1.In a hospital management system, there's a requirement to maintain a record of patients waiting in the emergency room. Implement a singly linked list to manage this queue efficiently. The system should allow receptionists to add patients to the end of the queue, doctors to remove patients from the front of the queue for examination, and nurses to move patients up in priority if their condition deteriorates. Additionally, the system should provide functionality to display the current queue, search for specific patients, and update patient information as needed. The goal is to streamline the patient management process, ensuring timely and efficient care delivery in the emergency room.

```
class Node:
    def __init__(self, name, count, description):
        self.name = name
        self.count = count
        self.description = description
        self.next = None

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

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

    def add_item(self, name, count, description):
        new_item = Node(name, count, description)
        if self.is_empty():
            self.head = new_item
        else:
            current = self.head
            while current.next:
                current = current.next
```

```
            current.next = new_item

    def remove_item(self):
        if self.is_empty():
            print("No items in the list.")
            return None
        removed_item = self.head
        self.head = self.head.next
        print(f"Removed item: {removed_item.name}")
        return removed_item

    def move_to_priority(self, name):
        if self.is_empty() or self.head.name == name:
            return
        prev = None
        current = self.head
        while current and current.name != name:
            prev = current
            current = current.next
        if current and prev:
            prev.next = current.next
            current.next = self.head
            self.head = current
            print(f"Moved {name} to the front of the list.")

    def search_item(self, name):
        current = self.head
        while current:
            if current.name == name:
                print(f"Found item: {current.name}, Count: {current.count}, Description:
{current.description}")
```

```
            return current
        current = current.next
    print(f"Item '{name}' not found.")
    return None

  def display_list(self):
    current = self.head
    if self.is_empty():
        print("The list is empty.")
        return
    print("Current List:")
    while current:
        print(f"{current.name}, Count: {current.count}, Description: {current.description}")
        current = current.next

def main():
  item_list = LinkedList()
  while True:
    print("\n1. Add Item")
    print("2. Remove Item")
    print("3. Move Item to Priority")
    print("4. Search Item")
    print("5. Display List")
    print("6. Exit")
    choice = input("Enter choice: ")

    if choice == '1':
        name = input("Enter item name: ")
        count = int(input("Enter count: "))
        description = input("Enter description: ")
        item_list.add_item(name, count, description)
```

```
        elif choice == '2':
            item_list.remove_item()
        elif choice == '3':
            name = input("Enter item name: ")
            item_list.move_to_priority(name)
        elif choice == '4':
            name = input("Enter item name: ")
            item_list.search_item(name)
        elif choice == '5':
            item_list.display_list()
        elif choice == '6':
            print("Exiting...")
            break
        else:
            print("Invalid choice.")

if __name__ == "__main__":
    main()
```

2.Implement a doubly circular linked list to efficiently manage customer orders in a retail management system. Each order consists of items with details like name, quantity, and price. The linked list structure should represent each order, with nodes corresponding to individual items. Essential functionalities include adding, removing, and updating items, as well as displaying order contents. The system must dynamically adjust to varying order sizes and offer search and retrieval capabilities for specific items. This implementation aims to streamline the order management process, ensuring quick access and updates within the retail system

```
class Node:
    def __init__(self, name, quantity, price):
        self.name = name
```

```
        self.quantity = quantity
        self.price = price
        self.next = None
        self.prev = None

    def __repr__(self):
        return f"Item: {self.name}, Quantity: {self.quantity}, Price: {self.price:.2f}"


class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

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

    def add_item(self, name, quantity, price):
        new_node = Node(name, quantity, price)
        if self.is_empty():
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
        print(f"Item '{name}' added.")

    def remove_item(self, name):
        if self.is_empty():
            print("The list is empty.")
            return
```

```python
        current = self.head
        while current:
            if current.name == name:
                if current.prev:
                    current.prev.next = current.next
                else:
                    self.head = current.next  # Removing the head node
                if current.next:
                    current.next.prev = current.prev
                else:
                    self.tail = current.prev  # Removing the tail node
                print(f"Item '{name}' removed.")
                return
            current = current.next
        print(f"Item '{name}' not found.")

    def update_item(self, name, quantity=None, price=None):
        if self.is_empty():
            print("The list is empty.")
            return

        current = self.head
        while current:
            if current.name == name:
                if quantity is not None:
                    current.quantity = quantity
                if price is not None:
                    current.price = price
                print(f"Item '{name}' updated.")
                return
```

```python
            current = current.next
        print(f"Item '{name}' not found.")

    def display_order(self):
        if self.is_empty():
            print("The order is empty.")
            return

        current = self.head
        print("\nOrder contents:")
        while current:
            print(current)
            current = current.next

    def search_item(self, name):
        if self.is_empty():
            print("The list is empty.")
            return

        current = self.head
        while current:
            if current.name == name:
                print(f"Item found: {current}")
                return
            current = current.next
        print(f"Item '{name}' not found.")


class Store:
    def __init__(self):
        self.order_list = DoublyLinkedList()
```

```
def menu(self):
    while True:
        print("\n--- Store Menu ---")
        print("1. Add Item")
        print("2. Display Order")
        print("3. Search Item")
        print("4. Update Item")
        print("5. Remove Item")
        print("6. Exit")
        choice = input("Select an option (1-6): ")
        if choice == '1':
            self.add_item()
        elif choice == '2':
            self.order_list.display_order()
        elif choice == '3':
            self.search_item()
        elif choice == '4':
            self.update_item()
        elif choice == '5':
            self.remove_item()
        elif choice == '6':
            print("Exiting the store.")
            break
        else:
            print("Invalid choice. Please try again.")

def add_item(self):
    name = input("Enter item name: ")
    quantity = int(input("Enter item quantity: "))
    price = float(input("Enter item price: "))
```

```
        self.order_list.add_item(name, quantity, price)

    def search_item(self):
        name = input("Enter item name to search: ")
        self.order_list.search_item(name)

    def update_item(self):
        name = input("Enter item name to update: ")
        quantity = input("Enter new quantity (leave blank for no change): ")
        price = input("Enter new price (leave blank for no change): ")
        quantity = int(quantity) if quantity else None
        price = float(price) if price else None
        self.order_list.update_item(name, quantity, price)

    def remove_item(self):
        name = input("Enter item name to remove: ")
        self.order_list.remove_item(name)


if __name__ == "__main__":
    store = Store()
    store.menu()
```

3.Imagine you are developing a system to manage student grades in a school. The grades are stored in a matrix where rows represent students, and columns represent subjects. However, most students do not take all subjects, resulting in a sparse matrix where most elements are zero. Given the sparse matrix representing student grades below, implement a solution using arrays to efficiently manage and manipulate the grades data:

Perform operations such as calculating the average grade for each subject, identifying students with the highest grades, and finding the subject with the highest average grade.

```
import numpy as np

class GradeManager:
    def __init__(self, grades):
        self.grades = np.array(grades)

    def average_per_subject(self):
        averages = []
        for j in range(self.grades.shape[1]):  # Iterate over subjects (columns)
            subject_grades = self.grades[:, j][self.grades[:, j] != 0]  # Non-zero grades for the subject
            average = subject_grades.mean() if subject_grades.size > 0 else 0
            averages.append(float(average))  # Convert to Python float
        return averages

    def highest_grades_per_student(self):
        highest_grades = []
        for i in range(self.grades.shape[0]):  # Iterate over students (rows)
            student_grades = self.grades[i, self.grades[i] != 0]  # Non-zero grades for the student
            highest = student_grades.max() if student_grades.size > 0 else 0
            highest_grades.append((i, int(highest)))  # Convert to Python int
        return highest_grades

    def subject_with_highest_average(self):
        averages = self.average_per_subject()
        highest_avg_index = np.argmax(averages)  # Index of the subject with the highest average
        return int(highest_avg_index), float(averages[highest_avg_index])  # Convert to Python types

if __name__ == "__main__":
    # Sparse matrix representing student grades
```

```
grades_matrix = [
    [0, 85, 0, 90],
    [80, 0, 78, 0],
    [0, 0, 0, 95],
    [70, 88, 0, 0],
    [0, 0, 100, 80]
]

manager = GradeManager(grades_matrix)

# Average grade for each subject
avg_subjects = manager.average_per_subject()
print("Average grades for each subject:", avg_subjects)

# Highest grades per student
highest_grades = manager.highest_grades_per_student()
print("Highest grades for each student:", highest_grades)

# Subject with the highest average grade
highest_avg_subject = manager.subject_with_highest_average()
print("Subject with highest average grade:", highest_avg_subject)
```

4.Develop a stack-based to-do list application for managing tasks. Tasks consist of descriptions and priority levels. Implement functionalities to add, remove, and display tasks based on priority. Optimize memory usage and facilitate efficient task management using the stack data structure

Consider the following initial tasks in the to-do list:

 Task: Complete project proposal

o Priority: High

 Task: Schedule team meeting

o Priority: Medium

 Task: Review draft presentation

o Priority: Low

 Task: Prepare weekly report

o Priority: High

 Task: Respond to client emails

o Priority: Medium.

```
class Task:
    def __init__(self, description, priority):
        self.description = description
        self.priority = priority

    def __repr__(self):
        return f"{self.priority}: {self.description}"

class Priority:
    HIGH = "High"
    MEDIUM = "Medium"
    LOW = "Low"

class TodoList:
    def __init__(self):
        self.tasks = []  # Stack-based storage for tasks

    def add_task(self, task):
        self.tasks.append(task)

    def remove_task(self):
        if self.tasks:
            return self.tasks.pop()  # Stack: LIFO
        return None
```

```
    def display_tasks(self):
        if not self.tasks:
            print("No tasks available.")
            return

        # Sort tasks by priority for display purposes
        sorted_tasks = sorted(self.tasks, key=self.get_priority_level, reverse=True)
        print("Current Tasks:")
        for task in sorted_tasks:
            print(task)

    @staticmethod
    def get_priority_level(task):
        if task.priority == Priority.HIGH:
            return 3
        elif task.priority == Priority.MEDIUM:
            return 2
        elif task.priority == Priority.LOW:
            return 1
        return 0

def initialize_tasks(todo_list):
    # Add initial tasks
    tasks = [
        Task("Complete project proposal", Priority.HIGH),
        Task("Review draft presentation", Priority.LOW),
        Task("Prepare weekly report", Priority.HIGH),
        Task("Respond to client emails", Priority.MEDIUM),
        Task("Schedule team meeting", Priority.MEDIUM),
    ]
```

```
    for task in tasks:
        todo_list.add_task(task)

def display_menu():
    print("\n--- To-Do List Menu ---")
    print("1. Add Task")
    print("2. Remove Task")
    print("3. Display Tasks")
    print("4. Exit")

def main():
    todo_list = TodoList()
    initialize_tasks(todo_list)

    while True:
        display_menu()
        choice = input("Select an option (1-4): ")

        if choice == '1':
            description = input("Enter task description: ")
            priority = input("Enter task priority (High, Medium, Low): ").capitalize()
            if priority in [Priority.HIGH, Priority.MEDIUM, Priority.LOW]:
                new_task = Task(description, priority)
                todo_list.add_task(new_task)
                print("Task added successfully.")
            else:
                print("Invalid priority. Please try again.")

        elif choice == '2':
            removed_task = todo_list.remove_task()
            if removed_task:
```

```
            print(f"Removed Task: {removed_task}")
          else:
            print("No tasks to remove.")

        elif choice == '3':
          todo_list.display_tasks()

        elif choice == '4':
          print("Exiting the application. Goodbye!")
          break

        else:
          print("Invalid option. Please choose again.")

if __name__ == "__main__":
    main()
```

5.Write a program for managing customer service requests in a call center. Each request includes customer details (name, contact information) and service requirements. Utilize a queue data

structure to efficiently handle incoming requests, ensuring they are processed in a first-come-first-served manner. Implement functionalities to add new requests, process them sequentially,

and track the status of ongoing requests.

Consider the following initial customer service requests:

 Request ID: 101

 Customer Name: John Doe

 Service Type: Technical Support

Request ID: 102

 Customer Name: Jane Smith

� Service Type: Billing Inquiry

Request ID: 103

� Customer Name: David Brown

� Service Type: Product Information

```
class CustomerServiceRequest:
    def __init__(self, request_id, customer_name, contact_info, service_type):
        self.request_id = request_id
        self.customer_name = customer_name
        self.contact_info = contact_info
        self.service_type = service_type
        self.next = None  # To connect the requests in the queue

    def __str__(self):
        return f"Request ID: {self.request_id}, Customer: {self.customer_name}, Service Type: 
{self.service_type}, Contact: {self.contact_info}"

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

    # Add a new service request to the queue
    def add_request(self, request_id, customer_name, contact_info, service_type):
        new_request = CustomerServiceRequest(request_id, customer_name, contact_info, 
service_type)
        if self.rear is None:
            self.front = self.rear = new_request
        else:
            self.rear.next = new_request
            self.rear = new_request
```

```
        print(f"Request Added: {new_request}")
        print("--------------------------------------------------------")

    # Process the next service request in the queue
    def process_request(self):
        if self.front is None:
            print("No pending requests. Queue is empty.")
            return

        request_to_process = self.front
        print(f"Processing Request: {request_to_process}")
        self.front = self.front.next  # Remove the processed request from the queue

        if self.front is None:  # If the queue is now empty, set rear to None as well
            self.rear = None

        print("--------------------------------------------------------")
        return request_to_process

    # Display all pending service requests in the queue
    def display_requests(self):
        if self.front is None:
            print("No pending requests. Queue is empty.")
            return

        current = self.front
        print("Pending Requests:")
        while current:
            print(current)
            print("--------------------------------------------------------")
            current = current.next
```

```python
def main():
    call_center = CallCenterQueue()

    while True:
        print("\n--- Call Center Management ---")
        print("1. Add New Request")
        print("2. Process Next Request")
        print("3. Display Pending Requests")
        print("4. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            request_id = int(input("Enter Request ID: "))
            customer_name = input("Enter Customer Name: ")
            contact_info = input("Enter Contact Information: ")
            service_type = input("Enter Service Type: ")
            call_center.add_request(request_id, customer_name, contact_info, service_type)
        elif choice == "2":
            call_center.process_request()
        elif choice == "3":
            call_center.display_requests()
        elif choice == "4":
            print("Exiting the system...")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

6.Implement a circular queue for managing customer orders in a drive-thru lane of a fast-food restaurant. Utilize the circular queue data structure to efficiently handle orders, ensuring fair processing and minimal waiting times. Implement functionalities to add new orders, process orders in a round-robin manner, and track the status of ongoing orders.

Consider the following initial customer orders in the drive-thru lane:

1. Order ID: 101

- Items: Burger, Fries, Drink

- Customer Name: Rahul Sharma

2. Order ID: 102

- Items: Chicken Sandwich, Salad, Drink

- Customer Name: Priya Patel

3. Order ID: 103

- Items: Pizza, Wings, Drink

- Customer Name: Aarav Gupta

```
class Order:
  def __init__(self, order_id, items, cus_name):
    self.order_id = order_id
    self.items = items
    self.cus_name = cus_name
    self.next = None


  def __str__(self):
    return f"Order ID: {self.order_id}, Items: {self.items}, Customer: {self.cus_name}"


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

```python
    def add_item(self, order_id, items, cus_name):
        new_order = Order(order_id, items, cus_name)
        if self.rear is None:
            self.front = self.rear = new_order
            self.rear.next = self.front  # Circular link
        else:
            self.rear.next = new_order
            self.rear = new_order
            self.rear.next = self.front  # Circular link

        print(f"Item added: {new_order}")
        print("----------------------------------------------------------------------------------------------------")

    def delete_item(self):
        if self.front is None:
            print("Order doesn't exist... Queue is empty")
            return None
        elif self.front == self.rear:  # Only one order in the queue
            temp = self.front
            print("Item Processed:", temp.order_id, "->", temp.items, "->", temp.cus_name)
            print("----------------------------------------------")
            self.front = self.rear = None
        else:
            temp = self.front
            print("Item Processed:", temp.order_id, "->", temp.items, "->", temp.cus_name)
            print("----------------------------------------------")
            self.front = self.front.next
            self.rear.next = self.front  # Maintain circular link
        return temp

    def display(self):
```

```
        if self.front is None:
            print("Queue is empty")
            return
        current = self.front
        print("Orders in the queue:")
        while True:
            print(current)
            print("------------------------------------------------------------------------------------------------------------")
            current = current.next
            if current == self.front:  # Circular condition
                break

def main():
    res = Restaurant()
    while True:
        print("\n--- One8 Commune Restaurant ---")
        print("1. Add Order")
        print("2. Process Order")
        print("3. Display Orders")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == "1":
            order_id = int(input("Enter order ID: "))
            items = input("Enter Items: ")
            cus_name = input("Enter Customer name: ")
            res.add_item(order_id, items, cus_name)
        elif choice == "2":
            res.delete_item()
        elif choice == "3":
            res.display()
```

```
        elif choice == "4":
            print("Closed...")
            break
        else:
            print("Invalid Choice")

if __name__ == "__main__":
    main()
```

7.Implement binary search tree and perform following operations: a) Insert (Handle insertion of duplicate entry) b) Delete c) Search d) Display tree (Traversal) e) Display - Depth of tree f) Display - Mirror image g) Create a copy h) Display all parent nodes with their child nodes i) Display leaf nodes j) Display tree level wise.

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

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

    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
```

```
        root.right = self.insert(root.right, key)
    else:
        print(f"Duplicate key '{key}' not inserted.")
    return root

def delete(self, root, key):
    if root is None:
        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 root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        temp = self.find_min(root.right)
        root.key = temp.key
        root.right = self.delete(root.right, temp.key)
    return root

def find_min(self, root):
    while root.left:
        root = root.left
    return root

def search(self, root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
```

```python
        return self.search(root.left, key)
    return self.search(root.right, key)

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

def depth(self, root):
    if root is None:
        return 0
    return 1 + max(self.depth(root.left), self.depth(root.right))

def mirror(self, root):
    if root:
        root.left, root.right = root.right, root.left
        self.mirror(root.left)
        self.mirror(root.right)

def copy_tree(self, root):
    if root is None:
        return None
    new_node = Node(root.key)
    new_node.left = self.copy_tree(root.left)
    new_node.right = self.copy_tree(root.right)
    return new_node

def display_parents_and_children(self, root):
    if root:
        print(f"Parent: {root.key}")
```

```
        if root.left:
            print(f"  Left Child: {root.left.key}")
        if root.right:
            print(f"  Right Child: {root.right.key}")
        self.display_parents_and_children(root.left)
        self.display_parents_and_children(root.right)

    def display_leaf_nodes(self, root):
        if root:
            if root.left is None and root.right is None:
                print(root.key, end=" ")
            self.display_leaf_nodes(root.left)
            self.display_leaf_nodes(root.right)

    def display_level_wise(self, root):
        if root is None:
            return
        queue = [root]
        while queue:
            current = queue.pop(0)
            print(current.key, end=" ")
            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)

# Example Usage
bst = BinarySearchTree()
root = None

# Insertion
```

```
keys = [50, 30, 70, 20, 40, 60, 80]
for key in keys:
    root = bst.insert(root, key)

print("Inorder Traversal:")
bst.inorder(root)
print("\n")

# Depth
print("Depth of the tree:", bst.depth(root))

# Mirror Image
print("\nMirror Image:")
bst.mirror(root)
bst.inorder(root)
print("\n")

# Create a Copy
copy_root = bst.copy_tree(root)
print("\nCopy (Inorder):")
bst.inorder(copy_root)
print("\n")

# Parent and Child Nodes
print("\nParent and Child Nodes:")
bst.display_parents_and_children(root)

# Leaf Nodes
print("\nLeaf Nodes:")
bst.display_leaf_nodes(root)
print("\n")
```

```
# Level-wise Display
print("\nLevel-wise Display:")
bst.display_level_wise(root)
print("\n")
```

8.Write a program to perform binary search for finding a specific word in a sorted array of dictionary entries and return its definition.

```
def binary_search(dictionary, word):
    left, right = 0, len(dictionary) - 1

    while left <= right:
        mid = (left + right) // 2
        mid_word, mid_definition = dictionary[mid]

        if mid_word == word:
            return f"Definition of '{word}': {mid_definition}"
        elif word < mid_word:
            right = mid - 1
        else:
            left = mid + 1

    return f"'{word}' not found in the dictionary."

# Dictionary entries (sorted by word)
dictionary = [
    ("apple", "A fruit that is sweet and crisp."),
    ("banana", "A long yellow tropical fruit."),
    ("cherry", "A small red fruit often used in desserts."),
```

```
    ("date", "A sweet fruit from a date palm tree."),
    ("grape", "A small round fruit used for making wine."),
    ("mango", "A juicy tropical fruit with a large pit."),
    ("orange", "A round citrus fruit with a tangy taste."),
]

# Taking input from the user
word_to_search = input("Enter a word to search in the dictionary: ")

# Calling the function and printing the result
print(binary_search(dictionary, word_to_search))
```

9.Construct an Expression Tree from postfix or prefix expressions. Perform recursive and non-recursive In-order, pre-order and post-order traversals.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None



def construct_postfix_expression_tree(postfix):
    stack = []

    for char in postfix:
        if char.isalnum():  # Operand
            node = TreeNode(char)
            stack.append(node)
        else:  # Operator
```

```
            right = stack.pop()
            left = stack.pop()
            node = TreeNode(char)
            node.left = left
            node.right = right
            stack.append(node)

    return stack[-1]  # Root of the tree


def construct_prefix_expression_tree(prefix):
    stack = []

    for char in reversed(prefix):
        if char.isalnum():  # Operand
            node = TreeNode(char)
            stack.append(node)
        else:  # Operator
            left = stack.pop()
            right = stack.pop()
            node = TreeNode(char)
            node.left = left
            node.right = right
            stack.append(node)

    return stack[-1]  # Root of the tree


# Recursive Traversals
def inorder_recursive(root):
    if root:
```

```
        inorder_recursive(root.left)
        print(root.value, end=" ")
        inorder_recursive(root.right)


def preorder_recursive(root):
    if root:
        print(root.value, end=" ")
        preorder_recursive(root.left)
        preorder_recursive(root.right)


def postorder_recursive(root):
    if root:
        postorder_recursive(root.left)
        postorder_recursive(root.right)
        print(root.value, end=" ")


# Non-Recursive Traversals (Using Stack)
def inorder_non_recursive(root):
    stack = []
    current = root

    while stack or current:
        if current:
            stack.append(current)
            current = current.left
        else:
            current = stack.pop()
            print(current.value, end=" ")
```

```
        current = current.right


def preorder_non_recursive(root):
    if not root:
        return

    stack = [root]

    while stack:
        node = stack.pop()
        print(node.value, end=" ")

        # Right child is pushed first so that left is processed first
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)


def postorder_non_recursive(root):
    if not root:
        return

    stack1 = [root]
    stack2 = []

    while stack1:
        node = stack1.pop()
        stack2.append(node)
```

```
        if node.left:
            stack1.append(node.left)
        if node.right:
            stack1.append(node.right)

    while stack2:
        print(stack2.pop().value, end=" ")


# Example Usage
if __name__ == "__main__":
    # Postfix Expression: AB+C* (Equivalent to (A + B) * C)
    postfix = "AB+C*"
    root_postfix = construct_postfix_expression_tree(postfix)

    print("In-order Traversal (Postfix):")
    inorder_recursive(root_postfix)  # Expected output: A + B * C
    print()

    print("Pre-order Traversal (Postfix):")
    preorder_recursive(root_postfix)  # Expected output: * + A B C
    print()

    print("Post-order Traversal (Postfix):")
    postorder_recursive(root_postfix)  # Expected output: A B + C *
    print()

    print("Non-recursive In-order Traversal (Postfix):")
    inorder_non_recursive(root_postfix)
    print()
```

```python
print("Non-recursive Pre-order Traversal (Postfix):")
preorder_non_recursive(root_postfix)
print()

print("Non-recursive Post-order Traversal (Postfix):")
postorder_non_recursive(root_postfix)
print()

# Prefix Expression: *+ABC (Equivalent to (A + B) * C)
prefix = "*+ABC"
root_prefix = construct_prefix_expression_tree(prefix)

print("\nIn-order Traversal (Prefix):")
inorder_recursive(root_prefix)  # Expected output: A + B * C
print()

print("Pre-order Traversal (Prefix):")
preorder_recursive(root_prefix)  # Expected output: * + A B C
print()

print("Post-order Traversal (Prefix):")
postorder_recursive(root_prefix)  # Expected output: A B + C *
print()

print("Non-recursive In-order Traversal (Prefix):")
inorder_non_recursive(root_prefix)
print()

print("Non-recursive Pre-order Traversal (Prefix):")
preorder_non_recursive(root_prefix)
print()
```

```
    print("Non-recursive Post-order Traversal (Prefix):")
    postorder_non_recursive(root_prefix)
    print()
```

10.A teacher at a local school needs to organize the grades of students to generate their final report cards. The school recently adopted a new grading system, and the grades are stored in a list.

Write a program to sort the grades of students using the insertion sort algorithm.

Read the grades of students from a file named "grades.txt". Each line in the file consists of a single integer representing a student's grade.

Implement the insertion sort algorithm to arrange the grades in ascending order.

Display the sorted grades on the console.

Save the sorted grades to a new file named "sorted_grades.txt".

```
def insertion_sort(grades):
    for i in range(1, len(grades)):
        key = grades[i]
        j = i - 1
        while j >= 0 and key < grades[j]:
            grades[j + 1] = grades[j]
            j -= 1
        grades[j + 1] = key

def read_grades_from_file(filename):
    with open(filename, 'r') as file:
        grades = [int(line.strip()) for line in file]
    return grades

def write_grades_to_file(grades, filename):
    with open(filename, 'w') as file:
```

```
        for grade in grades:
            file.write(f"{grade}\n")

def main():
    grades = read_grades_from_file("grades.txt")
    insertion_sort(grades)
    print("Sorted Grades:")
    for grade in grades:
        print(grade)
    write_grades_to_file(grades, "sorted_grades.txt")

if __name__ == "__main__":
    main()
```

11.Implement a basic student information system that utilizes hashing concepts to efficiently store and retrieve student records. The program should allow users to:

- Add new student records.
- Retrieve student information by their unique student ID.
- Implement basic operations for managing student records.

```
class StudentHashTable:
    def __init__(self):
        self.table = {}

    def add_student(self, student_id, student_name, student_age, student_grade):
        if student_id in self.table:
            print(f"Error: Student ID {student_id} already exists.")
        else:
            self.table[student_id] = {
                "name": student_name,
```

```
            "age": student_age,
            "grade": student_grade
        }
        print(f"Student {student_name} added successfully.")

    def retrieve_student(self, student_id):
        if student_id in self.table:
            student = self.table[student_id]
            print(f"Student ID: {student_id}")
            print(f"Name: {student['name']}")
            print(f"Age: {student['age']}")
            print(f"Grade: {student['grade']}")
        else:
            print(f"Error: Student ID {student_id} not found.")

    def remove_student(self, student_id):
        if student_id in self.table:
            del self.table[student_id]
            print(f"Student ID {student_id} removed successfully.")
        else:
            print(f"Error: Student ID {student_id} not found.")

    def display_all_students(self):
        if not self.table:
            print("No student records found.")
        else:
            print("All Student Records:")
            for student_id, details in self.table.items():
                print(f"Student ID: {student_id}, Name: {details['name']}, Age: {details['age']}, Grade: 
{details['grade']}")
```

```
def main():
    hash_table = StudentHashTable()

    while True:
        print("\nStudent Information System")
        print("1. Add New Student")
        print("2. Retrieve Student Information")
        print("3. Remove Student")
        print("4. Display All Students")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            student_id = input("Enter Student ID: ")
            student_name = input("Enter Student Name: ")
            student_age = input("Enter Student Age: ")
            student_grade = input("Enter Student Grade: ")
            hash_table.add_student(student_id, student_name, student_age, student_grade)

        elif choice == "2":
            student_id = input("Enter Student ID to retrieve: ")
            hash_table.retrieve_student(student_id)

        elif choice == "3":
            student_id = input("Enter Student ID to remove: ")
            hash_table.remove_student(student_id)

        elif choice == "4":
            hash_table.display_all_students()
```

```
        elif choice == "5":
            print("Exiting the Student Information System. Goodbye!")
            break

        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

12.Pimpri Chinchwad Municipal Corporation seeks an efficient solution for laying out a water pipeline network in a newly developed region. The objective is to ensure that every house within the area is connected to the pipeline network while minimizing the total cost of laying out the pipelines.

```
def prims_algorithm(graph, start_vertex=0):
    num_vertices = len(graph)
    mst_set = set()
    mst_edges = []
    min_edge = [float('inf')] * num_vertices
    min_edge[start_vertex] = 0
    parent = [-1] * num_vertices

    while len(mst_set) < num_vertices:
        min_weight = float('inf')
        u = -1

        for vertex in range(num_vertices):
            if vertex not in mst_set and min_edge[vertex] < min_weight:
                min_weight = min_edge[vertex]
```

```python
            u = vertex

        mst_set.add(u)

        if parent[u] != -1:
            mst_edges.append((parent[u], u, min_weight))

        for v, weight in graph[u]:
            if v not in mst_set and weight < min_edge[v]:
                min_edge[v] = weight
                parent[v] = u

    return mst_edges

def get_graph_input():
    num_vertices = int(input("Enter the number of vertices: "))
    graph = {i: [] for i in range(num_vertices)}
    num_edges = int(input("Enter the number of edges: "))
    print("Enter each edge in the format: vertex1 vertex2 weight")
    for _ in range(num_edges):
        vertex1, vertex2, weight = map(int, input().split())
        graph[vertex1].append((vertex2, weight))
        graph[vertex2].append((vertex1, weight))
    return graph

if __name__ == "__main__":
    graph = get_graph_input()
    mst = prims_algorithm(graph, start_vertex=0)
    print("\nMinimum Spanning Tree (MST) edges:")
    for u, v, weight in mst:
        print(f"Edge ({u}, {v}) with weight {weight}")
```