Lab Assignment-11.1

Mohammed Awez Ali

2303A51767

Batch-12

Lab 11 – Data Structures with AI: Implementing Fundamental Structures

Task Description #1 – Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

Prompt: Generate a python code implementing stack class including push, pop, peek, and is_empty methods with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code implementing stack class including push, pop, peek, and is_empty methods with inline suggestions, use pydoc to generate documentation
class Stack:
    """
    A Stack is a data structure that follows the Last In First Out (LIFO) principle.

    This Stack class provides methods to perform basic stack operations such as push, pop, peek, and check if the stack is empty.

    Methods:
    push(item): Adds an item to the top of the stack.
    pop(): Removes and returns the item at the top of the stack. Raises an exception if the stack is empty.
    peek(): Returns the item at the top of the stack without removing it. Raises an exception if the stack is empty.
    is_empty(): Returns True if the stack is empty, False otherwise.

    Example:
    >>> stack = Stack()
    >>> stack.is_empty()
    True
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack.peek()
    2
    >>> stack.pop()
    2
    >>> stack.is_empty()
    False
    """

    def __init__(self):
        """Initialize an empty stack."""
        self.items = [] # Create an empty list to hold the items in the stack

    def push(self, item):
        """Add an item to the top of the stack."""
        self.items.append(item) # Append the item to the end of the list (top of the stack)

    def pop(self):
        """Remove and return the item at the top of the stack. Raises an exception if the stack is empty."""
        if self.is_empty(): # Check if the stack is empty before attempting to pop
            raise IndexError("Stack underflow: Cannot pop from an empty stack.") # Raise an exception if trying to pop from an empty stack
        return self.items.pop() # Remove and return the last item in the list (top of the stack)

    def peek(self):
        """Return the item at the top of the stack without removing it. Raises an exception if the stack is empty."""
        if self.is_empty(): # Check if the stack is empty before attempting to peek
            raise IndexError("Stack underflow: Cannot peek from an empty stack.") # Raise an exception if trying to peek from an empty stack
        return self.items[-1] # Return the last item in the list (top of the stack) without removing it

    def is_empty(self):
        """Return True if the stack is empty, False otherwise."""
        return len(self.items) == 0 # Check if the length of the items list is zero to determine if the stack is empty
# Example usage of the Stack class
stack = Stack() # Create a new stack instance
print("Is the stack empty?", stack.is_empty()) # Check if the stack is empty
stack.push(1) # Push the number 1 onto the stack
stack.push(2) # Push the number 2 onto the stack
print("Top item in the stack:", stack.peek()) # Peek at the top item in the stack
print("Popped item from the stack:", stack.pop()) # Pop the top item from the stack
print("Is the stack empty?", stack.is_empty()) # Check if the stack is empty after popping an item
```

Output:


```
Is the stack empty? True
Top item in the stack: 2
Popped item from the stack: 2
Is the stack empty? False
```

Explanation: The Stack class creates a data structure that follows the Last-In-First-Out (LIFO) principle, where items added last are removed first, just like stacking plates. The __init__ method initializes an empty list to store stack items. The push() method adds items to the top of the stack by appending to the list, while the pop() method removes and returns the top item, raising an error if the stack is empty.
The peek() and is_empty() methods allow you to view the top item without removing it and check if the stack has any items, respectively.

Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

Prompt: Queue implementation using list with inline suggestions, use pydoc to generate documentation

Code:

```python
#Queue implementation using list with inline suggestions, use pydoc to generate documentation
class Queue:
    """
    A Queue is a data structure that follows the First In First Out (FIFO) principle.

    This Queue class provides methods to perform basic queue operations such as enqueue, dequeue, peek, and check if the queue is empty.

    Methods:
    enqueue(item): Adds an item to the end of the queue.
    dequeue(): Removes and returns the item at the front of the queue. Raises an exception if the queue is empty.
    peek(): Returns the item at the front of the queue without removing it. Raises an exception if the queue is empty.
    is_empty(): Returns True if the queue is empty, False otherwise.

    Example:
    >>> queue = Queue()
    >>> queue.is_empty()
    True
    >>> queue.enqueue(1)
    >>> queue.enqueue(2)
    >>> queue.peek()
    1
    >>> queue.dequeue()
    1
    >>> queue.is_empty()
    False
    """

    def __init__(self):
        """Initialize an empty queue."""
        self.items = [] # Create an empty list to hold the items in the queue

    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item) # Append the item to the end of the list (end of the queue)

    def dequeue(self):
        """Remove and return the item at the front of the queue. Raises an exception if the queue is empty."""
        if self.is_empty(): # Check if the queue is empty before attempting to dequeue
            raise IndexError("Queue underflow: Cannot dequeue from an empty queue.") # Raise an exception if trying to dequeue from an empty queue
        return self.items.pop(0) # Remove and return the first item in the list (front of the queue)

    def peek(self):
        """Return the item at the front of the queue without removing it. Raises an exception if the queue is empty."""
        if self.is_empty(): # Check if the queue is empty before attempting to peek
            raise IndexError("Queue underflow: Cannot peek from an empty queue.") # Raise an exception if trying to peek from an empty queue
        return self.items[0] # Return the first item in the list (front of the queue) without removing it

    def is_empty(self):
        """Return True if the queue is empty, False otherwise."""
        return len(self.items) == 0 # Check if the length of the items list is zero to determine if the queue is empty
# Example usage of the Queue class
queue = Queue() # Create a new queue instance
print("Is the queue empty?", queue.is_empty()) # Check if the queue is empty
queue.enqueue(1) # Enqueue the number 1 into the queue
queue.enqueue(2) # Enqueue the number 2 into the queue
print("Front item in the queue:", queue.peek()) # Peek at the front item in the queue
print("Dequeued item from the queue:", queue.dequeue()) # Dequeue the front item from the queue
print("Is the queue empty?", queue.is_empty()) # Check if the queue is empty after dequeuing an item
```

Output:

```
Is the queue empty? True
Front item in the queue: 1
Dequeued item from the queue: 1
Is the queue empty? False
```

Explanation: The Queue class implements a First-In-First-Out (FIFO) data structure where items are processed in the order they arrive, similar to a real-world queue at a store. The __init__ method sets up an empty list to hold queue items. The enqueue() method adds items to the back of the queue using append, while the dequeue() method removes items from the front using pop(0). The peek() and is_empty() methods provide ways to view the front item without removing it and to check if the queue is empty without modification.

Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Prompt: generate a python code for Singly Linked List implementation with methods and inline suggestions, use pydoc to generate documentation

Code:

```python
#generate a python code for Singly Linked List implementation with methods and inline suggestions, use pydoc to generate documentation
class Node:
    """
    A Node in a Singly Linked List.

    Attributes:
    data: The value stored in the node.
    next: A reference to the next node in the list.

    Example:
    >>> node1 = Node(10)
    >>> node2 = Node(20)
    >>> node1.next = node2
    >>> node1.data
    10
    >>> node1.next.data
    20
    """

    def __init__(self, data):
        """Initialize a new node with the given data and set next to None."""
        self.data = data # Store the data in the node
        self.next = None # Initialize the next reference to None
class SinglyLinkedList:
    """
    A Singly Linked List is a data structure that consists of a sequence of nodes, where each node contains data and a reference to the next node in the list.

    This SinglyLinkedList class provides methods to perform basic linked list operations such as inserting nodes, deleting nodes, and traversing the list.

    Methods:
    insert_at_end(data): Inserts a new node with the given data at the end of the list.
    delete_node(key): Deletes the first node with the specified key (data) from the list. Raises an exception if the key is not found.
    traverse(): Returns a list of all elements in the linked list in order.

    Example:
    >>> linked_list = SinglyLinkedList()
    >>> linked_list.insert_at_end(10)
    >>> linked_list.insert_at_end(20)
    >>> linked_list.traverse()
    [10, 20]
    >>> linked_list.delete_node(10)
    >>> linked_list.traverse()
    [20]
    """

    def __init__(self):
        """Initialize an empty singly linked list."""
        self.head = None # Initialize the head of the linked list to None
    def insert_at_end(self, data):
        """Insert a new node with the given data at the end of the list."""
        new_node = Node(data) # Create a new node with the given data
        if self.head is None: # If the list is empty, set the new node as the head
            self.head = new_node
            return
        last_node = self.head # Start from the head of the list
        while last_node.next: # Traverse to the end of the list
            last_node = last_node.next
        last_node.next = new_node # Set the next reference of the last node to the new node
    def delete_node(self, key):
        """Delete the first node with the specified key (data) from the list. Raises an exception if the key is not found."""
        current_node = self.head # Start from the head of the list
        previous_node = None # Initialize a variable to keep track of the previous node

        while current_node: # Traverse the list until the end
            if current_node.data == key: # Check if the current node's data matches the key
                if previous_node: # If there is a previous node, update its next reference to skip the current node
                    previous_node.next = current_node.next
                else: # If there is no previous node, it means we are deleting the head of the list
                    self.head = current_node.next # Update the head to point to the next node
                return # Exit after deleting the node
            previous_node = current_node # Update the previous node to be the current node
            current_node = current_node.next # Move to the next node

        raise ValueError(f"Key not found: {key}") # Raise an exception if the key is not found in the list
    def traverse(self):
        """Return a list of all elements in the linked list in order."""
        elements = [] # Create an empty list to hold the elements of the linked list
        current_node = self.head # Start from the head of the list
        while current_node: # Traverse the list until the end
            elements.append(current_node.data) # Append the current node's data to the elements list
            current_node = current_node.next # Move to the next node
        return elements # Return the list of elements in order
# Example usage of the SinglyLinkedList class
linked_list = SinglyLinkedList() # Create a new singly linked list instance
linked_list.insert_at_end(10) # Insert a new node with data 10 at the end of the list
linked_list.insert_at_end(20) # Insert a new node with data 20 at the end of the list
print("Elements in the linked list:", linked_list.traverse()) # Traverse the linked list and print the elements
linked_list.delete_node(10) # Delete the node with data 10 from the list
print("Elements in the linked list after deletion:", linked_list.traverse()) # Traverse the linked list again and print the elements after deletion
```

Output:

```
Elements in the linked list: [10, 20]
Elements in the linked list after deletion: [20]
```

Explanation: The Node class represents individual elements in a linked list, each containing data and a reference to the next node, forming a chain-like structure. The insert_at_end() method adds new items to the list by creating nodes and linking them together, while delete_node() removes items by updating the links between nodes. The traverse() method collects all elements in order, allowing you to see the entire list's contents.

Task Description #4 – Binary Search Tree (BST)

Task: Use AI to create a BST with insert and in-order traversal methods.

Prompt: Generate a python code for BST with insert and in-order traversal methods with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code for BST with insert and in-order traversal methods with inline suggestions, use pydoc to generate documentation
class TreeNode:
    """
    A TreeNode in a Binary Search Tree (BST).

    Attributes:
    data: The value stored in the node.
    left: A reference to the left child node.
    right: A reference to the right child node.

    Example:
    >>> node1 = TreeNode(10)
    >>> node2 = TreeNode(5)
    >>> node3 = TreeNode(15)
    >>> node1.left = node2
    >>> node1.right = node3
    >>> node1.data
    10
    >>> node1.left.data
    5
    >>> node1.right.data
    15
    """

    def __init__(self, data):
        """Initialize a new tree node with the given data and set left and right to None."""
        self.data = data # Store the data in the tree node
        self.left = None # Initialize the left child reference to None
        self.right = None # Initialize the right child reference to None
class BinarySearchTree:
    """
    A Binary Search Tree (BST) is a binary tree where each node has at most two children, and the left child is less than the parent node, while the right child is greater than the parent node.

    This BinarySearchTree class provides methods to perform basic BST operations such as insertion and in-order traversal.

    Methods:
    insert(data): Inserts a new node with the given data into the BST while maintaining the BST property.
    in_order_traversal(): Returns a list of all elements in the BST in sorted order (in-order traversal).

    Example:
    >>> bst = BinarySearchTree()
    >>> bst.insert(10)
    >>> bst.insert(5)
    >>> bst.insert(15)
    >>> bst.in_order_traversal()
    [5, 10, 15]
    """

    def __init__(self):
        """Initialize an empty binary search tree."""
        self.root = None # Initialize the root of the BST to None
    def insert(self, data):
        """Insert a new node with the given data into the BST while maintaining the BST property."""
        if self.root is None: # If the tree is empty, set the new node as the root
            self.root = TreeNode(data)
        else:
            self._insert_recursive(self.root, data) # Call the recursive helper method to insert the new node
    def _insert_recursive(self, current_node, data):
        """Helper method to recursively insert a new node into the BST."""
        if data < current_node.data: # If the new data is less than the current node's data
            if current_node.left is None: # If there is no left child, insert the new node here
                current_node.left = TreeNode(data)
            else:
                self._insert_recursive(current_node.left, data) # Recur on the left subtree
        else: # If the new data is greater than or equal to the current node's data
            if current_node.right is None: # If there is no right child, insert the new node here
                current_node.right = TreeNode(data)
            else:
                self._insert_recursive(current_node.right, data) # Recur on the right subtree
    def in_order_traversal(self):
        """Return a list of all elements in the BST in sorted order (in-order traversal)."""
        elements = [] # Create an empty list to hold the elements of the BST
        self._in_order_recursive(self.root, elements) # Call the recursive helper method to perform in-order traversal
        return elements # Return the list of elements in sorted order
    def _in_order_recursive(self, current_node, elements):
        if current_node is not None: # If the current node is not None
            self._in_order_recursive(current_node.left, elements) # Recur on the left subtree
            elements.append(current_node.data) # Append the current node's data to the elements list
            self._in_order_recursive(current_node.right, elements) # Recur on the right subtree
# Example usage of the BinarySearchTree class
bst = BinarySearchTree() # Create a new binary search tree instance
bst.insert(10) # Insert the number 10 into the BST
bst.insert(5) # Insert the number 5 into the BST
bst.insert(15) # Insert the number 15 into the BST
print("In-order traversal of the BST:", bst.in_order_traversal()) # Perform in-order traversal of the BST and print the elements in sorted order
```

Output:

```
In-order traversal of the BST: [5, 10, 15]
```

Explanation: The TreeNode class represents individual nodes in a binary search tree where each node has data and references to left and right children for organizing hierarchical data. The insert() method places new values in the correct positions to maintain the BST property using recursion. The in_order_traversal() method retrieves all elements in sorted order by recursively visiting left subtree, current node, then right subtree.

Task Description #5 – Hash Table

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

Prompt: Generate a python code to to implement a hash table with basic insert, search, and delete methods with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to to implement a hash table with basic insert, search, and delete methods with inline suggestions, use pydoc to generate documentation
class HashTable:
    """
    A Hash Table is a data structure that implements an associative array, a structure that can map keys to values.

    This HashTable class provides methods to perform basic hash table operations such as insertion, searching, and deletion of key-value pairs.

    Methods:
    insert(key, value): Inserts a key-value pair into the hash table. If the key already exists, it updates the value.
    search(key): Returns the value associated with the given key. Raises an exception if the key is not found.
    delete(key): Removes the key-value pair associated with the given key from the hash table. Raises an exception if the key is not found.

    Example:
    >>> hash_table = HashTable()
    >>> hash_table.insert("name", "Alice")
    >>> hash_table.insert("age", 30)
    >>> hash_table.search("name")
    'Alice'
    >>> hash_table.delete("age")
    >>> hash_table.search("age")
    Traceback (most recent call last):
        ... KeyError: 'Key not found: age'
    """

    def __init__(self):
        """Initialize an empty hash table."""
        self.table = {} # Create an empty dictionary to hold the key-value pairs in the hash table
    def insert(self, key, value):
        """Insert a key-value pair into the hash table. If the key already exists, it updates the value."""
        self.table[key] = value # Insert or update the key-value pair in the dictionary
    def search(self, key):
        """Return the value associated with the given key. Raises an exception if the key is not found."""
        if key in self.table: # Check if the key exists in the hash table
            return self.table[key] # Return the value associated with the key
        else:
            raise KeyError(f"Key not found: {key}") # Raise an exception if the key is not found
    def delete(self, key):
        """Remove the key-value pair associated with the given key from the hash table. Raises an exception if the key is not found."""
        if key in self.table: # Check if the key exists in the hash table
            del self.table[key] # Remove the key-value pair from the dictionary
        else:
            raise KeyError(f"Key not found: {key}") # Raise an exception if the key is not found
# Example usage of the HashTable class
hash_table = HashTable() # Create a new hash table instance
hash_table.insert("name", "Alice") # Insert a key-value pair into the hash table
hash_table.insert("age", 30) # Insert another key-value pair into the hash table
print("Value for key 'name':", hash_table.search("name")) # Search for the value associated with the key 'name' and print it
hash_table.delete("age") # Delete the key-value pair associated with the key 'age'
try:
    print("Value for key 'age':", hash_table.search("age")) # Attempt to search for the value associated with the key 'age' after deletion
except KeyError as e:
    print(e) # Print the exception message if the key 'age' is not found in the hash table after deletion
```

Output:

```
Value for key 'name': Alice
'Key not found: age'
```

Explanation: The HashTable class uses a Python dictionary to implement a fast lookup data structure where data is stored as key-value pairs for efficient retrieval. The insert() method stores or updates a key-value pair in the hash table with constant-time complexity. The search() method quickly finds and returns values associated with specific keys, raising an error if the key doesn't exist. The delete() method removes key-value pairs from the hash table, allowing you to manage and clean up stored data efficiently.

Task Description #6 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Prompt: Generate a python code to implement a graph using an adjacency list with methods for adding vertices, adding edges, and performing depth-first search (DFS) with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement a graph using an adjacency list with methods for adding vertices, adding edges, and performing depth-first search (DFS) with inline suggestions, use pydoc to generate documentation
class Graph:
    """
    A Graph is a data structure that consists of a set of vertices (nodes) and a set of edges (connections) between those vertices.

    This Graph class provides methods to perform basic graph operations such as adding vertices, adding edges, and performing depth-first search (DFS).

    Methods:
    add_vertex(vertex): Adds a vertex to the graph.
    add_edge(vertex1, vertex2): Adds an edge between two vertices in the graph. If the vertices do not exist, they are added to the graph.
    dfs(start_vertex): Performs depth-first search starting from the given vertex and returns a list of visited vertices in the order they were visited.

    Example:
    >>> graph = Graph()
    >>> graph.add_vertex("A")
    >>> graph.add_vertex("B")
    >>> graph.add_edge("A", "B")
    >>> graph.dfs("A")
    ['A', 'B']
    """

    def __init__(self):
        """Initialize an empty graph using an adjacency list."""
        self.adjacency_list = {} # Create an empty dictionary to hold the adjacency list representation of the graph
    def add_vertex(self, vertex):
        """Add a vertex to the graph."""
        if vertex not in self.adjacency_list: # Check if the vertex already exists in the graph
            self.adjacency_list[vertex] = [] # Add the vertex to the adjacency list with an empty list of neighbors
    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph. If the vertices do not exist, they are added to the graph."""
        self.add_vertex(vertex1) # Ensure vertex1 is added to the graph
        self.add_vertex(vertex2) # Ensure vertex2 is added to the graph
        self.adjacency_list[vertex1].append(vertex2) # Add vertex2 to the list of neighbors for vertex1
        self.adjacency_list[vertex2].append(vertex1) # Add vertex1 to the list of neighbors for vertex2 (undirected edge)
    def dfs(self, start_vertex):
        """Perform depth-first search starting from the given vertex and return a list of visited vertices in the order they were visited."""
        visited = set() # Create a set to keep track of visited vertices
        result = [] # Create a list to store the order of visited vertices

        def dfs_helper(vertex):
            if vertex not in visited: # Check if the vertex has not been visited
                visited.add(vertex) # Mark the vertex as visited
                result.append(vertex) # Add the vertex to the result list
                for neighbor in self.adjacency_list[vertex]: # Recur for all the neighbors of the vertex
                    dfs_helper(neighbor)
        dfs_helper(start_vertex) # Start the DFS traversal from the given vertex
        return result # Return the list of visited vertices in the order they were visited
# Example usage of the Graph class
graph = Graph() # Create a new graph instance
graph.add_vertex("A") # Add vertex 'A' to the graph
graph.add_vertex("B") # Add vertex 'B' to the graph
graph.add_edge("A", "B") # Add an edge between vertex 'A' and vertex 'B'
print("Depth-First Search starting from vertex 'A':", graph.dfs("A")) # Perform DFS starting from vertex 'A' and print the order of visited vertices
```

Output:

```
Depth-First Search starting from vertex 'A': ['A', 'B']
```

Explanation: The Graph class uses an adjacency list (dictionary) to represent connections between vertices, making it efficient for storing and traversing network-like structures. The add_vertex() method adds individual nodes to the graph, while add_edge() creates connections between vertices in both directions for undirected graphs.

Task Description #7 – Priority Queue

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

Prompt: Generate a python code to implement a priority queue using Python's heapq module with methods for adding elements, removing the highest priority element, and peeking at the highest priority element with inline suggestions, use pydoc to generate documentation

Code:

Output:


```
Highest priority element (peek): task2
Removed highest priority element: task2
Highest priority element after removal (peek): task1
```

Explanation: The PriorityQueue class uses Python's heapq module to manage elements based on priority levels, where lower numbers indicate higher priority for processing. The add() method inserts elements with a priority value into the heap, automatically maintaining the correct ordering. The remove() method extracts and returns the highest-priority element efficiently, and peek() allows viewing the top-priority element without disturbing the queue.

Task Description #8 – Deque

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

Prompt: Generate a python code to implement a double-ended queue using collections.deque with methods for adding and removing elements from both ends with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement a double-ended queue using collections.deque with methods for adding and removing elements from both ends with inline suggestions, use pydoc to generate documentation
class Deque:
    """
    A Double-Ended Queue (Deque) is a data structure that allows for efficient addition and removal of elements from both ends.

    This Deque class uses Python's collections.deque to implement a double-ended queue with methods for adding and removing elements from both ends."""

    def __init__(self):
        """Initialize an empty double-ended queue."""
        self.deque = collections.deque() # Create an empty deque using collections.deque
    def add_front(self, element):
        """Add an element to the front of the deque."""
        self.deque.appendleft(element) # Use appendleft to add an element to the front of the deque
    def add_rear(self, element):
        """Add an element to the rear of the deque."""
        self.deque.append(element) # Use append to add an element to the rear of the deque
    def remove_front(self):
        """Remove and return the element from the front of the deque. Raises an exception if the deque is empty."""
        if not self.deque: # Check if the deque is empty before attempting to remove an element from the front
            raise IndexError("Deque underflow: Cannot remove from an empty deque.") # Raise an exception if trying to remove from an empty deque
        return self.deque.popleft() # Use popleft to remove and return the element from the front of the deque
    def remove_rear(self):
        """Remove and return the element from the rear of the deque. Raises an exception if the deque is empty."""
        if not self.deque: # Check if the deque is empty before attempting to remove an element from the rear
            raise IndexError("Deque underflow: Cannot remove from an empty deque.") # Raise an exception if trying to remove from an empty deque
        return self.deque.pop() # Use pop to remove and return the element from the rear of the deque
# Example usage of the Deque class
deque = Deque() # Create a new double-ended queue instance
deque.add_front("front1") # Add an element "front1" to the front of the deque
deque.add_rear("rear1") # Add an element "rear1" to the rear of the deque
print("Removed from front:", deque.remove_front()) # Remove and print the element from the front of the deque
print("Removed from rear:", deque.remove_rear()) # Remove and print the element from the rear of the deque
```

Output:

```
Removed from front: front1
Removed from rear: rear1
```

Explanation: The Deque (Double-Ended Queue) class uses collections.deque .The add_front() and add_rear() methods allow you to add items to either end of the deque using appendleft and append operations. The remove_front() and remove_rear() methods extract items from either end using popleft and pop, raising errors if the deque is empty.

Task Description #9 Real-Time Application Challenge – Choose the Right Data Structure

I. Student Attendance Tracking – Daily log of students entering/exiting the campus.

Prompt: Generate a python code for student attendance tracking using hash table with methods for marking attendance, checking attendance, and generating attendance reports with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code for student attendance tracking using hash table with methods for marking attendance, checking attendance,
# and generating attendance reports with inline suggestions, use pydoc to generate documentation
class AttendanceTracker:
    """
    A Student Attendance Tracker is a system that allows for tracking the attendance of students in a class or course.

    This AttendanceTracker class uses a hash table (dictionary) to store student names and their attendance status,
    with methods for marking attendance, checking attendance, and generating attendance reports.

    Methods:
    mark_attendance(student_name): Marks the attendance of a student as present. If the student does not exist in the tracker, they are added with a status of present.
    check_attendance(student_name): Returns True if the student is marked as present, False if marked as absent, and None if the student is not found in the tracker.
    generate_report(): Returns a dictionary containing the attendance report with student names as keys and their attendance status as values.

    Example:
    >>> tracker = AttendanceTracker()
    >>> tracker.mark_attendance("Alice")
    >>> tracker.mark_attendance("Bob")
    >>> tracker.check_attendance("Alice")
    True
    >>> tracker.check_attendance("Charlie")
    None
    >>> tracker.generate_report()
    {'Alice': 'Present', 'Bob': 'Present'}
    """

    def __init__(self):
        """Initialize an empty attendance tracker."""
        self.attendance = {} # Create an empty dictionary to hold student names and their attendance status
    def mark_attendance(self, student_name):
        """Mark the attendance of a student as present. If the student does not exist in the tracker, they are added with a status of present."""
        self.attendance[student_name] = "Present" # Mark the student's attendance as "Present" in the dictionary
    def check_attendance(self, student_name):
        """Return True if the student is marked as present, False if marked as absent, and None if the student is not found in the tracker."""
        return self.attendance.get(student_name) # Return the attendance status of the student, or None if the student is not found
    def generate_report(self):
        """Return a dictionary containing the attendance report with student names as keys and their attendance status as values."""
        return self.attendance # Return the entire attendance dictionary as the report
# Example usage of the AttendanceTracker class
tracker = AttendanceTracker() # Create a new attendance tracker instance
tracker.mark_attendance("Alice") # Mark Alice's attendance as present
tracker.mark_attendance("Bob") # Mark Bob's attendance as present
print("Is Alice present?", tracker.check_attendance("Alice")) # Check if Alice is present and print the result
print("Is Charlie present?", tracker.check_attendance("Charlie")) # Check if Charlie is present
print("Attendance Report:", tracker.generate_report()) # Generate and print the attendance report
```

Output:

```
Is Alice present? Present
Is Charlie present? None
Attendance Report: {'Alice': 'Present', 'Bob': 'Present'}
```

Explanation: The AttendanceTracker class uses a hash table to store student attendance records, allowing schools or organizations to efficiently track who is present or absent. The mark_attendance() method records a student as present by adding or updating their entry in the dictionary. The check_attendance() method retrieves a student's attendance status, returning their status or None if they haven't been recorded. The generate_report() method returns the entire attendance dictionary, enabling administrators to get an overview of all students' attendance in a single call.

2. Event Registration System – Manage participants in events with quick search and removal.

Prompt: Generate a python code for event registration system using binary search tree with methods for registering participants, checking registration status, and generating participant lists with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code for event registration system using binary search tree with methods for registering participants, c
# hecking registration status, and generating participant lists with inline suggestions, use pydoc to generate documentation
class ParticipantNode:
    """
    A ParticipantNode in a binary search tree for an event registration system.

    Attributes:
    name: The name of the participant.
    left: A reference to the left child node.
    right: A reference to the right child node.

    Example:
    >>> node1 = ParticipantNode("Alice")
    >>> node2 = ParticipantNode("Bob")
    >>> node3 = ParticipantNode("Charlie")
    >>> node1.left = node2
    >>> node1.right = node3
    >>> node1.name
    'Alice'
    >>> node1.left.name
    'Bob'
    >>> node1.right.name
    'Charlie'
    """

    def __init__(self, name):
        """Initialize a new participant node with the given name and set left and right to None."""
        self.name = name # Store the participant's name in the node
        self.left = None # Initialize the left child reference to None
        self.right = None # Initialize the right child reference to None
class EventRegistrationSystem:
    """
    An Event Registration System is a system that allows for managing the registration of participants for an event.

    This EventRegistrationSystem class uses a binary search tree to store participant names and provides methods
    for registering participants, checking registration status, and generating participant lists.

    Methods:
    register_participant(name): Registers a participant with the given name. If the participant is already registered, it does not add them again.
    check_registration(name): Returns True if the participant is registered, False otherwise.
    generate_participant_list(): Returns a list of all registered participants in sorted order.

    Example:
    >>> system = EventRegistrationSystem()
    >>> system.register_participant("Alice")
    >>> system.register_participant("Bob")
    >>> system.check_registration("Alice")
    True
    >>> system.check_registration("Charlie")
    False
    >>> system.generate_participant_list()
    ['Alice', 'Bob']
    """
```

```python
    def __init__(self):
        """Initialize an empty event registration system."""
        self.root = None # Initialize the root of the binary search tree to None
    def register_participant(self, name):
        """Register a participant with the given name. If the participant is already registered, it does not add them again."""
        if self.root is None: # If the tree is empty, set the new participant as the root
            self.root = ParticipantNode(name)
        else:
            self._register_recursive(self.root, name) # Call the recursive helper method to register the participant
    def _register_recursive(self, current_node, name):
        """Helper method to recursively register a participant in the binary search tree."""
        if name < current_node.name: # If the new participant's name is less than the current node's name
            if current_node.left is None: # If there is no left child, insert the new participant here
                current_node.left = ParticipantNode(name)
            else:
                self._register_recursive(current_node.left, name) # Recur on the left subtree
        elif name > current_node.name: # If the new participant's name is greater than the current node's name
            if current_node.right is None: # If there is no right child, insert the new participant here
                current_node.right = ParticipantNode(name)
            else:
                self._register_recursive(current_node.right, name) # Recur on the right subtree
        else:
            pass # If the participant is already registered (name matches), do nothing
    def check_registration(self, name):
        """Return True if the participant is registered, False otherwise."""
        return self._check_registration_recursive(self.root, name) # Call the recursive helper method to check registration status
    def _check_registration_recursive(self, current_node, name):
        """Helper method to recursively check if a participant is registered in the binary search tree."""
        if current_node is None: # If the current node is None, the participant is not found
            return False
        if name == current_node.name: # If the current node's name matches the participant's name, they are registered
            return True
        elif name < current_node.name: # If the participant's name is less than the current node's name, search in the left subtree
            return self._check_registration_recursive(current_node.left, name)
        else: # If the participant's name is greater than the current node's name, search in the right subtree
            return self._check_registration_recursive(current_node.right, name) # Recur on the right subtree
    def generate_participant_list(self):
        """Return a list of all registered participants in sorted order."""
        participants = [] # Create an empty list to hold the participant names
        self._in_order_traversal(self.root, participants) # Call the recursive helper method to perform in-order traversal
        return participants # Return the list of participant names in sorted order
    def _in_order_traversal(self, current_node, participants):
        """Helper method to recursively perform in-order traversal of the binary search tree to generate a sorted list of participants."""
        if current_node is not None: # If the current node is not None
            self._in_order_traversal(current_node.left, participants) # Recur on the left subtree
            participants.append(current_node.name) # Append the current node's name to the participants list
            self._in_order_traversal(current_node.right, participants) # Recur on the right subtree
# Example usage of the EventRegistrationSystem class
system = EventRegistrationSystem() # Create a new event registration system instance
system.register_participant("Alice") # Register a participant named "Alice"
system.register_participant("Bob") # Register a participant named "Bob"
print("Is Alice registered?", system.check_registration("Alice")) # Check if "Alice" is registered and print the result
print("Is Charlie registered?", system.check_registration("Charlie")) # Check if "Charlie" is registered and print the result
print("Participant List:", system.generate_participant_list()) # Generate and print the list of registered participants in sorted order
```

Output:

```
Is Alice registered? True
Is Charlie registered? False
Participant List: ['Alice', 'Bob']
```

Explanation: The EventRegistrationSystem uses a binary search tree to organize registered participants alphabetically, allowing for efficient searching and sorted participant listings. The register_participant() method adds new participants in sorted order, preventing duplicates since the BST structure naturally handles this. The check_registration() method quickly searches for specific participants, and generate_participant_list() retrieves all registered participants in alphabetical order for event planning purposes.

3. Library Book Borrowing – Keep track of available books and their due dates.

Prompt: Generate a python code to implement Library Book Borrowing using Priority Queue with methods for adding books, borrowing books based on priority, and returning books with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Library Book Borrowing using Priority Queue with methods for adding books, borrowing books based on priority,
# and returning books with inline suggestions, use pydoc to generate documentation
class Book:
    """
    A Book class to represent a book in the library.

    Attributes:
    title: The title of the book.
    priority: The priority of the book (lower numbers indicate higher priority).
    Example:
    >>> book1 = Book("Book A", 1)
    >>> book2 = Book("Book B", 2)
    >>> book1.title
    'Book A'
    >>> book1.priority
    1
    >>> book2.title
    'Book B'
    >>> book2.priority
    2
    """

    def __init__(self, title, priority):
        """Initialize a new book with the given title and priority."""
        self.title = title # Store the title of the book
        self.priority = priority # Store the priority of the book
class Library:
    """
    A Library class to manage book borrowing using a priority queue.

    This Library class provides methods for adding books, borrowing books based on priority, and returning books.

    Methods:
    add_book(title, priority): Adds a book to the library with the specified title and priority.
    borrow_book(): Borrows the book with the highest priority (lowest priority value) from the library. Raises an exception if no books are available.
    return_book(book): Returns a borrowed book back to the library.

    Example:
    >>> library = Library()
    >>> library.add_book("Book A", 1)
    >>> library.add_book("Book B", 2)
    >>> borrowed_book = library.borrow_book()
    >>> borrowed_book.title
    'Book A'
    >>> library.borrow_book().title
    'Book B'
    >>> library.borrow_book()
    Traceback (most recent call last):
        ... Exception: No books available to borrow.
    """

    def __init__(self):
        """Initialize an empty library with a priority queue for books."""
        self.books = [] # Create an empty list to hold the books in the library
        self.borrowed_books = set() # Create a set to keep track of borrowed books
    def add_book(self, title, priority):
        """Add a book to the library with the specified title and priority."""
        book = Book(title, priority) # Create a new book instance with the given title and priority
        heapq.heappush(self.books, (priority, book)) # Use heapq to add the book to the priority queue based on its priority
    def borrow_book(self):
        """Borrow the book with the highest priority (lowest priority value) from the library. Raises an exception if no books are available."""
        while self.books: # While there are books available in the library
            priority, book = heapq.heappop(self.books) # Get the book with the highest priority (lowest priority value)
            if book not in self.borrowed_books: # Check if the book is not already borrowed
                self.borrowed_books.add(book) # Mark the book as borrowed
                return book # Return the borrowed book
        raise Exception("No books available to borrow.") # Raise an exception if no books are available to borrow
    def return_book(self, book):
        """Return a borrowed book back to the library."""
        if book in self.borrowed_books: # Check if the book is currently borrowed
            self.borrowed_books.remove(book) # Remove the book from the set of borrowed books
            heapq.heappush(self.books, (book.priority, book)) # Add the returned book back to the priority queue based on its priority
# Example usage of the Library class
library = Library() # Create a new library instance
library.add_book("Book A", 1) # Add a book with title "Book A" and priority 1 to the library
library.add_book("Book B", 2) # Add a book with title "Book B" and priority 2 to the library
borrowed_book = library.borrow_book() # Borrow the book with the highest priority from the library
print("Borrowed book:", borrowed_book.title) # Print the title of the borrowed book
print("Borrowing another book:", library.borrow_book().title) # Borrow another book and print its title
try:
    library.borrow_book() # Attempt to borrow a book when no books are available
except Exception as e:
    print(e) # Print the exception message if no books are available to borrow
```

Output:

```
Borrowed book: Book A
Borrowing another book: Book B
No books available to borrow.
```

Explanation: The Library class manages a collection of books with priority-based borrowing, where books with lower priority numbers are borrowed first, The add_book() method uses a heap to efficiently insert books into the library's collection. The borrow_book() method retrieves and removes the highest-priority book available for borrowing, while return_book() allows borrowed books to be added back to the collection for others to borrow.

4. Bus Scheduling System – Maintain bus routes and stop connections.

Prompt: Generate a python code to implement Bus Scheduling System using Graph with methods for adding bus routes, finding the shortest route between two stops, and generating a schedule for a given stop with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Bus Scheduling System using Graph with methods for adding bus routes, finding the shortest route between two stops,
#  and generating a schedule for a given stop with inline suggestions, use pydoc to generate documentation
class BusStopNode:
    """
    A BusStopNode in a graph for a bus scheduling system.

    Attributes:
    name: The name of the bus stop.
    neighbors: A dictionary where keys are neighboring bus stops and values are the distances to those stops.

    Example:
    >>> node1 = BusStopNode("Stop A")
    >>> node2 = BusStopNode("Stop B")
    >>> node3 = BusStopNode("Stop C")
    >>> node1.neighbors[node2] = 5
    >>> node1.neighbors[node3] = 10
    >>> node1.name
    'Stop A'
    >>> node1.neighbors[node2]
    5
    >>> node1.neighbors[node3]
    10
    """

    def __init__(self, name):
        """Initialize a new bus stop node with the given name and an empty dictionary of neighbors."""
        self.name = name # Store the name of the bus stop in the node
        self.neighbors = {} # Initialize an empty dictionary to hold neighboring bus stops and their distances
class BusSchedulingSystem:
    """
    A Bus Scheduling System is a system that allows for managing bus routes, finding the shortest route between two stops, and generating a schedule for a given stop.

    This BusSchedulingSystem class uses a graph to represent bus stops and routes, with methods for adding bus routes,
    finding the shortest route between two stops, and generating a schedule for a given stop.

    Methods:
    add_bus_route(stop1, stop2, distance): Adds a bus route between two stops with the specified distance. If the stops do not exist, they are added to the system.
    find_shortest_route(stop1, stop2): Returns the shortest route between two stops using Dijkstra's algorithm. Raises an exception if either stop is not found.
    generate_schedule(stop): Returns a list of neighboring stops and their distances from the given stop. Raises an exception if the stop is not found.

    Example:
    >>> system = BusSchedulingSystem()
    >>> system.add_bus_route("Stop A", "Stop B", 5)
    >>> system.add_bus_route("Stop A", "Stop C", 10)
    >>> system.find_shortest_route("Stop A", "Stop C")
    (['Stop A', 'Stop C'], 10)
    >>> system.generate_schedule("Stop A")
    [('Stop B', 5), ('Stop C', 10)]
    """

    def __init__(self):
        """Initialize an empty bus scheduling system."""
        self.stops = {} # Create an empty dictionary to hold bus stops and their corresponding nodes
    def add_bus_route(self, stop1, stop2, distance):
        """Add a bus route between two stops with the specified distance. If the stops do not exist, they are added to the system."""
        if stop1 not in self.stops: # Check if stop1 does not exist in the system
            self.stops[stop1] = BusStopNode(stop1) # Add stop1 to the system as a new bus stop node
        if stop2 not in self.stops: # Check if stop2 does not exist in the system
            self.stops[stop2] = BusStopNode(stop2) # Add stop2 to the system as a new bus stop node
        self.stops[stop1].neighbors[self.stops[stop2]] = distance # Add stop2 as a neighbor of stop1 with the specified distance
        self.stops[stop2].neighbors[self.stops[stop1]] = distance # Add stop1 as a neighbor of stop2 with the specified distance (undirected route)
    def find_shortest_route(self, stop1, stop2):
        """Return the shortest route between two stops using Dijkstra's algorithm. Raises an exception if either stop is not found."""
        if stop1 not in self.stops or stop2 not in self.stops: # Check if either stop1 or stop2 is not found in the system
            raise ValueError("One or both stops not found in the system.") # Raise an exception if either stop is not found
        # Implement Dijkstra's algorithm to find the shortest route between stop1 and stop2
        import heapq
        queue = [(0, self.stops[stop1], [])] # Initialize a priority queue with (distance, current_node, path)
        visited = set() # Create a set to keep track of visited nodes
        while queue: # While there are nodes to process in the priority queue
            (cost, current_node, path) = heapq.heappop(queue) # Get the node with the lowest cost
            if current_node.name in visited: # If the current node has already been visited, skip it
                continue
            visited.add(current_node.name) # Mark the current node as visited
            path = path + [current_node.name] # Update the path with the current node's name
            if current_node.name == stop2: # If we have reached the destination stop, return the path and total cost
                return (path, cost)
            for neighbor, distance in current_node.neighbors.items(): # Iterate through neighbors of the current node
                if neighbor.name not in visited: # If the neighbor has not been visited
                    heapq.heappush(queue, (cost + distance, neighbor, path)) # Add the neighbor to the priority queue with updated cost and path
        return None # Return None if there is no route between stop1 and stop2
    def generate_schedule(self, stop):
        """Return a list of neighboring stops and their distances from the given stop. Raises an exception if the stop is not found."""
        if stop not in self.stops: # Check if the given stop is not found in the system
            raise ValueError("Stop not found in the system.") # Raise an exception if the stop is not found
        neighbors = self.stops[stop].neighbors # Get the neighbors of the given stop
        return [(neighbor.name, distance) for neighbor, distance in neighbors.items()] # Return a list of neighboring stops and their distances from the given stop
# Example usage of the BusSchedulingSystem class
system = BusSchedulingSystem() # Create a new bus scheduling system instance
system.add_bus_route("Stop A", "Stop B", 5) # Add a bus route between "Stop A" and "Stop B" with a distance of 5
system.add_bus_route("Stop A", "Stop C", 10) # Add a bus route between "Stop A" and "Stop C" with a distance of 10
print("Shortest route from Stop A to Stop C:", system.find_shortest_route("Stop A", "Stop C")) # Find and print the shortest route from "Stop A" to "Stop C"
print("Schedule for Stop A:", system.generate_schedule("Stop A")) # Generate and print the schedule for "Stop A" showing neighboring stops and their distances
```

Output:

```
Shortest route from Stop A to Stop C: (['Stop A', 'Stop C'], 10)
Schedule for Stop A: [('Stop B', 5), ('Stop C', 10)]
```

Explanation:The BusSchedulingSystem uses a weighted graph to represent bus stops and routes with distances between them, enabling efficient route planning for transportation companies..

The add_bus_route() method creates bidirectional connections between stops with specific distances. The find_shortest_route() implements Dijkstra's algorithm to compute the shortest path between any two stops, while generate_schedule() lists all nearby stops and their distances for route planning and passenger information.

5. Cafeteria Order Queue – Serve students in the order they arrive.

Prompt: Generate a python code to implement Cafeteria Order Queue using Queue with methods for placing orders, processing orders, and viewing the current queue with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Cafeteria Order Queue using Queue with methods for placing orders, processing orders,
# and viewing the current queue with inline suggestions, use pydoc to generate documentation
from collections import deque
class CafeteriaOrderQueue:
    """
    A Cafeteria Order Queue is a system that allows for managing customer orders in a cafeteria setting.

    This CafeteriaOrderQueue class uses a queue (deque) to manage the order of customer requests, with methods for placing orders, processing orders, and viewing the current queue.

    Methods:
    place_order(order): Adds a new order to the end of the queue.
    process_order(): Removes and returns the next order to be processed from the front of the queue. Raises an exception if the queue is empty.
    view_queue(): Returns a list of all current orders in the queue in the order they will be processed.

    Example:
    >>> queue = CafeteriaOrderQueue()
    >>> queue.place_order("Order 1")
    >>> queue.place_order("Order 2")
    >>> queue.view_queue()
    ['Order 1', 'Order 2']
    >>> queue.process_order()
    'Order 1'
    >>> queue.view_queue()
    ['Order 2']
    """

    def __init__(self):
        """Initialize an empty cafeteria order queue."""
        self.orders = deque() # Create an empty deque to hold customer orders
    def place_order(self, order):
        """Add a new order to the end of the queue."""
        self.orders.append(order) # Use append to add a new order to the end of the deque
    def process_order(self):
        """Remove and return the next order to be processed from the front of the queue. Raises an exception if the queue is empty."""
        if not self.orders: # Check if the order queue is empty before attempting to process an order
            raise IndexError("No orders to process: The cafeteria order queue is empty.") # Raise an exception if trying to process from an empty queue
        return self.orders.popleft() # Use popleft to remove and return the next order from the front of the deque
    def view_queue(self):
        """Return a list of all current orders in the queue in the order they will be processed."""
        return list(self.orders) # Return a list representation of the current orders in the deque
# Example usage of the CafeteriaOrderQueue class
queue = CafeteriaOrderQueue() # Create a new cafeteria order queue instance
queue.place_order("Order 1") # Place a new order "Order 1" in the queue
queue.place_order("Order 2") # Place another order "Order 2" in the queue
print("Current order queue:", queue.view_queue()) # View and print the current order queue
print("Processing order:", queue.process_order()) # Process the next order and print it
print("Current order queue after processing:", queue.view_queue()) # View and print the current order queue after processing an order
```

Output:

```
Current order queue: ['Order 1', 'Order 2']
Processing order: Order 1
Current order queue after processing: ['Order 2']
Current order queue: ['Order 1', 'Order 2']
Processing order: Order 1
Current order queue: ['Order 1', 'Order 2']
Current order queue: ['Order 1', 'Order 2']
Processing order: Order 1
Current order queue after processing: ['Order 2']
```

Explanation: The CafeteriaOrderQueue uses a deque-based queue to manage customer orders in FIFO order, ensuring orders are processed fairly in the sequence they arrive. The place_order() method adds new orders to the back of the queue as customers place them.

The process_order() method removes and returns the oldest order from the front for preparation, raising an error if no orders exist.

The view_queue() method shows all pending orders in the order they'll be processed, helping staff manage kitchen workflow and customer expectations.

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

1. Shopping Cart Management – Add and remove products dynamically.

Prompt: Generate a python code to implement Shopping Cart Management using Linked List with methods for adding items, removing items, and viewing the cart contents with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Shopping Cart Management using Linked List with methods for adding items, removing items,
# and viewing the cart contents with inline suggestions, use pydoc to generate documentation
class CartItemNode:
    """
    A CartItemNode in a linked list for shopping cart management.

    Attributes:
    item_name: The name of the item in the cart.
    quantity: The quantity of the item in the cart.
    next: A reference to the next node in the linked list.

    Example:
    >>> node1 = CartItemNode("Apple", 3)
    >>> node2 = CartItemNode("Banana", 2)
    >>> node1.next = node2
    >>> node1.item_name
    'Apple'
    >>> node1.quantity
    3
    >>> node1.next.item_name
    'Banana'
    >>> node1.next.quantity
    2
    """

    def __init__(self, item_name, quantity):
        """Initialize a new cart item node with the given item name, quantity, and set next to None."""
        self.item_name = item_name # Store the name of the item in the cart
        self.quantity = quantity # Store the quantity of the item in the cart
        self.next = None # Initialize the next reference to None
class ShoppingCart:
    """
    A Shopping Cart is a system that allows for managing items that a customer intends to purchase.

    This ShoppingCart class uses a linked list to manage the items in the cart, with methods for adding items, removing items, and viewing the cart contents.

    Methods:
    add_item(item_name, quantity): Adds an item with the specified name and quantity to the cart. If the item already exists, it updates the quantity.
    remove_item(item_name): Removes an item from the cart by its name. Raises an exception if the item is not found.
    view_cart(): Returns a list of tuples containing item names and their quantities currently in the cart.

    Example:
    >>> cart = ShoppingCart()
    >>> cart.add_item("Apple", 3)
    >>> cart.add_item("Banana", 2)
    >>> cart.view_cart()
    [('Apple', 3), ('Banana', 2)]
    >>> cart.remove_item("Apple")
    >>> cart.view_cart()
    [('Banana', 2)]
    """

    def __init__(self):
        """Initialize an empty shopping cart."""
        self.head = None # Initialize the head of the linked list to None
    def add_item(self, item_name, quantity):
        """Add an item with the specified name and quantity to the cart. If the item already exists, it updates the quantity."""
        if self.head is None: # If the cart is empty, add the new item as the head of the linked list
            self.head = CartItemNode(item_name, quantity)
        else:
            current = self.head # Start from the head of the linked list
            while current is not None: # Traverse the linked list to find if the item already exists
                if current.item_name == item_name: # If the item is found, update its quantity
                    current.quantity += quantity
                    return
                if current.next is None: # If we reach the end of the list without finding the item, break out of the loop
                    break
                current = current.next # Move to the next node in the linked list
            current.next = CartItemNode(item_name, quantity) # Add a new node for the new item at the end of the linked list
    def remove_item(self, item_name):
        """Remove an item from the cart by its name. Raises an exception if the item is not found."""
        current = self.head # Start from the head of the linked list
        previous = None # Initialize a variable to keep track of the previous node
        while current is not None: # Traverse the linked list to find the item to remove
            if current.item_name == item_name: # If the item is found
                if previous is None: # If the item to remove is the head of the list
                    self.head = current.next # Update the head to the next node
                else:
                    previous.next = current.next # Bypass the current
                return # Return after removing the item
            previous = current # Update the previous node to the current node
            current = current.next # Move to the next node in the linked list
        raise ValueError("Item not found in the cart: Cannot remove an item that does not exist.") # Raise an exception if the item to remove is not found in the cart
    def view_cart(self):
        """Return a list of tuples containing item names and their quantities currently in the cart."""
        cart_contents = [] # Create an empty list to hold the cart contents
        current = self.head # Start from the head of the linked list
        while current is not None: # Traverse the linked list to gather item names and quantities
            cart_contents.append((current.item_name, current.quantity)) # Append a tuple of (item_name, quantity) to the cart contents list
            current = current.next # Move to the next node in the linked list
        return cart_contents # Return the list of items and their quantities in the cart
# Example usage of the ShoppingCart class
cart = ShoppingCart() # Create a new shopping cart instance
cart.add_item("Apple", 3) # Add 3 Apples to the cart
cart.add_item("Banana", 2) # Add 2 Bananas to the cart
print("Current cart contents:", cart.view_cart()) # View and print the current contents of the cart
cart.remove_item("Apple") # Remove Apples from the cart
print("Current cart contents after removing Apples:", cart.view_cart()) # View and print the current contents of the cart after removing Apples
print("Adding more Bananas to the cart...") # Print a message indicating that more Bananas are being added to the cart
cart.add_item("Banana", 3) # Add 3 more Bananas to the cart, which should update the quantity of Bananas in the cart
print("Current cart contents after adding more Bananas:", cart.view_cart()) # View and print the current contents of the cart after adding more Bananas
```

Output:

```
Current cart contents: [('Apple', 3), ('Banana', 2)]
Current cart contents after removing Apples: [('Banana', 2)]
Current order queue: ['Order 1', 'Order 2']
Processing order: Order 1
Current order queue after processing: ['Order 2']
Current cart contents: [('Apple', 3), ('Banana', 2)]
Current cart contents after removing Apples: [('Banana', 2)]
Adding more Bananas to the cart...
Current cart contents after adding more Bananas: [('Banana', 5)]
```

Explanation: The ShoppingCart class uses a linked list to dynamically manage items customers want to purchase, allowing efficient additions and removals as shoppers browse. The add_item() method adds new items to the cart or updates quantities if an item already exists.
The remove_item() method deletes items from the cart by unlinking nodes, while view_cart() returns all current items and quantities for checkout processing.

2. Order Processing System – Orders processed in the order they are placed.

Prompt: Generate a python code to implement Order Processing System using Queue with methods for placing orders, processing orders, and viewing the current queue with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Order Processing System using Queue with methods for placing orders, processing orders,
# and viewing the current queue with inline suggestions, use pydoc to generate documentation
from collections import deque
class OrderProcessingSystem:
    """
    An Order Processing System is a system that allows for managing customer orders in a business setting.

    This OrderProcessingSystem class uses a queue (deque) to manage the order of customer requests, with methods for placing orders, processing orders, and viewing the current queue.

    Methods:
    place_order(order): Adds a new order to the end of the queue.
    process_order(): Removes and returns the next order to be processed from the front of the queue. Raises an exception if the queue is empty.
    view_queue(): Returns a list of all current orders in the queue in the order they will be processed.

    Example:
    >>> ops = OrderProcessingSystem()
    >>> ops.place_order("Order 1")
    >>> ops.place_order("Order 2")
    >>> ops.view_queue()
    ['Order 1', 'Order 2']
    >>> ops.process_order()
    'Order 1'
    >>> ops.view_queue()
    ['Order 2']
    """

    def __init__(self):
        """Initialize an empty order processing system."""
        self.orders = deque() # Create an empty deque to hold customer orders
    def place_order(self, order):
        """Add a new order to the end of the queue."""
        self.orders.append(order) # Use append to add a new order to the end of the deque
    def process_order(self):
        """Remove and return the next order to be processed from the front of the queue. Raises an exception if the queue is empty."""
        if not self.orders: # Check if the order queue is empty before attempting to process an order
            raise IndexError("No orders to process: The order processing queue is empty.") # Raise an exception if trying to process from an empty queue
        return self.orders.popleft() # Use popleft to remove and return the next order from the front of the deque
    def view_queue(self):
        """Return a list of all current orders in the queue in the order they will be processed."""
        return list(self.orders) # Return a list representation of the current orders in the deque
# Example usage of the OrderProcessingSystem class
ops = OrderProcessingSystem() # Create a new order processing system instance
ops.place_order("Order 1") # Place a new order "Order 1" in the queue
ops.place_order("Order 2") # Place another order "Order 2" in the queue
print("Current order queue:", ops.view_queue()) # View and print the current order queue
print("Processing order:", ops.process_order()) # Process the next order and print it
print("Current order queue after processing:", ops.view_queue()) # View and print the current order queue after processing an order
```

Output:

```
Current order queue: ['Order 1', 'Order 2']
Processing order: Order 1
Current order queue after processing: ['Order 2']
```

Explanation: The OrderProcessingSystem uses a queue to manage orders from customers in the sequence received, ensuring fair processing and maintaining order history. The place_order() method adds new orders to the end of the queue. The process_order() method removes and returns the oldest order for fulfillment, throwing an error if the queue is empty. The view_queue() method displays all pending orders in the order they'll be handled, helping businesses track workflow and manage customer expectations about processing times.

3. Top-Selling Products Tracker – Products ranked by sales count.

Prompt: Generate a python code to implement Top-Selling Products Tracker using Priority Queue with methods for adding products, updating sales, and retrieving top-selling products with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Top-Selling Products Tracker using Priority Queue with methods for adding products, updating sales,
#  and retrieving top-selling products with inline suggestions, use pydoc to generate documentation
import heapq
class Product:
    """
    A Product in a top-selling products tracker using a priority queue.

    Attributes:
    name: The name of the product.
    sales: The number of sales for the product.
    Example:
    >>> product1 = Product("Product A", 100)
    >>> product2 = Product("Product B", 150)
    >>> product1.name
    'Product A'
    >>> product1.sales
    100
    >>> product2.name
    'Product B'
    >>> product2.sales
    150
    """

    def __init__(self, name, sales):
        """Initialize a new product with the given name and sales."""
        self.name = name # Store the name of the product
        self.sales = sales # Store the number of sales for the product
    def __lt__(self, other):
        """Define the less-than operator for comparing products based on sales for use in the priority queue."""
        return self.sales > other.sales # Return True if this product has more sales than the other product (for max-heap behavior)
class TopSellingProductsTracker:
    """
    A Top-Selling Products Tracker is a system that allows for tracking the top-selling products in a business.
    This TopSellingProductsTracker class uses a priority queue (heap) to manage products based on their sales, with methods for
    adding products, updating sales, and retrieving top-selling products.

    Methods:
    add_product(name, sales): Adds a new product with the specified name and sales to the tracker. If the product already exists, it updates the sales.
    update_sales(name, additional_sales): Updates the sales of an existing product by adding the additional sales. Raises an exception if the product is not found.
    get_top_selling_products(n): Returns a list of the top n selling products in descending order of sales.

    Example:
    >>> tracker = TopSellingProductsTracker()
    >>> tracker.add_product("Product A", 100)
    >>> tracker.add_product("Product B", 150)
    >>> tracker.update_sales("Product A", 50)
    >>> tracker.get_top_selling_products(1)
    [('Product A', 150)]
    >>> tracker.get_top_selling_products(2)
    [('Product A', 150), ('Product B', 150)]
    """

    def __init__(self):
        """Initialize an empty top-selling products tracker."""
        self.products = {} # Create an empty dictionary to hold products and their sales
        self.heap = [] # Create an empty list to be used as a heap for tracking top-selling products
    def add_product(self, name, sales):
        """Add a new product with the specified name and sales to the tracker. If the product already exists, it updates the sales."""
        if name in self.products: # Check if the product already exists in the tracker
            self.update_sales(name, sales) # If it exists, update its sales
        else:
            product = Product(name, sales) # Create a new product instance
            self.products[name] = product # Add the product to the products dictionary
            heapq.heappush(self.heap, product) # Push the new product onto the heap to maintain the priority queue
    def update_sales(self, name, additional_sales):
        """Update the sales of an existing product by adding the additional sales. Raises an exception if the product is not found."""
        if name not in self.products: # Check if the product is not found in the tracker
            raise ValueError("Product not found: Cannot update sales for a product that does not exist.") # Raise an exception if the product is not found
        product = self.products[name] # Get the existing product from the products dictionary
        product.sales += additional_sales # Update the sales of the product by adding the additional sales
        heapq.heapify(self.heap) # Re-heapify the heap to maintain the correct order after updating sales
    def get_top_selling_products(self, n):
        """Return a list of the top n selling products in descending order of sales."""
        return [(product.name, product.sales) for product in heapq.nsmallest(n, self.heap)] # Use nsmallest to get the top n products from the heap and return their names and sales as a list of tuples
# Example usage of the TopSellingProductsTracker class
tracker = TopSellingProductsTracker() # Create a new top-selling products tracker instance
tracker.add_product("Product A", 100) # Add a new product "Product A" with 100 sales to the tracker
tracker.add_product("Product B", 150) # Add a new product "Product B" with 150 sales to the tracker
tracker.update_sales("Product A", 50) # Update the sales of "Product A" by adding 50 additional sales
print("Top selling product:", tracker.get_top_selling_products(1)) # Get and print the top selling product
print("Top 2 selling products:", tracker.get_top_selling_products(2)) # Get and print the top 2 selling products
```

Output:

```
Top selling product: [('Product A', 150)]
Top 2 selling products: [('Product A', 150), ('Product B', 150)]
```

Explanation: The TopSellingProductsTracker uses a max-heap (priority queue) to efficiently identify and retrieve .The add_product() method inserts new products into the tracker or updates existing sales numbers. The update_sales() method increments sales for products and rebalances the heap to maintain correct ordering.
The get_top_selling_products() method quickly retrieves the top n selling products without having to sort the entire list.

4. Product Search Engine – Fast lookup of products using product ID.

Prompt: Generate a python code to implement Product Search Engine using Hash Table with methods for adding products, searching for products, and retrieving product details with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Product Search Engine using Hash Table with methods for adding products, searching for products,
# and retrieving product details with inline suggestions, use pydoc to generate documentation
class Product:
    """
    A Product in a product search engine using a hash table.
    Attributes:
    name: The name of the product.
    price: The price of the product.
    description: A brief description of the product.
    Example:
    >>> product1 = Product("Laptop", 999.99, "A high-performance laptop for gaming and work.")
    >>> product2 = Product("Smartphone", 499.99, "A sleek smartphone with a powerful camera.")
    >>> product1.name
    'Laptop'
    >>> product1.price
    999.99
    >>> product1.description
    'A high-performance laptop for gaming and work.'
    >>> product2.name
    'Smartphone'
    >>> product2.price
    499.99
    >>> product2.description
    'A sleek smartphone with a powerful camera.'
    """
    def __init__(self, name, price, description):
        """Initialize a new product with the given name, price, and description."""
        self.name = name # Store the name of the product
        self.price = price # Store the price of the product
        self.description = description # Store a brief description of the product
class ProductSearchEngine:
    """
    A Product Search Engine is a system that allows for managing and searching for products in a business setting.
    This ProductSearchEngine class uses a hash table (dictionary) to manage products based on their names, with methods for adding products, searching for products, and retrieving product details.
    Methods:
    add_product(name, price, description): Adds a new product with the specified name, price, and description to the search engine. If the product already exists, it updates the price and description.
    search_product(name): Returns True if a product with the specified name exists in the search engine, False otherwise.
    get_product_details(name): Returns a tuple containing the price and description of the product with the specified name. Raises an exception if the product is not found.
    Example:
    >>> search_engine = ProductSearchEngine()
    >>> search_engine.add_product("Laptop", 999.99, "A high-performance laptop for gaming and work.")
    >>> search_engine.add_product("Smartphone", 499.99, "A sleek smartphone with a powerful camera.")
    >>> search_engine.search_product("Laptop")
    True
    >>> search_engine.search_product("Tablet")
    False
    >>> search_engine.get_product_details("Smartphone")
    (499.99, 'A sleek smartphone with a powerful camera.')
    >>> search_engine.get_product_details("Tablet")
    ValueError: Product not found: Cannot retrieve details for a product that does not exist.
    """
    def __init__(self):
        """Initialize an empty product search engine."""
        self.products = {} # Create an empty dictionary to hold products and their details
    def add_product(self, name, price, description):
        """Add a new product with the specified name, price, and description to the search engine. If the product already exists, it updates the price and description."""
        self.products[name] = Product(name, price, description) # Add or update the product in the products dictionary using the product name as the key
    def search_product(self, name):
        """Return True if a product with the specified name exists in the search engine, False otherwise."""
        return name in self.products # Check if the product name exists in the products dictionary and return True or False accordingly
    def get_product_details(self, name):
        """Return a tuple containing the price and description of the product with the specified name. Raises an exception if the product is not found."""
        if name not in self.products: # Check if the product name is not found in the products dictionary
            raise ValueError("Product not found: Cannot retrieve details for a product that does not exist.") # Raise an exception if the product is not found
        product = self.products[name] # Get the product from the products dictionary
        return (product.price, product.description) # Return a tuple of (price, description) for the requested product
# Example usage of the ProductSearchEngine class
search_engine = ProductSearchEngine() # Create a new product search engine instance
search_engine.add_product("Laptop", 999.99, "A high-performance laptop for gaming and work.") # Add a new product "Laptop" to the search engine
search_engine.add_product("Smartphone", 499.99, "A sleek smartphone with a powerful camera.") # Add a new product "Smartphone" to the search engine
print("Is 'Laptop' available?", search_engine.search_product("Laptop")) # Search for "Laptop" and print the result
print("Is 'Tablet' available?", search_engine.search_product("Tablet")) # Search for "  Tablet" and print the result
print("Details of 'Smartphone':", search_engine.get_product_details("Smartphone")) # Get and print the details of "Smartphone"
try:
    print("Details of 'Tablet':", search_engine.get_product_details("Tablet")) # Attempt to get and print the details of "Tablet", which does not exist and should raise an exception
except ValueError as e:
    print(e) # Print the exception message if "Tablet" is not found in the search engine
```

Output:

```
Is 'Laptop' available? True
Is 'Tablet' available? False
Details of 'Smartphone': (499.99, 'A sleek smartphone with a powerful camera.')
Product not found: Cannot retrieve details for a product that does not exist.
```

Explanation: The ProductSearchEngine uses a hash table to enable quick searches for products by name and retrieve their details like price and description. The Product class stores individual product information including name, price, and description. The add_product() method indexes products by name for fast lookup and updates existing products with new prices or descriptions. The search_product() method instantly checks if a product exists in the inventory. The get_product_details() method returns price and description information for a specified product, throwing an error if the product isn't found.

5. Delivery Route Planning – Connect warehouses and delivery locations.

Prompt: Generate a python code to implement Delivery Route Planning using Graph with methods for adding routes, finding the shortest route between two locations, and generating a delivery schedule with inline suggestions, use pydoc to generate documentation

Code:

```python
#Generate a python code to implement Delivery Route Planning using Graph with methods for adding routes, finding the shortest route between two locations,
# and generating a delivery schedule with inline suggestions, use pydoc to generate documentation
class LocationNode:
    """
    A LocationNode in a graph for delivery route planning.

    Attributes:
    name: The name of the location.
    neighbors: A dictionary where keys are neighboring locations and values are the distances to those locations.

    Example:
    >>> node1 = LocationNode("Location A")
    >>> node2 = LocationNode("Location B")
    >>> node3 = LocationNode("Location C")
    >>> node1.neighbors[node2] = 5
    >>> node1.neighbors[node3] = 10
    >>> node1.name
    'Location A'
    >>> node1.neighbors[node2]
    5
    >>> node1.neighbors[node3]
    10
    """

    def __init__(self, name):
        """Initialize a new location node with the given name and an empty dictionary of neighbors."""
        self.name = name # Store the name of the location in the node
        self.neighbors = {} # Initialize an empty dictionary to hold neighboring locations and their distances
class DeliveryRoutePlanning:
    """
    A Delivery Route Planning system is a system that allows for managing delivery routes and schedules in a logistics setting.

    This DeliveryRoutePlanning class uses a graph to represent locations and routes, with methods for adding routes, finding the shortest route between two locations, and generating a delivery schedule.

    Methods:
    add_route(location1, location2, distance): Adds a delivery route between two locations with the specified distance. If the locations do not exist, they are added to the system.
    find_shortest_route(location1, location2): Returns the shortest route between two locations using Dijkstra's algorithm. Raises an exception if either location is not found.
    generate_delivery_schedule(location): Returns a list of neighboring locations and their distances from the given location. Raises an exception if the location is not found.

    Example:
    >>> planning = DeliveryRoutePlanning()
    >>> planning.add_route("Location A", "Location B", 5)
    >>> planning.add_route("Location A", "Location C", 10)
    >>> planning.find_shortest_route("Location A", "Location C")
    (['Location A', 'Location C'], 10)
    >>> planning.generate_delivery_schedule("Location A")
    [('Location B', 5), ('Location C', 10)]
    """
```

```python
    def __init__(self):
        """Initialize an empty delivery route planning system."""
        self.locations = {} # Create an empty dictionary to hold locations and their corresponding nodes
    def add_route(self, location1, location2, distance):
        """Add a delivery route between two locations with the specified distance. If the locations do not exist, they are added to the system."""
        if location1 not in self.locations: # Check if location1 does not exist in the system
            self.locations[location1] = LocationNode(location1) # Add location1 to the system as a new location node
        if location2 not in self.locations: # Check if location2 does not exist in the system
            self.locations[location2] = LocationNode(location2) # Add location2 to the system as a new location node
        self.locations[location1].neighbors[self.locations[location2]] = distance # Add location2 as a neighbor of location1 with the specified distance
        self.locations[location2].neighbors[self.locations[location1]] = distance # Add location1 as a neighbor of location2 with the specified distance (undirected route)
    def find_shortest_route(self, location1, location2):
        """Return the shortest route between two locations using Dijkstra's algorithm. Raises an exception if either location is not found."""
        if location1 not in self.locations or location2 not in self.locations: # Check if either location1 or location2 is not found in the system
            raise ValueError("One or both locations not found in the system.") # Raise an exception if either location is not found
        # Implement Dijkstra's algorithm to find the shortest route between location1 and location2
        import heapq
        queue = [(0, self.locations[location1], [])] # Initialize a priority queue with (distance, current_node, path)
        visited = set() # Create a set to keep track of visited nodes
        while queue: # While there are nodes to process in the priority queue
            (cost, current_node, path) = heapq.heappop(queue) # Get the node with the lowest cost
            if current_node.name in visited: # If the current node has already been visited, skip it
                continue
            visited.add(current_node.name) # Mark the current node as visited
            path = path + [current_node.name] # Update the path with the current node's name
            if current_node.name == location2: # If we have reached the destination location, return the path and total cost
                return (path, cost)
            for neighbor, distance in current_node.neighbors.items(): # Iterate through neighbors of the current node
                if neighbor.name not in visited: # If the neighbor has not been visited
                    heapq.heappush(queue, (cost + distance, neighbor, path)) # Add the neighbor to the priority queue with updated cost and path
        return None # Return None if there is no route between location1 and location2
    def generate_delivery_schedule(self, location):
        """Return a list of neighboring locations and their distances from the given location. Raises an exception if the location is not found."""
        if location not in self.locations: # Check if the given location is not found in the system
            raise ValueError("Location not found in the system.") # Raise an exception if the location is not found
        neighbors = self.locations[location].neighbors # Get the neighbors of the given location
        return [(neighbor.name, distance) for neighbor, distance in neighbors.items()] # Return a list of neighboring locations and their distances from the given location
# Example usage of the DeliveryRoutePlanning class
planning = DeliveryRoutePlanning() # Create a new delivery route planning instance
planning.add_route("Location A", "Location B", 5) # Add a delivery route between "Location A" and "Location B" with a distance of 5
planning.add_route("Location A", "Location C", 10) # Add a delivery route between "Location A" and "Location C" with a distance of 10
print("Shortest route from Location A to Location C:", planning.find_shortest_route("Location A", "Location C")) # Find and print the shortest route from "Location A" to "Location C"
print("Schedule for Location A:", planning.generate_delivery_schedule("Location A")) # Generate and print the schedule for "Location A" showing neighboring locations and their distances
```

Output:

```
Product not found: Cannot retrieve details for a product that does not exist.
Shortest route from Location A to Location C: (['Location A', 'Location C'], 10)
Schedule for Location A: [('Location B', 5), ('Location C', 10)]
```

Explanation: The DeliveryRoutePlanning system uses a weighted graph to map delivery locations and distances between them.

The add_route() method creates bidirectional routes between locations with specific distances or travel times. The find_shortest_route() uses Dijkstra's algorithm to compute the most efficient path between any two locations, while generate_delivery_schedule() lists nearby locations for route planning and driver instructions.