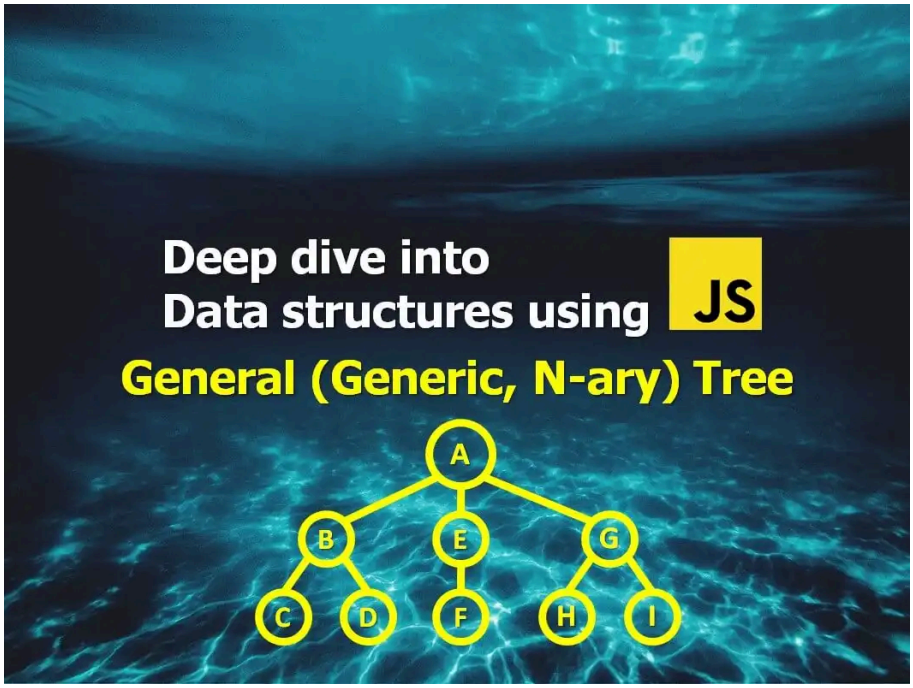
Deep Dive into Data structures using Javascript - General (Generic, N-ary) tree



What is a General (Generic, N-ary) Tree?

General Tree - which is also called ***Generic***, ***n-ary***, ***m-ary***, ***k-ary*** or ***k-way Tree*** is a non-linear, advanced, and powerful data structure that is widely used for storing hierarchically relational data. A node can have zero or any number of children, as well as there are no restrictions on how nodes are being inserted.

All Tree variations derive from General Tree. In other words, it is the common ancestor for all Tree types.

When the Tree is specialized, **"n"**, **"m"** or **"k"** letters in the name variations represent the upper bound / maximum number of child nodes on any level. For example, the most popular descendant **Binary Trees** has the ***"n = 2"*** as a restriction, and another descendant **Ternary Trees** is restricted with ***"n = 3"***.

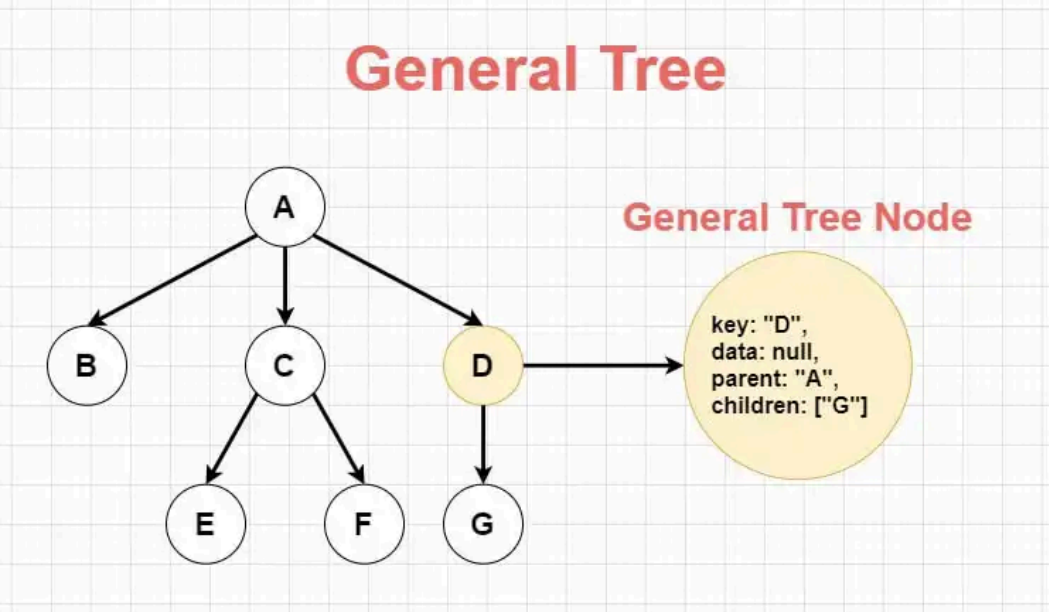
If the Tree is not specialized, it simply means there are no restrictions on the number of nodes at any level. Thus that's why we call it a General or Generic Tree in this case.

General Tree is not a built-in data structure in Javascript, but we can implement a custom one.

The tone of this article is assuming you are at least familiar with the Tree data structure concept. If that's not the case or you need a quick refreshment, I'd suggest you start from the "Introduction to Trees" article by following the link below, then come back and continue here later:

Deep dive into data structures using Javascript - Introduction to Trees

Anatomy of a General Tree



General Tree consists of connected nodes that resemble a hierarchy. If we take a closer look, we can see how non-linear character comes into the picture. When we think about a linear data structure like Linked List, it is easy to see the connected nodes forming a shape that looks like a line due to following a linear direction. But when we look at the Tree, we see that any node can connect to multiple nodes below - which also translates to having multiple directions.

General Tree always starts with a root node and every node in the Tree descends from the root node. There can be only one root in the Tree.

Each node in the General tree consists of 4 properties:

key: Unique identifier for the node.

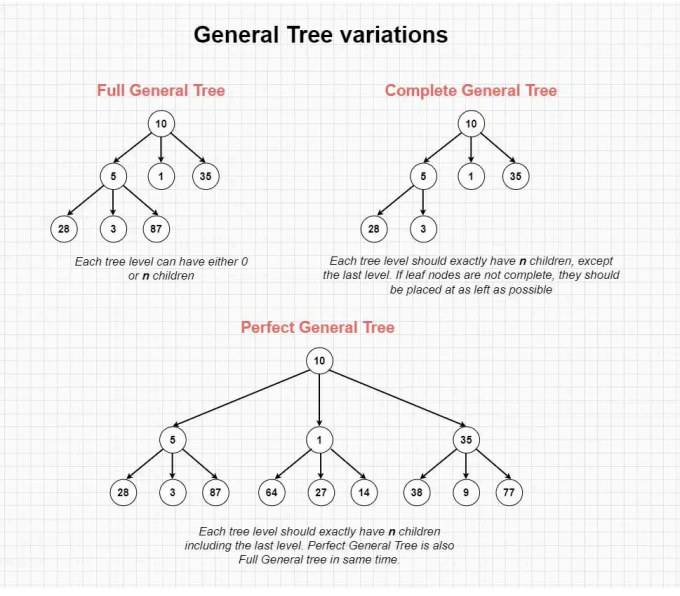
data: Holds the value / data for the node.

parent: Holds a reference (pointer) to the parent Node. If it is the root node, this value is “null”

children: An array that can hold any number of children nodes.

Variations of General tree

Variation in this context refers to how the nodes are placed inside the Tree. Even though the variation topic is often more popular with restricted / specialized Trees, it still applies to General Tree.



Full General Tree: Where each tree level can have either 0 or “n” children.

Complete General Tree: Where each tree level should exactly have “n” children, except the last level. If leaf nodes are not complete, they should be placed at as left as possible.

Perfect General Tree: Where each tree level should exactly have “n” children including the last level. Perfect General Tree is also Full General tree in same time.

When to use General Tree

Let’s start with taking a look at the Big O of common operations in General Tree:

Method	Worst case
add()	O(n)
remove()	O(n)
find()	O(n)
Breadth First / Depth First traversals	O(n)

The primary use of the General tree is managing hierarchical data. Even though the time complexity in operations has average speed, it still makes a lot of sense since it gives us out-of-the-box capabilities to efficiently store hierarchical type of data. Some examples are:

- Computer file systems
- Organization charts
- Any data that needs to be managed in hierarchical manner.

General Tree implementation in Javascript

We will be using ES6 Classes to build our General Tree. Implementing non-linear data structures are fairly complex, so we will go through a couple first steps in detail until we understand the key concepts.

Build a class for the Tree Node

Main building block is the Node element. We can use a class for it, so we can reuse it whenever we need to create a new Node:

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

```
// EXAMPLE:
const newNode = new Node(1, null)
console.log(newNode)

/* OUTPUT:
Node {
  key: 1,
  data: null,
  parent: null,
  children: []
}

*/
```

Build a class for the General Tree

Now we can move further and create a class for the General Tree. We know there should be a "root" property in the Tree as a reference to the root node, which we can place in the constructor:

```
class Tree {
  constructor() {
    this.root = null
  }
}

const tree = new Tree()
console.log(tree)




/* OUTPUT
Tree {
  root: null,
}
*/
```

Now that we have the base building blocks, we can extend the Tree class by introducing methods. Here is the list of methods we are going to implement:

- `add(key, value, parentKey)` - *add an element under a specific parent node*
- `find(targetKey)` - *finds an element that matches with given targetKey. uses depthFirstSearch under the hood.*
- `remove(targetKey)` - *removes the node and it's children that matches with given targetKey.*
- `depthFirstSearch(key, node(optional))` - *uses pre-order traversal with early exit*
- `printTreeAsString()` - *helper method to display tree as string in the console*

Understanding non-linear traversals

If we start building our Tree with the add method, as a first condition we need to check if the Tree is empty. In that case, root should point to the newNode:

```
add(key, value, parentKey) {
    // create a newNode:
    const newNode = new Node(key, value, parentKey)
```

```
    // if there is no root, root will be the newNode
    if (!this.root) {
       this.root = newNode
    }
    ....
  }
```

This will successfully handle adding a root node. But how about if we wanted to add another node under the root, or assuming we have a bigger tree and the target node is placed somewhere that is many levels deep? In both cases, we need to find the target parent using a helper find method. The same thing will also apply to removing a node.

Since we are dealing with a non-linear data structure, a simple loop won't do the trick here since it follows a single direction. Because the target parent node here can be in any direction. That's where the Tree traversal comes into the picture. Tree Traversal is a subject that needs its own article. To keep things simple I won't be diving deep into Tree traversal details here - but rather give a quick overview.

Basically, we will need a helper method to find any node in the Tree with the given key. This method is going to use depthFirstSearch as a helper method under the hood.

Depth first search or DFS is a traversal algorithm used in both Tree and Graph data structures, which uses backtracking as a concept. This is how it looks like in action:

Image Source: Wikimedia Commons

As the name suggests, we see a pattern scanning the Tree in depth-first fashion, then continues using the same pattern horizontally.

Now let's take a look at the depthFirstSearch method:

```
// using recursive Pre-order traversal with early exit (break) when it is found
  depthFirstSearch(targetKey, node = this.root) {
    // if the Tree is empty, throw an error
    if (!node) throw new Error('Tree is empty')

    let targetNode = false
    let found = false

    function recursiveDepthTraverse(currentNode) {
      // traverse over given node children elements
      for (let i = 0; i < currentNode.children.length; i++) {
        // if element matches targetKey, break here
        if (currentNode.children[i].key === targetKey) {
          found = true
          targetNode = currentNode.children[i]
          break
        } else {
          recursiveDepthTraverse(currentNode.children[i])
          // signal condition to stop the original loop if it is found
          if (found) {
            break
          }
        }
      }
    }

    recursiveDepthTraverse(node)
    if (!targetNode) {
      throw new Error(`on depthFirstSearch(): Target node with given key: "${targetKey}" is not found in the t
```

```
    } else {
      return targetNode
    }
  }
```

This might look confusing at first and can take a bit of time to understand, but what worked for me was playing around with it until it really made sense to me. I'd like to highlight 2 key points here:

1 - How we use recursion inside the for loop. As long as we don't reach the targetKey, we call the same function again by passing the child node on each iteration. For each index this goes on until we reach to it's deepest level. This behavior makes it possible to scan the whole tree using a depth-first pattern.

2 - How do we handle the early exit. If you have noticed, we have the recursive function as an inner function. Because as soon as we find the target, we want to stop the traversal. If there is no early exit logic, this function will keep running until every single node is visited in the tree - and that's not the point for this case.

If we kept the "found" state inside the recursive function, it would get lost in the next call - because with every recursive call we have a fresh start in the scope. Having the "found" variable at the outer scope makes it possible for the recursive function to both update and understand the stop signal by using the found variable.

As said earlier, non-linear data structures can be quite complex to deal with - but if you give enough time it will all make sense!

I hope this article helped you to understand what General Tree is and how it works! I'd like to encourage you to experiment with the implementation below in your favorite code editor. I've also included a helper method called "printTreeAsString" in the implementation, it will show you the tree in string format at your console, looking like this:

- node key: 1
  - node key: 3
    - node key: 5
    - node key: 6
  - node key: 2
  - node key: 4

Additionally, I've also included the sample data which will give you the output above where you'll find it below the Tree class. Thanks for reading!

Implementation of General Tree in Javascript:

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

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

  add(key, value, parentKey) {
```

```
    // create a newNode:
    const newNode = new Node(key, value, parentKey)

    // if there is no root, root will be the newNode
    if (!this.root) {
       this.root = newNode
    } else {
      try {
        // find the parent node for attachment:
        const targetParent = this.find(parentKey)
        // make the new node point to parent:
        newNode.parent = targetParent
        // add the newNode to targetParents children array:
        targetParent.children.push(newNode)
      } catch (err) {
        // error will occur if the node with parentKey is not found at this step
        console.error(`Error on add() method: Node with given parentKey: "${parentKey}" is not found in the Tree.
      }
    }
  }

  remove(targetKey) {
    // if the Tree is empty, throw an error
    if (!this.root) throw new Error('Tree is empty')

    // initialize a variable for targetNode that is going to be returned:
    let targetNode

    // if targetKey === root key, directly remove without doing a traversal:




    if (targetKey === this.root.key) {
      // update targetNode value with root
      targetNode = this.root
      // reset the root:
      this.root = null
      // exit
      return targetNode
    } else {
      try {
        // find the target node:
        targetNode = this.find(targetKey)

        // pick the parent:
        const parent = targetNode.parent

        // initialize a new array for new children array
        const updatedChildren = []

        // to remove the node with targetKey, exclude from it's parents children array:
        for (let i = 0; i < parent.children.length; i++) {
          if (targetKey !== parent.children[i].key) {
            updatedChildren.push(parent.children[i])
          }
        }

        // update the parent:
        parent.children = updatedChildren
        // exit:
        return targetNode
```

```
      } catch (err) {
        console.error(`Error on remove() method: Node with given targetKey: "${targetKey}" is not found in th
      }
    }
}

find(targetKey) {
  // if the Tree is empty, throw an error
  if (!this.root) throw new Error('Tree is empty')

  // if targetKey is root, return the root
  if (targetKey === this.root.key) {
    return this.root
  // else, do a depth first search:
  } else {
    return this.depthFirstSearch(targetKey, this.root)
  }
}

// using recursive Pre-order traversal with early exit (break) when it is found
depthFirstSearch(targetKey, node = this.root) {
  // if the Tree is empty, throw an error
  if (!node) throw new Error('Tree is empty')

  let targetNode = false
  let found = false

  function recursiveDepthTraverse(currentNode) {
    // traverse over given node children elements




    for (let i = 0; i < currentNode.children.length; i++) {
      // if element matches targetKey, break here
      if (currentNode.children[i].key === targetKey) {
        found = true
        targetNode = currentNode.children[i]
        break
      } else {
        recursiveDepthTraverse(currentNode.children[i])
        // signal condition to stop the original loop if it is found (goes to next step)
        if (found) {
          break
        }
      }
    }
  }

  recursiveDepthTraverse(node)
  if (!targetNode) {
    throw new Error(`on depthFirstSearch(): Target node with given key: "${targetKey}" is not found in the tree
  } else {
    return targetNode
  }
}

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} ${getTreeString(child, spaceCount + 4)}
      })

      return treeString
    }

    return console.log(`\n • node key: ${this.root.key} ${getTreeString(this.root, 4)}`)
  }
}

// SAMPLE DATA & PLAYGROUND SETUP:

const nums = [
  {
        id: 1,
        parentId: null,
        data: {
            name: 1
        }
    },
    {
        id: 3,
        parentId: 1,
        data: {
            name: 3
        }
    },




    {
        id: 2,
        parentId: 1,
        data: {
            name: 2
        }
    },
        {
        id: 4,
        parentId: 1,
        data: {
            name: 4
        }
    },
        {
        id: 5,
        parentId: 3,
        data: {
            name: 5
        }
    },
    {
        id: 6,
        parentId: 3,
        data: {
            name: 6
        }
    },
]
```

```
// Initialize
const tree = new Tree()

// Add the elements
nums.forEach(num => tree.add(num.id, num.data, num.parentId))

// Print Tree as string
tree.printTreeAsString()
```