

School of Computer Science and Artificial Intelligence

Lab Assignment # 11.3

Program : B. Tech (CSE)
Specialization : -
Course Title : AI Assisted Coding
Course Code : 23CS002PC304
Semester II
Academic Session : 2025-2026
Name of Student : P.Eshwar
Enrollment No. : 2403A51L26
Batch No. 51
Date : 24/02/26

Submission Starts here

Screenshots:

Task Description #1

Smart Contact Manager (Arrays & Linked Lists)

Scenario

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.

Tasks

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:
 - o Add a contact
 - o Search for a contact
 - o Delete a contact
 4. Use GitHub Copilot to assist in generating search and delete methods.
 5. Compare array vs. linked list approaches with respect to:
 - o Insertion efficiency
 - o Deletion efficiency
- Expected Outcome
- Two working implementations (array-based and linked-list-based).
 - A brief comparison explaining performance differences.

```
ai_coding.py > ⚡ ArrayContactManager > ✎ delete_contact
1 # Smart Contact Manager (Arrays & Linked Lists)
2 # Scenario
3 # SR University's student club requires a simple Contact Manager Application
4 # to store members' names and phone numbers. The system should support
5 # efficient addition, searching, and deletion of contacts.
6 # Tasks
7 # 1. Implement the contact manager using arrays (lists).
8 # 2. Implement the same functionality using a linked list for dynamic
9 # memory allocation.
10 # 3. Implement the following operations in both approaches:
11 # o Add a contact
12 # o Search for a contact
13 # o Delete a contact
14 # 4. Use GitHub Copilot to assist in generating search and delete methods.
15 # 5. Compare array vs. linked list approaches with respect to:
16 # o Insertion efficiency
17 # o Deletion efficiency
18 # Expected Outcome
19 # • Two working implementations (array-based and linked-list-based).
20 # • A brief comparison explaining performance differences.
21 # Array-based Contact Manager
22 class ArrayContactManager:
23     def __init__(self):
24         self.contacts = []
25
26     def add_contact(self, name, phone):
27         self.contacts.append({'name': name, 'phone': phone})
28
29     def search_contact(self, name):
30         for contact in self.contacts:
31             if contact['name'] == name:
32                 return contact
33         return None
34
35     def delete_contact(self, name):
36         for i, contact in enumerate(self.contacts):
37             if contact['name'] == name:
```

```
ai_coding.py > ArrayContactManager > delete_contact
22  class ArrayContactManager:
35      def delete_contact(self, name):
36          for i, contact in enumerate(self.contacts):
37              if contact['name'] == name:
38                  del self.contacts[i]
39                  return True
40          return False
41 # Linked List-based Contact Manager
42 class Node:
43     def __init__(self, name, phone):
44         self.name = name
45         self.phone = phone
46         self.next = None
47
48 class LinkedListContactManager:
49     def __init__(self):
50         self.head = None
51
52     def add_contact(self, name, phone):
53         new_node = Node(name, phone)
54         new_node.next = self.head
55         self.head = new_node
56
57     def search_contact(self, name):
58         current = self.head
59         while current:
60             if current.name == name:
61                 return {'name': current.name, 'phone': current.phone}
62             current = current.next
63     return None
64
65     def delete_contact(self, name):
66         current = self.head
67         prev = None
68         while current:
69             if current.name == name:
70                 if prev:
```

```

ai_coding.py > ⚡ ArrayContactManager > ⚡ delete_contact
48  class LinkedListContactManager:
49      def delete_contact(self, name):
50          current = self.head
51          prev = None
52          while current:
53              if current.name == name:
54                  if prev:
55                      prev.next = current.next
56                  else:
57                      self.head = current.next
58                  return True
59              prev = current
60              current = current.next
61          return False
62
63      # Performance Comparison
64      # Insertion Efficiency:
65      # - Array-based: O(1) for appending a contact.
66      # - Linked List-based: O(1) for adding a contact at the head.
67      # Deletion Efficiency:
68      # - Array-based: O(n) in the worst case (if the contact is at the end).
69      # - Linked List-based: O(n) in the worst case (if the contact is at the end).
70      # Overall, both approaches have similar time complexities for insertion and deletion, but the linked list can be more memory efficient for large data sets.
71
72      # Example Usage
73  if __name__ == "__main__":
74      # Using ArrayContactManager
75      array_manager = ArrayContactManager()
76      array_manager.add_contact("Alice", "1234567890")
77      array_manager.add_contact("Bob", "0987654321")
78      print(array_manager.search_contact("Alice")) # {'name': 'Alice', 'phone': '1234567890'}
79      print(array_manager.delete_contact("Bob")) # True
80      print(array_manager.search_contact("Bob")) # None
81
82      # Using LinkedListContactManager
83      linked_list_manager = LinkedListContactManager()

```

Output:

```

ai_coding.py > ⚡ ArrayContactManager > ⚡ delete_contact
90  array_manager = ArrayContactManager()
91  array_manager.add_contact("Alice", "1234567890")
92  array_manager.add_contact("Bob", "0987654321")
93  print(array_manager.search_contact("Alice")) # {'name': 'Alice', 'phone': '1234567890'}
94  print(array_manager.delete_contact("Bob")) # True
95  print(array_manager.search_contact("Bob")) # None
96
97  # Using LinkedListContactManager
98  linked_list_manager = LinkedListContactManager()
99  linked_list_manager.add_contact("Charlie", "5555555555")
100 linked_list_manager.add_contact("Dave", "4444444444")
101 print(linked_list_manager.search_contact("Charlie")) # {'name': 'Charlie', 'phone': '5555555555'}
102 print(linked_list_manager.delete_contact("Dave")) # True
103 print(linked_list_manager.search_contact("Dave")) # None
104
105

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.py

{'name': 'Alice', 'phone': '1234567890'}
True
None
{'name': 'charlie', 'phone': '5555555555'}
True
None

PS C:\Users\Eshwar\OneDrive\Desktop\python>

EXPLAINATION:

1. Array version is simple and memory efficient.
2. Linked list version is flexible and faster for insertions/deletions (especially at beginning).
3. Both structures take O(n) time for searching.

Task 2: Library Book Search System (Queues & Priority Queues)

Scenario

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:
 - o enqueue() method
 - o dequeue() method
4. Test the system with a mix of student and faculty requests.

Expected Outcome

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

```

ai_coding.py > ...
1  # Task 2: Library Book Search System (Queues & Priority Queues)
2  # Scenario
3  # The SRU Library manages book borrow requests. Students and faculty submit
4  # requests, but faculty requests must be prioritized over student requests.
5  # Tasks
6  # 1. Implement a Queue (FIFO) to manage book requests.
7  # 2. Extend the system to a Priority Queue, prioritizing faculty requests.
8  # 3. Use GitHub Copilot to assist in generating:
9  # o enqueue() method
10 # o dequeue() method
11 # 4. Test the system with a mix of student and faculty requests.
12 # Expected Outcome
13 # • Working queue and priority queue implementations.
14 # • Correct prioritization of faculty requests
15 < class BookRequest:
16     < def __init__(self, requester_type, book_title):
17         self.requester_type = requester_type # 'student' or 'faculty'
18         self.book_title = book_title
19
20     < def __repr__(self):
21         return f"{self.requester_type.capitalize()} request for '{self.book_title}'"
22
23 < class BookRequestQueue:
24     < def __init__(self):
25         self.queue = []
26
27     < def enqueue(self, request):
28         self.queue.append(request)
29
30     < def dequeue(self):
31         if not self.is_empty():
32             return self.queue.pop(0)
33         else:
34             raise IndexError("Queue is empty")
35
36     < def is_empty(self):
37         return len(self.queue) == 0

```

```
ai_coding.py > ...
23  class BookRequestQueue:
37  |     return len(self.queue) == 0
38  class PriorityBookRequestQueue:
39      def __init__(self):
40          self.faculty_queue = []
41          self.student_queue = []
42
43      def enqueue(self, request):
44          if request.requester_type == 'faculty':
45              self.faculty_queue.append(request)
46          else:
47              self.student_queue.append(request)
48
49      def dequeue(self):
50          if not self.is_empty():
51              if self.faculty_queue:
52                  return self.faculty_queue.pop(0)
53              else:
54                  return self.student_queue.pop(0)
55          else:
56              raise IndexError("Queue is empty")
57
58      def is_empty(self):
59          return len(self.faculty_queue) == 0 and len(self.student_queue) == 0
60
# Testing the system
61 if __name__ == "__main__":
62     # Create a priority queue
63     priority_queue = PriorityBookRequestQueue()
64
65     # Enqueue some requests
66     priority_queue.enqueue(BookRequest('student', 'Introduction to Algorithms'))
67     priority_queue.enqueue(BookRequest('faculty', 'Advanced Machine Learning'))
68     priority_queue.enqueue(BookRequest('student', 'Data Structures in Python'))
69     priority_queue.enqueue(BookRequest('faculty', 'Artificial Intelligence: A Modern Approach'))
70
71     # Dequeue and print requests
72     while not priority_queue.is_empty():
```

Output:

```

04
65     # Enqueue some requests
66     priority_queue.enqueue(BookRequest('student', 'Introduction to Algorithms'))
67     priority_queue.enqueue(BookRequest('faculty', 'Advanced Machine Learning'))
68     priority_queue.enqueue(BookRequest('student', 'Data Structures in Python'))
69     priority_queue.enqueue(BookRequest('faculty', 'Artificial Intelligence: A Modern Approach'))
70
71     # Dequeue and print requests
72     while not priority_queue.is_empty():
73         request = priority_queue.dequeue()
74         print(request)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python.py
Faculty request for 'Advanced Machine Learning'
Faculty request for 'Artificial Intelligence: A Modern Approach'
Student request for 'Introduction to Algorithms'
Student request for 'Data Structures in Python'
PS C:\Users\Eshwar\OneDrive\Desktop\python>

```

EXPLANATION:

1. Queue → serves requests in arrival order.
2. Priority Queue → serves based on importance.
3. Faculty are assigned **higher priority value**, so they are processed first.
4. A counter is used to maintain order when priorities are equal.

Task 3: Emergency Help Desk (Stack Implementation)

Scenario

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.

Tasks

1. Implement a Stack to manage support tickets.
2. Provide the following operations:
 - o push(ticket)
 - o pop()
 - o peek()
3. Simulate at least five tickets being raised and resolved.
4. Use GitHub Copilot to suggest additional stack operations such as:
 - o Checking whether the stack is empty
 - o Checking whether the stack is full (if applicable)

Expected Outcome

- Functional stack-based ticket management system.
- Clear demonstration of LIFO behavior.

```
ai_coding.py > ...
1 # Task 3: Emergency Help Desk (Stack Implementation)
2 # Scenario
3 # SR University's IT Help Desk receives technical support tickets from students
4 # and staff. While tickets are received sequentially, issue escalation follows a
5 # Last-In, First-Out (LIFO) approach.
6 # Tasks
7 # 1. Implement a Stack to manage support tickets.
8 # 2. Provide the following operations:
9 # o push(ticket)
10 # o pop()
11 # o peek()
12 # 3. Simulate at least five tickets being raised and resolved.
13 # 4. Use GitHub Copilot to suggest additional stack operations such as:
14 # o Checking whether the stack is empty
15 # o Checking whether the stack is full (if applicable)
16 # Expected Outcome
17 # • Functional stack-based ticket management system.
18 # • Clear demonstration of LIFO behavior.
19 class Stack:
20     def __init__(self, capacity):
21         self.stack = []
22         self.capacity = capacity
23
24     def push(self, ticket):
25         if len(self.stack) < self.capacity:
26             self.stack.append(ticket)
27             print(f"Ticket '{ticket}' added to the stack.")
28         else:
29             print("Stack is full. Cannot add more tickets.")
30
31     def pop(self):
32         if not self.is_empty():
33             ticket = self.stack.pop()
34             print(f"Ticket '{ticket}' resolved and removed from the stack.")
35             return ticket
36         else:
37             print("Stack is empty. No tickets to resolve.")
```

Ln 72, Col

```

ai_coding.py > ...
19   class Stack:
39
40       def peek(self):
41           if not self.is_empty():
42               print(f"Next ticket to resolve: '{self.stack[-1]}'")
43               return self.stack[-1]
44           else:
45               print("Stack is empty. No tickets to peek.")
46               return None
47
48       def is_empty(self):
49           return len(self.stack) == 0
50
51       def is_full(self):
52           return len(self.stack) >= self.capacity
53
# Simulating ticket management
54 help_desk = Stack(capacity=5)
55 help_desk.push("Ticket 1: Computer won't start")
56 help_desk.push("Ticket 2: Software installation issue")
57 help_desk.push("Ticket 3: Network connectivity problem")
58 help_desk.push("Ticket 4: Printer not working")
59 help_desk.push("Ticket 5: Email not syncing")
60 help_desk.peek()
61 help_desk.pop()
62 help_desk.pop()
63 help_desk.peek()
64 help_desk.push("Ticket 6: Password reset")
65 help_desk.peek()
66 help_desk.pop()
67 help_desk.pop()
68 help_desk.pop()
69 help_desk.pop() # Attempting to pop from an empty stack
70 help_desk.is_empty()
71 help_desk.is_full()
72
73

```

Output:

```

Ticket 'Ticket 4: Printer not working' added to the stack.
Ticket 'Ticket 5: Email not syncing' added to the stack.
Next ticket to resolve: 'Ticket 5: Email not syncing'
Ticket 'Ticket 5: Email not syncing' resolved and removed from the stack.
Ticket 'Ticket 4: Printer not working' resolved and removed from the stack.
Next ticket to resolve: 'Ticket 3: Network connectivity problem'
Ticket 'Ticket 6: Password reset' added to the stack.
Next ticket to resolve: 'Ticket 6: Password reset'
Ticket 'Ticket 6: Password reset' resolved and removed from the stack.
Ticket 'Ticket 3: Network connectivity problem' resolved and removed from the stack.
Ticket 'Ticket 2: Software installation issue' resolved and removed from the stack.
Ticket 'Ticket 1: Computer won't start' resolved and removed from the stack.

```

EXPLANATION:

1. Queue ensures fairness (first come, first served).
2. Priority Queue ensures importance (faculty first).
3. Both structures together simulate a realistic library request system.

Task 4: Hash Table**Objective**

To implement a Hash Table and understand collision handling.

Task Description

Use AI to generate a hash table with:

- Insert
- Search
- Delete

Starter Code

```
class HashTable:
```

```
    pass
```

Expected Outcome

- Collision handling using chaining
- Well-commented methods

```
ai_coding.py > ...
1  # Task 4: Hash Table
2  # Objective
3  # To implement a Hash Table and understand collision handling.
4  # Task Description
5  # Use AI to generate a hash table with:
6  # • Insert
7  # • Search
8  # • Delete
9  # Starter Code
10 # class HashTable:
11 # pass
12 # Expected Outcome
13 # • Collision handling using chaining
14 # • Well-commented methods
15 class HashTable:
16     def __init__(self, size=10):
17         """Initialize the hash table with a specified size."""
18         self.size = size
19         self.table = [[] for _ in range(size)] # Create a list of empty lists for chaining
20
21     def _hash(self, key):
22         """Generate a hash for the given key."""
23         return hash(key) % self.size
24
25     def insert(self, key, value):
26         """Insert a key-value pair into the hash table."""
27         index = self._hash(key)
28         # Check if the key already exists and update it
29         for i, (k, v) in enumerate(self.table[index]):
30             if k == key:
31                 self.table[index][i] = (key, value) # Update existing key
32                 return
33         # If the key does not exist, add a new key-value pair
34         self.table[index].append((key, value))
35
36     def search(self, key):
37         """Search for a value by its key in the hash table."""
```

```

ai_coding.py > ...
15  class HashTable:
36      def search(self, key):
37          """Search for a value by its key in the hash table."""
38          index = self._hash(key)
39          for k, v in self.table[index]:
40              if k == key:
41                  return v # Return the value if the key is found
42          return None # Return None if the key is not found
43
44      def delete(self, key):
45          """Delete a key-value pair from the hash table."""
46          index = self._hash(key)
47          for i, (k, v) in enumerate(self.table[index]):
48              if k == key:
49                  del self.table[index][i] # Remove the key-value pair
50                  return True # Return True if deletion was successful
51          return False # Return False if the key was not found
52
53 # Example usage:
54 if __name__ == "__main__":
55     hash_table = HashTable()
56     hash_table.insert("name", "Alice")
57     hash_table.insert("age", 30)
58     print(hash_table.search("name")) # Output: Alice

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.py
Alice
30
None
PS C:\Users\Eshwar\OneDrive\Desktop\python>

```

EXPLAINATION:

1. Structure

A hash table uses an array of buckets. Each bucket stores a list of key–value pairs. A hash function converts a key into an index where data should be stored.

2. Collision Handling (Chaining)

If two keys map to the same index, they are stored together in that bucket's list instead of replacing each other. This method is called chaining.

3. Operations

- Insert: Hash key → go to bucket → add/update pair.
- Search: Hash key → check bucket list → return value if found.
- Delete: Hash key → find pair in bucket → remove it.

Task 5: Real-Time Application Challenge

Scenario

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

Student Tasks

1. 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.

Expected Outcome

- Mapping table: Feature → Data Structure → Justification
- One fully working Python implementation

```

ai_coding.py > ...
1 # Task 5: Real-Time Application Challenge
2 # Scenario
3 # Design a Campus Resource Management System with the following
4 # features:
5 # • Student Attendance Tracking
6 # • Event Registration System
7 # • Library Book Borrowing
8 # • Bus Scheduling System
9 # • Cafeteria Order Queue
10 # Student Tasks
11 # 1. Choose the most appropriate data structure for each feature.
12 # 2. Justify your choice in 2-3 sentences.
13 # 3. Implement one selected feature using AI-assisted code generation.
14 # Expected Outcome
15 # • Mapping table: Feature → Data Structure → Justification
16 # • One fully working Python implementation
17 # Mapping table: Feature → Data Structure + Justification
18 mapping_table = {
19     "Student Attendance Tracking": ("Hash Table", "A hash table allows for efficient storage and retrieval of student attendance records using unique keys (student IDs)."),
20     "Event Registration System": ("Queue", "A queue is suitable for managing event registrations as it follows a first-come, first-served approach, allowing for easy insertion and removal of events from both ends."),
21     "Library Book Borrowing": ("Linked List", "A linked list can efficiently manage the dynamic nature of book borrowing and returning, allowing for easy insertion and deletion of books at any position in the list."),
22     "Bus Scheduling System": ("Graph", "A graph can represent the bus routes and stops effectively, allowing for complex scheduling and route optimization based on stops and times."),
23     "Cafeteria Order Queue": ("Queue", "A queue is ideal for managing cafeteria orders as it ensures that orders are processed in the order they were received, maintaining FIFO (First-In, First-Out) behavior."})
24 }
25 # Implementing the Student Attendance Tracking feature using a Hash Table
26 class AttendanceTracker:
27     def __init__(self):
28         self.attendance_records = {}
29
30     def mark_attendance(self, student_id, date):
31         if student_id not in self.attendance_records:
32             self.attendance_records[student_id] = []
33         self.attendance_records[student_id].append(date)
34
35     def get_attendance(self, student_id):
36         return self.attendance_records.get(student_id, [])
37
38
39
40
41
42
43
44
45
46
47

```

```

ai_coding.py > ...
26     class AttendanceTracker:
27         def __init__(self):
28             self.attendance_records = {}
29
30         def mark_attendance(self, student_id, date):
31             self.attendance_records[student_id].append(date)
32
33         def get_attendance(self, student_id):
34             return self.attendance_records.get(student_id, [])
35
36
37 # Example usage
38 attendance_tracker = AttendanceTracker()
39 attendance_tracker.mark_attendance("S001", "2024-09-01")
40 attendance_tracker.mark_attendance("S001", "2024-09-02")
41 attendance_tracker.mark_attendance("S002", "2024-09-01")
42 print(attendance_tracker.get_attendance("S001")) # Output: ['2024-09-01', '2024-09-02']
43 print(attendance_tracker.get_attendance("S002")) # Output: ['2024-09-01']
44 print(attendance_tracker.get_attendance("S003")) # Output: []
45
46
47

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.py
['2024-09-01', '2024-09-02']
['2024-09-01']
[]
PS C:\Users\Eshwar\OneDrive\Desktop\python>

```

EXPLANATION:

1. A hash table (dictionary in Python) is chosen because it provides O(1) average time complexity for both insertion and retrieval operations.

2. Each student ID is used as a key in the hash table, and the value is a list of attendance dates for that student.

3. This approach allows for efficient tracking of attendance records without needing to scan through all records every time a lookup is performed, making it scalable for large numbers of students.