Data Structures: Traversing Trees

Maria Betances Follow
Jan 25, 2018 · 5 min read

Trees are nonlinear data structures in that they are organized through relationships or hierarchies. This allows us to traverse them in multiple ways. To clarify, tree traversal refers to the process of visiting each individual node exactly once. For our traversal, we will focus on binary trees, which are trees that have a max of two children. You can check out my earlier post on binary search trees for more information in the link below.

Data Structures: Binary Search Trees Explained
Binary search trees allow us to efficiently store and update, in sorted order, a dynamically changing dataset. When…
medium.com

How to Traverse Trees?

There are two main approaches to tree traversal:

1. Breadth-first

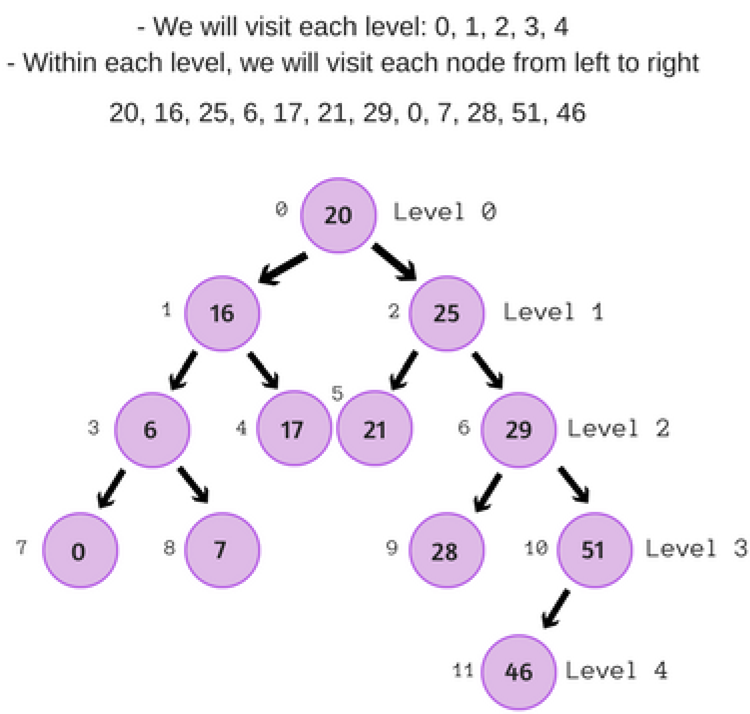
2. Depth-first

Breadth-first Traversal

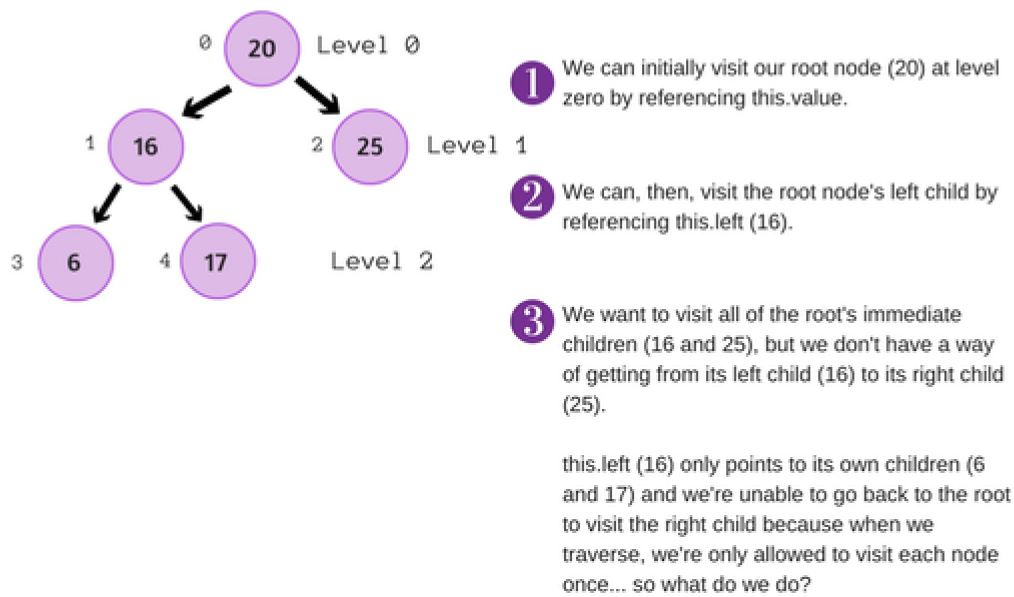
The breadth-first approach is leveraged when the levels of a tree have some meaning behind them. In breadth-first, you visit each level in your tree from top to bottom until you've traversed the entire tree. In each level, you visit each node, once, from left to right. Let's use the constructor from my earlier post to build our binary search tree (BST).

```
function BinarySearchTree (val) {
  this.value = val;
  this.left = null;  // the left child node
  this.right = null; // the right child node
}

let newTree = BinarySearchTree(20); // 20 node becomes our root

newTree.insert(25)
newTree.insert(21)
newTree.insert(16)
...
```

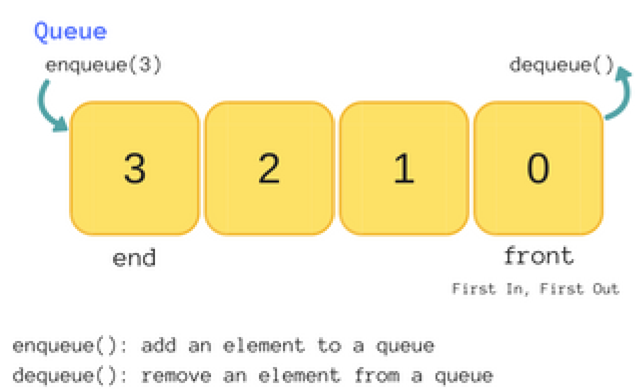
Let's walkthrough what happens in breadth-first leveraging `newTree`. Please note that breadth-first is less useful in BSTs as levels don't have any essential meaning, but to simplify this review we'll leverage the same BST for all examples:



To solve for breadth-first, we will require a data structure that will allow us to keep track of the relationships that exists between each node. By simply going down the binary tree, we will lose access to the parent and child relationships that exists. This refers to the `this.left` and `this.right` connections that have been established. For example:



We will need to use the queue data structure to be able to visit every node from left to right in every level. A queue is a linear data structure that follows the First In, First Out (FIFO) rule. You can think of a queue as consumers waiting in line at your local Starbucks; it's on a first come, first served basis. Consumers are served in the order that they are waiting in line. Those at the start of the line will be served before those at the end of the line.



Coding our Breadth-first Traversal

To code our breadth-first traversal, let's assume that we want to create an array that holds all the values in our binary search tree in level order. Let's write out a callback function, which is a function that is passed on to another function as a parameter, that will take in a value and push it to an array. We will pass this callback function to our breadth-first method later on.

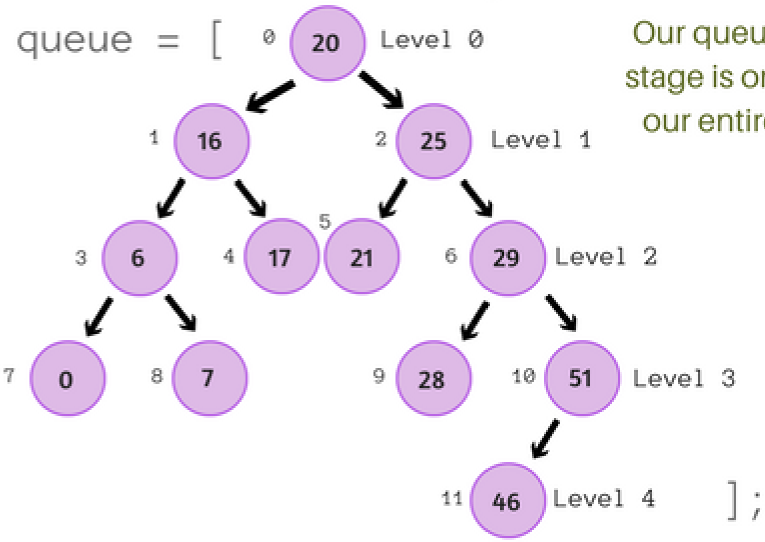
```
let levelOrderArray = [];

// ES6 syntax
const pushOrderNodes = (num) => {
  levelOrderArray.push(num);
};
```

We will now work on coding a simple breadth-first method that can take a callback function as a parameter (we will not account for error handling). We will leverage our constructor's prototype to create this method in order to save memory.

```
BinarySearchTree.prototype.traverseBreadthFirst = function (func) {
  let queue = [this];

  while (queue.length) {
    let currentNode = queue.shift();

    if (currentNode.left) {
      queue.push(currentNode.left)
    }

    if (currentNode.right) {
      queue.push(currentNode.right)
    }

    func(currentNode.value)
  }
};
```

Let's breakdown the first couple of loops in this code to understand what's really going on…

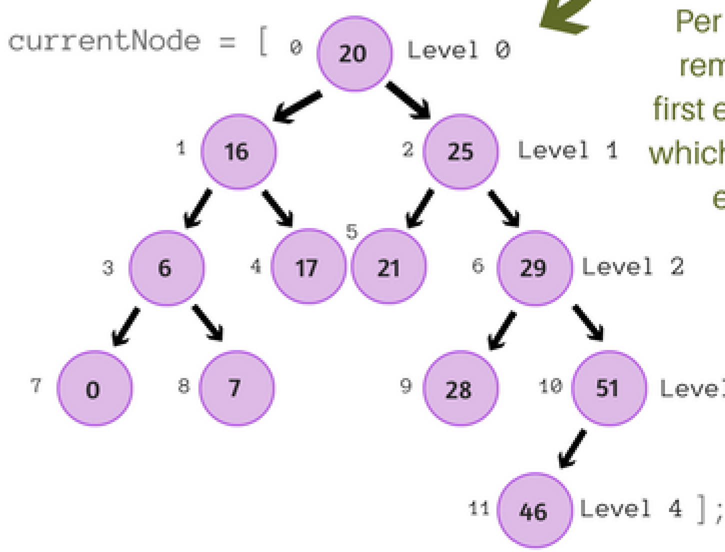
```
let queue = [this];
```



Our queue's length at this stage is one as it's holding our entire BST in index 0.

2

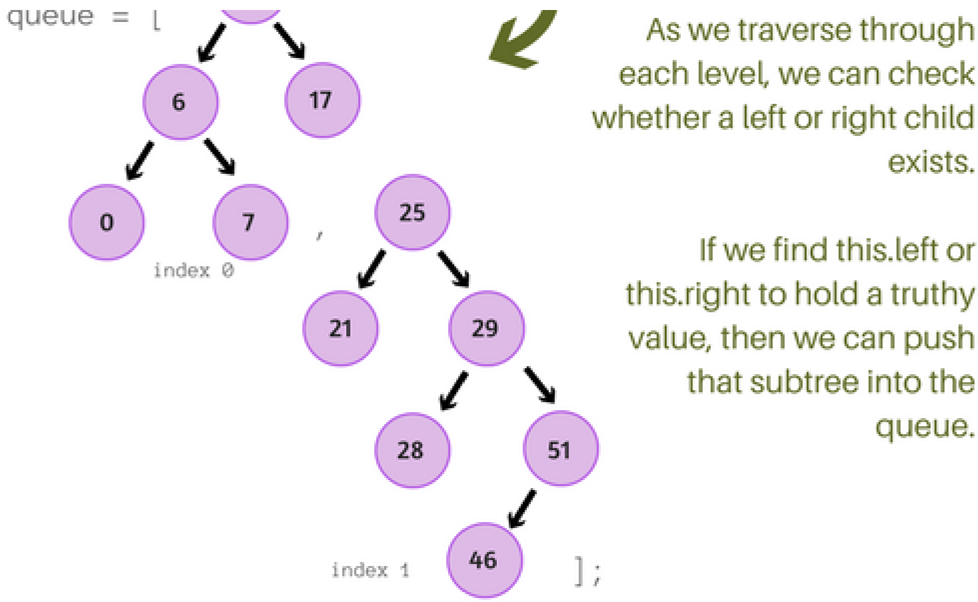
```
while (queue.length) {
   let currentNode = queue.shift();
```

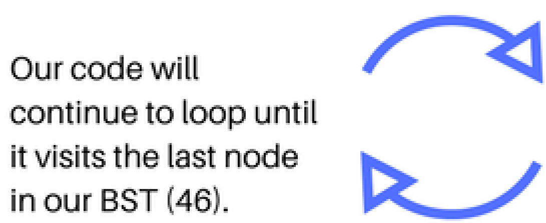


Per the FIFO rule, we will remove and process our first element in our queue, which is currently our only element in our queue.

3

```
if (currentNode.left) {
   queue.push(currentNode.left) };
if (currentNode.right) {
   queue.push(currentNode.right) };
```





Let’s now pass our callback function `pushOrderNodes` into our `traverseBreadthFirst` method. This will traverse our `newTree` and execute the passed in function `pushOrderNodes` for each node once.

```
newTree.traverseBreadthFirst(pushOrderNodes)
```

```
[ 20, 16, 25, 6, 17, 21, 29, 0, 7, 28, 51, 46 ]
```

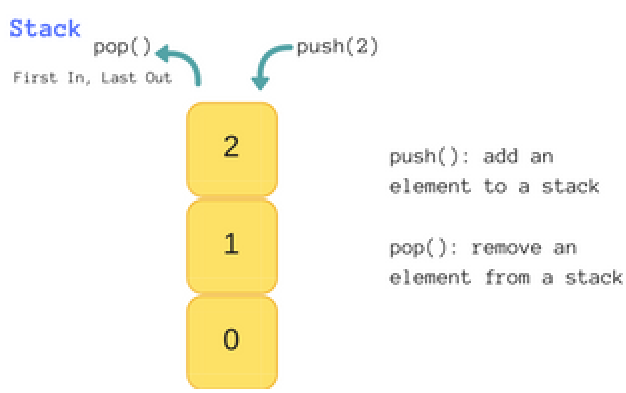
Our levelOrderArray now holds all our nodes in breadth-first order.

Depth-first Traversal

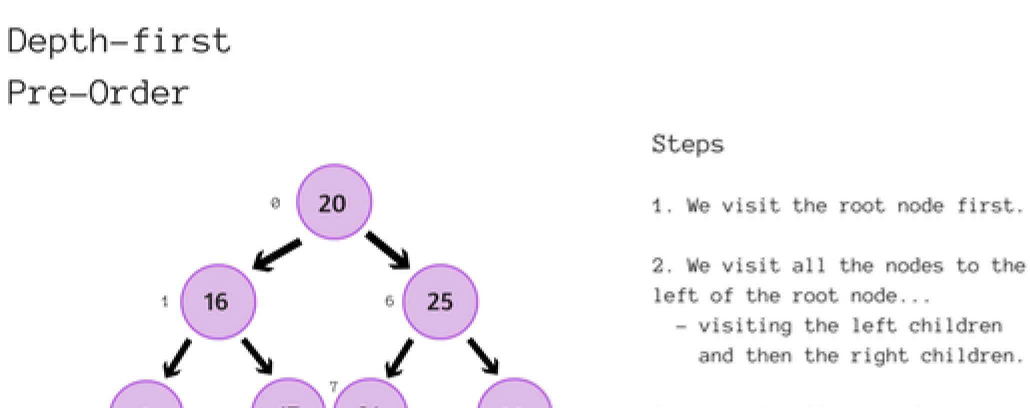
There are three types of depth-first traversal:

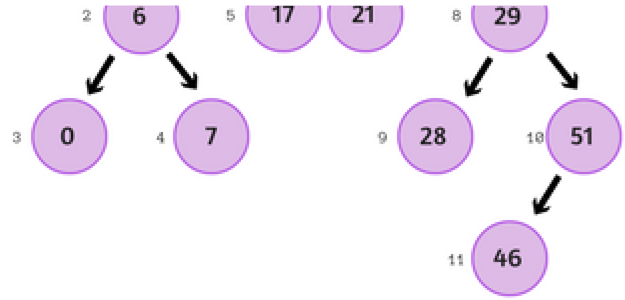
1. **pre-order:** visit the parent, then all the left children, and then all the right children.
2. **in-order:** visit the left child, then the parent, and then the right child. This approach is useful for BSTs as it traverses the nodes in sorted order.
3. **post-order:** visit the left child, then the right child, and then the parent.

For depth-first traversal, we will require a stack data structure, which follows a Last In, First Out (LIFO) rule. You can think of a stack as a stack of newspapers; the newspaper on top was the last added to the pile, but the first to be grabbed by a customer for purchase.



We can write one method to cover the three different types of depth-first traversal. Let's review in detail what our method should do:



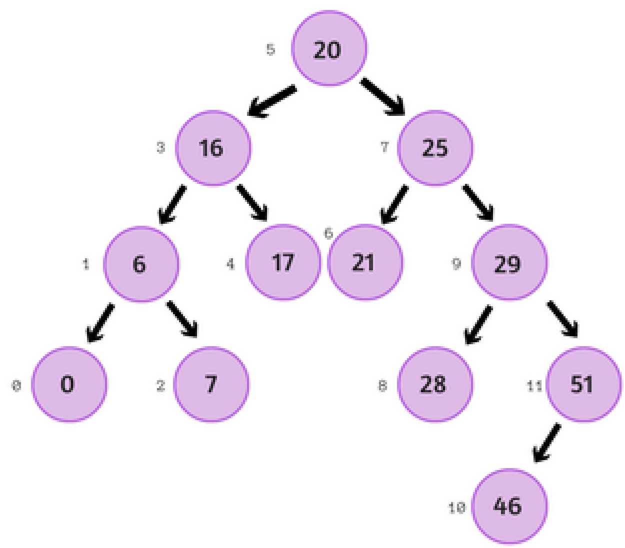


3. We visit all the nodes to the right of the root node...
   - visiting the left children and then the right children.

Order of traversal
20, 16, 6, 0, 7, 17, 25, 21, 29, 28, 51, 46

parent, left child, right child

Depth-first
In-Order



Steps

1. We visit the left most child node.

2. We visit the parent node.

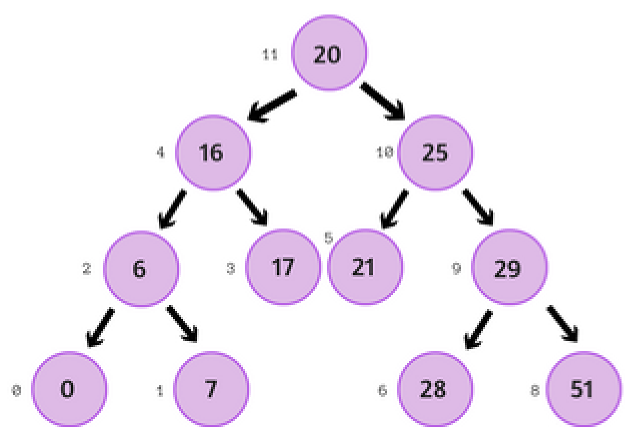
3. We visit that parent's right node.

4. We backtrack and repeat steps 1 through 3 again.

Order of traversal
0, 6, 7, 16, 17, 20, 21, 25, 28, 29, 46, 51

left child, parent, right child

Depth-first
Post-Order



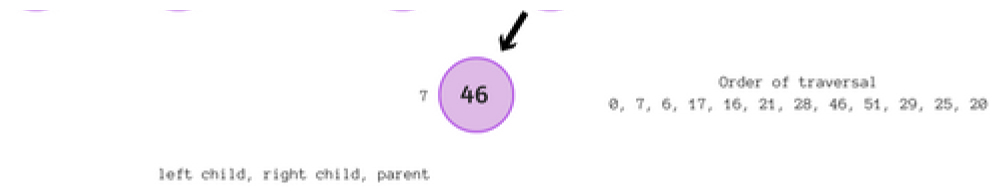
Steps

1. We visit the left most child node.

2. We visit its parent's right node.

3. We visit the parent node.

4. We backtrack and repeat steps 1 through 3 again.



Coding our Depth-first Traversal

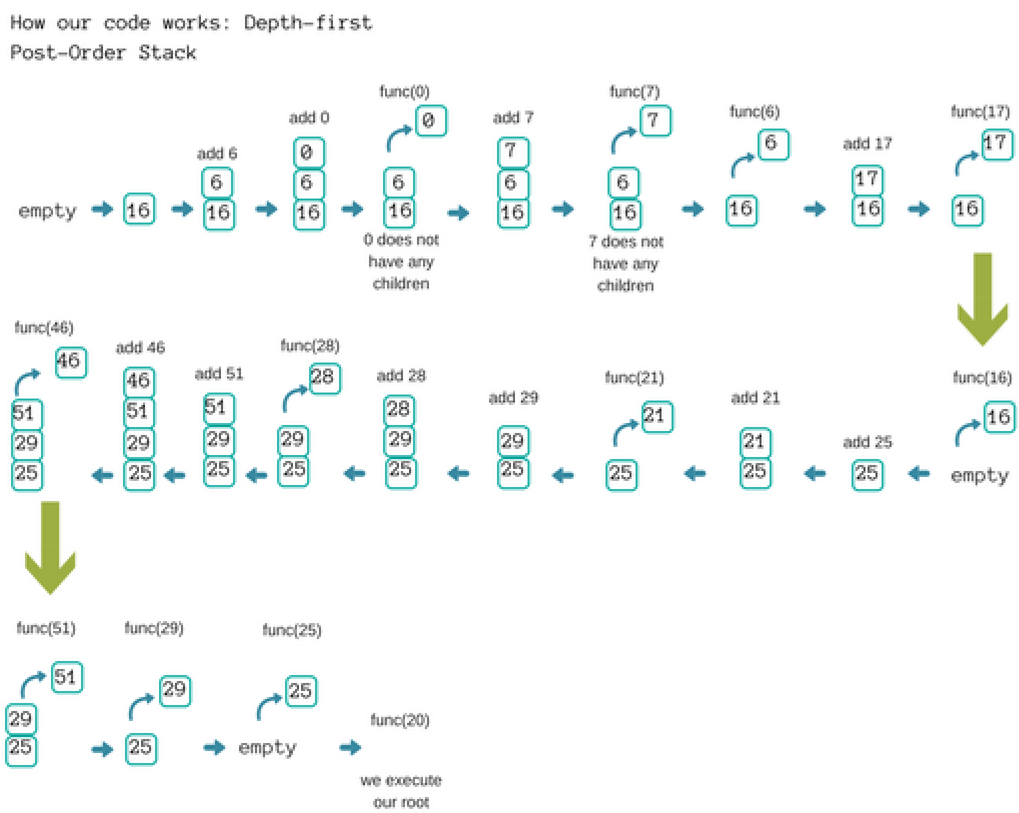
We will leverage BinarySearchTree's prototype to create the `traverseDepthFirst` method. Our `traverseDepthFirst` method will take in a callback function, similar to our `traverseBreadthFirst` method. The callback function will manipulate or process each node.

```
BinarySearchTree.prototype.traverseDepthFirst = function (func, type)
{
  if (type === 'pre-order') {
    func(this.value)
  }

  if (this.left) {
    this.left.traverseDepthFirst(func, type)
  }

  if (type === 'in-order') {
    func(this.value)
  }

  if (this.right) {
    this.right.traverseDepthFirst(func, type)
  }

  if (type === 'post-order') {
    func(this.value)
  }
};
```

Let's leverage our callback function from earlier, `pushOrderNodes`.

```
newTree.traverseDepthFirst(pushOrderNodes, 'post-order')
```

Let's review how we're leveraging the stack data structure in the `traverseDepthFirst` method for post-order traversal:



Applications of Breadth-first and Depth-first

We have implemented some basic traversal methods and you might be wondering, how are these actually used in real-world applications? Breadth-first and depth-first traversal are leveraged in a number of ways: social networks recommending users you may know, online travel companies recommending flights for your trip, GPS navigation systems finding nearby locations, etc. Now let's celebrate for implementing tree traversal in JavaScript.

Programming Trees Tree Traversal JavaScript Data Structures