****

**LINDA SELEMANI**

**H240731A**

**MASTERS SOFTWARE ENGINEERING**

**DATA STRUCTURES LABS**

1. Write a program to implement Fibonacci series

def fibonacci\_series (n):

# Initial two terms of the series

fib\_series = [0, 1]

if n < 0:

return "Please enter positive integer"

elif n == 1:

return [0]

elif n == 2:

return fib\_series

else:

while len(fib\_series) < n:

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

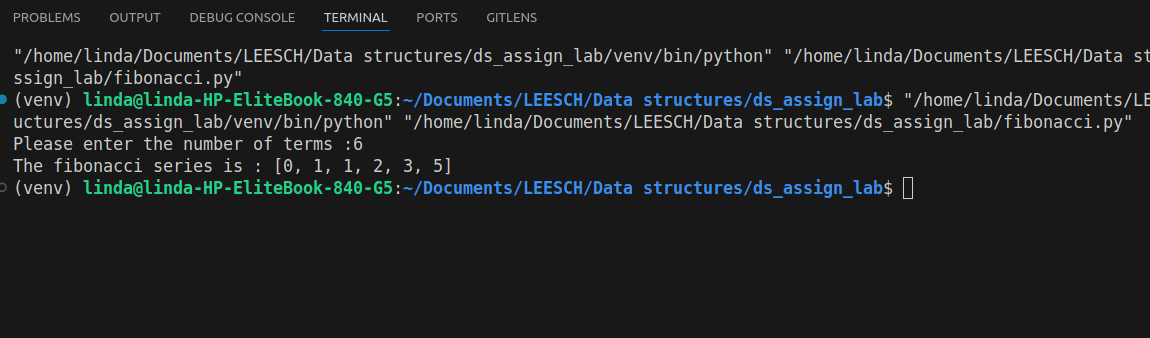
fib\_series.append(next\_term)

return fib\_series

num\_terms = int(input("Please enter the number of terms :"))

print("The fibonacci series is :", fibonacci\_series(num\_terms))

**Output**



1. Write a program to implement factorial of a number

def factorial\_iterative (n):

if n < 0 :

return "Factorial for neative numbers is undefined"

result = 1

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

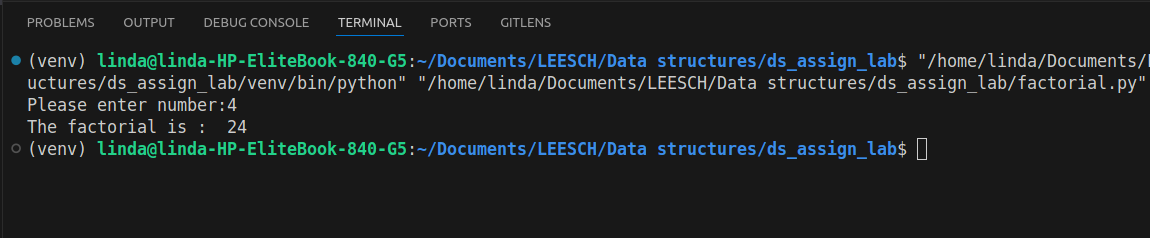
result \*= i

return result

num = int(input("Please enter number:"))

print ("The factorial is : ", factorial\_iterative(num))

**Output**



1. Write a program to implement a single linked list

class Node:

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

self.data = data

self.next = None

class SinglyLinkedList:

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

self.head = None

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

new\_node = Node(data)

if not self.head:

self.head = new\_node

else:

temp = self.head

while temp.next:

temp = temp.next

temp.next = new\_node

print(f"Inserted {data} at the end of the list.")

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

new\_node = Node(data)

new\_node.next = self.head

self.head = new\_node

print(f"Inserted {data} at the beginning of the list.")

def delete\_node(self, key):

temp = self.head

prev = None

if temp and temp.data == key:

self.head = temp.next

temp = None

print(f"Deleted node with value {key}.")

return

while temp and temp.data != key:

prev = temp

temp = temp.next

if temp is None:

print(f"Node with value {key} not found.")

return

prev.next = temp.next

temp = None

print(f"Deleted node with value {key}.")

def display(self):

"""Display the elements in the list."""

if not self.head:

print("The list is empty.")

return

temp = self.head

while temp:

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

temp = temp.next

print("None")

# Example usage

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

linked\_list = SinglyLinkedList()

linked\_list.insert\_at\_end(10)

linked\_list.insert\_at\_end(20)

linked\_list.insert\_at\_beginning(5)

linked\_list.display()

linked\_list.delete\_node(20)

linked\_list.display()

linked\_list.delete\_node(30) # Node not in the list

linked\_list.display()

**Output**



1. Write a program to implement a double linked list

class Node:

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

self.data = data

self.next = None

self.prev = None

class DoublyLinkedList:

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

self.head = None

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

new\_node = Node(data)

if not self.head:

self.head = new\_node

print(f"Inserted {data} at the end of the list.")

return

temp = self.head

while temp.next:

temp = temp.next

temp.next = new\_node

new\_node.prev = temp

print(f"Inserted {data} at the end of the list.")

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

new\_node = Node(data)

if self.head:

self.head.prev = new\_node

new\_node.next = self.head

self.head = new\_node

print(f"Inserted {data} at the beginning of the list.")

def delete\_node(self, key):

temp = self.head

if not temp:

print("The list is empty.")

return

if temp.data == key:

self.head = temp.next

if self.head:

self.head.prev = None

print(f"Deleted node with value {key}.")

return

while temp and temp.data != key:

temp = temp.next

if not temp:

print(f"Node with value {key} not found.")

return

if temp.next:

temp.next.prev = temp.prev

if temp.prev:

temp.prev.next = temp.next

print(f"Deleted node with value {key}.")

def display\_forward(self):

if not self.head:

print("The list is empty.")

return

temp = self.head

print("List in forward order:")

while temp:

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

temp = temp.next

print("None")

def display\_backward(self):

if not self.head:

print("The list is empty.")

return

temp = self.head

while temp.next:

temp = temp.next

print("List in backward order:")

while temp:

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

temp = temp.prev

print("None")

# Example usage

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

dll = DoublyLinkedList()

dll.insert\_at\_end(10)

dll.insert\_at\_end(20)

dll.insert\_at\_beginning(5)

dll.display\_forward()

dll.display\_backward()

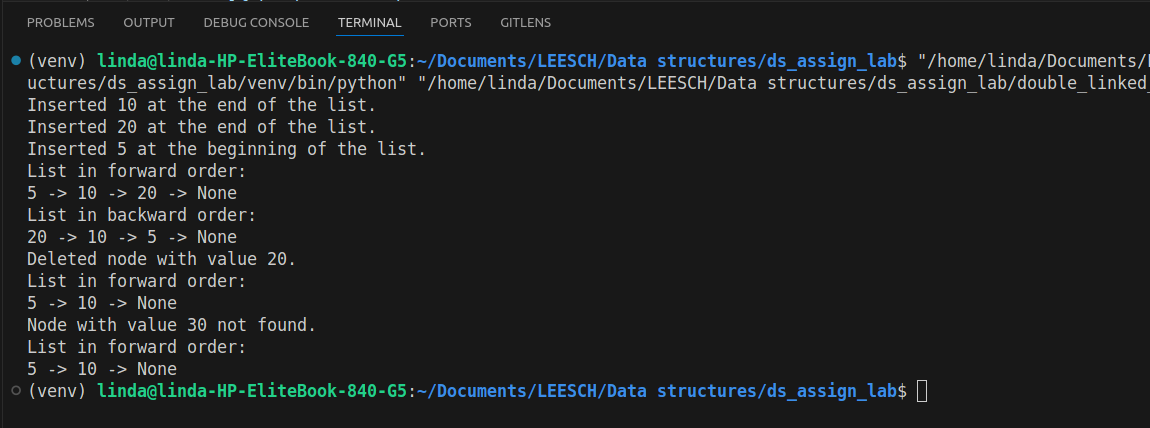
dll.delete\_node(20)

dll.display\_forward()

dll.delete\_node(30) # Node not in the list

dll.display\_forward()

**Output**



1. Write a program to add numbers in a 2-dimensional array

def sum\_2d\_array(array):

total\_sum = 0

for row in array:

total\_sum += sum(row)

return total\_sum

# Example usage

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

array = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

print("2D Array:")

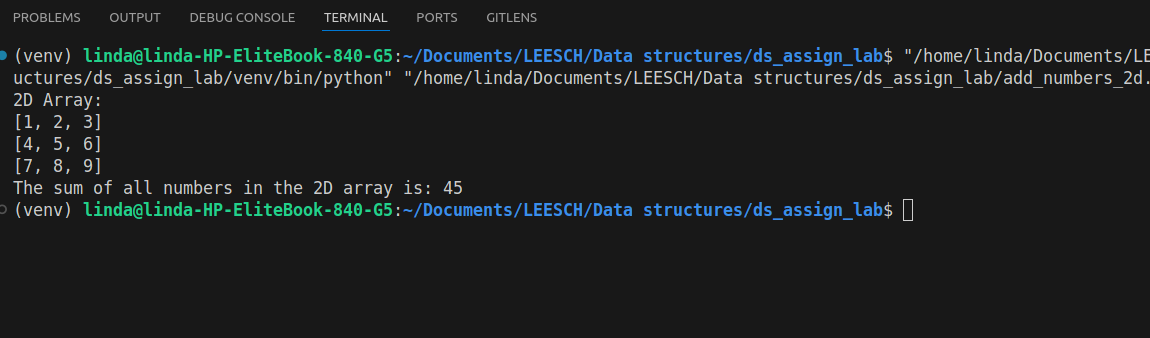
for row in array:

print(row)

result = sum\_2d\_array(array)

print(f"The sum of all numbers in the 2D array is: {result}")

Output



1. Write a program to implement a stack using a linked list

class Node:

# Class to represent a single node in the linked list.

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

self.data = data

self.next = None

class StackUsingLinkedList:

# Class to represent a stack implemented using a linked list.

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

self.top = None # Points to the top of the stack

def is\_empty(self):

# Check if the stack is empty.

return self.top is None

def push(self, data):

# Push a new element onto the stack.

new\_node = Node(data)

new\_node.next = self.top

self.top = new\_node

print(f"Pushed {data} onto the stack.")

def pop(self):

# Pop the top element from the stack.

if self.is\_empty():

print("Stack underflow! Cannot pop from an empty stack.")

return None

popped\_data = self.top.data

self.top = self.top.next

print(f"Popped {popped\_data} from the stack.")

return popped\_data

def peek(self):

# Return the top element without removing it.

if self.is\_empty():

print("Stack is empty.")

return None

return self.top.data

def display(self):

# Display all the elements in the stack.

if self.is\_empty():

print("Stack is empty.")

return

print("Stack elements:")

temp = self.top

while temp:

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

temp = temp.next

print("None")

# Example usage

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

stack = StackUsingLinkedList()

stack.push(10)

stack.push(20)

stack.push(30)

stack.display()

print(f"Top element is: {stack.peek()}")

stack.pop()

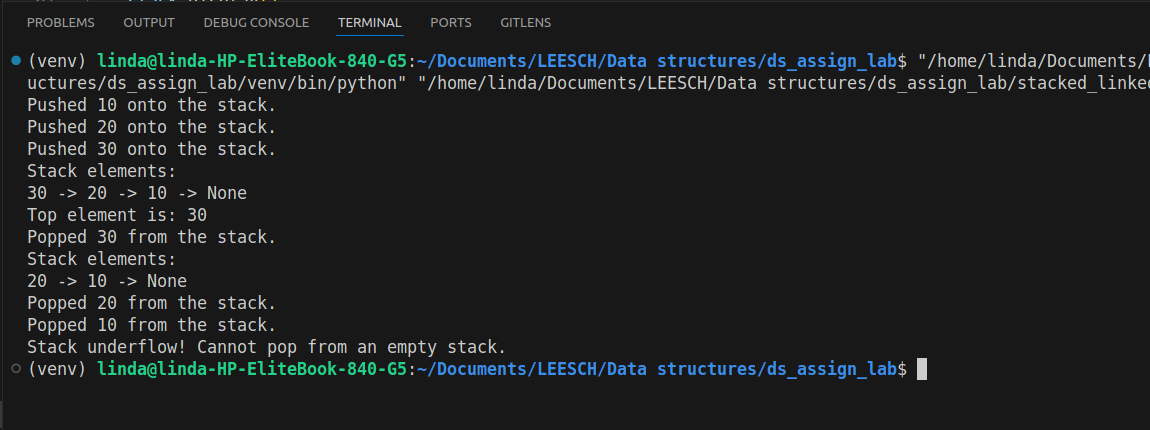
stack.display()

stack.pop()

stack.pop()

stack.pop() # Attempt to pop from an empty stack

**Output**



1. Write a program to implement a stack using an array

class StackUsingArray:

# Class to represent a stack implemented using an array.

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

self.capacity = capacity # Maximum capacity of the stack

self.stack = [] # Underlying list to store stack elements

def is\_empty(self):

# Check if the stack is empty.

return len(self.stack) == 0

def is\_full(self):

# Check if the stack is full.

return len(self.stack) == self.capacity

def push(self, data):

# Push a new element onto the stack.

if self.is\_full():

print("Stack overflow! Cannot push onto a full stack.")

return

self.stack.append(data)

print(f"Pushed {data} onto the stack.")

def pop(self):

# Pop the top element from the stack.

if self.is\_empty():

print("Stack underflow! Cannot pop from an empty stack.")

return None

return self.stack.pop()

def peek(self):

# Return the top element without removing it.

if self.is\_empty():

print("Stack is empty.")

return None

return self.stack[-1]

def display(self):

# Display all the elements in the stack.

if self.is\_empty():

print("Stack is empty.")

return

print("Stack elements (top to bottom):")

for i in range(len(self.stack) - 1, -1, -1):

print(self.stack[i])

# Example usage

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

stack = StackUsingArray(capacity=5)

stack.push(10)

stack.push(20)

stack.push(30)

stack.display()

print(f"Top element is: {stack.peek()}")

print(f"Popped element is: {stack.pop()}")

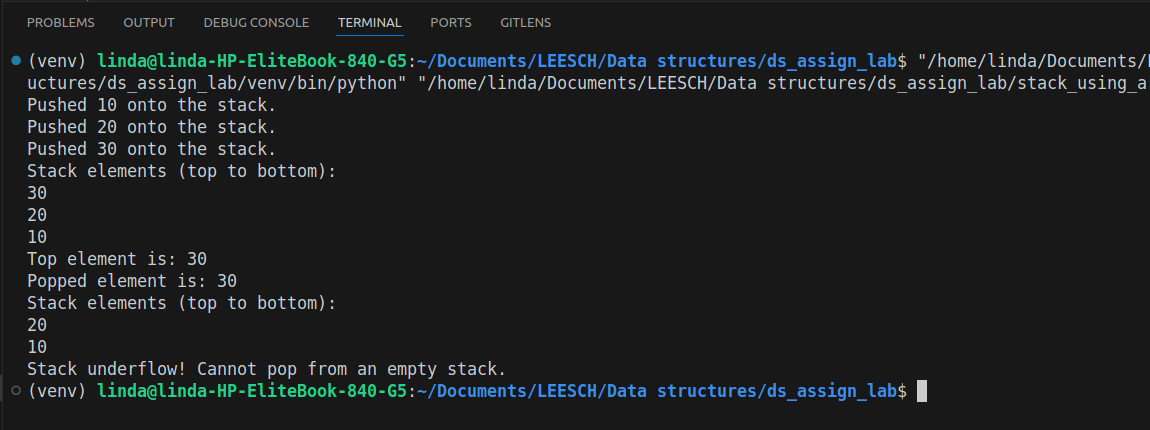
stack.display()

stack.pop()

stack.pop()

stack.pop() # Attempt to pop from an empty stack

**Output**



1. Write a program to implement infix to postfix conversion using stack

class InfixToPostfixConverter:

# Class to convert infix expression to postfix using a stack.

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

self.stack = [] # Stack to hold operators

self.precedence = {'+': 1, '-': 1, '\*': 2, '/': 2, '^': 3} # Operator precedence

self.associativity = {'+': 'L', '-': 'L', '\*': 'L', '/': 'L', '^': 'R'} # Associativity of operators

def is\_operator(self, char):

# Check if a character is an operator

return char in self.precedence

def is\_operand(self, char):

# Check if a character is an operand (alphanumeric).

return char.isalnum()

def precedence\_of(self, operator):

# Get the precedence of an operator.

return self.precedence.get(operator, 0)

def convert(self, infix):

# Convert an infix expression to a postfix expression.

postfix = [] # **Output** list for postfix expression

for char in infix:

if self.is\_operand(char):

postfix.append(char) # Append operands directly to postfix

elif char == '(':

self.stack.append(char) # Push '(' onto the stack

elif char == ')':

# Pop until '(' is encountered

while self.stack and self.stack[-1] != '(':

postfix.append(self.stack.pop())

self.stack.pop() # Pop the '(' from the stack

elif self.is\_operator(char):

# Pop operators from the stack while they have higher or equal precedence (left associative)

while (self.stack and self.stack[-1] != '(' and

(self.precedence\_of(self.stack[-1]) > self.precedence\_of(char) or

(self.precedence\_of(self.stack[-1]) == self.precedence\_of(char) and self.associativity[char] == 'L'))):

postfix.append(self.stack.pop())

self.stack.append(char) # Push the current operator onto the stack

# Pop remaining operators from the stack

while self.stack:

postfix.append(self.stack.pop())

return ''.join(postfix)

# Example usage

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

converter = InfixToPostfixConverter()

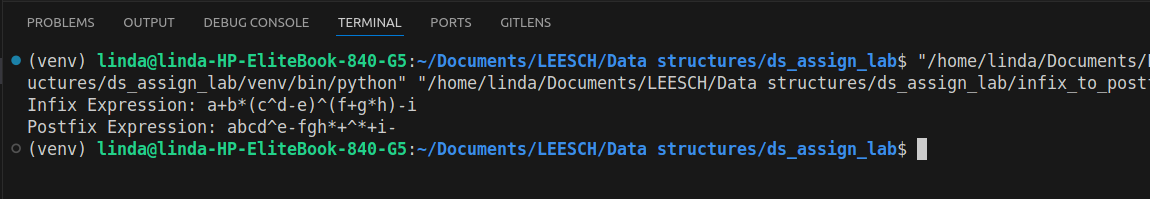
infix\_expression = "a+b\*(c^d-e)^(f+g\*h)-i"

print(f"Infix Expression: {infix\_expression}")

postfix\_expression = converter.convert(infix\_expression)

print(f"Postfix Expression: {postfix\_expression}")

**Output**



1. Write a program to implement a queue using a linked list

class Node:

# Class to represent a single node in the linked list.

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

self.data = data

self.next = None

class QueueUsingLinkedList:

# Class to represent a queue implemented using a linked list.

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

self.front = None # Points to the front of the queue

self.rear = None # Points to the rear of the queue

def is\_empty(self):

# Check if the queue is empty.

return self.front is None

def enqueue(self, data):

# Add an element to the rear of the queue

new\_node = Node(data)

if self.rear is None: # If the queue is empty

self.front = self.rear = new\_node

print(f"Enqueued {data} to the queue.")

return

self.rear.next = new\_node

self.rear = new\_node

print(f"Enqueued {data} to the queue.")

def dequeue(self):

# Remove an element from the front of the queue.

if self.is\_empty():

print("Queue underflow! Cannot dequeue from an empty queue.")

return None

dequeued\_data = self.front.data

self.front = self.front.next

if self.front is None: # If the queue becomes empty

self.rear = None

print(f"Dequeued {dequeued\_data} from the queue.")

return dequeued\_data

def peek(self):

# Get the front element without removing it.

if self.is\_empty():

print("Queue is empty.")

return None

return self.front.data

def display(self):

# Display all the elements in the queue.

if self.is\_empty():

print("Queue is empty.")

return

print("Queue elements (front to rear):")

temp = self.front

while temp:

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

temp = temp.next

print("None")

# Example usage

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

queue = QueueUsingLinkedList()

queue.enqueue(10)

queue.enqueue(20)

queue.enqueue(30)

queue.display()

print(f"Front element is: {queue.peek()}")

queue.dequeue()

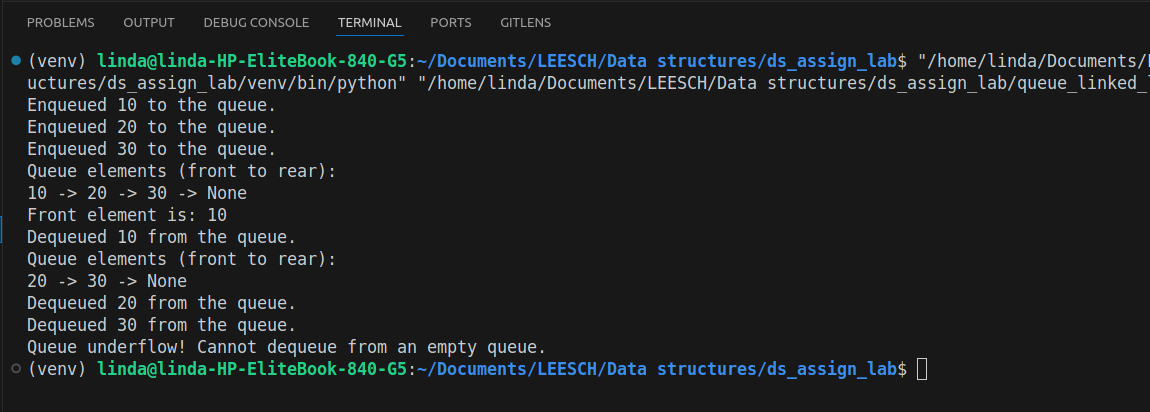
queue.display()

queue.dequeue()

queue.dequeue()

queue.dequeue() # Attempt to dequeue from an empty queue

**Output**



1. Write a program to implement a queue using an array

class QueueUsingArray:

# Class to represent a queue implemented using an array.

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

self.capacity = capacity # Maximum capacity of the queue

self.queue = [] # Underlying list to store queue elements

def is\_empty(self):

# Check if the queue is empty.

return len(self.queue) == 0

def is\_full(self):

# Check if the queue is full.

return len(self.queue) == self.capacity

def enqueue(self, data):

# Add an element to the rear of the queue.

if self.is\_full():

print("Queue overflow! Cannot enqueue to a full queue.")

return

self.queue.append(data)

print(f"Enqueued {data} to the queue.")

def dequeue(self):

# Remove an element from the front of the queue.

if self.is\_empty():

print("Queue underflow! Cannot dequeue from an empty queue.")

return None

dequeued\_data = self.queue.pop(0)

print(f"Dequeued {dequeued\_data} from the queue.")

return dequeued\_data

def peek(self):

# Get the front element without removing it.

if self.is\_empty():

print("Queue is empty.")

return None

return self.queue[0]

def display(self):

# Display all the elements in the queue.

if self.is\_empty():

print("Queue is empty.")

return

print("Queue elements (front to rear):", " -> ".join(map(str, self.queue)))

# Example usage

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

queue = QueueUsingArray(capacity=5)

queue.enqueue(10)

queue.enqueue(20)

queue.enqueue(30)

queue.display()

print(f"Front element is: {queue.peek()}")

queue.dequeue()

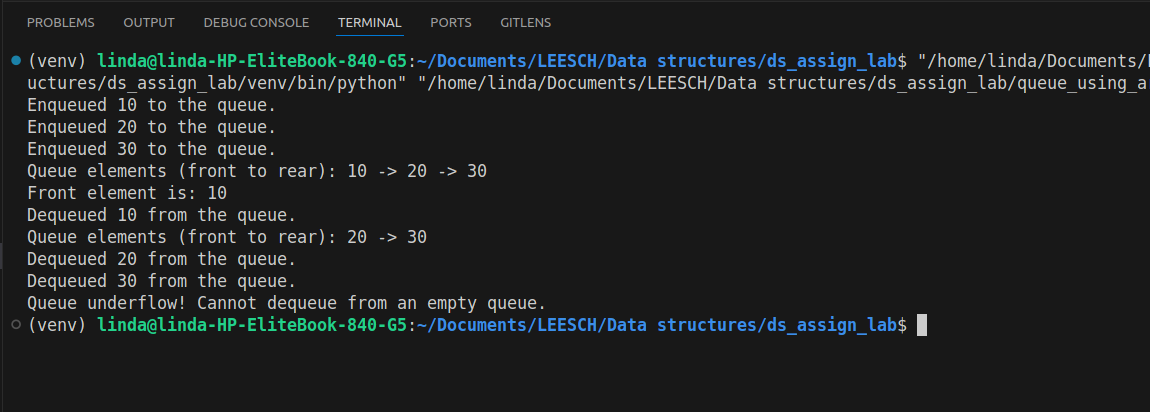
queue.display()

queue.dequeue()

queue.dequeue()

queue.dequeue() # Attempt to dequeue from an empty queue

**Output**



1. Write a program to implement a circular queue

class CircularQueue:

# Class to represent a circular queue.

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

self.capacity = capacity # Maximum capacity of the queue

self.queue = [None] \* capacity # Fixed-size array to store queue elements

self.front = -1 # Points to the front element in the queue

self.rear = -1 # Points to the rear element in the queue

def is\_empty(self):

# Check if the queue is empty.

return self.front == -1

def is\_full(self):

# Check if the queue is full.

return (self.rear + 1) % self.capacity == self.front

def enqueue(self, data):

# Add an element to the rear of the queue.

if self.is\_full():

print("Queue overflow! Cannot enqueue to a full queue.")

return

if self.is\_empty():

self.front = self.rear = 0 # First element

else:

self.rear = (self.rear + 1) % self.capacity # Circular increment

self.queue[self.rear] = data

print(f"Enqueued {data} to the queue.")

def dequeue(self):

# Remove an element from the front of the queue.

if self.is\_empty():

print("Queue underflow! Cannot dequeue from an empty queue.")

return None

dequeued\_data = self.queue[self.front]

if self.front == self.rear: # Only one element was in the queue

self.front = self.rear = -1 # Reset queue

else:

self.front = (self.front + 1) % self.capacity # Circular increment

print(f"Dequeued {dequeued\_data} from the queue.")

return dequeued\_data

def peek(self):

# Get the front element without removing it.

if self.is\_empty():

print("Queue is empty.")

return None

return self.queue[self.front]

def display(self):

# Display all the elements in the queue.

if self.is\_empty():

print("Queue is empty.")

return

print("Queue elements (front to rear):", end=" ")

i = self.front

while True:

print(self.queue[i], end=" ")

if i == self.rear:

break

i = (i + 1) % self.capacity # Circular increment

print()

# Example usage

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

queue = CircularQueue(capacity=5)

queue.enqueue(10)

queue.enqueue(20)

queue.enqueue(30)

queue.enqueue(40)

queue.display()

print(f"Front element is: {queue.peek()}")

queue.dequeue()

queue.dequeue()

queue.display()

queue.enqueue(50)

queue.enqueue(60)

queue.enqueue(70) # Queue becomes full

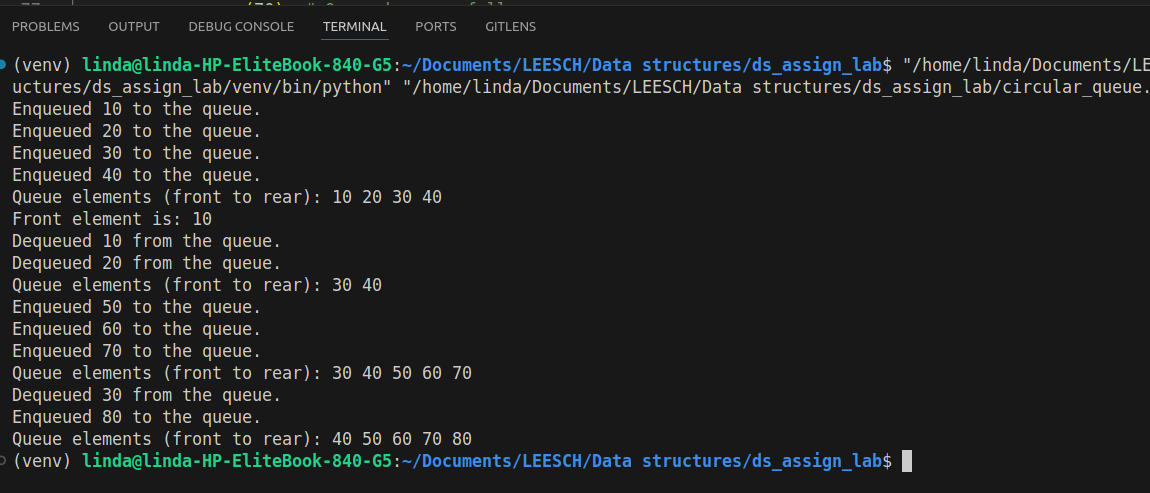
queue.display()

queue.dequeue()

queue.enqueue(80) # Demonstrates circular nature

queue.display()

**Output**



1. Write a program to implement the minimum/maximum heap.

import heapq

class MinHeap:

# Class to represent a minimum heap.

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

self.heap = [] # List to store heap elements

def push(self, data):

# Add an element to the heap.

heapq.heappush(self.heap, data)

print(f"Added {data} to Min Heap.")

def pop(self):

# Remove and return the smallest element from the heap.

if not self.heap:

print("Heap is empty. Cannot pop.")

return None

popped\_data = heapq.heappop(self.heap)

print(f"Popped {popped\_data} from Min Heap.")

return popped\_data

def peek(self):

# Get the smallest element without removing it.

if not self.heap:

print("Heap is empty.")

return None

return self.heap[0]

def display(self):

# Display the heap elements.

if not self.heap:

print("Heap is empty.")

return

print("Min Heap elements:", self.heap)

class MaxHeap:

# Class to represent a maximum heap.

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

self.heap = [] # List to store heap elements

def push(self, data):

# Add an element to the heap (inverted for max heap).

heapq.heappush(self.heap, -data) # Invert the value to simulate max heap

print(f"Added {data} to Max Heap.")

def pop(self):

# Remove and return the largest element from the heap.

if not self.heap:

print("Heap is empty. Cannot pop.")

return None

popped\_data = -heapq.heappop(self.heap) # Invert the value to get original

print(f"Popped {popped\_data} from Max Heap.")

return popped\_data

def peek(self):

# Get the largest element without removing it.

if not self.heap:

print("Heap is empty.")

return None

return -self.heap[0] # Invert the value to get original

def display(self):

# Display the heap elements.

if not self.heap:

print("Heap is empty.")

return

print("Max Heap elements:", [-x for x in self.heap]) # Invert the values for display

# Example usage

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

print("Min Heap Operations:")

min\_heap = MinHeap()

min\_heap.push(10)

min\_heap.push(20)

min\_heap.push(5)

min\_heap.push(30)

min\_heap.display()

print(f"Peek (Min): {min\_heap.peek()}")

min\_heap.pop()

min\_heap.display()

print("\nMax Heap Operations:")

max\_heap = MaxHeap()

max\_heap.push(10)

max\_heap.push(20)

max\_heap.push(5)

max\_heap.push(30)

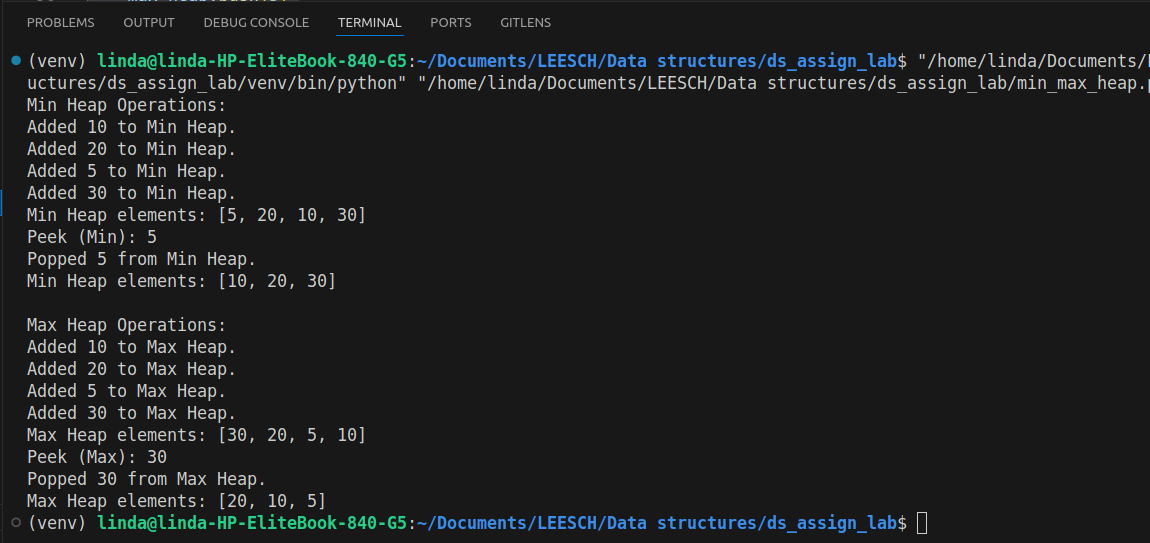
max\_heap.display()

print(f"Peek (Max): {max\_heap.peek()}")

max\_heap.pop()

max\_heap.display()

**Output**



1. Write a program to implement bubble sort

def bubble\_sort(arr):

n = len(arr)

for i in range (n):

for j in range (0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

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

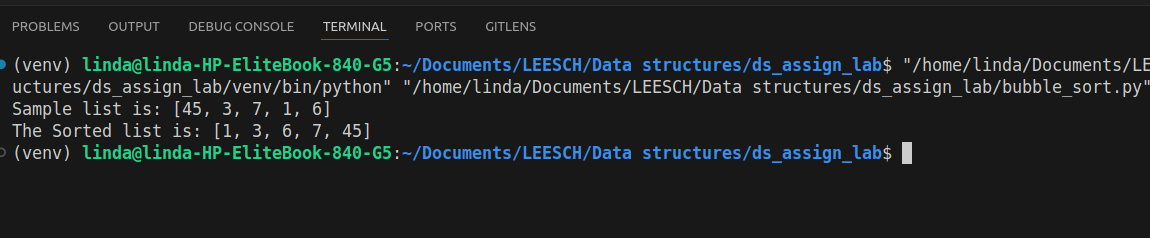
sample\_list = [45, 3, 7, 1, 6]

print("Sample list is:",sample\_list)

bubble\_sort(sample\_list)

print("The Sorted list is:",sample\_list)

**Output**



1. Write a program to implement insertion sort

def insertion\_sort(arr):

# Function to implement the insertion sort algorithm.

n = len(arr)

# Traverse through 1 to len(arr)

for i in range(1, n):

key = arr[i] # The current element to be inserted in the sorted portion

j = i - 1 # The index of the last element of the sorted portion

# Move elements of arr[0..i-1], that are greater than key, to one position ahead

while j >= 0 and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

# Insert the key at the correct position

arr[j + 1] = key

return arr

# Example usage

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

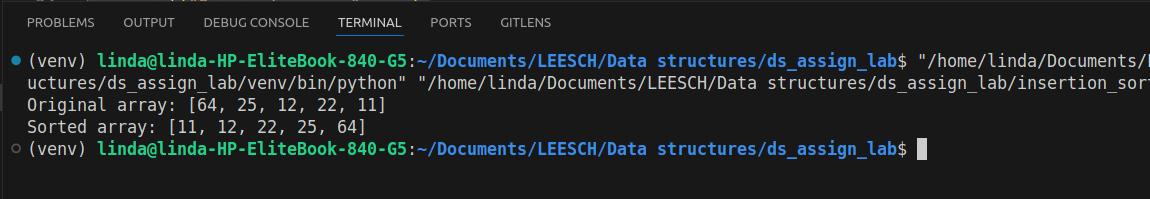
arr = [64, 25, 12, 22, 11]

print("Original array:", arr)

sorted\_arr = insertion\_sort(arr)

print("Sorted array:", sorted\_arr)

**Output**



1. Write a program to implement quick sort

def quick\_sort(arr):

# Function to implement the quick sort algorithm.

if len(arr) <= 1:

return arr

# Choose the pivot (we can choose different pivot strategies, here we choose the last element)

pivot = arr[-1]

# Partition the array into two sub-arrays: smaller and larger than the pivot

left = [x for x in arr[:-1] if x <= pivot]

right = [x for x in arr[:-1] if x > pivot]

# Recursively sort the left and right sub-arrays, and combine them with the pivot

return quick\_sort(left) + [pivot] + quick\_sort(right)

# Example usage

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

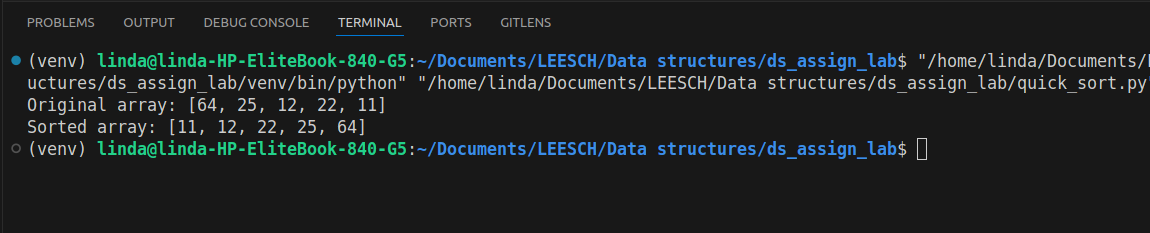
arr = [64, 25, 12, 22, 11]

print("Original array:", arr)

sorted\_arr = quick\_sort(arr)

print("Sorted array:", sorted\_arr)

**Output**



1. Write a program to implement merge sort

def merge\_sort(arr):

# Function to implement the merge sort algorithm.

if len(arr) <= 1:

return arr

# Find the middle point and divide the array into two halves

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

# Recursively sort both halves

left\_sorted = merge\_sort(left\_half)

right\_sorted = merge\_sort(right\_half)

# Merge the sorted halves and return the result

return merge(left\_sorted, right\_sorted)

def merge(left, right):

"""Function to merge two sorted lists into one sorted list."""

merged = []

i = j = 0

# Merge the two sorted lists

while i < len(left) and j < len(right):

if left[i] < right[j]:

merged.append(left[i])

i += 1

else:

merged.append(right[j])

j += 1

# If there are remaining elements in either left or right list, add them to merged

merged.extend(left[i:])

merged.extend(right[j:])

return merged

# Example usage

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

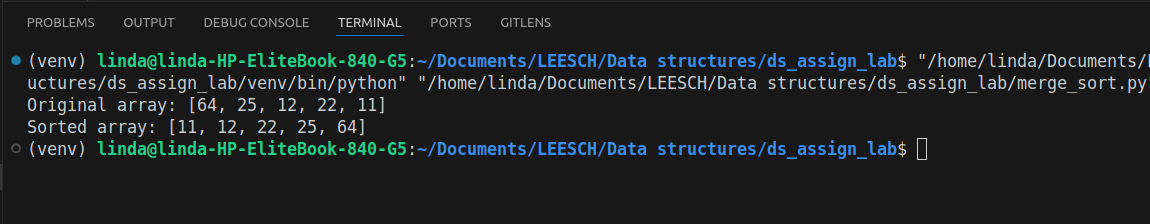
arr = [64, 25, 12, 22, 11]

print("Original array:", arr)

sorted\_arr = merge\_sort(arr)

print("Sorted array:", sorted\_arr)

**Output**



1. Write a program to implement heap sort

def heapify(arr, n, i):

# Helper function to maintain the heap property.

largest = i # Initialize largest as root

left = 2 \* i + 1 # Left child

right = 2 \* i + 2 # Right child

# If left child is larger than root

if left < n and arr[left] > arr[largest]:

largest = left

# If right child is larger than root

if right < n and arr[right] > arr[largest]:

largest = right

# If largest is not root, swap and recursively heapify the affected subtree

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i] # Swap

heapify(arr, n, largest)

def heap\_sort(arr):

"""Function to implement the heap sort algorithm."""

n = len(arr)

# Build a max heap

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i)

# One by one extract elements from the heap

for i in range(n - 1, 0, -1):

# Move current root to the end

arr[i], arr[0] = arr[0], arr[i] # Swap

heapify(arr, i, 0) # Heapify the reduced heap

return arr

# Example usage

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

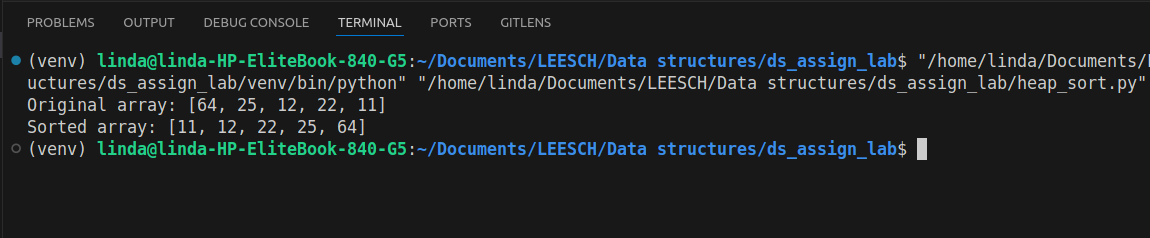
arr = [64, 25, 12, 22, 11]

print("Original array:", arr)

sorted\_arr = heap\_sort(arr)

print("Sorted array:", sorted\_arr)

**Output**



1. Write a program to implement radix sort

def counting\_sort(arr, exp):

"""Helper function to perform counting sort based on the digit represented by exp."""

n = len(arr)

**output** = [0] \* n # **Output** array to store sorted values

count = [0] \* 10 # Count array to store the frequency of digits (0-9)

# Store the count of occurrences in count[]

for i in range(n):

index = arr[i] // exp

count[index % 10] += 1

# Change count[i] so that it now contains the actual position of this digit in **output**[]

for i in range(1, 10):

count[i] += count[i - 1]

# Build the **output** array by placing elements in the correct position based on current digit

i = n - 1

while i >= 0:

index = arr[i] // exp

**output**[count[index % 10] - 1] = arr[i]

count[index % 10] -= 1

i -= 1

# Copy the **output** array to arr[], so that arr[] now contains sorted numbers

for i in range(n):

arr[i] = **output**[i]

def radix\_sort(arr):

"""Function to implement the radix sort algorithm."""

# Find the maximum number to determine the number of digits

max\_num = max(arr)

# Perform counting sort for every digit (exp is 10^i for the current digit)

exp = 1

while max\_num // exp > 0:

counting\_sort(arr, exp)

exp \*= 10

# Example usage

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

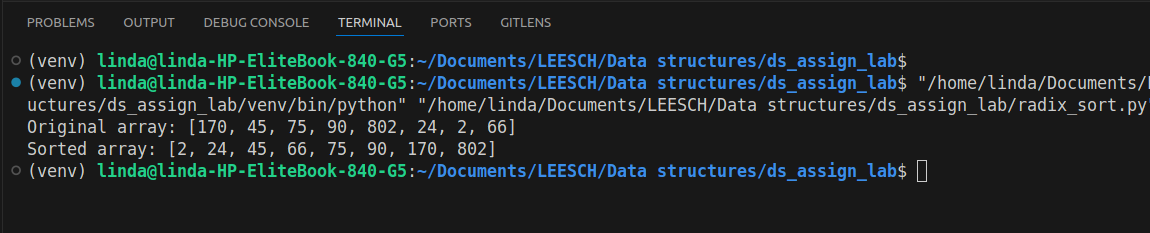
arr = [170, 45, 75, 90, 802, 24, 2, 66]

print("Original array:", arr)

radix\_sort(arr)

print("Sorted array:", arr)

**Output**



1. Write a program to implement breath first search

from collections import deque

class Graph:

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

self.graph = {} # Dictionary to store the graph

def add\_edge(self, u, v):

"""Adds an edge to the graph."""

if u not in self.graph:

self.graph[u] = []

if v not in self.graph:

self.graph[v] = []

self.graph[u].append(v)

self.graph[v].append(u) # Since it's an undirected graph

def bfs(self, start):

"""Breadth-First Search (BFS) starting from the 'start' node."""

visited = set() # Set to track visited nodes

queue = deque([start]) # Queue to store nodes to visit

visited.add(start)

print("BFS Traversal:", end=" ")

while queue:

node = queue.popleft() # Dequeue the node

print(node, end=" ")

# Enqueue unvisited neighbors

for neighbor in self.graph[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

# Example usage

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

# Create a graph instance

g = Graph()

# Add edges to the graph (undirected graph)

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.add\_edge(2, 5)

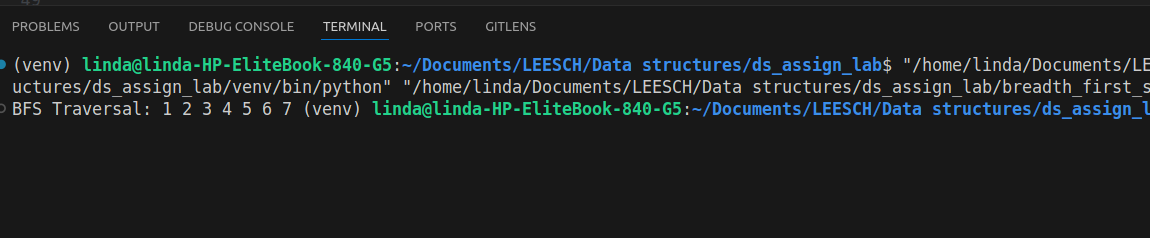
g.add\_edge(3, 6)

g.add\_edge(3, 7)

# Perform BFS traversal starting from node 1

g.bfs(1)

**Output**



1. Write a program to implement depth first search

class Graph:

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

self.graph = {} # Dictionary to store the graph

def add\_edge(self, u, v):

"""Adds an edge to the graph."""

if u not in self.graph:

self.graph[u] = []

if v not in self.graph:

self.graph[v] = []

self.graph[u].append(v)

self.graph[v].append(u) # Since it's an undirected graph

def dfs\_recursive(self, start, visited=None):

"""Performs DFS recursively starting from the 'start' node."""

if visited is None:

visited = set() # Initialize visited set

# Visit the current node and print it

visited.add(start)

print(start, end=" ")

# Visit all unvisited neighbors

for neighbor in self.graph[start]:

if neighbor not in visited:

self.dfs\_recursive(neighbor, visited)

def dfs\_iterative(self, start):

"""Performs DFS iteratively using a stack starting from the 'start' node."""

visited = set() # Set to track visited nodes

stack = [start] # Stack to store the nodes to visit

while stack:

node = stack.pop() # Pop the node from the stack

if node not in visited:

print(node, end=" ")

visited.add(node)

# Add all unvisited neighbors to the stack

# Reverse the neighbors to visit them in the correct order

for neighbor in reversed(self.graph[node]):

if neighbor not in visited:

stack.append(neighbor)

# Example usage

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

# Create a graph instance

g = Graph()

# Add edges to the graph (undirected graph)

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.add\_edge(2, 5)

g.add\_edge(3, 6)

g.add\_edge(3, 7)

# Perform DFS traversal (Recursive approach)

print("DFS Recursive Traversal:")

g.dfs\_recursive(1)

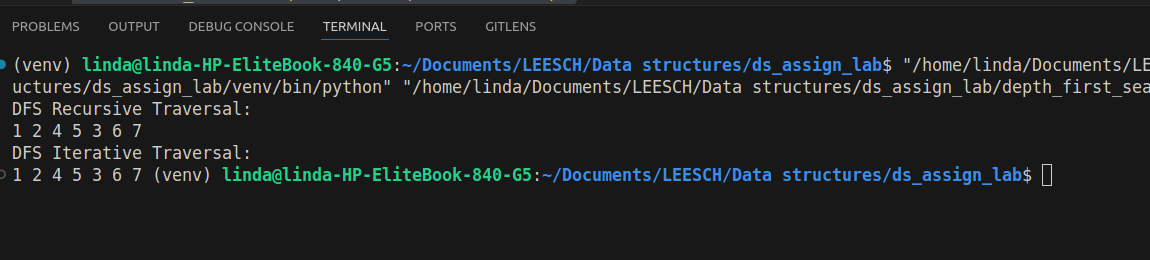
print()

# Perform DFS traversal (Iterative approach)

print("DFS Iterative Traversal:")

g.dfs\_iterative(1)

**Output**



1. Write a program to implement Kruskal’s algorithm.

class DisjointSet:

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

"""Initialize the disjoint-set (union-find) structure."""

self.parent = list(range(n)) # Each node is its own parent initially

self.rank = [0] \* n # Rank to keep the tree flat

def find(self, u):

"""Find the representative of the set containing 'u'."""

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u]) # Path compression

return self.parent[u]

def union(self, u, v):

"""Union the sets containing 'u' and 'v'."""

root\_u = self.find(u)

root\_v = self.find(v)

if root\_u != root\_v:

# Union by rank: attach the smaller tree to the root of the larger tree

if self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

class Kruskal:

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

"""Initialize the Kruskal's algorithm with the number of vertices."""

self.V = vertices # Number of vertices

self.edges = [] # List of edges in the graph

def add\_edge(self, u, v, weight):

"""Add an edge to the graph."""

self.edges.append((weight, u, v))

def kruskal\_mst(self):

"""Perform Kruskal's algorithm to find the MST."""

# Step 1: Sort edges by weight

self.edges.sort()

# Initialize disjoint set

disjoint\_set = DisjointSet(self.V)

mst = [] # To store the resulting MST

total\_weight = 0 # Total weight of MST

# Step 2: Process edges one by one

for weight, u, v in self.edges:

# Check if including this edge forms a cycle

if disjoint\_set.find(u) != disjoint\_set.find(v):

disjoint\_set.union(u, v) # Include this edge in MST

mst.append((u, v, weight)) # Add edge to the MST

total\_weight += weight # Add its weight to the total weight

# If we have included (V-1) edges, we can stop

if len(mst) == self.V - 1:

break

return mst, total\_weight

# Example usage

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

# Number of vertices

num\_vertices = 6

kruskal = Kruskal(num\_vertices)

# Adding edges (u, v, weight)

kruskal.add\_edge(0, 1, 4)

kruskal.add\_edge(0, 2, 3)

kruskal.add\_edge(1, 2, 1)

kruskal.add\_edge(1, 3, 2)

kruskal.add\_edge(2, 3, 4)

kruskal.add\_edge(3, 4, 5)

kruskal.add\_edge(4, 5, 6)

kruskal.add\_edge(2, 5, 7)

# Find the Minimum Spanning Tree (MST)

mst, total\_weight = kruskal.kruskal\_mst()

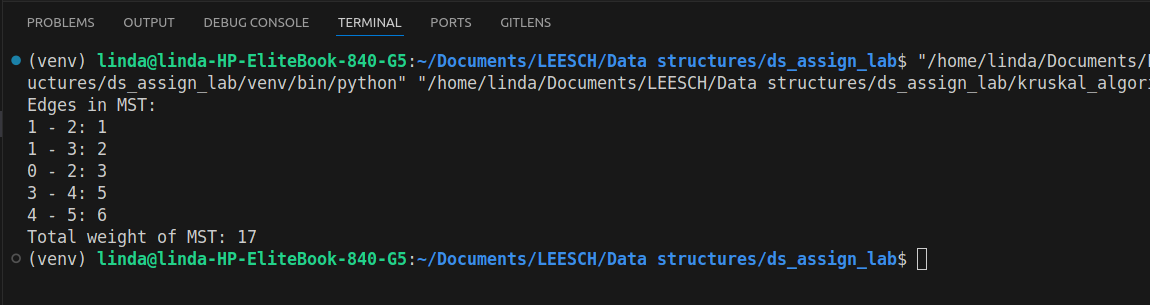
print("Edges in MST:")

for u, v, weight in mst:

print(f"{u} - {v}: {weight}")

print(f"Total weight of MST: {total\_weight}")

**Output**



1. Write a program to implement Dijkstra’ algorithm

import heapq # For priority queue (min-heap)

class Graph:

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

"""Initialize the graph with a number of vertices."""

self.V = vertices # Number of vertices

self.graph = {i: [] for i in range(vertices)} # Adjacency list to store graph

def add\_edge(self, u, v, weight):

"""Add an edge to the graph (directed, with weight)."""

self.graph[u].append((v, weight)) # (neighbor, weight)

def dijkstra(self, start):

"""Run Dijkstra's algorithm from the 'start' node."""

# Distance from start to each vertex, initialize with infinity

distances = [float('inf')] \* self.V

distances[start] = 0 # Distance to the start node is 0

# Priority queue to pick the node with the smallest tentative distance

pq = [(0, start)] # (distance, node)

while pq:

# Get the node with the smallest distance from the priority queue

current\_dist, u = heapq.heappop(pq)

# If the current distance is already greater than the recorded distance, skip

if current\_dist > distances[u]:

continue

# Relax the edges of the current node

for v, weight in self.graph[u]:

distance = current\_dist + weight

if distance < distances[v]: # If a shorter path is found

distances[v] = distance

heapq.heappush(pq, (distance, v)) # Push the neighbor into the priority queue

return distances

# Example usage

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

# Create a graph with 6 vertices

g = Graph(6)

# Add edges (u, v, weight)

g.add\_edge(0, 1, 7)

g.add\_edge(0, 2, 9)

g.add\_edge(0, 5, 14)

g.add\_edge(1, 2, 10)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 11)

g.add\_edge(2, 5, 2)

g.add\_edge(3, 4, 6)

g.add\_edge(4, 5, 9)

# Run Dijkstra's algorithm starting from node 0

start\_node = 0

distances = g.dijkstra(start\_node)

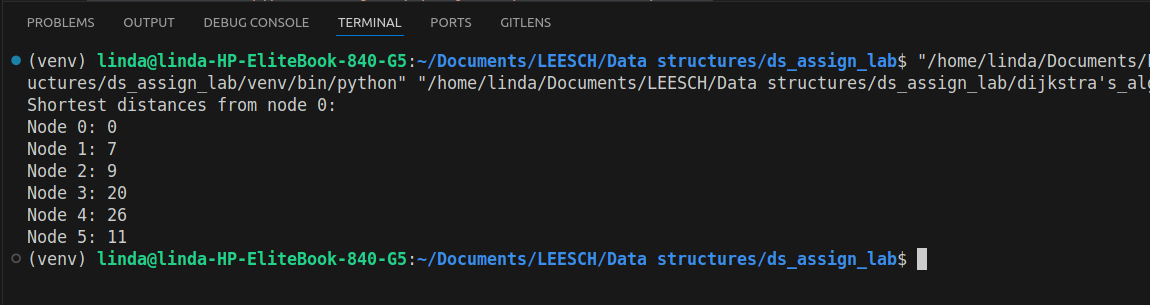
# Print the shortest distances from the start node to each vertex

print(f"Shortest distances from node {start\_node}:")

for node, dist in enumerate(distances):

print(f"Node {node}: {dist}")

**Output**



1. Write a program to implement the operations of Binary Search Trees

class Node:

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

"""Create a node with a given key."""

self.left = None

self.right = None

self.value = key

class BinarySearchTree:

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

"""Initialize an empty BST."""

self.root = None

def insert(self, root, key):

"""Insert a new node with the given key into the BST."""

if root is None:

return Node(key)

else:

if key < root.value:

root.left = self.insert(root.left, key) # Insert in left subtree

else:

root.right = self.insert(root.right, key) # Insert in right subtree

return root

def search(self, root, key):

"""Search for a node with the given key in the BST."""

if root is None or root.value == key:

return root

if key < root.value:

return self.search(root.left, key) # Search in left subtree

return self.search(root.right, key) # Search in right subtree

def find\_min(self, root):

"""Find the node with the minimum key."""

current = root

while current.left is not None:

current = current.left

return current

def find\_max(self, root):

"""Find the node with the maximum key."""

current = root

while current.right is not None:

current = current.right

return current

def delete(self, root, key):

"""Delete a node with the given key from the BST."""

if root is None:

return root

if key < root.value:

root.left = self.delete(root.left, key) # Delete from left subtree

elif key > root.value:

root.right = self.delete(root.right, key) # Delete from right subtree

else:

# Node to be deleted is found

if root.left is None:

return root.right

elif root.right is None:

return root.left

# Node has two children: Get the inorder successor (smallest in the right subtree)

min\_node = self.find\_min(root.right)

root.value = min\_node.value # Replace the value with the inorder successor

root.right = self.delete(root.right, min\_node.value) # Delete the inorder successor

return root

def inorder(self, root):

"""Inorder traversal of the BST."""

if root:

self.inorder(root.left)

print(root.value, end=" ")

self.inorder(root.right)

def preorder(self, root):

"""Preorder traversal of the BST."""

if root:

print(root.value, end=" ")

self.preorder(root.left)

self.preorder(root.right)

def postorder(self, root):

"""Postorder traversal of the BST."""

if root:

self.postorder(root.left)

self.postorder(root.right)

print(root.value, end=" ")

# Example usage

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

bst = BinarySearchTree()

root = None

# Insert elements into the BST

values = [20, 8, 22, 4, 12, 10, 14]

for value in values:

root = bst.insert(root, value)

print("Inorder Traversal:")

bst.inorder(root) # **Output**: 4 8 10 12 14 20 22

print()

print("Preorder Traversal:")

bst.preorder(root) # **Output**: 20 8 4 12 10 14 22

print()

print("Postorder Traversal:")

bst.postorder(root) # **Output**: 4 10 14 12 8 22 20

print()

# Search for a node

key = 10

result = bst.search(root, key)

if result:

print(f"Node with value {key} found!")

else:

print(f"Node with value {key} not found.")

# Find the minimum and maximum nodes

min\_node = bst.find\_min(root)

max\_node = bst.find\_max(root)

print(f"Minimum value in the BST: {min\_node.value}")

print(f"Maximum value in the BST: {max\_node.value}")

# Delete a node

key\_to\_delete = 12

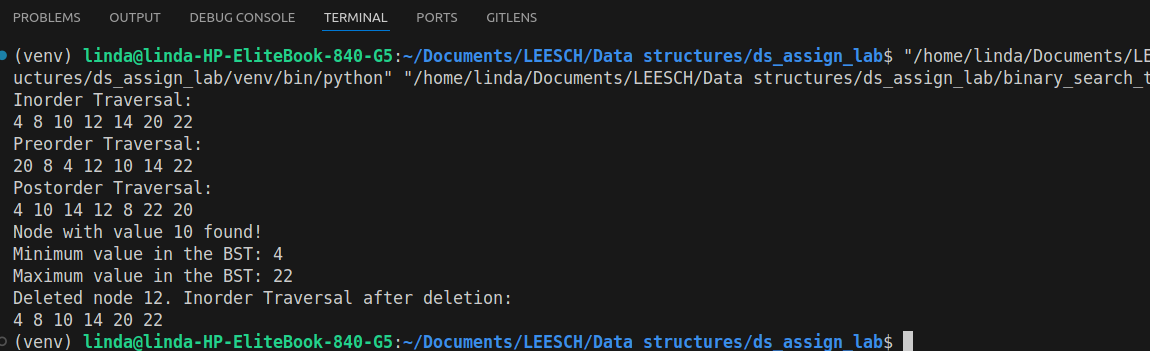
root = bst.delete(root, key\_to\_delete)

print(f"Deleted node {key\_to\_delete}. Inorder Traversal after deletion:")

bst.inorder(root) # **Output**: 4 8 10 14 20 22

print()

**Output**



1. Write a program to implement the operations of AVL trees.

class Node:

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

"""Create a node with a given key."""

self.key = key

self.left = None

self.right = None

self.height = 1 # Initially, the height of the node is 1

class AVLTree:

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

"""Initialize an empty AVL Tree."""

self.root = None

def get\_height(self, node):

"""Get the height of a node."""

if not node:

return 0

return node.height

def get\_balance\_factor(self, node):

"""Get the balance factor of a node."""

if not node:

return 0

return self.get\_height(node.left) - self.get\_height(node.right)

def right\_rotate(self, y):

"""Perform a right rotation on the given node."""

x = y.left

T2 = x.right

# Perform rotation

x.right = y

y.left = T2

# Update heights

y.height = max(self.get\_height(y.left), self.get\_height(y.right)) + 1

x.height = max(self.get\_height(x.left), self.get\_height(x.right)) + 1

return x # Return new root

def left\_rotate(self, x):

"""Perform a left rotation on the given node."""

y = x.right

T2 = y.left

# Perform rotation

y.left = x

x.right = T2

# Update heights

x.height = max(self.get\_height(x.left), self.get\_height(x.right)) + 1

y.height = max(self.get\_height(y.left), self.get\_height(y.right)) + 1

return y # Return new root

def insert(self, node, key):

"""Insert a new key into the AVL tree and balance it."""

# Perform the normal BST insert

if not node:

return Node(key)

if key < node.key:

node.left = self.insert(node.left, key)

else:

node.right = self.insert(node.right, key)

# Update height of this ancestor node

node.height = max(self.get\_height(node.left), self.get\_height(node.right)) + 1

# Get the balance factor of this ancestor node to check whether it became unbalanced

balance = self.get\_balance\_factor(node)

# Left Left Case

if balance > 1 and key < node.left.key:

return self.right\_rotate(node)

# Right Right Case

if balance < -1 and key > node.right.key:

return self.left\_rotate(node)

# Left Right Case

if balance > 1 and key > node.left.key:

node.left = self.left\_rotate(node.left)

return self.right\_rotate(node)

# Right Left Case

if balance < -1 and key < node.right.key:

node.right = self.right\_rotate(node.right)

return self.left\_rotate(node)

# Return the (unchanged) node pointer

return node

def inorder(self, root):

"""Inorder traversal of the AVL tree."""

if root:

self.inorder(root.left)

print(root.key, end=" ")

self.inorder(root.right)

def delete(self, node, key):

"""Delete a key from the AVL tree and balance it."""

# Step 1: Perform the normal BST delete

if not node:

return node

if key < node.key:

node.left = self.delete(node.left, key)

elif key > node.key:

node.right = self.delete(node.right, key)

else:

# Node to be deleted is found

if not node.left:

return node.right

elif not node.right:

return node.left

# Node with two children, get the inorder successor (smallest in the right subtree)

temp = self.get\_min\_value\_node(node.right)

node.key = temp.key

node.right = self.delete(node.right, temp.key)

# Step 2: Update height of the current node

node.height = max(self.get\_height(node.left), self.get\_height(node.right)) + 1

# Step 3: Get the balance factor of the current node and balance the tree

balance = self.get\_balance\_factor(node)

# Left Left Case

if balance > 1 and self.get\_balance\_factor(node.left) >= 0:

return self.right\_rotate(node)

# Right Right Case

if balance < -1 and self.get\_balance\_factor(node.right) <= 0:

return self.left\_rotate(node)

# Left Right Case

if balance > 1 and self.get\_balance\_factor(node.left) < 0:

node.left = self.left\_rotate(node.left)

return self.right\_rotate(node)

# Right Left Case

if balance < -1 and self.get\_balance\_factor(node.right) > 0:

node.right = self.right\_rotate(node.right)

return self.left\_rotate(node)

return node

def get\_min\_value\_node(self, node):

"""Get the node with the minimum value (leftmost node)."""

current = node

while current.left:

current = current.left

return current

# Example usage

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

avl\_tree = AVLTree()

# Insert nodes into the AVL tree

root = None

values = [20, 4, 15, 70, 50, 100, 10, 8, 30]

for value in values:

root = avl\_tree.insert(root, value)

print("Inorder Traversal after insertions:")

avl\_tree.inorder(root) # **Output** should be in sorted order

print()

# Delete a node

root = avl\_tree.delete(root, 70)

print("Inorder Traversal after deletion of 70:")

avl\_tree.inorder(root)

print()

# Delete another node

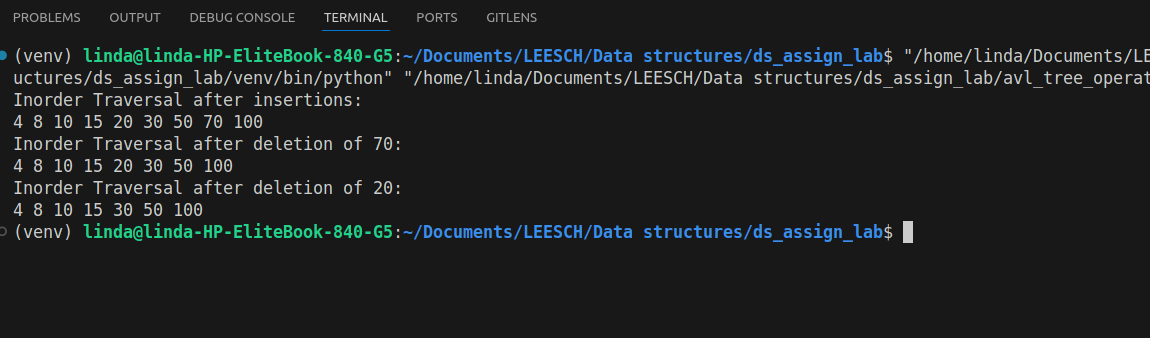
root = avl\_tree.delete(root, 20)

print("Inorder Traversal after deletion of 20:")

avl\_tree.inorder(root)

print()

**Output**



1. Write a program to implement the operations of Red – Black trees.

class Node:

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

"""Create a node with given key and color (red by default)."""

self.key = key

self.left = None

self.right = None

self.parent = None

self.color = "RED" # New nodes are always red by default

class RedBlackTree:

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

"""Initialize an empty Red-Black Tree."""

self.TNULL = Node(0)

self.TNULL.color = "BLACK" # Sentinel node is always black

self.root = self.TNULL

def left\_rotate(self, x):

"""Perform a left rotation."""

y = x.right

x.right = y.left

if y.left != self.TNULL:

y.left.parent = x

y.parent = x.parent

if x.parent == None:

self.root = y

elif x == x.parent.left:

x.parent.left = y

else:

x.parent.right = y

y.left = x

x.parent = y

def right\_rotate(self, x):

"""Perform a right rotation."""

y = x.left

x.left = y.right

if y.right != self.TNULL:

y.right.parent = x

y.parent = x.parent

if x.parent == None:

self.root = y

elif x == x.parent.right:

x.parent.right = y

else:

x.parent.left = y

y.right = x

x.parent = y

def insert(self, key):

"""Insert a new node with given key into the Red-Black tree."""

new\_node = Node(key)

new\_node.parent = None

new\_node.key = key

new\_node.left = self.TNULL

new\_node.right = self.TNULL

new\_node.color = "RED"

y = None

x = self.root

while x != self.TNULL:

y = x

if new\_node.key < x.key:

x = x.left

else:

x = x.right

new\_node.parent = y

if y == None:

self.root = new\_node

elif new\_node.key < y.key:

y.left = new\_node

else:

y.right = new\_node

if new\_node.parent == None:

new\_node.color = "BLACK"

return

if new\_node.parent.parent == None:

return

self.fix\_insert(new\_node)

def fix\_insert(self, k):

"""Fix violations caused by the insertion of a new node."""

while k.parent.color == "RED":

if k.parent == k.parent.parent.right:

u = k.parent.parent.left

if u.color == "RED":

u.color = "BLACK"

k.parent.color = "BLACK"

k.parent.parent.color = "RED"

k = k.parent.parent

else:

if k == k.parent.left:

k = k.parent

self.right\_rotate(k)

k.parent.color = "BLACK"

k.parent.parent.color = "RED"

self.left\_rotate(k.parent.parent)

else:

u = k.parent.parent.right

if u.color == "RED":

u.color = "BLACK"

k.parent.color = "BLACK"

k.parent.parent.color = "RED"

k = k.parent.parent

else:

if k == k.parent.right:

k = k.parent

self.left\_rotate(k)

k.parent.color = "BLACK"

k.parent.parent.color = "RED"

self.right\_rotate(k.parent.parent)

if k == self.root:

break

self.root.color = "BLACK"

def inorder\_helper(self, node):

"""Inorder traversal to print the tree."""

if node != self.TNULL:

self.inorder\_helper(node.left)

print(f"{node.key}({node.color})", end=" ")

self.inorder\_helper(node.right)

def print\_tree(self):

"""Print the tree in inorder traversal."""

self.inorder\_helper(self.root)

print()

# Example usage

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

rbt = RedBlackTree()

# Insert nodes into the Red-Black tree

values = [20, 15, 25, 10, 5, 1, 30]

for value in values:

rbt.insert(value)

print("Inorder Traversal of Red-Black Tree:")

rbt.print\_tree()

**Output**

