Algoritmos y Estructuras de Datos II

Trabajo Práctico 2: Diseño

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

Lollapatuza

Integrante	LU	Correo electrónico
Begalli, Juan Martin	139/22	juanbegalli@gmail.com
Aguirre, Victoria Inés	626/19	viaguirre1001@gmail.com
Maurer, Milagros	42/22	milagros.maurer@gmail.com
Padilla Galindo, Eimy Laura	G41120814	al2193042903@azc.uam.mx

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo Lollapatuza

Interfaz

se explica con: LOLLAPATUZA.

usa: PUESTO DE COMIDA.

géneros: `lolla`

Operaciones básicas de Lollapatuza

CREARLOLLA(**in** ps: `dicc(puestoid, puesto)` , **in** as: `vector(persona)`) $\rightarrow$ res : `lolla`
Pre $\equiv$ $\{\neg vacia?(as) \wedge vendenAlMismoPrecio(significados(ps)) \wedge NoVendieronAun(significados(ps)) \wedge \neg\emptyset?(as) \wedge \neg\emptyset?(claves(ps))\}$
Post $\equiv$ $\{res =_{\text{obs}}$ crearLolla(ps, as)$\}$
Complejidad: $O(A \times log(A))$
Descripción: Genera un nuevo sistema de lollapatuza.

REGISTRARCOMPRA(**in/out** l: `lolla`, **in** pi: `puestoid`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`)
Pre $\equiv$ $\{l =_{\text{obs}} l_0 \wedge a \in personas(l) \wedge def?(pi, puestos(l)) \wedge_{\text{L}} haySuficientes?(obtener(pi, puestos(l)), i, c))\}$
Post $\equiv$ $\{l =_{\text{obs}} vender(l_0, pi, a, i, c)\}$
Complejidad: $O(log(A) + log(I) + log(P))$
Descripción: Registra una compra de una cantidad de un ítem, realizada por una persona en un puesto.

HACKEAR(**in/out** l: `lolla`, **in** a: `persona`, **in** i: `item`)
Pre $\equiv$ $\{l =_{\text{obs}} l_0 \wedge ConsumioSinPromoEnAlgunPuesto(l, a, i) \wedge a \in personas(l)\}$
Post $\equiv$ $\{l =_{\text{obs}} Hackear(l_0, a, i)\}$
Complejidad: $O(log(A)) + O(log(I)))$ y en caso de que el puesto correspondiente deje de ser hackeable para ese item y esa persona $O(log(A)) + O(log(I)) + O(log(P)))$
Descripción: Hackea un item consumido por una persona.

GASTOTOTAL(**in** l: `lolla`, **in** a: `persona`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {a $\in$ personas(l)}
Post $\equiv$ $\{res =_{\text{obs}}$ gastoTotal(l,a)$\}$
Complejidad: $O(log(A))$
Descripción: Devuelve el gasto total de una persona.

PERSONAQUEMASGASTO(**in** l: `lolla`) $\rightarrow$ res : `persona`
Pre $\equiv$ {$\neg$ $\emptyset$? (personas(l))}
Post $\equiv$ $\{res =_{\text{obs}}$ masGasto(l)$\}$
Complejidad: $O(1)$
Descripción: Devuelve la persona que más dinero gastó.

MENORSTOCK(**in** l: `lolla`, **in** i: `item`) $\rightarrow$ res : `puestoid`
Pre $\equiv$ {True}
Post $\equiv$ $\{res =_{\text{obs}}$ menorStock(l,i)$\}$
Complejidad: $O(log(I) \times P)$
Descripción: Devuelve el ID del puesto con el menor stock para un item dado.

OBTENERPERSONAS(**in** l: `lolla`) $\rightarrow$ res : `Vector(persona)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}}$ personas(l)$\}$
Complejidad: $O(1)$
Descripción: Devuelve las personas del sistema.
Aliasing: Devuelve una referencia no modificable

OBTENERPUESTOS(**in** l: `lolla`) $\rightarrow$ res : `dicc(puestoid,puesto)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}}$ puestos(l)$\}$
Complejidad: $O(1)$

Descripción: Devuelve los puestos, con sus ids, del sistema.
Aliasing: Devuelve una referencia no modificable

Representación

Lolla se representa con estr

donde estr es tupla(*personas*: Vector(Persona) , *puestos*: diccLog(puestoid,puesto)
, *mayorConsumidora*: persona
, *consumosPorPersona*: diccLog(persona,diccLog(item,cant))
, *puestosHackeables*: diccLog(persona,diccLog(item,diccLog(puestoid,puntero(puesto))))
, *precios*: diccLog(item,nat) , *gastosPorPersona*: maxHeap(<nat, itGastoPersona>) ,
personasEnGasto: diccLog(persona, indice))

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$
($\forall$ a: persona)($esta?(a, e.personas)$ $\iff$ a $\in$ claves(e.personasEnGasto))$\wedge$

claves(e.personasEnGasto) = claves(e.puestosHackeables) $\wedge$

($\forall$ a: persona)($a\in$ claves(e.consumosPorPersona) $\Rightarrow_L$ esta?(a, e.personas))$\wedge$

long(e.personas) = long(e.gastosPorPersona) $\wedge$

sinRepetidos(significados(e.personasEnGasto)) $\wedge$ sinRepetidos(significados(e.puestos)) $\wedge$

esta?(e.mayorConsumidora, e.personas) $\wedge$

($\forall$ j: indice)($j\in$ significados(e.personasEnGasto) $\Rightarrow_L$ $0 \leq$ j $<$long(e.personas)) $\wedge$

($\forall$ a: persona)($\forall$ i: item)(($def?(e.puestosHackeables, a)\wedge$ def?(obtener(e.puestosHackeables, a), i)) $\Rightarrow_L$ i $\in$ claves(e.precios)) $\wedge$

($\forall$ a: persona)($\forall$ i: item)(($def?(e.consumosPorPersona, a)\wedge$ def?(obtener(e.consumosPorPersona, a), i)) $\Rightarrow_L$ i $\in$ claves(e.precios)) $\wedge$

($\forall$ a: persona)($\forall$ i: item)(($def?(e.puestosHackeables, a)\wedge$ def?(obtener(e.puestosHackeables,a),i)) $\Rightarrow_L$ (claves(obtener(obtener(e.puestosHackeables,a),i)) $\subset$ claves(e.puestos))) $\wedge$

($\forall$ a: persona)($\forall$ i: item)($\forall$ pi: puestoid)(($def?(e.puestosHackeables, a)\wedge$ def?(obtener(e.puestosHackeables,a),i) $\wedge$ def?(obtener(obtener(e.puestosHackeables,a),i),pi)) $\Rightarrow_L$ *obtener(obtener(obtener(e.puestosHackables,a),i),pi) $=_{obs}$ obtener(e.puestos,pi)) $\wedge$

($\forall$ t_1: tupla)($\forall$ t_2: tupla)($esta?$(t_1, e.gastosPorPersona) $\wedge$ esta?(t_2, e.gastosPorPersona)$\wedge t_1 \neq t_2 \Rightarrow_L \pi_2(t_1) \neq \pi_2(t_2)$)$\wedge$

($\forall$ t: tupla)($esta?(t, e.gastosPorPersona) \Rightarrow_L siguiente(\pi_2(t))$ $\in$ claves(e.personasEnGasto))$\wedge$

($\forall$ a: persona)($a\in$ claves(e.personasEnGasto) $\Rightarrow_L$ (($\exists$ it: itGastoPersona)($\exists$ gasto:nat)(esta?($<$gasto, it$>$, e.gastosPorPersona) $\wedge$ Siguiente(it) $=_{obs}$ a) $\wedge$ $e.gastosPorPersona[obtener(a, e.personasEnGasto)]$ $=_{obs}$ $<$gasto,it$>$))

Abs : estr e $\longrightarrow$ lolla l {Rep(e)}

Abs(e) $\equiv$ puestos(l) $=_{\text{obs}}$ e.puestos $\wedge$
(∀ p: persona)(p ∈ personas(l) $\Rightarrow_{\text{L}}$ (∃ i:nat)(0 ≤ i < |e.personas| $\wedge_{\text{L}}$ e.personas[i] $=_{\text{obs}}$ p))

Algoritmos

iCrearLolla(**in** ps: `diccLog(puestoId, puesto)`, **in** as: `Vector(persona)`) → $res : estr$

1: $res.personas \leftarrow as$
2: $res.puestos \leftarrow ps$
3: $res.mayorConsumidora \leftarrow as[0]$ ▷ pongo una persona cualquiera
4: $res.consumosPorPersona \leftarrow Vacio()$

5: $unPuesto \leftarrow SiguienteSignificado(itDiccLog)$ ▷ como los precios son los mismos no me importa el puesto
6: $res.precios \leftarrow unPuesto.precios$
7: $res.personasPorGasto \leftarrow Vacio()$
8: $res.GastoPorPersona \leftarrow maxHeapVacio()$
9: $j \leftarrow 0$
10: **while** j<long(as) **do** ▷ $O(A \times log(A))$
11: Definir(res.puestosHackeables,as[j],Vacio()) ▷ defino a todas las personas en puestos hackeables
12: $itPersona \leftarrow Definir(res.personasPorGasto, as[j], j)$ ▷ defino a todas las personas en personasPorGasto junto con el indice
13: $encolar(res.GastoPorPersona, < 0, itPersona >)$ ▷ armo el maxHeap
14: j++
15: **end while**

Complejidad: $O(A \times log(A))$
Justificacón:Hasta la linea 10 son todas operaciones con un costo de $\Theta(1)$. El while se va a ejecutar A veces, siendo A la cantidad de personas. Dentro del while voy definir a todas las personas en *puestosHackeables* ($O(log(A))$), a definir a todas las personas en *personasPorGasto* ($O(log(A))$) y a encolar las tuplas en *gastoPorPersona* ($O(log(A))$). Por lo que queda una complejidad de $O(A \times log(A))$.

iRegistrarCompra(**in/out** l: `estr`, **in** pi: `puestoid`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`)

```
if ¬ Definido?(l.consumosPorPersona,a) then                         ▷ actualizo consumos //O(log(A) + log(I))
    Definir(l.consumosPorPersona,a,Definir(Significado(l.consumosPorPersona,a),i,c))
else
    if ¬ Definido?(Significado(l.consumosPorPersona,a),i) then
        Definir(Significado(l.consumosPorPersona,a),i,c)
    else
        cantAnterior ← Significado(Significado(l.consumosPorPersona,a),i)
        Definir(Significado(l.consumosPorPersona,a),i,cantAnterior+c)
    end if
end if

puesto* puestoDeVenta
puestoDeVenta ← & Significado(l.puestos, pi)                         ▷ O(log(P)) Creamos un puntero al puesto
registrarVenta(*puestoDeVenta,a,i,c)                                 ▷ O(log(P) + log(A) + log(I))

gasto ← Significado(l.precios, i)×c                                  ▷ O(log(I))

l.gastoPorPersona[Significado(l.personasEnGasto, a)].first ←
l.gastoPorPersona[Significado(l.personasEnGasto, a)].first + aplicarDescuento(gasto, ObtenerDescuento(*puestoDeVenta, i, c))

heapify(l.gastosPorPersona, Significado(l.personasEnGasto, a))       ▷ O(log(A))

l.mayorConsumidor ← SiguienteClave(proximo(l.gastosPorPersona).second)   ▷ actualizo mayor consumidor

if obtenerDescuento(*puestoDeVenta, i, c) == 0 then                  ▷ O(log(A) + log(I) + log(P)))
    if Definido?(Significado(l.puestosHackeables, a), i) then        ▷ actualizo puestos hackeables
        if ¬Definido?(Significado(Significado(l.puestosHackeables, a),i), pi) then
            Definir(Significado(Significado(l.puestosHackeables, a),i),pi,puestoDeVenta)
        end if
    else
        Definir(Significado(l.puestosHackeables, a), i, Definir(Vacio(), pi, puestoDeVenta))
    end if
end if
```

Complejidad: $O(log(A) + log(I) + log(P))$
Justificación: Hago muchas operaciones de definir, buscar significado y chequear si una clave esta definida en diccionarios logaritmicos. En estos diccionarios las claves son iguales a A, I o P, por lo que sumo las complejidades log(A)+log(I)+log(P). Las operaciones auxiliares utilizadas tambien tienen alguna de esas complejidades. Las operaciones con punteros son O(1).

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

cantidadNueva ← Significado(Significado(l.consumosPorPersona, a), i) - 1
Definir(Significado(l.consumosPorPersona, a), i, cantidadNueva) ▷ actualizo consumos

l.gastoPorPersona[Significado(l.personasEnGasto, a)].first − = Significado(l.precios,i)
heapify(l.gastosPorPersona, Significado(l.personasEnGasto, a)) ▷ actualizo gastos//$O(log(A))$
l.mayorConsumidora ← SiguienteClave(proximo(l.gastosPorPersona).second) ▷ actualizo mayor consumidor

itPuesto ← CrearIt(Significado(Significado(l.puestosHackeables, a), i))
puestoAHackear ← SiguienteSignificado(itPuesto) ▷ Como recorro inorder, tomo el minimo id
hackeoPuesto(*puestoAHackear, a, i) ▷ $O(log(A) + log(I))$

if obtenerCantidadVendidaSinDesc(*puestoAHackear, a, i) == 0 **then** ▷ veo si el puesto dejó de ser hackeable
    borrar(Significado(Significado(l.puestosHackeables, a), i), SiguienteClave(itPuesto))
end if

Complejidad:$O(log(A)) + O(log(I)))$ y en caso de que el puesto correspondiente deje de ser hackeable para ese item y esa persona $O(log(A)) + O(log(I)) + O(log(P)))$
Justificación: Actualizar los consumos nos toma log(A)+log(I) ya que busca los significados en los diccionarios logarítmicos donde las claves son las personas y los items. Lo mismo pasa con actualizar los gastos y actualizar al mayor consumidor, accede a estos datos con el mismo costo. En el caso de que el puesto deje de ser hackeable, se necesitará borrar el puesto de *puestosHackeables*, por lo tanto la complejidad será log(A) + log(I) + log(P).

---

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

indicePers ← Significado(l.personasEnGasto, a) ▷ $O(log(A))$
res ← (l.gastosPorPersona[indicePers]).first ▷ $O(1)$

Complejidad: $O(log(A))$
Justificacion: Acceso al significado de la persona en el diccionario logaritmico de la estructura de representacion.

---

iPersonaQueMasGasto(**in** l : `estr`) → res : $persona$

res ← l.mayorConsumidora

Complejidad: $O(1)$
Justificación: Es un acceso a la tupla de estructura de representación por lo tanto es constante siendo $O(1)$.

---

iObtenerPersonas(**in** l : `estr`) → res : $vector(persona)$

res ← l.personas

Complejidad: $O(1)$
Justificación: Es un acceso a la tupla de estructura de representación, por lo tanto es constante siendo $O(1)$

---

iObtenerPuestos(**in** l : `estr`) → res : $dicc(puestoid, puesto)$

res ← l.puestos

Complejidad: $O(1)$
Justificación: Es un acceso a la tupla de estructura de representación por lo tanto es constante siendo $O(1)$.

iMenorStock(**in** l : estr,**in** i : item) $\to$ res : $puestoid$

```
itPuesto ← CrearIt(l.puestos)
while HaySiguiente(itPuesto) do
    if estaEnElMenu?(SiguienteSignificado(itPuesto),i) then
        res ← SiguienteClave(itPuesto)
        break                                              ▷ Hago un break del ciclo
    end if
    Avanzar(itPuesto)
end while

while HaySiguiente(itPuesto) do
    if estaEnElMenu?(SiguienteSignificado(itPuesto),i) then
        if obtenerStock(SiguienteSignificado(itPuesto), i) < ObtenerStock(res, i) then
            res ← SiguienteClave(itPuesto)
        else
            if obtenerStock(SiguienteSignificado(itPuesto), i) = ObtenerStock(res,i) &&
res > SiguienteClave(itPuesto) then
                res ← SiguienteClave(itPuesto)
            end if
        end if
    end if
    Avanzar(itPuesto)
end while
```

Complejidad: $O(log(I) \times P)$
Justificación:Dentro de ambos ciclos se recorren puestos de comida del festival, que nos lleva $O(P)$ y luego, dentro de esos ciclos usamos operaciones como obtenerStock, Definido? que nos lleva $O(log(I))$. Por lo tanto tenemos una complejidad de $O(log(I) \times P)$. En el caso de tener stock, se desempata por el mayor ID.

2. Módulo Puesto de Comida

Interfaz

se explica con: Puesto de Comida, Dinero

usa: Puesto de Comida, MaxHeap

géneros: puesto

Operaciones básicas de Puesto de Comida

CrearPuesto(**in** p : diccLog(item,nat),**in** s : diccLog(item,nat),**in** d : diccLog(item,diccLog(cant,desc)))
$\to$ res : puesto
Pre $\equiv$ $\{\neg\emptyset(claves(p)) \land claves(p) = claves(s) \land claves(d) \subseteq claves(p)\}$
Post $\equiv$ $\{res =_{\text{obs}}$ CrearPuesto(p,s,d)$\}$
Complejidad: $O(log(I) \times I \times m \times cant \times d)$
Descripción: Genera un nuevo puesto de comida.

obtenerStock(**in** p : puesto, **in** i : item) $\to$ res : cant
Pre $\equiv$ $\{i \in menu(p)\}$
Post $\equiv$ $\{res =_{\text{obs}}$ stock(p,i)$\}$
Complejidad: $O(log(I))$
Descripción: Devuelve el stock del item en el puesto p.
Aliasing: Devuelve una referencia modificable del stock

estaEnMenu?(**in** p: `puesto`,**in** i: `item`) $\to$ res : `bool`
Pre $\equiv$ $\{True\}$
Post $\equiv$ {Devuelve true si y solo si el item esta en el menu del puesto. En este caso, el menu serian los items del stock.}
Complejidad: $O(log(I))$

obtenerDescuento(**in** p: `puesto`, **in** i: `item`, **in** c: `cant`) $\to$ res : `desc`
Pre $\equiv$ $\{i \in menu(p)\}$
Post $\equiv$ $\{res =_{\text{obs}}$ descuento(p,i,c)$\}$
Complejidad: $O(log(I))$
Descripción: Devuelve el descuento de un item dada la cantidad del mismo.
Aliasing: Devuelve una referencia no modificable al descuento

aplicarDescuento(**in/out** n: `dinero`, **in** d: `desc`)
Pre $\equiv$ $\{n = n_0 \land d < 100\}$
Post $\equiv$ $\{n =_{\text{obs}}$ aplicarDescuento(n_0, d)$\}$
Complejidad: $O(1)$
Descripción: Devuelve el dinero con el descuento aplicado.

obtenerGasto(**in** p: `puesto`, **in** a: `persona`) $\to$ res : `dinero`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}}$ gastosDe(p,a)$\}$
Complejidad: $O(log(A))$
Descripción: Devuelve el gasto de una persona en el puesto p.
Aliasing: Devuelve una referencia modificable del gasto de una persona en determinado puesto

obtenerVentas(**in** p: `puesto`,**in** a: `persona`) $\to$ res : `diccLog(item,tupla<cantConDesc : cant, cantSinDesc: cant>)`
Pre $\equiv$ {true}
Post $\equiv$ $\{multVentas(res) =_{\text{obs}}$ ventas(p,a) $\}$
Complejidad: $O(log(A))$
Descripción: Devuelve las ventas de una persona en un puesto
Aliasing: Devuelve una referencia a las ventas de una persona en un puesto determinado

obtenerCantidadVendidaSinDesc(**in** p: `puesto`, **in** a: `persona`, **in** i: `item`) $\to$ res : `cant`
Pre $\equiv$ {i $\in$ menu(p) $\land$ ($\exists$ c: cant)(<i,c>$\in$ ventas(p,a))}
Post $\equiv$ {El resultado es igual a la suma de las cantidades vendidas sin descuento}
Complejidad: $O(log(I) + log(A))$
Descripción: Devuelve la cantidad vendida de un item por el puesto p a una determinada persona.
Aliasing: Devuelve una referencia modificable de la cantidad vendida sin descuento de un item por el puesto p a una determinada persona

registrarVenta(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`)
Pre $\equiv$ $\{p =_{\text{obs}} p_0 \land haySuficiente?(p, i, c)\}$
Post $\equiv$ $\{p =_{\text{obs}} vender(p_0, a, i, c)\}$
Complejidad: $O(log(I) + log(A))$
Descripción: Registra una venta de una cantidad de un item, realizada por un puesto a una persona determinada

hackeoPuesto(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`)
Pre $\equiv$ $\{p =_{\text{obs}} p_0 \land i \in menu(p) \land consumioSinPromo?(p, a, i)\}$
Post $\equiv$ $\{p =_{\text{obs}} olvidarItem(p_0, a, i)\}$
Complejidad: $O(log(A) + log(I))$

Descripción: Registra un hackeo de un item comprado sin descuento por una persona en un puesto

Representación

puesto **se representa con** estr

donde estr es tupla(*stock*: diccLog(item,nat), *vectorDescuentos*: DiccLog(item,Vector(desc))
, *gastoPorPersona*: diccLog(persona,nat), *precios*: diccLog(item,nat)
, *ventas*: diccLog(persona,diccLog(item,tupla<cantConDesc : cant, cantSinDesc: cant>)))

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
claves(e.stock) = claves(e.vectorDescuentos) = claves(e.precios) $\wedge$

claves(e.gastoPorPersona)=claves(e.ventas) $\wedge$

($\forall$ a: persona)($def?(e.ventas, a) \Rightarrow_L claves(obtener(e.ventas, a)) \subset$ claves(e.stock)) $\wedge$

($\forall$ a: persona)($def?(e.ventas, a) \Rightarrow_L obtener(e.gastoPorPersona, a) = calcularGasto(obtener(e.ventas, a), e)) \wedge$

($\forall$ i: item)($def?(e.vectorDescuentos, i) \Rightarrow_L$($\forall$ n: nat)(0$\leq$ n $<$ Longitud(obtener(i,e.vectorDescuentos)) $\Rightarrow_L$
0 $\leq$ obtener(i,e.vectorDescuentos)[n]$<$100)$\wedge$

((Longitud(obtener(i,e.vectorDescuentos)) == 0) $\vee$
(2 $\leq$ Longitud(obtener(i,e.vectorDescuentos)) $\leq$ obtener(e.stock,i)))

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

Abs(e) $\equiv$ ($\forall$ i: *item*)($i\in$ menu(p) $\Rightarrow_L$ i $\in$ claves(e.stock)) $\wedge$
($\forall$ i: *item*)($i\in$ menu(p) $\Rightarrow_L$ (precio(p,i) $=_{obs}$ obtener(e.precios,i))
$\wedge$ (stock(p,i) $=_{obs}$ obtener(e.stock,i))
$\wedge$ (($\forall$ c: *cant*)(0$\leq$ c $<$|obtener(e.vectorDescuentos, i)| $\Rightarrow_L$ descuento(p,i,c) $=_{obs}$ obtener(e.vectorDescuentos, i)[c]) $\vee$ (|obtener(e.vectorDescuentos, i)| $\leq$ c $\Rightarrow_L$ descuento(p, i, c) $=_{obs}$ obtener(e.vectorDescuentos, i)[|obtener(e.vectorDescuentos, i)| - 1]))
$\wedge$ (($\forall$ a: *persona*)($ventas(p, a) =_{obs} multVentas(significado(e.ventas, a)))))$

Especificacion de las operaciones auxiliares utilizadas en la Representación

TAD Puesto de comida extendido

extiende Puesto de comida

otras operaciones

calcularGasto : DiccLog(item, tupla<nat,nat>) c $\times$ puesto p $\longrightarrow$ nat

multVentas : DiccLog(item, tupla<nat,nat>) c $\longrightarrow$ multiconj(<item,cant>)

axiomas

($\forall$ p: puesto,$\forall$ c: DiccLog(item, tupla<nat,nat>))

calcularGasto(c, p) $\equiv$ **if** $\emptyset?(claves(c))$ **then**
0
else
π_2(Obtener(c,DameUno(claves(c))) $\times$ precio(p, DameUno(claves(c)))) +
gastosDe1Venta(p,<DameUno(claves(c)),π_1(Obtener(c, DameUno(claves(c))))>) +
calcularGasto(SinUnaClave(c)), p))
fi

multVentas(c) $\equiv$ **if** $\emptyset?(claves(c)$ **then**
$\emptyset$
else
Ag(<DameUno(claves(c)), π_1(Obtener(c, DameUno(claves(c)))) + π_2(Obtener(c, DameUno(claves(c))))>, multVentas(SinUnaClave(c)))
fi

Fin TAD

Algoritmos

iCrearPuesto(**in** p: `diccLog(item,nat)`, **in** s: `diccLog(item,nat)`, **in** d: `diccLog(item,diccLog(cant,desc))`) $\rightarrow$ res : estr

```
1:  res.stock ← s
2:  res.precios ← p
3:  res.gastoPorPersona ← Vacio()
4:  res.ventas ← Vacio()
5:  itItems ← CrearIt(d)
6:  diccVectores ← Vacio()
7:  while HaySiguiente(itItems) do                          ▷ O(log(I) × I × m × cant × d)
8:      vectorDesc ← Vacio()
9:      itCants ← crearIt(SiguienteSignificado(itItems))    ▷ Voy a recorrer el diccLog(cant,desc), Inorder
10:     i ← 0
11:     while i <SiguienteClave(itCants) do
12:         vectorDesc[i] = 0                               ▷ No hay descuento
13:         i ← i + 1
14:     end while
15:     while HaySiguiente(itCants) do
16:         desc ← SiguienteSignificado(itCants)            ▷ Me guardo el descuento a insertar
17:         Avanzar(itCants)                                ▷ Avanzo para poner cota con la prox cantidad
18:         while i <SiguienteClave(itCants) do
19:             vectorDesc[i] ← desc
20:             i ← i + 1
21:         end while
22:         if ¬ HaySiguiente(itCant) then
23:             vectorDesc[i] ← SiguienteSignificado(itCant) ▷ Si llegue a la ultima cantidad
24:         end if
25:     end while
26:     Definir(diccVectores,SiguienteClave(itItems),vectorDesc)
27:     AvanzarIt(itItems)
28: end while
29: res.vectorDescuentos ← diccVectores
```

Complejidad: $O(log(I) \times I \times m \times cant \times d)$
Justificación: Hasta la línea 6 son todas operaciones $O(1)$. La complejidad principal está cuando llenamos diccVectores. El while de L7 se va a ejecutar I veces siendo I la cantidad de items que tienen descuento (en el peor de los casos son todos los items del festival). Y al final de cada iteracion defino un elemento lo cual me toma log(I). Luego, el while de L11 se ejecuta m veces, siendo m la minima cantidad con descuento. El while de L15 itera cant veces, siendo cant el numero de cantidades con descuentos (#claves). Y el while de L18 itera d veces, siendo d la cantidad a la que aplico descuento. Por ende me queda una complejidad de $O(log(I) \times I \times m \times cant \times d)$.

iObtenerStock(**in** p: `estr`, **in** i: `item`) → res : nat

```
res ← Significado(p.stock, i)
```

Complejidad: $O(log(I))$
Justificación: Tenemos una función de costo logarítmico ya que la búsqueda y obtención de un elemento en un diccionario logarítmico es $O(log(I))$.

iEstaEnElMenu?(**in** p: `estr`,**in** i: `item`) → res : *bool*

```
res ← Definido?(p.stock,i)
```

Complejidad: $O(log(I))$
Justificacion: Busco si esta definido el item en el diccLog de stock. Lo cual tiene costo logaritmico con respecto a la cantidad de claves.

iObtenerDescuento(**in** p: `estr`, **in** i: `item`, **in** c: `cant`) → res : *desc*

```
if ¬ Definido?(p.VectorDescuentos,i) then                ▷ O(log(I))
    res ← 0
else
    longDesc ← Longitud(Significado(p.VectorDescuentos, i))
    if c >Significado(p.VectorDescuentos, i)[longDesc-1] then
        res ← Significado(p.VectorDescuentos, i)[longDesc-1]
    else
        res ← Significado(p.VectorDescuentos, i)[c]
    end if
end if
```

Complejidad: $O(log(I))$
Justificación: Se usan operaciones de diccLog en donde la cantidad de claves es igual a I. Indexaciones son O(1). Nos queda O(log(I))

iAplicarDescuento(**in** p: `dinero`, **in** d: `desc`) → res : dinero

```
res ← div(p × (100-d),100)
```

Complejidad: $O(1)$
Justificación: Es una operacion aritmetica. Asumimos que div esta definido como en la especificacion.

iObtenerGasto(**in** p: `estr`, **in** a: `persona`) → res : *dinero*

```
res ← Significado(p.gastoPorPersona, a)
```

Complejidad: $O(log(A))$
Justificación: Es una busqueda de significado en un diccLog con cantidad de claves igual a A, entonces es O(log(A))

iObtenerVentas(**in** p: `estr`, **in** a: `persona`) $\to$ $res : diccLog(item, tupla < cantConDesc : cant, cantSinDesc : cant >)$

```
if Definido?(p.ventas,a) then
    res ← Significado(p.ventas, a)
else
    res ← Vacio()
end if
```

Complejidad: $O(log(A))$
Justificación: Es una busqueda de significado en un diccLog con a lo sumo A cantidad de claves, entonces es O(log(A)). Como vamos definiendo las personas a medida que compran, primero nos preguntamos si ya compro.

---

iObtenerCantidadVendidaSinDesc(**in** p: `estr`, **in** a: `persona`, **in** i: `item`) $\to$ $res : cant$

```
res ← Significado(Significado(p.ventas, a), i).cantSinDesc
```

Complejidad: $O(log(A) + log(I))$
Justificación: Buscamos el significado en diccLogs donde las claves son las personas y los items. Por lo tanto nos queda una complejidad de log(A)+log(I)

---

iRegistrarVenta(**in/out** p: `estr`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`)

```
nuevoStock ← Significado(p.stock, i) - c
Definir(p.stock, i, nuevoStock)                                  ▷ actualizo stock //O(log(I))

gastoVentaSinDesc ← Significado(p.precios, i) × c
gastoVentaConDesc ← aplicarDescuento(gastoVentaSinDesc, obtenerDescuento(p,i,c))
nuevoGastoTotalPersona ← Significado(p.gastoPorPersona, a) + gastoVentaConDesc
Definir(p.gastoPorPersona, a, nuevoGastoTotalPersona) ▷ actualizo el gasto de la persona //O(log(A) + log(I))

if Definido?(Significado(p.ventas, a), i) then          ▷ actualizo ventas //O(log(A) + log(I))
    cantAnterior ← Significado(Significado(p.ventas, a), i)
    if obtenerDescuento(p,i,c) == 0 then
        Definir(Significado(p.ventas,a), i, <cantAnterior.cantSinDesc + c>, cantAnterior.cantConDesc>)
    else
        Definir(Significado(p.ventas,a), i, <cantAnterior.cantSinDesc, cantAnterior.cantConDesc + c >)
    end if
else
    if obtenerDescuento(p,i,c) == 0 then
        Definir(Significado(p.ventas, a), i, <c,0>)
    else
        Definir(Significado(p.ventas, a), i, <0,c>)
    end if
```

Complejidad: $O(log(A) + log(I))$
Justificación: Son operaciones sobre diccLogs cuyas claves son las personas y los items del festival. Por eso la complejidad total nos queda de $O(log(A) + log(I))$.

iHackeoPuesto(**in/out** p: `estr`, **in** a: `persona`, **in** i: `item`)

```
Definir(p.stock, i, Significado(p.stock, i) + 1)                                  ▷ actualizo stock
gastoHackeado ← Significado(p.gastoPorPersona, a) - Significado(p.precios, i)
Definir(p.gastoPorPersona, a, gastoHackeado)                                      ▷ actualizo gasto
cantidadVendidaHackeada ← Significado(Significado(p.ventas, a), i).cantSinDesc - 1
cantidadVendidaSinHackear ← Significado(Significado(p.ventas, a), i).cantConDesc
Definir(Significado(p.ventas, a), i, <cantidadVendidaSinHackear ,cantidadVendidaHackeada>) ▷ actualizo ventas
```

Complejidad: $O(log(A) + log(I))$
Justificación: Todas las operaciones consisten en Definir o acceder al significado de un diccLog, cuyas claves son A o I. Con lo cual su complejidad termina siendo de $O(log(A) + log(I))$. Ignorar: =0

3. Módulo MaxHeap

Interfaz

parametros formales
géneros $\langle gasto, itGastoPersona\rangle$
función COPIAR(**in** t: $\langle gasto, itGastoPersona\rangle$) $\rightarrow$ res : $\langle gasto, itGastoPersona\rangle$
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} t\}$
Complejidad: $\Theta(copy(t))$
Descripción: función de copia de $\langle gasto, itGastoPersona\rangle$'s

se explica con: COLA DE PRIORIDAD

usa: COLA DE PRIORIDAD

géneros: MAXHEAP

Operaciones básicas de maxHeap

MAXHEAPVACIO() $\rightarrow$ res : `maxHeap`
Pre $\equiv$ {True}
Post $\equiv$ $\{res =_{\text{obs}}$ vacia()$\}$
Complejidad: $O(1)$
Descripción: Genera un nuevo maxHeap vacío.

HEAPIFY(**in/out** m: `maxHeap`,**in** i: `nat`)
Pre $\equiv$ $\{m =_{\text{obs}} m_0 \wedge \neg vacia?(m)\}$
Post $\equiv$ {el maxHeap sigue manteniendo su relacion de orden}
Complejidad: $O(log(A))$
Descripción: Se hace un max-Heapify del maxHeap, haciendo sift-up o sift-down segun corresponda.

PROXIMO(**in** m: `maxHeap`) $\rightarrow$ res : $\langle gasto, itGastoPersona\rangle$
Pre $\equiv$ {$\neg$ vacia?(m)}
Post $\equiv$ {alias($res =_{\text{obs}}$ proximo(m))}
Complejidad: $O(1)$
Descripción: Devuelve la raiz del maxHeap.
Aliasing: Devuelve una referencia modificable. El maxHeap de entrada tambien es una referencia modificable. De esa forma puedo usar y modificar las estructuras con el iterador.

ENCOLAR(**in/out** m: `maxHeap`,**in** t: $\langle gasto, itGastoPersona\rangle$)
Pre $\equiv$ $\{m =_{\text{obs}} m_0\}$

Post $\equiv$ {m se devuelve con el elemento t ya insertado y acomodado. Y long(m) = long(m_0)+1}
Complejidad: $O(log(A))$
Descripción: Se inserta t en m y se lo acomoda hasta que se cumpla el invariante del maxHeap

DESENCOLAR(**in/out** m : `maxHeap`)
Pre $\equiv$ {$\neg$ vacia?(m) $\wedge$ m = m_0}
Post $\equiv$ {m $=_{\text{obs}}$ desencolar(m_0)}
Complejidad: $O(log(A))$
Descripción: Quita la raiz del maxHeap y reacomoda la estructura

PADRE(**in** i : `nat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = floor(i/2) - 1 \vee res = floor(i/2)\}$
Complejidad: $O(1)$
Descripción: Dado un i hace el calculo correspondiente para ubicar al nodo padre en el maxHeap

Representación

`maxHeap` **se representa con** `estr`

donde `estr` es `tupla`(*maxHeap*: `Vector(tupla(gasto: nat, it: itGastoPersona))`)

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
($\forall$ i: nat)((0 $\leq$i $\leq$ $\frac{long(e.maxHeap)}{2}$ $- 1$)$\Rightarrow_{\text{L}}$ π_1(e.maxheap[i]) $\geq$ π_1(e.maxheap[2i+1]) $\wedge$
π_1(e.maxheap[i]) $\geq$ π_1(e.maxheap[2i+2]))

Abs : estr e $\longrightarrow$ colaPrior(α): c {Rep(e)}

Abs(e) $\equiv$ vacia?(c) $\Rightarrow_L$ long(e.maxHeap) = 0 $\wedge$
($\forall$ c: colPrior)($\neg$vacio?(c) $\Rightarrow_L$ proximo(c) $=_{obs}$ prim(e.maxHeap) $\wedge$
desencolar(c) $=_{obs}$ fin(e.maxHeap))

Algoritmos

iMaxHeapVacio() $\rightarrow$ res : $estr$

```
res ← Vacio()                                        ▷ creo un vector vacio
```

Complejidad: $O(1)$
Justificación: Costo de crear un vector vacío.

iHeapify(**in/out** m : `estr`,**in** i : `nat`)

```
izq ← 2 × i + 1
der ← 2 × i + 2
max ← i
if izq ≤ Longitud(m) && m[izq].first>m[i].first then
    max ← izq
else
    if izq ≤ Longitud(m) && m[izq].first==m[i].first
&& SiguienteClave(m[izq].second) >SiguienteClave(m[i].second) then
        max ← izq
    end if
end if
if der ≤ Longitud(m) && m[der].first>m[i].first then
    max ← der
else
    if der ≤ Longitud(m)&&m[der].first==m[i].first
&& SiguienteClave(m[der].second) >SiguienteClave(m[i].second) then
        max ← der
    end if
end if
if max ≠ i then
    a ← m[i].first
    m[i].first ← m[max].first
    m[max].first ← a                                 ▷ hago swap
    heapify(m,max)
end if
```

Complejidad: $O(log(A))$
Justificación: Visto en la teorica.

iProximo(**in** m : `estr`) $\rightarrow$ $res :< gasto, itGastoPersona >$

```
res ← m[0]
```

Complejidad: $O(1)$
Justificación: Costo de obtener la primer posicion del vector.

iEncolar(**in/out** m : `estr`,**in** t : `<gasto,itGastoPersona>`)

```
AgregarAtras(m,t)
n ← Longitud(m)-1
while n >0 && m[padre(n)].first <m[n].first do
    temp ← m[padre(n)].first
    m[padre(n)].first ← m[n].first
    m[n].first ← temp
    n ← padre(n)
end while
```

Complejidad: $O(log(A))$
Justificación: Visto en la teorica.

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

```
m[0] ← m[Longitud(m)-1]
heapify(m,0)
```

Complejidad: $O(log(A))$
Justificación: Visto en la teorica.

iPadre(**in** n : `nat`) $\rightarrow res : nat$

```
if n mod 2 = 0 then
    res ← floor(n/2)-1
else
    res ← floor(n/2)
end if
```

Complejidad: $O(1)$
Justificación: Son todas operaciones aritmeticas.

4. Otros TADS

El TAD Descuento es renombre de Nat con género *desc*.