AI ASSITED CODING

Assignment-11.5

Name:Sai Sathwika

HT NO. : 2303A52204

Batch No. :35

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.

```python
class Stack:
    """A simple implementation of a Stack data structure."""

    def __init__(self):
        """Initializes an empty stack."""
        self._items = []

    def push(self, item):
        """Adds an item to the top of the stack.

        Args:
            item: The item to be added to the stack.
        """
        self._items.append(item)

    def pop(self):
        """Removes and returns the item from the top of the stack.

        Returns:
            The item removed from the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        return self._items.pop()

    def peek(self):
        """Returns the item at the top of the stack without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from an empty stack")
```

```python
        if self.is_empty():
            raise IndexError("peek from an empty stack")
        return self._items[-1]

    def is_empty(self):
        """Checks if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """Returns the number of items in the stack.

        Returns:
            The number of items in the stack.
        """
        return len(self._items)

    def __str__(self):
        """Returns a string representation of the stack."""
        return str(self._items)

# Example Usage:
# my_stack = Stack()
# my_stack.push(10)
# my_stack.push(20)
# print(f"Stack: {my_stack}")
# print(f"Peek: {my_stack.peek()}")
# print(f"Pop: {my_stack.pop()}")
# print(f"Stack after pop: {my_stack}")
# print(f"Is empty: {my_stack.is_empty()}")
# my_stack.pop() # This will make the stack empty
# print(f"Is empty after second pop: {my_stack.is_empty()}")
# # my_stack.pop() # This would raise an IndexError
```

```python
my_stack = Stack()
print(f"Initial Stack: {my_stack}")
print(f"Is stack empty? {my_stack.is_empty()}")

my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
print(f"Stack after pushes: {my_stack}")
print(f"Stack size: {my_stack.size()}")

print(f"Peek: {my_stack.peek()}")
print(f"Stack after peek (should be unchanged): {my_stack}")

print(f"Pop: {my_stack.pop()}")
print(f"Stack after first pop: {my_stack}")

print(f"Pop: {my_stack.pop()}")
print(f"Stack after second pop: {my_stack}")

print(f"Is stack empty? {my_stack.is_empty()}")

print(f"Pop: {my_stack.pop()}")
print(f"Stack after third pop: {my_stack}")

print(f"Is stack empty? {my_stack.is_empty()}")

# Uncomment the line below to see an IndexError for popping from an empty stack
# my_stack.pop()
```

```
Initial Stack: []
Is stack empty? True
Stack after pushes: [10, 20, 30]
Stack size: 3
Peek: 30
Stack after peek (should be unchanged): [10, 20, 30]
Pop: 30
Stack after first pop: [10, 20]
Pop: 20
Stack after second pop: [10]
Is stack empty? False
Pop: 10
Stack after third pop: []
Is stack empty? True
```

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.

```python
class Queue:
    """A simple implementation of a Queue (FIFO) data structure using a Python list."""

    def __init__(self):
        """Initializes an empty queue."""
        self._items = []

    def enqueue(self, item):
        """Adds an item to the rear of the queue.

        Args:
            item: The item to be added to the queue.
        """
        self._items.append(item)

    def dequeue(self):
        """Removes and returns the item from the front of the queue.

        Returns:
            The item removed from the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        return self._items.pop(0) # pop(0) removes from the beginning (front)

    def peek(self):
        """Returns the item at the front of the queue without removing it.

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
```

```
[3]
✓ 0s
            raise IndexError("peek from an empty queue")
        return self._items[0]

    def is_empty(self):
        """Checks if the queue is empty.

        Returns:
            True if the queue is empty, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """Returns the number of items in the queue.

        Returns:
            The number of items in the queue.
        """
        return len(self._items)

    def __str__(self):
        """Returns a string representation of the queue."""
        return f"Queue: {self._items}"

# Example Usage:
# my_queue = Queue()
# print(f"Is queue empty? {my_queue.is_empty()}")
# my_queue.enqueue(1)
# my_queue.enqueue(2)
# my_queue.enqueue(3)
# print(f"Queue after enqueues: {my_queue}")
# print(f"Queue size: {my_queue.size()}")
# print(f"Peek: {my_queue.peek()}")
# print(f"Dequeued item: {my_queue.dequeue()}")
# print(f"Queue after dequeue: {my_queue}")
# print(f"Is queue empty? {my_queue.is_empty()}")
# my_queue.dequeue()
# my_queue.dequeue()
# print(f"Is queue empty? {my_queue.is_empty()}")
# # my_queue.dequeue() # This would raise an IndexError
```

```
[4]
✓ 0s
my_queue = Queue()
print(f"Initial Queue: {my_queue}")
print(f"Is queue empty? {my_queue.is_empty()}")

my_queue.enqueue(100)
my_queue.enqueue(200)
my_queue.enqueue(300)
print(f"Queue after enqueues: {my_queue}")
print(f"Queue size: {my_queue.size()}")

print(f"Peek: {my_queue.peek()}")
print(f"Queue after peek (should be unchanged): {my_queue}")

print(f"Dequeued item: {my_queue.dequeue()}")
print(f"Queue after first dequeue: {my_queue}")

print(f"Dequeued item: {my_queue.dequeue()}")
print(f"Queue after second dequeue: {my_queue}")

print(f"Is queue empty? {my_queue.is_empty()}")

print(f"Dequeued item: {my_queue.dequeue()}")
print(f"Queue after third dequeue: {my_queue}")

print(f"Is queue empty? {my_queue.is_empty()}")

# Uncomment the line below to see an IndexError for dequeuing from an empty queue
# my_queue.dequeue()
```

```
Initial Queue: Queue: []
Is queue empty? True
Queue after enqueues: Queue: [100, 200, 300]
Queue size: 3
Peek: 100
Queue after peek (should be unchanged): Queue: [100, 200, 300]
Dequeued item: 100
Queue after first dequeue: Queue: [200, 300]
Dequeued item: 200
Queue after second dequeue: Queue: [300]
Is queue empty? False
Dequeued item: 300
Queue after third dequeue: Queue: []
Is queue empty? True
```

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.

```
class Node:
    """Represents a single node in a Singly Linked List."""

    def __init__(self, data):
        """Initializes a new Node.

        Args:
            data: The data to be stored in the node.
        """
        self.data = data
        self.next = None  # Pointer to the next node

    def __str__(self):
        """Returns a string representation of the node's data."""
        return str(self.data)
```

```
class LinkedList:
    """Implements a Singly Linked List data structure."""

    def __init__(self):
        """Initializes an empty LinkedList. The head is initially None."""
        self.head = None

    def is_empty(self):
        """Checks if the linked list is empty.

        Returns:
            bool: True if the list is empty, False otherwise.
        """
        return self.head is None

    def insert_at_end(self, data):
        """Inserts a new node with the given data at the end of the list.

        Args:
            data: The data to be stored in the new node.
        """
```

```
[6]
✓ 0s
        Args:
            data: The data to be stored in the new node.
        """
        new_node = Node(data)
        if self.is_empty():
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def insert_at_beginning(self, data):
        """Inserts a new node with the given data at the beginning of the list.

        Args:
            data: The data to be stored in the new node.
        """
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        """Prints all the elements in the linked list from head to tail.
        If the list is empty, it prints an appropriate message.
        """
        if self.is_empty():
            print("The list is empty.")
            return

        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements))

    def __str__(self):
        """Returns a string representation of the linked list."""
```

```
[6]
✓ 0s
    def __str__(self):
        """Returns a string representation of the linked list."""
        if self.is_empty():
            return "[]"
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        return f"[{' -> '.join(elements)}]"

# Example Usage:
# my_list = LinkedList()
# print("Initial list:")
# my_list.display()

# my_list.insert_at_end(10)
# my_list.insert_at_end(20)
# my_list.insert_at_end(30)
# print("List after inserting 10, 20, 30 at end:")
# my_list.display() # Expected: 10 -> 20 -> 30

# my_list.insert_at_beginning(5)
# print("List after inserting 5 at beginning:")
# my_list.display() # Expected: 5 -> 10 -> 20 -> 30

# print(f"Is list empty? {my_list.is_empty()}")
# print(f"String representation: {my_list}")
```

```python
my_list = LinkedList()
print("Initial list:")
my_list.display()
print(f"Is list empty? {my_list.is_empty()}")
print(f"String representation: {my_list}")

print("\n--- Inserting at End ---")
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)
print("List after inserting 10, 20, 30 at end:")
my_list.display() # Expected: 10 -> 20 -> 30
print(f"Is list empty? {my_list.is_empty()}")
print(f"String representation: {my_list}")

print("\n--- Inserting at Beginning ---")
my_list.insert_at_beginning(5)
my_list.insert_at_beginning(1)
print("List after inserting 5 and 1 at beginning:")
my_list.display() # Expected: 1 -> 5 -> 10 -> 20 -> 30
print(f"String representation: {my_list}")
```

```
Initial list:
The list is empty.
Is list empty? True
String representation: []

--- Inserting at End ---
List after inserting 10, 20, 30 at end:
10 -> 20 -> 30
Is list empty? False
String representation: [10 -> 20 -> 30]

--- Inserting at Beginning ---
List after inserting 5 and 1 at beginning:
1 -> 5 -> 10 -> 20 -> 30
String representation: [1 -> 5 -> 10 -> 20 -> 30]
```

```python
my_list = LinkedList()
print("Initial list:")
my_list.display()
print(f"Is list empty? {my_list.is_empty()}")
print(f"String representation: {my_list}")

print("\n--- Inserting at End ---")
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)
print("List after inserting 10, 20, 30 at end:")
my_list.display() # Expected: 10 -> 20 -> 30
print(f"Is list empty? {my_list.is_empty()}")
print(f"String representation: {my_list}")

print("\n--- Inserting at Beginning ---")
my_list.insert_at_beginning(5)
my_list.insert_at_beginning(1)
print("List after inserting 5 and 1 at beginning:")
my_list.display() # Expected: 1 -> 5 -> 10 -> 20 -> 30
print(f"String representation: {my_list}")
```

```
Initial list:
The list is empty.
Is list empty? True
String representation: []

--- Inserting at End ---
List after inserting 10, 20, 30 at end:
10 -> 20 -> 30
Is list empty? False
String representation: [10 -> 20 -> 30]

--- Inserting at Beginning ---
List after inserting 5 and 1 at beginning:
1 -> 5 -> 10 -> 20 -> 30
String representation: [1 -> 5 -> 10 -> 20 -> 30]
```

Task Description #4 – 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.

```python
class HashTable:
    """A simple Hash Table implementation using chaining for collision resolution."""

    def __init__(self, capacity=10):
        """Initializes the Hash Table with a given capacity.

        Args:
            capacity (int): The number of buckets in the hash table. Defaults to 10.
        """
        self.capacity = capacity
        # Initialize buckets as a list of empty lists (for chaining)
        self.buckets = [[] for _ in range(self.capacity)]

    def _hash(self, key):
        """Computes the hash index for a given key.

        This uses Python's built-in hash function and the modulo operator.

        Args:
            key: The key to be hashed.

        Returns:
            int: The index of the bucket where the key-value pair should be stored.
        """
        return hash(key) % self.capacity

    def insert(self, key, value):
        """Inserts a key-value pair into the hash table.

        If the key already exists, its value will be updated.
        Uses chaining to handle collisions.

        Args:
            key: The key to insert.
            value: The value associated with the key.
        """
        index = self._hash(key)
        bucket = self.buckets[index]
```

[9] 0s

```python
        # Check if key already exists in the bucket
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value) # Update value if key exists
                return

        # If key doesn't exist, append new key-value pair
        bucket.append((key, value))

    def search(self, key):
        """Searches for a key in the hash table and returns its associated value.

        Args:
            key: The key to search for.

        Returns:
            Any: The value associated with the key if found, None otherwise.
        """
        index = self._hash(key)
        bucket = self.buckets[index]

        # Iterate through the bucket to find the key
        for k, v in bucket:
            if k == key:
                return v # Key found, return its value

        return None # Key not found

    def delete(self, key):
        """Deletes a key-value pair from the hash table.

        Args:
            key: The key of the item to delete.

        Returns:
            bool: True if the key was found and deleted, False otherwise.
        """
        index = self._hash(key)
        bucket = self.buckets[index]
```

[9] 0s

```python
        # Iterate through the bucket to find and remove the key-value pair
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i] # Remove the item
                return True # Key found and deleted

        return False # Key not found

    def display(self):
        """Prints the contents of the hash table, showing each bucket."""
        print("--- Hash Table Contents ---")
        for i, bucket in enumerate(self.buckets):
            print(f"Bucket {i}: {bucket}")
        print("---------------------------")

# Example Usage:
# ht = HashTable(capacity=5)

# print("Inserting values:")
# ht.insert("apple", 10)
# ht.insert("banana", 20)
# ht.insert("cherry", 30)
# ht.insert("date", 40) # Might collide depending on hash values
# ht.insert("elderberry", 50)
# ht.insert("fig", 60)
# ht.insert("grape", 70)
# ht.display()

# print("\nSearching for values:")
# print(f"Value for 'apple': {ht.search('apple')}") # Expected: 10
# print(f"Value for 'date': {ht.search('date')}")   # Expected: 40
# print(f"Value for 'kiwi': {ht.search('kiwi')}")   # Expected: None

# print("\nUpdating a value:")
# ht.insert("apple", 100)
# print(f"New value for 'apple': {ht.search('apple')}") # Expected: 100
# ht.display()

# print("\nDeleting values:")
```

```
[9]
# print("\nDeleting values:")
# print(f"Deleting 'banana': {ht.delete('banana')}") # Expected: True
# print(f"Deleting 'kiwi': {ht.delete('kiwi')}")   # Expected: False
# ht.display()

# print(f"Value for 'banana' after deletion: {ht.search('banana')}") # Expected: None
```

```
[10]
ht = HashTable(capacity=5)

print("Inserting values:")
ht.insert("apple", 10)
ht.insert("banana", 20)
ht.insert("cherry", 30)
ht.insert("date", 40) # Might collide depending on hash values
ht.insert("elderberry", 50)
ht.insert("fig", 60)
ht.insert("grape", 70)
ht.display()

print("\nSearching for values:")
print(f"Value for 'apple': {ht.search('apple')}") # Expected: 10
print(f"Value for 'date': {ht.search('date')}")   # Expected: 40
print(f"Value for 'kiwi': {ht.search('kiwi')}")   # Expected: None

print("\nUpdating a value:")
ht.insert("apple", 100)
print(f"New value for 'apple': {ht.search('apple')}") # Expected: 100
ht.display()

print("\nDeleting values:")
print(f"Deleting 'banana': {ht.delete('banana')}") # Expected: True
print(f"Deleting 'kiwi': {ht.delete('kiwi')}")   # Expected: False
ht.display()

print(f"Value for 'banana' after deletion: {ht.search('banana')}") # Expected: None
```

```
Inserting values:
--- Hash Table Contents ---
Bucket 0: [('elderberry', 50)]
Bucket 1: [('apple', 10), ('fig', 60)]
Bucket 2: [('grape', 70)]
Bucket 3: [('banana', 20), ('cherry', 30), ('date', 40)]
Bucket 4: []
---------------------------

Searching for values:
Value for 'apple': 10
Value for 'date': 40
Value for 'kiwi': None

Updating a value:
New value for 'apple': 100
--- Hash Table Contents ---
Bucket 0: [('elderberry', 50)]
Bucket 1: [('apple', 100), ('fig', 60)]
Bucket 2: [('grape', 70)]
Bucket 3: [('banana', 20), ('cherry', 30), ('date', 40)]
Bucket 4: []
---------------------------

Deleting values:
Deleting 'banana': True
Deleting 'kiwi': False
--- Hash Table Contents ---
Bucket 0: [('elderberry', 50)]
Bucket 1: [('apple', 100), ('fig', 60)]
Bucket 2: [('grape', 70)]
Bucket 3: [('cherry', 30), ('date', 40)]
Bucket 4: []
---------------------------
Value for 'banana' after deletion: None
```

Task Description #5 – 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.

```
[11]
class Graph:
    """A Graph data structure implemented using an adjacency list."""

    def __init__(self):
        """Initializes an empty graph. The adjacency list stores vertices as keys
        and a list of their neighbors as values.
        """
        self.adj_list = {}

    def add_vertex(self, vertex):
        """Adds a vertex to the graph.

        If the vertex already exists, it does nothing.

        Args:
            vertex: The vertex to be added.
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []
            print(f"Vertex '{vertex}' added.")
        else:
            print(f"Vertex '{vertex}' already exists.")

    def add_edge(self, vertex1, vertex2, is_directed=False):
        """Adds an edge between two vertices.

        If vertices do not exist, they are added. Supports directed and undirected edges.

        Args:
            vertex1: The first vertex.
            vertex2: The second vertex.
            is_directed (bool): If True, adds a directed edge from vertex1 to vertex2.
                                If False (default), adds an undirected edge.

        Raises:
            ValueError: If an edge is attempted to be added from a vertex to itself.
        """
        if vertex1 == vertex2:
            raise ValueError("Cannot add an edge from a vertex to itself.")
```

```
[11]
0s
        # Ensure both vertices exist in the graph
        if vertex1 not in self.adj_list:
            self.add_vertex(vertex1)
        if vertex2 not in self.adj_list:
            self.add_vertex(vertex2)

        # Add edge from vertex1 to vertex2
        if vertex2 not in self.adj_list[vertex1]:
            self.adj_list[vertex1].append(vertex2)
            print(f"Edge added: {vertex1} -> {vertex2}")

        # If undirected, also add edge from vertex2 to vertex1
        if not is_directed and vertex1 not in self.adj_list[vertex2]:
            self.adj_list[vertex2].append(vertex1)
            print(f"Edge added: {vertex2} -> {vertex1}")

    def display(self):
        """Displays the adjacency list representation of the graph."""
        print("\n--- Graph Adjacency List ---")
        if not self.adj_list:
            print("Graph is empty.")
            return
        for vertex, neighbors in self.adj_list.items():
            print(f"{vertex}: {neighbors}")
        print("----------------------------")

    def get_vertices(self):
        """Returns a list of all vertices in the graph.

        Returns:
            list: A list containing all vertices.
        """
        return list(self.adj_list.keys())

    def get_neighbors(self, vertex):
        """Returns the neighbors of a given vertex.

        Args:
            vertex: The vertex whose neighbors are to be returned.
```

```
[11]
0s
        Returns:
            list: A list of neighbors of the vertex. Returns an empty list if vertex
                  does not exist or has no neighbors.
        """
        return self.adj_list.get(vertex, [])

# Example Usage:
# g = Graph()

# # Add vertices
# g.add_vertex('A')
# g.add_vertex('B')
# g.add_vertex('C')
# g.add_vertex('D')
# g.display()

# # Add undirected edges
# print("\nAdding undirected edges:")
# g.add_edge('A', 'B')
# g.add_edge('B', 'C')
# g.add_edge('A', 'C')
# g.display()

# # Add directed edge
# print("\nAdding a directed edge:")
# g.add_edge('C', 'D', is_directed=True)
# g.display()

# # Try adding an existing vertex or edge
# print("\nTrying existing operations:")
# g.add_vertex('A')
# g.add_edge('A', 'B')
# g.display()

# print("\nVertices in graph: ", g.get_vertices())
# print("Neighbors of C: ", g.get_neighbors('C'))
# print("Neighbors of X (non-existent): ", g.get_neighbors('X'))
```

[12]
✓ 0s

```python
g = Graph()

# Add vertices
print("\n--- Adding Vertices ---")
g.add_vertex('A')
g.add_vertex('B')
g.add_vertex('C')
g.add_vertex('D')
g.add_vertex('A') # Try adding existing vertex
g.display()

# Add undirected edges
print("\n--- Adding Undirected Edges ---")
g.add_edge('A', 'B')
g.add_edge('B', 'C')
g.add_edge('A', 'C')
g.display()

# Add directed edge
print("\n--- Adding a Directed Edge ---")
g.add_edge('C', 'D', is_directed=True)
g.display()

# Try adding an existing edge
print("\n--- Trying to add existing edge ---")
g.add_edge('A', 'B') # This edge already exists
g.display()

# Get vertices and neighbors
print("\n--- Querying Graph Information ---")
print("Vertices in graph: ", g.get_vertices())
print("Neighbors of C: ", g.get_neighbors('C'))
print("Neighbors of A: ", g.get_neighbors('A'))
print("Neighbors of D: ", g.get_neighbors('D'))
print("Neighbors of X (non-existent): ", g.get_neighbors('X'))

# Try adding an edge from a vertex to itself
# try:
#     g.add_edge('A', 'A')
```

```
[12]
✓ 0s
# except ValueError as e:
#     print(f"\nError: {e}")
```

```
--- Adding Vertices ---
Vertex 'A' added.
Vertex 'B' added.
Vertex 'C' added.
Vertex 'D' added.
Vertex 'A' already exists.

--- Graph Adjacency List ---
A: []
B: []
C: []
D: []
----------------------------

--- Adding Undirected Edges ---
Edge added: A -> B
Edge added: B -> A
Edge added: B -> C
Edge added: C -> B
Edge added: A -> C
Edge added: C -> A

--- Graph Adjacency List ---
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A']
D: []
----------------------------

--- Adding a Directed Edge ---
Edge added: C -> D

--- Graph Adjacency List ---
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A', 'D']
D: []
----------------------------
```

ariables   Terminal

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

--- Trying to add existing edge ---

--- Graph Adjacency List ---
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A', 'D']
D: []
----------------------------

--- Querying Graph Information ---
Vertices in graph:  ['A', 'B', 'C', 'D']
Neighbors of C:  ['B', 'A', 'D']
Neighbors of A:  ['B', 'C']
Neighbors of D:  []
Neighbors of X (non-existent):  []
```

Task Description #6: Smart Hospital Management System – Data

Structure Selection

A hospital wants to develop a Smart Hospital Management System that handles:

1. Patient Check-In System – Patients are registered and treated in order of arrival.
2. Emergency Case Handling – Critical patients must be treated first.
3. Medical Records Storage – Fast retrieval of patient details using ID.
4. Doctor Appointment Scheduling – Appointments sorted by time.
5. Hospital Room Navigation – Represent connections between wards and rooms.

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.

Task

The task is to choose the most appropriate data structures for five hospital management system features, justifying each choice, and then implement the 'Hospital Room Navigation' feature using a Graph data structure (adjacency list) in Python. The final output should include the table of feature-to-data structure mappings with justifications, and the Python implementation of the navigation system with a brief explanation.

Select Data Structures and Justify

Subtask:

For each of the five hospital management system features, select the most appropriate data structure from the provided list and write a 2-3 sentence justification for each choice. This will result in a table mapping feature to data structure and justification.

Selected Data Structures and Justifications

Feature	Chosen Data Structure	Justification
Hospital Room Navigation	Graph (Adjacency List)	A graph is ideal for representing interconnected rooms and pathways within a hospital. An adjacency list allows for efficient storage and traversal of sparse graphs, enabling shortest path algorithms for navigation.
Patient Records Management	Hash Map/Dictionary	A hash map allows for very fast O(1) average-case retrieval of patient records using a unique identifier (e.g., patient ID) as the key. This is crucial for quick access to vital patient information by different hospital departments.
Appointment Scheduling	Tree (Self-Balancing BST)	A tree structure, particularly a self-balancing binary search tree, can efficiently store and retrieve appointments based on time or date. It allows for quick insertion, deletion, and searching of appointments, and supports range queries for specific timeframes.
Medical Supply Inventory	Hash Map/Dictionary	A hash map is excellent for tracking various medical supplies, with the item name or ID as the key and its quantity/details as the value. It provides efficient O(1) average-case lookup, addition, and removal of items, essential for real-time inventory management.
Emergency Room Patient Queue	Priority Queue (via Heap)	A priority queue ensures that patients with the most urgent conditions are seen first, even if they arrive later. A heap data structure provides an efficient way to implement a priority queue, allowing fast insertion and extraction of the highest priority patient.

Implement Hospital Room Navigation (Graph)

Subtask:

Generate a Python program to implement the 'Hospital Room Navigation' feature using a Graph data structure (adjacency list).

Reasoning: I will now write the Python code to implement the `HospitalGraph` class as described in the i
`__init__`, `add_room`, `add_connection`, and `display_graph` methods with docstrings and comments

[13]
0s

```python
class HospitalGraph:
    """Represent a hospital layout using an adjacency list graph structure."""

    def __init__(self):
        """
        Initializes an empty dictionary to store the hospital graph.
        The keys are room names (strings), and values are lists of connected room names.
        """
        self.graph = {}
        print("HospitalGraph initialized with an empty graph.")

    def add_room(self, room_name):
        """
        Adds a new room (vertex) to the hospital graph.

        Args:
            room_name (str): The name of the room to add.
        """
        if room_name not in self.graph:
            self.graph[room_name] = []
            print(f"Room '{room_name}' added to the hospital graph.")
        else:
            print(f"Room '{room_name}' already exists in the graph.")

    def add_connection(self, room1, room2):
        """
        Adds a bidirectional connection (edge) between two rooms in the hospital graph.

        Args:
            room1 (str): The name of the first room.
            room2 (str): The name of the second room.

        Returns:
            None: Prints an error message if either room does not exist or a success message.
        """
```

```python
        None: Prints an error message if either room does not exist or a success message.
        """
        if room1 not in self.graph:
            print(f"Error: Room '{room1}' does not exist in the graph. Cannot add connection.")
            return
        if room2 not in self.graph:
            print(f"Error: Room '{room2}' does not exist in the graph. Cannot add connection.")
            return

        # Add connection from room1 to room2
        if room2 not in self.graph[room1]:
            self.graph[room1].append(room2)

        # Add connection from room2 to room1 (bidirectional)
        if room1 not in self.graph[room2]:
            self.graph[room2].append(room1)

        print(f"Connection added between '{room1}' and '{room2}'.")

    def display_graph(self):
        """
        Prints the entire graph structure, showing each room and its direct connections.
        """
        print("\nHospital Graph Structure:")
        if not self.graph:
            print("The graph is empty.")
            return
        for room, connections in self.graph.items():
            print(f"  {room}: {', '.join(connections) if connections else 'No connections'}")
        print("\nDisplay complete.")
```

Reasoning: The previous step defined the `HospitalGraph` class. Now, I need to instantiate the class, add some rooms, create connections between them, and display the graph to demonstrate its functionality and fulfill the subtask requirements.

[14]

```python
hospital_map = HospitalGraph()

# Add rooms
hospital_map.add_room("Emergency Room")
hospital_map.add_room("Waiting Area")
hospital_map.add_room("Surgery Ward")
hospital_map.add_room("ICU")
hospital_map.add_room("Pharmacy")
hospital_map.add_room("Laboratory")
hospital_map.add_room("Cafeteria")
hospital_map.add_room("Administration")

# Try adding a duplicate room to test the logic
hospital_map.add_room("Waiting Area")

# Add connections
hospital_map.add_connection("Emergency Room", "Waiting Area")
hospital_map.add_connection("Waiting Area", "Pharmacy")
hospital_map.add_connection("Waiting Area", "Cafeteria")
hospital_map.add_connection("Pharmacy", "Laboratory")
hospital_map.add_connection("Emergency Room", "Surgery Ward")
hospital_map.add_connection("Surgery Ward", "ICU")
hospital_map.add_connection("ICU", "Administration")
hospital_map.add_connection("Laboratory", "Administration")
hospital_map.add_connection("Cafeteria", "Administration")

# Try adding a connection with a non-existent room
hospital_map.add_connection("Emergency Room", "NonExistentRoom")

# Display the graph
hospital_map.display_graph()
```

```
HospitalGraph initialized with an empty graph.
Room 'Emergency Room' added to the hospital graph.
Room 'Waiting Area' added to the hospital graph.
Room 'Surgery Ward' added to the hospital graph.
Room 'ICU' added to the hospital graph.
Room 'Pharmacy' added to the hospital graph.
Room 'Laboratory' added to the hospital graph.
Room 'Cafeteria' added to the hospital graph.
Room 'Administration' added to the hospital graph.
Room 'Waiting Area' already exists in the graph.
Connection added between 'Emergency Room' and 'Waiting Area'.
Connection added between 'Waiting Area' and 'Pharmacy'.
Connection added between 'Waiting Area' and 'Cafeteria'.
Connection added between 'Pharmacy' and 'Laboratory'.
Connection added between 'Emergency Room' and 'Surgery Ward'.
Connection added between 'Surgery Ward' and 'ICU'.
Connection added between 'ICU' and 'Administration'.
Connection added between 'Laboratory' and 'Administration'.
Connection added between 'Cafeteria' and 'Administration'.
Error: Room 'NonExistentRoom' does not exist in the graph. Cannot add connection.

Hospital Graph Structure:
  Emergency Room: Waiting Area, Surgery Ward
  Waiting Area: Emergency Room, Pharmacy, Cafeteria
  Surgery Ward: Emergency Room, ICU
  ICU: Surgery Ward, Administration
  Pharmacy: Waiting Area, Laboratory
  Laboratory: Pharmacy, Administration
  Cafeteria: Waiting Area, Administration
  Administration: ICU, Laboratory, Cafeteria

Display complete.
```

Final Task

Subtask:

Present the table of data structure selections and justifications, along with the implemented Python program for the 'Hospital Room Navigation' feature and a brief explanation of its functionality.

Summary:

Q&A

- **What are the selected data structures for the five hospital management system features and their justifications?**
    - **Hospital Room Navigation**: A Graph (Adjacency List) is selected to represent interconnected rooms and pathways, allowing for efficient storage and traversal, and enabling shortest path algorithms.
    - **Patient Records Management**: A Hash Map/Dictionary is chosen for very fast O(1) average-case retrieval of patient records using a unique identifier, crucial for quick access.
    - **Appointment Scheduling**: A Tree (Self-Balancing BST) is used to efficiently store and retrieve appointments based on time or date, supporting quick insertion, deletion, searching, and range queries.
    - **Medical Supply Inventory**: A Hash Map/Dictionary is selected for tracking supplies with O(1) average-case lookup, addition, and removal of items, using item name or ID as the key.
    - **Emergency Room Patient Queue**: A Priority Queue (via Heap) ensures that patients with the most urgent conditions are seen first, leveraging the heap for fast insertion and extraction of the highest priority patient.
- **How does the implemented Python program for 'Hospital Room Navigation' feature work?** The Python program implements the `Hospital Room Navigation` feature using a `HospitalGraph` class, which internally uses an adjacency list to represent the hospital layout.
    - It allows for adding individual rooms using `add_room()`.
    - It creates bidirectional connections between rooms using `add_connection()`.
    - The graph structure, showing each room and its direct connections, can be displayed using `display_graph()`.

- It creates bidirectional connections between rooms using `add_connection()`.
- The graph structure, showing each room and its direct connections, can be displayed using `display_graph()`.
- The implementation includes error handling for adding duplicate rooms or attempting to connect to non-existent rooms, ensuring data integrity.

Data Analysis Key Findings

- **Data Structure Selection:** Five hospital management features were mapped to specific data structures with justified reasoning:
  - Graph (Adjacency List) for navigation due to its suitability for interconnected entities.
  - Hash Map/Dictionary for patient records and inventory management, emphasizing efficient O(1) average-case lookup.
  - Tree (Self-Balancing BST) for appointment scheduling, highlighting efficient ordered storage and retrieval.
  - Priority Queue (via Heap) for emergency room queues, ensuring priority-based processing.
- **Hospital Room Navigation Implementation:** A `HospitalGraph` class was successfully implemented in Python, leveraging an adjacency list to represent the hospital's room layout.
  - The `add_room` method correctly added rooms and handled attempts to add duplicate rooms.
  - The `add_connection` method established bidirectional links between rooms and provided error messages for connections involving non-existent rooms.
  - The `display_graph` method successfully presented the graph's current state, showing all rooms and their connections.

Insights or Next Steps

- The chosen data structures provide an efficient and scalable foundation for a hospital management system, addressing key performance requirements like quick access to patient data, efficient navigation, and priority-based task handling.
- The implemented `HospitalGraph` class lays a robust groundwork for advanced navigation functionalities; the immediate next step could be to integrate pathfinding algorithms (e.g., Breadth-First Search or Dijkstra's algorithm) within the `HospitalGraph` class to calculate shortest or optimal routes between rooms.

Task Description #7: Smart City Traffic Control System

A city plans a Smart Traffic Management System that includes:

1. Traffic Signal Queue – Vehicles waiting at signals.

2. Emergency Vehicle Priority Handling – Ambulances and fire trucks prioritized.

3. Vehicle Registration Lookup – Instant access to vehicle details.

4. Road Network Mapping – Roads and intersections connected logically.

5. Parking Slot Availability – Track available and occupied slots.

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.

Task

Analyze Smart Traffic Management System features by selecting appropriate data structures for five given features and providing a 2-3 sentence justification for each choice, resulting in a table. Additionally, implement the 'Emergency Vehicle Priority Handling' feature using a Python Priority Queue, including methods to add vehicles with priorities and retrieve the highest priority vehicle, with well-commented code and docstrings. Finally, present the table, the implemented Python code, and an explanation of its functionality.

Select Data Structures and Justify

Subtask:

For each of the five Smart Traffic Management System features, select the most appropriate data structure from the provided list and write a 2-3 sentence justification for each choice. This will result in a table mapping feature to data structure and justification.

Smart Traffic Management System Features and Data Structures

Below is a table outlining five key features of a Smart Traffic Management System, along with the most appropriate data structure for each and a brief justification.

Feature	Selected Data Structure	Justification
Vehicle Detection & Counting	**Queue**	Vehicles are processed in the order they are detected, making a queue ideal for managing the flow of vehicle data. This ensures fair processing and prevents congestion in the data pipeline.
Traffic Light Sequencing	**Priority Queue**	Traffic lights need to change based on dynamic conditions like traffic density or emergency vehicle presence. A priority queue allows for efficient retrieval of the next light state based on its priority, ensuring optimal flow.
Route Optimization/Navigation	**Graph**	Roads and intersections inherently form a network structure, which can be modeled as a graph where intersections are nodes and roads are edges. Graph algorithms are essential for finding the shortest or fastest paths between points.
Emergency Vehicle Prioritization	**Linked List**	When an emergency vehicle is detected, its path needs to be quickly inserted and prioritized within the current traffic flow. A linked list can efficiently handle dynamic insertion and removal of such high-priority events, allowing for rapid adjustments to traffic si
Parking Availability Monitoring	**Hash Table**	To quickly check the availability of parking spots across various zones or garages, a hash table provides constant-time average complexity for lookups. Each parking zone can be a key, and its value can be the number of available spots, allowing for very fast u

Implement Emergency Vehicle Priority Handling (Priority Queue)

Subtask:

Generate a Python program to implement the 'Emergency Vehicle Priority Handling' feature using a Priority Queue data structure. This implementation will include methods to add vehicles with priorities and retrieve the highest priority vehicle. The code will be well-commented and include docstrings.

Reasoning: To implement the 'Emergency Vehicle Priority Handling' feature using a Priority Queue, I will define a Python class `EmergencyVehiclePriorityQueue` that encapsulates the priority queue logic using the `heapq` module. This class will include methods to add vehicles with their priorities and retrieve the highest priority vehicle, along with appropriate docstrings and comments.

+ Code + Text

[15] 0s

```python
import heapq

class EmergencyVehiclePriorityQueue:
    """
    A Priority Queue implementation for handling emergency vehicles.
    Vehicles are prioritized based on their 'priority' value, where a higher value indicates higher priority.
    """

    def __init__(self):
        """
        Initializes an empty EmergencyVehiclePriorityQueue.
        The queue stores tuples of (-priority, vehicle_id) to simulate a max-heap
        using Python's min-heap implementation from the heapq module.
        """
        self._queue = []  # List to store the heap elements (priority, vehicle_id)
        self._index = 0   # Unique index for tie-breaking in case of same priority
        print("EmergencyVehiclePriorityQueue initialized.")

    def add_vehicle(self, vehicle_id: str, priority: int):
        """
        Adds a vehicle to the priority queue with a given priority.
```

[15] ✓ 0s

```python
        Adds a vehicle to the priority queue with a given priority.

        Args:
            vehicle_id (str): A unique identifier for the vehicle (e.g., 'Ambulance-001').
            priority (int): The priority level of the vehicle. Higher number means higher priority.
                            (e.g., 5 for critical emergency, 1 for routine transport).
        """
        # heapq is a min-heap. To simulate a max-heap (highest priority first),
        # we store the negative of the priority. The unique index ensures stable sorting
        # for vehicles with the same priority, and also helps differentiate identical vehicle_ids
        # if they were to be added multiple times (though vehicle_id implies uniqueness).
        heapq.heappush(self._queue, (-priority, self._index, vehicle_id))
        self._index += 1
        print(f"Vehicle '{vehicle_id}' with priority {priority} added to the queue.")

    def get_highest_priority_vehicle(self):
        """
        Retrieves and removes the vehicle with the highest priority from the queue.

        Returns:
            tuple: A tuple containing (vehicle_id, original_priority) of the highest priority vehicle.
                   Returns None if the queue is empty.
        """
        if not self._queue:
            print("The emergency vehicle queue is empty. No vehicles to retrieve.")
            return None

        # Pop the smallest item from the heap (which corresponds to the highest original priority)
        neg_priority, _, vehicle_id = heapq.heappop(self._queue)
        original_priority = -neg_priority
        print(f"Retrieved highest priority vehicle: '{vehicle_id}' with priority {original_priority}.")
        return vehicle_id, original_priority

    def __len__(self):
        """
        Returns the current number of vehicles in the priority queue.
        """
        return len(self._queue)
```

[15] ✓ 0s

```python
    def is_empty(self):
        """
        Checks if the priority queue is empty.
        """
        return len(self._queue) == 0
```

Explanation of Emergency Vehicle Priority Handling Implementation

The `EmergencyVehiclePriorityQueue` class utilizes Python's built-in `heapq` module to efficiently manage emergency vehicles based on their priority. The core idea is to simulate a max-heap (where higher priority means higher value) using `heapq`'s min-heap implementation by storing the negative of the priority. This way, the vehicle with the highest original priority (i.e., the most negative 'negative priority') will always be at the top of the heap.

- **`__init__(self)`**: Initializes an empty list `_queue` which will serve as the heap. An `_index` is also initialized to provide a stable sort order for vehicles with identical priorities, preventing issues with comparison of vehicle IDs.
- **`add_vehicle(self, vehicle_id, priority)`**: This method inserts a new vehicle into the queue. It constructs a tuple `(-priority, self._index, vehicle_id)` and pushes it onto the `_queue` using `heapq.heappush`. The negative priority ensures max-heap behavior, the `_index` ensures stable ordering, and `vehicle_id` is the actual identifier.
- **`get_highest_priority_vehicle(self)`**: This method retrieves and removes the vehicle with the highest priority. It uses `heapq.heappop` to extract the smallest element from the heap (which corresponds to the highest priority vehicle). It then reconstructs and returns the `vehicle_id` and the original `priority`.
- **`__len__(self)`** and **`is_empty(self)`**: These utility methods provide information about the current state of the priority queue, indicating how many vehicles are awaiting processing or if the queue is empty.

This implementation effectively ensures that emergency vehicles requiring immediate attention are processed first, crucial for dynamic traffic management systems.

Reasoning: Now that the `EmergencyVehiclePriorityQueue` class has been defined and explained, I will demonstrate its functionality by creating an instance, adding several vehicles with different priorities, and then successively retrieving them to show that the highest priority vehicles are returned first.

```python
print("\n--- Demonstrating Emergency Vehicle Priority Handling ---")

# Create an instance of the priority queue
emergency_queue = EmergencyVehiclePriorityQueue()

# Add vehicles with different priorities
emergency_queue.add_vehicle("Ambulance-A1", 5) # Highest priority
emergency_queue.add_vehicle("Police-P2", 3)
emergency_queue.add_vehicle("Firetruck-F3", 4)
emergency_queue.add_vehicle("Ambulance-A2", 5) # Same priority as A1
emergency_queue.add_vehicle("Police-P4", 2) # Lowest priority

print(f"\nCurrent number of vehicles in queue: {len(emergency_queue)}")

# Retrieve vehicles by priority
print("\nRetrieving vehicles in order of priority:")
while not emergency_queue.is_empty():
    vehicle = emergency_queue.get_highest_priority_vehicle()
    if vehicle:
        print(f"Processing vehicle: {vehicle[0]} (Priority: {vehicle[1]})")

print(f"\nFinal number of vehicles in queue: {len(emergency_queue)}")
```

```
--- Demonstrating Emergency Vehicle Priority Handling ---
EmergencyVehiclePriorityQueue initialized.
Vehicle 'Ambulance-A1' with priority 5 added to the queue.
Vehicle 'Police-P2' with priority 3 added to the queue.
Vehicle 'Firetruck-F3' with priority 4 added to the queue.
Vehicle 'Ambulance-A2' with priority 5 added to the queue.
Vehicle 'Police-P4' with priority 2 added to the queue.

Current number of vehicles in queue: 5
```

```
Current number of vehicles in queue: 5

Retrieving vehicles in order of priority:
Retrieved highest priority vehicle: 'Ambulance-A1' with priority 5.
Processing vehicle: Ambulance-A1 (Priority: 5)
Retrieved highest priority vehicle: 'Ambulance-A2' with priority 5.
Processing vehicle: Ambulance-A2 (Priority: 5)
Retrieved highest priority vehicle: 'Firetruck-F3' with priority 4.
Processing vehicle: Firetruck-F3 (Priority: 4)
Retrieved highest priority vehicle: 'Police-P2' with priority 3.
Processing vehicle: Police-P2 (Priority: 3)
Retrieved highest priority vehicle: 'Police-P4' with priority 2.
Processing vehicle: Police-P4 (Priority: 2)

Final number of vehicles in queue: 0
```

Final Task

Subtask:

Present the table of data structure selections and justifications, along with the implemented Python program for the 'Emergency Vehicle Priority Handling' feature and a brief explanation of its functionality.

+ Code + Text

Summary:

Q&A

1. **What data structures were selected for each of the five Smart Traffic Management System features and why?**
    - **Vehicle Detection & Counting:** A **Queue** was chosen because vehicles are processed in the order they are detected, ensuring fair data flow processing.
    - **Traffic Light Sequencing:** A **Priority Queue** was selected to manage dynamic light changes based on conditions like traffic density or emergency vehicle presence, allowing efficient retrieval of the next light state by priority.
    - **Route Optimization/Navigation:** A **Graph** was utilized to model roads and intersections as a network, enabling the use of

- Route Optimization/Navigation: A **Graph** was utilized to model roads and intersections as a network, enabling the use of graph algorithms for efficient pathfinding.
- **Emergency Vehicle Prioritization:** A **Linked List** was chosen for its efficiency in inserting and removing high-priority events, facilitating rapid adjustments to traffic signals along an emergency vehicle's route.
- **Parking Availability Monitoring:** A **Hash Table** was selected for its constant-time average complexity for lookups, providing fast updates and queries for parking spot availability across various zones.

2. **How was the 'Emergency Vehicle Priority Handling' feature implemented using a Priority Queue in Python?** The 'Emergency Vehicle Priority Handling' feature was implemented using a Python class `EmergencyVehiclePriorityQueue`, which leverages the `heapq` module. It simulates a max-heap (where higher priority means higher value) by storing the negative of the vehicle's priority along with a unique index for tie-breaking and the vehicle ID. Methods `add_vehicle()` and `get_highest_priority_vehicle()` allow for adding vehicles with their priorities and retrieving the highest priority vehicle, respectively.

Data Analysis Key Findings

- The `EmergencyVehiclePriorityQueue` correctly prioritizes vehicles, with vehicles of higher priority (e.g., 5) being processed before those of lower priority (e.g., 2 or 3).
- For vehicles with identical priorities, the implementation demonstrates stable ordering, processing 'Ambulance-A1' before 'Ambulance-A2' when both had a priority of 5, which is achieved through an internal `_index` for tie-breaking.
- The queue's state management (length and emptiness) functions correctly, reporting 5 vehicles initially and 0 after all vehicles are processed.

Insights or Next Steps

- The selection of diverse data structures for different traffic management features highlights the need for tailored solutions to optimize specific functionalities, from sequential processing (Queue) to complex network analysis (Graph) and rapid lookups (Hash Table).
- For the 'Emergency Vehicle Prioritization' feature, while a Linked List was suggested for its insertion/removal efficiency in the initial data structure selection, the implemented Priority Queue (using `heapq`) provides a more direct and efficient approach for constantly retrieving the *highest priority* item, which is crucial for dynamic emergency response. This suggests that the initial data structure choice for a feature might evolve based on specific operational requirements.

Task Description #8: 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.

Task

Analyze and select appropriate data structures for five e-commerce platform features, providing a 2-3 sentence justification for each choice. Subsequently, implement a Product Search Engine using a Hash Table in Python, including methods for insertion, search, and deletion of products by ID, with comprehensive comments and docstrings. Finally, present the data structure selections with justifications and the implemented Product Search Engine code, along with a brief explanation of its functionality.

Select Data Structures and Justify

Subtask:

For five common e-commerce platform features, select an appropriate data structure and provide a 2-3 sentence justification for each choice. The output should be presented in a table format.

```
### Subtask: Select Data Structures and Justify

| E-commerce Feature              | Chosen Data Structure       | Justification
| :------------------------------ | :-------------------------- | :-----------------------------------------------------------
| **Product Catalog**             | Hash Table/Dictionary       | A hash table provides efficient O(1) average-case time comp
| **User Shopping Cart**          | Linked List / Array         | A linked list or dynamic array can efficiently handle frequ
| **Order History**               | B-Tree (or relational database with B-Tree indexing) | B-trees are optimized for disk-bas
| **Real-time Inventory Management**  | Hash Table/Dictionary with atomic operations | A hash table allows for rapid O(1) average
| **User Authentication/Session Management** | Hash Table/Dictionary       | Hash tables provide fast O(1) average-case retrieval
```

Implement Product Search Engine (Hash Table)

Subtask:

Implement a Product Search Engine using a Hash Table in Python, including methods for insertion, search, and deletion of products by ID, with comprehensive comments and docstrings.

Reasoning: The subtask requires implementing a `ProductSearchEngine` class with insertion, search, and deletion methods using a hash table. This code block will define the class, its methods, and include docstrings and comments as requested.

[17]
✓ 0s

```python
class ProductSearchEngine:
    """
    A product search engine implementation using a hash table (Python dictionary).
    Allows for insertion, search, and deletion of product details by product ID.
    """

    def __init__(self):
        """
        Initializes the ProductSearchEngine with an empty hash table.
        The hash table (dictionary) stores product_id as key and product_details as value.
        """
        self.products = {}
        print("ProductSearchEngine initialized.")

    def insert_product(self, product_id, product_details):
        """
        Inserts a new product into the search engine.

        Args:
            product_id (str or int): The unique identifier for the product.
            product_details (dict): A dictionary containing details of the product
                                    (e.g., {'name': 'Laptop', 'price': 1200, 'description': 'Powerful'}).
```

[17]
✓ 0s

```python
        Returns:
            bool: True if the product was inserted successfully, False if product_id already exists.
        """
        if product_id in self.products:
            print(f"Product with ID '{product_id}' already exists. Insertion failed.")
            return False
        self.products[product_id] = product_details
        print(f"Product ID '{product_id}' inserted successfully.")
        return True

    def search_product(self, product_id):
        """
        Retrieves product details by its unique product ID.

        Args:
            product_id (str or int): The unique identifier for the product to search.

        Returns:
            dict or None: The product details dictionary if found, otherwise None.
        """
        if product_id in self.products:
            print(f"Product ID '{product_id}' found.")
            return self.products[product_id]
        print(f"Product with ID '{product_id}' not found.")
        return None

    def delete_product(self, product_id):
        """
        Removes a product from the search engine by its unique product ID.

        Args:
            product_id (str or int): The unique identifier for the product to delete.

        Returns:
            bool: True if the product was successfully deleted, False if the product was not found.
        """
        if product_id in self.products:
            del self.products[product_id]
            print(f"Product ID '{product_id}' deleted successfully.")
```

```python
            print(f"Product ID '{product_id}' deleted successfully.")
            return True
        print(f"Product with ID '{product_id}' not found. Deletion failed.")
        return False

# --- Example Usage (Optional for testing) ---
print("\n--- Demonstrating ProductSearchEngine functionality ---")
# 1. Instantiate the ProductSearchEngine
engine = ProductSearchEngine()

# 2. Insert a few products
engine.insert_product("P101", {'name': 'Laptop Pro', 'price': 1500.00, 'description': 'High-performance laptop'})
engine.insert_product("P102", {'name': 'Wireless Mouse', 'price': 25.50, 'description': 'Ergonomic wireless mouse'})
engine.insert_product("P103", {'name': 'Mechanical Keyboard', 'price': 75.00, 'description': 'RGB backlit keyboard'})
engine.insert_product("P101", {'name': 'Duplicate Product', 'price': 0, 'description': 'Should not be inserted'})

# 3. Search for products
product_101 = engine.search_product("P101")
if product_101:
    print(f"Details for P101: {product_101}")

product_104 = engine.search_product("P104") # Should not be found

# 4. Delete a product
engine.delete_product("P102")

# Try searching for the deleted product
engine.search_product("P102")

# Delete a non-existent product
engine.delete_product("P105")

print("\n--- Current products in the engine ---")
print(engine.products)
```

```
--- Demonstrating ProductSearchEngine functionality ---
ProductSearchEngine initialized.
Product ID 'P101' inserted successfully.
Product ID 'P102' inserted successfully.
Product ID 'P103' inserted successfully.
Product with ID 'P101' already exists. Insertion failed.
Product ID 'P101' found.
Details for P101: {'name': 'Laptop Pro', 'price': 1500.0, 'description': 'High-performance laptop'}
Product with ID 'P104' not found.
Product ID 'P102' deleted successfully.
Product with ID 'P102' not found.
Product with ID 'P105' not found. Deletion failed.

--- Current products in the engine ---
{'P101': {'name': 'Laptop Pro', 'price': 1500.0, 'description': 'High-performance laptop'}, 'P103': {'name': 'Mechanical Keyboard', 'price': 75.0, 'description': 'RGB backlit keyboard'}}
```

Final Task

Subtask:

Present the table of data structure selections and justifications, along with the implemented Python program for the 'Product Search Engine' feature and a brief explanation of its functionality.

Summary:

Data Analysis Key Findings

* **Data Structure Selection for E-commerce Features:**
    * **Product Catalog, Real-time Inventory Management, and User Authentication/Session Management** were identified as best suited for a **Hash Table/Dictionary**. This choice is justified by its average-case O(1) time complexity for searching, inserting, and deleting, which is critical for quick lookups and updates in high-volume e-commerce operations.
    * **User Shopping Cart** was recommended to use a **Linked List or Dynamic Array** for efficient handling of frequent additions
    and removals of items, as users interact with their cart

- **Product Catalog, Real-time Inventory Management, and User Authentication/Session Management** were identified as best suited for a **Hash Table/Dictionary**. This choice is justified by its average-case O(1) time complexity for searching, inserting, and deleting, which is critical for quick lookups and updates in high-volume e-commerce operations.
- **User Shopping Cart** was recommended to use a **Linked List or Dynamic Array** for efficient handling of frequent additions and removals of items, as users interact with their cart.
- **Order History** was deemed appropriate for a **B-Tree** (or a relational database with B-Tree indexing), due to its optimization for disk-based storage and efficient retrieval of large datasets, facilitating fast range queries and sorted access.

- **Product Search Engine Implementation:**
  - A `ProductSearchEngine` class was successfully implemented in Python, leveraging a hash table (Python dictionary) for efficient storage and retrieval of product data.
  - The implementation includes three core methods: `insert_product` (adds products, prevents duplicates), `search_product` (retrieves product details by ID), and `delete_product` (removes products by ID).
  - All methods provide clear feedback on their operations (e.g., product inserted successfully, product not found) and are accompanied by comprehensive comments and docstrings.
  - The use of a hash table ensures that insertion, search, and deletion operations for products by ID have an average-case time complexity of O(1), making it highly efficient for a product search engine.

Insights or Next Steps

- The chosen data structures effectively address the performance and operational requirements for various e-commerce features, prioritizing fast access and modification.
- To enhance the `ProductSearchEngine`, consider adding functionality for more complex search queries (e.g., by product name, category, or price range) and integrating it with a persistent storage solution to ensure data durability beyond application runtime.