DSA Laboratory

1 Array Operations:

Implement a function to find the maximum element in an array.

```
def find_max(arr):
   if len(arr) == 0:
      return None
   max_element = arr[0]
   for i in range(1, len(arr)):
      if arr[i] > max_element:
         max_element = arr[i]
   return max_element

# Example usage
arr = [5, 2, 9, 1, 7, 6, 3]
print("Maximum element:", find_max(arr))
```

Steps:

1. define a function `find_max` that takes an array as input.
2. first check if the array is empty. If so, return None.
3. initialize `max_element` with the first element of the array.
4. iterate through the array starting from the second element (index 1).
5. For each element, compare it with the current `max_element`.
6. If the current element is greater, update `max_element`.
7. After the loop, return the `max_element`.
8. In the example usage, create an array and call the function to find its maximum element.

Write a program to reverse an array in-place.

```
def reverse_array(arr):
   left = 0
   right = len(arr) - 1
   while left < right:
      arr[left], arr[right] = arr[right], arr[left]
      left += 1
      right -= 1

# Example usage
arr = [1, 2, 3, 4, 5]
print("Original array:", arr)
reverse_array(arr)
print("Reversed array:", arr)
```

Steps:

1. define a function reverse_array that takes an array as input.
2. initialize two pointers: left at the start of the array and right at the end.
3. enter a while loop that continues as long as left is less than right.
4. Inside the loop, swap the elements at left and right indices.
5. then move left one step to the right and right one step to the left.
6. The loop continues until the pointers meet in the middle.
7. In the example usage, create an array, print it, reverse it, and print the result.

2. Linked List Manipulation:

Singly linked list with a function to reverse it

```
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 reverse(self):
        prev = None
        current = self.head
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        self.head = prev

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Example usage
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(4)

print("Original linked list:")
ll.display()

ll.reverse()
print("Reversed linked list:")
```

Steps:

1. define a Node class to represent each element in the linked list.
2. define a LinkedList class with methods to append nodes and reverse the list.
3. The append method adds a new node to the end of the list.
4. The reverse method reverses the list by changing the direction of pointers:
    - use three pointers: prev, current, and next_node.
    - iterate through the list, reversing each pointer.
    - After the loop, set the head to the last node (now first).
5. The display method prints the list.
6. In the example usage, create a list, display it, reverse it, and display it again.

```
ll.display()
```

Function to detect a cycle in a linked list:

```
def has_cycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    while slow != fast:
        if not fast or not fast.next:
            return False
        slow = slow.next
        fast = fast.next.next

    return True

# Example usage
# Create a linked list with a cycle
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(4)

# Create a cycle by connecting the last node to the second node
last_node = ll.head
while last_node.next:
    last_node = last_node.next
last_node.next = ll.head.next

print("Has cycle:", has_cycle(ll.head))
```

Steps:

1. define a function has_cycle that takes the head of a linked list as input.
2. use two pointers: slow (moves one step at a time) and fast (moves two steps at a time).
3. enter a loop that continues until slow and fast meet or fast reaches the end.
4. If fast reaches the end, return False (no cycle).
5. If slow and fast meet, return True (cycle detected).
6. In the example usage, create a linked list with a cycle and test the function.

3. Stacks and Queues:

Stack implementation using arrays:

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

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None
```

1. define a Stack class using a list to store items.
2. push adds an item to the end of the list.
3. pop removes and returns the last item from the list.
4. peek returns the last item without removing it.
5. is_empty checks if the stack is empty.
6. size returns the number of items in the stack.

```
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None

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

    def size(self):
        return len(self.items)

# Example usage
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)

print("Stack size:", stack.size())
print("Top element:", stack.peek())
print("Popped element:", stack.pop())
print("Stack size after pop:", stack.size())
```

7. In the example usage, create a stack, push elements, and demonstrate various operations.

Queue using two stacks:

```
class Queue:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []

    def enqueue(self, item):
        self.stack1.append(item)

    def dequeue(self):
        if not self.stack2:
            if not self.stack1:
                return None
            while self.stack1:

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

    def is_empty(self):
        return len(self.stack1) == 0 and
len(self.stack2) == 0

    def size(self):
        return len(self.stack1) +
len(self.stack2)

# Example usage
```

Steps:

1. define a Queue class using two stacks: stack1 for enqueue and stack2 for dequeue.
2. enqueue adds an item to stack1.
3. dequeue works as follows:

a) If stack2 is empty, transfer all elements from stack1 to stack2 (reversing their order).

b) then pop and return the top element from stack2.

4. is_empty checks if both stacks are empty.
5. size returns the total number of elements in both stacks.
6. In the example usage, create a queue, enqueue elements, and demonstrate dequeue operation.

queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
 	
print("Queue size:", queue.size())
print("Dequeued element:", queue.dequeue())
print("Queue size after dequeue:", queue.size())	