CS402 ADA – Unit 5 NOTES

Binary Search Tree (BST)

(RGPV Exam-Oriented Notes | Easy Hinglish)

A. Introduction

What is BST?

BST ek special type ka Binary Tree hai jisme data specific order mein store hota hai.

Iski wajah se searching bahut fast ho jati hai.

Why Do We Need BST?

Normal array mein searching:

```
10 20 30 40 50
```

Worst case:

```
O(n)
```

BST mein searching:

```
O(log n)
```

(average case)

Isliye BST efficient hai.

Real-Life Analogy

Suppose school library mein books arranged hain:

```
A-M → Left Side

N-Z → Right Side
```

Agar "Python" book chahiye:

To poori library search nahi karni padegi.

BST bhi exactly yehi karta hai.

B. Definition

Exam Definition

A Binary Search Tree (BST) is a binary tree in which every node satisfies:

- Left subtree contains smaller values.
- Right subtree contains larger values.

Easy Explanation

Rule:

```
Left < Root < Right
```

Har node ke liye ye condition true honi chahiye.

C. Core Concept

Important Keywords

Node

Tree ka basic element.

```
[50]
```

Root Node

Topmost node.

```
50
```

Parent Node

Jiske children hote hain.

Child Node

Parent ke niche wala node.

Leaf Node

Jiska koi child nahi hota.

BST Property

Most Important Rule

```
Left Subtree < Root

Right Subtree > Root
```

Example:

```
        50
       /  \
     30    70
    / \   / \
   20 40 60 80
```

Check:

```
20 < 30 < 40

30 < 50 < 70

60 < 70 < 80
```

BST Valid.

D. Structure / Diagram

Example BST

```
              50
            /    \
          30      70
         /  \    /  \
       20   40 60   80
```

Labels

```
Root = 50

Parent = 30

Children = 20,40

Leaf Nodes = 20,40,60,80
```

E. Working

---

1. Insertion in BST

Insert:

```
50,30,70,20,40,60,80
```

Step 1

Insert 50

```
50
```

Step 2

Insert 30

```
30 < 50
```

Go Left

```
   50
  /
30
```

Step 3

Insert 70

```
70 > 50
```

Go Right

```
    50
   /  \
 30    70
```

Step 4

Insert 20

```
20 < 50

20 < 30
```

```
      50
     /  \
   30    70
  /
20
```

Continue similarly.

Final BST:

```
              50
            /    \
          30      70
         /  \    /  \
       20   40 60   80
```

BST Insertion Algorithm

Pseudocode

```
Insert(root,key)
```

```
if root == NULL

   create node

else if key < root

   insert left

else

   insert right
```

---

2. Searching in BST

Find:

```
60
```

Step 1

Start:

```
50
```

```
60 > 50
```

Go Right

Step 2

Current:

```
70
```

```
60 < 70
```

Go Left

Step 3

Current:

```
60
```

Found.

Search Algorithm

```
Search(root,key)

if root == NULL

   return NOT FOUND

if key == root

   FOUND

if key < root

   search left

else

   search right
```

---

3. Deletion in BST

Most Important Exam Topic

Case 1: Leaf Node

Delete:

```
20
```

```
20
```

Simply remove.

Case 2: One Child

Example:

```
50
 \
 70
  \
 80
```

Delete:

```
70
```

Connect:

```
50 → 80
```

Case 3: Two Children

Delete:

```
50
```

Find:

```
Inorder Successor
```

Replace root.

Then delete successor.

F. Complete Dry Run

Insert:

```
50 30 70 20 40
```

After 50

```
50
```

After 30

```
 50
 /
30
```

After 70

```
   50
  /  \
30    70
```

After 20

```
      50
     /  \
   30    70
  /
20
```

After 40

```
      50
     /  \
   30    70
  / \
 20 40
```

Final BST Created.

G. Complexity Analysis

Let:

```
n = number of nodes
```

Searching

Best Case:

O(1)

Average Case:

O(logn)

Worst Case:

Skew Tree

```
10
 \
 20
  \
  30
```

Complexity:

O(n)

Insertion

Average:

O(logn)

Worst:

O(n)

Deletion

Average:

O(logn)

Worst:

O(n)

Space Complexity

Recursive:

O(h)

where h = height.

H. Advantages

1. Fast Searching
2. Fast Insertion

3. Dynamic Structure
4. Ordered Data
5. Efficient Retrieval

I. Disadvantages

1. Skew Tree Problem
2. Worst Case O(n)
3. Extra Memory for Pointers
4. Balancing Required

J. Applications

- Database Indexing
- Dictionary Implementation
- File Systems
- Symbol Tables
- Searching Applications

K. Common Mistakes

❌ Left > Root likh dena

✔ Always:

```
Left < Root < Right
```

❌ Deletion cases bhool jana

✔ Three cases yaad rakho.

❌ Traversal order mix karna

✔ Inorder = Sorted Output

L. Viva Questions

What is BST?

A binary tree satisfying:

```
Left < Root < Right
```

Why BST is efficient?

Fast searching.

Complexity of BST Search?

Average:

O(logn)

What is Inorder Traversal?

Left → Root → Right

What is Root Node?

Topmost node.

M. Exam Keywords

Write these words:

- Binary Search Tree
- Left Subtree
- Right Subtree
- Root Node
- Leaf Node
- Search
- Insert
- Delete
- Inorder Successor
- Ordered Tree

N. Memory Tricks

BST Rule

```
L < R < R
```

Meaning:

```
Left < Root < Right
```

Deletion Cases

Mnemonic:

LOT

Leaf

One Child

Two Children

O. Comparison Table

BST vs Binary Tree

Feature	BST	Binary Tree
Ordering	Yes	No
Searching	Fast	Slow
Rule	Left < Root < Right	No Rule
Complexity	O(log n) Avg	O(n)

BST vs Array

Feature	BST	Array
Insert	Easy	Costly
Delete	Easy	Costly
Search	Fast	Linear
Dynamic	Yes	No

P. RGPV Exam Answers

2 Mark Answer

A Binary Search Tree (BST) is a binary tree in which all nodes in the left subtree are smaller than the root and all nodes in the right subtree are larger than the root.

---

5 Mark Answer

BST is a special binary tree following:

```
Left < Root < Right
```

Operations:

- Search
- Insert
- Delete

Advantages:

- Fast searching
- Ordered storage

Complexity:

Average:

O(logn)

---

7 Mark Answer

Binary Search Tree

BST is a binary tree where:

```
Left < Root < Right
```

Diagram:

```
     50
    /  \
  30    70
```

Operations:

1. Insertion
2. Searching
3. Deletion

Average Complexity:

O(logn)

Applications:

- Databases
- Dictionaries
- File Systems

---

10 Mark Topper Answer

Binary Search Tree (BST)

Definition

A BST is a binary tree in which the left subtree contains smaller values and the right subtree contains larger values.

Structure

```
          50
        /    \
      30      70
     / \     / \
   20 40   60 80
```

Operations

- Search
- Insert
- Delete

Algorithms

(Search, Insert, Delete pseudocode)

Complexity

Operation	Average	Worst
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Applications

- Database Indexing
- Symbol Tables
- Searching Systems

Conclusion

BST provides efficient searching, insertion, and deletion while maintaining ordered data.

Q. One Page Revision Sheet

BST

Rule:

```
Left < Root < Right
```

Diagram:

```
     50
    /  \
  30    70
```

Operations:

- Search
- Insert
- Delete

Deletion Cases:

1. Leaf
2. One Child
3. Two Children

Traversal:

- Inorder = LNR
- Preorder = NLR
- Postorder = LRN

Complexity:

Operation	**Average**
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

Memory Trick:

LOT

Leaf

One Child

Two Children

🔥 **Most Important RGPV Question:**
"Explain Binary Search Tree with insertion, deletion, traversal and complexity analysis." (7–10

Marks)

Tree Traversals (Inorder, Preorder, Postorder)

(RGPV Exam-Oriented Notes | Easy Hinglish)

A. Introduction

Tree Traversal ka matlab hai:

👉 Tree ke sabhi nodes ko ek specific order mein visit karna.

Jab hume tree ke data ko access, print ya process karna hota hai tab traversal use hota hai.

Why Do We Need Traversal?

Suppose tree hai:

```
    A
   / \
  B   C
 / \
D   E
```

Agar hume sabhi nodes print karne hain to kisi order mein visit karna padega.

Yehi Tree Traversal hai.

Real-Life Analogy

Family Tree dekhte waqt:

- Pehle Parent dekho
- Phir Children dekho

Ya

- Pehle Left Child
- Phir Parent
- Phir Right Child

Different visiting orders = Different Traversals

B. Definition

Exam Definition

Tree Traversal is the process of visiting every node of a tree exactly once in a specific order.

Easy Explanation

```
Tree ke har node ko
ek fixed sequence mein
visit karna
=
Traversal
```

C. Types of Tree Traversal

RGPV mein 3 Traversals sabse important hain:

1. Inorder Traversal (LNR)

```
Left
↓
Node
↓
Right
```

---

2. Preorder Traversal (NLR)

```
Node
↓
Left
↓
Right
```

---

3. Postorder Traversal (LRN)

```
Left
↓
Right
↓
Node
```

D. Example Tree

Hum isi tree par saare traversals karenge.

```
        A
      /   \
     B     C
    / \   / \
   D   E F   G
```

---

1. Inorder Traversal (LNR)

Rule

```
Left
Root
Right
```

Step-by-Step

Tree:

```
        A
      /   \
     B     C
    / \   / \
   D   E F   G
```

Left Subtree of A

Visit:

```
D → B → E
```

Visit Root

```
A
```

Right Subtree

Visit:

```
F → C → G
```

Final Inorder

```
D B E A F C G
```

Algorithm

```
Inorder(root)

if root != NULL

   Inorder(root->left)

   Visit(root)

   Inorder(root->right)
```

Memory Trick

LNR

```
Left
Node
Right
```

Important Point

For BST:

🔥 Inorder Traversal always gives sorted output.

---

2. Preorder Traversal (NLR)

Rule

```
Node
Left
Right
```

Step-by-Step

Root:

```
A
```

Left subtree:

```
B D E
```

Right subtree:

```
C F G
```

Final Preorder

```
A B D E C F G
```

Algorithm

```
Preorder(root)

if root != NULL

   Visit(root)

   Preorder(root->left)

   Preorder(root->right)
```

Memory Trick

NLR

```
Node
Left
Right
```

Use

Tree copy creation.

---

3. Postorder Traversal (LRN)

Rule

```
Left
Right
Node
```

Step-by-Step

Left subtree:

```
D E B
```

Right subtree:

```
F G C
```

Root:

```
A
```

Final Postorder

```
D E B F G C A
```

Algorithm

```
Postorder(root)

if root != NULL

   Postorder(root->left)

   Postorder(root->right)

   Visit(root)
```

Memory Trick

LRN

```
Left
Right
Node
```

Use

Tree deletion.

E. Complete Dry Run

Tree:

```
          50
        /    \
      30      70
     / \     / \
   20 40   60 80
```

Inorder

```
20 30 40 50 60 70 80
```

Preorder

```
50 30 20 40 70 60 80
```

Postorder

```
20 40 30 60 80 70 50
```

F. Complexity Analysis

Let:

```
n = number of nodes
```

Time Complexity

Each node visited once.

O(n)

Space Complexity

Recursion stack:

O(h)

where h = tree height.

Best Case

O(n)

Average Case

O(n)

Worst Case

O(n)

Why O(n)?

Har node exactly ek baar visit hota hai.

G. Advantages

Inorder

- BST ko sorted form mein print karta hai.

Preorder

- Tree copy banane mein useful.

Postorder

- Tree delete karne mein useful.

H. Disadvantages

1. Recursive calls memory use karti hain.
2. Large trees mein stack overflow ho sakta hai.

I. Applications

Inorder

- BST Sorting
- Database Queries

Preorder

- Expression Tree Copy
- Serialization

Postorder

- Tree Deletion
- Expression Evaluation

J. Common Mistakes

❌ Inorder aur Preorder mix kar dena.

✔ LNR vs NLR yaad rakho.

❌ Root position galat likhna.

✔ Traversal name se root ki position identify karo.

❌ BST ka sorted output bhool jana.

✔ Inorder = Sorted Output.

K. Viva Questions

What is Tree Traversal?

Visiting all nodes exactly once.

Types of Traversal?

- Inorder
- Preorder
- Postorder

Which traversal gives sorted BST?

Inorder Traversal.

Complexity?

O(n)

Which traversal is used for deletion?

Postorder.

L. Exam Keywords

Write these keywords:

- Tree Traversal
- Inorder
- Preorder
- Postorder
- Recursive Traversal
- Visit Node
- Left Subtree
- Right Subtree
- Depth First Traversal
- BST Sorted Output

M. Memory Tricks

Inorder

LNR

```
Left
Node
Right
```

Preorder

NLR

```
Node
Left
Right
```

Postorder

LRN

```
Left
Right
Node
```

Super Trick

```
IN  = Root Middle

PRE = Root First

POST = Root Last
```

🔥 Exam mein kabhi nahi bhoologe.

N. Comparison Table

Traversal	Order	Root Position
Inorder	LNR	Middle
Preorder	NLR	First
Postorder	LRN	Last

Uses Comparison

Traversal	Use
Inorder	Sorted Output
Preorder	Tree Copy
Postorder	Tree Deletion

O. RGPV Exam Answers

2 Mark Answer

Tree Traversal is the process of visiting every node of a tree exactly once in a specific order.

Types are Inorder, Preorder, and Postorder.

---

5 Mark Answer

Tree Traversals

1. Inorder (LNR)
2. Preorder (NLR)
3. Postorder (LRN)

Example:

```
    A
   / \
  B   C
```

Inorder:

```
B A C
```

Preorder:

```
A B C
```

Postorder:

```
B C A
```

Complexity:

O(n)

---

7 Mark Answer

Explain Tree Traversals

Tree Traversal means visiting all nodes of a tree exactly once.

Types:

Inorder

Left → Root → Right

Preorder

Root → Left → Right

Postorder

Left → Right → Root

Algorithms:

```
Inorder:
LNR

Preorder:
NLR
```

```
Postorder:
LRN
```

Time Complexity:

O(n)

Applications:

- BST Sorting
- Tree Copy
- Tree Deletion

---

10 Mark Topper Answer

Tree Traversals

Definition

Tree Traversal is the process of visiting each node exactly once in a particular order.

Types

1. Inorder (LNR)
2. Preorder (NLR)
3. Postorder (LRN)

Example Tree:

```
        A
      /   \
     B     C
    / \   / \
   D  E  F  G
```

Results:

```
Inorder:
D B E A F C G

Preorder:
A B D E C F G

Postorder:
D E B F G C A
```

Complexity:

Time:

O(n)

Space:

O(h)

Applications:

- BST Traversal
- Expression Trees
- Tree Deletion

Conclusion

Tree Traversals are fundamental techniques used to process all nodes of a tree systematically.

P. One Page Revision Sheet

Tree Traversals

Inorder

```
L N R
```

Output Sorted in BST

Preorder

```
N L R
```

Root First

Postorder

```
L R N
```

Root Last

Example

```
    A
   / \
  B   C
```

Inorder:

```
B A C
```

Preorder:

```
A B C
```

Postorder:

```
B C A
```

Complexity

Time:

O(n)

Space:

O(h)

Memory Trick

IN → Root Middle

PRE → Root First

POST → Root Last

Depth First Search (DFS)

(RGPV Exam-Oriented Notes | Easy Hinglish)

A. Introduction

What is DFS?

DFS (Depth First Search) ek graph traversal technique hai.

Isme hum graph ke ek node se start karte hain aur **jitna deep (depth) ja sakte hain utna jaate hain**, phir backtrack karke doosre nodes visit karte hain.

Why Do We Need DFS?

Graph ke sabhi vertices visit karne ke liye.

Uses:

- Path Finding
- Cycle Detection
- Topological Sorting
- Connected Components

Real-Life Analogy

Maze mein ghusne par:

```
Ek rasta pakdo
↓
End tak jao
↓
Dead End?
↓
Wapas aao
↓
Doosra rasta try karo
```

Yehi DFS hai.

B. Definition

Exam Definition

Depth First Search (DFS) is a graph traversal algorithm that explores a graph by going as deep as possible along each branch before backtracking.

Easy Explanation

```
Deep Jao
↓
Backtrack Karo
```

```
↓
Next Path Explore Karo
```

C. Core Concept

DFS uses:

Stack

ya

Recursion

Important Keywords

Vertex

Graph ka node.

```
A, B, C
```

Edge

Connection between vertices.

```
A ---- B
```

Visited Array

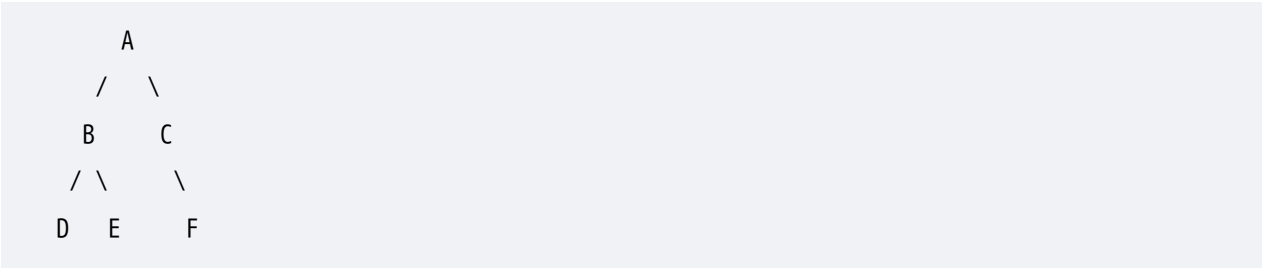
Track karta hai kaunsa node visit ho chuka hai.

Backtracking

Deep jaane ke baad previous node par return karna.

D. Graph Example

Graph:



Adjacency List:

```
A → B,C

B → D,E

C → F
```

E. Working

Step-by-Step Procedure

Step 1

Start node choose karo.

Suppose:

```
A
```

Step 2

Visit A.

Step 3

First unvisited neighbor visit karo.

```
B
```

Step 4

Again B ka first unvisited neighbor.

```
D
```

Step 5

D ke neighbors nahi.

Backtrack to B.

Step 6

Visit E.

Step 7

Backtrack to A.

Step 8

Visit C.

Step 9

Visit F.

DFS Traversal Order

```
A → B → D → E → C → F
```

DFS Diagram

```
       A
     /   \
    B     C
   / \     \
  D   E     F
```

Traversal:

```
A
↓
B
↓
D
↑
B
↓
E
↑
A
↓
C
↓
F
```

F. DFS Algorithm

Recursive Pseudocode

```
DFS(v)

mark v visited

print(v)

for each adjacent node u

   if u not visited

      DFS(u)
```

Simple Algorithm

```
1. Visit node
2. Mark visited
3. Go to first unvisited neighbor
4. Repeat
5. Backtrack when stuck
```

G. Complete Dry Run

Graph:

```
       A
     /   \
    B     C
   / \     \
  D   E     F
```

Initially

Visited:

```
{}
```

Visit A

```
{A}
```

Visit B

```
{A,B}
```

Visit D

```
{A,B,D}
```

D complete.

Backtrack.

Visit E

```
{A,B,D,E}
```

Backtrack.

Visit C

```
{A,B,D,E,C}
```

Visit F

```
{A,B,D,E,C,F}
```

Final DFS

```
A B D E C F
```

H. Complexity Analysis

Let:

```
V = Vertices

E = Edges
```

Time Complexity

Each vertex visited once.

Each edge explored once.

O(V+E)

Space Complexity

Visited Array + Stack

O(V)

Best Case

O(V+E)

Average Case

O(V+E)

Worst Case

O(V+E)

Why O(V+E)?

Har vertex ek baar visit hota hai.

Har edge ek baar check hoti hai.

I. Advantages

1. Simple implementation.
2. Less memory than BFS.
3. Useful for path finding.
4. Cycle detection possible.
5. Recursive solution easy.

J. Disadvantages

1. Deep recursion possible.
2. Stack overflow risk.
3. Shortest path guarantee nahi.

K. Applications

Path Finding

Cycle Detection

Topological Sorting

Connected Components

Maze Solving

AI Search Problems

L. Common Mistakes

❌ DFS aur BFS mix kar dena.

✔ DFS uses Stack/Recursion.

✔ BFS uses Queue.

❌ Visited array mention nahi karna.

✔ Always mention.

❌ Backtracking explain nahi karna.

✔ DFS ki core idea hai.

M. Viva Questions

What is DFS?

Depth First Search graph traversal algorithm.

Which data structure is used?

Stack.

Complexity?

O(V+E)

DFS uses recursion?

Yes.

Applications?

Cycle detection, path finding, topological sorting.

N. Exam Keywords

Write these keywords:

- Depth First Search
- Graph Traversal
- Stack
- Recursion
- Backtracking
- Visited Array

- Vertex
- Edge
- Connected Components
- Cycle Detection

O. Memory Tricks

DFS Rule

```
Deep
↓
Back
↓
Next
```

Mnemonic:

DBN

Deep

Back

Next

DFS Data Structure

```
DFS → Stack
```

Remember:

```
D = Deep

S = Stack
```

P. Comparison Table

DFS vs BFS

Feature	DFS	BFS
Data Structure	Stack	Queue
Traversal	Deep First	Level Wise
Memory	Less	More
Shortest Path	No	Yes
Technique	Backtracking	Breadth Expansion

DFS Recursive vs Iterative

Feature	Recursive	Iterative
Uses	Recursion	Stack
Simplicity	Easy	Moderate

Q. RGPV Exam Answers

2 Mark Answer

Depth First Search (DFS) is a graph traversal algorithm that explores a graph by going as deep as possible before backtracking.

---

5 Mark Answer

DFS is a graph traversal technique.

Steps:

1. Visit node.
2. Mark visited.
3. Visit unvisited neighbor.
4. Backtrack if necessary.

Uses:

- Cycle Detection
- Path Finding

Complexity:

O(V+E)

---

7 Mark Answer

Explain DFS Algorithm

DFS explores graph depth-wise using recursion or stack.

Algorithm:

```
DFS(v)

Visit(v)

for each neighbor

   if not visited

      DFS(neighbor)
```

Example Traversal:

```
A B D E C F
```

Complexity:

Time:

O(V+E)

Space:

O(V)

Applications:

- Path Finding
- Topological Sorting

---

10 Mark Topper Answer

Depth First Search (DFS)

Definition

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.

Algorithm

```
DFS(v)

mark visited

for each adjacent vertex

   if not visited

      DFS(vertex)
```

Diagram

```
       A
     /   \
    B     C
```

```
  / \    \
 D   E    F
```

Traversal

```
A B D E C F
```

Complexity

Time:

O(V+E)

Space:

O(V)

Applications

- Cycle Detection
- Path Finding
- Connected Components

Conclusion

DFS is an efficient graph traversal technique based on recursion and backtracking.

R. One Page Revision Sheet

DFS

Definition

Graph traversal using depth-first approach.

Data Structure

```
Stack
```

Working

```
Visit
↓
Go Deep
↓
Backtrack
↓
Next Node
```

Traversal Example

```
A B D E C F
```

Complexity

Time:

O(V+E)

Space:

O(V)

Applications

- Path Finding
- Cycle Detection
- Topological Sorting

Memory Trick

DBN

Deep

Back

Next

🔥 **Most Important RGPV Question:**
"Explain Depth First Search (DFS) with algorithm, example graph, traversal order and complexity analysis." (7 Marks

🔥 **Most Important RGPV Question:**
"Explain Inorder, Preorder and Postorder Traversals with example tree, algorithms and complexity analysis." (7 Marks)

Breadth First Search (BFS)

(RGPV Exam-Oriented Notes | Easy Hinglish)

A. Introduction

What is BFS?

BFS (Breadth First Search) ek graph traversal algorithm hai.

DFS ki tarah deep nahi jaata.

BFS pehle current level ke sabhi nodes visit karta hai, phir next level par move karta hai.

Why Do We Need BFS?

Jab hume:

- Shortest Path find karna ho
- Level-wise traversal karna ho
- Network routing karna ho

Tab BFS use karte hain.

Real-Life Analogy

Suppose tum ek building mein ho.

Agar kisi person ko dhoondhna hai:

DFS

Ek floor complete check karega.

BFS

Har floor ka first room check karega.

Level by level search karega.

B. Definition

Exam Definition

Breadth First Search (BFS) is a graph traversal algorithm that explores all neighboring vertices first before moving to the next level.

Easy Explanation

```
Current Level
↓
Next Level
↓
Next Level
```

Yani:

```
Level Wise Traversal
```

C. Core Concept

BFS uses:

Queue

Queue follows:

```
FIFO
```

First In First Out

Important Keywords

Vertex

Graph ka node.

Example:

```
A, B, C
```

Edge

Connection between vertices.

```
A ----- B
```

Queue

Nodes ko temporarily store karta hai.

Visited Array

Track karta hai kaunsa node visit ho chuka hai.

D. Graph Example

Graph:

```
        A
      /   \
     B     C
    / \   / \
   D   E F   G
```

Adjacency List:

```
A → B,C

B → D,E

C → F,G
```

E. Working

Step-by-Step Procedure

Step 1

Start node choose karo.

```
A
```

Step 2

Visit A.

Queue:

```
[A]
```

Step 3

Remove A.

Visit neighbors:

```
B,C
```

Queue:

```
[B,C]
```

Step 4

Remove B.

Visit:

```
D,E
```

Queue:

```
[C,D,E]
```

Step 5

Remove C.

Visit:

```
F,G
```

Queue:

```
[D,E,F,G]
```

Step 6

Continue until queue empty.

BFS Traversal Order

```
A B C D E F G
```

Level-wise View

```
Level 0

A

Level 1

B C

Level 2
```

```
D E F G
```

F. Queue Dry Run

Graph:

```
        A
      /   \
     B     C
    / \   / \
   D   E F   G
```

Initially

Queue:

```
[A]
```

Visited:

```
A
```

Remove A

Add:

```
B,C
```

Queue:

```
[B,C]
```

Remove B

Add:

```
D,E
```

Queue:

```
[C,D,E]
```

Remove C

Add:

```
F,G
```

Queue:

```
[D,E,F,G]
```

Final Traversal

```
A B C D E F G
```

G. BFS Algorithm

Pseudocode

```
BFS(start)

Create Queue

Mark start visited

Enqueue(start)

while Queue not empty
```

```
    node = Dequeue()

    print(node)

    for each adjacent vertex

        if not visited

            mark visited

            enqueue(vertex)
```

Flowchart

```
Start
  |
Visit Source
  |
Insert in Queue
  |
Queue Empty?
 / \
No  Yes
 |    |
Remove Node Stop
 |
Visit Neighbors
 |
Insert Unvisited Nodes
 |
Repeat
```

H. Complexity Analysis

Let:

```
V = Vertices

E = Edges
```

Time Complexity

Each vertex visited once.

Each edge checked once.

O(V+E)

Space Complexity

Queue + Visited Array

O(V)

Best Case

O(V+E)

Average Case

O(V+E)

Worst Case

O(V+E)

Why O(V+E)?

Every vertex:

Visited once.

Every edge:

Checked once.

I. Advantages

1. Finds shortest path in unweighted graph.
2. Simple implementation.
3. Level-wise traversal.
4. Useful in networking.
5. Guarantees minimum hops.

J. Disadvantages

1. More memory than DFS.
2. Queue required.
3. Large graphs may consume memory.

K. Applications

Shortest Path

Social Networks

Friend suggestions.

GPS Navigation

Network Broadcasting

Web Crawling

Google Search.

Peer-to-Peer Networks

L. Common Mistakes

❌ DFS aur BFS mix karna.

✔ BFS uses Queue.

✔ DFS uses Stack.

❌ Visited array mention nahi karna.

✔ Must write in exam.

❌ Queue operations skip kar dena.

✔ Dry run mein queue show karo.

M. Viva Questions

What is BFS?

Breadth First Search graph traversal algorithm.

Which data structure is used?

Queue.

Complexity?

O(V+E)

BFS gives shortest path?

Yes, in unweighted graphs.

BFS traversal style?

Level-wise.

N. Exam Keywords

Write these keywords:

- Breadth First Search
- Queue
- FIFO
- Graph Traversal
- Level Order Traversal
- Visited Array
- Shortest Path
- Vertex
- Edge
- Unweighted Graph

O. Memory Tricks

BFS Rule

```
Level
↓
Level
↓
Level
```

Mnemonic:

LLL

Level

Level

Level

Data Structure Trick

```
BFS → Queue
```

Remember:

```
B = Breadth

Q = Queue
```

P. Comparison Table

BFS vs DFS

Feature	**BFS**	**DFS**
Data Structure	Queue	Stack

Traversal	Level-wise	Depth-wise
Shortest Path	Yes	No
Memory	More	Less
Backtracking	No	Yes

Queue vs Stack

Queue	**Stack**
FIFO	LIFO
BFS	DFS

Q. RGPV Exam Answers

2 Mark Answer

Breadth First Search (BFS) is a graph traversal algorithm that visits all neighboring vertices before moving to the next level and uses a queue data structure.

---

5 Mark Answer

BFS traverses a graph level by level.

Steps:

1. Visit source node.
2. Insert into queue.
3. Remove node.
4. Visit neighbors.
5. Repeat until queue empty.

Complexity:

O(V+E)

Applications:

- Shortest Path
- Networking

---

7 Mark Answer

Explain BFS Algorithm

BFS is a graph traversal algorithm that explores vertices level by level using a queue.

Algorithm:

```
Visit source

Enqueue source

while queue not empty

   remove node

   visit neighbors

   enqueue neighbors
```

Example Traversal:

```
A B C D E F G
```

Complexity:

Time:

O(V+E)

Space:

O(V)

---

10 Mark Topper Answer

Breadth First Search (BFS)

Definition

BFS is a graph traversal algorithm that explores all neighboring vertices before moving to the next level.

Data Structure Used

Queue (FIFO)

Algorithm

```
BFS(start)

Visit source

Enqueue source

while queue not empty

   node = dequeue

   visit neighbors

   enqueue unvisited neighbors
```

Graph

```
          A
        /   \
       B     C
      / \   / \
     D   E F   G
```

Traversal

```
A B C D E F G
```

Complexity

Time:

O(V+E)

Space:

O(V)

Applications

- Shortest Path
- Social Networks
- Routing Algorithms

Conclusion

BFS is an efficient level-wise graph traversal technique that guarantees shortest path discovery in unweighted graphs.

R. One Page Revision Sheet

BFS

Definition

Level-wise graph traversal.

Data Structure

```
Queue (FIFO)
```

Working

```
Visit
↓
Enqueue
↓
Dequeue
↓
Visit Neighbors
↓
Repeat
```

Traversal Example

```
A B C D E F G
```

Complexity

Time:

O(V+E)

Space:

O(V)

Applications

- Shortest Path
- Routing
- Web Crawling

Memory Trick

BFS = Breadth = Queue = Level-wise

🔥 **Most Important RGPV Question:**

"Explain Breadth First Search (BFS) with algorithm, queue operations, example graph, traversal order and complexity analysis." (7 Marks)

Height Balanced Tree (AVL Tree)

(RGPV Exam-Oriented Notes | Easy Hinglish)

A. Introduction

What is AVL Tree?

AVL Tree ek **Height Balanced Binary Search Tree (BST)** hai.

Normal BST mein problem hoti hai:

```
10
  \
   20
     \
      30
        \
         40
```

Tree skewed ho jata hai.

Searching slow ho jati hai.

Complexity:

O(n)

Is problem ko solve karne ke liye AVL Tree use karte hain.

Why Do We Need AVL Tree?

AVL Tree ensure karta hai ki tree balanced rahe.

Result:

- Fast Searching
- Fast Insertion
- Fast Deletion

Average aur Worst Case:

O(logn)

Real-Life Analogy

Suppose books shelf mein properly balanced rakhi hain.

Agar saari books ek side rakh do:

```
||||||||||
```

To shelf unstable ho jayegi.

AVL Tree bhi tree ko balanced rakhta hai.

B. Definition

Exam Definition

AVL Tree is a self-balancing Binary Search Tree in which the difference between the heights of left and right subtrees of every node is at most 1.

Easy Explanation

AVL Tree = BST + Balance

Rule:

```
| Left Height - Right Height | ≤ 1
```

C. Core Concept

Height

Node se leaf tak longest path.

Example:

```
    10
   /
 20
 /
30
```

Height = 3

Balance Factor (BF)

Most Important Topic

Formula:

BF=Height(Left)−Height(Right)

Valid AVL Conditions

Balance Factor can be:

```
-1
0
+1
```

Only these values allowed.

Invalid AVL

If:

```
BF > 1
```

or

```
BF < -1
```

Tree unbalanced.

Rotation required.

D. AVL Tree Structure

Example:

```
        30
       /  \
     20    40
    /        \
   10         50
```

Balance Factors:

```
BF(30)=0

BF(20)=1
```

```
BF(40)=-1
```

AVL Valid.

E. Rotations in AVL Tree

🔥 Most Important RGPV Question

There are 4 Rotations.

---

1. LL Rotation

Occurs when:

```
Insertion in Left of Left
```

Example:

Insert:

```
30,20,10
```

Tree:

```
      30
     /
   20
  /
10
```

Unbalanced.

BF = +2

Rotation

```
Before

      30
     /
   20
  /
10
```

```
After

    20
   /  \
 10   30
```

Called:

Right Rotation

---

2. RR Rotation

Occurs when:

```
Insertion in Right of Right
```

Insert:

```
10,20,30
```

Before:

```
10
  \
   20
     \
      30
```

After:

```
   20
  /  \
 10   30
```

Called:

Left Rotation

---

3. LR Rotation

Occurs when:

```
Insertion in Right of Left
```

Example:

```
30,10,20
```

Before:

```
    30
   /
  10
    \
     20
```

Steps:

1. Left Rotation
2. Right Rotation

After:

```
    20
   /  \
```

```
   10   30
```

---

4. RL Rotation

Occurs when:

```
Insertion in Left of Right
```

Example:

```
10,30,20
```

Before:

```
10
  \
   30
   /
 20
```

Steps:

1. Right Rotation
2. Left Rotation

After:

```
     20
    /  \
   10   30
```

Memory Trick

LL → Right Rotate

RR → Left Rotate

LR → Left + Right

RL → Right + Left

F. AVL Insertion Algorithm

Steps

1. Insert node like BST.
2. Calculate Balance Factor.
3. Check imbalance.
4. Apply rotation.
5. Balance restored.

Pseudocode

```
Insert(node,key)

Insert like BST

Update Height

Compute BF

If BF > 1 or BF < -1

   Perform Rotation
```

G. Complete Dry Run

Insert:

```
10,20,30
```

Step 1

```
10
```

Step 2

```
10
  \
   20
```

Balanced.

Step 3

```
10
  \
   20
     \
      30
```

BF = -2

RR Case.

Apply Left Rotation

```
     20
    /  \
  10   30
```

Balanced.

H. Complexity Analysis

Let:

```
n = number of nodes
```

Search

O(logn)

Insert

O(logn)

Delete

O(logn)

Space Complexity

O(n)O(n)O(n)

Why O(log n)?

Tree always balanced.

Height remains:

O(logn)O(\log n)O(logn)

I. Advantages

1. Always balanced.
2. Fast searching.
3. Fast insertion.
4. Worst case efficient.
5. Better than BST.

J. Disadvantages

1. Rotations required.
2. Extra memory for height.
3. More complex than BST.

K. Applications

- Database Systems
- Memory Management
- Dictionary Implementation
- Routing Tables
- Search Engines

L. Common Mistakes

❌ BF formula wrong.

✔ BF = Left Height − Right Height

❌ LL and RR rotation confuse karna.

✔ LL → Right Rotate

✔ RR → Left Rotate

❌ AVL ko BST se different tree samajhna.

✔ AVL is a balanced BST.

M. Viva Questions

What is AVL Tree?

Self-balancing BST.

AVL Full Form?

Adelson-Velsky and Landis.

Balance Factor Formula?

BF=Height(Left)−Height(Right)

Allowed BF Values?

```
-1, 0, +1
```

Number of Rotations?

4

- LL
- RR
- LR

- RL

N. Exam Keywords

Write these keywords:

- AVL Tree
- Height Balanced Tree
- Balance Factor
- Rotation
- Self Balancing BST
- LL Rotation
- RR Rotation
- LR Rotation
- RL Rotation
- Height Difference

O. Comparison Table

BST vs AVL Tree

Feature	**BST**	**AVL**
Balanced	No	Yes
Search	O(n) Worst	O(log n)
Rotations	No	Yes
Complexity	Variable	Better
Performance	Lower	Higher

LL vs RR Rotation

Rotation	Cause	Fix
LL	Left-Left Insertion	Right Rotation
RR	Right-Right Insertion	Left Rotation

P. RGPV Exam Answers

2 Mark Answer

AVL Tree is a self-balancing Binary Search Tree in which the difference between heights of left and right subtrees is at most 1.

---

5 Mark Answer

AVL Tree maintains balance using Balance Factor.

Formula:

$BF = Height(Left) - Height(Right)$

Rotations:

- LL
- RR
- LR
- RL

Complexity:

$O(\log n)$

---

7 Mark Answer

Explain AVL Tree

AVL Tree is a self-balancing BST.

Condition:

```
BF = -1,0,+1
```

If imbalance occurs:

Rotations are applied.

Types:

1. LL
2. RR
3. LR
4. RL

Complexity:

Search:

O(logn)O(\log n)O(logn)

Applications:

- Databases
- Search Systems

---

10 Mark Topper Answer

AVL Tree (Height Balanced Tree)

Definition

AVL Tree is a self-balancing Binary Search Tree where the balance factor of each node lies between -1 and +1.

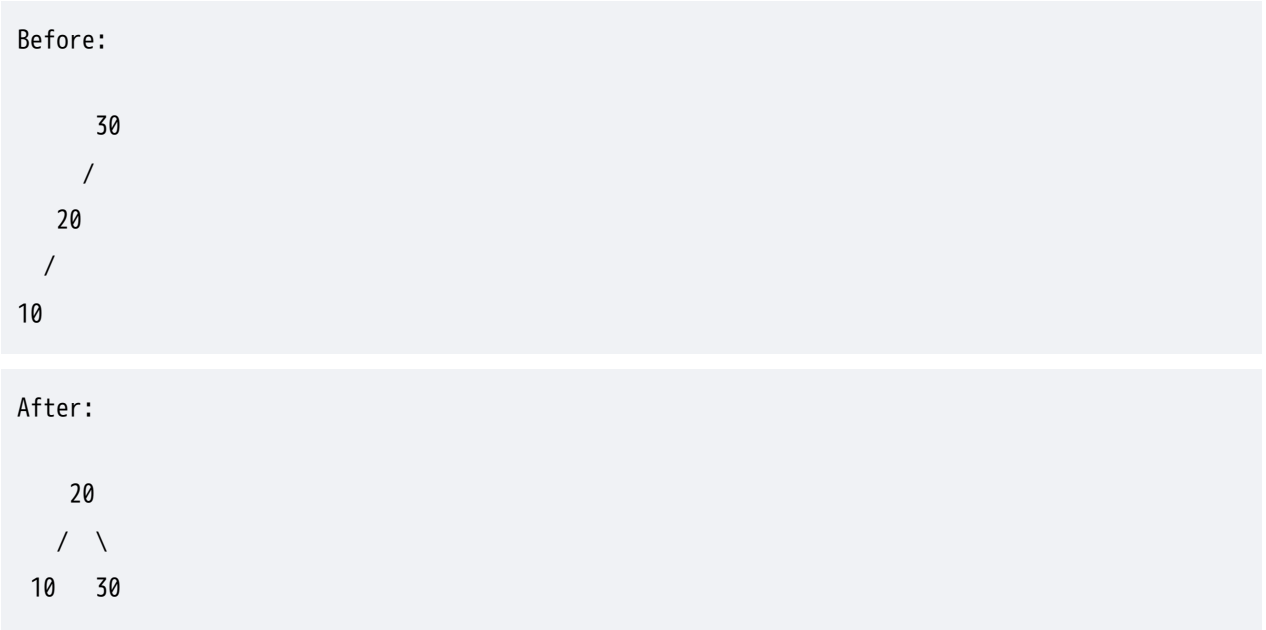
Balance Factor

$$BF = Height(Left) - Height(Right)$$

Rotations

1. LL → Right Rotation
2. RR → Left Rotation
3. LR → Left + Right Rotation
4. RL → Right + Left Rotation

Diagram



Complexity

Operation	Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

Applications

- Databases
- Routing Tables
- Search Engines

Conclusion

AVL Tree guarantees efficient operations by maintaining balance through rotations.

Q. One Page Revision Sheet

AVL Tree

Definition

Self-balancing BST.

Balance Factor

BF=Height(Left)−Height(Right)

Allowed Values

```
-1, 0, +1
```

Rotations

LL → Right

RR → Left

LR → Left + Right

RL → Right + Left

Complexity

Operation	Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

Memory Trick

LL → Right

RR → Left

LR → L+R

RL → R+L

🔥 **Most Important RGPV Question:**
"Explain AVL Tree with Balance Factor, LL/RR/LR/RL Rotations, insertion process and complexity analysis."

B-Tree

(RGPV Exam-Oriented Notes | Easy Hinglish)

A. Introduction

What is B-Tree?

B-Tree ek **self-balancing multiway search tree** hai.

BST mein ek node ke maximum 2 children hote hain.

B-Tree mein:

```
1 Node
↓
Many Children
```

ho sakte hain.

Why Do We Need B-Tree?

BST aur AVL Tree RAM ke liye achhe hain.

Lekin databases aur hard disks mein millions of records hote hain.

Wahan B-Tree use hota hai.

Examples:

- Database Indexing
- File Systems
- Search Engines

Real-Life Analogy

Library mein books ko shelves mein arrange kiya gaya hai.

Har shelf par multiple books hain.

Ek shelf se hi kai books mil jaati hain.

Ye B-Tree jaisa hai.

B. Definition

Exam Definition

A B-Tree is a self-balancing m-way search tree in which nodes contain multiple keys and multiple children while maintaining sorted order.

Easy Explanation

BST:

```
One Key
↓
Two Children
```

B-Tree:

```
Many Keys
↓
Many Children
```

C. Core Concept

Order of B-Tree

Most Important Concept

Suppose:

```
Order = m
```

Then:

Maximum Children

```
m
```

Maximum Keys

```
m - 1
```

Minimum Children

```
⌈m/2⌉
```

Minimum Keys

```
⌈m/2⌉ - 1
```

Example

Order:

```
m = 4
```

Then:

Maximum Children:

```
4
```

Maximum Keys:

```
3
```

Minimum Children:

```
2
```

Minimum Keys:

```
1
```

D. Properties of B-Tree

🔥 Frequently Asked Theory Question

Property 1

All leaves are at same level.

Property 2

Keys remain sorted.

Property 3

Node may contain multiple keys.

Property 4

Tree always balanced.

Property 5

Search, Insert, Delete are efficient.

E. Structure of B-Tree

Example:

Order = 4

```
          [20 | 40]

        /     |      \

     [10]   [30]   [50|60]
```

Explanation:

```
10 < 20

20 < 30 < 40

50 > 40
```

F. Search Operation

Search:

```
30
```

Tree:

```
          [20 | 40]

        /     |      \

     [10]   [30]   [50|60]
```

Step 1

Compare:

```
30
```

with:

```
20 | 40
```

Step 2

30 lies between:

```
20 and 40
```

Step 3

Move to middle child.

Step 4

Found:

```
30
```

Search Pseudocode

```
Search(node,key)

if key found

   return FOUND

choose proper child

Search(child,key)
```

G. Insertion in B-Tree

🔥 Most Important Numerical

Suppose Order = 3

Insert:

```
10,20,30
```

Step 1

Insert 10

```
[10]
```

Step 2

Insert 20

```
[10 | 20]
```

Step 3

Insert 30

Overflow occurs.

```
[10 | 20 | 30]
```

Split

Middle key goes up.

```
     [20]

    /    \

  [10]   [30]
```

Balanced tree obtained.

Why Split?

Node capacity exceed ho gayi.

B-Tree balance maintain karta hai.

H. Deletion in B-Tree

Deletion comparatively difficult hai.

Exam mein mostly theory poochte hain.

Cases

Case 1

Delete from leaf.

Simple removal.

Case 2

Delete internal node.

Replace with predecessor/successor.

Case 3

Underflow occurs.

Borrow or Merge.

I. Complete Dry Run

Insert:

```
10,20,30,40,50
```

After 10

```
[10]
```

After 20

```
[10 | 20]
```

After 30

Overflow.

Split.

```
     [20]

    /    \

  [10]   [30]
```

Insert 40

```
     [20]

    /    \
```

```
[10] [30|40]
```

Insert 50

Split right node.

```
        [20 | 40]

       /   |    \

    [10] [30] [50]
```

Final B-Tree.

J. Complexity Analysis

Let:

```
n = number of keys
```

Search

O(logn)

Insertion

O(logn)

Deletion

O(logn)

Space Complexity

O(n)

Why O(log n)?

Tree height very small rehti hai.

Nodes multiple keys store karte hain.

K. Advantages

1. Always balanced.
2. Fast searching.
3. Efficient disk access.
4. Suitable for databases.
5. Reduced tree height.

L. Disadvantages

1. Complex implementation.
2. Insertion and deletion difficult.
3. More memory overhead.

M. Applications

Database Indexing

MySQL

Oracle

SQL Server

File Systems

Windows NTFS

Linux File Systems

Search Engines

Large Data Storage

N. Common Mistakes

❌ B-Tree aur BST same samajhna.

✔ B-Tree = Multiway Tree.

❌ Order formula bhool jana.

✔ Very important.

❌ Split operation explain na karna.

✔ Exam mein zaroor likho.

O. Viva Questions

What is B-Tree?

Balanced multiway search tree.

Why B-Tree used?

Database indexing.

Maximum Keys in Order m?

```
m - 1
```

Maximum Children?

```
m
```

Complexity?

O(logn)

P. Exam Keywords

Write these keywords:

- B-Tree
- Multiway Search Tree
- Balanced Tree
- Order m
- Node Splitting
- Overflow
- Underflow
- Database Indexing
- Search Operation
- Disk Access

Q. Memory Tricks

B-Tree Formula Trick

For Order m:

```
Children = m

Keys = m-1
```

Remember:

CK Rule

Children = m

Keys = m−1

Overflow Rule

```
Overflow

↓

Split

↓

Middle Key Up
```

R. Comparison Table

BST vs B-Tree

Feature	BST	B-Tree
Keys per Node	1	Multiple
Children	2	Many
Balance	Not Always	Always
Height	Large	Small
Database Use	No	Yes

AVL vs B-Tree

Feature	AVL	B-Tree
Children	2	Multiple
Storage	RAM	Disk
Height	Larger	Smaller
Database Usage	Less	High

S. RGPV Exam Answers

2 Mark Answer

A B-Tree is a self-balancing multiway search tree that stores multiple keys in a node and maintains sorted order.

---

5 Mark Answer

B-Tree

B-Tree is a balanced search tree used in databases.

Properties:

- Multiple keys per node.
- Balanced structure.
- Efficient searching.

Applications:

- Database indexing
- File systems

Complexity:

O(logn)

---

7 Mark Answer

Explain B-Tree

B-Tree is a balanced multiway search tree.

Properties:

1. Sorted keys.
2. Multiple children.
3. All leaves at same level.

Insertion:

- Insert key.
- Handle overflow.
- Split node.

Complexity:

Search:

O(logn)

Applications:

- Databases
- File Systems

---

10 Mark Topper Answer

B-Tree

Definition

B-Tree is a self-balancing multiway search tree used for efficient storage and retrieval of large data.

Properties

- Balanced tree
- Multiple keys
- Multiple children
- Sorted structure

Example

```
          [20 | 40]

        /     |      \

     [10]   [30]   [50|60]
```

Insertion

Overflow → Split → Middle Key Up

Complexity

Operation	Complexity
Search	O(log n)

Insert	O(log n)
Delete	O(log n)

Applications

- Databases
- File Systems
- Search Engines

Conclusion

B-Tree is widely used where large amounts of data need fast searching and indexing.

T. One Page Revision Sheet

B-Tree

Definition

Balanced Multiway Search Tree

Order m

Children = m

Keys = m−1

Properties

- Balanced
- Sorted
- Multiple Keys
- Multiple Children

Insertion

```
Overflow
↓
Split
↓
Middle Key Up
```

Complexity

Operation	Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

Applications

- Database Indexing
- File Systems
- Search Engines

Memory Trick

CK Rule

Children = m

Keys = m−1

🔥 **Most Important RGPV Question:**

"Explain B-Tree with properties, insertion using node splitting, search operation and complexity analysis." (7–10 Marks)

2-3 Tree

2-3 Tree

(RGPV Exam-Oriented Notes | Easy Hinglish)

A. Introduction

What is 2-3 Tree?

2-3 Tree ek **balanced search tree** hai.

Ye B-Tree ka special case hai.

2-3 Tree mein node ke paas:

- 1 key ho sakti hai (2-node)
- 2 keys ho sakti hain (3-node)

Aur isi ke according children hote hain.

Why Do We Need 2-3 Tree?

Normal BST skew ho sakta hai:

```
10
  \
   20
     \
      30
```

Searching slow ho jaati hai.

2-3 Tree automatically balanced rehta hai.

Real-Life Analogy

Imagine ek school office hai.

Students ko sections mein divide kiya gaya hai.

Har section mein proper arrangement hai.

Isliye kisi student ko dhoondhna easy hai.

2-3 Tree bhi data ko balanced form mein organize karta hai.

B. Definition

Exam Definition

A 2-3 Tree is a balanced search tree in which each internal node is either:

- a 2-node having one key and two children, or
- a 3-node having two keys and three children.

All leaves are at the same level.

Easy Explanation

2-3 Tree mein:

```
Node Type 1

[20]
```

1 key → 2 children

```
Node Type 2
```

```
[20 | 40]
```

2 keys → 3 children

C. Core Concept

---

1. 2-Node

Contains:

```
1 Key
2 Children
```

Example:

```
     [30]
    /    \
```

---

2. 3-Node

Contains:

```
2 Keys
3 Children
```

Example:

```
  [20 | 40]
  /   |   \
```

Ordering Rule

For:

```
[20 | 40]
```

Left Child:

```
< 20
```

Middle Child:

```
20 to 40
```

Right Child:

```
> 40
```

D. Structure / Diagram

Example:

```
          [30]

        /     \

     [10]    [40 | 50]
```

Balanced Tree.

All leaves same level.

Another Example

```
             [20 | 40]

          /     |     \

       [10]   [30]   [50]
```

E. Search Operation

Search:

```
30
```

Tree:

```
              [20 | 40]

           /      |      \

        [10]    [30]    [50]
```

Step 1

Compare:

```
30
```

with:

```
20 | 40
```

Step 2

30 lies between:

```
20 and 40
```

Step 3

Go to middle child.

Step 4

Found:

```
30
```

Search Algorithm

```
Search(node,key)

if key found

   return FOUND

choose correct child

search(child,key)
```

F. Insertion Operation

🔥 Most Important Question

Insert:

```
10,20,30
```

Step 1

Insert 10

```
[10]
```

Step 2

Insert 20

```
[10 | 20]
```

Step 3

Insert 30

Overflow occurs.

```
[10 | 20 | 30]
```

Not allowed.

Split Node

Middle key moves up.

```
      [20]

     /    \

  [10]   [30]
```

Balanced tree created.

G. Complete Dry Run

Insert:

```
10,20,30,40,50
```

After 10

```
[10]
```

After 20

```
[10|20]
```

After 30

Split.

```
     [20]

    /    \

 [10]   [30]
```

After 40

```
     [20]

    /    \

 [10] [30|40]
```

After 50

Overflow.

Split right node.

```
         [20 | 40]

       /     |      \

     [10]  [30]   [50]
```

Final Tree.

H. Deletion Operation

Exam mein mostly theory poochte hain.

Cases

Case 1

Delete from leaf.

Simple delete.

Case 2

Node becomes empty.

Borrow from sibling.

Case 3

Borrow impossible.

Merge nodes.

I. Properties of 2-3 Tree

🔥 Frequently Asked

1. Every internal node is 2-node or 3-node.
2. Tree always balanced.
3. All leaves are at same level.
4. Search, Insert, Delete efficient.
5. Sorted order maintained.

J. Complexity Analysis

Let:

```
n = Number of Keys
```

Search

O(logn)

Insert

O(logn)

Delete

O(logn)

Space Complexity

O(n)

Why O(log n)?

Tree balanced rehta hai.

Height small hoti hai.

K. Advantages

1. Always balanced.
2. Fast searching.
3. Efficient insertion.
4. Small tree height.
5. Better than ordinary BST.

L. Disadvantages

1. Complex implementation.
2. Node splitting required.
3. Deletion difficult.

M. Applications

- Database Systems
- File Systems
- Indexing
- Search Applications
- Large Data Storage

N. Common Mistakes

❌ 2-node aur 3-node confuse karna.

✔ 2-node = 1 key

✔ 3-node = 2 keys

❌ Leaves same level property bhool jana.

✔ Very Important.

❌ Split operation explain nahi karna.

✔ Always write.

O. Viva Questions

What is 2-3 Tree?

Balanced search tree consisting of 2-nodes and 3-nodes.

Why called 2-3 Tree?

Nodes contain either:

- 2 children
- 3 children

Is it balanced?

Yes.

Complexity?

O(logn)

Is 2-3 Tree a BST?

Yes, balanced search tree.

P. Exam Keywords

Write these keywords:

- 2-Node
- 3-Node
- Balanced Search Tree
- Node Splitting
- Ordered Keys
- Same Level Leaves
- Search Operation
- Insert Operation
- Overflow
- Merge

Q. Memory Tricks

Node Types

```
2-Node

1 Key
2 Children
```

```
3-Node

2 Keys
3 Children
```

Mnemonic:

2 → 1 Key

3 → 2 Keys

Overflow Rule

```
Overflow
↓
Split
↓
Middle Up
```

R. Comparison Table

BST vs 2-3 Tree

Feature	**BST**	**2-3 Tree**
Balanced	No	Yes
Height	Larger	Smaller
Search	O(n) Worst	O(log n)
Keys/Node	1	Multiple

2-3 Tree vs B-Tree

Feature	**2-3 Tree**	**B-Tree**
Children	2 or 3	Many
Complexity	O(log n)	O(log n)
Flexibility	Less	More
Structure	Simpler	Generalized

S. RGPV Exam Answers

2 Mark Answer

A 2-3 Tree is a balanced search tree in which each internal node is either a 2-node (one key, two children) or a 3-node (two keys, three children).

---

5 Mark Answer

2-3 Tree

A 2-3 Tree is a balanced search tree.

Features:

- 2-node and 3-node.
- All leaves at same level.
- Balanced structure.

Complexity:

O(logn)

Applications:

- Database indexing
- File systems

---

7 Mark Answer

Explain 2-3 Tree

A 2-3 Tree is a balanced search tree containing:

- 2-node

- 3-node

Properties:

- Balanced
- Sorted
- Same level leaves

Insertion:

Overflow → Split → Middle Key Up

Complexity:

Search:

O(logn)

Applications:

- Databases
- File Systems

---

10 Mark Topper Answer

2-3 Tree

Definition

A 2-3 Tree is a balanced search tree where each node is either a 2-node or a 3-node.

Structure

```
          [20 | 40]

        /     |      \

     [10]   [30]   [50]
```

Properties

- Balanced
- Sorted
- All leaves same level

Insertion

Overflow causes node splitting.

Middle key moves upward.

Complexity

Operation	Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

Applications

- Database Indexing
- File Systems

Conclusion

2-3 Tree provides efficient searching and insertion while maintaining balance automatically.

T. One Page Revision Sheet

2-3 Tree

Definition

Balanced Search Tree

2-Node

```
1 Key
2 Children
```

3-Node

```
2 Keys
3 Children
```

Properties

- Balanced
- Sorted
- Same Level Leaves

Insertion

```
Overflow
↓
Split
↓
Middle Up
```

Complexity

Operation	**Complexity**
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

Memory Trick

2 → 1 Key

3 → 2 Keys

🔥 **Most Important RGPV Question:**
"Explain 2-3 Tree with structure, insertion using node splitting, properties and complexity analysis." (7 Marks)

NP-Completeness

(RGPV Exam-Oriented Notes | Easy Hinglish | Most Important Theory Topic)

A. Introduction

What is NP-Completeness?

NP-Completeness Algorithm Design ka ek important theoretical concept hai.

Ye batata hai:

> **Koi problem solve karna kitna difficult hai?**

Aur:

> **Kya uska fast algorithm exist karta hai ya nahi?**

Why Do We Need It?

Suppose tumhare paas 1000 cities ka Traveling Salesman Problem (TSP) hai.

Question:

```
Kya iska fast solution hai?
```

NP-Completeness isi type ke questions ka answer deti hai.

Real-Life Analogy

Suppose exam ka answer verify karna hai.

```
Answer diya hua hai
```

Verification:

```
2 minute
```

But answer khud find karna:

```
2 ghante
```

Yahi P aur NP ka basic difference hai.

B. Definition

Exam Definition

NP-Complete problems are those problems that belong to NP and are as hard as any problem in NP.

Easy Explanation

NP-Complete problems:

```
Solve karna difficult

Check karna easy
```

C. Core Concept

NP-Completeness samajhne ke liye 4 terms zaroori hain:

1. P Problems
2. NP Problems
3. NP-Hard Problems
4. NP-Complete Problems

---

1. P Problems

Definition

Problems that can be solved in polynomial time.

Meaning

Fast algorithm available.

Examples

- Binary Search
- Merge Sort
- BFS
- DFS

Complexity

Examples:

O(n)

O(nlogn)

O(n^2)

---

2. NP Problems

NP = Non-deterministic Polynomial Time

Definition

Problems whose solutions can be verified in polynomial time.

Meaning

Answer check karna easy hai.

Answer find karna difficult ho sakta hai.

Example

Sudoku

Given solution:

```
Check karna easy
```

Solution find karna:

```
Difficult
```

---

3. NP-Hard Problems

Definition

Problems at least as hard as NP problems.

Meaning

Very difficult problems.

Verification bhi difficult ho sakti hai.

Examples

- Optimization TSP
- Scheduling Problems

---

4. NP-Complete Problems

Most Important Topic

Definition

A problem is NP-Complete if:

Condition 1

Problem NP mein ho.

Condition 2

Every NP problem can be reduced to it.

Meaning

NP-Complete = Hardest Problems in NP.

D. Relationship Diagram

🔥 Frequently Asked Diagram

```
+----------------------+
|          NP          |
|                      |
|   +-------------+    |
|   | NP-Complete |    |
|   +-------------+    |
+----------------------+

NP-Hard includes NP-Complete
and may extend outside NP.
```

Better exam diagram:

```
            NP-Hard

      --------------------

      |                  |

      |  NP-Complete     |

      |                  |

      --------------------

             ∩
```

```
NP
```

E. P vs NP

Most Asked Theory Question

Feature	**P**	**NP**
Solve	Easy	Difficult
Verify	Easy	Easy
Polynomial Time	Yes	Verification Only
Examples	Sorting	Sudoku

F. Polynomial Time

Polynomial means:

O(n)

O(n^2)

O(n^3)

Non-Polynomial

Examples:

O(2^n)

O(n!)

These are generally hard problems.

G. Reducibility

Most Important Keyword

What is Reduction?

One problem ko dusri problem mein convert karna.

Example:

```
Problem A

↓

Convert

↓

Problem B
```

Agar B solve ho gaya,

A bhi solve ho jayega.

Why Important?

NP-Completeness prove karne ke liye.

H. Steps to Prove NP-Complete

🔥 Frequently Asked 5 Marks

Step 1

Show problem belongs to NP.

Step 2

Take known NP-Complete problem.

Step 3

Reduce known problem to new problem.

Step 4

Polynomial reduction prove karo.

Step 5

Conclude NP-Complete.

I. Examples of NP-Complete Problems

Very Important

Traveling Salesman Problem (Decision Version)

Hamiltonian Cycle

Graph Coloring

Knapsack Problem

Clique Problem

Vertex Cover

SAT Problem

J. Complexity Analysis

No known polynomial algorithm.

Generally:

```
Exponential Time
```

Examples:

O(2^n)

O(n!)

K. Advantages

1. Helps classify difficult problems.
2. Useful in algorithm design.
3. Identifies impossible optimizations.
4. Research applications.

L. Disadvantages

1. Difficult mathematics.
2. Hard proofs.

3. Mostly theoretical.

M. Applications

- Artificial Intelligence
- Scheduling
- Cryptography
- Optimization
- Network Design
- Operations Research

N. Common Mistakes

❌ NP = Not Possible

✔ Wrong

NP = Non-deterministic Polynomial Time

❌ NP-Complete and NP-Hard same samajhna.

✔ Different.

❌ P and NP confuse karna.

✔ P = Solve Fast

✔ NP = Verify Fast

O. Viva Questions

What is NP?

Problems whose solutions can be verified in polynomial time.

What is NP-Complete?

Hardest problems in NP.

What is NP-Hard?

Problems at least as hard as NP.

Give NP-Complete examples.

- Hamiltonian Cycle
- TSP
- Graph Coloring

What is Reduction?

Converting one problem into another.

P. Exam Keywords

Write these keywords:

- Polynomial Time
- Verification
- Reducibility
- NP
- NP-Hard
- NP-Complete
- Optimization
- Computational Complexity

- Decision Problem
- Exponential Complexity

Q. Memory Tricks

P vs NP

```
P

Solve Fast
```

```
NP

Check Fast
```

Mnemonic:

P = Produce Fast

NP = Notice Fast

NP-Complete

```
NP

+

Hardest
```

=

NP-Complete

R. Comparison Tables

P vs NP

Feature	P	NP
Solve	Fast	Hard
Verify	Fast	Fast
Complexity	Polynomial	Verification Polynomial

NP vs NP-Hard

Feature	NP	NP-Hard
Verification	Easy	May Not Be
Inside NP	Yes	Not Necessary
Difficulty	High	Higher

NP-Hard vs NP-Complete

Feature	NP-Hard	NP-Complete
Belongs to NP	Not Necessary	Yes
Verification	Not Required	Polynomial
Difficulty	Very Hard	Hardest in NP

S. RGPV Exam Answers

2 Mark Answer

NP-Complete problems are problems that belong to NP and are as hard as every other problem in NP.

---

5 Mark Answer

NP-Completeness

NP-Complete problems are the hardest problems in NP.

Conditions:

1. Problem must belong to NP.
2. Every NP problem can be reduced to it.

Examples:

- TSP
- Hamiltonian Cycle
- Graph Coloring

---

7 Mark Answer

Explain NP-Completeness

NP-Completeness is used to classify computationally difficult problems.

Definitions:

- P
- NP
- NP-Hard
- NP-Complete

A problem is NP-Complete if:

- It belongs to NP.
- Every NP problem reduces to it.

Examples:

- TSP
- Knapsack
- SAT

Applications:

- Optimization
- AI
- Scheduling

---

10 Mark Topper Answer

NP-Completeness

Definition

NP-Complete problems are the problems that belong to NP and are as hard as every other problem in NP.

Related Classes

- P
- NP
- NP-Hard
- NP-Complete

Diagram

```
            NP-Hard

      ------------------

      | NP-Complete   |
```

```
    -----------------

          ∩

          NP
```

Conditions

1. Problem belongs to NP.
2. Polynomial reduction possible.

Examples

- Hamiltonian Cycle
- TSP
- Graph Coloring
- Knapsack

Applications

- AI
- Scheduling
- Cryptography

Conclusion

NP-Complete problems represent the most difficult class of problems whose solutions can be verified in polynomial time.

T. One Page Revision Sheet

NP-Completeness

P

Solve Fast

NP

Verify Fast

NP-Hard

At least as hard as NP

NP-Complete

NP + Hardest

Conditions

1. Belongs to NP
2. Polynomial Reduction

Examples

- TSP
- Hamiltonian Cycle
- Graph Coloring
- Knapsack

Diagram

```
NP-Hard

   |

NP-Complete

   |
```

```
NP
```

Keywords

- Polynomial Time
- Verification
- Reduction
- NP-Hard
- NP-Complete

Memory Trick

P = Produce Fast

NP = Notice Fast

NPC = NP + Complete Hard

🔥 **Most Important RGPV Question:**
"Explain P, NP, NP-Hard and NP-Complete problems with suitable examples and relationship diagram." (7–10 Marks)

Unit-5 Last Night Priority

Study in this order:

1. BST 🔥
2. Tree Traversals 🔥
3. DFS 🔥
4. BFS 🔥
5. AVL Tree ⭐
6. B-Tree ⭐
7. NP-Completeness ⭐

8. 2-3 Tree ✔

These 8 topics cover almost the entire Unit-5 syllabus and typically account for the majority of Unit-5 marks in RGPV ADA exams.