1. **# A Huffman Tree Node**

**import** heapq

**class** node:

**def** \_\_init\_\_(self, freq, symbol, left**=**None, right**=**None):

        # frequency of symbol

        self.freq **=** freq

        # symbol name (character)

        self.symbol **=** symbol

        # node left of current node

        self.left **=** left

        # node right of current node

        self.right **=** right

        # tree direction (0/1)

        self.huff **=** ''

**def** \_\_lt\_\_(self, nxt):

**return** self.freq < nxt.freq

# utility function to print huffman

# codes for all symbols in the newly

# created Huffman tree

**def** printNodes(node, val**=**''):

    # huffman code for current node

    newVal **=** val **+** str(node.huff)

    # if node is not an edge node

    # then traverse inside it

**if**(node.left):

        printNodes(node.left, newVal)

**if**(node.right):

        printNodes(node.right, newVal)

        # if node is edge node then

        # display its huffman code

**if**(**not** node.left **and** **not** node.right):

**print**(f"{node.symbol} -> {newVal}")

# characters for huffman tree

chars **=** ['a', 'b', 'c', 'd', 'e', 'f']

# frequency of characters

freq **=** [5, 9, 12, 13, 16, 45]

# list containing unused nodes

nodes **=** []

# converting characters and frequencies

# into huffman tree nodes

**for** x **in** range(len(chars)):

    heapq.heappush(nodes, node(freq[x], chars[x]))

**while** len(nodes) > 1:

    # sort all the nodes in ascending order

    # based on their frequency

    left **=** heapq.heappop(nodes)

    right **=** heapq.heappop(nodes)

    # assign directional value to these nodes

    left.huff **=** 0

    right.huff **=** 1

    # combine the 2 smallest nodes to create

    # new node as their parent

    newNode **=** node(left.freq**+**right.freq, left.symbol**+**right.symbol, left, right)

    heapq.heappush(nodes, newNode)

# Huffman Tree is ready!

printNodes(nodes[0])

**output**

f -> 0

c -> 100

d -> 101

a -> 1100

b -> 1101

e -> 111

**Experiment No.-2**: Write a program that uses functions to perform the following operations on Binary Search Tree: i) Creation ii) Insertion iii) Deletion iv) Traversal (Inorder, Preorder,

Postorder)

**Inorder**

# Class describing a node of tree

class Node:

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

self.left = None

self.right = None

self.data = v

# Inorder Traversal

def printInorder(root):

if root:

# Traverse left subtree

printInorder(root.left)

# Visit node

print(root.data,end=" ")

# Traverse right subtree

printInorder(root.right)

# Driver code

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

# Build the tree

root = Node(100)

root.left = Node(20)

root.right = Node(200)

root.left.left = Node(10)

root.left.right = Node(30)

root.right.left = Node(150)

root.right.right = Node(300)

# Function call

print("Inorder Traversal:",end=" ")

printInorder(root)

# **Preorder** Traversal

def printPreOrder(node):

if node is None:

return

# Visit Node

print(node.data, end = " ")

# Traverse left subtree

printPreOrder(node.left)

# Traverse right subtree

printPreOrder(node.right)

# Driver code

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

# Build the tree

root = Node(100)

root.left = Node(20)

root.right = Node(200)

root.left.left = Node(10)

root.left.right = Node(30)

root.right.left = Node(150)

root.right.right = Node(300)

# Function call

print("Preorder Traversal: ", end = "")

printPreOrder(root)

**#PostOrder** Traversal

def printPostOrder(node):

if node is None:

return

# Traverse left subtree

printPostOrder(node.left)

# Traverse right subtree

printPostOrder(node.right)

# Visit Node

print(node.data, end = " ")

# Driver code

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

# Build the tree

root = Node(100)

root.left = Node(20)

root.right = Node(200)

root.left.left = Node(10)

root.left.right = Node(30)

root.right.left = Node(150)

root.right.right = Node(300)

# Function call

print("Postorder Traversal: ", end = "")

printPostOrder(root)

# insert operation in binary search tree

class Node:

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

self.left = None

self.right = None

self.val = key

**# A utility function to insert**

**# a new node with the given key**

def insert(root, key):

if root is None:

return Node(key)

if root.val == key:

return root

if root.val < key:

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

else:

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

return root

# A utility function to do inorder tree traversal

def inorder(root):

if root:

inorder(root.left)

print(root.val, end=" ")

inorder(root.right)

# Creating the following BST

# 50

# / \

# 30 70

# / \ / \

# 20 40 60 80

r = Node(50)

r = insert(r, 30)

r = insert(r, 20)

r = insert(r, 40)

r = insert(r, 70)

r = insert(r, 60)

r = insert(r, 80)

# Print inorder traversal of the BST

inorder(r)

**#Searching**

class Node:

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

self.key = key

self.left = None

self.right = None

# function to search a key in a BST

def search(root, key):

# Base Cases: root is null or key

# is present at root

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

return root

# Key is greater than root's key

if root.key < key:

return search(root.right, key)

# Key is smaller than root's key

return search(root.left, key)

# Creating a hard coded tree for keeping

# the length of the code small. We need

# to make sure that BST properties are

# maintained if we try some other cases.

root = Node(50)

root.left = Node(30)

root.right = Node(70)

root.left.left = Node(20)

root.left.right = Node(40)

root.right.left = Node(60)

root.right.right = Node(80)

# Searching for keys in the BST

print("Found" if search(root, 19) else "Not Found")

print("Found" if search(root, 80) else "Not Found")

**#Deletion**

class Node:

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

self.key = key

self.left = None

self.right = None

# Note that it is not a generic inorder successor

# function. It mainly works when the right child

# is not empty, which is the case we need in BST

# delete.

def get\_successor(curr):

curr = curr.right

while curr is not None and curr.left is not None:

curr = curr.left

return curr

# This function deletes a given key x from the

# given BST and returns the modified root of the

# BST (if it is modified).

def del\_node(root, x):

# Base case

if root is None:

return root

# If key to be searched is in a subtree

if root.key > x:

root.left = del\_node(root.left, x)

elif root.key < x:

root.right = del\_node(root.right, x)

else:

# If root matches with the given key

# Cases when root has 0 children or

# only right child

if root.left is None:

return root.right

# When root has only left child

if root.right is None:

return root.left

# When both children are present

succ = get\_successor(root)

root.key = succ.key

root.right = del\_node(root.right, succ.key)

return root

# Utility function to do inorder traversal

def inorder(root):

if root is not None:

inorder(root.left)

print(root.key, end=" ")

inorder(root.right)

# Driver code

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

root = Node(10)

root.left = Node(5)

root.right = Node(15)

root.right.left = Node(12)

root.right.right = Node(18)

x = 15

root = del\_node(root, x)

inorder(root)

print()

**Experiment No 3**: **Write a program that uses Stack operations to perform the following:**

**a. Converting infix expression into postfix expression**

**b. Evaluating the postfix expression.**

**# infix to postfix**

def prec(c):

if c == '^':

return 3

elif c == '/' or c == '\*':

return 2

elif c == '+' or c == '-':

return 1

else:

return -1

# Function to perform infix to postfix conversion

def infixToPostfix(s):

st = []

result = ""

for i in range(len(s)):

c = s[i]

# If the scanned character is

# an operand, add it to the output string.

if (c >= 'a' and c <= 'z') or (c >= 'A' and c <= 'Z') or (c >= '0' and c <= '9'):

result += c

# If the scanned character is an

# ‘(‘, push it to the stack.

elif c == '(':

st.append('(')

# If the scanned character is an ‘)’,

# pop and add to the output string from the stack

# until an ‘(‘ is encountered.

elif c == ')':

while st[-1] != '(':

result += st.pop()

st.pop()

# If an operator is scanned

else:

while st and (prec(c) < prec(st[-1]) or prec(c) == prec(st[-1])):

result += st.pop()

st.append(c)

# Pop all the remaining elements from the stack

while st:

result += st.pop()

print(result)

exp = "a+b\*(c^d-e)^(f+g\*h)-i"

infixToPostfix(exp)

**# Python program to evaluate value of a postfix expression**

# Class to convert the expression

**class** Evaluate:

    # Constructor to initialize the class variables

**def** \_\_init\_\_(self, capacity):

        self.top **=** **-**1

        self.capacity **=** capacity

        # This array is used a stack

        self.array **=** []

    # Check if the stack is empty

**def** isEmpty(self):

**return** True **if** self.top **==** **-**1 **else** False

    # Return the value of the top of the stack

**def** peek(self):

**return** self.array[**-**1]

    # Pop the element from the stack

**def** pop(self):

**if** **not** self.isEmpty():

            self.top **-=** 1

**return** self.array.pop()

**else**:

**return** "$"

    # Push the element to the stack

**def** push(self, op):

        self.top **+=** 1

        self.array.append(op)

    # The main function that converts given infix expression

    # to postfix expression

**def** evaluatePostfix(self, exp):

        # Iterate over the expression for conversion

**for** i **in** exp:

            # If the scanned character is an operand

            # (number here) push it to the stack

**if** i.isdigit():

                self.push(i)

            # If the scanned character is an operator,

            # pop two elements from stack and apply it.

**else**:

                val1 **=** self.pop()

                val2 **=** self.pop()

                self.push(str(eval(val2 **+** i **+** val1)))

**return** int(self.pop())

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    exp **=** "231\*+9-"

    obj **=** Evaluate(len(exp))

    # Function call

**print**("postfix evaluation: %d" **%** (obj.evaluatePostfix(exp)))

**Experiment 4: Write a program that uses functions to perform the insert and delete operations on an AVL Tree.**

**b. Write a program that uses functions to perform the insert and delete operations on a B-Tree.**

**#AVL Tree**

class Node:

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

self.key = key

self.left = None

self.right = None

self.height = 1

def height(N):

if N is None:

return 0

return N.height

def right\_rotate(y):

x = y.left

T2 = x.right

# Perform rotation

x.right = y

y.left = T2

# Update heights

y.height = max(height(y.left),

height(y.right)) + 1

x.height = max(height(x.left),

height(x.right)) + 1

# Return new root

return x

def left\_rotate(x):

y = x.right

T2 = y.left

# Perform rotation

y.left = x

x.right = T2

# Update heights

x.height = max(height(x.left),

height(x.right)) + 1

y.height = max(height(y.left),

height(y.right)) + 1

# Return new root

return y

def get\_balance(N):

if N is None:

return 0

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

def insert(node, key):

# 1. Perform the normal BST insertion

if node is None:

return Node(key)

if key < node.key:

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

elif key > node.key:

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

else: # Duplicate keys not allowed

return node

# 2. Update height of this ancestor node

node.height = max(height(node.left),

height(node.right)) + 1

# 3. Get the balance factor of this node

# to check whether this node became

# unbalanced

balance = get\_balance(node)

# If this node becomes unbalanced, then

# there are 4 cases

# Left Left Case

if balance > 1 and key < node.left.key:

return right\_rotate(node)

# Right Right Case

if balance < -1 and key > node.right.key:

return left\_rotate(node)

# Left Right Case

if balance > 1 and key > node.left.key:

node.left = left\_rotate(node.left)

return right\_rotate(node)

# Right Left Case

if balance < -1 and key < node.right.key:

node.right = right\_rotate(node.right)

return left\_rotate(node)

return node

def min\_value\_node(node):

current = node

# loop down to find the leftmost leaf

while current.left is not None:

current = current.left

return current

def delete\_node(root, key):

# STEP 1: PERFORM STANDARD BST DELETE

if root is None:

return root

# If the key to be deleted is smaller

# than the root's key, then it lies in

# left subtree

if key < root.key:

root.left = delete\_node(root.left, key)

# If the key to be deleted is greater

# than the root's key, then it lies in

# right subtree

elif key > root.key:

root.right = delete\_node(root.right, key)

# if key is same as root's key, then

# this is the node to be deleted

else:

# node with only one child or no child

if root.left is None or root.right is None:

temp = root.left if root.left else root.right

# No child case

if temp is None:

root = None

else: # One child case

root = temp

else:

# node with two children: Get the

# inorder successor (smallest in

# the right subtree)

temp = min\_value\_node(root.right)

# Copy the inorder successor's

# data to this node

root.key = temp.key

# Delete the inorder successor

root.right = delete\_node(root.right, temp.key)

# If the tree had only one node then return

if root is None:

return root

# STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

root.height = max(height(root.left),

height(root.right)) + 1

# STEP 3: GET THE BALANCE FACTOR OF THIS

# NODE (to check whether this node

# became unbalanced)

balance = get\_balance(root)

# If this node becomes unbalanced, then

# there are 4 cases

# Left Left Case

if balance > 1 and get\_balance(root.left) >= 0:

return right\_rotate(root)

# Left Right Case

if balance > 1 and get\_balance(root.left) < 0:

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

return right\_rotate(root)

# Right Right Case

if balance < -1 and get\_balance(root.right) <= 0:

return left\_rotate(root)

# Right Left Case

if balance < -1 and get\_balance(root.right) > 0:

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

return left\_rotate(root)

return root

def pre\_order(root):

if root is not None:

print("{0} ".format(root.key), end="")

pre\_order(root.left)

pre\_order(root.right)

# Driver Code

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

root = None

# Constructing tree given in the

# above figure

root = insert(root, 9)

root = insert(root, 5)

root = insert(root, 10)

root = insert(root, 0)

root = insert(root, 6)

root = insert(root, 11)

root = insert(root, -1)

root = insert(root, 1)

root = insert(root, 2)

print("Preorder traversal of the "

"constructed AVL tree is")

pre\_order(root)

root = delete\_node(root, 10)

print("\nPreorder traversal after"

" deletion of 10")

pre\_order(root)

**# Insertion in AVL**

class Node:

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

self.key = key

self.left = None

self.right = None

self.height = 1

# A utility function to get the

# height of the tree

def height(node):

if not node:

return 0

return node.height

# A utility function to right rotate

# subtree rooted with y

def right\_rotate(y):

x = y.left

T2 = x.right

# Perform rotation

x.right = y

y.left = T2

# Update heights

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

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

# Return new root

return x

# A utility function to left rotate

# subtree rooted with x

def left\_rotate(x):

y = x.right

T2 = y.left

# Perform rotation

y.left = x

x.right = T2

# Update heights

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

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

# Return new root

return y

# Get balance factor of node N

def get\_balance(node):

if not node:

return 0

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

# Recursive function to insert a key in

# the subtree rooted with node

def insert(node, key):

# Perform the normal BST insertion

if not node:

return Node(key)

if key < node.key:

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

elif key > node.key:

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

else:

# Equal keys are not allowed in BST

return node

# Update height of this ancestor node

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

# Get the balance factor of this ancestor node

balance = get\_balance(node)

# If this node becomes unbalanced,

# then there are 4 cases

# Left Left Case

if balance > 1 and key < node.left.key:

return right\_rotate(node)

# Right Right Case

if balance < -1 and key > node.right.key:

return left\_rotate(node)

# Left Right Case

if balance > 1 and key > node.left.key:

node.left = left\_rotate(node.left)

return right\_rotate(node)

# Right Left Case

if balance < -1 and key < node.right.key:

node.right = right\_rotate(node.right)

return left\_rotate(node)

# Return the (unchanged) node pointer

return node

# A utility function to print preorder

# traversal of the tree

def pre\_order(root):

if root:

print(root.key, end=" ")

pre\_order(root.left)

pre\_order(root.right)

# Driver code

root = None

# Constructing tree given in the above figure

root = insert(root, 10)

root = insert(root, 20)

root = insert(root, 30)

root = insert(root, 40)

root = insert(root, 50)

root = insert(root, 25)

# The constructed AVL Tree would be

# 30

# / \

# 20 40

# / \ \

# 10 25 50

print("Preorder traversal :")

pre\_order(root)