Height Balanced Trees: AVL Trees
(Insertion and Deletion in AVL Tree)

1. What is an AVL Tree?

- AVL tree is a self-balancing binary search tree.
- It maintains the balance factor of every node to be −1, 0 or +1.
- Balance factor (BF) = height(left subtree) − height(right subtree).
- If the balance factor becomes < −1 or > 1, the tree is unbalanced and rotations are performed to restore balance.

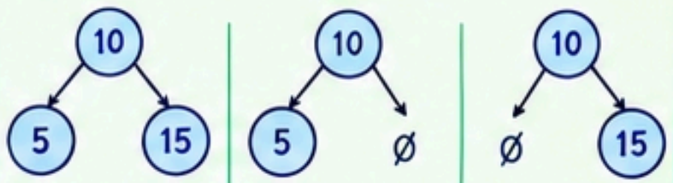
Key Point:

- Balance factor ∈ {−1, 0, +1}
- Height of left and right subtrees differ by at most 1.

2. AVL Tree Properties

- It is a Binary Search Tree (BST).
- It is height-balanced (BF ∈ {−1, 0, +1}).
- Maintains balance using rotations (LL, RR, LR, RL).
- Height is O(log n), so search, insertion and deletion are efficient.

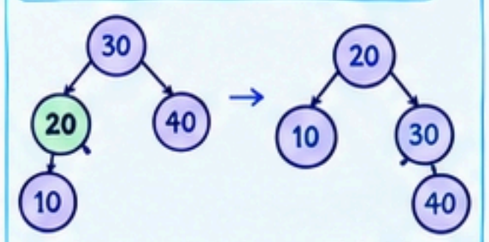
Balance Factor Examples



3. Rotations in AVL Tree

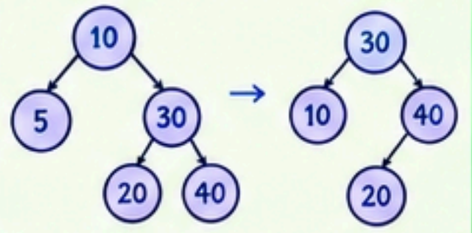
Used to restore balance when BF becomes ±2.

LL Rotation (Right Rotation)



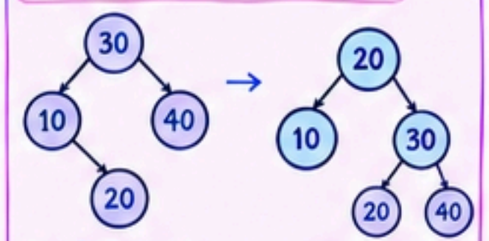
Used when imbalance is on left-left side.

RR Rotation (Left Rotation)



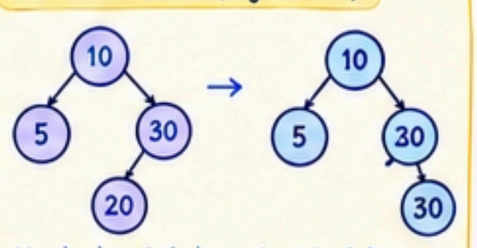
Used when imbalance is on right-right side.

LR Rotation (Left-Right)



Used when imbalance is left-right side.

RL Rotation (Right-Left)



Used when imbalance is right-left side.

4. Insertion in AVL Tree

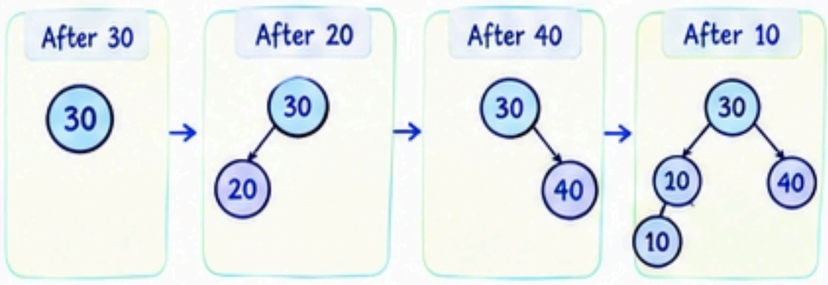
Steps:

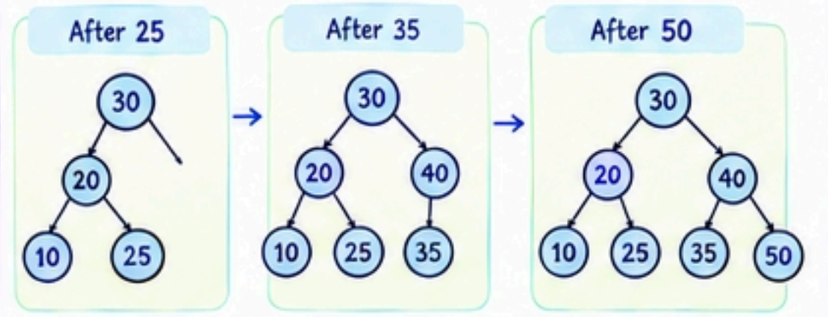
1. Insert the new node as in a BST.
2. Update the height of all ancestor nodes.
3. Check the balance factor.
4. If unbalanced, perform the required rotation (LL, RR, LR, RL).
5. Continue up to the root.

Note:

- Rotations are done only when BF becomes ±2.
- After rotation, the tree becomes balanced.

Example: Insert 30, 20, 40, 10, 25, 35, 50





Tree remains balanced after each insertion.

5. Deletion in AVL Tree

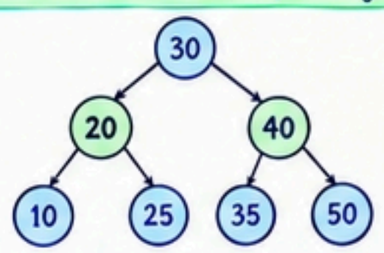
Steps:

1. Find the node to delete.
2. If node has 0 or 1 child, replace it with its child (or null).
3. If node has 2 children, replace it with inorder successor (smallest in right subtree).
4. Delete the successor node.
5. Update heights and check balance factors.
6. Perform rotations if needed.

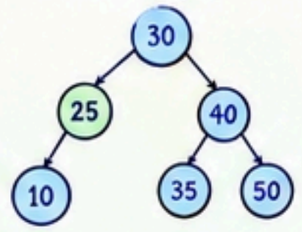
Note:

- Rebalancing may require multiple rotations (as in insertion).

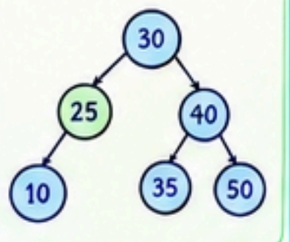
Example: Delete 20 from the following AVL tree



1. Find 20 and replace with its successor (25).



2. Delete 25 (leaf node).



Tree is rebalanced (BF of all nodes ∈ {−1, 0, +1}).

Key Takeaways

- AVL tree is a self-balancing BST with balance factor −1, 0, +1.
- Rotations (LL, RR, LR, RL) are used to restore balance.
- Insertion and deletion may require rotations and height updates.

AVL trees ensure:

- Balanced height
- O(log n) search, insert and delete
- Better performance than unbalanced BSTs.

M-way Tree
(Insertion and Deletion in M-way Tree)

1. What is an M-way Tree?

- An M-way tree is a tree in which each node can have at most M children.
- It is a generalization of a binary tree (where M = 2).
- Often used in file systems and databases (e.g., B-tree, B+ tree).

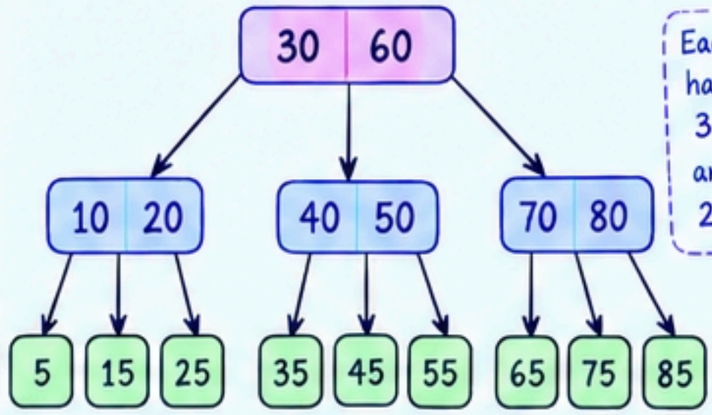
M = maximum number of children (order of the tree).

2. Structure and Properties

- Each node can have at most M children.
- A non-root node has at least ⌈M/2⌉ children (for a balanced M-way tree).
- Root can have 2 to M children.
- Each node stores up to (M − 1) keys.
- Keys in a node are in sorted order.
- Each internal node has (number of children = number of keys + 1).

For example, in a 3-way tree (M = 3):
- Each node can have at most 3 children.
- Each node can store at most 2 keys.

3. Example of an M-way Tree (M = 3)



Here, **M = 3** (maximum children), so each node can have up to 2 keys.

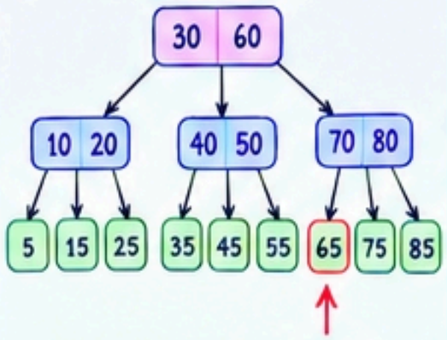
4. Insertion in M-way Tree

Steps:

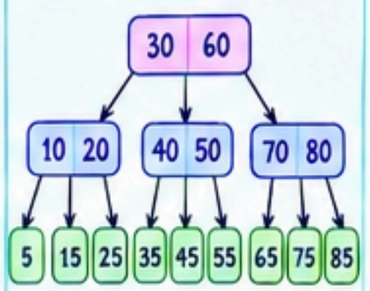
1. Find the correct leaf node (using key comparison).
2. If the leaf has space (less than M − 1 keys), insert the key in sorted order.
3. If the leaf is full, split the node and promote the middle key to parent.
4. Repeat the process if the parent becomes full.

Example: Insert 65 in the above 3-way tree

1. Find the leaf node

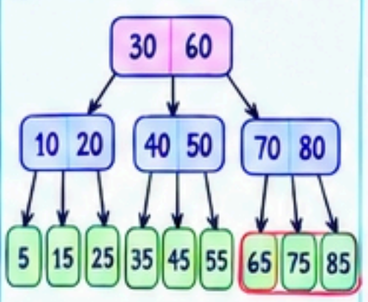


2. Insert 65



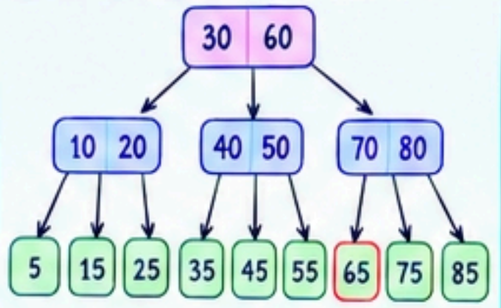
Leaf has space, so insert 65 in sorted order.

3. After Insertion



No split needed (as M = 3).

Final Tree



65 is successfully inserted.

5. Deletion in M-way Tree

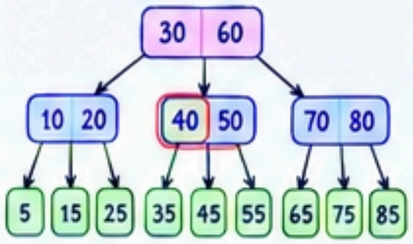
Steps:

1. Find the key in the tree (at a leaf or internal node).
2. If the key is in a leaf, remove it.
3. If the key is in an internal node, replace it with its in-order successor (smallest key in right subtree). or in-order predecessor (largest key in left subtree).
4. If the node becomes underfull (less than ⌈M/2⌉ keys), rebalance using borrow or merge.

Note:
- If rebalancing is required, we use borrow (from sibling) or merge (with sibling).

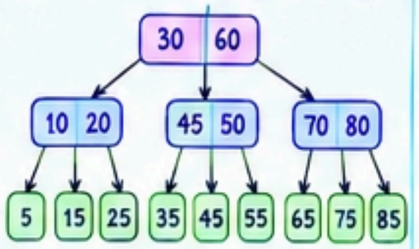
Example: Delete 40 in the above M-way tree (M = 3)

1. Find the key (40)



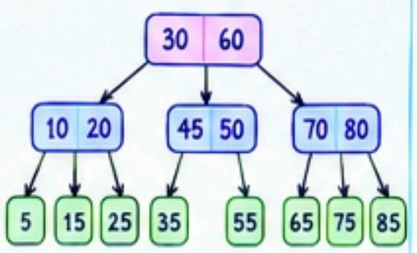
40 is in internal node.

2. Replace with successor



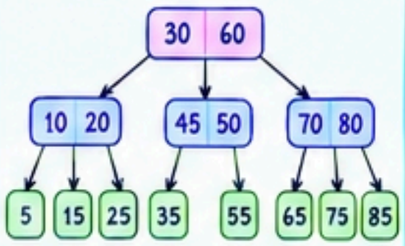
Successor of 40 is 45 (smallest in right subtree).

3. Delete 45 (from leaf)



Remove 45 from leaf.

4. Final Tree



Tree is balanced (no underflow).

Key Takeaways:

- M-way tree allows up to M children per node.
- Insertion: add key → split if full → promote middle key.
- Deletion: remove key → replace (if internal) → rebalance if underflow.
- Used in file systems (e.g., B-tree) and databases.

B Tree
(Insertion and Deletion in B Tree)

1. What is a B Tree?

- A B Tree is a self-balanced multiway search tree designed to work efficiently with large data sets and disk storage.
- Each node can have more than two children.
- Keys in each node are stored in sorted order.
- All leaves are at the same level.
- Used in databases, file systems and operating systems (e.g., NTFS, FAT).

Key Point:

- A B tree of order m has at most m children and at most m – 1 keys in a node.
- Each non-root node has at least ⌈m/2⌉ children and ⌈m/2⌉ – 1 keys.

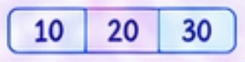
2. Structure and Properties

- Order (m) = maximum number of children.
- Maximum keys in a node = m – 1.
- Minimum keys in a non-root node = ⌈m/2⌉ – 1.
- Minimum children in a non-root node = ⌈m/2⌉.
- Root can have 2 to m children (if not leaf).
- All leaves are at the same depth.

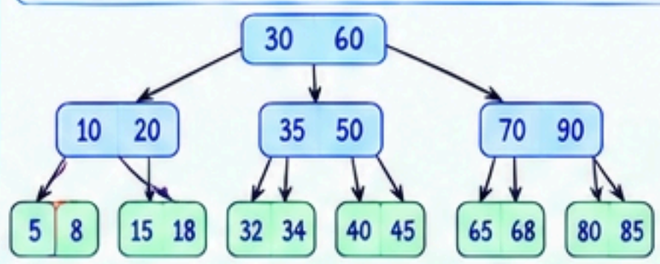
Example: B Tree of order 5 (m = 5)

- Max keys per node = 4
- Min keys (non-root) = 2
- Min children (non-root) = 3
- Root can have 2 to 5 children.

Node format:



(keys in sorted order)



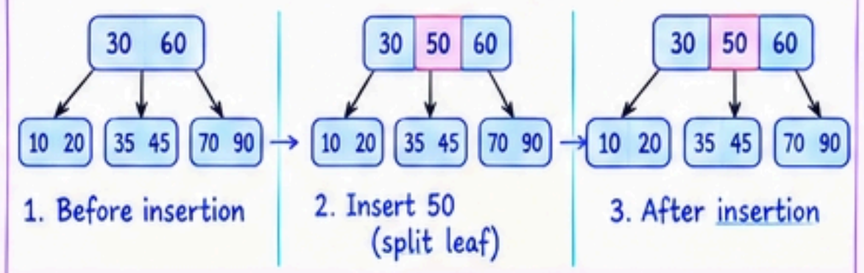
Here, m = 5 so each node can have max 4 keys and 5 children.

3. Insertion in B Tree

Steps:

1. Find the correct leaf node.
2. If the node has space, insert the key in sorted order.
3. If the node is full, split it and promote the middle key to its parent.
4. Repeat until the key is inserted.

Example: Insert 50 (order 5)



If a node becomes full, split and push the middle key up.

4. Deletion in B Tree

Steps:

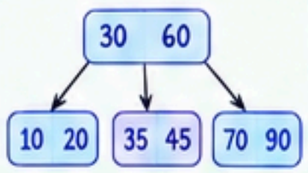
1. Find the key in the tree (leaf or internal node).
2. If it is in a leaf, remove it.
3. If it is in an internal node, replace it with its predecessor or successor (from leaf).
4. If the node has fewer than ⌈m/2⌉ keys, fix the underflow using:
   - Borrow from sibling (if possible).
   - Merge with sibling (if not possible).
5. Repeat until the tree satisfies all properties.

Important:

- After deletion, all nodes (except root) must have at least ⌈m/2⌉ keys (except root).
- If root becomes empty, its only child becomes the new root.

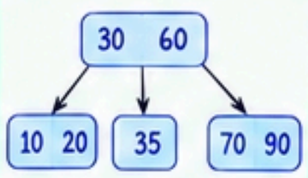
5. Example: Delete 45 (order 5)

1. Initial Tree



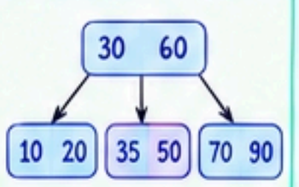
(45 is in a leaf)

2. Remove 45



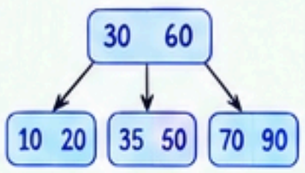
(leaf becomes underfull)

3. Borrow from sibling



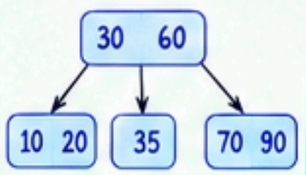
(borrow 50 from right sibling)

4. After borrowing



(node fixed)

5. Delete 50 (leaf)



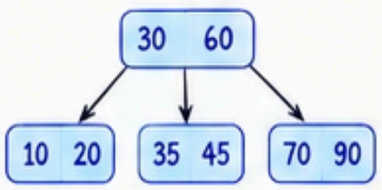
(still valid)

If borrowing is not possible, merge with a sibling and pull down the parent key.

If the root becomes empty, its only child becomes the new root.

6. Deletion (Merge Example)

Consider deleting 35 from the tree.



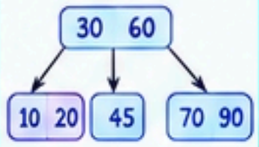
(35 is in internal node)

1. Replace 35 with its predecessor (20).
2. Delete 20 (leaf).
3. Node underflows → merge with sibling.
4. Adjust parent key.

After replace

(replace 35 with 20)

After delete & merge



(merge with sibling)

Final Tree

(balanced)

Key Takeaways:

- B tree is a multiway search tree with balanced height.
- Insertion: split full nodes and promote middle key.
- Deletion: handle underflow using borrow or merge.
- All leaves are at the same level.
- Each node (except root) has between ⌈m/2⌉ and m children.
- Used in databases, file systems and large-scale applications.

B+ Tree
(Balanced Tree for Efficient Searching)

1. What is a B+ Tree?

- A B+ Tree is a balanced tree data structure used for efficient searching, insertion and deletion.
- It is a variant of B Tree with two key features:

1. All actual data (records) are stored only in leaf nodes.
2. Internal nodes store only keys (no data).

- Leaf nodes are linked together in a linear linked list for sequential access.
- It is widely used in database and file systems (e.g., MySQL, file indexing).

2. Structure and Properties

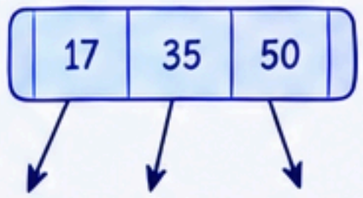
- Order (m): Maximum number of children per node.
- Internal node can have at most m children and at least $\lceil m/2 \rceil$ children (except root).
- Leaf node can have at most $m - 1$ keys and at least $\lceil (m-1)/2 \rceil$ keys (except root).
- All leaves are at the same level.
- Keys in a node are in sorted order.
- Leaf nodes are linked left to right.

For example, for order m = 4:

- Internal node: 2 to 4 children (1 to 3 keys)
- Leaf node: 2 to 4 keys

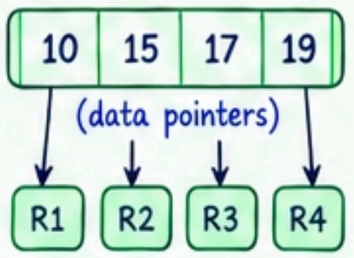
3. B+ Tree Node Types

Internal Node (index node)



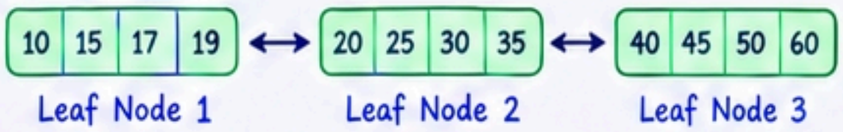
- Stores only keys.
- Points to child nodes.

Leaf Node

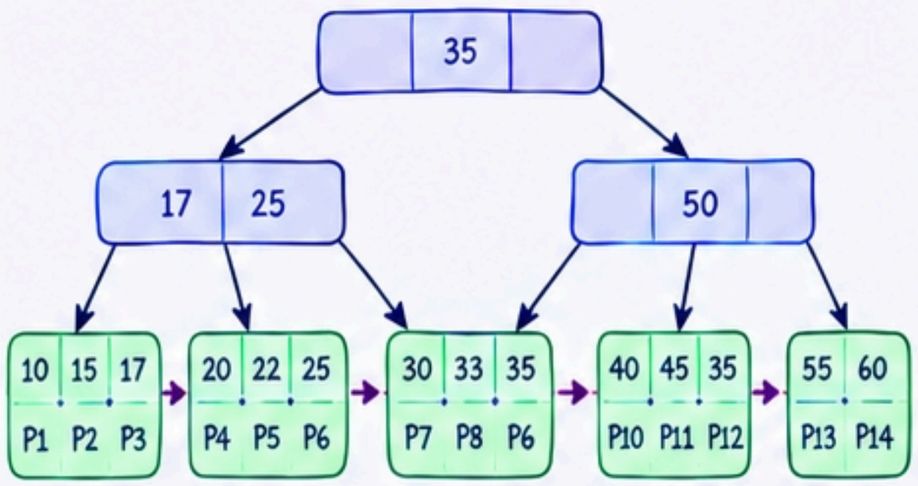


- Stores keys + record pointers.
- Linked with next leaf node.

Leaf Node Link



4. Example of B+ Tree (Order 4)



All data is in leaf nodes and leaves are linked.

5. Insertion in B+ Tree

1. Find the appropriate leaf node.
2. If there is space, insert the key in sorted order.
3. If the leaf is full, split the leaf and propagate the middle key to the parent.
4. If the parent is full, split the parent and propagate the middle key further.

Example: Insert 28

1. Goes to leaf [25, 28, 30] (full).
2. Split leaf → [25, 28] and [30].
3. Middle key 28 is inserted into parent.
4. If parent overflows, split and continue.

6. Deletion in B+ Tree

1. Find the key in the leaf node.
2. Remove the key from the leaf node.
3. If the leaf has at least $\lceil (m-1)/2 \rceil$ keys, stop (no rebalancing needed).
4. Otherwise, borrow from sibling (if possible) or merge with sibling.
5. If the parent becomes empty, adjust the parent (may need to merge/borrow at upper level).

Example: Delete 22

1. Remove 22 from leaf → [20, 25].
2. If still valid (enough keys), stop.
3. If not, borrow or merge and update parent.

7. Key Operations

- **Search:** Start at root, follow the correct child until a leaf node, then check the key.
- **Insert:** Find leaf, insert, split if needed, update parent.
- **Delete:** Find leaf, remove, rebalance (borrow/merge), update parent.

8. Advantages

- ✓ Faster search, insertion and deletion (balanced and shallow).
- ✓ All data in leaves → efficient range queries and sequential access.
- ✓ Linked leaves → easy to traverse in order.
- ✓ Widely used in databases and file systems.

9. Real-World Applications

Key Takeaways:

- B+ Tree = balanced tree + data only in leaves + linked leaves.
- Order m controls the number of children and keys per node.
- Used for efficient search, insertion and deletion.
- Ideal for large-scale and disk-based storage systems.

Red-Black Tree & Heap Trees

(Self-Balancing Search Tree | Complete Binary Tree with Heap Property)

1. What is a Red-Black Tree?

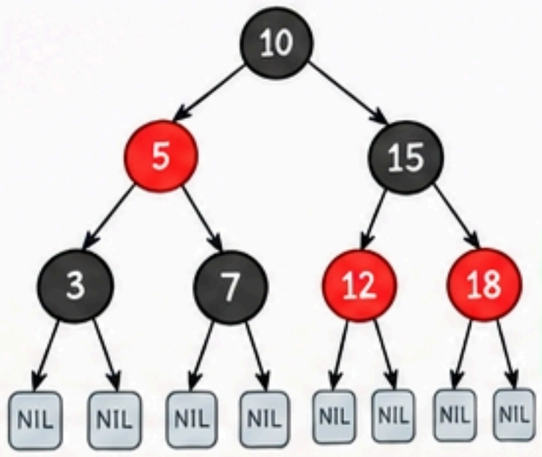
- A Red-Black Tree is a self-balancing binary search tree with nodes colored red or black.
- It maintains balance using specific coloring rules and rotations (after insertion/deletion).
- Used in applications like databases, memory management and standard libraries (e.g., C++ STL).

Key Point: It is a binary search tree + color rules + balancing properties.

2. Properties (Rules)

1. Each node is either Red or Black.
2. The root is always Black.
3. All leaves (NIL) are Black.
4. If a node is Red, both its children are Black (no two consecutive reds).
5. For each node, every path from the node to a leaf (NIL) has the same number of Black nodes (Black-height).

3. Example of a Red-Black Tree



= Red node
= Black node
= NIL (Black)

All paths from root to NIL have the same number of black nodes (Black-height = 2).

4. Key Operations

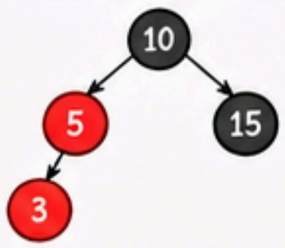
1. Search
- Like a normal BST (O(log n)).

2. Insert
- Insert as in BST, then fix color/rotation (O(log n)).

3. Delete
- Remove node, then fix using recoloring/rotation (O(log n)).

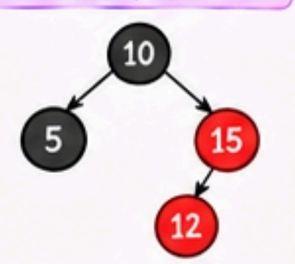
5. Balancing Cases (Insertion Example)

Case 1: Recoloring



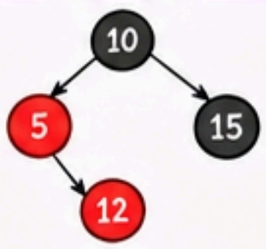
If uncle is red → recolor (parent, uncle, grandparent).

Case 2: Left Rotation



If node is right child of left parent → left rotate.

Case 3: Right Rotation



If node is left child of right parent → right rotate.

6. Time Complexity

- Search : O(log n)
- Insert : O(log n)
- Delete : O(log n)

(Height is always O(log n) due to self-balancing.)

1. What is a Heap Tree?

- A Heap is a complete binary tree that satisfies the heap property.
- It is commonly used for priority queues and heap sort.
- Two types: Max Heap and Min Heap.

2. Types of Heap

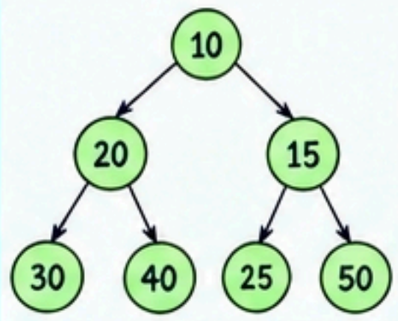
Max Heap
- Parent ≥ children
- Root contains the maximum value.

Min Heap
- Parent ≤ children
- Root contains the minimum value.

3. Properties

1. Complete binary tree (all levels filled from left to right).
2. Heap property is maintained (max or min).
3. Height is O(log n).

4. Example of a Min Heap



- Complete binary tree
- Parent ≤ children (min heap property)
- Smallest element at the root.

5. Array Representation

For a complete binary tree, we can store it in an array.

Example (Min Heap):

10	20	15	30	40	25	50
Index: 0		2	3	4	5	6

For node at index i:
- Parent = (i - 1) / 2
- Left child = 2i + 1
- Right child = 2i + 2

6. Operations

1. Insertion (Min Heap)
1. Add new element at end (maintain complete tree).
2. Percolate up (swap with parent if smaller).

Time: O(log n)

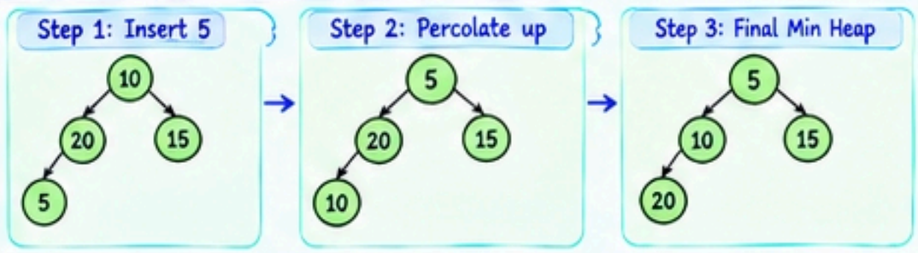
2. Deletion (Min Heap)
1. Remove root (minimum).
2. Replace with last element.
3. Percolate down (swap with smaller child).

Time: O(log n)

3. Heapify (Build Heap)
- Used to build heap from an array (bottom-up).
- Start from last non-leaf node and heapify down.

Time: O(n)

7. Example: Insertion in Min Heap



5 is smaller than 10, so it moves up to the root.

Key Takeaways:

- **Red-Black Tree:**
  - Self-balancing BST with color rules (red/black).
  - Guarantees O(log n) height.
  - Used in databases, memory management, C++ STL.
- **Heap Tree:**
  - Complete binary tree with heap property.
  - Max Heap / Min Heap.
  - Used for priority queues and heap sort.