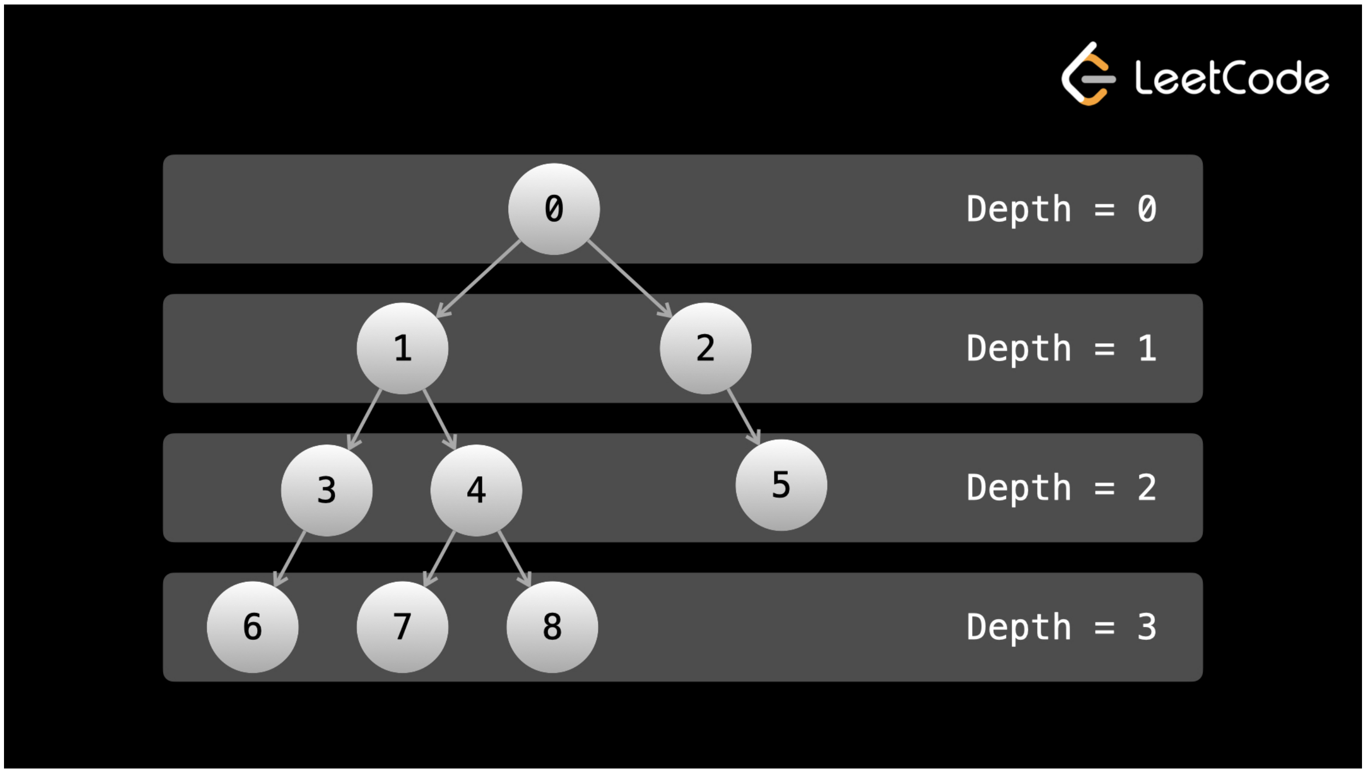
Binary trees - BFS

In DFS, we prioritized depth. In breadth-first search (BFS), we prioritize breadth. Recall that a node's depth is its distance from the root. In DFS, we always tried to go down as far as we could, increasing the depth of the current node until we reached a leaf. If you performed DFS on a large tree, the depth of the nodes you would traverse may look something like `0, 1, 2, 3, 4, 5, 6, ...`.

In BFS, **we traverse all nodes at a given depth before moving on to the next depth.** So if you performed BFS on a large complete binary tree, the depth of the nodes you would traverse would look like `0, 1, 1, 2, 2, 2, 2, 2, 3, 3, ...`.

A "complete" binary tree is one where every level (except possibly the last) is full, and all the nodes in the last level are as left as possible.

We can think of each depth of a tree as a "level", as if the tree was a building with the root being the top floor and the edges were staircases down to a lower floor.



A BFS performed on the tree above visit the nodes in the same order as their values. We visit all nodes at a depth `d` before visiting any node at a depth of `d + 1`.

While DFS was implemented using a stack (recursion uses a stack under the hood), BFS is implemented iteratively with a queue. You *can* implement BFS with recursion, but it wouldn't make sense as it's a lot more difficult without any benefit. So we will look only at iterative implementations in this course.

When to use BFS vs. DFS?

Earlier we mentioned that in many problems, it doesn't matter if you choose preorder, inorder, or postorder DFS; the important thing is that you just visit all the nodes. In a problem like this, then it doesn't matter if you use BFS either, because every "visit" to a node will store sufficient information irrespective of visit order.

Because of this, in terms of binary tree algorithm problems, it is very rare to find a problem that using DFS is "better" than using BFS. However, implementing DFS is usually quicker because it requires less code, and is easier to implement if using recursion, so for problems where BFS/DFS doesn't matter, most people end up using DFS.

On the flip side, there are quite a few problems where using BFS makes way more sense algorithmically than using DFS. Usually, this applies to any problem where we want to handle the nodes according to their level.

In an interview, you may be asked some trivia regarding BFS vs. DFS, such as their drawbacks. The main disadvantage of DFS is that you could end up wasting a lot of time looking for a value. Let's say you had a **huge** tree, and were looking a value that is stored in the root's right child. If you do DFS prioritizing left before right, then you will search the **entire** left subtree, which could be hundreds of thousands if not millions of operations. Meanwhile, the nodes is literally one operation away from the root. The main disadvantage of BFS is that if the node you're searching for is near the bottom, then you will waste a lot of time searching through all the levels to reach the bottom.

In terms of space complexity, DFS uses space linear with the height of the tree (the maximum depth), whereas BFS uses space linear with the level that has the most nodes. In some cases, DFS uses less space, in other cases, BFS uses less.

For example, in a perfect binary tree (all interior nodes have two children *and* all leaves have the same *depth*), DFS would uses $O(logn)$ space, whereas BFS would use $O(n)$. The final level in a perfect binary tree has $n/2$ nodes, but the tree only has a depth of $logn$.

However, if you have a lopsided tree (like a straight line), then BFS will have an $O(1)$ space complexity while DFS will have $O(n)$ (although, a lopsided tree is an edge case whereas a more full tree is the expectation).

Just like DFS, the code of BFS is very similar across different problems:

```
from collections import deque

def print_all_nodes(root):
    queue = deque([root])
    while queue:
        nodes_in_current_level = len(queue)
        # do some logic here for the current level

        for _ in range(nodes_in_current_level):
            node = queue.popleft()

            # do some logic here on the current node
            print(node.val)

            # put the next level onto the queue
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
```

Note for JavaScript users: JavaScript doesn't support a built-in efficient queue, but we can work around this by using a second array `nextQueue` to implement an efficient BFS.

At the start of each iteration inside the while loop (where the comment `"do some logic here on the current level"` is), the queue contains exactly all the nodes for the current level. In the beginning, that's just the root.

We then use a for loop to iterate over the current level. We store the number of nodes in the current level `nodesInCurrentLevel` before iterating to make sure the for loop doesn't iterate over any other nodes. The for loop visits each node in the current level and puts all the children (the next level's nodes in the queue).

Because we are removing from the left and adding on the right (opposite ends), after the loop finishes, the queue will be have none of the old level's nodes anymore and now will hold all the nodes in the next level. We move to the next while loop iteration and the process repeats.

With an efficient queue, the dequeue and enqueue operations are $O(1)$, which means that the time complexity of BFS is the same as DFS. Again the main idea is that we visit each node only once, so the time complexity is $O(nk)$ where n is the total number of nodes, and k is the amount of work we do at each node, usually $O(1)$.

Example 1: 199 - Binary Tree Right Side View

Given the `root` of a binary tree, imagine yourself standing on the right side of it. Return the values of the nodes yous can see ordered from top to bottom.

Essentially, this question is asking for the rightmost node at each level. If we do BFS with the same code format as above, we have a moment at each while loop iteration where we have an entire level in an array. If we prioritize the left children before the right children, then the final value at each iteration will be the rightmost node.

```
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []

        ans = []
        queue = deque([root])

        while queue:
            current_length = len(queue)
            ans.append(queue[-1].val) # this is the rightmost node for the
current level

            for _ in range(current_length):
                node = queue.popleft()
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

        return ans
```

The time and space complexities are $O(n)$.

Example 2: 515 - Find Largest Value in Each Tree Row

Given the `root` of a binary tree, return an array of the largest value in each row of the tree.

In the context of a binary tree, row and level mean the same thing. Each iteration inside the while loop represents going through a level. So we can simply use an integer `currMax` to find the largest value at each level.

```
class Solution:
    def largestValues(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []

        ans = []
        queue = deque([root])

        while queue:
            current_length = len(queue)
            curr_max = float("-inf") # this will store the largest value for
the current level

            for _ in range(current_length):
                node = queue.popleft()
                curr_max = max(curr_max, node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            ans.append(curr_max)

        return ans
```

The time and space complexities are $O(n)$ for the same reasons as usual.

For binary trees BFS vs. DFS rarely matters. However, there are some cases where BFS really shines, like in these examples. When we get to graph problems, we'll see more problems where BFS is very powerful.

SEE PROBLEMS 1302, 103