```
const Node = (node, nextNode = null, prevNode = null) => {
    let data = node;
    let next = nextNode;
    let prev = prevNode;
    return { data, next, prev }
}
```

```
const DLL = () => {
    let head = null
    let tail = null
    let size = 0


    const insertFirst = (data) => {
        size++
        let node = Node(data, null, null)
        if (tail) {
            tail.next = node
            node.prev = tail
            tail = node
            return node
        }

        head = node
        tail = node
        return node
    }
    return { insertFirst }
}
```

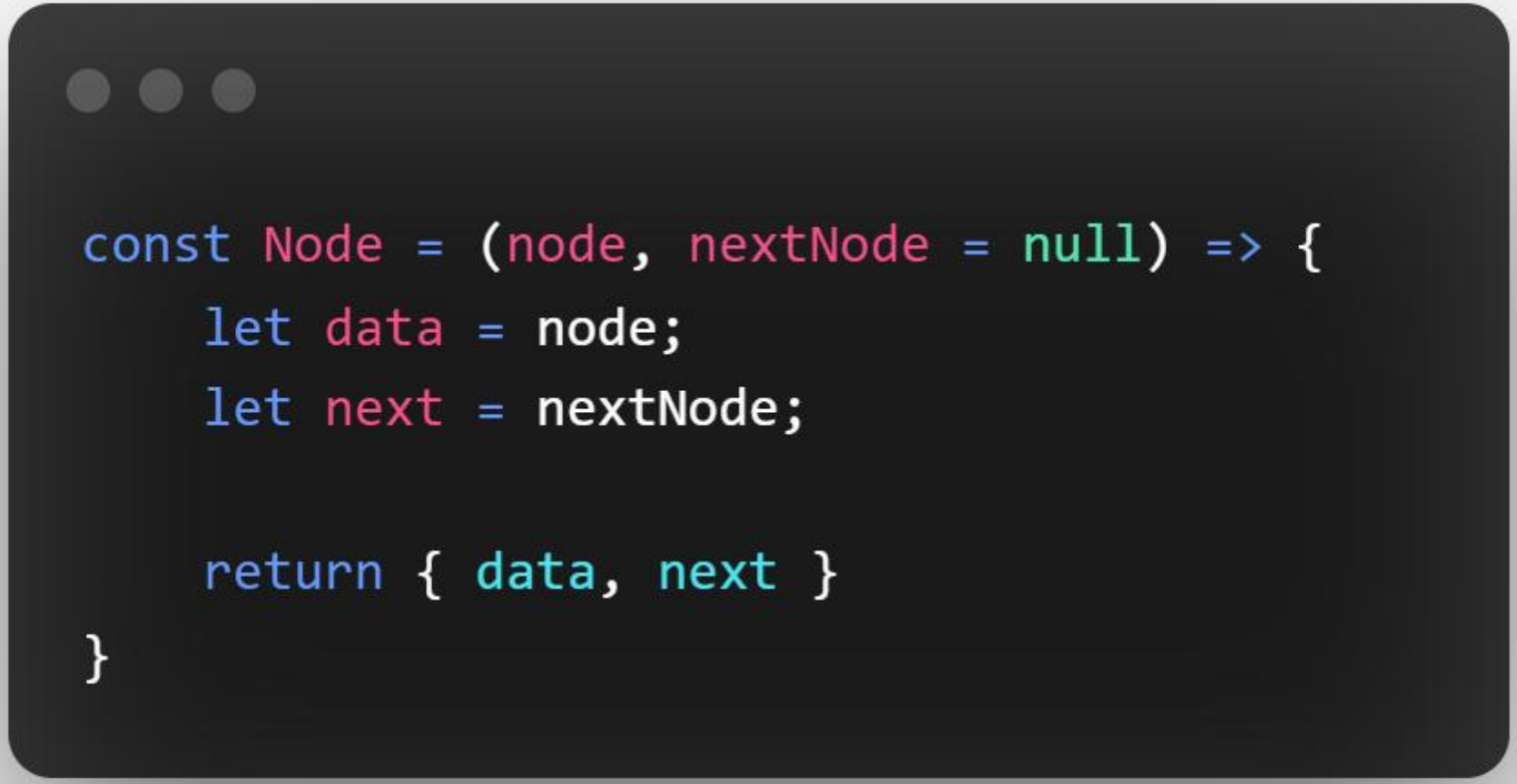
```
const DLL = () => {
    let head = null
    let tail = null
    let size = 0


    const insertBefore = (data, beforeData) => {
        if (!data || !beforeData) {
            return
        }

        let node = Node(data, null, null)
        let current, previous
        var count = 0
        current = head
        while (count < size) {
            count++

            if (current.data == beforeData) {
                node.next = current
                node.prev = current.prev
                current.prev.next = node
                current.prev = node
                size++
                break
            }
            current = current.next
        }
    }
    return { insertBefore }
}
```

```
const DLL = () => {
    let head = null
    let tail = null
    let size = 0


     const getAt = (index) => {
        let current = head
        let count = 0
        while (current) {
            if (count == index) {
                console.log(`node at ${index} ~> prev:${current.prev?.data} ~>
current:${current.data} ~> next:${current.next?.data}`)
            }
            count++
            current = current.next
        }
    }
    return { getAt }
}
```

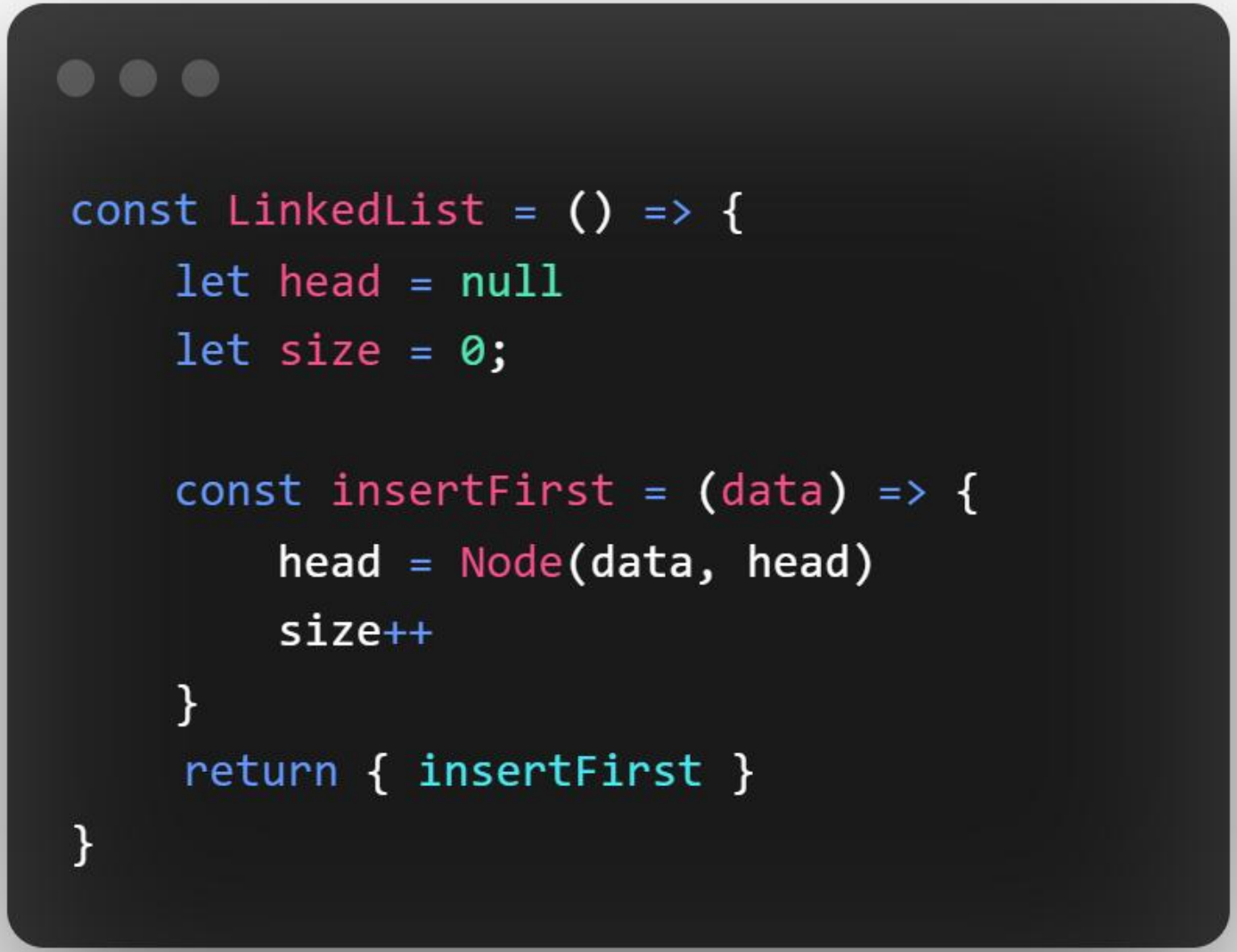
```
const DLL = () => {
    let head = null
    let tail = null
    let size = 0


     const deleteAt = (index) => {
        if (index > 0 && index > size) {
            return
        }

        if (index == 0) {
            head = null
            tail = null
        }

        let current = head
        let previous
        let count = 0

        while (count < index) {
            count++

            previous = current
            current = current.next
            current.prev = previous
        }
        previous.next = current.next
        current.next.prev = current.prev
        size--
    }
    return { deleteAt }
}
```

```
const DLL = () => {
    let head = null
    let tail = null
    let size = 0


     const printDLL = () => {
        let current = head;
        while (current) {
            console.log(
                `${current.prev?.data || '@'} ~> ${current.data} ~> ${current.next?.data
|| '@'}`
            );
            current = current.next;
        }
    }
    return { printDLL }
}
```

```
const { insertFirst, printDLL, insertBefore, deleteAt, getAt } = DLL()


insertFirst(3)
insertFirst(7)
insertFirst(10)
insertFirst(13)
insertFirst(23)
insertFirst(34)
printDLL()
console.log('-----------------insertBefore-------------------')
insertBefore(10000, 13)
printDLL()
deleteAt(2)
console.log('-------------------deleteAt-----------------')
printDLL()
getAt(3)

//# OUTPUT
// @ ~> 3 ~> 7
// 3 ~> 7 ~> 10
// 7 ~> 10 ~> 13
// 10 ~> 13 ~> 23
// 13 ~> 23 ~> 34
// 23 ~> 34 ~> @
// -----------------insertBefore-------------------
// @ ~> 3 ~> 7
// 3 ~> 7 ~> 10
// 7 ~> 10 ~> 10000
// 10 ~> 10000 ~> 13
// 10000 ~> 13 ~> 23
// 13 ~> 23 ~> 34
// 23 ~> 34 ~> @
// -------------------deleteAt-----------------
// @ ~> 3 ~> 7
// 3 ~> 7 ~> 10000
// 7 ~> 10000 ~> 13
// 10000 ~> 13 ~> 23
// 13 ~> 23 ~> 34
// 23 ~> 34 ~> @
// node at 3 ~> prev:10000 ~> current:13 ~> next:23
```



```
const Node = (node, nextNode = null) => {
    let data = node;
    let next = nextNode;

    return { data, next }
}
```



```
const LinkedList = () => {
    let head = null
    let size = 0;

    const insertFirst = (data) => {
        head = Node(data, head)
        size++
    }
    return { insertFirst }
}
```

```
const LinkedList = () => {
    let head = null
    let size = 0;

     const getAt = (index) => {

        let current = head
        let count = 0
        while (current) {
            if (count == index) {
                console.log(`node at ${index} is ~>[`, current.data, ']')
            }
            count++
            current = current.next
        }
    }
     return { getAt }
}
```

```
const LinkedList = () => {
    let head = null
    let size = 0;

    //# [] => [] => [beforeData] => [] => []
    //#          <=
    const insertBefore = (data, beforeData) => {

        if (!data || !beforeData) {
            return
        }

        let node = Node(data)
        let current, previous
        let count = 0
        current = head

        while (count < size) {
            previous = current
            count++
            current = current.next
            if (current.data == beforeData) {
                previous.next = node
                node.next = current
                size++
                break
            }
        }
    }
    return { insertBefore }
}
```

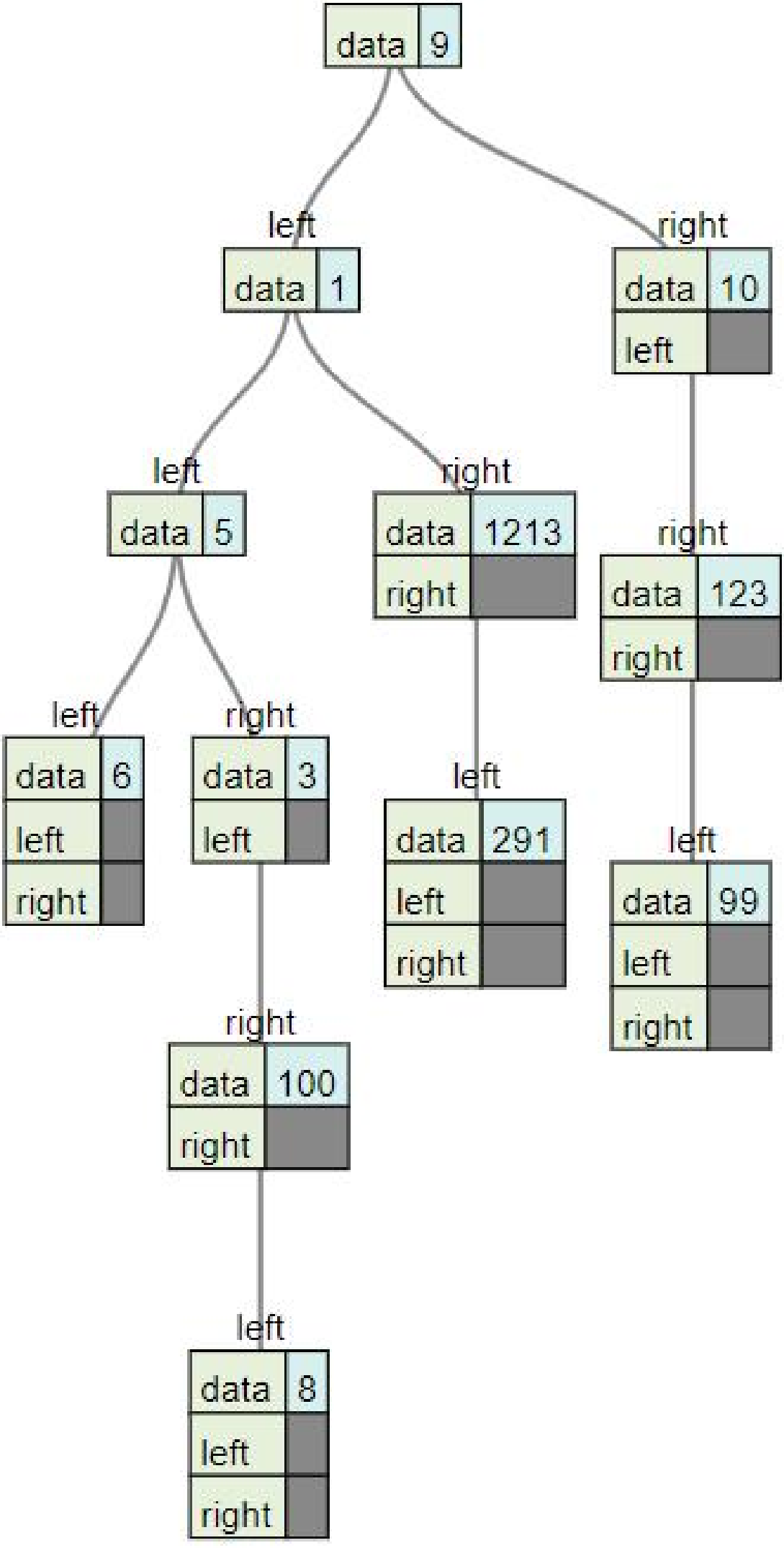
```
const LinkedList = () => {
    let head = null
    let size = 0;

    const insertAt = (data, index) => {

        if (index > 0 && index > size) {
            return
        }

        if (index === 0) {
            head = Node(data, head)
            return
        }
        let node = Node(data)
        let current, previous
        let count = 0
        current = head

        while (count < index) {
            previous = current
            count++
            current = current.next
        }

        node.next = current
        previous.next = node
        size++
    }
    return { insertAt }
}
```

```
const LinkedList = () => {
    let head = null
    let size = 0;

    const insertLast = (data) => {
        let node = Node(data);
        let current

        if (!head) {
            head = node
        } else {
            current = head
            while (current.next) {
                current = current.next
            }
            current.next = node
            size++
        }
    }
    return { insertLast }
}
```

```
const LinkedList = () => {
    let head = null
    let size = 0;

    const deleteAt = (index) => {
        if (index > 0 && index > size) {
            return
        }

        if (index === 0) {
            head = null
            return
        }

        let current = head
        let previous
        let count = 0

        while (count < index) {
            count++
            previous = current
            current = current.next
        }
        previous.next = current.next
        size--
    }
    return { deleteAt }
}
```

```
const LinkedList = () => {
    let head = null
    let size = 0;

    const clearList = () => {
        head = null
        size = 0
    }

    const printListData = () => {
        let current = head
        const listArray = []
        while (current) {
            listArray.push(current.data)
            current = current.next
        }
        console.log('Log: ~> file: LinkedListFunctional.js ~> line 134 ~> printListData ~>
listArray[', listArray.join(' ] ~> [ '), ']')

        console.log('size ~>', size)
    }
    return { clearList, printListData }
}
```

```
const { insertFirst, insertLast, insertAt, deleteAt, getAt, clearList, printListData,
insertBefore } = LinkedList()

insertFirst(40)
insertLast(1)
insertLast(23)
insertLast(98)
insertLast(100)
insertAt(99, 2)
deleteAt(3)
getAt(2)

printListData()

insertBefore('777', 98)

printListData()

//# OUTPUT
// node at 2 is ~>[ 99 ]
// Log: ~> file: LinkedListFunctional.js ~> line 134 ~> printListData ~> listArray[ 40 ]
~> [ 1 ] ~> [ 99 ] ~> [ 98 ] ~> [ 100 ]
// size ~> 5
// Log: ~> file: LinkedListFunctional.js ~> line 134 ~> printListData ~> listArray[ 40 ]
~> [ 1 ] ~> [ 99 ] ~> [ 777 ] ~> [ 98 ] ~> [ 100 ]
// size ~> 6
```


data 9
left
data 1
right
data 10
left
left
data 5
right
data 1213
right
right
data 123
right
left
data 6
left
right
right
data 3
left
left
data 291
left
right
left
data 99
left
right
right
data 100
right
left
data 8
left
right

```
const Node = (node, l = null, r = null) => {
    const data = node
    const left = l;
    const right = r

    return { data, left, right }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Generate tree
    const buildTree = (rawArray) => {
        index++
        if (rawArray[index] === -1 || !rawArray[index]) {
            return null
        }
        const newNode = Node(rawArray[index])
        newNode.left = buildTree(rawArray)
        newNode.right = buildTree(rawArray)
        tree = newNode
        return newNode
    }
    return { buildTree }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Pre Order preOrderTraversal AKA DFS
    const preOrderArray = []
    const preOrderTraversal = (root) => {
        if (!root) {
            return null
        }
        preOrderArray.push(root.data)
        preOrderTraversal(root.left)
        preOrderTraversal(root.right)
    }
    return { preOrderTraversal, preOrderArray }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // In Order preOrderTraversal AKA DFS
    const inOrderArray = []
    const inOrderTraversal = (root) => {
        if (!root) {
            return null
        }
        inOrderTraversal(root.left)
        inOrderArray.push(root.data)
        inOrderTraversal(root.right)
    }
    return { inOrderTraversal, inOrderArray }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Post Order preOrderTraversal AKA DFS
    const postOrderArray = []
    const postOrderTraversal = (root) => {
        if (!root) {
            return null
        }
        postOrderTraversal(root.left)
        postOrderTraversal(root.right)
        postOrderArray.push(root.data)
    }
    return { postOrderTraversal, postOrderArray }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Level Order AKA BFS
    const levelOrder = (root) => {
        let queue = []
        if (!root) {
            return null
        }
        queue.push(root)
        queue.push(null)
        while (queue.length) {
            const current = queue.shift()
            if (!current) {
                console.log('\n')
                if (!queue.length) {
                    break
                } else {
                    queue.push(null)
                }
            } else {
                console.log(current.data)
                if (current.left) {
                    queue.push(current.left)
                }
                if (current.right) {
                    queue.push(current.right)
                }
            }
        }
    }
    return { levelOrder }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Count Nodes
    const countNodes = (root) => {
        if (!root) {
            return 0
        }

        const leftNodes = countNodes(root.left)
        const rightNods = countNodes(root.right)

        return leftNodes + rightNods + 1
    }
    return { countNodes }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Check for identical tree
    const isIdentical = (root, subRoot) => {

        if (!subRoot && !root) {
            return true
        }

        if (!root || !subRoot) {
            return false
        }
        if (root.data = subRoot.data) {
            return isIdentical(root.left, subRoot.left) && isIdentical(root.right,
subRoot.right)
        }

        return false
    }
    return { isIdentical }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Check for identical tree
    const isIdentical = (root, subRoot) => {

        if (!subRoot && !root) {
            return true
        }

        if (!root || !subRoot) {
            return false
        }
        if (root.data = subRoot.data) {
            return isIdentical(root.left, subRoot.left) && isIdentical(root.right,
subRoot.right)
        }

        return false
    }
    // Check for subTree's existence
    const isSubTree = (root, subRoot) => {
        if (!subRoot) {
            return true
        }

        if (!root) {
            return false
        }

        if (root.data === subRoot.data) {
            if (isIdentical(root, subRoot)) {
                return true
            }
        }

        return isSubTree(root.left, subRoot) || isSubTree(root.right, subRoot)

    }
     return { isIdentical, isSubTree }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Get all leaf nodes
    const leafs = []
    const leafNode = (root) => {
        if (!root) {
            return null
        }
        if (!root.left && !root.right) {
            leafs.push(root.data)
        }
        leafNode(root.left)
        leafNode(root.right)
    }
    return { leafNode, leafs }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

     // Print tree
    const printTree = (tree) => {
        console.log('Log: ~> file: BinaryTree.js ~> line 30 ~> printTree ~> tree',
JSON.stringify(tree, undefined, 4))
    }
    return { printTree }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Replace the old node with new node
    const replaceNode = (root, oldData, newData) => {

        if (!oldData || !root) {
            return null
        }

        if (root.data === oldData) {
            root.data = newData
            return
        }

        replaceNode(root.left, oldData, newData)
        replaceNode(root.right, oldData, newData)
    }
    return { replaceNode }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    const eitherSideNodes = []
    let maxLevelR = 0
    // Side visible node
    const eitherSideVisible = (root, level, side) => {

        const [first, second] = side === 'l' ? ['left', 'right'] : ['right', 'left']


        if (!root) {
            return null
        }

        if (maxLevelR < level) {
            eitherSideNodes.push(root.data)
            maxLevelR = level
        }

        eitherSideVisible(root[first], level + 1, side)
        eitherSideVisible(root[second], level + 1, side)
    }
    return { eitherSideVisible }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Diameter of the tree recursive
    const diameterOfTree = (root) => {
      if (!root) {
        return 0
      }
      let diameterLeft = diameterOfTree(root.left)
      let diameterRight = diameterOfTree(root.right)
      let diameterRoot = heightOfThree(root.left) + heightOfThree(root.right) + 1
      return Math.max(diameterLeft, Math.max(diameterRight, diameterRoot))
    }
    return { diameterOfTree }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Height of Tree
    const heightOfTree = (root) => {
        if (!root) {
            return 0
        }

        const leftHeight = heightOfTree(root.left)
        const rightHeight = heightOfTree(root.right)

        return Math.max(leftHeight, rightHeight) + 1
    }
    return { heightOfTree }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Search for given node and return only its children's data
    const searchNode = (root, queryData) => {

        if (!queryData || !root) {
            return null
        }

        if (root.data === queryData) {
            return {
                data: root.data,
                left: root.left?.data || null,
                right: root.right?.data || null,
            }
        }

        const leftSearch = searchNode(root.left, queryData)
        const rightSearch = searchNode(root.right, queryData)
        return leftSearch || rightSearch
    }
    return { searchNode }
}
```

```
const BinaryTree = () => {
    let index = -1
    let tree = null

    // Sum of Nodes
    const sumOfNodes = (root) => {
        if (!root) {
            return 0
        }

        const leftNodes = sumOfNodes(root.left)
        const rightNods = sumOfNodes(root.right)

        return leftNodes + rightNods + root.data
    }
    return { sumOfNodes }
}
```

```
const Stack = () => {
    let items = []

    const push = (item) => {
        items.push(item)
        return
    }

    const pop = () => {
        if (items.length == 0)
            return "Stack is Empty";
        return items.pop()
    }

    const pick = () => {
        return items[items.length - 1]
    }

    const isEmpty = () => {
        return items.length === 0
    }

    const printStack = () => {
        let item = ''
        for (let i = 0; i < items.length; i++) {
            item += items[i] + (i == items.length - 1 ? '' : ' ~> ')
        }
        console.log(item)

    }
    return { push, pop, pick, isEmpty, printStack }
}
```

```
const postFix = (expression) => {
    const { push, pop } = Stack()

    for (let i = 0; i < expression.length; i++) {
        const item = expression[i];
        if (!isNaN(item)) {
            push(Number(item))
        } else {
            let value1 = pop()
            let value2 = pop()
            if (value1 !== 'Stack is Empty' && value2 !== 'Stack is Empty') {

                switch (item) {
                    case '+': {
                        push(value1 + value2)
                        break
                    }
                    case '-': {
                        push(value2 - value1)
                        break
                    }
                    case '/': {
                        push(value2 / value1)
                        break
                    }
                    case '*': {
                        push(value1 * value2)
                        break
                    }
                }
            }
            else return 'Stack is Empty'
        }
    }
    return pop();
}

console.log(postFix("123-+4+")); //# 4
console.log(postFix("43*1+"));   //# 13
console.log(postFix("43*1-+"));  //# Stack is Empty
```

```
const { push, pop, pick, isEmpty, printStack } = Stack()

push(2)
push(4)
push(6)
push(12)
push(654)
const picked = pick()
console.log('Log: ~> file: js ~> line 43 ~> picked', picked)
const popped = pop()
console.log('Log: ~> file: js ~> line 45 ~> popped', popped)
const isEmptyStack = isEmpty()
console.log('Log: ~> file: js ~> line 47 ~> isEmptyStack', isEmptyStack)
printStack()

//# OUTPUT
// Log: ~> file: js ~> line 43 ~> picked 654
// Log: ~> file: js ~> line 45 ~> popped 654
// Log: ~> file: js ~> line 47 ~> isEmptyStack false
// 2 ~> 4 ~> 6 ~> 12
```