20 mrks

1.Write a python program to perform following operations on BST.Insert,Display:

```
class Node:
   def __init__(self, key):
      self.key = key
      self.left = None
      self.right = None
class BST:
   def __init__(self):
      self.root = None
   def insert(self, key):
      if self.root is None:
         self.root = Node(key)
      else:
         self._insert(self.root, key)
   def _insert(self, current, key):
      if key < current.key:
         if current.left is None:
            current.left = Node(key)
         else:
            self._insert(current.left, key)
      elif key > current.key:
         if current.right is None:
            current.right = Node(key)
         else:
            self._insert(current.right, key)
      else:
         print(f"Key {key} already exists in the BST.")

   def display(self):
       print("BST in-order traversal:")
       self._inorder_traversal(self.root)
       print()
   def _inorder_traversal(self, current):
      if current is not None:
         self._inorder_traversal(current.left)
         print(current.key, end=" ")
         self._inorder_traversal(current.right)
if __name__ == "__main__":
   bst = BST()
   while True:
      print("\n1. Insert")
      print("2. Display")
      print("3. Exit")
```

```python
        choice = int(input("Enter your choice: "))
        if choice == 1:
            key = int(input("Enter the key to insert: "))
            bst.insert(key)
        elif choice == 2:
            bst.display()
        elif choice == 3:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")
```

2. Write Python program to merge two sorted linked lists.

```python
class Node:
    """A class representing a single node in a linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    """A class for creating and managing a linked list."""
    def __init__(self):
        self.head = None

    def append(self, data):
        """Append a new node to the end of the linked list."""
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        """Display the elements of the linked list."""
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
def merge_sorted_lists(list1, list2):
    """Merge two sorted linked lists into one sorted linked list."""
    dummy = Node(0)
    tail = dummy
    a = list1.head
    b = list2.head
```

```
    while a and b:
        if a.data < b.data:
            tail.next = a
            a = a.next
        else:
            tail.next = b
            b = b.next
        tail = tail.next
    if a:
        tail.next = a
    if b:
        tail.next = b
    merged_list = LinkedList()
    merged_list.head = dummy.next
    return merged_list
if __name__ == "__main__":
    list1 = LinkedList()
    list2 = LinkedList()
    print("Enter elements of first sorted linked list (comma-separated): ")
    elements1 = list(map(int, input().split(',')))
    for elem in elements1:
        list1.append(elem)
    print("Enter elements of second sorted linked list (comma-separated): ")
    elements2 = list(map(int, input().split(',')))
    for elem in elements2:
        list2.append(elem)
    print("\nFirst sorted linked list:")
    list1.display()
    print("Second sorted linked list:")
    list2.display()
    merged_list = merge_sorted_lists(list1, list2)
    print("\nMerged sorted linked list:")
    merged_list.display()
```

3. Write a python program to perform following operations on BST.
Create
Search
Display (Preorder / Inorder / Postorder)

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

```
def create(self, key):
    if self.root is None:
        self.root = Node(key)
    else:
        self._insert(self.root, key)
def _insert(self, current, key):
    if key < current.key:
        if current.left is None:
            current.left = Node(key)
        else:
            self._insert(current.left, key)
    elif key > current.key:
        if current.right is None:
            current.right = Node(key)
        else:
            self._insert(current.right, key)
    else:
        print(f"Key {key} already exists in the BST.")
def search(self, key):
    """Search for a key in the BST."""
    return self._search(self.root, key)
def _search(self, current, key):
    """Helper method for searching a key recursively."""
    if current is None:
        return False
    if current.key == key:
        return True
    elif key < current.key:
        return self._search(current.left, key)
    else:
        return self._search(current.right, key)
def display(self, order='inorder'):
    """Display the BST using the specified traversal."""
    if order == 'inorder':
        print("Inorder traversal:")
        self._inorder_traversal(self.root)
    elif order == 'preorder':
        print("Preorder traversal:")
        self._preorder_traversal(self.root)
    elif order == 'postorder':
        print("Postorder traversal:")
        self._postorder_traversal(self.root)
    else:
        print("Invalid order specified. Use 'inorder', 'preorder', or 'postorder'.")
    print()
```

```
    def _inorder_traversal(self, current):
        if current:
            self._inorder_traversal(current.left)
            print(current.key, end=" ")
            self._inorder_traversal(current.right)
    def _preorder_traversal(self, current):
        if current:
            print(current.key, end=" ")
            self._preorder_traversal(current.left)
            self._preorder_traversal(current.right)
    def _postorder_traversal(self, current):
        if current:
            self._postorder_traversal(current.left)
            self._postorder_traversal(current.right)
            print(current.key, end=" ")
if __name__ == "__main__":
    bst = BST()
    while True:
        print("\n1. Create (Insert)")
        print("2. Search")
        print("3. Display (Inorder, Preorder, Postorder)")
        print("4. Exit")
        choice = int(input("Enter your choice: "))
        if choice == 1:
            key = int(input("Enter the key to insert: "))
            bst.create(key)
        elif choice == 2:
            key = int(input("Enter the key to search: "))
            found = bst.search(key)
            if found:
                print(f"Key {key} found in the BST.")
            else:
                print(f"Key {key} not found in the BST.")
        elif choice == 3:
            order = input("Enter the traversal type (inorder, preorder, postorder): ").strip().lower()
            bst.display(order)
        elif choice == 4:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")
```

4. Python program for static implementation of Singly Linked List to perform Insert and Display operations.

```
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)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        print(f"Inserted {data} into the linked list.")
    def display(self):
        if self.head is None:
            print("The linked list is empty.")
            return
        current = self.head
        print("Linked List:", end=" ")
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
if __name__ == "__main__":
    sll = SinglyLinkedList()
    while True:
        print("\n1. Insert")
        print("2. Display")
        print("3. Exit")
        choice = int(input("Enter your choice: "))
        if choice == 1:
            data = int(input("Enter the data to insert: "))
            sll.insert(data)
        elif choice == 2:
            sll.display()
        elif choice == 3:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")
```

5. Write a python program to perform following operations on Binary Search Tree

i. Create

ii. Count non-leaf nodes

iii. Traversal (Prorder / Inorder / Postorder)

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def create(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)
    def _insert(self, current, key):
        if key < current.key:
            if current.left is None:
                current.left = Node(key)
            else:
                self._insert(current.left, key)
        elif key > current.key:
            if current.right is None:
                current.right = Node(key)
            else:
                self._insert(current.right, key)
        else:
            print(f"Key {key} already exists in the BST.")
    def count_non_leaf_nodes(self):
        return self._count_non_leaf_nodes(self.root)
    def _count_non_leaf_nodes(self, current):
        if current is None or (current.left is None and current.right is None):
            return 0
        return 1 + self._count_non_leaf_nodes(current.left) +
self._count_non_leaf_nodes(current.right)
    def traversal(self, order='inorder'):
        if order == 'inorder':
            print("Inorder traversal:")
            self._inorder_traversal(self.root)
        elif order == 'preorder':
            print("Preorder traversal:")
            self._preorder_traversal(self.root)
        elif order == 'postorder':
            print("Postorder traversal:")
            self._postorder_traversal(self.root)
        else:
```

```
            print("Invalid traversal type. Use 'inorder', 'preorder', or 'postorder'.")
        print()
    def _inorder_traversal(self, current):
        if current:
            self._inorder_traversal(current.left)
            print(current.key, end=" ")
            self._inorder_traversal(current.right)
    def _preorder_traversal(self, current):
        if current:
            print(current.key, end=" ")
            self._preorder_traversal(current.left)
            self._preorder_traversal(current.right)
    def _postorder_traversal(self, current):
        if current:
            self._postorder_traversal(current.left)
            self._postorder_traversal(current.right)
            print(current.key, end=" ")
if __name__ == "__main__":
    bst = BST()
    while True:
        print("\n1. Create (Insert)")
        print("2. Count Non-Leaf Nodes")
        print("3. Traversal (Inorder, Preorder, Postorder)")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            key = int(input("Enter the key to insert: "))
            bst.create(key)
        elif choice == 2:
            non_leaf_count = bst.count_non_leaf_nodes()
            print(f"Number of non-leaf nodes: {non_leaf_count}")
        elif choice == 3:
            order = input("Enter the traversal type (inorder, preorder, postorder): ").strip().lower()
            bst.traversal(order)
        elif choice == 4:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")
```

. 6. Python program for dynamic implementation of Singly Linked List to perform Insert and Display operations.

```
class Node:
    """A class representing a single node in the linked list."""
    def __init__(self, data):
```

```python
        self.data = data
        self.next = None
class SinglyLinkedList:
    """A dynamically implemented singly linked list."""
    def __init__(self):
        self.head = None
    def insert(self, data):
        """Insert a new node at the end of the linked list."""
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        print(f"Inserted {data} into the linked list.")
    def display(self):
        """Display all the elements in the linked list."""
        if self.head is None:
            print("The linked list is empty.")
            return
        current = self.head
        print("Linked List:", end=" ")
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
if __name__ == "__main__":
    sll = SinglyLinkedList()
    while True:
        print("\n1. Insert")
        print("2. Display")
        print("3. Exit")
        choice = int(input("Enter your choice: "))
        if choice == 1:
            data = int(input("Enter the data to insert: "))
            sll.insert(data)
        elif choice == 2:
            sll.display()
        elif choice == 3:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")
```

7. Write a python program to perform following operations on Binary Search Tree
i. Create
ii. Count leaf nodes
iii. Traversal (Prorder / Inorder / Postorder)

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def create(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)
    def _insert(self, current, key):
        if key < current.key:
            if current.left is None:
                current.left = Node(key)
            else:
                self._insert(current.left, key)
        elif key > current.key:
            if current.right is None:
                current.right = Node(key)
            else:
                self._insert(current.right, key)
        else:
            print(f"Key {key} already exists in the BST.")
    def count_leaf_nodes(self):
        return self._count_leaf_nodes(self.root)
    def _count_leaf_nodes(self, current):
        if current is None:
            return 0
        if current.left is None and current.right is None:
            return 1
        return self._count_leaf_nodes(current.left) + self._count_leaf_nodes(current.right)
    def traversal(self, order='inorder'):
        if order == 'inorder':
            print("Inorder traversal:")
            self._inorder_traversal(self.root)
        elif order == 'preorder':
            print("Preorder traversal:")
            self._preorder_traversal(self.root)
```

```
        elif order == 'postorder':
            print("Postorder traversal:")
            self._postorder_traversal(self.root)
        else:
            print("Invalid traversal type. Use 'inorder', 'preorder', or 'postorder'.")
        print()
    def _inorder_traversal(self, current):
        if current:
            self._inorder_traversal(current.left)
            print(current.key, end=" ")
            self._inorder_traversal(current.right)
    def _preorder_traversal(self, current):
        if current:
            print(current.key, end=" ")
            self._preorder_traversal(current.left)
            self._preorder_traversal(current.right)
    def _postorder_traversal(self, current):
        if current:
            self._postorder_traversal(current.left)
            self._postorder_traversal(current.right)
            print(current.key, end=" ")
if __name__ == "__main__":
    bst = BST()
    while True:
        print("\n1. Create (Insert)")
        print("2. Count Leaf Nodes")
        print("3. Traversal (Inorder, Preorder, Postorder)")
        print("4. Exit")
        choice = int(input("Enter your choice: "))

        if choice == 1:
            key = int(input("Enter the key to insert: "))
            bst.create(key)
        elif choice == 2:
            leaf_count = bst.count_leaf_nodes()
            print(f"Number of leaf nodes: {leaf_count}")
        elif choice == 3:
            order = input("Enter the traversal type (inorder, preorder, postorder): ").strip().lower()
            bst.traversal(order)
        elif choice == 4:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")
```

8. Python program to create a linked list in the sorted order.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SortedLinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if self.head is None or self.head.data >= data:
            new_node.next = self.head
            self.head = new_node
        else:
            current = self.head
            while current.next and current.next.data < data:
                current = current.next
            new_node.next = current.next
            current.next = new_node
        print(f"Inserted {data} into the linked list.")
    def display(self):
        if self.head is None:
            print("The linked list is empty.")
            return
        current = self.head
        print("Sorted Linked List:", end=" ")
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
if __name__ == "__main__":
    sll = SortedLinkedList()
    while True:
        print("\n1. Insert")
        print("2. Display")
        print("3. Exit")
        choice = int(input("Enter your choice: "))
        if choice == 1:
            data = int(input("Enter the data to insert: "))
            sll.insert(data)
        elif choice == 2:
            sll.display()
        elif choice == 3:
            print("Exiting...")
            break
        else:
```

```
            print("Invalid choice. Please try again.")
```

9. Write a python program to perform following operations on BST

i. Create

ii. Delete

iii. Traversal (Prorder / Inorder / Postorder)

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def create(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)
    def _insert(self, current, key):
        if key < current.key:
            if current.left is None:
                current.left = Node(key)
            else:
                self._insert(current.left, key)
        elif key > current.key:
            if current.right is None:
                current.right = Node(key)
            else:
                self._insert(current.right, key)
        else:
            print(f"Key {key} already exists in the BST.")
    def delete(self, key):
        self.root = self._delete(self.root, key)
    def _delete(self, current, key):
        if current is None:
            print(f"Key {key} not found.")
            return current
        if key < current.key:
            current.left = self._delete(current.left, key)
        elif key > current.key:
            current.right = self._delete(current.right, key)
        else:
            if current.left is None:
                return current.right
            elif current.right is None:
```

```
                return current.left
            successor = self._min_value_node(current.right)
            current.key = successor.key
            current.right = self._delete(current.right, successor.key)
        return current
    def _min_value_node(self, current):
        while current.left is not None:
            current = current.left
        return current
    def traversal(self, order='inorder'):
        if order == 'inorder':
            print("Inorder traversal:")
            self._inorder_traversal(self.root)
        elif order == 'preorder':
            print("Preorder traversal:")
            self._preorder_traversal(self.root)
        elif order == 'postorder':
            print("Postorder traversal:")
            self._postorder_traversal(self.root)
        else:
            print("Invalid traversal type. Use 'inorder', 'preorder', or 'postorder'.")
        print()
    def _inorder_traversal(self, current):
        if current:
            self._inorder_traversal(current.left)
            print(current.key, end=" ")
            self._inorder_traversal(current.right)
    def _preorder_traversal(self, current):
        if current:
            print(current.key, end=" ")
            self._preorder_traversal(current.left)
            self._preorder_traversal(current.right)
    def _postorder_traversal(self, current):
        if current:
            self._postorder_traversal(current.left)
            self._postorder_traversal(current.right)
            print(current.key, end=" ")
if __name__ == "__main__":
    bst = BST()
    while True:
        print("\n1. Create (Insert)")
        print("2. Delete")
        print("3. Traversal (Inorder, Preorder, Postorder)")
        print("4. Exit")
        choice = int(input("Enter your choice: "))
```

```
        if choice == 1:
            key = int(input("Enter the key to insert: "))
            bst.create(key)
        elif choice == 2:
            key = int(input("Enter the key to delete: "))
            bst.delete(key)
        elif choice == 3:
            order = input("Enter the traversal type (inorder, preorder, postorder): ").strip().lower()
            bst.traversal(order)
        elif choice == 4:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")
```

10. Write a python program for implementation of Doubly Linked List to perform Insert and Display operations.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
class DoublyLinkedList:
    def __init__(self):
        self.head = None
    def insert(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
        new_node.prev = last
    def display(self):
        if self.head is None:
            print("The list is empty.")
            return
        current = self.head
        while current:
            print(current.data, end=" <-> " if current.next else "")
            current = current.next
        print()
dll = DoublyLinkedList()
```

```
dll.insert(10)
dll.insert(20)
dll.insert(30)
dll.insert(40)
dll.display()
```

11. Write a python program to perform following operations on Binary Search Tree
i. Create
ii. Count total nodes
iii. Traversal (Prorder / Inorder / Postorder)

```
class Node:
   def __init__(self, data):
      self.data = data
      self.left = None
      self.right = None
class BinarySearchTree:
   def __init__(self):
      self.root = None
   def insert(self, data):
      if self.root is None:
         self.root = Node(data)
      else:
         self._insert_recursive(self.root, data)
   def _insert_recursive(self, root, data):
      if data < root.data:
         if root.left is None:
            root.left = Node(data)
         else:
            self._insert_recursive(root.left, data)
      elif data > root.data:
         if root.right is None:
            root.right = Node(data)
         else:
            self._insert_recursive(root.right, data)
   def count_nodes(self):
      return self._count_nodes_recursive(self.root)
   def _count_nodes_recursive(self, root):
      if root is None:
         return 0
      else:
         left_count = self._count_nodes_recursive(root.left)
         right_count = self._count_nodes_recursive(root.right)
         return 1 + left_count + right_count
   def preorder(self):
      result = []
```

```
        self._preorder_recursive(self.root, result)
        return result
    def _preorder_recursive(self, root, result):
        if root:
            result.append(root.data)
            self._preorder_recursive(root.left, result)
            self._preorder_recursive(root.right, result)
    def inorder(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result
    def _inorder_recursive(self, root, result):
        if root:
            self._inorder_recursive(root.left, result)
            result.append(root.data)
            self._inorder_recursive(root.right, result)
    def postorder(self):
        result = []
        self._postorder_recursive(self.root, result)
        return result
    def _postorder_recursive(self, root, result):
        if root:
            self._postorder_recursive(root.left, result)
            self._postorder_recursive(root.right, result)
            result.append(root.data)
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)
print(f"Total nodes: {bst.count_nodes()}")
print(f"Preorder Traversal: {bst.preorder()}")
print(f"Inorder Traversal: {bst.inorder()}")
print(f"Postorder Traversal: {bst.postorder()}")
```

12. Python program to create doubly linked list and search the given node in the Linked list.

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

```
        self.head = None
    def insert(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
        new_node.prev = last
    def search(self, value):
        current = self.head
        while current:
            if current.data == value:
                return True
            current = current.next
        return False
    def display(self):
        if self.head is None:
            print("The list is empty.")
            return
        current = self.head
        while current:
            print(current.data, end=" <-> " if current.next else "")
            current = current.next
        print()
dll = DoublyLinkedList()
dll.insert(10)
dll.insert(20)
dll.insert(30)
dll.insert(40)
dll.display()
search_value = 30
if dll.search(search_value):
    print(f"Node with value {search_value} found in the list.")
else:
    print(f"Node with value {search_value} not found in the list.")
```

14. Python program to create singly linked list and search the given node in the Linked list.

```
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)
        if self.head is None:
            self.head = new_node
            return

        last = self.head
        while last.next:
            last = last.next

        last.next = new_node
    def search(self, value):
        current = self.head
        while current:
            if current.data == value:
                return True
            current = current.next
        return False


    def display(self):
        if self.head is None:
            print("The list is empty.")
            return

        current = self.head
        while current:
            print(current.data, end=" -> " if current.next else "")
            current = current.next
        print()
sll = SinglyLinkedList()
sll.insert(10)
sll.insert(20)
sll.insert(30)
sll.insert(40)
sll.display()
search_value = 30
if sll.search(search_value):
    print(f"Node with value {search_value} found in the list.")
else:
    print(f"Node with value {search_value} not found in the list.")
```

16. Python program to create singly linked list and reverse the Linked list.

```
class Node:
    def __init__(self, data):
```

```python
        self.data = data  # Node's data
        self.next = None  # Pointer to the next node

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


    def insert(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


    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):
        if self.head is None:
            print("The list is empty.")
            return

        current = self.head
        while current:
            print(current.data, end=" -> " if current.next else "")
            current = current.next
        print()
sll = SinglyLinkedList()
sll.insert(10)
sll.insert(20)
```

```
sll.insert(30)
sll.insert(40)
print("Original List:")
sll.display()
sll.reverse()
print("Reversed List:")
sll.display()
```

17. Write a program to search an element using Linear Search.

```
def linear_search(arr, target):
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1
arr = [10, 20, 30, 40, 50, 60, 70]
target = 40
result = linear_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the list.")
```

18. Write a program to calculate indegree of a graph using adjacency matrix.

```
def calculate_indegree(adj_matrix):
    n = len(adj_matrix)
    indegree = [0] * n
    for j in range(n):
        for i in range(n):
            if adj_matrix[i][j] == 1:
                indegree[j] += 1
    return indegree
adj_matrix = [
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1],
    [1, 0, 0, 0]
]
indegree = calculate_indegree(adj_matrix)
for i, degree in enumerate(indegree):
    print(f"Indegree of vertex {i}: {degree}")
```

19. Write a program to search an element using Binary Search.

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

    while left <= right:
        mid = left + (right - left) // 2
```

```
        if arr[mid] == target:
            return mid


        elif arr[mid] > target:
            right = mid - 1

        else:
            left = mid + 1

    return -1
arr = [10, 20, 30, 40, 50, 60, 70, 80, 90]
target = 40
result = binary_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}.")
else:
    print(f"Element {target} not found in the list.")
```

20. Write a Python program to calculate outdegree of a graph using adjacency matrix.

```
def calculate_outdegree(adj_matrix):
    n = len(adj_matrix)
    outdegree = [0] * n
    for i in range(n):
        outdegree[i] = sum(adj_matrix[i])

    return outdegree
adj_matrix = [
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1],
    [1, 0, 0, 0]
]
outdegree = calculate_outdegree(adj_matrix)
for i, degree in enumerate(outdegree):
    print(f"Outdegree of vertex {i}: {degree}")
```