**Binary Search Tree (BST)**:

class Node:

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

self.key = key

self.left = None

self.right = None

class BST:

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

self.root = None

def insert(self, key):

def \_insert(root, key):

if root is None:

return Node(key)

if key < root.key:

root.left = \_insert(root.left, key)

elif key > root.key:

root.right = \_insert(root.right, key)

return root

self.root = \_insert(self.root, key)

def inorder\_display(self):

def \_inorder(root):

if root is not None:

\_inorder(root.left)

print(root.key, end=" ")

\_inorder(root.right)

\_inorder(self.root)

print()

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

bst = BST()

while True:

print("1. Insert")

print("2. Display (Inorder)")

print("3. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

key = int(input("Enter the value to insert: "))

bst.insert(key)

elif choice == 2:

print("Inorder Traversal of BST:")

bst.inorder\_display()

elif choice == 3:

print("Exiting...")

break

else:

print("Invalid choice. Please try again.")

**merge two sorted linked lists**.

class Node:

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

self.data = data

self.next = None

class LinkedList:

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

self.head = None

def append(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

return

current = self.head

while current.next:

current = current.next

current.next = new\_node

def print\_list(self):

current = self.head

while current:

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

current = current.next

print("None")

def merge\_sorted\_lists(list1, list2):

dummy = Node(0)

tail = dummy

while list1 and list2:

if list1.data < list2.data:

tail.next = list1

list1 = list1.next

else:

tail.next = list2

list2 = list2.next

tail = tail.next

tail.next = list1 if list1 else list2

return dummy.next

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

list1 = LinkedList()

list1.append(1)

list1.append(3)

list1.append(5)

list2 = LinkedList()

list2.append(2)

list2.append(4)

list2.append(6)

list1.print\_list()

list2.print\_list()

merged\_list = LinkedList()

merged\_list.head = merge\_sorted\_lists(list1.head, list2.head)

merged\_list.print\_list()

**implementation of Singly Linked List to perform Insert and Display operations.**

class Node:

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

self.data = data

self.next = None

class SinglyLinkedList:

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

self.head = None

def insert(self, data):

new\_node = Node(data)

new\_node.next = self.head

self.head = new\_node

def display(self):

if self.head is None:

print("The list is empty.")

return

current = self.head

while current:

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

current = current.next

print("None")

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

sll = SinglyLinkedList()

while True:

print("\nSingly Linked List Operations:")

print("1. Insert")

print("2. Display")

print("3. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

data = int(input("Enter data to insert: "))

sll.insert(data)

elif choice == 2:

sll.display()

elif choice == 3:

break

else:

print("Invalid choice. Please try again.")

**create a linked list in the sorted order.**

class Node:

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

self.data = data

self.next = None

class SortedLinkedList:

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

self.head = None

def insert\_sorted(self, data):

new\_node = Node(data)

if not self.head or self.head.data >= new\_node.data:

new\_node.next = self.head

self.head = new\_node

else:

current = self.head

while current.next and current.next.data < new\_node.data:

current = current.next

new\_node.next = current.next

current.next = new\_node

def display(self):

if not self.head:

print("The list is empty.")

return

current = self.head

while current:

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

current = current.next

print("None")

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

sll = SortedLinkedList()

while True:

print("\nSorted Linked List Operations:")

print("1. Insert")

print("2. Display")

print("3. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

data = int(input("Enter data to insert: "))

sll.insert\_sorted(data)

elif choice == 2:

sll.display()

elif choice == 3:

break

else:

print("Invalid choice. Please try again.")

**calculate outdegree of a graph using adjacency matrix.**

def calculate\_outdegree(graph):

outdegrees = []

for row in graph:

outdegree = sum(row)

outdegrees.append(outdegree)

return outdegrees

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

graph = [

[0, 1, 0, 0],

[0, 0, 1, 0],

[1, 0, 0, 1],

[0, 0, 0, 0]

]

outdegrees = calculate\_outdegree(graph)

for i, outdegree in enumerate(outdegrees):

print(f"Outdegree of vertex {i}: {outdegree}")

**static implementation of stack.**

class Stack:

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

self.size = size

self.stack = []

def is\_full(self):

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

def is\_empty(self):

return len(self.stack) == 0

def push(self, item):

if self.is\_full():

print("Stack Overflow")

else:

self.stack.append(item)

def pop(self):

if self.is\_empty():

print("Stack Underflow")

else:

return self.stack.pop()

def peek(self):

if self.is\_empty():

print("Stack is empty")

else:

return self.stack[-1]

def display(self):

if self.is\_empty():

print("Stack is empty")

else:

print("Stack:", self.stack)

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

stack = Stack(5)

while True:

print("\nStack Operations:")

print("1. Push")

print("2. Pop")

print("3. Peek")

print("4. Display")

print("5. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

item = int(input("Enter item to push: "))

stack.push(item)

elif choice == 2:

popped\_item = stack.pop()

if popped\_item:

print(f"Popped item: {popped\_item}")

elif choice == 3:

top\_item = stack.peek()

if top\_item:

print(f"Top item: {top\_item}")

elif choice == 4:

stack.display()

elif choice == 5:

break

else:

print("Invalid choice. Please try again.")