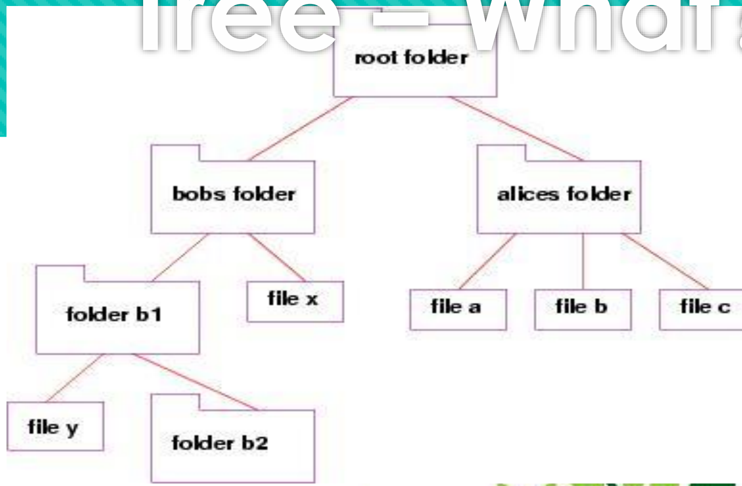
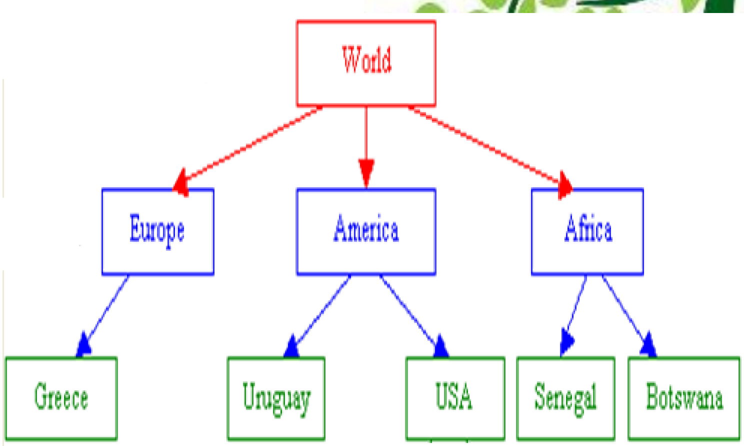
CS214-Data Structure

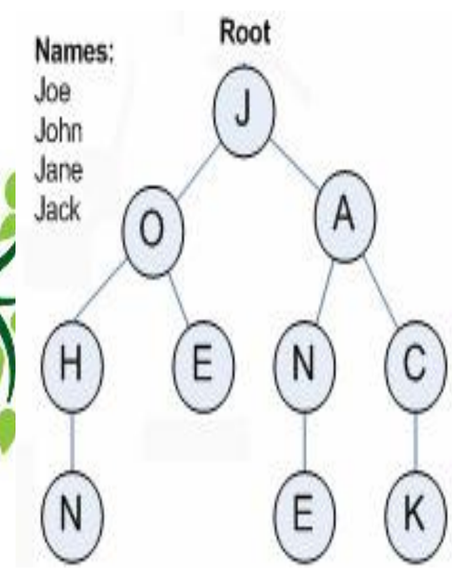
Lecturer: Dr. Salwa Osama

Trees

Tree – What?

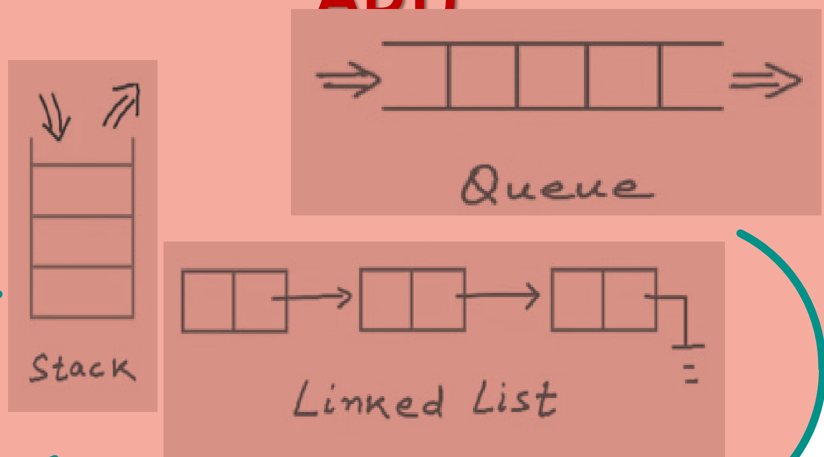

root folder
bobs folder
alices folder
folder b1
file x
file a
file b
file c
file y
folder b2

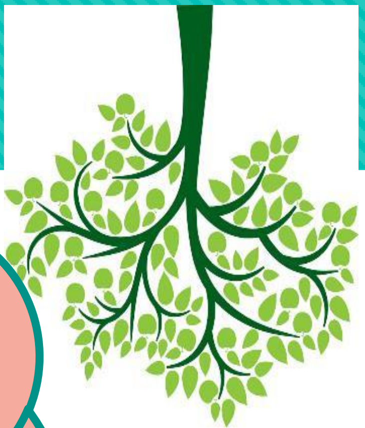

World
Europe
America
Africa
Greece
Uruguay
USA
Senegal
Botswana

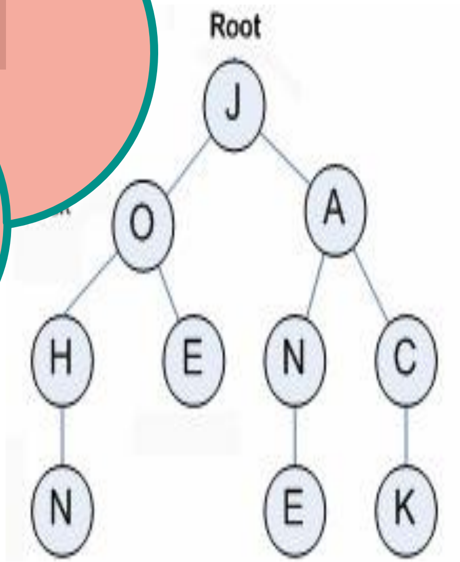

Names:
Joe
John
Jane
Jack
Root
J
O
A
H
E
N
C
N
E
K

Tree – What?

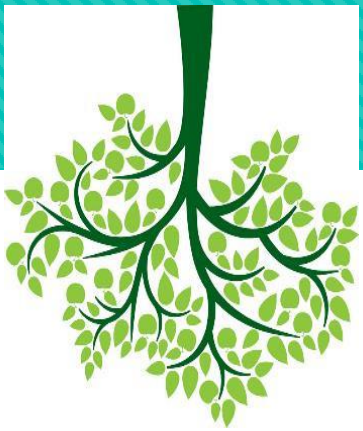
Remember (Linear ADT)



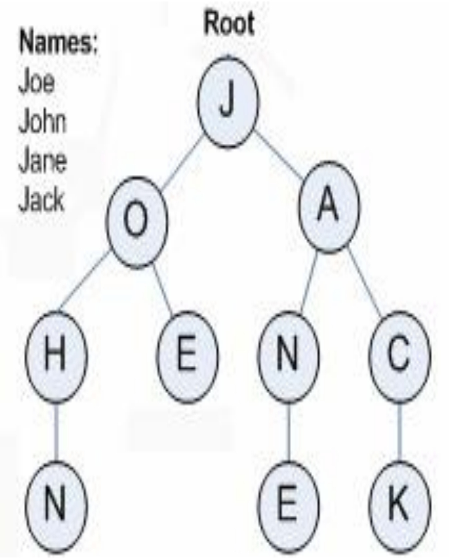




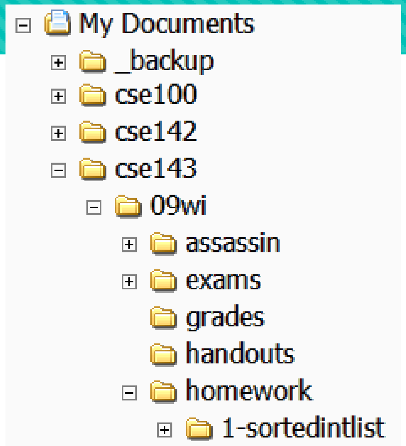
Tree – What?



- A tree is a collection of nodes.
- The collection **can be empty.**
- If not empty, a tree consists of:
  - ✓ a node **r (the root)**
  - ✓ zero or more nonempty subtrees T1, T2, ..., Tk, each of whose subtrees are connected by an edge from **r.**



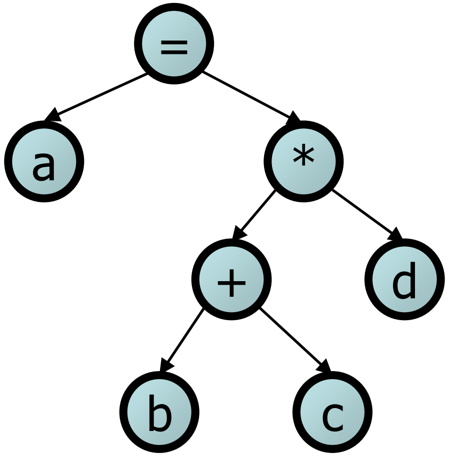
Trees in computer science

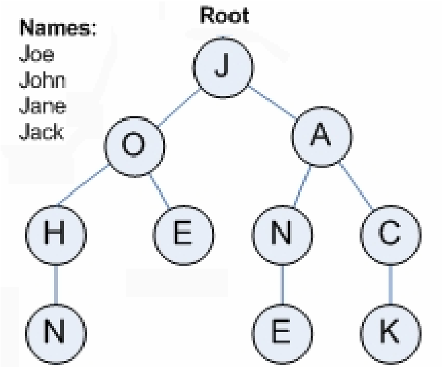


- Folders/files on a computer
- Family or organizational charts
- Compilers: expression parse Tree

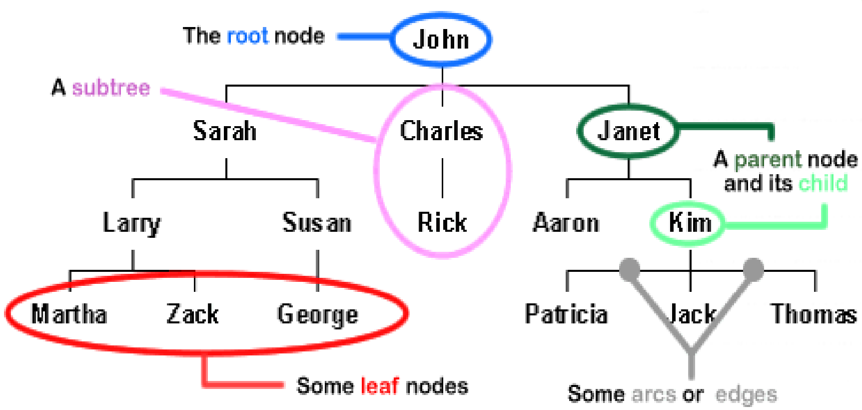
a = (b + c) * d;

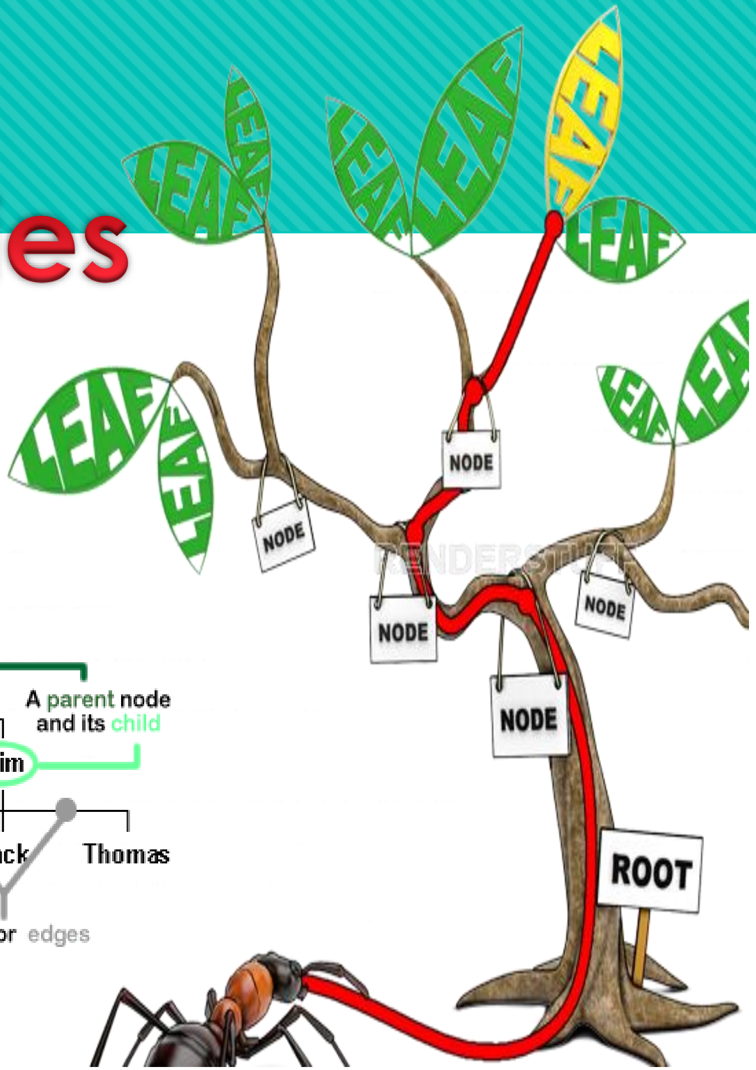
Decision Tree

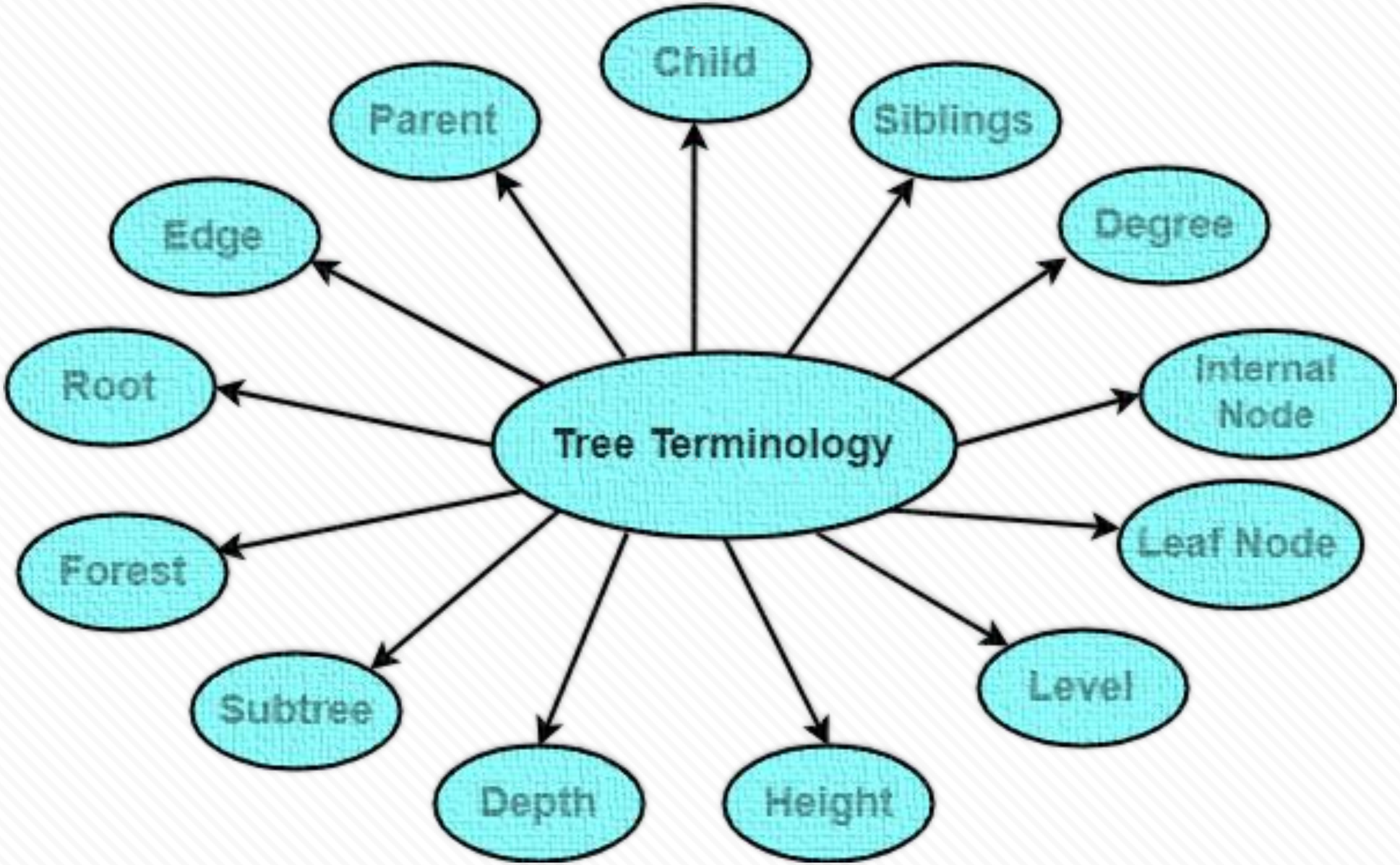




Terminologies

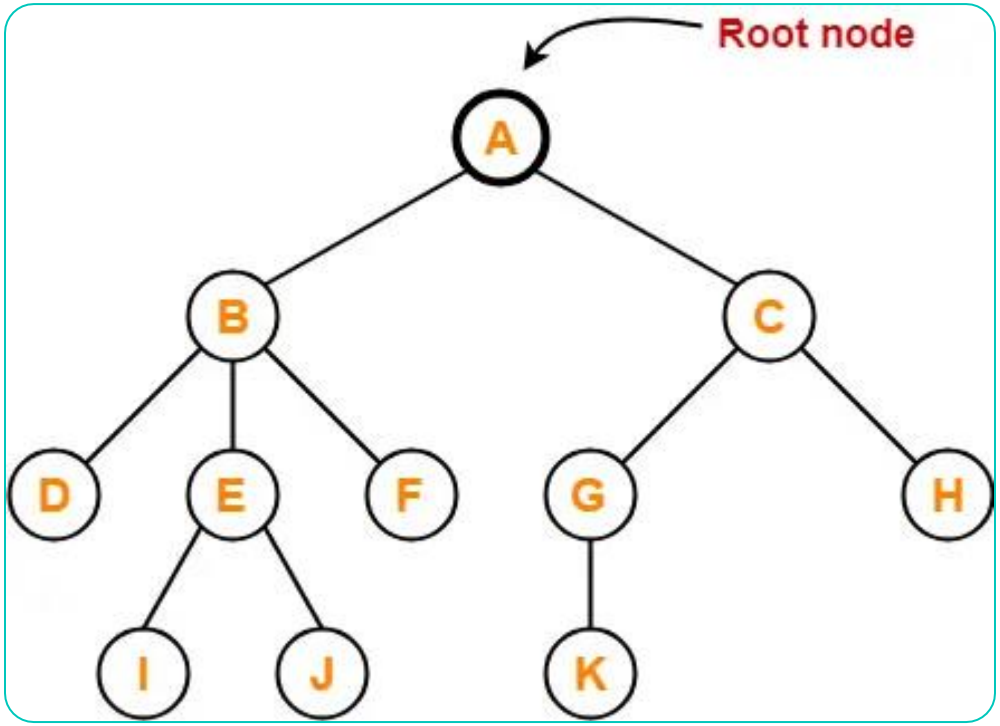





Child
Parent
Siblings
Edge
Degree
Root
Tree Terminology
Internal Node
Forest
Leaf Node
Subtree
Level
Depth
Height

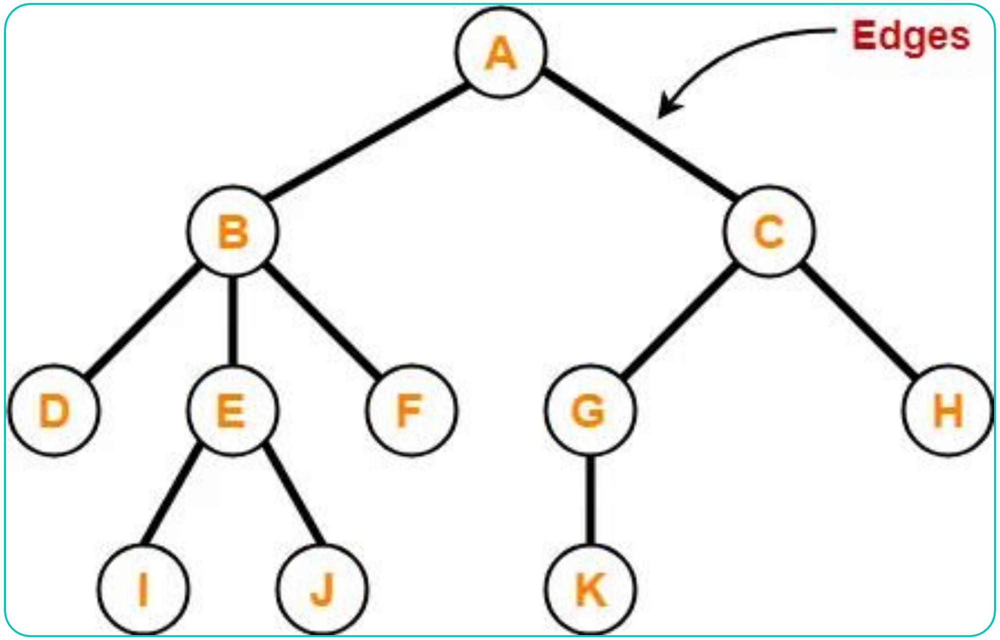
Root

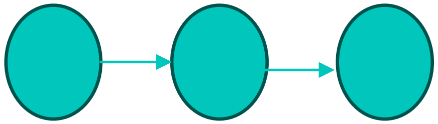
- The **first node** from where the tree originates is called as a root node.
- In any tree, there must be **only one root** node.
- We can **never have multiple root nodes** in a tree data structure.



Edge

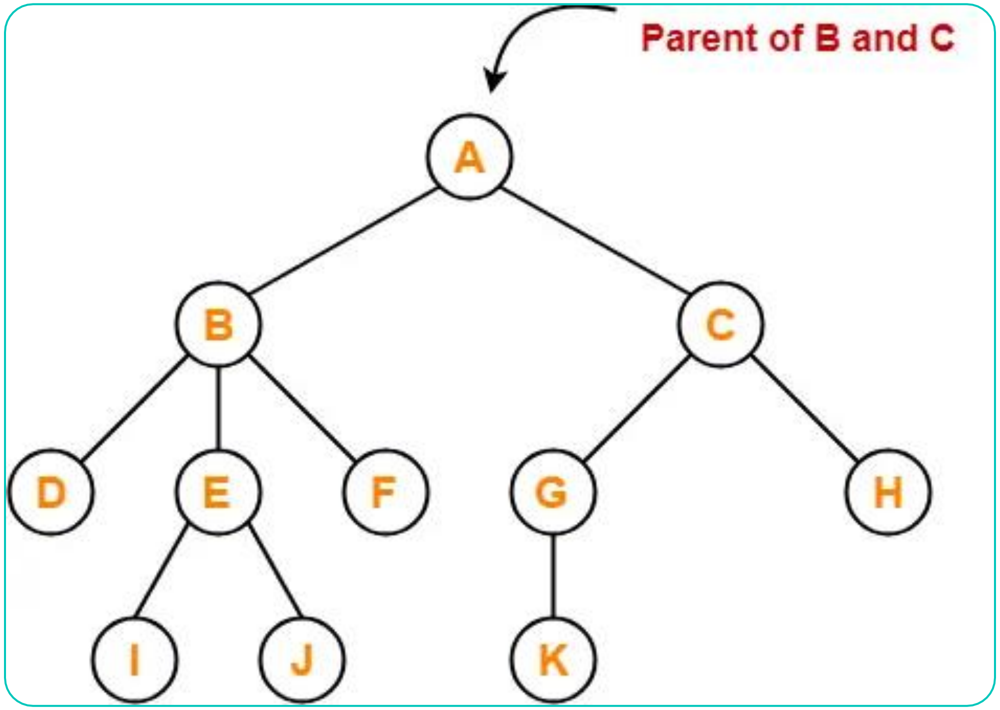
- The connecting link between any two nodes is called as an edge.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.





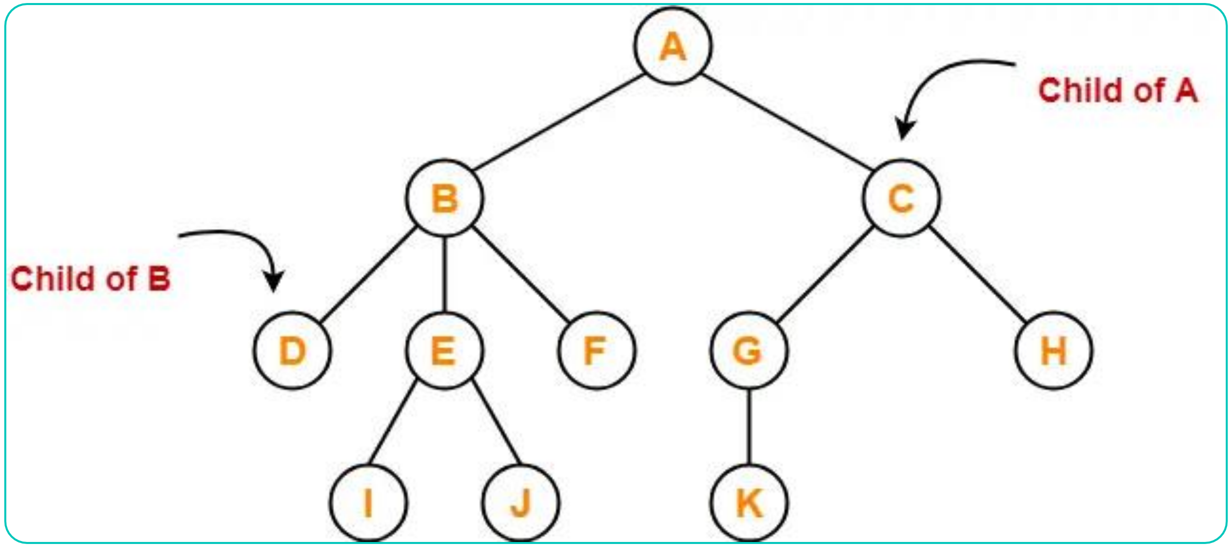
Parent

- The node which has a branch from it to any other node is called as a parent node.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



Child

- The node which is a descendant of some node is called as a child node.
- **All the nodes except root node are child nodes.**

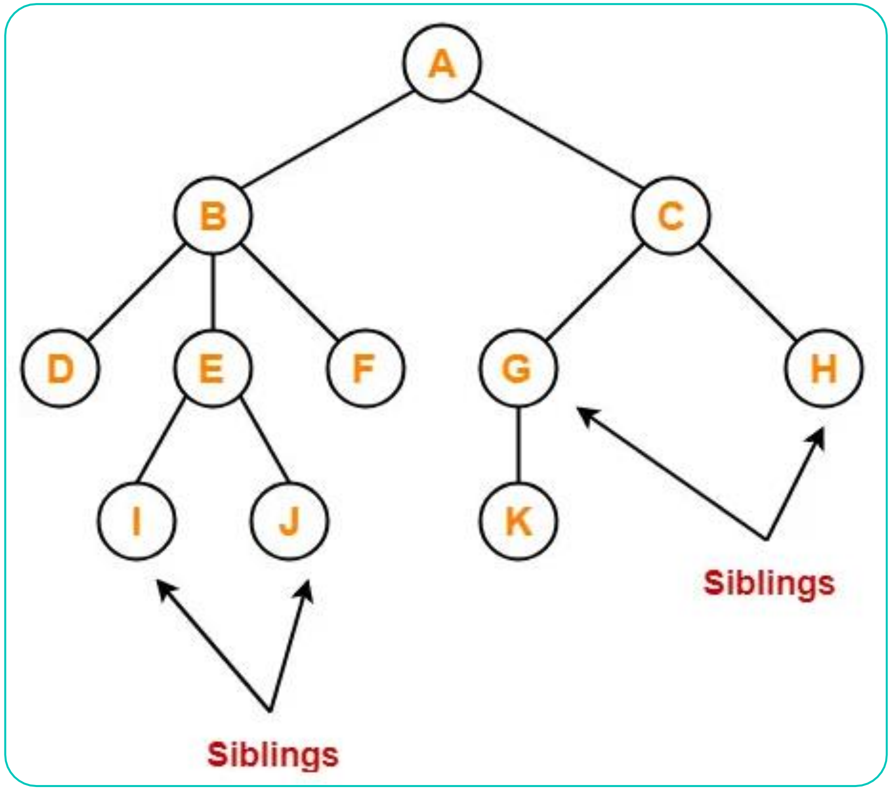


- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

Siblings

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

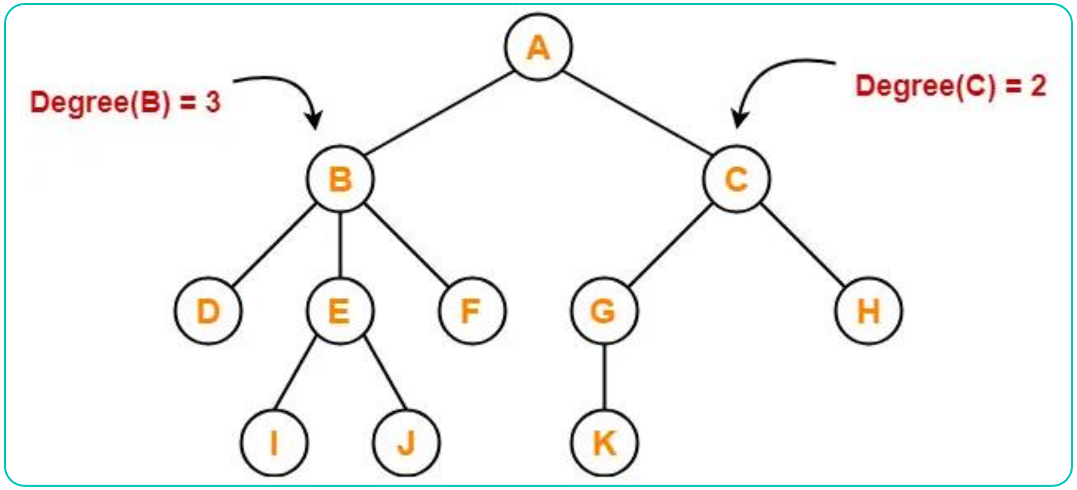
- Nodes which belong **to the same parent** are called as siblings.
- In other words, nodes with the same parent are sibling nodes.



Degree

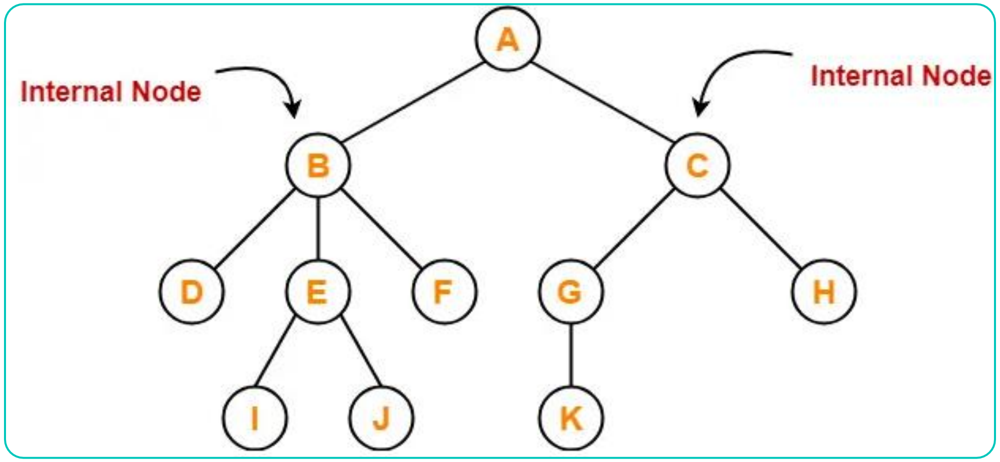
- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.



Internal Node

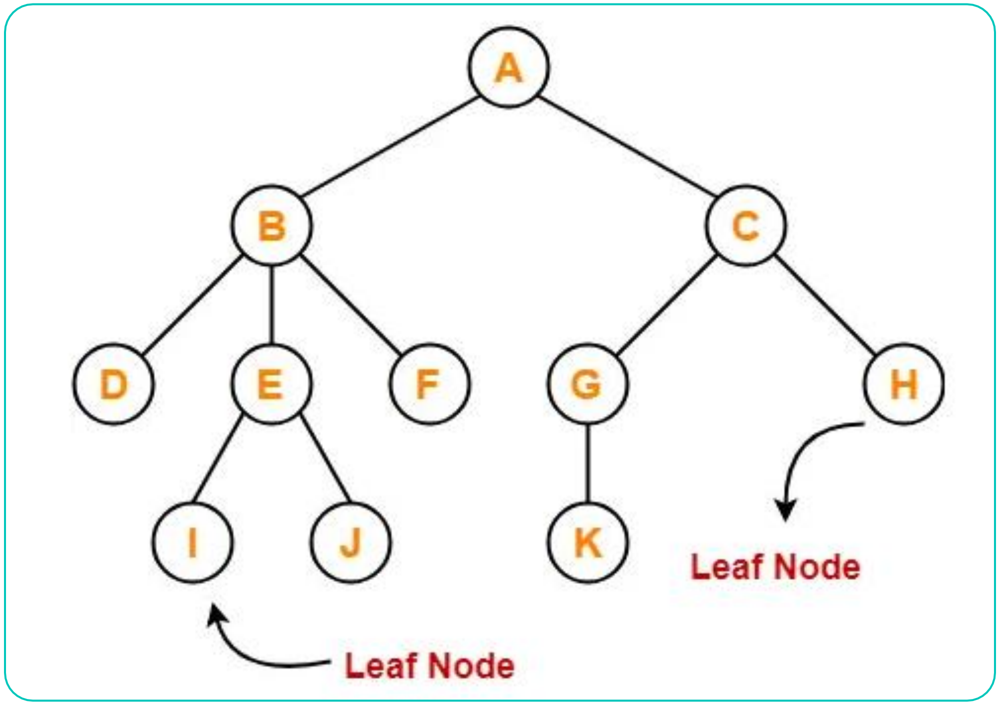
- The node which **has at least one child** is called as an internal node.
- Internal nodes are also called **as non-terminal** nodes.
- Every **non-leaf node** is an internal node.



nodes A, B, C, E and G are internal nodes.

Leaf Node

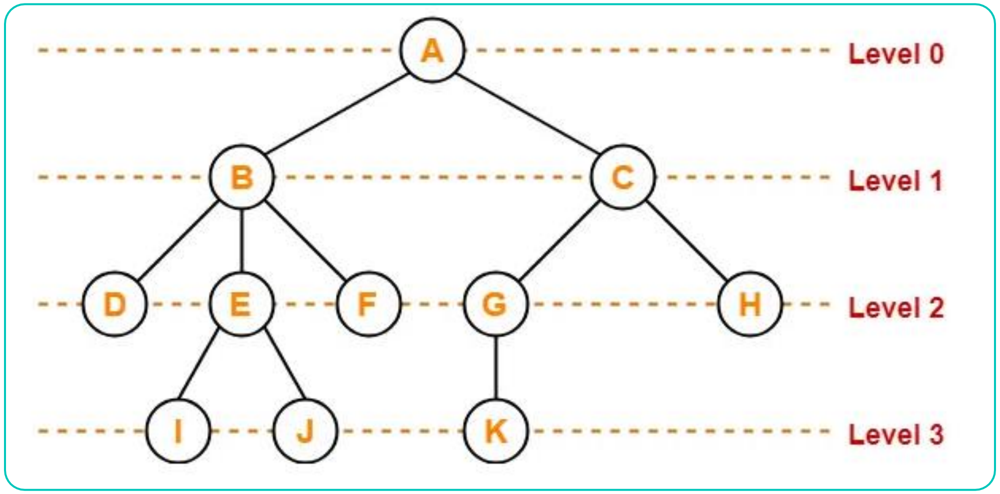
- The node which **does not have any child** is called as a leaf node.
- Leaf nodes are also called as **external nodes or terminal** nodes.



nodes D, I, J, F, K and H are leaf nodes.

Level

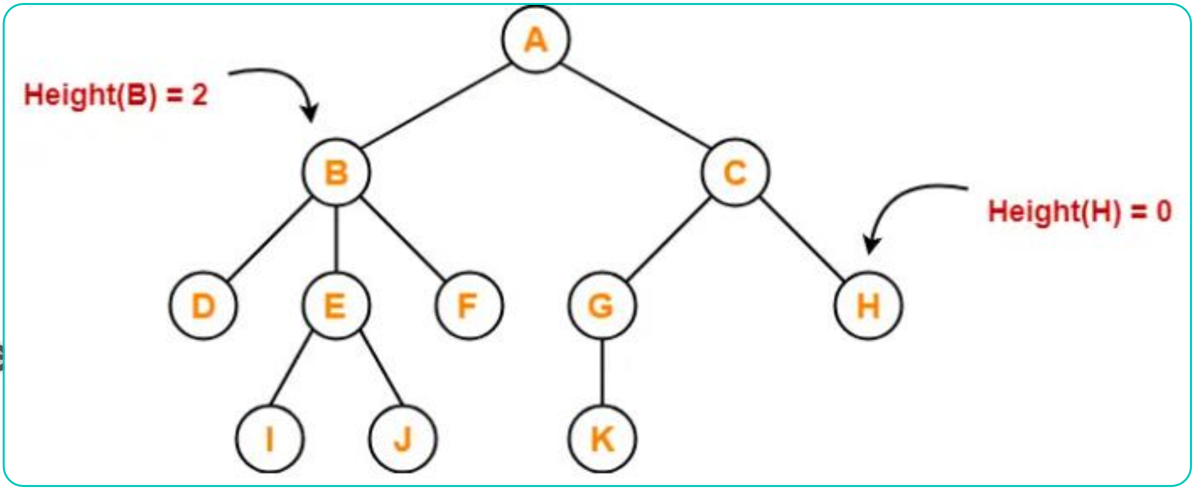
- In a tree, each **step from top to bottom** is called as level of a tree.
- The level count starts with **0** and increments by 1 at each level or step.



Height

- Total number of edges that lies on the **longest** path from any leaf node to a particular node is called as height of that node.
- **Height of a tree is the height of root node.**
- **Height of all leaf nodes = 0**

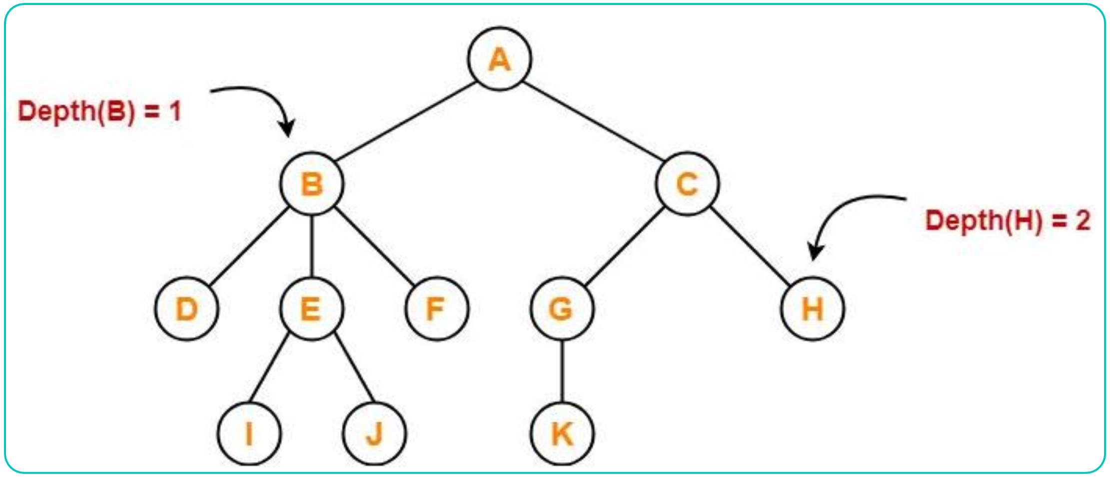
- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0



Depth

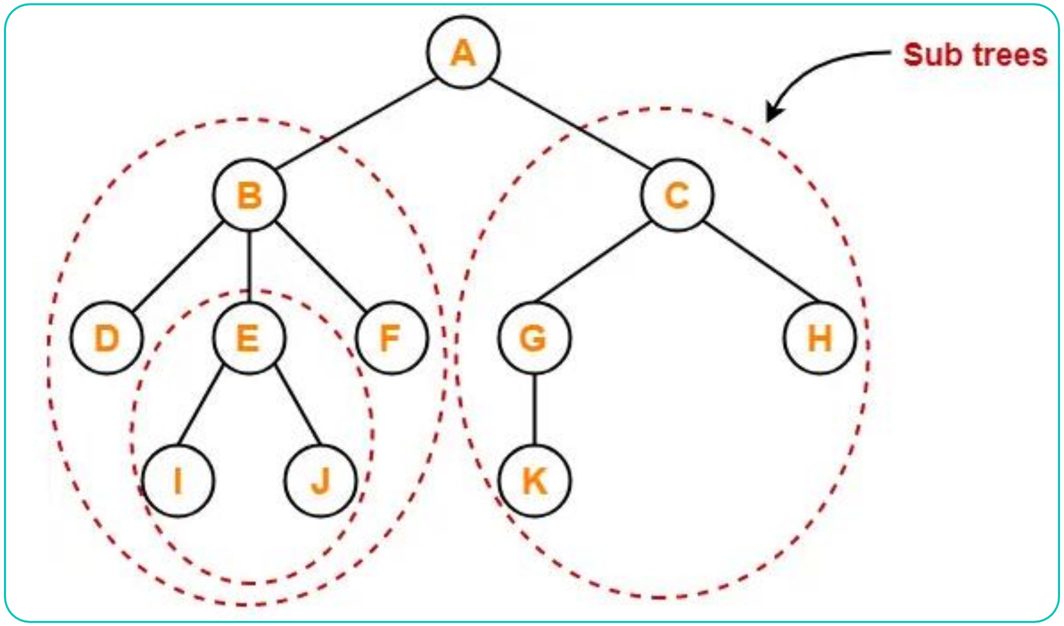
- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3

- Total number of edges from root node to a particular node is called as depth of that node.
- Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms "level" and "depth" are used interchangeably.

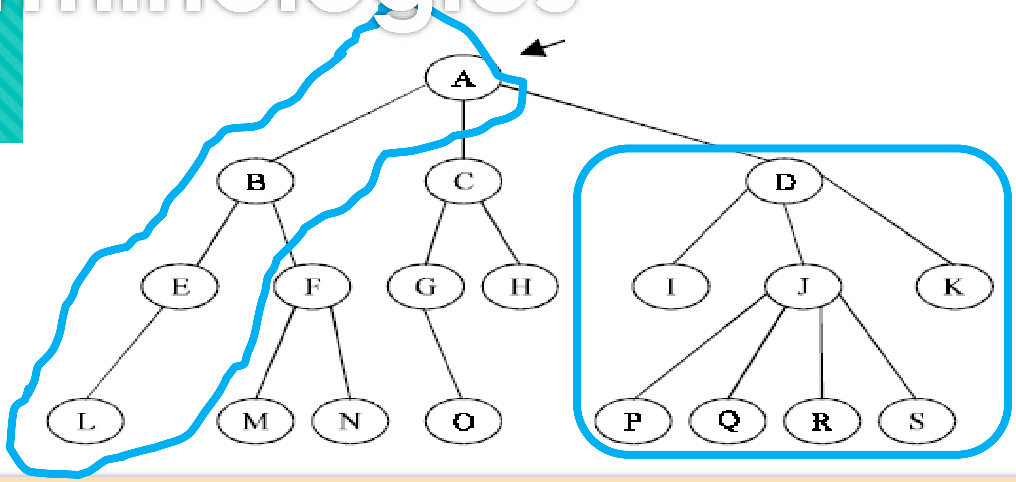


Subtree

- In a tree, each child from a node forms a subtree recursively.
- Every child node forms a subtree on its parent node.



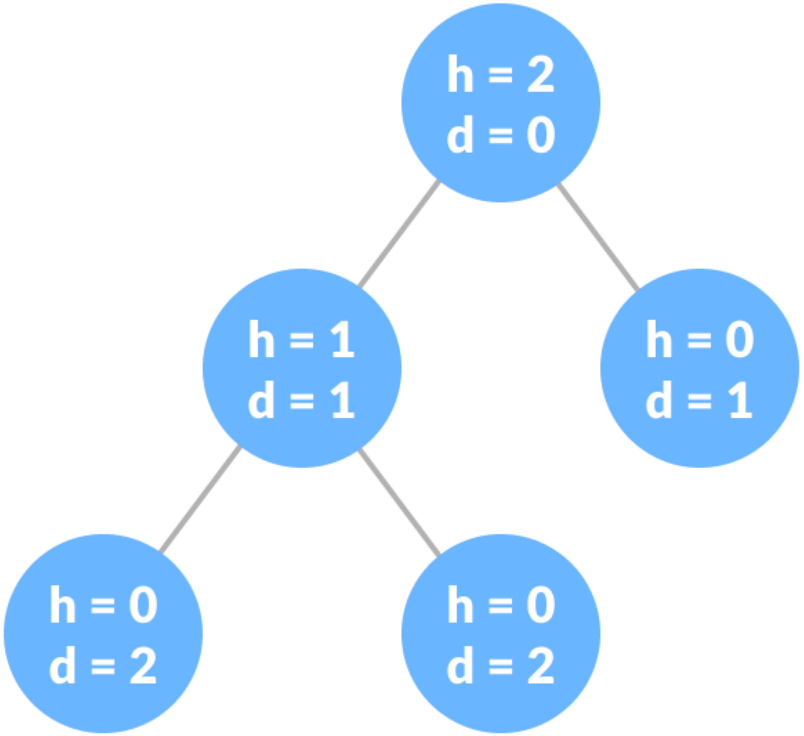
Terminologies



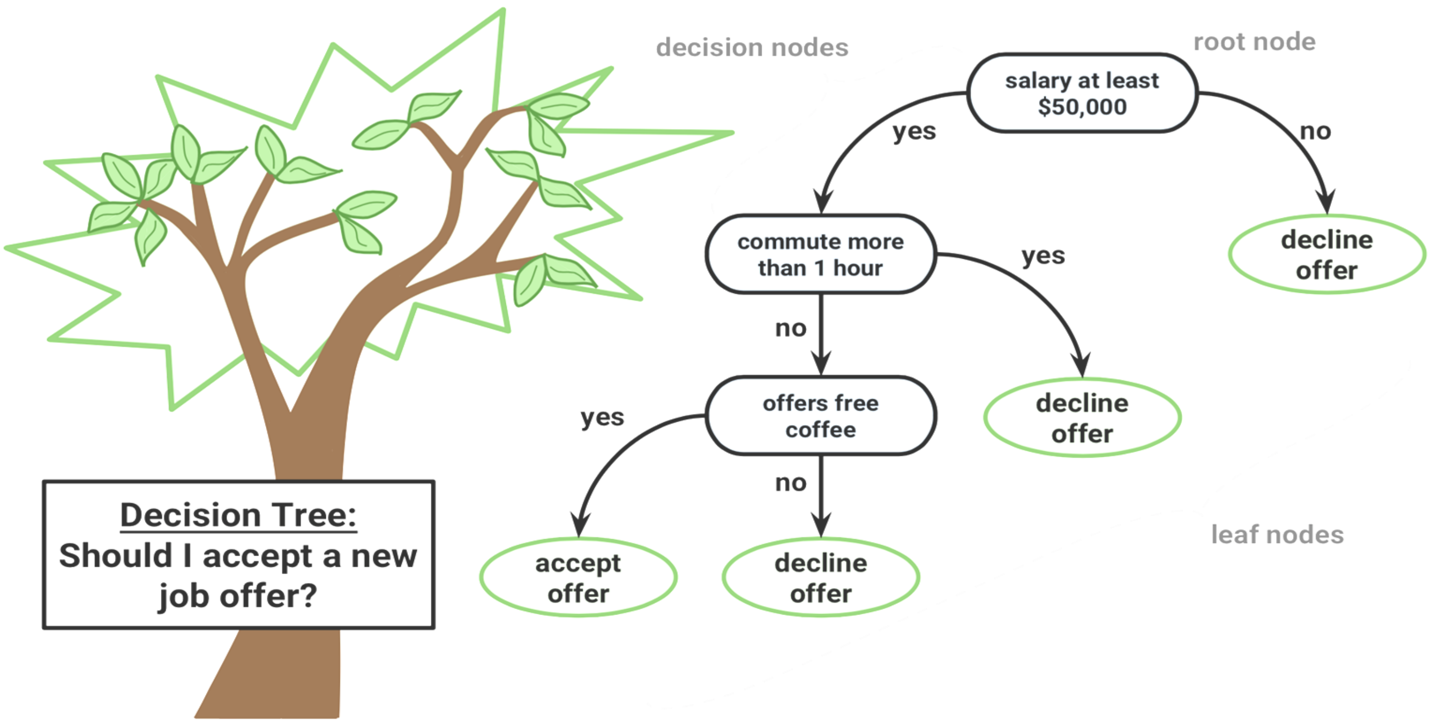
- **Path** : a sequence of edges.
- **Size** : the number of nodes in a tree.

Terminologies

- Height of a Tree
  - The height of a Tree is the height of the root node or the depth of the deepest node.



Binary Tree



Decision Tree:
Should I accept a new job offer?
decision nodes
root node
salary at least $50,000
yes
no
commute more than 1 hour
decline offer
yes
no
offers free coffee
decline offer
yes
no
accept offer
decline offer
leaf nodes

Binary Trees

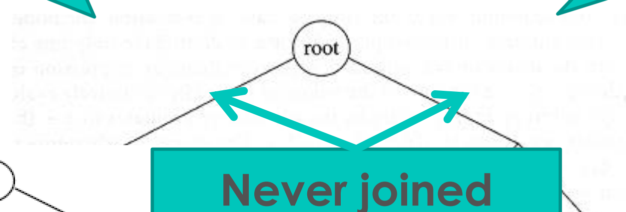
- A tree in which no node can have more than two children (tree of degree 2)
- …ee is an empty tree … node … wo children each of … inary tree

Left success or

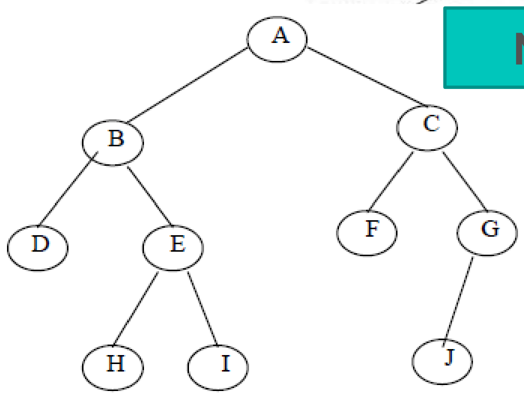
Right success or

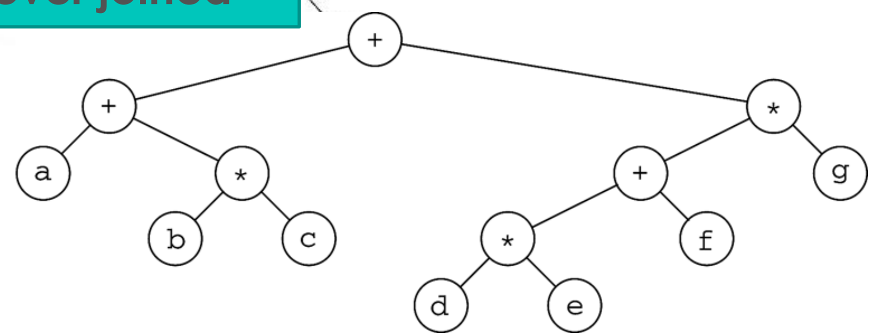
Examples

Generic binary tree



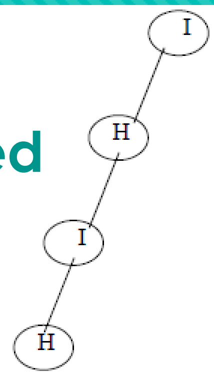
Never joined

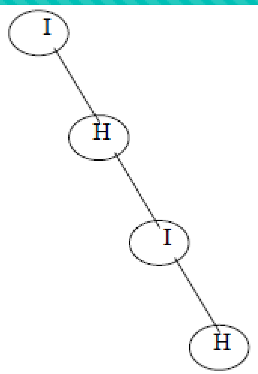




Binary Tree

Left Skewed

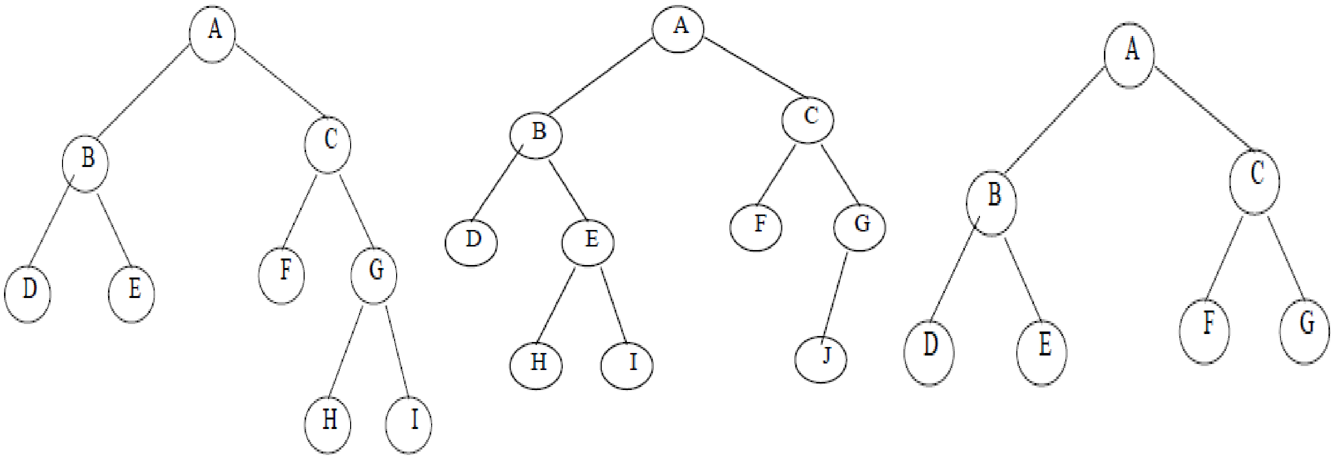




Right Skewed

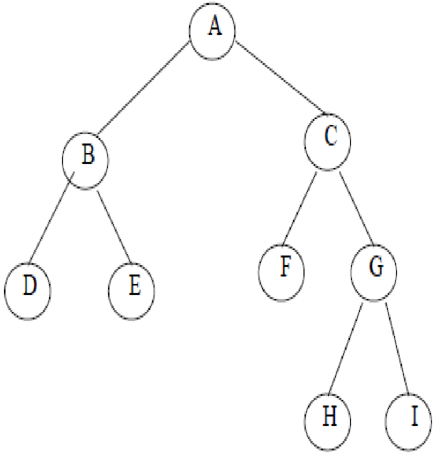
- If a binary tree has **only right sub trees**, then it is called **right skewed** binary tree.
- If a binary tree has **only left sub trees**, then it is called **left skewed** binary tree.

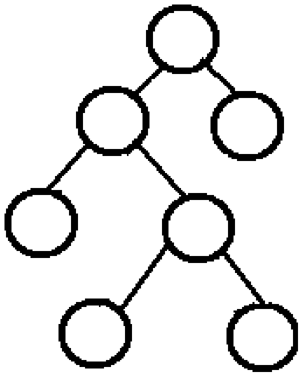
Complete Binary Tree

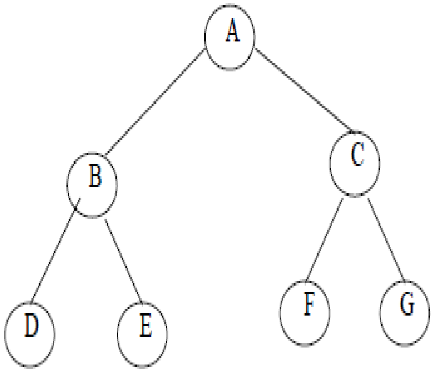


- a binary tree in which every level, except possibly the last, is completely filled- or has 2^L node.

Full (Strictly) Binary Tree

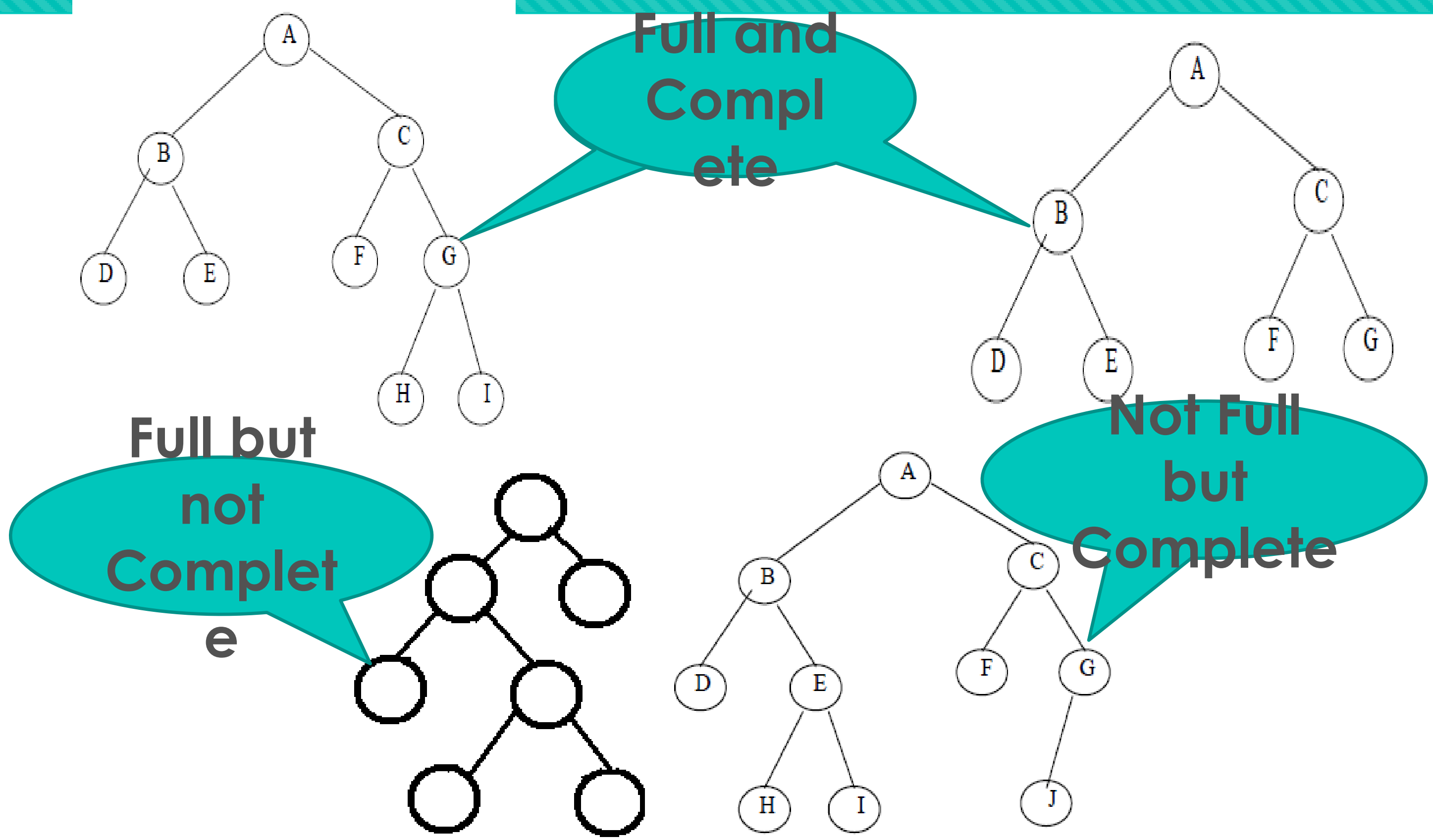






- A binary tree in which every node other than the leaves has exactly two children.

Binary Tree


A
B
C
D
E
F
G
H
I
Full and
Compl
ete
A
B
C
D
E
F
G
Full but
not
Complet
e
A
B
C
D
E
F
G
H
I
J
Not Full
but
Complete

Binary Tree Traversals

- **Traversal**: An examination of the elements of a tree.

- Common orderings for traversals:
  - **pre-order**: process root node, then its left/right subtrees
  - **in-order**: process left subtree, then root node, then right
  - **post-order**: process left/right subtrees, then root node

Traversal example

- in-order (LVR):

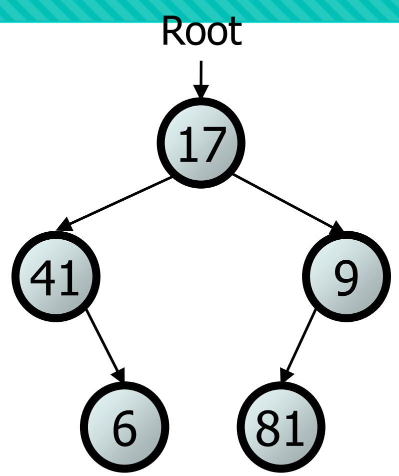
  41

  6

  17

  81

  9



Traversal example

- pre-order (VLR):

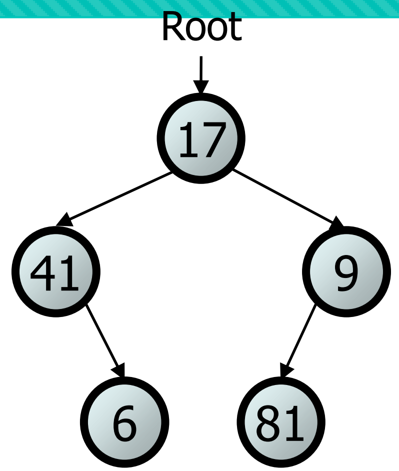
  17

  41

  6

  9

  81



Traversal example

- post-order (LRV):

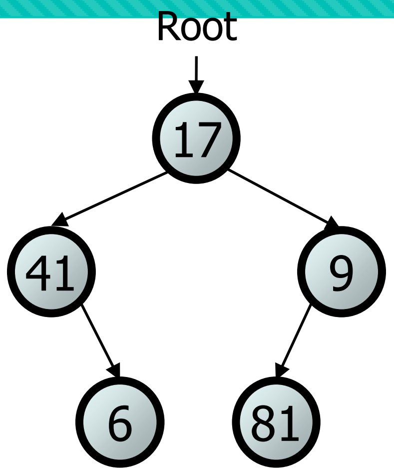
  6

  41

  81

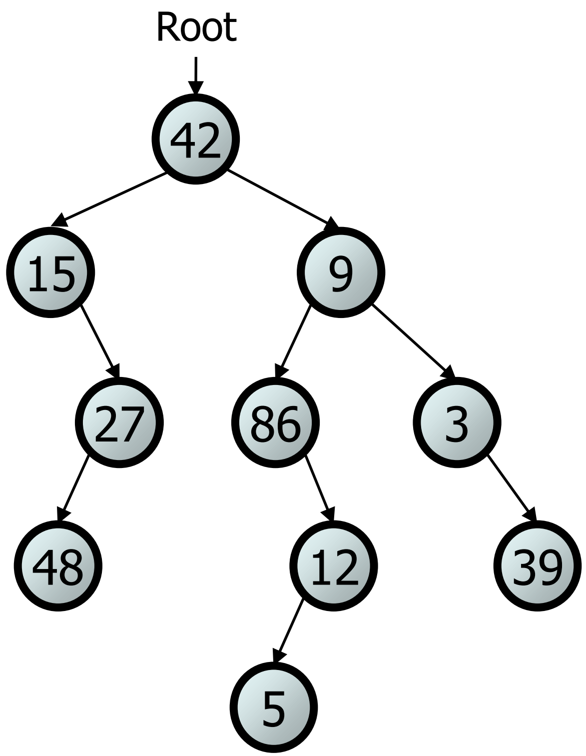
  9

  17



Exercise

- **Pre-order:**
- **In-order:**
- **Post-order:**



Exercise

- **Pre-order:**

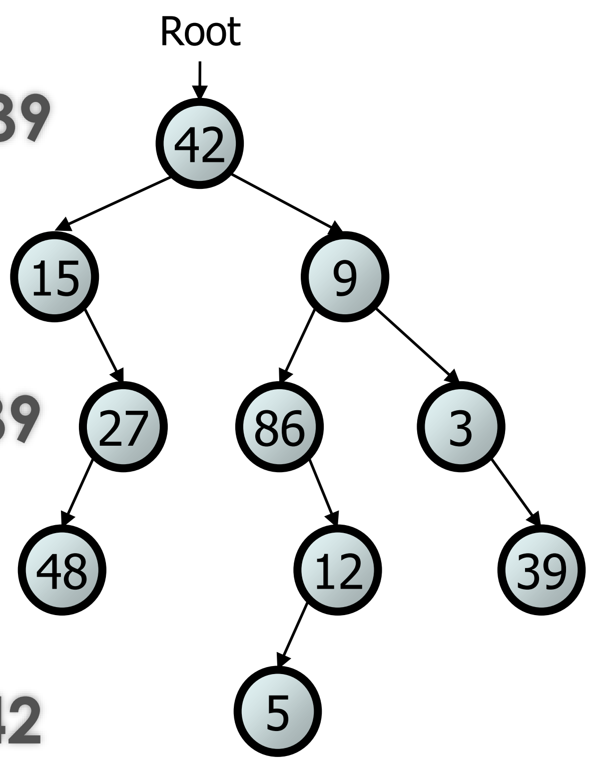
  42 15 27 48 9 86 12 5 3 39

- **In-order:**

  15 48 27 42 86 5 12 9 3 39

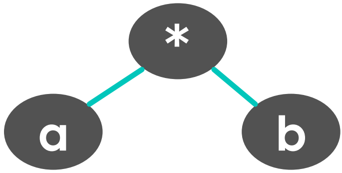
- **Post-order:**

  48 27 15 5 12 86 39 3 9 42



Example: Expression Trees

- It is **a binary tree** contains an **arithmetic expression** with some operators and operands.
- **Leaves** are **operands** (constants or variables)
- **The internal nodes** contain **operators**
- For each node contains an operator, its **left subtree** gives the **left operand**, and its **right subtree** gives the **right operand.**



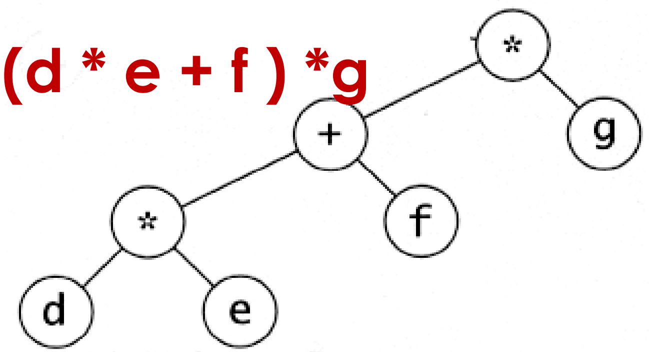
a* b

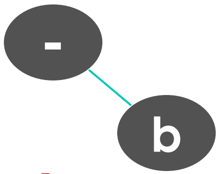
Example: Expression Trees

- **Building Expression Trees** has great importance in syntactical analysis and parsing, along with the validity of expressions

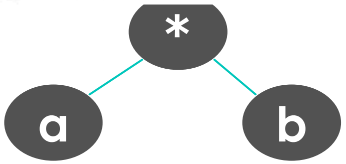
Example: Expression Trees

(d * e + f) *g

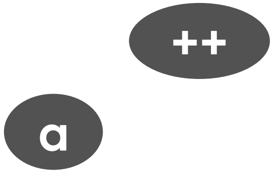




- b

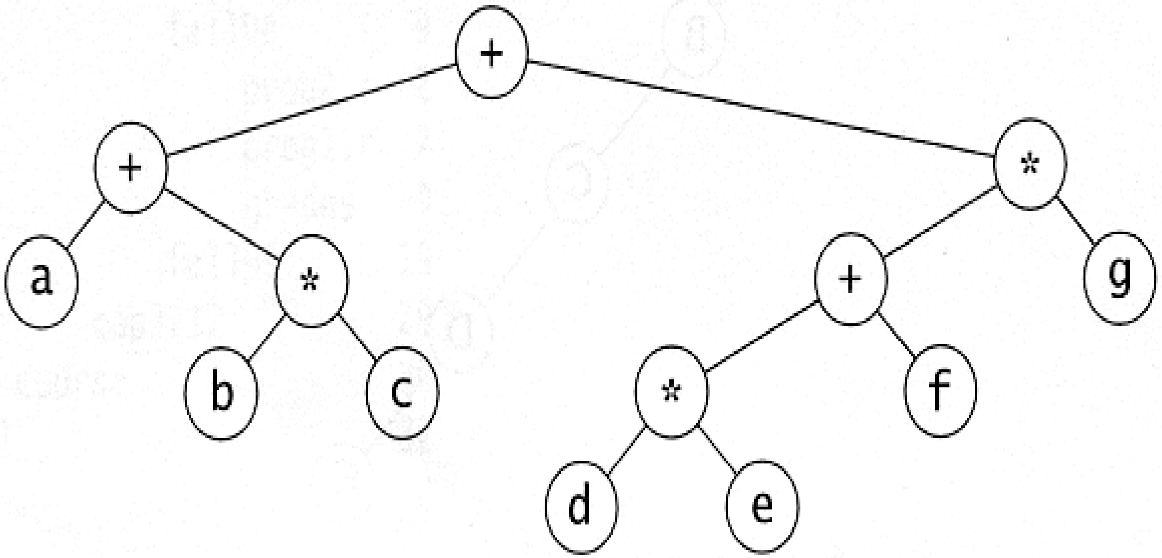


a* b

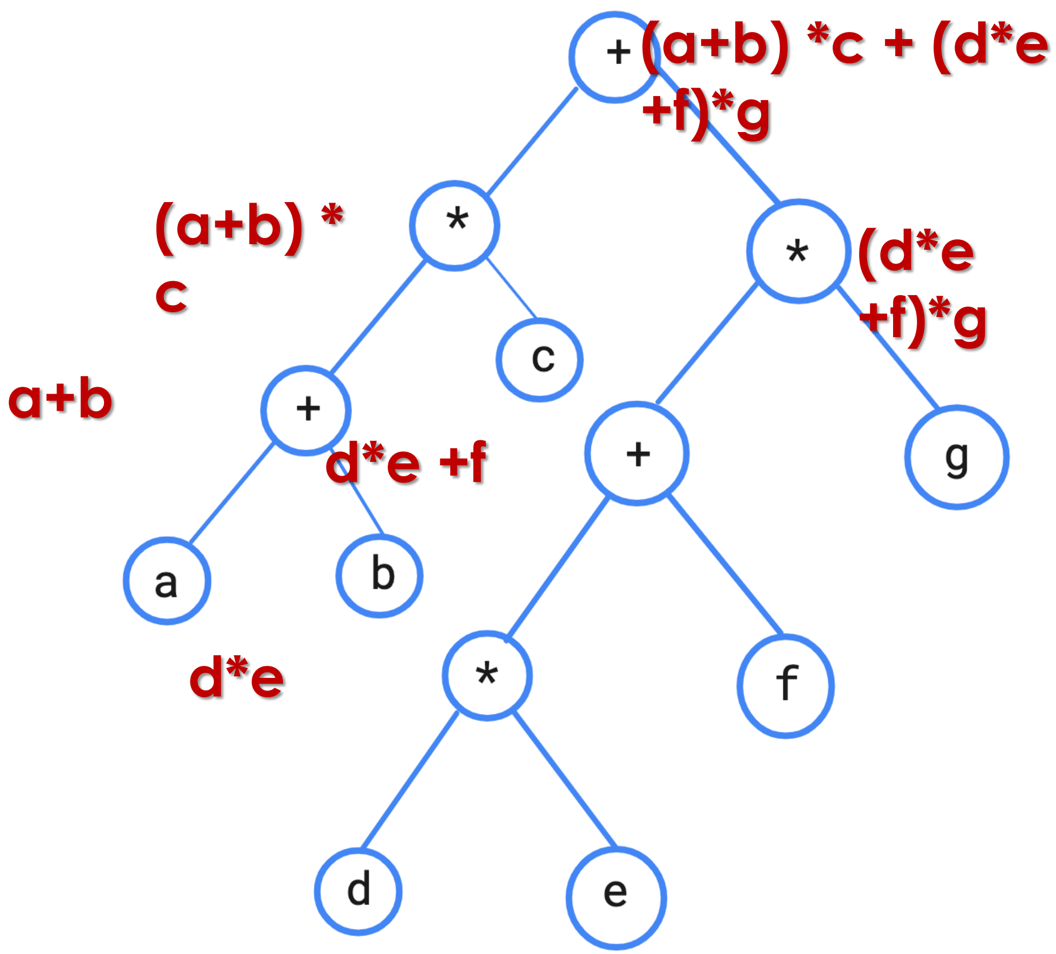


a++

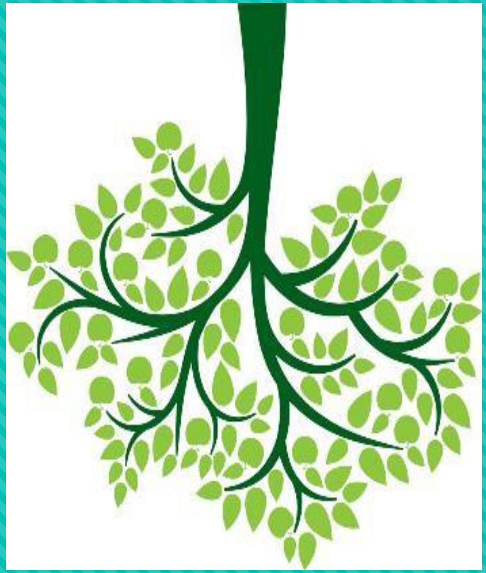
Example: Expression Trees



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


(a+b) *c + (d*e +f)*g
(a+b) * c
(d*e +f)*g
a+b
d*e +f
d*e
+
*
*
+
c
+
g
a
b
*
f
d
e

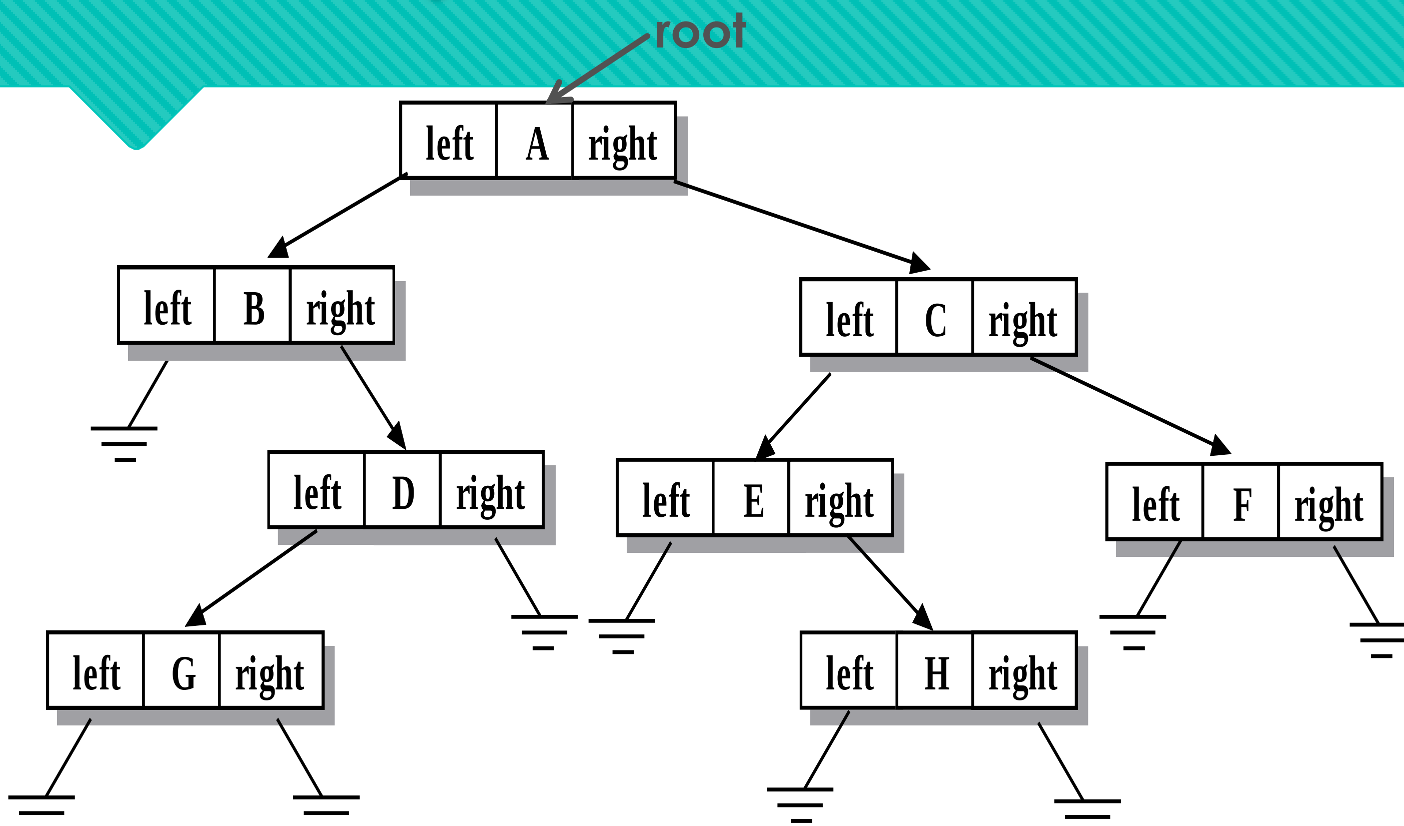
Tree



Implementation

User View

Tree Implementation



Tree Node Model

Tree Implementation

```
typedef struct node {
    EntryType      info;
    struct node    *right;
    struct node    *left;
} NodeType;

typedef  NodeType * TreeType
```

Tree Implementation

Pre: None.

Post: The tree is initialized to be empty.

```
void CreateTree(TreeType *t){*t=NULL;}
```

Pre: The tree is initialized.

Post: If the tree is empty (1) is returned. Else (0) is returned.

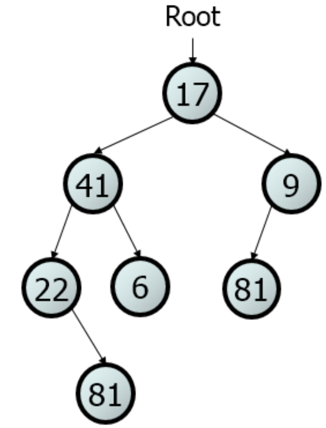
```
int EmptyTree(TreeType t){eturn (!t);}
```

Pre: The tree is initialized.

Post: If the tree is full (1) is returned. Else (0) is returned.

```
int FullTree(TreeType t){ return 0;}
```

Tree Implementation



Pre: The tree is initialized.

Post: The tree has been been traversed in infix order sequence.

```
void Inorder(TreeType t,
   void(*pvisit)(EntryType*)){

   Stack s;    NodeType *p=t;

   if(p){

      CreateStack(&s);

      do{

         while(p){Push(p, &s);    p=p->left;}

         Pop(&p, &s);

        (*pvisit)(&p->info);

         p=p->right);

      }while(!StackEmpty(&s) || p);

   }}}
```

Tree Implementation

Pre: The tree is initialized.

Post: The tree has been been traversed in infix order sequence.

```
void Inorder(TreeType t,
      void(*pvisit)(EntryType*)){
      if(t){
      Inorder(t->left, pvisit);
      (*pvisit)(&(t->info));
      Inorder(t->right, pvisit);
   }
}
```

Tree Implementation

Pre: The tree is initialized.

Post: The tree has been been traversed in prefix order sequence.

```
void Preorder(TreeType t,
  void(*pvisit)(EntryType*)){
      if(t){
      (*pvisit)(&(t->info));
      Preorder(t->left, pvisit);
      Preorder(t->right, pvisit);
    }
}
```

Tree Implementation

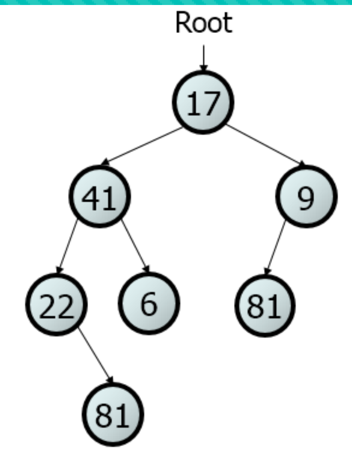
Pre: The tree is initialized.

Post: The tree has been been traversed in Postfix order sequence.

```
void Postorder(TreeType t,
  void(*pvisit)(EntryType*)){
      if(t){
      Postorder(t->left, pvisit);
      Postorder(t->right, pvisit);
   (*pvisit)(&(t->info));
   }
}
```

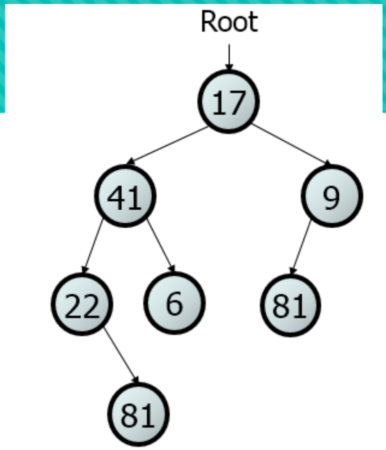
Tree Implementation

```
int Size(TreeType t){
    if (!t)
        return 0;
    return (1+Size(t->left)+
            Size(t->right));
}
```



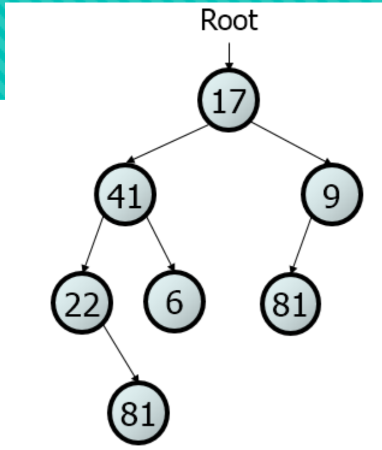
Tree Implementation

```
int height(TreeType t){
   if (!t)
         return 0;
    int a=height(t->left);
    int b=heigth(t->right);
    return (a>b)? 1+a : 1+b;
}
```



Tree Implementation

```
void ClearTree(Tree *t){
  if (*t){
   ClearTree(&(*t)->left);
   ClearTree(&(*t)->right);
   free(*t);
   *t=NULL;
  }
}
```




Any
Questions