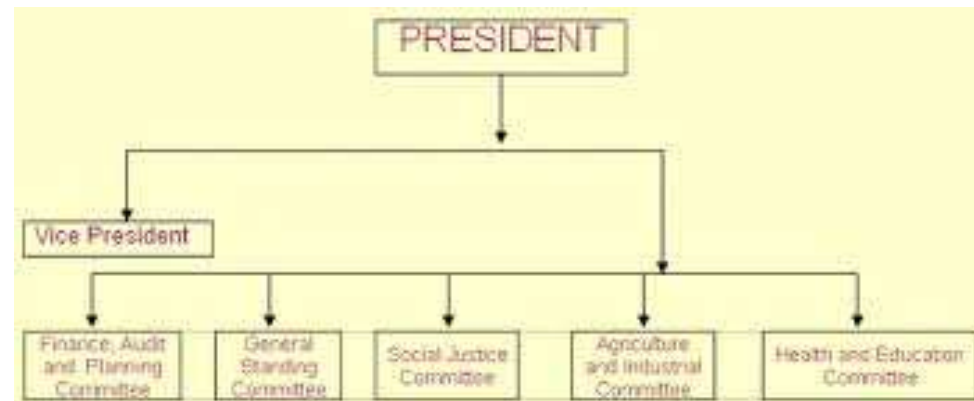


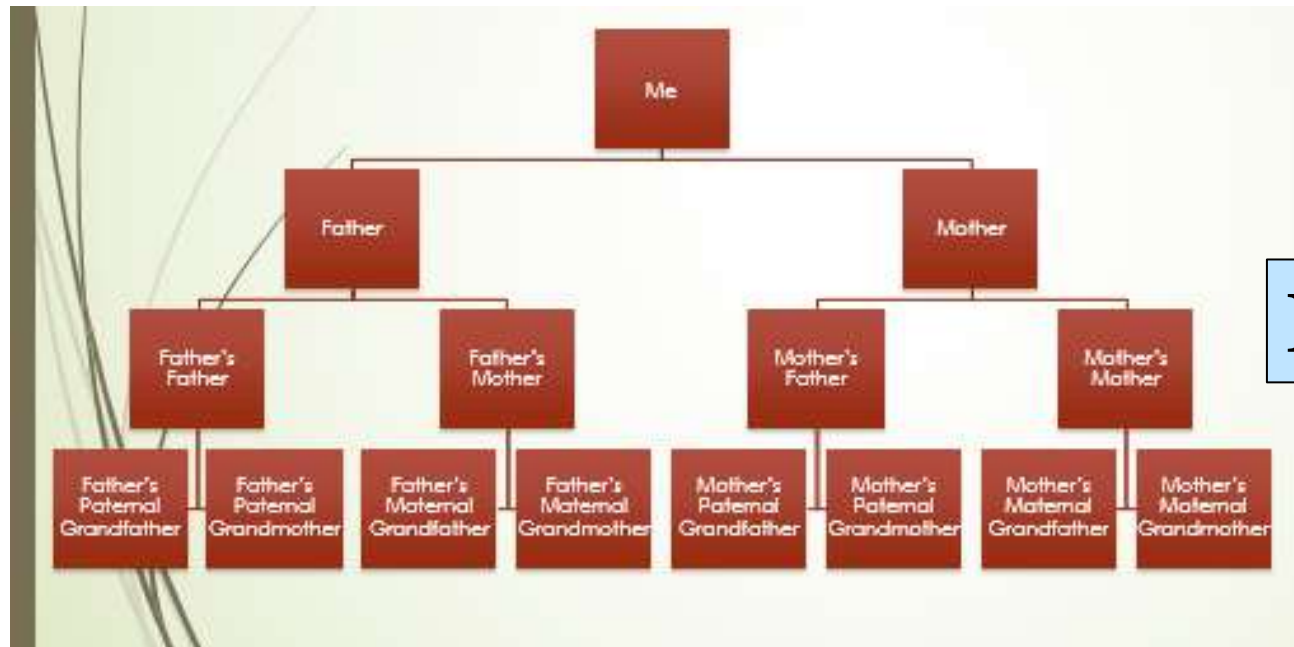
TREES

# Issues with Lists

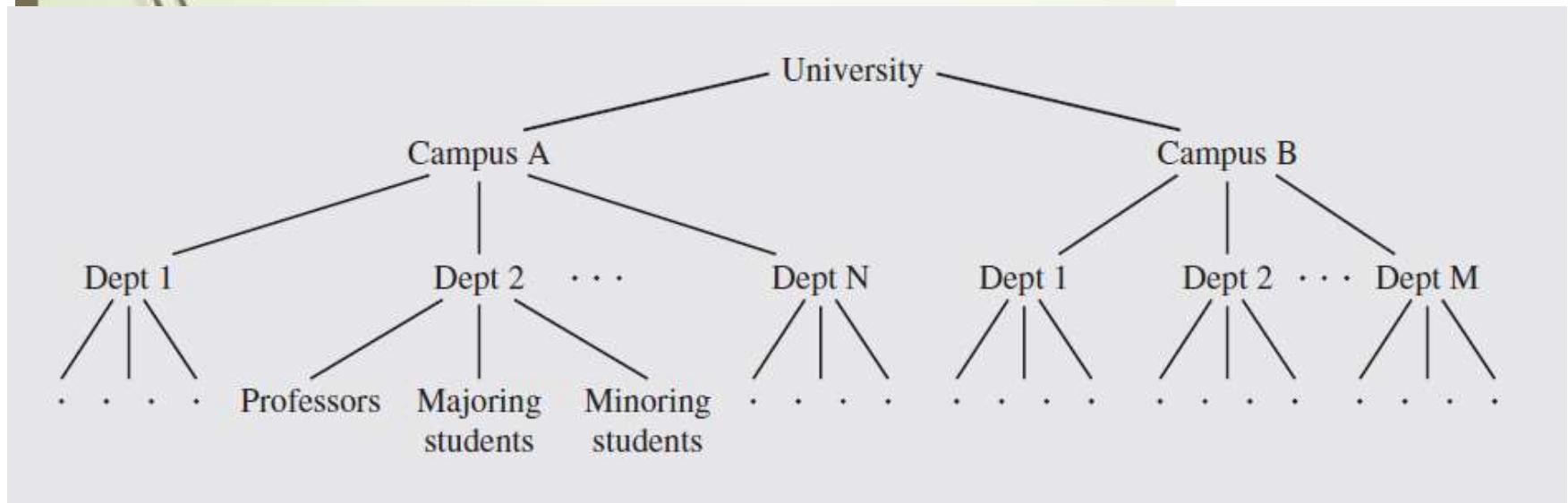
- Linked lists provide greater flexibility than Arrays
- **BUT** are linear so it is difficult to use them to organize a *hierarchical representation* of objects.
- How we represent hierarchy in real life ?



# Trees to represent *hierarchy*



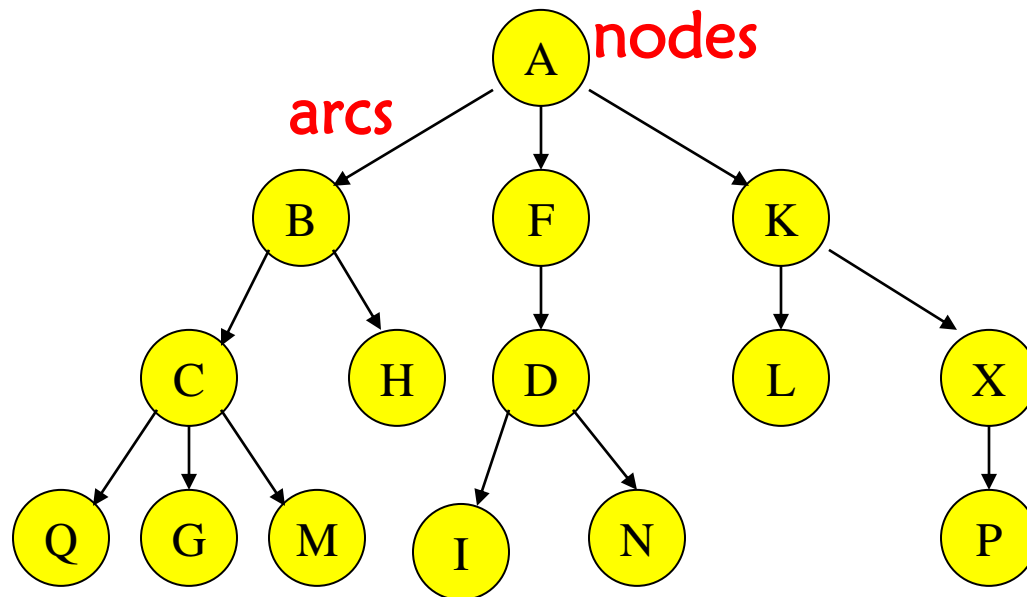
## EXAMPLES



# Trees for Hierarchical representation

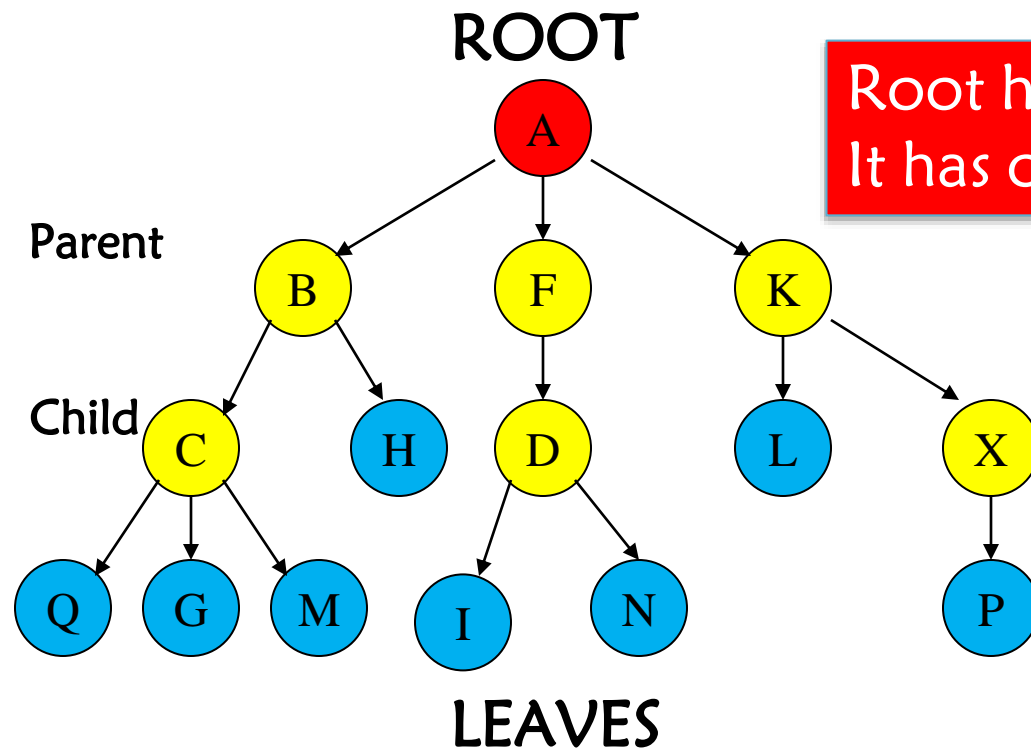
Trees overcome the limitation of linked list

Tree is a new data type that consists of *nodes* and *arcs*.



# Trees for Hierarchical representation

- Unlike natural trees, DS trees are **depicted upside down**
  - with the *root* at the top and
  - the *leaves* (*terminal nodes*) at the bottom.



Root has no parent.  
It has only child nodes.

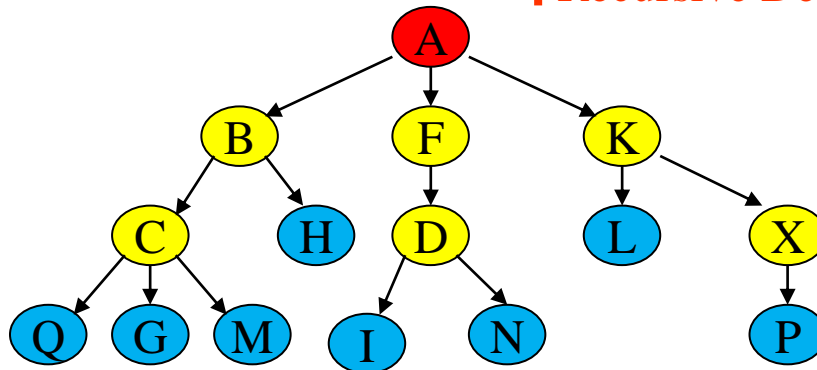
Leaves have no  
children

# Tree - Definition

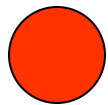
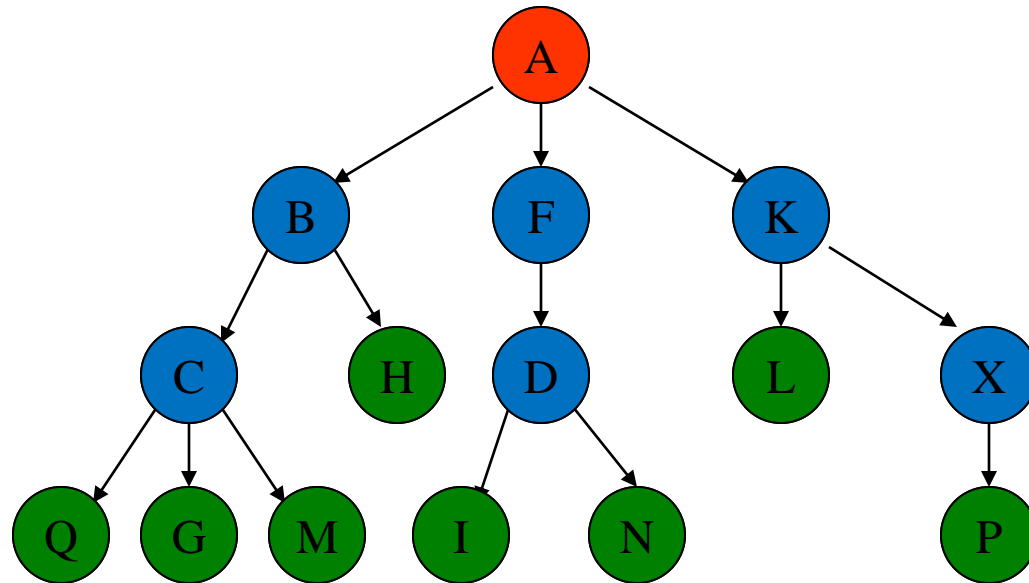
A tree is a finite set of one or more nodes such that:

1. There is a specially designated node, the *root*.
2. The remaining nodes are partitioned in  $n \geq 0$  disjoint sets  $T_1, T_2, \dots, T_n$ , where each of these sets is a tree.
3.  $T_1, T_2, \dots, T_n$  are called the *sub-trees* of the root.

↑ Recursive Definition



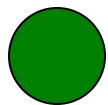
# Node Types



Root (no parent)

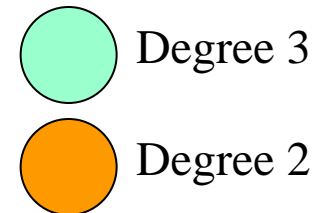
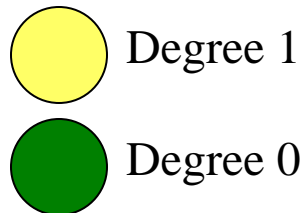
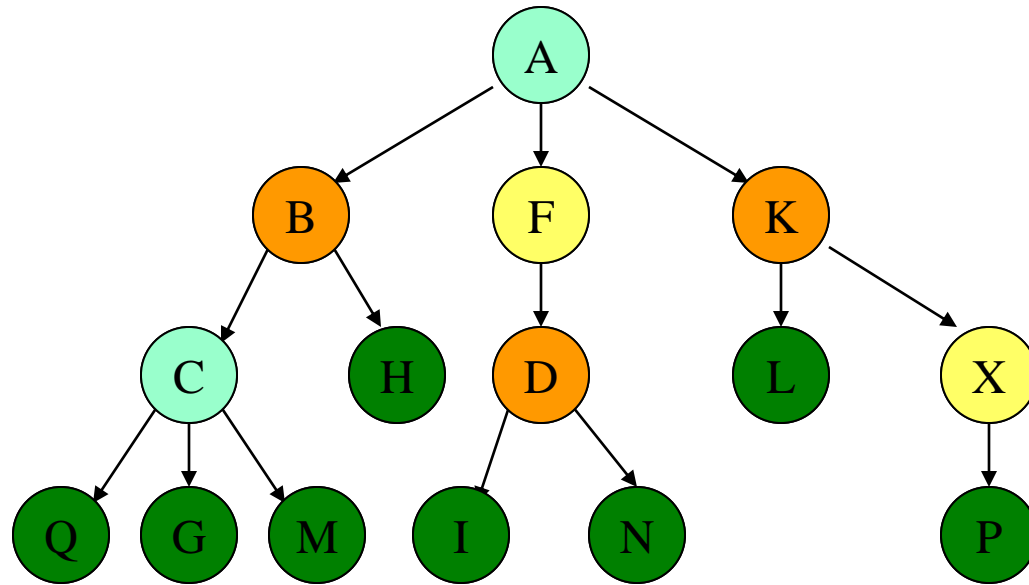


Intermediate nodes (has a parent and at least one child)



Leaf nodes (0 children)

# Degree of a Node: Number of Children





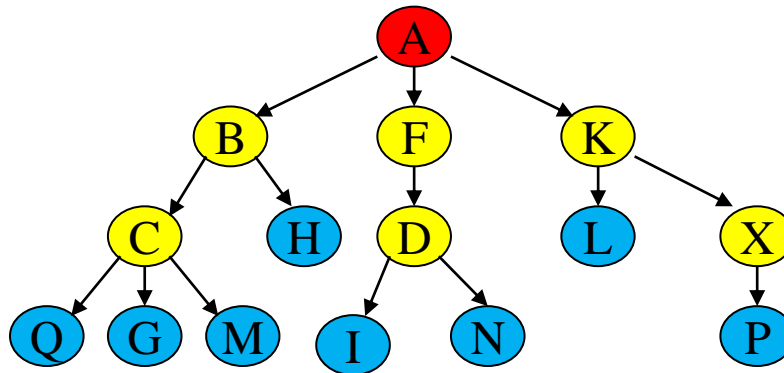
# Tree: Number of Edges

A tree is a collection of  $N$  nodes and  $N - 1$  edges.

There are  $N - 1$  edges follows from the fact that

Each edge connects some node to its parent

Every node except the root has one parent



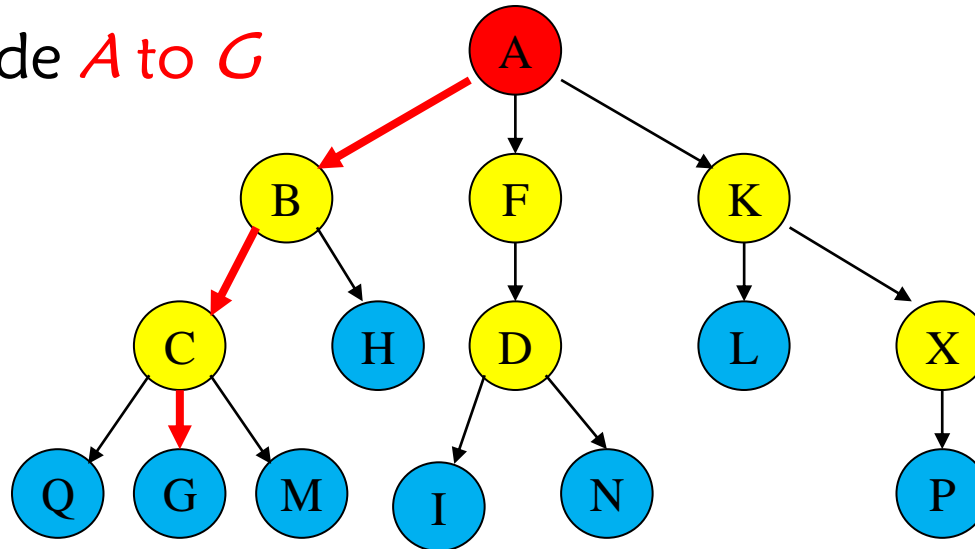
# Tree: Path

A **path** from node  $n_1$  to  $n_k$  is defined as a sequence of nodes  $n_1, n_2, \dots, n_k$  such that  $n_i$  is the parent of  $n_{i+1}$  for  $1 \leq i < k$ .

The **length** of this path is the number of edges on the path, namely,  $k - 1$ .

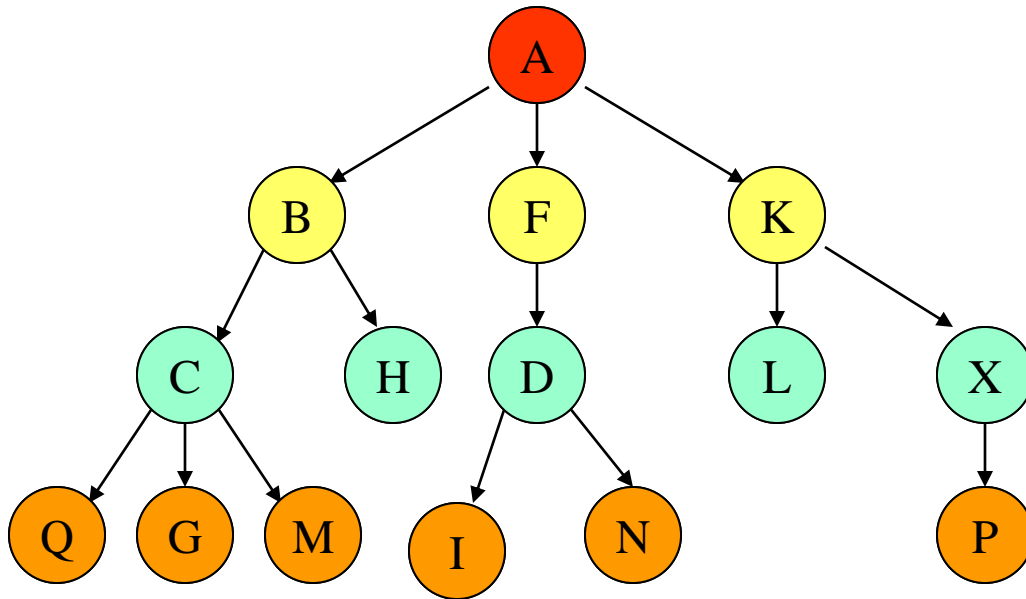
There is a path of length zero from every node to itself.

Path from node  $A$  to  $G$

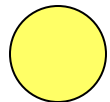


# Height, Level (depth) of a Node

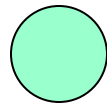
## Distance from the root



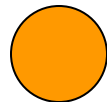
Level 0



Level 1



Level 2



Level 3

For any node  $n_i$ , the **depth** of  $n_i$  is the length of the unique path from the root to  $n_i$ .

**Level of a node is equal to its depth**

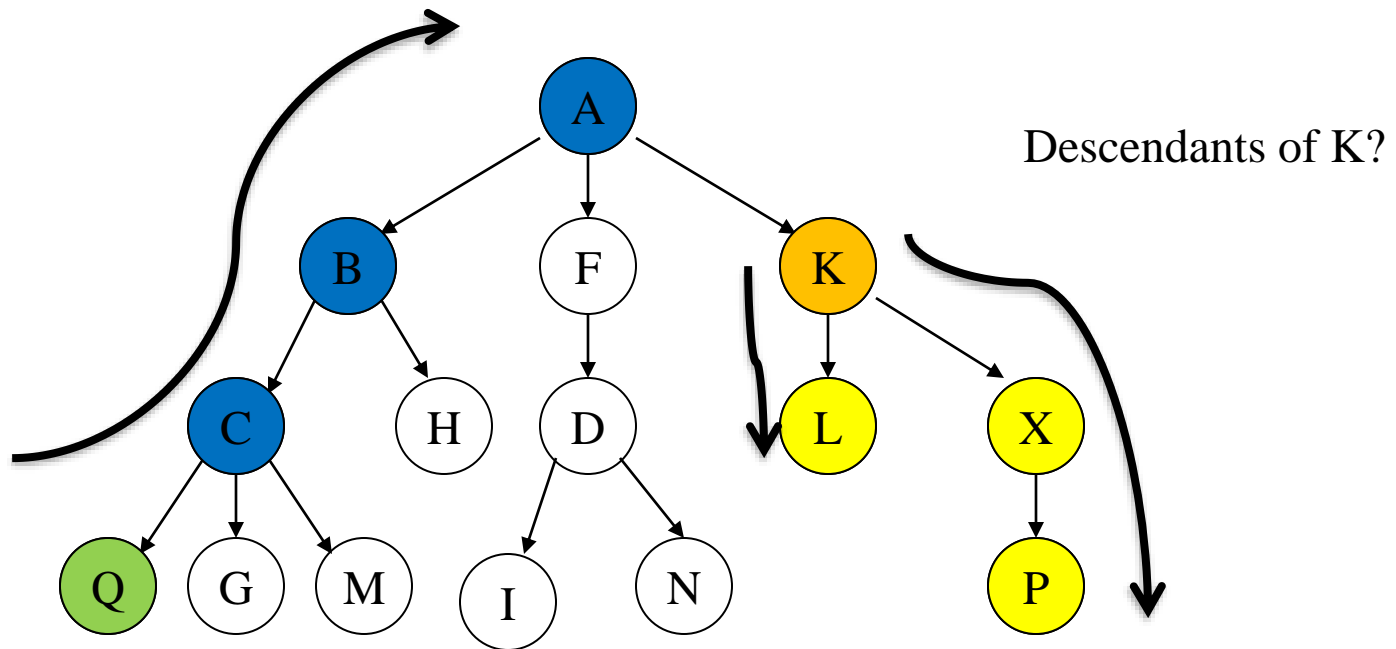
The **height** of  $n_i$  is the length of the longest path from  $n_i$  to a leaf.

All leaves are at height 0.  
height of a tree = height of the root.

Height of the Tree = Maximum Level

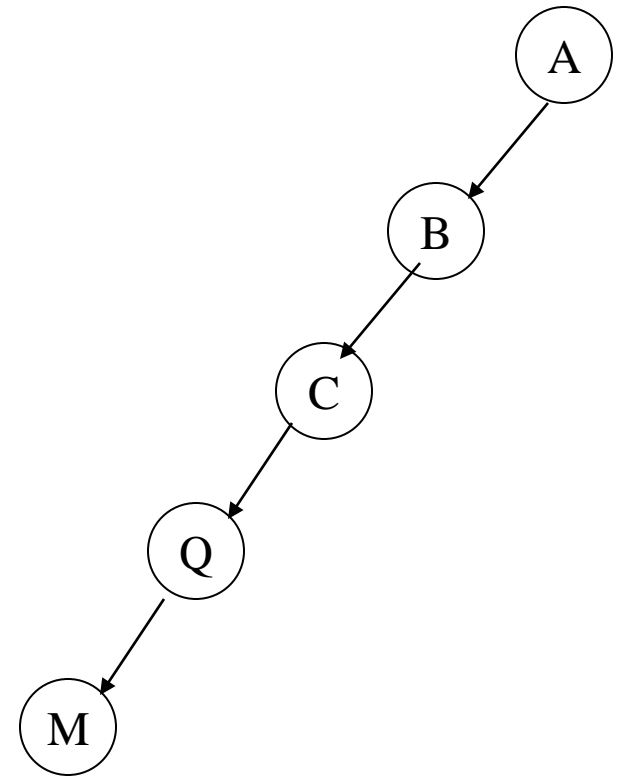
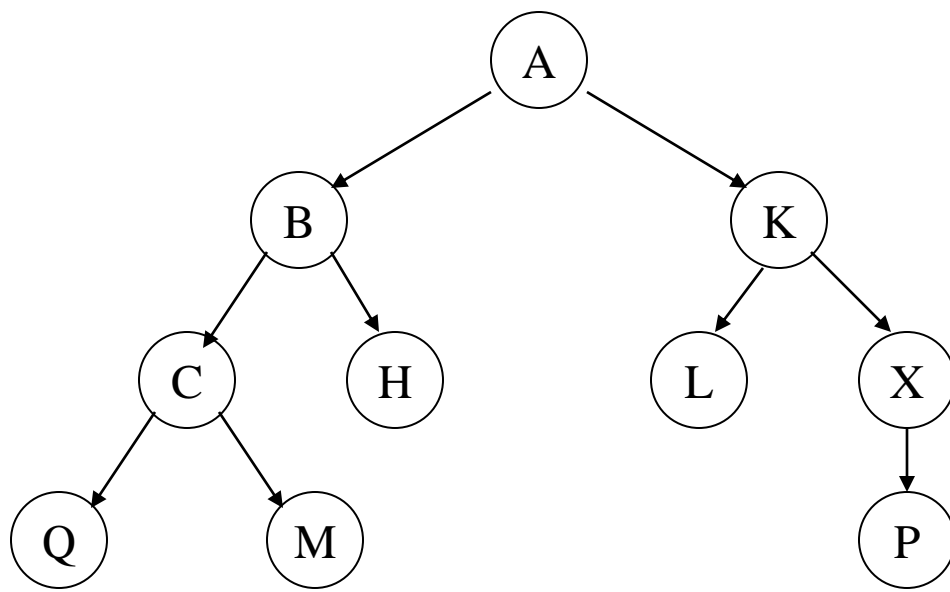
# Ancestors

- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Descendant of a node: child, grandchild, grand-grandchild, etc.



Ancestors of Q?

# Examples of Binary Trees



# Properties of Binary Trees

The maximum number of nodes on level  $i$  of a binary tree is  $2^i$

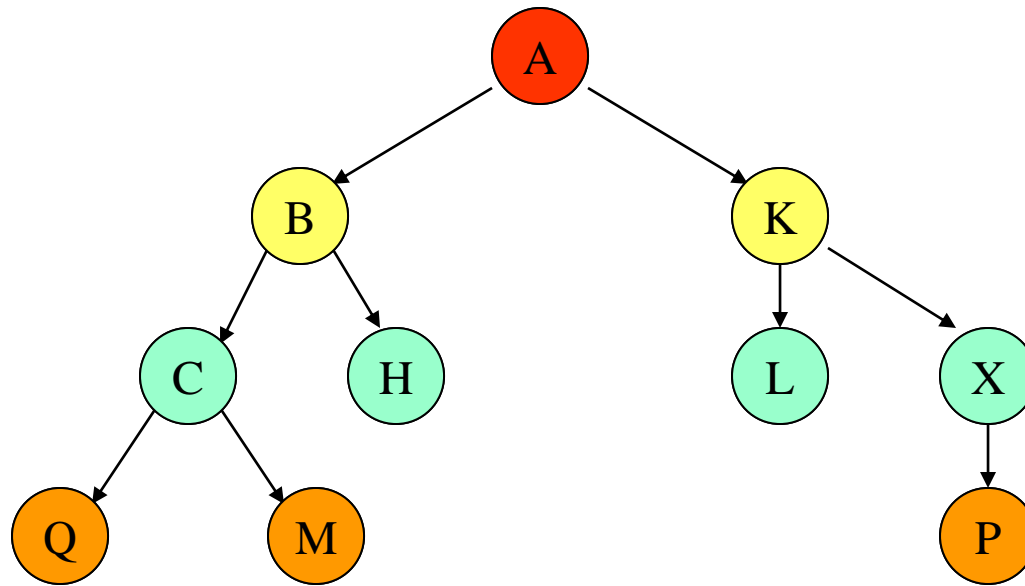
- WHY ?

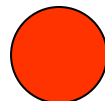
The maximum number of nodes in a binary tree of height  $k$  is  $2^{k+1} - 1$

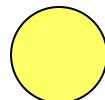
- WHY ?

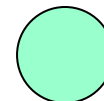
# Maximum number of nodes on a level

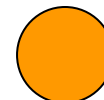
The maximum number of nodes on level  $i$  of a binary tree is  $2^i$



 Level 0       $2^0 = 1$

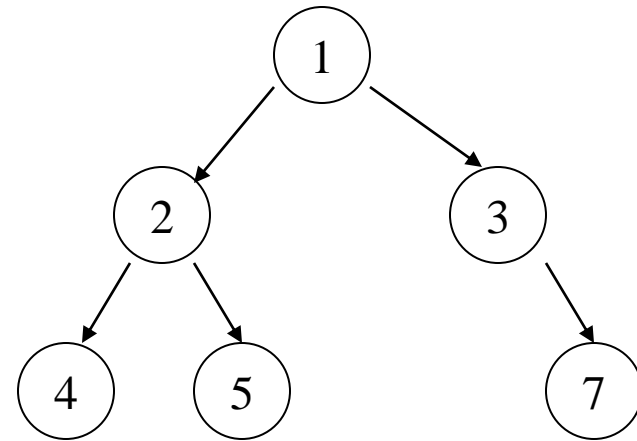
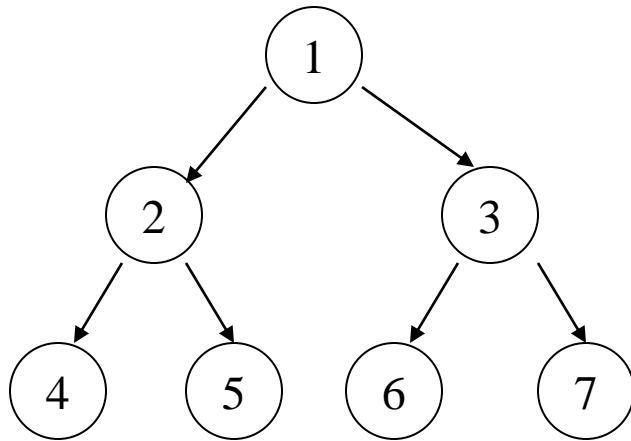
 Level 1       $2^1 = 2$

 Level 2       $2^2 = 4$

 Level 3       $2^3 = 8$

# Maximum no of nodes in a binary tree

The maximum number of nodes in a binary tree of height  $k$  is  $2^{k+1} - 1$





# Maximum no of nodes in a binary tree

- **Max number of nodes = Sum of nodes at each level**
- $2^0 + 2^1 + 2^2 + \dots + 2^k$ , where k is max level

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

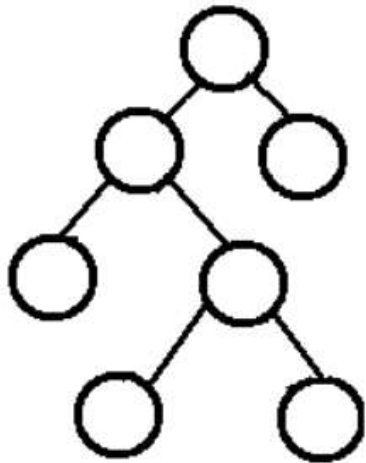
So,

$$\text{max number of nodes} = 2^{k+1} - 1$$

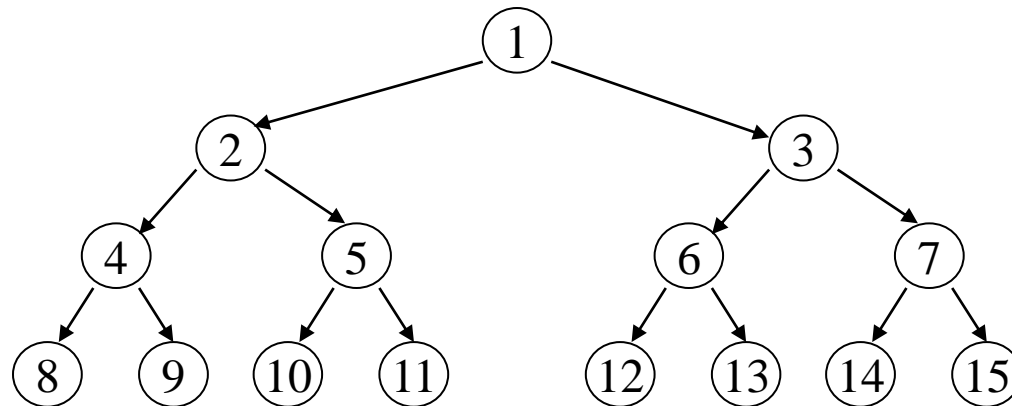
# Full Binary Tree

**Full Binary Tree** A Binary Tree is full if every node has 0 or 2 children.

We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.

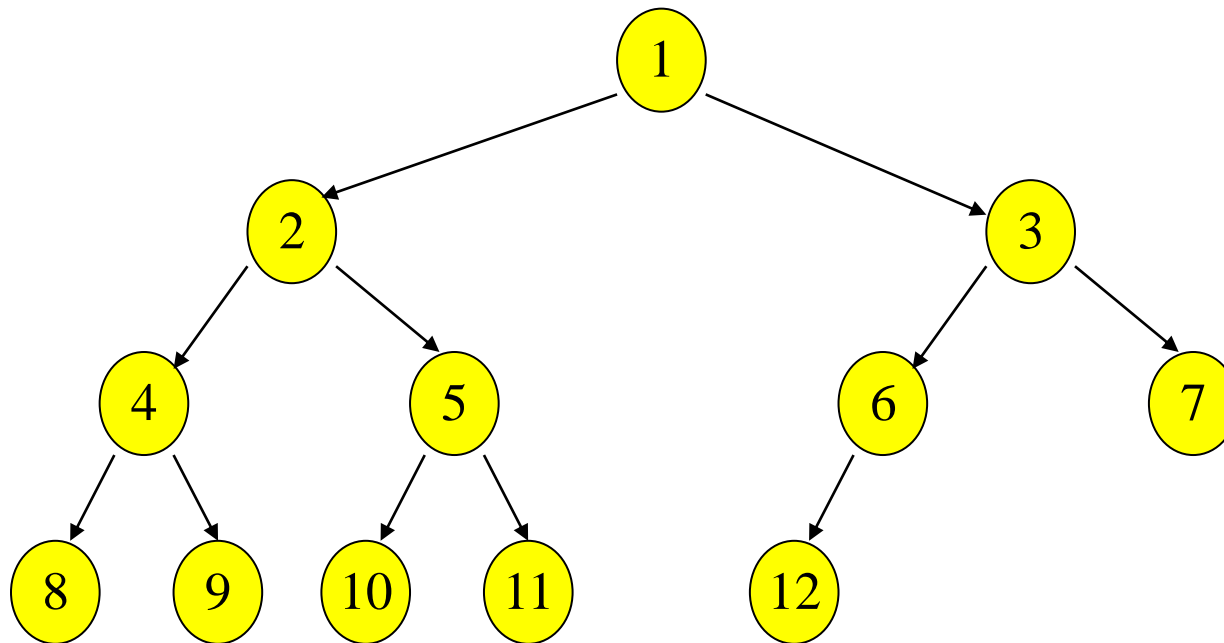


full tree



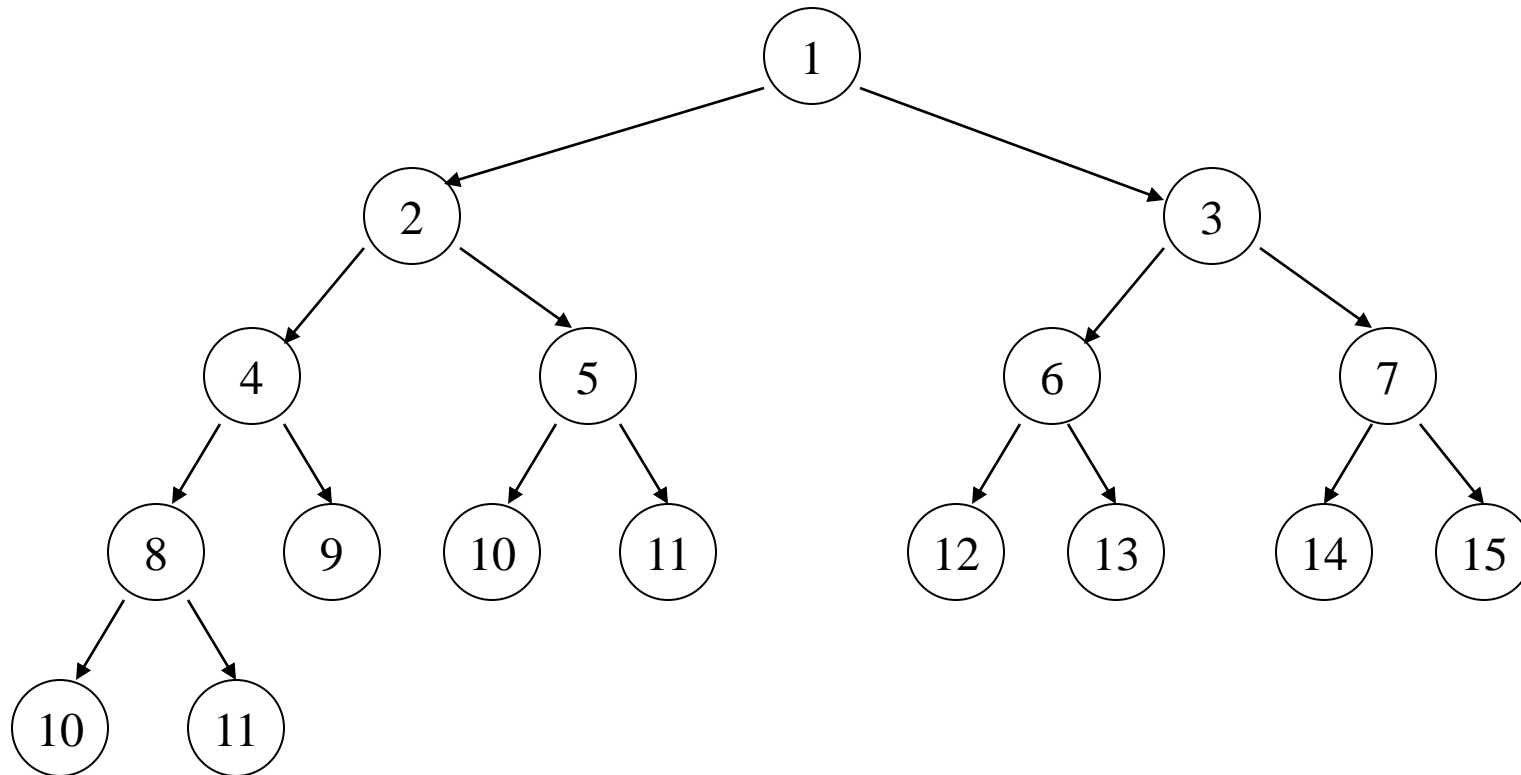
# Complete Binary Tree

A Complete binary tree is a binary tree that is filled, with the possible exception of the bottom level, which is filled from left to right.

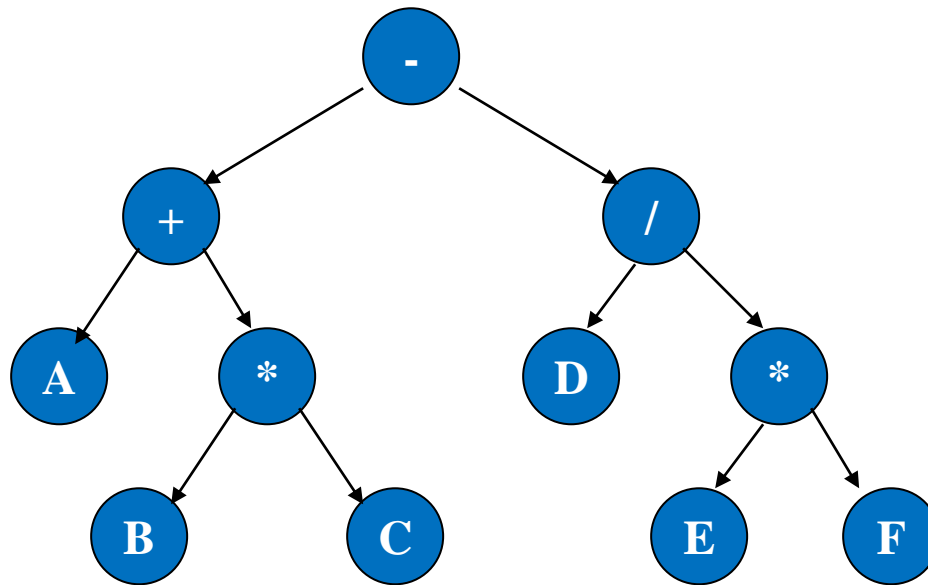


# Complete Binary Tree

**Is it a complete binary tree?**



# EXAMPLE OF BINARY TREE -EXPRESSION TREE

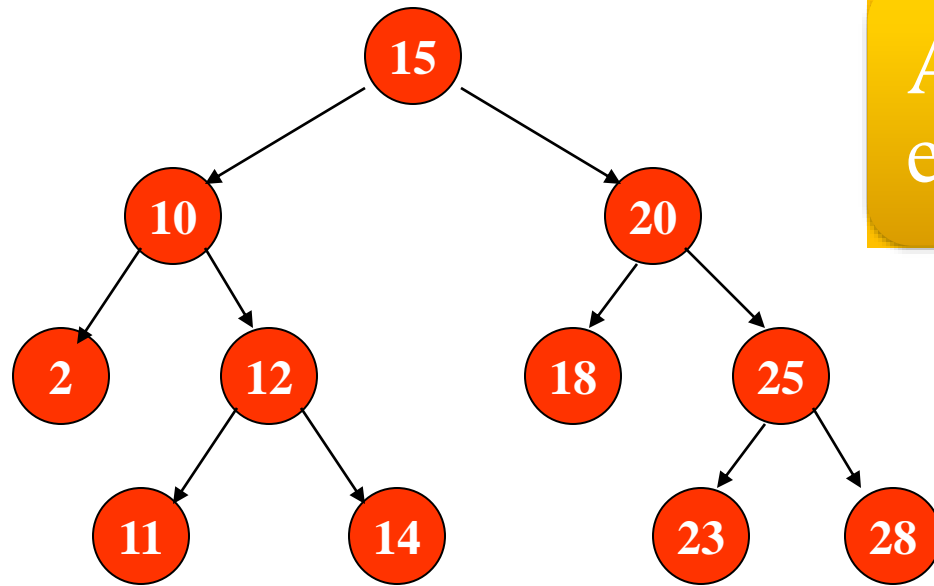


IN-ORDER(LNR):

$A+B*C-D/E*F$

A binary tree used in representing arithmetic expressions is called an **Expression tree**.

The internal nodes are operators, and the leaves are operands. The sub-trees are subexpressions.



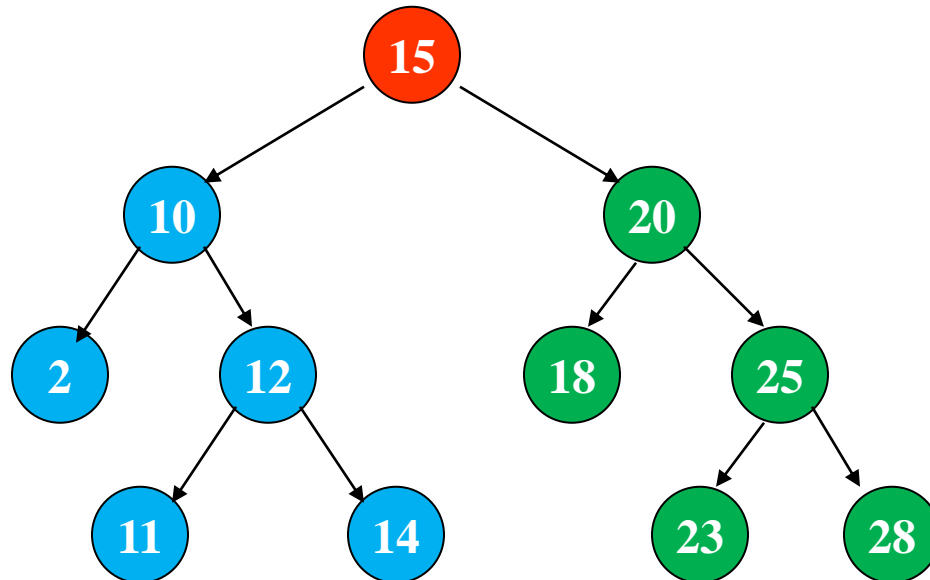
A tree that supports  
efficient search

**BST- Binary Search tree**

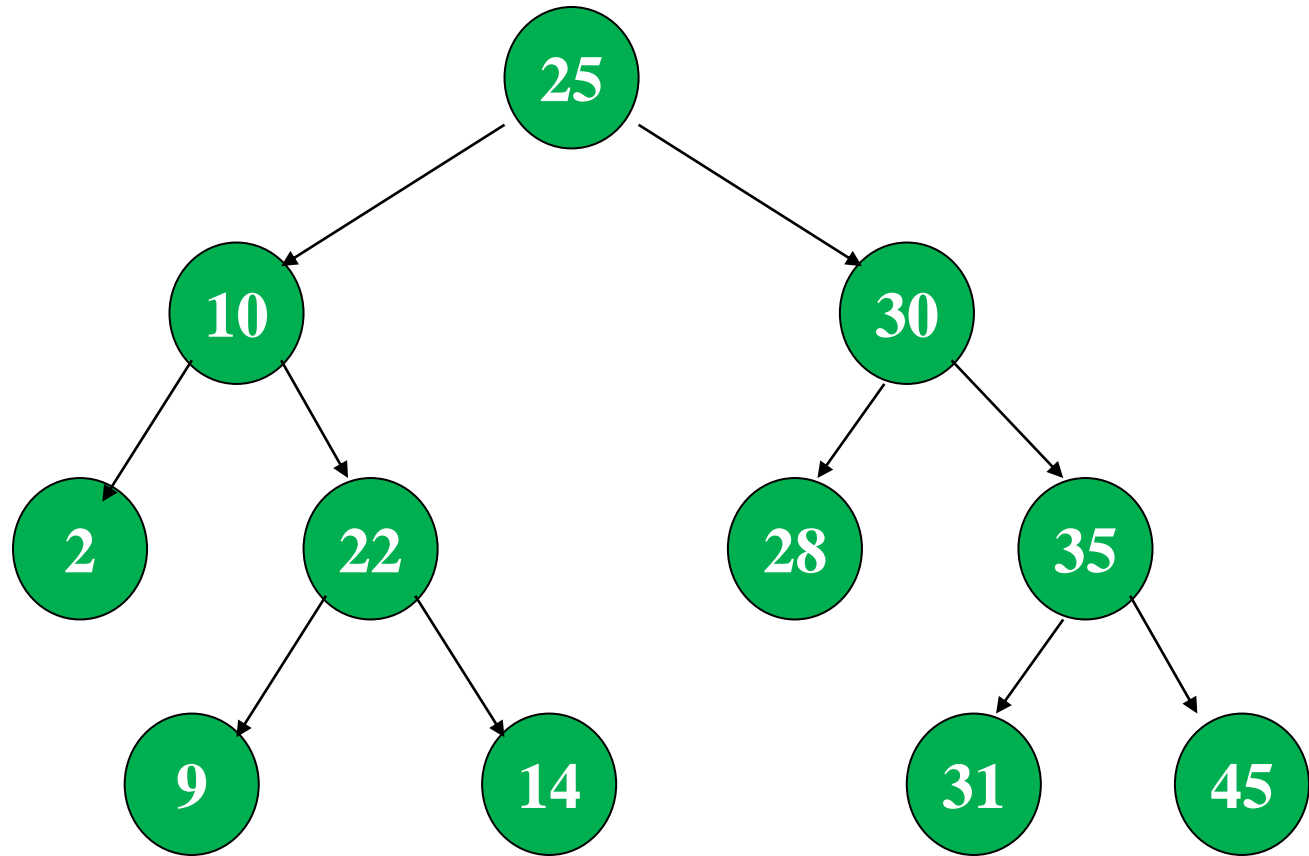
# Binary Search Tree (BST)

A BST is a binary tree with the following properties:

1. Data value in the root node is greater than all the data values stored in the left subtree and is less than or equal to all the values stored in the right subtree.
2. Both the left subtree and right subtree are BSTs.

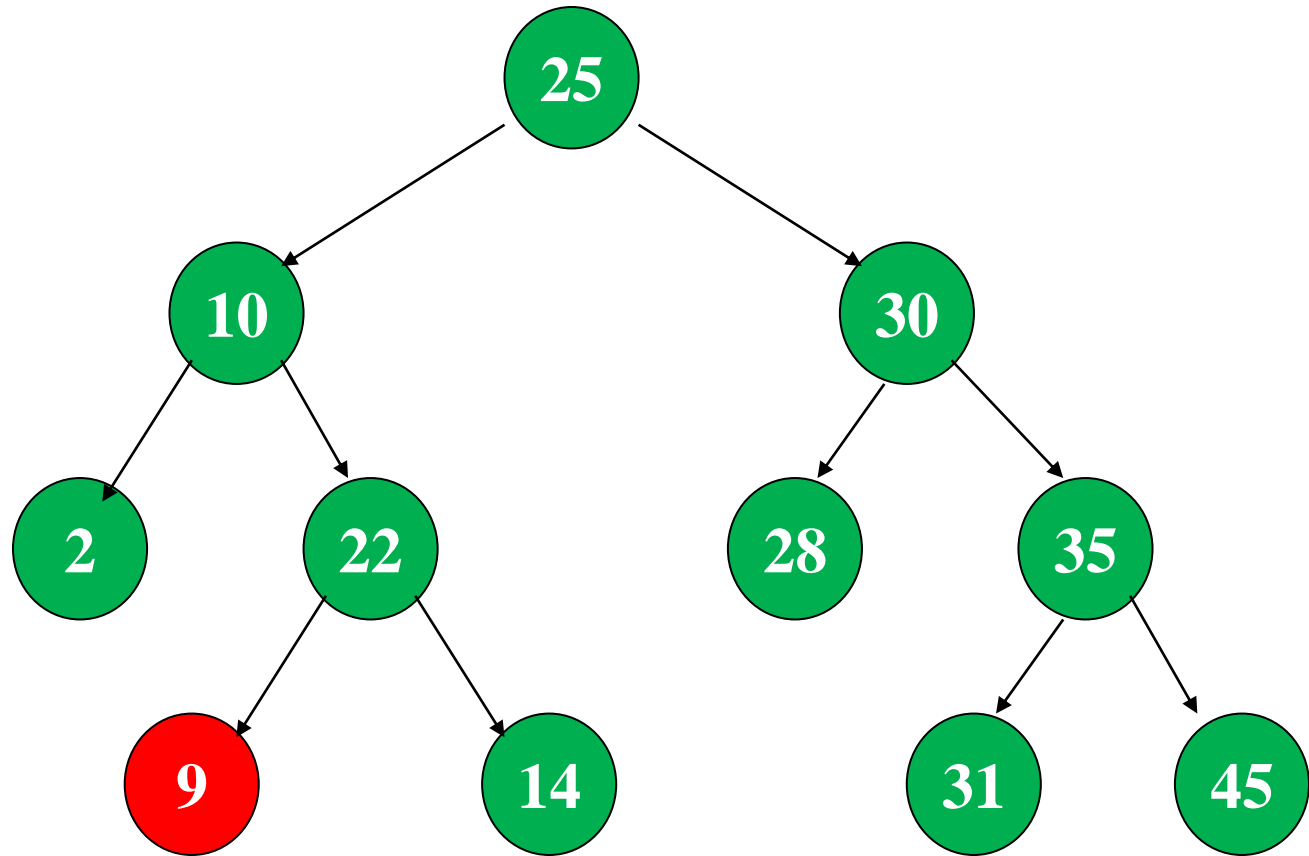


# Binary Search Tree (BST)

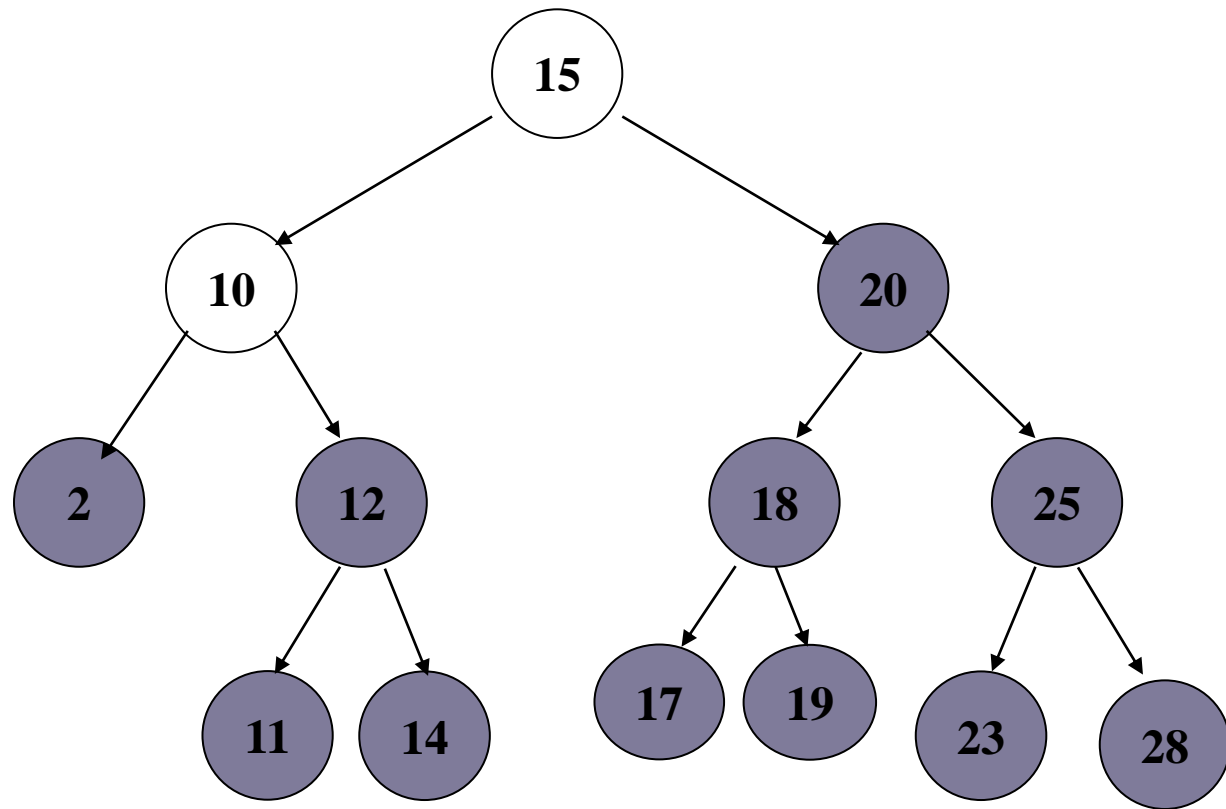




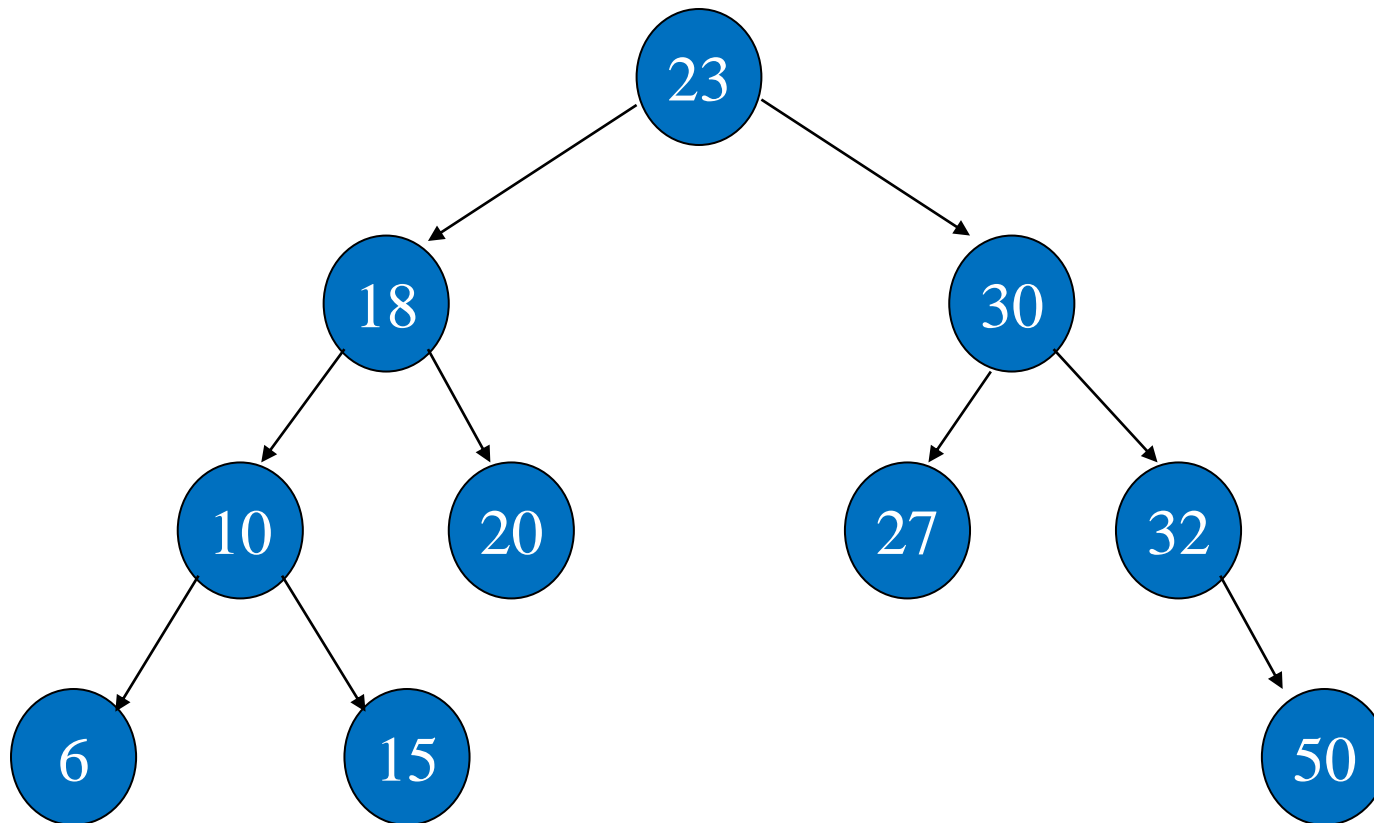
# Binary Search Tree (BST)



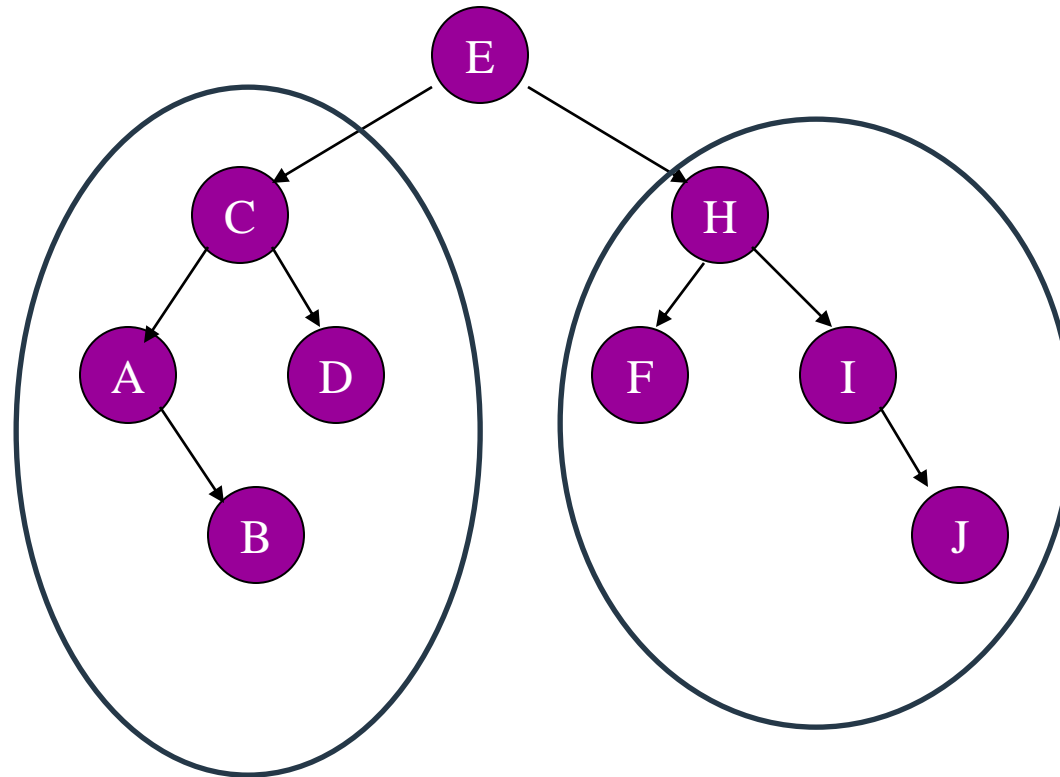
# Binary Search Tree (BST)



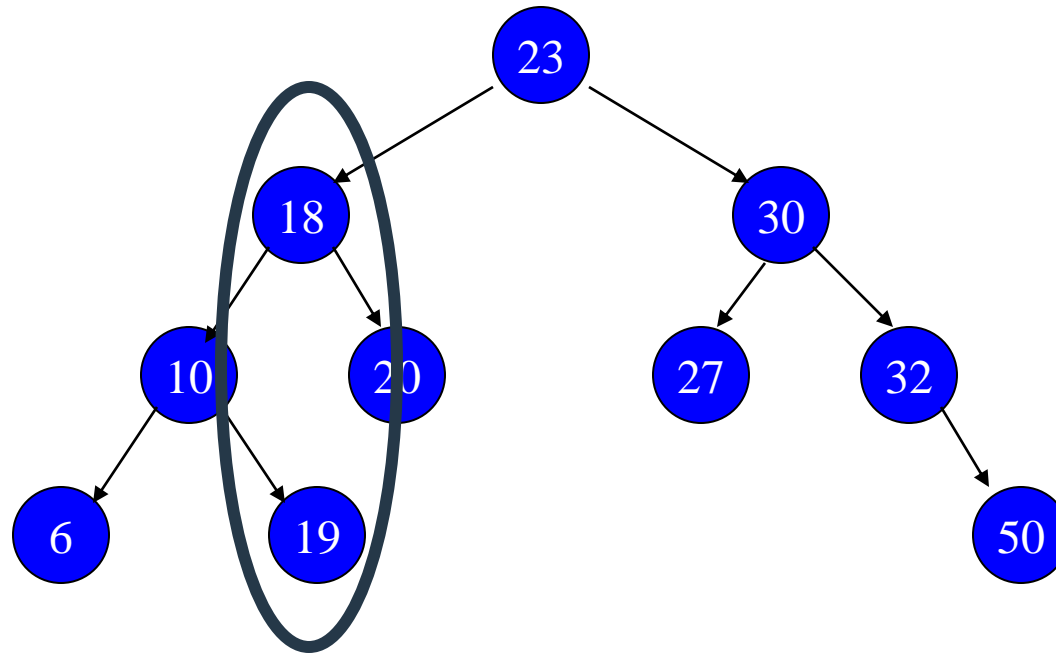
# Binary Search Tree (BST)



# Binary Search Tree (BST)



# Binary Search Tree (BST)

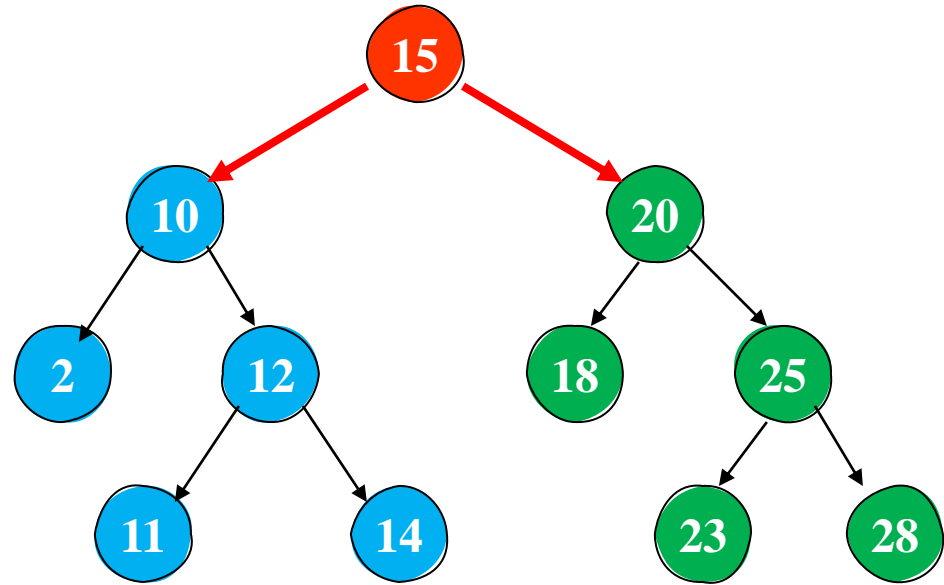


Violating the condition for BST

# Node

```
template <class T>
class Node {
public:
    T data;
    Node<T> * left;
    Node<T> * right;
    Node(T d = 0);
};
```

```
template <class type>
Node <type>::Node(type d) {
    data = d;
    left = NULL;
    right = NULL;
}
```

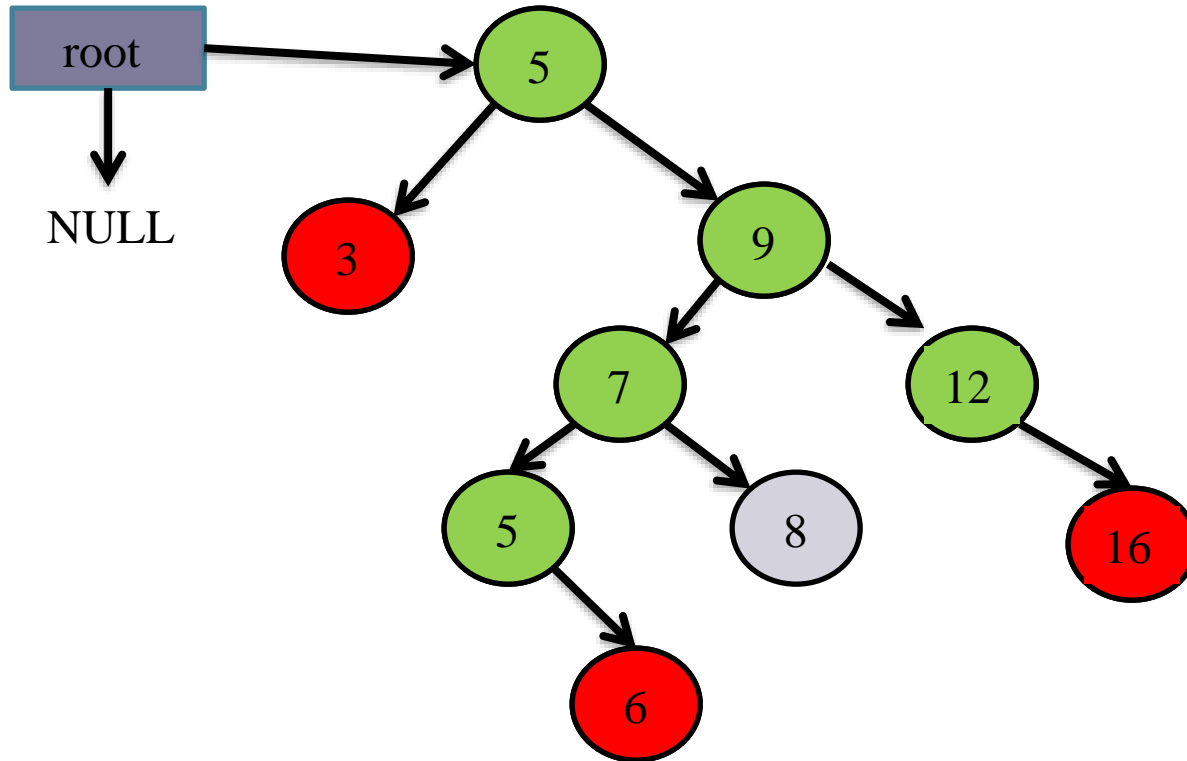


# Tree

```
template <class type>
class tree {
private:
    Node<type> * root;
public:
    tree();
    void insertR(type d, Node <type> *& node);
    void insertI(type d);
    void visit(Node<type> * ptr) { cout << ptr->data << " "; }
    bool searchR(Node<type> * node, type d);
    bool searchI(type);
    void deleteR(type d)
    void deleteI(type d);
    void getPredecessor(Node <type> * node, type & data);
    void Destroy(Node<type> *& node);
    ~tree() { Destroy(root); }
};
```

```
template <class type>
    tree <type>::tree() {
        root = NULL;
    }
```

# Recursive Search



1. Search 3
2. Search 6
3. Search 16

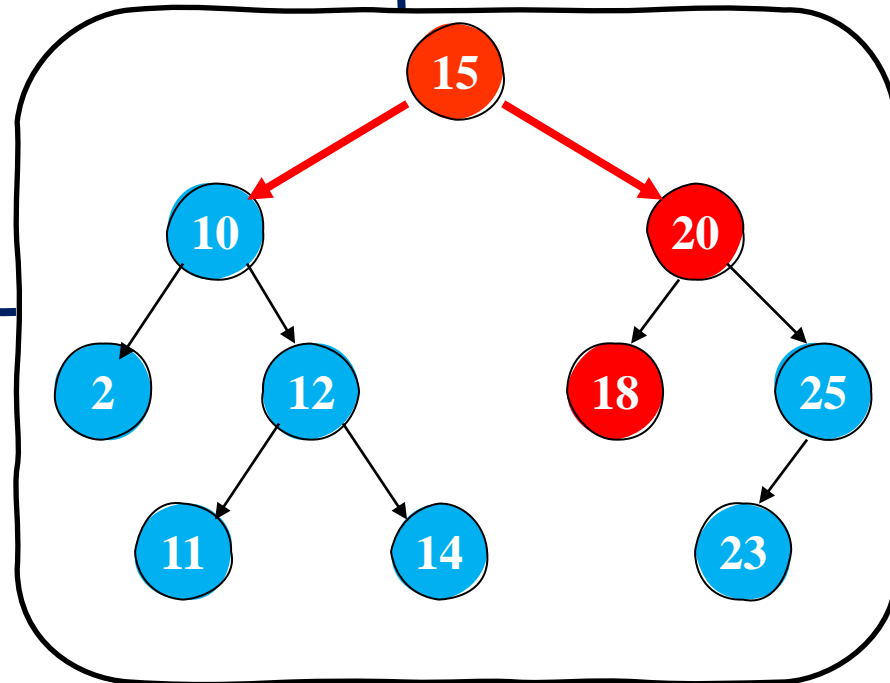
Write recursive version of search function



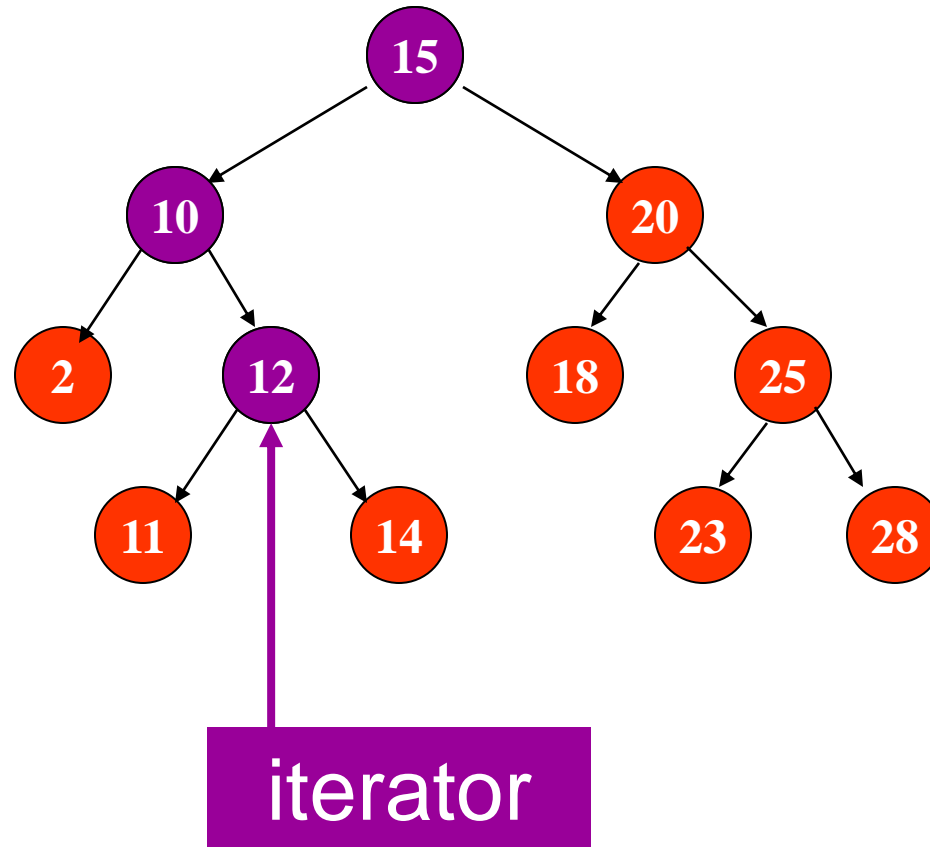
# Recursive search

```
template <class type>
bool tree<type>::searchR(Node<type>*node, type d)
{
    if (node) {
        if (node → data > d)
            searchR(node → left, d);
        else if (node → data < d)
            searchR(node → right, d);
        else
            return true;
    }
    else
        return false;
}
```

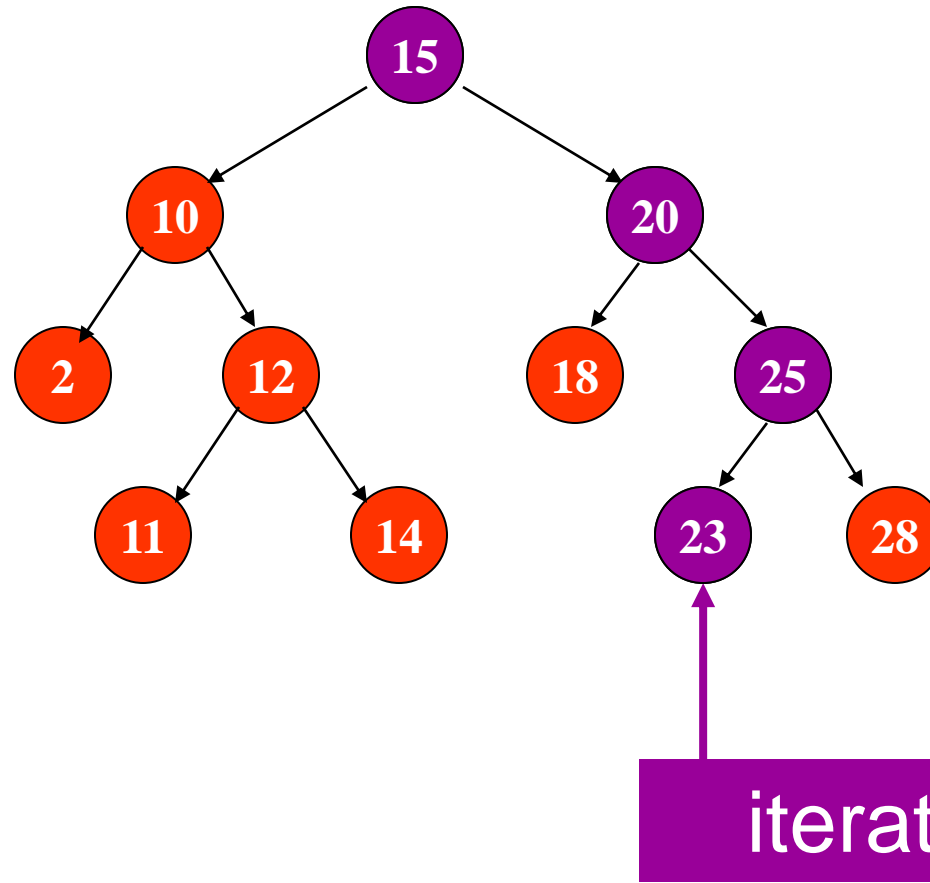
```
bool searchR(type d){
    return searchR(root,d);
}
```



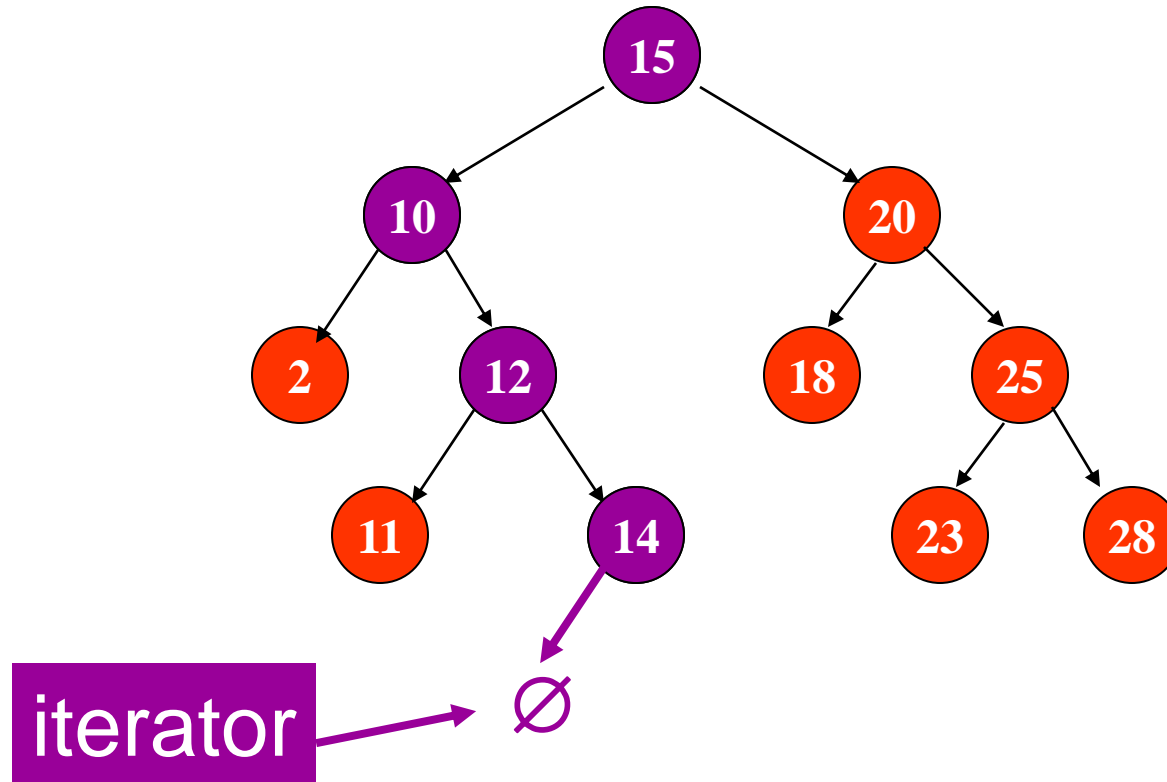
# Iterative search(12)



# search(23)



# search(13)

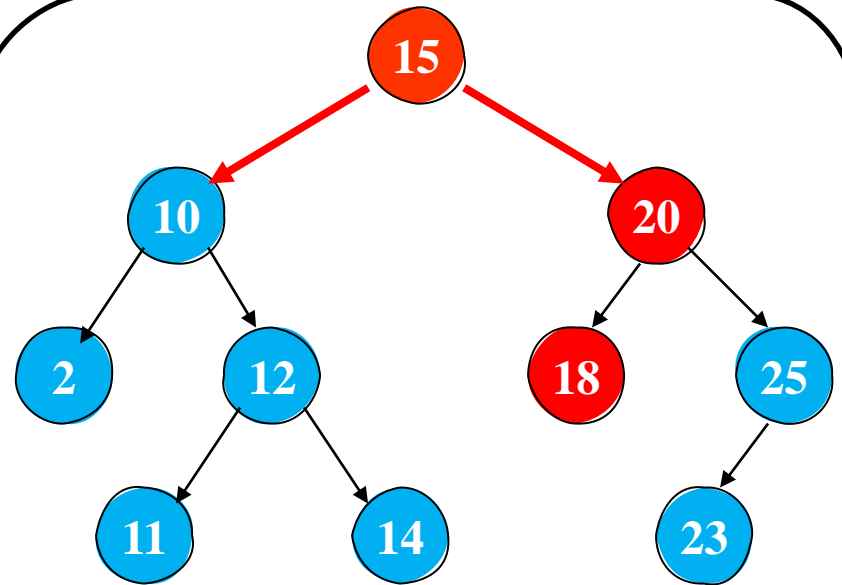


# Iterative Search

```
template <class type>
bool tree<type>::searchIterative(type key) {
    Node <type>* iter = root;
    bool flag = false;

    while (iter && !flag) {

        if (iter->data == key)
            flag = true;
        else if (iter->data > key)
            iter= iter->left;
        else
            iter= iter->right;
    }
    return flag;
}
```



# Search

- **Sorted Array: Binary Search**
  - **Linked List: Linear Search**
  - **Can we Apply the Binary Search on a linked list?**
    - Why not?
  - **Binary Search tree**
- $O(\log N)$
  - $O(N)$
  - $O(\lg n)$ 
    - Average case

# Sorted Array vs Binary Tree

- Array is a Rigid Structure
  - Fixed Size
  - Need to know the size of the largest data set
  - Wastage
- Search: Average case  $O(\log n)$
- Insertion:  $O(n)$
- Deletion:  $O(n)$

The **depth** of an average binary tree is considerably smaller than  $N$ .

An analysis shows that the **average depth is  $O(\sqrt{N})$** , and that for a special type of binary tree, namely the *binary search tree*, the average value of the **depth is  $O(\log N)$**

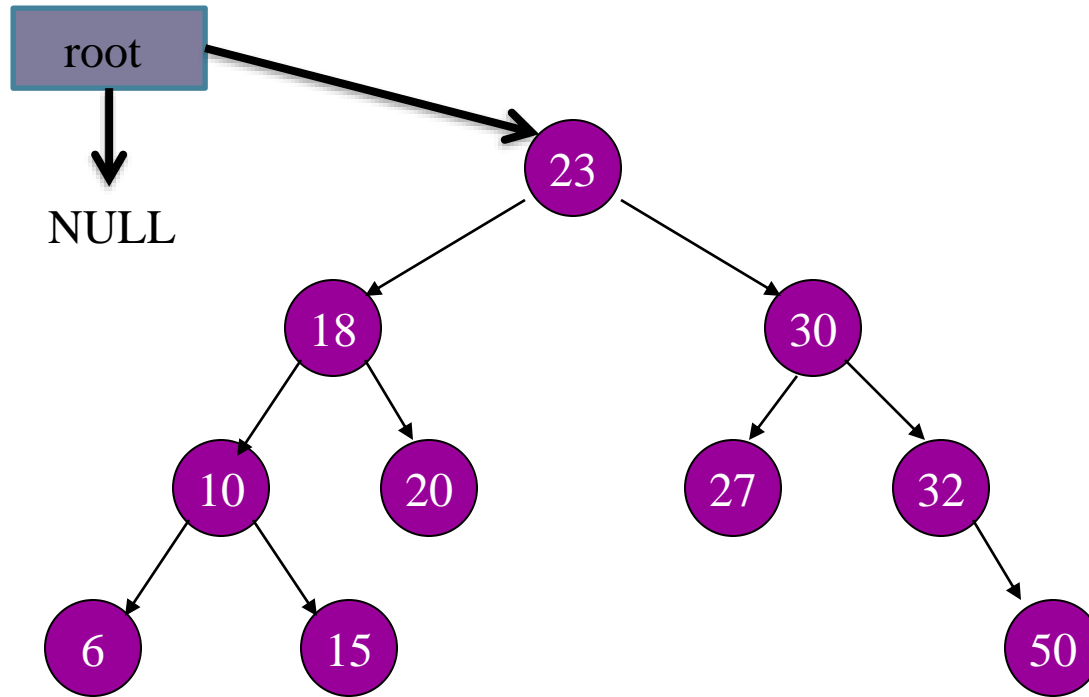
Unfortunately, the depth can be as large as  $N - 1$  (worst case)

# Binary Search Tree

- Binary Tree
- Dynamic Structure (size is flexible)
- Data is stored in a sorted fashion
- A **special class** of BST has the following properties:
  - Search:  $O(\log n)$
  - Insertion:  $O(\log n)$
  - Deletion:  $O(\log n)$



# Insertion



1. Insert 23
2. Insert 18
3. Insert 10
4. Insert 30
5. Insert 20
6. Insert 27
7. Insert 32
8. Insert 50
9. Insert 6
10. Insert 15

# Recursive insertion

```
template <class type>
void tree<type>::insertR(type d, Node <type> *& node){

    if(node==NULL){
        node = new Node<type>(d);
    }
    else if(node->data > d)
        insertR(d,node->left);
    else
        insertR(d,node->right);
}
```

```
void insertR(type d){

    insertR(d,root);

}
```