Breaking Down Breadth-First Search

Vaidehi Joshi · Follow
Published in basecs · 11 min read · Apr 10, 2017

When it comes to learning, there are generally two approaches one can take: you can either go wide, and try to cover as much of the spectrum of a field as possible, or you can go deep, and try to get really, really specific with the topic that you're learning. Most good learners know that, to some extent, everything you that you learn in life — from algorithms to basic life skills — involves some combination of these two approaches.

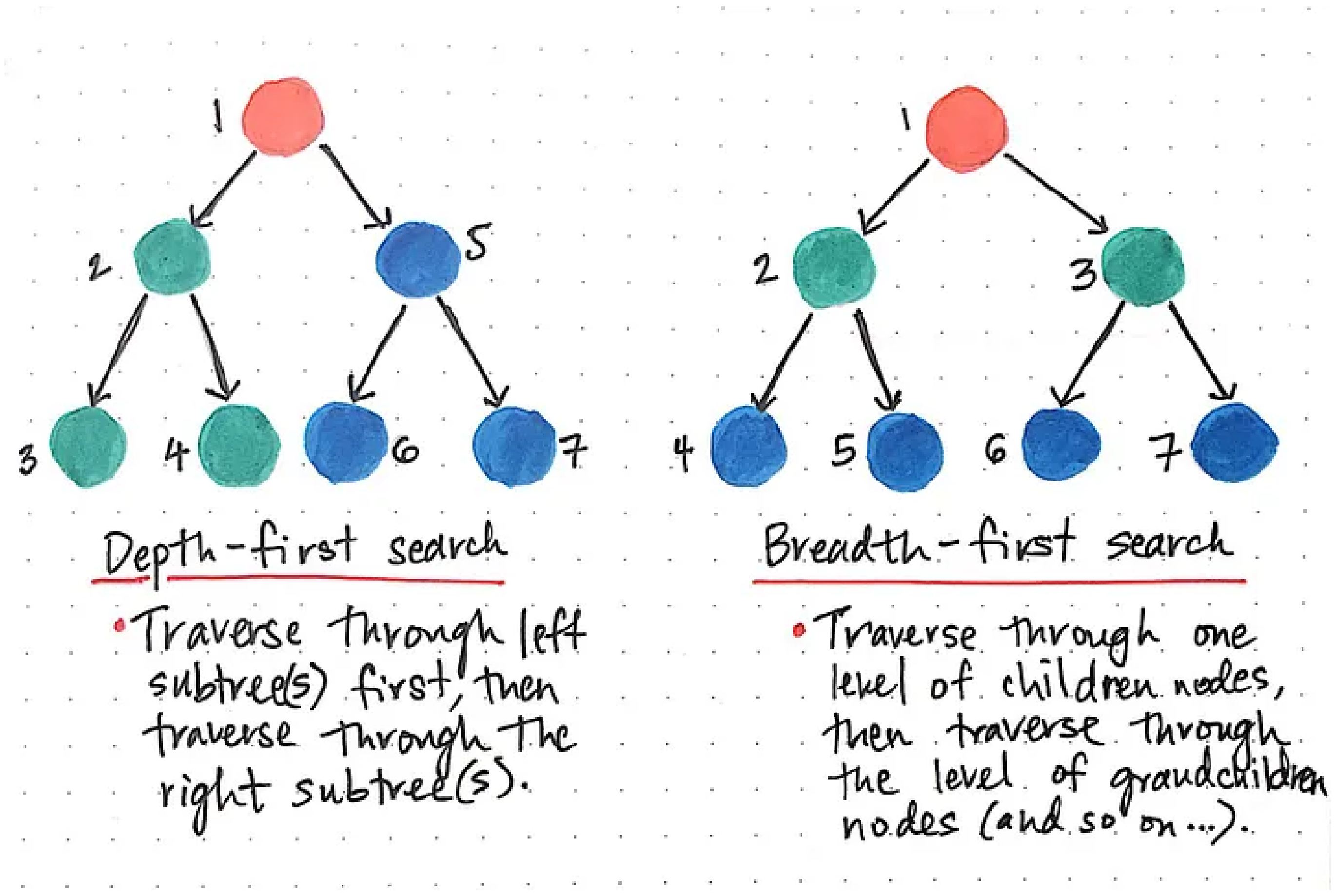
The same goes for computer science, problem solving, and data structures. Last week, we dove deep into depth-first search and learned what it means to actually traverse through a binary search tree. Now that we've gone deep, it makes sense for us to go wide, and understand the other common tree traversal strategy.

In other words, it's the moment you've all been waiting for: it's time to break down the basics of breadth-first search!

DFS and BFS: different, but equal

One of the best ways to understand what breadth-first search (BFS) is, exactly, is by understanding what it is *not*. That is to say, if we compare BFS to DFS, it'll be much easier for us to keep them straight in our heads. So, let's refresh our memory of depth-first search before we go any further.

We know that *depth-first search* is the process of traversing down through one branch of a tree until we get to a leaf, and then working our way back to the "trunk" of the tree. In other words, implementing a DFS means traversing down through the subtrees of a binary search tree.


1
2
5
3
4
6
7
Depth-first search
• Traverse through left subtree(s) first, then traverse through the right subtree(s).
1
2
3
4
5
6
7
Breadth-first search
• Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

Depth-first search as compared to breadth-first search

Okay, so how does breadth-first search compare to that? Well, if we think about it, the only real *alternative* to traveling down one branch of a tree and then another is to travel down the tree section by section — or, level by level. And that's exactly what BFS is!

Breadth-first search involves search through a tree one level at a time.

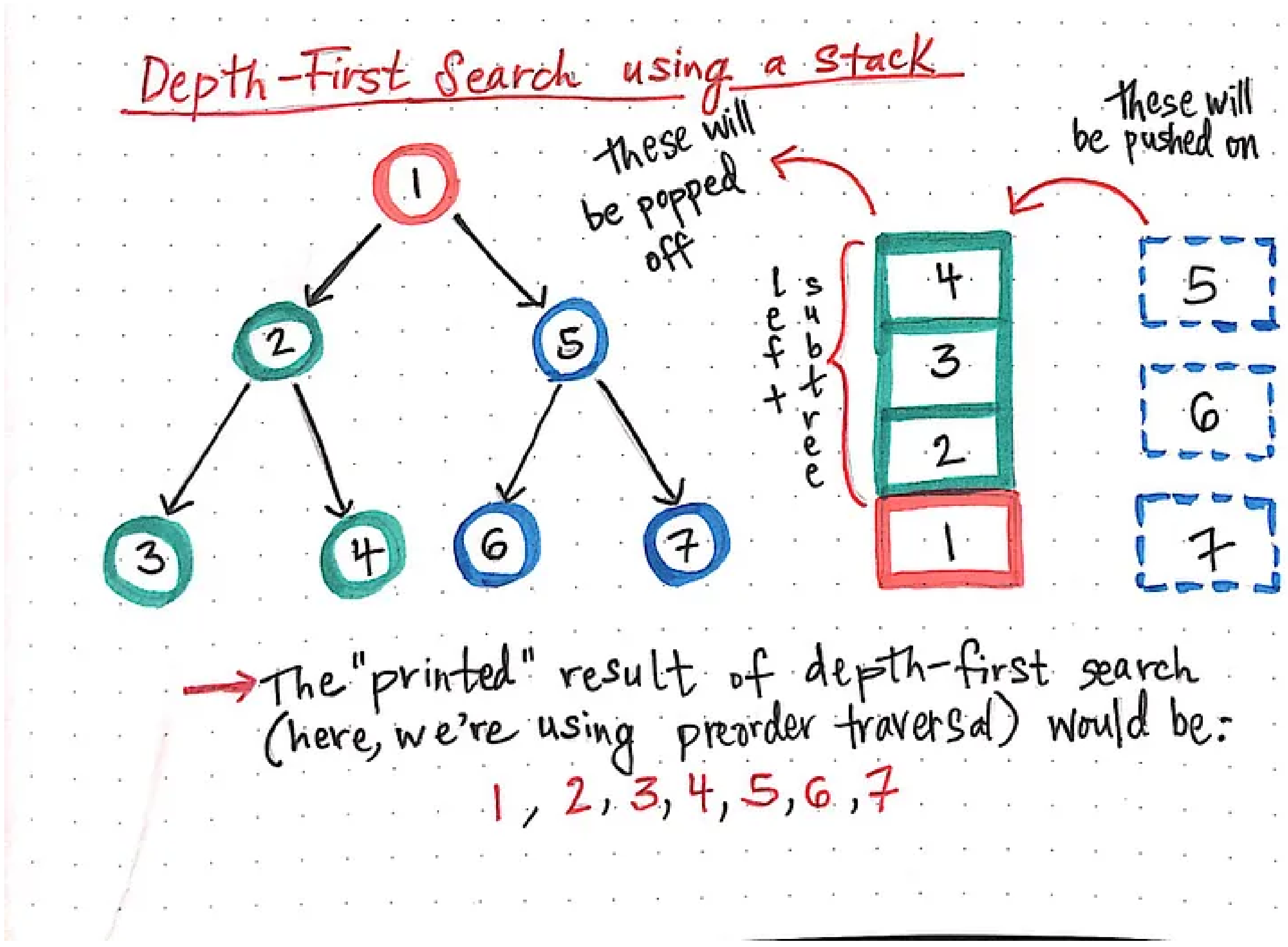
> We traverse through one entire level of children nodes first, before moving on to traverse through the grandchildren nodes. And we traverse through an entire level of grandchildren nodes before going on to traverse through great-grandchildren nodes.

Alright, that seems fairly clear. What else differentiates the two different types of tree traversal algorithms? Well, we've already covered the differences in the procedures of these two algorithms. Let's think about the other important aspect we haven't talked about yet: *implementation*.

First, let's start with what we know. How did we go about implementing depth-first search last week? You might remember that we learned three different methods — inorder, postorder, and preorder — of searching through a tree using DFS. Yet there was something super cool about how similar these three implementations; they could each be employed using *recursion*. We also know that, since DFS can be written as a recursive function, they can cause the call stack to grow to be as big as the longest path in the tree.

However, there was one thing I left out last week that seems good to bring up now (and maybe it's even a little bit obvious!): the call stack actually implements a stack data structure. Remember those? We learned about stacks awhile ago, but here they are again, showing up all over the place!

The really interesting thing about implementing depth-first search using a stack is that as we traverse through the subtrees of a binary search tree, each of the nodes that we "check" or "visit" gets added to the stack. Once we reach a leaf node — a node that has no children — we start popping off the nodes from the top of the stack. We end up at the root node again, and then can continue to traverse down the next subtree.


Depth-First Search using a Stack
these will be popped off
these will be pushed on
left subtree
1
2
5
3
4
6
7
The "printed" result of depth-first search (here, we're using preorder traversal) would be:
1, 2, 3, 4, 5, 6, 7

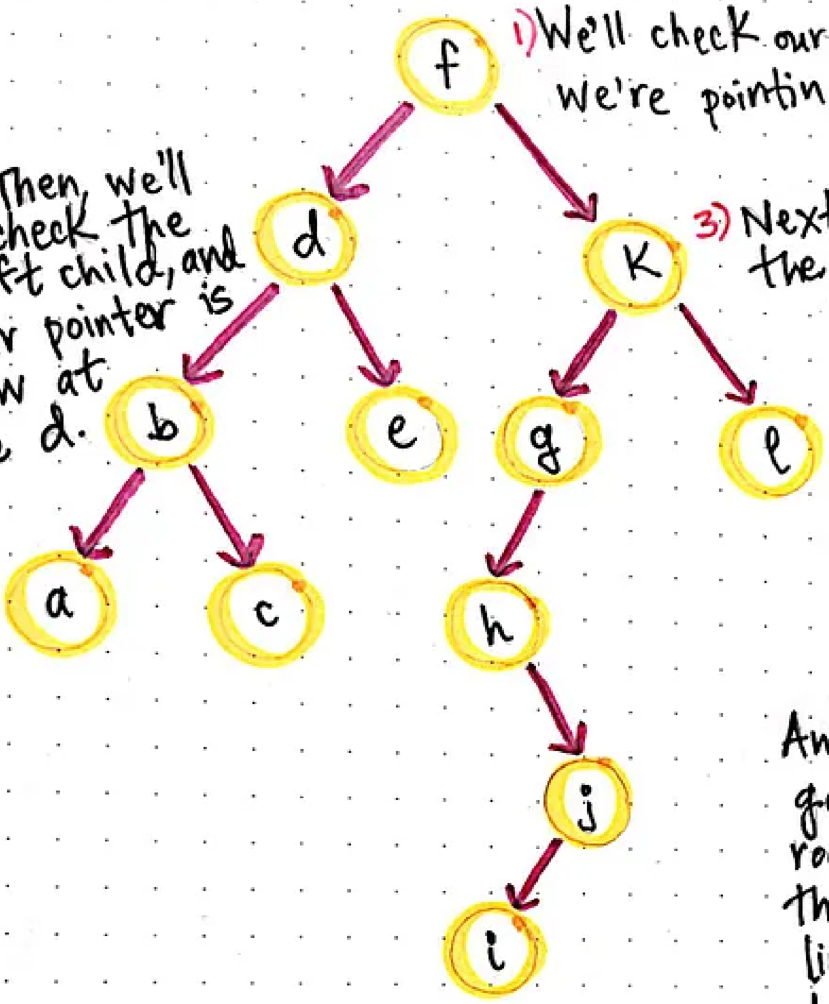
Implementing depth-first search using a stack data structure

In example DFS tree above, you'll notice that the nodes *2*, *3*, and *4* all get added to the top of the stack. When we get to the "end" of that subtree — that is to say, when we reach the leaf nodes of *3* and *4* — we start to pop off those nodes from our stack of "nodes to visit". You can see what will eventually happen with the right subtree: the nodes to visit will be pushed onto the call stack, we'll visit them, and systematically pop them off of the stack.

Eventually, once we've visited both the left and the right subtrees, we'll be back at the root node with nothing left to check, and our call stack will be empty.

So, we should be able to use a stack structure and do something similar with our BFS implementation…right? Well, I don't know if it will *work*, but I think it'll be helpful to at least start off by drawing out the algorithm we want to implement, and seeing how far we can get with it.

Let's try:



1) We'll check our root node first; we're pointing to node f.

2) Then, we'll check the left child, and our pointer is now at node d.

3) Next, we want to check the right child node.

But wait!

How?!

Node d and node K aren't connected by any link/edge.

And, we can't even go back to f, the root node, because there is no reverse link from the child d to the parent f!

So, how do we solve the problem of traversing in level order?

→ We need to keep a reference to all the children nodes of every node that we visit; otherwise, we'll never be able to go back to them later and "check" or "visit" them.

QUEUES TO THE RESCUE!

Attempting to traverse through a tree using BFS

Okay, so we have a graph to the left that we implemented DFS on last week. How might we use a BFS algorithm on it, instead?

Well, to start, we know that we want to check the root node first. That's the only node we'll have access to initially, and so we'll be "pointing" to node *f*.

Alright, now we'll have to check the children of this root node.

We want to check one child after another, so let's go to the left child first — node *d* is the node we're "pointing" to now (and the only node we have access to).

Next, we'll want to go to the right child node.

Uh oh. Wait, the root node isn't even available to us anymore! And we can't move in reverse, because binary trees don't have reverse links! How are we going to get to the right child node? And…oh no, the left child node *d* and the right child node *k* aren't linked at all. So, that means it's impossible for us to jump from one child to another because we don't have access to anything except for node *d*'s children.

Oh dear. We didn't get very far, did we? We'll have to figure out a different method of solving this problem. We need to figure out some way of implementing a tree traversal that will let us walk the tree in ***level order***. The most important thing we need to keep in mind is this:

> We need to keep a reference to all the children nodes of every node that we visit. Otherwise, we'll never be

able to go back to them later and visit them!

The more that I think about it, the more I feel like it's as though we want to keep a *list* of all the nodes we still need to check, isn't it? And the moment I want to keep a list of something, my mind immediately jumps to one data structure in particular: a queue, of course!

Let's see if queues can help us out with our BFS implementation.

Queues to the rescue!

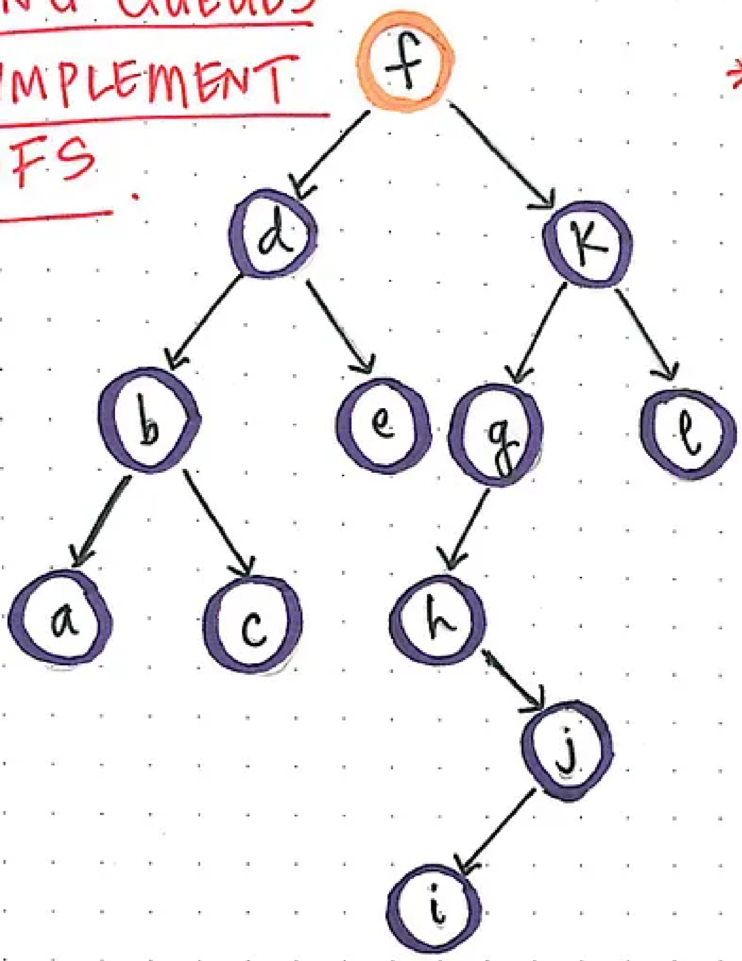
As it turns out, a major difference in depth-first search and breadth-first search is the data structure used to implement both of these very different algorithms.

While DFS uses a stack data structure, BFS leans on the queue data structure. The nice thing about using queues is that it solves the very problem we discovered earlier: it allows us to keep a reference to nodes that we want to come back to, even though we haven't checked/visited them yet.

We add nodes that we have discovered — but not yet visited — to our queue, and come back to them later.

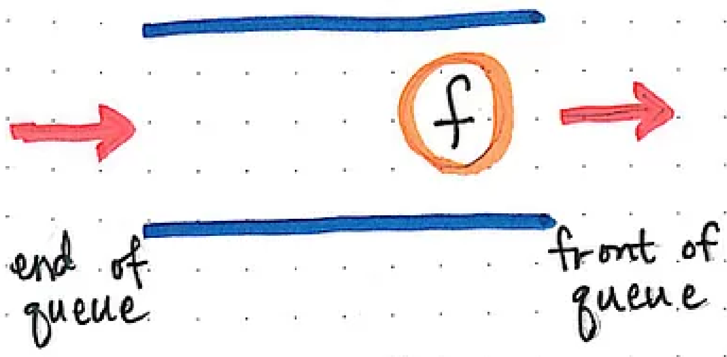
A common term for nodes that we add to our queue is **discovered nodes**; a discovered node is one that we add to our queue, whose location we know, but we have yet to actually visit. In fact, this is exactly what makes a queue the perfect structure for solving the BFS problem.

USING QUEUES TO IMPLEMENT BFS



*Discovered node: a node we add to our queue, whose location we know, but we haven't actually visited yet.

→ To start, the root node will be the only discovered node.



*Once we have at least one node enqueued (and our queue isn't empty) we can start visiting the nodes and enqueuing their children.

Steps to BFS :

1) Visit node f (print its value).
2) Enqueue left child.
3) Enqueue right child.

} Continue until our queue is totally empty!

Using queues to implement breadth-first search

In the graph to the left, we start off by adding the root node to our queue, since that's the only node we ever have access to (at least, initially) in a tree. This means that **the root node is the only discovered node to start**.

Once we have at least *one* node enqueued, we can start the process of visiting nodes, and adding references to their children nodes into our queue.

Okay, so this all might sound a little bit confusing. And that's okay! I think it'll be a lot easier to grasp if we break it down into simpler steps.

For every node in our queue — always starting with the root node — we'll want to do three things:

1. **Visit** the node, which usually just means printing out its value.
2. **Add** the node's **left child** to our queue.
3. **Add** the node's **right child** to our queue.

Once we do these three things, we can remove the node from our queue, because we don't need it anymore! We basically need to keep doing this repeatedly until we get to the point where our queue is empty.

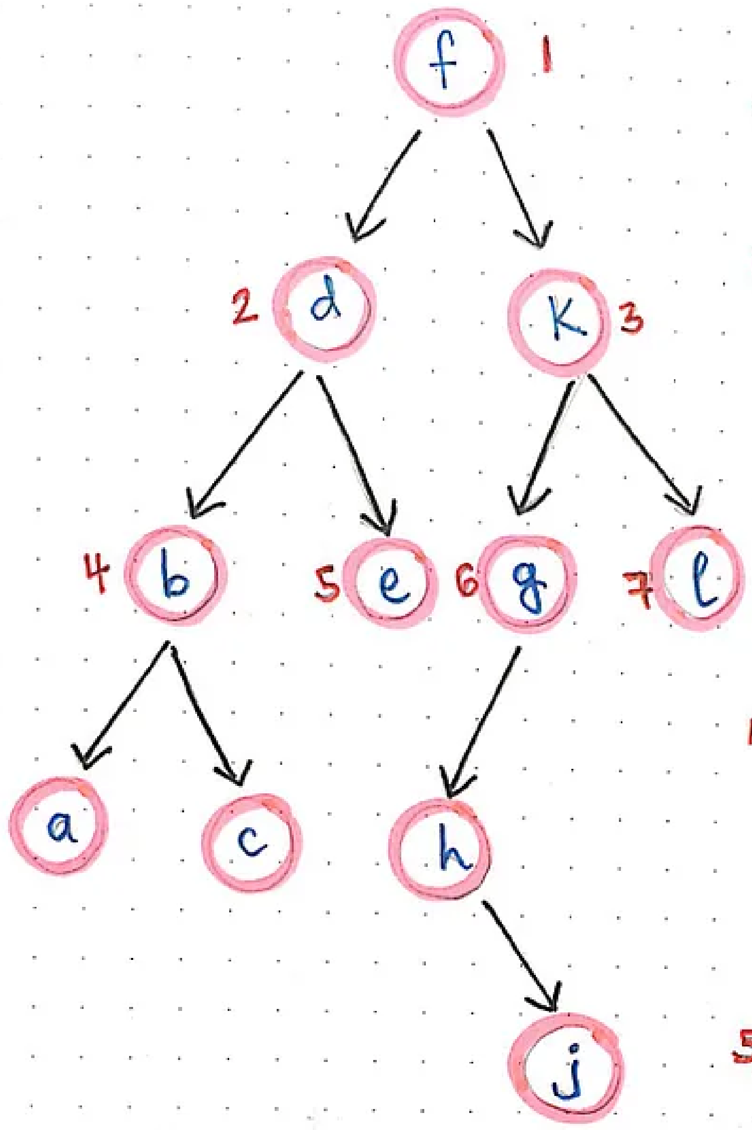
Okay, let's look at this in action!

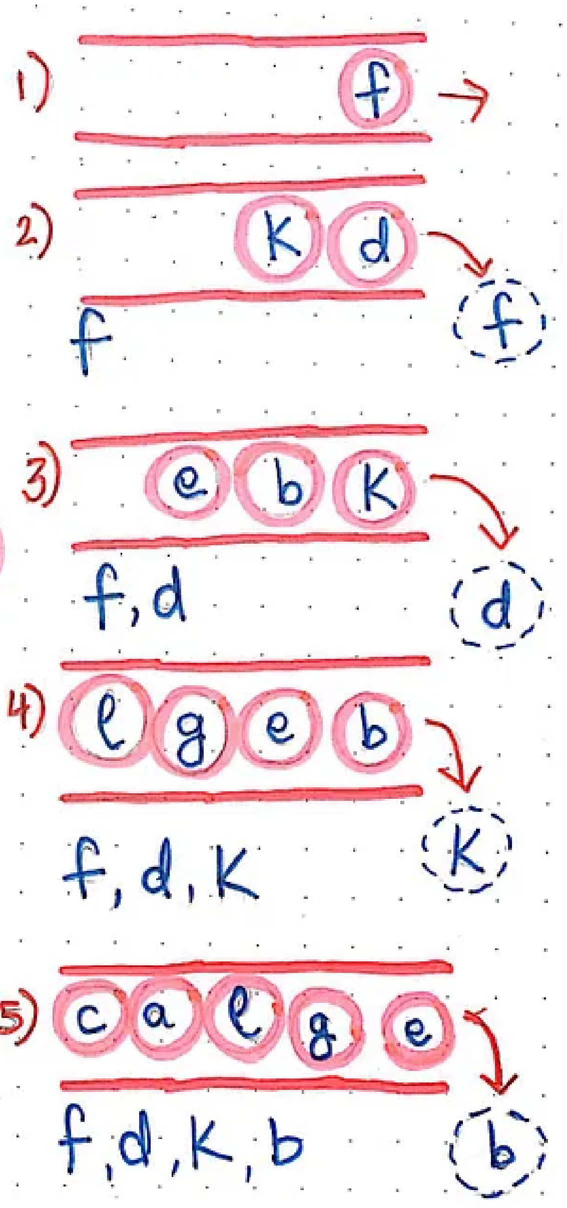
In the graph below, we start off with the root node, node *f*, as the only discovered node. Remember our three steps? Let's do them now:

1. We'll visit node *f* and print out its value.

2. We'll enqueue a reference to its left child, node *d*.
3. We'll enqueue a reference to its right child, node *k*.

And then, we'll remove node *f* from our queue!





But what about space-time complexity?

→ Visiting a node (reading its data and enqueuing its children) takes constant time. Since we are only visiting each node once, the time it will take us to use a BFS is O(n), where n is the number of nodes.

→ The space complexity depends on the size of the queue at its worst, which could be up to O(n) also.

Growing the queue structure in a breadth-first search implementation

The next node at the front of the queue is node *d*. Again, same three steps here: print out its value, add its left child, add its right child, and then remove it from the queue.

Our queue now has references to nodes *k*, *b*, and *e*. If we keep repeating this process systematically, we'll notice that we're actually traversing the graph and printing out the nodes in *level order.* Hooray! That's exactly what we wanted to do in the first place.

> The key to this working so well is the very nature of the queue structure. Queues follow the first-in, first-out (FIFO) principle, which means that whatever was enqueued first is the first item that will be read and removed from the queue.

Lastly, while we're on the topic of queues, it's worth mentioning that the space-time complexity of a BFS algorithm is *also* related to the queue that we use to implement it — who knew that queues would come back to be so useful, right?

The time complexity of a BFS algorithm depends directly on how much time it takes to visit a node. Since the time it takes to read a node's value and enqueue its children doesn't change based on the node, we can say that visiting a node takes constant time, or, *O(1)* time. Since we only visit each node in a BFS tree traversal exactly once, the time it will take us to read every node really just depends on how many nodes there are in the tree! If our tree has 15 nodes, it will take us O(15); but if our tree has 1500 nodes, it will take us O(1500). Thus, the time complexity of a breadth-first search algorithm takes linear time, or **O(n)**, where ***n*** is the number of nodes in the tree.

Space complexity is similar to this, has more to do with how much our queue grows and shrinks as we add the nodes that we need to check to it. In the worst case situation, we could potentially be enqueuing all the nodes in a tree if they are all children of one another, which means that we could possibly be using as much memory as there are nodes in the tree. If the size of the queue can grow to be the number of nodes in the tree, the space complexity for a BFS algorithm is also linear time, or **O(n)**, where ***n*** is the number of nodes in the tree.

This is all well and good, but you know what I'd really like to do right about now? I'd like to actually *write* one of these algorithms! Let's finally put all of this theory into practice.

Coding our first breadth-first search algorithm

We've made it! We're finally going to code our very first BFS algorithm. We did a little bit of this last week with DFS algorithms, so let's try to write a breadth-first search implementation of this, too.

You might remember that we wrote this in vanilla JavaScript last week, so we'll stick with that again for consistency's sake. In case you need a quick refresher, we decided to keep it simple, and write our node objects as Plain Old JavaScript Objects (POJO's), like this:

```
node1 = {
  data: 1,
  left: referenceToLeftNode,
  right: referenceToRightNode
};
```

Okay, cool. One step done.

But now that we know about queues and are certain that we'll need to use one to implement this algorithm…we should probably figure out how to do that in JavaScript, right? Well, as it turns out, it's really easy to create a queue-like object in JS!

We can use an array, which does the trick quite nicely:

```
// Create an empty queue.
var queue = [];
// Add values to the end of the queue.
queue.push(1);                      // queue is now [1]
queue.push(2);                      // queue is now [1, 2]
// Remove the value at the top of the queue.
var topOfQueueValue = queue.shift();
console.log(topOfQueueValue)        // returns 1
// The queue now has just one element in it.
console.log(queue)                  // returns [2]
```

queues.js hosted with ❤ by GitHub view raw

If we wanted to make this a bit fancier, we could also probably create a

Okay, let's write this puppy! We'll create a `levelOrderSearch` function, that takes in a `rootNode` object.

```javascript
function levelOrderSearch(rootNode) {
  // Check that a root node exists.
  if (rootNode === null) {
    return;
  }

  // Create our queue and push our root node into it.
  var queue = [];
  queue.push(rootNode);

  // Continue searching through as queue as long as it's not empty.
  while (queue.length > 0) {
    // Create a reference to currentNode, at the top of the queue.
    var currentNode = queue[0];

    // If currentNode has a left child node, add it to the queue.
    if (currentNode.left !== null) {
      queue.push(currentNode.left)
    }
    // If currentNode has a right child node, add it to the queue.
    if (currentNode.right !== null) {
      queue.push(currentNode.right)
    }
    // Remove the currentNode from the queue.
    queue.shift()
  }

  // Continue looping through the queue until it's empty!
}
```

level_order_search.js hosted with ❤ by GitHub view raw

Awesome! This is actually...fairly simple. Or at least, much simpler than I was expecting it to be. All we're doing here is using a `while` loop to continue doing those three steps of checking a node, adding its left child, and adding its right child. We continue iterating through the `queue` array until everything has been removed from it, and its length is `0`.

Amazing. Our algorithm expertise has skyrocketed in just a day! Not only do we know how to write recursive tree traversal algorithms, but now we also know how to write iterative ones. Who knew that algorithmic searches could be so empowering!

Resources

There is still a lot to learn about breadth-first search, and when it can be useful. Luckily, there are tons of resources that cover information that I couldn't fit into this post. Check out a few of the really good ones below.

1. DFS and BFS Algorithms using Stacks and Queues, Professor Lawrence L. Larmore
2. The Breadth-First Search Algorithm, Khan Academy
3. Data Structure — Breadth First Traversal, TutorialsPoint
4. Binary tree: Level Order Traversal, mycodeschool
5. Breadth-First Traversal of a Tree, Computer Science Department of Boston University

Programming JavaScript Computer Science Code Algorithms