# Chapter 4 - Trees & Graphs

// **Trees**

```
class Node {
	public String name;
	public Node[] children;
}
```

// **Trees vs. Binary Trees vs. Binary Search Tree**

- a binary tree is a tree in which each node has up to two children
- a node is called a "leaf" node if it has no children
- a binary search tree is a binary tree in which every node fits a specific ordering property: all left descendants <= n < all right descendants

// **Balanced vs. Unbalanced**

- balanced enough to ensure O(log N) times for insert and find but it's not necessarily as balanced as it could be
- Red Blac Trees & AVL trees

// **Complete Binary Tree** → every level of the tree is filled except for the last level (leafs)

// **Full Binary Tree** → is a binary tree in which every node has either zero or two children; no nodes have only one child.

// **Perfect Binary Trees** → is a binary tree that is both full and complete; it has exactly (2^k - 1) nodes, where k is the number of levels

// **Binary Tree Traversal**

// **In-Order Traversal** → visit the left branch, then the current node and then the right branch
// When performed on a binary search tree, it visits the nodes in ascending order

```
void inOrderTraversal(Node current) {
	if (node != null) {
		inOrderTraversal(current.left);
		visit(current);
		inOrderTraversal(current.right);
	}
}
```

// **Pre-Order Traversal** → visits the current node before its child nodes

// In a pre-order traversal, the root is always the first node visited

```
void preOrderTraversal(Node current) {
	if (node != null) {
		visit(current);
		preOrderTraversal(current.left);
		preOrderTraversal(current.right);
	}
}
```

// **Post-Order Traversal** → visits the current node after its child nodes
// In a post-order traversal, the root is always the last node visited

```
void postOrderTraversal(Node current) {
	if (node != null) {
		postOrderTraversal(current.left);
		postOrderTraversal(current.right);
		visit(current);
	}
}
```

// **Binary Heaps** → Min-Heaps and Max-Heaps
// A min-heap is a complete binary tree where each node is smaller than its children → the root is the smallest element in the tree

// We have two operations on a min-heap → insert and extract_min

// **Insert** → when we insert into a min heap we always add the element at the bottom of the tree (the rightmost spot so we maintain the complete tree property) → the we bubble up the element (swim) until it's higher than all its parents and smaller than all its children → this takes O(log n) time where n is the number of nodes in the heap.

// **Delete Min (Extract)** → the minimum element is always at the top → we remove it and swap it with the last element → then we bubble down the top (sink) until the min-heap property is restored → this algorithm also takes O(log n) time

// **Tries** → prefix tree; it's a variant of a n-ary tree in which characters are stored at each node → each path down the tree may represent a word

// The null nodes are often used to indicate complete words

// Very commonly a trie is used to store the entire language for quick prefix lookups

// A trie can check if a string is a valid prefix in O(k) time where k is the length of the string

// **Graphs**

- a tree is actually a type of graph, but not all graphs are trees
- a tree is a connected graph without cycles
- a graph is simply a collection of nodes with edges between (some of) them
- graphs can be either directed or undirected → directed edges are like a one-way street and undirected edges are like a two-way street
- a graph might consist of multiple isolated subgraphs → if there is a path between every pair of vertices it is called a connected graph
- a graph can have cycles → an acyclic graph is one without cycles

// **Adjacency List** → most common way to represent a graph; every vertex (or node) stores a list of adjacent vertices → in an undirected graph an edge like (v, w) would be stored twice

// **Adjacency Matrices** → n x n matrix where n is the number of nodes if matrix[i][j] == true → indicates an edge from i to j → in an undirected graph an adjacency matrix will be symmetric

// **Graph Search**

// **Depth-First Search (DFS)** → we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch → that it we go deep first before we go wide

// **Bradth-First Search (BFS)** → we start at the root (or another arbitrarily selected node) and explore each neighbour before going into any of their children. We go wide first before we go deep.

// **DFS** is often preferred if we want to visit every node in the graph
// **BFS** is generally better if we want to find the shortest path (or just any path)

// **DFS** → we visit a node a and then iterate through each of a's neighbours → when visiting a node b which is a neighbour of a, we visit all of b's neighbours first before going on to a's other neighbours → a exhaustively searches b's branch before any of its neighbours

// Tree traversals are a form of DFS → in graphs we need to check if a node was visited otherwise we could be caught in a infinite loop (graphs can have cycles)

```
void dfs(Node root) {
    if (root == null) {
        return;
    }
    visit(root);
```

```
        root.visited = true;
        for each (Node n in root.adjacent) {
                if (n.visited == false) {
                        dfs(n);
                }
        }
}
```

// **BFS** → it's not a recursive algorithms; it uses a **queue** → node a visits each of a's neighbours before visiting any of their neighbours → searching level by level out from a

```
void bfs(Node root) {
        Queue queue = new Queue();
        root.marked = true;
        queue.enqueue(root);

        while (!queue.isEmpty()) {
                Node r = queue.dequeue();
                visit(r);
                for each (Node n in r.adjacent) {
                        if (n.marked == false) {
                                n.marked = true;
                                queue.enqueue(n);
                        }
                }
        }
}
```

// **Bidirectional Search** → used to find the shortest path between a source and a destination node → it operates by running two BFS simultaneously (one from each node) → when their searches collide we have found a path

// **Route Between Nodes** → given a directed graph, design an algorithm to find out if there is a route between two nodes.

```
public class Question {

        public static boolean isRouteBetween(Graph g, int a, int b) {
                if (a == b) {
                        return true;
                }

                Queue<Integer> q = new Queue<>();
```

```
		boolean[] marked = new boolean[g.vertices()];

		q.add(a);
		marked[a] = true;
		while (!q.isEmpty()) {
			int v = q.remove();
			for (int w : g.adjacent(v)) {
				if (!marked[w]) {
					if (w == b) {
						return true;
					} else {
						marked[w] = true;
						q.add(w);
					}
				}
			}
			marked[v] = true;
		}
		return false;
	}
}
```

// **Minimal Tree** → given a sorted (ASC) array with unique integer elements create BST with minimal height

```
public class TreeNode {
  public int data;
  public TreeNode left;
  public TreeNode right;
  public TreeNode parent;
  private int size = 0;

  public TreeNode(int d) {
	data = d;
	size = 1;
  }

  private void setLeftChild(TreeNode left) {
	this.left = left;
	if (left != null) {
		left.parent = this;
	}
  }
```

```
private void setRightChild(TreeNode right) {
	this.right = right;
	if (right != null) {
		right.parent = this;
	}
}

public void insertInOrder(int d) {
	if (d <= data) {
		if (left == null) {
			setLeftChild(new TreeNode(d));
		} else {
			left.insertInOrder(d);
		}
	} else {
		if (right == null) {
			setRightChild(new TreeNode(d));
		} else {
			right.insertInOrder(d);
		}
	}
	size++;
}

public int size() {
	return size;
}

public boolean isBST() {
	if (left != null) {
		if (data < left.data || !left.isBST()) {
			return false;
		}
	}

	if (right != null) {
		if (data >= right.data || !right.isBST()) {
			return false;
		}
	}

	return true;
```

```
  }

  public int height() {
        int leftHeight = left != null ? left.height() : 0;
        int rightHeight = right != null ? right.height() : 0;
        return 1 + Math.max(leftHeight, rightHeight);
  }

  public TreeNode find(int d) {
        if (d == data) {
              return this;
        } else if (d <= data) {
              return left != null ? left.find(d) : null;
        } else if (d > data) {
              return right != null ? right.find(d) : null;
        }
        return null;
  }

  private static TreeNode createMinimalBST(int arr[], int start, int end){
        if (end < start) {
              return null;
        }
        int mid = (start + end) / 2;
        TreeNode n = new TreeNode(arr[mid]);
        n.setLeftChild(createMinimalBST(arr, start, mid - 1));
        n.setRightChild(createMinimalBST(arr, mid + 1, end));
        return n;
  }

  public static TreeNode createMinimalBST(int array[]) {
        return createMinimalBST(array, 0, array.length - 1);
  }
}
```

// **List of Depths** → given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth

```
public class QuestionDFS {

        public static void createLevelLinkedList(TreeNode root, List<LinkedList<TreeNode>>
lists, int level) {
```

```
		if (root == null) {
			return;
		}

		LinkedList<TreeNode> list = null;
		if (lists.size() == level) {
			list = new LinkedList<>();
			lists.add(list);
		} else {
			list = lists.get(level);
		}
		list.add(root);
		createLevelLinkedList(root.left, lists, level + 1);
		createLevelLinkedList(root.right, lists, level + 1);
	}

	public static ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
		ArrayList<LinkedList<TreeNode>> lists = new ArrayList<>();
		createLevelLinkedList(root, lists, 0);
		return lists;
	}
}

public class QuestionBFS {

	public static ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
		ArrayList<LinkedList<TreeNode>> lists = new ArrayList<>();
		LinkedList<TreeNode> current = new LinkedList<>();
		if (root != null) {
			current.add(root);
		}
		while (current.size() > 0) {
			lists.add(current);
			LinkedList<TreeNode> parents = current;
			current = new LinkedList<>();
			for (TreeNode parent : parents) {
				if (parent.left != null) {
					current.add(parent.left);
				}
				if (parent.right != null) {
					current.add(parent.right);
				}
			}
```

```
        }
        return lists;
    }
}
```

// **Check Balanced** → check if a binary tree is balanced

```
public class QuestionBrute {

    public static int getHeight(TreeNode root) {
        if (root == null) {
            return -1;
        }
        return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
    }

    public static boolean isBalanced(TreeNode root) {
        if (root == null) {
            return true;
        }
        int deltaHeight = getHeight(root.left) - getHeight(root.right);
        if (Math.abs(deltaHeight) > 1) {
            return false;
        } else {
            return isBalanced(root.left) && isBalanced(root.right);
        }
    }
}

public class QuestionImproved {

    public static int getHeight(TreeNode root) {
        if (root == null) {
            return -1;
        }

        int leftHeight = getHeight(root.left);
        if (leftHeight == Integer.MIN_VALUE) {
            return Integer.MIN_VALUE;
        }

        int rightHeight = getHeight(root.right);
```

```
        if (rightHeight == Integer.MIN_VALUE) {
            return Integer.MIN_VALUE;
        }

        int deltaHeight = leftHeight - rightHeight;
        if (Math.abs(deltaHeight) > 1) {
            return Integer.MIN_VALUE;
        } else {
            return Math.max(leftHeight, rightHeight) + 1;
        }
    }

    public static boolean isBalanced(TreeNode root) {
        return getHeight(root) != Integer.MIN_VALUE;
    }
}
```

// **Validate BST** → implement a function to check ig a binary tree is a binary search tree

// **SolutionA** → in order traversal and copy elements into an array → then check if the array is sorted → can't handle duplicates

```
public class QuestionA {
    private static Integer last = null;

    public static boolean checkBST(TreeNode node, boolean isLeft) {
        if (node == null) {
            return true;
        }

        if (! checkBST(node.left, true)) {
            return false;
        }

        // Check current
        if (last != null) {
            if (isLeft) {
                return last <= node.data;
            } else {
                return last < node.data;
            }
        }
        last = node.data;
```

```
		if (!checkBST(node.right, false)) {
			return false;
		}

		return true;
	}
}
```

// **Solution B** → min and max

```
public class QuestionB {

	public static boolean isBST(TreeNode node) {
		return isBST(node, null, null);
	}

	private static boolean isBST(TreeNode node, Integer min, Integer max) {
		if (node == null) {
			return true;
		}

		if ((min != null && node.data <= min) || (max != null && node.data > max)) {
			return false;
		}

		if (!isBST(node.left, min, node.data) || !isBST(node.right, node.data, max)) {
			return false;
		}

		return true;
	}
}
```

// **Successor** → write an algorithm to find the next node of a given node

```
public class Question {

	public static TreeNode inOrderSuccessor(TreeNode node) {
		if (node == null) {
			return null;
		}
```

```
        if (node.right != null) {
            return leftMostChild(node.right);
        } else {
            TreeNode q = node;
            TreeNode p = q.parent;
            while (p != null && p.left != q) {
                q = p;
                p = p.parent;
            }
            return p;
        }
    }

    private static TreeNode leftMostChild(TreeNode node) {
        if (node == null) {
            return null;
        }
        while (node.left != null) {
            node = node.left;
        }
        return node;
    }
}
```

// **Build Order** → we start by compiling the projects that have no incoming edges → we mark them as complete → we remove the outgoing edges → we repeat the steps (topological sort)

// **Solution A**

```
public class Question {

    public static Graph testCase() {
        Graph g = new Graph(6);

        g.add(0, 3);
        g.add(5, 1);
        g.add(1, 3);
        g.add(5, 0);
        g.add(3, 2);

        return g;
    }
```

```
        public static void compile() {
                Graph g = testCase();
                Queue<Integer> q = new Queue<>();
                boolean isProcessing = true;
                while (isProcessing) {
                        int count = 0;
                        for (int v : g.vertices()) {
                                if (g.incoming(v) == 0) {
                                        q.add(v);
                                        g.removeOutgoing(v);
                                        ++count;
                                }
                        }
                        if (count == 0) {
                                isProcessing = false;
                        }
                }
                if (q.size() == g.vertices()) {
                        System.out.println("Solution found");
                }
        }
}
```

// **Solution B** → using DFS

```
public class QuestionDFS {

  private static int[] marked;
  private static Stack<Integer> stack;

  public static Graph testCase() {
    Graph g = new Graph(6);

    g.add(0, 3);
    g.add(5, 1);
    g.add(1, 3);
    g.add(5, 0);
    g.add(3, 2);

    return g;
  }

  public static boolean dfs(Graph g, int v) {
```

```
    if (marked[v] == 1) {
      return false;
    }

    if (marked[v] == 0) {
      marked[v] = 1; // partial
      for (int w : g.adjacent(v)) {
        if (!dfs(g, w)) {
          return false;
        }
      }
      marked[v] = 2;
      stack.push(v);
    }
    return true;
  }

  public static boolean order(Graph g) {
    for (int v = 0; v < g.vertices(); v++) {
      if (marked[v] == 0) {
        if (!dfs(g, v)) {
          return false;
        }
      }
    }
    return true;
  }

  public static void main(String[] args) {
    Graph g = testCase();
    stack = new Stack<>();
    marked = new int[g.vertices()];
    if (order(g)) {
      System.out.println("Solution found");
    }
  }
```

// **First Common Ancestor** → find first common ancestor of two nodes in a binary tree

// **Solution A** → if each node has a link to its parents

```
public class QuestionA {

	public static TreeNode commonAncestor(TreeNode p, TreeNode q) {
		int delta = depth(p) - depth(q);
		TreeNode first = delta > 0 ? q : p; // shallower node
		TreeNode second = delta > 0 ? p : q; // deeper node
		second = goUpBy(second, Math.abs(delta));

		while (first != second && first != null && second != null) {
			first = first.parent;
			second = second.parent;
		}
		return first == null || second == null ? null : first;
	}

	private static TreeNode goUpBy(TreeNode node, int delta) {
		while (delta > 0 && node != null) {
			node = node.parent;
			--delta;
		}
		return node;
	}

	private static int depth(TreeNode node) {
		int depth = 0;
		while (node != null) {
			node = node.parent;
			depth++;
		}
		return depth;
	}
}
```

// **Solution B** → covers

```
public class QuestionB {

	public static TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
		if (!covers(root, p) || !covers(root, q)) {
			return null;
		} else if (covers(p, q)) {
			return p;
		} else if (covers(q, p)) {
```

```
            return q;
        }

        TreeNode sibling = getSibling(p);
        TreeNode parent = p.parent;

        while (!covers(sibling, q)) {
            sibling = getSibling(parent);
            parent = parent.parent;
        }

        return parent;
    }

    private static boolean covers(TreeNode root, TreeNode p) {
        if (root == null) {
            return false;
        }
        if (root == p) {
            return true;
        }
        return covers(root.left, p) || covers(root.right, p);
    }

    private static TreeNode getSibling(TreeNode node) {
        if (node == null || node.parent == null) {
            return null;
        }
        TreeNode parent = node.parent;
        return parent.left == node ? parent.right : parent.left;
    }
}
```

// **Solution C** → Without parent links

```
public class QuestionC {

    public static TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (!covers(root, p) || !covers(root, q)) {
            return null;
        }
        return ancestorHelper(root, p, q);
    }
```

```
    private static TreeNode ancestorHelper(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) {
            return root;
        }

        boolean pIsOnLeft = covers(root.left, p);
        boolean qIsOnLeft = covers(root.left, q);

        if (pIsOnLeft != qIsOnLeft) {
            return root;  // nodes are on different sides
        }

        TreeNode childSide = pIsOnLeft ? root.left : root.right;
        return ancestorHelper(childSide, p, q);
    }

    private static boolean covers(TreeNode root, TreeNode p) {
        if (root == null) {
            return false;
        }
        if (root == p) {
            return true;
        }
        return covers(root.left, p) || covers(root.right, p);
    }
}
```

// **BST Sequences** → binary search tree was created by reading from an array left to right; given a BST print all possible arrays

Different solution uploaded to Github (QuestionB)

// **Check Subtree** → T1 and T2 two binary trees with T1 much bigger. Write an algorithm to determine if T2 is a subtree of T1

// **Solution A** → As long as we represent the null nodes, the pre-order traversal of a tree if unique

```
public class QuestionA {

    public static boolean containsTree(TreeNode t1, TreeNode t2) {
        StringBuilder string1 = new StringBuilder();
        StringBuilder string2 = new StringBuilder();
```

```
        getOrderString(t1, string1);
        getOrderString(t2, string2);

        return string1.indexOf(string2.toString()) != -1;
    }

    private static void getOrderString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("X");
            return;
        }
        sb.append(node.data + " ");
        getOrderString(node.left, sb);
        getOrderString(node.right, sb);
    }
}
```

// O(n + m) time and space

// **Solution B** → matchTree for every node in T1 that's equal to the root of T2

```
public class QuestionB {


    public static boolean containsTree(TreeNode t1, TreeNode t2) {
        if (t2 == null) {
            return true;
        }
        return subTree(t1, t2);
    }

    public static boolean subTree(TreeNode r1, TreeNode r2) {
        if (r1 == null) {
            return false;
        } else if (r1.data == r2.data && matchTree(r1, r2)) {
            return true;
        }
        return subTree(r1.left, r2) || subTree(r1.right, r2);
    }

    public static boolean matchTree(TreeNode r1, TreeNode r2) {
        if (r1 == null && r2 == null) {
```

```
                return true;
            } else if (r1 == null || 2 == null) {
                return false;
            } else if (r1.data != r2.data) {
                return false;
            } else {
                return matchTree(r1.left, r2.left) && matchTree(r1.right, r2.right);
            }
        }
}
```