

# **ASSIGNMENT-11.1**

G. Srinidhi

2303A52285

Batch - 43

## **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.

The screenshot shows a Jupyter Notebook cell with the following Python code:

```
[1] class Stack:
    """
    Stack implementation using Python list (LIFO).
    Methods:
    push(item) -> Add element to top
    pop()       -> Remove top element
    peek()      -> View top element
    is_empty()  -> Check stack empty
    """

    def __init__(self):
        """Initialize empty stack."""
        self.items = []

    def push(self, item):
        """Insert element on top of stack."""
        self.items.append(item)

    def pop(self):
        """Remove and return top element."""
        return self.items.pop() if not self.is_empty() else None

    def peek(self):
        """Return top element without removing."""
        return self.items[-1] if not self.is_empty() else None

    def is_empty(self):
        """Return True if stack is empty."""
        return len(self.items) == 0

    # ===== Demo Input =====
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
print("Top element:", stack.peek())
print("Removed element:", stack.pop())
print("Is stack empty:", stack.is_empty())
print("Final stack:", stack.items)
```

Output:

```
... Top element: 30
Removed element: 30
Is stack empty: False
Final stack: [10, 20]
```

## 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.

The screenshot shows a Jupyter Notebook interface with the following details:

- File:** AIAC-11.1.ipynb
- Cells:** [2]
- Code Content:**

```
class Queue:  
    """  
        Queue implementation using Python list (FIFO).  
  
    Methods:  
        enqueue(item) -> Add element at rear  
        dequeue()      -> Remove element from front  
        peek()         -> View front element  
        size()         -> Return number of elements  
    """  
  
    def __init__(self):  
        """Initialize empty queue."""  
        self.items = []  
    def enqueue(self, item):  
        """Insert element at rear."""  
        self.items.append(item)  
    def dequeue(self):  
        """Remove and return front element."""  
        return self.items.pop(0) if self.items else None  
    def peek(self):  
        """Return front element without removing."""  
        return self.items[0] if self.items else None  
    def size(self):  
        """Return queue size."""  
        return len(self.items)  
  
# ===== Demo Input =====  
q = Queue()  
q.enqueue("A")  
q.enqueue("B")  
q.enqueue("C")  
print("Front element:", q.peek())  
print("Removed element:", q.dequeue())  
print("Queue size:", q.size())  
print("Final queue:", q.items)
```
- Output:**

```
Front element: A  
Removed element: A  
Queue size: 2  
Final queue: ['B', 'C']
```

## 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.

```
Final queue: ['B', 'C']

[3]
  class Node:
      """Represents a single node in the linked list."""
      def __init__(self, data):
          """Initialize node with data and next pointer."""
          self.data = data
          self.next = None

  class LinkedList:
      """
      Singly Linked List implementation.

      Methods:
      insert(data) -> Insert element at end
      display() -> Print linked list
      """

      def __init__(self):
          """Initialize empty linked list."""
          self.head = None

      def insert(self, data):
          """Insert a new node at the end of the list."""
          new_node = Node(data)
          if not self.head:
              self.head = new_node
              return
          temp = self.head
          while temp.next:
              temp = temp.next
          temp.next = new_node

      def display(self):
          """Display all elements of the linked list."""
          temp = self.head
          while temp:
              print(temp.data, end=" -> ")
              temp = temp.next
          print("None")

      # ===== Demo Input =====
      l1 = LinkedList()

      l1.insert(10)
      l1.insert(20)
      l1.insert(30)

      print("Linked List:")
      l1.display()

  ... Linked List:
  10 -> 20 -> 30 -> None
```

## 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

The screenshot shows a Jupyter Notebook interface with a dark theme. The code cell contains Python code for a Binary Search Tree (BST) class. The code includes an \_\_init\_\_ method for initializing a node with a key and left/right pointers, and an insert method for recursive insertion. It also includes an inorder traversal method for in-order traversal. The notebook then prompts the user to enter root value and number of nodes, followed by values to insert, and finally prints the in-order traversal of the tree.

```

[6]
class BST:
    """Binary Search Tree"""
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
    def insert(self, data):
        """Recursive insert"""
        if data < self.key:
            if self.left:
                self.left.insert(data)
            else:
                self.left = BST(data)
        else:
            if self.right:
                self.right.insert(data)
            else:
                self.right = BST(data)
    def inorder(self):
        """In-order traversal"""
        if self.left:
            self.left.inorder()
        print(self.key, end=" ")
        if self.right:
            self.right.inorder()
root_val = int(input("Enter root value: "))
root = BST(root_val)
n = int(input("Enter number of nodes: "))
for _ in range(n):
    root.insert(int(input("Enter value: ")))
print("Inorder Traversal:")
root.inorder()

```

... Enter root value: 3  
 Enter number of nodes: 4  
 Enter value: 8  
 Enter value: 7  
 Enter value: 6  
 Enter value: 5  
 Inorder Traversal:  
 3 5 6 7 8

## Task Description #5 – Hash Table

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

Sample Input Code:

class Hash Table:

pass

Expected Output:

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

The screenshot shows a Python code editor interface with a dark theme. On the left is a sidebar with various icons for file operations. The main area contains a code editor window with the following Python script:

```
class HashTable:
    """Hash Table using chaining"""
    def __init__(self, size=10):
        self.table = [[] for _ in range(size)]
    def insert(self, key, value):
        index = hash(key) % len(self.table)
        self.table[index].append((key, value))
    def search(self, key):
        index = hash(key) % len(self.table)
        for k, v in self.table[index]:
            if k == key:
                return v
        return "Not Found"
    def delete(self, key):
        index = hash(key) % len(self.table)
        self.table[index] = [(k, v) for k, v in self.table[index] if k != key]
ht = HashTable()
n = int(input("Enter number of key-value pairs: "))
for _ in range(n):
    k = input("Enter key: ")
    v = input("Enter value: ")
    ht.insert(k, v)
search_key = input("Enter key to search: ")
print("Result:", ht.search(search_key))
```

Below the code editor, the terminal output is displayed:

```
Enter number of key-value pairs: 4
Enter key: 1
Enter value: 2
Enter key: 3
Enter value: 4
Enter key: 5
Enter value: 6
Enter key: 7
Enter value: 8
Enter key to search: 5
Result: 6
```

## 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.

The screenshot shows a terminal window with the following content:

```
[28] 24s
▶ class Graph:
    """Graph using adjacency list"""

    def __init__(self):
        self.graph = {}

    def add_vertex(self, v):
        self.graph[v] = []

    def add_edge(self, v1, v2):
        self.graph[v1].append(v2)

    def display(self):
        for v in self.graph:
            print(v, "->", self.graph[v])

g = Graph()
v = int(input("Enter number of vertices: "))
for _ in range(v):
    g.add_vertex(input("Enter vertex: "))
e = int(input("Enter number of edges: "))
for _ in range(e):
    g.add_edge(input("From: "), input("To: "))
g.display()

...
Enter number of vertices: 3
Enter vertex: a
Enter vertex: b
Enter vertex: c
Enter number of edges: 2
From: a
To: b
From: a
To:
a -> ['b', ' ']
b -> []
c -> []
```

## Task Description #7 – Priority Queue

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

Sample Input Code:

```
class Priority Queue:
```

```
pass
```

**Expected Output:**

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

The screenshot shows a code editor interface with a dark theme. The menu bar includes File, Edit, View, Insert, Runtime, Tools, and Help. The toolbar has icons for Commands, Code, Text, and Run all. A sidebar on the left contains icons for file operations like New, Open, Save, Find, and others. The main area displays Python code for a Priority Queue using the heapq module. The code defines a class PriorityQueue with methods for enqueueing and dequeuing items based on priority. It uses the heap push/pop operations from heapq. A sample run of the code is shown at the bottom, demonstrating its functionality with user input and output.

```
import heapq

class PriorityQueue:
    """Priority Queue using heapq"""

    def __init__(self):
        self.queue = []

    def enqueue(self, item, priority):
        heapq.heappush(self.queue, (priority, item))

    def dequeue(self):
        print("Removed:", heapq.heappop(self.queue)[1])

    def display(self):
        print(self.queue)

pq = PriorityQueue()
n = int(input("Enter number of elements: "))
for _ in range(n):
    item = input("Enter item: ")
    priority = int(input("Enter priority: "))
    pq.enqueue(item, priority)
pq.dequeue()
pq.display()

...
Enter number of elements: 2
Enter item: 1
Enter priority: 2
Enter item: 2
Enter priority: 1
Removed: (1, '2')
[(2, '1')]
```

## Task Description #8 – Deque

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

Sample Input Code:

```
class Deque DS:
```

```
pass
```

Expected Output:

- Insert and remove from both ends with docstrings.

The screenshot shows a Jupyter Notebook interface with a dark theme. On the left is a sidebar with various icons for file operations. The main area has a toolbar with 'Commands', 'Code', 'Text', and 'Run all' buttons. A code cell is open, showing the following Python code:

```
[13] ⏪ from collections import deque

class DequeDS:
    """Double Ended Queue"""

    def __init__(self):
        self.dq = deque()

    def insert_front(self, item):
        self.dq.appendleft(item)

    def insert_rear(self, item):
        self.dq.append(item)

    def remove_front(self):
        return self.dq.popleft()

    def remove_rear(self):
        return self.dq.pop()

d = DequeDS()
n = int(input("Enter number of elements: "))
for _ in range(n):
    d.insert_rear(int(input("Enter element: ")))
print("Deque:", d.dq)
```

Below the code cell, the output of the execution is shown:

```
... Enter number of elements: 3
Enter element: 2
Enter element: 3
Enter element: 4
Deque: deque([2, 3, 4])
```

## Task Description #9 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.

```

Enter element: 4
Deque: deque([2, 3, 4])

[14] ✓ 24s  class CafeteriaQueue:
    """Serve students in FIFO order"""

    def __init__(self):
        self.queue = []
    def arrive(self, name):
        self.queue.append(name)
    def serve(self):
        print("Served:", self.queue.pop(0))
cq = CafeteriaQueue()
n = int(input("Enter number of students: "))
for _ in range(n):
    cq.arrive(input("Enter student name: "))
cq.serve()

...
Enter number of students: 4
Enter student name: nidhi
Enter student name: sanju
Enter student name: rosh
Enter student name: likki
Served: nidhi

```

## **Task Description #10: Smart E-Commerce Platform – Data Structure**

### **Challenge**

An e-commerce company wants to build a Smart Online Shopping System with:

1. Shopping Cart Management – Add and remove products dynamically.
2. Order Processing System – Orders processed in the order they are placed.
3. Top-Selling Products Tracker – Products ranked by sales count.
4. Product Search Engine – Fast lookup of products using product ID.
5. Delivery Route Planning – Connect warehouses and delivery locations.

**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.

Commands + Code ▾ + Text ▾ Run all ▾



[15]  
✓ 11s



```
class OrderQueue:  
    """Order processing using Queue"""\n\n    def __init__(self):  
        self.orders = []\n    def place_order(self, order):  
        self.orders.append(order)\n    def process_order(self):  
        print("Processed:", self.orders.pop(0))\noq = OrderQueue()  
n = int(input("Enter number of orders: "))  
for _ in range(n):  
    oq.place_order(input("Enter order ID: "))  
oq.process_order()
```

... Enter number of orders: 3  
Enter order ID: 2285  
Enter order ID: 2393  
Enter order ID: 2292  
Processed: 2285