# Binary Tree | Set 1 (Introduction)

**Trees:** Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

**Tree Vocabulary:**The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, a is a child of f and f is the parent of a. Finally, elements with no children are called leaves.

tree

----

j <-- root

/ \

f k

/ \ \

a h z <-- leaves

**Why Trees?**  
**1.** One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

-----------

/ <-- root

/ \

... home

/ \

ugrad course

/ / | \

... cs101 cs112 cs113

**2.** Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).  
**3.** Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).  
**4.** Like Linked Lists and unlike Arrays, Trees don’t have an upper limit on number of nodes as nodes are linked using pointers.

**Main applications of trees include:**  
**1.** Manipulate hierarchical data.  
**2.** Make information easy to search (see tree traversal).  
**3.** Manipulate sorted lists of data.  
**4.** As a workflow for compositing digital images for visual effects.  
**5.**Router algorithms  
**6.**Form of a multi-stage decision-making (see business chess).

**Binary Tree:** A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

**Binary Tree Representation in C:**A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.  
A Tree node contains following parts.  
1. Data  
2. Pointer to left child  
3. Pointer to right child

In C, we can represent a tree node using structures. Below is an example of a tree node with an integer data.

S

* Binary Tree | Set 2 (Properties)
* We have discussed [Introduction to Binary Tree in set 1](http://quiz.geeksforgeeks.org/binary-tree-set-1-introduction/). In this post, properties of binary are discussed.
* ***1) The maximum number of nodes at level ‘l’ of a binary tree is 2l-1***.  
  Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.  
  This can be proved by induction.  
  For root, l = 1, number of nodes = 21-1 = 1  
  Assume that maximum number of nodes on level l is 2l-1  
  Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. 2 \* 2l-1
* ***2) Maximum number of nodes in a binary tree of height ‘h’ is 2h – 1***.  
  Here height of a tree is maximum number of nodes on root to leaf path. Height of a leaf node is considered as 1.  
  This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is 1 + 2 + 4 + .. + 2h-1. This is a simple geometric series with h terms and sum of this series is 2h – 1.  
  In some books, height of a leaf is considered as 0. In this convention, the above formula becomes 2h+1 – 1
* ***3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is  ⌈ Log2(N+1) ⌉***  
  This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes   ⌈ Log2(N+1) ⌉ – 1
* ***4) A Binary Tree with L leaves has at least   ⌈ Log2L ⌉ + 1   levels***  
  A Binary tree has maximum number of leaves when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.
* L <= 2l-1 [From Point 1]
* Log2L <= l-1
* l >= ⌈ Log2L ⌉ + 1
* ***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
* See [Handshaking Lemma and Tree](http://www.geeksforgeeks.org/handshaking-lemma-and-interesting-tree-properties/) for proof.
* In the next article on tree series, we will be discussing [different types of Binary Trees and their properties](http://quiz.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/).
* Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
* Binary Tree | Set 3 (Types of Binary Tree)
* We have discussed [Introduction to Binary Tree in set 1](http://quiz.geeksforgeeks.org/binary-tree-set-1-introduction/) and [Properties of Binary Tree in Set 2](http://quiz.geeksforgeeks.org/binary-tree-set-2-properties/). In this post, common types of binary is discussed.
* 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.
* 18
* / \
* 15 30
* / \ / \
* 40 50 100 40
* 18
* / \
* 15 20
* / \
* 40 50
* / \
* 30 50
* 18
* / \
* 40 30
* / \
* 100 40
* ***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  
  See [Handshaking Lemma and Tree](http://www.geeksforgeeks.org/handshaking-lemma-and-interesting-tree-properties/) for proof.
* **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
* Following are examples of Complete Binary Trees
* 18
* / \
* 15 30
* / \ / \
* 40 50 100 40
* 18
* / \
* 15 30
* / \ / \
* 40 50 100 40
* / \ /
* 8 7 9
* Practical example of Complete Binary Tree is [Binary Heap](http://quiz.geeksforgeeks.org/binary-heap/).
* **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.  
  Following are examples of Perfect Binaryr Trees.
* 18
* / \
* 15 30
* / \ / \
* 40 50 100 40
* 18
* / \
* 15 30
* A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has 2h – 1 node.
* Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.
* **Balanced Binary Tree**  
  A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes. For Example, AVL tree maintain O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.
* **A degenerate (or pathological) tree**A Tree where every internal node has one child. Such trees are performance-wise same as linked list.
* 10
* /
* 20
* \
* 30
* \
* 40

# Handshaking Lemma and Interesting Tree Properties

**What is Handshaking Lemma?**  
[Handshaking lemma](http://en.wikipedia.org/wiki/Handshaking_lemma) is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even. The handshaking lemma is a consequence of the degree sum formula (also sometimes called the handshaking lemma)

[handshaking](http://d1gjlxt8vb0knt.cloudfront.net/wp-content/uploads/handshaking.png)

**How is Handshaking Lemma useful in Tree Data structure?**  
Following are some interesting facts that can be proved using Handshaking lemma.

***1) In a k-ary tree where every node has either 0 or k children, following property is always true.***

L = (k - 1)\*I + 1

Where L = Number of leaf nodes

I = Number of internal nodes

**Proof:**  
Proof can be divided in two cases.

**Case 1**(Root is Leaf):There is only one node in tree. The above formula is true for single node as L = 1, I = 0.

**Case 2**(Root is Internal Node): For trees with more than 1 nodes, root is always internal node. The above formula can be proved using Handshaking Lemma for this case. A tree is an undirected acyclic graph.

Total number of edges in Tree is number of nodes minus 1, i.e., |E| = L + I – 1.

All internal nodes except root in the given type of tree have degree k + 1. Root has degree k. All leaves have degree 1. Applying the Handshaking lemma to such trees, we get following relation.

Sum of all degrees = 2 \* (Sum of Edges)

Sum of degrees of leaves +

Sum of degrees for Internal Node except root +

Root's degree = 2 \* (No. of nodes - 1)

Putting values of above terms,

L + (I-1)\*(k+1) + k = 2 \* (L + I - 1)

L + k\*I - k + I -1 + k = 2\*L + 2I - 2

L + K\*I + I - 1 = 2\*L + 2\*I - 2

K\*I + 1 - I = L

(K-1)\*I + 1 = L

So the above property is proved using Handshaking Lemma, let us discuss one more interesting property.

   
   
***2) 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

**Proof:**  
Let number of nodes with 2 children be T. Proof can be divided in three cases.

**Case 1:** There is only one node, the relationship holds  
as T = 0, L = 1.

**Case 2:** Root has two children, i.e., degree of root is 2.

Sum of degrees of nodes with two children except root +

Sum of degrees of nodes with one child +

Sum of degrees of leaves + Root's degree = 2 \* (No. of Nodes - 1)

Putting values of above terms,

(T-1)\*3 + S\*2 + L + 2 = (S + T + L - 1)\*2

Cancelling 2S from both sides.

(T-1)\*3 + L + 2 = (S + L - 1)\*2

T - 1 = L - 2

T = L - 1

***Case 3:***Root has one child, i.e., degree of root is 1.

Sum of degrees of nodes with two children +

Sum of degrees of nodes with one child except root +

Sum of degrees of leaves + Root's degree = 2 \* (No. of Nodes - 1)

Putting values of above terms,

T\*3 + (S-1)\*2 + L + 1 = (S + T + L - 1)\*2

Cancelling 2S from both sides.

3\*T + L -1 = 2\*T + 2\*L - 2

T - 1 = L - 2

T = L - 1

Therefore, in all three cases, we get T = L-1.

We have discussed proof of two important properties of Trees using Handshaking Lemma. Many GATE questions have been asked on these properties, following are few links.

# Enumeration of Binary Trees

A Binary Tree is labeled if every node is assigned a label and a Binary Tree is unlabeled if nodes are not assigned any label.

Below two are considered same unlabeled trees

o o

/ \ / \

o o o o

Below two are considered different labeled trees

A C

/ \ / \

B C A B

**How many different Unlabeled Binary Trees can be there with n nodes?**

For n = 1, there is only one tree

o

For n = 2, there are two trees

o o

/ \

o o

For n = 3, there are five trees

o o o o o

/ \ / \ / \

o o o o o o

/ \ \ /

o o o o

The idea is to consider all possible pair of counts for nodes in left and right subtrees and multiply the counts for a particular pair. Finally add results of all pairs.

For example, let T(n) be count for n nodes.

T(0) = 1 [There is only 1 empty tree]

T(1) = 1

T(2) = 2

T(3) = T(0)\*T(2) + T(1)\*T(1) + T(2)\*T(0) = 1\*2 + 1\*1 + 2\*1 = 5

T(4) = T(0)\*T(3) + T(1)\*T(2) + T(2)\*T(1) + T(3)\*T(0)

= 1\*5 + 1\*2 + 2\*1 + 5\*1

= 14

The above pattern basically represents [n’th Catalan Numbers](http://www.geeksforgeeks.org/program-nth-catalan-number/). First few catalan numbers are 1 1 2 5 14 42 132 429 1430 4862,…  
[catalan](http://d30wr2otswzun8.cloudfront.net/wp-content/uploads/2015/12/catalan.gif)  
Here,  
T(i-1) represents number of nodes on the left-sub-tree  
T(n−i-1) represents number of nodes on the right-sub-tree

n’th Catalan Number can also be evaluated using direct formula.

T(n) = (2n)! / (n+1)!n!

Number of Binary Search Trees (BST) with n nodes is also same as number of unlabeled trees. The reason for this is simple, in BST also we can make any key as root, If root is i’th key in sorted order, then i-1 keys can go on one side and (n-i) keys can go on other side.

**How many labeled Binary Trees can be there with n nodes?**  
To count labeled trees, we can use above count for unlabeled trees. The idea is simple, every unlabeled tree with n nodes can create n! different labeled trees by assigning different permutations of labels to all nodes.

Therefore,

Number of Labeled Tees = (Number of unlabeled trees) \* n!

= [(2n)! / (n+1)!n!] × n!

For example for n = 3, there are 5 \* 3! = 5\*6 = 30 different labeled trees

**Reference:**  
<http://qa.geeksforgeeks.org/4343/many-labeled-unlabeled-binary-trees-are-possible-with-nodes/>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# Applications of tree data structure

**Difficulty Level:**Rookie

**Why Tree?**  
Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system  
———–

/ <-- root

/ \

... home

/ \

ugrad course

/ / | \

... cs101 cs112 cs113

2) 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.

3) 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.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don’t have an upper limit on number of nodes as nodes are linked using pointers.

[As per Wikipedia](http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Common_uses), following are the common uses of tree.  
1. Manipulate hierarchical data.  
2. Make information easy to search (see tree traversal).  
3. Manipulate sorted lists of data.  
4. As a workflow for compositing digital images for visual effects.  
5. Router algorithms