(15) Implement the following searching sorting in Python:-

(i) Linear Search.

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
arr = [1, 2, 3, 4, 5, 6]
print(search(arr, 6))
```

(ii) Binary Search

```
def binSearch(arr, x):
    l = 0
    r = len(arr) - 1
    while l <= r:
        mid = l + (r - l) // 2
        if arr[mid] == x
            return mid
        elif arr[mid] < x:
            l = mid + 1
        else:
            r = mid - 1
    return -1
arr = [1, 2, 3, 4, 5, 6]
print(binSearch(arr, 7))
```

// array needs to be sorted.

(iii) Selection Sort

```
def selectionSort(array, size):
    for ind in range(size):
        min_index = ind
        for j in range(ind+1, size):
            if array[j] < array[min_index]:
                min_index = j

    (array[ind], array[min_index]) = (array[min_index],
                                      array[ind])

arr = [10,9,2,1,4,3,7]
size = len(arr)
selectionSort(arr, size)
```

(iv) Bubble Sort

```
def bubbleSort(arr)
    for i in range(len(arr)):
        swap = false
        for j in range(0, len(arr)-i-1):
            if arr[j] > arr[j+1]:
                (arr[j], arr[j+1]) = (arr[j+1], arr[j])
                swap = true

        if swap == false:
            break

arr = [10,9,2,1,4,3,7]
bubbleSort(arr).
```

(iv) Insertion Sort

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j>=0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key

arr = [12, 11, 13, 5, 6]
insertionSort(arr).
```

(vi) Quick Sort

```
def partition(array, start, end):
    pivot = array[start]
    low = start + 1
    high = end

    while True:
        while low <= high and array[high] >= pivot:
            high = high - 1

        while low <= high and array[low] <= pivot:
            low = low + 1

        if low <= high:
            array[low], array[high] = array[high], array[low]
        else:
            break
    array[start], array[high] = array[high], array[start]
    return high.
```

```
def quickSort(array, start, end):
    if start >= end
        return
    p = partition(array, start, end)
    quickSort(array, start, p-1)
    quickSort(array, p+1, end)
array = [10, 9, 1, 2, 3, 6, 5, 4]
quickSort(array, 0, len(array) - 1)
print(array).
```

(16) Python program for Tower of Hanoi.

```
def TowerHanoi(n, source, destination, temp):
    if n == 1
        print("Move disk 1 from", source, "to", destination)
        return
    TowerHanoi(n-1, source, temp, destination)
    print("Move disk", n, "from", source, "to", destination)

n = 4
TowerHanoi(n, 'A', 'B', 'C')
```

(17) WAP to detect a cycle in linked list.

```
def detectLoop(self):
    s = set()
    temp = self.head
    while temp:
        if temp in s
            return true
        s.add(temp)
        temp = temp.next
    return false.
```

(18) WAP to find length of cycle in linked list.

```
def detectloop(self):
    if self.head is None:
        return 0
    slow = self.head
    fast = self.head
    flag = 0
    while (slow and slow.next and fast and fast.next and
           fast.next.next):
        if slow == fast and flag != 0:
            count = 1
            slow = slow.next
            while (slow != fast):
                slow = slow.next
                count += 1
            return count  # length of loop
        slow = slow.next
        fast = fast.next.next
        flag = 1
    return 0  # no loop
```

(19) Find intersection point of two linked list.

```
def interselt(head1, head2):
    while head2:
        temp = head1
        while temp:
            if temp == head2
                return head2
            temp = temp.next
        head2 = head2.next
    return None
```

(20) WAP to insert value in a sorted way in Linked list.

```
def SortedInsert(self, new_node):
    if self.head is None:
        new_node.next = self.head
        self.head = new_node

    elif self.head.data >= new_node.data:
        new_node.next = self.head
        self.head = new_node
    else:
        current = self.head
        while (current.next is not None and
               current.next.data < new_node.data):
            current = current.next

        new_node.next = current.next
        current.next = new_node.
```

(21) WAP to reverse linked list using recursion

```
def reverse(node):
    if (node == None):
        return node
    if (node.next == None):
        return node
    node = reverse(node.next)
    node.next.next = node
    node.next = None
    return node
```

(22) WAP to reverse linked list using loop.

```
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.
```

(23) WAP to sort a stack.

```
def sortStack(inputStack):
    temp = []
    while inputStack:
        currentEle = inputStack.pop()
    // find position in temporary stack to insert.
        while temp and temp[-1] > currentEle:
            inputStack.append(temp.pop())
        temp.append(currentEle)
    while temp:
        inputStack.append(temp.pop())
```

(24) WAP to reverse a stack.

```
def pop(stack):
    if (isEmpty(stack)):
        print("Underflow")
    return stack.pop()

def push(stack, item):
    stack.append(item)

def isEmpty(stack):
    return len(stack) == 0
```

```
def createStack():
    stack = []
    return stack

def insertBottom(stack, item):
    if isEmpty(stack):
        push(stack, item)
    else:
        temp = pop(stack)
        insertBottom(stack, item)
        push(stack, temp)

def reverse(stack):
    if not isEmpty(stack):
        temp = pop(stack)
        reverse(stack)
        insertBottom(stack, temp)
```

(25) WAP to check if parenthesis is balanced.

```
def isBalanced(exp):
    flag = True
    count = 0
    for x in range(len(exp)):
        if (exp[x] == '('):
            count = count + 1
        else:
            count = count - 1:
        if count < 0:
            flag = false
            break
```

```
if Count[i] != 0
    flag = False
return flag.
```

(26) Implement queue using 2 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:
            return -1;
        x = self.s1[-1]
        self.s1.pop()
        return x.
```

(27) Implement Queue using a single stack.

```
class Queue:
    def __init__(self):
        self.stack = []
    def enQueue(self, item):
        self.stack.append(item)
    def DeQueue(self):
        if len(self.stack) == 0:
            return None
        elif len(self.stack) == 1:
            return self.stack.pop()
        else:
            item = self.stack.pop()
            deq = self.DeQueue()
            self.stack.append(item)
            return deq
    def isEmpty(self):
        return len(self.stack) == 0
    def size(self)
        return len(self.stack)
```

(28) Implement stack using 2 queues.

```
from queue import Queue
class Stack:
    def __init__(self):
        self.q1 = Queue()
        self.q2 = Queue()
```

```
def push (self, item):
    self.q2.put (item)
    while not self.q1.empty():
        self.q2.put (self.q1.get())
    self.q1, self.q2 = self.q2, self.q1

def pop (self):
    if self.q1.empty ():
        return None
    return self.q1.get()

def top (self):
    if self.q1.empty():
        return None
    front = self.q1.get()
    self.q1.put (front)
    return front

def isEmpty (self):
    return self.q1.empty()

def size (self):
    return self.q1.size()
```

(29) WAP to find height of a binary search tree.

```
Class TreeNode:
    def __init__ (self, value):
        self.value = value
        self.left = None
        self.right = None
```

```
def height(root):
    if root is None:
        return -1
    left_height = height(root.left)
    right_height = height(root.right)
    return max(left_height, right_height) + 1
```

(30) WAP to insert an element into a Binary Search tree.

```
Class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
    def insert(root, value):
        if root is None:
            return TreeNode(value)
        if value < root.value:
            root.left = insert(root.left, value)
        else:
            root.right = insert(root.right, value)
        return root.
```

(31) WAP to delete an element from a binary search tree.

```
Class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def minValueNode(node):
        current = node.
```

```
    while current.left is not None:
        current = current.left
    return current

def deleteNode(root, value):
    if root is None:
        return root
    if value < root.value
        root.left = deleteNode(root.left, value)
    elif value > root.value:
        root.right = deleteNode(root.right, value)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        temp = minValueNode(root.right)
        root.value = temp.value
        root.right = deleteNode(root.right, temp.value)

    return root
```

(32) To define a class with Constructor and create objects.

```
class MyClass:
    def __init__(self, name, age):
        self.name = name
        self.age = age
obj1 = MyClass("James", 30)
print("Name", obj1.name, "Age", obj1.age)
```

(33) To define a new class from one or more existing classes.

```
class ParentClass:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print("Hello", self.name)

class ChildClass(ParentClass):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age
    def introduce(self):
        print(self.name, self.age)

child = ChildClass("James", 10)
child.greet()      # Hello, James
child.introduce()  # James, 10
```

(34) To overload the binary operators to perform operations on objects.

```
class list:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        if isinstance(other, list):
            return list(self.x + other.x, self.y + other.y)
```

```
        raise ValueError("Invalid")
    def __str__(self):
        return f"({self.x}, {self.y})"
lo = List(10, 5)
l1 = List(2, 3)
print("Addition", lo+l1)
```

(35) Write a class called Address that has two attributes: number and street name. Make sure you have an init method that initializes the object appropriately.

```
Class Address:
    def __init__(self, number, street_name):
        self.number = number
        self.street_name = street_name

addr = Address(11143, "NYC street")
print(addr.number) # 11143
print(addr.street_name) # NYC street.
```