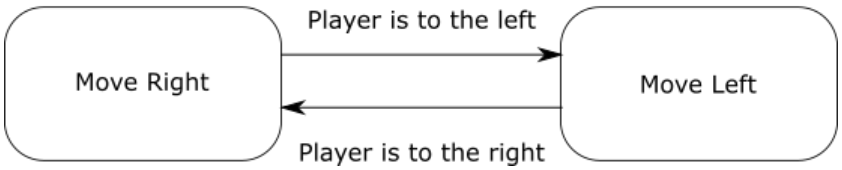
AI - Behaviour Trees

Introduction

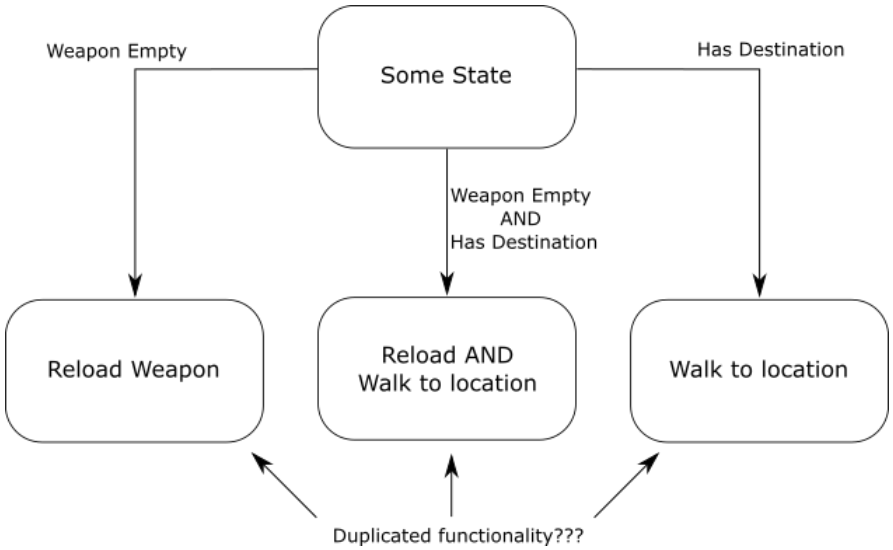
As an AI agent engages with its gameplay world, it must make decisions and take actions that should ultimately come together to form a cohesive, believable behaviour. Some of these actions will be singular states, while others need to be brought together into a series of events that must be completed one after the other. For example a simple AI in a 2D game might simply need to walk either left or right depending on the relative location of the player:



There's not much more to add for very basic AI. A more complex action might be to enter a restricted area of the level, requiring a keycard:

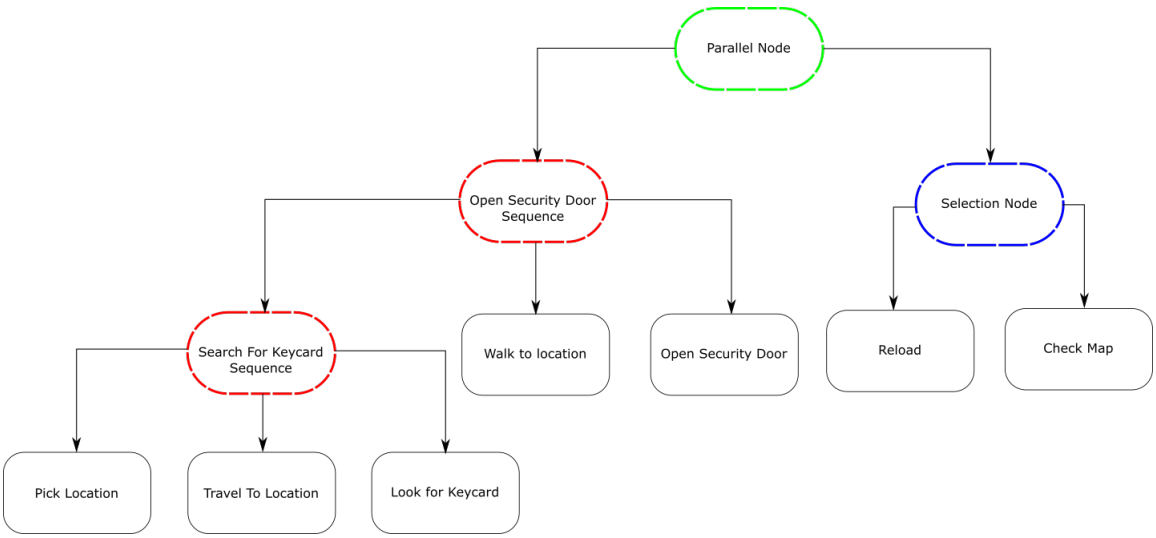
If any part of that series of actions cannot be completed, then the chain of events is broken, and the further states cannot be attempted. While these can be implemented using a state machine, a basic state machine doesn't really make clear the relative ordering of the events, or that they are all intrinsically chained together (while they are all each separate actions the AI must do, they must all be present, and ran in a specific order, for them to make any sense). A *hierarchical* state machine helps with this to some extent, as the related actions can be brought together into their own superstate.

There's a potential problem even with hierarchical state machines. What if, as the AI agent is going to find the keycard and enter the restricted area, they also might need to reload their weapon? If they're a well-trained enough bad guy they can do that while walking, but if they only need to *possibly* reload, how do we fully represent this in a state machine? There's two 'states' they are in simultaneously (walking and reloading), but since our classic state machine concept can only be in one state at a time, we end up having to duplicate some logic to produce the desired result.



Behaviour Trees

One potential solution to the problems described above is to rethink what a node in a state machine actually is. Instead of each node being a singular item of logic that makes the AI 'think', we could instead say that some nodes indicate that one or more transitions from it should be enacted immediately, with multiple nodes then being processed per-frame, rather than just one. This leads to a *tree* of nodes rather than just a graph, with the nodes at leaves known as *behaviours*. Here's an example of a simple behaviour tree:

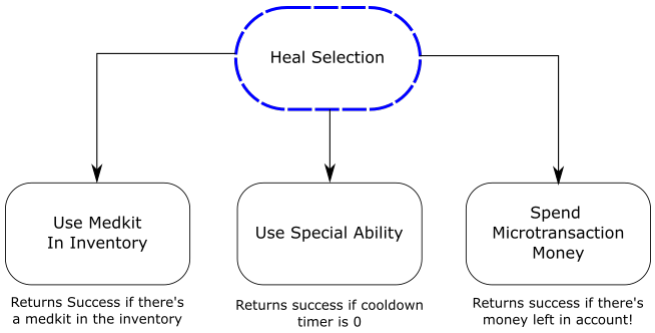


As you can probably tell, it replicates the walking and reloading described earlier. What makes a behaviour tree different than a state machine is that instead of simple transitions connecting each node, we have specialised nodes that operate upon their children in different ways. Instead of a single state machine node being updated, and then transitions checked for being able to move to a new state, in a behaviour tree the entire tree is evaluated, starting from the root. At the leaves of the tree are specific actions that the AI can attempt to enact; you can think of these as being the states of our state machine. The rest of the tree replaces our idea of a transition to and from a single state. These nodes make decisions over which nodes further down the tree (that is, their *child* nodes) should or should not be processed that particular frame. In this case, if the AI for whatever reason didn't need to open a security door, the left side of the tree would not need any further processing, as the child nodes should all in some way be supporting the logic required for the AI to complete that task.

What is common to each of these nodes is that they can each be in one of three main states - either they are **successful** in their operation, they have **failed** in their operation, or they are still **ongoing** in trying to complete that operation. For example, it might take some time for the character to finish reloading, and so for the duration of that period, the 'reload weapon' behaviour might be in progress; once finished it flips from ongoing to completed, or if for some reason it cannot be completed (perhaps there's no ammo left?) the state can move into a failed state. Unlike in a normal state machine, the AI discovering that it has no ammo left isn't the end of its state - it might decide to do something else, and this can be represented by specific node types within the tree reacting to success or failure messages from states in different ways.

Selection Nodes

If an AI has a choice of actions it could undertake, there must be some way of choosing which one is the best choice for a given situation. Behaviour trees encapsulate this logic in a few different ways. Perhaps the most simple is as a *selection*. A selection node will scan across its child nodes from left to right, and execute each in turn. If the child node reports success, or that it can continue its task, the selector node ceases scanning across the child nodes and returns that it has succeeded in finding a selection to run. If the child node reports failure, the selector node moves on, until eventually it finds a child node that *can* be executed without failure, or else it will also return failure, indicating that none of the selections beneath it can be processed at all. An example of a selector in action could be like this:



If the AI needs to heal in some way, it stands to reason that it needs to choose the best way to do so. If we assume that each of the ways outlined above will heal the AI back to full health, then it would seem sensible to not consider any of the other methods once a way to heal has been found. However, if the AI's inventory is empty, and so none of the methods can succeed, then the 'Heal' selector should return failure, indicating to the rest of the behaviour tree that it cannot heal, which may influence further choices (running away would seem a good idea!).

Sequence Nodes

An AI might sometimes need to perform an ordered set of operations - you can't make a cup of tea without first putting water in the kettle and boiling it! In a behaviour tree, this is known as a *sequence* of actions, as can be represented as a sequence node and its child nodes. When executed, a sequence node will scan from left to right across its children, attempting to execute the logic of each one in turn. Unlike a selection node, if a particular child node cannot currently complete, then the sequence stops immediately, and returns that child node's state - either *failure* or *ongoing*. This prevents the rest of the child nodes from executing, as there's presumably no benefit in doing so (there's nothing *stopping* you from pouring an empty kettle into a cup, but not much good will come of it!).

Parallel Nodes

Sometimes we might want an AI to do multiple things simultaneously. In the above example, the character can both search for their access key while reloading their weapon, should they need to. Selectors and sequences aren't suitable for this, as they'll only perform *one* action (although they use different rules to determine which). To remedy this, a *parallel* node can be used, which will execute *all* of its child nodes, until they are all complete. Despite the name, this execution probably *isn't* truly parallel across multiple *threads* of execution, but they will all be ran before another node is considered by the behaviour tree.

Action Nodes

The previously defined nodes all help define the more abstract decisions that an AI agent must make, but eventually, we are going to need some game-specific code to enact. The leaves of the behaviour tree define these specific actions that an AI must do - flicking a switch, walking to a location, or firing at a specific opponent within the world are examples of such actions, that the rest of the behaviour tree reasons about to choose the best one. Some of these actions may be provided by the core game engine, or make use of core engine functionality - a walking to a location action might begin with a call to the pathfinding code to determine the best way to get to that position, which will then be followed in further executions of the node. Other actions will be specific to the game being developed - a node to activate a powerup is an example of something that an engine can't really provide, and so a good behaviour tree system will provide a means by which such custom actions can be defined.

Decorator Nodes

Occasionally, it can be beneficial to modify the return state of a node. For example, consider an action that has been coded to return *success* if there is a player nearby. If at some other part in the tree we need to perform an action when there is *not* a player nearby, we'd need to create a new action, that performs the opposite task. In this example, that probably wouldn't be too much work, but it's still an example of something that we should try and avoid - more code means more room for errors! In behaviour trees, we can *decorate* a node, by adding in a node that receives the state of its child, and then returns a different state based on this. A simple example of such a decorator is an *inverter* - when executed, it'll execute a child node (we assume that a decorator has only a single child node in the tree), and receive its execution state; however, if this returned state is *Success*, the inverter will return *Failure*, and vice-versa. This allows us to flip the logic of an action node without having to specifically code more action logic. Another example could be a node which checks some external state upon execution, and only allows its child to be executed if this external state matches a given criteria. This would allow some basic conditional logic to be created without needing more action nodes to check for each potential value in the game.

Behaviour Tree Considerations

The key element that makes a behaviour tree different than a standard state machine is the ability to provide additional context to the connections between the actions the AI can run, and the ability for a tree to execute multiple actions simultaneously in a single frame. It also helps promote code reuse within the game itself, as an action node *could* be a child of multiple parents, and can be combined together in a variety of different ways using sequences.

Over the course of an AI agent's lifetime, it may well have to perform multiple actions, and perform each of those actions multiple times. For this reason, it's also common to have some means by which some subset of the tree can be 'reset' and allow it to be ran again, which could either be implemented as an extra 'Initialisation' state for a node to be in along with success or failure, or by having the leaf node actions keep track of additional variables that are changed externally to the behaviour tree itself.

A standard behaviour tree will execute the *entire* tree each frame to find the correct actions to execute. A large tree could contain 10s of nodes, making this quite a slow process. Some behaviour trees will therefore allow nodes to register 'events' such that only if a particular variable has changed (or some other action external to the tree has occurred) should a particular subset of the tree be re-evaluated; otherwise, only the leaf nodes that were executed in the previous frame are ran again. Doing this can massively reduce the computation required to run a behaviour tree, and should result in the same result, as the behaviour tree nodes being executed should only change if a particular subset of the game's values do.

Tutorial Code

To see how a behaviour tree might be helpful in creating more intelligent AI, we're going to implement a basic example of one, that performs a simple set of actions. Just as with the state machine and pushdown automata, we're going to represent our behaviour tree using a set of classes. This time around, we're going to use a combination of **virtual** functions and **std::Functions** to create the behaviour - the conditional logic of the tree is ideal for implementation via **virtual** functions, while at the leaf nodes, where the game-specific actions take place, we can inject custom functionality into the tree via a **std::Function**.

BehaviourNode Class

At its most basic, a behaviour tree node is something that can be executed, and which can return whether this execution has completed or not. To represent this execution state, we start with an **enum** that can represent *failure*, *success*, or whether the node is still *ongoing* in its execution. In order to allow our tree to be reset, and to allow nodes to make decisions based upon this, we're also going to give nodes an initial behaviour state of *initialise*. As a node could do a number of different things, this **BehaviourNode** base class is abstract, having a **pure virtual** *Execute* method that takes in a timestep, and returns the latest **BehaviourState**. To make it a little easier to debug, we'll also give each node a name as a string. Beyond this, there's not much else we can say about a behaviour node in our tree, so that's all for our initial class.

```
1  #pragma once
2  #include <string>
3
4  enum BehaviourState {
5     Initialise,
6     Failure,
7     Success,
8     Ongoing
9  };
10
11 class BehaviourNode  {
12    public:
13    BehaviourNode(const std::string &nodeName)   {
14       currentState   = Initialise;
15       name           = nodeName;
16    }
17    virtual ~BehaviourNode() {}
18    virtual BehaviourState Execute(float dt) = 0;
19
20    virtual void   Reset()         {currentState = Initialise; }
21    protected:
22    BehaviourState currentState;
23    std::string    name;
24 };
```

BehaviourNode class header

BehaviourNodeWithChildren Class

Some of the specialised node types in our behaviour tree may have a number of children that can in some way be iterated over. As not all node types should have this property, we don't have it in the abstract base class, but instead make a subclass of our newly created **BehaviourNode**, that additionally contains a **std::vector** of child nodes, along with the method to add nodes to it. Here we can also see why the Reset method (that should return the tree to a default state) is **virtual**, as in this case, the subclass must tell all of its child nodes that they too should be reset. In this case, the **BehaviourNodeWithChildren** class **shouldn't** be able to be instantiated on its own, so it doesn't define an **Execute** method.

```
1  #pragma once
2  #include "BehaviourNode.h"
3  #include <vector>
4
5  class BehaviourNodeWithChildren : public BehaviourNode   {
6     public:
7     BehaviourNodeWithChildren(const std::string& nodeName)
8        : BehaviourNode(nodeName){};
9
10    ~BehaviourNodeWithChildren() {
11       for (auto& i : childNodes) {
12          delete i;
13       }
14    }
15    void AddChild(BehaviourNode* n) {
16       childNodes.emplace_back(n);
17    }
18
19    void Reset() override {
20       currentState = Initialise;
21       for (auto& i : childNodes) {
22          i->Reset();
23       }
24    }
25    protected:
26    std::vector<BehaviourNode*> childNodes;
27 };
```

BehaviourNodeWithChildren class header

BehaviourSelector Class

A selector node picks from a series of child nodes, and tries to run their logic. Therefore, our class to represent this, **BehaviourSelector**, inherits from the **BehaviourNodeWithChildren** class. To select from its child nodes, we have a simple for loop to visit each in turn - if the child has failed, it'll move on to the next (which might succeed, or at least be ongoing), otherwise, it'll set it's state to that of its child, and return. We can create this logic using a simple **switch** statement. On line 15 and 16, we make use of the fact that a C++ switch statement can 'fall through' from once case to the next, so here it will run lines 18 and 19 if the case is **either** *Success* or *Ongoing*. If all of a selector node's children have failed, the selector itself fails, as indicated by line 23. This will cause any further child nodes to be left without running in the frame.

```
1  #pragma once
2  #include "BehaviourNodeWithChildren.h"
3
4  class BehaviourSelector: public BehaviourNodeWithChildren {
5     public:
```

```
6     BehaviourSelector(const std::string& nodeName)
7        : BehaviourNodeWithChildren(nodeName) {}
8     ~BehaviourSelector() {}
9     BehaviourState Execute(float dt) override {
10       //std::cout << "Executing selector " << name << "\n";
11       for (auto& i : childNodes) {
12          BehaviourState nodeState = i->Execute(dt);
13          switch (nodeState) {
14             case Failure: continue;
15             case Success:
16             case Ongoing:
17             {
18                currentState = nodeState;
19                return currentState;
20             }
21          }
22       }
23       return Failure;
24    }
25 };
```

BehaviourSelector class header

BehaviourSequence Class

A sequence node acts much in the same way as a selector, with one important difference - if it finds a node that fails, it too must fail. We can use the same **switch** based logic for this, simply changing the cases around - the success of a child indicates the sequence can **continue**, while failure now **returns** the state, bubbling the 'failure' state back up the behaviour tree for other nodes to work with.

```
1  #pragma once
2  #include "BehaviourNodeWithChildren.h"
3
4  class BehaviourSequence : public BehaviourNodeWithChildren {
5     public:
6     BehaviourSequence(const std::string& nodeName)
7        : BehaviourNodeWithChildren(nodeName) {}
8     ~BehaviourSequence() {}
9     BehaviourState Execute(float dt) override {
10       //std::cout << "Executing sequence " << name << "\n";
11       for (auto& i : childNodes) {
12          BehaviourState nodeState = i->Execute(dt);
13          switch (nodeState) {
14             case Success: continue;
15             case Failure:
16             case Ongoing:
17             {
18                currentState = nodeState;
19                return nodeState;
20             }
21          }
22       }
23       return Success;
24    }
25 };
```

BehaviourSequence class header

BehaviourAction Class

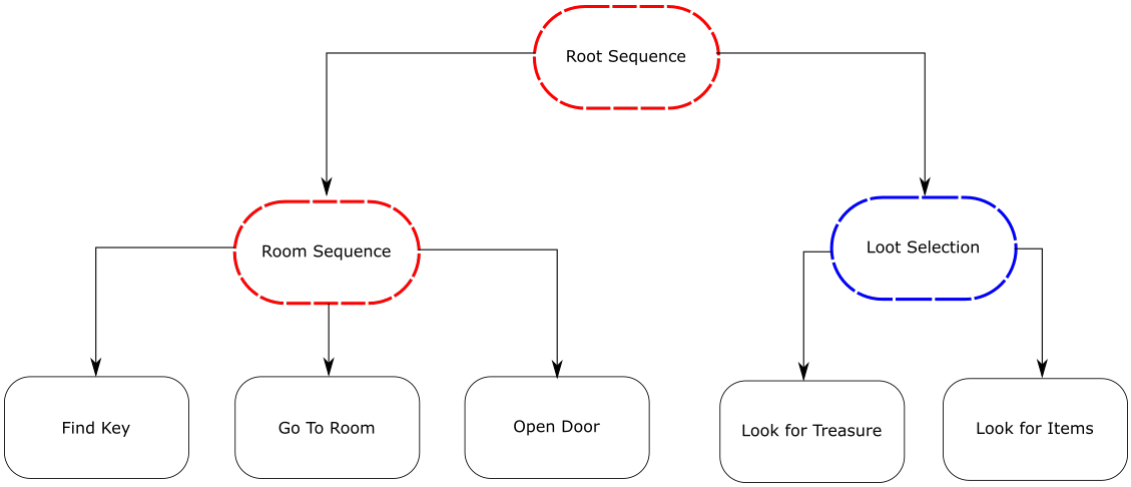
A behaviour tree is nothing without some actions to perform once its decisions have been made. To do this, we're going to have another class, **BehaviourAction**. Just as in the state machines tutorial, we're going to use a **std::function** to allow this class to enact custom logic without needing any further subclasses. The prototype for this function (line 5) takes in the timestep and our new node's current state, and returns a new **BehaviourState**, and will be called in the **BehaviourAction**'s *Execute* method.

```
1  #pragma once
2  #include "BehaviourNode.h"
3  #include <functional>
4
5  typedef std::function<BehaviourState(float, BehaviourState)>
6           BehaviourActionFunc;
7
8  class BehaviourAction : public BehaviourNode {
9     public:
10    BehaviourAction(const std::string& nodeName, BehaviourActionFunc f)
11       : BehaviourNode(nodeName) {
12       function = f; //sets our custom function!
13    }
14    BehaviourState Execute(float dt) override {
15       currentState = function(dt, currentState); //calls it!
16       return currentState;
17    }
18    protected:
19    BehaviourActionFunc function;
20 };
```

BehaviourAction class header

An example behaviour tree

That's all the 'skeleton' code we need to implement a basic behaviour tree. To show this off, we're going to create a simple behaviour tree, for an AI that tries to find a key to a room, goes to it, and then looks about in it searching for either treasure, or inventory items.



As in the other AI tutorials, we aren't really making a 'proper' game with graphics and physics here, just enough to demonstrate how a behaviour tree can help define logic. To create this example, we're going to have another function defined above the **main** function in the *main.cpp* file of the **GameTech** project.

To start off, we're going to have a couple of variables - we can assume that a 'proper' game would have a number of functions to provide access to important game state information (such as the distance between two objects), but for this simple example we'll define them here, and use lambda capture to access them.

We'll start by defining our action states, as this is where all of the custom logic will reside. The first **BehaviourAction** is to find a key for the door in our pretend game. Instead of having a full game, we'll instead say that to find a key takes a random amount of time, determined whether the state is first accessed (and is therefore in the *Initialise* state). Every time the state is executed, this timer will decrement by the timestep; should it reach zero, our AI 'finds' the key and the state succeeds, otherwise it will keep 'looking' by executing further steps.

```
1  void TestBehaviourTree() {
2     float behaviourTimer;
3     float distanceToTarget;
4     BehaviourAction* findKey = new BehaviourAction("Find Key",
5     [&](float dt, BehaviourState state)->BehaviourState{
6        if (state == Initialise) {
7           std::cout << "Looking for a key!\n";
8           behaviourTimer = rand() % 100;
9           state = Ongoing;
10       }
11       else if (state == Ongoing) {
12          behaviourTimer -= dt;
13          if (behaviourTimer <= 0.0f) {
14             std::cout << "Found a key!\n";
15             return Success;
16          }
17       }
18       return state; //will be 'ongoing' until success
19    }
20    );
```

main file TestBehaviourTree function

Once our AI finds a key, it should go to the room it is for. In a real game this would involve pathfinding and further decision making, but for now we'll again encapsulate this as a single variable, that will be used in much the same way as before, only this time we'll set the *distanceToTarget* variable elsewhere.

```
21    BehaviourAction* goToRoom = new BehaviourAction("Go To Room",
22    [&](float dt, BehaviourState state)->BehaviourState {
23       if (state == Initialise) {
24          std::cout << "Going to the loot room!\n";
25          state = Ongoing;
26       }
27       else if (state == Ongoing) {
28          distanceToTarget -= dt;
29          if (distanceToTarget <= 0.0f) {
30             std::cout << "Reached room!\n";
31             return Success;
32          }
33       }
34       return state; //will be 'ongoing' until success
35    }
36    );
```

main file TestBehaviourTree function

Doors need opening, so our AI will have a further state in its sequence to do just that. It doesn't take any time, so as soon as the state is first executed, it'll return *Success*, completing our sequence. As with the other nodes, this would probably have further logic in a 'real' game (it could have a timer on it, or a check for the AI having the correct stats to continue, and so forth), but it demonstrates our example sequence well enough.

```
37    BehaviourAction* openDoor = new BehaviourAction("Open Door",
38    [&](float dt, BehaviourState state)->BehaviourState {
39       if (state == Initialise) {
40          std::cout << "Opening Door!\n";
41          return Success;
42       }
43       return state;
44    }
45    );
```

main file TestBehaviourTree function

Once inside the room, our AI is going to try and find some loot. We'll represent this as a random chance of the state being successful, returning either *Success* or *Failure* based on a call to **rand()**.

```
46    BehaviourAction* lookForTreasure = new BehaviourAction(
47    "Look For Treasure",
48    [&](float dt, BehaviourState state)->BehaviourState {
49       if (state == Initialise) {
50          std::cout << "Looking for treasure!\n";
51          return Ongoing;
52       }
53       else if (state == Ongoing) {
54          bool found = rand() % 2;
55          if (found) {
56             std::cout << "I found some treasure!\n";
57             return Success;
58          }
59          std::cout << "No treasure in here...\n";
60          return Failure;
61       }
62       return state;
63    }
64    );
```

main file TestBehaviourTree function

If there's no treasure to be found in the room, maybe there's some items to take? We'll represent the chance of this in exactly the same way as treasure, calling **rand()**. In a 'real' game, such chances would probably be driven by a number of factors (a 'luck' character stat, or a calculation based on the character's level), but for now we'll use a simple **rand()** once again.

```
65    BehaviourAction* lookForItems = new BehaviourAction(
66    "Look For Items",
67    [&](float dt, BehaviourState state)->BehaviourState {
68       if (state == Initialise) {
69          std::cout << "Looking for items!\n";
70          return Ongoing;
71       }
72       else if (state == Ongoing) {
73          bool found = rand() % 2;
74          if (found) {
75             std::cout << "I found some items!\n";
76             return Success;
77          }
78          std::cout << "No items in here...\n";
79          return Failure;
80       }
81       return state;
82    }
83    );
```

main file TestBehaviourTree function

Now to connect all of our states together, based on the behaviour tree outlined earlier. We can do this via making use of the *AddChild* method on some instantiated sequences and selections.

```
84    BehaviourSequence* sequence =
85       new BehaviourSequence("Room Sequence");
86    sequence->AddChild(findKey);
87    sequence->AddChild(goToRoom);
88    sequence->AddChild(openDoor);
89
90    BehaviourSelector* selection =
91       new BehaviourSelector("Loot Selection");
92    selection->AddChild(lookForTreasure);
93    selection->AddChild(lookForItems);
94
95    BehaviourSequence* rootSequence =
96       new BehaviourSequence("Root Sequence");
97    rootSequence->AddChild(sequence);
98    rootSequence->AddChild(selection);
```

main file TestBehaviourTree function

To demonstrate the execution of the behaviour tree, we'll run it 5 times, resetting it each time, and checking to see whether the root of the tree eventually reaches a state of *Success* or *Failure*. Being a sequence node, it'll only return *Success* if both of its child states are also successful, so it'll only be successful if either treasure or items have been found.

```
99      for (int i = 0; i < 5; ++i) {
100        rootSequence->Reset();
101        behaviourTimer    = 0.0f;
102        distanceToTarget  = rand() % 250;
103        BehaviourState state = Ongoing;
104        std::cout << "We're going on an adventure!\n";
105        while (state == Ongoing) {
106          state = rootSequence->Execute(1.0f); //fake dt
107        }
108        if (state == Success) {
109          std::cout << "What a successful adventure!\n";
110        }
111        else if (state == Failure) {
112          std::cout << "What a waste of time!\n";
113        }
114      }
115      std::cout << "All done!\n";
116  }
```

main file TestBehaviourTree function

Conclusion

By adding a call to *TestBehaviourTree* at the start of the main function, we can now see an example of AI agents making decisions, based on a simple tree-like structure.
Behaviour trees are a popular choice for AI agents as they provide an intuitive means of chaining together operations, and making decisions based on the successes and failures of these operations. While a state machine could be created that performs the same actions as a behaviour tree, it is perhaps not always as intuitive to think how certain logic could be chained together, and concepts such as periodic decisions and interconnected states are handled in a much more intuitive manner with the separate nodes of a behaviour tree.

Further Work

1) Try creating a **ParallelBehaviour** subclass, that allows the AI to both search for treasure and items at the same time.

2) Decorator classes are a helpful addition to behaviour trees. They should always have a single child node, and should in some way base their Execute decision based on the result of executing this child node. Try making an invertor, so that the AI is only successful if the room has no treasure or items (perhaps it has been tasked with tidying up the level!).

3) Try making a subclass of **GameObject** that contains a **BehaviourNode** representing the root of a whole tree, that in some way informs what it should do from frame to frame. Something which searches for the player, and if the player is within a range will either chase them, or run away from them for a period of time, is a good basic behaviour to try and emulate.