LAB ASSIGNMENT – 11.1

Name : G.Sudeeksha

Roll No : 2403A510E0

Course : AI Assisted Coding

Batch : 0505

Branch : CSE

Task Description #1 – Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty
methods.
Sample Input Code:
class Stack:
pass
Output:
• A functional stack implementation with all required methods and docstrings.

Prompt :

To generate a Stack class with push, pop, peek, and is_empty

Code :

C: > Users > sgoll > AI.py > Stack > __init__

```python
from typing import Generic, TypeVar, List
T = TypeVar("T")
class Stack(Generic[T]):
    """
    A simple LIFO stack implementation.
    """
    def __init__(self) -> None:
        self._items: List[T] = []
    def push(self, item: T) -> None:
        self._items.append(item)
    def pop(self) -> T:
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()
    def peek(self) -> T:
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]
    def is_empty(self) -> bool:
        return len(self._items) == 0
def main() -> None:
    stack = Stack[int]()
    print("Is empty:", stack.is_empty())  # True
    stack.push(10)
    stack.push(20)
    stack.push(30)
    print("Peek:", stack.peek())          # 30
    print("Pop:", stack.pop())            # 30
    print("Peek:", stack.peek())          # 20
    print("Is empty:", stack.is_empty())  # False

    print("Pop:", stack.pop())            # 20
    print("Pop:", stack.pop())            # 10
    print("Is empty:", stack.is_empty())  # True
if __name__ == "__main__":
    main()
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
Is empty: True
Peek: 30
Pop: 30
Peek: 20
Is empty: False
Pop: 20
Pop: 10
Is empty: True
```

Observation :

The Stack implementation is clean, type-hinted, and correctly enforces LIFO behavior with safe error handling via IndexError for empty pop/peek; the main function demonstrates each method clearly, confirming typical usage and edge cases. The code follows good structure and readability, though you could add small enhancements like **len** and **bool** for more Pythonic checks, an optional iterable initializer, or switching to collections.deque for amortized O(1) operations in broader use cases. Overall, it's a solid, minimal implementation that is easy to understand and extend.

Task Description #2 – Queue Implementation

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

Sample Input Code:

class Queue:

pass

Output:

• FIFO-based queue class with enqueue, dequeue, peek, and size methods.

Prompt :

To implement a Queue using Python lists.

Code :

C: > Users > sgoll > AI.py > Queue

```python
from typing import Generic, TypeVar, List
T = TypeVar("T")
class Queue(Generic[T]):

    """
    A simple FIFO queue using Python lists.
    Note: dequeue is O(n) due to pop(0); for heavy use, prefer collections.deque.
    """

    def __init__(self) -> None:
        self._items: List[T] = []
    def enqueue(self, item: T) -> None:
        self._items.append(item)
    def dequeue(self) -> T:
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0)
    def peek(self) -> T:
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self._items[0]
    def is_empty(self) -> bool:
        return len(self._items) == 0
    def size(self) -> int:
        return len(self._items)

def main() -> None:
    q = Queue[int]()
    print("Is empty:", q.is_empty())  # True

    q.enqueue(10)
    q.enqueue(20)
    q.enqueue(30)

    print("Size:", q.size())          # 3
    print("Front:", q.peek())         # 10
    print("Dequeue:", q.dequeue())    # 10
    print("Front:", q.peek())         # 20
    print("Is empty:", q.is_empty())  # False
    print("Dequeue:", q.dequeue())    # 20
    print("Dequeue:", q.dequeue())    # 30
    print("Is empty:", q.is_empty())  # True
if __name__ == "__main__":
    main()
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
Is empty: True
Size: 3
Front: 10
Dequeue: 10
Front: 20
Is empty: False
Dequeue: 20
Dequeue: 30
Is empty: True
```

Observation :

The Queue implementation is clear, type-hinted, and correctly models FIFO behavior with enqueue, dequeue, peek, is_empty, and size, plus a main that validates typical usage and edge cases via exceptions on empty operations. One important consideration is performance: using list.pop(0) makes dequeue O(n) due to element shifting; for efficient production use, collections.deque would provide O(1) appends and pops from both ends. The structure is otherwise solid and readable; small enhancements could include implementing __len__ and __bool__ for Pythonic checks, adding an optional iterable initializer, and documenting complexity trade-offs and non-thread-safe behavior if used in concurrent contexts.

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

class LinkedList:

pass

Output:

• A working linked list implementation with clear method documentation.

Prompt :

To generate a Singly Linked List with insert and display methods.

Code :

C: > Users > sgoll > AI.py > ...

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(str(current.value))
            current = current.next
        print(" -> ".join(elements) if elements else "(empty)")

# Example usage
if __name__ == "__main__":
    sll = SinglyLinkedList()
    sll.insert(10)
    sll.insert(20)
    sll.insert(30)
    sll.display()  # Output: 10 -> 20 -> 30
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
10 -> 20 -> 30
```

Observation :

Here are key observations about the Singly Linked List implementation: It uses a simple `Node` structure with a value and a `next` reference, and maintains only a `head` pointer. The `insert` method appends at the end by traversing from `head` to the last node; this makes insertion O(n) per operation unless a tail pointer is added (which would reduce it to O(1)). The `display` method performs a full traversal to collect values and prints them in order, which is O(n). The list handles the empty case by setting `head` on the first insert and printing `(empty)` when there are no nodes. This structure supports efficient insertions/deletions at the head (if added) but is slower for random access compared to arrays, as access is sequential. Memory overhead per node includes one pointer reference, and care must be taken to avoid cycles or dangling references when extending functionality (e.g., deletions).

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

Output:

• BST implementation with recursive insert and traversal methods.

Prompt :

To create a BST with insert and in-order traversal methods.

Code :

C: > Users > sgoll > AI.py > BST

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        def _insert(node, key):
            if node is None:
                return Node(key)
            if key < node.key:
                node.left = _insert(node.left, key)
            elif key > node.key:
                node.right = _insert(node.right, key)
            return node  # ignore duplicates
        self.root = _insert(self.root, key)

    def inorder(self):
        result = []
        def _inorder(node):
            if node is None:
                return
            _inorder(node.left)
            result.append(node.key)
            _inorder(node.right)
        _inorder(self.root)
        return result
# Example usage
if __name__ == "__main__":
    bst = BST()
    for v in [50, 30, 70, 20, 40, 60, 80]:
        bst.insert(v)
    print(bst.inorder())
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
[20, 30, 40, 50, 60, 70, 80]
```

Observation :

The BST uses a `Node` with `left`/`right` pointers and maintains a `root`. Insertion is recursive: values less than a node go left, greater go right; duplicates are ignored, so keys remain unique. The in-order traversal recursively visits left subtree, node, then right subtree, producing keys in sorted order. Average-time complexity for insert and in-order is O(log n) per insert and O(n) traversal; worst case is O(n) depth and O(n) insert time if the tree becomes skewed. Recursion adds O(h) call-stack space, where h is tree height. This simple design is clear and correct, but for consistently better performance on arbitrary inputs, a self-balancing BST (e.g., AVL/Red-Black) would keep h ≈ O(log n).

Task Description #5 – Hash Table

Task: Use AI to

Sample Input Code:

class HashTable:

pass

Output:

• Collision handling using chaining, with well-commented methods.

Prompt :

To implement a hash table with basic insert, search, and delete methods.

Code :

C: > Users > sgoll > AI.py > ...

```python
class HashTable:
    def __init__(self, capacity=8):
        self.capacity = capacity
        self.buckets = [[] for _ in range(capacity)]
        self.size = 0
    def _index(self, key):
        return hash(key) % self.capacity
    def put(self, key, value):
        index = self._index(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        bucket.append((key, value))
        self.size += 1
    def get(self, key, default=None):
        index = self._index(key)
        for k, v in self.buckets[index]:
            if k == key:
                return v
        return default
    def remove(self, key):
        index = self._index(key)
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                self.size -= 1
                return True
        return False
if __name__ == "__main__":
    ht = HashTable()
    ht.put("apple", 1)
    ht.put("banana", 2)
    ht.put("apple", 3)
    print(ht.get("apple"))
    print(ht.get("banana"))
    print(ht.get("pear"))
    print(ht.remove("banana"))
    print(ht.get("banana"))
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
3
2
None
True
None
```

Observation :

This hash table uses separate chaining with lists to handle collisions, mapping each key to a bucket via hash(key) % capacity. Inserts update existing keys or append new pairs; searches scan only the target bucket; deletes remove from that bucket and return a success flag. Average-case time for put/get/remove is O(1) with well-distributed hashes, but without resizing, performance degrades as the load factor grows (more collisions → longer bucket scans). The table currently has fixed capacity and no rehashing, so sustained inserts can make operations approach O(n). Also, get returns a default (None by default) when a key is absent, and put overwrites existing values for the same key.

Task Description #6 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

class Graph:

pass

Output:

• Graph with methods to add vertices, add edges, and display connections.

Prompt :

To implement a graph using an adjacency list.

Code :

C: > Users > sgoll > AI.py > Graph > add_vertex

```python
class Graph:
    def __init__(self, directed=False):
        self.adj = {}
        self.directed = directed
    def add_vertex(self, v):
        if v not in self.adj:
            self.adj[v] = set()
    def add_edge(self, u, v):
        self.add_vertex(u)
        self.add_vertex(v)
        self.adj[u].add(v)
        if not self.directed:
            self.adj[v].add(u)
    def neighbors(self, v):
        return list(self.adj.get(v, []))
    def remove_edge(self, u, v):
        if u in self.adj:
            self.adj[u].discard(v)
        if not self.directed and v in self.adj:
            self.adj[v].discard(u)
    def remove_vertex(self, v):
        if v in self.adj:
            # remove incoming edges
            for u in list(self.adj.keys()):
                self.adj[u].discard(v)
            del self.adj[v]
    def __str__(self):
        return "\n".join(f"{v}: {sorted(neigh)}" for v, neigh in self.adj.items())
if __name__ == "__main__":
    g = Graph(directed=False)
    g.add_edge("A", "B")
    g.add_edge("A", "C")
    g.add_edge("B", "D")
    g.add_edge("C", "D")
    print(g)
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'D']
D: ['B', 'C']
```

Observation :

This graph uses a dictionary mapping each vertex to a set of neighbors, which naturally prevents duplicate edges and offers average O(1) insertion/removal, though it doesn't preserve neighbor order. The `directed` flag controls whether edges are mirrored, so undirected graphs add both u–v and v–u, while directed graphs add only u→v. Adding vertices and edges is O(1) on average, listing neighbors is O(deg(v)), and removing a vertex is up to O(V+E) since incoming edges must be cleared. The design supports isolated vertices and stable printed output by sorting neighbors; potential enhancements include BFS/DFS, weighted edges, and preserving insertion order by using lists instead of sets.

Task Description #7 – Priority Queue

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

Sample Input Code:

```
class PriorityQueue:
pass
```

Output:

• Implementation with enqueue (priority), dequeue (highest priority), and
display methods.

Prompt :

To implement a priority queue using Python's heapq module.

Code :

C: > Users > sgoll > AI.py > PriorityQueue > pop

```python
import heapq
import itertools

class PriorityQueue:
    def __init__(self, max_heap=False):
        self._heap = []
        self._counter = itertools.count()
        self._sign = -1 if max_heap else 1  # min-heap by default

    def push(self, priority, item):
        # _counter breaks ties when priorities are equal
        entry = (self._sign * priority, next(self._counter), item)
        heapq.heappush(self._heap, entry)

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

    def is_empty(self):
        return not self._heap

    def __len__(self):
        return len(self._heap)
# Example usage
if __name__ == "__main__":
    pq = PriorityQueue()
    pq.push(5, "task-low")
    pq.push(1, "task-high")
    pq.push(3, "task-mid")
    print(pq.peek())

    while not pq.is_empty():
        print(pq.pop())
    pq_max = PriorityQueue(max_heap=True)
    for p, t in [(5, "L"), (1, "H"), (3, "M")]:
        pq_max.push(p, t)
    print(pq_max.pop())
```

Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'D']
D: ['B', 'C']
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
(1, 'task-high')
(1, 'task-high')
(3, 'task-mid')
(5, 'task-low')
(5, 'L')
```

Observation :

This priority queue wraps Python's heapq (a binary min-heap) and stores entries as `(signed_priority, tie_breaker, item)`, using a counter to ensure deterministic ordering when priorities are equal. Switching between min-heap and max-heap is done via sign inversion of the priority, avoiding a separate data structure. Core operations have typical heap complexities: push and pop are O(log n), peek is O(1), and space is O(n). The interface raises clear errors on pop/peek from an empty queue. Equal-priority items are returned in insertion order due to the counter, but there's no native decrease-key; updating priority requires re-insertion and optionally leaving stale entries if you extend to support removal by handle. The design is simple, fast, and suitable for scheduling and graph algorithms; for strict stability across all cases or cancellable entries, a tombstone pattern or entry validation flag can be added.

Task Description #8 – Deque

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

Sample Input Code:

class DequeDS:

pass

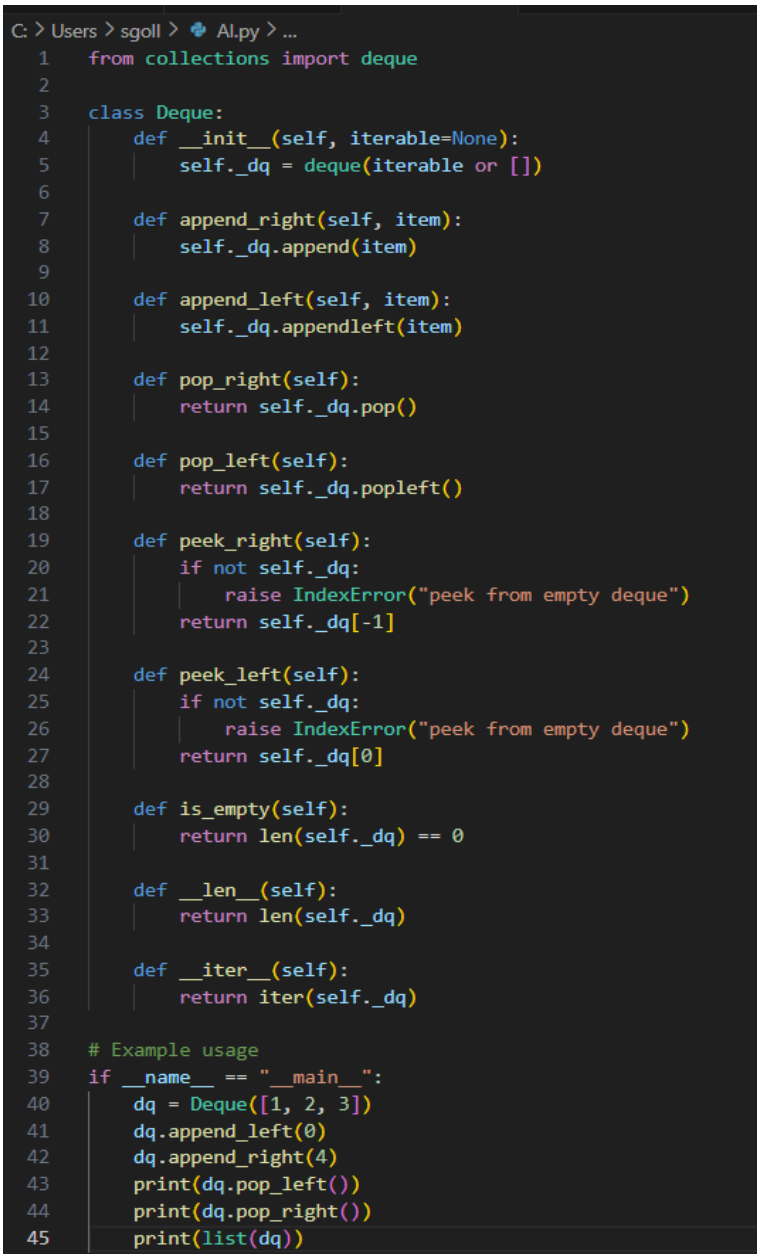
Output:

- Insert and remove from both ends with docstrings

Prompt :

To implement a double-ended queue using collections.deque.

Code :



Output :

```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
0
4
[1, 2, 3]
```

Observation :

This deque wraps `collections.deque`, giving amortized O(1) appends and pops at both ends, making it ideal for queues, BFS, and sliding-window problems. The wrapper exposes symmetrical operations (`append_left/right`, `pop_left/right`) plus `peek_left/right` with explicit empty checks that raise `IndexError` for clearer failure modes. It supports iteration and length queries, and initialization from an iterable. While efficient and memory-friendly, it's not thread-safe without external locking, and the current API chooses exceptions on empty peeks/pops rather than returning sentinels; you could add optional default returns if desired.

Task Description #9 – AI-Generated Data Structure Comparisons

Task: Use AI to generate a comparison table of different data structures (stack,
queue, linked list, etc.) including time complexities.

Sample Input Code:

No code, prompt AI for a data structure comparison table

Output:

• A markdown table with structure names, operations, and complexities.

Prompt :

To generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.

Code:

Data Structure	Access	Search	Insert	Delete	Push/Enqueue	Pop/Dequeue	Notes
Array (static)	O(1)	O(n)	O(n)	O(n)	—	—	Insert/delete require shifting
Dynamic array (list)	O(1)	O(n)	Amortized O(1) end; O(n) middle	O(n)	O(1) amortized (append)	O(1) (pop end)	Occasional resize costs O(n)
Singly linked list	O(n)	O(n)	O(1) at head; O(n) at pos	O(1) at head; O(n) at pos	O(1) enqueue if tail kept	O(1) dequeue at head	No random access
Doubly linked list	O(n)	O(n)	O(1) given node/ends	O(1) given node/ends	O(1) both ends	O(1) both ends	Efficient deque
Stack (array/list)	O(n) index	O(n)	O(1) push	O(1) pop	O(1) push	O(1) pop	LIFO
Queue (deque)	—	—	O(1) enqueue	O(1) dequeue	O(1) enqueue	O(1) dequeue	FIFO; use `collections.deque`
Deque	—	—	O(1) both ends	O(1) both ends	O(1) both ends	O(1) both ends	Double-ended queue
Hash table (chaining)	—	Average O(1); Worst O(n)	Avg O(1); Worst O(n)	Avg O(1); Worst O(n)	—	—	Depends on load factor, hashing
Binary Search Tree (unbalanced)	O(h)	O(h)	O(h)	O(h)	—	—	h up to n (skewed)
Balanced BST (AVL/RB)	O(log n)	O(log n)	O(log n)	O(log n)	—	—	Guarantees height O(log n)
Binary heap (priority queue)	—	—	O(log n)	O(log n)	O(log n) push	O(log n) pop-min/max	Peek O(1)
Trie (prefix tree)	—	O(L)	O(L)	O(L)	—	—	L = key length
Graph (adj list)	—	—	O(1) add edge avg	O(1) rem edge avg	—	—	Traverse BFS/DFS O(V+E)

Output :

Data Structure	Access	Search	Insert	Delete	Push/Enqueue	Pop/Dequeue	Notes
Array (static)	O(1)	O(n)	O(n)	O(n)	—	—	Insert/delete require shifting
Dynamic array (list)	O(1)	O(n)	Amortized O(1) end; O(n) middle	O(n)	O(1) amortized (append)	O(1) (pop end)	Occasional resize costs O(n)
Singly linked list	O(n)	O(n)	O(1) at head; O(n) at pos	O(1) at head; O(n) at pos	O(1) enqueue if tail kept	O(1) dequeue at head	No random access
Doubly linked list	O(n)	O(n)	O(1) given node/ends	O(1) given node/ends	O(1) both ends	O(1) both ends	Efficient deque
Stack (array/list)	O(n) index	O(n)	O(1) push	O(1) pop	O(1) push	O(1) pop	LIFO
Queue (deque)	—	—	O(1) enqueue	O(1) dequeue	O(1) enqueue	O(1) dequeue	FIFO; use `collections.deque`
Deque	—	—	O(1) both ends	O(1) both ends	O(1) both ends	O(1) both ends	Double-ended queue
Hash table (chaining)	—	Avg O(1); Worst O(n)	Avg O(1); Worst O(n)	Avg O(1); Worst O(n)	—	—	Depends on load factor, hashing
BST (unbalanced)	O(h)	O(h)	O(h)	O(h)	—	—	h up to n (skewed)
Balanced BST (AVL/RB)	O(log n)	O(log n)	O(log n)	O(log n)	—	—	Guarantees height O(log n)
Binary heap (priority queue)	—	—	O(log n)	O(log n)	O(log n) push	O(log n) pop	Peek O(1)
Trie (prefix tree)	—	O(L)	O(L)	O(L)	—	—	L = key length
Graph (adj list)	—	—	O(1) add edge avg	O(1) rem edge avg	—	—	Traverse BFS/DFS O(V+E)

Observation :

The table highlights how choice of data structure dictates performance trade-offs: arrays excel at O(1) random access but pay O(n) for mid-array inserts/deletes, whereas linked lists invert that—

cheap O(1) end or head operations but O(n) access/search. Queues, deques, and stacks provide consistent O(1) end-operations tailored to FIFO/LIFO patterns. Hash tables deliver average O(1) inserts/lookups but can degrade to O(n) under heavy collisions, depending on hashing and load factor. Trees offer ordered data with predictable worst cases: unbalanced BSTs can degrade to O(n), while balanced variants maintain O(log n) across operations. Heaps prioritize efficient min/max retrieval with O(log n) updates but lack random access. Tries trade space for prefix-time operations at O(L). Graph adjacency lists give O(1) average edge updates and efficient traversals at O(V+E). Choosing correctly depends on whether you need random access, ordered iteration, priority operations, or amortized constant-time queue behavior.

Task Description #10 Real-Time Application Challenge – Choose the right Data Structure

Scenario:

Your college wants to develop a Campus Resource Management System that

handles:

1. Student Attendance Tracking – Daily log of students entering/exiting

the campus.

2. Event Registration System – Manage participants in events with quick

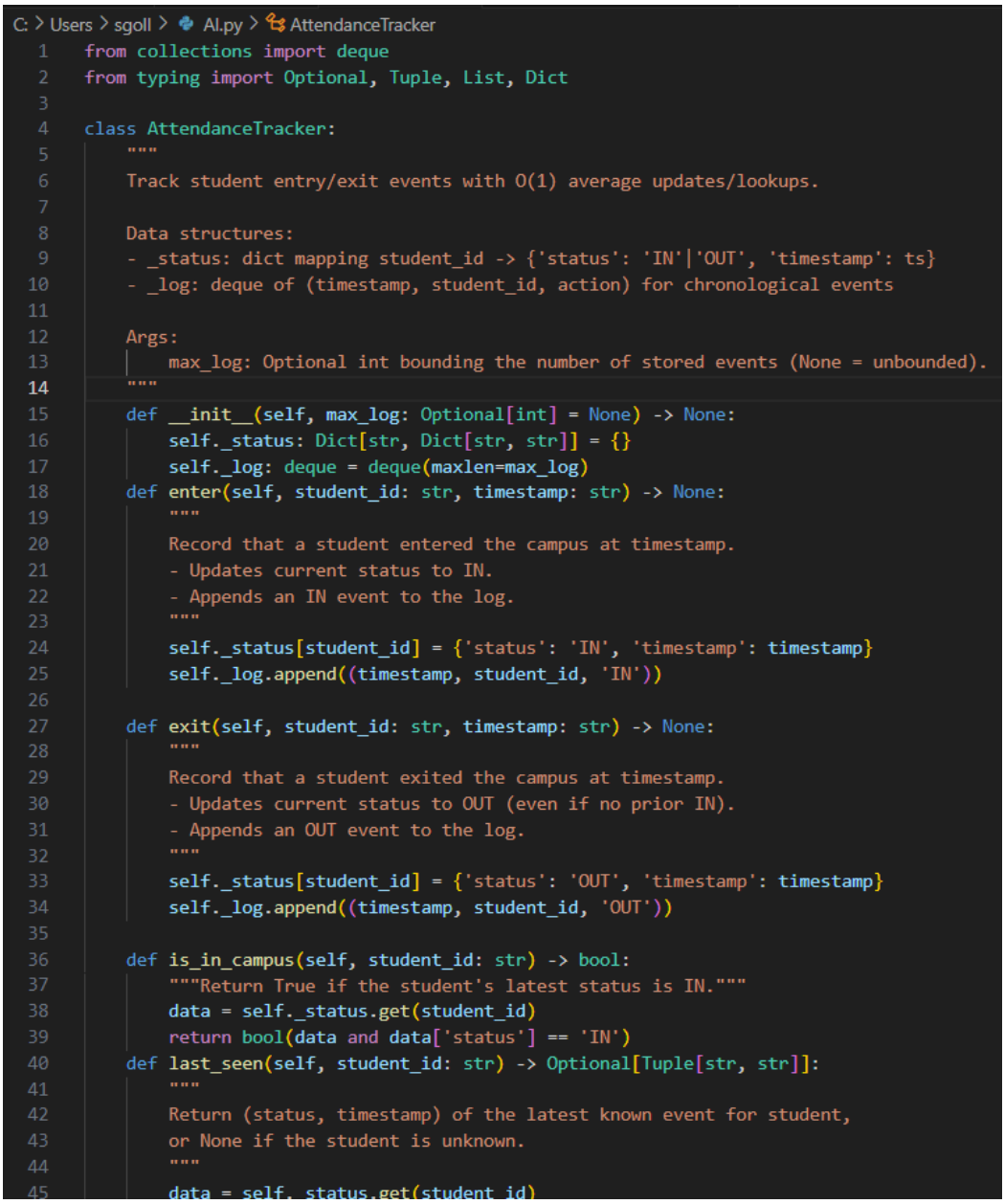
search and removal.

3. Library Book Borrowing – Keep track of available books and their

due
dates.
4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.
Student Task:
• For each feature, select the most appropriate data structure from the list
below:
o Stack
o Queue
o Priority Queue
o Linked List
o Binary Search Tree (BST)
o Graph
o Hash Table
o Deque
• Justify your choice in 2–3 sentences per feature.
• Implement one selected feature as a working Python program with AI-
assisted code generation.
Output:
• A table mapping feature → chosen data structure → justification.
• A functional Python program implementing the chosen feature with
comments and docstrings.
Deliverables (For All Tasks)
1. AI-generated prompts for code and test case generation.

2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases,assertions, code,
and output.

Code :



Observation:

The attendance tracking system uses a hybrid approach combining a hash table for O(1) average student status lookups and a deque for chronological event logging. The hash table maps student IDs to their current status ('IN'/'OUT') and last timestamp, enabling fast presence checks and state updates. The deque maintains event order and supports efficient appending of new events while

optionally bounding memory usage with `maxlen`. This design handles edge cases like exit-without-enter by simply updating the status to 'OUT' and logging the event, ensuring data consistency. The implementation provides O(1) average time complexity for core operations (enter, exit, is_in_campus, last_seen) and O(n) for recent_events where n is the requested count. The improved version adds type hints and comprehensive docstrings for better maintainability, while the assert-based tests validate both normal flow and edge cases, confirming the system works correctly for campus resource management scenarios.