```
class Node:
    """Class to represent a single node in the Binary Search Tree."""
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.count = 1  # For handling duplicates


class BinarySearchTree:
    """Binary Search Tree implementation with various operations."""
    def __init__(self):
        self.root = None

    # Insert node (handles duplicates)
    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key == root.key:
            root.count += 1  # Handle duplicates
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        return root

    def insert_keyclass Node:
    """Class to represent a single node in the Binary Search Tree."""
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.count = 1  # For handling duplicates


class BinarySearchTree:
    """Binary Search Tree implementation with various operations."""
    def __init__(self):
        self.root = None

    # Insert node (handles duplicates)
    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key == root.key:
            root.count += 1  # Handle duplicates
        elif key < root.key:
            root.left = self.insert(root.left, key)
```

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

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

    # Search for a key in the BST
    def search(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self.search(root.left, key)
        return self.search(root.right, key)

    # Delete a node
    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.count > 1:  # Handle duplicate nodes
                root.count -= 1
                return root

            # Node with one child or no child
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left

            # Node with two children
            successor = self.get_min(root.right)
            root.key = successor.key
            root.count = successor.count
            root.right = self.delete(root.right, successor.key)
        return root

    def delete_key(self, key):
        self.root = self.delete(self.root, key)

    # Get the minimum value node
    def get_min(self, root):
        current = root
```

```
        while current.left is not None:
            current = current.left
        return current

    # Depth of the tree
    def depth(self, root):
        if root is None:
            return 0
        return 1 + max(self.depth(root.left), self.depth(root.right))

    # Traversals
    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(f"{root.key}({root.count})", end=" ")
            self.inorder(root.right)

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

    def postorder(self, root):
        if root:
            self.postorder(root.left)
            self.postorder(root.right)
            print(f"{root.key}({root.count})", end=" ")

    # Display tree traversals
    def display_traversals(self):
        print("\nInorder Traversal:")
        self.inorder(self.root)
        print("\nPreorder Traversal:")
        self.preorder(self.root)
        print("\nPostorder Traversal:")
        self.postorder(self.root)
        print()

    # Display leaf nodes
    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)

    # Display parent-child relationships
```

```python
    def display_parent_child(self, root):
        if root:
            if root.left:
                print(f"Parent: {root.key} -> Left Child: {root.left.key}")
            if root.right:
                print(f"Parent: {root.key} -> Right Child: {root.right.key}")
            self.display_parent_child(root.left)
            self.display_parent_child(root.right)

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

    # Create a copy of the tree
    def copy_tree(self, root):
        if root is None:
            return None
        new_node = Node(root.key)
        new_node.count = root.count
        new_node.left = self.copy_tree(root.left)
        new_node.right = self.copy_tree(root.right)
        return new_node

    # Level-wise traversal
    def level_order(self, root):
        if root is None:
            return
        queue = [root]
        while queue:
            current = queue.pop(0)
            print(f"{current.key}({current.count})", end=" ")
            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)


# Menu-driven program
def menu():
    bst = BinarySearchTree()

    while True:
        print("\n=== Binary Search Tree Operations ===")
        print("1. Insert Key")
        print("2. Search Key")
```

```
print("3. Delete Key")
print("4. Display Tree Traversals")
print("5. Show Tree Depth")
print("6. Display Mirror Image")
print("7. Create and Display Copy of Tree")
print("8. Show Parent-Child Nodes")
print("9. Display Leaf Nodes")
print("10. Show Level-Wise Traversal")
print("11. Exit")

choice = input("Enter your choice: ")

if choice == "1":
    key = int(input("Enter key to insert: "))
    bst.insert_key(key)
elif choice == "2":
    key = int(input("Enter key to search: "))
    result = bst.search(bst.root, key)
    if result:
        print(f"Key {key} found with count {result.count}.")
    else:
        print(f"Key {key} not found.")
elif choice == "3":
    key = int(input("Enter key to delete: "))
    bst.delete_key(key)
elif choice == "4":
    bst.display_traversals()
elif choice == "5":
    print(f"Tree Depth: {bst.depth(bst.root)}")
elif choice == "6":
    bst.mirror(bst.root)
    print("Mirror Image of Tree (Inorder Traversal):")
    bst.inorder(bst.root)
    bst.mirror(bst.root)  # Revert to original
elif choice == "7":
    copied_root = bst.copy_tree(bst.root)
    print("Copy of Tree (Inorder Traversal):")
    bst.inorder(copied_root)
elif choice == "8":
    print("Parent-Child Relationships:")
    bst.display_parent_child(bst.root)
elif choice == "9":
    print("Leaf Nodes:")
    bst.display_leaf_nodes(bst.root)
elif choice == "10":
    print("Level-Wise Traversal:")
    bst.level_order(bst.root)
elif choice == "11":
```

```
            print("Exiting the program.")
            break
        else:
            print("Invalid choice! Please try again.")


# Run the program
if __name__ == "__main__":
    menu()
(self, key):
        self.root = self.insert(self.root, key)

    # Search for a key in the BST
    def search(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self.search(root.left, key)
        return self.search(root.right, key)

    # Delete a node
    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.count > 1:  # Handle duplicate nodes
                root.count -= 1
                return root

            # Node with one child or no child
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left

            # Node with two children
            successor = self.get_min(root.right)
            root.key = successor.key
            root.count = successor.count
            root.right = self.delete(root.right, successor.key)
        return root

    def delete_key(self, key):
```

```
    self.root = self.delete(self.root, key)

# Get the minimum value node
def get_min(self, root):
    current = root
    while current.left is not None:
        current = current.left
    return current

# Depth of the tree
def depth(self, root):
    if root is None:
        return 0
    return 1 + max(self.depth(root.left), self.depth(root.right))

# Traversals
def inorder(self, root):
    if root:
        self.inorder(root.left)
        print(f"{root.key}({root.count})", end=" ")
        self.inorder(root.right)

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

def postorder(self, root):
    if root:
        self.postorder(root.left)
        self.postorder(root.right)
        print(f"{root.key}({root.count})", end=" ")

# Display tree traversals
def display_traversals(self):
    print("\nInorder Traversal:")
    self.inorder(self.root)
    print("\nPreorder Traversal:")
    self.preorder(self.root)
    print("\nPostorder Traversal:")
    self.postorder(self.root)
    print()

# Display leaf nodes
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)

    # Display parent-child relationships
    def display_parent_child(self, root):
        if root:
            if root.left:
                print(f"Parent: {root.key} -> Left Child: {root.left.key}")
            if root.right:
                print(f"Parent: {root.key} -> Right Child: {root.right.key}")
            self.display_parent_child(root.left)
            self.display_parent_child(root.right)

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

    # Create a copy of the tree
    def copy_tree(self, root):
        if root is None:
            return None
        new_node = Node(root.key)
        new_node.count = root.count
        new_node.left = self.copy_tree(root.left)
        new_node.right = self.copy_tree(root.right)
        return new_node

    # Level-wise traversal
    def level_order(self, root):
        if root is None:
            return
        queue = [root]
        while queue:
            current = queue.pop(0)
            print(f"{current.key}({current.count})", end=" ")
            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)


# Menu-driven program
def menu():
    bst = BinarySearchTree()
```

```python
while True:
    print("\n=== Binary Search Tree Operations ===")
    print("1. Insert Key")
    print("2. Search Key")
    print("3. Delete Key")
    print("4. Display Tree Traversals")
    print("5. Show Tree Depth")
    print("6. Display Mirror Image")
    print("7. Create and Display Copy of Tree")
    print("8. Show Parent-Child Nodes")
    print("9. Display Leaf Nodes")
    print("10. Show Level-Wise Traversal")
    print("11. Exit")

    choice = input("Enter your choice: ")

    if choice == "1":
        key = int(input("Enter key to insert: "))
        bst.insert_key(key)
    elif choice == "2":
        key = int(input("Enter key to search: "))
        result = bst.search(bst.root, key)
        if result:
            print(f"Key {key} found with count {result.count}.")
        else:
            print(f"Key {key} not found.")
    elif choice == "3":
        key = int(input("Enter key to delete: "))
        bst.delete_key(key)
    elif choice == "4":
        bst.display_traversals()
    elif choice == "5":
        print(f"Tree Depth: {bst.depth(bst.root)}")
    elif choice == "6":
        bst.mirror(bst.root)
        print("Mirror Image of Tree (Inorder Traversal):")
        bst.inorder(bst.root)
        bst.mirror(bst.root)  # Revert to original
    elif choice == "7":
        copied_root = bst.copy_tree(bst.root)
        print("Copy of Tree (Inorder Traversal):")
        bst.inorder(copied_root)
    elif choice == "8":
        print("Parent-Child Relationships:")
        bst.display_parent_child(bst.root)
    elif choice == "9":
        print("Leaf Nodes:")
```

```
            bst.display_leaf_nodes(bst.root)
        elif choice == "10":
            print("Level-Wise Traversal:")
            bst.level_order(bst.root)
        elif choice == "11":
            print("Exiting the program.")
            break
        else:
            print("Invalid choice! Please try again.")


# Run the program
if __name__ == "__main__":
    menu()
```

8.Post pre expression tree

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


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

    # Construct from postfix expression
    def construct_from_postfix(self, postfix):
        stack = []
        for char in postfix:
            if char.isalnum():  # If operand, create a node and push to stack
                stack.append(TreeNode(char))
            else:  # If operator, pop two nodes, make them children, and push new node
to stack
                right = stack.pop()
                left = stack.pop()
                node = TreeNode(char)
                node.left = left
                node.right = right
                stack.append(node)
        self.root = stack.pop()  # Final tree root

    # Construct from prefix expression
    def construct_from_prefix(self, prefix):
        stack = []
        for char in reversed(prefix):
            if char.isalnum():  # If operand
                stack.append(TreeNode(char))
            else:  # If operator
                left = stack.pop()
                right = stack.pop()
                node = TreeNode(char)
                node.left = left
                node.right = right
                stack.append(node)
        self.root = stack.pop()  # Final tree root

    # Recursive Traversals
```

```
def inorder_recursive(self, node):
    if node:
        self.inorder_recursive(node.left)
        print(node.value, end=" ")
        self.inorder_recursive(node.right)

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

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

# Non-Recursive In-order Traversal
def inorder_non_recursive(self):
    stack, current = [], self.root
    while stack or current:
        if current:
            stack.append(current)
            current = current.left
        else:
            current = stack.pop()
            print(current.value, end=" ")
            current = current.right

# Non-Recursive Pre-order Traversal
def preorder_non_recursive(self):
    stack = [self.root]
    while stack:
        node = stack.pop()
        print(node.value, end=" ")
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

# Non-Recursive Post-order Traversal
def postorder_non_recursive(self):
    stack1, stack2 = [self.root], []
```

```python
        while stack1:
            node = stack1.pop()
            stack2.append(node)
            if node.left:
                stack1.append(node.left)
            if node.right:
                stack1.append(node.right)
        while stack2:
            node = stack2.pop()
            print(node.value, end=" ")


if __name__ == "__main__":
    expr_tree = ExpressionTree()

    # Example postfix expression
    postfix_expr = "ab+c*de/-"  # Represents ((a + b) * c) - (d / e)
    expr_tree.construct_from_postfix(postfix_expr)

    print("Recursive In-order Traversal:")
    expr_tree.inorder_recursive(expr_tree.root)
    print("\nNon-Recursive In-order Traversal:")
    expr_tree.inorder_non_recursive()

    print("\n\nRecursive Pre-order Traversal:")
    expr_tree.preorder_recursive(expr_tree.root)
    print("\nNon-Recursive Pre-order Traversal:")
    expr_tree.preorder_non_recursive()

    print("\n\nRecursive Post-order Traversal:")
    expr_tree.postorder_recursive(expr_tree.root)
    print("\nNon-Recursive Post-order Traversal:")
    expr_tree.postorder_non_recursive()
```

9.hashing-student

```
class StudentInformationSystem:
    def __init__(self):
        # Hash table to store student records
        self.records = {}

    def add_record(self, student_id, name, age, course):
        """Add a new student record."""
        if student_id in self.records:
            print(f"Student ID {student_id} already exists. Record not added.")
        else:
            self.records[student_id] = {"Name": name, "Age": age, "Course": course}
            print(f"Record added for Student ID {student_id}.")

    def retrieve_record(self, student_id):
        """Retrieve a student record by ID."""
        if student_id in self.records:
            print(f"Record for Student ID {student_id}: {self.records[student_id]}")
        else:
            print(f"No record found for Student ID {student_id}.")

    def delete_record(self, student_id):
        """Delete a student record by ID."""
        if student_id in self.records:
            del self.records[student_id]
            print(f"Record for Student ID {student_id} deleted.")
        else:
            print(f"No record found for Student ID {student_id}.")

    def display_all_records(self):
        """Display all student records."""
        if not self.records:
            print("No records found.")
        else:
            print("Student Records:")
            for student_id, details in self.records.items():
                print(f"ID: {student_id}, Details: {details}")

# Main Functionality
if __name__ == "__main__":
    sis = StudentInformationSystem()

    while True:
```

```
print("\n--- Student Information System ---")
print("1. Add Record")
print("2. Retrieve Record")
print("3. Delete Record")
print("4. Display All Records")
print("5. Exit")

choice = input("Enter your choice: ")

if choice == "1":
    student_id = input("Enter Student ID: ")
    name = input("Enter Student Name: ")
    age = int(input("Enter Student Age: "))
    course = input("Enter Student Course: ")
    sis.add_record(student_id, name, age, course)

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

elif choice == "3":
    student_id = input("Enter Student ID to delete: ")
    sis.delete_record(student_id)

elif choice == "4":
    sis.display_all_records()

elif choice == "5":
    print("Exiting the system. Goodbye!")
    break

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

10.dict:binary search

```
# Binary search function
def binary_search(dictionary, word_to_find):
    low, high = 0, len(dictionary) - 1

    while low <= high:
        mid = (low + high) // 2
        current_word = dictionary[mid][0]  # Access the word at the middle index

        if current_word == word_to_find:
            return dictionary[mid][1]  # Return the definition if found
        elif current_word < word_to_find:
            low = mid + 1  # Move to the right half
        else:
            high = mid - 1  # Move to the left half

    return None  # Word not found


# Main program
if __name__ == "__main__":
    # Sorted array of dictionary entries (word, definition)
    dictionary = [
        ("apple", "A sweet, edible fruit produced by an apple tree."),
        ("banana", "An elongated, edible fruit produced by various plants."),
        ("cat", "A small domesticated carnivorous mammal."),
        ("dog", "A domesticated carnivorous mammal that typically has a long snout."),
        ("elephant", "A large herbivorous mammal with a trunk."),
    ]

    print("Dictionary Search")
    word = input("Enter the word to search for: ").strip().lower()

    # Perform binary search
    definition = binary_search(dictionary, word)

    if definition:
        print(f"Definition of '{word}': {definition}")
    else:
        print(f"'{word}' not found in the dictionary.")
```

11.insertion sort

```
def insertion_sort(grades):
    """Sort the grades list using insertion sort."""
    for i in range(1, len(grades)):
        key = grades[i]
        j = i - 1
        # Move elements of grades[0..i-1] that are greater than key
        # to one position ahead of their current position
        while j >= 0 and grades[j] > key:
            grades[j + 1] = grades[j]
            j -= 1
        grades[j + 1] = key
    return grades


def read_grades(file_name):
    """Read grades from the file and return them as a list of integers."""
    with open(file_name, "r") as file:
        grades = file.readlines()
    return [int(grade.strip()) for grade in grades]


def save_grades(file_name, grades):
    """Save sorted grades to the file."""
    with open(file_name, "w") as file:
        for grade in grades:
            file.write(f"{grade}\n")


def main():
    input_file = "grades.txt"
    output_file = "sorted_grades.txt"

    # Step 1: Read grades from "grades.txt"
    try:
        grades = read_grades(input_file)
    except FileNotFoundError:
        print(f"Error: The file '{input_file}' was not found.")
        return
    except ValueError:
        print("Error: The file contains non-integer values.")
        return

    # Step 2: Sort grades using insertion sort
```

```
    sorted_grades = insertion_sort(grades)

    # Step 3: Display sorted grades
    print("Sorted Grades:")
    for grade in sorted_grades:
        print(grade)

    # Step 4: Save sorted grades to "sorted_grades.txt"
    save_grades(output_file, sorted_grades)
    print(f"Sorted grades have been saved to '{output_file}'.")


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

12.corp:mst

```
def prims_algorithm(graph, start_vertex=0):
    # Number of vertices in the graph
    num_vertices = len(graph)

    # Set to store vertices included in the MST
    mst_set = set()

    # List to store edges of the MST
    mst_edges = []

    # List to track minimum edge weights for each vertex
    min_edge = [float('inf')] * num_vertices
    min_edge[start_vertex] = 0

    # List to store the parent of each vertex in the MST
    parent = [-1] * num_vertices

    while len(mst_set) < num_vertices:
        # Find the vertex with the smallest edge weight not in the MST
        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]
                u = vertex

        # Add this vertex to the MST set
        mst_set.add(u)
```

```python
        # If u is not the start vertex, add edge (parent[u], u) to MST
        if parent[u] != -1:
            mst_edges.append((parent[u], u, min_weight))

        # Update the minimum edge weights for neighbors of u
        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():
    # Get the number of vertices from the user
    num_vertices = int(input("Enter the number of vertices: "))

    # Initialize an empty adjacency list for the graph
    graph = {i: [] for i in range(num_vertices)}

    # Get the number of edges
    num_edges = int(input("Enter the number of edges: "))

    # Get each edge input from the user
    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__":
    # Get graph input from the user
    graph = get_graph_input()

    # Run Prim's algorithm to find the MST
    mst = prims_algorithm(graph, start_vertex=0)

    # Print the edges in the MST
    print("\nMinimum Spanning Tree (MST) edges:")
    for u, v, weight in mst:
```

```
print(f"Edge ({v}, {u}) with weight {weight}")
```

2.sparse

```
from collections import defaultdict

# Sparse matrix representation of student grades
sparse = {
    (1, 1): 4,  # (student, subject): grade
    (1, 2): 5,
    (2, 1): 3,
    (2, 2): 4,
    (2, 3): 5,
    (3, 1): 2,
    (3, 3): 4,
}

no_std = 3  # Number of students
no_sub = 3  # Number of subjects

def avg_grade_per_sub_and_max_grade(sparse, no_std, no_sub):
    # Dictionaries to store totals, counts, and maximum grades
    sub_totals = defaultdict(int)
    sub_count = defaultdict(int)
    sub_max = defaultdict(lambda: float('-inf'))  # Initialize max to negative infinity

    # Step 1: Gather totals, counts, and max grades for each subject
    for (student, subject), grade in sparse.items():
        sub_totals[subject] += grade  # Sum grades for each subject
        sub_count[subject] += 1       # Count grades per subject
        sub_max[subject] = max(sub_max[subject], grade)  # Find maximum grade

    # Step 2: Calculate average grades for each subject
    sub_averages = {}
    for subject in range(1, no_sub + 1):  # Iterate through all subjects (1-based
indexing)
        if sub_count[subject] > 0:  # Avoid division by zero
            sub_averages[subject] = sub_totals[subject] / sub_count[subject]
        else:
            sub_averages[subject] = 0  # If no grades, average is 0

    # Step 3: Find the subject with the highest average grade
    highest_avg_subject = max(sub_averages, key=sub_averages.get)  # Subject
with max average
    highest_avg_value = sub_averages[highest_avg_subject]         # Highest average
value
```

```
    # Step 4: Print the results
    print("\nAverage grade per subject:")
    for subject, avg in sub_averages.items():
        print(f"Subject {subject}: {avg:.2f}")

    print("\nHighest grade per subject:")
    for subject, max_grade in sub_max.items():
        print(f"Subject {subject}: {max_grade}")

    print(f"\nSubject with the highest average grade: Subject {highest_avg_subject}")
    print(f"Highest average value: {highest_avg_value:.2f}")

# Run the function
avg_grade_per_sub_and_max_grade(sparse, no_std, no_sub)
```

1.patient

```
class PatientNode:
    def __init__(self, patient_id, name, condition):
        self.patient_id = patient_id
        self.name = name
        self.condition = condition
        self.next = None

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

    def add_patient(self, patient_id, name, condition):
        new_patient = PatientNode(patient_id, name, condition)
        if not self.head:  # If the list is empty
            self.head = new_patient
        else:
            current = self.head
            while current.next:  # Traverse to the end of the list
                current = current.next
            current.next = new_patient
        print(f"Patient {name} added successfully.")

    def remove_patient_by_id(self, patient_id):
        if not self.head:  # If the list is empty
            print("No patients in the emergency room.")
            return

        # If the head node matches the ID
```

```
        if self.head.patient_id == patient_id:
            removed_patient = self.head
            self.head = self.head.next  # Move head pointer
            print(f"Patient {removed_patient.name} with ID {removed_patient.patient_id}
has been removed.")
            return

        # Search for the patient in the list
        current = self.head
        while current.next and current.next.patient_id != patient_id:
            current = current.next

        if current.next:  # Patient found
            removed_patient = current.next
            current.next = current.next.next  # Bypass the removed patient
            print(f"Patient {removed_patient.name} with ID {removed_patient.patient_id}
has been removed.")
        else:
            print(f"No patient found with ID {patient_id}.")

    def search_patient(self, patient_id):
        current = self.head
        while current:
            if current.patient_id == patient_id:
                print(f"Patient Found: ID: {current.patient_id}, Name: {current.name},
Condition: {current.condition}")
                return
            current = current.next
        print("Patient not found.")

    def display_patients(self):
        if not self.head:
            print("No patients in the emergency room.")
            return
        current = self.head
        print("Current Patients in Emergency Room:")
        while current:
            print(f"ID: {current.patient_id}, Name: {current.name}, Condition:
{current.condition}")
            current = current.next

# Menu-Driven Program
def menu():
    er = EmergencyRoom()
```

```
    while True:
        print("\n=== Emergency Room Management ===")
        print("1. Add Patient")
        print("2. Remove Patient by ID")
        print("3. Search Patient")
        print("4. Display All Patients")
        print("5. Exit")
        choice = input("Enter your choice: ")

        if choice == "1":
            patient_id = int(input("Enter Patient ID: "))
            name = input("Enter Patient Name: ")
            condition = input("Enter Patient Condition: ")
            er.add_patient(patient_id, name, condition)
        elif choice == "2":
            patient_id = int(input("Enter Patient ID to Remove: "))
            er.remove_patient_by_id(patient_id)
        elif choice == "3":
            patient_id = int(input("Enter Patient ID to Search: "))
            er.search_patient(patient_id)
        elif choice == "4":
            er.display_patients()
        elif choice == "5":
            print("Exiting the program. Stay safe!")
            break
        else:
            print("Invalid choice! Please try again.")

# Run the menu
if __name__ == "__main__":
    menu()
```

3.inventory:double

```
class ProductNode:
    def __init__(self, product_id, name, quantity, price):
        self.product_id = product_id
        self.name = name
        self.quantity = quantity
        self.price = price
        self.prev = None  # Pointer to the previous product
        self.next = None  # Pointer to the next product


class InventoryManagementSystem:
```

```
def __init__(self):
    self.head = None  # Head of the doubly linked list
    self.tail = None  # Tail of the doubly linked list

def add_product(self, product_id, name, quantity, price):
    # Create a new product node
    new_product = ProductNode(product_id, name, quantity, price)

    if self.head is None:  # If the list is empty
        self.head = new_product
        self.tail = new_product
    else:  # Add the product to the end of the list
        self.tail.next = new_product
        new_product.prev = self.tail
        self.tail = new_product
    print(f"Product {name} added to inventory.")

def update_quantity(self, product_id, new_quantity):
    current = self.head

    while current:
        if current.product_id == product_id:  # Product found
            current.quantity = new_quantity
            print(f"Updated quantity of {current.name} to {new_quantity}.")
            return
        current = current.next

    print("Product not found in inventory.")

def calculate_total_inventory_value(self):
    current = self.head
    total_value = 0

    while current:
        total_value += current.quantity * current.price
        current = current.next

    return total_value

def display_inventory(self):
    if self.head is None:
        print("Inventory is empty.")
        return
```

```python
        current = self.head
        print("\n=== Inventory ===")
        while current:
            print(
                f"Product ID: {current.product_id}, Name: {current.name}, "
                f"Quantity: {current.quantity}, Price: {current.price:.2f}, "
                f"Value: {current.quantity * current.price:.2f}"
            )
            current = current.next


# Menu-Driven Program
def menu():
    ims = InventoryManagementSystem()

    while True:
        print("\n=== Inventory Management System ===")
        print("1. Add Product")
        print("2. Update Product Quantity")
        print("3. Calculate Total Inventory Value")
        print("4. Display All Products")
        print("5. Exit")
        choice = input("Enter your choice: ")

        if choice == "1":
            product_id = input("Enter Product ID: ")
            name = input("Enter Product Name: ")
            quantity = int(input("Enter Quantity: "))
            price = float(input("Enter Price: "))
            ims.add_product(product_id, name, quantity, price)
        elif choice == "2":
            product_id = input("Enter Product ID: ")
            new_quantity = int(input("Enter New Quantity: "))
            ims.update_quantity(product_id, new_quantity)
        elif choice == "3":
            total_value = ims.calculate_total_inventory_value()
            print(f"Total Inventory Value: {total_value:.2f}")
        elif choice == "4":
            ims.display_inventory()
        elif choice == "5":
            print("Exiting the program.")
            break
        else:
            print("Invalid choice! Please try again.")
```

```
# Run the program
if __name__ == "__main__":
    menu()

4stack:todo
.class TaskNode:
    """
    Represents a task in the to-do list stack.
    """
    def __init__(self, description, priority):
        self.description = description  # Task description
        self.priority = priority        # Task priority
        self.next = None                # Pointer to the next task


class ToDoStack:
    """
    Stack to manage tasks using a linked list.
    """
    def __init__(self):
        self.top = None  # Pointer to the top task in the stack

    def is_empty(self):
        """
        Check if the stack is empty.
        """
        return self.top is None

    def add_task(self, description, priority):
        """
        Add a new task to the stack.
        """
        new_task = TaskNode(description, priority)
        new_task.next = self.top  # Link the new task to the current top
        self.top = new_task       # Update the top pointer
        print(f"Task '{description}' with priority {priority} added.")

    def remove_task(self):
        """
        Remove the top task from the stack.
        """
        if self.is_empty():
```

```
            print("No tasks to remove. To-do list is empty.")
            return None
        removed_task = self.top
        self.top = self.top.next  # Update the top pointer to the next task
        print(f"Removed task: '{removed_task.description}' with priority
{removed_task.priority}.")
        return removed_task

    def display_tasks(self):
        """
        Display all tasks in the stack by priority.
        """
        if self.is_empty():
            print("To-do list is empty.")
            return
        print("\n=== To-Do List ===")
        current = self.top
        while current:
            print(f"[Priority {current.priority}] {current.description}")
            current = current.next


# Menu-Driven To-Do List Application
def menu():
    to_do_stack = ToDoStack()

    while True:
        print("\n=== Stack-Based To-Do List ===")
        print("1. Add Task")
        print("2. Remove Task")
        print("3. Display Tasks")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == "1":
            description = input("Enter task description: ")
            priority = input("Enter task priority: ")
            to_do_stack.add_task(description, priority)
        elif choice == "2":
            to_do_stack.remove_task()
        elif choice == "3":
            to_do_stack.display_tasks()
        elif choice == "4":
            print("Exiting the application.")
```

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


# Run the program
if __name__ == "__main__":
    menu()
```

5.queue:customer

```
class CustomerRequest:
    def __init__(self, request_id, customer_name, issue):
        self.request_id = request_id
        self.customer_name = customer_name
        self.issue = issue
        self.next = None


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

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

    def add_request(self, request_id, customer_name, issue):
        """Add a new customer request to the queue."""
        new_request = CustomerRequest(request_id, customer_name, issue)
        if self.rear is None:
            self.front = self.rear = new_request
        else:
            self.rear.next = new_request
            self.rear = new_request
        print(f"Request {request_id} from {customer_name} added to the queue.")

    def process_request(self):
        """Process the customer request at the front of the queue."""
        if self.is_empty():
            print("No requests to process.")
            return
        processed_request = self.front
        self.front = self.front.next
        if self.front is None:  # If the queue becomes empty
```

```
         self.rear = None
      print(f"Processing request {processed_request.request_id} from
{processed_request.customer_name}: {processed_request.issue}")

   def display_requests(self):
      """Display all requests in the queue."""
      if self.is_empty():
         print("No requests in the queue.")
         return
      print("Current requests in the queue:")
      current = self.front
      while current:
         print(f"Request ID: {current.request_id}, Customer: {current.customer_name},
Issue: {current.issue}")
         current = current.next


def menu():
   queue = CallCenterQueue()
   while True:
      print("\n******* Call Center Queue Management *******")
      print("1. Add Customer Request")
      print("2. Process Next Request")
      print("3. Display All Requests")
      print("4. Exit")
      choice = input("Enter your choice: ")

      if choice == "1":
         request_id = input("Enter request ID: ")
         customer_name = input("Enter customer name: ")
         issue = input("Enter issue description: ")
         queue.add_request(request_id, customer_name, issue)
      elif choice == "2":
         queue.process_request()
      elif choice == "3":
         queue.display_requests()
      elif choice == "4":
         print("Exiting system...")
         break
      else:
         print("Invalid choice! Please try again.")


if __name__ == "__main__":
```

```
    menu()
```

6.circulardrive tru orderd

```
class CircularQueue:
    """
    Circular Queue implementation for managing drive-thru orders.
    """
    def __init__(self, max_size):
        self.queue = [None] * max_size  # Fixed-size list
        self.front = -1  # Points to the first element
        self.rear = -1   # Points to the last element
        self.max_size = max_size

    def is_empty(self):
        """
        Check if the queue is empty.
        """
        return self.front == -1

    def is_full(self):
        """
        Check if the queue is full.
        """
        return (self.rear + 1) % self.max_size == self.front

    def enqueue(self, order):
        """
        Add a new order to the queue.
        """
        if self.is_full():
            print("Queue is full! Cannot add more orders.")
            return
        if self.is_empty():
            self.front = 0  # Initialize front if queue was empty
        self.rear = (self.rear + 1) % self.max_size  # Circular increment
        self.queue[self.rear] = order
        print(f"Order '{order}' added to the queue.")

    def dequeue(self):
        """
        Process and remove the order at the front of the queue.
        """
        if self.is_empty():
            print("No orders to process.")
```

```
            return None
        order = self.queue[self.front]
        if self.front == self.rear:  # Only one element in the queue
            self.front = self.rear = -1  # Reset the queue
        else:
            self.front = (self.front + 1) % self.max_size  # Circular increment
        print(f"Order '{order}' processed.")
        return order

    def display_queue(self):
        """
        Display all orders in the queue.
        """
        if self.is_empty():
            print("No pending orders.")
            return
        print("\n=== Current Orders ===")
        index = self.front
        while True:
            print(self.queue[index])
            if index == self.rear:
                break
            index = (index + 1) % self.max_size


# Menu-Driven Application
def menu():
    max_size = int(input("Enter maximum number of orders the queue can handle: "))
    drive_thru_queue = CircularQueue(max_size)

    while True:
        print("\n=== Drive-Thru Order Management ===")
        print("1. Add Order")
        print("2. Process Order")
        print("3. Display Orders")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == "1":
            order = input("Enter order description: ")
            drive_thru_queue.enqueue(order)
        elif choice == "2":
            drive_thru_queue.dequeue()
        elif choice == "3":
```

```
            drive_thru_queue.display_queue()
        elif choice == "4":
            print("Exiting the system.")
            break
        else:
            print("Invalid choice! Please try again.")


# Run the program
if __name__ == "__main__":
    menu()
```