Session 05 - Tree Structure

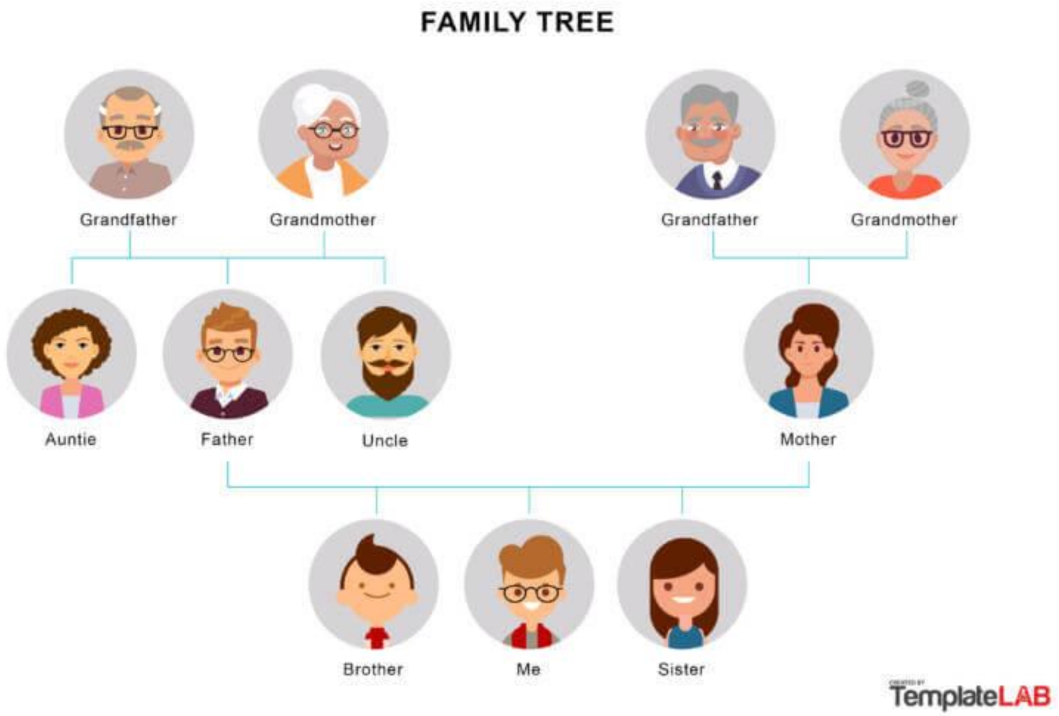
Instructor:

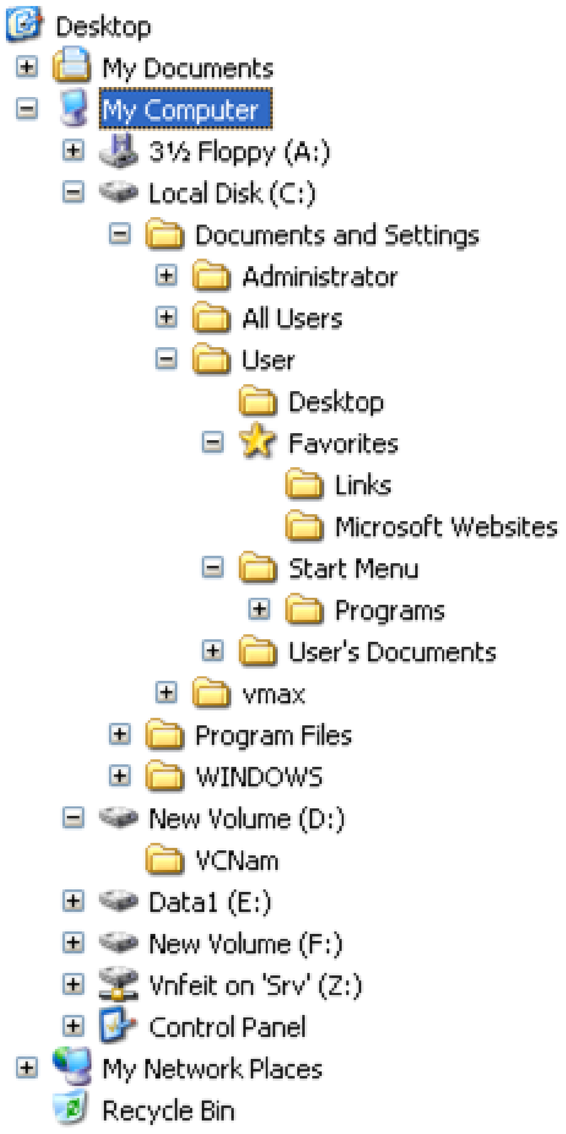
Dr. LE Thanh Tung

Content

Terminologies

Example

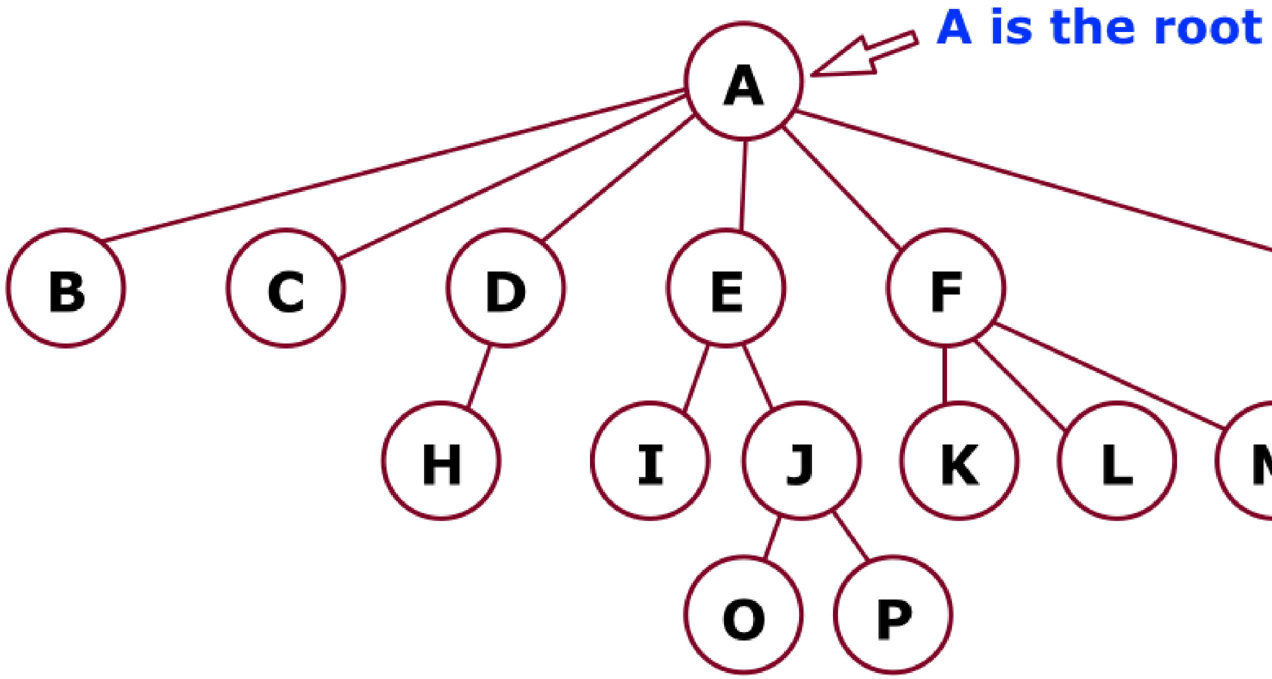

FAMILY TREE
Grandfather
Grandmother
Grandfather
Grandmother
Auntie
Father
Uncle
Mother
Brother
Me
Sister
TemplateLAB

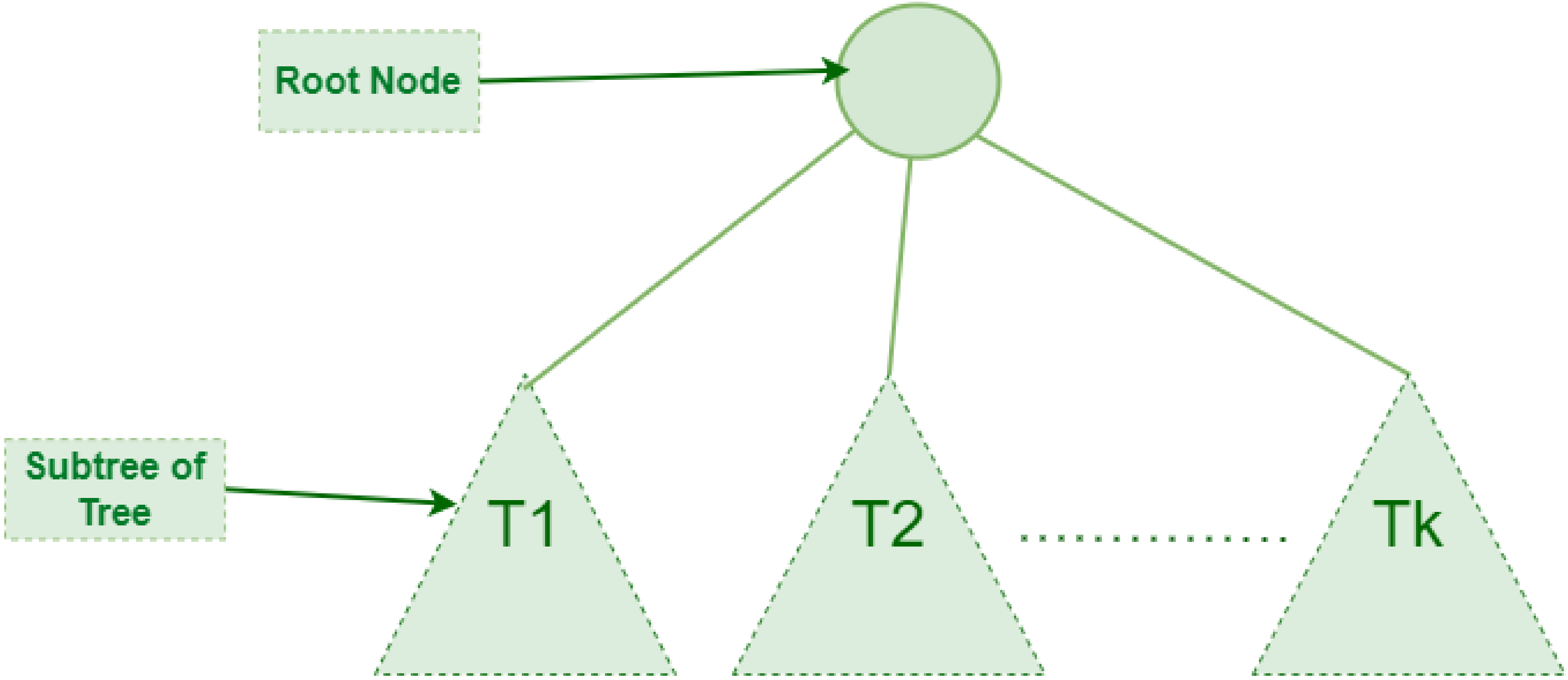

Desktop
My Documents
My Computer
3½ Floppy (A:)
Local Disk (C:)
Documents and Settings
Administrator
All Users
User
Desktop
Favorites
Links
Microsoft Websites
Start Menu
Programs
User's Documents
vmax
Program Files
WINDOWS
New Volume (D:)
VCNam
Data1 (E:)
New Volume (F:)
Vnfeit on 'Srv' (Z:)
Control Panel
My Network Places
Recycle Bin

Tree

- Used to represent **relationships**
- **Hierarchical** in nature
  - "Parent-child" relationship exists between nodes in tree.
  - Generalized to ancestor and descendant
  - Lines between the nodes are called edges
- A **subtree** in a tree is any node in the tree together with all of its descendants

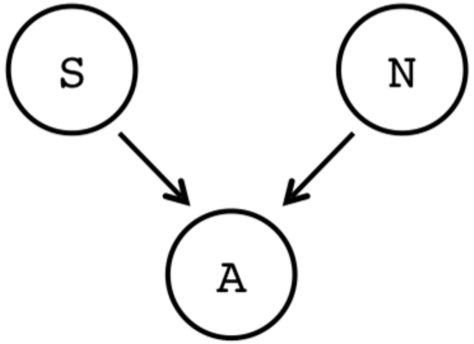
- A tree is
  - a collection of **nodes**, which can be empty.
  - If it is not empty, there is a root node, r, and zero or more non-empty subtrees, T1, T2, …, Tk, whose roots are connected by a directed **edge** from r.

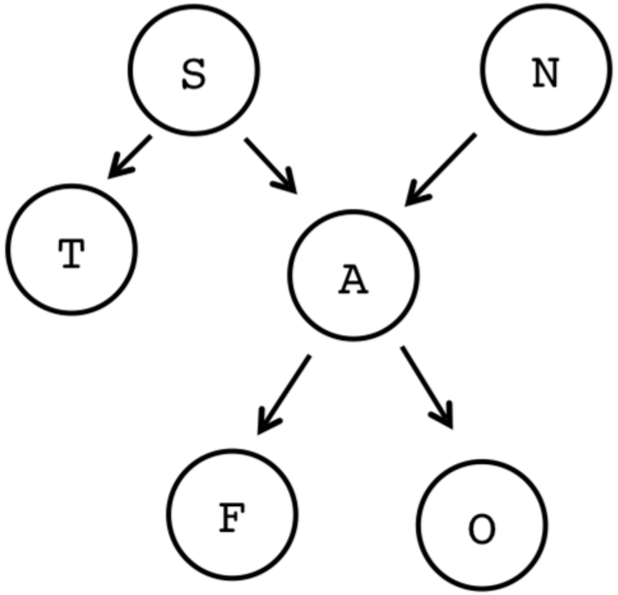



Root Node
Subtree of Tree
T1
T2
Tk

Tree

- Tree nodes can only have **one** parent
- They **cannot** have cycles
- There are **N** nodes and **N-1** edges in a tree
- A tree is a naturally **recursive** data structure

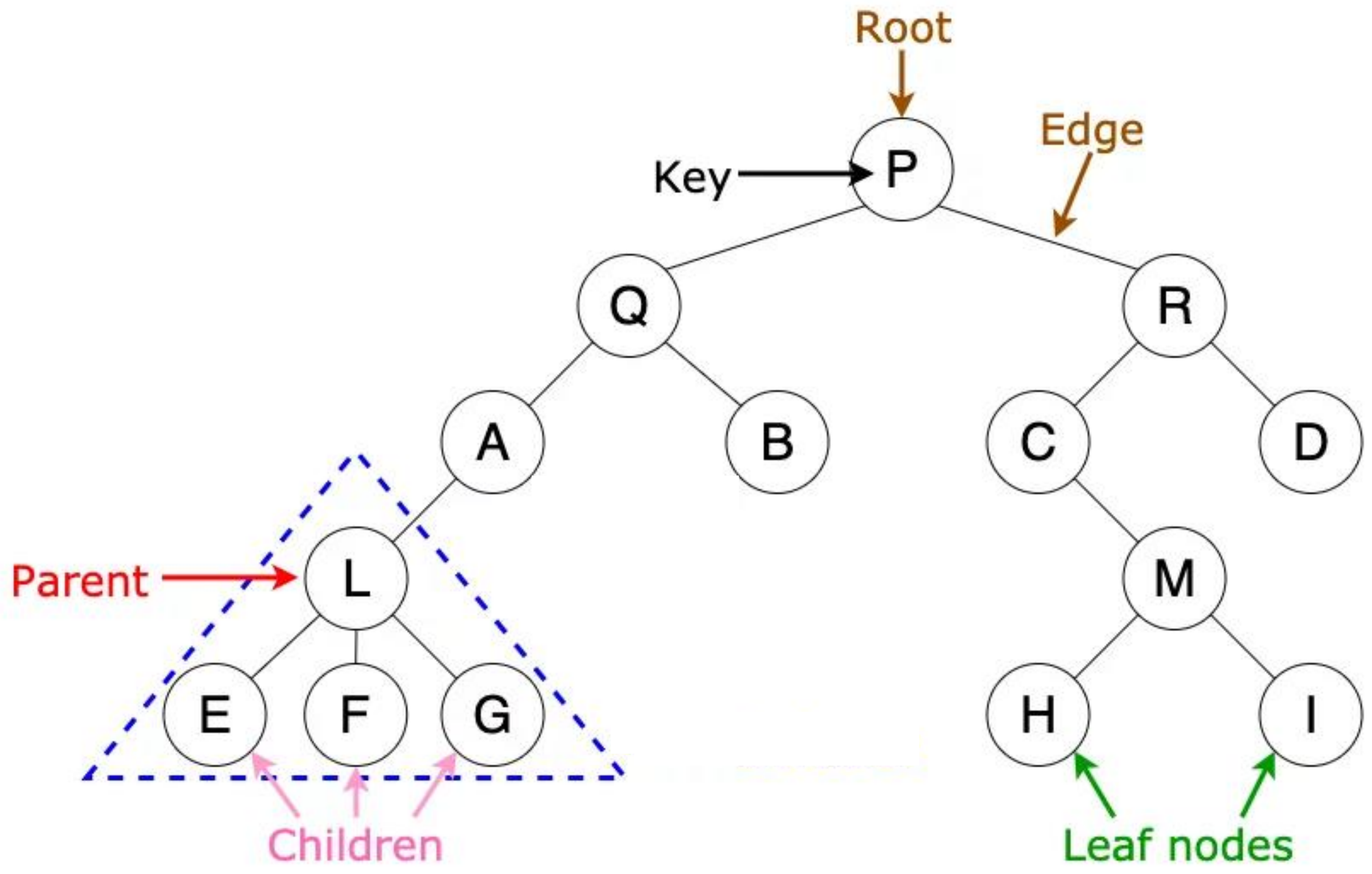




Terminologies

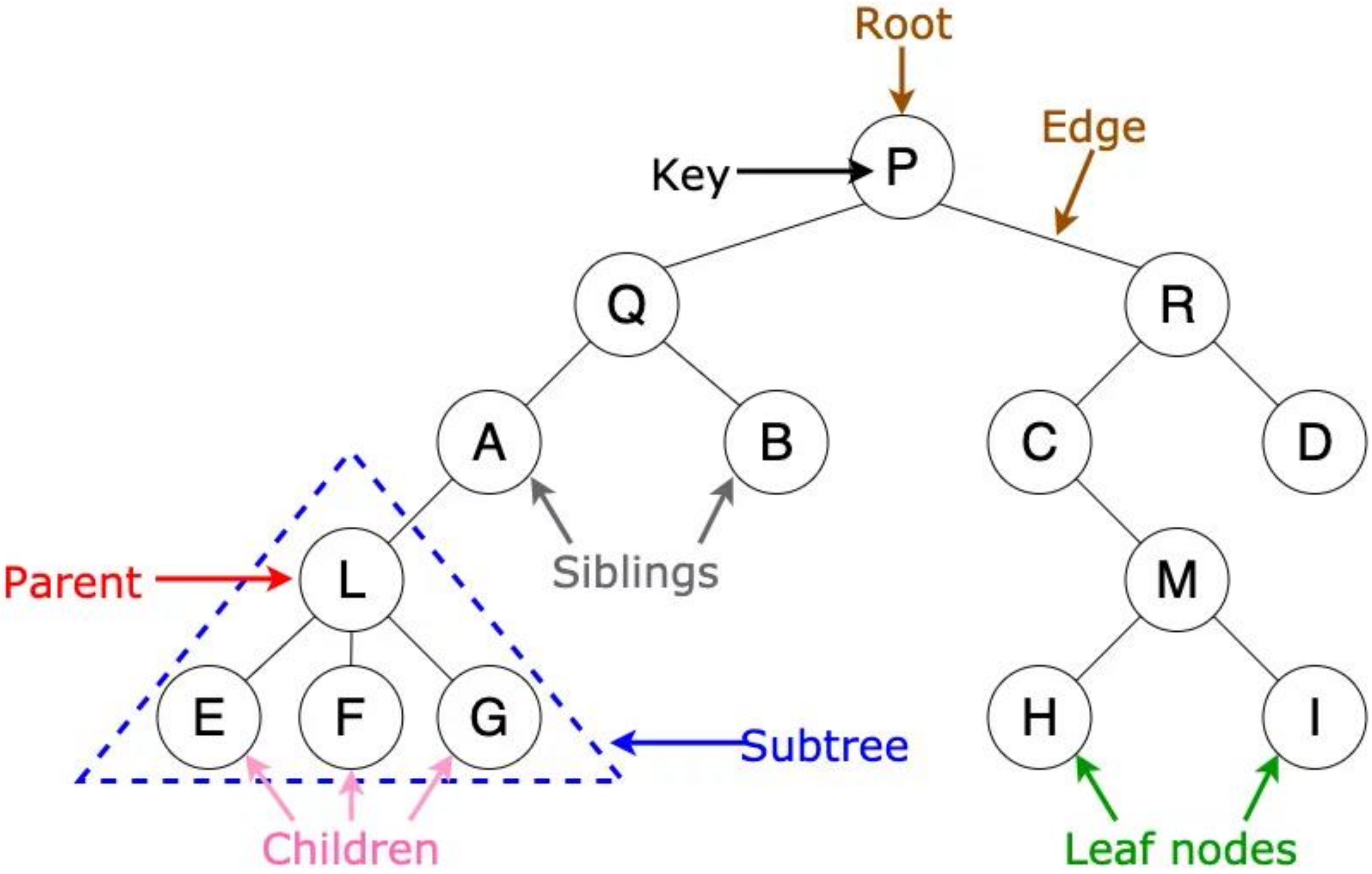
- **node**: an item/element in a tree.
- **parent** (of node n): The node directly above node n in the tree.
- **child** (of node n): The node directly below node n in the tree.
- **root**: The only node in the tree with no parent.
- **Leaf / external node/Terminal**: A node with no children.
- **Internal node**: the node which has at least one child
- **path**: A sequence of nodes and edges connecting a nodes with the nodes below it.

Terminologies



Terminologies

- **siblings**: Nodes with common parent.
- **ancestor** (of node n): a node on the path from the root to n.
- **descendant** (of node n): a node on the path from node n to a leaf.
- **subtree** (of node n): A tree that consists of a child (if any) of n and the child's descendants.


Root
Edge
Key
P
Q
R
A
B
C
D
Siblings
Parent
L
M
E
F
G
H
I
Subtree
Children
Leaf nodes

Terminologies

- **Degree/order**
  - Order of node n: number of children of node n.
  - Order of a tree: the maximum order of nodes in that tree.
- **Level** (of node n)
  - If `n` is the root of `T`, it is at level 1.
  - If `n` is not the root of `T`, its level is 1 greater than the level of its parent.
    - ```
      if node n is root:
          level(n) = 1
      ```
    - ```
      Otherwise:
          level(n) = 1 + level(parent(n))
      ```

- **Depth** (of node n)
  - is the number of edges from the root to the node
  - If `n` is the root of `T` , it has depth of 1.
  - If `n` is not the root of `T` , its depth is 1 greater than the depth of its parent.
    - `if node n is root:`

      `depth(n) = 1`
    - `Otherwise:`

      `depth(n) = 1 + depth(parent(n))`

- **Height of tree**: number of nodes in the longest path from the root to a leaf.

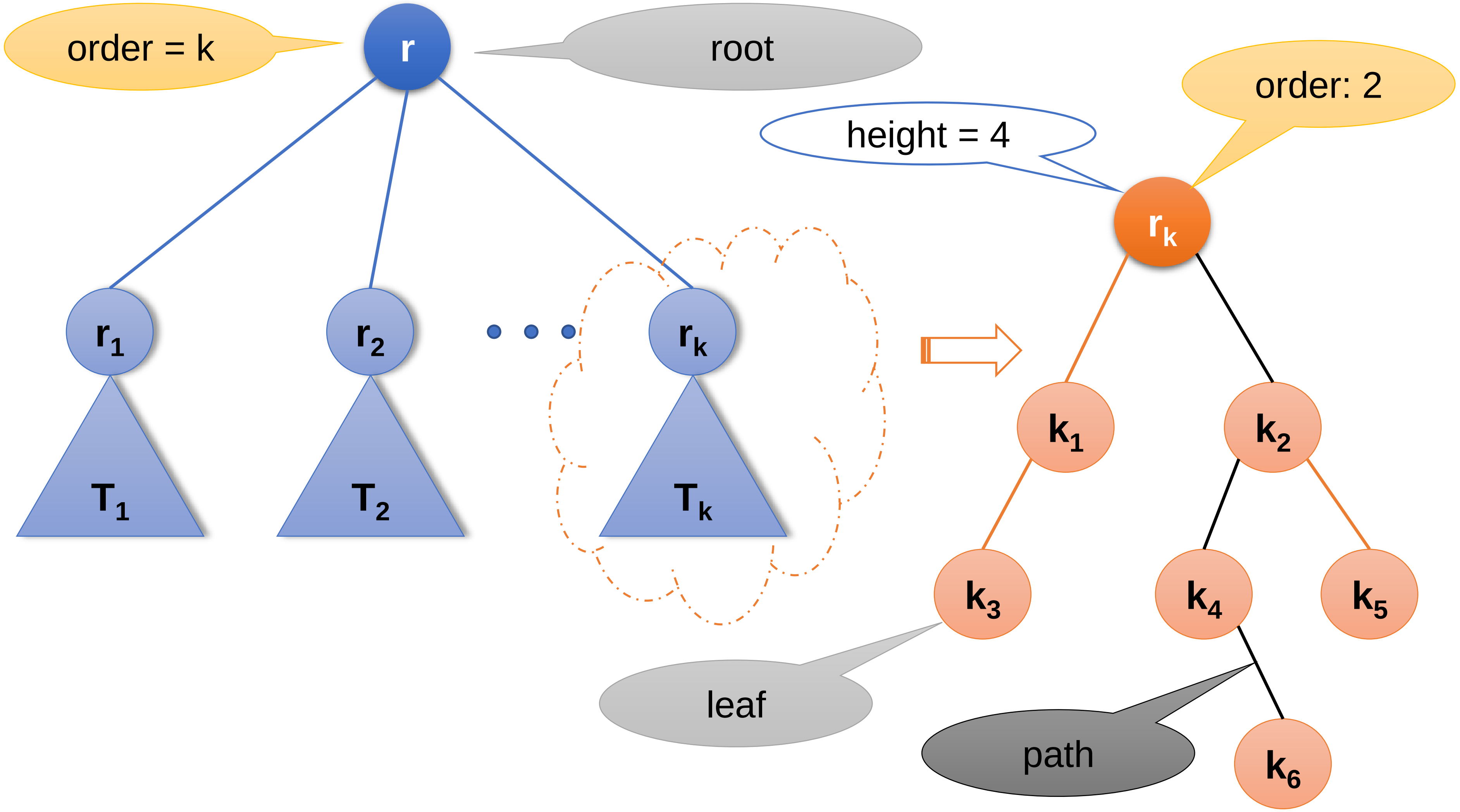
- Height of a tree T in terms of the levels of its nodes
  - If T is empty, its height is 0.
  - If T is not empty, its height is equal to the maximum level of its nodes.

Terminologies

- Height of tree T:

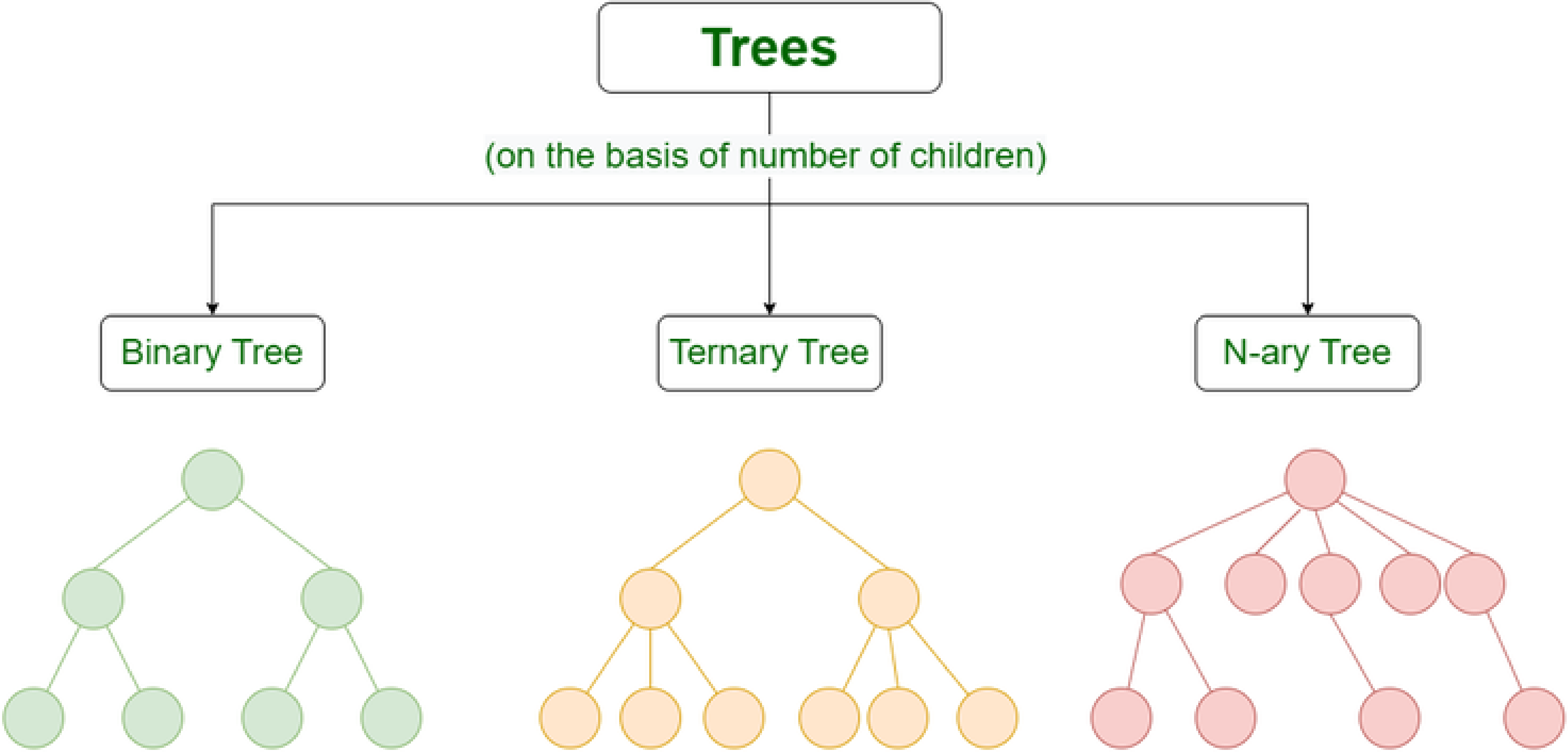
```
if T is empty:
    height(T) = 0
Otherwise:
    height (T) = max{level(Ni)}, Ni∈T
               = 1 + max{height(Ti)},
                         Ti is a subtree of T
```

Terminologies



Kinds of Tree

Kinds of Tree



General Tree

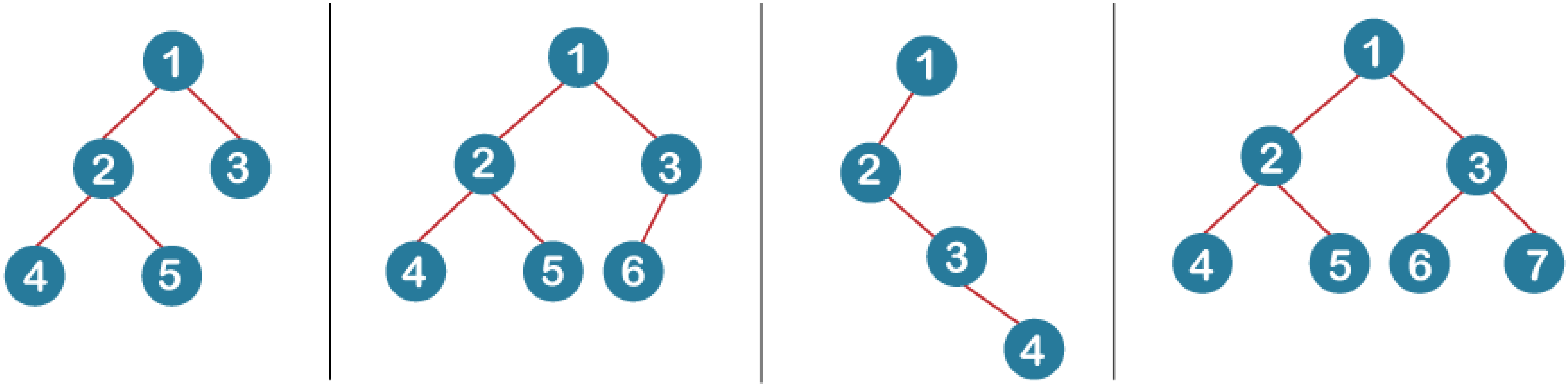
- Set T of one or more nodes such that T is partitioned into disjoint subsets
  - A single node r , the root
  - Sets that are general trees, called subtrees of r

N-ary Tree

- set T of nodes that is either empty or partitioned into disjoint subsets:
  - A single node r , the root
  - n possibly empty sets that are n-ary subtrees of r

Binary Tree

- Set T of nodes that is either empty or partitioned into disjoint subsets
  - Single node r , the root
  - Two possibly empty sets that are binary trees, called left and right subtrees of r



Traversals

Traversal

- Visit each node in a tree exactly once.
- Many operations need using tree traversals.
- The basic tree traversals:
  - Pre-order
  - In-order
  - Post-order

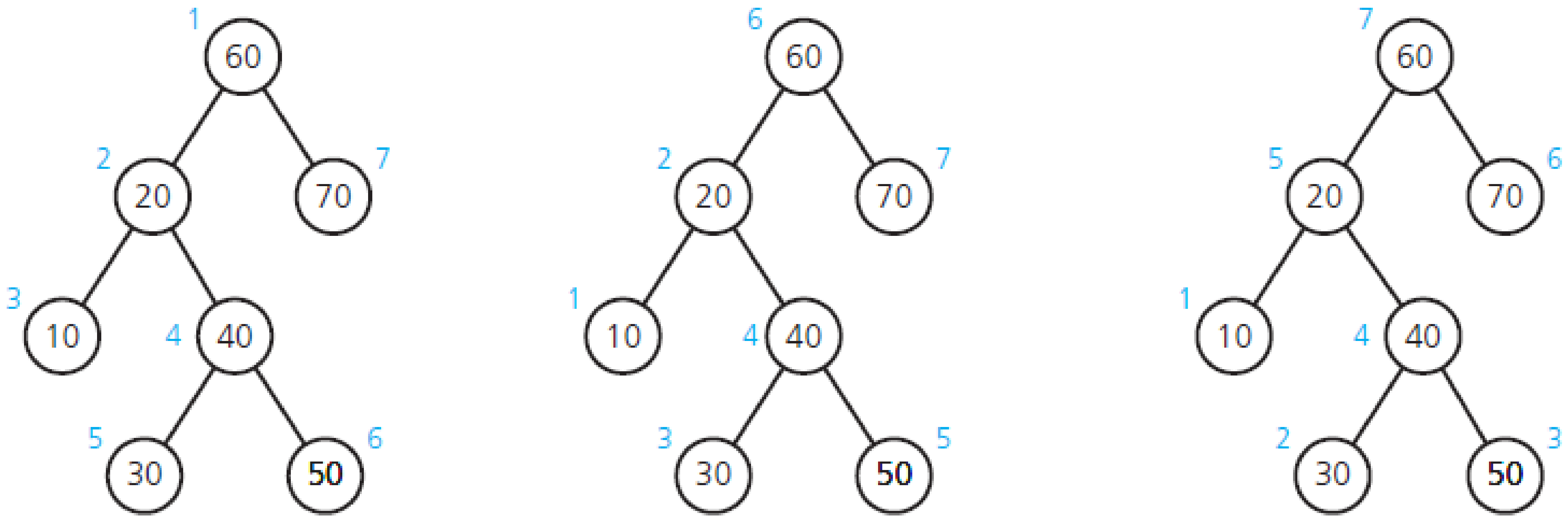
```
PreOrder(root)
{
    if root is empty
        Do_nothing;
    Visit root; //Print, Add, ...
    //Traverse every Child_i.
    PreOrder(Child_0);
    PreOrder(Child_1);
    …
    PreOrder(Child_k-1);
}
```

Post-order Traversal

```
PostOrder(root)
{
    if root is empty
        Do_nothing;
    //Traverse every Childi
    PostOrder(Child0);
    PostOrder(Child1);
    …
    PostOrder(Childk-1);
    Visit at root; //Print, Add, ...
}
```

```
InOrder(root)
{
    if root is empty
        Do_nothing;
    //Traverse the child at the first position
    InOrder(Child_0);
    Visit at root;
    //Traverse other children
    InOrder(Child_1);
    InOrder(Child_2);
    …
    InOrder(Child_{k-1});
}
```

Traversal

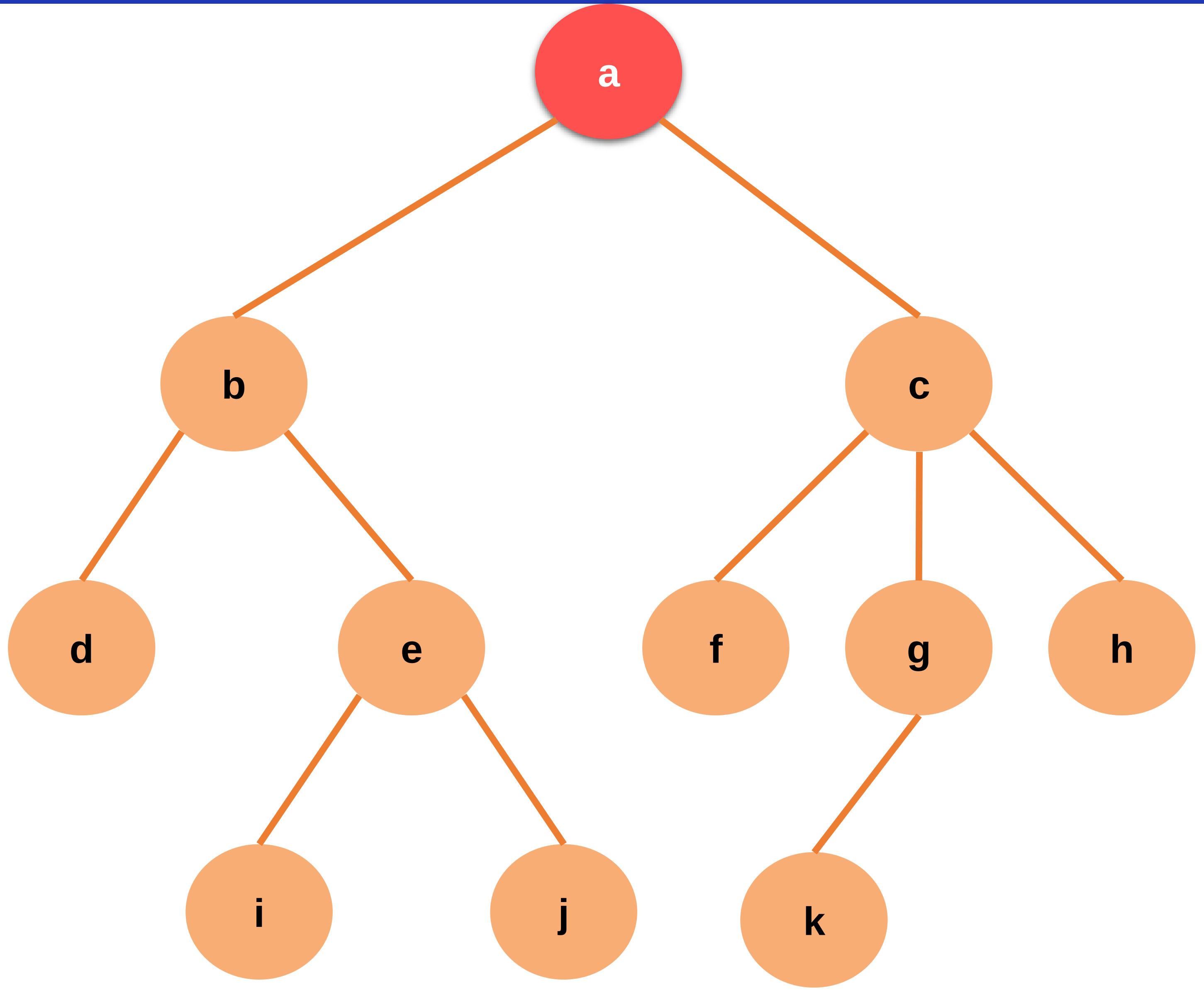


(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

Examples

- **Pre-order ?**
- **In-order ?**
- **Post-order ?**



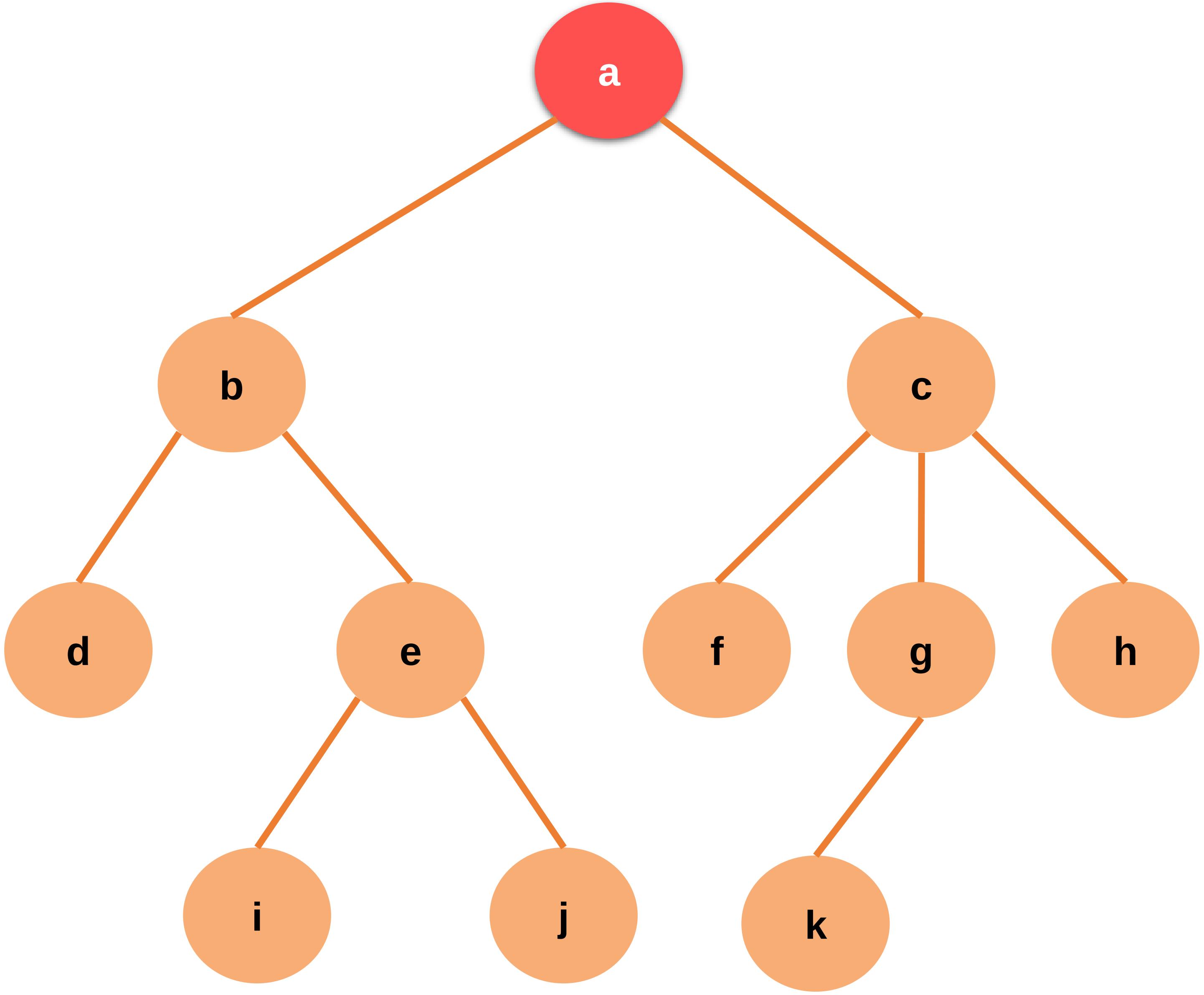
Examples

- **Pre-order ?**

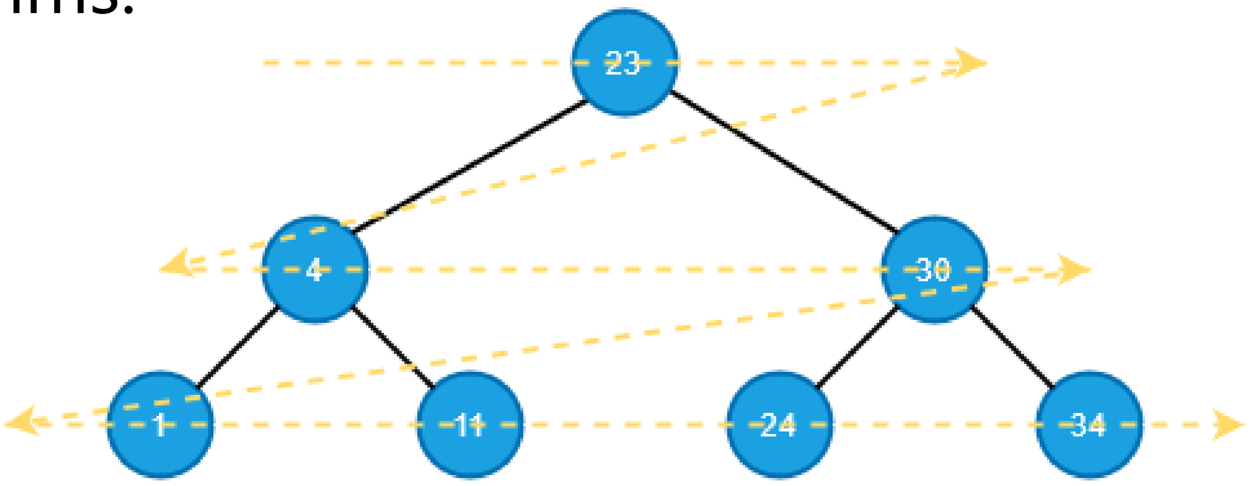
  a b d e i j c f g k h
- **In-order ?**

  d b i e j a f c k g h
- **Post-order ?**

  d i j e b f k g h c a



- Tree traversal algorithms can be classified broadly in two categories:
    - Depth-first search (DFS) algorithms:
        - Pre-order
        - In-order
        - Post-order



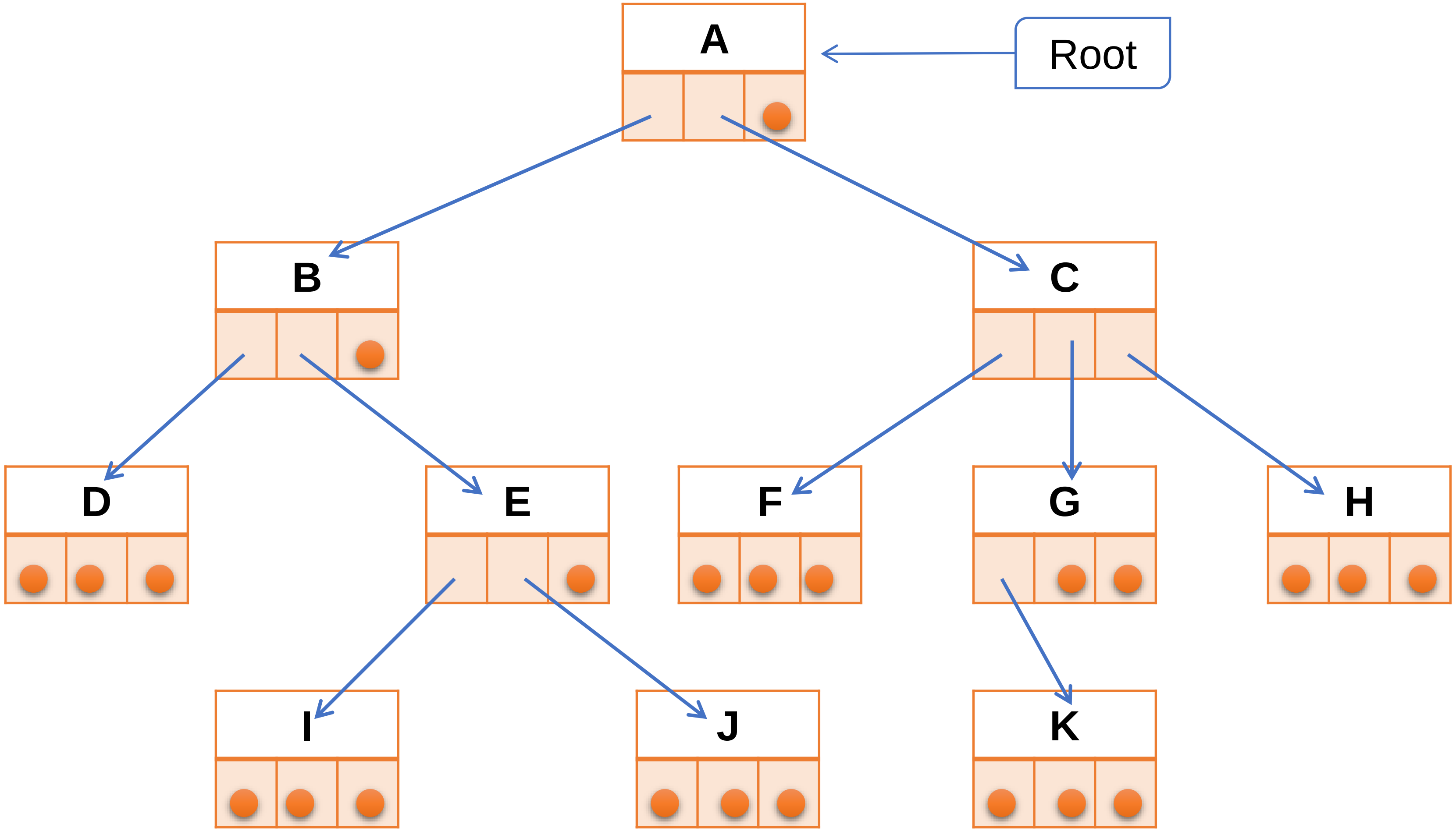
- Breadth-first search (BFS) algorithms:

level-order output: 23, 4, 30, 1, 11, 24, 34

    - Level order traversal: This visits nodes level-by-level and in left-to-right fashion at the same level

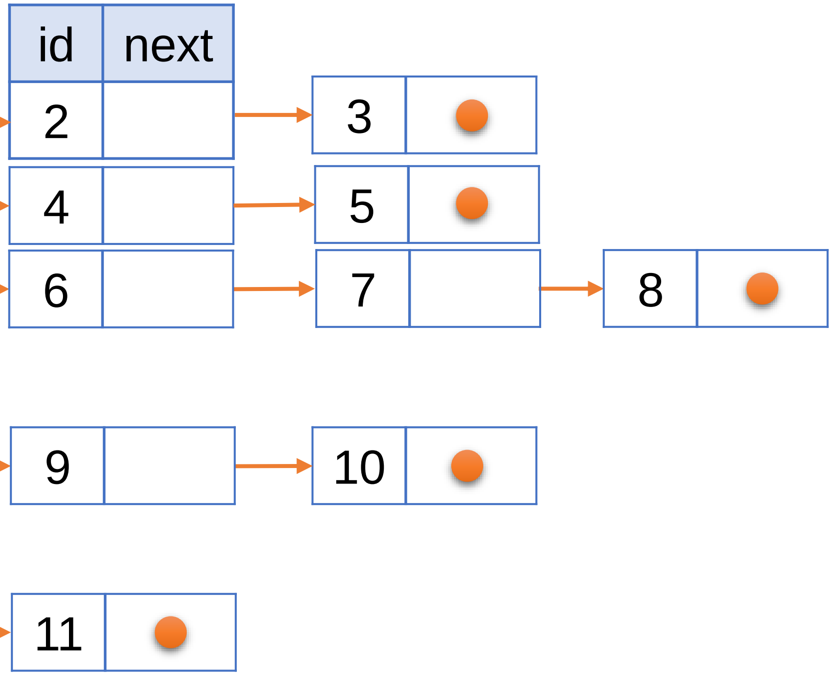
Tree Representation

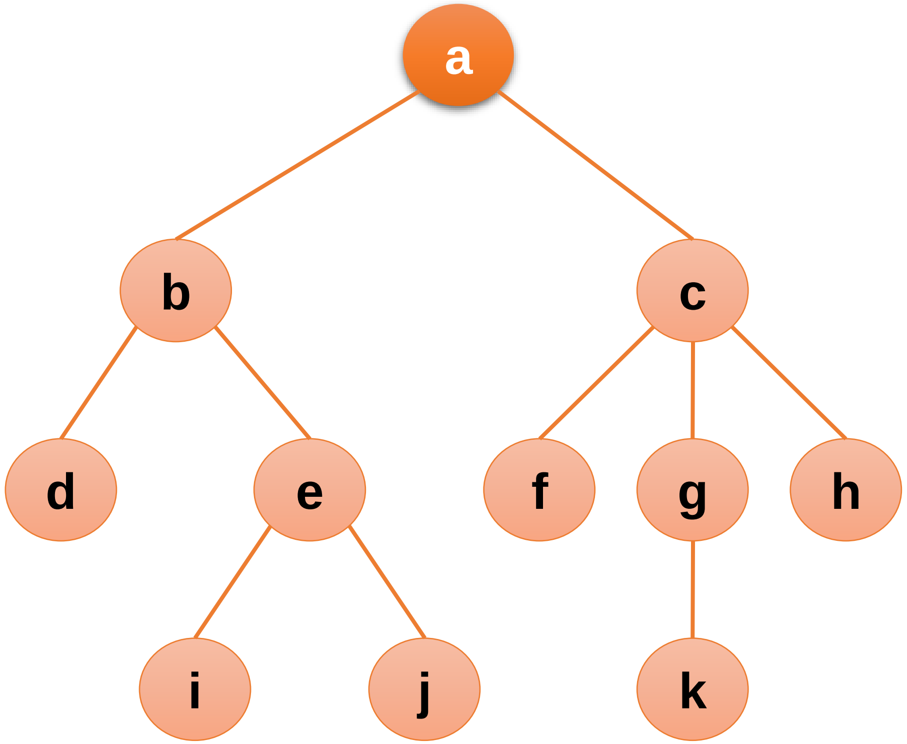
Tree Representation



Tree Representation

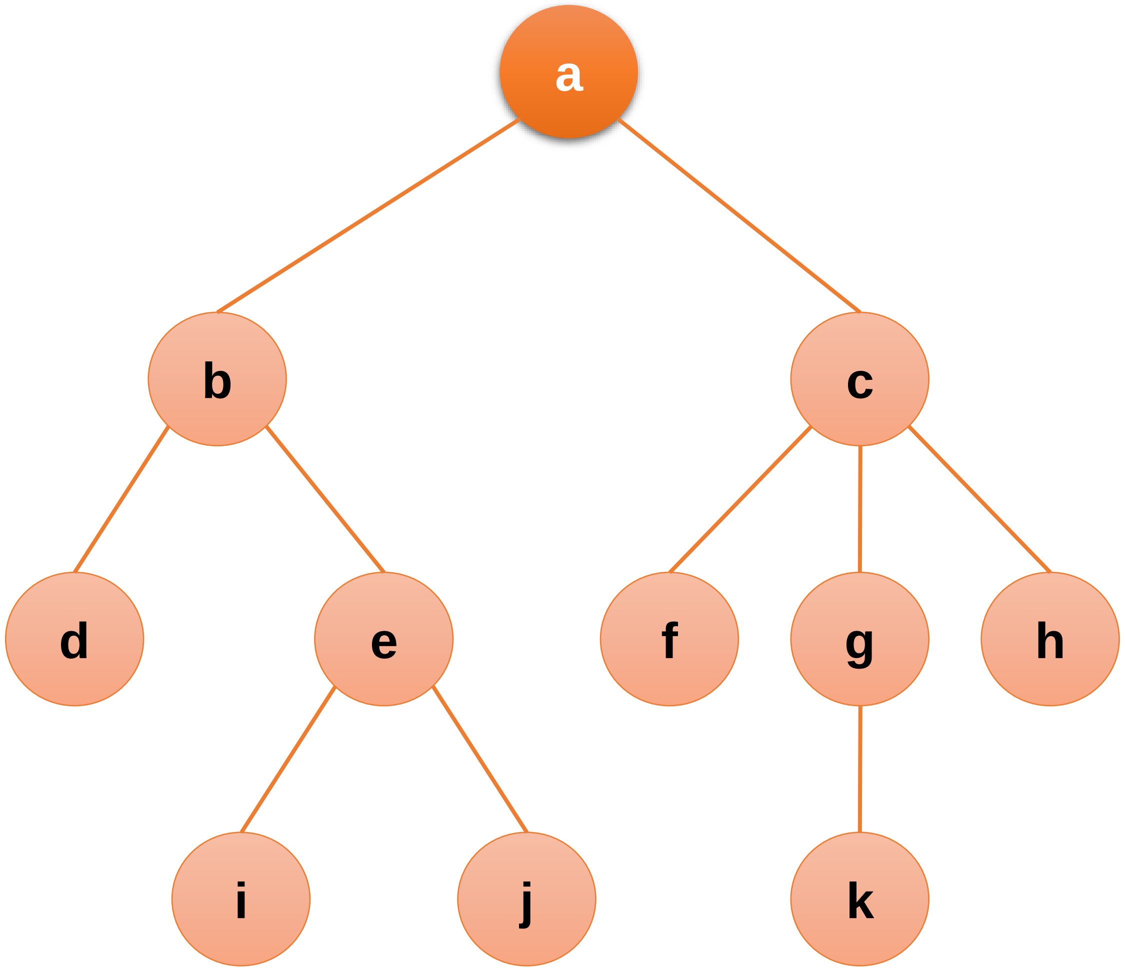
	info	child
1	a	
2	b	
3	c	
4	d	
5	e	
6	f	
7	g	
8	h	
9	i	
10	j	
11	k	



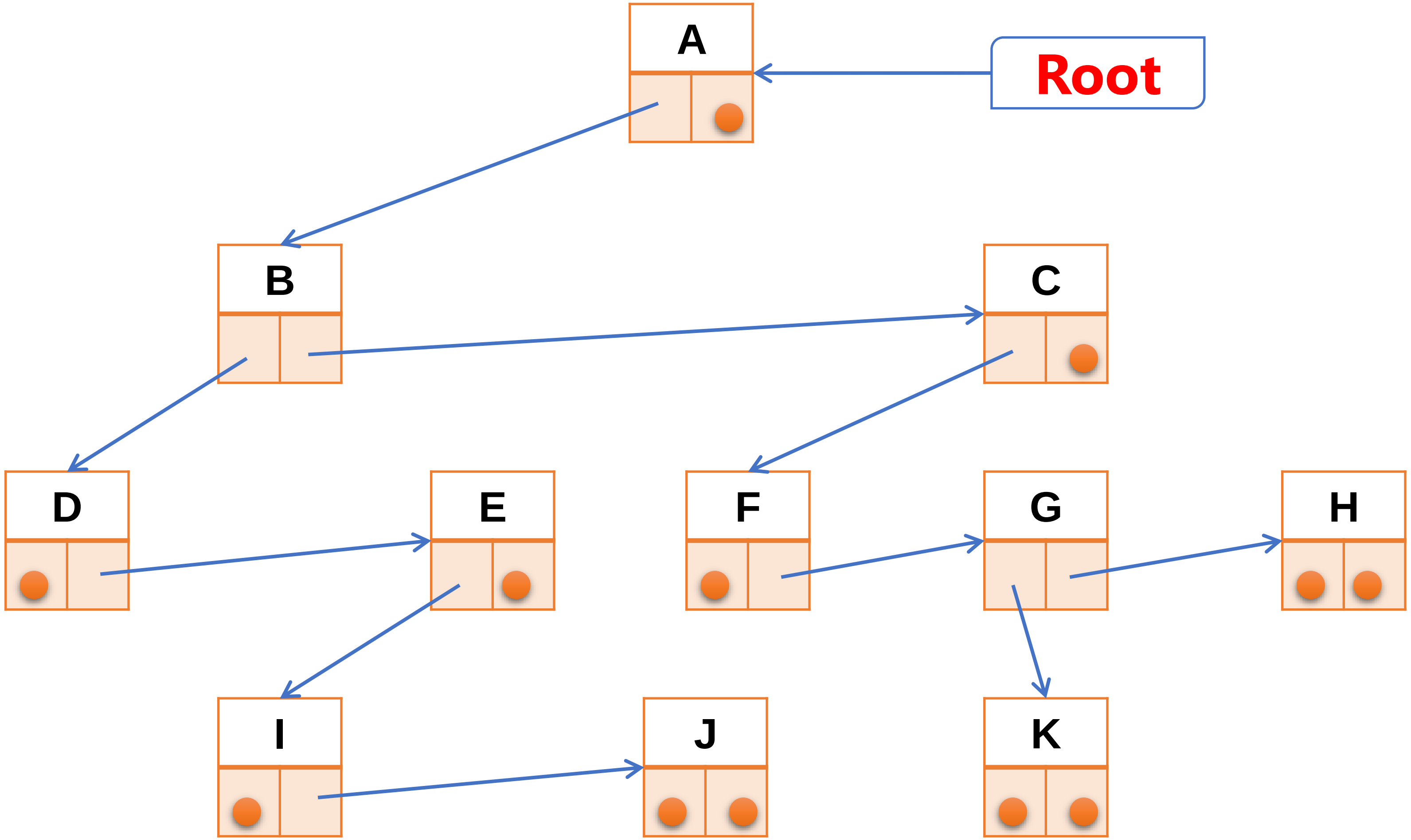


Tree Representation

	Info	Eldest Child	Next Sibling
1	a	2	0
2	b	4	3
3	c	6	0
4	d	0	5
5	e	9	0
6	f	0	7
7	g	11	8
8	h	0	0
9	i	0	10
10	j	0	0
11	k	0	0

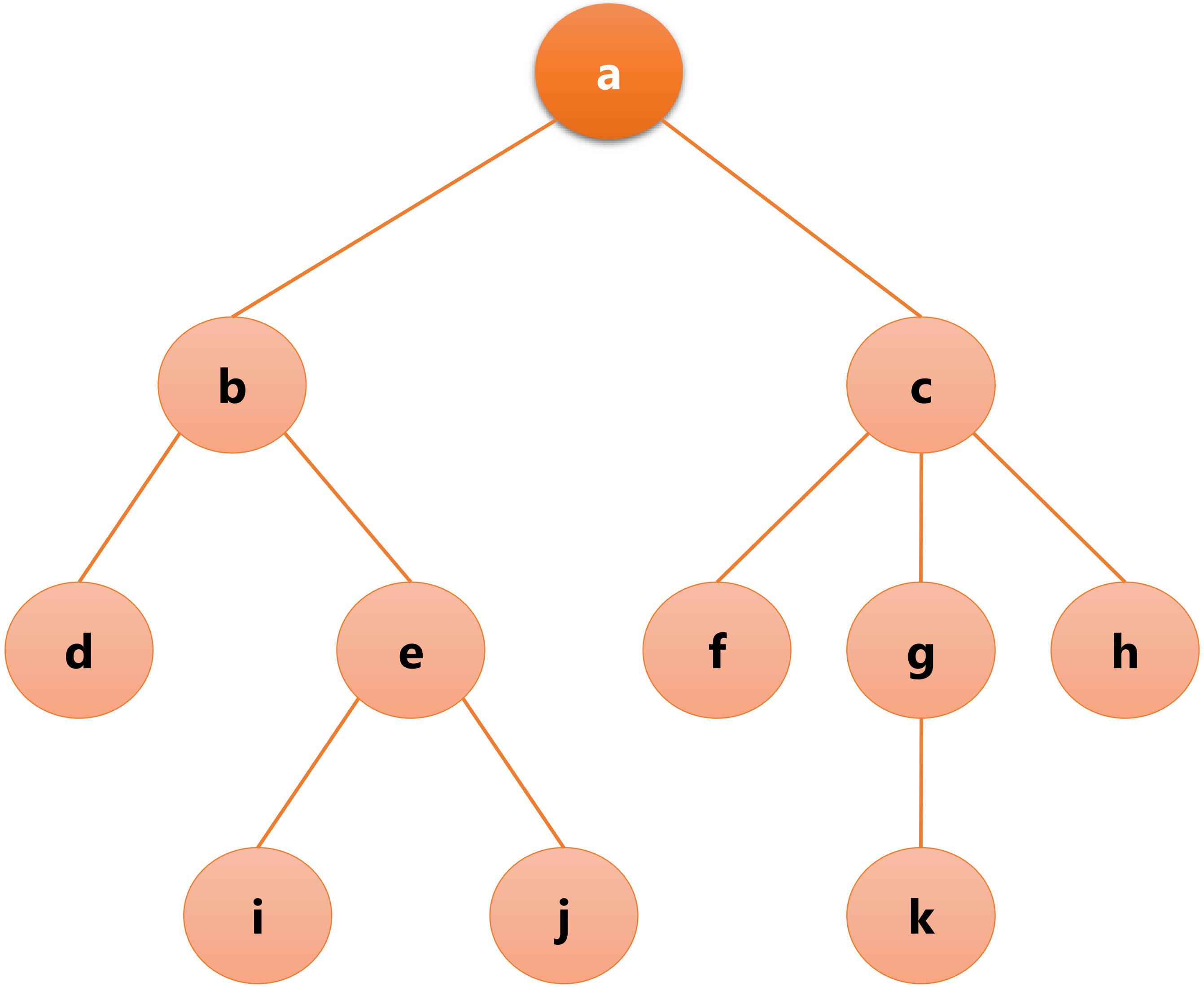


Tree Representation



Tree Representation

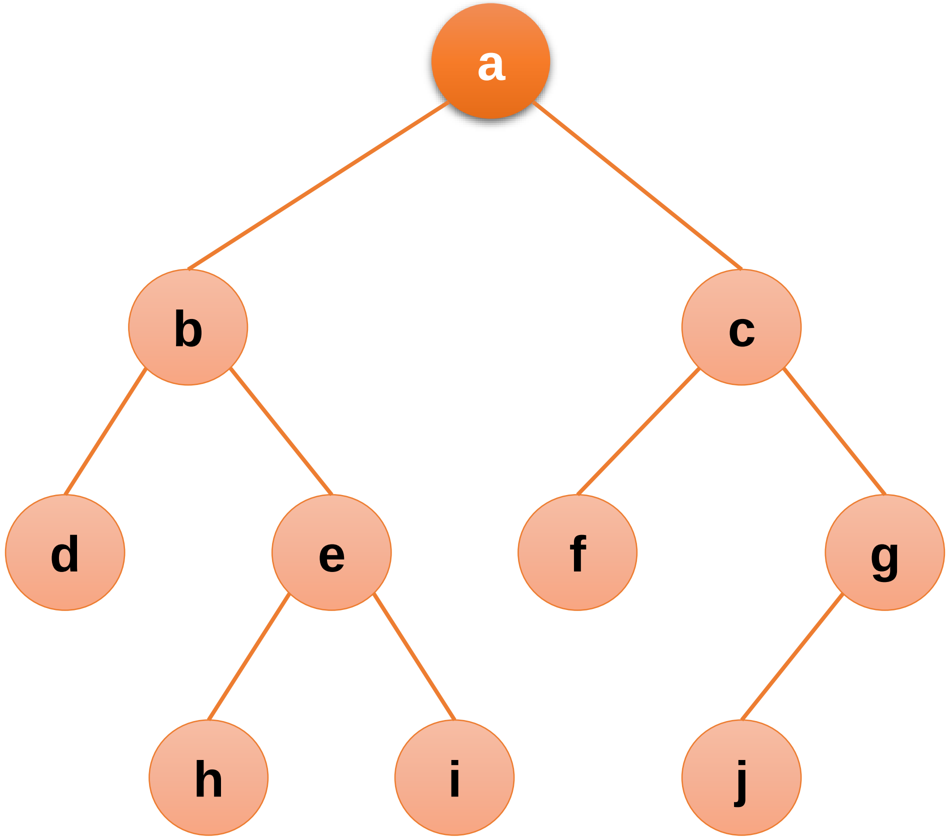
	Info	Parent
1	a	0
2	b	1
3	c	1
4	d	2
5	e	2
6	f	3
7	g	3
8	h	3
9	i	5
10	j	5
11	k	7



Binary Tree

Binary Tree

- Set T of nodes that is either empty or partitioned into disjoint subsets.
  - Single node r, the root
  - Two possibly empty sets that are binary trees, called left and right subtrees of r.
- Other definition: A rooted binary tree has a root node and every node has **at most** two children.

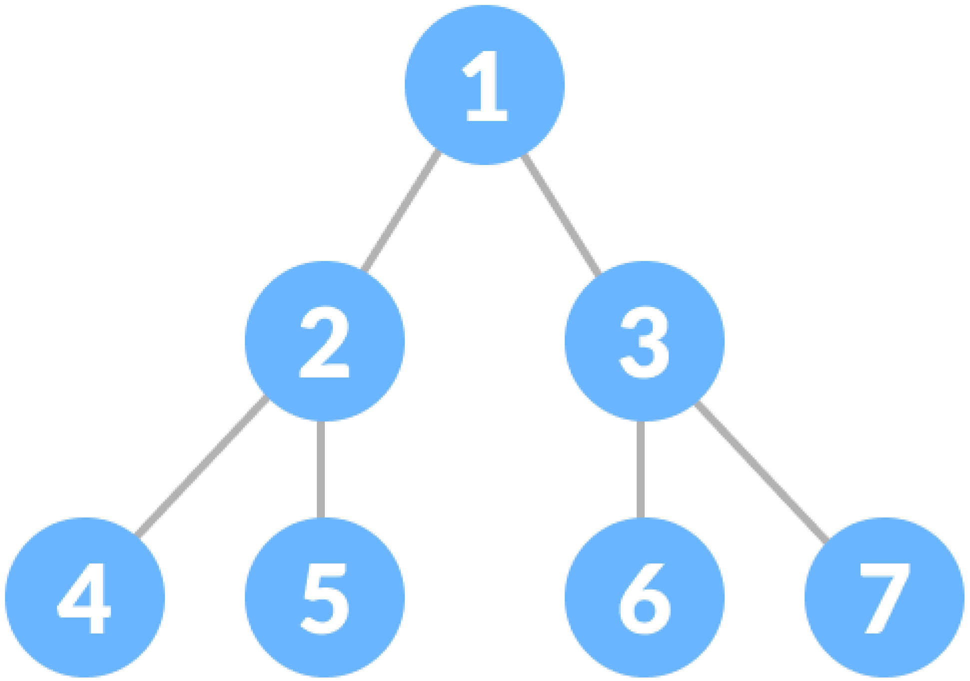


Types of Binary Tree

- Perfect binary tree
- Complete binary tree
- Full binary tree
- Heap

Perfect Binary Tree

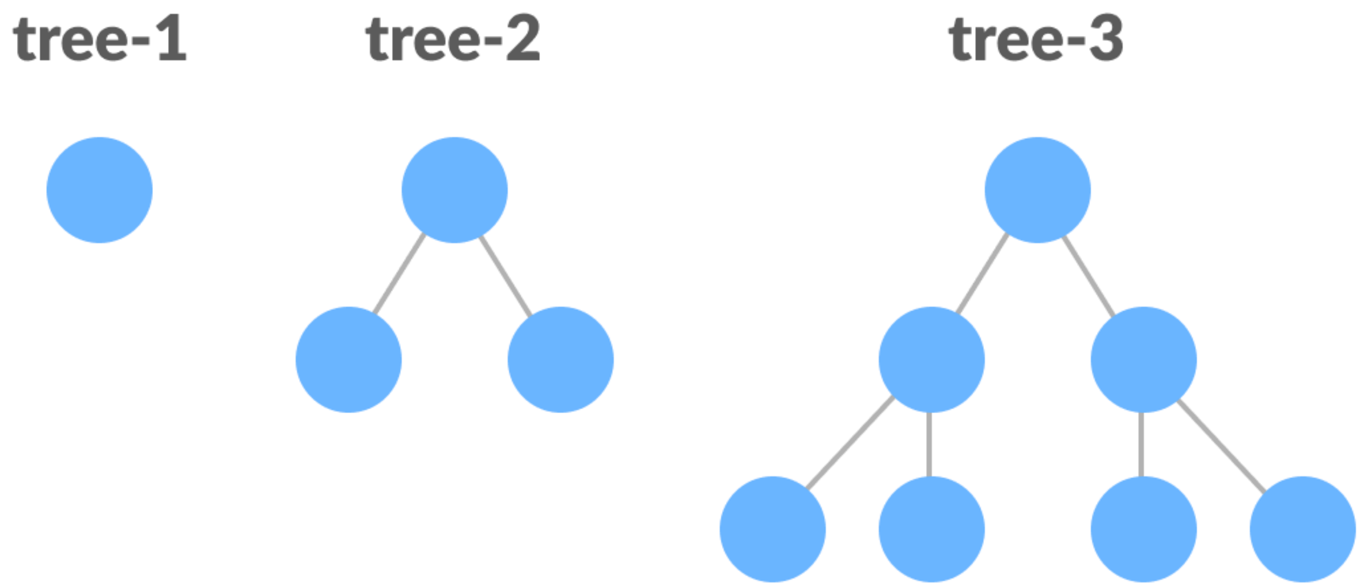
- A perfect binary tree is a binary tree in which
  - all interior nodes have two children
  - and all leaves have the same depth or same level.



- In a perfect binary tree of height h, all nodes that are at a level less than h have two children each.

Perfect Binary Tree

- If `T` is empty, `T` is a perfect binary tree of height 0.
- If `T` is not empty and has height $h > 0$, `T` is a perfect binary tree if its root's subtrees are both perfect binary trees of height $h - 1$.

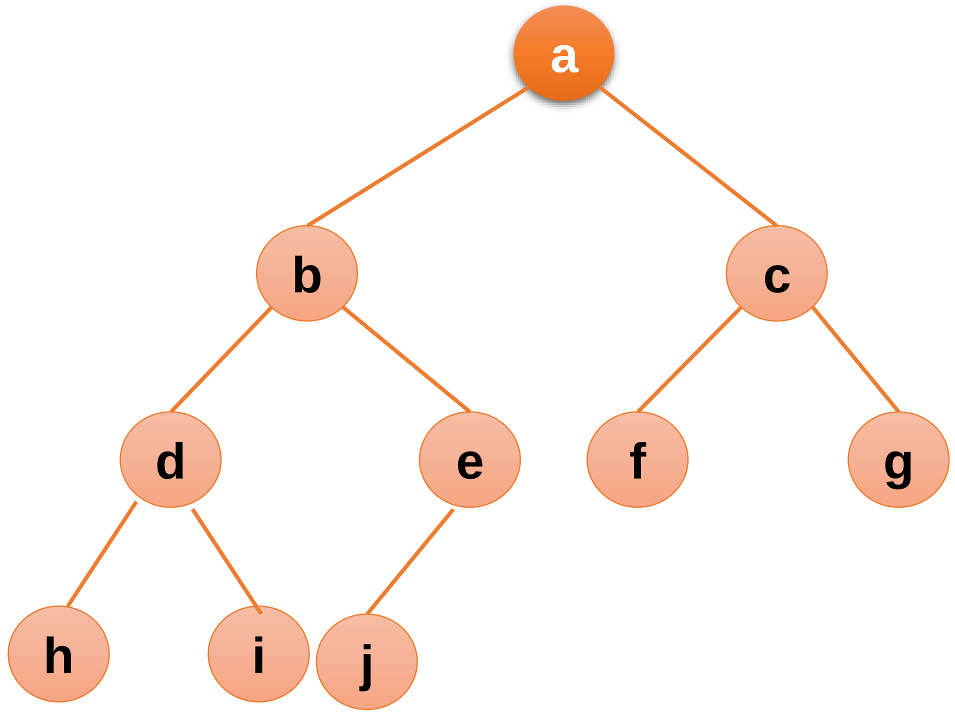


Complete Binary Tree

- A **complete binary tree** of height h is a binary tree that is perfect down to level h – 1, with level h filled in from left to right.
- In a complete binary tree every level, except possibly the last, is **completely filled**, and all nodes in the last level are as far left as possible.
- Other definition: A complete binary tree is a perfect binary tree whose rightmost leaves (perhaps all) have been removed.

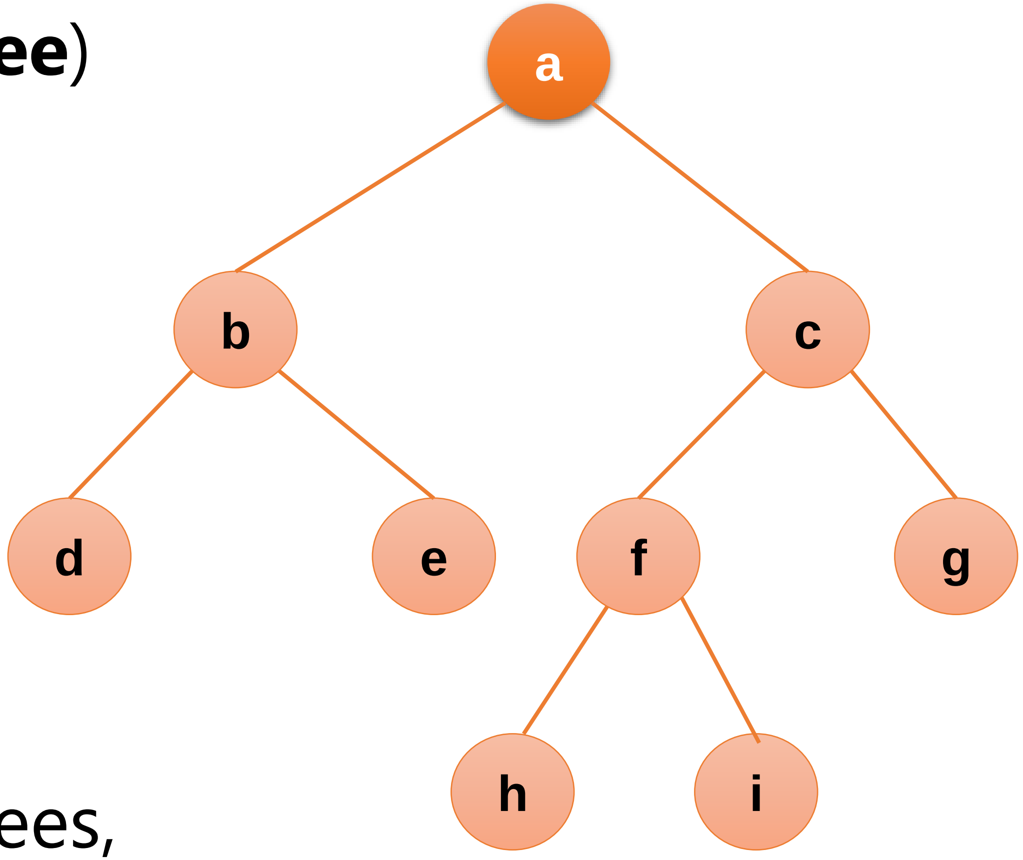
Complete Binary Tree

- A binary tree is complete if
  - All nodes at level h – 2 and above have two children each, and
  - When a node at level h – 1 has children, all nodes to its left at the same level have two children each, and
  - When a node at level h – 1 has one child, it is a left child



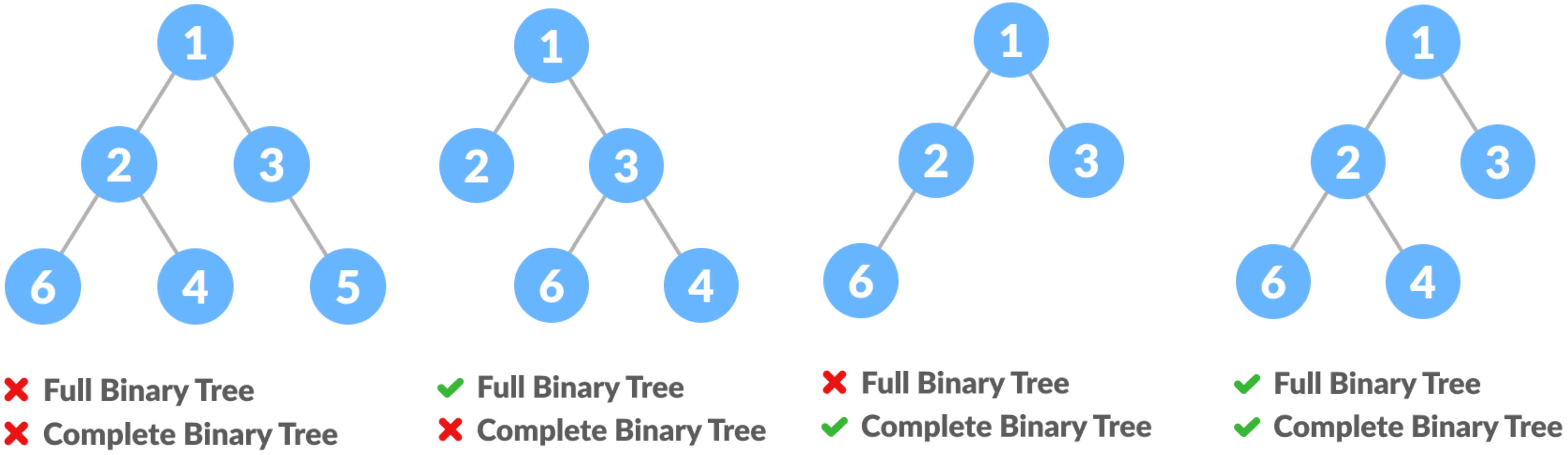
- A full binary tree (sometimes referred to as a **proper binary tree** or a **plane binary tree**) is a binary tree in which **every node has either 0 or 2 children**.

- A full binary tree is either:
  - A single vertex.
  - A tree whose root node has two subtrees, both of which are full binary trees.



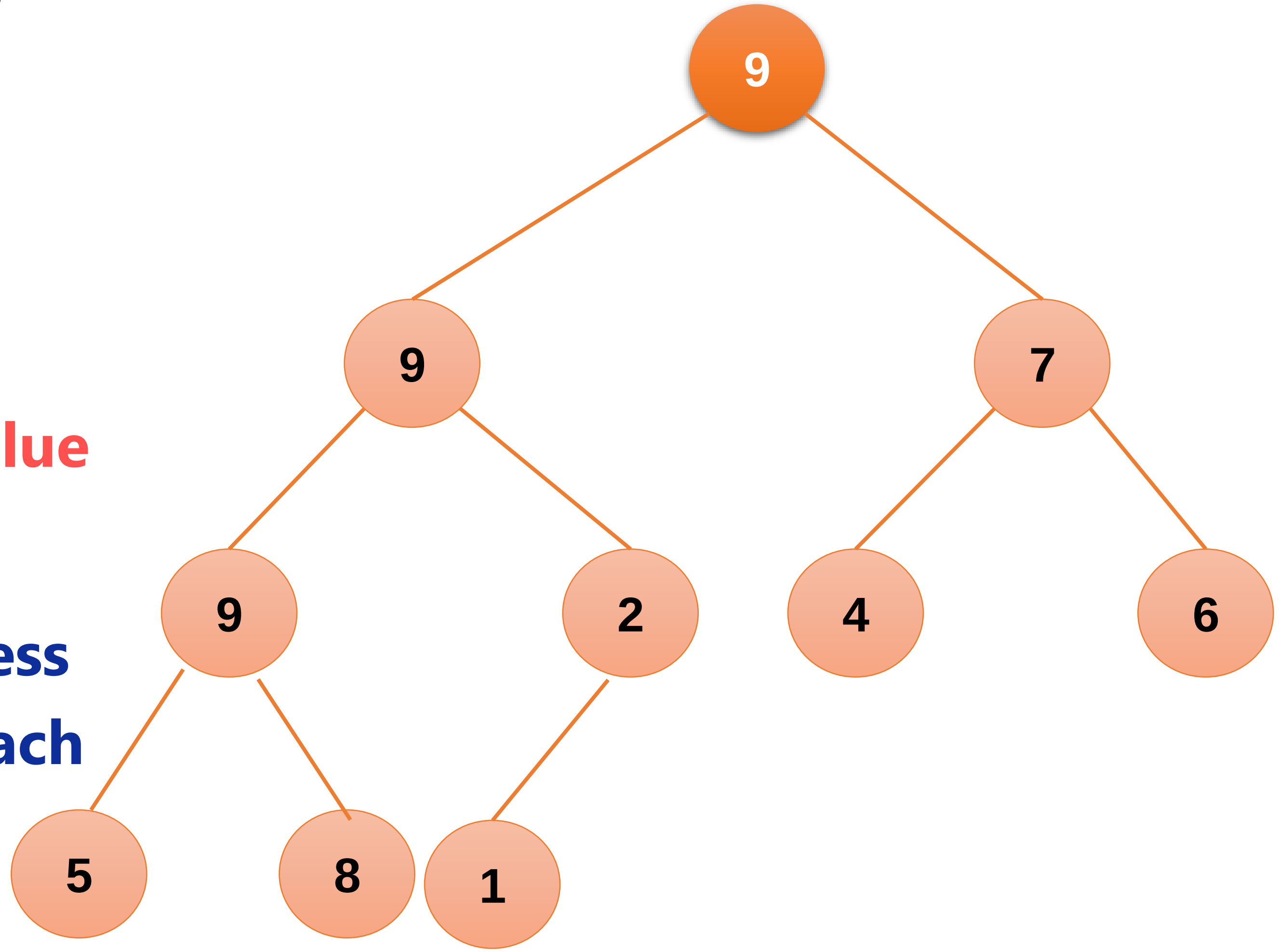
Complete/Full Binary Tree

- A complete binary tree is just like a full binary tree, but with two major differences
  - All the leaf elements must lean towards the left.
  - The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Heap

- A heap is a **complete binary tree** that either is empty or
  - Its root
    - **(Max-heap): Contains a value greater than or equal to the value in each of its children**
    - **(Min-heap): Contains a value less than or equal to the value in each of its children**
  - Has heaps as its subtrees

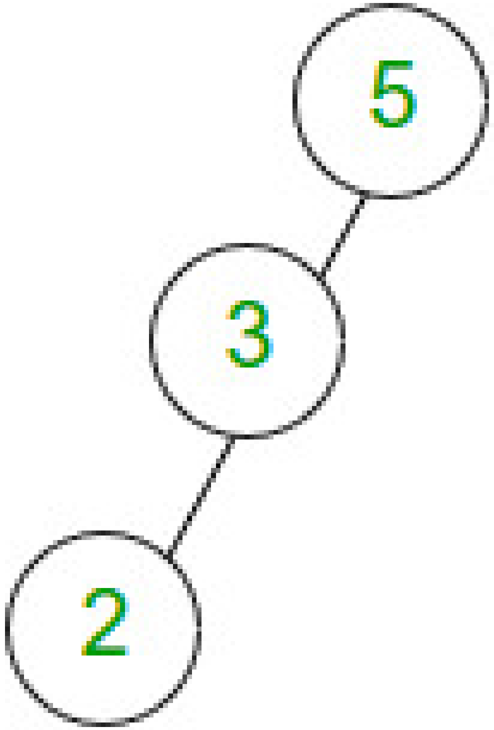


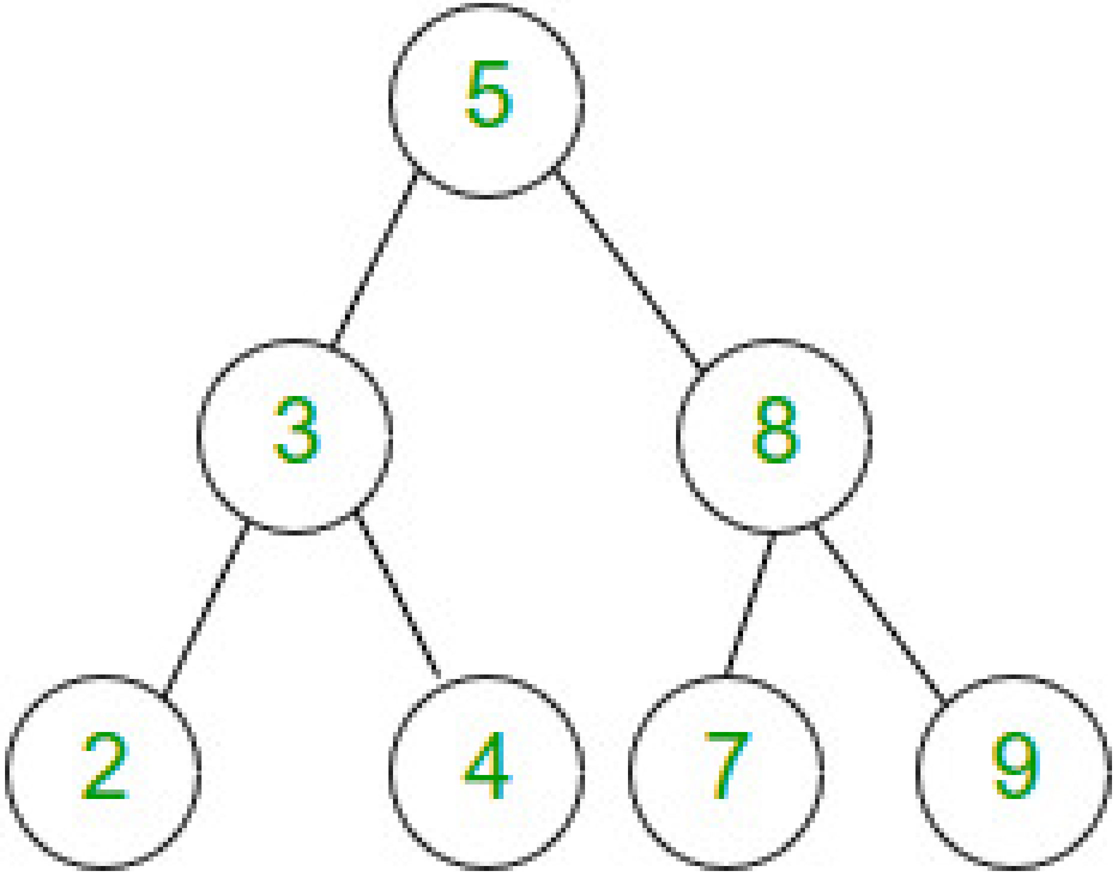
Exercise: Number of Nodes

- Given a binary tree T height of h.
  - **What is the maximum number of nodes?**
  - **What is the minimum number of nodes?**

Exercise: Number of Nodes

- Given a binary tree T height of h = 3.





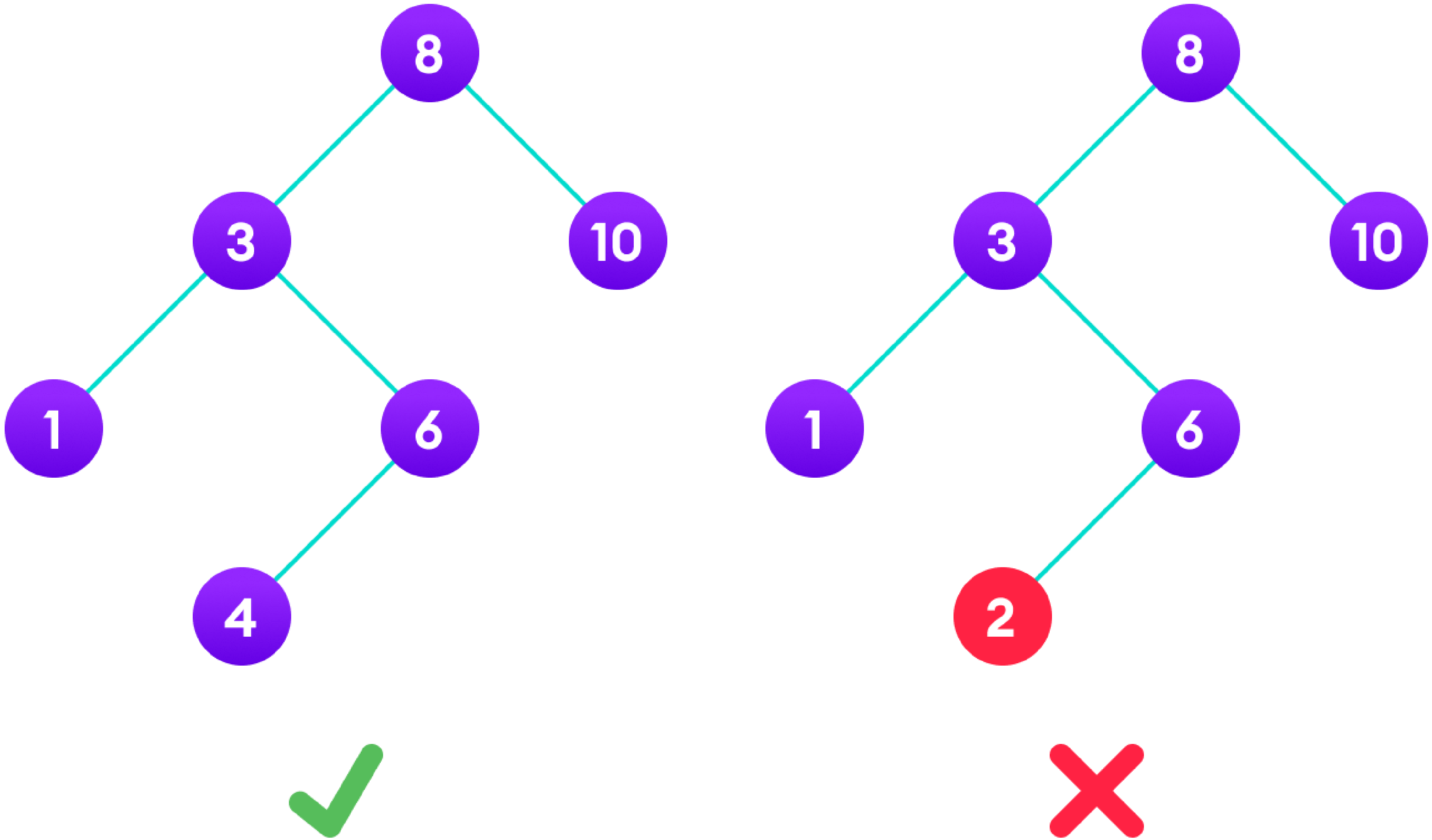
Exercise: Height of Tree

- Given a binary tree T with n nodes.
  - **What is the maximum height of that tree?**
  - **What is the minimum height of that tree?**

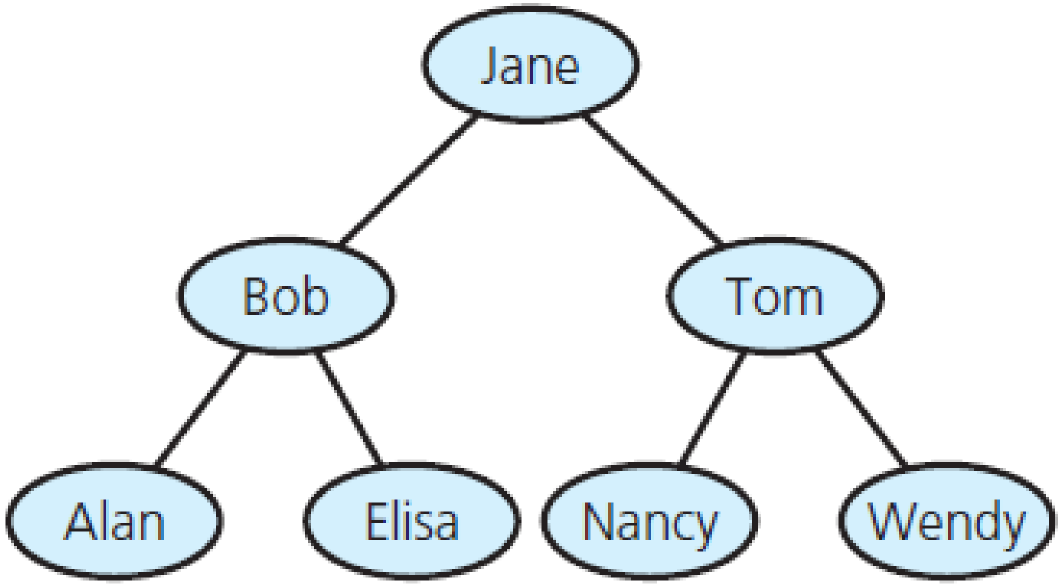
Facts about Perfect Binary Trees

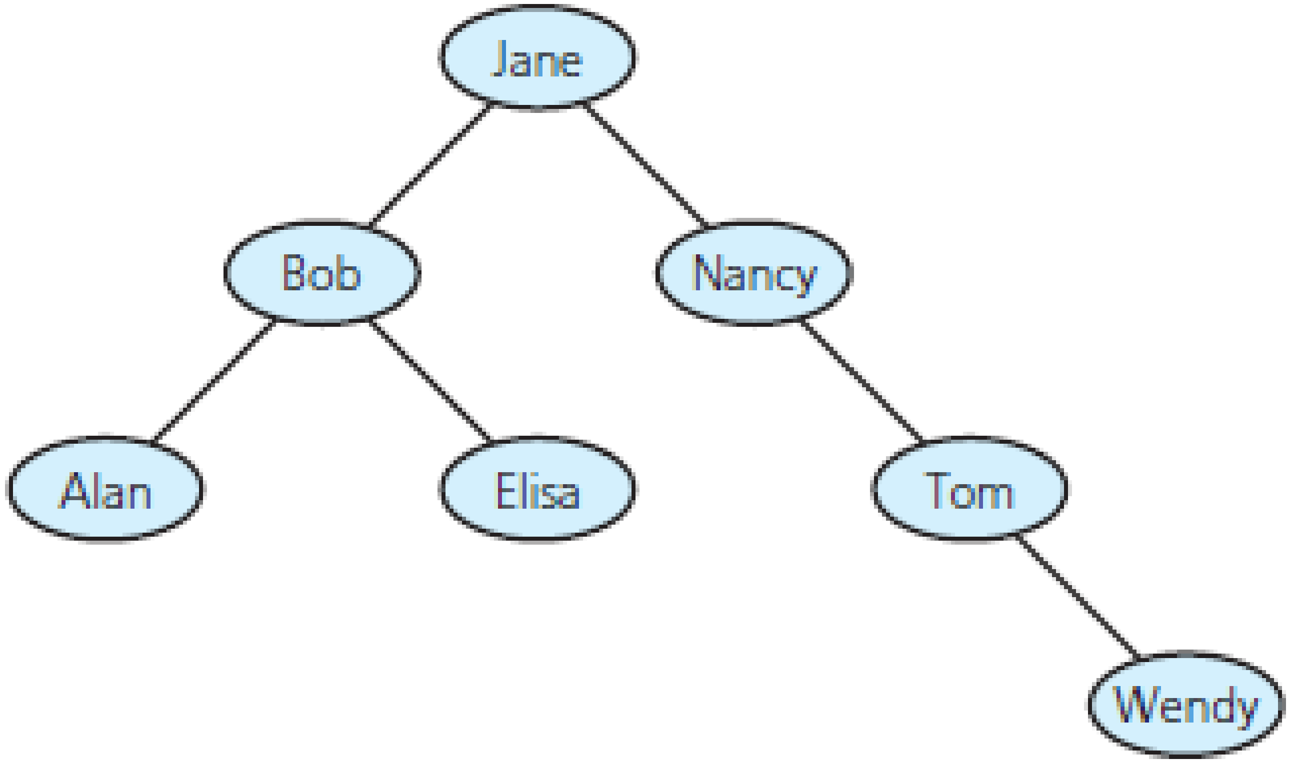
- A perfect binary tree of height $h \geq 0$ has $2^h - 1$ nodes.
- You cannot add nodes to a perfect binary tree without increasing its height.
- The maximum number of nodes that a binary tree of height h can have is $2^h - 1$.
- The minimum height of a binary tree with n nodes is

$$[\log_2(n + 1)]$$


8
3
10
1
6
4
8
3
10
1
6
2

Binary Search Tree





BSTree: Operations

- Insert (a key)
- Search (a key)
- Remove (a key)
- Traverse
- Sort (based on key value)
- Rotate (Left rotation, Right rotation)

```
Insert (root, Data)
{
   if (root is NULL) {
      Create a new_Node containing Data
      This new_Node becomes root of the tree
   }
   //Compare root's key with Key
   if root's Key is less than Data's Key
      Insert Key to the root's RIGHT subtree
   else if root's Key is greater than Data's Key
      Insert Key to the root's LEFT subtree
   else
      Do nothing //Explain why?
}
```

- Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?

15, 5, 12, 8, 23, 1, 17, 21

15, 20, 40, 25, 70, 90, 80, 55, 60, 65, 30, 75

9, 1, 4, 2, 3, 9, 5, 8, 6, 7, 4

- Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?

W, T, N, J, E, B, A

W, T, N, A, B, E, J

A, B, W, J, N, T, E

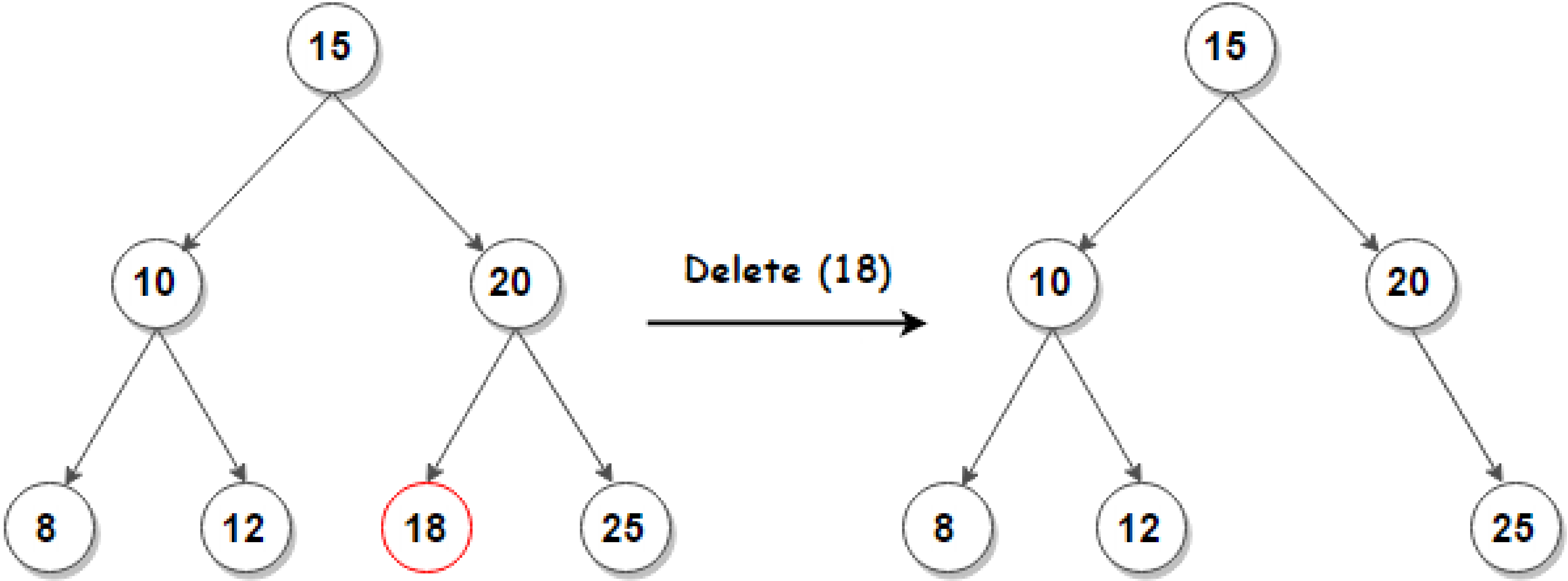
B, T, E, A, N, W, J

```
Search (root, Data)
{
   if (root is NULL) {
      return NOT_FOUND;
   }
   //Compare root's key with Key
   if root's Key is less than Data's Key
      Search Data in the root's RIGHT subtree
   else if root's Key is greater than Data's Key
      Search Data in the root's LEFT subtree
   else
      return FOUND //Explain why?
}
```

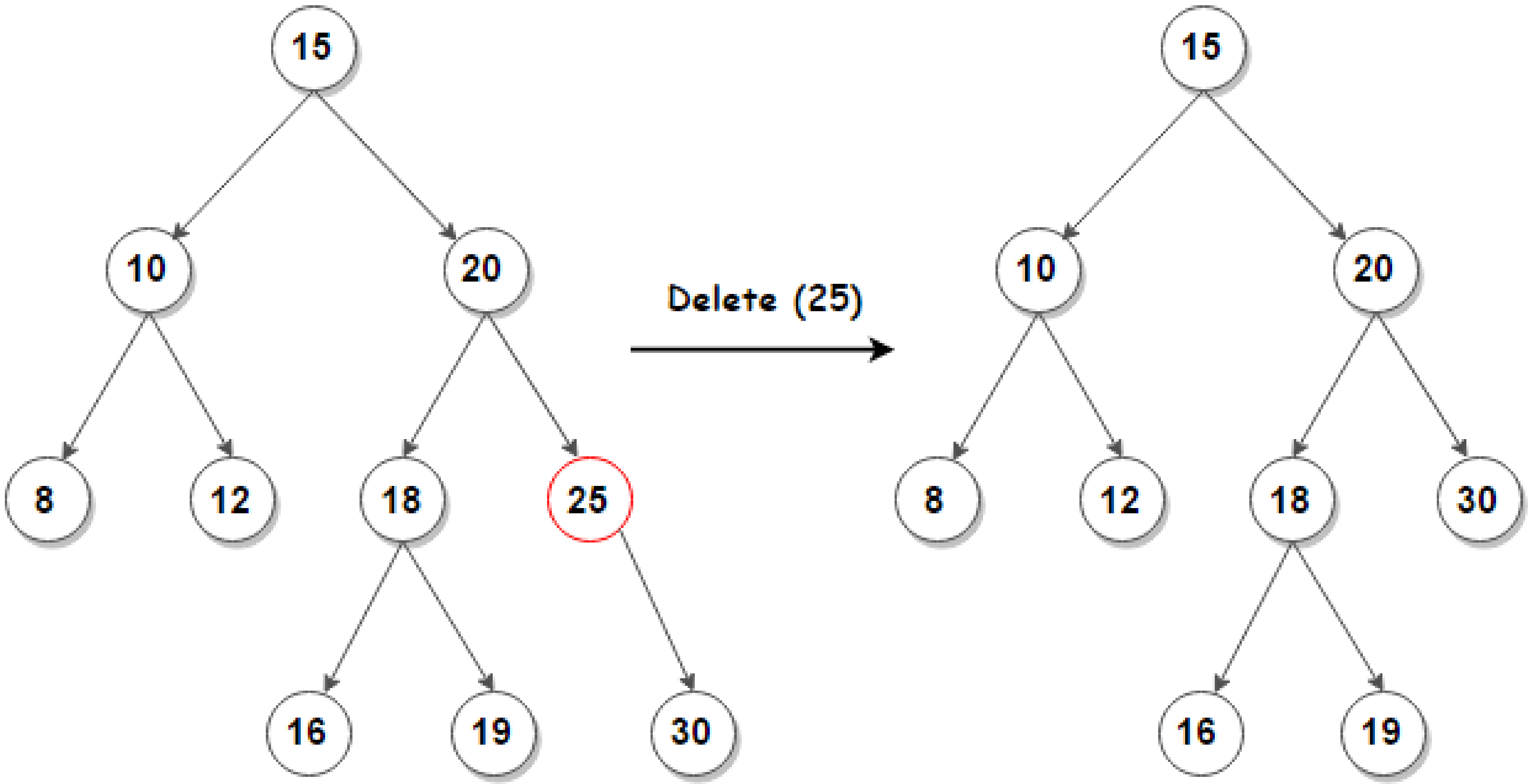
BSTree: Deletion

- When we delete a node, three possibilities arise.
- Node to be deleted:
  - **is leaf**: Simply remove from the tree.
  - has **only one child**: Copy the child to the node and delete the child
  - has **two children**:
    - Find in-order successor (**predecessor**) ***S_Node*** of the node.
    - Copy contents of ***S_Node*** to the node and delete the ***S_Node***.

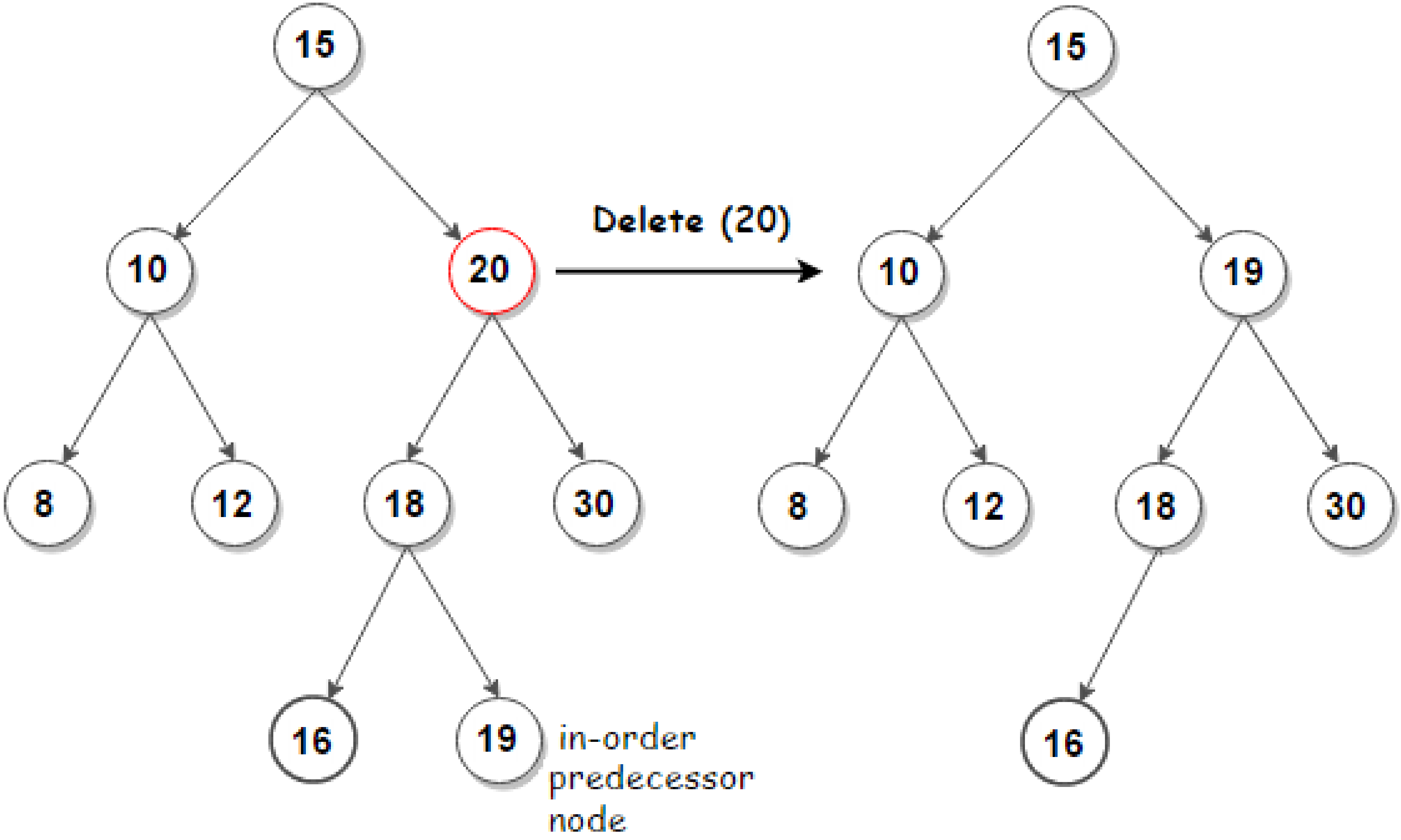
BSTree: Deletion



BSTree: Deletion



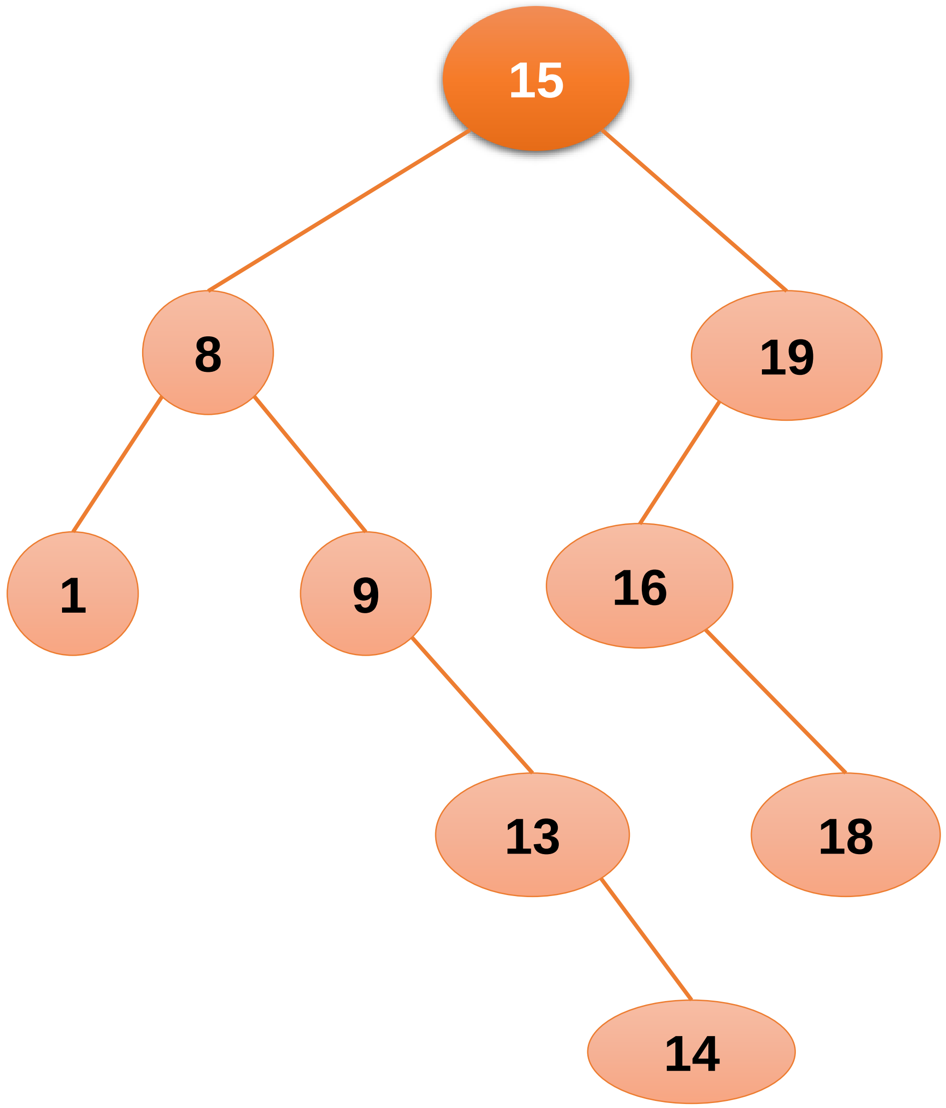
BSTree: Deletion



BStree: Traversals

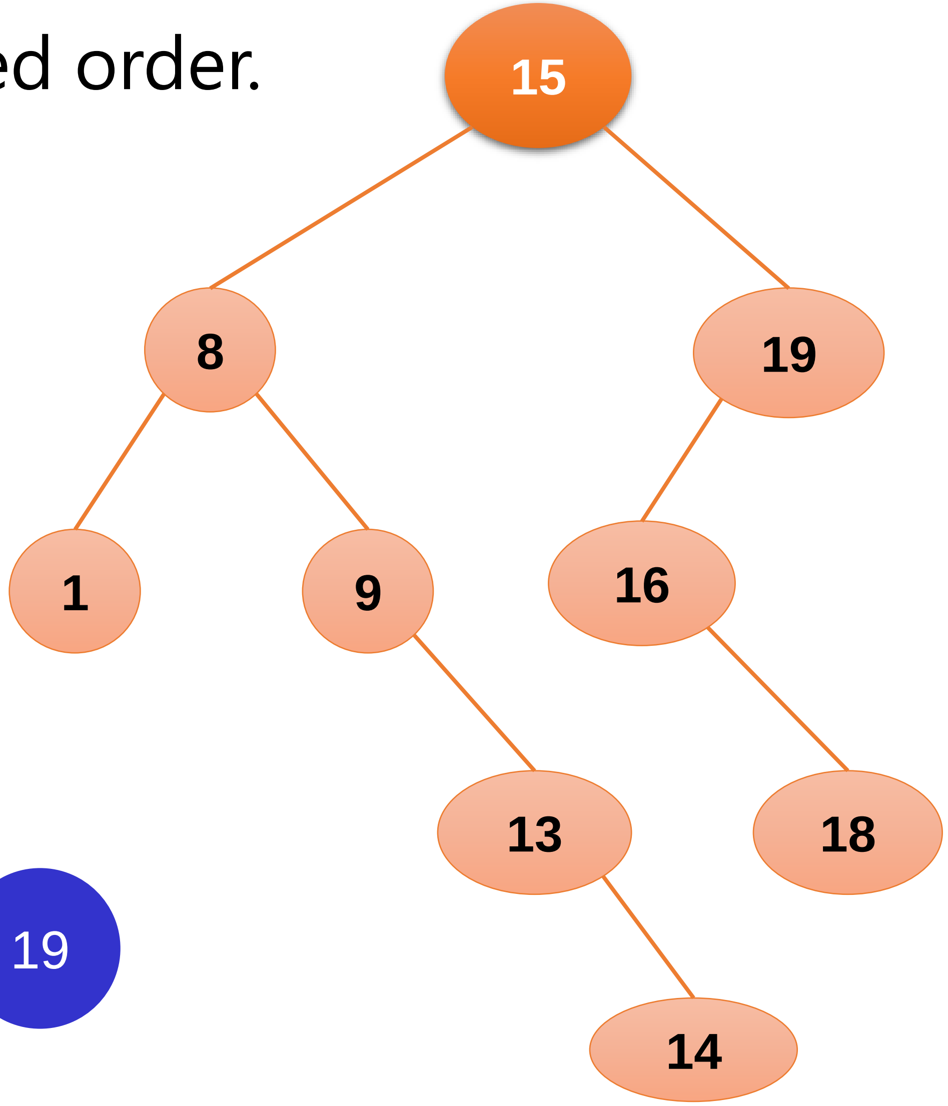
- **Pre-order:** Node - Left - Right
- **In-order:** Left - Node - Right
- **Post-order:** Left - Right - Node

- What are the pre-order, in-order and post-order traversals of this binary search tree?



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

- In-order traversal gives the key values in sorted order.



Efficiency of BST Operations

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Question

- Beginning with an empty binary search tree, what binary search tree is formed when inserting the following values in the order given?

2, 4, 6, 8, 10, 12, 14, 18, 20

- How about this tree?

Conclusion

- BST is fast in insertion and deletion when balanced. It is fast with a time complexity of O(log n).
- BST is also for fast searching, with a time complexity of O(log n) for most operations.
- We can also do range queries – find keys between N and M ($N \le M$).
- BST can automatically sort elements as they are inserted, so the elements are always stored in a sorted order.
- BST can be easily modified to store additional data or to support other operations. This makes it flexible.

THANK YOU
for YOUR ATTENTION