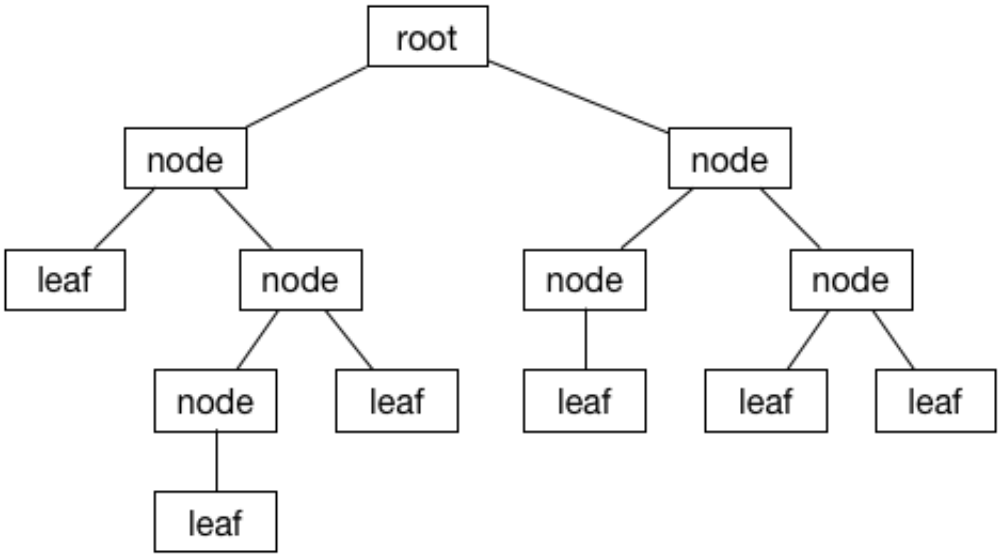
Binary Trees

Written by Sam Smith 12/16/2019, Swift 5

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.

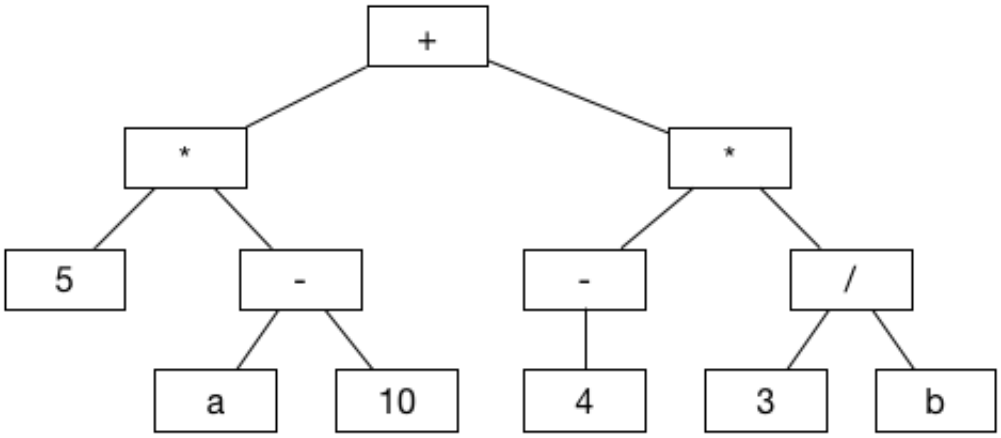
Here's an example from the Swift Algorithm Club.



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`).

Why Use Binary Trees

So, why should you use binary trees? They come in handy when implemented as Binary Search Trees, which get covered later, but their extra rules as compared to regular trees have a few benefits.

Like trees, binary trees have to be careful about how they add data. Specifically, they have to make sure that not everything is always on the left or right child. If every item added got thrown into one or the other, they're suddenly more complicated lists. They need to do what the can to make sure everything gets added evenly so that their height doesn't become equal to their count.

The limit on the number of children help prevent them from becoming arrays, as well. If you had a tree that had the root, and everything else in it was a child of that

root, it would become a more complicated array.

That's a common theme with several different data structures. Ignoring the constraints added to it ends up with a more complicated version of an existing data structure. All of them have different pros and cons, but no data structure should be just "a more complicated version of a different structure". The rules are what give them their benefits. No developer wants to make something more complicated just to make it more complicated.

Initial Declaration

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

```
public class BinaryTree<T> {
    var value: T?
    public var leftBinaryTree: BinaryTree?
    public var rightBinaryTree: BinaryTree?
    private var parent: BinaryTree?

    init(_ value: T) {
        self.value = value

        self.leftBinaryTree = nil
        self.rightBinaryTree = nil
        self.parent = 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.

Adding Elements

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 self.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: T) {
        // First, check if the tree is empty, assign to root if it
is
        if self.value == nil {
            self.value
        } else {
            let leftBinaryTreeIsNil = self.leftBinaryTree == nil
            let rightBinaryTreeIsNil = self.rightBinaryTree == nil

            if leftBinaryTreeIsNil {
```

```
                self.leftBinaryTree = BinaryTree(value)
                self.leftBinaryTree!.parent = self
            } else if !leftBinaryTreeIsNil && rightBinaryTreeIsNil
{
                self.rightBinaryTree = BinaryTree(value)
                self.rightBinaryTree!.parent = self
            } else if !leftBinaryTreeIsNil && !rightBinaryTreeIsNil
{
                if self.leftBinaryTree!.count <
self.rightBinaryTree!.count {
                    self.leftBinaryTree!.add(value)
                } else {
                    self.rightBinaryTree!.add(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.

Adding Another Tree

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.removeLeaf()
        }

        if let rightChild = self.rightBinaryTree {
            return rightChild.removeLeaf()
        }

        if let parent = self.parent {
            if parent.leftBinaryTree != nil &&
parent.leftBinaryTree! === self {
                parent.leftBinaryTree = nil
            } else {
                parent.rightBinaryTree = nil
            }
        }

        let returnTree = BinaryTree(self.value!)

        self.value = nil
```

```
        return returnTree
    }

    func addTree(_ newTreeNodes: BinaryTree) {
        while newTreeNodes.count > 0 {
            self.add(newTreeNodes.removeLeaf()!.value!)
        }
    }
}
```

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.

Before talking about removing items from the tree, it needs to be able to find its nodes efficiently.

Searching Elements

To find the right node, we have to first make sure `BinaryTree` conforms to `Equatable`. 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 where T: Equatable {
    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
            }
        }

        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 value: T) -> BinaryTree? {
        guard self.value != nil else { return nil }

        print("Checking \(self.value!)")

        if self.value! == value {
            return self
        }

        if let leftChild = self.leftBinaryTree {
            if let foundNode = leftChild.preOrderSearch(for: value)
{
                return foundNode
            }
        }

        if let rightChild = self.rightBinaryTree {
            if let foundNode = rightChild.preOrderSearch(for:
value) {
                return foundNode
            }
        }

        return nil
    }

    func postOrderSearch(for value: T) -> BinaryTree? {
        guard self.value != nil else { return nil }
```

```
        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
        }

        return nil
    }
}
```

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.

Removing Items

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 {
        guard self.value != nil else { return false }

        var nodeToRemove: BinaryTree? = nil

        switch searchMethod {
        case .inOrder:
```

```
            nodeToRemove = self.inOrderSearch(for: value)
        case .preOrder:
            nodeToRemove = self.preOrderSearch(for: value)
        case .postOrder:
            nodeToRemove = self.postOrderSearch(for: value)
        }

        guard var foundNodeToRemove = nodeToRemove else { return
false }

        let leftChild = foundNodeToRemove.leftBinaryTree
        let rightChild = foundNodeToRemove.rightBinaryTree
        let parent = foundNodeToRemove.parent

        if parent != nil {
            if leftChild == nil && rightChild == nil {
                if parent!.leftBinaryTree === foundNodeToRemove {
                    parent!.leftBinaryTree = nil
                } else {
                    parent!.rightBinaryTree = nil
                }

            } else if leftChild != nil && rightChild == nil {
                leftChild!.parent = foundNodeToRemove.parent

                if foundNodeToRemove.parent!.leftBinaryTree ===
foundNodeToRemove {
                    foundNodeToRemove.parent!.leftBinaryTree =
leftChild
                } else {
                    foundNodeToRemove.parent!.rightBinaryTree =
leftChild
                }

            } else if leftChild == nil && rightChild != nil {
                rightChild.parent = foundNodeToRemove.parent

                if foundNodeToRemove.parent!.leftBinaryTree ===
```

```
foundNodeToRemove {
                    foundNodeToRemove.parent!.leftBinaryTree =
rightChild
                } else {
                    foundNodeToRemove.parent!.rightBinaryTree =
rightChild
                }

            } else {
                let smallestChild = leftChild!.count >
rightChild.count ? rightChild! : leftChild!
                let largestChild = leftChild!.count >
rightChild!.count ? leftChild! : rightChild!

                largestChild.parent = foundNodeToRemove.parent

                if foundNodeToRemove.parent!.leftBinaryTree ===
foundNodeToRemove {
                    foundNodeToRemove.parent!.leftBinaryTree =
largestChild
                } else {
                    foundNodeToRemove.parent!.rightBinaryTree =
largestChild
                }

                let root = self.findRootNode()!

                root.addTree(smallestChild)
            }
        } else {
            if leftChild == nil && rightChild == nil {
                foundNodeToRemove = nil

            } else if leftChild != nil && rightChild == nil {
                foundNodeToRemove = leftChild!
                leftChild!.parent = nil

            } else if leftChild == nil && rightChild != nil {
```

```
                foundNodeToRemove = rightChild!
                rightChild!.parent = nil

            } else {
                let smallestChild = seftChild!.count >
rightChild!.count ? rightChild! : leftChild!
                let largestChild = leftChild!.count >
rightChild!.count ? leftChild! : rightChild!

                foundNodeToRemove = largestChild

                largestChild.leftBinaryTree?.parent =
foundNodeToRemove
                largestChild.rightBinaryTree?.parent =
foundNodeToRemove

                foundNodeToRemove.addTree(smallestChild)
            }
        }

        return true
    }
}
```

Breaking this down is going to be a little more complicated. There's a few cases that need to be kept in mind before tackling this (and honestly, before I considered them, I wrote a method that completely didn't work). The node being removed might be:

1. Not the root node, with no children
2. Not the root node, with one child
3. Not the root node, with two children
4. The root node, with no children
5. The root node, with one child

6. The root node, witch two children

There's some similarities between handling random nodes in the tree and handling removing the root node, but there are also extra gotchas that make it just easier to treat them all differently.

Not the Root Node, with No Children

The simplest condition is one where the node being removed is a leaf node. In this case, it does have a parent node, since it's not the root. In that case, all that needs to be done is:

1. Find if the node is the left or right child of the parent
2. Set the correct child to nil

You do still have to make sure not to set the `.rightBinaryTree` to nil if the node is actually a `.leftBinaryTree`, but that's pretty much all there is to it.

Not the Root Node, with One Child

The next simplest condition is when you have a node being removed that only has one child. Again, since it's not the root node, it has to have a parent node. In that case, all that needs done is:

1. Find if the node being removed is the left or right child of the parent
2. Set the correct child to the only child the node being removed has
3. Set the parent of the child node to the right parent

Basically, it's cutting the current node out of the hierarchy. Its parent sees its child as its own child, and its child sees its parent as its own parent.

Not the Root Node, with Two Children

Removing a node with two children is the most complicated case, even at the root node. You can't just remove the node from the hierarchy, since there are two children and the parent node should ideally have two children already. So you can't just assign both of these children to the parent. Darn binary tree rules.

There are a few different ways to handle this, but we'll do the easiest. We can replace the node being removed with its largest child, and then just add each node in the smallest child (which helps performance) to the tree as if it were a new node. This also helps keep some of the structure, so the tree doesn't wind up becoming a complicated list.

So, to remove a node with two children:

1. Find if the node being removed is the left or right child of the parent
2. Set the correct parent child to the largest child of the node being removed
3. Set the parent of the largest child of the node being removed to its parent
4. Find the root node of the tree
5. Add the smallest child of the node being removed to the root node so it can distribute it as it was set up to do

The Root Node with No Children

There isn't too much that changes when removing the root node. The biggest thing is it doesn't have a parent, so there's no need to check if it's the left or right child. Actually, not accounting for that can lead to issues. When the root node has no children, to remove it:

1. Set the `value` to nil

That's it.

The Root Node with One Child

The `remove(_: withSearchMethod:)` function doesn't use recursion like the other functions so far. The search functions are recursive themselves, but then it works with `foundNodeToRemove` rather than `self`. This makes removing the root node when it has one child pretty simple. If it used `self`, something like `self = self.rightBinaryTree` won't work. `foundNodeToRemove = foundNodeToRemove.rightBinaryTree` is just fine, though.

To remove the root node with one child:

1. Find out if `leftBinaryTree` or `rightBinaryTree` is not nil
2. Assign the non-nil child to `foundNodeToRemove`
3. Assign nil to `foundNodeToRemove.parent` since it's the new root

The `foundNodeToRemove.parent` needs set to nil since otherwise it will point at the previous root node.

The Root Node with Two Children

This case is also going to be pretty similar to any other node with two children. It doesn't need to search for the root, though, since it is the root. Like the previous case, it also doesn't need to change anything about its parent since it doesn't have one.

To remove the root node when it has two children:

1. Assign the largest child to `foundNodeToRemove`
2. Assign nil to `foundNodeToRemove.parent`
3. Add the smallest child back to the tree using `nodeToRemove.addTree()`

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.