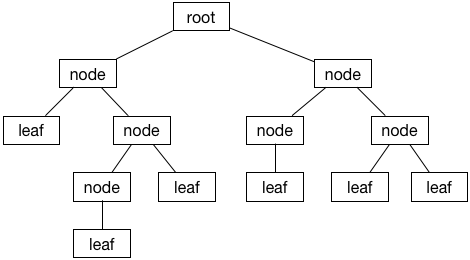
# Binary Trees

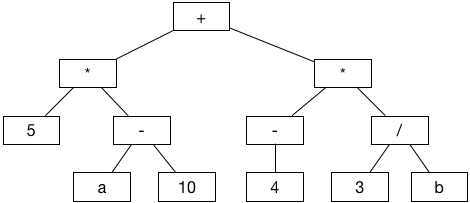
A binary tree is made up of connected nodes like regular trees. The main difference is that binary trees can only have up to 2 children (hence being a binary tree). If it does have two children, one is called the left child and the other is the right child. The children may or may not link back to their parent, so that may depend on your situation.

Here's an example from the Swift Algorithm Club.



Binary trees in this case don't have many special rules or anything like that. You'll see later in the course that if you do add some rules to a binary tree, it can make searching for data much easier.

One way ordinary binary trees can be used is to model arithmetic problems. The root node is some operator (like + or -) and the result of the left child will be added, subtracted, etc to the result of the right child. I say result because any of the nodes may be an operator, but they have to have two children (or one if it's a subtraction or addition). The leaf nodes will have the numbers in the equation, which may be further down on longer equations. You can put them in a binary tree and write a function that will find the final result, but we don't need to go that far into it. You can see an illustration courtesy of the Swift Algorithm Club below.



This tree represents (5 \* ( a - 10 ) + ( -4 \* ( 3 / b ) ). You can see that everything but the - sign above 4 has two children. Since the - sign is just making its child negative, it only needs one. Otherwise, it would be subtracting two numbers (like a - 10).

# How to Make a Binary Tree

Below is the starting code to making a binary tree (you may notice why I covered recursion before this).

class BinaryTree<T> {  
 var value: T?  
 private var leftBinaryTee: BinaryTree?  
 private var rightBinaryTree: BinaryTree?  
  
 init(\_ value) {  
 self.value = value  
  
 self.leftBinaryTree = nil  
 self.rightBinaryTree = nil  
 }  
}

It looks a lot like the recursive linked lists you saw before. I could change the names of the children to previous and next, and it would basically be identical. The main difference is how they're used though.

With the linked lists, everything was linear. In this binary tree, we'll be dividing the work between the two children instead of passing it down the line. This can improve performance since appending something doesn't mean it gets passed to every item saved.

We can start with appending an item, but first, we'll need to know the count of a tree first.

extension BinaryTree {  
 var count: Int {  
 guard value != nil else { return 0 }  
 return 1 + (self.leftBinaryTree?.count ?? 0) + (self.rightBinaryTree?.count ?? 0)  
 }  
}

OK, that does still look pretty close to a linked list, but this next part won't. To make it easier, I'm going to store whether the children are nil or not in some variables.

extension BinaryTree {  
 func add(\_ value) {  
 if self.value = nil {  
 self.value = value  
 } else {  
 let leftBinaryTreeIsNil = leftBinaryTree == nil  
 let rightBinaryTreeIsNil = rightBinaryTree == nil  
  
 if leftBinaryTreeIsNil {  
 self.leftBinaryTree = BinaryTree(value)  
 } else if !leftBinaryTreeIsNil && rightBinaryTreeIsNil {  
 self.rightBinaryTree = BinaryTree(value)  
 } else if !leftBinaryTreeIsNil && !rightBinaryTreeIsNil {  
 if self.leftBinaryTree!.count < self.rightBinaryTree!.count {  
 self.leftBinaryTree!.append(value)  
 } else {   
 self.rightBinaryTree!.append(value)  
 }  
 }  
 }  
 }  
}

To break that down a little -

1. I start by checking if the current value is nil. That would mean the tree is empty, so the value can just be saved in the existing value
2. If the current value is not nil, then the tree has at least one item, and the new value needs to be added to one of the children trees.
3. I save whether the children trees are nil to make the if statements a little bit clearer.
4. Next, if the left tree is nil, then we can just toss the value in a tree on that side.
5. If the left tree isn't nil, but the right one is, we can toss the value in there instead.
6. The last condition, if neither of the children are nil, is a little different. I check which one has the smallest count, and add the value to that. This helps the binary tree retain its efficiency. If every single item got added to the left or right tree, then the whole tree would be a more complicated list instead. That would lose any of the benefits of using a binary tree instead of a linked list.

Adding an element to a plain list runs in O(n) time. Adding an element to this tree takes O(h) time. h in this case is the height of the tree. Feel free to refer back to the previous article if you'd like a refresher on tree height.

It's easy to add a Swift Array to another Swift Array, so the BinaryTree might as well have some methods to handle doing the same with trees.

extension BinaryTree {  
 func removeLeaf() -> BinaryTree? {  
 guard self.value != nil else { return nil }  
  
 if let leftChild = self.leftBinaryTree {   
 return leftChild.removeLeaves()  
 }   
  
 if let rightChild = self.rightBinaryTree {  
 return rightChild.removeLeaves()  
 }  
  
 if let parent = self.parent {  
 if parent.leftBinaryTree == self {  
 parent.leftBinaryTree = nil  
 } else {  
 parent.rightBinaryTree = nil  
 }  
 }  
  
 self.value = nil  
  
 return self  
 }  
  
 func addTree(\_ newTreeNodes: BinaryTree) {  
 while newTreeNodes.count > 0 {  
 self.add(newTreeNodes.removeLeaf())  
 }  
 }  
}

The removeLeaves() method is new. If you're working with a linked list or array, it's pretty easy to just iterate over the entries in the collection being added. Trees don't have that linear structure. removeLeaves() will grab one of the leaves and return it. So, to add a tree, it just needs to be run until the tree being added has a count of 0.

Now, on to removing a node from the tree.

To find the right node, we have to first make sure BinaryTree conforms to Codable. We'll also just worry about finding a node based on its value.

extension BinaryTree: Equatable where T: Equatable {  
 public static func ==(lhs: BinaryTree<T>, rhs: BinaryTree<T>) -> Bool {  
 return lhs.value == rhs.value  
 }  
}

The extension is making BinaryTree conform to Equatable, but since not all of the types someone using the structure might as well, we can use where T: Equatable to require that anything stored in the tree can be used with ==.

We could go right into building the function to remove items, but that requires finding the element first. When you're using a tree, you might want to search for something without removing it, so writing a search function instead saves duplicating code.

There are three ways to search a tree.

1. In-Order, or depth-first. With this method, a node on the tree will search its left child, then itself, and then its right child.
2. Pre-order. First, a node will check itself, then its left child, and then its right child.
3. Post-order. A node will check its left child, its right child, and finally itself.

Here are examples

extension BinaryTree {   
 func inOrderSearch(for value: T) -> BinaryTree? {  
 guard self.value != nil else { return nil }  
  
 if let leftChild = self.leftBinaryTree {   
 if let nodeFound = leftChild.inOrderSearch(for: value) {  
 return nodeFound  
 }  
 }  
  
 // To illustrate the process, we'll print the current value  
 print("Checking \(self.value)")  
  
 if self.value! == value {  
 return self  
 }  
  
 if let rightChild = self.rightBinaryTree {  
 if let nodeFound = rightChild.inOrderSearch(for: value) {  
 return nodeFound  
 }  
 }  
  
 return nil  
 }  
  
 func preOrderSearch(for vaule: T) -> BinaryTree? {  
 // itself, left, right  
 guard self.value != nil else { return }  
  
 print("Checking \(self.value!)")  
  
 if self.value! == value {  
 return self  
 }  
  
 if let leftChild == self.leftBinaryTree {  
 if let foundNode == leftChild.preOrderSearch(for value: T) {  
 return foundNode  
 }  
 }  
  
 if let rightChild = self.rightBinaryTree {  
 if let foundNode = rightChild.preOrderSearch(for: value) {  
 return foundNode  
 }  
 }  
  
 return nil  
 }  
  
 func postOrderSearch(for value: T) -> BinaryTree? {  
 // left, right, itself  
 guard self.value != nil else { return }  
  
 if let leftChild = self.leftBinaryTree {  
 if let foundNode = leftChild.postOrderSearch(for: value) {  
 return foundNode  
 }  
 }  
  
 if let rightChild = self.rightBinaryTree {   
 if let foundNode = rightChild.postOrderSearch(for: value) {  
 return foundNode  
 }  
 }  
  
 print("Checking \(self.value!)")  
  
 if self.value! == value {  
 return self  
 }  
 }  
}

It may look like these functions aren't going to print anything if they find the value in one of the children trees. That's because recursion can make that seem a little more mixed up. If something is found in one of the child trees, then one of the child nodes had to have hit the if self.value! == value line. That's why the print statement is always coming directly before that check.

Now that we have a way to see if we have the node we're looking for, we can write a function to remove something. To make it interesting, the function will let whoever is using it decide which search method they want to use.

extension BinaryTree<T> {  
 // This is an enum to store the different supported search methods  
 public enum SearchMethod {  
 case .inOrder, .preOrder, .postOrder  
 }  
  
 // We'll need the root node for some of the functionality (and the   
 // removal could start at any node in the tree) so it might as well be a  
 // function that can be reused  
 func findRootNode() -> BinaryTree? {  
 guard self.value != nil else { return nil }  
  
 var currentNode = self  
  
 while currentNode.parent != nil {  
 currentNode = currentNode.parent  
 }  
  
 return currentNode  
 }  
  
 // The remove function will just return a Bool about whether or not the  
 // the value was found and removed  
 func remove(\_ value: T, withSearchMethod searchMethod: SearchMethod) -> Bool {  
 // If the value is nil, then the tree is empty and there's nothing to remove  
 guard self.value != nil else { return false }  
  
 var nodeToRemove: BinaryTree? = nil  
  
 // This switch statement will find the node to remove based on   
 // which search method people chose  
 switch searchMethod {  
 case .inOrder:  
 nodeToRemove = self.inOrderSearch(for: value)  
 case .preOrder:  
 nodeToRemove = self.preOrderSearch(for: value)  
 case .postOrder:  
 nodeToRemove = self.postOrderSearch(for: value)  
 }  
  
 guard let nodeToRemove = nodeToRemove else { return false }  
  
 let leftChild = self.leftBinaryTree  
 let rightChild = self.rightBinaryTree  
 let root = self.findRootNode()  
  
 // A leaf node has no children, and since it's the easiest case  
 // we can start there  
 if leftChild == nil && rightChild == nil {  
 if self.parent == nil {  
 self.value = nil  
 } else {  
 if self.parent!.leftBinaryTree == self {  
 self.parent!.leftBinaryTree = nil  
 } else {  
 self.parent!.rightBinaryTree = nil  
 }  
 } else if leftChild == nil || rightChild == nil {  
 // This is the case where there's only one child, which is also pretty  
 // easy. The child just needs to take the place of the current node.  
 var treeToKeep = leftChild == nil ? rightChild : leftChild  
  
 treeToKeep.parent = self.parent  
  
 if let parent = self.parent {  
 if parent.leftBinaryTree == self {  
 parent.leftBinaryTree = treeToKeep  
 } else {  
 parent.rightBinaryTree = treeToKeep  
 }  
 } else {  
 self.value = treeToKeep.value  
 self.leftBinaryTree = treeToKeep.leftBinaryTree  
 self.rightBinaryTree = treeToKeep.rightBinaryTree  
 }  
 } else {  
 // The most complicated situation is when the node to remove has two  
 // children. To keep this simple (ish), we can just replace the current  
 // node with the biggest tree, and then add the nodes in the smaller  
 // tree back to the root of the tree. It's not the fastest way,  
 // but will help keep the tree slightly more balanced.  
 var smallestChild: BinaryTree<T>?  
 var largestChild: BinaryTree<T>?  
  
 if leftChild!.count > rightChild!.count {  
 smallestChild = rightChild  
 largestChild = leftChild  
 } else {  
 smallestChild = leftChild  
 largestChild = rightChild  
 }  
  
 guard var smallestChild = smallestChild else { return }  
 guard var largestChild = largestChild else { return }  
  
 largestChild.parent = nodeToRemove.parent  
 largestChild.leftBinaryTree = nodeToRemove.leftBinaryTree  
 largestChild.rightBinaryTree = nodeToRemove.rightBinaryTree  
  
 if nodeToRemove == nodeToRemove.parent?.leftBinaryTree {  
 nodeToRemove.parent!.leftBinaryTree = largestChild  
 } else if nodeToRemove == nodeToRemove.parent?.rightBinaryTree {  
 nodeToRemove.parent!.rightBinaryTree = largestChild  
 }  
  
 root.addTree(smallestChild)  
 }  
  
 return true  
 }  
}

## Summary

There's a lot that goes into binary trees. They're tough, definitely. If they seem confusing at first, I'll include this in a Playground so there's an opportunity to practice using them or making one. This is going to come up in a later unit as well, so feel free to bookmark or refer back to this when we start covering Binary Search Trees.