Análisis Tarea 0

Estructura de datos y algoritmos.

Josefina Nicolet – N° alumno: 18642381

1) Estructuras usadas para cada elemento.

A continuación, se mencionarán todos los elementos utilizados y su justificación:

- Local: los locales son structs, que tienen guardado su id, la cantidad de mesas que tienen, el menú, la cantidad de comida que tiene el menú y un arreglo de punteros de mesas para así buscar las mesas con facilidad cuando se necesite hacer un evento, al igual que la comida en el menú. Se utiliza struct para que cada local tenga la información necesaria y que pueda ser diferente para cada uno, exceptuando el menú que es igual para todos, pero de esta forma los locales pueden operar con la comida que los clientes piden por mesa, así, aunque e supiera desde el inicio cuantos locales serían y estos no cambiarían, cada local debía guardar información aparte como las meas y el menú por lo que un arreglo no hubiera funcionado, mientras que el struct nos deja construir un arreglo con todos los locales guardando toda su información. Cabe destacar, que de esta manera la mayoría de los eventos ocurren en los locales y dentro de estas operaciones llamamos a otras.
- Mesa: las mesas son structs, para que cada mesa pueda tener sus propias funciones y atributos, tales como su id, su capacidad, el id del local en que están ubicadas, la cantidad de clientes que han llegado a la mesa y un arreglo de punteros de clientes para representar los asientos, los que parten en nulo y se van cambiando cuando llega un cliente. Las mesas tienen funciones que proceden a las funciones del local, para así tener bien separado los archivos, estas se encargan de trabajar con los clientes y llamar a otras funciones. Por la misma razón que la dicha en locales, las mesas además de un id, tienen información de clientes, por lo que es necesario un struct para poder operar con ellas internado y no perder información de los clientes.
- Cliente: Los clientes son structs, para que así cada cliente pueda tener funciones y atributos tales como su id, la cantidad de platos que ha pedido, la cantidad de costos que lleva (precio de los platos pedidos), un puntero a una cuenta (que es una lista ligada) que es la primera cuenta (comida) pedida y un puntero a la ultima cuenta que es la ultima comida pedida. Por otro lado, tiene funciones para operar con sus atributos y al mismo tiempo llamar a otras funciones para poder ver sus comidas pedidas.
- Cuenta: las cuentas son un struct que se utiliza como lista ligada, para así poder tener las cuentas de los clientes y poder agregar o eliminar comidas sin problemas, se hace de esta manera, ya que el cliente puede pedir todos los platos que desea, puede devolver el ultimo pedido o irse sin pagar sus pedidos, por lo que estos pasan a otro cliente, por lo que al hacer una lista ligada nos facilita estas acciones, y la forma de contruirla es con structs. Además también tiene “atributos”, son el item (la comida pedida) y next, que es el siguiente nodo de comida pedida, y funciones para modificar la lista ligada o para imprimir los pedidos de un cliente en la boleta de mesa.
- Comida: la comida es un struct, para poder lograr que cada comida tuviera su id y su precio “guardado” y así que las comidas se pudieran utilizar en otras estructuras sin perder su información y principalmente, que estas se pudieran usar en diferentes estructuras que se iban a editar, y no editar la comida en sí (menú y cuentas), que

tiene más de una información que guardar. Cabe destacar, que la comida solo tiene las funciones necesarias para crearse y destruirse.

- Menú: el menú es un arreglo, en donde cada item del arreglo es un puntero a una comida. Se hizo de este modo ya que, desde el inicio sabíamos cuántos ítems de comida tendría el menú, no se iba a editar y no tiene información que debe ser guardada además de las comidas, por lo que no era necesario un struct.

Principalmente, los elementos hechos como structs son aquellos que tienen más de una información por guardar, dado que con esta estructura podemos guardar la información junta con solo un puntero y editarla de manera sencilla sin errores.

2) Complejidad de cada evento.

Tomaremos los siguientes datos:
l->cantidad de locales
m_l->cantidad de mesas por local
c_m->cantidad de clientes que pueden estar en una mesa
f->cantidad de comida en el menú o cantidad de comida pedida por un cliente, dado que el cliente puede pedir todas las comidas que quiera y eso en el peor caso sería todo el menú.
Por otro lado, me guiare con lo creado en mi programa, por lo que no será teórico.
Eventos:

A) Apertura de restaurant:
   1) Open table: En primer lugar, se busca entre todos los locales cual es el local con el id que estoy buscando (o(l)), cuando lo encuentro se busca entre todas las mesas del local cual es la primera que no esta inicializada en el arreglo (dado que no se lleva cuenta de cuantas lleva inicializadas hasta ese momento) y cuando se encuentra la primera posición nula del arreglo de punteros se inicializa la mesa. Esto nos daría un orden de O(l*m_l) dado que dentro del for de búsqueda de local hay un for de búsqueda de mesa.
      Antes de esto, existe un for que saca información del archivo l*m_l veces, por cada cantidad de mesas, y lo dicho anteriormente está dentro de este for , por lo que el orden final sería de O(l*m_l*l*m_l) ~~ O(l^2 * m_l ^2). Al l y m_l ser números enteros, podemos simplificar a O(n^4).
   2) Menu item: Acá ya se tiene un arreglo creado con la cantidad de item en el menu, por lo que existe un for que, por cada item a ingresar, inicializamos la comida y luego la introducimos al arreglo, lo que nos da un O(f).
      Luego de eso, por cada local actualizamos la información del menú, por lo que tenemos que O(l).
      Como estas operaciones no son en cadena, el orden final del evento es dependiendo de que es mayor, siendo orden del número mayor, pero como f y l son números entero tenemos que la complejidad final del evento es de O(n).
   3) Customer: En este evento, primero se busca cual es el local que el cliente va con una función externa, por lo que con eso tenemos O(l). Luego, por cada mesa del

local encontrado (no es en cadena, por que sería una suma) se busca la mesa con el id que nos dan O(m_l), teniando hasta ahora O(l + m_l).
Con la mesa y el local buscados, se llama a la función "llega_cliente", en donde por cada asiento de la mesa buscamos el primer asiento vacio y lo inicializamos o imprimimos que no había asientos O(c_m).
Finalmente tenemos que la complejidad del evento es O(l + m_l + c_m), que dependerá de el numero mayor, pero como los tres son números enteros podemos decir que su complejidad es O(n).

4) Table status: Para este evento, nuevamente buscamos el local con el id entregado gracias a la función externa y luego de encontrarlo se busca la mesa entre las mesas del local, por lo que hasta ese momento tenemos complejidad O(l + m_l).
Luego, por cada cliente se imprime su id o "_" si no hay cliente, por lo que nuevamente es un for que va por todos los asientos de la mesa, dado que nos son iteraciones en cadena, nuevamente los sumamos, lo que nos da una complejidad final del evento O(l + m_l + c_m), que dependerá del nuemro mayor, pero como los tres son números enteros tenemos O(n).

B) Manejar pedidos:

1) Order create: primero se busca el local con el id que nos dan con la funcion externa O(l) y luego llamamos a la función "crear_orden", ahí se busca el item que se quiere agregar al cliente con un for que itera por todos los ítems (O(f)), lo que nos da hasta ahora O(l + f).
Luego, buscamos por cada mesa del local si el id es el que buscamos O(m_l), si es llamamos a la función "agregar_plato_cliente" en donde buscamos por cada cliente de la mesa si el id es el mismo que nos dan O(c_m), para después agregar la comida como el item de la cuenta. Por esta acción tenemos esas dos búsquedas en cadena, por lo que tenemos una complejidad de O(m_l * c_m).
Finalmente, nuestra complejidad sería O(l + f + m_l * c_m), como todo son números enteros, O(n + n + n^2), como m_l * c_m es mayor tenemos que nuestra complejidad final es O(m_l * c_m) ~~ O(n^2).

2) Order cancel: primero buscamos el local por todos los locales con la función externa O(l), luego llamamos a la función "cancelar_orden" en donde por cada mesa del local ya encontrado, buscamos la mesa que necesitamos O(m_l), dado que la mesa ya estaba definida anteriormente no necesitamos hacer otra iteración en cadena para buscar al cliente. Ahora, cuando ya encontramos la mesa buscamos entre todos los clientes de la mesa al cliente que queremos, cuando lo encontramos tenemos 2 opciones, si el cliente solo había pedido un solo plato eliminamos esa cuenta, o si el cliente tiene más de un plato llamamos a "eliminar_cuenta" en donde se recorre por toda la lista ligada de cuentas hasta el último elemento y lo eliminamos, en donde en el peor caso el cliente podría tener todos los ítems del manu en su cuenta por lo que sería recorrer f elementos.
Finalmente, tenemos O(l + m + c_m * f) ~~ O(c_m * f) ~~ O(n^2).

3) Bill create: Como en todos los eventos, buscamos el local que necesitamos O(l) y luego llamamos a la función “bill_create”, en donde recorremos todas las mesas del local para buscar la mesa que necesitamos O(m_l), son iteracions seguidas pero no en cadena (O(l + m_l)), luego llamamos a la función “bill_create_table” en donde recorremos cada cliente de la mesa, si este tiene una cuenta asociada (que sería el peor caso) recorremos cada elemento de la lista ligada (en el peor caso, todo el menú) para imprimir el item guardado en cada uno, lo que nos da una complejidad O(c_m * f).
   Finalmente, sumando los dos procesos, tenemos O(l + m_l + c_m * f) ~~ O(c_m * f) ~~ O(n^2).

C) Manejar restaurant:
   1) Change seats: Primero, se busca el local que necesitamos con la función externa, la que recorre todos los locales (O(l)) l uego se llama a la función “cambiar_asientos” en donde se recorren todas las mesas del local para encontarr la mesa 1 y la mesa 1, O(m_l) dado que la búsqueda de local y mesa son bloques separados. Ya con la mesas encontrada, por cada mesa hacemos un recorrido por sus asientos para encontrar a los clientes respectivos, esto se hace con for de los asientos de la mesa 1 y luego for de los asientos de la mesa dos (O(c_m + c_m)). Finalmente, con las mesas y los clientes encontrados se hace el cambio de orden 1, por lo que la complejidad final del evento sería O(l + m + c_m + c_m) ~~ O(4n) ~~ O(n).
   2) Perrou muerto: Como todas las anteriores, se busca el local con la función, que funciona como un bloque separado, O(l), luego se llama a la función “perro_muerto_location”, en donde se busca la mesa en cuestión, como la mesa ya estaba declarada no se necesita hacer bloques en cadena (O(m_l)). Una vez ya el recorrido por las mesas finalizado, llamamos a la función “perro_muerto”, en donde de recorren todos los clientes de la mesa para buscar al que se escapó (O(c_m)). Una vez terminado el recorrido de los clientes, se hace un segundo para buscar al primer cliente sentado en la mesa (en issues dijeron que siempre habría un primero, pero lo hice de esta manera por casos borde en donde el primero se haya ido antes) (O(c_m)). Una vez con ambos clientes identificados hago los cambios respectivos y se libera al cliente que se escapó, pero antes de eso su cuenta se cambia a nulo, por lo que no hay que recorrer esa lista ligada para liberarla.
      Finalmente, tenemos que la complejidad del evento es O(l + m_l + c_m + c_m) ~~ O(4n) ~~ O(n).

Principalmente me enfoque en buscar los locales y las mesas en bloques separados (declarándolas antes) para hacer el programa más eficiente y evitar hacer iteraciones en cadena, además de incluir breaks cuando los elementos eran encontrados para que en casos mejores no se tuviera que recorrer todo el arreglo.

3) Listas ligadas y Arrays.

A) Listas ligadas:

1) Ventajas:

- Tiene largo variable
- Almacena mediante punteros, como los punteros tienen el mismo largo, no es necesario que sea el mismo tipo.
- Podemos agregar o eliminar elementos y trozos enteros de la lista tan solo cambiando un puntero (de manera sencilla)

2) Desventajas:

- No podemos movernos a un lugar de la lista en un solo paso, se debe recorrer uno por uno. (búsqueda)
- Almacena de manera aleatoria, conectada con punteros, lo que hace más complejos los procesos
- Solo es un "linea recta" no podemos agrupar dentro de una lista. (no lista de listas)

B) Arrays:

1) Ventajas:

- Almacena elementos de manera continua en memoria (fácil de buscar, dado que no necesitamos punteros)
- Permite acceso a un índice en tiempo O(1)
- Podemos hacer matrices, arreglos de arreglos, y agrupar de diferentes maneras.

2) Desventajas:

- No podemos insertar elementos de diferentes tipos en el mismo arreglo
- No podemos modificar su cantidad de elementos (largo fijo)
- Se debe tener claro desde un principio la cantidad de elementos y la forma de agrupación, dado que no se puede modificar.

4) C v/S Python (velocidad).

Se sabe que C supera a Python en velocidad, dado que c trabaja directamente con la memoria, mientras que Python hace muchas cosas por detrás que uno no puede optimizar, por eso C es un lenguaje más optimo, dado que no se pierde tiempo realizando acciones no necesarias.

Por otra parte, Python es un lenguaje interpretado, mientras que C es compilado, al ser compilado este se comunica directamente con el PC (habla su mismo idioma). Python también es un lenguaje de alto nivel, lo que hace que exprese los programas de una manera sencilla, mientras que C es de bajo nivel, es trasladado fácilmente a lenguaje de máquina. Por último, Python es un lenguaje dinámicamente tipado, en donde el tipo de variables se deciden en tiempo de ejecución, mientras que C es fuertemente tipado, en donde los tipos de datos se definen antes y no se permiten violaciones de estos. Por estas razones, C tiene ventajas sobre Python, dado todo lo dicho anteriormente hace que C funcione de manera más eficiente y rápida al usar menos tiempo convirtiendo o decidiendo los tipos de datos en el momento.