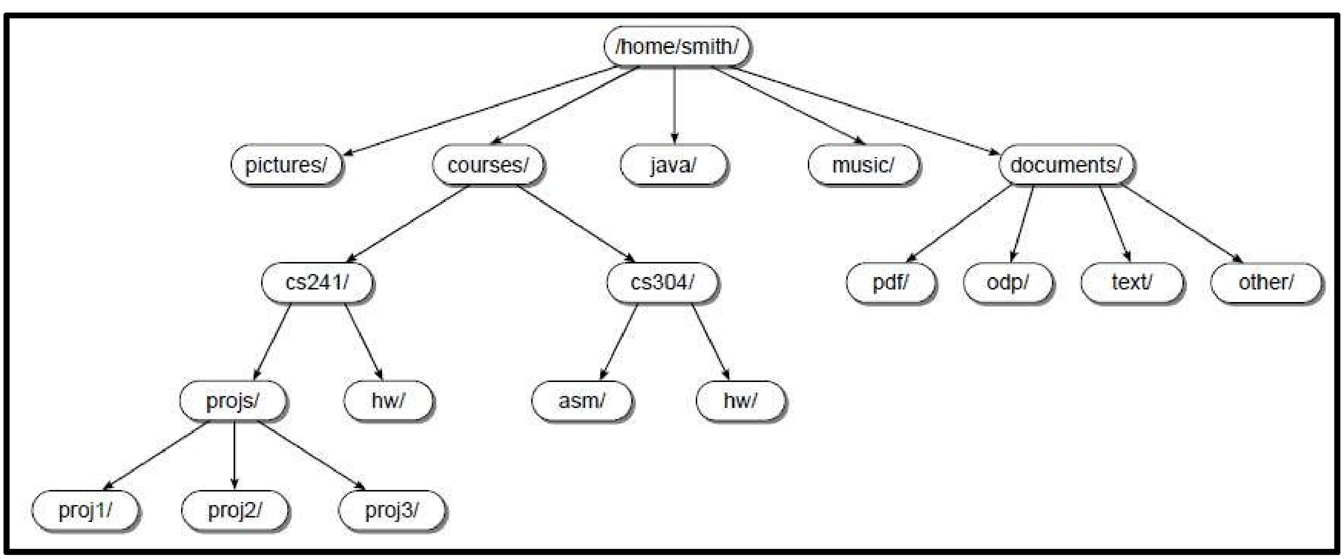
Week 11	The Tree data structure – Example: File explorer/Folder structure, Domain name server. Tree Terminologies, Tree node representation. Binary trees, Binary search trees, Properties, Implementation of tree operations – insertion, deletion, search, Tree traversals (in order, pre order and post order).

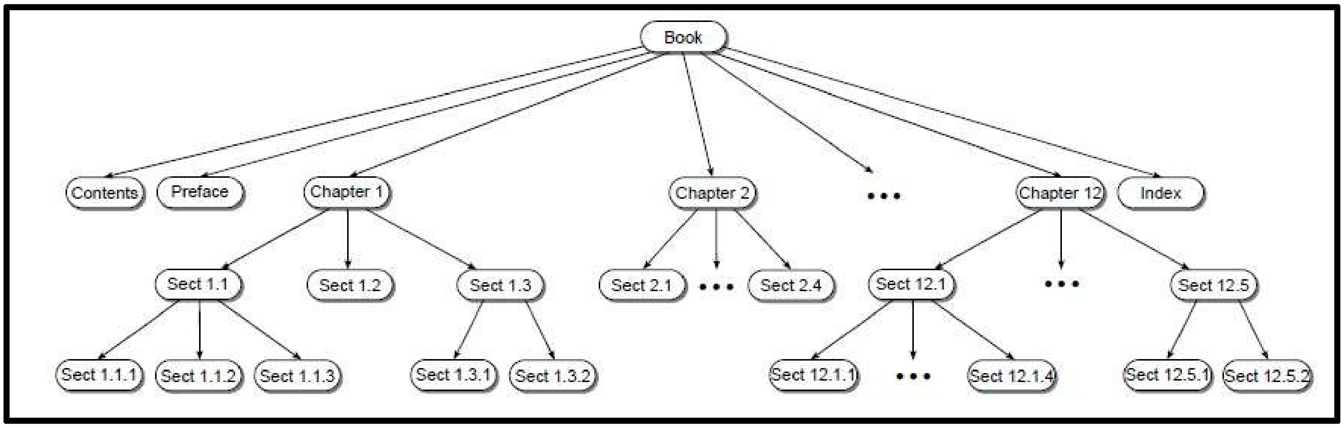
Tree Data Structure

- Tree Data Structure consists of nodes and edges that organize group of data in a hierarchical fashion.
- Tree data structures is specialized method to organize and store the data in the computer to be used effectively.
- Tree is a nonlinear data structure and hierarchical data structure which consist of collection of nodes such that each node of the tree stores a value and a list of reference to its children.
- The relationships between data elements in a tree are similar to a family tree: child, parent, ancestor etc.
- The data elements are stored in nodes and pairs of nodes are connected by edges.
- A classic example of a tree structure is:
  - ✓ File/Folder Explorer: The representation of files, directories and subdirectories in a file system,
  - ✓ Organization webpages in website,
  - ✓ Content/Index page of text book.

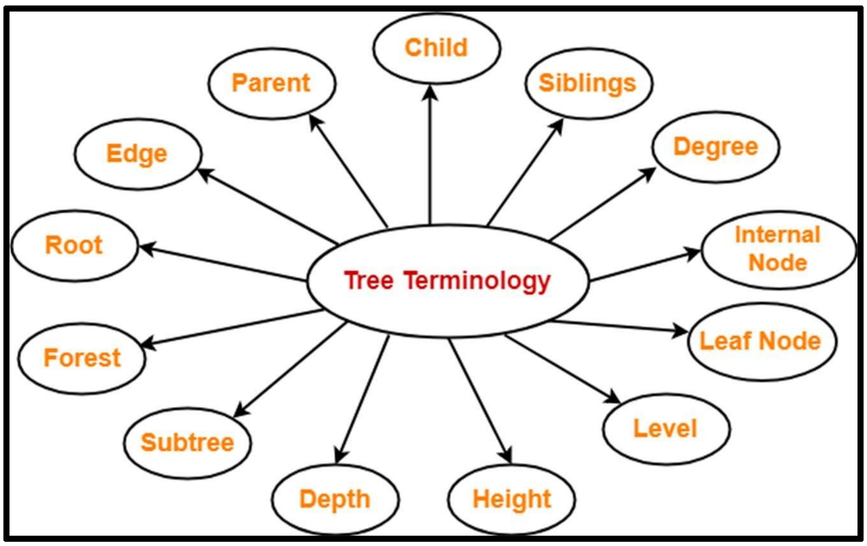
Example: File/Folder Explorer



Example: Book Index/Content Page Structure

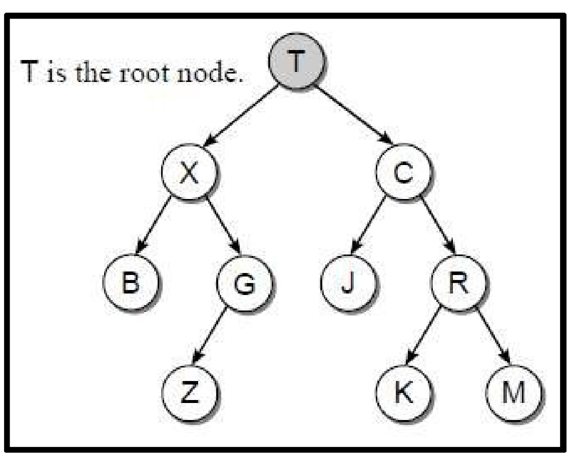


Tree Terminologies



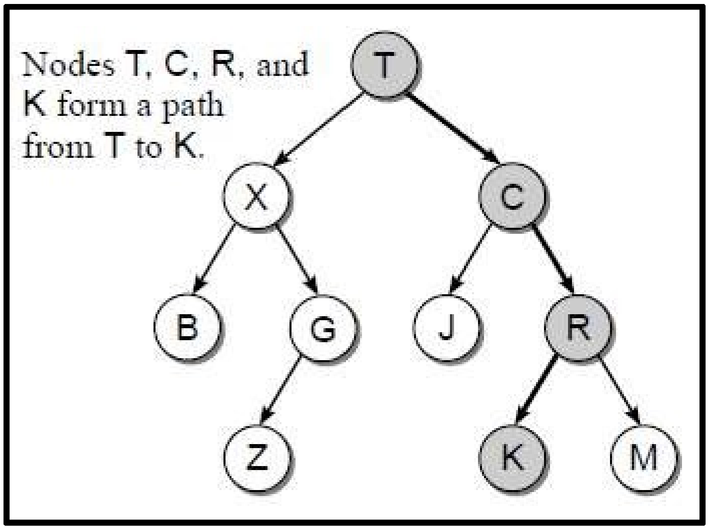
Root

- The topmost node of the tree is known as the root node.
- It provides the single access point into the tree structure.
- The root node is the only node in the tree that does not have an incoming edge.
- Consider the following sample tree, the node T is the root node here.



Path

- Is defined as sequence of nodes encountered and edges required to reach from a starting node to a destination form apath.
- The number of edges encountered in the path represents the length of path
- As shown in Figure, the nodes labeled T, C, R, and K form a path from node T to node K.



Parent

- A node that has atleast a single child is known as parent
- Every node, except the root, has a parent node, which is identified by the incoming edge.
- A node can have only one parent.

Children

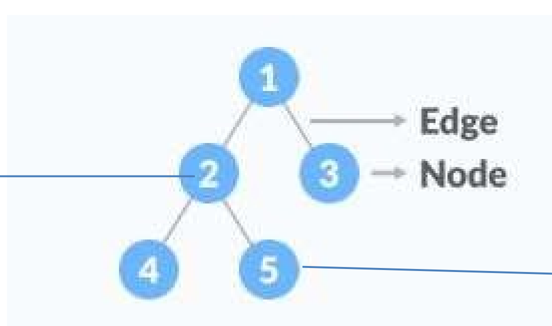
- A node which is reachable from any particular node X with a single edge becomes the child of X
- Each node can have one or more child nodes resulting in a parent-child hierarchy.

Nodes

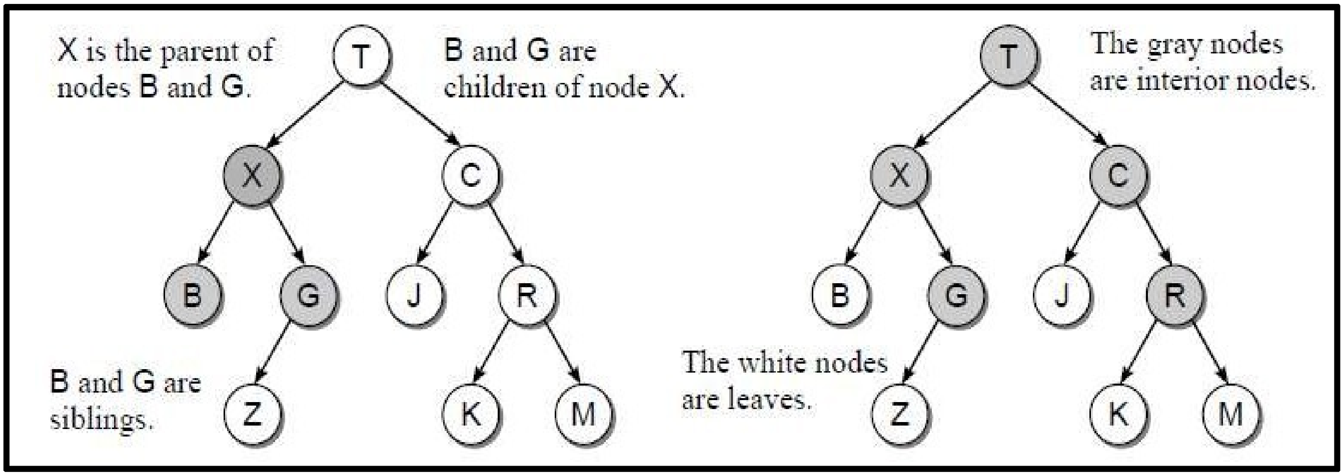
- A node is an entity that contains a key or value and reference to its child nodes.
- Nodes that have at least one child are known as **Interior nodes or Internal nodes**
- Nodes that have no children are known as **Leaf Nodes or External Nodes**

Edge: It is the link between any two nodes.

Internal Node

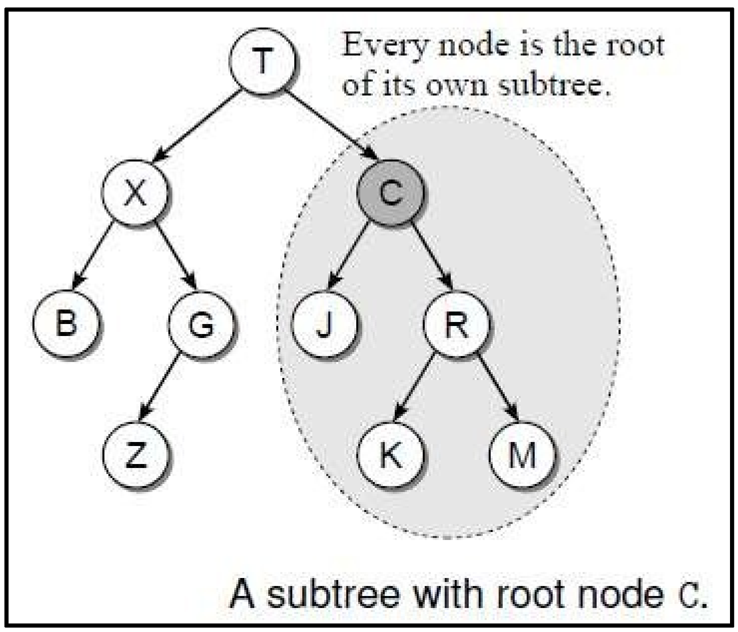


Leaf Node



Subtree

- Every node can be the root of its own subtree, which consists of a subset of nodes and edges of the larger tree.
- Figure below shows the subtree with node C as its root.



Relatives

- All of the nodes in a subtree are descendants of the subtree's root.
- In the example tree, nodes J, R, K, and M are descendants of node C.
- The ancestors of a node include the parent of the node, its grandparent, its great-grandparent, and so on all the way up to the root.
- In the example tree, nodes R, C, and T are ancestors of node M.
- The root node is the ancestor of every node in the tree and every node in the tree is a descendant of the root node.

Siblings: The nodes that have common parents are known as Siblings

Level of a node: The count of edges on the path from the root node to that node.

The root node has level 0.

Binary Tree

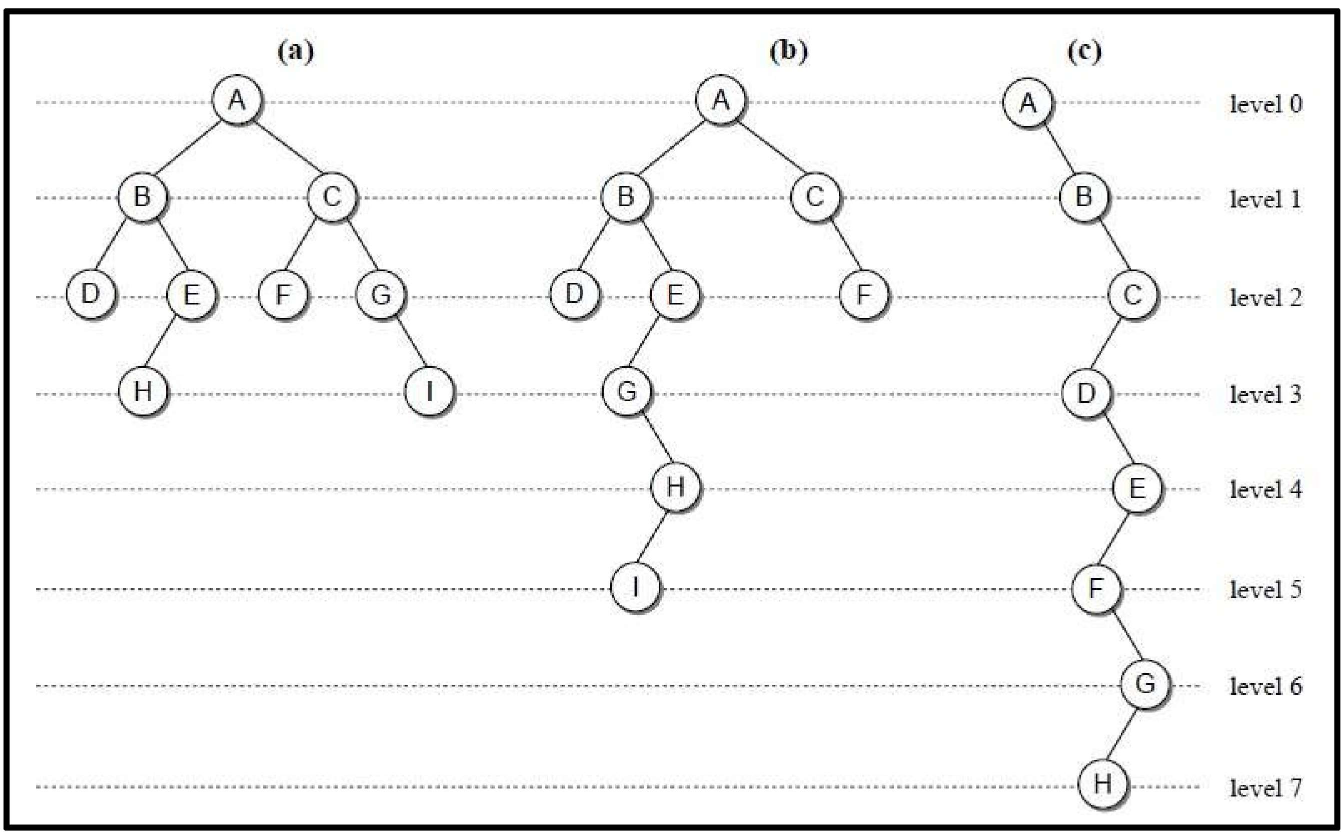
- One of the most commonly used trees in computer science is the binary tree.
- **A binary tree is a tree in which each node can have at most two children.**
- One child is identified as the left child and the other as the right child.

Properties

- Binary trees come in many different shapes and sizes.

Tree Size

- **The nodes in a binary tree are organized into levels with the root node at level 0, its children at level 1, the children of level one nodes are at level 2, and so on.**
- The binary tree in Figure (a), for example, contains two nodes at level one (B and C), four nodes at level two (D, E, F, and G), and two nodes at level three (H and I).
-
- The root node always occupies level zero.



Depth

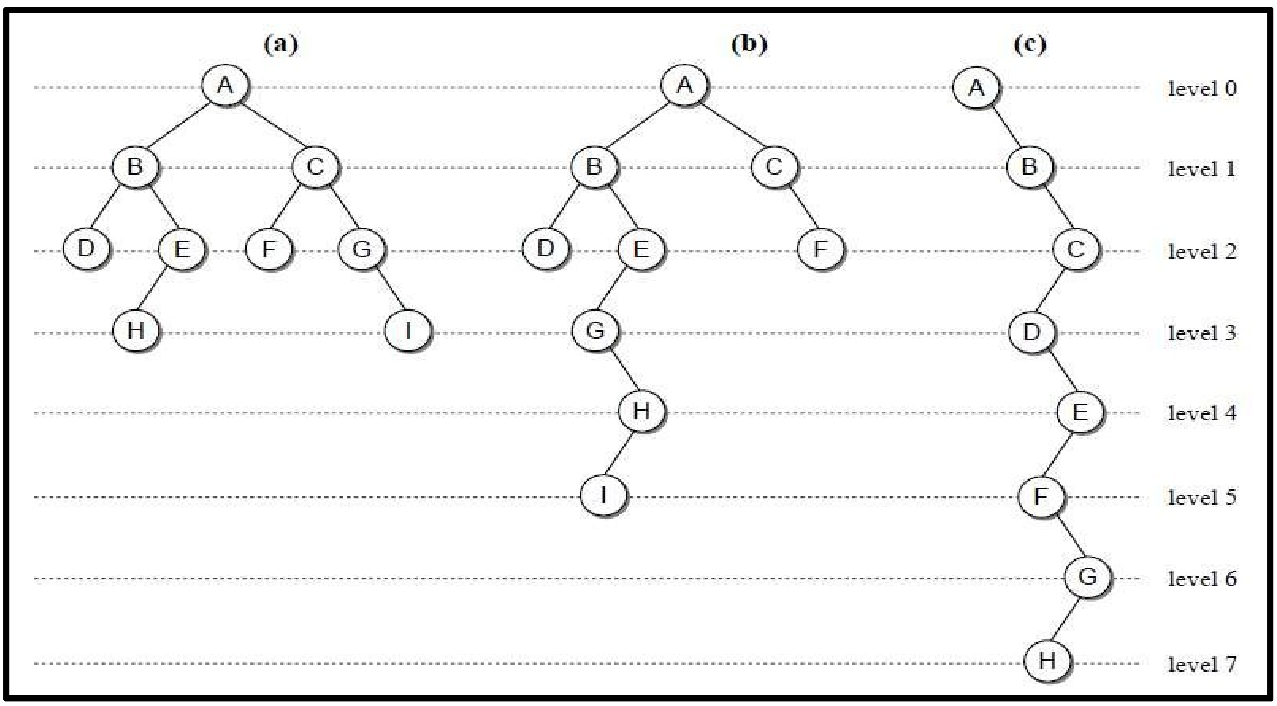
- **The depth of a node is the number of edges from root to that particular node**
- Consider node G in the three trees of Figure. In tree (a), G has a depth of 2, in tree (b) it has a

depth of 3, and in (c) its depth is 6.

Height:

- **The height of a node is the number of edges present in the longest path connecting that node to any leaf node**
- **Height of the Tree: The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree. Or It can also be defined as the height of a binary tree is defined as the height of its root node**
- For example, the three binary trees in Figure have different heights:

  (a) has a height of 3, (b) has a height of 5 (c) has a height of 7.
- **The width of a binary tree is the number of nodes on the level containing the most nodes.**
- In the three binary trees of Figure, (a) has a width of 4, (b) has a width of 3, and (c) has a width of 1.
- **Finally, the size of a binary tree is simply the number of nodes in the tree.**
- An empty tree has a height of 0 and a width of 0, and its size is 0.



Types of Binary Tree:

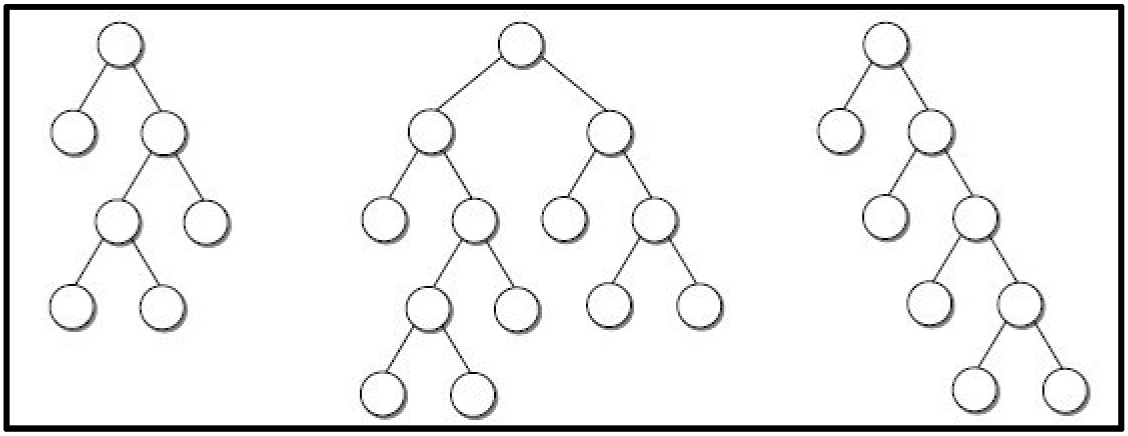
The following are the different types of Binary Tree

1) Full (Strictly/Proper) Binary Tree
2) Perfect/Complete Binary Tree
3) Almost Complete Binary Tree
4) Degenerated Binary Tree
5) Binary Search Tree

6) Balanced Tree

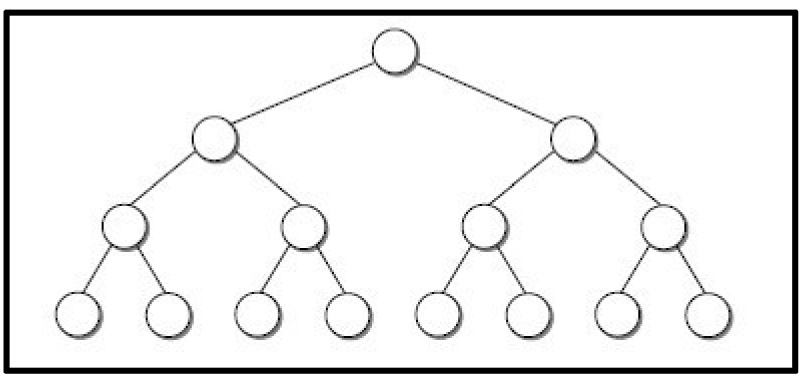
1) Full Binary Tree:

- **A full binary tree is a binary tree in which each interior node contains either no children or two children.**
- Full binary trees come in many different shapes, as illustrated in Figure.



2) Perfect Binary Tree

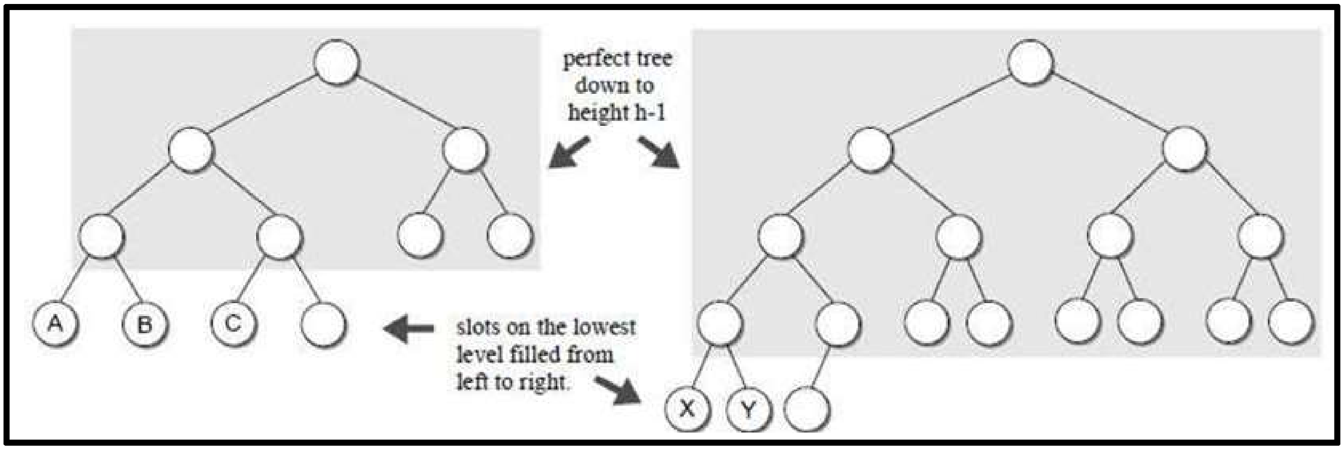
- **A perfect binary tree is a full binary tree in which all leaf nodes are at the same level.**
- The perfect tree has all possible node slots filled from top to bottom with no gaps, as illustrated in Figure.



3) Almost Complete Binary Tree

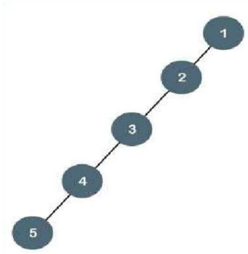
- An almost complete binary tree is a binary tree in which all the levels are completely filled except the last level, which is filled from the left.

▪ Consider the two almost complete binary trees in Figure.

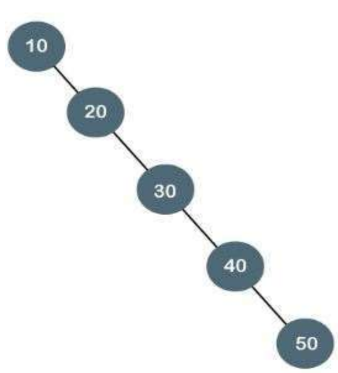


Degenerate Tree(Pathological tree):

- Degenerate binary tree is a binary tree in which every node has only a single child that can be either left or the right child.
- A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes.
- Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.
- In Left skewed binary tree every node has only left child as shown below



- In right skewed binary tree every node has only right child as shown below



Binary Tree Node Representation

- Nodes of a Binary trees are commonly implemented as an abstract data type similar to linked lists.
- Each node in a binary tree contains three fields:
  - ✓ data: contains value.
  - ✓ left: refers left node.
  - ✓ right: refers right node.

```
class _BTNode :
   def __init_( self, data ):
          self.data = data
          self.left = None
          self.right = None
```

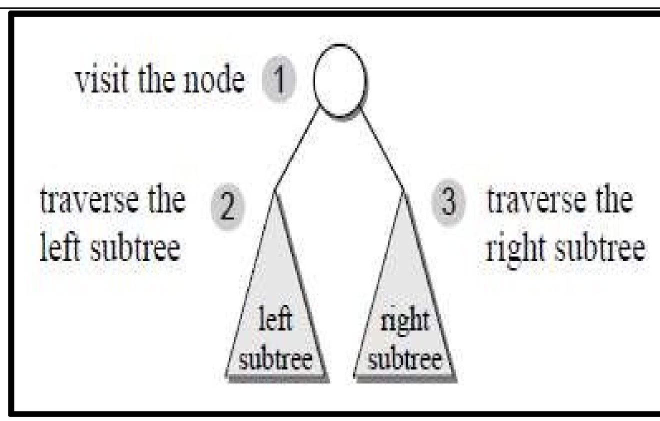
Tree Traversals

- Tree Traversal is defined as **"Visiting each and every node of a tree in some predefined order".**
- Three different types of Tree Traversals are:
  - ✓ **Pre-order Traversal**
  - ✓ **In-order Traversal**
  - ✓ **Post-order Traversal**

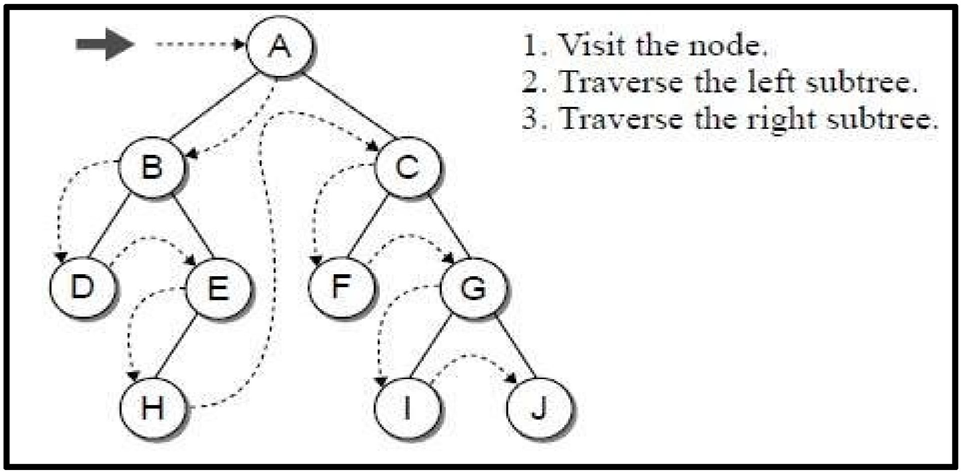
Pre-order Traversal

- Pre-order Traversal can be performed as follows:

✓ **First, visit the root node,**
✓ **Then, we traverse the left subtree,**
✓ **Finally, we traverse the right subtree**



- Consider the binary tree in Figure, **It's pre-order traversal: A, B, D, E, H, C, F, G, I, J.**

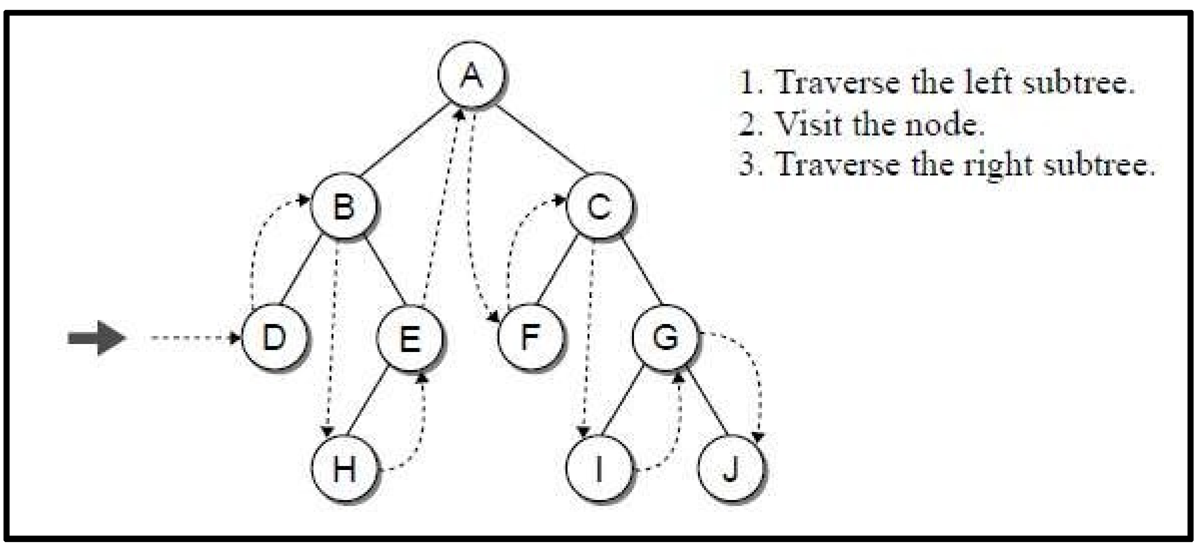


Python Function for Pre-order Traversal

```
def preorderTrav( subtree ):
    if subtree is not None :
        print( subtree.data )
        preorderTrav( subtree.left )
        preorderTrav( subtree.right )
```

In-order Traversal

- In-order Traversal can be performed as follows:
  - ✓ **First, we traverse the left subtree,**
  - ✓ **Then, visit the root node,**
  - ✓ **Finally, we traverse the right subtree.**
- Consider the binary tree in Figure: **It's in-order traversal: D, B, H, E, A, F, C, I, G, J.**



Python Function for In-order Traversal

```
def inorderTrav( subtree ):
    if subtree is not None :
        inorderTrav( subtree.left )
        print( subtree.data )
        inorderTrav( subtree.right )
```

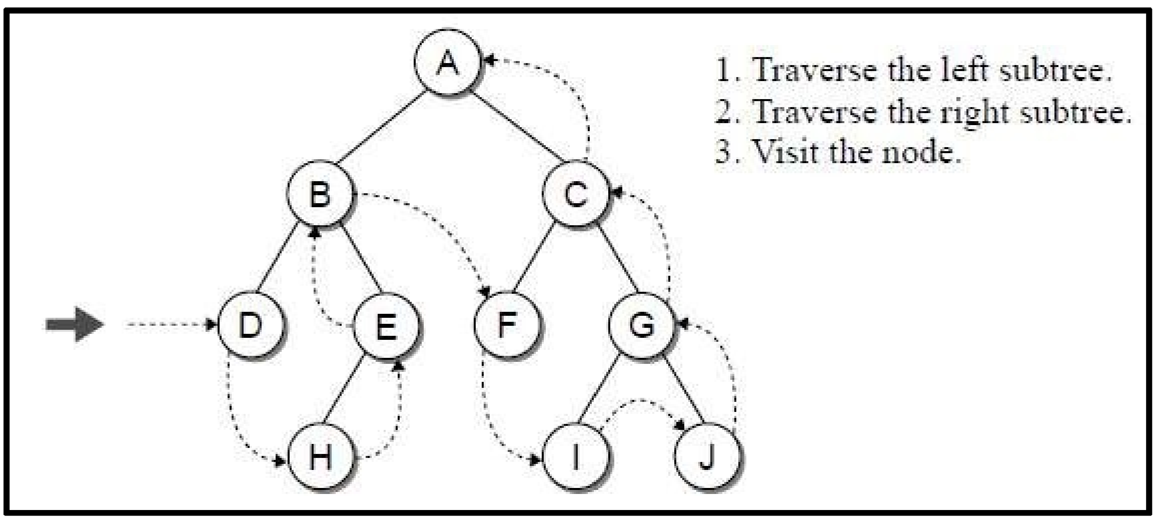
Post-order Traversal

- Post-order Traversal can be performed as follows:
  - ✓ **First, we traverse the left subtree,**
  - ✓ **Then, we traverse the right subtree.**
  - ✓ **Finally, visit the root node,**

Python Function for Post-order Traversal

```
def postorderTrav( subtree ):
    if subtree is not None :
        postorderTrav( subtree.left )
        postorderTrav( subtree.right )
        print( subtree.data )
```

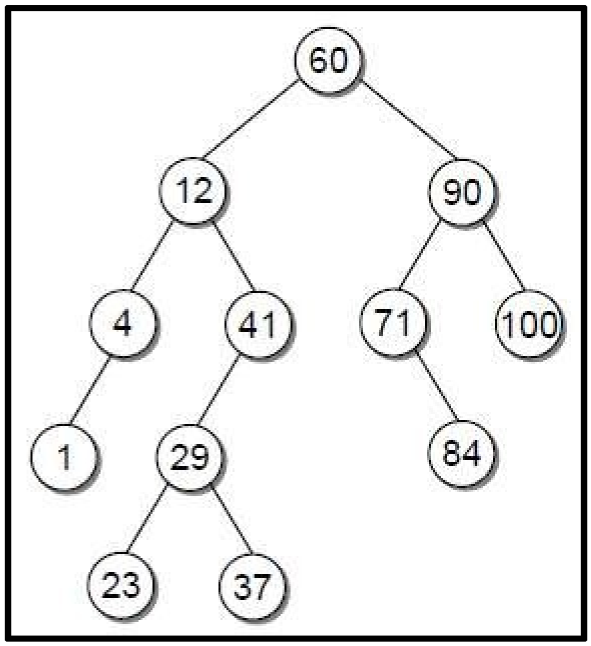
- Consider the binary tree in Figure, **It's post-order traversal: D, H, E,B, A, F, I,J, G,C,S.**



Binary Search Tree

- A binary search tree (BST) is a binary tree with the following properties:
  - ✓ The left subtree of a node must contain only the nodes with the keys lesser than the node's key
  - ✓ The right subtree of a node must contain only the nodes with the keys greater than or equal to the node's key
  - ✓ The left and right subtree each must also be a BST

- Consider the binary search tree in Figure, which contains integer values. The root node contains value 60 and all values in the root's left subtree are less than 60 and all of the values in the right subtree are greater than 60.



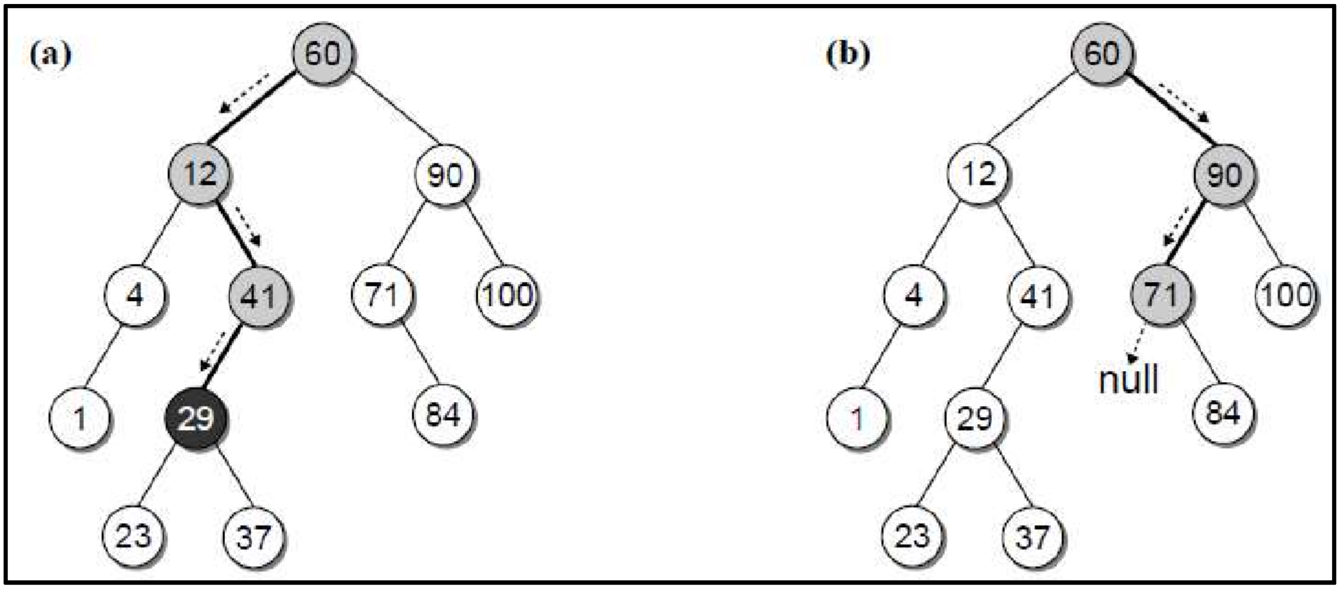
Operations of Binary Search Tree

- Searching
- Insertion
- Deletion

Searching

- Given a Binary Search Tree, we want to search the tree to determine if it contains a given target value or not.
- Our search must begin from root node.
- The target value is compared to the value in the root node as illustrated in Figure.
- If the root contains the target value, our search is over with a successful result.
- But if the target is not in the root, we must decide whether move to the left subtree or right subtree.
- If the target is less than the root's value, we move to left subtree.
- If the target is greater than or equal to the root's value, we move to right subtree.
- We repeat the comparison on the root node of the subtree and take the appropriate path.

- This process is repeated until target is located or we encounter a null child link.
- In Figure (a), we are searching target node 29, it results successful search.
- In Figure (b), we are searching target node 70, it results unsuccessful search.



Insertion

- When we create Binary Search Tree, nodes are inserted one at a time by creating new node.
- Suppose we want to build a binary search tree from the value list [60, 25, 100, 35, 17, 80] by inserting the keys in the order they are listed.

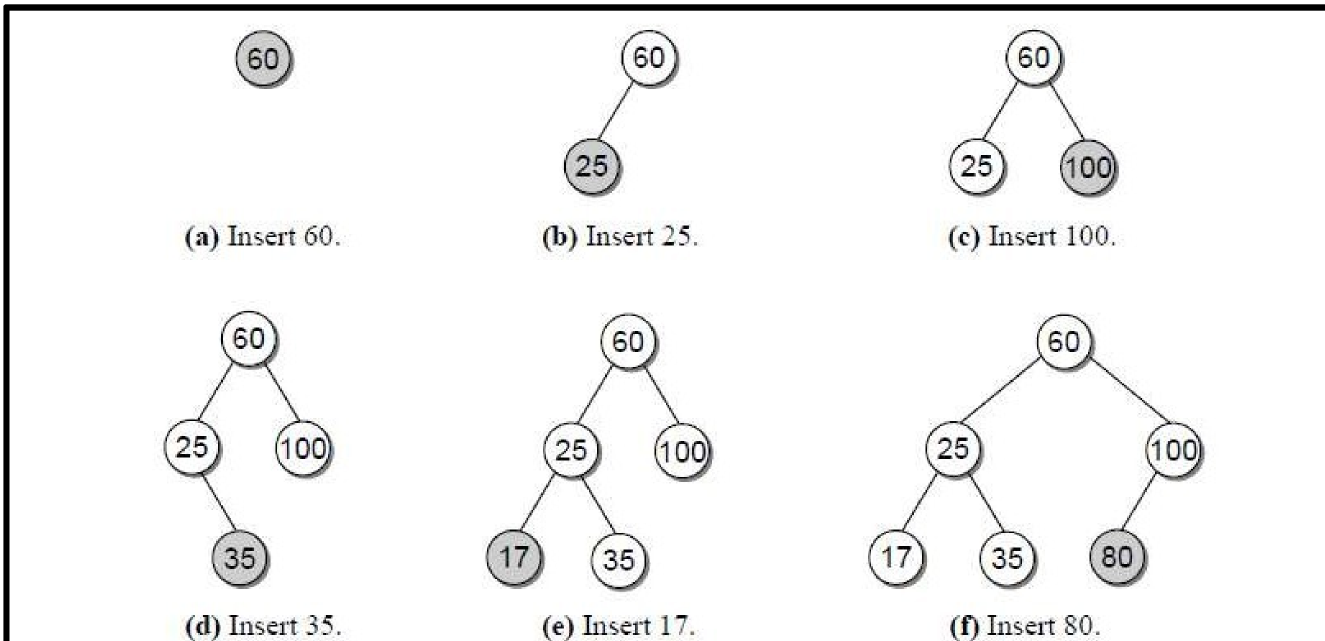


Figure 14.5: Building a binary tree by inserting the keys `[60, 25, 100, 35, 17, 80]`.

- ✓ We start by inserting value 60. A node is created and its data field set to that value.
- ✓ Since the tree is initially empty, this first node becomes the root of the tree (part a).

- ✓ Next, we insert value 25. Since it is smaller than 60, it has to be inserted to the left of the root, which means it becomes the left child of the root (part b).
- ✓ Value 100 is then inserted in a node linked as the right child of the root since it is larger than 60 (part c).
- ✓ Similarly, we insert 35, 17 and 80 (part d, e f).

Deletion

- Removing an element from a binary search tree is a bit more complicated than searching for an element or inserting a new element into the tree.
- A deletion involves searching for the node that contains the target value and then unlinking the node to remove it from the tree.
- When a node is removed, the remaining nodes must preserve the search tree property.
- There are three cases to consider once the node has been found:
    1. The node is a leaf.
    2. The node has a single child.
    3. The node has two children.

Removing a leaf node

- Removing a leaf node is the easiest among the three cases.
- Suppose we want to delete value 23 from the binary search tree in below Figure.
- After finding the node, it has to be unlinked, which can be done by setting the left child field of its parent, node 29, to None, as shown in Figure (a).

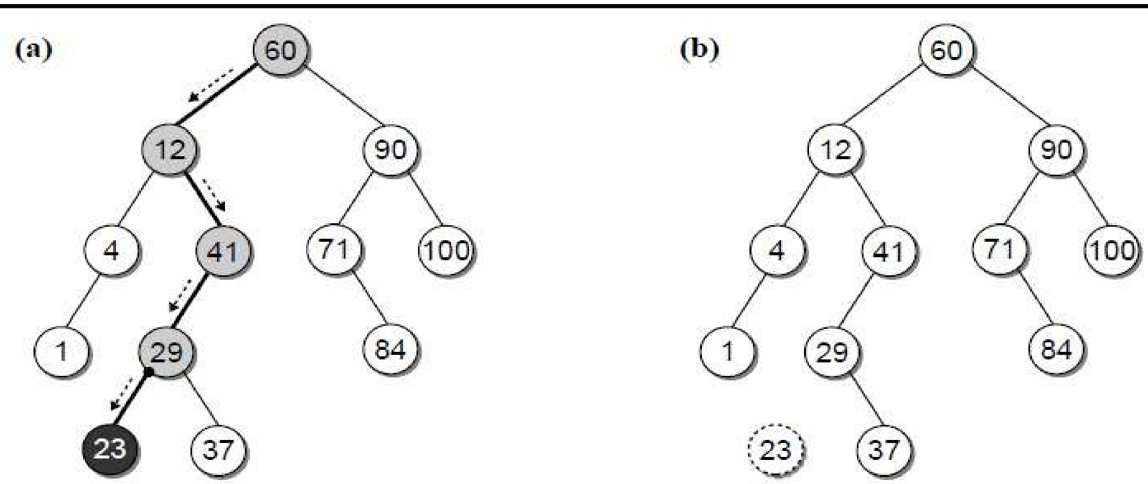


Figure 14.8: Removing a leaf node from a binary search tree: (a) finding the node and unlinking it from its parent; and (b) the tree after removing 23.

Removing an Interior Node with One Child

- If the node to be removed has a single child, it can be either the left or right child.
- Suppose we want to delete key value 41 from the binary search tree in Figure 14.1.
- Node 41 is the right child of node 12, all of the descendants of node 41 must also be larger than 12.
- Thus, we can set the link in the right child field of node 12 to reference node 29, as illustrated in Figure.
- Node 29 now becomes the right child of node 12 and all of the descendants of node 41 will be properly linked without losing any nodes.

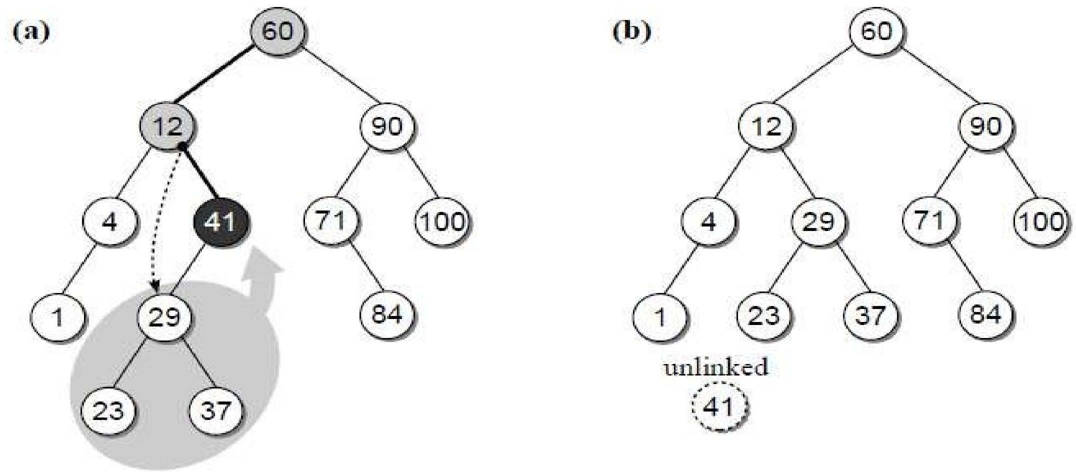


Figure 14.10: Removing an interior node (41) with one child: (a) redirecting the link from the node's parent to its child subtree; and (b) the tree after removing 41.

Removing an Interior Node with Two Children

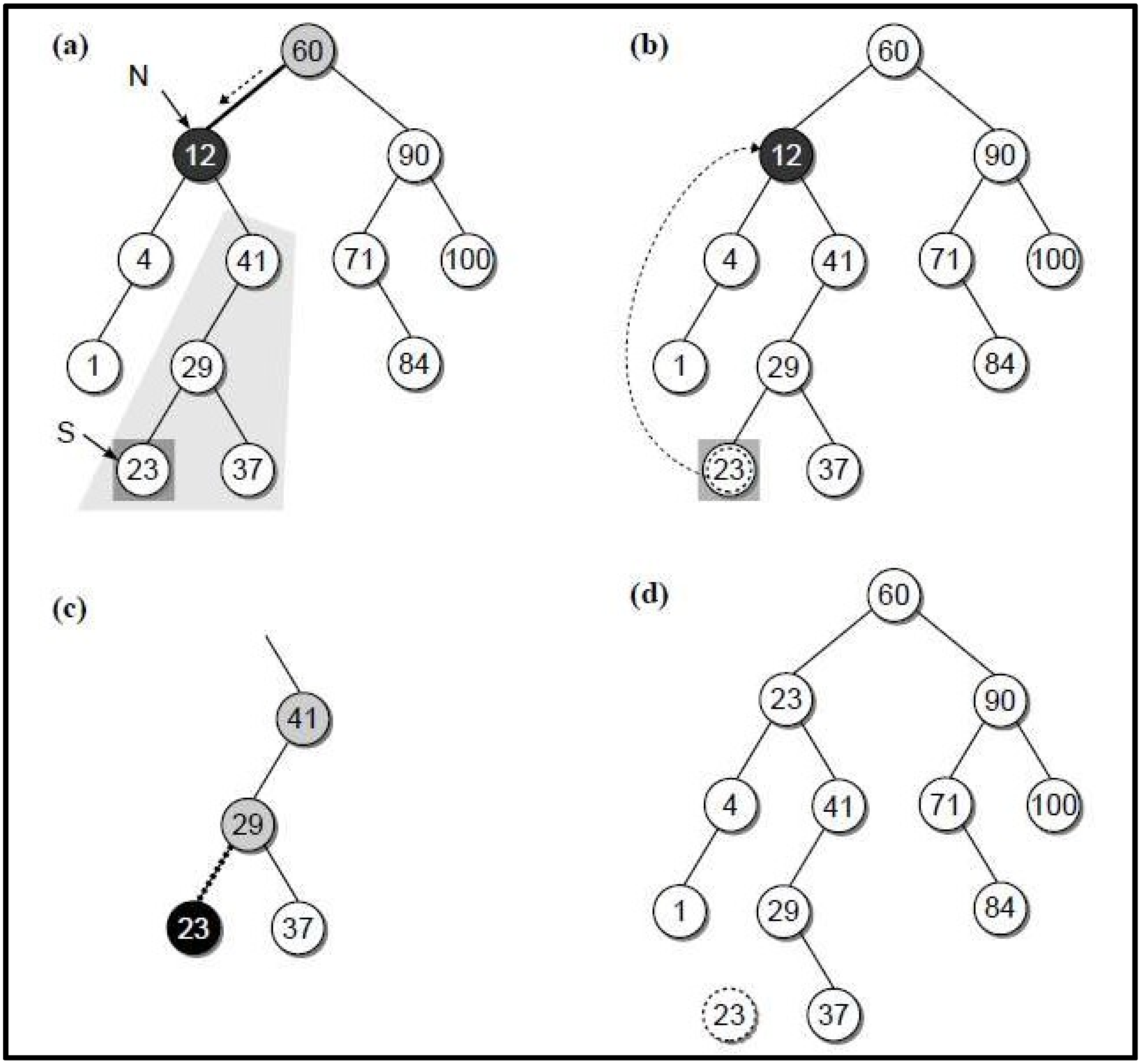
- The most difficult case is when the node to be deleted has two children.
- For example, suppose we want to remove node 12 from the binary search tree in Figure.
- The steps in removing a key from a binary search tree:

  (a) find the node, N, and its in order successor, S;// Inorder successor of a node N is a node with smallest key but the key must be greater than key of N

  (b) copy the successor key from node S to N;

  (c) remove the successor key from the right subtree of N; and

  (d) the tree after removing 12.


(a)
N
S
60
12
90
4
41
71
100
1
29
84
23
37
(b)
60
12
90
4
41
71
100
1
29
84
23
37
(c)
41
29
23
37
(d)
60
23
90
4
41
71
100
1
29
84
23
37

Program #1: Python Program to implement

```
class Node:
    def__init_(self,value):
        self.data = value
        self.left = None
        self.right =None
class BST:
    def__init_(self):
        self.root=None

    def search(self,key):
        curNode = self.root
        while curNode is not None:
            if key == curNode.data:
                return True
            elif key < curNode.data:
                    curNode = curNode.left
            else:
                    curNode=curNode.right
        return False
    def delete(self,key):
        curNode = self.root
        parentNode =None
        while curNode is not None:
            if key == curNode.data:
                if temp=="Left":
                    parentNode.left=None
                else:
                    parentNode.right=None
                print(key,"Node  Deleted")
                return True
            elif key < curNode.data:
                    parentNode = curNode
                    curNode = curNode.left
                    temp="Left"
            else:
                    parentNode = curNode
                    curNode=curNode.right
                    temp="Right"
        print(key,"Node not found")
        return False
```

```python
def insert(self,value):
    newNode=Node(value)
    if self.root is None:
        self.root = newNode
    else:
     curNode = self.root
     while curNode is not None:
        if value < curNode.data:
            if curNode.left is None:
                curNode.left=newNode
                break
            else:
                curNode = curNode.left
        else:
            if curNode.right is None:
                curNode.right=newNode
                break
            else:
                curNode=curNode.right



def preorder(self, rt):
    print(rt.data, end="\t")
    if rt.left is not None:
        self.preorder(rt.left)
    if rt.right is not None:
        self.preorder(rt.right)

def inorder(self, rt):
    if rt.left is not None:
        self.inorder(rt.left)
    print(rt.data, end="\t")
    if rt.right is not None:
        self.inorder(rt.right)
```

```python
    def postorder(self, rt):
        if rt.left is not None:
            self.postorder(rt.left)
        if rt.right is not None:
            self.postorder(rt.right)
        print(rt.data, end="\t")

BT = BST()

ls = [25,10,35,20,65,45,24]
for i in ls:
    BT.insert(i)

print("\nPre-order")
BT.preorder(BT.root)
print("\nIn-order")
BT.inorder(BT.root)
print("\nPost-order")
BT.postorder(BT.root)
print("\n35 exists:", BT.search(35))
print("65 exists:", BT.search(65))
BT.delete(75)
BT.delete(24)
print("In-order")
BT.inorder(BT.root)
```