|  |  |  |
| --- | --- | --- |
|  | **Lab 11 – Data Structures with AI: Implementing Fundamental Structures**  **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 Stack class with push, pop, peek, and is\_empty methods.  **CODE:**      **OUTPUT:**  **OBSERVATION:**  Uses a list to store items privately (self.\_items)Supports standard stack operations: push, pop, peek, and is\_empty Checks for empty stack before popping or peekingReturns None if operations are attempted on an empty stack avoids crashesPushes three values: 10, 20, 30Shows top element before and after poppingGracefully handles popping from an empty  **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.  **CODE:**      **OUTPUT:**  **OBSERVATION:**  Uses a list to store items privately (self.\_items)Supports standard stack operations: push, pop, peek, and is\_empty Checks for empty stack before popping or peekingReturns None if operations are attempted on an empty stack avoids crashesPushes three values: 10, 20, 30Shows top element before and after poppingGracefully handles popping from an 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:**  to generate a Singly Linked List with insert and display methods.  **CODE:**  **OUTPUT:**  **OBSERVATION:**  This Python code defines a basic singly linked list using a LinkedList class with an insert method that adds new nodes to the front of the list, and a display method to print the list elements. It initializes with an empty head and builds the list in reverse order of insertion, demonstrating how linked lists store data sequentially through node references. The sample usage shows the list being populated with values 30, 20, and 10, resulting in an output of 10 -> 20 -> 30, confirming correct insertion and traversal logic. It's a clean and effective introduction to linked list fundamentals.  **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.  **CODE:**      **OUTPUT:**  **OBSERVATION:**  **This Python code defines a Binary Search Tree (BST) with methods for inserting nodes and performing in-order traversal. It uses recursion to maintain BST properties, placing smaller values to the left and larger ones to the right. The in-order traversal prints elements in sorted order, confirming correct structure. Sample usage inserts multiple values and demonstrates traversal, making it a solid example of tree-based data organization and recursive logic.**  **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:**  **i**mplement a hash table with basic insert, search, and delete methods. **CODE:**  **OUTPUT:**  **OBSERVATION:**  **This Python code implements a basic hash table using a list of buckets and a simple modulo-based hash function. It supports key operations like insertion, search, update, deletion, and display, with collision handling via chaining (lists at each index). The sample usage shows how duplicate keys are updated, how missing keys are handled gracefully, and how the table's contents can be printed clearly. Overall, it's a solid and practical example of hash table mechanics, ideal for learning or lightweight applications.**  **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. **CODE:**  **OUTPUT:**  **OBSERVATION:**  **This code builds an undirected graph using an adjacency list. It supports adding vertices and edges, and displays connections clearly. Sample usage shows a small network with nodes A–D and their links, making it a simple and effective intro to graph structures.**  **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. **CODE:**  **OUTPUT:**  **OBSERVATION:**  **This Python code implements a priority queue using the heapq module, storing items as (priority, value) tuples to maintain order efficiently. It includes methods to push, pop, peek, and check if the queue is empty, with sample usage showing tasks being added and retrieved based on priority. The queue handles empty states gracefully and demonstrates correct behavior, making it a clear and practical example of heap-based priority management.**  **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 Python class DequeDS using collections.deque to support inserting and removing elements from both front and rear. Add clear docstrings for the class and methods, and include a short demo showing how to use the deque for different operations. **CODE:**    **OUTPUT:**  **OBSERVATION:**  **This code wraps Python’s deque to create a double-ended queue with methods to append, pop, peek, and display from both ends. It handles empty checks gracefully and shows how elements can be added or removed from either side. The sample usage confirms correct behavior and makes it a clear, flexible structure for queue-like tasks.**  **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 markdown table comparing common data structures (Stack, Queue, Linked List, Binary Search Tree, Hash Table, Graph, Deque, Priority Queue). Include their basic operations (insertion, deletion, search, access) and their time complexities in Big-O notation. **CODE:  OUTPUT:**    **OBSERVATION:**  **This code models a real-world cafeteria order system using Python’s deque for efficient queue management. It supports adding orders, serving them in FIFO order, and displaying the current queue. The sample usage shows how orders are processed and updated, making it a practical and intuitive example of applying data structures to everyday 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.   **PROMPT:**  **This table compares data structures by operation speed. Arrays are fast for access, slow for changes. Linked lists and queues are quick to insert/delete but slow to search. Hash tables are fastest overall (average O(1)). Trees and heaps offer balanced performance, ideal for sorted or priority data.**  **CODE:**  **OUTPUT:**  **OBSERVATION:**  This code models a real-world cafeteria order system using Python’s deque for efficient queue management. It supports adding orders, serving them in FIFO order, and displaying the current queue. The sample usage shows how orders are processed and updated, making it a practical and intuitive example of applying data structures to everyday scenarios.  Top of Form |  |