Algoritmos y Estructuras de Datos

Ejercicios resueltos en la clase del 10 de septiembre
Tipos Abstractos de Datos
Segundo Cuatrimestre 2025

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ejercicio del 1er recuperatorio del 1er cuatrimestre 2025

Enunciado

Nos piden especificar un sistema para modelar una red de telefonía celular. La red está compuesta de antenas a las que se conectan los teléfonos para recibir el servicio. Para cada antena sabemos qué antenas son vecinas a ella, y no hay antenas aisladas, es decir, sin antenas vecinas.

A medida que el teléfono se mueve por la red se va conectando a distintas antenas, sólo una por vez. En su recorrido, un teléfono que está conectado a alguna antena sólo puede pasar de las vecinas de esta. También puede pasar que se desconecte de la red, en cuyo caso más adelante podrá volver a conectarse en cualquier antena.

Se quiere poder obtener en cualquier momento el recorrido de un teléfono por las antenas de la red y qué teléfonos están conectados a una antena en particular.

En esta primer versión del sistema sabemos de antemano qué antenas componen la red y qué teléfonos la usan, aunque estos no están conectados a ninguna al iniciar el sistema.

El TAD que especifica este problema tiene las siguientes operaciones

- **crearRed** que inicializa el sistema.
- **conectarTeléfono** que registra cuando un teléfono se conecta a una antena.
- **desconectarTeléfono** que registra cuando un teléfono se desconecta de la red.
- **recorridoTeléfono** que devuelve el recorrido de un teléfono por la red.
- **conexionesAntena** que devuelve los telefonos conectados a una antena.

1. Para completar el TAD
   a) Complete las operaciones con todos sus parámetros, definiendo los renombres de tipos que considere necesarios.
   b) Defina los observadores del TAD.
   c) Especifique completamente las operaciones **crearRed** y **conectarTelefono**.
   d) Describa brevemente cómo resuelve las demás operaciones con sus observadores.

2. ¿Qué habría que modificar en este TAD para permitir que se agreguen antenas después de iniciado el sistema? Describa brevemente (sin especificar), si es necesario
   - Qué procs hay que agregar
   - Qué observadores hay que agregar
   - Qué observadores hay que modificar
   - Qué cambios hay que hacer en la especificación de los procs ya definidos

Desarrollo

1. a)

Identifiquemos primero qué entidades participan de las acciones que hace nuestro TAD. Podemos ver que la red interactúa con *antenas* y *teléfonos*. Detengámonos a ver si el enunciado habla de alguna característica de los mismos: Vemos que no. Entonces podemos identificarlos con un string o simplemente con un número.

Antena, Telefono `ES` $\mathbb{Z}$

crearRed

El enunciado nos dice que "En esta primer versión del sistema sabemos de antemano qué antenas componen la red y qué teléfonos la usan". Podemos asumir entonces que cuando el sistema arranca (mediante el proc crearRed) se le pasan las antenas y los teléfonos. Y qué ocurre con las relaciones de vecindad de las antenas? Podemos también asumir que están fijas.

£Qué tipo usamos para los teléfonos? Como no hay repetidos y no nos importa el orden, usaremos conjuntos. £Y para las antenas? Aprovechemos y usemos una estructura que nos permita expresar tanto las antenas disponibles como la relación de vecindad con otras. Nos decidimos por un dicc que tiene como clave una antena y como valor un conjunto de antenas, que serán sus vecinas (más tarde veremos que hay varias restricciones a ese dicc, que se expresarán en el requiere). Finalmente vemos que claramente todos los parámetros son *in*, pues no se tienen que modificar y el valor de retorno es la red creada.

`proc crearRed`(in a : dicc$\langle Antena, \text{conj}\langle Antena\rangle\rangle$, in t : conj$\langle Telefono\rangle$) : Red

igualdad

No hay mucha ciencia acá, es siempre igual...

`proc igualdad`(in r_1 : Red, in r_2 : Red) : bool

conectarTelefono

Según el enunciado, un teléfono puede conectarse a una antena por vez. Pasaremos entonces el teléfono y la antena a la que se quiere conectar. Como decíamos, las restricciones sobre qué conexiones son posibles y cuáles no, serán expresadas en el requiere.

Finalmente, vemos que el parámetro de la red debe ser inout (pues se modifica), los parámetros teléfono y antena deben ser in, y no necesitamos valor de retorno.

`proc conectarTelefono`(inout r : Red, in t : $Telefono$, in a : $Antena$)

desconectarTelefono

El primer impulso podría ser pedir como parámetros el teléfono y la antena. Pero... £es necesario pasarle la antena? Nuestro TAD debería conocer a qué antena está conectado cada teléfono en cada memento, no es razonable pedirle al usuario de nuestro TAD que nos lo diga él. Por lo tanto, nuestro proc desconectarTeléfono no tendrá un parámetro antena, tendrá sólo la red (inout) y el teléfono (in). No necesitamos valor de retorno.

`proc desconectarTelefono`(inout r : Red, in t : $Telefono$)

recorridoTelefono

El enunciado nos pide saber "el recorrido de un teléfono por las antenas de la red". Parece claro que el único parámetro que recibirá el proc (además de la red, en este caso como *in* pues no se debería modificar) será el teléfono. £Y qué devolvemos? Un recorrido tiene un orden y puede tener repetidos, por lo tanto la mejor opción es una secuencia de antenas, que va a representar las antenas por las que pasó ese teléfono.

`proc recorridoTelefono`(in r : Red, in t : $Telefono$) : seq$\langle Antena\rangle$

conexionesAntena

También nos piden saber "qué teléfonos están conectados a una antena en particular". Aquí entonces pediremos como parámetro una antena y devolveremos los teléfonos, y la mejor alternativa en este caso es un conjunto.

`proc conexionesAntena`(in $r : red$, in $a : Antena$) : conj$\langle Telefono \rangle$

1. b)

Leamos nuevamente el enunciado y analicemos qué cosas tiene que *recordar* el TAD para devolverle al usuario si lo pregunta o para restringir las operaciones.

- las antenas y los teléfonos
- las relaciones de vecindad entre las antenas
- a qué antena está conectado cada teléfono
- el recorrido de cada teléfono
- todos los teléfonos conectados a una antena

Hagamos una primera versión y después la vamos ajustando.
Para recordar las antenas y los teléfonos, podemos tener un conjunto de cada uno:

`obs` antenas : conj$\langle Antena \rangle$
`obs` teléfonos : conj$\langle Telefono \rangle$

Respecto de las relaciones de vecindad entre antenas, ya hablamos un poco antes. Podemos tener un dicc.

`obs` vecindad : dicc$\langle Antena,$ conj$\langle Antena \rangle \rangle$

Para saber a qué antena se conecta cada teléfono, también podríamos usar un dicc.

`obs` antenaDeTeléfono : dicc$\langle Telefono, Antena \rangle$

Y para el recorrido, una secuencia de antenas.

`obs` recorridoTeléfonos : dicc$\langle Telefono,$ seq$\langle Antena \rangle \rangle$

Finalmente, para saber qué teléfonos están conectados a una antena, también un dicc.

`obs` conexionesAntena : dicc$\langle Antena,$ conj$\langle Telefono \rangle \rangle$

Nos queda entonces algo así:

`obs` antenas : conj$\langle Antena \rangle$
`obs` teléfonos : conj$\langle Telefono \rangle$
`obs` vecindad : dicc$\langle Antena,$ conj$\langle Antena \rangle \rangle$
`obs` antenaDeTeléfono : dicc$\langle Telefono, Antena \rangle$
`obs` recorridoTeléfonos : dicc$\langle Telefono,$ seq$\langle Antena \rangle \rangle$
`obs` conexionesAntena : dicc$\langle Antena,$ conj$\langle Telefono \rangle \rangle$

Algo a tener en cuenta a la hora de elegir observadores es tratar de evitar la redundancia, que nos hará mucho más tediosa y confusa la especificación. Eso se logra intentando que la lista de observadores sea *minimal*: si se puede sacar un observador y aún así es posible seguir especificando todas las operaciones, entonces ese observador está de más y se debería eliminar. Analicemos nuestro caso:

- Según el enunciado, todas las antenas tienen vecinos. Entonces, el conjunto de antenas debería ser siempre igual al conjunto de claves de *vecindad*. Luego, el observador *antenas* está de más.

- Podemos asumir que todos los teléfonos están en *recorridoTeléfonos* (con una secuencia vacía si aún no fueron vistos). Luego, el observador *teléfonos* está de más.
- La antena de un teléfono en particular puede ser determinada por el observador conexionesAntena. Recuerden que acá no nos interesa la eficiencia (ya veremos eso en la segunda parte de la materia). Con que se pueda obtener el dato nos alcanza. Luego, el observador *antenaDeTeléfono* está de más.

Nos queda entonces la siguiente lista de observadores:

```
obs antenas : dicc⟨Antena, conj⟨Antena⟩⟩
obs recorridoTeléfonos : dicc⟨Telefono, seq⟨Antena⟩⟩
obs conexionesAntena : dicc⟨Antena, conj⟨Telefono⟩⟩
```

1. c)

Especifique completamente las operaciones **crearRed** y **conectarTelefono**.
En este punto escribiremos el *requiere* y *asegura* de las operaciones mencionadas. Lo mejor acá es descomponerlo en predicados y luego escribir cada uno de ellos.

Vamos primero con **crearRed**:
Como siempre, vuelta a leer el enunciado y a detectar requerimientos. Luego de un primer análisis nos quedó lo siguiente:

```
proc crearRed(in a : dicc⟨Antena, conj⟨Antena⟩⟩, in t : conj⟨Telefono⟩) : Red {
   requiere {
        noHayAntenasAisladas(a) ∧_L
        antenasEnSignificadosSonClaves(a) ∧
        vecinosEsMutuo(a)
   }
   asegura {
        res.antenas = a ∧ res.conexionesAntenas = ⟨⟩ ∧
        claves(res.recorridoTelefonos) = t ∧_L
        todosLosRecorridosSonVacios(res)
   }
}
```

y para **conectarTelefono**:

```
proc conectarTelefono(inout r : Red, in t : Telefono, in a : Antena) {
   requiere {
        r = R_0 ∧
        (t ∈ claves(r.recorridosTelefonos) ∧ a ∈ claves(r.conexionesAntenas)) ∧_L
             (¬estáConectado(r, t) ∨ conectaAVecina(r, t, a))
   }
   asegura {
        r.antenas = R_0.antenas ∧_L
        (
             ¬estáConectado(R_0, t) ∧
             conectadoA(R_0, r, t, a) ∧
             lasDemasAntenasNoCambian(R_0, r, t, ⟨a⟩)
        ) ∨ (
             estáConectado(R_0, t) ∧
             desconectadoDeAnterior(R_0, r, t) ∧
             conectadoA(R_0, r, t, a) ∧
             lasDemasAntenasNoCambian(R_0, r, t, ⟨a, ultimo(R_0.recorridoTelefonos[t])⟩)
        ) ∧
        r.recorridosTelefonos = setAt(R_0.recorridosTelefonos, t, R_0.recorridosTelefonos[t] ++ ⟨a⟩)
   }
}
```

Un par de observaciones:

- Recuerden especificar también, en el asegura, todo lo que **no** cambia, tanto en los observadores (r.antenas = R_0.antenas) como adentro de las estructuras (*lasDemasNoCambian*)
- recuerden que la operación *setKey* devuelve un dicc que es igual al pasado como parámetro, salvo que tiene una clave cambiada. Si no están seguros de cómo usarlo, siempre se puede lo mismo usando un predicado con cuantificadores (pueden hacerlo como ejercicio)

1. d)

- **desconectarTelefono** debe eliminar al teléfono pasado como parámetro del conjunto de telefonos en el que se encuentra en el observador *conexionesAntenas*
- **recorridoTelefono** debe devolver la secuencia asociada al teléfono pasado como parámetro en el observador *recorridoTeléfonos*
- **conexionesAntena** debe devolver el conjunto de teléfonos asociado a la antena pasada como parámetro en el observador *conexionesAntenas*

2.

Para que se puedan agregar antenas después de iniciado el sistem, lo primero que se debería tener es un proc *agregarAntena*. Debería tomar como parámetros la red (inout), la nueva antena (in) y el conjunto de antenas vecinas (in). Esa operación debería agregar la nueva antena con sus vecinas al observador *antenas* y eventualmente (depende de cómo modelemos las operaciones) agregar la nueva antena al observador *conexionesAntenas* con un conjunto vacío asociado.

No es necesario cambiar la igualdad, no es necesario cambiar los requiere y aseguda del resto de las operaciones ni tampoco cambiar los observadores.

TAD completo

Antena, Telefono ES $\mathbb{Z}$

TAD Red {

obs antenas : dicc$\langle Antena, \text{conj}\langle Antena\rangle\rangle$ // *Antenas y sus vecinas*

obs recorridoTelefonos : dicc$\langle Telefono, \text{seq}\langle Antena\rangle\rangle$ // *Teléfonos y sus recorridos*

obs conexionesAntenas : dicc$\langle Antena, \text{conj}\langle Telefono\rangle\rangle$ // *Antenas y los teléfonos que tiene conectados*

proc crearRed(in dicc$\langle Antena, \text{conj}\langle Antena\rangle\rangle$: $antenas$, in conj$\langle Telefono\rangle$: $telefonos$) : Red {

requiere {

$noHayAntenasAisladas(antenas) \wedge_L$

$antenasEnSignificadosSonClaves(antenas) \wedge$

$vecinosEsMutuo(antenas)$

}

asegura {

$res.antenas = antenas \wedge res.conexionesAntenas = \langle\rangle \wedge$

claves$(res.recorridoTelefonos) = telefonos \wedge_L$

$todosLosRecorridosSonVacios(res)$

}

}

pred noHayAntenasAisladas(dicc$\langle Antena, \text{conj}\langle Antena\rangle\rangle$: $antenas$) {

$(\forall a : Antena)\ (a \in \text{claves}(antenas) \rightarrow_L |antenas[a]| > 0)$

}

pred antenasEnSignificadosSonClaves(dicc$\langle Antena, \text{conj}\langle Antena\rangle\rangle$: $antenas$) {

$(\forall a : Antena)\ (a \in \text{claves}(antenas) \rightarrow_L$

$(\forall a' : Antena)\ (a' \in antenas[a] \rightarrow_L (a' \neq a \wedge a' \in \text{claves}(antenas))))$

}

pred vecinosEsMutuo(dicc$\langle Antena, \text{conj}\langle Antena\rangle\rangle$: $antenas$) {

$(\forall a, a' : Antena)\ ((a \in \text{claves}(antenas) \wedge a' \in \text{claves}(antenas)) \rightarrow_L a \in antenas[a'] \leftrightarrow a' \in antenas[a])$

}

pred todosLosRecorridosSonVacíos(res : Red) {

$(\forall t : Telefono)\ (t \in \text{claves}(res.recorridosTelefonos) \rightarrow_L res.recorridosTelefonos[res] = \langle\rangle)$

}

}

```
proc igualdad(in R_1, R_2 : Red) : bool {
    asegura {
        res = true ↔
            R_1.antenas = R_2.antenas ∧
            R_1.recorridosTelefonos = R_2.recorridosTelefonos ∧
            R_1.conexionesAntenas = R_2.conexionesAntenas
    }
}

proc conectarTelefono(inout r : Red, in t : Telefono, in a : Antena) {
    requiere {
        r = R_0 ∧
        (t ∈ claves(r.recorridosTelefonos) ∧ a ∈ claves(r.conexionesAntenas)) ∧_L
            (¬estáConectado(r, t) ∨ conectaAVecina(r, t, a))
    }
    asegura {
        r.antenas = R_0.antenas ∧_L
        (
            ¬estáConectado(R_0, t) ∧
            conectadoA(R_0, r, t, a) ∧
            lasDemasAntenasNoCambian(R_0, r, t, ⟨a⟩)
        ) ∨ (
            estáConectado(R_0, t) ∧
            desconectadoDeAnterior(R_0, r, t) ∧
            conectadoA(R_0, r, t, a) ∧
            lasDemasAntenasNoCambian(R_0, r, t, ⟨a, ultimo(R_0.recorridoTelefonos[t])⟩)
        ) ∧
        r.recorridosTelefonos = setAt(R_0.recorridosTelefonos, t, R_0.recorridosTelefonos[t] ++ ⟨a⟩)
    }
}

pred estáConectado(r : Red, t : Telefono) {
    (∃a : antena) (a ∈ claves(r.conexionesAntenas) ∧ t ∈ r.conexiones[a])
}

pred conectaAVecina(r : Red, t : Telefono, a : Antena) {
    a ∈ r.antenas[ultimo(r.recorridosTelefonos[t])]
}

pred desconectadoDeAnterior(R_0, r : Red, t : Telefono) {
    r.conexiones[ultimo(R_0.recorridoTelefonos[t])] = R_0.conexiones[ultimo(R_0.recorridoTelefonos[t])] − ⟨t⟩
}

pred conectadoA(R_0, r : Red, t : Telefono, a : Antena) {
    r.conexionesAntenas[a] = R_0.conexionesAntenas[a] ∪ ⟨t⟩
}

pred lasDemasAntenasNoCambian(R_0, r : Red, t : Telefono, antenas : conj⟨Antena⟩) {
    (∀a′ : Antena) ((a′ ∈ r.antenas ∧ a′ ∉ antenas) →_L
        (a′ ∈ claves(r.conexionesAntenas) ∧_L r.conexionesAntenas[a′] = R_0.conexionesAntenas[a′]))
}

aux ultimo(s : seq⟨T⟩) : T = s[|s| − 1]
```

```
    proc desconectarTelefono(inout r : Red, in t : Telefono) {
        requiere { r = R_0 ∧ estáConectado(r, t) }
        asegura {
            r.antenas = R_0.antenas ∧
            r.recorridosTelefonos = R_0.recorridosTelefonos ∧
            desconectadoDeAnterior(R_0, r, t) ∧
            lasDemasAntenasNoCambian(R_0, r, t, ⟨ultimo(R_0.recorridoTelefonos[t])⟩)
        }
    }

    proc recorridoTelefono(in r : Red, in t : Telefono) : seq⟨Antena⟩ {
        requiere { t ∈ claves(r.recorridosTelefonos) }
        asegura { res = r.recorridosTelefonos[t] }
    }

    proc conexionesAntena(in r : red, in  : a : Antena) : conj⟨Telefono⟩ {
        requiere { a ∈ claves(r.conexionesAntenas) }
        asegura { res = r.conexionesantenas[a] }
    }
}
```

Ejercicio del 1er parcial del 1er cuatrimestre 2025

Enunciado

La Panadería *El Progreso* recibe todos los días panes, facturas, masas, y otras delicias para venderla a sus clientes. Una vez que cierra sus puertas todo lo que no se vendió se envía a un comedero del barrio, por lo que todos los días se empieza con un stock completamente nuevo.

Los clientes son atendidos por estricto orden de llegada y realizan su pedido a algún empleado libre. Si no hay mercadería suficiente disponible para cumplir su pedido se retiran con las manos vacías. Caso contrario se le entrega y luego esperan para pagar, también en orden de llegada. Una vez que pagaron se retiran a disfrutar sus panificados.

Nos piden modelar en un TAD el funcionamiento de El Progreso según esta descripción, teniendo en cuenta que nos importa en todo momento saber cuántos clientes están esperando que los atiendan y cuántos están esperando para pagar.

a) Indique las operaciones (procs) del TAD con todos sus parámetros y los renombres de tipo que considere necesarios.

b) Describa el TAD en forma completa, indicando sus observadores, los requiere y asegura de las operaciones. Puede agregar los predicados y funciones auxiliares que necesite, con su correspondiente definición.

c) Cuando presentamos nuestro tad nos dicen que ahora quieren saber qué empleado atiende más clientes cada día. ¿Debería modificar su TAD para reflejar esto? ¿Cómo? Responda en palabras, en forma breve y precisa.

Desarrollo

Para este ejercicio, en vez de entrar en el detalle de cada paso de la resolución, queremos entrar más en detalles del modelado. Para esto es importante que antes de seguir leyendo traten de pensar el ejercicio por su cuenta y luego lo comparen con la solución que vamos a proponer. Como ya dijimos, esta solución no es la única posible, y vamos a mencionar algunas diferencias que son aceptables, pero también algunas que no.

El modelo del TAD son sus observadores y la interfaz, es decir, los `proc` que hay que especificar. Como metodología para encarar la resolución de un problema con TADs les proponemos empexar por los procedimientos y luego pensar los observadores. En este caso tendríamos:

Cliente, Mercadería, Cantidad ES $\mathbb{Z}$
`proc abrir`(in *stock* : dicc$\langle Mercadería, Cantidad\rangle$) : *Panadería*
`proc llegaCliente`(inout p : *Panadería*, in c : *Cliente*)
`proc atenderCliente`(inout p : *Panadería*, in m : *Mercadería*, in c : *Cantidad*)
`proc cobrarACliente`(inout p : *Panadería*)
`proc clientesEsperando`(in p : *Panadería*) : $\mathbb{Z}$
`proc clientesACobrar`(in p : *Panadería*) : $\mathbb{Z}$

Leer estos `proc` nos dice cómo estamos interpretando el enunciado:

- **abrir** recibe como parámetro todo el stock del día de operación, es decir, la panadería abre con su stock "adentro". Algunes definieron un procedimiento para "crear" la Panadería, y luego dos procedimientos para abrir y cerrar cada día, donde el procedimiento de abrir cada día es el que recibe el stock y el de cerrar lo vacía. Esto **es válido pero:**
  - No es lo que se pide, sólo nos interesa modelar cada día de operación. Esto pued no ser súper claro en el enunciado, y es el tipo de ambigüedades que tratamos de evitar, pero además
  - No hay garantía de que los procedimientos se llamen en un orden. Se puede poner un observador booleano *estáAbierto* y ponerlo como requerimiento para el resto de las operaciones, pero en general es buena práctica pensar nuestros modelos asumiendo que los procedimientos pueden llamarse siempre todos en cualquier orden
- **llegaCliente** sólo recibe como parámetro al cliente. Acá aparecieron algunas alternativas:
  - Poner como parámetro de entrada al cliente y el pedido que va a hacer. Esto **es válido**, el enunciado dice que los clientes hacen su pedido, pero no aclara cuándo "decide" qué comprar. En la solución nuestra el pedido aparece recién en el momento de atender por dos razones:
    - El autor del ejercicio jamás entró a una panadería sabiendo exactamente qué iba a comprar hasta el último momento.
    - Más objetivamente, es una buena práctica tratar de que las operaciones de un TAD sean lo más *atómicas* posible, es decir, que en la medida de lo posible hagan una sola cosa
  - Modelar al cliente como un struct $\langle \mathbb{Z}, \text{dicc}\langle Mercadería, Cantidad\rangle\rangle$, es decir, con un identificador y su pedido. Esto **está mal**:
    - A nivel mecánico, con ese modelo se puede tener a la misma persona dos veces en la cola, con distintos pedidos

- Si bien la primer objeción se puede "solucionar" en la especificación, esto la hace más compleja (lo que la hace más difícil de leer y da más margen a errores), pero más importante aún
- le estaríammos dando a la "entidad" cliente más información y responsabilidades de lo que necesita tener

- **atenderCliente** es el que recibe el pedido, siguiendo lo discutido para la operación anterior. Notemos que:
  - El cliente no es parámetro de entrada. Ponerlo **está mal**, ya que los clientes al llegar se ponen en la cola y se atiende al siguiente.
  - Tampoco es parametro de entrada el *empleado* que lo atiende. Modelar los empleados también **está mal**. Se los menciona en el ejercicio para evaluar que sepan distinguir elementos del problema que no es necesario modelar. El enunciado dice "algún empleado libre", pero no dice que nos interese saber quién fue que atendió.
- **cobrarACliente** no recibe parámetros más allá del TAD. El cliente debería ser el siguiente a cobrar.
- **clientesEsperando** y **clientesACobrar** responden directamente al último párrafo del enunciado. Presten atención a este tipo de requerimietnos del enunciado porque están pensados para guíarlos en cómo deberían observar el problema en su TAD.

A partir de estos procedimientos resulta razonable imaginar que los observadores sean

```
obs stock : dicc⟨Mercadería, Cantidad⟩
obs clientesEsperando : seq⟨Cliente⟩
obs clientesACobrar : seq⟨Cliente⟩
```

Es decir que tenemos:

- El stock del día, tal como entra en el procedimiento inicial, que debemos mantener durante el día (reducirlo cuando se venden productos)
- La cola de clientes que están esperando para hacer un pedido
- La cola de clientes que están esperando para pagar

Si en el modelo ingresamos al cliente con su pedido (y por lo tanto el pedido no es parámetro de atenderCliente), necesitamos un observador más: un diccionario que para cada cliente tenga su pedido, y referir a él en el momento de hacer las cuentas.

Con esto tenemos un modelo completo, pero aún quedan cosas sin definir, principalmente qué pasa si no hay stock de la mercadería que quiere el cliente. Esa decisión se va a ver refeljada únicamente en la especificación. Nosotros decidimos permitir atender clientes que tengan pedidos que no se pueden cumplir, en cuyo caso salen de la cola para pedir pero, obviamente, no pasan a la cola para pagar. Otra especificación posible es poner como requerimiento de atenderCliente que el pedido sea válido, pero esto restringe más que lo que el enunciado define.

TAD completo (incisos a) y b) del ejercicio)

```
TAD Panadería {
    obs stock : dicc⟨Mercadería, Cantidad⟩
    obs clientesEsperando : seq⟨Cliente⟩
    obs clientesACobrar : seq⟨Cliente⟩

    proc abrir(in stock : dicc⟨Mercadería, Cantidad⟩) : Panadería {
        requiere { cantidadesMayoresACero(stock) }
        asegura {
            res.stock = stock ∧
            res.clientesEsperando = ⟨⟩ ∧
            res.clientesACobrar = ⟨⟩
        }
    }

    pred cantidadesMayoresACero(stock : dicc⟨Mercadería, Cantidad⟩) {
        (∀m : Mercadería) (m ∈ claves(stock) →_L stock[m] > 0)
    }
```

```
proc igualdad(in P1 : Panadera, in P2 : Panadera) : bool {
    asegura { res = true ↔
        P1.stock = P2.stock ∧
        P1.clientesEsperando = P2.clientesEsperando ∧
        P1.clientesACobrar = P2.clientesACobrar }
}

proc llegaCliente(inout p : Panadería, in c : Cliente) {
    requiere { p = P0 ∧ clienteNuevo(c, p) }
    asegura {
        p.stock = P0.stock ∧
        p.clientesACobrar = P0.clientesACobrar ∧
        p.clientesEsperando = P0.clientesEsperando ++ ⟨c⟩
    }
}

pred clienteNuevo(p : Panadería, c : Cliente) {
    c ∉ p.clientesEsperando ∧ c ∉ p.clientesACobrar
}

proc atenderCliente(inout p : Panadería, in m : Mercadería, in c : Cantidad) {
    requiere { p = P0 ∧ hayClienteEsperando(p) ∧ existeProducto(p, m) }
    asegura {
        p.clientesEsperando = tail(P0.clientesEsperando) ∧
        (
            haySuficiente(P0, m, c) ∧
            p.stock = setKey(P0.stock, p, P0.stock[m] − c) ∧
            p.clientesACobrar = P0.clientesACobrar ++ ⟨head(P0.clientesEsperando)⟩
        )∨(
            ¬haySuficiente(P0, m, c) ∧
            p.stock = P0.stock
            p.clientesACobrar = P0.clientesACobrar
        )
    }
}

pred hayClienteEsperando(p : Panadería) { |p.clientesEsperando| > 0 }
pred existeProducto(p : Panadería, m : Mercadería) { m ∈ p.stock }
pred haySuficiente(p : Panadería, m : Mercadería, c : Cantidad) { m ∈ p.stock ∧L p.stock[m] ≥ c }

proc cobrarACliente(inout p : Panadería) {
    requiere { p = P0 ∧ |p.clientesACobrar| > 0 }
    asegura {
        p.stock = P0.stock ∧
        p.clientesEsperando = P0.clientesEsperando ∧
        p.clientesACobrar = tail(P0.clientesACobrar)
    }
}

proc clientesEsperando(in p : Panadería) : ℤ {
    asegura {
        res = |p.clientesEsperando|
    }
}

proc clientesACobrar(in p : Panadería) : ℤ {
    asegura {
        res = |p.clientesACobrar|
    }
}
}
```

Inciso c)

Será necesario:

- Agregar como observador un diccionario de Empleado a la cantidad de clientes que atiende
- Agregar el conjunto de empleados al `proc abrir`()
- Agregar el empleado como parámetro `in` en el `proc atender`()
- Sumar uno al valor correspondiente al empleado en el diccionario, en la especificación de esa operación