

Algoritmos y Estructuras de Datos 2.

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

TP 2 Diseño: Lollapatuza.

Integrante	LU	Correo electrónico
Nicolás Pachón Pintos	301/20	pintosn1511@gmail.com
Kevin Alexander Perez Marzo	770/16	stixerks@gmail.com
Javier Mareque	112/22	javiermaxmareque@gmail.com
Pablo Villavicencio	799/07	pablovillavicencio87@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega	Santiago	Aprobado
Segunda entrega		

Suman puntos extras por presentación y comentarios explicativos.

1. Modulo Puestos De Comida

Interfaz

se explica con: PUESTO DE COMIDA, PERSONA, ITEM, CANT

géneros: puesto

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 $\equiv \{res = crearPuesto(p, s, d)\}$ ✓

Complejidad: $O(I^2 + I \cdot cant)$

Descripción: Crea el puesto de comida a partir del stock para cada producto, sus precios y descuentos (en caso de que posea).

Aliasing: Devuelve una referencia modificable del puesto. Utilicen aliasing cuando haya aliasing entre parametros de entrada y salida.

MENU(**in** p : puesto) $\rightarrow res$: conjLineal(item)

Pre $\equiv \{true\}$

Post $\equiv \{res = menu(p)\}$

Complejidad: $O(I)$

Descripción: Se crea y devuelve un conjunto con todos los items que posee el puesto a la venta.

Aliasing: Devuelve una referencia no modificable del menu del puesto. ✓

PRECIO(**in** p : puesto, **in** i : item) $\rightarrow res$: nat

Pre $\equiv \{i \in Menu(p)\}$

Post $\equiv \{res = precio(p, i)\}$ ✓

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

Descripción: Se devuelve el costo que posee el item en el puesto.

Aliasing: Devuelve el precio del item por copia.

STOCKITEM(**in** p : puesto, **in** i : item) $\rightarrow res$: cant

Pre $\equiv \{i \in menu(p)\}$

Post $\equiv \{res = stock(p, i)\}$

Complejidad: $O(\log(I))$ ✓

Descripción: Se devuelve el stock del item que posee el puesto.

Aliasing: Devuelve el stock del item por copia.

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

Pre $\equiv \{i \in menu(p)\}$

Post $\equiv \{res = descuento(p, i, k)\}$

Complejidad: $O(\log(I))$ ✓

Descripción: Se devuelve el descuento que posee el item si se compra una cantidad k del mismo en el puesto

Aliasing: Devuelve el descuento del item por copia.

$\text{GASTOPERSONA}(\text{in } p : \text{puesto}, \text{in } per : \text{persona}) \rightarrow res : \text{nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res = \text{gastosDe}(p, per)\}$ ✓

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

Descripción: Se devuelve el gasto total de una persona en el puesto

Aliasing: Devuelve el gasto de una persona por copia.

$\text{VENTAS}(\text{in } p : \text{puesto}, \text{in } per : \text{persona}) \rightarrow res : \text{lista}(\text{tupla}(\text{item}, \text{cant}))$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{multiConj}(res) = \text{ventas}(p, per)\}$ ✓

.... multiConj(res) devuelve un multiconj donde para todo item las ocurrencias del item en el multiconj son iguales a las de la lista.

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

Descripción: Se devuelve una lista con las compras que haya hecho la persona en el puesto.

Aliasing: Devuelve una referencia no modificable de la lista.

$\text{VENDER}(\text{in/out } p : \text{puesto}, \text{in } per : \text{persona}, \text{in } i : \text{item}, \text{in } k : \text{cant})$

Pre $\equiv \{p = p_0 \wedge \text{haySuficiente?}(p, i, c)\}$ ✓

Post $\equiv \{p = \text{vender}(p_0, per, i, k)\}$

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

Descripción: Se actualiza el puesto luego de que la persona haya realizado una compra de una cantidad k del item i.

Aliasing: No tiene.

$\text{OLVIDARITEM}(\text{in/out } p : \text{puesto}, \text{in } per : \text{persona}, \text{in } i : \text{item}, \text{in } k : \text{cant})$

Pre $\equiv \{p = p_0 \wedge i \in \text{menu}(p) \wedge \text{ConsumioSinPromo}(p, per, i)\}$

Post $\equiv \{p = \text{olvidarItem}(p_0, per, i)\}$ ✓

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

Descripción: Se actualiza el puesto aumentando el stock en 1 para el item pasado como parametro y actualizando el gasto de la persona como si no hubiera comprado una unidad del item en cuestion.

Aliasing: No tiene.

$\text{CONSUMIOSINPROMO}(\text{in/out } p : \text{puesto}, \text{in } per : \text{persona}, \text{in } i : \text{item}) \rightarrow res : \text{bool}$

Pre $\equiv \{i \in \text{menu}(p)\}$

Post $\equiv \{res = \text{consumioSinPromo?}(p, per, i)\}$

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

Descripción: Se devuelve True en el caso en que la persona haya comprado al menos una vez el item i sin descuento en el puesto.

lo necesitan (??)

Aliasing: No tiene.

$\text{COPIAR}(\text{in } p : \text{puesto}) \rightarrow res : \text{puesto}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res = p\}$

Complejidad: `copy(Puesto)` Pueden ser más explícitos... depende del tamaño del menú y la cantidad de ventas realizadas.

Descripción: Se crea una copia del puesto pasado como parámetro.

Aliasing: Se devuelve una referencia modificable del puesto copiado.

Representación

puesto se representa con `estr`

donde `cant`, `item`, `persona` es `nat`

```
donde estr es tupla(stockYPrecio: diccLog(item, tupla(precio:dinero, stock:nat))  
    , descuento: diccLog(item, arregloDimensionable(nat)) ✓  
    , ventas: diccLog(persona, tupla(gasto:dinero, consumos:lista(tupla(item: item,  
    cant: cant)))) ✓  
    , itCompraSinDescuento: diccLog(persona, diccLog(item, lista(itLista(tupla(item:  
    item, cant: cant)))) )
```

Descripción de la estructura para facilitar la comprensión de los algoritmos: gracias!

1. `stockYPrecio`: diccionario que asocia a cada ítem del puesto su precio y stock. ✓
2. `descuento`: diccionario que asocia a cada ítem, que posee algún descuento, un arreglo sobre el cual se puede indexar con la cantidad para obtener el descuento (de la compra de dicha cantidad) para los casos donde la `cant` es menor o igual a la máxima cantidad sobre la cual hay descuento. ✓
3. `ventas`: diccionario que asocia a cada persona que haya comprado en algún puesto su gasto total y una lista de cada una de las compras (ítem y cantidad). ✓
este puesto en particular
4. `itComprasSinDescuento`: diccionario que asocia a cada persona que haya realizado alguna compra sin descuento con un diccionario que asocia a cada ítem que se haya comprado sin descuento una lista con iteradores hacia dichas compras. ✓

Rep : estr \rightarrow bool

Rep(e) \equiv true \iff (($\forall i$: item)(Definido?(e.stockYPrecio, i) \iff Definido?(e.descuento, i))) \wedge

($\forall p$: persona)(Definido?(e.ventas, p) \Rightarrow_L ($\forall n$: nat)($0 \leq n \leq$ Longitud(e.ventas(p).consumos)
 \Rightarrow_L Definido?(e.stockYPrecio, e.ventas(p).consumos[n]))) \wedge

($\forall i$: item)(Definido?(e.stockYPrecio, i) \Rightarrow_L e.stockYPrecio(i).stock \leq tamaño(e.descuento(i))) \wedge

los descuentos deberían ser independientes del stock actual.

($\forall p$: persona)((Definido?(e.itCompraSinDescuento, p) \Rightarrow Definido?(e.ventas, p)) \wedge_L

($\forall i$: item)(Definido?(e.itCompraSinDescuento(p, i) \Rightarrow_L ($\forall n$: nat)($0 \leq n \leq$ Longitud(e.itCompraSinDescuento(p, i)) \Rightarrow_L ($\exists a$: nat)($0 \leq a \leq$ Longitud(e.ventas(p).consumos) \Rightarrow_L Si-
 guiente(e.itCompraSinDescuento(p, i)[n]) = e.ventas(p).consumos[a] \wedge (\neg Definido?(descuento, i) \vee
 e.descuento(i)[e.ventas(p).consumos[a].cant] = 0))))))

\wedge ($\forall p$: persona)(Definido?(e.ventas, p) \Rightarrow_L e.ventas(p).gasto =
 Longitud(e.ventas(p).lista)

$\sum_{i=0} \text{AplicarDescuento}((e.\text{stockYPrecio}(e.\text{ventas}(p).\text{lista}[i].\text{item}).\text{precio}) \times (e.\text{ventas}(p).\text{lista}[i].\text{cant})$
 , e.descuento(e.ventas(p).lista[i].item)[e.ventas(p).lista[i].cant]))

Abs : estr $e \rightarrow$ puesto

{Rep(e)}

Abs(e) =_{obs} ($\forall p$: puesto | claves(e.stockYPrecio) = menu(p) \wedge_L

($\forall i$: item)($i \in$ menu(p) \Rightarrow_L π_1 (Significado(i, e.stockYPrecio)) = precio(p, i)

\wedge π_2 (Significado(i, e.stockYPrecio)) = stock(p, i)

\wedge ($\forall c$: cant)($c < \text{tam}(\text{Significado}(i, e.\text{descuento})) \Rightarrow_L$ Significado(i, e.descuento)[c] = descuen-
 to(p, i, c))

\wedge ($\forall c$: cant)($c \geq \text{tam}(\text{Significado}(i, e.\text{descuento})) \Rightarrow_L$

Significado(i, e.descuento)[$\text{tam}(\text{Significado}(i, e.\text{descuento}))$] = descuento(p, i, c))

\wedge multiConj(e.ventas(a).consumos) = ventas(p, a))

Algoritmos

```
iCrearPuesto(in  $p$ : dicc(item, nat), in  $s$ : dicc(item, nat), in  $d$ : dicc(item, dicc(cant, nat))  $\rightarrow res$ : estr
1:  $stockYPrecio \leftarrow VACIO()$   $\triangleright O(1)$ 
2:  $descuento \leftarrow VACIO()$   $\triangleright O(1)$ 
3:  $ventas \leftarrow VACIO()$   $\triangleright O(1)$ 
4:  $itComprasSinDescuento \leftarrow VACIO()$   $\triangleright O(1)$ 
5:  $it \leftarrow CREAMIT(p)$ 
6: while HAYSIGUIENTE( $it$ ) do  $\triangleright O(I^2)$ 
7:    $stockPrecio \leftarrow \langle SIGUIENTESIGNIFICADO(it), SIGNIFICADO(SIGUIENTECLAVE(it), s) \rangle$   $\triangleright O(1)$ 
8:    $DEFINIR(SIGUIENTECLAVE(it), stockPrecio, stockYPrecio)$   $\triangleright O(I)$ 
9:    $it \leftarrow AVANZAR(it)$ 
10: end while
11:  $it \leftarrow CREAMIT(d)$ 
12: while HAYSIGUIENTE( $it$ ) do  $\triangleright O(I^2 + I*cant)$ 
13:   Busco la maxima cantidad para la cual el item tiene descuento en el diccionario d y lo almaceno en la variable:
   cant  $\triangleright O(I)$ 
14:   // Creo un arreglo de tamaño: cant + 1, lleno de ceros
15:    $descuentos \leftarrow arregloDimensionable(cant + 1, 0)$   $\triangleright O(cant + 1)$ 
16:    $it2 \leftarrow CREAMIT(SIGNIFICADO(SIGUIENTECLAVE(it), d))$   $\triangleright O(I)$ 
17:   while HAYSIGUIENTE( $it2$ ) do  $\triangleright O(I)$ 
18:      $descuentos[SIGUIENTECLAVE(it2)] \leftarrow SIGUIENTESIGNIFICADO(it2)$   $\triangleright O(1)$ 
19:      $it2 \leftarrow AVANZAR(it2)$   $\triangleright O(1)$ 
20:   end while
21:    $dAplicable \leftarrow 0$ 
22:    $i \leftarrow 0$ 
23:   while  $i < LONGITUD(descuento)$  do  $\triangleright O(cant)$ 
24:     if  $descuento[i] \neq 0$  then
25:        $dAplicable \leftarrow descuentos[i]$ 
26:     end if
27:      $descuentos[i] \leftarrow dAplicable$ 
28:      $i \leftarrow i + 1$ 
29:   end while
30:    $DEFINIR(SIGUIENTECLAVE(it), descuentos, descuento)$   $\triangleright O(I)$ 
31:    $it \leftarrow AVANZAR(it)$ 
32: end while
33:  $res \leftarrow \langle stockYPrecio, descuento, ventas, itCompraSinDescuento \rangle$   $\nabla$ 
```

COMPLEJIDAD: $O(I^2 + I*cant)$

iMenu(in e : **estr**) $\rightarrow res$: conjLineal(item)

```
1:  $claves \leftarrow \text{VACÍO}()$ 
2:  $it \leftarrow \text{CREARIT}(e.stockYPrecio)$  ▷ O(1)
3: while HAYSIGUIENTE?(it) do el ciclo como un todo no es O(1)
4:    $\text{AVANZAR}(\text{SIGUIENTECLAVE}(it), claves)$  ▷ O(1)
5:    $it \leftarrow \text{AVANZAR}(it)$  ▷ O(1)
6: end while
7: // Se devuelven las claves de e.stockYPrecio como un conjLineal
8: return  $claves$  ▷ O(I)
```

COMPLEJIDAD: O(I)

iStockItem(in e : **estr**, in i : **item**) $\rightarrow res$: nat

```
1: return  $\text{SIGNIFICADO}(i, e.stockYPrecio).stock$ 
```

COMPLEJIDAD: O(1) O(log(I)) ✓

iDescuentoItem(in e : **estr**, in i : **item**, in k : **cant**) $\rightarrow res$: nat

```
1: if DEFINIDO?(i, e.descuento) then ▷ O(log(I))
2:    $maximaCant \leftarrow \text{TAMANIO}(\text{SIGNIFICADO}(i, e.descuento))$  ▷ O(log(I))
3:   if  $k > maximaCant$  then
4:     return  $\text{SIGNIFICADO}(i, e.descuento)[maximaCant]$  ▷ O(log(I))
5:   else
6:     return  $\text{SIGNIFICADO}(i, e.descuento)[k]$  ▷ O(log(I))
7:   end if
8: else
9:   return 0
10: end if
```

COMPLEJIDAD: O(LOG(I)) ✓

iGastoPersona(in e : **estr**, in p : **persona**) $\rightarrow res$: nat

```
1: if DEFINIDO?(p.ventas) then ▷ O(log A)
2:   return  $\text{SIGNIFICADO}(p, e.ventas).gasto$  ▷ O(log A)
3: else
4:   return 0
5: end if
```

COMPLEJIDAD: O(LOG(A)) ✓

iPrecio(in e : **estr**, in i : **item**) $\rightarrow res$: nat

```
   return  $\text{SIGNIFICADO}(i, e.stockYPrecio).precio$  ▷ O(log I)
```

COMPLEJIDAD: O(LOG(I)) ✓

iVentas(in e : estr, in p : persona) $\rightarrow res$: lista(tupla(item, nat))

1: **if** DEFINIDO?($p.ventas$) **then**

▷ $O(\log A)$

2: **return** SIGNIFICADO($p, e.ventas$).consumos

▷ $O(\log A)$

3: **else**

4: **return** VACÍA()

5: **end if**

COMPLEJIDAD: $O(\log(A))$



iVender(in/out e : estr, in p : persona, in i : item, in k : cant)

```
1:  $precio \leftarrow \text{SIGNIFICADO}(i, e.\text{stockYPrecio}).\text{precio} * k$  ▷  $O(\log I)$ 
2:  $descuento \leftarrow \text{DescuentoItem}(e, i, k)$  ▷  $O(\log I)$ 
3:  $precioTotal \leftarrow \text{AplicarDescuento}(precio, descuento)$  ▷  $O(1)$ 
4: if  $descuento \neq 0$  then
5:   if  $\text{DEFINIDO?}(p, e.\text{ventas})$  then ▷  $O(\log A)$ 
6:      $info \leftarrow \text{SIGNIFICADO}(p, e.\text{ventas})$  ▷  $O(\log A)$ 
7:      $info.gasto \leftarrow info.gasto + precioTotal$  ▷  $O(1)$ 
8:      $\text{AGREGAR}(< i, k >, info.consumos)$  ▷  $O(1)$ 
9:   else
10:     $consumos \leftarrow \text{VACÍA}()$  ▷  $O(1)$ 
11:     $\text{DEFINIR}(p, e.\text{ventas}, \text{Tupla}(precioTotal, \text{AGREGAR}(< i, k >, consumos)))$  ▷  $O(\log(A) + 1)$ 
12:  end if
13: else
14:   if  $\text{DEFINIDO?}(p, e.\text{ventas})$  then ▷  $O(\log A)$ 
15:      $info \leftarrow \text{SIGNIFICADO}(p, e.\text{ventas})$  ▷  $O(\log A)$ 
16:      $info.gasto \leftarrow info.gasto + precioTotal$ 
17:      $it \leftarrow \text{AGREGAR}(< i, k >, info.consumos)$  ▷  $O(1)$ 
18:     if  $\text{DEFINIDO?}(p, e.itComprasSinDescuento)$  then ▷  $O(\log A)$ 
19:       if  $\text{DEFINIDO?}(i, \text{SIGNIFICADO}(p, e.itComprasSinDescuento))$  then ▷  $O(\log I)$ 
20:          $\text{AGREGAR}(it, \text{SIGNIFICADO}(i, \text{SIGNIFICADO}(p, e.itComprasSinDescuento)))$ 
21:       else
22:          $lista \leftarrow \text{VACÍA}()$  ▷  $O(1)$ 
23:          $\text{AGREGAR}(it, lista)$  ▷  $O(1)$ 
24:          $\text{DEFINIR}(i, lista, \text{SIGNIFICADO}(p, e.itComprasSinDescuento))$  ▷  $O(\log I)$ 
25:       end if
26:     else
27:        $\text{DEFINIR}(p, \text{VACIO}(), e.itComprasSinDescuento)$  ▷  $O(\log A)$ 
28:        $lista \leftarrow \text{VACÍA}()$  ▷  $O(1)$ 
29:        $\text{AGREGAR}(it, lista)$  ▷  $O(1)$ 
30:        $\text{DEFINIR}(i, lista, \text{SIGNIFICADO}(p, e.itComprasSinDescuento))$  ▷  $O(\log I + \log A)$ 
31:     end if
32:   else
33:      $consumos \leftarrow \text{VACÍA}()$  ▷  $O(1)$ 
34:      $it \leftarrow \text{AGREGAR}(< i, k >, consumos)$  ▷  $O(\log A + 1)$ 
35:      $\text{DEFINIR}(p, \text{VACIO}(), e.itComprasSinDescuento)$  ▷  $O(\log A)$ 
36:      $lista \leftarrow \text{VACÍA}()$  ▷  $O(1)$ 
37:      $\text{AGREGAR}(it, lista)$  ▷  $O(1)$ 
38:      $\text{DEFINIR}(i, lista, \text{SIGNIFICADO}(p, e.itComprasSinDescuento))$  ▷  $O(\log I + \log A)$ 
39:   end if
40: end if
41:  $infoItem \leftarrow \text{SIGNIFICADO}(i, e.\text{stockYPrecio})$  ▷  $O(\log I)$ 
42:  $infoItem.stock \leftarrow infoItem.stock - 1$ 
```

COMPLEJIDAD: $O(\log(A) + \log(I))$

iOlvidarItem(in/out e : estr, in p : persona, in i : item)

```
1:  $itComprasSP \leftarrow \text{SIGNIFICADO}(i, \text{SIGNIFICADO}(p, e.itComprasSinDescuento))$   $\triangleright O(\log A + \log I)$ 
2: // Obtengo las compras sin descuento del item i realizadas por p, en el puesto
3: // Me quedo con la primera en el registro, dado que lo puedo hacer en  $O(1)$ 
4:  $it \leftarrow \text{PRIMERO}(itComprasSP)$   $\triangleright O(1)$ 
5: // Me fijo si hay que eliminar todo el registro o solo una unidad de la cantidad
6: if  $\text{SIGUIENTE}(it).cant = 1$  then  $\triangleright O(1)$ 
7:    $\text{ELIMINARSIGUIENTE}(it)$   $\triangleright O(1)$ 
8:    $\text{FIN}(listaItComprasSP)$   $\triangleright O(1)$ 
9: else
10:    $\text{SIGUIENTE}(it).cant \leftarrow \text{SIGUIENTE}(it).cant - 1$   $\triangleright O(1)$ 
11: end if
12: // Ajusto el stock del producto modificado
13:  $\text{SIGNIFICADO}(i, e.stockYPrecio).stock \leftarrow \text{SIGNIFICADO}(i, e.stockYPrecio).stock + 1$   $\triangleright O(\log I)$ 
14: // Ajusto el gasto total de la persona
15:  $precio \leftarrow \text{SIGNIFICADO}(i, e.stockYPrecio).precio$   $\triangleright O(\log I)$ 
16:  $\text{SIGNIFICADO}(p, e.ventas).gasto \leftarrow \text{SIGNIFICADO}(p, e.ventas).gasto - precio$   $\triangleright O(\log A)$ 
```

COMPLEJIDAD: $O(\log(A) + \log(I))$

iConsumioSinPromo(in e : estr, in p : persona, in i : item) $\rightarrow res$: bool

```
1: if  $\text{DEFINIDO?}(p, e.itComprasSinDescuento)$  then  $\triangleright O(\log A)$ 
2:   if  $\text{DEFINIDO?}(i, \text{SIGNIFICADO}(p, e.itComprasSinDescuento))$  then  $\triangleright O(\log I)$ 
3:     if  $\text{VACÍA?}(\text{SIGNIFICADO}(i, \text{SIGNIFICADO}(p, e.itComprasSinDescuento)))$  then
4:       return false
5:     else
6:       return true
7:     end if
8:   else
9:     return false
10:  end if
11: else
12:   return false
13: end if
```

COMPLEJIDAD: $O(\log(A) + \log(I))$

2. Modulo LollaPattooza

Interfaz

se explica con: LOLLAPATUZA, PUESTO DE COMIDA, ITEM, PERSONA, PUESTOId

géneros: lolla

NUEVOLOLLA(in $puestos$: diccLog(puestoId, puesto), in per : conj(personas)) $\rightarrow res$: lolla

Pre $\equiv \{ \text{vendenAlMismoPrecio}(\text{significados}(\text{puestos})) \wedge \text{NoVendieronAun}(\text{significados}(\text{puestos}))$

$\wedge \neg \emptyset?(per) \wedge \neg \emptyset?(claves(\text{puestos})) \}$

Post $\equiv \{res = crearLolla(puestos, per)\}$

Complejidad: $O(P * copy(puesto) + A)$

Descripción: Se crea un nuevo Lollapatuza a con los puestos y personas pasados como parametros.

Aliasing: Devuelve una referencia modificable del Lollapatuza creado. **No hay aliasing**

PUESTOS(**in** $l: lolla$) $\rightarrow res : diccLog(puestosId, puestos)$

Pre $\equiv \{true\}$

Post $\equiv \{res = puestos(l)\}$

Complejidad: $O(1)$

Descripción: Se devuelven los puestos junto con sus ids correspondientes.

Aliasing: Devuelve una referencia no modificable. **✓**

PERSONAS(**in** $l: lolla$) $\rightarrow res : conj(personas)$

Pre $\equiv \{true\}$

Post $\equiv \{res = personas(l)\}$ **✓**

Complejidad: $O(1)$

Descripción: Se devuelven todas las personas que estan registradas en el lollapatuza.

Aliasing: Devuelve una referencia no modificable.

REGISTRARCOMPRA(**in/out** $l: lolla$, **in** $p: persona$, **in** $pi: puestoId$, **in** $i: item$, **in** $k: cant$)

Pre $\equiv \{l = l_0 \wedge p \in personas(l) \wedge def?(pi, puestos(l)) \wedge_L$

$haySuficiente?(obtener(pi, puestos(l)), i, k)\}$

Post $\equiv \{l = vender(l_0, p, pi, k)\}$ **✓**

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

Descripción: Se actualiza el lollapatuza con la compra de una cantidad k del item i para la persona en el puesto pasado como parametro.

Aliasing: No tiene.

HACKEAR(**in/out** $l: lolla$, **in** $per: persona$, **in** $i: item$)

Pre $\equiv \{l = l_0 \wedge_L ConsumioSinPromoEnAlgunPuesto(l, a, i)\}$ **✓**

Post $\equiv \{l = hackear(l_0, per, i)\}$

Complejidad: $O(\log(A) + \log(I))$ o $O(\log(A) + \log(I) + \log(P))$ **cuando vale cada complejidad?**

Descripción: Se actualiza el lollapatuza eliminando la compra de un item para la persona pasada por parametro, en alguna compra del puesto de menor Id en el que la persona haya realizado al menos una compra de dicho item sin descuento.

Aliasing: No tiene.

GASTOTOTAL(**in** $l: lolla$, **in** $pers: persona$) $\rightarrow res : nat$

Pre $\equiv \{per \in personas(l)\}$

Post $\equiv \{res = gastoTotal(l, per)\}$

Complejidad: $O(\log(A))$ **✓**

Descripción: Se devuelve el gasto de una persona en el lollapatuza

Aliasing: Devuelve el gasto de la persona por copia.

PERSONAQUEMASGASTO(in l : lolla) $\rightarrow res$: per

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res = masGasto(l)\}$

Complejidad: $O(1)$

Descripción: Se devuelve la persona con el mayor gasto en el lollapatuza. ✓

Aliasing: Se devuelve la persona por copia.

MENORSTOCKITEM(in l : lolla, in i : item) $\rightarrow res$: puestoId

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res = menorStock(l, i)\}$

Complejidad: $O(P * \log(I))$ ✓

Descripción: Se devuelve el puesto del lollapatuza que posea la menor cantidad del item i en su stock. Desempata por menor Id.

Aliasing: Devuelve el id del puesto por copia.

Representación

lolla se representa con estr

donde *persona*, *gasto*, *puestoId*, *item* es **nat**

donde **estr** es `tupla(puestos: diccLog(puestoId, puesto) ✓
 , personas: conjLineal(persona) ✓
 , itPuestos: diccLog(puestoId, itDiccLog(puestoId, puesto)))
 , gastoPersonas: diccLog(persona, itcolaMaxprior(tupla(nat, persona))) ✓
 , personaQueMasGasto: colaMaxprior(tupla(gasto: nat, persona: persona)) ✓
 , puestoMenorId: diccLog(persona, diccLog(item, diccLog(puestoId,
 itDiccLog(puestoId, puesto)))) ✓
)`

Descripción de la estructura para facilitar la comprensión de los algoritmos:

1. **puestos:** diccionario que asocia a cada ID de un puesto con el puesto mismo.
2. **personas:** conjunto que contiene a todas las personas del Lollapatuza.
3. **itPuestos:** diccionario que asocia a cada ID de un puesto con el iterador a la ubicación del mismo en e.puestos.
para qué necesitan este diccionario si en puestos ya puedo buscar por puestoID ???
4. **gastoPersonas:** diccionario que asocia a cada persona que haya realizado alguna compra con un iterador a la cola de prioridad e.PersonaQueMasGasto.
5. **personaQueMasGasto:** cola de prioridad que toma una tupla de $\langle \text{gasto}, \text{persona} \rangle$ ordenada de mayor a menor en base al gasto del individuo. El gasto es el total gastado por esa persona.
6. **puestoMenorId:** diccionario que asocia a una persona que realizó una compra con otro diccionario, el cual asocia alguno de los items que compro con otro diccionario, que toma la ID de un puesto y devuelve un iterador a e.puestos

donde se ubica el puesto mismo. Llamado puestoMenorID pues es requerido para el hackeo, donde se busca el puesto de menor id donde se realizo una compra. Sobre el empate... **qué cosa sobre el empate...?** ✓

Rep : estr \rightarrow bool

$$\begin{aligned} \text{Rep}(e) \equiv \text{true} \iff & (\forall pi: \text{puestoId})(\text{Definido?}(e.\text{puestos}, pi) \Rightarrow_L \text{Definido?}(e.\text{itPuestos}, pi) \wedge (\nexists py: \text{puestoId})(\text{Definido?}(e.\text{puestos}, py) \wedge_L e.\text{puestos}(pi) = e.\text{puestos}(py))) \wedge \\ & (\forall p: \text{persona})(\text{Definido?}(e.\text{gastoPersonas}, p) \Rightarrow_L \text{Pertenece?}(p, e.\text{personas})) \wedge \checkmark \\ & (\forall p: \text{persona})(\text{Definido?}(e.\text{puestoMenorId}, p) \Rightarrow_L \text{Definido?}(e.\text{gastoPersonas}, p)) \wedge \checkmark \\ & ((\forall p: \text{persona})((\text{Definido?}(e.\text{puestoMenorId}, p) \Rightarrow_L (\forall i: \text{item})((\text{Definido?}(e.\text{puestoMenorId}(p), i) \Rightarrow_L \\ & (\forall pi: \text{puestoId})(\text{Definido?}(e.\text{puestoMenorId}(p, i), pi) \Rightarrow_L \text{Definido?}(e.\text{puestos}, pi) \wedge_L \text{Pertenece?}(i, \text{Menu}(e.\text{puestos}(pi))) \wedge \text{SiguienteSignificado}(e.\text{puestoMenorId}(p, i, pi) = \text{Significado}(e.\text{puestos}, pi)))))) \wedge \\ & (\forall pi: \text{puestoId})(\text{Definido?}(e.\text{itPuestos}, pi) \Rightarrow_L \text{SiguienteSignificado}(e.\text{itPuestos}, pi) = \text{Significado}(e.\text{puestos}, pi)) \wedge \\ & (\forall pa: \text{persona})(\text{Definido?}(e.\text{gastoPersonas}, pa) \Rightarrow_L (\nexists pb: \text{persona})(\text{Definido?}(e.\text{gastoPersonas}, pb) \wedge_L \text{Significado}(e.\text{gastoPersonas}, pa) = \text{Significado}(e.\text{gastoPersonas}, pb))) \wedge \\ & (\forall p: \text{persona})(\text{Definido?}(e.\text{gastoPersonas}, p) \Rightarrow_L \pi_2(\text{Siguiente}(\text{Significado}(e.\text{gastoPersonas}, p))) = p) \wedge \\ & (\text{tamano}(e.\text{personaQueMasGasto}) = \# \text{Claves}(e.\text{gastoPersonas})) \end{aligned}$$

Abs : estr $e \rightarrow$ lolla

{Rep(e)}

Abs(e) =_{obs} ($\forall l: \text{lolla} \mid (e.\text{puestos} = \text{puestos}(l)) \wedge e.\text{personas} = \text{personas}(l)$) ✓

Ojo, acá no hay para todo...

Algoritmos

iNuevoLolla(in *puesto*: diccLog(puestoId, puesto), in *per*: conj(persona)) → *res*: estr

```
1: // Inicializo las estructuras vacias que corresponden.
2: puestos ← VACÍO()                                ▷ O(1)
3: personas ← VACÍO()                                ▷ O(1)
4: itPuestos ← VACÍO()                                ▷ O(1)
5: gastoPersonas ← VACÍO()                            ▷ O(1)
6: personaQueMasGato ← VACÍA(CARDINAL(PER))           ▷ O(1)
7: puestoMenorId ← VACÍO()                            ▷ O(1)
8: it ← CREAMIT(per)
9: // Creamos el conjunto con las personas.
10: while HAYSIGUIENTE(it) do                        ▷ O(A)
11:   AGREGARRAPIDO(SIGUIENTE(it), personas)          ▷ O(1)
12: end while
13: it ← CREAMIT(puestos)
14: // Defino cada uno de los puestos en el diccionario que corresponde.
15: // A la vez guardo los iteradores y los defino en el dicc que corresponde.
16: while HAYSIGUIENTE(it) do                            ▷ O(P)
17:   itPuesto ← DEFINIR(SIGUIENTECLAVE(it), SIGUIENTESIGNIFICADO(it), puestos)    ▷ O(log(P) + copy(puesto))
18:   DEFINIR(SIGUIENTECLAVE(it), itPuesto, itPuestos)    ▷ O(log(P))
19: end while
20: res ← < puestos, personas, itPuestos, gastoPersonas, personaQueMasGasto, puestoMenorID >
21: return res
```

COMPLEJIDAD: $O(P * \text{COPY}(\text{PUESTO}) + A)$

iPuestos(in *e*: estr) → *res*: diccLog(puestoId, puesto)

```
1: return e.puestos
```

COMPLEJIDAD: $O(1)$

iPersonas(in *e*: estr) → *res*: conjLineal(persona)

```
1: return e.personas
```

COMPLEJIDAD: $O(1)$

iRegistrarCompra(in/out e : estr, in p : persona, in pi : puestoId, in i : item, in k : cant)

```

1: // Obtengo el iterador al puesto donde se realiza la compra
2: // En los casos donde deba actualizar e.puestoMenorId voy a necesitar guardarlo.
3:  $it \leftarrow \text{SIGNIFICADO}(pi, e.itPuestos)$  ✓  $\triangleright O(\log(P))$ 
4: // Una vez obtenido el puesto en el que se realiza la compra.
5: // Realizo la compra en ese puesto
6:  $puesto \leftarrow \text{siguienteSignificado}(it)$  ✓
7:  $vender(puesto, p, i, k)$   $\triangleright O(\log(A) + \log(I))$  ✓
8: // Obtengo el precio y el descuento para actualizar el gasto.
9:  $descuento \leftarrow \text{DescuentoItem}(puesto, i, k)$   $\triangleright O(\log(I))$ 
10:  $precio \leftarrow \text{Precio}(puesto, i) * k$   $\triangleright O(\log(I))$ 
11: // Veo si la persona realizo la compra con o sin descuento.
12: if  $descuento \neq 0$  then  $\triangleright O(I)$ 
13:    $gasto \leftarrow \text{AplicarDescuento}(precio, descuento)$   $\triangleright O(1)$ 
14:   // Actualizo el gasto.
15:   if  $\text{DEFINIDO?}(p, e.gastoPersonas)$  then
16:     //Tengo que actualizar el gasto de la persona, como tengo la informacion en una colaMaxprior
17:     // No nos sirve buscar, lo mejor es eliminar y encolar en complejidad logarítmica. No entendí el comentario.
18:      $itGasto \leftarrow \text{SIGNIFICADO}(p, e.gastoPersonas)$   $\triangleright O(\log(A))$  ✓
19:      $nuevoGasto \leftarrow gasto + \text{SIGUIENTE}(itGasto).gasto$   $\triangleright O(1)$ 
20:      $\text{ELIMINARSIGUIENTE}(itGasto)$   $\triangleright O(\log(A))$ 
21:      $nuevoIt \leftarrow \text{ENCOLAR}(< nuevoGasto, p >, e.personaQueMasGasto)$   $\triangleright O(\log(A))$ 
22:     // Redefino p en e.gastoPersonas. ✓
23:      $\text{DEFINIR}(p, nuevoIt, e.gastoPersona)$   $\triangleright O(\log(A))$ 
24:   else
25:      $nuevoIt \leftarrow \text{ENCOLAR}(< gasto, p >, e.personaQueMasGasto)$   $\triangleright O(\log(A))$ 
26:      $\text{DEFINIR}(p, nuevoIt, e.gastoPersona)$   $\triangleright O(\log(A))$ 
27:   end if
28: else
29:   if  $\text{DEFINIDO?}(p, e.gastoPersonas)$  then  $\triangleright O(\log(A))$ 
30:      $itGasto \leftarrow \text{SIGNIFICADO}(p, e.gastoPersonas)$   $\triangleright O(\log(A))$ 
31:      $nuevoGasto \leftarrow precio + \text{SIGUIENTE}(itGasto).gasto$   $\triangleright O(1)$ 
32:      $\text{ELIMINARSIGUIENTE}(itGasto)$   $\triangleright O(\log(A))$ 
33:      $nuevoIt \leftarrow \text{ENCOLAR}(< nuevoGasto, p >, e.personaQueMasGasto)$   $\triangleright O(\log(A))$ 
34:     // Redefino p en e.gastoPersonas.
35:      $\text{DEFINIR}(p, nuevoIt, e.gastoPersona)$   $\triangleright O(\log(A))$ 
36:   else
37:      $nuevoIt \leftarrow \text{ENCOLAR}(< precio, p >, e.personaQueMasGasto)$ 
38:      $\text{DEFINIR}(p, nuevoIt, e.gastoPersona)$ 
39:   end if
40:   // Ahora hay que actualizar e.puestoMenorId. ✓
41:   if  $\text{DEFINIDO?}(p, e.puestoMenorId)$  then  $\triangleright O(\log(A))$ 
42:     if  $\text{DEFINIDO?}(i, \text{SIGNIFICADO}(p, e.puestoMenorId))$  then  $\triangleright O(\log(A) + \log(I))$ 
43:       // Solo nos interesa definir el pi una sola vez.
44:       if not  $\text{DEFINIDO?}(pi, \text{SIGNIFICADO}(i, \text{SIGNIFICADO}(p, e.puestoMenorId)))$  then  $\triangleright O(\log(A) + \log(I) +$ 
 $\log(P))$ 
45:         //Guardo el iterador que cree en un principio.
46:          $\text{DEFINIR}(pi, it, \text{SIGNIFICADO}(i, \text{SIGNIFICADO}(p, e.puestoMenorId)))$   $\triangleright O(\log(A) + \log(I) + \log(P))$ 
47:       end if
48:     else
49:        $\text{DEFINIR}(i, \text{DEFINIR}(pi, it, \text{VACÍO}()), \text{SIGNIFICADO}(p, e.puestoMenorId))$   $\triangleright O(\log(A) + \log(I) + 1)$ 
50:     end if
51:   else
52:      $\text{DEFINIR}(p, \text{VACÍO}(), e.puestoMenorId)$   $\triangleright O(\log(A))$ 
53:      $\text{DEFINIR}(i, \text{DEFINIR}(pi, it, \text{VACÍO}()), \text{SIGNIFICADO}(p, e.puestoMenorId))$   $\triangleright O(\log(A) + \log(I) + 1)$ 
54:   end if
55: end if

```

COMPLEJIDAD: $O(\log(P) + \log(A) + \log(I))$

iHackear(in/out e : **estr**, in p : **persona**, in i : **item**)

```
1: // Busco el puesto a hackear creando el iterador al dicc(p, i, itpuesto) que corresponde (en este caso e.puestoMenorID).
2: // Asumiendo que este apunta al menor de las claves de dicho diccionario. ✓
3:  $it \leftarrow \text{CREARIT}(\text{SIGNIFICADO}(i, \text{SIGNIFICADO}(p, e.puestoMenorId)))$  ✓  $\triangleright O(\log(A) + \log(I))$ 
4:  $puesto \leftarrow \text{SIGUIENTESIGNIFICADO}(\text{SIGUIENTESIGNIFICADO}(it))$  ✓  $\triangleright O(1)$ 
5: // Eliminamos la compra.
6:  $olvidarItem(puesto, p, i)$  ✓  $\triangleright O(\log(A) + \log(I))$ 
7: // Ajusto el gasto de la persona como ya sabemos (eliminando y encolando).
8:  $itGasto \leftarrow \text{SIGNIFICADO}(p, e.gastoPersonas)$  ✓  $\triangleright O(\log(A))$ 
9:  $nuevoGasto \leftarrow \text{SIGUIENTE}(itGasto).gasto - \text{Precio}(puesto, i)$   $\triangleright O(\log(I))$ 
10:  $\text{ELIMINARSIGUIENTE}(itGasto)$  ✓  $\triangleright O(A)$ 
11:  $nuevoIt \leftarrow \text{ENCOLAR}(< nuevoGasto, p >, e.personaQueMasGasto)$   $\triangleright O(\log(A))$ 
12:  $\text{DEFINIR}(p, nuevoIt, e.gastoPersona)$   $\triangleright O(\log(A))$ 
13: // Chequeo si era la ultima compra de la persona en ese puesto.
14: if not  $\text{ConsumioSinPromo}(puesto, p, i)$  then  $\triangleright O(\log(A) + \log(I))$ 
15:    $\text{BORRAR}(\text{SIGUIENTECLAVE}(it), \text{SIGNIFICADO}(i, \text{SIGNIFICADO}(p, e.puestoMenorId)))$   $\triangleright O(\log(P) + \log(A) + \log(I))$  ✓
16: end if
```

COMPLEJIDAD: $O(\log(I) + \log(A)) / O(\log(A) + \log(I) + \log(P))$

iGastoTotal(in e : **estr**, in p : **persona**) $\rightarrow res$: **gasto**

```
1: if  $\text{DEFINIDO?}(p, e.gastoPersonas)$  then
2:   return  $\text{SIGUIENTE}(\text{SIGNIFICADO}(p, e.gastoPersonas)).gasto$   $\triangleright O(\log(A))$ 
3: else
4:   return 0
5: end if ✓
```

COMPLEJIDAD: $O(\log(A))$

iPersonaQueMasGasto(in e : **estr**) $\rightarrow res$: **persona**

```
1: return  $\text{PROXIMO}(e.personaQueMasGasto).persona$   $\triangleright O(1)$  ✓
```


COMPLEJIDAD: $O(1)$

Nota: en ningun momento consideramos el caso en el que hay mas de una persona que posee el máximo gasto, el enunciado nos dice que debemos desempatar por dni, pero nuestro algoritmo desempata por orden de compra. (dado que la en la cola eliminamos y encolamos cuando se realiza una compra).

Dado que guardan la tupla (gasto, persona) se soluciona facil "la cola de prioridad sobre (gasto, persona) implementa la relación de orden de manera lexicográfica, ordenando primero por gasto y desempatando por persona"

iMenorStockItem(in e : **estr**, in i : **item**) $\rightarrow res$: **nat**

```
1: // Iteramos todos los puestos para encontrar el que posea el menor stock.
2: // Para el item i.
3:  $it \leftarrow \text{CREARIT}(e.puestos)$   $\triangleright O(1)$ 
4:  $stockMin \leftarrow stockItem(\text{SIGUIENTESIGNIFICADO}(it), i)$   $\triangleright O(\log(I))$ 
5:  $idMin \leftarrow \text{SIGUIENTECLAVE}(it)$   $\triangleright O(1)$ 
6: while HAYSIGUIENTE?( $it$ ) do  $\triangleright O(P * \log(I))$ 
7:    $stock \leftarrow stockItem(\text{SIGUIENTESIGNIFICADO}(it), i)$   $\triangleright O(\log(I))$ 
8:    $idActual \leftarrow \text{SIGUIENTECLAVE}(it)$   $\triangleright O(1)$ 
9:   if  $stockMin > stock$  then  $\triangleright O(1)$ 
10:      $stockMin \leftarrow stock$   $\triangleright O(1)$ 
11:      $idMin \leftarrow idActual$   $\triangleright O(1)$ 
12:   else
13:     if  $stockMin = stock$  then  $\triangleright O(1)$ 
14:       if  $idMin > idActual$  then  $\triangleright O(1)$ 
15:          $idMin \leftarrow idActual$   $\triangleright O(1)$ 
16:       end if
17:     end if
18:   end if
19:    $it \leftarrow \text{AVANZAR}(it)$   $\triangleright O(1)$ 
20: end while
21: return  $idMin$ 
```



COMPLEJIDAD: $O(P * \log(I))$

3. Módulos extra

3.1. Cola de PrioridadFija M_Ax(Arreglo) (α)

parametros formales α

se explica con: COLA DE PRIORIDAD

generos colaPrior

Interfaz

TAMAÑO(**in** $colaP$: colaPrior) $\rightarrow res$: nat

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res = \text{tamaño}(colaP)\}$

Descripción: Devuelve el tamaño de la cola

Aliasing: No tiene

VACIA?(**in** $colaP$: colaPrior) $\rightarrow res$: bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res = \text{vacía?}(colaP)\}$

Descripción: Devuelve true si la cola esta vacia y false en caso contrario

Aliasing: No tiene

PROXIMO(**in** $colaP$: colaPrior) $\rightarrow res$: α

Pre $\equiv \{\neg \text{vacía?}(colaP)\}$

Post $\equiv \{res = \text{proximo}(colaP)\}$

Complejidad: $O(\text{copy}(\alpha))$

Descripción: Devuelve una copia del proximo de la cola

DESENCOLAR(**in/out** $colaP$: colaPrior)

Pre $\equiv \{colaP = colaP_0 \wedge \neg \text{vacía?}(colaP)\}$

Post $\equiv \{colaP = \text{desencolar}(colaP)\}$

Complejidad: $O(\log(\text{limite}(colaP)))$

Descripción: Desencola el elemento más prioritario de la cola

VACIA(**in** $limite$: nat) $\rightarrow res$: colaPrior

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res = \text{vacía}(limite)\}$

Complejidad: $O(limite)$

Descripción: Crea una cola vacia.

ENCOLAR(**in/out** $colaP$: colaPrior, **in** a : α) $\rightarrow res$: itColaP(α)

Pre $\equiv \{colaP = colaP_0\}$

Post $\equiv \{colaP = \text{encolar}(colaP_0, a)\}$

La interfaz del iterador iría después de la interfaz del módulo (lo importante es poner todo lo relativo a la interfaz primero y después todo lo relativo a implementación)

Complejidad: $O(\text{copy}(\alpha) + \log(n))$

Descripción: Encola el elemento a por copia

3.2. Cola de Prioridad Fija Max

Representación

La estructura donde representaremos la cola nace de la necesidad de obtener la información de los nodos del heap en tiempo constante y a la vez de querer que al encolar, podemos devolver un iterador que se mantuviese al encolar otros elementos. Esto nos lleva a que en lugar de colocar nuestros nodos en un arreglo, los coloquemos en una lista enlazada, estructura donde podremos aprovechar sus iteradores que cumplen con las propiedades dichas anteriormente.

Además, esta cola acotará el tiempo de complejidad (que será logarítmica en función a la cantidad de nodo) de las funciones más importantes de una cola (encolar, desencolar, eliminar un nodo). Lo que nos lleva a que a la hora de usar esta cola, le pediremos al usuario que nos brinde dicho límite.

Es como tener el módulo de lista enlazada sin iteradores para recorrerla. Pero como las funciones `agregarAtras` y `agregarAdelante` del módulo `ListaEnlazada` te devuelven un "dedo" que apunta a los nodos, uno puede almacenar los "dedos" que salen de usar `agregarAtras` y `agregarAdelante`, almacenarlos en una bolsa y luego recorrer los dedos en la bolsa. Así se podrá recorrer la cola.

Notar que si combinamos la cola que se quiere representar, junto con alguna otra estructura donde la búsqueda es logarítmica donde mantiene una relación biunívoca entre los iteradores que resultan de encolar y TODOS los elementos de la cola, se obtiene una especie de cola de prioridad, donde la búsqueda de objetos que sabemos que están, el encolado, desencolado, y eliminación de objetos que ya sabemos que están es $O(\log \text{limite})$.

Tiene justo lo necesario para que lo podamos usar en el `lola`. Y si nos corremos del contexto de usarlo en el `lolla`, la cola le faltarán varias funciones importantes, por ejemplo, una manera de recorrerla independiente de combinaciones con otras estructuras y la pertenencia de un nodo en la cola.

colaPrioridad se representa con `estr`

donde `estr` es `tupla(colaPrior: arreglo dimensionable de itLista(tuplaAlmacen)`
 `, listaAlmacen: lista(tuplaAlmacen)`
 `, tamanoCola: nat`
 `, limite: nat)`
donde `tuplaAlmacen` es `tupla(objeto: α`
 `, itPosicion: nat`
 `)`

Algoritmos

iTamaño (in $e: \text{estr}$) $\rightarrow res: \text{nat}$

$res \leftarrow e.tamanoCola$

return res

$\triangleright O(1)$

iVacía? (**in** $e: \text{estr}$) $\rightarrow res: \text{bool}$

$res \leftarrow e.tamanoCola == 0$ ✓ ▷ $O(1)$

return res

iProximo (**in** $e: \text{estr}$) $\rightarrow res: \alpha$

$res \leftarrow \text{SIGUIENTE}(e.colPrior[0]).objeto$ ↓ ▷ $O(1)$

return res

iVacía(**in** $limite: \text{nat}$) $\rightarrow res: \text{estr}$

$colaP \leftarrow \text{CREARARREGLO}(limite)$ ▷ $O(limite)$

$almacenList \leftarrow \text{VACIA}()$ ▷ Creo una lista vacía $O(1)$

$limiteFijo \leftarrow limite$ ▷ $O(1)$

$tamanoCola \leftarrow 0$ ▷ $O(1)$

$res \leftarrow \langle colaP, almacenList, tamanoCola, limiteFijo \rangle$ ▷ $O(1)$ ✓

return res

iEncolar(**in/out** $e: \text{estruct}$, **in** $elem: \alpha$)

$it \leftarrow \text{AGREGARATRAS}(e.list, \langle elem, e.tamanoCola \rangle)$ ▷ $O(1)$

$e.colPrior[e.tamanoCola] \leftarrow it$ ▷ $O(1)$

$e.tamanoCola \leftarrow e.tamanoCola + 1$ ▷ $O(1)$

$i \leftarrow e.tamanoCola - 1$

if $i \neq 0$ **then**

while $i > 0$ **and** $*(e.colPrior[\text{padre}(i)]).objeto < *(e.colPrior[i]).objeto$ **do** ▷ $O(\log(e.limite))$

$swapHeap(e, \text{padre}(i), i)$ ▷ $O(1)$

$i \leftarrow \text{padre}(i)$ ▷ $O(1)$

end while

end if

$dirCola \leftarrow \&(e)$ ▷ Guardo la dirección de memoria asociada al arreglo. $O(1)$

return $\langle it, dirCola \rangle$.

iDesencolar(**in/out** $e: \text{estr}$)

$\text{ELIMINAR}(e, 0)$

Algoritmos Auxiliares

SWAPHEAP(**in** $e: \text{estr}$, **in** $i: \text{nat}$, **in** $j: \text{nat}$)

Pre $\equiv \{i < e.tamanoCola \wedge j < e.tamanoCola \text{ y adem\u00e1s en } e.colPrior(i) \text{ y } e.colPrior(j) \text{ deben haber iteradores de la lista enlazada no inv\u00e1lidos}\}$

Post $\equiv \{\text{devuelve la estructura donde intercambia el } iTLista \text{ que estaba en la posici\u00f3n } i \text{ con la de } j, \text{ a su vez que actualizamos la posici\u00f3n de los mismos en } e.listaAlmacen}\}$

Complejidad[$O(1)$]

[**Descripci\u00f3n** Intercambia los nodos que est\u00e1n en i y j manteniendo el \u00cdndice donde se puede encontrar cada uno en su respectiva tuplaAlmacen]

ELIMINAR(**in** $e: \text{estr}$, **in** $i: \text{nat}$)

Pre $\equiv \{\text{true}\}$

iswapHeap(in/out e : **estruct**, in i : **nat**, in j : **nat**)

$itGuardarI \leftarrow \text{COPY}(e.colPrior[i])$ \triangleright Copio el iterador $O(1)$
 $itGuardarJ \leftarrow \text{COPY}(e.colPrior[j])$ \triangleright Copio el iterador $O(1)$
 $e.colPrior[i] \leftarrow itGuardarJ$ \triangleright Muevo a j. $O(1)$
 $\text{SIGUIENTE}(itGuardarJ).itPosicion \leftarrow i$ \triangleright Actualizo la posicion asociada al It movido. $O(1)$
 $e.colPrior[j] \leftarrow itGuardarI$ \triangleright Muevo a i. $O(1)$
 $\text{SIGUIENTE}(itGuardarI).itPosicion \leftarrow j$ \triangleright Actualizo la posicion asociada al It movido. $O(1)$

COMPLEJIDAD: $O(1)$

Post $\equiv \{i < e.tamanoCola \text{ y ademas, todo el subHeap que se representa a partir de tomar el índice } i \text{ como la raiz de un subHeap, tiene en todos sus nodos } i\text{Lista validos. Ademas tambien le pediremos que toda los ancestros del nodo } i \text{ tengan en todso sus lugares } it\text{Lista validos}\}$

Complejidad: La estructura movera el último nodo (que esta en el índice $e.tamano - 1$) del Heap hacia el i -esimo elemento del heap (el nodo que queremos eliminar). Luego hará un shiftup o un Heapify a partir de la i -ésima terna del heap dependiendo de como se tenga que arreglar el invariante

Complejidad $O(e.limite)$ en gral, o si nos ponemos finos $\max\{O(i-e.tamanoCola), O(i)\}$

iEliminar(in/out e : **estruct**, in i : **nat**)

$\text{SWAPHEAP}(e, i, tamCola - 1)$ \triangleright Swapeo el nodo que quiero borrar con el nodo que es facil de borrar, a.k.a el ultimo de la cola. Luego hay que sacar el ultimo de la cola y despues arreglar el invariante en el lugar i . No olvidar que swapHeap es $O(1)$
 $\text{ELIMINARSIGUIENTE}(e.colPrior[e.tamanoCola - 1])$ \triangleright Mato el lugar en la lista enlazada donde tenia lo importante. $O(1)$
 $e.tamanoCola \leftarrow e.tamanoCola - 1$ \triangleright Actualizo el tamaño de la cola. $O(1)$
if $\text{SIGUIENTE}(e.colPrior[padre(i)]).objeto > \text{SIGUIENTE}(e.colPrior[i]).objeto$ **then**
 $\text{HEAPIFY}(e, i)$ $\triangleright O(e.limite)$
else
 $aSubir \leftarrow i$
 while $i > 0$ and $\text{textscSiguiente}(e.colPrior[padre(aSubir)]).objeto < \text{textscSiguiente}(e.colPrior[aSubir]).objeto$ **do**
 \triangleright El loop es $O(\log e.limite)$
 $\text{SWAPHEAP}(e, aSubir, padre(aSubir))$ $\triangleright O(1)$
 $aSubir \leftarrow padre(aSubir)$ $\triangleright O(1)$
 end while
end if

HEAPIFY(in e : **estr**, in i : **nat**)

Pre $\equiv \{\text{true}\}$

Post $\equiv \{i < e.tamanoCola \text{ y todos los } it\text{Lista que esten en todo el subHeap representado como el subHeap que tiene como raiz/más prioridad el nodo que esta en la pos } i, \text{ tienen } it\text{ListaValidos}\}$

Complejidad: Mantiene el invariante del heap, swapeando los nodo i -esimo hacia abajo, en sentido prioritario del heap utilizando la funcoin auxiliar swapHeap

Complejidad: Como la version de heapify que esta implementada es por recursion, y a lo sumo hago $\log_2(e.tamanoCola)$ swapeos, se tiene que la funcion de tiempo de complejidad en el peor caso es $O(e.limite)$

iHeapify(in/out e : estr, in i : nat)

```
izq  $\leftarrow 2 * i + 1$ 
der  $\leftarrow 2 * i + 2$ 
max  $\leftarrow i$ 
if izq < e.tamanoCola and SIGUIENTE(e.colasPrior[izq]).objeto > SIGUIENTE(e.colasPrior[i]).objeto then
    max  $\leftarrow$  izq
end if
if der > e.tamanoCola and SIGUIENTE(e.colasPrior[der]).objeto > SIGUIENTE(e.colasPrior[max]).objeto then
    max  $\leftarrow$  der
end if
if max  $\neq$  i then
    SWAPHEAP( $e, i, max$ )
    HEAPIFY( $e, max$ )
end if
```

$\triangleright O(1)$
 $\triangleright O(\log(e.limite - 1))$

Nota: Dado un natural i , padre lo unico que hace es calcular el natural correspondiente al indice donde deberia estar el padre del nodo i -esimo

iPadre (in i : nat) $\rightarrow res$: nat

```
1: res  $\leftarrow (i/2)$ 
2: if i % 2 = 0 then
3:     res  $\leftarrow$  res - 1
4: end if
5: return res
```

Representación

itCola(α) se representa con estr



donde estr es tupla(itDeAlmacen: itLista(tuplaAlmacen)
, dirCola: puntero(puntero(arreglodimensionable de itLista(tuplaAlmacen))))

Algoritmos

dobles punteros? me parece que es uno solo.

EliminarSiguierte(in/out it : itCola(α))

```
indiceAEliminar  $\leftarrow$  SIGUIENTE( $it.itDeAlmacen$ ).itPosicion
ELIMINAR(* (IT.DIRCOLA), indiceAEliminar)
```

4. TAD COLA DE PRIORIDAD(α)

TAD COLA DE PRIORIDAD(α)

igualdad observacional

$$(\forall c, c' : \text{colaPrior}(\alpha)) \left(c =_{\text{obs}} c' \iff \begin{pmatrix} \text{vacía?}(c) =_{\text{obs}} \text{vacía?}(c') \wedge \text{tamaño}(c) =_{\text{obs}} \text{tamaño}(c') \\ \wedge \text{limite}(c) =_{\text{obs}} \text{limite}(c') \wedge_{\text{L}} (\neg \text{vacía?}(c) \Rightarrow_{\text{L}} (\text{próximo}(c) \\ =_{\text{obs}} \text{próximo}(c'))) \wedge (\neg \text{vacía?}(c) \Rightarrow_{\text{L}} (\text{desencolar}(c) =_{\text{obs}} \\ \text{desencolar}(c'))) \end{pmatrix} \right)$$

parámetros formales

gêneros α

Ya estaba definido en tads basicos....

$$\text{operaciones } \bullet < \bullet : \alpha \times \alpha \longrightarrow \text{bool}$$

g�neros	colaPrior(α)
---------	-----------------------

exporta	colaPrior(α), generadores, observadores
----------------	--

usa	BOOL
-----	------

observadores básicos

$$\text{tamanio} \quad : \quad \text{colaPrior}(\alpha) \quad \longrightarrow \quad \text{nat}$$
$$\text{limite} \quad : \text{colaPrior}(\alpha) \quad \longrightarrow \text{nat}$$
$$\text{próximo} \quad : \text{colaPrior}(\alpha) \, c \quad \longrightarrow \quad \alpha \quad \quad \quad \{\neg \text{vacía?}(c)\}$$
$$\text{desencolar} : \text{colaPrior}(\alpha) \ c \longrightarrow \text{colaPrior}(\alpha) \quad \{\neg \text{vacía?}(c)\}$$

generadores

$$\text{vacía} : \text{nat} \longrightarrow \text{colaPrior}(\alpha)$$
$$\text{encolar} \quad : \quad \alpha \times \text{colaPrior}(\alpha) \quad \longrightarrow \quad \text{colaPrior}(\alpha)$$
$$\text{axioms} \quad \forall c: \text{colaPrior}(\alpha), \forall e: \alpha, \forall l: \text{nat}$$
$$\text{limite}(\text{vacía}(1)) \equiv 1$$
$$\text{limite}(\text{encolar}(e, c)) \equiv \text{limite}(c)$$
$$\text{tamano}(\text{vacía}(l)) \equiv 0$$
$$\text{tamano}(\text{encolar}(e, c)) \equiv 1 + \text{tamano}(c)$$

```
vacía?(c)      ≡ if tamaño(c) = 0 then True else false fi
```

$$\text{pr\u00f3ximo}(\text{encolar}(e, c)) \quad \equiv \quad \text{if vac\u00eda?}(c) \vee_{\text{L}} \text{pr\u00f3ximo}(c) < e \text{ then } e \text{ else } \text{pr\u00f3ximo}(c) \text{ fi}$$
$$\text{desencolar}(\text{encolar}(e, c)) \equiv \text{if } \text{vacía?}(c) \vee_{\text{L}} \text{próximo}(c) < e \text{ then } c \text{ else } \text{encolar}(e, \text{desencolar}(c)) \text{ fi}$$

Fin TAD

5. Apendice.

5.1. Algoritmos Auxiliares

```
div (in  $n : \text{nat}$ , in  $k : \text{nat}$ )  $\rightarrow res : \text{nat}$ 
   $N \leftarrow n$ 
   $K \leftarrow k$ 
   $Div \leftarrow 0$ 
  while  $N \geq K$  do
     $Div \leftarrow 1 + Div$ 
     $N \leftarrow N - K$ 
  end while
  return  $Div$ 
```

```
aplicarDescuento (in  $p : \text{nat}$ , in  $d : \text{nat}$ )  $\rightarrow res : \text{nat}$ 
  return  $div(p * (100 - d), 100)$ 
```

5.2. Operaciones de Tad auxiliares.

```
multiConj : secu( $\alpha$ )  $\longrightarrow$  multiconj( $\alpha$ )
multiConj(s)  $\equiv$  if long(s) = 0 then  $\emptyset$  else Ag(prim(s), multiConj(fin(s))) fi
```