Write a function to see if a binary tree is *"superbalanced"* (a new tree property we just made up).

A tree is "superbalanced" if the difference between the depths of any two leaf nodes is no greater than one.

Here's a sample binary tree node class:

```ruby
class BinaryTreeNode

    attr_accessor :value
    attr_reader :left, :right

    def initialize(value)
        @value = value
        @left  = nil
        @right = nil
    end

    def insert_left(value)
        @left = BinaryTreeNode.new(value)
        return @left
    end

    def insert_right(value)
        @right = BinaryTreeNode.new(value)
        return @right
    end
end
```

Gotchas

Your first thought might be to write a recursive function, thinking, "the tree is balanced if the left subtree is balanced and the right subtree is balanced." This kind of approach works well for some other tree problems.

But this isn't quite true. Counterexample: suppose that from the root of our tree:

- The left subtree only has leaves at depths 10 and 11.

- The right subtree only has leaves at depths 11 and 12.

Both subtrees are balanced, but from the root we will have leaves at 3 different depths.

We could instead have our recursive function get the array of distinct leaf depths for each subtree. That could work fine. But let's come up with an iterative solution instead. It's usually better to use an iterative solution instead of a recursive one because it avoids stack overflow.

We can do this in $O(n)$ time and $O(n)$ space.

Breakdown

Sometimes it's good to start by rephrasing or "simplifying" the problem.

The requirement of "the difference between the depths of any two leaf nodes is no greater than 1" implies that we'll have to compare the depths of *all possible pairs* of leaves. That'd be expensive—if there are n leaves, there are n^2 possible pairs of leaves.

But we can simplify this requirement to require less work. For example, we could equivalently say:

- "The difference between the min leaf depth and the max leaf depth is 1 or less"
- "There are at most two distinct leaf depths, and they are at most 1 apart"

If you're having trouble with a recursive approach, try using an iterative one.

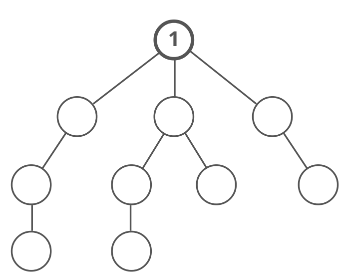
To get to our leaves and measure their depths, we'll have to traverse the tree somehow. **What methods do we know for traversing a tree?**

Depth-first

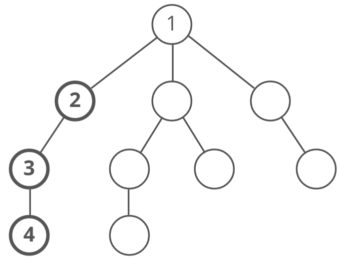
Depth-first search is a method for exploring a tree or graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one.

Depth-first search is like walking through a corn maze. You explore one path, hit a dead end, and go back and try a different one.

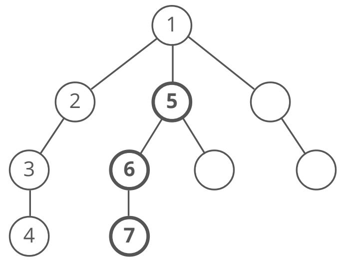
Here's a how a DFS would traverse this tree, starting with the root:



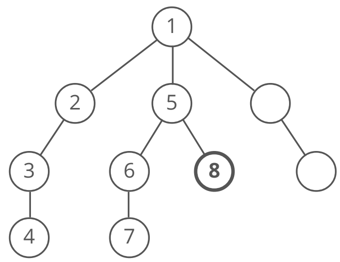
We'd go down the first path we find until we hit a dead end:



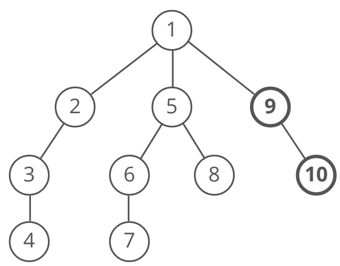
Then we'd do the same thing again—go down a path until we hit a dead end:



And again:



And again:



Until we reach the end.

Depth-first search is often compared with **breadth-first search**.

Advantages:

- Depth-first search on a binary tree *generally* requires less memory than breadth-first.
- Depth-first search can be easily implemented with recursion.

Disadvantages

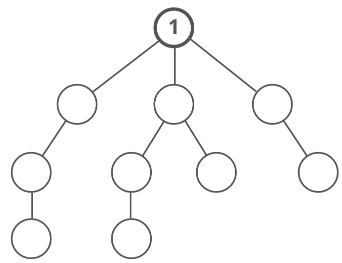
- A DFS doesn't necessarily find the shortest path to a node, while breadth-first search does.

and breadth-first↴

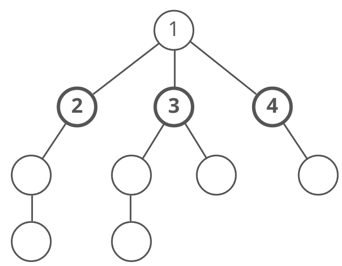
Breadth-first search (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.

Breadth-first search is like throwing a stone in the center of a pond. The nodes you explore "ripple out" from the starting point.

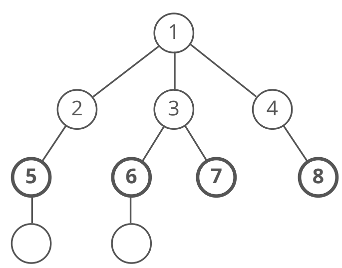
Here's a how a BFS would traverse this tree, starting with the root:



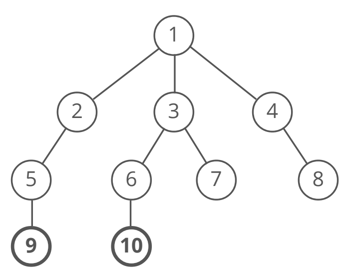
We'd visit all the immediate children (all the nodes that're one step away from our starting node):



Then we'd move on to all *those* nodes' children (all the nodes that're *two steps* away from our starting node):



And so on:



Until we reach the end.

Breadth-first search is often compared with **depth-first search**.

Advantages:

- A BFS will find the **shortest path** between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.

Disadvantages

- A BFS on a binary tree *generally* requires more memory than a DFS.

are common ways to traverse a tree. Which one should we use here?

The worst-case time and space costs of both are the same—you could make a case for either.

But one characteristic of our algorithm is that it could **short-circuit** and return `false` as soon as it finds two leaves with depths more than 1 apart. So maybe we should **use a traversal that will hit leaves as quickly as possible...**

Depth-first traversal will generally hit leaves before breadth-first, so let's go with that. How could we write a depth-first walk that also keeps track of our depth?

Solution

We do a depth-first walk through our tree, keeping track of the depth as we go. When we find a leaf, we throw its depth into an array of `depths` *if* we haven't seen that depth already.

Each time we hit a leaf with a new depth, there are two ways that our tree might now be unbalanced:

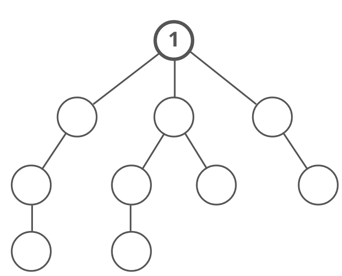
1. There are more than 2 different leaf depths
2. There are exactly 2 leaf depths and they are more than 1 apart.

Why are we doing a depth-first walk and not a breadth-first

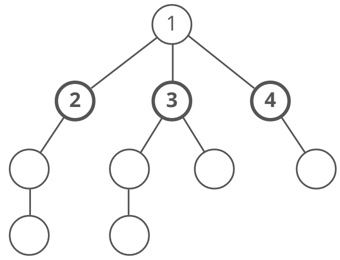
Breadth-first search (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.

Breadth-first search is like throwing a stone in the center of a pond. The nodes you explore "ripple out" from the starting point.

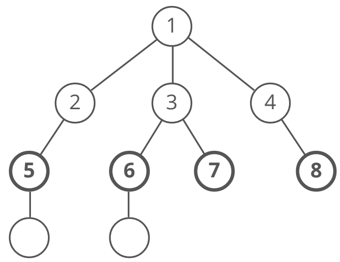
Here's a how a BFS would traverse this tree, starting with the root:



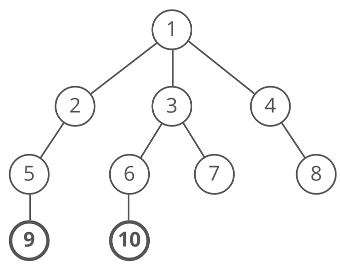
We'd visit all the immediate children (all the nodes that're one step away from our starting node):



Then we'd move on to all *those* nodes' children (all the nodes that're *two steps* away from our starting node):



And so on:



Until we reach the end.

Breadth-first search is often compared with **depth-first search**.

Advantages:

- A BFS will find the **shortest path** between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.

Disadvantages

- A BFS on a binary tree *generally* requires more memory than a DFS.

one? You could make a case for either. We chose depth-first because it reaches leaves faster, which allows us to short-circuit earlier in some cases.

Ruby

```ruby
def is_balanced(tree_root)

    # a tree with no nodes is superbalanced, since there are no leaves!
    if !tree_root
        return true
    end

    depths = [] # we short-circuit as soon as we find more than 2

    # we'll treat this array as a stack that will store pairs [node, depth]
    nodes = []
    nodes.push([tree_root, 0])

    while !nodes.empty?

        # pop a node and its depth from the top of our stack
        node, depth = nodes.pop

        # case: we found a leaf
        if !node.left && !node.right

            # we only care if it's a new depth
            if !depths.include? depth
                depths.push(depth)

                # two ways we might now have an unbalanced tree:
                #   1) more than 2 different leaf depths
                #   2) 2 leaf depths that are more than 1 apart
                if (depths.length > 2) || \
                        (depths.length == 2 && (depths[0] - depths[1]).abs > 1)
                    return false
                end
            end

        # case: this isn't a leaf - keep stepping down
        else
            if node.left
                nodes.push([node.left, depth + 1])
            end
            if node.right
                nodes.push([node.right, depth + 1])
            end
        end
    end

    return true
end
```

Complexity

$O(n)$ time and $O(n)$ space.

For time, the worst case is the tree *is* balanced and we have to iterate over *all n nodes* to make sure.

For the space cost, we have two data structures to watch: `depths` and `nodes`.

`depths` will never hold more than three elements, so we can write that off as $O(1)$.

Because we're doing a depth first search, `nodes` will hold at most d nodes where d is the depth of the tree (the number of levels in the tree from the root node down to the lowest node). So we *could* say our space cost is $O(d)$.

But we can also relate d to n. In a balanced tree, d is $O(\log_2(n))$ (/concept/binary-tree#property2). And the *more unbalanced* the tree gets, the closer d gets to n.

In the worst case, the tree is a straight line of right children from the root where every node in that line also has a left child. The traversal will walk down the line of right children, adding a new left child to `nodes` at each step. When the traversal hits the rightmost node, `nodes` will hold *half* of the n total nodes in the tree. Half n is $O(n)$, so our worst case space cost is $O(n)$.

What We Learned

This is an intro to some tree basics. If this is new to you, don't worry—it can take a few questions for this stuff to come together. We have a few more coming up.

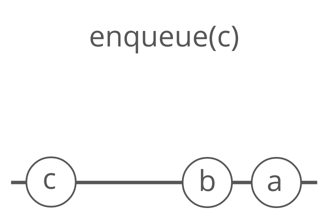
Particular things to note:

Focus on **depth-first vs breadth-first traversal**. You should be very comfortable with the differences between the two and the strengths and weaknesses of each.

You should also be very comfortable coding each of them up.

One tip: **Remember that breadth-first uses a queue**

A queue is like a line at the movie theater. It's "first in, first out" (FIFO), which means that the item that was put in the queue *longest ago* is the first item that comes out. "First come, first served."



Queues have two main methods:

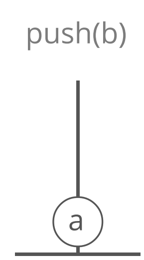
1. **`enqueue()`** : adds an item
2. **`dequeue()`** : removes and returns the next item in line

They can also include some utility methods:

1. **`peek()`** : returns the item at the front of the queue, without removing it.
2. **`is_empty()`** : returns true if the queue is empty, false otherwise

and depth-first uses a stack

A stack is like a stack of plates. It's "last in, first out" (LIFO), which means the item that was put in the stack *most recently* is the first item that comes out.



Stacks have two main methods:

1. **`push()`** : adds an item
2. **`pop()`** : removes and returns the top item

They can also include some utility methods:

1. **`peek()`** : returns the item on the top of the stack, without removing it.
2. **`is_empty()`** : returns `true` if the stack is empty, `false` otherwise

(could be the call stack or an actual stack object). That's not just a clue about implementation, it also helps with figuring out the differences in behavior. Those differences come from whether we visit nodes in the order we see them (first in, first out) or we visit the last-seen node first (last in, first out).