Maximiliano Martino
Aprobado

1. Módulo Lollapatuza

El módulo Lollapatuza provee un sistema de festival en el que se puede crear el festival, realizar transacciones, hackear a una persona en el festival, calcular el gasto de determinada persona, saber quien es la persona que mas gasto, calcular el puesto con menor stock de un item, obtener a las personas del festival y obtener todos los puestos del festival.

Interfaz

se explica con: LOLLAPATUZA.

géneros: `lolla`.

Operaciones básicas

CREARLOLLA(**in** p: `dicc(nat,puesto)`, **in** a: `conj(persona)`) $\rightarrow$ res : `lolla`
Pre $\equiv$ {vendenAlMismoPrecio(significados(p)) $\wedge$ NoVendieronAun(significados(ps)) $\wedge$ $\neg\emptyset$(a) $\wedge$ $\neg\emptyset$(claves(p))}
Post $\equiv$ {$res =_{\text{obs}}$ crearLolla(p,a)}
Complejidad: $\Theta(log(P) \times P + log(A) \times A + MaxId + A)$
Descripción: Crea un nuevo Lollapatuza
Aliasing: No presenta aspectos de aliasing.

TRANSACCIÓN(**in/out** l: `lolla`, **in** id: `nat` , **in** a: `persona` , **in** i: `item` , **in** c: `cant`)
Pre $\equiv$ {$l =_{\text{obs}} l_0 \wedge a \in personas(l) \wedge def?(id, puestos(l)) \wedge_l haySuficiente?(obtener(id, puestos(l)), i, c)$}
Post$\equiv$ {$l =_{\text{obs}}$ vender(l_0, id, a, i, c)}
Complejidad:$\Theta(log(A) + log(I) + log(P))$
Descripción: Registra la compra de una cantidad de un ítem particular, realizada por una persona en un puesto.
Aliasing: No presenta aspectos de aliasing.

HACKEAR(**in/out** l: `lolla`, **in** i: `item` , **in** a: `persona`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$l =_{\text{obs}} lo \wedge$ a $\in$ personas(l)}
Post $\equiv$ {$res =_{\text{obs}}$ ConsumioSinPromoEnAlgunPuestoPeroAEnPersonas(l, a, i) $\wedge_l$ (res $\Rightarrow_l$ l = hackear(lo, a, i)) ($\neg res \Rightarrow$ l = lo)}
Complejidad: $\Theta(log(A) + log(I) + log(P))$ en el peor caso.
Descripción: Hackear un ítem consumido por una persona. Se hackea el puesto de menor ID en el que la persona haya consumido ese ítem sin promoción.
Aliasing: No presenta aspectos de aliasing.

GASTOPERSONA(**in** l: `lolla`, **in** a: `persona`) $\rightarrow$ res : `nat`
Pre $\equiv$ {a $\in$ personas(l)}
Post $\equiv$ {$res =_{\text{obs}}$ gastoTotal(l,a)}
Complejidad: $\Theta(log(A))$
Descripción: Obtiene el gasto total de una persona.
Aliasing: No presenta aspectos de aliasing.

PERSONAQUEMASGASTO(**in** l: `lolla`) $\rightarrow$ res : `persona`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ masGasto(l)}
Complejidad: $\Theta(1)$
Descripción: Obtiene la persona que más dinero gastó. Si hay más de una persona que gastó el monto máximo, desempata por ID de la persona.
Aliasing: No presenta aspectos de aliasing.

MENORSTOCK(**in** l: `lolla`, **in** i: `item`, **out** x: `bool`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {((∃ p : puestoid)(def?(p, puestos(l)) $\wedge_l$ stock(obtener(p, puestos(l))) > 0) $\longleftrightarrow$ x) $\wedge_l$ (x$\Rightarrow$L res = menorStock(l, i))}
Complejidad: $\Theta(P * log(I))$
Descripción: Obtiene el ID del puesto de menor stock para un ítem dado. Si hay más de un puesto que tiene stock mínimo, da el de menor ID. Si el item no esta en el menu de los puestos, devuelve false.
Aliasing: No presenta aspectos de aliasing.

PERSONAS(**in** l: `lolla`) → res : `conj(persona)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ personas(l)}
Complejidad: Θ(1)
Descripción: Obtiene a todas las personas del festival.
Aliasing: Devuelve una referencia no modificable al conjunto de personas de l.

PUESTOS(**in** l: `lolla`) → res : `dicc(puestoid,puesto)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ puestosl)}
Complejidad: Θ(1)
Descripción: Obtiene a todos los puestos del festival.
Aliasing: Devuelve una referencia no modificable al diccionario de puestos de l.

Representación

Representación de Lollapatuza

El festival se representa con varios diccionarios logaritmicos, conjunto, listas, vectores, cola de prioridad y constantes.

`lolla` se representa con `estr`

✓ donde `estr` es `tupla`(*puestos*: `diccLog(id, puesto)`, *puestosIts*: `diccLog(id, itDiccLog(id, puesto))`, *personas*: `conj(persona)`, *gastoPersona*: `diccLog(persona, nat)`, *masGastaron*: `colaMin(tupla(id: persona, gasto: dinero))`, *ordenGasto*: `vector(nat)` , *comprasHackeables*: `diccLog(persona, diccLog(item, tupla(cola: colaMin(tupla(id: puestoid, it: itDiccLog(puestoid,puesto))), enCola: diccLog(puestoid,bool))))`)

Rep : estr ⟶ bool
Rep(e) ≡ true ⟺ (1) $\wedge_l$ (2) $\wedge_l$ (3) ∧ (4) $\wedge_l$ (5) ∧ (6) ∧ (7) ∧ (8) ∧ (9) ∧ (10)

✓ 1. (1) ≡ claves(e.puestos) = claves(e.puestosIts)

✓ 2. (2) ≡ (∀ i: id)(def?(i, e.puestos) $\Rightarrow_l$
Siguiente(significado(i,e.puestosIts)) = significado(i, e.puestos))

✓ 3. (3) ≡ claves(e.gastoPersona) = e.personas = claves(e.comprasHackeables)

✓ 4. (4) ≡ long(e.ordenGasto) = max(e.personas) + 1

✓ 5. (5) ≡ (∃ p: persona)(p ∈ *claves*(*e.gastoPersona*)$\wedge_l$ (∀p$_2$: persona)(p_2 ∈ *claves*(*e.gastoPersona*)$\wedge_l$ p ≠p$_2$ $\Rightarrow_l$ obtener(p, e.gastoPersona) > obtener(p_2, e.gastoPersona) ∧ p = proximo(e.masGastaron))

✓ 6. (6) ≡ (∀ p: persona, i: item)(def?(p, e.comprasHackeables) $\wedge_l$ def?(i, obtener(p, e.comprasHackeables)) $\Rightarrow_l$
(∀ id: puestoId)((def?(id, obtener(i, obtener(p, e.comprasHackeables)).enCola))⟺
(∃ t: tupla)(esta?(t, obtener(i, obtener(p, e.comprasHackeables)).cola) $\wedge_l$ t.id = id ∧ siguiente(t.it) = obtener(t.id, e.puestos))))

✓ 7. (7) ≡ (∀ a: persona)(a ∈ *claves*(*e.gastoPersona*)$\Rightarrow_l$
Significado(e.gastoPersona, a) = SumaGastoPuestoAUX(Significados(e.puesto),a))

✓ 8. (8) ≡ (∀ i:persona)(0 ≤ i < long(e.ordenGasto) $\Rightarrow_l$ ((i ∉ *e.personas*)⟺ e.ordenGasto[i] < 0))

✓ 9. (9) ≡ (∀ i: nat)(i < long(e.ordenGasto) $\Rightarrow_l$ (e.ordenGasto[i] ≥ 0 ⟺ e.masGastaron[e.ordenGasto[i]].id = i))

✓ 10. (10) ≡ (∀ i:int)(0 ≤ i < tamaño(e.masGastaron) $\Rightarrow_l$ e.masGastaron[i].gasto = obtener(e.gastoPersona, e.masGastaron[i].id))

Abs : estr e ⟶ Lollapatuza {Rep(e)}
Abs(e) ≡ (1) ∧ (2)

1. (1) ≡ puestos(l) =$_{obs}$ e.puestos
2. (2) ≡ personas(a) =$_{obs}$ e.personas

Funciones auxiliares (TAD)

- **SumaGastoPuestoAUX**: cp: conj(puestos), a: persona ⇒ nat
  SumaGastoPuestoAUX(cp , a) ≡ if (∅?(cp)) then 0 else gastoDe(dameUno(cp) ,a) +
  SumaGastoPuestoAUX(a,dameUno(sinUno(cp)))

Algoritmos

---

iCrearLolla(**in** p: `dicc(id,puesto)`, **in** a: `conj(persona)`) → l : estr

```
1:  l.puestos ← p
2:  l.personas ← a
3:  l.GastoPersona ← Vacio()
4:  l.comprasHackeables ← Vacio()
5:  i ← CrearIt(a)
6:  z ← CrearIt(a)
7:  l.masGastaron ← Vacia()
8:  l.ordenGasto ← Vacia()
9:  //ordenGasto es un vector cuyas posiciones se corresponden con las personas, y lo que hay en cada posicion es el
    orden que mantiene la persona en la cola de prioridad "masGastaron"
10: //el tamaño de ordenGasto es el del maxId de una persona del conjunto, y en las posiciones del vector que no se
    corresponden con ninguna persona que este en el conjunto, se le indexa un -1.
11: contador ← 0
12: maxPersona
13: while HaySiguiente(z) do
14:     maxPersona ← Siguiente(z)
15:     Avanzar(z)
16: end while
17: k ← 0
18: while k < maxPersona + 1 do
19:     AgregarAtras(l.ordenGasto, -1)
20:     k++
21:     //-1 es una persona que no existe
22: end while
23: while HaySiguiente(i) do
24:     Definir(l.GastoPersona, Siguiente(i), 0)
25:     Definir(l.comprasHackeables, Siguiente(i), Vacio())
26:     Encolar(l.masGastaron, ⟨Siguiente(i), 0⟩)
27:     l.ordenGasto[Siguiente(i)] ← contador
28:     contador++
29:     //asumimos que el diccionario de personas se recorre en orden creciente
30:     Avanzar(i)
31: end while
32: j ← CrearIt(l.puestos)
33: while HaySiguiente(j) do
34:     Definir(l.puestosIts, SiguienteClave(j), j)
35:     Avanzar(j)
36: end while
37:
```

Complejidad: El primer while recorre todas las personas del festival, A veces.El segundo while recorre MaxId veces y usa AgregarAtras que es constante. El tercer while tambien recorre todas las personas del festival, A veces.Luego cada definir cuesta $\Theta(log(A))$ y $\Theta(log(P))$, encolar cuesta $\Theta(log(A))$ y lo demas tambien es $\Theta(1)$. El cuarto while recorre todos los puestos , P veces y usa definir. La complejidad es entonces: $\Theta(1)+\sum_{i=1}^{P}(\Theta(log(i)))+\sum_{i=1}^{A}(\Theta(log(i))) = \Theta(log(P) * P + log(A) * A + MaxId + A)$ Donde A es la cantidad total de personas distintas en el festival, P es la cantidad total de puestos distintos en el festival y MaxId es el id mas grande de las personas.

---

iTransaccion(**in/out** l : `estr`, **in** id : `nat` , **in** a : `persona` , **in** i : `item` , **in** c : `cant`)

```
1: huboDesc
2: gasto ← Vender(Significado(e.puestos, id), a , i , c, huboDesc)
3: Significado(l.GastoPersona, a) + = gasto
4: SiftUpTrack(l.masGastaron, l.ordenGasto[a], l.ordenGasto, gasto)
5: //cada vez que hay una transaccion se reacomoda la cola con los gastos de las personas
6: //mediante la funcion SiftUpTrack, y un vector que llamamos ordenGasto, que guarda las posiciones en el heap
7: //antes de cada transaccion.
8: //no usamos heapify porque en peor caso habria que reacomodar todo el heap, y en ese caso no seria Log(A).
9: if ¬huboDesc then
10:     if ¬Definido(Significado(l.comprasHackeables, a), i) then
11:         Definir(Significado(l.comprasHackeables, a), i, ⟨ ColaVacia(), Vacio() ⟩))
12:     end if
13:     if ¬Definido?(Significado(Significado(l.comprasHackeables, a), i).enCola,id) then
14:         encolar(Significado(Significado(l.comprasHackeables, a), i).cola, ⟨ id, Significado(l.puestosIts, id) ⟩)
15:         Definir(Significado(Significado(l.comprasHackeables, a), i).enCola, id, true)
16:     end if
17: end if
```

Complejidad: Significado() en diccLog es $\Theta(log(n))$ y tambien tenemos $\Theta(log(P))$ para encolar un nuevo id de puesto en la cola de prioridad. Vender tiene complejidad $\Theta(log(A)+log(I))$. Luego, usamos SiftUpTrack que tiene complejidad $\Theta(log(A))$. Queda entonces: $\Theta(log(A))+\Theta(log(I))+\Theta(log(P)) = \Theta(log(A)+log(I)+log(P))$ donde P es cantidad total de puestos, A cantidad total de personas, I cantidad total de items en el puesto a comprar.

---

iHackear(**in/out** l : `estr`, **in** i : `item` , **in** a : `persona`) → res : bool

```
1: if Definido?(Significado(a, l.comprasHackeables), i) then
2:     noHack
3:     puestoId ← Proximo(Significado(Significado(l.comprasHackeables, a), i).cola).id
4:     puestoIt ← Proximo(Significado(Significado(l.comprasHackeables, a), i).cola).it
5:     gasto ← Olvidar(SiguienteSignificado(puestoIt), a, i, noHack)
6:     Significado(l.gastoPersona, a) − = gasto
7:     SiftDownTrack(l.masGastaron, l.ordenGasto[a] ,l.ordenGasto, gasto)
8:     //este caso es analogo al de la transaccion, cuando tenemos que reducir el gasto de una persona
9:     //para el reordenamiento del heap usamos SiftDownTrack, y tambien utilizamos el vector ordenGasto
10:    //para guardarnos las posiciones de las personas en la cola.
11:    if noHack then
12:        Desencolar(Significado(Significado(l.comprasHackeables, a), i).cola)
13:    end if
14:    return true
15: else
16:    return false
17: end if
```

Complejidad: Significado() y Definido?() Sobre I, Significado sobre A. También usamos SiftDownTrack que como opera sobre todas las personas del festival tiene una complejidad $\Theta(log(A))$. Olvidar() cuesta $\Theta(log(A))+\Theta(log(I))$, o sea no cambia la complejidad. Normalmente hackear es $\Theta(log(A)) + \Theta(log(I)) = \Theta(log(A) + log(I))$. Si el puesto deja de ser hackeable , para ese item, es decir la cantidad llega a cero, entonces podemos tener $\Theta(log(P))$ por desencolar ese id de puesto. (eliminar el nodo del heap es $O(log(n))$ en peor caso). Hackear entonces es $\Theta(log(A))+\Theta(log(I))+\Theta(log(P)) = \Theta(log(A)+log(I)+log(P))$ donde A es la cantidad de personas que compraron algo en el puesto e I es la cantidad de items distintos que compró la persona a en el puesto sin descuento.

GastoPersona(**in** l : `estr`, **in** a : `persona`) → res : nat

```
1: return Significado(l.GastoPersona, a)
```

Complejidad: $\Theta(log(A))$ donde A es la cantidad total de personas en el festival.

PersonaQueMasGasto(**in** l : `estr`) $\rightarrow$ res : persona

1: **return** Proximo(l.masGastaron)

Complejidad: $\Theta(1)$

---

iMenorStock(**in** l : `estr`, **in** i : `item`, **out** x : `bool`) $\rightarrow$ res : puestoid

1: $it \leftarrow CrearIt(l.puestos)$
2: $stockDelMenor \leftarrow 0$
3: $res \leftarrow 0$
4: **while** HaySiguiente(it) **do**
5: $p \leftarrow SiguienteSignificado(it)$
6: $pid \leftarrow SiguienteClave(it)$
7: **if** Stock(p, i) > 0 **then**
8: **if** stockDelMenor == 0 || (stockDelMenor > Stock(p, i)) || (stockDelMenor == Stock(p, i) & & res > pid) **then**
9: $stockDelMenor \leftarrow Stock(p, i)$
10: $res \leftarrow pid$
11: **end if**
12: **end if**
13: **end while**
14: $x \leftarrow stockDelMenor! = 0$
15: **return** res

Complejidad: El primer while recorre todos los puestos del festival, P veces, y por cada vez uso stock() que tiene complejidad $\Theta(log(I)))$. Exceptuando el resto que es $\Theta(1)$ queda: $\sum_1^P(\Theta(log(I))) = \Theta(P * log(I))$ donde P es la cantidad de puestos en el festival e I es la cantidad de items totales.

---

iPersonas(**in** l : `estr`)) $\rightarrow$ res : conj(persona)

1: **return** l.personas

Complejidad: $\Theta(1)$

---

iPuestos(**in** l : `estr`)) $\rightarrow$ res : dicc(id,puesto)

1: **return** l.Puestos

Complejidad: $\Theta(1)$

1.1. Modulo Puesto de Comida

se explica con: Puesto De Comida.
géneros: `puesto`.

Interfaz

Operaciones básicas

CrearPuesto(**in** p: `dicc(item,nat)`, **in** s: `dicc(item,nat)`, **in** d: `dicc(item,dicc(cant,nat))`) $\rightarrow$ res : `puesto`
Pre $\equiv$ {claves(p) = claves(s) $\wedge$ claves(d) $\subseteq$ claves(p)}
Post $\equiv$ {$res =_{\text{obs}}$ crearPuesto(p,s,d)}
Complejidad: $\Theta(I * m)$
Descripción: Crea un nuevo puesto
Aliasing: No presenta aspectos de aliasing. (1) + (((1))) = (I*m) Stock(**in** p: `puesto`, **in** i: `item`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$true$}
Post $\equiv$ {(i $\in$ $menu(p)$ $\Rightarrow_l$res= $stock(p, i)$)$\wedge$ ($\neg$($i \in menu(p) \Rightarrow_l$res= 0))}
Complejidad:$\Theta(log(I))$
Descripción: Devuelve el stock de un item en un puesto.
Aliasing: Devuelve una copia del valor del dicc stock.

Descuento(**in** p: `puesto`, **in** i: `item` , **in** c: `cant`) $\rightarrow$ res : `nat`
Pre $\equiv$ {i $\in$ $menu(p)$}
Post$\equiv$ {$res =_{\text{obs}}$ descuento(p,i,c)}
Complejidad: $\Theta(1)$
Descripción: Devuelve un descuento para un determinado item y una cantidad.
Aliasing: Devuelve una copia del valor del dicc descuentos.

Gasto(**in** p: `puesto`, **in** a: `persona`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}}$ gastosDeVentas(p, ventas(p, a))}
Complejidad: $\Theta(log(A))$
Descripción: Devuelve el gasto total de una persona en un puesto.
Aliasing: Devuelve copia.

Vender(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`, **out** $huboDesc$: `bool`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {$p = p_0 \wedge$ c $>$ 0 $\wedge$ stock(p_0, i) $\geq c$}
Post$\equiv$ {p = vender(p_0, a, i, c) $\wedge_l$ $res = gastosDe1Venta(p, <i, c>)$}
Complejidad:$\Theta(log(I) + log(A))$ donde I es la cantidad de items en el puesto y A es la cantidad de personas que compraron algo en el puesto.
Descripción: Vende un item, devuelve el costo de la operacion, y retorna true si hubo descuento en la venta.
Aliasing: Recibe huboDesc, referencia modificable y lo convierte en true si hubo descuento o false si no.

Olvidar(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `itembool`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {$p = p_0 \wedge$ i $\in$ menu(p) $\wedge$ consumioSinPromo?(p,a,i)}
Post $\equiv$ {olvidarItem(p_0, a, i) = p}
Complejidad:$\Theta(log(I) + log(A))$
Descripción: Olvida una venta de un item en un puesto, tambien repone el stock del item, devuelve el gasto de la persona en el puesto y retorna el valor del item.
Aliasing: Recibe noHack, referencia modificable y lo convierte en true si el puesto ya no es hackeable (ventasSin-

Descuento para un item en particular esta vacia)

Representación

Representación de Puesto de Comida

`puesto` se representa con `estr`

donde `estr` es `tupla`(*stock*: `diccLog(item, cantidad)`, *descuentos*: `diccLog(item, tupla(desc: vector(descuento), cortes: vector(nat))`, *gastoPorPersona*: `diccLog(persona, gastoTotalEnPuesto)`, *precios*: `diccLog(item, precio)`, *ventas*: `diccLog(persona, lista(tupla(item: item, cantidad: cantidad)))`, *ventasSinDescuento*: `diccLog(persona, diccLog(item, lista(itLista(tupla(item, cantidad)))))`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ (1) $\wedge$ (2) $\wedge_l$ (3) $\wedge$ (4) $\wedge$ (5)

1. (1) $\equiv$ claves(e.descuentos) $\subseteq$ claves(e.stock) $\wedge$ claves(e.precios) = claves(e.stock) $\wedge$ claves(e.ventas) = claves(e.gastoPorPersona) $\wedge$ claves(e.ventas) = claves(e.ventasSinDescuento)
2. (2) $\equiv$ ($\forall$ i: item)(def?(i, e.descuentos) $\Rightarrow_l$ ordenado(obtener(i, e.descuentos).cortes) $\wedge$ long(obtener(i, e.descuentos).cortes) > 0 $\wedge_l$ obtener(i, e.descuentos).cortes[0] = 0 $\wedge$ long(obtener(i, e.descuentos).desc) = obtener(i, e.descuentos).cortes[long(obtener(i, e.descuentos).cortes)]+1)
3. (3) $\equiv$ ($\forall$i: nat)(i < long(cortes)-1 $\Rightarrow_l$ ($\forall j : nat$)($cortes[i] < j < cortes[i+1] \Rightarrow_l$ desc[j] = desc[cortes[i]]))
4. (4) $\equiv$ ($\forall a : persona$)($def?(a, e.ventas) \Rightarrow_l$ gastosDeVentas(p, obtener(a, e.ventas)) = obtener(a, e.gastoPorPersona))
5. (5) $\equiv$ ($\forall a : persona, i : item$)($def?(a, e.ventasSinDescuento) \wedge_l$ def?(i, obtener(a, e.ventasSinDescuento))

Abs : estr e $\longrightarrow$ PuestoDeComida {Rep(e)}
Abs(e) $\equiv$ (1) $\wedge$ (2) $\wedge$ (3) $\wedge$ (4) $\wedge$ (5)

1. (1) $\equiv$ menu(p) $=_{\text{obs}}$ claves(e.stock)
2. (2) $\equiv$ ($\forall i : item$)($def?(i, e.precios) \Rightarrow_l$ precio(p,i) = obtener(i,e.precios) $\wedge$ stock(p,i) = obtener(i,e.stock))
3. (3) $\equiv$ ($\forall i$: $item$)($def?(i, e.descuentos) \Rightarrow_l$ ($\forall c$: $cantidad$)($def?(c, obtener(c, obtener(i, e.descuentos))) \Rightarrow_l$ descuento(p,i,c) = obtener(c, obtener(i,e.descuentos)))
4. (4) $\equiv$ ($\forall i$: $item$)($def?(i, e.descuentos) \Rightarrow_l$ ($\forall c$: $cantidad$)(¬def?(c,obtener(c, obtener(i,e.descuentos))) $\Rightarrow_l$ descuento(p,i,c) = 0 $\vee_l$ ($\exists$ c2: cantidad)((c2 < c) $\wedge_l$ obtener(c2, obtener(i,e.descuentos) = descuento(p,i,c))))
5. (5) $\equiv$ ($\forall a : persona$)($def?(a, e.ventas) \Rightarrow_l$ ventas(p,a) = obtener(a, e.ventas))

Algoritmos

---

iCrearPuesto(**in** p: `dicc(item,nat)`, **in** s: `dicc(item, nat)`, **in** d: `dicc(item, dicc(cant, nat))`) → res : estr

```
1:  res.stock ← s
2:  res.precios ← p
3:  res.GastoPorPersona ← Vacio()
4:  res.ventas ← Vacio()
5:  res.descuento ← Vacio()
6:  res.ventasSinDescuento ← Vacio()
7:  itItems ← CrearIt(d)
8:  while HaySiguiente(itItems) do
9:      desc ← Vacia()
10:     AgregarAtras(desc, 0)
11:     cortes ← Vacia()
12:     AgregarAtras(cortes, 0)
13:     itDesc ← CrearIt(SiguienteSignificado(itItems))
14:     pos ← 0
15:     prev ← 0
16:     while HaySiguiente(itDesc) do
17:         AgregarAtras(cortes, SiguienteClave(itDesc))
18:         while pos < SiguienteClave(itDesc) do
19:             AgregarAtras(desc, prev)
20:             pos++
21:         end while
22:         prev ← SiguienteSignificado(itDesc)
23:         Avanzar(itDesc)
24:     end while
25:     Definir(res.descuentos, SiguienteClave(itItems), ⟨desc, cortes⟩)
26:     Avanzar(itItems)
27: end while
28: return res
```

Complejidad: El while grande recorre todos los items del puesto, I veces. Los otros dos while iteran maxCant veces donde maxCant es la cantidad más grande del item a partir de la cual hay un descuento. Por lo que exluyendo el resto que es $\Theta(1)$ la complejidad total es: $\sum_1^I(\sum_1^m(\Theta(1))) = \Theta(I * m)$ donde I es la cantidad de items distintos en el puesto y m es la mayor cantidad a partir de la cual hay un descuento en cualquier item.

---

iStock(**in** e: `estr`, **in** i: `item`) → res : nat

```
if Definido?(e.stock, i) then
    res ← Significado(e.stock, i)
else
    res ← 0
end if
```

Complejidad: Definido?() y Significado() en diccLog es $\Theta(log(n))$
$\Theta(log(I)) + \Theta(log(I)) = \Theta(log(I))$ Donde I es la cantidad de items distintos en el puesto.

---

iDescuento(**in** e : `estr`, **in** i : `item` , **in** c : `cant`) $\rightarrow$ res : nat

```
1:  if Definido?(e.descuentos, i) then
2:      descuentosItem ← Significado(e.descuentos, i)
3:      if c ≥ Longitud(descuentosItem.desc) then
4:          res ← descuentosItem.desc[long(descuentosItem.desc) − 1]
5:      else
6:          res ← descuentosItem.desc[c]
7:      end if
8:  else
9:      res ← 0
10: end if
```

Complejidad: Como tenemos todos los descuentos organizados en un vector, lo unico que no es $\Theta(1)$ es Definido?() y Significado() por lo que queda una complejidad total de $\Theta(log(I))$ donde I es la cantidad de items distintos en el puesto.

---

iGasto(**in** e : `estr`, **in** a : `persona`) $\rightarrow$ res : nat

```
1: if Definido?(e.gastoPorPersona, a) then
2:     res ← Significado(e.gastoPorPersona, a)
3: else
4:     res ← 0
5: end if
```

Complejidad: $\Theta(log(A))$ por Definido?() y Significado() donde A es la cantidad de personas distintas que compraron algo en el puesto.

---

iVender(**in/out** p : `estr`, **in** a : `persona`, **in** i : `item`, **in** c : `cant`, **out** $huboDesc$: `bool`) $\rightarrow$ res : dinero

```
1:  Significado(p.stock, i) ← Significado(p.stock, i) − c
2:  desc ← Descuento(p, i, c)
3:  valorVenta ← ((100 − desc) ∗ Significado(p.precio, i))/100
4:  Significado(p.gastoPorPersona, a) ← Significado(p.gastoPorPersona, a) + valorVenta
5:  huboDesc ← desc > 0
6:  if ¬Definido?(p.ventas, a) then Definir(p.ventas, a, Vacía())
7:  end if
8:  it ← AgregarAtras(Significado(p.ventas, a), <i, c>)
9:  if ¬huboDesc then
10:     if ¬Definido?(p.ventasSinDescuento, a) then
11:         Definir(p.ventasSinDescuento, a, Vacío())
12:     end if
13:     if ¬Definido?(Significado(p.ventasSinDescuento, a), i) then
14:         Definir(Significado(p.ventasSinDescuento, a), i, Vacía())
15:     end if
16:     AgregarAtras(Significado(Significado(p.ventasSinDescuento, a), i), it)
17: end if
18: return valorVenta
```

Complejidad: Definido?() y Significado() sobre I y luego sobre A $\Rightarrow \Theta(log(A)) + \Theta(log(I)) = \Theta(log(A) + log(I))$

iOlvidar(**in/out** p: `estr`, **in** a: `persona`, **in** i: `item`, **out** $noHack$: `bool`) $\rightarrow$ res : dinero

1: $compra \leftarrow Ultimo(Significado(Significado(p.ventasSinDescuento, a), i))$
2: Siguiente(compra).cant−−
3: **if** Siguiente(compra).cant == 0 **then**
4: EliminarSiguiente(compra)
5: EliminarAnterior(CrearItUlt(Significado(Significado(p.ventasSinDescuento, a), i)))
6: **end if**
7: Significado(p.stock, i)++
8: $res \leftarrow Significado(p.precios, i)$
9: $Significado(p.gastoPorPersona, a) \leftarrow Significado(p.gastoPorPersona, a) - res$
10: $noHack \leftarrow false$
11: **if** EsVacía?(Significado(Significado(p.ventasSinDescuento,a), i)) **then**
12: $noHack \leftarrow true$
13: **end if**
14: **return** res

Complejidad: Significado() sobre I y luego sobre A $\Rightarrow \Theta(log(A) + log(I))$

2. Módulo Cola de prioridad min heap

Interfaz

Parámetros formales
géneros α
función COPIAR(**in** a : $colaMin(\alpha)$) $\rightarrow$ res : $colaMin(\alpha)$
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de cola min heap
función $\bullet < \bullet$(**in** $c1, c2$: $colaMin(\alpha)$) $\rightarrow$ res : $colaMin(\alpha)$
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (c1 < c2)$}
Complejidad: $\Theta(1)$
Descripción: función de copia de cola min heap

se explica con: COLA DE PRIORIDAD(α).

géneros: `colaMin(`α`)`.

Operaciones básicas

VACIA() $\rightarrow$ res : $colaMin(\alpha)$
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}}$ vacia}
Complejidad: $\Theta(1)$

HEAPIFY(**in** v : `vector(`α`)`) $\rightarrow$ res : `colaMin(`α`)`
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}}$ heapify(v)}
Complejidad: $\Theta(n)$

VACIA?(**in** c: `colaMin(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}}$ vacia?(c)}
Complejidad: $\Theta(1)$

ENCOLAR(**in/out** c: `colaMin(`α`)`, **in** e: α)
Pre $\equiv$ {c = c_0}

Post $\equiv$ {c = encolar(e, c_0)}
Complejidad: $\Theta(log(n))$

DESENCOLAR(**in/out** c: `colaMin(`α`)`)
Pre $\equiv$ {c = c_0}
Post $\equiv$ {$res =_{obs}$ $\neg$vacia(c_0) $\wedge_l$ (res $\Rightarrow_l$ c = desencolar(c_0)) $\wedge$ ($\neg$res $\Rightarrow$ c = c_0)}
Complejidad: $\Theta(log(n))$

PROXIMO(**in** c: `colaMin(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {$\neg$vacia(c)}
Post $\equiv$ {res $=_{obs}$ proximo(c)}
Complejidad: $\Theta(1)$

SIFTUPTRACK(**in/out** c: `colaMin(`$\langle$`persona, dinero`$\rangle$`)`,**in** i: `nat`, **in/out** v: `vector(dinero)`, **in** $canti$: `dinero`)
Pre $\equiv$ {c = c_0 $\wedge_l$ v = v_0 $\wedge_l$ $\neg$vacia(c) $\wedge_l$ $\neg$vacia(v) $\wedge_l$ long(v_0) = maxId(c_0)}
Post $\equiv$ {c $=_{obs}$ compraCola(c_0, i, canti) $\wedge_l$ v = fixVectorSegunCola(v_0, c)}
Complejidad:$\Theta(log(n))$

SIFTDOWNTRACK(**in/out** c: `colaMin(`$\langle$`persona, dinero`$\rangle$`)`,**in** i: `nat`, **in/out** v: `vector(dinero)`, **in** $canti$: `dinero`)
Pre $\equiv$ {c = c_0 $\wedge_l$ v = v_0 $\wedge_l$ $\neg$vacia(c) $\wedge$ $\neg$vacia(v) $\wedge_l$ long(v_0) = maxId(c_0)}
Post $\equiv$ {c $=_{obs}$ hackeoCola(c_0, i, canti) $\wedge_l$ v = fixVectorSegunCola(v_0, c)}
Complejidad:$\Theta(log(n))$ donde n es el tamaño de la cola.

Representación

Representación de cola de prioridad min heap

`colaMin(`α`)` **se representa con** `estr`

vector(α)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ ($\forall$i: nat)(0 < i < Longitud(e) $\Rightarrow_l$ e[i] $\leq$ e[(i-1)/2])

Abs : estr e $\longrightarrow$ Cola de prioridad {Rep(e)}
Abs(e) $\equiv$ c cola(α) / heapify(e) = c

Funciones auxiliares (TAD)

- **heapify**: v: secu(α) $\Rightarrow$ cola(α)
  heapify(v) $\equiv$ heapifyAux(v, long(v)/2 - 1)
- **heapifyAux**: v: secu(α), i: nat $\Rightarrow$ secu(α)
  heapifyAux(v, i) $\equiv$ if i > 0 then heapifyAux(siftDown(v, i), i-1) else siftDown(v, i)

Algoritmos

---

iVacia() $\rightarrow$ v : $colaMin(\alpha)$
  $vector(\alpha)$ v $\leftarrow$ $Vacia()$
  return v

  Complejidad: $\Theta(1)$

---

iHeapify(**in** v : `vector(`α`)`) $\rightarrow$ res : colaMin(α)

$i \leftarrow n/2$
while i >= 0 **do**
    $SiftDown(v, i)$
    $i--$
end while
return v

Complejidad: $\Theta(n)$, pues el numero maximo posible de intercambios es n.

---

iVacia?(**in** c: `estr`) $\rightarrow$ bool

return $Longitud(c) == 0$

Complejidad: $\Theta(1)$

---

iEncolar(**in/out** c: `estr`, **in** e: α)

$AgregarAtras(c, e)$
$SiftUp(c, Longitud(c) - 1)$

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

---

iDesencolar(**in/out** c: `estr`)

$Swap(c, 0, Longitud(c) - 1)$
$Eliminar(c, Longitud(c) - 1)$
$SiftDown(c, 0)$

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

---

iProximo(**in** c: `estr`) $\rightarrow$ res : α

return c[0]

Complejidad: $\Theta(1)$

SiftUp(**in/out** c: `estr`, **in** i: `nat`)

while $i > 0$ & & $c[i] < c[(i-1)/2]$ **do**
    $Swap(c, i, (i-1)/2)$
    $i \leftarrow (i-1)/2$
end while

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

SiftDown(**in/out** c: `estr`, **in** i: `nat`)

```
izq ← i * 2 + 1
der ← i * 2 + 2
while i < Longitud(c)/2 do
    if der < Longitud(c) & & min(c[izq], c[der]) < c[i] then
        Swap(c, i, minIndex(c, izq, der))
        i ← minIndex(c, izq, der)
        izq ← i * 2 + 1
        der ← i * 2 + 2
    else
        if c[izq] < c[i] then
            Longitud(c)/2 − 1
            Swap(c, i, izq)
            i ← izq
        end if
    end if
end while
```

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

SiftUpTrack(**in/out** c: `estr`, **in** i: `nat`, **in** v: `vector(dinero)`, **in** *canti*: `dinero`)

```
//SiftUpTrack es una version modificada de SiftUp, que nos permite modificar el vector ordenGasto, a medida que se produce el reordenamiento de las personas en el heap
c[i].gasto + = canti
while i > 0 && c[i] < c[(i − 1)/2] do
    actual ← c[i].id
    old ← c[(i-1)/2].id
    v[old] ← i
    v[actual] ← (i-1)/2
    Swap(c,i, (i-1)/2)
    i ← (i-1)/2
end while
```

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

SiftDownTrack(**in/out** c: `estr`, **in** i: `nat`, **in** v: `vector(dinero)`, **in** $canti$: `dinero`)

```
//SiftDownTrack es una version modificada de SiftDown, que nos permite modificar el vector ordenGasto, a medida que se produce el reordenamiento de las personas en el heap.
c[i].gasto − ← canti
izq ← i * 2 + 1
der ← i * 2 + 2
while i < Longitud(c)/2 do
    actual ← c[i].id
    if der < Longitud(c) & & min(c[izq], c[der]) < c[i] then
        nuevaPos ← minIndex(c, izq, der)
        old ← c[nuevaPos].id
        v[old] ← i
        v[actual] ← nuevaPos
        Swap(c, i, nuevaPos)
        i ← nuevaPos
        izq ← i * 2 + 1
        der ← i * 2 + 2
    else
        if c[izq] < c[i] then
            old ← c[nuevaPos].id
            v[old] ← i
            v[actual] ← izq
            Swap(c, i, izq)
            i ← izq
        end if
    end if
end while
```

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

A continuacion definimos las relaciones de orden para las tuplas que usamos a lo largo del trabajo.

• $<$ •(**in** $t1, t2$: $\langle$`puestoid, it(diccLog(puestoid, puesto))`$\rangle$) $\rightarrow$ *bool*

```
return t1.id < t2.id
```

• $<$ •(**in** $t1, t2$: $\langle$`id: persona, gasto: dinero`$\rangle$) $\rightarrow$ *bool*

```
return t1.gasto < t2.gasto || (t1.gasto == t2.gasto && t1.id < t2.id)
```