🌱 Basic Construction & Traversal Methods

1. `createNode()` – **Create a new node of the tree.**
2. `insertNode(root, value)` – **Insert a new node in binary tree.**
3. `deleteNode(root, value)` – **Delete a node from binary tree.**
4. `inorder(root)` – **Inorder traversal (Left → Root → Right).**
5. `preorder(root)` – **Preorder traversal (Root → Left → Right).**
6. `postorder(root)` – **Postorder traversal (Left → Right → Root).**
7. `levelOrder(root)` – **Level order (BFS) traversal.**
8. `height(root)` – **Find height (max depth) of tree.**
9. `size(root)` – **Count total number of nodes.**
10. `countLeaves(root)` – **Count number of leaf nodes.**

🌿 View & Printing Methods

11. `leftView(root)` – **Print left view of binary tree.**
12. `rightView(root)` – **Print right view of binary tree.**
13. `topView(root)` – **Print top view of binary tree.**
14. `bottomView(root)` – **Print bottom view of binary tree.**
15. `boundaryTraversal(root)` – **Print boundary nodes (anticlockwise).**
16. `verticalOrderTraversal(root)` – **Print nodes column-wise.**
17. `zigzagTraversal(root)` – **Print level order in zigzag pattern.**
18. `diagonalTraversal(root)` – **Print diagonal traversal.**

🌲 Property Checking Methods

19. `isBalanced(root)` – **Check if tree is height-balanced.**
20. `isFullBinaryTree(root)` – **Check if every node has 0 or 2 children.**
21. `isCompleteBinaryTree(root)` – **Check if tree is complete.**
22. `isPerfectBinaryTree(root)` – **Check if tree is perfect.**
23. `isSymmetric(root)` – **Check if tree is mirror of itself.**
24. `isSubtree(root, subRoot)` – **Check if one tree is a subtree of another.**
25. `isIdentical(root1, root2)` – **Check if two trees are identical.**
26. `childrenSumProperty(root)` – **Verify children sum property.**

✿ Path & Ancestor Methods

27. `findPath(root, node)` – **Return path from root to a given node.**
28. `rootToLeafPaths(root)` – **Print all root-to-leaf paths.**
29. `maxRootToLeafSum(root)` – **Find maximum path sum from root to leaf.**

30. `maxPathSum(root)` – **Find maximum path sum between any two nodes.**
31. `lowestCommonAncestor(root, n1, n2)` – **Find LCA of two nodes.**
32. `distanceBetweenNodes(root, n1, n2)` – **Find distance between two nodes.**
33. `printAncestors(root, target)` – **Print all ancestors of a node.**
34. `nodesAtDistanceK(root, target, K)` – **Print all nodes at distance K.**

🌱 Conversion Methods

35. `mirrorTree(root)` – **Convert tree into its mirror.**
36. `convertToSumTree(root)` – **Convert tree so each node = sum of children.**
37. `binaryTreeToDLL(root)` – **Convert binary tree to doubly linked list.**
38. `flattenToLinkedList(root)` – **Flatten tree into right-skewed linked list.**
39. `serialize(root)` – **Convert tree to array/string representation.**
40. `deserialize(data)` – **Rebuild tree from serialized data.**
41. `binaryTreeToBST(root)` – **Convert arbitrary tree into BST.**
42. `BSTToBalancedBST(root)` – **Convert skewed BST to balanced BST.**

🌾 Searching & Utility Methods

43. `findMax(root)` – **Find maximum element.**
44. `findMin(root)` – **Find minimum element.**
45. `searchNode(root, key)` – **Search a node with value = key.**
46. `sumOfNodes(root)` – **Find total sum of all nodes.**
47. `averageOfLevels(root)` – **Compute average at each level.**
48. `countFullNodes(root)` – **Count nodes with 2 children.**
49. `countHalfNodes(root)` – **Count nodes with exactly 1 child.**
50. `countNodesAtLevel(root, level)` – **Count nodes at given level.**

🌻 Advanced / Algorithmic Methods

51. `diameter(root)` – **Find diameter (longest path).**
52. `width(root)` – **Find maximum width of tree.**
53. `verticalSum(root)` – **Find sum of nodes in each vertical line.**
54. `sumOfLeafNodes(root)` – **Sum of all leaf node values.**
55. `sumOfLeftLeaves(root)` – **Sum of all left leaf nodes.**
56. `sumOfRightLeaves(root)` – **Sum of all right leaf nodes.**
57. `levelWithMaxSum(root)` – **Find level with maximum sum.**
58. `maxDepth(root)` – **Find maximum depth of tree.**
59. `minDepth(root)` – **Find minimum depth of tree.**
60. `kthAncestor(root, node, k)` – **Find K-th ancestor of given node.**

☘ Special Binary Tree Operations

61. `cloneTree(root)` – **Create deep copy of tree.**
62. `invertBinaryTree(root)` – **Invert tree (mirror flip).**
63. `sumAtLevel(root, k)` – **Sum of all nodes at level k.**
64. `deleteTree(root)` – **Delete entire tree (free memory).**
65. `constructTreeFromInPre(inorder, preorder)` – **Build tree from traversals.**
66. `constructTreeFromInPost(inorder, postorder)` – **Build tree from traversals.**
67. `printNodesWithoutSiblings(root)` – **Print nodes having no siblings.**
68. `findDeepestLeaf(root)` – **Find deepest leaf node.**
69. `sumOfNodesAtHeight(root, h)` – **Sum of all nodes at given height.**
70. `isSumTree(root)` – **Check if each node equals sum of children.**

🌳 Binary Tree with Queues/Stacks

71. `iterativeInorder(root)` – **Iterative inorder using stack.**
72. `iterativePreorder(root)` – **Iterative preorder using stack.**
73. `iterativePostorder(root)` – **Iterative postorder using two stacks.**
74. `levelOrderWithQueue(root)` – **BFS traversal using queue.**
75. `reverseLevelOrder(root)` – **Print reverse level order traversal.**

❁ Extra Useful Operations

76. `countNodesInCompleteTree(root)` – **Count nodes efficiently in complete tree.**
77. `findDeepestLeftLeaf(root)` – **Find deepest left leaf node.**
78. `findAllPathsWithSum(root, sum)` – **Find all paths with given sum.**
79. `checkCousins(root, a, b)` – **Check if two nodes are cousins.**
80. `sumReplacement(root)` – **Replace each node with sum of its subtree.**
81. `maxLevelSum(root)` – **Maximum sum among all levels.**
82. `areMirrorTrees(root1, root2)` – **Check if two trees are mirror images.**
83. `countSubtreesWithSum(root, X)` – **Count subtrees with sum X.**
84. `areCousins(root, x, y)` – **Determine if two nodes are cousins.**
85. `printVerticalOrder(root)` – **Print vertical traversal order.**
86. `getTreeHeightIterative(root)` – **Find height using queue.**
87. `printSpiralOrder(root)` – **Print spiral (zigzag) order.**
88. `sumOfLongestPath(root)` – **Sum along the longest root-to-leaf path.**
89. `nodesAtKDistanceFromLeaf(root, K)` – **Nodes at distance K from leaf.**
90. `minSwapsToConvertBTtoBST(root)` – **Find min swaps to convert to BST.**

✿ Mathematical / Structural Operations

91. `countNodesEqualToSumOfChildren(root)` – Count nodes where node = sum of children.
92. `maxSumNonAdjacentNodes(root)` – Max sum of non-adjacent nodes.
93. `maxSumLeafToLeafPath(root)` – Max path sum leaf-to-leaf.
94. `sumTreeCheck(root)` – Verify if binary tree is a sum tree.
95. `tiltOfBinaryTree(root)` – Find total tilt (difference of subtrees).
96. `sumOfKSmallestNodes(root, K)` – Sum of K smallest elements (if BST).
97. `sumOfKthLevelNodes(root, K)` – Sum at K-th level.
98. `replaceNodeWithSum(root)` – Replace each node with sum of subtree nodes.
99. `checkDuplicateSubtree(root)` – Detect duplicate subtree of size ≥ 2.
100. `serializeAndDeserializeBinaryTree(root)` – Full encode/decode of tree structure.

Would you like me to show a Java or C++ implementation template (class + key methods like insert, delete, traversal, height, etc.) next?

You said:

Binaery seach tree all methods

ChatGPT said:

Here’s a complete list of all important Binary Search Tree (BST) methods, grouped by purpose — covering everything from basic operations to advanced algorithms:

□ 1. Basic BST Operations

Operation	Description
`insert(Node* root, int key)`	Insert a new node with the given key.
`search(Node* root, int key)`	Search for a key in the BST.
`deleteNode(Node* root, int key)`	Delete a node with the given key.
`findMin(Node* root)`	Find the node with the minimum value.
`findMax(Node* root)`	Find the node with the maximum value.
`inorder(Node* root)`	Print the inorder traversal (sorted order).
`preorder(Node* root)`	Print the preorder traversal.
`postorder(Node* root)`	Print the postorder traversal.
`levelOrder(Node* root)`	Print nodes level by level using a queue.

🔍 2. Validation & Properties

Method	Description
`isBST(Node* root)`	Check if a binary tree is a valid BST.
`isBalanced(Node* root)`	Check if BST is height-balanced.
`height(Node* root)`	Find height of the BST.
`size(Node* root)`	Count total nodes in BST.
`leafCount(Node* root)`	Count total leaf nodes in BST.
`isFullBST(Node* root)`	Check if BST is full (every node has 0 or 2 children).
`isCompleteBST(Node* root)`	Check if BST is complete.
`childrenSumProperty(Node* root)`	Verify if node value equals sum of children.

🔁 3. Traversal Variations

Method	Description
`reverseInorder(Node* root)`	Print descending order of BST.
`spiralOrder(Node* root)`	Print nodes in spiral (zigzag) order.
`verticalOrder(Node* root)`	Print nodes in vertical order.
`boundaryTraversal(Node* root)`	Print boundary nodes of BST.

□ 4. Searching & Querying

Method	Description
`findKthSmallest(Node* root, int k)`	Find the k-th smallest element.
`findKthLargest(Node* root, int k)`	Find the k-th largest element.
`findFloor(Node* root, int key)`	Find floor (largest ≤ key).
`findCeil(Node* root, int key)`	Find ceil (smallest ≥ key).
`findInorderPredecessor(Node* root, Node* target)`	Find inorder predecessor of a node.
`findInorderSuccessor(Node* root, Node* target)`	Find inorder successor of a node.
`rangeSum(Node* root, int low, int high)`	Sum of all keys in a given range.
`countInRange(Node* root, int low, int high)`	Count nodes within a given range.

□ 5. Construction

Method	Description
`sortedArrayToBST(vector<int>& arr)`	Build a balanced BST from a sorted array.

Method	Description
`sortedListToBST(ListNode* head)`	**Convert a sorted linked list to a balanced BST.**
`constructBSTFromPreorder(vector<int>& preorder)`	**Construct BST from preorder traversal.**
`constructBSTFromPostorder(vector<int>& postorder)`	**Construct BST from postorder traversal.**
`constructBSTFromLevelOrder(vector<int>& level)`	**Construct BST from level order.**

---

6. Modification

Method	Description
`convertToGreaterSumTree(Node* root)`	**Convert BST to greater sum tree.**
`BSTtoDLL(Node* root)`	**Convert BST to Doubly Linked List.**
`mergeBST(Node* root1, Node* root2)`	**Merge two BSTs into one sorted list.**
`trimBST(Node* root, int low, int high)`	**Remove nodes outside given range.**
`balanceBST(Node* root)`	**Convert unbalanced BST to balanced BST.**
`mirrorBST(Node* root)`	**Create a mirror image of BST.**

---

7. Distance & Path Related

Method	Description
`LCA(Node* root, int a, int b)`	**Find Lowest Common Ancestor in BST.**
`distanceBetweenNodes(Node* root, int a, int b)`	**Find distance between two nodes.**
`maxRootToLeafSum(Node* root)`	**Find maximum sum path from root to leaf.**
`printPath(Node* root, int key)`	**Print path from root to given node.**
`allRootToLeafPaths(Node* root)`	**Print all root-to-leaf paths.**

🔢 8. Utility & Conversion

Method	Description
`cloneBST(Node* root)`	**Create a deep copy of BST.**
`serialize(Node* root)`	**Convert BST to string representation.**
`deserialize(string data)`	**Rebuild BST from serialized string.**

Method	Description
`BSTtoMinHeap(Node* root)`	Convert BST to Min Heap.
`sumOfLeafNodes(Node* root)`	Return sum of all leaf nodes.
`countNodesInRange(Node* root, int l, int r)`	Count nodes with keys in range [l, r].

⚙ 9. Advanced Operations

Method	Description
`deleteBST(Node* root)`	Delete the entire BST and free memory.
`checkPairWithSum(Node* root, int sum)`	Check if there exists a pair with given sum.
`twoSumBSTs(Node* root1, Node* root2, int target)`	Check if one element from each BST sums to target.
`BSTIterator`	Iterator class to traverse BST in sorted order.
`kthSmallestUsingIterator()`	Find k-th smallest using iterator.
`findMedianBST(Node* root)`	Find median of BST.
`countPairsWithGivenSum(Node* root1, Node* root2, int x)`	Count pairs from two BSTs whose sum is x.

Would you like me to provide C++ code implementations for all core methods (like `insert`, `search`, `delete`, `findMin`, `findMax`, `inorder`, `isBST`, `LCA`, etc.) next?
I can give a clean, reusable BST class with full implementation.

You said:

segment tree

ChatGPT said:

Here's the complete list of all important Segment Tree methods — covering construction, updates, queries, and advanced operations (for competitive programming & DSA interviews):

🌲 1. Basic Segment Tree Operations

Method	Description
`buildTree(int arr[], int tree[], int start, int end, int index)`	Build a segment tree from the given array.

Method	Description
`query(int tree[], int start, int end, int l, int r, int index)`	**Return the sum/min/max in range `[l, r]`.**
`update(int tree[], int start, int end, int indexToUpdate, int value, int index)`	**Update a value at a specific index.**
`build(vector<int>& arr)`	**Build tree using an iterative approach.**
`queryRange(int l, int r)`	**Query the range using a public function wrapper.**
`updateValue(int pos, int val)`	**Update element and rebuild relevant nodes.**

🔢 2. Types of Segment Trees

Method	Description
`buildSumTree()`	**Build tree for range sum queries.**
`buildMinTree()`	**Build tree for range minimum queries.**
`buildMaxTree()`	**Build tree for range maximum queries.**
`buildGCDTree()`	**Build tree for GCD queries.**
`buildXORTree()`	**Build tree for XOR queries.**
`buildProductTree()`	**Build tree for product of range elements (mod M).**

⚙ 3. Range Queries

Method	Description
`rangeSumQuery(int tree[], int start, int end, int l, int r, int index)`	**Get sum in range `[l, r]`.**
`rangeMinQuery(int tree[], int start, int end, int l, int r, int index)`	**Get minimum in range `[l, r]`.**
`rangeMaxQuery(int tree[], int start, int end, int l, int r, int index)`	**Get maximum in range `[l, r]`.**
`rangeGCDQuery(int tree[], int start, int end, int l, int r, int index)`	**Get GCD of elements in range.**
`rangeXORQuery(int tree[], int start, int end, int l, int r, int index)`	**Get XOR of elements in range.**

---

4. Updates

Method	Description
`pointUpdate(int tree[], int start, int end, int pos, int newVal, int index)`	**Update single element.**
`rangeUpdateAdd(int tree[], int lazy[], int start, int end, int l, int r, int val, int index)`	**Add value to all elements in range `[l, r]`.**
`rangeUpdateSet(int tree[], int lazy[], int start, int end, int l, int r, int val, int index)`	**Set all elements in range `[l, r]` to a value.**
`lazyPropagation()`	**Apply pending updates efficiently.**

💤 5. Lazy Propagation (Advanced)

Method	Description
`buildLazyTree()`	**Build segment tree with lazy propagation support.**
`updateLazy(int tree[], int lazy[], int start, int end, int l, int r, int val, int index)`	**Update range lazily.**
`queryLazy(int tree[], int lazy[], int start, int end, int l, int r, int index)`	**Query range with lazy propagation.**
`pushDown(int tree[], int lazy[], int index, int start, int end)`	**Push lazy updates to children.**
`pushUp(int tree[], int index)`	**Recalculate parent after child updates.**

□ 6. Advanced Segment Tree Variants

Method	Description
`SegmentTreeLazy`	**Class-based implementation supporting range updates and queries.**
`SegmentTree2D`	**Two-dimensional segment tree for matrix queries.**
`SegmentTreeWithMergeFunction`	**Template-based tree for custom operations (min, max, sum).**
`PersistentSegmentTree`	**Store multiple versions of segment trees for time-based queries.**
`DynamicSegmentTree`	**Handles large coordinate ranges using dynamic allocation.**
`SegmentTreeBeats`	**Advanced structure for range chmin/chmax/sum problems.**

□ 7. Query Types

Method	Description
`countInRange(int l, int r, int val)`	**Count how many elements ≤ `val` in range.**
`findKthSmallest(int l, int r, int k)`	**Find k-th smallest element in a range.**
`maxSubarraySumQuery(int l, int r)`	**Query for maximum subarray sum (Kadane-style).**
`rangeAverageQuery(int l, int r)`	**Find average in range `[l, r]`.**
`rangeFrequencyQuery(int l, int r, int x)`	**Count occurrences of `x` in `[l, r]`.**

□ 8. Utility Methods

Method	Description
`buildFromInput()`	**Build tree interactively from user input.**
`printTree()`	**Print tree elements for debugging.**
`merge(int a, int b)`	**Merge function used while building/querying tree.**
`getSize()`	**Return number of elements in array.**
`getTreeSize()`	**Return required segment tree size.**
`clearTree()`	**Reset tree and lazy arrays.**

⚡ 9. Real-world / Competitive Programming Use Cases

Problem Type	Segment Tree Usage
Range Sum Query	**Basic segment tree.**
Range Minimum/Maximum Query	**RMQ segment tree.**
Range Add and Query	**Lazy propagation tree.**
Dynamic Range XOR	**XOR segment tree.**
Counting Inversions	**Segment tree with coordinate compression.**
Maximum Subarray Sum	**Maintain sum, prefix, suffix, and total.**
Range Update with Modulo	**Lazy propagation + mod logic.**
K-th Order Statistics	**Persistent segment tree.**
Rectangle Sum	**2D segment tree.**
Number of elements greater than X	**Segment tree with ordered sets.**

□ 10. Template Class Structure (C++)

A typical reusable Segment Tree class includes:

```
class SegmentTree {
    vector<int> tree, lazy;
    int n;
public:
```

```
    SegmentTree(vector<int>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        lazy.resize(4 * n, 0);
        build(arr, 0, n - 1, 1);
    }

    void build(vector<int>& arr, int start, int end, int node);
    int query(int start, int end, int l, int r, int node);
    void update(int start, int end, int l, int r, int val, int node);
    void pushDown(int node, int start, int end);
};
```

Would you like me to give you a fully implemented C++ Segment Tree class (with build, query, update, and lazy propagation) — clean, reusable, and well-commented for contests?

You said:

graph

ChatGPT said:

Here’s the complete list of all important Graph methods — covering graph creation, traversal, shortest paths, connectivity, spanning trees, flow, and advanced algorithms used in competitive programming and interviews.

□ 1. Graph Representation

Method	Description
`addEdge(vector<int> adj[], int u, int v)`	**Add an undirected edge between `u` and `v`.**
`addDirectedEdge(vector<int> adj[], int u, int v)`	**Add a directed edge from `u` to `v`.**
`removeEdge(vector<int> adj[], int u, int v)`	**Remove an edge between `u` and `v`.**
`printGraph(vector<int> adj[], int V)`	**Print adjacency list of a graph.**
`adjMatrix(V)`	**Create an adjacency matrix representation.**
`adjList(V)`	**Create an adjacency list representation.**

🚶 2. Graph Traversal

Method	Description
`BFS(vector<int> adj[], int V, int start)`	**Breadth-First Search traversal.**
`DFS(vector<int> adj[], int V, int start)`	**Depth-First Search traversal.**
`iterativeDFS(vector<int> adj[], int start)`	**DFS using stack (iterative).**
`connectedComponents(vector<int> adj[], int V)`	**Count connected components in a graph.**
`isReachable(vector<int> adj[], int s, int d)`	**Check if node `d` is reachable from `s`.**
`countNodesAtLevel(vector<int> adj[], int V, int start, int level)`	**Count number of nodes at a given level.**

□ 3. Shortest Path Algorithms

Method	Description
`dijkstra(vector<vector<pair<int,int>>>& graph, int src)`	**Find shortest path from `src` (non-negative weights).**
`bellmanFord(vector<Edge>& edges, int V, int src)`	**Find shortest path from `src` (handles negative weights).**
`floydWarshall(vector<vector<int>>& dist, int V)`	**Find all-pairs shortest paths.**
`bfsShortestPathUnweighted(vector<int> adj[], int src)`	**Shortest path in unweighted graph using BFS.**
`spfa(vector<Edge>& edges, int V, int src)`	**Optimized Bellman-Ford using queue.**
`johnsonAlgorithm()`	**Find all-pairs shortest paths (sparse graphs).**

🕸️ 4. Topological Sorting (DAG)

Method	Description
`topologicalSortDFS(vector<int> adj[], int V)`	**Perform topological sort using DFS.**
`topologicalSortKahn(vector<int> adj[], int V)`	**Topological sort using Kahn’s algorithm (BFS + indegree).**
`checkCycleInDAG(vector<int> adj[], int V)`	**Detect cycle in directed graph using topological order.**
`allTopologicalOrders(vector<int> adj[], int V)`	**Print all possible topological orderings.**

⚙️ 5. Cycle Detection

Method	Description
`detectCycleUndirectedDFS(vector<int> adj[], int V)`	Detect cycle in undirected graph using DFS.
`detectCycleUndirectedBFS(vector<int> adj[], int V)`	Detect cycle using BFS (parent tracking).
`detectCycleDirectedDFS(vector<int> adj[], int V)`	Detect cycle in directed graph using recursion stack.
`detectCycleDirectedKahn(vector<int> adj[], int V)`	Detect cycle using Kahn's algorithm.

🌉 6. Bridges and Articulation Points

Method	Description
`findBridges(vector<int> adj[], int V)`	Find all bridges in a graph using Tarjan's algorithm.
`findArticulationPoints(vector<int> adj[], int V)`	Find all articulation points using DFS.
`criticalConnections()`	Return all critical edges in a network.

---

7. Strongly Connected Components (SCC)

Method	Description
`kosarajuSCC(vector<int> adj[], int V)`	Find all SCCs using Kosaraju's algorithm.
`tarjanSCC(vector<int> adj[], int V)`	Find SCCs using Tarjan's algorithm.
`condensedGraph()`	Build DAG from SCCs (component graph).

🌐 8. Minimum Spanning Tree (MST)

Method	Description
`kruskalMST(vector<Edge>& edges, int V)`	Find MST using Kruskal's algorithm.
`primMST(vector<vector<pair<int,int>>>& adj, int V)`	Find MST using Prim's algorithm.
`unionFindParent(int x)`	Find parent in Disjoint Set (DSU).
`unionByRank(int x, int y)`	Union two components using rank.
`totalMSTWeight()`	Return sum of weights in MST.

🌊 9. Network Flow & Matching

Method	Description
`edmondsKarp(vector<vector<int>>& capacity, int s, int t)`	**Find max flow using BFS-based Ford-Fulkerson.**
`dinicMaxFlow()`	**Efficient max flow using level graph.**
`pushRelabel()`	**Push-Relabel algorithm for max flow.**
`minCut()`	**Compute minimum cut from flow graph.**
`bipartiteMatchingDFS()`	**Find maximum bipartite matching.**
`hopcroftKarp()`	**Efficient maximum bipartite matching.**

10. Graph Coloring

Method	Description
`graphColoringBacktrack(vector<int> adj[], int m)`	**Solve m-coloring problem.**
`isBipartiteBFS(vector<int> adj[], int V)`	**Check bipartiteness using BFS.**
`isBipartiteDFS(vector<int> adj[], int V)`	**Check bipartiteness using DFS.**
`chromaticNumber()`	**Find minimum number of colors required.**

🔢 11. Special Graph Problems

Method	Description
`transitiveClosure(vector<int> adj[], int V)`	**Find transitive closure (reachability matrix).**
`cloneGraph(Node* node)`	**Clone a graph using DFS or BFS.**
`isTree(vector<int> adj[], int V)`	**Check if graph is a tree.**
`detectNegativeCycle(vector<Edge>& edges, int V)`	**Detect negative weight cycle.**
`findMotherVertex(vector<int> adj[], int V)`	**Find a mother vertex in a directed graph.**
`countPaths(vector<int> adj[], int src, int dest)`	**Count all paths from source to destination.**

Method	Description
`shortestPathDAG(vector<int> adj[], int V, int src)`	**Shortest path in a DAG.**

□ 12. Grid & Matrix Graph Problems

Method	Description
`bfsGrid(vector<vector<int>>& grid)`	**BFS traversal on a grid (4-directional).**
`dfsGrid(vector<vector<int>>& grid)`	**DFS traversal on a grid.**
`shortestPathInMaze()`	**Find shortest path in maze using BFS.**
`numberOfIslands()`	**Count number of islands (connected 1's).**
`floodFill()`	**Flood fill algorithm for image coloring.**
`minimumEffortPath()`	**Path with minimum effort difference.**

□ 13. Advanced Graph Algorithms

Method	Description
`LCAinTree()`	**Lowest Common Ancestor in a tree using Binary Lifting.**
`treeDiameter()`	**Find diameter of tree.**
`centroidDecomposition()`	**Decompose a tree into centroids.**
`heavyLightDecomposition()`	**Decompose tree for path queries.**
`eulerTour()`	**Perform Euler tour of tree.**
`binaryLifting()`	**Preprocess for fast ancestor queries.**
`cycleBasis()`	**Find fundamental cycles of a graph.**

⚡ 14. Utility & Helper Functions

Method	Description
`resetVisited()`	**Reset visited array before traversal.**
`printPath(vector<int>& parent, int dest)`	**Print path from source to destination.**
`convertToAdjMatrix()`	**Convert adjacency list to matrix.**
`convertToAdjList()`	**Convert matrix to adjacency list.**
`sortEdgesByWeight()`	**Sort all edges by weight (used in Kruskal).**
`reverseGraph()`	**Reverse all edges of a directed graph.**
`compressGraph()`	**Remove isolated nodes and compress graph.**

🌟 15. Real-world / Competitive Problems Using Graphs

Problem	Technique
Detect cycle in an undirected graph	DFS
Shortest path in road network	Dijkstra
Course scheduling	Topological Sort
Network delay time	Dijkstra / Bellman-Ford
Evaluate division queries	DFS with weights
Word ladder	BFS
Alien dictionary	Topological Sort
Cheapest flights within K stops	Modified BFS
City with smallest number of neighbors	Floyd–Warshall
Accounts merge	DSU
Redundant connection	Union-Find
Kruskal's MST	DSU + sorting edges
Counting connected provinces	DFS/BFS
Minimum height trees	Leaf trimming (BFS)
Clone graph	BFS/DFS
Find bridges	Tarjan’s Algorithm
Detect articulation points	DFS low-link values
Bipartite graph check	BFS/DFS
Number of islands	DFS/BFS on grid
Shortest path in DAG	Topological + DP

Would you like me to generate 100 one-line Graph problem statements with sample input/output format (like Codeforces/LeetCode) next?
That will give you a ready-made graph problem bank for daily practice.