2–3 TREE – Complete Notes

1. Definition

A 2–3 Tree is a self-balancing search tree in which:

- Every internal node has either 2 children (2-node) or 3 children (3-node).
- All leaf nodes are at the same level (height-balanced).
- It is used to store data in sorted order.

It is a special case of a B-tree of order 3.

2. Node Types in a 2–3 Tree

1. 2-Node

- **Contains 1 key**
- **Has 2 children**

```
| key |
 /     \
```

2. 3-Node

- **Contains 2 keys**
- **Has 3 children**

```
| key1 | key2 |
 /     |     \
```

3. Properties of 2–3 Tree

1. Every node is either a 2-node or 3-node
2. All leaves are at the same level
3. Keys are stored in sorted order
4. Search, insert, delete = O(log n)
5. Value ranges:
    - 2-node: one key → 2 children
    - 3-node: two keys → 3 children

4. Searching in a 2–3 Tree

Search like in BST but with 1 or 2 comparisons:

- For 2-node: compare with 1 key
- For 3-node: compare with 2 keys

Choose the child accordingly.

5. Insertion in 2–3 Tree (MOST IMPORTANT)

Steps:

1. Insert into leaf node.
2. If leaf is:
    - 2-node → becomes 3-node (simple insert)
    - 3-node → overflow happens → split
3. Splitting promotes the middle key to the parent.

Insertion Example (Step-by-step)

Insert: 10

```
[10]
```

Insert: 20

```
[10 20]
```

Insert: 30 → overflow (3 keys → split)

Keys: 10, 20, 30
Middle = 20

Split into:

```
  [20]
  /    \
[10]  [30]
```

Next insert: 5

Insert into leaf [10]:

```
  [20]
  /    \
[5 10] [30]
```

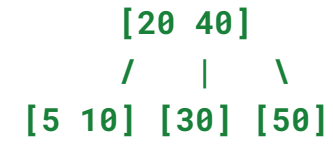
Next insert: 40

Insert into [30]:

```
  [20]
  /     \
[5 10] [30 40]
```

Next insert: 50 → split right node

[30 40 50] → middle is 40



6. Splitting Rules

If a 3-node gets one more key:

- Convert 3 keys → sort them → split into two 2-nodes
- Middle key moves to parent
- If parent overflows → split recursively

7. Deletion in 2–3 Tree

Deletion is more complex but rules are clear.

Steps:

1. Find and delete the key
2. If deletion occurs in internal node → replace with in-order predecessor or successor
3. After deletion:
    - If node sis till valid (has ≥1 key) → done
    - If node underflows (no key left) → fix it

TRAVERSAL IN 2–3 TREE (Full Detailed Notes)

A 2–3 tree node can be:

- 2-node → 1 key → 2 children
- 3-node → 2 keys → 3 children

This changes how traversal is performed.

Traversal order is similar to a BST, but with extra key and extra child in 3-node.

1. INORDER TRAVERSAL (Left → Keys → Right)

Rules for Inorder Traversal

For a 2-node `[K1]`

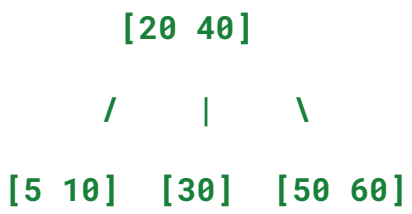
```
LEFT CHILD → K1 → RIGHT CHILD
```

For a 3-node `[K1 K2]`

```
LEFT CHILD → K1 → MIDDLE CHILD → K2 → RIGHT CHILD
```

✔ Example

Tree:



Inorder:

1. Traverse left subtree: `[5 10]`→ 5, 10
2. Visit first key: 20
3. Traverse middle: 30
4. Visit the second key: 40
5. Traverse right: 50, 60

Final Inorder Output:

```
5, 10, 20, 30, 40, 50, 60
```

PREORDER TRAVERSAL (Node → Subtrees)

Rules

For a 2-node `[K1]`

`K1 → LEFT CHILD → RIGHT CHILD`

For a 3-node `[K1 K2]`

`K1 → K2 → LEFT CHILD → MIDDLE CHILD → RIGHT CHILD`

(We print BOTH keys before children.)

✔ Example

Same tree:

```
        [20 40]
       /    |    \
  [5 10]  [30]  [50 60]
```

Preorder:

1. **Visit root keys: 20, 40**
2. **Preorder left subtree: 5, 10**
3. **Preorder middle: 30**
4. **Preorder right: 50, 60**

Final Preorder Output: `20, 40, 5, 10, 30, 50, 60`

3. POSTORDER TRAVERSAL (Subtrees → Node)

Rules

For a 2-node `[K1]`

`LEFT CHILD → RIGHT CHILD → K1`

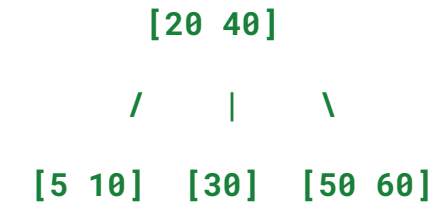
For a 3-node `[K1 K2]`

`LEFT CHILD → MIDDLE CHILD → RIGHT CHILD → K1 → K2`

(Visit children first, then keys.)

✔ Example

Tree:



Postorder:

1. **Left subtree: 5, 10**
2. **Middle: 30**
3. **Right: 50, 60**
4. **Root keys: 20, 40**

Final Postorder Output:

`5, 10, 30, 50, 60, 20, 40`

4. LEVEL ORDER TRAVERSAL (Breadth-First)

Not always asked, but useful.

Rules:

Use a queue.

Visit nodes level by level, keys inside node from left to right.

✔ Example:

Tree:

```
        [20 40]
       /    |    \
  [5 10]  [30]  [50 60]
```

Level Order:

```
20 40
5 10   30   50 60
```