Algoritmos y Estructuras de Datos II

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

Trabajo Práctico 2: Diseño

Lollapatuza

Alias de grupo: MVNWOFAHTVRARZOSADUY

Integrante	LU	Correo electrónico
Rafael Montero	1546/21	rafamontero1000@gmail.com
Esteban Mena	540/22	estebanpetiso@gmail.com
Mateo Lazarte	539/22	mateolazarte07@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Módulo Lollapatuza

Interfaz

se explica con: LOLLAPATUZA

géneros: `lolla`.

Operaciones básicas de Lollapatuza

CREARLOLLA(**in** ps : `dicc(idpuesto, 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: $\Theta(Alog(A))$
Descripción: Genera un nuevo lolla con los puestos de ps y las personas de as.

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)) $\wedge_{\text{L}}$ haySuficiente?(obtener(pi, puestos(l)), i, c)}
Post $\equiv$ {$l =_{\text{obs}}$ vender(l_0, pi, a, i, c)}
Complejidad: $\Theta(log(A) + log(I) + log(P))$
Descripción: Registra una compra de una persona de cierta cantidad de un item 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)}
Post $\equiv$ {$l =_{\text{obs}}$ hackear(l_0, a, i)}
Complejidad: $\Theta(log(A) + log(I) + log(P))$
Descripción: Hackea en el lolla a la persona a que haya comprado el item i en el puesto de menor id sin descuento.

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: $\Theta(log(A))$
Descripción: Obtiene el gasto total de una persona en el lolla.
Aliasing: Se devuelve una referencia no modificable al gasto de una persona

MASGASTO(**in** l : `lolla`) $\rightarrow$ res : `persona`
Pre $\equiv$ {$\neg\emptyset?$(personas(l))}
Post $\equiv$ {$res =_{\text{obs}}$ masGasto(l)}
Complejidad: $\Theta(1)$
Descripción: Obtiene la persona que más gastó en el lolla.
Aliasing: res no es modificable.

MENORSTOCK(**in** l : `lolla`, **in** i : `item`) $\rightarrow$ res : `idpuesto`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ menorStock(l, i)}
Complejidad: $\Theta(P * log(I))$
Descripción: Obtiene el idpuesto que tiene el menor stock del item i.
Aliasing: Se devuelve una referencia no modificable a un idpuesto.

OBTENERPERSONAS(**in** l : `lolla`) $\rightarrow$ res : `itConj(persona)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion(SecuSuby(res), personas(l)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripción: Obtiene un iterador bidireccional al primer elemento del conjunto de personas, puediendo recorrer los elementos aplicando iterativamente Siguiente.
Aliasing: El iterador no se invalida pues el conjunto que recorre no es modificable.

OBTENERPUESTOS(**in** l : `lolla`) $\rightarrow$ res : `itDicc(idpuesto,puesto)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion(SecuSuby(res), puestos(l)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripción: Obtiene un iterador al primer elemento del diccionario de idpuestos, puediendo recorrer los elementos aplicando iterativamente Siguiente.
Aliasing: El iterador se invalida sii se elimina el elemento siguiente del it sin utilizar la funcion EliminarSiguiente.

Representación

Representación del Lollapatuza

El lollapatuza se decidió representar con las siguientes estructuras:

- *puestos*, un diccionario avl que tiene de claves todas las ID de los puestos del lolla, y como signficado su puesto. El dicc avl nos permite buscar un puesto $\Theta(log(P))$;
- *stock*, un conjunto lineal que solo guarda las personas del lolla. Su única función es facilitar la creación de un iterador en $\Theta(1)$ que permita ver todas las personas del lolla;
- *gastoPersonas*, un diccionario avl que tiene de claves todas las personas que hayan hecho alguna compra en el lolla, y como significado su gasto total realizado. El dicc avl nos permite buscar y definir los gastos de las personas que realicen alguna compra en $\Theta(log(A))$;
- *gastos*, un diccionario avl que tiene de claves todos los gastos realizados por personas en el lolla, y como significado otro dicc avl con las personas que tengan el mismo gasto. El segundo diccionario funciona solo para almacenar personas, por lo que no nos interesa su significado y lo definimos siempre en NULL. Se debe apuntar y guardar con un iterador al mayor gasto realizado por una o varias personas. El dicc avl nos permite operar en $\Theta(log(A))$, donde A es la cantidad de personas y a lo sumo hay tantos gastos diferentes como personas en el lolla.;
- *masGasto*, que guarda el iterador al elemento de mayor gasto en *gastos*, que nos permite obtener en $\Theta(1)$ la persona que más gastó del lolla. En caso de haber varias personas, se desempata por menor ID de una persona creando un iterador al primer elemento del diccionario de personas. *masGasto* se debe actualizar si al registrar una compra aparece un gasto mayor. También nos facilita hackear ya que, asumiento recorrido inorder del dicc avl, el iterador anterior a *masGasto* siempre es el segundo gasto más alto en el lolla. Actualizarlo siempre será $\Theta(1)$.
- *comproSinDescuento*, un dicc avl que registra para cada persona cada ítem que compró sin descuento en qué puestos. Esta estructura facilita la búsqueda del puesto de menor ID que debe ser hackeado, creando un iterador al primer elemento del diccionario. Con esto, el costo de la búsqueda es $\Theta(log(A) + log(I))$. Cada vez que se compre sin descuento en un puesto, se debe agregar a la estructura un iterador que corresponda a su lugar en e.puestos. De esta manera podemos mantener actualizado el diccionario e.puestos en $\Theta(1)$ cada vez que ocurra un hackeo.

.

`lolla` se representa con `estr`

donde `estr` es `tupla(` *puestos*: `diccLog(idpuesto, puesto)`
, *personas*: `conj(persona)`
, *gastoPersonas*: `diccLog(persona, dinero)`
, *gastos*: `diccLog(dinero, diccLog(persona, NULL))`
, *masGasto*: `itDiccLog(dinero, diccLog(persona, NULL))`
, *comproSinDescuento*: `diccLog(persona, diccLog(item, diccLog(idpuesto, itDiccLog(idpuesto, puesto))))`)
donde `persona` es `nat`
donde `idpuesto` es `nat`
donde `item` es `nat`

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff 1 \wedge_{\text{L}} 2 \wedge 3 \wedge_{\text{L}} 4 \wedge_{\text{L}} 5 \wedge 6 \wedge_{\text{L}} 7 \wedge 8$
1 $\equiv$ claves(e.gastoPersonas) = e.personas
2 $\equiv$ #claves(e.gastos) $\leq$ #claves(e.gastoPersonas)
3 $\equiv$ ($\forall p$: persona)(def?(p, e.gastoPersonas) $\Rightarrow_{\text{L}}$ def?(obtener(p, e.gastoPersonas), e.gastos) $\wedge_{\text{L}}$
def?(p, obtener(obtener(p, e.gastoPersonas), e.gastos)) $\wedge_{\text{L}}$
NULL = obtener(p, obtener(obtener(p, e.gastoPersonas), e.gastos)))
4 $\equiv$ ($\forall d$: dinero)(def?(d, e.gastos) $\Rightarrow_{\text{L}}$ ($\forall p$: persona)(def?(p, obtener(d, e.gastos)) $\Rightarrow_{\text{L}}$
NULL = obtener(p, obtener(d, e.gastos)) $\wedge_{\text{L}}$ def?(p, gastoPersonas) $\wedge_{\text{L}}$ d = obtener(p, e.gastoPersonas)))
5 $\equiv$ esPermutación(SecuSuby(e.masGasto), e.gastos) $\wedge$ HaySiguiente?(e.masGasto) $\wedge_{\text{L}}$
esPermutación(Anteriores(e.masGasto), borrar(SiguienteClave(e.masGasto), e.gastos)) $\wedge$
$\neg(\exists d$: dinero)(def?(d, e.gastos) $\wedge$ d $\neq$ SiguienteClave(e.masGasto) $\wedge_{\text{L}}$ d > SiguienteClave(e.masGasto))
6 $\equiv$ claves(e.comproSinDescuento) $\subseteq$ e.personas

7 ≡ ($\forall p$: persona)(def?(p, e.comproSinDescuento) $\Rightarrow_{\text{L}}$ ($\forall i$: item)(def?(i, obtener(p, e.comproSinDescuento)) $\Rightarrow_{\text{L}}$ ($\forall pi$: idpuesto)(def?(pi, obtener(i, obtener(p, e.comproSinDescuento))) $\Rightarrow_{\text{L}}$ def?(pi, e.puestos) $\wedge_{\text{L}}$ haySiguiente?(obtener(pi, obtener(i, obtener(p, e.comproSinDescuento)))) $\wedge_{\text{L}}$ obtener(pi, e.puestos) = SiguienteSignificado(obtener(pi, obtener(i, obtener(p, e.comproSinDescuento)))) $\wedge$ pi = SiguienteClave(obtener(pi, obtener(i, obtener(p, e.comproSinDescuento)))) $\wedge$ esPermutación(SecuSuby(SiguienteSignificado(obtener(pi, obtener(i, obtener(p, e.comproSinDescuento))))), e.puestos) $\wedge_{\text{L}}$ i $\in$ menu(obtener(pi, e.puestos)) $\wedge_{\text{L}}$ consumioSinPromo?(obtener(pi, e.puestos), p, i))))

8 ≡ ($\forall p$: persona)(def?(p, e.gastoPersonas) $\Rightarrow_{\text{L}}$ obtener(p, e.gastoPersonas) = totalGasto(p, e.puestos))

totalGasto : persona × dicc(idpuesto, puesto) $\longrightarrow$ dinero
totalGasto(a, ps) ≡ **if** #claves(ps) = 0 **then**
0
else
gastosDe(obtener(dameUno(#claves(ps)), a)) +
totalGasto(a, borrar(dameUno(#claves(ps)), ps))
fi

Abs : estr e $\longrightarrow$ lolla {Rep(e)}
Abs(e) =$_{\text{obs}}$ l: lolla | e.puestos =$_{\text{obs}}$ puestos(l) $\wedge$ e.personas =$_{\text{obs}}$ personas(l)

Algoritmos

CrearLolla(**in** *ps*: `dicc(idpuesto, puesto)`, **in** *as*: `conj(persona)`) → *res* : *estr*

gastoPersonas ← Vacío()	▷ $\Theta(1)$
diccPersonas ← Vacío()	▷ $\Theta(1)$
it ← CrearIt(as)	▷ $\Theta(1)$
while haySiguiente?(it) **do**	▷ $\Theta(Alog(A))$
Definir(gastoPersonas, Siguiente(it), 0)	▷ $\Theta(log(A))$
Definir(diccPersonas, Siguiente(it), NULL)	▷ $\Theta(log(A))$
Avanzar(it)	▷ $\Theta(1)$
end while	
gastos ← Vacío()	▷ $\Theta(1)$
masGasto ← Definir(gastos, 0, diccPersonas)	▷ $\Theta(1)$
comproSinDescuento ← Vacío()	▷ $\Theta(1)$
res ← ⟨ ps, as, gastoPersonas, gastos, masGasto, comproSinDescuento ⟩	▷ $\Theta(1)$

Complejidad: $\Theta(Alog(A))$
Justificación: Se construyen las estrucutras utilizadas para representar lolla. Construir los diccionarios gastoPersonas y diccPersonas en base a recorrer todos los elementos de *as* A veces cuesta $\Theta(A(2log(A)) = \Theta(Alog(A))$. Las demas funciones y asignaciones son en $\Theta(1)$.

ObtenerPersonas(**in** *e* : `estr`) → *res* : *itConj*(*persona*)

res ← CrearIt(e.personas)	▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Crea y devuelve un iterador al primer elemento del conjunto que permite recorrerlo. Esto es $\Theta(1)$.

ObtenerPuestos(**in** *e* : `estr`) → *res* : *itDicc*(*idpuesto*, *puesto*)

res ← CrearIt(e.puestos)	▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Crea y devuelve un iterador al primer elemento del diccionario que permite recorrerlo. Esto es $\Theta(1)$.

RegistrarCompra(**in/out** e : `estr`, **in** pi : `idpuesto`, **in** a : `persona`, **in** i : `item`, **in** c : `cant`)

```
p ← Significado(e.puestos, pi)                                              ▷ Θ(log(P))
viejoGasto ← Significado(e.gastoPersonas, a)                                ▷ Θ(log(A))
gasto ← Vender(p, a, i, c)                                                  ▷ Θ(log(A) + log(I))
descuento ← ObtenerDescuento(p, i, c)                                       ▷ Θ(log(I))
Definir(e.gastoPersonas, a, viejoGasto + gasto)                             ▷ Θ(log(A))
//Algoritmo para actualizar (o no) masGasto y e.gastos
if viejoGasto + gasto > SiguienteClave(e.masGasto) then                     ▷ Es nuevo gasto máximo??
    personasMismoGasto ← Vacío()
    Definir(personasMismoGasto, a, NULL)
    e.masGasto ← Definir(e.gastos, viejoGasto + gasto, personasMismoGasto)  ▷ Θ(log(A))
else
    if viejoGasto + gasto == SiguienteClave(e.masGasto) then                ▷ Es igual al gasto máximo??
        tienenMaxGasto ← SiguienteSignificado(e.masGasto)
        Definir(tienenMaxGasto, a, NULL)                                    ▷ Θ(log(A))
    else
        if Definido?(e.gasto, viejoGasto + gasto) then                      ▷ Si no es maximo, está definido??
            personasMismoGasto ← Significado(e.gasto, viejoGasto + gasto)   ▷ Θ(log(A))
            Definir(personasMismoGasto, a, NULL)                            ▷ Θ(log(A))
        else                                                                ▷ Sino está definido, definir el gasto y la persona
            personasMismoGasto ← Vacío()
            Definir(personasMismoGasto, a, NULL)
            Definir(e.gastos, viejoGasto + gasto, personasMismoGasto)       ▷ Θ(log(A))
        end if
    end if
end if
viejoGastoPersonas ← Significado(e.gastos, viejoGasto)                      ▷ Θ(log(A))
if #Claves(viejoGastoPersonas) == 1 then    ▷ Si solo una persona tenía el viejo gasto, borramos la clave.
    Borrar(e.gastos, viejoGasto)                                            ▷ Θ(log(A))
else                                        ▷ Sino, borramos la persona de las personas con mismo gasto.
    Borrar(viejoGastoPersonas, a)                                           ▷ Θ(log(A))
end if
//Algoritmo para actualizar puestos hackeables con item i en la persona a
if descuento == 0 then                      ▷ Si la compra fue sin descuento, es una posible compra hackeable
    itPuesto ← Definir(e.puestos, pi, p)    ▷ Redefino el puesto para guardar un iterador al diccionario. //Θ(log(P))
    if not Definido?(e.comproSinDescuento, a) then   ▷ La persona no compró antes sin descuento? //Θ(log(A))
        puestosSinDescuento ← Vacío()
        Definir(puestosSinDescuento, pi, itPuesto)
        itemsSinDescuento ← Vacío()
        Definir(itemsSinDescuento, i, puestosSinDescuento)
        Definir(e.comproSinDescuento, a, itemsSinDescuento)                 ▷ Θ(log(A))
    else                                    ▷ Si sí, no había comprado ese item sin descuento?
        itemsSinDescuento ← Significado(e.comproSinDescuento, a)            ▷ Θ(log(I))
        if not Definido?(itemsSinDescuento, i) then                         ▷ Θ(log(I))
            puestosSinDescuento ← Vacío()
            Definir(puestosSinDescuento, pi, itPuesto)
            Definir(itemsSinDescuento, i, puestosSinDescuento)              ▷ Θ(log(I))
        else                    ▷ Si sí, solo agregamos el itPuesto al diccionario de puestos hackeables.
            puestosSinDescuento ← Significado(itemsSinDescuento, i)         ▷ Θ(log(I))
            Definir(puestosSinDescuento, pi, itPuesto)                      ▷ Θ(log(P))
        end if
    end if
end if
```

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

Justificación: Buscar el puesto en el diccLog es $\Theta(log(P))$. $Vender$ actualiza los registros dentro del puesto y entrega el gasto de la compra en $\Theta(log(A) + log(I))$. Actualizar e.gastoPersonas requiere buscar el viejo gasto de a y sumar el nuevo en $\Theta(log(A))$ pues también es un diccLog. La estructura e.gastos opera en $\Theta(log(A))$ porque a lo sumo hay tantos gastos como personas, donde A es cantidad de personas. Para actualizar e.gastos hay que ver si el nuevo gasto es mayor o igual al máximo, en ese caso hay que definirlo o agregarlo a las personas con maxGasto, respectivamente. Si es menor, definir el gasto con la persona si no estaba definido o, agregarlo a las personas con mismo gasto si lo estaba. Si la compra fue sin descuento, se debe registrar el idpuesto con su iterador en los diccionarios log de e.comproSinDescuento para a e i, esto es $\Theta(log(P) + log(A) + log(I))$. Todas las operaciones con iteradores son $\Theta(1)$.

Hackear(**in/out** e : `estr`, **in** a : `persona`, **in** i : `item`)

```
diccItems ← Significado(e.sinDescuento, a)                                   ▷ Θ(log(A))
diccPuestos ← Significado(diccItems, i)                                      ▷ Θ(log(I))
it ← CrearIt(diccPuestos)
menorIDPUESTO ← SiguienteClave(it)
puesto ← SiguienteSignificado(SiguienteSignificado(it))   ▷ Obtengo el iterador al puesto y luego obtengo el puesto
precio ← Precio(puesto, i)                                                   ▷ Θ(log(I))
OlvidarItem(puesto, a, i)                                                    ▷ Θ(log(A) + log(I))
gastoTot ← Significado(e.gastoPersonas, a)                                   ▷ Θ(log(A))
Definir(e.gastoPersonas, a, gastoTot - precio)                               ▷ Θ(log(A))
//Algoritmo para actualizar (o no) masGasto y e.gastos
personasMaxGasto ← SiguienteSignficado(e.masGasto)
if #Claves(personasMaxGasto) == 1 ∧ Definido?(personasMaxGasto, a) then      ▷ Θ(log(A))
    //Estaba a en las personas que más gastaron y era la única con ese gasto máximo?
    MaximoAnterior ← AnteriorClave(e.masGasto)   ▷ El it anterior al it máximo es el anterior máximo.
    if gastoTot - precio == MaximoAnterior then   ▷ El nuevo gasto es igual que el maximo anterior?
        tienenNuevoMaxGasto ← AnteriorSignificado(e.masGasto)
        Definir(tienenNuevoMaxGasto, a, NULL)                                ▷ Θ(log(A))
        Retroceder(e.masGasto)
    else
        if gastoTot - precio > MaximoAnterior then   ▷ A pesar del hackeo, el gasto sigue siendo el mayor?
            tienenNuevoMaxGasto ← Vacío()
            Definir(tienenNuevoMaxGasto, a, NULL)
            e.masGasto ← Definir(e.gastos, gastoTot - precio, tienenNuevoMaxGasto)   ▷ Θ(log(A))
        else                                                  ▷ El nuevo gasto ya no es el máximo
            Retroceder(e.masGasto)
            if Definido?(e.gastos, gastoTot - precio) then                   ▷ Θ(log(A))
                personasMismoGasto ← Significado(e.gastos, gastoTot - precio)   ▷ Θ(log(A))
                Definir(personasMismoGasto, a, NULL)                         ▷ Θ(log(A))
            else
                personasMismoGasto ← Vacío()
                Definir(personasMismoGasto, a, NULL)
                Definir(e.gastos, gastoTot - precio, personasMismoGasto)     ▷ Θ(log(A))
            end if
        end if
    end if
else                                                  ▷ El nuevo gasto no es el máximo
    if Definido?(e.gastos, gastoTot - precio) then
        personasMismoGasto ← Significado(e.gastos, gastoTot - precio)        ▷ Θ(log(A))
        Definir(personasMismoGasto, a, NULL)                                 ▷ Θ(log(A))
    else
        personasMismoGasto ← Vacío()
        Definir(personasMismoGasto, a, NULL)
        Definir(e.gastos, gastoTot - precio, personasMismoGasto)             ▷ Θ(log(A))
    end if
end if
viejoGastoPersonas ← Significado(e.gastos, gastoTot)                         ▷ Θ(log(A))
if #Claves(e.gastos, gastoTot) == 1 then   ▷ Si solo una persona tenía el viejo gasto, borramos la clave.
    Borrar(e.gastos, gastoTot)                                               ▷ Θ(log(A))
else                                   ▷ Sino, borramos la persona de las personas con mismo gasto.
    Borrar(viejoGastoPersonas, a)                                            ▷ Θ(log(A))
end if
if not EsHackeable?(puesto, a, i) then   ▷ Si el puesto deja de ser hackeable para i y a //Θ(log(A) + log(I))
    Borrar(diccPuestos, menorIDPUESTO)                                       ▷ Θ(log(P))
end if
```

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

Justificación: El puesto de menor ID se consigue buscando a e i en e.comproSinDescuento en $\Theta(log(A) + log(I))$. CrearIt apunta al it del puesto de menor id en el dicc de puestos. Se actualiza el registro del puesto con $OlvidarItem$. Se actualiza el gasto total de la persona restandole el precio del item hackeado en $\Theta(log(A))$. Si la persona tenía el máximo gasto, se debe ver el it anterior a e.masGasto (asumiento recorrido inorder y que e.masGasto siempre es el último elemento) para ver si sigue siendo mayor , si es igual o si es menor. En los dos últimos casos se debe retroceder el iterador. En el primero, se debe definir el nuevoGasto con a y actualizar el iterador, todo esto es $\Theta(log(A))$. Se suma $log(P)$ sii el puesto deja de ser hackeable para i en a. Las operaciones de iteradores son en $\Theta(1)$.

GastoTotal(**in** e : `estr`, **in** a : `persona`) $\rightarrow$ res : $dinero$

res $\leftarrow$ Significado(e.gastoPersonas, a)
Complejidad: $\Theta(log(A))$
Justificación: El diccionario logarítmico busca el significado de una clave en tiempo $\Theta(log(A))$, siendo A la cantidad de personas (claves).

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

diccPersonas $\leftarrow$ SiguienteSignificado(e.masGasto) ▷ $\Theta(1)$
it $\leftarrow$ CrearIt(diccPersonas) ▷ $\Theta(1)$
res $\leftarrow$ SiguienteClave(it) ▷ $\Theta(1)$
Complejidad: $\Theta(1)$
Justificación: e.masGasto guarda el iterador al elemento del diccionario de e.gasto con el gasto máximo de una persona o un grupo de personas. Como el recorrido del diccLog es inorder, $CrearIt$ entrega un iterador al diccionario con clave de menor id. Esto nos asegura desempatar por id si hay varias personas que comparten el gasto máximo. Todas las operaciones con los iteradores son en $\Theta(1)$.

MenorStock(**in** e : `estr`, **in** i : `item`) $\rightarrow$ res : $idpuesto$

```
it = CrearIt(e.puestos)
hayAlguno ← false
menorID ← SiguienteSignificado(it)
while HaySiguiente?(it) do                                          ▷ Θ(P * log(I))
    if PerteneceAlMenu?(SiguienteSignificado(it), i) then           ▷ Θ(log(I))
        if hayAlguno == false then
            res ← SiguienteClave(it)
            puestoRes ← SiguienteSignificado(it)
            hayAlguno ← true
        else
            stockRes ← Stock(puestoRes, i)                          ▷ Θ(log(I))
            stockNuevo ← Stock(SiguienteSignificado(it), i)         ▷ Θ(log(I))
            if nuevo < actual then
                res ← SiguienteClave(it)
                puestoRes ← SiguienteSignificado(it)
            else
                if nuevo == actual ∧ SiguienteSignificado(it) < puestoRes then
                    res ← SiguienteClave(it)
                    puestoRes ← SiguienteSignificado(it)
                end if
            end if
        end if
    end if
    if SiguienteSignificado(it) < menorID then
        menorID ← SiguienteSignificado(it)
    end if
    Avanzar(it)
end while
if hayAlguno == false then
    res ← menorID
end if
```

Complejidad: $\Theta(P * log(I))$
Justificación: Se hacen P iteraciones del while, donde P es la cantidad de puestos. Preguntar si el item pertenece al menú es $\Theta(log(I))$. Se pregunta por su stock y se lo compara con el anterior en $\Theta(log(I))$. Se va guardando en res el puesto con menor stock, y si empatan, el de menor ID. Luego, el algoritmo cuesta $\Theta(P * log(I))$.
En el caso de que no haya ningun puesto con el item en su menú, "hayAlguno.^{es} false y se entrega el puesto con el menor ID, implicando que el stock es 0 en todos los puestos.
Todas las operaciones con iteradores son en $\Theta(1)$.

---

2. Módulo PuestoDeComida

Interfaz

se explica con: PuestoDeComida

géneros: `puesto`.

Operaciones básicas de PuestoDeComida

CrearPuesto(**in** p: `dicc(item, dinero)`, **in** s: `dicc(item, nat)`, **in** d: `dicc(item, dicc(cant, nat))`) $\rightarrow$ res : `puesto`
Pre $\equiv$ {claves(p) = claves(s) $\wedge$ claves(d) $\subseteq$ claves(p)}
Post $\equiv$ {$res =_{\text{obs}}$ crearPuesto(p,s, d)}
Complejidad: $\Theta(I * cant * log(I))$
Descripción: Inicializa un puesto dada la información suministrada por el usuario.

ObtenerStock(**in** p: `puesto`, **in** i: `item`) $\rightarrow$ res : `nat`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {$res =_{\text{obs}}$ stock(p, i)}
Complejidad: $\Theta(log(I))$
Descripción: Da el stock de un item i en el puesto p.
Aliasing: Se devuelve una referencia no modificable al stock de un ítem.

ObtenerDescuento(**in** p: `puesto`, **in** i: `item`, **in** *cantidad*: `nat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {$res =_{\text{obs}}$ descuento(p, i, *cantidad*)}
Complejidad: $\Theta(log(I))$
Descripción: Da el descuento en el puesto p de un item i para una *cantidad* dada.
Aliasing: Se devuelve una referencia no modificable al descuento de un ítem.

ObtenerGastoPersona(**in** p: `puesto`, **in** a: `persona`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {$True$}
Post $\equiv$ {$res =_{\text{obs}}$ gastosDe(p, a)}
Complejidad: $\Theta(log(A))$
Descripción: Da el gasto realizado por una persona a en el puesto p.
Aliasing: Se devuelve una referencia no modificable al gasto de una persona.

Vender(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** c: `cant`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {$p =_{\text{obs}}$ p_0 $\wedge$ haySuficiente?(p,i,c)}
Post $\equiv$ {$p =_{\text{obs}}$ vender(p_0, a, i, c) $\wedge_{\text{L}}$ $res =_{\text{obs}}$ gastosDe1Venta(p, $\langle$i, c$\rangle$)}
Complejidad: $\Theta(log(A) + log(I))$
Descripción: Realiza una venta a la persona a del item i en c cantidades en el puesto p. Devuelve el gasto total de la venta realizada.
Aliasing: Se devuelve una referencia no modificable al gasto de una venta.

OlvidarItem(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`)
Pre $\equiv$ {$p =_{\text{obs}}$ p_0 $\wedge$ i $\in$ menu(p) $\wedge_{\text{L}}$ consumioSinPromo?(p, a, i)}
Post $\equiv$ {$p =_{\text{obs}}$ olvidarItem(p_0, a, i)}
Complejidad: $\Theta(log(A) + log(I))$
Descripción: Olvida el item i que la persona a compró sin descuento en el puesto p.

PerteneceAlMenu?(**in** p: `puesto`, **in** i: `item`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = true $\iff$ i $\in$ menu(p)}
Complejidad: $\Theta(log(I))$
Descripción: Devuelve true sí y solo sí el ítem i pertenece al menú de p.

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: $\Theta(log(I))$
Descripción: Devuelve el precio del ítem i en del menú de p.
Aliasing: Se devuelve una referencia no modificable al precio de un ítem.

EsHackeable?(**in** p: `puesto`, **in** a: `persona`, **in** i: `item`) $\rightarrow$ res : `bool`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {res = true $\iff$ consumioSinPromo?(p, a, i)}
Complejidad: $\Theta(log(A) + log(I))$
Descripción: Devuelve true sí y solo sí la persona a compró el ítem i sin descuento en el puesto.

Representación

Representación del PuestoDeComida

El puesto de comida se decidió representar con las siguientes estructuras:

- *menu*, un diccionario avl que tiene de claves todos los items que vende el puesto, y como signficado su precio. El dicc avl nos permite buscar el precio de un item en $\Theta(log(I))$;
- *stock*, un diccionario avl que tiene de claves todos los items que vende el puesto, y como signficado su stock. El dicc avl nos permite buscar y definir el stock de un item en $\Theta(log(I))$. Los items definidos en *stock* deben ser exactamente los mismos que los de *menu*.;
- *descuentos*, un diccionario avl que tiene de claves todos los items que tienen descuentos en el puesto, y como signficado un vector donde cada uno de sus elementos representa el descuento dado a la cantidad representada por su posición. Por ejemplo, el vector [0, 0, 5, 5, 10] nos indica que para 2 y 3 items hay 5 % de descuento, y 10 % para 4 o más items. El dicc avl nos permite buscar el vector de descuentos de un item en $\Theta(log(I))$. El vector debe estar ordenado y tener el último elemento como el mayor de todos. El vector nos permite acceder a cada descuento en $\Theta(1)$;
- *gastosPersonas*, un diccionario avl que tiene de claves todas las personas que hayan hecho alguna compra en el puesto, y como significado su gasto total realizado. El dicc avl nos permite buscar y definir los gastos de las personas que realicen alguna compra en $\Theta(log(A))$;
- *ventasConDescuento*, un diccionario avl que tiene de claves todas las personas que hayan hecho alguna compra con descuento en el puesto, y como significado una lista de tupla(item, nat) que registra cada una de las compras. Se decidió separar el registro de ventas en dos estructuras para facilitar el hackeo de una venta sin descuento, ya que aquellas ventas con descuento no necesitaremos modificarlas. En consecuencia, las almacenamos en la lista que nos permite agregar rápidamente en $\Theta(1)$, con el único costo de buscar a la persona en $\Theta(log(A))$;
- *ventasSinDescuento*, un diccionario avl que tiene de claves todas las personas que hayan hecho alguna compra sin descuento en el puesto, y como significado otro diccionario de los items comprados donde se obtiene un vector. Este vector almacena en cada nuevo elemento la cantidad de una nueva compra del item sin descuento. Solo necesitamos agregar al final, eliminar o modificar el último elemento del vector, cuyo costo es $\Theta(1)$. El tiempo en peor caso de buscar el vector es $\Theta(log(A) + log(I))$.

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

donde `estr` es `tupla(` *menu*: `diccLog(item, dinero)`
, *stock*: `diccLog(item, nat)`
, *descuentos*: `diccLog(item, vector(nat))`
, *gastosPersonas*: `diccLog(persona, dinero)`
, *ventasConDescuento*: `diccLog(persona, lista(⟨item, nat⟩))`
, *ventasSinDescuento*: `diccLog(persona, diccLog(item, vector(nat)))`)
donde `persona` es `nat`
donde `idpuesto` es `nat`
donde `item` es `nat`

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ 1 $\wedge_{\text{L}}$ 2 $\wedge_{\text{L}}$ 3 $\wedge$ (4 $\vee$ 5 $\vee$ 6) $\wedge_{\text{L}}$ 7 $\wedge$ 8
1 $\equiv$ claves(e.menu) = claves(e.stock) $\wedge_{\text{L}}$ claves(e.descuentos) $\subseteq$ claves(e.menu)
2 $\equiv$ ($\forall i$: item)(def?(i, e.descuentos) $\Rightarrow_{\text{L}}$ $\neg$vacía?(obtener(i, e.descuentos)) $\wedge_{\text{L}}$ estaOrdenado(obtener(i, e.descuentos)) $\wedge$ ultimoSinRepetirYNoEsCero(obtener(i, e.descuentos)))

3 $\equiv$ $(\forall p$: persona)(def?(p, e.gastosPersonas) $\Rightarrow_{\text{L}}$ (def?(p, e.ventasConDescuento) $\wedge$ $\neg$def?(p, e.ventasSinDescuento) $\wedge_{\text{L}}$ obtener(p, e.gastosPersonas) = gastoTotConDesc(obtener(p, e.ventasConDescuento), e.descuentos, e.menu)) $\vee$ ($\neg$def?(p, e.ventasConDescuento) $\wedge$ def?(p, e.ventasSinDescuento) $\wedge_{\text{L}}$ obtener(p, e.gastosPersonas) = gastoTotSinDesc(obtener(p, e.ventasSinDescuento), e.menu) $\vee$ (def?(p, e.ventasConDescuento) $\wedge$ def?(p, e.ventasSinDescuento) $\wedge_{\text{L}}$ obtener(p, e.gastosPersonas) = gastoTotConDesc(obtener(p, e.ventasConDescuento), e.descuentos, e.menu))) + gastoTotSinDesc(obtener(p, e.ventasSinDescuento), e.menu))

4 $\equiv$ $(\forall p$: persona)((def?(p, e.ventasConDescuento) $\wedge$ $\neg$def?(p, e.ventasSinDescuento)) $\Rightarrow_{\text{L}}$ def(p, e.gastosPersonas) $\wedge_{\text{L}}$ obtener(p, e.gastosPersonas) = gastoTotConDesc(obtener(p, e.ventasConDescuento), e.descuentos, e.menu))

5 $\equiv$ $(\forall p$: persona)(($\neg$def?(p, e.ventasConDescuento) $\wedge$ def?(p, e.ventasSinDescuento)) $\Rightarrow_{\text{L}}$ def(p, e.gastosPersonas) $\wedge_{\text{L}}$ obtener(p, e.gastosPersonas) = gastoTotSinDesc(obtener(p, e.ventasSinDescuento), e.menu))

6 $\equiv$ $(\forall p$: persona)((def?(p, e.ventasConDescuento) $\wedge$ def?(p, e.ventasSinDescuento)) $\Rightarrow_{\text{L}}$ def(p, e.gastosPersonas) $\wedge_{\text{L}}$ obtener(p, e.gastosPersonas) = gastoTotConDesc(obtener(p, e.ventasConDescuento), e.descuentos, e.menu) + gastoTotSinDesc(obtener(p, e.ventasSinDescuento), e.menu))

7 $\equiv$ $(\forall p$: persona)(def?(p, e.ventasConDescuento) $\Rightarrow_{\text{L}}$ ($\forall t$: $\langle$item, nat$\rangle$) (está?(obtener(p, e.ventasConDescuento)) $\Rightarrow_{\text{L}}$ def?(π_1(t), e.descuentos) $\wedge_{\text{L}}$ ((π_2(t) < long(obtener(π_1(t), e.descuentos)) $\wedge_{\text{L}}$ obtener(π_1(t), e.descuentos)[π_2(t)] > 0) $\vee$ π_2(t) $\geq$ long(obtener(π_1(t), e.descuentos)))))

8 $\equiv$ $(\forall p$: persona)((def?(p, e.ventasSinDescuento) $\Rightarrow_{\text{L}}$ ($\forall i$: item))(def?(i, e.ventasSinDescuento) $\Rightarrow_{\text{L}}$ ($\forall n$: nat)(está?(n, obtener(i, e.ventasSinDescuento)) $\Rightarrow_{\text{L}}$ (def?(i, e.descuentos) $\wedge_{\text{L}}$ obtener(i, e.descuentos)[n] = 0) $\vee$ $\neg$def?(i, e.descuentos))))

gastoTotConDesc : secu($\langle$item, nat$\rangle$) l $\times$ dicc(item, vector(nat)) d $\times$ dicc(item, dinero) m $\longrightarrow$ dinero {claves(d) $\subseteq$ claves(m) $\wedge$ ($\forall t$: $\langle$item, nat$\rangle$)(t $\in$ l $\Rightarrow_{\text{L}}$ def?(π_1(t), m))}

gastoTotSinDesc : dicc(item, vector(nat)) c $\times$ dicc(item, dinero) m $\longrightarrow$ dinero {claves(c) $\subseteq$ claves(m)}

sumarElementos : secu(nat) $\longrightarrow$ nat

gastoTotConDesc(l, d, m) $\equiv$ **if** vacia?(l) **then**
0
else
if def?(π_1(prim(l)), d) **then**
if π_2(prim(l)) < long(π_1(prim(l)), d) **then**
aplicarDescuento(obtener(π_1(prim(l)), m) * π_2(prim(l)), obtener(π_1(prim(l)), d)[π_2(prim(l))]) + gastoTotConDesc(fin(l), d, m)
else
aplicarDescuento(obtener(π_1(prim(l)), m) * π_2(prim(l)), obtener(π_1(prim(l)), d)[long(π_1(prim(l)), d) - 1]) + gastoTotConDesc(fin(l), d, m)
fi
else
obtener(π_1(prim(l)), m) * π_2(prim(l)) + gastoTotConDesc(fin(l), d, m)
fi
fi

gastoTotSinDesc(c, m) $\equiv$ **if** #claves(c) = 0 **then**
0
else
sumarElementos(obtener(dameUno(claves(c)), c)) * obtener((dameUno(claves(c)), m) + gastoTotSinDesc(borrar(dameUno(claves(c)), c), m)
fi

sumarElementos(s) $\equiv$ **if** vacia?(fin(s)) **then** prim(s) **else** prim(s) + sumarElementos(fin(s)) **fi**

estaOrdenado(v) $\equiv$ ($\forall k$: nat)(0 $\leq$ i < long(v) - 1 $\Rightarrow_{\text{L}}$ v[i] $\leq$ v[i + 1])

ultimoSinRepetirYNoEsCero(v) $\equiv$ ($\forall k$: nat)(0 $\leq$ i < long(v) - 1 $\Rightarrow_{\text{L}}$ v[i] $\neq$ v[long(v) - 1]) $\wedge$ v[long(v) - 1] $\neq$ 0

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

Abs(e) $=_{\text{obs}}$ p: puesto | menu(p) $=_{\text{obs}}$ claves(e.menu) $\wedge_{\text{L}}$
($\forall i$: item)(def?(i, e.menu) $\Rightarrow_{\text{L}}$ precio(p, i) $=_{\text{obs}}$ obtener(i, e.menu) $\wedge$
stock(p, i) $=_{\text{obs}}$ obtener(i, e.stock) $\wedge$
($\forall c$: nat)(descuento(p, i, c) $=_{\text{obs}}$ buscarPromo(e.descuentos, i, c))) $\wedge$
($\forall a$: persona)((def?(p, e.ventasConDescuento) $\wedge$ def?(p, e.ventasSinDescuento) $\Rightarrow_{\text{L}}$
ventas(p, a) $=_{\text{obs}}$ listaAMulticonj(obtener(p, e.ventasConDescuento)) $\cup$
diccVecAMulticonj(obtener(p, e.ventasSinDescuento))) $\vee$
(def?(p, e.ventasConDescuento) $\wedge$ $\neg$def?(p, e.ventasSinDescuento) $\Rightarrow_{\text{L}}$
ventas(p, a) $=_{\text{obs}}$ listaAMulticonj(obtener(p, e.ventasConDescuento))) $\vee$
($\neg$def?(p, e.ventasConDescuento) $\wedge$ def?(p, e.ventasSinDescuento) $\Rightarrow_{\text{L}}$
ventas(p, a) $=_{\text{obs}}$ diccVecAMulticonj(obtener(p, e.ventasSinDescuento))) $\vee$

$(\neg \text{def?}(p, e.\text{ventasConDescuento}) \wedge \neg \text{def?}(p, e.\text{ventasSinDescuento}) \Rightarrow_{\text{L}} \text{ventas}(p, a) =_{\text{obs}} \emptyset))$

buscarPromo : dicc(item, secu(nat)) × item × nat ⟶ nat
listaAMulticonj : secu(⟨item, nat⟩) ⟶ multiconj(⟨ item, nat ⟩)
diccVecAMulticonj : dicc(item, secu(nat)) ⟶ multiconj(⟨ item, nat ⟩)
vectorAMulticonj : secu(nat) × item ⟶ multiconj(⟨ item, nat ⟩)

```
buscarPromo(d, i, c) ≡ if ¬def?(i, d) then
                           0
                       else
                           if c < long(obtener(i, d)) then
                               obtener(i, d)[c]
                           else
                               obtener(i, d)[long(obtener(i, d)) - 1]
                           fi
                       fi
listaAMulticonj(s) ≡ if vacía?(s) then ∅ else Ag(prim(s), listaAMulticonj(fin(s))) fi
diccVecAMulticonj(d) ≡ if #claves(d) = 0 then
                           ∅
                       else
                           vectorAMulticonj(obtener(dameUno(claves(d)), d), dameUno(claves(d))) ∪
                           diccVecAMulticonj(borrar(dameUno(claves(d))))
                       fi
vectorAMulticonj(v, i) ≡ if vacía?(v) then ∅ else Ag(⟨ i, prim(v) ⟩, vectorAMulticonj(fin(s), i)) fi
```

Algoritmos

CrearPuesto(**in** p: `dicc(item, dinero)`, **in** s: `dicc(item, nat)`, **in** d: `dicc(item, dicc(cant, nat))`) $\rightarrow$ res : $estr$

```
descuentos ← Vacío()                                                          ▷ Θ(1)
//Creo un iterador para iterar sobre los items del diccionario d
itItem ← crearIt(d)                                                           ▷ Θ(1)
while haySiguiente?(itItem) do                                                ▷ Θ(I)
    //Creo un iterador para iterar sobre los descuentos del item actual
    itD ← crearIt(siguienteSignificado(itItem))                               ▷ Θ(1)
    if haySiguiente?(itD) then
        //Si hay algún descuento para este item, inicializamos el vector v, cuyo índice representará la cantidad del
        //item, y cuyos elementos representarán el descuento aplicable para esa cantidad
        v ← Vacía()                                                           ▷ Θ(1)
        //El primer elemento que me devuelve el iterador será el mínimo descuento posible, pues recorremos inorder
        //Previo a esa cantidad, todos los otros descuentos son 0. Lleno al vector v con 0s hasta el mínimo descuento
        i ← 0
        while i < SiguienteClave(itD) do                                      ▷ Θ(cant)
            AgregarAtras(v, 0)                                                ▷ Θ(1)
            i++                                                               ▷ Θ(1)
        end while
        //Me encargo de los n-1 elementos que comprenden descuentos ≠ 0. Voy mirando "hacia adelante".
        //Si luego del elemento siguiente no hay nada, salgo del while
        while haySiguiente?(Avanzar(itD)) do                                  ▷ Θ(cant)
            //Creo una variable auxiliar para guardar la clave actual
            k ← SiguienteClave(itD)                                           ▷ Θ(1)
            //Incremento a la variable k hasta alcanzar el próximo descuento, y mientras tanto voy llenando al vector
            //v hasta dicho descuento próximo
            while k < siguienteClave(Avanzar(itD)) do                         ▷ Θ(cant)
                AgregarAtras(v, siguienteSignificado(it))                     ▷ Θ(1)
                k++                                                           ▷ Θ(1)
            end while
            //Avanzo al siguiente descuento
            Avanzar(itD)
        end while
        //Sólo queda encargarme del elemento n-esimo de v (el último). Este será cant, el descuento máximo, y
        //corresponde al último elemento del diccionario recorrido inorder, que coincide con pedir el siguiente de
        //donde esta parado ahora el iterador. Lo hago, y anexo dicho elemento a v:
        AgregarAtras(v, siguienteSignificado(it))                             ▷ Θ(1)
        //Agrego este item y sus respectivos descuentos a la estructura descuento
        Definir(descuentos, SiguienteClave(itItem), v)                        ▷ Θ(log(I))
    end if
    //Si no entro al if, entonces dicho item no tiene descuentos y por lo tanto no lo incluimos en el diccionario.
    //Avanzo al siguiente item
    Avanzar(itItem)
end while
//inicializo a las estructuras ventasConDescuento, ventasSinDescuento y gastosPersonas. Estas se van a llenar a
//medida que se realicen ventas
ventasConDescuento ← Vacío()                                                  ▷ Θ(1)
ventasSinDescuento ← Vacío()                                                  ▷ Θ(1)
gastosPersonas ← Vacío()                                                      ▷ Θ(1)
res ← ⟨ p, s, descuentos, gastosPersonas, ventasConDescuento, ventasSinDescuento ⟩   ▷ Θ(1)
```

Complejidad: $\Theta(I * cant * log(I))$

Justificación: Utilizamos los diccionarios p y s provistos por el usuario para inicializar directamente las estructuras *menu* y *stock* en $\Theta(1)$. Luego, para la estructura *descuentos* iteramos por todos los items en $\Theta(log(I))$, y para cada item armamos un vector v en $\Theta(cant)$, que es el vector que tiene los descuentos para una cantidad dada del item, y que tiene largo *cant*, siendo *cant* el descuento máximo para ese item. A continuación definimos en $\Theta(log(I))$ el par clave-signficado con ese item y el vector v en la estructura *descuentos*. Finalmente, inicializamos las estructuras *ventasConDescuento*, *ventasSinDescuento* y *gastosPersonas* en $\Theta(1)$.

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

res $\leftarrow$ Significado(e.stock, i) ▷ $\Theta(log(I))$

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

Justificación: Accedemos, en el diccionario del stock, al significado de la clave indicada por el usuario, en $\Theta(log(I))$.

ObtenerDescuento(**in** e : `estr`, **in** i : `item`, **in** $cantidad$: `nat`) $\rightarrow$ res : nat

if Definido?(e.descuentos, i) **then** ▷ $\Theta log(I)$

//si el item tiene descuento, calculamos el largo del vector de descuentos

longVect $\leftarrow$ Longitud(Significado(e.descuentos, i)) ▷ $\Theta log(I)$

if cantidad $<$ longVect **then**

//si la cantidad pedida por el usuario es menor al largo del vector de descuentos, devolvemos el descuento

//por indexación

res $\leftarrow$ Significado(e.descuentos, i)[cantidad] ▷ $\Theta log(I)$

else

//si la cantidad pedida por el usuario supera el largo del vector de descuento, devolvemos el máximo

//descuento posible para ese item, que se encuentra en la ultima posición del vector de descuentos

res $\leftarrow$ Significado(e.descuentos, i)[longVect - 1] ▷ $\Theta log(I)$

end if

else

//si el item no está en descuentos, devolvemos 0

res $\leftarrow$ 0 ▷ $\Theta(1)$

end if

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

Justificación: Accedemos, en la estructura *descuentos*, al significado de la clave indicada por el usuario, en $\Theta(log(I))$. Luego, con el vector devuelto accedemos en $\Theta(1)$ a la posición correspondiente a la cantidad indicada por el usuario, para así devolverle el descuento pedido. Si la cantidad supera a la cantidad máxima para la que hay descuento, devolvemos el máximo descuento posible, accediendo a la ultima posición del vector de descuentos.

ObtenerGastoPersona(**in** e : `estr`, **in** a : `persona`) $\rightarrow$ res : $dinero$

if Definido?(e.gastosPersona, a) **then** ▷ $\Theta(log(A))$

//si la persona esta en gastosPersona, devolvemos el gasto realizado por esta

res $\leftarrow$ Significado(e.gastosPersona, a) ▷ $\Theta(log(A))$

else

//si la persona aun no compro, devolvemos 0

res $\leftarrow$ 0 ▷ $\Theta(1)$

end if

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

Justificación: Accedemos, en el diccionario de los gastos de personas, al significado de la clave indicada por el usuario, en $\Theta(log(A))$. En caso de que aún no haya realizado compras, devolvemos 0.

Vender(**in/out** e : `estr`, **in** a : `persona`, **in** i : `item`, **in** c : `cant`) $\rightarrow$ res : $dinero$

```
precio ← Significado(e.menu, i)                                          ▷ Θ(log(I))
descuento ← ObtenerDescuento(p, i)                                       ▷ Θ(log(I))
stock ← ObtenerStock(p, i)                                               ▷ Θ(log(I))
Definir(e.stock, i, stock - cant)                                        ▷ Θ(log(I))
gasto ← (precio * c) - (precio * c * descuento/100)
if Definido?(e.gastoPersonas, a) then
    gastoViejo ← Significado(e.gastosPersonas, a)                        ▷ Θ(log(A))
    Definir(e.gastosPersonas, a, gasto + gastoViejo)                     ▷ Θ(log(A))
else
    Definir(e.gastosPersonas, a, gasto)                                  ▷ Θ(log(A))
end if
if descuento == 0 then          ▷ Debemos decidir si guardar la venta en la estructura con o sin descuento
    if not Definido?(e.ventasSinDescuento, a) then                       ▷ Θ(log(A))
        vector ← Vacía()
        AgregarAtras(vector, c)                                          ▷ Θ(1)
        dicc ← Vacío()
        Definir(dicc, i, vector)                                         ▷ Θ(log(I))
        Definir(e.ventasSinDescuento, a, dicc)                           ▷ Θ(log(A))
    else
        itemsComprados ← Significado(e.ventasSinDescuento, a)            ▷ Θ(log(A))
        if not Definido?(itemsComprados, i) then
            vector ← Vacía()
            AgregarAtras(vector, c)                                      ▷ Θ(1)
            Definir(itemsComprados, i, vector)                           ▷ Θ(log(I))
        else
            vector ← Significado(itemsComprados, i)                      ▷ Θ(log(I))
            AgregarAtras(vector, c)                                      ▷ Θ(1)
        end if
    end if
else
    if Definido?(e.ventasConDescuento, a) then                           ▷ Θ(log(A))
        listaVentas ← Significado(e.ventasConDescuento, a)               ▷ Θ(log(A))
        AgregarAtras(listaVentas, ⟨i, c⟩)                                ▷ Θ(1)
    else
        listaVentas ← Vacía()
        AgregarAtras(listaVentas, ⟨i, c⟩)                                ▷ Θ(1)
        Definir(e.ventasConDescuento, a, listaVentas)                    ▷ Θ(log(A))
    end if
end if
```

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

Justificación: En una venta se debe averiguar el precio, el descuento y el stock del item ($\Theta(log(I))$). Al stock se le debe restar la cantidad comprada en $\Theta(log(I))$. Se calcula el nuevo gasto y sumarlo al gasto anterior de la persona en $\Theta(log(A))$.

Si la compra fue realizada sin descuento, se la debe registrar en e.ventasSinDescuento para la persona a, creando las estructuras necesarias si no estaban antes creadas. Cada posición del vector representa compras distintas del mismo item sin descuento, y el elemento la cantidad comprada, por lo que cumple con la representación del multiconjunto del TAD Puesto. La elección del vector facilita poder hackearlo en $\Theta(1)$. En el peor caso esto es $\Theta(log(A) + log(I))$.

Si la compra no fue realizada con descuento, se la registra como tupla $(item, cant)$ en la lista correspondiente a a, creando las estructuras necesarias si no estaban antes creadas. La lista permite repetidos. por lo que cumple con la representación del multiconjunto del TAD Puesto. Los items no hackeables nunca son modificados, por lo que solo nos importa que la complejidad de agregar en una lista es $\Theta(1)$. En el peor caso esto es $\Theta(log(A) + log(I))$.

OlvidarItem(**in/out** e : `estr`, **in** a : `persona`, **in** i : `item`)

```
stock ← ObtenerStock(p, i)                                                          ▷ Θ(log(I))
Definir(e.stock, i, stock + 1)                                                      ▷ Θ(log(I))
itemsSinDescuento ← Significado(e.ventasSinDescuento, a)                            ▷ Θ(log(A))
vectorVentas ← Significado(itemsSinDescuento, i)                                    ▷ Θ(log(I))
//Olvidamos un ítem de la última compra de ese item sin descuento
vectorVentas[Longitud(vectorVentas) - 1] ← vectorVentas[Longitud(vectorVentas) - 1] - 1   ▷ Θ(1)
if vectorVentas[Longitud(vectorVentas) - 1] == 0 then   ▷ Al hackear, quedó en 0 la cantidad de items vendidos?
    Comienzo(vectorVentas)                                          ▷ Entonces borramos la venta. //Θ(1)
    if Longitud(vectorVentas) == 0 then                  ▷ Borramos la clave i si el vector quedó vacío
        Borrar(itemsSinDescuento, i)                                                ▷ Θ(log(I))
        if #Claves(itemsSinDescuento) == 0 then          ▷ Borramos la clave a si el diccionario quedó vacío
            Borrar(e.ventasSinDescuento, a)                                         ▷ Θ(log(A))
        end if
    end if
end if
```

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

Justificación: Se repone un item en el stock en $\Theta(log(I))$. Se accede al vector de e.ventasSinDescuento correspondiente a i y a, recorriendo el diccLog en $\Theta(log(A) + log(I))$. Se resta un uno en el último elemento del vector, eliminando de esta forma el ítem comprado por la persona. Si el último elemento queda en 0, significa que hay que eliminar la compra. Si el vector queda vacío, significa que a nunca compró i sin descuento, por lo que se borra i del diccionario. Si el diccionario queda vacío, significa que a no compró ningún ítem sin descuento en el puesto, por lo que se borra a de e.ventasSinDescuento. De esta forma, es como si el ítem nunca se hubiese comprado. Esto es en peor caso $\Theta(log(A) + log(I))$.

PerteneceAlMenu?(**in** e : `estr`, **in** i : `item`) → res : $bool$

```
res ← Definido?(e.menu, i)                                                          ▷ Θ(log(I))
```

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

Justificación: La búsqueda de una clave i en el diccionario logarítmico $e.menu$ tiene un costo de $\Theta(log(I))$.

Precio(**in** e : `estr`, **in** i : `item`) → res : $dinero$

```
res ← Significado(e.menu, i)                                                        ▷ Θ(log(I))
```

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

Justificación: La búsqueda de un significado de una clave i en el diccionario logarítmico $e.menu$ tiene un costo de $\Theta(log(I))$.

EsHackeable?(**in** e : `estr`, **in** a : `persona`, **in** i : `item`) → res : $bool$

```
if Definido?(e.ventasSinDescuento, a) then                                          ▷ Θ(log(A))
    itemsComprados ← Signficado(e.ventasSinDescuento, a)                            ▷ Θ(log(A))
    if Definido?(itemsComprados, i) then                                            ▷ Θ(log(I))
        res ← true
    else
        res ← false
    end if
else
    res ← false
end if
```

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

Justificación: Se busca la persona a en el diccionario de ventas sin descuento en $\Theta(log(A))$. Si está definida, quiere decir que compró algún ítem sin descuento en el puesto. Buscamos entonces en el diccionario el ítem i en $\Theta(log(I))$. Si está definido devolvemos true, pues existe una compra hackeable para a de i. En el caso en que ninguno de los dos esté definido, quiere decir que no existe una compra hackeable para a de i, por lo que devolvemos false.