Restaurant Table Management System

Problem Statement

In busy restaurants, managing customers efficiently is critical to improve customer satisfaction and to optimize table use. Customer seating is challenging especially during peak hours when customers arrive with or without reservations. There are several key challenges including minimizing waiting times, efficiently allocating tables based on customer size and giving priority to reservations.

This project aims to provide a simple Python-based simulation of a restaurant's table management system. It models customers arriving (both walk-ins and reservations), waiting in a priority queue, being seated at available tables, and tables progressing through various dining stages before being freed and cleaned. It simulates the queue management for both reservations and walk-ins, assigning tables based on customer size and table availability, updating table statuses and table lifecycles.

Chosen Data Structures

1. **Priority Queue**
   For our restaurant table management system, we have used the priority queue that is created based on reservation status and the reservation time. Reservation status is our first parameter we used to create the priority queue. We identify customers with reservations as 0 and walk-in customers as 1. In this way, when creating a queue, the people who have made reservations will get dequeued first, ensuring that we provide higher priority to the customers with reservation. Reservation time is the second parameter which is also the tie breaker in case we have two customers and both of them have made reservations. When there are two customers with reservations (0 and 0), then we give higher priority to one who arrived at the restaurant earliest (Let's say 8:00PM and 7:30PM and we give priority to the one who arrived at 7:30PM). Walk-in customers will get dequeued only after customers with reservations are dequeued. The method is self__lt__() in Customer class, and this serves as the comparator for the sort.

2. **TimSort (Insertion Sort/ Merge Sort)**
   In order to be able to implement our priority queue, in addition to the method: self__lt__() in Customer class, we had to use the .sort() method of python. We use self._queue.sort() method in our PriorityQueue class. Timsort is the combination of Insertion sort and the Merge sort. Insertion sort is when we insert the unsorted list into the correct position through iterations, while the Merge sort uses divide and conquer way to sort, meaning the unsorted list is divided until the smallest and then, put the numbers into the correct position by merging again. However, Timsort

only uses merge sort when the elements in the list are 64 or above. Since our customer queue is only around 10 and 20, the primary sorting algorithm that Timsort uses is insertion. TimSort is the standard sorting algorithm that python uses. Due to time constraints, instead of implementing this manually, we used python's built-in method.

3. **Array/List**

In our restaurant table management system, Array is used in multiple places.

(a) To store the tables

In our system, there are 4 types of tables:
(i) Table named August for 1-2 capacity,
(ii) Table named Spring for 3-4 capacity,
(iii) Table named Winter for 5-6 capacity,
(iv) Table named Fall for 7-8 capacity.
List is used to store the Table name and capacity and iterate easily to find a matching table based on the customer group size.

(b) To store the waiting list of customers (restaurant.cust_waiting_table)

List is used to store the list of customers who could not be immediately seated due to a lack of available tables after processing the list of customers from priority queue.

Code Walkthrough

1. Customer Class
   a. **def __init__(self, focal_name, size, is_reservation = False, reservation_time = None):**
   We initialize the constructor with the customer's name, the number of people per customer, reservation or no reservation (we keep the default as False), and the reservation time (we keep the default as none). We set the customer_type. If reservation is true, customer_type = 'reservation' else customer_type = 'walk-in'
   b. **def setting_priority(self):**
   This is where we lay out parameters to create priority queue, giving higher priority to customers with reservations and to those who arrived at the restaurant early.
   c. **def __lt__(self, other):**
   We use this as the comparator to create a priority queue.

2. Priority Queue Class
   a. We initialize the constructor with self._queue = [], for appending the customers. '._' is used to remind ourselves it is not recommended to access or modify from outside of the class.
   b. **def enqueue(self, customer_data):**

queue the customer by appending, and sort the queue. First, we made sure that the customer_data is in Customer class. Otherwise, we give a warning that only customers in Customer class can be queued.

c. **def dequeue(self):**
First, we make sure that the queue is not empty. After that we dequeue the customers.

d. **def is_front(self):**
This is to get the peek (the first one to be out) of the queue.

e. **Def is_empty(self):**
Check if the queue is empty or not.

f. **def queue_size(self):**
Get the size of the queue.

3. Table Class
   a. **def__init__(self, name, capacity)**:
   We initialize the constructor with the table's name, capacity with attributes such as is_occupied, status, clean, current_customer, order_stage and seated_time.
   b. **def is_available(self, customer_size)**:
   This method checks whether the table is available to seat new customers based on attributes such as its availability status, cleanliness and comparing the customer capacity and the table capacity.
   c. **def seat_customer(self, customer):**
   If the table is available and capacity matches, assign the customer to the table and change the table's status to occupied and mark it as dirty.
   d. **def free_table(self):**
   Reset the table's occupied status and seated_time and mark it as dirty.
   e. **def clean_table(self):**
   Reset the table's status to available and mark it as clean.
   f. **def update_table_status(table, stage):**
   This method manually advances the order stage of a dining table (such as ordered, food served, paid etc. When the stage is paid, it indicates that the table is free and now dirty.

4. Restaurant Class
   a. **def__init__(self, tables, cust_queue):**
   We initialize the constructor with an empty list of tables and an instance of PriorityQueue for cust_queue.
   b. **def setup_tables(self):**
   It makes the table list with various table names of different capacities.
   c. **Free_table and clean_table**: calls the respective methods from table class for table operations.
   d. **def assign_table(self):**
   We made the condition statement with a while loop, which will continue looping until the cust_queue (customer queue) becomes empty. We pop the First customer of the queue, (who is the highest priority) and then we assign tables. We have 4 different tables (table for 2 and less, table for 3 and 4 people, table for 5 and 6 people, and table for 7 and 8

people), so we assign tables to customers in accordance with the size of the customer. suitable_table is set as None initially. When a suitable table is found, we assign that table to the customer. When the table runs out, the customer is added to the cust_waiting_table, which is the customer list who are waiting for the table.

e. **def print_status(self):**
   It iterates the list of tables and prints their statuses such as table name, current_status, order_stage, clean and customer.

Complexity Analysis

1. Customer Class
   The time complexity of all the methods of Customer class is O(1).

2. Priority Queue Class
   The time complexity of enqueue() in PriorityQueue is O(N^2) while in the best case scenario, it is O(N). O(N) happens when the list is already sorted, which can happen only rarely.
   The time complexity of dequeue() in PriorityQueue is O(N) since we pop each customer per time.

3. Table Class
   The time complexity of all methods of Table Class is O(1).

4. Restaurant Class:
   The time complexity for set_up methods, clean_table, free_table and print_status is O(1).
   The time complexity for assign_ table is O(N) since we have to dequeue each time when we assign a table to a customer.
