SR UNIVERSITY

School of Computer Science and Artificial Intelligence

Lab Assignment # 11.2

Program : B. Tech (CSE)
Specialization : -
Course Title : AI Assisted Coding
Course Code : 23CS002PC304
Semester : II
Academic Session : 2025-2026
Name of Student : Mohammed Hashir Sayam Hussain
Enrollment No. : 2403A51L27
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:

```python
class Stack:
    def __init__(self):
        """Initialize an empty stack."""
        self.items = []

    def push(self, item):
        """Add an item to the top of the stack.

        Args:
            item: The item to be added to the stack.
        """
        self.items.append(item)

    def pop(self):
        """Remove and return the top item from the stack.

        Returns:
            The top item from the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot pop from an empty stack")
        return self.items.pop()

    def peek(self):
        """Return the top item from the stack without removing it.

        Returns:
            The top item from the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot peek at an empty stack")
        return self.items[-1]

    def is_empty(self):
        """Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.items) == 0

    def size(self):
        """Return the number of items in the stack.

        Returns:
            The number of items in the stack.
        """
        return len(self.items)

    def __str__(self):
        """Return a string representation of the stack.

        Returns:
            A string representation of the stack.
        """
        return f"Stack({self.items})"
```

```python


# Example usage and testing
if __name__ == "__main__":
    # Create a new stack
    stack = Stack()

    # Test is_empty on empty stack
    print(f"Is stack empty? {stack.is_empty()}")
    print(f"Stack size: {stack.size()}")
    print()

    # Push items onto the stack
    print("Pushing items: 10, 20, 30")
    stack.push(10)
    stack.push(20)
    stack.push(30)
    print(f"Stack: {stack}")
    print(f"Is stack empty? {stack.is_empty()}")
    print(f"Stack size: {stack.size()}")
    print()

    # Peek at the top item
    print(f"Peek at top: {stack.peek()}")
    print(f"Stack after peek: {stack}")
    print()

    # Pop items from the stack
    print("Popping items:")
    print(f"Popped: {stack.pop()}")
    print(f"Stack: {stack}")
    print(f"Popped: {stack.pop()}")
    print(f"Stack: {stack}")
    print(f"Popped: {stack.pop()}")
    print(f"Stack: {stack}")
    print()

    # Test is_empty after popping all items
    print(f"Is stack empty? {stack.is_empty()}")
    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:

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

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

        Args:
            item: The item to add to the queue
        """
        self.items.append(item)

    def dequeue(self):
        """Remove and return the front item from the queue.

        Returns:
            The front item of the queue

        Raises:
            IndexError: If the queue is empty
        """
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue")
        return self.items.pop(0)

    def peek(self):
        """Return the front item of the queue without removing it.

        Returns:
            The front item of the queue

        Raises:
            IndexError: If the queue is empty
        """
        if self.is_empty():
            raise IndexError("Cannot peek at an empty queue")
        return self.items[0]

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

        Returns:
            The number of items in the queue
        """
        return len(self.items)

    def is_empty(self):
        """Check if the queue is empty.

        Returns:
            True if the queue is empty, False otherwise
        """
        return len(self.items) == 0


# Example usage and testing
if __name__ == "__main__":
    # Create a new queue
    q = Queue()
      Ctrl+L to chat, Ctrl+K to generate
    # Test enqueue
    print("Enqueueing items: 1, 2, 3, 4, 5")
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    q.enqueue(4)
    q.enqueue(5)

    # Test size
    print(f"Queue size: {q.size()}")

    # Test peek
    print(f"Peek at front: {q.peek()}")

    # Test dequeue
    print("\nDequeueing items:")
    while not q.is_empty():
        print(f"  Dequeued: {q.dequeue()}, Remaining size: {q.size()}")

    # Test empty queue
    print(f"\nQueue is empty: {q.is_empty()}")

    # Test error handling
    try:
        q.dequeue()
    except IndexError as e:
        print(f"Error caught: {e}")

    try:
        q.peek()
    except IndexError as e:
        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

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:

SR UNIVERSITY

```python
class Node:
    """Node class to represent a single node in the linked list"""
    def __init__(self, data):
        self.data = data  # Data stored in the node
        self.next = None  # Reference to the next node


class SinglyLinkedList:
    """Singly Linked List implementation with insert and display methods"""

    def __init__(self):
        self.head = None  # Head pointer pointing to the first node

    def insert(self, data):
        """
        Insert a new node at the end of the linked list

        Args:
            data: The data to be inserted into the linked list
        """
        new_node = Node(data)

        # If the list is empty, make the new node the head
        if self.head is None:
            self.head = new_node
        else:
            # Traverse to the end of the list
            current = self.head
            while current.next is not None:
                current = current.next
            # Insert the new node at the end
            current.next = new_node

    def insert_at_beginning(self, data):
        """
        Insert a new node at the beginning of the linked list

        Args:
            data: The data to be inserted into the linked list
        """
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        """
        Display all elements in the linked list
        """
        if self.head is None:
            print("Linked List is empty")
            return

        current = self.head
        elements = []
        while current is not None:
            elements.append(str(current.data))
            current = current.next

        # Display in format: data1 -> data2 -> data3 -> None
        print(" -> ".join(elements) + " -> None")


# Example usage
if __name__ == "__main__":
    # Create a new linked list
    ll = SinglyLinkedList()

    # Insert some elements
    print("Inserting elements into the linked list...")
    ll.insert(10)
    ll.insert(20)
    ll.insert(30)
    ll.insert(40)

    # Display the linked list
    print("\nLinked List contents:")
    ll.display()

    # Insert at beginning
    print("\nInserting 5 at the beginning...")
    ll.insert_at_beginning(5)
    ll.display()

    # Create an empty list
    print("\nCreating an empty linked list:")
    empty_ll = SinglyLinkedList()
    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:

```python
class Node:
    def __init__(self, key: int):
        self.key = key
        self.left: "Node | None" = None
        self.right: "Node | None" = None


class BST:
    def __init__(self):
        self.root: Node | None = None

    def insert(self, key: int) -> None:
        """Insert key into the BST (duplicates go to the right)."""
        if self.root is None:
            self.root = Node(key)
            return

        cur = self.root
        while True:
            if key < cur.key:
                if cur.left is None:
                    cur.left = Node(key)
                    return
                cur = cur.left
            else:
                if cur.right is None:
                    cur.right = Node(key)
                    return
                cur = cur.right

    def inorder(self) -> list[int]:
        """Return keys in in-order (sorted) order."""
        result: list[int] = []

        def dfs(n: Node | None) -> None:
            if n is None:
                return
            dfs(n.left)
            result.append(n.key)
            dfs(n.right)

        dfs(self.root)
        return result


if __name__ == "__main__":
    bst = BST()
    for x in [7, 3, 9, 1, 5, 8, 10]:
        bst.insert(x)
    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:

```python
class HashTable:
    pass
```

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

Code:

```python
class HashTable:
    """
    Hash table using separate chaining (list of buckets).

    Methods:
      - insert(key, value): add/update a key
      - search(key): return value or None if not found
      - delete(key): remove key, return True if removed else False
    """

    def __init__(self, capacity: int = 8) -> None:
        if capacity < 1:
            raise ValueError("capacity must be >= 1")
        self._capacity = capacity
        self._buckets = [[] for _ in range(self._capacity)]  # list[list[tuple[key, value]]]
        self._size = 0

    def _index(self, key) -> int:
        return hash(key) % self._capacity

    def _rehash(self, new_capacity: int) -> None:
        old_items = []
        for bucket in self._buckets:
            old_items.extend(bucket)

        self._capacity = new_capacity
        self._buckets = [[] for _ in range(self._capacity)]
        self._size = 0

        for k, v in old_items:
            self.insert(k, v)

    def insert(self, key, value) -> None:
        # Resize when load factor gets too high (simple rule-of-thumb)
        if (self._size + 1) / self._capacity > 0.75:
            self._rehash(self._capacity * 2)

        idx = self._index(key)
        bucket = self._buckets[idx]

        for i, (k, _) in enumerate[Any](bucket):
            if k == key:
                bucket[i] = (key, value)  # update existing
                return

        bucket.append((key, value))
        self._size += 1

    def search(self, key):
        idx = self._index(key)
        bucket = self._buckets[idx]
        for k, v in bucket:
            if k == key:
                return v
        return None

    def delete(self, key) -> bool:
        idx = self._index(key)
        bucket = self._buckets[idx]

        for i, (k, _) in enumerate[Any](bucket):
            if k == key:
                bucket.pop(i)
                self._size -= 1
                return True

        return False

    def __len__(self) -> int:
        return self._size

    def __contains__(self, key) -> bool:
        return self.search(key) is not None

    def __repr__(self) -> str:
        return f"HashTable(size={self._size}, capacity={self._capacity})"


if __name__ == "__main__":
    ht = HashTable()
    ht.insert("name", "Alice")
    ht.insert("age", 20)
    ht.insert("age", 21)  # update

    print(ht)                   # HashTable(...)
    print(ht.search("name"))  # Alice
    print(ht.search("age"))   # 21
    print(ht.search("x"))     # None

    print(ht.delete("age"))   # True
    print(ht.delete("age"))   # False
    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:

```python
class Graph:
    """
    Graph implemented using an adjacency list.

    - By default the graph is undirected.
    - Set directed=True for a directed graph.
    """

    def __init__(self, directed: bool = False):
        self.directed = directed
        # adjacency list: vertex -> set of neighbor vertices
        self.adj: dict[object, set[object]] = {}

    def add_vertex(self, v: object) -> None:
        """Add a vertex if it doesn't already exist."""
        if v not in self.adj:
            self.adj[v] = set[object]()

    def add_edge(self, u: object, v: object) -> None:
        """Add an edge u -> v (and v -> u if undirected)."""
        self.add_vertex(u)
        self.add_vertex(v)
        self.adj[u].add(v)
        if not self.directed:
            self.adj[v].add(u)

    def remove_edge(self, u: object, v: object) -> None:
        """Remove an edge u -> v (and v -> u if undirected), if present."""
        if u in self.adj:
            self.adj[u].discard(v)
        if not self.directed and v in self.adj:
            self.adj[v].discard(u)

    def remove_vertex(self, v: object) -> None:
        """Remove a vertex and all edges incident to it."""
        if v not in self.adj:
            return

        # Remove edges from neighbors to v
        for n in list[object](self.adj[v]):
            self.remove_edge(v, n)

        # In directed graphs, also remove incoming edges to v
        if self.directed:
            for u in self.adj:
                self.adj[u].discard(v)

        del self.adj[v]

    def neighbors(self, v: object) -> list[object]:
        """Return neighbors of v as a sorted list when possible."""
        if v not in self.adj:
            return []
        try:
            return sorted(self.adj[v])
        except TypeError:
            return list[object](self.adj[v])

    def bfs(self, start: object) -> list[object]:
        """Breadth-first traversal order starting from start."""
        if start not in self.adj:
            return []

        visited = {start}
        queue = [start]
        order: list[object] = []

        while queue:
            v = queue.pop(0)
            order.append(v)
            for n in self.neighbors(v):
                if n not in visited:
                    visited.add(n)
                    queue.append(n)

        return order

    def dfs(self, start: object) -> list[object]:
        """Depth-first traversal order starting from start."""
        if start not in self.adj:
            return []

        visited: set[object] = set[object]()
        order: list[object] = []

        def _visit(v: object) -> None:
            visited.add(v)
            order.append(v)
            for n in self.neighbors(v):
                if n not in visited:
                    _visit(n)

        _visit(start)
        return order

    def __str__(self) -> str:
        lines = []
        for v in self.adj:
            lines.append(f"{v} -> {self.neighbors(v)}")
        return "\n".join(lines)


if __name__ == "__main__":
    g = Graph(directed=False)  # change to True for a directed graph
    g.add_edge("A", "B")
    g.add_edge("A", "C")
    g.add_edge("B", "D")
    g.add_edge("C", "D")
    g.add_edge("D", "E")

    print("Adjacency list:")
    print(g)
    print()
    print("BFS from A:", g.bfs("A"))
    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:

```python
import heapq
from itertools import count

class PriorityQueue:
    """
    Min-priority queue by default (smaller priority value = served first).
    For max-priority behavior, push with -priority.
    """
    def __init__(self):
        self._heap = []
        self._seq = count[int]()  # tie-breaker for equal priorities (FIFO)

    def push(self, item, priority: int):
        heapq.heappush(self._heap, (priority, next(self._seq), item))

    def pop(self):
        if not self._heap:
            raise IndexError("pop from empty PriorityQueue")
        priority, _, item = heapq.heappop(self._heap)
        return item, priority

    def peek(self):
        if not self._heap:
            raise IndexError("peek from empty PriorityQueue")
        priority, _, item = self._heap[0]
        return item, priority

    def __len__(self):
        return len(self._heap)

    def empty(self):
        return len(self._heap) == 0


if __name__ == "__main__":
    pq = PriorityQueue()
    pq.push("low", 5)
    pq.push("urgent", 1)
    pq.push("medium", 3)
    pq.push("also urgent (arrives later)", 1)

    while not pq.empty():
        item, pr = pq.pop()
        print(pr, item)

    # Max-priority example (bigger number = served first):
    maxpq = PriorityQueue()
    for item, pr in [("A", 10), ("B", 2), ("C", 10)]:
        maxpq.push(item, -pr)  # negate priority

    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:

```python
class DequeDS:
    pass
```

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

Code:

```python
from __future__ import annotations

from collections import deque
from typing import Deque, Generic, Iterator, Optional, TypeVar

T = TypeVar("T")


class DequeDS(Generic[T]):
    """
    Double-ended queue (deque) implemented using collections.deque.
    Supports O(1) append/pop operations on both ends.
    """

    def __init__(self, items: Optional[Iterator[T]] = None) -> None:
        self._dq: Deque[T] = deque[T](items or [])

    # --- Add operations ---
    def add_front(self, item: T) -> None:
        """Insert item at the front (left)."""
        self._dq.appendleft(item)

    def add_rear(self, item: T) -> None:
        """Insert item at the rear (right)."""
        self._dq.append(item)

    # --- Remove operations ---
    def remove_front(self) -> T:
        """Remove and return the front (left) item."""
        if self.is_empty():
            raise IndexError("remove_front from empty deque")
        return self._dq.popleft()

    def remove_rear(self) -> T:
        """Remove and return the rear (right) item."""
        if self.is_empty():
            raise IndexError("remove_rear from empty deque")
        return self._dq.pop()

    # --- Peek operations ---
    def peek_front(self) -> T:
        """Return the front (left) item without removing it."""
        if self.is_empty():
            raise IndexError("peek_front from empty deque")
        return self._dq[0]

    def peek_rear(self) -> T:
        """Return the rear (right) item without removing it."""
        if self.is_empty():
            raise IndexError("peek_rear from empty deque")
        return self._dq[-1]

    # --- Utility ---
    def is_empty(self) -> bool:
        return len(self._dq) == 0

    def size(self) -> int:
        return len(self._dq)

    def clear(self) -> None:
        self._dq.clear()

    def __len__(self) -> int:
        return len(self._dq)

    def __iter__(self) -> Iterator[T]:
        return iter(self._dq)

    def __repr__(self) -> str:
        return f"DequeDS({list[T](self._dq)!r})"


if __name__ == "__main__":
    d = DequeDS[int]()
    d.add_front(10)    # [10]
    d.add_rear(20)     # [10, 20]
    d.add_front(5)     # [5, 10, 20]
    print("Deque:", d)
    print("Front:", d.peek_front())
    print("Rear:", d.peek_rear())
    print("Remove front:", d.remove_front())  # 5
    print("Remove rear:", d.remove_rear())    # 20
    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:

```python
from __future__ import annotations

from dataclasses import dataclass
from collections import deque
import heapq
import itertools
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[str, Dict[str, 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:
        days = self._records.get(student_id, {})
        if not days:
            return 0.0
        present_count = sum(1 for v in days.values() if v)
        return (present_count / len(days)) * 100.0


# ==========================
# 2) Event Registration System (Queue)
# ==========================


class EventRegistrationSystem:
    """
    Data structure: Queue (collections.deque)
    - FIFO registration requests + FIFO waitlist.
    """

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

    def __init__(self) -> None:
        self._events: Dict[str, EventRegistrationSystem.Event] = {}
        self._confirmed: Dict[str, Set[str]] = {}  # event_id -> set(student_id)
        self._requests: Dict[str, Deque[str]] = {}  # event_id -> queue(student_id)
        self._waitlist: Dict[str, Deque[str]] = {}  # event_id -> queue(student_id)

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

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

    def process_next_request(self, event_id: str) -> Optional[str]:
        """
        Processes ONE pending request in FIFO order.
        Returns the student_id that got confirmed (or None if no request).
        """
        self._ensure_event(event_id)
        q = self._requests[event_id]
        if not q:
            return None

        student_id = q.popleft()
        if len(self._confirmed[event_id]) < self._events[event_id].capacity:
            self._confirmed[event_id].add(student_id)
            return student_id

        self._waitlist[event_id].append(student_id)
        return None

    def cancel_registration(self, event_id: str, student_id: str) -> None:
        self._ensure_event(event_id)
        if student_id in self._confirmed[event_id]:
            self._confirmed[event_id].remove(student_id)
            self._promote_from_waitlist(event_id)
            return
        self._remove_from_queue(self._requests[event_id], student_id)
        self._remove_from_queue(self._waitlist[event_id], student_id)

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

    def waitlist_list(self, event_id: str) -> List[str]:
        self._ensure_event(event_id)
        return list[str](self._waitlist[event_id])

    def _promote_from_waitlist(self, event_id: str) -> None:
        if len(self._confirmed[event_id]) >= self._events[event_id].capacity:
            return
        wl = self._waitlist[event_id]
        while wl and len(self._confirmed[event_id]) < self._events[event_id].capacity:
            self._confirmed[event_id].add(wl.popleft())

    def _remove_from_queue(self, q: Deque[str], student_id: str) -> None:
        # deque doesn't support fast middle removal; rebuild for clarity.
        if not q:
            return
        new_q = deque[str](x for x in q if x != student_id)
        q.clear()
        q.extend(new_q)

    def _ensure_event(self, event_id: str) -> None:
        if event_id not in self._events:
            raise KeyError(f"Unknown event_id: {event_id}")


# ==========================
# 3) Library Book Borrowing (BST)
# ==========================


@dataclass
class Book:
    isbn: str
    title: str
    total_copies: int
    available_copies: int


class _BookBSTNode:
    def __init__(self, key_isbn: str, book: Book) -> None:
        self.key = key_isbn
        self.book = book
        self.left: Optional[_BookBSTNode] = None
        self.right: Optional[_BookBSTNode] = 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[_BookBSTNode] = None
        self._loans: Dict[Tuple[str, str], int] = {}  # (student_id, isbn) -> 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
            return

        book = Book(isbn=isbn, title=title, total_copies=copies, available_copies=copies)
        self._root = self._insert(self._root, isbn, book)

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

    def borrow(self, student_id: str, isbn: str) -> bool:
        book = self.find(isbn)
        if not book or book.available_copies <= 0:
            return False
        book.available_copies -= 1
        self._loans[(student_id, isbn)] = self._loans.get((student_id, isbn), 0) + 1
        return True

    def return_book(self, student_id: str, isbn: str) -> bool:
        key = (student_id, isbn)
        if self._loans.get(key, 0) <= 0:
            return False
```

```python
        book = self.find(isbn)
        if not book:
            return False
        self._loans[key] -= 1
        book.available_copies += 1
        return True

    def catalog_in_order(self) -> List[Book]:
        out: List[Book] = []
        self._in_order(self._root, out)
        return out

    def _insert(self, node: Optional[_BookBSTNode], isbn: str, book: Book) -> _BookBSTNode:
        if node is None:
            return _BookBSTNode(isbn, book)
        if isbn < node.key:
            node.left = self._insert(node.left, isbn, book)
        else:
            node.right = self._insert(node.right, isbn, book)
        return node

    def _in_order(self, node: Optional[_BookBSTNode], out: List[Book]) -> None:
        if node is None:
            return
        self._in_order(node.left, out)
        out.append(node.book)
        self._in_order(node.right, out)


# ==========================
# 4) Bus Scheduling System (Graph)
# ==========================


class BusNetwork:
    """
    Data structure: Graph (adjacency list)
    - stop -> list of (neighbor_stop, travel_minutes)
    - shortest path uses Dijkstra (non-negative weights).
    """

    def __init__(self) -> None:
        self._adj: Dict[str, List[Tuple[str, int]]] = {}

    def add_stop(self, stop: str) -> None:
        self._adj.setdefault(stop, [])

    def add_route(self, a: str, b: str, minutes: int, bidirectional: bool = True) -> None:
        if minutes < 0:
            raise ValueError("minutes must be non-negative")
        self.add_stop(a)
        self.add_stop(b)
        self._adj[a].append((b, minutes))
        if bidirectional:
            self._adj[b].append((a, minutes))

    def shortest_path(self, start: str, end: str) -> Tuple[int, List[str]]:
        if start not in self._adj or end not in self._adj:
            raise KeyError("start/end stop not found")

        dist: Dict[str, int] = {start: 0}
        prev: Dict[str, Optional[str]] = {start: None}
        pq: List[Tuple[int, str]] = [(0, start)]

        while pq:
            d, u = heapq.heappop(pq)
            if d != dist.get(u, 10**18):
                continue
            if u == end:
                break
            for v, w in self._adj[u]:
                nd = d + w
                if nd < dist.get(v, 10**18):
                    dist[v] = nd
                    prev[v] = u
                    heapq.heappush(pq, (nd, v))

        if end not in dist:
            return (10**18, [])

        # Reconstruct path
        path: List[str] = []
        cur: Optional[str] = end
        while cur is not None:
            path.append(cur)
            cur = prev.get(cur)
        path.reverse()
        return dist[end], path


# ==========================
# 5) Cafeteria Order Queue (Priority Queue)
# ==========================



@dataclass(frozen=True)
class CafeteriaOrder:
    order_id: int
    student_id: str
    item: str
    priority: int  # higher number => higher priority


class CafeteriaOrderSystem:
    """
    Data structure: Priority Queue (heapq)
    - Serve highest priority first; tie-break by arrival order.
    """

    def __init__(self) -> None:
        self._heap: List[Tuple[int, int, CafeteriaOrder]] = []
        self._counter = itertools.count(1)

    def place_order(self, student_id: str, item: str, priority: int = 0) -> CafeteriaOrder:
        order_id = next(self._counter)
        order = CafeteriaOrder(order_id=order_id, student_id=student_id, item=item, priority=priority)
        # heapq is min-heap; invert priority so bigger priority pops first
        heapq.heappush(self._heap, (-priority, order_id, order))
        return order

    def serve_next(self) -> Optional[CafeteriaOrder]:
        if not self._heap:
            return None
        _, _, order = heapq.heappop(self._heap)
        return order

    def pending_count(self) -> int:
        return len(self._heap)


# ==========================
# Demo (optional)
# ==========================


def main() -> None:
    # Attendance
    att = AttendanceTracker()
    att.mark("S1", "2026-02-17", True)
    att.mark("S1", "2026-02-18", False)
    print("Attendance S1 %:", round(att.attendance_percent("S1"), 2))

    # Events
    events = EventRegistrationSystem()
    events.create_event("E100", "AI Workshop", capacity=2)
    for sid in ["S1", "S2", "S3"]:
        events.request_registration("E100", sid)
        events.process_next_request("E100")
    print("Confirmed E100:", events.confirmed_list("E100"))
    print("Waitlist E100:", events.waitlist_list("E100"))
    events.cancel_registration("E100", "S2")
    print("After cancel S2, confirmed:", events.confirmed_list("E100"))

    # Library (BST)
    lib = LibrarySystem()
    lib.add_book("9780134685991", "Effective Java", copies=2)
    lib.add_book("9781492051367", "Fluent Python", copies=1)
    print("Borrow Fluent Python:", lib.borrow("S1", "9781492051367"))
    print("Borrow Fluent Python again:", lib.borrow("S2", "9781492051367"))
    print("Catalog in order:", [(b.isbn, b.available_copies) for b in lib.catalog_in_order()])

    # Bus network (Graph)
    buses = BusNetwork()
    buses.add_route("Hostel", "Gate", 8)
    buses.add_route("Gate", "Library", 6)
    buses.add_route("Hostel", "Cafeteria", 5)
    buses.add_route("Cafeteria", "Library", 4)
    minutes, path = buses.shortest_path("Hostel", "Library")
    print("Shortest bus path Hostel->Library:", path, "minutes:", minutes)

    # Cafeteria (Priority Queue)
    cafe = CafeteriaOrderSystem()
    cafe.place_order("S1", "Sandwich", priority=0)
    cafe.place_order("S2", "Coffee", priority=2)  # higher priority
    cafe.place_order("S3", "Burger", priority=1)
    print("Serve order:", cafe.serve_next())
    print("Serve order:", cafe.serve_next())


if __name__ == "__main__":
    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
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:

```python
from collections import deque
import heapq
from typing import Dict, List, Tuple, Optional


# ---------------------------
# Product model
# ---------------------------
class Product:
    def __init__(self, product_id: int, name: str, price: float):
        self.id = product_id
        self.name = name
        self.price = price

    def __repr__(self):
        return f"Product(id={self.id}, name='{self.name}', price={self.price})"


# ---------------------------
# Product Search Engine (Hash Table)
# ---------------------------
class ProductSearchEngine:
    def __init__(self):
        # Hash Table: product_id -> Product
        self.products: Dict[int, Product] = {}

    def add_product(self, product: Product):
        self.products[product.id] = product

    def get_product(self, product_id: int) -> Optional[Product]:
        return self.products.get(product_id)

    def remove_product(self, product_id: int):
        self.products.pop(product_id, None)


# ---------------------------
# Shopping Cart (Linked List)
# ---------------------------
class CartNode:
    def __init__(self, product: Product, quantity: int):
        self.product = product
        self.quantity = quantity
        self.next: Optional["CartNode"] = None


class ShoppingCart:
    def __init__(self):
        self.head: Optional[CartNode] = None

    def add_product(self, product: Product, quantity: int = 1):
        """
        If product already exists in the list, increase quantity.
        Otherwise, add new node at the front (O(1) insertion).
        """
        node = self.head
        while node:
            if node.product.id == product.id:
                node.quantity += quantity
                return
            node = node.next

        new_node = CartNode(product, quantity)
        new_node.next = self.head
        self.head = new_node

    def remove_product(self, product_id: int, quantity: int = None):
        """
        Remove some or all quantity of a product.
        If quantity is None or reaches 0, remove the node.
        """
        prev = None
        node = self.head

        while node:
            if node.product.id == product_id:
                if quantity is None or node.quantity <= quantity:
                    # delete the node
                    if prev:
                        prev.next = node.next
                    else:
                        self.head = node.next
                else:
                    node.quantity -= quantity
                return
            prev = node
            node = node.next

    def list_items(self) -> List[Tuple[Product, int]]:
        result = []
        node = self.head
        while node:
            result.append((node.product, node.quantity))
            node = node.next
        return result

    def total_price(self) -> float:
        return sum(node.product.price * node.quantity
                   for node in self._iter_nodes())

    def _iter_nodes(self):
        node = self.head
        while node:
            yield node
            node = node.next


# ---------------------------
# Order Processing System (Queue)
# ---------------------------
class Order:
    _next_id = 1

    def __init__(self, cart_snapshot: List[Tuple[Product, int]]):
        self.id = Order._next_id
        Order._next_id += 1
        self.items = cart_snapshot  # List of (Product, quantity)

    def __repr__(self):
        return f"Order(id={self.id}, items={[(p.id, q) for p, q in self.items]})"


class OrderProcessingSystem:
    def __init__(self):
        # Queue of orders (FIFO)
        self.queue: deque[Order] = deque[Order]()

    def place_order(self, cart: ShoppingCart) -> Order:
        order = Order(cart.list_items())
        self.queue.append(order)
        return order

    def process_next_order(self) -> Optional[Order]:
        if not self.queue:
            return None
        return self.queue.popleft()

    def pending_orders(self) -> int:
        return len(self.queue)


# ---------------------------
# Top-Selling Products Tracker (Priority Queue / Max-Heap)
# ---------------------------
class TopSellingProductsTracker:
    def __init__(self):
        # product_id -> sales_count
        self.sales: Dict[int, int] = {}
        # priority queue entries: (-sales_count, product_id)
        self.heap: List[Tuple[int, int]] = []

    def record_sale(self, product_id: int, quantity: int = 1):
        self.sales[product_id] = self.sales.get(product_id, 0) + quantity
        # Push new priority entry; lazy update (we'll verify against self.sales on pop)
        heapq.heappush(self.heap, (-self.sales[product_id], product_id))

    def top_k(self, k: int) -> List[Tuple[int, int]]:
        """
        Returns list of (product_id, sales_count) for top k products.
        Uses lazy removal from the heap to keep it consistent.
        """
```

```python
        result = []
        seen = set[Any]()

        while self.heap and len(result) < k:
            neg_sales, pid = heapq.heappop(self.heap)
            current_sales = self.sales.get(pid, 0)

            if current_sales == -neg_sales and pid not in seen:
                result.append((pid, current_sales))
                seen.add(pid)

        # push back the elements we popped that are still valid
        for pid in seen:
            heapq.heappush(self.heap, (-self.sales[pid], pid))

        return result


# ---------------------------
# Delivery Route Planning (Graph + Dijkstra)
# ---------------------------
class DeliveryRoutePlanner:
    def __init__(self):
        # Graph as adjacency list: node -> list of (neighbor, distance)
        self.graph: Dict[str, List[Tuple[str, float]]] = {}

    def add_location(self, name: str):
        if name not in self.graph:
            self.graph[name] = []

    def add_route(self, from_loc: str, to_loc: str, distance: float, bidirectional: bool = True):
        self.add_location(from_loc)
        self.add_location(to_loc)
        self.graph[from_loc].append((to_loc, distance))
        if bidirectional:
            self.graph[to_loc].append((from_loc, distance))

    def shortest_path(self, start: str, end: str) -> Tuple[float, List[str]]:
        """
        Dijkstra's algorithm: returns (distance, path).
        Distance is float('inf') if no path exists.
        """
        if start not in self.graph or end not in self.graph:
            return float('inf'), []

        # min-heap: (distance, node, path)
        heap = [(0.0, start, [start])]
        visited = set[Any]()

        while heap:
            dist, node, path = heapq.heappop(heap)
            if node in visited:
                continue
            visited.add(node)

            if node == end:
                return dist, path

            for neighbor, weight in self.graph[node]:
                if neighbor not in visited:
                    heapq.heappush(heap, (dist + weight, neighbor, path + [neighbor]))

        return float('inf'), []


# ---------------------------
# Example usage
# ---------------------------
if __name__ == "__main__":
    # Product search engine
    search_engine = ProductSearchEngine()
    p1 = Product(1, "Laptop", 1000.0)
    p2 = Product(2, "Phone", 500.0)
    p3 = Product(3, "Headphones", 100.0)
    for p in (p1, p2, p3):
        search_engine.add_product(p)

    # Shopping cart
    cart = ShoppingCart()
    cart.add_product(search_engine.get_product(1), 1)
    cart.add_product(search_engine.get_product(2), 2)
    cart.add_product(search_engine.get_product(3), 3)
    cart.remove_product(3, 1)  # remove 1 headphone

    print("Cart items:", cart.list_items())
    print("Total price:", cart.total_price())

    # Order processing
    ops = OrderProcessingSystem()
    order1 = ops.place_order(cart)
    print("Placed order:", order1)
    print("Pending orders:", ops.pending_orders())
    processed = ops.process_next_order()
    print("Processed order:", processed)
    print("Pending orders:", ops.pending_orders())

    # Top-selling products
    tracker = TopSellingProductsTracker()
    tracker.record_sale(1, 10)  # Laptop sold 10
    tracker.record_sale(2, 5)   # Phone sold 5
    tracker.record_sale(3, 7)   # Headphones sold 7
    print("Top 2 products (id, sales):", tracker.top_k(2))

    # Delivery route planner
    planner = DeliveryRoutePlanner()
    planner.add_route("WarehouseA", "City1", 10.0)
    planner.add_route("WarehouseA", "City2", 20.0)
    planner.add_route("City1", "City2", 5.0)
    planner.add_route("City2", "City3", 7.0)

    dist, path = planner.shortest_path("WarehouseA", "City3")
    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_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
```