6
Behavior Trees

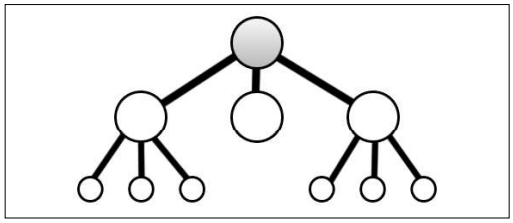
Behavior trees (**BTs**) have been gaining popularity among game developers very steadily. Over the last decade, BTs have become the pattern of choice for many AAA studios when it comes to implementing AI for their agents. Games like Halo and Gears of War are among the more famous franchises to make extensive use of BTs. An abundance of computing power in PCs, gaming consoles, and mobile devices has made them a good option for implementing AI in games of all types and scopes.

In this chapter, we will cover the following topics:

- The basics of a behavior tree
- The benefits of using existing behavior tree solutions
- How to implement our own behavior tree framework
- How to implement a basic tree using our framework

Learning the basics of behavior trees

It is called a tree because it is a hierarchical, branching system of nodes with a common parent, known as the root. As you've surely learned from reading this book, by now, behavior trees, too, mimic the real thing they are named after—in this case, trees. If we were to visualize a behavior tree, it would look something like the following figure:



A basic tree structure

Of course, behavior trees can be made up of any number of nodes and children nodes. The nodes at the very end of the hierarchy are referred to as leaf nodes, just like a tree. Nodes can represent behaviors or tests. Unlike state machines, which rely on transition rules to traverse through it, a BT's flow is defined strictly by each node's order within the larger hierarchy. A BT begins evaluating from the top (based on the preceding visualization) of the tree, then continues through each child, which, in turn, runs through each of its children until a condition is met or the leaf node is reached. BTs always begin evaluating from the root node.

Understanding different node types

The names of the different types of nodes may vary depending on who you ask, and even nodes themselves are sometimes referred to as tasks. While the complexity of a tree is dependent entirely upon the needs of the AI, the high-level concepts about how BTs work are fairly easy to understand if we look at each component individually. The following is true for each node regardless of what type of node we're referring to. A node will always return one of the following states:

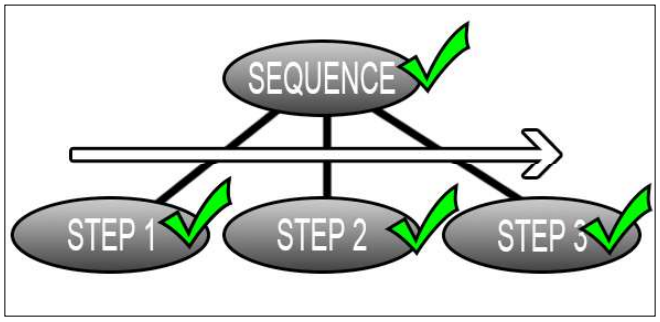
- **Success**: The condition the node was checking for has been met.
- **Failure**: The condition the node was checking for was not, and will not be met.
- **Running**: The validity of the condition the node is checking for has not been determined. Think of this as our "please wait" state.

Due to the potential complexity of a BT, most implementations are asynchronous, which, at least for Unity, means that evaluating a tree will not block the game from continuing other operations. The evaluation process of the various nodes in a BT can take several frames, if necessary. If you had to evaluate several trees on any number of agents at a time, you can imagine how it would negatively affect the performance of the program to have to wait for each of them to return a true or false to the root node. This is why the "running" state is important.

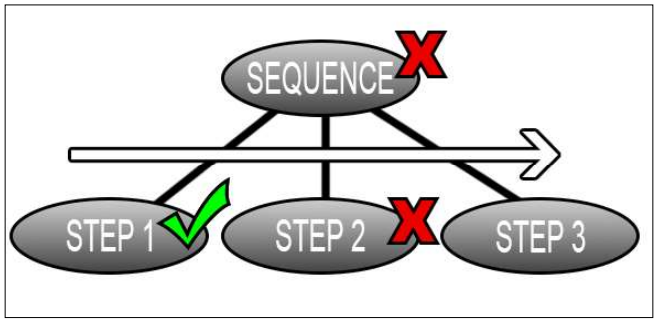
Defining composite nodes

Composite nodes are called so as they have one or more children. Their state is based entirely upon the result of evaluating its children, and while its children are being evaluated, it will be in a "running" state. There are a couple of composite node types, which are mostly defined by how their children are evaluated:

- **Sequences**: The defining characteristic of a sequence is that the entire sequence of children needs to complete successfully in order for it to evaluate as a success itself. If any of the children at any step of the sequence return false, the sequence itself will report a failure. It is important to note that, in general, sequences are executed from left to right. The following figures show a successful sequence and a failed sequence, respectively:



A successful sequence node



An unsuccessful sequence node

- **Selectors**: By comparison, selectors are much more forgiving parents to their children nodes. If any one of the children nodes in a selector sequence returns true, the selector says, "eh, good enough!" and returns true immediately, without evaluating any more of its children. The only way a selector node will return false is if all of its children are evaluated and none of them return a success.

Of course, each composite node type has its use depending on the situation. You can think of the different types of sequence nodes as "and" and "or" conditionals.

Understanding decorator nodes

The biggest difference between a composite node and a decorator node is that a decorator can have exactly one child and one child only. At first, this may seem unnecessary as you would, in theory, be able to get the same functionality by containing the condition in the node itself rather than relying on its child, but the decorator node is special in that it essentially takes the state returned by the child and evaluates the response based on its own parameters. A decorator can even specify how its children are evaluated and how often they are. These are some common decorator types:

- **Inverter**: Think of the inverter as a NOT modifier. It takes the opposite of the state returned by its child. For example, if the child returns TRUE, the decorator evaluates as FALSE, and vice versa. This is the equivalent of having the `!` operator in front of a Boolean in C#.
- **Repeater**: This repeats the evaluation of the child a specified (or infinite) number of times until it evaluates as either TRUE or FALSE as determined by the decorator. For example, you may want to wait indefinitely until a certain condition is met, such as "having enough energy" before a character uses an attack.
- **Limiter**: This simply limits the number of times a node will be evaluated to avoid getting an agent stuck in an awkward infinite behavior loop. This decorator, in contrast to the repeater, can be used to make sure a character only tries to, for example, kick the door open so many times before giving up and trying something else.

Some decorator nodes can be used for debugging and testing your trees. For example:

- **Fake state**: This always evaluates true or false as specified by the decorator. This is very helpful for asserting certain behavior in your agent. You can also have the decorator maintain a fake "running" state indefinitely to see how other agents around it will behave, for example.
- **Breakpoint**: Just like a breakpoint in code, you can have this node fire off logic to notify you via debug logs or other methods that the node has been reached.

These types are not monolithic archetypes that are mutually exclusive. You can combine these types of nodes to suit your needs. Just be careful not to combine too much functionality into one decorator to the point where it may be more efficient or convenient to use a sequence node instead.

Describing the leaf node

We briefly covered leaf nodes earlier in the chapter to make a point about the structure of a BT, but leaf nodes, in reality, can be just about any sort of behavior. They are magical in the sense that they can be used to describe any sort of logic your agent can have. A leaf node can specify a walk function, shoot command, or kick action. It doesn't matter what it does or how you decide to have it evaluate its states, it just has to be the last node in its own hierarchy and return any of the three states a node can return.

Evaluating the existing solutions

The unity asset store is an excellent resource for developers. Not only are you able to purchase art, audio, and other kinds of assets, but it is also populated with a large number of plugins and frameworks. Most relevant to our purposes, there are a number of behavior tree plugins available on the asset store, ranging from free to a few hundred dollars. Most, if not all, provide some sort of GUI to make visualizing and arranging a fairly painless experience.

There are many advantages of going with an off-the-shelf solution from the asset store. Many of the frameworks include advanced functionality such as runtime (and often visual) debugging, robust APIs, serialization, and data-oriented tree support. Many even include sample leaf logic nodes to use in your game, minimizing the amount of coding you have to do to get up and running.

The previous edition of this book, *Unity 4.x Game AI Programming*, focused on developer AngryAnt's Behave plugin, which is currently available as Behave 2 for Unity on the asset store as a paid plugin, which continues to be an excellent choice for your behavior tree needs (and so much more). It is a very robust, performant, and excellently designed framework.

Some other alternatives are Behavior Machine and Behavior Designer, which offer different pricing tiers (Behavior Machine even offers a free edition) and a wide array of useful features. Many other options can be found for free around the Web as both generic C# and Unity-specific implementations. Ultimately, as with any other system, the choice of rolling your own or using an existing solution will depend on your time, budget, and project.

Implementing a basic behavior tree framework

While a fully-fledged implementation of a behavior tree with a GUI and its many node types and variations is outside the scope of this book, we can certainly focus on the core principles to get a solid grasp on what the concepts we've covered in this chapter look similar to in action. Provided with this chapter is the basic framework for a behavior tree. Our example will focus on simple logic to highlight the functionality of the tree rather than muddy up the example with complex game logic. The goal of our example is to make you feel comfortable with what can seem like an intimidating concept in game AI, and give you the necessary tools to build your own tree and expand upon the provided code if you do so.

Implementing a base Node class

There is a base functionality that needs to go into every node. Our simple framework will have all the nodes derived from a base abstract `Node.cs` class. This class will provide said base functionality or at least the signature to expand upon that functionality:

```
using UnityEngine;
using System.Collections;

[System.Serializable]
public abstract class Node {

    /* Delegate that returns the state of the node.*/
    public delegate NodeStates NodeReturn();

    /* The current state of the node */
    protected NodeStates m_nodeState;

    public NodeStates nodeState {
        get { return m_nodeState; }
    }

    /* The constructor for the node */
    public Node() {}

    /* Implementing classes use this method to evaluate the desired
set of conditions */
    public abstract NodeStates Evaluate();

}
```

The class is fairly simple. Think of `Node.cs` as a blueprint for all the other node types to be built upon. We begin with the `NodeReturn` delegate, which is not implemented in our example, but the next two fields are. However, `m_nodeState` is the state of a node at any given point. As we learned earlier, it will be either `FAILURE`, `SUCCESS`, or `RUNNING`. The `nodeState` value is simply a getter for `m_nodeState` since it is protected and we don't want any other area of the code directly setting `m_nodeState` inadvertently.

Next, we have an empty constructor, for the sake of being explicit, even though it is not being used. Lastly, we have the meat and potatoes of our `Node.cs` class—the `Evaluate()` method. As we'll see in the classes that implement `Node.cs`, `Evaluate` is where the magic happens. It runs the code that determines the state of the node.

Extending nodes to selectors

To create a selector, we simply expand upon the functionality that we described in the `Node.cs` class:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Selector : Node {
    /** The child nodes for this selector */
    protected List<Node> m_nodes = new List<Node>();


    /** The constructor requires a list of child nodes to be
     * passed in*/
    public Selector(List<Node> nodes) {
        m_nodes = nodes;
    }

    /* If any of the children reports a success, the selector will
     * immediately report a success upwards. If all children fail,
     * it will report a failure instead.*/
    public override NodeStates Evaluate() {
        foreach (Node node in m_nodes) {
            switch (node.Evaluate()) {
                case NodeStates.FAILURE:
                    continue;
                case NodeStates.SUCCESS:
                    m_nodeState = NodeStates.SUCCESS;
                    return m_nodeState;
```

```
                    case NodeStates.RUNNING:
                        m_nodeState = NodeStates.RUNNING;
                        return m_nodeState;
                    default:
                        continue;
                }
            }
            m_nodeState = NodeStates.FAILURE;
            return m_nodeState;
        }
    }
```

As we learned earlier in the chapter, selectors are composite nodes; this means that they have one or more child nodes. These child nodes are stored in the `m_nodes List<Node>` variable. Though it's conceivable that one could extend the functionality of this class to allow adding more child nodes after the class has been instantiated, we initially provide this list via the constructor.

The next portion of the code is a bit more interesting as it shows us a real implementation of the concepts we learned earlier. The `Evaluate()` method runs through all of its child nodes and evaluates each one individually. As a failure doesn't necessarily mean a failure for the entire selector, if one of the children returns `FAILURE`, we simply continue onto the next one. Inversely, if any child returns `SUCCESS`, then we're all set—we can set this node's state accordingly and return that value. If we make it through the entire list of child nodes and none of them have returned `SUCCESS`, then we can essentially determine that the entire selector has failed and we assign and return a `FAILURE` state.

Moving on to sequences

Sequences are very similar in their implementation, but as you might have guessed by now, the `Evaluate()` method behaves differently:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Sequence : Node {
    /** Chiildren nodes that belong to this sequence */
    private List<Node> m_nodes = new List<Node>();

    /** Must provide an initial set of children nodes to work */
    public Sequence(List<Node> nodes) {
        m_nodes = nodes;
```

```
    }

    /* If any child node returns a failure, the entire node fails.
Whence all
     * nodes return a success, the node reports a success. */
    public override NodeStates Evaluate() {
        bool anyChildRunning = false;

        foreach(Node node in m_nodes) {
            switch (node.Evaluate()) {
                case NodeStates.FAILURE:
                    m_nodeState = NodeStates.FAILURE;
                    return m_nodeState;
                case NodeStates.SUCCESS:
                    continue;
                case NodeStates.RUNNING:
                    anyChildRunning = true;
                    continue;
                default:
                    m_nodeState = NodeStates.SUCCESS;
                    return m_nodeState;
            }
        }
        m_nodeState = anyChildRunning ? NodeStates.RUNNING :
NodeStates.SUCCESS;
        return m_nodeState;
    }
}
```

The `Evaluate()` method in a sequence will need to return true for all the child nodes, and if any one of them fails during the process, the entire sequence fails, which is why we check for `FAILURE` first and set and report it accordingly. A `SUCCESS` state simply means we get to live to fight another day, and we continue onto the next child node. If any of the child nodes are determined to be in the `RUNNING` state, we report that as the state for the node and then the parent node or the logic driving the entire tree can re-evaluate it again.

Implementing a decorator as an inverter

The structure of `Inverter.cs` is a bit different, but it derives from `Node`, just like the rest of the nodes. Let's take a look at the code and spot the differences:

```
using UnityEngine;
using System.Collections;

public class Inverter : Node {
    /* Child node to evaluate */
    private Node m_node;

    public Node node {
        get { return m_node; }
    }

    /* The constructor requires the child node that this inverter
decorator
     * wraps*/
    public Inverter(Node node) {
        m_node = node;
    }

    /* Reports a success if the child fails and
     * a failure if the child succeeds. Running will report
     * as running */
    public override NodeStates Evaluate() {
        switch (m_node.Evaluate()) {
            case NodeStates.FAILURE:
                m_nodeState = NodeStates.SUCCESS;
                return m_nodeState;
            case NodeStates.SUCCESS:
                m_nodeState = NodeStates.FAILURE;
                return m_nodeState;
            case NodeStates.RUNNING:
                m_nodeState = NodeStates.RUNNING;
                return m_nodeState;
        }
        m_nodeState = NodeStates.SUCCESS;
        return m_nodeState;
    }
}
```

As you can see, since a decorator only has one child, we don't have `List<Node>`, but rather a single node variable, `m_node`. We pass this node in via the constructor (essentially requiring it), but there is no reason you couldn't modify this code to provide an empty constructor and a method to assign the child node after instantiation.

The `Evalute()` implementation implements the behavior of an inverter that we described earlier in the chapter—when the child evaluates as `SUCCESS`, the inverter reports a `FAILURE`, and when the child evaluates as `FAILURE`, the inverter reports a `SUCCESS`. The `RUNNING` state is reported normally.

Creating a generic action node

Now we arrive at `ActionNode.cs`, which is a generic leaf node to pass in some logic via a delegate. You are free to implement leaf nodes in any way that fits your logic, as long as it derives from `Node`. This particular example is equal parts flexible and restrictive. It's flexible in the sense that it allows you to pass in any method matching the delegate signature, but is restrictive for this very reason—it only provides one delegate signature that doesn't take in any arguments:

```
using System;
using UnityEngine;
using System.Collections;

public class ActionNode : Node {
    /* Method signature for the action. */
    public delegate NodeStates ActionNodeDelegate();

    /* The delegate that is called to evaluate this node */
    private ActionNodeDelegate m_action;

    /* Because this node contains no logic itself,
     * the logic must be passed in in the form of
     * a delegate. As the signature states, the action
     * needs to return a NodeStates enum */
    public ActionNode(ActionNodeDelegate action) {
        m_action = action;
    }

    /* Evaluates the node using the passed in delegate and
     * reports the resulting state as appropriate */
    public override NodeStates Evaluate() {
        switch (m_action()) {
            case NodeStates.SUCCESS:
                m_nodeState = NodeStates.SUCCESS;
```

```
                return m_nodeState;
            case NodeStates.FAILURE:
                m_nodeState = NodeStates.FAILURE;
                return m_nodeState;
            case NodeStates.RUNNING:
                m_nodeState = NodeStates.RUNNING;
                return m_nodeState;
            default:
                m_nodeState = NodeStates.FAILURE;
                return m_nodeState;
        }
    }
}
```

The key for making this node work is the `m_action` delegate. For those familiar with C++, a delegate in C# can be thought of as a function pointer of sorts. You can also think of a delegate as a variable containing (or more accurately, pointing to) a function. This allows you to set the function to be called at runtime. The constructor requires you to pass in a method matching its signature, and is expecting that method to return a `NodeStates` enum. That method can implement any logic you want as long as these conditions are meant. Unlike other nodes we've implemented, this one doesn't fall through to any state outside of the switch itself, so it defaults to a `FAILURE` state. You may choose to default to a `SUCCESS` or `RUNNING` state, if you so wish, by modifying the default return.

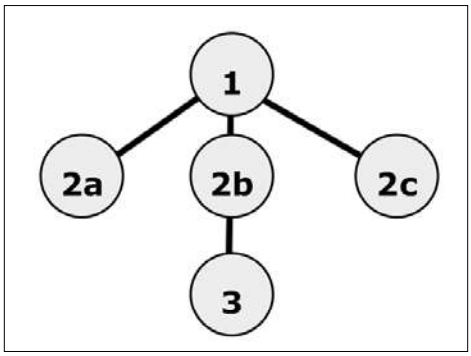
You can easily expand on this class by deriving from it or simply making the changes to it that you need. You can also skip this generic action node altogether and implement one-off versions of specific leaf nodes, but it's good practice to reuse as much code as possible. Just remember to derive from `Node` and implement the required code!

Testing our framework

The framework that we just reviewed is nothing more than this. It provides us with all the functionality we need to make a tree, but we have to make the actual tree ourselves. For the purposes of this book, a somewhat manually constructed tree is provided.

Planning ahead

Before we set up our tree, let's look at what we're trying to accomplish. It is often helpful to visualize a tree before implementing it. Our tree will count up from zero to a specified value. Along the way, it will check whether certain conditions are met for that value and report its state accordingly. The following diagram illustrates the basic hierarchy for our tree:



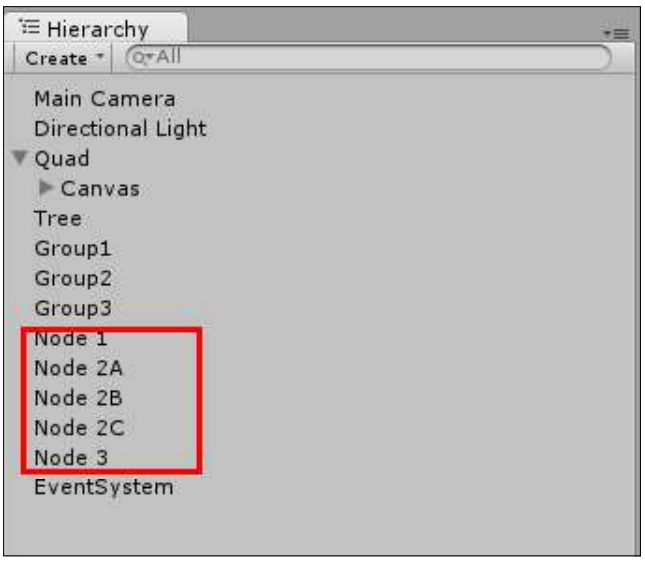
For our tests, we will use a three-tier tree, including the root node:

- **Node 1**: This is our root node. It has children, and we want to be able to return a success if any of the children is a success, so we'll implement it as a selector.
- **Node 2a**: We'll implement this node using an `ActionNode`.
- **Node 2b**: We'll use this node to demonstrate how our inverter works.
- **Node 2c**: We'll run the same `ActionNode` from node 2a again, and see how that affects our tree's evaluation.
- **Node 3**: Node 3 happens to be the lone node in the third tier of the tree. It is the child of the 2b decorator node. This means that if it reports `SUCCESS`, 2b will report a `FAILURE`, and vice versa.

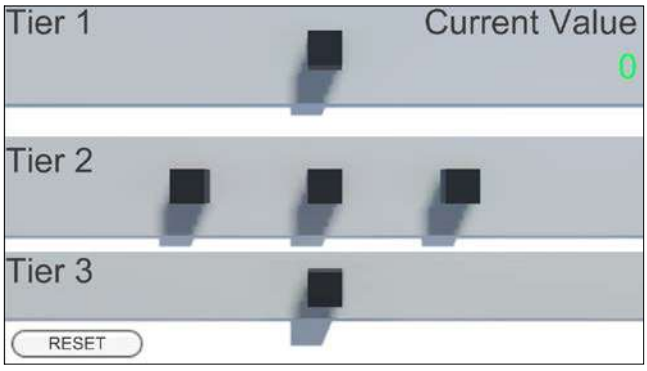
At this point, we're still a bit vague on the implementation details, but the preceding diagram will help us to visualize our tree as we implement it in code. Keep it handy for reference as we go through the code.

Examining our scene setup

We've now looked at the basic structure of our tree, and before we jump in and dig into the actual code implementation, let's look at our scene setup. The following screenshot shows our hierarchy; the nodes are highlighted for emphasis:



The setup is quite simple. There is a quad with a world-space canvas, which is simply to display some information during the test. The nodes highlighted in the preceding screenshot will be referenced in the code later, and we'll be using them to visualize the status of each individual node. The actual scene looks something like the following screenshot:



Our actual layout mimics the diagram we created earlier

As you can see, we have one node or box representing each one of the nodes that we laid out in our planning phase. These are referenced in the actual test code and will be changing colors according to the state that is returned.

Exploring the MathTree code

Without further ado, let's have a look at the code driving our test. This is `MathTree.cs`:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System.Collections.Generic;

public class MathTree : MonoBehaviour {
    public Color m_evaluating;
    public Color m_succeeded;
    public Color m_failed;

    public Selector m_rootNode;

    public ActionNode m_node2A;
    public Inverter m_node2B;
    public ActionNode m_node2C;
    public ActionNode m_node3;

    public GameObject m_rootNodeBox;
    public GameObject m_node2aBox;
    public GameObject m_node2bBox;
    public GameObject m_node2cBox;
    public GameObject m_node3Box;

    public  int m_targetValue = 20;
    private int m_currentValue = 0;

    [SerializeField]
    private Text m_valueLabel;
```

The first few variables are simply used for debugging. The three color variables are the colors we'll be assigning to our node boxes to visualize their state. By default, `RUNNING` is yellow, `SUCCESS` is green, and `FAILED` is red. This is pretty standard stuff; let's move along.

We then declare our actual nodes. As you can see, `m_rootNode` is a selector as we mentioned earlier. Notice that we do not assign any of the node variables yet, since we have to pass in some data to their constructors.

We then have the references to the boxes we saw in our scene. These are just GameObjects that we drag-and-drop into the inspector (we'll have a look at that after we inspect the code).

We then have a couple of int values, which will make more sense as we look at the logic, so we'll skip over these. Lastly, we have a unity UI Text variable that will display some values for us during the test.

Let's get into the initialization of our actual nodes:

```
    /* We instantiate our nodes from the bottom up, and assign the
children
     * in that order */
   void Start () {
        /** The deepest-level node is Node 3, which has no children.
*/
        m_node3 = new ActionNode(NotEqualToTarget);

        /** Next up, we create the level 2 nodes. */
        m_node2A = new ActionNode(AddTen);

        /** Node 2B is a selector which has node 3 as a child, so
we'll pass
         * node 3 to the constructor */
        m_node2B = new Inverter(m_node3);

        m_node2C = new ActionNode(AddTen);

        /** Lastly, we have our root node. First, we prepare our list
of children
         * nodes to pass in */
        List<Node> rootChildren = new List<Node>();
        rootChildren.Add(m_node2A);
        rootChildren.Add(m_node2B);
        rootChildren.Add(m_node2C);

        /** Then we create our root node object and pass in the list
*/
        m_rootNode = new Selector(rootChildren);

        m_valueLabel.text = m_currentValue.ToString();

        m_rootNode.Evaluate();

        UpdateBoxes();
   }
```

For the sake of organization, we declare our nodes from the bottom of the tree to the top of the tree, or the root node. We do this because we cannot instantiate a parent without passing in its child nodes, so we have to instantiate the child nodes first. Notice that `m_node2A`, `m_node2C`, and `m_node3` are action nodes, so we pass in delegates (we'll look at these methods next). Then, `m_node2B`, being a selector, takes in a node as a child, in this case, `m_node3`. After we've declared these tiers, we throw all the tier 2 nodes into a list because our tier 1 node, the root node, is a selector that requires a list of children to be instantiated.

After we've instantiated all of our nodes, we kick off the process and begin evaluating our root node using its `Evaluate()` method. The `UpdateBoxes()` method simply updates the box game objects that we declared earlier with the appropriate colors; we'll look at that up ahead in this section:

```
private void UpdateBoxes() {
        /** Update root node box */
        if (m_rootNode.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_rootNodeBox);
        } else if (m_rootNode.nodeState == NodeStates.FAILURE) {
            SetFailed(m_rootNodeBox);
        }

        /** Update 2A node box */
        if (m_node2A.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_node2aBox);
        } else if (m_node2A.nodeState == NodeStates.FAILURE) {
            SetFailed(m_node2aBox);
        }

        /** Update 2B node box */
        if (m_node2B.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_node2bBox);
        } else if (m_node2B.nodeState == NodeStates.FAILURE) {
            SetFailed(m_node2bBox);
        }

        /** Update 2C node box */
        if (m_node2C.nodeState == NodeStates.SUCCESS) {
            SetSucceeded(m_node2cBox);
        } else if (m_node2C.nodeState == NodeStates.FAILURE) {
            SetFailed(m_node2cBox);
        }

        /** Update 3 node box */
```

```
            if (m_node3.nodeState == NodeStates.SUCCESS) {
                SetSucceeded(m_node3Box);
            } else if (m_node3.nodeState == NodeStates.FAILURE) {
                SetFailed(m_node3Box);
            }
        }
```

There is not a whole lot to discuss here. Do notice that because we set this tree up manually, we check each node individually and get its `nodeState` and set the colors using the `SetSucceeded` and `SetFailed` methods. Let's move on to the meaty part of the class:

```
    private NodeStates NotEqualToTarget() {
            if (m_currentValue != m_targetValue) {
                return NodeStates.SUCCESS;
            } else {
                return NodeStates.FAILURE;
            }
        }

        private NodeStates AddTen() {
            m_currentValue += 10;
            m_valueLabel.text = m_currentValue.ToString();
            if (m_currentValue == m_targetValue) {
                return NodeStates.SUCCESS;
            } else {
                return NodeStates.FAILURE;
            }
        }
```

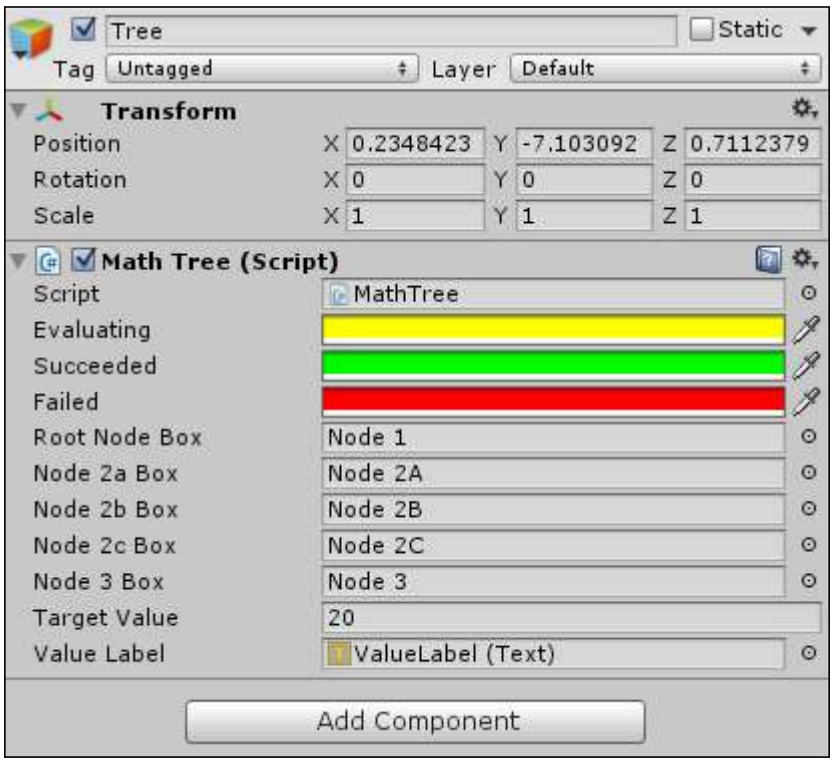
First, we have `NotEqualToTarget()`, which is the method we passed into our decorator's child action node. We're essentially setting ourselves up for a double negative here, so try to follow along. This method returns a success if the current value is *not* equal to the target value, and returns false otherwise. The parent inverter decorator will then evaluate to the opposite of what this node returns. So, if the value is not equal, the inverter node will fail; otherwise, it will succeed. If you're feeling a bit lost at this point, don't worry. It will all make sense when we see this in action.

The next method is the `AddTen()` method, which is the method passed into our other two action nodes. It does exactly what the name implies—it adds 10 to our `m_currentValue` variable, then checks if it's equal to our `m_targetValue`, and evaluates as `SUCCESS` if so, and `FAILURE`, if not.

The last few methods are self-explanatory so we will not go over them.

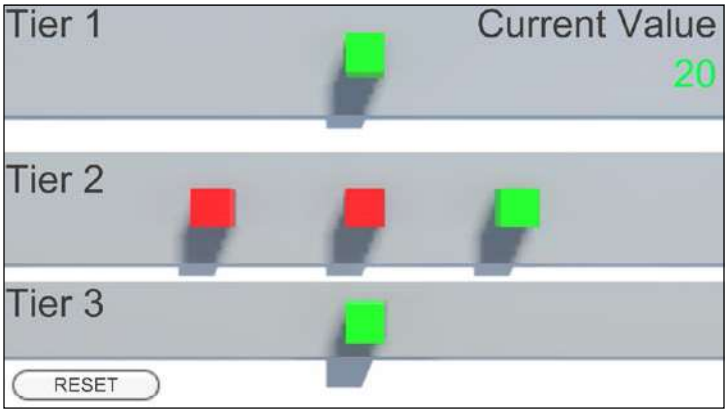
Executing the test

Now that we have a pretty good idea of how the code works, let's see it in action. First thing first, however. Let's make sure our component is properly setup. Select the **Tree** game object from the hierarchy, and its inspector should look similar to this:



The default settings for the component

As you can see, the state colors and box references have already been assigned for you, as well as the `m_valueLabel` variable. The `m_targetValue` variable has also been assigned for you via code. Make sure to leave it at (or set it to) 20 before you hit play. Play the scene, and you'll see your boxes lit up, as shown in the following screenshot:



The boxes lit up, indicating the result of each node's evaluation

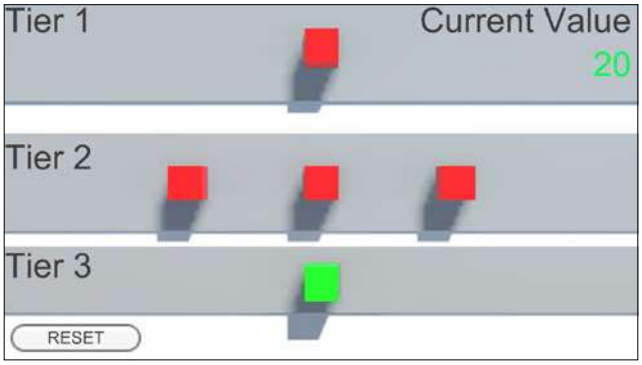
As we can see, our root node evaluated to `SUCCESS`, which is what we intended, but let's examine why, one step at a time, starting at tier 2:

- **Node 2A**: We started with `m_currentValue` at `0`, so upon adding `10` to it, it's still not equal to our `m_targetValue` (`20`) and it fails. Thus, it is red.
- **Node 2B**: As it evaluates its child, once again, `m_currentValue` and `m_targetValue` are not equal. This returns `SUCCESS`. Then, the inverter logic kicks in and reverses this response so that it reports `FAILURE` for itself. So, we move on to the last node.
- **Node 2C**: Once again, we add `10` to `m_currentValue`. It then becomes `20`, which is equal to `m_targetValue`, and evaluates as `SUCCESS`, so our root node is successful as result.

The test is simple, but it illustrates the concepts clearly. Before we consider the test a success, let's run it one more time, but change `m_targetValue` first. Set it to `30` in the inspector, as shown in the following screenshot:

The updated value is highlighted

A small change to be sure, but it will change how the entire tree evaluates. Play the scene again, and we will end up with the set of nodes lit up, as shown in the following screenshot:



A clearly different from our first test

As you can see, all but one of the child nodes of our root failed, so it reports `FAILURE` for itself. Let's look at why:

- **Node 2A**: Nothing really changes here from our original example. Our `m_currentValue` variable starts at `0` and ends up at `10`, which is not equal to our `m_targetValue` of `30`, so it fails.
- **Node 2B**: This evaluates its child once more, and because the child node reports `SUCCESS`, it reports `FAILURE` for itself, and we move on to the next node.
- **Node 2C**: Once again, we add `10` to our `m_currentValue` variable, adding up to `20`, which, after having changed the `m_targetValue` variable, no longer evaluates to `SUCCESS`.

The current implementation of the nodes will have unevaluated nodes default to `SUCCESS`. This is because of our enum order, as you can see in `NodeState.cs`:

```
public enum NodeStates {
    SUCCESS,
    FAILURE,
    RUNNING,
}
```

In our enum, `SUCCESS` is the first enumeration, so if a node never gets evaluated, the default value is never changed. If you were to change the `m_targetValue` variable to `10`, for example, all the nodes would light up to green. This is simply a by-product of our test implementation and doesn't actually reflect any design issues with our nodes. Our `UpdateBoxes()` method updates all the boxes whether they were evaluated or not. In this example, node 2A would immediately evaluate as `SUCCESS`, which, in turn, would cause the root node to report `SUCCESS`, and neither nodes 2B, 2C, nor 3 would be evaluated at all, having no effect on the evaluation of the tree as a whole.

You are highly encouraged to play with this test. Change the root node implementation from a selector to a sequence, for example. By simply changing `public Selector m_rootNode;` to `public Sequence m_rootNode;` and `m_rootNode = new Selector(rootChildren);` to `m_rootNode = new Sequence(rootChildren);`, you can test a completely different set of functionality.

Summary

In this chapter, we dug in to how a behavior tree works and then we looked at each individual type of node that can make up a behavior tree. We also learned the different scenarios where some nodes would be more helpful than others. After looking at some off-the-shelf solutions available on the Unity asset store, we applied this knowledge by implementing our own basic behavior tree framework in C# and explored the inner workings. With the knowledge and the tools out of the way, we created a sample behavior tree using our framework to test the concepts learned throughout the chapter. This knowledge prepares us to harness the power of behavior trees in games and take our AI implementations to the next level.

In the next chapter, *Chapter 7*, *Using Fuzzy Logic to Make Your AI Seem Alive*, we'll look at new ways to add complexity and functionality to the concepts we've learned in this chapter, modifying behavior trees, and FSMs, which we covered in *Chapter 2*, *Finite State Machines and You*, via the concept of fuzzy logic.