| SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE | | DEPARTMENT OF COMPUTER SCIENCE ENGINEERING | |
|---|---|---|---|
| **Program Name:** B. Tech | | **Assignment Type: Lab** | **Academic Year:**2025-2026 |
| **Course Coordinator Name** | Venkataramana Veeramsetty | | |
| **Instructor(s) Name** | Dr. V. Venkataramana (Co-ordinator)<br>Dr. T. Sampath Kumar<br>Dr. Pramoda Patro<br>Dr. Brij Kishor Tiwari<br>Dr.J.Ravichander<br>Dr. Mohammand Ali Shaik<br>Dr. Anirodh Kumar<br>Mr. S.Naresh Kumar<br>Dr. RAJESH VELPULA<br>Mr. Kundhan Kumar<br>Ms. Ch.Rajitha<br>Mr. M Prakash<br>Mr. B.Raju<br>Intern 1 (Dharma teja)<br>Intern 2 (Sai Prasad)<br>Intern 3 (Sowmya)<br>NS_2 ( Mounika) | | |
| **Course Code** | 24CS002PC215 | **Course Title** | AI Assisted Coding |
| **Year/Sem** | II/I | **Regulation** | R24 |
| **Date and Day of Assignment** | Week6 - Monday | **Time(s)** | |
| **Duration** | 2 Hours | **Applicable to Batches** | |
| **AssignmentNumber:11.1**(Present assignment number)/**24**(Total number of assignments) | | | |

| | **Question** | ***Expected Time to complete*** |
|---|---|---|
| 1 | **Lab 11 – Data Structures with AI: Implementing Fundamental Structures**<br>**Lab Objectives**<br>• Use AI to assist in designing and implementing fundamental data structures in Python.<br>• Learn how to prompt AI for structure creation, optimization, and documentation.<br>• Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, | Week 6 - Monday |

Graphs, and Hash Tables.

- Enhance code quality with AI-generated comments and performance suggestions.

**Task Description #1 – Stack Implementation**

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

Sample Input Code:

```
class Stack:
    pass
```

Expected Output:

- A functional stack implementation with all required methods and docstrings.

## PROMPT :

Generate a Python Stack class starting from Implement the methods push(item), pop(), peek(), and is_empty(). Each method must include clear docstrings explaining its purpose. The output should be a fully functional stack implementation with example usage demonstrating all methods.

## CODE:

```python
# Create an instance of the Stack class
my_stack = Stack()

# Demonstrate push
print("Pushing elements onto the stack:")
my_stack.push(10)
print(f"Stack after pushing 10: {my_stack.items}")
my_stack.push(20)
print(f"Stack after pushing 20: {my_stack.items}")
my_stack.push(30)
print(f"Stack after pushing 30: {my_stack.items}")

# Demonstrate peek
print("\nPeeking at the top element:")
print(f"Top element: {my_stack.peek()}")
print(f"Stack after peeking: {my_stack.items}") # Stack remains unchanged

# Demonstrate is_empty
print("\nChecking if the stack is empty:")
print(f"Is stack empty? {my_stack.is_empty()}")

# Demonstrate pop
print("\nPopping elements from the stack:")
popped_item = my_stack.pop()
print(f"Popped item: {popped_item}")
print(f"Stack after popping: {my_stack.items}")

popped_item = my_stack.pop()
print(f"Popped item: {popped_item}")
print(f"Stack after popping: {my_stack.items}")

popped_item = my_stack.pop()
print(f"Popped item: {popped_item}")
print(f"Stack after popping: {my_stack.items}")

# Demonstrate is_empty after popping all elements
print("\nChecking if the stack is empty after popping all elements:")
print(f"Is stack empty? {my_stack.is_empty()}")

# Demonstrate popping from an empty stack
print("\nAttempting to pop from an empty stack:")
popped_item = my_stack.pop()
print(f"Popped item: {popped_item}")
```

# OUTPUT:

```
Pushing elements onto the stack:
Stack after pushing 10: [10]
Stack after pushing 20: [10, 20]
Stack after pushing 30: [10, 20, 30]

Peeking at the top element:
Top element: 30
Stack after peeking: [10, 20, 30]

Checking if the stack is empty:
Is stack empty? False

Popping elements from the stack:
Popped item: 30
Stack after popping: [10, 20]
Popped item: 20
Stack after popping: [10]
Popped item: 10
Stack after popping: []

Checking if the stack is empty after popping all elements:
Is stack empty? True

Attempting to pop from an empty stack:
Popped item: Stack is empty
```

## OBSERVATION:

The output shows that the `Stack` class correctly implements the `push`, `pop`, `peek`, and `is_empty` methods, demonstrating the Last-In, First-Out (LIFO) behavior and handling the case of popping from an empty stack.

---

**Task Description #2 – Queue Implementation**

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

```
class Queue:
    pass
```

Expected Output:

- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

## PROMPT:

Implement a Queue using Python lists , the code output must contain FIFO-based queue class with enqueue, dequeue, peek, and size methods.

## CODE:

```python
# Create an instance of the Queue class
my_queue = Queue()

# Demonstrate enqueue
print("Enqueuing elements into the queue:")
my_queue.enqueue(100)
print(f"Queue after enqueuing 100: {my_queue.items}")
my_queue.enqueue(200)
print(f"Queue after enqueuing 200: {my_queue.items}")
my_queue.enqueue(300)
print(f"Queue after enqueuing 300: {my_queue.items}")

# Demonstrate peek
print("\nPeeking at the front element:")
print(f"Front element: {my_queue.peek()}")
print(f"Queue after peeking: {my_queue.items}") # Queue remains unchanged

# Demonstrate size
print("\nChecking the size of the queue:")
print(f"Queue size: {my_queue.size()}")

# Demonstrate dequeue
print("\nDequeuing elements from the queue:")
dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeuing: {my_queue.items}")

dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeuing: {my_queue.items}")

# Demonstrate is_empty
print("\nChecking if the queue is empty:")
print(f"Is queue empty? {my_queue.is_empty()}")

dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeuing: {my_queue.items}")

# Demonstrate is_empty after dequeuing all elements
print("\nChecking if the queue is empty after dequeuing all elements:")
print(f"Is queue empty? {my_queue.is_empty()}")

# Demonstrate dequeuing from an empty queue
print("\nAttempting to dequeue from an empty queue:")
dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
```

# OUTPUT:

```
Enqueuing elements into the queue:
Queue after enqueuing 100: [100]
Queue after enqueuing 200: [100, 200]
Queue after enqueuing 300: [100, 200, 300]

Peeking at the front element:
Front element: 100
Queue after peeking: [100, 200, 300]

Checking the size of the queue:
Queue size: 3

Dequeuing elements from the queue:
Dequeued item: 100
Queue after dequeuing: [200, 300]
Dequeued item: 200
Queue after dequeuing: [300]

Checking if the queue is empty:
Is queue empty? False
Dequeued item: 300
Queue after dequeuing: []

Checking if the queue is empty after dequeuing all elements:
Is queue empty? True

Attempting to dequeue from an empty queue:
Dequeued item: Queue is empty
```

## OBSERVATION:

The output shows that the Queue class correctly implements the FIFO principle: elements are added to the end (enqueue) and removed from the beginning (dequeue). It also correctly handles peeking at the front element and indicates when the queue is empty.

---

**Task Description #3 – Linked List**

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

```
class Node:
    pass

class LinkedList:
    pass
```

Expected Output:

- A working linked list implementation with clear method documentation.

## PROMPT:

Generate a Singly Linked List with insert and display methods. Sample Input Code: class Node: pass class LinkedList: pass. The output should contain a working linked list implementation with clear method documentation.

## CODE:

```
# Example Usage:
# Create an instance of the LinkedList
my_list = LinkedList()

# Insert elements
print("Inserting elements:")
my_list.insert(5)
my_list.insert(10)
my_list.insert(15)

# Display the linked list
print("\nLinked list after insertion:")
my_list.display()

# Insert another element
print("\nInserting another element:")
my_list.insert(20)

# Display the linked list again
print("\nLinked list after another insertion:")
my_list.display()
```

```
Inserting elements:

Linked list after insertion:
15 -> 10 -> 5

Inserting another element:

Linked list after another insertion:
20 -> 15 -> 10 -> 5
```

## OBSERVATION:

The code defines a basic Singly Linked List with methods to `insert` new nodes at the beginning and `display` the list's elements. The output shows that elements are inserted at the head, resulting in the list being displayed in reverse order of insertion, which is the expected behavior for this type of insertion.

---

**Task Description #4 – Binary Search Tree (BST)**

Task: Use AI to create a BST with insert and in-order traversal

methods.

Sample Input Code:

```
class BST:
    Pass
```

Expected Output:

- BST implementation with recursive insert and traversal methods.

## PROMPT:

Create a BST with insert and in-order traversal methods. The code must have the implementation of BST with recursive insert and traversal methods.

## CODE:

```
# Example Usage:
# Create an instance of the BST
my_bst = BST()

# Insert elements
print("Inserting elements into the BST:")
elements_to_insert = [50, 30, 70, 20, 40, 60, 80]
for element in elements_to_insert:
    my_bst.insert(element)
    print(f"Inserted {element}")

# Perform in-order traversal
print("\nIn-order traversal of the BST:")
traversed_elements = my_bst.in_order_traversal()
print(traversed_elements)
```

```
Inserting elements into the BST:
Inserted 50
Inserted 30
Inserted 70
Inserted 20
Inserted 40
Inserted 60
Inserted 80

In-order traversal of the BST:
[20, 30, 40, 50, 60, 70, 80]
```

## OBSERVATION:

The code implements a Binary Search Tree with recursive insertion and in-order traversal. The output demonstrates that inserting elements maintains the BST property (smaller keys on the left, larger on the right), and the in-order traversal correctly visits the nodes in ascending order of their keys.

---

**Task Description #5 – Hash Table**

Task: Use AI to implement a hash table with basic insert, search, and delete methods.

Sample Input Code:

```
class HashTable:
    pass
```

Expected Output:

- Collision handling using chaining, with well-commented methods.

## PROMPT:

Implement a hash table with basic insert, search, and delete methods. The expected output should contain the Collision handling using chaining, with well-commented methods.

## CODE:

```python
class HashTable:
    """
    A simple hash table implementation with chaining for collision handling.
    """
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(self.size)] # Initialize the table with empty lists for chaining

    def _hash(self, key):
        """
        Simple hash function using modulo division.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Inserts a key-value pair into the hash table.
        Handles collisions using chaining.
        """
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                # Update value if key already exists
                self.table[index][i] = (key, value)
                return
        # Add new key-value pair if key does not exist
        self.table[index].append((key, value))

    def search(self, key):
        """
        Searches for a key in the hash table.
        Returns the value if found, otherwise returns None.
        """
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return value if key is found
        return None  # Return None if key is not found

    def delete(self, key):
        """
        Deletes a key-value pair from the hash table.
        """
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i] # Delete the key-value pair
                return
        print(f"Key '{key}' not found.") # Print message if key is not found

    def display(self):
        """
        Displays the contents of the hash table.
        """
        for index in range(self.size):
            print(f"Bucket {index}: {self.table[index]}")
```

```python
# Create a hash table with size 10
ht = HashTable(10)

# Insert some key-value pairs
ht.insert("apple", 1)
ht.insert("banana", 2)
ht.insert("cherry", 3)
ht.insert("date", 4)
ht.insert("banana", 20) # Update value for existing key

# Display the hash table
print("Hash Table after insertions:")
ht.display()

# Search for keys
print("\nSearching for 'apple':", ht.search("apple"))
print("Searching for 'grape':", ht.search("grape"))

# Delete a key
print("\nDeleting 'banana'")
ht.delete("banana")

# Display the hash table after deletion
print("\nHash Table after deletion:")
ht.display()

# Try deleting a non-existent key
print("\nDeleting 'grape'")
ht.delete("grape")
```

```
Hash Table after insertions:
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: [('apple', 1)]
Bucket 4: []
Bucket 5: [('banana', 20)]
Bucket 6: [('date', 4)]
Bucket 7: []
Bucket 8: []
Bucket 9: [('cherry', 3)]

Searching for 'apple': 1
Searching for 'grape': None

Deleting 'banana'

Hash Table after deletion:
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: [('apple', 1)]
Bucket 4: []
Bucket 5: []
Bucket 6: [('date', 4)]
Bucket 7: []
Bucket 8: []
Bucket 9: [('cherry', 3)]

Deleting 'grape'
Key 'grape' not found.
```

# OBSERVATION:

the code successfully demonstrates the core functionalities of the hash table: inserting/updating key-value pairs, searching for keys, deleting keys, and handling non-existent keys, all while using chaining for collision resolution as intended.

---

**Task Description #6 – Graph Representation**

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

```
class Graph:
    pass
```

Expected Output:

- Graph with methods to add vertices, add edges, and display connections.

# PROMPT:

Implement a graph using an adjacency list. Sample Input Code: class Graph: pass . also the output should contain Graph with methods to add vertices, add edges, and display connections

# CODE:

```
class Graph:
    def __init__(self):
        self.adj_list = {}

    def add_vertex(self, vertex):
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.adj_list and vertex2 in self.adj_list:
            self.adj_list[vertex1].append(vertex2)
            # For an undirected graph, add the reverse edge as well
            self.adj_list[vertex2].append(vertex1)
        else:
            print(f"One or both vertices ({vertex1}, {vertex2}) not found.")

    def display_connections(self):
        for vertex in self.adj_list:
            print(f"{vertex}: {self.adj_list[vertex]}")

# Example Usage:
graph = Graph()

graph.add_vertex('A')
graph.add_vertex('B')
graph.add_vertex('C')
graph.add_vertex('D')

graph.add_edge('A', 'B')
graph.add_edge('A', 'C')
graph.add_edge('B', 'D')
graph.add_edge('C', 'D')

graph.display_connections()
```

```
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'D']
D: ['B', 'C']
```

# OBSERVATION:

This code implements a graph data structure using an adjacency list. It allows you to add vertices, add edges between them, and display the connections of each vertex. The example usage demonstrates how to create a graph, add vertices and edges, and then print the adjacency list representation.

**Task Description #7 – Priority Queue**

Task: Use AI to implement a priority queue using Python's heapq module.

Sample Input Code:

```
class PriorityQueue:
    pass
```

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

## PROMPT:

Implement a priority queue using Python's heapq module. Sample Input Code: class PriorityQueue: pass . The expected output should be of Implementation with enqueue (priority), dequeue (highest priority), and display methods.

## CODE:

```python
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0 # To ensure stable sorting for items with the same priority

    def enqueue(self, item, priority):
        # Items are tuples of (priority, index, item)
        heapq.heappush(self._queue, (-priority, self._index, item)) # Negative priority for max-heap
        self._index += 1

    def dequeue(self):
        if not self._queue:
            return None
        return heapq.heappop(self._queue)[-1] # Return the item

    def display(self):
        print("Priority Queue (highest priority first):")
        # Create a temporary list to display in order without modifying the original heap
        temp_list = sorted(self._queue, reverse=True)
        for priority, index, item in temp_list:
            print(f"  Priority: {-priority}, Item: {item}")

# Example Usage:
pq = PriorityQueue()

pq.enqueue("Task A", 3)
pq.enqueue("Task B", 1)
pq.enqueue("Task C", 2)
pq.enqueue("Task D", 3)

pq.display()

print("\nDequeueing highest priority item:")
dequeued_item = pq.dequeue()
print(f"Dequeued: {dequeued_item}")

pq.display()

print("\nDequeueing next highest priority item:")
dequeued_item = pq.dequeue()
print(f"Dequeued: {dequeued_item}")

pq.display()
```

# OUTPUT:

```
Priority Queue (highest priority first):
  Priority: 1, Item: Task B
  Priority: 2, Item: Task C
  Priority: 3, Item: Task D
  Priority: 3, Item: Task A

Dequeueing highest priority item:
Dequeued: Task A
Priority Queue (highest priority first):
  Priority: 1, Item: Task B
  Priority: 2, Item: Task C
  Priority: 3, Item: Task D

Dequeueing next highest priority item:
Dequeued: Task D
Priority Queue (highest priority first):
  Priority: 1, Item: Task B
  Priority: 2, Item: Task C
```

## OBSERVATION:

This code implements a priority queue using Python's heapq module. It allows you to add items with a given priority (enqueue), remove the item with the highest priority (dequeue), and display the items in the queue based on their priority. The example demonstrates adding tasks with different priorities and then dequeueing them in order of highest priority.

---

**Task Description #8 – Deque**

Task: Use AI to implement a double-ended queue using collections.deque.

Sample Input Code:

```
class DequeDS:
    pass
```

Expected Output:

- Insert and remove from both ends with docstrings.

## PROMPT:

Implement a double-ended queue using collections.deque. Sample Input Code: class DequeDS: pass . The expected output should Insert and remove from both ends with docstrings.

## CODE:

```python
from collections import deque

class DequeDS:
    def __init__(self):
        """Initializes an empty deque."""
        self._deque = deque()

    def append_right(self, item):
        """Appends an item to the right end of the deque."""
        self._deque.append(item)

    def append_left(self, item):
        """Appends an item to the left end of the deque."""
        self._deque.appendleft(item)

    def pop_right(self):
        """Removes and returns an item from the right end of the deque."""
        if not self._deque:
            return None  # Or raise an error, depending on desired behavior
        return self._deque.pop()

    def pop_left(self):
        """Removes and returns an item from the left end of the deque."""
        if not self._deque:
            return None  # Or raise an error, depending on desired behavior
        return self._deque.popleft()

    def display(self):
        """Displays the current contents of the deque."""
        print("Deque contents:", list(self._deque))

# Example Usage:
dq = DequeDS()

dq.append_right(1)
dq.append_left(2)
dq.append_right(3)

dq.display()

print("\nPopping from the right:", dq.pop_right())
dq.display()

print("Popping from the left:", dq.pop_left())
dq.display()
```

# OUTPUT:

```
Deque contents: [2, 1, 3]

Popping from the right: 3
Deque contents: [2, 1]
Popping from the left: 2
Deque contents: [1]
```

# OBSERVATION:

This code defines a class DequeDS that uses Python's built-in `collections.deque` to create a double-ended queue. It provides methods to add elements to the left or right (`append_left`, `append_right`) and remove elements from the left or right (`pop_left`, `pop_right`). The example demonstrates these operations.

**Task Description #9 – AI-Generated Data Structure Comparisons**

Task: Use AI to generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.

Sample Input Code:

# No code, prompt AI for a data structure comparison table

Expected Output:

- A markdown table with structure names, operations, and complexities.

## PROMPT:

Generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities. Sample Input Code: # No code, prompt AI for a data structure comparison table . expected output is : A markdown table with structure names, operations, and complexities.

## OUTPUT:

| Data Structure | Operation | Average Time Complexity | Worst-Case Time Complexity | Notes |
|---|---|---|---|---|
| **Array** | Access (by index) | O(1) | O(1) | |
| | Search (linear) | O(n) | O(n) | |
| | Search (binary) | O(log n) | O(log n) | Requires sorted array |
| | Insertion | O(n) | O(n) | Depends on position, shifting needed |
| | Deletion | O(n) | O(n) | Depends on position, shifting needed |
| **Linked List** | Access (by index) | O(n) | O(n) | Must traverse from head/tail |
| | Search | O(n) | O(n) | Must traverse from head |
| | Insertion (head) | O(1) | O(1) | |
| | Insertion (tail) | O(1) | O(1) | With tail pointer |
| | Insertion (middle) | O(n) | O(n) | Requires traversal |
| | Deletion (head) | O(1) | O(1) | |
| | Deletion (tail) | O(n) | O(n) | O(1) with doubly linked list and tail ptr |
| | Deletion (middle) | O(n) | O(n) | Requires traversal |
| **Stack** | Push | O(1) | O(1) | Usually implemented with array or list |
| | Pop | O(1) | O(1) | |
| | Peek/Top | O(1) | O(1) | |
| **Queue** | Enqueue | O(1) | O(1) | Usually implemented with array or list |
| | Dequeue | O(1) | O(1) | |
| | Peek/Front | O(1) | O(1) | |
| **Hash Table** | Search | O(1) | O(n) | Worst case with collisions |
| | Insertion | O(1) | O(n) | Worst case with collisions |
| | Deletion | O(1) | O(n) | Worst case with collisions |
| **Tree (Binary Search Tree)** | Search | O(log n) | O(n) | Worst case for skewed tree |
| | Insertion | O(log n) | O(n) | Worst case for skewed tree |
| | Deletion | O(log n) | O(n) | Worst case for skewed tree |
| **Graph** | Adding Vertex | O(1) | O(1) | Adjacency List |
| | Adding Edge | O(1) | O(1) | Adjacency List |
| | Checking Edge | O(deg(v)) | O(V) | Adjacency List |
| | Adding Vertex | O(V^2) | O(V^2) | Adjacency Matrix (initialization) |
| | Adding Edge | O(1) | O(1) | Adjacency Matrix |
| | Checking Edge | O(1) | O(1) | Adjacency Matrix |

## OBSERVATION:

This table provides a quick reference for the typical time

complexities of common operations (like access, search, insertion, and deletion) across various fundamental data structures, highlighting the performance differences between them in average and worst-case scenarios.

---

**Task Description #10 Real-Time Application Challenge – Choose the Right Data Structure**

**Scenario:**

Your college wants to develop a Campus Resource Management System that handles:

1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due dates.
4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
  - Stack
  - Queue
  - Priority Queue
  - Linked List
  - Binary Search Tree (BST)
  - Graph
  - Hash Table
  - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

✅ Deliverables (For All Tasks)

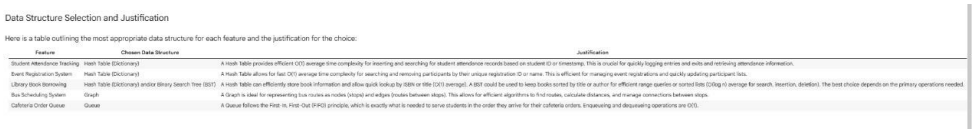
1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

Top of Form

## PROMPT:

Your college wants to develop a Campus Resource Management System that handles: Student Attendance Tracking – Daily log of students entering/exiting the campus. Event Registration System – Manage participants in events with quick search and removal. Library Book Borrowing – Keep track of available books and their due dates. Bus Scheduling System – Maintain bus routes and stop connections. Cafeteria Order Queue – Serve students in the order they arrive. Student Task: For each feature, select the most appropriate data structure from the list below: Stack Queue Priority Queue Linked List Binary Search Tree (BST) Graph Hash Table Deque Justify your choice in 2–3 sentences per feature. Implement one selected feature as a working Python program.

## OUTPUT:



Data Structure Selection and Justification

Here is a table outlining the most appropriate data structure for each feature and the justification for the choice:

| Feature | Chosen Data Structure | Justification |
|---|---|---|
| Student Attendance Tracking | Hash Table (Dictionary) | A Hash Table provides efficient O(1) average time complexity for inserting and searching for student attendance records based on student ID or timestamp. This is crucial for quickly logging entries and exits and retrieving attendance information. |
| Event Registration System | Hash Table (Dictionary) | A Hash Table allows for fast O(1) average time complexity for searching and removing participants by their unique registration ID or name. This is efficient for managing event registrations and quickly updating participant lists. |
| Library Book Borrowing | Hash Table (Dictionary) and/or Binary Search Tree (BST) | A Hash Table can efficiently store book information and allow quick lookup by ISBN or title (O(1) average). A BST could be used to keep books sorted by title or author for efficient range queries or sorted lists (O(log n) average for search, insertion, deletion). The best choice depends on the primary operations needed. |
| Bus Scheduling System | Graph | A Graph is ideal for representing bus routes as nodes (stops) and edges (routes between stops). This allows for efficient algorithms to find routes, calculate distances, and manage connections between stops. |
| Cafeteria Order Queue | Queue | A Queue follows the First-In, First-Out (FIFO) principle, which is exactly what is needed to serve students in the order they arrive for their cafeteria orders. Enqueueing and dequeueing operations are O(1). |

## OBSERVATION:

| | | |
|---|---|---|
| | This markdown table proposes appropriate data structures for different campus resource management features, such as using a Hash Table for attendance tracking and event registration, a Graph for bus scheduling, and a Queue for cafeteria orders, along with brief justifications based on the operational requirements of each feature. | |