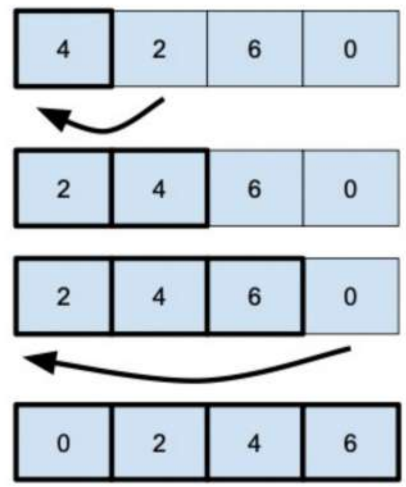
Lecture 8
Binary Tree

Dr. Sara S. Elhishi
Information Systems Dept.
Mansoura University, Egypt

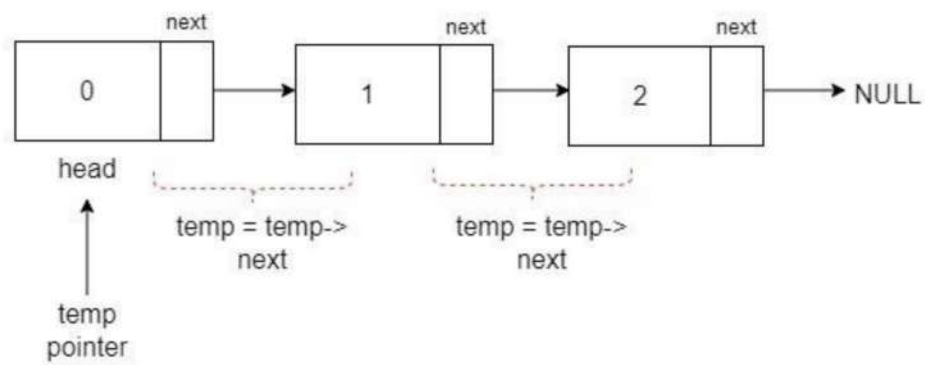
Array

- Searching could be fast in an ordered array (i.e. O(log N).
- Insertion and Deletion require Shifting; N/2 on average moves.



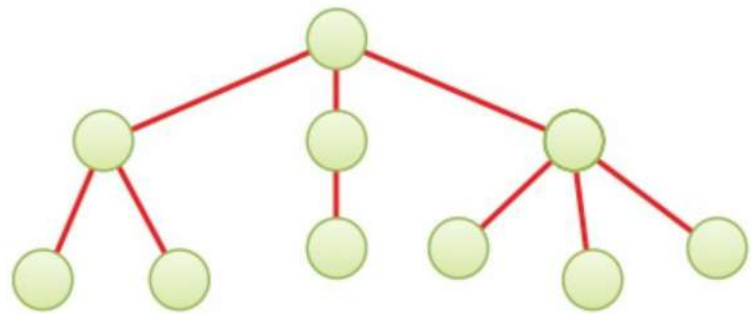
Linked List

- Straightforward Insertion and Deletion.
- However, Searching is time-consuming in the worst case O(N).



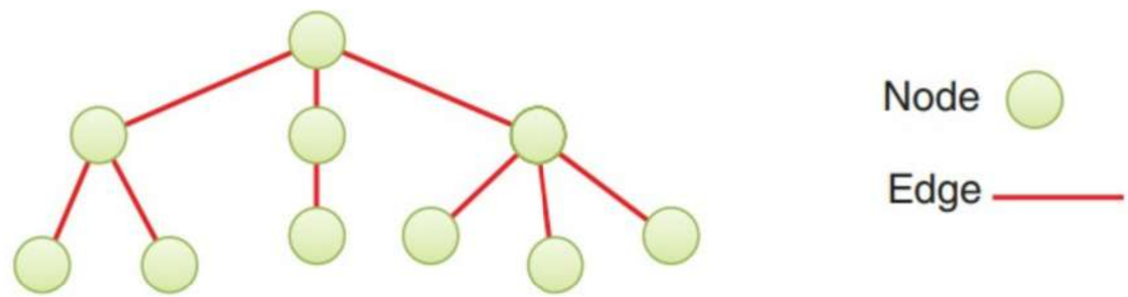
Tree To The Rescue

- A data structure that enables rapid insertion and deletion of linked lists, as well as efficient searching of ordered arrays.
- Trees possess each of these attributes and are, moreover, one of the most captivating data structures.



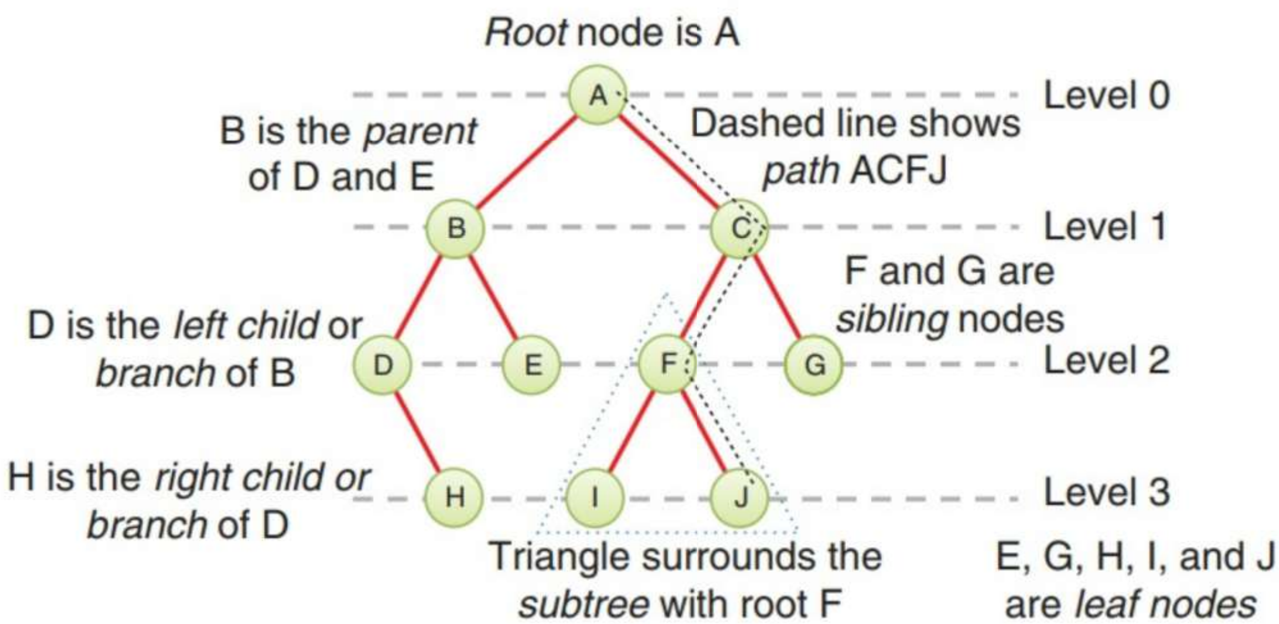
What is a Tree?

- A tree consists of **nodes** connected by **edges**.
- Trees are much like Graphs, but with different edge Configuration.
- Lines/edges symbolize convenience.
- There exist various types of trees, categorized based on the quantity of their edges.
- Here, we are dealing with Binary Trees.



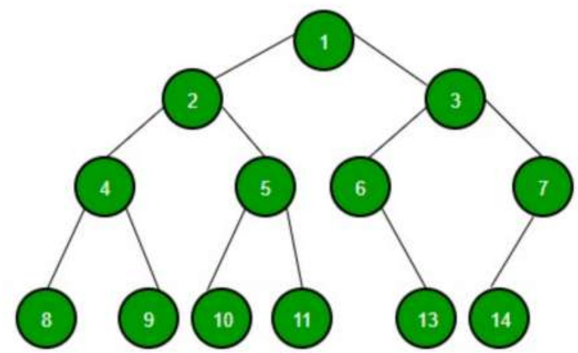
Tree Terminology

- Root
- Path
- Parent
- Child
- Sibling
- Leaf
- Subtree
- Visiting
- Traversing
- Levels/depth
- Keys



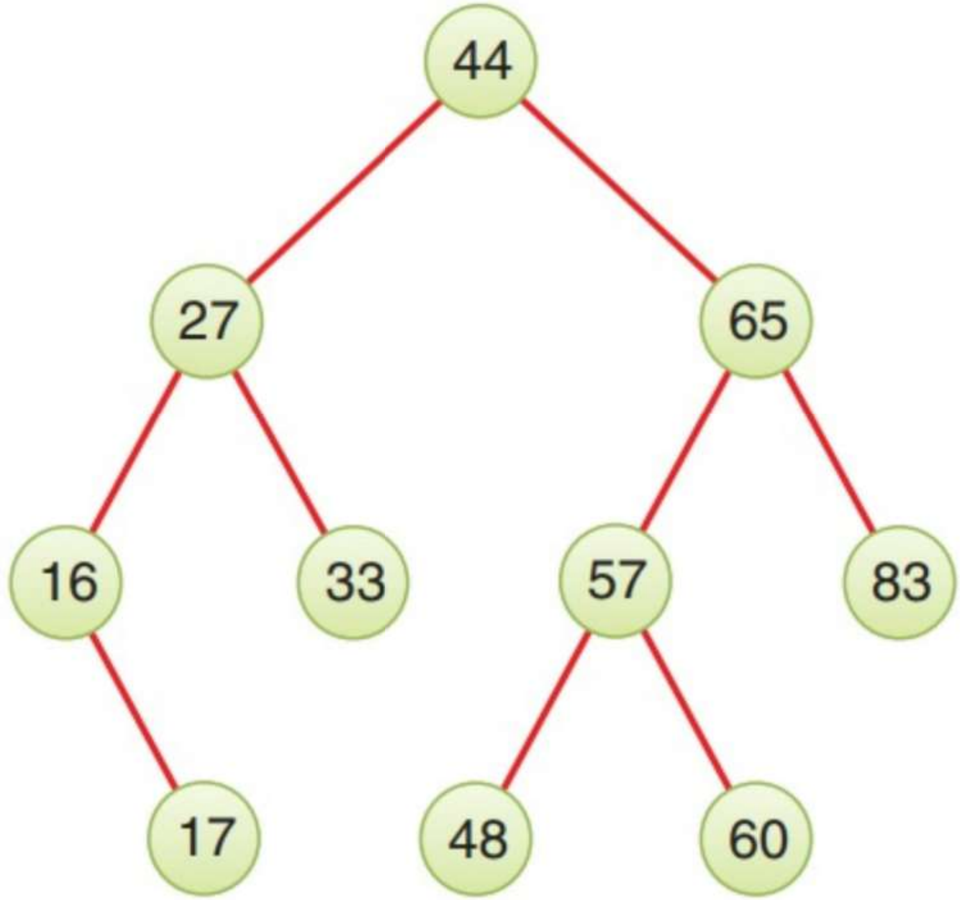
Binary Tree

- A Tree with a maximum of 2 children for each node.
- Each node may have both left and right offspring, a left child, a right child only, or no children at all.



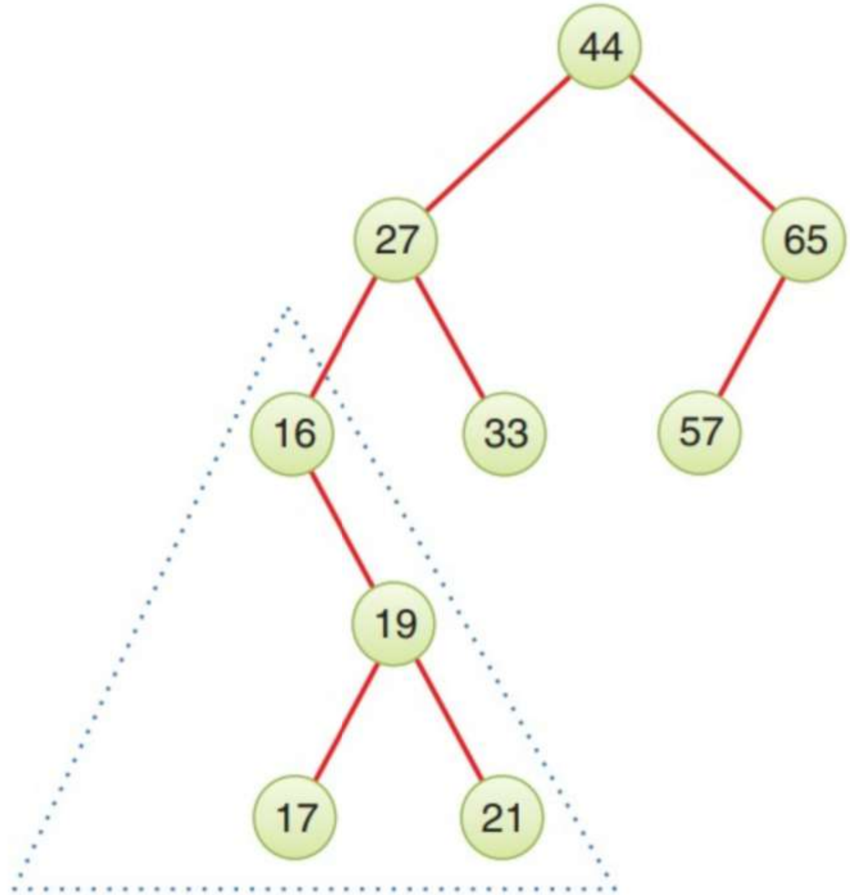
Binary Search Trees

Left < Parent <= Right



Unbalanced Trees

Are characterized by the concentration of their nodes on either side of the root



Tree Implementation

- Store the nodes in separate memory regions and establish connections between them by using references
- Representing a tree in memory by storing nodes in certain positions as corresponding units in an array.
- We will implement the linkage option.

BinarSearchTree.py

- An empty tree, the constructor sets the reference to the root node to None.

```python
class BinarySearchTree(object):
    def __init__(self):
        self.__root = None  # Initialy its empty
```

The Node Class

```python
class BinarySearchTree(object):

    class __Node(object):
        def __init__(self, key, data, left = None, right = None):
            self.key = key
            self.data = data
            self.leftChild = left
            self.rightChild = right

        # represent a node as a string
        def __str__(self):
            return "{" + str(self.key) + ", " + str(self.data) + "}"

    def __init__(self):
        self.__root = None  # Initialy its empty

    def isEmpty(self):
        return self.__root is None

    def root(self):
        if self.isEmpty():
            raise Exception(['No root node in empty tree'])
        return (self.__root.data, self.__root.key)
```

Finding A Node

```python
# Find an internal node whose key matches goal
def __find(self, goal):
    current = self.__root # start at root
    parent = self         # parent of current node
    # loop while there is a tree left to explore and
    # current key isn't the goal
    while (current and goal != current.key):
        parent = current    # one level down
        current = ( # advance goal to left subtree or rith based on goal
            current.leftChild if goal < current.key else current.righChild
        )
    # if the loop ends with a node return it or None with parent
    return(current, parent)

# get data assciated with a goal key
def search(self, goal):
    node, p= self.__find(goal)
    return node.data if node else None
```

Tree Efficiency

- If the tree is balanced, the time complexity is O(log N) time
- If the tree is entirely imbalanced, and the time complexity is O(N)

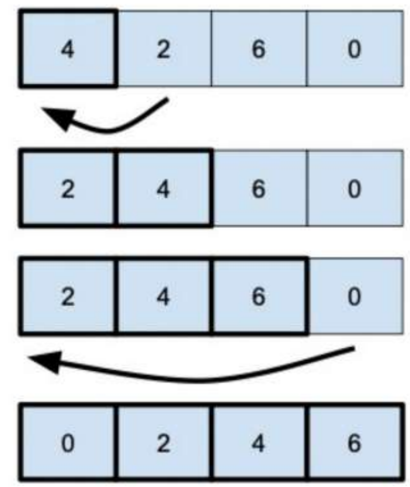
To Be Continued …

Lecture 9
Binary Tree

Dr. Sara S. Elhishi
Information Systems Dept.
Mansoura University, Egypt

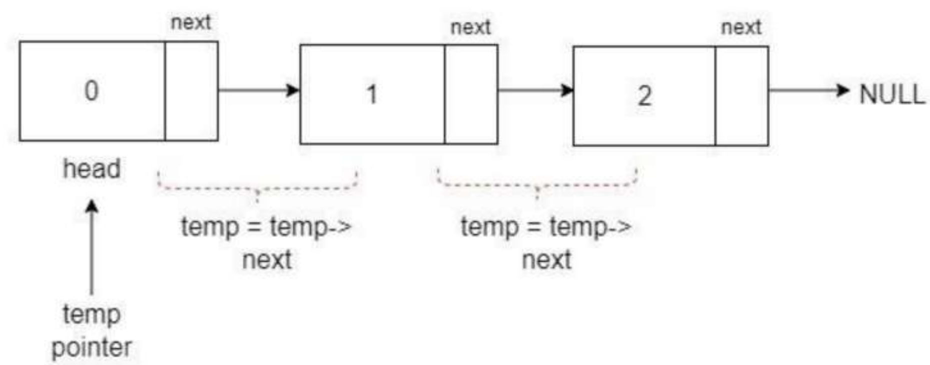
Array

- Searching could be fast in an ordered array (i.e. O(log N).
- Insertion and Deletion require Shifting; N/2 on average moves.



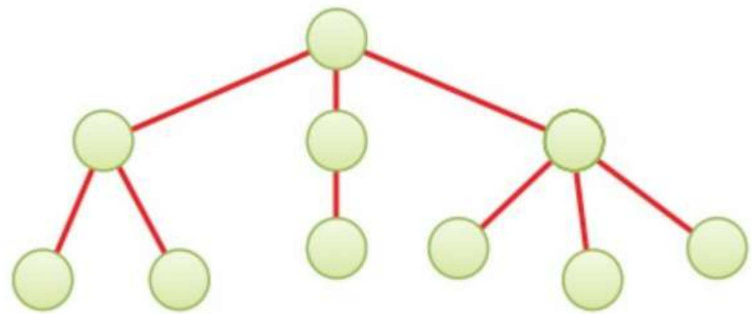
Linked List

- Straightforward Insertion and Deletion.
- However, Searching is time-consuming in the worst case O(N).



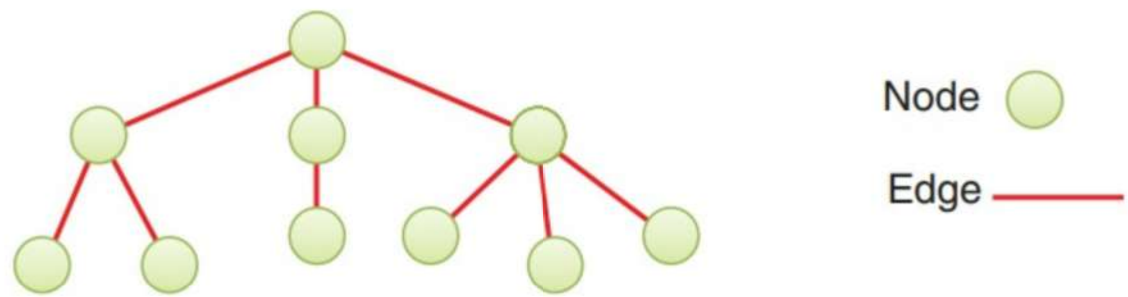
Tree To The Rescue

- A data structure that enables rapid insertion and deletion of linked lists, as well as efficient searching of ordered arrays.
- Trees possess each of these attributes and are, moreover, one of the most captivating data structures.



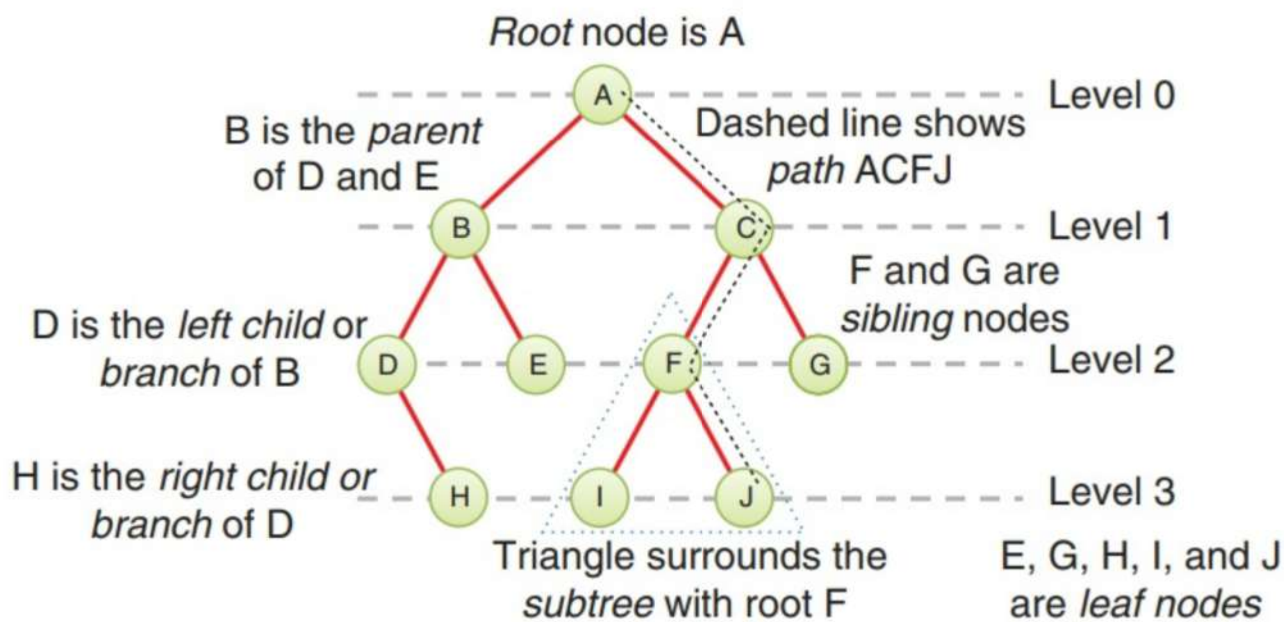
What is a Tree?

- A tree consists of **nodes** connected by **edges**.
- Trees are much like Graphs, but with different edge Configuration.
- Lines/edges symbolize convenience.
- There exist various types of trees, categorized based on the quantity of their edges.
- Here, we are dealing with Binary Trees.



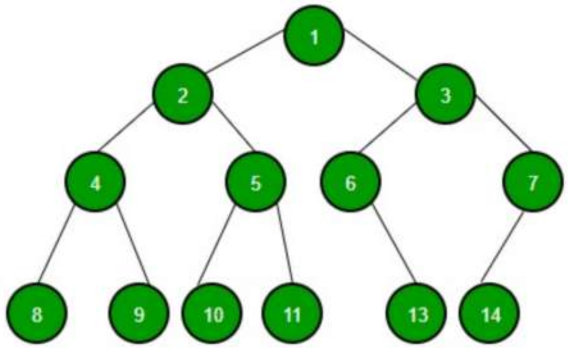
Tree Terminology

- Root
- Path
- Parent
- Child
- Sibling
- Leaf
- Subtree
- Visiting
- Traversing
- Levels/depth
- Keys



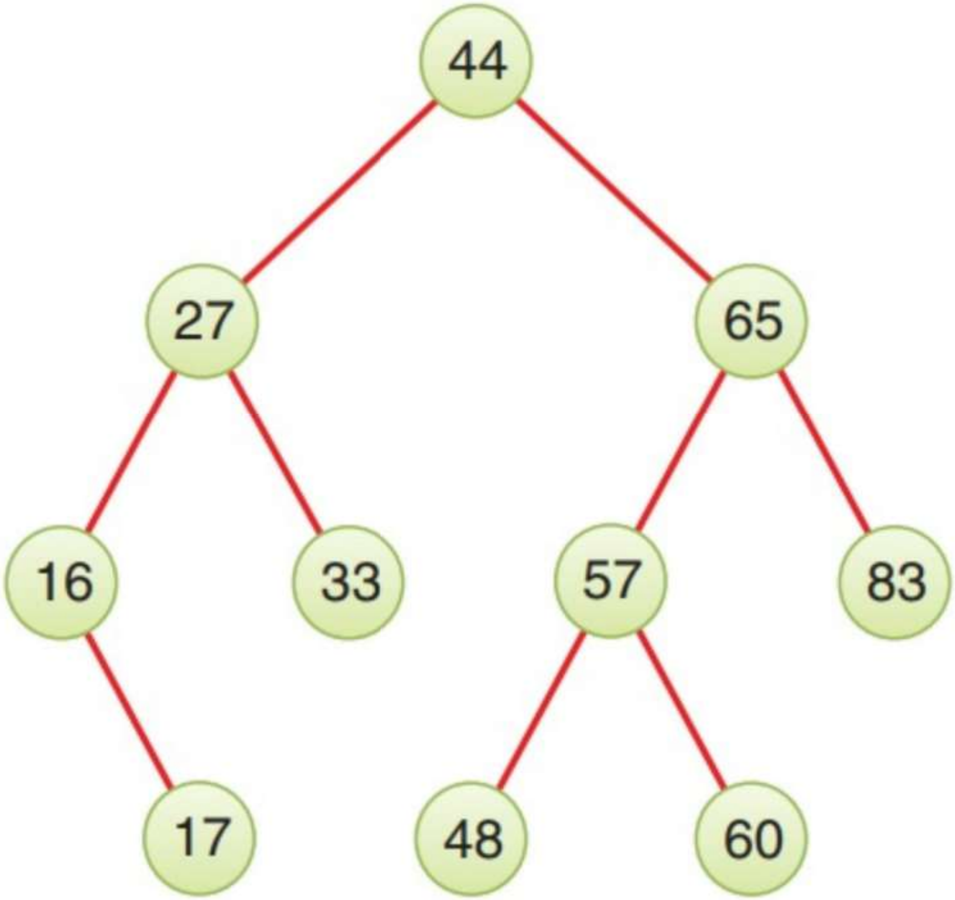
Binary Tree

- A Tree with a maximum of 2 children for each node.
- Each node may have both left and right offspring, a left child, a right child only, or no children at all.



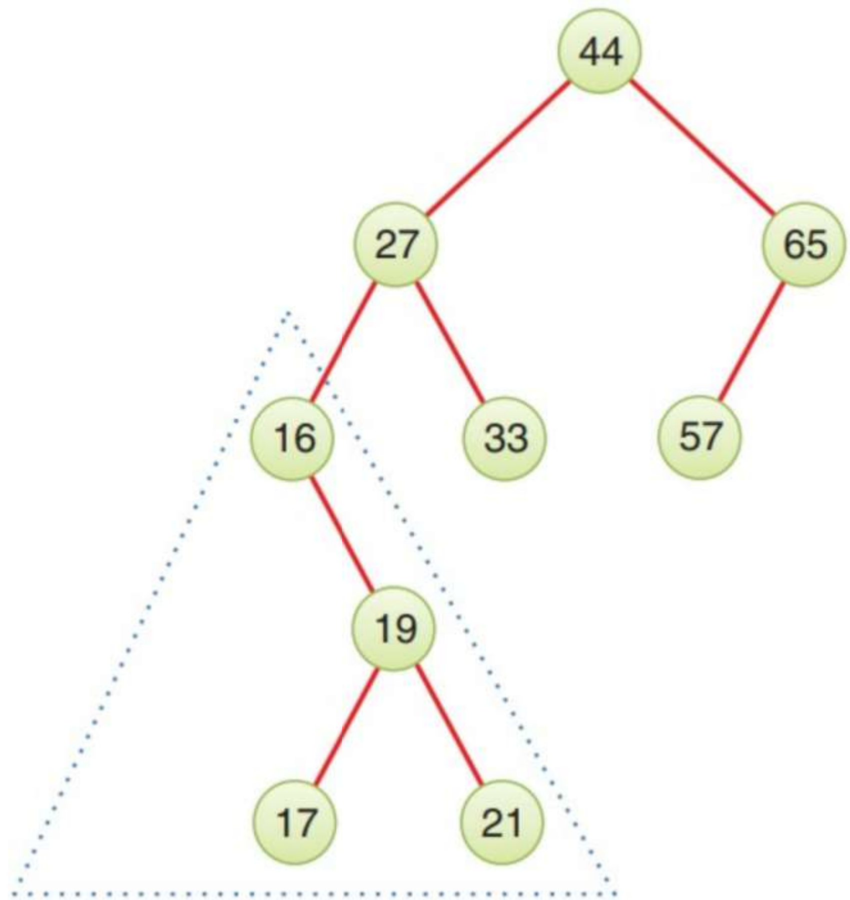
Binary Search Trees

Left < Parent <= Right



Unbalanced Trees

Are characterized by the concentration of their nodes on either side of the root



Tree Implementation

- Store the nodes in separate memory regions and establish connections between them by using references
- Representing a tree in memory by storing nodes in certain positions as corresponding units in an array.
- We will implement the linkage option.

BinarSearchTree.py

- An empty tree, the constructor sets the reference to the root node to None.

```python
class BinarySearchTree(object):
    def __init__(self):
        self.__root = None  # Initialy its empty

```

The Node Class

```python
class BinarySearchTree(object):

    class __Node(object):
        def __init__(self, key, data, left = None, right = None):
            self.key = key
            self.data = data
            self.leftChild = left
            self.rightChild = right

        # represent a node as a string
        def __str__(self):
            return "{" + str(self.key) + ", " + str(self.data) + "}"

    def __init__(self):
        self.__root = None  # Initialy its empty

    def isEmpty(self):
        return self.__root is None

    def root(self):
        if self.isEmpty():
            raise Exception(['No root node in empty tree'])
        return (self.__root.data, self.__root.key)
```

Finding A Node

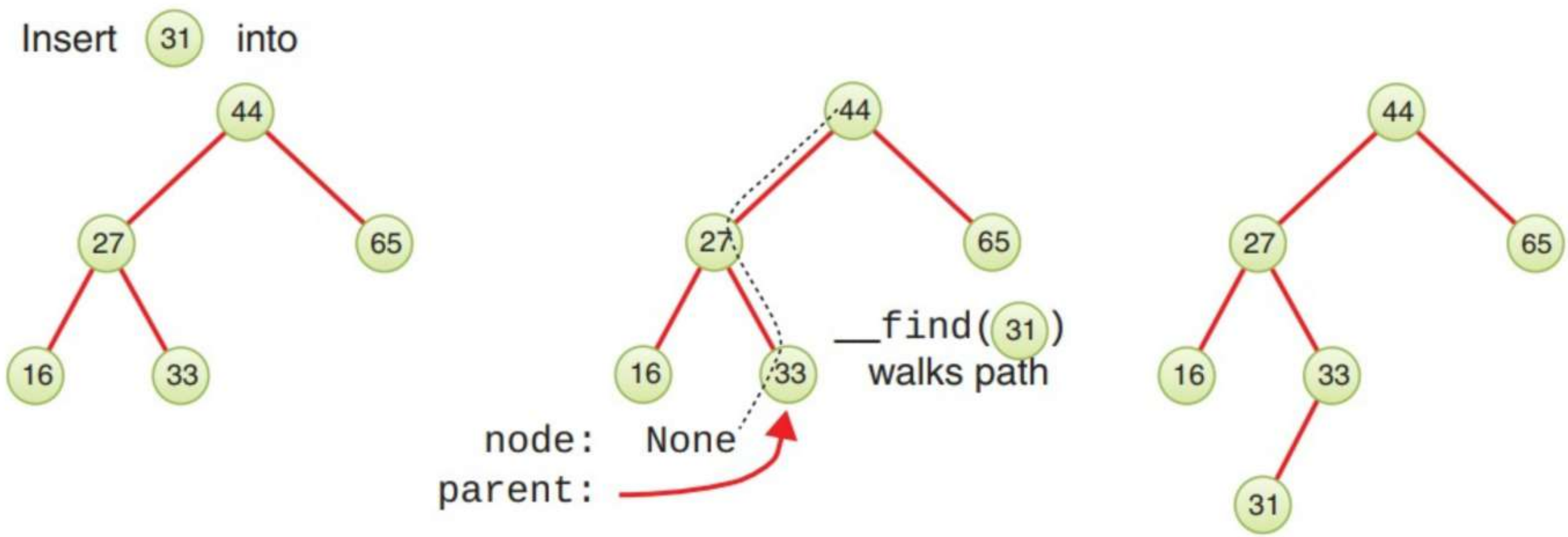
```python
    # Find an internal node whose key matches goal
    def __find(self, goal):
        current = self.__root # start at root
        parent = self         # parent of current node
        # loop while there is a tree left to explore and
        # current key isn't the goal
        while (current and goal != current.key):
            parent = current    # one level down
            current = ( # advance goal to left subtree or rith based on goal
                current.leftChild if goal < current.key else current.righChild
            )
        # if the loop ends with a node return it or None with parent
        return(current, parent)

    # get data assciated with a goal key
    def search(self, goal):
        node, p= self.__find(goal)
        return node.data if node else None
```

Tree Efficiency

- If the tree is balanced, the time complexity is O(log N) time
- If the tree is entirely imbalanced, and the time complexity is O(N)

Inserting A Node

- To insert a node key (31)
- Traverse to find the appropriate location
- Find(31)
- The function __find(31) terminates when the parent node is located at the leaf node with key 33

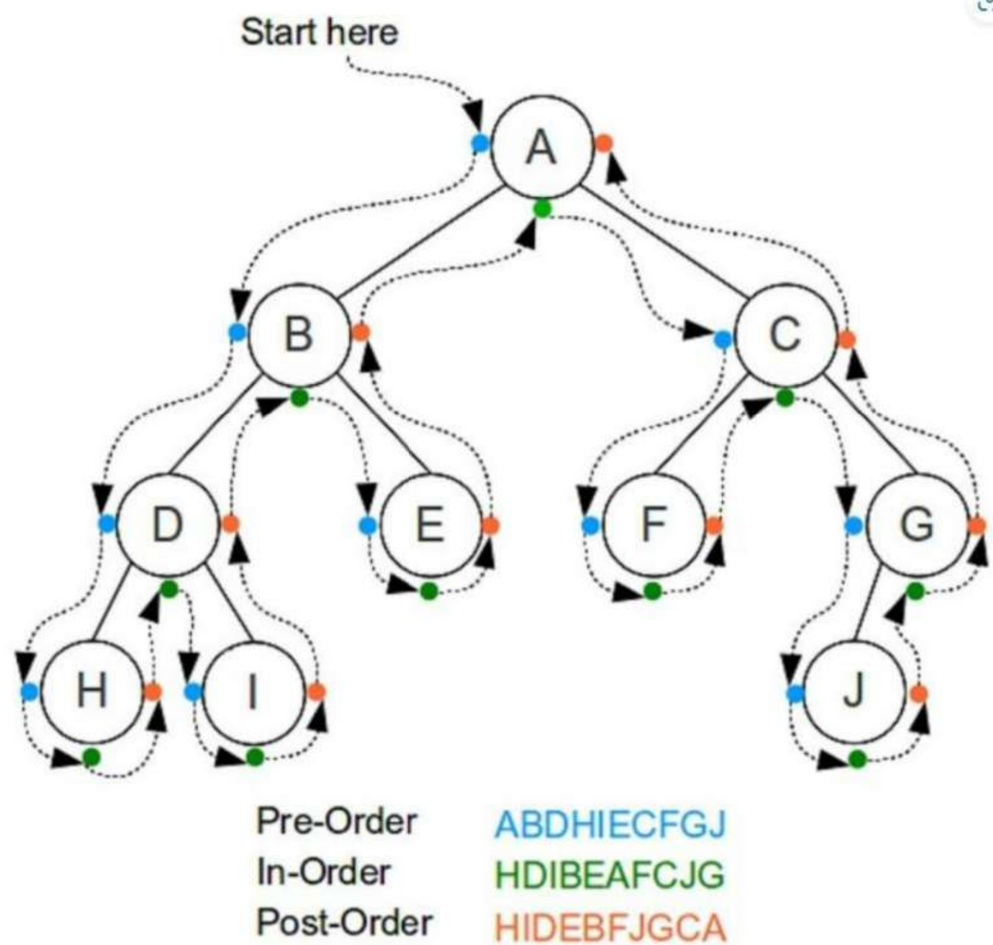


The insert() Method

```python
    # Insert a new node in the binary tree
    def insert(self, key, data):
        node, parent = self.__find(key) # get key and its parent
        if node:     # if we find a node
            node.data = data     # update node's data
            return False         # return flag for no insertion

        if parent is self:       # for empty trees, insert new node
            self.__root =  self.__Node(key, data)
        elif key < parent.key:     # insert left
            parent.leftChild = self.__Node(key, data, left = node)
        else:                    # insert right
            parent.rightChild = self.__Node(key, data, right = node)
        return True
```

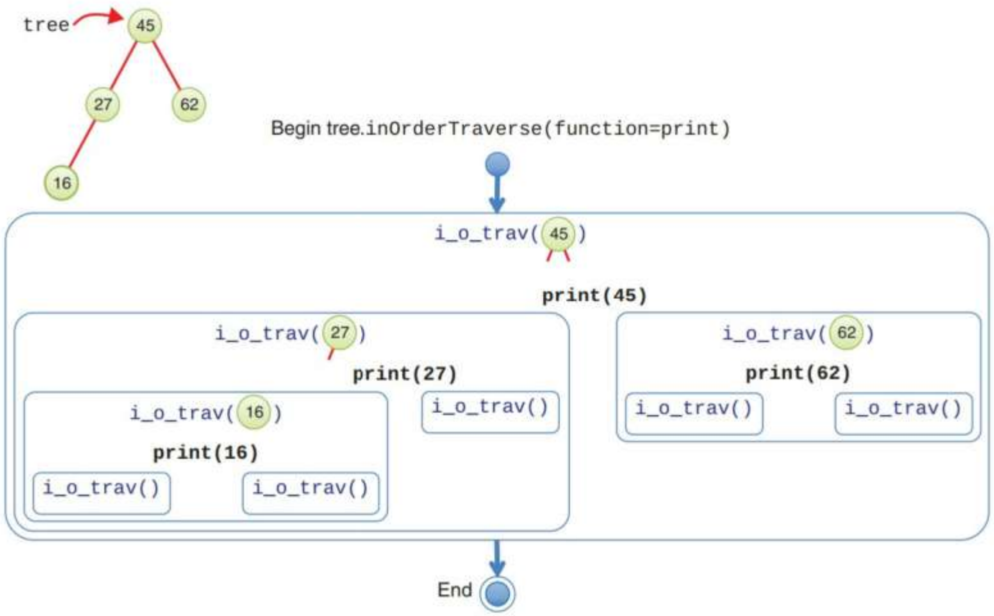
Traversing A Tree

- Pre-Order
- In-Order
- Post-Order



Traversal Involves Recursion

1. Call itself to traverse the node's left subtree.
2. Visit the node.
3. Call itself to traverse the node's right subtree.

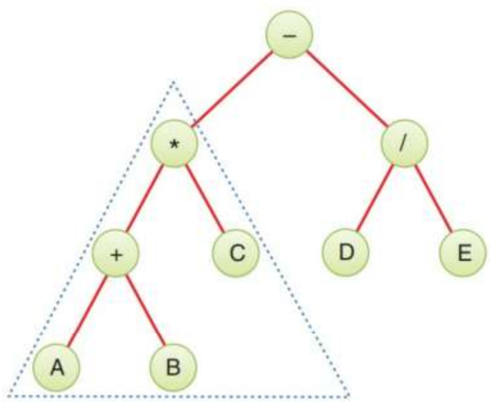


Python Code for In-Order Traversal

```python
class BinarySearchTree(object):
    # In-Order Traversal
    def inOderTraverse(self, function = print):
        # call recursive version, start at root
        self.__inOrderTraverse(self.__root, function= function)

    # Recursive version on sub trees
    def __inOrderTraverse(self, node, function):
        if node:
            self.__inOrderTraverse(node.leftChild, function) # process left sub tree
            function(node)  # visit node (print)
            self.__inOrderTraverse(self.rightChild, function)   # pocess right sub tree
```

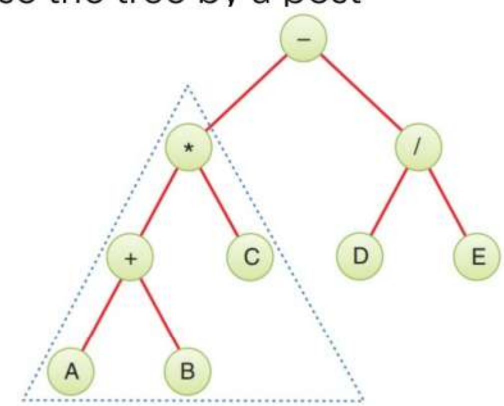
What purpose does having three traversal orders serve?

- In-order traversal ensures that the keys in binary search trees are arranged in climbing order.
- Pre and post order are highly beneficial when developing systems that engage in the parsing or analysis of algebraic expressions.
- For instance, we can represent the infix expression (A+B) * C - D / E as shown.



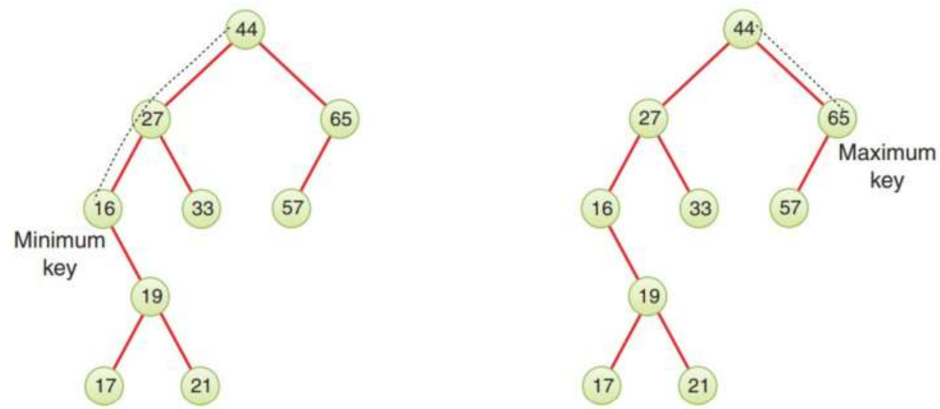
What purpose does having three traversal orders serve?

- In-order traversal ensures that the keys in binary search trees are arranged in climbing order.
- Pre and post order are highly beneficial when developing systems that engage in the parsing or analysis of algebraic expressions.
- For instance, we can represent the infix expression (A+B) * C - D / E as shown.
- To produce the equivalent postfix expression, all we need is to traverse the tree by a post-order traversal **AB+C*DE/–**



Finding Minimum and Maximum Key Values

- Finding Minimum involves processing The left branch.
- The opposite goes for Maximum



Finding Minimum and Maximum Key Values

```python
class BinarySearchTree(object):
    # finding minmum key value
    def minNode(self):
        if self.isEmpty():
            raise Exception('No minimum node in empty tree')

        node = self.__root              # start at root
        while node.leftChild:           # while has a left child
            node = node.leftChild       # follow left child reference
        return (node.key, node.data)
```

Deleting A Node

01

The node to be deleted is a leaf (has no children).

02

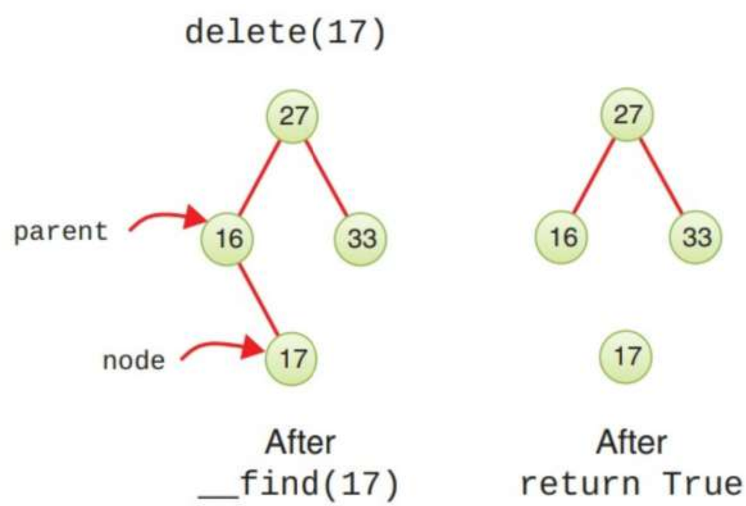
The node to be deleted has one child.

03

The node to be deleted has two children.

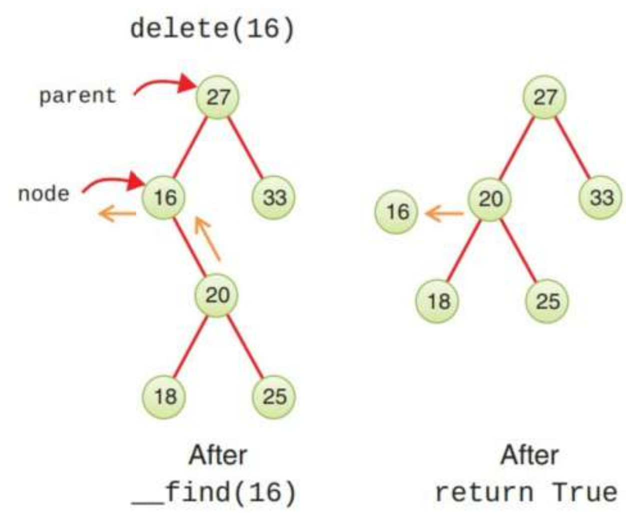
Case 1: The Node to Be Deleted Has No Children

- Modify the parent to have no children

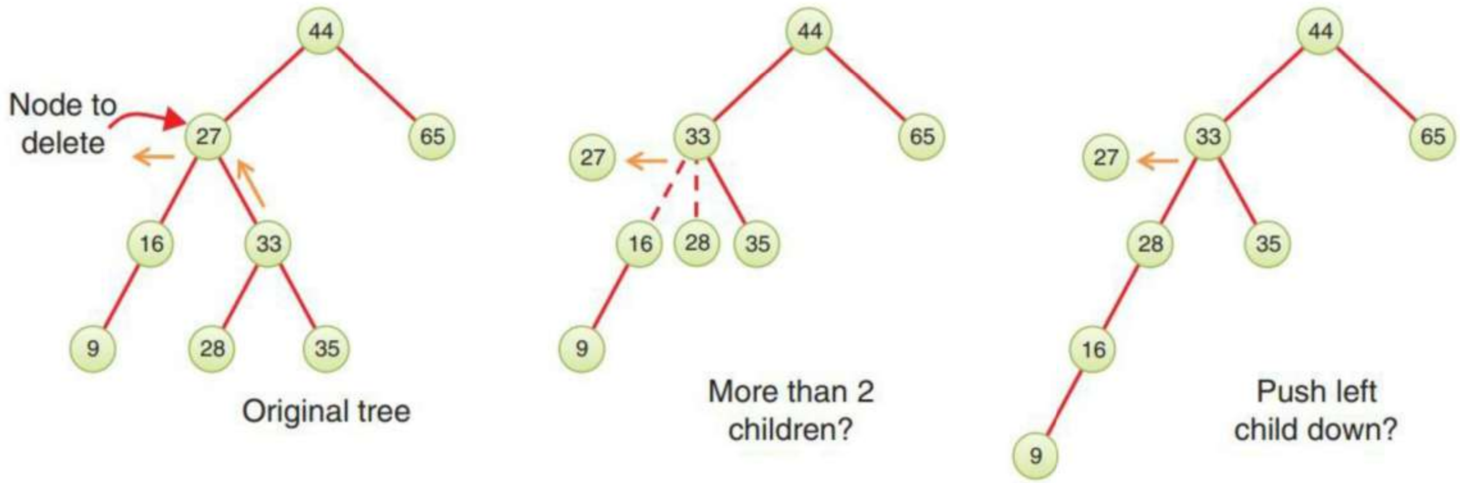


Case 2: The Node to Be Deleted Has One Child

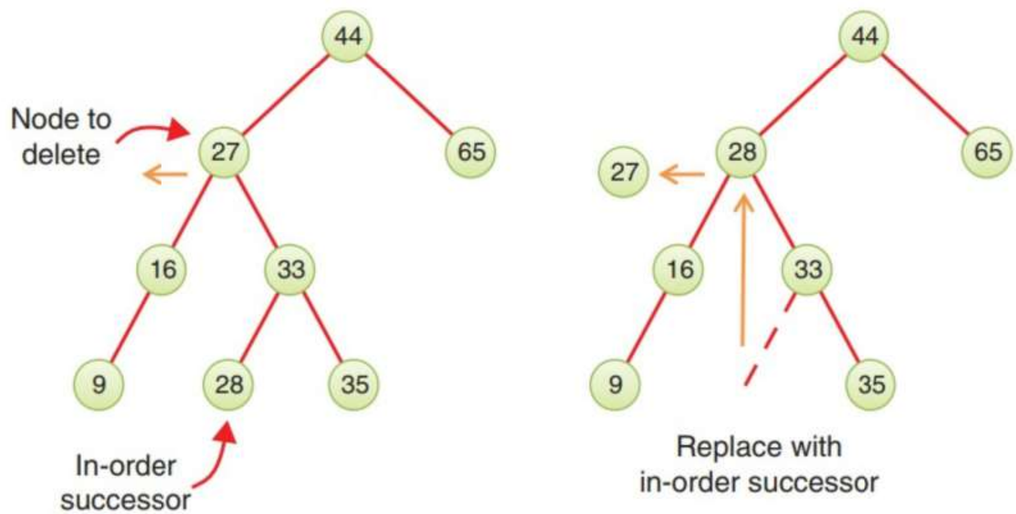
- Find(node, parent)
- If the node has only a left child:
  - parent's left = node's left
  - Parent's right = node's left
- If the node has only a right child:
  - Parent's left=node's right
  - Parent's right = node's right



Case 3: The Node to Be Deleted Has Two Children



Case 3: The Node to Be Deleted Has Two Children



Printing Trees

```python
class BinarySearchTree(object):

    # Printing Trees
    def print(self, indentBy= 4): # indent each level by some blanks
        # start at root with no indent
        self.__pTree(self.__root, 'ROOT:    ', "", indentBy)

    # Recursively print a subtree, sideways with the root node left justified
    # nodeType shows the relation to its
    # parent and the indent shows its level
    # Increase indent level for subtrees
    def __pTree(self, node, nodeType, indent, indentBy=4):
        if node:
            self.__pTree(node.rightChild, "RIGHT: ", # Print the right
                         indent + " " * indentBy, indentBy) # subtree
            print(indent + nodeType, node) # Print this node
            self.__pTree(node.leftChild, "LEFT: ", # Print the left
                         indent + " " * indentBy, indentBy) # subtree
```

Thanks