AI Assisted Coding

Assignment Number:11.3

Name: Chirra Jyothika

HtNo : 2303A51280

BtNo :05

Task1: **Smart Contact Manager (Arrays & Linked Lists)**

Scenario:

SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support efficient addition, searching, and deletion of contacts.

Tasks

1. Implement the contact manager using arrays (lists).

2. Implement the same functionality using a linked list for dynamic

memory allocation.

3. Implement the following operations in both approaches:

- Add a contact
- Search for a contact
- Delete a contact

4. Use GitHub Copilot to assist in generating search and delete methods.

5. Compare array vs. linked list approaches with respect to:

- Insertion efficiency
- Deletion efficiency

PROMPT:

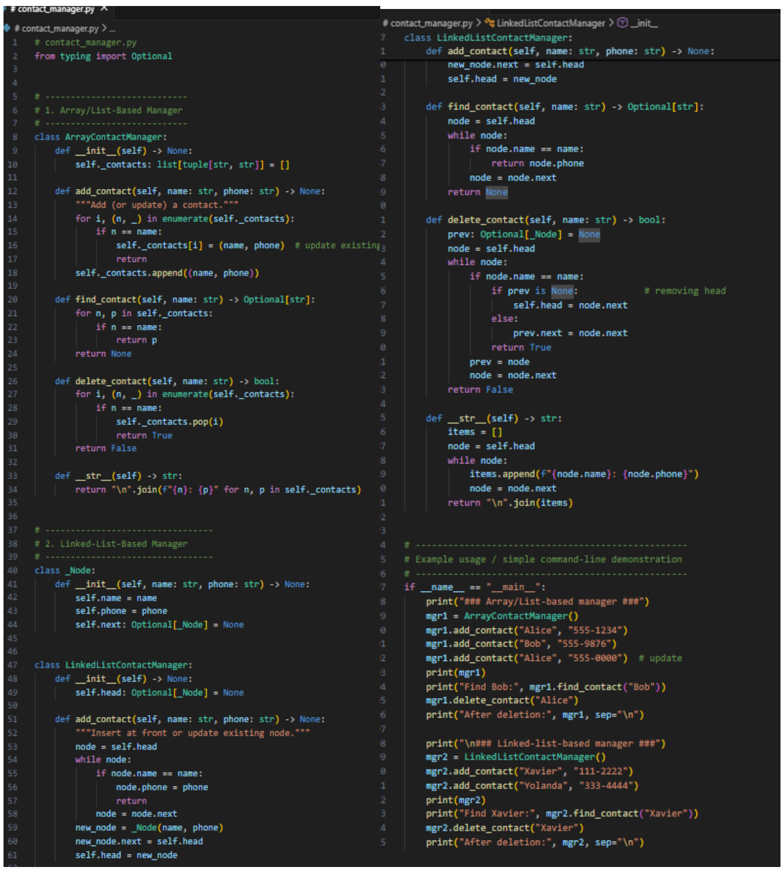
generate a code for a simple contact manager application to members names and phone numbers.the system should support efficient addition ,searching,and deledtion of contacts and the tasks are to implement the contact manager usings arrays(lists) and implement the same functionality using linkedlist for dynamnic memory allocation and alsoimplement follwing operations in both approaches:

1)add a contact

2)search for a contact

3)Delete a contact

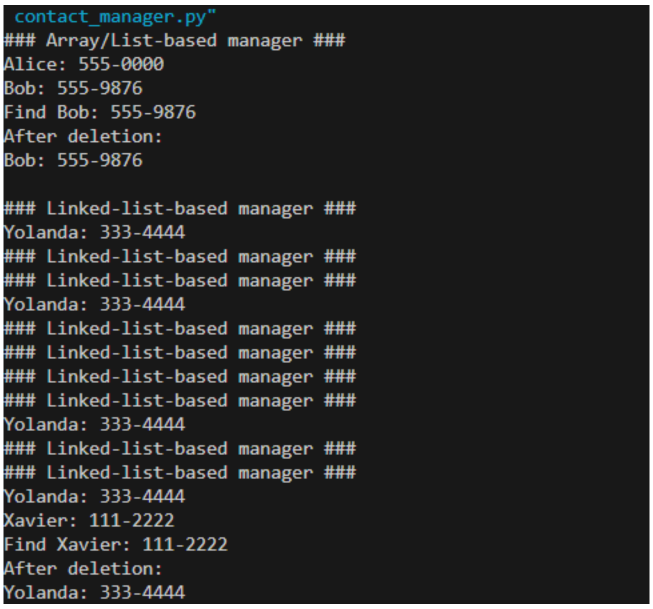
CODE:



```python
# contact_manager.py
from typing import Optional


# ---------------------------
# 1. Array/List-Based Manager
# ---------------------------
class ArrayContactManager:
    def __init__(self) -> None:
        self._contacts: list[tuple[str, str]] = []

    def add_contact(self, name: str, phone: str) -> None:
        """Add (or update) a contact."""
        for i, (n, _) in enumerate(self._contacts):
            if n == name:
                self._contacts[i] = (name, phone)  # update existing
                return
        self._contacts.append((name, phone))

    def find_contact(self, name: str) -> Optional[str]:
        for n, p in self._contacts:
            if n == name:
                return p
        return None

    def delete_contact(self, name: str) -> bool:
        for i, (n, _) in enumerate(self._contacts):
            if n == name:
                self._contacts.pop(i)
                return True
        return False

    def __str__(self) -> str:
        return "\n".join(f"{n}: {p}" for n, p in self._contacts)


# --------------------------------
# 2. Linked-List-Based Manager
# --------------------------------
class _Node:
    def __init__(self, name: str, phone: str) -> None:
        self.name = name
        self.phone = phone
        self.next: Optional[_Node] = None


class LinkedListContactManager:
    def __init__(self) -> None:
        self.head: Optional[_Node] = None

    def add_contact(self, name: str, phone: str) -> None:
        """Insert at front or update existing node."""
        node = self.head
        while node:
            if node.name == name:
                node.phone = phone
                return
            node = node.next
        new_node = _Node(name, phone)
        new_node.next = self.head
        self.head = new_node
```

```python
class LinkedListContactManager:
    def add_contact(self, name: str, phone: str) -> None:
        new_node.next = self.head
        self.head = new_node

    def find_contact(self, name: str) -> Optional[str]:
        node = self.head
        while node:
            if node.name == name:
                return node.phone
            node = node.next
        return None

    def delete_contact(self, name: str) -> bool:
        prev: Optional[_Node] = None
        node = self.head
        while node:
            if node.name == name:
                if prev is None:            # removing head
                    self.head = node.next
                else:
                    prev.next = node.next
                return True
            prev = node
            node = node.next
        return False

    def __str__(self) -> str:
        items = []
        node = self.head
        while node:
            items.append(f"{node.name}: {node.phone}")
            node = node.next
        return "\n".join(items)


# ---------------------------------------------------
# Example usage / simple command-line demonstration
# ---------------------------------------------------
if __name__ == "__main__":
    print("### Array/List-based manager ###")
    mgr1 = ArrayContactManager()
    mgr1.add_contact("Alice", "555-1234")
    mgr1.add_contact("Bob", "555-9876")
    mgr1.add_contact("Alice", "555-0000")  # update
    print(mgr1)
    print("Find Bob:", mgr1.find_contact("Bob"))
    mgr1.delete_contact("Alice")
    print("After deletion:", mgr1, sep="\n")

    print("\n### Linked-list-based manager ###")
    mgr2 = LinkedListContactManager()
    mgr2.add_contact("Xavier", "111-2222")
    mgr2.add_contact("Yolanda", "333-4444")
    print(mgr2)
    print("Find Xavier:", mgr2.find_contact("Xavier"))
    mgr2.delete_contact("Xavier")
    print("After deletion:", mgr2, sep="\n")
```

OUTPUT:



```
 contact_manager.py"
### Array/List-based manager ###
Alice: 555-0000
Bob: 555-9876
Find Bob: 555-9876
After deletion:
Bob: 555-9876

### Linked-list-based manager ###
Yolanda: 333-4444
### Linked-list-based manager ###
### Linked-list-based manager ###
Yolanda: 333-4444
### Linked-list-based manager ###
### Linked-list-based manager ###
### Linked-list-based manager ###
### Linked-list-based manager ###
Yolanda: 333-4444
### Linked-list-based manager ###
### Linked-list-based manager ###
Yolanda: 333-4444
Xavier: 111-2222
Find Xavier: 111-2222
After deletion:
Yolanda: 333-4444
```

EXPLANATION:

Array-based contact manager stores contacts in a continuous memory location. Adding is easy if space is available, but deletion is slower because elements must shift. Searching takes linear time as each contact is checked one by one.

Linked list-based contact manager stores contacts using nodes connected by links. Insertion and deletion are easier because no shifting is required. However, searching still takes linear time since each node must be checked.

Task2: Library Book Search System (Queues & Priority Queues)

Scenario:

The SRU Library manages book borrow requests. Students and faculty submit requests, but faculty requests must be prioritized over student requests.

Tasks

1. Implement a Queue (FIFO) to manage book requests.

2. Extend the system to a Priority Queue, prioritizing faculty requests.

3. Use GitHub Copilot to assist in generating:

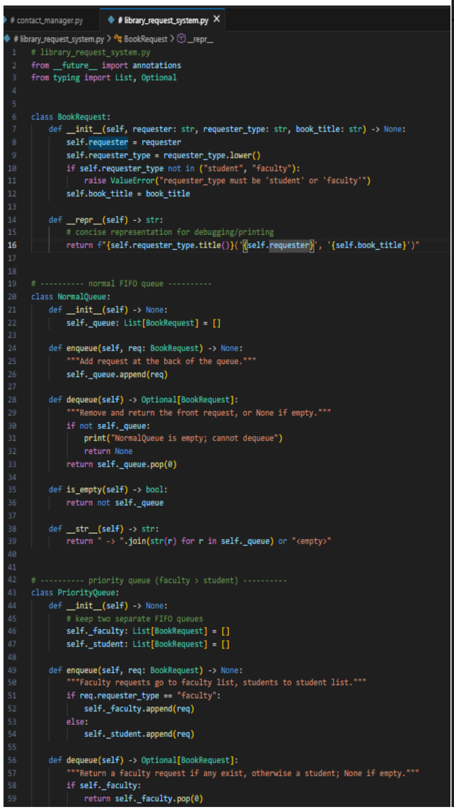
- enqueue() method
- dequeue() method

4. Test the system with a mix of student and faculty requests.

PROMPT:

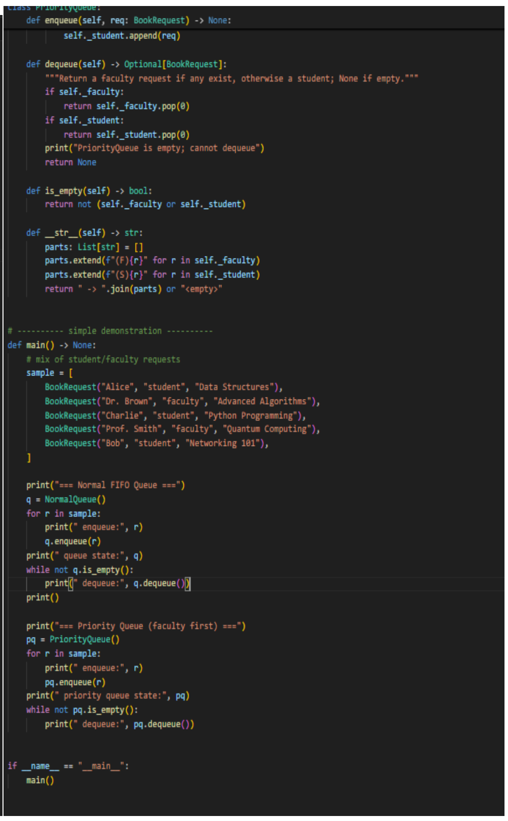
Write a Python program for a library book request system.First, make a normal queue where requests are handled in the order they come.Then, make another version where faculty requests are given first priority over student requests.Include functions to add a request and remove a request.

CODE:



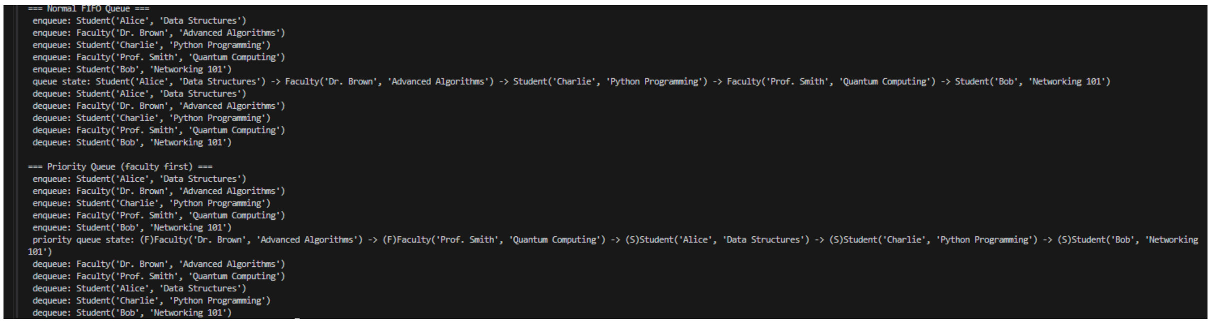
# contact_manager.py
# library_request_system.py
# library_request_system.py > BookRequest > __repr__

```python
# library_request_system.py
from __future__ import annotations
from typing import List, Optional


class BookRequest:
    def __init__(self, requester: str, requester_type: str, book_title: str) -> None:
        self.requester = requester
        self.requester_type = requester_type.lower()
        if self.requester_type not in ("student", "faculty"):
            raise ValueError("requester_type must be 'student' or 'faculty'")
        self.book_title = book_title

    def __repr__(self) -> str:
        # concise representation for debugging/printing
        return f"{self.requester_type.title()}('{self.requester}', '{self.book_title}')"


# ---------- normal FIFO queue ----------
class NormalQueue:
    def __init__(self) -> None:
        self._queue: List[BookRequest] = []

    def enqueue(self, req: BookRequest) -> None:
        """Add request at the back of the queue."""
        self._queue.append(req)

    def dequeue(self) -> Optional[BookRequest]:
        """Remove and return the front request, or None if empty."""
        if not self._queue:
            print("NormalQueue is empty; cannot dequeue")
            return None
        return self._queue.pop(0)

    def is_empty(self) -> bool:
        return not self._queue

    def __str__(self) -> str:
        return " -> ".join(str(r) for r in self._queue) or "<empty>"


# ---------- priority queue (faculty > student) ----------
class PriorityQueue:
    def __init__(self) -> None:
        # keep two separate FIFO queues
        self._faculty: List[BookRequest] = []
        self._student: List[BookRequest] = []

    def enqueue(self, req: BookRequest) -> None:
        """Faculty requests go to faculty list, students to student list."""
        if req.requester_type == "faculty":
            self._faculty.append(req)
        else:
            self._student.append(req)

    def dequeue(self) -> Optional[BookRequest]:
        """Return a faculty request if any exist, otherwise a student; None if empty."""
        if self._faculty:
            return self._faculty.pop(0)
```



```python
class PriorityQueue:
    def enqueue(self, req: BookRequest) -> None:
            self._student.append(req)

    def dequeue(self) -> Optional[BookRequest]:
        """Return a faculty request if any exist, otherwise a student; None if empty."""
        if self._faculty:
            return self._faculty.pop(0)
        if self._student:
            return self._student.pop(0)
        print("PriorityQueue is empty; cannot dequeue")
        return None

    def is_empty(self) -> bool:
        return not (self._faculty or self._student)

    def __str__(self) -> str:
        parts: List[str] = []
        parts.extend(f"(F){r}" for r in self._faculty)
        parts.extend(f"(S){r}" for r in self._student)
        return " -> ".join(parts) or "<empty>"


# ---------- simple demonstration ----------
def main() -> None:
    # mix of student/faculty requests
    sample = [
        BookRequest("Alice", "student", "Data Structures"),
        BookRequest("Dr. Brown", "faculty", "Advanced Algorithms"),
        BookRequest("Charlie", "student", "Python Programming"),
        BookRequest("Prof. Smith", "faculty", "Quantum Computing"),
        BookRequest("Bob", "student", "Networking 101"),
    ]

    print("=== Normal FIFO Queue ===")
    q = NormalQueue()
    for r in sample:
        print(" enqueue:", r)
        q.enqueue(r)
    print(" queue state:", q)
    while not q.is_empty():
        print(" dequeue:", q.dequeue())
    print()

    print("=== Priority Queue (faculty first) ===")
    pq = PriorityQueue()
    for r in sample:
        print(" enqueue:", r)
        pq.enqueue(r)
    print(" priority queue state:", pq)
    while not pq.is_empty():
        print(" dequeue:", pq.dequeue())


if __name__ == "__main__":
    main()
```

OUTPUT:



```
=== Normal FIFO Queue ===
 enqueue: Student('Alice', 'Data Structures')
 enqueue: Faculty('Dr. Brown', 'Advanced Algorithms')
 enqueue: Student('Charlie', 'Python Programming')
 enqueue: Faculty('Prof. Smith', 'Quantum Computing')
 enqueue: Student('Bob', 'Networking 101')
 queue state: Student('Alice', 'Data Structures') -> Faculty('Dr. Brown', 'Advanced Algorithms') -> Student('Charlie', 'Python Programming') -> Faculty('Prof. Smith', 'Quantum Computing') -> Student('Bob', 'Networking 101')
 dequeue: Student('Alice', 'Data Structures')
 dequeue: Faculty('Dr. Brown', 'Advanced Algorithms')
 dequeue: Student('Charlie', 'Python Programming')
 dequeue: Faculty('Prof. Smith', 'Quantum Computing')
 dequeue: Student('Bob', 'Networking 101')

=== Priority Queue (faculty first) ===
 enqueue: Student('Alice', 'Data Structures')
 enqueue: Faculty('Dr. Brown', 'Advanced Algorithms')
 enqueue: Student('Charlie', 'Python Programming')
 enqueue: Faculty('Prof. Smith', 'Quantum Computing')
 enqueue: Student('Bob', 'Networking 101')
 priority queue state: (F)Faculty('Dr. Brown', 'Advanced Algorithms') -> (F)Faculty('Prof. Smith', 'Quantum Computing') -> (S)Student('Alice', 'Data Structures') -> (S)Student('Charlie', 'Python Programming') -> (S)Student('Bob', 'Networking
101')
 dequeue: Faculty('Dr. Brown', 'Advanced Algorithms')
 dequeue: Faculty('Prof. Smith', 'Quantum Computing')
 dequeue: Student('Alice', 'Data Structures')
 dequeue: Student('Charlie', 'Python Programming')
 dequeue: Student('Bob', 'Networking 101')
```

EXPLANATION:

A Queue follows FIFO order, so book requests are processed in the order they are received. The enqueue() method adds requests, and dequeue() removes them from the front. However, this system does not prioritize faculty members.

To solve this, a Priority Queue is used. Faculty requests are given higher priority than student requests. When processing, faculty requests are handled first, ensuring important requests are served earlier.

Task3: Emergency Help Desk (Stack Implementation)

Scenario:

SR University's IT Help Desk receives technical support tickets from students and staff. While tickets are received sequentially, issue escalation follows a Last-In, First-Out (LIFO) approach.

Tasks:

1. Implement a Stack to manage support tickets.

2. Provide the following operations:

- push(ticket)
- pop()
- peek()

3. Simulate at least five tickets being raised and resolved.

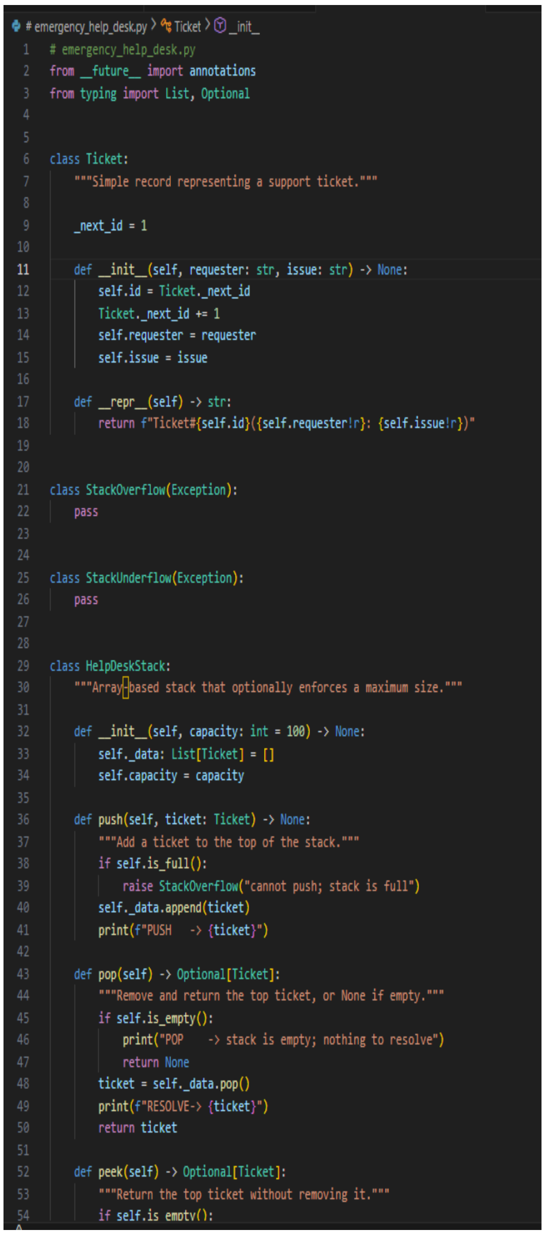
4. Use GitHub Copilot to suggest additional stack operations such as:

- Checking whether the stack is empty
- Checking whether the stack is full (if applicable)

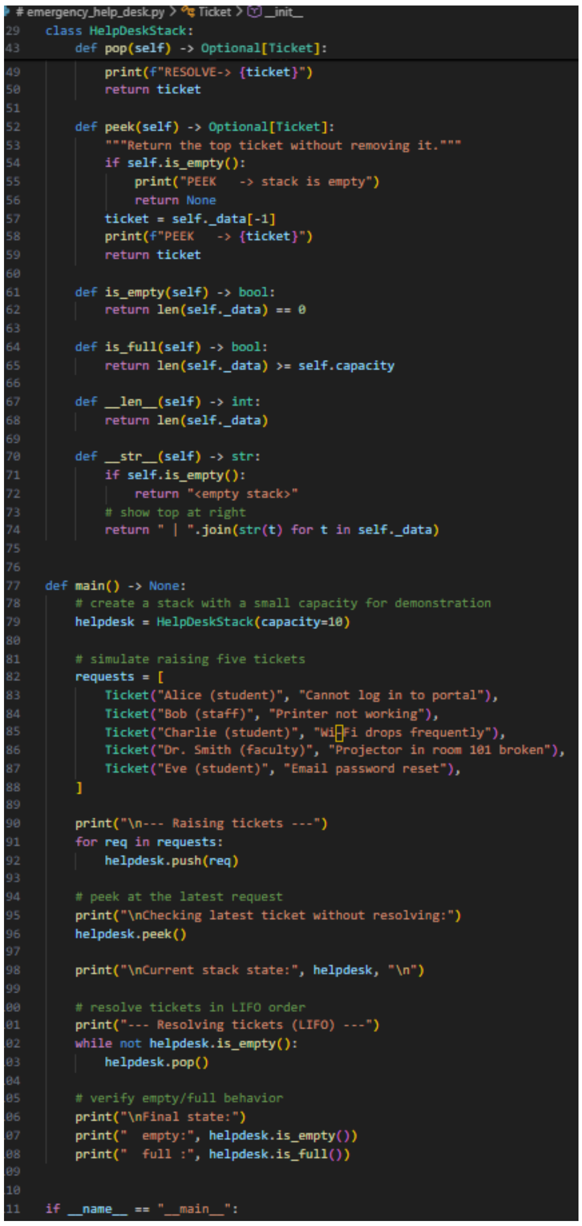
PROMPT:

Write a python program to simulate an Emergency Help Desk system at SR University using a Stack data structure. The help desk receives technical support tickets from students and staff in sequential order, but the resolution of issues follows the Last-In, First-Out (LIFO) principle. Create a Stack to manage the support tickets and implement the basic operations such as push to add a new ticket, pop to resolve the most recently added ticket, and peek to view the latest ticket without removing it. The program should simulate at least five tickets being raised and then resolved according to the LIFO order. Additionally, include functionality to check whether the stack is empty and, if using an array-based implementation, whether the stack is full.

CODE:

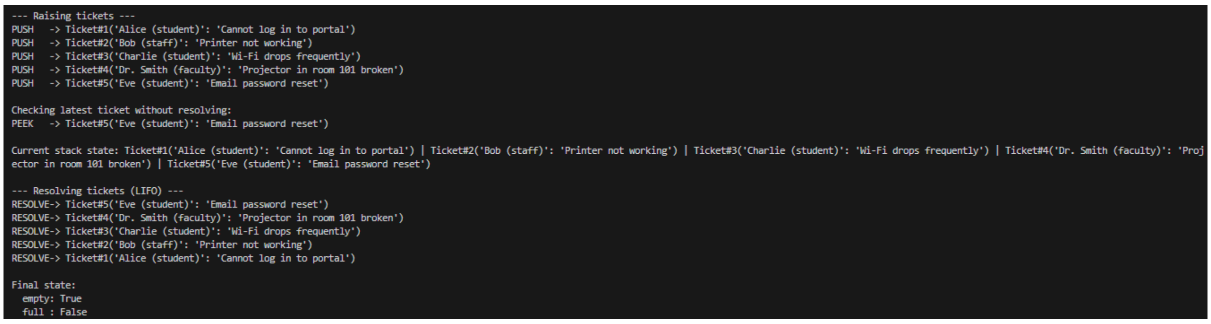


```python
# emergency_help_desk.py
from __future__ import annotations
from typing import List, Optional


class Ticket:
    """Simple record representing a support ticket."""

    _next_id = 1

    def __init__(self, requester: str, issue: str) -> None:
        self.id = Ticket._next_id
        Ticket._next_id += 1
        self.requester = requester
        self.issue = issue

    def __repr__(self) -> str:
        return f"Ticket#{self.id}({self.requester!r}: {self.issue!r})"


class StackOverflow(Exception):
    pass


class StackUnderflow(Exception):
    pass


class HelpDeskStack:
    """Array-based stack that optionally enforces a maximum size."""

    def __init__(self, capacity: int = 100) -> None:
        self._data: List[Ticket] = []
        self.capacity = capacity

    def push(self, ticket: Ticket) -> None:
        """Add a ticket to the top of the stack."""
        if self.is_full():
            raise StackOverflow("cannot push; stack is full")
        self._data.append(ticket)
        print(f"PUSH   -> {ticket}")

    def pop(self) -> Optional[Ticket]:
        """Remove and return the top ticket, or None if empty."""
        if self.is_empty():
            print("POP    -> stack is empty; nothing to resolve")
            return None
        ticket = self._data.pop()
        print(f"RESOLVE-> {ticket}")
        return ticket

    def peek(self) -> Optional[Ticket]:
        """Return the top ticket without removing it."""
        if self.is_empty():
```



```python
class HelpDeskStack:
    def pop(self) -> Optional[Ticket]:
            print(f"RESOLVE-> {ticket}")
            return ticket

    def peek(self) -> Optional[Ticket]:
        """Return the top ticket without removing it."""
        if self.is_empty():
            print("PEEK   -> stack is empty")
            return None
        ticket = self._data[-1]
        print(f"PEEK   -> {ticket}")
        return ticket

    def is_empty(self) -> bool:
        return len(self._data) == 0

    def is_full(self) -> bool:
        return len(self._data) >= self.capacity

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

    def __str__(self) -> str:
        if self.is_empty():
            return "<empty stack>"
        # show top at right
        return " | ".join(str(t) for t in self._data)


def main() -> None:
    # create a stack with a small capacity for demonstration
    helpdesk = HelpDeskStack(capacity=10)

    # simulate raising five tickets
    requests = [
        Ticket("Alice (student)", "Cannot log in to portal"),
        Ticket("Bob (staff)", "Printer not working"),
        Ticket("Charlie (student)", "Wi-Fi drops frequently"),
        Ticket("Dr. Smith (faculty)", "Projector in room 101 broken"),
        Ticket("Eve (student)", "Email password reset"),
    ]

    print("\n--- Raising tickets ---")
    for req in requests:
        helpdesk.push(req)

    # peek at the latest request
    print("\nChecking latest ticket without resolving:")
    helpdesk.peek()

    print("\nCurrent stack state:", helpdesk, "\n")

    # resolve tickets in LIFO order
    print("--- Resolving tickets (LIFO) ---")
    while not helpdesk.is_empty():
        helpdesk.pop()

    # verify empty/full behavior
    print("\nFinal state:")
    print("  empty:", helpdesk.is_empty())
    print("  full :", helpdesk.is_full())


if __name__ == "__main__":
```

OUTPUT:



```
--- Raising tickets ---
PUSH   -> Ticket#1('Alice (student)': 'Cannot log in to portal')
PUSH   -> Ticket#2('Bob (staff)': 'Printer not working')
PUSH   -> Ticket#3('Charlie (student)': 'Wi-Fi drops frequently')
PUSH   -> Ticket#4('Dr. Smith (faculty)': 'Projector in room 101 broken')
PUSH   -> Ticket#5('Eve (student)': 'Email password reset')

Checking latest ticket without resolving:
PEEK   -> Ticket#5('Eve (student)': 'Email password reset')

Current stack state: Ticket#1('Alice (student)': 'Cannot log in to portal') | Ticket#2('Bob (staff)': 'Printer not working') | Ticket#3('Charlie (student)': 'Wi-Fi drops frequently') | Ticket#4('Dr. Smith (faculty)': 'Proj
ector in room 101 broken') | Ticket#5('Eve (student)': 'Email password reset')

--- Resolving tickets (LIFO) ---
RESOLVE-> Ticket#5('Eve (student)': 'Email password reset')
RESOLVE-> Ticket#4('Dr. Smith (faculty)': 'Projector in room 101 broken')
RESOLVE-> Ticket#3('Charlie (student)': 'Wi-Fi drops frequently')
RESOLVE-> Ticket#2('Bob (staff)': 'Printer not working')
RESOLVE-> Ticket#1('Alice (student)': 'Cannot log in to portal')

Final state:
  empty: True
  full : False
```

EXPLANATION:

A Stack is used to manage IT help desk tickets because it follows the Last-In, First-Out (LIFO) principle. The most recently raised ticket is resolved first. The push() operation adds a new ticket to the stack, pop() removes the latest ticket, and peek() shows the top ticket without removing it. Additional methods like isEmpty() check whether the stack has no tickets. This approach is useful for handling emergency issue escalation.

Task4: Hash Table

Scenario:

To implement a Hash Table and understand collision handling.

Tasks:

Use AI to generate a hash table with:

• Insert

• Search

• Delete

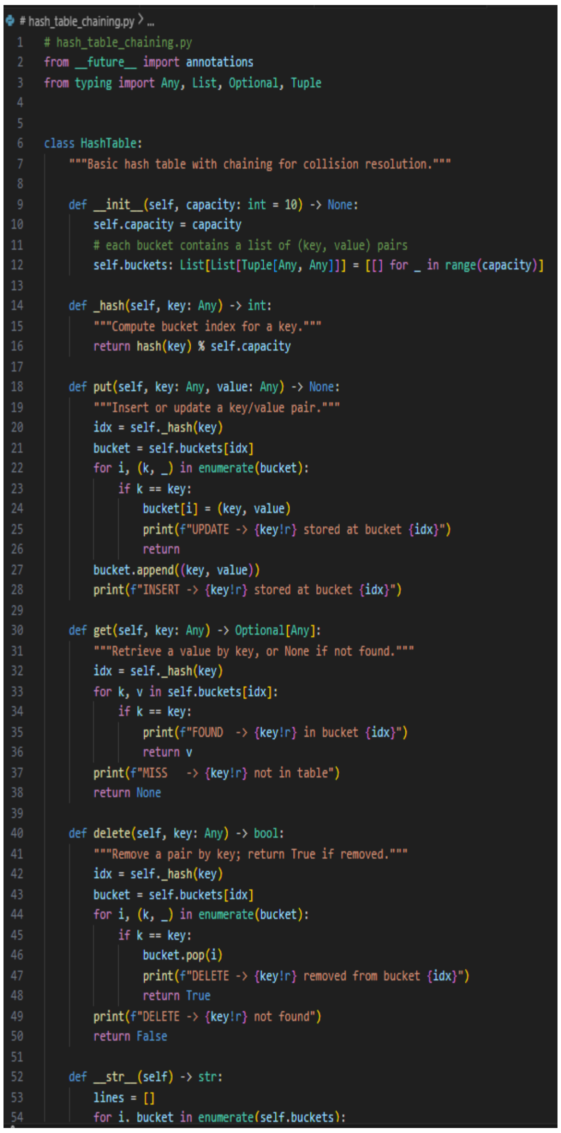
Starter Code

```
class HashTable:

pass
```

PROMPT:

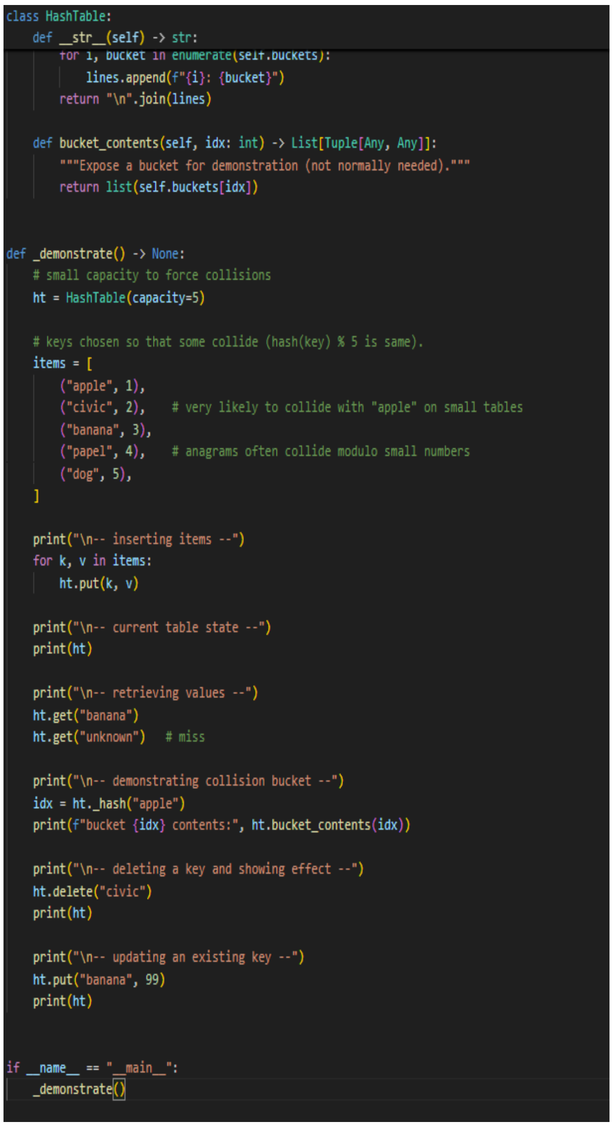
Write a Python program to implement a Hash Table and demonstrate how collision handling works using chaining. The objective of this task is to understand how data is stored, retrieved, and deleted efficiently using a hash function, and how collisions are managed when multiple keys map to the same index.

CODE:



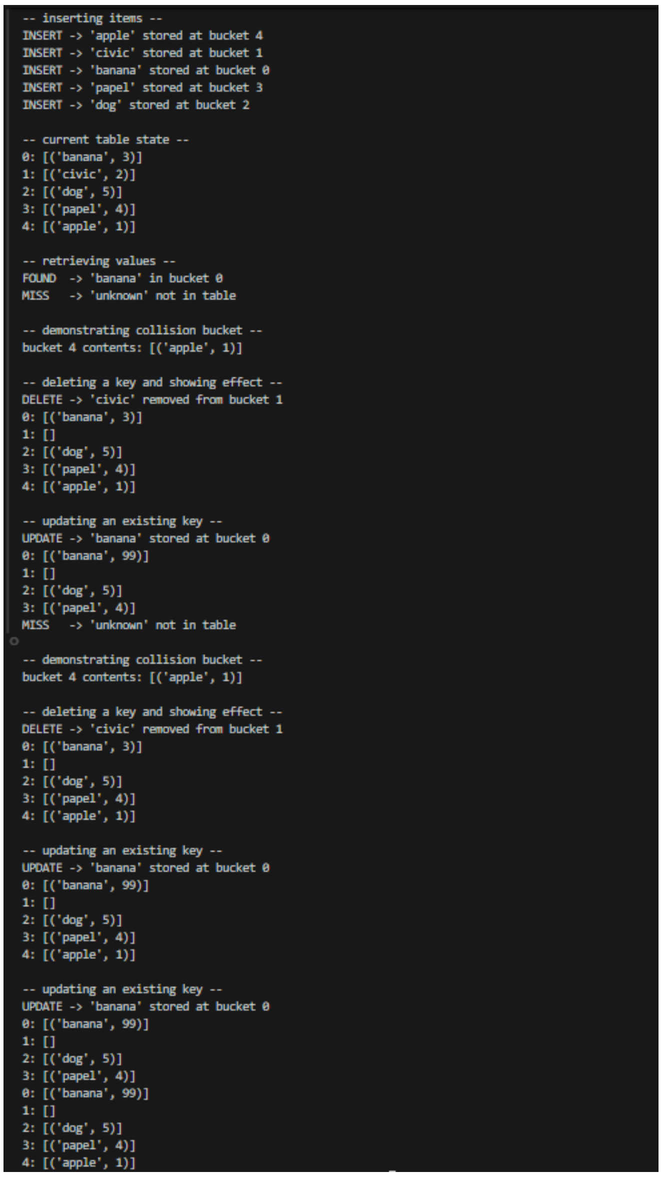
hash_table_chaining.py > ...

```python
# hash_table_chaining.py
from __future__ import annotations
from typing import Any, List, Optional, Tuple


class HashTable:
    """Basic hash table with chaining for collision resolution."""

    def __init__(self, capacity: int = 10) -> None:
        self.capacity = capacity
        # each bucket contains a list of (key, value) pairs
        self.buckets: List[List[Tuple[Any, Any]]] = [[] for _ in range(capacity)]

    def _hash(self, key: Any) -> int:
        """Compute bucket index for a key."""
        return hash(key) % self.capacity

    def put(self, key: Any, value: Any) -> None:
        """Insert or update a key/value pair."""
        idx = self._hash(key)
        bucket = self.buckets[idx]
        for i, (k, _) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                print(f"UPDATE -> {key!r} stored at bucket {idx}")
                return
        bucket.append((key, value))
        print(f"INSERT -> {key!r} stored at bucket {idx}")

    def get(self, key: Any) -> Optional[Any]:
        """Retrieve a value by key, or None if not found."""
        idx = self._hash(key)
        for k, v in self.buckets[idx]:
            if k == key:
                print(f"FOUND  -> {key!r} in bucket {idx}")
                return v
        print(f"MISS   -> {key!r} not in table")
        return None

    def delete(self, key: Any) -> bool:
        """Remove a pair by key; return True if removed."""
        idx = self._hash(key)
        bucket = self.buckets[idx]
        for i, (k, _) in enumerate(bucket):
            if k == key:
                bucket.pop(i)
                print(f"DELETE -> {key!r} removed from bucket {idx}")
                return True
        print(f"DELETE -> {key!r} not found")
        return False

    def __str__(self) -> str:
        lines = []
        for i, bucket in enumerate(self.buckets):
```



```python
class HashTable:
    def __str__(self) -> str:
        for i, bucket in enumerate(self.buckets):
            lines.append(f"{i}: {bucket}")
        return "\n".join(lines)

    def bucket_contents(self, idx: int) -> List[Tuple[Any, Any]]:
        """Expose a bucket for demonstration (not normally needed)."""
        return list(self.buckets[idx])


def _demonstrate() -> None:
    # small capacity to force collisions
    ht = HashTable(capacity=5)

    # keys chosen so that some collide (hash(key) % 5 is same).
    items = [
        ("apple", 1),
        ("civic", 2),    # very likely to collide with "apple" on small tables
        ("banana", 3),
        ("papel", 4),    # anagrams often collide modulo small numbers
        ("dog", 5),
    ]

    print("\n-- inserting items --")
    for k, v in items:
        ht.put(k, v)

    print("\n-- current table state --")
    print(ht)

    print("\n-- retrieving values --")
    ht.get("banana")
    ht.get("unknown")   # miss

    print("\n-- demonstrating collision bucket --")
    idx = ht._hash("apple")
    print(f"bucket {idx} contents:", ht.bucket_contents(idx))

    print("\n-- deleting a key and showing effect --")
    ht.delete("civic")
    print(ht)

    print("\n-- updating an existing key --")
    ht.put("banana", 99)
    print(ht)


if __name__ == "__main__":
    _demonstrate()
```

OUTPUT:



```
-- inserting items --
INSERT -> 'apple' stored at bucket 4
INSERT -> 'civic' stored at bucket 1
INSERT -> 'banana' stored at bucket 0
INSERT -> 'papel' stored at bucket 3
INSERT -> 'dog' stored at bucket 2

-- current table state --
0: [('banana', 3)]
1: [('civic', 2)]
2: [('dog', 5)]
3: [('papel', 4)]
4: [('apple', 1)]

-- retrieving values --
FOUND  -> 'banana' in bucket 0
MISS   -> 'unknown' not in table

-- demonstrating collision bucket --
bucket 4 contents: [('apple', 1)]

-- deleting a key and showing effect --
DELETE -> 'civic' removed from bucket 1
0: [('banana', 3)]
1: []
2: [('dog', 5)]
3: [('papel', 4)]
4: [('apple', 1)]

-- updating an existing key --
UPDATE -> 'banana' stored at bucket 0
0: [('banana', 99)]
1: []
2: [('dog', 5)]
3: [('papel', 4)]
MISS   -> 'unknown' not in table

-- demonstrating collision bucket --
bucket 4 contents: [('apple', 1)]

-- deleting a key and showing effect --
DELETE -> 'civic' removed from bucket 1
0: [('banana', 3)]
1: []
2: [('dog', 5)]
3: [('papel', 4)]
4: [('apple', 1)]

-- updating an existing key --
UPDATE -> 'banana' stored at bucket 0
0: [('banana', 99)]
1: []
2: [('dog', 5)]
3: [('papel', 4)]
4: [('apple', 1)]

-- updating an existing key --
UPDATE -> 'banana' stored at bucket 0
0: [('banana', 99)]
1: []
2: [('dog', 5)]
3: [('papel', 4)]
0: [('banana', 99)]
1: []
2: [('dog', 5)]
3: [('papel', 4)]
4: [('apple', 1)]
```

EXPLANATION:

A Hash Table stores key-value pairs using a hash function to generate an index.If two keys map to the same index, a collision occurs. Collision is handled using chaining, where multiple values are stored in a list at the same index. Insert, search, and delete operations usually take O(1) time on average.

Task5: Real-Time Application Challenge

Scenario:

Design a Campus Resource Management System with the following features:

• Student Attendance Tracking

• Event Registration System

• Library Book Borrowing

• Bus Scheduling System

• Cafeteria Order Queue

Tasks:

1. Choose the most appropriate data structure for each feature.

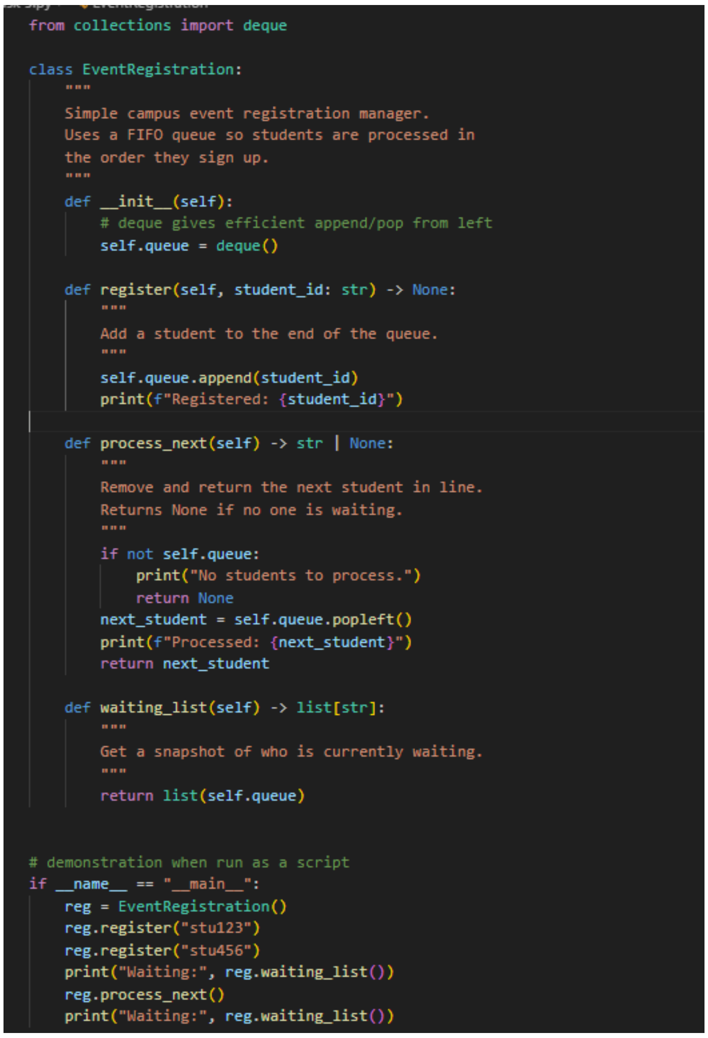
2. Justify your choice in 2–3 sentences.

3. Implement one selected feature using AI-assisted code generation**.**

PROMPT:

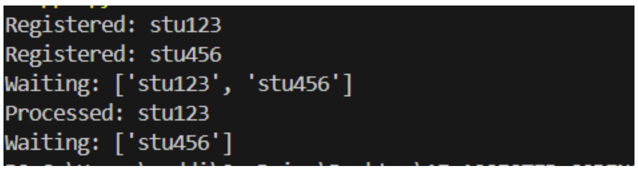
Design a Campus Resource Management System that efficiently manages multiple campus services including student attendance tracking, event registration, library book borrowing, bus scheduling, and cafeteria order processing. For each feature, select the most appropriate data structure based on how the data is stored, accessed, and updated. Clearly explain your choice in two to three sentences by describing why the selected data structure best fits the real-time requirements of that feature.

CODE:



```python
from collections import deque

class EventRegistration:
    """
    Simple campus event registration manager.
    Uses a FIFO queue so students are processed in
    the order they sign up.
    """
    def __init__(self):
        # deque gives efficient append/pop from left
        self.queue = deque()

    def register(self, student_id: str) -> None:
        """
        Add a student to the end of the queue.
        """
        self.queue.append(student_id)
        print(f"Registered: {student_id}")

    def process_next(self) -> str | None:
        """
        Remove and return the next student in line.
        Returns None if no one is waiting.
        """
        if not self.queue:
            print("No students to process.")
            return None
        next_student = self.queue.popleft()
        print(f"Processed: {next_student}")
        return next_student

    def waiting_list(self) -> list[str]:
        """
        Get a snapshot of who is currently waiting.
        """
        return list(self.queue)


# demonstration when run as a script
if __name__ == "__main__":
    reg = EventRegistration()
    reg.register("stu123")
    reg.register("stu456")
    print("Waiting:", reg.waiting_list())
    reg.process_next()
    print("Waiting:", reg.waiting_list())
```

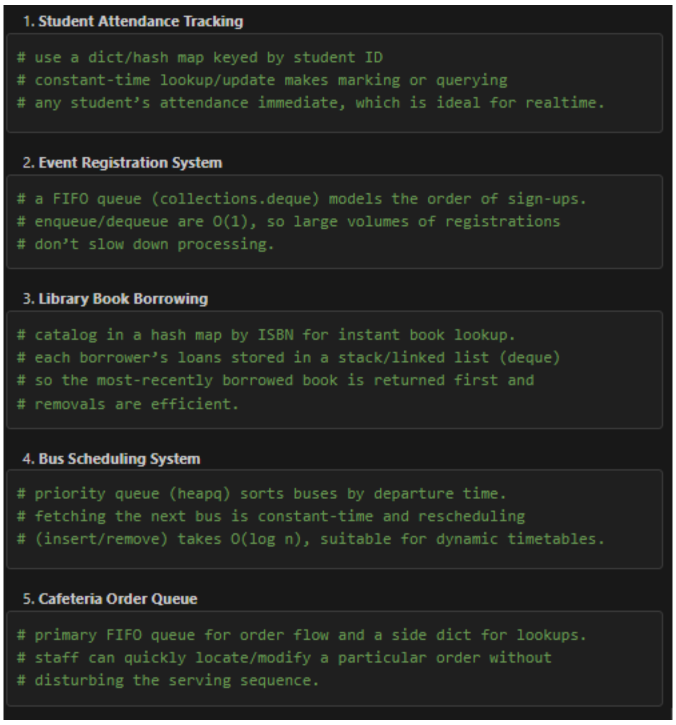
OUTPUT:



```
Registered: stu123
Registered: stu456
Waiting: ['stu123', 'stu456']
Processed: stu123
Waiting: ['stu456']
```

EXPLANATION:

I’ve chosen the Event Registration feature for the code sample. It demonstrates the queue structure .



1. **Student Attendance Tracking**

```
# use a dict/hash map keyed by student ID
# constant-time lookup/update makes marking or querying
# any student’s attendance immediate, which is ideal for realtime.
```

2. **Event Registration System**

```
# a FIFO queue (collections.deque) models the order of sign-ups.
# enqueue/dequeue are O(1), so large volumes of registrations
# don’t slow down processing.
```

3. **Library Book Borrowing**

```
# catalog in a hash map by ISBN for instant book lookup.
# each borrower’s loans stored in a stack/linked list (deque)
# so the most-recently borrowed book is returned first and
# removals are efficient.
```

4. **Bus Scheduling System**

```
# priority queue (heapq) sorts buses by departure time.
# fetching the next bus is constant-time and rescheduling
# (insert/remove) takes O(log n), suitable for dynamic timetables.
```

5. **Cafeteria Order Queue**

```
# primary FIFO queue for order flow and a side dict for lookups.
# staff can quickly locate/modify a particular order without
# disturbing the serving sequence.
```