Restaurante Ocupado

NOTA: SI USTED ESTÁ LEYENDO ESTE DOCUMENTO SI HABERLO DESCOMPRIMIDO, CIÉRRELO INMEDIATAMENTE Y DESCOMPRIMA EL CONTENIDO DEL ARCHIVO ZIP QUE RECIBIÓ EN EL ESCRITORIO. UNA VEZ HECHO ESTO, REANUDE SU TRABAJO DESDE ALLÍ. TRABAJAR ANTES DE DESCOMPRIMIR LE HARÁ PERDER SUS CAMBIOS, Y NO TENDRÁ OPORTUNIDAD DE RECLAMAR. LAMENTABLEMENTE ESTO HA OCURRIDO Y NO SE PUEDE HACER NADA.

Cierto restaurante de la ciudad tiene tal fama que los clientes hacen cola en la entrada para poder disfrutar de su menú. El restaurante tiene N mesas (numeradas de 0 a $N-1$). Para simplificar consideraremos que las mesas son individuales y pueden estar ocupadas por un solo cliente a la vez (realmente un cliente puede ser un grupo y una mesa tener varios asientos y el problema sería el mismo). Cuando un nuevo cliente llega al restaurante, siempre se ubica al final de la cola, a menos que esta esté vacía. Si no hay nadie en la cola, y hay alguna mesa no ocupada, el cliente pasa directamente a la mesa. Si hay varias mesas desocupadas ocupará la mesa de menor numeración (usted puede interpretar que a menor índice corresponde una mesa con mejor ubicación). Cuando una mesa se vacía se pasará inmediatamente al cliente al frente de la cola de entrada (a menos que la cola esté vacía).

Cada cliente desea consumir una cierta cantidad M_i de platos (cantidad que puede ser diferente para cada cliente). Se considera que el tiempo de consumir un plato es el mismo independientemente del plato y del cliente. Por lo tanto un cliente con mayor cantidad de platos a consumir demorará más en terminar (tener ocupada la mesa) que un cliente con menor cantidad de platos a consumir. Note por lo tanto que aunque un cliente haya ocupado antes que otro una mesa no necesariamente saldrá del restaurante antes que el otro (puede que consuma más platos que uno que llegó después).

En este restaurante todo el trabajo de entrar y servir a los clientes lo hace un único mesero. Cuando se invoca al método `LlegaCliente` el cliente se pone en la cola de la entrada, el mesero revisa la cola de la entrada y hace pasar exactamente a un cliente a ocupar la primera mesa vacía (la de menor numeración). Luego el mesero da una ronda por todas las mesas para que cada cliente consuma exactamente uno de sus platos (incluyendo el cliente que puede haber acabado de pasar).

En el restaurante solo hay una caja para pagar. En el método `ProximoAPagar` (supuestamente invocado por el cajero) el mesero envía a la caja a un cliente que ya haya consumido todos sus platos (si hay varios en ese caso se envía al cliente de la mesa de menor numeración). Inmediatamente en este momento se pasa el primero de la cola de entrada (si es que hay clientes esperando en la cola) para la mesa que se acaba de desocupar, y el mesero vuelve a dar una única ronda por todas las mesas (donde cada cliente consume exactamente uno de sus platos).

Si al invocar `ProximoAPagar`, después de una ronda, no hay ningún cliente que haya consumido todos sus platos el mesero se mantiene dando rondas por todas las mesas hasta que al terminar una ronda completa haya al menos un cliente que haya consumido todos sus platos. Una vez llevado a este cliente a la caja,

entonces vuelve a mirar a la cola, ubica a un cliente nuevo en la mesa que se acaba de vaciar (en caso de existir algún cliente en la cola) y vuelve a dar exactamente una ronda por todas las mesas.

Es decir, el método `ProximoAPagar` garantiza que el mesero se mantendrá dando rondas y atendiendo clientes hasta que algún cliente termine o devolverá `null` en caso de que no haya ningún cliente.

NOTAS:

El restaurante lleva también ciertas estadísticas sobre su funcionamiento. El método `PlatosConsumidos` nos da la cantidad de platos que hasta el momento se han consumido en una determinada mesa. La propiedad `EnLaCola` devuelve la cantidad de clientes que se encuentran esperando en la cola. El método `PorPagar` nos da a todos los clientes que al ser invocado el método ya han consumido todos sus platos y están esperando en sus mesas por ser llamados a pagar. Los clientes se devuelven en el mismo orden de numeración de las mesas (0 a $N - 1$).

Implementación

Usted debe implementar una solución que simule el funcionamiento del restaurante. Para ello, debe implementar la siguiente `interface`.

```
public interface IRestaurante
{
    /// <summary>
    /// Llega un nuevo cliente al restaurante ocupa mesa o se pone en la cola de entrada.
    /// </summary>
    void LlegaCliente(Cliente cliente);

    /// <summary>
    /// Devuelve el próximo cliente a pagar.
    /// </summary>
    Cliente ProximoAPagar();

    /// <summary>
    /// Devuelve la cantidad de clientes en la cola de la entrada.
    /// </summary>
    int EnLaCola { get; }

    /// <summary>
    /// Devuelve la cantidad de platos que se han consumido en una mesa.
    /// </summary>
    int PlatosConsumidos(int mesa);

    /// <summary>
    /// Devuelve todos los clientes que han terminado y esperan por pagar.
    /// </summary>
    IEnumerable<Cliente> PorPagar();
}
```

La clase `Cliente` almacena la cantidad de platos que este cliente desea consumir.

```
public class Cliente
{
    public int PlatosAConsumir { get; private set; }

    public Cliente(int platos)
    {
        PlatosAConsumir = platos;
    }
}
```

En la solución entregada usted encontrará una biblioteca de clases con los tipos definidos anteriormente, así como una implementación vacía de la clase `Restaurante`. Esta clase ya implementa la interface `IRestaurante`, y le servirá de base. Asegúrese de emplear esta clase pues su constructor es necesario para el correcto funcionamiento del probador.

```
public class Restaurante : IRestaurante
{
    public Restaurante(int mesas)
    {
        throw new NotImplementedException();
    }

    public void LlegaCliente(Cliente cliente)
    {
        throw new NotImplementedException();
    }

    public Cliente ProximoAPagar()
    {
        throw new NotImplementedException();
    }

    public int EnLaCola
    {
        get { throw new NotImplementedException(); }
    }

    public int PlatosConsumidos(int mesa)
    {
        throw new NotImplementedException();
    }

    IEnumerable<Cliente> PorPagar()
    {
        throw new NotImplementedException();
    }

}
```

Además, encontrará una aplicación de consola con un conjunto reducido de casos de prueba. Es su responsabilidad adicionar los casos que considere necesarios para validar su implementación.

Ejemplo

NOTA: Las letras que se les han dado a los clientes son a modo ilustrativo para facilitar la comprensión visual del ejemplo. En su solución esto no es necesario. Los clientes se distinguen por la referencia a un valor tipo `Cliente` que se recibe como parámetro.

Acción	Devuelve	EnLaCola	Mesa 0	Mesa 1	Mesa 2
`inicio`					
Creado un restaurante de 3 mesas. La cola y las mesas están vacía.					
`LlegaCliente(A(4))`			`A(3)`		
Llegó un cliente a consumir 4 platos y lo ubicó en la mesa 0 y dio una ronda. El cliente consume su primer plato.					
`LlegaCliente(B(3))`			`A(2)`	`B(2)`	
Llegó un cliente a consumir 3 platos y lo ubicó en la mesa 1 y dio una ronda.					
`LlegaCliente(C(1))`			`A(1)`	`B(1)`	`C(0)`
Llegó un cliente a consumir 1 plato y lo ubicó en la mesa 2 y dio una ronda. Note que en esta misma ronda termina C, que es el cliente que acaba de entrar.					
`PorPagar`	`C`		`A(1)`	`B(1)`	`C(0)`
Enumerable con los clientes que ya han terminado. Note que es solo informativa. La información en el restaurante se mantiene igual y C se debe mantener en su mesa hasta que sea llamado a pagar.					
`LlegaCliente(D(1))`		`D(1)`	`A(0)`	`B(0)`	`C(0)`
Llega cliente como todas las mesas están ocupadas, el mesero lo pone en la cola y da una ronda por las mesas. Como C ya había terminado su estado no cambia, pero A y B consumen ambos su último plato.					
`PorPagar`	`A, B, C`	`D(1)`	`A(0)`	`B(0)`	`C(0)`
Ahora el enumerable de esperando por pagar es A, B, C.					
`LlegaCliente(E(3))`		`D(1), E(3)`	`A(0)`	`B(0)`	`C(0)`
Llega cliente y se pone en la cola porque todas las mesas están ocupadas. Aunque el mesero da una ronda por las mesas para ver si los clientes consumen no puede liberar ninguna mesa porque no ha sido invocado el método `ProximoAPagar`.					
`ProximoAPagar`	`A`	`E(3)`	`D(0)`	`B(0)`	`C(0)`
El cliente A sale a pagar, se libera le mesa 0, se pasa al cliente D de la cola a la mesa 0 y da una ronda por todas las mesas. En la mesa 0 queda el cliente D que ha consumido entonces su único plato.					
`ProximoAPagar`	`D`		`E(2)`	`B(0)`	`C(0)`
El cliente D sale a pagar (note que según este sistema sale antes que B y C aunque llegó después). Queda libre la mesa 0 se pasa a E que estaba en la cola y se da una ronda de consumo por todas las mesas.					
`ProximoAPagar`	`B`		`E(1)`		`C(0)`
El cliente B sale a pagar. No hay nadie en cola para entrar. Se da una ronda de consumo por todas las mesas.					
`PorPagar`	`C`		`E(1)`		`C(0)`
`LlegaCliente(F(5))`			`E(0)`	`F(4)`	`C(0)`
Llega cliente F, como no hay nadie en cola se pasa a mesa 1 y se da una ronda de consumo.					
`ProximoAPagar`	`E`			`F(3)`	`C(0)`
`ProximoAPagar`	`C`			`F(2)`	
`ProximoAPagar`	`F`				
Se pide cliente a pagar, de inicio no hay ninguna mesa con un cliente con sus platos consumidos. El mesero se mantiene dando rondas hasta que un cliente ha consumido todos sus platos y este es el que sale. Posteriormente se dirige a la cola, pero ya no queda nadie esperando.					

`ProximoAPagar`	`null`				
No hay clientes a pagar.					
`PlatosConsumidos(1)`	`8`				
En la mesa 1 se han consumido 8 platos (3 por el cliente B y 5 por el cliente F).					

Notas

- Como habrá notado, el mesero solamente se mueve en dos ocasiones: inmediatamente después de una llamada a `LlegaCliente`, pasa al primero de la cola en caso de ser necesario, y da exactamente una ronda por cada mesa. Durante una llamada a `ProximoAPagar` da tantas rondas por las mesas como haga falta para que salga el próximo cliente (sin atender en todo este proceso a la cola). Si ya hubiere algún cliente con 0 platos, entonces no es necesario dar ninguna ronda. Únicamente después de que algún cliente sale, el mesero atiende nuevamente a la cola, y da una ronda más (exactamente igual que si se hubiera hecho `LlegaCliente`).
- Cada ronda es completa, es decir se debe garantizar que el mesero pase por todas las mesas aun cuando encuentre un cliente que ya ha consumido todos sus platos.
- El mesero no libera mesas, para que una mesa quede libre tiene que ser como resultado de una llamada al método `ProximoAPagar`. Si en una ronda producto de una llamada a `LlegaCliente` algún cliente en alguna mesa ya tenía todos sus platos consumidos, el mesero simplemente lo ignora y sigue atendiendo al resto de las mesas hasta completar la ronda.
- `ProximoAPagar` debe devolver **exactamente la misma instancia** de `Cliente` que haya sido recibida en su momento en una llamada a `LlegaCliente`. Usted **no puede** modificar la clase `Cliente` ni el valor de su propiedad `PlatosAConsumir` (que por demás tiene un `set` privado).
- Note que los clientes no necesariamente pagan en el orden en que terminan. Cuando se invoque al método `ProximoAPagar` **siempre** pagará primero el cliente de la mesa con menor numeración (que haya terminado, claro está), aunque hubiere terminado después de otro cliente en otra mesa con mayor numeración.
- Entre dos llamadas consecutivas a `ProximoAPagar` se puede realizar cualquier cantidad de llamadas a `LlegaCliente`, sin que esto afecte el funcionamiento del sistema.
- Si no quedan clientes, su implementación de `ClienteAPagar` debe devolver `null`. Además de esto, usted no necesita realizar ninguna otra validación. Siempre se pasará una instancia de `Cliente` no `null`, y siempre se preguntará por una mesa válida en el método `PlatosConsumidos`.