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
Montaron, Josemaría	328/22	pepemontaron@gmail.com
Plotek, Magalí	1535/21	magaliplotek+qva@gmail.com
Mizrahi, Rafael	282/22	rafamizrahi30@gmail.com
Pivato, Santiago Ezequiel	426/22	santiagopivato@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega	Maximiliano Martino	Aprobado
Segunda entrega		

1. Introducción

✓El objetivo del trabajo consiste en realizar el diseño de los módulos *PuestodeComida* y *Lollapatuza*, con el fin de implementar el sistema del Lollapatuza. Los módulos están compuestos por una interfaz, donde se presentan las operaciones públicas para que el usuario las utilice; una estructura de representación, que viene acompañada de su invariante y función de abstracción, y los correspondientes algoritmos para las funciones. Para realizarlo, utilizamos la especificación de los tipos correspondientes y los módulos básicos presentados por la catedra. Además, diseñamos nuestro propio módulo: *ConjuntoLogarítmico*; para facilitar las operaciones de búsqueda y de agregar en complejidad logarítmica sin tener que recurrir al diccionario implementado en AVL.

2. Desarrollo

Módulo Puesto de Comida

El módulo puesto de comida provee una forma de representar el funcionamiento de un puesto donde se venden items a personas.

El módulo Puesto De Comida provee una tupla la cual utiliza otros módulos, principalmente diccionario logarítmico, para brindar acceso a la cantidad disponible de cada ítem, a su precio y al descuento que se aplica al comprar una cantidad exacta del mismo. Además, permite ver el gasto total de una persona en ese puesto, si es que realizó alguna compra, y llevar un registro de qué ítems y cuántos compró de cada uno, y si lo hizo con descuento.

Las complejidades se van a expresar en función de I, cant y A. Donde I es la cantidad de items del puesto, A es la cantidad de personas que hicieron alguna compra en el puesto, y cant es la cantidad de descuentos distintos que tiene un item.

Interfaz

se explica con: Puesto de Comida

géneros: `puesto`.

Operaciones básicas de puesto

NuevoPuesto(**in** p: `dicc(item, nat)`, **in** s: `dicc(item, nat)`, **in** d: `dicc(item, dicc(cant, nat))`) $\rightarrow$ res : `puesto`
✓**Pre** $\equiv$ {clavesCoinciden(p, s, d) $\wedge$ descuentosVálidos(d)}
✓**Post** $\equiv$ {$res =_{\text{obs}}$ crearPuesto(p, s, d)}
Complejidad: $O(I * (log(I) + cant^2))$
Descripción: Crea un nuevo puesto
Aliasing: Todos los tipos no primitivos de entrada se pasan por referencia y res devuelve una referencia modificable

ObtenerStock(**in** p: `puesto`, **in** i: `item`) $\rightarrow$ res : `cant`
✓**Pre** $\equiv$ {$i \in$ menu(p)}
✓**Post** $\equiv$ {$res =_{\text{obs}}$ stock(p, i)}
Complejidad: $O(\log I)$
Descripción: Devuelve un natural que representa la cantidad disponible del item dado.

ObtenerDescuento(**in** p: `puesto`, **in** i: `item`, **in** c: `cant`) $\rightarrow$ res : `nat`
✓**Pre** $\equiv$ {$i \in$ menu(p)}
✓**Post** $\equiv$ {$res =_{\text{obs}}$ descuentos(p, i, c)}
Complejidad: $O(\log I + \log cant)$
Descripción: Devuelve un natural, o un cero, que representa el descuento para el item y la cantidad dada

ObtenerGasto(**in** p: `puesto`, **in** a: `persona`) $\rightarrow$ res : `dinero`
✓**Pre** $\equiv$ {true}
✓**Post** $\equiv$ {$res =_{\text{obs}}$ ventas(p, a)}
Complejidad: $O(\log A)$, donde A es la cantidad de personas que hicieron alguna compra en el puesto
Descripción: Devuelve un natural que representa la cantidad de dinero que lleva gastado la persona en ese puesto

ObtenerPrecio(**in** p: `puesto`, **in** i: : item) $\rightarrow$ res : `dinero`
Pre $\equiv \{i \in \text{menu}(p)\}$
Post $\equiv \{res =_{\text{obs}} \text{precio(p, i)}\}$
Complejidad: $O(\log I)$
Descripción: Devuelve el precio para un item dado

Vender(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`)
Pre $\equiv \{p =_{\text{obs}} p_0 \wedge i \in \text{menu}(p) \wedge_L c \leq \text{stock}(p, i)\}$
Post $\equiv \{p =_{\text{obs}} \text{vender}(p_0, a, i, c)\}$
Complejidad: $O(\log I + \log A + \log cant)$
Descripción: Resta la cantidad al stock de un item, agrega la venta a los gastos de la persona y, el valor de la misma a la cantidad total de dinero que lleva gastado la persona

TieneDescuento?(**in** p: `puesto`, **in** i: `item`, **in** c: `cant`) $\rightarrow$ res : `bool`
Pre $\equiv \{i \in \text{menu}(p)\}$
Post $\equiv \{res =_{\text{obs}} true \iff \text{descuento}(p, i, c) \neq 0\}$
Complejidad: $O(\log I)$
Descripción: Devuelve true si y solo si ese item tiene un descuento, un natural distinto de cero, para esa cantidad, de lo contrario devuelve false

CalcularPrecio(**in** p: `puesto`, **in** i: `item`, **in** c: `cant`) $\rightarrow$ res : `dinero`
Pre $\equiv \{i \in \text{menu}(p) \wedge_L \text{c} \leq stock(\text{p,i})\}$
Post $\equiv \{res =_{\text{obs}} \text{aplicarDecuento}(\text{precio}(p, i, c), \text{descuento}(p, i, c))\}$
Complejidad: $O(\log I + \log cant)$
Descripción: Devuelve el precio de un item una vez aplicado el descuento para esa respectiva cantidad, si es que tiene

AnularCompra(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`) $\rightarrow$ res : `cant`
Pre $\equiv \{p =_{\text{obs}} p_0 \wedge i \in \text{menu}(p) \wedge_L \text{consumióSinPromo?}(p, a, i)\}$
Post $\equiv \{p =_{\text{obs}} \text{olvidarItem}(p_0, a, i) \wedge res =_{\text{obs}} \text{stock}(p, i) \text{ - stock}(p_0, i))\}$
Complejidad: $O(\log A + \log I)$
Descripción: Agrega al stock de un item la cantidad de la última compra hecha por la persona dada sin descuento, elimina esa misma compra de los gastos de la persona y resta el valor de la misma a la cantidad total de dinero que lleva gastado. También, devuelve la cantidad que fue borrada del item.

CompróItemSinDecuento?(**in** p: `puesto`, **in** a: `persona`, **in** i: `item`) $\rightarrow$ res : `bool`
Pre $\equiv \{i \in \text{menu}(p)\}$
Post $\equiv \{res =_{\text{obs}} \text{consumióSinPromo}(p, a, i)\}$
Complejidad: $O(\log A + \log I)$
Descripción: Devuelve true si la persona compro ese item sin descuento

Representación

Representación del puesto de comida

El objetivo de este módulo es implementar una instancia de Puesto De Comida. Para esto hicimos uso de una tupla, la cual contiene cinco diccionarios logaritmicos. El diccionario gastos devuelve para cada persona un diccionario de items que a su vez devuelve un arreglo de dos listas. En la primera lista del arreglo se registrarán las cantidades compradas sin descuento, mientras que en la segunda se registrarán aquellas compradas con descuento. Esta decisión nos permite simplificar la operación *hackear* perteneciente al módulo Lollapatuza.

Llamaremos diccLog al diccionario logarítmico para diferenciarlo del lineal.

`puesto` se representa con `pue`

donde `pue` es `tupla`(
stocks: `diccLog(item, stock)`,
precios: `diccLog(item, precio)`,
descuentos: `diccLog(item, vector(tupla(c: cant, d: nat)))`,
gastosTotales: `diccLog(persona, dinero)` ,
gastos: `diccLog(persona, diccLog(item, arreglo[2] de lista(cant)))`)

Rep : pue $\longrightarrow$ bool

✓ Rep(e) $\equiv$ true $\Longleftrightarrow$

(∀i: item)(def?(i, e.descuentos)$\rightarrow_L$ (∀t: tupla(cant, nat))(está?(t, obtener(i, e.descuentos)) $\rightarrow_L$ $\pi_2(t) \leq$ 100)) $\wedge_L$
claves(e.stocks) = claves(e.precios) $\wedge$ claves(e.descuentos) $\subseteq$ claves(e.stocks) $\wedge_L$
claves(e.gastosTotales) = claves(e.gastos) $\wedge_L$
(∀a:persona)(def?(a, e.gastos) $\rightarrow_L$ claves(obtener(a, e.gastos)) $\subseteq$ claves(e.stock)) $\wedge_L$
(∀a:persona)(def?(a, e.gastos) $\rightarrow_L$ (∀i: item) (def?(i, obtener(a,e.gastos)) $\rightarrow_L$ (∀c:cant)((está?(c, prim(obtener(i,obtener(a,e.gastos)))) $\rightarrow$ ((¬def?(i, e.descuentos)) $\vee$ (def?(i, e.descuentos) $\wedge_L$ (∀t:tupla(cant,nat))(está?(t, obtener(i, e.descuentos)) $\rightarrow$ π_1(t) >c)))) $\wedge$ (está?(c, ult(obtener(i,obtener(a,e.gastos))))) $\rightarrow$ def?(i, e.descuentos) $\wedge_L$ (∃t:tupla(cant,nat))(está?(t, obtener(i, e.descuentos)) $\wedge$ π_1(t) <c)))))) $\wedge_L$
(∀a:persona)(def?(a, e.gastosTotales) $\rightarrow_L$ obtener(a, e.gastosTotales) = ObtenerGastoTotal(multiconjuntoGastos(claves(obtener(a,e.gastos)), obtener(a, e.gastos)), e)) $\wedge$
(∀i: item)(def?(i, e.descuentos) $\rightarrow_L$ (∀j : nat)(0 $\leq$ j <long(obtener(i, e.descuentos)) - 1 $\rightarrow_L$ π_1(obtener(i, e.descuentos)[j]) <π_1(obtener(i, e.descuentos)[j+1])))

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

✓ Abs(e) $\equiv$ p: Puesto/ menú(p) $=_{obs}$ claves(e.precios) $\wedge_L$ (∀i: item)(i $\in$ menu(p) $\rightarrow_L$ (stock(p, i) $=_{obs}$ obtener(i, e.stocks) $\wedge$ precio(p, i) $=_{obs}$ obtener(o, e.precios))) $\wedge$ (∀a: persona)((def?(a, e.gastos) $\rightarrow_L$ ventas(p,a) $=_{obs}$ multiconjuntoGastos(claves(obtener(a, e.gastos)), obtener(a, e.gastos))) $\wedge$ (¬def?(a, e.gastos) $\rightarrow_L$ ventas(p,a) = $\emptyset$)) $\wedge$ (∀i :item)((def?(i, e.descuentos) $\rightarrow_L$ descuento(p, i, c) = obtenerDescuento(c, obtener(i, e.descuentos)))

Especificación de las operaciones auxiliares utilizadas en la representación

✓ multiconjuntoGastos : conj(item)c $\times$ dicc(item $\times$ secu(secu(cant)))d $\longrightarrow$ multiconj($\langle$ item, cant $\rangle$)
{c $\subseteq$ claves(d) $\wedge$(∀i: item)(def?(i,d) $\rightarrow_L$ long(obtener(i,d)) = 2)}

✓ multiconjuntoGastos(c, d) $\equiv$ **if** $\emptyset$?(c) **then**
$\emptyset$
else
{multiconjuntoGastosAux(dameUno(c), prim(obtener(dameUno(c), d)) & ult(obtener(dameUno(c), d)))} $\cup$ {multiconjuntoGastos(sinUno(c), d)}
fi

✓ multiconjuntoGastosAux : item $\times$ secu(cant) $\longrightarrow$ multiconj($\langle$ item, cant $\rangle$)

✓ multiconjuntoGastosAux(i, s) $\equiv$ **if** vacía?(s) **then** $\emptyset$ **else** {$\langle$ i, prim(s) $\rangle$} $\cup$ multiconjuntoGastosAux(i, fin(s)) **fi**

✓ ObtenerGastoTotal : multiconj($\langle$item $\times$ cant$\rangle$) $\times$ pue $\longrightarrow$ dinero

✓ ObtenerGastoTotal(m, e) $\equiv$ **if** $\emptyset$?(m) **then**
0
else
CalcularPrecioDeVenta(dameUno(m), e) + ObtenerGastoTotal(sinUno(m), e)
fi

✓ CalcularPrecioDeVenta : $\langle$item $\times$ cant$\rangle$ $\times$ pue $\longrightarrow$ dinero

✓ CalcularPrecioDeVenta(t, e) $\equiv$ obtener(π_1(t), e.precios) * π_2(t) * (100 - obtenerDescuento(π_2(t), obtener(π_1(t), e.descuentos)) / 100

✓obtenerDescuento : cant × secu(tupla(cant × nat)) ⟶ nat

✓obtenerDescuento(c, s) ≡ **if** vacía?(s) **then** 0 **else** (**if** π_1(ult(s)) <c **then** π_2(ult(s)) **else** obtenerDescuento(c, com(s))) **fi**) **fi**

Algoritmos

✓**iNuevoPuesto**(**in** p: `diccLog(item, nat)`, **in** s: `diccLog(item, cant)`, **in** d: `diccLog(item, dicc(cant, nat))` → res : pue

```
res.stocks ← s                                                          ▷ O(1)
res.precios ← p                                                         ▷ O(1)
res.gastosTotales← vacío()                                              ▷ O(1)
res.gastos← vacío()                                                     ▷ O(1)
res.descuentos ← vacío()                                                ▷ O(1)
itD ← CrearIt(d)                                                        ▷ O(1)
while HaySiguiente(itD) do                                              ▷ O(I)
    Definir(res.descuentos, siguienteClave(itD), vacía())               ▷ O(log I)
    itCant ← CrearIt(siguienteSignificado(itD))                         ▷ O(1)
    v ← significado(res.descuentos, siguienteClave(itD))                ▷ O(log I)
    while HaySiguiente(itCant) do                                       ▷ O(cant)
        AgregarAtrás(v, siguiente(itCant))                              ▷ O(1)
        Avanzar(itCant)                                                 ▷ O(1)
    end while
    i ← 0                                                               ▷ O(1)
    while i <Longitud(v) do                                             ▷ O(cant)
        indiceMin ← i                                                   ▷ O(1)
        j ← i                                                           ▷ O(1)
        while j <Longitud(v) do                                         ▷ O(cant)
            if v[j].c <v[indiceMin].c then                              ▷ O(1)
                indiceMin ← j                                           ▷ O(1)
            end if
            j ← j + 1                                                   ▷ O(1)
        end while
        aux ← v[i]                                                      ▷ O(1)
        v[i] ← v[indiceMin]                                             ▷ O(1)
        v[indiceMin] ← aux                                              ▷ O(1)
        i ← i + 1                                                       ▷ O(1)
    end while
    Avanzar(itD)                                                        ▷ O(1)
end while
```

✓Complejidad: $O(I * (log(I) + cant^2))$

✓Justificación: se busca sobre todos los items, lo cual tiene una complejidad de $O(I)$, y luego dentro se definen las funciones con complejidad logaritmica que se aplican a I, y se accede en complejidad cuadratica a *cant*

✓**iObtenerStock**(**in** p: `pue`, **in** i: `item`) → res: cant

```
res ← significado(p.stocks, i)                                          ▷ O(log I)
```

Complejidad: $O(\log I)$

✓Justificación: Como stocks se representa con un diccionario logarítmico, encontrar al item i para obtener su stock tiene una complejidad O(log I), siendo I todos los items del puesto p.

---

iObtenerDescuento(**in** p : pue, **in** i : item, **in** c : cant) → res: nat

```
if TieneDescuento?(p, i, c) then                                  ▷ O(log I)
    s ← significado(p.descuentos,i)                               ▷ O(log I)
    i ← 0                                                         ▷ O(1)
    d ← Longitud(s) - 1                                           ▷ O(1)
    if c >= s[d].c then                                           ▷ O(1)
        res ← s[d].d                                              ▷ O(1)
    else
        while (i <d-1) do                                         ▷ O(log cant)
            m ← (i + d) / 2                                       ▷ O(1)
            if c >= s[m].c then                                   ▷ O(1)
                i ← m                                             ▷ O(1)
            else
                d ← m                                             ▷ O(1)
            end if
        end while
        res ← s[i].d                                              ▷ O(1)
    end if
else
    res ← 0                                                       ▷ O(1)
end if
```

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

Justificación: Primero verifica si el item i tiene un descuento para esa cantidad. En el caso de que no, devuelve 0. Si la tiene, realiza una busqueda binaria sobre el vector de descuentos para cantidades de cada item en el diccionario *descuentos*. Finalmente, por álgebra de órdenes la complejidad resulta $O(\log I + \log cant)$

---

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

```
res ← significado(p.gastosTotales, a)                             ▷ O(log A)
```

Complejidad: $O(\log A)$

Justificación: Como gastosTotales se representa con un diccionario logarítmico, buscar a la persona a toma, en peor caso, una complejidad $O(\log A)$ donde A es la cantidad total de personas que compraron al menos una vez en el puesto p.

---

iObtenerPrecio(**in** p : pue, **in** i : item) → res: dinero

```
res ← significado(p.precios, i)                                   ▷ O(log I)
```

Complejidad: $O(\log I)$

Justificación: En la estructura, precios es un diccionario logarítmico. Entonces, encontrar al item i tiene una complejidad O(log I).

---

iVender(**in/out** p : pue, **in** a : persona, **in** i : item, **in** c : cant)

```
significado(p.stocks, i) ← significado(p.stocks, i) - c                    ▷ O(log I)
if !definido?(p.gastosTotales, a) then                                     ▷ O(log A)
    definir(p.gastosTotales, a, 0)                                         ▷ O(log A)
end if
precio ← calcularPrecio(p,i,c)                                             ▷ O(log I + log cant)
significado(p.gastosTotales, a) ← significado(p.gastosTotales, a) + precio ▷ O(log A)
if !definido?(p.gastos, a) then                                            ▷ O(log A)
    definir(p.gastos, a, arreglo[2])                                       ▷ O(log A)
    significado(p.gastos, a)[0] = vacía()                                  ▷ O(log A)
    significado(p.gastos, a)[1] = vacía()                                  ▷ O(log A)
end if
itemCantidad ← ⟨item: i, cant: c⟩                                          ▷ O(1)
if tieneDescuento?(p, i ,c) then                                           ▷ O(log I + log cant)
    agregarAtrás(significado(p.gastos, a)[1], itemCantidad)                ▷ O(log A)
else
    agregarAtrás(significado(p.gastos, a)[0], itemCantidad)                ▷ O(log A)
end if
```

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

Justificación: El algoritmo llama a funciones de complejidad O(log I), O(log A), O(1) y O(log I + log $cant$). Aplicando álgebra de órdenes, la complejidad queda de O(log I + log A + log $cant$).

---

iTieneDescuento?(**in** p : pue, **in** i : item, **in** c : cant) → res : bool

```
res ← EsVacía?(significado(p.descuentos, i)) && c >= significado(p.descuentos, i)[0].c   ▷ O(log I)
```

Es la negacion

Complejidad: $O(\log I)$

Justificación: Acceder al item i en descuentos tiene complejidad O(log I), y verificar si la cantidad c está definida O(log cant). La complejidad del algoritmo es O(log I + log cant).

---

iCalcularPrecio(**in** p : pue, **in** i : item, **in** c : cant) → res : dinero

```
precio ← significado(p.precios, i)                                         ▷ O(log I)
if TieneDescuento?(p, i, c) then                                           ▷ O(log I)
    descuento ← ObtenerDescuento(p,i,c)                                    ▷ O(log I + log cant)
    res ← precio * c * (100 - descuento) / 100                             ▷ O(1)
else
    res ← precio * c                                                       ▷ O(1)
end if
```

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

Justificación: Este algoritmo llama a la función tieneDescuento y obtenerDescuento, cuya complejidad es O(log I + log $cant$). Por álgebra de órdenes, dado que las demás son de una complejidad menor, el algoritmo queda con complejidad O(log I + log $cant$).

iAnularCompra(**in/out** p : `estr`, **in** a : `persona`, **in** i : `item`) → res : cant

cantidad ← primero(significado(significado(p.gastos, a), i)[0])	▷ $O(\log A + \log I)$
precio ← significado(p.precios, i) * cantidad	▷ $O(\log I)$
significado(p.gastosTotales, a) ← significado(p.gastosTotales, a) - precio	▷ $O(\log A)$
fin(significado(significado(p.gastos, a), i)[0])	▷ $O(1)$
significado(p.stock, i) ← significado(p.stock, i) + cantidad	▷ $O(\log I)$
res ← cantidad	▷ $O(1)$

Complejidad: $O(\log I + \log A)$
Justificación: El algoritmo llama a funciones $O(\log A + \log I)$, O(log I), O(log A), y O(1). Aplicando álgebra de órdenes, la complejidad de total queda O(log A + log I)

---

iCompróItemSinDescuento?(**in** p : `pue`, **in** a : `persona`, **in** i : `item`) → res : bool

res ← !EsVacía?(significado(significado(p.gastos, a), i)[0])	▷ $O(\log A + \log I)$

Complejidad: $O(\log A + \log I)$
Justificación: la funcion que se usa en el algoritmo es de complegidad logaritmica, y se usa para A y para I, por lo que la complejidad es O(log A + log I)

Módulo Lollapatuza

✓ El módulo Lollapatuza provee una tupla la cual utiliza otros módulos, principalmente diccionario y conjunto logarítmico, para brindar acceso a los puestos disponibles, a las personas registradas y a aquella que más dinero gastó. Además, permite ver dónde una persona realizó la compra sin descuento de un item.

Interfaz

se explica con: LOLLAPATUZA

géneros: `lolla`.

Operaciones básicas de lollapatuza

NUEVOSISTEMA(**in** ps : `dicc(id, puesto)`, **in** as : `conj(persona)`) → res : `lolla`
✓**Pre** ≡ {vendenAlMismoPrecio(significados(ps)) ∧ NoVendieronAun(significados(ps)) ∧ ¬∅?(as) ∧¬∅ ?(claves(ps))}
✓**Post** ≡ {res =$_{\text{obs}}$ crearLolla(ps, as,)}
Complejidad: $O(logA + P * logP)$
Descripción: Crea un nuevo sistema
Aliasing: Todos los tipos no primitivos de entrada se pasan por referencia. Devuelve una referencia modificable de lollapatuza

REGISTRARCOMPRA(**in/out** $lolla$: `l` **in** pi : `puestoId`, **in** a : `persona`, **in** i : `item`, **in** c : `cant`)
✓**Pre** ≡ {l_0 =$_{\text{obs}}$ l ∧a ∈ personas(l) ∧ def?(pi, puestos(l)) ∧$_L$ haySuficiente?(obtener(pi, puestos(l)), i, c)}
✓**Post** ≡ {l =$_{\text{obs}}$ vender(l_0, pi, a, i, c)}
Complejidad: $O(\log(A) + \log(I) + \log(P) + \log(cant))$, donde i es la cantida de items del puesto
Descripción: Modifica los gastos y el stock en el puesto correspondiente. Agrega el valor de la compra al gasto total de la persona y genera un nuevo gasto para esa persona en gastos ordenados, habiendo borrado el viejo anteriormente. Si la compra no tiene descuento, agrega el puesto a puestos ordenados y en caso de ser este el puesto con id minimo para esta persona y este item modifica también próximo hackeo.

HACKEARPERSONA(**in/out** l : `lolla`, **in** a : `persona`, **in** i : `item`)
✓**Pre** ≡ {l =$_{\text{obs}}$ l_0∧ ConsumioSinPromoEnAlgunPuesto(l, a, i)}
✓**Post** ≡ {l =$_{\text{obs}}$ hackear(l_0, a, i)}
Complejidad: $O(\log(A) + \log(I) + \log(P))$, donde i es la cantida de items del pueso
Descripción: Modifica los gastos y el stock en el puesto correspondiente. Resta el valor de la compra hackeada al gasto total de la persona y genera un nuevo gasto para esa persona en gastos ordenados, habiendo borrado el viejo anteriormente. En caso de que ese item ya no pueda ser hackeado en ese puesto para esa persona, se modifica próximo hackeo y puestos ordenados.

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

GASTOMÁXIMO(**in** l : `lolla`) → res : `persona`
✓**Pre** ≡ {true}
✓**Post** ≡ {res =$_{\text{obs}}$ masGasto(l, a)}
Complejidad: $O(1)$
Descripción: Devuelve la persona que más gastó

PUESTOMENOSSTOCK(**in** l : `lolla`, **in** i : `item`) → res : `puestoId`
✓**Pre** ≡ {true}
✓**Post** ≡ {res =$_{\text{obs}}$ menorStock(l, i)}
Complejidad: $O(P * \log I)$
Descripción: Devuelve el puesto que tiene el menor stock de ese item, en caso de que allá dos iguales, devuelve el que tiene menor id

PERSONAS(**in** l : `lolla`) → res : `conj(persona)`
✓**Pre** ≡ {true}
✓**Post** ≡ {res =$_{\text{obs}}$ personas(l)}

Complejidad: $O(1)$
Descripción: Devuelve el conjunto de personas que estan registradas en el Lollapatuza
Aliasing: Devuelve una referencia no modificable

PUESTOSLOLLA(**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 que se encuentran en el Lollapatuza con su respectivo id
Aliasing: Devuelve una referencia no modificable

Especificación de las operaciones auxiliares utilizadas en la interfaz

esPermutacion? : secu(tupla(κ, σ)) $\times$ dicc(κ, σ) $\longrightarrow$ bool

esPermutacion?(s, d) $\equiv$ d = secuADicc(s) $\wedge$ #claves(d) = long(s)

secuADicc : secu(tupla(κ, σ)) $\longrightarrow$ dicc(κ, σ)

secuADicc(s) $\equiv$ **if** vacia?(s) **then** vacio **else** definir(Π_1(prim(s)), Π_2(prim(s)), secuADict(fin(s))) **fi**

Representación

Representación de lollapatuza

El objetivo de este módulo es implementar un instancia de Lollapatuza. Para esto hicimos uso de una tupla, la cual contiene diccionarios logaritmicos y conjuntos logaritmicos. Para poder cumplir con las complejidades pedidas en algúnos algoritmos decidimos implementar diccionarios cuyo significados fueran *iteradores* de otros diccionarios.

Llamaremos *diccLog* al diccionario logarítmico para diferenciarlo del lineal.

La función de orden total para el tipo tupla(gasto: dinero, per: persona) la definimos como: una tupla t_1 de este tipo es menor a otra t_2 si y solo sí $t_1.gasto$ es mayor que $t_2.gasto$, o si $t_1.gasto$ es igual $t_2.gasto$ pero $t_1.persona$ es mayor que $t_2.persona$.

La función de orden total para el tipo tupla(id: puestoid, iterador: itDiccLog(puestoid, puesto)) la definimos como: una tupla t_1 de este tipo es menor a otra t_2 si y solo sí $t_1.id$ es menor que $t_2.id$.

`lollapatuza` **se representa con** `lol`

donde `lol` es `tupla`(*puestos*: `diccLog(puestoid, puesto)` ,
iteradoresPorPuesto: `diccLog(puestoid, itDiccLog(puestoid, puesto))` ,
gastoTotalPorPersona: `diccLog(persona, dinero)`,
personas: `conjLog(persona)` ,
personaQueMásGastó: `tupla(gasto: dinero, per: persona)`,
gastosOrdenados: `conjLog(tupla(gasto: dinero, per: persona))` ,
próximoHackeo: `diccLog(persona, diccLog(item, tupla(id: puestoid, iterador: itDiccLog(puestoid, puesto))))` ,
puestosOrdenados: `diccLog(persona, diccLog(item, conjLog(tupla(id: puestoid, iterador: itDiccLog(puestoid, puesto)))))`)

Rep : lol $\longrightarrow$ bool

✓Rep(e) $\equiv$ true $\iff$
claves(e.puestos) = claves(e.iteradoresPorPuesto) $\wedge_L$
vendenAlMismoPrecio(significados(e.puestos)) $\wedge_L$
(∀p: puestoid)(def?(p, e.puestos) $\rightarrow_L$ siguiente(obtener(p, e.iteradoresPorPuesto)) = ⟨p, obtener(p, e.puestos)⟩) $\wedge_L$
claves(e.gastosOrdenados) ∪ claves(e.gastoTotalPorPersona) ∪ claves(e.próximoHackeo) ⊆ e.personas $\wedge_L$
e.personasQueMásGastó = máx(e.gastosOrdenados) $\wedge_L$
diccionarioAConjunto(e.gastoTotalPorPersonas, claves(e.gastoTotalPorPersona)) = e.gastosOrdenados $\wedge_L$
claves(e.próximoHackeo) = claves(e.puestosOrdenados) $\wedge_L$
(∀a: persona)(def?(a, e.puestosOrdenados) $\rightarrow_L$ (∀i: item)(def?(i, obtener(a, e.puestosOrdenados))$\rightarrow_L$ (∀t: tupla(puestoid, itDiccLog(puestoid,puesto)))(t ∈ obtener(i,(obtener(a, e.puestosOrdenados)))) → i ∈ menu(obtener(π_1(t), e.puestos)))) $\wedge_L$
(∀a: persona)(def?(a, e.puestosOrdenados) $\rightarrow_L$ claves(a, e.puestosOrdenados) = claves(obtener(a, e.próximoHackeo))) $\wedge_L$
(∀a: persona)(def?(a, e.próximoHackeo) $\rightarrow_L$ (∀i: item)(def?(i, obtener(a, e.próximoHackeo))$\rightarrow_L$) obtener(i, obtener(a, e.próximoHackeo)) = mín(obtener(i, obtener(a, e.puestosOrdenados)))) $\wedge_L$
(∀a: persona)(def?(a, e.puestosOrdenados) $\rightarrow_L$ (∀i: item)(def?(i, obtener(a, e.puestosOrdenados))$\rightarrow_L$ obtener(i, obtener(a, e.puestosOrdenados)) ⊆ diccionarioAConjunto(e.iteradoresPorPuesto, claves(e.iteradoresPorPuesto)))) $\wedge_L$
(∀a: persona)(def?(a, e.puestosOrdenados) $\rightarrow_L$ (∀i: item)(def?(i, obtener(a, e.puestosOrdenados))$\rightarrow_L$ (∀t: tupla(puestoid, itDiccLog(puestoid,puesto)))(t ∈ obtener(i,(obtener(a, e.puestosOrdenados))) $\rightarrow_L$ ¬vacía?(prim(obtener(i,obtener(a, gastos(obtener(π_1, e.puestos))))))))) $\wedge_L$
(∀a: persona)(def?(a, e.gastoTotalPorPersona) $\rightarrow_L$ calcularGastoTotal(e.puestos, claves(e.puestos), a) = obtener(a, e.gastoTotalPorPersona))

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

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

Especificación de las operaciones auxiliares utilizadas en la representación

✓diccionarioAConjunto : dicc($\kappa \times \sigma$)d × conj(κ)c $\longrightarrow$ conj(tupla(κ, σ)) {c ⊆ claves(d)}

✓diccionarioAConjunto(d,c) $\equiv$ **if** ∅?(c) **then**
∅
else
{⟨ dameUno(c), obtener(dameUno(c), d) ⟩} ∪ diccionarioAConjunto(d, sinUno(c))
fi

✓calcularGastoTotal : dicc(puestoid × puesto)d × conj(puestoid)c × persona $\longrightarrow$ dinero {c ⊆ claves(d)}

✓calcularGastoTotal(d,c,a) ≡ **if** vacío?(c) **then**
0
else
gastoDe(obtener(dameUno(c), d), a) + calcularGastoTotal(d, sinUno(c), a)
fi

Algoritmos

✓**iNuevoSistema**(**in** *ps*: `dicc(puestoid, puesto)`, **in** *as*: `conj(persona)`) → *res* : lol

res.puestos ← ps	▷ $O(1)$
res.iteradoresPorPuesto ← Vacío()	▷ $O(1)$
itPuesto = CrearIt(res.puestos)	▷ $O(1)$
while HaySiguiente(itPuesto) **do**	▷ $O(P)$
Definir(res.iteradoresPorPuesto, siguienteClave(itPuesto), itPuesto)	▷ $O(\log P)$
Avanzar(itPuesto)	▷ $O(1)$
end while	
res.gastoTotalPorPersona ← Vacío()	▷ $O(1)$
res.personas ← as	▷ $O(1)$
res.personaQueMásGastó ← ⟨0, mín(as)⟩	▷ $O(\log A)$
res.gastosOrdenados = agregar(Vacío(), ⟨0, mín(as)⟩)	▷ $O(logA)$
res.próximoHackeo ← Vacío()	▷ $O(1)$
res.puestosOrdenados ← Vacío()	▷ $O(1)$

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

✓Justificación: El ciclo se repite P veces porque debe verificar todos los puestos del sistema. Dentro del ciclo hay una operación de complejidad O(log P). Además, se le debe asignar un valor a la persona que más gastó. Como al arrancar el sistema todos gastaron lo mismo (0), se asigna al de menor id, y esa busqueda es O(log A). Finalmente, la complejidad del algoritmo resulta O(log A + P * log P).

---

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

puesto ← significado(l.puestos, pi) ▷ $O(\log P)$
vender(puesto, a, i, c) ▷ $O(\log I + \log A + \log cant)$
if !definido?(l.gastoTotalPorPersona, a) **then** ▷ $O(\log A)$
definir(l.gastoTotalPorPersona, a, 0) ▷ $O(\log A)$
end if
eliminar(l.gastosOrdenados, ⟨significado(l.gastoTotalPorPersona, a), a⟩) ▷ $O(\log A)$
precio ← calcularPrecio(puesto,i,c) ▷ $O(\log I + \log cant)$
significado(l.gastoTotalPorPersona, a) ← significado(l.gastoTotalPorPersona, a) + precio ▷ $O(\log A)$
agregarRápido(l.gastosOrdenados, ⟨significado(l.gastoTotalPorPersona, a), a⟩) ▷ $O(\log A)$
l.personaQueMásGastó ← máx(l.gastosOrdenados) ▷ $O(\log A)$
if !tieneDescuento?(puesto, i, c) **then** ▷ $O(\log I)$
itPuesto ← significado(l.iteradoresPorPuesto, pi) ▷ $O(\log P)$
if !Definido?(l.puestosOrdenados, a) **then** ▷ $O(\log A)$
Definir(l.puestosOrdenados, a, Vacío()) ▷ $O(\log A)$
end if
if !Definido?(singificado(l.puestosOrdenados, a), i) **then** ▷ $O(\log I + \log A)$
Definir(singificado(l.puestosOrdenados, a), i, Vacío()) ▷ $O(\log I + \log A)$
end if
agregar(significado(singificado(l.puestosOrdenados, a), i), ⟨pi, itPuesto⟩) ▷ $O(\log I + \log A + \log P)$
if !Definido?(l.próximoHackeo, a) **then** ▷ $O(\log A)$
Definir(l.próximoHackeo, a, Vacío()) ▷ $O(\log A)$
end if
if !Definido?(singificado(l.próximoHackeo, a), i) **then** ▷ $O(\log I + \log A)$
Definir(singificado(l.próximoHackeo, a), i, ⟨pi, itPuesto⟩) ▷ $O(\log I + \log A)$
else
siginficado(significado(l.próximoHackeo, a), i) ← mín(significado(singificado(l.puestosOrdenados, a), i)) ▷ $O(\log P + \log A + \log I)$
end if
end if

Complejidad: $O(\log P + \log A + \log I + \log cant)$
Justificación: La primera linea usa una función logarítmica para los puestos, por lo que ya existe un O($\log P$). Luego, la función vender tiene complejidad logaritmica para los parámetros I, A, y $cant$. No hay ninguna parte del codigo que genere mayor complejidad que estas lineas. por álgebra de órdenes, quedaría como O($\log I$ + $\log A$ + $\log cant$ + $\log P$)

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

id ← significado(significado(l.próxmoHackeo, a), i).id ▷ $O(\log I + \log A)$
itPuesto ← significado(significado(l.próxmoHackeo, a), i).iterador ▷ $O(\log I + \log A)$
puesto ← siguienteSignificado(itPuesto) ▷ $O(1)$
eliminar(l.gastosOrdenados, ⟨significado(l.gastoTotalPorPersona, a), a⟩) ▷ $O(\log A)$
cantidad ← anularCompra(puesto, a, i) ▷ $O(\log I + \log A)$
precio ← obtenerPrecio(puesto, i) * cantidad ▷ $O(\log I)$
significado(l.gastoTotalPorPersona, a) ← singificado(l.gastoTotalPorPersona, a) - precio ▷ $O(\log A)$
agregar(l.gastosOrdenados, ⟨significado(l.gastoTotalPorPersona, a), a⟩) ▷ $O(\log A)$
l.personaQueMásGastó ← máx(l.gastosOrdenados) ▷ $O(\log A)$
if !compróItemSinDescuento?(puesto, a, i) **then** ▷ $O(\log I + \log A)$
  eliminar(significado(significado(l.puestosOrdenados, a), i), ⟨id, itPuesto⟩) ▷ $O(\log I + \log A + \log P)$
  if EsVacío?(significado(significado(l.puestosOrdenados, a), i)) **then** ▷ $O(\log I + \log A)$
    borrar(significado(l.puestosOrdenados), a) ▷ $O(\log A)$
    borrar(significado(l.próximoHackeo), a) ▷ $O(\log A)$
  else
    singificado(significado(l.próxmioHackeo, a), i) ← mín(significado(significado(l.puestosOrdenados, a), i)) ▷ $O(\log I + \log A + \log P)$
  end if
end if

Complejidad: $O(\log I + \log A + \log P)$
Justificación: En el peor caso debe eliminar el puesto de puestosOrdenados, porque ya no quedan items para hackear en ese puesto. Esto tiene complejidad de O($\log I$ + $\log A$ + $\log P$). Por álgebra de órdenes, como ninguna otra función tiene una complejidad menor, la complejidad del algoritmo resulta O($\log I$ + $\log A$ + $\log P$). En el resto de los casos, como no se ejecuta el bloque que está adentro del if, la complejidad es O($\log I$ + $\log A$) ya que no se ejecuta ninguna instrucción que sea O($\log P$).

---

iGastoPersona(**in** l : `lol`, **in** a : `persona`) → res: dinero

res ← significado(l.gastoTotalPorPersona, a)

Complejidad: $O(\log A)$
Justificación: La función usada es logarítmica y se usa el A que son la cantidad de personas, entonces la complejidad es O($\log A$).

---

iGastoMáximo(**in** l : `lol`) → res: persona

res ← l.personaQueMásGastó

Complejidad: $O(1)$
Justificación: Accede a la función *personaQueMasGasto* en complejidad constante.

---

iPersonas(**in** l : `lol`) → res: conj(persona)

res ← l.personas

Complejidad: $O(1)$
Justificación: Accede a la función *personas* en complejidad constante.

iPuestosLolla(**in** l : `lol`) → res: dicc(puestoid, puesto)
res ← l.puestos

Complejidad: $O(1)$
Justificación: Accede a la función *personaQueMasGasto* en complejidad constante.

iPuestoMenosStock(**in** l : `lol`, **in** a : `persona`, **in** i : `item`)
itPuesto ← crearIt(l.puestos) ▷ $O(1)$
idMenorStock ← siguienteClave(itPuesto) ▷ $O(1)$
puestoMenorStock ← siguienteSignificado(itPuesto) ▷ $O(1)$
while haySiguiente?(itPuesto) **do** ▷ $O(P)$
puesto ← siguienteSignificado(itPuesto) ▷ $O(1)$
id ← siguienteClave(itPuesto) ▷ $O(1)$
if obtenerStock(puesto, i) < obtenerStock(puestoMenorStock, i) || (obtenerStock(puesto, i) == obtenerStock(puestoMenorStock, i) && id < idMenorStock)) **then** ▷ $O(\log I)$
idMenorStock ← id ▷ $O(1)$
puestoMenorStock ← puesto ▷ $O(1)$
end if
avanzar(itPuesto) ▷ $O(1)$
res ← idMenorStock ▷ $O(1)$
end while

Complejidad: $O(P * \log I)$
Justificación: El ciclo se repite P veces pues debe ver todos los puestos del sistema. De esta forma, la guarda del If se va a verificar P veces, con una complejidad de O(log I). Finalmente, como todas las demás operaciones son constantes, la complejidad del algoritmo es de O(P * log I).

Módulo Conjunto Logarítmico(α)

El módulo Conjunto Logarítmico provee un conjunto básico en el que se puede insertar, eliminar, y testear pertenencia en tiempo logarítmico (de comparaciones y/o copias). Además, aprovechando el uso de árboles balanceados, permite encontrar el elemento minimo del conjunto en complejidad constante.

Se requiere que exista una función de orden total definida para el tipo α.

Para describir la complejidad de las operaciones, vamos a llamar $copy(a)$ al costo de copiar el elemento $a \in \alpha$ y $equal(a_1, a_2)$ al costo de evaluar si dos elementos $a_1, a_2 \in \alpha$ son iguales (i.e., *copy* y *equal* son funciones de α y $\alpha \times \alpha$ en $\mathbb{N}$, respectivamente). Se le llamará *compare* al costo de evaluar si un elemento a_1 es menor que otro a_2.

Interfaz

parámetros formales
géneros α

función $\bullet = \bullet$(**in** $a_1 : \alpha$, **in** $a_2 : \alpha$) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (a_1 = a_2)$}
Complejidad: $\Theta(equal(a_1, a_2))$
Descripción: función de igualdad de α's

función Copiar(**in** $a : \alpha$) $\rightarrow res$: α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's

función $\bullet < \bullet$(**in** $a_1 : \alpha$, **in** $a_2 : \alpha$) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (a_1 < a_2)$)}
Complejidad: $\Theta(compare(a))$
Descripción: función de orden total de α's

se explica con: Conj(α).

géneros: `conj`(α).

Operaciones básicas de conjunto logarítmico

Vacío() $\rightarrow res$: `conj`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \emptyset$}
Complejidad: $\Theta(1)$
Descripción: genera un conjunto vacío.

Agregar(**in/out** c: `conj`(α), **in** $a : \alpha$)
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}} Ag(a, c_0)$}
Complejidad: $\Theta\left(\log(\sum_{a' \in c} equal(a, a')) + copy(a)\right)$
Descripción: agrega el elemento a al conjunto.
Aliasing: el elemento a se agrega por copia.

EsVacío?(**in** c: `conj`(α)) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \emptyset?(c)$}
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si c esta vacío.

Pertenece?(**in** c: `conj`(α), **in** $a : \alpha$) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a \in c$)}
Complejidad: $\Theta\left(\log(\sum_{a' \in c} equal(a, a'))\right)$
Descripción: devuelve `true` si y sólo a pertenece al conjunto.

Eliminar(**in** c: `conj`(α), **in** $a : \alpha$)

Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} c \setminus \{a\})\}$
Complejidad: $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
Descripción: elimina a de c, si es que estaba.

MÍN(**in** c: `conj(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ $\{\#c > 0\}$
Post $\equiv$ $\{res \in c \wedge (\forall e\text{: } \alpha)(e \in c \rightarrow res \leq e\ )\ )\}$
Complejidad: $\Theta(log(\#c) + copy(\alpha))$
Descripción: devuelve el menor elemento del conjunto según lo indique la función de orden total.
Aliasing: se devuelve res por copia

Representación

Representación del Conjunto

En este módulo vamos a utilizar un diccionario logarítmico para representar el conjunto. La idea es que el conjunto de claves del diccionario represente el conjunto logarítmico. Utilizamos este representación puesto que el módulo diccionario logarítmico ya está diseñado por la cátedra de la materia.
Llamaremos diccLog al diccionario logarítmico para diferenciarlo del lineal.

`conjLog(`α`)` **se representa con** `diccLog(`α`, bool)`

Rep : diccLog(α,bool) $\longrightarrow$ bool
Rep(d) $\equiv$ true
Abs : diccLog(α,bool) d $\longrightarrow$ conjLog(α) {Rep(d)}
Abs(d) $\equiv$ claves(d)

Algoritmos

---

iVacío() $\rightarrow$ res : diccLog(α, bool)
res $\leftarrow$ Vacío() ▷ $O(1)$

Complejidad: $O(1)$

---

iAgregar(**in/out** c: `diccLog(`α`, bool)`, **in** e: α)
Definir(c, a, True) ▷ $O(\log n)$

Complejidad: $O(\log n)$, donde n es la cantidad de claves del diccionario.

---

iEsVacío?(**in** c: `diccLog(`α`, bool))` $\rightarrow$ `res:` α
res $\leftarrow$ #Claves(c) == 0 ▷ $O(1)$

Complejidad: $O(1)$

---

iPertenece?(**in** c: `diccLog(`α`, bool)`, **in** e: α) $\rightarrow$ res: α

✓ res $\leftarrow$ Definido?(c, e) ▷ $O(\log n)$

Complejidad: $O(\log n)$, donde n es la cantidad de claves del diccionario.

iEliminar(**in/out** c: `diccLog(`α`, bool)`, **in** e: α)

✓ Borrar(c,e) ▷ $O(\log n)$

Complejidad: $O(\log n)$, donde n es la cantidad de claves del diccionario.

iMín(**in** c: `diccLog(`α`, bool)`) $\rightarrow$ res: α

✓ res $\leftarrow$ siguienteClave(CrearIt(c)) ▷ $O(1)$

Complejidad: $O(1)$

✓ 3. Conclusiones

Concluímos que resulta útil la etapa de diseño de un problema para así escoger las estructuras ideales para luego implementarlo de manera eficiente. En el caso de los módulos diseañados en este trabajo práctico, elegimos para su reperentación combinar principalmente diccionarios logaritmicos. De esta forma, pudimos cumplir con la complejidad pedida en las operaciones.