**Date:**

**WEEK-1**

**1) Aim**: Implement Binary Tree Inorder Traversal

**Program:**

class Solution:

    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:

        ans = []

        if not root: return

        def inorder(node):

            if not node:

                return

            inorder(node.left)

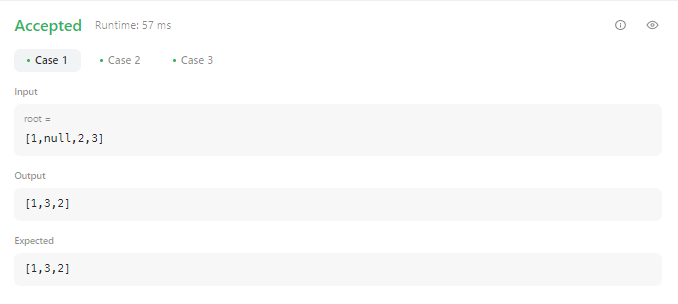
            ans.append(node.val)

            inorder(node.right)

        inorder(root)

        return ans

**Output:**



**Result:** Program executed successfully

**2) Aim**: Implement Binary Tree Preorder Traversal

**Program:**

class Solution:

    def preorderTraversal(self, root):

        p=[]

        self.Traverse(root,p)

        return p

    def Traverse(self,root,p):

        if root:

            p.append(root.val)

            self.Traverse(root.left,p)

            self.Traverse(root.right,p)

**Output:**



**Result:** Program executed successfully

**3) Aim**: Implement Binary Tree Postorder Traversal

**Program:**

class Solution:

    def postorderTraversal(self, root):

        p=[]

        self.Traverse(root,p)

        return p

    def Traverse(self,root,p):

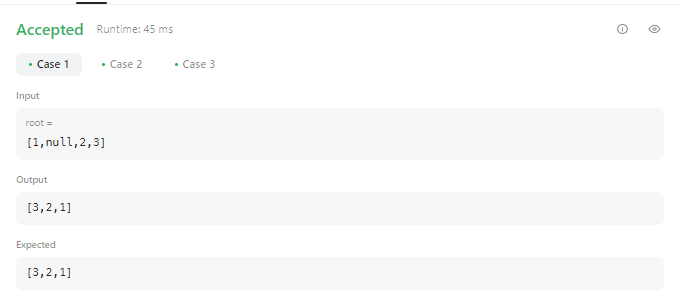
        if root:

            self.Traverse(root.left,p)

            self.Traverse(root.right,p)

            p.append(root.val)

**Output:**



**Result:** Program executed successfully

**Date:**

**WEEK-2**

**Aim**: Implement Validate Stack Sequences

**Program:**

class Solution:

    def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:

        stack=[]

        for i in pushed:

            stack.append(i)

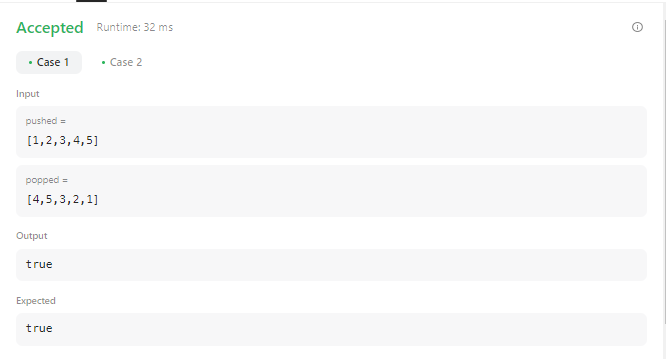
            while stack and popped and stack[-1] == popped[0]:  # stack top

                stack.pop()

                popped.pop(0)

        return not stack

**Output:**



**Result:** Program executed successfully

**Date:**

**WEEK-3**

**1) Aim**: Implement Stack using Queues

**Program:**

from collections import deque

class MyStack:

    def \_\_init\_\_(self):

        self.queue1 = deque()

        self.queue2 = deque()

    def push(self, x: int) -> None:

        self.queue1.append(x)

    def pop(self) -> int:

        while len(self.queue1) > 1:

            self.queue2.append(self.queue1.popleft())

        item = self.queue1.popleft()

        self.queue1, self.queue2 = self.queue2, self.queue1

        return item

    def top(self) -> int:

        while len(self.queue1) > 1:

            self.queue2.append(self.queue1.popleft())

        item = self.queue1.popleft()

        self.queue2.append(item)

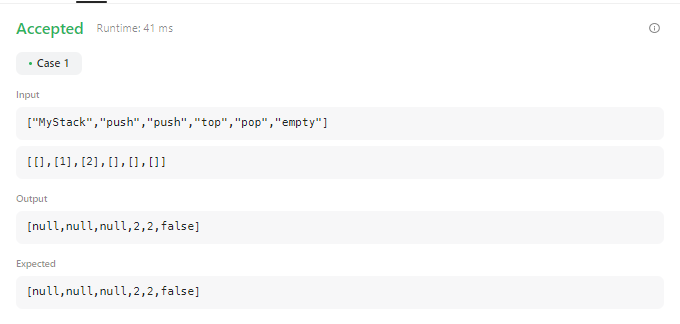
        self.queue1, self.queue2 = self.queue2, self.queue1

        return item

    def empty(self) -> bool:

        return len(self.queue1) == 0

**Output:**

****

**Result:** Program executed successfully

**2) Aim**: Implement Queues using Stack

**Program:**

class MyQueue:

    def \_\_init\_\_(self):

        self.\_stack1 = []

        self.\_stack2 = []

        self.\_length = 0

    def push(self, x: int) -> None:

        self.\_stack1.append(x)

        self.\_length += 1

        return None

    def pop(self) -> int:

        if self.\_stack2:

            self.\_length -= 1

            return self.\_stack2.pop()

        else:

            while self.\_stack1:

                self.\_stack2.append(self.\_stack1.pop())

            if self.\_stack2:

                self.\_length -= 1

                return self.\_stack2.pop()

            else:

                raise Exception()

    def peek(self) -> int:

        if self.\_stack2:

            return self.\_stack2[-1]

        else:

            while self.\_stack1:

                self.\_stack2.append(self.\_stack1.pop())

            if self.\_stack2:

                return self.\_stack2[-1]

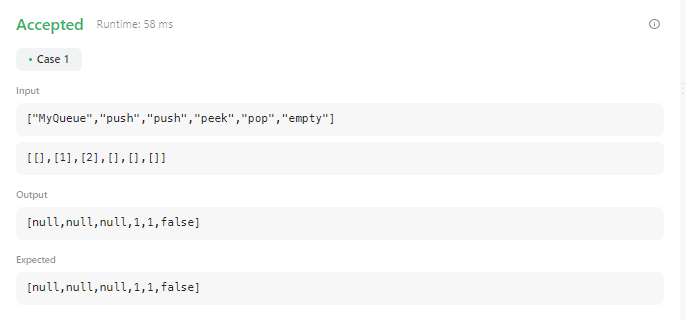
            else:

                raise Exception()

    def empty(self) -> bool:

        return not bool(self.\_length)

**Output:**



**Result:** Program executed successfully

**Date:**

**WEEK-4**

**1) Aim**: Implement Circular Queue

**Program:**

class MyCircularQueue:

    def \_\_init\_\_(self, k: int):

        self.queue = [None] \* k

        self.head = 0

        self.tail = 0

        self.size = 0

        self.capacity = k

    def enQueue(self, value: int) -> bool:

        if self.isFull():

            return False

        self.queue[self.tail] = value

        self.tail = (self.tail + 1) % self.capacity

        self.size += 1

        return True

    def deQueue(self) -> bool:

        if self.isEmpty():

            return False

        self.head = (self.head + 1) % self.capacity

        self.size -= 1

        return True

    def Front(self) -> int:

        if self.isEmpty():

            return -1

        return self.queue[self.head]

    def Rear(self) -> int:

        if self.isEmpty():

            return -1

        return self.queue[(self.tail - 1 + self.capacity) % self.capacity]

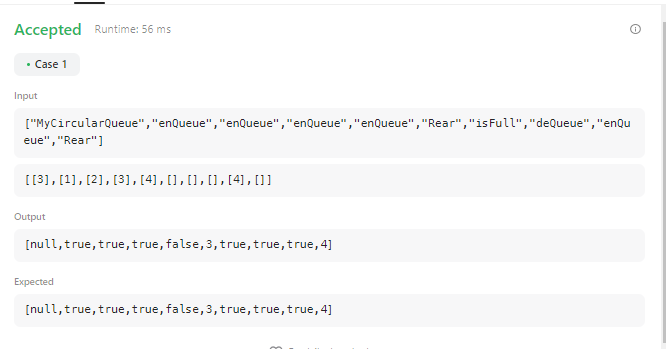
    def isEmpty(self) -> bool:

        return self.size == 0

    def isFull(self) -> bool:

        return self.size == self.capacity

**Output:**



**Result:** Program executed successfully

**2) Aim**: Implement Circular Deque

**Program:**

class MyCircularDeque:

    def \_\_init\_\_(self, k: int):

        self.k = k

        self.deque = []

    def insertFront(self, value: int) -> bool:

        if len(self.deque) < self.k:

            self.deque = [value] + self.deque

            return True

        else:

            return False

    def insertLast(self, value: int) -> bool:

        if len(self.deque) < self.k:

            self.deque = self.deque + [value]

            return True

        else:

            return False

    def deleteFront(self) -> bool:

        if self.deque:

            self.deque = self.deque[1:]

            return True

        else:

            return False

    def deleteLast(self) -> bool:

        if self.deque:

            self.deque = self.deque[:-1]

            return True

        else:

            return False

    def getFront(self) -> int:

        return self.deque[0] if self.deque else -1

    def getRear(self) -> int:

        return self.deque[-1] if self.deque else -1

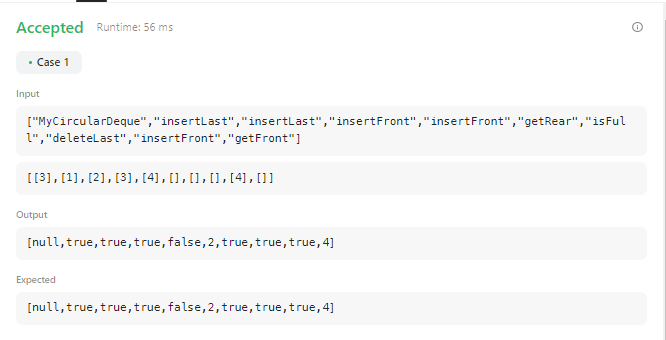
    def isEmpty(self) -> bool:

        return not self.deque

    def isFull(self) -> bool:

        return len(self.deque) == self.k

**Output:**

 **Result:** Program executed successfully

**Date:**

**WEEK-5**

**1) Aim**: Implement Same Tree

**Program:**

class Solution:

    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:

        # If both are none, return True

        if(p == None and q == None):

            return True

        if(p == None and q != None):

            return False

        if(q == None and p != None):

            return False

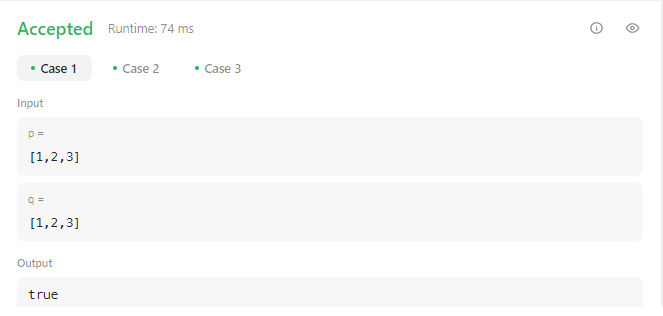
        if(p != None and q!= None):

            if(p.val != q.val):

                return False

            return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

**Output:**



**Result:** Program executed successfully

**2) Aim**: Implement Symmetric Tree

**Program:**

class Solution:

  def isSymmetric(self, root):

    if root is None:

      return True

    else:

      return self.isMirror(root.left, root.right)

  def isMirror(self, left, right):

    if left is None and right is None:

      return True

    if left is None or right is None:

      return False

    if left.val == right.val:

      outPair = self.isMirror(left.left, right.right)

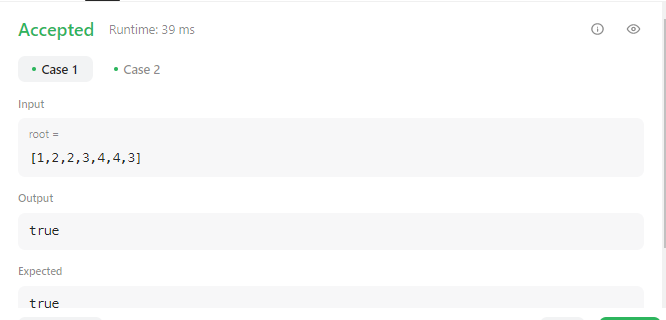
      inPiar = self.isMirror(left.right, right.left)

      return outPair and inPiar

    else:

      return False

**Output:**



**Result:** Program executed successfully

**Date:**

**WEEK-6**

**1) Aim:** Implement Counting Bits

**Program:**

class Solution:

    def countBits(self, n: int) -> List[int]:

        res = [0]

        for num in range(1, n+1):

            count = 0

            while num:

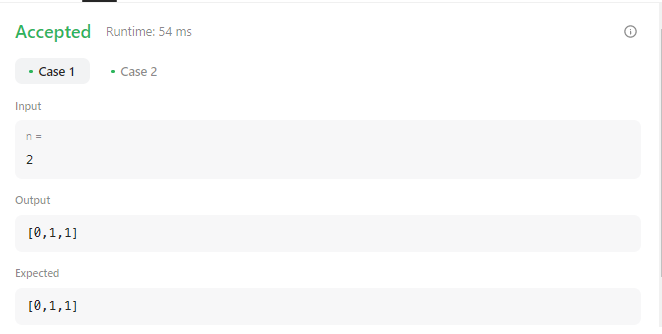
                count += num & 1

                num = num >> 1

            res.append(count)

        return res

**Output:**

****

**Result:** Program executed successfully

**2) Aim:** Implement all possible full binary trees

**Program:**

class Solution:

    def allPossibleFBT(self, n: int) -> List[Optional[TreeNode]]:

        def possibles(m):

            if m == 0: yield TreeNode(0)

            for p in range(1, m, 2):

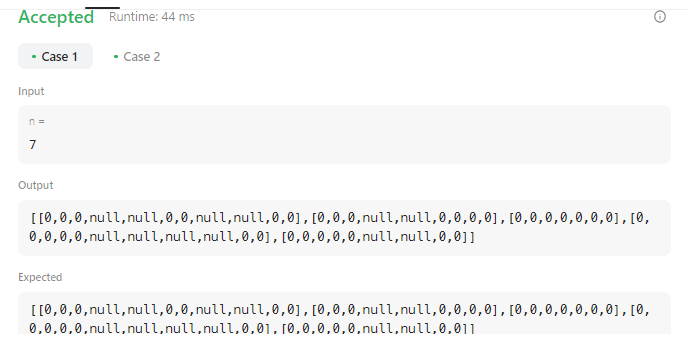
                for left in possibles(p - 1):

                    for right in possibles(m - p - 1):

                        yield TreeNode(0, left, right)

        return possibles(n-1)

**Output:**



**Result:** Program executed successfully

**Date:**

**WEEK-7**

**Aim:** Implement Regular Expression Matching

**Program:**

class Solution:

    def isMatch(self, s: str, p: str) -> bool:

        @cache

        def dfs(i, j):

            if i >= len(s) and j >= len(p):

                return True

            if j >= len(p):

                return False

            match = i < len(s) and (s[i] == p[j] or p[j] == '.')

            if (j+1) < len(p) and p[j+1] == "\*":

                return dfs(i, j+2) or (match and dfs(i+1, j))

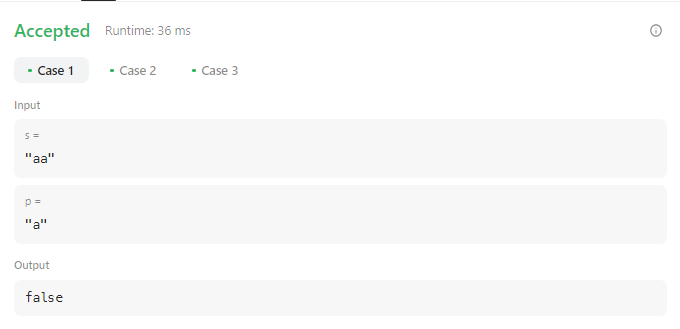
            if match:

                return dfs(i+1, j+1)

            return False

        return dfs(0, 0)

**Output:**



**Result:** Program executed successfully