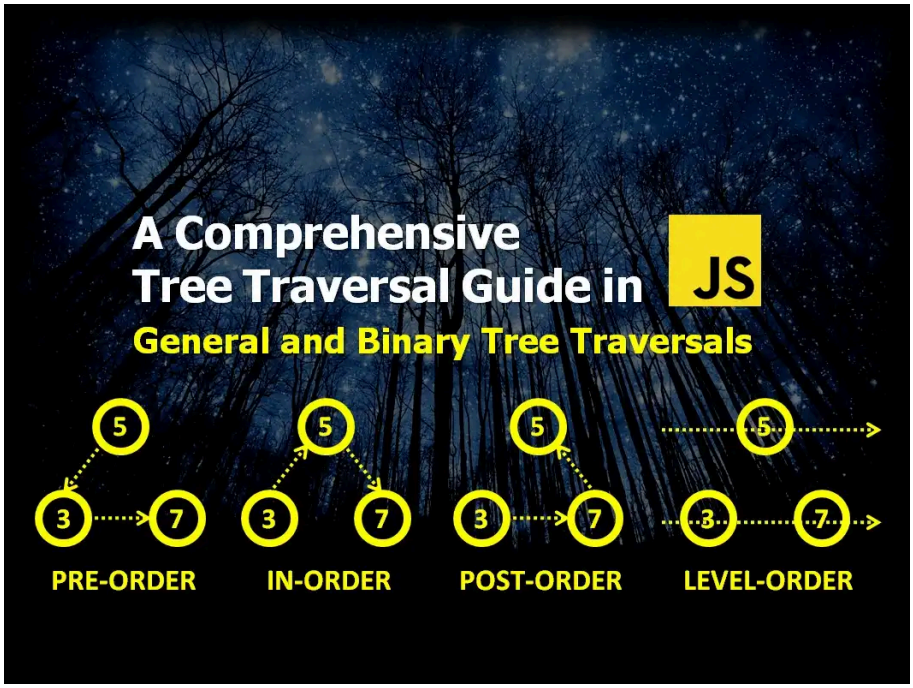
A Comprehensive Tree Traversal Guide in Javascript - General and Binary Tree Traversals



Trees are a fundamental data structure in computer science that are used to represent hierarchical relationships between data. They can be found in a variety of applications, such as file systems, database indexes, decision trees in artificial intelligence, and more.

Tree traversal is an algorithmic concept that plays a key role when working with the Trees. I'd also like to add these are not only exclusive to Trees, they are also applicable to Graphs. Even though in this article we will be only using Trees for the examples, this knowledge will still help with Graphs as Traversals are widely used in Graphs and are essential in solving problems.

"Traversing a tree" means visiting each node in the tree exactly once, in a specific order. Since Trees are non-linear data structures, there are a lot more directions and logic involved compared to linear data structures.

If we take a look at a linear data structure - let's say Arrays. We can simply visit every element by using a simple loop, where we have only 2 possible directions: either start from index zero to the last index or in reverse order - from last index to index zero. But when it comes to Trees, visiting each node / element takes more logic. This makes the traversing trees more complex, but this complexity also opens the door to many possibilities out of the box - something we couldn't have with linear data structures like Arrays.

The tone of this article is assuming you are at least familiar with the concept of Tree data structure, how General Tree and Binary Tree works. If that's not the case or you need a quick refreshment, I'd suggest you start from the links below in order, then come back and continue here later:

Deep Dive into Data structures using Javascript - Introduction to Trees

Deep Dive into Data structures using Javascript - General (Generic, N-ary) tree

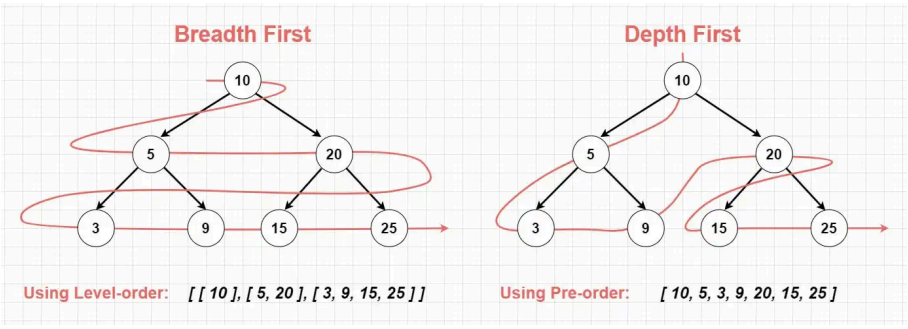
Deep Dive into Data structures using Javascript - Binary Search Tree

On another note, this article is intended to be comprehensive guide to the Tree traversals (including both General & Binary Trees). As a result, it is quite lengthy - almost at the size of a mini-book. If you're looking for a specific section, feel free to use the table of contents below to directly navigate to the part you are interested.

Types of Tree Traversals

We can categorize Tree traversals under 2 main categories:

- Breadth First (Level-order)
- Depth First (Pre-order, In-order, Post-order)



Breadth first

Breadth First is a type of tree traversal algorithm that explores a tree by visiting all the nodes in the same level before moving on to the next level. This approach is sometimes also referred to as Level-order traversal, since there is only one type of traversal related with it.

Level-order traversal

Level-order traversal algorithm starts at the root node of the tree and visits each level of the tree from left to right before moving on to the next level. This means that the algorithm will visit all the nodes at a given level before moving down to the next level.

To implement a Level-order traversal algorithm, we typically use a queue data structure. We start by enqueuing the root node, then dequeue it, visit the node, and enqueue its children (if any) from left to right. We repeat this process until the queue is empty.

Use cases for Level-order traversal:

- **Finding the shortest path between two nodes:** If you need to find the shortest path between two nodes in a tree, it can be used to traverse the tree level by level until the destination node is found. Since it visits all the nodes at a particular level before moving on to the next level, it guarantees that the shortest path between the nodes will be found.
- **Finding all the nodes at a particular level:** This can be helpful in scenarios where you need to perform a specific operation on all the nodes at a particular level, or if you need to count the number of nodes at a particular level.
- **Checking if a tree is balanced:** This can also be used to check if a binary tree is balanced, meaning that the height difference between the left and right subtrees of every node is at most one. By traversing the tree level by level, it can be used to determine the depth of each node and compare the depths of sibling nodes to check for balance.

Depth First

Depth First is a type of tree traversal algorithm that explores a tree by visiting the nodes in a specific order that emphasizes depth. There are three variations of Depth First traversal: Pre-order, In-order, and Post-order. Variations refers to the steps to take when visiting each subtree.

Now remembering these 3 type of orders can be tricky at the beginning - but there is a good memorizing tactic you can use. Before we go further, let's take a look at it. All it takes is just making a connection with the first words of the order variations: "PRE, IN, POST". Follow these 2 basic rules:

- Names of orders “PRE”, “IN” and “POST” refers to the position of when the root node will be visited.
- No matter the position of root, left will be always somewhere before the right.

When we see the order “PRE”, it means root will be placed as “PREVIOUS” to the left and right:

ROOT - Left - Right

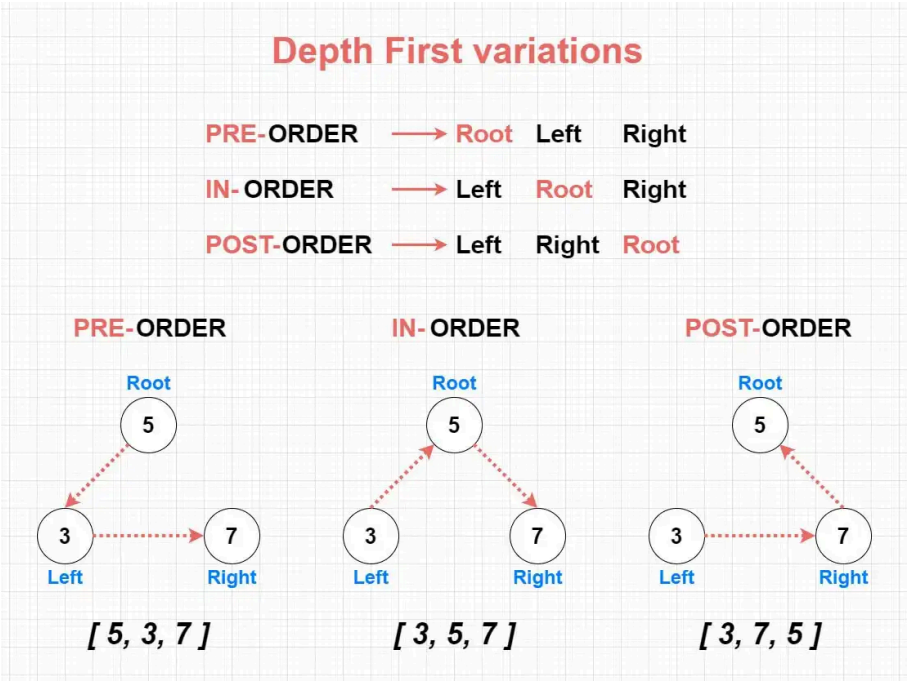
When we see the order “IN”, it means root will be placed “IN” between left and right:

Left - ROOT - Right

When we see the order “POST”, it means the root will be placed after the left and right:

Left - Right - ROOT

Take a look at the visual:



To implement Depth First traversal, we typically use a stack data structure. We start by pushing the root node onto the stack and then visit it. If the current node has a left child, we push it onto the stack and continue to the left child. If the current node doesn’t have a left child, we pop the top node from the stack and move to its right child. We continue this process until the stack is empty. Now let’s go through the each variation in more detail:

Pre-order traversal

It visits the root node first, then visits the left subtree, and finally visits the right subtree. This means that the algorithm will traverse the entire left subtree before moving on to the right subtree. The process is repeated recursively for each subtree until all nodes have been visited.

Use cases for Pre-order traversal:

- **Copying a binary tree:** Pre-order traversal can be used to copy a binary tree. The algorithm starts by copying the root node, then recursively copying the left and right subtrees.
- **Creating a prefix expression:** Pre-order traversal can be used to generate a prefix expression for an expression tree. The algorithm starts by visiting the root node and then recursively visiting the left and right subtrees.
- **Flattening a binary tree:** Pre-order traversal can be used to flatten a binary tree into a linked list. The algorithm starts by visiting the root node, then recursively flattening the left and right subtrees and attaching them to the root node.

In-order traversal

It visits the left subtree first, then visits the root node, and finally visits the right subtree. This means that the algorithm will visit the nodes in ascending order if the tree contains numbers or alphabetical order if the tree contains strings. The process is repeated recursively for each subtree until all nodes have been visited.

Use cases for In-order traversal:

- **Printing a binary search tree in order:** In-order traversal can be used to print the nodes of a binary search tree in ascending order. Since binary search trees are organized in a way that ensures that all the nodes in the left subtree are smaller than the root node, and all the nodes in the right subtree are larger than the root node, In-order traversal will visit the nodes in ascending order.

- **Evaluating mathematical expressions in infix notation:** In-order traversal can be used to evaluate mathematical expressions in infix notation. The algorithm starts by visiting the left subtree, then the root node, and finally the right subtree. By doing this in the correct order, the algorithm will ensure that the correct order of operations is followed.

Post-order traversal

It visits the left subtree first, then visits the right subtree, and finally visits the root node. This means that the algorithm will traverse the entire left and right subtrees before visiting the root node. The process is repeated recursively for each subtree until all nodes have been visited.

Use cases for Post-order traversal:

- **Deleting a binary tree:** Post-order traversal can be used to delete a binary tree. The algorithm starts by recursively deleting the left and right subtrees, then deletes the root node.
- **Calculating the height of a binary tree:** Post-order traversal can be used to calculate the height of a binary tree. The algorithm starts by recursively calculating the height of the left and right subtrees, then adding 1 to the larger height and returning it as the height of the current node.
- **Creating a postfix expression:** Post-order traversal can be used to generate a postfix expression for an expression tree. The algorithm starts by recursively visiting the left and right subtrees, then visiting the root node.

Breadth First vs Depth First

The main difference between them is:

- Breadth first traversal explores the tree level by level, visiting all the nodes in a given level before moving on to the next level.

- Depth first traversal explores the tree by visiting the nodes in a specific order that emphasizes depth.

Due to following different directions, both approaches have their advantages and disadvantages. Choosing which algorithm to use depends on the problem you're trying to solve and the characteristics of the tree you're working with. Let's take a closer look at each approach:

Breadth First Approach

Breadth first traversal (also known as level-order traversal) is a very systematic approach that guarantees that all nodes at a particular level are visited before moving on to the next level. This makes it useful in a variety of applications, such as finding the shortest path between two nodes, finding all the nodes at a particular level, or checking if a binary tree is balanced.

However, breadth first traversal can be less efficient than depth first traversal, especially if the tree is very deep. This is because breadth first traversal requires storing all the nodes at a particular level in memory before moving on to the next level. For very deep trees, this can quickly consume a lot of memory.

Depth First Approach

Depth first traversal algorithms (pre-order, in-order, and post-order) explore the tree by visiting the nodes in a specific order that emphasizes depth. This approach is often more memory-efficient than breadth first traversal, as it only requires storing the path from the root to the current node. It can also be more flexible than breadth first traversal, as it allows you to prioritize certain branches of the tree or visit nodes in a specific order.

However, if the tree is in an unbalanced state and you are using a recursive approach - you may face some issues. Because recursive approach tends to prioritize visiting nodes on one side of the tree before moving on to the other side. This can result in deeper nodes being visited later in the process, which could impact the efficiency of certain algorithms that rely on visiting nodes in a specific order.

In this case, it may be more appropriate to use an iterative approach or breadth first search. With the iterative approach using explicit stack implementation, it is possible to alternate between left and right subtrees, which can help address this issue.

But if the tree is balanced, recursive approach can be a good choice, because it has a space complexity of O(log n) and can be implemented more simpler than an iterative approach or Level-order traversal.

When to use Breadth First or Depth First

So, which traversal algorithm should you use? It depends on the problem you're trying to solve and the characteristics of the tree you're working with.

- Use breadth first traversal if you need to find the shortest path between two nodes, find all the nodes at a particular level, or check if a binary tree is balanced.
- Use depth first traversal if you want to prioritize certain branches of the tree or visit nodes in a specific order.
- If memory usage is a concern, depth first traversal may be a better choice for very deep trees. If the tree is very wide, however, breadth first traversal may be more memory-efficient.

At the end, the choice of which traversal algorithm to use depends on the specifics of your problem and your data structure.

Using Recursive vs Iterative Methods for Traversals

Recursive and iterative methods are two commonly used approaches for implementing tree traversals. Each method has its own strengths and weaknesses, and the choice of which to use can depend on various factors, such as the specific traversal algorithm, the size and structure of the tree, and the programming language and platform being used.

Recursive Traversal

Recursive traversal involves calling a function recursively on the tree nodes to visit them in a specific order. For example, in pre-order traversal, we visit the current node, then recursively visit the left and right subtrees. Recursive traversal can be simple to implement and can make the code more readable and concise. It can also be more intuitive to think about traversing a tree recursively, especially when the traversal algorithm involves backtracking.

On the other hand, recursive traversal can be inefficient in terms of memory usage, as it creates a new function call stack frame for each node visited. If the tree is very deep or unbalanced, this can lead to a stack overflow or other memory-related issues.

Iterative Traversal

Iterative traversal involves using a stack or queue data structure to keep track of the nodes to visit. For example, in pre-order traversal, we start with the root node and push it onto the stack. We then repeatedly pop a node from the stack, visit it, and push its right and left subtrees onto the stack in that order. This continues until the stack is empty. Iterative traversal can be more memory-efficient than recursive traversal, as it avoids creating a new stack frame for each node visited. It can also be more scalable, as it can handle large trees more easily.

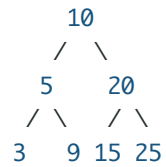
On the other hand, iterative traversal can be more complex to implement and can make the code less readable and harder to debug. It can also be harder to reason about the order in which nodes are visited, especially if the traversal algorithm involves backtracking.

How does Recursion includes Automatic Backtracking with Call Stack

Recursion is a technique in programming where a function calls itself, and it plays a significant role in tree traversals. When the function is called, the computer stores all the necessary data to complete the task in the call stack. As the function recursively calls itself, each call adds a new layer to the stack, creating a stack frame.

When the function encounters a base case, the function starts to resolve the stack from the last call to the first call, each call returning a value to the previous call. This process is called backtracking. In the case of tree traversals, we use backtracking to go back to the previous node after traversing its children.

For example, let's say we have a binary search tree that looks like this:



If we print the values with pre-order, we will get: "10, 5, 3, 9, 20, 15, 25".

When we start traversing the tree using recursion, we begin with node 10. We then traverse the left side of the tree, which takes us to node 5, then to node 3, and finally to node 9. Since node 9 doesn't have a left child, we backtrack to node 3 and then to node 5. We then move to the right side of the tree, which takes us to node 20. We traverse the left side of the right subtree, which takes us to node 15, and finally to node 25. Since node 25 doesn't have any children, we backtrack to node 15, then to node 20, and finally to the root node, node 10. This way, we have traversed the whole tree in pre-order.

This process of recursive function calls and backtracking is handled by the call stack. When we call a function, a new frame is added to the stack, and when a function returns, the frame is popped off the stack. By keeping track of the frames on the stack, we can backtrack through the recursive calls and effectively traverse the tree.

Overall, recursion and backtracking allow us to easily traverse trees and other complex data structures in a concise and effective way. But we must be careful to avoid infinite loops and excessive stack usage, which can cause our programs to crash.

Implementation of Tree traversals

In this section, we will explore the iterative and recursive implementations of General and Binary tree traversals, and see how they can be used to process tree nodes efficiently.

General Tree Traversal Implementations in Javascript

Now we will go through the iterative and recursive implementations of the General tree traversals: level-order, pre-order, and post-order. You might be thinking why is there no in-order traversal included.

Since General (n-ary) trees do not have a fixed number of children, there is no such “standard” definition of applying in-order traversal to them. Even though there are ways to specify in-order traversal definition for a General tree, but still it is not something that is often used for them in practice.

For example, you might stumble upon a question like “N-ary Tree Inorder Traversal”. I won’t be going through it’s details here, but to save you some confusion I’d like to point out in such cases problem refers to a “specific definition” of an in-order traversal for a specific tree - not to a “standard definition”. A specific definition could be: “Traverse all children except the last one (as a LEFT side placement), then visit ROOT, then visit the last child (as a RIGHT side placement)“. Hope this gave you a clear understanding.

Big O for the iterative and recursive traversal methods for General trees are:

Time complexity:

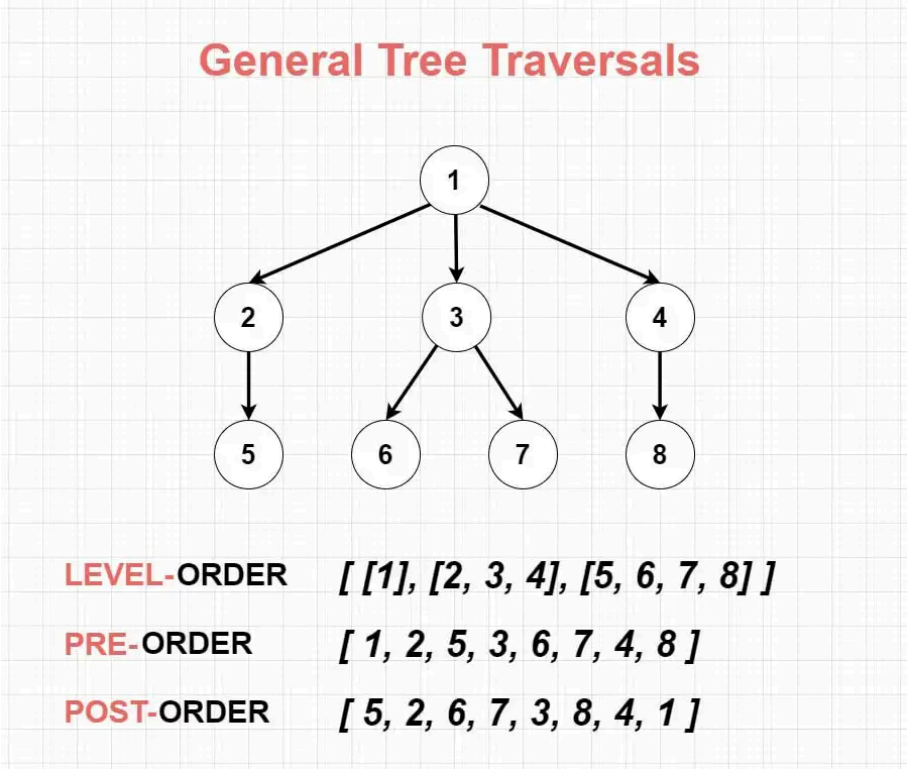
- **Level-order traversal:** O(n), where n is the number of nodes in the tree.
- **Pre-order traversal:** O(n), where n is the number of nodes in the tree.
- **Post-order traversal:** O(n), where n is the number of nodes in the tree.

Space complexity:

- **Level-order traversal:** O(n), where n is the number of nodes in the tree. This is because the traversal requires a queue to store the nodes at each level, and in the worst case scenario, all nodes in the tree are added to the queue.
- **Pre-order and post-order traversals:** O(h), where h is the height of the general tree. This is because the function call stack is used to store the function calls for each node in the tree in the case of recursive implementations, or a managed stack or queue is used to store nodes that still need to be visited in the case of iterative implementations. The maximum depth of the function call stack or managed stack/queue is equal to the height of the tree.

Therefore, for very deep trees or large data sets the iterative implementation might be a better choice to avoid stack overflow errors. This is because with iterative implementation we keep the nodes in a stack or queue instead of using the function call stack. This helps to prevent filling up the function call stack while having more granular control over the memory. Because the size of the stack or queue can be managed / adjusted to the size of tree being traversed, rather than being constrained by the maximum size of the function call stack.

Both the iterative and recursive implementations are included for each traversal method below. For simplicity I have included manual insertion code to create a starter tree & as well as printTreeAsString method to view the tree as string in the console. Just enough code to test all traversals in your favorite code editor. I have also added code comments to make it as easy as possible to follow them along.




```
class Node {
  constructor(key, data) {
    this.key = key
    this.data = data
    this.children = []




  }
}

class GeneralTree {
  constructor() {
    this.root = null
  }

  // helper to print the tree as string
  printTreeAsString() {
    if (!this.root) throw new Error('Tree is empty')

    const getTreeString = (node = this.root, spaceCount = 0) => {
      let treeString = "\n";

      node.children.forEach((child) => {
        treeString += `${" ".repeat(spaceCount)}• node | key: ${child.key} - ${child.data} ${getTreeStrir
      })

      return treeString
    }

    return console.log(`\n • node | key: ${this.root.key} - ${this.root.data} ${getTreeString(this.root,
  }

  // Iterative level-order traversal
  levelOrderTraversalIterative() {
    // Check if the tree is empty
    if (!this.root) throw new Error('Tree is empty')
```

```
  // Initialize a queue to keep track of nodes to be visited, an array to store results, and a count fo
  const queue = [{ node: this.root, level: 0 }]
  const result = [[]]
  let currentLevel = 0

  // Iterate over the queue while it's not empty
  while (queue.length) {
    // Remove the first node from the queue and add its data to the result array
    const { node, level } = queue.shift()
    if (level > currentLevel) {
      currentLevel = level
      result.push([])
    }
    result[currentLevel].push(node.data)

    // Add all the children of the current node to the end of the queue, along with their level informa
    for (const childNode of node.children) {
      queue.push({ node: childNode, level: level + 1 })
    }
  }

  // Return the result array
  return result
}

// Recursive level-order traversal
levelOrderTraversalRecursive() {
  // Check if the tree is empty




  if (!this.root) throw new Error('Tree is empty')

  // Initialize an array to store results and an object to keep track of nodes by level
  const result = []
  const nodesByLevel = {}

  // Define a recursive function to traverse the tree
  function traverse(currentNode, currentLevel) {
    // Base case: if the node is null, return
    if (!currentNode) return

    // Add the current node's data to the result array for the current level
    if (!nodesByLevel[currentLevel]) {
      nodesByLevel[currentLevel] = []
      result.push(nodesByLevel[currentLevel])
    }
    nodesByLevel[currentLevel].push(currentNode.data)

    // Recursively traverse the children of the current node
    for (const childNode of currentNode.children) {
      traverse(childNode, currentLevel + 1)
    }
  }

  // Start the traversal with the root node at level 0
  traverse(this.root, 0)

  // Return the result array
  return result
```

```
}

// Iterative pre-order traversal
preOrderTraversalIterative() {
  // Check if the tree is empty
  if (!this.root) throw new Error('Tree is empty')

  // Create a stack to hold nodes to be processed and
  // an array to hold the result of the traversal
  const stack = [this.root]
  const result = []

  // Loop through the stack until all nodes have been processed
  while (stack.length) {
    // Pop the last node from the stack and add its data to the result array
    const currentNode = stack.pop()
    result.push(currentNode.data)

    // Since we want to traverse in pre-order,
    // we want to visit the leftmost child first.
    // Since the children array is stored in reverse order
    // in this implementation, we need to push them to stack in reverse.
    for (let i = currentNode.children.length - 1; i >= 0; i--) {
      stack.push(currentNode.children[i])
    }
  }

  // Return the result array
  return result




}

// Recursive pre-order traversal
preOrderTraversalRecursive() {
  // Check if the tree is empty
  if (!this.root) throw new Error('Tree is empty')

  // Create an array to hold the result of the traversal
  const result = []

  // Define a recursive helper function to traverse the tree
  function traverse(currentNode) {
    // If the current node is null, return
    if (!currentNode) return

    // Add the node's data to the result array
    result.push(currentNode.data)

    // Recursively traverse each of the node's children
    for (const childNode of currentNode.children) {
      traverse(childNode)
    }
  }

  // Call the helper function with the root node to start the traversal
  traverse(this.root)

  // Return the result array
  return result
```

```
  }

  // Iterative post-order traversal
  postOrderTraversalIterative() {
    // Check if the tree is empty
    if (!this.root) throw new Error('Tree is empty')

    // Initialize a stack and a result array
    const stack = [this.root]
    const result = []

    // Loop through the stack
    while (stack.length) {
      // Get the top of the stack
      const currentNode = stack.pop()

      // Add the current node's data to the result
      result.push(currentNode.data)

      // Push the current node's children to the stack
      for (let i = 0; i < currentNode.children.length; i++) {
        stack.push(currentNode.children[i])
      }
    }

    // Return the result array, reversed
    return result.reverse()
  }




  // Recursive post-order traversal
  postOrderTraversalRecursive() {
    // Check if the tree is empty
    if (!this.root) throw new Error('Tree is empty')

    // Initialize a result array
    const result = []

    // Recursive helper function to traverse the tree in post-order
    function traverse(currentNode) {
      if (!currentNode) return

      // Traverse the children of the current node
      for (const childNode of currentNode.children) {
        traverse(childNode)
      }

      // Add the current node's data to the result
      result.push(currentNode.data)
    }

    // Start the traversal from the root
    traverse(this.root)

    // Return the result array
    return result
  }
}
```

```
const ceo = new Node(1, '#1 CEO')

// Build departments
const operationsManager = new Node(2, '#2 Operations Manager')
const marketingManager = new Node(3, '#3 Marketing Manager')
const financeManager = new Node(4, '#4 Finance Manager')

// add staff to departments
operationsManager.children.push(new Node(5, '#5 Operation Staff'))
marketingManager.children.push(new Node(6, '#6 Marketing Staff'))
marketingManager.children.push(new Node(7, '#7 Content Marketer'))
financeManager.children.push(new Node(8, '#8 Assistant Accountant'))

// link the departments with ceo
ceo.children.push(operationsManager, marketingManager, financeManager)

// create the tree
const tree = new GeneralTree()

// attach the ceo to tree root
tree.root = ceo

// See the tree as string
tree.printTreeAsString()

console.log('LEVEL-ORDER TRAVERSAL:')
tree.levelOrderTraversalIterative()




console.log('PRE-ORDER TRAVERSAL:')
tree.preOrderTraversalIterative()

console.log('POST-ORDER TRAVERSAL:')
tree.postOrderTraversalIterative()
```

Binary Tree Traversal Implementations in Javascript

In this section we will cover the iterative and recursive implementations of the four standard traversal techniques for Binary trees: level-order, pre-order, in-order, and post-order. Binary trees have a fixed structure with at most two children for each node, which allows for a well-defined set of traversal techniques.

I'd like to mention these traversals will also apply to Binary tree variants - for example Binary Search Tree, Red-black Tree and AVL Tree. Because they do follow the same structure: consisting of nodes with at most 2 child nodes.

Big O for the iterative and recursive traversal methods for Binary trees are:

Time complexity

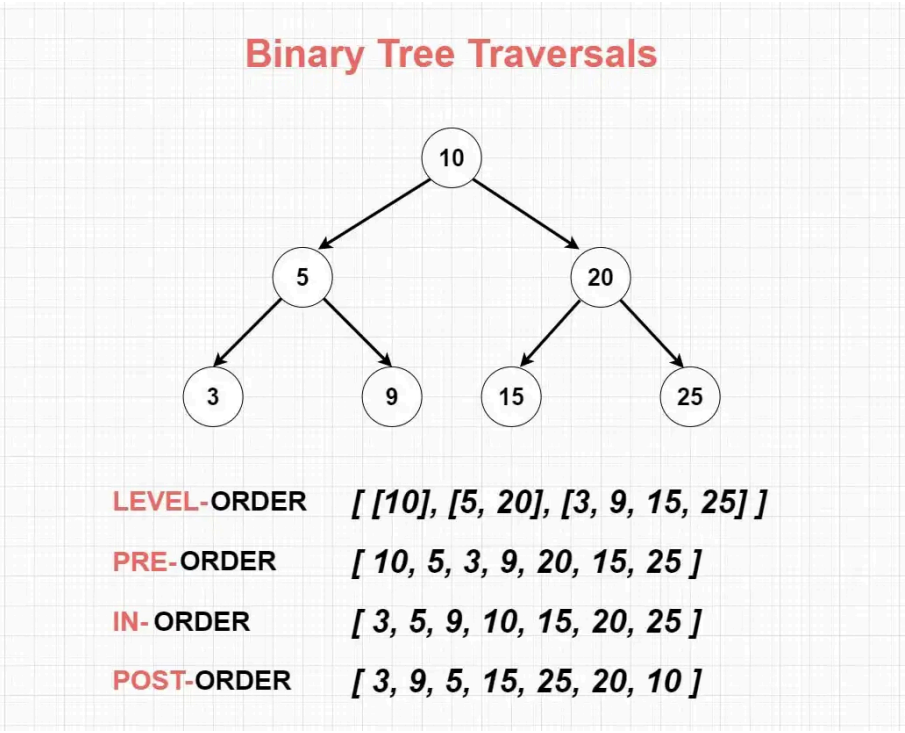
- **Level-order traversal:** O(n), where n is the number of nodes in the tree.
- **Pre-order traversal:** O(n), where n is the number of nodes in the tree.
- **In-order traversal:** O(n), where n is the number of nodes in the tree.
- **Post-order traversal:** O(n), where n is the number of nodes in the tree.

Space complexity:

- **Level-order traversal:** O(w), where w is the maximum width of the binary tree, i.e., the maximum number of nodes in any level of the tree - because the traversal requires a queue to store the nodes at each level. Even though O(w) doesn't fall in the main Big O complexities list, it still represents a linear complexity - but we cannot just call it as O(n). Because O(w) is specific to the maximum width of a binary tree, while O(n) is directly relative to the input size.
- You may have also noticed space complexity is different compared to General Tree. The reason is Binary tree has a well-defined structure, where each node has at most two child nodes. Due to it's structure, maximum width of a Binary tree is typically much smaller than the number of nodes in the tree.
- **Pre-order, in-order, and post-order traversals:** O(h), where h is the height of the Binary tree. Similar to General tree, reason is the function call stack is used to store the function calls for each node in the tree in the case of recursive implementations, or a managed stack or queue is used to store nodes that still need to be visited in the case of iterative implementations. The maximum depth of the function call stack or managed stack/queue is equal to the height of the tree.

Both the iterative and recursive implementations are included for each traversal method below. For simplicity I have included manual insertion code to create a starter tree. Just enough code to test all traversals in your favorite code editor. I have also added code comments to make it as easy as possible to follow them along.

With this section, we came to the end of this article. I hope it helped you to understand what Tree traversals are and how do they work for General and Binary trees! If this is your first time on Tree traversals, don't expect to understand everything at once - this is a fairly complex topic. Try them out and give it some time until it makes sense. Thanks for reading!




```
class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
```

```
}

class BinaryTree {
  constructor() {
    this.root = null
  }

  // Iterative level-order traversal
  levelOrderTraversalIterative() {
    // Initialize an empty array to hold the results of the traversal
    const result = []

    // If the tree is not empty, initialize a queue with the root node
    if (this.root) {
      const queue = [this.root]

      // Loop through each node in the queue
      while (queue.length > 0) {
        // Initialize an empty array to hold the values of the current level
        const currentLevel = []

        // Get the number of nodes in the current level
        const levelSize = queue.length

        // Loop through each node in the current level
        for (let i = 0; i < levelSize; i++) {
          // Dequeue the first node from the queue and add its value to the current level array
          const currentNode = queue.shift()
          currentLevel.push(currentNode.value)




          // If the dequeued node has left or right children, enqueue them
          if (currentNode.left) {
            queue.push(currentNode.left)
          }
          if (currentNode.right) {
            queue.push(currentNode.right)
          }
        }

        // Add the current level array to the result array
        result.push(currentLevel)
      }
    }

    // Return the result array
    return result
  }

  // Recursive level-order traversal
  levelOrderTraversalRecursive() {
    const temp = []

    // Define a recursive helper function to traverse the tree
    function traverse(node, level) {
      // If this level has not yet been defined in the temp array, create it
      if (!temp[level]) {
        temp[level] = []
      }
```

```
    // Push the current node's value into the temp array at the current level
    temp[level].push(node.value)

    // If the current node has a left child, traverse it at the next level
    if (node.left) {
      traverse(node.left, level + 1)
    }

    // If the current node has a right child, traverse it at the next level
    if (node.right) {
      traverse(node.right, level + 1)
    }
  }

  // Call the recursive helper function with the root node at the first level
  traverse(this.root, 0)

  return temp
}

// Iterative pre-order traversal
preOrderTraversalIterative() {
  const stack = []
  const result = []

  // Push root node to stack
  stack.push(this.root)




  while (stack.length) {
    // Pop top node from stack and add its value to result array
    const node = stack.pop()
    result.push(node.value)

    // Push right child to stack first, so that it is processed after the left child
    if (node.right) {
      stack.push(node.right)
    }
    if (node.left) {
      stack.push(node.left)
    }
  }

  return result
}

// Recursive pre-order traversal
/*
 NOTE:
 Result array here is passed as an argument to each instance of the function
 This means that every instance of the function
 has a reference to the same array object in memory.
 When an instance of the function appends a value to the result array,
 that change is visible to every other instance of the function.

 Same approach is also used for in-order and post-order
 recursive methods.
*/
```

```
preOrderTraversalRecursive(node = this.root, result = []) {
  if (node) {
    // Add node value to result array
    result.push(node.value)

    // Traverse left subtree recursively
    this.preOrderTraversalRecursive(node.left, result)

    // Traverse right subtree recursively
    this.preOrderTraversalRecursive(node.right, result)
  }

  return result
}

// Iterative in-order traversal
inOrderTraversalIterative() {
  // create an empty array to store the traversal result
  const result = []
  // create an empty stack to keep track of nodes
  const stack = []

  // start at the root node of the tree
  let curr = this.root

  // while there are still nodes to visit
  while (curr || stack.length > 0) {
    // traverse to the leftmost leaf node of the subtree
    while (curr) {




      // push the current node onto the stack
      stack.push(curr)
      // move to the left child node
      curr = curr.left
    }

    // process the node at the top of the stack
    curr = stack.pop()  // pop the top node off the stack
    result.push(curr.value)  // add the node's value to the result array

    // traverse the right subtree of the processed node
    curr = curr.right  // move to the right child node
  }

  return result
}

// Recursive in-order traversal
inOrderTraversalRecursive(node = this.root, result = []) {
  // if the node is null, return the current result array
  if (!node) {
    return result
  }

  // traverse the left subtree
  this.inOrderTraversalRecursive(node.left, result)

  // process the current node
  result.push(node.value)
```

```
  // traverse the right subtree
  this.inOrderTraversalRecursive(node.right, result)

  return result
}

// Iterative post-order traversal
postOrderTraversalIterative() {
  // initialize an empty array to store the result
  const result = []

  // initialize an empty stack to keep track of nodes to visit
  const stack = []

  // start traversal at the root node
  let curr = this.root

  // initialize a variable to keep track of the last visited node
  let lastVisitedNode = null

  // loop through all nodes in the tree
  while (curr || stack.length > 0) {
    // traverse to the leftmost leaf node of the subtree
    while (curr) {
      // add the current node to the stack
      stack.push(curr)

      // move to the left child




      curr = curr.left
    }

    // check if right child is present and not already visited
    const topNode = stack[stack.length - 1]
    if (topNode.right && topNode.right !== lastVisitedNode) {
      // traverse right subtree
      curr = topNode.right
    } else {
      // process the node at the top of the stack
      result.push(topNode.value)

      // mark the node as visited
      lastVisitedNode = stack.pop()
    }
  }

  return result
}

// Recursive post-order traversal
postOrderTraversalRecursive(node = this.root, result = []) {
  if (!node) {
    return result
  }

  // traverse left subtree
  this.postOrderTraversalRecursive(node.left, result)
```

```
    // traverse right subtree
    this.postOrderTraversalRecursive(node.right, result)

    // process current node
    result.push(node.value)

    return result
  }
}

// initialize the tree:
const tree = new BinaryTree()

// create the root node:
const rootNode = new Node(10)

// add some values
rootNode.left = new Node(5)
rootNode.left.left = new Node(3)
rootNode.left.right = new Node(9)

rootNode.right = new Node(20)
rootNode.right.left = new Node(15)
rootNode.right.right = new Node(25)

// attach it to the tree
tree.root = rootNode
```