**EXP 6**

# Step 2: Initialize and declare variables

class Node:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.val = key

# Step 3: Construct the Tree

def insert(root, key):

# If the tree is empty, return a new node

if root is None:

return Node(key)

# Otherwise, recur down the tree

if key < root.val:

root.left = insert(root.left, key)

else:

root.right = insert(root.right, key)

return root

# Step 4-8: Searching for the key in the binary search tree

def search(root, key):

# Step 5: Start with the root node

# Step 6: Compare the key with the data value of the root node

if root is None or root.val == key:

return root

# Step 7: If key is less than the data value of the root node, search in the left subtree

if key < root.val:

return search(root.left, key)

# Step 8: Otherwise, search in the right subtree

return search(root.right, key)

# Step 9: Terminate

# Example Usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Create the root node

root = None

# Insert data into the binary search tree

data = [20, 10, 30, 5, 15, 25, 35]

for value in data:

root = insert(root, value)

# Search for a key

key = 25

result = search(root, key)

if result:

print(f"Key {key} found in the binary search tree.")

else:

print(f"Key {key} not found in the binary search tree.")

**EXP 7**

# Step 2: Define the Node class

class Node:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.val = key

self.height = 1

# Step 1: Insert a new node in the AVL tree

def insert(root, key):

# Normal BST insertion

if root is None:

return Node(key)

elif key < root.val:

root.left = insert(root.left, key)

else:

root.right = insert(root.right, key)

# Step 2: Update the height of the current node

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

# Step 3: Get the balance factor of this node

balance = get\_balance(root)

# Step 4: Left Left Case

if balance > 1 and key < root.left.val:

return right\_rotate(root)

# Step 4: Left Right Case

if balance > 1 and key > root.left.val:

root.left = left\_rotate(root.left)

return right\_rotate(root)

# Step 5: Right Right Case

if balance < -1 and key > root.right.val:

return left\_rotate(root)

# Step 5: Right Left Case

if balance < -1 and key < root.right.val:

root.right = right\_rotate(root.right)

return left\_rotate(root)

return root

# Step 3: Get the height of the node

def get\_height(node):

if not node:

return 0

return node.height

# Step 3: Calculate the balance factor

def get\_balance(node):

if not node:

return 0

return get\_height(node.left) - get\_height(node.right)

# Step 4: Perform right rotation (for Left Left and Right Left cases)

def right\_rotate(z):

y = z.left

T3 = y.right

# Perform rotation

y.right = z

z.left = T3

# Update heights

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

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

# Return the new root

return y

# Step 5: Perform left rotation (for Right Right and Left Right cases)

def left\_rotate(z):

y = z.right

T2 = y.left

# Perform rotation

y.left = z

z.right = T2

# Update heights

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

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

# Return the new root

return y

# Utility function to print the tree (inorder traversal)

def inorder\_traversal(root):

if root is not None:

inorder\_traversal(root.left)

print(root.val, end=" ")

inorder\_traversal(root.right)

# Example Usage:

if \_\_name\_\_ == "\_\_main\_\_":

root = None

data = [10, 20, 30, 40, 50, 25]

# Inserting data into the AVL Tree

for value in data:

root = insert(root, value)

# Printing the inorder traversal of the AVL Tree

print("Inorder traversal of the constructed AVL tree is:")

inorder\_traversal(root)

**EXP 8**

from collections import deque

# Function to perform Breadth-First Search (BFS)

def bfs(graph, start\_node):

visited = set() # To track visited nodes

queue = deque([start\_node]) # Step 1: Push the root node in the Queue

while queue: # Step 2: Loop until the queue is empty

node = queue.popleft() # Step 3: Remove the node from the Queue

if node not in visited:

print(node, end=" ") # Process the node (here we print it)

visited.add(node) # Mark the node as visited

# Step 4: Add all unvisited adjacent nodes to the queue

for neighbor in graph[node]:

if neighbor not in visited:

queue.append(neighbor)

# Example graph represented as an adjacency list

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

print("BFS traversal starting from node A:")

bfs(graph, 'A')

**DFS**

# Function to perform Depth-First Search (DFS)

def dfs(graph, start\_node):

visited = set() # To track visited nodes

stack = [start\_node] # Step 1: Push the root node in the Stack

while stack: # Step 2: Loop until stack is empty

node = stack[-1] # Step 3: Peek the node of the stack

if node not in visited:

print(node, end=" ") # Process the node (here we print it)

visited.add(node) # Mark the node as visited

# Step 4: Find unvisited child nodes and push them to the stack

unvisited\_neighbors = [neighbor for neighbor in graph[node] if neighbor not in visited]

if unvisited\_neighbors:

stack.append(unvisited\_neighbors[0]) # Push the first unvisited child node onto the stack

else:

stack.pop() # Step 5: Pop the node from the stack if it has no unvisited neighbors

# Example graph represented as an adjacency list

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

print("DFS traversal starting from node A:")

dfs(graph, 'A')

EXP 9.A

# Function to perform Bubble Sort

def bubble\_sort(arr):

n = len(arr)

# Traverse through all elements in the array

for i in range(n):

# Last i elements are already sorted, so we can skip them

for j in range(0, n-i-1):

# Step 1: Compare the current element with the next element

if arr[j] > arr[j+1]: # Step 2: If the current element is greater, swap them

arr[j], arr[j+1] = arr[j+1], arr[j]

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

arr = [64, 34, 25, 12, 22, 11, 90]

print("Original array:", arr)

bubble\_sort(arr)

print("Sorted array:", arr)

**9.B**

# Function to perform Selection Sort

def selection\_sort(arr):

n = len(arr)

# Traverse through all array elements

for i in range(n):

# Step 1: Set MIN to the current index

min\_index = i

# Step 2: Search the minimum element in the remaining unsorted part of the array

for j in range(i+1, n):

if arr[j] < arr[min\_index]:

min\_index = j

# Step 3: Swap the found minimum element with the element at index MIN

arr[i], arr[min\_index] = arr[min\_index], arr[i]

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

arr = [64, 25, 12, 22, 11]

print("Original array:", arr)

selection\_sort(arr)

print("Sorted array:", arr)

**9.C**

# Function to perform Insertion Sort

def insertion\_sort(arr):

# Traverse from the second element to the end of the array

for i in range(1, len(arr)):

key = arr[i] # Step 2: Pick the first element from the unsorted part

j = i - 1

# Step 3 and Step 4: Compare with all elements in the sorted sub-list and shift elements

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j] # Shift elements greater than the key

j -= 1

# Step 5: Insert the value in its correct position

arr[j + 1] = key

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

arr = [12, 11, 13, 5, 6]

print("Original array:", arr)

insertion\_sort(arr)

print("Sorted array:", arr)