AI-ASSISTED CODING

ASSIGNMENT-11.1

Data Structures with AI: Implementing Fundamental Structures

NAME: GATTU VARSHITH

HALL-TICKET: 2403A510E3

BATCH NO: 05

**TASK-1**:

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 GENERATED:**

class Stack:

    """

    A simple implementation of a stack (LIFO: Last In, First Out).

    """

    def \_\_init\_\_(self):

        """Initialize an empty stack."""

        self.items = []

    def push(self, item):

        """Add an item to the top of the stack."""

        self.items.append(item)

    def pop(self):

        """

        Remove and return the top item of the stack.

        Raises IndexError if the stack is empty.

        """

        if self.is\_empty():

            raise IndexError("Pop from empty stack")

        return self.items.pop()

    def peek(self):

        """

        Return the top item without removing it.

        Raises IndexError if the stack is empty.

        """

        if self.is\_empty():

            raise IndexError("Peek from empty stack")

        return self.items[-1]

    def is\_empty(self):

        """Check if the stack is empty."""

        return len(self.items) == 0

# ---------- Test Code ----------

if \_\_name\_\_ == "\_\_main\_\_":

    s = Stack()

    print("Is stack empty?", s.is\_empty())

    s.push(10)

    s.push(20)

    s.push(30)

    print("Top element:", s.peek())   # Should print 30

    print("Popped element:", s.pop()) # Removes 30

    print("Top element after pop:", s.peek())  # Should print 20

    print("Is stack empty?", s.is\_empty())

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**OBSERVATION :**

1. **Start:** stack is empty → []
2. **push(10):** → [10]
3. **push(20):** → [10, 20]
4. **push(30):** → [10, 20, 30]
5. **peek():** shows 30 (top element), stack still [10, 20, 30]
6. **pop():** removes 30, stack becomes [10, 20]
7. **peek():** shows 20
8. **is\_empty():** returns False (because [10, 20] is not empty).

TASK -2: Use AI to implement a Queue using Python lists.  
Sample Input Code:  
class Queue:  
pass

PROMPT:

Generate and implement a Queue using Python lists.

CODE GENERATED:

class Queue:

    """

    A simple implementation of a queue (FIFO: First In, First Out).

    """

    def \_\_init\_\_(self):

        """Initialize an empty queue."""

        self.items = []

    def enqueue(self, item):

        """

        Add an item to the end of the queue.

        Args:

            item: The element to be added.

        """

        self.items.append(item)

    def dequeue(self):

        """

        Remove and return the front item from the queue.

        Returns:

            The first inserted element.

        Raises:

            IndexError: If the queue is empty.

        """

        if self.is\_empty():

            raise IndexError("Dequeue from empty queue")

        return self.items.pop(0)

    def peek(self):

        """

        Return the front item without removing it.

        Returns:

            The first inserted element.

        Raises:

            IndexError: If the queue is empty.

        """

        if self.is\_empty():

            raise IndexError("Peek from empty queue")

        return self.items[0]

    def is\_empty(self):

        """

        Check if the queue is empty.

        Returns:

            True if empty, False otherwise.

        """

        return len(self.items) == 0

# ---------- Test Code ----------

if \_\_name\_\_ == "\_\_main\_\_":

    q = Queue()

    print("Is queue empty?", q.is\_empty())

    q.enqueue(10)

    q.enqueue(20)

    q.enqueue(30)

    print("Front element:", q.peek())   # Should print 10

    print("Dequeued element:", q.dequeue())  # Removes 10

    print("Front element after dequeue:", q.peek())  # Should print 20

    print("Is queue empty?", q.is\_empty())

OUTPUT:  
A black screen with white text

AI-generated content may be incorrect.

OBSERVATION:

1. Start: [] (empty queue)

2. enqueue(10): [10]

3. enqueue(20): [10, 20]

4. enqueue(30): [10, 20, 30]

5. peek(): front = 10, queue stays [10, 20, 30]

6. dequeue(): removes 10, queue becomes [20, 30]

7. peek(): front = 20

8. is\_empty(): → False

TASK – 3: Use AI to generate a Singly Linked List with insert and display methods.  
Sample Input Code:  
class Node:  
pass  
class LinkedList:  
pass

PROMPT:

Generate a Singly Linked List with insert and display methods.

CODE GENERATED:

class Node:

    """

    A node in a singly linked list.

    """

    def \_\_init\_\_(self, data):

        """

        Initialize a node with data and no next reference.

        """

        self.data = data

        self.next = None

class SinglyLinkedList:

    """

    A simple singly linked list with insert and display methods.

    """

    def \_\_init\_\_(self):

        """Initialize an empty list."""

        self.head = None

    def insert(self, data):

        """

        Insert a new node at the end of the list.

        Args:

            data: The data value to store in the new node.

        """

        new\_node = Node(data)

        if self.head is None:  # If list is empty

            self.head = new\_node

        else:

            current = self.head

            while current.next:  # Traverse to the end

                current = current.next

            current.next = new\_node

    def display(self):

        """

        Display all nodes in the list as a Python list.

        """

        elements = []

        current = self.head

        while current:

            elements.append(current.data)

            current = current.next

        print("Linked List:", elements)

# ---------- Test Code ----------

if \_\_name\_\_ == "\_\_main\_\_":

    ll = SinglyLinkedList()

    ll.insert(10)

    ll.insert(20)

    ll.insert(30)

    ll.display()  # Expected: Linked List: [10, 20, 30]

OUTPUT:



OBSERVATION:

A singly linked list is a linear data structure where each node points to the next, so after inserting 10, 20, 30 → [10 → 20 → 30].

1. Start: [] (empty)

2. Insert(10): [10]

3. insert(20): [10 → 20]

4. Insert(30): [10 → 20 → 30]

TASK -4:

Use AI to create a BST with insert and in-order traversal methods.  
Sample Input Code:  
class BST:  
pass

PROMPT:

create a BST with insert and in-order traversal methods in python.

CODE GENERATED:

class Node:

    """

    A node in the Binary Search Tree.

    """

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

class BST:

    """

    A simple Binary Search Tree with insert and in-order traversal.

    """

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, data):

        """

        Insert data into the BST.

        """

        if self.root is None:

            self.root = Node(data)

        else:

            self.\_insert\_recursive(self.root, data)

    def \_insert\_recursive(self, current, data):

        if data < current.data:

            if current.left is None:

                current.left = Node(data)

            else:

                self.\_insert\_recursive(current.left, data)

        elif data > current.data:

            if current.right is None:

                current.right = Node(data)

            else:

                self.\_insert\_recursive(current.right, data)

        # If equal, do nothing (ignoring duplicates)

    def inorder(self):

        """

        Perform in-order traversal and return elements as a list.

        """

        result = []

        self.\_inorder\_recursive(self.root, result)

        return result

    def \_inorder\_recursive(self, node, result):

        if node:

            self.\_inorder\_recursive(node.left, result)

            result.append(node.data)

            self.\_inorder\_recursive(node.right, result)

# ---------- Test Code ----------

if \_\_name\_\_ == "\_\_main\_\_":

    bst = BST()

    bst.insert(50)

    bst.insert(30)

    bst.insert(70)

    bst.insert(20)

    bst.insert(40)

    bst.insert(60)

    bst.insert(80)

    print("In-order Traversal:", bst.inorder())

OUTPUT:  


OBSERVATION:

In a Binary Search Tree (BST), left child < root < right child, and an in-order traversal always gives sorted data.

Observation (insert 50, 30, 70, 20, 40, 60, 80):  
In-order traversal → [20, 30, 40, 50, 60, 70, 80].

TASK -5:

Use AI to implement a hash table with basic insert, search, and delete methods.  
Sample Input Code:  
class HashTable:  
pass

PROMPT:

Generate and implement a hash table with basic insert, search, and delete methods.

CODE GENERATED:

class HashTable:

    """

    A simple hash table implementation with insert, search, and delete methods.

    Uses separate chaining for collision handling.

    """

    def \_\_init\_\_(self, size=10):

        """Initialize the hash table with empty buckets."""

        self.size = size

        self.table = [[] for \_ in range(size)]

    def \_hash(self, key):

        """Compute the hash index for a given key."""

        return hash(key) % self.size

    def insert(self, key, value):

        """

        Insert a key-value pair into the hash table.

        If key already exists, update its value.

        """

        index = self.\_hash(key)

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

            if k == key:  # Update existing

                self.table[index][i] = (key, value)

                return

        self.table[index].append((key, value))  # Add new

    def search(self, key):

        """

        Search for a key and return its value.

        Returns None if key not found.

        """

        index = self.\_hash(key)

        for k, v in self.table[index]:

            if k == key:

                return v

        return None

    def delete(self, key):

        """

        Delete a key-value pair by key.

        Returns True if deleted, False if not found.

        """

        index = self.\_hash(key)

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

            if k == key:

                del self.table[index][i]

                return True

        return False

    def display(self):

        """Display the contents of the hash table."""

        for i, bucket in enumerate(self.table):

            print(f"Bucket {i}: {bucket}")

# ---------- Test Code ----------

if \_\_name\_\_ == "\_\_main\_\_":

    ht = HashTable()

    ht.insert("apple", 100)

    ht.insert("banana", 200)

    ht.insert("grape", 300)

    print("Search apple:", ht.search("apple"))   # 100

    print("Search banana:", ht.search("banana")) # 200

    ht.delete("apple")

    print("Search apple after delete:", ht.search("apple")) # None

    ht.display()

OUTPUT:  
A computer screen shot of a black screen

AI-generated content may be incorrect.

OBSERVATION:

A hash table maps keys to indexes using a hash function, giving fast insert, search, delete operations.

1. Start → all buckets empty [].

2. Insert "apple" → 100 → goes to some bucket → [('apple', 100)].

3. Insert "banana" → 200 → another bucket → [('banana', 200)].

4. Insert "grape" → 300 → [('grape', 300)].

5. Search "banana" → found → 200.

6. Delete "apple" → removed from its bucket → "apple" not found anymore.

TASK -6:

Use AI to implement a graph using an adjacency list.  
Sample Input Code:  
class Graph:  
pass

PROMPT:

Generate and implement a graph using an adjacency list in python.

CODE GENERATED:

class Graph:

    """

    A simple graph implementation using an adjacency list.

    """

    def \_\_init\_\_(self):

        """Initialize an empty adjacency list."""

        self.adj\_list = {}

    def add\_vertex(self, vertex):

        """

        Add a vertex to the graph.

        """

        if vertex not in self.adj\_list:

            self.adj\_list[vertex] = []

    def add\_edge(self, v1, v2):

        """

        Add an edge between two vertices (undirected graph).

        """

        if v1 not in self.adj\_list:

            self.add\_vertex(v1)

        if v2 not in self.adj\_list:

            self.add\_vertex(v2)

        self.adj\_list[v1].append(v2)

        self.adj\_list[v2].append(v1)  # Comment this line for directed graph

    def display(self):

        """

        Display the adjacency list of the graph.

        """

        for vertex, neighbors in self.adj\_list.items():

            print(f"{vertex} -> {neighbors}")

# ---------- Test Code ----------

if \_\_name\_\_ == "\_\_main\_\_":

    g = Graph()

    g.add\_edge("A", "B")

    g.add\_edge("A", "C")

    g.add\_edge("B", "D")

    g.add\_edge("C", "D")

    g.display()

OUTPUT:  
A group of white text on a black background

AI-generated content may be incorrect.

OBSERVATION:

A graph using an adjacency list stores each vertex with a list of its connected neighbors.

1. Start → Graph empty {}

2. Add edge A–B → A -> [B], B -> [A]

3. Add edge A–C → A -> [B, C], C -> [A]

4. Add edge B–D → B -> [A, D], D -> [B]

5. Add edge C–D → C -> [A, D], D -> [B, C]

TASK -7:

Use AI to implement a priority queue using Python’s heapq module.  
Sample Input Code:  
class PriorityQueue:  
pass

PROMPT:

Generate and implement a priority queue using Python’s heapq module.

CODE GENERATED:

import heapq

class PriorityQueue:

    def \_\_init\_\_(self):

        self.queue = []

    def push(self, item, priority):

        # heapq uses min-heap, so priority is first

        heapq.heappush(self.queue, (priority, item))

    def pop(self):

        if self.queue:

            return heapq.heappop(self.queue)[1]

        return None

    def peek(self):

        if self.queue:

            return self.queue[0][1]

        return None

    def is\_empty(self):

        return len(self.queue) == 0

# Example usage

pq = PriorityQueue()

pq.push("eat", 2)

pq.push("sleep", 3)

pq.push("code", 1)

print("Peek:", pq.peek())      # → "code"

print("Pop:", pq.pop())        # → "code"

print("Pop:", pq.pop())        # → "eat"

print("Is Empty?", pq.is\_empty())

OUTPUT:  
A black background with white text

AI-generated content may be incorrect.

OBSERVATION:

A priority queue retrieves elements based on priority, not insertion order.

1. Start → empty queue []
2. Push "eat", 2 → queue: [("eat", 2)]
3. Push "sleep", 3 → queue: [("eat", 2), ("sleep", 3)]
4. Push "code", 1 → queue: [("code", 1), ("sleep", 3), ("eat", 2)] (heap automatically orders by priority)
5. Peek → "code" (highest priority, smallest number)
6. Pop → "code", queue now [("eat", 2), ("sleep", 3)]
7. Pop → "eat", queue now [("sleep", 3)]

TASK -8:

Use AI to implement a double-ended queue using collections.deque.  
Sample Input Code:  
class DequeDS:  
pass

PROMPT:

Generate and implement a double-ended queue using collections.deque.

CODE GENERATED:

from collections import deque

class DoubleEndedQueue:

    """

    A simple double-ended queue (deque) implementation using collections.deque

    """

    def \_\_init\_\_(self):

        self.dq = deque()

    def add\_front(self, item):

        """Add an item to the front of the deque."""

        self.dq.appendleft(item)

    def add\_rear(self, item):

        """Add an item to the rear of the deque."""

        self.dq.append(item)

    def remove\_front(self):

        """Remove and return the front item."""

        if self.is\_empty():

            return None

        return self.dq.popleft()

    def remove\_rear(self):

        """Remove and return the rear item."""

        if self.is\_empty():

            return None

        return self.dq.pop()

    def peek\_front(self):

        """Return the front item without removing it."""

        if self.is\_empty():

            return None

        return self.dq[0]

    def peek\_rear(self):

        """Return the rear item without removing it."""

        if self.is\_empty():

            return None

        return self.dq[-1]

    def is\_empty(self):

        """Check if deque is empty."""

        return len(self.dq) == 0

    def display(self):

        """Display all items in the deque."""

        print("Deque:", list(self.dq))

# ---------- Test Code ----------

if \_\_name\_\_ == "\_\_main\_\_":

    deq = DoubleEndedQueue()

    deq.add\_rear(10)

    deq.add\_rear(20)

    deq.add\_front(5)

    deq.display()  # Deque: [5, 10, 20]

    print("Front item:", deq.peek\_front())  # 5

    print("Rear item:", deq.peek\_rear())    # 20

    deq.remove\_front()

    deq.display()  # Deque: [10, 20]

    deq.remove\_rear()

    deq.display()  # Deque: [10]

OUTPUT:  
A black screen with white text

AI-generated content may be incorrect.

OBSERVATION:

A deque allows insertion and deletion from both ends efficiently.

1. Start → empty deque []

2. add\_rear(10) → [10]

3. add\_rear(20) → [10, 20]

4. add\_front(5) → [5, 10, 20]

5. peek\_front → 5, peek\_rear → 20

6. remove\_front → [10, 20]

7. remove\_rear → [10]

TASK -9:

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

PROMPT:

Compare Stack, Queue, Linked List, BST, Hash Table, Heap, and Graph with their time complexities.

CODE GENERATED:

# Define table data

headers = ["Data Structure", "Access", "Search", "Insertion", "Deletion", "Notes"]

rows = [

    ["Stack", "O(n)", "O(n)", "O(1)", "O(1)", "LIFO structure"],

    ["Queue", "O(n)", "O(n)", "O(1)", "O(1)", "FIFO structure"],

    ["Singly Linked List", "O(n)", "O(n)", "O(1)\*", "O(1)\*", "Single pointer"],

    ["Doubly Linked List", "O(n)", "O(n)", "O(1)\*", "O(1)\*", "Two pointers per node"],

    ["Binary Search Tree", "O(log n) avg", "O(log n) avg", "O(log n) avg", "O(log n) avg", "Unbalanced may degrade to O(n)"],

    ["Hash Table", "O(1) avg", "O(1) avg", "O(1) avg", "O(1) avg", "Handles collisions"],

    ["Heap (Min/Max)", "O(n)", "O(n)", "O(log n)", "O(log n)", "Priority queue"],

    ["Graph (Adjacency List)", "O(V+E)", "O(V+E)", "O(1)", "O(V)", "Efficient for sparse graphs"]

]

# Function to print markdown table

def print\_markdown\_table(headers, rows):

    # Header

    print("| " + " | ".join(headers) + " |")

    print("|" + "|".join(["-" \* len(h) for h in headers]) + "|")

    # Rows

    for row in rows:

        print("| " + " | ".join(row) + " |")

# Print table

print\_markdown\_table(headers, rows)

OUTPUT:  
A screen shot of a computer program

AI-generated content may be incorrect.

OBSERVATION:

Different data structures have different strengths; their time complexities vary by operation.

1. Stack / Queue → fast insertion/deletion at one end (O(1)), slower access/search (O(n))

2. Linked Lists → fast insertion/deletion at head/tail (O(1)), slower access/search (O(n))

3. BST → average O(log n) for search/insert/delete; can degrade to O(n) if unbalanced

4. Hash Table → very fast average O(1) for search/insert/delete, handles collisions

5. Heap → good for priority operations (O(log n) insert/delete), slower search (O(n))

6. Graph (Adjacency List) → access/search depend on vertices and edges (O(V+E))

TASK -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:  
o Stack  
o Queue  
o Priority Queue  
o Linked List  
o Binary Search Tree (BST)  
o Graph  
o Hash Table  
o 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.

PROMPT:

Implement a cafeteria order system using a Queue.

The system should allow:

1. Adding a new student order.

2. Serving the next student in line.

3. Displaying all pending orders.

Include comments, docstrings, and 3 assert test cases.

CODE GENERATED:

from collections import deque

class CafeteriaQueue:

    """

    A simple cafeteria order system using a Queue (FIFO).

    """

    def \_\_init\_\_(self):

        """Initialize an empty order queue."""

        self.queue = deque()

    def add\_order(self, student\_name):

        """Add a new student order to the queue."""

        self.queue.append(student\_name)

    def serve\_order(self):

        """Serve the next student in line and return their name."""

        if self.queue:

            return self.queue.popleft()

        return None

    def pending\_orders(self):

        """Return a list of all pending orders."""

        return list(self.queue)

# ---------- Test Cases ----------

cafeteria = CafeteriaQueue()

# Adding orders

cafeteria.add\_order("Alice")

cafeteria.add\_order("Bob")

cafeteria.add\_order("Charlie")

# Assert pending orders

assert cafeteria.pending\_orders() == ["Alice", "Bob", "Charlie"]

# Serving orders

assert cafeteria.serve\_order() == "Alice"

assert cafeteria.serve\_order() == "Bob"

# Pending orders after serving

assert cafeteria.pending\_orders() == ["Charlie"]

# Serve remaining order

assert cafeteria.serve\_order() == "Charlie"

assert cafeteria.serve\_order() is None  # Queue is empty

print("All tests passed!")

OUTPUT:



OBSERVATION:

* FIFO order maintained: Alice → Bob → Charlie
* Adding orders: O(1)
* Serving orders: O(1)
* Using deque ensures efficient operations at both ends.