02 de Junio de 2023

Algoritmos y Estructuras de Datos II

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

TP de Diseño

Lollapatuza

Grupo Compsci

Integrante	LU	Correo electrónico
Rossi, Facundo	303/22	facurossi0211@gmail.com
Alfaro, Gianfranco	148/22	gian1986x@gmail.com
Castro Sendega, Emiliano Gaston	467/22	emilianocastrosendega@gmail.com
Axel, Campoverde	258/22	a.i.cpvd3000@gmail.com

Correcciones:
- La funcion eliminar del heap no cumple las complejidades que indican.

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega	Maximiliano Martino	Aprobado
Segunda entrega		

```
1  // Aclaraciones: ✓1. Asumimos que el iterador de diccLog recorre en orden de menor a mayor las claves como
   comentaron por mail.
2  //             ✓2. Asumimos que agregarAtras del módulo vector es siempre θ(1) (en el caso de un vector
   de nats o tuplas por ejemplo) como nos comentaron por mail.
3  //             ✓3. Asumimos que si tenemos un puntero a un significado de un diccionario, si se redefine
   la clave actualizando ese significado, el puntero no se invalida.
4
5  // Puesto De Comida
6  Interfaz
7      se explica con: PuestoDeComida
8
9      géneros: puesto
10
11     operaciones:
12         ABRIRPUESTO(in s: dicc(nat, nat), in m: dicc(nat, nat), in desc: dicc(nat, dicc(nat, nat))) → 
   res: puesto
13       ✓Pre ≡ {claves(s) = claves(m) ∧ (claves(desc) ⊆ claves(m) ∨ ∅?(claves(desc)))}
14       ✓Post ≡ {res =obs crearPuesto(m, s, desc)}
15         Descripción: Inicializa un nuevo puesto a partir de un menú, stock de los items del menú y
   descuentos sobre los items (puede no haber).
16         Complejidad: θ(I * (log(I) + D' * D)), donde I = #claves(m), D = máxCantDescuentos(desc) y D' =
   máxCantidadMinimaConDescuento(desc).
17         Aliasing: res es modificable.
18
19         VENDER(in per: nat, in i: nat, in cant: nat, in/out p: puesto)
20       ✓Pre ≡ {i ∈ menu(p) ∧ p =obs p₀ ∧L stock(p, i) ≥ cant}
21       ✓Post ≡ {p =obs vender(p₀, per, i, cant))}
22         Descripción: Realiza una venta de cant unidades de i a la persona per.
23         Complejidad: θ(log(A) + log(I)), donde I = #menu(p) y A es la cantidad de personas que hicieron
   una compra en el puesto.
24
25         PORCENTAJEDESCUENTO(in i: nat, in cant: nat, in p: puesto) → res: nat
26       ✓ Pre ≡ {i ∈ menu(p)}
27       ✓ Post ≡ {res =obs descuento(p, i, cant)}
28         Descripción: Devuelve el porcentaje de descuento aplicable a la cantidad cant para el ítem i.
29         Complejidad: θ(log(I)), donde I = #menu(p).
30         Aliasing: res es no modificable.
31
32         ESTÁENMENÚ?(in i: nat, in p: puesto) → res: bool
```

```
33        ✓Pre ≡ {true}
34        ✓Post ≡ {res =obs i ∈ menu(p)}
35          Descripción: Devuelve true si y sólo si el puesto p vende el ítem i.
36          Complejidad: θ(log(I)), donde I = #menu(p).
37
38          STOCKITEM(in i: nat, in p: puesto) → res: nat
39        ✓Pre ≡ {i ∈ menu(p)}
40        ✓Post ≡ {res =obs stock(p, i)}
41          Descripción: Devuelve el stock que tiene el puesto p del ítem i.
42          Complejidad: θ(log(I)), donde I = #menu(p).
43          Aliasing: res es no modificable.
44
45          CUÁNTOGASTÓ?(in per: nat, in p: puesto) → res: nat
46        ✓Pre ≡ {true}
47        ✓Post ≡ {res =obs gastosDe(p, per)}
48          Descripción: Devuelve cuánto gastó la persona per en el puesto p.
49          Complejidad: θ(log(A)), donde A es la cantidad de personas que hicieron una compra en el puesto.
50          Aliasing: res es no modificable.
51
52          PRECIO(in i: nat, in p: puesto) → res: nat
53        ✓Pre ≡ {i ∈ menu(p)}
54        ✓Post ≡ {res =obs precio(p,i)}
55          Descripción: Devuelve el precio que tiene el ítem i en el puesto p.
56          Complejidad: θ(log(I)), donde I = #menu(p).
57          Aliasing: res es no modificable.
58
59          COMPRÓITEMSINDESCUENTO?(in per: nat, in i: nat, in p: puesto) → res: bool
60        ✓Pre ≡ {i ∈ menu(p)}
61        ✓Post ≡ {res =obs consumioSinPromo?(p, per, i)}
62          Descripción: Devuelve true si y solo si la persona per realizó una compra del ítem i en la cual
    no se le aplicó un descuento.
63          Complejidad: θ(log(A) + log(I)), donde I = #menu(p) y A es la cantidad de personas que hicieron
    una compra en el puesto.
64
65          OLVIDARCOMPRASINDESCUENTO(in per: nat, in i: nat, in/out p: puesto)
66        ✓Pre ≡ {p =obs p₀ ∧ i ∈ menu(p) ∧L consumioSinPromo?(p, per, i)}
67        ✓Post ≡ {p =obs olvidarItem(p₀, per, i)}
68          Descripción: Elimina el registro de la compra de una unidad del ítem i (aunque haya sido una
    compra de más de una unidad, tan solo se olvida una) hecha por la persona per en la cual no se le aplicó
    un descuento.
```

```
69          Complejidad: θ(log(A) + log(I)), donde I = #menu(p) y A es la cantidad de personas que hicieron
    una compra en el puesto.
70
71          COPIAR(in p: puesto) → res: puesto
72        ✓Pre ≡ {true}
73        ✓Post ≡ {res =obs p}
74          Complejidad: θ(I + A + L), donde I = #menu(p), A es la cantidad de personas que hicieron una
    compra en el puesto y L es la cantidad de compras que hizo la persona que más veces compró.
75          Descripción: Genera una nueva copia del puesto.
76
77  Fin Interfaz
78
79  - Especificación de funciones auxiliares usadas:
80  TAD Diccionario Extendido
81
82      géneros: dicc(clave, significado)
83
84    ✓extiende: DICCIONARIO(CLAVE, SIGNIFICADO)
85
86    ✓otras operaciones:
87          máxCantDescuentos: dicc(nat, dicc(nat, nat)) → nat
88          máxCantidadMinimaConDescuento: dicc(nat, dicc(nat, nat)) → nat
89          máxCantidadMinimaConDescuentoItem: dicc(nat, nat) → nat
90
91    ✓axiomas:
92          (∀d: dicc(nat, dicc(nat, nat)), d': dicc(nat, nat))
93          máxCantDescuentos(d) ≡ if ∅(claves(d)) then 0 else máx(#claves(obtener(dameUno(claves(d)), d)),
    máxCantClavesEnSignificado(borrar(dameUno(claves(d))), d)) fi
94          máxCantidadMinimaConDescuentoItem(d') ≡ if ∅(claves(d')) then 0 else máx(dameUno(claves(d')),
    máxCantidadMinimaConDescuentoItem(borrar(dameUno(claves(d')), d')) fi
95          máxCantidadMinimaConDescuento(d) ≡ if ∅(claves(d)) then 0 else
    máx(máxCantidadMinimaConDescuentoItem(obtener(dameUno(claves(d)), d))),
    máxCantidadMinimaConDescuento(borrar(dameUno(claves(d))), d)) fi
96
97  Fin TAD
98
99  ✓// item es renombre de nat.
100 ✓// persona es renombre de nat, representa el DNI de una persona.
101 ✓// cantidad es renombre de nat.
102 ✓// puestoID es renombre de nat.
103 ✓// precio es renombre de nat.
```

```
104 ✓// porcentaje es renombre de nat.
105
106  representación:
107      puesto se representa con estr, donde
108          estr es tupla⟨
109              menú: diccLog(item, precio)
110              stock: diccLog(item, cantidad)
111              descuentos: diccLog(item, vector(porcentaje))
112              gastoPorPersona: diccLog(persona, nat)
113              comprasPorPersona: diccLog(persona, diccLog(item, lista(cantidad)))
114              comprasSinDescuento: diccLog(persona, diccLog(item, cola(itLista(cantidad))))
115          ⟩
116
117  invariante de representación:
118      Rep: estr → bool
119    ✓(∀p: estr) Rep(p) ≡ true ⇔ (1) ∧ (2)  (3) ∧ (4) ∧ʟ (5) ∧ (6) ∧ʟ (7) ∧ʟ (8)
120
121  // Los ítems de menú y stock son los mismos, y todos los ítems de descuento están en el menú:
122 ✓(1) ≡ claves(p.menú) = claves(p.stock) ∧ claves(p.descuentos) ⊆ claves(p.menú)
123  // Las personas que tienen gastos registrados son las mismas que tienen compras registradas, y las que
     tienen alguna compra sin descuento están incluidas en las que tienen alguna compra:
124 ✓(2) ≡ claves(p.gastoPorPersona) = claves(p.comprasPorPersona) ∧ claves(p.comprasSinDescuento) ⊆
     claves(p.comprasPorPersonas)
125  // Los ítems que figuren como comprados por una persona deben estar en el menú:
126 ✓(3) ≡ (∀a: persona)(def?(a, p.comprasPorPersona) ⇒ʟ claves(obtener(a, p.comprasSinDescuento)) ⊆
     claves(p.menú)))
127  // Los ítems que figuren como comprados sin descuento por una persona deben estar en el menú:
128 ✓(4) ≡ (∀a: persona)(def?(a, p.comprasSinDescuento) ⇒ʟ claves(obtener(a, p.comprasSinDescuento)) ⊆
     claves(p.menú)))
129  // Los gastos de cada persona corresponden con las compras que hicieron en base a los precios del menú y
     sus descuentos:
130 ✓(5) ≡ (∀a: persona)(def?(a, p.gastoPorPersona) ⇒ʟ obtener(a, p.gastoPorPersona) = ∑i∈items(∑j=0 to l-
     1(cantidades[j] x obtener(i, p.menú) × (100-discount(p.descuentos, i, cantidades[j]))/100)))
131  // Si un ítem figura como comprado sin descuento por alguna persona, entonces también está definido en
     comprasPorPersona para esa persona.
132 ✓(6) ≡ (∀a: persona)(def?(a, p.comprasSinDescuento) ⇒ʟ (∀i: item)(def?(i, obtener(a,
     p.comprasSinDescuento)) ⇒ʟ def?(i, obtener(a, p.comprasPorPersona))))
133  // Ninguna persona puede tener más cantidad iteradores de compras sin descuento que la longitud de su
     lista de compras.
```

```
(7) ≡ (∀a: persona)(def?(a, p.comprasSinDescuento) ⇒ʟ (∀i: item)(def?(i, obtener(a,
p.comprasSinDescuento)) ⇒ʟ tamaño(obtener(i, obtener(a, p.comprasSinDescuento))) ≤ long(obtener(i,
obtener(a, p.comprasPorPersona))))
// Para cada persona que haya hecho una compra sin descuento y para cada item que compró sin descuento,
cada iterador en la cola de iteradores debe 'estar apuntando' efectivamente a una posición de la lista en
p.comprasPorPersona donde la cantidad comprada no tiene descuento aplicable.
(8) ≡ (∀per: persona)(def?(per, p.comprasSinDescuento) ⇒ʟ (∀i: item)(def?(i, obtener(per,
p.comprasSinDescuento)) ⇒ʟ (∀n : nat)(n < tamaño(obtener(i, obtener(per, p.comprasSinDescuento))) ⇒ʟ
discount(p.descuentos, i, Siguiente(obtener(i, obtener(per, p.comprasSinDescuento))[n])) = 0 ∧
long(Anteriores(obtener(i, obtener(per, p.comprasSinDescuento))[n])) < long(cantidades) ∧ʟ
cantidades[long(Anteriores(obtener(i, obtener(per, p.comprasSinDescuento))[n]))] = Siguiente(obtener(i,
obtener(per, p.comprasSinDescuento))[n]))
// En (5), (6) y (8):
// items ≡ claves(obtener(a, p.comprasPorPersona))
// l ≡ long(obtener(i, obtener(a, p.comprasPorPersona)))
// cantidades ≡ obtener(i, obtener(a, p.comprasPorPersona))

función de abstracción:
    Abs: estr e → puesto {Rep(e)}
    (∀e: estr) Abs(e) =obs p: puesto | menu(p) =obs claves(e.menú) ∧ʟ (∀i: item)(i ∈ menu(p) ⇒ʟ
precio(p, i) =obs obtener(i, e.menú) ∧ stock(p, i) =obs obtener(i, e.stock) ∧ (∀c: cant)(descuento(p, i,
c) =obs discount(e.descuentos, i, c))) ∧ (∀a: persona)((¬def?(a, e.comprasPorPersona) ⇒ ventas(p, a)
=obs ∅) ∧ (def?(a, e.comprasPorPersona) ⇒ʟ ventas(p, a) =obs hacerMultiConjVentas(a, obtener(a,
e.comprasPorPersona))


- Auxiliares usados para el invariante de representación y la función de abstracción:
•[•] : secu(α) s x nat n → α {n < long(s)}
•[•] : cola(α) c x nat n → α {n < tamaño(c)}
discount: dicc(item, secu(nat)) d x item i x cant c → nat
hacerMultiConjVentas: dicc(item, secu(nat)) → multiconj(<item, cant>)
hacerTuplasItemCantidad: item i x secu(nat) → multiconj(<item, cant>)
(∀d: dicc(item, secu(nat)), i: item, c: cant, s: secu(α), c: cola(α))

s[i] ≡ if i = 0? then prim(s) else fin(s)[i - 1] fi

c[i] ≡ if i = 0? then próximo(c) else desencolar(c)[i - 1] fi

discount(d, i, c) ≡ if ¬def?(i, d) then
    0
```

```
161  else
162      if long(obtener(i, d)) ≤ c then
163          obtener(i, d)[long(obtener(i, d))- 1]
164      else
165          obtener(i, d)[c]
166      fi
167  fi
168
169  ✓hacerMultiConjVentas(d) ≡ if #claves(d) = 0? then ∅ else hacerTuplasItemCantidad(dameUno(claves(d)),
     ✓obtener(dameUno(claves(d)), d)) ∪ hacerMultiConjVentas(borrar(dameUno(claves(d)), d))
170  ✓hacerTuplasItemCantidad(i,s) ≡ if vacía?(s) then ∅ else <i, prim(s)> ∪ hacerTuplasItemCantidad(i, fin(s))
     fi
171
172  algoritmos:
173    ✓iAbrirPuesto(in s: dicc(nat, nat), in m: dicc(nat, nat), in desc: dicc(nat, dicc(nat,nat))) → p: estr
174          p.menú ← vacío() // θ(1).
175          p.stock ← vacío() // θ(1).
176          p.gastoPorPersona ← vacío() // θ(1).
177          p.comprasPorPersona ← vacío() // θ(1).
178          p.comprasSinDescuento ← vacío() // θ(1).
179          // Guardo el menú y el stock.
180          itStock ← crearIt(s) // θ(1)
181          itMenu ← crearIt(m) // θ(1))
182          while(haySiguiente(itMenu) and haySiguiente(itStock)) do // Este ciclo itera I veces con I =
     #claves(s) = #claves(m).
183              definir(p.menú, siguienteClave(itMenu), siguienteSignificado(itMenu)) // θ(log(k)) con k ≤ I.
184              definir(p.stock, siguienteClave(itStock), siguienteSignificado(itStock)) // θ(log(k)) con k ≤
     I.
185              avanzar(itStock) // θ(1)
186              avanzar(itMenu) // θ(1)
187          end while
188          // Defino el vector de descuentos para cada ítem que tiene algún descuento.
189          p.descuentos ← vacío() // θ(1).
190          it ← crearIt(desc) // θ(1)
191          while(haySiguiente(it)) do // Este ciclo itera I' veces con I' = #claves(desc). En el peor caso I
     = I' (todos los ítems tienen algún descuento).
192              // Creo un iterador para el diccionario de descuentos del ítem.
193              diccItem ← siguienteSignificado(it) // θ(1)
194              itDiccItem ← crearIt(diccItem) // θ(1)
195              // Busco el descuento con la mayor cantidad mínima. Ej: Si hay descuento para 2 o + (9%), 4 o
     + (12%) y 6 o + (18%), maxCantMinima sería 6.
```

```
196             maxCantMinima ← 0
197             while(haySiguiente(itDiccItem)) do // Este ciclo itera D veces con D = cantidad de descuentos
    del ítem.
198                 if siguienteClave(itDiccItem) > maxCantMinima // θ(1)
199                     maxCantMinima ← siguienteClave(itDiccItem) // θ(1)
200                 end if
201                 avanzar(itDiccItem) // θ(1)
202             end while
203             // Creo el vector donde se van a guardar los descuentos.
204             vectorPorcDescItem ← vacía() // θ(1)
205             // Ahora guardo los descuentos. Siguiendo el ejemplo anterior debería quedar:
    <0,0,9,9,12,12,18>.
206             descTmp ← 0
207             for int i = 0 to maxCantMinima // Este ciclo itera D' veces con D' = maxCantMinima. D' ≥ D.
208                 if definido?(diccItem, i) // θ(D) (suponiendo que el diccionario de descuentos para un
    ítem recibido es un diccionario con complejidades lineales)
209                     descTmp ← significado(diccItem, i) // θ(D)
210                 end if
211                 agregarAtras(vectorPorcDescItem, descTmp) // θ(1)
212             end for
213             // El peor caso del ciclo de arriba es que haya un descuento para cada 0 ≤ i ≤ maxCantMinima.
214             // Por último defino el par <clave, significado> en el diccionario de p.descuentos
215             definir(p.descuentos, siguienteClave(it), vectorPorcDesItem) // θ(log(j)) + θ(D') ∈ θ(log(j)
    + D') con j ≤ I'.
216             avanzar(it)
217         end while
218       ✓// Complejidad de peor caso (ignoro algunos θ(1)): ∑I k=1 2*θ(log(k)) + ∑I j=1 (∑D i=1 θ(1) +
    ∑D' i=1 2*θ(D) + θ(log(j) + D')) = 2 * I * θ(log(I)) + ∑I j=1 (θ(D) + 2*D'*θ(D) + θ(log(j)) + θ(D')) = 2 * I
    * θ(log(I)) + I * θ(D) + 2*D'*θ(D)*I + I * θ(log(I)) + I * θ(D') ∈ θ(I * log(I) + D'*D*I) ∈ θ(I * (log(I)
    + D' * D))
219
220     ✓ iVender(in per: nat, in i: nat, in cant: nat, in/out p: puesto)
221         // Busco el gasto actual de la persona.
222         if definido?(p.gastoPorPersona, per) // θ(log(A)) con A = cantidad de personas que compraron en
    el puesto.
223             gasto ← significado(p.gastoPorPersona, per) // θ(log(A)) con A = cantidad de personas que
    compraron en el puesto.
224         else
225             gasto ← 0
226         end if
227         // Calculo cuánto acaba de gastar
```

```
228         descuentoAplicable ← porcentajeDescuento(i, cant, p) // θ(porcentajeDescuento) = θ(log(I)) con I
    = cantidad de ítems que vende el puesto.
229         precioMenu ← precio(i,p) // θ(precio) = θ(log(I))
230         nuevoGasto ← ((precioMenu * cant * (100 - descuentoAplicable)) / 100) + gasto //  θ(1)
231         // Cambio el gasto
232         definir(p.gastoPorPersona, per, nuevoGasto) // θ(log(A))
233         // Agrego la cantidad que compró del ítem a las compras de la persona de i. También modifico
    comprasSinDescuento en el caso que la compra haya sido sin descuento.
234         if !definido?(p.comprasPorPersona, per) // θ(log(A))
235             // Caso donde es la primera compra de la persona en el puesto.
236             listaCant ← vacía() // θ(1)
237             iteradorCompra ← agregarAtras(listaCant, cant) // θ(1)
238             definir(p.comprasPorPersona, per, definir(vacío(), i, listaCant)) // θ(log(A))
239             // Si la compra fue sin descuento, defino la persona en comprasSinDescuento con significado
    un diccionario con el ítem como clave y la cola con el iterador como significado.
240             if descuentoAplicable == 0
241                 colaIteradores ← vacía() // θ(1)
242                 encolar(colaIteradores, iteradorCompra) // θ(1)
243                 definir(p.comprasSinDescuento, per, definir(vacío(), i, colaIteradores)) // θ(log(A'))
    con A' = cantidad de personas que compraron sin descuento en el puesto. Peor caso A' = A.
244             end if
245         else
246             // Caso donde la persona ya hizo alguna compra en el puesto.
247             // Debo revisar si ya compró el ítem de esta compra o no.
248             if !definido?(significado(p.comprasPorPersona, per), i) // θ(log(A)) + θ(log(I')) con I'=
    cantidad de ítems comprados por la persona. Peor caso I=I'.
249                 // Caso donde nunca había hecho una compra de i.
250                 listaCant ← vacía() // θ(1)
251                 iteradorCompra ← agregarAtras(listaCant, cant) // θ(1)
252                 definir(significado(p.comprasPorPersona, per), i, listaCant) // θ(log(A)) + θ(log(I')) ∈
    θ(log(A') + log(I'))
253                 // Reviso si fue una compra sin descuento.
254                 if descuentoAplicable == 0
255                     // Debo revisar si es la primera compra sin descuento de la persona o la primera sin
    descuento del ítem.
256                     if !definido?(p.comprasSinDescuento, per) // θ(log(A'))
257                         // Caso en el que nunca hizo una compra sin descuento.
258                         colaIteradores ← vacía() // θ(1)
259                         encolar(colaIteradores, iteradorCompra) // θ(1)
260                         definir(p.comprasSinDescuento, per, definir(vacío(), i, colaIteradores)) //
    θ(log(A'))
```

```
                    else if definido?(p.comprasSinDescuento, per) and !definido?
(significado(p.comprasSinDescuento, per), i) // θ(log(A')) + θ(log(A')) + θ(log(I'')) ∈ θ(log(A') +
log(I'')) con I'' = cantidad de ítems sin descuento comprados por la persona. Peor caso I''= I.
                        // Caso donde ya hizo una compra sin descuento pero no del ítem i.
                        colaIteradores ← vacía() // θ(1)
                        encolar(colaIteradores, iteradorCompra) // θ(1)
                        definir(significado(p.comprasSinDescuento, per), i, colaIteradores) // θ(log(A'))
+ θ(log(I'')) ∈ θ(log(A') + log(I''))
                    else
                        // Caso donde ya hizo una compra sin descuento del ítem i.
                        encolar(significado(significado(p.comprasSinDescuento, per), i), iteradorCompra)
// θ(log(A')) + θ(log(I'')) + θ(1) ∈ θ(log(A') + log(I''))
                    end if
                end if
            else
                // Caso donde ya hizo una compra del ítem i.
                agregarAtras(significado(significado(p.comprasPorPersona, per), i), cant) // θ(log(A)) +
θ(log(I'')) + θ(1) ∈ θ(log(A') + log(I''))
                // Y modifico comprasSinDescuento si la compra fue sin descuento.
                if descuentoAplicable == 0
                    // Debo revisar si es la primera compra sin descuento de la persona o la primera sin
descuento del ítem.
                    if !definido?(p.comprasSinDescuento, per) // θ(log(A'))
                        // Caso en el que nunca hizo una compra sin descuento.
                        colaIteradores ← vacía() // θ(1)
                        encolar(colaIteradores, iteradorCompra) // θ(1)
                        definir(p.comprasSinDescuento, per, definir(vacío(), i, colaIteradores)) //
θ(log(A'))
                    else if definido?(p.comprasSinDescuento, per) and !definido?
(significado(p.comprasSinDescuento, per), i) // θ(log(A')) + θ(log(A')) + θ(log(I'')) ∈ θ(log(A') +
log(I''))
                        // Caso donde ya hizo una compra sin descuento pero no del ítem i.
                        colaIteradores ← vacía() // θ(1)
                        encolar(colaIteradores, iteradorCompra) // θ(1)
                        definir(significado(p.comprasSinDescuento, per), i, colaIteradores)
                    else
                        // Caso donde ya hizo una compra sin descuento del ítem i.
                        encolar(significado(significado(p.comprasSinDescuento, per), i), iteradorCompra)
// θ(log(A')) + θ(log(I'')) + θ(1) ∈ θ(log(A') + log(I''))
                    end if
                end if
```

```
292         end if
293         // Y por último modifico el stock del item
294         definir(p.stock, i, stockItem(i, p) - cant) // θ(log(I))
295         // En el peor caso A = A' y I = I' = I''. Luego la complejidad temporal de peor caso del
    algoritmo es θ(log(A) + log(I)).
296
297   iEstáEnMenú?(in i: nat, in p: estr) → res: bool
298       res ← definido?(p.menú, i) // θ(log(I)) con I = #claves(p.menú)
299
300   iStockItem(in i: nat, in p: estr) → res: nat
301       res ← significado(p.stock, i) // θ(log(I)) con I = #claves(p.menú)
302
303   iPorcentajeDescuento(in i: nat, in cant: nat, in p: estr) → res: nat
304         res ← 0
305         // Reviso si el item tiene algún descuento
306         hayDesc ← definido?(p.descuentos, i) // θ(log(I))
307         if hayDesc
308              // Obtengo el vector con los descuentos.
309              descuentosItem ← significado(p.descuentos, i) // θ(log(I))
310             // Si cant es mayor que la mayor cantidad mínima para la cual aplica un descuento (mayor que
    la long del vector - 1), devuelvo el valor guardado en la última posición del vector.
311             if cant > longitud(descuentosItem) - 1 // θ(1)
312                 res ← descuentosItem[longitud(descuentosItem) - 1] // θ(1).
313             else
314             // Caso contrario, indexo directamente en el vector para buscar el % de descuento.
315                 res ← descuentosItem[cant] // θ(1)
316             end if
317         end if
318     // En el análisis de complejidad I = cantidad de ítems del puesto que tienen descuento. En el peor
    caso van a ser todos los ítems del menú.
319     // El peor caso es cuando el ítem tiene algún descuento y entramos en el primer if (nos hace buscar
    de nuevo en el diccionario para obtener el signficiado). En el segundo if ninguna de las dos ramas nos
    hace realizar muchas más operaciones elementales que en la otra, así que podríamos tomar tanto la
    positiva como la negativa para el análisis.
320   // Luego la complejidad temporal de peor caso de este algoritmo es θ(log(I)).
321
322   iCuántoGastó?(in per: nat, in  p: estr) → res: nat
323       if definido?(p.gastoPorPersona, per) // θ(log(A)) con A = cantidad de personas que compraron en
    el puesto = #claves(p.gastoPorPersona).
324             res ← significado(p.gastoPorPersona, per) // θ(log(A))
325         else
326             res ← 0
```

```
327         end if
328     // Complejidad temporal del peor caso: θ(log(A))
329
330     iPrecio(in i: nat, in p: estr) → res: nat
331         res ← significado(p.menú, i) // θ(log(I)) con I = cantidad de ítems en el menú = #claves(p.menú).
332
333     iCompróItemSinDescuento?(in per: nat, in i: nat, in p: estr) → res: bool
334         res ← true
335         if !definido?(p.comprasSinDescuento, per) // θ(log(A)) con A = cantidad de personas que hicieron
    una compra sin descuento en el puesto = #claves(p.comprasSinDescuento).
336             // Si la persona (DNI) no es una clave en comprasSinDescuento entonces devuelvo false.
337             res ← false
338         else
339             // Caso contrario podría ser que el ítem i no sea una clave en el significado de per en
    comprasSinDescuento o que sí lo sea pero que la cola esté vacía (porque se fueron borrando compras por
    ejemplo). En cualquiera de estos casos también devuelvo false.
340             if !definido?(significado(p.comprasSinDescuento, per), i) or (definido?
    (significado(p.comprasSinDescuento, per), i) and esVacía?(significado(significado(p.comprasSinDescuento,
    per), i))) // θ(log(A) + log(I)) con I = cantidad de ítems sin descuento que compró la persona per.
341                 res ← false
342             end if
343         end if
344     // En el peor caso A = cantidad de personas que hicieron una compra en el puesto =
    #claves(p.gastoPorPersona) e I = cantidad de ítems del puesto = #claves(p.menú)
345     // Luego la complejidad temporal del peor caso es: θ(log(A) + log(I)).
346
347     iOlvidarCompraSinDescuento(in per: nat, in i: nat, in/out p: estr)
348         // Obtengo el valor de una unidad del ítem i en el puesto.
349         precioItem ← precio(i, p) // θ(log(I))
350         // Modifico el gasto de la persona en el puesto.
351         definir(p.gastoPorPersona, per, significado(p.gastoPorPersona, per) - precioItem) // θ(log(A))
352         // Ahora busco el próximo iterador a una compra sin descuento de la lista de compras del ítem i
    de per y disminuyo la cantidad de esa compra en 1.
353         // Me guardo la cantidad de ítems de esa compra.
354         cantItemsComprados ← siguiente(próximo(significado(significado(p.comprasSinDescuento, per), i)))
    // θ(log(A')) + θ(log(I)) + θ(1) + θ(1) ∈ θ(log(A') + log(I))
355         // Elimino la cantidad de ítems de la compra de la lista.
356         eliminarSiguiente(próximo(significado(significado(p.comprasSinDescuento, per), i))) // θ(log(A'))
    + θ(log(I)) + θ(1) + θ(1) ∈ θ(log(A') + log(I))
357         if cantItemsComprados - 1 > 0
358             // Si la compra a borrar fue de más de un elemento, agrego la compra modificada a la lista.
```

```
                ✗ Aca les falta actualizar con el nuevo iterador al nuevo nodo de la lista
359             agregarComoSiguiente(próximo(significado(significado(p.comprasSinDescuento, per), i)),
    cantItemsComprados - 1) // θ(log(A')) + θ(log(I)) + θ(1) + θ(1) ∈ θ(log(A') + log(I))
360         else
361             // Caso contrario no la agrego y desencolo el iterador pues el siguiente ya no necesariamente
    será la cantidad de una compra sin descuento por lo que 'deja de ser útil'.
362             desencolar(significado(significado(p.comprasSinDescuento, per), i)) // θ(log(A') + log(I))
363         end if
364         // Y por último modifico el stock del puesto.
365       ✓ definir(p.stock, i, significado(p.stock, i) + 1) // θ(log(I'))
366     // A = cantidad de personas que compraron en el puesto = #claves(p.gastoPorPersona). A' = cantidad de
    personas que compraron en el puesto sin descuento = #claves(p.comprasSinDescuento). I = cantidad de ítems
    que compro per sin descuento. I' = cantidad de ítems del menú del puesto = #claves(p.menú).
367   ✓ // En el peor caso A = A' e I = I'. Luego la complejidad temporal de peor caso de este algoritmo es:
    θ(log(A) + log(I))
368
369   ✓ iCopiar(in p: estr) → res: estr
370         res.menú ← copiar(p.menú) // θ(I') con I' = #claves(p.menú).
371         res.stock ← copiar(p.stock) // θ(I')
372         res.descuentos ← copiar(p.descuentos) // θ(∑k∈K(copy(significado(k, p.descuentos)))) = θ(∑k∈K l)
    donde K = claves(p.descuentos) y l = long(significado(k, p.descuentos)).
373         res.gastoPorPersona ← copiar(p.gastoPorPersona) // θ(A') con A' = #claves(p.gastoPorPersona)
374         res.comprasPorPersona ← copiar(p.comprasPorPersona) // θ(∑a∈A(copy(significado(a,
    p.comprasPorPersona)))) = θ(∑a∈A(∑i∈I l') con l' = long(significado(significado(a, p.comprasPorPersona),
    i)).
375         res.comprasSinDescuento ← copiar(p.comprasSinDescuento) // θ(∑a∈A(copy(significado(a,
    p.comprasSinDescuento)))) = θ(∑a∈A(∑i∈I l') con l' = tamaño(significado(significado(a,
    p.comprasSinDescuento), i)).
376         // Para el análisis de complejidad tomo en peor caso que A = claves(p.comprasPorPersona) =
    claves(p.comprasSinDescuento) y que todas las personas compraron todos los ítems y alguna vez fue sin
    descuento. I = claves(p.menú).
377         // θ(∑a∈A(∑i∈I l') lo acoto por la longitud de la lista más larga de compras que tenga alguna
    persona = L.
378         // θ(∑a∈A(∑i∈I l') lo acoto por la cola de mayor tamaño de iteradores a compras sin descuento que
    tenga alguna persona = T.
379       ✓ // Luego la complejidad temporal de peor caso de copiar un puesto es: θ(I') + θ(A') + θ(L) + θ(T)
    ∈ θ(I' + A' + L) pues T ≤ L.
380
381 // Lollapatuza
382
383 Interfaz
384     se explica con: Lollapatuza, PuestoDeComida
```

```
385
386     géneros: lolla
387
388     operaciones:
389         INICIAR(in d: dicc(nat, puesto), in pers: conj(nat)) → res: lolla
390         ✓Pre ≡ {¬∅?(pers) ∧ ¬∅?(claves(d)) ∧ (∀n, k: nat)(def?(n, d) ∧ def?(k,d) ∧ n ≠ k ⇒L (∀i:nat)(i ∈
    menu(obtener(n, d)) ∧ i ∈ menu(obtener(n, k))) ⇒L precio(obtener(n, d), i) = precio(obtener(k, d), i)) ∧
    (∀a, k: nat)(a ≠ k ∧ def?(k, d) ⇒L ∅?(ventas(obtener(k,d), a)))}
391         ✓Post ≡ {res =obs crearLolla(d, pers)}
392           Descripción: Genera un nuevo sistema para el festival a partir de las personas participantes y
    los puestos de comida. Cada puesto está vinculado a un ID único.
393           Complejidad: θ(∑p∈PUE(copy(p)) + A * log(A)), con PUE = significados(d) y A = #(pers).
394           Aliasing: res es modificable.
395
396           REGISTRARCOMPRA(in per: nat, in id: nat, in i: nat, in cant: nat, in/out l: lolla)
397         ✓Pre ≡ {l = l₀ ∧ per ∈ personas(l) ∧ def?(id, puestos(l)) ∧L i ∈ menu(obtener(id, puestos(l))) ∧L
    stock(obtener(id,puestos(l)), i) ≥ cant}
398         ✓Post ≡ {l =obs vender(l₀, id, per, i, cant)}
399           Descripción: Registra en el sistema la compra de una cantidad cant del ítem i en el puesto con
    identificador 'id' hecha por la persona per.
400           Complejidad: θ(log(I) + log(P) + log(A)), donde I = #items(l), P = #claves(puestos(l)) y A =
    #personas(l).
401
402           HACKEAR(in per: nat, in i: nat, in/out l: lolla)
403         ✓Pre ≡ {per ∈ personas(l) ∧L (∃id: nat)(def?(id, puestos(l)) ∧L consumioSinPromo?(obtener(id,
    puestos(l)), a, i))}
404         ✓Post ≡ {res =obs hackear(l, per, i)}
405           Descripción: Hackea la compra de una unidad (sin promoción) del ítem i realizada por la persona
    per del puesto de menor ID donde hizo una compra de i sin descuento. Con hackear se quiere decir que se
    elimina todo registro de la transacción en el sistema, tanto para la persona como el puesto. Es decir, la
    persona nunca compró esa unidad del ítem en el puesto y el puesto nunca la vendió.
406           Complejidad: θ(log(A) + log(I) + log(P)), donde I = #items(l), P = #claves(puestos(l)) y A =
    #personas(l).
407
408           GASTOPERSONA(in per: nat, in l: lolla) → res: nat
409         ✓Pre ≡ {per ∈ personas(l)}
410         ✓Post ≡ {res =obs gastoTotal(l, per)}
411           Descripción: Devuelve cuánto dinero gastó un cierto participante en el festival.
412           Complejidad: θ(log(A)), donde A = #personas(l).
413
414           PERSONAQUEMÁSGASTÓ(in l: lolla) → res: nat
```

```
415         Pre ≡ {true}
416         Post ≡ {res =obs masGasto(l)}
417           Descripción: Devuelve la persona (su DNI) que más dinero gastó en el festival. Si hay más de una
    persona que gastó el monto máximo, desempata por la persona de 'menor ID' (DNI).
418           Complejidad: θ(1)
419
420           PUESTOCONMENORSTOCK(in i: nat, in l: lolla) → res: nat
421         Pre ≡ {true}
422         Post ≡ {res =obs menorStock(l, i)}
423           Descripción: Devuelve el ID del puesto que tiene la menor cantidad de stock disponible del ítem
    i. En caso de empate se devuelve el puesto de menor ID.
424           Complejidad: θ(P), con P = #claves(puestos(l)).
425
426           PARTICIPANTES(in l: lolla) → res: conj(nat)
427       Pre ≡ {true}
428       Post ≡ {res =obs personas(l)}
429           Descripción: Devuelve las personas (sus DNIs) en el festival.
430           Complejidad: θ(1)
431           Aliasing: res es una referencia no modificable.
432
433           PUESTOSCONSUSIDS(in l: lolla) → res: dicc(nat, puesto)
434       Pre ≡ {true}
435       Post ≡ {res =obs puestos(l)}
436           Descripción: Devuelve un diccionario tal que los pares clave-significado son <id, puesto>.
437           Complejidad: θ(1)
438           Aliasing: res es una referencia no modificable.
439
440           COMPRÓELITEMSINDESCUENTO?(in per: nat, in i: nat, in l: lolla) → res: bool
441       Pre ≡ {true}
442       Post ≡ {res =obs consumioSinPromoPuestos(per, i, puestos(l))}
443           Descripción: Devuelve true si y solo si la persona per compró el ítem i sin que se le aplique un
    descuento a la compra en alguno de los puestos del festival.
444           Complejidad: θ(P * (log(A) + log(I))), donde I = #items(l), A = #personas(l) y P =
    #claves(puestos(l)).
445
446           LOVENDEALGUNPUESTO?(in i: nat, in l: lolla, out ids: conj(nat)) → res: bool
447         Pre ≡ {true}
448         Post ≡ {(res =obs (∃id: nat)(def?(id, puestos(l)) ∧L i ∈ menu(obtener(id, puestos(l))))) ∧L res ⇒
    L (∀id: nat)(id ∈ ids ⟹ def?(id, puestos(l)) ∧L i ∈ menu(obtener(id, puestos(l))))}
449           Descripción: Devuelve true si y solo si el ítem i está en el menú de alguno de los puestos del
    festival. En el caso que haya alguno, en ids se devuelve el conjunto de todos los id de los puestos que
```

```
lo venden, caso contrario se devuelve un conjunto vacío.
        Complejidad: θ(P * log(I)), donde P = #claves(puestos(l)) e I = #items(l).

        STOCKITEMPUESTO(in i: nat, in k: nat, in l: lolla) → res: nat
        Pre ≡ {def?(k, puestos(l)) ∧Ł i ∈ menu(obtener(k, puestos(l)))}
        Post ≡ {res =obs stock(obtener(k, puestos(l)), i)}
        Descripción: Devuelve el stock disponible del ítem i en el puesto identificado por el ID k.
        Complejidad: θ(log(I)), con I = #items(l).

Fin Interfaz


- Especificación de funciones auxiliares usadas:

TAD Lollapatuza Extendido

    géneros: lolla

    extiende: LOLLAPATUZA

    otras operaciones:
        items: lolla → conj(nat)
        itemsAux: lolla l × dicc(nat, puesto) d → conj(nat) {(∀i:nat)(def?(i, d) ⇒ def?(i, puestos(l)) ∧
Ł obtener(i,d) = obtener(i, puestos(l)))}

    axiomas:
        (∀l: lolla, d: dicc(nat, puesto))
        items(l) ≡ items(l, puestos(l))
        itemsAux(l, d) ≡ if ∅?(claves(d)) then ∅ else menu(obtener(dameUno(claves(d)), d)) ∪ itemsAux(l,
borrar(dameUno(claves(d)), d)) fi

Fin TAD

// item es renombre de nat.
// persona es renombre de nat, representa el DNI de una persona.
// cantidad es renombre de nat.
// puestoID es renombre de nat.
// precio es renombre de nat

representación:
    lolla se representa con estr, donde
```

```
488         estr es tupla⟨
489             puestos: diccLog(puestoID, puesto)
490             personas: conj(persona)
491             laQueMásGastó: colaPriorTupla(gasto: nat, persona)
492             cuántoGastóCadaUna: diccLog(persona, nat)
493             IDpuestosDondeComproSinDescuento: diccLog(persona, diccLog(item, diccLog(puestoID,
    puntero(puesto))))
494         ⟩
495
496 invariante de representación:
497     Rep: estr → bool
498   ✓(∀l: estr) Rep(l) ≡ true ⇔ (1) ∧ (2) ∧L (3) ∧ (4) ∧ (5) ∧ (6)
499
500 // Todos los puestos venden al mismo precio los ítems que comparten en su menú:
501 ✓(1) ≡ (∀id₀, id₁: puestoID)(def?(id₀, l.puestos) ∧ def?(id₁, l.puestos) ⇒L
    vendenMismoPrecio(obtener(id₀, l.puestos), obtener(id₁, l.puestos)))
502 // Coinciden las personas entre los diccionarios y el conjunto de personas. Coincide el tamaño de la cola
    de prioridad con la cantidad de personas:
503 ✓(2) ≡ l.personas = claves(l.cuántoGastóCadaUna) ∧ claves(l.IDpuestosDondeComproSinDescuento) ⊆ l.personas
    ∧ tamaño(l.laQueMásGastó) = #(l.personas)
504 // Todas las personas tienen que estar en la cola de prioridad con su respectivo gasto:
505 ✓(3) ≡ (∀a: persona)(a ∈ l.personas ⇒L (∃! j: nat)(j < #(l.personas) ∧L elementoPos(l.laQueMásGastó, j) =
    <obtener(a, l.cuántoGastóCadaUna), a>))
506 // No existe una persona que tenga mayor gasto que la próxima en la cola de prioridad.
507 ✓(4) ≡ ¬(∃a: persona)(a ∈ l.personas ∧ a ≠ Π₂(próximo(l.laQueMásGastó)) ∧L obtener(a,
    l.cuántoGastóCadaUna) > Π₁(próximo(l.laQueMásGastó)))
508 // Todo ítem que figura como comprado sin descuento por alguna persona debe ser vendido por algún puesto,
    para cada ítem todos los puestos en los que hizo una compra sin descuento deben ser parte del diccionario
    de puestos y deben vender el ítem. La persona debe haber hecho una compra sin descuento en ese puesto y
    la desreferencia del puntero coincide con el puesto en el diccionario de puestos:
509 ✓(5) ≡ (∀a: persona)(def?(a, l.IDpuestosDondeComproSinDescuento) ⇒L (∀i:item)(def?(i, obtener(a,
    l.IDpuestosDondeComproSinDescuento)) ⇒ (∃id:puestoID)(def?(id, l.puestos) ∧L i ∈ menu(obtener(id,
    l.puestos)))))
510 ∧ (∀a: persona)(def?(a, l.IDpuestosDondeComproSinDescuento) ⇒L (∀i:item)(def?(i, itemsQueComproSinDesc)
    ⇒L (∀id:puestoID)(def?(id, puestosDondeComproItemSinDesc) ⇒L def?(id, l.puestos) ∧L i ∈ menu(obtener(id,
    l.puestos)) ∧L consumioSinPromo?(obtener(id, l.puestos), a, i) ∧ obtener(id, l.puestos) = *obtener(id,
    puestosDondeComproItemSinDesc)))
511 ✓// En (4):
512 // itemsQueComproSinDesc ≡ obtener(a, l.IDpuestosDondeComproSinDescuento)
```

```
// puestosDondeComproItemSinDesc ≡ obtener(i, obtener(a, l.IDpuestosDondeComproSinDescuento))
// El gasto de una persona debe coincidir con la suma de lo que gastó en cada puesto:
(6) ≡ (∀a: persona)(a ∈ l.personas ⇒ʟ obtener(a, l.cuántoGastóCadaUna) = sumaGastosPuestos(l.puestos, a))


función de abstracción:
  Abs: estr e → lolla {Rep(e)}
    (∀e: estr) Abs(e) =obs l: lolla | puestos(l) =obs e.puestos ∧ personas(l) =obs e.personas

Auxiliares usados en el invariante de representación:
        tamaño: colaPrior(α) → nat
        colaASec: colaPrior(α) → secu(α)
        borrarPos: colaPrior(α) c × nat i → colaPrior(α) {i < tamaño(c)}
        borrarPosAux: colaPrior(α) c × nat i × secu(α) s → colaPrior(α) {i < tamaño(c) ∧ (∀a:α)(está?(a, s) ⟹ está?(a, colaASec(c)))}
        elementoPos: colaPrior(α) c × nat i → α {i < tamaño(c)}
        sumaGastosPuestos: dicc(puestoID, puesto) × persona a → nat

        (∀c: colaPrior(α), i: nat, s: secu(α), d: dicc(puestoID, puesto), a: persona)
        tamaño(c) ≡ long(colaASec(c))
        colaASec(c) ≡ if vacía?(c) then <> else próximo(c) • colaASec(desencolar(c)) fi
        borrarPos(c, i) ≡ borrarPosAux(c, i, quitar(colaASec(c), i))
        borrarPosAux(c, i, s) ≡ if long(s) - 1 = 0? then encolar(prim(s), vacía) else encolar(prim(s), borrarPosAux(c, i, fin(s))) fi
        elementoPos(c, i) ≡ colaASec(c)[i]
        sumaGastosPuestos(d, a) ≡ if ∅?(claves(d)) then 0 else gastosDe(obtener(dameUno(claves(d)), d), a) + sumaGastosPuestos(borrar(dameUno(claves(d)), d), a) fi

algoritmos:
     iIniciar(in pues: dicc(nat, puesto), in pers: conj(nat)) → l: estr
          l.puestos ← vacío() // θ(1)
          l.personas ← vacío() // θ(1)
          l.cuántoGastóCadaUna ← vacío() // θ(1)
          l.IDpuestosDondeComproSinDescuento ← vacío() // θ(1)
          l.laQueMásGastó ← vacía() // θ(1)
          itPuestos ← crearIt(pues) // θ(1)
          itPersonas ← crearIt(pers) // θ(1)
          // Defino en l.puestos cada puesto que habrá en el evento a partir del diccionario pues.
          while(haySiguiente(itPuestos)) do
              definir(l.puestos, siguienteClave(itPuestos), siguienteSignificado(itPuestos))
```

```
550                 avanzar(itPuestos)
551             end while
552             // Agrego a l.personas las personas que habrá en el evento dadas por el conj(nat) pers.
    También las encolo en laQueMásGastó con gasto 0 a todas y las defino en cuántoGastóCadaUna y
    IDpuestosDondeComproSinDescuento.
553             while(haySiguiente(itPersonas)) do // θ(1)
554                 agregarRapido(l.personas, siguiente(itPersonas)) // θ(1)
555                 encolar(l.laQueMásGastó, <0, siguiente(itPersonas)>) // θ(1) en la primera iteración,
    luego θ(log(i)). 1 ≤ i ≤ #(pers) = A.
556                 definir(l.cuántoGastóCadaUna, siguiente(itPersonas), 0) // θ(1) en la primera iteración,
    luego θ(log(i)).
557                 definir(l.IDpuestosDondeComproSinDescuento, siguiente(itPersonas), vacío()) // θ(1) en la
    primera iteración, luego θ(log(i)).
558                 avanzar(itPersonas)
559             end while
560         // La complejidad del primer while es: ∑p∈P(copy(p)) con P = significados(pues).
561         // La complejidad del segundo while es: ∑^A_{i=1} 3*log(i) = 3 * ∑^A_{i=1} log(i) = log(∏^A_{i=0} i) =
    log(A!) ~ A * log(A) ∈ θ(A * log(A)) usando la aproximación de Stirling. A = #(pers).
562         // Luego la complejidad temporal es: θ(∑p∈P(copy(p)) + A * log(A)).
563
564
565     iRegistrarCompra(in per: nat, in id: nat, in i: nat, in cant: nat, in/out l: estr)
566             // Modifico el puesto donde se hizo la venta
567             Vender(per, i, cant, significado(p.puestos, id))) // θ(log(P)) + θ(Vender) ∈ θ(log(P) +
    log(I) + log(A))
568             // Actualizo cuánto gastó la persona per
569             descuentoAplicable ← porcentajeDescuento(i, cant, significado(p.puestos, id)) //
    θ(porcentajeDescuento) + θ(log(P)) ∈ θ(log(I) + log(P))
570             precioMenu ← precio(i, puesto) // θ(precio) = θ(log(I))
571             totalCompra ← ((precioMenu * cant * (100 - descuentoAplicable)) / 100) // θ(1)
572             nuevoGasto ← significado(l.cuántoGastóCadaUna, per) + totalCompra // θ(log(A))
573             // Actualizo la información de la persona que más gastó y del gasto.
574             posActual ← posiciónEnCola(<significado(l.cuántoGastóCadaUna, per), per>, l.laQueMásGastó) //
    θ(log(A)) + θ(posiciónEnCola) ∈ θ(log(A)) + θ(log(A)) ∈ θ(log(A))
575             eliminar(l.laQueMásGastó, posActual) // θ(eliminar) ∈ θ(log(A))
576             definir(l.cuántoGastóCadaUna, per, nuevoGasto) // θ(log(A))
577             encolar(l.laQueMásGastó, <nuevoGasto, per>) // θ(encolar) ∈ θ(log(A))
578             // Si la compra es sin descuento, actualizo IDpuestosDondeComproSinDescuento.
579             if descuentoAplicable == 0
580                 if !definido?(significado(l.IDpuestosDondeComproSinDescuento, per), i) // θ(log(A)) +
    θ(log(I)) ∈ θ(log(A) + log(I'))
```

```
581                     // Caso en el que es la primera compra sin descuento del ítem i.
582                     puntero(puesto) p ← &significado(p.puestos, id) // θ(1)
583                     definir(significado(l.IDpuestosDondeComproSinDescuento, per), i, definir(vacío(), id,
    p)) // θ(log(A)) + θ(log(I')) + θ(1) + θ(1) ∈ θ(log(A) + log(I')).
584                 else
585                     // Caso donde la persona ya hizo una compra sin descuento del ítem i.
586                     if !definido?(significado(significado(l.IDpuestosDondeComproSinDescuento, per), i),
    id) // θ(log(A)) + θ(log(I')) + θ(log(P')) ∈ θ(log(A) + log(I') + log(P'))
587                         // En el caso que no había sido realizada una compra del ítem i sin descuento en
    este puesto, defino el id y el puntero en el diccionario.
588                         puntero(puesto) p ← &significado(p.puestos, id) // θ(1)
589                         definir(significado(significado(l.IDpuestosDondeComproSinDescuento, per), i), id,
    p) // θ(log(A)) + θ(log(I')) + θ(log(P')) ∈ θ(log(A) + log(I') + log(P'))
590                     end if
591                 end if
592             end if
593             // A = cantidad de personas, P = cantidad de puestos, I = cantidad de ítems del puesto (en el
    peor caso I es el total de ítems del festival), I' = cantidad de ítems que compró la persona sin
    descuento (peor caso I = I'), P' = cantidad de puestos donde la persona compró el ítem sin descuento
    (peor caso P' = P).
594             // Complejidad temporal de peor caso: θ(log(I) + log(P) + log(A))
595
596         iGastoPersona(in per: nat, in l: estr) → res: nat
597             res ← significado(l.cuántoGastóCadaUna, per) // θ(log(A)) con A =
    #claves(l.cuántoGastóCadaUna) = #l.personas
598
599         iParticipantes(in l: estr) → res: conj(nat)
600             res ← l.personas
601             // Complejidad: θ(1), res se pasa por referencia (no modificable).
602
603         iPuestosConSusIDS(in l: estr) → res: dicc(nat, puesto)
604             res ← l.puestos
605             // Complejidad: θ(1), res se pasa por referencia (no modificable).
606
607         iPuestoConMenorStock(in i: nat, in l: estr) → res: nat
608             it ← crearIt(l.puestos) // θ(1)
609             // Guardo el ID de los puestos con el item en su menú y el stock del ítem.
610             puestosItem ← vacío() // diccionario lineal // θ(1)
611             while(haySiguiente(it)) do
612                 if estáEnMenú?(i, proxPuesto)
```

```
613                     definirRapido(puestosItem,  siguienteClave(it), stockItem(i,
    siguienteSignificado(it)))
614                 end if
615                 it ← avanzar(it)
616             end while
617             it2 ← crearIt(puestosItem)
618             if !haySiguiente(it2)
619                 // Si no hay siguiente quiere decir que el ítem no era vendido por ningún puesto, por lo
    cual devuelvo el menor ID de todos.
620                 itTmp ← crearIt(l.puestos)
621                 // El iterador de diccionario logarítmico recorre las claves en orden de menor a mayor,
    así que siguiente(itTmp) será el menor ID.
622                 res ← siguienteClave(itTmp)
623             else
624                 // Caso contrario hay aunque sea un puesto que vende el ítem. Recorro el diccionario que
    relaciona el ID de los puestos con el ítem en su menú con el stock del ítem.
625                 res ← siguienteClave(it2)
626                 stockRes ← siguienteSignificado(it2)
627                 while(haySiguiente(it2)) do
628                     // Actualizo res si el stock es menor
629                     if siguienteSignificado(it2) < stockRes
630                         stockRes ← siguienteSignificado(it2)
631                         res ← siguienteClave(it2)
632                     // Actualizo res si el stock es igual pero el ID es menor
633                     else if siguienteSignificado(it2) == stockRes and siguienteClave(it2) < res
634                         stockRes ← siguienteSignificado(it2)
635                         res ← siguienteClave(it2)
636                     end if
637                 end while
638             end if
639         // Complejidad: El peor caso es cuando todos los puestos venden el ítem i. El primer while itera
    P veces, con P = #claves(l.puestos), y cada iteración cuesta la definición rápida en el diccionario
    puestosItem: ∑k∈K(θ(1) + θ(1)) = 2 * θ(1) * ∑k∈K(1) ∈ θ(#K) ∈ θ(P), K = claves(l.puestos).
640         // Y como en peor caso todos los puestos venden el ítem i, entonces el segundo while (del else)
    también itera P veces, y cada iteración lleva tiempo constante pues son comparaciones y asignaciones
    básicas (de naturales), por lo que el segundo while también es θ(P).
641         // Por lo tanto la complejidad temporal de peor caso de este algoritmo es: θ(P) + θ(P) ∈ θ(P).
642
643         iPersonaQueMásGastó(in l: estr) → res: nat
644             res ← próximo(l.laQueMásGastó).second
645         // Complejidad: θ(1) + θ(1) ∈ θ(1)
```

```
646
647         iHackear(in per: nat, in i: nat, in/out l: estr)
648             // Creo un iterador para el diccionario de puestos donde per compró el ítem i sin descuentos.
649             it ← crearIt(significado(significado(l.IDpuestosDondeComproSinDescuento), per), i) //
    θ(log(A) + log(I))
650             // El iterador de diccLog recorre las claves en orden de menor a mayor, por lo cual el menor
    ID será siguienteClave de it y siguienteSignificado será el puntero al puesto a hackear.
651             // Modifico la información de la persona que más gastó y del gasto de la persona.
652             precioItem ← precio(i, *siguienteSignificado(it)) // θ(log(P)) + θ(precio) ∈ θ(log(P) +
    log(I'))
653             nuevoGasto ← significado(l.gastoPorPersona, per) - precioItem // θ(log(A))
654             posCola ← posiciónEnCola(<nuevoGasto + precioItem, per>, l.laQueMásGastó) //
    θ(posiciónEnCola) ∈ θ(log(A))
655             eliminar(l.laQueMásGastó, posCola) // θ(eliminar) ∈ θ(log(A))
656             encolar(l.laQueMásGastó, <nuevoGasto, per>) // θ(encolar) ∈ θ(log(A))
657             definir(l.gastoPorPersona, per, nuevoGasto) // θ(log(A))
658             // Borro el registro de la compra de la unidad del ítem de la información del puesto.
659             olvidarCompraSinDescuento(per, i, *siguienteSignificado(it))
660             // Reviso si el puesto deja de ser hackeable para esta persona e ítem. En tal caso elimino la
    clave del diccionario de puestos donde la persona compró el ítem sin descuento.
661             if !compróItemSinDescuento(per, i, *siguienteSignificado(it)) // Función del módulo puesto de
    comida // θ(log(A') + log(I'))
662                 borrar(significado(significado(l.IDpuestosDondeComproSinDescuento, per), i),
    siguienteClave(it)) // θ(log(A) + log(I) + log(P'))
663             end if
664         // A = #l.personas. P = #claves(l.puestos). I = cantidad de ítems sin descuento comprados por
    per. I' = cantidad de ítems que vende el puesto identificado por IDpuesto. A' = cantidad de personas que
    hicieron una compra en el puesto identificado por IDpuesto. P' = #puestos donde per hizo una compra sin
    descuento.
665         // En peor caso A = A', I = I'= cantidad de ítems que se venden en el festival y P' = P. Luego el
    peor caso es cuando el puesto deja de ser hackeable y la complejidad temporal es: θ(log(A) + log(I) +
    log(P)).
666
667     iCompróElItemSinDescuento?(in per: nat, in i: nat, in l: estr) → res: bool
668             res ← false
669             itPuestos ← crearIt(l.puestos) // θ(1)
670             while(haySiguiente(itPuestos)) do
671                 if compróItemSinDescuento?(per, i, siguienteSignificado(itPuestos)) //
    θ(compróItemSinDescuento?) ∈ θ(log(A) + log(I))
672                     res ← true
673                     break
```

```
674                 end if
675                 avanzar(itPuestos) // θ(1)
676             end while
677         // A = cantidad de personas que hicieron una compra en el puesto. I = cantidad de ítems que vende el puesto.
678         // Complejidad temporal de peor caso (cuando no compró el ítem sin descuento): La complejidad va a estar dada por lo que cueste cada iteración del while, que en este caso es θ(log(A) + log(I)) y en peor caso I = #ítems que se venden en el festival y A = #l.personas. Luego como el while en peor caso itera P veces con P = #claves(l.puestos),
679         // entonces la complejidad temporal de peor caso de este algoritmo es θ(P * (log(A) + log(I))). ✓
680
681       ✓iLoVendeAlgunPuesto?(in i: nat, in l: lolla, out ids: conj(nat)) → res: bool
682             ids ← vacío() // conjunto lineal //  θ(1)
683             itPuestos ← crearIt(l.puestos) // θ(1)
684             while(haySiguiente(itPuestos)) do
685                 if estáEnMenú?(i, siguienteSignificado(itPuestos)) // θ(estáEnMenú) ∈ θ(log(I))
686                     res ← true
687                     agregarRapido(ids, siguienteClave(itPuestos)) // θ(1) + θ(1) ∈ θ(1)
688                 end if
689                 avanzar(itPuestos) // θ(1)
690             end while   ✗ Falta devolver res <- false
691         // Complejidad: La complejidad va a estar dada por lo que cueste cada iteración del while que itera P = #claves(l.puestos), es decir θ(log(I)) donde en peor caso I = #ítems que se venden en el festival. Por lo que la complejidad temporal de peor caso de este algoritmo es θ(P * log(I)).
692
693       ✓iStockItemPuesto(in i: nat, in id: nat, in l: estr) → res: nat
694             res ← stockItem(i, significado(l.puestos, id))
695         ✓// Complejidad: θ(stockItem) ∈ θ(log(I)), donde en peor caso I = #ítems que se venden en el festival.
696
697 // Módulo cola de prioridad de tuplas.
698 Interfaz
699     parámetros formales:
700         géneros: α, β
701   ✓// Con esto queremos decir que la prioridad al momento de encolar será por el que tenga mayor primera coordenada, y en caso de empate en la primera, por el de menor segunda coordenada. No estábamos seguros de dónde iba.
702         funciones:
703             •<•(in t₀: tupla(α, β), in t₁: tupla(α, β)) → res: bool
704           ✓Pre ≡ {true}
705           ✓Post ≡ {(res =obs Π₁(t₀) < Π₁(t₁)) ∨ (res =obs (Π₁(t₀) = Π₁(t₁) ∧ Π₂(t₀) < Π₂(t₁))}
```

```
706             Descripción: Función de comparación de tuplas.
707             Complejidad: O(1)
708             Aliasing: No presenta aspectos de aliasing
709
710   ✓se explica con: Cola de Prioridad(α)
711
712     géneros: colaPriorTupla(α, β)
713
714     operaciones:
715         VACÍA() → res: colaPriorTupla(α, β)
716       ✓ Pre ≡ {true}
717       ✓ Post ≡ {res =obs vacía}
718         Descripción: Genera una cola de prioridad de tuplas vacía.
719         Complejidad: O(1)
720
721         VACÍA?(in c: colaPriorTupla(α, β)) → res: bool
722       ✓ Pre ≡ {true}
723       ✓ Post ≡ {res =obs vacía?()}
724         Descripción: Devuelve true si y solo si la cola de prioridad es vacía.
725         Complejidad: O(1)
726
727         PRÓXIMO(in c: colaPriorTupla(α, β)) → res: tupla(α, β)
728       ✓ Pre ≡ {¬vacía?(c)}
729       ✓ Post ≡ {res =obs próximo(c)}
730         Descripción: Devuelve el siguiente elemento de la cola de prioridad.
731         Complejidad: O(1)
732         Aliasing: res es no modificable.
733
734         ENCOLAR(in/out c: colaPriorTupla(α, β), in t: tupla(α, β))
735       ✓ Pre ≡ {¬vacía?(c) ∧ c =obs c₀}
736       ✓ Post ≡ {c =obs encolar(t, c₀)}
737         Descripción: Encola un elemento en c según su prioridad (la prioridad la tiene el de mayor
    primera coordenada y en caso de empate se desampata por el de menor segunda coordenada).
738         Complejidad: O(log(C)), donde C = tamaño(c₀).
739         Aliasing: t se encola por copia.
740
741         DESENCOLAR(in/out c: colaPriorTupla(α, β))
742       ✓ Pre ≡ {¬vacía?(c) ∧ c =obs c₀}
743       ✓ Post ≡ {c =obs desencolar(c₀)}
744         Descripción: Desencola el próximo elemento de la cola c.
745         Complejidad: O(log(C)), donde C = tamaño(c₀).
746
```

```
747         ELIMINAR(in/out c: colaPriorTupla(α, β), in i: nat)
748       ✓ Pre ≡ {i < tamaño(c) ∧ c =obs c₀}
749       ✓ Post ≡ {c =obs borrarPos(c₀, i)}
750         Descripción: Elimina el elemento en la posición i de la cola.
751         Complejidad: θ(log(C)), donde C = tamaño(c₀).
752
753         ESTÁENLACOLA?(in a: tupla(α, β), in c: colaPriorTupla(α, β)) → res: bool
754       ✓ Pre ≡ {true}
755       ✓ Post ≡ {res =obs (∃i: nat)(i < tamaño(c) ∧L elementoPos(c, i) = a)}
756         Descripción: Devuelve true si y sólo si a está en la cola.
757         Complejidad: θ(log(C)), donde C = tamaño(c).
758
759         POSICIÓNENCOLA(in a: tupla(α, β), in c: colaPriorTupla(α, β)) → res: nat
760       ✓ Pre ≡ {(∃i: nat)(i < tamaño(c) ∧L elementoPos(c, i) = a)}
761       ✓ Post ≡ {elementoPos(c, res) =obs a}
762         Descripción: Devuelve en que posición de la cola está a.
763         Complejidad: θ(log(C)), donde C = tamaño(c).
764
765
766 Fin Interfaz
767
768 - Especificación de las operaciones auxiliares utilizadas:
769
770 TAD Cola de Prioridad Extendida(α)
771
772     extiende: COLA DE PRIORIDAD(α)
773
774   ✓ otras operaciones:
775         tamaño: colaPrior(α) → nat
776         colaASec: colaPrior(α) → secu(α)
777         borrarPos: colaPrior(α) c × nat i → colaPrior(α) {i < tamaño(c)}
778         borrarPosAux: colaPrior(α) c × nat i × secu(α) s → colaPrior(α) {i < tamaño(c) ∧ (∀a:α)(está?(a,
    s) ⟹ está?(a, colaASec(c)))}
779         elementoPos: colaPrior(α) c × nat i → α {i < tamaño(c)}
780
781   ✓ axiomas:
782         (∀c: colaPrior(α), i: nat, s: secu(α))
783         tamaño(c) ≡ long(colaASec(c))
784         colaASec(c) ≡ if vacía?(c) then <> else próximo(c) • colaASec(desencolar(c)) fi
785         borrarPos(c, i) ≡ borrarPosAux(c, i, quitar(colaASec(c), i))
```

```
786         borrarPosAux(c, i, s) ≡ if long(s) - 1 = 0? then encolar(prim(s), vacía) else encolar(prim(s),
    borrarPosAux(c, i, fin(s))) fi
787         elementoPos(c, i) ≡ colaASec(c)[i]
788
789 Fin TAD
790
791 ✓TAD Secuencia Extendida(α)
792
793     extiende: SECUENCIA(α)
794
795   ✓otras operaciones:
796         quitar: secu(α) s × nat i → secu(α) {i < long(s)}
797         •[•]: secu(α) s × nat i → secu(α) {i < long(s)}
798
799   ✓axiomas:
800         (∀s: secu(α), i: nat)
801         quitar(s, i) ≡ if i = 0? then fin(s) else prim(s) • quitar(fin(s), i - 1) fi
802         s[i] ≡ if i = 0? then prim(s) else fin(s)[i - 1] fi
803
804 Fin TAD
805
806 representación:
807   ✓colaPriorTupla(α, β) se representa con estr, donde
808         estr es tupla⟨
809             cola: vector(tupla(α, β))
810             posicionPorElemento: diccLog(tupla(α, β), nat)
811           ⟩
812
813 ✓invariante de representación:
814   ✓Rep: estr → bool
815     (∀e: estr) Rep(e) ≡ long(e.cola) = #claves(e.posicionPorElemento) ∧L (¬vacía?(e.cola) ⇒L (∀i: nat)(i
    < long(e.cola) ⇒L ((2*i+1 < long(e.cola) ⇒L (Π₁(e.cola[i]) < Π₁(e.cola[2*i+1]) ∨ ( Π₁(e.cola[i]) =
    Π₁(e.cola[2*i+1]) ∧ Π₂(e.cola[i]) < Π₂(e.cola[2*i+1]))) ∧ (2*i+2 < long(e.cola) ⇒L (Π₁(e.cola[i]) <
    Π₁(e.cola[2*i+2]) ∨ ( Π₁(e.cola[i]) = Π₁(e.cola[2*i+2]) ∧ Π₂(e.cola[i]) < Π₂(e.cola[2*i+2])))))) ∧
    (∀i:nat)(i < long(e.cola) ⇒L def?(e.cola[i], e.posicionPorElemento) ∧L obtener(e.cola[i],
    e.posicionPorElemento) = i)) ∧ (∀t:tupla(α, β))(def?(t, e.posicionPorElemento) ⇒L obtener(t,
    e.posicionPorElemento) < long(e.cola) ∧L e.cola[obtener(t, e.posicionPorElemento)] = t)
816
817 ✓// Para la función de abstracción suponemos que encolar del TAD Cola de Prioridad encola los elementos de
    la manera que nosotros definimos.
```

```
función de abstracción:
  ✓ Abs: estr e → colaPior(α) {Rep(e)}
      (∀e: estr) Abs(e) =obs c: colaPrior(α) | vacía?(c) =obs vacía?(e.cola) ∧L (¬vacía?(c) ⇒L próximo(c)
=obs prim(e.cola) ∧ desencolar(c) =obs secuAColaPrior(fin(e.cola)))

//✓Auxiliares
    •[•] : secu(α) s x nat n → α {n < long(s)}
   ✓secuAColaPrior: secu(α) → colaPrior(α)
    (∀s: secu(α), n: nat)
    s[n] ≡ if n = 0? then prim(s) else fin(s)[n-1] fi
  ✓ secuAColaPrior(s) ≡ if vacía?(s) then
        vacía
    else
        encolar(prim(s), secuAColaPrior(fin(s)))
    fi

algoritmos:
   ✓iVacía() → res: estr
        res.cola ← vacía() // θ(1)
        res.posicionPorElemento ← vacío() // θ(1)
    // Complejidad: θ(1)

  ✓ iVacía?(in c: estr) → res: bool
        res ← esVacío?(c.cola) and #claves(c.posicionPorElemento) == 0
    // Complejidad: θ(1)

    // Pre: la cola no es vacía
  ✓ iPróximo(in c: estr) → res: tupla(α, β)
        res ← c.cola[0]
    // Complejidad: θ(1)

  ✓ iEncolar(in/out c: estr, in a: tupla(α, β))
        // Agrego el elemento a lo último
        agregarAtras(c.cola, a)
        i ← longitud(c.cola) - 1
        // Lo defino en el diccionario
        definir(c.posicionPorElemento, a, longitud(c.cola) - 1)
        // Lo "subo" para asegurar que se mantiene la propiedad de heap.
        while i > 0 and (c.cola[i].first > c.cola[((i-1) / 2)].first or (c.cola[i].first == c.cola[((i-1)
/ 2)].first and c.cola[i].second < c.cola[((i-1) / 2)].second)) do // En (i-1) / 2 tomo la parte entera
de la división.
```

```
856             tmp ← c.cola[i]
857             c.cola[i] ← c.cola[((i-1) / 2)]
858             c.cola[((i-1) / 2)] ← tmp
859             // Actualizo las posiciones del diccionario
860             definir(c.posicionPorElemento, c.cola[i], (i-1) / 2)
861             definir(c.posicionPorElemento, c.cola[(i-1) / 2], i)
862             i ← (i-1) / 2
863         end while
864   ✓// Encolar un elemento en un heap en peor caso es θ(log(n)) con n la cantidad de elementos en el
    heap. Es decir, el caso donde el elemento que se encola debe ser la raíz del heap, por lo cual se van a
    realizar una cantidad de swapeos proporcional a la altura del heap, es decir log(n). Luego las
    definiciones en el diccionario también pertenecen a θ(log(n)).
865     // Por lo tanto la complejidad temporal de peor caso es θ(log(n)) con n = longitud(c.cola).
866
867 ✗ ✗iDesencolar(in/out c: estr)
868         eliminar(c, 0)
869     // Complejidad: θ(eliminar) ∈ θ(log(n)) con n = longitud(c.cola). Esta no es la complejidad de eliminar.
870
871   ✓iEstáEnLaCola?(in a: tupla(α, β), in c: estr) → res: bool
872         res ← definido?(a, c.posicionPorElemento)
873     // Complejidad: θ(log(n)) con n = longitud(c.cola) = #claves(c.posicionPorElemento).
874
875   ✓iPosiciónEnCola(in a: tupla(α, β), in c: estr) → res: nat
876         res ← significado(c.posicionPorElemento, a)
877     // Complejidad: θ(log(n)) con n = longitud(c.cola) = #claves(c.posicionPorElemento).
878
879 ✗ ✗iEliminar(in/out c: estr, in i: nat)
880         if i == longitud(c.cola) - 1 Entiendo que buscan capturar el caso en que tiene un solo elemento,
881             comienzo(c.cola)          pero esta condicion no dice eso.
882             c.posicionPorElemento ← vacío() ✗ Porque asignan Vacio()?
883         else
884             c.cola[i] ← c.cola[longitud(c.cola) - 1]
885             borrar(c.posicionPorElemento, c.cola[longitud(c.cola) - 1])
886             comienzo(c.cola)
887             definir(c.posicionPorElemento, c.cola[i], longitud(c.cola) - 1)
888           ✓if i > 0 and c.cola[i].first > c.cola[(i-1) / 2].first
889                   while i > 0 and (c.cola[i].first > c.cola[((i-1) / 2)].first or (c.cola[i].first ==
    c.cola[((i-1) / 2)].first and c.cola[i].second < c.cola[((i-1) / 2)].second)) do
890                     tmp ← c.cola[i]
891                     c.cola[i] ← c.cola[((i-1) / 2)]
892                     c.cola[((i-1) / 2)] ← tmp
```

```
893                     // Actualizo las posiciones del diccionario
894                     definir(c.posicionPorElemento, c.cola[i], (i-1) / 2)
895                     definir(c.posicionPorElemento, c.cola[(i-1) / 2], i)
896                     i ← (i-1) / 2
897                 end while
898             else
899                 // Bajar(p): while p no es hoja and prioridad(p) < prioridad(algun hijo) do intercambiar
    p con el hijo de mayor prioridad
900                 while !(2*i + 1 >= Longitud(c.cola) and 2*i + 2 >= Longitud(c.cola)) and (2*i+1 <
    Longitud(c.cola) and (c[i].first < c[2*i+1].first or c[i].first == c[2*i+1].first and c[i].second <
    c[2*i+1].second) or (2*i+2 < Longitud(c.cola) and (c[i].first < c[2*i+2].first or c[i].first ==
    c[2*i+2].first and c[i].second < c[2*i+2].second)) do
901                     // Intercambio con el hijo de mayor prioridad
902                     // Caso tiene dos hijos. Me fijo cuál tiene mayor prioridad por el primer elemento de la
    tupla, o el segundo si empatan en el primero.
903                 if 2*i+1 < Longitud(c.cola) and 2*i+2 < Longitud(c.cola)
904                     if c.cola[2*i+1].first > c.cola[2*i+2].first
905                         tmp ← c.cola[2*i+1]
906                         c.cola[2*i+1] ← c.cola[i]
907                         c.cola[i] ← tmp
908                         // Actualizo las posiciones del diccionario
909                         definir(c.posicionPorElemento, c.cola[i] , 2*i + 1)
910                         definir(c.posicionPorElemento, c.cola[2*i+1], i)
911                         i ← 2 * i + 1
912                     else
913                     // Me fijo si empatan en la primera coordenada
914                         if c.cola[2*i+1].first == c.cola[2*i+2].first
915                         // Desempato por la segunda
916                             if c.cola[2*i+1].second < c.cola[2*i+2].second
917                                 tmp ← c.cola[2*i+1]
918                                 c.cola[2*i+1] ← c.cola[i]
919                                 c.cola[i] ← tmp
920                                 // Actualizo las posiciones del diccionario
921                                 definir(c.posicionPorElemento, c.cola[i] , 2*i + 1)
922                                 definir(c.posicionPorElemento, c.cola[2*i+1], i)
923                                 i ← 2 * i + 1
924                             else
925                                 tmp ← c.cola[2*i+2]
926                                 c.cola[2*(i+2)] ← c.cola[i]
927                                 c.cola[i] ← tmp
928                                  // Actualizo las posiciones del diccionario
929                                 definir(c.posicionPorElemento, c.cola[i] , 2*i + 2)
```

```
930                             definir(c.posicionPorElemento, c.cola[2*i+2], i)
931                             i ← 2 * i + 2
932                         end if
933                     else
934                         tmp ← c.cola[2*i+2]
935                         c.cola[2*(i+2)] ← c.cola[i]
936                         c.cola[i] ← tmp
937                          // Actualizo las posiciones del diccionario
938                         definir(c.posicionPorElemento, c.cola[i] , 2*i + 2)
939                         definir(c.posicionPorElemento, c.cola[2*i+2], i)
940                         i ← 2 * i + 2
941                 end if
942             end if
943         ✓ // Caso solo hijo izquierdo
944             else if 2*i + 1 < Longitud(c.cola)
945                 tmp ← c.cola[2*i+1]
946                 c.cola[2*i+1] ← c.cola[i]
947                 c.cola[i] ← tmp
948                 // Actualizo las posiciones del diccionario
949                 definir(c.posicionPorElemento, c.cola[i] , 2*i + 1)
950                 definir(c.posicionPorElemento, c.cola[2*i+1], i)
951                 i ← 2 * i + 1
952         ✓ // Caso solo hijo derecho
953             else
954                 tmp ← c.cola[2*i+2]
955                 c.cola[2*i+2] ← c.cola[i]
956                 c.cola[i] ← tmp
957                  // Actualizo las posiciones del diccionario
958                 definir(c.posicionPorElemento, c.cola[i] , 2*i + 2)
959                 definir(c.posicionPorElemento, c.cola[2*i+2], i)
960                 i ← 2 * i + 2
961             end if
962         end if
963     // Eliminar un elemento en un heap dada su posición en peor caso es θ(log(n)) con n la cantidad de
    elementos en el heap. Es decir el caso donde se elimina la raíz, entonces se intercambia con el último y
    luego se realizan los swapeos necesarios para mantener la propiedad de heap, que en peor caso son
    proporcionales a la altura del heap, es decir log(n). Luego las definiciones en el diccionario también
    pertenecen a θ(log(n)).
964     // Por lo tanto la complejidad temporal de peor caso es θ(log(n)) con n = longitud(c.cola).
```

✗ Esta no es la complejidad del algoritmo. Iteran log(n) y luego dentro del while defininen, lo que cuesta log(n) tambien.
Por lo tanto la complejidad del algoritmo es log(n)^2