Describe y justifica la estructura de datos usada para cada elemento (mesa, persona, cuenta, etc.).

- **Chain**: Chain es un struct con un array de **N_LOCATIONS locations** con el fin de almacenar todas las locations, utilicé un array debido a que será más sencillo acceder a sus elementos y es fácil construirlo debido a que conocemos **N_LOCATIONS.**

- **Location**: Location es un struct con un array de **N_TABLES tables** y un **id** con el fin de almacenar todas las tables para una misma **location**, utilicé un array debido a que será más sencillo acceder a sus elementos y es fácil construirlo debido a que conocemos **N_TABLES**.

- **Table**: Table es un struct con un **id**, **capacity**, un array de **capacity clients** y un **state** (true o false) con el fin de conocer la disponibilidad de la mesa y almacenar clientes en puestos, fue útil utilizar un array para de manera sencilla evitar que se superará la capacidad máxima **capacity**.

- **Client**: Client es un struct con una **order** y un **id**. Esto es útil para poder asignar las **order** por personas.

- **Menu**: Menu es un struct con un array de **N_ITEMS item** con el fin de almacenar todos los ítems, utilicé un array debido a que será más sencillo acceder a todos los ítems debido a que conocemos **N_ITEMS**.

- **Item**: Item es un struct con **price**, **id**, este almacena la información sobre un plato del menú.

- **Order**: Order en un struct con una lista ligada de **ítems**. En este caso no utilicé array debido a que se pueden agregar a la orden cuantos **ítems** se quiera por lo que no es posible definir un array.

Calcula y justifica la complejidad en notación O para cada uno de los eventos del programa.

- **CUSTOMER**

1. Se asocian las variables capacidad y ocupado → O(1)
2. Se recorre un puestos de la mesa buscando un puesto que sea igual a NULL → O(1)
3. Si no hay puestos igual a NULL se imprime un mensaje y se termina el algoritmo → O(1)
4. Se asigna un cliente a ese puesto → O(1)
5. Si aún quedan puestos por recorrer se vuelve a 2 → O(n-1)

La complejidad de CUSTOMER sería de O(n)

- **TABLE-STATUS**

1. Se imprime la capacidad de la mesa → O(1)
2. Se recorre un puesto de la mesa buscando un puesto que tenga asignado a un cliente → O(1)

3. Si el puesto tiene asignado un cliente aumenta la cantidad de clientes sentados en esa mesa → O(1)

4. Si aún quedan puestos por recorrer se vuelve a 2 → O(n-1)

5. Se imprime la capacidad restante de la mesa → O(1)

6. Se recorre nuevamente un puesto de la mesa buscando que tenga asignado a un cliente → O(1)

7. Si el puesto tiene asignado un cliente se imprime su id → O(1)

8. Si aún quedan puestos por recorrer se vuelve a 6 → O(n-1)

La complejidad de TABLE-STATUS es de O(n^2)

- **ORDER-CREATE**

1. Se recorre un ítem del menú buscando aquel que tenga asociado el id requerido → O(1)

2. Se busca si la mesa está habilitada, si no, termina → O(1)

3. Se recorre un puesto de la mesa buscando un puesto que tenga asignado al cliente del id correspondiente → O(1)

4. Si el id del ítem recorrido no corresponde con el correspondiente se vuelve a 1 → O(n-1)

5. Si el id del cliente recorrido no corresponde con el correspondiente se vuelve a 3 → O(n-1)

6. Se agrega la nueva orden al cliente → O(1)

La complejidad de ORDER-CREATE es de O(n^2)

- **ORDER-CANCEL**

1. Se recorre un puesto en la mesa→ O(1)

2. Si no hay un cliente asociado al puesto se vuelve a 1 → O(1)

3. Si el id del puesto no coincide con el id correspondiente al cliente se vuelve a 1 → O(n-1)

4. Se asignan los nodos para asignar el segundo valor (curr) y otro para asignar el primer valor (prev) → O(1)

5. Si en el primer puesto de la orden no hay un ítem asociado se vuelve a 1 → O(1)

6. Si el segundo puesto es el que no tiene un ítem asociado se elimina el primer ítem de la orden y termina → O(1)

7. Si el tercer puesto es el que no tiene un ítem asociado se elimina el segundo ítem de la orden y termina → O(1)

8. Si el siguiente a curr no es nulo, prev pasa a ser curr y curr ahora es el siguiente y se vuelve a 8 → O(n*log(n))

9. Borramos el último nodo que sería curr → O(1)

10. Redefinimos el último nodo como el prev que sería el anterior al último → O(1)

La complejidad de ORDER-CANCEL es de O(n^2)

- **BILL-CREATE**

1. Se recorre un puesto en la mesa→ O(1)

2. Si no hay un cliente asociado al puesto se vuelve a 1 → O(n)

3. Se recorre un pedido asociado al cliente → O(1)

4. Se actualiza el total a pagar del cliente y se imprime el producto → O(1)

5. Si aún hay productos se vuelve a 3 → O(n-1)

6. Si es el primer puesto, el cliente se vuelve el con mayor platos pedidos, el que más gasta y el que menos gasta → O(1)

7. Si no es el primer puesto, se compara para ver si cumple con tener la mayor cantidad platos pedidos, ser el que más gasta o el que menos gasta → O(1)

8. Se imprime la información restante → O(1)

La complejidad de BILL-CREATE es de O(n^2)

- **CHANGE-SEATS**

1. Se recorre un puesto en la mesa→ O(1)

2. Si no hay un cliente asociado al puesto se vuelve a 1 → O(1)

3. Si el cliente no es el correspondiente se vuelve a 1 → O(n-1)

4. Se guardan variables con la información del cliente → O(1)

5. Se recorre un puesto en la mesa→ O(1)

6. Si no hay un cliente 2 asociado al puesto se vuelve a 1 → O(1)

7. Si el cliente 2 no es el correspondiente se vuelve a 1 → O(n-1)

8. Se guardan variables con la información del cliente 2 → O(1)

9. Se intercambian los puestos el cliente y el cliente 2 → O(1)

La complejidad de CHANGE-SEATS es de O(n^2)

- **PERROU-MUERTO**

1. Se recorre un puesto en la mesa→ O(1)
2. Si no hay un cliente asociado al puesto se vuelve a 1 → O(1)
3. Si el cliente no es el correspondiente se vuelve a 1 → O(n-1)
4. Se accede al primer pedido del cliente → O(1)
5. Se asigna el pedido al primer cliente de la mesa → O(1)
6. Si aún hay más pedidos se vuelve a 4 → O(n-1)
7. Se elimina el primer pedido del cliente si no hay un segundo pedido → O(1)
8. Se elimina el siguiente pedido, si aún hay pedidos se vuelve a 8 → O(n-1)
9. Se elimina la orden asociada y el cliente → O(1)

La complejidad de PERROU-MUERTO es de O(n^3)

Comentar al menos 3 ventajas y desventajas de tanto listas ligadas y de arrays.

Listas Ligadas

- *Ventajas*

1. Tiene un fácil manejo para ingresar y eliminar elementos de ella.
2. Existe un ahorra en la memoria pues esta se aloca solo al ingresar un elemento a la lista sin ser necesario asignarle un espacio de memoria anteriormente.
3. Es una estructura de datos dinámica, lo que quiere decir que no hay que proporcionar un tamaño de memoria inicial, se encoge y agranda a medida sea necesario.

- *Desventajas*

1. Utilizan más memoria que los arreglos debido al almacenamiento utilizado por sus punteros.
2. Son secuenciales por lo que siempre deben leerse en orden
3. El coste de acceso a algún elemento de la lista depende mucho de su tamaño

Arrays

- *Ventajas*

1. Se puede manejar fácilmente a través de asignaciones de id, buscando en este por su posición directamente

2. Útil al minuto de querer aplicar algoritmos de búsqueda.

3. Fácilmente se pueden realizar matrices y otros almacenamientos más complejos.

- *Desventajas*

1. Necesidad de asignar con anterioridad un espacio de la memoria fijo para su almacenamiento.

2. Su concatenación con otro array requiere de crear un nuevo array

3. También deben crearse nuevos arrays al querer fragmentar uno

Investigar y comentar ventajas de C versus Python. (Principalmente orientado a la velocidad de cada uno)

Python es un lenguaje mucho más amigable y sencillo de usar que C debido a que es altamente complejo. Python está libre del uso de comas constantemente, punteros y otros conceptos complejos.

Como Python es interpretado a diferencia del lenguaje compilado C, la CPU procesa más instrucciones lo cual hace que correr un código en Python se haga mucho más lento ya que esté usa un intérprete.