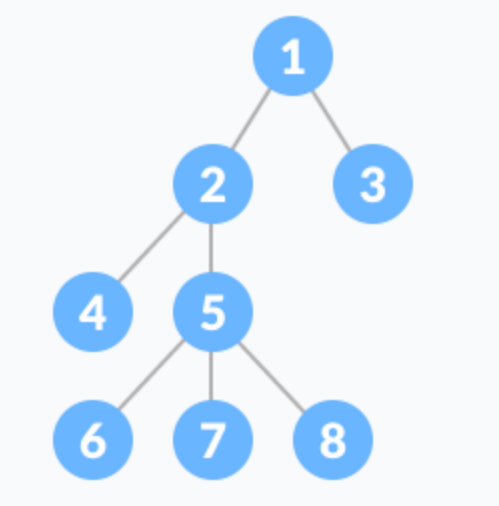
DATA STRUCTURES – Trees & Hashing

UNIT-4&5

Dr. SELVA KUMAR S
ASSISTANT PROFESSOR
B.M.S. COLLEGE OF ENGINEERING

Trees - Introduction

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.



Properties of Trees

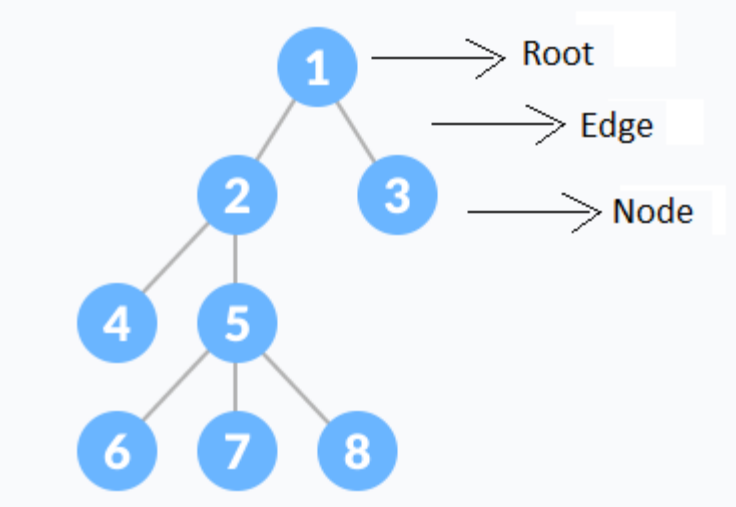
- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has n-1 edges.
- A graph is a tree if and if only if it is minimally connected.
- Any connected graph with n vertices and n-1 edges is a tree.

Tree Applications

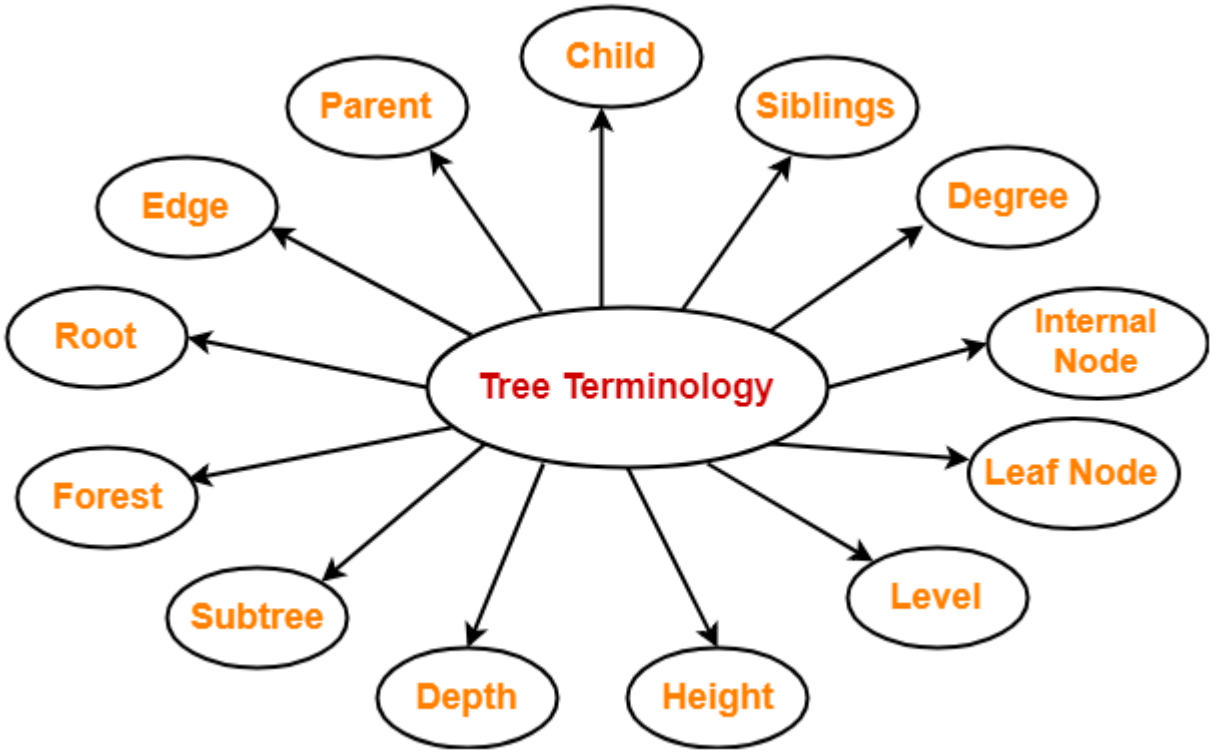
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Tree terminologies

- **Node** - A node is an entity that contains a key or value and pointers to its child nodes.
  - The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.
  - The node having at least a child node is called an **internal node**.
- **Edge -**It is the link between any two nodes.
- **Root -** It is the topmost node of a tree.

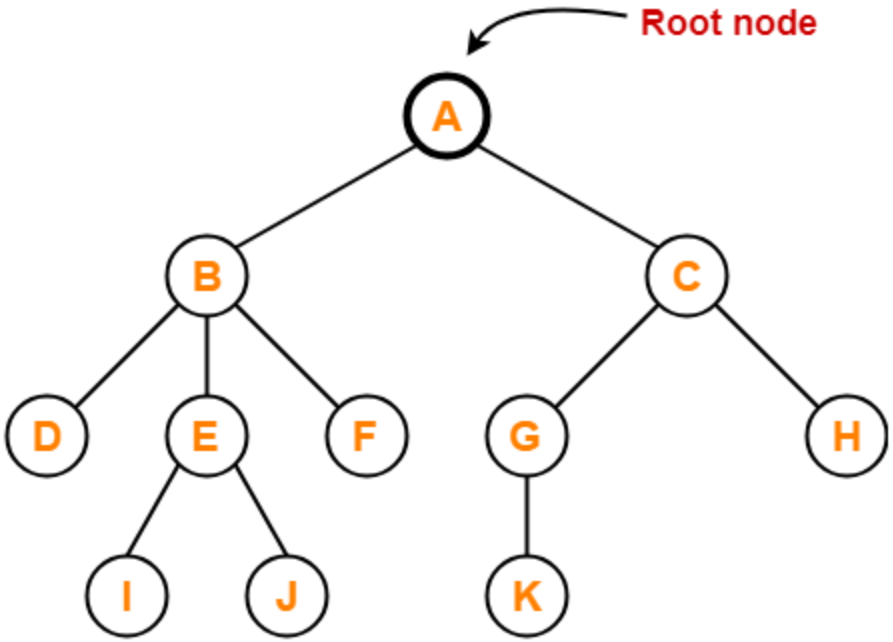


Tree terminologies



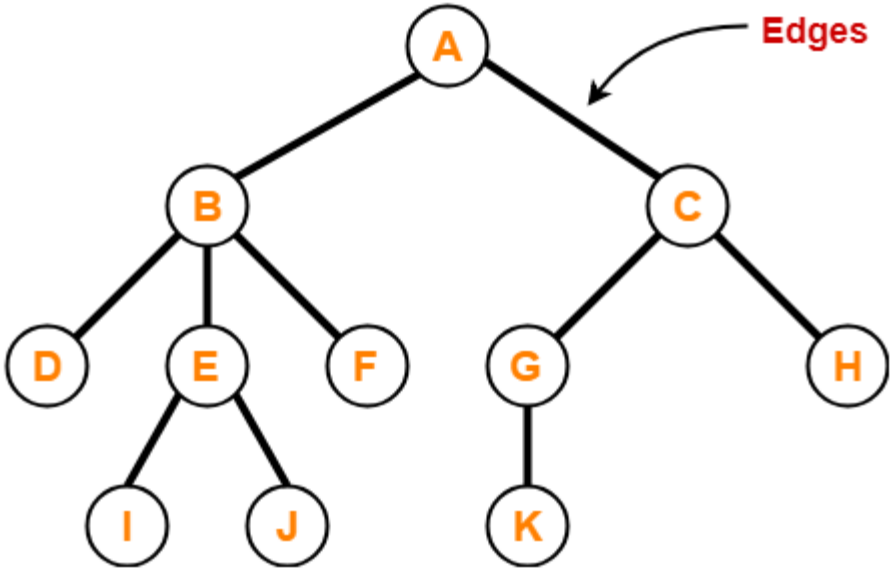
Root

- 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.



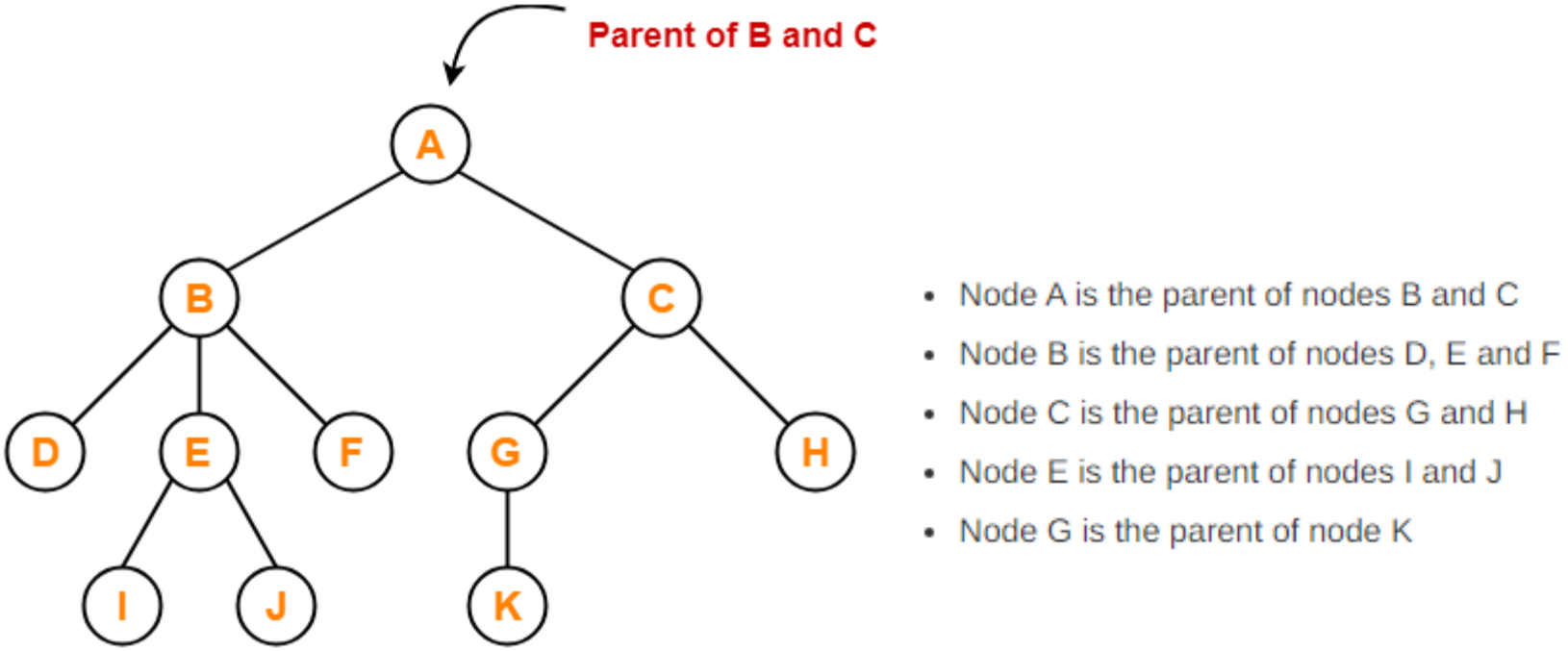
Edge

- 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.



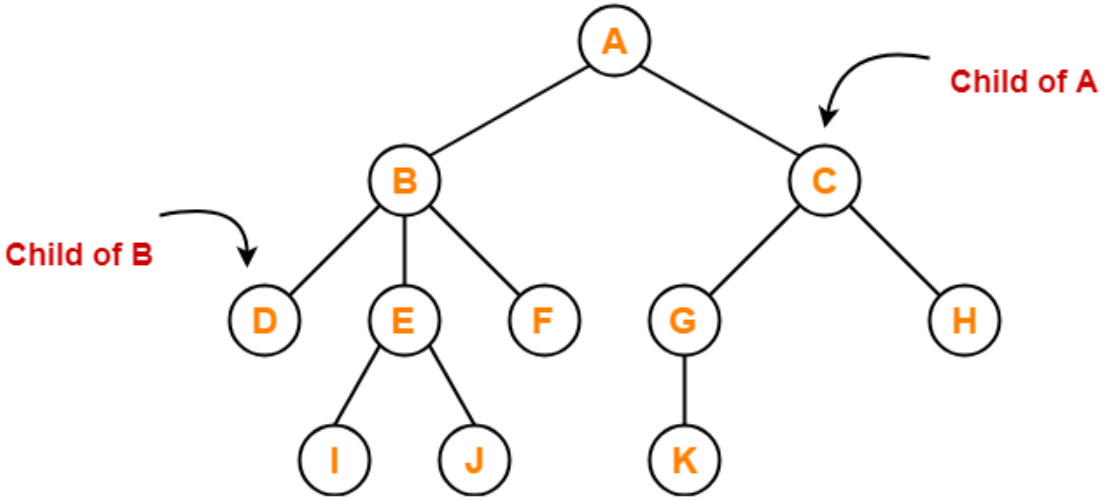
Parent

- 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

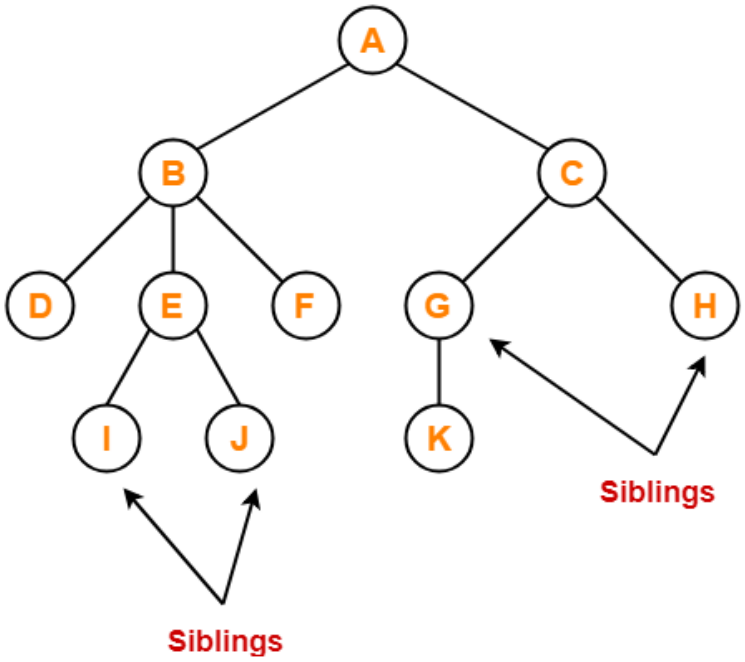
- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.



- 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

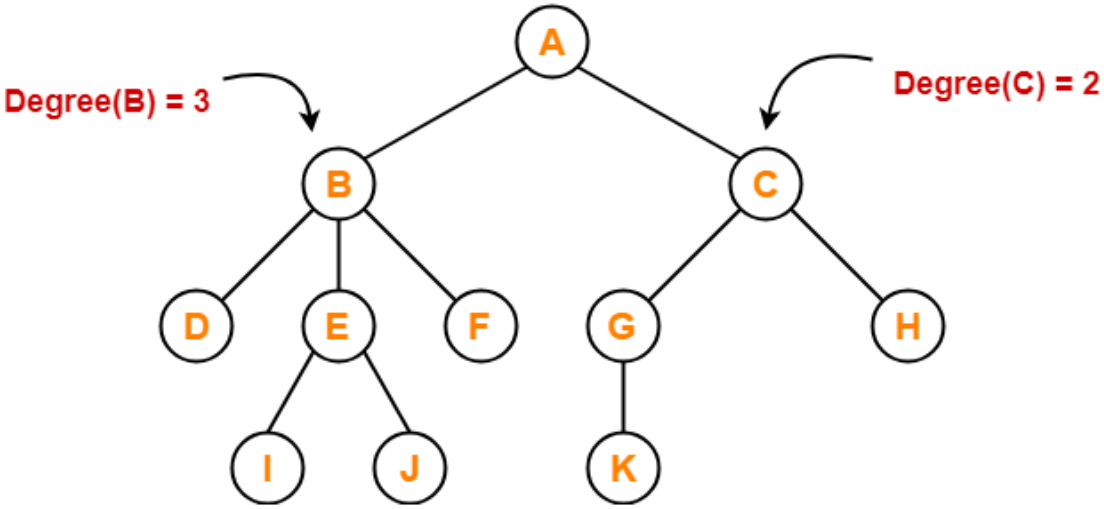
- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.



- 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

Degree

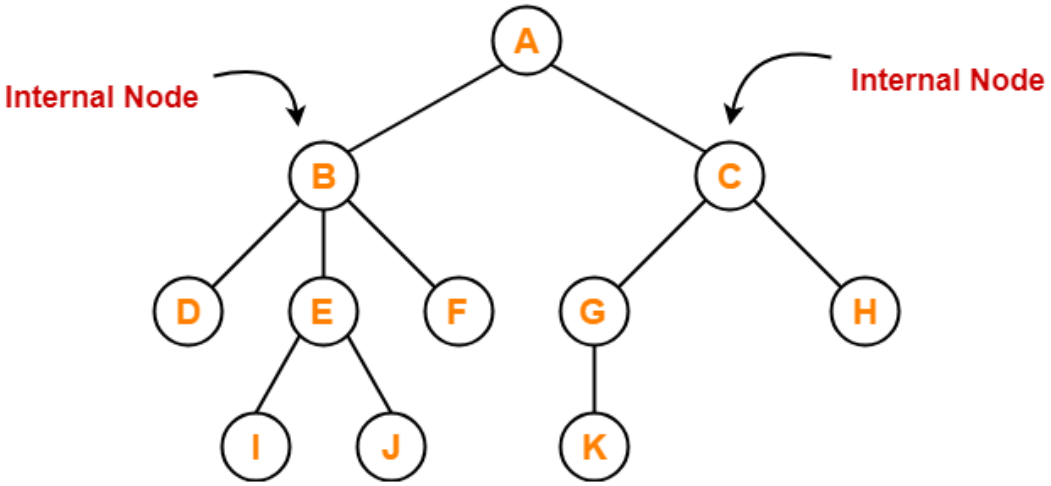
- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.



- 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

Internal node

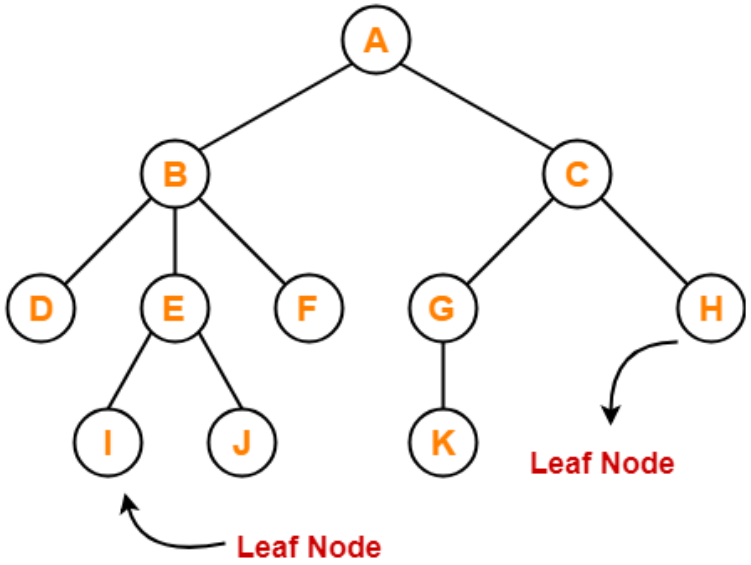
- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



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

Leaf node

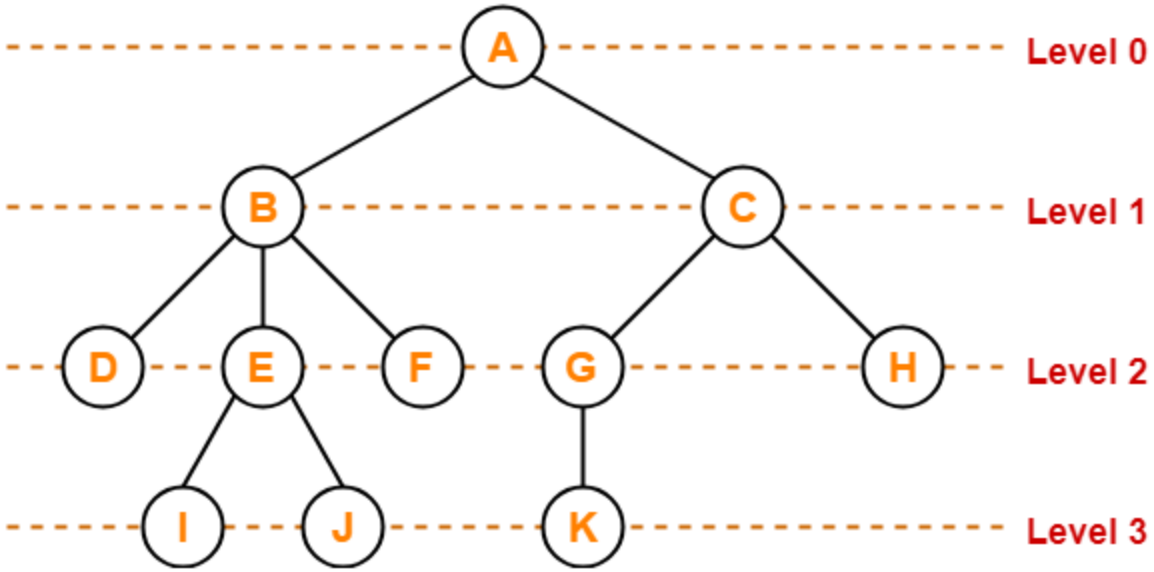
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



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

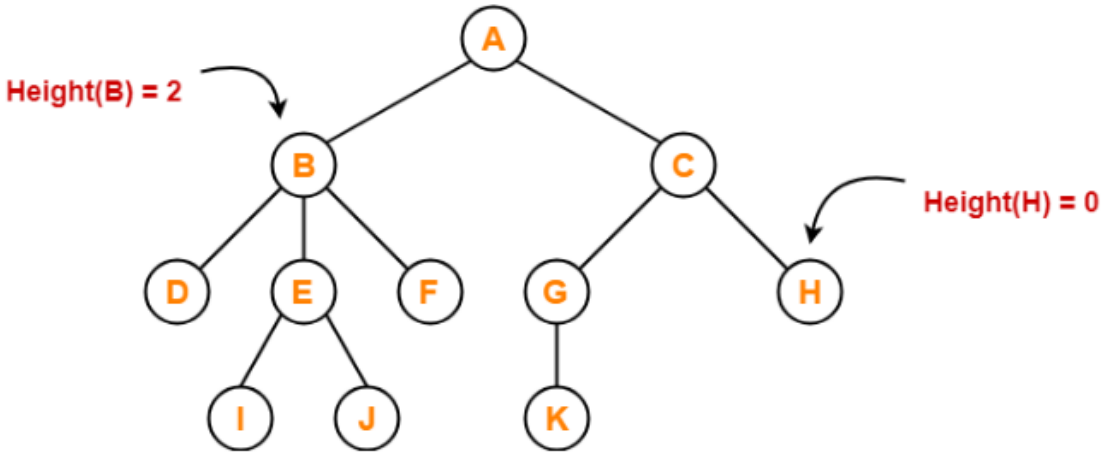
Level

- 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

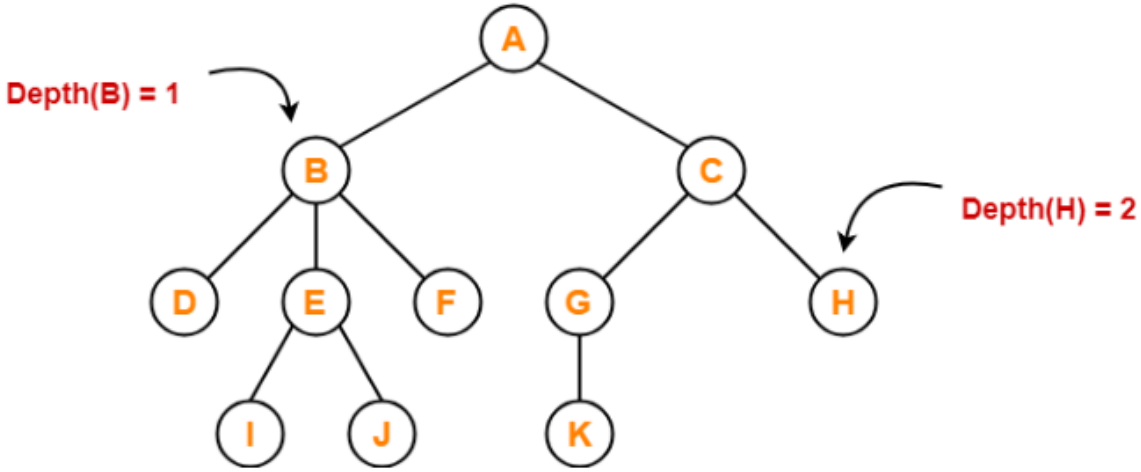
- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0



- 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

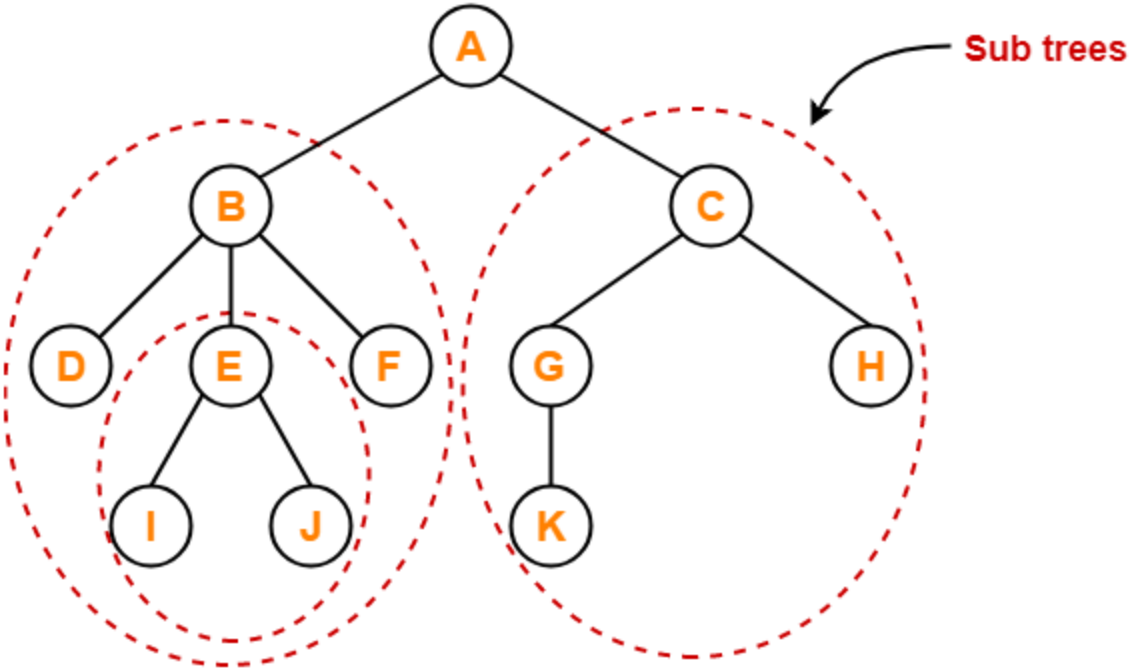
- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



- 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
- Depth of node K = 3

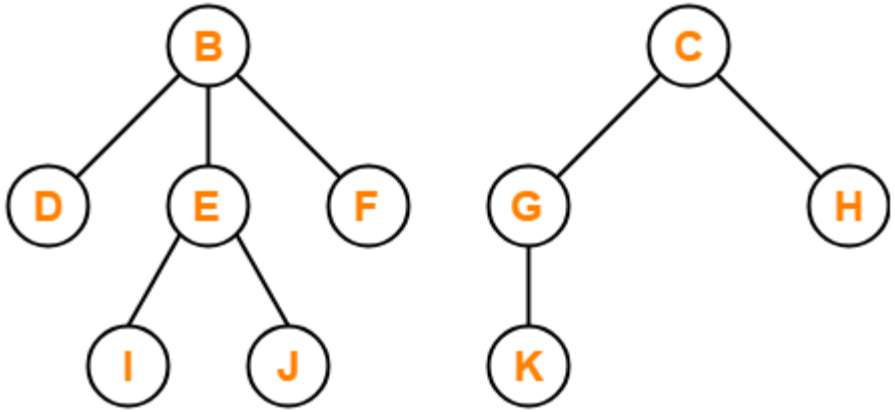
Subtree

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



Forest

- A forest is a set of disjoint trees.



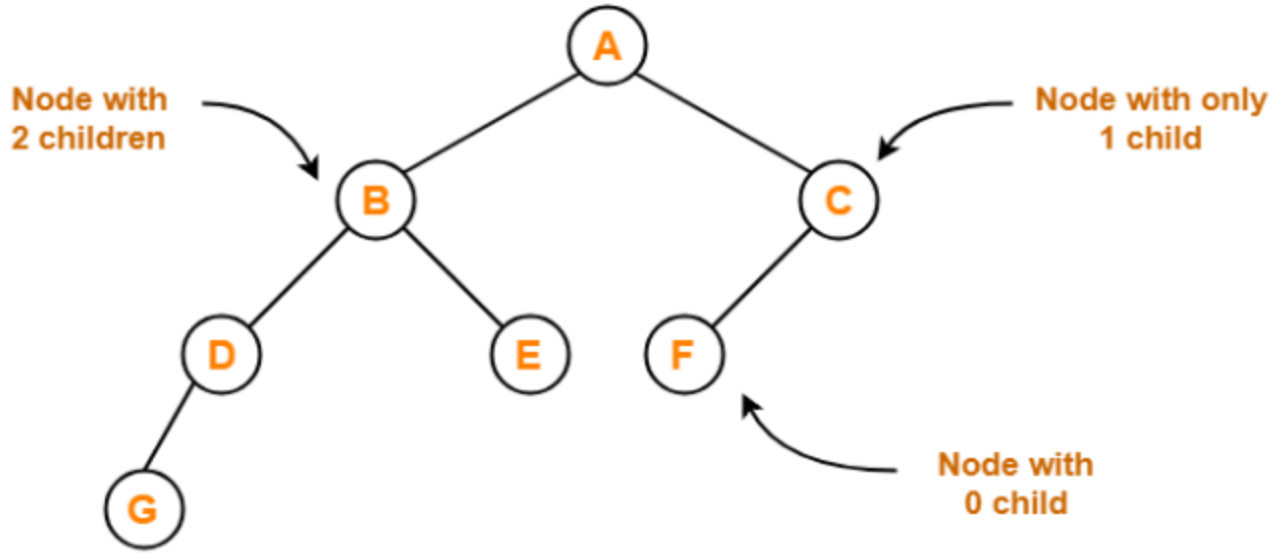
Forest

Types of Tree

- General Tree
- Binary Tree
- Binary Search Tree
- AVL Tree
- Red-Black Tree
- N-ary Tree

Binary Tree

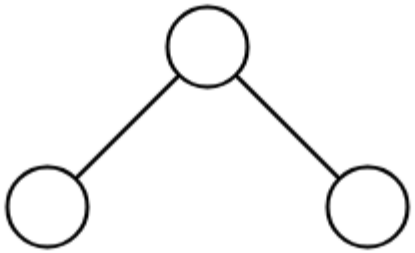
- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.



Binary Tree Example

Unlabeled Binary Tree

- A binary tree is unlabeled if its nodes are not assigned any label.

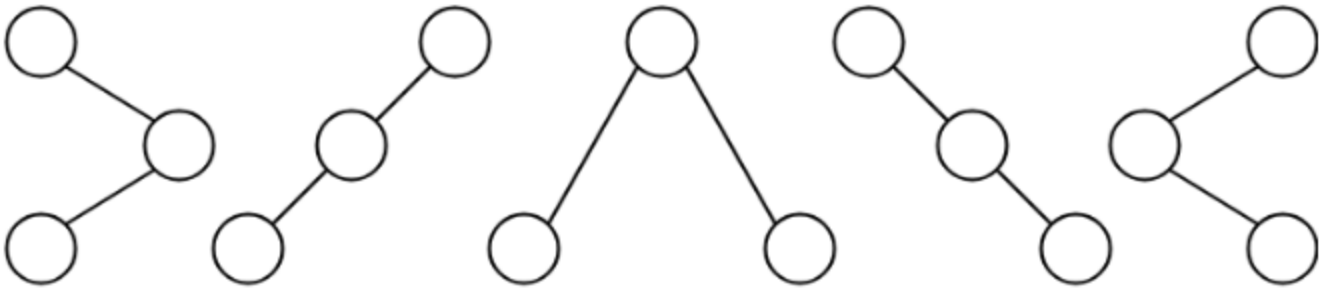


Unlabeled Binary Tree

Number of different Binary Trees possible with 'n' unlabeled nodes $= \frac{{}^{2n}C_n}{n+1}$

Example

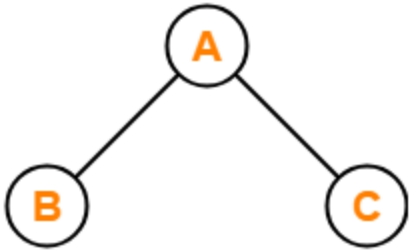
- Consider we want to draw all the binary trees possible
- Number of binary trees possible with 3 unlabeled nodes
- $= {}^{2 \times 3}C_3 / (3 + 1)$
- $= {}^{6}C_3 / 4$
- $= 5$



Binary Trees Possible With 3 Unlabeled Nodes

Labeled Binary Tree

- A binary tree is labelled if all its nodes are assigned a label.



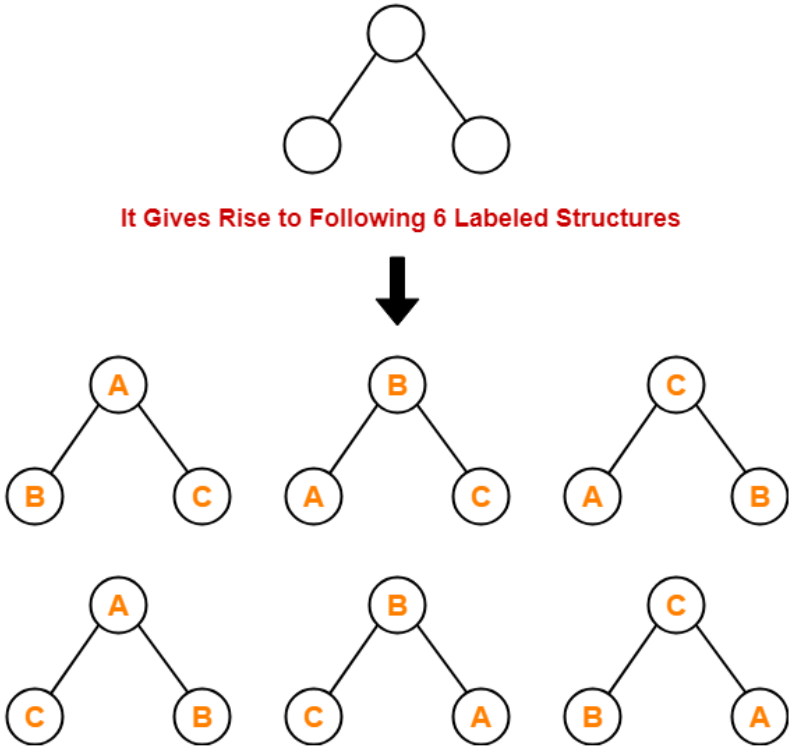
Labeled Binary Tree

Number of different Binary Trees possible with 'n' labeled nodes

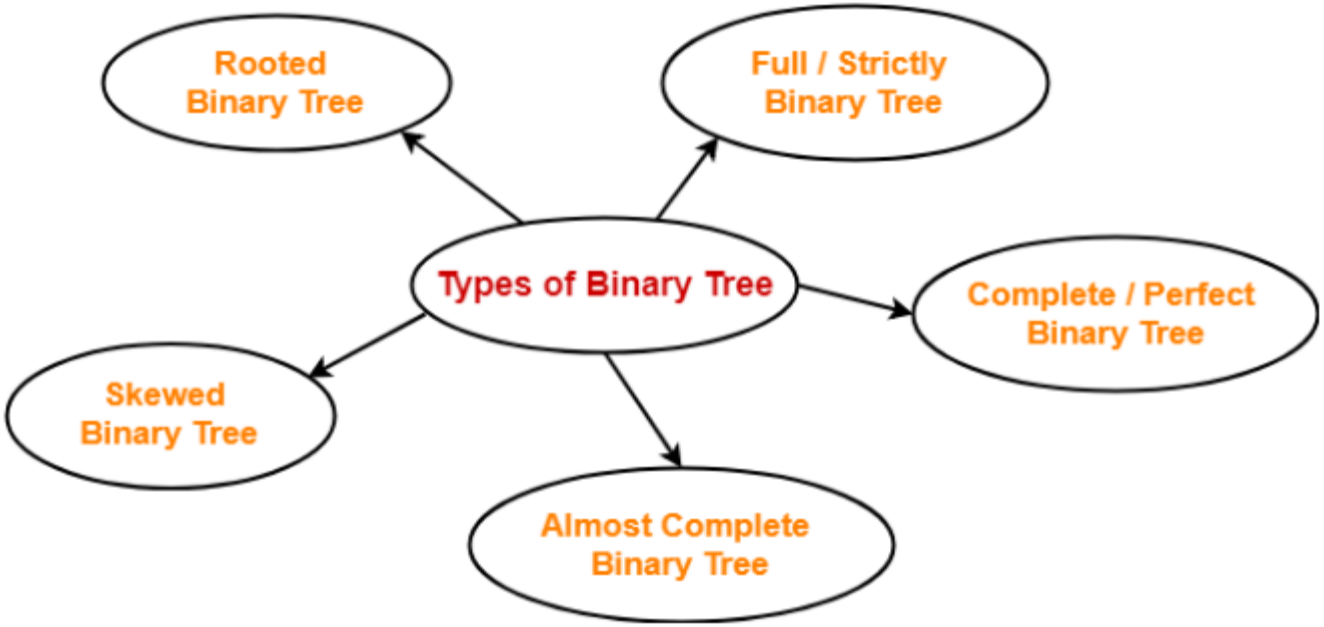
$$= \frac{{}^{2n}C_n}{n+1} \times n!$$

Example

- Consider we want to draw all the binary trees possible with 3 labeled nodes.
- Number of binary trees possible with 3 labeled nodes
- $= \{ {}^{2 \times 3}C_3 / (3 + 1) \} \times 3!$
- $= \{ {}^{6}C_3 / 4 \} \times 6$
- $= 5 \times 6$
- $= 30$

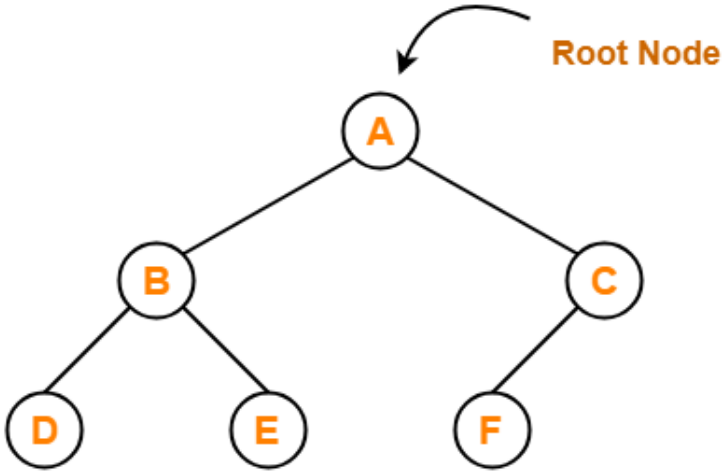


Types of Binary Trees



Rooted Binary Tree

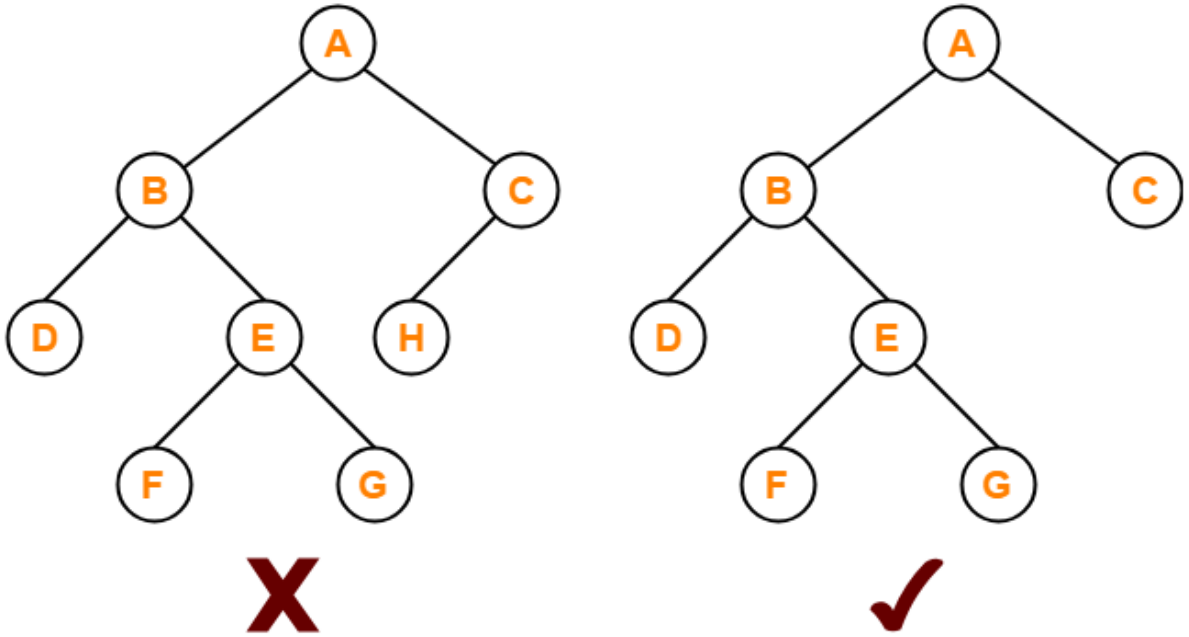
- A **rooted binary tree** is a binary tree that satisfies the following 2 properties:
  - It has a root node.
  - Each node has at most 2 children.



Rooted Binary Tree

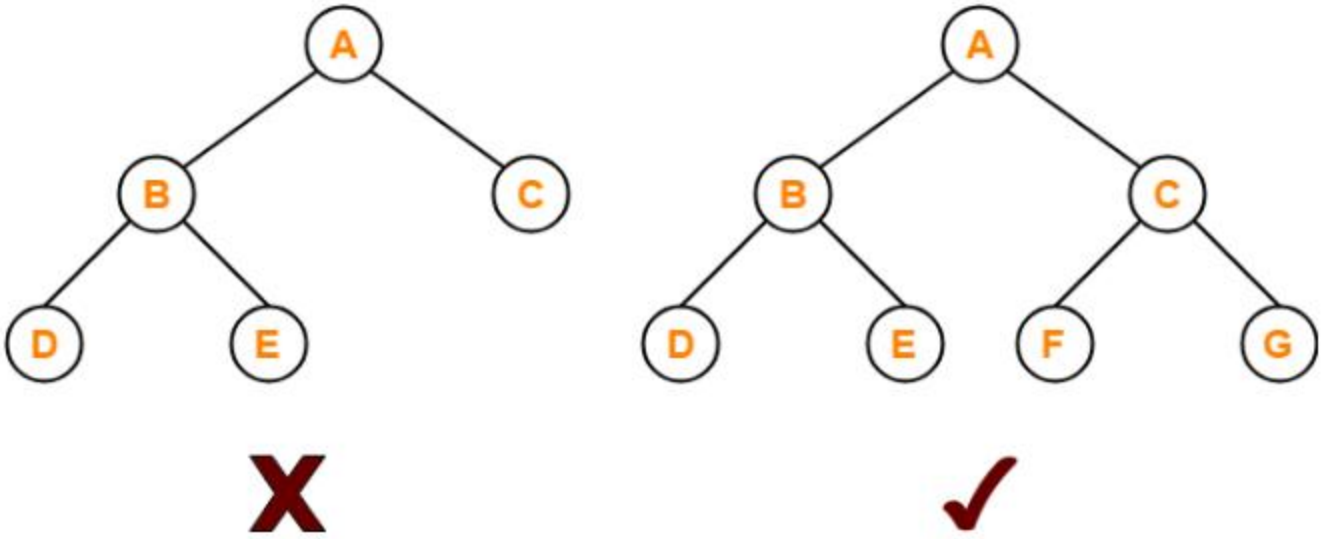
Full/Strictly Binary Tree

- A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.
- Full binary tree is also called as **Strictly binary tree**.



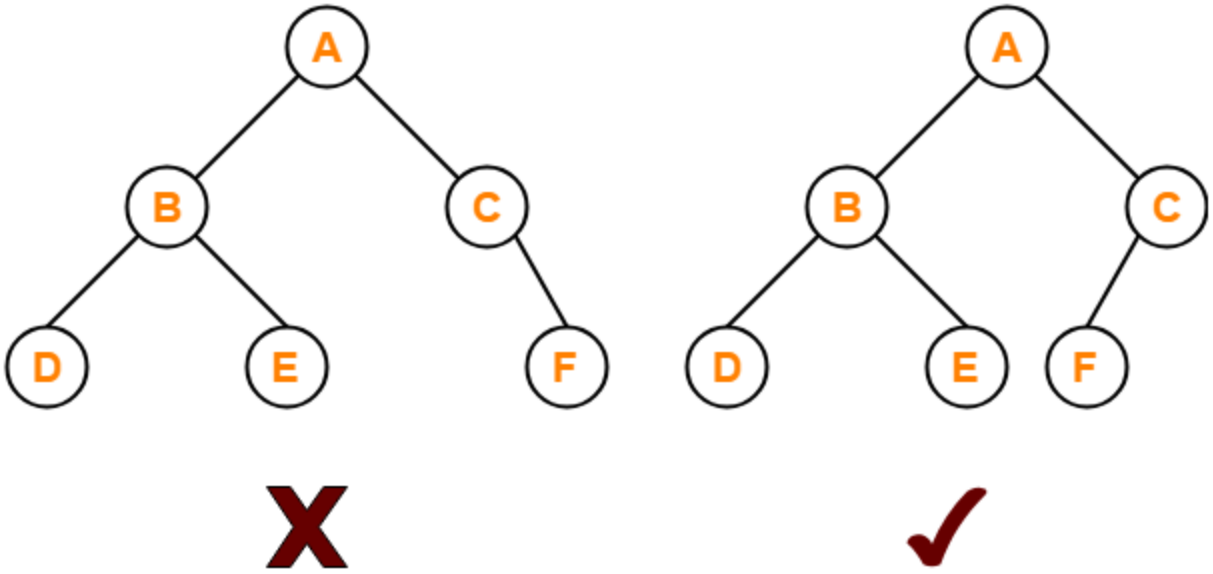
Complete /Perfect Binary Tree

- 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.



Almost Complete Binary Tree

- An **almost complete binary tree** is a binary tree that 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.

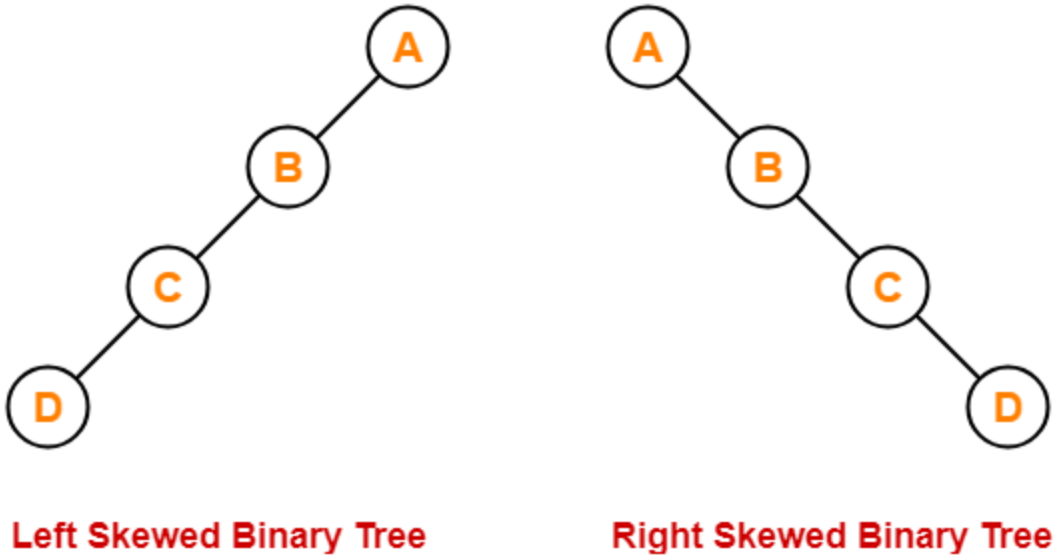


Skewed Binary Tree

- A **skewed binary tree** is a binary tree that 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).



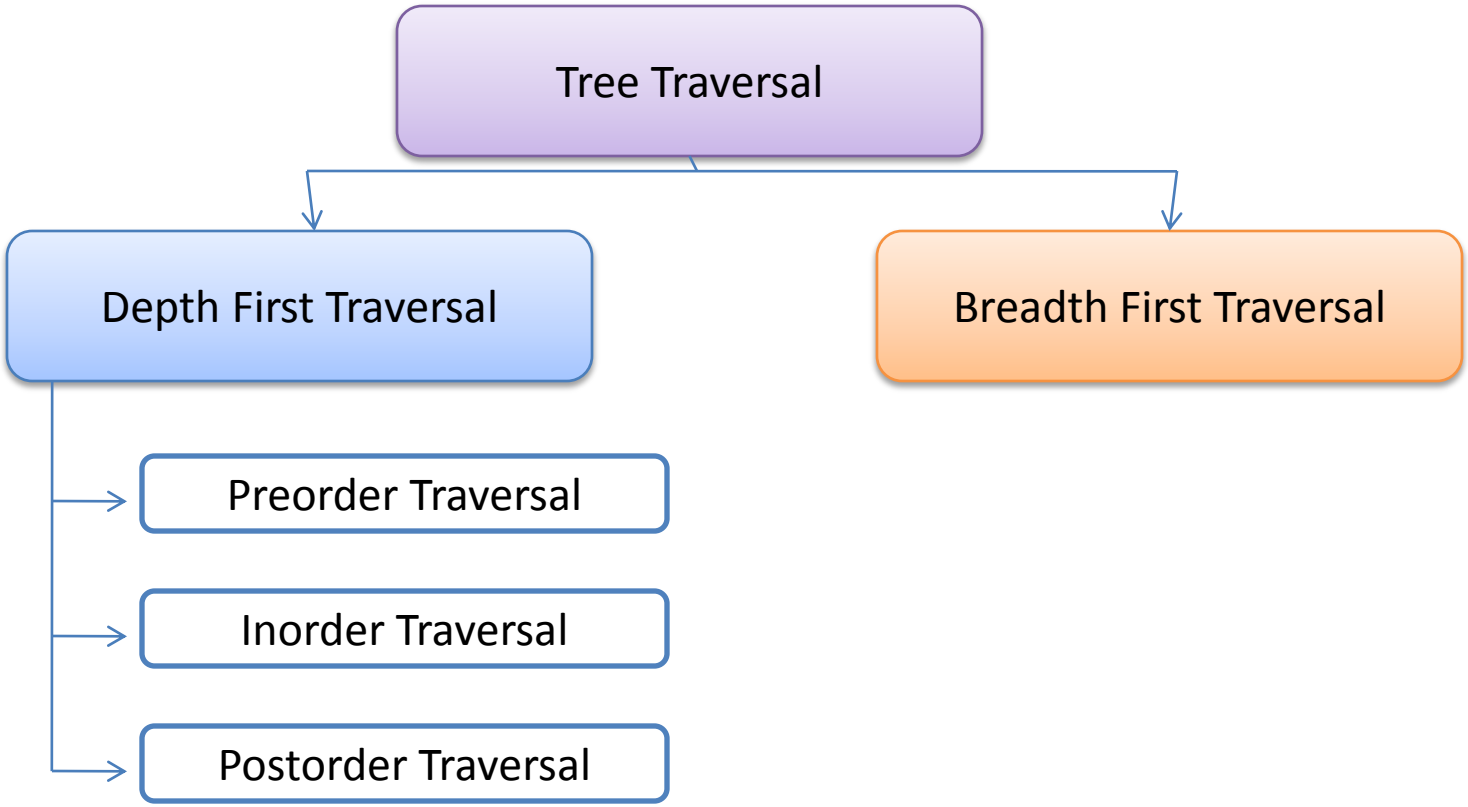
Left Skewed Binary Tree

Right Skewed Binary Tree

Tree Traversal

- In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.
- Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.

Tree traversal techniques

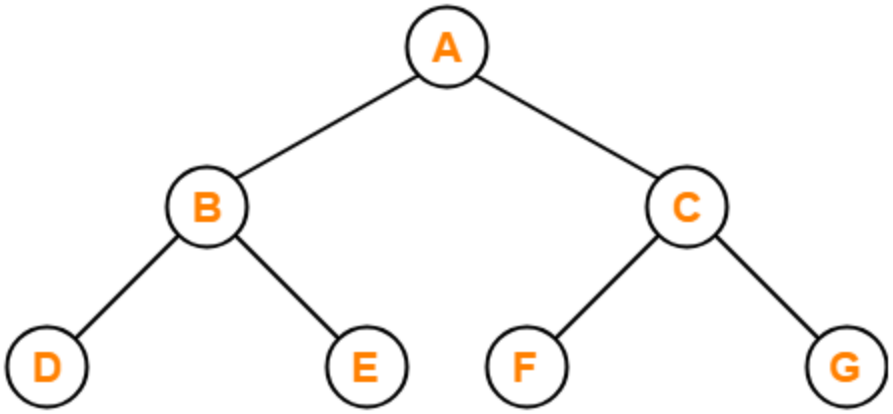


Depth First Traversal

- Following three traversal techniques fall under Depth First Traversal-
  1. Preorder Traversal
  2. Inorder Traversal
  3. Postorder Traversal

Preorder Traversal

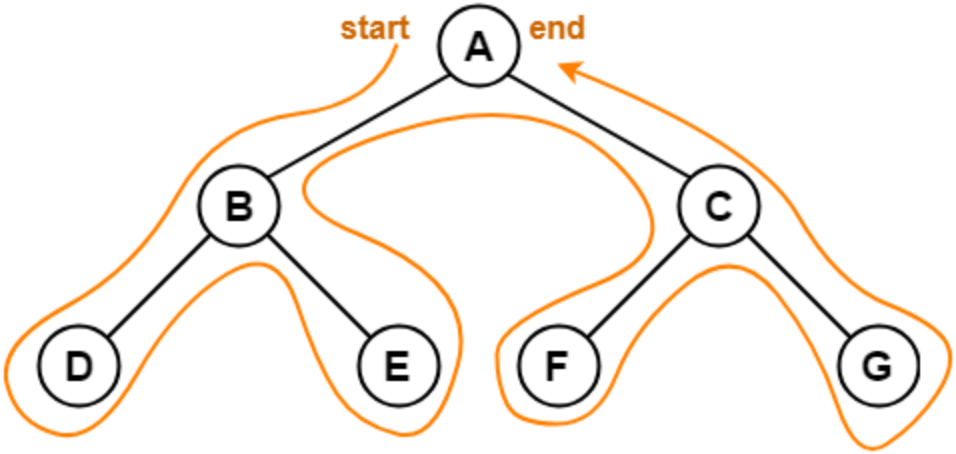
- **Algorithm-**
  - Visit the root
  - Traverse the left sub tree i.e. call Preorder (left sub tree)
  - Traverse the right sub tree i.e. call Preorder (right sub tree)



Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal Shortcut

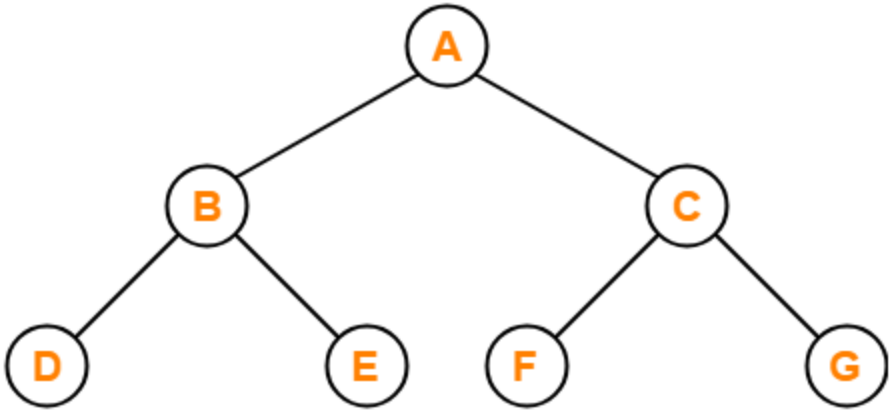
Traverse the entire tree starting from the root node keeping yourself to the left.



Preorder Traversal : A , B , D , E , C , F , G

Inorder Traversal

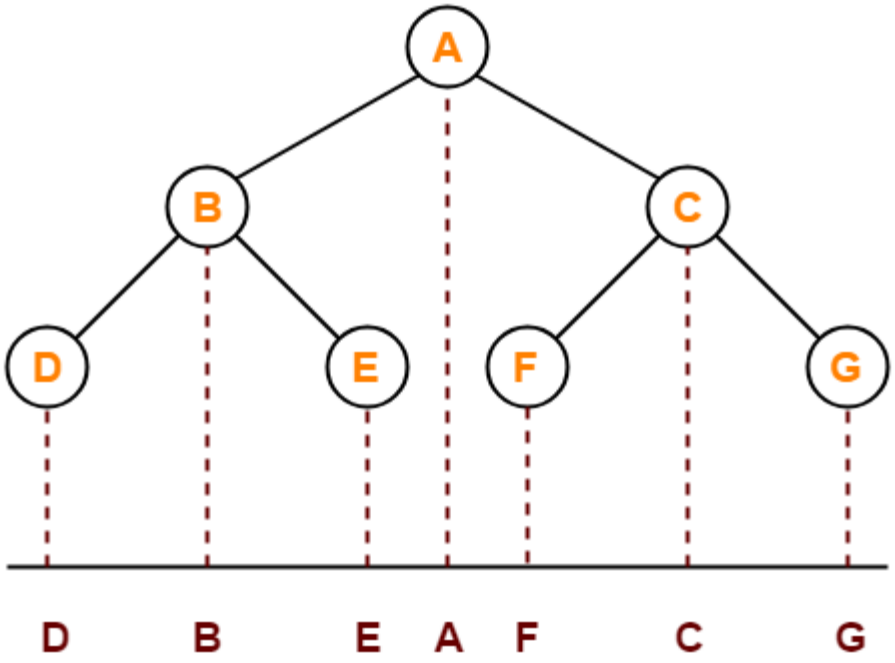
- **Algorithm-**
  - Traverse the left sub tree i.e. call Inorder (left sub tree)
  - Visit the root
  - Traverse the right sub tree i.e. call Inorder (right sub tree)



Inorder Traversal : D , B , E , A , F , C , G

Inorder Traversal Shortcut

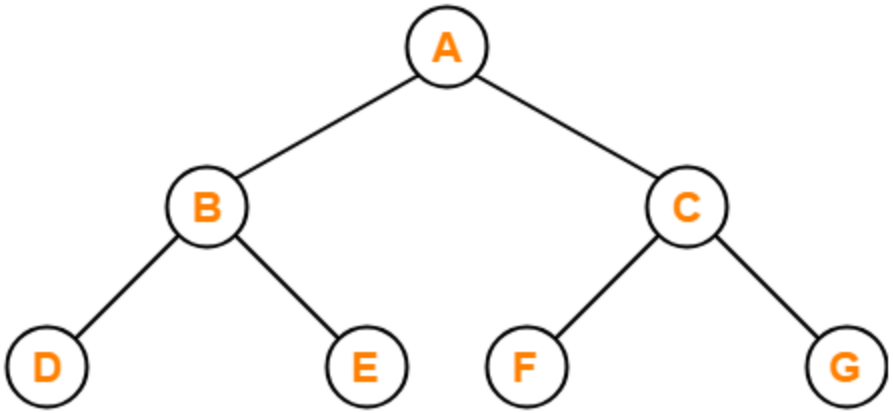
Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Inorder Traversal : D , B , E , A , F , C , G

Postorder Traversal

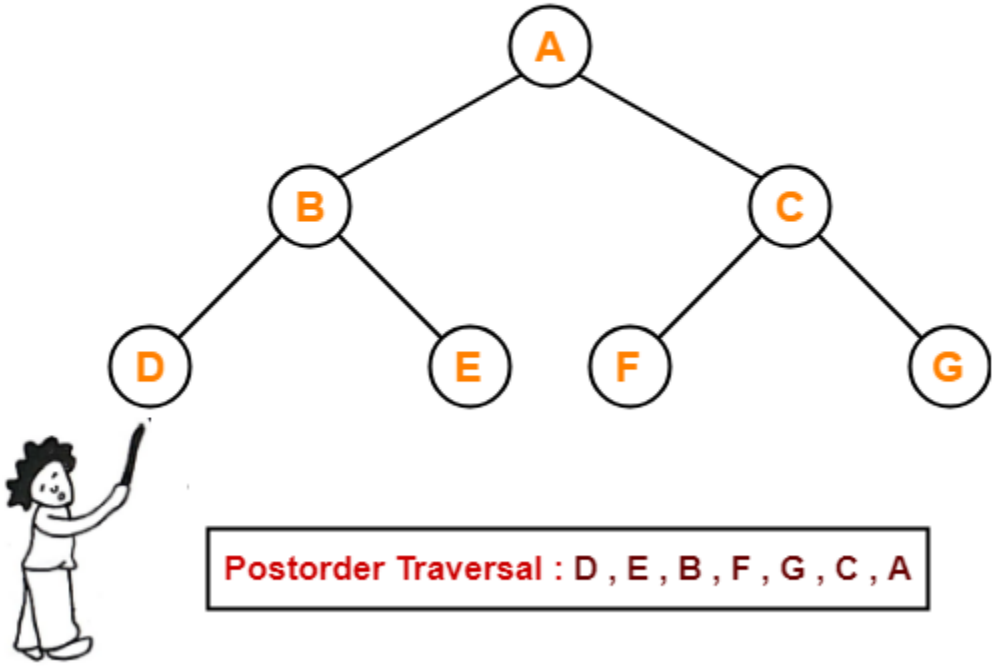
- **Algorithm-**
  - Traverse the left sub tree i.e. call Postorder (left sub tree)
  - Traverse the right sub tree i.e. call Postorder (right sub tree)
  - Visit the root



Postorder Traversal : D , E , B , F , G , C , A

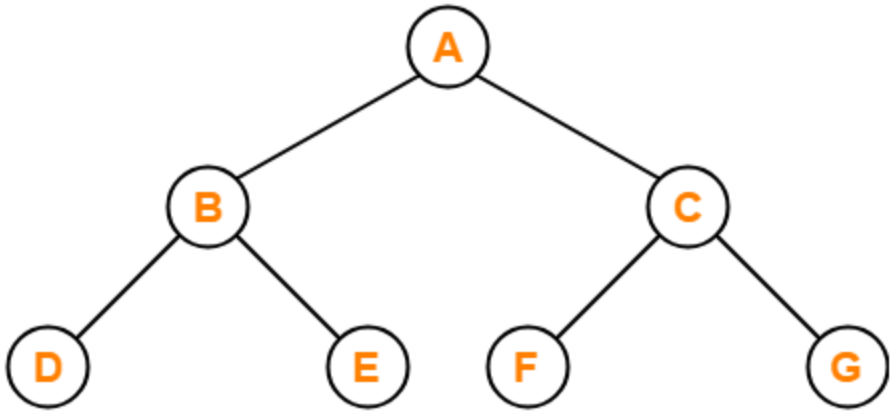
Postorder Traversal Shortcut

Pluck all the leftmost leaf nodes one by one.



Breadth First Search

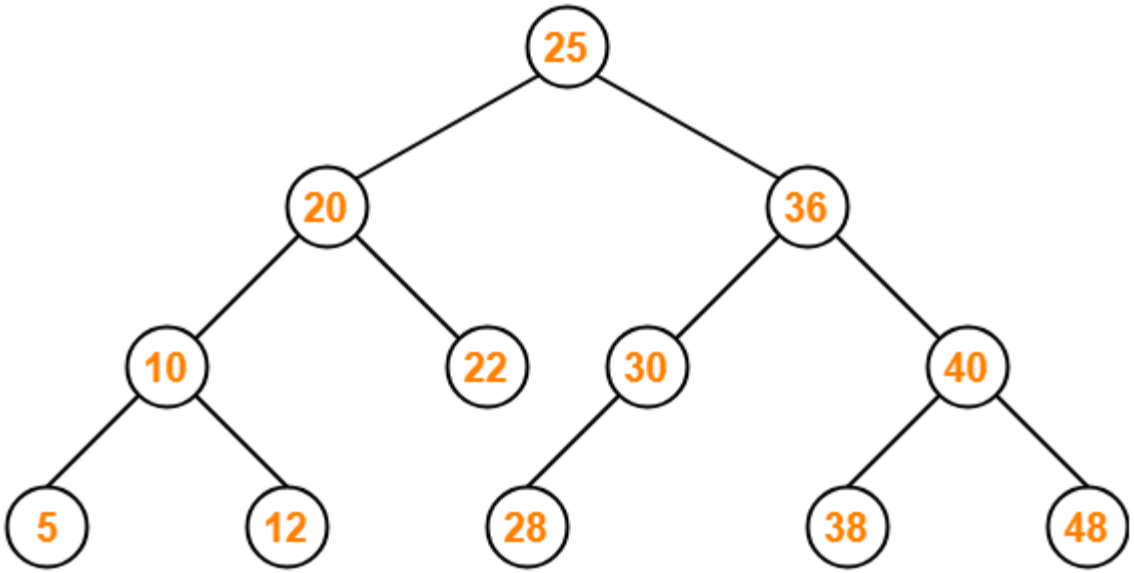
- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.



Level Order Traversal : A , B , C , D , E , F , G

Binary Search Tree construction

- In a binary search tree (BST), each node contains:
  - Only smaller values in its left sub tree
  - Only larger values in its right sub tree



BST construction

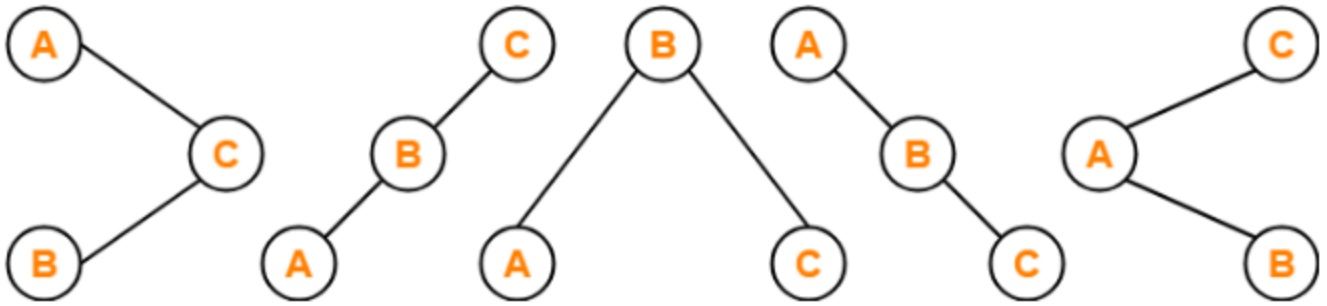
- Number of distinct binary search trees possible with 3 distinct Nodes

  $= {}^{2\times3}C_3 / 3+1$

  $= {}^{6}C_3 / 4$

  $= 5$

- If three distinct Nodes are A, B and C, then 5 distinct binary search trees are:



Example

- Construct a Binary Search Tree (BST) for the following sequence of numbers:

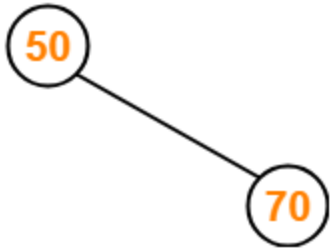
  50, 70, 60, 20, 90, 10, 40, 100

- When elements are given in a sequence,
  - Always consider the first element as the root node.
  - Consider the given elements and insert them in the BST one by one.

Insert 50-

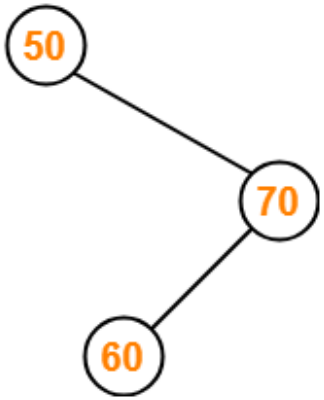
Insert 70-

- As 70 > 50, so insert 70 to the right of 50.



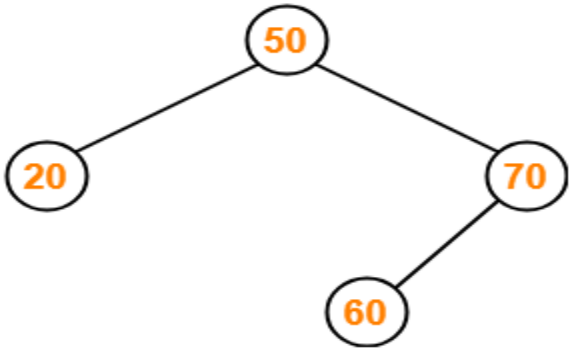
Insert 60-

- As 60 > 50, so insert 60 to the right of 50.
- As 60 < 70, so insert 60 to the left of 70.



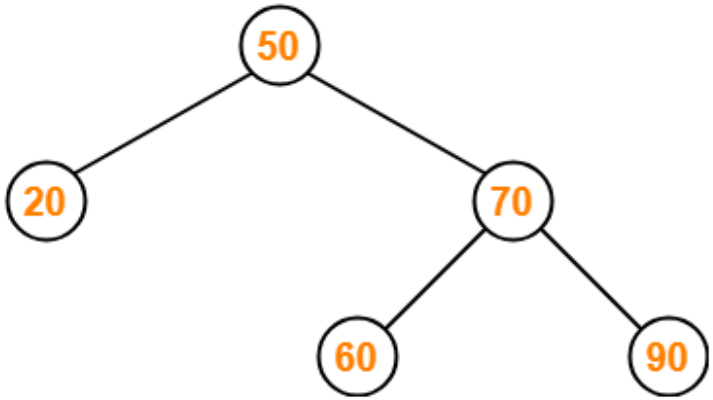
Insert 20-

- As 20 < 50, so insert 20 to the left of 50.



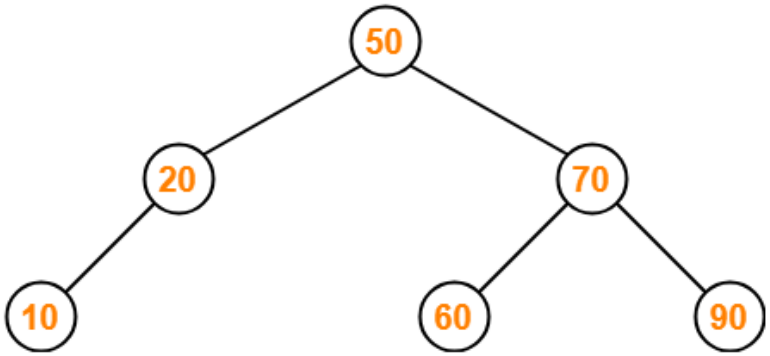
Insert 90-

- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.



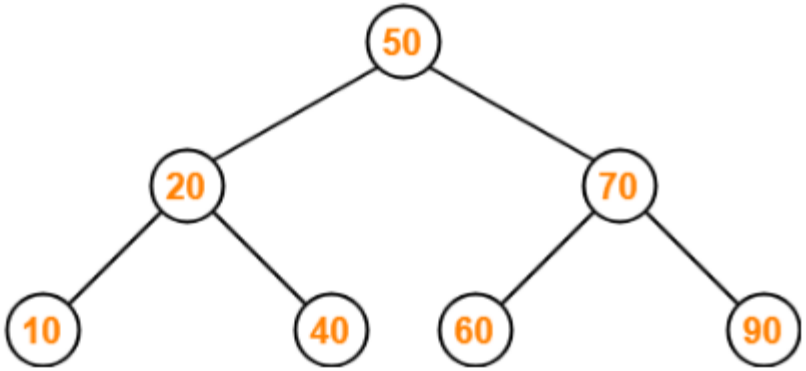
Insert 10-

- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



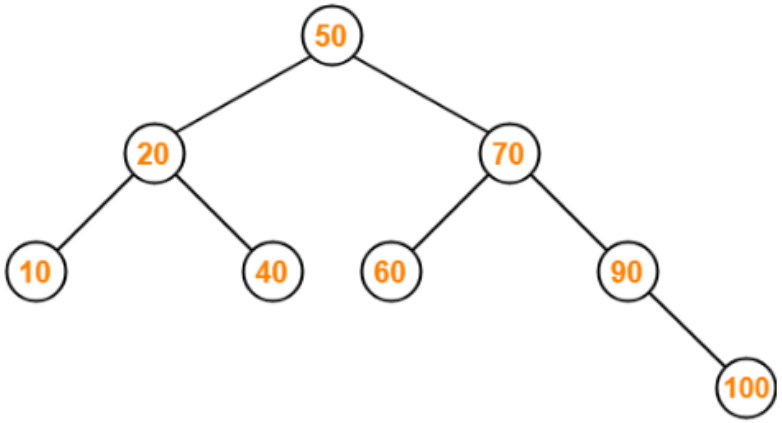
Insert 40-

- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.



Insert 100-

- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.



Practice problem

- **Problem-01:**
- A binary search tree is generated by inserting in order of the following integers-

  50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

- The number of nodes in the left subtree and right subtree of the root respectively is _____.
  - (4, 7)
  - (7, 4)
  - (8, 3)
  - (3, 8)

Practice problem

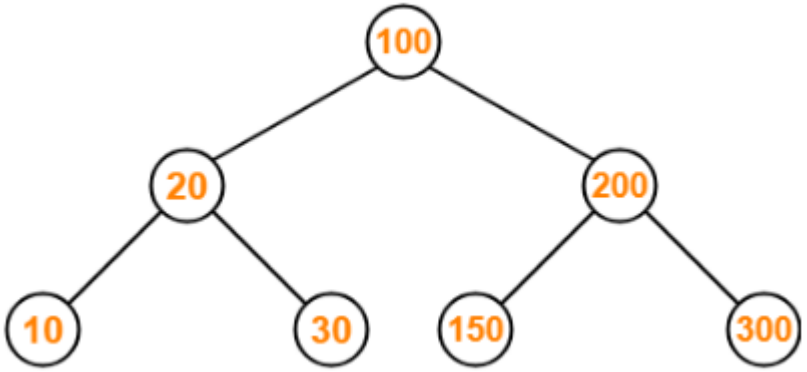
- **Problem-02:**
- How many distinct binary search trees can be constructed out of 4 distinct keys?
  - 5
  - 14
  - 24
  - 35

C code – BST construction

```
typedef struct BST
{
        int data;
        struct BST *left;
        struct BST *right;
}node;

node *create()
{
        node *temp;
        printf("nEnter data:");
        temp=(node*)malloc(sizeof(node));
        scanf("%d",&temp->data);
        temp->left=temp->right=NULL;
        return temp;
}

void insert(node *root,node *temp)
{
        if(temp->data<root->data)
        {
                if(root->left!=NULL)
                        insert(root->left,temp);
                else
                        root->left=temp;
        }

        if(temp->data>root->data)
        {
                if(root->right!=NULL)
                        insert(root->right,temp);
                else
                        root->right=temp;
        }
}
```

```
int main()
{
        char ch;
        node *root=NULL,*temp;

        do
        {
                temp=create();
                if(root==NULL)
                        root=temp;
                else
                        insert(root,temp);

                printf("nDo you want to enter more(y/n)?");
                getchar();
                scanf("%c",&ch);
        }while(ch=='y'|ch=='Y');

        printf("nPreorder Traversal: ");
        preorder(root);
        return 0;
```

BST Traversal



Preorder Traversal-

100 , 20 , 10 , 30 , 200 , 150 , 300

Inorder Traversal-

10 , 20 , 30 , 100 , 150 , 200 , 300

Postorder Traversal-

10 , 30 , 20 , 150 , 300 , 200 , 100

- Inorder traversal of a binary search tree always yields all the nodes in increasing order.

C Code – BST Traversal

```
Void inorder(node *root)
{
   if(root!=NULL)
   {
        inorder(root->left);
        printf("%d ",root->data);
        inorder(root->right);
   }
}
```

C Code – BST Traversal

```
void postorder(node *root)
{
   if(root!=NULL)
   {
        postorder(root->left);
        postorder(root->right);
        printf("%d ",root->data);
   }
}
```

C Code – BST Traversal

```
void preorder(node *root)
{
   if(root!=NULL)
   {
        printf("%d ",root->data);
        preorder(root->left);
        preorder(root->right);
   }
}
```

Practice

- Suppose the numbers 7 , 5 , 1 , 8 , 3 , 6 , 0 , 9 , 4 , 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers.
- What is the inorder traversal sequence of the resultant tree?

A. 7 , 5 , 1 , 0 , 3 , 2 , 4 , 6 , 8 , 9
B. 0 , 2 , 4 , 3 , 1 , 6 , 5 , 9 , 8 , 7
C. 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9
D. 9 , 8 , 6 , 4 , 2 , 3 , 0 , 1 , 5 , 7

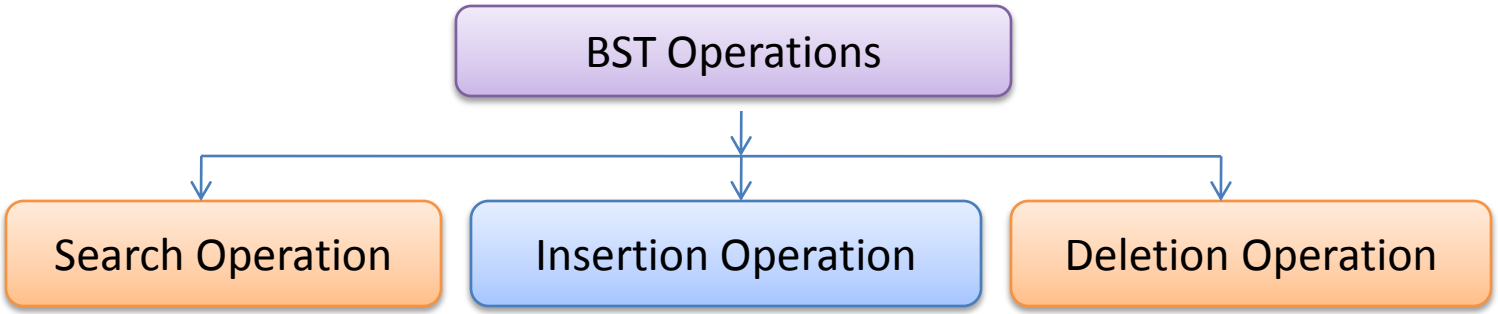
Problem-2

- The preorder traversal sequence of a binary search tree is-
  - 30 , 20 , 10 , 15 , 25 , 23 , 39 , 35 , 42
- Which one of the following is the postorder traversal sequence of the same tree?

A. 10 , 20 , 15 , 23 , 25 , 35 , 42 , 39 , 30
B. 15 , 10 , 25 , 23 , 20 , 42 , 35 , 39 , 30
C. 15 , 20 , 10 , 23 , 25 , 42 , 35 , 39 , 30
D. 15 , 10 , 23 , 25 , 20 , 35 , 42 , 39 , 30

BST Operations

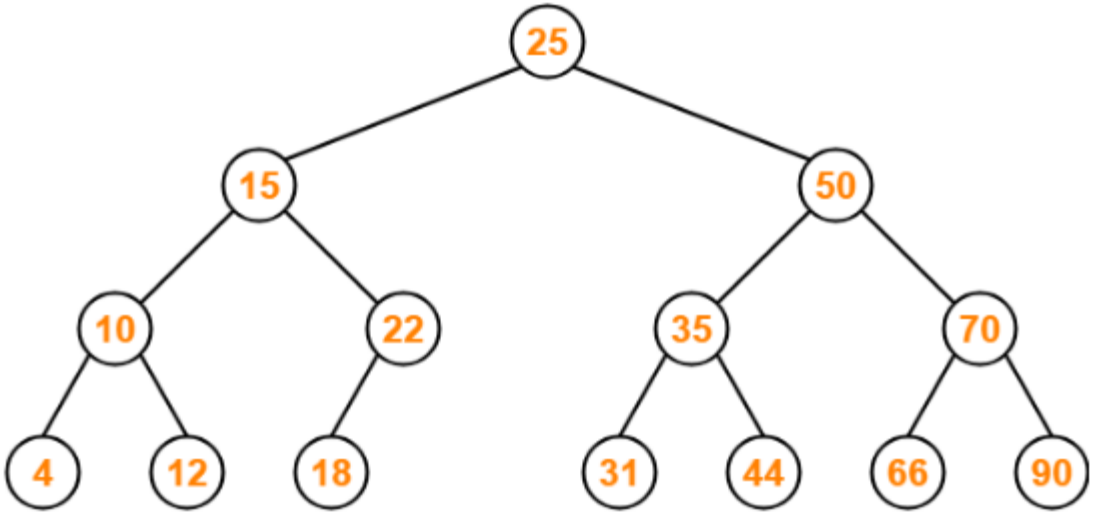
- Commonly performed binary search tree operations are:



Search Operation

- Search Operation is performed to search a particular element in the Binary Search Tree.
- For searching a given key in the BST,
  - Compare the key with the value of root node.
  - If the key is present at the root node, then return the root node.
  - If the key is greater than the root node value, then recur for the root node's right subtree.
  - If the key is smaller than the root node value, then recur for the root node's left subtree.

Consider key = 45 has to be searched in the given BST-



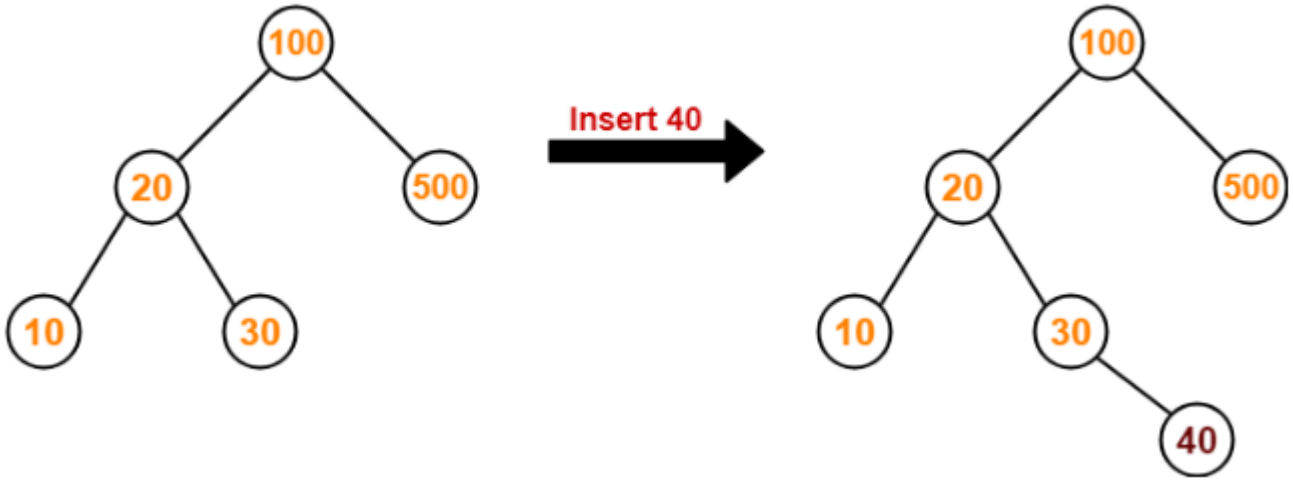
- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

C Code – BST Search Operation

```
struct node *search(struct node *node, int key) {
  // Return NULL if the tree is empty
  if (node == NULL) return NULL;
  if (node->key == key) return node->data;
  if (key < node->key)
    search(node->left, key);
  else
    search(node->right, key);
}
```

Insertion operation

- The insertion of a new key always takes place as the child of some leaf node.
- For finding out the suitable leaf node,
  - Search the key to be inserted from the root node till some leaf node is reached.
  - Once a leaf node is reached, insert the key as child of that leaf node.



- We start searching for value 40 from the root node 100.
- As 40 < 100, so we search in 100's left subtree.
- As 40 > 20, so we search in 20's right subtree.
- As 40 > 30, so we add 40 to 30's right subtree.

C Code – Insertion Operation

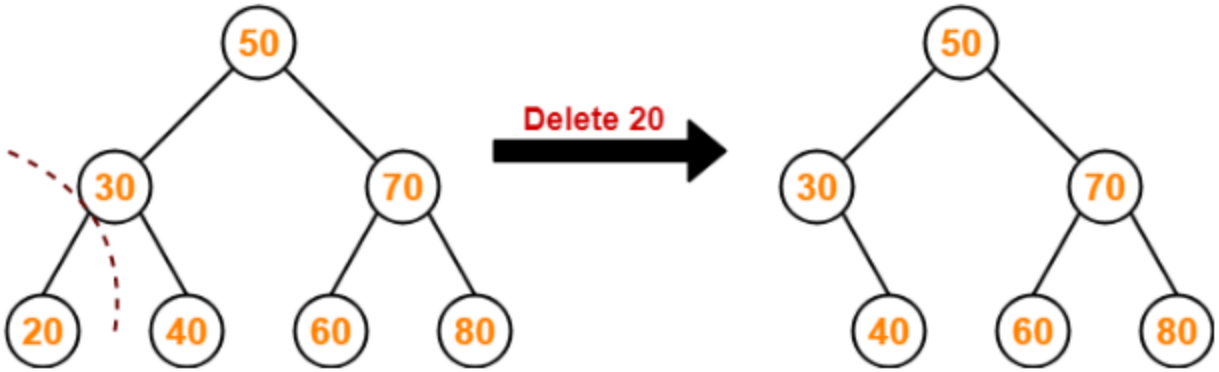
```
// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}
// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}
```

Deletion Operation

- Deletion Operation is performed to delete a particular element from the Binary Search Tree.
- When it comes to deleting a node from the binary search tree, three cases are possible.

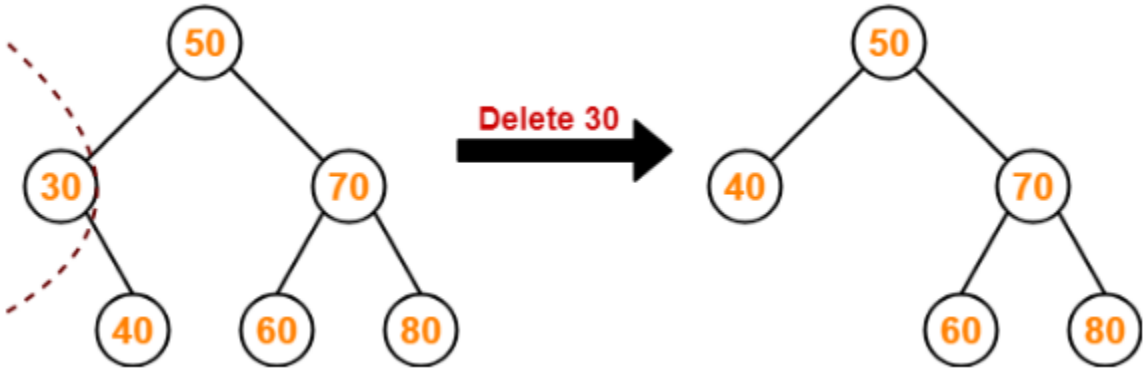
Case-01: Deletion Of A Node Having No Child (Leaf Node)

- Just remove / disconnect the leaf node that is to deleted from the tree.



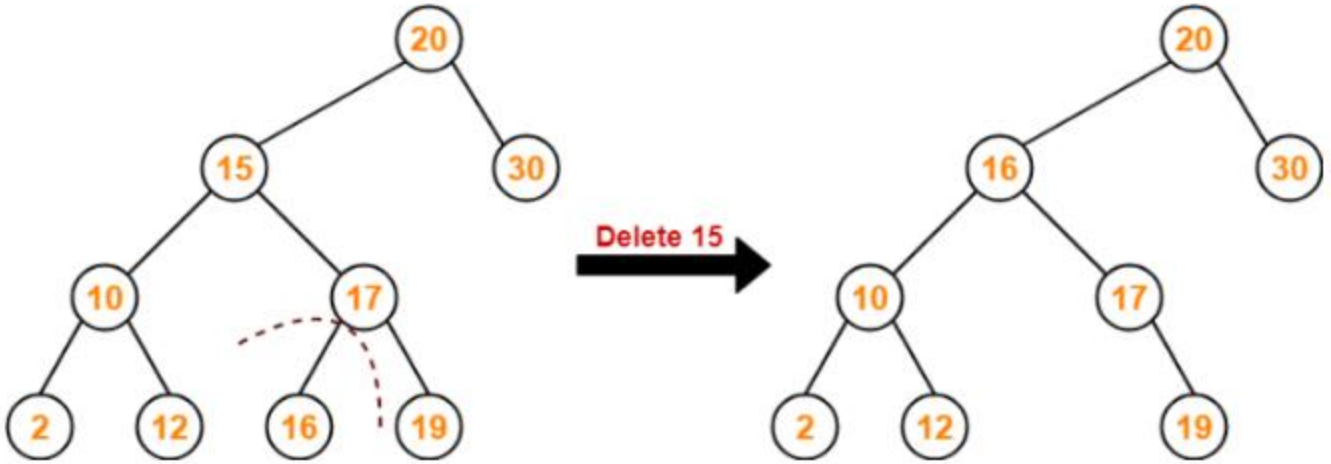
Case-02: Deletion Of A Node Having Only One Child

- Consider the following example where node with value = 30 is deleted from the BST.

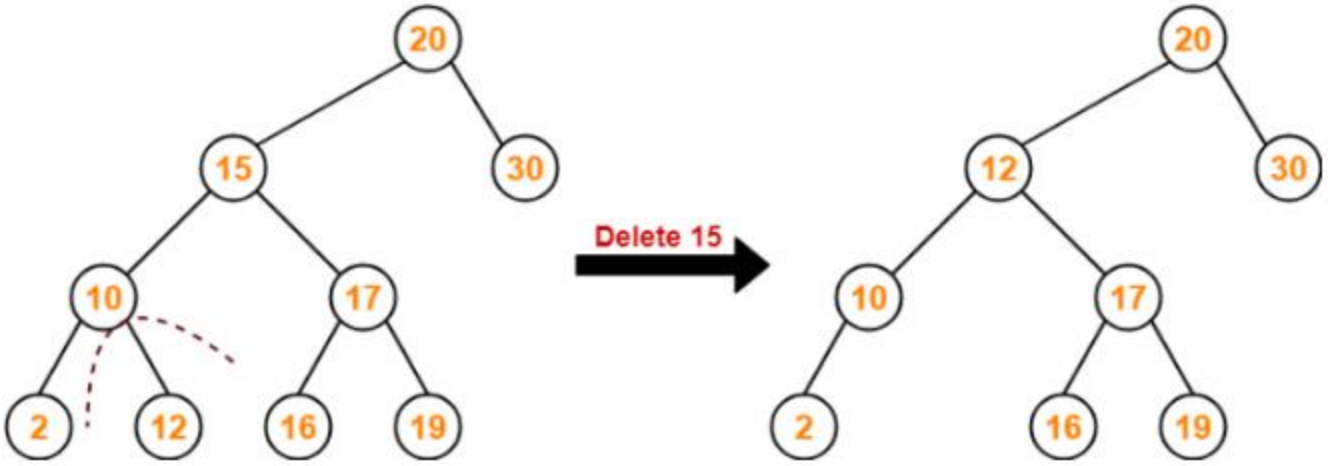


Case-03: Deletion Of A Node Having Two Children

- Consider the following example where node with value = 15 is deleted from the BST
- Method-1:
  - Visit to the right subtree of the deleting node.
  - Pluck the least value element called as inorder successor.
  - Replace the deleting element with its inorder successor.



- Method-2:
  - Visit to the left subtree of the deleting node.
  - Pluck the greatest value element called as inorder successor.
  - Replace the deleting element with its inorder successor.



C code – BST deletion

```
//Function to find minimum in a tree.
Node* FindMin(Node* root)
{
        while(root->left != NULL) root = root->left;
        return root;
}

// Function to search a delete a value from tree.
struct Node* Delete(struct Node *root, int data) {
        if(root == NULL) return root;
        else if(data < root->data) root->left = Delete(root->left,data);
        else if (data > root->data) root->right = Delete(root->right,data);
        else {
                // Case 1:  No child
                if(root->left == NULL && root->right == NULL) {
                        free(root);
                        root = NULL;
                }
                //Case 2: One child
                else if(root->left == NULL) {
                        struct Node *temp = root;
                        root = root->right;
                        free(temp);
                }
                else if(root->right == NULL) {
                        struct Node *temp = root;
                        root = root->left;
                        free(temp);
                }
                // case 3: 2 children
                else {
                        struct Node *temp = FindMin(root->right);
                        root->data = temp->data;
                        root->right = Delete(root->right,temp->data);
                }
        }
        return root;
}
```

Hashing

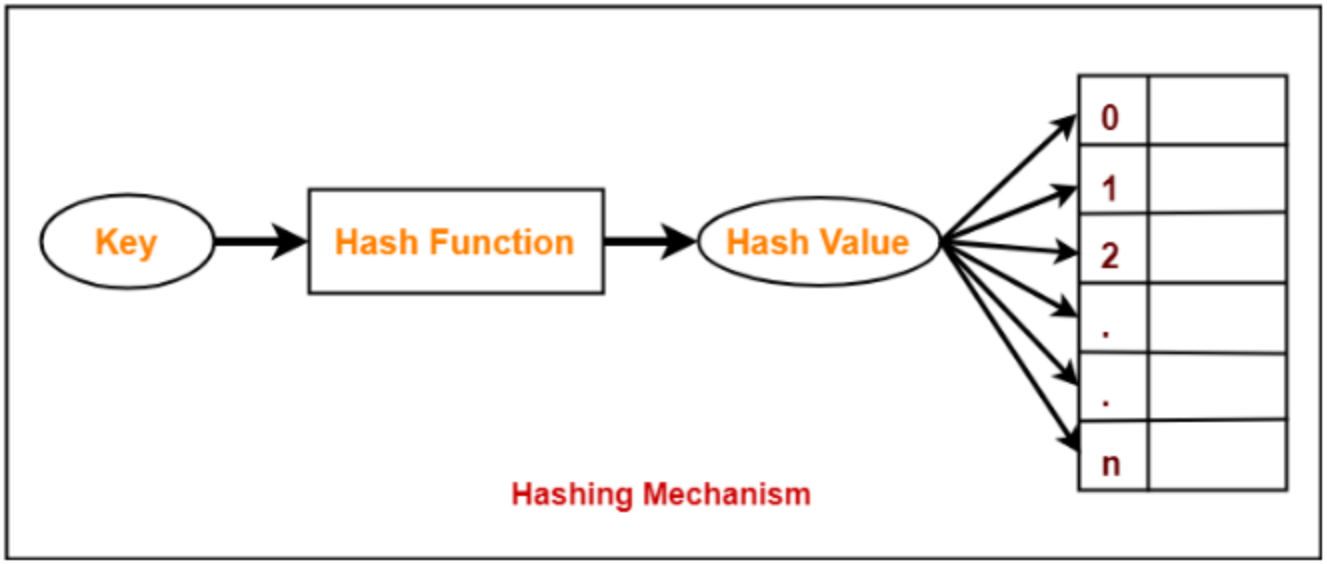
- Hashing is a well-known technique to search any particular element among several elements.
- It minimizes the number of comparisons while performing the search.
- Unlike other searching techniques,
  - Hashing is extremely efficient.
  - The time taken by it to perform the search does not depend upon the total number of elements.
  - It completes the search with constant time complexity O(1).

Hashing Mechanism

- An array data structure called as **Hash table** is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.

Hash Key value

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be be stored in the hash table.
- Hash key value is generated using a hash function.



Hash Function

- Hash function is a function that maps any big number or string to a small integer value.
- Hash function takes the data item as an input and returns a small integer value as an output.
- The small integer value is called as a hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.

Types of Hash Functions

- There are various types of hash functions available such as-
- Mid Square Hash Function
- Division Hash Function
- Folding Hash Function etc

- It depends on the user which hash function wants to use.

Properties of Hash Function

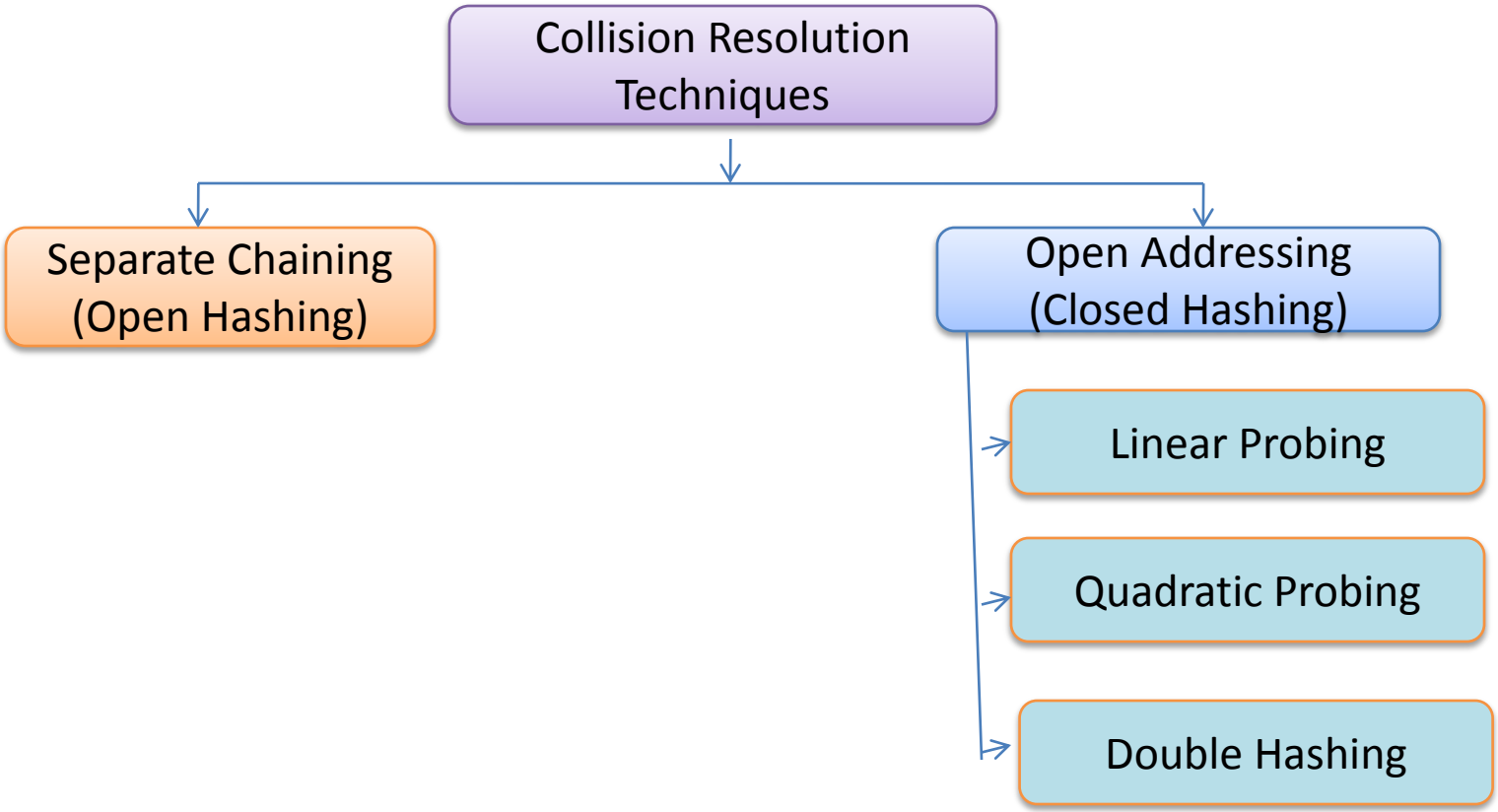
- The properties of a good hash function are-
  - It is efficiently computable.
  - It minimizes the number of collisions.
  - It distributes the keys uniformly over the table.

Collision in Hashing

- Hash function is used to compute the hash value for a key.
- Hash value is then used as an index to store the key in the hash table.
- Hash function may return the same hash value for two or more keys.
- When the hash value of a key maps to an already occupied bucket of the hash table, it is called as a **Collision**.

Collision Resolution Techniques

- Collision Resolution Techniques are the techniques used for resolving or handling the collision.



Separate Chaining

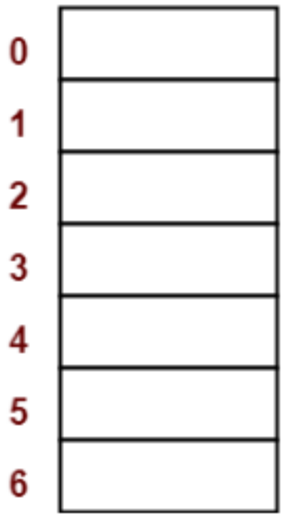
- To handle the collision,
  - This technique creates a linked list to the slot for which collision occurs.
  - The new key is then inserted in the linked list.
  - These linked lists to the slots appear like chains.
  - That is why, this technique is called as **separate chaining**.

Example-Separate Chaining

- Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-
- 50, 700, 76, 85, 92, 73 and 101

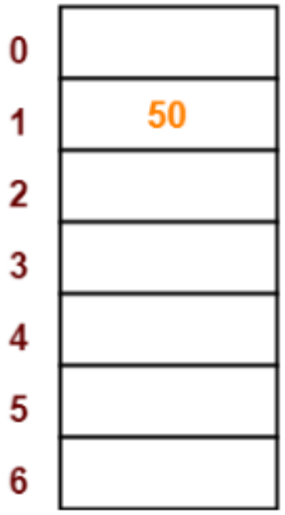
Step-1

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



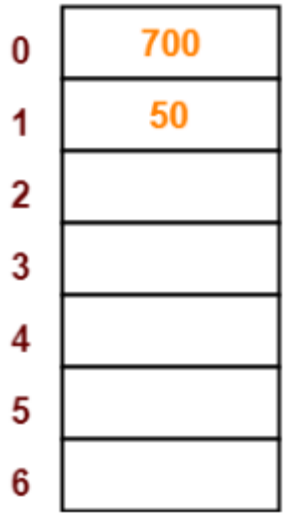
Step-2

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



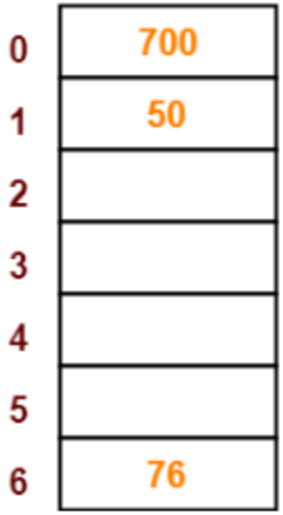
Step-3

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



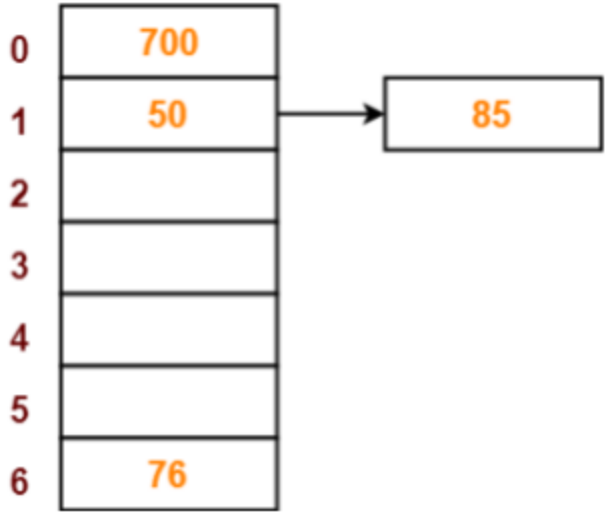
Step-4

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



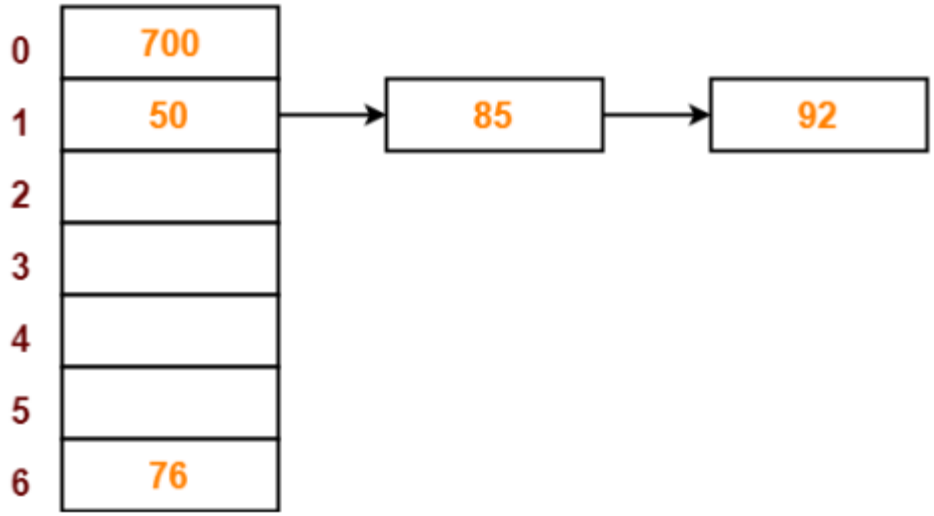
Step-5

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



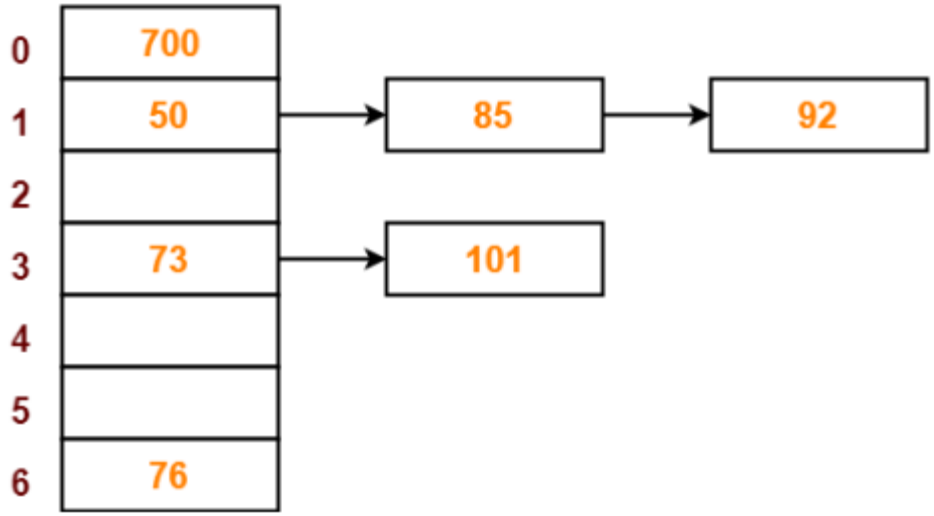
Step-6

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



Step-7

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-



Open Addressing

- In open addressing,
  - Unlike separate chaining, all the keys are stored inside the hash table.
  - No key is stored outside the hash table.

- Techniques used for open addressing are-
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Operations in Open Addressing

- Insert Operation:
  - Hash function is used to compute the hash value for a key to be inserted.
  - Hash value is then used as an index to store the key in the hash table.

- In case of collision,
  - Probing is performed until an empty bucket is found.
  - Once an empty bucket is found, the key is inserted.
  - Probing is performed in accordance with the technique used for open addressing.

Open Addressing

- Search Operation:
- To search any particular key,
  - Its hash value is obtained using the hash function used.
  - Using the hash value, that bucket of the hash table is checked.
  - If the required key is found, the key is searched.
  - Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
  - The empty bucket indicates that the key is not present in the hash table.

Open Addressing

- Delete Operation:
  - The key is first searched and then deleted.
  - After deleting the key, that particular bucket is marked as “deleted”.

1. Linear Probing

- In linear probing,
  - When collision occurs, we linearly probe for the next bucket.
  - We keep probing until an empty bucket is found.

- **Advantage-**

  - It is easy to compute.

- **Disadvantage-**

  - The main problem with linear probing is clustering.
  - Many consecutive elements form groups.
  - Then, it takes time to search an element or to find an empty bucket.

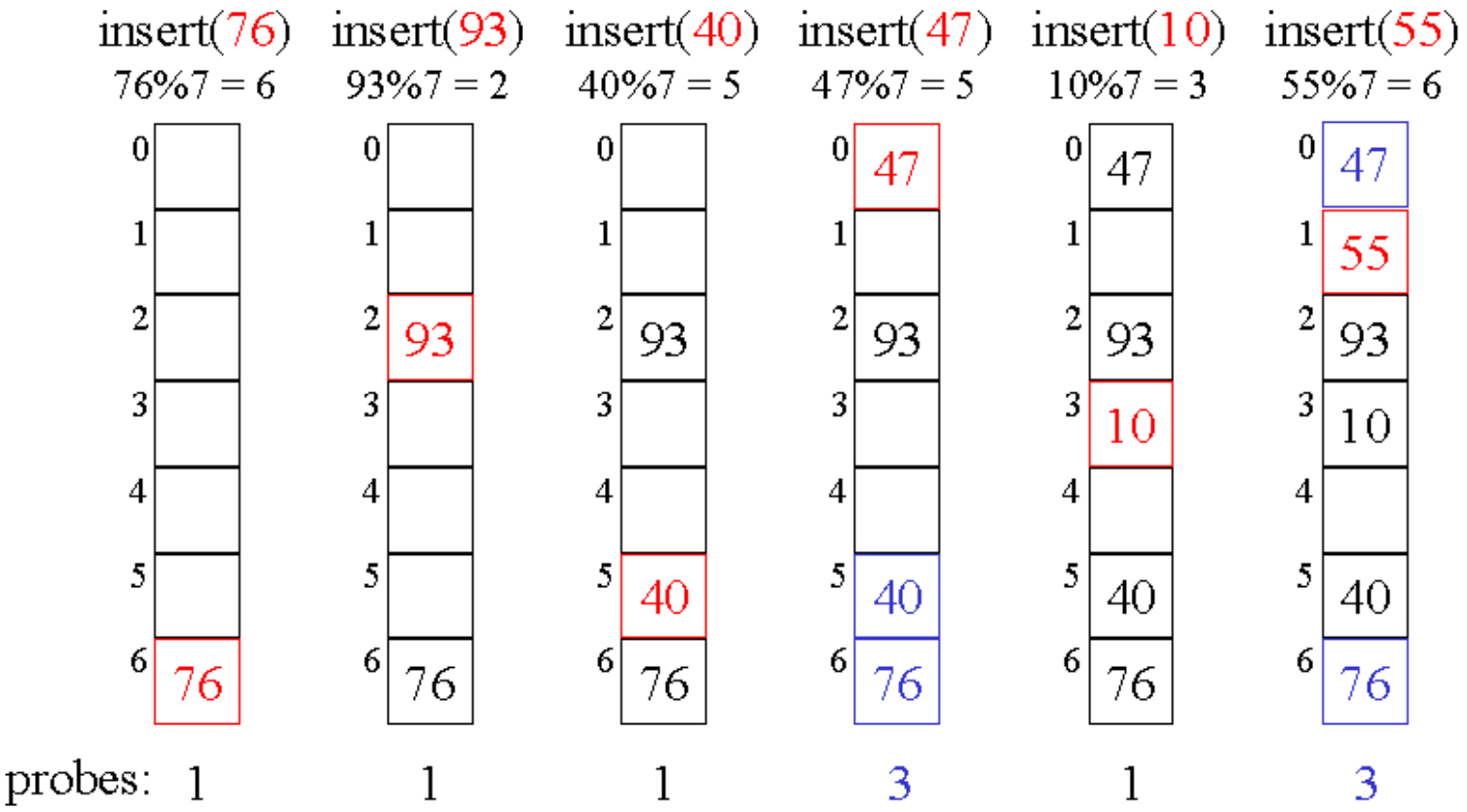
Linear Probing

$$f(i) = i$$

- Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - …
- findEntry using linear probing:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k);
  do {
    entry = &table[probePoint];
    probePoint = (probePoint + 1) % size;
  } while (!entry->isEmpty() && entry->key != k);
  return !entry->isEmpty();
}
```

Example- Linear Probing



2. Quadratic Probing

- In quadratic probing,
- When collision occurs, we probe for i^2‘th bucket in i^{th} iteration.
- We keep probing until an empty bucket is found.

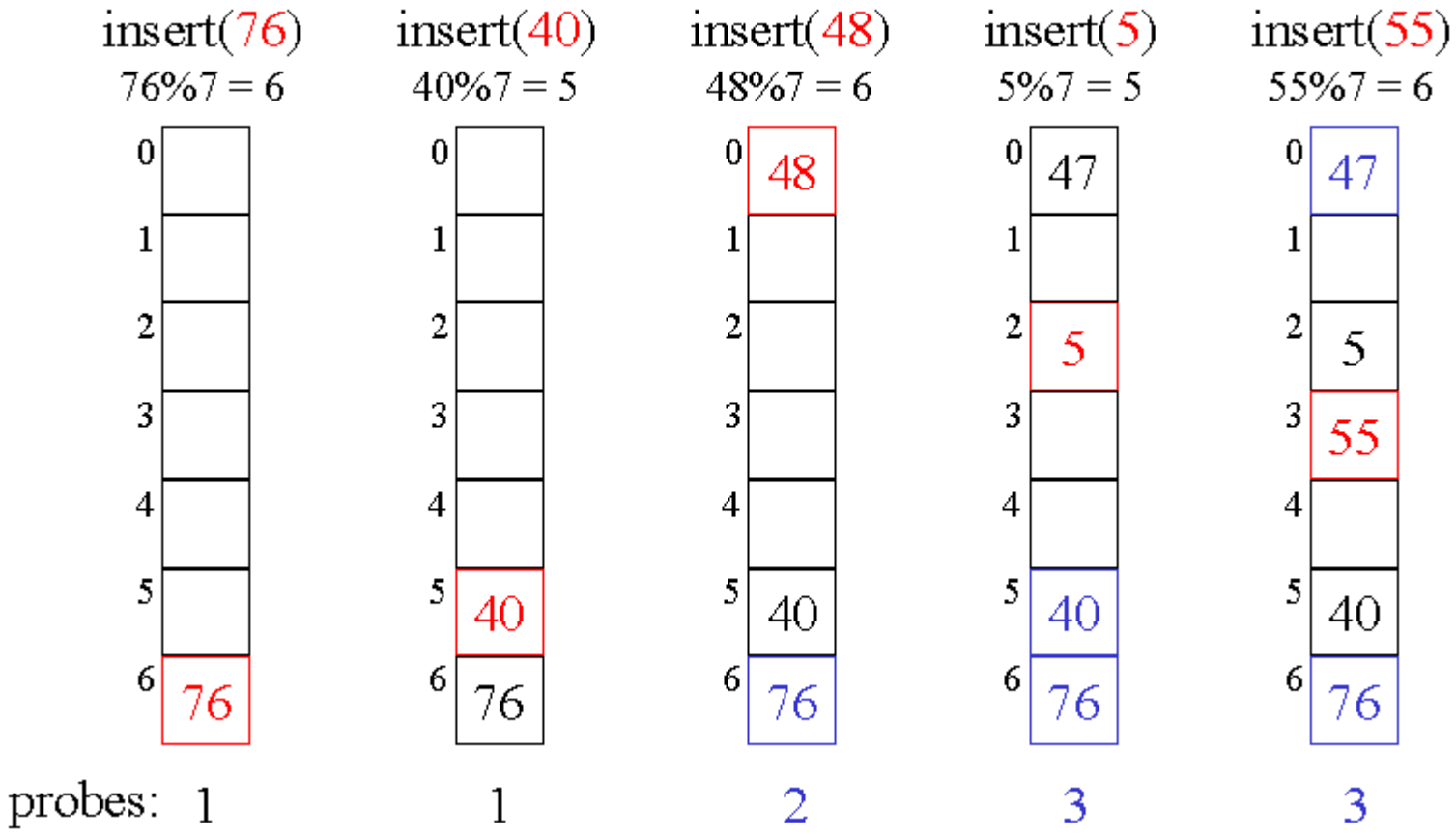
Quadratic Probing

$$f(i) = i^2$$

- Probe sequence is
  - h(k) mod size
  - (h(k) + 1) mod size
  - (h(k) + 4) mod size
  - (h(k) + 9) mod size
  - …
- findEntry using quadratic probing:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k), numProbes = 0;
  do {
    entry = &table[probePoint];
    numProbes++;
    probePoint = (probePoint + 2*numProbes - 1) % size;
  } while (!entry->isEmpty() && entry->key != key);
  return !entry->isEmpty();
}
```

Example – Quadratic Probing



Double Hashing

- In double hashing,
- We use another hash function hash2(x) and look for i * hash2(x) bucket in i^{th} iteration.
- It requires more computation time as two hash functions need to be computed.

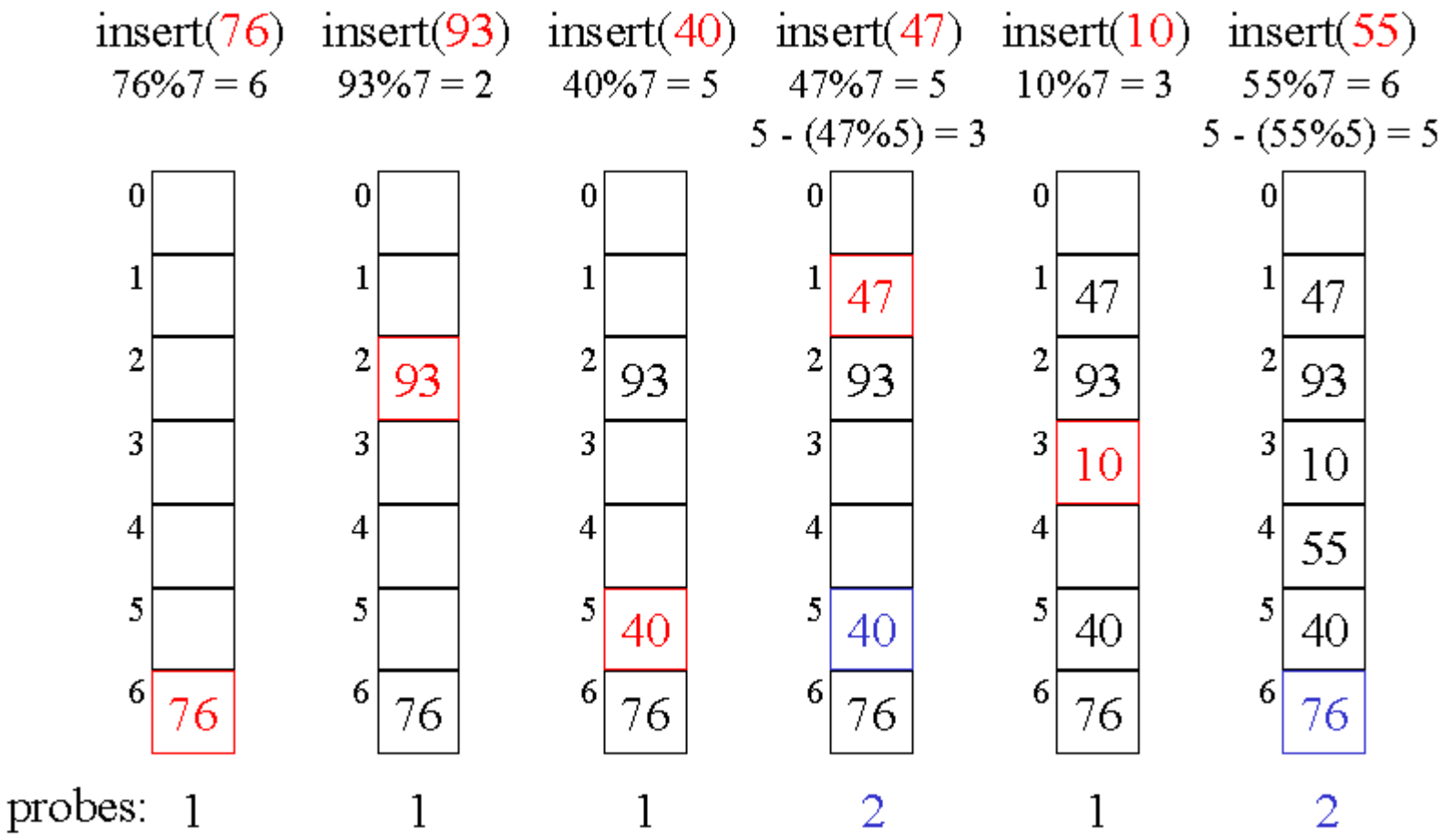
Double Hashing

$$f(i) = i \cdot hash_2(x)$$

- Probe sequence is
  - $h_1(k) \bmod size$
  - $(h_1(k) + 1 \cdot h_2(x)) \bmod size$
  - $(h_1(k) + 2 \cdot h_2(x)) \bmod size$
  - …
- Code for finding the next linear probe:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k), hashIncr = hash2(k);
  do {
    entry = &table[probePoint];
    probePoint = (probePoint + hashIncr) % size;
  } while (!entry->isEmpty() && entry->key != k);
  return !entry->isEmpty();
}
```

Example –Double Hashing



Separate chaining Vs Open Addressing

Separate Chaining	Open Addressing
Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is present outside the hash table.
The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
Deletion is easier.	Deletion is difficult.
Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.
Some buckets of the hash table are never used which leads to wastage of space.	Buckets may be used even if no key maps to those particular buckets.

Comparison of Open Addressing Techniques

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Example

```c
#include<stdio.h>

#define size 7

int arr[size];


void init()
{
    int i;
    for(i = 0; i < size; i++)
        arr[i] = -1;
}

void insert(int value)
{
    int key = value % size;

    if(arr[key] == -1)
    {
        arr[key] = value;
        printf("%d inserted at arr[%d]\n", value,key);
    }
    else
    {
        printf("Collision : arr[%d] has element %d already!\n",ke
        printf("Unable to insert %d\n",value);
    }
}
```

```c
void del(int value)
{
    int key = value % size;
    if(arr[key] == value)
        arr[key] = -1;
    else
        printf("%d not present in the hash table\n",value);
}

void search(int value)
{
    int key = value % size;
    if(arr[key] == value)
        printf("Search Found\n");
    else
        printf("Search Not Found\n");
}

void print()
{
    int i;
    for(i = 0; i < size; i++)
        printf("arr[%d] = %d\n",i,arr[i]);
}
```

```c
int main()
{
    init();
    insert(10); //key = 10 % 7 ==> 3
    insert(4);  //key = 4 % 7  ==> 4
    insert(2);  //key = 2 % 7  ==> 2
    insert(3);  //key = 3 % 7  ==> 3 (collision)

    printf("Hash table\n");
    print();
    printf("\n");

    printf("Deleting value 10..\n");
    del(10);
    printf("After the deletion hash table\n");
    print();
    printf("\n");

    printf("Deleting value 5..\n");
    del(5);
    printf("After the deletion hash table\n");
    print();
    printf("\n");

    printf("Searching value 4..\n");
    search(4);
    printf("Searching value 10..\n");
    search(10);

    return 0;
}
```

Questions

- implementation to find leaf count of a given Binary tree

```
int getLeafCount(Node node)
{
    if (node == null)
        return 0;
    if (node.left == null && node.right == null)
        return 1;
    else
        return getLeafCount(node.left) +
 getLeafCount(node.right);
}
```

- *counting the number of nodes in a Tree*

```
int countnodes(struct node *root)
{
if(root != NULL)
{
countnodes(root->left);
count++;
countnodes(root->right);
}
return count;
}
```

```
// Function to find k'th smallest element in BST
// Here i denotes the number of nodes processed so far
int kthSmallest(Node* root, int *i, int k)
{
    // base case
    if (root == nullptr)
        return INT_MAX;

    // search in left subtree
    int left = kthSmallest(root->left, i, k);

    // if k'th smallest is found in left subtree, return it
    if (left != INT_MAX)
        return left;

    // if current element is k'th smallest, return its value
    if (++*i == k)
        return root->data;

    // else search in right subtree
    return kthSmallest(root->right, i, k);
}

// Function to find k'th smallest element in BST
int kthSmallest(Node* root, int k)
{
    // maintain index to count number of nodes processed so far
    int i = 0;

    // traverse the tree in in-order fashion and return k'th element
    return kthSmallest(root, &i, k);
}
```

```
// Function to determine if given Binary Tree is a BST or not by keeping a
// valid range (starting from [MIN_VALUE, MAX_VALUE]) and keep shrinking
// it down for each node as we go down recursively
bool isBST(Node* node, int minKey, int maxKey)
{
   // base case
   if (node == NULL)
      return true;

   // if node's value fall outside valid range
   if (node->data < minKey || node->data > maxKey)
      return false;

   // recursively check left and right subtrees with updated range
   return isBST(node->left, minKey, node->data) &&
          isBST(node->right, node->data, maxKey);
}
```

```
int maxDepth(struct node* node)
{
  if (node==NULL)
     return 0;
  else
  {
     /* compute the depth of each subtree */
     int lDepth = maxDepth(node->left);
     int rDepth = maxDepth(node->right);

     /* use the larger one */
     if (lDepth > rDepth)
        return(lDepth+1);
     else return(rDepth+1);
  }
}
```

```
Input: inorder[] = {5, 10, 40, 30, 28}
Output: root of following tree
         40
       /   \
      10     30
     /         \
    5           28

Input: inorder[] = {1, 5, 10, 40, 30,
                    15, 28, 20}
Output: root of following tree
          40
        /   \
       10     30
      /         \
     5           28
    /           /  \
   1          15    20
```

For example, if the given traversal is {1, 7, 5, 50, 40, 10}, then following tree should be constructed and root of the tree should be returned.

```
     10
   /   \
  5     40
 /  \      \
1    7      50
```

```
           10
         /    \
        /      \
{1, 7, 5}       {50, 40}
```