1) Write an algorithm for depth first traversal of a graph

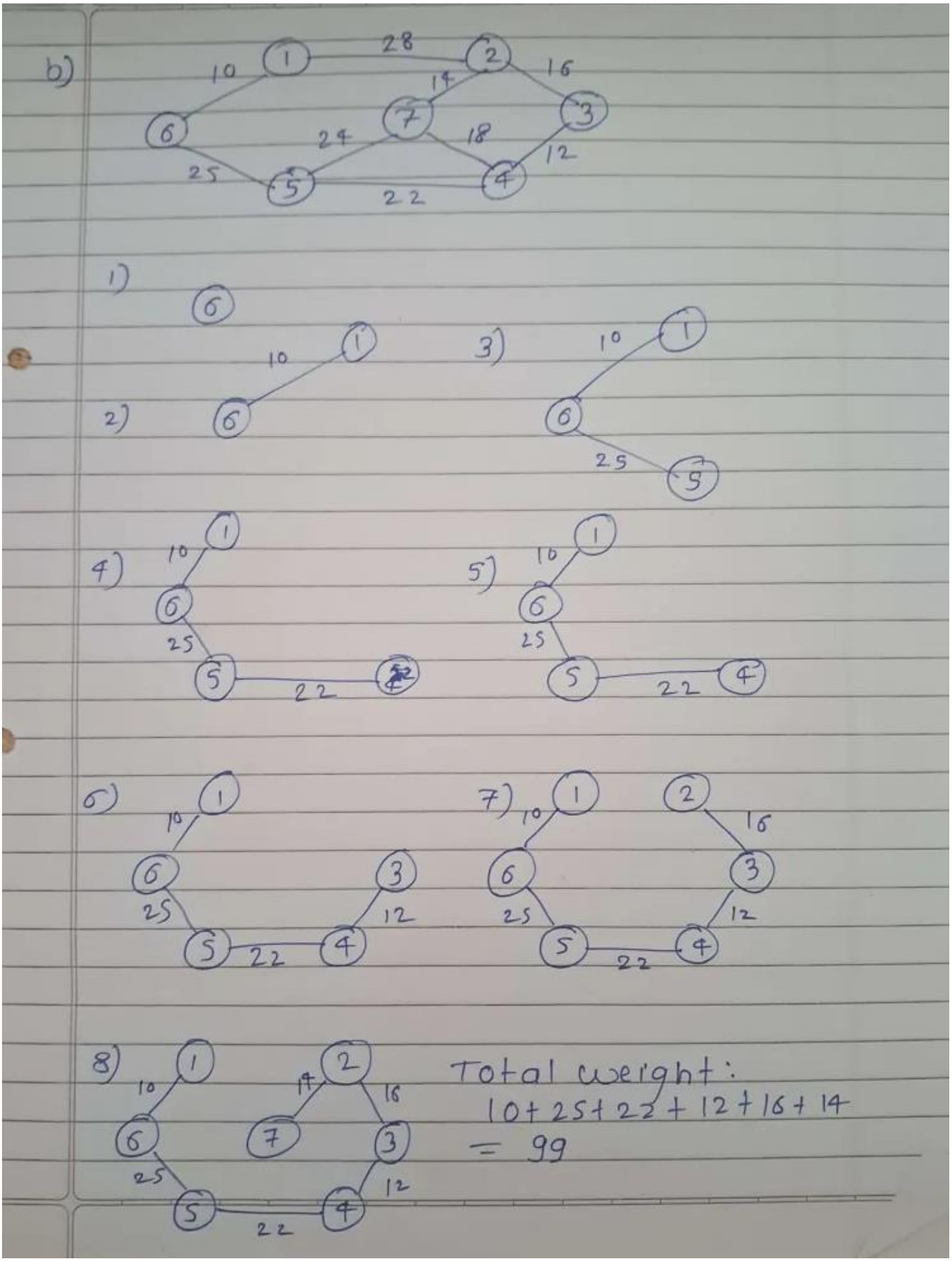
DFS - Algorithm

- **Step 1** - Define a Stack of size number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3** - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4** - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5** - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.

28

- **Step 7** - When stack becomes Empty, then produce final

2) Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm staring from vertex 6.



3) What is topological sorting? Find topological sorting of given graph.

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.

Topological Sorting for a graph is not possible if the graph is not a DAG.

Example 3.10.1 *Sort the digraph for topological sort using source removal algorithm.*

SPPU : May-14, Marks 3

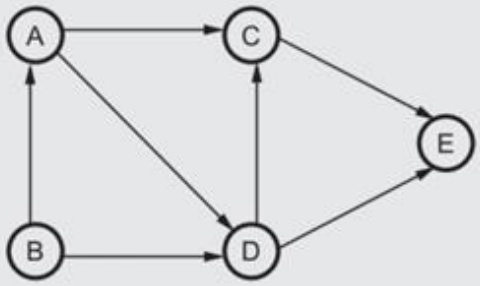


Fig. 3.10.1

Solution : We will follow following steps to obtain topologically sorted list.

Choose vertex B, because it has no incoming edge, delete it along with its adjacent edges.

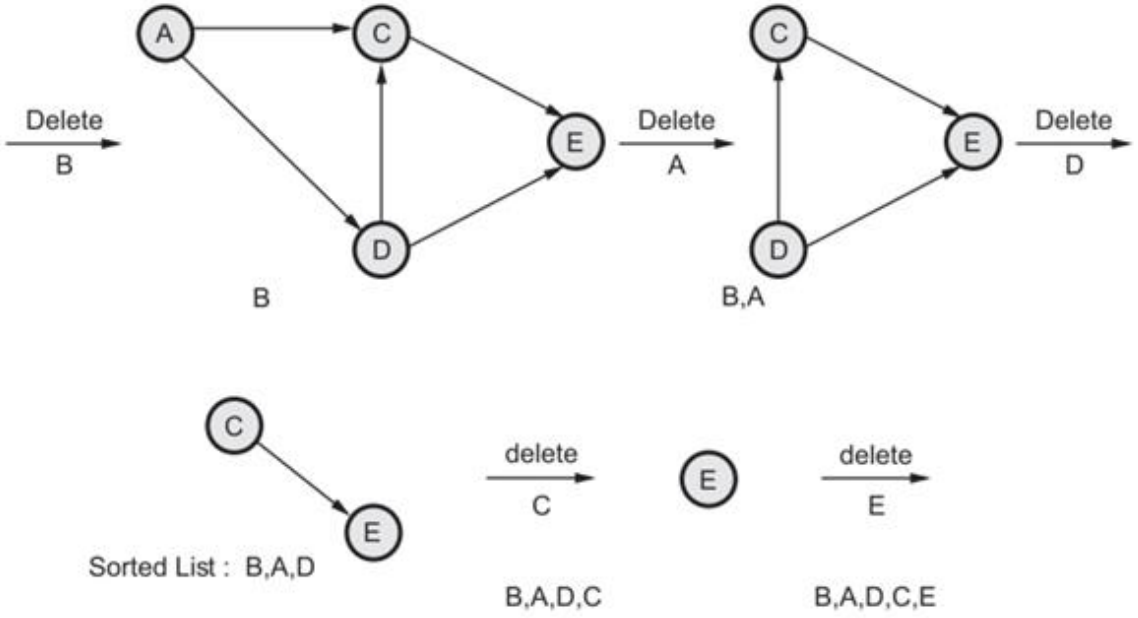


Fig. 3.10.2

Hence the list after topological sorting will be B, A, D, C, E.

4) Write an algorithm for breadth first traversal of a graph.

Step 1 - Define a **Queue** of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

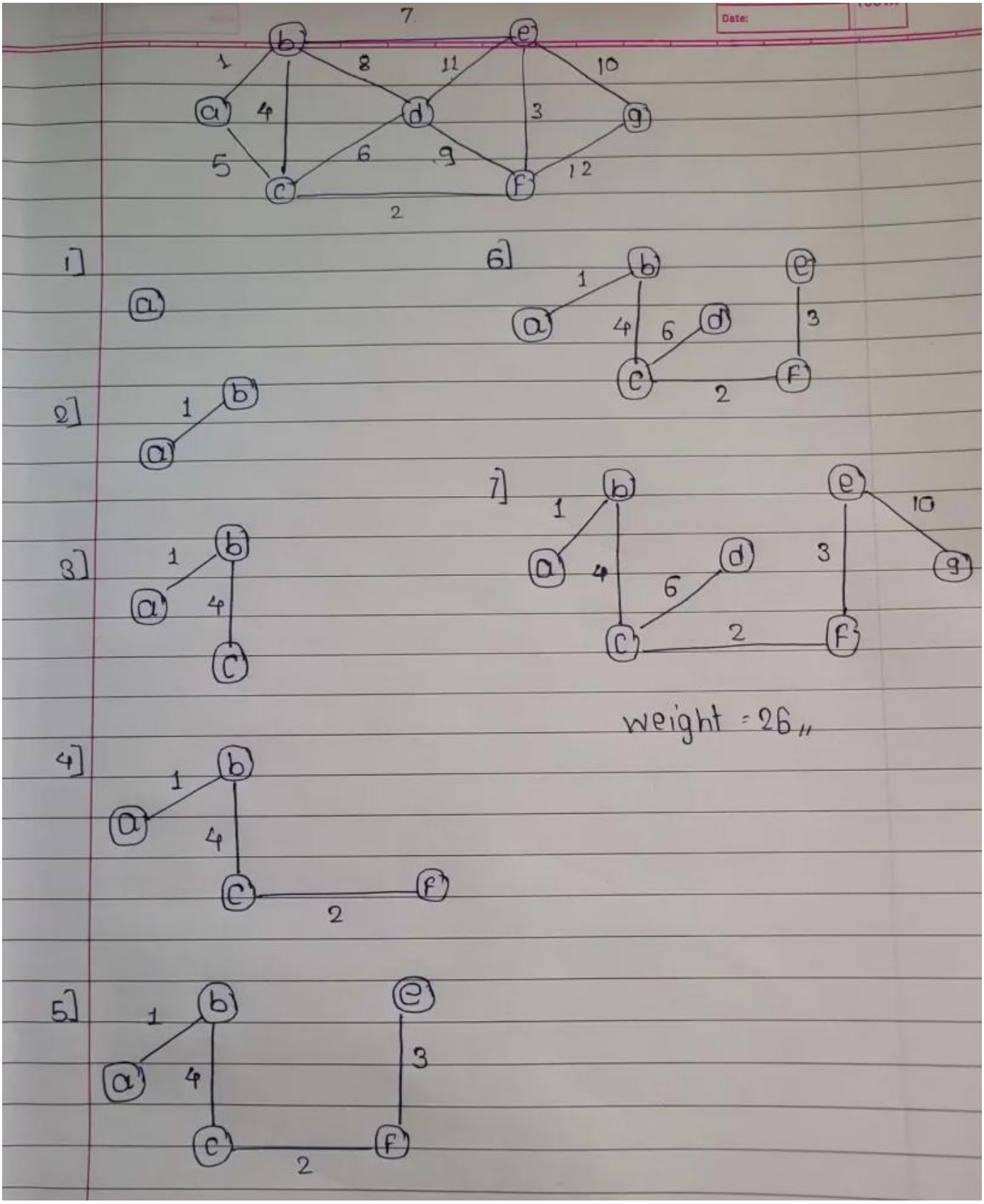
Step 3 - Visit all the **non-visited adjacent vertices** of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then **delete that vertex**.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

5) Using Prim's Algorithm, find the cost of minimum spanning tree (MST) of the given graph starting from vertex 'a'



6) Define the following terms : i) Complete Graph ii) Connected Graph iii) Subgraph

3.1.4 A Complete Graph

An undirected graph, in which every vertex has an edge to all other vertices is called a complete graph.

A complete graph with N vertices has $\frac{N(N-1)}{2}$ edges.

Example of a complete graph is shown in Fig. 3.1.4.

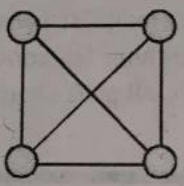


Fig. 3.1.4 : A complete graph

In a complete graph, these is an edge between every pair of vertices.

Number of edges (a pair of vertices) in a graph with n vertices is equal to combination of n elements, taking 2 at a time :

3.1.9 Connected Graph

A graph is said to be connected if there exists a path between every pair of vertices V_i and V_j.

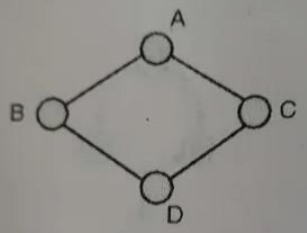
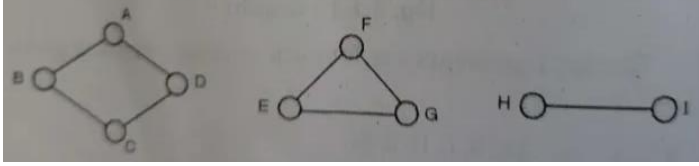
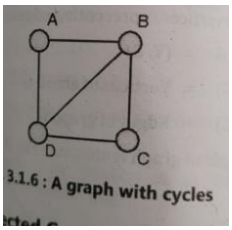


Fig. 3.1.7 : A connected graph





3.1.6 : A graph with cycles

3.1.10 Subgraph

A subgraph of G is a graph G_1 such that $V(G_1)$ is a subset of $V(G)$ and $E(G_1)$ is a subset of $E(G)$.

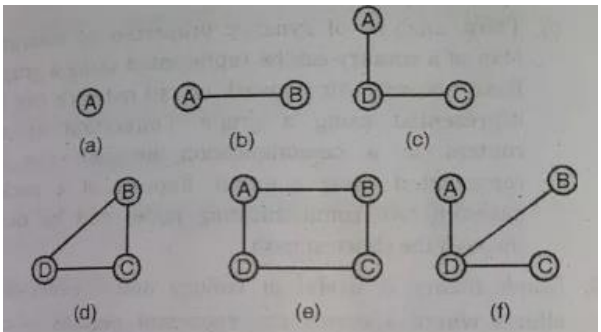


Fig. 3.1.9 : Some subgraphs of the graph of Fig. 3.1.6

7) Write Floyd Warshall Algorithm

1. Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

2. Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

3. Similarly, A^2 is created using A^1. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in

Calculate the distance from the source vertex to destination vertex through this vertex 2

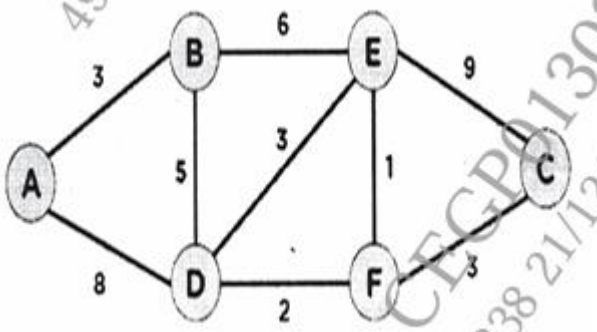
4. Similarly, A^3 and A^4 is also created.

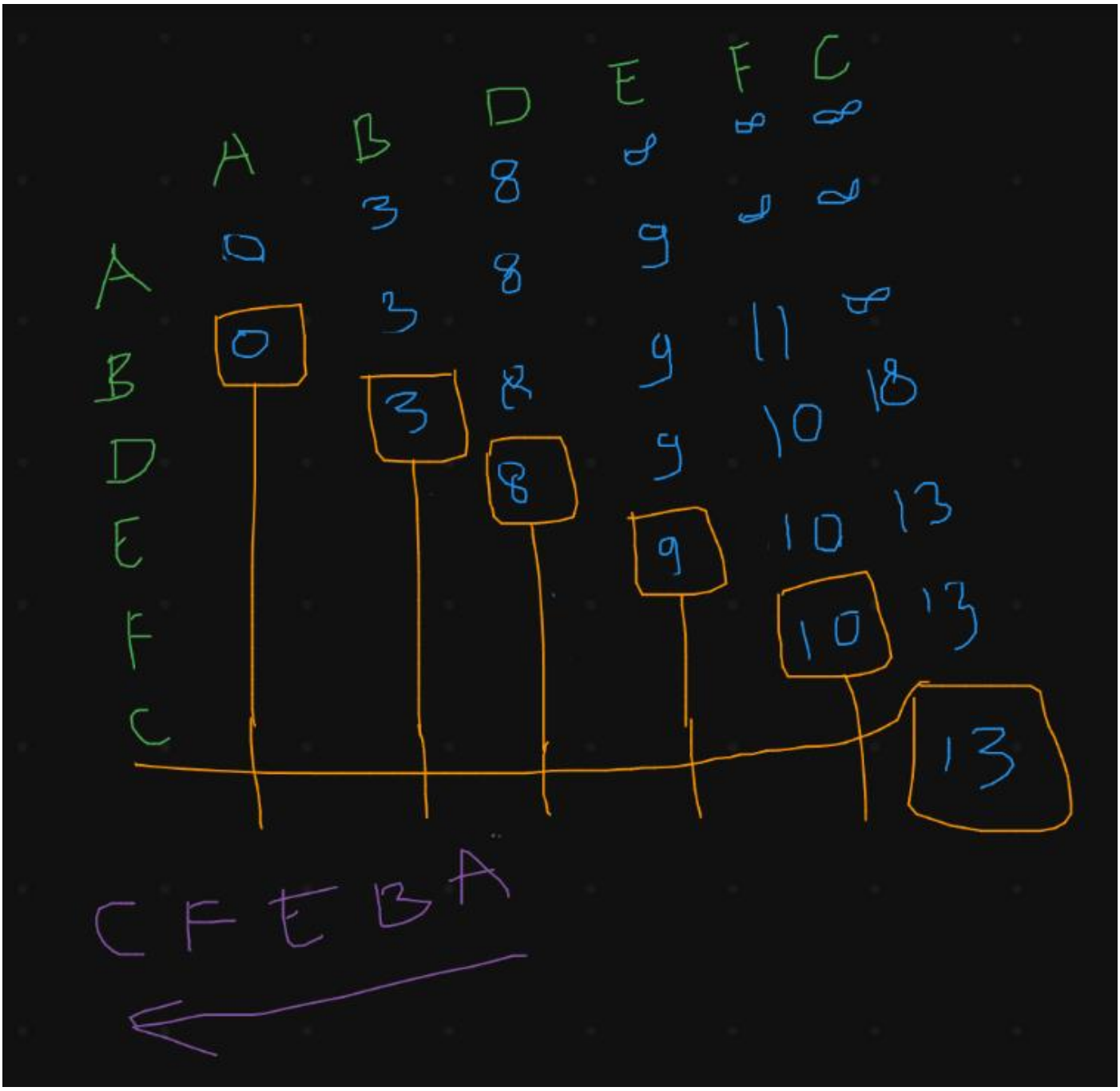
Calculate the distance from the source vertex to destination vertex through this vertex 3

Calculate the distance from the source vertex to destination vertex through this vertex 4

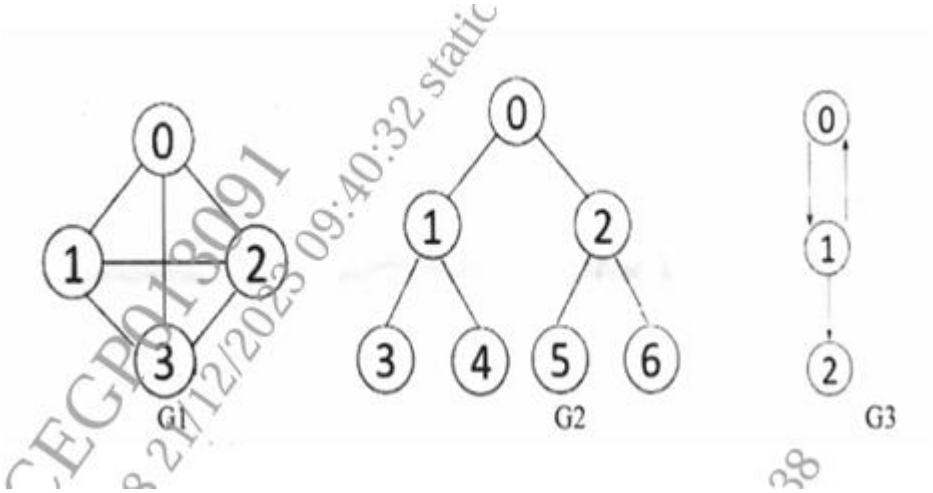
A4 gives the shortest path between each pair of vertices.

8) Apply Dijkstra's Algorithm for the graph given below, and find the shortest path from node A to node C.

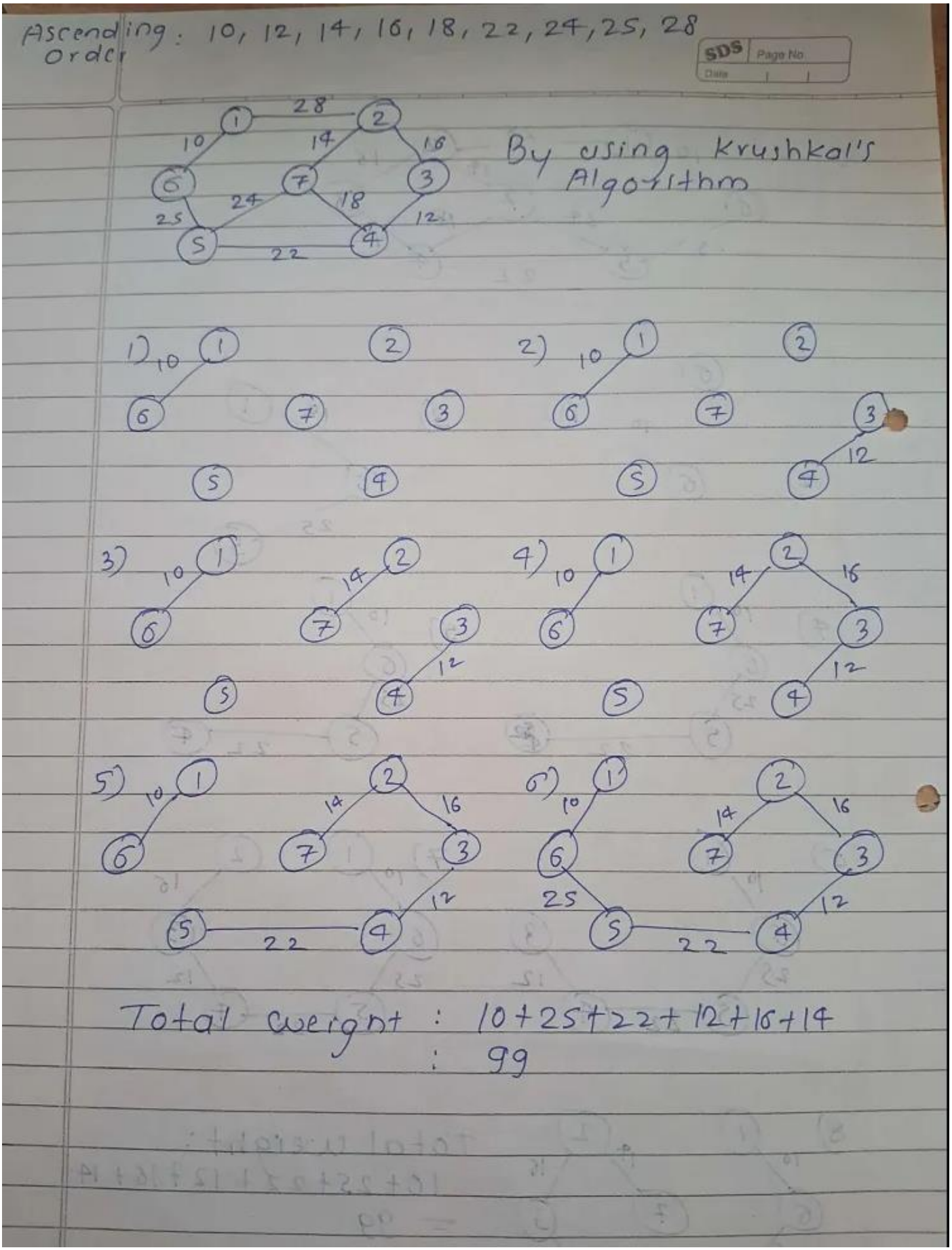




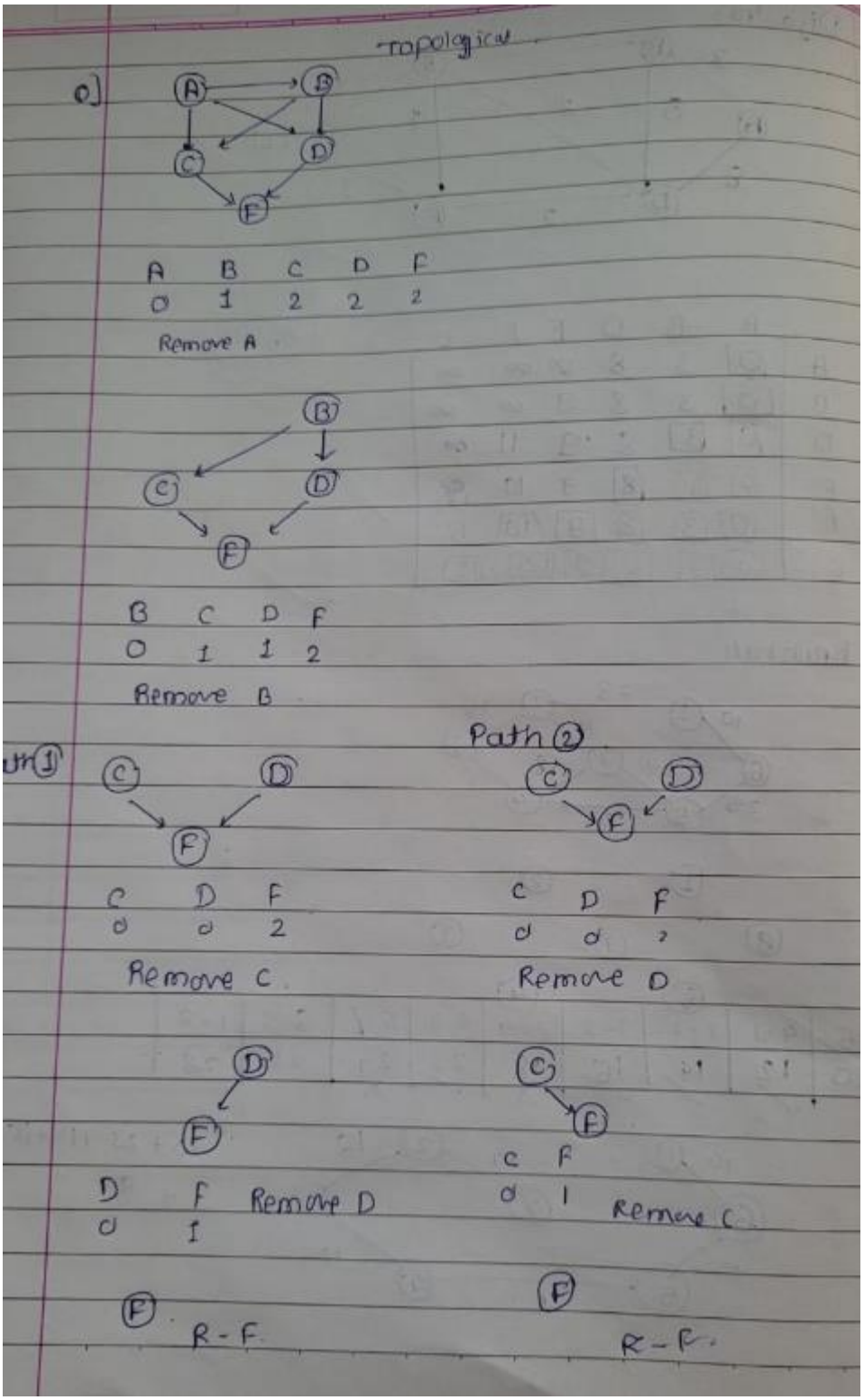
9) Define indegree & outdegree of a directed graph. Write degree for G1 & G2. Write indegree & outdegree of each vertex for G3 graph.



10) Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm.



11) Find the number of different topological orderings possible for the given graph



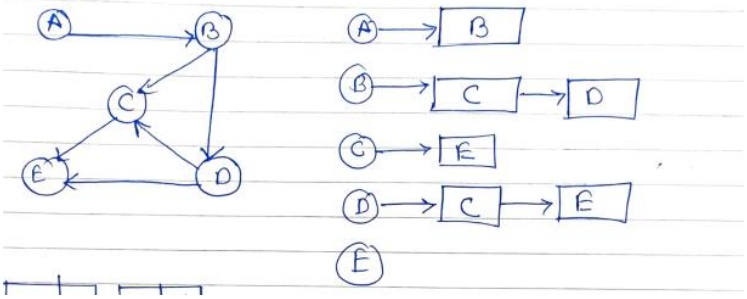
12) Elaborate following terminologies : i) Graph ii) Adjacency List iii) Adjacency Matrix

- A Graph is a non-linear data structure consisting of **nodes** and **edges**.
- The nodes are sometimes also referred to as **vertices** and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a Graph can be defined as,

A graph is a pair of set **<V, E>**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.

Adjacency List.

- In this representation, every vertex of graph contains list of its adjacency vertices.
- A linked representation is an adjacency list.
- You keep a list of neighbours for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighbouring vertices.



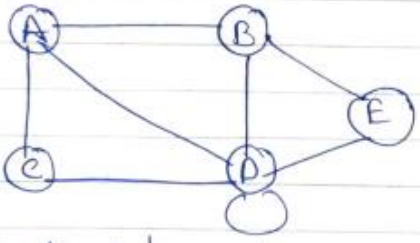
1 Adjacency Matrix :- In this representation, graph can be represented using a matrix of size total no. of vertices by total no. of vertices;

means if a graph with 4 vertices can be represented using a matrix of 4 × 4 size.

In this matrix, rows and columns both represent vertices.

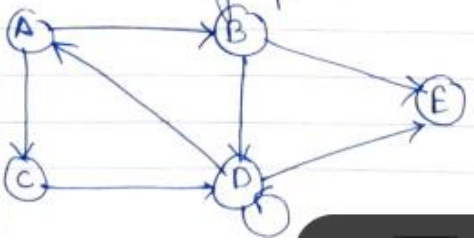
This matrix is filled with either 1 or 0. Here, 1 represent there is an edge from row vertex to column vertex and 0 represent there is no edge from row vertex to column vertex.

For undirected graph



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

For directed graph



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

13) Differentiate between tree and graph

Feature	Tree	Graph
Definition	A connected acyclic graph.	A collection of nodes (vertices) and edges (links).
Cycles	No cycles allowed (acyclic).	Cycles may or may not be present.
Connectivity	Always connected (one path between any two nodes).	May be connected or disconnected.
Edges	Has exactly `n-1` edges for `n` nodes.	Can have any number of edges (0 to `n(n-1)/2`).
Hierarchy	Hierarchical structure (parent-child).	No strict hierarchy; more general connections.
Direction	Usually directed (in rooted trees).	Can be directed or undirected.
Traversal	DFS, BFS used; simpler due to structure.	DFS, BFS used; can be more complex due to cycles.
Example	Family tree, file system hierarchy.	Social network, road map, web links.

14) Write pseudo code for Floyd-Warshall algorithm.

```
Input:
    A 2D array dist[][] representing the weight of the edge from vertex i to vertex j.
    - If there is no edge, set dist[i][j] = ∞
    - Set dist[i][i] = 0 for all i

Algorithm:

for k from 0 to V-1:                    // k is the intermediate vertex
    for i from 0 to V-1:                // i is the starting vertex
        for j from 0 to V-1:            // j is the ending vertex
            if dist[i][k] + dist[k][j] < dist[i][j]:
                dist[i][j] = dist[i][k] + dist[k][j]

Output:
    dist[][] will now contain the shortest distances between all pairs of vertices
```

Notes:

- `V` is the number of vertices in the graph.
- The triple loop checks and updates the shortest path from every vertex `i` to `j` using `k` as an intermediate step.

15) Write Prim's algorithm to find minimum spanning tree

Step 1: Keep a track of all the vertices that have been visited and added to the spanning tree.

Step 2: Initially remove all the parallel edges and loops.

Step 3: Choose a random vertex, and add it to the spanning tree. This becomes the root node.

Step 4: Add a new vertex, say x, such that

x is not in the already built spanning tree.

x is connected to the built spanning tree using minimum weight edge. (Thus, x can be adjacent to any of the nodes that have already been added in the spanning tree).

Adding x to the spanning tree should not form cycles.

Step 5: Repeat the Step 4, till all the vertices of the graph are added to the spanning tree.

Step 6: Print the total cost of the spanning tree.

16) Write the applications of : i) Graph ii) BFS iii) DFS

1) Applications of Graphs:

Graphs are widely used in various fields. Some key applications include:

Application Area	Description
Social Networks	Nodes represent people, edges represent relationships/friendships.
Computer Networks	Routers and devices as nodes, connections as edges.
Google Maps / GPS	Locations as nodes, roads as edges for route finding.
Web Crawling	Pages are nodes, hyperlinks are edges.
Dependency Graphs	Used in compilers, task scheduling (e.g., package installations).
Recommendation Systems	Products/users are nodes; preferences form edges.
AI / Game Development	Pathfinding algorithms on maps or grids (e.g., A*, Dijkstra).

2) Applications of BFS (Breadth-First Search):

BFS explores level by level and is useful where shortest path or minimal steps are needed.

Application	Description
Shortest Path in Unweighted Graphs	BFS finds shortest path by exploring all neighbors first.
Web Crawlers	Visit web pages level-wise from a source link.
Social Networking	Finding people within 'k' connections (friend suggestions).
Broadcasting in Networks	Data spread evenly like BFS across nodes.
Cycle Detection in Undirected Graphs	Can be used to detect loops/cycles.
AI / Puzzle Solving	State-space search like solving Rubik's cube or chess moves.

3) Applications of DFS (Depth-First Search):

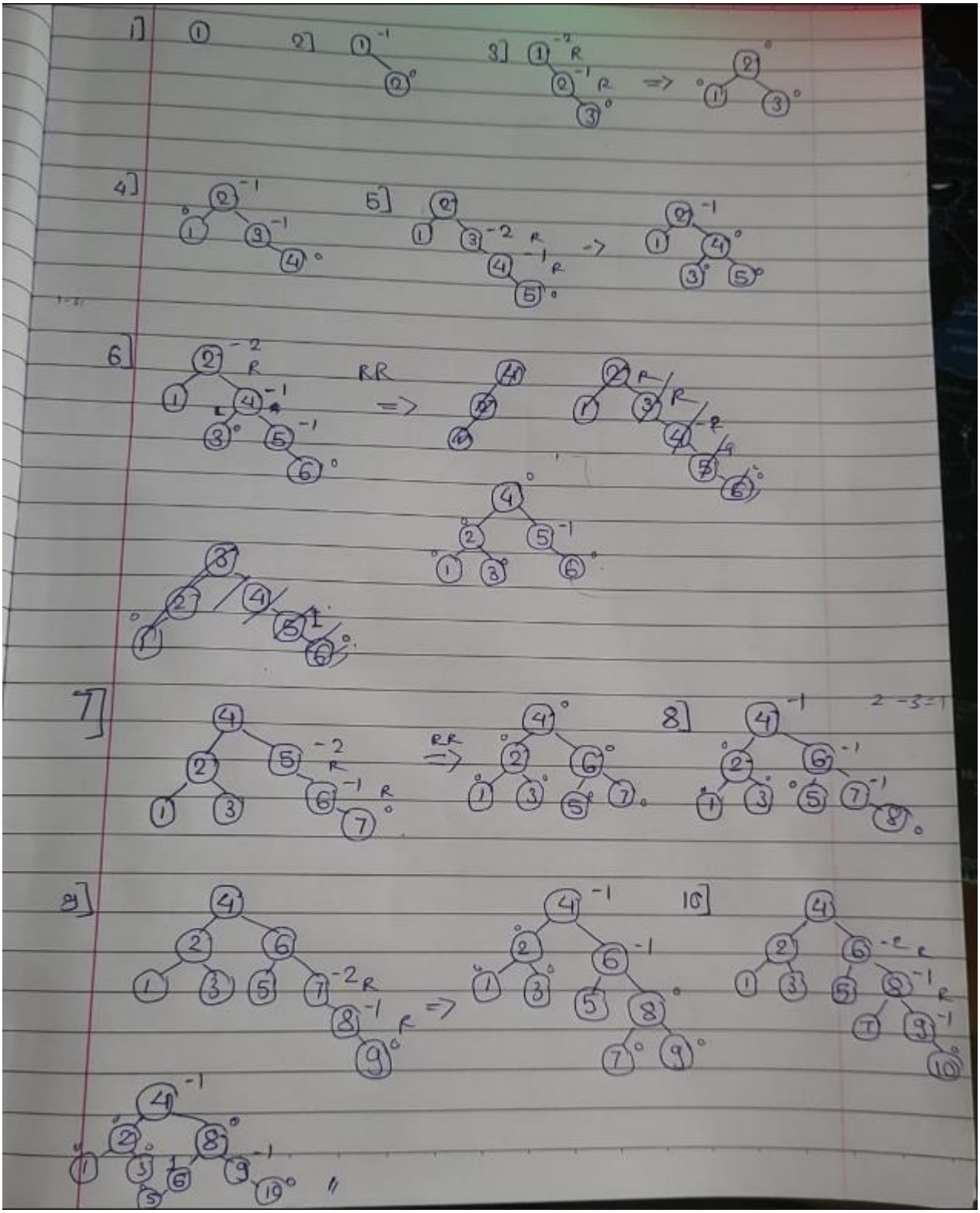
DFS goes deep into one path before backtracking, useful for complex traversal or backtracking.

Application	Description
Topological Sorting	For scheduling tasks with dependencies (like course prerequisites).
Cycle Detection in Directed Graphs	DFS is ideal for detecting cycles.
Maze Solving / Path Finding	DFS explores one full path before backtracking, useful in backtracking problems.
Connected Components	Identify all connected subgraphs in a graph.
Solving Puzzles / Games	Like Sudoku, N-Queens using recursive DFS.
Artificial Intelligence (Game Tree)	Simulate different game move paths.

DSA

UNIT 4

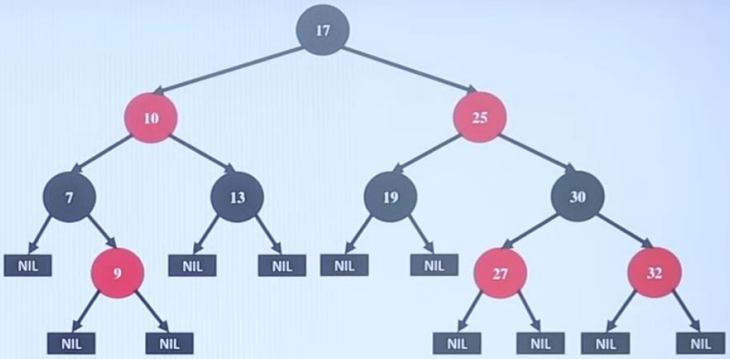
17) Construct an AVL Tree by inserting numbers from 1 to 8



18) Define Red Black tree. List its properties. Give example of it.

- A Red-Black Tree is a type of self-balancing binary search tree with specific properties that ensure it remains approximately balanced, providing efficient insertion, deletion, and lookup operations. It was introduced by Rudolf Bayer in 1972.

Properties of a Red-Black Tree



1. **Node Color**:
   Each node is either red or black.
2. **Root Property**:
   The root is always black.
3. **Red Node Property**:
   Red nodes cannot have red children (i.e., no two red nodes can be adjacent).
4. **Black Depth Property**:
   Every path from a node to its descendant leaves must have the same number of black nodes.
5. **Leaf Nodes**:
   All leaf nodes (NIL nodes) are black and do not store any data. They are placeholders used to maintain the tree's structure.

19) Write functions for RR and RL rotation with respect to AVL tree.

🔁 RR Rotation (Right-Right Case) – *Single Left Rotation*

When it happens:

- A new node is inserted into the **right subtree** of the **right child** of an unbalanced node.
- This causes the balance factor of the node to become **-2**, and the balance factor of its right child is **≤ 0**.

Logic:

1. Identify the unbalanced node (let's call it `A`).
2. Let `B` be `A`'s right child.
3. Perform a **single left rotation**:
   - `A` becomes the **left child** of `B`.
   - The **left subtree of** `B` (if any) becomes the **right subtree of** `A`.
4. Update the heights of the affected nodes.

Goal: Shift the heavier subtree (right-right chain) up to restore balance.

🔁 RL Rotation (Right-Left Case) – *Double Rotation: Right then Left*

When it happens:

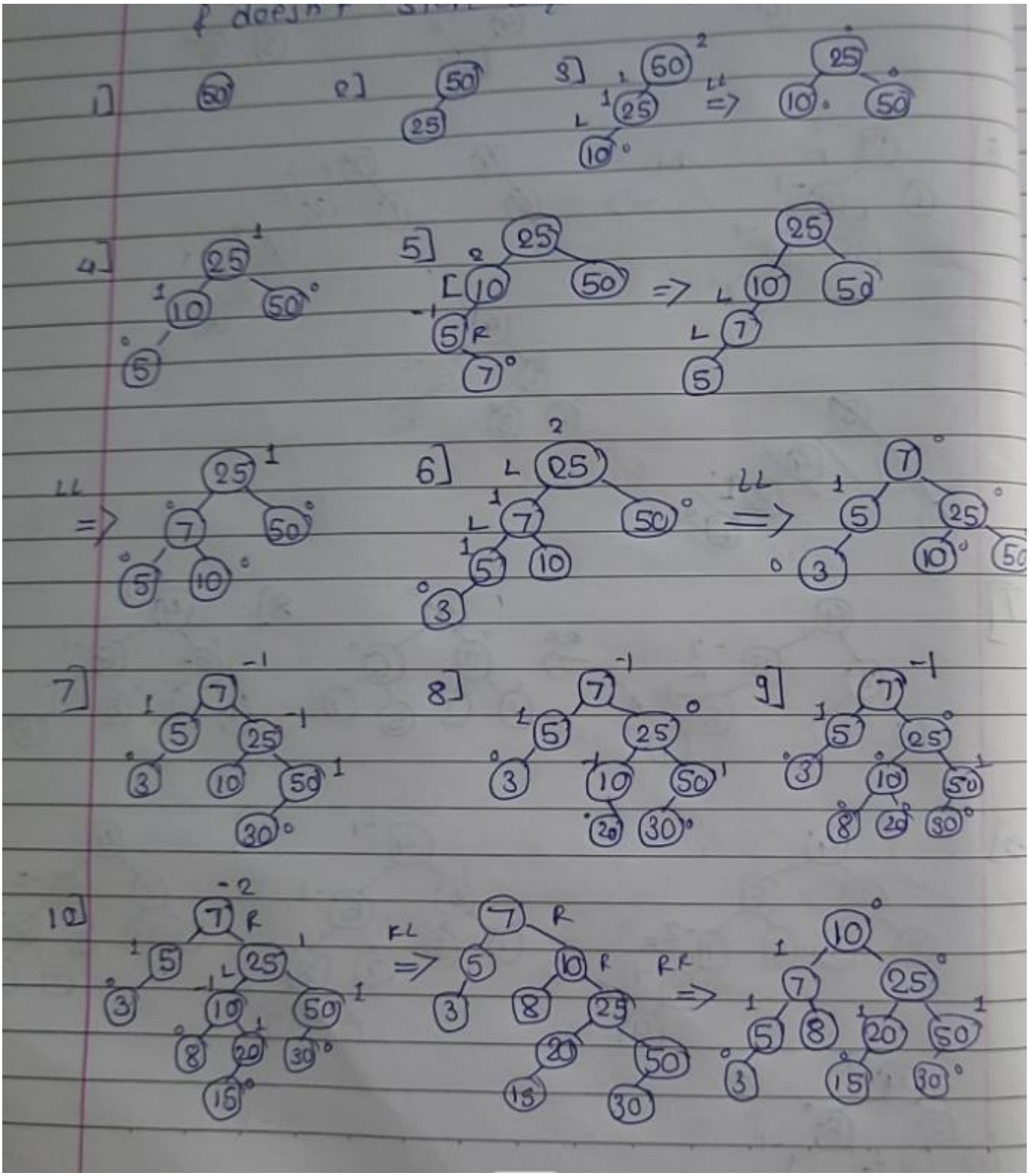
- A new node is inserted into the **left subtree** of the **right child** of an unbalanced node.
- This causes the balance factor of the node to be **-2**, and the balance factor of its right child is **> 0**.

Logic:

1. Identify the unbalanced node (`A`).
2. Let `B` be `A`'s right child, and `C` be `B`'s left child.
3. First, perform a **right rotation** on `B`:
   - `C` becomes the new right child of `A`.
4. Then perform a **left rotation** on `A`:
   - `C` becomes the new root of this subtree.
   - `A` becomes `C`'s left child, and `B` becomes `C`'s right child.
5. Update the heights of the affected nodes.

Goal: Break the zig-zag pattern and rebalance by lifting the middle node `C` up.

20) Construct an AVL Tree for following data : 50, 25, 10, 5, 7, 3, 30, 20, 8, 15

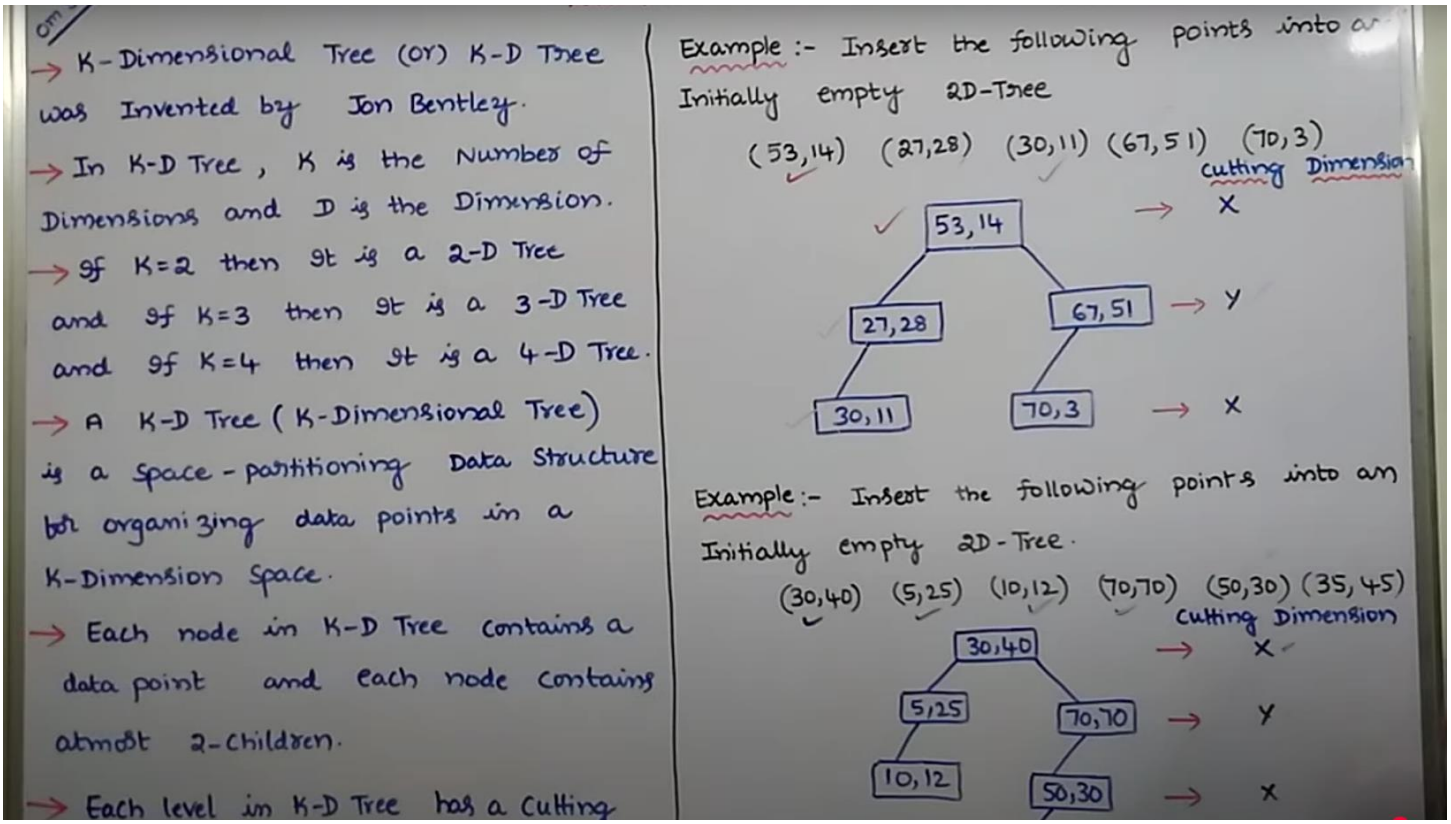


21) Explain with example K dimensional tree.

A **K-dimensional (K-d) tree** is a **binary search tree** used to organize points in a **K-dimensional space**. It is commonly used for multidimensional searches like range queries and nearest neighbor searches.

✅ Core Idea

- Each node represents a point in K-dimensional space.
- The tree cycles through the dimensions to split the space:
  - At depth `d`, split on dimension `d % K`.
- Left subtree contains points **less than** the node on that dimension.
- Right subtree contains points **greater than or equal to** the node on that dimension.



22) Explain static and dynamic tree tables with suitable example.

Representation of Symbol Table

Symbol table can be represented by

* Static Tree table
* Dynamic Tree table

Static Symbol table are in general used to store fixed amount of information whereas the dynamic Symbol tables are used for storing the information in dynamic form.

* Static tree tables :- when Symbol are known in advance and no insertion and deletion is allowed, it is called Static tree table
* Example of this type of table is a reserved word table in a compiler.
* Dynamic tree table is used when Symbols are not known in advance but are inserted as they come and deleted if not required.

Example of dynamic Symbol table : An AVL tree is Possible to implement with the help of Static Symbol table.

Example of Static Symbol table : Optimal Binary Search tree (OBST), Huffman coding is Possible to implement with the help of Static Symbol table.

This can be represented as a **static tree table** because:

- The hierarchy is known beforehand.
- No frequent updates.
- Used mostly for display purposes.

📄 Static Table Data:

ID	Name	ParentID
1	CEO	NULL
2	CTO	1
3	Dev Manager	2
4	QA Manager	2
5	CFO	1
6	Accountant	5

🌳 What Are Tree Tables?

Tree tables are hierarchical data structures used to represent data in a tree-like format—commonly in databases, UI tables, and memory structures. They allow parent-child relationships between rows (nodes).

📌 Static Tree Table

🔷 Definition:

A **static tree table** has a **fixed structure** defined at compile-time or load-time. The relationships and data are **not meant to change** during runtime. These are typically **read-only** structures.

✅ Example:

Suppose you have a **company org chart** that doesn't change frequently:

objectivec Copy Edit

```
CEO
├── CTO
│   ├── Dev Manager
│   └── QA Manager
└── CFO
    └── Accountant
```

🔄 Dynamic Tree Table

🔷 Definition:

A **dynamic tree table** supports **runtime modifications** such as insertion, deletion, and restructuring of nodes. It's used where hierarchy **frequently changes**, like in **file systems** or **project task managers**.

✅ Example:

A file system directory:

arduino Copy Edit

```
/home
├── user1
│   └── docs
└── user2
    └── photos
```

Here, users can:

- Add/delete folders.
- Move items between directories.
- Dynamically expand/collapse subtrees.

📄 Dynamic Table Data (updatable in real-time):

ID	Name	ParentID
1	home	NULL
2	user1	1
3	docs	2
4	user2	1
5	photos	4

If a new folder `music` is added under `user2`, it updates dynamically:

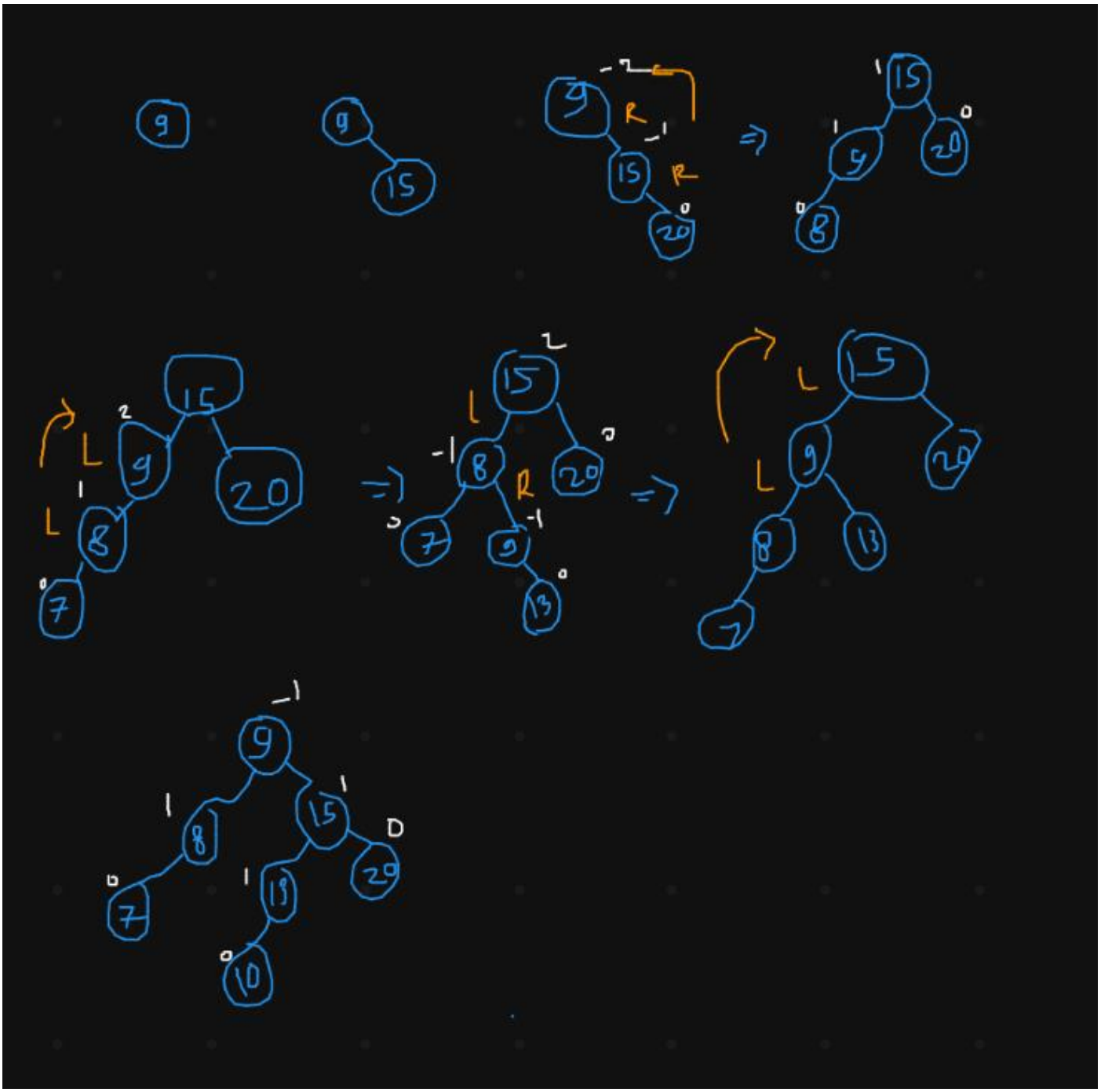
sql Copy Edit

```sql
INSERT INTO tree_table VALUES (6, 'music', 4);
```

🔍 Key Differences

Feature	Static Tree Table	Dynamic Tree Table
Structure	Fixed	Modifiable at runtime
Use case	Org charts, category trees	File explorers, task managers
Performance	Faster (no updates)	Slower (due to updates)

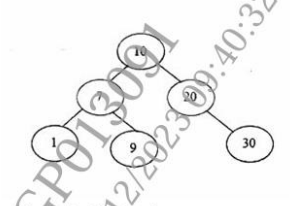
23) Construct AVL tree for insertion of following data 9, 15, 20, 8, 7, 13, 10.



24) Draw splay tree after

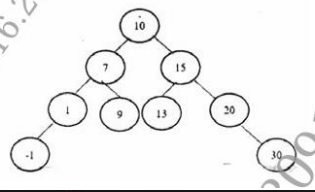
i) Zig rotation

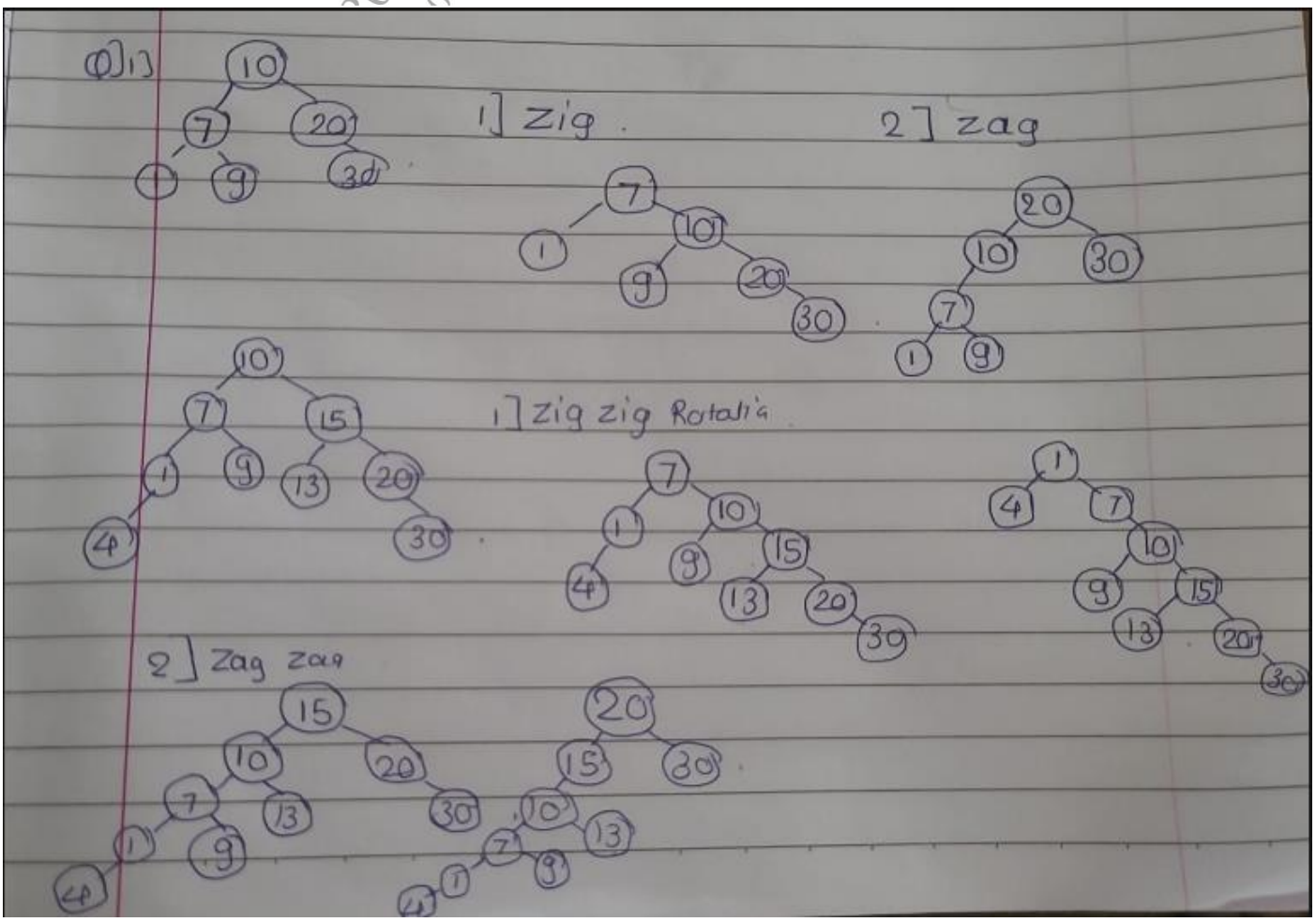
ii) Zag rotation for follwoing tree-



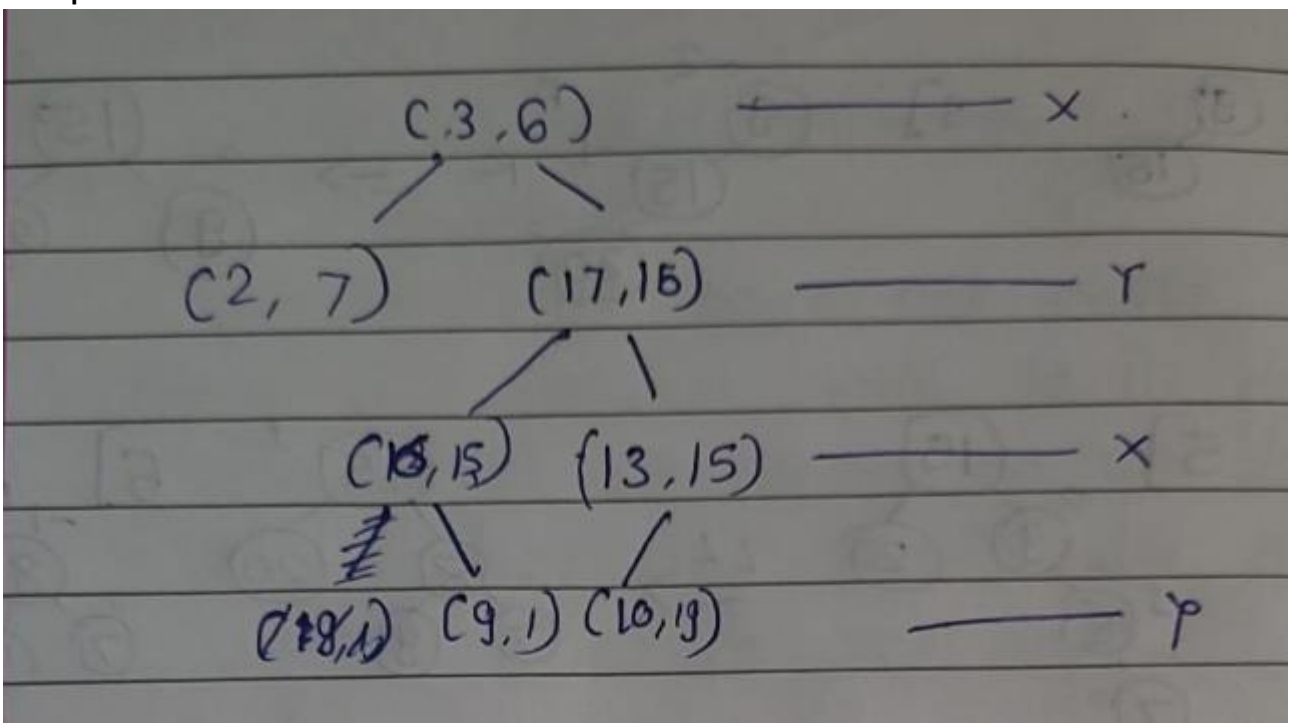
iii) Zig Zig Rotation

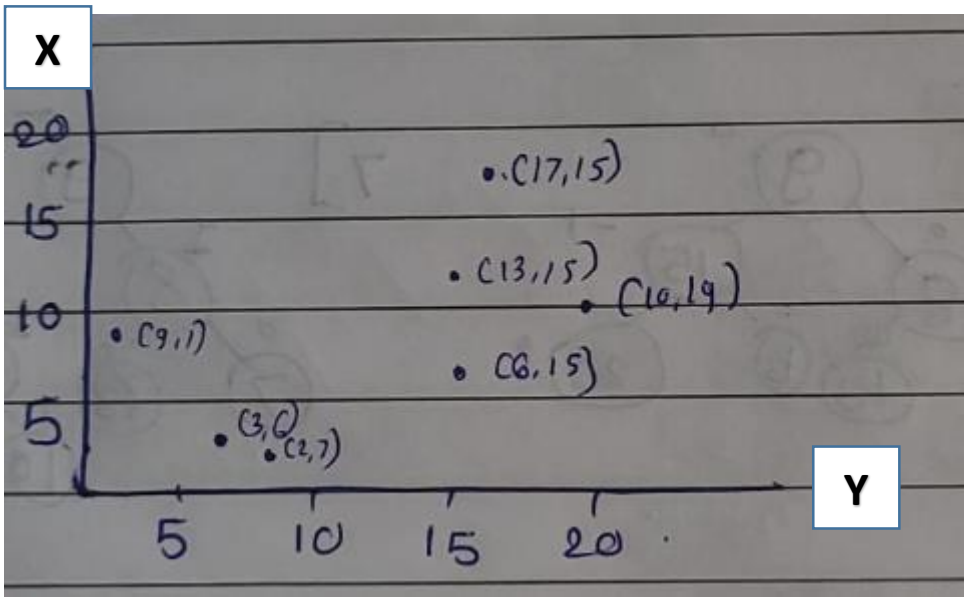
iv) Zag Zag Rotations for following tree-



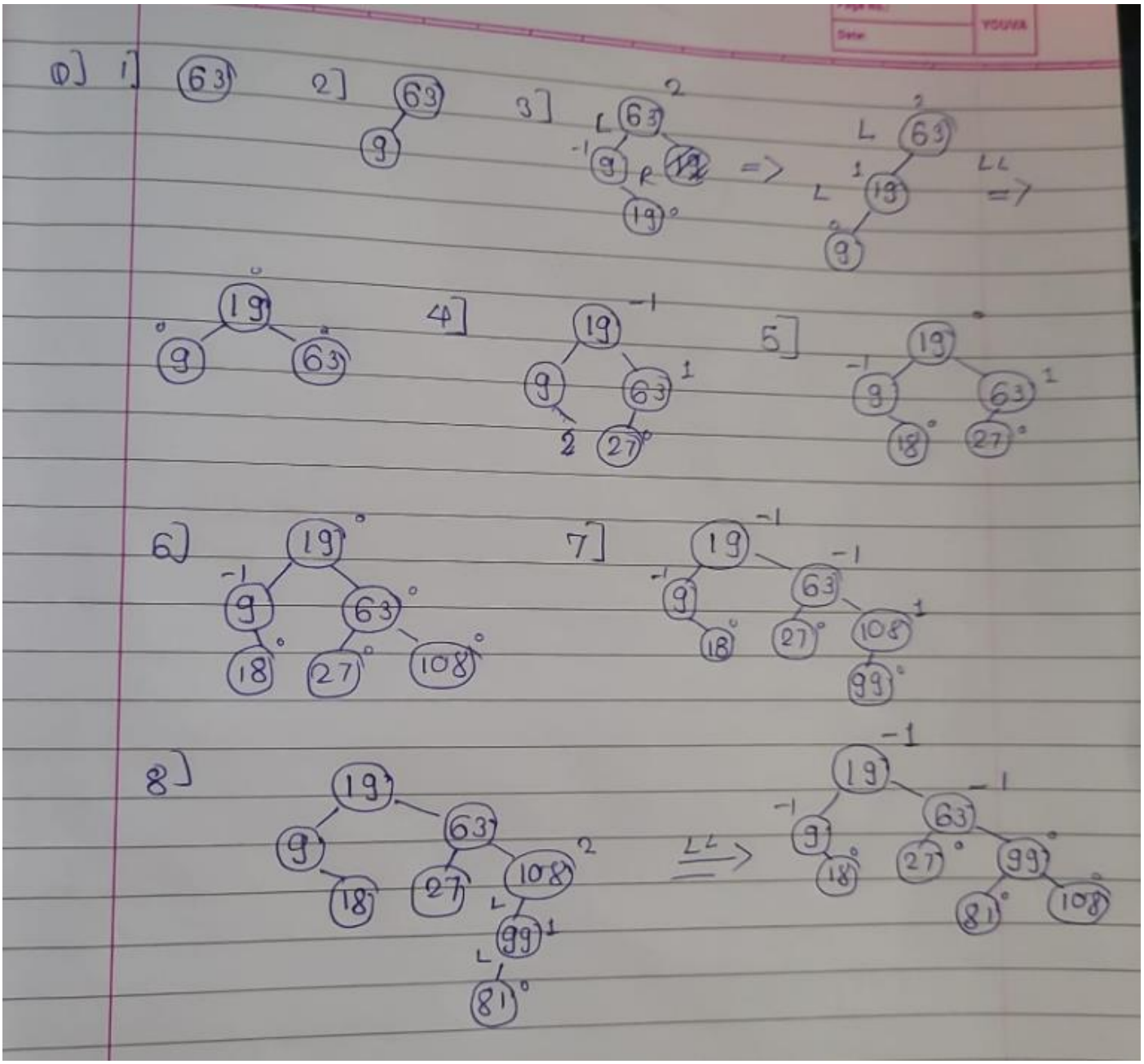


25) Create 2D tree for following data: (3, 6), (17, 15), (13, 15), (6, 12), (9, 1) (2,7), (10, 19). Also plot all the points in XY plane.





26) Construct AVL tree for insertion of following data: 63, 9, 19, 27, 18, 108, 99, 81



27) Write the functions for split & skew operations in AA tree.

🔁 `Skew()` – Logic

Goal:
Remove a **left horizontal link**, which violates the AA tree invariant that **left children must be at a lower level than their parent**.

When to apply:
If a node's **left child exists** and its **level is equal** to the node's level.

What it does:

- Perform a **right rotation** between the node and its left child.
- This fixes the issue by making the former left child the new parent and decreasing the depth of the left link.

Effect:
Balances the tree by ensuring all horizontal (same-level) links go **to the right** only.

🔁 `Split()` – Logic

Goal:
Break **two consecutive right horizontal links**, which would make the tree unbalanced and deeper on the right side.

When to apply:
If a node has a **right child**, and that right child also has a **right child**, and both are at the **same level** as the node.

What it does:

- Perform a **left rotation** between the node and its right child.
- Increase the level of the new parent (the rotated node) by 1.

Effect:
Prevents the tree from degenerating into a right-leaning linked list. Maintains logarithmic depth.

28) Explain following terms w.r.t. symbol table : i) Insert & lookup operations ii) Advantages iii) Disadvantages

Symbol Table – Overview

A **symbol table** is a data structure used by **compilers or interpreters** to store **information about identifiers** (like variable names, function names, objects, etc.) appearing in the source code.

---

i) Insert Operation

Purpose:
To **add a new identifier** (symbol) to the symbol table with relevant attributes like type, scope, memory location, etc.

When it's used:
During **declaration** of variables, functions, classes, etc.

Example:
If the compiler encounters `int x;`
→ It inserts an entry: `{ name: "x", type: "int", scope: "local/global", memory_location: ... }`

ii) Lookup Operation

Purpose:
To **retrieve information** about a symbol when it's **referenced** in code.

When it's used:
During **type checking, code generation, or semantic analysis**, the compiler looks up identifiers to verify their declarations.

Example:
If code has `x = x + 1;`
→ Lookup checks whether `x` was declared and retrieves its type or scope to ensure correctness.

---

iii) Advantages of Symbol Tables

1. ✅ **Efficient identifier management** – Keeps track of variables, functions, scopes.
2. ✅ **Enables semantic checks** – Ensures variables are declared before use.
3. ✅ **Supports nested scopes** – Helps in managing local/global variables.
4. ✅ **Optimizes memory usage** – Manages memory locations for variables.
5. ✅ **Crucial for code generation** – Helps map identifiers to actual memory addresses.

iv) Disadvantages of Symbol Tables

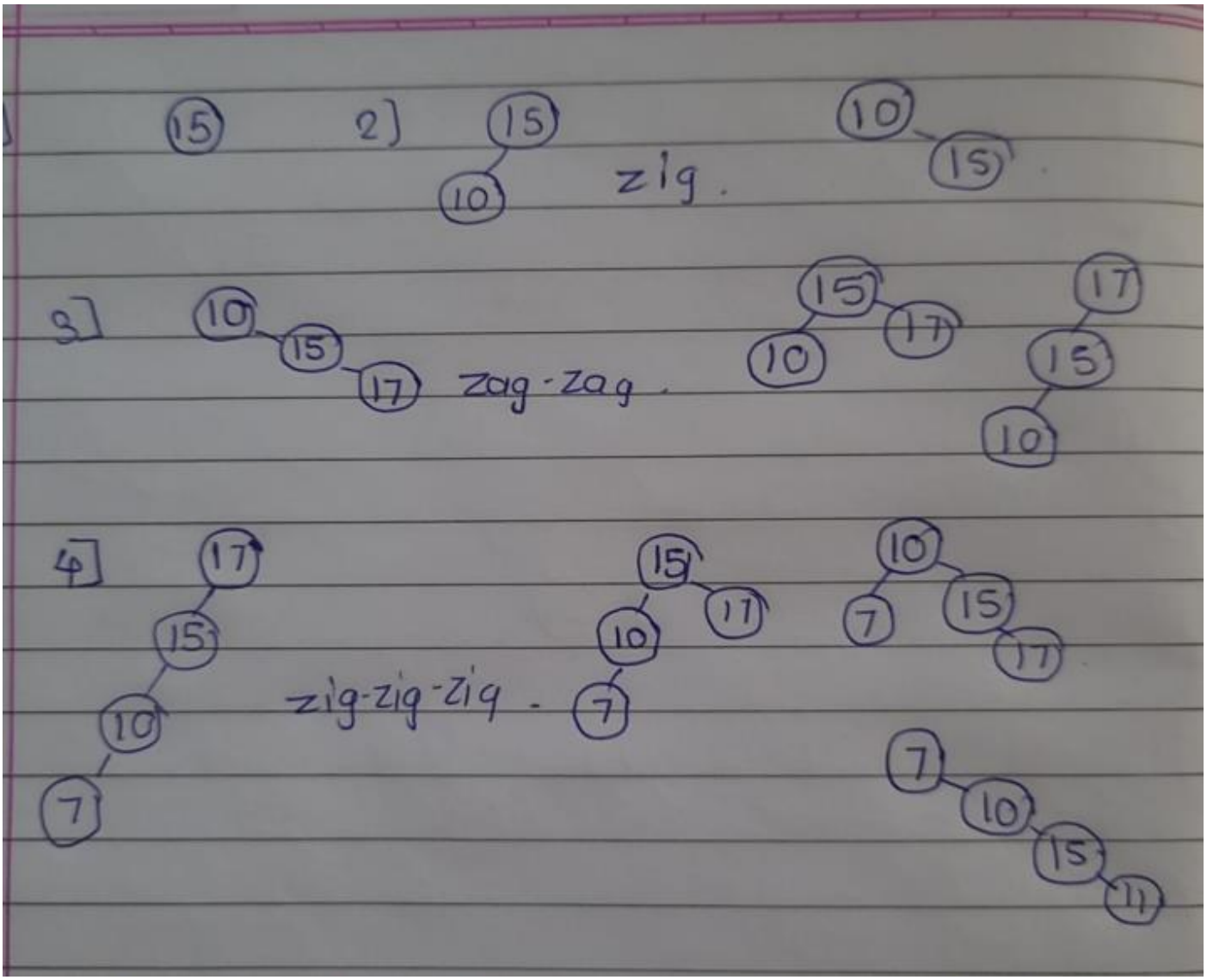
1. ❌ **Performance overhead** – Especially for large programs or with poor data structure choices.
2. ❌ **Complexity in scope handling** – Managing nested scopes can require extra logic (e.g., stack of tables).
3. ❌ **Memory consumption** – Symbol tables can grow large, consuming memory if not managed properly.
4. ❌ **Implementation difficulty** – Requires careful planning of structure (e.g., using hash tables, trees, etc.).

📌 Common Data Structures Used

- **Hash Tables** – Fast lookup and insertion (most common).
- **Binary Search Trees (BSTs)** – Useful for sorted order.
- **Tries** – Efficient for storing keywords and identifiers.
- **Stacks of Tables** – For handling scopes in block-structured languages.

29) Construct an AVL tree having the following elements : H, I, J, B, A, E, C, F, D, G

30) Insert 15, 10, 17, 7 in splay tree.



31) What is the need of AA tree? List the five invariants that AA tree must satisfy.

✅ What is the Need for an AA Tree?

The **AA Tree** is a type of self-balancing binary search tree designed to **simplify the implementation** of balancing logic while maintaining good performance. It is primarily used to:

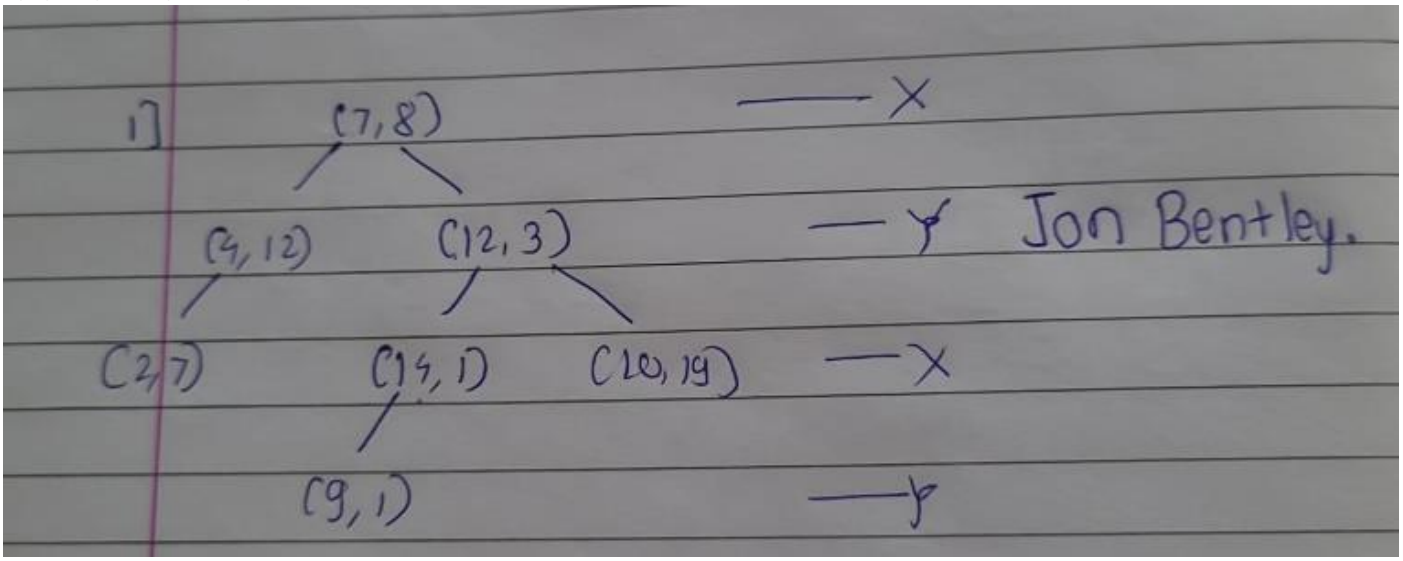
1. ✅ **Maintain a balanced BST** with simpler logic than red-black trees.
2. ✅ Ensure **logarithmic time** for insertions, deletions, and lookups.
3. ✅ Provide a data structure that is **easier to code, test, and debug** compared to AVL or red-black trees.
4. ✅ Offer a good **balance between simplicity and performance** in practice.

📏 Five Invariants of an AA Tree

To preserve its structure and ensure balance, an AA tree must satisfy these **five invariants**:

1. **Invariant 1 – Left Children Rule**:
   The **left child of a node must have a level strictly less** than its parent.
   (Prevents left horizontal links.)
2. **Invariant 2 – Right Children Rule**:
   The **right child of a node may have the same level** or one level less than the parent.
   (Allows right horizontal links, like red links in red-black trees.)
3. **Invariant 3 – No Right-Right Chain at Same Level**:
   A node **cannot have a right child and right grandchild both at the same level**.
   (Prevents long right chains, preserving balance.)
4. **Invariant 4 – Level of Leaf Nodes**:
   All **leaf nodes must have level 1**.
5. **Invariant 5 – Node with Level > 1 Must Have Two Children**:
   Any node with **level greater than 1 must have both left and right children** (non-null).

32) Who developed K-D tree? What is the purpose of K-O tree? Insert step by step (7, 8), (12, 3), (14, 1), (4, 12), (9, 1), (2, 7) and (10, 19) into K-D tree.



33) Show the balanced AVL tree after deletion of mentioned node : i) Delete 30 ii) Delete 55 iii) Delete 60