INSERTING LINKED LIST:

class Node:

def init(self,data):

self.data = data

self.next = None

class LinkedList:

def init(self):

self.head = None #starting poing of linked list

def insert(self,data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

else:

temp = self.head

while temp.next:

temp = temp.next #moves temp node

temp.next = new\_node

def display(self):

temp = self.head

while temp:

print(temp.data, end="->")

temp =temp.next

print("None")

ll=LinkedList()

ll.insert(10)

ll.insert(20)

ll.insert(30)

ll.insert(40)

ll.display()

o/p:

10->20->30->40->None

SUM OF THE LINKED LIST:

class Node:

def init(self,data):

self.data = data

self.next = None

class LinkedList:

def init(self):

self.head = None #starting poing of linked list

def insert(self,data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

else:

temp = self.head

while temp.next:

temp = temp.next #moves temp node

temp.next = new\_node

def display(self):

temp = self.head

while temp:

print(temp.data, end="->")

temp =temp.next

print("None")

def add\_ll(self):

temp=self.head

sum=0

while temp:

sum=sum+temp.data

temp=temp.next

return sum

ll=LinkedList()

ll.insert(10)

ll.insert(20)

ll.insert(30)

ll.insert(40)

ll.display()

print("sum of elements in the linked list")

ans=ll.add\_ll()

print(ans)

O/P:

10->20->30->40->None

sum of elements in the linked list

100

INSERT THE ELEMENT AT THE BEGINING:

class Node:

def init(self,data):

self.data = data

self.next = None

class LinkedList:

def init(self):

self.head = None #starting poing of linked list

def insert\_end(self,data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

else:

temp = self.head

while temp.next:

temp = temp.next #moves temp node

temp.next = new\_node

def display(self):

temp = self.head

while temp:

print(temp.data, end="->")

temp =temp.next

print("None")

def add\_ll(self):

temp=self.head

sum=0

while temp:

sum=sum+temp.data

temp=temp.next

return sum

def count(self):

pass

def insert\_begining(self,data):

new\_node=Node(data)

new\_node.next=self.head

self.head=new\_node

ll=LinkedList()

ll.insert\_end(10)

ll.insert\_begining(7000)

ll.insert\_end(20)

ll.insert\_end(30)

ll.insert\_end(40)

ll.display()

print("sum of elements in the linked list")

ans=ll.add\_ll()

print(ans)

class Node:

def init(self,data):

self.data = data

self.next = None

class LinkedList:

def init(self):

self.head = None #starting poing of linked list

def insert\_end(self,data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

else:

temp = self.head

while temp.next:

temp = temp.next #moves temp node

temp.next = new\_node

def display(self):

temp = self.head

while temp:

print(temp.data, end="->")

temp =temp.next

print("None")

def add\_ll(self):

temp=self.head

sum=0

while temp:

sum=sum+temp.data

temp=temp.next

return sum

def count(self):

pass

def insert\_begining(self,data):

new\_node=Node(data)

new\_node.next=self.head

self.head=new\_node

def delete\_begining(self):

self.head=self.head.next

def delete\_end(self):

temp = self.head

while temp.next.next:

temp = temp.next

temp.next = None

def insert\_position(self,pos,data):

if(pos==0):

self.insert\_begining(data)

else:

new\_node=Node(data)

temp=self.head

for \_ in range(pos-1):

if temp is None:

print("position out of bounds")

return

temp=temp.next

new\_node.next=temp.next

temp.next=new\_node

def delete\_position(self,pos):

if(pos==0):

self.delete-begining(data)

else:

temp = self.head

for \_ in range(pos-1):

if temp.next is None:

print("position out of bound")

return

temp.next = temp.next.next

def delete\_value(self,value):

if self.head.data == value:

self.head = self.head.next

return

temp = self.head

while temp.next and temp.next.data != value:

temp = temp.next

if temp.next is None:

print("value not found")

return

temp.next = temp.next.next

ll=LinkedList()

ll.insert\_begining(10)

ll.insert\_begining(20)

ll.insert\_begining(30)

ll.insert\_position(3,900)

ll.insert\_end(40)

ll.delete\_position(2)

ll.delete\_value(20)

ll.display()

print("sum of elements in the linked list")

ans=ll.add\_ll()

print(ans)

def check\_password\_strength(password):

score = 0

DAY-6

# Rule 1: Only lowercase letters

if password.islower():

score = 6

# Rule 2: Has lowercase, uppercase, and numbers

if any(c.islower() for c in password) and \

any(c.isupper() for c in password) and \

any(c.isdigit() for c in password):

score = 8

# Rule 3: Has lowercase, uppercase, numbers, and special characters

if any(c.islower() for c in password) and \

any(c.isupper() for c in password) and \

any(c.isdigit() for c in password) and \

any(not c.isalnum() for c in password): # special characters

score = 10

# Check the score and print result

if score == 6:

print("Password is week")

elif score == 8:

print("Password is strong")

elif score == 10:

print("Password is very strong")

# Test the function

check\_password\_strength("tiger") # Output: week

check\_password\_strength("LioN123") # Output: strong

check\_password\_strength("LionTig@r123") # Output: very strong

Inheritance:

class Grandfather:

def grandfather\_method(self):

return "This is grandfather method"

class Father(Grandfather): # Inherits from Grandfather

def father\_method(self):

return "This is father method"

class Mother:

def mother\_method(self):

return "This is mother method"

class Child(Father, Mother): # Inherits from both Father and Mother

def child\_method(self):

return "This is child method"

# Creating proper object instances

grandfather\_object = Grandfather()

father\_object = Father()

mother\_object = Mother()

child\_object = Child()

# Accessing methods correctly

print(grandfather\_object.grandfather\_method())

print(father\_object.father\_method())

print(father\_object.grandfather\_method()) # Inherited

print(mother\_object.mother\_method())

print(child\_object.child\_method())

print(child\_object.father\_method()) # Inherited

print(child\_object.grandfather\_method()) # Inherited

print(child\_object.mother\_method())

DAY-7

class Node():

def init(self):

self.left=None

self.right=None

self.data=None

def postorder\_traversal(self,Node):

if Node:

self.postorder\_traversal(Node.left)

self.postorder\_traversal(Node.right)

print(Node.data)

tree=Node()

tree.data=1

tree.left=Node()

tree.left.data=2

tree.right=Node()

tree.right.data=3

tree.left.left=Node()

tree.left.left.data=4

tree.left.right=Node()

tree.left.right.data=5

tree.right.left=Node()

tree.right.left.data=6

tree.right.right=Node()

tree.right.right.data=7

tree.postorder\_traversal(Node=tree)

class Node():

def init(self):

self.left=None

self.right=None

self.data=None

def postorder\_traversal(self,Node):

if Node:

self.postorder\_traversal(Node.left)

self.postorder\_traversal(Node.right)

print(Node.data)

def Height(self,Node):

if Node is None:

return 0

else:

return max(self.Height(Node.left),self.Height(Node.right))+1

tree=Node()

tree.data=1

tree.left=Node()

tree.left.data=2

tree.right=Node()

tree.right.data=3

tree.left.left=Node()

tree.left.left.data=4

tree.left.right=Node()

tree.left.right.data=5

tree.right.left=Node()

tree.right.left.data=6

tree.right.right=Node()

tree.right.right.data=7

tree.postorder\_traversal(Node=tree)

print(tree.Height(Node=tree))

class Node():

def init(self):

self.left=None

self.right=None

self.data=None

def postorder\_traversal(self,Node):

if Node:

self.postorder\_traversal(Node.left)

self.postorder\_traversal(Node.right)

print(Node.data)

def count\_nodes(self,Node):

if Node is None:

return 0

return 1+self.count\_nodes(Node.left)+self.count\_nodes(Node.right)

tree=Node()

tree.data=1

tree.left=Node()

tree.left.data=2

tree.right=Node()

tree.right.data=3

tree.left.left=Node()

tree.left.left.data=4

tree.left.right=Node()

tree.left.right.data=5

tree.right.left=Node()

tree.right.left.data=6

tree.right.right=Node()

tree.right.right.data=7

tree.postorder\_traversal(Node=tree)

print(tree.count\_nodes(Node=tree))