Algoritmos y Estructuras de Datos II

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

Trabajo práctico 2: Diseño

Lollapatuza

alias: JUSCIKTYMTQNAEURXVZQ

Integrante	LU	Correo electrónico
Bustos, Juan	19/22	juani8.bustos@gmail.com
Dominguez, Leonardo	285/22	leodomingue2016@gmail.com
Nandín, Matías	227/22	imatinandin@gmail.com
Marín, Candela Emilia	1405/21	canmarin17@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

TAD stock es Nat
TAD descuentos es nat

1. Módulo Puesto de Comida

Interfaz

se explica con: TAD Puesto de Comida

géneros: `puesto`

Servicios Usados:
DiccLog :

Vacío() → res : dicc(κ, σ)
Complejidad: $\theta(1)$
Descripción: Genera un diccionario vacío.

Definir(in/out d : dicc(κ, σ), in k : κ, in s : σ) → res : itDicc(κ, σ)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)) + copy(k) + copy(s))$, donde K = claves(d)
Descripción: Define la clave k con el significado s en el diccionario. Retorna un iterador al elemento recién agregado.
Aliasing: Los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

Definido?(in d : dicc(κ, σ), in k : κ) → res : bool
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: Devuelve true si y sólo k está definido en el diccionario.

Significado(in d : dicc(κ, σ), in k : κ) → res : σ
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: Devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

Borrar(in/out d : dicc(κ, σ), in k : κ)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: Elimina la clave k y su significado de d.

Copiar(in d : dicc(κ, σ)) → res : dicc(κ, σ)
Complejidad: $\theta(\sum_{k' \in K}(copy(k) + copy(significado(k,d))))$, donde K = claves(d)
Descripción: Genera una copia nueva del diccionario.

Lista Enlazada :
Vacía() → res : lista(α)
Complejidad : $\theta(1)$
Descripción: Genera una lista vacìa

AgregarAtras(in/out l : lista(α), *in* $a : \alpha$) → res : $itLista(\alpha)$
Complejidad : $\theta(copy(a))$
Descripción: Agrega el elemento a como ultimo elemento de la lista. Retorna un iterador a l, de forma tal que Siguiente devuelva a.

Primero(in l : lista(α)) → $res : \alpha$
Complejidad : $\theta(1)$
Descripción: Devuelve el primer elemento de la lista.
Aliasing: res es modificable si y solo si l es modificable.

Fin(in/out l : lista(α))
Complejidad : $\theta(1)$
Descripción: Elimina el primero elemento de l

Longitud(in l : lista(α)) $\rightarrow$ *res* : *nat*
Complejidad : $\theta(copy(a))$
Descripción: Devuelve la cantidad de elementos que tiene la lista

Copiar(in l : lista(α)) $\rightarrow$ *res* : *lista*(α)
Complejidad : $\theta(\sum_{i=1}^{t} copy(l[i]))$, donde t = long(l).
Descripción: Genera una copia nueva de la lista

Vector :
Vacía() $\rightarrow$ res : vector(α)
Complejidad : $\theta(1)$
Descripción: Genera un vector vacìo

Longitud(in v : vector(α)) $\rightarrow$ *res* : *nat*
Complejidad : $\theta(1)$
Descripción: Devuelve la cantidad de elementos que contiene el vector

Copiar(in v : vector(α)) $\rightarrow$ *res* : *vector*(α)
Complejidad : $\theta(\sum_{i=1}^{l} copy(v[i]))$, donde l = long(v).
Descripción: Genera una copia nueva del vector

AgregarAtras(in/out v : vector(α), *in a* : α)
Complejidad : $\theta(f(long(v)) + copy(\alpha))$.
Descripción: Agrega el elemento a como el ultimo elemento del vector.
Aliasing: El elemento a se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando long(v) es potencia de 2.

•[•](in v : vector(α), *ini* : *nat*) $\rightarrow$ *res* : α
Complejidad : $\theta(1)$.
Descripción: Devuelve el elemento que se encuentra en la i- esima posicion del vector en base 0. Es decir, v[i] devuelve el elemento que se encuentra en la posicion i + 1.
Aliasing: res es modificable si y solo si v es modificable.

Operaciones básicas de Puesto de Comida

STOCK(**in** p : `puesto`, **in** i : `item`) $\rightarrow$ *res* : `stock`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {res $=_{obs}$ Stock(p,i)}
Complejidad: $\Theta(\log(I))$
Descripción: Devuelve el stock de un ítem
Aliasing: Devuelve una referencia del significado del diccionario DiccStock

DESCUENTO(**in** p : `puesto`, **in** i : `item`, **in** *cant* : `nat`) $\rightarrow$ *res* : `descuentos`
Pre $\equiv$ {cant $\geq 0 \wedge$ i $\in$ menu(p)}
Post $\equiv$ {res $=_{obs}$ Descuento(p,i,cant)}
Complejidad: $\Theta(\log(I))$
Descripción: Dado un puesto, item y cantidad, retorna la cantidad de descuento a aplicarle
Aliasing: res es modificable si y solo si ListaDescuento es modificable

NuevoPuesto(**in** *dPrecio*: `dicc(item, precio)`, **in** *dDesc*: `dicc(item, dicc(cantidad, nat))`, **in** *dStock*: `dicc(item, cantidad)`) $\rightarrow$ *res* : `puesto`
Pre $\equiv$ {claves(*dPrecio*) $=_{\text{obs}}$ claves(*dStock*) $\wedge$ claves(*dDesc*) $\subseteq$ claves(*dPrecio*))}
Post $\equiv$ {*res* $=_{\text{obs}}$ crearPuesto(*dPrecio*, *dStock*, *dDesc*)}
Complejidad: $\Theta(\text{I} + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$
Descripción: Crea un puesto de comida dado un menú, una lista de descuentos y un stock
Aliasing: No hay aliasing ya que se crea por copia cada una de las estructuras

GastosDe(**in** *p*: `puesto`, **in** *a*: `persona`) $\rightarrow$ *res* : `dinero`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ gastosDe(p,a)}
Complejidad: $\Theta(\log(A))$
Descripción: Dado un puesto y un cliente, informa el gasto de dicho cliente en ese puesto
Aliasing: res es modificable si y solo si diccVentas es modificable

Hackeo(**in/out** *p*: `puesto`, **in** *i*: `item`, **in** *a*: `persona`)
Pre $\equiv$ {p $=_{\text{obs}}$ p_0 $\wedge$ i $\in$ menu(p) $\wedge_{\text{L}}$ consumioSinPromo?(p,i,a)}
Post $\equiv$ {p $=_{\text{obs}}$ OlvidarItem(p_0, i, a)}
Complejidad: $\Theta(\log(I) + \log(A))$
Descripción: Elimina del historial de la persona una venta hackeable (no tiene promo) que incluya al item en dicho puesto. Además se actualiza el stock, deshaciendo la venta.
Aliasing: No aplica

CalcularCostoVenta(**in** *p*: `puesto`, **in** *i*: `item`, **in** *cant*: `nat`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {i $\in$ menu(p) $\wedge$ cant > 0}
Post $\equiv$ {res $=_{\text{obs}}$ gastosDeVenta(p, <i,cant>)}
Complejidad: $\Theta(log(I))$
Descripción: Dada una compra calcula el precio de ésta, aplicando el descuento correspondiente si es que tiene
Aliasing: No aplica

RegistrarVenta(**in/out** *p*: `puesto`, **in** *i*: `item`, **in** *a*: `persona`, **in** *cant*: `nat`)
Pre $\equiv$ { p $=_{\text{obs}}$ p_0 $\wedge$ i $\in$ menu(p) $\wedge$ cant $\leq$ stock(p, i)}
Post $\equiv$ { p $=_{\text{obs}}$ ventas(p_0,a, i, cant) }
Complejidad: $\Theta(\log(I) + \log(A))$
Descripción: Registra una venta, actualizando el gasto de la persona, el historial de ventas y el stock del item en el puesto.
Aliasing: No aplica

VentaSinPromo?(**in** *p*: `puesto`, **in** *i*: `item`, **in** *cant*: `nat`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {i $\in$ menu(p) $\wedge$ cant > 0 }
Post $\equiv$ {res $=_{\text{obs}}$ (Descuento(p,i,cant) $= 0$?)}
Complejidad: $\Theta(\log(I))$
Descripción: Dada una venta en el puesto de comida, verifica si hubo o no descuento.
Aliasing: No aplica

esHackeable?(**in** *p*: `puesto`, **in** *i*: `item`, **in** *a*: `persona`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {res $=_{\text{obs}}$ consumioSinPromo?(p,a,i)}
Complejidad: $\Theta(\log(A) + \log(I))$
Descripción: Dado un item y una persona, devuelve true si ésta realizó una compra sin promo del item
Aliasing: No aplica

copiar(**in** *p*: `puesto`) $\rightarrow$ *res* : `puesto`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ p}
Complejidad: $\Theta(I + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item) + A * I * CantidadVentasActuales)$
Descripción: Dado un puesto, devuelve una copia del mismo
Aliasing: La versión copiada no tiene aliasing con ninguna de las estructuras originales

Especificación de las operaciones auxiliares utilizadas en la representación

TAD Puesto De Comida Extendido

otras operaciones

Suma : secu(<item×cant>) × puesto ⟶ Nat

GastoVenta : estr e × item i × cant c ⟶ Nat {i ∈ menu(e) ∧ c >0}

Precio : item i × estr e ⟶ nat {i ∈ menu(e)}

EncontrarDescuento : estr e × item i × cant c ⟶ nat {i ∈ menu(e) ∧ c >0}

ItemsDentro? : secu(<item×cant>) × Conj(item) ⟶ Bool

EsHackeoSubDeVenta : secu(itLista) × secu(<item×cant>) ⟶ Bool

VentaDePersonaItem? : secu(itLista) × item × persona × estr e ⟶ Bool {i ∈ menu(e)}

SonHackeables? : secu(itLista) × estr ⟶ Bool

axiomas

Suma(hist, e) ≡ **if** vacia?(hist) **then**
0
else
GastoVenta(π_1(prim(hist)), π_2(prim(hist)), e) + Suma(Fin(hist), e)
fi

Precio(i,e) ≡ Obtener(i, e.diccPrecio)

GastoVenta(e, i, c) ≡ AplicarDescuento(c x Precio(i, e), EncontrarDescuento(e, i, c))

EncontrarDescuento(e, i, c) ≡ **if** Obtener(i, e.diccDescuentos).max ≥ c **then**
Obtener(i, e.diccDescuentos).listaDescuentos[c]
else
Obtener(i,e.diccDescuentos).listaDescuentos[Obtener(i,e.diccDescuentos).max]
fi

ItemsDentro?(hist, items) ≡ **if** vacia?(hist) **then**
true
else
if π_1(Prim(hist)) ∈ items **then** ItemsDentro?(fin(hist), items) **else** false **fi**
fi

EsHackeoSubDeVenta(vHackeables, ventas) ≡ **if** vacia?(vHackeables) **then**
true
else
if esta?(Siguiente(Prim(vHackebles)), ventas) **then**
EsHackeoSubDeVenta(Fin(vHackeables), ventas)
else
false
fi
fi

VentaDePersonaItem?(vHackeables, i, p, e) ≡ **if** Vacia?(vHackeables) **then**
true
else
if π_1(Siguiente(Prim(vHackebles))) = i **then**
if Esta?(Siguiente(Prim(vHackebles)),
Obtener(p, e.diccVentas).historial) **then**
VentaDePersonaItem?(Fin(vHackeables), i, p, e)
else
false
fi
else
false
fi
fi

SonHackeables?(vHackeables, e) $\equiv$ **if** Vacia?(vHackeables) **then**
true
else
if (EncontrarDescuento(e, π_1(Siguiente(Prim(vHackebles))), π_2(Siguiente(Prim(vHackebles)))) = 0) **then**
SonHackeables?(Fin(vHackeables), e)
else
false
fi
fi

Fin TAD

Representación

Representación de puesto de comida

`Puesto` **se representa con** `estr`

donde `estr` es `tupla`(*diccPrecio*: `diccLog(item, precio)`
, *diccStock*: `diccLog(item, cantidad)`
, *diccDescuentos*: `diccLog(Item, InfoDescuento)`
, *diccVentas*: `diccLog(persona,registroVentasPersona)`
, *diccHackeos*: `diccLog(persona, diccLog(item, sec(itLista)))`
)

diccPrecio: Es un diccionario logarítmico que representa al menú del puesto. Éste asocia a cada item con su precio.
diccStock: Es un diccionario logarítmico que representa al stock del puesto. Éste asocia a cada item con su stock actual.
diccDescuentos: Es un diccionario logarítmico que representa a los descuentos del puesto. Éste asocia a cada item con descuento con los diferentes porcentajes que podría tener según su cantidad.
diccVentas: Es un diccionario logarítmico que asocia a las personas con su registro de venta.
diccHackeos: Es un diccionario logarítmico que relaciona a una persona con todas las ventas hackeables de un item específico. La secuencia se utiliza para guardar iteradores apuntando a las ventas hackeables dentro de diccVentas.

donde `InfoDescuento` es `tupla`(*max*: `nat`
, *listaDescuento*: `vector(max+1)`
)
max: Es un natural que representa la cantidad mínima a comprar de un item para conseguir el mayor porcentaje de descuento en dicho puesto.
listaDescuento: Es un vector donde cada posición representa una cantidad a comprar del item y devuelve el procentaje de descuento apropiado. Su longitud es igual a la cantidad mínima a comprar de un item para conseguir el mayor descuento + 1.

donde `registroVentasPersona` es `tupla`(*total*: `nat`
, *historial*: `lista(tupla(item,cant))`
)

total: Es un natural que representa el gasto total realizado por una persona en el puesto.
historial: Es una lista que representa el historial de venta de una persona. Contiene todas las ventas en forma de tupla, con su respectivo item y cantidad.

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$

//Deben estar los mismos items tanto en el stock como en el menu de precios
$Claves(e.diccPrecio) = Claves(e.diccStock) \wedge$

//Los items que tienen descuentos deben estar dentro del menu
$Claves(e.diccDescuentos) \subseteq Claves(e.diccPrecio) \wedge$

//Si un item tiene descuento, entonces la lista de descuento debe existir y tener un tamaño igual a la maxima cantidad de Descuento + 1. Además en dicha lista la cantidad 0 no tiene descuento
$(\forall\ i : item)(def?(i, e.diccDescuentos) \Rightarrow_{\text{L}} (long(obtener(i, e.diccDescuentos).listaDescuento) = obtener(e.diccDescuentos, i).max + 1 \wedge$
$obtener(i, e.diccDescuentos).listaDescuento[0] = 0)) \wedge_{\text{L}}$

//Todos los valores que corresponden al descuento de un item deben encontrarse entre 1 y 100
$(\forall\ i : item)(def?(i, e.diccDescuentos) \Rightarrow_{\text{L}} ((\forall\ cant : nat)(0 \leq cant < obtener(i, e.diccDescuentos).max + 1 \Rightarrow_{\text{L}} 0 \leq obtener(i, e.diccDescuentos).listaDescuento[cant] < 100)) \wedge$

//La sumatoria de las compras de una persona debe ser igual al total dentro de DiccVentas
$(\forall\ a : persona)(def?(a, e.diccVentas) \Rightarrow_{\text{L}} \pi_1(obtener(a, e.diccVentas)) = Suma(\pi_2(obtener(p, e.diccVentas), e)) \wedge$

//Todos los items comprados por una persona deben corresponder a los productos dentro del menu
$(\forall\ a : persona)(def?(a, e.diccVentas) \Rightarrow_{\text{L}} ItemsDentro?(obtener(a, e.diccVentas).historial, claves(e.diccStock)$
$\wedge$

// Si una persona puede ser hackeada, entonces tambien debe tener compras registradas
$(\forall\ a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} Def?(a, e.diccVentas)) \wedge$

//Todas las ventas Hackeables de una persona deben pertenecer a sus ventas totales
$(\forall\ a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} ((\forall\ i : item)((def?(i, obtener(a, e.diccHackeos)))) \Rightarrow_{\text{L}} EsHackeoSubDeVenta?(obtener(i, obtener(a, e.diccHackeos)), obtener(a, e.diccVentas).historial))) \wedge$

//Todas las compras hackeables de una persona e item determinado deben pertenecer a dicha persona e item
$(\forall a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} ((\forall\ i : item)((def?(i, obtener(a, e.diccHackeos)))) \Rightarrow_{\text{L}} VentaDePersonaItem?(obtener(i, obtener(a, e.diccHackeos)), a, i, e)) \wedge$

//Cada item hackeable deben ser verdaderamente hackeable (No tiene promocion)
$(\forall\ a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} ((\forall\ i : item)(def?(i, obtener(a, e.diccHackeos))) \Rightarrow_{\text{L}} SonHackeables?(obtener(i, obtener(p, e.diccHackeos)), e))$

Abs : estr e $\longrightarrow$ Puesto {Rep(e)}

Abs(e) $=_{\text{obs}}$ p: Puesto |

$menu(p) =_{\text{obs}} Claves(e.diccPrecio) \wedge_{\text{L}}$
$(\forall\ i : item)(i \in Claves(e.diccPrecio) \Rightarrow_{\text{L}}$
$(Precio(p, i) =_{\text{obs}} obtener(i, e.diccPrecio) \wedge$
$Stock(p, i) =_{\text{obs}} obtener(i, e.diccStock) \wedge$
$(\forall\ cant : nat)$
$(descuento(p, i, c) =_{\text{obs}} EncontrarDescuento(e, i, c)))\wedge_{\text{L}}$
$(\forall\ a : persona)(a \in Claves(e.diccVentas) \Rightarrow_{\text{L}}$
$ventas(p, a) =_{\text{obs}} listaAMulticonj(Obtener(p, e.diccVentas).historial))$

Algoritmos

Algoritmos del módulo Puesto De Comida

---

iStock(**in** p: `puesto`, **in** i: `item`) $\rightarrow$ res : stock

1: $res \leftarrow Significado(p.diccStock, i)$ ▷ $\Theta(Log(I))$

Complejidad: $\Theta(log(I))$

---

iDescuento(**in** p: `puesto`, **in** i: `item`, **in** $cant$: `nat`) $\rightarrow$ res : descuentos

1: // Verifico que el item tenga algun descuento registrado
2: **if** ($Definido?(p.diccDescuentos, i)$) **then** ▷ $\Theta(Log(I))$
3:
4: // DescuentoInfo.listaDescuento tiene guardado en cada posición el porcentaje de descuento
5: // Asignado a esa cantidad
6: // Si la cantidad es mayor al máximo, devolvemos el descuento más grande
7: $descuentoInfo \leftarrow Significado(p.diccDescuentos, i)$ ▷ $\Theta(Log(I))$
8: **if** ($cant > descuentoInfo.max$)**then** ▷ $\Theta(1)$
9: $res \leftarrow descuentoInfo.listaDescuento[descuentoInfo.max]$ ▷ $\Theta(1)$
10: **else**
11: $res \leftarrow descuentoInfo.listaDescuento[cant]$ ▷ $\Theta(1)$
12: **else**
13: $res \leftarrow 0$ ▷ $\Theta(1)$

Complejidad: $\Theta(log(I))$
Justificación: Primero verificamos si el item tiene un descuento registrado, buscándolo en DiccDescuento. En caso de que no sea asi, devolvemos directamente cero. Esta búsqueda tiene una complejidad de $\Theta(log(I))$ por ser un diccionario logarítmico.

Si hay un descuento registrad se accede a su información, en la cual se encuentra la cantidad mínima para obtener el mayor descuento y el vector de descuentos. Esta operación tardará Θ(Log(I)).

Por último, verificamos si la cantidad a comprar es mayor que la cantidad mínima para obtener el mayor descuento. En tal caso devolvemos el mayor porcentaje. De no ser asi, accedemos al vector según la cantidad, ya que cada una de sus posiciones contiene el porcentaje apropiado para ella. Estas operaciones solo tendrían una complejidad temporal de $\theta(1)$.

Por lo tanto, la complejidad final quedará en θ(Log(I))

NuevoPuesto(in *dPrecio*: `dicc(item, precio)`, **in** *dDesc*: `dicc(item, dicc(cantidad, nat))`, **in** *dStock*: `dicc(item, cantidad)`) → *res* : puesto

```
1: // Copiamos los argumentos para evitar aliasing
2: copiaDiccionarioPrecio ← Copiar(dPrecio)                                   ▷ Θ(I)
3: copiaDiccionarioStock ← Copiar(dStock)                                     ▷ Θ(I)
4:
5: // Construimos los vectores de Descuentos por cada item
6: diccionarioDescuentos ← Vacio()                                            ▷ Θ(1)
7: itDiccionario ← CrearIt(diccionarioDescuentos)                             ▷ Θ(1)
8:
9: // Agregamos los descuentos
10: while (HaySiguiente(itDiccionario)) then                                  ▷ Θ(1)
11:
12:     // Creamos un vector vacío en donde asociaremos cada posición con su cantidad y su descuento correspondiente.
13:     vectorDescuentos ← Vacio()                                            ▷ Θ(1)
14:     diccCantDescuentos ← SiguienteSignificado(itDiccionario)              ▷ Θ(1)
15:     itDescuento ← CrearIt(diccCantDescuentos)                             ▷ Θ(1)
16:     indexActual ← 0                                                       ▷ Θ(1)
17:     DescuentoActual ← 0                                                   ▷ Θ(1)
18:
19:     // Iteramos por cada cantidad de item en donde el descuento aumenta
20:     while (HaySiguiente(itDescuento)) then                                ▷ Θ(1)
21:
22:         // Mientras nuestro index sea menor a la cantidad en donde el descuento aumenta, seguimos agregando
23:         // El porcentaje anterior (el cual incialmente es 0)
24:         while (indexActual < SiguienteClave(itDescuento)) then            ▷ Θ(1)
25:             AgregarAtras(vectorDescuentos, DescuentoActual)               ▷ Θ(I)
26:             index++                                                       ▷ Θ(1)
27:
28:         //Al llegar a una cantidad con un nuevo descuento, empezamos a registrar este.
29:         DescuentoActual ← SiguienteSignificado(itDescuento)               ▷ Θ(1)
30:         Avanzar(itDescuento)                                              ▷ Θ(1)
31:
32:     // Dado que el ciclo parará antes de agregar el último descuento, lo colocamos de forma manual y guardamos su
33:     // Cantidad minima.
34:     AgregarAtras(vectorDescuentos, DescuentoActual)                       ▷ Θ(I)
35:     max ← AnteriorClave(itDescuento)                                      ▷ Θ(1)
36:
37:     // Asociamos la información del descuento con el item
38:     infoDescuento ← < max, vectorDescuento >                              ▷ Θ(1)
39:     Definir(diccionarioDescuentos, Siguiente(itDiccionario), infoDescuento)   ▷ Θ(Log(I) + (Max + 1))
40:     Avanzar(itDiccionario)                                                ▷ Θ(1)
41: // Inicializamos los diccionarios restantes
42: diccVentas ← Vacio()                                                      ▷ Θ(1)
43: diccHackeos ← Vacio()                                                     ▷ Θ(1)
44: res ← < copiaDiccionarioPrecio, copiaDiccionarioStock, diccionarioDescuentos, diccVentas, diccHackeos > Θ(1)
```

Complejidad: $\Theta(I + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$

Justificación: Primero copiamos los diccionarios dPrecio y dDesc con el fin de evitar problemas de aliasing. Copiarlos costará la cantidad de items de dicho puesto, ya que la copia de un solo item será $\Theta(1)$; con lo cual al final esto todo tardará $\Theta(I)$.

Luego inicializamos el diccionario en donde se guardará la información de cada descuento según su cantidad. Para conseguir ésta información tendremos que iterar por cada uno de los items y por sus descuentos.

Por cada item con descuento iniciaremos un vector. Luego agregaremos uno por uno el descuento asociado a la cantidad igual a la posición. Por ejemplo, si un item tuviese estos descuentos asociados a su cantidad:

[(2,15),(5,40),(7,90)]

El vector resultante sería:

[0,0,15,15,15,40,40,90]

Para esto deberemos iterar por cada cantidad de descuento máxima de cada item. Todo esto tendra una complejidad de $\Theta(\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item)))$

Por último inicializamos dos vectores vacios necesarios para la estrucutra. Esto es $\Theta(1)$

iGastosDe(**in** p : `puesto`, **in** a : `persona`) $\rightarrow$ res : nat

```
1: if (Definido?(p.diccVentas, a)) then                          ▷ Θ(Log(P))
2:     res ← Significado(p.diccVentas, a).total                  ▷ Θ(Log(P))
3: else
4:     res ← 0                                                   ▷ Θ(1)
```

Complejidad: $\Theta(log(P))$

---

iCalcularCostoVenta(**in** p : `puesto`, **in** i : `item`, **in** $cant$: `nat`) $\rightarrow$ res : nat

```
1:     precioUnit ← Significado(p.diccPrecio, i)                 ▷ Θ(Log(I))
2:     descProd ← iDescuento(p, i, cant)                         ▷ Θ(Log(I))
3:     res ← cant * div(precioUnit * (100 − descProd), 100)      ▷ Θ(1)
```

Complejidad: $\Theta(log(I))$

Justificación: Recuperar el valor del item tendrá una complejidad de $\Theta(log(I))$ dado que se encuentra guardado en un diccionario logaritmico. Encontrar su descuento correspondiente, tardará $\Theta(log(I))$.

Por último, calcular el gasto final será $\Theta(1)$, por solo usar operaciones básicas.

---

iVentaSinPromo?(**in** p : `puesto`, **in** i : `item`, **in** $cant$: `nat`) $\rightarrow$ res : $bool$

```
1: descProd ← iDescuento(p, i, cant)                             ▷ Θ(Log(I))
2: if (descProd == 0) then                                       ▷ Θ(1)
3:     res ← true                                                ▷ Θ(1)
4: else
5:     res ← false                                               ▷ Θ(1)
```

Complejidad: $\Theta(log(I))$

iRegistrarVenta(**in/out** p: `puesto`, **in** i: `item`, **in** a: `persona`, **in** $cant$: `nat`)

```
1: stockDisponible ← iStock(p, i)                                                    ▷ Θ(Log(I))
2:
3: //Eliminamos del stock la cantidad comprada
4: stockDisponible ← stockDisponible − cant                                          ▷ Θ(1)
5:
6: //Calculamos el costo de la venta
7: precioVenta ← CalcularCostoVenta(p, i, cant)                                      ▷ Θ(Log(I))
8:
9:
10: // Verificamos si la persona ya tuvo alguna venta en el puesto. En caso contrario, inicializamos su historial.
11: if (!Definido?(p.diccVentas, a)) then                                            ▷ Θ(Log(A))
12:     Definir(p.diccVentas, a, < 0, Vacia() >)                                     ▷ Θ(Log(A))
13:
14: // Actualizamos el gasto total de la persona y su historial
15: gastosHastaLaFecha ← Significado(p.diccVentas, a)                                ▷ Θ(Log(A))
16: gastosHastaLaFecha.total ← gastosHastaLaFecha.total + precioVenta                ▷ Θ(1)
17: iteradorVenta ← AgregarAtras(gastosHastaLaFecha.historial, < i, cant >)          ▷ Θ(1)
18:
19: // Si la venta es hackeable, guardamos el iterador que apunta a esta venta en DiccHackeos
20: if (iVentaSinPromo?(p, i, cant)) then                                            ▷ Θ(Log(I))
21:
22:     // Si la persona nunca tuvo una venta hackeable, inicializamos su diccionario
23:     if (!Definido?(p.diccHackeos, a)) then                                       ▷ Θ(Log(A))
24:         Definir(p.diccHackeos, a, Vacio())                                       ▷ Θ(Log(A))
25:     diccHackeosItems ← Significado(p.diccHackeos, a)                             ▷ Θ(Log(A))
26:
27:     // Si la persona nunca tuvo una venta hackeable con este item, inicializamos su diccionario
28:     if (!Definido?(diccHackeosItems, i)) then                                    ▷ Θ(Log(I))
29:         Definir(diccHackeosItems, i, Vacia())                                    ▷ Θ(Log(I))
30:     listaVentasHackeables ← Significado(diccHackeosItems, i)                     ▷ Θ(Log(I))
31:     AgregarAtras(listaVentasHackeables, iteradorVenta)                           ▷ Θ(1)
```

Complejidad: $\Theta(log(I))+\Theta(Log(A)) = \Theta(Log(I) + Log(A))$

Justificación: En primer lugar, obtenemos el stock del item para actualizarlo. Esto tardará Θ(log(I)).

Luego calcularemos el costo de la venta, lo que también tardará Θ(log(I)). Este valor se usará para actualizar el historial de la persona, pero antes tendremos que verificar si ya tiene un historial inicializado, lo que tardará Θ(log(A)).

Actualizar el gasto de la persona y su historial tendrá una complejidad de $\Theta(log(A))$.

Por último, chequearemos si la venta puede ser hackeable. En caso de que lo sea, agregaremos un iterador apuntado hacia ella en la lista de ventas hackeables. Todo esto tendrá una complejidad de $\Theta(log(I))$

La complejidad de todo el algoritmo quedará como $\Theta(Log(I) + Log(A))$

---

iEsHackeable(**in** p: `puesto`, **in** i: `item`, **in** a: `persona`) → res : $bool$

```
1: diccHackeosItems ← Significado(p.diccHackeos, a)                                  ▷ Θ(Log(A))
2: listaVentasHackeables ← Significado(diccHackeosItems, i)                          ▷ Θ(Log(I))
3: if (longitud(listaVentasHackeables)! = 0) then                                    ▷ Θ(1)
4:     res ← true                                                                    ▷ Θ(1)
5: else
6:     res ← false                                                                   ▷ Θ(1)
```

Complejidad: $\Theta(log(I))+\Theta(log(A)) = \Theta(log(I) + log(A))$

iHackeo(in/out p: `puesto`, **in** i: `item`, **in** a: `persona`)

1: $diccHackeosItems \leftarrow Significado(p.diccHackeos, a)$ ▷ $\Theta(log(A))$
2: $listaVentasHackeables \leftarrow Significado(diccHackeosItems, i)$ ▷ $\Theta(log(I))$
3:
4: // Obtengo el iterador apuntado a la primera venta hackeable y guardo en itemHackeado y cantHackeado su información
5: $itVentaHackeable \leftarrow Primero(listaVentasHackeables)$ ▷ $\Theta(1)$
6: $ventaHackeable \leftarrow Siguiente(itVentaHackeable)$ ▷ $\Theta(1)$
7: $itemHackeado \leftarrow \pi_1(ventaHackeable)$ ▷ $\Theta(1)$
8: $cantHackeado \leftarrow \pi_2(ventaHackeable)$ ▷ $\Theta(1)$
9:
10: // Actualizo el Stock, agregándole la cantidad hackeada.
11: $stockRestante \leftarrow iStock(p, itemHackeado)$ ▷ $\Theta(log(I))$
12: $stockRestante \leftarrow stockRestante + cantHackeado$ ▷ $\Theta(1)$
13:
14: // Actualizo el costo de la persona, restándole el gasto de la venta Hackeada
15: $precioVenta \leftarrow CalcularCostoVenta(p, itemHackeado, cantHackeado)$ ▷ $\Theta(Log(I))$
16: $gastosHastaLaFecha \leftarrow Significado(p.diccVentas, a)$ ▷ $\Theta(log(A))$
17: $gastosHastaLaFecha.total \leftarrow gastosHastaLaFecha.total - precioVenta$ ▷ $\Theta(1)$
18:
19: //Elimino la venta del historial y su iterador.
20: $EliminarSiguiente(itVentaHackeable)$ ▷ $\Theta(1)$
21: $Fin(listaVentasHackeables)$ ▷ $\Theta(1)$

Complejidad: $\Theta(log(I))+\Theta(log(A)) = \Theta(log(I) + log(A))$
Justificación: Lo primero que hacemos es obtener el diccionario de hackeos de la persona para obtener la lista de ventas hackeables con el item dado. Todo esto tardará $\Theta(Log(A))$ y $\Theta(Log(I))$, correspondientemente.

Luego obtengo el iterador apuntando a la primera venta hackeable de la lista, guardando la cantidad y el item. Dado que tenemos el iterador podemos encontrar esta iinformacion en $\Theta(1)$.

El siguiente paso será actualizar tanto el stock como el gasto de la persona, cancelando los cambios causados por la venta. Esto tardará Θ(Log(A)) para modificar el gasto de la persona y Θ(Log(I)) para modificar el stock.

Finalmente eliminamos la venta del historial y su iterador en itVentaHackeable. Todo esto tardará $\Theta(1)$ ya que se trata de operaciones sobre iteradores.

iCopiar(**in** p: `puesto`) $\rightarrow$ res : $puesto$

1: //Copio las estructuras de Puesto que no utilizan iteradores
2: $diccPrecioCopia \leftarrow copiar(p.diccPrecio)$ ▷ $\Theta(I)$
3: $diccStockCopia \leftarrow copiar(p.diccStock)$ ▷ $\Theta(I)$
4: $diccDescuentosCopia \leftarrow copiar(p.infoDescuentos)$ ▷ $\Theta(\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$
5: $diccVentasCopia \leftarrow copiar(p.diccVentas)$ ▷ $\Theta(VentasTotales)$
6:
7: // Creo iteradores que apunten a las ventas copiadas hackeables
8: $diccHackeosCopia \leftarrow Vacio()$ ▷ $\Theta(1)$
9: $itHackeos \leftarrow CrearIt(e.diccHackeos)$ ▷ $\Theta(1)$
10:
11: // Por cada persona y por cada uno de sus items comprados busco las ventas hackeables originales y creo los nuevos iteradores apuntando a las copias de las ventas.
12: **while** ($HaySiguiente(itHackeos)$ ▷ $\Theta(1)$
13: $personaHackeada \leftarrow SiguienteClave(itHackeos)$ ▷ $\Theta(1)$
14: $Definir(diccHackeoCopia, personaHackeada, vacio())$ ▷ $\Theta(Log(P))$
15: $DiccItemsHackCopia \leftarrow Significado(diccHackeoCopia, personaHackeada)$ ▷ $\Theta(Log(P))$
16: $itItemsHack \leftarrow CrearIt(SiguienteSignificado(itHackeos))$ ▷ $\Theta(1)$
17:
18: // Por cada item que tiene la posibilidad de ser hackeable creo los nuevos iteradores que apuntan a las copias de
19: // Ventas
20: **while** (($HaySiguiente(itItemsHack)$) ▷ $\Theta(1)$
21: $itemHackeado \leftarrow SiguienteClave(itItemsHack)$ ▷ $\Theta(1)$
22: $Definir(DiccItemsHackCopia, itemHackeado, vacia())$ ▷ $\Theta(Log(I))$
23: $ListaVentasHackCopia \leftarrow Significado(DiccItemsHackCopia, itemHackeado)$ ▷ $\Theta(Log(I))$
24:
25: // Recorro las ventas copiadas y guardo los iteradores de las cuales que se encontraban originalmente en la
26: // lista de VentasHackeables
27: $itListaVentasHackeables \leftarrow CrearIt(SiguienteSignificado(itItemsHack))$ ▷ $\Theta(1)$
28: $itListaVentas \leftarrow CrearIt(Significado(diccVentasCopia, personaHackeada).historial)$ ▷ $\Theta(Log(P))$
29: **while** ($HaySiguiente(itListaVentasHackeables)$) ▷ $\Theta(1)$
30:
31: // Cuando encuentro una venta que coincida con los iteradores de la lista original
32: // Guardo el iterador de la lista copiada
33: $iventaHackeable \leftarrow Siguiente(Siguiente(itListaVentasHackeables))$ ▷ $\Theta(1)$
34: $ventaActual \leftarrow Siguiente(itListaVentas)$ ▷ $\Theta(1)$
35: **if** ($ventaHackeable == ventaActual$) ▷ $\Theta(1)$
36: $Agregar(ListaVentasHackCopia, itListaVentas)$ ▷ $\Theta(1)$
37:
38: //Busco la siguiente venta Hackeable
39: $Avanzar(itListaVentasHackeables)$ ▷ $\Theta(1)$
40: $Avanzar(itListaVentas)$ ▷ $\Theta(1)$
41: $res \leftarrow < diccPrecioCopia, diccStockCopia, diccDescuentosCopia, diccVentasCopia, diccHackeosCopia >$ ▷ $\Theta(1)$

Complejidad: $\Theta(I) + \Theta(\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item)) + \Theta(VentasTotales) + \Theta(LogP) + \Theta(Log(I)) = \Theta(I + Cantidad\ total\ de\ Descuentos + A * I * CantidadVentasActuales)$

Justificación: Lo primero que hacemos es copiar las estrucuturas que no utilizan iteradores. Entre ellas diccPrecio y diccStock, las cuales ambas tienen una clave por item y significados naturales, por esto la copia tendrá complejidad Θ(I). Por otro lado, diccDescuentos contiene un vector de igual longitud que la mayor cantidad de descuento por cada item, por lo tanto la complejidad de la copia será la suma de las longitudes de estos vectores, lo cual es $\Theta(\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$. Por último DiccVentas contiene el historial de compra de cada persona, por lo cual tiene en él todas las ventas realizadas. Es por eso que el gasto de copiarlo es la cantidad de ventas realizadas hasta el momento (la complejidad es Θ(Ventas Totales))

Luego tendremos que copiar diccHackeos, la cual utiliza iteradores. Por esta razón, tendremos que recorrer por cada iterador guardado y crear su equivalente en la versión de copia. Para esto empezamos iterando por cada persona hackeada, consiguiendo las listas de ventas hackeables de cada item. Por cada una de estas tendremos que recorrer la copia de la lista de ventas hasta encontrar su equivalente y guardar el nuevo iterador. Es decir, por cada persona y por cada item tendremos que iterar por la lista de ventas, quedando con una complejidad de Θ(A * I * CantidadVentasActuales)

2. Módulo Lollapatuza

Interfaz

se explica con: LollaPatuza

géneros: `lolla`

Servicios Usados:
DiccLog :

Vacío() → res : dicc(κ, σ)
Complejidad: $\theta(1)$
Descripción: Genera un diccionario vacío.

Definir(in/out d : dicc(κ, σ), in k : κ, in s : σ) → res : itDicc(κ, σ)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)) + copy(k) + copy(s))$, donde K = claves(d)
Descripción: Define la clave k con el significado s en el diccionario. Retorna un iterador al elemento recién agregado.
Aliasing: Los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

Definido?(in d : dicc(κ, σ), in k : κ) → res : bool
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: Devuelve true si y sólo k está definido en el diccionario.

Significado(in d : dicc(κ, σ), in k : κ) → res : σ
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: Devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

Borrar(in/out d : dicc(κ, σ), in k : κ)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: Elimina la clave k y su significado de d.

Copiar(in d : dicc(κ, σ)) → res : dicc(κ, σ)
Complejidad: $\theta(\sum_{k' \in K}(copy(k) + copy(significado(k,d))))$, donde K = claves(d)
Descripción: Genera una copia nueva del diccionario.

Lista Enlazada :
Vacía() → res : lista(α)
Complejidad : $\theta(1)$
Descripción: Genera una lista vacìa

AgregarAtras(in/out l : lista(α), *in a* : α) → *res* : *itLista*(α)
Complejidad : $\theta(copy(a))$
Descripción: Agrega el elemento a como ultimo elemento de la lista. Retorna un iterador a l, de forma tal que Siguiente devuelva a.

Primero(in l : lista(α)) → *res* : α
Complejidad : $\theta(1)$
Descripción: Devuelve el primer elemento de la lista.
Aliasing: res es modificable si y solo si l es modificable.

Fin(in/out l : lista(α))
Complejidad : $\theta(1)$
Descripción: Elimina el primero elemento de l

Longitud(in l : lista(α)) $\rightarrow$ res : nat
$Complejidad$: $\theta(copy(a))$
Descripción: Devuelve la cantidad de elementos que tiene la lista

Copiar(in l : lista(α)) $\rightarrow$ res : $lista(\alpha)$
$Complejidad$: $\theta(\sum_{i=1}^{t} copy(l[i]))$, donde t = long(l).
Descripción: Genera una copia nueva de la lista

Vector :
Vacía() $\rightarrow$ res : vector(α)
$Complejidad$: $\theta(1)$
Descripción: Genera un vector vacìo

Longitud(in v : vector(α)) $\rightarrow$ res : nat
$Complejidad$: $\theta(1)$
Descripción: Devuelve la cantidad de elementos que contiene el vector

Copiar(in v : vector(α)) $\rightarrow$ res : $vector(\alpha)$
$Complejidad$: $\theta(\sum_{i=1}^{l} copy(v[i]))$, donde l = long(v).
Descripción: Genera una copia nueva del vector

AgregarAtras(in/out v : vector(α), in a : α)
$Complejidad$: $\theta(f(long(v)) + copy(\alpha))$.
Descripción: Agrega el elemento a como el ultimo elemento del vector.
Aliasing: El elemento a se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando long(v) es potencia de 2.

•[•](in v : vector(α), ini : nat) $\rightarrow$ res : α
$Complejidad$: $\theta(1)$.
Descripción: Devuelve el elemento que se encuentra en la i- esima posicion del vector en base 0. Es decir, v[i] devuelve el elemento que se encuentra en la posicion i + 1.
Aliasing: res es modificable si y solo si v es modificable.

Operaciones básicas de Lollapatuza

PERSONAS(**in** l : `lolla`) $\rightarrow$ res : `listaEnlazada(persona)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ Personas(l)}
Complejidad: $\Theta(1)$
Descripción: Devuelve una lista con la cantidad de personas que hay en lolla
Aliasing: res es modificable si y solo si diccPersona es modificable

PUESTOS(**in** l : `lolla`) $\rightarrow$ res : `listaEnlazada(tupla<ID, puesto>)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ Puestos(l)}
Complejidad: $\Theta(1)$
Descripción: Devuelve una lista con todos los puestos y su respectivo ID que hay en lolla
Aliasing: res es modificable si y solo si los dIccPuestos son modificablen

CREARLOLLA(**in** ps : `dicc(id, puesto)`, **in** as : `conj(persona)`) $\rightarrow$ res : `lolla`
Pre $\equiv$ {($\neg$ $\emptyset$? (claves(ps)) $\wedge$ $\neg$ $\emptyset$? (as)) $\wedge_{\text{L}}$ vendenAlMismoPrecio(ps) $\wedge$ noVendieronAun(ps)}
Post $\equiv$ {res $=_{\text{obs}}$ CrearLolla(ps,as)}
Complejidad: Θ(P * (I + log(P) + $\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item)) + A*\log(A)$)
Descripción: Dado un diccionario con todos los ID y sus respectivos puestos y un conjunto de personas, crea una nueva instancia de Lollapatuza.
Aliasing: No hay aliasing ya que se crea por copia cada una de las estructuras

VENDER(**in/out** l : `lolla`, **in** ID : `nat`, **in** a : `personas`, **in** i : `item`, **in** *cant* : `nat`)
Pre $\equiv$ {l = l_0 $\wedge$ a $\in$ Personas(l) $\wedge$ def?(ID, puestos(l)) $\wedge_L$ i $\in$ menu(obtener(ID, puestos(l))) $\wedge_L$ haySuficiente?(obtener(ID, puestos(l)), i, cant))}
Post $\equiv$ {Vender($l_0, puestoId, a, i, cant$)}
Complejidad:$\Theta(\log(A) + \log(I) + \log(P))$
Descripción: Dado un item, una persona y una cantidad a comprar, se registra la venta en el puesto de comida con mismo ID. Esto actualiza el gasto total de la persona junto al puesto
Aliasing: No aplica

HACKEAR(**in/out** l : `lolla`, **in** a : `persona`, **in** i : `item`)
Pre $\equiv$ {l = l_0 $\wedge$ a $\in$ Personas(l) $\wedge_L$ ConsumioSinPromoPuestos?(a, i, Puestos(l))}
Post $\equiv$ {l $=_{obs}$ Hackear(l_0, a, i)}
Complejidad: $\Theta(\log(A) + \log(I) + \log(P))$ si el puesto deja de ser hackeable luego de la operacion.
En caso de que siga siendo hackeable será $\Theta(\log(A) + \log(I))$
Descripción: Dada una persona y un item, se hackea al puesto con menor ID que contenga una venta hackeable de la persona e item. Esto actualizará al puesto de comida y al gasto total de la persona
Aliasing: No aplica

GASTOTOTALDE(**in** l : `lolla`, **in** a : `persona`) $\rightarrow$ *res* : `dinero`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ GastosTotales(l,a)}
Complejidad: $\Theta(\log(A))$
Descripción: Dado un Lolla y una persona, devuelve los gastos totales de esta dentro del Lolla.
Aliasing: Devuelve un valor por referencia por lo cual modificarlo actua directamente en diccPersona

PERSONAMASGASTO(**in** l : `lolla`) $\rightarrow$ *res* : `persona`
Pre $\equiv$ {$\neg$ $\emptyset$?(personas(l))}
Post $\equiv$ {res $=_{obs}$ masGasto(l)}
Complejidad: $\Theta(1)$
Descripción: Dado un Lolla, nos devuelve la persona con la mayor cantidad de gastos
Aliasing: Devuelve un valor por referencia por lo cual modificarlo actua directamente en colaPrioridadGastos

MENORSTOCK(**in** l : `lolla`, **in** i : `item`) $\rightarrow$ *res* : `ID`
Pre $\equiv$ {True}
Post $\equiv$ {res $=_{obs}$ menorStock(l, i)}
Complejidad: $\Theta(P * Log(I))$
Descripción: Dado un Lolla, nos devuelve el ID del puesto con menor stock de un item dado (o si hay mas de uno, el que tenga menor ID)
Aliasing: No aplica

Especificación de las operaciones auxiliares utilizadas en la representación

TAD LollaPatuza extendido

otras operaciones

sumaHistorialPersona : conj(ID) $\times$ Dicc(ID $\times$ puesto) $\times$ persona $\longrightarrow$ Nat
EstanClavesEnLista? : conj(ID) $\times$ secu(<ID $\times$ puesto>) $\longrightarrow$ bool
EstanEnTupla? : ID $\times$ secu(<ID $\times$ puesto>) $\longrightarrow$ bool

axiomas

sumaHistorialPersona(conjClaves, DiccPuestos, a) $\equiv$ **if** $\emptyset$?(conjClaves) **then**
0
else
GastoDe(Obtener(DameUno(conjClaves), DiccPuestos), a)
+ sumaHistorialPersona(Fin(conjClaves), DiccPuestos, a)
fi

EstanClavesEnLista?(conj, lista) $\equiv$ **if** $\emptyset$?(conj) **then**
true
else
if (EstaEnTupla?(DameUno(conj), lista)) **then**
EstanClavesEnLista(SinUno(conj),lista)
else
false
fi
fi

EstaEnTupla?(ID, secu) $\equiv$ **if** vacia?(secu) **then**
false
else
if (ID = π_1(Prim(secu)) **then**
True
else
EstaEnTupla?(ID, fin(secu))
fi
fi

Fin TAD

Representación

Representación de Lollapatuza

`Lolla` **se representa con** `estr`

donde `lolla` es `tupla`(*diccPuestos*: `diccLog(Id, puesto)`
, *diccIteradoresPuestos*: `diccLog(Id, ItDiccLog)`
, *listaPuestos*: `listaEnlazada(tupla<Id, puesto>)`
, *diccPersonas*: `diccLog(persona, nat)`
, *listaPersonas*: `listaEnlazada(persona)`
, *colaPrioridadGastos*: `diccLog(<nat, persona>, persona)`
, *diccInfoHackeo*: `diccLog(persona,diccLog(item,diccLog(Id,ItDiccLog)))`
)

diccPuestos: Es un diccionario logaritmico que guarda los puestos de comida registrados en el Lolla según su ID.
diccIteradoresPuestos: Es un diccionario logaritmico que asocia un ID de puesto con un iterador que apunta a su respectivo puesto en diccPuestos. Lo usamos para poder obtener en tiempo logaritmico un puntero al puesto y poder utilizarlo en otras estructuras. Concretamente para guardar los puestos que tienen la posibilidad de ser hackeables en Ivender.
listaPuestos: Es una lista enlazada de tuplas que contienen un ID y el puesto asosicado a dicho ID. Lo utilizamos para devolver en $\theta(1)$ los puestos del Lolla con su ID.
diccPersonas: Es un diccionario logaritmico que asocia a una persona registrada en lolla con sus gastos totales (Es decir, la suma total de sus gastos en los distintos puestos).
listaPersonas: Es una lista enlazada que contiene las personas registradas en el Lolla
colaPrioridadGastos: Es un diccionario logaritmico que utilizamos para ordenar de forma creciente a las personas, según sus gastos realizados y su identificador (el cual es el mismo que el de su significado). Para guardar las tuplas se usa la siguiente relacion:

$$<a, b> > <c, d> \leftrightarrow a > c \vee (a = c \wedge b > d)$$

Con esto podemos encontrar a la persona con más gastos en $\theta(1)$, tal como si se tratase de un heap.
diccInfoHackeo: Es un diccionario logaritmico que relaciona a una persona con todos los puestos en donde realizo una compra hackeable de un item especifico. Utilizamos el ultimo diccionario que aparece en la estructura para obtener de forma inmediata un iterador que apunta al puesto con menor ID .

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$

//Debe haber al menos un puesto en el lolla
$\neg\emptyset?(Claves(e.diccPuestos))\wedge$

//Todos los puestos deben vender sus items al mismo precio
$\neg(\exists p1, p2 : Puesto, i : Item)(def?(p1, e.diccPuestos) \wedge def?(p2, e.diccPuestos) \wedge i \in menu(p1) \wedge i \in menu(p2) \wedge_{\text{L}} precio(i, p1) \neq precio(i, p2)) \wedge$

//Todos los puestos tienen su iterador guardado en DiccIteradoresPuestos
Claves(e.DiccIteradoresPuestos) = Claves(e.DiccPuestos) $\wedge_{\text{L}}$

//Los Iteradores deben apuntar al Puesto con su mismo ID en DiccPuestos
$(\forall\ ID : nat)(def?(ID, e.DiccIteradoresPuestos) \Rightarrow_{\text{L}} Siguiente(Obtener(ID, e.DiccIteradoresPuestos)) = Obtener(ID, e.DiccPuestos))\wedge$

//Todos los Puestos deberian tener una Tupla en ListaPuestos
(EstanClavesEnLista?(claves(e.diccPuestos),e.ListasPuestos)$\wedge$
$Claves(e.diccPuestos) = long(e.ListasPuestos)\wedge$

//Todas las Tuplas en ListaPuestos deben tener un ID y el puesto de Comida relacionado con dicho ID
$(\forall\ ID : nat)(def?(ID, e.diccPuestos) \Rightarrow_{\text{L}} Esta?(<ID, Significado(e.diccPuestos, ID)>), e.ListaPuestos))\wedge$

//La suma de los gastos de todos los Puestos debe ser igual al gasto total guardado en DiccPersonas
$(\forall\ ID : nat)(def?(ID, e.diccPuestos) \Rightarrow_{\text{L}} (obtener(a, e.DiccPersona) = sumaHistorialPersona(Claves(e.diccPuestos), e.DiccPuestos, a)\wedge$

//Los puestos a hackear deben pertenecer al diccionario de puestos
$\neg(\exists a : persona, I : item, ID : nat)(Def?(a, e.DiccInfoHackeo) \wedge_{\text{L}} Def?(i, Obtener(a, e.DiccInfoHackeo)) \wedge_{\text{L}} Def?(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))) \wedge_{\text{L}} \neg Def?(ID, e.DiccPuestos))\wedge$

//Los puestos a hackear deben coincidir con el puesto de mismo ID de diccPuestos
$\neg(\exists a : persona, I : item, ID : nat)(Def?(a, e.DiccInfoHackeo) \wedge_{\text{L}} Def?(i, Obtener(a, e.DiccInfoHackeo)) \wedge_{\text{L}} Def?(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))) \wedge_{\text{L}} Obtener(ID, e.DiccPuestos) \neq Siguiente(Obtener(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))))) \wedge$

//Los puestos a hackear deben deben ser hackeables, es decir, que tengan productos sin promocion
$\neg(\exists a : persona, I : item, ID : nat)(Def?(a, e.DiccInfoHackeo) \wedge_{\text{L}} Def?(i, Obtener(a, e.DiccInfoHackeo)) \wedge_{\text{L}} Def?(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))) \wedge_{\text{L}} ventasSinPromo(Obtener(ID, e.DiccPuestos)) = 0) \wedge$

//Cada ID debe de pertenecer a un solo puesto
$\neg(\exists\ ID1, ID2 : nat)((def?(ID1, e.DiccIteradoresPuestos) \wedge def?(ID2, e.DiccIteradoresPuestos) \wedge ID1 \neq ID2) \wedge_{\text{L}} (Obtener(ID1, e.DiccIteradoresPuestos) = Obtener(ID2, e.DiccIteradoresPuestos))) \wedge$

//Cada puesto en la lista de puestos debe de ser unico, no se pueden repetir
NoHayRepetidos(e.ListaPuestos) $\wedge$

//Las claves que aparecen en el diccionario de personas deben ser las mismas que se encuentran en la lista de personas
EstanClavesEnLista?(claves(e.DiccPersonas), e.ListaPersonas) $\wedge \#Claves(e.DiccPersonas) = longitud(e.ListaPersonas)\wedge$

//Todas las personas en el DiccPersonas deben tener su gasto correctamente registrado en colaPrioridadGastos
#Claves(e.colaPrioridadGastos) = #Claves(e.DiccPersonas) $\wedge(\forall\ a : personas)(def?(a, e.DiccPersonas) \wedge_{\text{L}} def?(< Obtener(e.DiccPersonas, a), a >, e.colaPrioridadGastos))\wedge$

//la persona dentro de la tupla Clave de colaPrioridadGastos debe coincidir con la de su significado
$\neg(\exists gasto : nat, a : persona)(def?(< gasto, a >, e.colaPrioridadGastos) \wedge_{\text{L}}$
$Obtener(< gasto, a >, e.colaPrioridadGastos) \neq a) \wedge$

//No puede haber personas repetidas en la lista de lolla
NoHayRepetidos(e.ListaPersonas) ∧

//Las personas hackeadas deben de pertnecer a la lista de personas de Lolla
Claves(e.DiccInfoHackeo) ⊆ $Claves(e.DiccPersonas)$

Abs : estr e $\longrightarrow$ lollapatuza {Rep(e)}

Abs(e) $=_{\text{obs}}$ lolla: lollapatuza |
$puestos(lolla) =_{\text{obs}} e.DiccPuestos \wedge$
$personas(lolla) =_{\text{obs}} claves(e.DiccPersonas)$

Algoritmos

Algoritmos del módulo Lollapatuza

---

iCrearLolla(**in** ps : `dicc(id, puesto)`, **in** as : `conj(persona)`) $\rightarrow$ res : lolla

1: // Copio los argumentos para evitar Aliasing y inicializo las estructuras
2: $diccPuesto \leftarrow copiar(ps)$ ▷ $\Theta(P * (I + \sum CantDeDescuentoMaxDeCadaItem))$
3: $itPuestos \leftarrow CrearIt(ps)$ ▷ $\Theta(1)$
4: $listaPuestos \leftarrow vacio()$ ▷ $\Theta(1)$
5:
6: // Agrego los puestos a Lista Puestos y guardo los iteradores de puestos
7: $diccIteradoresPuestos \leftarrow vacio()$ ▷ $\Theta(1)$
8: **while** $HaySiguiente(itPuestos)$ **then** ▷ $\Theta(P * Log(P))$
9: $AgregarAtras(listaPuestos, < SiguienteClave(itPuestos), SiguienteSignificado(itPuestos) >)$
10: $Definir(diccIteradoresPuestos, SiguienteClave(itPuestos), itPuestos)$
11: $Avanzar(itPuestos)$
12:
13: // Declaro los gastos iniciales de las personas, tanto en diccVentas como en el arbol de
14: // Gastos. Por ser una tupla, la clave mayor dependera de <a, b> > <c,d> $\leftrightarrow$ a > c $\vee$ (a = c $\wedge$ b > d)
15: $itPersonas \leftarrow CrearIt(as)$ ▷ $\Theta(1)$
16: $diccPersonas \leftarrow vacio()$ ▷ $\Theta(1)$
17: $colaPrioridadGastos \leftarrow vacio()$ ▷ $\Theta(1)$
18: **while** $HaySiguiente(itPersonas)$ **then** ▷ $\Theta(A * Log(A))$
19: $Definir(diccPersonas, Siguiente(itPersonas), 0)$;
20: $Definir(colaPrioridadGastos, < 0, Siguiente(itPersonas) >, Siguiente(itPersonas))$;
21: $Avanzar(itPersonas)$
22:
23: // Creo una copia del conjunto de personas e inicializo un diccionario para guardar la informacion de hackeos
24: $listaPersonas \leftarrow copiar(as)$ ▷ $\Theta(A)$
25: $diccInfoHackeo \leftarrow vacio()$ ▷ $\Theta(1)$
26: $res \leftarrow < diccPuesto, DiccIteradoresPuestos, listaPuestos, diccPersonas, listaPersonas,$
colaPrioridadGastos,diccInfoHackeo> ▷ $\Theta(1)$

Complejidad: $\Theta(P * (I + \log(P) + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item)) + A * \log(A))$
Justificación: En primer lugar copiamos los puestos de comida de Lolla. La funcion Copiar tendra la misma complejidad que la cantidad de puestos * el costo de copiarlos. Copiar un puesto tardará $\Theta(I + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item) + A * I * CantidadVentasActuales)$. Ya que conocemos que no se realizo ninguna venta, la complejidad se simplifica a $\Theta(P * (I + \sum CantDeDescuentoMaxDeCadaItem))$.

Luego guardo los puestos en ListasPuestos y los iteradores de DiccPuestos en DiccIteradoresPuestos. Agregar a un diccionario tendra costo logaritmico respecto a su cantidad de claves, las cuales aumentan en cada ciclo. Tendremos que el costo del while será $\theta(\sum_{n=0}^{P} log(n))$. Esta sumatoria puede ser acotada por P*Log(P)

Después, declarar los gastos inciales de cada una de las personas tardará A*Log(A) dado la lógica mencionada en el párrafo anterior

Por último creamos un copia de la lista de personas. Esto tardará igual a la cantidad de personas en el lolla ya que copiar una persona tarda $\theta(1)$.

---

iPersonas(**in** l : `lolla`) $\rightarrow$ res : listaEnlazada(persona)

1: $res \leftarrow l.listaPersonas$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iPuestos(**in** l : `lolla`) $\rightarrow$ res : listaEnlazada(puesto)

1: $res \leftarrow l.listaPuestos$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iVender(**in/out** l : `lolla`, **in** $puestoId$: `nat`, **in** a : `persona`, **in** i : `item` **in** $cant$: `nat`) $\rightarrow$ res : bool

1: $puestoVendedor \leftarrow Significado(l.diccPuestos, puestoId)$ ▷ $\Theta(log(P))$
2: $gastoPersona \leftarrow Significado(l.diccPersonas, a)$ ▷ $\Theta(log(A))$
3:
4: // Elimino la clave anterior de la cola de prioridad
5: $Borrar(l.colaPrioridadGastos, < gastoPersona, a >)$; ▷ $\Theta(log(A))$
6: $iRegistrarVenta(puestoVendedor, i, a, cant)$ ▷ $\Theta(log(I) + log(A))$
7:
8: // Si la venta fue hackeable, guardo el puesto de comida como opción a hackear
9: **if** ($iVentaSinPromo(puestoVendedor, i, cant)$) **then** ▷ $\Theta(log(I))$
10:
11: // Defino los diccionarios necesarios si aun no lo estaban
12: **if** (!$Definido?(l.diccInfoHackeo, a)$) **then** ▷ $\Theta(log(A))$
13: $Definir(l.diccInfoHackeo, a, vacio())$; ▷ $\Theta(log(A))$
14: $diccInfoHackeoItem \leftarrow Significado(l.diccInfoHackeo, a)$ ▷ $\Theta(log(A))$
15: **if** (!$Definido?(diccInfoHackeoItem, i)$) **then** ▷ $\Theta(log(I))$
16: $Definir(diccInfoHackeoItem, i, vacio())$; ▷ $\Theta(log(I))$
17: $diccPuestosHackeo \leftarrow Significado(diccInfoHackeoItem, i)$ ▷ $\Theta(log(I))$
18:
19: // Si el puesto ya había sido guardado como una opción, no se hace nada. En caso contrario, agrego su iterador
20: // y uso su ID como Clave
21: **if** (!$Definido?(diccPuestosHackeo, puestoId)$) **then** ▷ $\Theta(log(P))$
22: $PuestoIterador \leftarrow Significado(diccIteradoresPuestos, puestoId)$ ▷ $\Theta(log(P))$
23: $Definir(l.diccPuestosHackeo, puestoId, PuestoIterador)$ ▷ $\Theta(log(P))$
24:
25: //Actualizo el gasto de la persona
26: $CostoVenta \leftarrow iCalcularCostoVenta(puestoVendedor, i, cant)$ ▷ $\Theta(log(I))$
27: $gastoPersona \leftarrow gastoPersona + CostoVenta$ ▷ $\Theta(1)$
28:
29: //Agrego de nuevo el gasto de la persona en la cola de prioridad.
30: $Definir(l.colaPrioridadGastos, < gastoPersona, a >, a)$ ▷ $\Theta(Log(A))$

Complejidad: $\Theta(log(A)) + \Theta(log(P)) + \Theta(log(I)) = \Theta(log(A) + log(P) + log(I))$
Justificación: Lo primero que realizamos es obtener el puesto en donde se hace la venta y el gasto actual de la persona. Ya que esta información se encuentra guardada en diccionarios logarítmicos (implementados sobre un AVL), la complejidad de recuperar estos valores será Θ(Log(P)), dado que hay una clave por cada puesto; y Θ(Log(A)), ya que existe una clave por cada persona.

Luego borramos el gasto desactualizado de colaPrioridadGasto, lo que tardará Θ(Log(A)), ya que se trata de un Diccionario logarítmico con una clave por persona. Paso siguiente, realizaremos el registro de la venta dentro del puesto, lo que tiene una complejidad temporal de Θ(log(I) + Log(A))

Después verificamos si la venta fue hackeable. Si se da el caso, definimos los diccionarios dentro de DiccInfoHackeo (si es que estos no estaban definidos hasta el momento) y agregamos un iterador al puesto si no estaba aun como opcion. Todas estas operaciones tardaran en conjunto Θ(Log(I) + Log(A) + Log(P))

Por último, obtenemos el gasto de la venta, actualizamos el gasto de la persona y lo agregamos a la cola de prioridad. Conocer el gasto tendrá comprejidad log(I), actualizar el gasto tendrá como complejidad $\Theta(1)$ y colocarlo en la cola de prioridad Θ(Log(A))

iGastoTotalDe(**in** l : `lolla`, **in** a : `persona`) $\rightarrow$ res : nat

1: **if** ($!Definido?(l.diccPersonas, a)$) **then** ▷ $\Theta(Log(A))$
2: $res \leftarrow 0$ ▷ $\Theta(1)$
3: $res \leftarrow Significado(l.diccPersonas, a)$ ▷ $\Theta(Log(A))$

Complejidad: $\Theta(log(A))$

---

iHackear(**in/out** l : `lolla`, **in** a : `persona`, **in** i : `item`)

1: // Obtengo los puestos que pueden ser hackeados con la persona e item dados
2: $diccInfoHackeoItem \leftarrow Significado(l.diccInfoHackeo, a)$ ▷ $\Theta(log(A))$
3: $diccPuestosHackeo \leftarrow Significado(diccInfoHackeoItem, i)$ ▷ $\Theta(log(I))$
4:
5: //Agarro el primer Iterador del Arbol AVL, al ser en InOrder debería ser extremo izquierdo, y por eso el mínimo.
6: $itIDMenor \leftarrow crearIt(diccPuestosHackeo)$ ▷ $\Theta(1)$
7:
8: //El significado del diccionario es un iterador apuntando a un puesto válido.
9: $itPuesto \leftarrow SiguienteSignificado(itIDMenor)$ ▷ $\Theta(1)$
10: $PuestoConMenorId \leftarrow SiguienteSignificado(itIDMenor)$ ▷ $\Theta(1)$
11:
12: //Elimino el gasto desactualizado del puesto de Comida
13: $gastoPersona \leftarrow Significado(l.diccPersona, a)$ ▷ $\Theta(log(A))$
14: $Borrar(l.colaPrioridadGastos, < gastoPersona, p >)$ ▷ $\Theta(log(P))$
15: //Le resto el gasto de la persona en dicho puesto al gasto total de la misma, ya que después del hackeo esté se le sumará el actualizado.
16: $gastoPersona \leftarrow gastoPersona - iGastosDe(PuestoConMenorId, a)$ ▷ $\Theta(Log(P))$
17:
18: // Corro el Hackeo del puesto
19: $iHackeo(PuestoConMenorId, a, i)$ ▷ $\Theta(log(I) + log(A))$
20:
21: // Si sigue siendo hackeable el puesto no hacemos nada. En el caso contrario lo eliminamos del diccionario.
22: $sigueHackeable \leftarrow iEsHackeable?(PuestoConMenorId, a, i)$ ▷ $\Theta(Log(A) + Log(I))$
23: **if** ($!sigueHackeable$) **then** ▷ $\Theta(1)$
24: $EliminarSiguiente(itIDMenor)$ ▷ $\Theta(Log(P)$
25:
26: // Actualizo el gasto de la persona y lo agrego a la cola de prioridad
27: $gastoPersona \leftarrow gastoPersona + iGastosDe(PuestoConMenorId, a)$ ▷ $\Theta(log(P))$
28: $Definir(l.colaPrioridadGastos, < gastoPersona, a >, a)$ ▷ $\Theta(log(A))$

Complejidad: $\Theta(log(A))+\Theta(log(P))+ \Theta(log(I)) = \Theta(log(A) + log(P) + log(I))$
Justificación: Lo primero que hacemos es obtener los puestos con la posibilidad de ser hackeados dado la persona e item dado. Para eso, primero buscamos el diccionario de items hackeables relacionados con aquella persona; y luego, los puestos relacionados a dicho item. Esto tendrá complejidad $\Theta(Log(A))$ y $\Theta(Log(I))$, respectivamente.

Despues, tomamos el primer iterador del diccionario (como sabemos que se encuentra implementado sobre un árbol AVL y que se recorre InOrder, el primer iterador será el de menor clave, en este caso el de menor ID).

Este iterador tiene como valor un iterador que apunta al puesto hackeable con menor ID, en donde correremos el hackeo. Previamente, borraremos el gasto desactualizado de la persona en gastoPersona y en la cola de prioridad, lo cual tendrá una complejidad de $\Theta((log(A))$ y $\Theta((log(P))$

Una vez ejecutado el hackeo verificaremos si ese puesto sigue siendo hackeable, si no es asi lo eliminaremos del diccionario de dicho item. Esto tiene una complejidad $\Theta(Log(A) + Log(I))$ cuando chequeamos, y $\Theta(Log(P))$ para eliminarlo. Por último, actualizamos el gasto de la persona y lo agregamos a la cola de prioridad. La complejidad para esto es $\Theta((log(P))$ y $\Theta((log(A))$, respectivamente

iMenorStock(**in** l : `lolla`, **in** i : `item`) $\rightarrow$ res : nat

```
1:  // Inicializamos el iterador para recorrer los puestos y declaramos al último puesto como el puesto con menor Stock.
2:  itDiccionario ← CrearItUlt(l.diccPuestos)                                          ▷ Θ(1)
3:  puestoMinId ← anteriorClave(itDiccionario)                                         ▷ Θ(1)
4:  MinStock ← 0                                                                       ▷ Θ(1)
5:  StockEncontrado ← false                                                            ▷ Θ(1)
6:
7:  while (HayAnterior(itDiccionario))                                                 ▷ Θ(1)
8:      puestoActual ← AnteriorSignificado(itDiccionario)                              ▷ Θ(1)
9:
10:     // Mientras que no encuentre un puesto que tenga al item en su lista de stock,
11:     // Reemplazamos automaticamente el puestoMinId al siguiente puesto
12:     if (!StockEncontrado) then                                                     ▷ Θ(1)
13:         PuestoMinId ← AnteriorClave(itDiccionario)                                 ▷ Θ(1)
14:
15:         // Verificamos si el puesto contiene al item en su lista de Stock
16:         if (Definido?(puestoActual.diccStock, i) then                              ▷ Θ(Log(I))
17:
18:             // Si es así, modificamos la flag StockEncontrado y actualizo el mínimo stock encontrado.
19:             MinStock ← Copiar(iStock(puestoActual, i))                             ▷ Θ(Log(I))
20:             StockEncontrado ← true                                                 ▷ Θ(1)
21:
22:     // Si ya se encontro un puesto con el item en el stock, lo comparamos con el puesto actual para ver cual es menor.
23:     else if (Definido?(puestoActual.diccStock, i)) then                            ▷ Θ(Log(I))
24:         stockActual ← iStock(puestoActual, i)                                      ▷ Θ(1)
25:
26:         // Si tienen el mismo stock, ganará el puesto actual ya que debe entregar el puesto con menor ID.
27:         // Al recorrerse en InOrder, se que el ID Actual será menor a todos los anteriores
28:         if (stockActual ≤ MinStock) then                                           ▷ Θ(1)
29:
30:             // Reemplazamos el valor del mínimo id por el valor actual y también el stock.
31:             PuestoMinId ← AnteriorClave(itDiccionario)                             ▷ Θ(1)
32:             MinStock ← Copiar(stockActual)                                         ▷ Θ(Log(I))
33:     Retroceder(itDiccionario)
34: res ← PuestoMinId                                                                 ▷ Θ(1)
```

Complejidad: $\Theta(P * Log(I))$

Justificación: En primer lugar inicializaremos el iterador para empezar a recorrer por los puestos del Lolla. Esto tomará $\Theta(1)$.

Por cada ciclo verificaremos si ya se encontró un puesto de comida con el item registrado en su stock. De no ser el caso, se disminuira el ID del puesto con menor stock. De esta forma si el item no se encuentra en ningún puesto, se devolverá el que tenga el menor ID, dado que se itera por InOrder. También verificaremos si el puesto contiene al item en su stock, lo que tardará θ(Log(I)). De ser así, activamos la flag y actualizamos el mínimo stock. Una vez encontrado un puesto con stock del item, chequearemos si el puesto actual tambien lo hace, si es asi, compararemos quien tiene el menor stock. Esto se hará en tiempo θ(Log(I)). Por último, si el puesto tuvo menor o igual stock que el mínimo(ya que se nos pide el menor ID), actualizaremos el id del menor stock, el menor stock encontrado y continuaremos con el ciclo. Para resumir, estamos iterando por cada puesto dentro del Lolla, en donde en cada iteración se realizan 3 operaciones independientes con complejidad θ(Log(I)). Por lo tanto la complejidad resultante será θ(P * Log(I)).

iPersonaMasGasto(**in** l : `lolla`) $\rightarrow$ res: $persona$

1: //CrearItUlt devuelve el iterador tal que Anterior es igual al último elemento del orden.
2: //Ya que este es InOrder de un AVL, sabemos que el último se trata de la clave más grande.
3: //Como las claves tienen la forma tupla(<Gasto, Persona>), conocemos que el más grande será la tupla con el mayor
4: //Gasto y el mayor ID, ya que el orden de las tuplas se calcula con la fórmula:
5: <a, b> > <c,d> $\leftrightarrow$ a > c $\vee$ (a = c $\wedge$ b > d)
6:
7: //El significado de colaPrioridadGastos es igual al ID de la persona que también se encuentra en su clave.
8: $res \leftarrow AnteriorSignificado(CrearItUlt(l.colaPrioridadGastos))$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$