Tree

(Non-Linear Data Structure)

Presenter:

Prashant Kumar Jha(26)
Prajwal Bhandari (24)
Prabhat Ale (22)
Prabin Adhikari (23)

What is a tree?

- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- consists of nodes with a parent-child relation.
- Applications:
- Organization charts
- File systems
- Programming environments

Tree

- In computer science , a tree is a widely used data structure that emulates a hierarchical tree structure with a set of linked nodes.

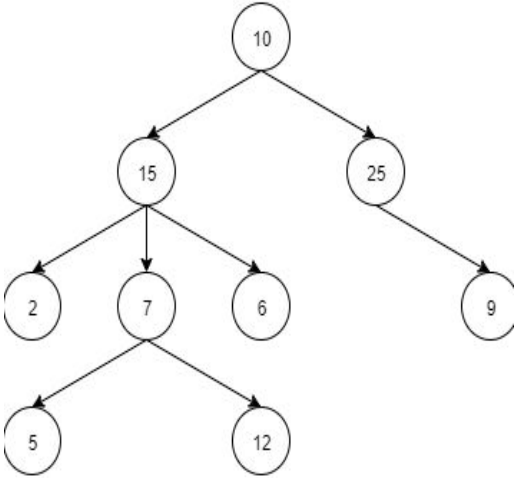
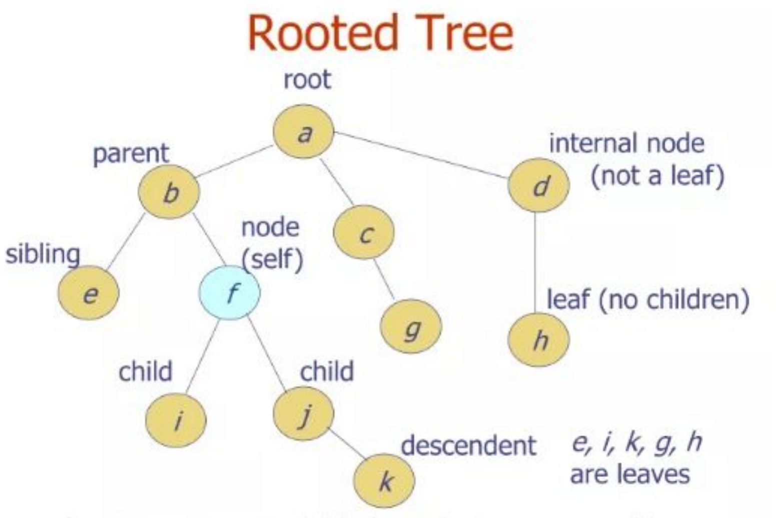


Fig: A tree strcuture

Tree Representation:



Contd..

- Root-The top most node
- Children
- Parent
- Sibling- Have same parent
- Leaf-Has no child

Tree Structure

Trees Data Structures

- Tree
  - Nodes
  - Each node can have 0 or more children
  - A node can have at most one parent
- Binary tree
  - Tree with 0-2 children per node

Terminology Used In Tree:

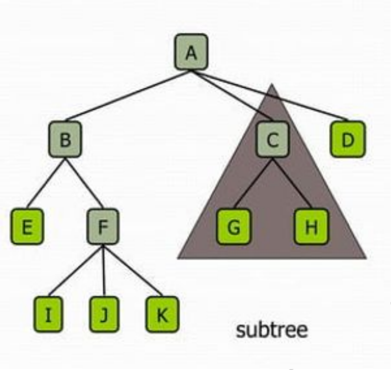
- Root = no parent
- Leaf = no child
- Interior = non-leaf
- Height = distance from root to leaf
- Path = Source Node to destination node
- Edge = link between two node

Contd..

- Path: Traversal from node to node along the edges results in a sequence called path.
- Root: Node at the top of the tree.
- Parent: Any node, except root has exactly one edge running upward to another node. The node above it is called parent.
- Child: Any node may have one or more lines running downward to other nodes. Nodes below are children.
- Leaf: A node that has no children.
- Subtree: Any node can be considered to be the root of a subtree, which consists of its children and its children's children and so on.

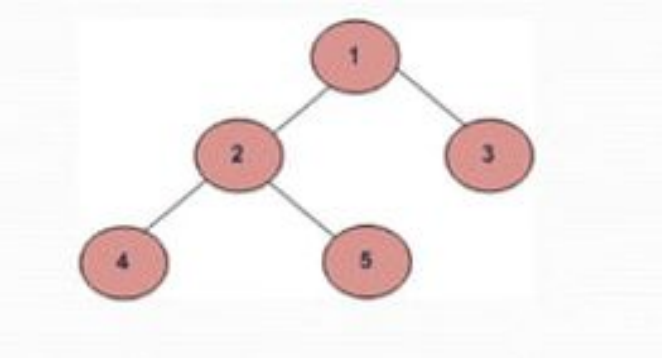
Contd..

- Root: node without parent (A)
- Siblings: nodes share the same parent
- Internal node: node with at least one child (A, B, C, F)
- External node (leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node(3)
- Degree of a node: the number of its children
- Degree of a tree the maximum number of its node.
- Subtree: tree consisting of a node and its descendant



Tree Traversal

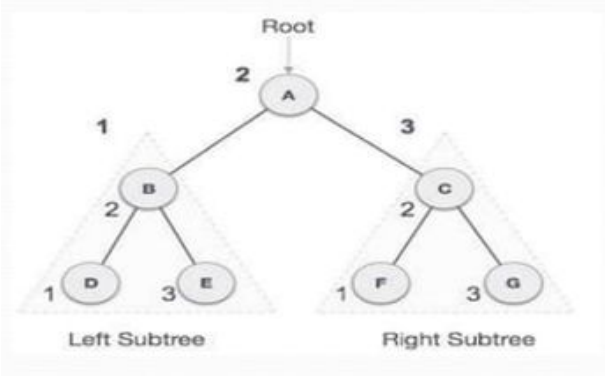
- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.
- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node.
- That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree



a) Inorder (Left, Root, Right): 42513
b) Preorder (Root, Left, Right): 12453
c) Postorder (Left, Right, Root): 45231

Contd..

- In-order Traversal(Left, Root, Right)
- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
- We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be D→B→E→A → F→C→G



Pre-Order Traversal

- Preorder (Root, Left, Right)
- We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be.

  A→B→D→E → C→F→G

Post-Order Traversal

- Post order (Left, Right, Root)
- We start from A, and following pre-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be.

  D→E→B→F→ G→C→A

Binary Tree

- Each node in the binary tree can have utmost two child nodes (left child and right child).
- Each node has both a left subtree and a right subtree (either of which may be empty). The node’s left (right) subtree consists of the node’s left (right) child together with that child’s own children, grandchildren, etc.
- Strictly Binary Tree: If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. A strictly binary tree with n leaves always contains 2n - 1 nodes.
-
- Complete Binary Tree: A complete binary tree of depth d is a strictly binary tree all of whose leaves are at level d.

Contd….

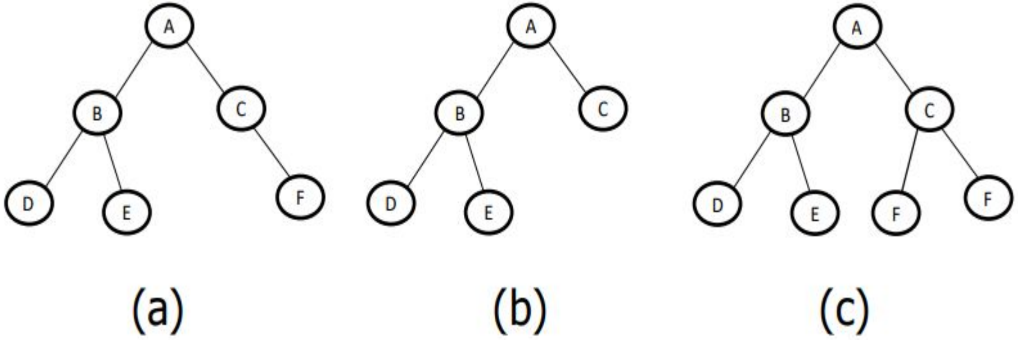


Fig: (a) Binary tree (b) Strictly binary tree (c) Complete binary tree

Contd…..

- If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l+1.
- A binary tree can contain at most 2^l nodes at level l.
- A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^l nodes at each level between 0 and d. That is, the binary tree of depth d contains exactly 2^d nodes at level d.
- The total number of nodes in a complete binary tree of depth d, tn, equals the sum of the number of nodes at each level between 0 and d. Thus

$$tn = 2^0 + 2^1 + \ldots + 2^d = \sum_{j=0}^{d} 2^j = 2^{d+1} - 1$$

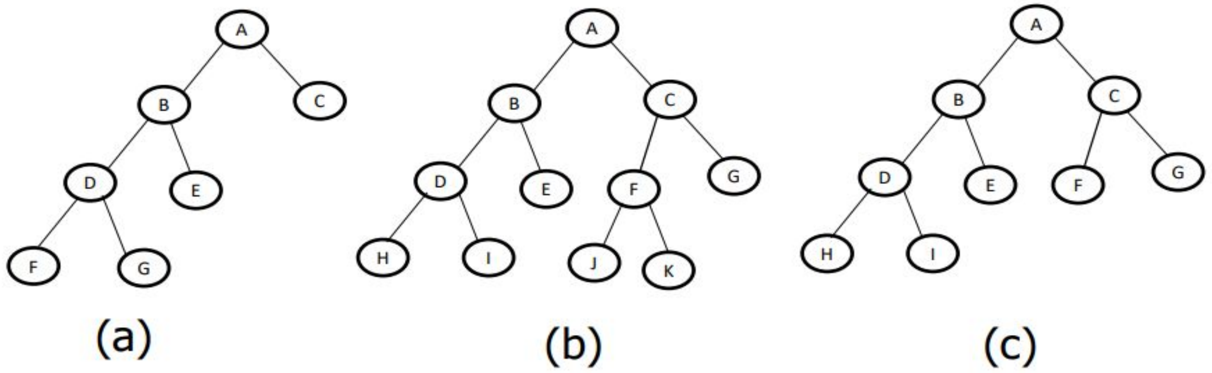
Contd...

- Since, all leaves in a complete binary tree are at level d, the tree contains 2^d leaves, and therefore, $2^d - 1$ nonleaf nodes.
- If the number of nodes, tn, in a complete binary tree is known, we can compute its depth

$$d = \log_2(tn + 1) - 1$$

- A **almost complete binary tree** is a binary tree in which (a) every level of the tree is completely filled except the last level. (b) Also, in the last level, nodes should be attached starting from left-most position.

Contd…



The strictly binary tree in figure (a) is not almost complete, since it violates condition a. The strictly binary tree of (b) is not almost complete since it satisfies condition a but violates condition b. The strictly binary tree of (c) satisfies both conditions a and b and is therefore an almost complete binary tree.

Binary Search Tree

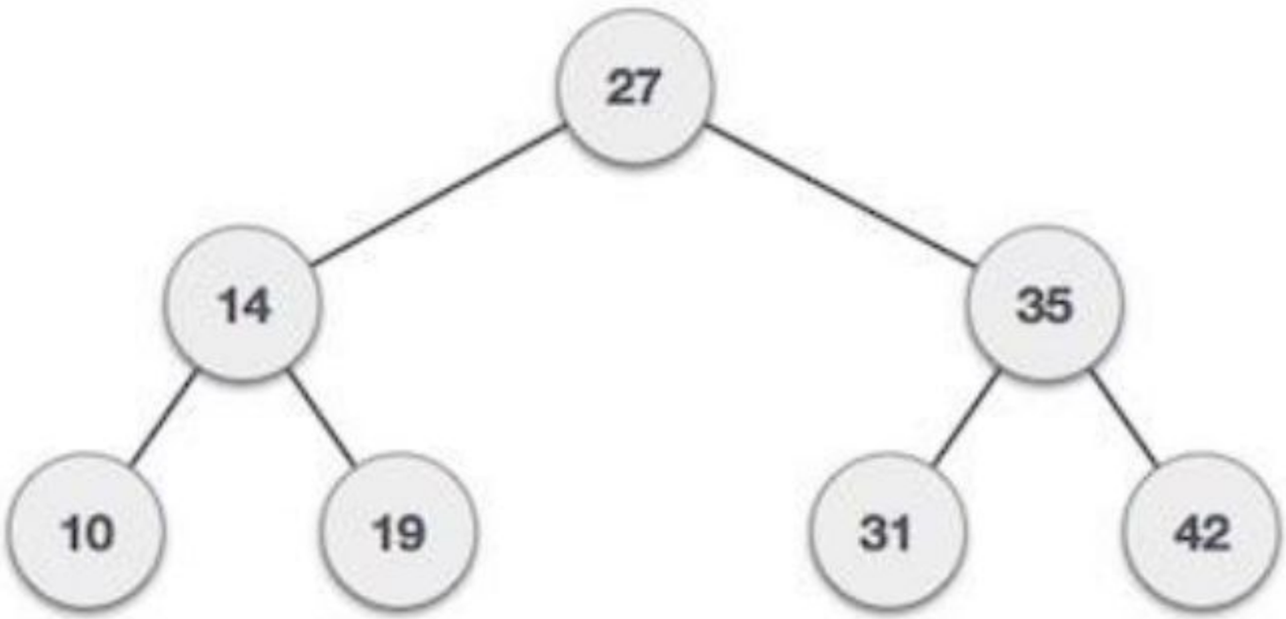
Prabhat Ale
22

Binary Search Tree

A **binary search tree** (or **BST**) is a binary tree with the following property. For any node in the binary tree, if that node contains information *info*:

- Its left subtree (if nonempty) contains only nodes with information less than *info*.
- Its right subtree (if nonempty) contains only nodes with information greater than *info*.

Examples Of Binary Search Tree



Searching In Binary Search Tree

Search: To search for a given target value in a binary search tree

- Compare the target value with the element in the root node.
- If the target value is **equal**, the search is successful.
- If target value is **less**, search the left subtree.
- If target value is **greater**, search the right subtree.
- If the subtree is **empty**, the search is unsuccessful.

Insertion In A Binary Search Tree

1. **Start at the Root**:
   - Begin at the root of the tree.
2. **Compare Value**:
   - Compare the value to be inserted with the value of the current node.
3. **Move Left or Right**:
   - If the value is less than the current node's value, move to the left child.
   - If the value is greater than the current node's value, move to the right child.
4. **Find the Right Spot**:
   - Repeat steps 2 and 3 until you find an empty spot (null) where the new value can be inserted.
5. **Insert the Value**:
   - Insert the new value at the found spot.

Deletion In A Binary Search Tree

1. **Start at the Root:**

   - Begin at the root of the tree.

2. **Find the Node to Delete:**

   - Use the same method as insertion to find the node with the value to be deleted.

3. **Case 1: Node has No Children (Leaf Node):**

   - Simply remove the node from the tree.

Deletion In A Binary Search Tree

4. **Case 2: Node has One Child**:
   - Remove the node and link its parent directly to its child.

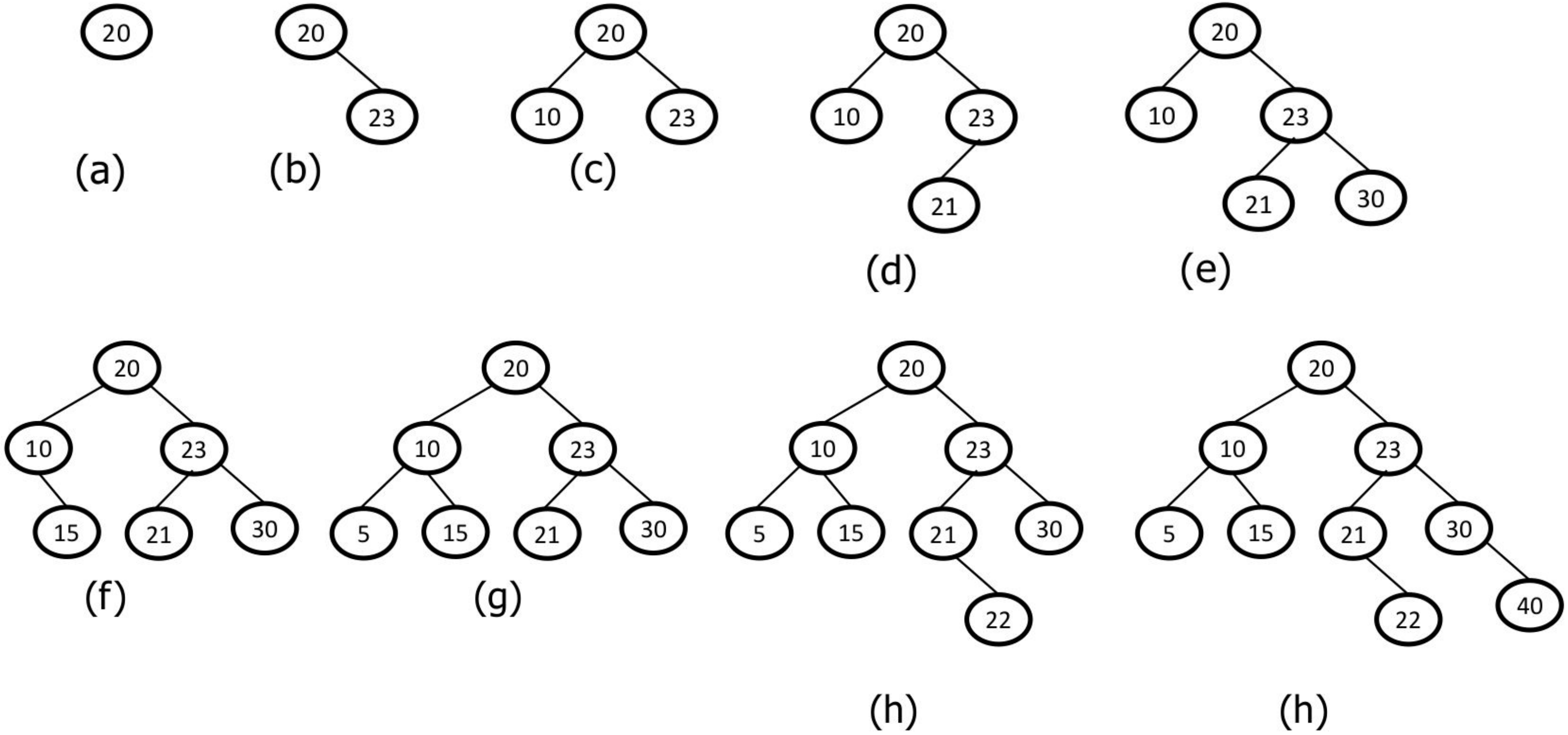
5. **Case 3: Node has Two Children**:
   - Find the node's in-order successor (the smallest value in the right subtree) or in-order predecessor (the largest value in the left subtree).
   - Replace the node's value with the successor's (or predecessor's) value.
   - Delete the successor (or predecessor) node, which will now be a simpler case (either a leaf or having one child).

Example:

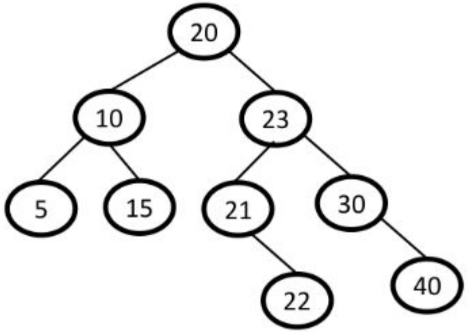
Example: Starting with empty BST, show the effect of successively adding the following numbers as keys: 20, 23, 10, 21, 30, 15, 5, 22 and 40. Also, show the effect of successively deleting 10, 30, 20 from the resulting BST.

Example:

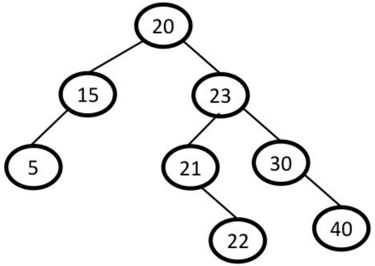
After successively adding 20, 23, 10, 21, 30, 15, 5, 22 and 40.



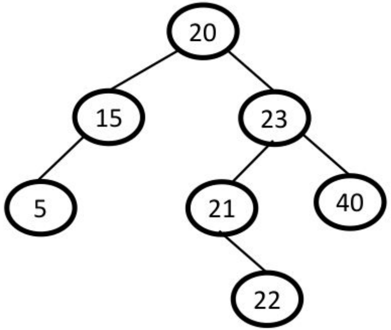
Example:



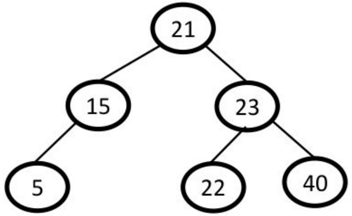
After deleting 10.



After deleting 30.



After deleting 20.



Time Complexity Analysis:

- **Analysis:** Let number of nodes in the binary search tree be n.
  - If the tree is balanced, the maximum number of comparisons for search, insert, and delete operations is $\lfloor \log_2 n \rfloor + 1$. Hence, best-case time complexity is $O(\log_2 n)$.
  - If the tree is not balanced, maximum number of comparisons search, insert, and delete operations is n. Hence, worst-case time complexity is $O(n)$.

Presentation on DSA

MDS 1st Semester

Topic: (unit-8)

AVL Trees

Prabin Adhikari
Roll No : 23

Definition

An AVL tree is a binary search tree with a ***balance* condition**.

AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis

AVL tree *approximates* the ideal tree (**completely balanced tree**).

AVL tree maintains a height close to the minimum.

In AVL tree, **balance factor** of every node is -1,0 or +1.

Where, Balance factor = height of left sub tree – height of right sub tree.

Every AVL tree is binary search tree, but every binary search tree need not to be AVL tree.

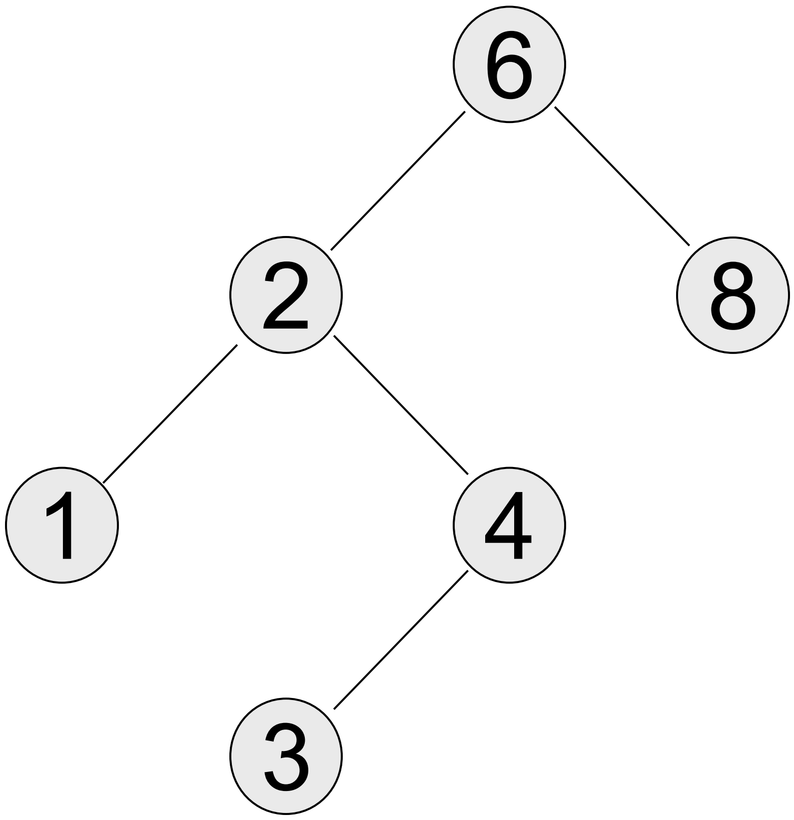
Operation perform as search, insertion, deletion with **O(log n) time complexity.**

.

Balance condition ?
Height ?

An AVL tree is a binary search tree such that for any node in the tree, the height of the left and right subtrees can differ by at most 1.

- Height is the length of the longest path from root to a leaf
- Here in the figure height of the tree is 3, subtree rooted by node 2 is 2, by 8 is 0. Tree is unbalanced



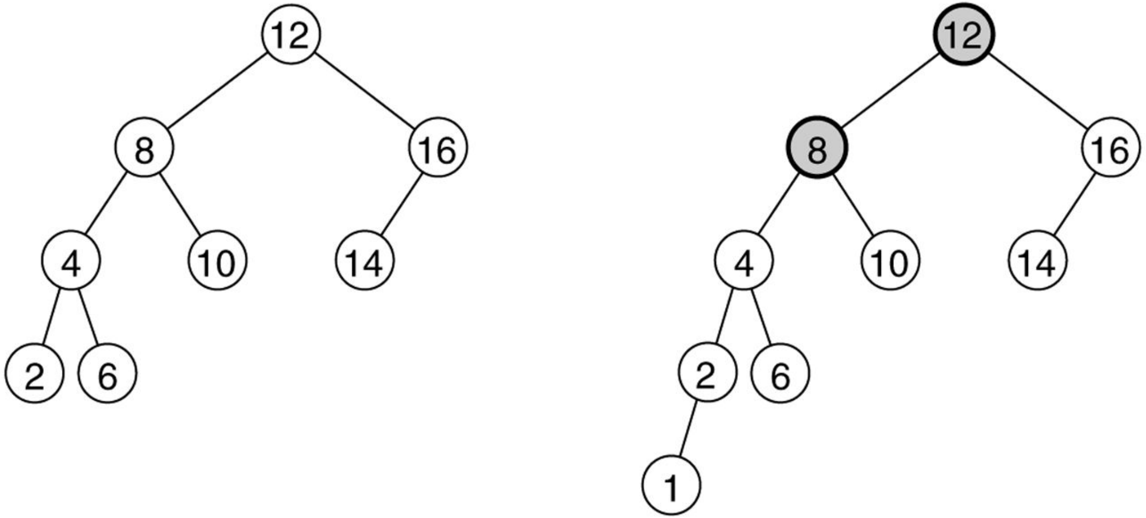
Examples

Two binary search trees:

(a) an AVL tree

(b) not an AVL tree

(unbalanced nodes are darkened)



(a) (b)

Properties of AVL Tree

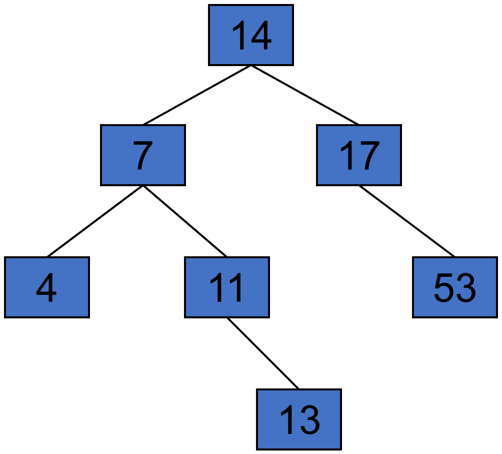
- The depth of a typical node in an AVL tree is very close to the optimal *log N*.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.

- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.

- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

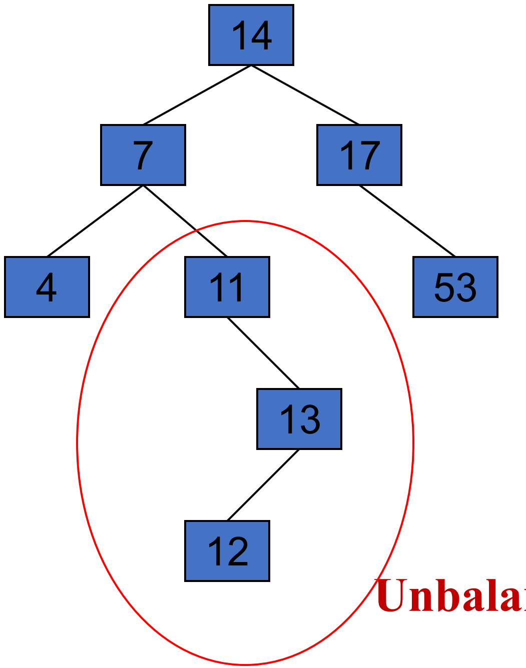
Simple Insertion operation on AVL Tree (Example)

Insert 14, 17, 7, 53, 4,11,13 into an empty AVL tree



Balanced AVL Tree

Now insert 12

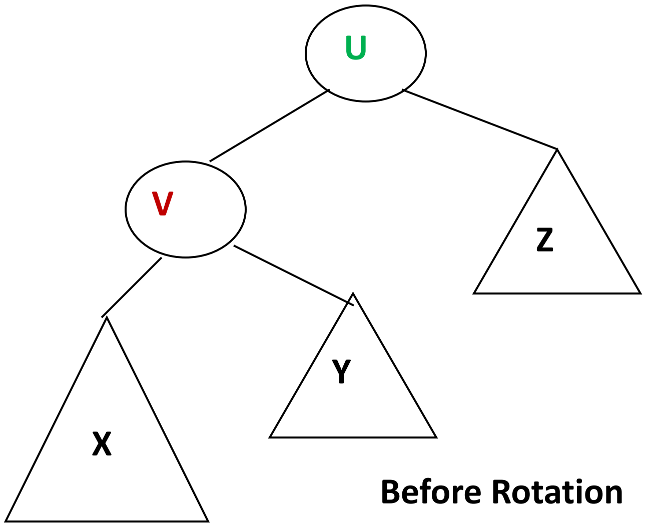


Unbalanced (let’s balance)

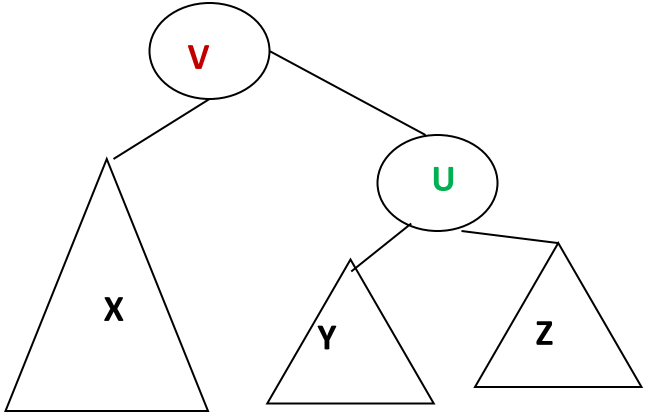
For Balancing the Tree, Single and Double rotation is used

Insertion in left child of left subtree

Single Rotation



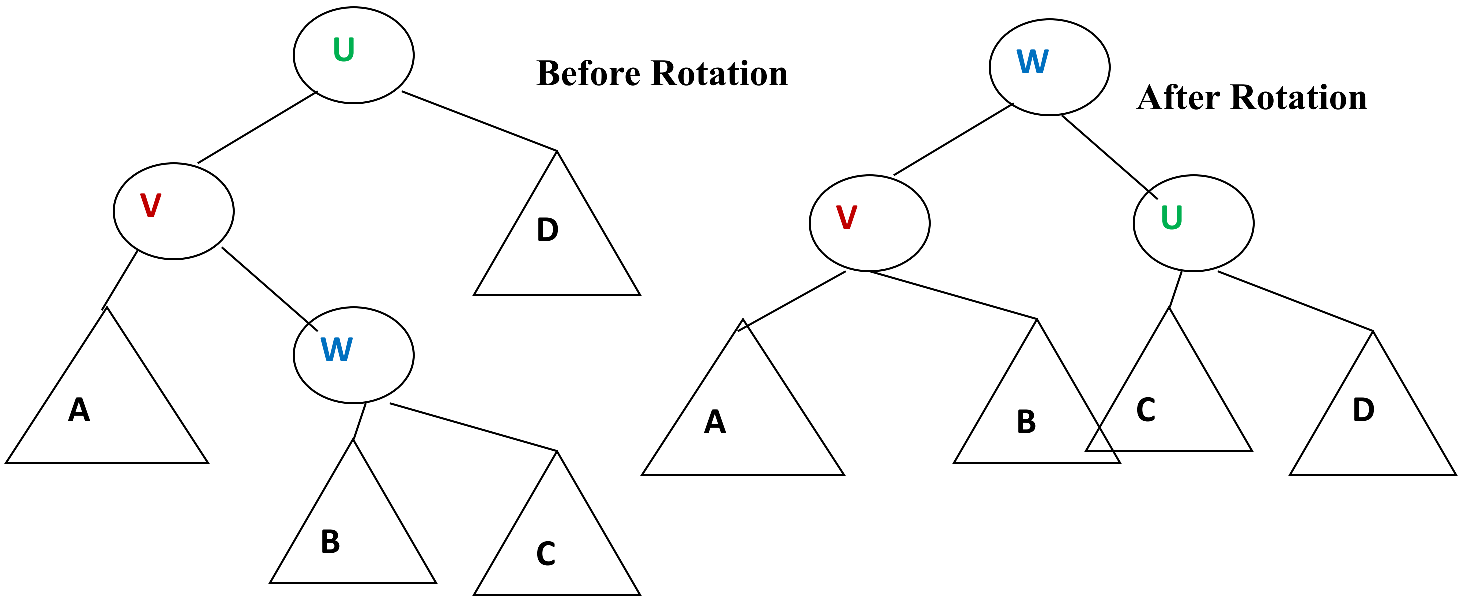
Before Rotation



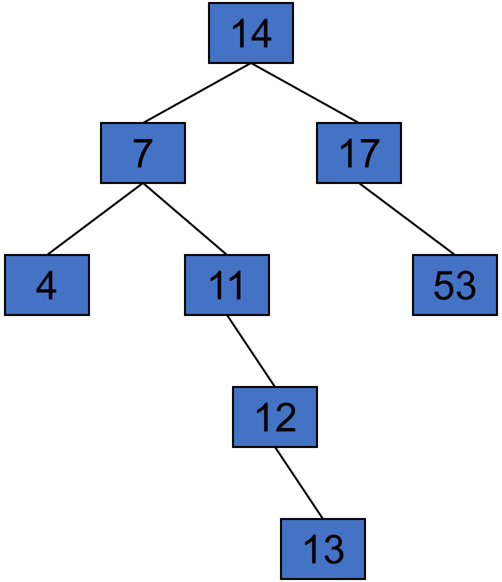
After Rotation

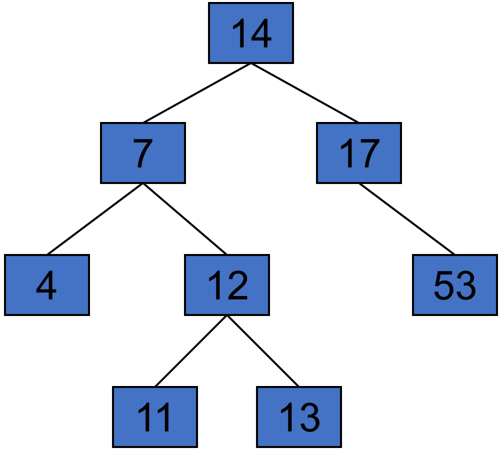
Double Rotation

Suppose, imbalance is due to an insertion in the left subtree of right child
Single Rotation does not work!



Above Example





Finally Balanced

Insertion operation Example (step wise)

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

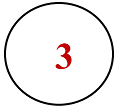


Fig 1

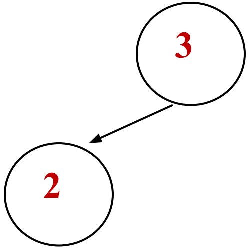


Fig 2

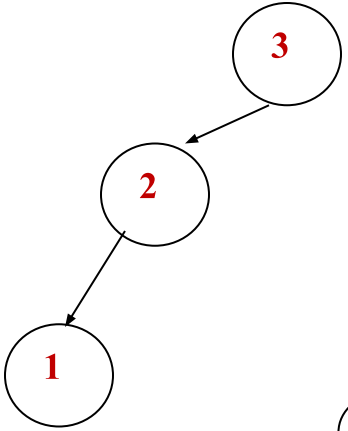


Fig 3

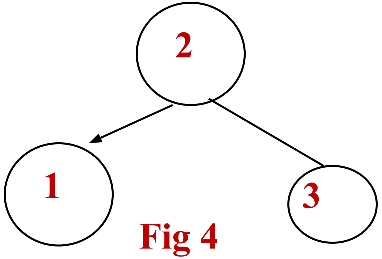


Fig 4

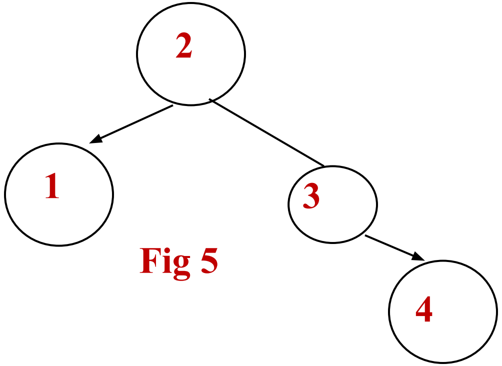


Fig 5

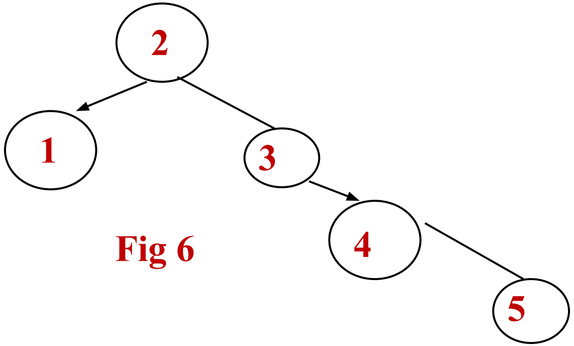


Fig 6

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

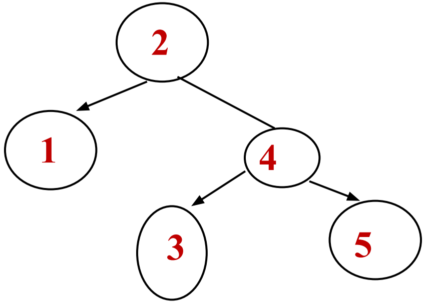


Fig 7

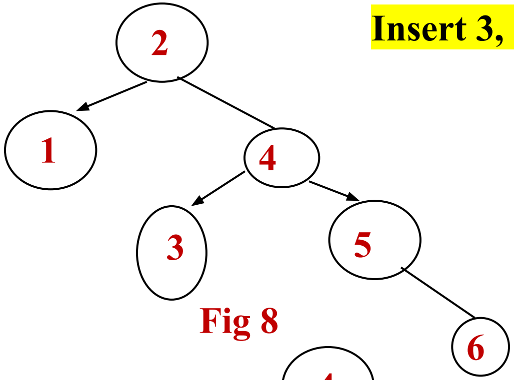


Fig 8

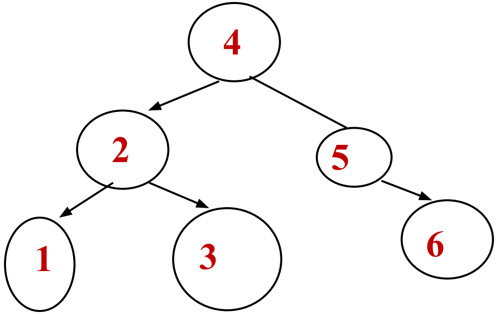


Fig 9

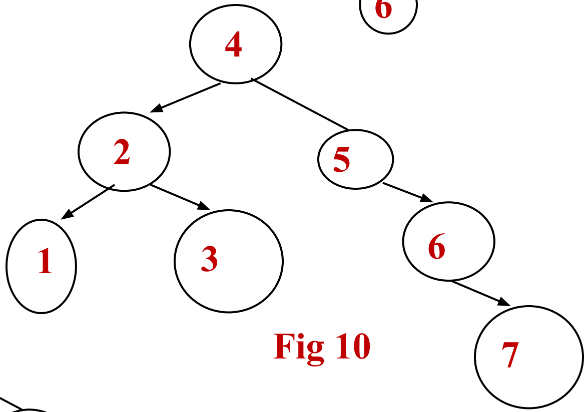


Fig 10

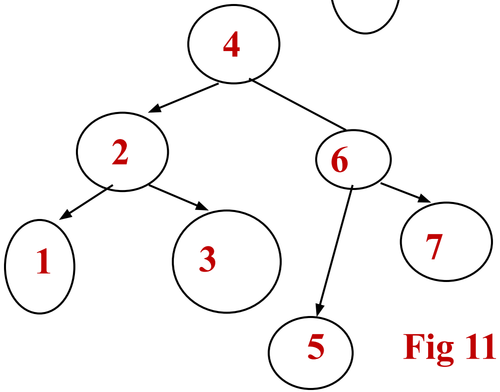


Fig 11

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

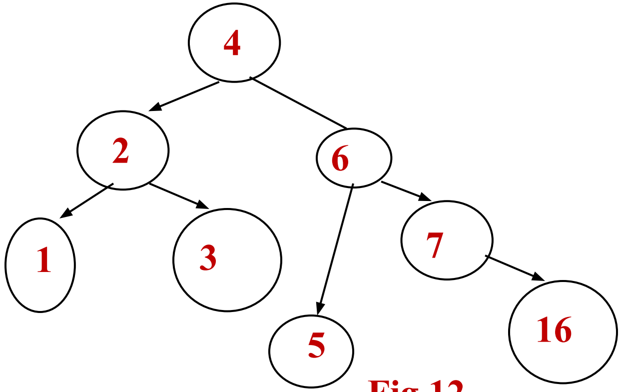


Fig 12

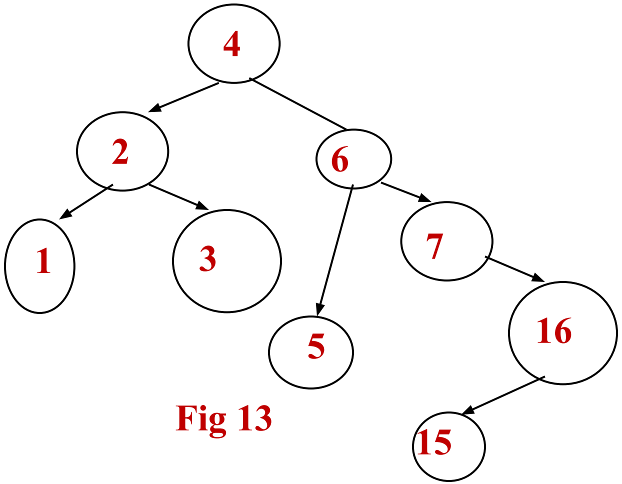


Fig 13

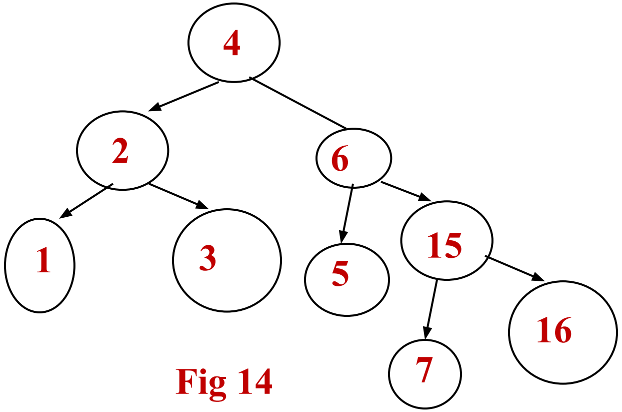


Fig 14

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

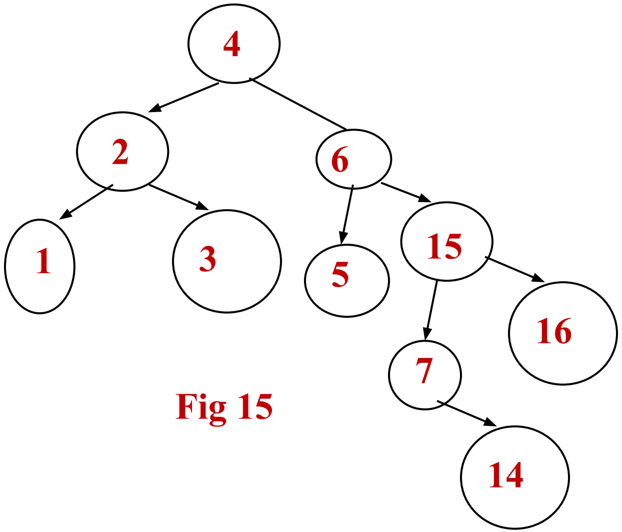


Fig 15

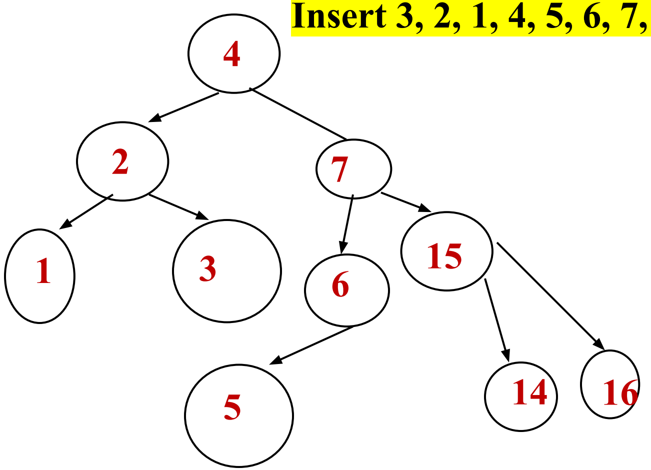
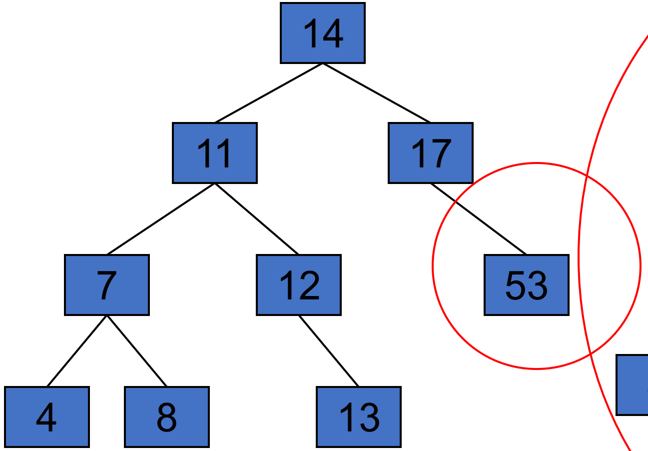


Fig 16

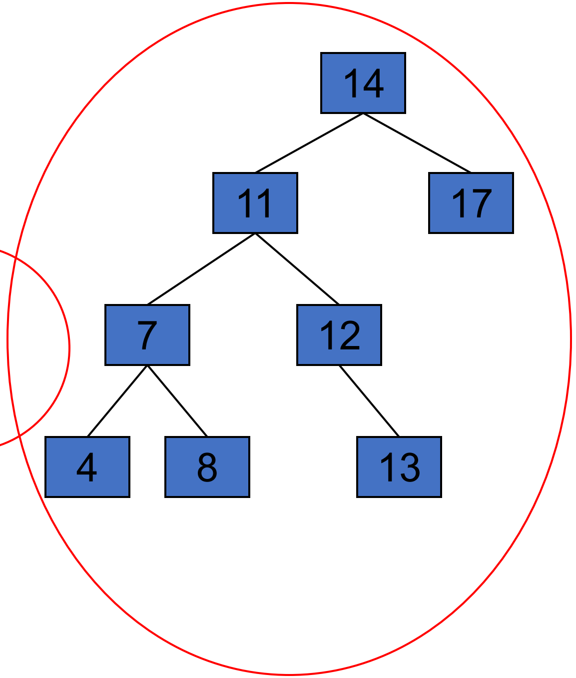
- Deletions also can be done with similar rotations

Example of Deletion Operation from below AVL Tree

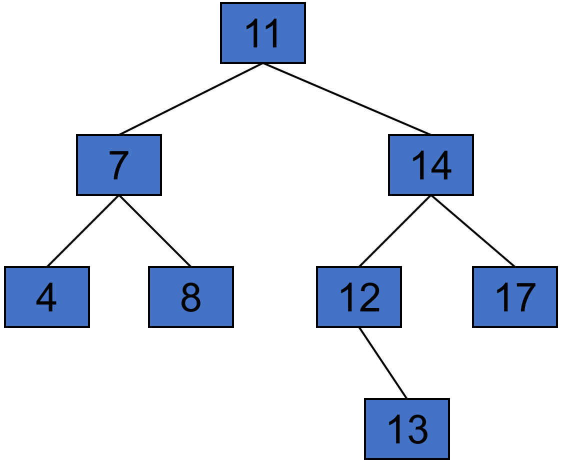
Now remove 53



Now remove 53, unbalanced



Balanced!



Applications

Databases:
- Indexing
- Query optimization

Memory Management:
- Dynamic memory allocation

File Systems:
- Directory management
- Metadata management

Networking:
- Routing tables
- Prefix matching

Compilers:
- Syntax trees
- Symbol tables

Gaming:
- Collision detection
- Leaderboard management

Geographic Information Systems (GIS):
- Spatial data indexing
- Map data management

Cryptography:
- Digital certificates

Artificial Intelligence:
- Decision trees
- Search algorithms

Financial Systems:
- Transaction management
- Order matching

Thank You !!