Lab 11: Data Structures with AI

AI-Assisted Coding - Implementing Fundamental Structures

Course Code: 23CS002PC304 2303a54028

Academic Year: 2025-2026 batch_47a

Lab Objectives

• Use AI to assist in designing and implementing fundamental data structures in Python

• Learn how to prompt AI for structure creation, optimization, and documentation

• Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables

• Enhance code quality with AI-generated comments and performance suggestions

Task 1: Stack Implementation

AI PROMPT

Create a Python Stack class with the following methods: - push(item): Add an item to the top - pop(): Remove and return the top item - peek(): View the top item without removing - is_empty(): Check if stack is empty Include comprehensive docstrings for each method.

INPUT CODE

```
class Stack:
    """A simple stack implementation using a list."""


    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 from the stack."""
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.items.pop()


    def peek(self):
        """Return the top item without removing it."""
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.items[-1]


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


    def __str__(self):
        """String representation of the stack."""
        return f"Stack({self.items})"



# Demo Code
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
print(f"Stack after pushes: {stack}")
print(f"Peek: {stack.peek()}")
```

```
print(f"Pop: {stack.pop()}")
print(f"Stack after pop: {stack}")
print(f"Is empty: {stack.is_empty()}")
```

OUTPUT

```
Stack after pushes: Stack([10, 20, 30])
Peek: 30
Pop: 30
Stack after pop: Stack([10, 20])
Is empty: False
```

Task 2: Queue Implementation

AI PROMPT

Create a Python Queue class with the following methods: - enqueue(item): Add an item to the rear - dequeue(): Remove and return the front item - peek(): View the front item without removing - size(): Return the number of items Include comprehensive docstrings for each method.

INPUT CODE

```
class Queue:
    """A FIFO queue implementation using a list."""


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


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


    def dequeue(self):
        """Remove and return the front item from the queue."""
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.items.pop(0)


    def peek(self):
        """Return the front item without removing it."""
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.items[0]


    def size(self):
        """Return the number of items in the queue."""
        return len(self.items)


    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.items) == 0


    def __str__(self):
        """String representation of the queue."""
        return f"Queue({self.items})"



# Demo Code
queue = Queue()
queue.enqueue("Customer 1")
```

```
queue.enqueue("Customer 2")
queue.enqueue("Customer 3")
print(f"Queue after enqueues: {queue}")
print(f"Peek: {queue.peek()}")
print(f"Dequeue: {queue.dequeue()}")
print(f"Queue after dequeue: {queue}")
print(f"Size: {queue.size()}")
```

OUTPUT

```
Queue after enqueues: Queue(['Customer 1', 'Customer 2', 'Customer 3'])
Peek: Customer 1
Dequeue: Customer 1
Queue after dequeue: Queue(['Customer 2', 'Customer 3'])
Size: 2
```

Task 3: Linked List Implementation

AI PROMPT

Create a Python Singly Linked List with: - Node class with data and next pointer - insert(data): Add a node at the end - display(): Print all elements - insert_at_beginning(data): Add a node at the start Include comprehensive docstrings for each method.

INPUT CODE

```
class Node:
    """A node in a singly linked list."""

    def __init__(self, data):
        """Initialize a node with data."""
        self.data = data
        self.next = None


class LinkedList:
    """A singly linked list implementation."""

    def __init__(self):
        """Initialize an 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

        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def insert_at_beginning(self, data):
        """Insert a new node at the beginning of the list."""
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        """Display all elements in the list."""
        if not self.head:
            return "List is empty"
```

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



# Demo Code
ll = LinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)
print(f"Linked List: {ll.display()}")
ll.insert_at_beginning(5)
print(f"After inserting 5 at beginning: {ll.display()}")
```

OUTPUT

```
Linked List: 10 -> 20 -> 30
After inserting 5 at beginning: 5 -> 10 -> 20 -> 30
```

Task 4: Binary Search Tree (BST)

AI PROMPT

Create a Python Binary Search Tree with: - TreeNode class with value, left, and right pointers - insert(value): Add a value to the BST - inorder_traversal(): Return in-order traversal Include comprehensive docstrings and recursive implementation.

INPUT CODE

```
class TreeNode:
    """A node in a binary search tree."""

    def __init__(self, value):
        """Initialize a tree node with a value."""
        self.value = value
        self.left = None
        self.right = None



class BST:
    """A binary search tree implementation."""

    def __init__(self):
        """Initialize an empty BST."""
        self.root = None

    def insert(self, value):
        """Insert a value into the BST."""
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        """Helper method for recursive insertion."""
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def inorder_traversal(self):
        """Return in-order traversal of the BST."""
```

```
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        """Helper method for in-order traversal."""
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)



# Demo Code
bst = BST()
values = [50, 30, 70, 20, 40, 60, 80]
for val in values:
    bst.insert(val)
print(f"Inserted values: {values}")
print(f"In-order traversal: {bst.inorder_traversal()}")
```

OUTPUT

```
Inserted values: [50, 30, 70, 20, 40, 60, 80]
In-order traversal: [20, 30, 40, 50, 60, 70, 80]
```

Task 5: Hash Table Implementation

AI PROMPT

Create a Python Hash Table with: - insert(key, value): Add a key-value pair - search(key): Find a value by key - delete(key): Remove a key-value pair - Collision handling using chaining (linked lists) Include comprehensive docstrings.

INPUT CODE

```
class HashTable:
    """A hash table implementation with chaining for collision handling."""

    def __init__(self, size=10):
        """Initialize a hash table with a given size."""
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """Hash function to compute index for a key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash(key)
        # Check if key exists and update
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        # Add new key-value pair
        self.table[index].append((key, value))

    def search(self, key):
        """Search for a value by key."""
        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 from the hash table."""
        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 hash table contents."""
        items = []
        for i, bucket in enumerate(self.table):
            if bucket:
                items.append(f"Bucket {i}: {bucket}")
        return "\n".join(items)



# Demo Code
ht = HashTable()
ht.insert("apple", 100)
ht.insert("banana", 200)
ht.insert("cherry", 300)
print("Hash Table Contents:")
print(ht.display())
print(f"\nSearch 'banana': {ht.search('banana')}")
ht.delete("banana")
print("\nAfter deleting 'banana':")
print(ht.display())
```

OUTPUT

```
Hash Table Contents:
Bucket 3: [('apple', 100)]
Bucket 5: [('banana', 200)]
Bucket 7: [('cherry', 300)]

Search 'banana': 200

After deleting 'banana':
Bucket 3: [('apple', 100)]
Bucket 7: [('cherry', 300)]
```

Task 6: Graph Representation

AI PROMPT

Create a Python Graph with: - add_vertex(vertex): Add a vertex - add_edge(v1, v2): Add an edge between vertices - display(): Show all connections Include comprehensive docstrings.

INPUT CODE

```
class Graph:
    """A graph implementation using adjacency list."""

    def __init__(self):
        """Initialize an empty graph."""
        self.graph = {}

    def add_vertex(self, vertex):
        """Add a vertex to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, v1, v2):
        """Add an edge between two vertices (undirected graph)."""
        # Add vertices if they don't exist
        self.add_vertex(v1)
        self.add_vertex(v2)
        # Add edge (undirected)
        if v2 not in self.graph[v1]:
            self.graph[v1].append(v2)
        if v1 not in self.graph[v2]:
            self.graph[v2].append(v1)

    def display(self):
        """Display all vertices and their connections."""
        result = []
        for vertex in self.graph:
            connections = ", ".join(map(str, self.graph[vertex]))
            result.append(f"{vertex} -> [{connections}]")
        return "\n".join(result)


# Demo Code
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "D")
g.add_edge("C", "D")
g.add_edge("D", "E")
print("Graph Connections:")
print(g.display())
```

OUTPUT

```
Graph Connections:
A -> [B, C]
B -> [A, D]
C -> [A, D]
D -> [B, C, E]
E -> [D]
```

Task 7: Priority Queue

AI PROMPT

Create a Python Priority Queue with: - enqueue(item, priority): Add item with priority - dequeue(): Remove and return highest priority item - display(): Show all items Use heapq module for efficient operations.

INPUT CODE

```
import heapq

class PriorityQueue:
    """A priority queue implementation using heapq."""

    def __init__(self):
        """Initialize an empty priority queue."""
        self.heap = []
        self.counter = 0  # For tie-breaking

    def enqueue(self, item, priority):
        """Add an item with a priority (lower number = higher priority)."""
        heapq.heappush(self.heap, (priority, self.counter, item))
        self.counter += 1

    def dequeue(self):
        """Remove and return the highest priority item."""
        if self.is_empty():
            raise IndexError("Priority queue is empty")
        priority, _, item = heapq.heappop(self.heap)
        return item, priority

    def is_empty(self):
        """Check if the priority queue is empty."""
        return len(self.heap) == 0

    def display(self):
        """Display all items in the priority queue."""
        items = [(item, priority) for priority, _, item in sorted(self.heap)]
        result = []
        for item, priority in items:
            result.append(f"Priority {priority}: {item}")
        return "\n".join(result)


# Demo Code
pq = PriorityQueue()
pq.enqueue("Task A", 3)
pq.enqueue("Task B", 1)
```

```
pq.enqueue("Task C", 2)
print("Priority Queue:")
print(pq.display())
print(f"\nDequeue: {pq.dequeue()}")
print(f"Dequeue: {pq.dequeue()}")
```

OUTPUT

```
Priority Queue:
Priority 1: Task B
Priority 2: Task C
Priority 3: Task A


Dequeue: ('Task B', 1)
Dequeue: ('Task C', 2)
```

Task 8: Deque Implementation

AI PROMPT

Create a Python Deque class with: - add_front(item): Add item to front - add_rear(item): Add item to rear - remove_front(): Remove from front - remove_rear(): Remove from rear Use collections.deque for efficient operations.

INPUT CODE

```
from collections import deque

class DequeDS:
    """A double-ended queue implementation using collections.deque."""

    def __init__(self):
        """Initialize an empty deque."""
        self.deque = deque()

    def add_front(self, item):
        """Add an item to the front of the deque."""
        self.deque.appendleft(item)

    def add_rear(self, item):
        """Add an item to the rear of the deque."""
        self.deque.append(item)

    def remove_front(self):
        """Remove and return an item from the front."""
        if self.is_empty():
            raise IndexError("Deque is empty")
        return self.deque.popleft()

    def remove_rear(self):
        """Remove and return an item from the rear."""
        if self.is_empty():
            raise IndexError("Deque is empty")
        return self.deque.pop()

    def is_empty(self):
        """Check if the deque is empty."""
        return len(self.deque) == 0

    def __str__(self):
        """String representation of the deque."""
        return f"Deque({list(self.deque)})"
```

```
# Demo Code
dq = DequeDS()
dq.add_rear(10)
dq.add_rear(20)
dq.add_front(5)
print(f"Deque: {dq}")
print(f"Remove front: {dq.remove_front()}")
print(f"Remove rear: {dq.remove_rear()}")
print(f"Deque after removals: {dq}")
```

OUTPUT

```
Deque: Deque([5, 10, 20])
Remove front: 5
Remove rear: 20
Deque after removals: Deque([10])
```

Task 9: Campus Resource Management System

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

Recommended Data Structures:

Feature	Data Structure	Justification
Attendance Tracking	Stack / Deque	Track entry/exit in chronological order with ability to access both ends
Event Registration	Hash Table	O(1) search and removal by student ID for quick registration management
Library Books	Priority Queue	Books sorted by due date priority, process overdue books first
Bus Scheduling	Graph	Represent bus routes and stops as vertices and edges for route planning
Cafeteria Queue	Queue	FIFO ensures students are served in order of arrival

Example Implementation: Cafeteria Order Queue

INPUT CODE

```
class CafeteriaQueue:
    """Cafeteria order management using Queue (FIFO)."""

    def __init__(self):
        self.orders = []
        self.order_number = 1

    def place_order(self, student_name, items):
        """Place a new order in the queue."""
        order = {
            'order_num': self.order_number,
            'student': student_name,
            'items': items
        }
        self.orders.append(order)
        print(f"Order #{self.order_number} placed for {student_name}")
        self.order_number += 1

    def serve_next(self):
        """Serve the next order in queue."""
        if not self.orders:
            return "No orders to serve"
        order = self.orders.pop(0)
        items_str = ", ".join(order['items'])
        return f"Serving Order #{order['order_num']} - {order['student']}:
{items_str}"

    def view_queue(self):
        """Display all pending orders."""
        if not self.orders:
            return "Queue is empty"
        result = []
        for order in self.orders:
            items_str = ", ".join(order['items'])
            result.append(f"Order #{order['order_num']} - {order['student']}:
{items_str}")
        return "\n".join(result)



# Demo Code
cafeteria = CafeteriaQueue()
cafeteria.place_order("Alice", ["Pizza", "Coke"])
cafeteria.place_order("Bob", ["Burger", "Fries"])
cafeteria.place_order("Charlie", ["Salad", "Juice"])
print("\nCurrent Queue:")
print(cafeteria.view_queue())
print("\n" + cafeteria.serve_next())
```

```
print(cafeteria.serve_next())
print("\nRemaining Queue:")
print(cafeteria.view_queue())
```

OUTPUT

```
Order #1 placed for Alice
Order #2 placed for Bob
Order #3 placed for Charlie

Current Queue:
Order #1 - Alice: Pizza, Coke
Order #2 - Bob: Burger, Fries
Order #3 - Charlie: Salad, Juice

Serving Order #1 - Alice: Pizza, Coke
Serving Order #2 - Bob: Burger, Fries

Remaining Queue:
Order #3 - Charlie: Salad, Juice
```

Task 10: Smart E-Commerce Platform

Scenario:

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

Recommended Data Structures:

Feature	Data Structure	Justification
Shopping Cart	Stack / Deque	Dynamic add/remove with undo functionality (LIFO for recent additions)
Order Processing	Queue	FIFO ensures orders are processed in placement order
Top Products	Priority Queue / BST	Maintain products sorted by sales count for quick top-N retrieval
Product Search	Hash Table	O(1) lookup by product ID for instant search results
Delivery Routes	Graph	Model delivery network with shortest path algorithms

Example Implementation: Product Search Engine

INPUT CODE

```
class ProductSearchEngine:
    """Fast product lookup using Hash Table."""

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

    def add_product(self, product_id, name, price, category):
        """Add a product to the database."""
        self.products[product_id] = {
            'name': name,
            'price': price,
            'category': category
        }
        print(f"Product added: {name} (ID: {product_id})")

    def search(self, product_id):
        """Search for a product by ID (O(1) lookup)."""
        if product_id in self.products:
            product = self.products[product_id]
            return product['name'] + " - $" + product['price'] + " - " +
product['category']
        return "Product not found"

    def delete_product(self, product_id):
        """Remove a product from the database."""
        if product_id in self.products:
            name = self.products[product_id]['name']
            del self.products[product_id]
            return f"Deleted: {name}"
        return "Product not found"

    def list_all(self):
        """List all products."""
        if not self.products:
            return "No products available"
        result = ["All Products:"]
        for pid, info in self.products.items():
            result.append("  ID " + pid + ": " + info['name'] + " - $" +
info['price'] + " - " + info['category'])
        return "\n".join(result)


# Demo Code
store = ProductSearchEngine()
store.add_product("P001", "Laptop", "999.99", "Electronics")
store.add_product("P002", "Mouse", "29.99", "Accessories")
store.add_product("P003", "Keyboard", "79.99", "Accessories")
```

```
print("\n" + store.list_all())
print("\nSearch P002: " + store.search("P002"))
print(store.delete_product("P002"))
print("\n" + store.list_all())
```

OUTPUT

```
Product added: Laptop (ID: P001)
Product added: Mouse (ID: P002)
Product added: Keyboard (ID: P003)

All Products:
  ID P001: Laptop - $999.99 - Electronics
  ID P002: Mouse - $29.99 - Accessories
  ID P003: Keyboard - $79.99 - Accessories

Search P002: Mouse - $29.99 - Accessories
Deleted: Mouse

All Products:
  ID P001: Laptop - $999.99 - Electronics
  ID P003: Keyboard - $79.99 - Accessories
```