

# Tree (Part 1)





# Basics

---

- A tree  $T$  is a finite, non-empty set of nodes,  $T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_n$ ,
- with the following properties:
  - A designated node of the singleton,  $r$ , is called the root of the tree.
  - The remaining nodes are partitioned into  $n \geq 0$  subsets  $T_1, T_2, \dots, T_n$ , each of which is a tree.



# Definition

---

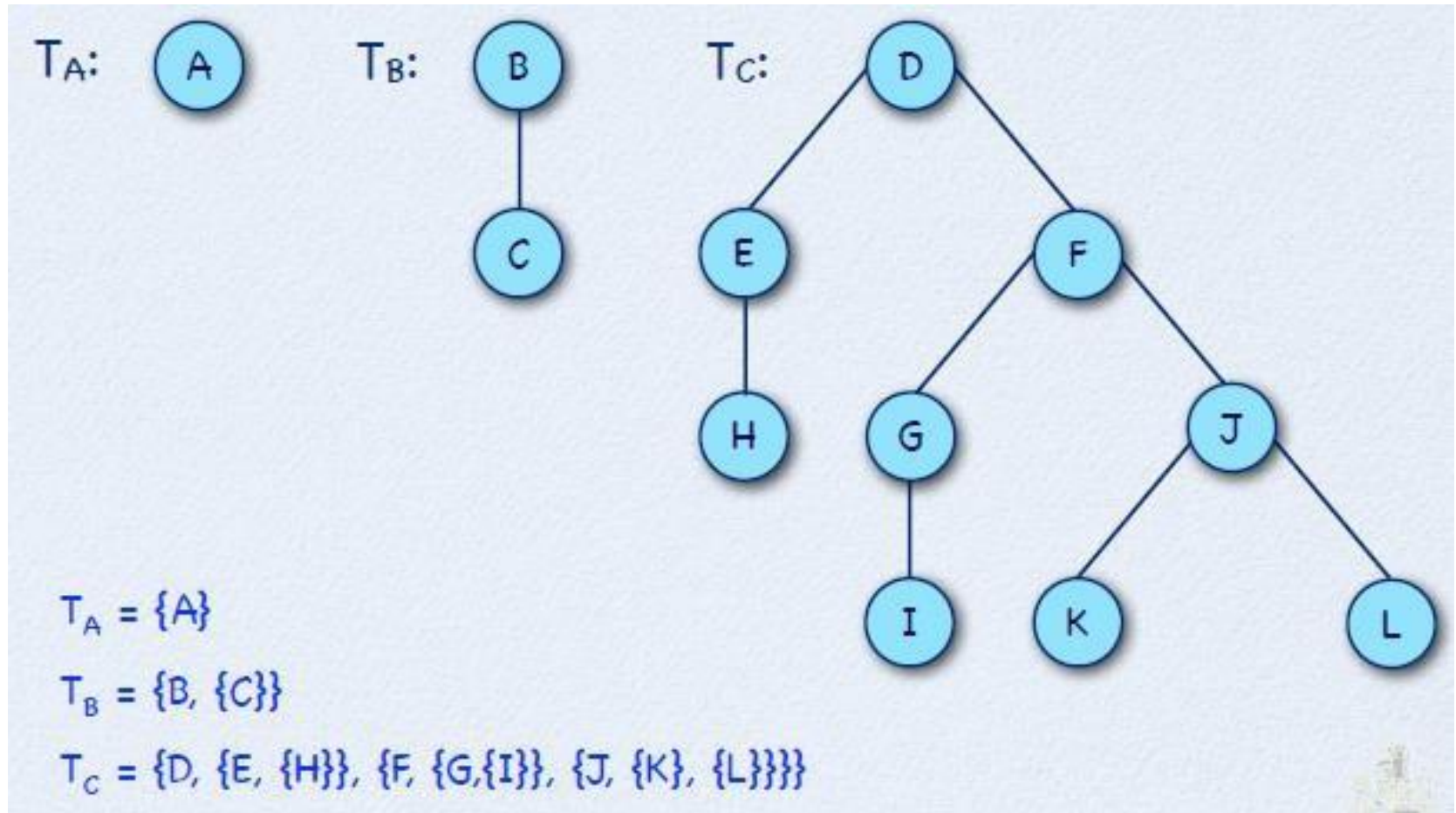
## ■ Tree

- ☐ undirected graph
- ☐ any two nodes are connected by exactly one edge
- ☐ no loop

**Why are trees in computer science generally drawn upside down?**

<https://www.quora.com/Why-are-trees-in-computer-science-generally-drawn-upside-down-from-how-trees-are-in-real-life>

# Tree Examples





# Parent and Children

---

- The root node  $r$  of tree  $T$  is the parent of all the roots  $r_i$  of the subtrees  $T_i$ ,  $1 < i \leq n$ .
- Each root  $r_i$  of subtree  $T_i$  of tree  $T$  is called a child of  $r$ .



# Degree

---

- The **degree** of a node is the number of subtrees associated with that node.
- A node of degree zero has no subtrees. Such a node is called a **leaf**.
- Two roots  $r_i$  and  $r_j$  of distinct subtrees  $T_i$  and  $T_j$  with the same parent in tree  $T$  are called **siblings**.

# Structure of a Tree

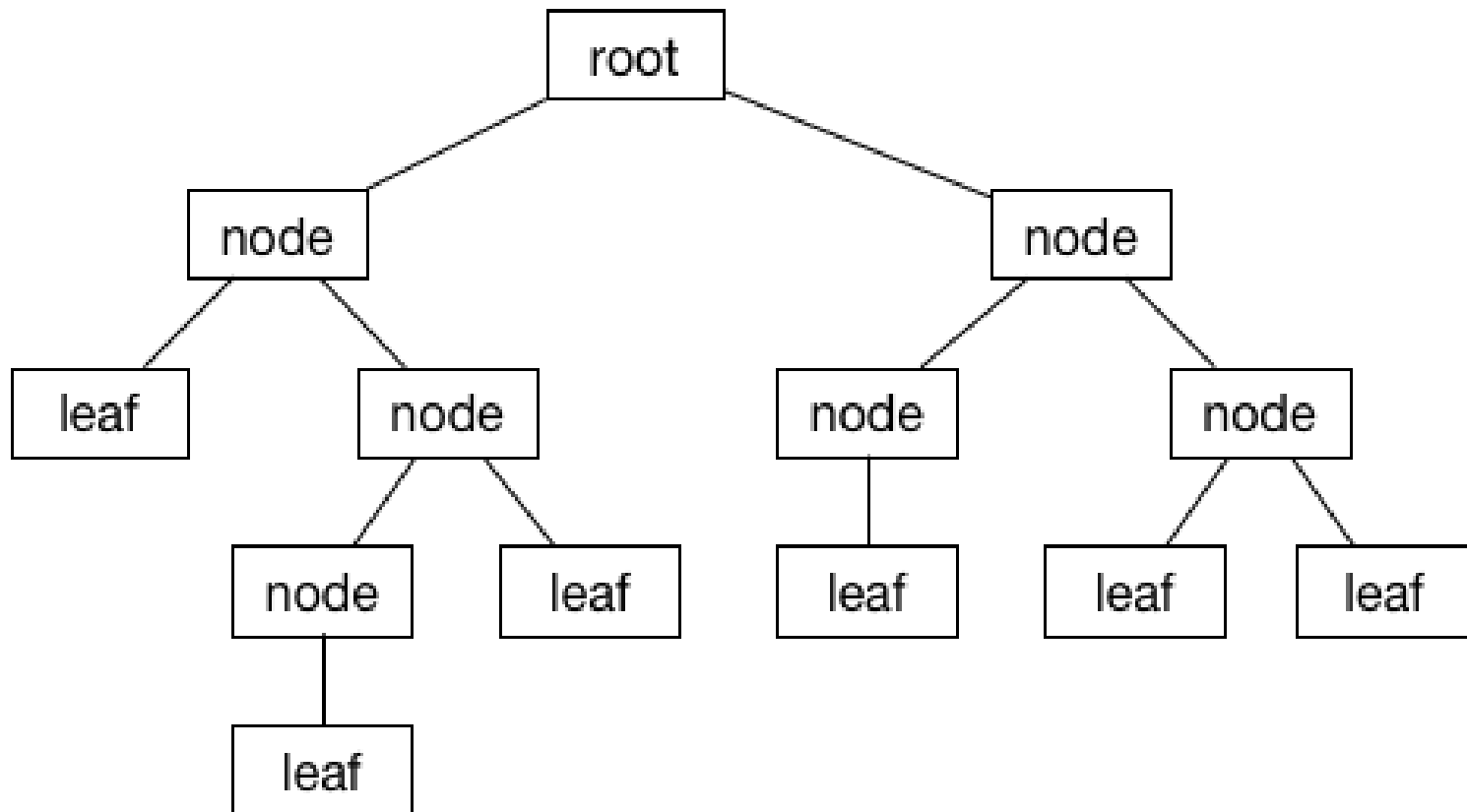


Figure from: <https://www.raywenderlich.com/990-swift-algorithm-club-swift-binary-search-tree-data-structure>



# Path and Path Length

- Given a tree  $T$  containing the set of nodes  $R$ , a **path** in  $T$  is defined as a non-empty sequence of nodes.

- $P = \{r_1, r_2, \dots, r_k\}$

where  $r_i \in R$ , for  $1 \leq i \leq k$  such that the  $i^{\text{th}}$  node in the sequence,  $r_i$ , is the parent of the  $(i+1)^{\text{th}}$  node in the sequence  $r_{i+1}$ .

- The length of path  $P$  is  $k-1$ .

**$\in$  symbol means “is an element of”**





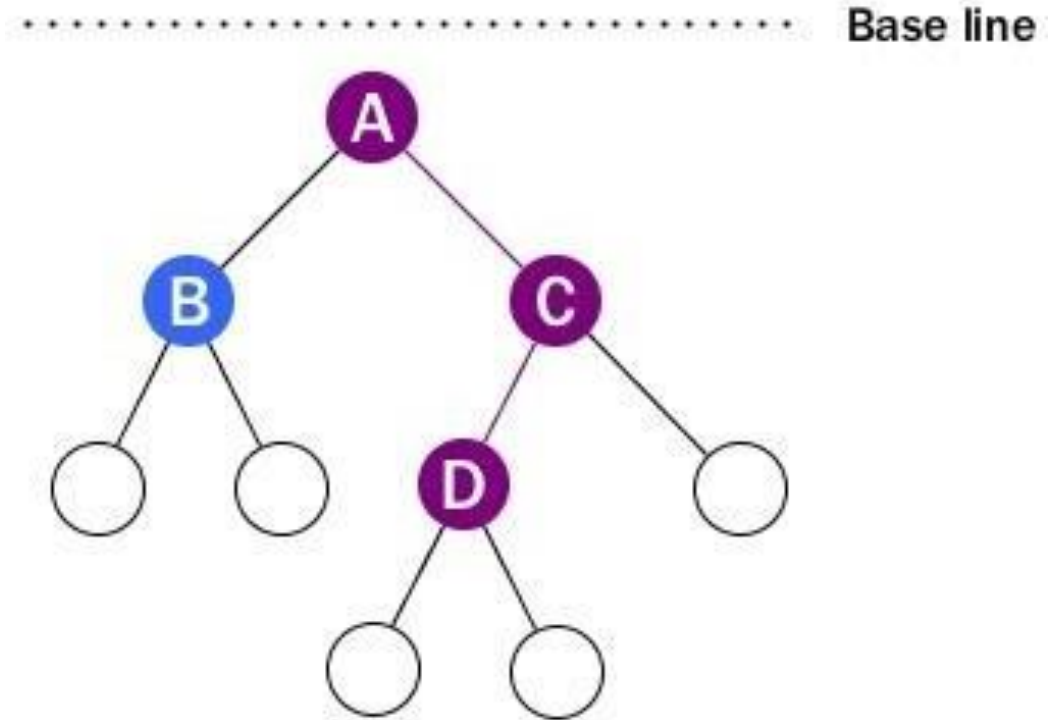
# Depth, Height and Level

- The **depth** of a node is the number of edges from the node to the **tree's** root node.
  - A root node will have a **depth** of 0.
- The **height of a node** is the number of edges on **the longest path** from the node to a leaf.
  - *Height of tree* – The height of a tree is the number of edges on the longest downward path between the root and a leaf.
  - A leaf node will have a **height** of 0.
- The **level of a node** is defined by 1 + the number of connections between the node and the root.

$$\text{i.e. level} = \text{depth} + 1$$

# Example

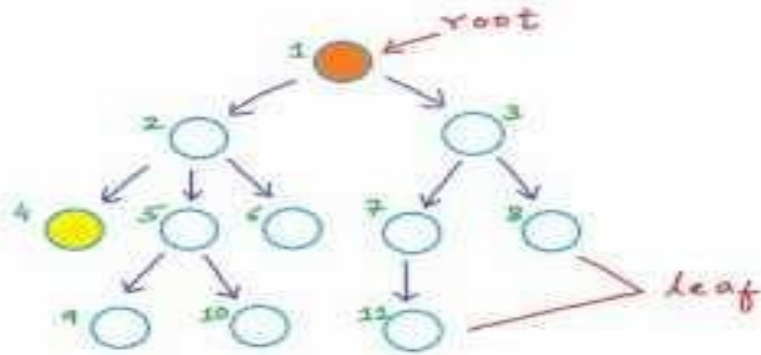
- The height of the tree is 3
- The height of Node C is 2 (count from bottom)
- The depth of Node D is 2 (count from top)



# Introduction to Trees



## Introduction to Trees



root  
children  
Parent  
Sibling → have same parent  
leaf → has no child

mycodeschool.com

# Nodes With the Same Degree

---



- The general case allows each node in a tree to have a different degree (The number of subtrees of a node is called the **degree** of the node). We now consider a variant of trees in which each node has the same degree.
- Unfortunately, it is not possible to construct a tree that has a finite number of nodes, which all have the same degree  $N$ , except the trivial case  $N = 0$ .
- We need a special notion, called empty tree, to realize trees in which all nodes have the same degree.



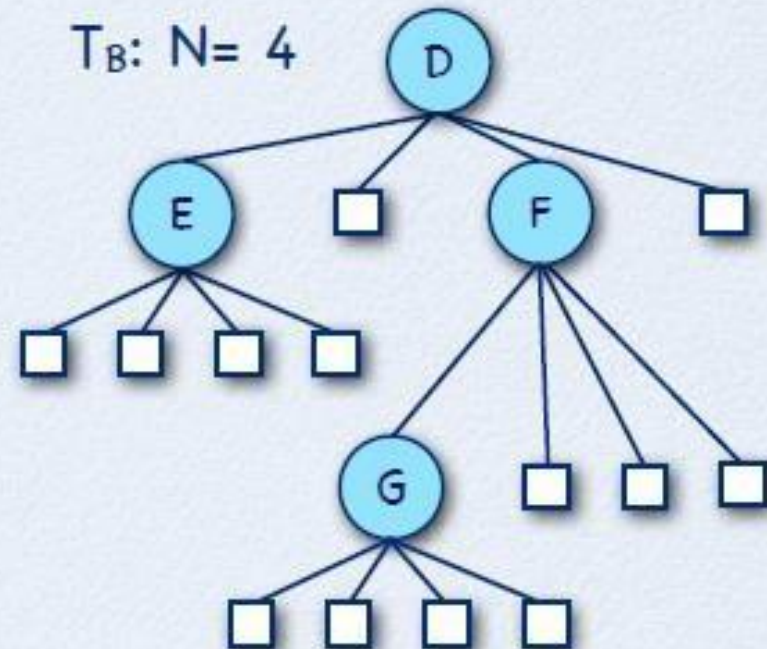
# N-ary tree

---

- An N-ary tree  $T$ ,  $N \geq 1$ , is a finite set of nodes with one of the following properties:
  - Either the set is empty,  $T = \emptyset$ , or
  - The set consists of a root,  $R$ , and exactly  $N$  distinct
- N-ary trees, That is, the remaining nodes are partitioned into  $N \geq 1$  subsets,  $T_1, T_2, \dots, T_N$ , each of which is an N-ary tree such that

$$T = \{R, T_1, T_2, \dots, T_N\}.$$

# N-ary Tree Examples



# Ntree Class



```
NTree.h
6  template<class T, int N>
7  class NTree
8  {
9  private:
10     const T* fKey;           // 0 for empty NTree
11     NTree<T,N>* fNodes[N];
12
13     NTree();                 // empty NTree
14
15 public:
16     static NTree<T,N> NIL;    // sentinel
17
18     NTree( const T& aKey );    // root with N subtrees
19     ~NTree();
20
21     bool isEmpty() const;
22     const T& key() const;
23     NTree& operator[] ( int aIndex ) const;
24     void attachNTree( int aIndex, NTree<T,N>* aNTree );
25     NTree* detachNTree( int aIndex );
26 };
27
```

Line: 4 Column: 16 C++ Tab Size: 4 NTREE.H

# The Private Empty NTree Constructor

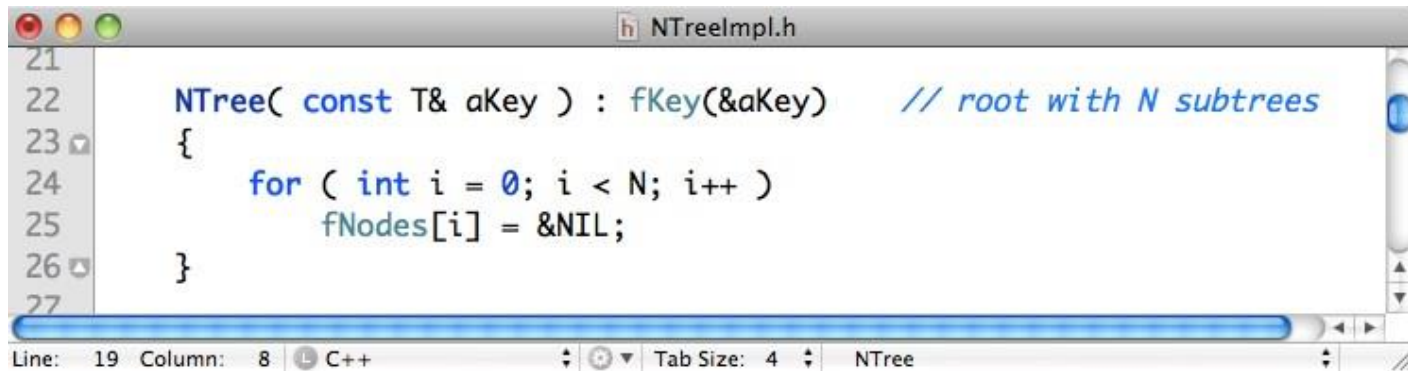


```
NTreeImpl.h
13
14     NTree() : fKey((T*)0)           // empty NTree
15     {
16         for ( int i = 0; i < N; i++ )
17             fNodes[i] = &NIL;
18     }
19
Line: 21 Column: 1 C++ Tab Size: 4 NTree
```

- We use  $(T^*)0$ , the null pointer, to initialize the fKey with a suitable value of type pointer to T.
- Each subtree-node is set to the location of NIL, the sentinel node for NTree.
- This constructor is only used to set up the sentinel for NTree.



# The Public NTree Constructor



```
21
22 NTree( const T& aKey ) : fKey(&aKey)    // root with N subtrees
23 {
24     for ( int i = 0; i < N; i++ )
25         fNodes[i] = &NIL;
26 }
27
```

Line: 19 Column: 8 C++ Tab Size: 4 NTree

- We store the address of the reference aKey in fKey.
- Each child node in a non-empty NTree leaf node is set to the location of NIL, the sentinel node for NTree.

In computer programming, a **sentinel node** is a specifically designated **node** used with linked lists and **trees** as a traversal path terminator. This type of **node** does not hold or reference any data managed by the data structure.

# The NTree Destructor



```
NTreempl.h
27
28 ~NTree()
29 {
30     for ( int i = 0; i < N; i++ )
31         if ( fNodes[i] != &NIL )
32             delete fNodes[i];
33 }
34
```

Line: 20 Column: 5 C++ Tab Size: 4 NTree

- In the destructor of NTree only non-sentinel nodes are destroyed.



# The NTree Sentinel

```
93  
94 template<class T, int N>  
95 NTree<T, N> NTree<T, N>::NIL;  
96
```

Line: 20 Column: 5 C++ Tab Size: 4 NTree

- Static instance variables, like the NTree sentinel NIL, need to be initialized **outside** the class definition.
- Here, NIL is initialized using the private default constructor.
- The scope of NIL is Ntree. It means that all members of Ntree are available, including the private constructor to initialize NIL.

# The NTree Auxiliaries



```
h NTreeImpl.h
35 bool isEmpty() const
36 {
37     return this == &NIL;
38 }
39
40 const T& key() const
41 {
42     if ( isEmpty() )
43         throw std::domain_error( "Empty NTree!" );
44
45     return *fKey;
46 }
47
```

Line: 32 Column: 1 C++ Tab Size: 4 ~NTree

# Attaching a New Subtree



```
NTreeImpl.h
60
61 void attachNTree( int aIndex, NTree<T,N>* aNTree )
62 {
63     if ( isEmpty() )
64         throw std::domain_error( "Empty NTree!" );
65
66     if ( (aIndex >= 0) && (aIndex < N) )
67     {
68         if ( fNodes[aIndex] != &NIL )
69             throw std::domain_error( "Non-empty subtree present!" );
70
71         fNodes[aIndex] = aNTree;
72     }
73     else
74         throw std::out_of_range( "Illegal subtree index!" );
75 }
76
```

Line: 20 Column: 5 C++ Tab Size: 4 NTree

# Accessing a Subtree (Operator [])



```
NTreeImpl.h
47
48 NTree& operator[]( int aIndex ) const
49 {
50     if ( isEmpty() )
51         throw std::domain_error( "Empty NTree!" );
52
53     if ( (aIndex >= 0) && (aIndex < N) )
54     {
55         return *fNodes[aIndex];    // return reference to subtree
56     }
57     else
58         throw std::out_of_range( "Illegal subtree index!" );
59 }
60
```

Line: 42 Column: 25 C++ Tab Size: 4 key

- We return a reference to the subtree rather than a pointer. This way, we prevent accidental manipulations outside the tree structure.

# Removing a Subtree



```
NTreeImpl.h
77 NTree* detachNTree( int aIndex )
78 {
79     if ( isEmpty() )
80         throw std::domain_error( "Empty NTree!" );
81
82     if ( (aIndex >= 0) && (aIndex < N) )
83     {
84         NTree<T,N>* Result = fNodes[aIndex];
85         fNodes[aIndex] = &NIL;
86         return Result;           // return subtree
87     }
88     else
89         throw std::out_of_range( "Illegal subtree index!" );
90 }
91
```

Line: 44 Column: 1 C++ Tab Size: 4 key



# A NTree Example



```
NTreeTest.cpp

8  int main()
9  {
10     string s1( "Hello World!" );
11     string s2( "A" );
12     string s3( "B" );
13     string s4( "C" );
14
15     NTree<string,3> A3Tree( s1 );
16
17     NTree<string,3> STree1( s2 );
18     NTree<string,3> STree2( s3 );
19     NTree<string,3> STree3( s4 );
20
21     A3Tree.attachNTree( 0, &STree1 );
22     A3Tree.attachNTree( 1, &STree2 );
23     A3Tree.attachNTree( 2, &STree3 );
24
25     cout << "Key: " << A3Tree.key() << endl;
26     cout << "Key: " << A3Tree[1].key() << endl;
27
28     A3Tree.detachNTree( 0 );
29     A3Tree.detachNTree( 1 );
30     A3Tree.detachNTree( 2 );
31
32     return 0;
33 }
```

Line: 1 Column: 1 C++ Tab Size: 4





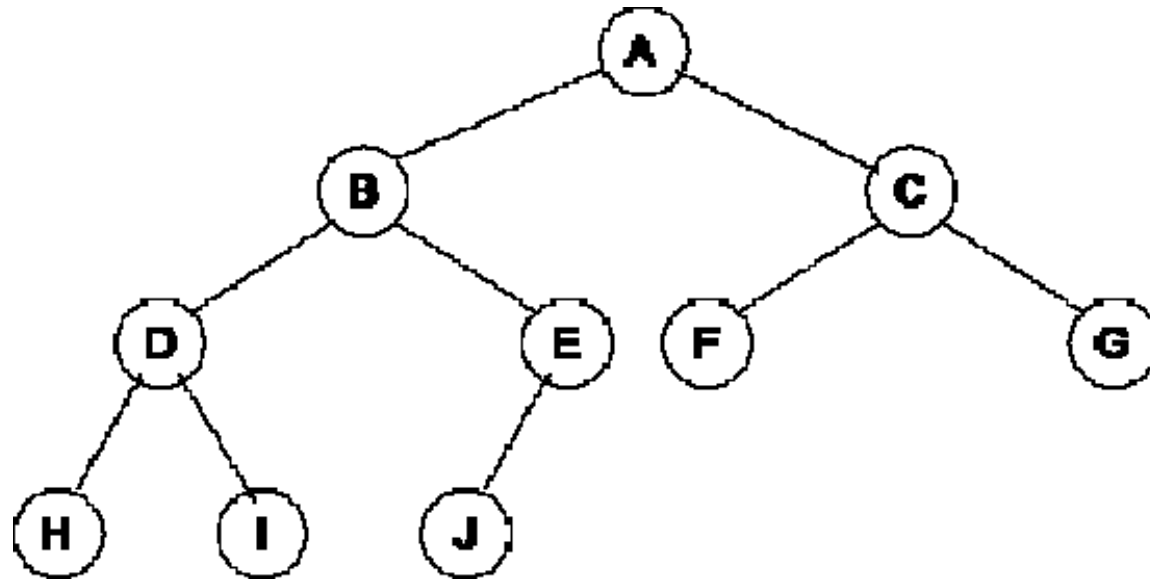
# Binary Trees (2-ary Trees)

---

- Definition: A Binary Tree is a Tree in which each node has a degree of **at most** 2.
- A binary tree  $T$  is a finite set of nodes with one of the following properties:
  - Either the set is empty,  $T = \emptyset$ , or
  - The set consists of a root,  $R$ , and exactly 2 distinct binary trees  $TL$  and  $TR$ ,  $T = \{R, TL, TR\}$ .
- The tree  $TL$  is called the **left** subtree of  $T$  and the tree  $TR$  is called the **right** subtree of  $T$ .

# Example : Binary Tree

---



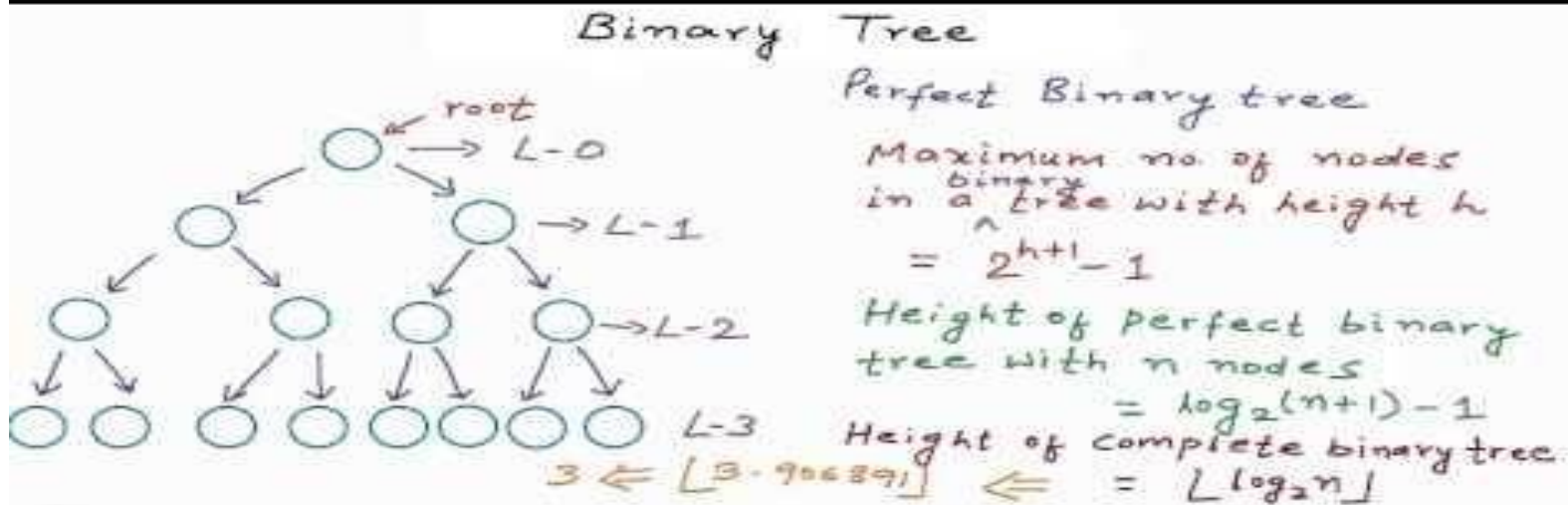
# BTree Class



```
h BTree.h
6  template<class T>
7  class BTree
8  {
9  private:
10     const T* fKey;           // 0 for empty BTree
11     BTree<T>* fLeft;
12     BTree<T>* fRight;
13
14     BTree();                 // empty BTree
15
16 public:
17     static BTree<T> NIL;     // sentinel
18
19     BTree( const T& aKey );   // root with 2 subtrees
20     ~BTree();
21
22     bool isEmpty() const;
23     const T& key() const;
24     BTree& left() const;
25     BTree& right() const;
26     void attachLeft( BTree<T>* aBTree );
27     void attachRight( BTree<T>* aBTree );
28     BTree* detachLeft();
29     BTree* detachRight();
30 };
31
```

Line: 4 Column: 21 C++ Tab Size: 4 BTREE.H

# Binary Tree

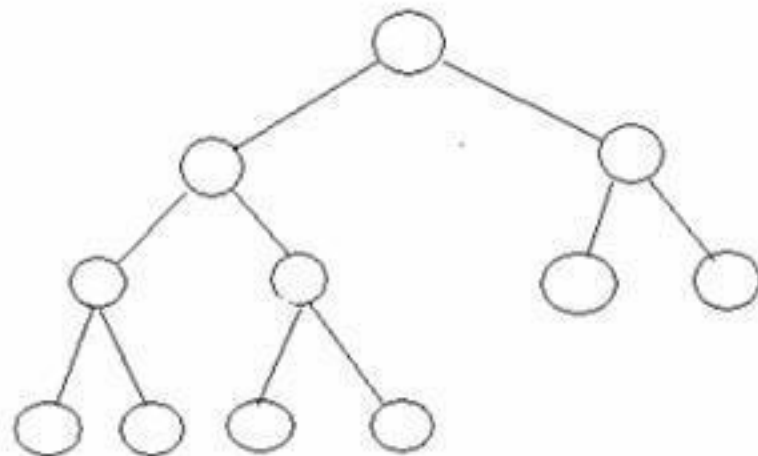
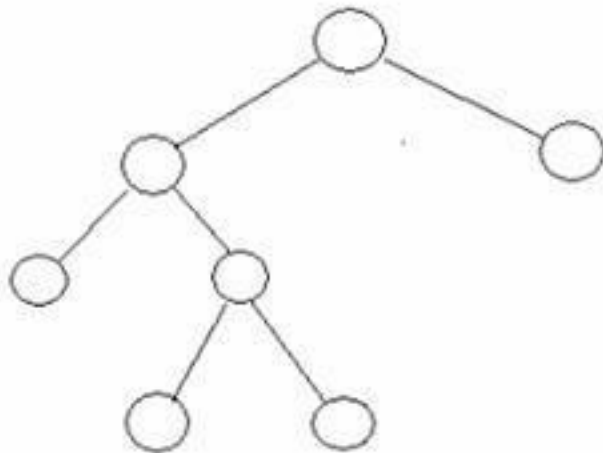




# More on binary tree

## ■ Full Binary tree:

- Every node should have exactly 2 nodes except the leaves. It is also called as **Strict Binary Tree** or **Proper Binary Tree**.
- Every node must have either 0 or 2 children.

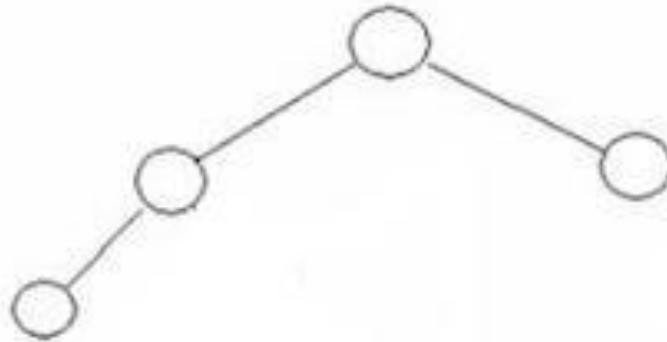
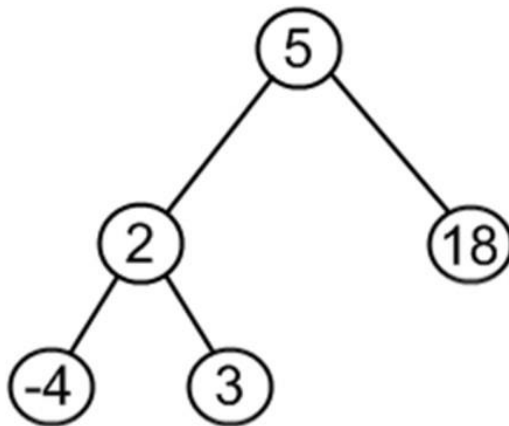




# More on binary tree

## ■ Complete Binary Tree:

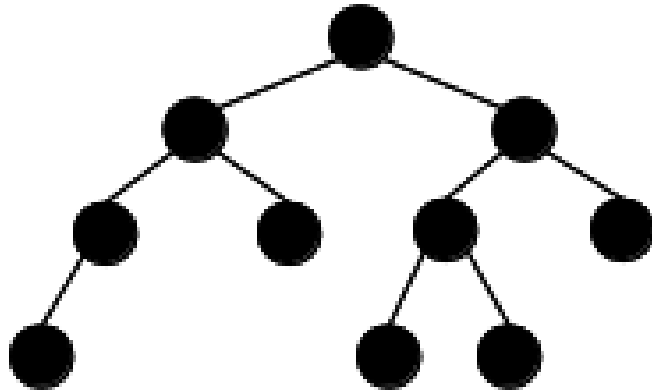
- It is a binary tree in which every level (except possibly the last) is completely filled, and all nodes are **as far left as possible**.



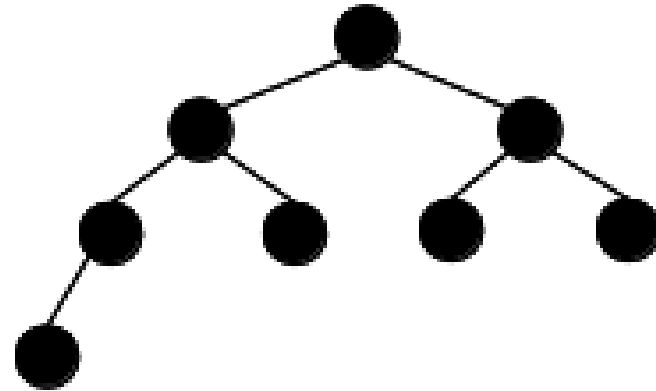
# More on binary tree



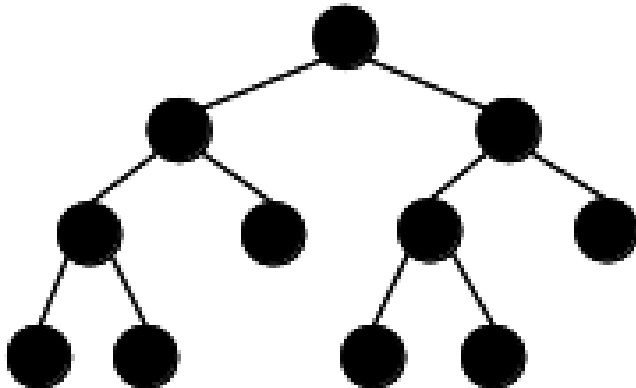
Neither complete nor full



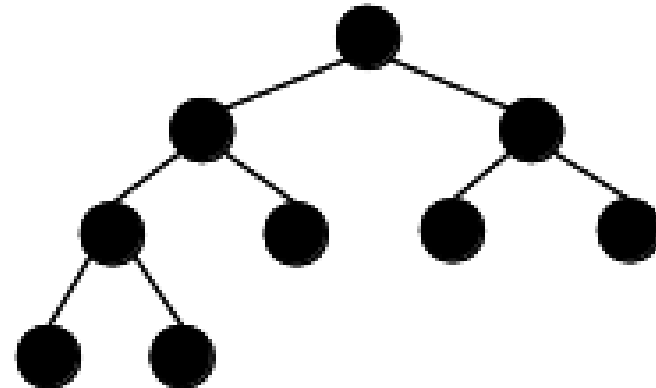
Complete but not full



Full but not complete



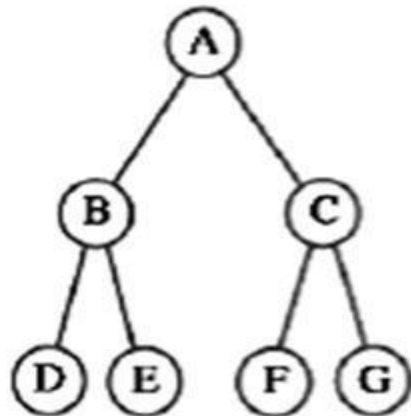
Complete and full



# More on binary tree

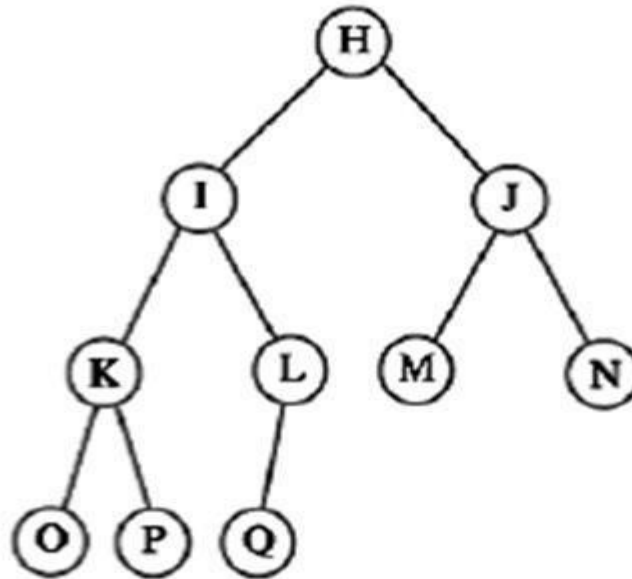


(a) Full tree

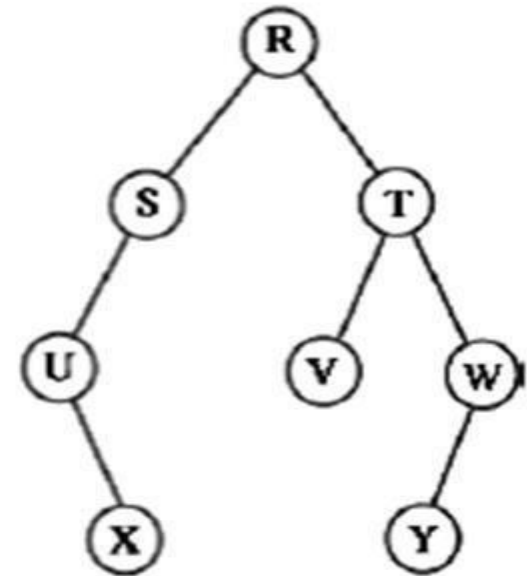


Left children: B, D, F  
Right children: C, E, G

(b) Complete tree



(c) Tree that is not full and not complete





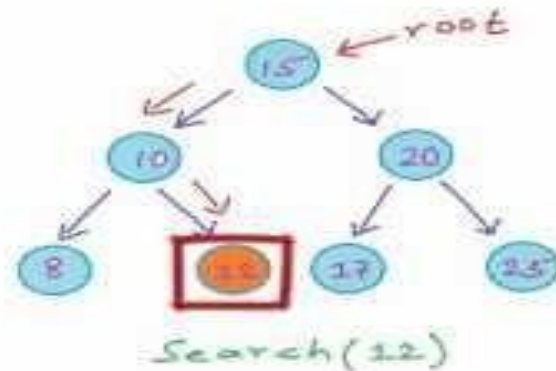
# Binary Search Tree



## Binary Search Tree

### Binary Search Tree (BST)

↳ a binary tree in which for each node, value of all the nodes in left subtree is lesser or equal and value of all the nodes in right subtree is greater.



mycodeschool.com



# Tree Traversal

---

- Many different algorithms for manipulating trees exist, but these algorithms have in common that they systematically visit all the nodes in the tree.
- There are essentially two methods for visiting all nodes in a tree:
  - Depth-first traversal (Depth First Search, DFS),
  - Breadth-first traversal (Breadth First Search, BFS)

# Depth-first Traversal

---



## ■ Pre-order traversal:

- Visit the **root first**; and then
- Do a preorder traversal of each of the subtrees of the root one-by-one in the order given (from **left to right**).

## ■ Post-order traversal:

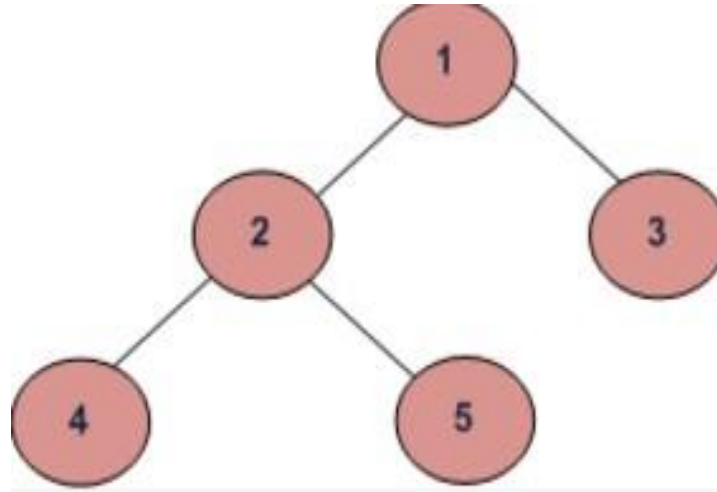
- Do a postorder traversal of each of the subtrees of the root one-by-one in the order given (from left to right); and
- **then** visit the root.

## ■ In-order traversal:

- Traverse the left subtree; and then
- Visit the root; and then
- Traverse the right subtree.



# Example of Depth-first Traversal

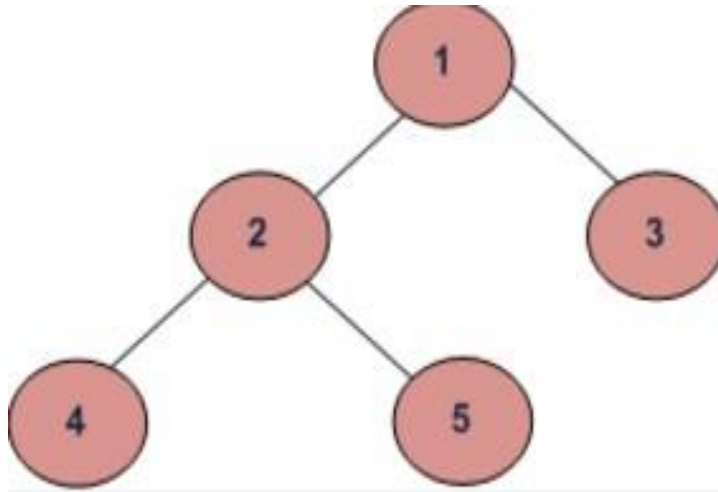


- (a) Preorder (Root, Left, Right) : 1 2 4 5 3
- (b) Postorder (Left, Right, Root) : 4 5 2 3 1
- (c) Inorder (Left, Root, Right) : 4 2 5 1 3



# Breadth-first Traversal

- Breadth-first traversal visits the nodes of a tree in the order of their depth (from left to right).

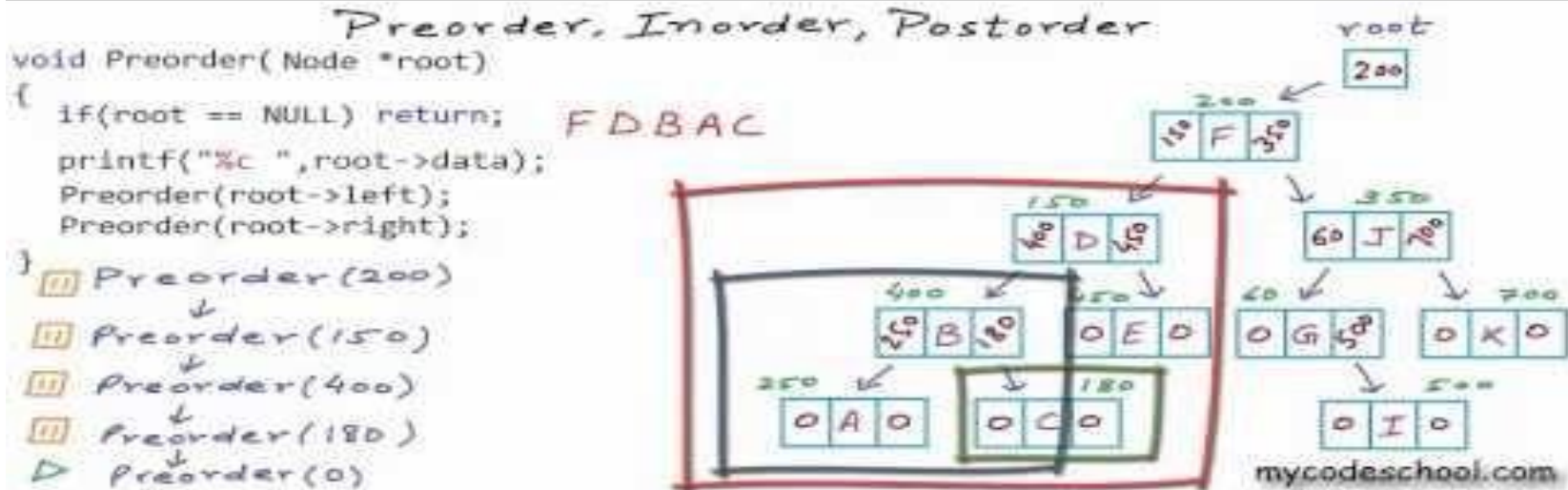


- BFS: 1 2 3 4 5

# DFS: Preorder, Inorder and Postorder



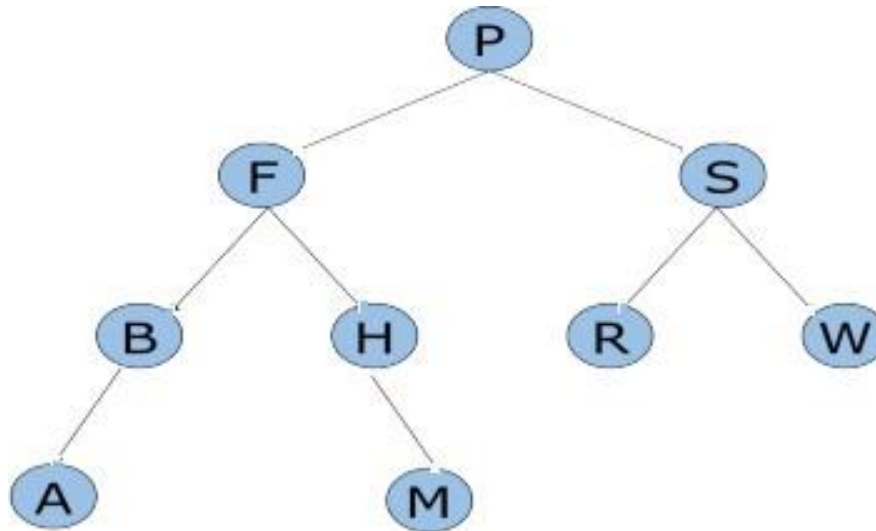
Note: Slight mistake for Inorder example, there is no H in the actual answer.





# Exercise

- Write the pre-order, in-order, post-order traversal of the following binary tree:



- Is the binary tree complete or full?



# Tricks for Preorder, Inorder and Postorder

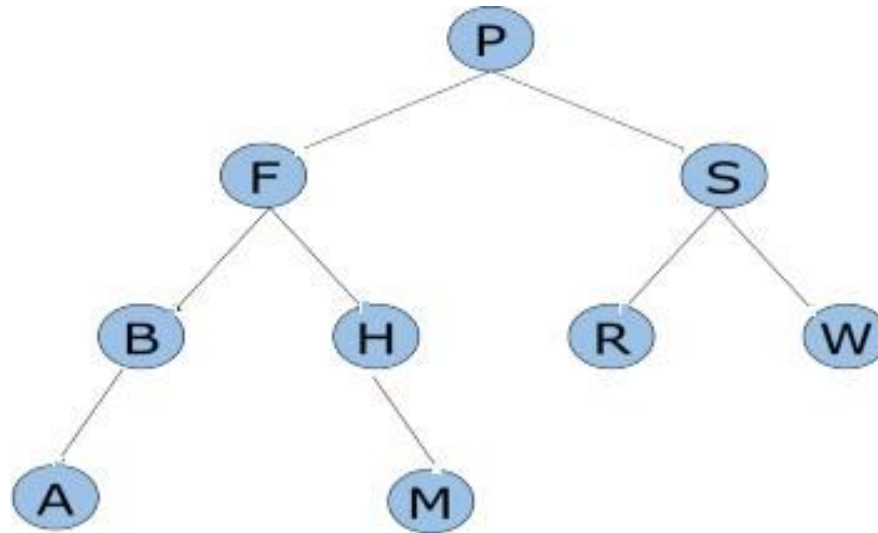
**Note: The video is in Hindi but the tricks for solving are great.**



**TRICK for Preorder,  
Inorder, Postorder with  
Example (Imp)**

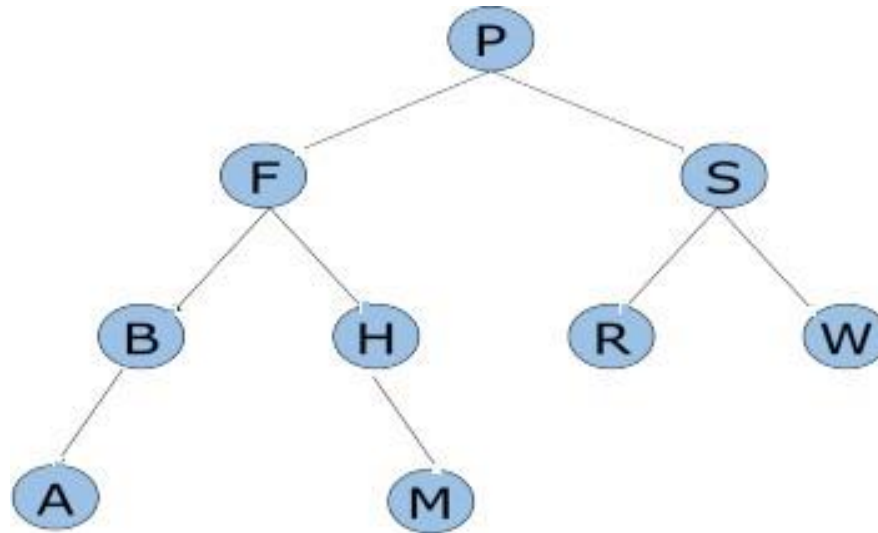


# Exercise (Answers)



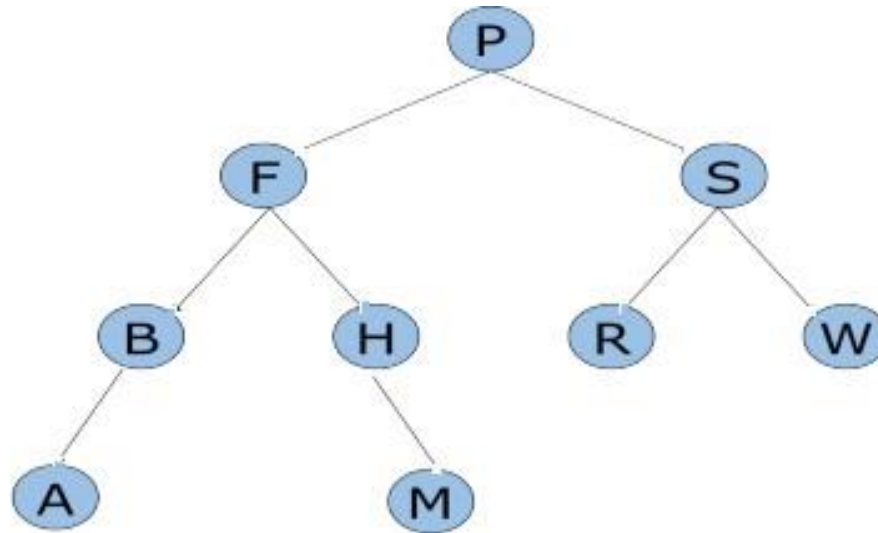
- Pre-order Traversal (Root, Left, Right):
  - P, F, B, A, H, M, S, R, W

# Exercise (Answers)



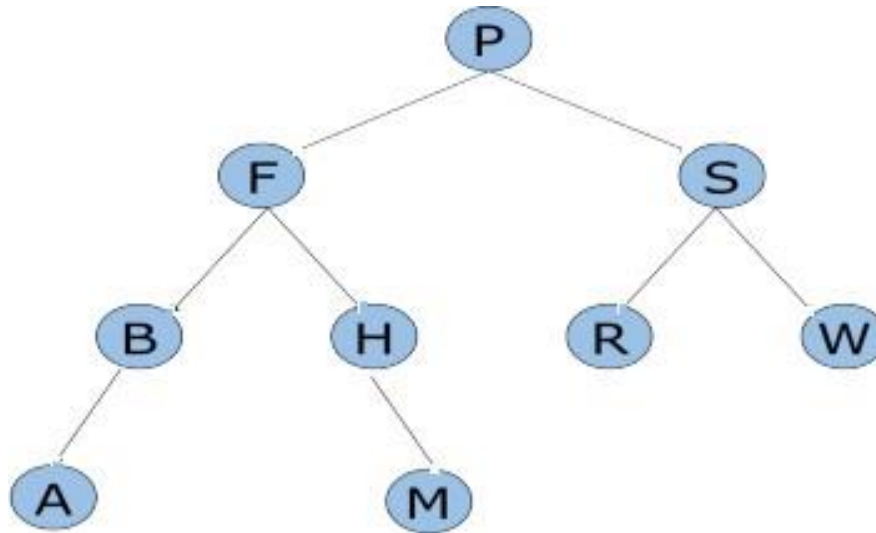
- In-order Traversal (Left, Root, Right):
  - A, B, F, H, M, P, R, S, W

# Exercise (Answers)



- Post-order Traversal (Left, Right, Root):
  - A, B, M, H, F, R, W, S, P

# Exercise (Answers)



- The binary tree is not complete and not full.
- It is not complete as some of the nodes have 1 leaf. In a complete binary tree, each nodes much have 0 or 2 children.
  - It is not full as not all nodes are as far left as possible.

# End of Tree (Part 1)

---



More about Trees in next week's lecture!