Algoritmos y Estructuras de Datos II

Trabajo Práctico 2

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

Lollapatuza

TADgrupo

Integrante	LU	Correo electrónico
Aquilante, Juan Pablo	755/18	aquilantejp@outlook.es
Bocanegra, Oscar Iván	537/22	oscarbocanegraush@gmail.com
Flores Galvan, Silvina	689/22	silu204@gmail.com
Sanchez, Gastón	361/22	gasanchez@dc.uba.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega	Alexis	A
Segunda entrega		

Muy buen TP.
Tienen algunos bugs, ojo de cara al TP3. Pero muy buen trabajo

Índice

1. Módulo Lollapatuza **3**
1.1. Interfaz . . . 3
1.2. Operaciones básicas de Lollapatuza . . . 3
1.3. Representación del lollapatuza . . . 4
1.4. Algoritmos . . . 7

2. Módulo PuestoDeComida **10**
2.1. Interfaz del módulo . . . 10
2.2. Operaciones básicas de Puesto . . . 10
2.3. Representación de puesto . . . 11
2.4. Algoritmos . . . 13

3. Módulo Cola de Prioridad Acotada($\langle\alpha, \beta\rangle$) **15**
3.1. TAD Cola de Prioridad Acotada(γ) . . . 15
3.2. Interfaz . . . 16
3.3. Representación . . . 17
3.4. Algoritmos . . . 18
3.5. Funciones auxiliares . . . 19

1. Módulo Lollapatuza

1.1. Interfaz

Interfaz

parámetros formales

géneros lolla

función $\bullet = \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (k_1 = k_2)$}
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ's

función COPIAR(**in** $k : \kappa$) $\rightarrow$ *res* : κ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} k$}
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

se explica con: LOLLAPATUZA

géneros: `lolla`

1.2. Operaciones básicas de Lollapatuza

CREARLOLLA(**in** *ps* : `dicc(puestoID,puesto)`, **in** *as* : `conj(persona)`) $\rightarrow$ *res* : `lolla`
Pre $\equiv$ {vendenAlMismoPrecio(significados(ps)) $\wedge$ NoVendieronAun(significados(ps)) $\wedge$ $\neg\emptyset?$(as) $\wedge$ $\neg\emptyset?$(claves(ps))}
Post $\equiv$ {res $=_{\text{obs}}$ crearLolla(ps,as)}
Complejidad: $\mathcal{O}(A * log(A))$
Descripción: genera un nuevo Lollapatuza.
Aliasing: Se devuelve una referencia modificable

VENDER(**in/out** l : `lolla`, **in** pi : `puestoID`, **in** a : `persona`, **in** i : `item`, **in** c : `cant`)
Pre $\equiv$ {$l =_{\text{obs}} l_0 \wedge$ pi $\in$ claves(l) $\wedge$ a $\in$ personas(l) $\wedge$ haySuficiente?(obtener(pi, puestos(l)), i, c)}
Post $\equiv$ {l $=_{\text{obs}}$ vender(l_0, pi, a, i, c)}
puestos **Descripción:** vende una cantidad de un item determinado de un puesto del Lollapatuza.

HACKEAR(**in/out** l : `lolla`, **in** a : `persona`, **in** i : `item`)
Pre $\equiv$ {l $=_{\text{obs}}$ $l_0 \wedge$ ConsumioSinPromoEnAlgunPuesto(l, a, i)}
Post $\equiv$ {l $=_{\text{obs}}$ hackear(l_0, a, i)}
Complejidad: $\mathcal{O}(log(A) + log(P) + log(I))$
Descripción: elimina el consumo del item pasado por parametro a la persona determinada, de alguna compra en la que no se le haya aplicado un descuento a este item. De existir el consumo en múltiples puestos de comida, se eliminará el mismo de alguno de ellos, viéndose afectado su Stock.

PUESTOS(**in** l : `lolla`) $\rightarrow$ *res* : `dicc(puestoid, puesto)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ puestos(l)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los puestos de comida con sus IDs del Lollapatuza.
Aliasing: Se devuelve una referencia modificable

PERSONAS(**in** l : `lolla`) $\rightarrow$ *res* : `conj(persona)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ personas(l)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve el conjunto de personas en el Lollapatuza
Aliasing: se devuelve el res por referencia

masGasto(**in** l : `lolla`) $\rightarrow$ *res* : `persona`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ masGasto(l)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve la persona que más gastó en el Lollapatuza.
Aliasing: se devuelve res por referencia

GastoTotal(**in** l : `lolla`, **in** p : `persona`) $\rightarrow$ *res* : `dinero`
Pre $\equiv$ {p $\in$ personas(l)}
Post $\equiv$ {res $=_{\text{obs}}$ gastoTotal(l,p)}
Complejidad: $\mathcal{O}(log(A))$
Descripción: devuelve lo que gasto la persona en el lollapatuza

MenorStock(**in** l : `lolla`, **in** i : `item`) $\rightarrow$ *res* : `puestoid`
Pre $\equiv$ {True}
Post $\equiv$ {res $=_{\text{obs}}$ menorStock(l,i)}
Descripción: devuelve lo que gasto la persona en el lollapatuza

Orden ?

Representación

1.3. Representación del lollapatuza

`lollapatuza` **se representa con** `estr`

donde `estr` es `tupla`(*personas*: `conj(persona)`
, *punterosAGastos*: `diccLog(persona, puntero(dinero))`
, *gastosPersona*: `colaPriorA(`$\langle$`dinero: dinero, per: persona`$\rangle$`)`
, *puestos*: `diccLog(puestoID, puesto)`
, *hackeables*: `diccLog(persona, diccLog(item,`
`diccLog(puestoID, puntero(puesto))))`
)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$
(1) $\wedge$ (2) $\wedge$ (3) $\wedge$ (4) $\wedge$ (5) $\wedge$ (6) $\wedge$ (7) $\wedge$ (8) $\wedge$ (9) $\wedge$ (10) $\wedge$ (11) $\wedge$ (12) $\wedge$ (13)

(1) $\equiv$ Personas no es vacío.
$\neg(\emptyset(e.personas))$

(2) $\equiv$ Todas las personas en punterosAGastos son las personas del Lollapatuza.
claves($e.punterosAGastos$) = $e.personas$

(3) $\equiv$ Todos los punteros de punterosAGastos son distintos.
$(\forall pe1, pe2$: puntero(dinero))((∃ per1,per2: persona)($per1 \neq per2 \wedge$ {per1,per2} $\subseteq$ claves($e.punterosAGastos$) $\wedge_{\text{L}}$
obtener($per1, e.punterosAGastoss$) = $pe1 \wedge$ obtener($per2, e.punterosAGastoss$) = $pe2$) $\Rightarrow_{\text{L}}$ $pe1 \neq pe2$)

(4) $\equiv$ No hay personas repetidas en gastosPersona.
$(\forall i, j$: nat$)(0 \leq i, j <$ long($e.gastosPersona$) $\Rightarrow_{\text{L}}$ $\pi_2(e.gastosPersona[i]) = \pi_2(e.gastosPersona[j]) \iff i = j$)

(5) $\equiv$ Todas las personas de gastosPersona son las personas en el Lollapatuza.
$(\forall i$: nat$)(0 \leq i <$ long($e.gastosPersona$) $\Rightarrow_{\text{L}}$ $\pi_2(e.gastosPersona$[i]) $\in e.personas$)

Te faltaría que gastosPersona tiene a todas las personas

(6) $\equiv$ Para todo item vendido en el Lollapatuza, la cantidad de existencias vendidas en todos los puestos del mismo no puede exceder las existencias totales de dicho item.
$(\forall i$: item$)(i \in$ itemsLolla(significados($e.puestos$)) $\Rightarrow_{\text{L}}$
cantVendida(significados($e.puestos$),$e.personas$,i) $\leq$ stockTotal(significados($e.puestos$),i))

(7) $\equiv$ El gasto por persona equivale a la suma de sus gastos en cada puesto.
$(\forall per$: persona$)(per \in e.personas \Rightarrow_{\text{L}} (\exists\ d$: dinero$)(d =$ *(obtener($e.punterosAGastos$, per)) $\wedge_{\text{L}}$ gastoPersona(per, $e.puestos$) = d))

(8) ≡ Hay al menos algún puesto.
$\neg(\emptyset(\text{claves}(e.puestos)))$

(9) ≡ En puestos no hay repetidos.
$(\forall p1, p2$: puesto$)((\exists$ ID1, ID2 : puestoID$)(ID1 \neq ID2 \wedge \{ID1, ID2\} \subseteq \text{claves}(e.puestos) \wedge_{\text{L}} \text{obtener}(ID1, e.puestos) = p1 \wedge \text{obtener}(ID2, e.puestos) = p2) \Rightarrow_{\text{L}} p1 \neq p2)$
Esto no es verdad. Podés tener puestos observacionalmente iguales. Justamente es por lo que tenés ids. Para cuando cambien.

(10) ≡ Todos los puestos venden el item al mismo precio.
vendenAlMismoPrecio(significados(e.puestos))

(11) ≡ Todas las personas de hackeables son válidas.
claves(e.hackeables) ⊆ e.personas

(12) ≡ Todas los puestosIDs que estan en hackeables son IDs de los puestos del lolla, y los punteros de hackeables apuntan a sus puestos correspondientes. Y todas las personas de hackeables hicieron una compra sin descuento
$(\forall per$: persona$)(per \in$ claves(e.hackeables) $\Rightarrow_{\text{L}}$
$(\forall i$: item$)($item $\in$ claves(obtener(per, e.hackeables)) $\Rightarrow_{\text{L}}$
$(\forall ID$: puestoID$)($ID $\in$ claves(obtener(i, obtener(per, e.hackeables))) $\wedge$ ID $\in$ claves(e.puestos) $\Rightarrow_{\text{L}}$
$(\exists$ pe: puntero(puesto))(pe = obtener(ID, obtener(i, obtener(per, e.hackeables))) $\wedge$ def?(ID, e.puestos) $\wedge_{\text{L}}$ obtener(ID, e.puestos) = *pe))))))

(13) ≡ Todas las personas en hackeables hicieron una compra sin descuento del item en el puesto.
$(\forall per$: persona$)(per \in$ claves(e.hackeables) $\Rightarrow_{\text{L}}$
$(\forall i$: item$)($item $\in$ claves(obtener(per, e.hackeables)) $\Rightarrow_{\text{L}}$
$(\forall ID$: puestoID$)($ID $\in$ claves(obtener(i, obtener(per, e.hackeables))) $\Rightarrow_{\text{L}}$
$(\forall pe$: puntero(puesto))(pe = obtener(ID, obtener(i, obtener(per, e.hackeables))) $\Rightarrow_{\text{L}}$
$\neg(\emptyset$ = ventasSinPromo(*pe, per, i)))

```
cantVendida : conj(puestos) × conj(persona) × item  ⟶  cant
cantVendida(cp,cper,i) ≡ if vacio?(cp) then
                            0
                         else
                            cantVendidaPuesto(dameUno(cp),cper,i) + cantVendida(sinUno(cp),cper,i)
                         fi
cantVendidaPuesto : puesto × conj(persona) × item  ⟶  cant
cantVendidaPuesto(p,cper,i) ≡ if ∅?(cper) then
                                 0
                              else
                                 cantVendidaPersona(ventas(p,dameUno(cper)),i) +
                                 cantVendidaPuesto(p,sinUno(cper),i)
                              fi
cantVendidaPersona : multiconj(⟨item × cant⟩) × item  ⟶  cant
cantVendidaPersona(vp,i) ≡ if vacia?(vp) then
                              0
                           else
                              if π1(dameUno(vp))= i then
                                 π2(dameUno(vp))+cantVendidaPersona(sinUno(vp),i)
                              else
                                 cantVendidaPersona(fin(vp),i)
                              fi
                           fi
personasDeGastos : colaPriorA(⟨dinero × persona⟩)  ⟶  conj(persona)
personasDeGastos(cPA) ≡ if vacia?(cPA) then
                           ∅
                        else
                           Ag(π2(proximo(cPA)), personasDeGastos(desencolar(cPA)))
                        fi
stockTotal : conj(puestos) × item  ⟶  cant
stockTotal(cp,i) ≡ if vacio?(cp) then 0 else stock(dameUno(cp),i) + stockTotal(sinUno(cp),i) fi
```

itemsLolla : conj(puesto) $\longrightarrow$ conj(item)
itemsLolla(cp) $\equiv$ **if** vacio?(cp) **then** $\emptyset$ **else** menu(dameUno(cp)) $\cup$ itemsLolla(sinUno(cp)) **fi**
sumaDeGastos : colaPriorA($\langle$dinero $\times$ persona$\rangle$) $\longrightarrow$ nat
sumaDeGastos(c) $\equiv$ **if** vacia?(c) **then** 0 **else** π_1(proximo(c)) + sumaDeGastos(desencolar(c)) **fi**
gastoMaximoPosible : dicc(puestoID $\times$ puesto)) $\longrightarrow$ nat
gastoMaximoPosible(d) $\equiv$ **if** vacia?(claves(d)) **then**
0
else
gMaximoPosiblePuesto(obtener(dameUno(claves(d)))),menu(obtener(dameUno(claves(d))))
+ gastoMaximoPosible(borrar(dameUno(claves(d)),d)
fi
gMaximoPosiblePuesto : puesto $\times$ conj(item) $\longrightarrow$ nat
gMaximoPosiblePuesto(p,m) $\equiv$ **if** vacio?(m) **then**
0
else
precio(p,dameUno(m)) $\times$ stock(p,dameUno(m)) +
gMaximoPosiblePuesto(p,sinUno(m))
fi
long : colaPriorA(α) $\longrightarrow$ nat
long(c) $\equiv$ **if** vacio?(c) **then** 0 **else** 1 + long(desencolar(c)) **fi**
$\bullet$[$\bullet$] : colaPriorA(α) $\longrightarrow$ α
c[i] $\equiv$ **if** $i = 0$ **then** proximo(c) **else** desencolar(c)[i-1] **fi**
gastoPersona : per $\times$ diccLog(puestoID $\times$ puesto) $\longrightarrow$
gastoPersona(per, d) $\equiv$ sumaPorPuesto(per, significados(d))
sumaPorPuesto : per $\times$ conj(puesto) $\longrightarrow$ dinero
sumaPorPuesto(per, c) $\equiv$ **if** #c = 1 **then**
gastoEnPuesto(ventas(dameUno(c), per), dameUno(c))
else
gastoEnPuesto(ventas(dameUno(c), per), dameUno(c)) + sumaPorPuesto(c)
fi
gastoEnPuesto : multiconj($\langle$item $\times$ cant$\rangle$) $\times$ puesto $\longrightarrow$ dinero
gastoEnPuesto(m, p) $\equiv$ **if** $\emptyset$?(m) **then**
0
else
aplicarDescuento(precio(p, π_1(dameUno(m))), descuento(p, π_1(dameUno(m)),
π_2(dameUno(m)))) $\times$ π_2(dameUno(m)) + gastoEnPuesto(sinUno(m), p)
fi
Abs : estr e $\longrightarrow$ lolla {Rep(e)}
Abs(e) $\equiv$ puestos(l) $=_{\text{obs}}$ e.puestos $\wedge$ personas(l) $=_{\text{obs}}$ e.personas

1.4. Algoritmos

Algoritmos

---

iCrearLolla(**in** *ps*: `dicc(puestoID,puesto)`, **in** *as*: `conj(persona)`) → *res* : lollapatuza

1: diccGastos ← Vacio() ▷ $\mathcal{O}(1)$
2: cola ← Vacio() ▷ $\mathcal{O}(1)$
3: it ← crearIt(as) ▷ $\mathcal{O}(1)$
4: **while** HaySiguiente(it) **do** ▷ $\mathcal{O}(A)$
5: puntero ← Encolar(cola,⟨ 0, Siguiente(it) ⟩) ▷ $\mathcal{O}(log(A))$
6: Definir(diccGastos, Siguiente(it), puntero) ▷ $\mathcal{O}(log(A))$
7: Avanzar(it) ▷ $\mathcal{O}(1)$
8: **end while**
9: res ← ⟨ as, diccGastos, cola, ps, Vacio() ⟩ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(A * log(A))$
Justificación: El while se realiza A veces, y por cada bucle se define una nueva clave en el diccionario diccGastos en log(A) y se encola la cola que tiene en su significado, también en log(A).

---

iVender(**in/out** *l*: `lolla`, **in** *pid*: `puestoID`, **in** *a*: `persona`, **in** *i*: `item`, **in** *c*: `cant`)

1: *//Accedo al puesto donde se va a realizar la venta*
2: p ← Significado(l.puestos, pid) ▷ $\mathcal{O}(log(P))$
3: precioItem ← Significado(p.menu, i) ▷ $\mathcal{O}(log(I))$
4: *//Defino el descuento,*
5: *//si no tiene descuento el array devuelve* 0
6: **if** Definido?(p.descuentos,i) **then** ▷ $\mathcal{O}(log(I))$
7: descuento ← Significado(p.descuentos,i)[cant] ▷ $\mathcal{O}(log(I))$
8: **else**
9: descuento ← 0 ▷ $\mathcal{O}(1)$
10: **end if**
11: *//Registro la venta en el puesto*
12: Vender(p, a, i, c) ▷ $\mathcal{O}(log(I) + log(A))$
13: gastoVenta ← cant × precioItem × (100-descuento)/100 ▷ $\mathcal{O}(1)$
14: *//Si fue sin descuento y el puesto no era hackeable, añadirlo a hackeables*
15: **if** descuento = 0 ∧
16: ¬ Definido?(Significado(Significado(l.hackeables, a), i), pid) **then** ▷ $\mathcal{O}(log(A) + log(I) + log(P))$
17: *//creo un puntero y lo defino en l.hackeables*
18: punteroP ← &p
19: Definir(Significado(Significado(l.hackeables,a),i),pid,punteroP) ▷ $\mathcal{O}(log(A) + log(I) + log(P))$
20: **end if**
21: *//Actualizo el gasto total de la persona en el lollapatuza*
22: punteroAGasto ← Significado(l.punterosAGastos,a) ▷ $\mathcal{O}(log(A))$
23: gastoActualizado ← *punteroAGasto + gastoVenta
24: gastoPer ← <gastoActualizado,a> ▷ $\mathcal{O}(1)$
25: punteroDinero ← Encolar(l.gastosPersona,gastoPer) ▷ $\mathcal{O}(log(A))$
26: *//Actualizo el puntero del gasto de la persona*
27: Definir(l.punterosAGastos,a,punteroDinero) ▷ $\mathcal{O}(log(A))$

No veo donde lo sacás del heap, volves a heapify y volvés a encolar.
Esto es O(log<dinero>) no Persona

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

---

iPuestos(**in** *l*: `lollapatuza`) → *res* : diccLog(puestoID,puestos)

1: res ← l.puestos
Complejidad: $\mathcal{O}(1)$ ▷ $\mathcal{O}(1)$

---

iPersonas(**in** l : `lollapat`, **in** k : κ) $\rightarrow$ res : $conj(persona)$

1: res $\leftarrow$ l.personas ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$

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

1: itP $\leftarrow$ CrearIt(Significado(Significado(l.hackeables, a), item)) ▷ $\mathcal{O}(log(A) + log(I))$
2: p $\leftarrow$ *SiguienteSignificado(itP) ▷ $\mathcal{O}(1)$
3: listaVentas $\leftarrow$ Significado(Significado(p.ventasSinDesc, a), i) ▷ $\mathcal{O}(log(A) + log(I))$
4: itLista $\leftarrow$ CrearIt(listaVentas) ▷ $\mathcal{O}(1)$
5: it $\leftarrow$ Primero(listaVentas) ▷ $\mathcal{O}(1)$
6: **if** $\neg$ Siguiente(it).cant = 1 **then** ▷ $\mathcal{O}(1)$
7: Siguiente(it).cant $\leftarrow$ (Siguiente(it).cant - 1) ▷ $\mathcal{O}(1)$
8: **else**
9: EliminarSiguiente(it) ▷ $\mathcal{O}(1)$
10: **if** Longitud(listaVentas) = 1 **then** //Si el puesto deja de ser hackeable, cuesta O(log(P)) borrarlo del diccionario
11: Definir(Significado(p.ventasSinDesc, a), i, Vacia()) ▷ $\mathcal{O}(log(A) + log(I))$
12: Borrar(Significado(Significado(l.hackeables, a), i), SiguienteClave(itP)) ▷ $\mathcal{O}(log(P) + log(A) + log(I))$
13: **end if**
14: EliminarSiguiente(itLista) ▷ $\mathcal{O}(1)$
15: **end if**
16: Definir(p.stock, i, Significado(p.stock, i) + 1) ▷ $\mathcal{O}(log(I))$
17: precioItem $\leftarrow$ Significado(p.menu, i) ▷ $\mathcal{O}(log(I))$
18: Definir(p.gastosDe, a, Significado(p.gastosDe, a) - precioItem) ▷ $\mathcal{O}(log(A))$
19: puntero $\leftarrow$ Significado(l.punterosAGastos, a) ▷ $\mathcal{O}(log(A))$
20: gastoAnterior $\leftarrow$ *puntero ▷ $\mathcal{O}(1)$
21: puntGastoAct $\leftarrow$ Encolar(l.gastosPersona, $\langle$gastoAnterior - precioItem, a$\rangle$) ▷ $\mathcal{O}(log(A))$
22: Definir(l.punterosAGastos, a, puntGastoAct) ▷ $\mathcal{O}(log(A))$
Complejidad: $\mathcal{O}(log(A) + log(P) + log(I))$
Justificación: El iterador del diccionario recorre inorder. Por lo tanto en linea 1 el it apunta al puesto de menor ID.

iGastoTotal(**in** l : `lolla`, **in** a : `persona`) $\rightarrow$ res : nat

1: gasto $\leftarrow$ *(Significado(l.punterosAGastos, a)) ▷ $\mathcal{O}(log(A))$
2: res $\leftarrow$ gasto ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(log(A))$

iQuienGastoMas(**in** l : `lolla`) $\rightarrow$ res : persona

res $\leftarrow$ (Proximo(l.gastosPersona)).per ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$

Ojo que del otro lado está llamado masGastó (En la ifaz)

iMenorStock(**in** l : `lolla`, **in** i : `item`) $\rightarrow$ res : puestoID

```
1:  itP ← CrearIt(l.puestos)                                                          ▷ O(1)
2:  if Definido?(SiguienteSignificado(itP).stock, i) then                             ▷ O(log(I))
3:      minStock ← Significado(SiguienteSignificado(itP).stock, i)                    ▷ O(log(I))
4:  else
5:      minStock ← 0                                                                  ▷ O(1)
6:  end if
7:  IDminStock ← SiguienteClave(itP)                                                  ▷ O(1)
8:  while HaySiguiente(itP) do                                                        ▷ O(P)
9:      if Definido?(SiguienteSignificado(itP).stock, i) then                         ▷ O(log(I))
10:         stockActual ← Significado(SiguienteSignificado(itP).stock, i)             ▷ O(log(I))
11:         IDstockActual ← SiguienteClave(itP)                                       ▷ O(1)
12:         if stockActual < minStock ∨ (stockActual = minStock ∧ IDstockActual < IDminStock) then  ▷ O(1)
13:             minStock ← stockActual                                                ▷ O(1)
14:             IDminStock ← IDstockActual                                            ▷ O(1)
15:         end if
16:     else
17:         IDstockActual ← SiguienteClave(itP)                                       ▷ O(1)
18:         if ¬(minStock = 0) then
19:             minStock ← 0                                                          ▷ O(1)
20:         end if
21:         if IDstockActual < IDminStock then
22:             IDminStock ← IDstockActual                                            ▷ O(1)
23:         end if
24:         Avanzar(itP)                                                              ▷ O(1)
25:     end if
26: end while
27: res ← IDminStock                                                                  ▷ O(1)
```

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

Cómo lo avanzás en O(1) con AVL ?

+ log(I) por si P es 0

2. Módulo PuestoDeComida

Interfaz

2.1. Interfaz del módulo

parámetros formales
géneros puesto
función $\bullet = \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (k_1 = k_2)$}
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ's

función COPIAR(**in** $k : \kappa$) $\rightarrow$ res : κ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} k$}
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

se explica con: PUESTODECOMIDA

géneros: `puesto`

2.2. Operaciones básicas de Puesto

Poco declarativo. Es mejor poner nombres a los parámetros de los dics.

CREARPUESTO(**in** p : `dicc(item,nat)`, **in** s : `dicc(item,nat)`,**in** d : `dicc(item,dicc(nat,nat))` $\rightarrow$ res : `puesto`
Pre $\equiv$ {claves(p) = claves(s) $\wedge$ claves(p) $\subseteq$ claves(d)}
Post $\equiv$ {res $=_{\text{obs}}$ crearPuesto(m,s,d)}
Complejidad: $\mathcal{O}(I + stockInicial + log(cant))$
Descripción: genera un nuevo puesto.
Aliasing: res se devuelve como una referencia modificable

VENDER(**in/out** p : `puesto, in` a : `persona, in` i : `item, in` c : `cant`)
Pre $\equiv$ {p_0 = p $\wedge$ haySuficiente?(p, i, c)}
Post $\equiv$ {p $=_{\text{obs}}$ vender(p,a,i,c)}
Complejidad: $\mathcal{O}(log(A) + log(I))$
Descripción: vende una cantidad de un item dado a una persona dada.

MENU(**in** p : `puesto`) $\rightarrow$ res : `conj(item)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ menu(p)}
Complejidad: $\mathcal{O}(I)$
Descripción: devuelve el menú de un puesto.
Aliasing: res se devuelve como una referencia no modificable

PRECIO(**in** p : `puesto`, **in** i : `item`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {res $=_{\text{obs}}$ precio(p,i)}
Complejidad: $\mathcal{O}(log(I))$
Descripción: devuelve el precio del item en el menu del puesto.

STOCK(**in** p : `puesto`,**in** i : `item`) $\rightarrow$ res : `cant`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {res $=_{\text{obs}}$ stock(p, i)}
Complejidad: $\mathcal{O}(log(I))$
Descripción: devuelve el stock de un item.

DESCUENTO(**in** p: `puesto`,**in** i: `item`, **in** c: `cant`) $\rightarrow$ res : `nat`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {res $=_{\text{obs}}$ descuento(p, i, c)}
Complejidad: $\mathcal{O}(log(I))$
Descripción: devuelve el descuento correspondiente al puesto, item y cantidad comprados.

GASTOSDE(**in** p: `puesto`,**in** a: `persona`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ gastosDe(p, a)}
Complejidad: $\mathcal{O}(log(A))$
Descripción: devuelve el gasto realizado de la persona en el puesto

Representación

2.3. Representación de puesto

`puesto` **se representa con** `estr`

donde `estr` es `tupla`(*menu*: `diccLog(item,precio)`
, *stock*: `diccLog(item,cant)`
, *gastosDe*: `diccLog(persona,nat)`
, *descuentos*: `diccLog(item, arreglo_dimensionable de nat)`
, *ventasSinDesc*: `diccLog(persona, diccLog(item, lista(itLista)))`
, *ventas*: `diccLog(persona, lista(⟨item: item, cant: cant⟩))`
)

Poco killer la solución, pero funciona

listaAMulticonj : secu($\langle$item $\times$ cant$\rangle$) $\longrightarrow$ multiconj(item,cant)
listaAMulticonj(l) $\equiv$ **if** vacia?(l) **then** $\emptyset$ **else** Ag(prim(l), listaAMulticonj(fin(l)) **fi**
$\bullet[\bullet]$: secu(α) $\times$ nat $\longrightarrow$ α
s[i] $\equiv$ **if** $i = 0$ **then** prim(s) **else** fin(s)[$i - 1$] **fi**
Rep : puesto $\longrightarrow$ bool
Rep(l) $\equiv$ true $\Longleftrightarrow$
(1) $\wedge$ (2) $\wedge$ (3) $\wedge$ (4) $\wedge_{\text{L}}$ (5) $\wedge$ (6) $\wedge$ (7) $\wedge$ (8) $\wedge$ (9)

(1) $\equiv$ Los items que se venden en el menu deben ser los items del stock del puesto.
claves(e.menu) = claves(e.stock)

(2) $\equiv$ Los items que tienen descuento estan contenidos en los items que se venden en el menú.
claves(e.descuentos) $\subseteq$ claves(e.menu).

No deben haber repetidos en el arreglo de descuentos. (y ordenado?)

(3) $\equiv$ Los items que aparecen en las ventas del puesto deben estar contenidos en el menú del puesto.
($\forall p$: persona)(p $\in$ claves(e.ventasSinDesc) $\Rightarrow_{\text{L}}$
(($\forall i$: item)(($\exists$ n:nat)(0 $\leq$ n $<$ long(obtener(p,e.ventas())) $\wedge_{\text{L}}$ π_1(obtener(p,e.ventas())[n]) = i) $\Rightarrow_{\text{L}}$
i $\in$ claves(e.menu))))

Esto no implica que esté en ventas. Implica que si está en ventas, está en claves).
Les está faltando ver que esté en ventas.

(4) $\equiv$ Las personas en la lista de gastos del puesto son las mismas que las personas en las ventas del puesto.
claves(e.ventas) = claves(e.gastosDe)

(5) $\equiv$ El gasto de una persona debe coincidir con el costo de las compras de la misma en el puesto, considerando el precio, stock y descuento de items comprados.
($\forall p$: persona)(p $\in$ claves(e.gastosDe) $\Rightarrow_{\text{L}}$
obtener(p,e.gastosDe) = gastoPuestoPersona(obtener(p,e.ventas),e.menu,e.descuentos))

(6) $\equiv$ Las existencias vendidas totales de un item no pueden exceder el stock del item.
($\forall p$: persona)(p $\in$ claves(e.ventas) $\Rightarrow_{\text{L}}$
(($\forall i$: item)(($\exists$ n:nat)(0 $\leq$ n $<$ long(obtener(p,e.ventas())) $\wedge_{\text{L}}$ π_1(obtener(p,e.ventas())[n]) = i) $\Rightarrow_{\text{L}}$
cantVendidaPuesto(claves(e.ventas),i) $\leq$ obtener(i, e.stock))))

(7) y (8) ≡ Cada iterador de la venta de una persona que sea sin descuento, debe apuntar a una venta de esa persona con una cantidad o item asociado para los cuales un descuento no es aplicable; además, el item asociado al iterador debe ser el mismo que esta asociado a la venta de la persona.
($\forall p$: persona)(p ∈ claves(e.ventasSinDesc) $\Rightarrow_{\text{L}}$
(($\forall i$: item)(definido?(i,obtener(p,e.ventasSinDesc)) $\Rightarrow_{\text{L}}$
(($\forall itL$: itBi(secu(<item,cant>)))(está?(itL,obtener(i,obtener(p,e.ventasSinDesc))) $\Rightarrow_{\text{L}}$
(¬definido?(i,e.descuentos) $\vee_L$ obtener(i,e.descuentos)[π_2(siguiente(itL))]=0) ∧ (π_1(siguiente(itL)=i)))))))

Idem anterior. Deben hacer primero el Y y después el implica

(9) ≡ Cada iterador de la venta de una persona que sea sin descuento, esta asociado unívocamente a una venta de la persona.
($\forall p$: persona)(p ∈ claves(e.ventasSinDesc) $\Rightarrow_{\text{L}}$
(($\forall i$: item)(definido?(i,obtener(p,e.ventasSinDesc)) $\Rightarrow_{\text{L}}$
(($\forall itL1, itL2$: itBi(secu(<item,cant>)))
(está?(itL1,obtener(i,obtener(p,e.ventasSinDesc))) ∧ (está?(itL2,obtener(i,obtener(p,e.ventasSinDesc))) ∧
itL1 ≠ itL2) $\Rightarrow_{\text{L}}$
((∃ i,j:nat)((0 ≤ i,j < long(obtener(p,e.ventas)) ∧ i ≠ j)$\wedge_{\text{L}}$
siguiente(itL1)=obtener(p,e.ventas)[i] ∧ siguiente(itL2)=obtener(p,e.ventas)[j])))))))

Bien la idea, nuevamente ojo con los dobles implica

gastoPuestoPersona : dicc(item × secu(⟨cant × bool⟩)) × dicc(item × precio) × dicc(item × ad(nat)) ⟶ dinero
gastoPuestoPersona(vp,m,d) ≡ **if** vacia?(vp) **then**
0
else
aplicarDescuento(obtener(π_1(prim(vp))),e.descuentos[π_2(prim(vp))]) × π_2(vp) +
gastoPuestoPersona(fin(vp),m,d)
fi

cantVendidaPuesto : conj(persona) × item ⟶ cant
cantVendidaPuesto(cp,i) ≡ **if** ∅?(cp) **then**
0
else
cantVendidaPersona(obtener(dameUno(cp),e.ventas),i) +
cantVendidaPuesto(sinUno(cp),i)
fi

cantVendidaPersona : secu(⟨item × cant⟩) × item ⟶ cant
cantVendidaPersona(vp,i) ≡ **if** vacia?(vp) **then**
0
else
if π_1(prim(vp))= i **then**
π_2(prim(vp))+cantVendidaPersona(fin(vp),i)
else
cantVendidaPersona(fin(vp),i)
fi
fi

Abs : estr e ⟶ puesto {Rep(e)}
Abs(e) ≡ p: puesto /
menu(p) =$_{\text{obs}}$ claves(e.menu) ∧
stock(p) =$_{\text{obs}}$ claves(e.stock) ∧$_{\text{L}}$
($\forall i$: item)(i ∈ menu(p) ⇒ precio(p, i) =$_{\text{obs}}$ obtener(e.menu, i) ∧ stock(p, i) =$_{\text{obs}}$ obtener(e.stock, i)) ∧
($\forall i$: item)(def?(i,e.descuentos) $\Rightarrow_{\text{L}}$(($\forall c$: cant)(0 ≤ c < long(obtener(i,e.descuentos)) $\Rightarrow_{\text{L}}$descuento(p, i, c)
=$_{\text{obs}}$ obtener(i,e.descuentos)[c]))) ∧
($\forall i$: item)(¬ def?(i,e.descuentos) $\Rightarrow_{\text{L}}$(($\forall c$: cant)(descuento(p, i, c) =$_{\text{obs}}$ 0)))∧
($\forall a$: persona)(a ∈ claves(e.ventas) $\Rightarrow_{\text{L}}$ ventas(p, a) =$_{\text{obs}}$ listaAMulticonj(obtener(e.ventas, a)))

Debe ser un implica. Porque sino es falso. Todo ítem no pertenece al menú

2.4. Algoritmos

Algoritmos

iCrearPuesto(**in** m: `dicc(item,nat)`, **in** s: `dicc(item,nat)`, **in** d: `dicc(item,dicc(cant,nat))`) $\rightarrow$ res : puesto

```
1:  itD ← CrearIt(d)                                                          ▷ O(1)
2:  descuentos ← Vacio()                                                      ▷ O(1)
3:  while HaySiguiente(itD) do                                                ▷ O(I)
4:      stockInicial ← Significado(s, SiguienteClave(itD))                    ▷ O(log(I))
5:      arr ← arreglo_dimensionable[1...stockInicial]                         ▷ O(stockInicial)
6:      Definir(descuentos, SiguienteClave(itD), arr)                         ▷ O(log(I))
7:      c ← Significado(d, SiguienteClave(itD))                               ▷ O(log(I))
8:      itC ← CrearIt(c)                                                      ▷ O(1)
9:      i ← 1                                                                 ▷ O(1)
10:     minDescuento ← SiguienteSignificado(itC)                              ▷ O(1)
11:     while i ≤ stockInicial do                                             ▷ O(stockInicial)
12:         if i < minDescuento then                                          ▷ O(1)
13:             arr[i] ← 0                                                    ▷ O(1)
14:         else
15:             if Definido?(c, i) then                                       ▷ O(log(cant))
16:                 arr[i] ← Significado(c, i)                                ▷ O(log(cant))
17:                 ultI ← i                                                  ▷ O(1)
18:             else
19:                 arr[i] ← Significado(c, ultI)                             ▷ O(log(cant))
20:             end if
21:         end if
22:         i ← i + 1                                                         ▷ O(1)
23:     end while
24:     Avanzar(itD)                                                          ▷ O(1)
25: end while
26: res ← ⟨ m, s, Vacio(), descuentos, Vacio(), Vacio() ⟩                     ▷ O(1)
```

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

Justificación: Los iteradores de los diccionarios recorren inorder.

I*log(I + stockInicial) + I * (stock Inicial * log(cant))

Stock inicial * log(cant) -- OJO

---

iVender(**in** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`)

```
1:  if Definido?(p.gastosDe, a) then
2:      nuevoGasto ← Significado(p.gastosDe, a) + Precio(p, i) × (1 - (Descuento(p, i, c) / 100))   ▷ O(log(A) + log(I))
3:  else
4:      nuevoGasto = Precio(p, i) × (1 - (Descuento(p, i, c) / 100))          ▷ O(log(I))
5:  end if
6:  Definir(p.gastosDe, a, nuevoGasto)                                        ▷ O(log(A))
7:  ventas ← Significado(p.ventas, a)                                         ▷ O(log(A))
8:  it ← AgregarAtras(ventas, <i, c>)                                         ▷ O(1)
9:  if Descuento(p, i, c) = 0 then                                            ▷ O(log(I))
10:     ventasSinDescuento ← Significado(Significado(p.ventasSinDesc, a), i)  ▷ O(log(A) + log(I))
11:     AgregarAtras(ventasSinDescuento, it)                                  ▷ O(1)
12: end if
```

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

iMenu(**in** p : `puesto`) $\rightarrow$ res : conj(item)

```
1: it ← CrearIt(p.menu)                                   ▷ O(1)
2: conjItems ← Vacio()                                    ▷ O(1)
3: while HaySiguiente(it) do                              ▷ O(I)
4:     AgregarRapido(conjItems, Siguiente(it))            ▷ O(1)
5:     Avanzar(it)                                        ▷ O(1)
6: end while
7: res ← conjItems                                        ▷ O(1)
```

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

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

```
1: res ← Significado(p.stock, i)                          ▷ O(log(I))
```

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

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

```
1: res ← Significado(p.menu, i)                           ▷ O(log(I))
```

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

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

```
1: if Definido?(p.descuentos, i) then                     ▷ O(log(I))
2:     res ← Significado(p.descuentos, i)[c]              ▷ O(log(I))
3: else
4:     res ← 0                                            ▷ O(1)
5: end if
```

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

iGastosDe(**in** p : `puesto`, **in** a : `persona`) $\rightarrow$ res : nat

```
1: res ← Significado(p.gastosDe, a)                       ▷ O(log(A))
```

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

3. Módulo Cola de Prioridad Acotada($\langle\alpha, \beta\rangle$)

El módulo Cola de Prioridad Acotada ($\langle\alpha, \beta\rangle$) proporciona una estructura de datos que permite almacenar elementos y acceder a ellos según su prioridad. Esta implementación utiliza un vector para representar el heap, una estructura de árbol binario completa.

En esta implementación, los elementos se almacenan en el vector heap de acuerdo con su prioridad, de manera que el elemento de mayor prioridad se encuentra en la posición raíz del árbol binario. La propiedad del heap asegura que para cada nodo, el valor de su padre es mayor o igual que los valores de sus hijos.

3.1. TAD Cola de Prioridad Acotada(γ)

TAD Cola de prioridad acotada(γ)

igualdad observacional

$$(\forall c, c' : \text{colaPriorA}(\gamma)) \left(c =_{\text{obs}} c' \iff \begin{pmatrix} \text{cota(c)} =_{\text{obs}} \text{cota(c')} \wedge \text{vacía?}(c) =_{\text{obs}} \text{vacía?}(c') \wedge_{\text{L}} \\ (\neg\text{vacía?}(c) \Rightarrow_{\text{L}} (\text{próximo}(c) =_{\text{obs}} \text{próximo}(c') \wedge \\ \qquad \text{desencolar}(c) =_{\text{obs}} \text{desencolar}(c')) \end{pmatrix} \right)$$

parámetros formales

géneros γ

operaciones $\bullet < \bullet : \gamma \times \gamma \longrightarrow$ bool — Relación de orden total estricto[1]

géneros colaPriorA(γ)

exporta colaPriorA(γ), generadores, observadores

usa Bool

observadores básicos

cota	: colaPriorA(γ)	$\longrightarrow$ nat	
vacía?	: colaPriorA(γ)	$\longrightarrow$ bool	
próximo	: colaPriorA(γ) c	$\longrightarrow$ γ	$\{\neg$ vacía?(c)$\}$
desencolar	: colaPriorA(γ) c	$\longrightarrow$ colaPriorA(γ)	$\{\neg$ vacía?(c)$\}$

generadores

vacía	: nat *cota*	$\longrightarrow$ colaPriorA(γ)	$\{cota > 0\}$
encolar	: γ e $\times$ colaPriorA(γ) c	$\longrightarrow$ colaPriorA(γ)	$\{$respetaCota(c, e)$\}$

otras operaciones

longitud	: colaPriorA(γ)	$\longrightarrow$ nat

axiomas $\forall c$: colaPriorA(γ), $\forall e$: γ, $\forall n$: nat

cota(vacía(n))	$\equiv$	n
cota(encolar(e, c))	$\equiv$	cota(c)
vacía?(vacía(n))	$\equiv$	true
vacía?(encolar(e, c))	$\equiv$	false
próximo(encolar(e, c))	$\equiv$	**if** vacía?(c) $\vee_{\text{L}}$ proximo(c) $< e$ **then** e **else** próximo(c) **fi**
desencolar(encolar(e, c))	$\equiv$	**if** vacía?(c) $\vee_{\text{L}}$ proximo(c) $< e$ **then** c **else** encolar(e, desencolar(c)) **fi**
longitud(c)	$\equiv$	**if** vacía?(c) **then** 0 **else** 1 + longitud(desencolar(c)) **fi**

Fin TAD

[1] Una relación es un orden total estricto cuando se cumple:

Antirreflexividad: $\neg\, a < a$ para todo $a : \gamma$

Antisimetría: $(a < b \;\Rightarrow\; \neg\, b < a)$ para todo $a, b : \gamma$, $a \neq b$

Transitividad: $((a < b \wedge b < c) \;\Rightarrow\; a < c)$ para todo $a, b, c : \gamma$

Totalidad: $(a < b \vee b < a)$ para todo $a, b : \gamma$

3.2. Interfaz

Interfaz

parámetros formales

géneros α

función $\bullet < \bullet$(**in** e_1: $\langle \alpha, \beta \rangle$, **in** e_2: $\langle \alpha, \beta \rangle$) $\rightarrow$ *res***función** : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (e_1 < e_2)$}
Complejidad: $\mathcal{O}(e_1 < e_2)$
Descripción: relación de orden de $\langle \alpha, \beta \rangle$'s

COPIAR(**in** e: $\langle \alpha, \beta \rangle$) $\rightarrow$ res : $\langle \alpha, \beta \rangle$
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} e$}
Complejidad: $\mathcal{O}(copy(e))$
Descripción: función de copia de $\langle \alpha, \beta \rangle$'s

se explica con: COLA DE PRIORIDAD ACOTADA(α)

géneros: `colaPriorA(`$\langle \alpha, \beta \rangle$`)`.

Operaciones básicas de cola de prioridad Acotada

VACÍA() $\rightarrow$ res : `colaPriorA(`$\langle \alpha, \beta \rangle$`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia}
Complejidad: $\mathcal{O}(1)$
Descripción: genera una cola vacía.

ENCOLAR(**in/out** c: `colaPriorA(`$\langle \alpha, \beta \rangle$`)`, **in** e: (`(`$\langle \alpha, \beta \rangle$`)`) $\rightarrow$ res : `puntero(`α`)`
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ encolar(e, c_0)}
Complejidad:O(log(#c))
Descripción: Encola un elemento en la cola de prioridad. El elemento se agrega al final del vector que representa el heap y luego se restaura la propiedad del heap.

DESENCOLAR(**in/out** c: `colaPriorA(`$\langle \alpha, \beta \rangle$`)`)
Pre $\equiv$ {$c =_{\text{obs}} c_0 \wedge \neg$ vacia?(c)}
Post $\equiv$ {$c =_{\text{obs}}$ desencolar(e, c_0)}
Complejidad:O(log(#c))
Descripción: Elimina el elemento de mayor prioridad de la cola. El elemento de la raíz se intercambia con el último elemento en el vector,se elimina del vector y luego se restaura la propiedad del heap.

PRÓXIMO(**in** c: `colaPriorA(`$\langle \alpha, \beta \rangle$`)`) $\rightarrow$ res : α
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$res =_{\text{obs}}$ próximo?(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve el elemento de mayor prioridad en la cola.

VACÍA?(**in** c: `colaPriorA(`$\langle \alpha, \beta \rangle$`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia?(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve un booleano diciendo si la cola es o no vacía.

3.3. Representación

Representación

Representación de la cola de prioridad acotada

La cola de prioridad acotada se representa con un vector, que representa a un heap. La cota representa la cantidad de elementos maxima que puede tener la cola de prioridad.

`colaPriorA(`$\langle\alpha$`, `$\beta\rangle$`)` se representa con `estr`

donde `estr` es `tupla(`*heap*: `arreglo_dimensionable de` $\langle$`a: `α`,b: itDiccLog`$\rangle$, *indices*: `diccLog(`β`, nat)` , *longitud*: `nat` , *cota*: `nat`)

Rep : colaPriorA(α) $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
$(\forall i \in [0, \text{Longitud}(c) - 1],\quad \text{If } 2i + 1 < \text{Longitud}(c),\ \text{then } c[i] \geq c[2i + 1] \wedge$
$\forall i \in [0, \text{Longitud}(c) - 1],\quad \text{If } 2i + 2 < \text{Longitud}(c),\ \text{then } c[i] \geq c[2i + 2]) \wedge$
$(c.longitud \leq c.cota) \wedge$
$(\forall b : \beta)(Definido?(c.indices, b) \Rightarrow_{\text{L}} (\exists i : nat)(0 \leq i < c.longitud \wedge_{\text{L}} *c.heap[i].b = Significado(c.indices, b)) \wedge$
$(\forall t :< \alpha, \beta >)(\exists i : nat)(0 \leq i < c.longitud \wedge_{\text{L}} c.heap[i] = tupla) \Rightarrow_{\text{L}} (\exists b : \beta)(Definido?(c.indices, b) \wedge_{\text{L}} Significado(c.indices, b) = tupla.b))$

Abs : estr e $\longrightarrow$ colaPriorA($\langle\alpha, \beta\rangle$) {Rep($e$)}

Abs(e) $\equiv$ c: colaPriorA($\langle\alpha, \beta\rangle$) /
vacía?(c) = (e.longitud = 0) $\wedge$ ($\neg$ vacia?(c) $\Rightarrow$ próximo(c) = $\langle$c.heap[0].a, π_1(Actual(c.heap[0].b))$\rangle$ $\wedge$
desencolar(c) = arrayACola(e.heap, e.indices, e.longitud))

arrayACola : ad($\langle\alpha \times$ itDiccLog$\rangle$) $\times$ diccLog($\beta \times$ nat) $\times$ nat $\longrightarrow$ colaPriorA($\langle\alpha, \beta\rangle$)

arrayACola(a, d, l) $\equiv$ **if** $l = 0$ **then**
vacía
else
encolar($\langle\pi_1(a[l$ - 1]), π_2(Actual($\pi_2(a[l$ - 1]))))$\rangle$, arrayACola($a, d, l - 1$))
fi

3.4. Algoritmos

Algoritmos

•>•(in $e1$: ⟨a: α,b: β⟩)), in $e2$: ⟨a: α, b: β⟩) → res : bool

1: res ← (e1.a > e2.a ∨ (e1.a = e2.a ∧ e1.b > e2.b)) ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$

---

iVacía(in $cota$: nat) → res : colaPriorA(⟨α, β⟩)

1: res ← ⟨arreglo_dimensionable(0..cota - 1), Vacio(), 0, cota⟩ ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$

---

iEncolar(in/out c: colaPriorA(⟨α, β⟩), **in** e: ⟨alfa: α,beta: β⟩) → res : puntero(α)

1: **if** Definido?(c.indices, e.beta) **then**
2: i ← Significado(c.indices, e.beta) ▷ $\mathcal{O}(1)$
3: **if** e > c.heap[i] **then** ▷ $\mathcal{O}(1)$
4: index ← HeapifyUp(c, i) ▷ $\mathcal{O}(log(\#c))$
5: **else**
6: index ← HeapifyDown(c, i) ▷ $\mathcal{O}(log(\#c))$
7: **end if**
8: **else**
9: c.longitud ← c.longitud + 1 ▷ $\mathcal{O}(1)$
10: it ← Definir(c.indices, e.beta, c.longitud - 1) ▷ $\mathcal{O}(log(\#c))$
11: c.heap[c.longitud - 1] ← ⟨e.alfa, it⟩ ▷ $\mathcal{O}(1)$
12: index ← HeapifyUp(c, c.longitud - 1) ▷ $\mathcal{O}(1)$
13: **end if**
14: puntero ← &c.heap[index] ▷ $\mathcal{O}(1)$
15: res ← puntero ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(log(\#c) + copy(e))$
Justificación: En el peor caso, la cantidad de claves en el diccionario c.indices es igual a #c, donde #c es c.longitud.

---

iDesencolar(in/out c: colaPriorA(⟨α, β⟩))

1: c.heap[0] ← c.heap[c.longitud - 1] ▷ $\mathcal{O}(1)$
2: c.longitud ← c.longitud - 1 ▷ $\mathcal{O}(1)$
3: c.heap ← HeapifyDown(c.heap, 0) ▷ $\mathcal{O}(log(\#c))$
Complejidad: $\mathcal{O}(log(\#c))$

---

iPróximo(in c: colaPriorA(⟨α, β⟩)) → res : α

1: res ← ⟨c.heap[0].a, SiguienteClave(c.heap[0].b)⟩ ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$

---

iVacía?(in c: colaPriorA(⟨α, β⟩)) → res : $bool$

1: res ← (c.longitud = 0) ▷ $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$

3.5. Funciones auxiliares

Funcion privada: HeapifyUp

Descripción: Esta función se utiliza después de encolar un elemento en el heap y se encarga de mantener la propiedad del heap llamada 'orden del padre'. Compara el elemento en la posición index con su padre y, si el elemento es mayor, los intercambia. Luego, se repite este proceso de comparación e intercambio ascendiendo en el árbol hasta que el elemento se encuentra en la posición correcta.

Pre $\equiv$ {El índice debe estar dentro de los límites del vector heap.}

Post $\equiv$ {El elemento en la posición índice se ha colocado en la posición correcta de acuerdo con la propiedad del heap 'orden del padre'. Devuelve el índice de la posición correcta.}

---

iHeapifyUp(**in/out** c: `colaPriorA(`$\langle\alpha,\ \beta\rangle$`)`, **in** i: `nat`) $\rightarrow$ res : nat

```
1:  if i > 0 then
2:      indexPadre ← ⌊ (i - 1) / 2 ⌋                                   ▷ O(1)
3:      while c.heap[i] > c.heap[indexPadre] do                        ▷ O(log(#c))
4:          indexPadre ← ⌊ (i - 1) / 2 ⌋                               ▷ O(1)
5:          temp ← c.heap[i]                                           ▷ O(1)
6:          c.heap[i] ← c.heap[indexPadre]                             ▷ O(1)
7:          c.heap[indexPadre] ← temp                                  ▷ O(1)
8:          i ← indexPadre                                             ▷ O(1)
9:          SiguienteSignificado(c.heap[i].b) = i                      ▷ O(1)
10:         SiguienteSignificado(c.heap[indexPadre].b) = indexPadre    ▷ O(1)
11:     end while
12: end if
13: res ← i                                                            ▷ O(1)
```

Complejidad: $\mathcal{O}(log(\#c))$

Justificación: Donde $\#c$ es la logitud de c.heap.

Funcion privada: HeapifyDown

Descripción: Esta función se utiliza después de desencolar un elemento del heap y se encarga de mantener la propiedad del heap llamada 'orden del hijo'. Compara el elemento en la posición índice con sus hijos y, si alguno de los hijos es mayor, intercambia el elemento con el hijo de mayor valor. Luego, se repite este proceso de comparación e intercambio descendiendo en el árbol hasta que el elemento se encuentra en la posición correcta.

Pre ≡ {El índice debe estar dentro de los límites del vector heap.}

Post ≡ {El elemento en la posición índice se ha colocado en la posición correcta de acuerdo con la propiedad del heap 'orden del hijo'. Devuelve el índice de la posición correcta.}

---

iHeapifyDown(**in/out** c: `colaPriorA(`$\langle\alpha,\ \beta\rangle$`)`, **in** i: `nat`) → res : nat

```
1:  largo ← c.longitud                                                    ▷ O(1)
2:  while i < largo do                                                    ▷ O(log(#c))
3:      hijoIzq ← 2 × i + 1                                               ▷ O(1)
4:      hijoDer ← 2 × i + 2                                               ▷ O(1)
5:      máximo ← i                                                        ▷ O(1)
6:      if hijoIzq < largo ∧ c.heap[hijoIzq] > c.heap[máximo] then        ▷ O(1)
7:          SiguienteSignificado(c.heap[maximo]) = hijoIzq                ▷ O(1)
8:          SiguienteSignificado(c.heap[hijoIzq]) = maximo                ▷ O(1)
9:          máximo ← hijoIzq                                              ▷ O(1)
10:     else
11:         if hijoDer < largo ∧ c.heap[hijoDer] > c.heap[máximo] then    ▷ O(1)
12:             SiguienteSignificado(c.heap[maximo]) = hijoDer            ▷ O(1)
13:             SiguienteSignificado(c.heap[hijoDer]) = maximo            ▷ O(1)
14:             máximo ← hijoDer                                          ▷ O(1)
15:         end if
16:     end if
17:     if máximo != i then                                               ▷ O(1)
18:         temp ← c.heap[i]                                              ▷ O(1)
19:         c.heap[i] ← c.heap[máximo]                                    ▷ O(1)
20:         c.heap[máximo] ← temp                                         ▷ O(1)
21:     else
22:         break
23:     end if
24: end while
25: res ← máximo
```

Complejidad: $\mathcal{O}(log(\#c))$

Justificación: Donde $\#c$ es la logitud de c.heap.