ASSIGNMENT-11.3

NAME:G.Harika

HT NO:2303A51612

BATCH:05

Task-01:

Prompt:

SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support efficient addition, searching, and deletion of contacts.

1. Implement the contact manager using arrays (lists).
2. Implement the same functionality using a linked list for dynamic memory allocation.
3. Implement the following operations in both approaches Add a contact,Search for a contact Compare array vs. linked list approaches with respect to:Insertion efficiency,Deletion efficiency

Code:

```
class ArrayContactManager:
    def __init__(self):
self.contacts = []
    def add_contact(self, name, phone):
        self.contacts.append({'name': name, 'phone': phone})
    def search_contact(self, name):
for contact in self.contacts:
if contact['name'] == name:
return contact         return "Contact
not found."
    def delete_contact(self, name):
        for i, contact in enumerate(self.contacts):
if contact['name'] == name:
                del self.contacts[i]
                return "Contact deleted."
return "Contact not found."
# Linked List-based Contact Manager
class Node:     def __init__(self,
name, phone):
```

```
        self.name = name
self.phone = phone          self.next
= None class
LinkedListContactManager:
    def __init__(self):
self.head = None
    def add_contact(self, name, phone):
new_node = Node(name, phone)
new_node.next = self.head          self.head
= new_node
    def search_contact(self, name):
        current = self.head
while current:
            if current.name == name:                  return {'name':
current.name, 'phone': current.phone}              current
=current.next
  return "Contact not found."
    def delete_contact(self, name):
        current = self.head
prev = None          while
current:
            if current.name == name:
                if prev:
                    prev.next = current.next
                else:
                    self.head = current.next                  return
"Contact deleted."              prev = current
current = current.next          return "Contact not
found." array_manager = ArrayContactManager()
array_manager.add_contact("Alice", "123-456-7890")
```

print(array_manager.search_contact("Alice")) # Output: {'name': 'Alice', 'phone': '123-456-7890'}

print(array_manager.delete_contact("Alice")) # Output: Contact deleted.

print(array_manager.search_contact("Alice")) # Output: Contact not found.

Linked List-based Contact Manager linked_list_manager

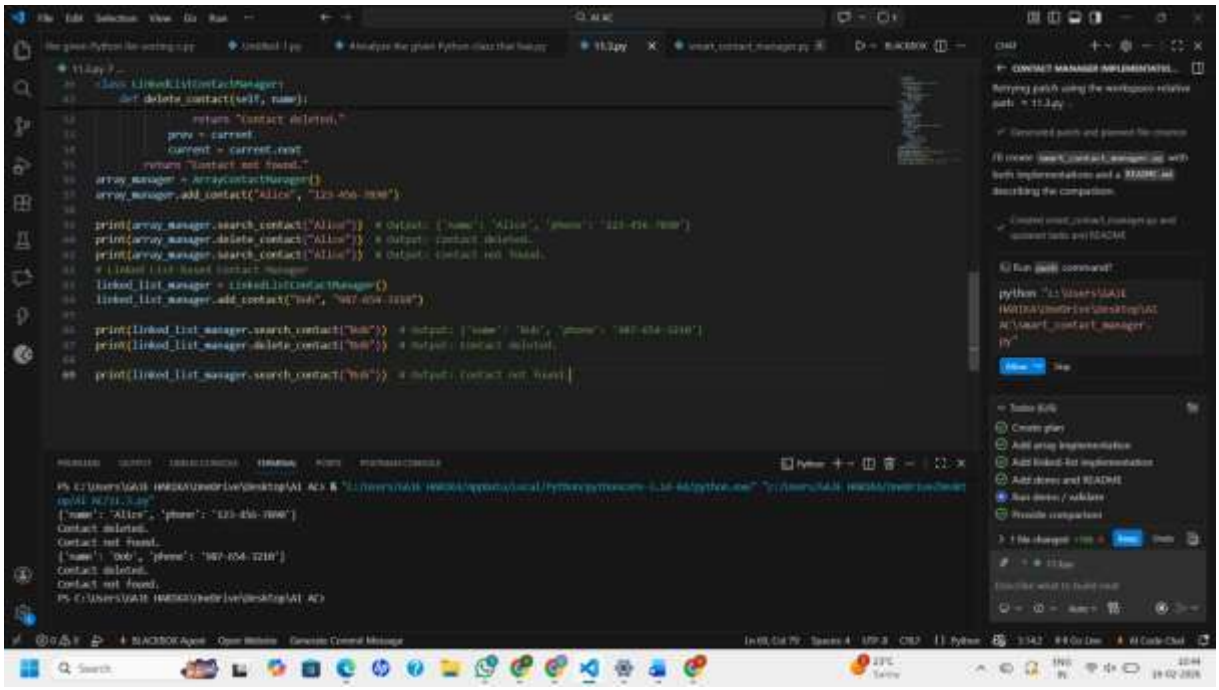
= LinkedListContactManager()

linked_list_manager.add_contact("Bob", "987-654-3210")

print(linked_list_manager.search_contact("Bob")) # Output: {'name': 'Bob', 'phone': '987-654-3210'}

print(linked_list_manager.delete_contact("Bob")) # Output: Contact deleted.

print(linked_list_manager.search_contact("Bob")) # Output: Contact not found.



Comparison:

Performance Comparison

Insertion Efficiency:

- Array-based: O(1) for appending a contact.

- Linked List-based: O(1) for adding a contact at the head.

Deletion Efficiency:

- Array-based: O(n) in the worst case (if the contact is at the end).

- Linked List-based: O(n) in the worst case (if the contact is atthe end), but O(1) if the contact is at the head.

Conclusion:

Both implementations have similar insertion efficiency, but the linkedlist can be more efficient for deletions if the contact is near the head,while the array may require shifting elements, leading to O(n) time complexity.

Task-02

Prompt:

The SRU Library manages book borrow requests. Students and faculty submit requests,

but faculty requests must be prioritized over student requests.

Tasks

1. Implement a Queue (FIFO) to manage book requests.

2. Extend the system to a Priority Queue, prioritizing faculty requests.

3. Use GitHub Copilot to assist in generating:enqueue() method, dequeue() method

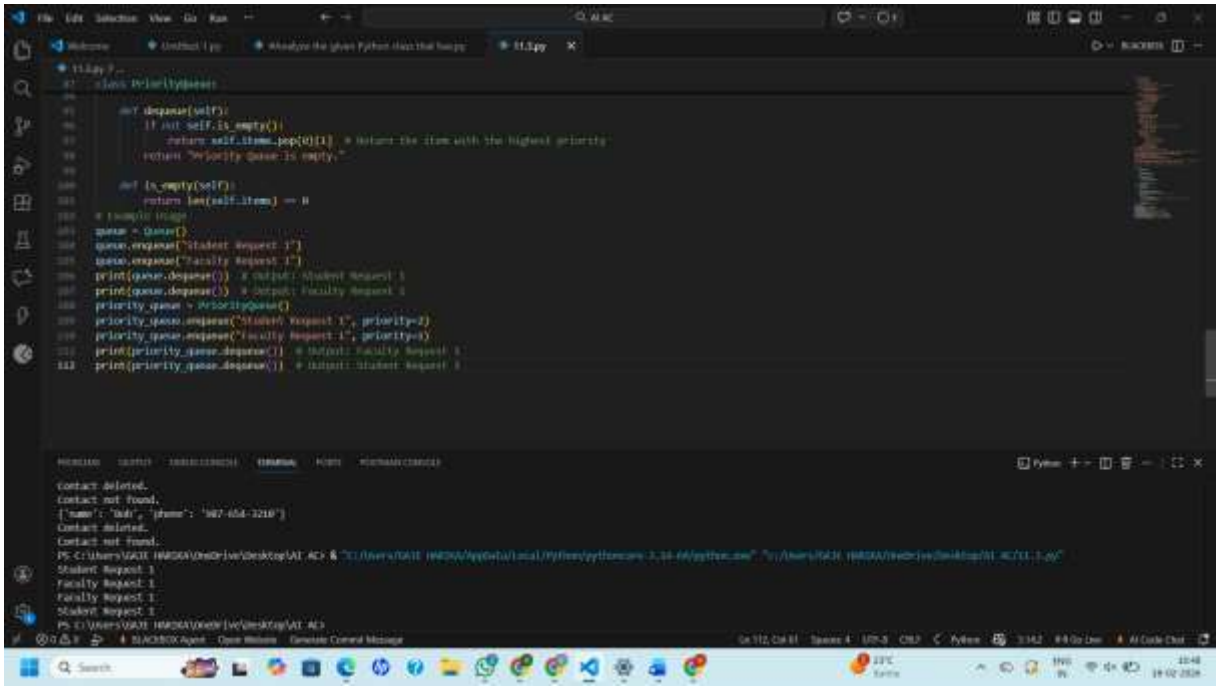
4. Test the system with a mix of student and faculty requests.

Working queue and priority queue implementations,Correct prioritization of faculty requests.

Code:

```
class Queue:     def
__init__(self):
self.items = []
   def enqueue(self, item):
self.items.append(item)
   def dequeue(self):         if
not self.is_empty():
return self.items.pop(0)
return "Queue is empty."
   def is_empty(self):
       return len(self.items) == 0
class PriorityQueue:     def
__init__(self):         self.items
= []
   de              enqueue(self,              item,              priority):
self.items.append((priority,                                  item))
self.items.sort(key=lambda x: x[0])  # Sort by priority
```

```
  def dequeue(self):        if
not self.is_empty():
            return self.items.pop(0)[1]  # Return the item with the highest priority
return "Priority Queue is empty."

  def is_empty(self):
      return len(self.items) == 0
# Example Usage queue = Queue() queue.enqueue("Student
Request 1") queue.enqueue("Faculty Request 1")
print(queue.dequeue())  # Output: Student Request 1
print(queue.dequeue())  # Output: Faculty Request 1
priority_queue = PriorityQueue()
priority_queue.enqueue("Student Request 1", priority=2)
priority_queue.enqueue("Faculty Request 1", priority=1)
print(priority_queue.dequeue())  # Output: Faculty Request 1
print(priority_queue.dequeue())  # Output: Student Request 1
```



Explanation:

The code implements two classes, Queue and PriorityQueue, to manage book requests in a library system. The Queue class follows the First-In-First-Out (FIFO) principle, allowing items to be enqueued and dequeued in the order they were added. The PriorityQueue class allows for prioritization of requests, where each item is associated with a priority level. When items are enqueued, they are sorted based on their priority, ensuring that higher-priority items are dequeued first. The example usage demonstrates how both classes work, showing the correct order of processing requests based on their type (student vs. faculty).

Task-03:

Prompt:

SR University's IT Help Desk receives technical support tickets from students and

staff. While tickets are received sequentially, issue escalation follows a

Last-In, First-Out (LIFO) approach. implement a Stack to manage support tickets.

Operations like push(ticket),pop(), peek()

Simulate at least five tickets being raised and resolved.

1.Checking whether the stack is empty

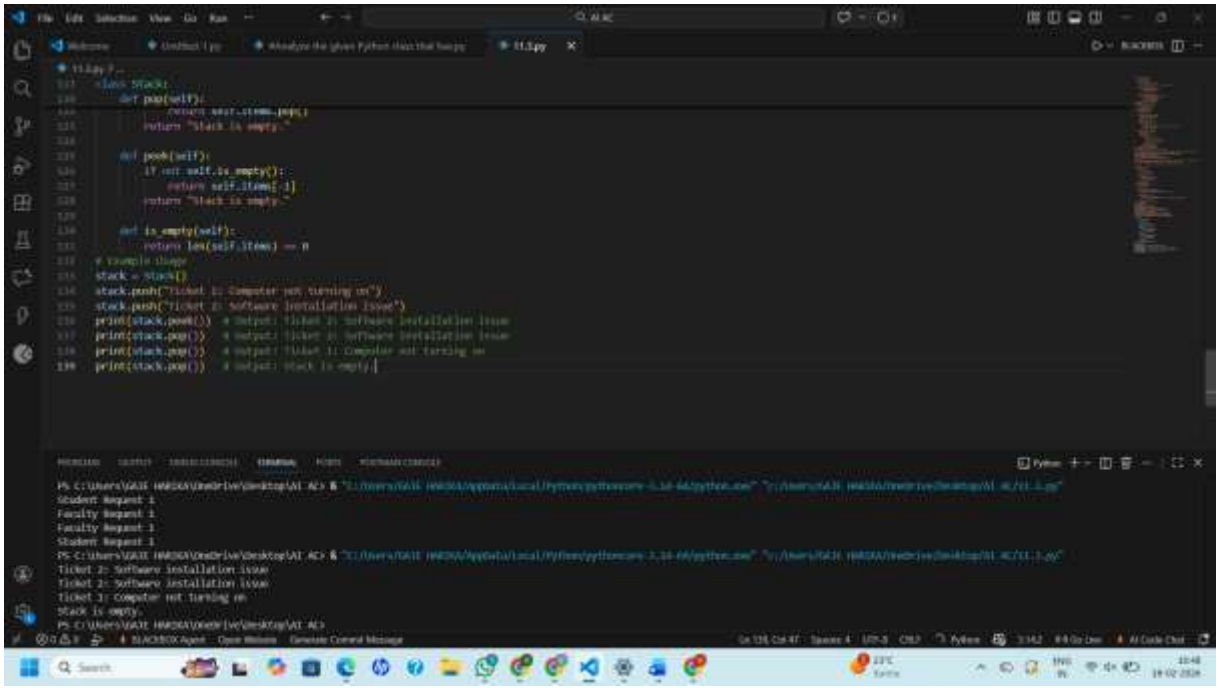
2.Checking whether the stack is full (if applicable)

Code:

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


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


   def pop(self):        if not
self.is_empty():            return
self.items.pop()        return
"Stack is empty."


  def peek(self):
      if not self.is_empty():
return self.items[-1]       return
"Stack is empty."    def
is_empty(self):
```

return len(self.items) == 0

Example Usage stack = Stack() stack.push("Ticket 1: Computer not turning on") stack.push("Ticket 2: Software installation issue") print(stack.peek()) # Output: Ticket 2: Software installation issue print(stack.pop()) # Output: Ticket 2: Software installation issue print(stack.pop()) # Output: Ticket 1: Computer not turning on print(stack.pop()) # Output: Stack is empty.



Explanation:

The Stack class implements a basic stack data structure using a list. It provides methods to push items onto the stack, pop items from the stack, peek at the top item, and check if the stack is empty. The example usage demonstrates how to use the stack to manage support tickets, showing the LIFO behavior where the most recently added ticket is resolved first.

Task-04:

Prompt:

To implement a Hash Table and understand collision handling. generate a hash table with:

Insert, Search,Delete

Starter Code class

HashTable:

pass

Collision handling using chaining,Well-commented methods

Code:

```
class HashTable:     def
__init__(self, size=10):
       self.size = size         self.table = [[] for _ in range(size)]  # Create a list of
empty lists for chaining


   def _hash(self, key):
       return hash(key) % self.size  # Simple hash function


   def insert(self, key, value):
index = self._hash(key)
       # Check if the key already exists and update it
for i, (k, v) in enumerate(self.table[index]):
           if k == key:                 self.table[index][i] = (key, value)  #
Update existing key
               return
       # If the key does not exist, add a new key-value pair
self.table[index].append((key, value))


  def search(self, key):
      index = self._hash(key)
for k, v in self.table[index]:
          if k == key:
              return v  # Return the value associated with the key
      return "Key not found."


   def delete(self, key):
       index = self._hash(key)         for i, (k, v) in
enumerate(self.table[index]):             if k ==
key:
```

del self.table[index][i] # Remove the key-value pair

return "Key deleted." return "Key not found." # Example

Usage hash_table = HashTable() hash_table.insert("name",

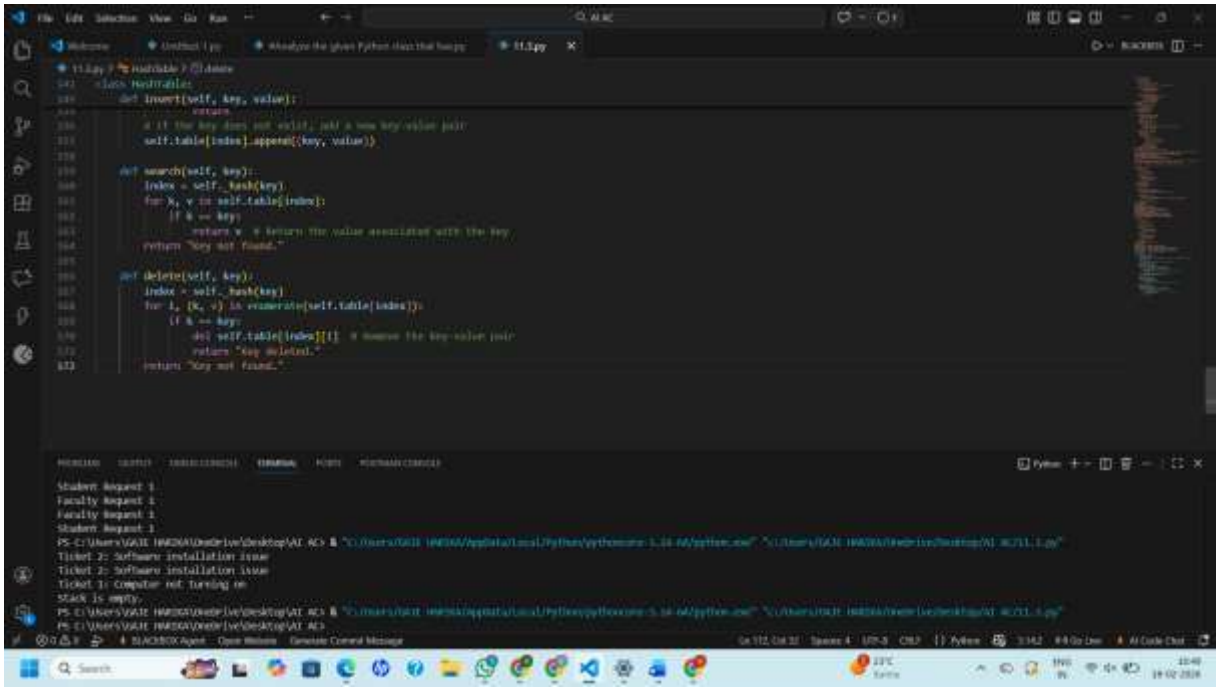
"Alice") hash_table.insert("age", 30)

print(hash_table.search("name")) # Output: Alice

print(hash_table.search("age")) # Output: 30

print(hash_table.delete("name")) # Output: Key deleted.

print(hash_table.search("name")) # Output: Key not found.



Explanation:

The HashTable class implements a simple hash table with chaining for collision handling.

The constructor initializes the hash table with a specified size and creates a list of empty lists for chaining.

The _hash method computes the hash value of a key and maps it to an index in the table.

The insert method checks if the key already exists in the corresponding bucket (list) and updates it if found; otherwise, it appends a new key-value pair to the bucket.

The search method looks for the key in the appropriate bucket and returns its value if found, or a "Key not found" message if it does not exist.

The delete method searches for the key in the bucket and removes it if found, returning a "Key deleted" message; if the key is not found, it returns a "Key not found" message.

Task-05

Prompt:

Design a Campus Resource Management System with the following features:

• Student Attendance Tracking

• Event Registration System

• Library Book Borrowing

• Bus Scheduling System

• Cafeteria Order Queue

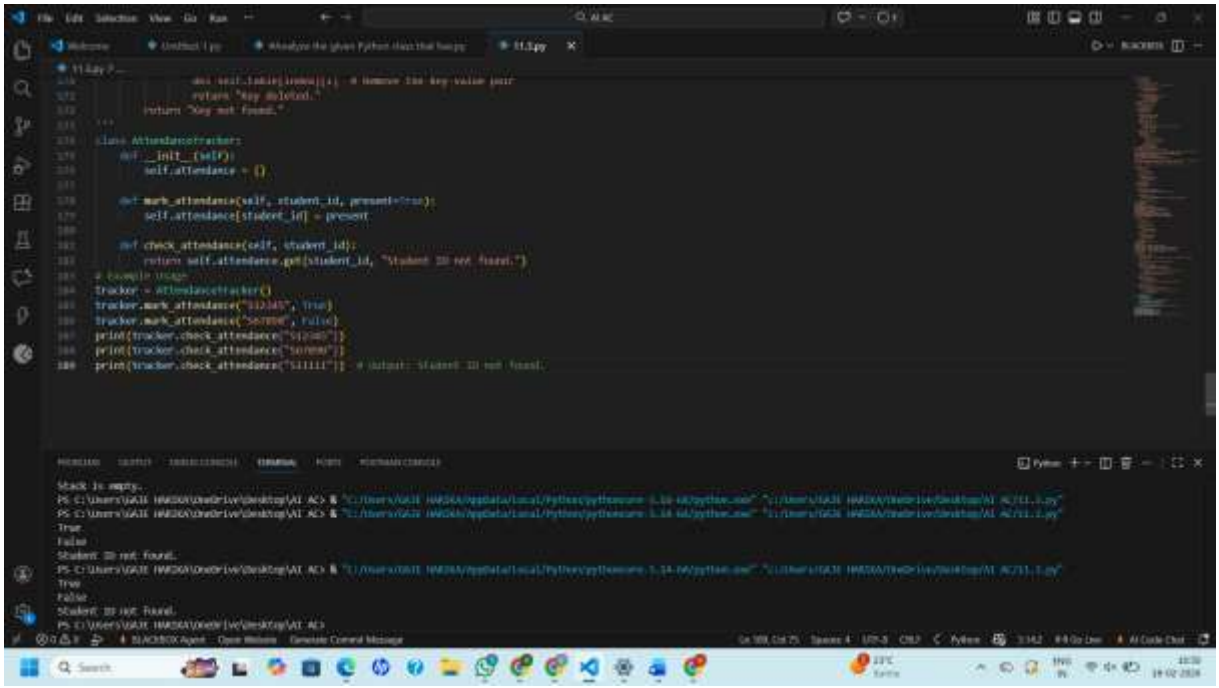
Choose the most appropriate data structure for each feature.

2. Justify your choice in 2–3 sentences.

3. Implement one selected feature using AI-assisted code generation.

Code:

```
class AttendanceTracker:
def __init__(self):
        self.attendance = {}


    def mark_attendance(self, student_id, present=True):
        self.attendance[student_id] = present


    def check_attendance(self, student_id):
        return self.attendance.get(student_id, "Student ID not found.")
# Example Usage
tracker = AttendanceTracker() tracker.mark_attendance("S12345", True)
tracker.mark_attendance("S67890", False)
print(tracker.check_attendance("S12345"))
print(tracker.check_attendance("S67890"))
print(tracker.check_attendance("S11111"))  # Output: Student ID not found.
```



Explanation:

The AttendanceTracker class uses a hash table (dictionary) to store attendance records, where the student ID is the key and the attendance status (present or not) is the value. The mark_attendance method allows marking a student's attendance, while the check_attendance method retrieves the attendance status for a given student ID, returning a message if the ID is not found.