

UNDERGRADUATE LEARNING SERIES / UNIT III

DATA STRUCTURES TREES & GRAPHS

BST, AVL से Shortest Path एवं MST तक

Hindi-English explanations के साथ

14 अध्यायों में Unit III की पूर्ण तैयारी

TREES / AVL

BFS / DFS

DIJKSTRA / MST

Hindi + English Edition

Concepts · Diagrams · Algorithms · MCQs

पुस्तक का परिचय

Unit III / How to Use This Book

यह पुस्तक दिए syllabus के Trees और Graphs topics cover करती है: tree definition एवं types, binary-tree properties, insertion/deletion/search, preorder/postorder/inorder traversals, BST implementation, AVL trees; graph definition एवं types, adjacency/incidence representations using matrix और linked lists, BFS/DFS, connectivity, weighted graphs, shortest paths, spanning trees, minimum spanning trees, Kruskal's और Prim's algorithms.

14 अध्यायों में Hindi-English explanation, sharp vector diagrams, comparison tables, solved numericals, complete pseudocode, dry runs और 40 MCQs with answer key हैं। अंतिम अध्याय written-exam practice और tree/graph lab देता है।

Learning method

Figure draw करें, traversal manually लिखें, फिर pseudocode dry-run करें। Graph algorithms में visited, parent, distance, key और selected edges का अर्थ अलग समझें। MCQs पहले स्वयं solve करें।

Conventions और Assumptions

Language-independent pseudocode है, directly runnable source code नहीं। Assignment $\leftarrow$, missing node NIL; "report error" current operation समाप्त करता है। Height edges में: empty tree -1, leaf 0, root depth 0. BST duplicate keys ignore करता है। AVL में stored heights इसी convention पर हैं।

Graph traversal neighbors alphabetical order में जहाँ specified है। Main worked graph undirected, connected और nonnegative weighted है। Dijkstra nonnegative weights माँगता है; MST में negative weights allowed हैं। No-edge marker और zero-weight edge अलग रखें। Generic binary-tree deletion और BST deletion अलग policies हैं।

Syllabus Mapping

Topic	Chapters
Tree terminology, binary types/properties/operations	1-3
Traversals, BST, AVL	4-6
Graph types, representations, BFS/DFS, connectivity	7-10
Weighted graphs and shortest paths	11
Spanning trees, Kruskal and Prim	12-13
Revision, MCQs and practical work	14

Algorithm complexity depends on representation and helper data structures. Tables state those assumptions. Diagrams show logical relationships, not exact memory addresses.

विषय-सूची

Contents / Chapters 1-14

01. Trees की Definition एवं Terminology	4
02. Binary Trees: Types एवं Properties	6
03. Binary Tree Representation एवं Basic Operations	7
04. Tree Traversal Algorithms	9
05. Binary Search Trees (BST)	11
06. AVL Trees एवं Rotations	13
07. Graph Definition एवं Types	15
08. Graph Representation	17
09. Breadth First एवं Depth First Traversal	19
10. Connectivity of Graphs	21
11. Weighted Graphs एवं Shortest Path	23
12. Spanning Trees, MST एवं Kruskal's Algorithm	25
13. Prim's Algorithm एवं MST Comparison	27
14. Complete Revision एवं Unit Test	29

UNIT III / CHAPTER 01

अध्याय 1: Trees की Definition एवं Terminology

Hierarchy और parent-child relationships

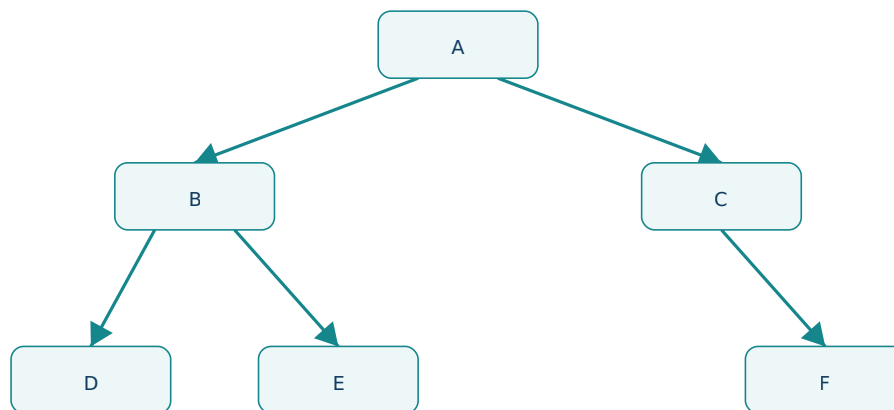


Figure 1.1: Shared binary tree. Root depth 0; leaves D, E, F have depth 2.

Hindi समझ: Tree में data को hierarchy के रूप में रखते हैं। Root से नीचे parent-child संबंध बनते हैं; leaf के children नहीं होते। Height और depth की अलग definitions याद रखें।

1.1 Tree क्या है?

Graph-theoretic tree connected undirected acyclic graph है। Rooted tree में एक node root चुनी जाती है; उससे parent-child hierarchy बनती है। Data structures में “tree” अक्सर rooted organization दर्शाता है। Empty tree को representation में allow किया जा सकता है; $n-1$ edge property nonempty tree के लिए है।

File hierarchy, organization chart, expression syntax और search indexes tree applications हैं। Ordinary tree graph में any two vertices के बीच exactly one simple path होता है। Directed rooted-tree drawings में arrows parent से child दिखा सकती हैं, लेकिन underlying graph-tree definition undirected है।

1.2 Terminology

Term	अर्थ
Root	Distinguished top node; no parent
Parent/child	Direct upper/lower relation
Siblings	Same parent वाले children
Leaf/external node	No children
Internal node	At least one child

Term	अर्थ
Ancestor/descendant	Path-based upper/lower relation
Subtree	Node और उसके descendants
Forest	Disjoint trees का collection

1.3 Degree, Depth एवं Height

Node degree in rooted-tree terminology = number of children. Graph vertex degree = number of incident edges; nonroot node के लिए parent edge भी count होती है. दोनों meanings mix न करें.

Depth = root से node तक edges. Root depth 0. Node height = longest descending path के edges; leaf height 0. Tree height = root height. Empty-tree height convention here -1. Some books node/level counts use करती हैं; formulas से पहले convention check करें.

1.4 Shared Example

Root A; children B, C; B's children D, E; C's right child F. Nodes 6, edges 5, leaves D, E, F. $\text{depth}(E)=2$, $\text{height}(B)=1$, $\text{height}(\text{tree})=2$. B, C siblings; A ancestor of F; B subtree has B, D, E.

1.5 Types एवं Need

General tree arbitrary child count; m-ary tree at most m children under this book's usage; binary tree at most 2 ordered positions left/right. Search trees impose ordering; balanced trees control height. Trie organizes prefixes; heap imposes priority order. Not every binary tree is a BST.

Exam point: Tree is both a hierarchical data structure and a special graph. For nonempty n-node tree: $\text{edges}=n-1$, unique simple paths, connected and acyclic.

Practice: In shared example graph $\text{degree}(B)=3$ but $\text{child-degree}(B)=2$. Explain why both statements correct.

UNIT III / CHAPTER 02

अध्याय 2: Binary Trees: Types एवं Properties

Two ordered child positions

Hindi समझ: Full, complete और perfect समान शब्द नहीं हैं। Full में हर node के 0 या 2 children होते हैं; complete में अंतिम level बाएँ से भरता है; perfect में सभी levels पूरी तरह भरे हैं।

2.1 Definition एवं Types

Binary tree each node has at most two children, designated left and right. A single left child and a single right child are different ordered structures.

Type	Definition
Full/proper/strict	Every node has 0 or 2 children
Complete	All levels except last full; last filled left-to-right
Perfect	All internal nodes have 2 children; all leaves same depth
Skewed	Each internal node has only one child; often all same side
Height-balanced	Subtree-height differences bounded by stated rule

Perfect is both full and complete. Full need not be complete. Complete need not be full. "Balanced" needs a precise criterion; AVL uses local difference at most 1 at every node.

2.2 Maximum Counts

With root depth 0 and height h edges:

- Maximum nodes at depth $d = 2^d$.
- Maximum total nodes through height $h = 2^{(h+1)} - 1$.
- Maximum leaves at depth $h = 2^h$.
- Minimum nodes for arbitrary binary tree of height $h = h + 1$, using a chain.

For nonempty n nodes: minimum possible height $\lceil \log_2(n+1) \rceil - 1 = \lceil \log_2 n \rceil$; maximum $n - 1$. The lower bound is achievable by complete shapes; arbitrary shape may have larger height.

2.3 Full-tree Relation

Let n_0 leaves, n_1 one-child nodes, n_2 two-child nodes. Total nodes $n = n_0 + n_1 + n_2$. Edges $= n_1 + 2n_2 = n - 1$. Therefore **$n_0 = n_2 + 1$** for any nonempty binary tree. In full tree $n_1 = 0$, $n = 2n_2 + 1 = 2n_0 - 1$.

2.4 Solved Numericals

Q1: Perfect height 3: nodes 15, leaves 8, internal 7.

Q2: Full tree internal nodes 9: leaves 10, total 19.

Q3: $n = 10$ arbitrary binary tree: min height 3, max height 9.

Q4: Binary tree has 8 two-child nodes and 5 one-child nodes: leaves 9, total 22, edges 21.

2.5 Common Traps

A complete tree may have one child at last internal node, necessarily left under standard definition. Full doesn't mean every level filled. Height 3 means 4 levels here. Node-degree 2 may mean children, not graph incident-degree 2.

Practice: Perfect height 4: total 31 nodes, 16 leaves. Verify both formulas.

UNIT III / CHAPTER 03

अध्याय 3: Binary Tree Representation एवं Basic Operations

Array slots, linked nodes और generic tree policies

Hindi समझ: Array representation में children indices से मिलते हैं; linked representation में left/right references से. सामान्य binary tree में insertion/deletion की policy तय करनी पड़ती है-यह BST ordering से अलग है.

3.1 Sequential/Array Representation

Zero-based complete-tree indexing: root 0, left child $2i+1$, right child $2i+2$, parent $\lfloor (i-1)/2 \rfloor$ for $i>0$. Child formulas give candidate indices; node existence/capacity must be checked.

One-based alternative: root 1, left $2i$, right $2i+1$, parent $\lfloor i/2 \rfloor$. Never combine conventions. Sparse/skewed shapes waste array slots with this positional scheme. Complete trees fit compactly.

3.2 Linked Representation

Node has data, left, right. Missing child=NIL; root=NIL empty. Representation supports arbitrary shapes without positional holes, but needs references and allocation overhead. Optionally parent link helps upward navigation.

Feature	Positional array	Linked nodes
Child access	Index arithmetic	Reference fields
Complete shape	Compact	Link overhead
Skewed shape	Many gaps	Only actual nodes
Growth	Capacity/resizing	Allocate nodes

3.3 Generic Binary Tree Search

No ordering rule, so key may be anywhere. DFS/BFS may visit all n nodes: worst $\Theta(n)$. Equal keys need policy: first match, all matches or unique identifiers. BST comparison-based search in chapter 5 is different.

3.4 Level-order Insertion Policy

A generic binary tree has no universal insertion rule. This example policy keeps insertion shape complete:

```
INSERT_LEVEL_ORDER(root, key)
  node ← new node(key, NIL, NIL)
  if root = NIL
    return node
  Q ← queue containing root
```

```
while Q nonempty
  p ← dequeue(Q)
  if p.left = NIL
    p.left ← node
    return root
  enqueue(Q, p.left)
  if p.right = NIL
    p.right ← node
    return root
  enqueue(Q, p.right)
```

Worst $\Theta(n)$ search and $O(n)$ queue space. Allocation failure must be handled before changing links. Existing holes filled in level-order, not according to key value.

3.5 Generic Deletion Policy

For an unordered tree, one possible policy: find target and deepest rightmost node by level-order, copy deepest node's payload to target, then remove deepest node from its parent. If only root exists, deleting it returns NIL. If key absent, unchanged. This does not preserve BST ordering; do not use it for BST deletion.

Parent tracking is needed to unlink deepest node. If target equals deepest node, simply unlink it. Identity-sensitive objects may require structural replacement rather than payload copy; define contract.

Practice: Zero-based index 4 children9, 10, parent1. Explain why an index within capacity may still represent a missing node.

UNIT III / CHAPTER 04

अध्याय 4: Tree Traversal Algorithms

Preorder, Inorder, Postorder एवं Level-order

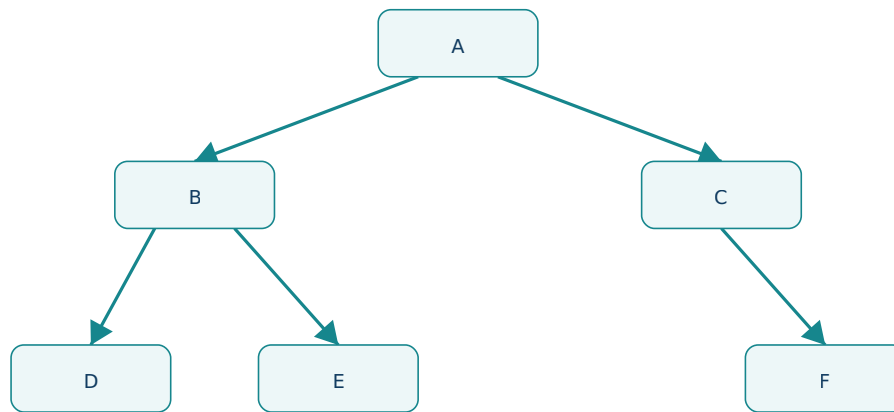


Figure 4.1: Preorder A B D E C F; inorder D B E A C F; postorder D E B F C A.

Hindi समझ: Traversal का अंतर root को visit करने के समय में है। Preorder में पहले root, inorder में दोनों subtrees के बीच root, postorder में अंत में root आता है।

4.1 Traversal Meaning

Traversal visits every node systematically. DFS traversals differ by when current node is processed relative to left/right subtrees. NIL base case essential. Shared tree: A root; B, C children; B→D, E; C→right F.

Traversal	Rule	Shared example
Preorder	Root, Left, Right	A B D E C F
Inorder	Left, Root, Right	D B E A C F
Postorder	Left, Right, Root	D E B F C A
Level-order	Depth by depth, left-to-right	A B C D E F

```

PREORDER(node)
  if node = NIL: return
  visit node.data
  PREORDER(node.left)
  PREORDER(node.right)

INORDER(node)
  if node = NIL: return
  INORDER(node.left)
  visit node.data
  INORDER(node.right)
  
```

```

POSTORDER(node)
    if node = NIL: return
    POSTORDER(node.left)
    POSTORDER(node.right)
    visit node.data

```

All $\Theta(n)$ node visits; recursion auxiliary $O(h+1)$, where h edge-height. Worst skew $O(n)$, balanced $O(\log n)$. Output storage excluded.

4.2 Iterative Preorder using Stack

```

if root ≠ NIL: push root
while stack nonempty
    p ← pop()
    visit p.data
    if p.right ≠ NIL: push p.right
    if p.left ≠ NIL: push p.left

```

Push right before left so left processed first. No visited set needed for proper acyclic tree; malformed structures with cycles violate tree invariant.

4.3 Level-order using Queue

enqueue root if non- NIL; repeatedly dequeue, visit, enqueue left then right existing children. $\Theta(n)$ time; $O(w)$ auxiliary queue, w maximum level width, hence $O(n)$ worst case.

4.4 Applications एवं Reconstruction

Preorder useful to serialize hierarchical structure with sufficient NIL markers. Inorder of BST returns sorted keys. Postorder processes children before parent, useful expression evaluation and deletion cleanup. Level-order shows breadth/levels.

Unique node labels plus inorder and preorder (or inorder and postorder) determine unique binary tree. Preorder+postorder alone do not uniquely determine arbitrary binary tree; full-tree assumptions can change result. Duplicate labels require additional information.

Practice: Root2, left1, right3: preorder2, 1, 3; inorder1, 2, 3; postorder1, 3, 2.

UNIT III / CHAPTER 05

अध्याय 5: Binary Search Trees (BST)

Ordered search, insertion और deletion

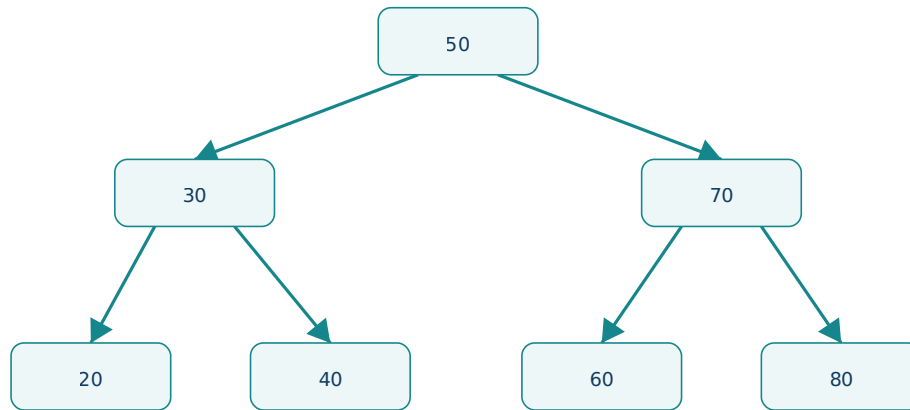


Figure 5.1: BST from insertion sequence 50, 30, 70, 20, 40, 60, 80.

Hindi समझ: BST में हर node के बाएँ छोटे और दाएँ बड़े keys होते हैं। Search में पूरा tree नहीं छानते; comparison के अनुसार एक subtree चुनते हैं। Skewed BST फिर भी धीमा हो सकता है।

5.1 BST Invariant

Here keys unique: every key in left subtree strictly smaller than nodekey; every key in right strictly larger. Duplicate insertion returns unchanged. Other duplicate policies possible, but must define them consistently.

```
SEARCH(root, key)
  p ← root
  while p ≠ NIL
    if key = p.key: return p
    if key < p.key: p ← p.left
    else: p ← p.right
  return NIL

INSERT(root, key)
  if root = NIL: return new node(key)
  if key < root.key
    root.left ← INSERT(root.left, key)
  else if key > root.key
    root.right ← INSERT(root.right, key)
  return root
```

Operations $O(h+1)$; balanced height $\Theta(\log n)$, worst skew $h=n-1 \rightarrow \Theta(n)$. Ordinary BST not automatically balanced.

5.2 Insertion Example

Insert 50, 30, 70, 20, 40, 60, 80. Root50; left30 with 20, 40; right70 with 60, 80. Search60: comparisons50→70→60. Inorder20, 30, 40, 50, 60, 70, 80.

5.3 Deletion Cases

Leaf: remove parent link. One child: replace node with child. Two children: replace key/payload by inorder successor(minimum of right subtree), then delete that successor from right. Predecessor(maximum left) alternative valid.

```
DELETE(root, key)
  if root = NIL: return NIL
  if key < root.key
    root.left ← DELETE(root.left, key)
  else if key > root.key
    root.right ← DELETE(root.right, key)
  else
    if root.left = NIL
      child ← root.right
      release root if manually managed
      return child
    if root.right = NIL
      child ← root.left
      release root if manually managed
      return child
    s ← minimum node in root.right
    root.key ← s.key
    root.payload ← s.payload if records carry payload
    root.right ← DELETE(root.right, s.key)
  return root
```

Caller must store returned root, including top-level root replacement. Read child before free. Copy entire associated record, not key alone if payload belongs to key.

5.4 Worked Deletion

Delete 50 from example: successor 60 copied to root; original 60 leaf removed from 70.left. Sorted remaining 20, 30, 40, 60, 70, 80. Root key60, not70. Absent key deletion leaves invariant unchanged.

5.5 Implementation Choices

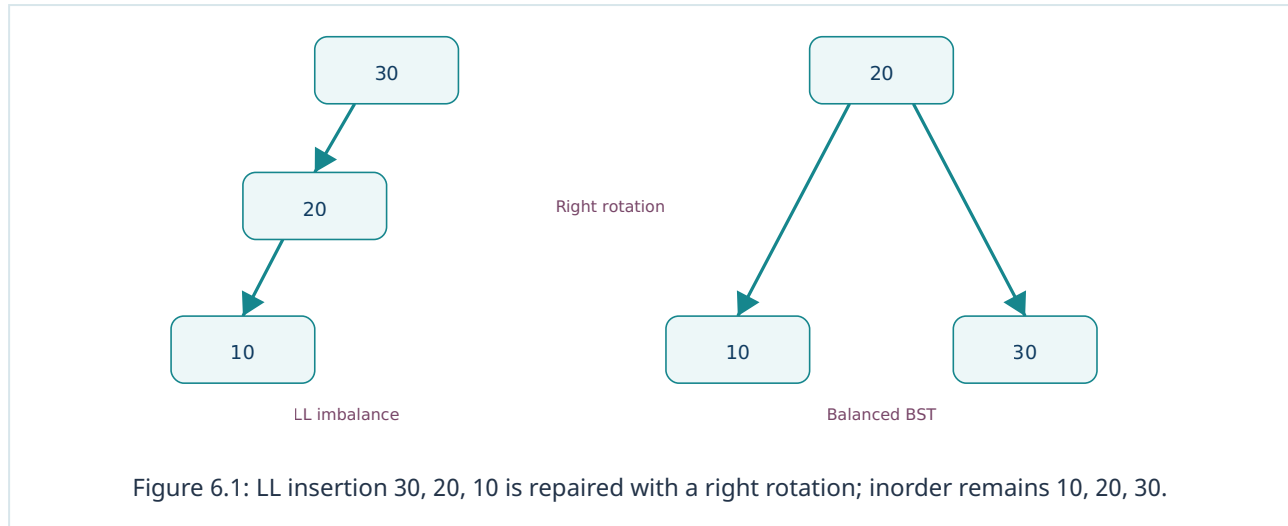
Linked nodes simplest; array of node-records with child indices also possible. Sorted array supports binary search but shifting insertion; it is not the same thing as a linked BST. Random insert assumptions may yield expected log height, but worst case guarantee needs balancing.

Practice: Increasing1..5 builds right chain height 4. Explain why binary search tree and binary search algorithm do not imply identical memory layout.

UNIT III / CHAPTER 06

अध्याय 6: AVL Trees एवं Rotations

Height balance after updates



Hindi समझ: AVL, BST की ordering को बनाए रखते हुए height balance नियंत्रित करता है। Rotation से keys का inorder क्रम नहीं बदलता। हर affected ancestor की height दोबारा update करना आवश्यक है।

6.1 Definition

AVL is self-balancing BST. Every node balance factor $BF = \text{height}(\text{left}) - \text{height}(\text{right})$ belongs $\{-1, 0, 1\}$. Height convention empty=-1, leaf 0. New AVL node height=0. Store height in each node; update after child changes. Search/insert/delete $O(\log n)$ because height $O(\log n)$.

6.2 Four Cases

Imbalance pattern	Repair
LL	Right rotation
RR	Left rotation
LR	Left rotation at left child, then right at node
RL	Right rotation at right child, then left at node

“LL” names insertion path, not rotation direction. Insert 30, 20, 10 → LL → root 20 with children 10, 30. Insert 10, 20, 30 → RR → same balanced shape. Insert 30, 10, 20 → LR; insert 10, 30, 20 → RL.

6.3 Rotation Algorithms

```

HEIGHT(NIL) = -1
UPDATE(p): p.height ← 1 + max(HEIGHT(p.left), HEIGHT(p.right))

ROTATE_RIGHT(y)
    x ← y.left
    middle ← x.right
    x.right ← y
    y.left ← middle
    UPDATE(y)
    UPDATE(x)
    return x

ROTATE_LEFT(x)
    y ← x.right
    middle ← y.left
    y.left ← x
    x.right ← middle
    UPDATE(x)
    UPDATE(y)
    return y

```

Required child exists for applicable rotation. Update old root first, then new root. Middle subtree preserved; rotations change shape but maintain inorder key order.

6.4 General Re balance: Insert एवं Delete

```

REBALANCE(p)
    if p = NIL: return NIL
    UPDATE(p)
    balance ← HEIGHT(p.left) - HEIGHT(p.right)
    if balance > 1
        if HEIGHT(p.left.left) < HEIGHT(p.left.right)
            p.left ← ROTATE_LEFT(p.left)
        return ROTATE_RIGHT(p)
    if balance < -1
        if HEIGHT(p.right.right) < HEIGHT(p.right.left)
            p.right ← ROTATE_RIGHT(p.right)
        return ROTATE_LEFT(p)
    return p

```

Recursive BST insert/delete के return path पर REBALANCE call और returned root store करें. Deletion child balance 0 case single rotation से handle हो सकती है; inserted key-based case checks deletion के लिए पर्याप्त नहीं. Deletion multiple ancestors को rebalance कर सकती है.

6.5 Minimum Nodes Numerical

Minimum AVL nodes for height h : $N(-1)=0$, $N(0)=1$, $N(h)=1+N(h-1)+N(h-2)$. Thus $N(1)=2$, $N(2)=4$, $N(3)=7$, $N(4)=12$. This recurrence implies logarithmic height, not perfect shape. AVL need not complete or perfect.

Practice: Insert 40, 20, 30. Pattern LR, final root 30, left 20, right 40. State BF values after repair.

UNIT III / CHAPTER 07

अध्याय 7: Graph Definition एवं Types

Vertices, edges और general connections

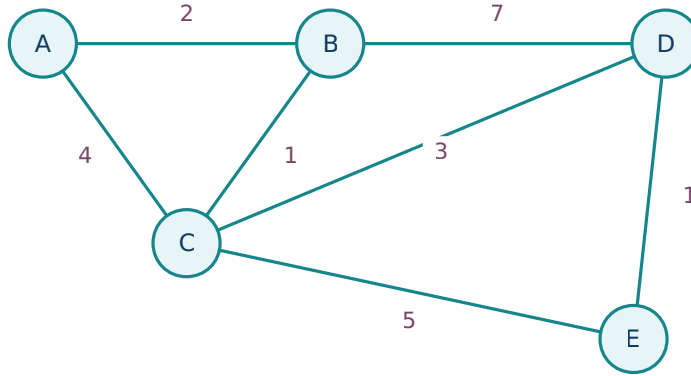


Figure 7.1: Shared undirected weighted graph. Edge weights are costs, not vertex labels.

Hindi समझ: Graph में hierarchy की पाबंदी नहीं होती: cycles, multiple connections और disconnected parts संभव हैं। Direction बताती है किस ओर जा सकते हैं; weight बताता है edge का cost.

7.1 Graph Model

$G=(V, E)$. Vertices represent entities; edges relationships. Undirected edge $\{u, v\}$ has no orientation; directed edge (u, v) is arc from u to v . Graph need not connected or acyclic. Tree is connected acyclic undirected special case.

Type	Key property
Simple undirected	No selfloops or parallel edges
Directed/digraph	Ordered arcs
Multigraph	Parallel edges may exist
Weighted	Edge cost/weight associated
Connected	Every pair has path in undirected graph
Complete	Every distinct pair adjacent
Bipartite	Vertices partition into 2 sets; edges across sets

Definitions of multigraph/pseudograph and loop allowance vary; state allowed features explicitly. Weighted/unweighted is independent of directed/undirected.

7.2 Terminology

Adjacent vertices share edge; incident edge touches vertex. Walk may repeat vertices/edges. Trail repeats no edges. Simple path repeats no vertices. Cycle closed path with distinct internal vertices under standard definition; selfloop cases depend graph type. Reachability respects arc orientation.

7.3 Degree Properties

Loopless undirected sum degrees = $2|E|$. Undirected selfloop contributes 2 to degree if allowed. Directed sum indegrees = sum outdegrees = $|E|$. Complete simple undirected K_n has $n(n-1)/2$ edges; loopless complete directed graph with both arcs has $n(n-1)$.

7.4 Shared Weighted Example

Vertices A, B, C, D, E; undirected edges:

Edge	Weight
A-B	2
A-C	4
B-C	1
B-D	7
C-D	3
C-E	5
D-E	1

Graph has 5 vertices, 7 edges, connected, contains cycles. Degrees A2, B3, C4, D3, E2; sum $14=2 \times 7$. This graph is used for representation, traversals, shortest path and MST chapters.

7.5 Applications

Transport networks, communication links, course prerequisites, social relationships and state transitions. Choose orientation/weight meaning first. An edge “distance” is not automatically time, cost or probability.

Practice: Simple undirected 6 vertices maximum 15 edges. Directed 6 vertices no loops maximum 30 arcs.

UNIT III / CHAPTER 08

अध्याय 8: Graph Representation

Adjacency, incidence, matrix एवं linked lists

Hindi समझ: Adjacency matrix vertex-to-vertex संबंध दिखाती है; incidence matrix vertex-to-edge संबंध. Linked adjacency/incidence lists sparse graphs में उपयोगी होती हैं. Zero-weight edge को missing edge से अलग पहचानें.

8.1 Adjacency Matrix

n vertices की $n \times n$ matrix. Unweighted simple graph में $M[i, j]=1$ iff edge exists, otherwise 0. Undirected matrix symmetric. Diagonal 0 if no selfloops. Directed graph's matrix need not symmetric.

Shared graph के weighted matrix में "-" no edge; diagonal 0 denotes zero-length self distance, not a loop:

From/To	A	B	C	D	E
A	0	2	4	-	-
B	2	0	1	7	-
C	4	1	0	3	5
D	-	7	3	0	1
E	-	-	5	1	0

Zero-weight edges possible हैं; इसलिए weighted representation में absent edge को 0 मानना ambiguous हो सकता है. Use separate existence flag or ∞ /NIL sentinel as algorithm requires.

8.2 Adjacency List / Linked Representation

Each vertex holds list of neighbor records, optionally weight. Shared example sorted lists:

```
A: (B,2), (C,4)
B: (A,2), (C,1), (D,7)
C: (A,4), (B,1), (D,3), (E,5)
D: (B,7), (C,3), (E,1)
E: (C,5), (D,1)
```

Undirected edge appears in both endpoint lists: $2m$ neighbor entries for loopless graph. Directed adjacency lists usually store outgoing arcs once; incoming lists optional. Linked nodes or resizable arrays can hold adjacency entries—"list" does not mandate pointers.

8.3 Incidence Matrix

Rows vertices, columns edges. For loopless undirected graph $I[v, e]=1$ when vertex touches edge, otherwise 0. Each edge column has two 1s.

Vertex	AB	AC	BC	BD	CD	CE	DE
A	1	1	0	0	0	0	0
B	1	0	1	1	0	0	0
C	0	1	1	0	1	1	0
D	0	0	0	1	1	0	1
E	0	0	0	0	0	1	1

Directed incidence convention here -1 at source, +1 at destination; other textbooks reverse signs. Selfloops require explicit convention; oriented incidence can map a loop to zero column and thus lose loop distinction.

8.4 Incidence Lists एवं Edge Objects

Incidence-list representation each vertex references its incident edge records. Each edge record stores endpoints and optional weight. Undirected edge object can be referenced by both endpoint lists, rather than storing two independent edge objects. Adjacency lists emphasize neighbors; incidence lists emphasize incident edges. Both linked organizations support traversal.

8.5 Space एवं Operation Comparison

Representation	Space	Edge existence check	Enumerate neighbors of v
Adjacency matrix	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n)$ row scan
Un sorted adjacency list	$\Theta(n+m)$	$O(\text{degree}(v))$	$\Theta(\text{degree}(v))$
Incidence matrix	$\Theta(nm)$	May need edge scan	Inspect incidence row
Edge list	$\Theta(m)$, plus vertex data	$O(m)$ un sorted	$O(m)$ un sorted

Space excludes weight-size variation. Dense graphs may suit matrix; sparse graphs usually list saves space. Exact complexities change with hash/sorted neighbor containers.

Practice: $n=100$, $m=150$: matrix 10000 entries; undirected adjacency list 300 neighbor records plus 100 heads. Byte counts need element/reference sizes.

UNIT III / CHAPTER 09

अध्याय 9: Breadth First एवं Depth First Traversal

Queue, recursion/stack और visited marks

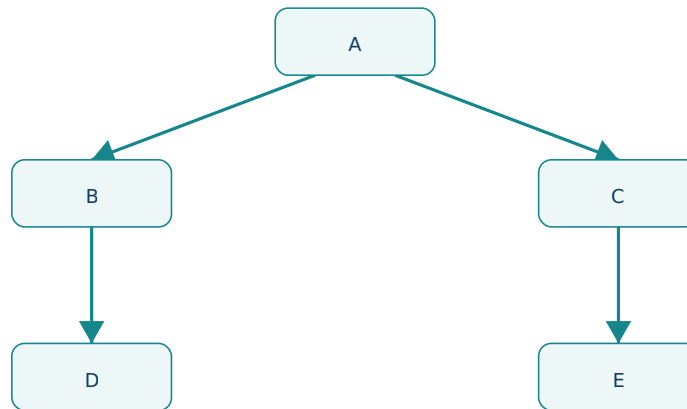


Figure 9.1: Separate practice graph: BFS A B C D E; DFS A B D C E with alphabetical neighbors.

Hindi समझ: BFS queue से स्तर-दर-स्तर आगे बढ़ती है। DFS recursion या stack से एक path पर गहराई तक जाती है और फिर लौटती है। Visited marks cycles में दोबारा processing रोकते हैं।

9.1 Why Visited Set?

Graphs may have cycles and multiple paths. Visited marks prevent repeated processing/infinite revisits. Mark at discovery before enqueueing/recursing. With alphabetical neighbor order, shared graph from A gives BFS A, B, C, D, E and recursive DFS A, B, C, D, E; equal outputs here do not mean algorithms identical.

9.2 BFS Algorithm

```

BFS(G, start)
  visited[start] ← true
  distance[start] ← 0
  parent[start] ← NIL
  Q ← queue containing start
  while Q nonempty
    u ← dequeue(Q)
    visit u
    for v in neighbors(u) in specified order
      if not visited[v]
        visited[v] ← true
        distance[v] ← distance[u]+1
        parent[v] ← u
        enqueue(Q, v)
  
```

Other vertices initially unvisited; distance ∞ , parent NIL. Source must exist. BFS distances count edges, not arbitrary weighted cost. Each reached vertex enqueued once.

9.3 BFS Dry Run

Removed	Newly discovered	Queue after processing
A	B, C	[B, C]
B	D	[C, D]
C	E	[D, E]
D	None	[E]
E	None	[]

BFS parent edges AB, AC, BD, CE. Distances A0, B1, C1, D2, E2. These are minimum edge counts, despite differing weights.

9.4 DFS Algorithm

```

DFS(G, u)
    visited[u] ← true
    visit u
    for v in neighbors(u) in specified order
        if not visited[v]
            parent[v] ← u
            DFS(G, v)

```

Shared graph alphabetical exploration follows A→B→C→D→E, then backtracks. DFS parents AB, BC, CD, DE. Iterative stack version possible; exact order depends on when marked and push order, so state implementation.

9.5 Entire Graph एवं Complexity

One start visits only reachable component. To traverse all vertices: initialize visited once; loop vertices, call BFS/DFS when unvisited. This gives traversal forest. With adjacency list total $\Theta(n+m)$ for entire graph; adjacency matrix $\Theta(n^2)$. Extra visited/parent space $O(n)$; queue or recursion $O(n)$ worst case. Output and input storage excluded.

9.6 Applications एवं Traps

BFS shortest unweighted path, levels. DFS reachability, cycle-analysis foundations, topological methods in DAGs. Deep recursive DFS can exceed callstack. Marking only at dequeue can create duplicate queue entries unless additional discovery state used. Traversal order depends on neighbor order; not universally unique.

Practice: For edges AB, AC, BD, CE, alphabetical from A: BFS A, B, C, D, E; DFS A, B, D, C, E. This demonstrates order difference.

UNIT III / CHAPTER 10

अध्याय 10: Connectivity of Graphs

Components, reachability और direction

Hindi समझ: Undirected connectivity में हर pair के बीच path चाहिए. Directed strong connectivity में path दोनों दिशाओं में उपलब्ध होना चाहिए; केवल एक source से सब तक पहुँच जाना पर्याप्त नहीं है.

10.1 Undirected Connectivity

Connected graph: every pair of vertices has path. Connected component maximal connected vertex set. In undirected graph one BFS/DFS from any vertex reaches all vertices iff nonempty graph connected. Isolated vertex is its own component.

```
COUNT_COMPONENTS(G)
  visited ← false for every vertex
  components ← 0
  for v in all vertices
    if not visited[v]
      components ← components+1
      DFS(G, v)
  return components
```

Adjacency list $\Theta(n+m)$, space $O(n)$. Empty graph component count 0; empty graph “connected” conventions vary, so handle explicitly.

10.2 Example

$V=\{A, B, C, D, E, F\}$; edges AB, BC, DE ; components $\{A, B, C\}, \{D, E\}, \{F\}$:3. A BFS from A reaches 3 vertices, not whole graph. Additional starts D, F create forest.

10.3 Directed Strong एवं Weak Connectivity

Concept	Definition
Strongly connected	Every u can reach every v respecting arcs
Weakly connected	Underlying undirected graph connected
Strongly connected component	Maximal mutually reachable set

$A \rightarrow B \rightarrow C$ is weakly connected, not strongly connected. $A \rightarrow B \rightarrow C \rightarrow A$ is strongly connected. Reaching all vertices from one source does not prove strong connectivity: $A \rightarrow B$ alone counterexample.

10.4 Strong Connectivity Check

For nonempty directed graph: run DFS from s in G , and from s in transpose G^T (arcs reversed). If both reach all vertices, graph strongly connected. First ensures $s \rightarrow v$, second $v \rightarrow s$. Then $u \rightarrow s \rightarrow v$ provides arbitrary reachability. $\Theta(n+m)$ with lists including transpose construction.

Kosaraju/Tarjan find all strongly connected components, but detailed SCC algorithms are additional study, not needed to define connectivity.

10.5 Bridges एवं Articulation Points

Bridge edge removal increases undirected component count. Articulation vertex removal (and incident edges) increases component count relative original graph. In tree with $n > 1$ every edge bridge. In shared weighted graph no edge bridge: each lies on cycle. These concepts evaluate network vulnerability.

10.6 Relationship with Trees

Connected undirected n -vertex graph with $n-1$ edges is a tree. Acyclic undirected graph with c components has $m = n - c$. Disconnected graph cannot have spanning tree covering all vertices using its own edges; may have spanning forest.

Practice: Two disconnected cycles have no bridges but are not connected. Explain why “no bridges” doesn't imply connected.

UNIT III / CHAPTER 11

अध्याय 11: Weighted Graphs एवं Shortest Path

BFS, Dijkstra और weight assumptions

Hindi समझ: Weighted shortest path में total cost कम करना है, केवल edges की संख्या नहीं. Dijkstra के लिए weights nonnegative होने चाहिए. Parent references से actual path वापस बनाया जाता है.

11.1 Path Cost

Weighted path cost = sum edge weights. Shortest path minimizes cost; not necessarily fewest edges. BFS is correct for unweighted graphs or uniform positive edge cost when edge count corresponds to cost. Arbitrary weights require appropriate algorithm.

11.2 Algorithm Selection

Situation	Suitable method
Unweighted/equal positive costs	BFS
Nonnegative weights	Dijkstra
Negative edges permitted	Bellman-Ford, with cycle handling
All-pairs paths	Floyd-Warshall overview

Dijkstra's standard finalization proof requires nonnegative weights. Zero-weight edges allowed. Bellman-Ford can report reachable negative cycles; vertices reachable from such cycle may have no finite shortest cost. In undirected graphs, negative edge enables negative closed walk if repeated edges allowed, so minimum walk cost issue arises.

11.3 Dijkstra: Simple Selection Version

```

DIJKSTRA(G, source)
  for each vertex v
    dist[v] ← infinity
    parent[v] ← NIL
    settled[v] ← false
  dist[source] ← 0
  repeat at most n times
    u ← unsettled vertex with smallest dist
    if no such u or dist[u] = infinity: break
    settled[u] ← true
    for each outgoing edge (u,v) of weight w
      require w ≥ 0
      if not settled[v] and dist[u]+w < dist[v]
        dist[v] ← dist[u]+w
        parent[v] ← u
  return (dist, parent)

```

Input weights validated before algorithm use, rather than discovering negative edge only after partial output. Infinity sentinel and addition must avoid numeric overflow. Missing edge is not zero edge.

11.4 Solved Shared Graph: Source A

Settled vertex	Updated tentative values	Settled distance
A	B2, C4	A0
B	C3, D9	B2
C	D6, E8	C3
D	E7	D6
E	No change	E7

Final distances A0, B2, C3, D6, E7. Shortest A→E path A-B-C-D-E cost 2+1+3+1=7. BFS can find A-C-E in 2 edges, but cost 9. Fewer edges does not mean lower cost.

11.5 Path Reconstruction एवं Complexity

Follow parent from target to source, then reverse. Unreachable vertex $\text{dist} \infty$; report no path. Parent of source NIL. Ties may yield different valid shortest path trees.

Array selection +matrix neighbor scans $\Theta(n^2)$ time, $O(n)$ working state. List +binary heap conventional $O((n+m)\log n)$, with implementation details affecting heap entries; auxiliary space includes heap. Shortest path tree minimizes root-to vertex distances, not total edge weight.

Practice: Triangle AB1, BC1, AC3: from A shortest to C via B cost 2. Aweight-2directed edge violates Dijkstra assumption.

UNIT III / CHAPTER 12

अध्याय 12: Spanning Trees, MST एवं Kruskal's Algorithm

Minimum total connecting cost

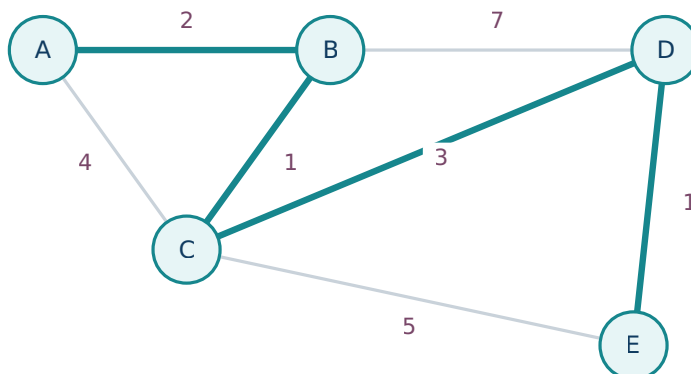


Figure 12.1: Highlighted MST edges AB, BC, CD, DE; total weight 7. Gray edges are excluded.

Hindi समझ: MST सभी vertices को न्यूनतम total edge cost से जोड़ता है और कोई cycle नहीं रखता. Kruskal छोटे-weight edges चुनता है, पर वही edge स्वीकार करता है जो अलग components को जोड़ती हो.

12.1 Spanning Tree

For connected undirected G , spanning tree contains all n vertices, is connected acyclic and has $n-1$ edges. Spanning tree may not be unique. Spanning forest covers components of disconnected graph, edges $n-c$ for c components.

Minimum spanning tree(MST) minimizes **sum of selected edge weights**, not sum source-to vertex distances. Assume undirected connected graph; negative, zero and positive weights all allowed. Directed minimum arborescence is different problem.

12.2 Cut एवं Cycle Principles

Cut partitions vertices into two nonempty sets. A minimum weight edge crossing cut is safe for some MST, subject compatible partial forest. Cycle's strictly heaviest edge cannot belong to any MST. When weights tie, exclusion/uniqueness statements need care: do not discard all arbitrarily tied edges.

Distinct edge weights guarantee unique MST; repeated weights may or may not cause multiple MSTs.

12.3 Kruskal Algorithm

Sort edges nondecreasing weight; accept edge if it joins different components. Disjoint Set Union(DSU) detects components without graph DFS at each step.

```

KRUSKAL(G)
  makeSet(v) for each vertex v
  result ← empty edge list
  for edge (u,v,w) in nondecreasing weight order
    if find(u) ≠ find(v)
      append edge to result
      union(u,v)
      if size(result) = n-1: break
  if n > 0 and size(result) ≠ n-1
    report no spanning tree; result is minimum forest
  return result

```

Sorting ties can use any fixed order; MST cost same, edge choices may vary. Selfloops always rejected because endpoints same component. Parallel edges can be handled as distinct records.

12.4 Solved Shared Graph

Sorted BC1, DE1, AB2, CD3, AC4, CE5, BD7.

Considered	Decision	Reason
BC1	Accept	Join B, C
DE1	Accept	Join D, E
AB2	Accept	Join A with BC component
CD3	Accept	Join ABC and DE components

Four edges selected; **MST total=1+1+2+3=7**. No need process remaining edges once n-1 accepted. If continued, AC, CE, BD form cycles and would be rejected.

12.5 Complexity एवं DSU

Sort $O(m \log m)$. Path compression + union by rank/size gives amortized $O(\alpha(n))$ per DSU operation; total $O(m \log m + m\alpha(n) + n)$. Elementary union implementations may be slower. DSU stores parent/rank arrays $O(n)$; edge list storage $O(m)$.

Practice: Triangle edges AB1, BC2, AC5 → MST AB, BC, total 3. All three edges weight 1 → multiple MSTs, total 2.

UNIT III / CHAPTER 13

अध्याय 13: Prim's Algorithm एवं MST Comparison

One growing tree और cheapest crossing edge

Hindi समझ: Prim current tree से बाहर जाने वाली सबसे सस्ती edge चुनता है. उसकी key एक connecting edge का cost है; Dijkstra की distance source से पूरा path cost है. दोनों values को mix न करें.

13.1 Prim's Idea

Start any vertex; repeatedly add cheapest edge connecting current tree to outside vertex. Unlike Dijkstra, $key[v]$ is cost of one edge connecting v to tree, not source path distance. Undirected connected weighted graph assumed. Negative weights allowed for MST.

```

PRIM(G, start)
  key[v] ← infinity, parent[v] ← NIL,
  inTree[v] ← false for each vertex v
  key[start] ← 0
  repeat n times
    u ← outside vertex with smallest key
    if no such u or key[u] = infinity
      report graph disconnected
    inTree[u] ← true
    for edge (u,v,w)
      if not inTree[v] and w < key[v]
        key[v] ← w
        parent[v] ← u
  return edges(parent[v], v) for v ≠ start

```

Start0key is artificial roots election marker, not a graph self edge. Minimum selection and ties require consistent policy. For forest, restart unvisited component explicitly instead of claiming one MST.

13.2 Solved Shared Graph: Start A

Selected vertex	Added edge	New/remaining keys
A	None	B2, C4
B	AB2	C1, D7
C	BC1	D3, E5
D	CD3	E1
E	DE1	Done

MST edges AB2, BC1, CD3, DE1, total 7. Notice $key[E]=1$ after D, while Dijkstra shortest distance[E]=7. Prim does not accumulate path sum during key relaxation.

13.3 Prim versus Kruskal

Aspect	Prim	Kruskal
Growth	One tree per connected component	Forest components merge
Choice	Minimum crossing edge	Globally sorted edge that doesn't form a cycle
Main helper	Keys +minpriority selection	Sorting +DSU
Matrix version	$\Theta(n^2)$	Usually uses edge list
Heap/list version	$O((n+m)\log n)$	$O(m \log m)$ dominant

Dense matrix graphs often use simple Prim. Sparse edge lists suit Kruskal. Both greedy MST algorithms; neither unconditionally replaces other for every implementation.

13.4 MST versus Shortest Path Tree

Triangle AB2, AC2, BC1. Prim MST may AB+BC, total 3. Dijkstra from A selects AB+AC for distances 2, 2, total selected cost 4. MST minimizes network total; shortest path tree minimizes distance from source. Shared main example happens to yield same edge path for both, which is not general property.

13.5 Correctness Intuition

At each step cut=(in tree, outside); chosen cheapest crossing edge is safe by cut principle. Adding exactly one outside vertex cannot create cycle. After all n vertices, $n-1$ parent edges give connected tree.

Practice: Starting vertex and tie order may change chosen MST if multiple optima, but minimum total cost unchanged. For negative edge AB-2, BC1, AC4, MST cost -1; Dijkstra restrictions do not apply to Prim.

UNIT III / CHAPTER 14

अध्याय 14: Complete Revision एवं Unit Test

Trees-Graphs concepts, numericals और algorithms

Hindi समझ: Exam में definition के साथ diagram, algorithm, dry run और complexity लिखें. Numerical से पहले height/index convention और graph algorithm से पहले weight/connectivity assumptions बताएँ.

14.1 Formula Sheet

Concept	Formula / condition
Nonempty tree	$m=n-1$
Binary depth d maximum	2^d nodes
Perfect binary height h edges	$2^{h+1}-1$ nodes, 2^h leaves
Binary leaf relation	$n_0=n_2+1$
Zero-based tree children	$2i+1, 2i+2$
AVL balance	$\text{height}(\text{left})-\text{height}(\text{right}) \in \{-1, 0, 1\}$
Simple undirected max edges	$n(n-1)/2$
Undirected degrees sum	$2m$, loop counting correctly
Directed degrees sums	$\sum \text{in} = \sum \text{out} = m$
Forest c components	$m=n-c$ when acyclic
MST connected n vertices	$n-1$ selected edges

14.2 Algorithm Selection

Task	Method / assumption
Binary tree inorder	Left, Root, Right
Ordered key search	BST/AVL, explicit duplicate policy
Reachability	BFS/DFS
Unweighted shortest path	BFS
Nonnegative weighted shortest	Dijkstra
Negative edge SSSP	Bellman-Ford with cycle checks
Minimum connecting cost	Prim/Kruskal, undirected graph

14.3 Solved Mixed Questions

Q1: Perfect binary height 5 → 63 nodes, 32 leaves.

Q2: Binary tree with 12 two child nodes → 13 leaves, regardless one child count.

Q3: Full tree with 13 leaves → 12 internal, 25 total nodes.

Q4: $K_7 \rightarrow 21$ edges; degree each 6, total degrees 42.

Q5: Forest 10 vertices, 3 components $\rightarrow 7$ edges.

Q6: Shared graph $A \rightarrow E$ minimum edge count 2 but shortest weighted cost 7; MST total 7. These 7s coincide by example, not definition.

14.4 Unit Test: 40 MCQs

One best answer. Height counts edges, root depth 0; simple graphs unless stated otherwise.

1. Nonempty tree with n vertices has: A. $n-1$ edges B. n edges always C. n^2 edges D. No edges always
2. Binary node has at most: A. 0 children B. 2 children C. 3 children D. n children necessarily
3. Leaf height in this book: A. -1 B. n C. 0 D. 1
4. Full binary tree every node has: A. Exactly 1 child B. Exactly 2 always C. 3 children D. 0 or 2 children
5. Complete matrix-style tree last level fills: A. Left to right B. Right to left always C. Randomly D. Only middle
6. Perfect tree height 3 total nodes: A. 16 B. 15 C. 8 D. 7
7. Nonempty binary tree leaf relation: A. $n_0 = n_1$ always B. $n_0 = 0$ C. $n_0 = n_2 + 1$ D. $n_0 = n_2 - 1$
8. Zero-based index 3 left child: A. 6 B. 8 C. 1 D. 7
9. Preorder: A. Root, Left, Right B. Left, Root, Right C. Left, Right, Root D. Right, Left, Root always
10. Postorder processes root: A. Only at level 0 B. After children C. First always D. Never
11. BST inorder unique keys returns: A. BFS order B. Random order C. Sorted ascending D. Always descending
12. Ordinary BST worst height: A. $\log n$ always B. 0 always C. n^2 D. $n-1$
13. BST two-child deletion can use successor: A. Minimum in right subtree B. Maximum in right always C. Any key D. Root parent always
14. Generic unordered tree search worst: A. Impossible B. $\Theta(n)$ C. $\Theta(1)$ always D. $\Theta(\log n)$ always
15. AVL balance factor allowed: A. Any integer B. Always 0 C. -1, 0, 1 D. Only 2
16. LL imbalance fixed by: A. Left rotation always B. Node free only C. BFS D. Right rotation
17. LR repair: A. Left at child then right at node B. Two left always C. Two right always D. None
18. Rotation preserves: A. Root identity always B. Inorder key sequence C. Exact old shape D. Every old height
19. Graph representation $G=(V, E)$: A. Vectors only B. Memory slots C. Vertices and edges D. Values and exponents
20. K_5 undirected edges: A. 20 B. 5 C. 25 D. 10
21. Undirected degrees sum: A. $2m$ B. m C. n D. $m/2$
22. Directed indegrees sum: A. Always 0 B. m C. $2m$ D. n^2
23. Undirected adjacency matrix: A. Always sparse B. Not matrix C. Symmetric D. Always diagonal ones
24. Incidence matrix rows/columns: A. Edges/vertices in this book B. Paths/cycles C. Weights/roots D. Vertices/edges
25. Loopless undirected incidence edge column has: A. Two 1s B. One 1 C. All 1s D. All 0s
26. Adjacency list space: A. $\Theta(1)$ B. $\Theta(n+m)$ C. $\Theta(n^2)$ always D. $\Theta(nm)$ always
27. BFS uses: A. Sorting mandatory B. DSU only C. Queue D. Only recursion
28. DFS commonly uses: A. FIFO only B. Incidence weights C. MST only D. Recursion/stack
29. BFS optimal path guarantee is for: A. Unweighted edge counts B. Arbitrary negative weights C. All weighted costs D. Longest paths
30. Single BFS on disconnected graph visits: A. Only isolated nodes B. Reachable component C. All vertices always D. No vertices
31. $A \rightarrow B \rightarrow C$ is: A. Undirected tree by orientation rule B. Complete C. Weakly connected not strong D. Strongly connected
32. Strong connectivity check uses G and: A. Same graph only always B. MST C. Stack capacity D. Transpose G

33. Standard Dijkstra needs: A. Nonnegative weights B. Negative weights mandatory C. Equal weights mandatory D. No edges
34. Relax Dijkstra edge(u, v, w) with: A. key root only B. $\text{dist}[u] + w$ C. w alone always D. $\text{degree}[u] + w$
35. MST minimizes: A. Number vertices B. Graph degree C. Selected edge weight sum D. Every root path distance
36. Connected MST n vertices edges: A. n B. n^2 C. $2n$ D. $n-1$
37. Kruskal cycle helper: A. DSU B. Recursion factorial C. Array stack only D. Inorder
38. Prim's $\text{key}[v]$ represents: A. BFS level B. Cheapest connecting to current tree C. Always source path distance D. Degree
39. Negative MST weights: A. Dijkstra mandatory B. Make cycle mandatory C. Allowed D. Always invalid
40. Distinct edge weights guarantee: A. Unique shortest path always B. Perfect tree C. Graph disconnected D. Unique MST for connected undirected graph

उत्तर-कुंजी:

1-A, 2-B, 3-C, 4-D, 5-A, 6-B, 7-C, 8-D, 9-A, 10-B, 11-C, 12-D, 13-A, 14-B, 15-C, 16-D, 17-A, 18-B, 19-C, 20-D, 21-A, 22-B, 23-C, 24-D, 25-A, 26-B, 27-C, 28-D, 29-A, 30-B, 31-C, 32-D, 33-A, 34-B, 35-C, 36-D, 37-A, 38-B, 39-C, 40-D.

14.5 Written Exam Practice

5 marks: Tree terminology; full versus complete; BST invariant; AVL rotation types; adjacency versus incidence; strong versus weak connectivity; shortest path tree versus MST.

10 marks:

1. Binary tree properties with derivation, numericals and representation.
2. Preorder, inorder, postorder algorithms with shared tree outputs.
3. BST search, insertion, deletion: all three cases and complexity.
4. AVL rotations: diagrams and height updates.
5. Graph matrix/list representations with space comparison.
6. BFS/DFS algorithms, dryruns and disconnected graph handling.
7. Dijkstra's algorithm with weight assumptions and solved table.
8. Prim and Kruskal with MST cost, cut/cycle intuition and comparison.

14.6 Practical Projects एवं Boundary Tests

Tree lab: Insert 50, 30, 70, 20, 40, 60, 80; print traversals; search present/absent; delete leaf, one child, two children and root. Validate inorder strict sorted after every operation. Repeat with AVL and check all balance factors/heights.

Graph lab: Build shared graph from edge list. Generate adjacency and incidence representations. Run BFS, DFS, Dijkstra source A, Prim source A, Kruskal. Verify distances[0, 2, 3, 6, 7] and MST total 7. Add isolated F: traversals need new start, Dijkstra $\text{dist } F = \infty$, full MST absent, minimum spanning forest still valid.

Boundary checklist: Empty/single tree; duplicate BST keys; skew tree; all four AVL cases; isolated vertex; disconnected graph; zero edge weight; negative mst edge; equal weights; invalid negative Dijkstra input. Track conventions and assumptions along with answers.