|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **NAME:H.V.S.VYSHNAVI** | | | | | **BATCH NO:06** | | | | |
| **H.NO:2403A51182** | | | |  | | |  | | |
|  | | | |  | | | | | |
|  | | | |  | | | | | |
| **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) | | | | | | | | | |
|  | **Q.No.** | **Question** | | | | | | ***Expected Time***  ***to complete*** |  |
|  | 1 | **Lab 11 – Data Structures with AI: Implementing Fundamental Structures**  **Lab Objectives**   * Use AI to assist in designing and implementing fundamental data structures in Python. * Learn how to prompt AI for structure creation, optimization, and documentation. * Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, 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  **PROMPT:**  generate a Stack class with push, pop, peek, and is\_empty methods.  Sample Input Code:  class Stack:  pass**CODE:**      **Output:**    **Observation:**  The class correctly models a **stack** with essential operations (push, pop, peek, is\_empty). It uses Python lists efficiently, making all major operations (push, pop, peek) run in **O(1) time**. However, returning strings like "Stack is empty" instead of raising exceptions may not always be the best design for larger applications.  **Task Description #2 – Queue Implementation**  **PROMPT:**  implement a Queue using Python lists.  Sample Input Code:  class Queue:  pass  **Code:**      **Output:**    **Observation**:  .enqueue adds items at the **end** of the list.  **.** dequeue removes items from the **front** of the list, making it a **FIFO** (First In, First Out) structure.  .peek shows the front item without removing it.  **.** is\_empty checks if the queue has elements.  . Using pop(0) makes dequeue **O(n)** (slower for very large queues). For efficiency, Python’s collections.deque is usually preferred.  **Task Description #3 – Linked List**  **PROMPT:**  generate a Singly Linked List with insert and display methods.  Sample Input Code:  class Node:  pass  **Code:**    **Output:**    **Observation:**   * Node stores **data** and a **reference (next)** to the next node. * insert always adds at the **end** of the list. * display traverses from the **head** until the end (None). * The structure follows **sequential linking** instead of using Python’s built-in list.   **Task Description #4 – Binary Search Tree (BST)**  Task: Use AI to create a BST with insert and in-order traversal methods.  **PROMPT:**  to create a BST with insert and in-order traversal methods.  Sample Input Code:  class BST:  pass.  **Code**:        **Output:**    **OBSERVATION:**   **insert** maintains BST property:  **.**Smaller values go **left**, larger values go **right**.   **in\_order\_traversal** visits nodes in **sorted order (ascending)**.   Duplicate values are ignored in this simple implementation.  **Task Description #5 – Hash Table**  Task: Use AI to implement a hash table with basic insert, search, and delete methods.  **PROMPT:**  implement a hash table with basic insert, search, and delete methods.  Sample Input Code:  class HashTable:  pass**Code:**      **Output:**   **Observation:**  * Uses Python’s built-in hash() function for generating indexes. * **Collision Handling** → implemented via **chaining** (list of lists). * insert: O(1) average, O(n) worst (if many collisions). * search & delete: also O(1) average case.   **Task Description #6 – Graph Representation**  Task: Use AI to implement a graph using an adjacency list.  Sample Input Code:    class Graph:  pass  **Output:**    **Task Description #7 – Priority Queue**  Task:  **PROMPT:**  to implement a priority queue using Python’s heapq module.  Sample Input Code:  class PriorityQueue:  pass  **Code:**      **Output:**    **OBSERVATION:**   **Heap Property**   * Python’s heapq implements a **min-heap**, meaning the smallest value has the highest priority. * That’s why in this implementation, the **lowest priority number** comes out first.    **Tie-Breaking with \_index**   * If two elements have the same priority, Python would otherwise try to compare the items themselves (which may fail for different data types). * To avoid this, we use an \_index counter so that items are ordered by **insertion order** when priorities match.    **Time Complexity**   * push() → O(log n) because it adds a new element into the heap. * pop() → O(log n) because it removes the smallest element from the heap. * peek() → O(1) because it just returns the first element without removing it.    **Space Complexity**   * O(n) for storing all elements in the queue   **Task Description #8 – Deque**  Task:  **PROMPT:**  to implement a double-ended queue using collections.deque.  Sample Input Code:  class DequeDS:  pass**Code:**        **Output:**    **OBSERVATION:**   1. collections.deque is optimized for **fast appends and pops** from both ends (O(1) time complexity). 2. Unlike Python lists, deques don’t suffer performance issues when inserting/removing from the front. 3. Supports **stack and queue behavior**:    * Queue → append() + popleft()    * Stack → append() + pop() 4. Useful for **sliding window problems**, **palindrome checking**, and **task scheduling**.   **Task Description #9 – AI-Generated Data Structure Comparisons**  Task:  **PROMPT:**  generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.  Sample Input Code:  # No code, for a data structure comparison table  **Task Description #10 Real-Time Application Challenge – Choose the Right Data Structure**  **PROMPT:**  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:  o Stack  o Queue  o Priority Queue  o Linked List  o Binary Search Tree (BST)  o Graph  o Hash Table  o Deque  **CODE:**        **Output:**    ✅ 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 | | | | | | Week6 - Monday |  |