Algoritmos y Estructuras de Datos II

Trabajo Práctico 2

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

Lollapatuza

Trabajo práctico 2: Diseño

Integrante	LU	Correo electrónico
Guarnaccio, Augusto	248/22	augustoguarnaccio@gmail.com
Fernández Zaragoza, Mariano	416/22	marianofzaragoza@gmail.com
Kaufmann, Tadeo	505/22	tadeokaufmann1@gmail.com
Barracchia, Azul	178/22	azulbarracchia@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega	Maximiliano Martino	Aprobado (-)
Segunda entrega		

Para el TP3 van a tener que arreglar lo que se marca en las correcciones.

1. Lollapatuza

Interfaz

se explica con: LOLLAPATUZA

géneros: `lollapatuza`.

Operaciones básicas de Lollapatuza

CREARLOLLA(**in** *puestos* : `diccLog(PuestoID, puesto)`, **in** *personas* : `conj(persona)`) → *res* : `lolla`
✓**Pre** ≡ {vendenAlMismoPrecio(significados(puestos)) ∧ NoVendieronAún(significados(puestos)) ∧ ¬∅?(personas) ∧ ¬∅?(puestos)}
✓**Post** ≡ {res $=_{obs}$ CrearLolla(puestos, personas)}
Complejidad: $O(P * I * log(I) + A * log(A))$
Descripción: inicializamos un Lollapatuza con sus puestos, sus personas, un diccionario de precios, un diccionario vacío de cuánto gastó cada persona y una variable que va a almacenar a la persona que más dinero gastó (como todavía no se gastó simplemente almacenamos a la de menor ID).
Aliasing: se devuelve res como referencia modificable.

REGISTRARCOMPRA(**in/out** l : `lolla`, **in** i : `item`, **in** *cant* : `cantidad`, **in** *per* : `persona`, **in** id : `PuestoID`)
✓**Pre** ≡ {l = l_0 ∧ def?(p, puestos(l)) ∧ per ∈ personas(l) ∧ haySuficiente?(obtener(p, puestos(l)), i, cant)}
✓**Post** ≡ {l $=_{obs}$ vender(l_0 , id, per, i, cant) $\wedge_L$
(∀ *per*2: persona)(per2 ∈ personas(l) ⇒ L gastoTotal(l, per2) $=_{obs}$ gastoTotal(vender(l_0 , id, per, i, cant))) ∧
masGasto(l) $=_{obs}$ masGasto(vender(l_0, id, per, i, cant) ∧
(∀ i2: item)(menorStock(l, i2) $=_{obs}$ menorStock(vender(l_0, id, per, i, cant), i2}
Complejidad: $O(log(A) + log(I) + log(P) + log(Cant))$
Descripción: registra la compra de una cantidad de un ítem particular, realizada por una persona en un puesto.
Aliasing: no devuelve nada, solo modifica nuestra estructura.

HACKEAR(**in/out** l : `lolla`, **in** i : `item`, **in** *per* : `persona`)
✓**Pre** ≡ {l = l_0 ∧ ConsumioSinPromoEnAlgunPuesto(l, per, i)}
✓**Post** ≡ {l $=_{obs}$ hackear(l_0, per, i)}
Complejidad: O(log(A) + log(I) + log(P))
Descripción: se hackea el puesto de menor ID en el que la persona haya consumido ese item sin promocion
Aliasing: no devuelve nada, solo modifica nuestra estructura.

OBTENERGASTOTOTALDEUNAPERSONA(**in** l : `lolla`, **in** *per* : `persona`) → *res* : `dinero`
✓**Pre** ≡ {per ∈ personas(l)}
✓**Post** ≡ {res $=_{obs}$ gastoTotal(l,per)}
Complejidad: $O(log(A))$
Descripción: dada una persona y un lolla devuelve el gasto total de esa persona.
Aliasing: devuelve por referencia no modificable.

PERSONAQUEMÁSGASTÓ(**in** l : `lolla`) → *res* : `persona`
✓**Pre** ≡ {true}
✓**Post** ≡ {res $=_{obs}$ masGastó(l)}
Complejidad: $O(1)$
Descripción: dado un lolla nos devuelve la persona que más gastó.
Aliasing: devuelve res como referencia no modificable.

PUESTOCONMENORSTOCKDEITEM(**in** l : `lolla`, **in** i : `item`) → *res* : `puestoID`
✓**Pre** ≡ {true}
✓**Post** ≡ {res $=_{obs}$ menorStock(l,i)}
Complejidad: $O(P * log(I))$
Descripción: Devuelve el Puesto con menor stock para cierto ítem, si varios puestos tienen la misma cantidad mínima de stock, entonces se decide por el ID más bajo. Si el ítem no está en ningún menú de un puesto se devuelve el puesto con menor ID. Para chequear eso nos fijamos si está definido o no en precios cuyas claves son todos los ítems del lolla.
Aliasing: se devuelve res como referencia no modificable.

ObtenerPersonas(**in** l : `lolla`) → res : `conj(persona)`
✓**Pre** ≡ {true}
✓**Post** ≡ {res $=_{obs}$ personas(l)}
Complejidad: $O(1)$
Descripción: dado un lolla nos devuelve el conjunto de personas que asistieron.
Aliasing: devuelve por referencia no modificable.

ObtenerPuestos(**in** l : `lolla`) → res : `diccLog(puestoID, puesto)`
✓**Pre** ≡ {true}
✓**Post** ≡ {res $=_{obs}$ puestos(l)}
Complejidad: $O(1)$
Descripción: dado un lolla nos devuelve el conjunto de puestos que lo componen.
Aliasing: devuelve por referencia no modificable.

Operaciones auxiliares de Lollapatuza

PreciosLolla(**in** *puestos* : `diccLog(puestoID, puesto)`) → res : `diccLog(ítem, dinero)`
✓**Pre** ≡ {true}
✓**Post** ≡ {(∀ p: puesto) (p ∈ significados(puestos) ⇒ L menu(p) ⊆ claves(res)) ∧
(∀ i: item)(i ∈ claves(res) ⇒ L (∃p: puesto)(p ∈ significados(puestos) $\wedge_{L}$
i ∈ menu(p) ∧ precio(p, i) $=_{obs}$ obtener (i, res)))}
Complejidad: $O(P * I * log(I))$
Descripción: recorre el diccionario de precios de cada puesto y se fija si cada uno de los ítems de su menú está definido en el diccionario de precios del Lolla. Si no está definido lo define. Si está definido, sigue recorriendo.
Aliasing: devuelve res como referencia no modificable

PersonaConMenorID(**in** *personas* : `conj(personas)`) → res : `persona`
✓**Pre** ≡ {¬∅?(personas)}
✓**Post** ≡ {¬(∃p: persona)(p ∈ personas ⇒ p < res)}
Complejidad: $O(A)$
Descripción: Devuelve la persona con menor ID entre un conjunto de personas.
Aliasing: devolvemos por referencia no modificable.

GastosEnCero(**in** *personas* : `conj(persona)`) → res : `diccLog(persona, dinero)`
✓**Pre** ≡ {true}
✓**Post** ≡ { claves(res) $=_{obs}$ personas $\wedge_{L}$ (∀ p: personas)(p ∈ personas ⇒ L obtener(p,res) $=_{obs}$ 0)}
Complejidad: $O(A * log(A))$
Descripción: genera un diccionario de gastos cuyas claves son todas las personas y sus significados son todos los gastos en 0.
Aliasing: devuelve res por referencia modificable

CrearPersonaPorGastos(**in** *personas* : `conj(persona)`) → res : `diccLog(dinero, diccLog(persona, dinero))`
✓**Pre** ≡ {true}
✓**Post** ≡ {claves(obtener(0,res))$=_{obs}$ personas ∧ claves(res) = 1}
Complejidad: $O(A * log(A))$
Descripción: se inicializa un diccionario donde las claves van a representar el gasto y los significados, las personas asociadas a esos gastos. En este caso devuelve un diccLog(dinero, diccLog(persona, dinero)) pero en diccLog(persona, dinero) dinero siempre va a ser 0 puesto que no nos interesa su significado solo nos interesa la forma del diccLog. También decidimos llamarlo dinero y no escribir directamente 0 o un valor cualquiera para reutilizar la operación iGastosEnCero.

Aliasing: devuelve res como referencia modificable

Representación

Representación del Lollapatuza

`lollapatuza` se representa con `estr`

donde `estr` es `tupla`(*Precios*: `diccLog(Item, Dinero)`
, *Puestos*: `diccLog(PuestoID, Puesto)`
, *Personas*: `conj(Persona)`
, *PersonaQueMásGastó*: `Persona`
, *GastoTotalPorPersona*: `diccLog(Persona, Dinero)`
, *HackeosPosibles*: `diccLog(item, diccLog(persona, diccLog(puestoID, Puesto)))`
, *PersonaPorGasto*: `diccLog(dinero, diccLog(persona, dinero))`)

Rep : estr $\longrightarrow$ bool

✓Rep(e) $\equiv$ true $\iff$ 1 $\wedge_{\text{L}}$ 2 $\wedge$ 3 $\wedge$ 4 $\wedge$ 5 $\wedge$ 6 $\wedge$ 7 $\wedge$ 8 $\wedge$ 9 $\wedge$ 10 $\wedge$ 11 $\wedge$ 12 $\wedge$ 13 $\wedge$ 14 $\wedge$ 15

✓1 $\equiv$ ($\forall$ p: puesto)(p $\in$ puestos(l) $\Rightarrow$ claves(p.precios) $\subseteq$ (claves(e.Precios))

✓2 $\equiv$ claves(e.HackeosPosibles) $\subseteq$ claves(e.precios)

✓3 $\equiv$ ($\forall$ $p1, p2$: puesto)($\forall$ i: item)($\forall$ c: cant)(def?(i, π1(DescuentoPorCantidadPorItem)) $\wedge_{\text{L}}$ def?(c, obtener(i, π(DescuentoPorCantidadPorItem)) (def?(i, π2(DescuentoPorCantidadPorItem)) $\wedge_{\text{L}}$ def?(c, obtener(i, π2(DescuentoPorCantidadPorItem)) $\Rightarrow$ obtener(c, obtener(i, π1(DescuentoPorCantidadPorItem)) = obtener(c, obtener(i, π2(DescuentoPorCantidadPorItem))

✓4 $\equiv$ ($\forall$ p: puesto)(claves(p.GastosPorPersona) $\subseteq$ claves(e.Personas))

✓5 $\equiv$ e.PersonaQueMasGasto $\in$ e.Personas

✓6 $\equiv$ claves(e.GastoTotalPorPersona) = e.Personas

✓7 $\equiv$ ($\forall$ i: item)($\forall$ per: persona)(def?(i, e.HackeosPosibles) $\wedge_{\text{L}}$ def?(per, obtener(i, e.HackeosPosibles)) $\Rightarrow$ obtener(per, obtener(i, e.hackeosPosibles)) $\in$ e.Personas

✓8 $\equiv$ ($\forall$ d: dinero)(d $\in$ claves(PersonaPorGasto) $\Rightarrow$ L claves(obtener(e.PersonaPorGasto, d)) $\subseteq$ personas(e))

✓9 $\equiv$ ($\forall$ per: persona)(per $\in$ claves(GastoTotalPorPersona) $\Rightarrow$ obtener(GastoTotalPorPersona, per) < obtener(GastoTotalPorPersona, e.PersonaQueMasGasto) v ((obtener(GastoTotalPorPersona, per) = obtener(GastoTotalPorPersona, e.PersonaQueMasGasto)) y e.PersonaQueMasGasto < per

✓10 $\equiv$ ($\forall$ per: persona)(per $\in$ claves(GastoTotalPorPersona) $\Rightarrow$ obtener(e.GastoTotalPorPersona, per) = sumaGastosPorPersona(per, puestos(e))

✓11 $\equiv$ ($\forall$ d: dinero)($\forall$ per: persona)(per $\in$ claves(e.GastoTotalPorPersona) $\wedge_{\text{L}}$ d = obtener(per, e.GastoTotalPorPerosona) $\iff$ d $\in$ claves(e.PersonaPorGasto) $\wedge_{\text{L}}$ per $\in$ claves(obtener(d, e.PersonaPorGasto))

✓12 $\equiv$ ($\forall$ i: item)(def?(i, e.HackeosPosibles) $\Rightarrow$ L ($\forall$ per: persona)(def?(per, obtener(i, e.HackeosPosibles)) $\Rightarrow$ L ($\forall$ pID: puestoID)(def?(pID, obtener(per, obtener(i, e.HackeosPosibles))) $\Rightarrow$ L def?(per, obtener(pID, obtener(per, obtener(i, e.HackeosPosibles))).VentasSinDescuento) $\Rightarrow$ L def?(i, obtener(per, obtener(pID, obtener(per, obtener(i, e.HackeosPosibles))).ventas))

✓ 13 ≡ (∀ *d*: dinero)(def?(d, e.PersonaPorGasto) ⇒ L (∀ *per*: persona)(def?(per, obtener(d, e.PersonaPorGasto)) ⇒ L obtener(per, obtener(d, e.PersonaPorGasto)) $=_{obs}$ 0)

✓ 14 ≡ (∀ *i*: item)(∀ *per*: persona)(def?(i, e.HackeosPosibles) $\wedge_L$ def?(per, obtener(i, e.HackeosPosibles ⇒ claves(obtener(per, obtener(i, e.HackeosPosibles) ⊆ claves(e.Puestos))

✓ 15 ≡ (∀ *i*: item)(∀ *per*: persona)(∀ *ID*: puestoID)(def?(i, e.HackeosPosibles) $\wedge_L$ def?(per, obtener(i, e.HackeosPosibles)) $\wedge_L$ def?(ID, obtener(per, obtener(i, e.HackeosPosibles))) ⇒ obtener(ID, obtener(per, obtener(i, e.HackeosPosibles))) $=_{obs}$ obtener(ID, e.Puestos))

✓ SumaGastosPorPersona: persona per x conj(puestos) ps ⇒ dinero

✓ SumaGastosPorPersona(per,ps) ≡ if vacio?(ps) 0 else obtener (per, DameUno(ps).GastoPorPersona) + SumaGastosPorPersona(per, SinUno(ps))

✓ Abs : estr e ⟶ lolla {Rep(e)}

✓ Abs(e) ≡ l : lolla /
puestos(l) =obs e.Puestos ∧
personas(l) =obs e.Personas

Algoritmos

✓ **iCrearLolla**(**in** *puestos* : `diccLog(puestoID, puesto)`, **in** *personas* : `conj(persona)`) → *res* : *lolla*

1. res.Precios ← iPreciosLolla(puestos)
2. res.Puestos ← puestos
3. res.Personas ← personas
4. res.PersonaQueMásGastó ← iPersonaConMenorID(personas)
5. res.GastoTotalPorPersona ← iGastosEnCero(personas)
6. res.HackeosPosibles ← Vacío()
7. res.PersonaPorGasto ← iCrearPersonaPorGasto

✓ Complejidad: $O(P * I * log(I) + A * log(A))$

✓ Justificación: Se usa la función iPreciosLolla que tiene complejidad O(P * I * log(I)). Se copian los parámetros de entrada, para el diccLog(PuestoID, puesto) lleva tiempo de ejecucion: cantidad de puestos por copiar un puesto donde no se haya vendido, o sea O(P*I*Cant), y copiar personas tiene complejidad de O(A). Además se llama a iPersonaConMenorID con complejidad O(A), iGastosEnCero con O(A*log(A)), y iCrearPersonaPorGasto con O(Log(A)*A). Inicializar un diccLog cuesta O(1). En total la complejidad es O(P * I * log(I) + P*I*Cant + A + A + A*log(A) + Log(A)*A) = O(P * I * log(I) + P*I*Cant + Log(A)*A)

✓ Podemos afirmar que cumple con el contrato porque para una entrada que cumple con la pre, el algoritmo se comporta como la especificación de crearLola(puestos,personas). Para esos parametros, devuelve una estructura que permite mediante la aplicación de la función de abstracción vincularla con alguna instancia válida de nuestro TAD.

---

iRegistrarCompra(**in/out** *l* : `lolla`, **in** *i* : `item`, **in** *cant* : `cantidad` **in** *per* : `persona` **in** *id* : `PuestoID`)

```
puesto ← Significado(l.Puestos, id)
iVender(puesto, per, i, cant)
gastoAnterior ← Significado(l.GastoTotalPorPersona, per)
gastoTotal ← gastoAnterior + iGastoDeVenta(puesto, i, cant)
diccPersonas ← significado (l.PersonaPorGasto, gastoAnterior)
si #claves (diccPersonas) == 1 entonces
    Borrar(l.PersonaPorGasto, gastoAnterior)
else
    Borrar(diccPersonas,per)
si ¬ Definido?(l.PersonaPorGasto,gastoTotal entonces
    diccPersonasAgregar ← Definir (vacío(), per, 0)
    Definir(l.PersonaPorGasto, gastoTotal, diccPersonasAgregar)
else
    Definir(significado(l.PersonaPorGasto, gastoTotal), per , 0)
Definir(l.GastoTotalPorPersona, per, gastoTotal)
si Significado(l.GastoTotalPorPersona, l.PersonaQueMasGasto) < gastoTotal || (Significado(l.GastoTotalPorPersona, l.PersonaQueMasGasto) == gastoTotal && per <l.PersonaQueMasGasto)) entonces
    l.PersonaQueMásGastó ← per
si cant - iCantidadConDescuento(p, i, cant) ≠ 0 entonces
    si ¬ definido?(l.hackeosPosibles, per) entonces
        definir(l.HackeosPosibles, per, Vacio())
        definir(significado(l.hackeosPosibles, per), i, vacío())
        definir(significado(significado(l.hackeosPosibles, per)), i), id, puesto)
    si ¬ definido(significado(l.hackeosPosibles, per), i entonces
        definir(significado(l.hackeosPosibles, per), i, vacío())
        definir(significado(significado(l.hackeosPosibles, per)), i), id, puesto)
    si ¬ definido((significado(significado(l.hackeosPosibles, per)), i), id entonces
        definir(significado(significado(l.hackeosPosibles, per)), i), id, puesto)
```

Tienen definido distinto la estructura

✗ Falta el caso en que si esta todo definido, y se compran mas del mismo item sin descuento para el mismo puesto-persona

✓ Complejidad: $O(log(A) + log(I) + log(p) + log(cant))$

✓ Justificación:Usamos Borrar , Definir, Significado que todas tienen como complejidad O(log(#claves)) en nuestro caso puede ser personas, items, puestos, o dinero Pero todos se hacen por separado asi que se suman. En el caso de dinero, es cuando lo usamos sobre el diccionario PersonaPorGasto y la cantidad de claves como maximo es la cantidad de personas. O(log(A)) + O(log(I)) + O(log(P). En el caso particular que definimos en HackeosPosibles y ponemos como significado un puesto lo pasamos por referencia ahorrandonos el costo de copiarlo. Usamos funciones auxiliares: iVender que cuesta O(log(I)+log(A)+ log(cant)) , iGastoDeVenta cuesta O(log(I) + log(cant)) y CantidadConDescuento que tiene complejidad O(log(I) + log(cant)). En total la complejidad queda O(log(A) + log(I) + log(P) + log(Cant))

✓Usando la misma precondición que en vender del TAD lolla, este algoritmo se comporta igual que ese generador.

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

```
1:  itPuestoAHackear ← CrearIt(Significado(Significado(l.HackeosPosibles, per)), i))
2:  puestoAHackear ← itPuestoAHackear.SiguienteSignificado
3:  IDPuestoAHackear ← itPuestoAHackear.SiguienteClave
4:  itVentasSinDescuento ← Significado(Significado(puestoAHackear.VentasSinDescuentos, persona), i)
5:  si (itVentasSinDescuento.Siguiente).cantidad == 1 entonces
6:      Borrar(Significado(Significado(l.HackeosPosibles, per)), i), itPuestoAHackear.SiguienteClave)
7:      gastoAnterior ← Significado(l.GastoTotalPorPersona, per)
8:      diccPersonas ← Significado(l.PersonaPorGasto, gastoAnterior)
9:  si #Claves(diccPersonas) == 1 entonces
10:     Borrar(l.PersonaPorGasto, gastoAnterior)
11: else
12:     Borrar(diccPersonas, per)
13: gastoActualizado ← Significado(l.GastoTotalPorPersona, per) - iObtenerPrecio(PuestoAHackear, i)
14: Definir(l.GastoTotalPorPersona, per, gastoActualizado)
15: si ¬ Definido?(l.PersonaPorGasto, gastoActualizado) entonces
16:     diccPersonasAgregar ← Definir(Vacío(), per, 0)
17:     Definir(l.PersonaPorGasto, gastoActualizado, diccPersonasAgregar)
18: else
19:     Definir(Significado(l.PersonaPorGasto, gastoActualizado), per, 0)
20: iHackearPuesto(i, per, PuestoAHackear)
21: si l.PersonaQueMasGasto == per entonces
22:     itGastoMax ← CrearUltIt(l.PersonaPorGasto)
23:     diccPersonasQueMasGastaron ← Significado(l.PersonaPorGasto, itGastoMax.anteriorClave)
24:     itPersonaMenorID ← CrearIt(diccPersonasQueMasGastaron)
25:     l.PersonaQueMasGasto ← itPersonaMenorID.SiguienteClave
```

Mal identado, esta adentro o fuera del if-else?

Complejidad: $O(log(A) + log(I))$

Justificación:Al principio se crea un iterador al ultimo diccionario de hackeos posibles que es el diccionario que dado un PuestoID devuelve un puesto por referencia(al puesto que esta en el diccionario puestos de Lola). Al crear el iterador se crea en la clave más chica es decir en el PuestoID menor que es lo que se pide para hackear. Luego se guarda esa referencia en una variable llamada PuestoAHackear, como la referencia es al diccionario puestos de lola en caso de que se elimine la clave PuestoID de puestos a hackear (que ocurre si se hackea un puesto que solo tiene 1 item vendido sin descuento) la variable PuestoAHackear sigue apuntando a la clave,significado del diccionario Puestos. Luego se accede al iterador que esta en VentasSinDescuento para ver si la cantidad del item a Hackear es 1. Para acceder si usa significado 2 veces primero sobre un diccionario cuyas claves son del primero persona y del segundo item por lo tanto cuesta O(log(p)) + O(log(I)) .Si la cantidad es 1 se borra el puesto del diccionario puestoHackeables como dijimos antes que podia ocurrir. Usamos Borrar , Definir, Significado que todas tienen como complejidad O(log(#claves)) en nuestro caso puede ser personas, items o dinero. En el caso de dinero, es cuando lo usamos sobre el diccionario PersonaPorGasto y la cantidad de claves como maximo es la cantidad de personas. Entonces las complejidades seria O(log(A)) + O(log(I)) y siempre se suman. Luego se llama a iObtenerPrecio cuya complejidad es O(log(I)). También se llama a iHackearPuesto que tiene complejidad O(log(A) +log(I)). Todas las operaciones con iteradores cuestan O(1). La complejidad total seria O(log(A) + log(I))

Se comporta igual que hackear del tad lollapatuza. Siempre que se haya vendido en algun puesto sin descuento, se hackea el puesto de menor ID donde haya comprado ese item esa persona pasados por parametro. Como dijimos en el 1er parrafo de la justificacion de complejidad, hackeamos un puesto que puede ser hackeable(esta en HackeosPosibles) y usamos el de menorID. Luego al llamar a iHackearPuesto se modifican los parametros correctos del PuestoAHackear que es una referencia al puesto que esta en el diccionario Puestos. Luego se modifican las otras estructuras de Lolla como GastoTotalPorPersona, PersonaPorGasto, PersonaQueMasGasto, PuestosHackeables.

iObtenerGastoTotalDeUnaPersona(**in** l : `lolla`, **in** per : `persona`) → res : $dinero$

1: res ← Significado(l.GastoTotalPorPersona, per)

Complejidad: $O(log(A))$
Justificación: La complejidad es O(Log(A)) ya que usamos la operación Significado del módulo Diccionario Logarítmico que tiene complejidad O (log(#claves)). En el peor caso posible un puesto puede tener definido el gasto de todas las personas que asisten al lollapatuza, lo que se representa con A.

El algoritmo se comporta como la especificación de gastoTotal(l, per). Para una persona que está entre aquellas que asistieron al lolla nos devuelve todo lo que lleva gastado. Si no compró nada el código devuelve res = 0.

---

iPersonaQueMásGastó(**in** l : `lolla`) → res : $persona$

1: res ← l.PersonaQueMasGasto

Complejidad: $O(1)$
Justificación: La complejidad es O(1) ya que simplemente retornamos una de las partes de nuestra estructura. Como es algo que ya tenemos almacenado y, por lo tanto precalculado, el costo de la operación es el costo del return. En este caso como devolvemos un nat el costo de hacerlo por copia o referencia es el mismo, O(1).

El algoritmo se comporta como la especificación de másGastó(l). Para una persona que está entre aquellas que asistieron al lolla nos devuelve la que menos gastó. En caso de haber empate, se devuelve la de menor ID. Si todavía nadie gastó nada se devuelve una cualquiera, en este caso la de menor ID.

---

iPuestoConMenorStockDeItem(**in** l : `lolla`, **in** i : `item`) → res : ID

1: **si** Definido?(i, l.Precios) **entonces**
2: it ← CrearIt(l.puestos)
3: resPuesto ← it.SiguienteSignificado
4: res ← it.SiguienteClave
5: **mientras** it.HaySiguiente **hacer**
6: **si** Definido?(i, resPuesto.StockPorItem) **entonces**
7: **si** Significado(it.SiguienteSignificado.StockPorItem, i) < Significado(resPuesto.StockPorItem, i) ∨ (Significado(it.SiguienteSignificado.StockPorItem, i) == Significado(resPuesto.StockPorItem, i) ∧ it.SiguienteClave <res)) **entonces**
8: resPuesto ← it.SiguienteSignifcado
9: res ← it.SiguienteClave
10: **else**
11: resPuesto ← it.SiguienteSignifcado
12: res ← it.SiguienteClave
13: it.Avanzar
14: **else**
15: itPuestoMenorID ← CrearIt(l.Puestos)
16: res ← itPuestoMenorID.SiguienteClave

Complejidad: $O(P * log(I))$
Justificación: Usamos Definido? y Significado que todas tienen como complejidad O(log(#claves)) en nuestro caso son items. O(log(I). Se hace un ciclo donde se itera por todos todos los puestos y como en ese ciclo hay operaciones mencionadas la complejidad de esto es O(P * Log(I)). Las operaciones básicas de iteradores cuestan O(1). Entonces la complejidad total es O(P * log(I))

El algoritmo cumple el contrato porque se comporta como la funcionAuxiliar MenorStock del Tad lollapatuza.

---

iObtenerPersonas(**in** l : `lolla`) → res : $conj(persona)$

1: res ← l.Personas

Complejidad: $O(1)$

Justificación: La complejidad es O(1) ya que simplemente retornamos una de las partes de nuestra estructura. Como es algo que ya tenemos almacenado y, por lo tanto precalculado, el costo de la operación es el costo del return. En este caso como devolvemos un conjunto, para no pagar el costo de copiarlo entero lo devolvemos como referencia lo que convierte el costo de la operación en O(1).

El algoritmo se comporta como la especificación de personas(l). Para un lolla nos devuelve el conjunto de personas que asistieron.

---

iObtenerPuestos(**in** l : `lolla`) → res : $diccLog(ID, puesto)$

1: res ← l.Puestos

Complejidad:$O(1)$

Justificación: La complejidad es O(1) ya que simplemente retornamos una de las partes de nuestra estructura. Como es algo que ya tenemos almacenado y, por lo tanto precalculado, el costo de la operación es el costo del return. En este caso como devolvemos un conjunto, para no pagar el costo de copiarlo entero lo devolvemos como referencia lo que convierte el costo de la operación en O(1).

El algoritmo se comporta como la especificación de puestos(l). Para un lolla nos devuelve el conjunto de puestos que asistieron.

---

iPreciosLolla(**in** $puestos$: `diccLog(puestoID, puesto)`) → res : $diccLog(item, dinero)$

```
1: itPuestos ← CrearIt(puestos)
2: res ← Vacio()
3: mientras itPuestos.HaySiguiente hacer
4:     itItems ← itItems.Crear(itPuestos.SiguienteSignificado.precios)
5:     mientras itItems.HaySiguiente hacer
6:         si ¬ Definido?(itItem.SiguienteClave, res) entonces
7:             Definir(res, itItem.SiguienteClave, itItem.SiguienteSignificado)
8:         itItems.Avanzar
9:     itPuestos.Avanzar
```

Complejidad: $O(P * I * log(I))$

Justificación: Este algoritmo utiliza varias operaciones con iteradores y todas cuestan O(1). Con un iterador a diccionario puestos, se hace un ciclo que recorre todos los puestos y en cada puesto se hace un ciclo interno donde itera sobre el diccionario precios cuyas claves son todos los items. También en este ciclo interno se define res que es un dicc que tiene items como claves y esto tiene una complejidad de O(log(I)). Por lo tanto la complejidad total es O(P)*(O(I) * O(log(I))) = O(P * I * log(I))

Se cumple que todos los items que estan en menu de puestos estan incluidos en claves de res ya que en el algo se itera por todos los ítems de todos lo puestos y se definen como claves. Luego se cumple que para todos los items existe algún puesto que tiene a ese item en menu y además el preguntar por su precio en menu del puesto es igual a preguntar el precio en la estructura de lola porque cada vez que definimos el significado del diccionario res lo definimos como el precio que tiene en ese menu.

---

iPersonaConMenorID(**in** *personas* : `conj(persona)`) → *res* : persona

```
1: it ← CrearIt(personas)
2: res ← it.Siguiente
3: mientras it.HaySiguiente hacer
4:     si it.Siguiente < res entonces
5:         res ← it.Siguiente
6: it.Avanzar
7:
```

Complejidad: $O(A)$

Justificación: La complejidad es O(A) ya que recorremos todo el conjunto de personas que asistieron. Como es un conjunto tenemos que iterar hasta el final para saber cuál es la de menor ID, por lo que la complejidad termina por ser el cardinal del conjunto que en este caso es A. Asumimos que el costo de las operaciones del iterador es O(1).

Cumple con el contrato porque para un conjunto no vacío de personas (no podemos tener un lolla vacío) nos devuelve sí o sí aquella que tiene el menor ID posible de todos.

---

iGastosEnCero(**in** *personas* : `conj(persona)`) → *res* : diccLog(persona, dinero)

```
1: it ← CrearIt(personas)
2: res ← Vacío()
3: mientras it.HaySiguiente hacer
4:     Definir(res, it.siguiente, 0)
5:     it.Avanzar
```

Complejidad: $O(A * log(A))$

Justificación: El algoritmo recorre personas todas las personas y las define como clave del diccionario GastoPorPersona con significado 0. Como recorre todas las personas es O(A) y mientras lo hace define y esta operación cuesta O(log(#claves)). Como en el peor caso están todas las personas seria O(log(A)) hecho A veces. Por lo tanto, la complejidad seria O(A*log(A)). Asumimos que el costo de las operaciones del iterador es O(1).

Lo cumple ya que las claves del diccionario creado son las personas y los significados del diccionario devuelto son todo 0. =0

---

iCrearPersonaPorGastos(**in** *personas* : `conj(persona)`) → *res* : diccLog(dinero, diccLog(persona,dinero))

```
1: res ← vacío()
2: Definir(res, 0, iGastosPorPersona(persona))
```

Complejidad: $O(A * log(A))$

Justificación: Como en este algoritmo se llama al algoritmo iGastosEnCero, la complejidad va a ser O(Log(A) * A) más la complejidad de definir en un diccionario lineal que es O(Log(claves)) = O(Log(A)). Por lo tanto la complejidad es O(Log(A)*A)

Cumple la post debido a que la unica clave del diccionario devuelto es 0 y el significado de esa clave tiene un diccionario que sus claves son todas las personas que hay.

2. Puesto De Comida

Interfaz

se explica con: PuestoDeComida

géneros: `puesto`

Operaciones básicas de Puesto de comida

CrearPuesto(**in** *precios*: `diccLog(Item, dinero)`, **in** *stock*: `diccLog(item, cantidad)`, **in** *descuentos*: `diccLog(item, diccLog(cantidad, descuento))`) → *res* : `puesto`
✓**Pre** ≡ {claves(stock) $=_{obs}$ claves(precios) $\wedge_L$ claves(descuentos) $\subseteq$ claves(precio)}
✓**Post** ≡ {res $=_{obs}$ crearPuesto(precios, stock, descuentos)}
Complejidad: O(I ∗ Cant)
Descripción: inicializamos un puesto con todos los items, con sus respectivos descuentos, cantidades con descuentos, precios y stocks, y con gastos por persona y ventas vacíos.
Aliasing: devolvemos res por copia.

ObtenerElStock(**in** *p*: `puesto`, **in** *i*: `item`) → *res* : `cantidad`
✓**Pre** ≡ { i ∈ menu(p)}
✓**Post** ≡ {res $=_{obs}$ stock(p, i)}
Complejidad: O(log(I)
Descripción: dado un ítem que está definido en el menú de un puesto nos devuelve el stock de dicho ítem.
Aliasing: devolvemos res como una referencia modificable.

ObtenerDescuento(**in** *p*: `puesto`, **in** *i*: `item`, **in** *cant*: `cantidad`) → *res* : `descuento`
✓**Pre** ≡ {i ∈ menu(p)}
✓**Post** ≡ {res $=_{obs}$ descuento(p,i,cant))}
Complejidad: O(log(I) + log(Cant))
Descripción: dado un ítem y una cantidad se devuelve el descuento asociado a ese ítem. Si para esa cantidad no hay descuento entonces busca la cantidad más cercana para la que hay descuento. Esta cantidad más cercana debe ser menor a la pasada por parámetro.
Aliasing: devolvemos res como referencia no modificable.

ObtenerGasto(**in** *p*: `puesto` **in** *per*: `persona`) → *res* : `nat`
✓**Pre** ≡ {true}
✓**Post** ≡ {res $=_{obs}$ gastosDe(p, per)}
Complejidad: O(log(A))
Descripción: dado un puesto y una persona nos devuelve el gasto de esa persona en el puesto. Como gastosPorPersona se inicializa vacío, si la persona no está definida en ese diccionario quiere decir que no compró nada y por lo tanto devuelve 0.
Aliasing: devolvemos res como referencia no modificable.

✓**Operaciones auxiliares de Puesto de Comida**

Vender(**in/out** *p*: `puesto`, **in** *per*: `persona`, **in** *i*: `item`, **in** *cant*: `cantidad`)
✓**Pre** ≡ { p = $p_0 \wedge$ haySuficiente?(p,i,cant)}
✓**Post** ≡ { menu(p0) $=_{obs}$ menu(p) $\wedge_L$
stock(p0,i) - c = stock(p,i) (∀ *i*2: item)(i2 ∈ menu(p) i2 ≠ i ⇒ L stock(p0,i2) = stock(p,i2)
gastosDe(vender(p0,per,i,cant), per) = gastosDe(p,per)
(∀ *i*2: item)(i2 ∈ menu(p) ⇒ L precio(p,i2) =obs precio(p0,i2))
(∀ *i*2: item)(∀ *c*: cant)(i2 ∈ menu(p) ⇒ L descuento(p0,i2,c) $=_{obs}$ descuento(p,i2,c)) ventas(p,i2) = ventas(vender(p0,per,i2,c)) }
Complejidad: O(log(I) + log(cant) + log(A))
Descripción: restamos la cantidad vendida al stock del dicc StockPorItem y agregamos el gasto de la compra al dicc gastoPorPersona. Agregamos el gasto de la compra al dicc gastoPorPersona y actualizamos Ventas y VentasSinDescuentos.
Aliasing: no devolvemos nada, se modifica el puesto con la ejecución de la función

ObtenerPrecio(**in** *p*: `puesto` **in** *i*: `item`) → *res* : `dinero`

✓**Pre** ≡ {i ∈ menu(p)}
✓**Post** ≡ {res =$_{obs}$ precio(p, i)}
Complejidad: O(log(I))
Descripción: dado un ítem nos devuelve el precio de ese ítem en el menú.
Aliasing: devolvemos res como referencia no modificable

TRANSFORMARACANTIDADESCONDESCUENTOS(**in** *descuentos*: `diccLog(item, diccLog(cantidad, descuento))`) → *res* : `diccLog(item, vector(cantidad))`
✓**Pre** ≡ {true}
Post ≡ {(∀ *i*: item)(∀ *c*: cant)(def?(i, descuentos) ⟺ def?(i,res))
✓(def?(i, descuentos) ∧$_L$ c ∈ claves(obtener(i, descuentos)) ⟺ def?(i, res) ∧$_L$ esta?(i, obtener(item, res))
(∀ *cantidades*: vector(cantidad))(∀ *n*: nat)(cantidades ∈ significados(res) 0 ≤ n < long(cantidades) - 1 ⇒ cantidades[n] <cantidades[n + 1])}
Complejidad: O(I ∗ Cant)
Descripción: Transforma el diccionario de entrada llamado descuentos a un nuevo diccionario que tiene como clave los ítems y como significado un vector con las cantidades que tienen descuentos ordenadas.
Aliasing: Devolvemos por referencia no modificable.

GASTODEVENTA(**in** *p*: `puesto`, **in** *i*: `item`, **in** *cant*: `cantidad`) → *res* : `puesto`
✓**Pre** ≡ {haySuficiente?(p,i,cant)}
✓**Post** ≡ {res =$_{obs}$ gastosDe1Venta(p, <i, cant>)}
Complejidad: O(log(I) + log(cant))
Descripción: se fija cuál es el máximo descuento para una cantidad de un ítem y devuelve el gasto de la venta teniendo en cuenta ese descuento. El descuento solo se usa una vez.
Aliasing: Devolvemos por referencia no modificable.

HACKEARPUESTO(**in/out** *p*: `puesto`, **in** *per*: `persona`, **in** *i*: `item`)
✓**Pre** ≡ {p = p_0 i ∈ menu(p) ∧$_L$ ConsumioSinPromo?(p,per,i)}
✓**Post** ≡ {menu(p) =$_{obs}$ menu(p0) ∧$_L$
(∀ *i2*: item)(i2 ∈ menu(p) ⇒ L precio(p,i2) =$_{obs}$ precio(p0,i2)
(∀ *i2*: item)(i2 ∈ menu(p) i2 ≠ i ⇒ L stock(p,i2) =$_{obs}$ stock(p0,i2)) stock(p,i) =$_{obs}$ stock(p0,i) + 1)
(∀ *i2*: item)(i2 ∈ menu(p) ⇒ L (∀ *per2*: persona)(ventas(p,per) =$_{obs}$ ventas(olvidarItem(p,per,i2),per2))
(∀ *i2*: item)(i2 ∈ menu(p) ⇒ L (∀ *c*: cant)(descuentos(p,i2,c) = descuentos(p0,i2,c)}
Complejidad: O(log(A) + log(I))
Descripción: Se Hackea un puesto, esto aumenta el stock en uno del item hackeado, adémas modifica el diccionario ventas restando en uno la cantidad vendida de ese item. Si solo se vendio ese item 1 vez sin descuento se borra la tupla que tiene al item y la cantidad vendida de ventas.
Aliasing: no devolvemos nada, se modifica el puesto con la ejecución de la función.

CANTIDADCONDESCUENTO(**in** *p*: `puesto`, **in** *i*: `item`, **in** *cant*: `cantidad`) → *res* : `cantidad`
✓**Pre** ≡ {true}
✓**Post** ≡ {descuento(p,i,cant) = descuento(p,i,res)
res ≠ 0 ⇒ L descuento(p,i,cant) ≠ descuento(p,i,res - 1)) v
res = 0 ∧$_L$ (∀ *c*: cant)(c ≤ cant ⇒ descuento(p,i,cant) = 0)}
Complejidad: O(log(I) + log(Cant))
Descripción: devuelve la cantidad máxima para la que hay descuento, que sea menor o igual a la cantidad pasada como parámetro.
Aliasing: Devolvemos por referencia no modificable.

COPIARPUESTO(**in** *p*: `puesto`) → *res* : `puesto`
✓**Pre** ≡ {true}
✓**Post** ≡ {res =obs p}
Complejidad: O(I * Cant + A * Cantidad de Ventas + A * I)

Descripción: genera una copia nueva de puesto

Representación

Representación del Puesto de comida

`puesto de comida` **se representa con** `puesto`

donde `puesto` es `tupla`(*StockPorItem*: `diccLog(item, cantidad)`
, *DescuentoPorCantidadPorItem*: `diccLog(item, diccLog(cantidad, descuento))`
, *GastoPorPersona*: `diccLog(persona, dinero)`
, *Precios*: `diccLog(item, dinero)`
, *Ventas*: `diccLog(persona, lista enlazada(<item: item, cantidad: cantidad>)`
, *VentasSinDescuentos*: `diccLog(persona, diccLog(item, itLista(<item, cantidad>)))`)

Rep : puesto $\longrightarrow$ bool

✓Rep(*puesto*) ≡ true $\iff$ **1** $\wedge_{\text{L}}$ **2** $\wedge$3 $\wedge$ 4 $\wedge$ 5 $\wedge$ 6 $\wedge$ 7 $\wedge$ 8 $\wedge$ 9 $\wedge$ 10 $\wedge$ 11 $\wedge$ 12 $\wedge$ 13 $\wedge$ 14 $\wedge$ 15 $\wedge$ 16

✓**1 ≡ claves(e.StockPorItem) = claves(e.Precios)**

✓**2 ≡ claves(e.CantidadesConDescuento) $=_{\text{obs}}$ claves(e.DescuentoPorCantidadPorItem)**

✓**3 ≡ claves(e.gastoPorPersona) $=_{\text{obs}}$ claves(e.Ventas)**

✓**4 ≡ claves(e.CantidadesConDescuento) $\subset$ claves(e.Precios)**

✓**5 ≡ claves(e.VentasSinDescuento) $\subset$ claves(e.Ventas)**

✓**6 ≡ ($\forall$ *per*: persona)(def?(per, e.Ventas) $\Rightarrow$ L ($\forall$ *t*: <item, cant>)((t $\in$ obtener(per, e.Ventas) $\Rightarrow$ L π1.t $\subset$ claves(e.Precios)))**

✓**7 ≡ ($\forall$ *per*: persona)(def?(per, e.Ventas) $\Rightarrow$ L (($\forall$ *i*: item)(i $\in$ (claves(obtener(per, e.VentasSinDescuento)) $\Rightarrow$ L ($\exists$ :: ccant)(<i,c>$\in$ obtener(per, e.Ventas))**

✓**8 ≡ ($\forall$ *i*: item)(def?(i, e.DescuentoPorCantidadPorItem) $\Rightarrow$ L ($\forall$ *c*: cant)(def?(c, obtener(i, e.DescuentoPorCantPorItem)) $\Rightarrow$ L 0 <obtener(c, obtener(i, e.DescuentoPorCantPorItem)) <100)**

✓**9 ≡ ($\forall$ *i*: item)(def?(i, e.DescuentosPorCantidadPorItem) $\Rightarrow$ 0 $\notin$ claves(obtener(i, e.DescuentosPorCantidadPorItem)) $\wedge_{\text{L}}$ ($\forall$ *c*: cant)(def?(c, obtener(i, e.DescuentosPorCantidadPorItem) $\Rightarrow$ i $\neq$ 0)**

✓**10 ≡ ($\forall$ *per*: persona)(def?(per, e.VentasSinDescuentos) $\Rightarrow$ def?(per, e.Ventas) $\wedge_{\text{L}}$ def?(per, e.Ventas) $\Rightarrow$ claves(e.GastosPorPersona))**

✓**11 ≡ ($\forall$ *per*: persona)($\forall$ *t*: <item, cant>)(def?(per, e.ventas) $\wedge_{\text{L}}$ t $\in$ obtener(per, e.ventas) $\Rightarrow$ π2 t $\neq$ 0)**

✓**12 ≡ ($\forall$ *per*: persona)(def?(per, e.Ventas) $\Rightarrow$ L ($\forall$ *t*: <item, cant>)((t $\in$ obtener(per, e.ventas) $\wedge_{\text{L}}$ (def?(obtener(π1(t), e.VentasSinDescuento), π1(t)) $\wedge_{\text{L}}$ (no existe)($\neg\exists i$: $t : itBi(< item, cant >)(it =_{\text{obs}} obtener(it, obtener(per, e.VentasSinDescuento)) \wedge_{\text{L}} siguiente(it) =_{\text{obs}} t)$ $\Rightarrow$ π2(t) $\in$ obtener(π1(t), e.cantidadesConDescuentos))**

✓**13 ≡ ($\forall$ *i*: item)(def?(i, e.DescuentoPorCantidadPorItem) $\Rightarrow$ L ($\forall$ *c*: cant)(def?(c, obtener(i, e.DescuentoPorCantPorItem)) $\Rightarrow$ L está?(obtener(i, e.CantidadesConDescuento)))**

✓**14 ≡ ($\forall$ *i*: item)(def?(i, e.Precios) $\Rightarrow$ obtener(i, e.Precios) $\neq$ 0)**

✓**15 ≡ ($\forall$ *i*: item)($\forall$ *per*: persona)($\forall$ *it*: itBi(<item, cant>)(siguiente(it) $\in$ obtener(per, e.Ventas) $\wedge_{\text{L}}$ π1(siguiente(it)) $=_{\text{obs}}$ i)**

✓**16 ≡ ($\forall$ *per*: persona)(def?(per, e.GastoPorPersona) $\Rightarrow$ obtener(per, e.GastosPorPersona) $=_{\text{obs}}$ SumaDeVentas(e, per, e.Ventas, e.VentasSinDescuento))**

✓($\forall$ *i*: item) ($\forall$ *p*: puesto) ($\forall$ *d*1: dicc(persona, secu(<item, cant>))) ($\forall$ *d*2, *d*: dicc(persona, dicc(item, itBi(<item, cant>))) ($\forall$ *s*: secu(<item, cantidad>)))

✓SumaDeVentas: puesto p x persona per x dicc(persona, secu(<item, cant>)) d1 x dicc(persona, dicc(item, itBi(<item, cant>)) d2 $\Rightarrow$ dinero

✓{per $\in$ claves(d1) $\wedge$ (def?(per, d2) $\Rightarrow$ L ($\forall$ *i*: item)(i $\in$ claves(obtener(per, d2)) $\Rightarrow$ L esta?(siguiente(obtener(i, obtener(per, d2))), obtener(per, d1))))))}

SumaDeVentas(p, per, d1, d2) ≡ if (¬ def?(per, d2)) then sumaSecuConDescuento(p, obtener(per, d1)) else sumaSecu(p, per, obtener(per, d1), obtener(per, d2))

sumaSecuConDescuento: puesto x secu(<item, cantidad>) ⇒ dinero

sumaSecuConDescuento(p, s) ≡ if vacía?(s) then 0 else aplicarDescuento(precio(π1(prim(s)), p), descuento(p, π1(prim(s)), π2(prim(s)) + sumaSecu(fin(s))

sumaSecu: puesto x persona x secu(<item, cant>) x diccLog(item,itBi(<item, cant>)) ⇒ dinero

sumaSecu(p, per, s, d) ≡ if vacío?(claves(d) then sumaSecuConDescuento(p, s) else (if def?(π1(prim(s)), d) then precio(π1(prim(s)), p) ∗ π2(prim(s))) + sumaSecu(p, per, fin(s), borrar(π1(prim(s)), d) else aplicarDescuento(precio(π1(prim(s)), p), descuento(p, π1(prim(s)), π2(prim(s))) + sumaSecu(p, per, fin(s), borrar(π1(prim(s)), d)

Abs : estr e ⟶ puesto {Rep(e)}

Abs(e) ≡ p : puesto /
menu(p) =obs claves(e.Precios) $\wedge_{\text{L}}$
(∀ i: item)(i ∈ $menu(p)$ ⇒ L [precio(p,i) $=_{\text{obs}}$ obtener(i, e.Precios) ∧
stock(p,i) $=_{\text{obs}}$ obtener(i, e.StockPorItem)] ∧
[(∀ c: cant)(descuento(p,i,c) $=_{\text{obs}}$ obtener(c, obtener(i, e.DescuentoPorCantidadPorItem))] ∧
(∀ per: persona)(diccionariosCantidadesPorItemMultiConj(ventas(p,per)) $=_{\text{obs}}$ diccionariosCantidadesPorItemSec(obtener(per, e.Ventas)))

(∀ mc: muticonj(<item, cant>)) (∀ i: item) (∀ s: secu(<item, cant>))

diccionariosCantidadesPorItemMultiConj : muticonj(<item, cant>) ⇒ dicc(item, cant) diccionariosCantidadesPorItemMultConj(mc) ≡ if ∅?(mc) then vacío else (if def?(π1(dameUno(mc))) then diccionariosCantidadesPorItemMultConj(sinUno(mc)) else definir(π1(dameUno(mc)), sumaDeCantidadesMc(π1(dameUno(mc)), mc), diccionariosCantidadesPorItemMultConj(sinUno(mc))))

sumaDeCantidadesMc : item x muticonj(<item, cant>) ⇒ cant sumaDeCantidadesMc(i, mc) ≡ if ∅?(mc) then 0 else sumaDeCantidadesMc(i, sinUno(mc)) + (if π1(dameUno(mc)) = i then π2(dameUno(mc)) else 0

diccionariosCantidadesPorItemSec : secu(<item, cant>) ⇒ dicc(item, cant) diccionariosCantidadesPorItemSec(s) ≡ if vacía(s) then vacío else (if def?(π1(prim(s))) then diccionariosCantidadesPorItemSec(fin(s)) else definir(π1(prim(s)), sumaDeCantidadesSec(π1(prim(s)), s), diccionariosCantidadesPorItemSec(fin(s))))

sumaDeCantidadesSec: item x secu(<item, cant>) ⇒ cant sumaDeCantidadesSec(i, s) ⇒ if vacía?(s) then

0 else sumaDeCantidadesSec(i, fin(s)) + (if $\pi 1$(prim(s)) = i then $\pi 2$(prim(s)) else 0

Algoritmos

---

iCrearPuesto(**in** *precios*: `diccLog(Item, dinero)`, **in** *stock*: `diccLog(item, cantidad)`, **in** *descuentos*: `diccLog(item, diccLog(cantidad, descuento))`) → *res* : puesto

1: res.StockPorItem ← stock
2: res.DescuentoPorCantidadPorItem ← descuentos
3: res.CantidadesConDescuento ← iTransformarACantidadesConDescuentos(descuentos)
4: res.GastoPorPersona ← Vacío()
5: res.Precios ← precios
6: res.Ventas ← Vacío()
7: res.VentasSinDescuentos ← Vacío()

Esto no aparece en la estructura

Complejidad: $O(I * Cant)$

Justificacion: Esta va a estar determinada por copiar cada una de las componentes a las distintas partes de nuestra estructura. En este caso, las operaciones de copia van a ser: la operación de copia de un diccLog y la operación iTransformarACantidadesConDescuentos(descuentos). Copiar un diccLog lleva O(claves* (Copiar(Clave) + Copiar(significado)). Copiar el parámetro de entrada Stock es O(I), copiar descuentos es O(I * Cant), copiar precios es O(I). En CantidadesConDescuento se llama a la funcion iTransfomarACantidadesConDescuentos(descuentos) que tiene un complejidad de O(I * Cant). Despues GastoPorPersona ,Ventas y VentasSinDescuento se inicializan en vacio, lo que tiene complejidad de O(1). Entonces la complejidad total es = O(I * Cant).

Podemos afirmar que cumple con el contrato porque para una entrada que cumple con la pre, el algoritmo se comporta como la especificación de crearPuesto(precios, stock, descuento). Para esos diccionarios, devuelve una estructura que permite mediante la aplicación de la función de abstracción vincularla con alguna instancia válida de nuestro TAD.

---

iObtenerElStock(**in** *p*: `puesto` **in** *i*: `item`) → *res* : nat

1: res ← Significado(p.StockPorItem, i)

Complejidad: $O(log(I))$

Justificacion: la complejidad es O(Log(I)) ya que usamos la operación Significado del módulo Diccionario Logarítmico que tiene complejidad O(log(#claves)). En el peor caso posible un puesto puede tener stock para todos los ítems del lollapatuza, lo que se representa con I.

El algoritmo se comporta como la especificación de stock(p, i). Siempre que entre un ítem que esté en menú (cosa de poder usar Significado sin que se indefina) devuelve correctamente el stock de ese ítem.

---

iObtenerDescuento(**in** p : `puesto` **in** i : `item` **in** $cant$: `cantidad`) → res : descuento

```
1: si Definido?(p.DescuentoPorCantidadPorItem, i) entonces
2:     diccDescuento ← Significado(p.DescuentoPorCantidadPorItem, item)
3:     cantConDescuento ← iCantidadConDescuento(p, i, cant)
4:     si cantConDescuento == 0 entonces
5:         res ← 0
6:     else
7:         res ← Significado(diccDescuento, cantConDescuento)
8: else
9:     res ← 0
```

No recibe estos parametros

Complejidad: $O(log(I) + log(Cant))$

Justificación: La complejidad es O(log (I) + log (Cant)). Esta va a estar determinada por la búsqueda del descuento. El peor caso posible sería el caso donde para un item el cual tiene definido una serie de descuentos, la cantidad pasada por parámetro no esté definida en diccLog (cantidad, descuento) y no solo no esté definida si no que esté acotada entre la cantidad mínima y la cantidad máxima para las cuales tenemos descuento. En este caso, chequea si el ítem está definido, obtiene el diccionario asociado a esa definición, chequea si la cantidad está definida al diccionario asociado a ese ítem, chequea si la cantidad es menor a la primera del vector, chequea si la cantidad es mayor a la última del vector, luego hace una búsqueda binaria sobre el vector, cuya longitud está acotada por Cant, y finalmente devuelve la cantidad más cercana y que precede a la pasada por parámetro. Expresado en término de complejidades tenemos: O(log (I)) + O(log(I)) + O(log(I)) + O(1) + O(1) + O(log(Cant))+ O(log(I)) = O(4log(I) + 2 + O(log(Cant))) = O(log(I)+ log(Cant)). La complejidad no se ve afectada por como se devuelve el resultado ya que al devolver un nat es lo mismo devolverlo por referencia o copia porque el costo es O(1).

Cumple con el contrato ya que para un item que está en el menú nuestro algoritmo se comporta como descuento(p, i, cant). Esto lo vemos ya que cumple con los 3 casos: devuelve 0 si la cantidad pasada por parámetro es 0 o el ítem no tiene descuento asociado, devuelve el descuento si la cantidad está definida en el diccLog(cant, descuento) o devuelve el descuento asociado a la cantidad más cercana a la pasada por parámetro (esta cantidad más cercana va a ser menor a la pasada por parámetro).

iObtenerGasto(**in** p : `puesto` **in** per : `persona`) → res : nat

```
1: si Definido?(p.GastosPorPersona, per)) entonces
2:     res ← Significado(p.GastosPorPersona, per)
3: else
4:     res ← 0
```

Complejidad: $O(log(I))$

Justificación: la complejidad es O(Log(A)) ya que usamos la operación Significado del módulo Diccionario Logarítmico que tiene complejidad O (log(claves)). En el peor caso posible un puesto puede tener definido el gasto de todas las personas que asisten al lollapatuza, lo que se representa con A.

El algoritmo se comporta como la especificación de gastoDe(p, per). El código devuelve res = 0 cuando la persona no está definida en el diccionario (en la especificación devuelve 0 si la venta de persona asociadas a ese puesto es vacía) y devuelve el valor del significado asociado a la clave persona cuando esta sí está definida.

iVender(**in**/**out** p: `puesto` **in** per: `persona` **in** i: `item` **in** c: `cantidad`)

```
1: Definir(p.StockPorItem, i, iObtenerElStock(p,i) - cant)
2: gastoTotal ← iGastoDeVenta(p,i,cant)
3: Definir(p.GastoPorPersona, per, iObtenerGasto(p,per) + gastoTotal)
4: cantConDescuento ← iCantidadConDescuento(p,i,cant)
5: Definir(p.Ventas, per, AgregarAtrás(Significado(p.Ventas, per), <i, cantConDescuento>))
6: si cant - cantConDescuento ≠ 0 entonces
7:     si Definido?(per, p.VentasSinDescuentos)  Definido?(item, Significado(p.VentasSinDescuentos, per) entonces
8:         (Significado((Significado(p.VentasSinDescuentos, per), i).Siguente)).cantidad ← (Significado((Significado(p.VentasSinDescuentos, per), i).Siguiente).cantidad + (cant - cantConDescuento)
9:     else
10:        Definir(p.Ventas, per, AgregarAdelante(Significado(p.Ventas, per), <i, cant - cantConDescuento>))
11:        itVentasDescuentoDeItem ← itVentasDescuentoDeItem.Crear(Significado(p.Ventas, per))
12:        si Definido?(per, p.VentasSinDescuentos entonces
13:            Definir(Significado(p.VentasSinDescuentos, per), i, itVentasDescuentoDeItem.Siguiente)
14: else
15:     Definir(p.VentasSinDescuentos, per, Vacio())
16:     Definir(Significado(p.VentasSinDescuentos, per), i, itVentasDescuentoDeItem.Siguiente)
```

Esto esta mal identado?
Cualquiera sea el caso, esta mal.
Faltan casos de los if-else

Complejidad: $O(log(I) + log(cant) + log(A))$

Justificación: La complejidad es O(log(I) + log(cant) + log(A)). Esta va a estar determinada por la actualización del stock, la actualización del diccionario GastoPorPersona, la actualización de Ventas y la actualización de VentasSinDescuentos. En este algoritmo el peor caso se da una vez que entramos a la rama positiva del primer if. Aquí dentro, cualquiera de las ramas que elijamos da lo mismo porque tienen todas la misma complejidad. Por lo tanto da igual que consideremos como peor caso ya que la complejidad será la misma. Por ello vamos a considerar aquello que pasa cuando cant - cantConDescuento != 0 y Definido?(per, p.VentasSinDescuentos) Definido?(item, Significado(p.VentasSinDescuentos, per). En este caso, define un nuevo stock, calcula cuánto va a ser el gasto de la persona, redefine el diccionario GastoPorPersona, guarda la cantidad para la cual hay algún descuento, redefine ventas agregando el ítem con su cantidad consumida, luego entra a la rama positiva del primer if. Aquí entra a la rama positiva del primer if anidado y redefine la última cantidad sin descuento con la nueva. Expresado en término de complejidades tenemos: O(log(I)) + O(log (I) + O(log (Cant)) + O (log(A)) + O(log(I)) + O(log(cant)) + O(log (A)) + O(log (A)) + O(1) +O(1) + O(log(A)) + O(log(A)) + O(log (A)) + O(log(I)) + O(1) = O(4 * log(I) + 2 log(cant) + 6 * log (A) + 1) = O (log(I)+log(A)+log(cant))

Cumple con el contrato ya que para un ítem del cual hay suficiente cantidad solo se modifican el stock asociado a ese ítem, las ventas asociadas a esa persona dado ese ítem y lo que lleva gastado esa persona en el puesto. El resto de las partes de la estructura se mantienen igual.

---

iObtenerPrecio(**in** p: `puesto` **in** i: `item`) → res : dinero

```
1: res ← Significado(p.Precios, i)
```

Complejidad: $O(log(I))$

Justificación: La complejidad es O(Log(I)) ya que usamos la operación Significado del módulo Diccionario Logarítmico que tiene complejidad O(log(#claves)). En el peor caso posible un puesto puede tener en el menú todos los ítems del lollapatuza, lo que se representa con I.

El algoritmo se comporta como la especificación de precio(p,i). Siempre que entre un ítem que esté en menú (cosa de poder usar Significado sin que se indefina) devuelve correctamente el precio de ese ítem. =0

iTransformarACantidadesConDescuento(**in** *descuentos*: `diccLog(item, diccLog(cantidad, descuento))`) → *res* : diccLog(item, vector(cantidad)

```
1:  res ← Vacio()
2:  itItems.Crear(descuentos)
3:  mientras itItems.HaysSiguiente hacer
4:      vector = Vacía()
5:      itCantidades.Crear(itItems.SiguienteSignificado)
6:      mientras itCantidades.HaySiguiente hacer
7:          agregarAtras(vector, itCantidades.SiguienteClave)
8:      itCantidades.Avanzar
9:      Definir(res, itItems.SiguienteClave, vector)
10:     itItems.Avanzar
```

Complejidad: $O(I * log(I) + I * cant)$

Justificación: La complejidad es O(I * log(I) + I * cant). Este Algoritmo primero usa Vacío que cuesta O(1), luego se utilizan varias operaciones de iteradores que cuestan todas O(1). Luego se itera sobre las claves de descuentos, que en el peor caso son todos los items. El ciclo interior itera a su vez sobre las claves del diccionario (de cantidades) asociado a cada ítem que cuesta O(cant) donde cant es la cantidad máxima para la cual hay descuento. La cantidad de claves en el peor caso corresponde a que esten definidos descuentos para todas las cantidades desde 1 hasta cant. También se llama a Definir que cuesta log(I) pero como la complejidad. Por lo tanto ambas complejidades se multiplican dando una complejidad de O(I) * (O(log(I)) + O(cant). La complejidad no se ve aumentada por cómo se devuelve la estructura porque la devolvemos por referencia y no por copia.

Como el ciclo externo itera sobre todos los ítems de descuentos definiendo el ítem en res nos aseguramos de que las claves son iguales en ambos diccionario. Como el ciclo interno agrega al vector que después se define como significado de res para un ítem, nos aseguramos de que todas las cantidades que son claves del significado de descuentos para un ítem a su vez son elementos del vector agregado. Además como el iterador que recorre el diccionario itera en orden de menor a mayor y cuando agregamos una cantidad a un vector la agregamos atrás se cumple que los vectores definidos estarán ordenados.

---

iGastoDeVenta(**in** *p*: `puesto` **in** *i*: `item` **in** *cant*: `cantidad`) → *res* : puesto

```
1:  descuentoPosible ← iObtenerDescuento(p,i,cant)
2:  si descuentoPosible == 0 entonces
3:      res ← cant * iObtenerPrecio(p,i)
4:  else
5:      cantidadConDescuento ← iCantidadConDescuento(p,i,cant)
6:      res ← (cant-cantidadConDescuento) * iObtenerPrecio(p,i) + cantidadConDescuento * iObtenerPrecio(p,i) *
(100 - iObtenerDescuento(p, i, cantidadConDescuento) * 100)
```

Complejidad: $O(log(I) + log(cant))$

Justificación: La complejidad es O(log(I) + log(cant)). En este algoritmo el peor caso se da cuando entramos a la rama negativa del if ya que aquí es dónde se dan la mayor cantidad de operaciones. Así, calcula si ese ítem tiene descuento, compara para ver si es 0, obtiene la cantidad para la cual hay descuento y calcula el gasto total obteniendo los precios y los descuentos. Expresado en términos de complejidades: O(log(I) + log(cant)) + O(log(I)) + O(log(I) + O(log(I)) + O(log(I) + log(cant)) = O(5log(I) + 2 log(cant)) = O (log(I) + log(cant)) La complejidad no se ve afectada por como se devuelve el resultado ya que al devolver un nat es lo mismo devolverlo por referencia o copia porque el costo es O(1).

Cumple con el contrato ya que para un ítem del cual hay suficiente cantidad nos calcula el costo de esa venta aplicando el descuento según corresponda o no, comportándose como la función gastosDe1Venta(p,<i,c>) =0

iHackearPuesto(**in/out** p : `puesto` **in** per : `persona` **in** i : `item`)

1: it.VentasSinDescuento ← Significado(Significado(p.VentasSinDescuentos, per), i)
2: **si** 1 <(it.VentasSinDescuento.siguiente).cantidad **entonces**
3: (it.VentasSinDescuento.siguiente).cantidad ← (it.VentasSinDescuento.siguiente).cantidad - 1
4: **else**
5: it.VentasSinDescuento.EliminarSiguiente
6: borrar(significado(VentasSinDescuentos, per), i)
7: Definir(p.GastoPorPersona, per, iObtenerGasto(p, per) - ObtenerPrecio(p,i))
8: Definir(p.StockPorItem, i, iObtenerElStock(p, i) + 1)

Falta actualizar p.Ventas

Complejidad: $O(log(A) + log(I))$

Justificación: Este algoritmo primero accede al iterador que está en el diccionario VentaSinDescuento para eso se usa la operacion Significado. Primero se usa sobre el primer diccionario, que tiene como claves a las personas y en el peor caso estan todas las personas por lo que cuesta O(Log(A)), y luego en el otro, donde las claves son los items; siguiendo la misma logica cuesta O(Log(I)). Luego hace operaciones con el iterador que todas cuestan O(1), después en la rama negativa del if se borra una clave, como las claves son personas eso cuesta en el peor caso O(log(A)). Después se define una clave en el diccionario gastoPorPersona y eso cuesta O(log(A)), se llama a iObtenerGasto que cuesta O(log(A)), a su vez se llama a iObtenerPrecio que cuesta O(Log(I)). Por ultimo se define una clave (item) en el diccionario StockPorItem lo cual cuesta O(log(I)). Luego los ordenes de complejidad se suman y como son todo O(log(A)) y O(log(i)) el orden final de HackearPuesto es O(log(A)) + O(log(I)).

Cumple la postcondición siempre que esa persona haya consumido en ese puesto sin descuento, debido a que en ese caso en nuestra estructura de puestos en el diccionario de ventas va a haber una tupla con ese ítem y la cantidad comprada sin descuento y vamos a tener un iterador que apunte a esa tupla que está en el diccionario VentasSinDescuento. Luego con ese iterador acomodamos las ventas cumpliendo una parte de la post. A su vez se resta el gasto de la persona, se suma uno al stock de puesto y como no modificamos las otras estructuras se cumple que todo queda igual, esto significa que el menú, precio, descuento quedan como estaban antes de hackear.

Ojo con los parametros

iCantidadConDescuento(in p: `puesto` **in** i: `item`) → res : dinero

```
1:  si Definido?(p.DescuentoPorCantidadPorItem, i) entonces
2:      vectorCantidades ← Significado(p.CantidadesConDescuento, i)
3:      si cant <vectorCantidades[0] entonces
4:          res ← 0
5:      else
6:          si vectorCantidades[longitud(vectorCantidades) - 1 <= cant] entonces
7:              res ← vectorCantidades[longitud(vectorCantidades) - 1]
8:          else
9:              ultimaPos ← longitud(vectorCantidades) - 1
10:             primeraPos ← 0
11:             mientras primeraPos + 1 <ultimaPos hacer
12:                 medio ← (ultimaPos + primeraPos) / 2
13:                 si cant <vectorCantidades[medio] entonces
14:                     ultimaPos ← medio
15:                 else
16:                     primeraPos ← medio
17:             res ← vectorCantidades[primeraPos]
18: else
19:     res ← 0
```

Complejidad: $O(log(I) + log(cant))$

Justificación: La complejidad es O(log(I) + log(cant)). En este algoritmo el peor caso se da cuando Definido?(p.DescuentoPorCantidadPorItem, i) da true y luego entra al último else de todos esos if anidados. Así, si el ítem está definido en DescuentoPorCantidadPorItem, primero guarda Significado(p.CantidadesConDescuento, i) y luego hace una búsqueda binaria para hallar esa cantidad. Expresado en términos de complejidades: O(log(I)) + O(log(I)) + O(1) + O(1) + O(1) + O(1) + O(log(cant)) + O(1) + O(1) = O(2log(I)+log(cant)+1) O (log(I) + log(cant)). La complejidad no se ve afectada por como se devuelve el resultado ya que al devolver un nat es lo mismo devolverlo por referencia o copia porque el costo es O(1).

Cumple el contrato ya que el algoritmo busca en el vector que es el significado de un item de cantidades con descuento la cantidad máxima para la cual tiene descuento ese ítem. Por lo tanto, se cumple que el descuento de un ítem en el puesto para la cantidad pasada es lo mismo que el descuento de un ítem en el puesto para una cantidad igual a res.