Practical - 1

Write a program to implement a singly linked list as an ADT that supports the following operations:

I. Insert an element x at the beginning of the singly linked list.
II. Insert an element x at i th position in the singly linked list.
III. Remove an element from the beginning of the singly linked list.
IV. Remove an element from ith position in the singly linked list.
V. Search for an element x in the singly linked list and return its pointer.

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

 # Insert an element x at the beginning of the singly linked list.

```
def insert_at_beginning(self, x):
     new_node = Node(x)
     new_node.next = self.head
     self.head = new_node
```

 # Insert an element x at i-th position in the singly linked list.

```
     def insert_at_position(self, x, i):
        new_node = Node(x)
        if i == 0:
            self.insert_at_beginning(x)
            return
        temp = self.head
        count = 0
        while temp is not None and count < i - 1:
            temp = temp.next
            count += 1
        if temp is None:
            print("Index out of bounds")
            return
```

```
        new_node.next = temp.next
        temp.next = new_node
```

 # Remove an element from the beginning of the singly linked list.

```
  def remove_from_beginning(self):
        if self.head is None:
            print("List is empty")
            return
        self.head = self.head.next
```

 # Remove an element from i-th position in the singly linked list.

```
  def remove_from_position(self, i):
        if self.head is None:
            print("List is empty")
            return
        if i == 0:
            self.head = self.head.next
            return
        temp = self.head
        count = 0
        while temp is not None and count < i - 1:
            temp = temp.next
            count += 1
        if temp is None or temp.next is None:
            print("Index out of bounds")
            return
        temp.next = temp.next.next
```

 # Search for an element x in the singly linked list and return its pointer.

```
def search(self, x):
        temp = self.head
        while temp:
            if temp.data == x:
                return temp
            temp = temp.next
        return None

def display(self):
        temp = self.head
```

```
        elements = []
        while temp:
            elements.append(str(temp.data))
            temp = temp.next
        print(" -> ".join(elements) + " -> None")
```

 # Example usage:

```
if __name__ == "__main__":
    ll = SinglyLinkedList()
    ll.insert_at_beginning(10)
    ll.insert_at_beginning(20)
    ll.insert_at_position(30, 1)
    ll.display()

    search_result = ll.search(10)
    if search_result:
        print("Search 10: Found")
    else:
        print("Search 10: Not Found")

    ll.remove_from_position(1)
    ll.display()
```

Output:

```
20 -> 30 -> 10 -> None
Search 10: Found
20 -> 10 -> None
```

Practical - 2

Write a program to implement a doubly linked list as an ADT that supports the following operations
i. Insert an element x at the beginning of the doubly linked list.
ii. Insert an element x at the end of the doubly linked list.
iii. Remove an element from the beginning of the doubly linked list.
iv. Remove an element from the end of the 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
        self.tail = None
```

 # Insert an element x at the beginning of the doubly linked list.

```
    def insert_at_beginning(self, x):
        new_node = Node(x)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
```

 # Insert an element x at the end of the doubly linked list.

```
    def insert_at_end(self, x):
        new_node = Node(x)
        if self.tail is None:
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
```

```
            self.tail = new_node
```

 # Remove an element from the beginning of the doubly linked list.

```
    def remove_from_beginning(self):
        if self.head is None:
            print("List is empty")
            return
        if self.head == self.tail:
            self.head = self.tail = None
        else:
            self.head = self.head.next
            self.head.prev = None
```

 # Remove an element from the end of the doubly linked list.

```
    def remove_from_end(self):
        if self.tail is None:
            print("List is empty")
            return
        if self.head == self.tail:
            self.head = self.tail = None
        else:
            self.tail = self.tail.prev
            self.tail.next = None

 def display(self):
        temp = self.head
        elements = []
        while temp:
            elements.append(str(temp.data))
            temp = temp.next
        print(" <-> ".join(elements) + " <-> None")
```

 # Example usage:

```
if __name__ == "__main__":
    dll = DoublyLinkedList()
    dll.insert_at_beginning(10)
    dll.insert_at_beginning(20)
    dll.insert_at_end(30)
    dll.display()
```

```
dll.remove_from_beginning()
dll.display()

dll.remove_from_end()
dll.display()
```

Output:

```
20 <-> 10 <-> 30 <-> None
10 <-> 30 <-> None
10 <-> None
```

Practical - 3

Write a program to implement a circular linked list as an ADT that supports the following operations:
i. Insert an element x in the list.
ii. Remove an element from the list.
iii. Search for an element x in the list and return its pointer.

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

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

 # I. Insert an element x in the list.

```
    def insert(self, x):
        new_node = Node(x)
        if self.head is None:
            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
```

 # Remove an element from the list.

```
    def remove(self, x):
        if self.head is None:
            print("List is empty")
            return
        temp = self.head
        prev = None
        while True:
```

```
            if temp.data == x:
                if prev is None:
                    if temp.next == self.head:
                        self.head = None
                    else:
                        last = self.head
                        while last.next != self.head:
                            last = last.next
                        self.head = temp.next
                        last.next = self.head
                else:
                    prev.next = temp.next
                return
            prev = temp
            temp = temp.next
            if temp == self.head:
                break
        print("Element not found")
```

 # Search for an element x in the list and return its pointer.

```
    def search(self, x):
        if self.head is None:
            return None
        temp = self.head
        while True:
            if temp.data == x:
                return temp
            temp = temp.next
            if temp == self.head:
                break
        return None

    def display(self):
        if self.head is None:
            print("List is empty")
            return
        temp = self.head
        elements = []
        while True:
            elements.append(str(temp.data))
```

```python
            temp = temp.next
            if temp == self.head:
                break
        print(" -> ".join(elements) + " -> (head)")
```

 # Example usage:

```python
if __name__ == "__main__":
    cll = CircularLinkedList()
    cll.insert(10)
    cll.insert(20)
    cll.insert(30)
    cll.display()

    cll.remove(20)
    cll.display()

    search_result = cll.search(30)
    if search_result:
        print("Search 30: Found")
    else:
        print("Search 30: Not Found")
```

OUTPUT:

```
10 -> 20 -> 30 -> (head)
10 -> 30 -> (head)
Search 30: Found
```

Practical - 4

Implement Stack as an ADT and use it to evaluate a prefix/postfix expression.

```
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

    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)
```

 # Function to evaluate a postfix expression

```
def evaluate_postfix(expression):
    stack = Stack()
    for char in expression:
        if char.isdigit():
            stack.push(int(char))
        else:
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                stack.push(a + b)
            elif char == '-':
                stack.push(a - b)
            elif char == '*':
```

```
                stack.push(a * b)
            elif char == '/':
                stack.push(a / b)
    return stack.pop()
```

#Function to evaluate a prefix expression

```
def evaluate_prefix(expression):
    stack = Stack()
    for char in reversed(expression):
        if char.isdigit():
            stack.push(int(char))
        else:
            a = stack.pop()
            b = stack.pop()
            if char == '+':
                stack.push(a + b)
            elif char == '-':
                stack.push(a - b)
            elif char == '*':
                stack.push(a * b)
            elif char == '/':
                stack.push(a / b)
    return stack.pop()
```

 # Example usage:

```
if __name__ == "__main__":
    postfix_expr = "53+62/*35*+"
    prefix_expr = "+*23/84"

    print("Postfix Evaluation:", 
evaluate_postfix(postfix_expr))
    print("Prefix Evaluation:", evaluate_prefix(prefix_expr))
```

Output:

```
Postfix Evaluation: 39.0
Prefix Evaluation: 8.0
```

Practical - 5

Implement Queue as an ADT.

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

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

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        return None

    def front(self):
        if not self.is_empty():
            return self.items[0]
        return None

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

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

if __name__ == "__main__":
    q = Queue()
    q.enqueue(10)
    q.enqueue(20)
    q.enqueue(30)
    print("Front element:", q.front())
    print("Dequeued element:", q.dequeue())
    print("Queue size:", q.size())
```

```
Front element: 10
Dequeued element: 10
Queue size: 2
```

Practical - 6

Write a program to implement a Binary Search Tree as an ADT which supports the following operations:
i. Insert an element x
ii. Delete an element x
iii. Search for an element x in the BST
iv. Display the elements of the BST in preorder, inorder, and postorder traversal.

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

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

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, root, key):
        if not root:
            return Node(key)
        if key < root.key:
            root.left = self._insert(root.left, key)
        else:
            root.right = self._insert(root.right, key)
        return root
   def delete(self, key):
        self.root = self._delete(self.root, key)

    def _delete(self, root, key):
        if not root:
            return root
        if key < root.key:
            root.left = self._delete(root.left, key)
        elif key > root.key:
            root.right = self._delete(root.right, key)
```

```
        else:
            if not root.left:
                return root.right
            if not root.right:
                return root.left
            temp = self._min_value_node(root.right)
            root.key = temp.key
            root.right = self._delete(root.right,
temp.key)
        return root

    def _min_value_node(self, node):
        while node.left:
            node = node.left
        return node

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, root, key):
        if not root or root.key == key:
            return root
        return self._search(root.left, key) if key <
root.key else self._search(root.right, key)


    def preorder(self, root):
        if root:
            print(root.key, end=" ")
            self.preorder(root.left)
            self.preorder(root.right)



  def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=" ")
            self.inorder(root.right)

    def postorder(self, root):
        if root:
```

```
            self.postorder(root.left)
            self.postorder(root.right)
            print(root.key, end=" ")

# Example Usage:
bst = BST()
for num in [50, 30, 70, 20, 40, 60, 80]:
    bst.insert(num)

print("Inorder Traversal:")
bst.inorder(bst.root)
print("\nSearch 40:", "Found" if bst.search(40) else "Not
Found")
bst.delete(40)
print("Inorder after deleting 40:")
bst.inorder(bst.root)
```

OUTPUT:

```
Inorder Traversal:
20 30 40 50 60 70 80
Search 40: Found
Inorder after deleting 40:
20 30 50 60 70 80
```

Practical - 7

Write a program to implement the insert and search operation in AVL trees.

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = self.right = None
        self.height = 1

def height(n):
    return n.height if n else 0

def balance(n):
    return height(n.left) - height(n.right) if n else 0

def right_rotate(y):
    x = y.left
    y.left = x.right
    x.right = y
    y.height = 1 + max(height(y.left), height(y.right))
    x.height = 1 + max(height(x.left), height(x.right))
    return x

def left_rotate(x):
    y = x.right
    x.right = y.left
    y.left = x
    x.height = 1 + max(height(x.left), height(x.right))
    y.height = 1 + max(height(y.left), height(y.right))
    return y



def insert(root, key):
    if not root:
        return Node(key)
    if key < root.key:
        root.left = insert(root.left, key)
```

```
    else:
        root.right = insert(root.right, key)

    root.height = 1 + max(height(root.left),
height(root.right))
    b = balance(root)

    if b > 1 and key < root.left.key:
        return right_rotate(root)
    if b < -1 and key > root.right.key:
        return left_rotate(root)
    if b > 1 and key > root.left.key:
        root.left = left_rotate(root.left)
        return right_rotate(root)
    if b < -1 and key < root.right.key:
        root.right = right_rotate(root.right)
        return left_rotate(root)

    return root

def search(root, key):
    if not root:
        return False
    if key == root.key:
        return True
    return search(root.left, key) if key < root.key else
search(root.right, key)

# Function to print vertically
def print_vertical(root, level=0, prefix="Root: "):
    if root:
        print_vertical(root.right, level + 1, "R--- ")
        print("     " * level + prefix + str(root.key))
        print_vertical(root.left, level + 1, "L--- ")
```

 # Main Execution

```
root = None
values = [30, 20, 40, 10, 25, 50, 5]  # Example values

for val in values:
```

```
    root = insert(root, val)

print("AVL Tree (vertical layout):")
print_vertical(root)

# Search example
val = 25
print(f"\nSearch {val}: {'Found' if search(root, val) else
'Not Found'}")
```

OUTPUT:

```
AVL Tree (vertical layout):
        R--- 50
    R--- 40
Root: 30
        R--- 25
    L--- 20
        L--- 10
            L--- 5

Search 25: Found
```