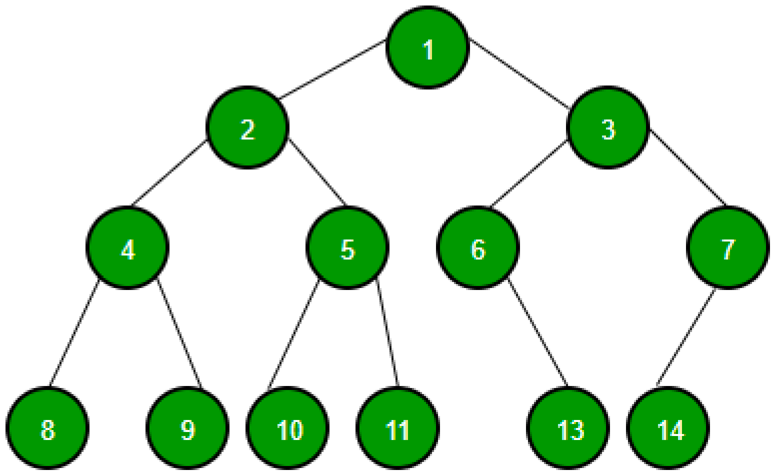
Binary Tree

What is Binary Tree key Structure?

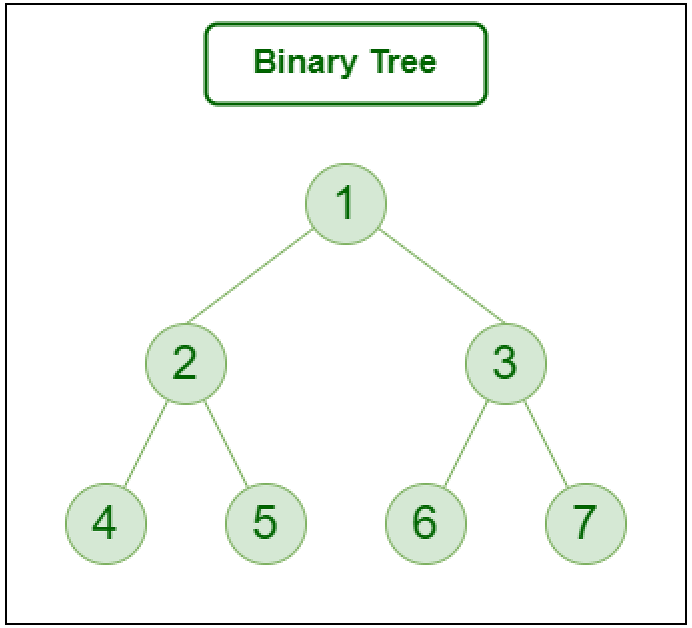
- Binary Tree is defined as a tree key structure where each node has at most 2 children.
- Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



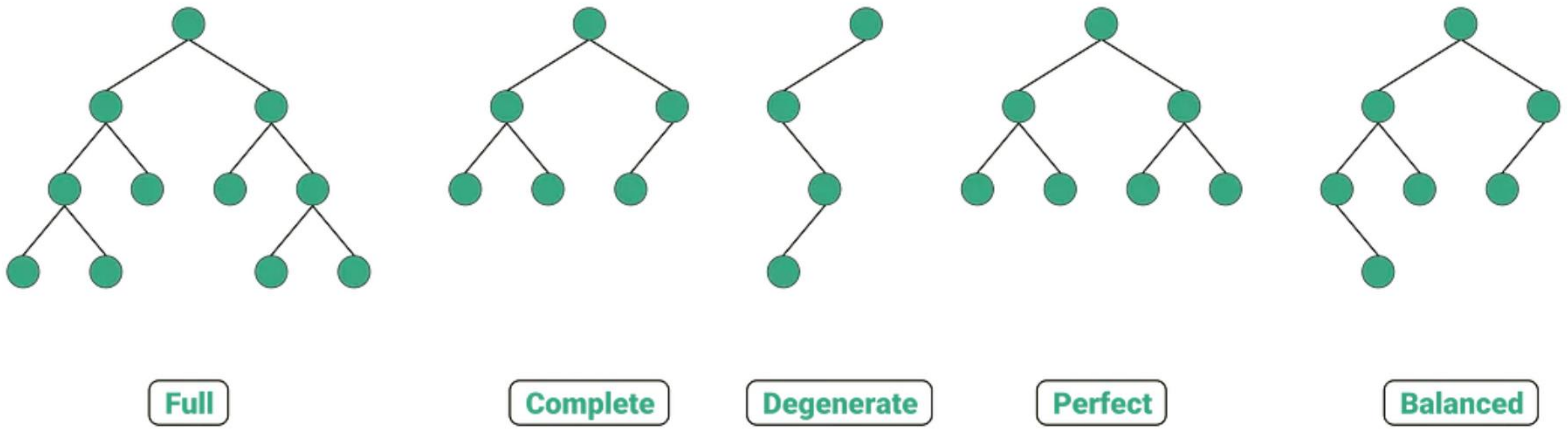
Binary Tree Representation

- A Binary tree is represented by a pointer to the topmost node (commonly known as the “root”) of the tree.
- If the tree is empty, then the value of the root is NULL. Each node of a Binary Tree contains the following parts:

1. key
2. Pointer to left child
3. Pointer to right child



Types of Binary Trees:



Full Binary Tree:

A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

Complete Binary Tree:

A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.

Perfect Binary Tree :

A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.

Degenerate or Pathological Tree:

A degenerate or pathological tree is the tree having a single child either left or right.

Basic Operation On Binary Tree:

- Inserting an element.
- Removing an element.
- Searching for an element.
- Traversing the tree

Tree Traversal:

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- Because, all nodes are connected via edges (links) we always start from the root (head) node.
- That is, we cannot randomly access a node in a tree. Tree Traversal algorithms can be classified broadly into two categories:
    1. Depth-First Search (DFS) Algorithms
    2. Breadth-First Search (BFS) Algorithms

Tree Traversal using Depth-First Search (DFS) algorithm can be further classified into three categories:

1. **Inorder Traversal (left-root-right):**

   Here, the traversal is left child – root – right child. It means that the left child is traversed first then its root node and finally the right child.

2. **Preorder Traversal (root-left-right):**

   Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.

3. **Postorder Traversal (left-right-root):**

   Here, the traversal is left child – right child – root. It means that the left child has traversed first then the right child and finally its root node.

Tree Traversal using Breadth-First Search (BFS) algorithm can be further classified into one category:

Level Order Traversal: Visit nodes level-by-level and left-to-right fashion at the same level.

Here, the traversal is level-wise.

It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

Node

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

Insertion

```
def insert(root,key):
    if root is None:
        root = Node(key)
        return root
    else:
        queue = [root]
        while queue:
            current_node = queue.pop(0)
            if current_node.left is None:
                current_node.left = Node(key)
                break
            else:
                queue.append(current_node.left)
            if current_node.right is None:
                current_node.right = Node(key)
                break
            else:
                queue.append(current_node.right)
```

Traversal

```
#Inorder traversal of a binary tree
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.key, end=" ")
    inorder(root.right)
def preorder(root):
    if root is None:
        return
    print(root.key, end=" ")
    preorder(root.left)
    preorder(root.right)
def postorder(root):
    if root is None:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.key, end=" ")
```

```
def delete(root, key):
    if root == None:
        return None
    if root.left == None and root.right == None:
        if root.key == key:
            return None
        else:
            return root
    key_node = None
    c_n = None
    q = []
    q.append(root)
    while q:
            c_n = q.pop(0)
            if c_n.key == key:
                key_node = c_n
            if c_n.left:
                q.append(c_n.left)
            if c_n.right:
                q.append(c_n.right)
```

```
RightMost_node=c_n #outside while loop
if key_node:
    ele = RightMost_node.key
    deleteDeepest(root, RightMost_node)
    key_node.key = ele
return root
```

Delete Deepest Node

```
def deleteDeepest(root, d_node):
    q = []
    q.append(root)
    while q:
        c_n = q.pop(0)
        if c_n is d_node:
            c_n = None
            return
        if c_n.left:
            if c_n.left is d_node:
                c_n.left = None
                return
            else:
                q.append(c_n.left)
        if c_n.right:
            if c_n.right is d_node:
                c_n.right = None
                return
            else:
                q.append(c_n.right)
```