Binary Tree (Properties):

1. **The maximum number of nodes at level ‘l’ of a binary tree is 2l-1**.
2. **Maximum number of nodes in a binary tree of height ‘h’ is *2h – 1*.**

***3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is  ⌈ Log2(N+1) ⌉***

***4) A Binary Tree with L leaves has at least   ⌈ Log2L ⌉ + 1   levels***

***5) In Binary tree, number of leaf nodes is always one more than nodes with two children***.

L = T + 1

Where L = Number of leaf nodes

T = Number of internal nodes with two children

Following are common types of Binary Trees.

**Full Binary Tree** A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.

***In a Full Binary, number of leaf nodes is number of internal nodes plus 1***  
       L = I + 1  
Where L = Number of leaf nodes, I = Number of internal nodes

**Complete Binary Tree:** A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

**Perfect Binary Tree** A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

1. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). [Self-balancing search trees](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)like [AVL](http://en.wikipedia.org/wiki/AVL_tree) and [Red-Black trees](http://en.wikipedia.org/wiki/Red-black_tree)guarantee an upper bound of O(Logn) for search.
2. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). [Self-balancing search trees](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)like [AVL](http://en.wikipedia.org/wiki/AVL_tree) and [Red-Black trees](http://en.wikipedia.org/wiki/Red-black_tree) guarantee an upper bound of O(Logn) for insertion/deletion.

// A utility function to search a given key in BST

public Node search(Node root, int key)

{

    // Base Cases: root is null or key is present at root

    if (root==null || root.key==key)

        return root;

    // val is greater than root's key

    if (root.key > key)

        return search(root.left, key);

    // val is less than root's key

    return search(root.right, key);

}

Asdjbksdabsdkazbdemsadnnsadm,annncmxz,cnmznxcmzxncmncznmnvxcvxvvxx