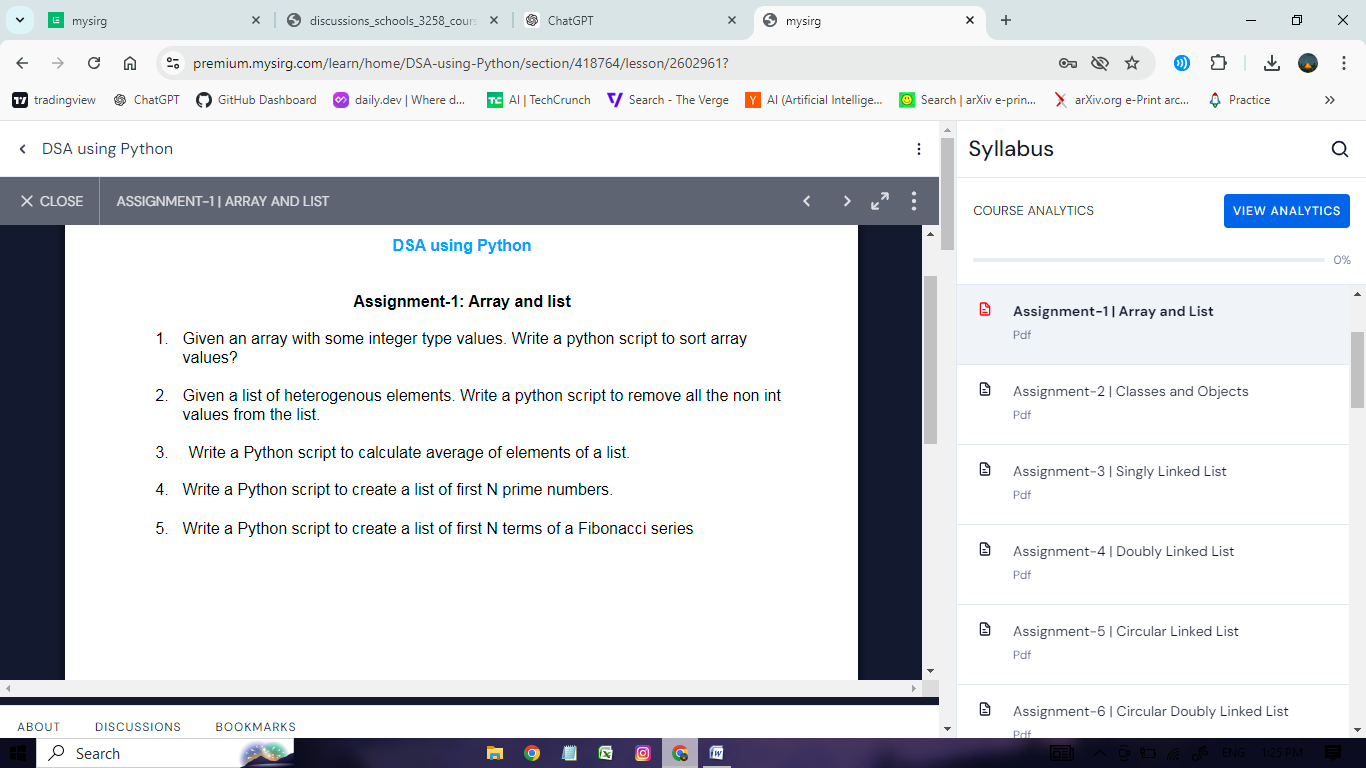
DATA STRUCTURE AND ALGORITHM

PYTHON IMPLANTATION

-------------------------------------------------------------------------------------------------------



------------------------------------------------------------------------------------------------------------------------------------------------

# ans1 (array treated as list)

list = [1, 2, 5, 80, 0, 3]

list.sort()

print(list)

# ans1

from array import \*

a1 = array('i', [2, 1, 5, 4, 9, 6])

a1\_sort = sorted(a1)

print(a1\_sort)

# ans2

l1 = [1, 'shikhar', 1.5, 8, 9, 6.5, 2, 4, True]

l2 = []

for item in l1:

if type(item) == int:

l2.append(item)

print(l2)

# ans3

l1 = [1, 5, 6, 10]

sum = 0

for item in l1:

sum += item

no\_of\_item = len(l1)

avg = sum / no\_of\_item

print(avg)

# ans4

num = int(input("Enter the number: "))

prime\_numbers = []

for i in range(2, num):

is\_prime = True

# Check if 'i' is divisible by any number from 2 to (i-1)

for j in range(2, i):

if i % j == 0:

is\_prime = False

break

if is\_prime:

prime\_numbers.append(i)

print(f"Prime numbers up to {num} are: {prime\_numbers}")

# ans5

def fibonacci(n):

fib\_list = [0, 1]

while len(fib\_list) < n:

next\_term = fib\_list[-1] + fib\_list[-2]

fib\_list.append(next\_term)

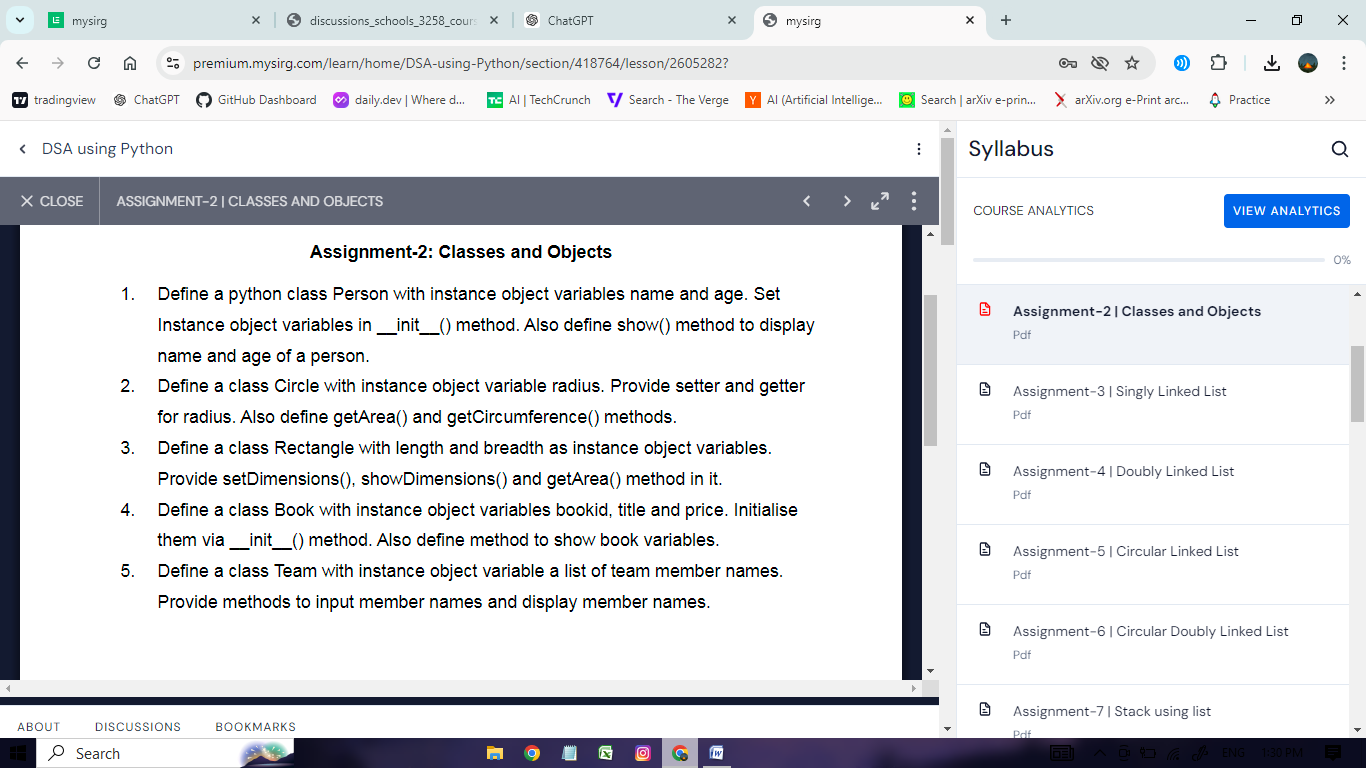
return fib\_list

n = int(input("How many fibonacci numbers you need: "))

fibonacci\_list = fibonacci(n)

print(fibonacci\_list)

#===============================================================



================================================================

#Q1

class Person:

def \_\_init\_\_(self, name=None, age=None):

self.name = name

self.age = age

def setName(self, name):

self.name = name

def setAge(self, age):

self.age = age

def show(self):

print(self.name)

print(self.age)

p1 = Person("Shikhar", 18)

p2 = Person()

p2.setAge(16)

p2.setName("Harshit")

p2.show()

p1.show()

# ===============================================================

#Q2

class Circle:

def \_\_init\_\_(self, radius=None):

self.radius = radius

def setRadius(self, radius):

self.radius = radius

def getRadius(self):

print(self.radius)

def getArea(self):

return 3.14 \* (self.radius \*\* 2)

def getCircumference(self):

return 2 \* 3.14 \* self.radius

c1 = Circle()

c1.setRadius(3)

c1.getRadius()

print(c1.getArea())

print(c1.getCircumference())

# q3

class Rectangle:

def \_\_init\_\_(self, length=None, breadth=None):

self.length = length

self.breadth = breadth

def setDimensions(self, length, breadth):

self.length = length

self.breadth = breadth

def showDimensions(self):

print(f"The length of the rectangle is {self.length} and the breadth is {self.breadth}")

def getArea(self):

print(f"The area of the rectangle is {self.length \* self.breadth}")

r = Rectangle()

r.setDimensions(5, 7)

r.showDimensions()

r.getArea()

# q4

class Book:

def \_\_init\_\_(self, bookid, title, price):

self.bookid = bookid

self.title = title

self.price = price

def show(self):

print(f"The ID of the book is {self.bookid}, the title is {self.title}, and the price is {self.price}")

b = Book(11, "Happy Life", 100)

b.show()

# q5

class Team:

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

self.team\_members = []

def input\_member(self):

while True:

member = input("Enter a team member name (or 'e' for exit): ")

if member.lower() == 'e':

break

else:

self.team\_members.append(member)

def display\_member(self):

if not self.team\_members:

print("No team members found")

else:

print("Team members are:")

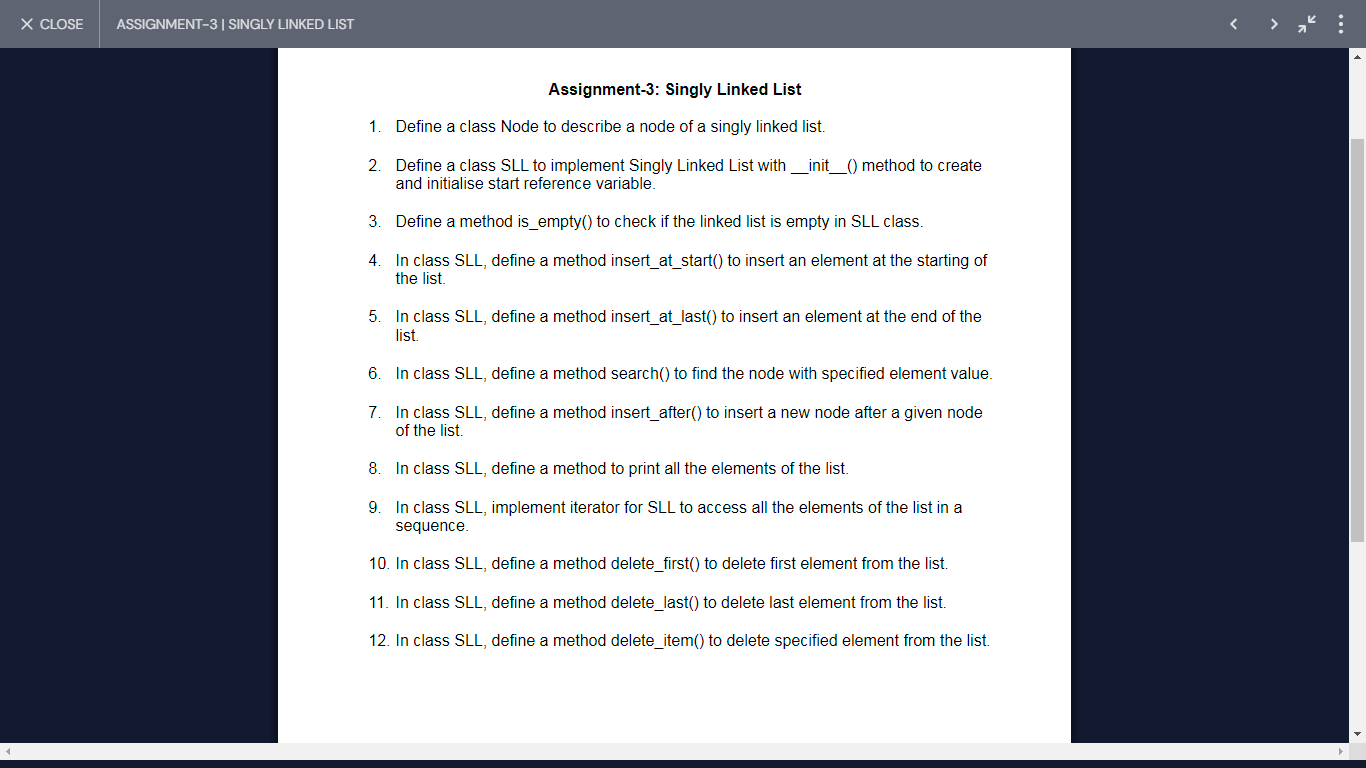
print(f"{self.team\_members}")

t = Team()

t.input\_member()

t.display\_member()

#===============================================================



# q1

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

# q2

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

# driver code

mylist = SLL()

# q3

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self): # q3 implement this line

return self.start == None

mylist = SLL()

# q4 (Insert at start

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self):

return self.start == None

def insert\_at\_start(self, data): # q4 implement this line

n = Node(data, self.start) # create a node name is n and pass the item is data and next is start

self.start = n # value of n assign to the start (now list is connected)

# mylist = SLL()

# q5 (insert at last)

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self):

return self.start == None

def insert\_at\_start(self, data):

n = Node(data, self.start)

self.start = n

def insert\_at\_last(self, data): # q5 implement this line

n = Node(data)

if not self.is\_empty(): # add the node if list is not empty

temp = self.start # we need traversing now create temp variable which refer to the first node

while temp.next is not None: # check if temp.next is none or not

temp = temp.next

temp.next = n # add node when the loop is end (that means temp.next is none)

else:

self.start = n # when list is empty

mylist = SLL()

# q6 (find node)

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self):

return self.start == None

def insert\_at\_start(self, data): # q4 implement this line

n = Node(data, self.start)

self.start = n

def insert\_at\_last(self, data): # q5 implement this line

n = Node(data)

if not self.is\_empty():

temp = self.start

while temp.next is not None:

temp = temp.next

temp.next = n

else:

self.start = n

def search(self, data): # q6 implement this line

temp = self.start # we need traversing

while temp is not None:

if temp.item == data:

return temp # In the temp reference of those node

temp = temp.next

return None # When the element is not present

mylist = SLL()

# q7 (insert a new node after a given node

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self):

return self.start == None

def insert\_at\_start(self, data): # q4 implement this line

n = Node(data, self.start)

self.start = n

def insert\_at\_last(self, data): # q5 implement this line

n = Node(data)

if not self.is\_empty():

temp = self.start

while temp.next is not None:

temp = temp.next

temp.next = n

else:

self.start = n

def search(self, data): # q6 implement this line

temp = self.start

while temp is not None:

if temp.item == data:

return temp

temp = temp.next

return None

def insert\_after(self, temp, data): # q7 implement this line

if temp is not None:

n = Node(data, temp.next)

temp.next = n

mylist = SLL()

# q8 (print all the elements of the list)

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self):

return self.start == None

def insert\_at\_start(self, data): # q4 implement this line

n = Node(data, self.start)

self.start = n

def insert\_at\_last(self, data): # q5 implement this line

n = Node(data)

if not self.is\_empty():

temp = self.start

while temp.next is not None:

temp = temp.next

temp.next = n

else:

self.start = n

def search(self, data): # q6 implement this line

temp = self.start

while temp is not None:

if temp.item == data:

return temp

temp = temp.next

return None

def insert\_after(self, temp, data): # q7 implement this line

if temp is not None:

n = Node(data, temp.next)

temp.next = n

def print\_list(self): # q8 implement this line

temp = self.start

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

# driver code

mylist = SLL()

mylist.insert\_at\_start(20)

mylist.insert\_at\_start(10)

mylist.insert\_at\_last(30)

mylist.insert\_after(mylist.search(20), 25) # we need to call search function

mylist.print\_list()

#========================================================

# q10,11,12 (Delete Node)

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class SLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self):

return self.start == None

def insert\_at\_start(self, data): # q4 implement this line

n = Node(data, self.start)

self.start = n

def insert\_at\_last(self, data): # q5 implement this line

n = Node(data)

if not self.is\_empty():

temp = self.start

while temp.next is not None:

temp = temp.next

temp.next = n

else:

self.start = n

def search(self, data): # q6 implement this line

temp = self.start

while temp is not None:

if temp.item == data:

return temp

temp = temp.next

return None

def insert\_after(self, temp, data): # q7 implement this line

if temp is not None:

n = Node(data, temp.next)

temp.next = n

def print\_list(self): # q8 implement this line

temp = self.start

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

def delete\_first(self): # q10 (Delete the first node)

if self.start is not None:

self.start = self.start.next

def delete\_last(self): # q11 (Delete the last node)

if self.start is None:

pass

elif self.start.next is None:

self.start = None

else:

temp = self.start

while temp.next.next is not None: # (let temp.last is last node)

temp = temp.next

temp.next = None

def delete\_item(self, data): # q12 (Delete the particular node)

if self.start is None:

pass

elif self.start.next is None: # if list has only 1 node

if self.start.item == data:

self.start = None

else:

temp = self.start

if temp.item == data: # if the first node is the particular node

self.start = temp.next

else:

while temp.next is not None: # temp refers to the node before the one to be deleted

if temp.next.item == data: # found the node to delete

temp.next = temp.next.next

break

temp = temp.next

# driver code

mylist = SLL()

mylist.insert\_at\_start(20)

mylist.insert\_at\_start(10)

mylist.insert\_at\_last(30)

mylist.insert\_after(mylist.search(20), 25) # we need to call search function

mylist.print\_list()

mylist.delete\_first()

mylist.print\_list()

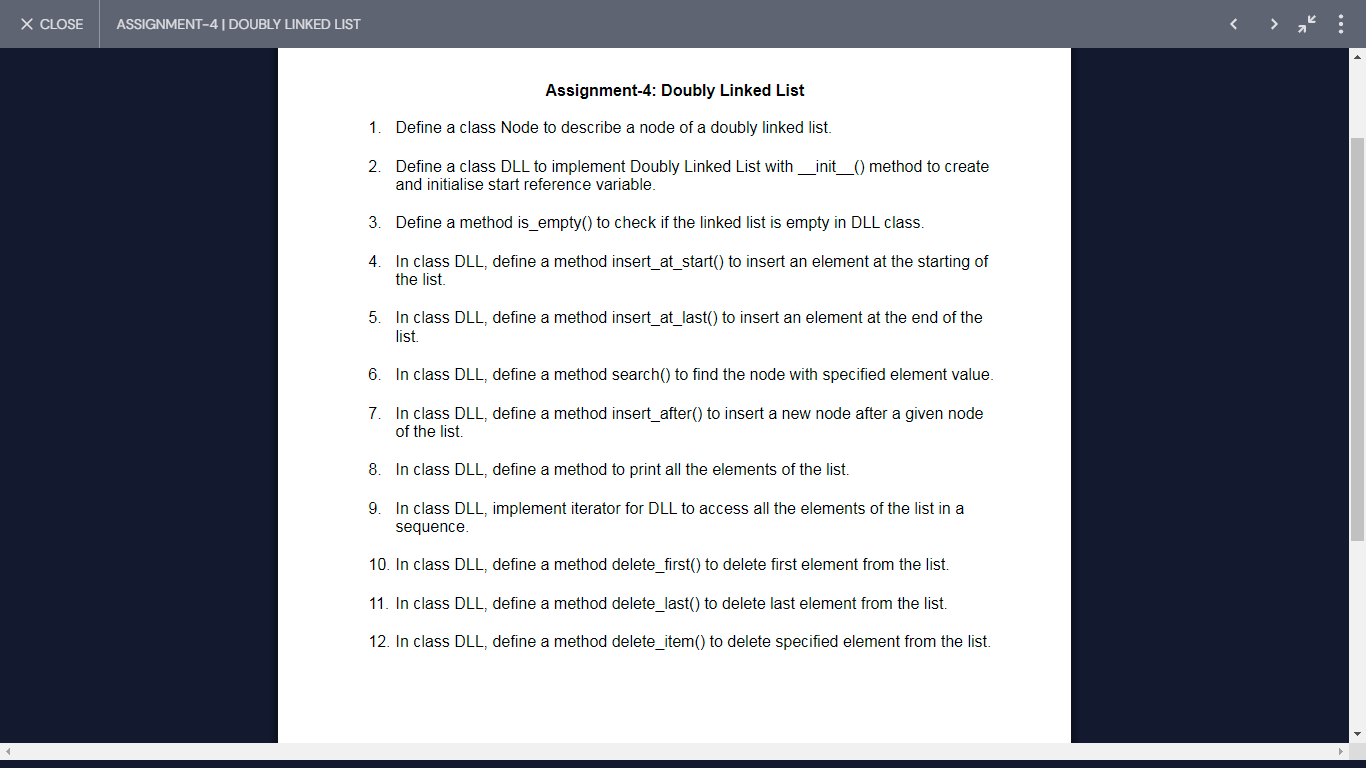
mylist.delete\_last()

mylist.print\_list()

mylist.delete\_item(20)

mylist.print\_list()

================================================================



# q1

class Node:

def \_\_init\_\_(self, prev=None, item=None, next=None):

self.prev = prev

self.item = item

self.next = next

# q2

class DLL:

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self): # q3

return self.start == None

def insert\_at\_start(self, data): # q4

n = Node(None, data, self.start)

if not self.is\_empty():

self.start.prev = n

self.start = n

def insert\_at\_last(self, data): # q5

temp = self.start

if self.start != None:

while temp.next != None:

temp = temp.next

n = Node(temp, data, None)

if temp == None:

self.start = n

else:

temp.next = n

else:

self.start = Node(None, data, None)

def search(self, data): # q6

temp = self.start

while temp is not None:

if temp.item == data:

return temp

temp = temp.next

return None

def insert\_after(self, temp, data): # q7

if temp is not None:

n = Node(temp, data, temp.next)

if temp.next is not None:

temp.next.prev = n

temp.next = n

def print\_list(self): # q8

temp = self.start

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print()

def delete\_first(self): # q10

if self.start is not None:

self.start = self.start.next

if self.start is not None:

self.start.prev = None

def delete\_last(self): # q11

if self.start is None:

pass

elif self.start.next is None:

self.start = None

else:

temp = self.start

while temp.next is not None:

temp = temp.next

temp.prev.next = None

def delete\_item(self, data): # q12

if self.start is None:

pass

else:

temp = self.start

while temp is not None:

if temp.item == data:

if temp.next is not None:

temp.next.prev = temp.prev

if temp.prev is not None:

temp.prev.next = temp.next

else:

self.start = temp.next

break

temp = temp.next

def \_\_iter\_\_(self): # q9

return DLLIterator(self.start)

class DLLIterator:

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

self.current = start

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if not self.current:

raise StopIteration

data = self.current.item

self.current = self.current.next

return data

# Driver code

mylist = DLL()

mylist.insert\_at\_start(10)

mylist.insert\_at\_last(20)

mylist.insert\_after(mylist.search(10), 15)

mylist.insert\_at\_start(20)

mylist.print\_list()

mylist.delete\_item(20)

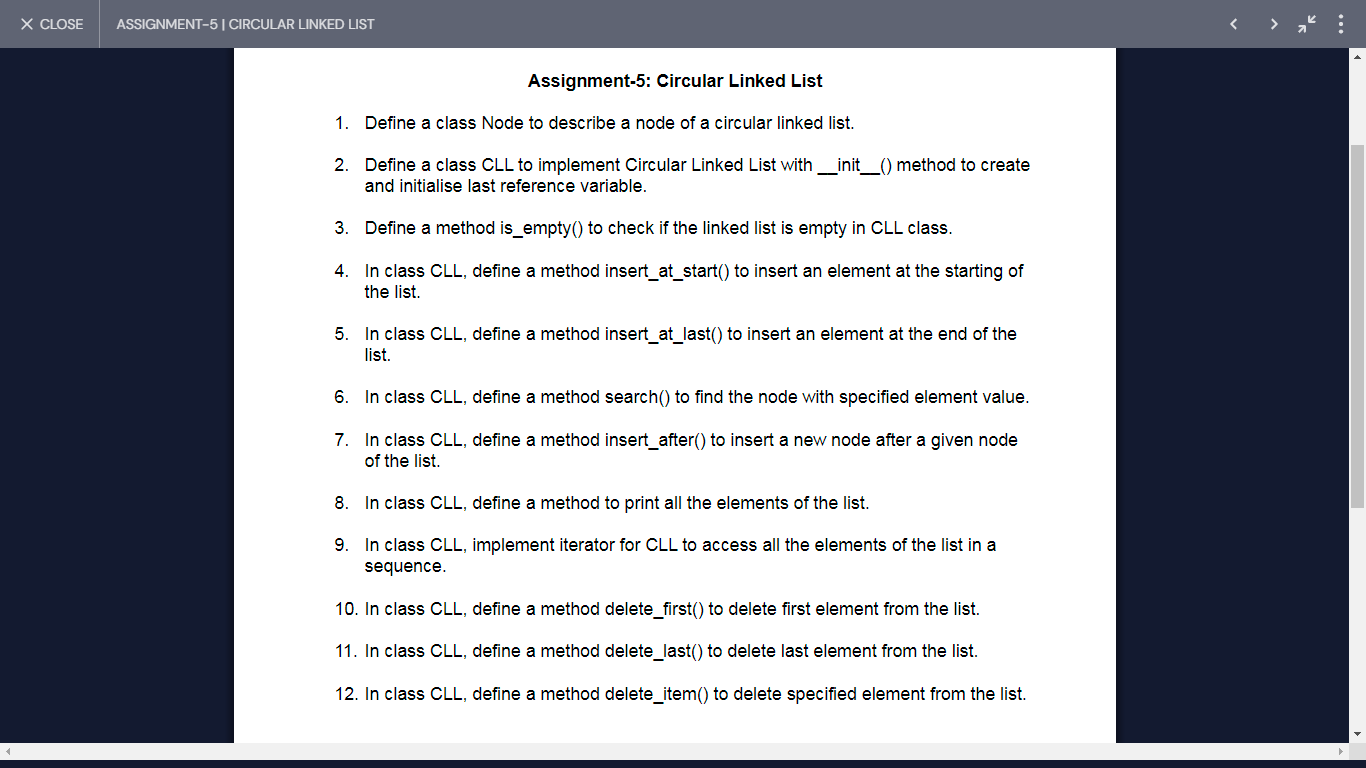
mylist.delete\_last()

mylist.delete\_first()

for x in mylist:

print(x, end=' ')

print()



=================================================================# q1

class Node:

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

# q2

class CLL:

def \_\_init\_\_(self, last=None):

self.last = last

# q3

def is\_empty(self):

return self.last == None

# q4

def insert\_at\_start(self, data):

n = Node(data)

if self.is\_empty():

n.next = n

self.last = n

else:

n.next = self.last.next

self.last.next = n

# q5

def insert\_at\_last(self, data):

n = Node(data)

if self.is\_empty():

n.next = n

self.last = n

else:

n.next = self.last.next

self.last.next = n

self.last = n

# q6

def search(self, data):

if self.is\_empty():

return None

temp = self.last.next

while temp != self.last:

if temp.item == data:

return temp

temp = temp.next

if temp.item == data:

return temp

return None

# q7

def insert\_after(self, temp, data):

if temp is not None:

n = Node(data, temp.next)

temp.next = n

if temp == self.last:

self.last = n

# q8

def print\_list(self):

if not self.is\_empty():

temp = self.last.next

while temp != self.last:

print(temp.item, end=' ')

temp = temp.next

print(temp.item)

# q10

def delete\_first(self):

if not self.is\_empty():

if self.last.next == self.last:

self.last = None

else:

self.last.next = self.last.next.next

# q11

def delete\_last(self):

if not self.is\_empty():

if self.last.next == self.last:

self.last = None

else:

temp = self.last.next

while temp.next != self.last:

temp = temp.next

temp.next = self.last.next

self.last = temp

# q12

def delete\_item(self, data):

if not self.is\_empty():

if self.last.next == self.last:

if self.last.item == data:

self.last = None

else:

if self.last.next.item == data:

self.delete\_first()

else:

temp = self.last.next

while temp != self.last:

if temp.next == self.last and self.last.item == data:

self.delete\_last()

break

if temp.next.item == data:

temp.next = temp.next.next

break

temp = temp.next

# q9

def \_\_iter\_\_(self):

if self.last == None:

return CLLIterator(None)

else:

return CLLIterator(self.last.next)

class CLLIterator:

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

self.current = start

self.start = start

self.count = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current == None:

raise StopIteration

if self.current == self.start and self.count == 1:

raise StopIteration

else:

self.count = 1

data = self.current.item

self.current = self.current.next

return data

# Driver code

cll = CLL()

cll.insert\_at\_start(10)

cll.insert\_at\_start(20)

cll.insert\_at\_last(30)

cll.insert\_at\_last(40)

cll.insert\_after(cll.search(10), 50) # 20 10 50 30 40

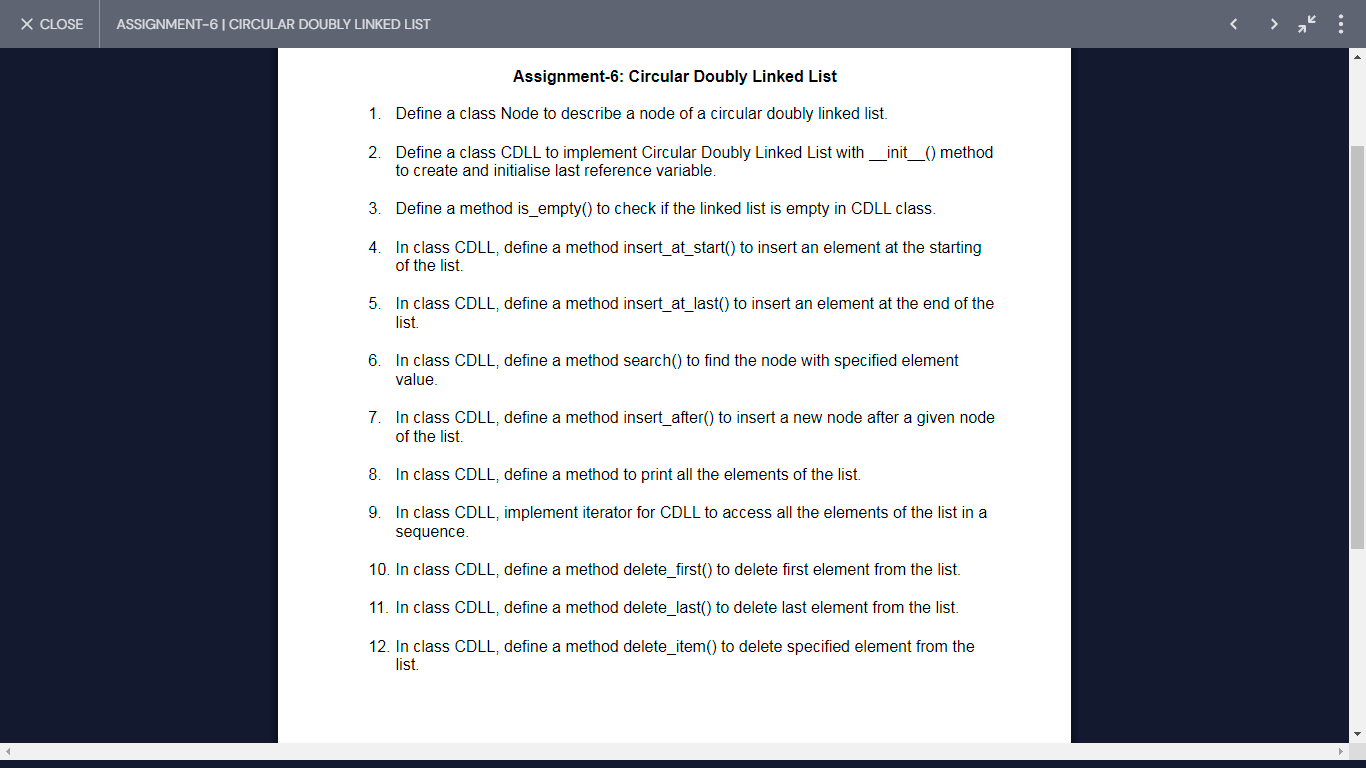
for x in cll:

print(x, end=' ')

print()

cll.print\_list()

=================================================================



=================================================================

class Node: # q1

def \_\_init\_\_(self, item=None, prev=None, next=None):

self.item = item

self.prev = prev

self.next = next

class CDLL: # q2

def \_\_init\_\_(self, start=None):

self.start = start

def is\_empty(self): # q3

return self.start is None

def insert\_at\_start(self, data): # q4

n = Node(data)

if self.is\_empty():

n.next = n

n.prev = n

self.start = n

else:

n.next = self.start

n.prev = self.start.prev

self.start.prev.next = n

self.start.prev = n

self.start = n

def insert\_at\_last(self, data): # q5

n = Node(data)

if self.is\_empty():

n.next = n

n.prev = n

self.start = n

else:

n.next = self.start

n.prev = self.start.prev

self.start.prev.next = n

self.start.prev = n

def search(self, data): # q6

temp = self.start

if temp is None:

return None

if temp.item == data:

return temp

else:

temp = temp.next

while temp != self.start:

if temp.item == data:

return temp

temp = temp.next

return None

def insert\_after(self, temp, data): # q7

if temp is not None:

n = Node(data)

n.next = temp.next

n.prev = temp

temp.next.prev = n

temp.next = n

def print\_list(self): # q8

temp = self.start

if temp is not None:

print(temp.item, end=' ')

temp = temp.next

while temp != self.start:

print(temp.item, end=' ')

temp = temp.next

print()

def delete\_first(self): # q10

if self.start is not None:

if self.start.next == self.start:

self.start = None

else:

self.start.prev.next = self.start.next

self.start.next.prev = self.start.prev

self.start = self.start.next

def delete\_last(self): # q11

if self.start is not None:

if self.start.next == self.start:

self.start = None

else:

self.start.prev.prev.next = self.start

self.start.prev = self.start.prev.prev

def delete\_item(self, data): # q12

if self.start is not None:

temp = self.start

if temp.item == data:

self.delete\_first()

else:

temp = temp.next

while temp != self.start:

if temp.item == data:

temp.next.prev = temp.prev

temp.prev.next = temp.next

break

temp = temp.next

def \_\_iter\_\_(self): # q9

return CDLLIterator(self.start)

class CDLLIterator:

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

self.current = start

self.start = start

self.count = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current is None:

raise StopIteration

if self.current == self.start and self.count == 1:

raise StopIteration

else:

self.count = 1

data = self.current.item

self.current = self.current.next

return data

# Testing the CDLL

mylist = CDLL()

mylist.insert\_at\_start(10)

mylist.insert\_at\_start(5)

mylist.insert\_at\_last(20)

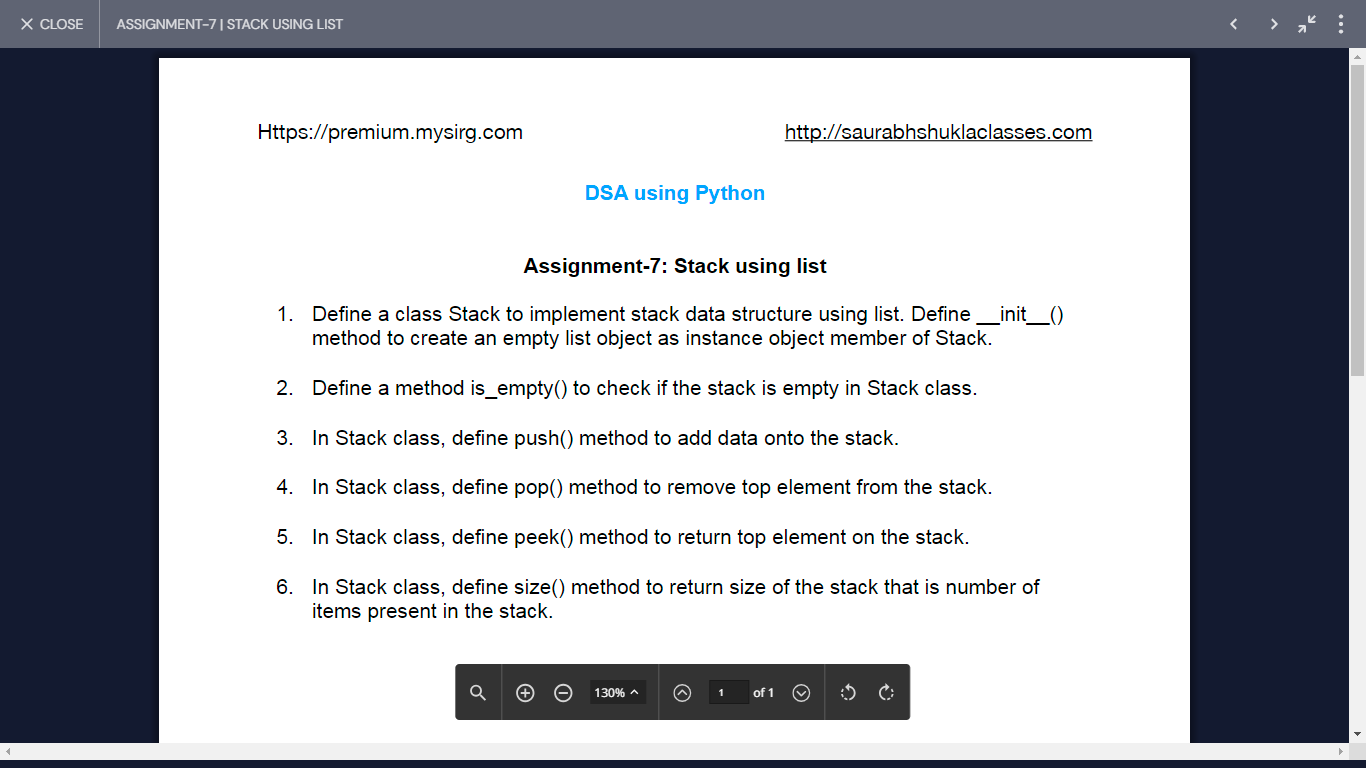
mylist.insert\_at\_last(30)

mylist.insert\_at\_last(40)

mylist.insert\_after(mylist.search(30), 35)

for x in mylist:

print(x, end=' ')



class Stack: # Q1

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

self.items = []

def is\_empty(self): # Q2

return len(self.items) == 0

def push(self, data): # Q3

self.items.append(data)

def pop(self): # Q4

if not self.is\_empty():

return self.items.pop()

else:

raise IndexError("Stack is Empty")

def peek(self): # Q5

if not self.is\_empty():

return self.items[-1]

else:

raise IndexError("Stack is Empty")

def size(self): # Q6

return len(self.items)

# Testing the Stack

s1 = Stack()

s1.push(10)

s1.push(20)

s1.push(40)

print("Top element is", s1.peek())

print("Removed element is", s1.pop())

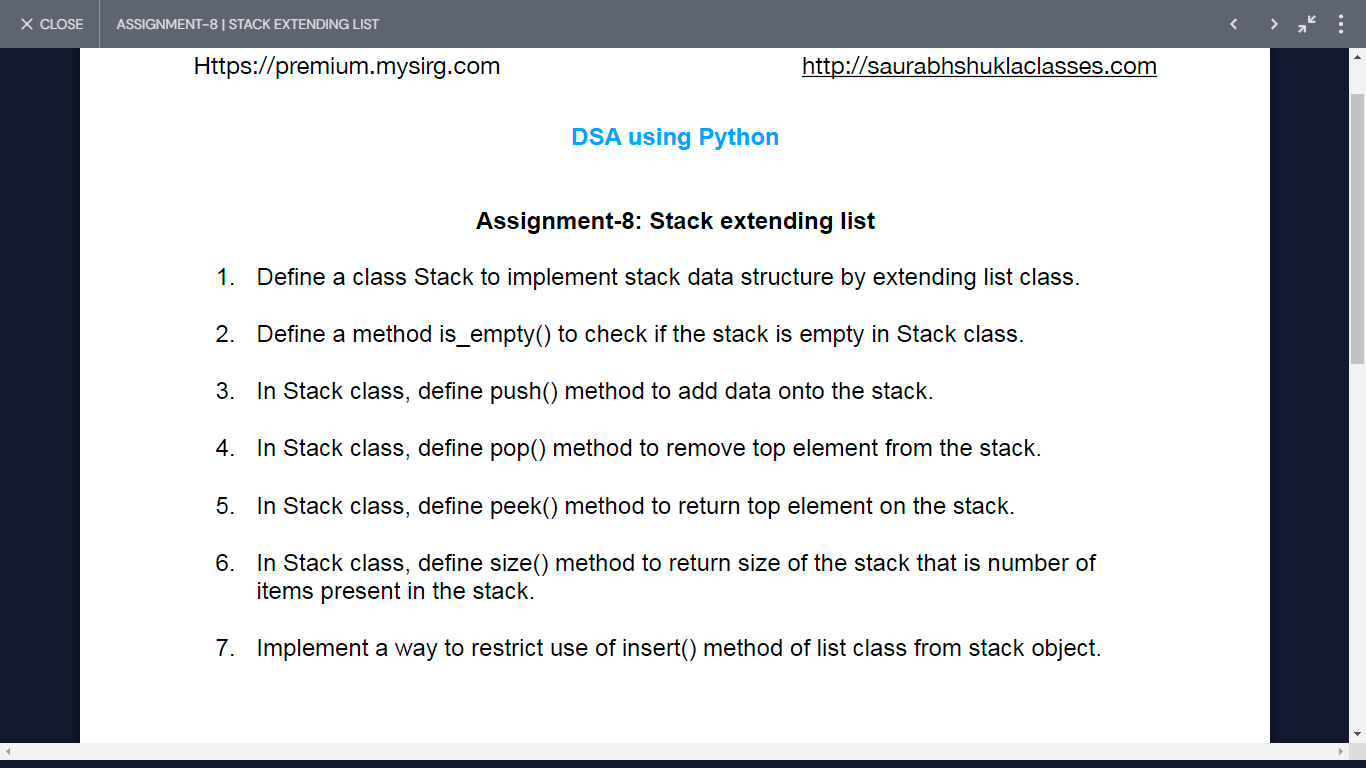
print("Top element after pop is", s1.peek())

OutPut:

Top element is 40

Removed element is 40

Top element after pop is 20



# Assignment: Stack extending list Solution:8

class Stack(list): # q1

def is\_empty(self): # q2

return len(self) == 0

def push(self, data): # q3

self.append(data)

def pop(self): # q4

if not self.is\_empty():

return super().pop()

else:

raise IndexError("Stack is Empty")

def peek(self): # q5

if not self.is\_empty():

return self[-1]

else:

raise IndexError("Stack is Empty")

def size(self): # q6

return len(self)

def insert(self, index, data):

raise AttributeError("No attribute 'insert' in Stack")

# Demo example

s1 = Stack()

s1.push(10)

s1.push(20)

s1.push(40)

print("Top value =", s1.peek()) # Should print the top value of the stack

print("Size of stack =", s1.size()) # Should print the size of the stack

print("Pop value =", s1.pop()) # Should remove and print the top value

print("Top value after pop =", s1.peek()) # Should print the new top value after pop

print("Size of stack after pop =", s1.size()) # Should print the size of the stack after pop

Output:

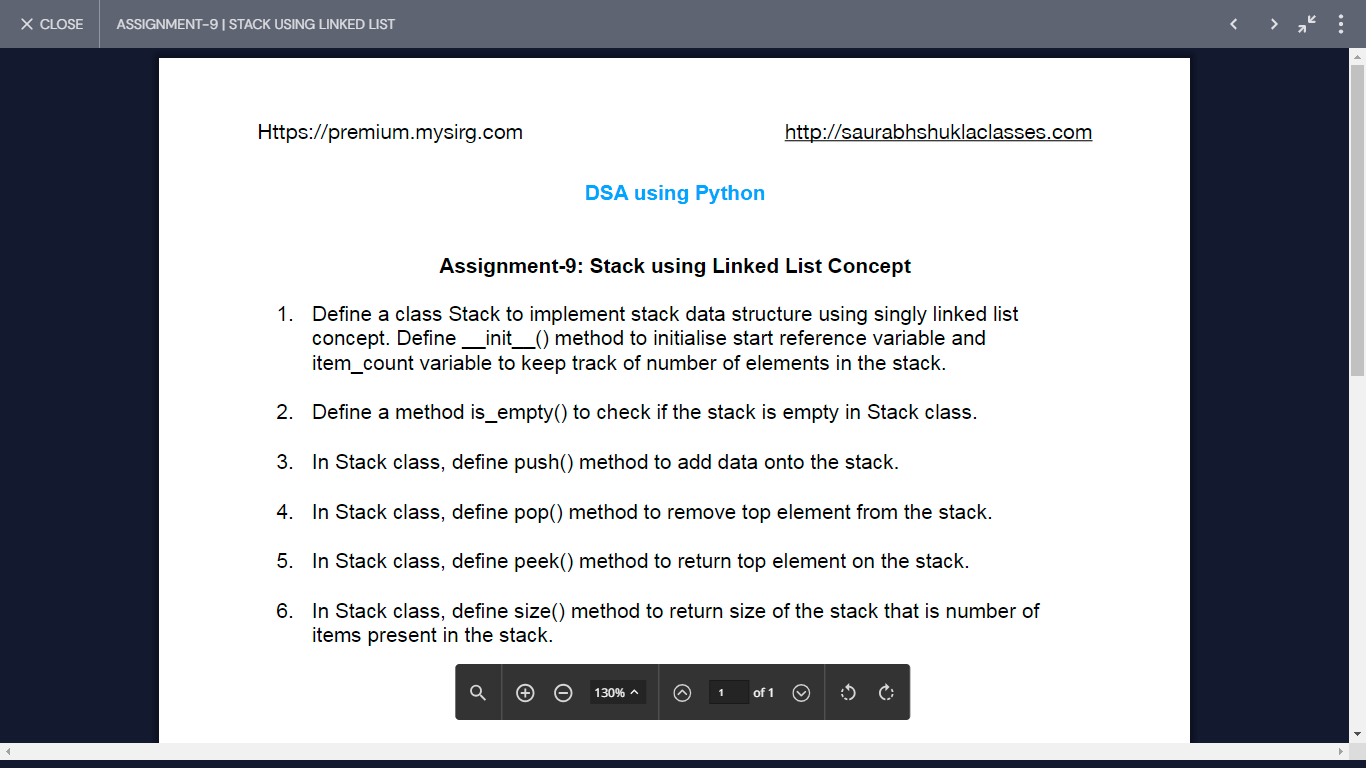
Top value = 40

Size of stack = 3

Pop value = 40

Top value after pop = 20

Size of stack after pop = 2



# Assignment 09 (Stack implementation using SLL)

class Node: # q1

def \_\_init\_\_(self, item=None, next=None):

self.item = item

self.next = next

class Stack:

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

self.start = None

self.item\_count = 0

def is\_empty(self): # q2

return self.start is None

def push(self, data): # q3

n = Node(data, self.start)

self.start = n

self.item\_count += 1

def pop(self): # q4

if not self.is\_empty():

data = self.start.item

self.start = self.start.next

self.item\_count -= 1

return data

else:

raise IndexError("Stack is Empty.")

def peek(self): # q5

if not self.is\_empty():

return self.start.item

else:

raise IndexError("Stack is Empty.")

def size(self): # q6

return self.item\_count

# Demo example

s1 = Stack()

s1.push(10)

s1.push(20)

s1.push(40)

print("Total elements in the stack =", s1.size())

print("Top element on the stack is", s1.peek())

print("Popped element is", s1.pop())

print("Total elements in the stack =", s1.size())

print("Top element on the stack is", s1.peek())

Output:

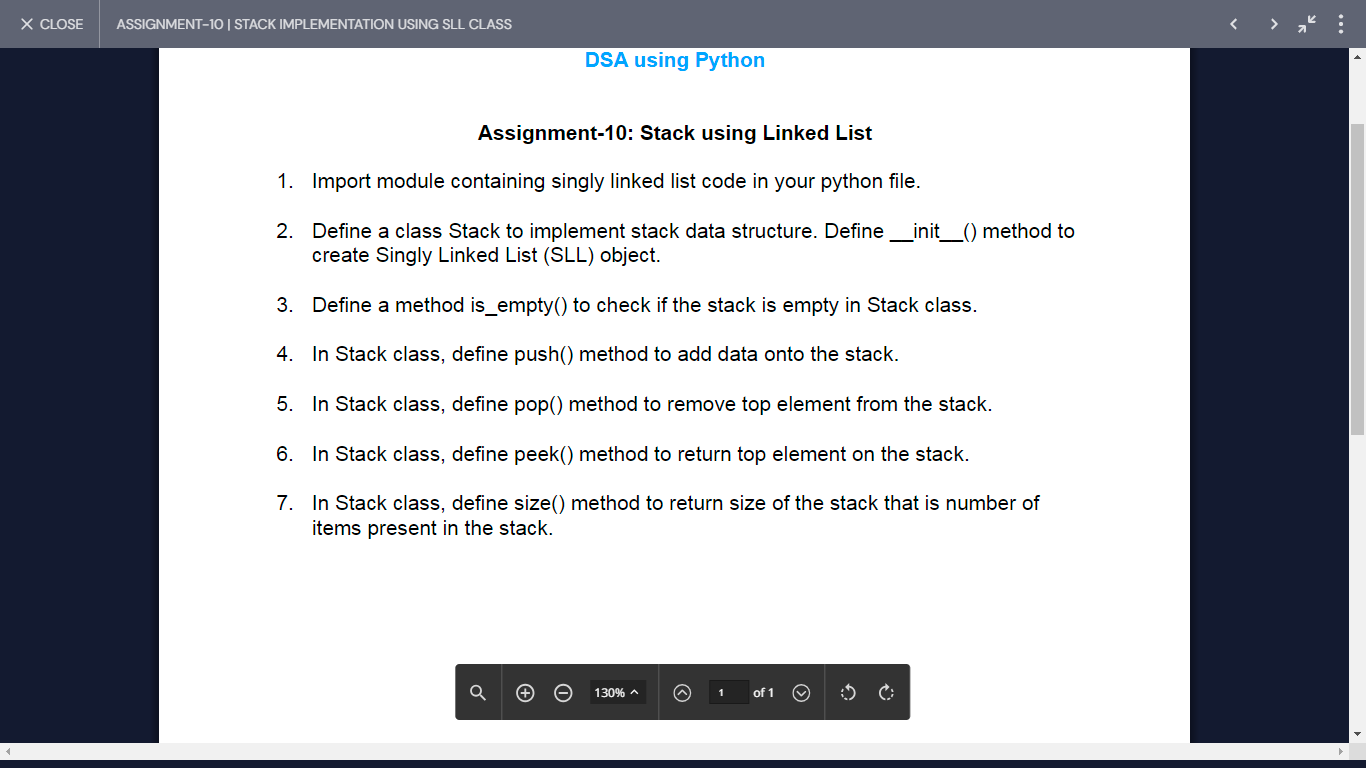
Total elements in the stack = 3

Top element on the stack is 40

Popped element is 40

Total elements in the stack = 2

Top element on the stack is 20



from Assign3 import \* # q1

class Stack: # q2

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

self.mylist = SLL()

self.item\_count = 0

def is\_empty(self): # q3

return self.mylist.is\_empty()

def push(self, data): # q4

self.mylist.insert\_at\_start(data)

self.item\_count += 1

def pop(self): # q5

if not self.is\_empty():

self.mylist.delete\_first()

self.item\_count -= 1

else:

raise IndexError("Stack is Empty.")

def peek(self): # q6

if not self.is\_empty():

return self.mylist.start.item

else:

raise IndexError("Stack is Empty.")

def size(self): # q7

return self.item\_count

# Demo example

s = Stack()

s.push(10)

s.push(20)

s.push(30)

print("Top element is", s.peek())

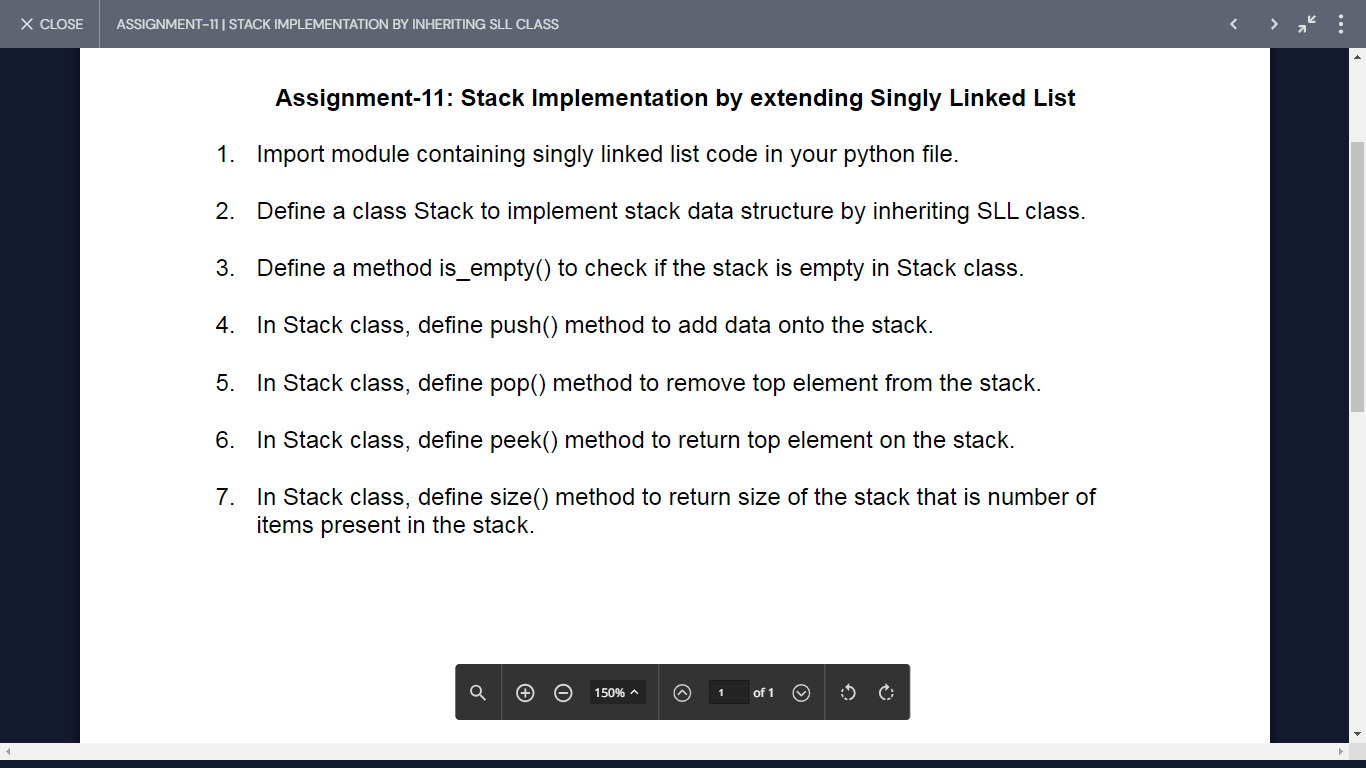
s.pop()

print("Top element is", s.peek())

Output:

Top element is 30

Top element is 20



from Assign3 import \* # q1

class Stack(SLL): # q2

def \_\_init\_\_(self): # Initialize inherited SLL

super().\_\_init\_\_()

self.item\_count = 0

def is\_empty(self): # q3

return super().is\_empty()

def push(self, data): # q4

self.insert\_at\_start(data)

self.item\_count += 1

def pop(self): # q5

if not self.is\_empty():

self.delete\_first()

self.item\_count -= 1

else:

raise IndexError("Stack underflow")

def peek(self): # q6

if not self.is\_empty():

return self.start.item

else:

raise IndexError("Stack underflow")

def size(self): # q7

return self.item\_count

# Demo example

s1 = Stack()

s1.push(10)

s1.push(20)

s1.push(40)

print("Top element on the stack:", s1.peek())

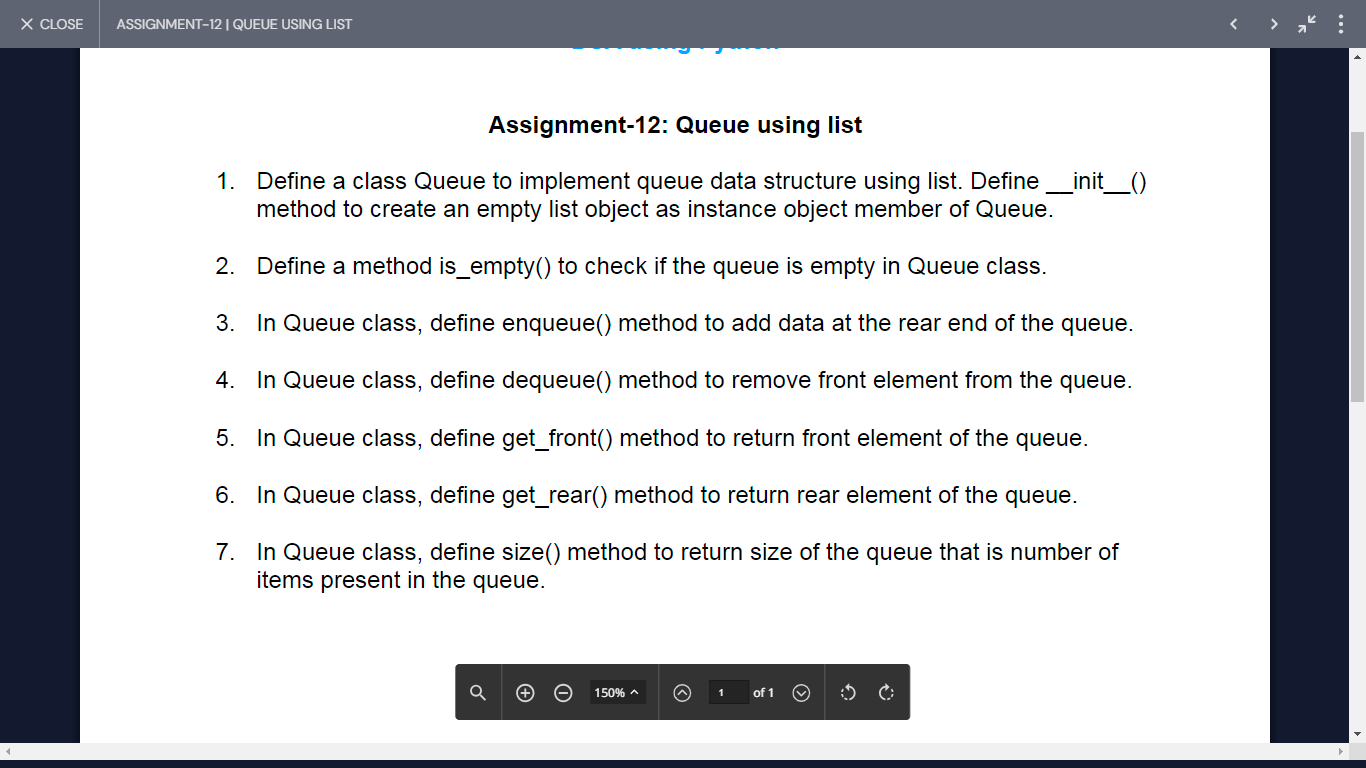
s1.pop()

print("Top element on the stack:", s1.peek())

Output:

Top element on the stack: 40

Top element on the stack: 20



from Assign3 import \* # q1

class Stack(SLL): # q2

def \_\_init\_\_(self): # Initialize inherited SLL

super().\_\_init\_\_()

self.item\_count = 0

def is\_empty(self): # q3

return super().is\_empty()

def push(self, data): # q4

self.insert\_at\_start(data)

self.item\_count += 1

def pop(self): # q5

if not self.is\_empty():

self.delete\_first()

self.item\_count -= 1

else:

raise IndexError("Stack underflow")

def peek(self): # q6

if not self.is\_empty():

return self.start.item

else:

raise IndexError("Stack underflow")

def size(self): # q7

return self.item\_count

# Demo example

s1 = Stack()

s1.push(10)

s1.push(20)

s1.push(40)

print("Top element on the stack:", s1.peek())

s1.pop()

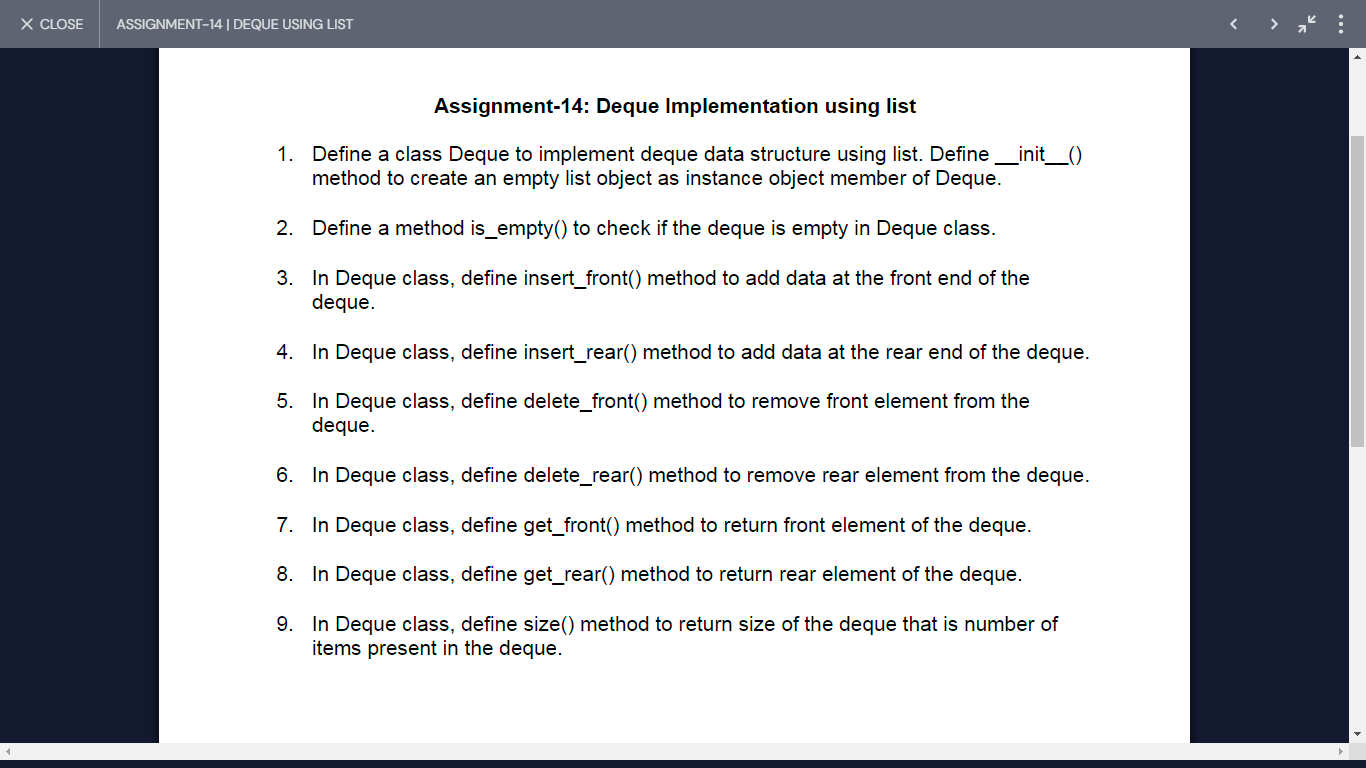
print("Top element on the stack:", s1.peek())

Output:

Queue Underflow

Front = 10 Rear = 40

Queue has now 3 elements



class Deque: # q1

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

self.items = []

self.item\_count = 0

def is\_empty(self): # q2

return len(self.items) == 0

def insert\_front(self, data): # q3

self.items.insert(0, data)

self.item\_count += 1

def insert\_rear(self, data): # q4

self.items.append(data)

self.item\_count += 1

def delete\_front(self): # q5

if not self.is\_empty():

self.items.pop(0)

self.item\_count -= 1

else:

raise IndexError("Deque Underflow")

def delete\_rear(self): # q6

if not self.is\_empty():

self.items.pop()

self.item\_count -= 1

else:

raise IndexError("Deque Underflow")

def get\_front(self): # q7

if not self.is\_empty():

return self.items[0]

else:

raise IndexError("Deque Underflow")

def get\_rear(self): # q8

if not self.is\_empty():

return self.items[-1]

else:

raise IndexError("Deque Underflow")

def size(self): # q9

return self.item\_count

# Demo example

d = Deque()

d.insert\_rear(10)

d.insert\_front(20)

print("Size after inserts:", d.size())

print("Rear element:", d.get\_rear())

print("Front element:", d.get\_front())

d.delete\_rear()

d.delete\_front()

print("Is deque empty?", d.is\_empty())

d.insert\_rear(10)

d.insert\_front(20)

d.insert\_rear(30)

d.insert\_front(40)

print("Rear element:", d.get\_rear())

print("Front element:", d.get\_front())

print("Size after more inserts:", d.size())

Output:

Size after inserts: 2

Rear element: 10

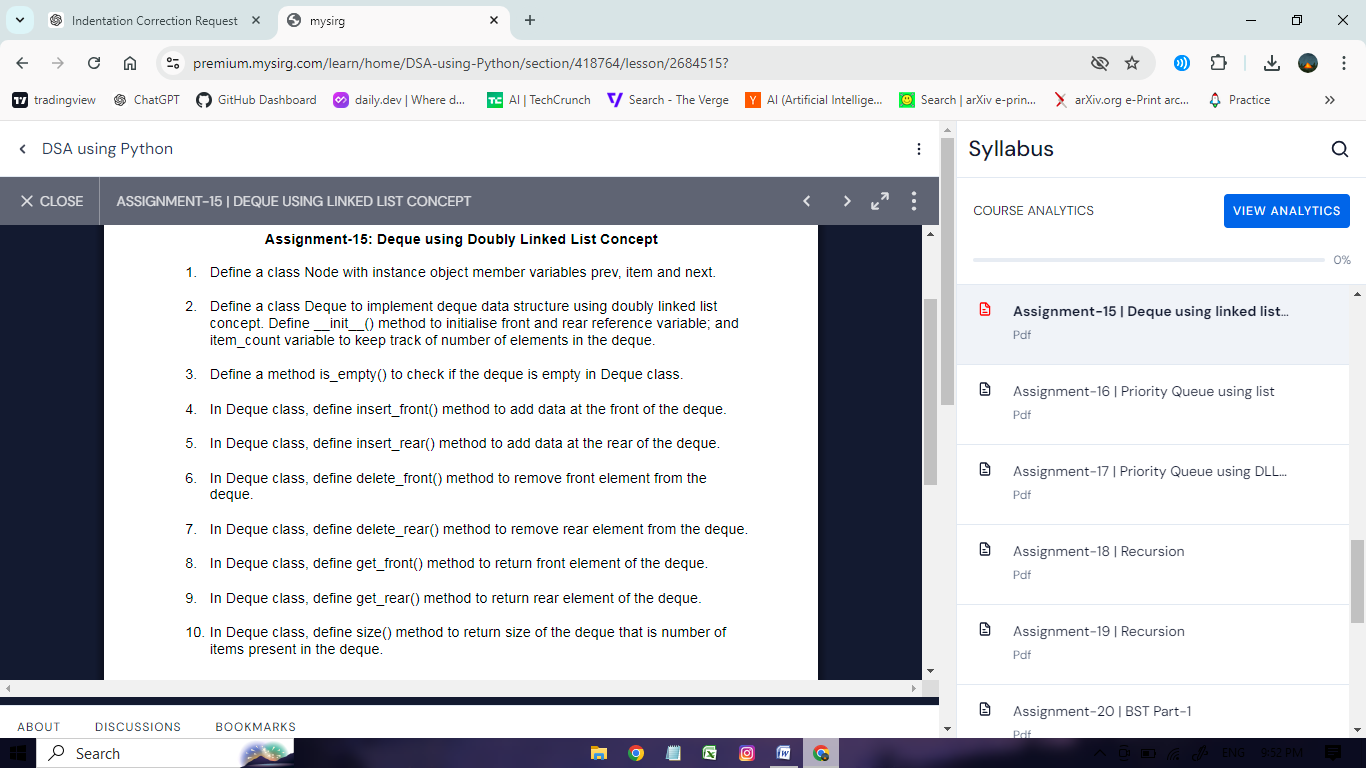
Front element: 20

Is deque empty? True

Rear element: 30

Front element: 40

Size after more inserts: 4



class Node: # q1

def \_\_init\_\_(self, item=None, prev=None, next=None):

self.prev = prev

self.item = item

self.next = next

class Deque: # q2

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

self.front = None

self.rear = None

self.item\_count = 0

def is\_empty(self): # q3

return self.front is None

def insert\_front(self, data): # q4

n = Node(data, None, self.front)

if self.is\_empty():

self.rear = n

else:

self.front.prev = n

self.front = n

self.item\_count += 1

def insert\_rear(self, data): # q5

n = Node(data, self.rear)

if self.is\_empty():

self.front = n

else:

self.rear.next = n

self.rear = n

self.item\_count += 1

def delete\_front(self): # q6

if not self.is\_empty():

if self.front == self.rear:

self.front = None

self.rear = None

else:

self.front = self.front.next

if self.front is not None:

self.front.prev = None

self.item\_count -= 1

def delete\_rear(self): # q7

if not self.is\_empty():

if self.front == self.rear:

self.front = None

self.rear = None

else:

self.rear = self.rear.prev

if self.rear is not None:

self.rear.next = None

self.item\_count -= 1

def get\_front(self): # q8

if not self.is\_empty():

return self.front.item

def get\_rear(self): # q9

if not self.is\_empty():

return self.rear.item

def size(self): # q10

return self.item\_count

# Demo example

d1 = Deque()

d1.insert\_front(10)

d1.insert\_front(20)

d1.insert\_rear(30)

d1.insert\_rear(40)

print("Size:", d1.size())

print("Front:", d1.get\_front())

print("Rear:", d1.get\_rear())

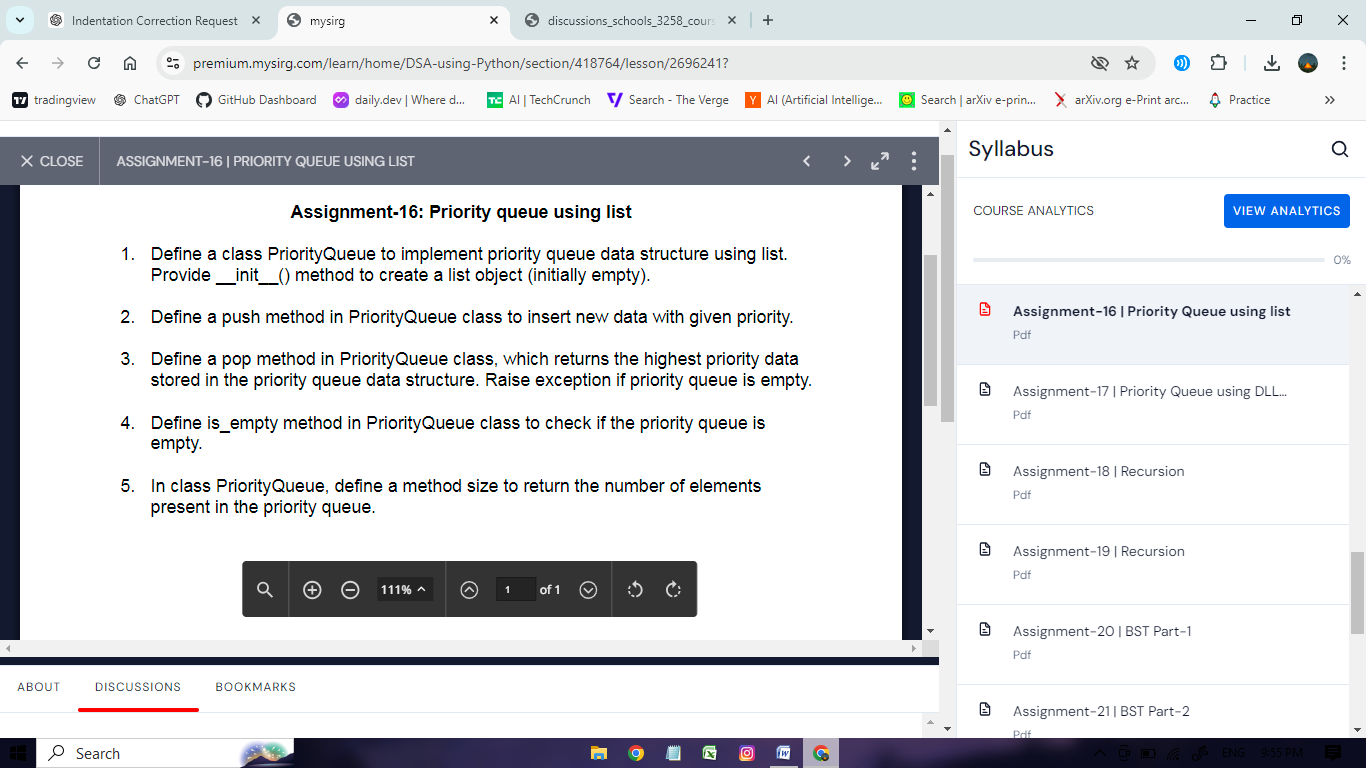
d1.delete\_rear()

d1.delete\_front()

print("Size after deletions:", d1.size())

print("Front after deletions:", d1.get\_front())

print("Rear after deletions:", d1.get\_rear())



class PriorityQueue: # q1

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

self.items = []

def push(self, data, priority): # q2

index = 0

while index < len(self.items) and self.items[index][1] <= priority:

index += 1

self.items.insert(index, (data, priority))

def is\_empty(self): # q4

return len(self.items) == 0

def pop(self): # q3

if self.is\_empty():

raise IndexError("Priority Queue is Empty.")

return self.items.pop(0)[0]

def size(self): # q5

return len(self.items)

# Demo example

p = PriorityQueue()

p.push("Shikhar", 4)

p.push("Arjun", 7)

p.push("Asmia", 2)

p.push("Harshit", 5)

p.push("Shiv", 1)

while not p.is\_empty():

print(p.pop())

Output:

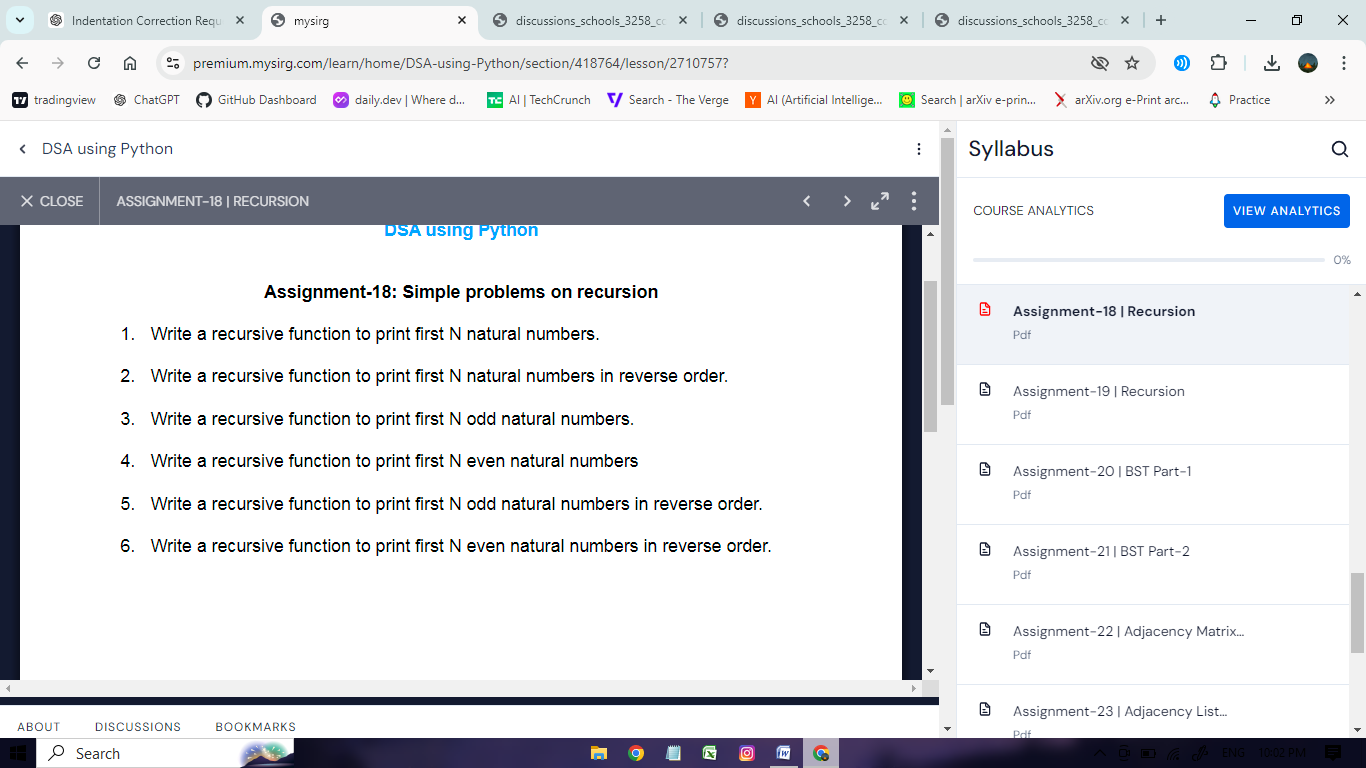
Arjun

Harshit

Shikhar

Asmia

Shiv



def printN(n): # q1

if n > 0:

printN(n - 1)

print(n, end=' ')

printN(10)

print()

def printNreverse(n): # q2

if n > 0:

print(n, end=' ')

printNreverse(n - 1)

printNreverse(10)

print()

def printNodd(n): # q3

if n > 0:

printNodd(n - 1)

print(2 \* n - 1, end=' ')

printNodd(10)

print()

def printNEven(n): # q4

if n > 0:

printNEven(n - 1)

print(2 \* n, end=' ')

printNEven(10)

print()

def printNoddreverse(n): # q5

if n > 0:

print(2 \* n - 1, end=' ')

printNoddreverse(n - 1)

printNoddreverse(10)

print()

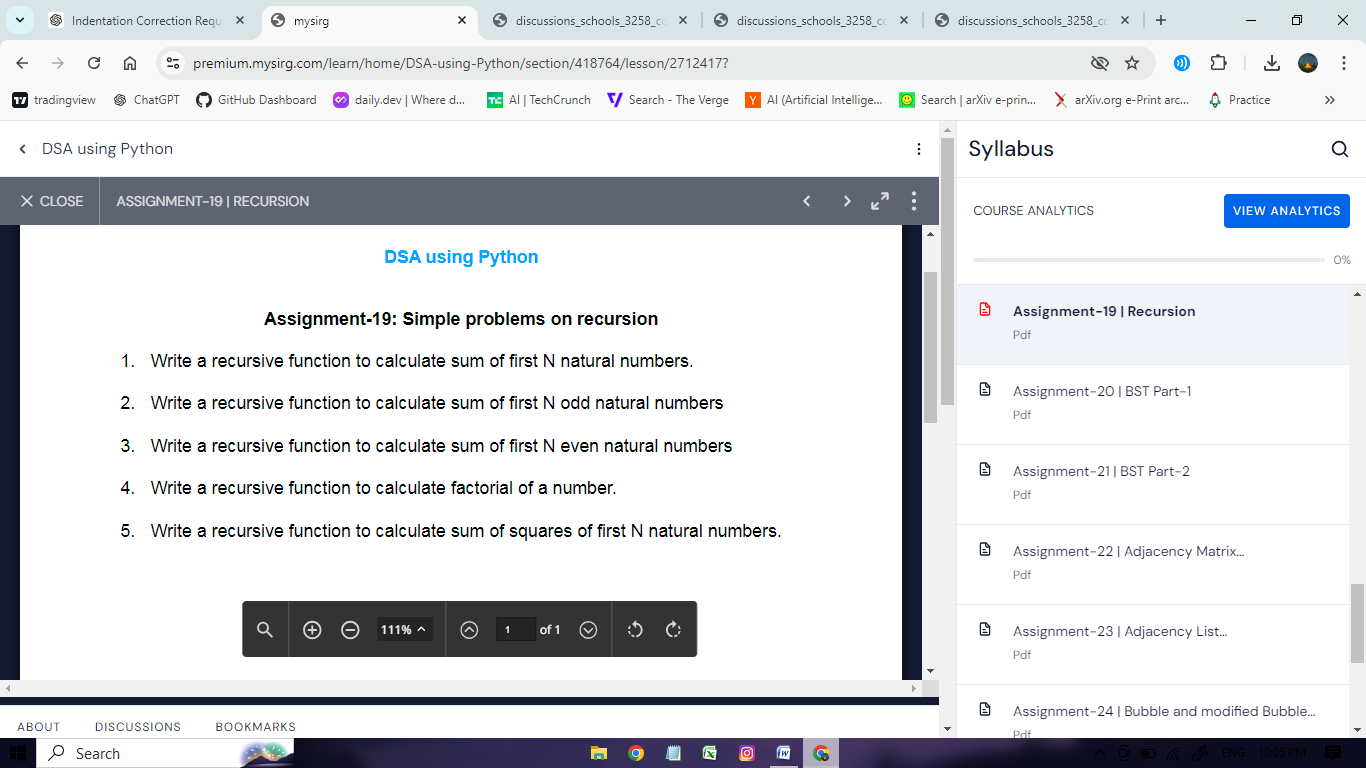
def printNEvenreverse(n): # q6

if n > 0:

print(2 \* n, end=' ')

printNEvenreverse(n - 1)

printNEvenreverse(10)



def SumN(n): # q1

if n == 1:

return 1

return n + SumN(n - 1)

print("Sum of first 10 natural numbers:", SumN(10))

def SumNodd(n): # q2

if n == 1:

return 1

return 2 \* n - 1 + SumNodd(n - 1)

print("Sum of first 10 odd numbers:", SumNodd(10))

def SumEven(n): # q3

if n == 1:

return 2

return 2 \* n + SumEven(n - 1)

print("Sum of first 10 even numbers:", SumEven(10))

def fact(n): # q4

if n == 0:

return 1

return n \* fact(n - 1)

print("Factorial of 5:", fact(5))

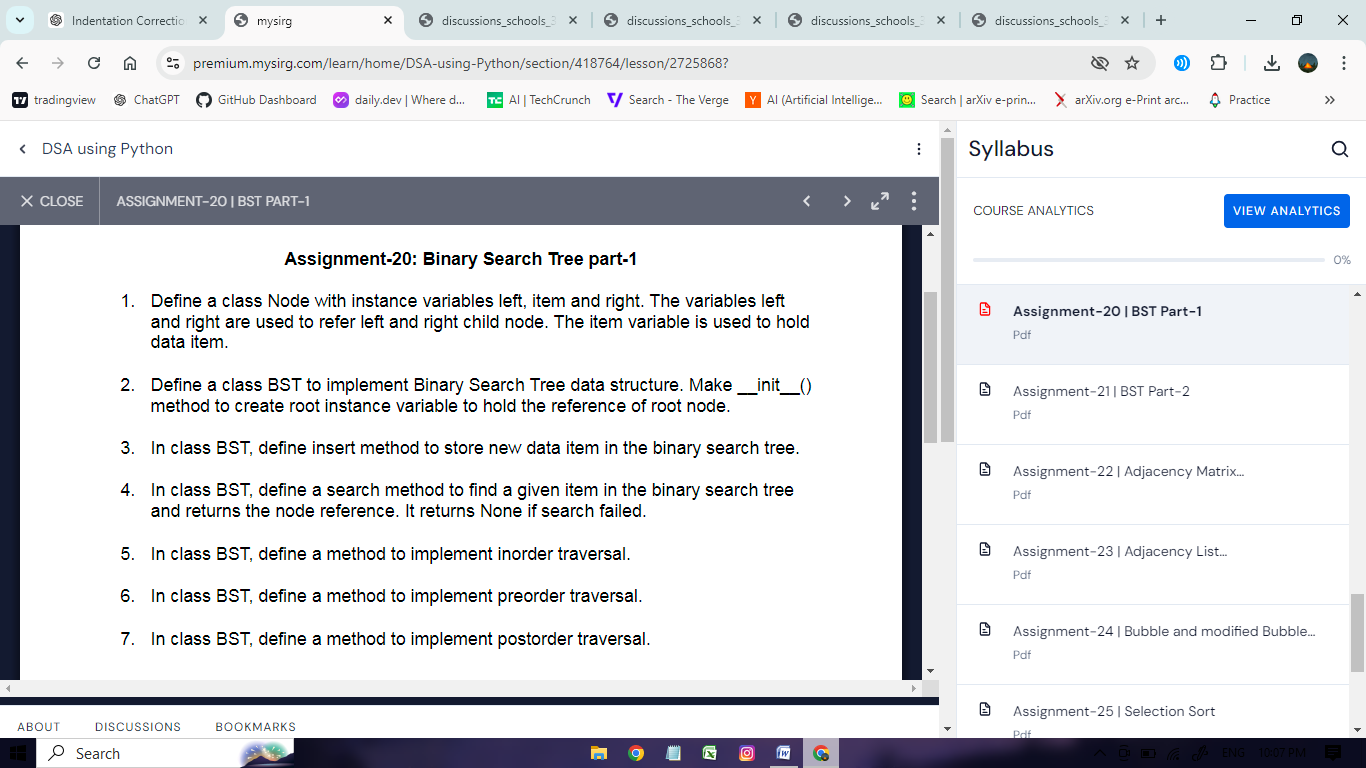
def SumNsquare(n): # q5

if n == 1:

return 1

return n \* n + SumNsquare(n - 1)

print("Sum of squares of first 5 natural numbers:", SumNsquare(5))



class Node: # q1

def \_\_init\_\_(self, item=None, left=None, right=None):

self.item = item

self.left = left

self.right = right

class BST: # q2

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

self.root = None

def insert(self, data): # Q3

self.root = self.rinsert(self.root, data)

def rinsert(self, root, data):

if root is None:

return Node(data)

if data < root.item:

root.left = self.rinsert(root.left, data)

elif data > root.item:

root.right = self.rinsert(root.right, data)

return root

def search(self, data): # q4

return self.rsearch(self.root, data)

def rsearch(self, root, data):

if root is None or root.item == data:

return root

if data < root.item:

return self.rsearch(root.left, data)

else:

return self.rsearch(root.right, data)

def inorder(self): # q5

result = []

self.rinorder(self.root, result)

return result

def rinorder(self, root, result):

if root:

self.rinorder(root.left, result)

result.append(root.item)

self.rinorder(root.right, result)

def preorder(self): # q6

result = []

self.rpreorder(self.root, result)

return result

def rpreorder(self, root, result):

if root:

result.append(root.item)

self.rpreorder(root.left, result)

self.rpreorder(root.right, result)

def postorder(self): # q7

result = []

self.rpostorder(self.root, result)

return result

def rpostorder(self, root, result):

if root:

self.rpostorder(root.left, result)

self.rpostorder(root.right, result)

result.append(root.item)

# Example usage

bst = BST()

bst.insert(10)

bst.insert(5)

bst.insert(20)

bst.insert(3)

bst.insert(7)

bst.insert(15)

bst.insert(25)

print("Inorder traversal:", bst.inorder())

print("Preorder traversal:", bst.preorder())

print("Postorder traversal:", bst.postorder())

# Search for a value

search\_value = 7

found\_node = bst.search(search\_value)

if found\_node:

print(f"Node with value {search\_value} found.")

else:

print(f"Node with value {search\_value} not found.")

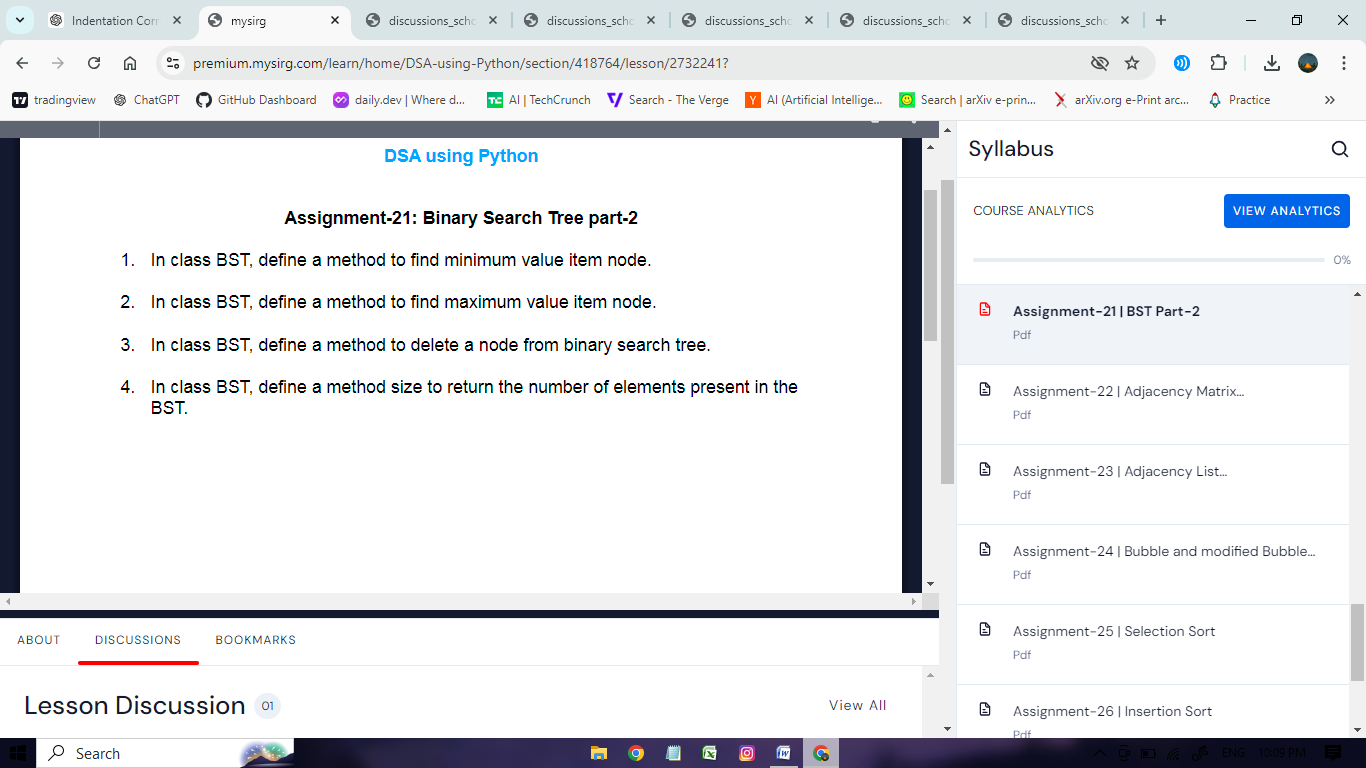
Output:

Inorder traversal: [3, 5, 7, 10, 15, 20, 25]

Preorder traversal: [10, 5, 3, 7, 20, 15, 25]

Postorder traversal: [3, 7, 5, 15, 25, 20, 10]

Node with value 7 found.



class Node:

def \_\_init\_\_(self, item=None, left=None, right=None):

self.item = item

self.left = left

self.right = right

class BST:

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

self.root = None

def insert(self, data):

self.root = self.rinsert(self.root, data)

def rinsert(self, root, data):

if root is None:

return Node(data)

if data < root.item:

root.left = self.rinsert(root.left, data)

elif data > root.item:

root.right = self.rinsert(root.right, data)

return root

def search(self, data):

return self.rsearch(self.root, data)

def rsearch(self, root, data):

if root is None or root.item == data:

return root

if data < root.item:

return self.rsearch(root.left, data)

else:

return self.rsearch(root.right, data)

def inorder(self):

result = []

self.rinorder(self.root, result)

return result

def rinorder(self, root, result):

if root:

self.rinorder(root.left, result)

result.append(root.item)

self.rinorder(root.right, result)

def preorder(self):

result = []

self.rpreorder(self.root, result)

return result

def rpreorder(self, root, result):

if root:

result.append(root.item)

self.rpreorder(root.left, result)

self.rpreorder(root.right, result)

def postorder(self):

result = []

self.rpostorder(self.root, result)

return result

def rpostorder(self, root, result):

if root:

self.rpostorder(root.left, result)

self.rpostorder(root.right, result)

result.append(root.item)

def min\_value(self, temp): # q1

current = temp

while current.left is not None:

current = current.left

return current.item

def max\_value(self, temp): # q2

current = temp

while current.right is not None:

current = current.right

return current.item

def delete(self, data): # q3

self.root = self.rdelete(self.root, data)

def rdelete(self, root, data):

if root is None:

return root

if data < root.item:

root.left = self.rdelete(root.left, data)

elif data > root.item:

root.right = self.rdelete(root.right, data)

else:

if root.left is None:

return root.right

elif root.right is None:

return root.left

root.item = self.min\_value(root.right)

root.right = self.rdelete(root.right, root.item)

return root

def size(self): # q4

return len(self.inorder())

# Example usage

bst = BST()

bst.insert(10)

bst.insert(5)

bst.insert(20)

bst.insert(3)

bst.insert(7)

bst.insert(15)

bst.insert(25)

print("Inorder traversal:", bst.inorder())

print("Preorder traversal:", bst.preorder())

print("Postorder traversal:", bst.postorder())

print("Minimum value:", bst.min\_value(bst.root))

print("Maximum value:", bst.max\_value(bst.root))

# Delete a node

bst.delete(20)

print("Inorder traversal after deleting 20:", bst.inorder())

print("Size of the BST:", bst.size())

Output:

Inorder traversal: [3, 5, 7, 10, 15, 20, 25]

Preorder traversal: [10, 5, 3, 7, 20, 15, 25]

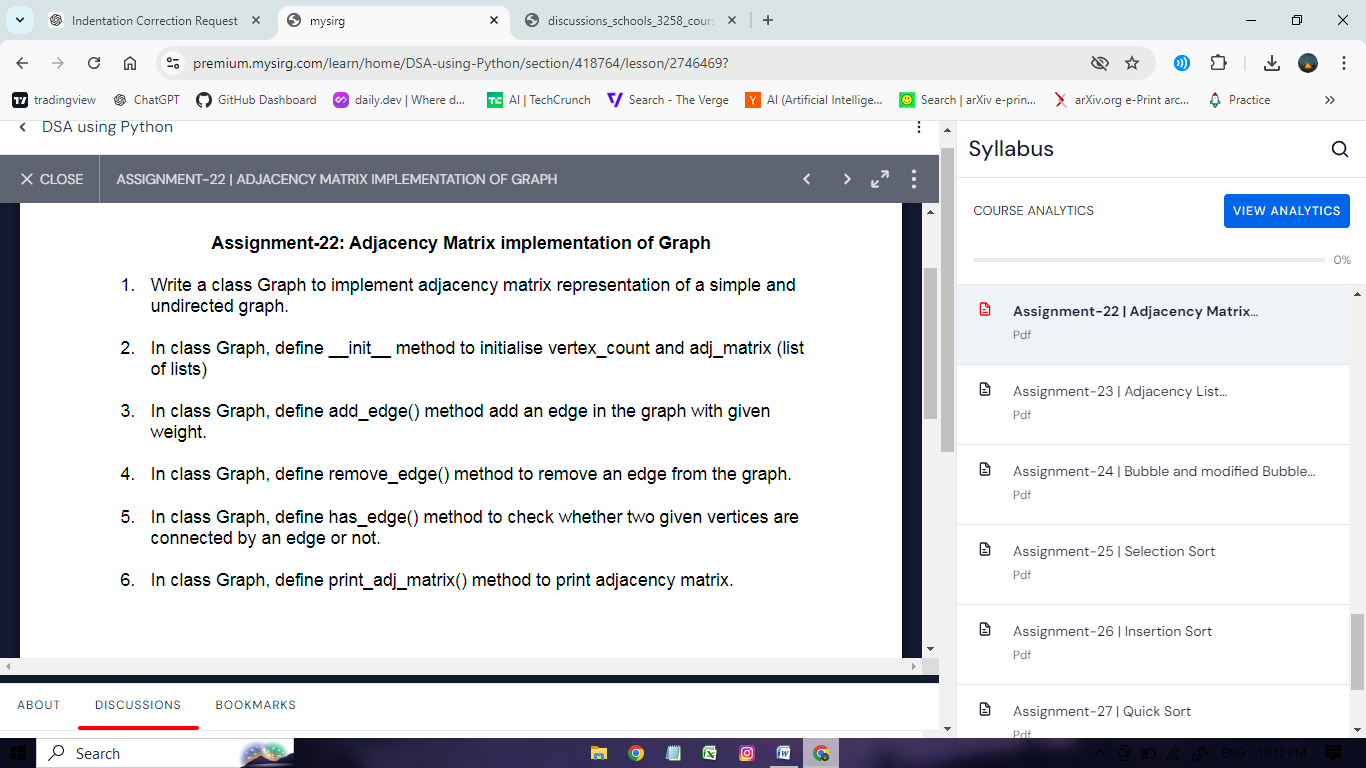
Postorder traversal: [3, 7, 5, 15, 25, 20, 10]

Minimum value: 3

Maximum value: 25

Inorder traversal after deleting 20: [3, 5, 7, 10, 15, 25]

Size of the BST: 6



class Graph:

def \_\_init\_\_(self, vno): # q2

self.vertex\_count = vno

self.adj\_matrix = [[0] \* vno for \_ in range(vno)]

def add\_edge(self, u, v, weight=1): # q3

if 0 <= u < self.vertex\_count and 0 <= v < self.vertex\_count:

self.adj\_matrix[u][v] = weight

self.adj\_matrix[v][u] = weight

else:

print("Invalid vertex")

def remove\_edge(self, u, v): # q4

if 0 <= u < self.vertex\_count and 0 <= v < self.vertex\_count:

self.adj\_matrix[u][v] = 0

self.adj\_matrix[v][u] = 0

else:

print("Invalid vertex")

def has\_edge(self, u, v): # q5

if 0 <= u < self.vertex\_count and 0 <= v < self.vertex\_count:

return self.adj\_matrix[u][v] != 0

else:

print("Invalid vertex")

return False

def print\_adj\_matrix(self): # q6

for row\_list in self.adj\_matrix:

print(" ".join(map(str, row\_list)))

# Example usage

g = Graph(5)

g.add\_edge(0, 1)

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 3)

g.add\_edge(3, 4)

g.print\_adj\_matrix()

Output:

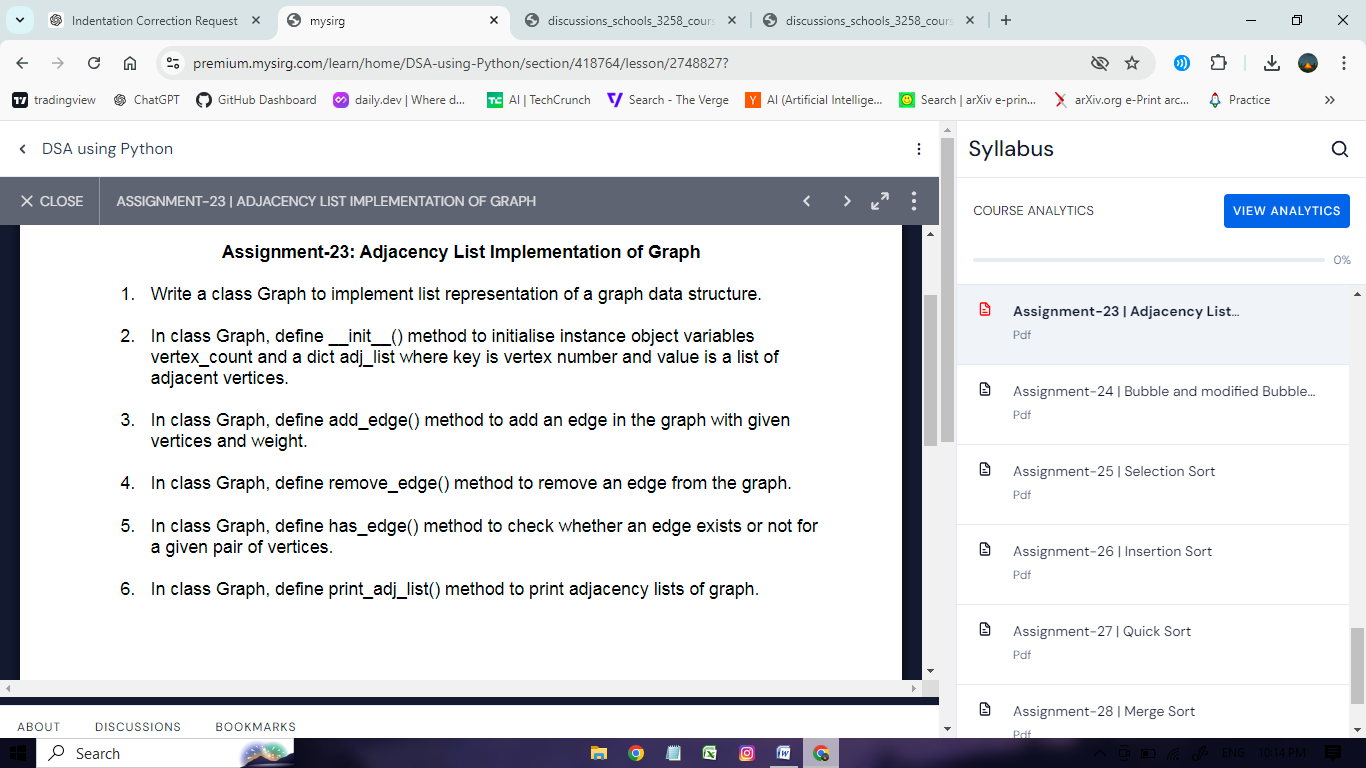
0 1 0 0 0

1 0 1 1 0

0 1 0 1 0

0 1 1 0 1

0 0 0 1 0



class Graph: # q1

def \_\_init\_\_(self, vno): # q2

self.vertex\_count = vno

self.adj\_list = {v: [] for v in range(vno)}

def add\_edge(self, u, v, weight=1): # q3

if 0 <= u < self.vertex\_count and 0 <= v < self.vertex\_count:

self.adj\_list[u].append((v, weight))

self.adj\_list[v].append((u, weight))

else:

print("Invalid vertices")

def remove\_edge(self, u, v): # q4

if 0 <= u < self.vertex\_count and 0 <= v < self.vertex\_count:

self.adj\_list[u] = [(vertex, weight) for vertex, weight in self.adj\_list[u] if vertex != v]

self.adj\_list[v] = [(vertex, weight) for vertex, weight in self.adj\_list[v] if vertex != u]

else:

print("Invalid vertices")

def has\_edge(self, u, v): # q5

if 0 <= u < self.vertex\_count and 0 <= v < self.vertex\_count:

return any(vertex == v for vertex, \_ in self.adj\_list[u])

else:

print("Invalid vertices")

return False

def print\_adj\_list(self): # q6

for vertex, neighbors in self.adj\_list.items():

print("V", vertex, ":", neighbors)

# Example usage

g = Graph(5)

g.add\_edge(0, 1)

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.add\_edge(3, 4)

g.print\_adj\_list()

Output:

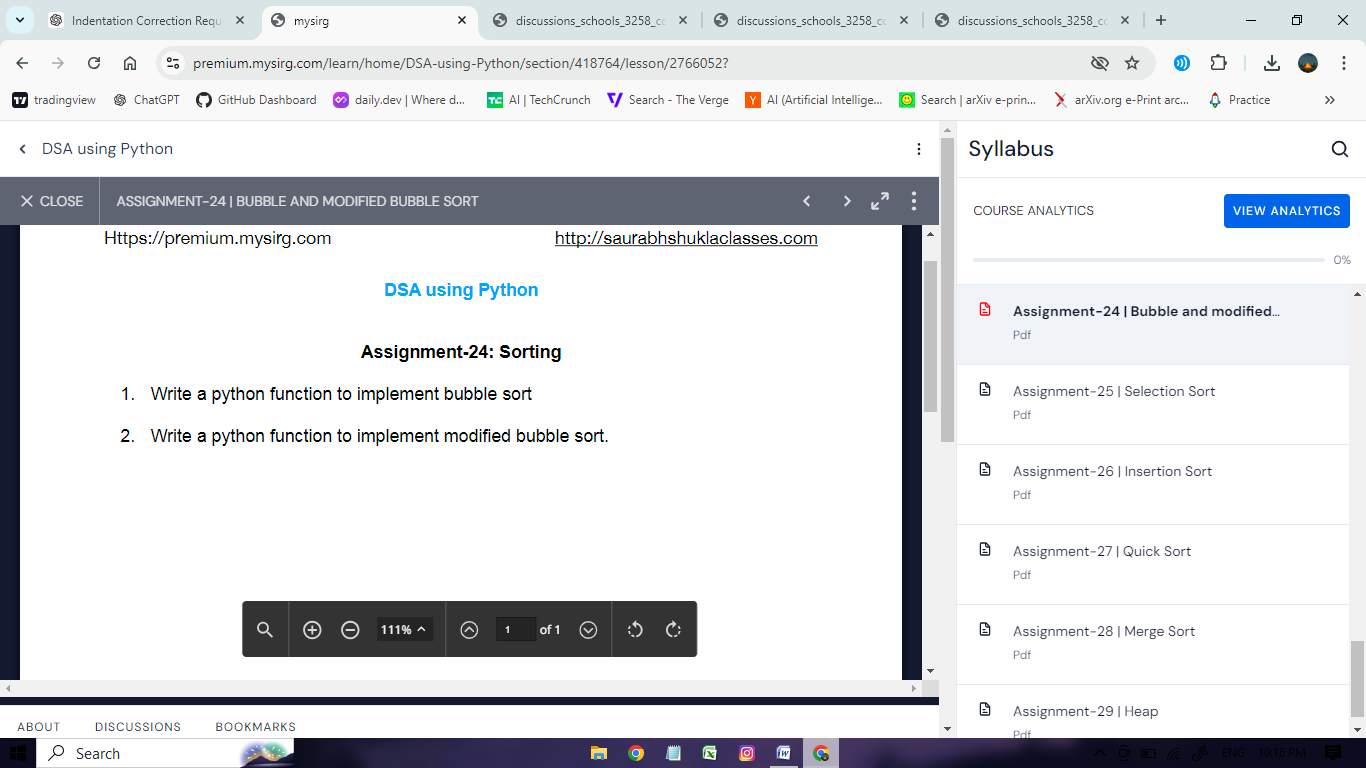
V 0 : [(1, 1)]

V 1 : [(0, 1), (2, 1), (3, 1)]

V 2 : [(1, 1), (4, 1)]

V 3 : [(1, 1), (4, 1)]

V 4 : [(2, 1), (3, 1)]



def bubble\_sort(data\_list): # q1

for r in range(1, len(data\_list)):

for i in range(len(data\_list) - r):

if data\_list[i] > data\_list[i + 1]:

data\_list[i], data\_list[i + 1] = data\_list[i + 1], data\_list[i]

l = [24, 67, 12, 89, 25, 50]

bubble\_sort(l)

print("Bubble Sort Result:", l)

def modified\_bubble\_sort(data\_list): # q2

for r in range(1, len(data\_list)):

flag = False

for i in range(len(data\_list) - r):

if data\_list[i] > data\_list[i + 1]:

data\_list[i], data\_list[i + 1] = data\_list[i + 1], data\_list[i]

flag = True

if not flag:

break

l = [24, 67, 12, 89, 25, 50]

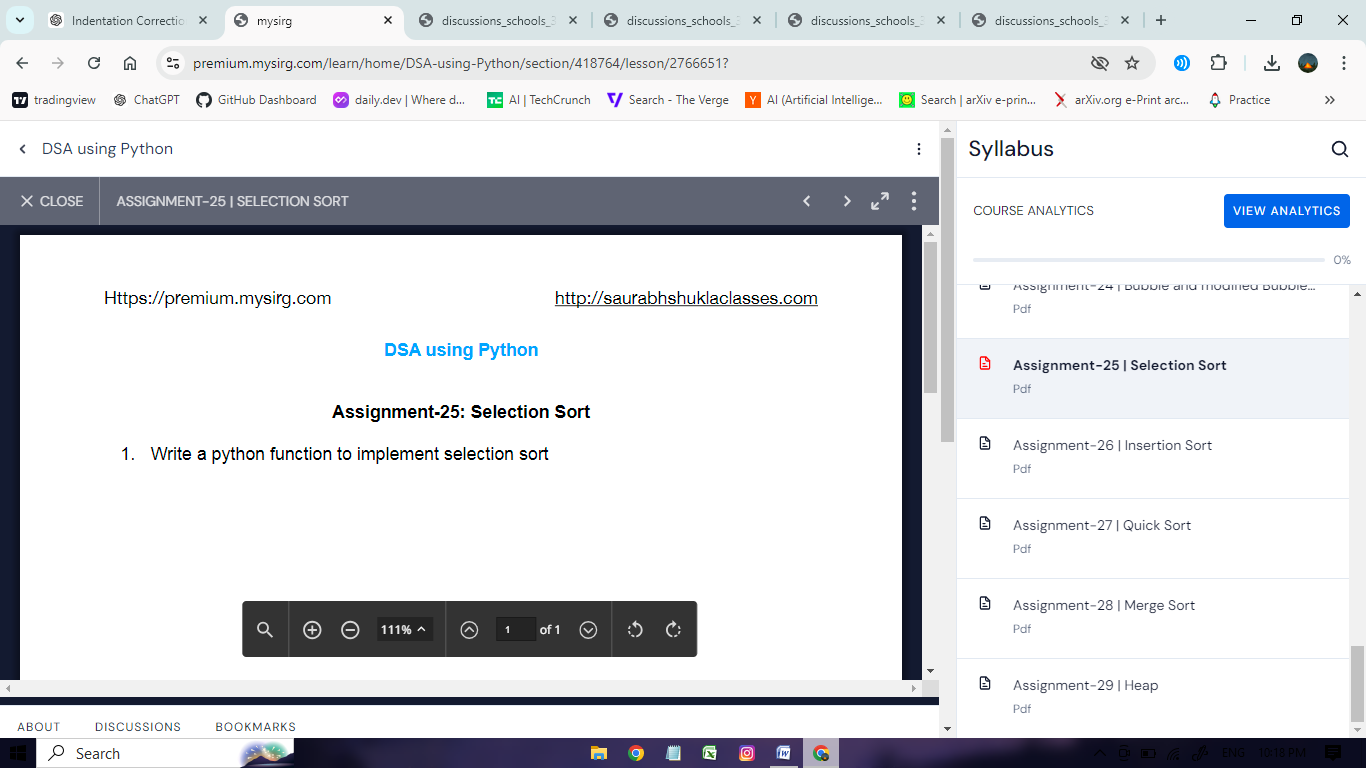
modified\_bubble\_sort(l)

print("Modified Bubble Sort Result:", l)

Output:

Bubble Sort Result: [12, 24, 25, 50, 67, 89]

Modified Bubble Sort Result: [12, 24, 25, 50, 67, 89]



def selection\_sort(list1):

n = len(list1)

for i in range(n - 1):

# Assume the current position is the minimum

min\_index = i

# Find the minimum element in the remaining unsorted portion

for j in range(i + 1, n):

if list1[j] < list1[min\_index]:

min\_index = j

# Swap the found minimum element with the element at the current position

list1[i], list1[min\_index] = list1[min\_index], list1[i]

# Test the selection sort function

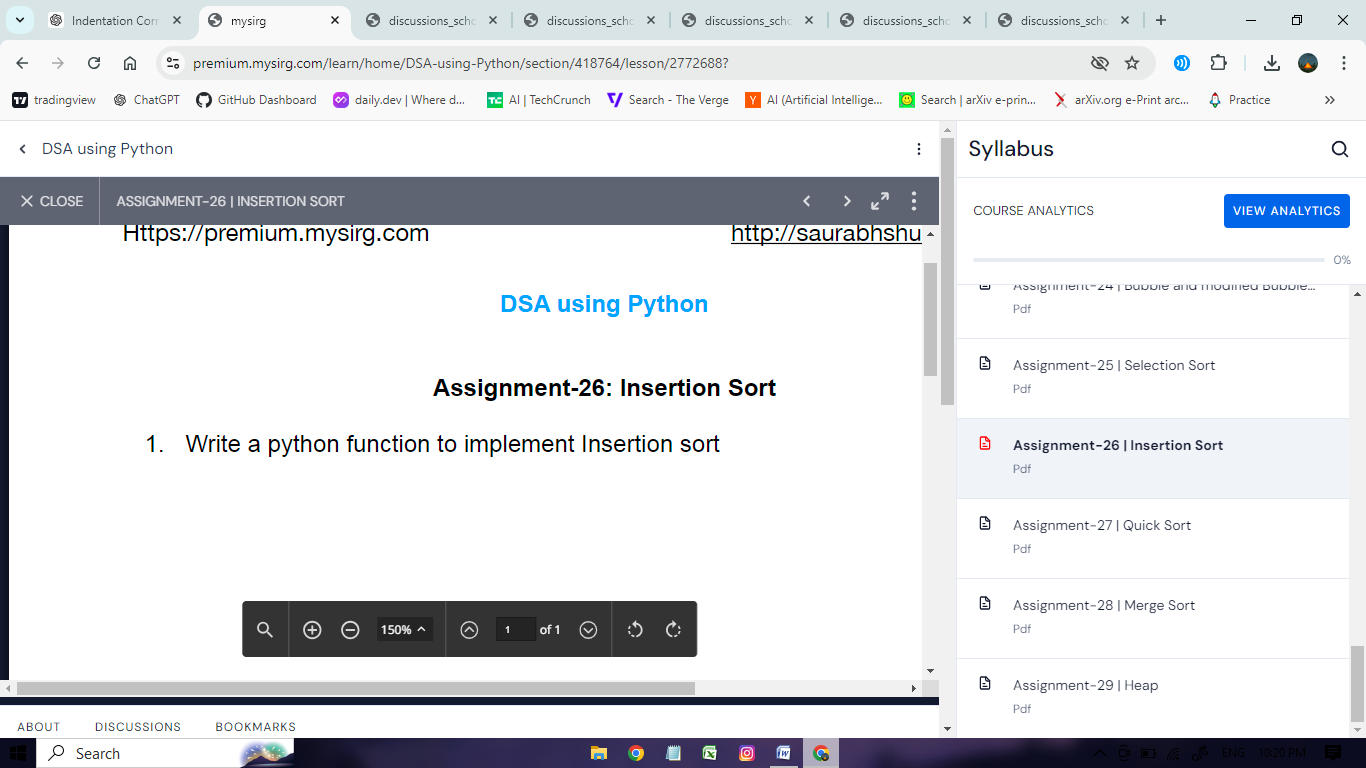
l1 = [64, 35, 89, 21, 72, 13]

selection\_sort(l1)

print("Sorted list:", l1)

Output:

Sorted list: [13, 21, 35, 64, 72, 89]



def insertion\_sort(list1):

# Iterate through each element in the list starting from the second element

for i in range(1, len(list1)):

temp = list1[i] # The element to be inserted

j = i - 1

# Shift elements of the sorted segment to the right as long as they are greater than temp

while j >= 0 and temp < list1[j]:

list1[j + 1] = list1[j]

j -= 1

# Place temp in its correct position

list1[j + 1] = temp

# Test the insertion sort function

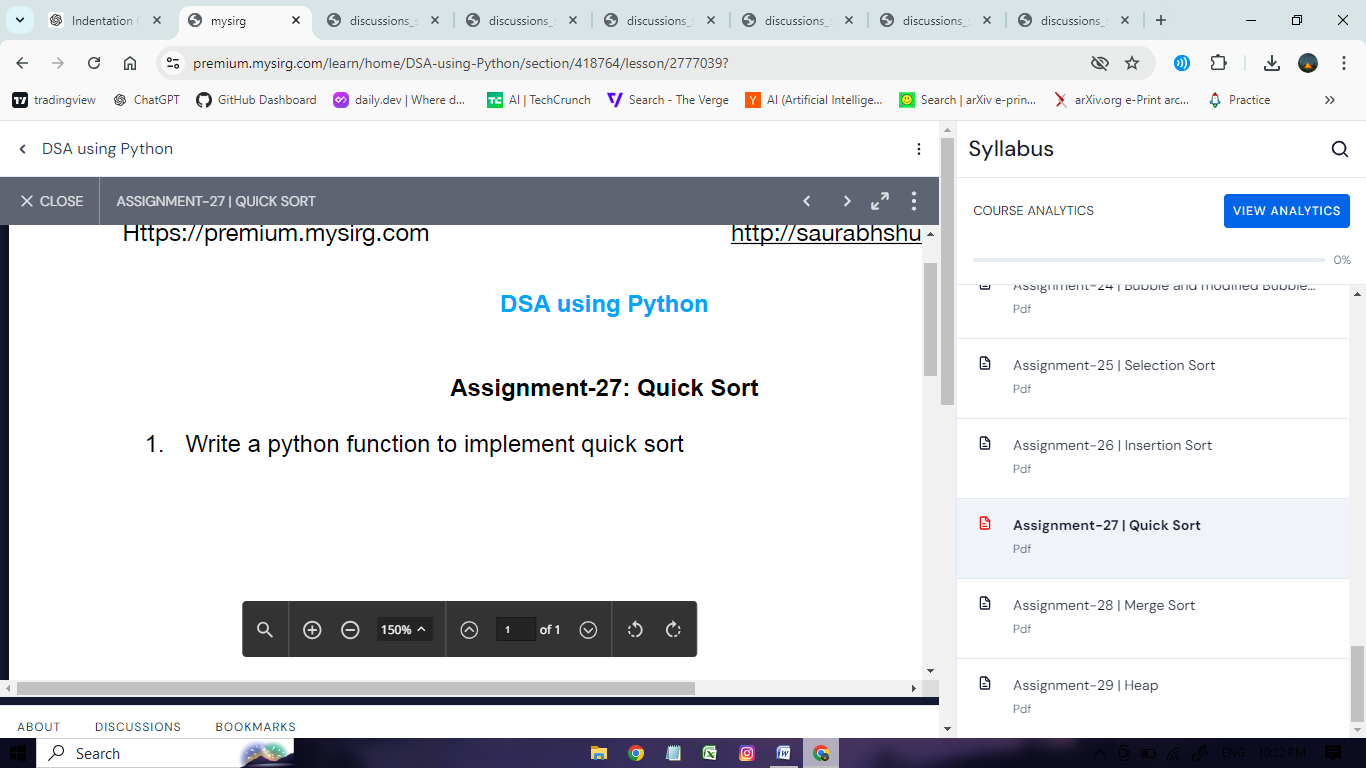
mylist = [25, 37, 11, 14, 60, 82, 18, 41]

insertion\_sort(mylist)

print("Sorted list:", mylist)

Output;

Sorted list: [11, 14, 18, 25, 37, 41, 60, 82]



def quick\_sort(list1):

# Base case: lists with 0 or 1 elements are already sorted

if len(list1) <= 1:

return list1

else:

pivot = list1[0] # Choose the first element as the pivot

# Partition the list into elements less than or equal to the pivot and greater than the pivot

lesser = [x for x in list1[1:] if x <= pivot]

greater = [x for x in list1[1:] if x > pivot]

# Recursively apply quick sort to the partitions and combine them with the pivot

return quick\_sort(lesser) + [pivot] + quick\_sort(greater)

# Test the quick sort function

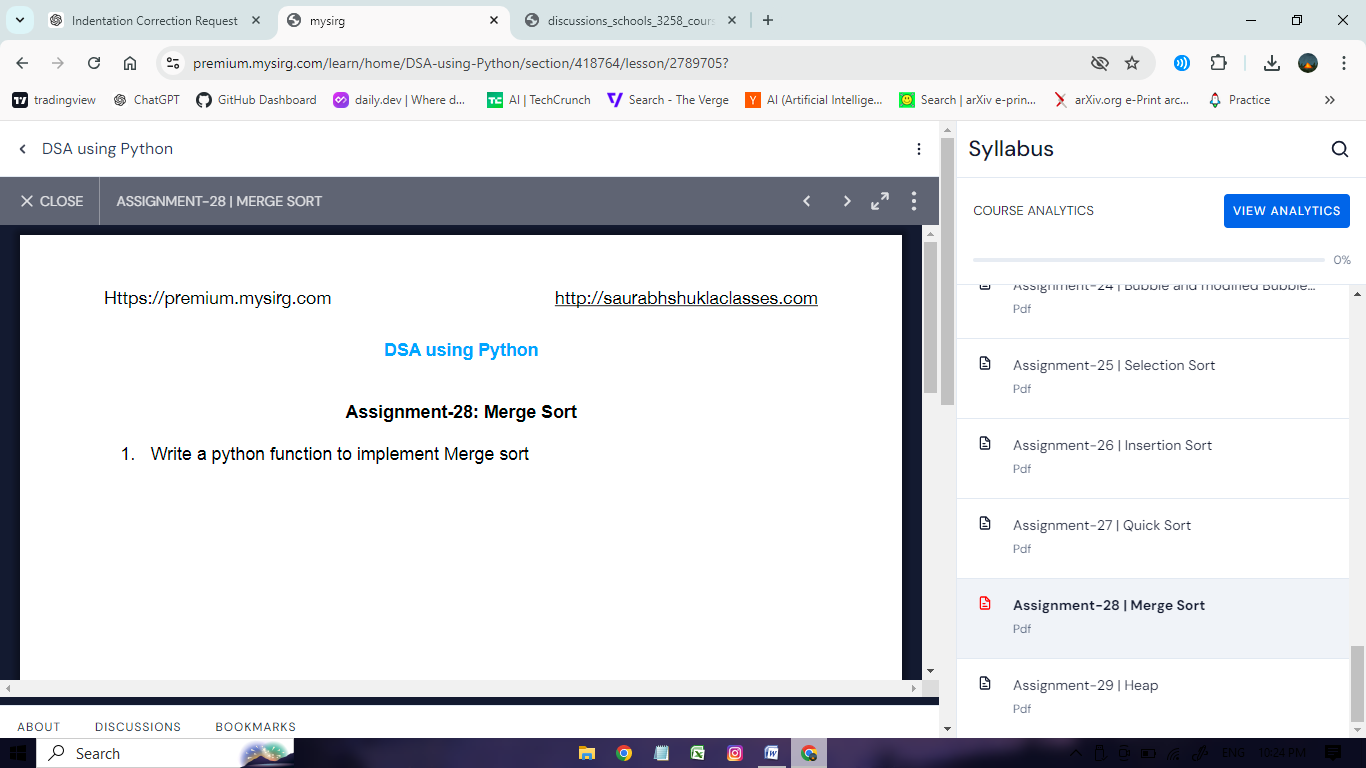
mylist = [53, 11, 72, 68, 41, 25, 18, 37, 44, 80]

mylist = quick\_sort(mylist)

print("Sorted list:", mylist)

Output:

Sorted list: [11, 18, 25, 37, 41, 44, 53, 68, 72, 80]



def merge\_sort(list1):

if len(list1) > 1:

mid = len(list1) // 2

leftlist = list1[:mid]

rightlist = list1[mid:]

# Recursively split and sort both halves

merge\_sort(leftlist)

merge\_sort(rightlist)

# Merge the sorted halves

i = j = k = 0

while i < len(leftlist) and j < len(rightlist):

if leftlist[i] < rightlist[j]:

list1[k] = leftlist[i]

i += 1

else:

list1[k] = rightlist[j]

j += 1

k += 1

# Copy any remaining elements of leftlist

while i < len(leftlist):

list1[k] = leftlist[i]

i += 1

k += 1

# Copy any remaining elements of rightlist

while j < len(rightlist):

list1[k] = rightlist[j]

j += 1

k += 1

# Test the merge sort function

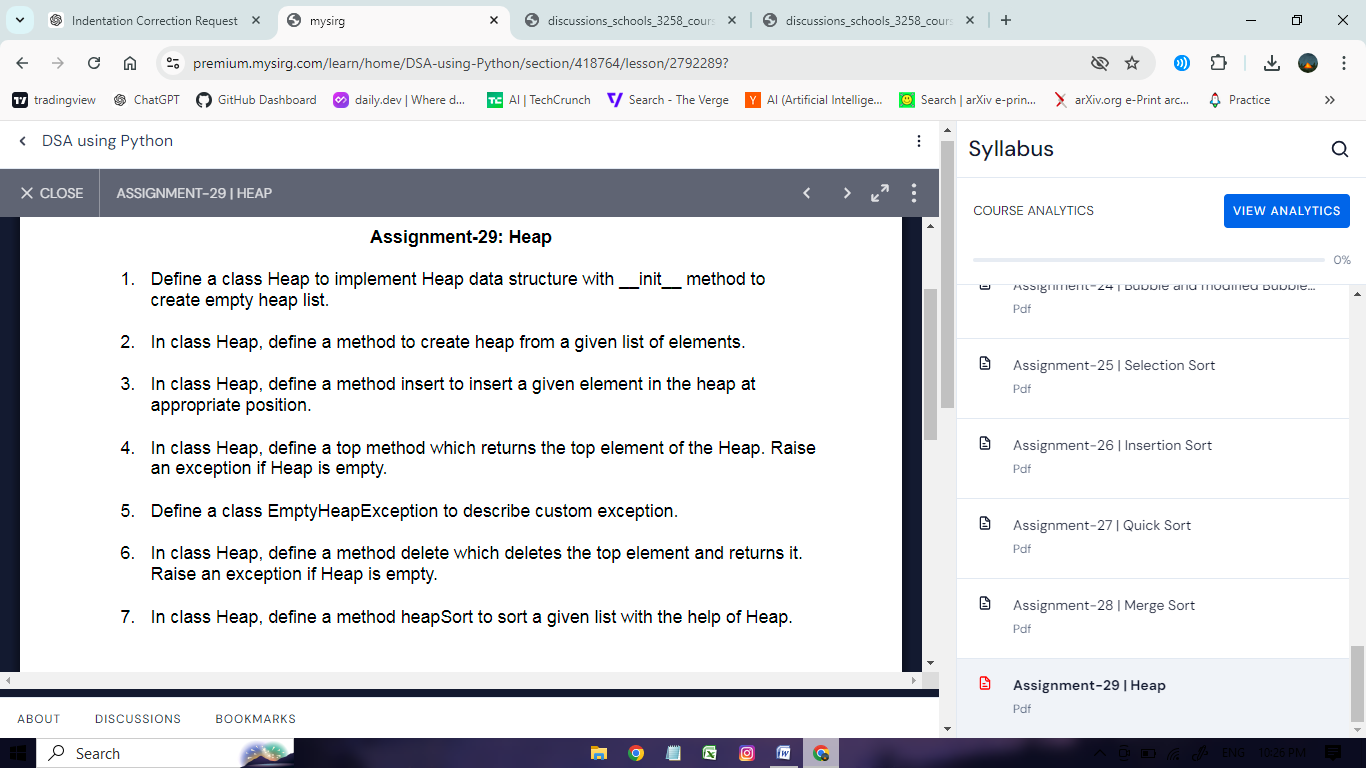
mylist = [75, 29, 83, 42, 16, 90, 50, 36, 20, 71, 32, 92, 7]

merge\_sort(mylist)

print("Sorted list:", mylist)

Output:

Sorted list: [7, 16, 20, 29, 32, 36, 42, 50, 71, 75, 83, 90, 92]



class Heap:

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

self.heap = []

def createHeap(self, list1):

for e in list1:

self.insert(e)

def insert(self, e):

self.heap.append(e)

index = len(self.heap) - 1

parentIndex = (index - 1) // 2

while index > 0 and self.heap[parentIndex] < e:

self.heap[index] = self.heap[parentIndex]

index = parentIndex

parentIndex = (index - 1) // 2

self.heap[index] = e

def top(self):

if len(self.heap) == 0:

raise EmptyHeapException()

return self.heap[0]

def delete(self):

if len(self.heap) == 0:

raise EmptyHeapException()

if len(self.heap) == 1:

return self.heap.pop()

max\_value = self.heap[0]

temp = self.heap.pop()

index = 0

leftChildIndex = 2 \* index + 1

rightChildIndex = 2 \* index + 2

while leftChildIndex < len(self.heap):

if rightChildIndex < len(self.heap):

if self.heap[leftChildIndex] > self.heap[rightChildIndex]:

if self.heap[leftChildIndex] > temp:

self.heap[index] = self.heap[leftChildIndex]

index = leftChildIndex

else:

break

else:

if self.heap[rightChildIndex] > temp:

self.heap[index] = self.heap[rightChildIndex]

index = rightChildIndex

else:

break

else:

if self.heap[leftChildIndex] > temp:

self.heap[index] = self.heap[leftChildIndex]

index = leftChildIndex

else:

break

leftChildIndex = 2 \* index + 1

rightChildIndex = 2 \* index + 2

self.heap[index] = temp

return max\_value

def heapSort(self, list1):

self.createHeap(list1)

list2 = []

try:

while True:

list2.insert(0, self.delete())

except EmptyHeapException:

pass

return list2

class EmptyHeapException(Exception):

def \_\_init\_\_(self, msg="Empty Heap"):

self.msg = msg

def \_\_str\_\_(self):

return self.msg

# Test the heap sort function

list1 = [34, 56, 12, 78, 43, 25, 10, 60]

h = Heap()

sorted\_list = h.heapSort(list1)

print("Sorted list:", sorted\_list)

Output:

Sorted list: [10, 12, 25, 34, 43, 56, 60, 78]