

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	: I.Sathwik Rajeshwara Chary
Enrollment No.	: 2403A51L03
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  # Example usage and testing
65  if __name__ == "__main__":
66      # Create a new stack
67      stack = Stack()
68
69      # Test is_empty on empty stack
70      print(f"Is stack empty? {stack.is_empty()}")
71      print(f"Stack size: {stack.size()}")
72      print()
73
74      # Push items onto the stack
75      print("Pushing items: 10, 20, 30")
76      stack.push(10)
77      stack.push(20)
78      stack.push(30)
79      print(f"Stack: {stack}")
80      print(f"Is stack empty? {stack.is_empty()}")
81      print(f"Stack size: {stack.size()}")
82      print()
83
84      # Peek at the top item
85      print(f"Peek at top: {stack.peek()}")
86      print(f"Stack after peek: {stack}")
87      print()
88
89      # Pop items from the stack
90      print("Popping items:")
91      print(f"Popped: {stack.pop()}")
92      print(f"Stack: {stack}")
93      print(f"Popped: {stack.pop()}")
94      print(f"Stack: {stack}")
95      print(f"Popped: {stack.pop()}")
96      print(f"Stack: {stack}")
97      print(f"Popped: {stack.pop()}")
98      print(f"Stack: {stack}")
99      print()
100
101      # Test is_empty after popping all items
102      print(f"Is stack empty? {stack.is_empty()}")
103      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

```

Task Description #2 – Queue Implementation

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

Sample Input Code:

class Queue:

Pass

Prompt:

complete the queue class with enqueue, dequeue, peek, and size methods and code

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     Ctrl+L to chat Ctrl+K to generate
62
63     # Test enqueue
64     print("Enqueuing items: 1, 2, 3, 4, 5")
65     q.enqueue(1)
66     q.enqueue(2)
67     q.enqueue(3)
68     q.enqueue(4)
69     q.enqueue(5)
70
71     # Test size
72     print(f"\nQueue size: {q.size()}")
73
74     # Test peek
75     print(f"\nPeek at front: {q.peek()}")
76
77     # Test dequeue
78     print("\nDequeueing items:")
79     while not q.is_empty():
80         print(f"\tDequeued: {q.dequeue()}, Remaining size: {q.size()}")
81
82     # Test empty queue
83     print(f"\nQueue is empty: {q.is_empty()}")
84
85     # Test error handling
86     try:
87         q.dequeue()
88     except IndexError as e:
89         print(f"\nError caught: {e}")
90
91     try:
92         q.peek()
93     except IndexError as e:
94         print(f"\nError 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
5

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

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

Dequeued: 4, Remaining size: 1
Dequeued: 5, Remaining size: 0

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      def __init__(self, data):
4          self.data = data # Data stored in the node
5          self.next = None # Reference to the next node
6
7
8  class singlyLinkedList:
9      """Singly Linked List implementation with insert and display methods"""
10
11     def __init__(self):
12         self.head = None # Head pointer pointing to the first node
13
14     def insert(self, data):
15         """
16             Insert a new node at the end of the linked list
17
18             Args:
19                 data: The data to be inserted into the linked list
20
21         """
22         new_node = Node(data)
23
24         # If the list is empty, make the new node the head
25         if self.head is None:
26             self.head = new_node
27         else:
28             # Traverse to the end of the list
29             current = self.head
30             while current.next is not None:
31                 current = current.next
32             # Insert the new node at the end
33             current.next = new_node
34
35     def insert_at_beginning(self, data):
36         """
37             Insert a new node at the beginning of the linked list
38
39             Args:
40                 data: The data to be inserted into the linked list
41
42         """
43         new_node = Node(data)
44         new_node.next = self.head
45         self.head = new_node
46
47     def display(self):
48         """
49             Display all elements in the linked list
50
51             If self.head is None:
52                 print("Linked List is empty")
53                 return
54
55             current = self.head
56             elements = []
57             while current is not None:
58                 elements.append(str(current.data))
59                 current = current.next
60
61             # Display in format: data1 -> data2 -> data3 -> None
62             print(" -> ".join(elements) + " -> None")
63
64     # Example usage
65     if __name__ == "__main__":
66         # Create a new linked list
67         ll = singlyLinkedList()
68
69         # Insert some elements
70         print("Inserting elements into the linked list...")
71         ll.insert(10)
72         ll.insert(20)
73         ll.insert(30)
74         ll.insert(40)
75
76         # Display the linked list
77         print("\nlinked list contents:")
78         ll.display()
79
80         # Insert at beginning
81         print("\nInserting 5 at the beginning...")
82         ll.insert_at_beginning(5)
83         ll.display()
84
85         # Create an empty list
86         print("\nCreating an empty linked list:")
87         empty_ll = singlyLinkedList()
88         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[object]()
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[object](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[object](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[object]()
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
95         return order
96
97     def __str__(self) -> str:
98         lines = []
99         for v in self.adj:
100             lines.append(f'{v} -> {self.neighbors(v)}')
101
102         return '\n'.join(lines)
103
104 if __name__ == "__main__":
105     g = Graph(directed=False) # change to True for a directed graph
106     g.add_edge("A", "B")
107     g.add_edge("A", "C")
108     g.add_edge("B", "D")
109     g.add_edge("C", "D")
110     g.add_edge("D", "E")
111
112     print("Adjacency list:")
113     print(g)
114     print()
115     print("BFS from A:", g.bfs("A"))
116     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     def __init__(self):
10         self._heap = []
11         self._seq = count(int) # tie-breaker for equal priorities (FIFO)
12
13     def push(self, item, priority: int):
14         heapq.heappush(self._heap, (priority, next(self._seq), item))
15
16     def pop(self):
17         if not self._heap:
18             raise IndexError("pop from empty PriorityQueue")
19         priority, _, item = heapq.heappop(self._heap)
20         return item, priority
21
22     def peek(self):
23         if not self._heap:
24             raise IndexError("peek from empty PriorityQueue")
25         priority, _, item = self._heap[0]
26         return item, priority
27
28     def __len__(self):
29         return len(self._heap)
30
31     def empty(self):
32         return len(self._heap) == 0
33
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:

1. Student Attendance Tracking
2. Event Registration System
3. Library Book Borrowing
4. Bus Scheduling System
5. 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 collections import defaultdict
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[Set[str], Dict[str, bool]] = {}
20
21     def mark(self, student_id: str, date: str, present: bool) -> None:
22         self._records[date][student_id] = present
23
24     def is_present((self, student_id: str, date: str)) -> Optional[bool]:
25         return self._records.get(date).get(student_id, None)
26
27     def get_presents((self, student_id: str, days: int)) -> int:
28         days = self._records.get(date).values()
29         if not days:
30             return 0
31         present_count = sum(1 for v in days if v)
32         return (present_count / len(days)) * 100.0
33
34
35 # =====#
36 # 2) Event Registration System (Queue)
37 # =====#
38
39 class EventRegistrationSystem:
40     """
41         Data structure: Queue (collections.deque)
42         FIFO registration requests + FIFO waitlist.
43     """
44
45     @dataclass(frozen=True)
46     class Event:
47         event_id: str
48         capacity: int
49
50     def __init__(self) -> None:
51         self._events: Dict[str, EventRegistrationSystem.Event] = {}
52         self._requests: Deque[EventRegistrationSystem.Event] = {} # event_id -> student_id
53         self._waitlist: Deque[EventRegistrationSystem.Event] = {} # event_id -> student_id
54
55     def create_event((self, event_id: str, name: str, capacity: int)) -> None:
56         if capacity < 0:
57             raise ValueError("capacity must be >= 0")
58         self._events[event_id] = Event(event_id, name, capacity)
59
60     def add_request((self, event_id: str, student_id: str)) -> None:
61         self._requests[event_id].append(student_id)
62
63     def process_new_request((self, event_id: str)) -> Optional[str]:
64         """
65             Processes ONE pending request in FIFO order.
66             Returns the student_id that got confirmed (or None if no request).
67         """
68         self._ensure_event(event_id)
69         q = self._requests[event_id]
70         if not q:
71             return None
72
73         student_id = q.pop()
74         if student_id in self._confirmed[event_id]:
75             return None
76
77         if student_id in self._registered[event_id] or student_id in self._waitlist[event_id]:
78             self._requests[event_id].append(student_id)
79
80     def process_new_request((self, event_id: str)) -> Optional[str]:
81         """
82             Processes ONE pending request in FIFO order.
83             Returns the student_id that got confirmed (or None if no request).
84         """
85         self._ensure_event(event_id)
86         q = self._requests[event_id]
87         if not q:
88             return None
89
90         student_id = q.pop()
91         if student_id in self._confirmed[event_id]:
92             return None
93
94         if len(self._confirmed[event_id]) < self._events[event_id].capacity:
95             self._confirmed[event_id].append(student_id)
96             return student_id
97
98         self._waitlist[event_id].append(student_id)
99         return None
100
101     def cancel_registration((self, event_id: str, student_id: str)) -> None:
102         if student_id in self._confirmed[event_id]:
103             self._confirmed[event_id].remove(student_id)
104
105         if student_id in self._waitlist[event_id]:
106             self._waitlist[event_id].remove(student_id)
107
108         return None
109
110     def remove_from_queue((self, event_id: str, student_id: str)) -> None:
111         self._remove_from_queue(event_id, self._waitlist[event_id], student_id)
112
113     def confirmed_list((self, event_id: str)) -> List[str]:
114         self._ensure_event(event_id)
115         return self._confirmed[event_id]
116
117     def waitlist_list((self, event_id: str)) -> List[str]:
118         self._ensure_event(event_id)
119         return list(self._waitlist[event_id])
120
121     def promote_from_waitlist((self, event_id: str)) -> None:
122         if len(self._confirmed[event_id]) >= self._events[event_id].capacity:
123             return
124
125         q = self._waitlist[event_id]
126         if not q:
127             return
128
129         self._confirmed[event_id].add(q.pop())
130
131     def remove_from_queue((self, event_id: str, student_id: str)) -> None:
132         if not student_id in self._confirmed[event_id]:
133             raise KeyError(f"Unknown student_id: {student_id}")
134
135
136 # =====#
137 # 3) Library Book Borrowing (BST)
138 # =====#
139
140 class BookNode:
141     """
142         Data structure: BST (by ISBN) for catalog/inventory search and ordered traversal.
143         Borrowing increments available_copies; returning decrements.
144     """
145
146     def __init__(self, key_isbn: str, book: Book) -> None:
147         self._left: BookNode = None
148         self._book: Book = book
149         self._right: Optional[BookNode] = None
150         self._available_copies: int = available_copies
151
152
153     class BookBSTNode:
154         def __init__(self, key_isbn: str, book: Book) -> None:
155             self._left: BookBSTNode = None
156             self._book: Book = book
157             self._right: Optional[BookBSTNode] = None
158             self._available_copies: int = available_copies
159
160
161     class LibrarySystem:
162         """
163             Data structure: BST (by ISBN) for catalog/inventory search and ordered traversal.
164             Borrowing decrements available_copies; returning increments.
165         """
166
167         def __init__(self) -> None:
168             self._root: Optional[BookBSTNode] = None
169             self._loan: Dict[Tuple[str, str], int] = {} # (student_id, isbn) -> count borrowed
170
171         def add_book(self, isbn: str, title: str, copies: int = 1) -> None:
172             if copies < 0:
173                 raise ValueError("copies must be > 0")
174
175             existing = self._find(isbn)
176             if existing:
177                 existing.total_copies += copies
178                 existing.available_copies -= copies
179             else:
180                 book = Book(isbn=isbn, title=title, total_copies=copies, available_copies=copies)
181                 self._root = self._insert(isbn, title, book, self._root)
182
183         def find(self, isbn: str) -> Optional[Book]:
184             if self._root is None:
185                 return None
186
187             while self._root:
188                 if self._root.isbn == isbn:
189                     return self._root.book
190
191                 if self._root.isbn < isbn:
192                     self._root = self._root.right
193
194                 if self._root.isbn > isbn:
195                     self._root = self._root.left
196
197         def borrow((self, student_id: str, isbn: str)) -> bool:
198             book = self._find(isbn)
199             if not book:
200                 return False
201
202             book.available_copies -= 1
203             self._loan[(student_id, isbn)] = self._loan.get((student_id, isbn), 0) + 1
204             return True
205
206         def return_book((self, student_id: str, isbn: str)) -> bool:
207             key = (student_id, isbn)
208             if not key in self._loan or self._loan[key] < 0:
209                 return False
210
211             self._loan[key] = 0
212
213
214

```

```

148     book = self._find(tsh)
149     if not book:
150         return False
151     self._log.info("book added: %s", str(book))
152     book.available_copies += 1
153     return True
154 
155     def catalog_in_order(self) -> List[Book]:
156         books = []
157         self._catalog_in_order(self.root, books)
158         return books
159 
160     def insert(self, node: Optional[_BookBSTNode], tsh: str, book: Book) -> _BookBSTNode:
161         if not node:
162             return node
163         if tsh < node.key:
164             node.left = self._insert(node.left, tsh, book)
165         else:
166             node.right = self._insert(node.right, tsh, book)
167         return node
168 
169     def _catalog_in_order(self, node: Optional[_BookBSTNode], out: List[Book]) -> None:
170         if not node:
171             return
172         self._catalog_in_order(node.left, out)
173         if node.key == tsh:
174             out.append(book)
175         self._catalog_in_order(node.right, out)
176 
177     # -----
178 
179     # 4) Bus Scheduling System (Script)
180     # -----
181 
182     class BusNetwork:
183         """
184             Data structure: Graph (adjacency list)
185             - stop -> list of (neighbor_stop, travel_minutes).
186             - shortest path uses Dijkstra (non-negative weights).
187         """
188 
189         def __init__(self):
190             self._adj = Dict[Set[int], List[Tuple[int, int]]] = {}
191 
192         def add_stop(self, stop: str) -> None:
193             self._adj[stop].setdefault(stop, [])
194 
195         def add_route(self, start: str, end: str, minutes: int, bidirectional: bool = True) -> None:
196             if minutes < 0:
197                 raise ValueError("Minutes must be non-negative")
198             self.add_stop(start)
199             self.add_stop(end)
200             self._adj[start].append((end, minutes))
201             if bidirectional:
202                 self._adj[end].append((start, minutes))
203 
204         def shortest_path(self, start: str, end: str) -> Tuple[Dict[str, int], List[str]]:
205             if start not in self._adj or end not in self._adj:
206                 raise KeyError("Start/end stop not found")
207 
208             dist: Dict[str, int] = dict()
209             prev: Dict[str, Optional[str]] = {start: None}
210             enq: List[Stop] = [(0, start)]
211 
212             while enq:
213                 d, u = heapq.heappop(enq)
214                 if d > dist.get(u, float("inf")):
215                     continue
216                 if u == end:
217                     break
218                 for v, w in self._adj[u].items():
219                     if v != u:
220                         if d + w < dist.get(v, float("inf")):
221                             dist[v] = d + w
222                             prev[v] = u
223                             heapq.heappush(enq, (d + w, v))
224 
225             if end not in dist:
226                 return (None, [])
227 
228             # Reconstruct path
229             path: List[str] = []
230             cur: Stop = end
231             while cur is not None:
232                 path.append(cur)
233                 cur = prev.get(cur)
234             path.reverse()
235             return dist[end], path
236 
237     # -----
238 
239     # 5) Cafeteria Order Queue (Priority Queue)
240     # -----
241 
242     class CafeteriaOrderSystem:
243         """
244             Data class (Frozen=True)
245             class CafeteriaOrder:
246                 order_id: int
247                 student_id: str
248                 item: str
249                 priority: int # higher number => higher priority
250 
251         class CafeteriaOrderSystem:
252             """
253                 Data structure: Priority Queue (heapq)
254                 - Serve highest priority first; tie-break by arrival order.
255             """
256 
257             def __init__(self) -> None:
258                 self._order_heap: List[Optional[CafeteriaOrder]] = []
259                 self._counter = itertools.count(1)
260 
261             def place_order(self, student_id: str, item: str, priority: int = 0) -> CafeteriaOrder:
262                 order_id = next(self._counter)
263                 order = CafeteriaOrder(order_id=order_id, student_id=student_id, item=item, priority=priority)
264                 heapq.heappush(self._order_heap, (priority, order_id, order))
265                 return order
266 
267             def serve_next(self) -> Optional[CafeteriaOrder]:
268                 if not self._order_heap:
269                     return None
270                 _, _, order = heapq.heappop(self._order_heap)
271                 return order
272 
273             def pending_count(self) -> int:
274                 return len(self._order_heap)
275 
276             # -----
277             # Demo (optional)
278             # -----
279 
280             def main() -> None:
281                 att = AttendanceTracker()
282                 events = EventRegistrationSystem()
283                 lib = LibrarySystem()
284 
285                 events.register("workshop", "workshop", capacity=2)
286                 for sid in ["S1", "S2", "S3"]:
287                     events.request_registration(sid)
288                 events.confirm_registration("S1")
289                 print("Confirmed S1", events.confirmed_list("EIM"))
290                 print("Waiting S1", events.waiting_list("EIM"))
291                 events.cancel_registration("S1")
292                 print("After cancel S2, confirmed", events.confirmed_list("EIM"))
293 
294                 # Library (BS)
295                 lib = LibrarySystem()
296                 lib.add_book("9781491931317", "Effective Java", copies=2)
297                 lib.add_book("9781491931367", "Fluent Python", copies=1)
298                 print("Borrow Fluent Python", lib.borrow("S1", "9781491931367"))
299                 print("Return Fluent Python", lib.return("S1", "9781491931367"))
300                 print("Catalog in order", lib.catalog_in_order())
301 
302                 # Bus Network (Graph)
303                 buses = BusNetwork()
304                 buses.add_stop("Hostel", "Hostel", 0)
305                 buses.add_stop("Library", "Library", 0)
306                 buses.add_stop("Cafeteria", "Cafeteria", 0)
307                 buses.add_route("Hostel", "Cafeteria", 5)
308                 buses.add_route("Cafeteria", "Hostel", 5)
309                 bus = buses.shortest_path("Hostel", "Library")
310                 print("Shortest bus path Hostel->library", bus, "minutes", 5)
311 
312                 # Cafeteria (Priority Queue)
313                 cafe = CafeteriaOrderSystem()
314                 cafe.place_order("S1", "Burger", priority=0) # Higher priority
315                 cafe.place_order("S2", "Coffees", priority=1)
316                 cafe.place_order("S3", "Burger", priority=0)
317                 cafe.serve_next()
318                 print("Serve order", cafe.serve_next())
319 
320             if __name__ == "__main__":
321                 main()
322 
```

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  # -----
7  # Product model
8  # -----
9  class Product:
10     def __init__(self, product_id: int, name: str, price: float):
11         self.id = product_id
12         self.name = name
13         self.price = price
14
15     def __repr__(self):
16         return f"Product(id={self.id}, name='{self.name}', price={self.price})"
17
18
19  # -----
20  # Product Search Engine (Hash Table)
21  # -----
22  class ProductSearchEngine:
23      def __init__(self):
24          self.products: Dict[int, Product] = {}
25
26      def add_product(self, product: Product):
27          self.products[product.id] = product
28
29      def get_product(self, product_id: int) -> Optional[Product]:
30          return self.products.get(product_id)
31
32      def remove_product(self, product_id: int):
33          self.products.pop(product_id, None)
34
35
36  # -----
37  # Shopping Cart (linked list)
38  # -----
39  class CartNode:
40      def __init__(self, product: Product, quantity: int):
41          self.product = product
42          self.quantity = quantity
43          self.next: Optional["CartNode"] = None
44
45
46  class ShoppingCart:
47      def __init__(self):
48          self.head: Optional[CartNode] = None
49
50      def add_product(self, product: Product, quantity: int = 1):
51          ...
52
53          If product already exists in the list, increase quantity.
54          Otherwise, add new node at the front (O(1)) insertion.
55          ...
56
57          node = self.head
58
59          while node:
60              if node.product.id == product.id:
61                  node.quantity += quantity
62                  return
63              node = node.next
64
65
66          new_node = CartNode(product, quantity)
67          new_node.next = self.head
68          self.head = new_node
69
70
71      def remove_product(self, product_id: int, quantity: int = None):
72          ...
73
74          Remove some or all quantity of a product.
75          If quantity is None or reaches 0, remove the node.
76
77          prev = None
78          node = self.head
79
80          while node:
81              if node.product.id == product_id:
82                  if quantity is None or node.quantity <= quantity:
83                      # delete the node
84                      if prev:
85                          prev.next = node.next
86                      else:
87                          self.head = node.next
88
89              else:
90                  node.quantity -= quantity
91
92              prev = node
93              node = node.next
94
95
96      def list_items(self) -> List[Tuple[Product, int]]:
97          result = []
98          node = self.head
99          while node:
100              result.append((node.product, node.quantity))
101              node = node.next
102
103          return result
104
105      def total_price(self) -> float:
106          return sum(node.product.price * node.quantity
107                     for node in self._iter_nodes())
108
109      def _iter_nodes(self):
110          node = self.head
111          while node:
112              yield node
113              node = node.next
114
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
131  class OrderProcessingSystem:
132      def __init__(self):
133          # Queue of orders (FIFO)
134          self.queue: deque[Order] = deque(Order())
135
136      def place_order(self, cart: ShoppingCart) -> Order:
137          order = Order(cart.list_items())
138          self.queue.append(order)
139
140          return order
141
142      def process_next_order(self) -> Optional[Order]:
143          if not self.queue:
144              return None
145          return self.queue.popleft()
146
147      def pending_orders(self) -> int:
148          return len(self.queue)
149
150
151  # -----
152  # Top-Selling Products Tracker (Priority Queue / Max-Heap)
153  # -----
154  class TopSellingProductsTracker:
155      def __init__(self):
156          self.sales: Dict[int, int] = {}
157          self.sales_count: Dict[int, int] = {}
158          # priority queue entries: (-sales_count, product_id)
159          self.heap: List[Tuple[int, int]] = []
160
161      def record_sale(self, product_id: int, quantity: int = 1):
162          self.sales[product_id] = self.sales.get(product_id, 0) + quantity
163
164          # Push new priority entry, lazy update (won't verify against self.sales on pop)
165          heapq.heappush(self.heap, (-self.sales[product_id], product_id))
166
167      def top_k(self, k: int) -> List[Tuple[int, int]]:
168          ...
169
170          Returns list of (product_id, sales_count) for top k products.
171          Uses lazy removal from the heap to keep it consistent.
172          ...
173

```

```

165     result = []
166     seen = set()
167
168     while self.heap and len(result) < k:
169         neg_sales, pid = heapq.heappop(self.heap)
170         current_sales = self.sales.get(pid, 0)
171
172         if current_sales == -neg_sales and pid not in seen:
173             result.append((pid, current_sales))
174             seen.add(pid)
175
176     # push back the elements we popped that are still valid
177     for pid in seen:
178         heapq.heappush(self.heap, (-self.sales[pid], pid))
179
180     return result
181
182 # -----
183 # Delivery Route Planning (Graph + Dijkstra)
184 # -----
185 class DeliveryRoutePlanner:
186     def __init__(self):
187         # Graph as adjacency list: node -> list of (neighbor, distance)
188         self.graph: Dict[str, List[Tuple[str, float]]] = {}
189
190     def add_location(self, name: str):
191         if name not in self.graph:
192             self.graph[name] = []
193
194     def add_route(self, from_loc: str, to_loc: str, distance: float, bidirectional: bool = True):
195         self.add_location(from_loc)
196         self.add_location(to_loc)
197         self.graph[from_loc].append((to_loc, distance))
198         if bidirectional:
199             self.graph[to_loc].append((from_loc, distance))
200
201     def shortest_path(self, start: str, end: str) -> Tuple[float, List[str]]:
202         """
203             Dijkstra's algorithm returns (distance, path).
204             Distance in float("inf") if no path exists.
205         """
206         if start not in self.graph or end not in self.graph:
207             return float("inf"), []
208
209         # min-heap: (distance, node, path)
210         heap = [(0.0, start, [start])]
211         visited = set(heap[0])
212
213         while heap:
214             dist, node, path = heapq.heappop(heap)
215             if node in visited:
216                 continue
217             visited.add(node)
218
219             if node == end:
220                 return dist, path
221
222             for neighbor, weight in self.graph[node]:
223                 if neighbor not in visited:
224                     heapq.heappush(heap, (dist + weight, neighbor, path + [neighbor]))
225
226         return float("inf"), []
227
228 # -----
229 # Example usage:
230 # -----
231 # If name == "main":
232 # Product search engine
233 search_engine = ProductSearchEngine()
234 p1 = Product(1, "Laptop", 1000.0)
235 p2 = Product(2, "Phone", 500.0)
236 p3 = Product(3, "Headphones", 100.0)
237 for p in (p1, p2, p3):
238     search_engine.add_product(p)
239
240 # Shopping cart
241 cart = ShoppingCart()
242 cart.add_product(search_engine.get_product(1), 2)
243 cart.add_product(search_engine.get_product(2), 2)
244 cart.add_product(search_engine.get_product(3), 3)
245 cart.remove_product(3, 1) # remove 1 headphones
246
247 print("Cart items:", cart.list_items())
248 print("Total price:", cart.total_price())
249
250 # Order processing
251 ops = OrderProcessingSystem()
252 order1 = ops.place_order(cart)
253 print("Placed order:", order1)
254 print("Pending orders:", ops.pending_orders())
255 processed = ops.process_next_order()
256 print("Processed order?", processed)
257 print("Pending orders:", ops.pending_orders())
258
259 # Top selling products
260 tracker = TopSellingProductsTracker()
261 tracker.record_sale(1, 10) # Laptop sold 10
262 tracker.record_sale(2, 5) # Phone sold 5
263 tracker.record_sale(3, 7) # Headphones sold 7
264 print("Top 2 products (id, sales):", tracker.top_k(2))
265
266 # Delivery route planner
267 planner = DeliveryRoutePlanner()
268 planner.add_route("WarehouseA", "City1", 10.0)
269 planner.add_route("WarehouseA", "City2", 20.0)
270 planner.add_route("City1", "City2", 5.0)
271 planner.add_route("City1", "City3", 7.0)
272
273 dist, path = planner.shortest_path("WarehouseA", "City3")
274 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:\2403A51L03\3-2\AI_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

```