PRACTICAL 1

AIM : Write a program for implementing a MINSTACK which should support operations like push, pop, overflow, underflow, display.

SOURCE CODE:

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.minStack = []
    def push(self, val):
            self.stack.append(val)
            val = min(val, self.minStack[-1] if self.minStack else val)
            self.minStack.append(val)

    def pop(self):
        self.stack.pop()
        self.minStack.pop()
    def getMin(self) -> int:
        return self.minStack[-1]

    def top(self) -> int:
        return self.stack[-1]
min_stack = MinStack()
min_stack.push(-3)
min_stack.push(5)
min_stack.push(2)
min_stack.push(1)
print("Top element: ", min_stack.top())
print("Minimum element: ", min_stack.getMin())
min_stack.pop()
print("After pop, minimum element: " , min_stack.getMin())
```

<u>OUTPUT:</u>

```
Top element:  1
Minimum element:  -3
After pop, minimum element:  -3


...Program finished with exit code 0
Press ENTER to exit console.
```

<u>Result :</u> The above code is successfully executed in Lab.

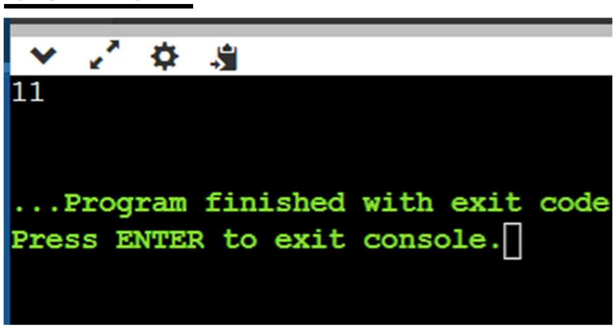
PRACTICAL 2

AIM: Write a program to deal with real-world situations where Stack data structure is widely used.

SOURCE CODE:

```
s = "23*5+"
stack = []
for i in s:
    if(i.isdigit()):
        stack.append(int (i))
    else:
        a = stack.pop()
        b = stack.pop()

        if i == '+':
            stack.append(b+a)
        elif i == '-':
            stack.append(b-a)
        elif i == '*':
            stack.append(b*a)
        elif i == '/':
            stack.append(b/a)

print(stack.pop())
```

OUTPUT:



Result : The above code is successfully executed in Lab.

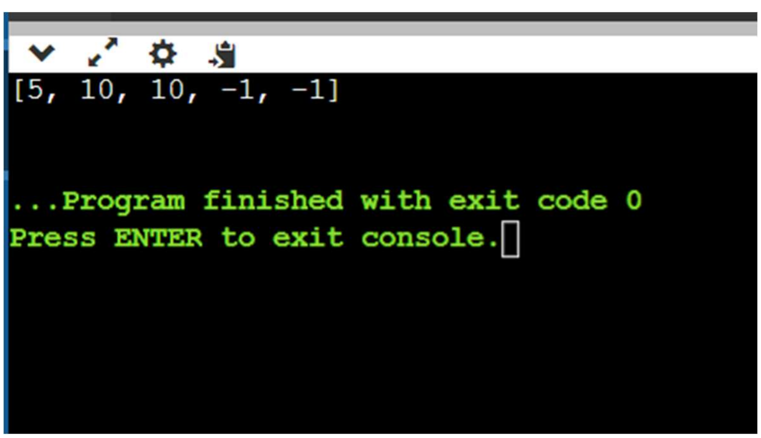
PRACTICAL 3

AIM: Write a program for finding NGE NEXT GREATER ELEMENT from an array.

SOURCE CODE:

```
a = [4,5,2,10,8]
n = len(a)
stack = []
res = [0]*n
for i in range(n-1, -1, -1):
    while stack and stack[-1] <= a[i]:
        stack.pop()
    if not stack:
        res[i] = -1
    else:
        res[i] = stack[-1]
    stack.append(a[i])
print(res)
```

OUTPUT:



Result : The above code is successfully executed in Lab.

PRACTICAL 4

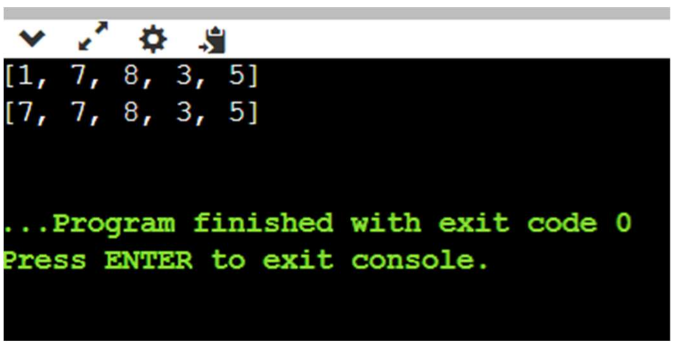
AIM: Write a program to design a circular queue(k) which Should implement given functions.

SOURCE CODE:

```
n = 5
a = [0]*n
front = -1
rear = -1
def enqueue(a,x):
    global front,rear
    if front and rear == -1:
        front = rear = 0
        a[rear] = x
    elif (rear + 1)% n == front:
        print("Queue is full")
    else:
        rear = (rear+1)%n
        a[rear] = x
def dequeue(a):
    global front,rear
    if front and rear == -1:
        print("Queue is empty")
    elif front == rear:
        front = rear = -1
    else:
        front = (front+1)%n

enqueue(a,1)
enqueue(a,7)
enqueue(a,8)
enqueue(a,3)
enqueue(a,5)

print(a)
```

```
dequeue(a)
dequeue(a)
dequeue(a)
enqueue(a,7)

print(a)
```

OUTPUT:



Result: The above code is successfully executed in Lab.

PRACTICAL 5

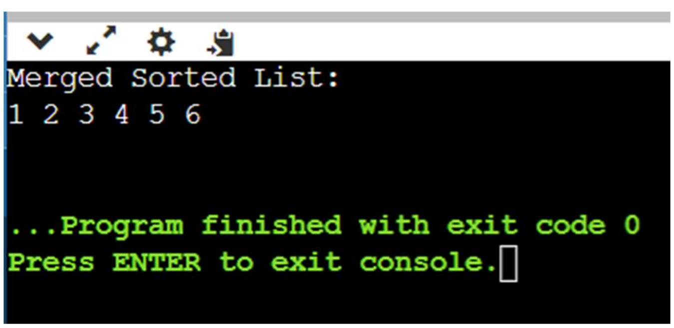
AIM: Write a Program to Merge two linked lists(sorted).

SOURCE CODE:-

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def sort_two_lists(list1, list2):
    t1 = list1
    t2 = list2
    dummy_node = Node(-1)
    temp = dummy_node

    while t1 is not None and t2
is not None:
        if t1.data < t2.data:
            temp.next = t1
            temp = t1
            t1 = t1.next
        else:
            temp.next = t2
            temp = t2
            t2 = t2.next

    if t1:
        temp.next = t1
    else:
        temp.next = t2

    return dummy_node.next

# Function to print the
elements of a linked list
def print_linked_list(head):
```

```
    temp = head
    while temp:
        print(temp.data, end=" ")
        temp = temp.next
    print()

# Example usage
if __name__ == "__main__":

    list1 = Node(1)
    list1.next = Node(3)
    list1.next.next = Node(5)

    list2 = Node(2)
    list2.next = Node(4)
    list2.next.next = Node(6)

    merged_list =
sort_two_lists(list1, list2)
    print("Merged Sorted List:")

print_linked_list(merged_list)
```

OUTPUT:



Result : The above code is successfully executed in Lab.

PRACTICAL 6

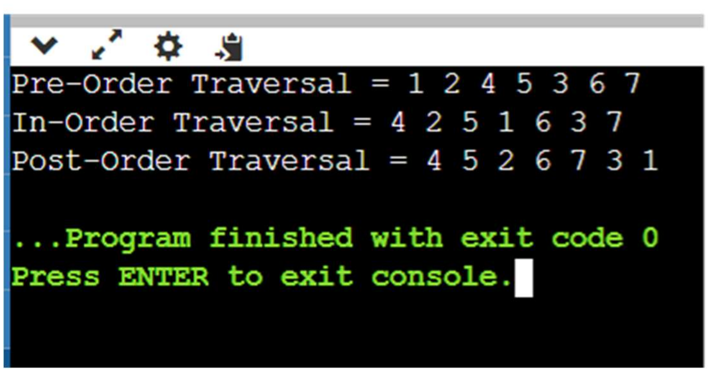
AIM: Write a Program to Understand and implement Tree traversals i.e. Pre-Order Post-Order, In-Order.

SOURCE CODE:

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def preOrderTraversal(root):
    if root is None:
        return
    print(root.val, end=" ")
    preOrderTraversal(root.left)
    preOrderTraversal(root.right)

def inOrderTraversal(root):
    if root is None:
        return
    inOrderTraversal(root.left)
    print(root.val, end=" ")
    inOrderTraversal(root.right)

def postOrderTraversal(root):
    if root is None:
        return
    postOrderTraversal(root.left)

postOrderTraversal(root.right)
    print(root.val, end=" ")
```

```
if __name__ == "__main__":
    Root = TreeNode(1)
    Root.left = TreeNode(2)
    Root.right = TreeNode(3)
    Root.left.left = TreeNode(4)
    Root.left.right = TreeNode(5)
    Root.right.left = TreeNode(6)
    Root.right.right = TreeNode(7)

    print("Pre-Order Traversal =", end=" ")
    preOrderTraversal(Root)
    print()
    print("In-Order Traversal =", end=" ")
    inOrderTraversal(Root)
    print()
    print("Post-Order Traversal =", end=" ")
    postOrderTraversal(Root)
```

OUTPUT:



Result : The above code is successfully executed in Lab.

PRACTICAL 7

AIM: Write a Program to verify and validate mirrored trees or not.

SOURCE CODE:

```
class BinaryTree:
    def __init__(self, root, val =0):
        self.val = val
        self.left = None
        self.right = None

class Solution:
    def isSymmetric(self, root):
        if not root:
            print("Not Symmetric Tree")
        else:
            return self.recursiveCall(root.right, root.left)
    def recursiveCall(self, left, right):
        if left or right:
            return False
        elif left != right:
            return False
        return self.recursiveCall(left.left, right.right)
        return self.recursiveCall(left.right, right.left)

obj1 = BinaryTree(1)
obj1.left = BinaryTree(2)
obj1.right = BinaryTree(2)
obj1.left.left = BinaryTree(3)


obj2 = Solution()
result = obj2.isSymmetric(obj1)
print("Is the tree Symmetric ? ", result)
```

OUTPUT:

```
Is the tree Symmetric ?  False


...Program finished with exit code 0
Press ENTER to exit console.
```

Result: The above code is successfully executed in Lab.

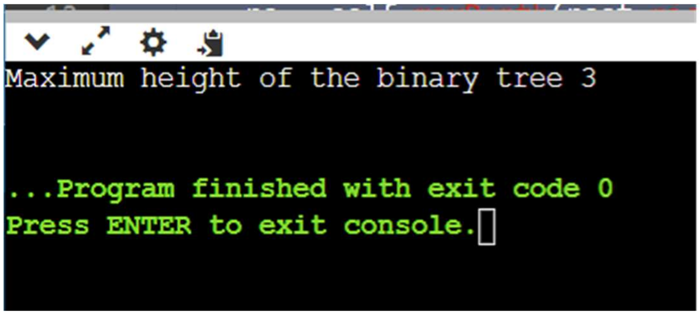
PRACTICAL 8

AIM: Write a Program to determine the depth of a given Tree by Implementing MAXDEPTH.

SOURCE CODE:

```
class TreeNode:
    def __init__(self,val=0,left=None,right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def maxDepth(self,root):
        if not root:
            return 0

        ls = self.maxDepth(root.left)
        rs = self.maxDepth(root.right)

        return (1+max(ls,rs))

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

op = Solution()
result = op.maxDepth(root)
print("Maximum height of the binary tree",result)
```

OUTPUT:



Result: The above code is successfully executed in Lab.

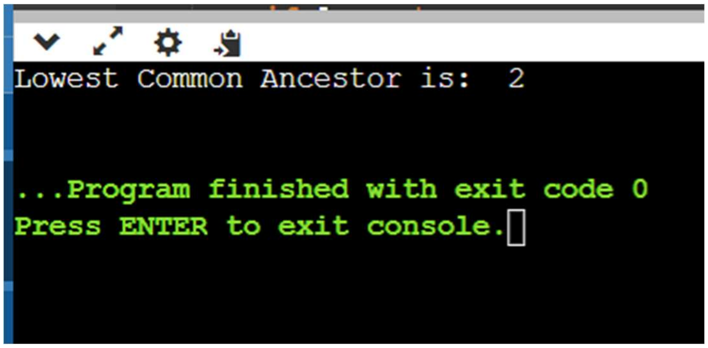
PRACTICAL 9

AIM: Write a program for Lowest Common Ancestors.

SOURCE CODE:

```
class BinaryTree:
    def __init__(self, val=0, left=None, right=None):
            self.val=val
            self.left=left
            self.right=right

class Solution:
    def lowCan(self, root, P, Q):
        if not root or root == P or root == Q:
            return root
        ls = self.lowCan(root.left, P, Q)
        rs = self.lowCan(root.right, P, Q)

        if ls and rs:
            return root
        return ls or rs

obj1 = BinaryTree(1)
obj1.left = BinaryTree(2)
obj1.left.left = BinaryTree(4)
obj1.left.right = BinaryTree(5)
obj1.right = BinaryTree(3)
obj1.right.left = BinaryTree(6)
obj1.right.left.left = BinaryTree(8)
obj1.right.left.left.right = BinaryTree(9)
obj1.right.right = BinaryTree(7)

obj2 = Solution()
P = obj1.left.left
Q = obj1.left.right
result = obj2.lowCan(obj1, P, Q)
print("Lowest Common Ancestor is: ", result.val)
```

OUTPUT:



Result: The above code is successfully executed in Lab.

PRACTICAL 10

AIM: Write a program to build Binary Search Tree(BST).

SOURCE CODE:

```
#create a class to build a binary tree
class BinaryTree:
    def __init__(self, val=0, right=None, left=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def insert(self, cur, val):
        if cur is None:
            return BinaryTree(val)
        elif val < cur.val:
            cur.left = self.insert(cur.left, val)
        else:
            cur.right = self.insert(cur.right, val)
        return cur

    def inOrder(self, cur):
        if cur:
            cur.left = self.inOrder(cur.left)
            print(cur.val)
            cur.right = self.inOrder(cur.right)

obj1 = None
obj2 = Solution()
obj1 = obj2.insert(obj1,8)
obj1 = obj2.insert(obj1,10)
obj1 = obj2.insert(obj1,7)
obj1 = obj2.insert(obj1,14)
obj1 = obj2.insert(obj1,11)
obj1 = obj2.insert(obj1,5)
obj1 = obj2.insert(obj1,15)

result = obj2.inOrder(obj1)
print(result)
```

OUTPUT:

```
5
7
8
10
11
14
15
None


...Program finished with exit code 0
Press ENTER to exit console.
```

Result: The above code is successfully executed in Lab.

PRACTICAL 11

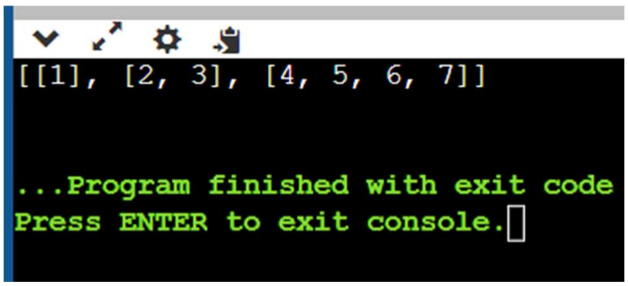
AIM: Write a Program to Traverse a Tree using Level Order Traversal.

SOURCE CODE:

```
class BinaryTree:
    def __init__(self, val = 0, right = None, left = None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrder(self, root):
        if root is None:
            return
        result = [[root.val]]
        Q = []
        Q.append(root)
        while Q:
            temp = []
            for i in range(len(Q)):
                a = Q.pop(0)
                if a.left:
                    temp.append(a.left.val)
                    Q.append(a.left)
                if a.right:
                    temp.append(a.right.val)
                    Q.append(a.right)
            if len(temp) > 0:
                result.append(temp)
        return result


obj1 = BinaryTree(1)
obj1.left = BinaryTree(2)
obj1.left.left = BinaryTree(4)
obj1.left.right = BinaryTree(5)
obj1.right = BinaryTree(3)
obj1.right.left = BinaryTree(6)
obj1.right.right = BinaryTree(7)
```

```
obj2 = Solution()
result1 = obj2.levelOrder(obj1)
print(result1)
```

OUTPUT:



Result: The above code is successfully executed in Lab.

PRACTICAL 12

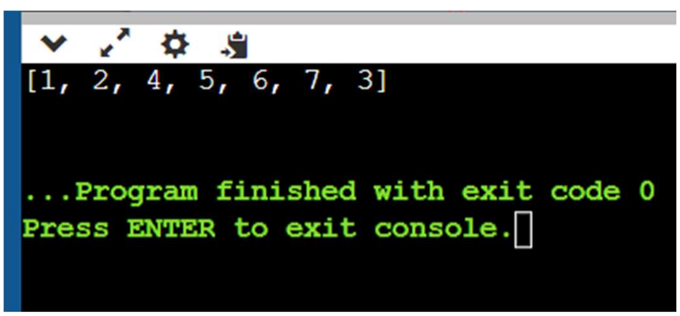
AIM: Write a Program to perform Boundary Traversal on BST.

SOURCE CODE:

```
# Boundary Traversal in a BST
class BinaryTree:
    def __init__(self, val = 0, left = None, right = None):
        self.val = val
        self.left = left
        self.right = right

def leftSub(root, res):
    if root:
        if root.left:
            res.append(root.val)
            leftSub(root.left, res)

        elif root.right:
            res.append(root.val)
            leftSub(root.right, res)

def leaf(root, res):
    if root:
        leaf(root.left, res)
        if not root.left and not root.right:
            res.append(root.val)
        leaf(root.right, res)

def rightSub(root, res):
    if root:
        if root.right:
            if root.right:
                rightSub(root.right, res)

            elif root.left:
                rightSub(root.left, res)

            if root.right or root.left:
                res.append(root.val)
```

```
def boundary(root):
    if root is None:
        return []
    res = [root.val]
    leftSub(root.left, res)
    leaf(root, res)
    rightSub(root.right, res)
    return res

obj1 = BinaryTree(1)
obj1.left = BinaryTree(2)
obj1.left.left = BinaryTree(4)
obj1.left.right = BinaryTree(5)
obj1.right = BinaryTree(3)
obj1.right.right = BinaryTree(7)
obj1.right.left = BinaryTree(6)

res1 = boundary(obj1)
print(res1)
```

OUTPUT:



Result: The above code is successfully executed in Lab.