**Assigment 2**

**Name: Omerullah Ansari**

**ID: 65584**

Q1

def findSecondSmallest(arr):

if len(arr) < 2:

return None

first, second = float('inf'), float('inf')

for number in arr:

if number < first:

second = first

first = number

elif first < number < second:

second = number

return second

def main():

arr = []

print("Enter 10 numbers:")

for \_ in range(10):

while True:

try:

number = int(input(f"Enter number {\_+1}: "))

arr.append(number)

break

except ValueError:

print("Invalid input, please enter an integer.")

secondSmallest = findSecondSmallest(arr)

if secondSmallest is None:

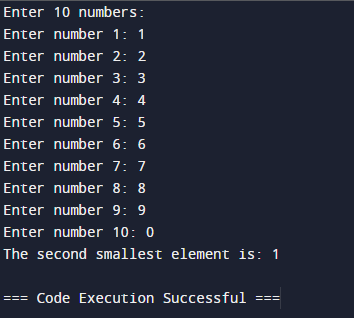
print("Array does not have enough elements.")

else:

print(f"The second smallest element is: {secondSmallest}")

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

main()



Q2

X = [[1,2,3,4],

[5,6,7,8]]

Y = [[1,2,3],

[4,5,6],

[7,8,9],

[10,11,12]]

result = [[0,0,0],

[0,0,0]]

for i in range(len(X)):

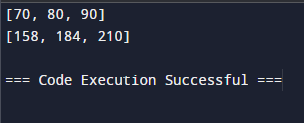
for j in range(len(Y[0])):

for k in range(len(Y)):

result[i][j] += X[i][k] \* Y[k][j]

for r in result:

print(r)



Q3

class Node:

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

self.data = data

self.next = None

class LinkedList:

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

self.head = None

def append(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

return

last = self.head

while last.next:

last = last.next

last.next = new\_node

def print\_list(self):

current = self.head

while current:

print(current.data, end=",")

current = current.next

print("None")

def bubble\_sort\_linked\_list(linked\_list):

end = None

while end != linked\_list.head:

p = linked\_list.head

while p.next != end:

q = p.next

if p.data > q.data:

p.data, q.data = q.data, p.data

p = p.next

end = p

def selection\_sort\_linked\_list(linked\_list):

start = linked\_list.head

while start:

min\_node = start

r = start.next

while r:

if r.data < min\_node.data:

min\_node = r

r = r.next

start.data, min\_node.data = min\_node.data, start.data

start = start.next

import random

SSL = LinkedList()

for \_ in range(20):

SSL.append(random.randint(1, 100))

print("Original List:")

SSL.print\_list()

print("\nBubble Sort:")

bubble\_sort\_linked\_list(SSL)

SSL.print\_list()

print("\nSelection Sort:")

SSL = LinkedList()

for \_ in range(20):

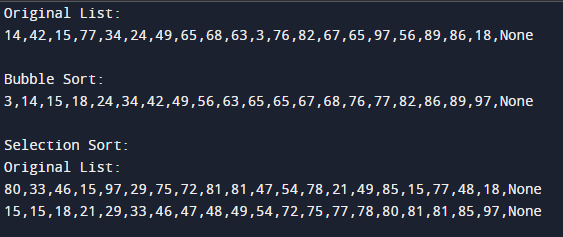
SSL.append(random.randint(1, 100))

print("Original List:")

SSL.print\_list()

selection\_sort\_linked\_list(SSL)

SSL.print\_list()



Q4

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)

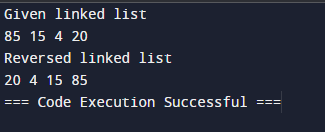
print ("Given linked list")

llist.printList()

llist.reverse()

print ("\nReversed linked list")

llist.printList()



Q5

class Node:

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

self.data = x

self.next = None

class LinkedList:

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

self.head = None

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

def detectLoop(self):

s = set()

temp = self.head

while (temp):

if (temp in s):

return True

s.add(temp)

temp = temp.next

return False

llist = LinkedList()

llist.push(20)

llist.push(4)

llist.push(15)

llist.push(10)

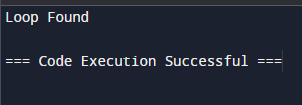
llist.head.next.next.next.next = llist.head

if(llist.detectLoop()):

print("Loop Found")

else:

print("No Loop ")



Q6

class TreeNode:

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

self.key = key

self.left = None

self.right = None

class BinarySearchTree:

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

self.root = None

def insert(self, key):

if self.root is None:

self.root = TreeNode(key)

else:

self.\_insert(self.root, key)

def \_insert(self, node, key):

if key < node.key:

if node.left is None:

node.left = TreeNode(key)

else:

self.\_insert(node.left, key)

else:

if node.right is None:

node.right = TreeNode(key)

else:

self.\_insert(node.right, key)

def descending\_traversal(self):

elements = []

self.\_descending\_traversal(self.root, elements)

return elements

def \_descending\_traversal(self, node, elements):

if node:

self.\_descending\_traversal(node.right, elements)

elements.append(node.key)

self.\_descending\_traversal(node.left, elements)

def find\_min(self):

if self.root is None:

return None

current = self.root

while current.left:

current = current.left

return current.key

def find\_max(self):

if self.root is None:

return None

current = self.root

while current.right:

current = current.right

return current.key

def delete\_min(self):

if self.root is None:

return

self.root = self.\_delete\_min(self.root)

def \_delete\_min(self, node):

if node.left is None:

return node.right

node.left = self.\_delete\_min(node.left)

return node

bst = BinarySearchTree()

values = [20, 10, 30, 5, 15, 25, 35]

for value in values:

bst.insert(value)

print("Descending Traversal:", bst.descending\_traversal())

print("Minimum Value:", bst.find\_min())

print("Maximum Value:", bst.find\_max())

bst.delete\_min()

print("Minimum Value after deleting min:", bst.find\_min())

