# Experiment 5

**Title of the Laboratory Exercise**: Binary Tree

**1. Aim:**

To understand and implement the basic operations in full binary tree using python.

**2. Objective:**

To execute the below operations in a full binary tree:

1. Search − Searches an element in a tree.
2. Insert − Inserts an element in a tree.
3. Pre-order Traversal − Traverses a tree in a pre-order manner.
4. In-order Traversal − Traverses a tree in an in-order manner.
5. Post-order Traversal − Traverses a tree in a post-order manner.

**3. Exercise:**

Construct a full binary tree with 10 nodes, where the data item inserted at every node should be a random value between 1 and 100. Add the following methods to the class named "FullBinaryTree" and perform the operation on the constructed full binary tree.

1. find\_min(): finds the minimum element stored in the constructed Full binary tree.
2. find\_max(): finds the maximum element stored in the constructed Full binary tree.
3. calculate\_sum(): calculates the sum of all elements stored in the constructed Full binary tree.
4. pre\_order\_traversal(): performs pre-order traversal of the constructed Full binary tree.
5. post\_order\_traversal(): performs post-order traversal of the constructed Full binary tree.
6. in\_order\_traversal(): performs in-order traversal of the constructed Full binary tree.

**4. Experimental Procedure**

* 1. **Algorithm design**

import random

class Node:

def init:

data, left, right = None

class FullBinaryTree:

def init:

self.root = None

def construct\_full\_binary\_tree(self, n=10, min\_value=1, max\_value=100):

values = random.randint(min\_value, max\_value)

self.root = constructtree(values, 0)

def \_construct\_tree(self, values, index):

if index < len(values):

node.left = constructtree(values, 2 \* index + 1)

node.right = constructtree(values, 2 \* index + 2)

return node

return None

def find\_min:

return min(data, findmin(node.left), findmin(node.right))

def findmax:

return max(data, findmax(node.left), findmax(node.right))

def calculatesum:

if not node:

return 0

return data + calculatesum(node.left) + calculatesum(node.right)

def pre\_order\_traversal(self, node):

stack = []

current = node

while stack or current:

if current:

stack.append(current)

current = current.left

else:

current = stack.pop()

current = current.right

def post\_order\_traversal(self, node):

stack1 = []

stack2 = []

if node:

stack1.append(node)

while stack1:

current = stack1.pop()

stack2.append(current)

if current.left:

append(current.left)

if current.right:

append(current.right)

def inordertraversal:

stack = []

current = node

while stack or current:

if current:

stack.append(current)

current = current.left

else:

current = stack.pop()

current = current.right

main:

fbt = FullBinaryTree()

constructfullbinarytree()

inordertraversal(root)

preordertraversal(root)

postordertraversal(root)

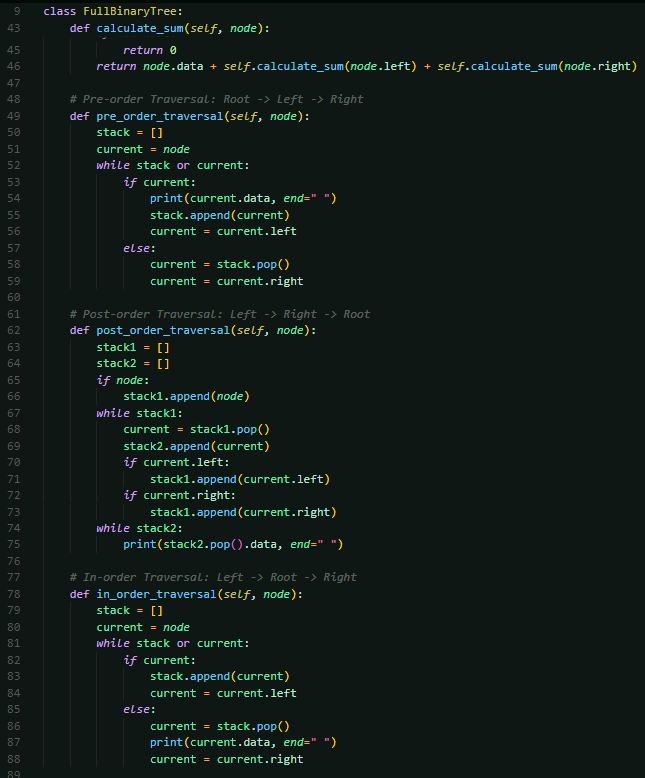
print(findmin(root))

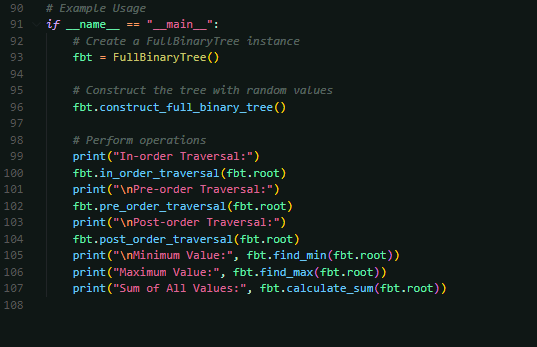
print(findmax(root))

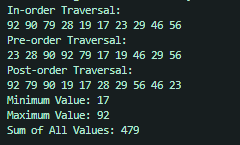
print(calculate\_sum(root))

* 1. **Program**

****

****

****

* 1. **Presentation of the results**
  2. **Analysis and discussions**

**Insert (insert and \_insert):**

**Operation:** Adds a new node to the tree, ensuring the tree remains full.

**Time Complexity**: O(n) (skewed tree)

**Pre-order Traversal**:

**Operation:** Visits nodes in the order: root, left subtree, right subtree

**Time Complexity:** O(n)

**Post-order Traversal**:

**Operation:** Visits nodes in the order: left subtree, right subtree, root

**Time Complexity:** O(n)

**Level-order Traversal**:

**Operation:** Visits nodes level by level from top to bottom and left to right

**Time Complexity**: O(n)

**Branch-wise Traversal**:

**Operation:** Similar to level-order but focuses on nodes at each branch level.

**Time Complexity**: O(n)

**Find Minimum (find\_min**):

**Operation:** Recursively finds the minimum value by comparing the node values

**Time Complexity**: O(n)

**Find Maximum (find\_max):**

**Operation**: Recursively finds the maximum value by comparing the node values.

**Time Complexity**: O(n)

|  |  |
| --- | --- |
| **FUNCTION** | **Time Complexity**: |
| Pre-order Traversal: | O(n) |
| Post-order Traversal: | O(n) |
| Level-order Traversal: | O(n) |
| **Branch-wise** Traversal: | O(n) |
| Find Minimum (find\_min): | O(n) |
| Find Maximum (find\_max): | O(n) |
| Insert (insert and \_insert): | O(n) |

# Experiment 6

**Title of the Laboratory Exercise:** Binary Search Tree

**1. Aim:**

To understand and implement the basic operations in Binary Search Tree using python.

**2. Objective:**

To execute the below operations in a Binary Search Tree (BST):

1. Search − Searches an element in a BST.
2. Insert − Inserts an element in a BST.
3. Delete − Deletes an element in a BST.
4. Check the balance of the BST.
5. Determine the height of the BST.

**3. Exercise:**

Construct a binary search tree with the below values: {12, 35, 14, 97, 36, 65, 89}. Write a python program to perform the following operations:

1. Insert a new element which is having a value equivalent to the “last two digits of your roll number”.
2. To determine the height of the constructed BST.
3. Delete any element from the constructed BST.
4. To check if the constructed BST is Balanced or not.

**4. Experimental Procedure**

* 1. **Algorithm design**

class Node:

def init:

data, left, right

class BST:

def init:

root = None

def insert(self, data):

if root is None:

root = Node(data)

else:

insert(data)

def insert:

if data < current.data:

if left is None:

left = Node(data)

else:

insert(data)

else:

if right is None:

right = Node(data)

else:

insert(data)

def height:

if not node:

return -1

leftheight = height(node.left)

rightheight = height(node.right)

return 1 + max(leftheight, rightheight)

def delete:

root = delete(data)

def delete:

if data < node.data:

node.left = delete(data)

elif data > node.data:

node.right = delete(data)

else:

not node.right:

return node.left

temp = minvaluenode(node.right)

node.data = temp.data

node.right = delete(temp.data)

return node

def minvaluenode:

current = node

while current.left:

current = current.left

return current

def is\_balanced:

if not node:

return True

left\_height = height(node.left)

right\_height = height(node.right)

if abs(left\_height - right\_height) > 1:

return False

return isbalanced(node.left) and isbalanced(node.right)

def inorder:

if node:

inorder(node.left)

print(node.data, end=" ")

self.in\_order(node.right)

bst = BST()

values = [12, 35, 14, 97, 36, 65, 89]

for value in values:

bst.insert(value)

roll\_number\_last\_two\_digits = 12

bst.insert(roll\_number\_last\_two\_digits)

tree\_height = bst.height(bst.root)

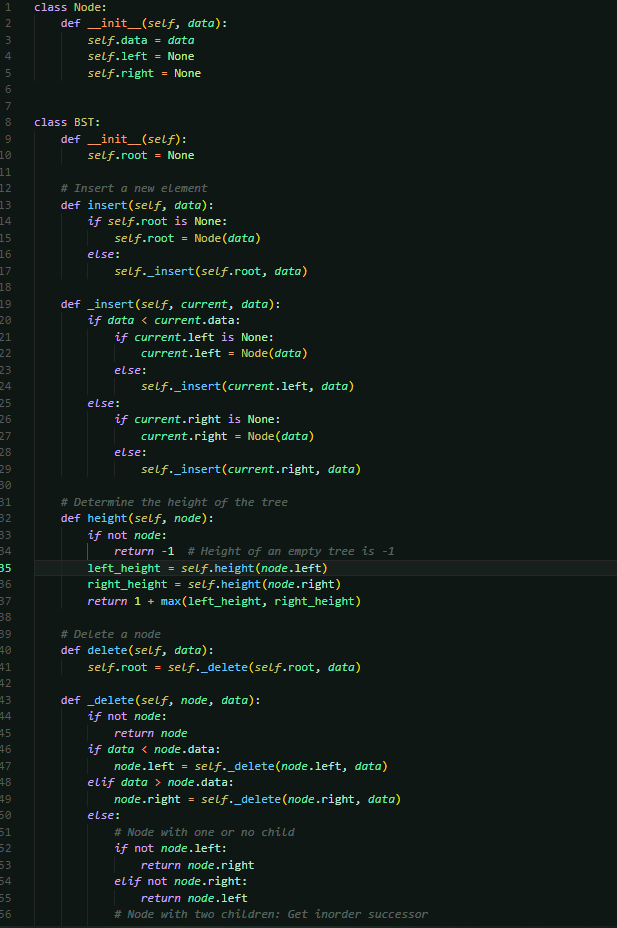
print(f"Height of the BST: {tree\_height}")

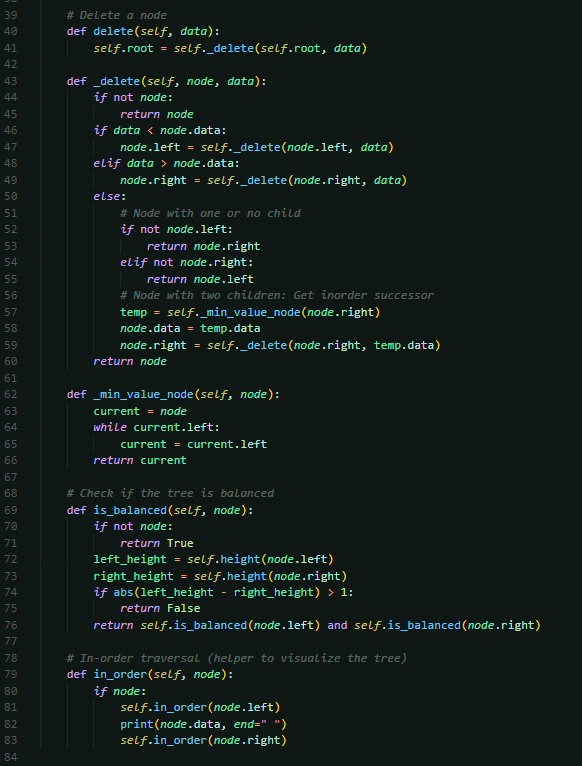
bst.delete(14)

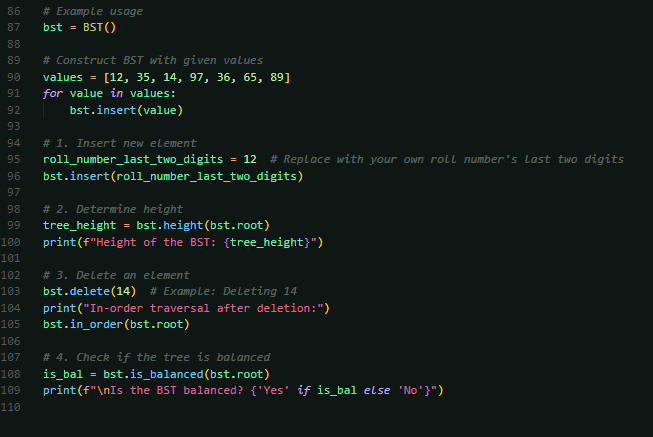
bst.in\_order(bst.root)

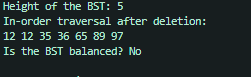
is\_bal = bst.is\_balanced(bst.root)

* 1. **Program**

****

****

****

* 1. **Presentation of the results**
  2. **Analysis and discussions**

**Insert a New Element**:

**Operation:** Adds a new node to the BST while maintaining the BST property (left child < parent node < right child).

**Time Complexity**: O(log n)

**Determine the Height of BST**:

**Operation:** Calculates the height of the BST, which is the number of edges on the longest path from the root to a leaf**.**

**Time Complexity**: O(n)

**Delete an Element:**

**Operation:** Removes a node from the BST while maintaining the BST property. Depending on the node to be deleted (leaf, one child, two children), different cases need to be handled.

**Time Complexity:** O(n^2

**In-order Traversal**:

**Operation:** Visits nodes in the order: left subtree, root, right subtree.

**Time Complexity:** O(n)

|  |  |
| --- | --- |
| **FUNCTION** | **Time Complexity**: |
| **Insert a New Element**: | O(log n) |
| **Determine the Height of BST**: | O(n) |
| **Delete an Element:** | O(n^2 |
| **In-order Traversal**: | O(n) |

# Experiment 7

**Title of the Laboratory Exercise**: Heap

**1. Aim:**

To understand and implement the basic operations in Heap using python.

**2. Objective:**

To execute the below operations in a Heap: <https://medium.com/techie-delight/heap-practice-problems-and-interview-questions-b678ff3b694c>

**3. Exercise:**

10, 12, 14, 16, 18 and 20 and perform the following operation on the constructed Heap Tree.

1. Insert a new element whose value is equivalent to the sum of the digits of your roll number.
2. Find the maximum element in the constructed Max Heap.
3. Delete the root element (maximum element) two times from the Max Heap.

**4. Experimental Procedure**

* 1. **Algorithm design**

import heapq

class MaxHeap:

init:

self.heap = []

def buildheap:

heap = [-el for el in elements]

heapify(heap)

def insert:

heappush(self.heap, -value)

def findmax:

return -self.heap[0] if self.heap else None

def delete\_max(self):

return -heapq.heappop(self.heap) if self.heap else None

def print\_heap:

print([-el for el in self.heap])

\_\_main\_\_:

elements = [10, 12, 14, 16, 18, 20]

heap = MaxHeap()

heap.buildheap(elements)

print("Initial Max Heap:")

heap.print\_heap()

roll\_number = 412012

sum\_of\_digits = sum(digit)

heap.insert(sum\_of\_digits)

heap.print\_heap()

max\_element = heap.find\_max()

print(max\_element)

deleted\_1 = heap.delete\_max()

print(deleted\_1)

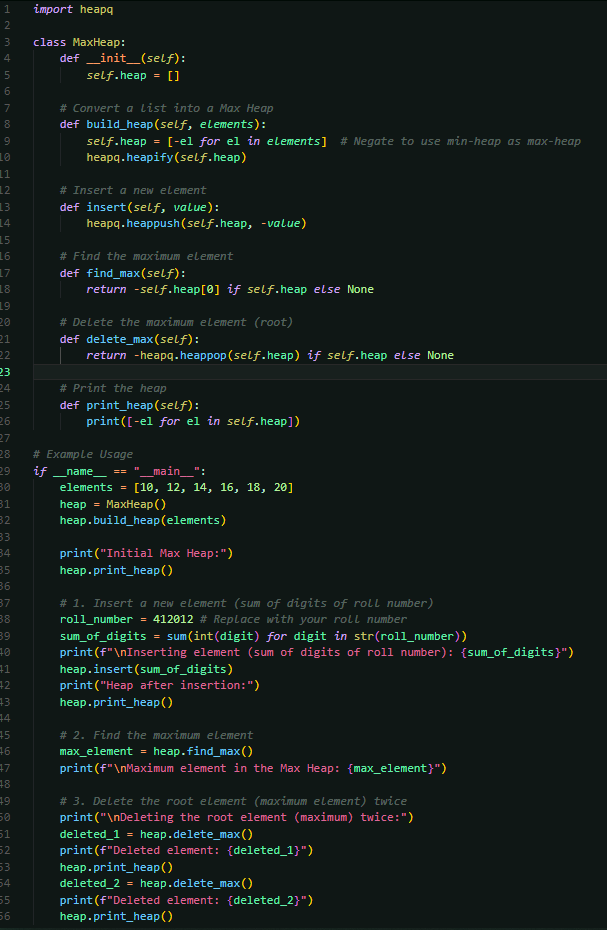
heap.print\_heap()

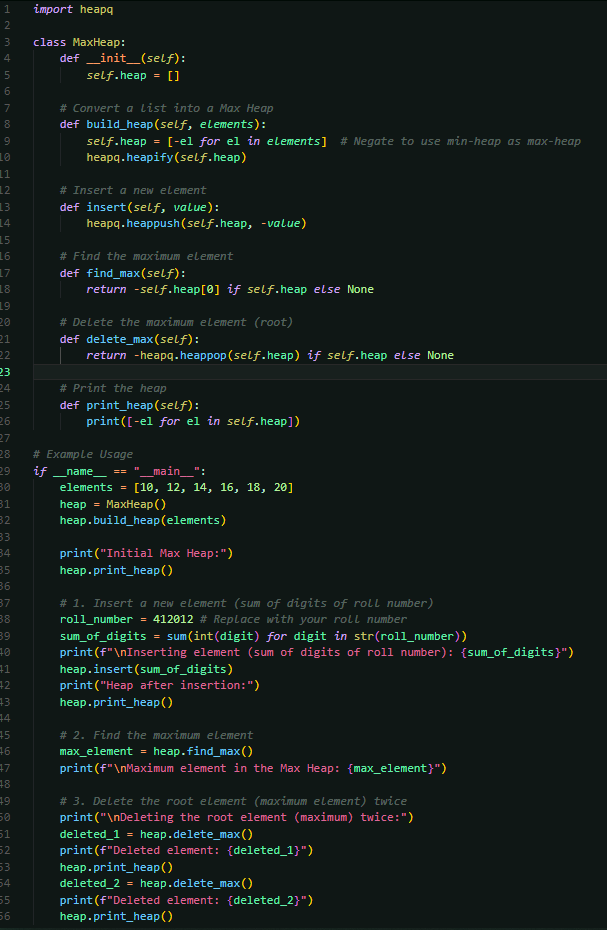
deleted\_2 = heap.delete\_max()

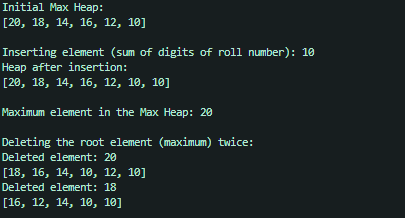
print(deleted\_2)

heap.print\_heap()

* 1. **Program**

****

****

****

* 1. **Presentation of the results**
  2. **Analysis and discussions**

**Heap** **Construction**:

**Operation**: Converts an unsorted list of elements into a Max Heap. This is done by negating the elements to leverage Python's heapq (which is a Min Heap) to simulate a Max Heap.

**Time** **Complexity**: O(n)

**Insertion**:

**Operation**: Inserts a new element into the Max Heap while maintaining the heap property. The element is added, and then the heap is restructured (up-heap) to ensure the max-heap property is upheld

**Time** **Complexity**: O(log n)

**Find** **Maximum**:

**Operation**: Retrieves the maximum element in the Max Heap, which is always at the root (index 0).

**Time** **Complexity**: O(1)

**Delete** **Maximum**:

**Operation**: Deletes the maximum element (root) from the Max Heap. The last element is moved to the root, and then the heap is restructured (down-heap) to maintain the max-heap property.

**Time Complexity:** O(log n)

# Experiment 8

**Title of the Laboratory Exercise**: AVL Tree

**1. Aim:**

To understand and implement the basic operations in AVL using python.

**2. Objective:**

To execute the below operations in an AVL Tree:

1. Left rotation
2. Right rotation
3. Left-Right rotation
4. Right-Left rotation

**3. Exercise:**

Implement a Python program that constructs an AVL tree having the following elements: Z, I, J, F, A, E, C, P, B, D, H, N. Consider the order of the elements in ascending order. Explain the rotations diagrammatically.

**4. Experimental Procedure**

* 1. **Algorithm design**

class Node:

def init:

self.data, left, right, height = 1

class AVLTree:

def init:

root = None

def height:

return node.height if node else 0

def getbalance:

return height(node.left) - height(node.right)

def rightrotate:

y = z.left

T3 = y.right

y.right = z

z.left = T3

z.height = 1 + max(height(z.left), height(z.right))

y.height = 1 + max(height(y.left), height(y.right))

return y

def leftrotate:

y = z.right

T2 = y.left

y.left = z

z.right = T2

z.height = 1 + max(height(z.left), height(z.right))

y.height = 1 + max(height(y.left), height(y.right))

return y

def insert:

if data < root.data:

left = insert(left, data)

elif data > root.data:

right = insert(right, data)

else:

return root

root.height = 1 + max(height(root.left), height(root.right))

balance = getbalance(root)

if balance > 1 and data < left.data:

return rightrotate(root)

if balance < -1 and data > root.right.data:

return eftrotate(root)

if balance > 1 and data > root.left.data:

root.left = leftrotate(root.left)

return rightrotate(root)

if balance < -1 and data < root.right.data:

root.right = rightrotate(root.right)

return leftrotate(root)

return root

def inorder:

if root:

inorder(root.left)

print(root.data)

inorder(root.right)

\_\_main\_\_:

elements = ["Z", "I", "J", "F", "A", "E", "C", "P", "B", "D", "H", "N"]

elements.sort()

avl = AVLTree()

root = None

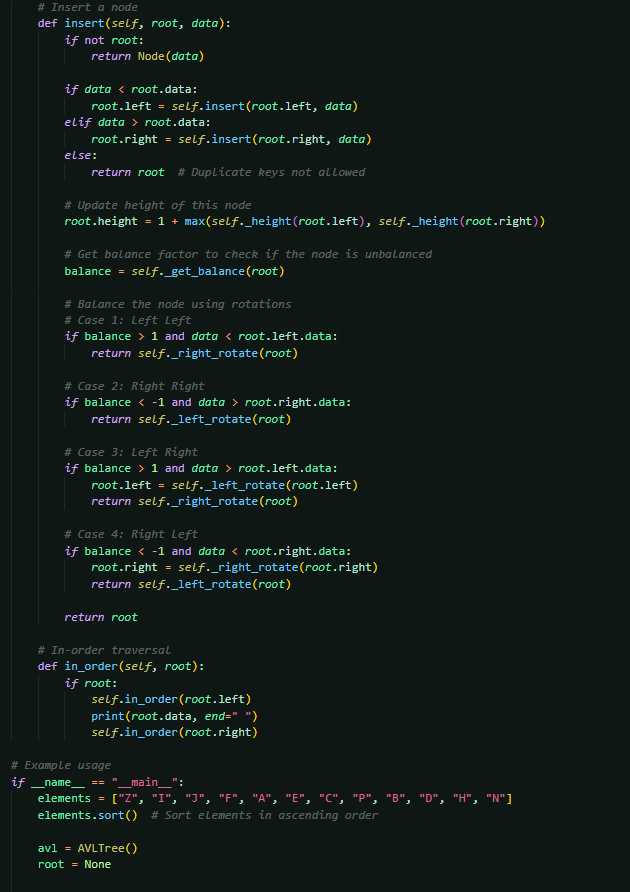
for element in elements:

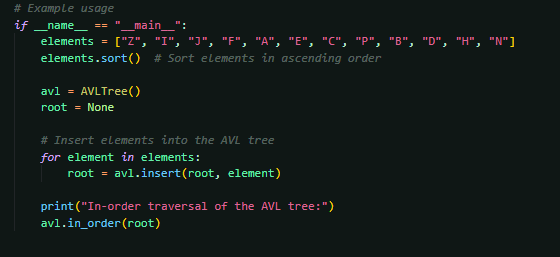
root = avl.insert(root, element)

avl.in\_order(root)

* 1. **Program**

****

****

****

****

* 1. **Presentation of the results**
  2. **Analysis and discussions**

**Insertion**:

**Operation:** Inserts a new node into the AVL tree. After the insertion, the tree checks for balance and performs rotations (if necessary) to maintain the AVL property (balance factor between -1 and 1).

**Time** **Complexity**: O(logn)

**Height** **Calculation**:

**Operation:** Calculates the height of a node, which is the number of edges on the longest path from that node to a leaf node. The height is updated during insertions and rotations.

**Time** **Complexity**: O(1)

**In-order** **Traversal**:

**Operation:** Visits nodes in the order: left subtree, root, right subtree. This results in nodes being printed in ascending order for an AVL tree.

**Time** **Complexity**: O(n)

**Rotations**:

**Operation:** Rotations are used to maintain the balance of the AVL tree. There are four types of rotations: right rotation, left rotation, left-right rotation, and right-left rotation, depending on the imbalance type.

**Time** **Complexity**: O(1)

# Experiment 9

**Title of the Laboratory Exercise**: Quick Sort

**1. Aim:**

To implement Quick Sort Algorithm using Python

**2. Objective:**

1. To understand the concept of Quick Sort Algorithm
2. To learn how to implement Quick Sort Algorithm using Python
3. To analyze the time complexity of Quick Sort Algorithm

**3. Exercise:**

In this exercise, you will implement Quick Sort Algorithm using Python. Follow the steps below:

**Step 1:** Write a function called quick\_sort that takes an array of integers as input and returns a sorted array.

**Step 2:** Implement the Quick Sort Algorithm. The steps of the Quick Sort Algorithm are as follows:

i. Choose a pivot element from the array (can be the first or last element).

ii. Partition the array into two subarrays: one with elements less than or equal to the pivot, and one with elements greater than the pivot.

iii. Recursively sort the two subarrays.

**Step 3:** Test your implementation using a test case that includes a list of 10 unsorted integers.

**Step 4:** Analyze the time complexity of Quick Sort Algorithm.

**Step 5:** Submit your code along with a brief explanation of the Quick Sort Algorithm and its time complexity analysis.

Note: You can use the time module in Python to measure the time taken by your quick\_sort function to sort an array.

**4. Experimental Procedure**

* 1. **Algorithm design**

import time

def quick sort:

if len <= 1:

return arr

pivot = arr[-1]

if x <= pivot

left = [x in arr[:-1]]

if x > pivot

right = [x in arr[:-1]]

return quick sort(left) + [pivot] + quick sort(right)

main

unsorted array = [12, 7, 5, 9, 3, 11, 1, 4, 10, 8]

start time = time()

sorted array = quick sort(unsorted array)

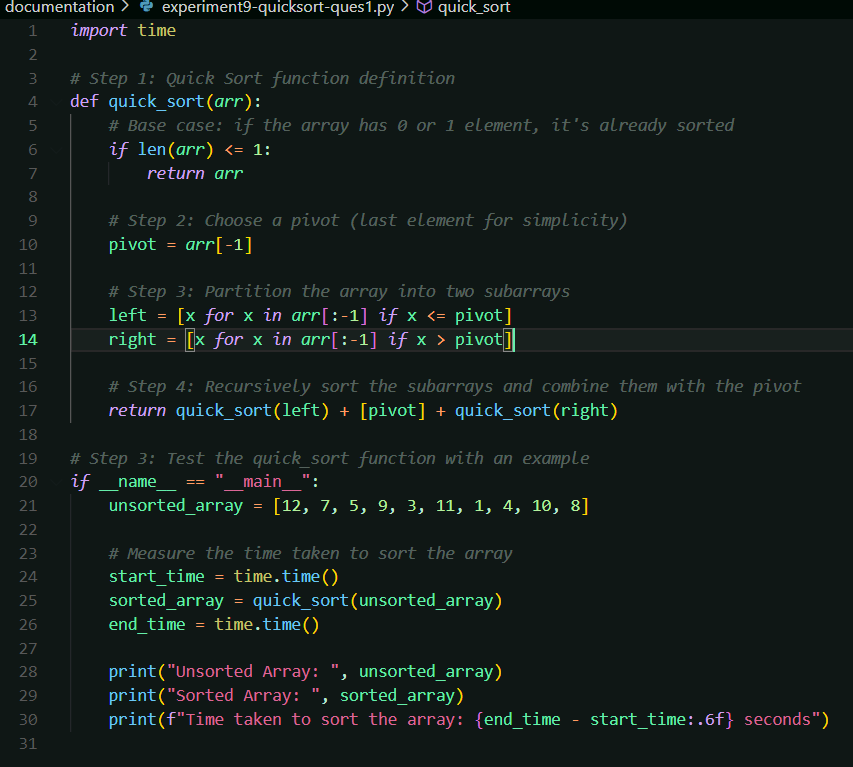
end time = time()

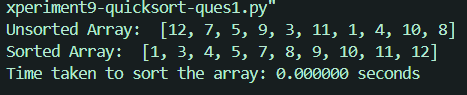
print(unsorted array)

print(sorted array)

time taken = end\_time - start\_time:.6f

print(time taken)

* 1. **Program**
  2. **Presentation of the results**

****

* 1. **Analysis and discussions**

**Quick Sort (Main Function):**

**Operation**: Quick sort recursively divides the array into smaller subarrays based on a pivot, sorts the subarrays, and then combines them.

**Time Complexity**: O(n log n)

**Partitioning:**

**Operation**: The array is partitioned into two subarrays: one containing elements less than or equal to the pivot and the other containing elements greater than the pivot.

**Time Complexity**: O(n)

**Recursive Calls:**

**Operation**: After partitioning, quick sort is recursively called on the left and right subarrays.

**Time Complexity**: O(log n)

**List Comprehensions (for partitioning):**

**Operation**: Creates two subarrays (left and right) based on whether the elements are less than or greater than the pivot.

**Time Complexity**: O(n) / O(n log n)

**Time Measurement:**

**Operation**: Measures the time taken to perform the sorting operation using the time module.

**Time Complexity**: O(1)