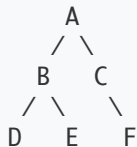


Tree Data Structure - Complete Notes

What is a Tree?

A **tree** is a hierarchical data structure consisting of nodes connected by edges. Unlike linear data structures (arrays, linked lists), trees are non-linear and represent hierarchical relationships.

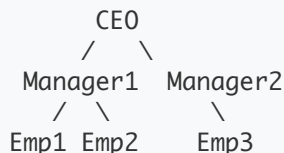
Example:



This represents a family tree, organization structure, or file system where:

- A is at the top
- B and C are connected to A
- D and E are connected to B
- F is connected to C

Visual Representation:



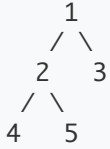
Uses of Tree Data Structure

1. **File Systems** - Folders and files in Windows/Mac/Linux
 2. **DOM (Document Object Model)** - HTML structure in web pages
 3. **Database Indexing** - B-trees in databases for fast search
 4. **Decision Making** - Decision trees in AI/ML
 5. **Autocompletion** - Trie trees in search engines
 6. **Expression Parsing** - Compilers use expression trees
 7. **Routing Tables** - Network routing algorithms
 8. **Hierarchical Data** - Organization charts, family trees
-

Types of Trees

1. Binary Tree

A tree where each node has **at most 2 children** (left and right).



JavaScript Implementation:

```

class Node {
  constructor(data) {
    this.data = data;
    this.left = null;
    this.right = null;
  }
}

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

  // Insert nodes level by level
  insert(data) {
    const newNode = new Node(data);

    if (!this.root) {
      this.root = newNode;
      return;
    }

    const queue = [this.root];
    while (queue.length > 0) {
      const current = queue.shift();

      if (!current.left) {
        current.left = newNode;
        return;
      } else {
        queue.push(current.left);
      }

      if (!current.right) {
        current.right = newNode;
        return;
      } else {
        queue.push(current.right);
      }
    }
  }
}

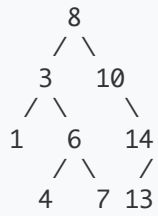
// Usage
const tree = new BinaryTree();
tree.insert(1);
tree.insert(2);
tree.insert(3);
tree.insert(4);
tree.insert(5);

```

2. Binary Search Tree (BST)

A binary tree with ordering property:

- **Left child < Parent**
- **Right child > Parent**



JavaScript Implementation:

```

class BSTNode {
  constructor(data) {
    this.data = data;
    this.left = null;
    this.right = null;
  }
}

class BinarySearchTree {
  constructor() {
    this.root = null;
  }

  insert(data) {
    const newNode = new BSTNode(data);

    if (!this.root) {
      this.root = newNode;
      return;
    }

    this._insertNode(this.root, newNode);
  }

  _insertNode(node, newNode) {
    if (newNode.data < node.data) {
      if (!node.left) {
        node.left = newNode;
      } else {
        this._insertNode(node.left, newNode);
      }
    } else {
      if (!node.right) {
        node.right = newNode;
      } else {
        this._insertNode(node.right, newNode);
      }
    }
  }

  search(data) {
    return this._searchNode(this.root, data);
  }

  _searchNode(node, data) {
    if (!node) return false;

    if (data < node.data) {
      return this._searchNode(node.left, data);
    } else if (data > node.data) {
      return this._searchNode(node.right, data);
    } else {
      return true; // Found
    }
  }

  findMin(node = this.root) {
    if (!node) return null;
    while (node.left) {
      node = node.left;
    }
    return node.data;
  }

  findMax(node = this.root) {
    if (!node) return null;
    while (node.right) {

```

```

        node = node.right;
    }
    return node.data;
}
}

// Usage
const bst = new BinarySearchTree();
bst.insert(8);
bst.insert(3);
bst.insert(10);
bst.insert(1);
bst.insert(6);
bst.insert(14);
bst.insert(4);
bst.insert(7);
bst.insert(13);

console.log(bst.search(6)); // true
console.log(bst.search(15)); // false
console.log(bst.findMin()); // 1
console.log(bst.findMax()); // 14

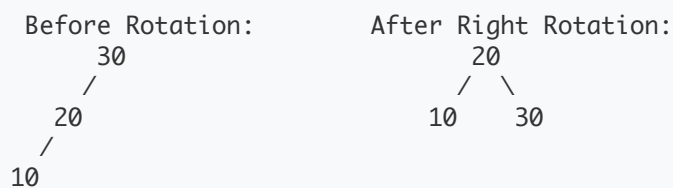
```

Time Complexity:

- Search: $O(\log n)$ average, $O(n)$ worst case
- Insert: $O(\log n)$ average, $O(n)$ worst case
- Delete: $O(\log n)$ average, $O(n)$ worst case

3. AVL Tree (Self-Balancing BST)

A BST that automatically balances itself to maintain $O(\log n)$ operations. The difference between heights of left and right subtrees cannot be more than 1.



Balance Factor = height(left) - height(right)
Must be -1, 0, or 1

JavaScript Implementation:

```

class AVLNode {
  constructor(data) {
    this.data = data;
    this.left = null;
    this.right = null;
    this.height = 1;
  }
}

class AVLTree {
  constructor() {
    this.root = null;
  }

  getHeight(node) {
    return node ? node.height : 0;
  }

  getBalance(node) {
    return node ? this.getHeight(node.left) - this.getHeight(node.right) : 0;
  }

  rightRotate(y) {
    const x = y.left;
    const T2 = x.right;

    x.right = y;
    y.left = T2;

    y.height = Math.max(this.getHeight(y.left), this.getHeight(y.right)) + 1;
    x.height = Math.max(this.getHeight(x.left), this.getHeight(x.right)) + 1;

    return x;
  }

  leftRotate(x) {
    const y = x.right;
    const T2 = y.left;

    y.left = x;
    x.right = T2;

    x.height = Math.max(this.getHeight(x.left), this.getHeight(x.right)) + 1;
    y.height = Math.max(this.getHeight(y.left), this.getHeight(y.right)) + 1;

    return y;
  }

  insert(data) {
    this.root = this._insertNode(this.root, data);
  }

  _insertNode(node, data) {
    if (!node) return new AVLNode(data);

    if (data < node.data) {
      node.left = this._insertNode(node.left, data);
    } else if (data > node.data) {
      node.right = this._insertNode(node.right, data);
    } else {
      return node; // Duplicate values not allowed
    }

    node.height =
      1 + Math.max(this.getHeight(node.left), this.getHeight(node.right));

    const balance = this.getBalance(node);
  }
}

```

```

// Left Left Case
if (balance > 1 && data < node.left.data) {
    return this.rightRotate(node);
}

// Right Right Case
if (balance < -1 && data > node.right.data) {
    return this.leftRotate(node);
}

// Left Right Case
if (balance > 1 && data > node.left.data) {
    node.left = this.leftRotate(node.left);
    return this.rightRotate(node);
}

// Right Left Case
if (balance < -1 && data < node.right.data) {
    node.right = this.rightRotate(node.right);
    return this.leftRotate(node);
}

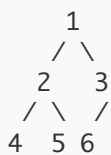
return node;
}
}

// Usage
const avl = new AVLTree();
avl.insert(10);
avl.insert(20);
avl.insert(30);
avl.insert(40);
avl.insert(50);
avl.insert(25);

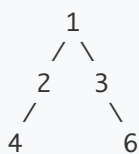
```

4. Complete Binary Tree

Every level is completely filled except possibly the last level, which is filled from left to right.



✓ Complete Binary Tree

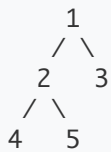


x NOT Complete (gap in level)

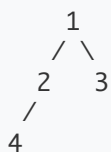
Used in: Heap data structure

5. Full Binary Tree

Every node has either 0 or 2 children (no node has only 1 child).



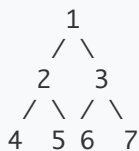
✓ Full Binary Tree



✗ NOT Full (node 2 has only 1 child)

6. Perfect Binary Tree

All internal nodes have 2 children and all leaves are at the same level.



✓ Perfect Binary Tree

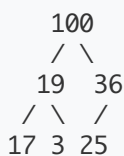
Properties:

- Total nodes = $2^h - 1$ (where h is height)
- Leaf nodes = $2^{(h-1)}$

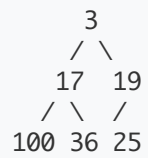
7. Binary Heap (Min Heap & Max Heap)

A complete binary tree where each node follows heap property.

Max Heap: Parent \geq Children



Min Heap: Parent \leq Children



JavaScript Implementation:

```

class MinHeap {
  constructor() {
    this.heap = [];
  }

  getParentIndex(i) {
    return Math.floor((i - 1) / 2);
  }
  getLeftChildIndex(i) {
    return 2 * i + 1;
  }
  getRightChildIndex(i) {
    return 2 * i + 2;
  }

  swap(i, j) {
    [this.heap[i], this.heap[j]] = [this.heap[j], this.heap[i]];
  }

  insert(value) {
    this.heap.push(value);
    this.heapifyUp(this.heap.length - 1);
  }

  heapifyUp(index) {
    let currentIndex = index;

    while (currentIndex > 0) {
      const parentIndex = this.getParentIndex(currentIndex);

      if (this.heap[currentIndex] < this.heap[parentIndex]) {
        this.swap(currentIndex, parentIndex);
        currentIndex = parentIndex;
      } else {
        break;
      }
    }
  }

  extractMin() {
    if (this.heap.length === 0) return null;
    if (this.heap.length === 1) return this.heap.pop();

    const min = this.heap[0];
    this.heap[0] = this.heap.pop();
    this.heapifyDown(0);

    return min;
  }

  heapifyDown(index) {
    let smallest = index;
    const left = this.getLeftChildIndex(index);
    const right = this.getRightChildIndex(index);

    if (left < this.heap.length && this.heap[left] < this.heap[smallest]) {
      smallest = left;
    }

    if (right < this.heap.length && this.heap[right] < this.heap[smallest]) {
      smallest = right;
    }

    if (smallest !== index) {
      this.swap(index, smallest);
      this.heapifyDown(smallest);
    }
  }
}

```

```

    }

    peek() {
        return this.heap.length > 0 ? this.heap[0] : null;
    }
}

// Usage
const minHeap = new MinHeap();
minHeap.insert(3);
minHeap.insert(17);
minHeap.insert(19);
minHeap.insert(100);
minHeap.insert(36);
minHeap.insert(25);

console.log(minHeap.extractMin()); // 3
console.log(minHeap.peak()); // 17

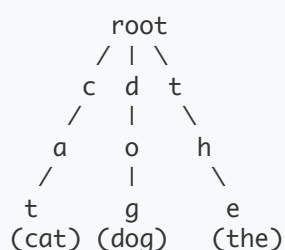
```

Applications:

- Priority Queue
- Heap Sort
- Dijkstra's Algorithm
- Finding K largest/smallest elements

8. Trie (Prefix Tree)

A tree used to store strings where each node represents a character. Commonly used for autocomplete and spell checking.



JavaScript Implementation:

```

class TrieNode {
  constructor() {
    this.children = {};
    this.isEndOfWord = false;
  }
}

class Trie {
  constructor() {
    this.root = new TrieNode();
  }

  insert(word) {
    let node = this.root;

    for (let char of word) {
      if (!node.children[char]) {
        node.children[char] = new TrieNode();
      }
      node = node.children[char];
    }

    node.isEndOfWord = true;
  }

  search(word) {
    let node = this.root;

    for (let char of word) {
      if (!node.children[char]) {
        return false;
      }
      node = node.children[char];
    }

    return node.isEndOfWord;
  }

  startsWith(prefix) {
    let node = this.root;

    for (let char of prefix) {
      if (!node.children[char]) {
        return false;
      }
      node = node.children[char];
    }

    return true;
  }

  getAllWords(node = this.root, prefix = "", words = []) {
    if (node.isEndOfWord) {
      words.push(prefix);
    }

    for (let char in node.children) {
      this.getAllWords(node.children[char], prefix + char, words);
    }

    return words;
  }

  autocomplete(prefix) {
    let node = this.root;

    for (let char of prefix) {

```

```

    if (!node.children[char]) {
        return [];
    }
    node = node.children[char];
}

return this.getAllWords(node, prefix);
}
}

// Usage
const trie = new Trie();
trie.insert("apple");
trie.insert("app");
trie.insert("application");
trie.insert("apply");
trie.insert("banana");

console.log(trie.search("app")); // true
console.log(trie.search("appl")); // false
console.log(trie.startsWith("app")); // true
console.log(trie.autocomplete("app")); // ["app", "apple", "application", "apply"]

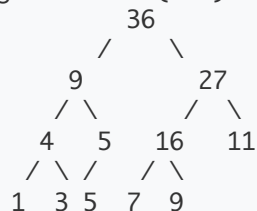
```

9. Segment Tree

Used for range queries (sum, min, max) on arrays.

Array: [1, 3, 5, 7, 9, 11]

Segment Tree (sum):



JavaScript Implementation:

```

class SegmentTree {
  constructor(arr) {
    this.n = arr.length;
    this.tree = new Array(4 * this.n);
    this.build(arr, 0, 0, this.n - 1);
  }

  build(arr, node, start, end) {
    if (start === end) {
      this.tree[node] = arr[start];
      return;
    }

    const mid = Math.floor((start + end) / 2);
    const leftChild = 2 * node + 1;
    const rightChild = 2 * node + 2;

    this.build(arr, leftChild, start, mid);
    this.build(arr, rightChild, mid + 1, end);

    this.tree[node] = this.tree[leftChild] + this.tree[rightChild];
  }

  query(left, right) {
    return this._query(0, 0, this.n - 1, left, right);
  }

  _query(node, start, end, left, right) {
    if (right < start || left > end) {
      return 0;
    }

    if (left <= start && end <= right) {
      return this.tree[node];
    }

    const mid = Math.floor((start + end) / 2);
    const leftChild = 2 * node + 1;
    const rightChild = 2 * node + 2;

    const leftSum = this._query(leftChild, start, mid, left, right);
    const rightSum = this._query(rightChild, mid + 1, end, left, right);

    return leftSum + rightSum;
  }

  update(index, value) {
    this._update(0, 0, this.n - 1, index, value);
  }

  _update(node, start, end, index, value) {
    if (start === end) {
      this.tree[node] = value;
      return;
    }

    const mid = Math.floor((start + end) / 2);
    const leftChild = 2 * node + 1;
    const rightChild = 2 * node + 2;

    if (index <= mid) {
      this._update(leftChild, start, mid, index, value);
    } else {
      this._update(rightChild, mid + 1, end, index, value);
    }

    this.tree[node] = this.tree[leftChild] + this.tree[rightChild];
  }
}

```

```

    }
  }

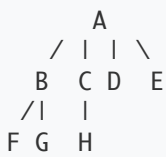
  // Usage
  const arr = [1, 3, 5, 7, 9, 11];
  const segTree = new SegmentTree(arr);

  console.log(segTree.query(1, 4)); // 24 (3 + 5 + 7 + 9)
  segTree.update(2, 10);
  console.log(segTree.query(1, 4)); // 29 (3 + 10 + 7 + 9)

```

10. N-ary Tree

A tree where each node can have up to N children.



JavaScript Implementation:


```

class NaryNode {
  constructor(data) {
    this.data = data;
    this.children = [];
  }
}

class NaryTree {
  constructor() {
    this.root = null;
  }

  // Level order traversal
  levelOrder() {
    if (!this.root) return [];

    const result = [];
    const queue = [this.root];

    while (queue.length > 0) {
      const node = queue.shift();
      result.push(node.data);

      for (let child of node.children) {
        queue.push(child);
      }
    }

    return result;
  }

  // Find max depth
  maxDepth(node = this.root) {
    if (!node) return 0;

    let maxChildDepth = 0;
    for (let child of node.children) {
      maxChildDepth = Math.max(maxChildDepth, this.maxDepth(child));
    }

    return 1 + maxChildDepth;
  }
}

// Usage
const tree = new NaryTree();
tree.root = new NaryNode(1);
tree.root.children.push(new NaryNode(2));
tree.root.children.push(new NaryNode(3));
tree.root.children.push(new NaryNode(4));
tree.root.children[0].children.push(new NaryNode(5));
tree.root.children[0].children.push(new NaryNode(6));

console.log(tree.levelOrder()); // [1, 2, 3, 4, 5, 6]
console.log(tree.maxDepth()); // 3

```

Binary Trees vs Arrays vs Linked Lists

Operation	Array	Linked List	Binary Search Tree
Search	$O(n)$ or $O(\log n)$ if sorted	$O(n)$	$O(\log n)$ average

Operation	Array	Linked List	Binary Search Tree
Insert	$O(n)$ - shift elements	$O(1)$ at head, $O(n)$ otherwise	$O(\log n)$ average
Delete	$O(n)$ - shift elements	$O(1)$ at head, $O(n)$ otherwise	$O(\log n)$ average
Access by index	$O(1)$	$O(n)$	N/A
Memory	Contiguous	Non-contiguous	Non-contiguous
Sorted order	Can be sorted	Requires traversal	Naturally sorted

When to Use What?

Use Arrays when:

- Need fast random access by index
- Size is known and fixed
- Memory locality is important

Use Linked Lists when:

- Frequent insertions/deletions at beginning
- Size is dynamic
- Don't need random access

Use Binary Search Trees when:

- Need sorted data
- Frequent insertions, deletions, AND searches
- Need range queries
- Want balanced performance

Example Comparison:

```
// Array - Fast access, slow insertion
const arr = [1, 2, 3, 4, 5];
console.log(arr[2]); // 0(1) - Fast
arr.splice(2, 0, 10); // 0(n) - Slow, shifts elements

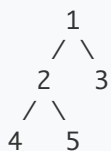
// Linked List - Slow access, fast insertion at head
class ListNode {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
let head = new ListNode(1);
head.next = new ListNode(2);
// Access 3rd element: 0(n) - Must traverse
// Insert at head: 0(1) - Fast

// BST - Balanced performance
const bst = new BinarySearchTree();
bst.insert(5);
bst.insert(3);
bst.insert(7);
bst.search(3); // 0(log n) - Good
bst.insert(4); // 0(log n) - Good
```

Tree Traversals

1. Preorder Traversal (Root → Left → Right)

Process: Visit root first, then left subtree, then right subtree.



Preorder: 1, 2, 4, 5, 3

JavaScript Code:

```

function preorderTraversal(root) {
  const result = [];

  function traverse(node) {
    if (!node) return;

    result.push(node.data); // Root
    traverse(node.left); // Left
    traverse(node.right); // Right
  }

  traverse(root);
  return result;
}

// Iterative approach using stack
function preorderIterative(root) {
  if (!root) return [];

  const result = [];
  const stack = [root];

  while (stack.length > 0) {
    const node = stack.pop();
    result.push(node.data);

    if (node.right) stack.push(node.right); // Push right first
    if (node.left) stack.push(node.left); // Then left
  }

  return result;
}

```

Use Cases:

- Copy a tree
- Get prefix expression
- Serialize a tree

2. Inorder Traversal (Left → Root → Right)

Process: Visit left subtree first, then root, then right subtree.



Inorder: 1, 2, 3, 4, 5, 6, 7
(Sorted order in BST!)

JavaScript Code:

```
function inorderTraversal(root) {
  const result = [];

  function traverse(node) {
    if (!node) return;

    traverse(node.left); // Left
    result.push(node.data); // Root
    traverse(node.right); // Right
  }

  traverse(root);
  return result;
}

// Iterative approach using stack
function inorderIterative(root) {
  const result = [];
  const stack = [];
  let current = root;

  while (current || stack.length > 0) {
    while (current) {
      stack.push(current);
      current = current.left;
    }

    current = stack.pop();
    result.push(current.data);
    current = current.right;
  }

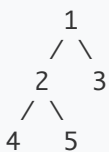
  return result;
}
```

Use Cases:

- Get sorted elements from BST
- Check if tree is BST
- Find kth smallest element

3. Postorder Traversal (Left → Right → Root)

Process: Visit left subtree first, then right subtree, then root.



Postorder: 4, 5, 2, 3, 1

JavaScript Code:

```

function postorderTraversal(root) {
  const result = [];

  function traverse(node) {
    if (!node) return;

    traverse(node.left); // Left
    traverse(node.right); // Right
    result.push(node.data); // Root
  }

  traverse(root);
  return result;
}

// Iterative approach using two stacks
function postorderIterative(root) {
  if (!root) return [];

  const result = [];
  const stack1 = [root];
  const stack2 = [];

  while (stack1.length > 0) {
    const node = stack1.pop();
    stack2.push(node);

    if (node.left) stack1.push(node.left);
    if (node.right) stack1.push(node.right);
  }

  while (stack2.length > 0) {
    result.push(stack2.pop().data);
  }

  return result;
}

```

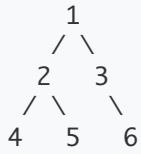
Use Cases:

- Delete a tree
- Get postfix expression
- Calculate directory size

Traversal Exercises

Exercise 1: Level Order Traversal (BFS)

Problem: Visit tree level by level from left to right.



Output: `[[1], [2, 3], [4, 5, 6]]`

Solution:

```

function levelOrder(root) {
  if (!root) return [];

  const result = [];
  const queue = [root];

  while (queue.length > 0) {
    const levelSize = queue.length;
    const currentLevel = [];

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();
      currentLevel.push(node.data);

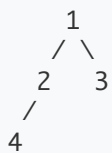
      if (node.left) queue.push(node.left);
      if (node.right) queue.push(node.right);
    }

    result.push(currentLevel);
  }

  return result;
}
  
```

Exercise 2: Find Maximum Depth

Problem: Find the maximum depth (height) of the tree.



Output: 3

Solution:

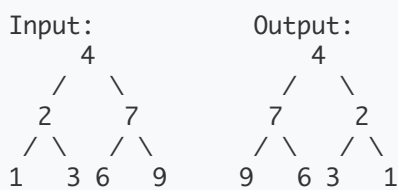
```
function maxDepth(root) {
  if (!root) return 0;

  const leftDepth = maxDepth(root.left);
  const rightDepth = maxDepth(root.right);

  return 1 + Math.max(leftDepth, rightDepth);
}
```

Exercise 3: Invert Binary Tree

Problem: Mirror the tree (swap left and right children).



Solution:

```
function invertTree(root) {
  if (!root) return null;

  // Swap children
  [root.left, root.right] = [root.right, root.left];

  // Recursively invert subtrees
  invertTree(root.left);
  invertTree(root.right);

  return root;
}
```

Exercise 4: Check if Same Tree

Problem: Check if two trees are identical.

Solution:

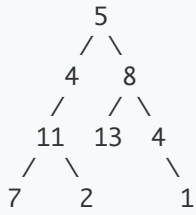
```
function isSameTree(p, q) {
  if (!p && !q) return true;
  if (!p || !q) return false;

  if (p.data !== q.data) return false;

  return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```


Exercise 5: Path Sum

Problem: Check if there's a root-to-leaf path with a given sum.



Target: 22

Output: true (5 → 4 → 11 → 2)

Solution:

```

function hasPathSum(root, targetSum) {
  if (!root) return false;

  // Leaf node check
  if (!root.left && !root.right) {
    return root.data === targetSum;
  }

  const remainingSum = targetSum - root.data;
  return (
    hasPathSum(root.left, remainingSum) || hasPathSum(root.right, remainingSum)
  );
}
  
```

Exercise 6: Lowest Common Ancestor (LCA) in BST

Problem: Find the lowest common ancestor of two nodes in a BST.



LCA(2, 8) = 6

LCA(2, 4) = 2

Solution:

```
function lowestCommonAncestor(root, p, q) {
  if (!root) return null;

  // Both nodes are in left subtree
  if (p < root.data && q < root.data) {
    return lowestCommonAncestor(root.left, p, q);
  }

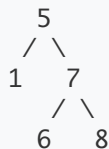
  // Both nodes are in right subtree
  if (p > root.data && q > root.data) {
    return lowestCommonAncestor(root.right, p, q);
  }

  // One node is on left, other on right (or one is root)
  return root.data;
}
```

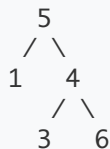
Exercise 7: Validate Binary Search Tree

Problem: Check if a binary tree is a valid BST.

Valid BST:



Invalid BST:



Solution:

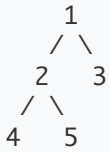
```
function isValidBST(root, min = -Infinity, max = Infinity) {
  if (!root) return true;

  // Check if current node violates BST property
  if (root.data <= min || root.data >= max) {
    return false;
  }

  // Check left subtree (values must be < root.data)
  // Check right subtree (values must be > root.data)
  return (
    isValidBST(root.left, min, root.data) &&
    isValidBST(root.right, root.data, max)
  );
}
```

Exercise 8: Diameter of Binary Tree

Problem: Find the longest path between any two nodes.



Diameter: 3 (4 → 2 → 1 → 3 or 5 → 2 → 1 → 3)

Solution:

```

function diameterOfBinaryTree(root) {
  let diameter = 0;

  function height(node) {
    if (!node) return 0;

    const leftHeight = height(node.left);
    const rightHeight = height(node.right);

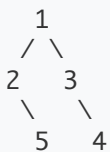
    // Update diameter if path through this node is longer
    diameter = Math.max(diameter, leftHeight + rightHeight);

    return 1 + Math.max(leftHeight, rightHeight);
  }

  height(root);
  return diameter;
}
  
```

Exercise 9: Binary Tree Right Side View

Problem: Get values visible from the right side of the tree.



Output: [1, 3, 4]

Solution:

```

function rightSideView(root) {
  if (!root) return [];

  const result = [];
  const queue = [root];

  while (queue.length > 0) {
    const levelSize = queue.length;

    for (let i = 0; i < levelSize; i++) {
      const node = queue.shift();

      // Last node in current level
      if (i === levelSize - 1) {
        result.push(node.data);
      }

      if (node.left) queue.push(node.left);
      if (node.right) queue.push(node.right);
    }
  }

  return result;
}

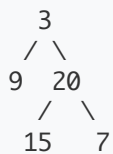
```

Exercise 10: Construct Binary Tree from Inorder and Preorder

Problem: Build tree from traversal arrays.

Preorder: [3, 9, 20, 15, 7]
 Inorder: [9, 3, 15, 20, 7]

Output Tree:



Solution:

```
function buildTree(preorder, inorder) {  
  if (preorder.length === 0) return null;  
  
  const rootVal = preorder[0];  
  const root = new Node(rootVal);  
  
  const rootIndex = inorder.indexOf(rootVal);  
  
  // Split arrays for left and right subtrees  
  const leftInorder = inorder.slice(0, rootIndex);  
  const rightInorder = inorder.slice(rootIndex + 1);  
  
  const leftPreorder = preorder.slice(1, 1 + leftInorder.length);  
  const rightPreorder = preorder.slice(1 + leftInorder.length);  
  
  root.left = buildTree(leftPreorder, leftInorder);  
  root.right = buildTree(rightPreorder, rightInorder);  
  
  return root;  
}
```

Complete Working Example

Here's a complete example demonstrating all concepts:

```
// Node class
class TreeNode {
  constructor(data) {
    this.data = data;
    this.left = null;
    this.right = null;
  }
}

// Create a sample tree
//      1
//     /\
//    2  3
//   /\  \
//  4  5  6

const root = new TreeNode(1);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
root.left.left = new TreeNode(4);
root.left.right = new TreeNode(5);
root.right.right = new TreeNode(6);

// Test all traversals
console.log("Preorder:", preorderTraversal(root)); // [1, 2, 4, 5, 3, 6]
console.log("Inorder:", inorderTraversal(root)); // [4, 2, 5, 1, 3, 6]
console.log("Postorder:", postorderTraversal(root)); // [4, 5, 2, 6, 3, 1]
console.log("Level Order:", levelOrder(root)); // [[1], [2, 3], [4, 5, 6]]

// Test other functions
console.log("Max Depth:", maxDepth(root)); // 3
console.log("Has Path Sum (12):", hasPathSum(root, 7)); // true (1→2→4)
console.log("Right Side View:", rightSideView(root)); // [1, 3, 6]

// Create and test BST
const bst = new BinarySearchTree();
[8, 3, 10, 1, 6, 14, 4, 7, 13].forEach((val) => bst.insert(val));

console.log("BST Search 6:", bst.search(6)); // true
console.log("BST Min:", bst.findMin()); // 1
console.log("BST Max:", bst.findMax()); // 14

// Create and test MinHeap
const heap = new MinHeap();
[5, 3, 7, 1, 9, 2].forEach((val) => heap.insert(val));

console.log("Heap Extract Min:", heap.extractMin()); // 1
console.log("Heap Peek:", heap.peek()); // 2

// Create and test Trie
const trie = new Trie();
["apple", "app", "application", "apply"].forEach((word) => trie.insert(word));

console.log("Trie Search 'app':", trie.search("app")); // true
console.log("Trie Autocomplete 'app':", trie.autocomplete("app"));
// ["app", "apple", "application", "apply"]
```

Interview Tips

Common Interview Questions

1. Traversal Questions:

- Print tree in spiral/zigzag order
- Vertical order traversal
- Boundary traversal

2. BST Questions:

- Find kth smallest/largest element
- Convert sorted array to BST
- Delete node in BST

3. Path Questions:

- All root-to-leaf paths
- Maximum path sum
- Path with given sum

4. Structure Questions:

- Serialize and deserialize tree
- Check if balanced
- Mirror tree

Time Complexity Reference

Operation	Binary Tree	BST (balanced)	BST (skewed)
Search	$O(n)$	$O(\log n)$	$O(n)$
Insert	$O(n)$	$O(\log n)$	$O(n)$
Delete	$O(n)$	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$	$O(n)$

Key Points to Remember

1. **Recursion is your friend** - Most tree problems are solved recursively
2. **Base cases** - Always handle null nodes
3. **BST property** - Left < Root < Right
4. **Traversal order matters** - Choose based on problem requirement
5. **Use helper functions** - Keep main function clean
6. **Queue for level order** - Stack for DFS, Queue for BFS
7. **Practice drawing** - Visualize before coding

8. Consider edge cases - Empty tree, single node, skewed tree

Common Patterns

1. DFS Pattern (Recursion):

```
function dfs(node) {  
  if (!node) return;  
  // Process node  
  dfs(node.left);  
  dfs(node.right);  
}
```

2. BFS Pattern (Queue):

```
function bfs(root) {  
  const queue = [root];  
  while (queue.length > 0) {  
    const node = queue.shift();  
    // Process node  
    if (node.left) queue.push(node.left);  
    if (node.right) queue.push(node.right);  
  }  
}
```

3. Divide and Conquer:

```
function solve(root) {  
  if (!root) return baseCase;  
  const left = solve(root.left);  
  const right = solve(root.right);  
  return combine(left, right);  
}
```

Summary

Trees are powerful hierarchical data structures with many real-world applications. Master the basics:

1. **Understand different tree types** - Binary, BST, AVL, Heap, Trie
2. **Practice traversals** - Preorder, Inorder, Postorder, Level Order
3. **Learn recursion** - Essential for tree problems
4. **Know time complexities** - Important for interviews
5. **Code regularly** - Practice common patterns

Next Steps:

- Solve 50+ tree problems on LeetCode
- Implement all tree types from scratch

- Practice drawing trees while solving
- Learn advanced topics: Red-Black Trees, B-Trees, Splay Trees

Good luck with your interviews! 🚀