

Tree

Teaching Hours : 8

Syllabus

- Tree
- Concept and Definition
- Binary Tree
- Introduction and application
- Operation
- Types of Binary Tree
 - Complete
 - Strictly
 - Almost Complete
- Huffman algorithm
- Binary Search Tree
 - Insertion
 - Deletion
 - Searching
- Tree Traversal
 - Pre-order traversal
 - In-order traversal
 - Post-order traversal

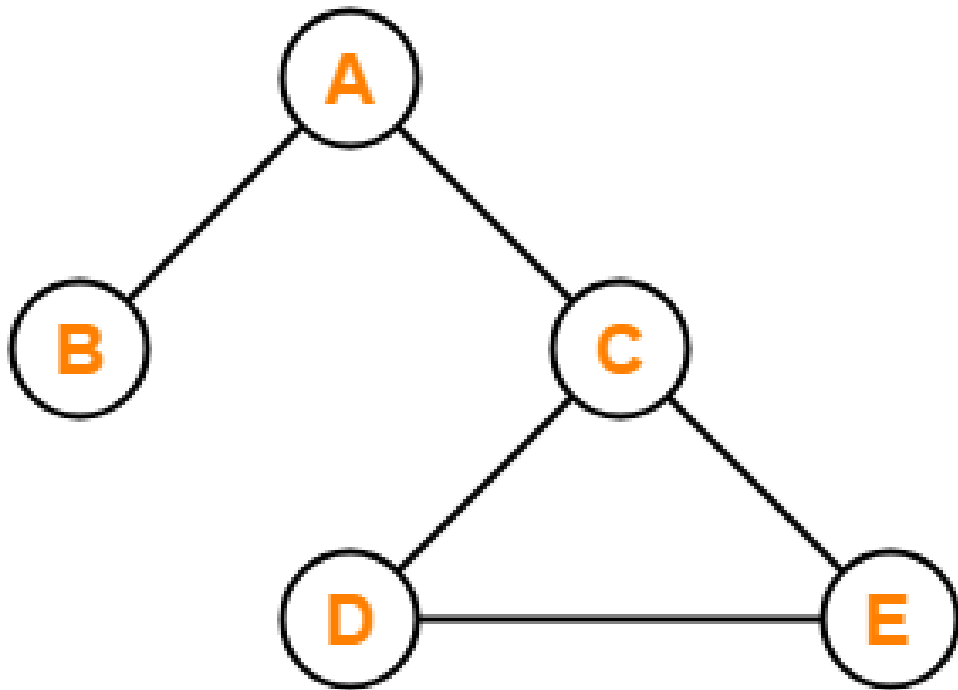
Tree

- A tree is a **non linear data structure** in which items are arranged in a **sorted sequence**. It is used to **represent hierarchical relationship** existing amongst several data items.

In other words,

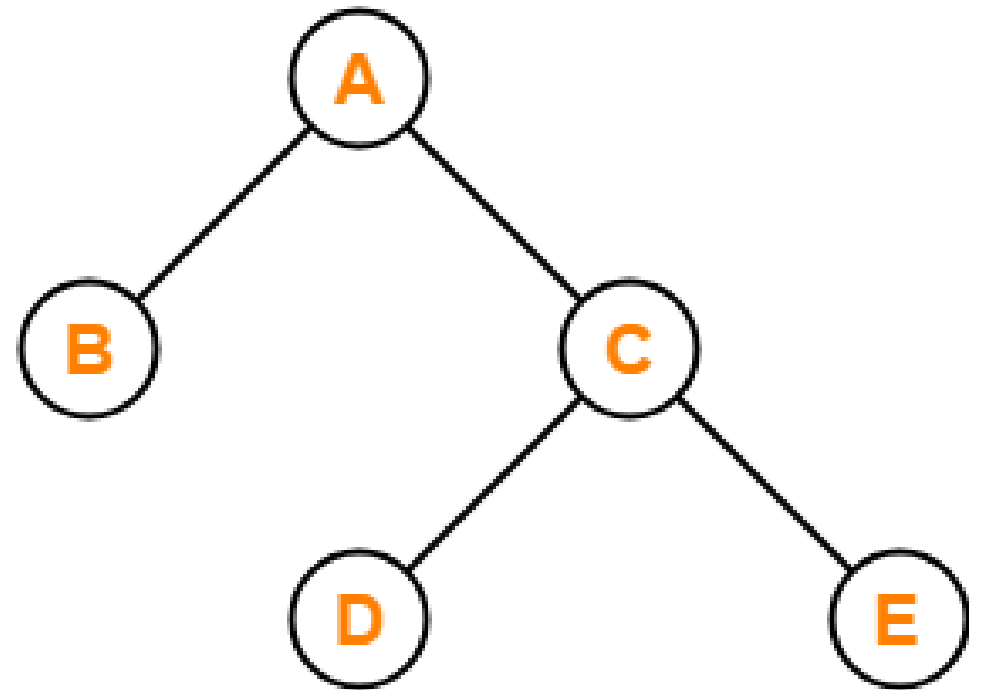
- A tree is an **abstract** model of a **hierarchical structure** that consists of nodes with a **parent-child** relationship.
 - Tree is a sequence of nodes.
 - There is a starting node known as root node.
 - Every node other than the root has a parent node.
 - Nodes may have any number of children.

A Tree



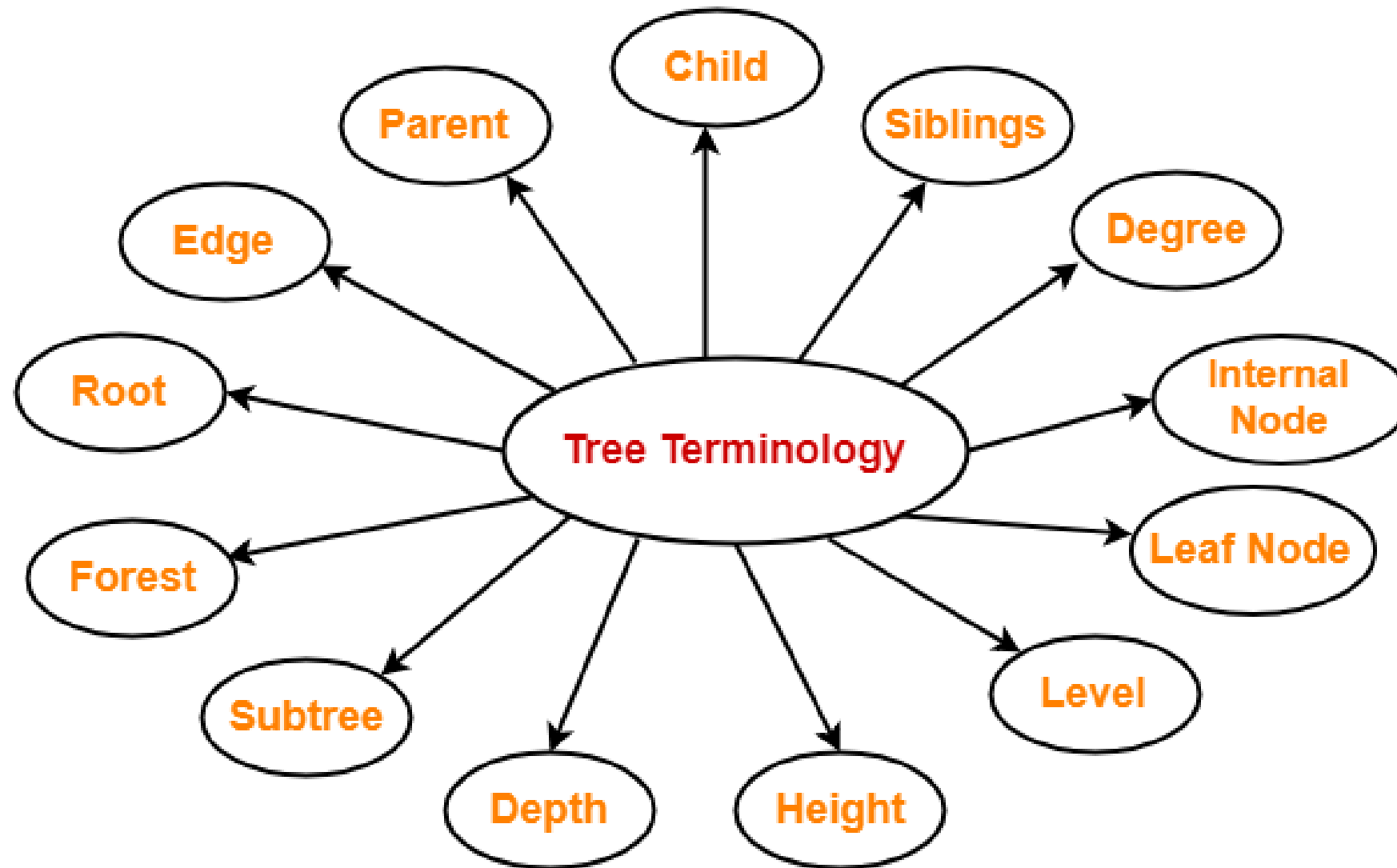
X

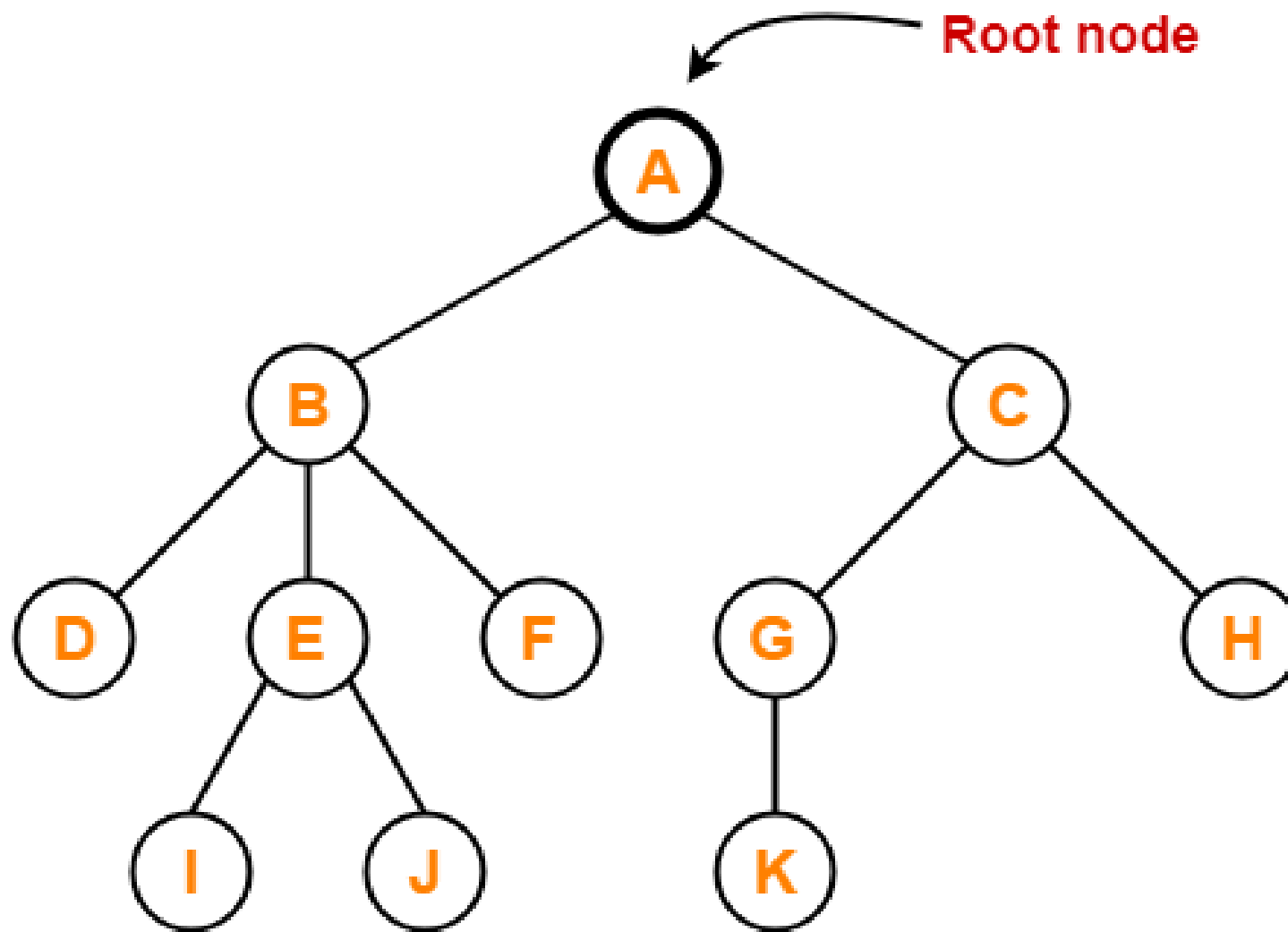
This graph is not a Tree

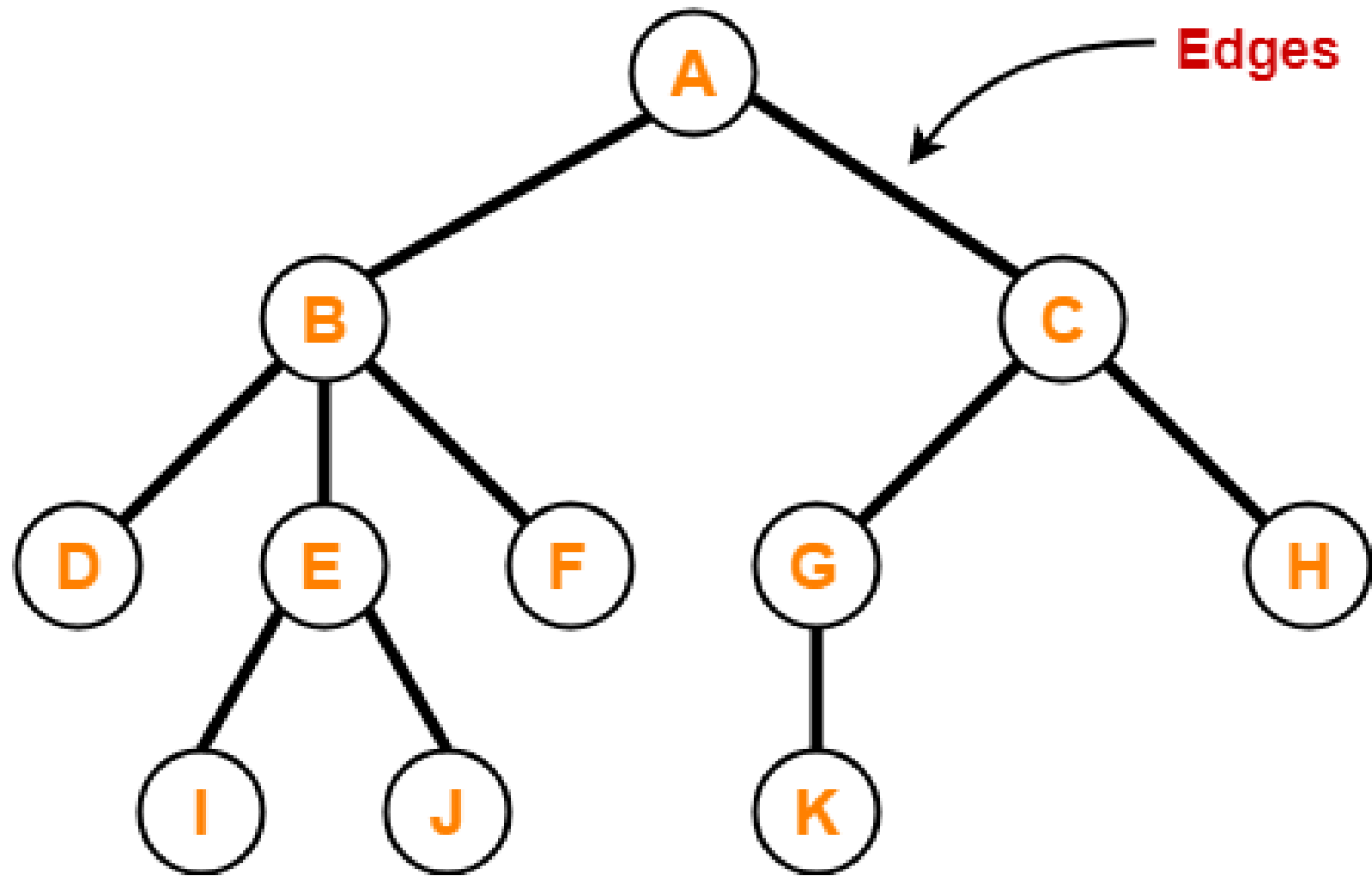


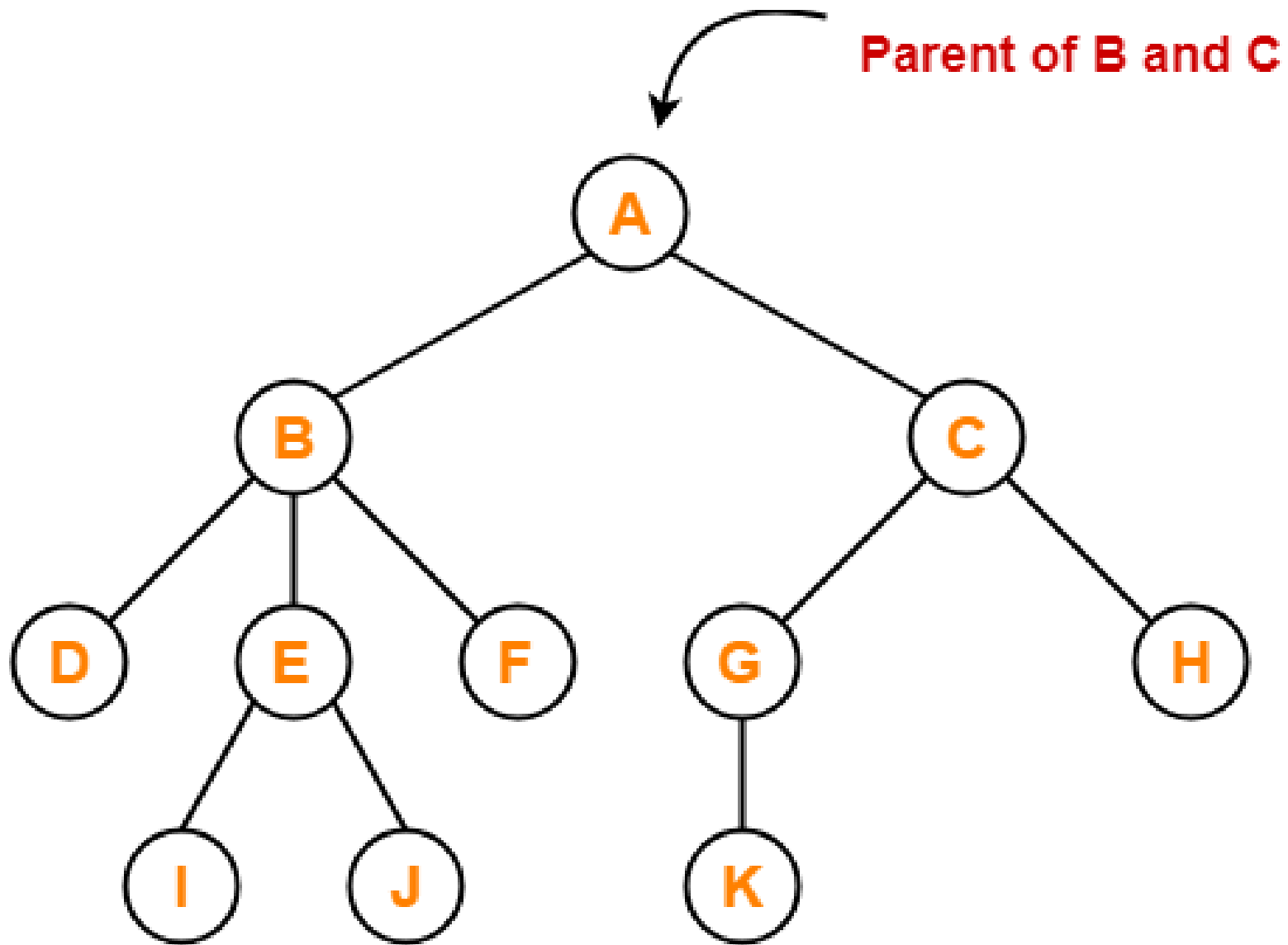
✓

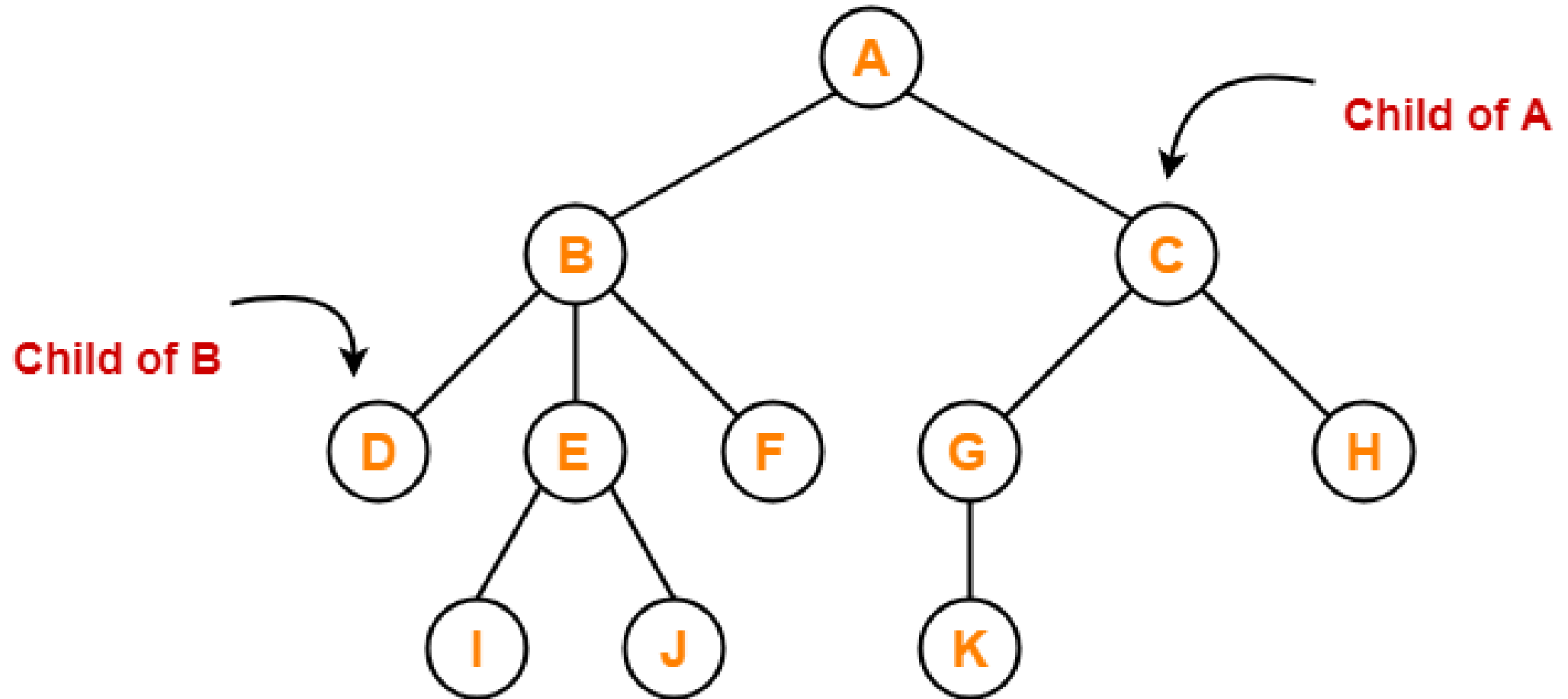
This graph is a Tree

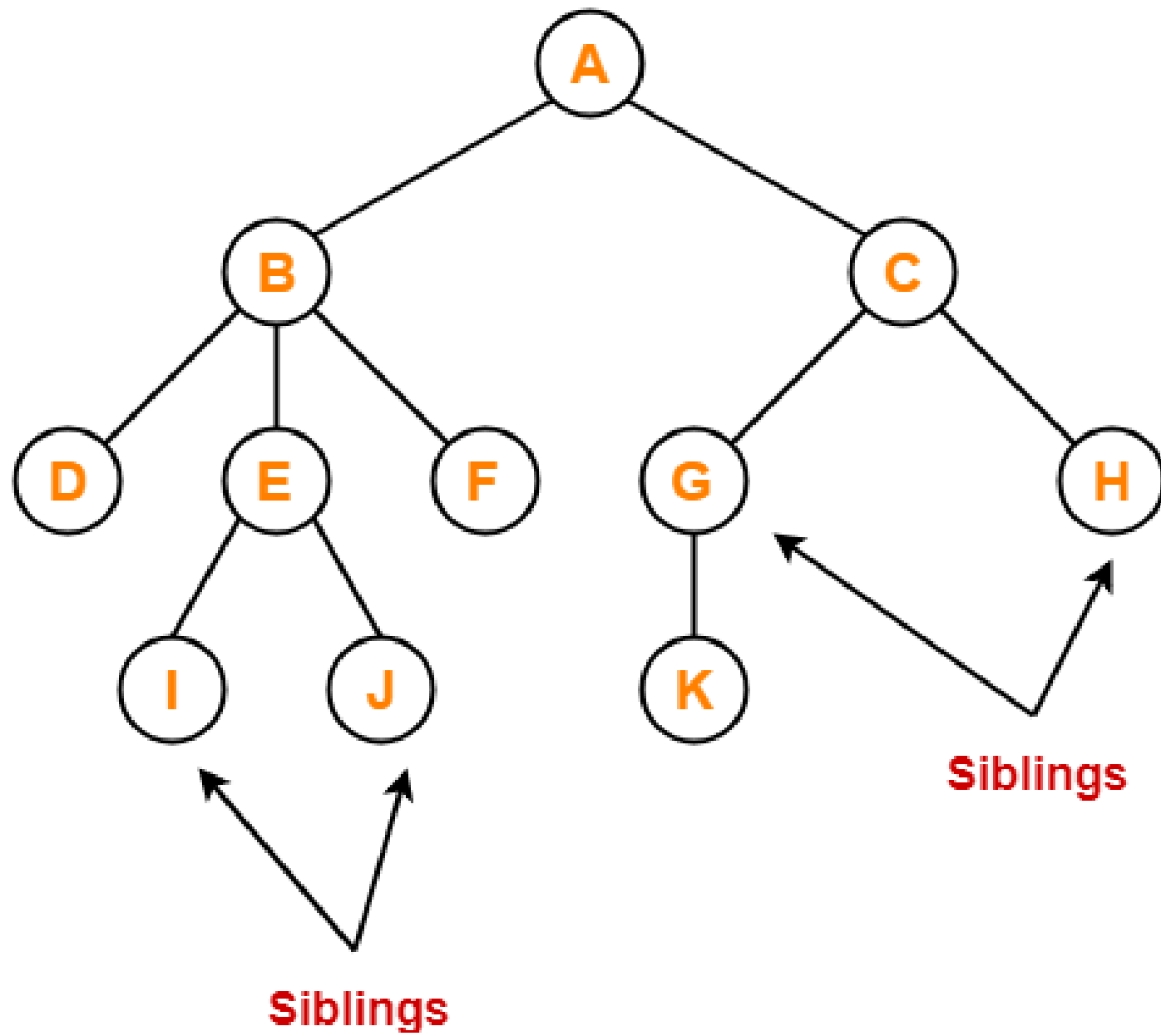


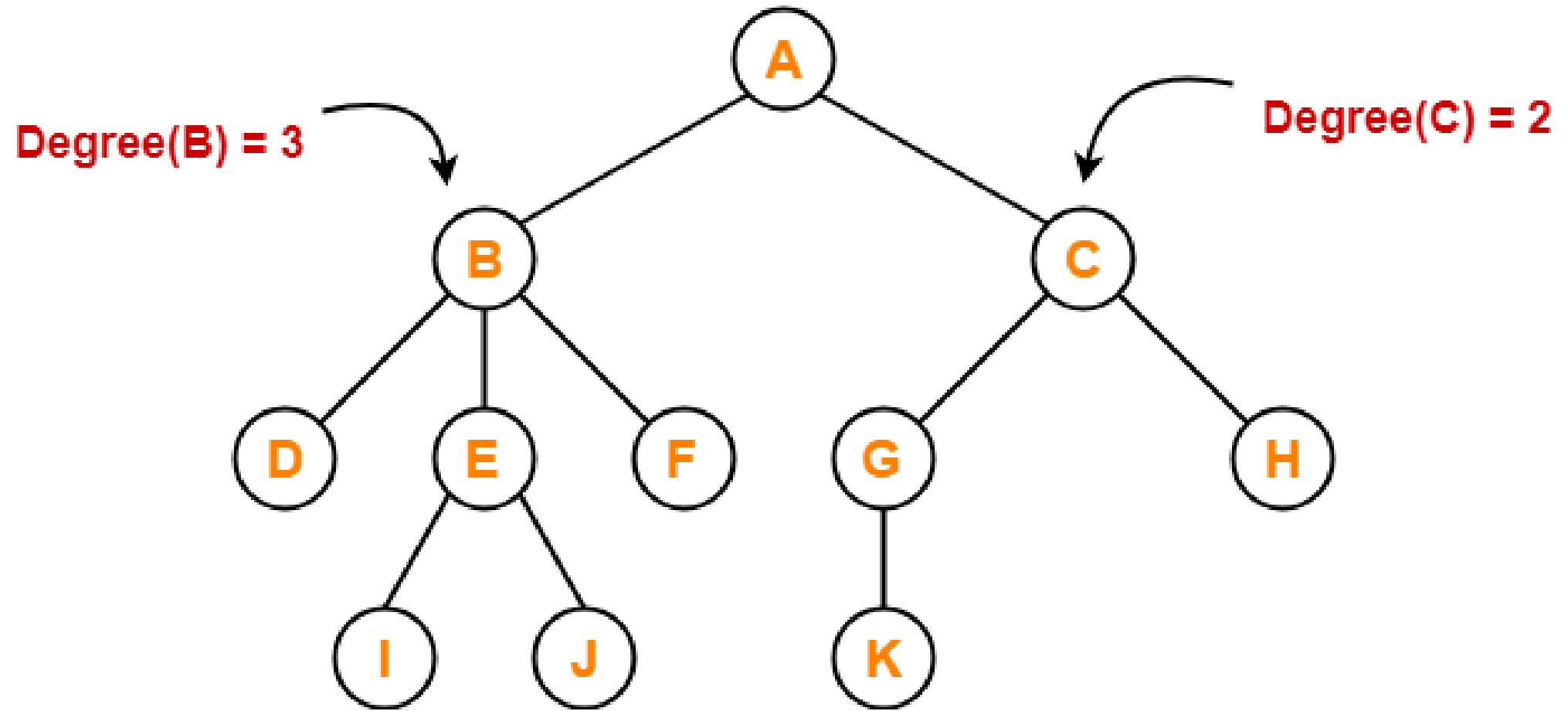


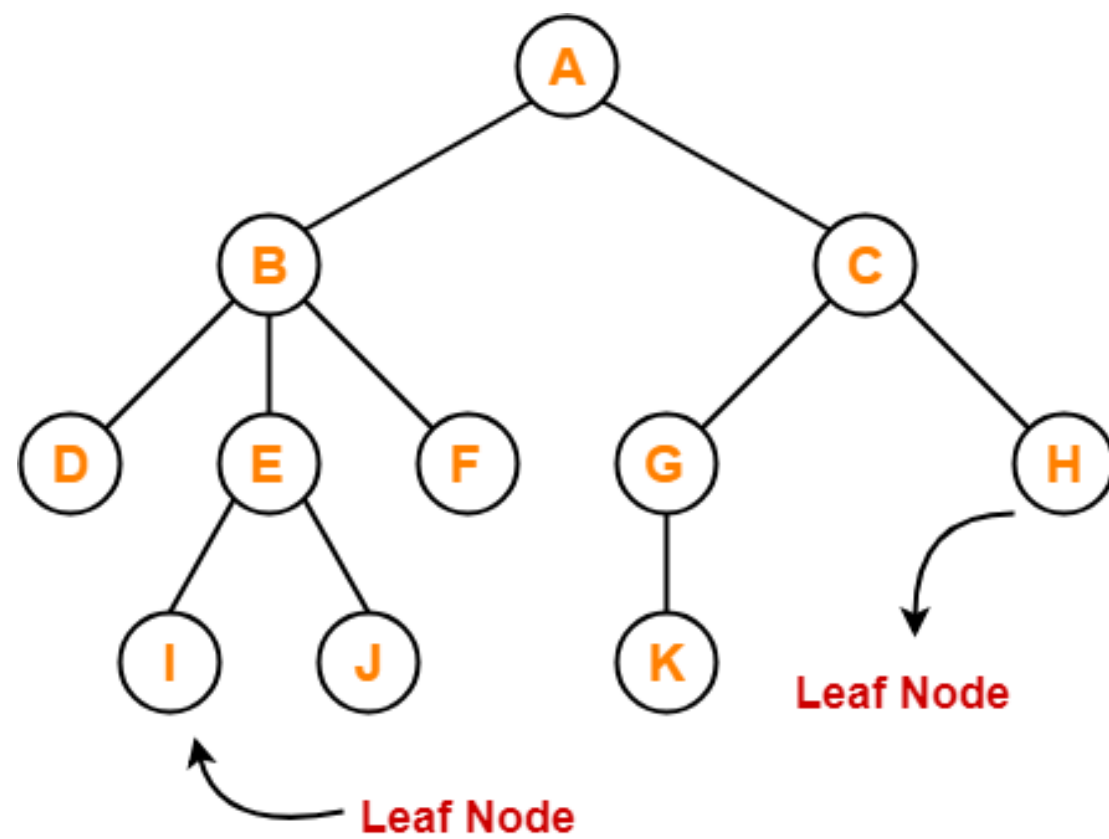
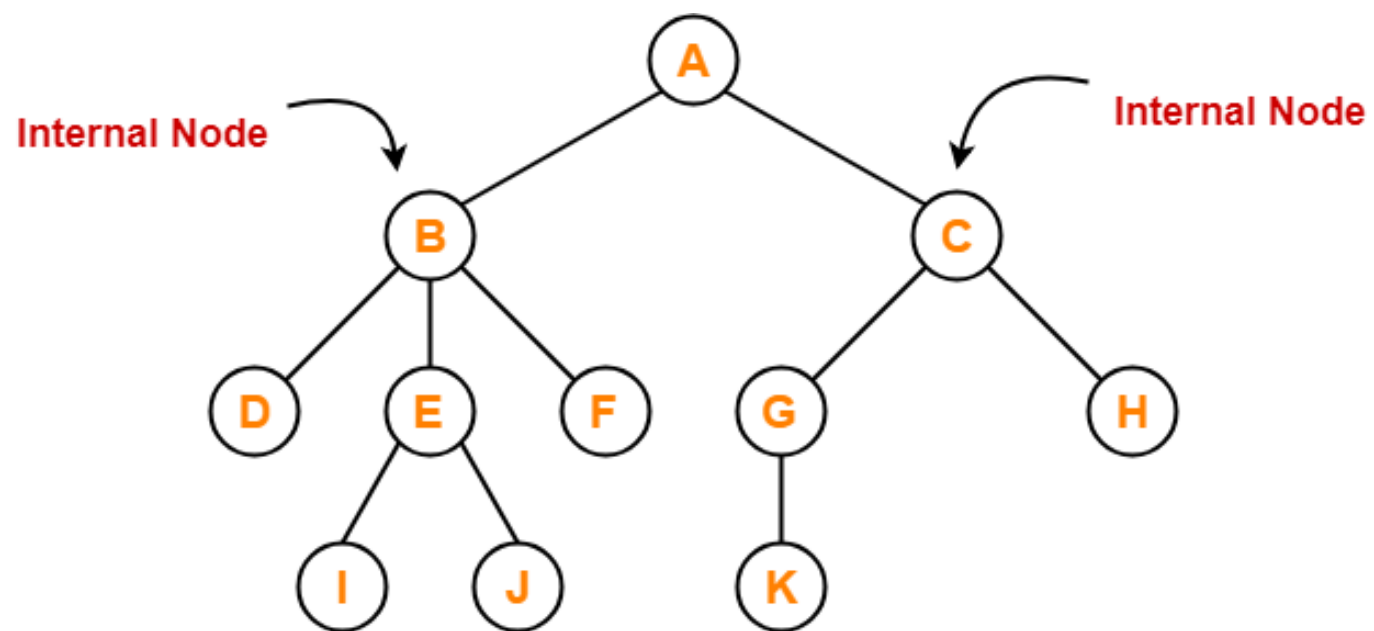


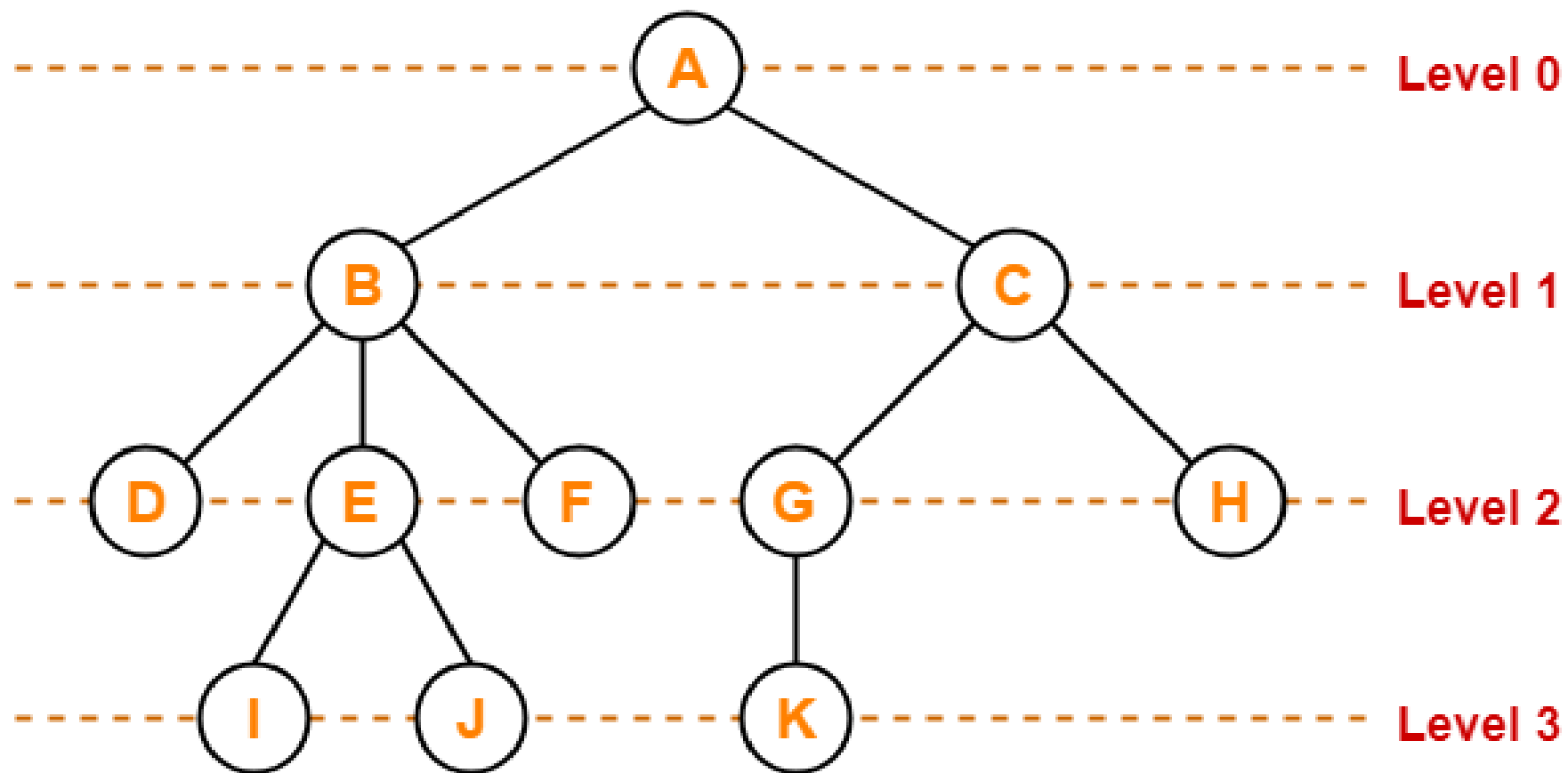


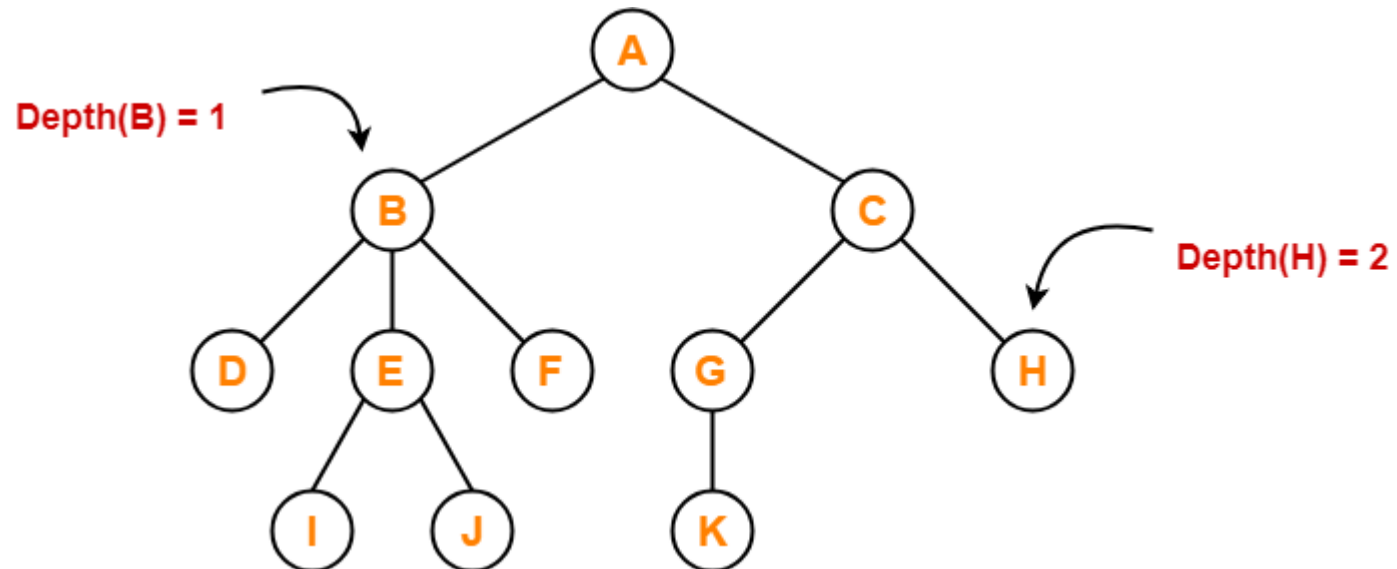
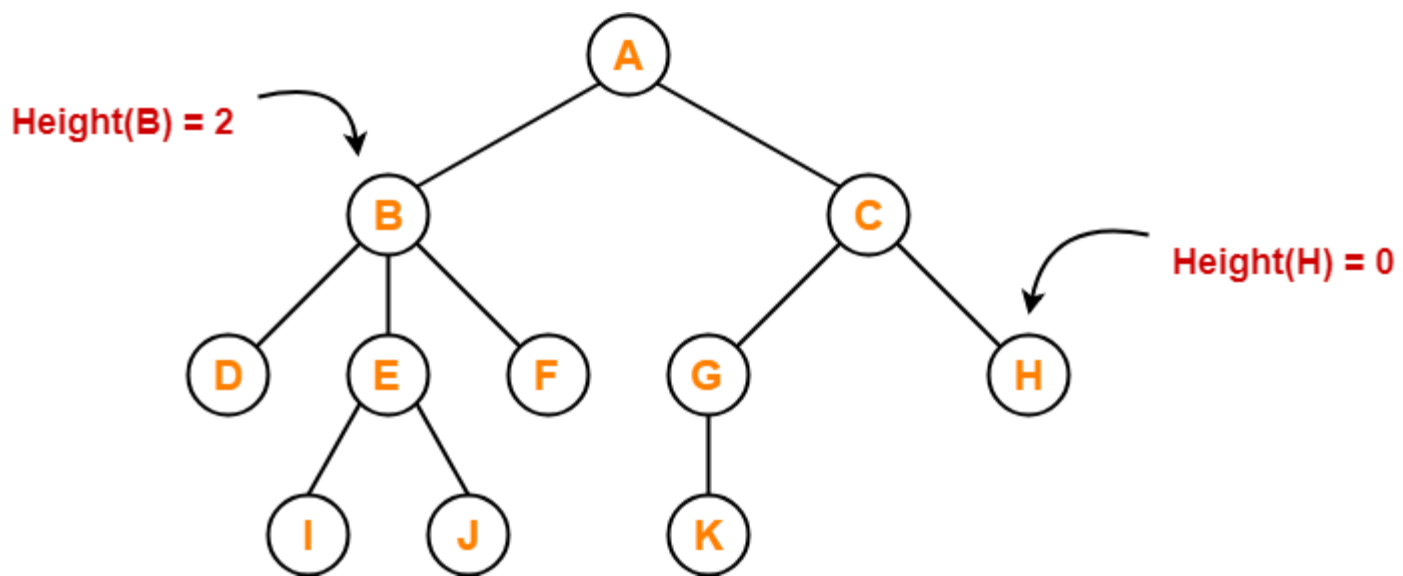


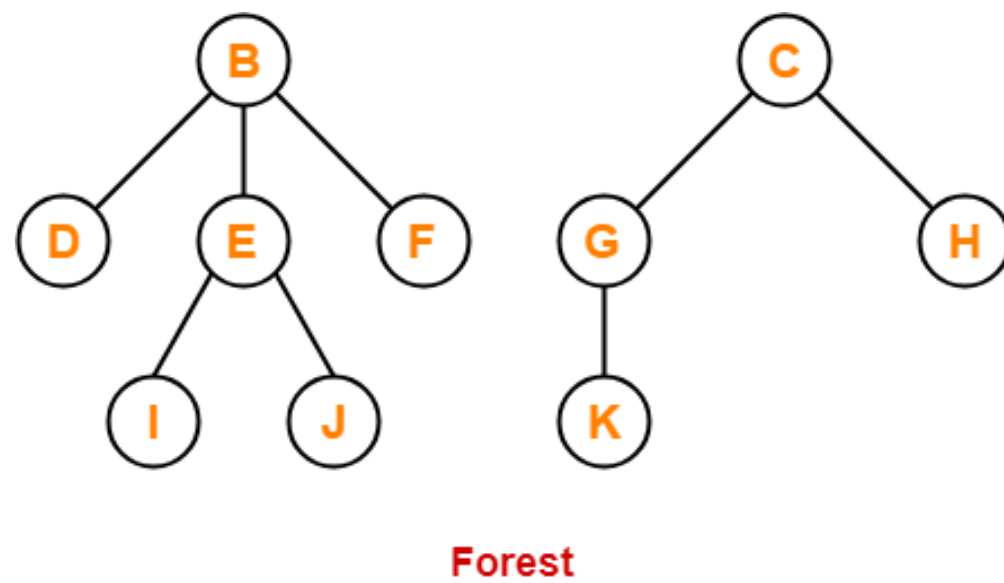
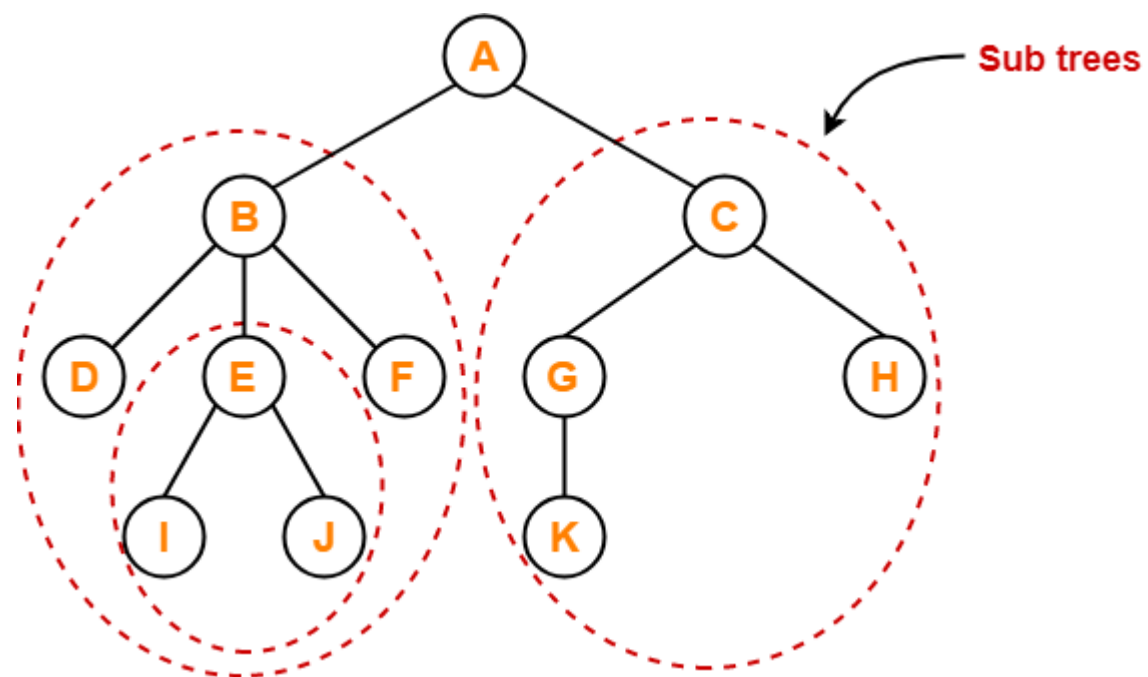












Tree Terminology

- Root:
 - Specially designed data item in the tree. It is the first in the hierarchical arrangement of data items. A is the Root in the above tree.
- Nodes:
 - Each data item in a tree is called a node. It is the basic structure in a tree. It specifies the data information and links(branches) to other data items. There are 13 Nodes in above tree.
- Degree of a node:
 - It is the number of subtrees of a node in a given tree. In above tree,
 - The degree of A is 3
 - The degree of C is 1
 - The degree of B is 2
 - The degree of H is 0
 - The degree of I is 3
- Degree of a Tree
 - It is the maximum degree of nodes in a given tree. In the above tree the node A has degree 3 and another node I is also having its degree 3. In all this value 3 is maximum. So the degree of the above tree is 3.

Tree Terminology

- Terminal Nodes

- A node with degree zero is called a terminal node or a leaf. In the above tree, there are 7 terminal nodes. They are E, J, G, H, K, L and M.

- Non Terminal Nodes

- Any node (except the root node) whose degree is not zero is called non-terminal node. Non terminal nodes are the intermediate nodes in traversing the given tree from its root node to the terminal nodes(leaves). There are 5 non-terminal nodes.

- Siblings

- The children nodes of a given parent node are called siblings. They are also called brothers. In the above tree,
 - E and G are siblings of parent node B.
 - K L and M are siblings of parent node I.

- Edge

- It is a connecting line of two nodes. That is the line drawn from one node to another node is called an edge.

Tree Terminology

- Level
 - The entire tree structure is levelled in such a way that the root node is always at level 0. Than its immediate children are at level 1 and their immediate children are at level 2 and so on upto the terminal nodes. In general, If a node is at level n , then its children will be at level $n+1$.
- Path
 - It is sequence of consecutive edges from the source node to the destination node. In the above tree, the path between A and J is given by the node pairs,
 - (A,B), (B,F) and (F, J)
- Depth
 - It is the maximum level of any node in a given tree. In the above tree, the root node A has the maximum level(3). That is the number of levels one can descend the tree from its root to the terminal nodes(leaves). The term height is also used to denote the depth.
- Forest
 - It is a set of disjoint trees. In a given tree, if you remove its root node then it becomes a forest.

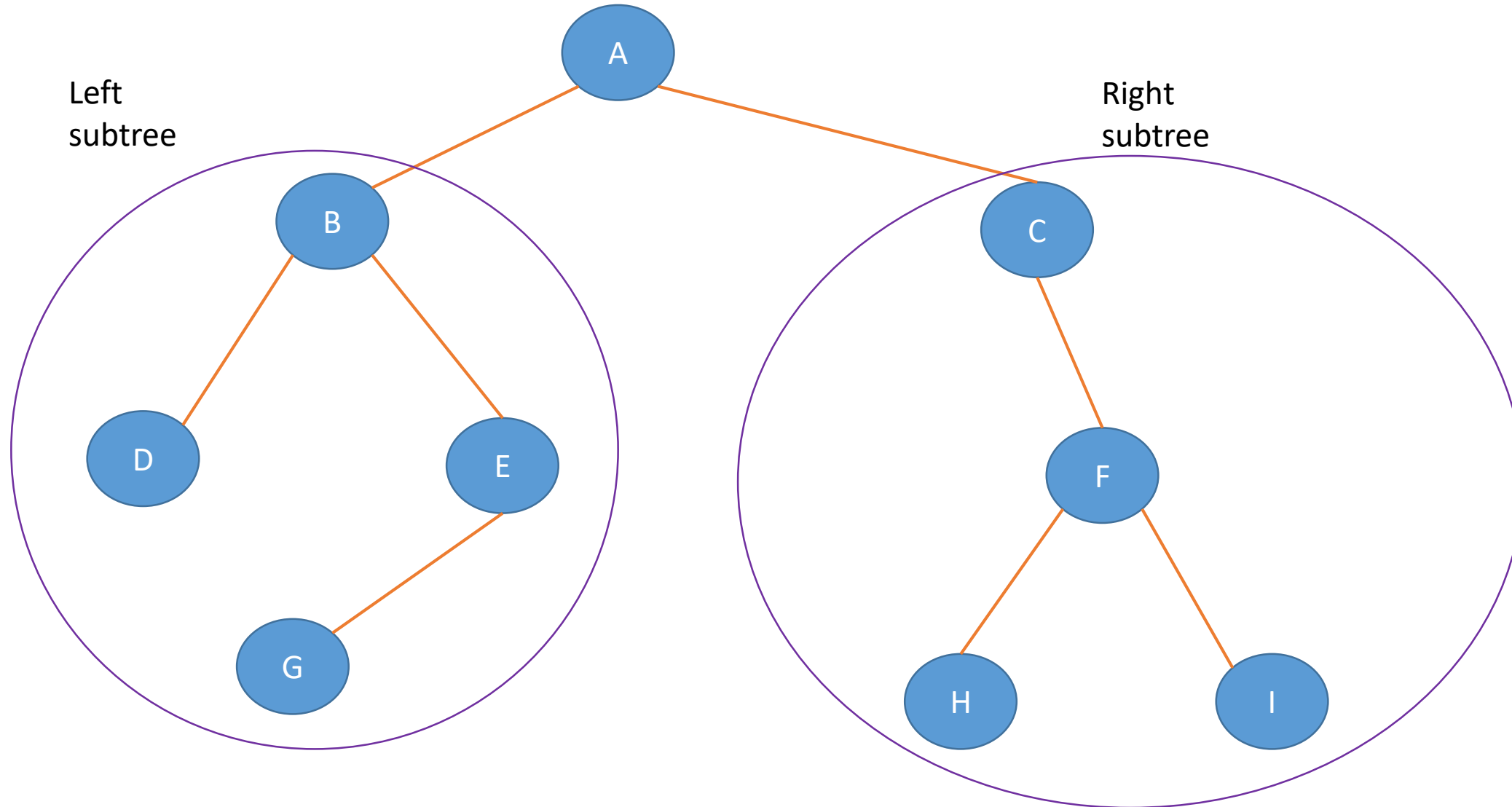
Binary Tree

- A binary tree is **finite set of data items** that is **either empty or** is partitioned into **three disjoint subsets**. The first subset contains a single element called **the root** of the tree. The other two subsets are **themselves binary trees** called the **left and right subtrees** of the original tree.

Binary Tree

Left
subtree

Right
subtree



Binary tree properties:

- If a binary tree contains m nodes at level l , it contains at most $2m$ nodes at level $l+1$.
- Since a binary tree can contain at most 1 node at level 0 (the root), it contains at most 2^l nodes at level l .

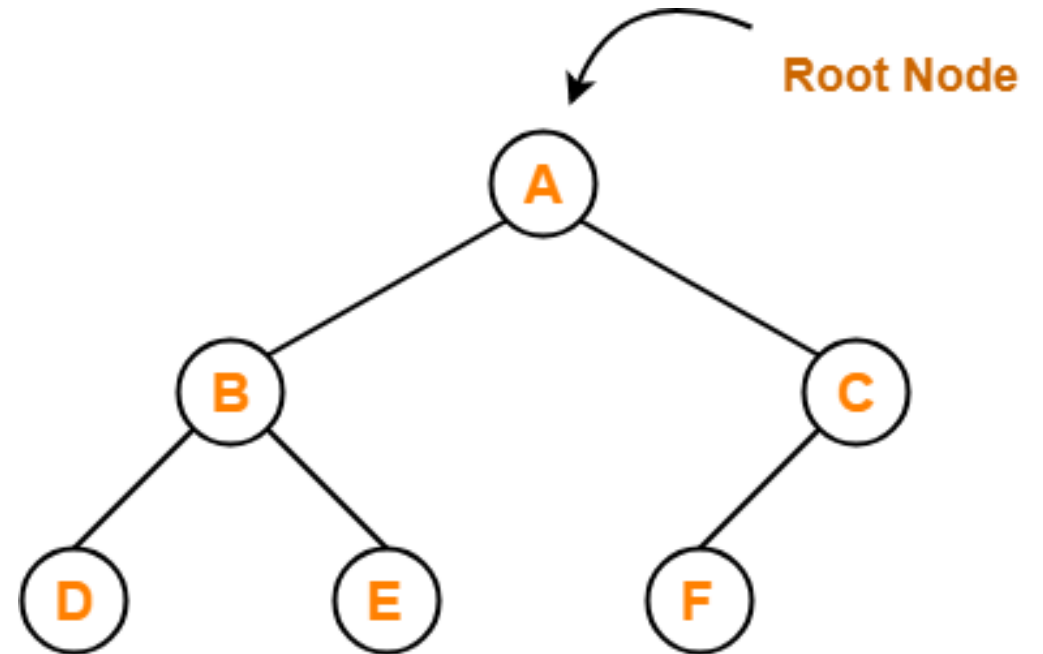
Types of binary tree

- Rooted Binary Tree
- Full / Strictly Binary Tree
- Complete / Perfect Binary Tree
- Almost Complete Binary Tree
- Skewed Binary Tree

1. Rooted Binary Tree-

Satisfies the following 2 properties-

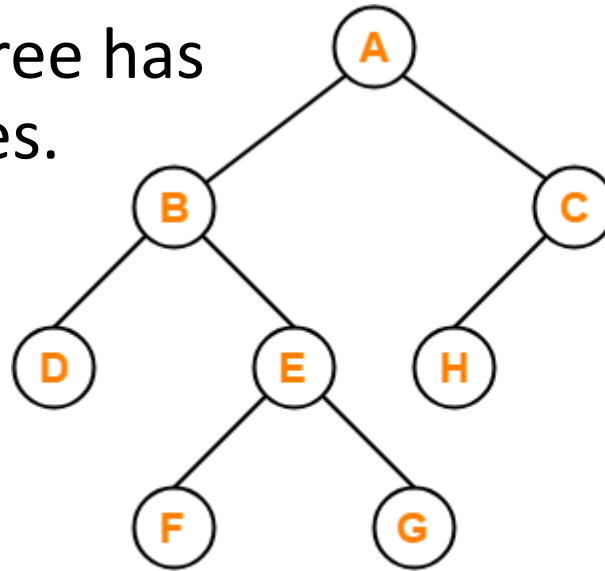
- It has a root node.
- Each node has at most 2 children.



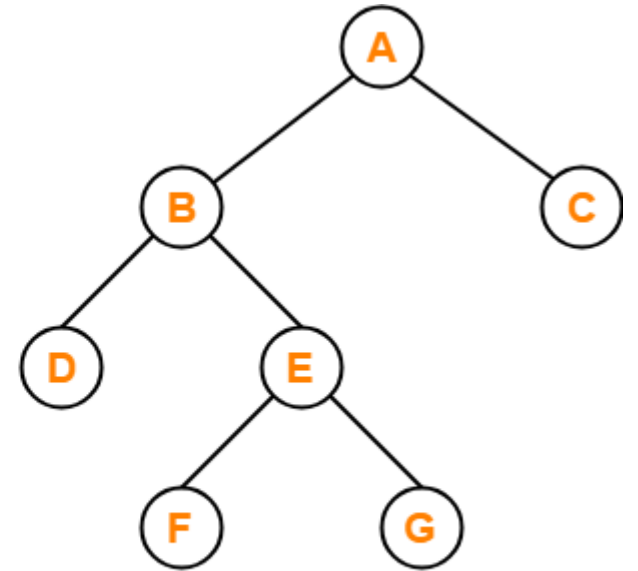
Rooted Binary Tree

2. Full / Strictly Binary Tree-

- A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.
- Every non leaf node in a binary tree has non empty left and right sub trees.



X

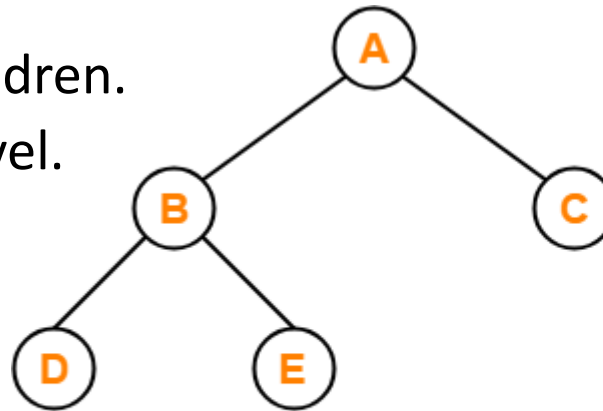


✓

3. Complete / Perfect Binary Tree-

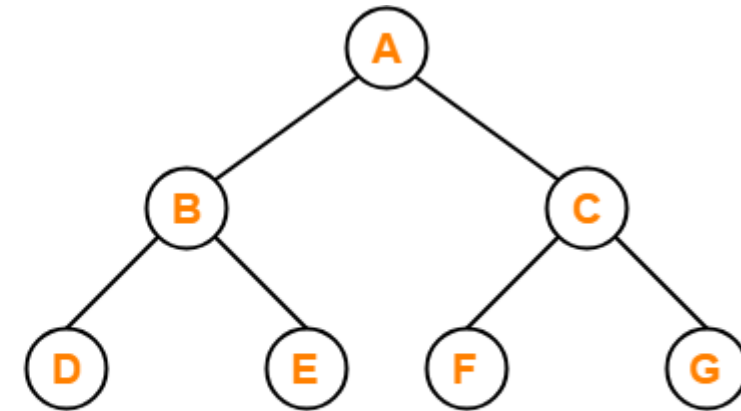
- A **complete binary tree** is a binary tree that satisfies the following 2 properties-
- Every internal node has exactly 2 children.
- All the leaf nodes are at the same level.

Or



- A complete binary tree of depth d is strictly binary tree **if all of whose leaves are at level d .**
- A complete binary tree with depth d has 2^d **leaves** and $2^d - 1$ **non-leaf** nodes(internal)

X

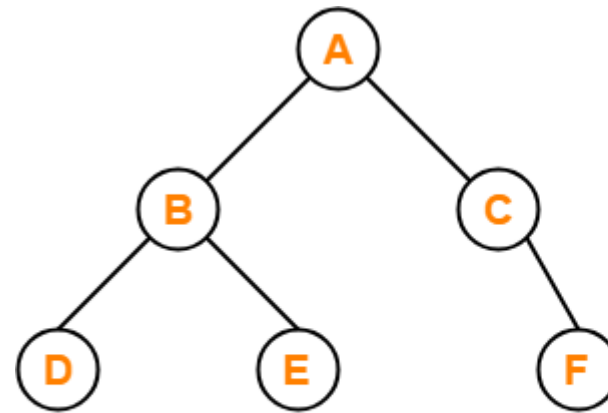


✓

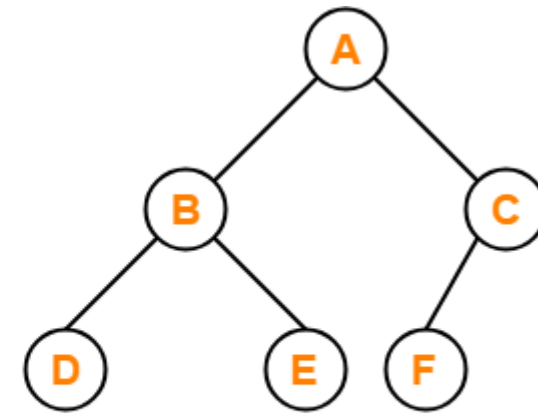
4. Almost Complete Binary Tree-

Satisfies the following 2 properties-

- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.



X



✓

Almost Complete Binary Tree

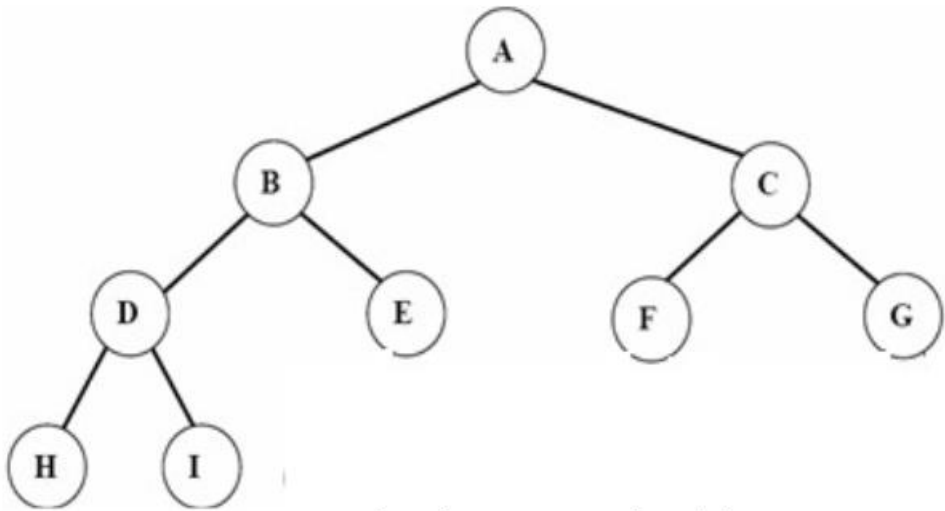


Fig Almost complete binary tree.

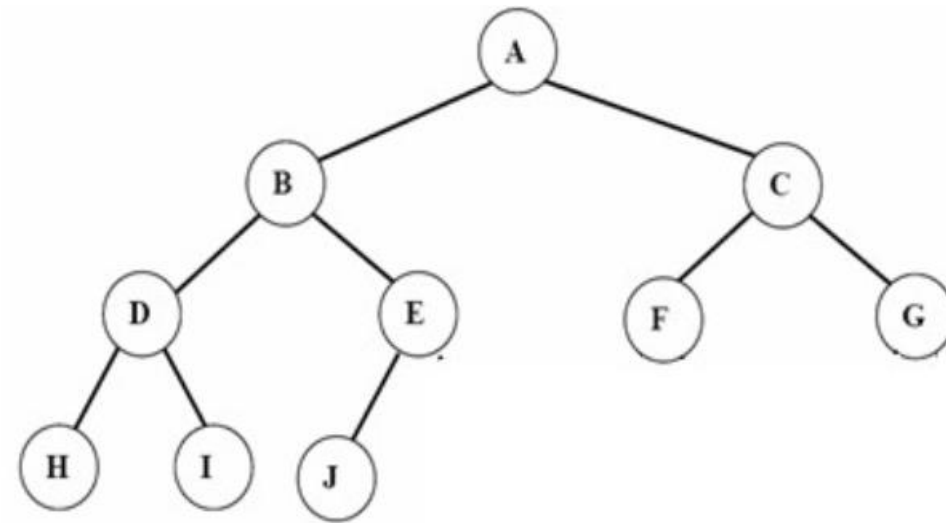
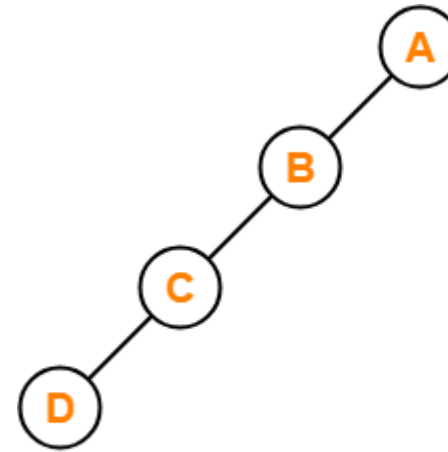
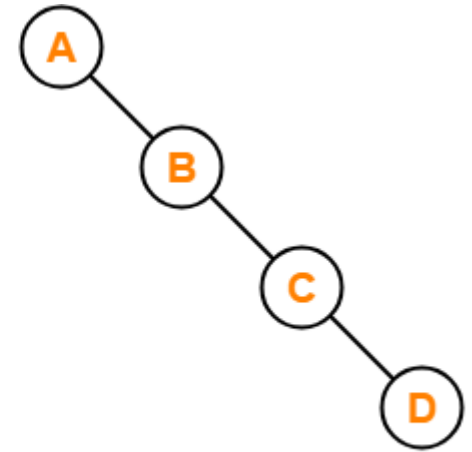


Fig Almost complete binary tree but not strictly binary tree.
Since node E has a left son but not a right son.

5. Skewed Binary Tree-



Left Skewed Binary Tree



Right Skewed Binary Tree

- Satisfies the following 2 properties-
 - All the nodes except one node has one and only one child.
 - The remaining node has no child.
- **OR**
- A **skewed binary tree** is a binary tree of n nodes such that its depth is $(n-1)$.

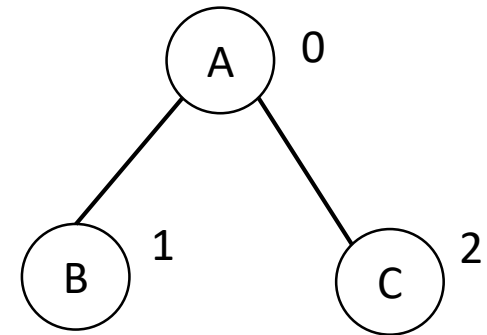
Operations on Binary tree:

- **Father(n,T):** Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- **LeftChild(n,T):** Return the left child of node n in tree T. Return NULL if n does not have a left child.
- **RightChild(n,T) :**Return the right child of node n in tree T. Return NULL if n does not have a right child.
- **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- **Sibling(n,T):** return the sibling node of node n in tree T. Return NULL if n has no sibling.
- **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- **Size(T):** Return the number of nodes in tree T
- **MakeEmpty(T):** Create an empty tree T
- **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- **Preorder(T):** Traverses all the nodes of tree T in preorder.
- **postorder(T):** Traverses all the nodes of tree T in postorder
- **Inorder(T):** Traverses all the nodes of tree T in inorder.

Array Representation of Binary Tree

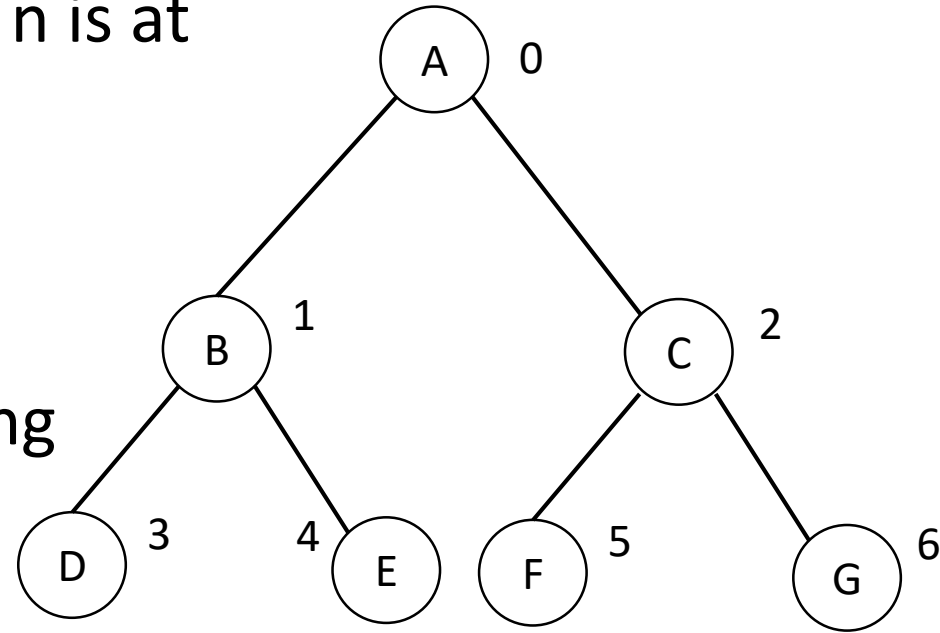
- An array can be used to store nodes of the tree.
- The nodes stored in an array can be accessed sequentially.
- In C , array start with index 0 to Maxsize-1.

A	BT(0)
B	BT(1)
C	BT(2)



How to identify Father, the left child and the right child of an arbitrary node?

- For any node n , ($0 \leq n \leq \text{Maxsize} - 1$) then we have
 - 1. $\text{Father}(n)$. The father of node having index n is at $\text{floor}(n-1)/2$ if n is not equal to 0.
 - 2. $\text{LChild}(n)$: $2n+1$
 - 3. $\text{RChild}(n)$: $2n+2$
 - 4. Siblings: If left is given by n then right sibling is given by $n+1$ else $n-1$.



Linked list representation for Binary tree:

```
struct bnode
{
    Int info;
    struct bnode *left;
    struct bnode *right;
};
struct bnode *root=NULL;
```

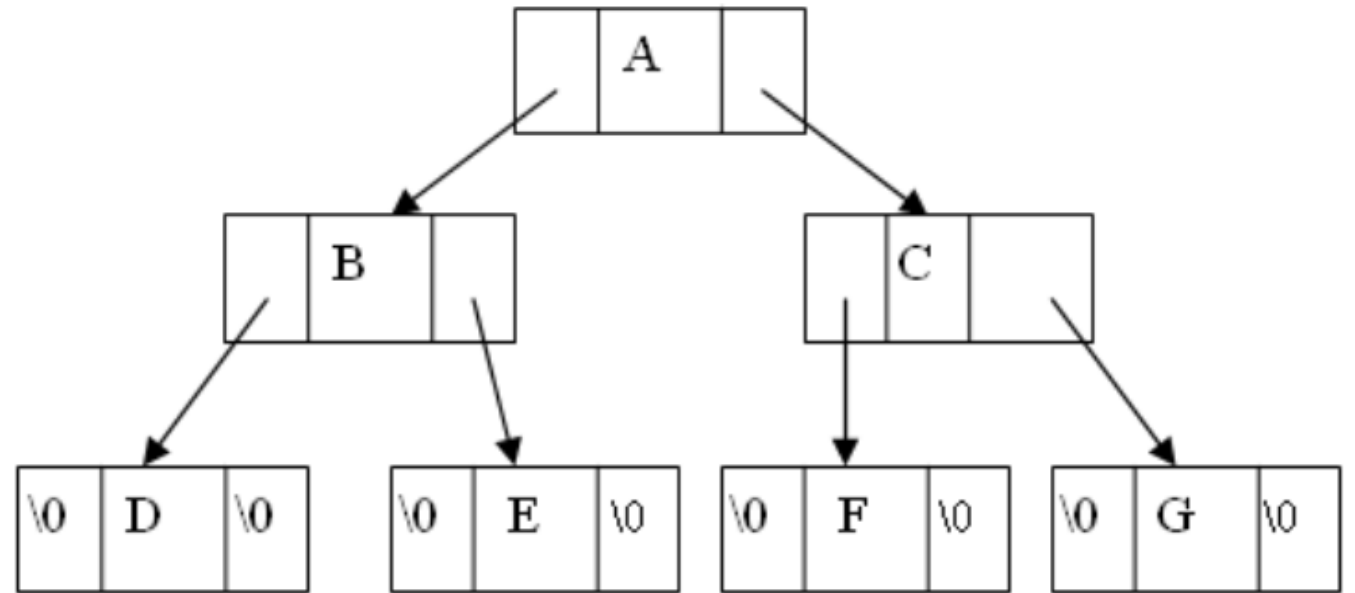


Fig: Structure of Binary tree

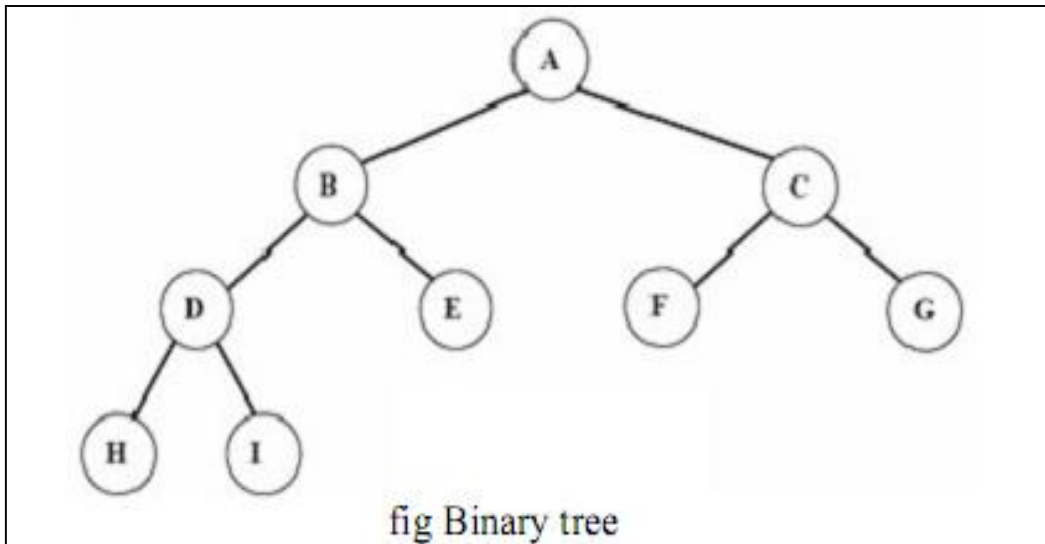
Tree traversal

- The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner.
- There are three popular methods of traversal
 - i. Pre-order traversal
 - ii. In-order traversal
 - iii. Post-order traversal

1. Pre-order traversal:

The preorder traversal of a nonempty binary tree is defined as follows:

- i. Visit the root node
- ii. Traverse the left sub-tree in preorder
- iii. Traverse the right sub-tree in preorder



The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

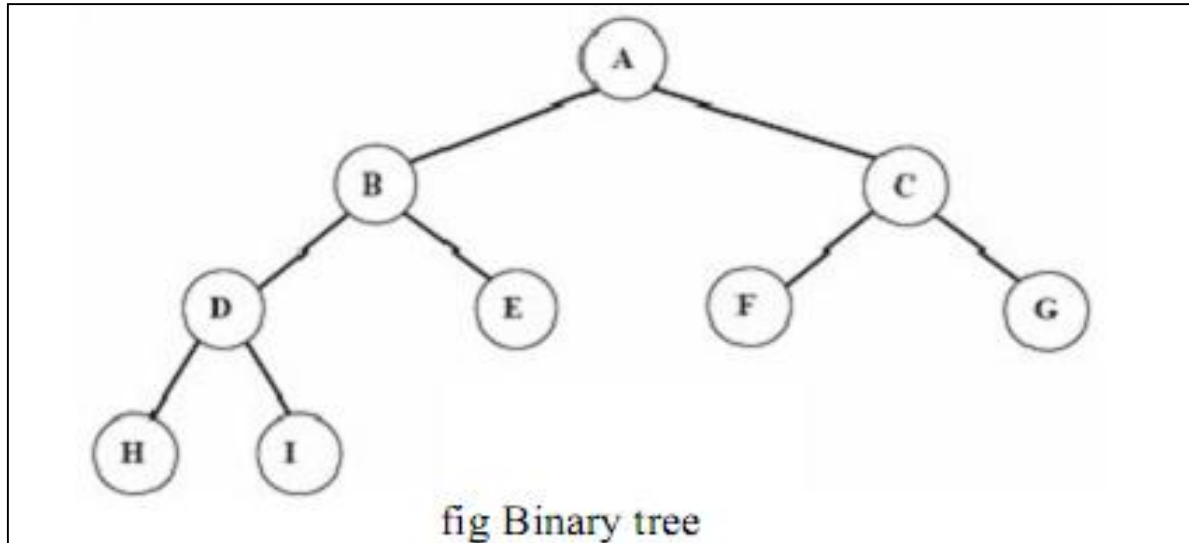
C function for preorder traversing:

```
void preorder(struct bnode *root)
{
    if(root!=NULL)
    {
        printf("%c", root->info);
        preorder(root->left);
        preorder(root->right);
    }
}
```

2. In-order traversal:

The inorder traversal of a nonempty binary tree is defined as follows:

- i. Traverse the left sub-tree in inorder
- ii. Visit the root node
- iii. Traverse the right sub-tree in inorder



C function for inorder traversing:

```
void inorder(struct binaryTreenode *root)
{
    if(root!=NULL)
    {
        inorder(root->left);
        printf("%c", root->info);
        inorder(root->right);
    }
}
```

The inorder traversal output of the given tree is: H D I B E A F C
G

3. Post-order traversal:

The post-order traversal of a nonempty binary tree is defined as follows:

- i. Traverse the left sub-tree in post-order
- ii. Traverse the right sub-tree in post-order
- iii. Visit the root node

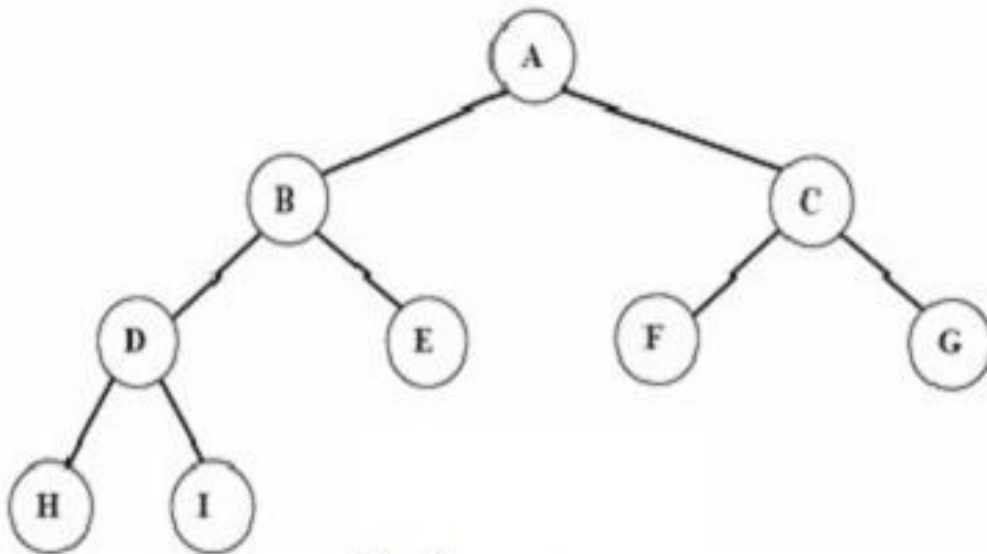


fig Binary tree

C function for post-order traversing:

```
void post-order(struct bnode *root)
{
    if(root!=NULL)
    {
        post-order(root->left);
        post-order(root->right);
        printf("%c", root->info);
    }
}
```

The post-order traversal output of the given tree is: H I D E B F G
C A

Q. Design a binary tree whose traversal sequences are given below:

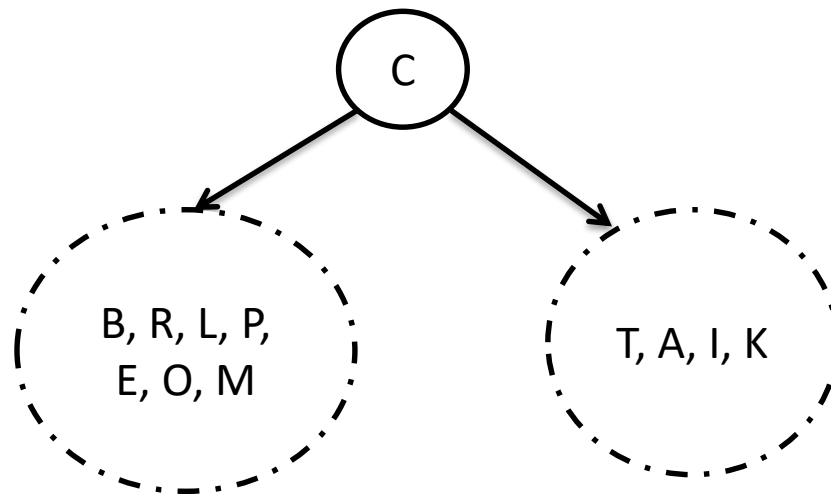
Pre-order: CPRBLOEMATIK

In-Order: BRLPEOMCTAIK

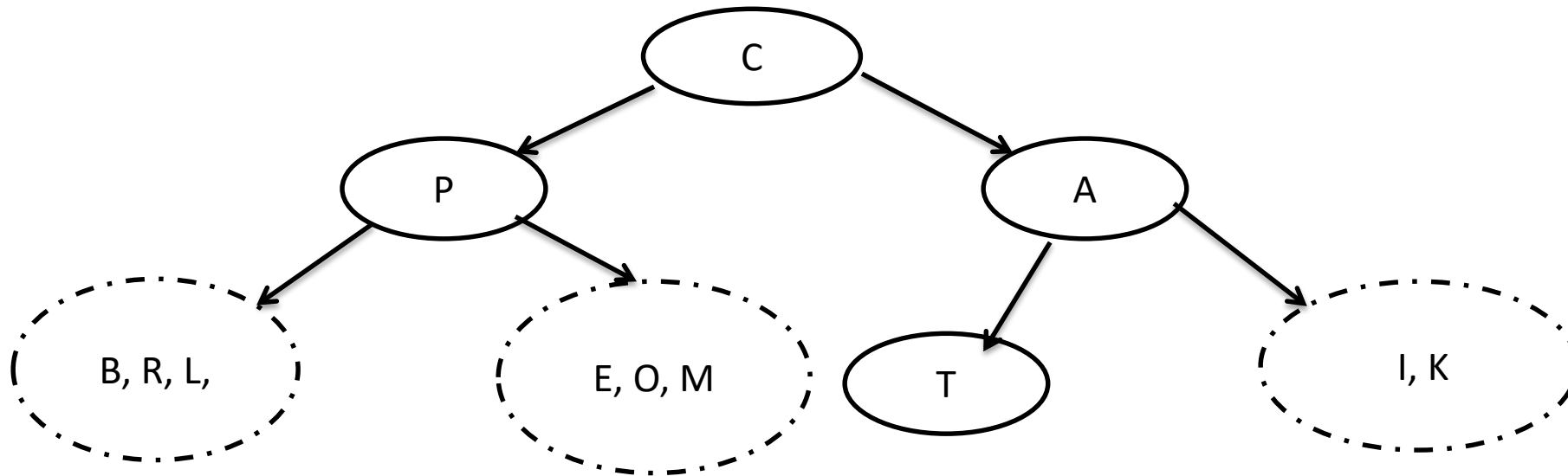
Solution:

Since preorder traverses from root of the tree, C is root of the tree.

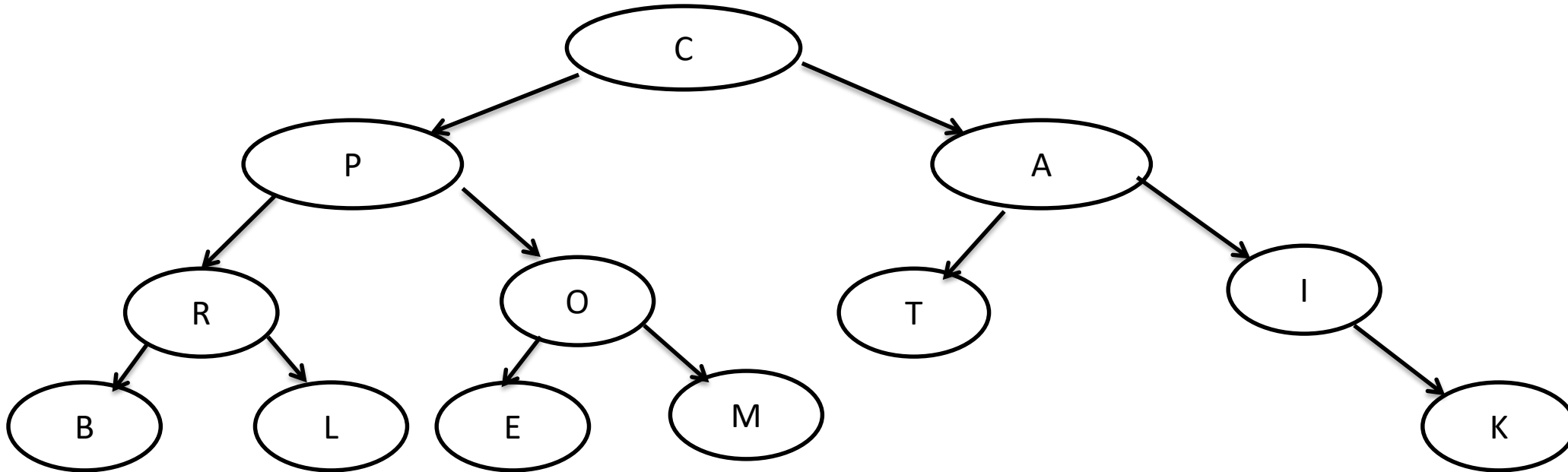
The { B, R, L, P, E, O, M } and { T, I, A, K } are respectively left and right sub tree . (from inorder sequence)



Similarly, for left sub tree { B, R, L, P, E, O, M}, root is P (*from preorder sequence*)
and {B, R, L} is left sub-tree and {E, O, M} is right sub-tree. (*From inorder sequence*)
For right sub-tree {T, A, I, K}, A is the root. (*from preorder sequence*)
T is left sub-tree and {I, K } is right sub-tree.



Similarly, R is root for sub-tree {B, R, L} (from preorder sequence)
B is left child and L right child of R (from inorder sequence)
O is root for sub-tree {E, O, M} (from preorder sequence)
E is left child and M is right child of O (from inorder sequence)
In the same way, I is root and K is right child for sub-tree { I, K }



Q. A binary tree T has 12 nodes. The in-order and post-order traversals of T yield the following sequence of nodes:

In-order : VPNAQRSOKBTM

Post-order : VANRQPBKOMTS

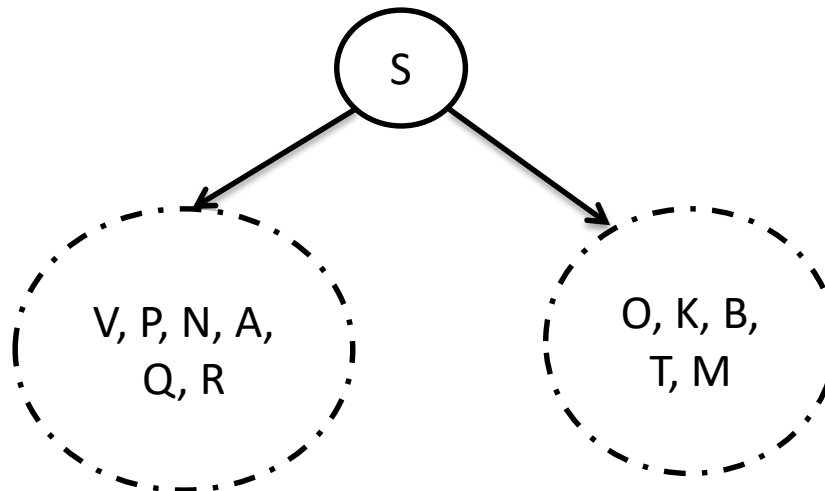
Construct the Binary tree T showing each step. Explain how you can arrive at solution in brief.

Solution:

Since post-order traversal visits all nodes in left sub-tree, right sub-tree and root recursively, the last node in the sequence is root of the tree. That is, S is the root.

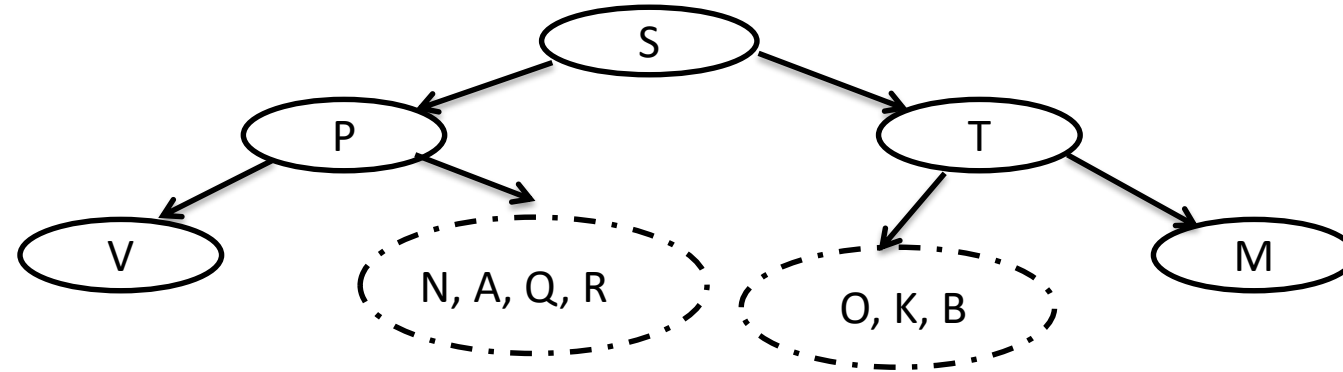
In-order traversal visits the nodes in Left child, root and right child in recursive way, {V,P,N,A,Q,R} is left sub-tree and {O,K,B,T,M} is the right sub tree.

Step -1:

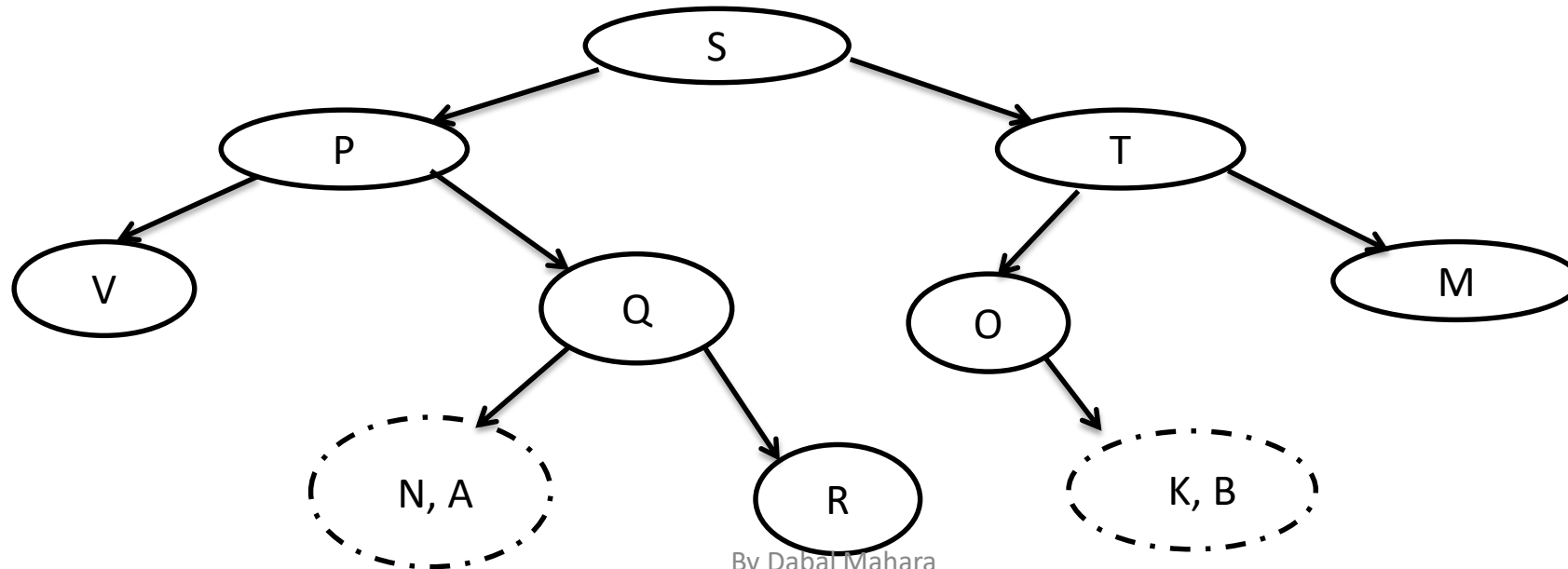


Step -2: The root of { V, P, N, A, Q, R} is P obtained from postorder sequence. Left subtree and right sub trees are {v} and {N, A, Q, R} obtained from inorder sequence.

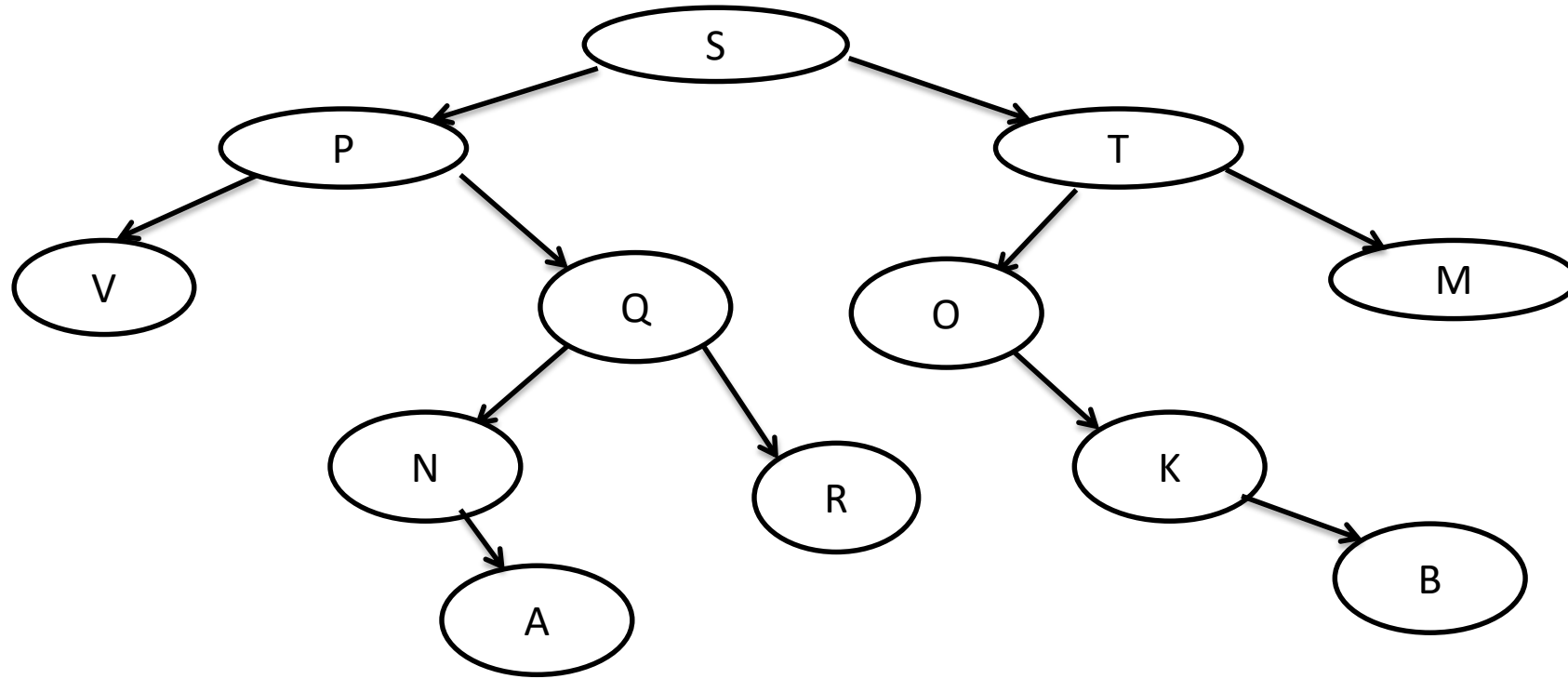
Similarly, for { O, K, T, B, M}, root is T and left sub tree {O, K, B} and right sub tree is {M}.



Step -3: In the same way, root and sub trees are obtained.



Step -4: The final tree is given below.

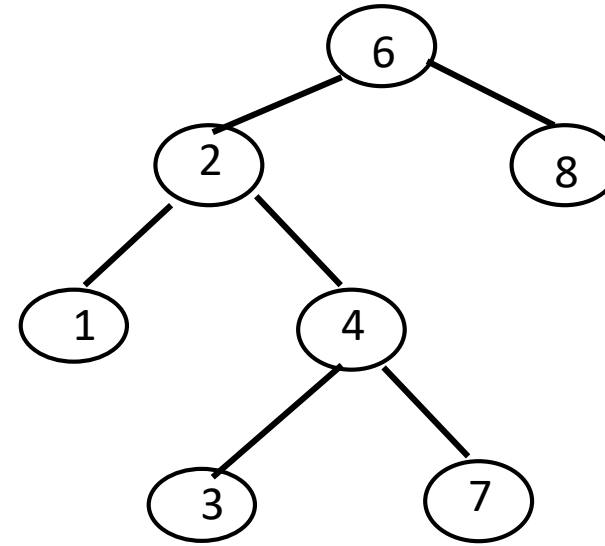
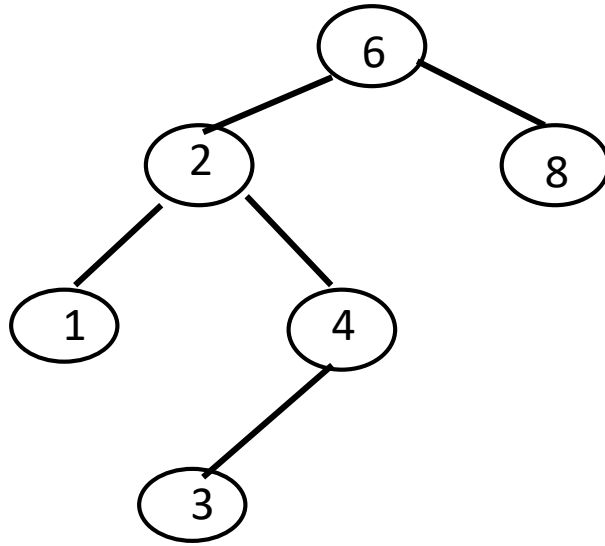


Thus, root of the tree is obtained from post-order sequence scanning from right to left and left and right sub-trees of corresponding root are obtained from inorder sequence.

Binary search tree(BST):

- A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:
 - All keys in the left sub-tree of the root are smaller than the key in the root node.
 - All keys in the right sub-tree of the root are greater than or equal to the key in the root node.
- The left and right sub-trees of the root are again binary search trees.

Binary Search Trees



- The tree on the left is a binary search tree, but the tree on the right is not.

Operations on Binary search tree(BST):

Following operations can be done in BST:

- **Search(k, T):**Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- **Insert(k, T):**Insert a new node with value k in the info field in the tree T such that the **property of BST is maintained**.
- **Delete(k, T):**Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- **FindMin(T), FindMax(T):**Find minimum and maximum element from the given nonempty BST.

Searching through the BST:

- Idea: Compare the target value with the element in the root node.
 - If the target value is equal, the search is successful.
 - If target value is less, search the left subtree.
 - If target value is greater, search the right subtree.
 - If the subtree is empty, the search is unsuccessful.

BST search algorithm

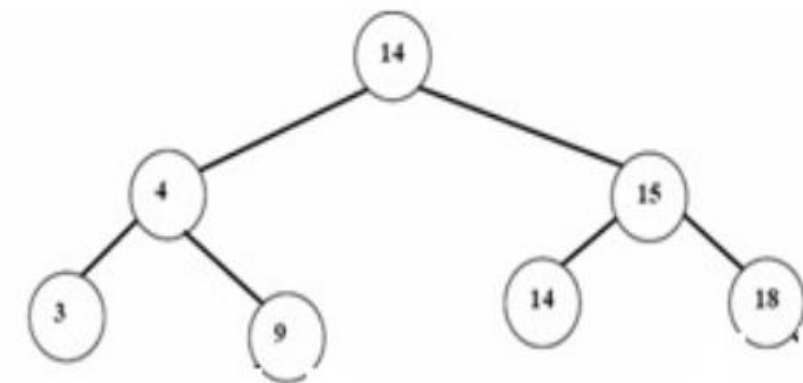
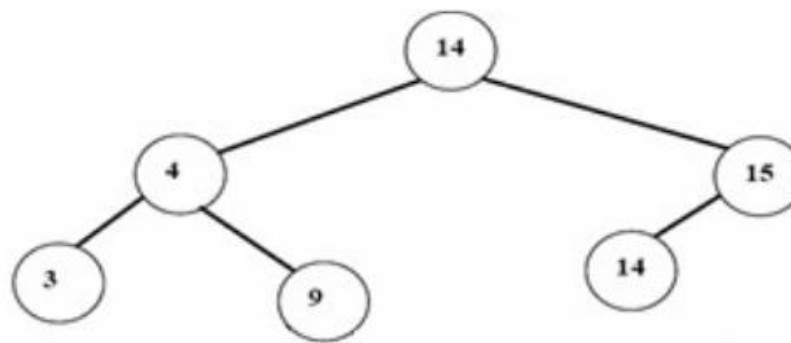
To find which if any node of a BST contains an element equal to target:

1. Set curr to the BST's root.
2. Repeat:
 - 2.1. If Curr is null:
 - 2.1.1. Terminate with answer none.
 - 2.2. Otherwise, if target is equal to curr's element:
 - 2.2.1. Terminate with answer curr.
 - 2.3. Otherwise, if target is less than curr's element:
 - 2.3.1. Set curr to curr's left child.
 - 2.4. Otherwise, if target is greater than curr's element:
 - 2.4.1. Set curr to curr's right child.

C function for BST searching:

```
Void BinSearch(struct bnode *root , int key)
{
    if(root == NULL)
    {
        printf("The number does not exist");
        exit(1);
    }
    else if (key == root->info)
    {
        printf("The searched item is found");
    }
    else if(key < root->info)
        return BinSearch(root->left, key);
    else
        return BinSearch(root->right, key);
}
```

BST insertion algorithm:



insert(18)

- To insert the element into a BST:
 1. set curr to the BST's root
 2. Repeat:
 - 2.1. If curr is null:
 - 2.1.1. Replace the null with elem 's link.
 - 2.1.2. Terminate.
 - 2.2. Otherwise, if elem is equal to curr's element:
 - 2.2.1. Terminate.
 - 2.3. Otherwise, if elem is less than curr's element:
 - 2.3.1 If no left child, then attach elem to left of curr, then terminate.
 - 2.3.1. else set curr to curr's left child
 - 2.4. Otherwise, if elem is greater than curr's element:
 - If no right child, then attach elem to right of curr, then terminate.
 - 2.4.1. set curr to curr's right child.

3. End

C function for BST insertion:

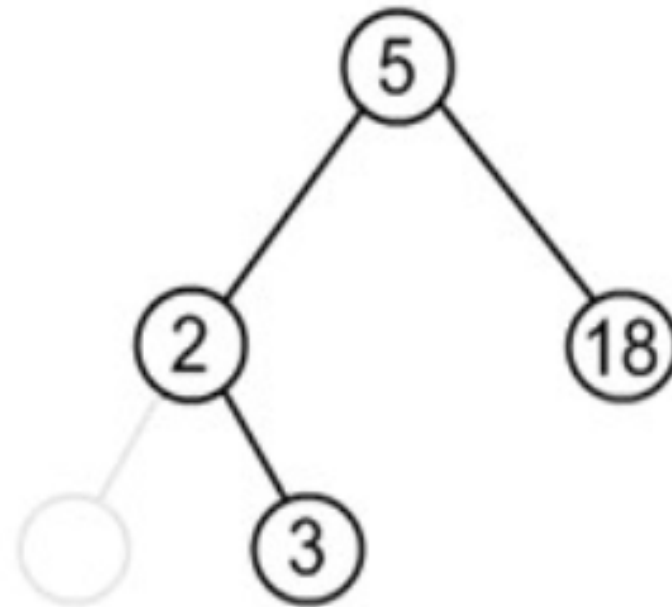
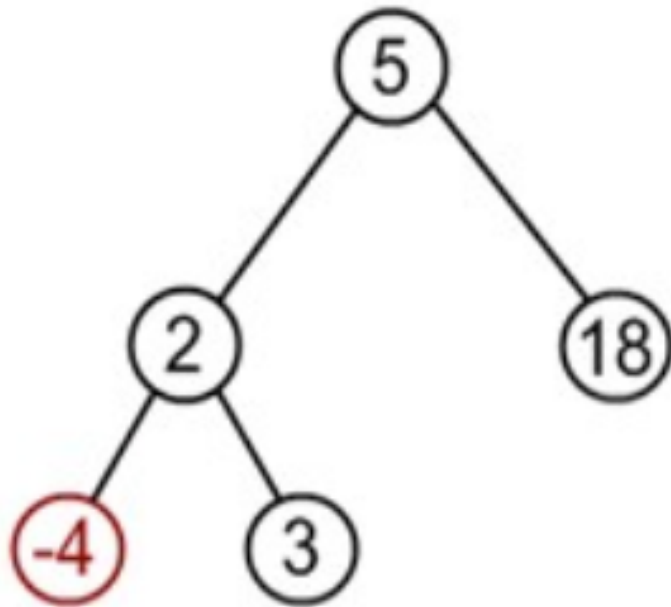
```
void insert(struct bnode *root, int item)
{
    if(root=NULL)
    {
        root=(struct bnode*)malloc (sizeof(struct bnode));
        root->left=root->right=NULL;
        root->info=item;
    }else
    {
        if(item < root->info)
            root->left = insert(root->left, item);
        else
            root->right = insert(root->right, item);
    }
}
```

Deleting a node from the BST:

While deleting a node from BST, there may be three cases:

1. *The node to be deleted may be a leaf node:*

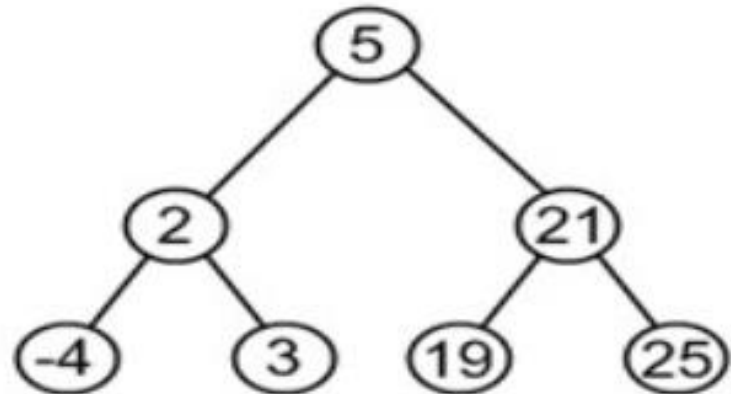
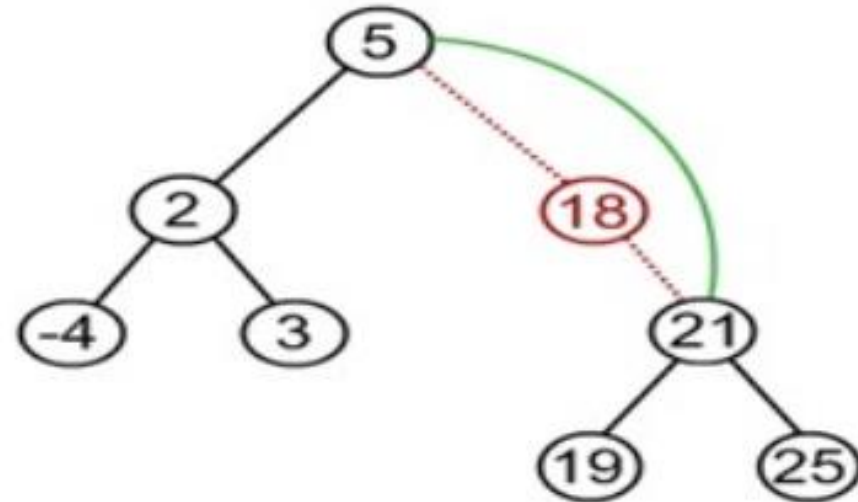
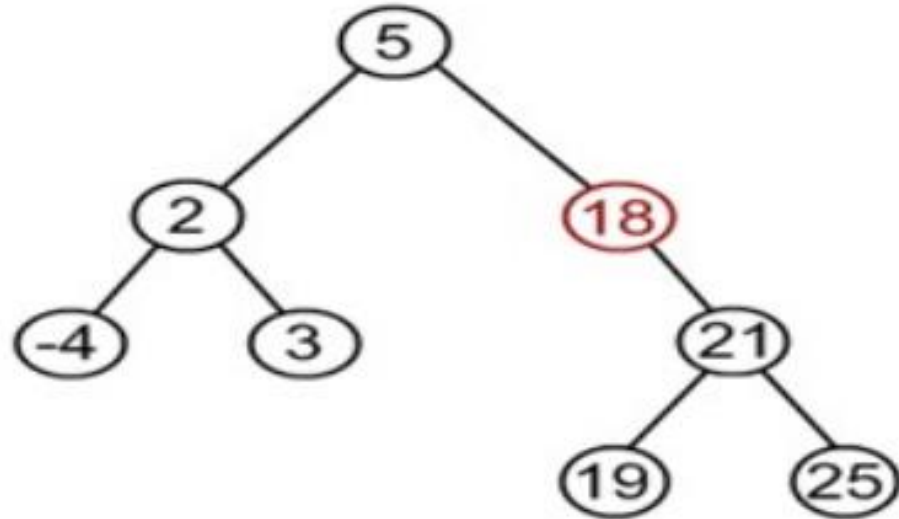
In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.



Suppose node to be deleted is -4

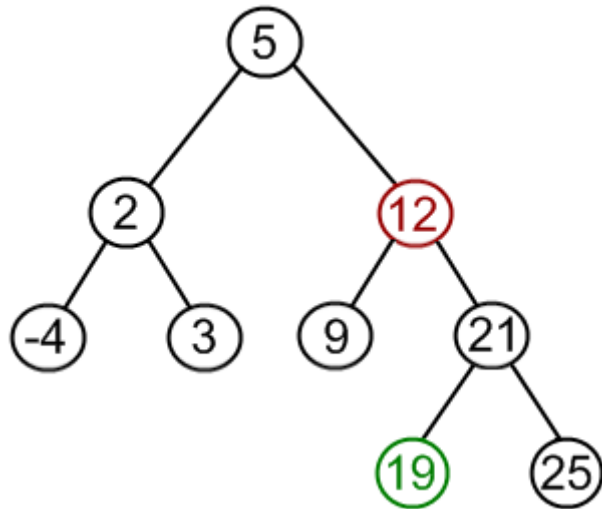
2. The node to be deleted has one child:

In this case the child of the node to be deleted is appended to its parent node.
Suppose node to be deleted is 18

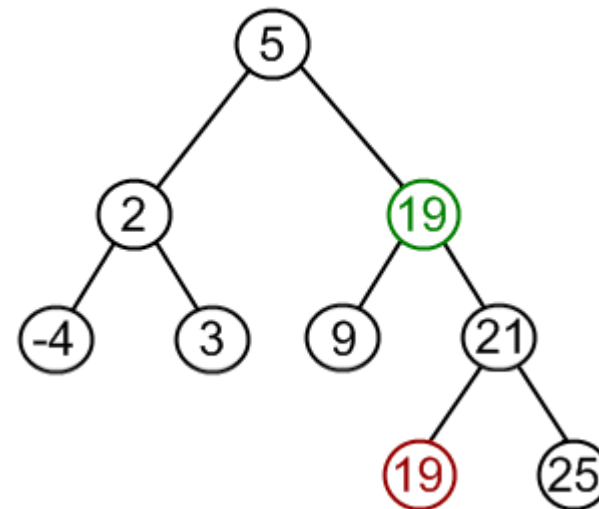


Node to be deleted has two children

- Find minimum element in the **right subtree** of the node to be removed.
- Or remove the node with inorder successor of the deleted node.



- Then remove duplicate from subtree



General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf node at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer.
4. if a node to be deleted has one child then connect it's child pointer with it's parent pointer and delete it from the tree.
5. if a node to be deleted has two children then replace the node being deleted either by
 - a. right most node of it's left sub-tree or
 - b. left most node of it's right sub-tree.
6. End

The delnode function:

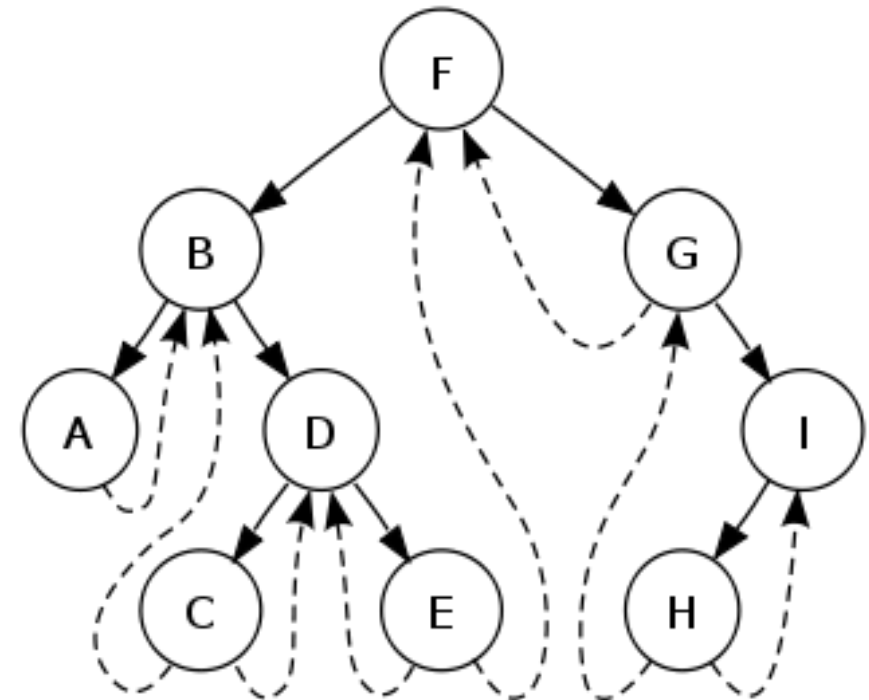
```
struct bstnode *delnode(struct bstnode *root, int item)
{
    struct bstnode *temp;
    if(root==NULL)
    {
        printf("Empty tree");
        return;
    }
    else if(item<root->info)
        root->left=delnode(root->left, item);
    else if(item>root->info)
        root->right=delnode(root->right, item);
    else if(root->left!=NULL && root->right!=NULL) //node has two children
    {
        temp=findmin(root->right);
        root->info=temp->info;
        root->right=delnode(root->right, root->info);
    }
    else
    {
        temp=root;
        if(root->left==NULL)
            root=root->right;
        else if(root->right==NULL)
            root=root->left;
        free(temp);
    }
    return(root);
}
```



```
struct bstnode * findmin(struct bstnode * root)
{
    if(root==NULL)
        return NULL;
    else if(root->left ==NULL)
        return head;
    else
        return(findmin(root->left));
}
```

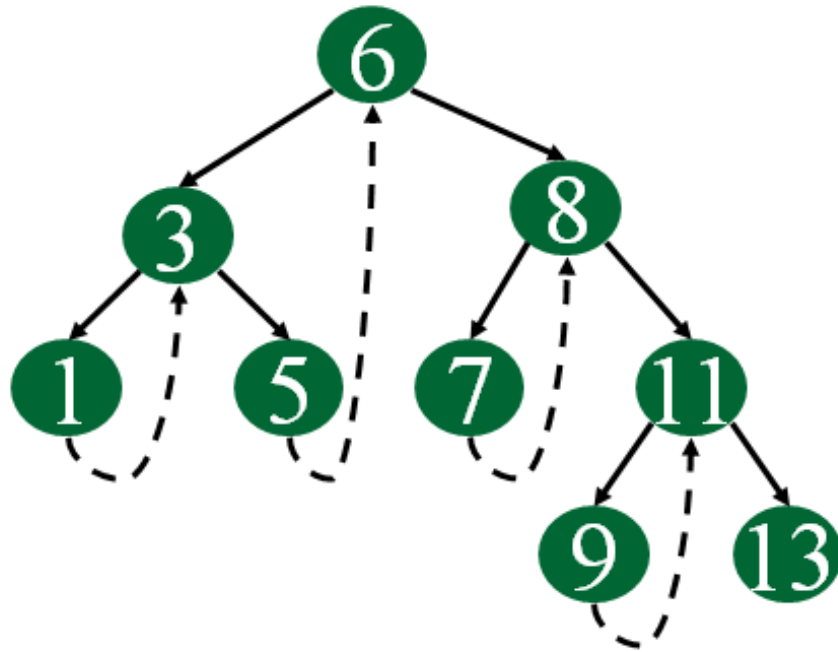
Threaded binary tree

- A binary tree is *threaded* by making all **right child pointers** that would normally be null **point to the inorder successor** of the node (if it exists), and all **left child pointers** that would normally be null point to the **inorder predecessor** of the node.
- ABCDEFGHI

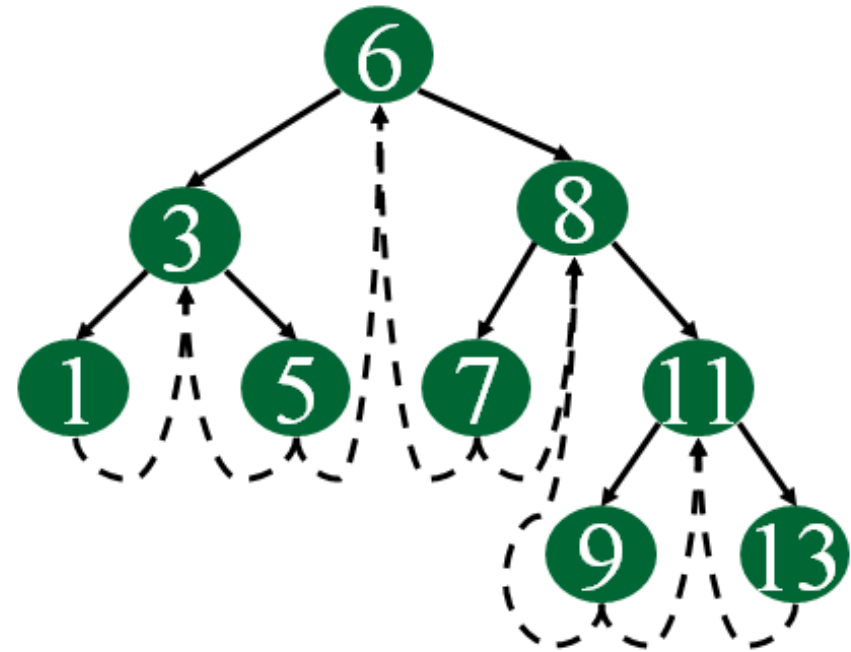


Types of threaded binary trees

- Single Threaded: each node is threaded towards **either** the in-order predecessor **or** successor (left **or** right).
- Double threaded: each node is threaded towards **both** the in-order predecessor **and** successor (left **and** right).



Single Threaded Binary Tree



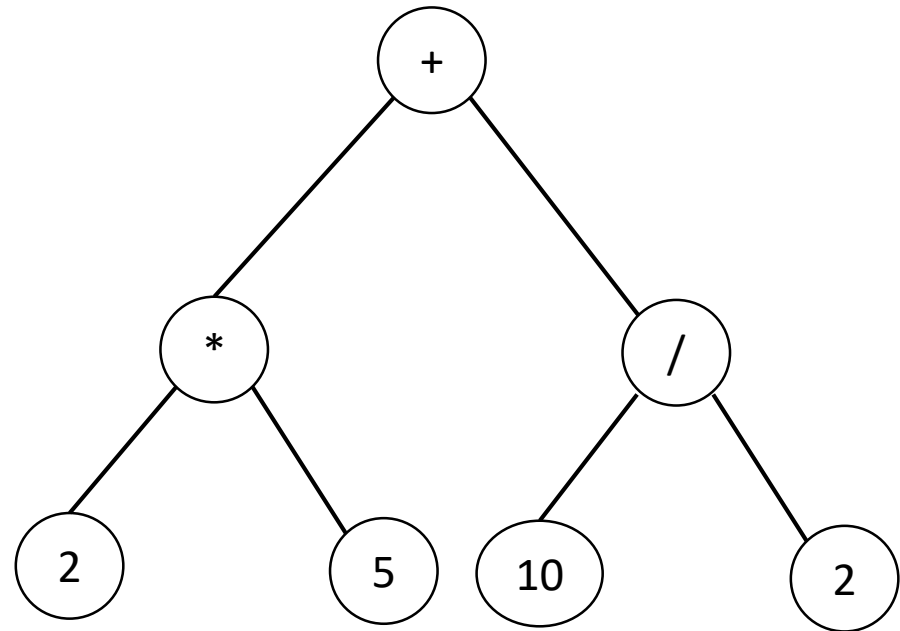
Double Threaded Binary Tree

Why do we need Threaded Binary Tree?

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal

Heterogonous binary tree

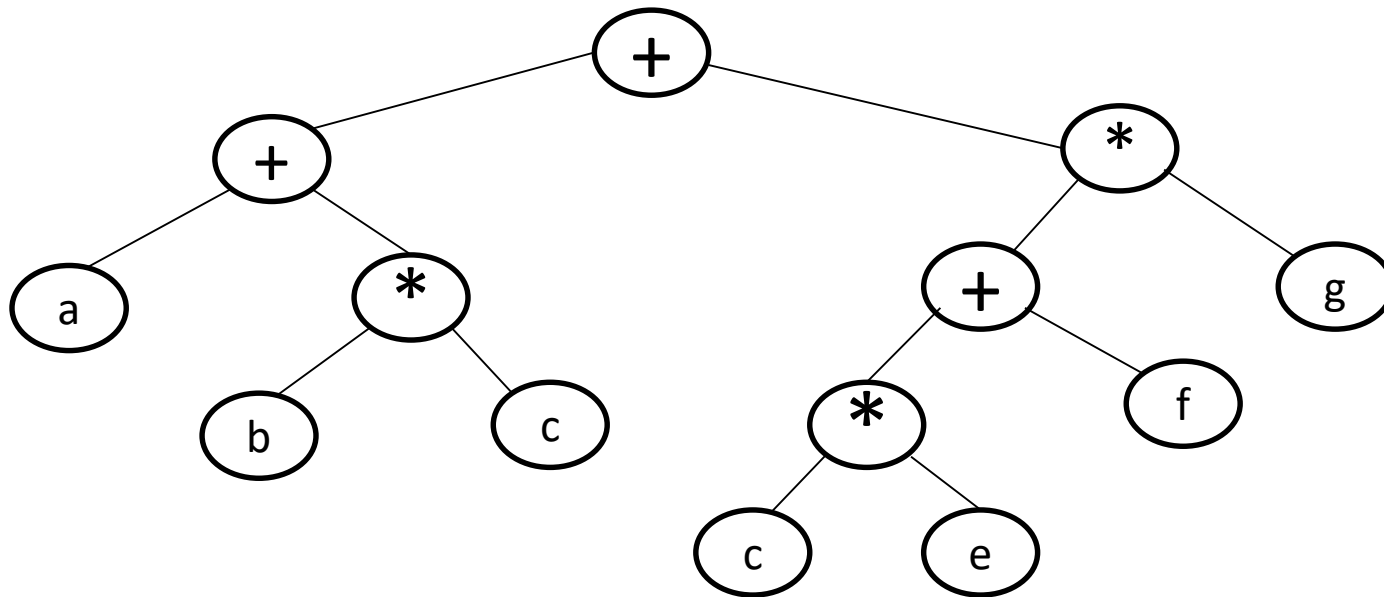
- Often the information contained in different nodes of binary tree is not all of the same type



Expression Tree

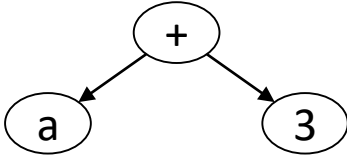
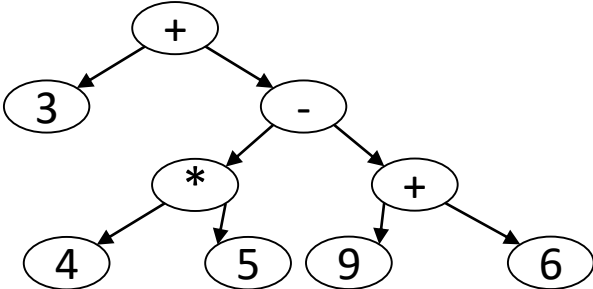
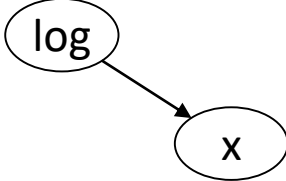
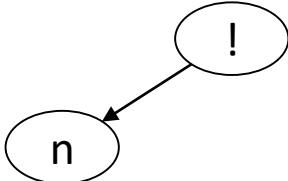
An expression tree is a **strictly binary tree** in which **leaf node contains operands** and **non-leaf node contain operators**. Root nodes contain the operators that is applied to the result of left and right sub trees. The operators, constants, and variables are arranged in such a way that an inorder traversal of the tree produces the original expression without parentheses.

Example: 1 An expression tree of an expression $(a + b * c) + ((c * e + f) * g)$



Exercise: Construct an expression tree of an expression: $(a + b) / ((c - d) * e)$

Expression Tree Examples

Expression	Expression Tree	Inorder Traversal Result
$(a+3)$	 <pre> graph TD A((+)) --> B((a)) A --> C((3)) </pre>	$a + 3$
$3+(4*5-(9+6))$	 <pre> graph TD A((+)) --> B((3)) A --> C((-)) C --> D((*)) C --> E((+)) D --> F((4)) D --> G((5)) E --> H((9)) E --> I((6)) </pre>	$3+4*5-9+6$
$\log(x)$	 <pre> graph TD A((log)) --> B((x)) </pre>	$\log x$
$n!$	 <pre> graph TD A((!)) --> B((n)) </pre>	$n !$

Constructing an Expression Tree From Postfix Expression

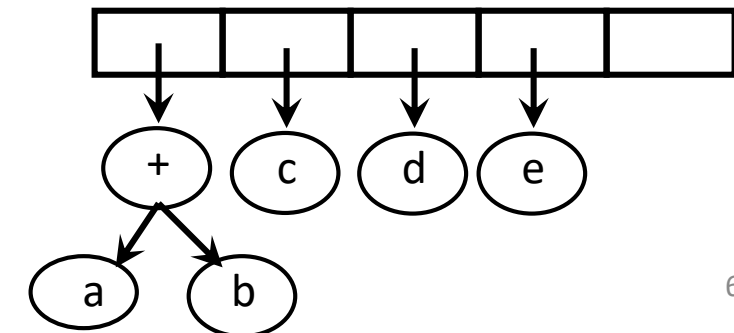
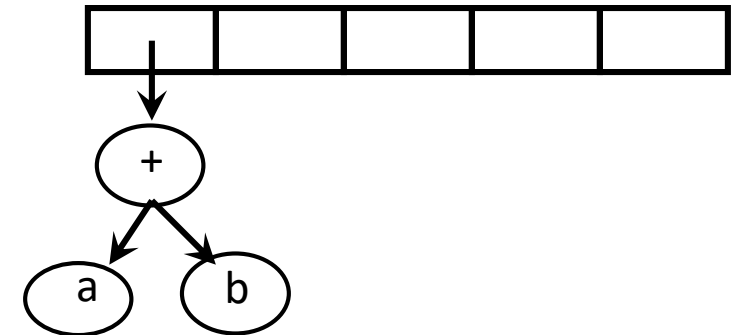
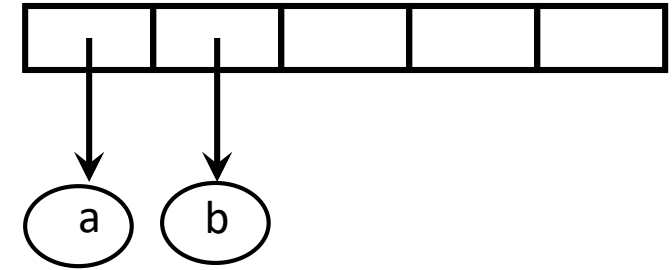
Algorithm

- We read the expression one symbol at a time.
- If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.
- If the symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 , respectively. A pointer to this new tree is then pushed onto the stack.

Example:

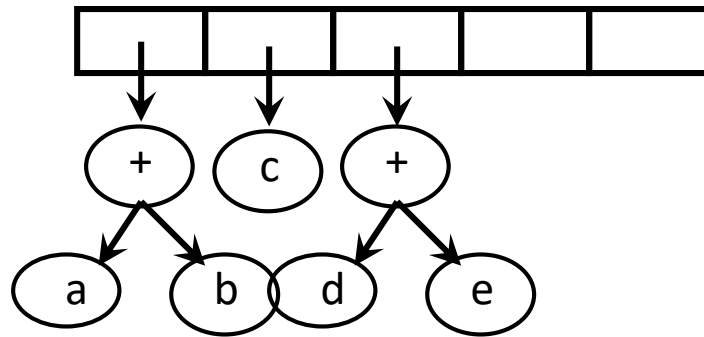
- Input: $ab+cde+**$

- The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack. For convenience, we will have the stack grow from left to right in the diagrams.
- Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.
- Next, c , d , and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.

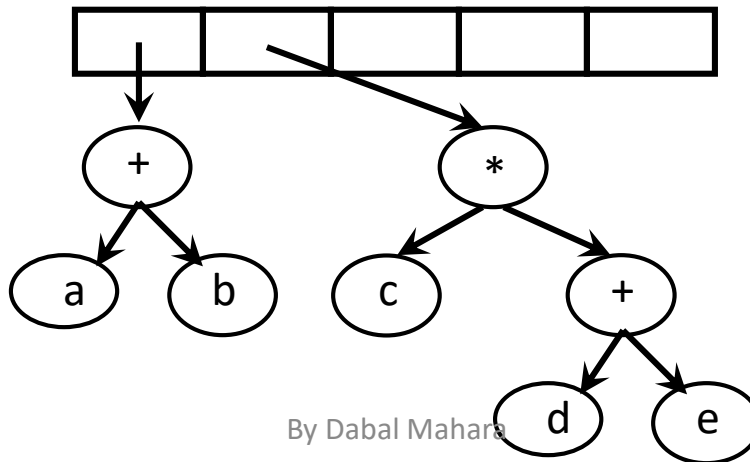


Constructing an Expression Tree

- Now a '+' is read, so two trees are merged.

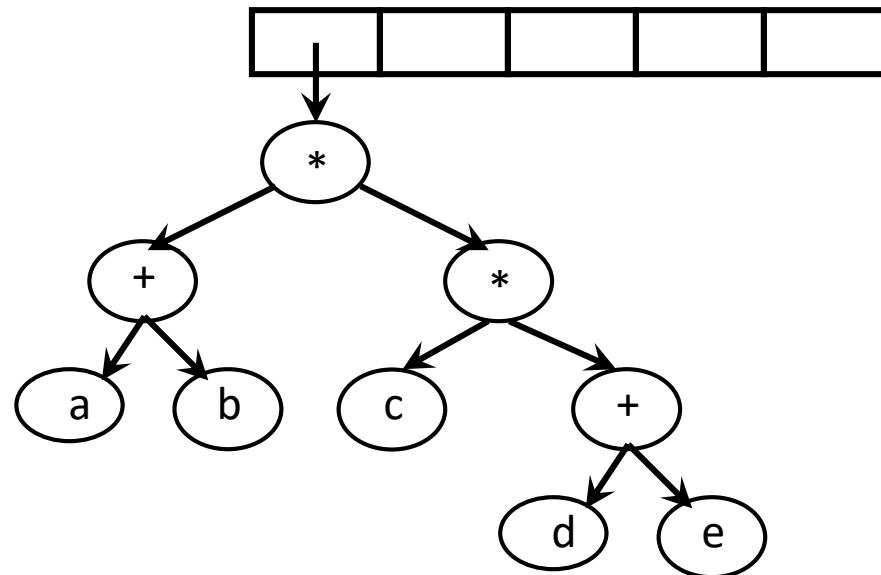


- Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Constructing an Expression Tree

- Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



Exercise: Construct an expression from postfix expression:
FG+\$

AB+C*DC--

The Huffman Coding

- Proposed by Dr. David A. Huffman at MIT in 1952
 - “*A Method for the Construction of Minimum Redundancy Codes*”
- Huffman coding is a form of statistical coding
 - more frequently used symbols have shorter code words
 - Not all characters occur with the same frequency!
 - Yet all characters are allocated the same amount of space
 - 1 char = 1 byte, be it e or x
- Huffman coding is a technique used to compress files for transmission
 - Works well for text and fax transmissions

Huffman Code

- **Huffman codes** is text compression method, which relies on the **relative frequency** (i.e., the number of occurrences of a symbol) with which different symbols appear in a text
- Uses extended binary trees
- Variable-length codes that satisfy the property, where **no code is a prefix of another**
- **Huffman tree** is a binary tree with minimum weighted external path length for a given set of frequencies (weights).

Huffman Algorithm

1. Initially, each symbol is one node tree by itself. That is, there is forest of n nodes each labelled with symbol and weight.
2. Repeat while $n > 1$
 - i. Select the two trees T_1 and T_2 with minimum weights from the forest.
 - ii. Merge T_1 and T_2 to form T_3 with weight sum of both trees.
 - iii. Insert T_3 into the forest.
3. Assign 0 to each left path of tree and 1 to right path. The code for each symbol is obtained by collecting bits on the paths from root to the leaf node labelled with the symbol.

Example: 1

Let us take any four characters and their frequencies as follows:

Char	frequency
E	10
T	7
O	5
A	3

- Without using Huffman coding, these four characters can be represented by using two bits as: 00 -> E, 01 -> T, 10 -> O and 11 -> A. This is fixed length coding. So, bits required to encode all the characters in the above message will be:

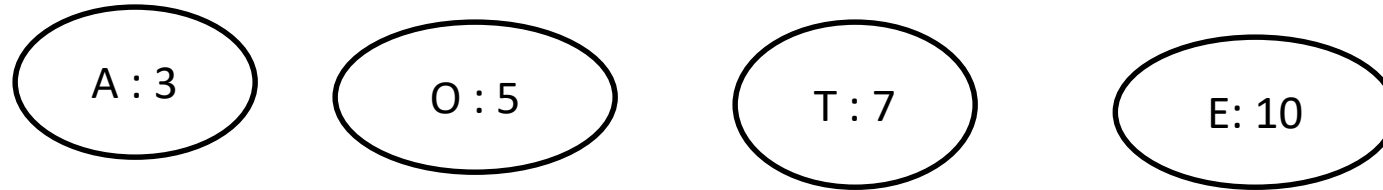
$$(10 + 7 + 5 + 3) * 2 = 50 \text{ bits}$$

- Huffman coding technique uses variable length code depending upon the frequency of characters used in the text. Its coding technique is as follows:

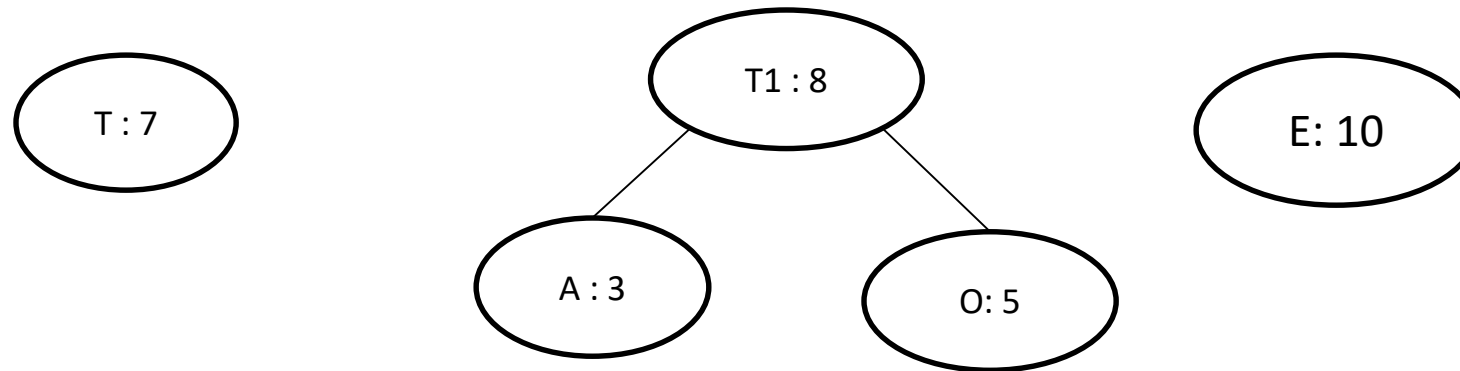
First sort the characters by their frequency in non-decreasing order.

Char	frequency
A	3
O	5
T	7
E	10

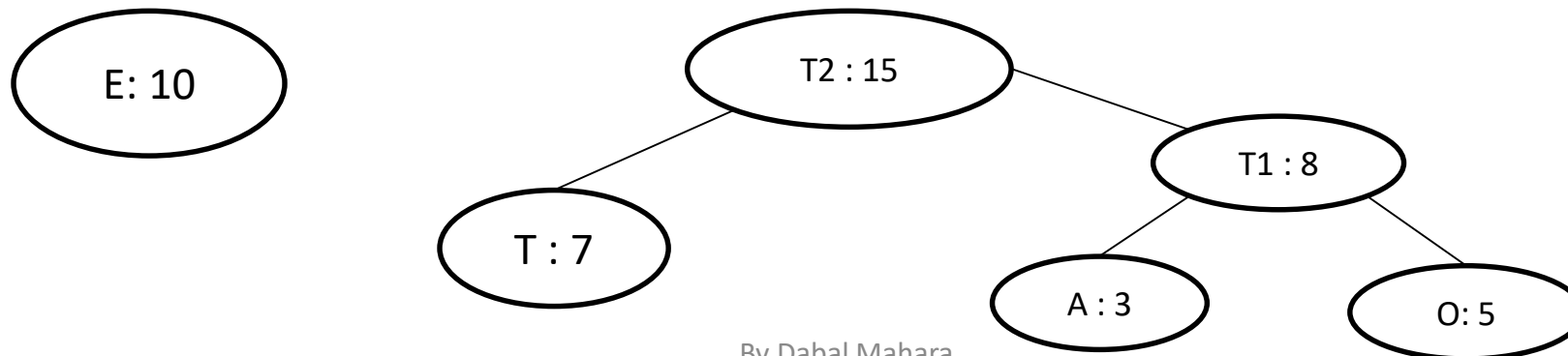
1. Create a forest of one node trees for each character as:



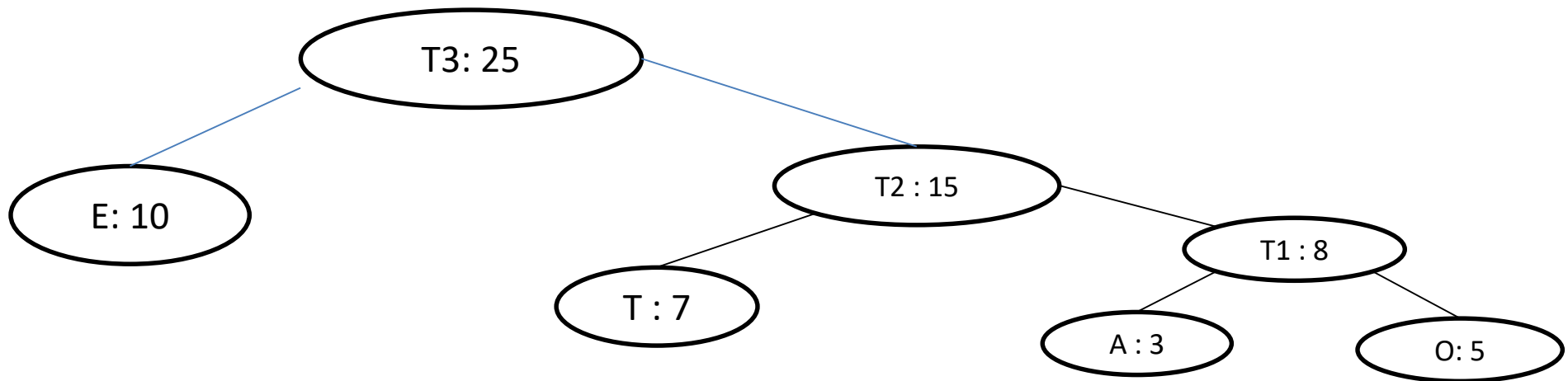
2. Merge two minimum trees as a single tree and add it to the forest in the order.



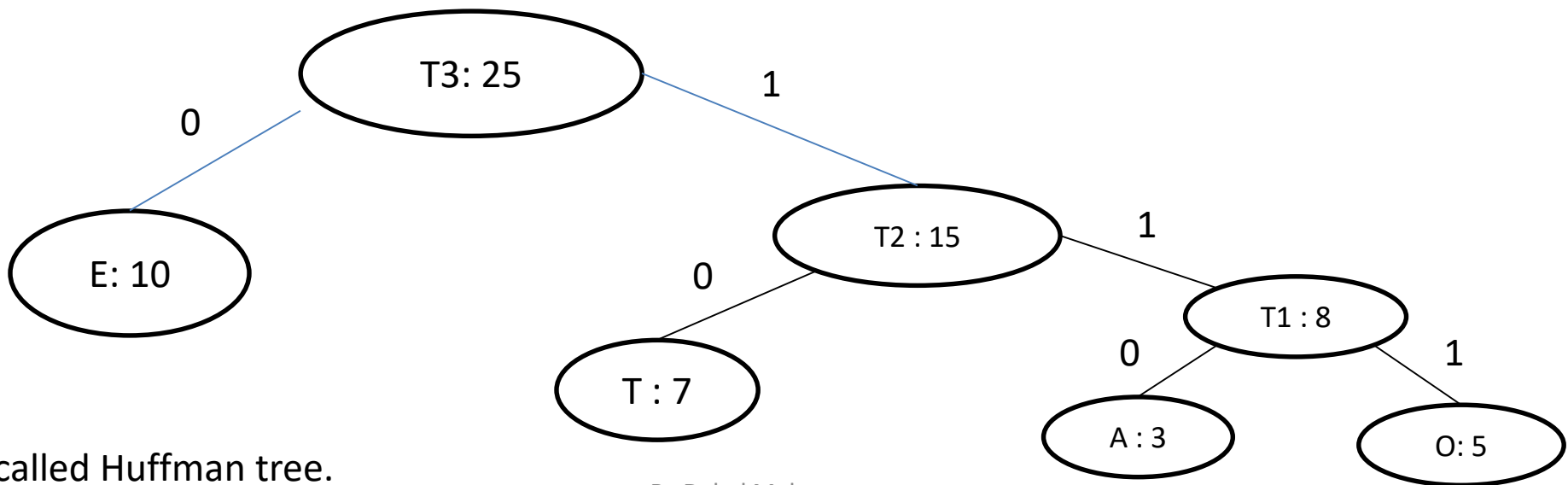
3. Repeat step 2 again.



4. Repeat the step 2 again.



5. Assign 0 to left path and 1 to right path in the tree.



This tree is called Huffman tree.

Now the code for each characters are as obtained by collecting the bits from root to the character node.

The codes are:

Char	code	frequency
E	0	10
T	10	7
A	110	3
O	111	5

Therefore, total number of bits required = $10 * 1 + 7 * 2 + 3 * 3 + 5 * 3 = 48$ bits

The space saved = $(50-48)/50 * 100 \% = 4 \%$

Example: 2

Let us take any four characters and their frequencies as follows:

Char	frequency
a	10
e	15
i	12
s	3
t	4
Sp (space)	13
nl (new line)	1

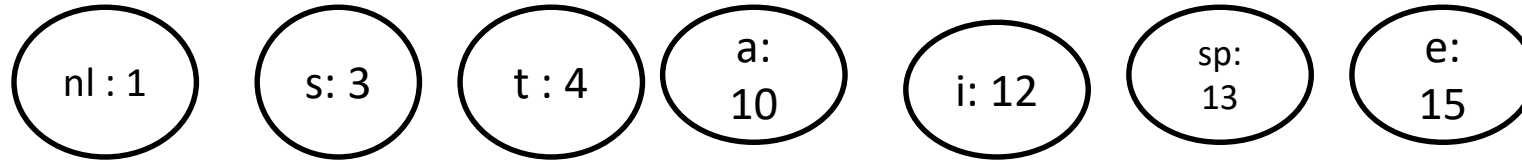
Fixed Length Format: These 7 characters can be represented by three bits.
So, total number of bits = $(10+15+12+3+4+13+1) * 3 = 174$

Using Huffman Coding

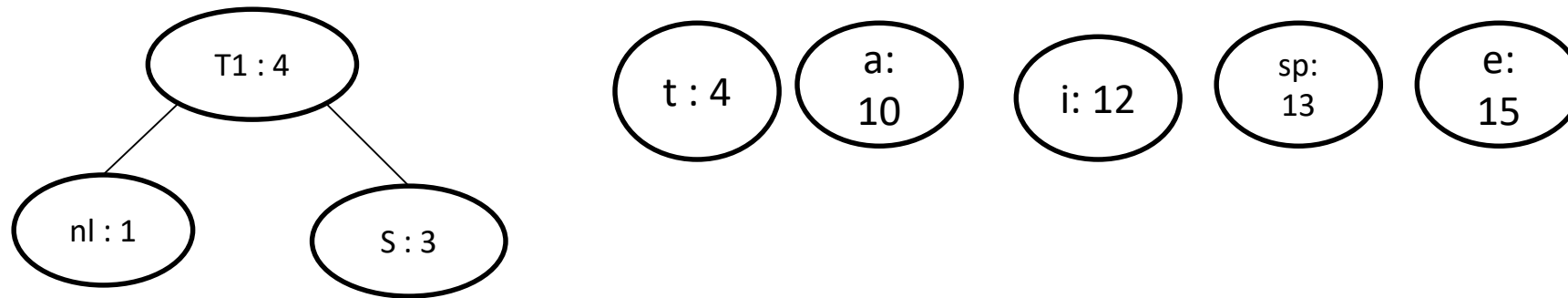
First sort the characters by their frequency in non-decreasing order.

Char	frequency
nl (new line)	1
s	3
t	4
a	10
i	12
Sp (space)	13
e	15

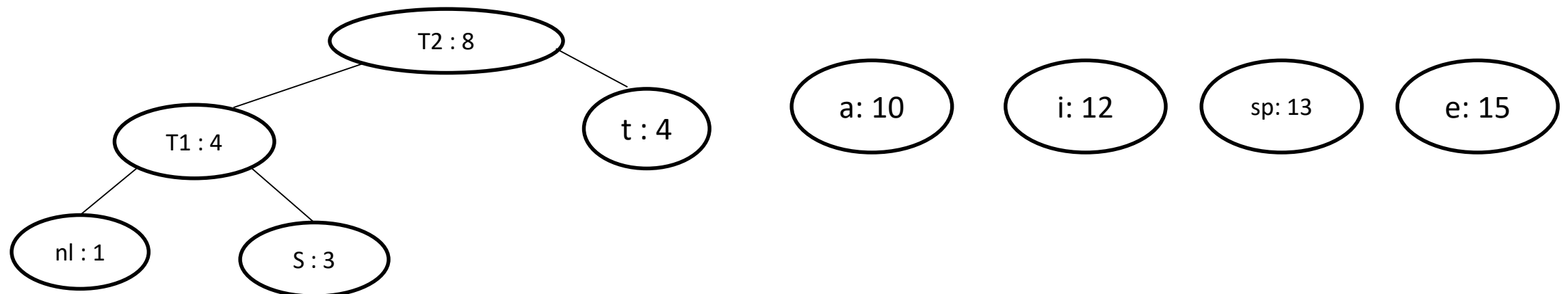
1. Create a forest of one node trees for each character as:



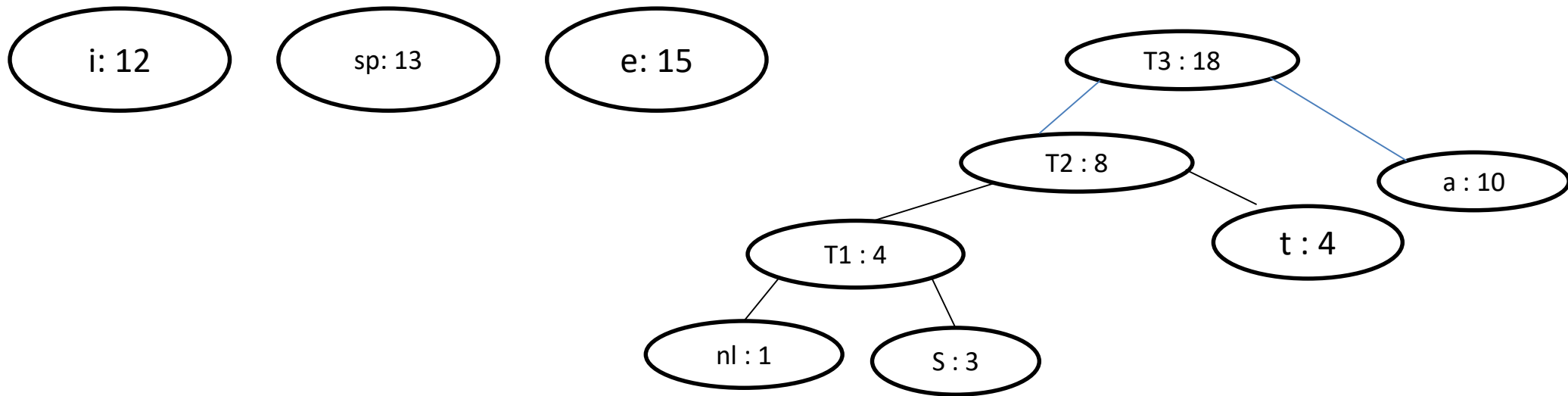
2. Merge two minimum trees as a single tree and add it to the forest in the order.



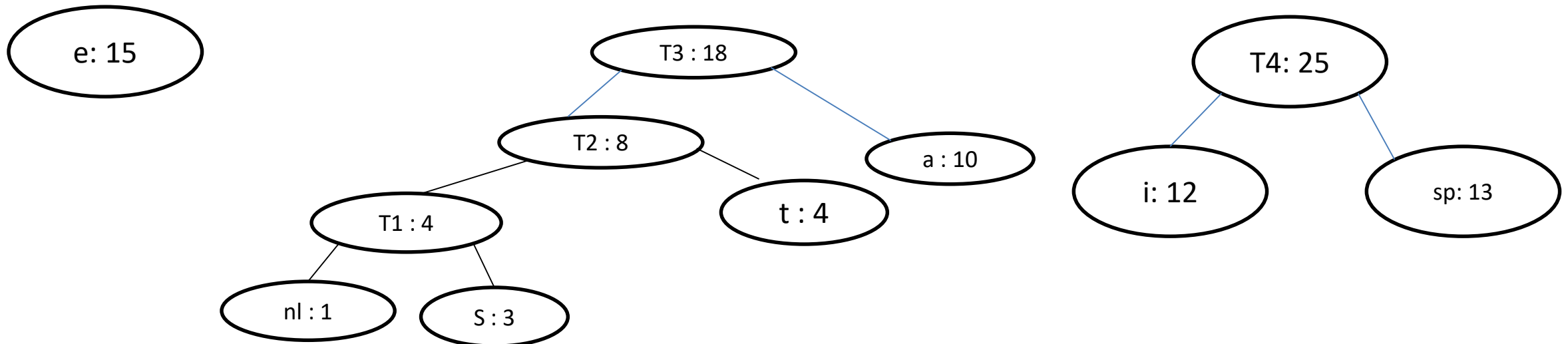
3. Repeat step 2 again.



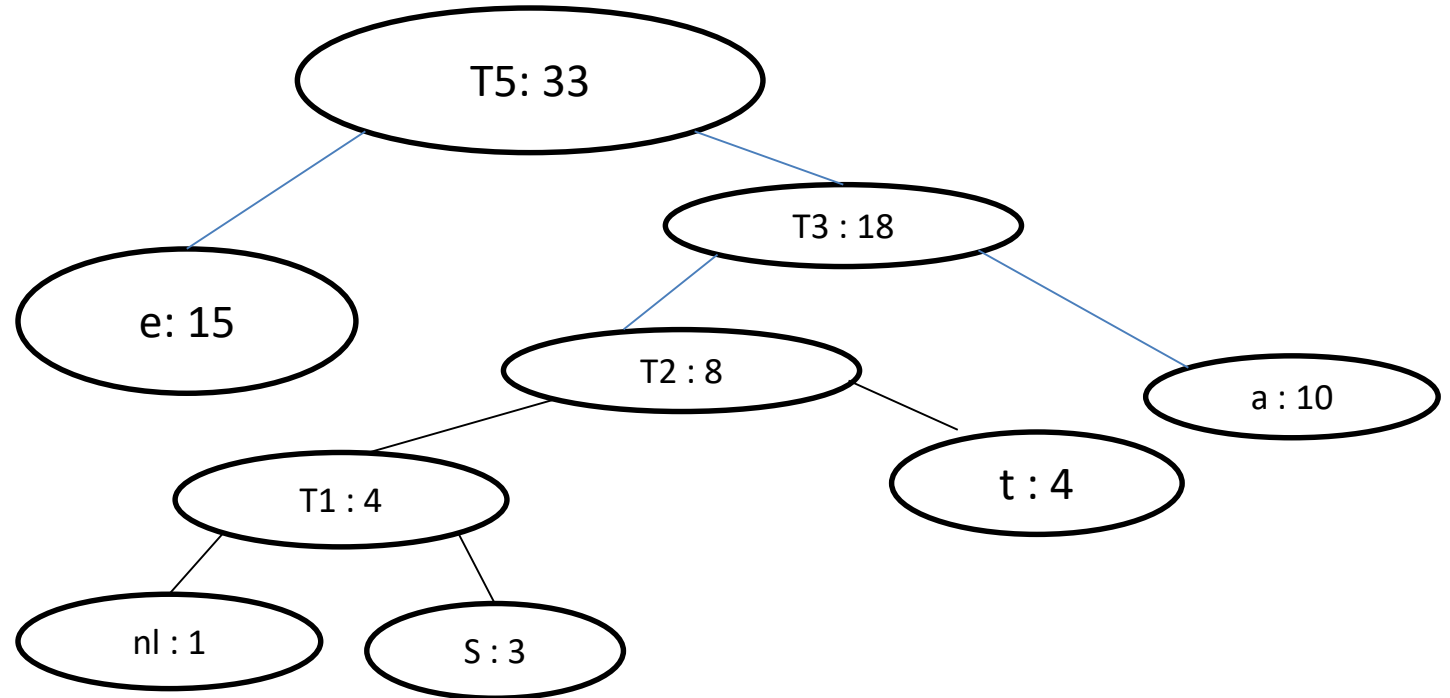
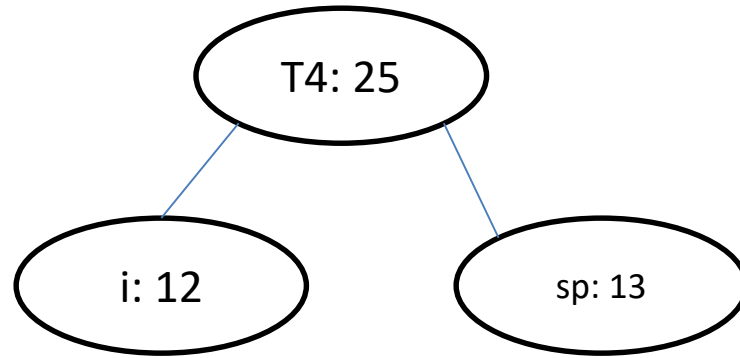
4. Repeat step 2 again.



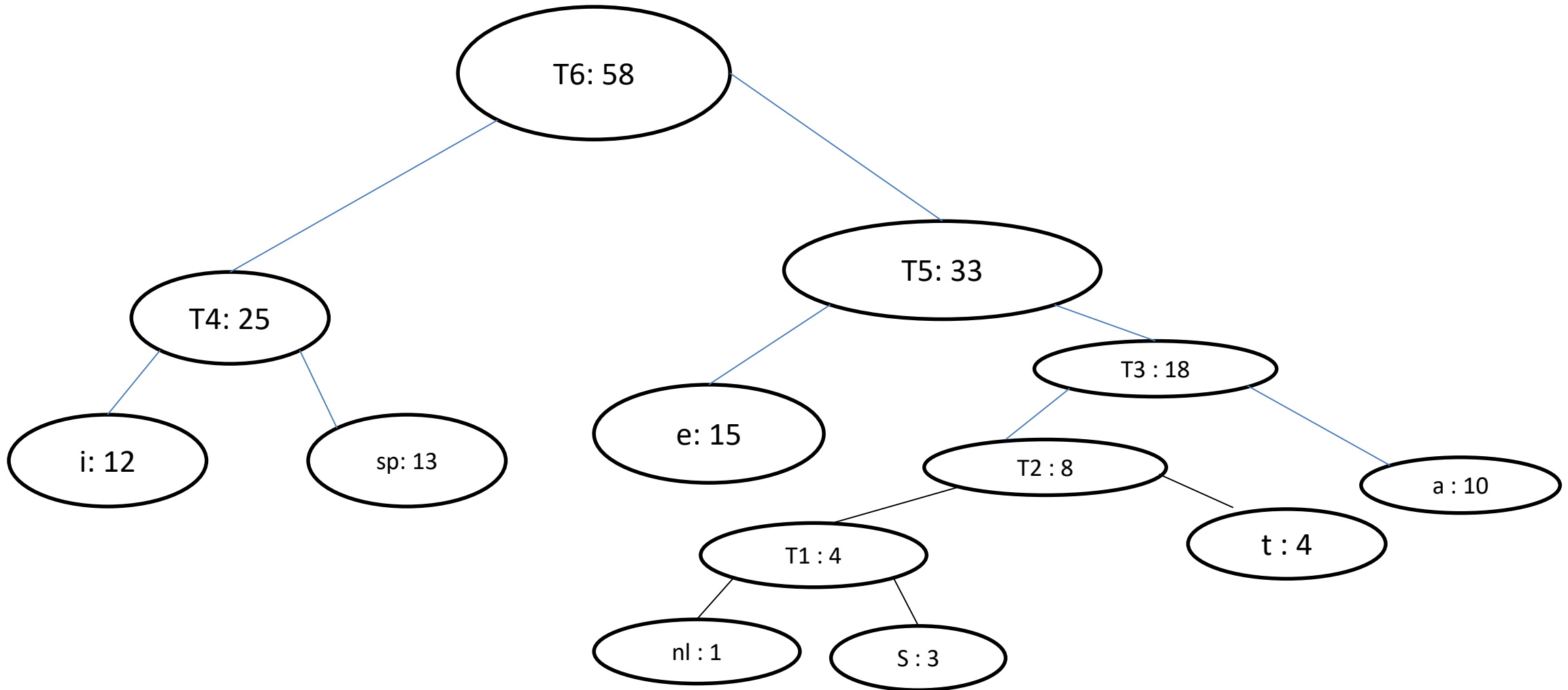
5. Repeat step 2 again.



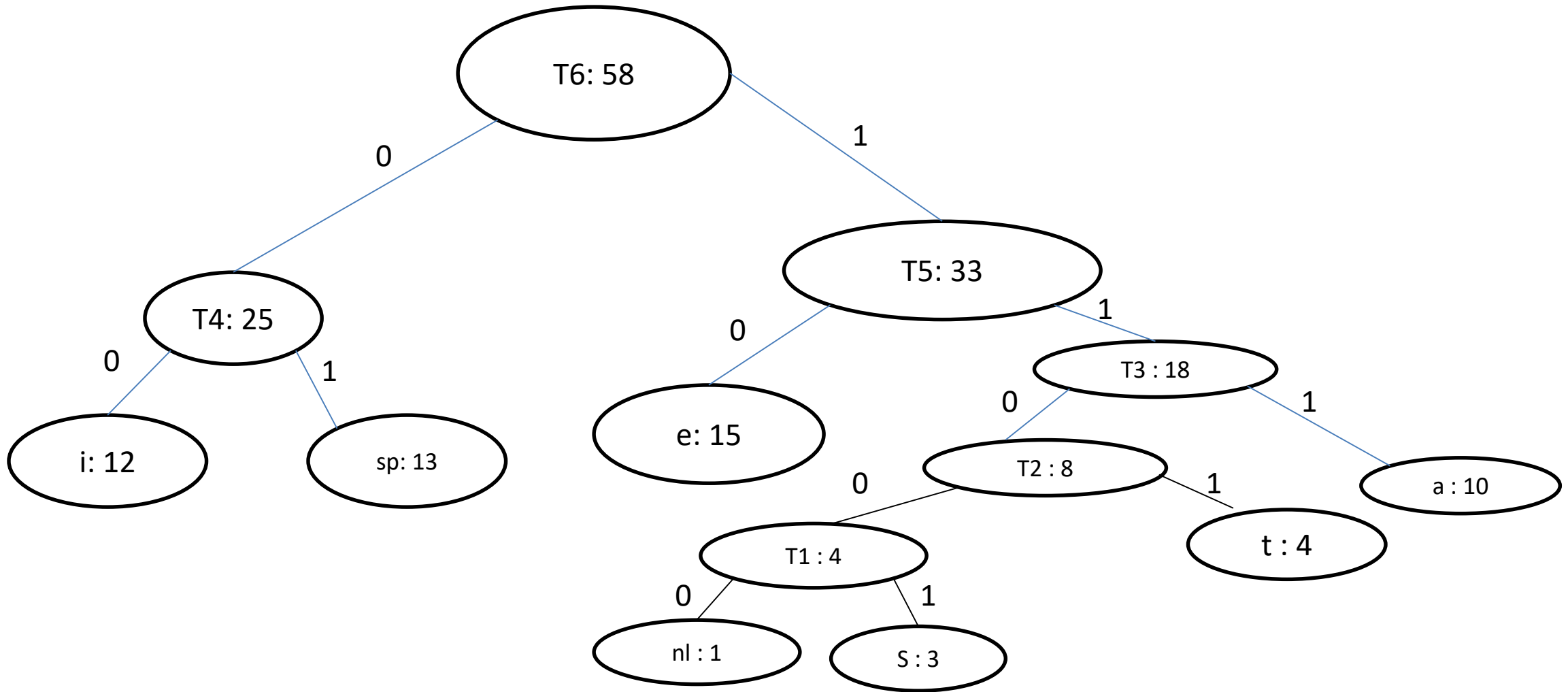
6. Repeat step 2 again.



7. Repeat step 2 again.



8. Assign 0 to left path and 1 to right path in the tree.



This tree is called Huffman tree.

Now the code for each characters are as obtained by collecting the bits from root to the character node.

The codes are:

Char	code	frequency
i	00	12
sp	01	13
e	10	15
nl	11000	1
S	11001	3
t	1101	4
a	111	10

Therefore, total number of bits required

$$= 12*2 + 13 *2 + 15 *2 + 1 *5 + 3 * 5 + 4*4 + 10 * 3$$

$$= 146 \text{ bits}$$

$$\text{The space saved} = (174-146)/174 * 100 \%$$

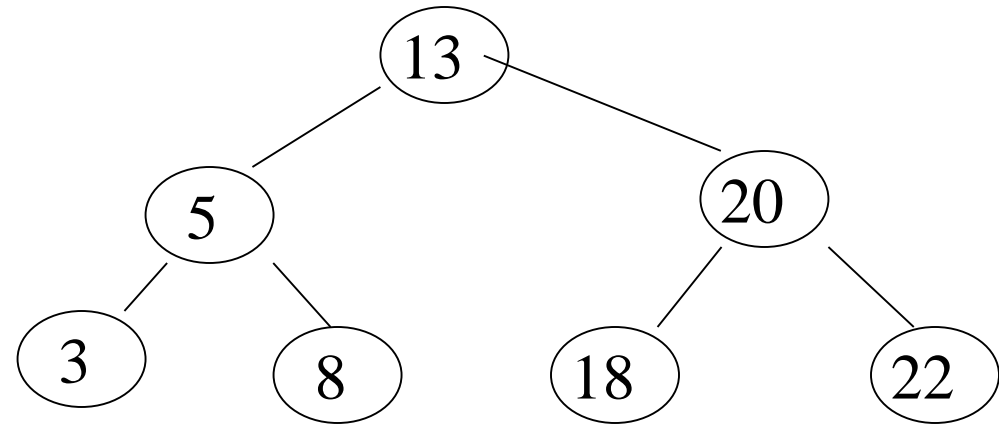
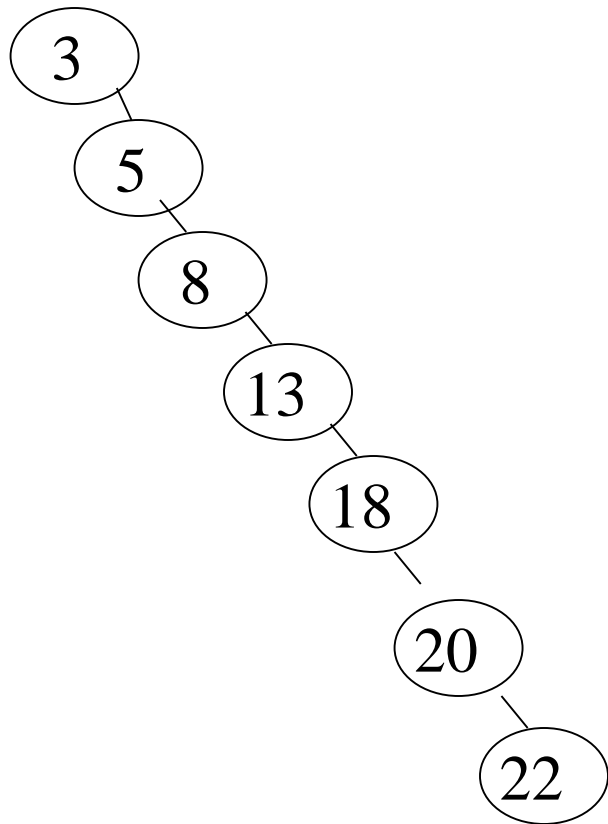
$$= 16.09 \%$$

The Essential Feature

- A B C D E F G H I J K L M N O P Q R S T U V W X Y Z space
- 2 5 1 1 1 1 1 1 2 3 1 2

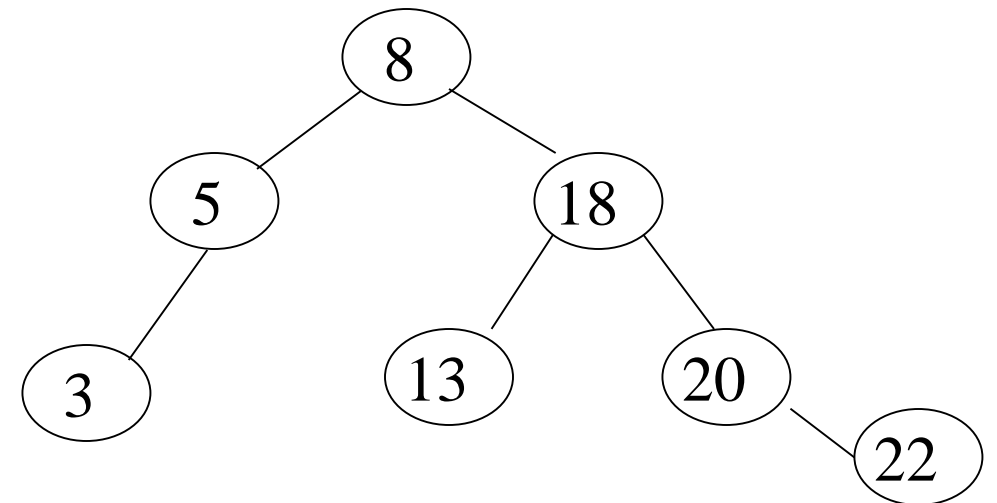
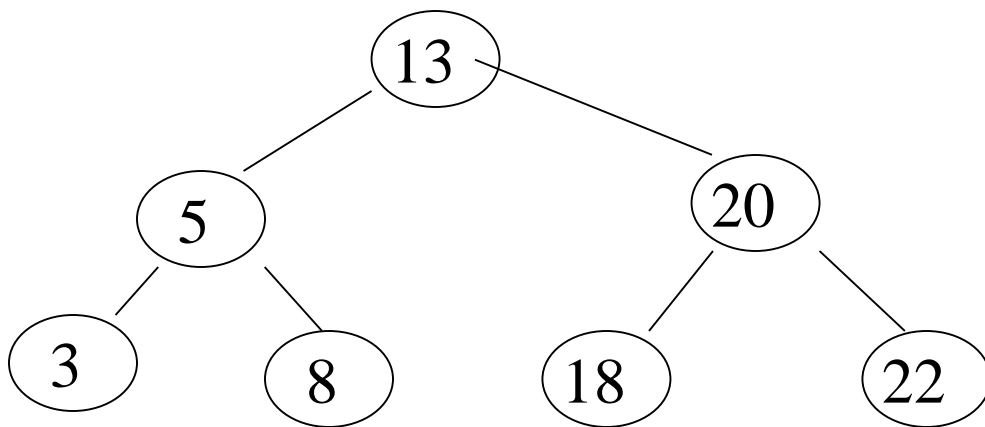
Motivation(AVL Tree)

- When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22



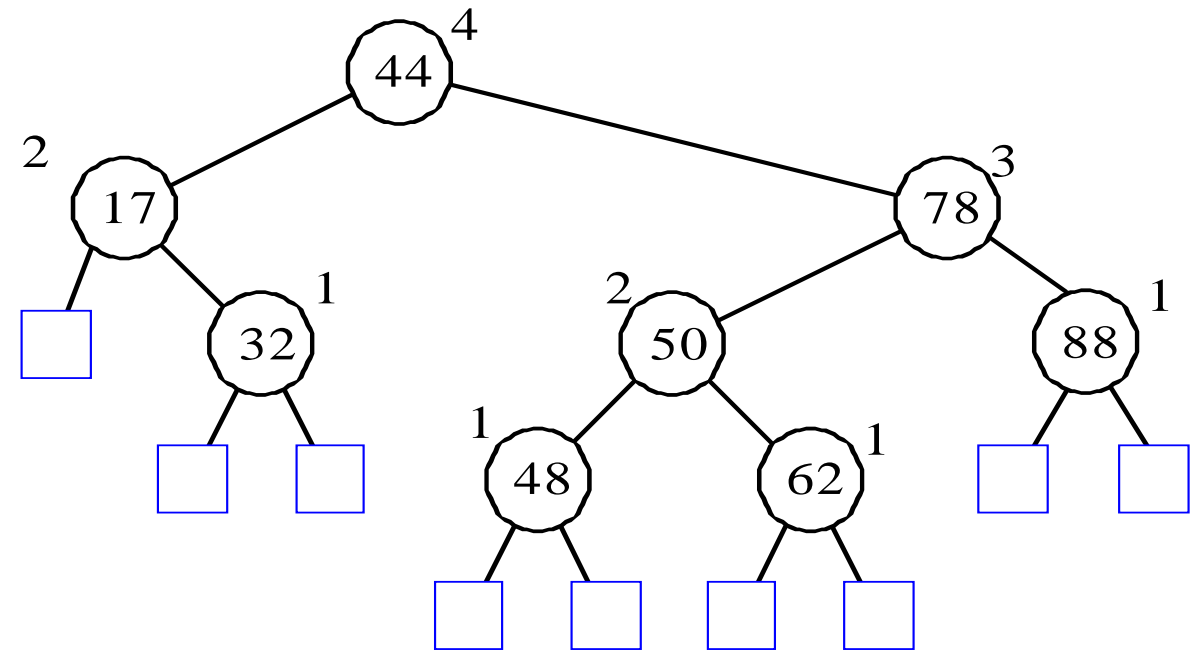
Motivation

- Complete binary tree is hard to build when we allow dynamic insert and remove.
 - We want a tree that has the following properties
 - Tree height = $O(\log(N))$
 - allows dynamic insert and remove with $O(\log(N))$ time complexity.
 - The AVL tree is one of this kind of trees.



AVL (Adelson-Velskii and Landis) Trees

- An AVL Tree is a **binary search tree** such that for every internal node v of T , the *heights of the children of v can differ by at most 1*.

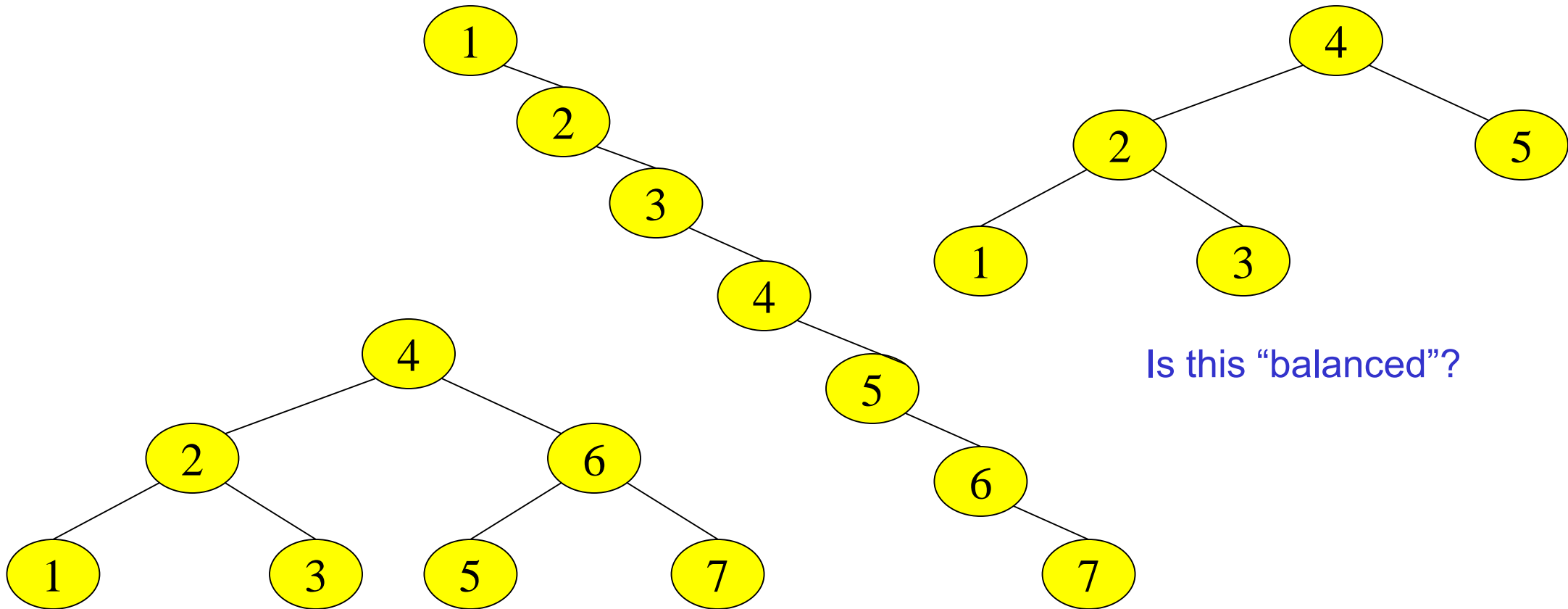


An example of an AVL tree where the heights are shown next to the nodes:

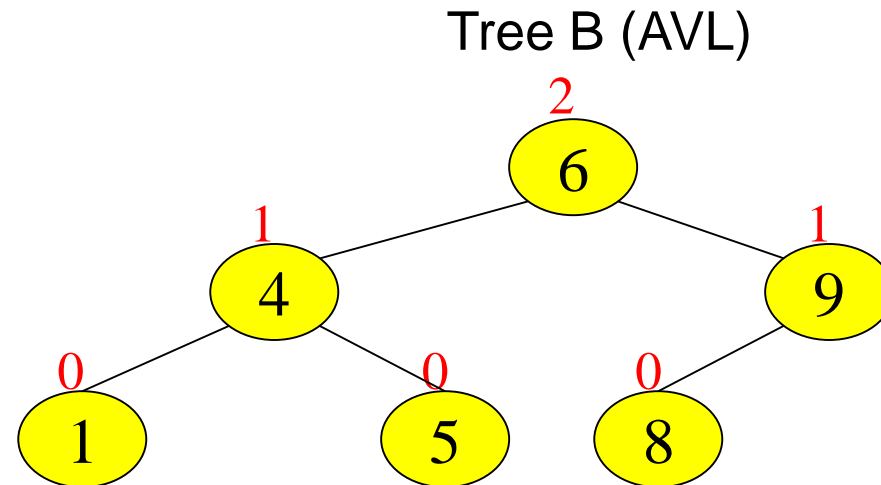
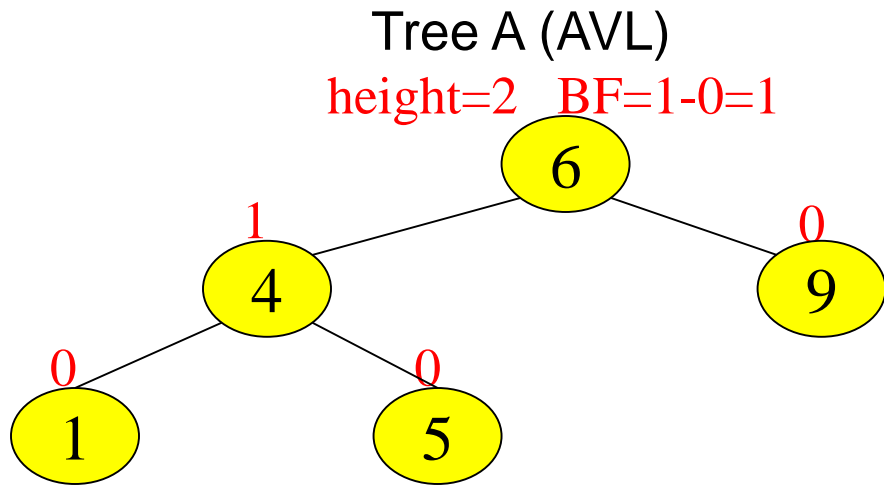
AVL (Adelson-Velskii and Landis) Trees

- AVL tree is a binary search tree with balance condition
 - To ensure depth of the tree is $O(\log(N))$
 - And consequently, search/insert/remove complexity bound $O(\log(N))$
- Balance condition
 - For **every node** in the tree, height of left and right subtree can differ by at most 1

Balanced and unbalanced BST



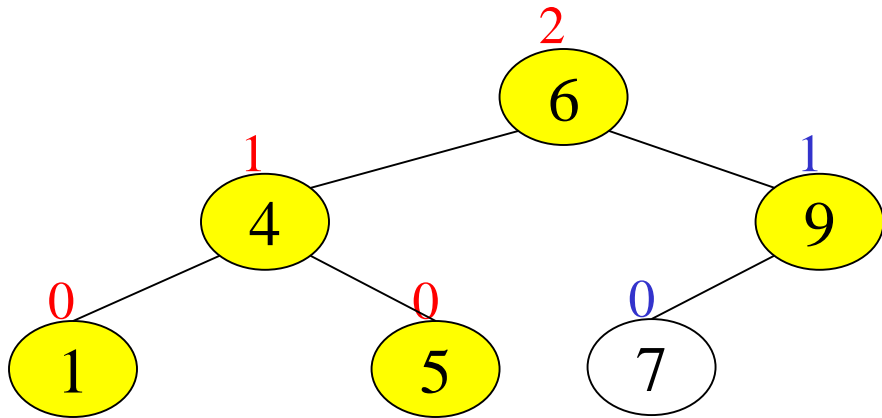
Node Heights



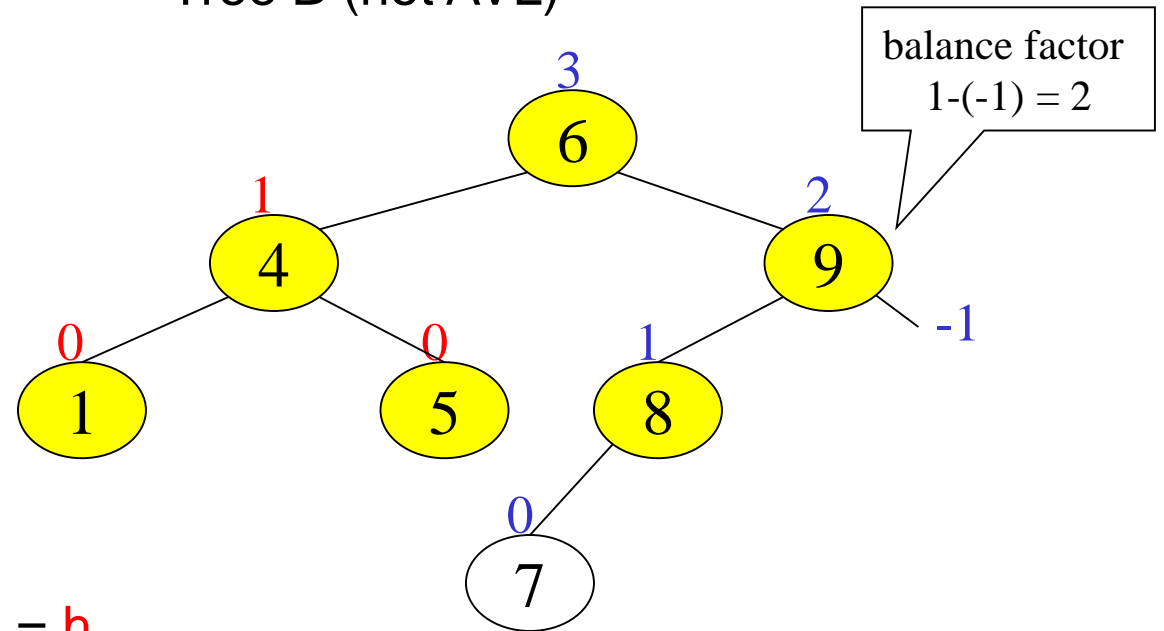
height of node = h
balance factor = $h_{\text{left}} - h_{\text{right}}$
empty height = -1

Node Heights after Insert 7

Tree A (AVL)



Tree B (not AVL)



height of node = h
balance factor = $h_{\text{left}} - h_{\text{right}}$
empty height = -1

Insertions in AVL Trees

There are two categories of rotations, which each contain 2 types of rotations. Each rotation deals with a particular type of imbalance that may arise.

Single rotations

There are 2 rotations in this category:

Right rotation —to deal with the left-left imbalance.

Left rotation —to deal with the right-right imbalance.

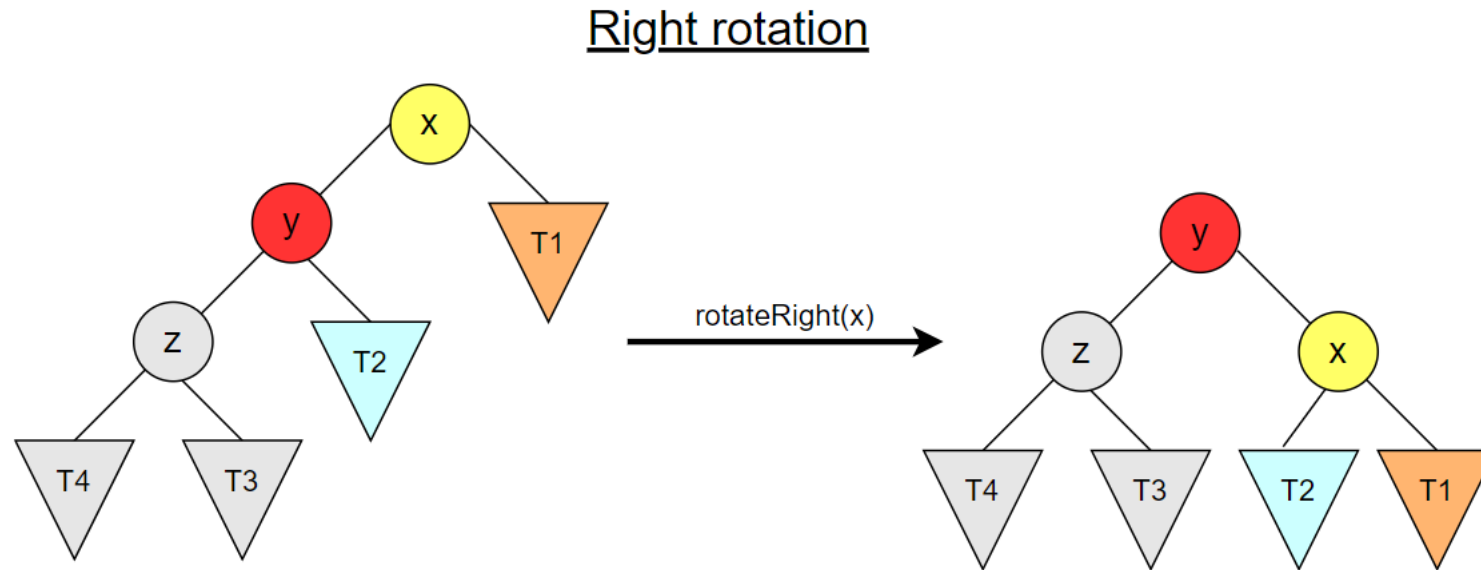
Double rotations

There are 2 rotations in this category:

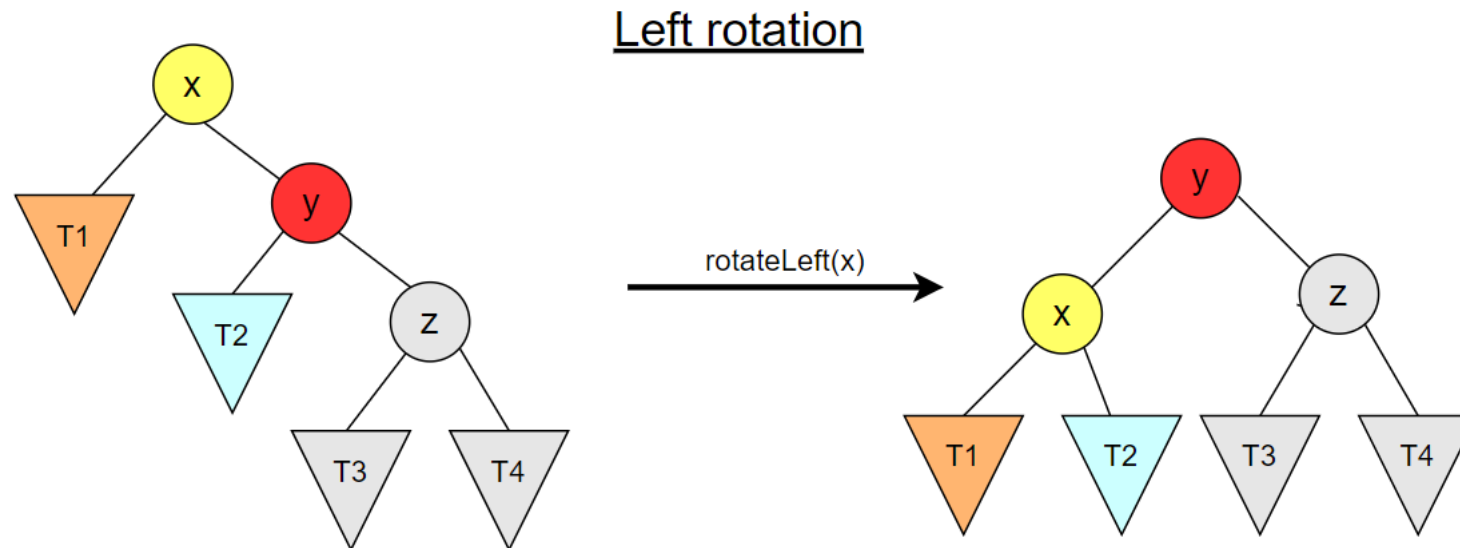
Right-left rotation — to deal with the right-left imbalance.

Left-right rotation —to deal with the left-right imbalance.

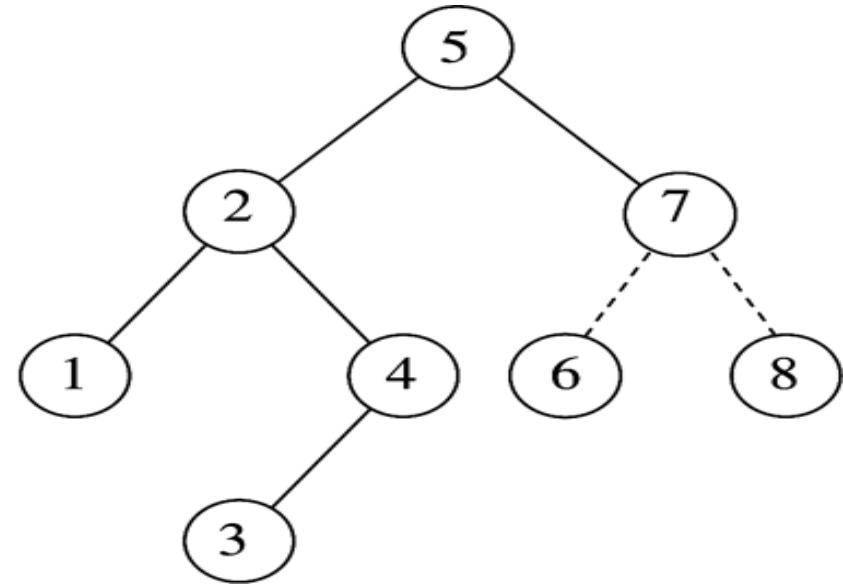
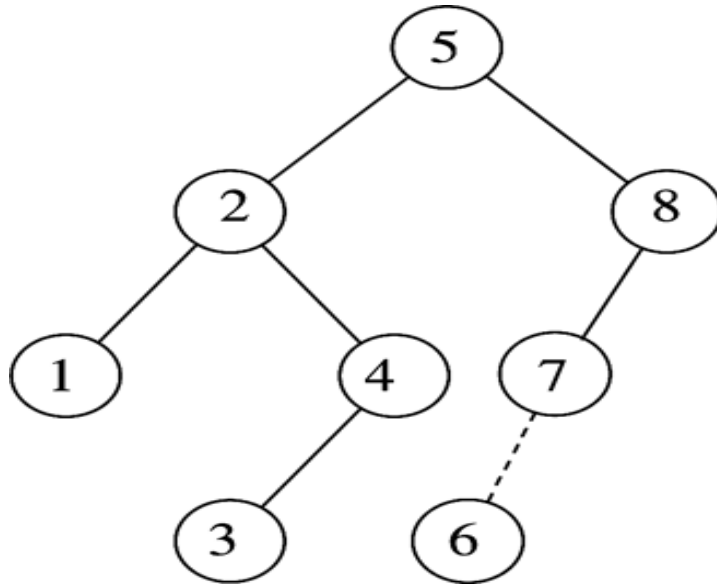
Single Rotation (Deal with left-left imbalance)



Single Rotation (Deal with right-right imbalance)



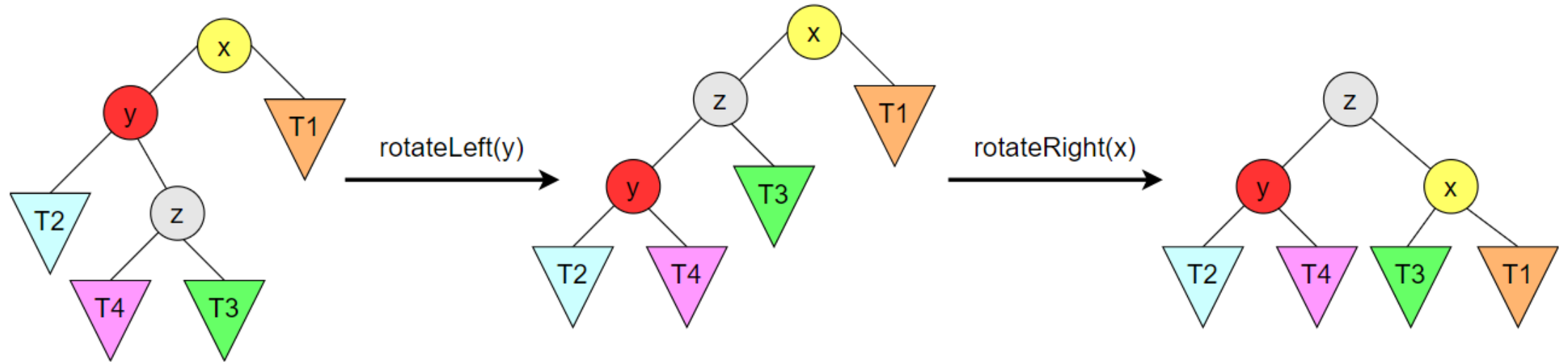
Example



- After inserting 6
 - Balance condition at node 8 is violated

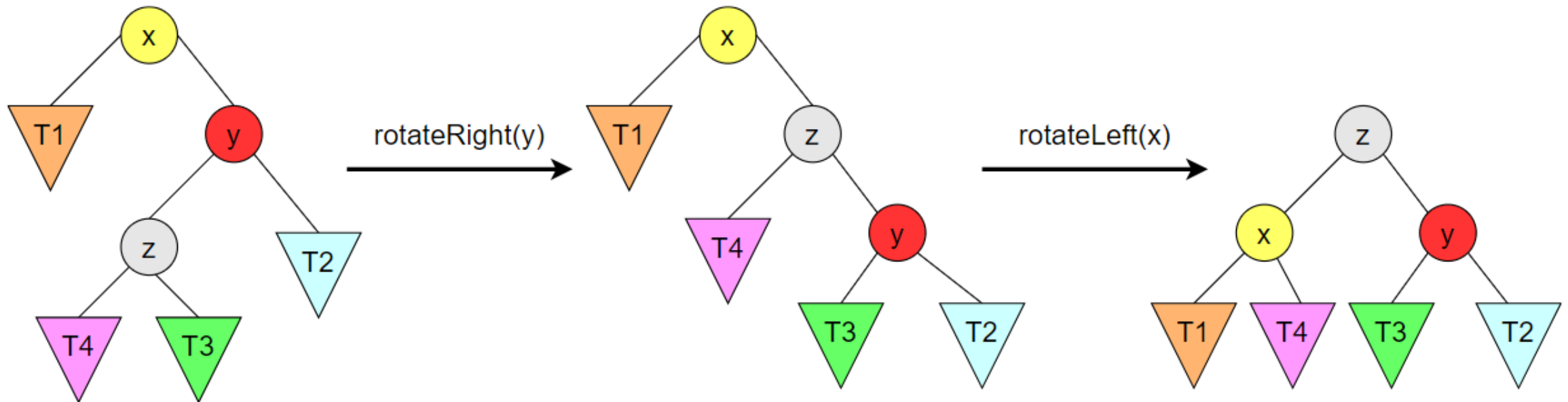
Double Rotation — to deal with the left-right imbalance.

Left-Right rotation



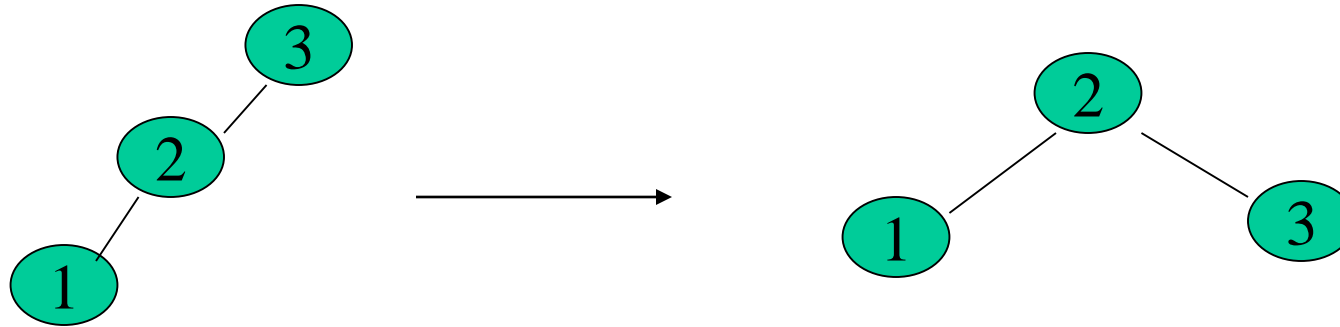
Double Rotation — to deal with the right-left imbalance.

Right-Left rotation



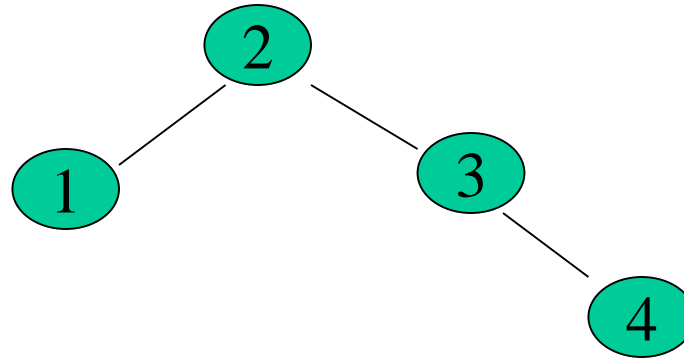
Example

- Inserting 3, 2, 1, and then 4 to 7 sequentially into empty AVL tree

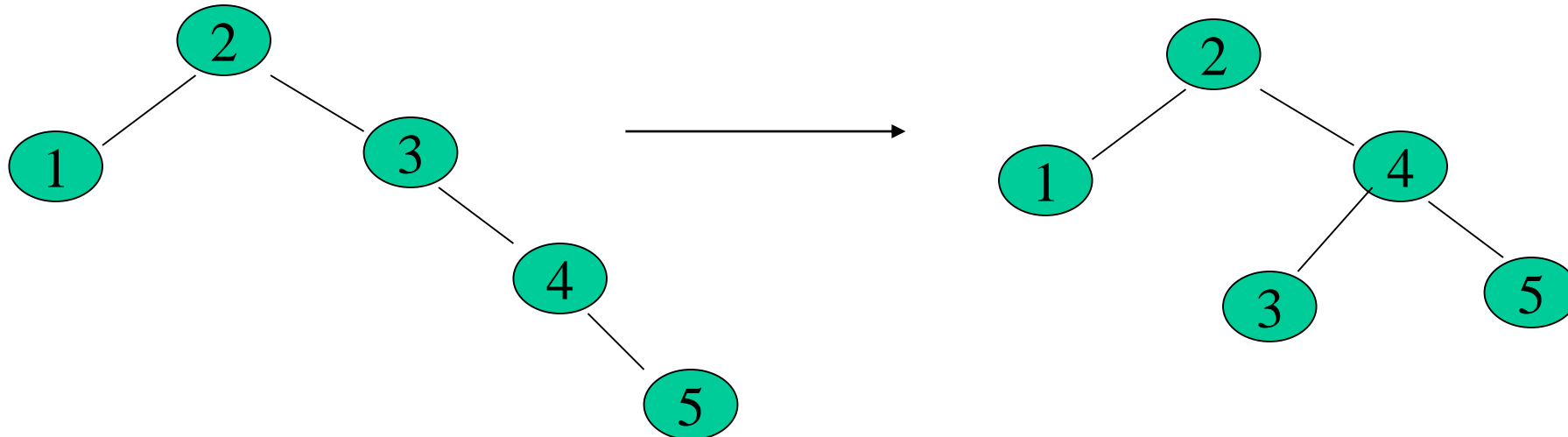


Example (Cont' d)

- Inserting 4

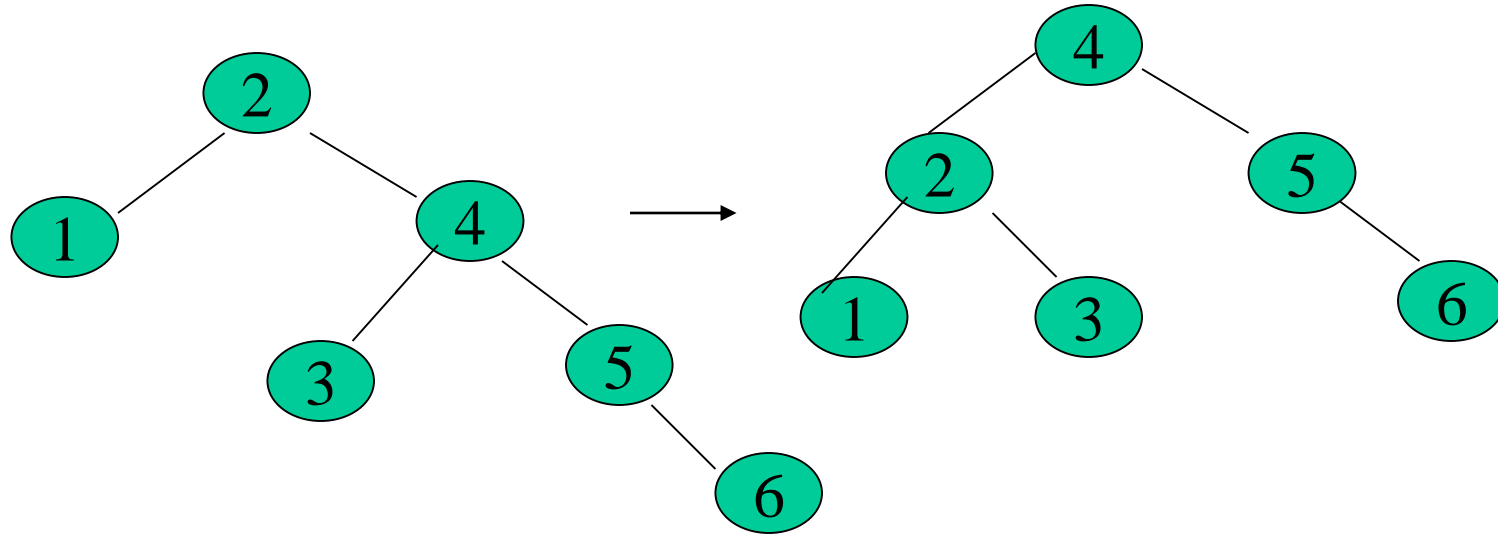


- Inserting 5



Example (Cont' d)

- Inserting 6



- Inserting 7

