DSA TOPICS

1. STACK

Stack एक डेटा स्ट्रक्चर है जिसमें आप डेटा को "Last In, First Out (LIFO)" तरीके से स्टोर करते हैं।
मतलब:

> *जो चीज़ सबसे बाद में डाली गई है, वही सबसे पहले बाहर निकलेगी।*

जैसे कि एक टिफ़िन बॉक्स का ढेर समझो:

- सबसे पहले जो बॉक्स नीचे रखा, वो सबसे बाद में निकलेगा।
- सबसे आखिरी में रखा गया बॉक्स सबसे ऊपर होता है और वही पहले निकलेगा।
  Stack की ज़रूरत क्यों है?
  Stack का इस्तेमाल कंप्यूटर साइंस में बहुत जगह होता है, जैसे:

1. फंक्शन कॉल्स – जब एक फंक्शन दूसरे फंक्शन को कॉल करता है, तो Stack की मदद से ट्रैक रखा जाता है कि कौन-कौन से फंक्शन एक्टिव हैं।
2. Undo Feature – जैसे MS Word में जब आप Ctrl + Z करते हैं, तो Stack के ज़रिए पिछले actions को स्टोर और वापस लाया जाता है।
3. Expression Evaluation – जैसे `(a + b) * c` को समझने और हल करने के लिए Stack यूज़ होता है।
4. Backtracking Algorithms – जैसे Maze Solving, Sudoku आदि में Stack इस्तेमाल होता है।
5. Browser History – आपने जो पेज आखिरी में ओपन किया, Back दबाने पर वही पहले जाता है — Stack की वजह से।

> **Note**
> Stack used in Recursion (पुनरावृत्ति) में Stack का बहुत ही अहम रोल होता है

Stack कैसे काम करता है?
Stack में दो मुख्य operations होते हैं:

- Push (डालना): Stack में नया आइटम ऊपर डालना।
- Pop (निकालना): Stack से सबसे ऊपर का आइटम निकालना।

```
class Stack {
  constructor() {
    this.stack = [];
  }

  insertElement(element) {
    return this.stack.push(element);
```

```
  }

  removeElements() {
    return this.stack.pop();
  }
  peek() {
    return this.stack[this.stack.length - 1];
  }

  isEmpty() {
    return this.stack.length === 0;
  }

  size() {
    return this.stack.length;
  }

  clearStack() {
    this.stack = [];
  }

  isAvailable(data) {
    return this.stack.includes(data);
  }

  reverseTheArray() {
    this.stack.reverse();
  }

  printStack() {
    let str = "";
    for (let i = 0; i < this.stack.length; i++) {
      str += this.stack[i] + "\n";
    }
    return str;
  }
}
```

2. Queue

Queue (क्यू) भी एक डेटा स्ट्रक्चर है, जो `First In, First Out (FIFO)` तरीके से काम करता है।

> **Note**
>
> जैसे राशन की लाइन, मंदिर की लाइन, या टिकट खिड़की की लाइन —
> जो व्यक्ति **सबसे पहले लाइन में आया**, उसी को **सबसे पहले सेवा मिलेगी**।

Queue का इस्तेमाल क्यों ज़रूरी है?

- CPU Task Scheduling – कौन-से काम पहले करने हैं, ये Queue से तय होता है।
- Printer Queue – कौन-सा डॉक्यूमेंट पहले प्रिंट होगा।
- Call Center Systems – कॉल किसे पहले मिलेगी।
- Breadth-First Search (Graph/Tree Traversal)– Queue बहुत ज़रूरी है।
- Asynchronous Tasks / Event Loop – `JS` में भी event queue होता है।

```
class Queue {
  constructor() {
    this.queue = [];
  }
  enqueue(data) {
    this.queue.push(data);
  }
  dequeue() {
    if (this.isEmpty()) {
      return "Queue is empty";
    }
    return this.queue.shift();
  }
  isEmpty() {
    return this.queue.length === 0;
  }
  peek() {
    if (this.isEmpty()) {
      return "Queue is empty";
    }
    return this.queue[0];
  }
  printQueue() {
    let str = "";
    let queueLength = this.queue.length;
    for (let i = 0; i < queueLength; i++) {
      str += this.queue[i] + "\n";
    }
    return str;
  }
}
```

Linked List

Linked List क्या है?
Linked List एक linear data structure है, जिसमें elements (nodes) एक दूसरे से link होकर जुड़े होते हैं, ना कि array की तरह सीधे-सीधे memory में पड़े होते हैं।

Linked List vs Array

Point	Array	Linked List
Memory Allocation	Continuous memory block में	Scattered memory में, हर node में next का address होता है
Access	Index से सीधे किसी भी element तक पहुँचा जा सकता है, जैसे "file खोलकर page no. 5 पढ़ना"	Direct access नहीं होता — एक-एक करके travers करना पड़ता है, जैसे "chain of clues" follow करना
Insertion/Deletion (बीच में)	महँगा होता है, क्योंकि shifting करनी पड़ती है	Fast होता है, सिर्फ pointers बदलने होते हैं
Traversal	Random access allowed है	सिर्फ़ sequential access है
Use Case Analogy	जैसे किताब जिसमें हर page numbered है, कोई भी page खोल सकते हो	जैसे investigation chain — पहले clue से शुरुआत, फिर दूसरा, फिर तीसरा...
Flexibility (Size)	Fixed/Static – पहले से तय करना पड़ता है	Dynamic – जितना चाहिए उतना जोड़ सकते हो
Efficiency	Memory waste हो सकता है अगर array में space बच जाए	Memory efficient क्योंकि हर node जितनी चाहिए उतनी memory लेता है

Linked List क्या है?
मान लीजिए आपके पास एक "चेन" है जिसमें हर कड़ी (node) दो चीज़ें रखती है:

- एक data (मतलब value)
- और दूसरी कड़ी का पता (next)
  Linked List Node

```
class Node {
  constructor(data, next = null) {
    this.data = data;
    this.next = next;
  }
}
// Linked List
class LinkedList {
  constructor() {
    this.head = null;
  }
}
//Add at Beginning
LinkedList.prototype.insertAtBeginning = function(data) {
  const newNode = new Node(data, this.head);
  this.head = newNode;
}
 //Add at End
 LinkedList.prototype.insertAtLast = function(data) {
```

```
  const newNode = new Node(data);
  if (!this.head) {
    this.head = newNode;
    return;
  }
  let last = this.head;
  while (last.next) {
    last = last.next;
  }
  last.next = newNode;
}
//Delete First Node
LinkedList.prototype.deleteFirstNode = function() {
  if (!this.head) return;
  this.head = this.head.next;
}
 // Delete Last Node
 LinkedList.prototype.removeLastNode = function() {
  if (!this.head) return;
  if (!this.head.next) {
    this.head = null;
    return;
  }
  let secondLast = this.head;
  while (secondLast.next.next) {
    secondLast = secondLast.next;
  }
  secondLast.next = null;
}
 //Delete by Value (Key)
 LinkedList.prototype.removeInBetweenElement = function(key) {
  if (!this.head) return;
  if (this.head.data === key) {
    this.head = this.head.next;
    return;
  }

  let current = this.head;
  while (current.next) {
    if (current.next.data === key) {
      current.next = current.next.next;
      return;
    }
    current = current.next;
  }
}
//Search Element
LinkedList.prototype.search = function(key) {
  let current = this.head;
  while (current) {
```

```
    if (current.data === key) return true;
    current = current.next;
  }
  return false;
}
 //Print Linked List (Traversal):
 LinkedList.prototype.printList = function() {
  let current = this.head;
  let result = [];
  while (current) {
    result.push(current.data);
    current = current.next;
  }
  console.log(result.join(" -> "));
}
// Reverse Linked List:
LinkedList.prototype.reverseList = function() {
  let prev = null;
  let current = this.head;
  let next = null;

  while (current) {
    next = current.next;
    current.next = prev;
    prev = current;
    current = next;
  }
  this.head = prev;
}
//क्योंकि Linked List में last से start नहीं कर सकते। हमें manually links पलटने होते हैं।
LinkedList.prototype.detectCycle = function() {
  let slow = this.head;
  let fast = this.head;

  while (fast !== null && fast.next !== null) {
    slow = slow.next;
    fast = fast.next.next;

    if (slow === fast) return true; // cycle detected
  }

  return false; // no cycle
}
function mergeTwoListsRecursive(l1, l2) {
  if (!l1) return l2;
  if (!l2) return l1;

  if (l1.data < l2.data) {
    l1.next = mergeTwoListsRecursive(l1.next, l2);
    return l1;
```

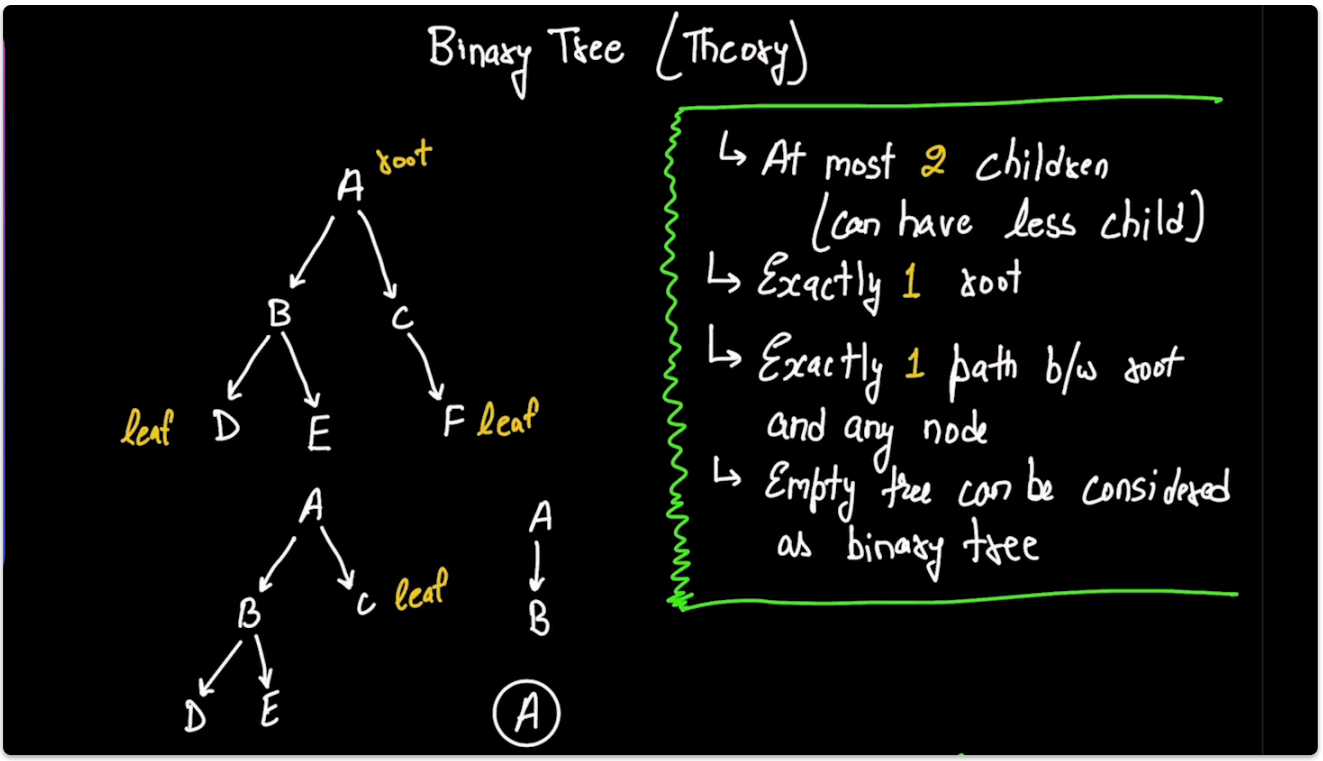
```
  } else {
    l2.next = mergeTwoListsRecursive(l1, l2.next);
    return l2;
  }
}
LinkedList.prototype.findMiddle = function() {
  let slow = this.head;
  let fast = this.head;

  while (fast !== null && fast.next !== null) {
    slow = slow.next;
    fast = fast.next.next;
  }

  return slow; // This is the middle node
}
LinkedList.prototype.isPalindrome = function() {
  if (!this.head || !this.head.next) return true;

  // Step 1: Find middle
  let slow = this.head;
  let fast = this.head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
  }

  // Step 2: Reverse second half
  let prev = null;
  while (slow) {
    let next = slow.next;
    slow.next = prev;
    prev = slow;
    slow = next;
  }

  // Step 3: Compare both halves
  let left = this.head;
  let right = prev;
  while (right) {
    if (left.data !== right.data) return false;
    left = left.next;
    right = right.next;
  }

  return true;
}
function addTwoNumbers(l1, l2) {
  let dummy = new Node(0);
  let current = dummy;
```

```
  let carry = 0;

  while (l1 || l2 || carry) {
    const val1 = l1 ? l1.data : 0;
    const val2 = l2 ? l2.data : 0;
    const sum = val1 + val2 + carry;

    carry = Math.floor(sum / 10);
    current.next = new Node(sum % 10);
    current = current.next;

    if (l1) l1 = l1.next;
    if (l2) l2 = l2.next;
  }

  return dummy.next;
}
function getIntersectionNode(headA, headB) {
  if (!headA || !headB) return null;

  let a = headA;
  let b = headB;

  while (a !== b) {
    a = a ? a.next : headB;
    b = b ? b.next : headA;
  }

  return a;
}
```

कैसे काम करता है?

- नया node बनाओ।
- उसका next पुराने head को बना दो।
- नए node को head बना दो।

Binary Tree And search Tree



Binary Search Tree (BST) के नियम :

- Root Node ज़रूरी है
  - हर Binary Tree की शुरुआत एक Root Node से होती है।
  - यह Tree का सबसे ऊपर का node होता है।
- हर Node में तीन values होती हैं:
  - `key` (मतलब value/data)
  - `left` (left child)
  - `right` (right child)
- BST की property (Very Important Rule):
  - अगर नया value < current node का value हो → तो left side जाएँ।
  - अगर नया value > current node का value हो → तो right side जाएँ।
  - यह process तब तक repeat होता है जब तक सही जगह ना मिल जाए।
- Tree को Balance करना ज़रूरी नहीं है
  - Tree एक ही साइड में grow कर सकता है (जैसे सब values increasing हों तो सिर्फ right में grow होगा)।
  - इसे Unbalanced BST कहते हैं।

BST में Node कैसा होता है?

```
class BstNode {
  constructor(key) {
    this.key = key;
    this.left = null;
    this.right = null;
  }
}
class BinarySearchTree {
```

```
  constructor() {
    this.root = null; // initially tree empty है
  }
   insert(key) {
    const newNode = new BstNode(key);
    if (!this.root) {
      this.root = newNode; // अगर root null है तो वही root बन जाएगा
    } else {
      this.insertNode(this.root, newNode); // नहीं तो recursive insert
    }
  }
    insertNode(node, newNode) {
    // अगर new key छोटा है तो left में जाएँ
    if (newNode.key < node.key) {
      if (node.left === null) {
        node.left = newNode;
      } else {
        this.insertNode(node.left, newNode); // और deep जाकर insert करें
      }
    } else {
      // अगर new key बड़ा है तो right में जाएँ
      if (node.right === null) {
        node.right = newNode;
      } else {
        this.insertNode(node.right, newNode);
      }
    }
  }
}
```

Binary Search Tree (BST) से Node Delete कैसे करते हैं?

मान लीजिए आपके पास एक BST है, और आपको उसमें से कोई एक node हटाना है। अब ये तीन स्थितियाँ बन सकती हैं

1. Node मिला ही नहीं (Not Found):

- अगर tree खाली है या आप जिस key को ढूंढ रहे हैं वो कहीं नहीं है —
  तो हम **null return कर देते हैं**, मतलब कुछ delete नहीं हुआ।

2. Node मिल गया (Found) – अब 3 cases आते हैं:

A. Leaf Node (जिसके कोई बच्चे नहीं हैं):

- ऐसे node को बस सीधा **हटा दो**, क्यूंकि इससे कोई जुड़ा नहीं है।

> *याद रखें: "पत्ता (Leaf) काट दो!"*

- B. One Child (केवल एक बच्चा है – या left या right):
  अब आप node को delete कर दो और **उसके बच्चे को ऊपर भेज दो**,
  जिससे जो उससे जुड़ा था वो अब सीधे उसके बच्चे से जुड़ जाए।

> *याद रखें: "बायपास कर दो!"*

C. Two Children (दोनों बच्चे हैं):
ये थोड़ा tricky है — आप क्या करते हो?

1. उसके right subtree में सबसे छोटा node (inorder successor) ढूंढो।
2. उस छोटे वाले का key copy कर दो current node में।
3. फिर उस छोटे वाले node को delete करो, क्यूंकि उसका काम हो गया।

```
class BstNode {
    constructor(key) {
        this.key = key;
        this.left = null;
        this.right = null;
    }
}
class BinarySearchTree {
    constructor() {
        this.root = null;
    }
    delete(key) {
        this.root = this.deleteNode(this.root, key);
    }
    deleteNode(node, key) {
        if (node === null) {
            return null; // Node not found, nothing to delete
        }
        if (key < node.key) {
            node.left = this.deleteNode(node.left, key); // Traverse left
        } else if (key > node.key) {
            node.right = this.deleteNode(node.right, key); // Traverse
right
        } else {
            // Node with the key found
            if (node.left === null && node.right === null) {
                return null; // Leaf node
            } else if (node.left === null) {
                return node.right; // Only right child
            } else if (node.right === null) {
                return node.left; // Only left child
            } else {
                // Node with two children: Get the inorder successor
(smallest                       in the right subtree)
```

```
                let tempNode = this.findMinNode(node.right);
                node.key = tempNode.key; // Copy the inorder successor's
value                           to this node
                node.right = this.deleteNode(node.right, tempNode.key); //
Delete the inorder successor
            }
        }
        return node; // Return the (potentially unchanged) node pointer
    }

    findMinNode(node) {
        while (node.left !== null) {
            node = node.left; // Traverse to the leftmost node
        }
        return node; // Return the node with the smallest key
    }
}
```

Different methods

```
class BstNode {
  constructor(key) {
    this.key = key;
    this.left = null;
    this.right = null;
  }
}

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


  search(value, node = this.root) {
    if (!node) return false;
    if (value === node.key) return true;
    return value < node.key
      ? this.search(value, node.left)
      : this.search(value, node.right);
  }

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

  findMax(node = this.root) {
```

```
    if (!node) return null;
    while (node.right) node = node.right;
    return node.key;
  }

  inOrderTraversal(node = this.root, result = []) {
    if (node) {
      this.inOrderTraversal(node.left, result);
      result.push(node.key);
      this.inOrderTraversal(node.right, result);
    }
    return result;
  }

  preOrderTraversal(node = this.root, result = []) {
    if (node) {
      result.push(node.key);
      this.preOrderTraversal(node.left, result);
      this.preOrderTraversal(node.right, result);
    }
    return result;
  }

  postOrderTraversal(node = this.root, result = []) {
    if (node) {
      this.postOrderTraversal(node.left, result);
      this.postOrderTraversal(node.right, result);
      result.push(node.key);
    }
    return result;
  }

  height(node = this.root) {
    if (!node) return -1;
    return 1 + Math.max(this.height(node.left), this.height(node.right));
  }

  isBalanced(node = this.root) {
    if (!node) return true;
    const lh = this.height(node.left);
    const rh = this.height(node.right);
    return (
      Math.abs(lh - rh) <= 1 &&
      this.isBalanced(node.left) &&
      this.isBalanced(node.right)
    );
  }

  findLCA(node, n1, n2) {
    if (!node) return null;
```

```
    if (n1 < node.key && n2 < node.key)
      return this.findLCA(node.left, n1, n2);
    if (n1 > node.key && n2 > node.key)
      return this.findLCA(node.right, n1, n2);
    return node.key;
  }

  isValidBST(node = this.root, min = -Infinity, max = Infinity) {
    if (!node) return true;
    if (node.key <= min || node.key >= max) return false;
    return (
      this.isValidBST(node.left, min, node.key) &&
      this.isValidBST(node.right, node.key, max)
    );
  }

  kthSmallest(k) {
    let count = 0, result = null;
    const inorder = (node) => {
      if (!node || result !== null) return;
      inorder(node.left);
      count++;
      if (count === k) result = node.key;
      inorder(node.right);
    };
    inorder(this.root);
    return result;
  }

  kthLargest(k) {
    let count = 0, result = null;
    const reverseInorder = (node) => {
      if (!node || result !== null) return;
      reverseInorder(node.right);
      count++;
      if (count === k) result = node.key;
      reverseInorder(node.left);
    };
    reverseInorder(this.root);
    return result;
  }

  isSymmetric() {
    const isMirror = (t1, t2) => {
      if (!t1 && !t2) return true;
      if (!t1 || !t2) return false;
      return (
        t1.key === t2.key &&
        isMirror(t1.left, t2.right) &&
        isMirror(t1.right, t2.left)
```

```
      );
    };
    return isMirror(this.root, this.root);
  }
}
```

5. `inOrderTraversal()`

काम: Tree को sorted order में print करता है।
लॉजिक:

- पहले left `subtree` → फिर खुद का key → फिर right `subtree`।
  Output हमेशा sorted होगा।

6. `preOrderTraversal()`

काम: Tree को पहले root फिर left और फिर right के हिसाब से print करता है।
लॉजिक:

- पहले खुद को print करो → फिर left → फिर right।

7. `postOrderTraversal()`

काम: पहले पूरे subtree को process करता है फिर खुद को।
लॉजिक:

- पहले left → फिर right → फिर खुद को।

8. `height()`

काम: Tree की ऊँचाई बताता है (root से नीचे की सबसे लंबी depth)।
लॉजिक:

- Height = 1 + max(left की height, right की height)।

9. `isBalanced()`

काम: चेक करता है कि हर node का left और right subtree की ऊँचाई में ज़्यादा अंतर तो नहीं है।
लॉजिक:

- हर node पे difference ≤ 1 होना चाहिए।

10. `findLCA(node1, node2)`

काम: दो nodes का lowest common ancestor (LCA) ढूंढता है।
लॉजिक:

- अगर दोनों node values root से छोटी हैं → left में जाओ।

- अगर दोनों बड़ी हैं → right में जाओ।
- अगर एक left और एक right में है → यही common ancestor है।

11. isValidBST()

काम: चेक करता है कि tree BST की properties follow कर रहा है या नहीं।
लॉजिक:

- हर node की value, उसके left subtree से बड़ी और right subtree से छोटी होनी चाहिए।

12. `kthSmallest(k)`

काम: kth smallest element देता है।
लॉजिक:

- In-order traversal (sorted order) करो।
- k बार count करते जाओ → k पर पहुंचते ही return कर दो।

13. `kthLargest(k)`

काम: kth सबसे बड़ा element देता है।
लॉजिक:

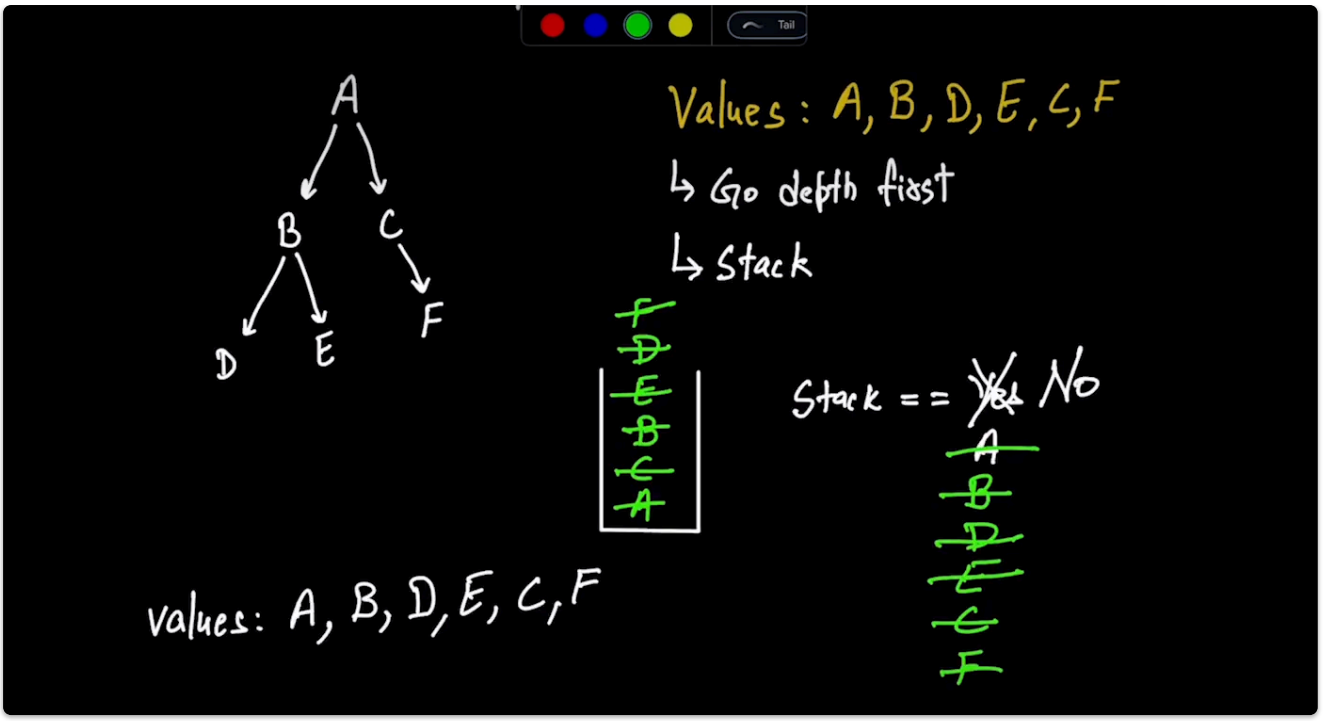
- Reverse in-order traversal (right → root → left) करो।
- count बढ़ाते जाओ → k पर पहुंचो → return करो।

14. `isSymmetric()`

काम: Tree mirror image है या नहीं ये बताता है
लॉजिक:

- Left और Right subtree एक-दूसरे के mirror होने चाहिए।
- Left.left = Right.right && Left.right = Right.left होना चाहिए हर step पर।

Depth First Search Approach Algorithm(DFS)



DFS में क्या चाहिए:

- एक root node – यही से tree की यात्रा शुरू होगी।
- एक stack – ताकि हम हर node के visit करने का track रख सकें (LIFO concept – जो पहले गया वो बाद में निकलेगा)।
- एक values array – जिसमे हम visit किए गए node को record रखेंगे।

`DFS` का Story Flow (Iterative Approach):

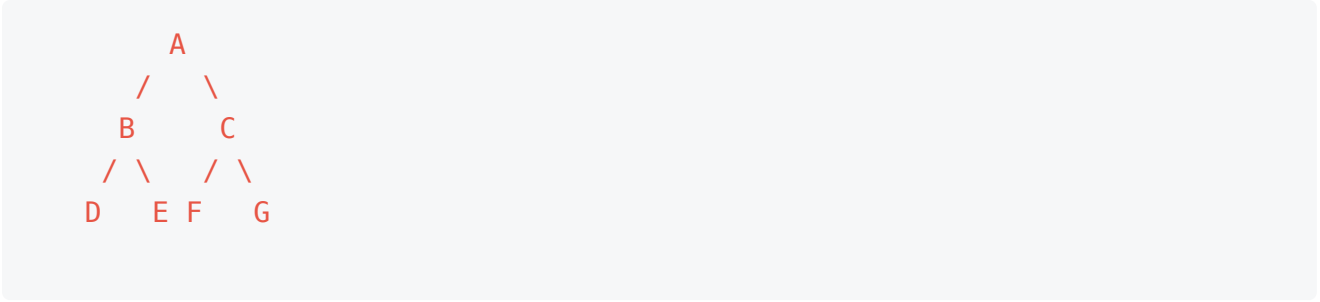
- सबसे पहले root node को stack में डाल दो और values array खाली रखो।
- जब तक stack में कुछ है:
  - stack से top element निकालो (`pop()` करो)
  - उसका key values array में डालो
  - अगर उस node के right child है → उसे stack में डालो
  - फिर अगर left child है → उसे भी डालो
    क्यों? क्योंकि stack LIFO है – हम चाहते हैं कि left पहले निकले
- जब तक सब node explore ना हो जाए, यही process repeat होता रहेगा।
- अंत में values array पूरा DFS order में मिलेगा।

Recursive DFS (Pre-order Traversal)

```
const recursiveDepthFirstTraversal = (root) => {
    if (!root) return [];
    const leftValues = recursiveDepthFirstTraversal(root.left);
    const rightValues = recursiveDepthFirstTraversal(root.right);
    return [root.key, ...leftValues, ...rightValues];
};
```

Level Order Traversal (BFS)

क्या होता है Level Order Traversal?
Level Order Traversal का मतलब है – हर level के सारे nodes को left से right तक एक-एक करके visit करना, फिर अगले level पर जाना।



Traversal Output होगा: `A → B → C → D → E → F → G`

क्या चाहिए?

1. Queue – क्योंकि हमें पहले आया node पहले process करना है → FIFO (First In First Out)
2. Values Array – visit किए गए nodes को record करने के लिए

Step by Step Logic (In Hindi):

1. अगर root ही नहीं है, तो empty array return करो।
2. एक queue बनाओ और उसमें root node डालो।
3. जब तक queue में कोई element हो:
   - `shift()` से queue का पहला element निकालो (यानि front of the line)
   - उसकी key values array में डालो
   - अगर उसका left child है, तो उसे queue में डालो
   - अगर उसका right child है, तो उसे भी queue में डालो
4. इस तरह से हम एक-एक करके हर level के सारे nodes process करते जाते हैं।

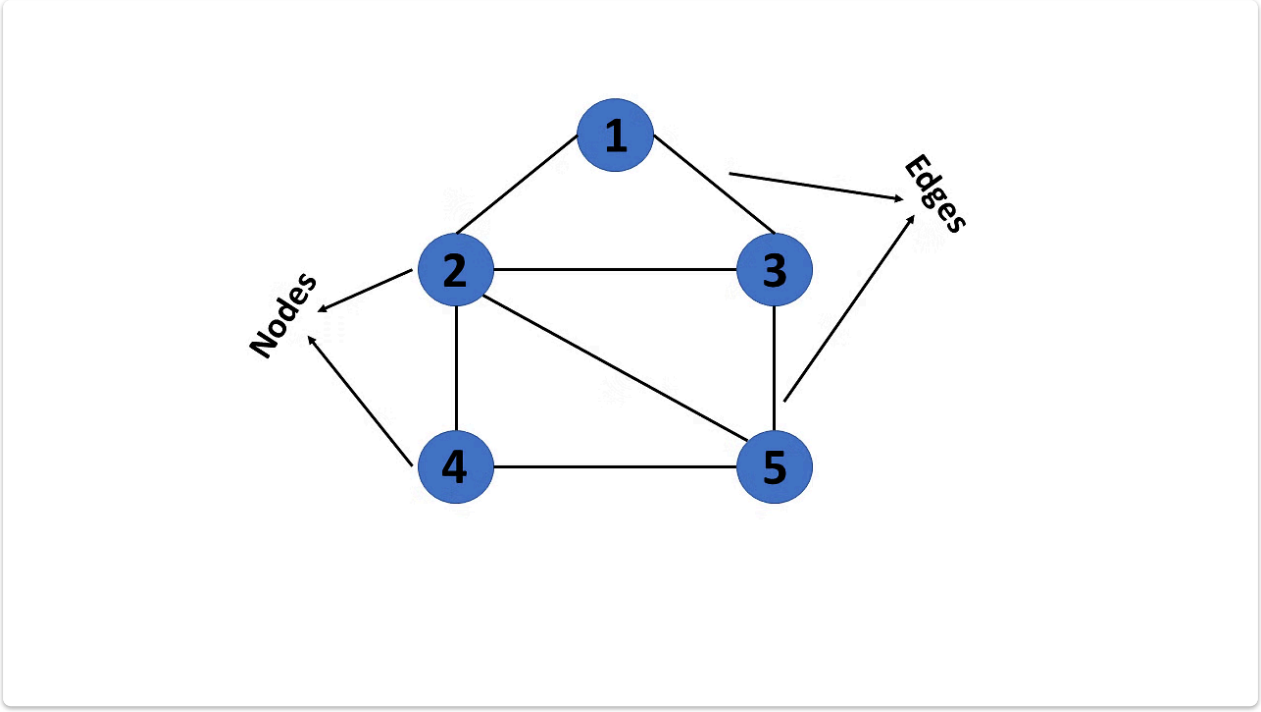
```
const levelOrderTraversal = (root) => {
    if (!root) return [];
    const queue = [root];
    const values = [];

    while (queue.length > 0) {
        const node = queue.shift();
        values.push(node.key);

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

    return values;
```

```
};
```

Graphs

Graph (ग्राफ) एक ऐसा डेटा स्ट्रक्चर होता है जो **nodes (जिसे vertex/vertices भी कहते हैं)** और **edges (connections या रास्ते)** से बना होता है।

- **Node (Vertex):** एक पॉइंट या जगह (जैसे "A", "B", "C")
- **Edge:** दो nodes के बीच का कनेक्शन या रास्ता



2. ग्राफ के प्रकार (Types of Graphs)

a. `Undirected Graph` (अनिर्दिष्ट ग्राफ)

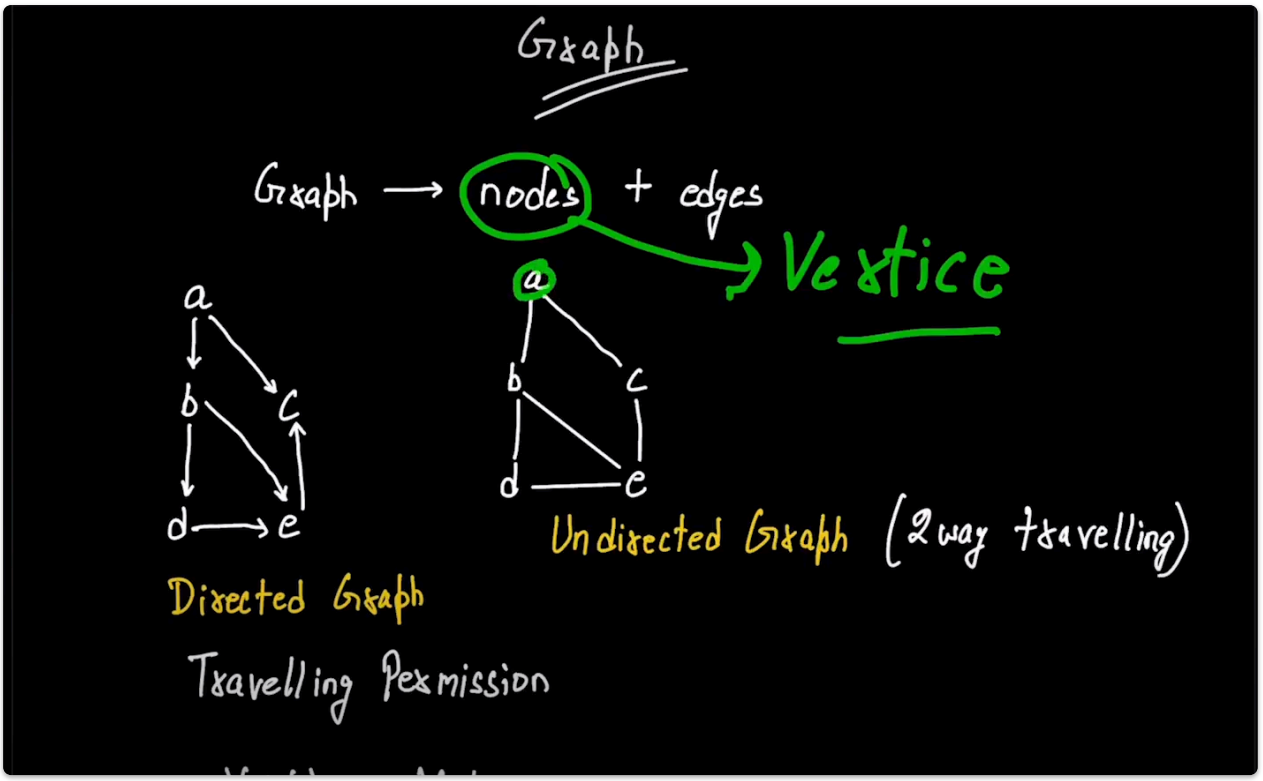
- इसमें कनेक्शन दोनों दिशाओं में होता है।
- अगर A → B है, तो इसका मतलब B → A भी हो सकता है।
- यहाँ दोनो directions में travelling की permission होती है।
- अगर A और B के बीच connection (edge) है, तो
  Use case: दोस्ती (Friendship) — अगर A, B का दोस्त है, तो B भी A का दोस्त है।
  b. `Directed Graph` (निर्दिष्ट ग्राफ / Digraph)
- इसमें edges एक दिशा में होते हैं (one-way).
- अगर A → B है, तो B → A जरूरी नहीं है।

1. `Cyclic` Graph:

- Contains one or more **cycles**, which are paths that start and end at the same vertex.

1. `Acyclic` Graph:

- No cycles are present in the graph.
- A common example is a Tree, which is a special type of acyclic graph.



3.Graph क्यों और कहां काम आता है?

- Social network (Facebook: friend suggestion)
- Navigation (Google Maps)
- Recommendation engine
- Dependency trees (coding, tasks, etc.)

Adjacency List क्या होता है?

> *Adjacency List एक ऐसा तरीका है जिसमें हर node (vertex) के साथ उसके connected nodes (neighbors) को लिस्ट किया जाता है।*

यह structure हमें ये बताता है:

- कौन-कौन से node connected हैं?
- कहाँ से कहाँ flow (या arrow)** जा सकता है — यानी travelling permission.

```
{
  A: ["B", "C"],
  B: ["A", "D"],
  C: ["A"],
  D: ["B"]
}
```

यहाँ से पता चलता है

- A से B और C जा सकते हैं
- B से A और D जा सकते हैं (दोनों तरफ allowed है क्योंकि undirected है).

> ⚠ **Warning**
> **DFS (Depth First Search)** और **BFS (Breadth First Search)** दोनों ही **Graph के ज़्यादातर problems को solve करने की core techniques** हैं

DFS (Depth First Search) — गहराई में जाता है

- Stack (या recursion) का use करता है
- एक node से शुरुआत करके जितना possible हो अंदर जाता है, फिर backtrack करता है

DFS का use:

- Cycle detection
- Tree/graph traversal
- Topological sorting (DAG में)
- Connected components
- Path exist करता है या नहीं चेक करना

BFS (Breadth First Search) — लेयर-बाय-लेयर जाता है

- Queue का use करता है
- पहले एक level explore करता है, फिर next
  BFS का use:
- Shortest path (Unweighted graph में)
- Minimum steps/levels
- Level-order traversal
- Friend suggestion / Nearby nodes

तुलना (DFS vs BFS)

Feature	DFS	BFS
Strategy	गहराई में जाता है	चौड़ाई में लेयर-बाय-लेयर जाता है
Use Cases	Cycle, backtracking, components	Shortest path, level-based
Data Structure	Stack / Recursion	Queue
Space Complexity	O(h) (h = depth)	O(w) (w = width/level size)

Depth-First Search (DFS):

we only care that is node visited

```
function depthFSIterative(graph, start) {
  const stack = [start]; // स्टैक में शुरुआत का नोड डालो
  const visited = new Set(); // विज़िटेड नोड्स को ट्रैक करने के लिए Set बनाओ
  while (stack.length > 0) { // जब तक स्टैक खाली नहीं हो जाता
    const node = stack.pop(); // स्टैक से टॉप नोड निकालो (LIFO)

    if (!visited.has(node)) { // अगर ये नोड पहले विज़िट नहीं हुआ
      visited.add(node); // विज़िटेड में इस नोड को ऐड कर दो

      for (const neighbour of graph[node]) { // इसके सभी नेबर्स को देखो
        if (!visited.has(neighbour)) { // जो विज़िटेड नहीं हैं
          stack.push(neighbour); // उन्हें स्टैक में डाल दो
        }
      }
    }
  }
  return Array.from(visited); // विज़िट किए गए नोड्स को ऐरे में बदलकर लौटाओ
};
```

Recursive DFS (with visited tracking)

```
function depthFSRecursive(graph, node, visited = new Set(), result = []) {
  visited.add(node); // नोड को विज़िट कर लिया
  result.push(node); // रिज़ल्ट में डाल दो

  for (const neighbour of graph[node]) {
    if (!visited.has(neighbour)) {
      depthFSRecursive(graph, neighbour, visited, result); // recursion से
बाकी को विज़िट करो
    }
  }

  return result;
}
```

DFS without visited set (may revisit nodes — not ideal for cycles)

```
function depthFSAllowRevisit(graph, start) {
  const stack = [start];
  const result = [];

  while (stack.length > 0) {
    const node = stack.pop();
    result.push(node); // रिकॉर्ड कर लो, चाहे वो पहले आया हो या नहीं

    for (const neighbour of graph[node]) {
```

```
      stack.push(neighbour); // हर बार नेबर्स को स्टैक में डाल दो
    }
  }

  return result;
}
```

BFS (Breadth First Search) क्या है?

- Graph traversal की एक technique है जहाँ हम सबसे पहले current node के *सभी आस-पास (neighbours)* को विज़िट करते हैं और फिर उनके neighbours को।
- मान लो तुम किसी स्टेशन पर खड़े हो — पहले वहाँ के सभी नजदीकी स्टेशन पर जाओ, फिर उन स्टेशन से आगे के और... और ऐसे ही आगे बढ़ते जाओ।
- हम एक Queue (FIFO) यूज़ करते हैं ताकि पहले आए हुए nodes को पहले process करें।

```
function breadthFirstSearch(graph, start) {
  const queue = [start];          // 1. Start node को queue में डालो
  const visited = new Set();      // 2. Track रखने के लिए Set
  const result = [];              // 3. Traverse का order
  visited.add(start);
  while (queue.length > 0) {
    const node = queue.shift();   // 4. FIFO: सबसे पहले आए node को निकालो
    result.push(node);

    for (const neighbour of graph[node]) {
      if (!visited.has(neighbour)) {
        visited.add(neighbour);    // 5. उसे visited में डाल दो
        queue.push(neighbour);     // 6. और queue में डाल दो future
traversal के लिए
      }
    }
  }
  return result;
}
```

search Node

To determine whether there is a directed path from a start node to a dest node in a Directed Acyclic Graph (DAG)

JavaScript Code (DFS using Stack)

```
function hasPath(graph, start, dest) {
  const stack = [start];
  const visited = new Set();
```

```
  while (stack.length > 0) {
    const node = stack.pop();

    if (node === dest) return true;
    if (visited.has(node)) continue;

    visited.add(node);

    for (let neighbor of graph[node]) {
      stack.push(neighbor);
    }
  }

  return false;
}
```

Code (BFS using Queue)

```
function hasPath(graph, start, dest) {
  const queue = [start];
  const visited = new Set();
  while (queue.length > 0) {
    const node = queue.shift();
    if (node === dest) return true;
    if (visited.has(node)) continue;
    visited.add(node);
    for (let neighbor of graph[node]) {
      queue.push(neighbor);
    }
  }
  return false;
}
```

JavaScript Code (DFS)

```
function hasPath(graph, start, dest) {
  const visited = new Set();

  function dfs(node) {
    if (node === dest) return true;
    if (visited.has(node)) return false;

    visited.add(node);

    for (let neighbor of graph[node]) {
```

```
      if (dfs(neighbor)) return true;
    }

    return false;
  }

  return dfs(start);
}
```

recursive solution

```
function hasPath(graph, start, dest, visited = new Set()) {
  if (start === dest) return true;
  if (visited.has(start)) return false;
  visited.add(start);
  for (let neighbor of graph[start]) {
    if (hasPath(graph, neighbor, dest, visited)) {
      return true;
    }
  }
  return false;
}
```