DSA

1)Write a Python Program to search a particular data from the given array of numbers using the Linear Search Method.

Algorithm for Linear Search

1. Start.
2. Take the array and the target element as input.
3. Loop through each element in the array:
    - If the current element is equal to the target, return the index and stop.
4. If the target element is not found, return a message saying "Not Found."
5. End.

Python Program for Linear Search

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return f"Element found at index {i}"
    return "Element not found"


# Example usage
numbers = [10, 20, 30, 40, 50]
key = int(input("Enter the number to search: "))
result = linear_search(numbers, key)
print(result)
```

2)Write a Python Program to search a particular data from the given array of strings using the Linear Search Method.

Algorithm for Linear Search (Strings)

1. Start.
2. Take an array of strings and the target string as input.
3. Loop through each element in the array:
    - If the current string is equal to the target, return its index and stop.
4. If the target string is not found, return a message saying "Not Found."
5. End.

```
def linear_search(arr, target):
   for i in range(len(arr)):
       if arr[i] == target:
           return f"Element found at index {i}"
   return "Element not found"


# Example usage
strings = ["apple", "banana", "cherry", "date", "elderberry"]
key = input("Enter the string to search: ")
result = linear_search(strings, key)
print(result)
```

3)Write a Python program to search a particular data from the given array of numbers using the Binary Search Method.

Algorithm for Binary Search

1. Start.
2. Take a sorted array and the target element as input.
3. Initialize two pointers:
   - left = 0 (start of the array)
   - right = length of the array - 1 (end of the array)
4. While left <= right:
   - Calculate mid = (left + right) // 2.
   - If the middle element is the target, return its index and stop.
   - If the middle element is less than the target, update left = mid + 1.
   - If the middle element is greater than the target, update right = mid - 1.
5. If the target is not found, return "Not Found."
6. End.

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

  while left <= right:
    mid = (left + right) // 2
    if arr[mid] == target:
      return f"Element found at index {mid}"
    elif arr[mid] < target:
      left = mid + 1
    else:
```

```
            right = mid - 1

    return "Element not found"

# Example usage
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90]
key = int(input("Enter the number to search: "))
result = binary_search(numbers, key)
print(result)
```

4)Write a Python Program to search a particular data from the given array of strings using the Binary Search Method.

Algorithm for Binary Search (Strings)

1. Start.
2. Take a sorted array of strings and the target string as input.
3. Initialize two pointers:
    - left = 0 (start of the array)
    - right = length of the array - 1 (end of the array)
4. While left <= right:
    - Calculate mid = (left + right) // 2.
    - If the middle string is the target, return its index and stop.
    - If the middle string is lexicographically less than the target, update left = mid + 1.
    - If the middle string is lexicographically greater than the target, update right = mid - 1.
5. If the target is not found, return "Not Found."
6. End.

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

  while left <= right:
    mid = (left + right) // 2
    if arr[mid] == target:
      return f"Element found at index {mid}"
    elif arr[mid] < target:
      left = mid + 1
    else:
```

```
            right = mid - 1

    return "Element not found"

# Example usage
strings = ["apple", "banana", "cherry", "date", "elderberry", "fig", "grape"]
key = input("Enter the string to search: ")
result = binary_search(strings, key)
print(result)
```

5)Write a Python Program to sort an array of numbers using the Bubble Sort Method

Algorithm for Bubble Sort

1. Start.
2. Take an array as input.
3. Set n to the length of the array.
4. Repeat the following steps for i from 0 to n-1:
    - Initialize swapped as False.
    - For each j from 0 to n-i-2:
        - If the current element arr[j] is greater than the next element arr[j+1], swap them.
        - Set swapped to True.
    - If no swaps were made, break the loop.
5. Print the sorted array.
6. End.

```
def bubble_sort(arr):
  n = len(arr)
  for i in range(n):
    swapped = False
    for j in range(0, n-i-1):
      if arr[j] > arr[j+1]:
        arr[j], arr[j+1] = arr[j+1], arr[j]
        swapped = True
    if not swapped:
      break


# Example usage
```

```
numbers = [64, 34, 25, 12, 22, 11, 90]

print("Original array:", numbers)

bubble_sort(numbers)

print("Sorted array:", numbers)
```

6)Write a Python Program to sort an array of strings using the Bubble Sort Method

Algorithm for Bubble Sort (Strings)

1. **Start.**
2. Take an array of strings as input.
3. Set n to the length of the array.
4. Repeat the following steps for i from 0 to n-1:
    - Initialize swapped as False.
    - For each j from 0 to n-i-2:
        - If the current string arr[j] is lexicographically greater than the next string arr[j+1], swap them.
        - Set swapped to True.
    - If no swaps were made, break the loop.
5. Print the sorted array.
6. **End.**

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break


# Example usage
```

```
strings = ["banana", "apple", "cherry", "date", "fig", "elderberry", "grape"]

print("Original array:", strings)

bubble_sort(strings)

print("Sorted array:", strings)
```

7)Write a Python Program to sort an array of numbers using the Quick Sort Method.

Algorithm for Quick Sort (Numbers)

1. **Start.**
2. If the length of the array is 1 or less, return the array (base case).
3. Choose a pivot element (middle element for simplicity).
4. Partition the array into three sub-arrays:
    - left: elements less than the pivot
    - middle: elements equal to the pivot
    - right: elements greater than the pivot
5. Recursively apply Quick Sort on the left and right sub-arrays.
6. Combine the sorted sub-arrays and return the result.
7. **End.**

```
def quick_sort(arr):
   if len(arr) <= 1:
      return arr
   pivot = arr[len(arr) // 2]
   left = [x for x in arr if x < pivot]
   middle = [x for x in arr if x == pivot]
   right = [x for x in arr if x > pivot]
   return quick_sort(left) + middle + quick_sort(right)


# Example usage
numbers = [64, 34, 25, 12, 22, 11, 90]
print("Original array:", numbers)
sorted_numbers = quick_sort(numbers)
```

```
print("Sorted array:", sorted_numbers)
```

8)Write a Python Program to sort an array of strings using the Quick Sort Method.

Algorithm for Quick Sort (Strings)

1. **Start.**
2. If the length of the array is 1 or less, return the array (base case).
3. Choose a pivot element (middle element for simplicity).
4. Partition the array into three sub-arrays:
   - left: strings lexicographically less than the pivot
   - middle: strings equal to the pivot
   - right: strings lexicographically greater than the pivot
5. Recursively apply Quick Sort on the left and right sub-arrays.
6. Combine the sorted sub-arrays and return the result.
7. **End.**

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)


# Example usage
```

```
strings = ["banana", "apple", "cherry", "date", "fig", "elderberry", "grape"]
print("Original array:", strings)
sorted_strings = quick_sort(strings)
print("Sorted array:", sorted_strings)
```

9)Write a Python Program to sort an array of numbers using the Merge Sort Method

Algorithm for Merge Sort (Numbers)

1. **Start.**
2. If the array has one element or is empty, return the array (base case).
3. Divide the array into two halves:
   - left = first half of the array
   - right = second half of the array
4. Recursively apply merge sort on left and right.
5. Merge the two sorted halves:
   - Compare elements from left and right one by one.
   - Append the smaller element to the result.
   - If one list is exhausted, append the remaining elements of the other list.
6. Return the merged and sorted array.
7. **End.**

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

    mid = len(arr) // 2
```

```
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

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

# Example usage
numbers = [64, 34, 25, 12, 22, 11, 90]
print("Original array:", numbers)
sorted_numbers = merge_sort(numbers)
```

print("Sorted array:", sorted_numbers)

Write a Python Program to implement

12)Write python program for heap sort in ascending order using maxheapify method

:

Algorithm:

1. **Build a Max Heap**:
   - Start from the last non-leaf node and apply the Max-Heapify method.
2. **Heap Sort**:
   - Swap the root (maximum element) of the heap with the last element.
   - Decrease the heap size by one and apply Max-Heapify on the root element.
   - Repeat this process until the heap size becomes 1.

Python Program:

```
# Function to perform Max-Heapify
def max_heapify(arr, n, i):
    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
```

```python
    # If right child is larger than largest
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        max_heapify(arr, n, largest)  # Recursively heapify the affected subtree

# Function to build a Max-Heap
def build_max_heap(arr):
    n = len(arr)
    # Start from the last non-leaf node and heapify each node
    for i in range(n//2 - 1, -1, -1):
        max_heapify(arr, n, i)

# Function to perform Heap Sort
def heap_sort(arr):
    n = len(arr)

    # Build a max heap
    build_max_heap(arr)

    # One by one extract elements from the heap
    for i in range(n-1, 0, -1):
```

```
        arr[i], arr[0] = arr[0], arr[i]  # Swap the root (max element) with the last element
        max_heapify(arr, i, 0)  # Heapify the root after the swap


# Main driver function to test the program
if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6, 7]
    print("Unsorted array:", arr)
    heap_sort(arr)
    print("Sorted array:", arr)
```

Explanation:

1. **max_heapify(arr, n, i)**: This function ensures the subtree rooted at index i is a max heap. It compares the root with its left and right children, and if necessary, swaps elements to maintain the max-heap property.
2. **build_max_heap(arr)**: This function builds a max heap from an unsorted array by calling max_heapify on all non-leaf nodes starting from the last non-leaf node.
3. **heap_sort(arr)**: This is the main sorting function. It first builds a max heap, then repeatedly swaps the root (maximum element) with the last element, reducing the heap size and calling max_heapify to maintain the heap.

Output for the given input:

Unsorted array: [12, 11, 13, 5, 6, 7]

Sorted array: [5, 6, 7, 11, 12, 13]

This program sorts the array in ascending order using the Heap Sort algorithm with the Max-Heapify method.

13)Write a Python Program to implement a Singly Linked List with the following operations: (i) Insert at beginning, (ii) Insert at end, (iii) Insert after, (iv) Delete (v) Search, (vii) Display.

 Insert at Beginning:

- Create a new node and set its next pointer to the current head.
- Set the head to the new node.

 Insert at End:

- Traverse the list to find the last node.
- Set the last node's next pointer to the new node.

 Insert After:

- Traverse the list to find the given node.
- Create a new node and set its next pointer to the next node of the given node.
- Set the next pointer of the given node to the new node.

 Delete:

- Traverse the list to find the node to be deleted.
- Update the previous node’s next pointer to the node to be deleted's next pointer.

 Search:

- Traverse the list to find a node with the given value.

 Display:

- Traverse the list from the head and print each node's data.

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


class SinglyLinkedList:
```

```
def __init__(self):
    self.head = None

# (i) Insert at Beginning
def insert_at_beginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node

# (ii) Insert at End
def insert_at_end(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        return
    last = self.head
    while last.next:
        last = last.next
    last.next = new_node

# (iii) Insert After a given node
def insert_after(self, prev_data, data):
    current = self.head
    while current:
        if current.data == prev_data:
```

```
            new_node = Node(data)
            new_node.next = current.next
            current.next = new_node
            return
        current = current.next
    print("Node with data", prev_data, "not found.")

# (iv) Delete a node
def delete(self, key):
    current = self.head

    # If the node to be deleted is the head node
    if current and current.data == key:
        self.head = current.next
        current = None
        return

    # Search for the node to be deleted
    prev = None
    while current:
        if current.data == key:
            break
        prev = current
        current = current.next
```

```
        # If the node is not found
        if not current:
            print("Node with data", key, "not found.")
            return

        # Unlink the node from the linked list
        prev.next = current.next
        current = None

    # (v) Search for a node
    def search(self, key):
        current = self.head
        while current:
            if current.data == key:
                return True
            current = current.next
        return False

    # (vi) Display the list
    def display(self):
        current = self.head
        if not current:
            print("List is empty.")
            return
        while current:
```

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

            current = current.next

        print("None")


# Main function to test the linked list operations

if __name__ == "__main__":

    linked_list = SinglyLinkedList()


    # Insert at beginning

    linked_list.insert_at_beginning(10)

    linked_list.insert_at_beginning(20)

    linked_list.insert_at_beginning(30)

    linked_list.display()  # Output: 30 -> 20 -> 10 -> None


    # Insert at end

    linked_list.insert_at_end(40)

    linked_list.insert_at_end(50)

    linked_list.display()  # Output: 30 -> 20 -> 10 -> 40 -> 50 -> None


    # Insert after a node

    linked_list.insert_after(20, 25)

    linked_list.display()  # Output: 30 -> 20 -> 25 -> 10 -> 40 -> 50 -> None


    # Delete a node

    linked_list.delete(10)
```

```
linked_list.display()  # Output: 30 -> 20 -> 25 -> 40 -> 50 -> None


# Search for a node
print(linked_list.search(25))  # Output: True
print(linked_list.search(100))  # Output: False
```

14)Write a Python Program to implement a Doubly Linked List with the following operations: (i) Insert at beginning, (ii) Insert at end, (iii) Insert after, (iv) Delete (v) Search, (vii) Display

Algorithm for Doubly Linked List Operations:

1. **Insert at Beginning**:
   - Create a new node.
   - Set the new node's next pointer to the current head.
   - Set the current head's previous pointer to the new node.
   - Update the head to the new node.
2. **Insert at End**:
   - Create a new node.
   - Traverse the list to find the last node.
   - Set the last node's next pointer to the new node.
   - Set the new node's previous pointer to the last node.
3. **Insert After**:
   - Traverse the list to find the node after which insertion is needed.
   - Create a new node and set its next pointer to the next node of the current node.
   - Set the next node's previous pointer to the new node.

- Set the current node's next pointer to the new node and the new node's previous pointer to the current node.

4. **Delete**:
   - Traverse the list to find the node to be deleted.
   - Update the previous node's next pointer and the next node's previous pointer to remove the node.
5. **Search**:
   - Traverse the list, and compare each node's data with the search key.
   - Return True if the node is found, otherwise return False.
6. **Display**:
   - Traverse the list from the head, and print each node's data.

Python Program for Doubly 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


    # (i) Insert at Beginning
    def insert_at_beginning(self, data):
        new_node = Node(data)
```

```
    new_node.next = self.head
    if self.head:
        self.head.prev = new_node
    self.head = new_node

# (ii) Insert at End
def insert_at_end(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        return
    last = self.head
    while last.next:
        last = last.next
    last.next = new_node
    new_node.prev = last

# (iii) Insert After a given node
def insert_after(self, prev_data, data):
    current = self.head
    while current:
        if current.data == prev_data:
            new_node = Node(data)
            new_node.next = current.next
            if current.next:
```

```
                current.next.prev = new_node
            current.next = new_node
            new_node.prev = current
            return
        current = current.next
    print("Node with data", prev_data, "not found.")


# (iv) Delete a node
def delete(self, key):
    current = self.head
    if current and current.data == key:
        if current.next:
            current.next.prev = None
        self.head = current.next
        current = None
        return
    while current:
        if current.data == key:
            break
        current = current.next
    if not current:
        print("Node with data", key, "not found.")
        return
    if current.next:
        current.next.prev = current.prev
```

```
            if current.prev:
                current.prev.next = current.next
            current = None

        # (v) Search for a node
        def search(self, key):
            current = self.head
            while current:
                if current.data == key:
                    return True
                current = current.next
            return False

        # (vi) Display the list
        def display(self):
            current = self.head
            if not current:
                print("List is empty.")
                return
            while current:
                print(current.data, end=" <-> ")
                current = current.next
            print("None")

# Main function to test the linked list operations
```

```
if __name__ == "__main__":
    dll = DoublyLinkedList()

    # Insert at beginning
    dll.insert_at_beginning(10)
    dll.insert_at_beginning(20)
    dll.insert_at_beginning(30)
    dll.display()

    # Insert at end
    dll.insert_at_end(40)
    dll.insert_at_end(50)
    dll.display()

    # Insert after a node
    dll.insert_after(20, 25)
    dll.display()

    # Delete a node
    dll.delete(10)
    dll.display()

    # Search for a node
    print(dll.search(25))  # Output: True
    print(dll.search(100))  # Output: False
```

15)Write a Python Program to implement a Circular Singly Linked List with the following operations: (i) Insert at beginning, (ii) Insert at end, (iii) Insert after, (iv) Delete (v) Search, (vii) Display

Algorithm for Circular Singly Linked List Operations:

1. **Insert at Beginning**:
    - Create a new node.
    - Set the new node's next pointer to the head.
    - Traverse to the last node and set its next pointer to the new node.
    - Update the head to the new node.
2. **Insert at End**:
    - Create a new node.
    - Traverse to the last node.
    - Set the last node's next pointer to the new node.
    - Set the new node's next pointer to the head.
3. **Insert After**:
    - Traverse the list to find the node after which insertion is needed.
    - Create a new node and set its next pointer to the next node of the current node.
    - Set the current node's next pointer to the new node.
4. **Delete**:
    - Traverse the list to find the node to be deleted.
    - Update the previous node's next pointer to skip the node to be deleted.
5. **Search**:
    - Traverse the list, and compare each node's data with the search key.
    - Return True if the node is found, otherwise return False.
6. **Display**:
    - Traverse the list from the head, and print each node's data until we reach the head again.

Python Program for Circular Singly Linked List:

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


class CircularSinglyLinkedList:
    def __init__(self):
        self.head = None


    # (i) Insert at Beginning
    def insert_at_beginning(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            new_node.next = self.head
        else:
            new_node.next = self.head
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            self.head = new_node
```

```
# (ii) Insert at End
def insert_at_end(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        new_node.next = self.head
    else:
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = new_node
        new_node.next = self.head

# (iii) Insert After a given node
def insert_after(self, prev_data, data):
    current = self.head
    while current:
        if current.data == prev_data:
            new_node = Node(data)
            new_node.next = current.next
            current.next = new_node
            return
        current = current.next
        if current == self.head:
            break
```

```
        print("Node with data", prev_data, "not found.")

    # (iv) Delete a node
    def delete(self, key):
        current = self.head
        prev = None
        if current and current.data == key:
            if current.next == self.head:  # Only one node in the list
                self.head = None
            else:
                while current.next != self.head:
                    current = current.next
                current.next = self.head.next
                self.head = self.head.next
            return
        while current:
            if current.data == key:
                break
            prev = current
            current = current.next
            if current == self.head:
                break
        if not current:
            print("Node with data", key, "not found.")
            return
```

```
    prev.next = current.next

# (v) Search for a node
def search(self, key):
    current = self.head
    if not current:
        return False
    while current:
        if current.data == key:
            return True
        current = current.next
        if current == self.head:
            break
    return False

# (vi) Display the list
def display(self):
    current = self.head
    if not current:
        print("List is empty.")
        return
    while current:
        print(current.data, end=" -> ")
        current = current.next
        if current == self.head:
```

```
            break
        print("Head")

# Main function to test the linked list operations
if __name__ == "__main__":
    cll = CircularSinglyLinkedList()

    # Insert at beginning
    cll.insert_at_beginning(10)
    cll.insert_at_beginning(20)
    cll.insert_at_beginning(30)
    cll.display()

    # Insert at end
    cll.insert_at_end(40)
    cll.insert_at_end(50)
    cll.display()

    # Insert after a node
    cll.insert_after(20, 25)
    cll.display()

    # Delete a node
    cll.delete(10)
    cll.display()
```

```
# Search for a node
print(cll.search(25))  # Output: True
print(cll.search(100))  # Output: False
```

15. Circular Singly Linked List Operations

Question: Write a Python Program to implement a Circular Singly Linked List with the following operations:

- Insert at beginning
- Insert at end
- Insert after
- Delete
- Search
- Display

Brief Algorithm Steps:

1. Define a Node class with data and next pointers.
2. Implement CircularSinglyLinkedList class.
3. Define methods:
   - **Insert at beginning**: Add a new node at the start.
   - **Insert at end**: Add a new node at the end.
   - **Insert after**: Insert a node after a specified node.
   - **Delete**: Remove a node.
   - **Search**: Search for a node by value.
   - **Display**: Traverse and print the list.

Brief Code:

python

Copy code

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


class CircularSinglyLinkedList:
    def __init__(self):
        self.head = None
```

```
def insert_beginning(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        self.head.next = self.head
    else:
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = new_node
        new_node.next = self.head
        self.head = new_node

def insert_end(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        self.head.next = self.head
    else:
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = new_node
        new_node.next = self.head

def display(self):
    if not self.head:
        print("List is empty")
        return
    temp = self.head
    while True:
```

```python
            print(temp.data, end=" -> ")
            temp = temp.next
            if temp == self.head:
                break
        print("... (circular)")


# Example usage
circular_list = CircularSinglyLinkedList()
circular_list.insert_beginning(10)
circular_list.insert_end(20)
circular_list.display()
```

---

16. Circular Doubly Linked List Operations

Question: Write a Python Program to implement a Circular Doubly Linked List with the following operations:

- Insert at beginning
- Insert at end
- Insert after
- Delete
- Search
- Display

Brief Algorithm Steps:

1. Define a Node class with data, prev, and next pointers.
2. Implement CircularDoublyLinkedList class.
3. Define methods:
   - **Insert at beginning**: Add a node at the start.
   - **Insert at end**: Add a node at the end.
   - **Insert after**: Insert a node after a given node.
   - **Delete**: Remove a node.
   - **Search**: Find a node by value.
   - **Display**: Traverse and print the list in both directions.

Brief Code:

python

Copy code

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

class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None

    def insert_beginning(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            self.head.next = self.head
            self.head.prev = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.prev = temp
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def insert_end(self, data):
        new_node = Node(data)
```

```
        if not self.head:
            self.head = new_node
            self.head.next = self.head
            self.head.prev = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.prev = temp
            new_node.next = self.head
            self.head.prev = new_node

    def display(self):
        if not self.head:
            print("List is empty")
            return
        temp = self.head
        while True:
            print(temp.data, end=" <-> ")
            temp = temp.next
            if temp == self.head:
                break
        print("... (circular)")

# Example usage
doubly_circular_list = CircularDoublyLinkedList()
doubly_circular_list.insert_beginning(10)
doubly_circular_list.insert_end(20)
doubly_circular_list.display()
```

17. PUSH and POP Operations on a Stack using an Array

Question: Write a Python Program to perform PUSH (with lower alphabet) and POP (equivalent upper alphabet) operations on a Stack using an Array.

Brief Algorithm Steps:

1. Create a stack using a list.
2. Define a method push to insert elements into the stack.
3. Define a method pop to remove elements from the stack and convert them to uppercase.
4. Display the stack after each operation.

Brief Code:

python

Copy code

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, data):
        self.stack.append(data)

    def pop(self):
        if self.stack:
            return self.stack.pop().upper()
        return None

# Example usage
stack = Stack()
stack.push('a')
stack.push('b')
print(stack.pop())  # Output: 'B'
```

---

18. PUSH and POP Operations on a Stack using a Linked List

Question: Write a Python Program to perform PUSH and POP operations on a Stack using a Linked List.

Brief Algorithm Steps:

1. Create a Node class with data and next pointers.
2. Implement a Stack class using the linked list.
3. Define methods push to add elements to the stack and pop to remove and return the element.

Brief Code:

python

Copy code

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

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

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

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

# Example usage
stack = Stack()
```

```
stack.push('a')
stack.push('b')
print(stack.pop())  # Output: 'b'
```

---

19. Print a String in Reverse using Recursion

Question: Write a Python program to print a given string in reverse using recursion.

Brief Algorithm Steps:

1. Base case: If the string is empty, return.
2. Otherwise, print the last character and call the function recursively with the remaining string.

Brief Code:

python

Copy code

```
def reverse_string(s):
    if len(s) == 0:
        return
    print(s[-1], end="")
    reverse_string(s[:-1])


# Example usage
reverse_string("hello")  # Output: 'olleh'
```

---

20. Convert Infix to Postfix

Question: Write a Python program to convert a given infix expression to postfix.

Brief Algorithm Steps:

1. Use a stack to store operators.
2. Traverse the infix expression:
    - If operand, add it directly to the result.
    - If operator, pop from stack and add to result if higher precedence, then push the current operator.
3. Pop all remaining operators from the stack.

Brief Code:

python

Copy code

```
def infix_to_postfix(expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    stack = []
    result = []
    for char in expression:
        if char.isalnum():
            result.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                result.append(stack.pop())
            stack.pop()
        else:
            while stack and stack[-1] != '(' and precedence[char] <= precedence[stack[-1]]:
                result.append(stack.pop())
            stack.append(char)
    while stack:
        result.append(stack.pop())
    return ''.join(result)

# Example usage
print(infix_to_postfix("a+b*(c^d-e)"))  # Output: 'abcd^e-*+'
```

---

21. Convert Infix to Prefix

Question: Write a Python program to convert a given infix expression to prefix.

Brief Algorithm Steps:

1. Reverse the infix expression.
2. Replace (with) and vice versa.

3. Convert the modified expression to postfix.
4. Reverse the postfix expression to get the prefix.

Brief Code:

python

Copy code

```
def infix_to_prefix(expression):
    expression = expression[::-1]
    expression = expression.replace('(', 'temp').replace(')', '(').replace('temp', ')')
    return infix_to_postfix(expression)[::-1]

# Example usage
print(infix_to_prefix("a+b*(c^d-e)"))  # Output: '+a*b^-cd+e'
```

Yeh raha complete solution for questions 22 to 30:

---

22. Evaluate Postfix Expression

Question: Write a Python program to evaluate a given/postfix expression.

Brief Algorithm Steps:

1. Create an empty stack.
2. Traverse through the expression:
    - If operand, push it onto the stack.
    - If operator, pop two operands, apply the operator, and push the result back.
3. After processing, the stack will contain the result.

Brief Code:

python

Copy code

```
def evaluate_postfix(expression):
    stack = []
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
```

```
        else:
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                stack.append(a + b)
            elif char == '-':
                stack.append(a - b)
            elif char == '*':
                stack.append(a * b)
            elif char == '/':
                stack.append(a / b)
    return stack.pop()

# Example usage
print(evaluate_postfix("23*5+"))  # Output: 11
```

---

23. Traverse Singly Linked List in Reverse Using Recursion

Question: Write a Python program to create a Singly Linked List and traverse it in reverse order using recursion.

Brief Algorithm Steps:

1. Recursively traverse the list until the last node.
2. In the recursion unwinding phase, print the node data.

Brief Code:

python

Copy code

```
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 reverse_traverse(self, node):
        if node is None:
            return
        self.reverse_traverse(node.next)
        print(node.data, end=" ")

# Example usage
ll = SinglyLinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)
ll.reverse_traverse(ll.head)  # Output: 10 20 30
```

---

24. INSERT and DELETE Operations on a Linear Queue Using an Array

Question: Write a Python Program to perform INSERT and DELETE operations on a Linear Queue using an Array.

Brief Algorithm Steps:

1. **Insert**: Add an element at the rear end and increment the rear pointer.
2. **Delete**: Remove the element from the front and increment the front pointer.

Brief Code:

python

Copy code

```
class Queue:
```

```python
    def __init__(self, size):
        self.queue = [None] * size
        self.front = self.rear = -1
        self.size = size

    def insert(self, data):
        if self.rear == self.size - 1:
            print("Queue is full")
        elif self.front == -1:
            self.front = self.rear = 0
            self.queue[self.rear] = data
        else:
            self.rear += 1
            self.queue[self.rear] = data

    def delete(self):
        if self.front == -1:
            print("Queue is empty")
        else:
            print("Deleted:", self.queue[self.front])
            self.front += 1

# Example usage
queue = Queue(5)
queue.insert(10)
queue.insert(20)
queue.delete()  # Output: Deleted: 10
```

---

25. INSERT and DELETE Operations on a Linear Queue Using a Linked List

Question: Write a Python Program to perform INSERT and DELETE operations on a Linear Queue using a Linked List.

Brief Algorithm Steps:

1. **Insert**: Create a new node, and if the queue is empty, set both front and rear to this node. Otherwise, insert at the rear.
2. **Delete**: Remove the node from the front and adjust the front pointer.

Brief Code:

python

Copy code

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


class Queue:
    def __init__(self):
        self.front = self.rear = None


    def insert(self, data):
        new_node = Node(data)
        if not self.rear:
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node


    def delete(self):
        if not self.front:
            print("Queue is empty")
        else:
            print("Deleted:", self.front.data)
            self.front = self.front.next
```

```
# Example usage
queue = Queue()
queue.insert(10)
queue.insert(20)
queue.delete()  # Output: Deleted: 10
```

---

26. Binary Search Tree (BST) and In-Order Traversal

Question: Write a Python Program to implement a Binary Search Tree (BST) and traverse it in In-Order.

Brief Algorithm Steps:

1. Create a Node class with data, left, and right.
2. Create a BST class with methods to insert nodes and traverse in in-order.
3. In in-order traversal, visit the left subtree, then the node, and finally the right subtree.

Brief Code:

python

Copy code

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


class BST:
    def __init__(self):
        self.root = None


    def insert(self, data):
        if not self.root:
            self.root = Node(data)
        else:
            self._insert(self.root, data)
```

```
    def _insert(self, node, data):
        if data < node.data:
            if node.left:
                self._insert(node.left, data)
            else:
                node.left = Node(data)
        else:
            if node.right:
                self._insert(node.right, data)
            else:
                node.right = Node(data)

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.data, end=" ")
            self.inorder(node.right)

# Example usage
bst = BST()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.inorder(bst.root)  # Output: 30 50 70
```

---

27. Binary Search Tree (BST) Pre-Order and Post-Order Traversal

Question: Write a Python Program to traverse a Binary Search Tree (BST) in Pre-Order and Post-Order.

Brief Algorithm Steps:

1. Pre-Order: Visit the node first, then traverse the left subtree, followed by the right.
2. Post-Order: Traverse the left subtree first, then the right, and visit the node last.

Brief Code:

python

Copy code

```
class BST:
    # Same insert and inorder as previous code

    def preorder(self, node):
        if node:
            print(node.data, end=" ")
            self.preorder(node.left)
            self.preorder(node.right)

    def postorder(self, node):
        if node:
            self.postorder(node.left)
            self.postorder(node.right)
            print(node.data, end=" ")

# Example usage
bst = BST()
bst.insert(50)
bst.insert(30)
bst.insert(70)
print("Pre-Order:")
bst.preorder(bst.root)  # Output: 50 30 70
print("\nPost-Order:")
bst.postorder(bst.root)  # Output: 30 70 50
```

---

28. Binary Search Tree (BST) Pre-Order Traversal

Question: Write a Python Program to implement a Binary Search Tree (BST) and traverse it in Pre-Order.

Brief Algorithm Steps:

1. Same as the previous, but only do the Pre-Order traversal.

Brief Code:

python

Copy code

```
# Use same BST code as above with the preorder method
```

---

29. Breadth First Search (BFS)

Question: Write a Python Program to implement Breadth First Search.

Brief Algorithm Steps:

1. Use a queue to explore nodes level by level.
2. Visit each node, add it to the queue, and process all the nodes in the queue.

Brief Code:

python

Copy code

```
from collections import deque


def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex, end=" ")
            queue.extend(graph[vertex] - visited)


# Example usage
graph = {
    1: {2, 3},
    2: {4, 5},
```

```
    3: {6, 7},
    4: set(),
    5: set(),
    6: set(),
    7: set()
}
bfs(graph, 1)  # Output: 1 2 3 4 5 6 7
```

---

30. Depth First Search (DFS)

Question: Write a Python Program to implement Depth First Search.

Brief Algorithm Steps:

1. Use a stack to explore the graph nodes.
2. Start from the source node, mark it as visited, and push it onto the stack.
3. Explore the neighbors of the node recursively, marking each node as visited as you go deeper.
4. Backtrack when all neighbors are visited.

Brief Code:

python

Copy code

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)


# Example usage
graph = {
    1: {2, 3},
```

```
    2: {4, 5},
    3: {6, 7},
    4: set(),
    5: set(),
    6: set(),
    7: set()
}

dfs(graph, 1)  # Output: 1 2 4 5 3 6 7
```