```
#2.a   Multiplication without using * Operator
def multiply(x, y):
    if y < 0:
        return -multiply(x, -y)
    elif y == 0:
        return 0
    elif y == 1:
        return x
    else:
        return x + multiply(x, y - 1)

print(multiply(5, 5));
#2.b   Tower’s of Hanoi
def TowerOfHanoi(n , source, destination, auxiliary):
    if n==1:
        print ("Move disk 1 from source",source,"to destination",destination)
        return
    TowerOfHanoi(n-1, source, auxiliary, destination)
    print ("Move disk",n,"from source",source,"to destination",destination)
    TowerOfHanoi(n-1, auxiliary, destination, source)

n = 2
TowerOfHanoi(n,'A','B','C')

#1.c        Ackermann’s Problem
def A(m, n, s ="% s"):
    print(s % ("A(% d, % d)" % (m, n)))
    if m == 0:
        return n + 1
    if n == 0:
        return A(m - 1, 1, s)
    n2 = A(m, n - 1, s % ("A(% d, %% s)" % (m - 1)))
    return A(m - 1, n2, s)

print(A(1, 2))
#2.d      Convert given number Decimal to Binary and Binary to Decimal
binary_string = input("Enter a binary number :")

try:
    decimal = int(binary_string,2)
    print("The decimal value is :", decimal)

except ValueError:
    print("Invalid binary number")

#2.f     Convert given Digit to String
num = 10
print("Type of variable before conversion : ", type(num))
converted_num = str(num)
print("Type After conversion : ",type(converted_num))
print(num)
print(converted_num)

#3.1      Euclid’s Algorothm ( Greatest Common Divisor)
def gcd(m,n):
    if m< n:
        (m,n) = (n,m)
    if(m%n) == 0:
        return n
    else:
        return (gcd(n, m % n))
print(gcd(8,12))
```

```
#3.2   Check the given number is Prime or Not
num = 29
flag = False
if num > 1:
    for i in range(2, num):
        if (num % i) == 0:
            flag = True
            break
if flag:
    print(num, "is not a prime number")
else:
    print(num, "is a prime number")
```

```
#3.3  find Prime Factors of a given Number
import math
def primefactors(n):
    while n % 2 == 0:
        print (2),
        n = n / 2
    for i in range(3,int(math.sqrt(n))+1,2):
        while (n % i == 0):
            print (i)
            n = n / i
    if n > 2:
        print (n)
n = int(input("Enter the number for calculating the prime factors :\n"))
primefactors(n)
```

```
#3.4      Binomial Coefficient
def binomialCoeff(n, k):
    if k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k)
n = 5
k = 2
print ("Value of C(%d,%d) is (%d)" % (n, k, binomialCoeff(n, k)))

#4.1   Implement two Stacks in single Array
import math
class twoStacks:
    def __init__(self, n):
        self.size = n
        self.arr = [None] * n
        self.top1 = math.floor(n/2) + 1
        self.top2 = math.floor(n/2)
    def push1(self, x):
        if self.top1 > 0:
            self.top1 = self.top1 - 1
            self.arr[self.top1] = x
        else:
            print("Stack Overflow by element : ", x)
    def push2(self, x):
        if self.top2 < self.size - 1:
            self.top2 = self.top2 + 1
            self.arr[self.top2] = x
        else:
            print("Stack Overflow by element : ", x)
    def pop1(self):
        if self.top1 <= self.size/2:
            x = self.arr[self.top1]
            self.top1 = self.top1 + 1
            return x
        else:
            print("Stack Underflow")
            exit(1)
    def pop2(self):
        if self.top2 >= math.floor(self.size/2) + 1:
            x = self.arr[self.top2]
            self.top2 = self.top2 - 1
            return x
        else:
            print("Stack Underflow")
            exit(1)
if __name__ == '__main__':
    ts = twoStacks(5)
    ts.push1(5)
    ts.push2(10)
    ts.push2(15)
    ts.push1(11)
    ts.push2(7)
    print("Popped element from stack1 is : " + str(ts.pop1()))
    ts.push2(40)
    print("Popped element from stack2 is : " + str(ts.pop2()))

#4.2    Infix to Postfix Conversion
class Conversion:
        def __init__(self,capacity):
            self.top = -1
            self.capacity = capacity
            self.array = []
            self.output = []
            self.precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
        def isEmpty(self):
            return True if self.top == -1 else False
        def peek(self):
            return self.array[-1]
        def pop(self):
            if not self.isEmpty():
                self.top -= 1
                return self.array.pop()
            else:
                return "$"
        def push(self, op):
            self.top += 1
            self.array.append(op)
        def isOperand(self, ch):
            return ch.isalpha()
        def notGreater(self, i):
            try:
                a = self.precedence[i]
                b = self.precedence[self.peek()]
                return True if a <= b else False
            except KeyError:
                return False
        def infixToPostfix(self, exp):
            for i in exp:
                if self.isOperand(i):
                    self.output.append(i)
                elif i == '(':
                    self.push(i)
                elif i == ')':
```

```
                    while((not self.isEmpty()) and
                          self.peek() != '('):
                        a = self.pop()
                        self.output.append(a)
                    if (not self.isEmpty() and self.peek() != '('):
                        return -1
                    else:
                        self.pop()
                else:
                    while(not self.isEmpty() and self.notGreater(i)):
                        self.output.append(self.pop())
                    self.push(i)
            while not self.isEmpty():
                self.output.append(self.pop())
            print("".join(self.output))
if __name__ == '__main__':
    exp = "a+b"
    obj = Conversion(len(exp))
    obj.infixToPostfix(exp)

#4.3    Infix to prefix Conversion
def isOperator(x):
    if x == "+":
        return True
    if x == "-":
        return True
    if x == "/":
        return True
    if x == "*":
        return True
    return False
```

```
def postToPre(post_exp):
    s = []
    length = len(post_exp)
    for i in range(length):
        if (isOperator(post_exp[i])):
            op1 = s[-1]
            s.pop()
            op2 = s[-1]
            s.pop()
            temp = post_exp[i] + op2 + op1
            s.append(temp)
        else:
            s.append(post_exp[i])
    ans = ""
    for i in s:
        ans += i
    return ans
if __name__ == "__main__":
    post_exp = "AB+CD-"
    print("Prefix : ", postToPre(post_exp))
```

#5.1 Implement Queue Operations using Two Stacks

```
class Queue:
    def __init__(self):
        self.s1 = []
        self.s2 = []
    def enQueue(self, x):
        while len(self.s1) != 0:
            self.s2.append(self.s1[-1])
            self.s1.pop()
        self.s1.append(x)
        while len(self.s2) != 0:
            self.s1.append(self.s2[-1])
            self.s2.pop()
    def deQueue(self):
        if len(self.s1) == 0:
            print("Q is Empty")
        x = self.s1[-1]
        self.s1.pop()
        return x
if __name__ == '__main__':
    q = Queue()
    q.enQueue(1)
    q.enQueue(2)
    q.enQueue(3)
    print(q.deQueue())
    print(q.deQueue())
    print(q.deQueue())
```

#5.2 Generate Binary Numbers between 1 to N using a Queue

```
def generatePrintBinary(n):
    from queue import Queue
    q = Queue()
    q.put("1")
    while(n > 0):
        n -= 1
        s1 = q.get()
        print(s1)
        s2 = s1
        q.put(s1+"0")
        q.put(s2+"1")
if __name__ == "__main__":
    n = 10
    generatePrintBinary(n)
```

```
#6.1   Implementation of Reverse a Singly Linked List
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def reverse(self):
        prev = None
        current = self.head
        while(current is not None):
            next = current.next
            current.next = prev
            prev = current
            current = next
        self.head = prev
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node
    def printList(self):
        temp = self.head
        while(temp):
            print (temp.data,end=" ")
            temp = temp.next
llist = LinkedList()
llist.push(20)
llist.push(4)

llist.push(15)

llist.push(85)

print ("Given Linked List")

llist.printList()

llist.reverse()

print ("\nReversed Linked List")

llist.printList()
```

#6.2   Swapping of Two nodes in a Singly Linked List without Swapping Data

#7.1   Concatenate two Circular Linked List

#7.2   Maximum and minimum value in linkedlist

```
class Node:

    def __init__(self,data):

        self.data = data;

        self.next = None;

class CreateList:

    #Declaring head and tail pointer as null.

    def __init__(self):

        self.head = Node(None);

        self.tail = Node(None);

        self.head.next = self.tail;

        self.tail.next = self.head;


    #This function will add the new node at the end of the list.

    def add(self,data):

        newNode = Node(data);

        #Checks if the list is empty.

        if self.head.data is None:

            #If list is empty, both head and tail would point to new node.
            self.head = newNode;
            self.tail = newNode;
            newNode.next = self.head;
        else:
            #tail will point to new node.
            self.tail.next = newNode;
            #New node will become new tail.
            self.tail = newNode;
            #Since, it is circular linked list tail will point to head.
            self.tail.next = self.head;

    #Finds out the minimum value node in the list
    def minNode(self):
        current = self.head;
        #Initializing min to initial node data
        minimum = self.head.data;
        if(self.head == None):
            print("List is empty");
        else:
            while(True):
                #If current node's data is smaller than min
                #Then replace value of min with current node's data
                if(minimum > current.data):
                    minimum = current.data;
                current= current.next;
                if(current == self.head):
                    break;
        print("Minimum value node in the list: "+ str(minimum));

    #Finds out the maximum value node in the list
    def maxNode(self):
        current = self.head;
        #Initializing max to initial node data
        maximum = self.head.data;
        if(self.head == None):
            print("List is empty");
        else:
            while(True):
                #If current node's data is greater than max
                #Then replace value of max with current node's data
                if(maximum < current.data):
                    maximum = current.data;
                current= current.next;
                if(current == self.head):
                    break;
        print("Maximum value node in the list: "+ str(maximum));
class CircularLinkedList:
    cl = CreateList();
    #Adds data to the list
    cl.add(5);
    cl.add(20);
    cl.add(10);
    cl.add(1);
    #Prints the minimum value node in the list
    cl.minNode();
    #Prints the maximum value node in the list
    cl.maxNode();
```

#8.1   Check whether two Binary Trees are Identical or Not

```
class Node:
    def __init__(self,data):
        self.data=data
        self.left=None
        self.right=None
root1=Node(1)
leftroot1=Node(2)
rightroot1=Node(3)
root1.left=leftroot1
root1.right=rightroot1
leftroot12=Node(4)
rightroot12=Node(5)
leftroot1.left=leftroot12
rightroot1.left=rightroot12
root2=Node(1)
leftroot2=Node(2)
rightroot2=Node(8)
root2.left=leftroot2
root2.right=rightroot2
leftroot21=Node(4)
rightroot21=Node(5)
leftroot2.left=leftroot21
rightroot2.left=rightroot21
def preorder(root):
    if not root:
        return None
    print(root.data)
    preorder(root.left)
    preorder(root.right)
preorder(root1)
print("-------")
preorder(root2)
print("-------")
def identical(root1,root2):
    if not root1 or not root2:
        return True
    if root1 is not None and root2 is not None:
        return(root1.data==root2.data and identical(root1.left,root2.left) and
identical(root1.right,root2.right))
    return False
if __name__=="__main__":
    if identical(root1,root2):
        print("Identical")
    else:
        print("Not Identical")


#8.2     Height of Binary Tree
class TreeNode:
    def __init__(self,data):
        self.data=data
        self.leftChild=None
        self.rightChild=None
def height(BT):
    if BT is None:
        return 0
    else:
        ldepth=height(BT.leftChild)
        rdepth=height(BT.rightChild)
        if ldepth>rdepth:
            return ldepth+1
        else:
            return rdepth+1
BT=TreeNode(100)
BT.leftChild=TreeNode(200)
BT.rightChild=TreeNode(300)
BT.leftChild.leftChild=TreeNode(400)

BT.leftChild.rightChild=TreeNode(400)

BT.leftChild.leftChild.leftChild=TreeNode(400)

BT.rightChild.rightChild=TreeNode(400)

print("Height of tree is",height(BT))
```

#8.3   Height Balanced

```
class Node:
   def __init__(self, data):
      self.data = data
      self.left = None
      self.right = None
def height(root):
   if root is None:
      return 0
   return max(height(root.left), height(root.right)) + 1
def isBalanced(root):
   if root is None:
      return True
   lh = height(root.left)
   rh = height(root.right)
   if (abs(lh - rh) <= 1) and isBalanced(
         root.left) is True and isBalanced(root.right) is True:
      return True
   return False
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.left.left.left = Node(8)
```

```
if isBalanced(root):
    print("Tree is balanced")
else:
    print("Tree is not balanced")
```