Chapter 7
Tree

CH-7 Contents

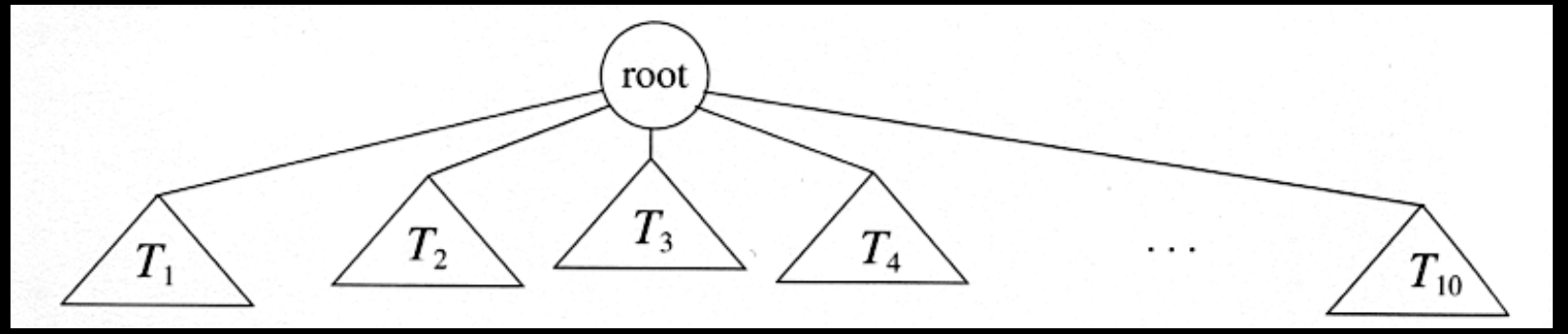
1. Trees, binary trees and binary search trees
2. Implementing binary trees
3. Binary Search tree operation
    1. Insertion
    2. Deletion
    3. Searching
    4. Traversal

Trees

- Linear access time of **linked lists** and **arrays** is expensive
  - **Requires O(N)** running time for most of basic operations like search, insert and delete
- Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?
  - The answer is yes, **tree data structures** require **O(log N)** for most of these operations
  - However, unlike Array and Linked List, which are **linear data structures**, **tree is hierarchical** (or non-linear) data structure.

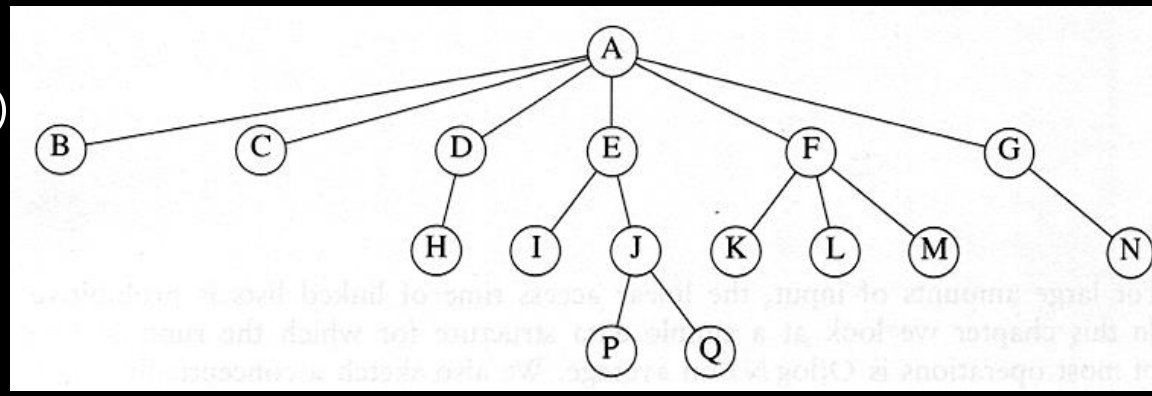
Trees

- A tree is a collection of nodes
  - The collection can be empty
- (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *sub-trees* T_1, T_2, …., T_k, each of whose roots are connected by an *edge* from r. Each sub tree itself satisfies the definition of a tree.



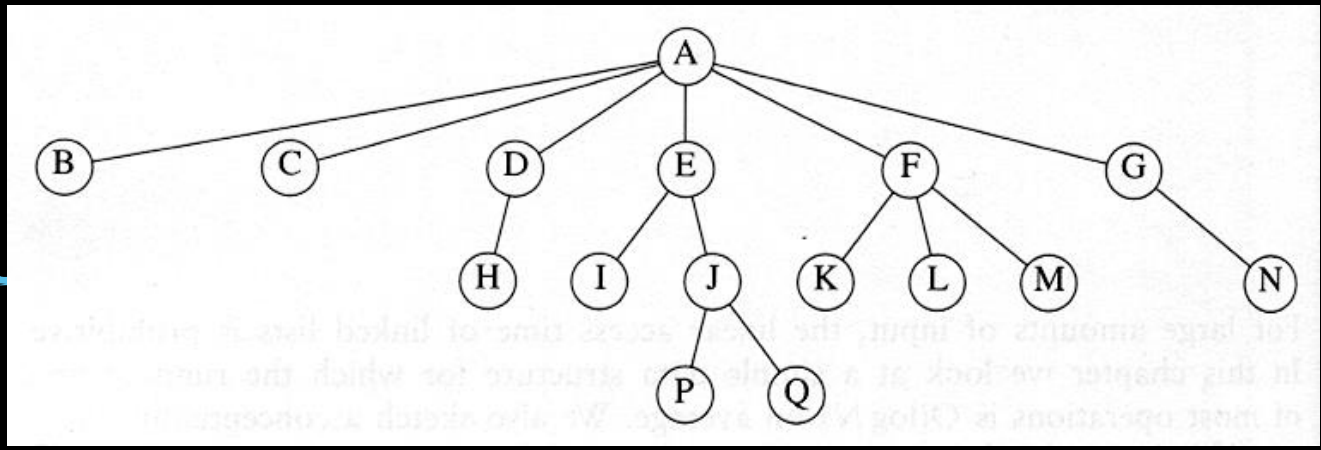
Some Tree Terminologies

- ***Child* and *parent***
  - Every node except the root has one parent (J is a parent of P and Q)
  - **P** and **Q** are child of **J**
  - A node can have an arbitrary number of children (A has 6 while D has 1 child)
- ***Leaves/External Nodes***
  - Nodes with no children (B, C, H, I, P, Q, K, L, M, N)
- ***Sibling***
  - nodes with same parent (Example, P and Q)



- ***Internal node***
  - A node with at least one child (A,D,E,F,G,J)
- ***Degree*: the number of possible child of a node**
  - Degree of node F = 3, node N=0, degree of the tree=degree of node having maximum degree. (=6)

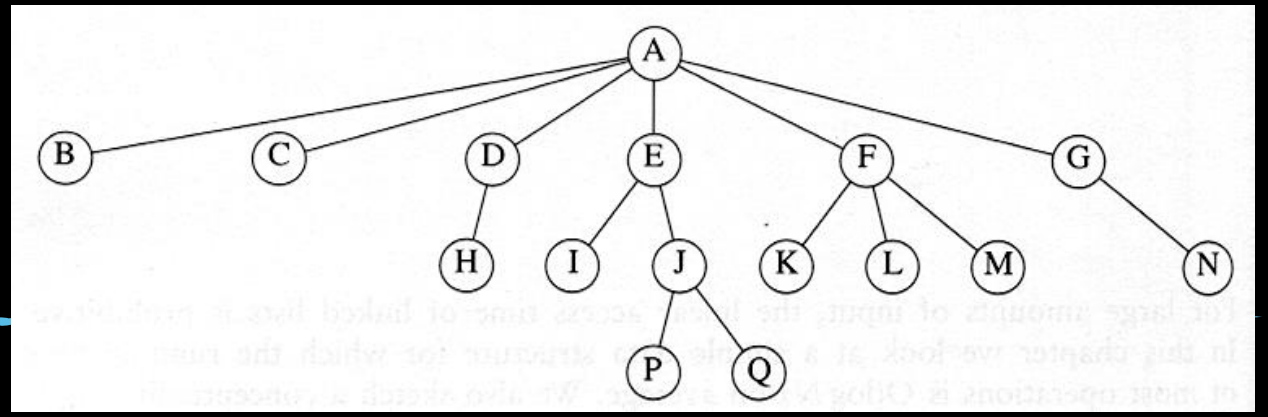
Some Terminologies



- **Path** – Path refers to the sequence of nodes along the edges of a tree.
  - ***Path length:*** the number of edges that must be traversed to get from one node to another

- *Length*
  - Number of edges on the path from node x to node y
- *Depth* of a node
  - Number of edges from the root to that node (Depth of C =1, of A = 0)
  - The depth of a tree is equal to the depth of the deepest leaf (=3)
    - Because, the deepest are P and Q and their depth is 3

Some Terminologies



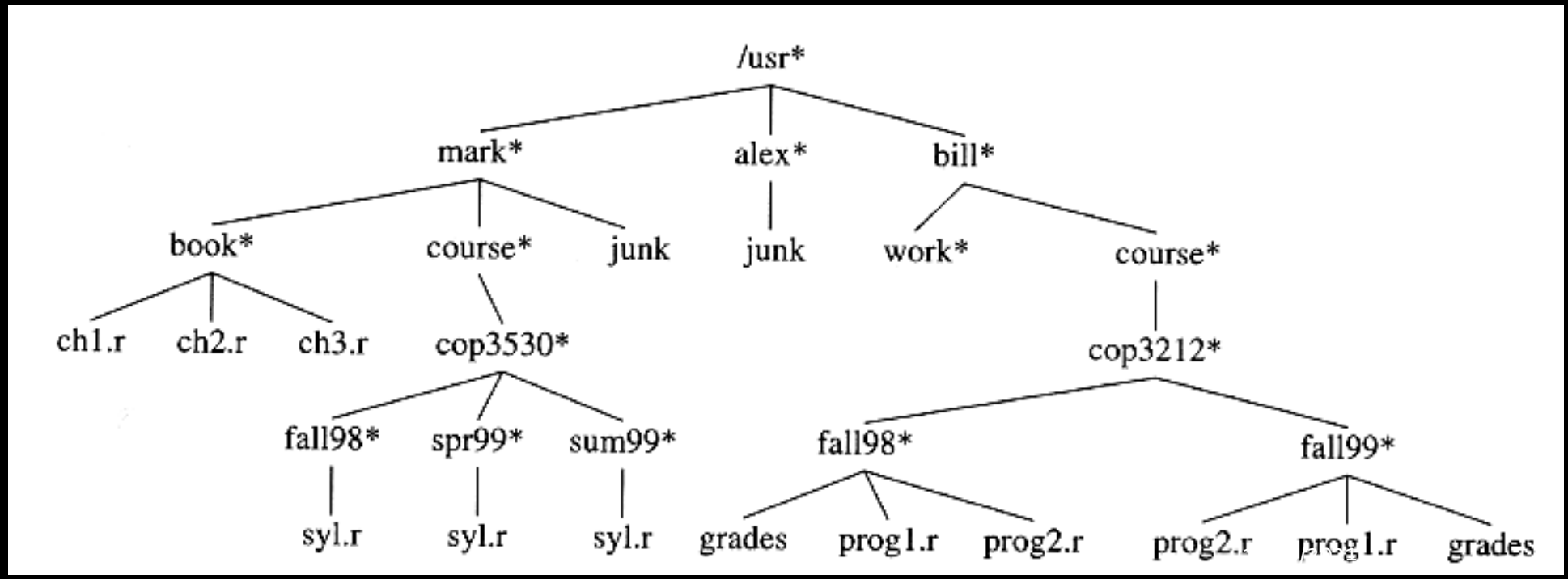
- ***Level***
  - *Level of node n, is the depth of node n.*
  - *The level of a node is one greater than the level of its parent.*
- ***Height*** of a node
  - length of the longest path from that node to a leaf (E=2)
  - all leaves are at height o
  - The height of a tree is equal to the height of the root
- *Ancestor* and *descendant*
  - The ancestors of a node are all the nodes along the path from the root to the node.
    - Parent, grand parent and great grand parents of anode
  - *Descendant* of a node are nodes reachable by repeated proceeding from parent to child.
    - *Child, grand child and great grand child of a node*

Applications of Trees

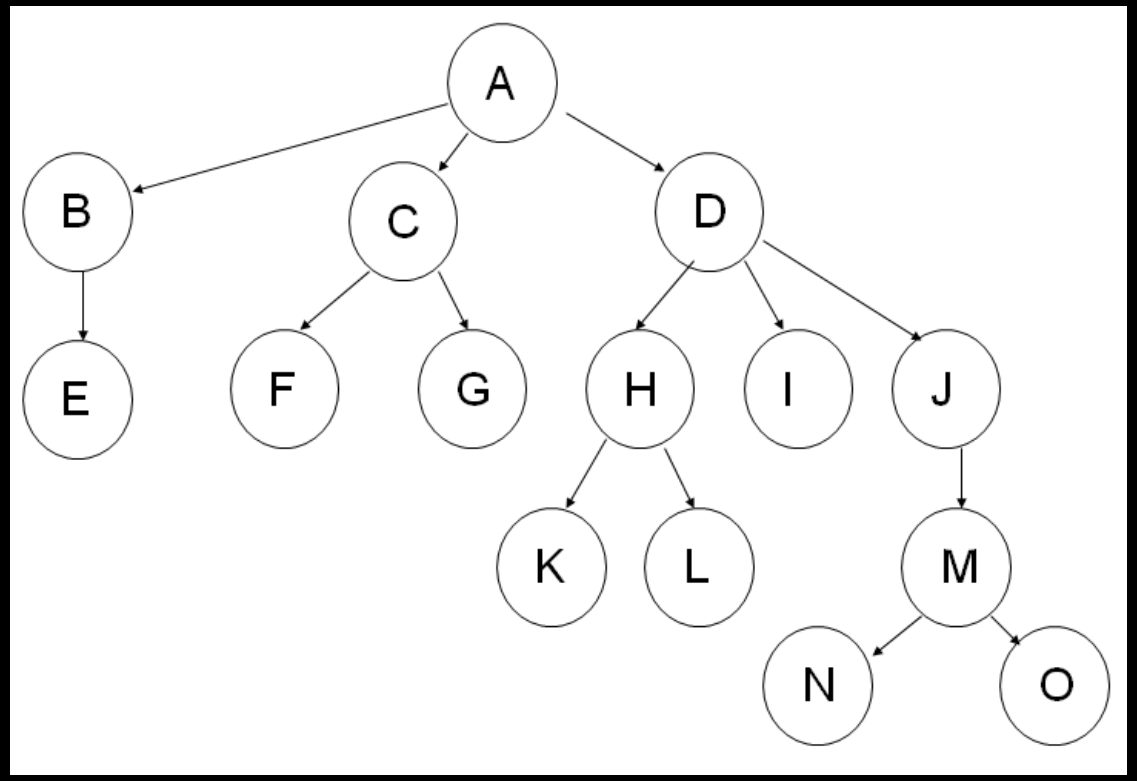
- Some of tree applications are:
  - **Implementing the file system** of several operating systems
  - **Evaluation of arithmetic expressions**
  - **Reduction of the time complexity** of some searching operations
  - **Storing hierarchies** in organizations
  - representing other data structres like Set ADT

Example: UNIX Directory

- Tree is useful to represent hierarchical data
- One of its application a file system used by many systems
- The following is an example of unix file system



Exercise: Given Tree

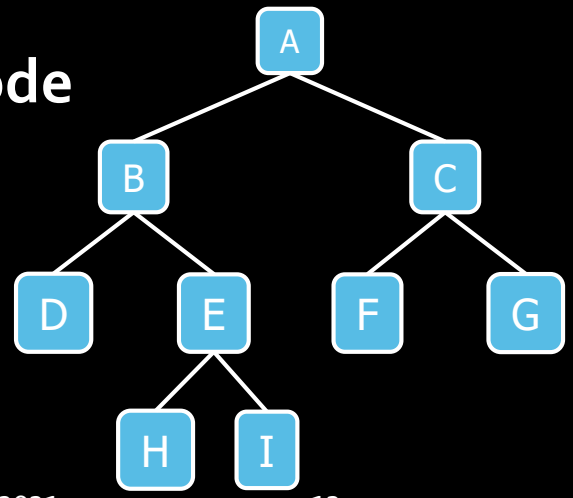


1. What is the root node of the tree?
2. How many sub-trees that the root contains?
3. List all the external nodes?
4. What is the depth of the node that contains M?
5. What is the height of the root?
6. What is the height of the node L?

Binary Tree

Binary Tree

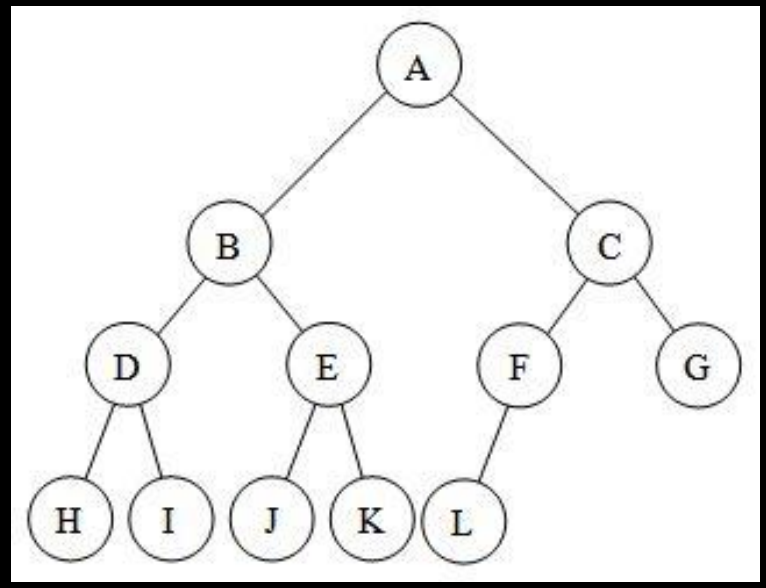
- **A general tree** is a tree where each node may have zero or more children (a binary tree is a specialized case of a general tree).
  - General trees are used to model applications such as file systems.
- ***Binary tree:*** each node has at most two children
  - The possible children are usually referred to as **the left child** and **the right child**
  - **A unique path exists from the root to every other node**



- Applications:
  - arithmetic expressions
  - decision processes
  - searching

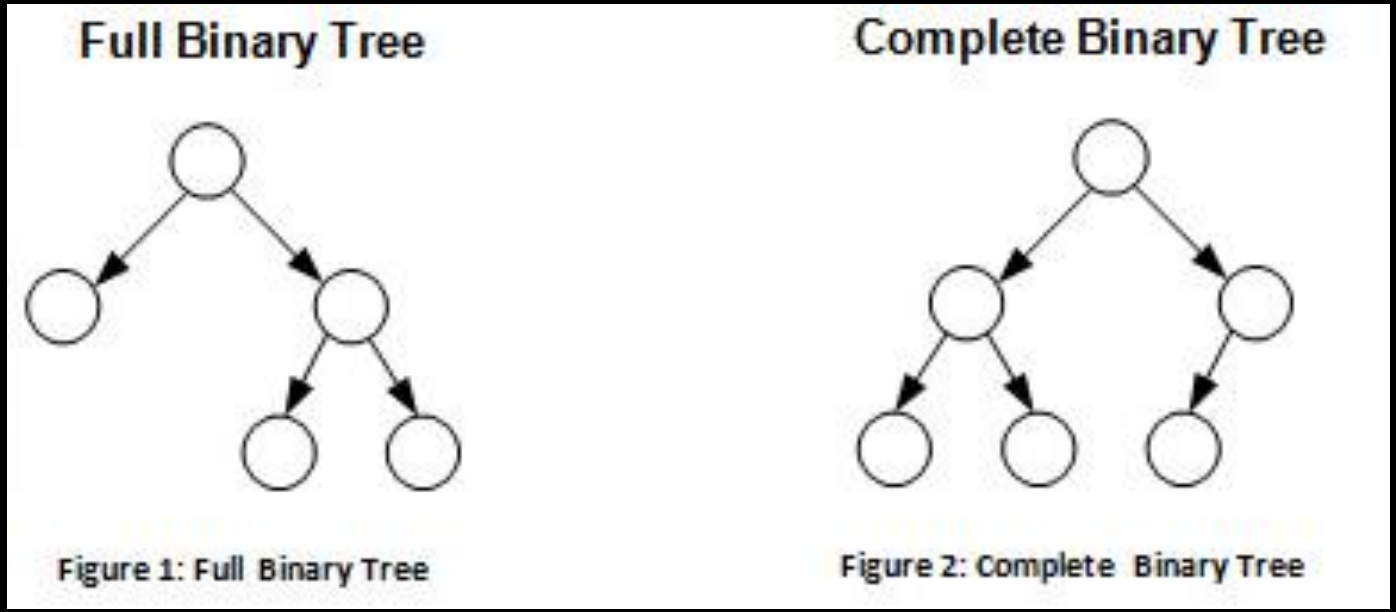
Complete Binary Tree

- A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
  - Nodes are filled from left to right in each level



Full Binary Trees

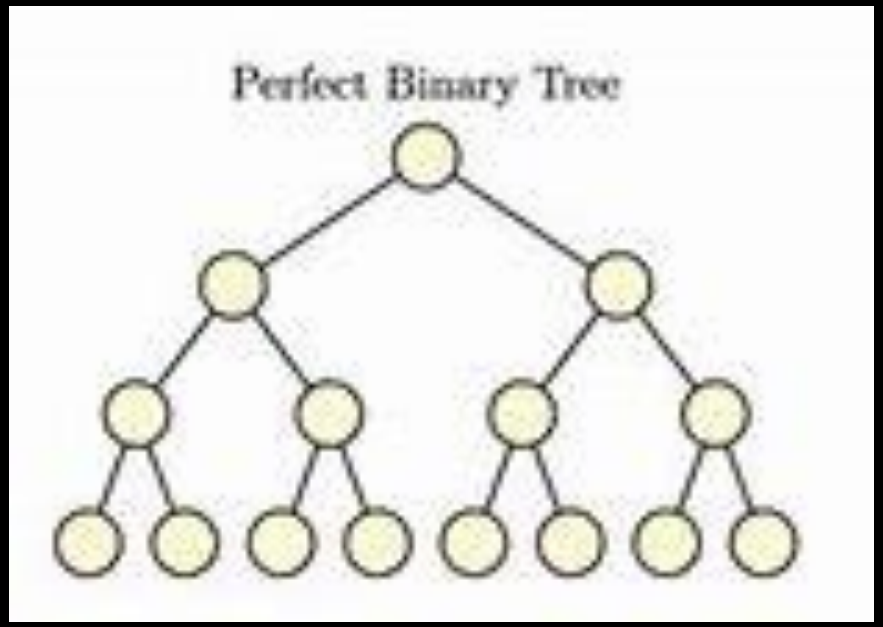
- A full binary tree is a binary tree in which every node has either zero or two children. That is, no nodes have only one child.



Perfect Binary Trees

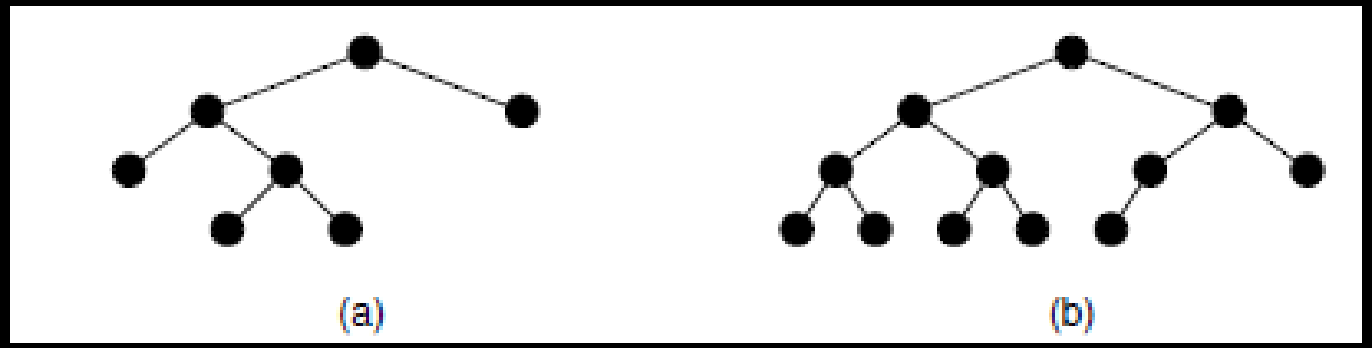
- A perfect binary tree is one that is both full and complete.
- All leaf nodes will be at the same level, and this level has the maximum number of nodes.

Note that perfect trees are rare in real life, as a perfect tree must have exactly $2^{L+1}- 1$ nodes where L is the number of levels



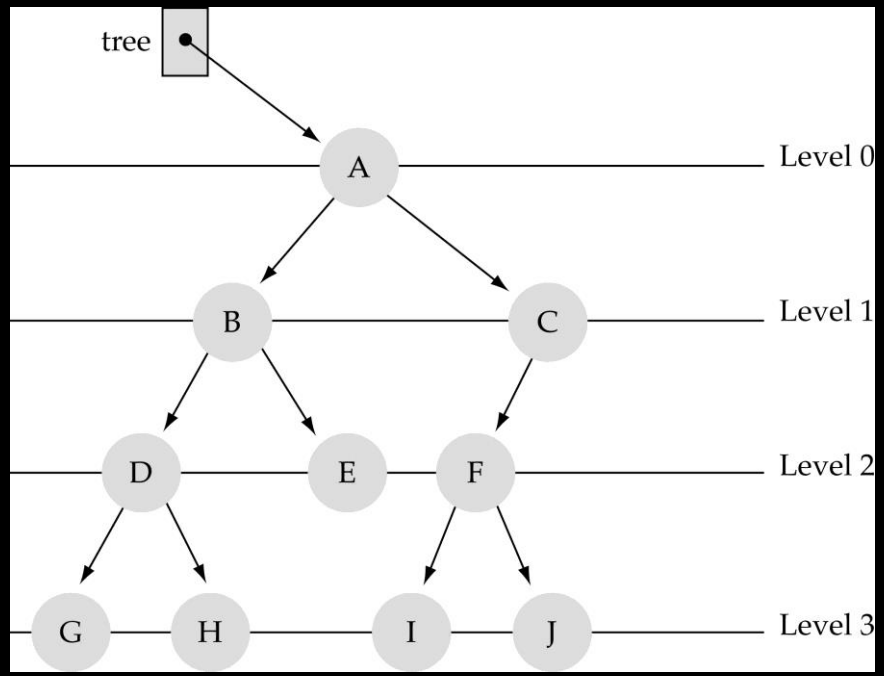
Exercise: Say True/False Based on figure a and b

- Both are binary trees
- Both are full binary trees
- Both are comp.lete binary trees
- A) is full binary while B) is complete binary tree
- B) is full binary while A) is complete binary tree
- A) is perfect while B)is complete



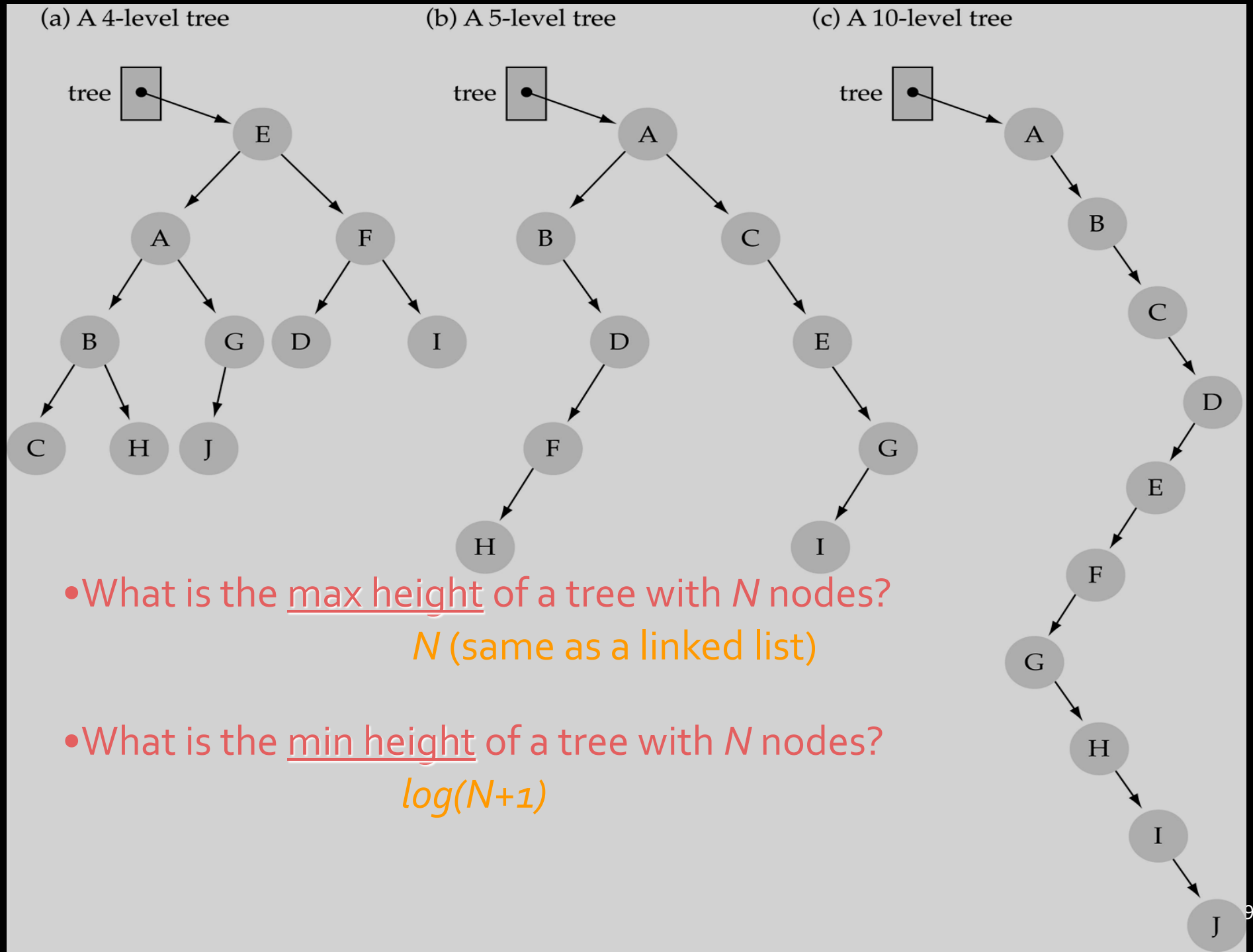
Binary Tree: # of nodes

- What is the MAX # of nodes at some level L?
  - The max # of nodes at **level L is 2^L**
    - Where L=0, 1, 2, … L -1



Why height (H) is important?

- What is the **total # nodes N** of a perfect binary tree **with height H**?
  - perfect binary tree: Every node has exactly two children and all the leaves are on the same level.
  - **$N = 2^{(H+1)} - 1$**
- What is the **height H** of a perfect binary tree with **N nodes**?
  - **$H = \log(N+1) - 1$ → O(Log N)**
- **Why height (H) is important?**
  - Tree operations (e.g., insert, delete, retrieve etc.) are typically expressed in terms of H.
  - So, H determines running time!



- What is the max height of a tree with *N* nodes?

  N (same as a linked list)

- What is the min height of a tree with *N* nodes?

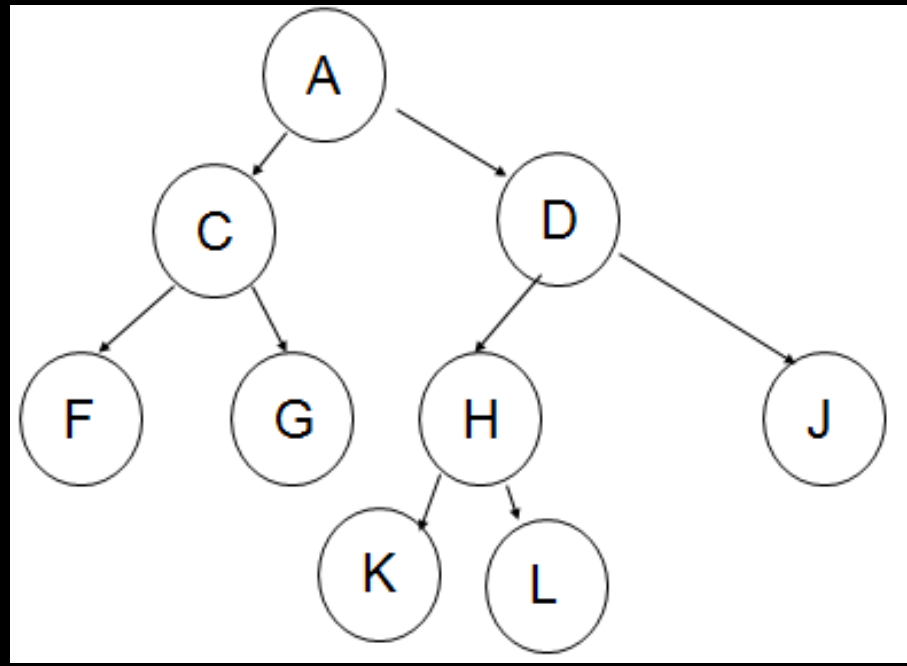
  log(N+1)

Tree traversal

- Traversing a tree means processing it in such a way that each node is visited for processing only once.
- There are three traversals methods used to visit/print out the node/data in a tree in a certain order
- **Pre-order traversal**
  - Print the data at the root
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree
- **In-order Traversal**
  - Recursively print out all data in the left subtree
  - Print the data at the root
  - Recursively print out all data in the right subtree
- **Post-order Traversal**
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree
  - Print the data at the root

Traversal Applications

- Make a clone
- Determine height
- Determine number of nodes
- representing arithmetic expression



More on Tree traversal

- You may 'pass through' a node as many times as you like but you can only process the node once.

- During a **pre-order traversal** each node is processed before any nodes in its subtrees
- During an **in-order traversal** each node is processed after all nodes in its left subtree but before any nodes in its right subtree.
- During a **post-order traversal** each node is processed after all nodes in both its subtrees

Preorder, Postorder and Inorder

- Once the expression tree is built, all the three forms of an algebraic expression (infix, prefix, and postfix) are immediately available to us if we know exactly how the corresponding tree should be traversed.

- **Preorder traversal**
  - node, left, right (recursively)
  - It produces prefix expression
    - ++a*bc*+*defg

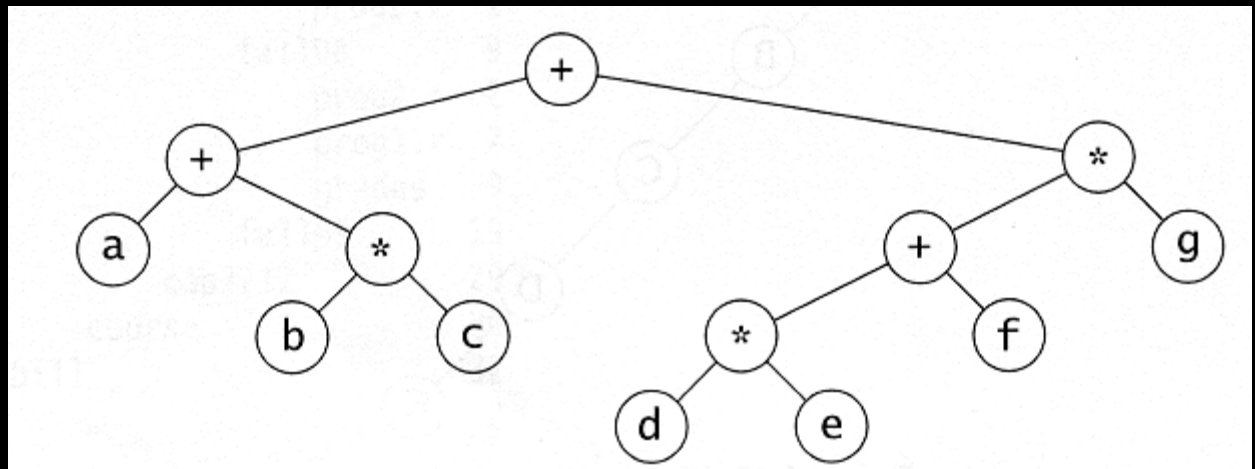
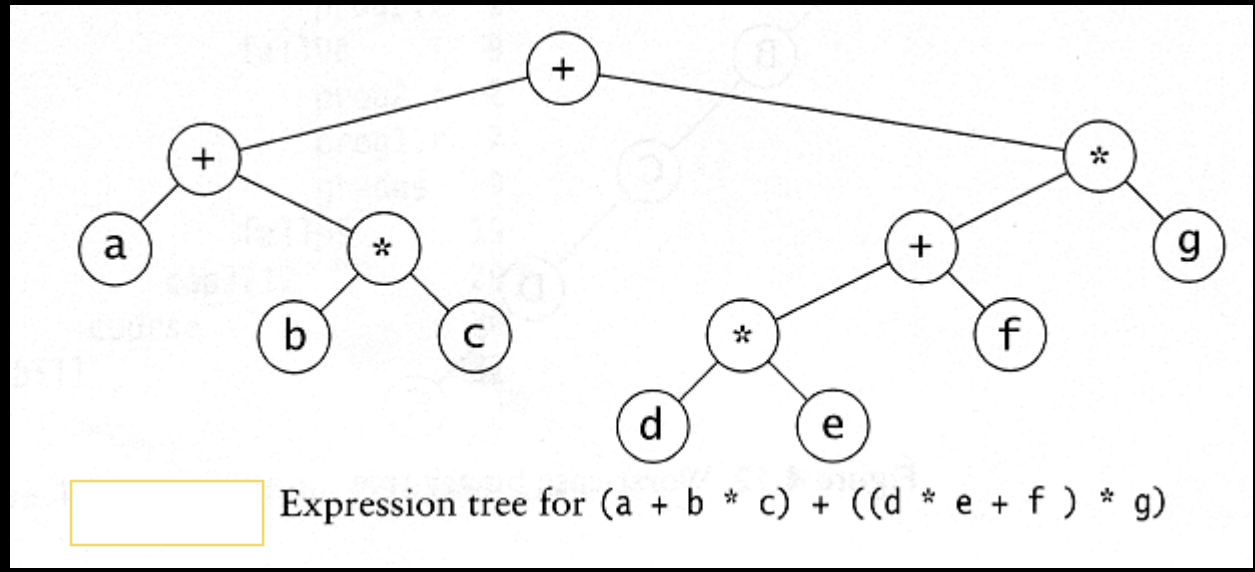


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Preorder, Postorder and Inorder

- **Post-order traversal**
  - left, right, node (recursively))
  - Gives postfix expression
    - abc*+de*f+g*+

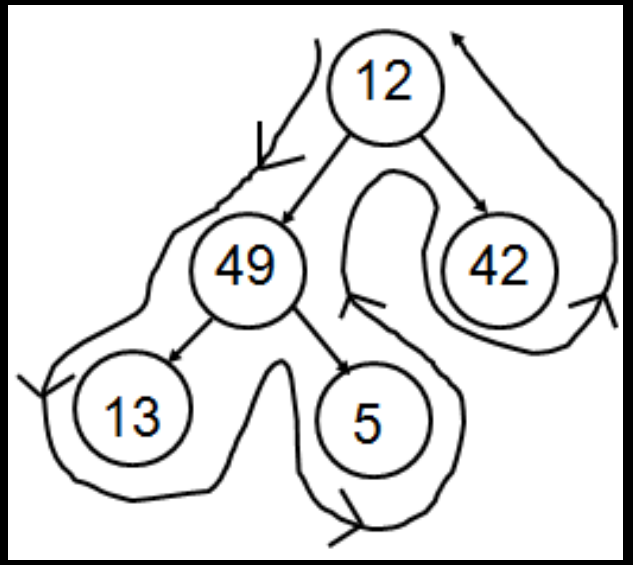
- **In-order traversal**
  - left, node, right. (recursively)
  - Gives infix expression
    - a+b*c+d*e+f*g



Expression tree for (a + b * c) + ((d * e + f) * g)

More on tree Traversal

- Determining the results of traversal (i.e order of the nodes visited) manually is confusing at times
- One of the techniques to determine the results of a traversal on a given tree, though, is to draw a path around the tree.
  - Start on the left side of the root and trace around the tree. The path should stay close to the tree

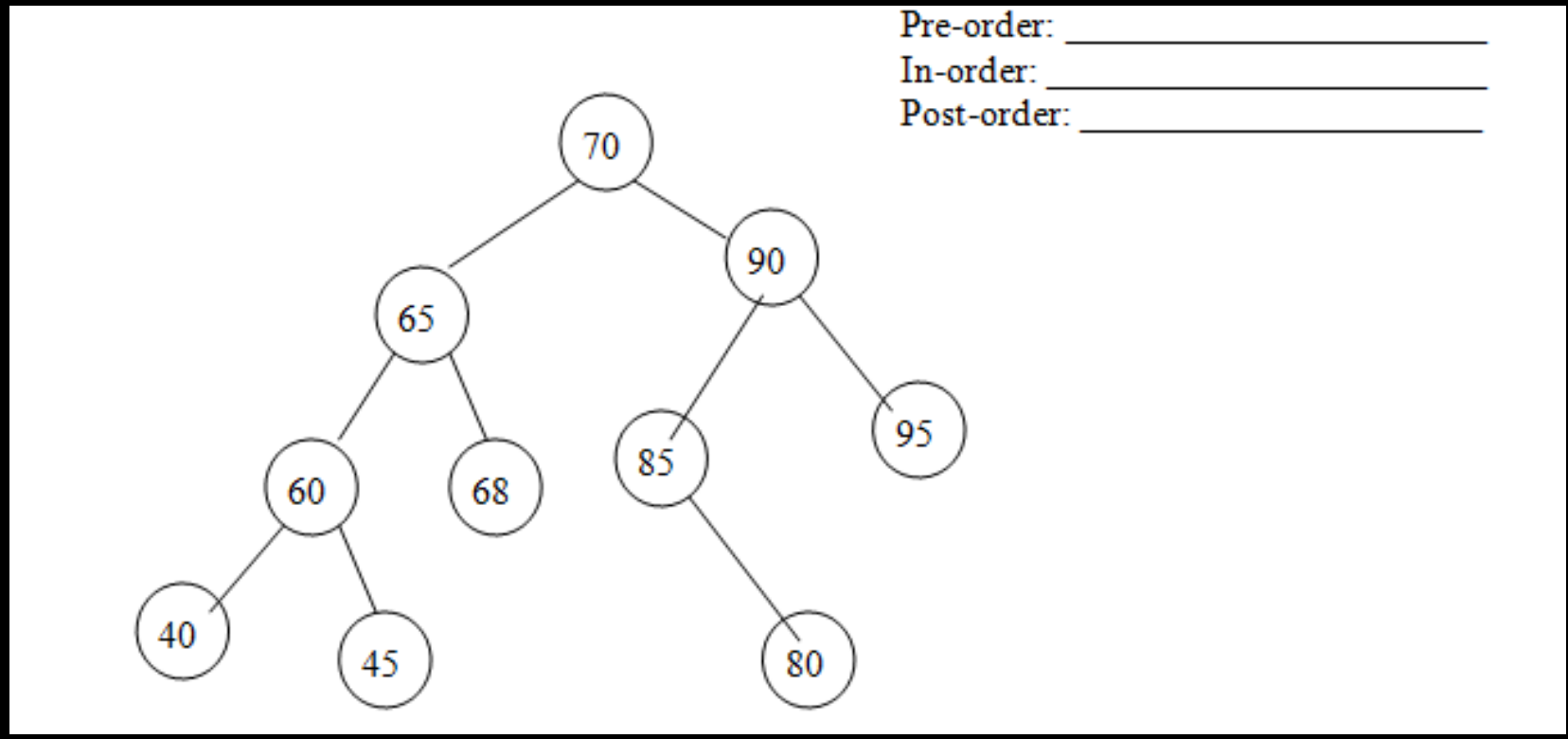


More on tree traversal

- **Pre order**: process when pass down left side of node: 12 49 13 5 42
- **In order:** process when pass underneath node: 13 49 5 12 42
- **Post order**: process when pass up right side of node: 13 5 49 42 12

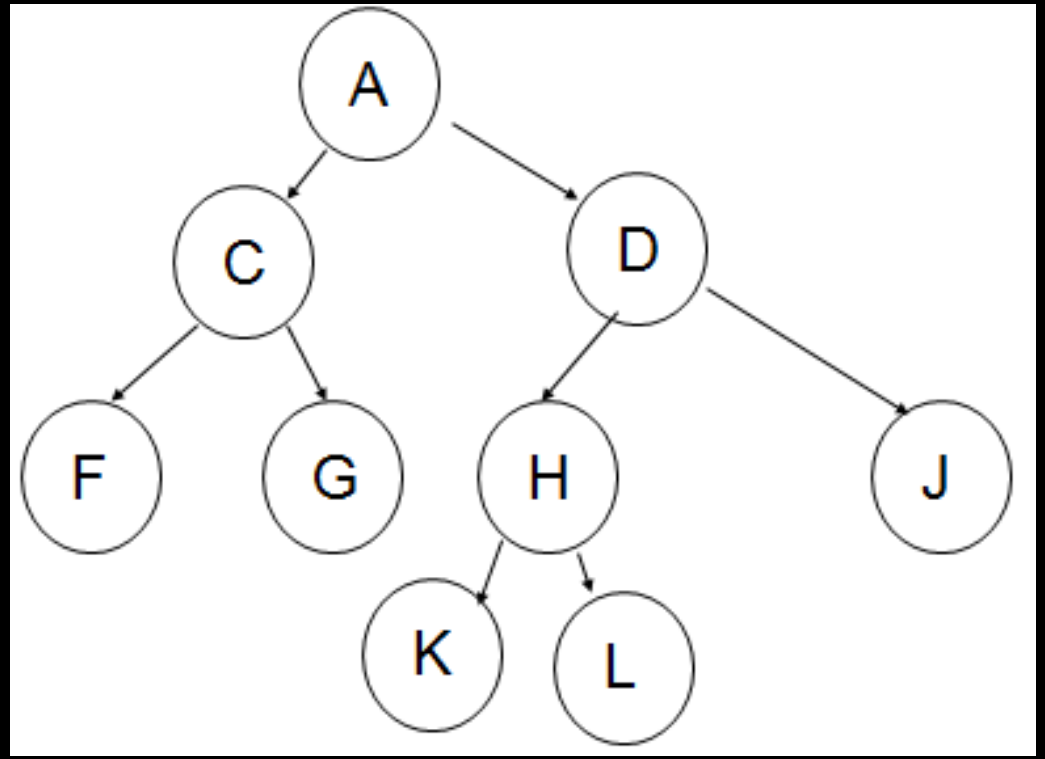
Exercise 1

- Show list of nodes when the following tree is traversed in: the tree in



Exercise 2

- What is the result of a given traversal type
  - **Pre order**
  - **In order**
  - **Post order**
  - **Level order**



How to search a binary tree?

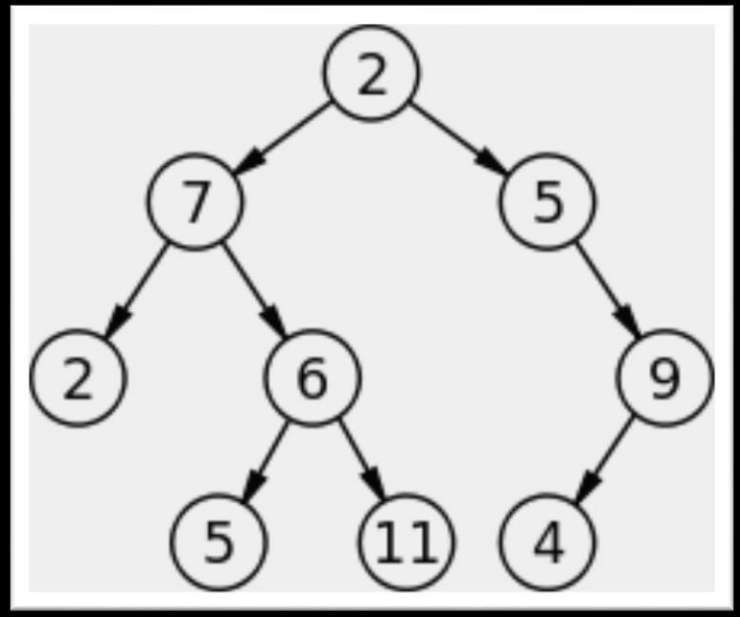
1. Start at the root
2. Search the tree level by level, until you find the element you are searching for or you reach a leaf.

Question

- Is this better than searching a linked list?

Answer

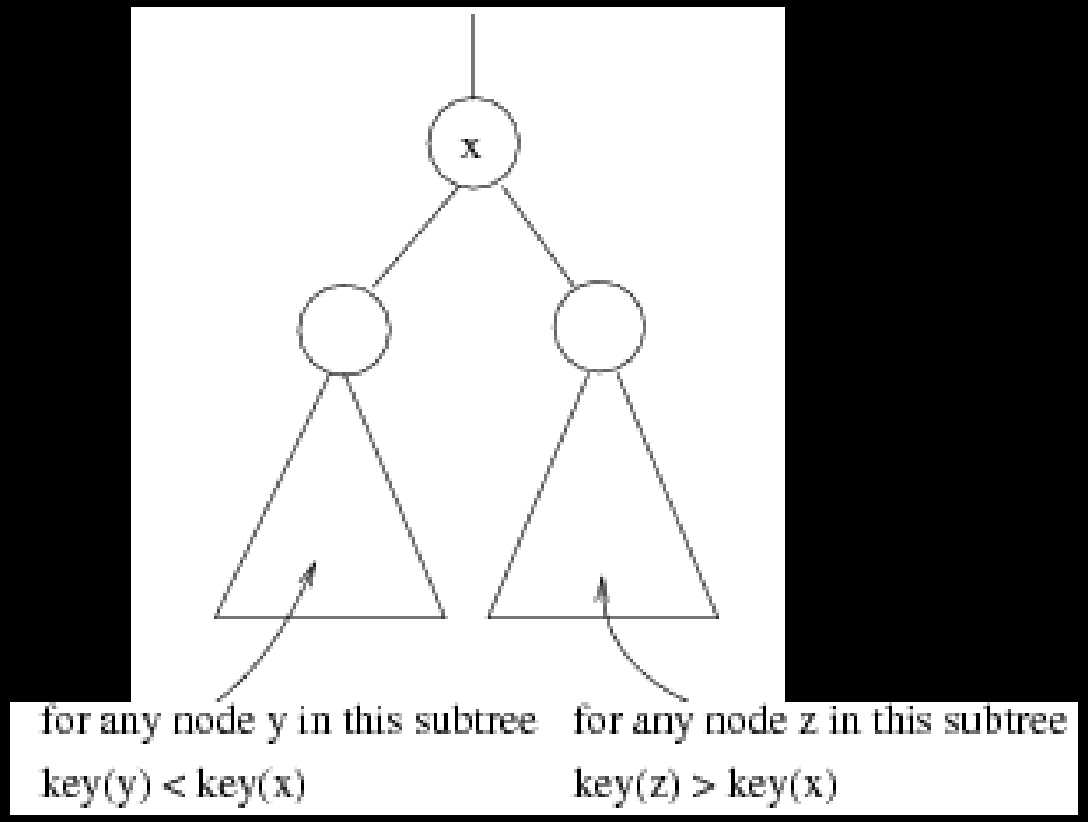
- No, it takes a linear time→ O(N)



Binary Search Tree(BST)

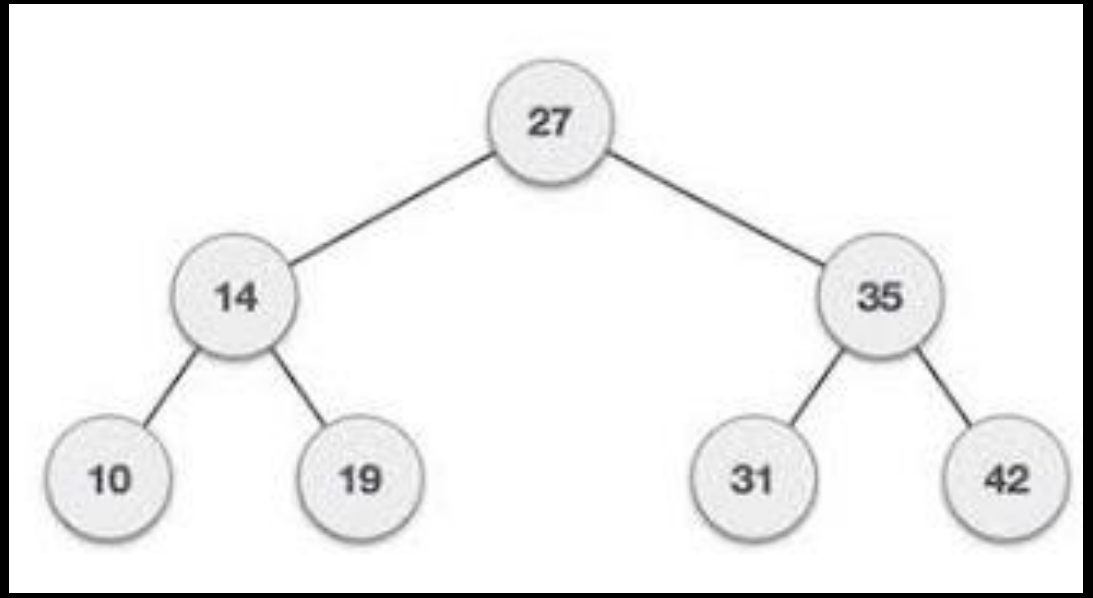
Binary Search Tree (BST)

- A data structure for efficient **searching**, **insertion** and **deletion**.
- **Binary search tree property**
  - For every node X
  - All the keys in its left sub-tree are smaller than the key value in X
  - All the keys in its right sub-tree are larger than the key value in X

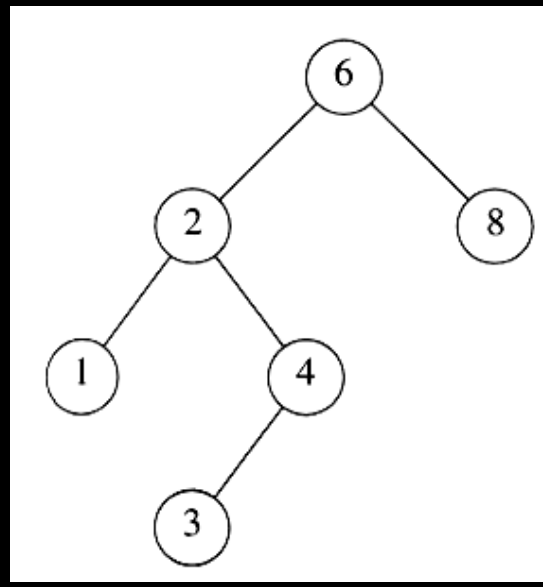


BST Representation

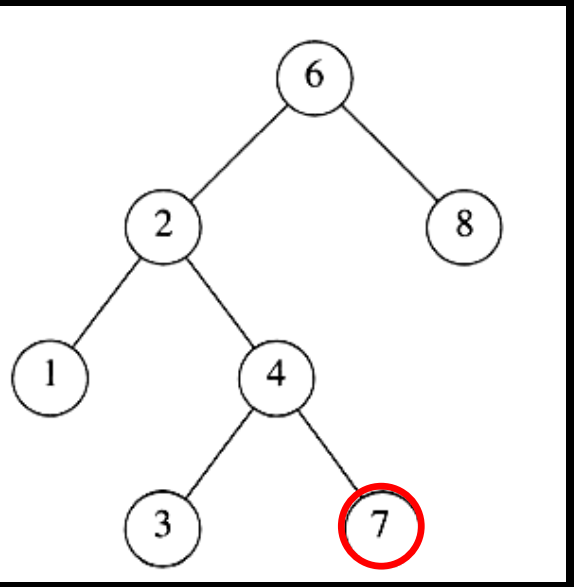
- Binary Search tree exhibits a special behavior.
  - A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



Binary Search Trees



A binary search tree



Not a binary search tree

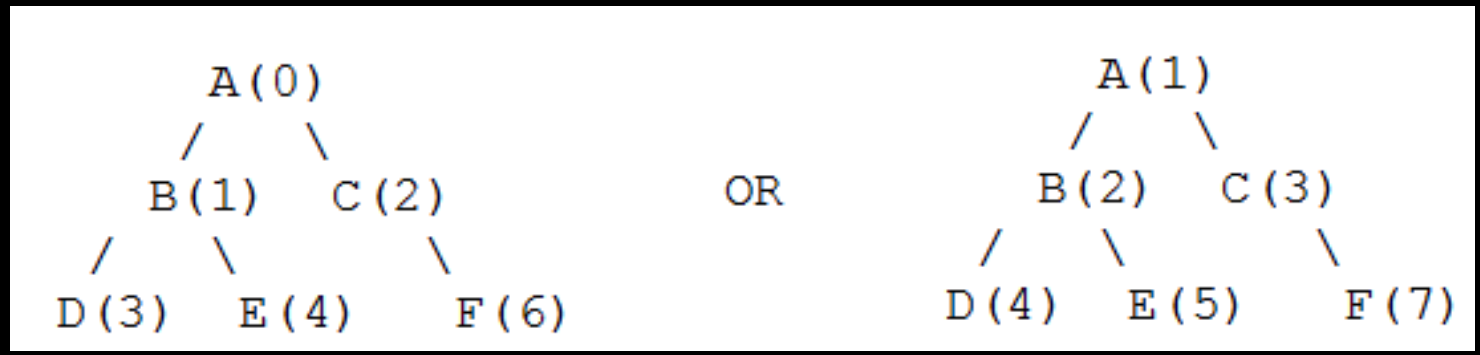
Implementing binary trees

Implementing Binary Search Tree

- A tree is represented by a pointer to the topmost node in tree called root.
- If the tree is empty, then value of root is NULL.
  - A tree node contains following parts
    - **Data**
    - **Pointer to left child**
    - **Pointer to right child**
- It can be implemented in at least two ways:
  - Array
  - Linked List

Array (Sequential) Representation

- To represent tree using array, numbering of nodes can start either from 0 to (n-1) or 1 to n .



- **First case(0 to n-1)**
  - if (say) Parent=p;
  - Left child=(2*p)+1
  - Right child=(2*p)+2;
- **Second case(1 to n)**
  - if (say) Parent=p;
  - Left child=(2*p);
  - Right child=(2*p)+1;

Limitation of array implementation

- This implementation may be inconvenient, as is often the case with **static allocation**,
  - Since the size of the array has to become part of the program and must known in advance.
- It is a problem because the data may overflow the array if too little space is allocated, or memory space may be wasted if too much space is allocated.

Linked representation of binary search trees

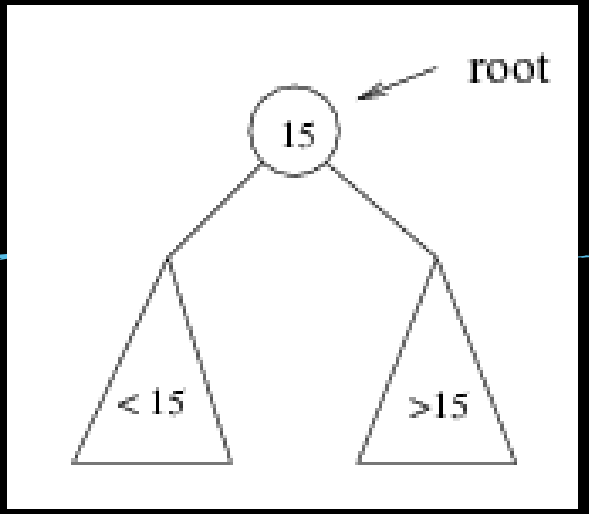
- Because each node in a binary tree may have two child nodes, a node in a linked representation has two pointer fields
- We will use the following general structure specification

```
struct Node
 {
     int data; //the data type can be any appropriate type
     node  *left;
     node  *right;
 };
        Node *root = NULL; // The root node of the tree
```

BST Basic Operations

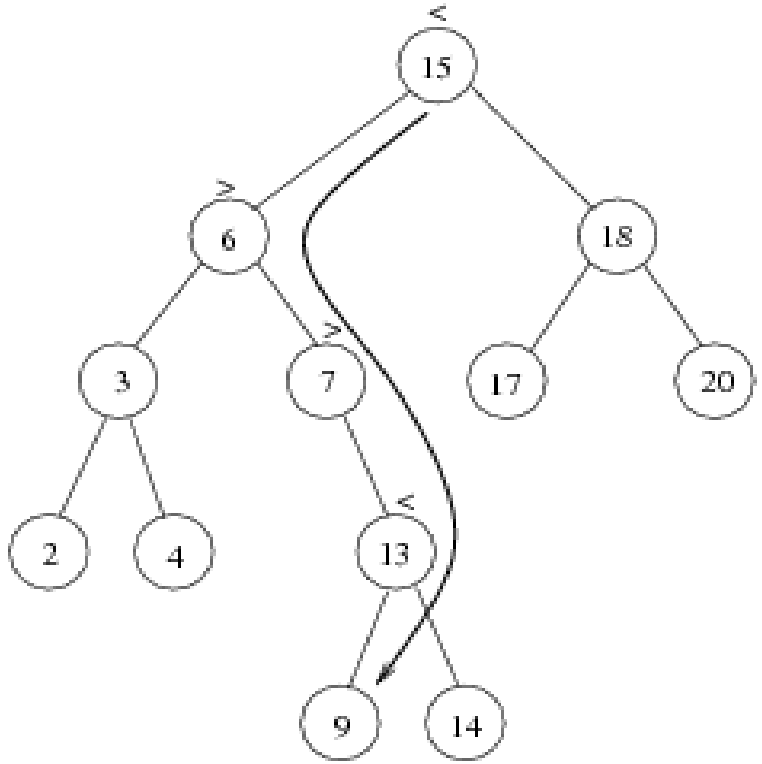
- The basic operations that can be performed on a binary search tree data structure, are the following
  - **Search** – Searches an element in a tree.
  - **Insert** – Inserts an element in a tree/create a tree.
  - **Delete**– Deletes an element from a tree
  - **Traversal**
    - **Pre-order Traversal** – Traverses a tree in a pre-order manner.
    - **In-order Traversal** – Traverses a tree in an in-order manner.
    - **Post-order Traversal** – Traverses a tree in a post-order manner.

Searching BST



- For every node X
  - All the keys in its left sub-tree are smaller than the key value in X
  - All the keys in its right sub-tree are larger than the key value in X
- **Example**: search for 15
  - If we are searching for 15, then we are done.
  - If we are searching for a key < 15, then we should search in the left subtree.
  - If we are searching for a key > 15, then we should search in the right subtree.

Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Searching (Find)

- Find X: return a pointer to the node that has key X, or NULL if there is no such node

```
Node *searchBST(Node *root, int x)
{
  If(root==NULL || root->num==x)
    Return (root)
  else if(root->num>x)
    Return (searchBST(root->left, x)
  else
    Return (searchBST(root->right, x))
  }
```

- Time complexity
  - O(height of the tree)

findMin

- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```
Node*findMin(node* root)
{
  If(root==NULL)
     Return Null;
  Else if(root->left==Null)
     Return root
  Else
     Return(findMin(root->left)
}
```

- Time complexity = O(height of the tree)

findMax

- Finds the maximum element in BST
- Start at the root and go right as long as there is a right child. The stopping point is the largest element

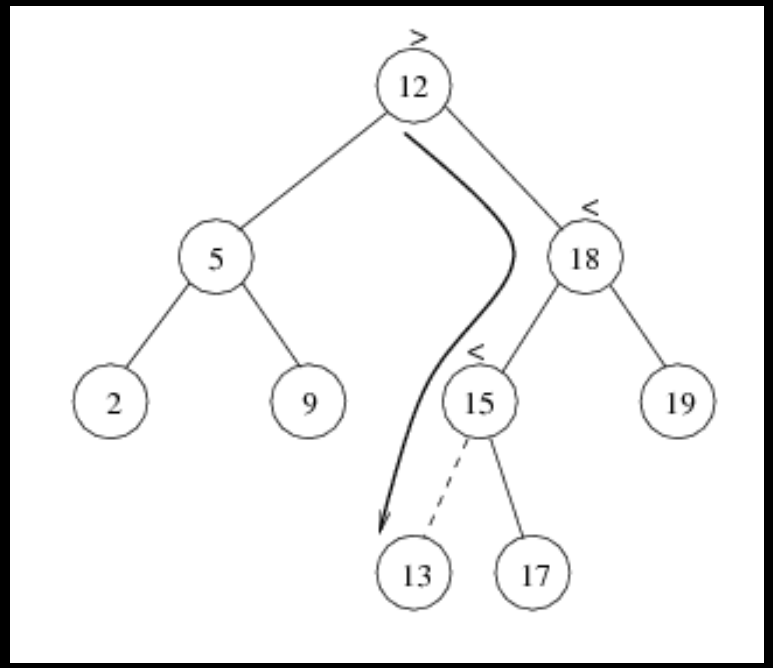
```
Node*findMax(node*root)
{
 If(root==NULL)
   Return Null;
 Else if(root->right==Null)
   Return root
 Else
   Return(findMax(root->right)
}
```

Inserting node in BST

- When a new node is inserted the definition of BST should be preserved.
- There are two cases to consider
  - There is no data in the tree (root=null)
    - root=newnode;
  - There is data
    - Search the appropriate position
    - Insert the node in that position.

Example-insert node13

- **Proceed down** the tree as you would **with a find**
- If X is **found, do nothing** (or update something)
- Otherwise, **insert X at the last spot** on the path traversed



- **Time complexity: O(height of the tree)**

Insert node

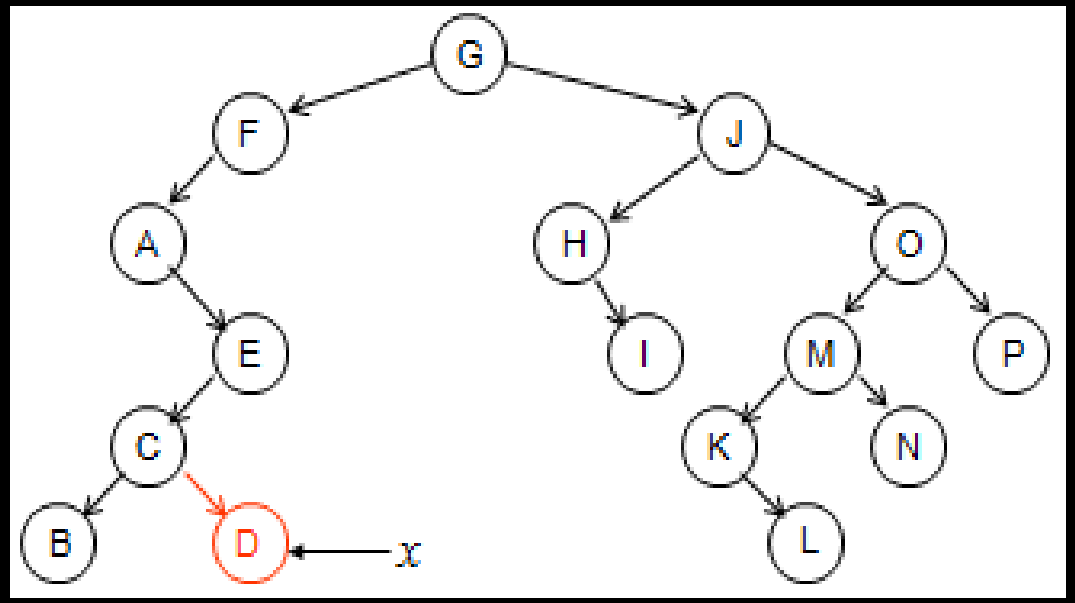
```
void insertNode( Node &root, pnode p)
{
 if(root==NULL)
  {
     root=p;
     root->left=NULL;
     root->right=NULL;
  }
  else if(root->data>p->data)
            add(root->left,p);
     else
            add(root->right,p);
}
```

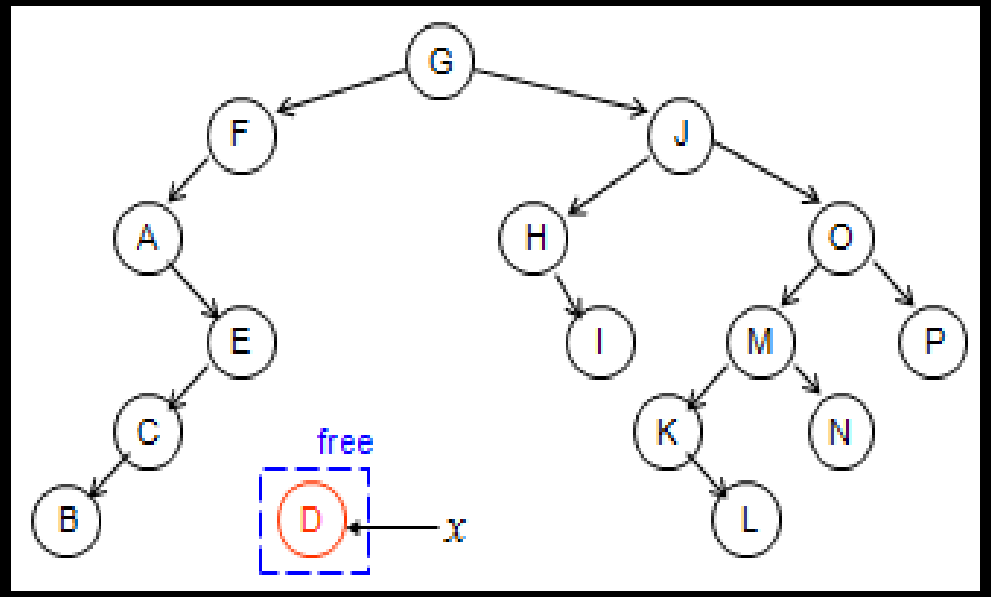
Deletion: BST

- When we delete a node, we need to consider how we **take care of the children of the deleted node**.
  - This has to be done such that the **property of the search tree is maintained**.
- Deletion under different cases
  - Case 1: the node is a leaf
  - Case 2: the node has one child
  - Case 3: the node has 2 children

Case 1

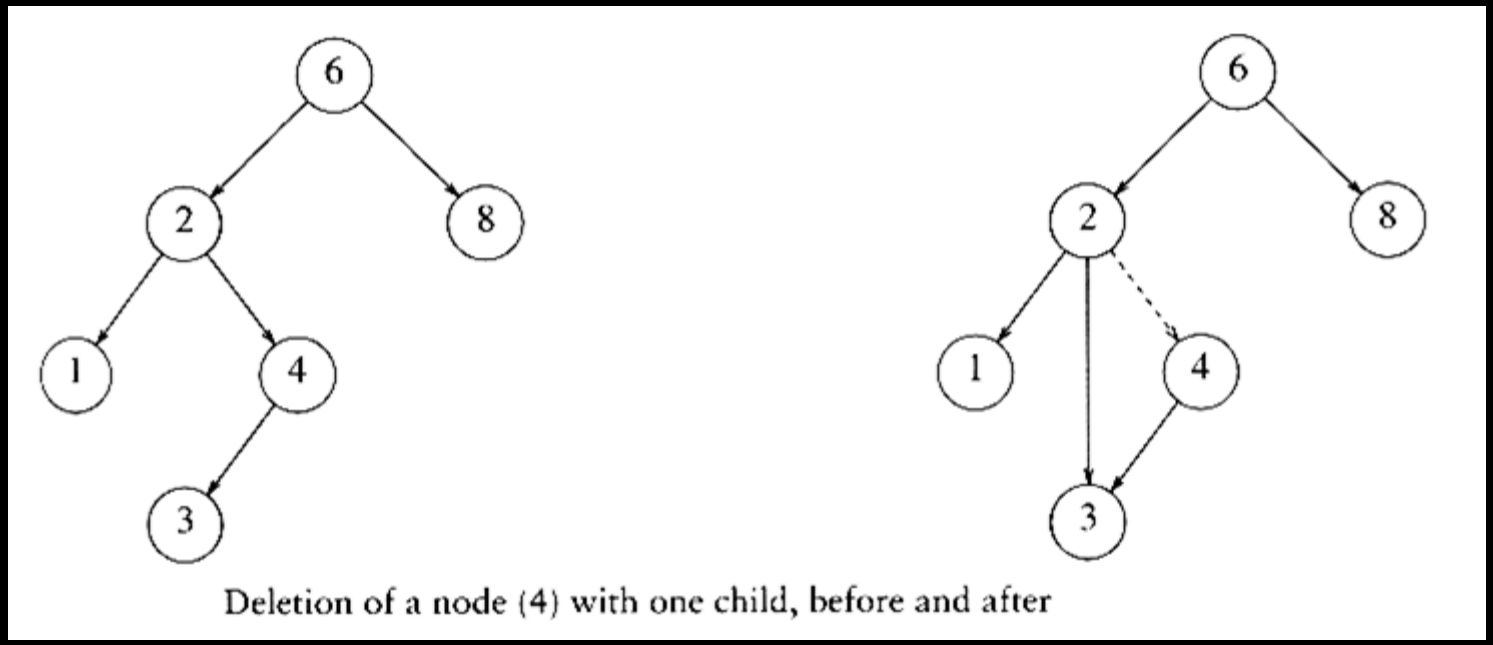
- **Case 1: the node is a leaf**
  - Delete it immediately





Case 2

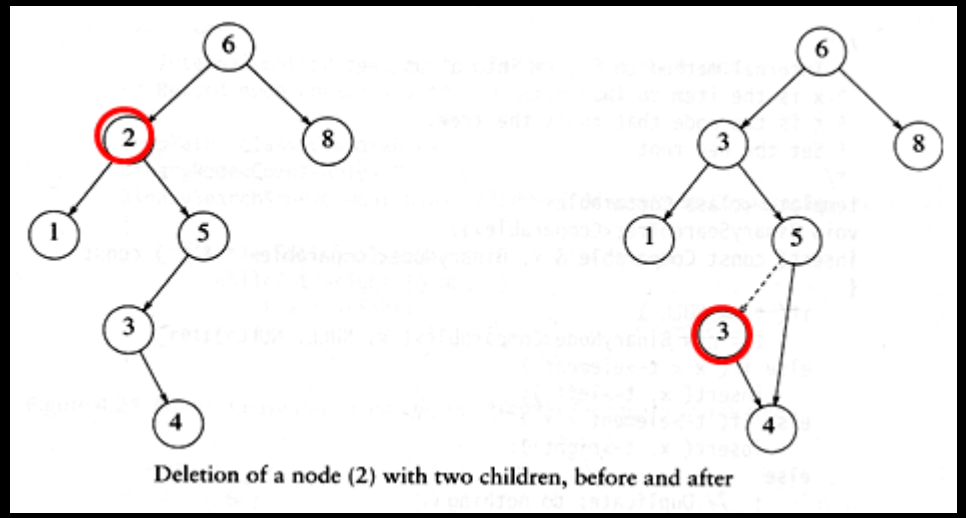
- **Case 2: the node has one child**
  - Adjust a pointer from the parent to bypass that node



Deletion of a node (4) with one child, before and after

Case 3

- **Case 3: the node has 2 children**
  - Replace the key of that node with the minimum element at the right sub-tree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.



Deletion of a node (2) with two children, before and after

Traversal: Binary Search Tree

- Many algorithms require all nodes of a binary tree be visited and the contents of each node processed or examined.
- There are 4 traditional types of traversals
  - Preorder traversal: process the root, then process all sub trees (left to right)
  - **In order traversal**: process the left sub tree, process the root, process the right sub tree
  - **Post order traversal:** process the left sub tree, process the right sub tree, then process the root
  - **Level order traversal:** starting from the root of a tree, process all nodes at the same depth from left to right, then proceed to the nodes at the next depth

Algorithm: Preorder Traversal

```
void preOrder(treePointer ptr)
{
    (ptr != NULL)
    {
        visit(t);
        preOrder(ptr->leftChild);
        preOrder(ptr->rightChild);
    }
}
```

Algorithm: In order Traversal

```
void inOrder(treePointer ptr)
{
    (ptr != NULL)
    {
            inOrder(ptr->leftChild);
            visit(t);
            inOrder(ptr->rightChild);
    }
}
```

Algorithm: Post order Traversal

```
void postOrder(treePointer ptr)
{
    (ptr != NULL)
    {
            postOrder(ptr->leftChild);
            postOrder(ptr->rightChild);
            visit(t);
    }
}
```

Algorithm: Level order Traversal

```
Let ptr be a pointer to the tree root.

while (ptr != NULL)

{

  visit node pointed at by ptr  and put its
 children on a FIFO queue;

  if FIFO queue is empty, set ptr = NULL;

  otherwise, delete a node from the FIFO queue
 and call it ptr;

}
```

Balancing a Tree

- BSTs where introduced because in theory they give nice fast search time.
  - Example1:
    - Design a BST using the following sequence of data: 6, 4, 8, 5, 7, 3, 9,10
  - Example2:
    - Design a BST using the following sequence of data: 3, 4, 5, 6, 7, 8 ,9 ,10
- We have seen that **depending on how the data arrives** the tree can degrade into a linked list
- So what is a good programmer to do.
  - Of course, they are to **balance the tree.**

Next Time!!