What is a Tree?

A tree is a hierarchical data structure, consists of nodes (vertices) that are connected using pointers (edges). Trees are similar to Graphs; the key differentiating point is that a cycle cannot exist in a Tree.

The basic structure of a tree consists of the following components:

Nodes: Hold data
Root: The uppermost node of a tree
Parent Node: A node which is connected to one or more nodes on the lower level (Child Nodes)
Child Node: A node which is linked to an upper node (Parent Node)
Sibling Node: Nodes that have the same Parent Node
Leaf Node: A node that doesn’t have any Child Node

Terminology and Formulas

Some other common terminologies used in trees are:

- **Sub-tree**: For a particular non-leaf node, a collection of nodes, essentially the tree, starting from its child node. The tree formed by a node and its descendants.
- **Degree of a node**: Total number of children of a node
- **Length of a path**: The number of edges in a path
- **Depth of a node *n***: The length of the path from a node *n* to the root node. The depth of the root node is 0.
- **Level of a node *n***: (Depth of a Node)+1
- **Height of a node *n***: The length of the path from *n* to its deepest descendant. So the height of the tree itself is the height of the root node and the height of leaf nodes is always 0.
- **Height of a Tree**: Height of its root node

Types of Trees

Trees being advanced data structures, offer a wide variety of types to provide an efficient solution, specific to a particular use. Trees are extensively used in Artificial Intelligence and complex algorithms to provide an efficient storage mechanism for problem-solving. Based on the structure, height, and other features like time/space complexity, there are different types of trees.

The most commonly used types are listed below:

- Binary Trees
- BinarySearchTree
- AVL Tree
- Red-Black Tree
- 2-3 Tree

The N-ary Tree

In graph theory, an N-ary tree is a rooted tree in which each node has no more than N children. It is also sometimes known as a k-way tree, a k-ary tree, or an M-ary tree. A binary tree is a special case where k=2, so they can have a maximum of **2** child nodes and a minimum of **0** child nodes.

Balanced Tree

A balanced tree is a tree in which almost all leaf nodes are present at the same level. This condition is generally applied to all sub-trees. This means that all the sub-trees in a tree need to be balanced, no matter how many there are. Mathematically, it can be expressed as:

Height(Tree)= O(log_2 (nodes))Height(Tree)=O(log2(nodes))

Or in simpler words, make the tree "height-balanced"; i.e. the difference between the height of the left and right sub-trees of each node should not be more than one. Mathematically, it can be written as:

|Height(LeftSubTree) - Height(RightSubTree) |<= 1

High-level Algorithm to determine if a tree is height-balanced

1. Start from the leaf nodes and move towards the root
2. Along with traversing the tree, compute heights of the *left-subtree* and *right-subtree* of each node. The height of a leaf node is always **0**
3. At each node, check if the difference between the height of the left and right sub-tree is more than **1**, if so, it means that the tree is not balanced.
4. If you have completely traversed the tree and haven't caught the above condition, then the tree is balanced.

What is a Binary Tree?

A binary tree is a tree in which each node has between 0-2 children. They're called the left and right children of the node.

Types of Binary Trees

Complete Binary Trees

A *complete binary tree* is a binary tree in which all the levels of the tree are fully filled, except for perhaps the last level which can be filled from left to right.

Full Binary Trees

- In a full or 'proper' binary tree, every node has **0** or **2** children. No node can have 1 child.
- The total number of nodes in a full binary tree of height '**h**' can be expressed as:

  $2h+1 \leq$ total number of nodes $\leq [2^{(h+1)}] - 1$

Perfect Binary Trees

A Binary tree is said to be **Perfect** if all its internal nodes have two children and all leaves are at the same level. Also note that,

- the total number of nodes in a perfect binary tree of height **'h'** are given as: [2^(h+1)]−1
- the total number of leaf nodes are given as 2^h or (n+1)/2

There are many other advanced trees derived from the basic structure of binary trees. These types will be discussed in the upcoming lessons. Some of the most common ones are:

- Complete Binary Tree
- Skewed Binary Tree
- Binary Search Tree
- AVL Tree

Complete Binary Tree:

Here are some more detailed properties of them.

- All the levels are completely filled except possibly the last one
- Nodes at the last level are as far left as possible
- The total number of nodes, n, in a complete binary tree of height **"h"** are: 2h ≤ nodes ≤ 2h+1−1 (This is again based on the Geometric Series formula)
- The total number of non-leaf nodes,ni in a complete binary tree of height "h" are expressed as a range like so:

  2^(h−1) ≤ ni ≤ [2^h] − 1

- The total number of leaf-nodes,ne in a complete binary tree of height **"h"** is expressed as a range like so:

  2^(h−1) ≤ ne ≤ 2^h

- The nodes, n, are present in between the range of:

  2^h ≤ n ≤ (2^h+1) − 1

Insertion in Complete Binary Trees

The following rules apply when inserting a value in a Complete Binary Tree:

- Nodes are inserted level by level
- Fill in the left-subtree before moving to the right one

Explanation

As you can see in the animation above, Node 4 was inserted as a left child of Node 2 to meet the property of complete binary trees. In a Complete Binary Tree there exist no node that has a right child but not a left child. So during Insertion, make sure to insert a node as a left child first if it’s empty to fill in the left sub-tree before moving to right sub-tree.

Skewed Binary Trees:

Skewed Binary Trees are Binary trees such that all the nodes except one have one and only one child. All of the children nodes are either left or right child nodes so the entire tree is positioned to the left or the right side. This type of Binary Tree structure should be avoided at all costs because the time complexity of most operations will be high.

Left-Skewed Binary Trees

The two types of Skewed Binary Trees are based on which side the tree leans towards. The left-skewed binary tree has all left child nodes.

Right-Skewed Binary Trees

Right skewed binary trees have all right nodes.

Binary Search Tree (BST):

Binary Search Trees (BSTs) are a special kind of binary tree where each node of the tree has key-value pairs. These key-value pairs can be anything, like (username,bank) or (employee,employeeID). For all the nodes in a BST, the values of all the keys in the left sub-tree of the node are less than the value of the nodes themselves. All the keys in the right subtree are greater than the values of the node. This is referred to as the BST rule.

NodeValues(leftsubtree) <= CurrentNodeValue < NodeValues(rightsubtree)

Implementing a Binary Search Tree in Python

```
Class Node:
        def __init__(self, val):
                self.val = val
                self.left_child = None
                self.right_child = None
                self.parent = None

Class Binary_search_tree:
        def __init__(self, val):
                self.root = Node(val)

BST = Binary_search_tree(6)
print(BST.root.val)
```

OUTPUT:

```
6
```

Binary Search Tree Insertion Algorithm

1. Start from the root node
2. Check if the value to be inserted is greater than the root/current node's value

3. If yes, then repeat the steps above for the right subtree, otherwise repeat the steps above for the left sub-tree of the current node.
4. Repeat until you find a node that has no right/left child to move onto. Insert the given value there and update the parent node accordingly.

Binary Search Tree Insertion (Implementation)

There are two ways to code a BST insert function

- *Iteratively*
- *Recursively*

Insert Implementation (Iterative)

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None
        self.parent = None

    def insert(self, val):
        current = self
        parent = None
        while current:
            parent = current
            if val < current.val:
                current = current.left_child
            else:
                current = current.right_child
        if val < parent.val:
            parent.left_child = Node(val)
        else:
            parent.right_child = Node(val)
```

```
Class Binary_search_tree:
    def __init__(self, val):
        self.root = Node(val)

    def insert(self, val):
        if self.root:
            return self.root.insert(val)
        else:
            self.root = Node(val)
            return True

import random

def display(node):
    lines, _, _, _ = display_aux(node)
    for line in lines:
        print(line)

def _display_aux(node):
    # Return list of strings, width, height and horizontal coordinate of the root
    # No child
    if node.right_child is None and node.left_child is None:
        line = str(node.val)
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    # Only left child
    if node.right_child is None:
        lines, n, p, x = _display_aux(node.left_child)
        s = str(node.val)
        u = len(s)
        first_line = (x + 1) * ‘ ’ + (n – x – 1) * ‘_’ + s
        second_line = x * ‘ ‘ + ‘/ ’ + (n – x – 1 + u) * ‘ ‘
        shift_lines = [line + u * ‘ ‘ for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2
        # Only right child
        if node.left_child is None:
            lines, n, p, x = _display_aux(node.right_child)
            s = str(node.val)
            u = len(s)
            first_line = s + x * ‘_’ + (n – x) * ‘ ‘
            second_line = (u + x) * ‘ ‘ + ‘ \\ ’ + (n – x – 1) * ‘ ‘
            shifted_lines = [u * ‘ ‘ + line for line in lines]
            final_lines = [first_line, second_line] + shifted_lines
            return final_lines, n + u, p + 2, u // 2

        # Two children
        left, n, p, x = _display_aux(node.left_child)
        right, m, q, y = _display_aux(node.right_child)
        s = '%s’ % node.val
        u = len(s)
        first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_‘ + s + y * ‘_’ + (m – y) * ‘ ‘
        second_line = x * ‘ ‘ + ‘ / ’ + (n – x – 1 + u + y) * ‘ ‘ + ‘ \\‘ + (m – y – 1) *
‘ ‘
        if p < q:
            left += [n * ‘ ‘] * (q – p)
        elif q < p:
            right += [m * ‘ ‘] * (p – q)
        zipped_lines = zip(left, right)
        lines = [first_line,second_line] + [a+u * ‘ ‘ + b for a, b in zipped_lines]
        return lines, n + m + u, max(p, q) + 2, n + u // 2
```

```
BST = Binary_search_tree(50)
for _ in range(15):
    ele = random.randint(0, 100)
    print(“Inserting “ + str(ele) + “:”)
    BST.insert(ele)
    # We have hidden the code for this function but it is available for use!
    display(BST.root)
    print(‘\n’)
```

Insert Implementation (Recursive)

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None

    def insert(self, val):
        if val < self.val:
            if self.left_child:
                self.left_child.insert(val)
            else:
                self.left_child = Node(val)
                return
        else:
            if self.right_child:
                self.right_child.insert(val)
            else:
                self.right_child = Node(val)
                return
```

```
Class Binary_search_tree:
    def __init__(self, val):
        self.root = Node(val)

    def insert(self, val):
        if self.root:
            return self.root.insert(val)
        else:
            self.root = Node(val)
            return True

import random
def display(node):
    lines, _, _, _ = _display_aux(node)
    for line in lines:
        print(line)

def _display_aux(node):
#Returns list of strings, width, height, and horizontal coordinate of the root
    #No child
    if node.right_child is None and node.left_child is None:
        line = ‘%s’ % node.val
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    #Only left child
    if node.right_child is None and node.left_child is None:
        lines, n, p, x = _display_aux(node.left_child)
        s = ‘%s’ % node.val
        u = len(s)
        first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_’ + s
```

```
            second_line = x * ‘ ‘ + ‘/’ + (n – x – 1 + u) * ‘ ‘
            shifted_lines = [line + u * ‘ ‘ for line in lines]
            final_lines = [first_line, second_line] + shifted_lines
            return final_lines, n + u, p + 2, n + u // 2

        #Only right child
        if node.left_child is None:
            lines, n, p, x = _display_aux(node.right_child)
            s = ‘%s’ % node.val
            u = len(s)
            first_line = s + x * ‘_’ + (n – x) * ‘ ‘
            second_line = (u + x) * ‘ ‘ + ‘\\’ + (n – x – 1) * ‘ ‘
            shifted_lines = [u * ‘ ‘ + line for line in lines]
            final_lines = [first_line, second_line] + shifted_lines
            return final _lines, n + u, p + 2, u // 2

        #Two children
        left, n, p, x = _display_aux(node.left_child)
        right, m, q, y = _display_aux(node.right_child)
        s = ‘%s’ % node.val
        u = len(s)
        first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_’ + s + y * ‘_’ + (m – y) * ‘ ‘
        second_line = x * ‘ ‘ + ‘/’ + (n – x – 1 +u + y) * ‘ ‘ + ‘ \\ ’ + (m – y – 1) * ‘ ‘
        if p < q:
            left += [n * ‘ ‘] * (q – p)
        elif q < p:
            right += [m * ‘ ‘] * (p – q)
        zipped_lines = zip(left, right)
        lines = [first_line, second_line] + [a + u * ‘ ‘  + b for a, b in zipped_lines]
        return lines, n + m + u, max(p, q) + 2, n + u // 2

BST = Binary_search_tree(50)
for _ in range(15):
        ele = random.randint(0, 100)
        print(“Inserting “ + str(ele) + “:”)
        BST.insert(ele)
        display(BST.root)
        print(‘\n’)
```

Searching in a Binary Search Tree (Implementation)

Here is a high-level description of the algorithm:

1. Set the ‘current node’ equal to root.
2. If the value is less than the ‘current node’s’ value, then move on to the left-subtree otherwise move on to the right sub-tree.
3. Repeat until the value at the ‘current node’ is equal to the value searched or it becomes None.
4. Return the current node.

Iterative Search Implementation

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None

    def insert(self, val):
        if self is None:
            self = Node(val)
            return
        current = self
        while current:
            parent = current
            if val = current.val:
                current = current.left_child
            else:
                current = current.right_child

        if val < parent.val:
            parent.left_child = Node(val)
        else:
            parent.right_child = Node(val)

    def search(self, val):
        current = self
        while current is not None:
            if val < current.val:
                current = current.left_child
            elif val > current.val:
                current = current.right_child
            else:
                return True
        return False

Class Binary_search_tree:
    def __init__(self, val):
        self.root = Node(val)

    def insert(self, val):
        if self.root:
            return self.root.insert(val)
        else:
            self.root = Node(val)
            return True
    def search(self, val):
        if self.root:
            return self.root.search(val)
        else:
            return False

import random
def display(node):
    lines, _, _, _ = display_aux(node)
    for line in lines:
        print(line)

def _display_aux(node):
    # Return list of strings, width, height and horizontal coordinate of the root
    # No child
    if node.right_child is None and node.left_child is None:
        line = str(node.val)
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    # Only left child
    if node.right_child is None:
        lines, n, p, x = _display_aux(node.left_child)
        s = str(node.val)
        u = len(s)
        first_line = (x + 1) * ‘ ’ + (n – x – 1) * ‘_’ + s
        second_line = x * ‘ ‘ + ‘/ ’ + (n – x – 1 + u) * ‘ ‘
        shift_lines = [line + u * ‘ ‘ for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2
        # Only right child
        if node.left_child is None:
            lines, n, p, x = _display_aux(node.right_child)
            s = str(node.val)
```

```
                u = len(s)
                first_line = s + x * ‘_’ + (n – x) * ‘ ‘
                second_line = (u + x) * ‘ ‘ + ‘ \\ ’ + (n – x – 1) * ‘ ‘
                shifted_lines = [u * ‘ ‘ + line for line in lines]
                final_lines = [first_line, second_line] + shifted_lines
                return final_lines, n + u, p + 2, u // 2

            # Two children
            left, n, p, x = _display_aux(node.left_child)
            right, m, q, y = _display_aux(node.right_child)
            s = '%s’ % node.val
            u = len(s)
            first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_‘ + s + y * ‘_’ + (m – y) * ‘ ‘
            second_line = x * ‘ ‘ + ‘ / ’ + (n – x – 1 + u + y) * ‘ ‘ + ‘ \\‘ + (m – y – 1) *
‘ ‘
            if p < q:
                left += [n * ‘ ‘] * (q – p)
            elif q < p:
                right += [m * ‘ ‘] * (p – q)
            zipped_lines = zip(left, right)
            lines = [first_line,second_line] + [a+u * ‘ ‘ + b for a, b in zipped_lines]
            return lines, n + m + u, max(p, q) + 2, n + u // 2

BST = Binary_search_tree(50)
for _ in range(15):
    ele = random.randint(0, 100)
    print(“Inserting “ + str(ele) + “:”)
    BST.insert(ele)
    # We have hidden the code for this function but it is available for use!
display(BST.root)
print(‘\n’)
print(BST.search(50))
print(BST.search(11))
```

Recursive Search Implementation

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None

    def insert(self, val):
        if val < self.val:
            if self.left_child:
                self.left_child.insert(val)


            else:
                self.left_child = Node(val)
                return
        else:
            if self.right_child:
                self.right_child.insert(val)
            else:
                self.right_child = Node(val)
                return

    def search(self, val):
        if val < self.val:
            if self.left_child:
                return self.left_child.search(val)
            else:
                return False
        elif val > self.val:
            if self.right_child:
                return self.right_child.search(val)
                else:
                    return False
            else:
                return True
            return False

Class Binary_search_tree:
    def __init__(self, val):
        self.root = Node(val)

    def insert(self, val):
        if self.root:
            return self.root.insert(val)
        else:
            self.root = Node(val)
            return True

    def search(self, val):
        if self.root:
            return self.root.search(val)
        else:
            return False
import random
def display(node):
    lines, _, _, _ = _display_aux(node)
    for line in lines:
        print(line)
def _display_aux(node):
#Returns list of strings, width, height, and horizontal coordinate of the root
    #No child
    if node.right_child is None and node.left_child is None:
        line = str(node.val)
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle
#Only left child
if node.right_child is None:
        lines, n, p, x = _display_aux(node.left_child)
        s = str(node.val)
        u = len(s)
        first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_’ + s
        second_line = x * ‘ ‘ + ‘/’ + (n – x – 1 + u) * ‘ ‘
        shifted_lines = [line + u * ‘ ‘ for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2
#Only right child
if node.left_child is None:
        lines, n, p, x = _display_aux(node.right_child)
        s = str(node.val)
        u = len(s)
        first_line = s + x * ‘_’ + (n – x) * ‘ ‘
        second_line = (u + x) * ‘ ‘ + ‘\\’ + (n – x – 1) * ‘ ‘
        shifted_lines = [u * ‘ ‘ + line for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final _lines, n + u, p + 2, u // 2

#Two children
left, n, p, x = _display_aux(node.left_child)
right, m, q, y = _display_aux(node.right_child)
s = ‘%s’ % node.val
u = len(s)
first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_’ + s + y * ‘_’ + (m – y) * ‘ ‘
second_line = x * ‘ ‘ + ‘/’ + (n – x – 1 +u + y) * ‘ ‘ + ‘ \\ ’ + (m – y – 1) * ‘ ‘
if p < q:
        left += [n * ‘ ‘] * (q – p)
    elif q < p:
        right += [m * ‘ ‘] * (p – q)
    zipped_lines = zip(left, right)
    lines = [first_line, second_line] + [a + u * ‘ ‘  + b for a, b in zipped_lines]
    return lines, n + m + u, max(p, q) + 2, n + u // 2

BST = Binary_search_tree(50)
for _ in range(15):
    ele = random.randint(0, 100)
    BST.insert(ele)
display(BST.root)
print(‘\n’)
print(BST.search(15))
print(BST.search(50))
```

Deletion in a Binary Search Tree

To delete a node in a BST, you will search for it and, once found, you’ll make it None by making the left or right child of its parent None. However, to make things simpler, we’ve identified six possible cases involved in BST node deletion. We’ll tackle each one separately.

1. Deleting in an empty tree
2. Deleting a node with no children, i.e., a leaf node.
3. Deleting a node which has one child only
    1. Deleting a node which has a right child only
    2. Deleting a node which has a left child only
4. Deleting a node with two children

1. Deleting an empty tree

If the given starting node is Null then do nothing and return False. This is an edge case for error handling.

2. Deleting a Leaf Node

When the node to be deleted is a leaf node in a Binary Search Tree, we simply remove that leaf node. We do this by making the parent node's left or right child (whichever one the leaf node was) None.

3. Deleting a node which has one child

We search for the node, once the node is found we check if and how many children it has. If it has only one child, we check the parent node to see if the current node is the left or right child and then replace its child node with the current node.

4. Deleting a node with two children

1. From the given node to be deleted, find either the node with the smallest value in the right sub-tree or the node with the largest value in the left sub-tree. Suppose you want to find the smallest value in the right sub-tree; you do this by moving on to every node's left child until the last left child is reached.
2. Replace the node to be deleted with the node found (the smallest node in the right sub-tree or the largest node in the left sub-tree).
3. Finally, delete the node found (the smallest in the right sub-tree).

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None
    def insert(self, val):
        if val < self.val:
            if self.left_child:
                self.left_child.insert(val)
            else:
                self.left_child = Node(val)
                return
        else:
            if self.right_child:
                self.right_child.insert(val)
            else:
                self.right_child = Node(val)
                return

def search(self, val):
    if val < self.val:
        if self.left_child:
            return self.left_child.search(val)
        else:
            return False
    elif val > self.val:
        if self.right_child:
            return self.right_child.search(val)
        else:
            return False
    else:
        return True
    return False
def delete(self, val):
    if val < self.val:
        if self.left_child:
            self.left_child = self.left_child.delete(val)
        else:
            print(str(val) + “not found in the tree”)
            return self
        elif val > self.val:
            if self.right_child:
                self.right_child = self.right_child.delete(val)
                else:
                    print(str(val) + “not found in the tree”)
                    return self
            else:
                if self.left_child is None and self.right_child is None:
                    self = None
                    return None
                elif self.left_child is None:
                    tmp = self.right_child
                    self = None
                    return tmp
                else:
                    current = self.right_child
                    while current.left_child is not None:
                        current = current.left_child
                    self.val = current.val
                    self.right_child= self.right_child.delete(current.val)
        return self

Class Binary_search_tree:
    def __init__(self, val):
        self.root = Node(val)

    def insert(self, val):
        if self.root:
            return self.root.insert(val)
        else:
            self.root = Node(val)
            return True

    def search(self, val):
        if self.root:
            return self.root.search(val)
        else:
            return False

    def delete(self, val):
        if self.root is not None:
            self.root = self.root.delete(val)

import random
def display(node):
    lines, _, _, _ = _display_aux(node)
    for line in lines:
        print(line)

def _display_aux(node):
#Returns list of strings, width, height, and horizontal coordinate of the root
    #None
    if node is None:
        line = ‘Empty tree!’
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    if node.right_child is None and node.left_child is None:
        line = str(node.val)
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    #Only left child
    if node.right_child is None:
        lines, n, p, x = _display_aux(node.left_child)
        s = str(node.val)
        u = len(s)
        first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_’ + s
        second_line = x * ‘ ‘ + ‘/’ + (n – x – 1 + u) * ‘ ‘
        shifted_lines = [line + u * ‘ ‘ for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2

    #Only right child
    if node.left_child is None:
        lines, n, p, x = _display_aux(node.right_child)
        s = str(node.val)
        u = len(s)
        first_line = s + x * ‘_’ + (n – x) * ‘ ‘
        second_line = (u + x) * ‘ ‘ + ‘\\’ + (n – x – 1) * ‘ ‘
        shifted_lines = [u * ‘ ‘ + line for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final _lines, n + u, p + 2, u // 2

    #Two children
    left, n, p, x = _display_aux(node.left_child)
    right, m, q, y = _display_aux(node.right_child)
    s = ‘%s’ % node.val
    u = len(s)
    first_line = (x + 1) * ‘ ‘ + (n – x – 1) * ‘_’ + s + y * ‘_’ + (m – y) * ‘ ‘
    second_line = x * ‘ ‘ + ‘/’ + (n – x – 1 +u + y) * ‘ ‘ + ‘ \\ ’ + (m – y – 1) * ‘ ‘
    if p < q:
        left += [n * ‘ ‘] * (q – p)
    elif q < p:
        right += [m * ‘ ‘] * (p – q)
    zipped_lines = zip(left, right)
    lines = [first_line, second_line] + [a + u * ‘ ‘  + b for a, b in zipped_lines]
    return lines, n + m + u, max(p, q) + 2, n + u // 2
```

```
BST = Binary_search_tree(6)
BST.insert(3)
BST.insert(2)
BST.insert(4)
BST.insert(-1)
BST.insert(1)
BST.insert(-2)
BST.insert(8)
BST.insert(7)
print(“before deletion:”)
display(BST.root)
BST.delete(10)
print(“after deletion:”)
display(BST.root)
```

Pre-Order Traversal

In this traversal, the elements are traversed in “root-left-right” order. We first visit the root/parent node, then the left child, and then the right child. Here is a high-level description of the algorithm for *Pre-Order* traversal, starting from the root node:

1. Visit the current node, i.e., print the value stored at the node
2. Call the preOrderPrint() function on the left sub-tree of the ‘current Node
3. Call the preOrderPrint() function on the right sub-tree of the ‘current Node

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None
```

```
def insert(self, val):
    if self is None:
        self = Node(val)
        return
    current = self
    while current:
        parent = current
        if val < current.val:
            current = current.left_child
        else:
            current = current.right_child
    if val < parent.val:
        parent.left_child = Node(val)
    else:
        parent.right_child = Node(val)

def search(self, val):
    if self is None:
        return self
    current = self
    while current and current.val != val:
        if val < current.val:
            current = current.left_child
        else:
            current = current.right_child
    return current
def copy(self, node2):
    self.val = node2.val
    if node2.left_child:
        self.left_child = node2.left_child
    if node2.right_child:
        self.right_child = node2.right_child
```

```
def delete(self, val):
    if self is None:
        return False
    node = self
    while node and node.val != val:
        parent  = node
        if val < node.val:
            node = node.left_child
        else:
            node = node.right_child

    if node is None or node.val != val:
        return False

    elif node.left_child is None and node.right_child is None:
        if val < parent.value:
            parent.left_child = None
        else:
            parent.right_child = None
        return True

    elif node.left_child is None and node.right_child is None:
        if val < parent.value:
            parent.left_child = None
        else:
            parent.right_child = None
        return True

    elif node.left_child and node.right_child is None:
        if parent is None:
        ‘’’ Have to create a deepcopy because ‘self’ is local variable
        and changing it will not overwrite ‘root’ in the
    binary_search_tree class’’’
```

```
            self.copy(self.left_child)
            self.left_child = None
        elif val < parent.val:
            parent.left_child = node.left_child
        else:
            parent.right_child = node.left_child
        return True
    elif node.right_child and node.left_child is None:
        if parent is None:
            self.copy(self.right_child)
            self.right_child = None
        elif val < parent.val:
            parent.left_child = node.right_child
        else:
            parent.right_child = node.right_child
        return True
    else:
        replace_node_parent = node
        replace_node = node.right_child
        while replace_node.left_child:
            replace_node_parent = replace_node
            replace_node = replace_node.left_child

        node.val = replace_node.val
        if replace_node.right_child:
            if replace_node_parent.val > replace_node.val:
                replace_node_parent.left_child=replace_node.right_child
        elif replace_node_parent.val < replace_node.val:
            replace_node_parent.right_child = replace_node.right_child
        else:
```

```
                    if replace_node.val < replace_node_parent.val:
                        replace_node_parent.left_child = None
                    else:
                        replace_node_parent.right_child = None

Class Binary_search_tree:
    def __init__(self, val):
        self.root = Node(val)
    def set_root(self, val):
        self.root = Node(val)
    def get_root(self):
        return self.root.get()
    def insert(self, val):
        self.root.insert(val)
    def search(self, val):
        return self.root.search(val)

def  pre_order_print(node):
    if node is not None:
        print(node.val)
        pre_order_print(node.left_child)
        pre_order_print(node.right_child)


BST = Binary_search_tree(6)
BST.insert(4)
BST.insert(9)
BST.insert(5)
BST.insert(2)
BST.insert(8)
BST.insert(12)


pre_order_print(BST.root)
```

Post-Order Traversal

In post-order traversal, the elements are traversed in “left-right-root” order. We first visit the left child, then the right child, and then finally the root/parent node. Here is a high-level description of the post-order traversal algorithm,

1. Traverse the left sub-tree of the ‘currentNode’ recursively by calling the postOrderPrint() function on it
2. Traverse the right sub-tree of the ‘currentNode’ recursively by calling the postOrderPrint() function on it
3. Visit current node and print its value

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None

    def insert(self, val):
        if self is None:
            self = Node(val)
            return
        current = self
        while current:
            parent = current
            if val < current.val:
                current = current.left_child
            else:
                current = current.right_child
        if val < parent.val:
            parent.left_child = Node(val)
        else:
                parent.right_child = Node(val)
def search(self, val):
        if self is None:
                return self
        current = self
        while current and current.val != val:
                if val < current.val:
                        current = current.left_child
                else:
                        current = current.right_child
        return current
def copy(self, node2):
        self.val = node2.val
        if node2.left_child:
                self.left_child = node2.left_child
        if node2.right_child:
                self.right_child = node2.right_child
def delete(self, val):
        if self is None:
                return False
        node = self
        while node and node.val != val:
                parent  = node
                if val < node.val:
                        node = node.left_child
                else:
                        node = node.right_child

        if node is None or node.val != val:
                return False
        elif node.left_child is None and node.right_child is None:
                if val < parent.value:
```

```
                parent.left_child = None
            else:
                parent.right_child = None
            return True
        elif node.left_child and node.right_child is None:
            if parent is None:
            ‘’’ Have to create a deepcopy because ‘self’ is local variable
            and changing it will not overwrite ‘root’ in the
        binary_search_tree class’’’
            self.copy(self.left_child)
            self.left_child = None
        elif val < parent.val:
            parent.left_child = node.left_child
        else:
            parent.right_child = node.left_child
        return True
    elif node.right_child and node.left_child is None:
        if parent is None:
            self.copy(self.right_child)
            self.right_child = None
        elif val < parent.val:
            parent.left_child = node.right_child
        else:
            parent.right_child = node.right_child
        return True
    else:
        replace_node_parent = node
        replace_node = node.right_child
        while replace_node.left_child:
            replace_node_parent = replace_node
            replace_node = replace_node.left_child
```

```
                node.val = replace_node.val
                if replace_node.right_child:
                        if replace_node_parent.val > replace_node.val:
                                replace_node_parent.left_child=replace_node.right_child
                elif replace_node_parent.val < replace_node.val:
                        replace_node_parent.right_child = replace_node.right_child
                else:
                        if replace_node.val < replace_node_parent.val:
                                replace_node_parent.left_child = None
                        else:
                                replace_node_parent.right_child = None
Class Binary_search_tree:
        def __init__(self, val):
                self.root = Node(val)
        def set_root(self, val):
                self.root = Node(val)
        def get_root(self):
                return self.root.get()
        def insert(self, val):
                self.root.insert(val)
        def search(self, val):
                return self.root.search(val)
def post_order_print(node):
        if node is not None:
                post_order_print(node.left_child)
                post_order_print(node.right_child)

BST = Binary_search_tree(6)
BST.insert(4)
BST.insert(9)
BST.insert(5)
BST.insert(2)
BST.insert(8)
BST.insert(12)

post_order_print(BST.root)
```

In-Order Traversal

In In-order traversal, the elements are traversed in “left-root-right” order so they are traversed *in order*. In other words, elements are printed in sorted ascending order with this traversal. We first visit the left child, then the root/parent node, and then the right child. Here is a high-level description of the in-order traversal algorithm,

1. Traverse the left sub-tree of the ‘currentNode’ recursively by calling the inOrderPrint() function on it.
2. Visit the current node and print its value
3. Traverse the right sub-tree of the ‘currentNode’ recursively by calling the inOrderPrint() function on it.

```
Class Node:
    def __init__(self, val):
        self.val = val
        self.left_child = None
        self.right_child = None

    def insert(self, val):
        if self is None:
            self = Node(val)
            return
        current = self
        while current:
                parent = current
                if val < current.val:
                        current = current.left_child
                else:
                        current = current.right_child
        if val < parent.val:
                parent.left_child = Node(val)
        else:
                parent.right_child = Node(val)

def search(self, val):
        if self is None:
                return self
        current = self
        while current and current.val != val:
                if val < current.val:
                        current = current.left_child
                else:
                        current = current.right_child
        return current

def copy(self, node2):
        self.val = node2.val
        if node2.left_child:
                self.left_child = node2.left_child
        if node2.right_child:
                self.right_child = node2.right_child

def delete(self, val):
        if self is None:
                return False
        node = self
        while node and node.val != val:
                parent  = node
                if val < node.val:
                        node = node.left_child
                else:
                        node = node.right_child

        if node is None or node.val != val:
                return False

        elif node.left_child is None and node.right_child is None:
                if val < parent.value:
                        parent.left_child = None
                else:
                        parent.right_child = None
                return True

        elif node.left_child and node.right_child is None:
                if parent is None:
                ‘’’ Have to create a deepcopy because ‘self’ is local variable
                and changing it will not overwrite ‘root’ in the
        binary_search_tree class’’’
                self.copy(self.left_child)
                self.left_child = None
        elif val < parent.val:
                parent.left_child = node.left_child
        else:
                parent.right_child = node.left_child
        return True

elif node.right_child and node.left_child is None:
        if parent is None:
                self.copy(self.right_child)
                self.right_child = None
            elif val < parent.val:
                parent.left_child = node.right_child
            else:
                parent.right_child = node.right_child
            return True
        else:
            replace_node_parent = node
            replace_node = node.right_child
            while replace_node.left_child:
                replace_node_parent = replace_node
                replace_node = replace_node.left_child

            node.val = replace_node.val
            if replace_node.right_child:
                if replace_node_parent.val > replace_node.val:
                    replace_node_parent.left_child=replace_node.right_child
            elif replace_node_parent.val < replace_node.val:
                replace_node_parent.right_child = replace_node.right_child
            else:
                if replace_node.val < replace_node_parent.val:
                    replace_node_parent.left_child = None
                else:
                    replace_node_parent.right_child = None

Class Binary_search_tree:
    def __init__(self, val):
        self.root = Node(val)
    def set_root(self, val):
        self.root = Node(val)
    def get_root(self):
        return self.root.get()
    def insert(self, val):
        self.root.insert(val)
    def search(self, val):
        return self.root.search(val)

def in_order_print(node):
    if node is not None:
        in_order_print(node.left_child)
        print(node.val)
        in_order_print(node.right_child)

BST = Binary_search_tree(6)
BST.insert(4)
BST.insert(9)
BST.insert(5)
BST.insert(2)
BST.insert(8)
BST.insert(12)

in_order_print(BST.root)
```