1. Implement stack using linked list.
2. Write the algorithm of PUSH and POP operations on the stack

Ans :

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def is_empty(self):
        return self.top is None

    def push(self, data):
        new_node = Node(data)
        if self.is_empty():
            self.top = new_node
        else:
            new_node.next = self.top
            self.top = new_node

    def pop(self):
        if self.is_empty():
            return None
        popped_data = self.top.data
        self.top = self.top.next
        return popped_data

    def peek(self):
        if self.is_empty():
            return None
        return self.top.data

# Example usage:
stack = Stack()

stack.push(1)
stack.push(2)
stack.push(3)

print("Stack:")
while not stack.is_empty():
    print(stack.pop())

# Output:
# 3
```

```
# 2
# 1
```

3. Queue

POINTER IMPLEMENTATION

Enqueue-

```
new_node = Node(data)
if self.is_empty():
   self.front = self.rear = new_node
else:
   self.rear.next = new_node
   self.rear = new_node
```

Dequeu –

```
if self.is_empty():
   return None  # Queue underflow
dequeued_data = self.front.data
self.front = self.front.next
if self.front is None:
   self.rear = None
return dequeued_data
```

ARRAY IMPLEMENTATION

Enqueue –

```
def enqueue(self, data):
   if self.is_full():
      raise Exception("Queue is full")
   self.rear += 1
   self.array[self.rear] = data
   self.size += 1
```

Dequeue –

```
def dequeue(self):
   if self.is_empty():
      raise Exception("Queue is empty")
   dequeued_data = self.array[self.front]
   self.front += 1
   self.size -= 1
   return dequeued_data
```

CIRCULAR QUEUE –

```
def enqueue(self, data):
   if self.is_full():
      raise Exception("Queue is full")
   self.rear = (self.rear + 1) % self.max_size
   self.array[self.rear] = data
   self.size += 1
```

Dequeue –

```
def dequeue(self):
    if self.is_empty():
        raise Exception("Queue is empty")
    dequeued_data = self.array[self.front]
    self.front = (self.front + 1) % self.max_size
    self.size -= 1
    return dequeued_data
```

Priority Queue using heap –

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def insert(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def delete(self):
        if not self.is_empty():
            return heapq.heappop(self.elements)[1]

    def peek(self):
        if not self.is_empty():
            return self.elements[0][1]

    def is_empty(self):
        return len(self.elements) == 0

    def size(self):
        return len(self.elements)
```

4. Singly Linked List

```
# Define a Node class to represent individual elements in the linked list.
class Node:
    def __init__(self, data):
        self.data = data    # Data stored in the node.
        self.next = None    # Reference to the next node in the list.

# Define a SinglyLinkedList class to manage the linked list.
class SinglyLinkedList:
    def __init__(self):
        self.head = None    # Reference to the first node in the list (initially None).

    # Insert a new node at the beginning of the list.
    def insert_at_beginning(self, data):
        new_node = Node(data)  # Create a new node with the given data.
```

```
    new_node.next = self.head  # Set the new node's "next" reference to the current head.
    self.head = new_node  # Update the head to point to the new node.

# Insert a new node at the end of the list.
def insert_at_end(self, data):
    new_node = Node(data)  # Create a new node with the given data.
    if not self.head:
        self.head = new_node  # If the list is empty, set the new node as the head and return.
        return
    current = self.head
    while current.next:
        current = current.next  # Traverse the list until the last node is reached.
    current.next = new_node  # Set the "next" reference of the last node to the new node.

# Delete the first node in the list.
def delete_at_beginning(self):
    if self.head:
        self.head = self.head.next  # Update the head to point to the next node.

# Delete the last node in the list.
def delete_at_end(self):
    if not self.head:
        return  # If the list is empty, nothing to delete.
    if not self.head.next:
        self.head = None  # If there's only one node, set the head to None.
        return
    current = self.head
    while current.next.next:
        current = current.next  # Traverse the list until the second-to-last node is reached.
    current.next = None  # Set the "next" reference of the second-to-last node to None.

# Search for a specific value in the list.
def search(self, target):
    current = self.head
    while current:
        if current.data == target:
            return True  # If the target is found, return True.
        current = current.next
    return False  # If the target is not found, return False.

# Traverse and print the elements in the list.
def traverse(self):
    current = self.head
    while current:
        print(current.data, end=" -> ")  # Print the data in the current node.
        current = current.next
    print("None")  # Print "None" to indicate the end of the list.
```

```
    # Calculate the length of the list.
    def length(self):
        count = 0
        current = self.head
        while current:
            count += 1  # Increment the count for each node in the list.
            current = current.next
        return count  # Return the count as the length of the list.
```

5. Doubly Linked list

```
# Define a Node class to represent individual elements in the doubly linked list.
class Node:
    def __init__(self, data):
        self.data = data    # Data stored in the node.
        self.next = None    # Reference to the next node in the list.
        self.prev = None    # Reference to the previous node in the list.

# Define a DoublyLinkedList class to manage the doubly linked list.
class DoublyLinkedList:
    def __init__(self):
        self.head = None    # Reference to the first node in the list (initially None).
        self.tail = None    # Reference to the last node in the list (initially None).

    # Insert a new node at the beginning of the list.
    def insert_at_beginning(self, data):
        new_node = Node(data)  # Create a new node with the given data.
        new_node.next = self.head  # Set the new node's "next" reference to the current head.
        if self.head:
            self.head.prev = new_node  # Set the previous reference of the current head to the
new node.
        self.head = new_node  # Update the head to point to the new node.
        if not self.tail:
            self.tail = new_node  # If the list was empty, set the tail to the new node.

    # Insert a new node at the end of the list.
    def insert_at_end(self, data):
        new_node = Node(data)  # Create a new node with the given data.
        new_node.prev = self.tail  # Set the new node's "previous" reference to the current tail.
        if self.tail:
            self.tail.next = new_node  # Set the next reference of the current tail to the new node.
        self.tail = new_node  # Update the tail to point to the new node.
        if not self.head:
            self.head = new_node  # If the list was empty, set the head to the new node.

    # Delete the first node in the list.
    def delete_at_beginning(self):
        if self.head:
            self.head = self.head.next  # Update the head to point to the next node.
```

```python
        if self.head:
            self.head.prev = None  # Set the previous reference of the new head to None.
        else:
            self.tail = None  # If there are no more nodes, set the tail to None.

    # Delete the last node in the list.
    def delete_at_end(self):
        if self.tail:
            self.tail = self.tail.prev  # Update the tail to point to the previous node.
            if self.tail:
                self.tail.next = None  # Set the next reference of the new tail to None.
            else:
                self.head = None  # If there are no more nodes, set the head to None.

    # Search for a specific value in the list.
    def search(self, target):
        current = self.head
        while current:
            if current.data == target:
                return True  # If the target is found, return True.
            current = current.next
        return False  # If the target is not found, return False.

    # Traverse and print the elements in the list from the head to the tail.
    def traverse_forward(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")  # Print the data in the current node.
            current = current.next
        print("None")  # Print "None" to indicate the end of the list.

    # Traverse and print the elements in the list from the tail to the head.
    def traverse_backward(self):
        current = self.tail
        while current:
            print(current.data, end=" <-> ")  # Print the data in the current node.
            current = current.prev
        print("None")  # Print "None" to indicate the end of the list.

    # Calculate the length of the list.
    def length(self):
        count = 0
        current = self.head
        while current:
            count += 1  # Increment the count for each node in the list.
            current = current.next
        return count  # Return the count as the length of the list.
```

PRINT ELEMENTS FROM HEAD TO TAIL AND FROM TAIL TO HEAD

```
# Define a DoublyLinkedList class to manage the doubly linked list.
class DoublyLinkedList:
    # ... (other methods as defined in the previous response)

    # Traverse and print the elements in the list from the head to the tail.
    def traverse_forward(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")  # Print the data in the current node.
            current = current.next
        print("None")  # Print "None" to indicate the end of the list.

    # Traverse and print the elements in the list from the tail to the head.
    def traverse_backward(self):
        current = self.tail
        while current:
            print(current.data, end=" <-> ")  # Print the data in the current node.
            current = current.prev
        print("None")  # Print "None" to indicate the end of the list.

# Create a new doubly linked list
dll = DoublyLinkedList()

# Insert elements into the doubly linked list
dll.insert_at_end(1)
dll.insert_at_end(2)
dll.insert_at_end(3)
dll.insert_at_end(4)

# Display the list from head to tail
print("Doubly Linked List (Head to Tail):")
dll.traverse_forward()

# Display the list from tail to head
print("\nDoubly Linked List (Tail to Head):")
dll.traverse_backward()
```

6. Bubble Sort

```
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all elements in the list
    for i in range(n):
        # Last i elements are already in place, so no need to check them
        for j in range(0, n - i - 1):
            # Compare adjacent elements
```

```
            if arr[j] > arr[j + 1]:
                # Swap if the element found is greater than the next element
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(my_list)
print("Sorted list:", my_list)
```

7. SELECTION SORT

```
def selection_sort(arr):
    n = len(arr)

    # Traverse through all elements in the list
    for i in range(n):
        # Find the minimum element in the unsorted portion
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        # Swap the minimum element with the first element in the unsorted portion
        arr[i], arr[min_index] = arr[min_index], arr[i]
        or use a temp variable
        temp = a[i]
        a[i] = a[min_index]
        a[min_index]= temp
# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
selection_sort(my_list)
print("Sorted list:", my_list)
```

8. Insertion Sort

```
def insertion_sort(arr):
    n = len(arr)

    # Traverse through all elements in the list
    for i in range(1, n):
        current_element = arr[i]
        j = i - 1

        # Move elements of the sorted portion to their correct positions
        # to make space for the current element
        while j >= 0 and current_element < arr[j]:
            arr[j + 1] = arr[j]
```

```
            j -= 1

        # Place the current element in its correct sorted position
        arr[j + 1] = current_element

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
insertion_sort(my_list)
print("Sorted list:", my_list)
```

9. MERGE SORT

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Split the input array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort both halves
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    # Merge the sorted halves into a single sorted list
    result = []
    i = j = 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            result.append(left_half[i])
            i += 1
        else:
            result.append(right_half[j])
            j += 1

    result.extend(left_half[i:])
    result.extend(right_half[j:])

    return result

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
sorted_list = merge_sort(my_list)
print("Sorted list:", sorted_list)
```

10. HEAP SORT

```
def heapify(arr, n, i):
    largest = i  # Initialize the largest as the root
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    # If the left child is larger than the root
    if left_child < n and arr[left_child] > arr[largest]:
        largest = left_child

    # If the right child is larger than the largest so far
    if right_child < n and arr[right_child] > arr[largest]:
        largest = right_child

    # If the largest is not the root, swap them
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        heapify(arr, n, largest)  # Recursively heapify the affected sub-tree

def heap_sort(arr):
    n = len(arr)

    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements from the heap one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap
        heapify(arr, i, 0)  # Heapify the reduced heap

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
heap_sort(my_list)
print("Sorted list:", my_list)
```

11. BINARY SEARCH

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2  # Calculate the midpoint

        if arr[mid] == target:  # If the midpoint element is equal to the target
            return mid  # Target found, return its index
```

```
        elif arr[mid] < target:  # If the midpoint element is less than the target
            left = mid + 1  # Search the right half
        else:
            right = mid - 1  # If the midpoint element is greater than the target, search the left half

    return -1  # Target not found in the array

# Example usage:
my_list = [11, 22, 25, 34, 64, 90]
target = 25
result = binary_search(my_list, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```

12. LINEAR SEARCH

```
def linear_search(arr, target):
    for index, element in enumerate(arr):
        if element == target:
            return index  # Target found, return its index
    return -1  # Target not found in the array

# Example usage:
my_list = [11, 22, 25, 34, 64, 90]
target = 25
result = linear_search(my_list, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the array.")
```