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

4.1 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))

4.2 Program

```python
import random

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

class FullBinaryTree:
    def __init__(self):
        self.root = None

    # Construct a full binary tree with random values
    def construct_full_binary_tree(self, n=10, min_value=1, max_value=100):
        """Constructs a full binary tree with n nodes."""
        if n <= 0:
            return
        values = [random.randint(min_value, max_value) for _ in range(n)]
        self.root = self._construct_tree(values, 0)

    def _construct_tree(self, values, index):
        """Helper function to recursively construct a full binary tree."""
        if index < len(values):
            node = Node(values[index])
            node.left = self._construct_tree(values, 2 * index + 1)
            node.right = self._construct_tree(values, 2 * index + 2)
            return node
        return None

    # Find the minimum value in the tree
    def find_min(self, node):
        if not node:
            return float('inf')
        return min(node.data, self.find_min(node.left), self.find_min(node.right))

    # Find the maximum value in the tree
    def find_max(self, node):
        if not node:
            return float('-inf')
        return max(node.data, self.find_max(node.left), self.find_max(node.right))

```

```python
class FullBinaryTree:
    def calculate_sum(self, node):
            return 0
        return node.data + self.calculate_sum(node.left) + self.calculate_sum(node.right)

    # Pre-order Traversal: Root -> Left -> Right
    def pre_order_traversal(self, node):
        stack = []
        current = node
        while stack or current:
            if current:
                print(current.data, end=" ")
                stack.append(current)
                current = current.left
            else:
                current = stack.pop()
                current = current.right

    # Post-order Traversal: Left -> Right -> Root
    def post_order_traversal(self, node):
        stack1 = []
        stack2 = []
        if node:
            stack1.append(node)
        while stack1:
            current = stack1.pop()
            stack2.append(current)
            if current.left:
                stack1.append(current.left)
            if current.right:
                stack1.append(current.right)
        while stack2:
            print(stack2.pop().data, end=" ")

    # In-order Traversal: Left -> Root -> Right
    def in_order_traversal(self, node):
        stack = []
        current = node
        while stack or current:
            if current:
                stack.append(current)
                current = current.left
            else:
                current = stack.pop()
                print(current.data, end=" ")
                current = current.right
```

```python
# Example Usage
if __name__ == "__main__":
    # Create a FullBinaryTree instance
    fbt = FullBinaryTree()

    # Construct the tree with random values
    fbt.construct_full_binary_tree()

    # Perform operations
    print("In-order Traversal:")
    fbt.in_order_traversal(fbt.root)
    print("\nPre-order Traversal:")
    fbt.pre_order_traversal(fbt.root)
    print("\nPost-order Traversal:")
    fbt.post_order_traversal(fbt.root)
    print("\nMinimum Value:", fbt.find_min(fbt.root))
    print("Maximum Value:", fbt.find_max(fbt.root))
    print("Sum of All Values:", fbt.calculate_sum(fbt.root))
```

4.3 Presentation of the results

```
In-order Traversal:
92 90 79 28 19 17 23 29 46 56
Pre-order Traversal:
23 28 90 92 79 17 19 46 29 56
Post-order Traversal:
92 79 90 19 17 28 29 56 46 23
Minimum Value: 17
Maximum Value: 92
Sum of All Values: 479
```

4.4 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

4.1 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)
```

4.2 Program

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


class BST:
    def __init__(self):
        self.root = None

    # Insert a new element
    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert(self.root, data)

    def _insert(self, current, data):
        if data < current.data:
            if current.left is None:
                current.left = Node(data)
            else:
                self._insert(current.left, data)
        else:
            if current.right is None:
                current.right = Node(data)
            else:
                self._insert(current.right, data)

    # Determine the height of the tree
    def height(self, node):
        if not node:
            return -1  # Height of an empty tree is -1
        left_height = self.height(node.left)
        right_height = self.height(node.right)
        return 1 + max(left_height, right_height)

    # Delete a node
    def delete(self, data):
        self.root = self._delete(self.root, data)

```

```python
    # Delete a node
    def delete(self, data):
        self.root = self._delete(self.root, data)

    def _delete(self, node, data):
        if not node:
            return node
        if data < node.data:
            node.left = self._delete(node.left, data)
        elif data > node.data:
            node.right = self._delete(node.right, data)
        else:
            # Node with one or no child
            if not node.left:
                return node.right
            elif not node.right:
                return node.left
            # Node with two children: Get inorder successor
            temp = self._min_value_node(node.right)
            node.data = temp.data
            node.right = self._delete(node.right, temp.data)
        return node

    def _min_value_node(self, node):
        current = node
        while current.left:
            current = current.left
        return current

    # Check if the tree is balanced
    def is_balanced(self, node):
        if not node:
            return True
        left_height = self.height(node.left)
        right_height = self.height(node.right)
        if abs(left_height - right_height) > 1:
            return False
        return self.is_balanced(node.left) and self.is_balanced(node.right)

    # In-order traversal (helper to visualize the tree)
    def in_order(self, node):
        if node:
            self.in_order(node.left)
            print(node.data, end=" ")
            self.in_order(node.right)
```

```python
# Example usage
bst = BST()

# Construct BST with given values
values = [12, 35, 14, 97, 36, 65, 89]
for value in values:
    bst.insert(value)

# 1. Insert new element
roll_number_last_two_digits = 12  # Replace with your own roll number's last two digits
bst.insert(roll_number_last_two_digits)

# 2. Determine height
tree_height = bst.height(bst.root)
print(f"Height of the BST: {tree_height}")

# 3. Delete an element
bst.delete(14)  # Example: Deleting 14
print("In-order traversal after deletion:")
bst.in_order(bst.root)

# 4. Check if the tree is balanced
is_bal = bst.is_balanced(bst.root)
print(f"\nIs the BST balanced? {'Yes' if is_bal else 'No'}")
```

4.3 Presentation of the results

```
Height of the BST: 5
In-order traversal after deletion:
12 12 35 36 65 89 97
Is the BST balanced? No
```

4.4 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**

4.1 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()
```

4.2 Program

```python
import heapq

class MaxHeap:
    def __init__(self):
        self.heap = []

    # Convert a list into a Max Heap
    def build_heap(self, elements):
        self.heap = [-el for el in elements]  # Negate to use min-hea
        heapq.heapify(self.heap)

    # Insert a new element
    def insert(self, value):
        heapq.heappush(self.heap, -value)

    # Find the maximum element
    def find_max(self):
        return -self.heap[0] if self.heap else None

```

```python

    # Delete the maximum element (root)
    def delete_max(self):
        return -heapq.heappop(self.heap) if self.heap else None

    # Print the heap
    def print_heap(self):
        print([-el for el in self.heap])

# Example Usage
if __name__ == "__main__":
    elements = [10, 12, 14, 16, 18, 20]
    heap = MaxHeap()
    heap.build_heap(elements)

    print("Initial Max Heap:")
    heap.print_heap()

    # 1. Insert a new element (sum of digits of roll number)
    roll_number = 412012 # Replace with your roll number
    sum_of_digits = sum(int(digit) for digit in str(roll_number))
    print(f"\nInserting element (sum of digits of roll number): {sum_of_digits}")
    heap.insert(sum_of_digits)
    print("Heap after insertion:")
    heap.print_heap()

    # 2. Find the maximum element
    max_element = heap.find_max()
    print(f"\nMaximum element in the Max Heap: {max_element}")

    # 3. Delete the root element (maximum element) twice
    print("\nDeleting the root element (maximum) twice:")
    deleted_1 = heap.delete_max()
    print(f"Deleted element: {deleted_1}")
    heap.print_heap()
    deleted_2 = heap.delete_max()
    print(f"Deleted element: {deleted_2}")
    heap.print_heap()
```

4.3 Presentation of the results

```
Initial Max Heap:
[20, 18, 14, 16, 12, 10]

Inserting element (sum of digits of roll number): 10
Heap after insertion:
[20, 18, 14, 16, 12, 10, 10]

Maximum element in the Max Heap: 20

Deleting the root element (maximum) twice:
Deleted element: 20
[18, 16, 14, 10, 12, 10]
Deleted element: 18
[16, 12, 14, 10, 10]
```

4.4 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**

4.1 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)

4.2 Program

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 1  # Height of the node


class AVLTree:
    def __init__(self):
        self.root = None

    # Get the height of a node
    def _height(self, node):
        return node.height if node else 0

    # Get the balance factor of a node
    def _get_balance(self, node):
        if not node:
            return 0
        return self._height(node.left) - self._height(node.right)

    # Right rotate
    def _right_rotate(self, z):
        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3

        # Update heights
        z.height = 1 + max(self._height(z.left), self._height(z.right))
        y.height = 1 + max(self._height(y.left), self._height(y.right))

        # Return new root
        return y

    # Left rotate
    def _left_rotate(self, z):
        y = z.right
        T2 = y.left

        # Perform rotation
        y.left = z
        z.right = T2

        # Update heights
        z.height = 1 + max(self._height(z.left), self._height(z.right))
        y.height = 1 + max(self._height(y.left), self._height(y.right))

        # Return new root
        return y

    # Insert a node
    def insert(self, root, data):
```

```python
    # Insert a node
    def insert(self, root, data):
        if not root:
            return Node(data)

        if data < root.data:
            root.left = self.insert(root.left, data)
        elif data > root.data:
            root.right = self.insert(root.right, data)
        else:
            return root  # Duplicate keys not allowed

        # Update height of this node
        root.height = 1 + max(self._height(root.left), self._height(root.right))

        # Get balance factor to check if the node is unbalanced
        balance = self._get_balance(root)

        # Balance the node using rotations
        # Case 1: Left Left
        if balance > 1 and data < root.left.data:
            return self._right_rotate(root)

        # Case 2: Right Right
        if balance < -1 and data > root.right.data:
            return self._left_rotate(root)

        # Case 3: Left Right
        if balance > 1 and data > root.left.data:
            root.left = self._left_rotate(root.left)
            return self._right_rotate(root)

        # Case 4: Right Left
        if balance < -1 and data < root.right.data:
            root.right = self._right_rotate(root.right)
            return self._left_rotate(root)

        return root

    # In-order traversal
    def in_order(self, root):
        if root:
            self.in_order(root.left)
            print(root.data, end=" ")
            self.in_order(root.right)

# Example usage
if __name__ == "__main__":
    elements = ["Z", "I", "J", "F", "A", "E", "C", "P", "B", "D", "H", "N"]
    elements.sort()  # Sort elements in ascending order

    avl = AVLTree()
    root = None
```

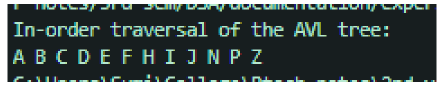
```python
# Example usage
if __name__ == "__main__":
    elements = ["Z", "I", "J", "F", "A", "E", "C", "P", "B", "D", "H", "N"]
    elements.sort()  # Sort elements in ascending order

    avl = AVLTree()
    root = None

    # Insert elements into the AVL tree
    for element in elements:
        root = avl.insert(root, element)

    print("In-order traversal of the AVL tree:")
    avl.in_order(root)
```

4.3 Presentation of the results


In-order traversal of the AVL tree:
A B C D E F H I J N P Z

4.4 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**

   4.1 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)

4.2 Program

documentation > experiment9-quicksort-ques1.py > quick_sort

```python
import time

# Step 1: Quick Sort function definition
def quick_sort(arr):
    # Base case: if the array has 0 or 1 element, it's already sorted
    if len(arr) <= 1:
        return arr

    # Step 2: Choose a pivot (last element for simplicity)
    pivot = arr[-1]

    # Step 3: Partition the array into two subarrays
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]

    # Step 4: Recursively sort the subarrays and combine them with the pivot
    return quick_sort(left) + [pivot] + quick_sort(right)

# Step 3: Test the quick_sort function with an example
if __name__ == "__main__":
    unsorted_array = [12, 7, 5, 9, 3, 11, 1, 4, 10, 8]

    # Measure the time taken to sort the array
    start_time = time.time()
    sorted_array = quick_sort(unsorted_array)
    end_time = time.time()

    print("Unsorted Array: ", unsorted_array)
    print("Sorted Array: ", sorted_array)
    print(f"Time taken to sort the array: {end_time - start_time:.6f} seconds")
```

4.3 Presentation of the results

```
xperiment9-quicksort-ques1.py"
Unsorted Array:  [12, 7, 5, 9, 3, 11, 1, 4, 10, 8]
Sorted Array:  [1, 3, 4, 5, 7, 8, 9, 10, 11, 12]
Time taken to sort the array: 0.000000 seconds
```

4.4 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)