

School of Computer Science and Artificial Intelligence**Lab Assignment # 11.1**

Program	: B. Tech (CSE)
Specialization	: -
Course Title	: AI Assisted Coding
Course Code	: 23CS002PC304
Semester	: II
Academic Session	: 2025-2026
Name of Student	: G.Maniprasad
Enrollment No.	: 2403A51L48
Batch No.	: 51
Date	: 17/02/26

Submission Starts here**Screenshots:****Task Description #1 – Stack Implementation**

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

Sample code:

class Stack:

 Pass

Prompt:

generate a Stack class with push, pop, peek, and is_empty Methods and code.

Code:

```
1  class Stack:
2      def __init__(self):
3          """Initialize an empty stack."""
4          self.items = []
5
6      def push(self, item):
7          """Add an item to the top of the stack.
8
9          Args:
10             item: The item to be added to the stack.
11
12             """
13             self.items.append(item)
14
15     def pop(self):
16         """Remove and return the top item from the stack.
17
18         Returns:
19             The top item from the stack.
20
21         Raises:
22             IndexError: If the stack is empty.
23
24             if self.is_empty():
25                 raise IndexError("Cannot pop from an empty stack")
26             return self.items.pop()
27
28     def peek(self):
29         """Return the top item from the stack without removing it.
30
31         Returns:
32             The top item from the stack.
33
34         Raises:
35             IndexError: If the stack is empty.
36
37             if self.is_empty():
38                 raise IndexError("Cannot peek at an empty stack")
39             return self.items[-1]
40
41     def is_empty(self):
42         """Check if the stack is empty.
43
44         Returns:
45             True if the stack is empty, False otherwise.
46
47             return len(self.items) == 0
48
49     def size(self):
50         """Return the number of items in the stack.
51
52         Returns:
53             The number of items in the stack.
54
55             return len(self.items)
56
57     def __str__(self):
58         """Return a string representation of the stack.
59
60         Returns:
61             A string representation of the stack.
62
63             return f"Stack({self.items})"
```

```

63
64
65 # Example usage and testing
66 if __name__ == "__main__":
67     # Create a new stack
68     stack = Stack()
69
70     # Test is_empty on empty stack
71     print(f"Is stack empty? {stack.is_empty()}")
72     print(f"Stack size: {stack.size()}")
73     print()
74
75     # Push items onto the stack
76     print("Pushing items: 10, 20, 30")
77     stack.push(10)
78     stack.push(20)
79     stack.push(30)
80     print(f"Stack: {stack}")
81     print(f"Is stack empty? {stack.is_empty()}")
82     print(f"Stack size: {stack.size()}")
83     print()
84
85     # Peek at the top item
86     print(f"Peek at top: {stack.peek()}")
87     print(f"Stack after peek: {stack}")
88     print()
89
90     # Pop items from the stack
91     print("Popping items:")
92     print(f"Popped: {stack.pop()}")
93     print(f"Stack: {stack}")
94     print(f"Popped: {stack.pop()}")
95     print(f"Stack: {stack}")
96     print(f"Popped: {stack.pop()}")
97     print(f"Stack: {stack}")
98     print()
99
100    # Test is_empty after popping all items
101    print(f"Is stack empty? {stack.is_empty()}")
102    print(f"Stack size: {stack.size()}")

```

Output:

```

Is stack empty? True
Stack size: 0

Pushing items: 10, 20, 30
Stack: Stack([10, 20, 30])
Is stack empty? False
Stack size: 3

Peek at top: 30
Stack after peek: Stack([10, 20, 30])

Popping items:
Popped: 30
Stack: Stack([10, 20])
Popped: 20
Stack: Stack([10])
Popped: 10
Stack: Stack([])

Is stack empty? True
Stack size: 0

```

PS C:\2403A51L03\3-2\AI_A_C\Cursor AI>

Task Description #2 – Queue Implementation

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

Sample Input Code:

```

class Queue:
    Pass

```

Prompt:

Code:

```

1  class Queue:
2      def __init__(self):
3          """Initialize an empty queue."""
4          self.items = []
5
6      def enqueue(self, item):
7          """Add an item to the rear of the queue.
8
9          Args:
10             item: The item to add to the queue
11
12         self.items.append(item)
13
14     def dequeue(self):
15         """Remove and return the front item from the queue.
16
17         Returns:
18             The front item of the queue
19
20         Raises:
21             IndexError: If the queue is empty
22
23         if self.is_empty():
24             raise IndexError("Cannot dequeue from an empty queue")
25         return self.items.pop(0)
26
27     def peek(self):
28         """Return the front item of the queue without removing it.
29
30         Returns:
31             The front item of the queue
32
33         Raises:
34             IndexError: If the queue is empty
35
36         if self.is_empty():
37             raise IndexError("Cannot peek at an empty queue")
38         return self.items[0]
39
40     def size(self):
41         """Return the number of items in the queue.
42
43         Returns:
44             The number of items in the queue
45
46         return len(self.items)
47
48     def is_empty(self):
49         """Check if the queue is empty.
50
51         Returns:
52             True if the queue is empty, False otherwise
53
54         return len(self.items) == 0
55
56
57 # Example usage and testing
58 if __name__ == "__main__":
59     # Create a new queue
60     q = Queue()
61     # Print initial state (Ctrl+K to generate
62     # Test enqueue
63     print("Enqueuing items: 1, 2, 3, 4, 5")
64     q.enqueue(1)
65     q.enqueue(2)
66     q.enqueue(3)
67     q.enqueue(4)
68     q.enqueue(5)
69
70     # Test size
71     print(f"Queue size: {q.size()}")
72
73     # Test peek
74     print(f"Peek at front: {q.peek()}")
75
76     # Test dequeue
77     print("Dequeueing items:")
78     while not q.is_empty():
79         print(f"  Dequeued: {q.dequeue()}, Remaining size: {q.size()}")
80
81     # Test empty queue
82     print("\nQueue is empty: {q.is_empty()}")
83
84     # Test error handling
85     try:
86         q.dequeue()
87     except IndexError as e:
88         print(f"Error caught: {e}")
89
90     try:
91         q.peek()
92     except IndexError as e:
93         print(f"Error caught: {e}")

```

Output:

```

Enqueuing items: 1, 2, 3, 4, 5
Queue size: 5
Peek at front: 1

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Enqueuing items: 1, 2, 3, 4, 5
Queue size: 5
Peek at front: 1

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Peek at front: 1

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Dequeueing items:
Dequeued: 1, Remaining size: 4
Dequeued: 2, Remaining size: 3
Dequeued: 3, Remaining size: 2
Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

Queue is empty: True
Error caught: Cannot dequeue from an empty queue
Error caught: Cannot peek at an empty queue

```

Task Description #3 – Linked List

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

Sample Input Code:

class Node:

Pass

Prompt:

generate a Singly Linked List with insert and display methods with code

Code:


```

1  class Node:
2      """Node class to represent a single node in the linked list"""
3
4      def __init__(self, data):
5          self.data = data # Data stored in the node
6          self.next = None # Reference to the next node
7
8
9  class SinglyLinkedList:
10     """Singly Linked List implementation with insert and display methods"""
11
12     def __init__(self):
13         self.head = None # Head pointer pointing to the first node
14
15     def insert(self, data):
16         """
17             Insert a new node at the end of the linked list
18
19             Args:
20                 data: The data to be inserted into the linked list
21             .....
22
23             new_node = Node(data)
24
25             # If the list is empty, make the new node the head
26             if self.head is None:
27                 self.head = new_node
28             else:
29                 # Traverse to the end of the list
30                 current = self.head
31                 while current.next is not None:
32                     current = current.next
33                 # Insert the new node at the end
34                 current.next = new_node
35
36     def insert_at_beginning(self, data):
37         """
38             Insert a new node at the beginning of the linked list
39
40             Args:
41                 data: The data to be inserted into the linked list
42             .....
43
44             new_node = Node(data)
45             new_node.next = self.head
46             self.head = new_node
47
48     def display(self):
49         """
50             Display all elements in the linked list
51             .....
52
53             if self.head is None:
54                 print("Linked List is empty")
55             return
56
57             current = self.head
58
59             elements = []
60             while current is not None:
61                 elements.append(str(current.data))
62                 current = current.next
63
64             # Display in format: data1 -> data2 -> data3 -> None
65             print(" -> ".join(elements) + " -> None")
66
67
68     # Example usage
69     if __name__ == "__main__":
70         # Create a new Linked List
71         ll = SinglyLinkedList()
72
73         # Insert some elements
74         print("Inserting elements into the linked list...")
75         ll.insert(10)
76         ll.insert(20)
77         ll.insert(30)
78         ll.insert(40)
79
80         # Display the linked List
81         print("\nlinked List contents:")
82         ll.display()
83
84         # Insert at beginning
85         print("\nInserting 5 at the beginning...")
86         ll.insert_at_beginning(5)
87         ll.display()
88
89         # Create an empty list
90         print("\nCreating an empty linked list:")
91         empty_ll = SinglyLinkedList()
92         empty_ll.display()

```

Output:

```
Inserting elements into the linked list...
Inserting elements into the linked list...
```

```
Linked List contents:
```

```
Linked List contents:
```

```
Linked List contents:
```

```
10 -> 20 -> 30 -> 40 -> None
```

```
10 -> 20 -> 30 -> 40 -> None
```

```
Inserting 5 at the beginning...
```

```
Inserting 5 at the beginning...
```

```
5 -> 10 -> 20 -> 30 -> 40 -> None
```

```
5 -> 10 -> 20 -> 30 -> 40 -> None
```

```
Creating an empty linked list:
```

```
Linked List is empty
```

```
Linked List is empty
```

Task Description #4 – Binary Search Tree (BST)

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

Sample Input Code:

```
class BST:
```

```
    pass
```

Prompt:

create a BST with insert and in-order traversal methods and code

Code:

```

1  class Node:
2      def __init__(self, key: int):
3          self.key = key
4          self.left: "Node | None" = None
5          self.right: "Node | None" = None
6
7
8  class BST:
9      def __init__(self):
10         self.root: Node | None = None
11
12     def insert(self, key: int) -> None:
13         """Insert key into the BST (duplicates go to the right)."""
14         if self.root is None:
15             self.root = Node(key)
16             return
17
18         cur = self.root
19         while True:
20             if key < cur.key:
21                 if cur.left is None:
22                     cur.left = Node(key)
23                     return
24                 cur = cur.left
25             else:
26                 if cur.right is None:
27                     cur.right = Node(key)
28                     return
29                 cur = cur.right
30
31     def inorder(self) -> list[int]:
32         """Return keys in in-order (sorted) order."""
33         result: list[int] = []
34
35         def dfs(n: Node | None) -> None:
36             if n is None:
37                 return
38             dfs(n.left)
39             result.append(n.key)
40             dfs(n.right)
41
42         dfs(self.root)
43         return result
44
45
46 if __name__ == "__main__":
47     bst = BST()
48     for x in [7, 3, 9, 1, 5, 8, 10]:
49         bst.insert(x)
50     print("In-order:", bst.inorder())

```

Output:

```
In-order: [1, 3, 5, 7, 8, 9, 10]
```

Task Description #5 – Hash Table

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

Sample Input Code:

```
class HashTable:
    pass
```

Prompt:

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

Code:

```
1  class HashTable:
2      """
3          Hash table using separate chaining (list of buckets).
4
5          Methods:
6              - insert(key, value): add/update a key
7              - search(key): return value or None if not found
8              - delete(key): remove key, return True if removed else False
9      """
10
11     def __init__(self, capacity: int = 8) -> None:
12         if capacity < 1:
13             raise ValueError("capacity must be >= 1")
14         self._capacity = capacity
15         self._buckets = [[] for _ in range(self._capacity)] # List[List[tuple[key, value]]]
16         self._size = 0
17
18     def __index(self, key) -> int:
19         return hash(key) % self._capacity
20
21     def _rehash(self, new_capacity: int) -> None:
22         old_items = []
23         for bucket in self._buckets:
24             old_items.extend(bucket)
25
26         self._capacity = new_capacity
27         self._buckets = [[] for _ in range(self._capacity)]
28         self._size = 0
29
30         for k, v in old_items:
31             self.insert(k, v)
32
33     def insert(self, key, value) -> None:
34         # Resize when load factor gets too high (simple rule-of-thumb)
35         if (self._size + 1) / self._capacity > 0.75:
36             self._rehash(self._capacity * 2)
37
38         idx = self._index(key)
39         bucket = self._buckets[idx]
40
41         for i, (k, _) in enumerate[Any](bucket):
42             if k == key:
43                 bucket[i] = (key, value) # update existing
44                 return
45
46         bucket.append((key, value))
47         self._size += 1
48
49     def search(self, key):
50         idx = self._index(key)
51         bucket = self._buckets[idx]
52         for k, v in bucket:
53             if k == key:
```

```

54             return v
55     return None
56
57     def delete(self, key) -> bool:
58         idx = self._index(key)
59         bucket = self._buckets[idx]
60
61         for i, (k, _) in enumerate[Any](bucket):
62             if k == key:
63                 bucket.pop(i)
64                 self._size -= 1
65                 return True
66
67         return False
68
69     def __len__(self) -> int:
70         return self._size
71
72     def __contains__(self, key) -> bool:
73         return self.search(key) is not None
74
75     def __repr__(self) -> str:
76         return f"HashTable(size={self._size}, capacity={self._capacity})"
77
78
79 if __name__ == "__main__":
80     ht = HashTable()
81     ht.insert("name", "Alice")
82     ht.insert("age", 20)
83     ht.insert("age", 21) # update
84
85     print(ht)           # HashTable(...)
86     print(ht.search("name")) # Alice
87     print(ht.search("age")) # 21
88     print(ht.search("x")) # None
89
90     print(ht.delete("age")) # True
91     print(ht.delete("age")) # False
92     print(len(ht))        # 1

```

Output:

```

HashTable(size=2, capacity=8)
Alice
21
None
HashTable(size=2, capacity=8)
Alice
21
None
21
None
True
False
1
True
False
1
False
1

```

Task Description #6 – Graph Representation

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



Sample Input Code:

class Graph:

```
    pass
```

Prompt:

implement a graph using an adjacency list with code

Code:

```

1  class Graph:
2      """
3          Graph implemented using an adjacency list.
4
5          - By default the graph is undirected,
6          - Set directed=True for a directed graph.
7      """
8
9
10     def __init__(self, directed: bool = False):
11         self.directed = directed
12         # adjacency list: vertex -> set of neighbor vertices
13         self.adj: dict[object, set[object]] = {}
14
15     def add_vertex(self, v: object) -> None:
16         """Add a vertex if it doesn't already exist."""
17         if v not in self.adj:
18             self.adj[v] = set()
19
20     def add_edge(self, u: object, v: object) -> None:
21         """Add an edge u -> v (and v -> u if undirected)."""
22         self.add_vertex(u)
23         self.add_vertex(v)
24         self.adj[u].add(v)
25         if not self.directed:
26             self.adj[v].add(u)
27
28     def remove_edge(self, u: object, v: object) -> None:
29         """Remove an edge u -> v (and v -> u if undirected), if present."""
30         if u in self.adj:
31             self.adj[u].discard(v)
32         if not self.directed and v in self.adj:
33             self.adj[v].discard(u)
34
35     def remove_vertex(self, v: object) -> None:
36         """Remove a vertex and all edges incident to it."""
37         if v not in self.adj:
38             return
39
40         # Remove edges from neighbors to v
41         for n in list(self.adj[v]):
42             self.remove_edge(v, n)
43
44         # In directed graphs, also remove incoming edges to v
45         if self.directed:
46             for u in self.adj:
47                 self.adj[u].discard(v)
48
49         del self.adj[v]
50
51     def neighbors(self, v: object) -> list[object]:
52         """Return neighbors of v as a sorted list when possible."""
53         if v not in self.adj:
54             return []
55         try:
56             return sorted(self.adj[v])
57         except TypeError:
58             return list(self.adj[v])
59
60     def bfs(self, start: object) -> list[object]:
61         """Breadth-first traversal order starting from start."""
62         if start not in self.adj:
63             return []
64
65         visited = {start}
66         queue = [start]
67         order: list[object] = []
68
69         while queue:
70             v = queue.pop(0)
71             for n in self.neighbors(v):
72                 if n not in visited:
73                     visited.add(n)
74                     queue.append(n)
75
76         return order
77
78     def dfs(self, start: object) -> list[object]:
79         """Depth-first traversal order starting from start."""
80         if start not in self.adj:
81             return []
82
83         visited: set[object] = set()
84         order: list[object] = []
85
86         def _visit(v: object) -> None:
87             visited.add(v)
88             order.append(v)
89             for n in self.neighbors(v):
90                 if n not in visited:
91                     _visit(n)
92
93         _visit(start)
94         return order
95
96     def __str__(self) -> str:
97         lines = []
98         for v in self.adj:
99             lines.append(f"({v}) -> {self.neighbors(v)}")
100        return "\n".join(lines)
101
102
103 if __name__ == "__main__":
104     g = Graph(directed=False) # change to True for a directed graph
105     g.add_edge("A", "B")
106     g.add_edge("A", "C")
107     g.add_edge("B", "D")
108     g.add_edge("C", "D")
109     g.add_edge("D", "E")
110
111     print("Adjacency list:")
112     print(g)
113     print()
114     print("BFS from A:", g.bfs("A"))
115     print("DFS from A:", g.dfs("A"))

```

Output:

```
Adjacency list:  
A -> ['B', 'C']  
B -> ['A', 'D']  
Adjacency list:  
A -> ['B', 'C']  
B -> ['A', 'D']  
A -> ['B', 'C']  
B -> ['A', 'D']  
B -> ['A', 'D']  
C -> ['A', 'D']  
D -> ['B', 'C', 'E']  
E -> ['D']  
  
BFS from A: ['A', 'B', 'C', 'D', 'E']  
DFS from A: ['A', 'B', 'D', 'C', 'E']  
D -> ['B', 'C', 'E']  
E -> ['D']  
  
BFS from A: ['A', 'B', 'C', 'D', 'E']  
DFS from A: ['A', 'B', 'D', 'C', 'E']  
E -> ['D']  
  
BFS from A: ['A', 'B', 'C', 'D', 'E']  
DFS from A: ['A', 'B', 'D', 'C', 'E']  
BFS from A: ['A', 'B', 'C', 'D', 'E']  
DFS from A: ['A', 'B', 'D', 'C', 'E']  
DFS from A: ['A', 'B', 'D', 'C', 'E']
```

Task Description #7 – Priority Queue

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

Sample Input Code:

```
class PriorityQueue:
```

```
    pass
```

Prompt:

implement a priority queue using Python's heapq module with code

Code:

```

1  import heapq
2  from itertools import count
3
4  class PriorityQueue:
5      """
6          Min-priority queue by default (smaller priority value = served first).
7          For max-priority behavior, push with -priority.
8      """
9
10     def __init__(self):
11         self._heap = []
12         self._seq = count(int) # tie-breaker for equal priorities (FIFO)
13
14     def push(self, item, priority: int):
15         heapq.heappush(self._heap, (priority, next(self._seq), item))
16
17     def pop(self):
18         if not self._heap:
19             raise IndexError("pop from empty PriorityQueue")
20         priority, _, item = heapq.heappop(self._heap)
21         return item, priority
22
23     def peek(self):
24         if not self._heap:
25             raise IndexError("peek from empty PriorityQueue")
26         priority, _, item = self._heap[0]
27         return item, priority
28
29     def __len__(self):
30         return len(self._heap)
31
32     def empty(self):
33         return len(self._heap) == 0
34
35 if __name__ == "__main__":
36     pq = PriorityQueue()
37     pq.push("low", 5)
38     pq.push("urgent", 1)
39     pq.push("medium", 3)
40     pq.push("also urgent (arrives later)", 1)
41
42     while not pq.empty():
43         item, pr = pq.pop()
44         print(pr, item)
45
46     # Max-priority example (bigger number = served first):
47     maxpq = PriorityQueue()
48     for item, pr in [("A", 10), ("B", 2), ("C", 10)]:
49         maxpq.push(item, -pr) # negate priority
50
51     print("max first:", maxpq.pop()) # returns (item, neg_priority)

```

Output:

```

1 urgent
1 also urgent (arrives later)
3 medium
5 low
max first: ('A', -10)

```

Task Description #8 – Deque

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

Sample Input Code:

```
class DequeDS:
```

```
    pass
```

Prompt:

implement a double-ended queue using collections.deque with code

Code:

```

1   from __future__ import annotations
2
3   from collections import deque
4   from typing import Deque, Generic, Iterator, Optional, TypeVar
5
6   T = TypeVar("T")
7
8
9   class DequeDS(Generic[T]):
10      """
11      Double-ended queue (deque) implemented using collections.deque.
12      Supports O(1) append/pop operations on both ends.
13      """
14
15      def __init__(self, items: optional[Iterator[T]] = None) -> None:
16          self._dq: Deque[T] = deque[T](items or [])
17
18      # --- Add operations ---
19      def add_front(self, item: T) -> None:
20          """Insert item at the front (left)."""
21          self._dq.appendleft(item)
22
23      def add_rear(self, item: T) -> None:
24          """Insert item at the rear (right)."""
25          self._dq.append(item)
26
27      # --- Remove operations ---
28      def remove_front(self) -> T:
29          """Remove and return the front (left) item."""
30          if self.is_empty():
31              raise IndexError("remove_front from empty deque")
32          return self._dq.popleft()
33
34      def remove_rear(self) -> T:
35          """Remove and return the rear (right) item."""
36          if self.is_empty():
37              raise IndexError("remove_rear from empty deque")
38          return self._dq.pop()
39
40      # --- Peek operations ---
41      def peek_front(self) -> T:
42          """Return the front (left) item without removing it."""
43          if self.is_empty():
44              raise IndexError("peek_front from empty deque")
45          return self._dq[0]
46
47      def peek_rear(self) -> T:
48          """Return the rear (right) item without removing it."""
49          if self.is_empty():
50              raise IndexError("peek_rear from empty deque")
51          return self._dq[-1]
```

```

52
53     # --- utility ---
54     def is_empty(self) -> bool:
55         return len(self._dq) == 0
56
57     def size(self) -> int:
58         return len(self._dq)
59
60     def clear(self) -> None:
61         self._dq.clear()
62
63     def __len__(self) -> int:
64         return len(self._dq)
65
66     def __iter__(self) -> Iterator[T]:
67         return iter(self._dq)
68
69     def __repr__(self) -> str:
70         return f"DequeDS({list[T](self._dq)!r})"
71
72
73 if __name__ == "__main__":
74     d = DequeDS[int]()
75     d.add_front(10)    # [10]
76     d.add_rear(20)    # [10, 20]
77     d.add_front(5)    # [5, 10, 20]
78     print("Deque:", d)
79     print("Front:", d.peek_front())
80     print("Rear:", d.peek_rear())
81     print("Remove front:", d.remove_front()) # 5
82     print("Remove rear:", d.remove_rear())   # 20
83     print("Deque now:", d)

```

Output:

```

Deque: DequeDS([5, 10, 20])
Front: 5
Rear: 20
Remove front: 5
Remove rear: 20
Deque now: DequeDS([10])

```

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

Prompt:

Solve this clearly and concisely.

Design a Campus Resource Management System code with:

Student Attendance Tracking

Event Registration System

Library Book Borrowing

Bus Scheduling System

Cafeteria Order Queue

Choose the best data structure for each feature from:

Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, Deque

Output as a table:

Feature | Data Structure | 2–3 sentence justification

Code:

```

1  from __future__ import annotations
2
3  from dataclasses import dataclass
4  from collections import deque
5  from typing import Deque, Dict, List, Optional, Set, Tuple
6
7  # *****
8
9  # 1) Student Attendance Tracking (Hash Table)
10
11
12 class AttendanceTracker:
13     """
14         Data structure: Hash Table (Python dict)
15         student_id -> (date:str -> present_bool)
16     """
17
18     def __init__(self) -> None:
19         self._records: Dict[student_id, Dict[date, bool]] = {}
20
21     def add_record(self, student_id: student_id, date: str, present: bool) -> None:
22         if student_id not in self._records:
23             self._records.setdefault(student_id, {}).update({date: present})
24
25     def is_present(self, student_id: student_id, date: str) -> Optional[bool]:
26         return self._records.get(student_id, {}).get(date)
27
28     def attendance_percent(self, student_id: student_id) -> float:
29         days = len(self._records.get(student_id, {}))
30         if days == 0:
31             return 0.0
32         present_count = sum(1 for v in days.values() if v)
33         return (present_count / len(days)) * 100.0
34
35
36 # *****
37
38 class EventRegistrationSystem:
39     """
40         Data structure: Queue (collections.deque)
41         FIFO registration requests + FIFO waitlist.
42     """
43
44     @dataclass(frozen=True)
45     class Event:
46         name: str
47         max: int
48         capacity: int
49
50     def __init__(self) -> None:
51         self._events: Dict[event_id, Event] = {}
52         self._confirmed: Dict[event_id, Set[event_id]] = {} # event_id -> {student_id}
53         self._request: Dict[event_id, Set[event_id]] = {} # event_id -> {student_id}
54         self._waitlist: Dict[event_id, Deque[student_id]] = {} # event_id -> deque(student_id)
55
56     def create_event(self, event_id: event_id, name: str, capacity: int) -> None:
57         if capacity < 0:
58             raise ValueError("Capacity must be > 0")
59         self._events[event_id] = self.Event(name=name, capacity=capacity)
60         self._confirmed[event_id] = set()
61         self._request[event_id] = set()
62         self._waitlist[event_id] = deque()
63
64     def request_registration(self, event_id: str, student_id: str) -> None:
65         self._ensure_event(event_id)
66         if student_id in self._confirmed[event_id]:
67             return
68         if student_id in self._request[event_id] or student_id in self._waitlist[event_id]:
69             raise ValueError("Event ID already requested")
70         self._request[event_id].append(student_id)
71
72     def process_next_request(self, event_id: str) -> Optional[event_id]:
73
74         Processes ONE pending request in FIFO order.
75         Returns the student_id that got confirmed (or None if no request).
76
77         self._ensure_event(event_id)
78         if student_id in self._request[event_id]:
79             self._confirm(event_id, student_id)
80             if not self._request[event_id]:
81                 return None
82
83             student_id = self._poplast()
84             if len(self._request[event_id]) < self._events[event_id].capacity:
85                 self._confirmed[event_id].add(student_id)
86             return student_id
87
88         self._waitlist[event_id].append(student_id)
89
90
91     def cancel_registration(self, event_id: str, student_id: str) -> None:
92         self._ensure_event(event_id)
93         if student_id in self._confirmed[event_id]:
94             self._remove_confirmation(student_id)
95             self._promote_from_waitlist(event_id)
96             return
97
98
99

```

```

101     return
102     self._remove_from_queue(self._events[event_id], student_id)
103     self._remove_from_queue(self._waitlist[event_id], student_id)
104
105     def confirmed_list(self, event_id: str) -> list[str]:
106         return sorted(self._confirmed[event_id])
107
108     def waitlist_list(self, event_id: str) -> list[str]:
109         self._ensure_event(event_id)
110
111     def _remove_from_waitlist(self, event_id: str):
112         if len(self._confirmed[event_id]) >= self._events[event_id].capacity:
113             self._remove_from_queue(self._confirmed[event_id])
114
115         w = self._waitlist[event_id]
116         while w and len(self._confirmed[event_id]) < self._events[event_id].capacity:
117             self._confirmed[event_id].append(w.pop(0))
118
119     def _remove_from_queue(self, q: deque[str], student_id: str) -> None:
120         if not q:
121             return
122
123         queue = deque([x for x in q if x != student_id])
124         q.clear()
125         q.extend(queue)
126
127     def _ensure_event(self, event_id: str) -> None:
128         if event_id not in self._events:
129             raise KeyError(f"Unknown event_id {event_id}")
130
131
132 # =====
133 # Library Book Borrowing (BT)
134 # =====
135
136 @dataclass
137 class Book:
138     id: str
139     title: str
140     total_copies: int
141     available_copies: int
142
143     class _BookBTHNode:
144         def __init__(self, key: str, value: str, book: Book) -> None:
145             self.key = key
146             self.value = value
147             self.book = book
148             self.left: Optional[_BookBTHNode] = None
149             self.right: Optional[_BookBTHNode] = None
150
151     class LibrarySystem:
152         """
153             Data structure: BST (by ISBN) for catalog/inventory search and ordered traversal.
154             Increasing increments available copies; returning increments.
155         """
156
157         def __init__(self) -> None:
158             self._root: Optional[_BookBTHNode] = None
159             self._loans: dict[tuple(str, str), int] = {} # (student_id, isbn) -> count borrowed
160
161         def add_book(self, str, title: str, copies: int = 1) -> None:
162             if copies < 0:
163                 raise ValueError("Copies must be > 0")
164
165             existing = self._find(isbn)
166             if existing:
167                 existing.total_copies += copies
168                 existing.available_copies += copies
169             else:
170                 book = Book(isbn, title, total_copies=copies, available_copies=copies)
171                 self._root = self._insert(self._root, isbn, title, book)
172
173         def find(self, isbn: str) -> Optional[Book]:
174             node = self._root
175             parent = None
176
177             if isbn == node.key:
178                 return node.book
179             node = node.left if isbn < node.key else node.right
180
181             return None
182
183         def borrow(self, student_id: str, isbn: str) -> bool:
184             book = self._find(isbn)
185             if not book or book.available_copies < 0:
186                 return False
187
188             book.available_copies -= 1
189             self._loans[(student_id, isbn)] = self._loans.get((student_id, isbn), 0) + 1
190             return True
191
192         def return_book(self, student_id: str, isbn: str) -> bool:
193             if student_id not in self._loans:
194                 if self._loans.get(student_id, 0) <> 0:
195                     return False
196
197             self._loans[student_id] -= 1
198             if self._loans.get(student_id, 0) <> 0:
199                 return False
200
201             self._loans.pop(student_id)
202
203             book = self._find(isbn)
204             if not book or book.available_copies < 0:
205                 return False
206
207             book.available_copies += 1
208             self._loans.pop((student_id, isbn))
209
210             return True
211
212         def _find(self, key: str) -> Optional[_BookBTHNode]:
213             node = self._root
214             parent = None
215
216             while node:
217                 if key == node.key:
218                     return node
219                 parent = node
220                 node = node.left if key < node.key else node.right
221
222             return parent
223
224         def _insert(self, node: Optional[_BookBTHNode], key: str, value: str, book: Book) -> None:
225             if not node:
226                 node = self._BookBTHNode(key, value, book)
227
228             if key < node.key:
229                 node.left = self._insert(node.left, key, value, book)
230             elif key > node.key:
231                 node.right = self._insert(node.right, key, value, book)
232
233             return node
234
235         def _find(self, key: str) -> Optional[_BookBTHNode]:
236             node = self._root
237             parent = None
238
239             while node:
240                 if key == node.key:
241                     return node
242                 parent = node
243                 node = node.left if key < node.key else node.right
244
245             return parent
246
247         def _find(self, key: str) -> Optional[_BookBTHNode]:
248             node = self._root
249             parent = None
250
251             while node:
252                 if key == node.key:
253                     return node
254                 parent = node
255                 node = node.left if key < node.key else node.right
256
257             return parent
258
259         def _find(self, key: str) -> Optional[_BookBTHNode]:
260             node = self._root
261             parent = None
262
263             while node:
264                 if key == node.key:
265                     return node
266                 parent = node
267                 node = node.left if key < node.key else node.right
268
269             return parent
270
271         def _find(self, key: str) -> Optional[_BookBTHNode]:
272             node = self._root
273             parent = None
274
275             while node:
276                 if key == node.key:
277                     return node
278                 parent = node
279                 node = node.left if key < node.key else node.right
280
281             return parent
282
283         def _find(self, key: str) -> Optional[_BookBTHNode]:
284             node = self._root
285             parent = None
286
287             while node:
288                 if key == node.key:
289                     return node
290                 parent = node
291                 node = node.left if key < node.key else node.right
292
293             return parent
294
295         def _find(self, key: str) -> Optional[_BookBTHNode]:
296             node = self._root
297             parent = None
298
299             while node:
300                 if key == node.key:
301                     return node
302                 parent = node
303                 node = node.left if key < node.key else node.right
304
305             return parent
306
307         def _find(self, key: str) -> Optional[_BookBTHNode]:
308             node = self._root
309             parent = None
310
311             while node:
312                 if key == node.key:
313                     return node
314                 parent = node
315                 node = node.left if key < node.key else node.right
316
317             return parent
318
319         def _find(self, key: str) -> Optional[_BookBTHNode]:
320             node = self._root
321             parent = None
322
323             while node:
324                 if key == node.key:
325                     return node
326                 parent = node
327                 node = node.left if key < node.key else node.right
328
329             return parent
330
331         def _find(self, key: str) -> Optional[_BookBTHNode]:
332             node = self._root
333             parent = None
334
335             while node:
336                 if key == node.key:
337                     return node
338                 parent = node
339                 node = node.left if key < node.key else node.right
340
341             return parent
342
343         def _find(self, key: str) -> Optional[_BookBTHNode]:
344             node = self._root
345             parent = None
346
347             while node:
348                 if key == node.key:
349                     return node
350                 parent = node
351                 node = node.left if key < node.key else node.right
352
353             return parent
354
355         def _find(self, key: str) -> Optional[_BookBTHNode]:
356             node = self._root
357             parent = None
358
359             while node:
360                 if key == node.key:
361                     return node
362                 parent = node
363                 node = node.left if key < node.key else node.right
364
365             return parent
366
367         def _find(self, key: str) -> Optional[_BookBTHNode]:
368             node = self._root
369             parent = None
370
371             while node:
372                 if key == node.key:
373                     return node
374                 parent = node
375                 node = node.left if key < node.key else node.right
376
377             return parent
378
379         def _find(self, key: str) -> Optional[_BookBTHNode]:
380             node = self._root
381             parent = None
382
383             while node:
384                 if key == node.key:
385                     return node
386                 parent = node
387                 node = node.left if key < node.key else node.right
388
389             return parent
390
391         def _find(self, key: str) -> Optional[_BookBTHNode]:
392             node = self._root
393             parent = None
394
395             while node:
396                 if key == node.key:
397                     return node
398                 parent = node
399                 node = node.left if key < node.key else node.right
400
401             return parent
402
403         def _find(self, key: str) -> Optional[_BookBTHNode]:
404             node = self._root
405             parent = None
406
407             while node:
408                 if key == node.key:
409                     return node
410                 parent = node
411                 node = node.left if key < node.key else node.right
412
413             return parent
414
415         def _find(self, key: str) -> Optional[_BookBTHNode]:
416             node = self._root
417             parent = None
418
419             while node:
420                 if key == node.key:
421                     return node
422                 parent = node
423                 node = node.left if key < node.key else node.right
424
425             return parent
426
427         def _find(self, key: str) -> Optional[_BookBTHNode]:
428             node = self._root
429             parent = None
430
431             while node:
432                 if key == node.key:
433                     return node
434                 parent = node
435                 node = node.left if key < node.key else node.right
436
437             return parent
438
439         def _find(self, key: str) -> Optional[_BookBTHNode]:
440             node = self._root
441             parent = None
442
443             while node:
444                 if key == node.key:
445                     return node
446                 parent = node
447                 node = node.left if key < node.key else node.right
448
449             return parent
450
451         def _find(self, key: str) -> Optional[_BookBTHNode]:
452             node = self._root
453             parent = None
454
455             while node:
456                 if key == node.key:
457                     return node
458                 parent = node
459                 node = node.left if key < node.key else node.right
460
461             return parent
462
463         def _find(self, key: str) -> Optional[_BookBTHNode]:
464             node = self._root
465             parent = None
466
467             while node:
468                 if key == node.key:
469                     return node
470                 parent = node
471                 node = node.left if key < node.key else node.right
472
473             return parent
474
475         def _find(self, key: str) -> Optional[_BookBTHNode]:
476             node = self._root
477             parent = None
478
479             while node:
480                 if key == node.key:
481                     return node
482                 parent = node
483                 node = node.left if key < node.key else node.right
484
485             return parent
486
487         def _find(self, key: str) -> Optional[_BookBTHNode]:
488             node = self._root
489             parent = None
490
491             while node:
492                 if key == node.key:
493                     return node
494                 parent = node
495                 node = node.left if key < node.key else node.right
496
497             return parent
498
499         def _find(self, key: str) -> Optional[_BookBTHNode]:
500             node = self._root
501             parent = None
502
503             while node:
504                 if key == node.key:
505                     return node
506                 parent = node
507                 node = node.left if key < node.key else node.right
508
509             return parent
510
511         def _find(self, key: str) -> Optional[_BookBTHNode]:
512             node = self._root
513             parent = None
514
515             while node:
516                 if key == node.key:
517                     return node
518                 parent = node
519                 node = node.left if key < node.key else node.right
520
521             return parent
522
523         def _find(self, key: str) -> Optional[_BookBTHNode]:
524             node = self._root
525             parent = None
526
527             while node:
528                 if key == node.key:
529                     return node
530                 parent = node
531                 node = node.left if key < node.key else node.right
532
533             return parent
534
535         def _find(self, key: str) -> Optional[_BookBTHNode]:
536             node = self._root
537             parent = None
538
539             while node:
540                 if key == node.key:
541                     return node
542                 parent = node
543                 node = node.left if key < node.key else node.right
544
545             return parent
546
547         def _find(self, key: str) -> Optional[_BookBTHNode]:
548             node = self._root
549             parent = None
550
551             while node:
552                 if key == node.key:
553                     return node
554                 parent = node
555                 node = node.left if key < node.key else node.right
556
557             return parent
558
559         def _find(self, key: str) -> Optional[_BookBTHNode]:
560             node = self._root
561             parent = None
562
563             while node:
564                 if key == node.key:
565                     return node
566                 parent = node
567                 node = node.left if key < node.key else node.right
568
569             return parent
570
571         def _find(self, key: str) -> Optional[_BookBTHNode]:
572             node = self._root
573             parent = None
574
575             while node:
576                 if key == node.key:
577                     return node
578                 parent = node
579                 node = node.left if key < node.key else node.right
580
581             return parent
582
583         def _find(self, key: str) -> Optional[_BookBTHNode]:
584             node = self._root
585             parent = None
586
587             while node:
588                 if key == node.key:
589                     return node
590                 parent = node
591                 node = node.left if key < node.key else node.right
592
593             return parent
594
595         def _find(self, key: str) -> Optional[_BookBTHNode]:
596             node = self._root
597             parent = None
598
599             while node:
600                 if key == node.key:
601                     return node
602                 parent = node
603                 node = node.left if key < node.key else node.right
604
605             return parent
606
607         def _find(self, key: str) -> Optional[_BookBTHNode]:
608             node = self._root
609             parent = None
610
611             while node:
612                 if key == node.key:
613                     return node
614                 parent = node
615                 node = node.left if key < node.key else node.right
616
617             return parent
618
619         def _find(self, key: str) -> Optional[_BookBTHNode]:
620             node = self._root
621             parent = None
622
623             while node:
624                 if key == node.key:
625                     return node
626                 parent = node
627                 node = node.left if key < node.key else node.right
628
629             return parent
630
631         def _find(self, key: str) -> Optional[_BookBTHNode]:
632             node = self._root
633             parent = None
634
635             while node:
636                 if key == node.key:
637                     return node
638                 parent = node
639                 node = node.left if key < node.key else node.right
640
641             return parent
642
643         def _find(self, key: str) -> Optional[_BookBTHNode]:
644             node = self._root
645             parent = None
646
647             while node:
648                 if key == node.key:
649                     return node
650                 parent = node
651                 node = node.left if key < node.key else node.right
652
653             return parent
654
655         def _find(self, key: str) -> Optional[_BookBTHNode]:
656             node = self._root
657             parent = None
658
659             while node:
660                 if key == node.key:
661                     return node
662                 parent = node
663                 node = node.left if key < node.key else node.right
664
665             return parent
666
667         def _find(self, key: str) -> Optional[_BookBTHNode]:
668             node = self._root
669             parent = None
670
671             while node:
672                 if key == node.key:
673                     return node
674                 parent = node
675                 node = node.left if key < node.key else node.right
676
677             return parent
678
679         def _find(self, key: str) -> Optional[_BookBTHNode]:
680             node = self._root
681             parent = None
682
683             while node:
684                 if key == node.key:
685                     return node
686                 parent = node
687                 node = node.left if key < node.key else node.right
688
689             return parent
690
691         def _find(self, key: str) -> Optional[_BookBTHNode]:
692             node = self._root
693             parent = None
694
695             while node:
696                 if key == node.key:
697                     return node
698                 parent = node
699                 node = node.left if key < node.key else node.right
700
701             return parent
702
703         def _find(self, key: str) -> Optional[_BookBTHNode]:
704             node = self._root
705             parent = None
706
707             while node:
708                 if key == node.key:
709                     return node
710                 parent = node
711                 node = node.left if key < node.key else node.right
712
713             return parent
714
715         def _find(self, key: str) -> Optional[_BookBTHNode]:
716             node = self._root
717             parent = None
718
719             while node:
720                 if key == node.key:
721                     return node
722                 parent = node
723                 node = node.left if key < node.key else node.right
724
725             return parent
726
727         def _find(self, key: str) -> Optional[_BookBTHNode]:
728             node = self._root
729             parent = None
730
731             while node:
732                 if key == node.key:
733                     return node
734                 parent = node
735                 node = node.left if key < node.key else node.right
736
737             return parent
738
739         def _find(self, key: str) -> Optional[_BookBTHNode]:
740             node = self._root
741             parent = None
742
743             while node:
744                 if key == node.key:
745                     return node
746                 parent = node
747                 node = node.left if key < node.key else node.right
748
749             return parent
750
751         def _find(self, key: str) -> Optional[_BookBTHNode]:
752             node = self._root
753             parent = None
754
755             while node:
756                 if key == node.key:
757                     return node
758                 parent = node
759                 node = node.left if key < node.key else node.right
760
761             return parent
762
763         def _find(self, key: str) -> Optional[_BookBTHNode]:
764             node = self._root
765             parent = None
766
767             while node:
768                 if key == node.key:
769                     return node
770                 parent = node
771                 node = node.left if key < node.key else node.right
772
773             return parent
774
775         def _find(self, key: str) -> Optional[_BookBTHNode]:
776             node = self._root
777             parent = None
778
779             while node:
780                 if key == node.key:
781                     return node
782                 parent = node
783                 node = node.left if key < node.key else node.right
784
785             return parent
786
787         def _find(self, key: str) -> Optional[_BookBTHNode]:
788             node = self._root
789             parent = None
790
791             while node:
792                 if key == node.key:
793                     return node
794                 parent = node
795                 node = node.left if key < node.key else node.right
796
797             return parent
798
799         def _find(self, key: str) -> Optional[_BookBTHNode]:
800             node = self._root
801             parent = None
802
803             while node:
804                 if key == node.key:
805                     return node
806                 parent = node
807                 node = node.left if key < node.key else node.right
808
809             return parent
810
811         def _find(self, key: str) -> Optional[_BookBTHNode]:
812             node = self._root
813             parent = None
814
815             while node:
816                 if key == node.key:
817                     return node
818                 parent = node
819                 node = node.left if key < node.key else node.right
820
821             return parent
822
823         def _find(self, key: str) -> Optional[_BookBTHNode]:
824             node = self._root
825             parent = None
826
827             while node:
828                 if key == node.key:
829                     return node
830                 parent = node
831                 node = node.left if key < node.key else node.right
832
833             return parent
834
835         def _find(self, key: str) -> Optional[_BookBTHNode]:
836             node = self._root
837             parent = None
838
839             while node:
840                 if key == node.key:
841                     return node
842                 parent = node
843                 node = node.left if key < node.key else node.right
844
845             return parent
846
847         def _find(self, key: str) -> Optional[_BookBTHNode]:
848             node = self._root
849             parent = None
850
851             while node:
852                 if key == node.key:
853                     return node
854                 parent = node
855                 node = node.left if key < node.key else node.right
856
857             return parent
858
859         def _find(self, key: str) -> Optional[_BookBTHNode]:
860             node = self._root
861             parent = None
862
863             while node:
864                 if key == node.key:
865                     return node
866                 parent = node
867                 node = node.left if key < node.key else node.right
868
869             return parent
870
871         def _find(self, key: str) -> Optional[_BookBTHNode]:
872             node = self._root
873             parent = None
874
875             while node:
876                 if key == node.key:
877                     return node
878                 parent = node
879                 node = node.left if key < node.key else node.right
880
881             return parent
882
883         def _find(self, key: str) -> Optional[_BookBTHNode]:
884             node = self._root
885             parent = None
886
887             while node:
888                 if key == node.key:
889                     return node
890                 parent = node
891                 node = node.left if key < node.key else node.right
892
893             return parent
894
895         def _find(self, key: str) -> Optional[_BookBTHNode]:
896             node = self._root
897             parent = None
898
899             while node:
900                 if key == node.key:
901                     return node
902                 parent = node
903                 node = node.left if key < node.key else node.right
904
905             return parent
906
907         def _find(self, key: str) -> Optional[_BookBTHNode]:
908             node = self._root
909             parent = None
910
911             while node:
912                 if key == node.key:
913                     return node
914                 parent = node
915                 node = node.left if key < node.key else node.right
916
917             return parent
918
919         def _find(self, key: str) -> Optional[_BookBTHNode]:
920             node = self._root
921             parent = None
922
923             while node:
924                 if key == node.key:
925                     return node
926                 parent = node
927                 node = node.left if key < node.key else node.right
928
929             return parent
930
931         def _find(self, key: str) -> Optional[_BookBTHNode]:
932             node = self._root
933             parent = None
934
935             while node:
936                 if key == node.key:
937                     return node
938                 parent = node
939                 node = node.left if key < node.key else node.right
940
941             return parent
942
943         def _find(self, key: str) -> Optional[_BookBTHNode]:
944             node = self._root
945             parent = None
946
947             while node:
948                 if key == node.key:
949                     return node
950                 parent = node
951                 node = node.left if key < node.key else node.right
952
953             return parent
954
955         def _find(self, key: str) -> Optional[_BookBTHNode]:
956             node = self._root
957             parent = None
958
959             while node:
960                 if key == node.key:
961                     return node
962                 parent = node
963                 node = node.left if key < node.key else node.right
964
965             return parent
966
967         def _find(self, key: str) -> Optional[_BookBTHNode]:
968             node = self._root
969             parent = None
970
971             while node:
972                 if key == node.key:
973                     return node
974                 parent = node
975                 node = node.left if key < node.key else node.right
976
977             return parent
978
979         def _find(self, key: str) -> Optional[_BookBTHNode]:
980             node = self._root
981             parent = None
982
983             while node:
984                 if key == node.key:
985                     return node
986                 parent = node
987                 node = node.left if key < node.key else node.right
988
989             return parent
990
991         def _find(self, key: str) -> Optional[_BookBTHNode]:
992             node = self._root
993             parent = None
994
995             while node:
996                 if key == node.key:
997                     return node
998                 parent = node
999                 node = node.left if key < node.key else node.right
1000
1001            return parent
1002
1003        def _find(self, key: str) -> Optional[_BookBTHNode]:
1004            node = self._root
1005            parent = None
1006
1007            while node:
1008                if key == node.key:
1009                    return node
1010                parent = node
1011                node = node.left if key < node.key else node.right
1012
1013            return parent
1014
1015        def _find(self, key: str) -> Optional[_BookBTHNode]:
1016            node = self._root
1017            parent = None
1018
1019            while node:
1020                if key == node.key:
1021                    return node
1022                parent = node
1023                node = node.left if key < node.key else node.right
1024
1025            return parent
1026
1027        def _find(self, key: str) -> Optional[_BookBTHNode]:
1028            node = self._root
1029            parent = None
1030
1031            while node:
1032                if key == node.key:
1033                    return node
1034                parent = node
1035                node = node.left if key < node.key else node.right
1036
1037            return parent
1038
1039        def _find(self, key: str) -> Optional[_BookBTHNode]:
1040            node = self._root
1041            parent = None
1042
1043            while node:
1044                if key == node.key:
1045                    return node
1046                parent = node
1047                node = node.left if key < node.key else node.right
1048
1049            return parent
1050
1051        def _find(self, key: str) -> Optional[_BookBTHNode]:
1052            node = self._root
1053            parent = None
1054
1055            while node:
1056                if key == node.key:
1057                    return node
1058                parent = node
1059                node = node.left if key < node.key else node.right
1060
1061            return parent
1062
1063        def _find(self, key: str) -> Optional[_BookBTHNode]:
1064            node = self._root
1065            parent = None
1066
1067            while node:
1068                if key == node.key:
1069                    return node
1070                parent = node
1071                node = node.left if key < node.key else node.right
1072
1073            return parent
1074
1075        def _find(self, key: str) -> Optional[_BookBTHNode]:
1076            node = self._root
1077            parent = None
1078
1079            while node:
1080                if key == node.key:
1081                    return node
1082                parent = node
1083                node = node.left if key < node.key else node.right
1084
1085            return parent
1086
1087        def _find(self, key: str) -> Optional[_BookBTHNode]:
1088            node = self._root
1089            parent = None
1090
1091            while node:
1092                if key == node.key:
1093                    return node
1094                parent = node
1095                node = node.left if key < node.key else node.right
1096
1097            return parent
1098
1099        def _find(self, key: str) -> Optional[_BookBTHNode]:
1100            node = self._root
1101            parent = None
1102
1103            while node:
1104                if key == node.key:
1105                    return node
1106                parent = node
1107                node = node.left if key < node.key else node.right
1108
1109            return parent
1110
1111        def _find(self, key: str) -> Optional[_BookBTHNode]:
1112            node = self._root
1113            parent = None
1114
1115            while node:
1116                if key == node.key:
1117                    return node
1118                parent = node
1119                node = node.left if key < node.key else node.right
1120
1121            return parent
1122
1123        def _find(self, key: str) -> Optional[_BookBTHNode]:
1124            node = self._root
1125            parent = None
1126
1127            while node:
1128                if key == node.key:
1129                    return node
1130                parent = node
1131                node = node.left if key < node.key else node.right
1132
1133            return parent
1134
1135        def _find(self, key: str) -> Optional[_BookBTHNode]:
1136            node = self._root
1137            parent = None
1138
1139            while node:
1140                if key == node.key:
1141                    return node
1142                parent = node
1143                node = node.left if key < node.key else node.right
1144
1145            return parent
1146
1147        def _find(self, key: str) -> Optional[_BookBTHNode]:
1148            node = self._root
1149            parent = None
1150
1151            while node:
1152                if key == node.key:
1153                    return node
1154                parent = node
1155                node = node.left if key < node.key else node.right
1156
1157            return parent
1158
1159        def _find(self, key: str) -> Optional[_BookBTHNode]:
1160            node = self._root
1161            parent = None
1162
1163            while node:
1164                if key == node.key:
1165                    return node
1166                parent = node
1167                node = node.left if key < node.key else node.right
1168
1169            return parent
1170
1171        def _find(self, key: str) -> Optional[_BookBTHNode]:
1172            node = self._root
1173            parent = None
1174
1175            while node:
1176                if key == node.key:
1177                    return node
1178                parent = node
1179                node = node.left if key < node.key else node.right
1180
1181            return parent
1182
1183        def _find(self, key: str) -> Optional[_BookBTHNode]:
1184            node = self._root
1185            parent = None
1186
1187            while node:
1188                if key == node.key:
1189                    return node
1190                parent = node
1191                node = node.left if key < node.key else node.right
1192
1193            return parent
1194
1195        def _find(self, key: str) -> Optional[_BookBTHNode]:
1196            node = self._root
1197            parent = None
1198
1199            while node:
1200                if key == node.key:
1201                    return node
1202                parent = node
1203                node = node.left if key < node.key else node.right
1204
1205            return parent
1206
1207        def _find(self, key: str) -> Optional[_BookBTHNode]:
1208            node = self._root
1209            parent = None
1210
1211            while node:
1212                if key == node.key:
1213                    return node
1214                parent = node
1215                node = node.left if key < node.key else node.right
1216
1217            return parent
1218
1219        def _find(self, key: str) -> Optional[_BookBTHNode]:
1220            node = self._root
1221            parent = None
1222
1223            while node:
1224                if key == node.key:
1225                    return node
1226                parent = node
1227                node = node.left if key < node.key else node.right
1228
1229            return parent
```

```

196     book = self._find(idn)
197     if not book:
198         raise ValueError
199     self._loan(book) - 1
200     book.available_copies += 1
201     return book
202
203     def _insert_in_order(self) > List[Book]:
204         with List[Book] + [1]
205             self._in_order(node.root, out)
206         return out
207
208     def _insert_in_order(self, node: Optional[_BookBTHNode], token: str, book: Book) > _BookBTHNode:
209         if node is None:
210             return _BookBTHNode(token, book)
211         if node.left is None:
212             node.left = self._insert(node.left, token, book)
213         else:
214             node.right = self._insert(node.right, token, book)
215         return node
216
217     def _in_order(self, node: Optional[_BookBTHNode], out: List[Book]) > None:
218         if node is None:
219             return
220         self._in_order(node.left, out)
221         out.append(node.book)
222         self._in_order(node.right, out)
223
224
225     # 4) Bus Scheduling System (Graph)
226     # -----
227
228     class BusNetwork:
229
230         Data structure: Graph (adjacency list)
231         stop -> list of (neighbor_stop, travel_minutes).
232         Unweighted graph with edges (non-negative weights).
233         ...
234
235         def __init__(self) > None:
236             self._adj: Dict[str, List[Tuple[str, int]]] = {}
237
238         def add_stop(self, stop: str) > None:
239             self._adj[stop] = []
240
241         def add_route(self, start: str, end: str, minutes: int, bidirectional: bool = True) > None:
242             if minutes < 0:
243                 raise ValueError("minutes must be non-negative")
244             if start not in self._adj or end not in self._adj:
245                 raise KeyError(f"start or end stop not found")
246
247             dist = self._dist(start, end) + minutes
248             if bidirectional:
249                 self._adj[start].append((end, minutes))
250                 self._adj[end].append((start, minutes))
251             else:
252                 self._adj[start].append((end, minutes))
253
254         def shortest_path(self, start: str, end: str) > Tuple[int, List[str]]:
255             if start not in self._adj or end not in self._adj:
256                 raise KeyError(f"start or end stop not found")
257
258             dist = self._dist(start, end) + 1e-10
259             prev = {start: None}
260             pq: List[Tuple[int, str]] = [(0, start)]
261
262             while pq:
263                 _, cur = heappop(pq)
264                 if cur == end:
265                     break
266                 for v, w in self._adj[cur]:
267                     if v != start:
268                         if dist <= self._dist(v, w) + w:
269                             dist = self._dist(v, w) + w
270                             prev[v] = cur
271                             heappush(pq, (dist, v))
272
273             if end not in dist:
274                 return (float("inf"), [])
275
276             # Reconstruct path
277             path: List[str] = []
278             cur: Optional[str] = end
279             while cur is not None:
280                 path.append(cur)
281                 cur = prev.get(cur)
282             path.reverse()
283             return dist, path
284
285
286     # 5) Cafeteria Order Queue (Priority Queue)
287
288
289     @Dataclass(frozen=True)
290     class CafeteriaOrder:
291         id: int
292         student_id: str
293         item: str
294         priority: int
295
296         priority_id: int # Higher number => higher priority
297
298
299     class CafeteriaOrderSystem:
300
301         Data structure: Priority Queue [Heap].
302         Serve highest priority first; tie-break by arrival order.
303         ...
304
305         def __init__(self) > None:
306             self._heap: List[Tuple[int, int, CafeteriaOrder]] = []
307             self._counter = iterools.count(1)
308
309         def place_order(self, student_id: str, item: str, priority: int = 0) > CafeteriaOrder:
310             order = CafeteriaOrder(self._counter, student_id, student_id, item, priority)
311             heappush(self._heap, (priority, order))
312             return order
313
314         def serve_next(self) > Optional[CafeteriaOrder]:
315             if not self._heap:
316                 raise ValueError("No orders")
317             _, _, order = heappop(self._heap)
318             return order
319
320         def pending_count(self) > int:
321             return len(self._heap)
322
323
324         @Demo (optional)
325         @Demo (optional)
326
327
328         def main() > None:
329
330             att = AttendanceTracker()
331             att.mark("S1", "2020-02-17", True)
332             att.mark("S2", "2020-02-17", False)
333             print("Attendance S1: ", min(att.attendance_percent("S1"), 2))
334
335             events = EventRegistrationSystem()
336             events.create_event("E100", "Workshop", capacity=2)
337             att.mark("S1", "2020-02-17", True)
338             for i in range(2):
339                 events.request_registration("E100", i)
340             print("Confirmed E100: ", events.confirmed_list("E100"))
341             print("Waitlist E100: ", events.waitlist_list("E100"))
342             print("After cancel S2, confirmed: ", events.confirmed_list("E100"))
343
344             lib = LibrarySystem()
345             lib.add_book("9781446409951", "Effective Java", copies=2)
346             lib.add_book("9781446409951", "Fluent Python", copies=2)
347             print("Newer Fluent Python", lib.borrow("11", "9781446409951"))
348             print("Newer Fluent Python again", lib.borrow("12", "9781446409951"))
349             print("Older Java", lib.borrow("10", "9781446409951"))
350
351             print("Shortest bus path Hotel->library", path, "minutes:", minutes)
352
353             # Cafeteria (Priority Queue)
354             cafe = CafeteriaOrderSystem()
355             cafe.place_order("S1", "Burger", priority=0)
356             cafe.place_order("S1", "Burger", priority=1) # higher priority
357             cafe.place_order("S1", "Burger", priority=2)
358             cafe.place_order("S1", "Burger", priority=3)
359             print("Serve order 1, cafe.serve_next()")
360             print("Serve order 2, cafe.serve_next()")
361             print("Serve order 3, cafe.serve_next()")
362
363
364             if __name__ == "__main__":
365                 main()
366
367

```

Output:

```

Attendance S1 %: 50.0
Confirmed E100: ['S1', 'S2']
Waitlist E100: ['S3']
After cancel S2, confirmed: ['S1', 'S3']
Borrow Fluent Python: True
Borrow Fluent Python again: False
Catalog in order: [('9780134685991', 2), ('9781492051367', 0)]
Shortest bus path Hostel->Library: ['Hostel', 'Cafeteria', 'Library'] minutes: 9
Attendance S1 %: 50.0
Confirmed E100: ['S1', 'S2']
Waitlist E100: ['S3']
After cancel S2, confirmed: ['S1', 'S3']
Borrow Fluent Python: True
Borrow Fluent Python again: False
Catalog in order: [('9780134685991', 2), ('9781492051367', 0)]
Shortest bus path Hostel->Library: ['Hostel', 'Cafeteria', 'Library'] minutes: 9
Waitlist E100: ['S3']
After cancel S2, confirmed: ['S1', 'S3']
Borrow Fluent Python: True
Borrow Fluent Python again: False
Catalog in order: [('9780134685991', 2), ('9781492051367', 0)]
Shortest bus path Hostel->Library: ['Hostel', 'Cafeteria', 'Library'] minutes: 9
Borrow Fluent Python again: False
Catalog in order: [('9780134685991', 2), ('9781492051367', 0)]
Shortest bus path Hostel->Library: ['Hostel', 'Cafeteria', 'Library'] minutes: 9
Serve order: CafeteriaOrder(order_id=2, student_id='S2', item='Coffee', priority=2)
Shortest bus path Hostel->Library: ['Hostel', 'Cafeteria', 'Library'] minutes: 9
Serve order: CafeteriaOrder(order_id=2, student_id='S2', item='Coffee', priority=2)
Serve order: CafeteriaOrder(order_id=2, student_id='S2', item='Coffee', priority=2)
Serve order: CafeteriaOrder(order_id=3, student_id='S3', item='Burger', priority=1)
Serve order: CafeteriaOrder(order_id=3, student_id='S3', item='Burger', priority=1)

```

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

Prompt:

Solve this clearly and concisely.

Design a Smart E-Commerce Platform with:

Shopping Cart Management – Add/remove products dynamically

Order Processing System – Process orders in placement order

Top-Selling Products Tracker – Rank products by sales count

Product Search Engine – Fast lookup using product ID

Delivery Route Planning – Connect warehouses and delivery locations

Choose the most appropriate data structure for each feature from:

Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, Deque

Output as a table:

Feature | Data Structure | 2–3 sentence justification

Code:

```

1  from collections import deque
2  import heapq
3  from typing import Dict, List, Tuple, Optional
4
5
6  # Product model
7  # -----
8  class Product:
9      def __init__(self, product_id: int, name: str, price: float):
10         self.id = product_id
11         self.name = name
12         self.price = price
13
14     def __repr__(self):
15         return f"Product(id={self.id}, name='{self.name}', price={self.price})"
16
17
18 # Product Search Engine (Hash Table)
19 # -----
20 class ProductSearchEngine:
21     def __init__(self):
22         # Hash Table: product_id -> Product
23         self.products: Dict[int, Product] = {}
24
25     def add_product(self, product: Product):
26         self.products[product.id] = product
27
28     def get_product(self, product_id: int) -> Optional[Product]:
29         return self.products.get(product_id)
30
31     def remove_product(self, product_id: int):
32         self.products.pop(product_id, None)
33
34
35 # Shopping Cart (Linked List)
36 # -----
37 class CartNode:
38     def __init__(self, product: Product, quantity: int):
39         self.product = product
40         self.quantity = quantity
41         self.next: Optional["CartNode"] = None
42
43
44 class ShoppingCart:
45     def __init__(self):
46         self.head: Optional[CartNode] = None
47
48     def add_product(self, product: Product, quantity: int = 1):
49         """
50             If product already exists in the list, increase quantity.
51             Otherwise, add new node at the front (O(1) insertion).
52         """
53         node = self.head
54         while node:
55             if node.product.id == product.id:
56                 node.quantity += quantity
57                 return
58             node = node.next
59
60         new_node = CartNode(product, quantity)
61         new_node.next = self.head
62         self.head = new_node
63
64     def remove_product(self, product_id: int, quantity: int = None):
65         """
66             Remove some or all quantity of a product.
67             If quantity is None or reaches 0, remove the node.
68         """
69         prev = None
70         node = self.head
71
72         while node:
73             if node.product.id == product_id:
74                 if quantity is None or node.quantity <= quantity:
75                     if prev:
76                         prev.next = node.next
77                     else:
78                         self.head = node.next
79
80             node = node.next
81
82

```

```

93         else:
94             node.quantity -= quantity
95
96     prev = node
97     node = node.next
98
99     def list_items(self) -> List[Tuple[Product, int]]:
100         result = []
101         node = self.head
102         while node:
103             result.append((node.product, node.quantity))
104             node = node.next
105
106         def total_price(self) -> float:
107             return sum(node.product.price * node.quantity
108                         for node in self._iter_nodes())
109
110         def _iter_nodes(self):
111             node = self.head
112             while node:
113                 yield node
114                 node = node.next
115
116             # -----
117             # Order Processing System (Queue)
118             # -----
119             class Order:
120                 _next_id = 1
121
122                 def __init__(self, cart_snapshot: List[Tuple[Product, int]]):
123                     self.id = Order._next_id
124                     Order._next_id += 1
125                     self.items = cart_snapshot # list of (Product, quantity)
126
127                 def __repr__(self):
128                     return f"Order(id={self.id}, items=[({p.id}, q) for p, q in self.items])"
129
130             class OrderProcessingSystem:
131                 def __init__(self):
132                     # Queue of orders (FIFO)
133                     self.queue: Queue[Order] = deque(Order())
134
135                 def place_order(self, cart: ShoppingCart) -> Order:
136                     order = Order(cart.list_items())
137                     self.queue.append(order)
138                     return order
139
140                 def process_next_order(self) -> Optional[Order]:
141                     if not self.queue:
142                         return None
143                     return self.queue.popleft()
144
145                 def pending_orders(self) -> int:
146                     return len(self.queue)
147
148             # -----
149             # Top-Selling Products Tracker (Priority Queue / Max-Heap)
150             # -----
151             class TopSellingProductsTracker:
152                 def __init__(self):
153                     # product_id -> sales_count
154                     self.sales: dict[int, int] = {}
155                     # product_id -> (-sales_count, product_id)
156                     self.heap: list[Tuple[int, int]] = []
157
158                 def record_sale(self, product_id: int, quantity: int = 1):
159                     neg_sales, pid = heappush(self.heap, (-self.sales.get(pid, 0),
160 # Puts the negative of sales into the heap (with a negative sign on pop)
161                     heappush(self.heap, (-self.sales.get(product_id), product_id)))
162
163                 def top_k(self, k: int) -> List[Tuple[int, int]]:
164                     """ Returns list of (product_id, sales_count) for top k products.
165                     Uses lazy removal from the heap to keep it consistent.
166                     """
167
168                     result = []
169                     seen = set[Any]()
170
171                     while self.heap and len(result) < k:
172                         neg_sales, pid = heappop(self.heap)
173                         current_sales = self.sales.get(pid, 0)
174
175                         if current_sales == -neg_sales and pid not in seen:
176                             result.append((pid, current_sales))
177                             seen.add(pid)
178
179                     # push back the elements we popped that are still valid
180                     for pid in seen:
181                         heappush(self.heap, (-self.sales.get(pid), pid))
182
183                     return result
184
185             # -----
186             # Delivery Route Planning (Graph + Dijkstra)
187             # -----
188             class DeliveryRoutePlanner:
189                 def __init__(self):
190                     # adjacency list: node -> list of (neighbor, distance)
191                     self.graph: Dict[str, List[Tuple[str, float]]] = {}
192
193                 def add_location(self, name: str):
194                     if name not in self.graph:
195                         self.graph[name] = []
196
197                 def add_route(self, from_loc: str, to_loc: str, distance: float, bidirectional: bool = True):
198                     self.add_location(from_loc)
199                     self.add_location(to_loc)
200                     self.graph[from_loc].append((to_loc, distance))
201                     if bidirectional:
202                         self.graph[to_loc].append((from_loc, distance))
203
204                 def shortest_path(self, start: str, end: str) -> Tuple[float, List[str]]:
205                     """ Dijkstra's algorithm returns (distance, path).
206                     Distance is float("inf") if no path exists.
207                     """
208                     if start not in self.graph or end not in self.graph:
209                         return float("inf"), []
210
211                     # min-heap: (distance, node, path)
212                     heap = [(0.0, start, [start])]
213                     visited = set[bool]()
214
215                     while heap:
216                         dist, node, path = heappop(heap)
217                         if node in visited:
218                             continue
219                         visited.add(node)
220
221                         if node == end:
222                             return dist, path
223
224                         for neighbor, weight in self.graph[node]:
225                             if neighbor not in visited:
226                                 heappush(heap, (dist + weight, neighbor, path + [neighbor]))
227
228                     return float("inf"), []
229
230             # -----
231             # Example usage
232             # -----
233             if __name__ == "__main__":
234                 search_engine = ProductSearchEngine()
235                 p1 = Product(1, "Laptop", 1000.0)
236                 p2 = Product(2, "Phone", 500.0)
237                 p3 = Product(3, "Tablet", 800.0)
238                 for p in (p1, p2, p3):
239                     search_engine.add_product(p)

```

```

162             result = []
163             seen = set[Any]()
164
165             while self.heap and len(result) < k:
166                 neg_sales, pid = heappop(self.heap)
167                 current_sales = self.sales.get(pid, 0)
168
169                 if current_sales == -neg_sales and pid not in seen:
170                     result.append((pid, current_sales))
171                     seen.add(pid)
172
173             # push back the elements we popped that are still valid
174             for pid in seen:
175                 heappush(self.heap, (-self.sales.get(pid), pid))
176
177             return result
178
179         # -----
180         # Delivery Route Planning (Graph + Dijkstra)
181         # -----
182         class DeliveryRoutePlanner:
183             def __init__(self):
184                 # adjacency list: node -> list of (neighbor, distance)
185                 self.graph: Dict[str, List[Tuple[str, float]]] = {}
186
187                 def add_location(self, name: str):
188                     if name not in self.graph:
189                         self.graph[name] = []
190
191                 def add_route(self, from_loc: str, to_loc: str, distance: float, bidirectional: bool = True):
192                     self.add_location(from_loc)
193                     self.add_location(to_loc)
194                     self.graph[from_loc].append((to_loc, distance))
195                     if bidirectional:
196                         self.graph[to_loc].append((from_loc, distance))
197
198                 def shortest_path(self, start: str, end: str) -> Tuple[float, List[str]]:
199                     """ Dijkstra's algorithm returns (distance, path).
200                     Distance is float("inf") if no path exists.
201                     """
202                     if start not in self.graph or end not in self.graph:
203                         return float("inf"), []
204
205                     # min-heap: (distance, node, path)
206                     heap = [(0.0, start, [start])]
207                     visited = set[bool]()
208
209                     while heap:
210                         dist, node, path = heappop(heap)
211                         if node in visited:
212                             continue
213                         visited.add(node)
214
215                         if node == end:
216                             return dist, path
217
218                         for neighbor, weight in self.graph[node]:
219                             if neighbor not in visited:
220                                 heappush(heap, (dist + weight, neighbor, path + [neighbor]))
221
222                     return float("inf"), []
223
224             # -----
225             # Example usage
226             # -----
227             if __name__ == "__main__":
228                 search_engine = ProductSearchEngine()
229                 p1 = Product(1, "Laptop", 1000.0)
230                 p2 = Product(2, "Phone", 500.0)
231                 p3 = Product(3, "Tablet", 800.0)
232                 for p in (p1, p2, p3):
233                     search_engine.add_product(p)

```

```

239 # Shopping cart
240 cart = ShoppingCart()
241 cart.add_product(search_engine.get_product(1), 1)
242 cart.add_product(search_engine.get_product(2), 2)
243 cart.add_product(search_engine.get_product(3), 3)
244 cart.remove_product(3, 1) # remove 1 headphone
245
246 print("Cart items:", cart.list_items())
247 print("Total price:", cart.total_price())
248
249 # Order processing
250 ops = OrderProcessingSystem()
251 order1 = ops.place_order(cart)
252 print("Placed order:", order1)
253 print("Pending orders:", ops.pending_orders())
254 processed = ops.process_next_order()
255 print("Processed order:", processed)
256 print("Pending orders:", ops.pending_orders())
257
258 # Top-selling products
259 tracker = TopSellingProductsTracker()
260 tracker.record_sale(1, 10) # Laptop sold 10
261 tracker.record_sale(2, 5) # Phone sold 5
262 tracker.record_sale(3, 7) # Headphones sold 7
263 print("Top 2 products (id, sales):", tracker.top_k(2))
264
265 # Delivery route planner
266 planner = DeliveryRoutePlanner()
267 planner.add_route("WarehouseA", "City1", 10.0)
268 planner.add_route("WarehouseA", "City2", 20.0)
269 planner.add_route("City1", "City2", 5.0)
270 planner.add_route("City2", "City3", 7.0)
271
272 dist, path = planner.shortest_path("WarehouseA", "City3")
273 print("Shortest route WarehouseA -> City3: [", path, "] distance:", dist)

```

Output:

```

Cart items: [(Product(id=3, name='Headphones', price=100.0), 2), (Product(id=2, name='Phone', price=500.0), 2), (Product(id=1, name='Laptop', price=1000.0), 1)]
Total price: 2200.0
Placed order: Order(id=1, items=[(3, 2), (2, 2), (1, 1)])
Pending orders: 1
Processed order: Order(id=1, items=[(3, 2), (2, 2), (1, 1)])
Pending orders: 0
Top 2 products (id, sales): [(1, 10), (3, 7)]
Shortest route WarehouseA -> City3: ['WarehouseA', 'City1', 'City2', 'City3'] distance: 22.0
PS C:\2403A51\03\3-2\AI_A_C\cursor AI>

Total price: 2200.0
Placed order: Order(id=1, items=[(3, 2), (2, 2), (1, 1)])
Pending orders: 1
Processed order: Order(id=1, items=[(3, 2), (2, 2), (1, 1)])
Pending orders: 0
Top 2 products (id, sales): [(1, 10), (3, 7)]
Shortest route WarehouseA -> City3: ['WarehouseA', 'City1', 'City2', 'City3'] distance: 22.0
Pending orders: 0
Top 2 products (id, sales): [(1, 10), (3, 7)]
Shortest route WarehouseA -> City3: ['WarehouseA', 'City1', 'City2', 'City3'] distance: 22.0
Shortest route WarehouseA -> City3: ['WarehouseA', 'City1', 'City2', 'City3'] distance: 22.0

```