1. **Write a program to insert N employee information into the Linked List.**

**a. Write a function to Search an Employee and display his details.**

**b. Display the Average salary of all the Employees with designation Manager.**

Ans :

class Employee:

def \_\_init\_\_(self, emp\_id, name, designation, salary):

self.emp\_id = emp\_id

self.name = name

self.designation = designation

self.salary = salary

self.next = None

class EmployeeLinkedList:

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

self.head = None

def insert\_employee(self, emp\_id, name, designation, salary):

new\_employee = Employee(emp\_id, name, designation, salary)

if not self.head:

self.head = new\_employee

else:

new\_employee.next = self.head

self.head = new\_employee

def search\_employee(self, emp\_id\_to\_find):

current = self.head

while current:

if current.emp\_id == emp\_id\_to\_find:

return current

current = current.next

return None

def average\_salary\_of\_managers(self):

total\_salary = 0

manager\_count = 0

current = self.head

while current:

if current.designation == "Manager":

total\_salary += current.salary

manager\_count += 1

current = current.next

if manager\_count == 0:

return 0 # Avoid division by zero

else:

return total\_salary / manager\_count

# Function to input N employee details into the linked list

def input\_employee\_data(N):

emp\_list = EmployeeLinkedList()

for \_ in range(N):

emp\_id = int(input("Enter Employee ID: "))

name = input("Enter Employee Name: ")

designation = input("Enter Employee Designation: ")

salary = float(input("Enter Employee Salary: "))

emp\_list.insert\_employee(emp\_id, name, designation, salary)

return emp\_list

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

N = int(input("Enter the number of employees: "))

employee\_list = input\_employee\_data(N)

# Search for an employee by ID

emp\_id\_to\_find = int(input("Enter the Employee ID to search: "))

found\_employee = employee\_list.search\_employee(emp\_id\_to\_find)

if found\_employee:

print("Employee Found:")

print(f"ID: {found\_employee.emp\_id}, Name: {found\_employee.name}, Designation: {found\_employee.designation}, Salary: {found\_employee.salary}")

else:

print("Employee not found.")

# Calculate and display the average salary of Managers

avg\_salary = employee\_list.average\_salary\_of\_managers()

if avg\_salary > 0:

print(f"\nAverage Salary of Managers: {avg\_salary}")

else:

print("\nNo Managers found or the average salary is zero.")

1. **Consider two liked list A and B in sorted order. Write a program to merge the linked list A and B such that the final list C is in sorted order**

Ans :

class Node:

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

self.data = data

self.next = None

class LinkedList:

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

self.head = None

def insert(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

else:

current = self.head

while current.next:

current = current.next

current.next = new\_node

def display(self):

current = self.head

while current:

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

current = current.next

print("None")

def merge\_sorted\_lists(list\_A, list\_B):

merged\_list = LinkedList()

current\_A, current\_B = list\_A.head, list\_B.head

while current\_A and current\_B:

if current\_A.data < current\_B.data:

merged\_list.insert(current\_A.data)

current\_A = current\_A.next

else:

merged\_list.insert(current\_B.data)

current\_B = current\_B.next

# Append any remaining elements from list A and list B

while current\_A:

merged\_list.insert(current\_A.data)

current\_A = current\_A.next

while current\_B:

merged\_list.insert(current\_B.data)

current\_B = current\_B.next

return merged\_list

# Create two sorted linked lists A and B

list\_A = LinkedList()

list\_B = LinkedList()

elements\_A = [1, 3, 5, 7, 9]

elements\_B = [2, 4, 6, 8, 10]

for elem in elements\_A:

list\_A.insert(elem)

for elem in elements\_B:

list\_B.insert(elem)

# Merge the two sorted lists into a new sorted list C

list\_C = merge\_sorted\_lists(list\_A, list\_B)

# Display the merged list C

print("Merged Sorted List C:")

list\_C.display()

1. **Consider two liked list A of size N. Write a program to Split the linked list two parts i.e. B and C each of size N/2.**

Ans :

class Node:

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

self.data = data

self.next = None

class LinkedList:

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

self.head = None

def insert(self, data):

new\_node = Node(data)

if not self.head:

self.head = new\_node

else:

current = self.head

while current.next:

current = current.next

current.next = new\_node

def display(self):

current = self.head

while current:

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

current = current.next

print("None")

def split\_linked\_list(list\_A):

list\_B = LinkedList()

list\_C = LinkedList()

# Find the size of list\_A

size\_A = 0

current = list\_A.head

while current:

size\_A += 1

current = current.next

# Calculate the size of each part (N/2)

size\_B = size\_A // 2

# Split the list into two parts

current = list\_A.head

for i in range(size\_B):

list\_B.insert(current.data)

current = current.next

while current:

list\_C.insert(current.data)

current = current.next

return list\_B, list\_C

# Create a linked list A

list\_A = LinkedList()

# Insert elements into list A

elements = [1, 2, 3, 4, 5, 6, 7, 8]

for elem in elements:

list\_A.insert(elem)

# Split list A into two parts B and C

list\_B, list\_C = split\_linked\_list(list\_A)

# Display the two split lists B and C

print("List B:")

list\_B.display()

print("\nList C:")

list\_C.display()

1. **Write a program to create a single linked list. The address of the head is to be stored in a separate structure which has two fields struct head { struct node \*head\_ptr; int num; }; The head\_ptr will store the pointer to the head node num: Is the count of the number of elements present in the linked list. Write implementations for insertion and deletion which will update the head structure appropriately**

Ans :

#include <stdio.h>

#include <stdlib.h>

// Node structure

struct node {

int data;

struct node \*next;

};

// Head structure

struct head {

struct node \*head\_ptr;

int num;

};

// Function to insert a new node at the beginning

void insert(struct head \*list\_head, int data) {

struct node \*new\_node = malloc(sizeof(struct node));

new\_node->data = data;

new\_node->next = list\_head->head\_ptr;

list\_head->head\_ptr = new\_node;

list\_head->num++;

}

// Function to delete a node with a given value

void delete\_node(struct head \*list\_head, int value) {

struct node \*current = list\_head->head\_ptr, \*prev = NULL;

while (current) {

if (current->data == value) {

if (prev) prev->next = current->next;

else list\_head->head\_ptr = current->next;

free(current);

list\_head->num--;

return;

}

prev = current;

current = current->next;

}

printf("Element not found: %d\n", value);

}

// Function to display the linked list

void display(struct head \*list\_head) {

struct node \*current = list\_head->head\_ptr;

printf("Linked List: ");

while (current) {

printf("%d -> ", current->data);

current = current->next;

}

printf("NULL\n");

}

int main() {

struct head list\_head = {NULL, 0};

insert(&list\_head, 1);

insert(&list\_head, 2);

insert(&list\_head, 3);

display(&list\_head);

delete\_node(&list\_head, 2);

display(&list\_head);

return 0;

}

1. **WAP to implement two stacks in one array A[1 .. N] in such a way that neither stack overflows unless the total number of elements in both stacks together is N.**

Ans :

class TwoStacksInOneArray:

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

self.N = N

self.array = [None] \* N

self.top1 = -1 # Top of stack 1

self.top2 = N # Top of stack 2 (starting from the end)

def push1(self, data):

if self.top1 < self.top2 - 1:

self.top1 += 1

self.array[self.top1] = data

else:

print("Stack 1 overflow")

def push2(self, data):

if self.top1 < self.top2 - 1:

self.top2 -= 1

self.array[self.top2] = data

else:

print("Stack 2 overflow")

def pop1(self):

if self.top1 >= 0:

data = self.array[self.top1]

self.top1 -= 1

return data

else:

print("Stack 1 is empty")

return None

def pop2(self):

if self.top2 < self.N:

data = self.array[self.top2]

self.top2 += 1

return data

else:

print("Stack 2 is empty")

return None

def display(self, stack\_num):

if stack\_num == 1:

print("Stack 1:", self.array[:self.top1 + 1])

elif stack\_num == 2:

print("Stack 2:", self.array[self.top2:])

else:

print("Invalid stack number")

# Create an instance of TwoStacksInOneArray with N=10

stacks = TwoStacksInOneArray(10)

stacks.push1(1)

stacks.push1(2)

stacks.push1(3)

stacks.push2(10)

stacks.push2(9)

stacks.display(1) # Display Stack 1

stacks.display(2) # Display Stack 2

stacks.pop1() # Pop from Stack 1

stacks.pop2() # Pop from Stack 2

stacks.display(1) # Display Stack 1

stacks.display(2) # Display Stack 2

1. . **WAP to implement stack operations PUSH and POP using 2 Queues.**

Ans :

from collections import deque

class StackUsingQueues:

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

self.queue1 = deque()

self.queue2 = deque()

def push(self, data):

# Push the element to an empty queue (queue1 or queue2)

if len(self.queue1) == 0:

self.queue1.append(data)

else:

self.queue2.append(data)

def pop(self):

if len(self.queue1) == 0 and len(self.queue2) == 0:

print("Stack is empty")

return None

# Move all elements from the non-empty queue to the other queue

if len(self.queue1) == 0:

while len(self.queue2) > 1:

self.queue1.append(self.queue2.popleft())

return self.queue2.popleft()

else:

while len(self.queue1) > 1:

self.queue2.append(self.queue1.popleft())

return self.queue1.popleft()

def is\_empty(self):

return len(self.queue1) == 0 and len(self.queue2) == 0

def size(self):

return len(self.queue1) + len(self.queue2)

# Create a stack using two queues

stack = StackUsingQueues()

stack.push(1)

stack.push(2)

stack.push(3)

print("Stack:")

while not stack.is\_empty():

print(stack.pop())

stack.push(4)

stack.push(5)

print("\nStack:")

while not stack.is\_empty():

print(stack.pop())

1. **WAP to implement Queue operations INSERT and DELETE using 2 stacks.**

Ans :

class QueueUsingStacks:

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

self.stack1 = []

self.stack2 = []

def insert(self, data):

# Push the element onto stack1

self.stack1.append(data)

def delete(self):

if not self.stack1 and not self.stack2:

print("Queue is empty")

return None

# If stack2 is empty, move all elements from stack1 to stack2

if not self.stack2:

while self.stack1:

self.stack2.append(self.stack1.pop())

# Pop the front element from stack2

return self.stack2.pop()

def is\_empty(self):

return not self.stack1 and not self.stack2

# Create a queue using two stacks

queue = QueueUsingStacks()

queue.insert(1)

queue.insert(2)

queue.insert(3)

print("Queue:")

while not queue.is\_empty():

print(queue.delete())

queue.insert(4)

queue.insert(5)

print("\nQueue:")

while not queue.is\_empty():

print(queue.delete())

1. **Write a program to insert N Student information into the Doubly Linked List. Write a function to**

**a. Search a Student and display his details.**

**b. Display the details of the students who have scored above 90 in Math’s and Science.**

Ans :

class Student:

def \_\_init\_\_(self, name, roll\_number, math\_score, science\_score):

self.name = name

self.roll\_number = roll\_number

self.math\_score = math\_score

self.science\_score = science\_score

self.prev = None

self.next = None

class DoublyLinkedList:

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

self.head = None

def insert\_student(self, name, roll\_number, math\_score, science\_score):

new\_student = Student(name, roll\_number, math\_score, science\_score)

if not self.head:

self.head = new\_student

else:

new\_student.next = self.head

self.head.prev = new\_student

self.head = new\_student

def search\_student(self, roll\_number\_to\_find):

current = self.head

while current:

if current.roll\_number == roll\_number\_to\_find:

return current

current = current.next

return None

def display\_students\_above\_90(self):

current = self.head

while current:

if current.math\_score > 90 and current.science\_score > 90:

print(f"Name: {current.name}, Roll Number: {current.roll\_number}, Math Score: {current.math\_score}, Science Score: {current.science\_score}")

current = current.next

# Function to input N student details into the doubly linked list

def input\_student\_data(N):

student\_list = DoublyLinkedList()

for i in range(N):

name = input(f"Enter Student {i + 1} Name: ")

roll\_number = int(input(f"Enter Student {i + 1} Roll Number: "))

math\_score = float(input(f"Enter Math Score for Student {i + 1}: "))

science\_score = float(input(f"Enter Science Score for Student {i + 1}: "))

student\_list.insert\_student(name, roll\_number, math\_score, science\_score)

return student\_list

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

N = int(input("Enter the number of students: "))

student\_list = input\_student\_data(N)

# Search for a student by roll number

roll\_number\_to\_find = int(input("Enter the Roll Number to search: "))

found\_student = student\_list.search\_student(roll\_number\_to\_find)

if found\_student:

print("Student Found:")

print(f"Name: {found\_student.name}, Roll Number: {found\_student.roll\_number}, Math Score: {found\_student.math\_score}, Science Score: {found\_student.science\_score}")

else:

print("Student not found.")

# Display details of students who scored above 90 in Math and Science

print("\nStudents who scored above 90 in Math and Science:")

student\_list.display\_students\_above\_90()

1. **Write a program to sort an array using following Sorting Techniques:**

**a. Bubble Sort**

**b. Selection Sort**

**c. Insertion Sort**

**d. Quick Sort**

**e. Merge Sort**

**f. Heap Sort**

Ans :

# 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]

# Selection Sort

def selection\_sort(arr):

n = len(arr)

for i in range(n):

min\_index = i

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

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

min\_index = j

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

# Insertion Sort

def insertion\_sort(arr):

n = len(arr)

for i in range(1, n):

key = arr[i]

j = i - 1

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

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

# Quick Sort

def quick\_sort(arr):

if len(arr) <= 1:

return arr

else:

pivot = arr[0]

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

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

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

# Merge Sort

def merge\_sort(arr):

if len(arr) <= 1:

return arr

def merge(left, right):

result = []

i = j = 0

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

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:])

result.extend(right[j:])

return result

mid = len(arr) // 2

left = merge\_sort(arr[:mid])

right = merge\_sort(arr[mid:])

return merge(left, right)

# Heap Sort

def heap\_sort(arr):

import heapq

heapq.heapify(arr)

sorted\_arr = []

while arr:

sorted\_arr.append(heapq.heappop(arr))

return sorted\_arr

# Test the sorting techniques

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

print("Original Array:", arr)

bubble\_sort(arr.copy())

print("Bubble Sort:", arr)

selection\_sort(arr.copy())

print("Selection Sort:", arr)

insertion\_sort(arr.copy())

print("Insertion Sort:", arr)

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

sorted\_arr = quick\_sort(arr.copy())

print("Quick Sort:", sorted\_arr)

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

sorted\_arr = merge\_sort(arr.copy())

print("Merge Sort:", sorted\_arr)

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

sorted\_arr = heap\_sort(arr.copy())

print("Heap Sort:", sorted\_arr)

1. **Write a program to search an element from the array using following Searching Techniques: a. Linear Search b. Binary Search**

Ans :

# Linear Search

def linear\_search(arr, target):

for i in range(len(arr)):

if arr[i] == target:

return i

return -1

# Binary Search (Assumes the array is sorted)

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

# Test the searching techniques

arr = [1, 3, 5, 7, 9, 11, 13, 15]

# Element to search for

target = 7

# Linear Search

linear\_result = linear\_search(arr, target)

if linear\_result != -1:

print(f"Linear Search: Element {target} found at index {linear\_result}")

else:

print(f"Linear Search: Element {target} not found")

# Binary Search

binary\_result = binary\_search(arr, target)

if binary\_result != -1:

print(f"Binary Search: Element {target} found at index {binary\_result}")

else:

print(f"Binary Search: Element {target} not found")

1. . **Write a program to convert a given min heap to a max heap**

Ans :

def max\_heapify(arr, n, i):

largest = i

left\_child = 2 \* i + 1

right\_child = 2 \* i + 2

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

largest = left\_child

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

largest = right\_child

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

max\_heapify(arr, n, largest)

def convert\_min\_heap\_to\_max\_heap(arr):

n = len(arr)

# Start from the last non-leaf node and heapify all nodes

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

max\_heapify(arr, n, i)

# Test the conversion

min\_heap = [3, 5, 9, 6, 8, 20, 10, 12, 18, 9]

print("Min Heap:", min\_heap)

convert\_min\_heap\_to\_max\_heap(min\_heap)

print("Max Heap:", min\_heap)

1. **WAP to perform following operations a. Extracting Maximum element from Heap b. Inserting element into Heap**

Ans :

def extract\_max(arr):

if len(arr) == 0:

return None # The heap is empty

if len(arr) == 1:

return arr.pop() # There is only one element in the heap

# Swap the root (maximum element) with the last element

arr[0], arr[-1] = arr[-1], arr[0]

# Extract the maximum element (now at the end)

max\_element = arr.pop()

# Maintain the max heap property by heapifying the root

max\_heapify(arr, len(arr), 0)

return max\_element

# Example usage:

max\_heap = [10, 8, 7, 6, 5, 3, 2, 1]

print("Max Heap:", max\_heap)

max\_element = extract\_max(max\_heap)

print("Extracted Maximum Element:", max\_element)

print("Updated Max Heap:", max\_heap)

def insert(arr, key):

# Append the new element to the end of the heap

arr.append(key)

# Heapify the element up to its correct position

i = len(arr) - 1

while i > 0 and arr[i] > arr[(i - 1) // 2]:

arr[i], arr[(i - 1) // 2] = arr[(i - 1) // 2], arr[i]

i = (i - 1) // 2

# Example usage:

max\_heap = [10, 8, 7, 6, 5, 3, 2, 1]

print("Max Heap:", max\_heap)

insert(max\_heap, 9)

print("Max Heap after Insertion:", max\_heap)