Algoritmos y Estructuras de Datos 2

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

Trabajo práctico 2: Diseño

Lollapatuza

Chip

Integrante	LU	Correo electrónico
Beren, Manuel Andrés	320/22	m.beren83@gmail.com
Carrillo, Mariano	358/22	carr.mariano@gmail.com
Roitman, Sofía	563/21	sofiroit@gmail.com
Valentini, Justo Agustin	566/22	justovalentini@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega	*Maximiliano Martino*	Aprobado
Segunda entrega		

1 Módulos de referencia

✓TAD PERSONA es nat
✓TAD ITEM es nat
✓TAD DESCUENTO es nat
✓TAD CANT es nat
✓TAD GASTO es nat

✓ Dado que los iteradores de diccLog recorren la estructura según un orden determinado, decidimos que las claves se recorran de mayor a menor.
Para el análisis de complejidad, P se refiere a la cantidad de puestos, A se refiere a la cantidad de personas, I se refiere a la cantidad de ítems y cantMax es la cantidad máxima para el cual un producto tiene un descuento definido.

Puesto

Interfaz

se explica con: PUESTO

géneros: `puesto`

Operaciones básicas de puesto

CREARPUESTO(**in** *precios*: `diccLog(item, nat)`, **in** *stock*: `diccLog(item, nat)`, **in** *descuentos*: `diccLog(item, diccLog(nat,nat))`) → *res* : `puesto`
✓**Pre** ≡ {$claves(precios) = claves(stock) \wedge claves(descuentos) \subset claves(precios)$}
✓**Post** ≡ {res =$_{obs}$ crearPuesto(precios, stock, descuentos)}
Complejidad: $\mathcal{O}(copy(diccLog) + I * cantMax)$
Descripción: Genera un nuevo puesto.
Aliasing: Devuelve el puesto por copia.

STOCK(**in** *p*: `puesto`, **in** *i*: `item`) → *res* : `nat`
✓**Pre** ≡ {def?(p.stock, i)}
✓**Post** ≡ {res =$_{obs}$ stock(p, i)}
Complejidad: $\mathcal{O}(log(I))$
Descripción: Devuelve el stock de un producto.
Aliasing: El stock del producto es modificable.

OBTENERDESCUENTO(**in** *p*: `puesto`, **in** *i*: `item` **in** *c*: `nat`) → *res* : `nat`
✓**Pre** ≡ {def?(p.precios, i)}
✓**Post** ≡ {res =$_{obs}$ descuento(p, i, c)}
Complejidad: $\mathcal{O}(log(I) + log(cantMax))$
Descripción: Devuelve el descuento al comprar una cantidad determinada de un producto.
Aliasing: El descuento no es modificable.

OBTENERGASTO(**in** *p*: `puesto`, **in** *a*: `persona`) → *res* : `nat`
✓**Pre** ≡ {def?(p.ventas, a)}
✓**Post** ≡ {res =$_{obs}$ gastosDeVentas(p, ventas(p,a))}
Complejidad: $\mathcal{O}(log(A))$
Descripción: Devuelve el gasto realizado por una persona en el puesto.
Aliasing: El gasto es modificable.

Representación

Representación de puesto

`puesto` **se representa con** `ePuesto`

donde `ePuesto` es `tupla`(*id*: `nat` , *stock*: `diccLog(item, stock)` , *descuento*: `diccLog(item,diccLog(cant, descuento))` , *ventas*: `diccLog(persona, gasto)` , *precios*: `diccLog(item, nat)` , *histCompras*: `tupla`(*sinDesc* : *diccLog*(*persona*, *diccLog*(*item*, *nat*)), *conDesc*: `diccLog(persona, diccLog(item, nat))`)

Rep : ePuesto ⟶ bool

✓ Rep(e) $\equiv$ true $\iff$ $claves(e.precios) = claves(e.stock) \wedge claves(e.descuento) \subset claves(e.precios)$
$\wedge\, claves(e.ventas) = claves(e.histCompras.sinDesc) = claves(e.histCompras.conDesc)$
$\wedge\, (\forall i : item)(def?(i, e.descuento) \rightarrow_L (\forall n : nat)(min(claves(obtener(i, e.descuento))) \leq n < min(claves(obtener(i, e.descuento))) + \#claves(obtener(i, e.descuento)) \rightarrow_L$
$0 < obtener(n, obtener(i, e.descuento)) \leq obtener(n+1, obtener(i, e.descuento)) < 100))$
$\wedge\, (\forall p : persona)(def?(p, e.ventas) \rightarrow_L obtener(p, e.ventas) = sumaCompras(p, e.histCompras, e.precios, e.descuento))$
$\wedge\, (\forall p : persona)(def?(p, e.histCompras.sinDesc) \rightarrow_L \emptyset?(claves(obtener(p, e.histCompras.sinDesc)) \vee (\forall i : item)(def?(i, obtener(p, e.histCompras.sinDesc))$
$\rightarrow_L obtener(i, obtener(p, e.histCompras.sinDesc)) < min(claves(obtener(i, e.descuento)))$
$\wedge\, (\forall p : persona)(def?(p, e.histCompras.conDesc) \rightarrow_L \emptyset?(claves(obtener(p, e.histCompras.conDesc)) \vee (\forall i : item)(def?(i, obtener(p, e.histCompras.conDesc))$
$\rightarrow_L min(claves(obtener(i, e.descuento))) \leq obtener(i, obtener(p, e.histCompras.conDesc))))$

Abs : ePuesto e $\longrightarrow$ puesto {Rep(e)}

✓ Abs(e) $=_{\text{obs}}$ p: puesto | $menu(p) = claves(e.precios) \wedge (\forall i : item)(def?(i, e.precios) \rightarrow_L precio(p, i) =_{\text{obs}} obtener(i, e.precio)$
$\wedge stock(p, i) =_{\text{obs}} obtener(i, e.stock)) \wedge descuento(p, i, c) =_{\text{obs}} (\forall i : item)(def?(i, e.descuento) \rightarrow_L$
$(\forall c : cant)(def?(c, obtener(i, e.descuento)) \rightarrow_L$
$descuento(p, i, c) =_{\text{obs}} obtener(c, obtener(c.descuento))))$
$\wedge\, (\forall a : persona)(def?(a, e.ventas) \rightarrow_L separarSinYConDesc(ventas(p, a)) =_{\text{obs}}$
$< obtener(a, e.histCompras.sinDesc), obtener(a, e.histCompras.conDesc) >)$

Operaciones auxiliares para la representación de puesto

✓ min : conj(nat)c $\longrightarrow$ nat {$\neg\emptyset?$(c)}

✓ min(c) $\equiv$ minimo(dameUno(c),sinUno(c))

✓ minimo : nat $\times$ conj(nat) $\longrightarrow$ nat

✓ minimo(n,c) $\equiv$ **if** $\emptyset?$(c) **then**
n
else
if n < dameUno(c) **then** minimo(n,sinUno(c)) **else** minimo(dameUno,sinUno(c)) **fi**
fi

✓ sumaCompras : persona $\times$ <diccLog(persona, diccLog(item, nat)), diccLog(persona, diccLog(item, nat))>
$\times$ diccLog(item, nat) $\times$ diccLog(item, diccLog(cant, descuento)) >) $\rightarrow$ nat

✓ sumaCompras(p, <sinDesc, conDesc >, precios, desc) $\equiv$ Calcula el gasto total de una persona a partir de su historial de compras.

✓ separarSinYConDesc : multiconj(<item $\times$ cant>) $\longrightarrow$ <diccLog(item, cant $\times$ diccLog(item, cant)>

✓ separarSinYConDesc(m) $\equiv$ Dado un multiconjunto del historial de compras de una persona, convierte este en una tupla de dos diccLogs separando según si cada compra tuvo descuento o no. En caso de que la persona no haya hecho ninguna compra con o sin descuento, el diccionario va a ser vacio.

✓ El rep verifica que los ítems definidos en precios y stock sean los mismos, y que cada ítem que tenga descuento sea un ítem válido. Chequea que los diccionarios con las claves persona sean iguales. Revisa que toda las cantidades entre el min y el max esten definidas en descuento y que el porcentaje de descuento sea valido (se asume que el descuento aumenta proporcionalmente a la cantidad). Por ultimo, se fija que el gasto total coincida con calcular el gasto a partir del historial de compra y que las compras en sinDesc y conDesc hayan sido efectivamente realizadas sin y con

descuento respectivamente.

Algoritmos

---

icrearPuesto(**in** *precios*: `diccLog(item, nat)`, **in** *stock*: `diccLog(item, nat)`, **in** *descuentos*: `diccLog(item, diccLog(nat, nat))` $\rightarrow$ *res*: puesto

1: $res.id \leftarrow 0$
2: $res.stock \leftarrow stock$ ▷ $\mathcal{O}(copy(diccLog))$
3: $res.ventas \leftarrow vacio()$ ▷ $\mathcal{O}(1)$
4: $res.precios \leftarrow precios$ ▷ $\mathcal{O}(copy(diccLog))$
5: $res.histCompras.sinDesc \leftarrow vacio()$ ▷ $\mathcal{O}(1)$
6: $res.histCompras.conDesc \leftarrow vacio()$ ▷ $\mathcal{O}(1)$
7: itDiccLog $it \leftarrow crearIt(descuentos)$
8: diccLog(item,diccLog(nat,nat)) $descuentoExtenso \leftarrow vacio()$
9: **while** haySiguiente(it) **do** ▷ $\mathcal{O}(I)$
10: nat $item \leftarrow siguienteClave(it)$
11: $definir(descuentoExtenso, item, vacio())$ ▷ $\mathcal{O}(log(I))$
12: itDiccLog $itC \leftarrow crearIt(siguienteSignificado(it))$
13: **while** haySiguiente(itC) **do** ▷ $\mathcal{O}(cantMax)$
14: nat $desc \leftarrow siguienteSignificado(itC)$
15: nat $cant \leftarrow siguienteClave(itC)$
16: avanzar(itC)
17: **if** (haySiguiente(itC)) **then**
18: nat $cantSiguiente \leftarrow siguienteClave(itC)$
19: **while** cant $\neq$ cantSiguiente **do**
20: $definir(significado(descuentoExtenso, item), cant, desc)$ ▷ $\mathcal{O}(log(I))$
21: $cant \leftarrow cant + 1$
22: **end while**
23: **else**
24: $definir(descuentoExtenso, item, definir(vacio(), cant, desc))$
25: **end if**
26: avanzar(it)
27: **end while**
28: **end while**
29: $res.descuento \leftarrow descuentoExtenso$ ▷ $\mathcal{O}(copy(copy(diccLog)))$
30: **return** res

Complejidad: $\mathcal{O}(copy(diccLog) + I * cantMax)$ ✗ Revisar esta complejidad

---

✓ El codigo hasta la linea 6 es simplemente asignar las variables de entrada a sus respectivos en el res. De la linea 7 en adelante, se construye un diccionario que contiene hasta la cantMaxima para cada cantidad de productos ingresada con su descuento. Es decir si yo tenia descuento para 3 empandas(10 desc) en adelante y 6(30 des) en adelante(y ninguna mas) en el diccionarioExtenso tengo: para 3 empanadas 10, para 4 10, para 5 10, para 6 30.

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

1: **return** $significado(p.stock, i)$ ▷ $\mathcal{O}(log(I))$

Complejidad: $\mathcal{O}(log(I))$

---

iobtenerDescuento(**in** p: `puesto`, **in** i: `item`, **in** c: `nat`) → res: nat

1: $res \leftarrow 0$
2: diccLog $descItem \leftarrow significado(p.descuento, i)$ ▷ $\mathcal{O}(log(I))$
3: nat $max \leftarrow siguienteClave(crearIt(descItem))$ ▷ $\mathcal{O}(1)$
4: **if** max ≤ c **then**
5: $res \leftarrow max$
6: **else**
7: **if** definido?(descItem,c) **then**
8: $res \leftarrow significado(descItem, c)$ ▷ $\mathcal{O}(log(cantMax))$
9: **end if**
10: **end if**
11: **return** res

Complejidad: $\mathcal{O}(log(I) + log(cantMax))$

---

iobtenerGasto(**in** p: `puesto`, **in** a: `persona`) → res: nat

1: **return** res $res \leftarrow significado(p.ventas, a)$ ▷ $\mathcal{O}(log(A))$

Complejidad: $\mathcal{O}(log(A))$

Lollapatuza

Interfaz

se explica con: LOLLAPATUZA

géneros: `lolla`

Operaciones básicas de lolla

CREARLOLLA(**in** *puestos*: `diccLog(id, puesto)`, **in** *personas*: `conj(persona)`) → *res* : `lolla`
Pre ≡ {vendenAlMismoPrecio(puestos) ∧ noVendieronAun(significados(puestos)) ∧ ¬∅?(claves(puestos))}
Post ≡ {res =$_{obs}$ crearLolla(puestos, personas)}
Complejidad: $\mathcal{O}(P + A * log(A))$
Descripción: Crea un Lollapatuza.
Aliasing: Devuelve el lolla por copia.

COMPRAR(**in/out** l: `lolla`, **in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** *cant*: `nat`)
Pre ≡ $\{p = p_0 \wedge l = l_0 \wedge a \in personas \wedge (\exists id : nat)(significado(l.idPuestos, id) =_{obs} p) \wedge i \in menu(p) \wedge_L haySuficiente(p, i, c)\}$
Post ≡ $\{l =_{obs} vender(l, p.id, a, i, c)\}$
Complejidad: $\mathcal{O}$(log(A) + log(I) + log(P) + log(cantMax))
Descripción: Registra una compra de un ítem hecha por una persona en un puesto.
Aliasing: Modifica el lolla y el puesto dados.

HACKEAR(**in/out** l: `lolla`, **in** a: `persona`, **in** i: `item`)
Pre ≡ $\{l = l_0 \wedge consumioSinPromoPuestos(a, i, puestos(l))\}$
Post ≡ {l =$_{obs}$ hackear(l_0, a , i)}
Complejidad: $\mathcal{O}(log(I) + log(A) + log(P))$ si el puesto deje de ser hackeable;
$\mathcal{O}(log(I) + log(A))$ si el puesto sigue siendo hackeable
Descripción: Hackea un puesto dado un ítem y una persona.
Aliasing: Modifica el lolla dado y el puesto de menor ID en el que la persona consumió el ítem sin descuento.

GASTOTOTAL(**in** l: `lolla`, **in** a: `persona`) → *res* : `nat`
Pre ≡ {a ∈ personas(l)}
Post ≡ {res =$_{obs}$ gastoTotal(l, a)}
Complejidad: $\mathcal{O}(log(A))$
Descripción: Devuelve el gasto total de una persona.
Aliasing: El gasto es modificable.

MAXCONSUMIDOR(**in** l: `lolla`) → *res* : `persona`

Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ masGasto(l)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la persona que más gastó.
Aliasing: No es modificable.

PUESTOIDMENORSTOCK(**in** l: `lolla`, **in** i: `item`) $\rightarrow$ res : `puesto`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ menorStock(l,i)}
Complejidad: $\mathcal{O}(P * log(I))$
Descripción: Devuelve el puesto con menor stock para un ítem dado, en caso de haber más de un puesto con esa cantidad, devuelve el de menor ID.
Aliasing: El puesto no es modificable.

INFOSISTEMAPERSONAS(**in** l: `lolla`) $\rightarrow$ res : `conj(persona)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ personas(l)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve las personas presentes en el lolla.
Aliasing: Devuelve una referencia no modificable de las personas.

INFOSISTEMAPUESTOS(**in** l: `lolla`) $\rightarrow$ res : `diccLog(id, puesto)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ puestos(l)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los puestos de comida con sus ID.
Aliasing: Devuelve una referencia no modificable de los puestos.

Representación

Representación de lollapatuza

`lolla` **se representa con** `eLolla`

donde `eLolla` es `tupla`(*maxConsumidor*: `diccLog(gasto, diccLog(persona,nat))` , *personas*: `conj(persona)` , *idPuestos*: `diccLog(id, puesto)` , *gastoTotal*: `diccLog(persona, gasto)` , *comprasHack*: `diccLog(persona, diccLog(item, minHeap(puesto)))`)

Rep : eLolla $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ $e.personas = claves(e.gastoTotal) = unionClaves(significados(e.maxConsumidor))$
$\wedge\ (\forall c, c' : conj(personas))(c, c' \in significados(e.maxConsumidor) \wedge c \neq c' \rightarrow_l c \cap c' = \emptyset)$
$\wedge\ (\forall i : id)(def?(i, e.idPuestos) \rightarrow_l obtener(i, e.idPuestos).id = i)$
$\wedge\ claves(e.maxConsumidor) = multiConjAConj(significados(gastoTotal))$
$\wedge\ (\forall a : persona)(def?(a, e.comprasHack) \rightarrow_l (\forall i : item)(def?(i, obtener(a, e.comprasHack)) \rightarrow_l$
$(\forall p : puesto)(esta?(p, obtener(i, obtener(a, e.comprasHack))) \rightarrow p \in significados(e.idPuestos))))$
Abs : eLolla e $\longrightarrow$ lolla {Rep(e)}
Abs(e) $=_{\text{obs}}$ l: lolla | $puestos(l) =_{\text{obs}} e.idPuestos \wedge personas(l) =_{\text{obs}} e.personas$

Operaciones auxiliares para la representación de puesto

unionClaves : multiConj(diccLog(persona $\times$ nat)) $\longrightarrow$ conj(persona)
unionClaves(mc) $\equiv$ extrae las claves de los dicclog y luego une los conjuntos
multiConjAConj : multiConj(gasto) $\longrightarrow$ conj(gasto)

✓multiConjAConj(mc) ≡ transforma el mc en un conjunto

Algoritmos

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

1: $res.personas \leftarrow personas$
2: $res.idPuestos \leftarrow vacio()$
3: itPuestos $itP \leftarrow crearIt(puestos)$
4: **while** haySiguiente?(itP) **do** ▷ $\mathcal{O}(P)$
5: siguienteSignificado(it).id ← siguienteClave(it)
6: avanzar(it)
7: **end while**
8: $res.maxConsumidor \leftarrow definir(vacio(), 0, vacio())$
9: $res.gastoTotal \leftarrow vacio()$
10: $res.comprasHack \leftarrow vacio()$
11: itConj $itC \leftarrow crearIt(personas)$
12: **while** haySiguiente(itC) **do** ▷ $\mathcal{O}(A * log(A))$
13: $definir(significado(res.maximoConsumidor, 0), siguiente(itC), 1)$
14: $definir(res.gastoTotal, siguiente(itC), 0)$
15: $definir(res.comprasHack, siguiente(itC), vacio())$
16: avanzar(itC)
17: **end while**
18: **return** res

✓Complejidad: $\mathcal{O}(P + A * log(A))$

Para maxConsumidor el enunciado no especifica si hay empate en el consumo si se devuelve el que tenga mayor o menor ID, por lo tanto asumimos que devuelve el de mayor ID. El significado de cada persona dentro de cada gasto es 1, dado que usamos esta estructura únicamente para que se mantenga en orden por ID.

Si no requieren el significado, usen un conjunto que tambien les mantiene el orden.

✓**icomprar**(**in/out** l: `lolla`, **in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** $cant$: `nat`)

1: $significado(p.stock, i) \leftarrow significado(p.stock, i) - cant$ ▷ $\mathcal{O}(log(I))$
2: nat $descuento \leftarrow obtenerDescuento(p, i, cant)$ ▷ $\mathcal{O}(log(I) + log(cantMax))$
3: nat $gastoActualizado \leftarrow significado(l.gastoTotal, a) + (significado(p.precios, i) * cant) * (1 - descuento/100)$
4: $borrar(significado(l.maxConsumidor, significado(l.gastoTotal, a)), a)$ ▷ $\mathcal{O}(log(A))$
5: **if** $\#claves(significado(l.maxConsumidor, significado(l.gastoTotal, a))) == 0$ **then**
6: $borrar(l.maxConsumidor, significado(l.gastoTotal, a))$ ▷ $\mathcal{O}(log(A))$
7: **end if**
8: **if** definido?(l.maxConsumidor,gastoActualizado) **then** ▷ $\mathcal{O}(log(A))$
9: definir(significado(l.maxConsumidor,gastoActualizado),a,1) ▷ $\mathcal{O}(log(A))$
10: **else**
11: definir(l.maxConsumidor,gastoActualizado,definir(vacio(),a,1)) ▷ $\mathcal{O}(log(A))$
12: **end if**
13: **if** descuento == 0 **then**
14: **if** ¬ definido?(significado(l.comprasHack,a),i) **then** ▷ $\mathcal{O}(log(A))$
15: $definir(significado(l.comprasHack, a), i, vacia())$
16: **end if**
17: $encolar(significado(significado(l.comprasHack, a), i), p)$ ▷ $\mathcal{O}(log(A) + log(I) + log(P))$
18: **if** definido?(p.histCompras.sinDesc, a) **then** ▷ $\mathcal{O}(log(A)$
19: **if** definido?(significado(p.histCompras.sinDesc, a), i) **then** ▷ $\mathcal{O}(log(A) + log(I))$
20: $significado(significado(p.histCompras.sinDesc, a), i) \leftarrow$
$significado(significado(p.histCompras.sinDesc, a), i) + cant$ ▷ $\mathcal{O}(log(A) + log(I))$
21: **else**
22: $definir(significado(p.histCompras.sinDesc, a), i, cant)$ ▷ $\mathcal{O}(log(A))$
23: **end if**
24: **else**
25: diccLog(item, nat) $nuevo \leftarrow definir(vacio(), item, cant)$ ▷ $\mathcal{O}(1)$
26: $definir(p.histCompras.sinDesc, a, nuevo)$ ▷ $\mathcal{O}(log(A))$
27: **end if**
28: **else**
29: **if** $\neg definido?(p.histCompras.conDesc, a)$ **then** ▷ $\mathcal{O}(log(A))$
30: diccLog(item, nat) $nuevo \leftarrow definir(vacio, i, cant)$
31: definir(p.histCompras.conDesc, a, nuevo)
32: **else**
33: **if** $\neg definido?(significado(p.histCompras.conDesc, a), i)$ **then** ▷ $\mathcal{O}(log(A) + log(I))$
34: diccLog(item, nat) $nuevo \leftarrow definir(vacio, i, cant)$
35: definir(significado(p.histCompras.conDesc), a, nuevo)
36: **else**
37: $significado(significado(p.histCompras.conDesc, a), i) \leftarrow$
$significado(significado(p.histCompras.conDesc, a), i) + cant$ ▷ $\mathcal{O}(log(A) + log(I))$
38: **end if**
39: **end if**
40: **end if**
41: $significado(p.ventas, a) \leftarrow significado(p.ventas, a) + (significado(p.precios, i) * cant) * (1 - descuento/100)$ ▷ $\mathcal{O}(log(A) + log(I))$
42: $significado(l.gastoTotal, a) \leftarrow gastoActualizado$ ▷ $\mathcal{O}(log(A))$
Complejidad: $\mathcal{O}(log(A) + log(I) + log(P) + log(cantMax))$

✓El código de comprar primero actualiza el gasto y borra a la persona del diccionario de todas las personas que habían gastado lo mismo antes de la compra, en caso de que sea la única persona que tenía ese gasto, borra la clave. Luego, verifica si hay alguien que haya gastado la misma cantidad que la persona que compra (con el gasto de esta actualizado), en caso de haberlo, agrega la persona al diccionario, caso contrario, define una nueva clave para el gasto. Si la compra fue sin descuento, si es el primer ítem que compra la persona, lo define dentro de sus compras hackeables y sus compras sin descuento, si no, simplemente suma la cantidad comprada a sus compras hackeables y a sus compras sin descuento, además encola el puesto en los posibles puestos hackeables. Si la compra fue hecha sin descuento, hace lo mismo que en el caso anterior, omitiendo la parte de las compras hackeables. Finalmente, actualiza el gasto total de la persona en el lolla y en el puesto.

✓ **ihackear**(**in/out** l : `lolla`, **in** a : `persona`, **in** i : `item`) → res: bool

1: puesto $puestoHack \leftarrow proximo(significado(sgnificado(l.comprasHack, a), i))$ ▷ $\mathcal{O}(log(A) + log(I))$
2: nat $precioItem \leftarrow significado(puestoHack.percios, i)$ ▷ $\mathcal{O}(log(I))$
3: $significado(puestoHack.ventas, a) \leftarrow significado(puestoHack.ventas, a) - precioItem$ ▷ $\mathcal{O}(log(A))$
4: $significado(puestoHack.stock, i) \leftarrow significado(puestoHack.stock, i) + 1$ ▷ $\mathcal{O}(log(I))$
5: nat $comprasItemPersonaSinDesc \leftarrow significado(significado(puestoHack.histCompras.sinDesc, a), i)$ ▷ $\mathcal{O}(log(A) + log(I))$
6: $comprasItemPersonaSinDesc \leftarrow comprasItemPersonaSinDesc - 1$
✗ 7: **if** comprasItemPersonaSinDesc = 0 **then** Que pasa si la persona ya no tiene compras para un item? Nunca actualizan l.comprasHack() ▷ $\mathcal{O}(log(A) + log(I))$
✗ 8: $desencolar(significado(significado(l.comprasHack, a), i))$ ▷ $\mathcal{O}(log(A) + log(I) + log(P))$
9: **end if**
10: $borrar(significado(l.maxConsumidor, significado(l.gastoTotal, a)), a)$ ▷ $\mathcal{O}(log(A))$
11: **if** #claves(significado(l.maxConsumidor,significado(l.gastoTotal,a))) = 0 **then** ▷ $\mathcal{O}(log(A))$
12: $borrar(l.maxConsumidor, significado(l.gastoTotal, a))$ ▷ $\mathcal{O}(log(A))$
13: **end if**
14: nat $gastoActualizado \leftarrow significado(l.gastoTotal, a) - precioItem$ ▷ $\mathcal{O}(log(A))$
15: $significado(l.gastoTotal, a) \leftarrow gastoActualizado$ ▷ $\mathcal{O}(log(A))$
16: **if** definido?(l.maxConsumidor,gastoActualizado) **then** ▷ $\mathcal{O}(log(A))$
17: definir(significado(l.maxConsumidor,gastoActualizado),a,1) ▷ $\mathcal{O}(log(A))$
18: **else**
19: definir(l.maxConsumidor,gastoActualizado,definir(vacio(),a,1)) ▷ $\mathcal{O}(log(A))$
20: **end if**

✓ Complejidad(el puesto deja de ser hackeable): $\mathcal{O}(log(A) + log(I) + log(P))$
✓ Complejidad(el puesto sigue siendo hackeable): $\mathcal{O}(log(A) + log(I))$

✓ **igastoTotal**(**in** l : `lolla`, **in** a : `persona`) → res: nat

1: $res \leftarrow significado(l.gastoTotal, a)$
2: **return** res
✓ Complejidad: $\mathcal{O}$(log(A))

✓ **imaxConsumidor**(**in** l : `lolla`) → res: persona

1: itDiccLog $it1 \leftarrow crearIt(l.maxConsumidor)$ ▷ $\mathcal{O}(1)$
2: itDiccLog $it2 \leftarrow crearIt(siguienteSignificado(it1))$ ▷ $\mathcal{O}(1)$
3: $res \leftarrow siguienteClave(it2)$
4: **return** res
✓ Complejidad: $\mathcal{O}(1)$

✓**ipuestoIDMenorStock**(**in** l : `lolla`, **in** i : `item`) → res: puesto

```
1: itDiccLog it ← crearIt(l.idpuestos)
2: res ← siguienteSignificado(it)
3: nat min ← 0
4: if definido?(res.stock, i) then
5:     min ← significado(res.stock, i)
6: end if
7: while haySiguiente(it) do                                          ▷ O(P)
8:     if definido?(siguienteSignficado(it).stock, i) then            ▷ O(log(I))
9:         if significado(siguienteSignificado(it).stock, i) ≤ min then   ▷ O(log(I))
10:            res ← siguienteSignificado(it)
11:            min ← significado(siguienteSignificado(it).stock, i)   ▷ O(log(I))
12:        end if
13:    end if
14:    avanzar(it)
15: end while
16: if res.id = siguienteClave(it) ∧ ¬definido?(res.stock,i) ∧ #claves(l.iduestos) ≠ 1  then
17:    res ← anteriorSignificado(it)
18: end if
19: return res
```

Complejidad: $\mathcal{O}(P * log(I))$

✓El último if sirve en caso de que ningún puesto tenga el ítem y haya mas de un puesto, devuelve el ultimo puesto que recorre el iterador, el cual es el puesto de menor id(esto es para que cumpla con la funcion de que devuelve el puesto de menor id)

✓**iinfoPersonas**(**in** l : `lolla`) → res: conj(persona)

```
1: res ← l.personas
2: return res
```

✓Complejidad: $\mathcal{O}(1)$

✓**iinfoPuestos**(**in** l : `lolla`) → res: conj(puesto)

```
1: res ← l.puestos
2: return res
```

✓Complejidad: $\mathcal{O}(1)$

minHeap(puesto)

Interfaz

se explica con: COLA DE PRIORIDAD(α)

géneros: `minHeap(puesto)`

Operaciones básicas de minHeap

VACIA() → res : `minHeap`
✓**Pre** ≡ {true}
✓**Post** ≡ {res =$_{obs}$ vacía()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un heap vacío.
Aliasing: No genera Aliasing.

PROXIMO(**in** h : `minHeap(puesto)`) → res : `puesto`
✓**Pre** ≡ {¬ vacia?(h)}
✓**Post** ≡ {res =$_{obs}$ proximo(h)}
Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve una copia del elemento de mínima prioridad.
Aliasing: No genera Aliasing.

ENCOLAR(**in/out** h: `minHeap(puesto)`, **in** p: `puesto`)
Pre $\equiv$ {h=h_0 $\wedge \neg pertenece(h, p)$}
Post $\equiv$ {h $=_{obs}$ encolar(h_0, p)}
Complejidad: $\mathcal{O}(log(n))$
Descripción: Agrega un elemento al heap.
Aliasing: No genera Aliasing.

DESENCOLAR(**in/out** h: `minHeap(puesto)`)
Pre $\equiv$ {h=h_0 $\wedge$ $\neg$ vacia?(h)}
Post $\equiv$ {h $=_{obs}$ desencolar(h_0)}
Complejidad: $\mathcal{O}(log(n))$
Descripción: Elimina el primer elemento del heap.
Aliasing: No genera Aliasing.
Extensión cola de prioridad
pertenece: minHeap(puesto) $\times$ puesto $\rightarrow$ bool
h, p $\equiv$ **if** vacía(h) **then** false **else** p=proximo(c) $\vee$ pertenece(desencolar(h), p) **fi**

Representación

Representación de minHeap

`MinHeap` **se representa con** `vector(puesto)`

Rep : minHeap $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ ($\forall i$: nat)(0 $\leq$ i < long(e) $\rightarrow_L$ ((2i+1 <long(e) $\rightarrow_L$ e[i].id<e[2i+1].id) $\wedge$ (2i+2 <long(e) $\rightarrow_L$ e[i].id<e[2i+2].id)))
Hay casos en los que se indefine
Abs : minHeap e $\longrightarrow$ minHeap {Rep(e)}
Abs(e) $=_{obs}$ h: minHeap | vacio(h)$=_{obs}$vacio?(e) $\wedge$ $\neg$vacio(h)$\rightarrow_L$ (proximo(h)$=_{obs}$prim(e) $\wedge$ mismosElementos(desencolar(h), fin(e)) $\wedge$ rep(fin(e))
mismosElementos : coladeprioridad(α)$\times$secu(α) $\longrightarrow$ bool
mismosElementos(c, s) $\equiv$ La función mismosElementos compara que los elementos de la cola sean los mismos que los de la secuencia.

El criterio por el cual se mantiene ordenado el heap es que el id del puesto en la raíz sea menor al id de todos sus hijos.

Algoritmos

---

vacia() $\rightarrow$ *res*: minHeap(puesto)
1: *res* $\leftarrow$ *vector.vacia*()
2: **return** res
Complejidad: $\mathcal{O}(1)$

---

proximo(**in** h: `minHeap(puesto)`) $\rightarrow$ *res*: puesto
1: *res* $\leftarrow$ h[0]
2: **return** res
Complejidad: $\mathcal{O}(1)$

---

encolar(**in/out** h : `minHeap(puesto)`, **in** p : `puesto`)

1: $agregarAtras(h, p)$
2: nat $i \leftarrow longitud(h) - 1$
3: **while** h[(i-1)/2].id>h[i].id && i!=0 **do** ▷ $\mathcal{O}(log(longitud(h)))$
4: $h[i] \leftarrow h[(i-1)/2]$
5: $h[(i-1)/2] \leftarrow n$ Que es n?
6: $i \leftarrow (i-1)/2$
7: **end while**

Complejidad: $\mathcal{O}(log(longitud(h))$

---

desencolar(**in/out** h : `minHeap(puesto)`)

1: puesto $var \leftarrow h[longitud(h) - 1]$
2: $h[0] \leftarrow var$ No eliminan correctamente el maximo. Revisar algoritmo
3: $h.eliminar(longitud(h) - 1)$ ▷ $\mathcal{O}(1)$
4: nat $i \leftarrow 0$
5: **while** h[i].id>h[2i+1].id || h[i].id>h[2i+2].id && i<longitud(h) **do** ▷ $\mathcal{O}(log(longitud(h)))$
6: **if** h[2i+1].id<h[2i+2].id **then**
7: $h[i] \leftarrow h[2i+1]$
8: $h[2i+1] \leftarrow var$
9: $i \leftarrow 2i+1$
10: **else**
11: $h[i] \leftarrow h[2i+2]$
12: $h[2i+2] \leftarrow var$
13: $i \leftarrow 2i+2$
14: **end if**
15: **end while**

Complejidad: $\mathcal{O}(log(longitud(h))$