

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.peak()}")
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          self.items.append(item)
13
14      def dequeue(self):
15          """Remove and return the front item from the queue.
16
17          Returns:
18             The front item of the queue
19
20          Raises:
21             IndexError: If the queue is empty
22          """
23          if self.is_empty():
24              raise IndexError("Cannot dequeue from an empty queue")
25          return self.items.pop(0)
26
27      def peek(self):
28          """Return the front item of the queue without removing it.
29
30          Returns:
31             The front item of the queue
32
33          Raises:
34             IndexError: If the queue is empty
35          """
36          if self.is_empty():
37              raise IndexError("Cannot peek at an empty queue")
38          return self.items[0]
39
40      def size(self):
41          """Return the number of items in the queue.
42

```

```

43          Returns:
44             The number of items in the queue
45          """
46          return len(self.items)
47
48      def is_empty(self):
49          """Check if the queue is empty.
50
51          Returns:
52             True if the queue is empty, False otherwise
53          """
54          return len(self.items) == 0
55
56
57 # Example usage and testing
58 if __name__ == "__main__":
59     # Create a new queue
60     q = Queue()
61
62     # Test enqueue
63     print("Enqueueing items: 1, 2, 3, 4, 5")
64     q.enqueue(1)
65     q.enqueue(2)
66     q.enqueue(3)
67     q.enqueue(4)
68     q.enqueue(5)
69
70     # Test size
71     print(f"Queue size: {q.size()}")
72
73     # Test peek
74     print(f"Peek at front: {q.peek()}")
75
76     # Test dequeue
77     print("\nDequeueing items:")
78     while not q.is_empty():
79         print(f"  Dequeued: {q.dequeue()}, Remaining size: {q.size()}")
80
81     # Test empty queue
82     print(f"Queue is empty: {q.is_empty()}")
83
84     # Test error handling
85     try:
86         q.dequeue()
87     except IndexError as e:
88         print(f"Error caught: {e}")
89
90     try:
91         q.peek()
92     except IndexError as e:
93         print(f"Error caught: {e}")

```

Output:

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

Dequeuing 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

Dequeuing 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

Dequeuing 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

Dequeuing 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

Dequeuing 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         new_node = Node(data)
22
23         # If the list is empty, make the new node the head
24         if self.head is None:
25             self.head = new_node
26         else:
27             # Traverse to the end of the list
28             current = self.head
29             while current.next is not None:
30                 current = current.next
31             # Insert the new node at the end
32             current.next = new_node
33
34     def insert_at_beginning(self, data):
35         """
36         Insert a new node at the beginning of the linked list
37
38         Args:
39             data: The data to be inserted into the linked list
40         """
41         new_node = Node(data)
42         new_node.next = self.head
43         self.head = new_node
44
45     def display(self):
46         """
47         Display all elements in the linked list
48         """
49         if self.head is None:
50             print("Linked list is empty")
51             return
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 # Example usage
64 if __name__ == "__main__":
65     # Create a new Linked List
66     ll = SinglyLinkedList()
67
68     # Insert some elements
69     print("Inserting elements into the linked list...")
70     ll.insert(10)
71     ll.insert(20)
72     ll.insert(30)
73     ll.insert(40)
74
75     # Display the linked list
76     print("\nLinked List contents:")
77     ll.display()
78
79     # Insert at beginning
80     print("\nInserting 5 at the beginning...")
81     ll.insert_at_beginning(5)
82     ll.display()
83
84     # Create an empty list
85     print("\nCreating an empty linked list:")
86     empty_ll = SinglyLinkedList()
87     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 if __name__ == "__main__":
46     bst = BST()
47     for x in [7, 3, 9, 1, 5, 8, 10]:
48         bst.insert(x)
49     print("In-order:", bst.inorder())
50

```

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.

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         self.directed = directed
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         """Add a vertex if it doesn't already exist."""
16         if v not in self.adj:
17             self.adj[v] = set[object]()
18
19     def add_edge(self, u: object, v: object) -> None:
20         """Add an edge u -> v (and v -> u if undirected)."""
21         self.add_vertex(u)
22         self.add_vertex(v)
23         self.adj[u].add(v)
24         if not self.directed:
25             self.adj[v].add(u)
26
27     def remove_edge(self, u: object, v: object) -> None:
28         """Remove an edge u -> v (and v -> u if undirected), if present."""
29         if u in self.adj:
30             self.adj[u].discard(v)
31         if not self.directed and v in self.adj:
32             self.adj[v].discard(u)
33
34     def remove_vertex(self, v: object) -> None:
35         """Remove a vertex and all edges incident to it."""
36         if v not in self.adj:
37             return
38
39         # Remove edges from neighbors to v
40         for n in list[object](self.adj[v]):
41             self.remove_edge(v, n)
42
43         # In directed graphs, also remove incoming edges to v
44         if self.directed:
45             for u in self.adj:
46                 self.adj[u].discard(v)
47
48         del self.adj[v]
49
50     def neighbors(self, v: object) -> list[object]:
51         """Return neighbors of v as a sorted list when possible."""
52         if v not in self.adj:
53             return []
54         try:
55             return sorted(self.adj[v])
56         except TypeError:
57             return list[object](self.adj[v])
58
59     def bfs(self, start: object) -> list[object]:
60         """Breadth-first traversal order starting from start."""
61         if start not in self.adj:
62             return []
63
64         visited = {start}
65         queue = [start]
66         order: list[object] = []

```

```

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 dfs(self, start: object) -> list[object]:
79     """Depth first traversal order starting from start."""
80     if start not in self.adj:
81         return []
82
83     visited: set[object] = set[object]()
84     order: list[object] = []
85
86     def _visit(v: object) -> None:
87         visited.add(v)
88         order.append(v)
89         for n in self.neighbors(v):
90             if n not in visited:
91                 _visit(n)
92
93     _visit(start)
94     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[0] # 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 if __name__ == "__main__":
35     pq = PriorityQueue()
36     pq.push("low", 5)
37     pq.push("urgent", 1)
38     pq.push("medium", 3)
39     pq.push("also urgent (arrives later)", 1)
40
41     while not pq.empty():
42         item, pr = pq.pop()
43         print(pr, item)
44
45     # Max-priority example (bigger number = served first):
46     maxpq = PriorityQueue()
47     for item, pr in [("A", 10), ("B", 2), ("C", 10)]:
48         maxpq.push(item, -pr) # negate priority
49
50     print("max first:", maxpq.pop()) # returns (item, neg_priority)
51

```

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:

[illegible]

```

100     book = self.find(tshs)
101     if not book:
102         return False
103     self._book[book] = 1
104     book.available_copies -= 1
105     return True
106
107 def _ordering_in_order(self) -> List[Book]:
108     out: List[Book] = []
109     self._in_order(self._root, out)
110     return out
111
112 def _insert(self, node: Optional[BookSTNode], tshs: str, book: Book) -> BookSTNode:
113     if node is None:
114         return BookSTNode(tshs, book)
115     if tshs < node.key:
116         node.left = self._insert(node.left, tshs, book)
117     else:
118         node.right = self._insert(node.right, tshs, book)
119     return node
120
121 def _in_order(self, node: Optional[BookSTNode], out: List[Book]) -> None:
122     if node is None:
123         return
124     self._in_order(node.left, out)
125     out.append(node.book)
126     self._in_order(node.right, out)
127
128 # =====
129 # 4) Bus Scheduling System (Graph)
130 # =====
131
132 class BusNetwork:
133     """
134     Data structure: Graph (adjacency list)
135     stop -> list of (neighbor, stop, travel_minutes)
136     .shortest_path uses Dijkstra (non-negative weights).
137     """
138
139     def __init__(self) -> None:
140         self._adj: Dict[str, List[Tuple[str, int]]] = {}
141
142     def add_stop(self, stop: str) -> None:
143         self._adj.setdefault(stop, [])
144
145     def add_route(self, s: str, t: str, minutes: int, bidirectional: bool = True) -> None:
146         if minutes < 0:
147             raise ValueError("minutes must be non-negative")
148         self.add_stop(s)
149         self.add_stop(t)
150         self._adj[s].append((t, minutes))
151         if bidirectional:
152             self._adj[t].append((s, minutes))
153
154     def shortest_path(self, start: str, end: str) -> Tuple[int, List[str]]:
155         if start not in self._adj or end not in self._adj:
156             raise KeyError("start/end stop not found")
157
158         dist: Dict[str, int] = {start: 0}
159         prev: Dict[str, Optional[str]] = {start: None}
160         pq: List[Tuple[int, str]] = [(0, start)]
161
162         while pq:
163             d, u = heapq.heappop(pq)
164             if d > dist.get(u, 10**18):
165                 continue
166             if u == end:
167                 for v, w in self._adj[u]:
168                     nd = d + w
169                     if nd < dist.get(v, 10**18):
170                         dist[v] = nd
171                         prev[v] = u
172                         heapq.heappush(pq, (nd, v))
173             if end not in dist:
174                 return (10**18, [])
175
176         # Reconstruct path
177         path: List[str] = []
178         cur: Optional[str] = end
179         while cur is not None:
180             path.append(cur)
181             cur = prev.get(cur)
182         path.reverse()
183         return dist[end], path
184
185 # =====
186 # 5) Cafeteria Order Queue (Priority Queue)
187 # =====

```

```

100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Output:

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

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

Prompt:

Solve this clearly and concisely.

Design a Smart E-Commerce Platform with:

Shopping Cart Management – Add/remove products dynamically

Order Processing System – Process orders in placement order

Top-Selling Products Tracker – Rank products by sales count

Product Search Engine – Fast lookup using product ID

Delivery Route Planning – Connect warehouses and delivery locations

Choose the most appropriate data structure for each feature from:

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

Output as a table:

Feature | Data Structure | 2–3 sentence justification

Code:

```

1  from collections import deque
2  import heapq
3  from typing import Dict, List, Tuple, Optional
4
5
6  # -----
7  # Product model
8  # -----
9  class Product:
10     def __init__(self, product_id: int, name: str, price: float):
11         self.id = product_id
12         self.name = name
13         self.price = price
14
15     def __repr__(self):
16         return f"Product(id={self.id}, name='{self.name}', price={self.price})"
17
18
19 # -----
20 # Product Search Engine (Hash Table)
21 # -----
22 class ProductSearchEngine:
23     def __init__(self):
24         # 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

```

```

81         else:
82             node.quantity -= quantity
83         return
84         prev = node
85         node = node.next
86
87     def list_items(self) -> List[Tuple[Product, int]]:
88         result = []
89         node = self.head
90         while node:
91             result.append((node.product, node.quantity))
92             node = node.next
93         return result
94
95     def total_price(self) -> float:
96         return sum(node.product.price * node.quantity
97                     for node in self._iter_nodes())
98
99     def _iter_nodes(self):
100         node = self.head
101         while node:
102             yield node
103             node = node.next
104
105 # -----
106 # Order Processing System (Queue)
107 # -----
108 class Order:
109     next_id = 1
110
111     def __init__(self, cart_snapshot: List[Tuple[Product, int]]):
112         self.id = Order.next_id
113         Order.next_id += 1
114         self.items = cart_snapshot # List of (Product, quantity)
115
116     def __repr__(self):
117         return f"Order(id={self.id}, items=[{(p.id, q) for p, q in self.items}])"
118
119
120 class OrderProcessingSystem:
121     def __init__(self):
122         # Queue of orders (FIFO)
123         self.queue: deque[Order] = deque[Order]()
124
125     def place_order(self, cart: ShoppingCart) -> Order:
126         order = Order(cart.list_items())
127         self.queue.append(order)
128         return order
129
130     def process_next_order(self) -> Optional[Order]:
131         if not self.queue:
132             return None
133         return self.queue.popleft()
134
135     def pending_orders(self) -> int:
136         return len(self.queue)
137
138 # -----
139 # Top-Selling Products Tracker (Priority Queue / Max-Heap)
140 # -----
141 class TopSellingProductsTracker:
142     def __init__(self):
143         # product_id -> sales_count
144         self.sales: Dict[int, int] = {}
145         # priority queue entries: (sales_count, product_id)
146         self.heap: List[Tuple[int, int]] = []
147
148     def record_sale(self, product_id: int, quantity: int = 1):
149         self.sales[product_id] = self.sales.get(product_id, 0) + quantity
150         # Push new priority entry; lazy update (we'll verify against self.sales on pop)
151         heapq.heappush(self.heap, (-self.sales[product_id], product_id))
152
153     def top_k(self, k: int) -> List[Tuple[int, int]]:
154         """
155         Returns list of (product_id, sales_count) for top k products.
156         Uses lazy removal from the heap to keep it consistent.
157         """
158

```

```

159     result = []
160     seen = set[bool]()
161
162     while self.heap and len(result) < k:
163         neg_sales, pid = heapq.heappop(self.heap)
164         current_sales = self.sales.get(pid, 0)
165
166         if current_sales == -neg_sales and pid not in seen:
167             result.append(pid, current_sales)
168             seen.add(pid)
169
170         # Push back the elements we popped that are still valid
171         for pid in seen:
172             heapq.heappush(self.heap, (-self.sales[pid], pid))
173
174     return result
175
176 # -----
177 # Delivery Route Planning (Graph + Dijkstra)
178 # -----
179 class DeliveryRoutePlanner:
180     def __init__(self):
181         # Graph as adjacency list: node -> List of (neighbor, distance)
182         self.graph: Dict[str, List[Tuple[str, float]]] = {}
183
184     def add_location(self, name: str):
185         if name not in self.graph:
186             self.graph[name] = []
187
188     def add_route(self, from_loc: str, to_loc: str, distance: float, bidirectional: bool = True):
189         self.add_location(from_loc)
190         self.add_location(to_loc)
191         self.graph[from_loc].append((to_loc, distance))
192         if bidirectional:
193             self.graph[to_loc].append((from_loc, distance))
194
195     def shortest_path(self, start: str, end: str) -> Tuple[float, List[str]]:
196         """
197         Dijkstra's algorithm: returns (distance, path).
198         Distance is float('inf') if no path exists.
199         """
200         if start not in self.graph or end not in self.graph:
201             return float('inf'), []
202
203         # min-heap: (distance, node, path)
204         heap = [(0.0, start, [start])]
205         visited = set[bool]()
206
207         while heap:
208             dist, node, path = heapq.heappop(heap)
209             if node in visited:
210                 continue
211             visited.add(node)
212
213             if node == end:
214                 return dist, path
215
216             for neighbor, weight in self.graph[node]:
217                 if neighbor not in visited:
218                     heapq.heappush(heap, (dist + weight, neighbor, path + [neighbor]))
219
220         return float('inf'), []
221
222 # -----
223 # Example usage
224 # -----
225 if __name__ == "__main__":
226     # Product search engine
227     search_engine = ProductSearchEngine()
228     p1 = Product(1, "Laptop", 1000.0)
229     p2 = Product(2, "Phone", 500.0)
230     p3 = Product(3, "Headphones", 100.0)
231     for p in (p1, p2, p3):
232         search_engine.add_product(p)

```

```

230 # 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:\2403A51103\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

```