

School of Computer Science and Artificial Intelligence

Lab Assignment # 11.1

Program	: B. Tech (CSE)
Specialization	: -
Course Title	: AI Assisted Coding
Course Code	: 23CS002PC304
Semester	: II
Academic Session	: 2025-2026
Name of Student	: I.Sathwik Rajeshwara Chary
Enrollment No.	: 2403A51L03
Batch No.	: 51
Date	: 17/02/26

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

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

Sample code:

```
class Stack:  
    Pass
```

Prompt:

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

Code:

```

1 class Stack:
2     def __init__(self):
3         """Initialize an empty stack."""
4         self.items = []
5
6     def push(self, item):
7         """Add an item to the top of the stack.
8
9         Args:
10            item: The item to be added to the stack.
11        """
12        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

```

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

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         elements = []
55         while current is not None:
56             elements.append(str(current.data))
57             current = current.next
58
59         # Display in format: data1 -> data2 -> data3 -> None
60         print(" -> ".join(elements) + " -> None")
61
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.

Sample Input Code:

```
class Graph:
    pass
```

Prompt:

implement a graph using an adjacency list with code

Code:

```

1 class Graph:
2     """
3     Graph implemented using an adjacency list.
4
5     - By default the graph is undirected.
6     - Set directed=True for a directed graph.
7     """
8
9     def __init__(self, directed: bool = False):
10         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 if __name__ == "__main__":
103     g = Graph(directed=False) # change to True for a directed graph
104     g.add_edge("A", "B")
105     g.add_edge("A", "C")
106     g.add_edge("B", "D")
107     g.add_edge("C", "D")
108     g.add_edge("D", "E")
109
110     print("Adjacency list:")
111     print(g)
112     print()
113     print("BFS from A:", g.bfs("A"))
114     print("DFS from A:", g.dfs("A"))

```

Output:

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

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

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

BFS from A: ['A', 'B', 'C', 'D', 'E']
DFS from A: ['A', 'B', 'D', 'C', 'E']
BFS from A: ['A', 'B', 'C', 'D', 'E']
DFS from A: ['A', 'B', 'D', 'C', 'E']
DFS from A: ['A', 'B', 'D', 'C', 'E']
```

Task Description #7 – Priority Queue

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

Sample Input Code:

```
class PriorityQueue:
    pass
```

Prompt:

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

Code:

```

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

```

Output:

```

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

```

Task Description #8 – Deque

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

Sample Input Code:

```

class DequeDS:
    pass

```

Prompt:

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

Code:

```

1  from __future__ import annotations
2
3  from collections import deque
4  from typing import Deque, Generic, Iterator, Optional, TypeVar
5
6  T = TypeVar("T")
7
8
9  class DequeDS(Generic[T]):
10     """
11     Double-ended queue (deque) implemented using collections.deque.
12     Supports O(1) append/pop operations on both ends.
13     """
14
15     def __init__(self, items: Optional[Iterator[T]] = None) -> None:
16         self._dq: Deque[T] = deque[T](items or [])
17
18     # --- Add operations ---
19     def add_front(self, item: T) -> None:
20         """Insert item at the front (left)."""
21         self._dq.appendleft(item)
22
23     def add_rear(self, item: T) -> None:
24         """Insert item at the rear (right)."""
25         self._dq.append(item)
26
27     # --- Remove operations ---
28     def remove_front(self) -> T:
29         """Remove and return the front (left) item."""
30         if self.is_empty():
31             raise IndexError("remove_front from empty deque")
32         return self._dq.popleft()
33
34     def remove_rear(self) -> T:
35         """Remove and return the rear (right) item."""
36         if self.is_empty():
37             raise IndexError("remove_rear from empty deque")
38         return self._dq.pop()
39
40     # --- Peek operations ---
41     def peek_front(self) -> T:
42         """Return the front (left) item without removing it."""
43         if self.is_empty():
44             raise IndexError("peek_front from empty deque")
45         return self._dq[0]
46
47     def peek_rear(self) -> T:
48         """Return the rear (right) item without removing it."""
49         if self.is_empty():
50             raise IndexError("peek_rear from empty deque")
51         return self._dq[-1]
52
53     # --- Utility ---
54     def is_empty(self) -> bool:
55         return len(self._dq) == 0
56
57     def size(self) -> int:
58         return len(self._dq)
59
60     def clear(self) -> None:
61         self._dq.clear()
62
63     def __len__(self) -> int:
64         return len(self._dq)
65
66     def __iter__(self) -> Iterator[T]:
67         return iter(self._dq)
68
69     def __repr__(self) -> str:
70         return f"DequeDS({list[T](self._dq)!r})"
71
72
73 if __name__ == "__main__":
74     d = DequeDS[int]()
75     d.add_front(10)    # [10]
76     d.add_rear(20)    # [10, 20]
77     d.add_front(5)    # [5, 10, 20]
78     print("Deque:", d)
79     print("Front:", d.peek_front())
80     print("Rear:", d.peek_rear())
81     print("Remove front:", d.remove_front()) # 5
82     print("Remove rear:", d.remove_rear())   # 20
83     print("Deque now:", d)

```

Output:

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

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

Prompt:

Solve this clearly and concisely.

Design a Campus Resource Management System code with:

1. Student Attendance Tracking
2. Event Registration System
3. Library Book Borrowing
4. Bus Scheduling System
5. Cafeteria Order Queue

Choose the best data structure for each feature from:

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

Output as a table:

Feature | Data Structure | 2–3 sentence justification

Code:


```

1 from __future__ import annotations
2
3 from dataclasses import dataclass
4 from collections import deque
5 import heapq
6 import itertools
7 from typing import Deque, Dict, List, Optional, Set, Tuple
8
9
10 # =====
11 # 1) Student Attendance Tracking (Hash Table)
12 # =====
13
14 class AttendanceTracker:
15     """
16     Data structure: Hash Table (Python dict)
17     student_id -> (date_str -> present_bool)
18     """
19
20     def __init__(self) -> None:
21         self._records: Dict[str, Dict[str, bool]] = {}
22
23     def mark(self, student_id: str, date: str, present: bool) -> None:
24         self._records.setdefault(student_id, {})[date] = present
25
26     def is_present(self, student_id: str, date: str) -> Optional[bool]:
27         return self._records.get(student_id, {}).get(date)
28
29     def attendance_average(self, student_id: str) -> float:
30         days = self._records.get(student_id, {})
31         if not days:
32             return 0.0
33         present_count = sum(1 for v in days.values() if v)
34         return (present_count / len(days)) * 100.0
35
36 # =====
37 # 2) Food Registration System (Deque)
38 # =====
39
40 class FoodRegistrationSystem:
41     """
42     Data structure: Queue (collections.deque)
43     . FIFO registration requests = FIFO waitlist.
44     """
45
46     @dataclass(frozen=True)
47     class Event:
48         event_id: str
49         name: str
50         capacity: int
51
52     def __init__(self) -> None:
53         self._events: Dict[str, Event] = {}
54         self._confirmed: Dict[str, Deque[str]] = {} # event_id -> queue(student_id)
55         self._waitlist: Dict[str, Deque[str]] = {} # event_id -> queue(student_id)
56
57     def create_event(self, event_id: str, name: str, capacity: int) -> None:
58         if capacity < 0:
59             raise ValueError("Capacity must be >= 0")
60         self._events[event_id] = self.Event(event_id, name, capacity)
61         self._confirmed.setdefault(event_id, deque())
62         self._waitlist.setdefault(event_id, deque())
63
64     def request_registration(self, event_id: str, student_id: str) -> None:
65         self._enqueue_event(event_id)
66         if student_id in self._confirmed[event_id]:
67             return
68         if student_id in self._waitlist[event_id] or student_id in self._waitlist[event_id]:
69             self._requests[event_id].append(student_id)
70
71     def process_next_request(self, event_id: str) -> Optional[str]:
72         """
73         Processes ONE pending request in FIFO order.
74         Returns the student_id that got confirmed (or None if no request).
75         """
76         self._dequeue_event(event_id)
77         q = self._requests[event_id]
78         if not q:
79             return None
80         student_id = q.popleft()
81         if len(self._confirmed[event_id]) < self._events[event_id].capacity:
82             self._confirmed[event_id].append(student_id)
83             return student_id
84         self._waitlist[event_id].append(student_id)
85         return None
86
87     def cancel_registration(self, event_id: str, student_id: str) -> None:
88         self._dequeue_event(event_id)
89         if student_id in self._confirmed[event_id]:
90             self._confirmed[event_id].remove(student_id)
91             self._requests[event_id].remove(student_id)
92             return
93         self._requests[event_id].remove(student_id)
94         self._remove_from_queue(self._waitlist[event_id], student_id)
95         self._remove_from_queue(self._waitlist[event_id], student_id)
96
97     def confirm_event(self, event_id: str) -> List[str]:
98         self._process_event(event_id)
99         return sorted(self._confirmed[event_id])
100
101     def waitlist_list(self, event_id: str) -> List[str]:
102         self._process_event(event_id)
103         return list(self._waitlist[event_id])
104
105     def process_event(self, event_id: str) -> None:
106         if len(self._confirmed[event_id]) > self._events[event_id].capacity:
107             return
108         w = self._waitlist[event_id]
109         while w and len(self._confirmed[event_id]) < self._events[event_id].capacity:
110             self._confirmed[event_id].append(w.popleft())
111
112     def remove_from_queue(self, q: Deque[str], student_id: str) -> None:
113         # queue doesn't support fast middle removal, rebuild for clarity.
114         if not q:
115             return
116         new_q = deque()
117         for x in q:
118             if x != student_id:
119                 new_q.append(x)
120         q.clear()
121         q.extend(new_q)
122
123     def remove_event(self, event_id: str) -> None:
124         if event_id not in self._events:
125             raise KeyError(f"Unknown event_id: {event_id}")
126
127 # =====
128 # 3) Library Book Borrowing (BST)
129 # =====
130
131 @dataclass
132 class Book:
133     isbn: str
134     title: str
135     total_copies: int
136     available_copies: int
137
138 class BookSTNode:
139     def __init__(self, key: str, val: Book) -> None:
140         self.key = key
141         self.val = val
142         self.left: Optional[BookSTNode] = None
143         self.right: Optional[BookSTNode] = None
144
145 class LibrarySystem:
146     """
147     Data structure: BST (By ISBN) for catalog/inventory search and ordered traversal.
148     Borrowing increments available copies, returning increments.
149     """
150
151     def __init__(self) -> None:
152         self._root: Optional[BookSTNode] = None
153         self._books: Dict[str, Book] = {} # (student_id, isbn) -> current borrowing
154
155     def add_book(self, isbn: str, title: str, copies: int) -> None:
156         if copies <= 0:
157             raise ValueError("copies must be > 0")
158         existing = self.find(isbn)
159         if existing:
160             existing.total_copies += copies
161             existing.available_copies += copies
162             return
163         book = Book(isbn=title, title=title, total_copies=copies, available_copies=copies)
164         self._root = self._insert(self._root, isbn, book)
165
166     def find(self, isbn: str) -> Optional[Book]:
167         node = self._root
168         while node:
169             if isbn < node.key:
170                 node = node.left
171             elif isbn > node.key:
172                 node = node.right
173             else:
174                 return node
175         return None
176
177     def borrow(self, student_id: str, isbn: str) -> bool:
178         book = self.find(isbn)
179         if not book or book.available_copies <= 0:
180             return False
181         book.available_copies -= 1
182         self._books[(student_id, isbn)] = self._books.get((student_id, isbn), 0) + 1
183         return True
184
185     def return_book(self, student_id: str, isbn: str) -> bool:
186         key = (student_id, isbn)
187         if key in self._books:
188             return True
189         return False

```



```

100     book = self.find(book)
101     if not book:
102         return False
103     self.available_copies -= 1
104     return True
105
106 def catalog_in_order(self) -> list(Book):
107     out_list[book] = []
108     self.catalog_in_order_recursive(out_list, None)
109     return out_list
110
111 def insert(self, node: Optional[BookNode], lib: str, book: Book) -> BookNode:
112     if node is None:
113         return BookNode(lib, book)
114     if lib < node.lib:
115         node.left = self.insert(node.left, lib, book)
116     else:
117         node.right = self.insert(node.right, lib, book)
118     return node
119
120 def in_order(self, node: Optional[BookNode], out: list(Book)) -> None:
121     if node is None:
122         return
123     self.in_order(node.left, out)
124     out.append(node.book)
125     self.in_order(node.right, out)
126
127 # =====
128 # 4) Bus Scheduling System (Graph)
129 # =====
130
131 class BusNetwork:
132     """
133     Data structure: Graph (adjacency list)
134     stop -> list of (neighbor_stop, travel_minutes)
135     - shortest path uses Dijkstra (non-negative weights).
136     """
137
138     def __init__(self) -> None:
139         self.adj: dict[str, list[tuple[str, int]]] = {}
140
141     def add_stop(self, stop: str) -> None:
142         self.adj.setdefault(stop, [])
143
144     def add_route(self, s: str, t: str, minutes: int, bidirectional: bool = True) -> None:
145         if minutes < 0:
146             raise ValueError("minutes must be non-negative")
147         self.add_stop(s)
148         self.add_stop(t)
149         self.adj[s].append((t, minutes))
150         if bidirectional:
151             self.adj[t].append((s, minutes))
152
153     def shortest_path(self, start: str, end: str) -> tuple[list, list[str]]:
154         if start not in self.adj or end not in self.adj:
155             raise ValueError("start/end stops not found")
156
157         dist: dict[str, int] = {start: 0}
158         prev: dict[str, Optional[str]] = {start: None}
159         pq: list[tuple[int, str]] = [(0, start)]
160
161         while pq:
162             d, u = heapq.heappop(pq)
163             if d != dist.get(u, 10**18):
164                 continue
165             if u == end:
166                 break
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
174         if end not in dist:
175             return (10**18, [])
176
177         # Reconstruct path
178         path: list[str] = []
179         cur: Optional[str] = end
180         while cur is not None:
181             path.append(cur)
182             cur = prev.get(cur)
183         path.reverse()
184         return dist[end], path
185
186 # =====
187 # 5) Cafeteria Order Queue (Priority Queue)
188 # =====
189
190 @dataclass(frozen=True)
191 class CafeteriaOrder:
192     order_id: int
193     student_id: str
194     item: str
195     priority: int # Higher number == Higher priority
196
197 class CafeteriaOrderSystem:
198     """
199     Data structure: Priority Queue (heap)
200     - Serve highest priority first; tie-break by arrival order.
201     """
202
203     def __init__(self) -> None:
204         self.heap: list[tuple[int, CafeteriaOrder]] = []
205         self.counter = itertools.count()
206
207     def place_order(self, student_id: str, item: str, priority: int = 0) -> CafeteriaOrder:
208         order_id = next(self.counter)
209         order = CafeteriaOrder(order_id, student_id, item, item, priority)
210         # Push to heap, insert priority on-heap priority over first
211         heapq.heappush(self.heap, (-priority, order_id, order))
212         return order
213
214     def serve_next(self) -> Optional[CafeteriaOrder]:
215         if not self.heap:
216             return None
217         _, order_id, order = heapq.heappop(self.heap)
218         return order
219
220     def pending_count(self) -> int:
221         return len(self.heap)
222
223 # =====
224 # Demo (optional)
225 # =====
226
227 def main() -> None:
228     # Attends
229     att = AttendanceTracker()
230     att.mark("S1", "2024-03-17", True)
231     att.mark("S1", "2024-03-18", False)
232     print(f"Attendance S1: {att.attendance_percent('S1', 2)}")
233
234     # Events
235     events = EventRegistrationSystem()
236     events.create_event("T100", "AI Workshop", capacity=2)
237     for sid in ["S1", "S2", "S3"]:
238         events.request_registration("T100", sid)
239     events.process_next_request("T100")
240     print(f"Confirmed S1: {events.confirmed_list('T100')}")
241     print(f"Waitlist T100: {events.waitlist_list('T100')}")
242     events.cancel_registration("T100", "S2")
243     print(f"After cancel S2 confirmed: {events.confirmed_list('T100')}")
244
245     # Library (BFS)
246     lib = LibrarySystem()
247     lib.add_book("9781449159999", "Effective Java", copies=2)
248     lib.add_book("9781492011677", "Fluent Python", copies=1)
249     print(f"Borrow fluent Python: {lib.borrow('lib', '9781492011677')}")
250     print(f"Borrow fluent Python again: {lib.borrow('lib', '9781492011677')}")
251     print(f"Catalog in order: {lib.catalog_in_order()}")
252
253     # Bus network (Graph)
254     buses = BusNetwork()
255     buses.add_route("Hostel", "Mata", 8)
256     buses.add_route("Mata", "Library", 4)
257     buses.add_route("Hostel", "Cafeteria", 5)
258     buses.add_route("Cafeteria", "Library", 4)
259     minutes, path = buses.shortest_path("Hostel", "Library")
260     print(f"Shortest bus path Hostel->Library: {path}, {minutes}, minutes")
261
262     # Cafeteria (Priority Queue)
263     cafe = CafeteriaOrderSystem()
264     cafe.place_order("S1", "Sandwich", priority=0)
265     cafe.place_order("S2", "Coffee", priority=2)
266     cafe.place_order("S3", "Burger", priority=1)
267     print("Serve order", cafe.serve_next())
268     print("Serve order", cafe.serve_next())
269
270     if __name__ == "__main__":
271         main()
272

```

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
10 class Product:
11     def __init__(self, product_id: int, name: str, price: float):
12         self.id = product_id
13         self.name = name
14         self.price = price
15
16     def __repr__(self):
17         return f"Product(id={self.id}, name='{self.name}', price={self.price})"
18
19 # -----
20 # Product Search Engine (Hash Table)
21 # -----
22
23 class ProductSearchEngine:
24     def __init__(self):
25         # Hash Tables: product_id -> Product
26         self.products: Dict[int, Product] = {}
27
28     def add_product(self, product: Product):
29         self.products[product.id] = product
30
31     def get_product(self, product_id: int) -> Optional[Product]:
32         return self.products.get(product_id)
33
34     def remove_product(self, product_id: int):
35         self.products.pop(product_id, None)
36
37 # -----
38 # Shopping Cart (linked list)
39 # -----
40
41 class CartNode:
42     def __init__(self, product: Product, quantity: int):
43         self.product = product
44         self.quantity = quantity
45         self.next: Optional[CartNode] = None
46
47
48 class ShoppingCart:
49     def __init__(self):
50         self.head: Optional[CartNode] = None
51
52     def add_product(self, product: Product, quantity: int = 1):
53         """
54         If product already exists in the list, increase quantity.
55         Otherwise, add new node at the front (O(1) insertion).
56         """
57         node = self.head
58         while node:
59             if node.product.id == product.id:
60                 node.quantity += quantity
61                 return
62             node = node.next
63
64         new_node = CartNode(product, quantity)
65         new_node.next = self.head
66         self.head = new_node
67
68     def remove_product(self, product_id: int, quantity: int = None):
69         """
70         Remove some or all quantity of a product.
71         If quantity is None or reaches 0, remove the node.
72         """
73         prev = None
74         node = self.head
75
76         while node:
77             if node.product.id == product_id:
78                 if quantity is None or node.quantity <= quantity:
79                     # delete the node
80                     if prev:
81                         prev.next = node.next
82                     else:
83                         self.head = node.next
84                 else:
85                     node.quantity -= quantity
86                     return
87                 prev = node
88                 node = node.next
89
90     def list_items(self) -> List[Tuple[Product, int]]:
91         result = []
92         node = self.head
93         while node:
94             result.append((node.product, node.quantity))
95             node = node.next
96         return result
97
98     def total_price(self) -> float:
99         return sum(node.product.price * node.quantity
100                    for node in self._iter_nodes())
101
102     def _iter_nodes(self):
103         node = self.head
104         while node:
105             yield node
106             node = node.next
107
108 # -----
109 # Order Processing System (Queue)
110 # -----
111
112 class Order:
113     __next_id = 1
114
115     def __init__(self, cart_snapshot: List[Tuple[Product, int]]):
116         self.id = Order.__next_id
117         Order.__next_id += 1
118         self.items = cart_snapshot # list of (Product, quantity)
119
120     def __repr__(self):
121         return f"Order(id={self.id}, items=[{(p.id, q) for p, q in self.items}])"
122
123
124 class OrderProcessingSystem:
125     def __init__(self):
126         # Queue of orders (FIFO)
127         self.queue: deque[Order] = deque[Order]()
128
129     def place_order(self, cart: ShoppingCart) -> Order:
130         order = Order(cart.list_items())
131         self.queue.append(order)
132         return order
133
134     def process_next_order(self) -> Optional[Order]:
135         if not self.queue:
136             return None
137         return self.queue.popleft()
138
139     def pending_orders(self) -> int:
140         return len(self.queue)
141
142 # -----
143 # Top-Selling Products Tracker (Priority Queue / Max-Heap)
144 # -----
145
146 class TopSellingProductsTracker:
147     def __init__(self):
148         # product_id -> sales_count
149         self.sales: Dict[int, int] = {}
150         # priority queue: entries < (sales_count, product_id)
151         self.heap: List[Tuple[int, int]] = []
152
153     def record_sale(self, product_id: int, quantity: int = 1):
154         self.sales[product_id] = self.sales.get(product_id, 0) + quantity
155         # Push new priority entry; lazy update (we'll verify against self.sales on pop)
156         heapq.heappush(self.heap, (-self.sales[product_id], product_id))
157
158     def top_k(self, k: int) -> List[Tuple[int, int]]:
159         """
160         Returns list of (product_id, sales_count) for top k products.
161         Uses lazy removal from the heap to keep it consistent.
162         """

```

```

162         result = []
163         seen = set()
164
165         while self.heap and len(result) < k:
166             neg_sales, pid = heapq.heappop(self.heap)
167             current_sales = self.sales.get(pid, 0)
168
169             if current_sales == -neg_sales and pid not in seen:
170                 result.append(pid, current_sales)
171                 seen.add(pid)
172
173             # push back the elements we popped that are still valid
174             for pid in seen:
175                 heapq.heappush(self.heap, (-self.sales[pid], pid))
176
177         return result
178
179
180 # =====
181 # Delivery Route Planning (Graph = DiGraph)
182 # =====
183 class DeliveryRoutePlanner:
184     def __init__(self):
185         # Graph as adjacency list: node -> list of (neighbor, distance)
186         self.graph = Dict[str, List[Tuple[str, float]]] = {}
187
188     def add_location(self, name: str):
189         if name not in self.graph:
190             self.graph[name] = []
191
192     def add_route(self, from_loc: str, to_loc: str, distance: float, bidirectional: bool = True):
193         self.add_location(from_loc)
194         self.add_location(to_loc)
195         self.graph[from_loc].append((to_loc, distance))
196         if bidirectional:
197             self.graph[to_loc].append((from_loc, distance))
198
199     def shortest_path(self, start: str, end: str) -> Tuple[float, List[str]]:
200         # Dijkstra's algorithm: returns (distance, path).
201         # Distance is float('inf') if no path exists.
202         if start not in self.graph or end not in self.graph:
203             return float('inf'), []
204
205         # min-heap: (distance, node, path)
206         heap = [(0.0, start, [start])]
207         visited = set()
208
209         while heap:
210             dist, node, path = heapq.heappop(heap)
211             if node in visited:
212                 continue
213             visited.add(node)
214
215             if node == end:
216                 return dist, path
217
218             for neighbor, weight in self.graph[node]:
219                 if neighbor not in visited:
220                     heapq.heappush(heap, (dist + weight, neighbor, path + [neighbor]))
221
222         return float('inf'), []
223
224
225 # =====
226 # Example usage
227 # =====
228 if __name__ == "__main__":
229     # Product search engine
230     search_engine = ProductSearchEngine()
231     p1 = Product(1, "Laptop", 1000.0)
232     p2 = Product(2, "Phone", 500.0)
233     p3 = Product(3, "Headphones", 100.0)
234     for p in (p1, p2, p3):
235         search_engine.add_product(p)
236
237     # Shopping cart
238     cart = ShoppingCart()
239     cart.add_product(search_engine.get_product(1), 1)
240     cart.add_product(search_engine.get_product(2), 2)
241     cart.add_product(search_engine.get_product(3), 3)
242     cart.remove_product(1, 1) # remove 1 laptop
243
244     print("Cart items:", cart.list_items())
245     print("Total price:", cart.total_price())
246
247     # Order processing
248     ops = OrderProcessingSystem()
249     order1 = ops.place_order(cart)
250     print("Placed order:", order1)
251     print("Pending orders:", ops.pending_orders())
252     processed = ops.process_next_order()
253     print("Processed order:", processed)
254     print("Pending orders:", ops.pending_orders())
255
256     # Top-selling products
257     tracker = TopSellingProductsTracker()
258     tracker.record_sale(1, 10) # Laptop sold 10
259     tracker.record_sale(2, 5) # Phone sold 5
260     tracker.record_sale(3, 7) # Headphones sold 7
261     print("Top 2 products (id, sales):", tracker.top_k(2))
262
263     # Delivery route planner
264     planner = DeliveryRoutePlanner()
265     planner.add_route("WarehouseA", "City1", 10.0)
266     planner.add_route("WarehouseA", "City2", 20.0)
267     planner.add_route("City1", "City2", 5.0)
268     planner.add_route("City2", "City3", 7.0)
269
270     dist, path = planner.shortest_path("WarehouseA", "City3")
271     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
Shortest route WarehouseA -> City3: ['WarehouseA', 'City1', 'City2', 'City3'] distance: 22.0

```