

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	: M.Naniprasad
Enrollment No.	: 2403A51L11
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         self.items.append(item)
13
14     def pop(self):
15         """Remove and return the top item from the stack.
16
17         Returns:
18             The top item from the stack.
19
20         Raises:
21             IndexError: If the stack is empty.
22         """
23         if self.is_empty():
24             raise IndexError("Cannot pop from an empty stack")
25         return self.items.pop()
26
27     def peek(self):
28         """Return the top item from the stack without removing it.
29
30         Returns:
31             The top item from the stack.
32
33     Raises:
34         IndexError: If the stack is empty.
35     """
36     if self.is_empty():
37         raise IndexError("Cannot peek at an empty stack")
38     return self.items[-1]
39
40     def is_empty(self):
41         """Check if the stack is empty.
42
43         Returns:
44             True if the stack is empty, False otherwise.
45         """
46         return len(self.items) == 0
47
48     def size(self):
49         """Return the number of items in the stack.
50
51         Returns:
52             The number of items in the stack.
53         """
54         return len(self.items)
55
56     def __str__(self):
57         """Return a string representation of the stack.
58
59         Returns:
60             A string representation of the stack.
61         """
62         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:

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

```

Output:

```

Enqueueing 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

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

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             ...
52
53         current = self.head
54
55         elements = []
56         while current is not None:
57             elements.append(str(current.data))
58             current = current.next
59
60         # Display in format: data1 -> data2 -> data3 -> None
61         print(" -> ".join(elements) + " -> None")
62
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      def __init__(self, directed: bool = False):
10         """
11             # adjacency list: vertex -> set of neighbor vertices
12             self.adj: dict[object, set[object]] = {}
13
14     def add_vertex(self, v: object) -> None:
15         """
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         """
22             Add an edge u -> v (and v -> u if undirected).
23             self.add_vertex(u)
24             self.add_vertex(v)
25             self.adj[u].add(v)
26             if not self.directed:
27                 self.adj[v].add(u)
28
29     def remove_edge(self, u: object, v: object) -> None:
30         """
31             Remove an edge u -> v (and v -> u if undirected), if present.
32             if u in self.adj:
33                 self.adj[u].discard(v)
34             if not self.directed and v in self.adj:
35                 self.adj[v].discard(u)
36
37     def remove_vertex(self, v: object) -> None:
38         """
39             Remove a vertex and all edges incident to it.
40             if v not in self.adj:
41                 return
42
43             # Remove edges from neighbors to v
44             for n in list[object](self.adj[v]):
45                 self.remove_edge(v, n)
46
47             # In directed graphs, also remove incoming edges to v
48             if self.directed:
49                 for u in self.adj:
50                     self.adj[u].discard(v)
51
52             del self.adj[v]
53
54     def neighbors(self, v: object) -> list[object]:
55         """
56             Return neighbors of v as a sorted list when possible.
57             if v not in self.adj:
58                 return []
59             try:
60                 return sorted(self.adj[v])
61             except TypeError:
62                 return list[object](self.adj[v])
63
64     def bfs(self, start: object) -> list[object]:
65         """
66             Breadth-first traversal order starting from start.
67             if start not in self.adj:
68                 return []
69
70             visited = {start}
71             queue = [start]
72             order: list[object] = []
73
74             while queue:
75                 current = queue.pop(0)
76                 for neighbor in self.neighbors(current):
77                     if neighbor not in visited:
78                         visited.add(neighbor)
79                         queue.append(neighbor)
80                         order.append(neighbor)
81
82             return order
```

```

67
68     while queue:
69         v = queue.pop(0)
70         order.append(v)
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 bfs(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([start])
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     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:

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:

```

from future import annotations
from dataclasses import dataclass
from collections import deque
from typing import List, Dict, Set, Tuple
from abc import ABC, abstractmethod
from typing import Deque, Dict, List, Optional, Set, Tuple

# =====#
# 1) Student Attendance Tracking (Hash Table)
# =====#
class AttendanceTracker:
    """
    Data structure: Hash Table (Python dict)
    student_id -> (date_str -> present_bool)
    ...
    """

    def __init__(self) -> None:
        self._records: Dict[student_id, Dict[date, bool]] = {}

    def mark(self, student_id: str, date: str, present: bool) -> None:
        self._records.setdefault(student_id, {})[date] = present

    def is_present(self, student_id: str, date: str) -> Optional[bool]:
        return self._records.get(student_id, {}).get(date)

    def attendance_percent(self, student_id: str) -> float:
        date_count, present_count = 0, 0
        for date in self._records.get(student_id, {}):
            date_count += 1
            if self.is_present(student_id, date):
                present_count += 1
        return 8.8 if date_count == 0 else present_count / date_count * 100.0

    # =====#
    # 2) Event Registration System (Queue)
    # =====#
    class EventRegistrationSystem:
        """
        Data structure: Queue (collections.deque)
        FIFO registration requests + FIFO waitlist.
        ...
        """

        @dataclass(frozen=True)
        class Event:
            name: str
            capacity: int
            date: str
            name: str
            capacity: int

        def __init__(self) -> None:
            self._confirmed: Dict[event_id, Set[str]] = {}
            self._confirmed_event_id: Set[event_id] = set()
            self._confirmed_student_id: Dict[event_id, Set[student_id]] = {}
            self._waitlist: Dict[event_id, Deque[student_id]] = {}

        def create_event(self, event_id: str, name: str, capacity: int) -> None:
            if capacity < 0:
                raise ValueError("Capacity must be >= 0")
            self._confirmed[event_id].add(event_id)
            self._confirmed.add(event_id)
            self._confirmed_student_id[event_id] = set()
            self._waitlist[event_id] = deque()

        def request_registration(self, event_id: str, student_id: str) -> None:
            if student_id in self._confirmed[event_id]:
                return
            if student_id in self._waitlist[event_id]:
                return
            if student_id in self._confirmed_student_id[event_id]:
                return
            self._requests.append((event_id, student_id))

        def request_registration(self, event_id: str, student_id: str) -> None:
            if student_id in self._confirmed[event_id]:
                return
            if student_id in self._waitlist[event_id]:
                return
            if student_id in self._confirmed_student_id[event_id]:
                return
            self._requests.append((event_id, student_id))

        def process_next_request(self, event_id: str) -> Optional[str]:
            """
            Processes ONE pending request if FIFO order.
            Returns the student_id that got confirmed (or None if no request).
            ...
            """
            if not self._requests:
                return None
            q, q_size = self._requests.pop(0), len(self._requests)
            if q_size == 1:
                student_id = q[0]
                if len(self._confirmed[event_id]) < self._events[event_id].capacity:
                    self._confirmed[event_id].add(student_id)
                    self._confirmed_student_id[event_id].add(student_id)
                    self._waitlist[event_id].append(student_id)
                return student_id
            return None

        def cancel_registration(self, event_id: str, student_id: str) -> None:
            if student_id not in self._confirmed[event_id]:
                return
            if student_id not in self._confirmed_student_id[event_id]:
                return
            self._confirmed[event_id].remove(student_id)
            self._confirmed_student_id[event_id].remove(student_id)
            self._waitlist[event_id].append(student_id)
            return

        def remove_from_waitlist(self, event_id: str) -> None:
            if event_id not in self._waitlist:
                return
            self._waitlist[event_id].clear()

        def confirmed_list(self, event_id: str) -> List[str]:
            return list(self._confirmed[event_id])

        def remove_from_waitlist(self, event_id: str) -> None:
            if event_id not in self._waitlist:
                return
            self._waitlist[event_id].clear()

        def promote_from_waitlist(self, event_id: str) -> None:
            if event_id not in self._waitlist:
                return
            while len(self._waitlist[event_id]) < self._events[event_id].capacity:
                self._confirmed[event_id].add(self._waitlist[event_id].pop())
            self._waitlist[event_id].clear()

        def remove_from_queue(self, q: Deque[tuple], student_id: str) -> None:
            if not q:
                return
            q_size = len(q)
            for i in range(q_size):
                if q[i][1] == student_id:
                    q.clear()
                    q.extend(q[q_size:i])
                    break

        def promote_if_full(self, event_id: str) -> None:
            if event_id not in self._events:
                raise KeyError("Unknown event_id: " + event_id)

        # =====#
        # 3) Library Book Borrowing (BST)
        # =====#
        class Book:
            title: str
            total_copies: int
            available_copies: int

        class BookBorrowMode:
            def __init__(self, key_isbn: str, book: Book) -> None:
                self._book = book
                self._left = Optional[BookBorrowMode] = None
                self._right = Optional[BookBorrowMode] = None

        class LibrarySystem:
            """
            Data structure: BST (by ISBN) for catalog/inventory search and ordered traversal.
            .. borrowing decrements available copies; returning increments.
            ...
            """

            def __init__(self) -> None:
                self._root: Optional[BookBorrowMode] = None
                self._loans: Dict[tuple, str] = {}

            def __len__(self) -> None:
                if self._root is None:
                    return 0
                return self._root._count_borrowed

            def add_book(self, isbn: str, title: str, copies: int = 1) -> None:
                if copies < 0:
                    raise ValueError("Copies must be >= 0")

                existing = self._find(isbn)
                if existing:
                    existing.total_copies += copies
                    existing.available_copies -= copies
                else:
                    new_node = BookBorrowMode(key_isbn=isbn, book=Book(title=title, total_copies=copies, available_copies=copies))
                    self._add(new_node)

            def _add(self, node: BookBorrowMode) -> None:
                if self._root is None:
                    self._root = node
                elif node.key_isbn < self._root.key_isbn:
                    self._root.left = node
                else:
                    self._root.right = node

            def find(self, isbn: str, title: str) -> Optional[Book]:
                node = self._root
                while node:
                    if node.key_isbn == isbn:
                        return node.book
                    if node.key_isbn < isbn:
                        node = node.right
                    else:
                        node = node.left
                return None

            def borrow(self, student_id: str, title: str) -> bool:
                book = self._find(title)
                if book is None:
                    return False
                if book.available_copies < 1:
                    return False
                self._loans[(student_id, title)] = 0 + 1
                return True

            def return_book(self, student_id: str, title: str) -> bool:
                if student_id not in self._loans:
                    return False
                if self._loans[student_id, title] < 1:
                    return False
                self._loans[student_id, title] -= 1
                if self._loans[student_id, title] < 0:
                    self._loans[student_id, title] = 0
                return True

```

```

100     book = self._find(idbk)
101     if book is None:
102         return False
103     book.available_copies -= 1
104     book.available_costes += 1
105     return True
106
107     def catalog_in_order(self) -> List[Book]:
108         out: List[Book] = []
109         self._order(book, out)
110         return out
111
112     def _insert(self, node: Optional[_BookBSTNode], tsbk: str, book: Book) -> _BookBSTNode:
113         if node is None:
114             return _BookBSTNode(sbsk, book)
115         if tsbk < node.key:
116             node.left = self._insert(node.left, tsbk, book)
117         else:
118             node.right = self._insert(node.right, tsbk, book)
119         return node
120
121     def _order(self, node: Optional[_BookBSTNode], out: List[Book]):
122         if node is None:
123             return
124         self._order(node.left, out)
125         out.append(node.book)
126         self._order(node.right, out)
127
128     # -----
129     # (d) Bus Scheduling System (Graph)
130     # -----
131
132     class BusNetwork:
133         """
134             Structure: Graph (adjacency list)
135             - stop -> list of (stop, stop, travel_minutes)
136             - shortest path uses Dijkstra (non-negative weights).
137         """
138
139         def __init__(self):
140             self._dict: Dict[Stop, List[Tuple[Stop, int]]] = {}
141
142         def add_stop(self, stop: Stop) -> None:
143             self._dict[stop] = []
144
145         def add_route(self, a: Stop, b: Stop, minutes: int, bidirectional: bool = True) -> None:
146             if minutes < 0:
147                 raise ValueError("minutes must be non-negative")
148             self.add_stop(a)
149             self.add_stop(b)
150             self._dict[a].append((b, minutes))
151             if bidirectional:
152                 self._dict[b].append((a, minutes))
153
154         def shortest_path(self, start: Stop, end: Stop) -> Tuple[List[Stop], List[Stop]]:
155             if start not in self._dict or end not in self._dict:
156                 raise KeyError(f"start/end stop not found")
157
158             dist: Dict[Stop, int] = {start: 0}
159             prev: Dict[Stop, Optional[Stop]] = {start: None}
160             pq: List[Tuple[int, Stop]] = [(0, start)]
161
162             while pq:
163                 d_u, u = heapq.heappop(pq)
164                 if d_u > dist[u]:
165                     continue
166
167                 for v in self._dict[u]:
168                     rd = d_u + v[1]
169                     if rd < dist[v[0]]:
170                         dist[v[0]] = rd
171                         prev[v[0]] = u
172                         heapq.heappush(pq, (rd, v[0]))
173
174             if end not in dist:
175                 return (None, None)
176
177             # Reconstruct path
178             path: List[Stop] = []
179             cur: Stop = end
180             while cur is not None:
181                 path.append(cur)
182                 cur = prev[cur]
183             path.reverse()
184             return dist[end], path
185
186         # -----
187         # (e) Cafeteria Order Queue (Priority Queue)
188         # -----
189
190         @dataclass(frozen=True)
191         class CafeteriaOrder:
192             order_id: int
193             student_id: str
194             item: str
195             priority: int
196             priority: int # higher number => higher priority
197
198         class CafeteriaOrderSystem:
199
200             Data structure: Priority Queue (heapq)
201             - Serve highest priority first; tie-break by arrival order.
202             ...
203
204             def __init__(self) -> None:
205                 self._heap: List[(CafeteriaOrder, int)] = []
206                 self._counter: Iterator[int] = count(1)
207
208             def place_order(self, student_id: str, item: str, priority: int = 0) -> CafeteriaOrder:
209                 order_id = next(self._counter)
210                 order = CafeteriaOrder(order_id, student_id, item, priority=priority)
211                 heapq.heappush(self._heap, (priority, order_id, order))
212                 return order
213
214             def serve_next(self) -> Optional[CafeteriaOrder]:
215                 if len(self._heap) == 0:
216                     return None
217                 _, _, order = heapq.heappop(self._heap)
218                 return order
219
220             def pending_count(self) -> int:
221                 return len(self._heap)
222
223             # -----
224             # Demo (optional)
225             # -----
226
227             def main() -> None:
228                 # Attendance
229                 att = AttendanceTracker()
230                 att.create_event("E100", "All workshop", capacity=2)
231                 for sid in ("S1", "S2", "S3"):
232                     events.request_registration("E100", sid)
233                 events.confirm_registration("E100")
234                 print("Confirmed E100: ", events.confirmed.list("E100"))
235                 print("Unconfirmed E100: ", events.unconfirmed.list("E100"))
236                 events.cancel_registration("E100", "S2")
237                 print("After cancel S2, confirmed: ", events.confirmed.list("E100"))
238
239                 # Library
240                 lib = LibrarySystem()
241                 lib.add_book("9781492081360", "Effective Java", copies=2)
242                 lib.add_book("9781492081360", "Fluent Python", copies=1)
243                 print("Borrowed flamer Python from library")
244                 print("Return book to library again: lib.borrow("S1", "9781492081360")")
245                 print("Catalog in order: ", (b.isbn, b.available_copies) for b in lib.catalog_in_order())
246
247                 # Bus network (Graph)
248                 busnet = BusNetwork()
249                 busnet.add_stop("G1", "GATE", 0)
250                 busnet.add_stop("G2", "GATE", 0)
251                 busnet.add_stop("L1", "Library", 0)
252                 busnet.add_stop("H1", "Hostel", 0)
253                 busnet.add_stop("L2", "Library", 0)
254                 busnet.add_stop("H2", "Hostel", 0)
255                 minutes, path = busnet.shortest_path("Hostel", "Library")
256                 print(f"Shortest bus path Hostel->Library: {path}, {minutes} minutes")
257
258                 # cafeteria (Priority Queue)
259                 cafe = Cafeteria()
260                 cafe.add_order("S1", "SaladBox", priority=0)
261                 cafe.place_order("S1", "Coffee", priority=2) # Higher priority
262                 cafe.serve_order("S1", "SaladBox")
263                 print("Serve order: ", cafe.serve_next())
264                 print("Serve order: ", cafe.serve_next())
265
266             if __name__ == "__main__":
267                 main()

```

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
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         # Hash Table: product_id -> Product
25         self.products: Dict[int, Product] = {}
26
27     def add_product(self, product: Product):
28         self.products[product.id] = product
29
30     def get_product(self, product_id: int) -> Optional[Product]:
31         return self.products.get(product_id)
32
33     def remove_product(self, product_id: int):
34         self.products.pop(product_id, None)
35
36
37 # -----
38 # Shopping Cart (Linked List)
39 # -----
40 class CartNode:
41     def __init__(self, product: Product, quantity: int):
42         self.product = product
43         self.quantity = quantity
44         self.next: Optional["CartNode"] = None
45
46
47 class ShoppingCart:
48     def __init__(self):
49         self.head: Optional[CartNode] = None
50
51     def add_product(self, product: Product, quantity: int = 1):
52         """
53             If product already exists in the list, increase quantity.
54             Otherwise, add new node at the front (O(1) insertion).
55         """
56         node = self.head
57         while node:
58             if node.product.id == product.id:
59                 node.quantity += quantity
60                 return
61             node = node.next
62
63         new_node = CartNode(product, quantity)
64         new_node.next = self.head
65         self.head = new_node
66
67     def remove_product(self, product_id: int, quantity: int = None):
68         """
69             Remove some or all quantity of a product.
70             If quantity is None or reaches 0, remove the node.
71         """
72         prev = None
73         node = self.head
74
75         while node:
76             if node.product.id == product_id:
77                 if quantity is None or node.quantity <= quantity:
78                     # delete the node
79                     if prev:
80                         prev.next = node.next
81                     else:
82                         self.head = node.next

```

```

83         else:
84             node.quantity -= quantity
85             return
86         prev = node
87         node = node.next
88     node = self.head
89     while node:
90         result.append((node.product, node.quantity))
91         node = node.next
92     return result
93 
94     def total_price(self) -> float:
95         return sum(node.product.price * node.quantity
96                     for node in self._iter_nodes())
97 
98     def _iter_nodes(self):
99         node = self.head
100         while node:
101             yield node
102             node = node.next
103 
104     # -----
105     # Order Processing System (Queue)
106     # -----
107     class Order:
108         _next_id = 1
109 
110         def __init__(self, cart_snapshot: List[Tuple[Product, int]]):
111             self.id = Order._next_id
112             Order._next_id += 1
113             self.items = cart_snapshot # list of (product, quantity)
114 
115         def __repr__(self):
116             return f'Order(id={self.id}, items={[(p.id, q) for p, q in self.items]})'
117 
118     class OrderProcessingSystem:
119         def __init__(self):
120             # Queue of orders (FIFO)
121             self.queue: deque[Order] = deque(Order())
122 
123         def place_order(self, cart: ShoppingCart) -> Order:
124             order = Order(cart.list_items())
125             self.queue.append(order)
126             return order
127 
128         def process_next_order(self) -> Optional[Order]:
129             if not self.queue:
130                 return None
131             return self.queue.popleft()
132 
133         def pending_orders(self) -> int:
134             return len(self.queue)
135 
136     # -----
137     # Top-Selling Products Tracker (Priority Queue / Max-Heap)
138     # -----
139     class TopSellingProductsTracker:
140         def __init__(self):
141             # product_id -> sales_count
142             self.sales: Dict[int, int] = {}
143             # priority queue entries: (-sales_count, product_id)
144             self.heap: List[Tuple[int, int]] = []
145 
146         def record_sale(self, product_id: int, quantity: int = 1):
147             self.sales[product_id] = self.sales.get(product_id, 0) + quantity
148             # push new popularity entry; lazy update (we'll verify against self.sales on pop)
149             heapq.heappush(self.heap, (-self.sales[product_id], product_id))
150 
151         def top_k(self, k: int) -> List[Tuple[int, int]]:
152             """
153                 Returns list of (product_id, sales_count) for top k products.
154                 Uses lazy removal from the heap to keep it consistent.
155             """
156 
157         return result
158 
159     # -----
160     # Delivery Route Planning (Graph + Dijkstra)
161     # -----
162     class DeliveryRoutePlanner:
163         def __init__(self):
164             # Graph as adjacency list: node -> list of (neighbor, distance)
165             self.graph: Dict[str, List[Tuple[str, float]]] = {}
166 
167         def add_location(self, name: str):
168             if name not in self.graph:
169                 self.graph[name] = []
170 
171         def add_route(self, from_loc: str, to_loc: str, distance: float, bidirectional: bool = True):
172             self.add_location(from_loc)
173             self.add_location(to_loc)
174             self.graph[from_loc].append((to_loc, distance))
175             if bidirectional:
176                 self.graph[to_loc].append((from_loc, distance))
177 
178         def shortest_path(self, start: str, end: str) -> tuple[float, List[str]]:
179             """
180                 Dijkstra's algorithm: returns (distance, path).
181                 Distance is float('inf') if no path exists.
182                 ...
183                 If start not in self.graph or end not in self.graph:
184                     return float('inf'), []
185             """
186             # min heap: (distance, node, path)
187             heap = [(0.0, start, [start])]
188             visited = set()
189 
190             while heap:
191                 dist, node, path = heapq.heappop(heap)
192                 if node in visited:
193                     continue
194                 visited.add(node)
195 
196                 if node == end:
197                     return dist, path
198 
199                 for neighbor, weight in self.graph[node]:
200                     if neighbor not in visited:
201                         heapq.heappush(heap, (dist + weight, neighbor, path + [neighbor]))
202 
203             return float('inf'), []
204 
205         # -----
206         # Example usage
207         # -----
208         if __name__ == '__main__':
209             search_engine = ProductSearchEngine()
210             p1 = Product("laptop", 1000.0)
211             p2 = Product("phone", 500.0)
212             p3 = Product("camera", 800.0)
213             for p in (p1, p2, p3):
214                 search_engine.add_product()

```

```

162         result = []
163         seen = set()
164 
165         while self.heap and len(result) < k:
166             neg_sales, pid = heapq.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             heapq.heappush(self.heap, (-self.sales[pid], pid))
176 
177         return result
178 
179     # -----
180     # Delivery Route Planning (Graph + Dijkstra)
181     # -----
182     class DeliveryRoutePlanner:
183         def __init__(self):
184             # Graph as 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             """
200                 Dijkstra's algorithm: returns (distance, path).
201                 Distance is float('inf') if no path exists.
202                 ...
203                 If start not in self.graph or end not in self.graph:
204                     return float('inf'), []
205             """
206             # min heap: (distance, node, path)
207             heap = [(0.0, start, [start])]
208             visited = set()
209 
210             while heap:
211                 dist, node, path = heapq.heappop(heap)
212                 if node in visited:
213                     continue
214                 visited.add(node)
215 
216                 if node == end:
217                     return dist, path
218 
219                 for neighbor, weight in self.graph[node]:
220                     if neighbor not in visited:
221                         heapq.heappush(heap, (dist + weight, neighbor, path + [neighbor]))
222 
223             return float('inf'), []
224 
225         # -----
226         # Example usage
227         # -----
228         if __name__ == '__main__':
229             search_engine = ProductSearchEngine()
230             p1 = Product("laptop", 1000.0)
231             p2 = Product("phone", 500.0)
232             p3 = Product("camera", 800.0)
233             for p in (p1, p2, p3):
234                 search_engine.add_product()

```

```

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:\2403A51L03\3-2\AT_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

```