

## AIAC-Ass-11.3

**Name:T.shivani**

**Ht.No:2303A51312**

**Batch:05**

**Prompt: Task 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.

**Code: class ContactManagerArray:**

```
def __init__(self):  
    self.contacts = []
```

```
def add_contact(self, name, phone):  
    self.contacts.append((name, phone))
```

```
def search_contact(self, name):  
    for contact in self.contacts:  
        if contact[0] == name:  
            return contact[1]  
    return None
```

```
def delete_contact(self, name):  
    for i, contact in enumerate(self.contacts):  
        if contact[0] == name:  
            del self.contacts[i]  
            return True  
    return False
```

# Linked list implementation

```
class ContactNode:  
    def __init__(self, name, phone):  
        self.name = name  
        self.phone = phone  
        self.next = None  
  
class ContactManagerLinkedList:  
    def __init__(self):  
        self.head = None  
  
    def add_contact(self, name, phone):  
        new_node = ContactNode(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 current.phone
```

```
        current = current.next
```

```
    return None
```

```
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 True
```

```
        prev = current
```

```
        current = current.next
```

```
    return False
```

```
#example usage
```

```
contact_manager_array = ContactManagerArray()
```

```
contact_manager_array.add_contact("Alice", "123-456-7890")
```

```
contact_manager_array.add_contact("Bob", "987-654-3210")
```

```

print(contact_manager_array.search_contact("Alice")) # Output: 123-456-7890

contact_manager_array.delete_contact("Bob")

print(contact_manager_array.search_contact("Bob")) # Output: None

contact_manager_linked_list = ContactManagerLinkedList()

contact_manager_linked_list.add_contact("Charlie", "555-555-5555")

contact_manager_linked_list.add_contact("Dave", "444-444-4444")

print(contact_manager_linked_list.search_contact("Charlie")) # Output: 555-555-5555

contact_manager_linked_list.delete_contact("Dave")

print(contact_manager_linked_list.search_contact("Dave")) # Output: None

```

The screenshot shows a VS Code editor with a Python file named 'Ass-11.3.py'. The code defines two classes: `ContactManagerArray` and `ContactManagerLinkedList`. The `ContactManagerArray` class has methods for adding, deleting, and searching contacts. The `ContactManagerLinkedList` class has methods for adding, deleting, and searching contacts. The script demonstrates the usage of both classes. The terminal output shows the results of the operations: `print(contact_manager_array.search_contact("Alice"))` outputs `123-456-7890`, `print(contact_manager_array.search_contact("Bob"))` outputs `None`, `print(contact_manager_linked_list.search_contact("Charlie"))` outputs `555-555-5555`, and `print(contact_manager_linked_list.search_contact("Dave"))` outputs `None`.

```

46 class ContactManagerArray:
47     def __init__(self):
48         self.contacts = {}
49     def add_contact(self, name, phone):
50         self.contacts[name] = phone
51     def delete_contact(self, name):
52         if name in self.contacts:
53             del self.contacts[name]
54         return True
55     def search_contact(self, name):
56         if name in self.contacts:
57             return self.contacts[name]
58         return None
59
60 class ContactManagerLinkedList:
61     def __init__(self):
62         self.head = None
63     def add_contact(self, name, phone):
64         new_contact = ContactNode(name, phone)
65         new_contact.next = self.head
66         self.head = new_contact
67     def delete_contact(self, name):
68         if self.head is None:
69             return False
70         prev = None
71         current = self.head
72         while current:
73             if current.name == name:
74                 if prev is None:
75                     self.head = current.next
76                 else:
77                     prev.next = current.next
78                 return True
79             prev = current
80             current = current.next
81         return False
82     def search_contact(self, name):
83         if self.head is None:
84             return None
85         current = self.head
86         while current:
87             if current.name == name:
88                 return current.phone
89             current = current.next
90         return None
91
92 # Sample usage
93 contact_manager_array = ContactManagerArray()
94 contact_manager_array.add_contact("Alice", "123-456-7890")
95 contact_manager_array.add_contact("Bob", "987-654-3210")
96 print(contact_manager_array.search_contact("Alice")) # Output: 123-456-7890
97 contact_manager_array.delete_contact("Bob")
98 print(contact_manager_array.search_contact("Bob")) # Output: None
99
100 contact_manager_linked_list = ContactManagerLinkedList()
101 contact_manager_linked_list.add_contact("Charlie", "555-555-5555")
102 contact_manager_linked_list.add_contact("Dave", "444-444-4444")
103 print(contact_manager_linked_list.search_contact("Charlie")) # Output: 555-555-5555
104 contact_manager_linked_list.delete_contact("Dave")
105 print(contact_manager_linked_list.search_contact("Dave")) # Output: None
106

```

```

PS C:\Users\Shivani T\OneDrive\Desktop\AI- assisted> python -c "from Ass-11.3.py import *; contact_manager_array = ContactManagerArray(); contact_manager_array.add_contact('Alice', '123-456-7890'); contact_manager_array.add_contact('Bob', '987-654-3210'); print(contact_manager_array.search_contact('Alice')); contact_manager_array.delete_contact('Bob'); print(contact_manager_array.search_contact('Bob')); contact_manager_linked_list = ContactManagerLinkedList(); contact_manager_linked_list.add_contact('Charlie', '555-555-5555'); contact_manager_linked_list.add_contact('Dave', '444-444-4444'); print(contact_manager_linked_list.search_contact('Charlie')); contact_manager_linked_list.delete_contact('Dave'); print(contact_manager_linked_list.search_contact('Dave'))"
123-456-7890
None
555-555-5555
None

```

## 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 at the 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.

**Explanation:** Array-based approach is simpler and more efficient for searching and deleting elements in a small to medium-sized list, but it has fixed size and requires shifting elements during insertion/deletion. Linked list approach is more flexible for dynamic data, but it uses more memory due to storing pointers and has slower search times.

### **Prompt: #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:

# enqueue() method

# 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.

# Queue implementation

**Code: from collections import deque**

**class BookRequestQueue:**

**def \_\_init\_\_(self):**

**self.queue = deque()**

```
def enqueue(self, request):  
    self.queue.append(request)
```

```
def dequeue(self):  
    if self.queue:  
        return self.queue.popleft()  
    return None
```

**# Priority Queue implementation**

```
class BookRequestPriorityQueue:
```

```
    def __init__(self):  
        self.faculty_queue = deque()  
        self.student_queue = deque()
```

```
    def enqueue(self, request, is_faculty=False):  
        if is_faculty:  
            self.faculty_queue.append(request)  
        else:  
            self.student_queue.append(request)
```

```
    def dequeue(self):  
        if self.faculty_queue:  
            return self.faculty_queue.popleft()  
        elif self.student_queue:  
            return self.student_queue.popleft()  
        return None
```

**# Example usage**

```
book_request_queue = BookRequestQueue()
```

```

book_request_queue.enqueue("Student Request 1")

book_request_queue.enqueue("Faculty Request 1")

print(book_request_queue.dequeue()) # Output: Student Request 1

print(book_request_queue.dequeue()) # Output: Faculty Request 1

book_request_priority_queue = BookRequestPriorityQueue()

book_request_priority_queue.enqueue("Student Request 2")

book_request_priority_queue.enqueue("Faculty Request 2", is_faculty=True)

print(book_request_priority_queue.dequeue()) # Output: Faculty Request 2

print(book_request_priority_queue.dequeue()) # Output: Student Request 2

```

The screenshot shows a VS Code editor with a Python file named 'Ass-11.3.py'. The code implements a 'BookRequestQueue' and a 'BookRequestPriorityQueue'. The 'BookRequestQueue' uses a standard queue (FIFO), while the 'BookRequestPriorityQueue' uses a priority queue where faculty requests are processed before student requests. The terminal output shows the execution of the code, demonstrating that the priority queue correctly processes faculty requests first.

```

121 # Priority Queue Implementation
122 class BookRequestPriorityQueue:
123     def __init__(self):
124         self.faculty_queue = deque()
125         self.student_queue = deque()
126
127     def enqueue(self, request, is_faculty=False):
128         if is_faculty:
129             self.faculty_queue.append(request)
130         else:
131             self.student_queue.append(request)
132
133     def dequeue(self):
134         if self.faculty_queue:
135             return self.faculty_queue.popleft()
136         elif self.student_queue:
137             return self.student_queue.popleft()
138         return None
139
140 # Example usage
141 book_request_queue = BookRequestQueue()
142 book_request_queue.enqueue("Student Request 1")
143 book_request_queue.enqueue("Faculty Request 1")
144 print(book_request_queue.dequeue()) # Output: Student Request 1
145 print(book_request_queue.dequeue()) # Output: Faculty Request 1
146
147 book_request_priority_queue = BookRequestPriorityQueue()
148 book_request_priority_queue.enqueue("Student Request 2")
149 book_request_priority_queue.enqueue("Faculty Request 2", is_faculty=True)
150 print(book_request_priority_queue.dequeue()) # Output: Faculty Request 2
151 print(book_request_priority_queue.dequeue()) # Output: Student Request 2

```

```

PS C:\Users\Shivani T\OneDrive\Desktop\AI- assisted> & 'c:\Users\Shivani T\AppData\local\Python\pythoncore-3.14-64\python.exe' 'c:\Users\Shivani T\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bu
nched\lib\debugpy\launcher' -S7365 -.- 'C:\Users\Shivani T\OneDrive\Desktop\AI- assisted\Ass-11.3.py'
Student Request 1
Faculty Request 1
Faculty Request 2
Student Request 2
PS C:\Users\Shivani T\OneDrive\Desktop\AI- assisted>

```

**Explanation:** The queue implementation manages requests in a first-in-first-out manner, while the priority queue implementation ensures that faculty requests are processed before student requests, regardless of their order of arrival.

**Prompt: #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  
#ast-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

**Code:** class HelpDeskStack:

```
def __init__(self):
```

```
    self.stack = []
```

```
def push(self, ticket):
```

```
    self.stack.append(ticket)
```

```
def pop(self):
```

```
    if not self.is_empty():
```

```
        return self.stack.pop()
```

```
    return None
```

```
def peek(self):
```



```
if not self.is_empty():
```

```
    return self.stack[-1]
```

```
return None
```

```
def is_empty(self):
```

```
    return len(self.stack) == 0
```

```
# Example usage
```

```
help_desk_stack = HelpDeskStack()
```

```
help_desk_stack.push("Ticket 1: Computer not turning on")
```

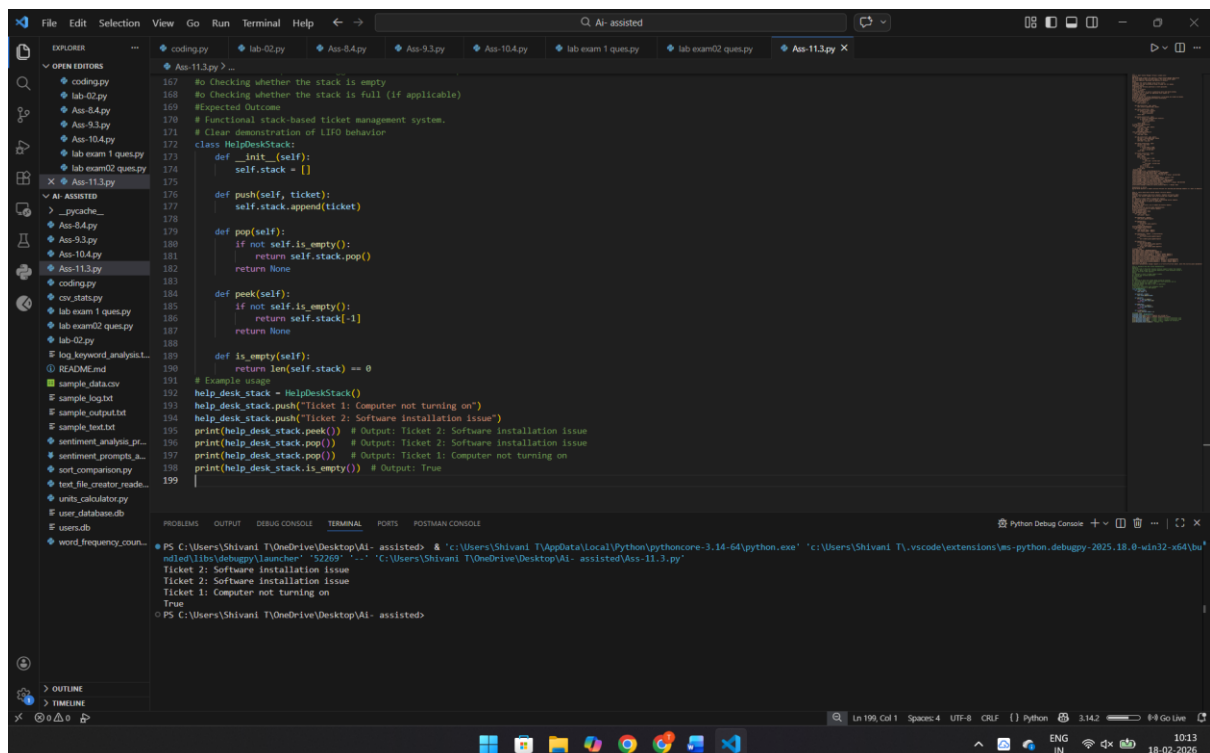
```
help_desk_stack.push("Ticket 2: Software installation issue")
```

```
print(help_desk_stack.peek())
```

```
print(help_desk_stack.pop())
```

```
print(help_desk_stack.pop())
```

```
print(help_desk_stack.is_empty())
```



```
File Edit Selection View Go Run Terminal Help
Ass-11.3.py
167 # Checking whether the stack is empty
168 # Checking whether the stack is full (if applicable)
169 # Expected Outcome
170 # Functional stack-based ticket management system.
171 # Clear demonstration of LIFO behavior
172 class HelpDeskStack:
173     def __init__(self):
174         self.stack = []
175
176     def push(self, ticket):
177         self.stack.append(ticket)
178
179     def pop(self):
180         if not self.is_empty():
181             return self.stack.pop()
182         return None
183
184     def peek(self):
185         if not self.is_empty():
186             return self.stack[-1]
187         return None
188
189     def is_empty(self):
190         return len(self.stack) == 0
191
192 # Example usage
193 help_desk_stack = HelpDeskStack()
194 help_desk_stack.push("Ticket 1: Computer not turning on")
195 help_desk_stack.push("Ticket 2: Software installation issue")
196 print(help_desk_stack.peek()) # Output: Ticket 2: Software installation issue
197 print(help_desk_stack.pop()) # Output: Ticket 2: Software installation issue
198 print(help_desk_stack.pop()) # Output: Ticket 1: Computer not turning on
199 print(help_desk_stack.is_empty()) # Output: True
200
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
Python Debug Console
PS C:\Users\Shivani T\OneDrive\Desktop\AI- assisted> & 'c:\Users\Shivani T\AppData\Local\Python\pythoncore-3.14-64\python.exe' 'c:\Users\Shivani T\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bin\debugpy_launcher.py' --52269 -- "C:\Users\Shivani T\OneDrive\Desktop\AI- assisted\Ass-11.3.py"
Ticket 2: Software installation issue
Ticket 2: Software installation issue
Ticket 1: Computer not turning on
True
PS C:\Users\Shivani T\OneDrive\Desktop\AI- assisted>
```

**Explanation:** The stack implementation allows for managing support tickets in a last-in-first-out manner, ensuring that the most recently raised ticket is addressed first. The additional operations provide functionality to check the top ticket and whether the stack is empty.

**Prompt:** To implement a Hash Table and understand collision handling.

to 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)]
```

```
def _hash(self, key):
```

```
    return hash(key) % self.size
```

```
def insert(self, key, value):
```

```
    index = self._hash(key)
```

```
    for i, (k, v) in enumerate(self.table[index]):
```

```
        if k == key:
```

```
            self.table[index][i] = (key, value) # Update existing key
```

```
            return
```

```
    self.table[index].append((key, value)) # Insert new key-value pair
```

```
def search(self, key):
```

```
    index = self._hash(key)
```

```

for k, v in self.table[index]:
    if k == key:
        return v
return None # 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 True
    return False # Key not found

```

# Example usage

```

hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
print(hash_table.search("name"))
print(hash_table.search("age"))
hash_table.delete("name")
print(hash_table.search("name"))

```

The screenshot shows a Python IDE with a file explorer on the left containing files like `text_file_creator_read...`, `units_calculator.py`, `user_database.db`, `users.db`, and `word_frequency_coun...`. The main editor area displays the Python code for the hash table implementation. The terminal window at the bottom shows the execution of the example usage code, with the following output:

```

PS C:\Users\Shivani T\OneDrive\Desktop\Ai- assisted> & 'c:\Users\Shivani T\AppData\Local\Python\pythoncore-3.14-6
8.0-win32-x64\bundled\libs\debugpy\launcher' '58426' '--' 'C:\Users\Shivani T\OneDrive\Desktop\Ai- assisted\Ass-11
Alice
30
None
PS C:\Users\Shivani T\OneDrive\Desktop\Ai- assisted>

```

**Explanation:** The hash table implementation uses chaining to handle collisions, allowing multiple key-value pairs to be stored at the same index. The insert method adds or update

key-value pairs, the search method retrieves values based on keys, and the delete method removes key-value pairs from the table.

**Prompt:** 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

Choose the most appropriate data structure for each feature.

Justify your choice in 2–3 sentences.

Implement one selected feature

**Code:** class AttendanceTracker:

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

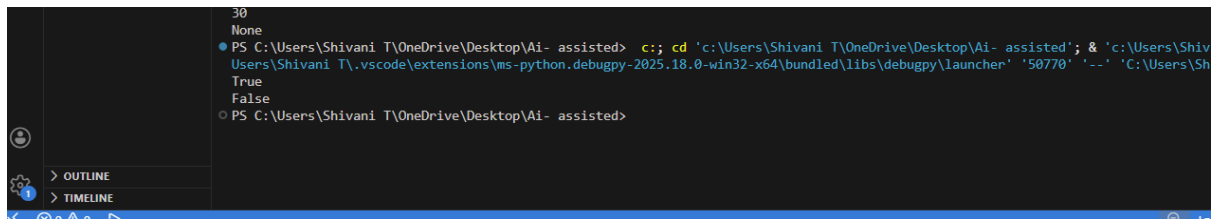
def mark_attendance(self, student_id):
    self.attendance[student_id] = True

def check_attendance(self, student_id):
    return self.attendance.get(student_id, False)
```

# Example usage

```
tracker = AttendanceTracker()
tracker.mark_attendance("S12345")
print(tracker.check_attendance("S12345"))
```

```
print(tracker.check_attendance("S54321"))
```



```
30
None
• PS C:\Users\Shivani T\OneDrive\Desktop\Ai- assisted> c:: cd 'c:\Users\Shivani T\OneDrive\Desktop\Ai- assisted'; & 'c:\Users\Shiv
Users\Shivani T\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '50770' '--' 'C:\Users\Sh
True
False
○ PS C:\Users\Shivani T\OneDrive\Desktop\Ai- assisted>
```

**Explanation:** The Attendance Tracker uses a hash table (dictionary) to store student IDs and their attendance status. This allows for efficient insertion and lookup operations, making it easy to mark and check attendance for a large number of students.