAllChildren` method removes all the children for the given node. This is useful when we want to clear a tree of nodes because removing the reference from a node to its children makes it so the garbage collector can collect the nodes.

The `List RemoveAt` method is O(Count - index) where index is the location we want to remove. We're always passing in an index of Count - 1 (because we're going backwards through the list), so the body of the loop is O(1) because Count - (Count - 1) is 1. The loop executes n times, where n is the number of children, so the RemoveAllChildren operation is O(n).

```
        /// <summary>
        /// Converts the node to a string
        /// </summary>
        /// <returns>the string</returns>
        public override string ToString()
        {
            StringBuilder nodeString = new StringBuilder();
            nodeString.Append("[Node Value: " + value +
                " Parent: ");
            if (parent != null)
            {
                nodeString.Append(parent.Value);
            }
            else
            {
                nodeString.Append("null");
            }
            nodeString.Append(" Children: ");
            for (int i = 0; i < children.Count; i++)
            {
                nodeString.Append(children[i].Value + " ");
            }
            nodeString.Append("]");
            return nodeString.ToString();
        }

        #endregion
    }
}
```

I implemented the `ToString` method to support the test cases, but being able to get a string representation of the node could be useful in other scenarios as well. The ToString operation is clearly O(n) because the for loop executes n times where n is the number of children.

The `TestTreeNode` class in the code accompanying this reading runs a variety of tests against the code above. Feel free to take a look at the test cases if you'd like.

Tree

Okay, on to the `Tree` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Trees
{
    /// <summary>
    /// A tree
    /// </summary>
    /// <typeparam name="T">type of values stored in tree</typeparam>
    class Tree<T>
    {
        #region Fields

        TreeNode<T> root = null;
        List<TreeNode<T>> nodes = new List<TreeNode<T>>();

        #endregion
```

Our `Tree` class is also a generic, of course. We need a field that holds a reference to the root of the tree and a field that holds a list of the nodes in the tree. Unlike a graph, which doesn't have a set starting point, for a tree the root is the starting point for traversing the tree.

```
        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="value">value of the root node</param>
        public Tree(T value)
        {
            root = new TreeNode<T>(value, null);
            nodes.Add(root);
        }

        #endregion
```

Our constructor creates a new tree node with the given value, sets that node as the root of the tree, and adds the root to the list of nodes in the tree.

```
#region Properties

/// <summary>
/// Gets the number of nodes in the tree
/// </summary>
public int Count
{
    get { return nodes.Count; }
}

/// <summary>
/// Gets the root of the tree
/// </summary>
public TreeNode<T> Root
{
    get { return root; }
}

#endregion
```

We provide a `Count` property to get the number of nodes in the tree and the `Root` property gets the root of the tree since most consumers of the class will start traversing the tree at the root.

```
/// <summary>
/// Clears all the nodes from the tree
/// </summary>
public void Clear()
{
    // remove all the children from each node
    // so nodes can be garbage collected
    foreach (TreeNode<T> node in nodes)
    {
        node.Parent = null;
        node.RemoveAllChildren();
    }

    // now remove all the nodes from the tree and set root to null
    for (int i = nodes.Count - 1; i >= 0; i--)
    {
        nodes.RemoveAt(i);
    }
    root = null;
}
```

We first unlink all the child nodes for each node so the garbage collector can collect them. After doing that, we remove all the nodes from the tree.

The second loop executes n times, where n is the number of nodes in the tree, but what about the first loop?

We know the loop executes n times, where n is the number of nodes in the tree. We also know that the `TreeNode RemoveAllChildren` method is O(n), where n is the number of children that node has. We can see that n is defined in two different ways, so how do we figure out the complexity of the loop?

The key is that each node can only have one parent (the root doesn't have a parent). That means that each node can only be removed as a child of some parent once. We iterate over all the nodes in the loop but the body of the loop only removes a total of n nodes (where n is the number of nodes in the graph) as children over all the iterations of the loop body. That makes the first loop O(n).

With two O(n) loops and a constant-time operation setting root to null, the Clear operation is O(n).

```
/// <summary>
/// Adds the given node to the tree. If the given node is
/// null the method returns false. If the parent node is null
/// or isn't in the tree the method returns false. If the given
/// node is already a child of the parent node the method returns
/// false
/// </summary>
/// <param name="node">node to add</param>
/// <returns>true if the node is added, false otherwise</returns>
public bool AddNode(TreeNode<T> node)
{
    if (node == null ||
        node.Parent == null ||
        !nodes.Contains(node.Parent))
    {
        return false;
    }
    else if (node.Parent.Children.Contains(node))
    {
        // node already a child of parent
        return false;
    }
    else
    {
        // add child as tree node and as a child to parent
        nodes.Add(node);
        return node.Parent.AddChild(node);
    }
}
```

The `AddNode` method is more complicated than we saw when adding a node to a graph. We start by making sure the node that was passed in isn't null and has a valid parent. The node need to have a valid parent because if it doesn't, we'd actually be adding another root to the tree (because the new node wouldn't be connected to the tree). Trees are only allowed to have a single root. We also don't add the node if that node is already a child of the parent because that would give us a duplicate edge between the parent and the new node. If none of the error conditions apply, we add the new node to the list of tree nodes and add it as a child of its parent. Although the new node already

links back to the parent, the parent doesn't have the new node as a child yet; that's why we need to call the `AddChild` method.

The `List Contains` method performas a linear search, so the first and second Boolean expressions contribute O(n). The `List Add` method is O(n) if the list needs to be expanded and in the worst case (where the root holds all the other nodes as children) the call to the `AddChild` method is also O(n). That means the AddNode operation is O(n), where n is the number of nodes in the tree.

```
/// <summary>
/// Removes the given node from the tree. If the node isn't
/// found in the tree, the method returns false.
///
/// Note that the subtree with the node to remove as its
/// root is pruned from the tree
/// </summary>
/// <param name="removeNode">node to remove</param>
/// <returns>true if the node is removed, false otherwise</returns>
public bool RemoveNode(TreeNode<T> removeNode)
{
    if (removeNode == null)
    {
        return false;
    }
    else if (removeNode == root)
    {
        // removing the root clears the tree
        Clear();
        return true;
    }
    else
    {
        // remove as child of parent
        bool success = removeNode.Parent.RemoveChild(removeNode);
        if (!success)
        {
            return false;
        }

        // remove node from tree
        success = nodes.Remove(removeNode);
        if (!success)
        {
            return false;
        }

        // check for branch node
        if (removeNode.Children.Count > 0)
        {
            // recursively prune subtree
            IList<TreeNode<T>> children = removeNode.Children;
            for (int i = children.Count - 1; i >= 0; i--)
            {
                RemoveNode(children[i]);
            }
```

```
            }

            return true;
        }
    }
```

The `RemoveNode` method removes the node with the given value from the tree. Because nodes can only have a single parent, the entire subtree rooted at the node being removed is pruned from the tree.

The if and else if clauses handle a couple of special cases, with the body of the else clause doing a "normal" node removal. Removing the node as a child of its parent and removing the node from the list of nodes for the tree should be easy to understand. The pruning code is more interesting, because it recursively prunes the subtree of the node being removed; this is the first time we've implemented recursion in one of our data structures. If you want to really understand how this method works, draw a tree on a piece of paper and "call" the `RemoveNode` method for one of your branch nodes.

We know the `Clear` method is O(n), so the if and else if clauses contribute O(n) to the complexity. The call to the `RemoveChild` method is O(n) in the worst case (where the root holds all the other nodes as children) and the call to the `List Remove` method is also O(n) because that method does a linear search for the element to remove. That means everything except for the pruning of the subtree is O(n).

What about the recursion from `RemoveNode` calling itself to prune the subtree? Instead of counting iterations for a loop, we need to count how many recursions we'd have in the worst case. In the worst case, every node in the tree has a single child, which really makes the tree into a doubly-linked list. If we're removing the node just below the root (remember, we handle removing the root as a special case), we'll recurse n - 2 times to prune the subtree rooted at the node just below the root.

Everything except the recursion is O(n); think of this as the body of a loop. We recurse n - 2 times in the worst case; think of this as the loop itself iterating n - 2 times. That makes the RemoveNode operation an $O(n^2)$ operation.

```
/// <summary>
/// Finds a tree node with the given value. If there
/// are multiple tree nodes with the given value the
/// method returns the first one it finds
/// </summary>
/// <param name="value">value to find</param>
/// <returns>tree node or null if not found</returns>
public TreeNode<T> Find(T value)
{
    foreach (TreeNode<T> node in nodes)
    {
        if (node.Value.Equals(value))
        {
            return node;
        }
```

```
            }
            return null;
        }
```

The `Find` method walks the list of nodes in the tree, returning the node with the given value if it finds it. If it doesn't find the value in the tree, it returns null. Because we allow duplicate values in our tree, the `Find` method returns the fist node it finds with the given value.

The loop executes n times, so Find is an O(n) operation where n is the number of nodes in the tree.

```
        /// <summary>
        /// Converts the tree to a comma-separated string of nodes
        /// </summary>
        /// <returns>comma-separated string of nodes</returns>
        public override String ToString()
        {
            StringBuilder builder = new StringBuilder();
            builder.Append("Root: ");
            if (root != null)
            {
                builder.Append(root.Value + " ");
            }
            else
            {
                builder.Append("null");
            }
            for (int i = 0; i < Count; i++)
            {
                builder.Append(nodes[i].ToString());
                if (i < Count - 1)
                {
                    builder.Append(",");
                }
            }
            return builder.ToString();
        }

        #endregion
    }
}
```

I implemented the `ToString` method to support the test cases, but being able to get a string representation of the tree could be useful in other scenarios as well.

The ToString operation is O(n). The for loop executes n times where n is the number of nodes in the tree and, inside the loop, the `TreeNode ToString` method is O(n) where n is the number of children the node has. Because each node can only have one parent, the total number of children in the graph is n - 1 (the root isn't a child). This is a similar analysis to our analysis of the first loop in the `Clear` method.

The `TestTree` class in the code accompanying this reading runs a variety of tests against the code above. Feel free to take a look at the test cases if you'd like.