DATA STRUCTURES LAB

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))

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()

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()

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}")

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

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

1. Write a program to implement infix to postfix conversion using stack
2. Write a program to implement a queue using a linked list
3. Write a program to implement a queue using an array
4. Write a program to implement a circular queue
5. Write a program to implement the minimum/maximum heap.
6. Write a program to implement bubble sort
7. Write a program to implement insertion sort
8. Write a program to implement quick sort
9. Write a program to implement merge sort
10. Write a program to implement heap sort
11. Write a program to implement radix sort
12. Write a program to implement breath first search
13. Write a program to implement depth first search
14. Write a program to implement Kruskal’s algorithm.
15. Write a program to implement Dijkstra’ algorithm
16. Write a program to implement the operations of Binary Search Trees
17. Write a program to implement the operations of AVL trees.
18. Write a program to implement the operations of Red – Black trees.