# 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: JUSCIKTYMTQNAEURXVZQ**



| Integrante | LU | Correo electrónico |
|---|---|---|
| Bustos, Juan | 19/22 | juani8.bustos@gmail.com |
| Dominguez, Leonardo | 285/22 | leodomingue2016@gmail.com |
| Nandín, Matías | 227/22 | imatinandin@gmail.com |
| Marín, Candela Emilia | 1405/21 | canmarin17@gmail.com |



**Reservado para la cátedra**

| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |

# 1. Módulo Lollapatuza

## Interfaz

**se explica con**: LollaPatuza

**géneros**: `lolla`

**Servicios Usados**:
**DiccLog** :
**Vacío**() → res : dicc(q, b)
Complejidad: (1)
Descripción: genera un diccionario vacío.

**Definir**(in/out d : dicc(q, b), in k : q, in s : b) → res : itDicc(q, b)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)) + copy(k) + copy(s))$, donde K = claves(d)
Descripción: Define la clave k con el signicado s en el diccionario. Retorna un iterador al elemento recién agregado.
Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

**Definido?**(in d : dicc(q, b), in k : q) → res : bool
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: devuelve true si y sólo k está definido en el diccionario.

**Significado**(in d : dicc(q, b), in k : q) → res : b
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

**Borrar**(in/out d : dicc(q, b), in k : q)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: elimina la clave k y su significado de d.

**Copiar**(in d : dicc(q, b)) → res : dicc(q, b)
Complejidad: $\theta(\sum_{k' \in K}(copy(k) + copy(significado(k,d))))$, donde K = claves(d)
Descripción: genera una copia nueva del diccionario.

**Lista Enladaza** :
**Vacía**() → res : lista($\alpha$)
$Complejidad : \theta(1)$
Descripción: genera una lista vacıa

**agregarAtras**(in/out l : lista($\alpha$), $in\ a : \alpha$) → $res : itLista(\alpha)$
$Complejidad : \theta(copy(a))$
Descripción: agrega el elemento a como ultimo elemento de la lista. Retorna un iterador a l, de forma tal que Siguiente devuelva a.

**Primero**(in l : lista($\alpha$)) → $res : \alpha$
$Complejidad : \theta(1)$
Descripción: devuelve el primer elemento de la lista.
Aliasing: res es modificable si y solo si l es modificable.

**Fin**(in/out l : lista($\alpha$))
$Complejidad : \theta(1)$
Descripción: Elimina el primero elemento de l

**Longitud**(in l : lista($\alpha$)) $\rightarrow$ $res$ : $nat$
$Complejidad$ : $\theta(copy(a))$
Descripción: Devuelve la cantidad de elementos que tiene la lista

**Copiar**(in l : lista($\alpha$)) $\rightarrow$ $res$ : $lista(\alpha)$
$Complejidad$ : $\theta(\sum_{i=1}^{t} copy(l[i]))$, donde t = long(l).
Descripción: Genera una copia nueva de la lista

**Vector** :
**Vacía**() $\rightarrow$ res : vector($\alpha$)
$Complejidad$ : $\theta(1)$
Descripción: Genera un vector vacıo

**Longitud**(in v : vector($\alpha$)) $\rightarrow$ $res$ : $nat$
$Complejidad$ : $\theta(1)$
Descripción: Devuelve la cantidad de elementos que contiene el vector

**Copiar**(in v : vector($\alpha$)) $\rightarrow$ $res$ : $vector(\alpha)$
$Complejidad$ : $\theta(\sum_{i=1}^{l} copy(v[i]))$, donde l = long(v).
Descripción:Genera una copia nueva del vector

**AgregarAtras**(in/out v : vector($\alpha$), $in\ i$ : $nat$)
$Complejidad$ : $\theta(f(long(v)) + long(v) - i + copy(a)))$.
Descripción: agrega el elemento a como ultimo elemento del vector.
Aliasing: El elemento a se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando long)v= es potencia de 2.

**•[•]**(in v : vector($\alpha$), $ini$ : $nat$) $\rightarrow$ $res$ : $\alpha$
$Complejidad$ : $\theta(1)$.
Descripción: devuelve el elemento que se encuentra en la i- esima posicion del vector en base 0. Es decir, v[i] devuelve el elemento que se encuentra en la posici on i + 1.
Aliasing: res es modificable si y s olo si v es modificable.

## Operaciones básicas de Lollapatuza

PERSONAS(**in** $l$ : `lolla`) $\rightarrow$ $res$ : `listaEnlazada (persona)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{\text{obs}}$ Personas(l)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve un conjunto con la cantidad de personas que hay en lolla
**Aliasing:** res es modificable si y solo si diccPersona es modificable

PUESTOS(**in** $l$ : `lolla`) $\rightarrow$ $res$ : `listaEnlazada(tupla<ID, puesto>)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{\text{obs}}$ Puestos(l)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve un diccionario con todos los puestos y su respectivo ID que hay en lolla
**Aliasing:** res es modificable si y solo si los dIccPuestos son modificablen

CREARLOLLA(**in** $ps$ : `dicc(id, puesto)`, **in** $as$ : `conj(persona)`) $\rightarrow$ $res$ : `lolla`
**Pre** $\equiv$ {($\neg$ $\emptyset$? (claves(ps)) $\wedge$ $\neg$ $\emptyset$? (as)) $\wedge_{\text{L}}$ vendenAlMismoPrecio(ps) $\wedge$ noVendieronAun(ps)}
**Post** $\equiv$ {res $=_{\text{obs}}$ CrearLolla(ps,as)}
**Complejidad:** $\Theta$(P * (I + log($P$) + $\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item)) + A*\log(A)$
**Descripción:** Dado un diccionario con todos los ID y sus respectivos puestos y un conjunto de personas, crea una nueva instancia de Lollapatuza.
**Aliasing:** No hay aliasing ya que se crea por copia cada una de las estructuras

VENDER(**in/out** $l$ : `lolla`, **in** $ID$ : `nat`, **in** $a$ : `personas`, **in** $i$ : `item`, **in** $cant$ : `nat`) $\rightarrow$ $res$ : `lolla`
**Pre** $\equiv$ {l = $l_0$ $\wedge$ a $\in$ Personas(l) $\wedge$ def?(ID, puestos(l)}
**Post** $\equiv$ {(res $=_{obs}$ i $\in$ menu(p) $\wedge_L$ haySuficiente?(obtener(ID, puestos(l)), i, cant))) $\wedge$
res $\Rightarrow_L$ l $=_{obs}$ Vender($l_0, puestoId, a, i, cant$)$\wedge$ $\neg$Res $\Rightarrow_L$ l=$l_0$}
**Complejidad:**$\Theta(\log(A) + \log(L)+\log(P) + \log(Cant))$
**Descripción:** Dado un item, el ID de un puesto y la cantidad que una persona de lolla desea comprar, si la cantidad pedida no supera al stock retorna verdadero.
**Aliasing:** No aplica

HACKEAR(**in/out** $l$ : `lolla`, **in** $a$ : `persona`, **in** $i$ : `item`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {l = $l_0$ $\wedge$ a $\in$ Personas(l)}
**Post** $\equiv$ {(res $=_{obs}$ ConsumioSinPromoPuestos?(a, i, Puestos(l)) $\wedge$
(res $\Rightarrow_L$ l $=_{obs}$ Hackear($l_0$, a, i) $\wedge$ $\neg$ res $\Rightarrow_L$ l $=_{obs}$ $l_0$)}
**Complejidad:** $\Theta(\log(A) + \log(L) + \log(P))$
**Descripción:** Dada una persona y un item, si dicha persona compro dicho item sin decuento, entonces se hackea y retorna true
**Aliasing:** No aplica

GASTOTOTALDE(**in** $l$ : `lolla`, **in** $a$ : `persona`) $\rightarrow$ $res$ : `dinero`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ GastosTotales(l,a)}
**Complejidad:** $\Theta(\log(A))$
**Descripción:** Dado un Lolla y una persona, informa los gastos de esa persona en todo el lolla
**Aliasing:** Devuelve un valor por referencia por lo cual modificarlo actua directamente diccPersona

PERSONAMASGASTO(**in** $l$ : `lolla`) $\rightarrow$ $res$ : `persona`
**Pre** $\equiv$ {$\neg$ $\emptyset$?(personas(l))}
**Post** $\equiv$ {res $=_{obs}$ masGasto(l)}
**Complejidad:** $\Theta()$
**Descripción:** Dado un Lolla, nos devuelve la persona con mas gastos en lolla
**Aliasing:** Devuelve un valor por referencia por lo cual modificarlo actua directamente en ArbolGastos

MENORSTOCK(**in** $l$ : `lolla`, **in** $i$ : `item`) $\rightarrow$ $res$ : `ID`
**Pre** $\equiv$ {True}
**Post** $\equiv$ {res $=_{obs}$ menorStock(l, i)}
**Complejidad:** $\Theta()$
**Descripción:** Dado un Lolla, nos devuelve el ID del puesto con menor stock de un item dado (o si hay mas de uno, el que tenga menor ID)
**Aliasing:** No aplica

## Especificación de las operaciones auxiliares utilizadas la representacion

**TAD** LollaPatuza extendido

**otras operaciones**
sumaHistorialPersona : Conjunto $\times$ Diccionario $\times$ persona $\longrightarrow$ Nat
EstanClavesEnLista? : Conjunto $\times$ Lista $\longrightarrow$ bool

**axiomas**
sumaHistorialPersona(conjClaves, DiccPuestos, a) $\equiv$ **if** $\emptyset$?(conjClaves) **then**
0
**else**
GastoDe(Obtener(DameUno(conjClaves),
DiccPuestos), a)
+ sumaHistorialPersona(Fin(conjClaves),
DiccPuestos, a)
**fi**

EstanClavesEnLista?(conj, lista) ≡ **if** ∅?(conj) **then**
true
**else**
**if** (esta?(DameUno(conj), lista)) **then**
EstanClavesEnLista(SinUno(conj),lista)
**else**
false
**fi**
**fi**

**Fin TAD**

# Representación

## Representación de Lollapatuza

`Lolla` **se representa con** `estr`

donde `lolla` es `tupla`(*diccPuestos*: `diccLog(Id, puesto)`
, *diccIteradoresPuestos*: `diccLog(Id, ItDiccLog)`
, *listaPuestos*: `listaEnlazada(tupla<Id, puesto>)`
, *diccPersonas*: `diccLog(persona,totalPersona)`
, *listaPersonas*: `listaEnlazada(persona)`
, *colaPrioridadGastos* : `colaPrio(<nat, persona>, persona)`
, *diccInfoHackeo* : `diccLog(persona,diccLog(item,diccLog(Id,ItDiccLog)))`
)

Rep : estr ⟶ bool

Rep($e$) $\equiv$ true $\iff$

//Debe haber al menos un puesto en el lolla
$\neg\emptyset?(Claves(e.diccPuestos))\wedge$

//Todos los puestos deben vender sus items al mismo precio
$\neg(\exists p1, p2 : Puesto, i : Item)(def?(p1, e.diccPuestos) \wedge def?(p2, e.diccPuestos) \wedge i \in menu(p1) \wedge i \in menu(p2) \wedge_{\text{L}} precio(i, p1) \neq precio(i, p2)) \wedge$

//Todos los puestos tienen su iterador guardado en DiccIteradoresPuestos
Claves(e.DiccIteradoresPuestos) = Claves(e.DiccPuestos) $\wedge$

//Los Iteradores deben apuntar al Puesto con su mismo ID en DiccPuestos
$(\forall\ ID : nat)(def?(ID, e.DiccIteradoresPuestos) \Rightarrow_{\text{L}} Siguiente(Obtener(ID, e.DiccIteradoresPuestos)) = Obtener(ID, e.DiccPuestos))\wedge$

//Todos los Puestos deberian tener una Tupla en ListaPuestos
(EstanClavesEnLista?(claves(e.diccPuestos), e.ListasPuestos) $\wedge Claves(e.diccPuestos) = long(e.ListasPuestos)\wedge$

//Todas las Tuplas en ListaPuestos deben tener un ID y el puesto de Comida relacionado con dicho ID
$(\forall\, ID : nat)(def?(ID, e.diccPuestos) \Rightarrow_{\text{L}} Esta?(<ID, Significado(e.diccPuestos, ID)>), e.ListaPuestos))\wedge$

//La suma de los gastos de todos los Puestos debe ser igual al gasto total guardado en DiccPersonas
$(\forall\ ID : nat)(def?(ID, e.diccPuestos) \Rightarrow_{\text{L}} (obtener(a, e.DiccPersona) = sumaHistorialPersona(Claves(e.diccPuestos), e.DiccPuestos, a)\wedge$

//Los puestos a hackear deben pertenecer al diccionario de puestos
$\neg(\exists a : persona, I : item, ID : nat)(Def?(a, e.DiccInfoHackeo) \wedge_{\text{L}} Def?(i, Obtener(a, e.DiccInfoHackeo)) \wedge_{\text{L}} Def?(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))) \wedge_{\text{L}} \neg Def?(ID, e.DiccPuestos))\wedge$

//Los puestos a hackear deben coincidir con el puesto de mismo ID de diccPuestos
$\neg(\exists a : persona, I : item, ID : nat)(Def?(a, e.DiccInfoHackeo) \wedge_{\text{L}} Def?(i, Obtener(a, e.DiccInfoHackeo)) \wedge_{\text{L}} Def?(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))) \wedge_{\text{L}} Obtener(ID, e.DiccPuestos) \neq Siguiente(Obtener(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))))) \wedge$

//Los puestos a hackear deben deben ser hackeables, es decir, que tengan productos sin promocion
$\neg(\exists a : persona, I : item, ID : nat)(Def?(a, e.DiccInfoHackeo) \wedge_{\text{L}} Def?(i, Obtener(a, e.DiccInfoHackeo)) \wedge_{\text{L}} Def?(ID, Obtener(i, Obtener(a, e.DiccInfoHackeo))) \wedge_{\text{L}} ventasSinPromo(Obtener(ID, e.DiccPuestos)) = 0) \wedge$

//Cada ID debe de pertenecer a un solo puesto
$\neg(\exists\ ID1, ID2 : nat)((def?(ID1, e.DiccIteradoresPuestos) \wedge def?(ID2, e.DiccIteradoresPuestos) \wedge ID1 \neq ID2) \wedge_{\text{L}} (Obtener(ID1, e.DiccIteradoresPuestos) = Obtener(ID2, e.DiccIteradoresPuestos))) \wedge$

//Cada puesto en la lista de puestos debe de ser unico, no se pueden repetir
NoHayRepetidos(e.ListaPuestos) $\wedge$

//Las claves que aparecen en el diccionario de personas deben ser las mismas que se encuentran en la lista de personas
EstanClavesEnLista?(claves(e.DiccPersonas), e.ListaPersonas) $\wedge \#Claves(e.DiccPersonas) = longitud(e.ListaPersonas)\wedge$

//Todas las personas en el DiccPersonas deben tener su gasto correctamente registrado en ArbolGastos
#Claves(e.ArbolGastos) = #Claves(e.DiccPersonas) $\wedge(\forall\ a : personas)(def?(a, e.DiccPersonas) \wedge_{\text{L}} def?(< Obtener(e.DiccPersonas, a), a >, e.ArbolGastos))\wedge$

//la persona dentro de la tupla Clave de ArbolGastos debe coincidir con la de su significado
$\neg(\exists gasto : nat, a : persona)(def?(< gasto, a >, e.ArbolGastos) \wedge_{\text{L}} Obtener(< gasto, a >, e.ArbolGastos) \neq a) \wedge$

//No puede haber personas repetidas en la lista de lolla
NoHayRepetidos(e.ListaPersonas) ∧

//Las personas hackeadas deben de pertnecer a la lista de personas de Lolla
Claves(e.DiccInfoHackeo) ⊆ $Claves(e.DiccPersonas)$

Abs : estr $e$ ⟶ lollapatuza {Rep($e$)}

Abs($e$) =$_{obs}$ lolla: lollapatuza |
$puestos(lolla) =_{obs} e.DiccPuestos \wedge$
$personas(lolla) =_{obs} claves(e.DiccPersonas)$

# Algoritmos

## Algoritmos del módulo Lollapatuza

---

**iCrearLolla**(**in** $ps$ : `dicc(id, puesto)`, **in** $as$ : `conj(persona)`) → $res$ : lolla

1: // Copio los argumentos para evitar Aliasing y inicializo las estructuras
2: $diccPuesto \leftarrow copiar(ps)$ ▷ $\Theta(P * (I * \sum CantdeDescuentoMaxdecadaItem))$
3: $itPuestos \leftarrow CrearIt(ps)$ ▷ $\Theta(1)$
4: $listaPuestos \leftarrow vacio()$ ▷ $\Theta(1)$
5: // Agrego los puestos a Lista Puestos y guardo los iteradores de puestos
6: $diccIteradoresPuestos \leftarrow vacio()$ ▷ $\Theta(1)$
7: **while** $HaySiguiente(itPuestos)$ **then** ▷ $\Theta(1)$
8: $AgregarAtras(listaPuestos, < SiguienteClave(itPuestos), SiguienteSignificado(itPuestos) >)$ ▷ $\Theta(1)$
9: $Definir(diccIteradoresPuestos, SiguienteClave(itPuestos), itPuestos)$ ▷ $\Theta(P * Log(P))$
10: $Avanzar(itPersonas)$ ▷ $\Theta(1)$
11: // Declaro los gastos iniciales de las personas, tanto en diccVentas como en el arbol de
12: // Gastos. Por ser una tupla, la clave mayor dependera de <a, b> > <c,d> ↔ a > b ∨ (a = b ∧ b > c)
13: $itPersonas \leftarrow CrearIt(as)$ ▷ $\Theta(1)$
14: $diccPersonas \leftarrow vacio()$ ▷ $\Theta(1)$
15: $arbolGasto \leftarrow vacio()$ ▷ $\Theta(1)$
16: **while** $HaySiguiente(itPersonas)$ **then** ▷ $\Theta(A)$
17: $definir(diccPersonas, Siguiente(itPersonas), 0)$; ▷ $\Theta(A)$
18: $definir(arbolGasto, < 0, Siguiente(itPersonas) >, Siguiente(itPersonas))$; ▷ $\Theta(A)$
19: $Avanzar(itPersonas)$ ▷ $\Theta(1)$
20: $listaPersonas \leftarrow copiar(as)$ ▷ $\Theta(A)$
21: $diccInfoHackeo \leftarrow vacio()$ ▷ $\Theta(1)$
22: $res \leftarrow < diccPuesto, DiccIteradoresPuestos, listaPuestos, diccPersonas, listaPersonas, arbolGasto, diccInfoHackeo >$ ▷ $\Theta(1)$

Complejidad: $\Theta(P * (I + \log(P) + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item)) + A * \log(A)$
Justificación: El algoritmo tiene whiles con norma de complejidad $\Theta(1)$ que corren una cantidad de P y A ciclos, en cada ciclo se ejecuta un Definir y operaciones tales de complejidad $\Theta(Log(AoP))$, todo estos ciclos pueden ser acotados por A * Log(A) y P * Log(P). El costo de copiar los Puestos de Comida se simplifica a $\Theta(I)$ mas la suma de todas las cantidades maxima de descuento, ya que conocemos que no se realizaron ninguna venta aun.

---

**iPersonas**(**in** $l$ : `lolla`) → $res$ : listaEnlazada(persona)

1: $res \leftarrow l.listaPersonas$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iPuestos**(**in** $l$ : `lolla`) → $res$ : listaEnlazada(puesto)

1: $res \leftarrow l.listaPuestos$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iVender**(**in/out** $l$ : `lolla`, **in** $puestoId$ : `nat`, **in** $a$ : `persona`, **in** $i$ : `item` **in** $cant$ : `nat`) $\rightarrow$ $res$ : bool

1: $puestoVendedor \leftarrow Significado(l.diccPuestos, puestoId)$ ▷ $\Theta(log(P))$
2: $gastoPersona \leftarrow Significado(l.diccPersonas, a)$ ▷ $\Theta(log(A))$
3: // Elimino la clave anterior del arbol de gasto
4: $Borrar(l.arbolGasto, < gastoPersona, p >)$; ▷ $\Theta(log(P))$
5: $resultado \leftarrow SeRealizoVenta(puestoVendedor, i, a, cant)$ ▷ $\Theta(log(I) + log(A))$
6: // Si todo salio bien y la venta fue hackeable, guardar al puesto de comida como opcion a hackear
7: **if** $(resultado \wedge iVentaSinPromo(puestoVendedor, i, cant))$ **then** ▷ $\Theta(log(I))$
8: // Defino los diccionarios necesarios si aun no lo estaban
9: **if** $(!Definido?(l.diccInfoHackeo, a))$ **then** ▷ $\Theta(log(A))$
10: $Definir(l.diccInfoHackeo, a, vacio())$; ▷ $\Theta(1)$
11: $diccInfoHackeoItem \leftarrow Significado(l.diccInfoHackeo, a)$ ▷ $\Theta(log(I))$
12: **if** $(!Definido?(diccInfoHackeoItem, i))$ **then** ▷ $\Theta(Log(P))$
13: $Definir(diccInfoHackeoItem, i, vacio())$; ▷ $\Theta(1)$
14: $diccPuestosHackeo \leftarrow Significado(diccInfoHackeoItem, i)$ ▷ $\Theta(log(I))$
15: // Si el puesto ya habia sido guardado como una opcion, no hago nada. En caso contrario, agregó su iterador y
16: //uso su ID como Clave
17: **if** $(!Definido?(diccPuestosHackeo, puestoId))$ **then** ▷ $\Theta(log(P))$
18: $PuestoIterador \leftarrow Significado(diccIteradoresPuestos, puestoId)$ ▷ $\Theta(log(P))$
19: $Definir(diccPuestosHackeo, puestoId, PuestoIterador)$ ▷ $\Theta(log(P))$
20: // Si la operacion no tuvo errores, actualizo el gasto de la persona
21: **if** $(resultado)$ **then** ▷ $\Theta(1)$
22: $< CostoVenta, estadoCostoVenta > \leftarrow iCalcularCostoVenta(puestoVendedor, i, cant)$ ▷ $\Theta(Log(I))$
23: **if** $(estadoCostoVenta)$ **then** ▷ $\Theta(1)$
24: $gastoPersona \leftarrow gastoPersona + CostoVenta$ ▷ $\Theta(1)$
25: $res \leftarrow true$ ▷ $\Theta(1)$
26: **else**
27: $res \leftarrow false$ ▷ $\Theta(1)$
28: //Agrego de nuevo el gasto de la persona en el Arbol gastos, si la operacion salio mal en algun momento, sera la //Misma clave que antes de la venta. Si la operacion salio bien, se le sumara el gasto de la venta.
29: $Definir(l.arbolGasto, < gastoPersona, p >, p)$ ▷ $\Theta(Log(P))$

Complejidad: $\Theta(log(A)) + \Theta(log(P)) + \Theta(log(I)) + \Theta(log(1)) = \Theta(log(A) + log(P) + log(I))$
Justificación: Las funciones más costosas del algoritmo tratan de recuperar valores de un diccionarios o definirlos. Estas dos operaciones tienen una complejidad logarítmica respecto a la cantidad de sus claves.

---

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

1: **if** $(!Definido?(l.diccPersonas, a))$ **then** ▷ $\Theta(Log(A))$
2: $res \leftarrow 0$ ▷ $\Theta(1)$
3: $res \leftarrow Significado(l.diccPersonas, a)$ ▷ $\Theta(Log(A))$

Complejidad: $\Theta(log(A))+\Theta(1) = \Theta(log(A))$

**iHackear(in/out** $l$ : `lolla`, **in** $a$ : `persona`, **in** $i$ : `item`) $\rightarrow$ $res$ : bool

```
1: // Primero Chequeo que existe un puesto hackeable con esta combinacion de persona e item
2: if (Definido?(l.diccInfoHackeo, a)) then                                          ▷ Θ(Log(A))
3:     diccInfoHackeoItem ← Significado(diccInfoHackeo, a)                           ▷ Θ(Log(A))
4:     if (Definido?(diccInfoHackeoItem, i)) then                                    ▷ Θ(Log(I))
5:         diccPuestosHackeo ← Significado(diccInfoHackeoItem, i)                    ▷ Θ(Log(I))
6:         // Agarro el primer Iterador del Arbol AVL, al ser en InOrder deberia ser extremo izquierdo, y por eso el
7:         // Minimo.
8:         itIdMenor ← CrearIt(diccPuestosHackeo)                                    ▷ Θ(1)
9:         if (HaySiguiente?(itIdMenor)) then                                        ▷ Θ(1)
10:        // El significado del Diccionario deberia ser un iterador apuntando a un puesto valido
11:        itPuesto ← SiguienteSignificado(itIdMenor)                                ▷ Θ(1)
12:        puestoConMenorId ← SiguienteSignificado(itIdMenor)                        ▷ Θ(1)
13:        // Elimino el gasto desactualizado del puesto de Comida
14:        gastoPersona ← Significado(l.diccPersona, a)                              ▷ Θ(Log(A))
15:        Borrar(l.arbolGasto, < gastoPersona, p >);                                ▷ Θ(Log(P))
16:        gastoPersona ← gastoPersona − iGastosDe(PuestoConMenorId, a)              ▷ Θ(Log(P))
17:        // Corro el hackeo del puesto y verifico que halla salido todo bien
18:        resultadoHackeo ← iHackeo(PuestoConMenorId, a, i)                         ▷ Θ(Log(I) + log(A))
19:        if (resultadoHackeo) then                                                 ▷ Θ(1)
20:            // Si sigue siendo hackeable el puesto no hacemos nada. De caso contrario lo eliminamos del diccionario.
21:            sigueHackeable ← esHackeable?(puestoConMenorId, a, i)                 ▷ Θ(Log(I) + log(A))
22:            if (!sigueHackeable) then                                             ▷ Θ(1)
23:                EliminarSiguiente(ItIdMenor)                                      ▷ Θ(1)
24:            //Agrego el gasto actualizado del puesto de Comida
25:            gastoPersona ← gastoPersona + iGastosDe(puestoConMenorId, a)          ▷ Θ(Log(P))
26:            Definir(l.arbolGasto, < gastoPersona, p >), p                         ▷ Θ(Log(P))
27:            else
28:                res ← false                                                       ▷ Θ(1)
29:        else
30:            res ← false                                                           ▷ Θ(1)
31:    else
32:        res ← false                                                               ▷ Θ(1)
33: else
34:    res ← false                                                                   ▷ Θ(1)
```

Complejidad: $\Theta(log(A))+\Theta(log(P))+\Theta(log(I))+\Theta(1) = \Theta(log(A)+log(P)+log(I))$

Justificación: En el caso de que el puesto siga siendo Hackeable luego de ser Hackeado, la Complejidad termina siendo: $\Theta(log(A)+log(I))$, ya que tenemos guardado de antemano el ID del Puesto para Hackear y podemos accederlo en $\Theta(1)$. En el caso de que el puesto deje de ser Hackeable, necesitamos borrar el puesto de la lista de puesto hackeables, aunque eliminarlo cuesta $\Theta(1)$ por el uso de iteradores, realisticamente tarda $\Theta(log(P))$, ya que el AVL en que esta implementado el diccionario necesita rebalancear. Por eso la complejidad del peor caso es: $\Theta(log(A)+log(P)+log(I))$

**iMenorStock**(**in** $l$ : `lolla`, **in** $i$ : `iten`) → $res$ : nat

```
1: // Itero por los puestos hasta encontrar el primero que tenga el item en su stock.
2: itDiccionario ← CrearIt(l.diccPuestos)                                   ▷ Θ(1)
3: puestoMinId ← SiguienteClave(itDiccionario)                              ▷ Θ(1)
4: MinStock ← 0                                                             ▷ Θ(1)
5: while (HaySiguiente(itDiccionario))                                      ▷ Θ(1)
6:     puestoActual ← SiguienteSignificado(itDiccionario)                   ▷ Θ(1)
7:     < stock, resultado >← iStock(puestoActual, i)                        ▷ Θ(Log(I))
8:     if (resultado) then                                                  ▷ Θ(1)
9:         puestoMinId ← SiguienteClave(itDiccionario)                      ▷ Θ(1)
10:        // Copio para evitar aliasing.
11:        MinStock ← Copiar(stock)                                         ▷ Θ(1)
12:        break
13:    Avanzar(itDiccionario)
14: // Si no se encontro el item en ningun puesto, se devuelve el de menor ID. El que se encuentra en el primer iterador
    //(Propiedad de AVL en InOrder)
15: if (!HaySiguiente(itDiccionario)) then                                  ▷ Θ(1)
16:     res ← puestoMinId                                                   ▷ Θ(1)
17: else
18:     Avanzar(itDiccionario)                                              ▷ Θ(1)
19:     // Itero por el resto de Puestos, cada vez que encuentro uno con menor Stock, remplazo PuestoMinId, PuestoMin,
    //MinStock
20:     while (HaySiguiente(itDiccionario))                                 ▷ Θ(1)
21:         puestoActual ← SiguienteSignificado(itDiccionario)              ▷ Θ(1)
22:         < stockActual, resultado >← iStock(puestoActual, i)             ▷ Θ(Log(I))
23:         if (resultado) then                                             ▷ Θ(1)
24:             if (stockActual < MinStock) then                            ▷ Θ(1)
25:                 puestoMinId ← SiguienteClave(itDiccionario)             ▷ Θ(1)
26:                 puestoMin ← puestoActual                                ▷ Θ(1)
27:     res ← PuestoMinId                                                   ▷ Θ(1)
```

Complejidad: $\Theta(P * Log(I))$
Justificación: En todos los casos debo recorrer todos los puestos para encontrar el menor Stock, por eso la complejidad termina siendo $\Theta(P * Log(I))$, ya que tardo $\Theta(Log(I))$ en obtener el Stock del puesto.

---

**iPersonaMasGasto**(**in** $l$ : `lolla`) → $res$: *persona*

```
1: //CrearItUlt devuelve el iterador tal que Anterior es igual al último elemento del orden.
2: //Ya que este es InOrder de un AVL, sabemos que el último se trata de la clave más grande.
3: //En este caso la tupla<Gasto, Persona ID>, como hemos declarado anteriormente
4: //<a, b> > <c,d> ↔ a > b ∨ (a = b ∧ b > c)
5: res ← AnteriorSignificado(CrearItUlt(l.arbolGasto))                      ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

# 2. Módulo Puesto de Comida

# Interfaz

**se explescuentoica con**: TAD Puesto de Comida

**géneros**: `puesto`

**Servicios Usados**:
**DiccLog** :
**Vacío**() → res : dicc(q, b)
Complejidad: (1)
Descripción: genera un diccionario vacío.

**Definir**(in/out d : dicc(q, b), in k : q, in s : b) → res : itDicc(q, b)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)) + copy(k) + copy(s))$, donde K = claves(d)
Descripción: Define la clave k con el signicado s en el diccionario. Retorna un iterador al elemento recién agregado.
Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

**Definido?**(in d : dicc(q, b), in k : q) → res : bool
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: devuelve true si y sólo k está definido en el diccionario.

**Significado**(in d : dicc(q, b), in k : q) → res : b
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

**Borrar**(in/out d : dicc(q, b), in k : q)
Complejidad: $\theta(Log(\sum_{k' \in K} equal(k,k)))$, donde K = claves(d)
Descripción: elimina la clave k y su significado de d.

**Copiar**(in d : dicc(q, b)) → res : dicc(q, b)
Complejidad: $\theta(\sum_{k' \in K}(copy(k) + copy(significado(k,d))))$, donde K = claves(d)
Descripción: genera una copia nueva del diccionario.

**Lista Enladaza** :
**Vacía**() → res : lista($\alpha$)
$Complejidad : \theta(1)$
Descripción: genera una lista vacıa

**agregarAtras**(in/out l : lista($\alpha$), $in\ a : \alpha$) → $res : itLista(\alpha)$
$Complejidad : \theta(copy(a))$
Descripción: agrega el elemento a como ultimo elemento de la lista. Retorna un iterador a l, de forma tal que Siguiente devuelva a.

**Primero**(in l : lista($\alpha$)) → $res : \alpha$
$Complejidad : \theta(1)$
Descripción: devuelve el primer elemento de la lista.
Aliasing: res es modificable si y solo si l es modificable.

**Fin**(in/out l : lista($\alpha$))
$Complejidad : \theta(1)$
Descripción: Elimina el primero elemento de l

**Longitud**(in l : lista($\alpha$)) → $res : nat$
$Complejidad : \theta(copy(a))$
Descripción: Devuelve la cantidad de elementos que tiene la lista

**Copiar**(in l : lista($\alpha$)) → $res : lista(\alpha)$
$Complejidad : \theta(\sum_{i=1}^{t} copy(l[i]))$, donde t = long(l).
Descripción: Genera una copia nueva de la lista

**Vector** :
**Vacía**() → res : vector($\alpha$)
$Complejidad : \theta(1)$

Descripción: Genera un vector vacıo

**Longitud**(in v : vector($\alpha$)) $\rightarrow$ $res$ : $nat$
$Complejidad$ : $\theta(1)$
Descripción: Devuelve la cantidad de elementos que contiene el vector

**Copiar**(in v : vector($\alpha$)) $\rightarrow$ $res$ : $vector(\alpha)$
$Complejidad$ : $\theta(\sum_{i=1}^{l} copy(v[i]))$, donde l = long(v).
Descripción:Genera una copia nueva del vector

**AgregarAtras**(in/out v : vector($\alpha$), $in\ i : nat$)
$Complejidad$ : $\theta(f(long(v)) + long(v) - i + copy(a)))$.
Descripción: agrega el elemento a como ultimo elemento del vector.
Aliasing: El elemento a se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando long)v= es potencia de 2.

**•[•]**(in v : vector($\alpha$), $ini : nat$) $\rightarrow$ $res : \alpha$
$Complejidad$ : $\theta(1)$.
Descripción: devuelve el elemento que se encuentra en la i- esima posicion del vector en base 0. Es decir, v[i] devuelve el elemento que se encuentra en la posici on i + 1.
Aliasing: res es modificable si y s olo si v es modificable.

## Operaciones básicas de Puesto de Comida

STOCK(**in** $p$: `puesto`, **in** $i$: `item`, **out** $s$: `stock`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {($res =_{\text{obs}}$ i $\in$ menu(p)) $\wedge$ (res $\Rightarrow_{\text{L}}$ s $=_{\text{obs}}$ Stock(p,i))}
**Complejidad:** $\Theta(\log(I))$
**Descripción:** Devuelve el stock de un ítem
**Aliasing:** Devuelve una referencia del significado del diccionario DiccStock

DESCUENTO(**in** $p$: `puesto`, **in** $i$: `item`, **in** $cant$: `nat`, **out** $desc$: `decuento`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {cant $\geq$ 0}
**Post** $\equiv$ {($res =_{\text{obs}}$ i $\in$ menu(p)) $\wedge$ ((res $\Rightarrow_{\text{L}}$ desc $=_{\text{obs}}$ Descuento(p,i,cant)) $\wedge$ ($\neg$ res $\Rightarrow_{\text{L}}$ desc $=_{\text{obs}}$ 0)}
**Complejidad:** $\Theta(\log(L))$
**Descripción:** Dado un puesto, item y cantidad, retorna la cantidad de descuento a aplicarle
**Aliasing:** descuento es modificable si y solo si ListaDescuento es modificable

NUEVOPUESTO(**in** $dPrecio$: `dicc(item, precio)`, **in** $dDesc$: `dicc(item, dicc(cantidad, nat))`, **in** $dStock$: `dicc(item, cantidad)`) $\rightarrow$ $res$ : `puesto`
**Pre** $\equiv$ {claves($dPrecio$) $=_{\text{obs}}$ claves($dStock$) $\wedge$ claves($dDesc$) $\subseteq$ claves($dPrecio$))}
**Post** $\equiv$ {$res =_{\text{obs}}$ crearPuesto($dPrecio, dStock, dDesc$)}
**Complejidad:** $\Theta(\text{I} + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$
**Descripción:** Crea un puesto de comida dado un menu, una lista de descuentos y un stock
**Aliasing:** No hay aliasing ya que se crea por copia cada una de las estructuras

GASTOSDE(**in** $p$: `puesto`, **in** $a$: `persona`) $\rightarrow$ $res$ : `dinero`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{\text{obs}}$ gastosDe(p,a)}
**Complejidad:** $\Theta(\log(P))$
**Descripción:** Dado un puesto y un cliente, informa el gasto de dicho cliente en ese puesto
**Aliasing:** res es modificable si y solo si diccVentas es modificable

HACKEO(**in/out** $p$: `puesto`, **in** $i$: `item`, **in** $a$: `persona`)
**Pre** $\equiv$ {p $=_{\text{obs}}$ $p_0$ $\wedge$ i $\in$ menu(p) $\wedge_{\text{L}}$ consumioSinPromo?(p,i,a)}
**Post** $\equiv$ {p $=_{\text{obs}}$ OlvidarItem($p_0$, i, a)}
**Complejidad:** $\Theta(\log(L) + \log(A))$
**Descripción:** Dado un ítem comprado por una persona en un puesto lo olvida del historial de ventas (lo borra de la lista y lo resta al gasto total) y actualiza el stock de dicho puesto

**Aliasing:** No aplica

CALCULARCOSTOVENTA(**in** $p$: `puesto`, **in** $i$: `item`, **in** $cant$: `nat`, **out** $costo$: `nat`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {(res $=_{obs}$ i $\in$ menu(p)) $\wedge$ (res $\Rightarrow_L$ costo $=_{obs}$ gastosDe1Venta(p, <i,cant>))}
**Complejidad:** $\Theta(log(I))$
**Descripción:** Dado una compra de un item del menú y una cantidad, calcula el precio de los items comprados aplicando el descuento correspondiente (si es que existe)
**Aliasing:** No aplica

SEREALIZOVENTA(**in/out** $p$: `puesto`, **in** $i$: `item`, **in** $cant$: `nat`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ { p$=_{obs}$ p$_0$}
**Post** $\equiv$ {(res $=_{obs}$ i $\in$ menu(p) $\wedge$ cant $\leq$ stock(p)) $\wedge$ ((res $\Rightarrow_L$ p=ventas(p$_0$,a) $\wedge$ p = stock(p$_0$,i)) $\wedge$ ($\neg$ res $\Rightarrow_L$ p = p$_0$))}
**Complejidad:** $\Theta(\log(L) + \log(A)$
**Descripción:** Dado una compra realizada, actualiza el historial de las ventas (lo agrega tanto a la lista, como lo suma al gasto total) y actualiza el stock de dicho puesto
**Aliasing:** No aplica

VENTASINPROMO?(**in** $p$: `puesto`, **in** $i$: `item`, **in** $cant$: `nat`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {i $\in$ menu(p) $\wedge$ cant $> 0$ }
**Post** $\equiv$ {res $=_{obs}$ ( Descuento(p,i,cant) = 0)}
**Complejidad:** $\Theta(\log(I)$ )
**Descripción:** Dada una venta en puesto de comida, verifica si hubo o no descuento. Devuelve Falso si existe un descuento del producto en dicho puesto, de lo contrario devuelve verdadero (es decir, si el descuento es igual a 0)
**Aliasing:** No aplica

ESHACKEABLE?(**in** $p$: `puesto`, **in** $i$: `item`, **in** $a$: `persona`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {(res $=_{obs}$ (i $\in$ $menu(p)$)) $\wedge$ ($res \Rightarrow_L consumioSinPromo?(p, a, i)$)}
**Complejidad:**$\Theta(\log(A) + \log(I)$ )
**Descripción:** Dado un item y una persona, devuelve true si existe una venta de dicho item hecha por aquella persona que puede ser hackeable
**Aliasing:** No aplica

COPIAR(**in** $p$: `puesto`) $\rightarrow$ $res$ : `puesto`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{obs}$ p}
**Complejidad:** $\Theta(I + CantidadtotaldeDescuentos + A * I * CantidadVentasActuales))$
**Descripción:** Dado un puesto, devuelve una copia del mismo
**Aliasing:** La version copiada no tiene aliasing con ninguna de las estructuras originales

(

## Especificación de las operaciones auxiliares utilizadas la representacion

**TAD** Puesto De Comida Extendido

**otras operaciones**

Suma : Lista $\times$ Puesto $\longrightarrow$ Nat
GastoVenta : puesto $\times$ item $i$ $\times$ cant $c$ $\longrightarrow$ Nat {i $\in$ menu(e) $\wedge$ c $>$0}
Precio : item $i$ $\times$ Puesto $\longrightarrow$ nat {i $\in$ menu(e)}
EncontrarDescuento : Puesto $\times$ item $i$ $\times$ cant $c$ $\longrightarrow$ nat {i $\in$ menu(e) $\wedge$ c $>$0}
ItemsDentro? : Lista $\times$ Conjunto $\longrightarrow$ Bool
EsHackeoSubDeVenta : Secuencia $\times$ lista $\longrightarrow$ Bool
VentaDePersonaItem? : Secuencia $\times$ item $\times$ persona $\times$ Puesto $\longrightarrow$ Bool {i $\in$ menu(e)}
SonHackeables? : Secuencia $\times$ Puesto $\longrightarrow$ Bool

**axiomas**

Suma(hist, e) $\equiv$ **if** vacia?(hist) **then**
0
**else**
GastoVenta($\pi_1$(prim(hist)), $\pi_2$(prim(hist)), e) + Suma(Fin(hist), e)
**fi**

Precio(i,e) ≡ Obtener(i, e.diccPrecio)
GastoVenta(e, i, c) ≡ AplicarDescuento(c x Precio(i, e), EncontrarDescuento(e, i, c))
EncontrarDescuento(e, i, c) ≡ **if** Obtener(i, e.diccDescuentos).max ≥ c **then**
Obtener(i, e.diccDescuentos).listaDescuentos[c]
**else**
Obtener(i,e.diccDescuentos).listaDescuentos[Obtener(i,e.diccDescuentos).max]
**fi**
ItemsDentro?(hist, items) ≡ **if** vacia?(hist) **then**
true
**else**
**if** $\pi_1$(Prim(hist)) ∈ items **then** ItemsDentro?(fin(hist), items) **else** false **fi**
**fi**
EsHackeoSubDeVenta(vHackeables, ventas) ≡ **if** vacia?(vHackeables) **then**
true
**else**
**if** esta?(Siguiente(Prim(vHackebles)), ventas) **then**
EsHackeoSubDeVenta(Fin(vHackeables), ventas)
**else**
false
**fi**
**fi**
VentaDePersonaItem?(vHackeables, i, p, e) ≡ **if** Vacia?(vHackeables) **then**
true
**else**
**if** $\pi_1$(Siguiente(Prim(vHackebles))) = i **then**
**if** Esta?(Siguiente(Prim(vHackebles)),
Obtener(p, e.diccVentas).historial) **then**
VentaDePersonaItem?(Fin(vHackeables), i, p, e)
**else**
false
**fi**
**else**
false
**fi** false
**fi**
itDiccionario)) (1) 21: puestoActual ← SiguienteSigni
SonHackeables?(vHackeables, e) ≡ **if** Vacia?(vHackeables) **then**
true
**else**
**if** (EncontrarDescuento(e, $\pi_1$(Siguiente(Prim(vHackebles))),
$\pi_2$(Siguiente(Prim(vHackebles)))) = 0) **then**
SonHackeables?(Fin(vHackeables), e)
**else**
false
**fi**
**fi**

**Fin TAD**

# Representación

## Representación de puesto de comida

`Puesto` **se representa con** `estr`

donde `estr` es `tupla`(*diccPrecio*: `diccLog(item, precio)`
, *diccStock*: `diccLog(item, cantidad)`
, *diccDescuentos*: `diccLog(Item, InfoDescuento)`
, *diccVentas*: `diccLog(persona,registroVentasPersona)`
, *diccHackeos*: `diccLog(persona, diccLog(item, sec(itLista)))`
)

donde `InfoDescuento` es `tupla`(*max*: `nat`
, *listaDescuento*: `vector(max+1)`
)

donde `registroVentasPersona` es `tupla`(*total*: `nat`
, *historial*: `lista(tupla(item,cant))`
)

Rep : estr $\longrightarrow$ bool
Rep($e$) $\equiv$ true $\Longleftrightarrow$

//Deben estar los mismos items tanto en el stock como en el menu de precios)
$Claves(e.diccPrecio) = Claves(e.diccStock) \wedge$

//Los items que tienen descuentos deben estar dentro del menu
$Claves(e.diccDescuentos) \subseteq Claves(e.diccPrecio) \wedge$

//Si un item tiene descuento, entonces la lista de descuento debe existir y tener un tamaño igual a la maxima cantidad de Descuento + 1. Además en dicha lista la cantidad 0 no tiene descuento
$(\forall\ i : item)(def?(i, e.diccDescuentos) \Rightarrow_{\text{L}} (long(obtener(i, e.diccDescuentos).listaDescuento) = obtener(e.diccDescuentos, i).max + 1 \wedge$
$obtener(i, e.diccDescuentos).listaDescuento[0] = 0)) \wedge_{\text{L}}$

//Todos los valores que corresponden al descuento de un item deben encontrarse entre 1 y 100
$(\forall\ i : item)(def?(i, e.diccDescuentos) \Rightarrow_{\text{L}} ((\forall\ cant : nat)(0 \leq cant < obtener(i, e.diccDescuentos).max + 1 \Rightarrow_{\text{L}} 0 \leq obtener(i, e.diccDescuentos).listaDescuento[cant] < 100)) \wedge$

//la sumatoria de las compras de una persona debe ser igual al total dentro de DiccVentas
$(\forall\ a : persona)(def?(a, e.diccVentas) \Rightarrow_{\text{L}} \pi_1(obtener(a, e.diccVentas)) = Suma(\pi_2(obtener(p, e.diccVentas), e)) \wedge$

//Todos los items comprados por una persona deben corresponder a los productos dentro del menu
$(\forall\ a : persona)(def?(a, e.diccVentas) \Rightarrow_{\text{L}} ItemsDentro?(obtener(a, e.diccVentas).historial, claves(e.diccStock)$
$\wedge$

// Si una persona puede ser hackeada, entonces tambien debe tener compras registradas
$(\forall\ a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} Def?(a, e.diccVentas)) \wedge$

//todas las ventas Hackeables de una persona deben pertenecer a sus ventas totales
$(\forall\ a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} ((\forall\ i : item)((def?(i, obtener(a, e.diccHackeos)))) \Rightarrow_{\text{L}} EsHackeoSubDeVenta?(obtener(i, obtener(a, e.diccHackeos)), obtener(a, e.diccVentas).historial))) \wedge$

//Todas las compras hackeables de una persona e item determinado deben pertenecer a dicha persona e item
$(\forall\ a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} ((\forall\ i : item)((def?(i, obtener(a, e.diccHackeos)))) \Rightarrow_{\text{L}} VentaDePersonaItem?(obtener(i, obtener(a, e.diccHackeos)), a, i, e)) \wedge$

//Cada item hackeable deben ser verdaderamente hackeable (No tiene promocion)
$(\forall\ a : persona)(def?(a, e.diccHackeos) \Rightarrow_{\text{L}} ((\forall\ i : item)(def?(i, obtener(a, e.diccHackeos))) \Rightarrow_{\text{L}} SonHackeables?(obtener(i, obtener(p, e.diccHackeos)), e))$

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

Abs($e$) $=_{\text{obs}}$ p: Puesto |

$menu(p) =_{\text{obs}} Claves(e.diccPrecio) \wedge$
$(\forall\ i : item)(i \in Claves(e.diccPrecio) \Rightarrow_{\text{L}}$
$(Precio(p,i) =_{\text{obs}} obtener(i, e.diccPrecio) \wedge$
$Stock(p,i) =_{\text{obs}} obtener(i, e.diccStock) \wedge$
$(\forall\ cant : nat)$
$(descuento(p,i,c) =_{\text{obs}} EncontrarDescuento(e,i,c))) \wedge$
$(\forall\ a : persona)(a \in Claves(e.diccVentas) \Rightarrow_{\text{L}}$
$ventas(p,a) =_{\text{obs}} listaAMulticonj(Obtener(p, e.diccVentas).historial))$

# Algoritmos

## Algoritmos del módulo Puesto De Comida

---

**iStock**(**in** $p$: `puesto`, **in** $i$: `item`, **out** $s$: `stock`) $\rightarrow$ $res$ : nat

1: **if** ($Definido?(p.diccStock, i)$) **then** ▷ $\Theta(Log(I))$
2: $stock \leftarrow Significado(p.diccStock, i)$ ▷ $\Theta(Log(I))$
3: $res \leftarrow true$ ▷ $\Theta(1)$
4: **else**
5: $stock \leftarrow 0$ ▷ $\Theta(1)$
6: $res \leftarrow false$ ▷ $\Theta(1)$

Complejidad: $\Theta(log(I))+\Theta(1) = \Theta(log(I))$

---

**iDescuento**(**in** $p$: `puesto`, **in** $i$: `item`, **in** $cant$: `nat`, **out** $desc$: `decuento`) $\rightarrow$ $res$ : bool

1: //Verifico que el item forme parte de los items vendidos
2: **if** (!$Definido?(p.diccPrecios, i)$) **then** ▷ $\Theta(Log(I))$
3: $desc \leftarrow 0$ ▷ $\Theta(1)$
4: $res \leftarrow false$ ▷ $\Theta(1)$
5: **else**
6: //Verifico que el item tenga algun descuento registrado
7: **if** ($Definido?(p.diccDescuentos, i)$) **then** ▷ $\Theta(Log(I))$
8: //descuentoInfo.listaDescuento tiene guardado en
9: //cada index el porcentaje de descuento que esa cantidad de item deberia tener.
10: //Si es mayor que el mayor descuento, agarramos el max.
11: $descuentoInfo \leftarrow Significado(p.diccDescuentos, i)$ ▷ $\Theta(Log(I))$
12: **if** ($cant > descuentoInfo.max$)**then** ▷ $\Theta(1)$
13: $descuento \leftarrow descuentoInfo.listaDescuento[descuentoInfo.max - 1]$ ▷ $\Theta(1)$
14: $res \leftarrow true$ ▷ $\Theta(1)$
15: **else**
16: $descuento \leftarrow descuentoInfo.listaDescuento[cant]$ ▷ $\Theta(1)$
17: $res \leftarrow true$ ▷ $\Theta(1)$
18: **else**
19: $descuento \leftarrow 0$ ▷ $\Theta(1)$
20: $res \leftarrow true$ ▷ $\Theta(1)$

Complejidad: $\Theta(log(I)) + \Theta(1) = \Theta(log(I))$
Justificación: Gracias a tener un vector que al indexear por cantidad da directamente el porcentaje, devolver el Descuentos es $\Theta(1)$, solo queda la complejidad de obtener tal vecto, al estar guardado en un DiccLog, el costo de obtenerlo es $\Theta(Log(I))$

---

**NuevoPuesto(in** $dPrecio$: `dicc(item, precio)`, **in** $dDesc$: `dicc(item, dicc(cantidad, nat))`, **in** $dStock$: `dicc(item, cantidad)`) $\rightarrow$ $res$ : puesto

```
1: //Copiamos los argumentos para evitar aliasing
2: copiaDiccionarioPrecio ← Copiar(dPrecio)                                   ▷ Θ(I)
3: copiaDiccionarioStock ← Copiar(dStock)                                     ▷ Θ(I)
4: //Construimos los vectores de Descuentos por cada item
5: diccionarioDescuentos ← Vacio()                                            ▷ Θ(1)
6: itDiccionario ← CrearIt(diccionarioDescuentos)                             ▷ Θ(1)
7: //Agregamos los descuentos
8: while (HaySiguiente(itDiccionario)) then                                   ▷ Θ(1)
9:     vectorDescuentos ← Vacio()                                             ▷ Θ(1)
10:    diccCantDescuentos ← SiguienteSignificado(itDiccionario)               ▷ Θ(1)
11:    itDescuento ← CrearIt(diccCantDescuentos)                              ▷ Θ(1)
12:    indexActual ← 0                                                        ▷ Θ(1)
13:    DescuentoActual ← 0                                                    ▷ Θ(1)
14:    //Empezamos con un descuento de 0 y vamos avanzando de a 1 hasta llegar al primer Descuento.
15:    //Agregando 0 al vector en cada paso.
16:    //Cada vez que llegamos a un nuevo descuento, guardamos su porcentaje y seguimos avanzando a la
17:    //cantidad siguiente, agregando el porcentaje al vector en cada paso
18:    while (HaySiguiente(itDescuento)) then                                 ▷ Θ(1)
19:        while (indexActual < SiguienteClave(itDescuento)) then             ▷ Θ(1)
20:            AgregarAtras(vectorDescuentos, DescuentoActual)                ▷ Θ(I)
21:            index++                                                        ▷ Θ(1)
22:        DescuentoActual ← SiguienteSignificado(itDescuento)                ▷ Θ(1)
23:        Avanzar(itDescuento)                                               ▷ Θ(1)
24:    Agregar(vectorDescuentos, DescuentoActual)                             ▷ Θ(I)
25:    max ← AnteriorClave(itDescuento)                                       ▷ Θ(1)
26:    infoDescuento ← < max, vectorDescuento >                               ▷ Θ(1)
27:    Definir(diccionarioDescuentos, Siguiente(itDiccionario), infoDescuento)  ▷ Θ(Log(I))
28:    Avanzar(itDiccionario)                                                 ▷ Θ(1)
29: diccVentas ← Vacio()                                                      ▷ Θ(1)
30: diccHackeos ← Vacio()                                                     ▷ Θ(1)
31: res ← < copiaDiccionarioPrecio, copiaDiccionarioStock, diccionarioDescuentos, diccVentas, diccHackeos >  Θ(1)
```

Complejidad: $\Theta(I + \sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$

Justificación: El costo de copiar un diccionario donde tanto las claves como significados son números enteros es igual a la cantidad de claves (o sea $\Theta(I)$). Para crear el vector que devuelve el porcentaje correspondiente, indexeando por la cantidad, tendremos que iterar desde 0 hasta la cantidad maxima de descuento( y a su vez por cada Item) tardamos una complejidad de $\Theta(\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$

---

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

```
1: if (Definido?(p.diccVentas, a)) then                                       ▷ Θ(Log(P))
2:     res ← Significado(p.diccVentas, a).total                               ▷ Θ(Log(P))
3: else
4:     res ← 0                                                                ▷ Θ(1)
```

Complejidad: $\Theta(log(P)) + \Theta(1) = \Theta(log(P))$

**iCalcularCostoVenta(in/out** $l$ : `lst`, **in** $a$ : $\alpha$) $\rightarrow$ $res$ : *bool*

```
1:  //Verifico que la venta este registrada en la lista de precios
2:  if Definido?(p.diccPrecio, i) then                                   ▷ Θ(Log(I))
3:      precioUnit ← Significado(p.diccPrecio, i)                        ▷ Θ(Log(I))
4:      < descProd, descValido >← iDescuento(p, i, cant)                  ▷ Θ(Log(I))
5:      //Si no hubo un error al agarrar los descuentos, calculo el precio final.
6:      if !descValido then                                              ▷ Θ(1)
7:          costo ← 0                                                    ▷ Θ(1)
8:          res ← false                                                  ▷ Θ(1)
9:      else
10:         costo ← cant * div(precioUnit * (100 − descProd), 100)       ▷ Θ(1)
11:         res ← true                                                   ▷ Θ(1)
12: else
13:     costo ← 0                                                        ▷ Θ(1)
14:     res ← false                                                      ▷ Θ(1)
```

Complejidad: $\Theta(log(I)) + \Theta(1) = \Theta(log(I))$
Justificación: Verificar que un elemento existe en un DiccLog, como el obtenerlo cuestan $\Theta(log(I))$, gracias a que obtener el porcentaje de descuento es $\Theta(log(I))$, la complejidad queda $\Theta(log(I))$

---

**iVentaSinPromo?(in** $p$ : `puesto`, **in** $i$ : `item`, **in** $cant$ : `nat`) $\rightarrow$ $res$ : *bool*

```
1:  < descProd, descValido >← iDescuento(p, i, cant)                     ▷ Θ(Log(I))
2:  if (descProd == 0) then                                              ▷ Θ(1)
3:      res ← true                                                       ▷ Θ(1)
4:  else
5:      res ← false                                                      ▷ Θ(1)
```

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

**iSeRealizoVenta**(**in/out** $p$ : `puesto`, **in** $i$ : `item`, **in** $cant$ : `nat`) $\rightarrow$ $res$ : $bool$

```
1: //Intento guardar el stock actual del item. Si no puedo, devuelvo error
2: < stockDisponible, resultado > ← iStock(p, i)                                  ▷ Θ(Log(I))
3: if (!resultado) then                                                            ▷ Θ(1)
4:     res ← false                                                                 ▷ Θ(1)
5: else
6:     //Verifico que hay stock suficiente para la venta
7:     if (stockDisponible < c) then                                               ▷ Θ(1)
8:         res ← false                                                             ▷ Θ(1)
9:     else
10:        //Descuento del stock la cantidad de compra
11:        stockDisponible ← stockDisponible − c                                   ▷ Θ(1)
12:        //Intento calcular el costo de la venta para actualizar gastos
13:        < precioVenta, resultadoOperacion > ← CalcularCostoVenta(p, i, c).costo ▷ Θ(Log(I))
14:        if (resultadoOperacion) then                                            ▷ Θ(1)
15:            if (!Definido?(p.diccVentas, a)) then                               ▷ Θ(Log(A))
16:                Definir(p.diccVentas, a, < 0, Vacia() >)                        ▷ Θ(Log(A))
17:            //Actualizo el gasto total de la persona y su historial
18:            gastosHastaLaFecha ← Significado(p.diccVentas, a)                   ▷ Θ(Log(P))
19:            gastosHastaLaFecha.total ← gastosHastaLaFecha.total + precioVenta   ▷ Θ(1)
20:            iteradorVenta ← AgregarAtras(gastosHastaLaFecha.historial, < i, c >) ▷ Θ(1)
21:            //Si la venta es hackeable, guardamos el iterador en la lista de ventas hackeables de esa persona
22:            if (iVentaSinPromo?(p, i, c)) then                                  ▷ Θ(Log(I))
23:                if (!Definido?(p.diccHackeos, a)) then                          ▷ Θ(Log(A))
24:                    Definir(p.diccHackeos, a, Vacio())                          ▷ Θ(Log(A))
25:                diccHackeosItems ← Significado(p.diccHackeos, a)                ▷ Θ(Log(A))
26:                if (!Definido?(diccHackeosItems, i)) then                       ▷ Θ(Log(I))
27:                    Definir(diccHackeosItems, i, Vacia())                       ▷ Θ(Log(I))
28:                listaVentasHackeables ← Significado(diccHackeosItems, i)        ▷ Θ(Log(I))
29:                AgregarAtras(listaVentasHackeables, iteradorVenta)              ▷ Θ(1)
30:            res ← true                                                          ▷ Θ(1)
31:        else
32:            res ← false                                                         ▷ Θ(1)
```

Complejidad: $\Theta(log(I))+\Theta(Log(A))$ + $\Theta(1)$ = $\Theta(Log(I) + Log(A)$

Justificación: Dado que solo se utilizaron funciones de DiccLog, verificacion de claves y obtencion de significados, la complejidad es igual al logaritmo de sus claves. El resto de las funciones utilizadas se tratan de agregar nuevos elementos a listas enlazadas, que tiene una complejidad de $\theta(1)$. Por lo cual termina quedando la complejidad $\theta(Log(I) + log(A))$

---

**iEsHackeable**(**in** $p$ : `puesto`, **in** $i$ : `item`, **in** $a$ : `persona`) $\rightarrow$ $res$ : $bool$

```
1: diccHackeosItems ← Significado(p.diccHackeos, a)                                ▷ Θ(Log(A))
2: listaVentasHackeables ← Significado(diccHackeosItems, i)                        ▷ Θ(Log(I))
3: if (longitud(listaVentasHackeables)! = 0) then                                  ▷ Θ(1)
4:     res ← true                                                                  ▷ Θ(1)
5: else
6:     res ← false                                                                 ▷ Θ(1)
```

Complejidad: $\Theta(log(I))+\Theta(log(A))$ + $\Theta(1)$ = $\Theta(log(I) + log(A))$

**iHackeo(in/out** $p$: `puesto`, **in** $i$: `item`, **in** $a$: `persona`**)** $\rightarrow$ $res$ : $bool$

1: $diccHackeosItems \leftarrow Significado(p.diccHackeos, a)$ ▷ $\Theta(Log(A))$
2: $listaVentasHackeables \leftarrow Significado(diccHackeosItems, i)$ ▷ $\Theta(I)$
3: // guardo el iterador y la tupla venta.
4: $itVentaHackeable \leftarrow Primero(listaVentasHackeables)$ ▷ $\Theta(1)$
5: $ventaHackeable \leftarrow Siguiente(itVentaHackeable)$ ▷ $\Theta(1)$
6: $itemHackeado \leftarrow \pi_1(ventaHackeable)$ ▷ $\Theta(1)$
7: $cantHackeado \leftarrow \pi_2(ventaHackeable)$ ▷ $\Theta(1)$
8: // Actualizo el Stock, agregandole la cantidad Hackeada.
9: $< stockRestante, resultado > \leftarrow iStock(p, itemHackeado)$ ▷ $\Theta(Log(I))$
10: **if** $(resultado)$ **then** ▷ $\Theta(1)$
11: $stockRestante \leftarrow stockRestante + cantHackeado$ ▷ $\Theta(1)$
12: // Actualizar el costo de la persona, restandole el gasto de la venta Hackeada
13: $precioVenta \leftarrow CalcularCostoVenta(p, itemHackeado, cantHackeado).costo$ ▷ $\Theta(Log(I))$
14: $gastosHastaLaFecha \leftarrow Significado(p.diccVentas, a)$ ▷ $\Theta(Log(A))$
15: $gastosHastaLaFecha.total \leftarrow gastosHastaLaFecha.total - precioVenta$ ▷ $\Theta(1)$
16: //Elimino la venta del historial y su iterador.
17: $EliminarSiguiente(itVentaHackeable)$ ▷ $\Theta(1)$
18: $Fin(listaVentasHackeables)$ ▷ $\Theta(1)$
19: $res \leftarrow true$ ▷ $\Theta(1)$
20: **else**
21: $res \leftarrow false$ ▷ $\Theta(1)$

Complejidad: $\Theta(log(I))+\Theta(log(A))$ + $\Theta(1)$ = $\Theta(log(I) + log(A))$
Justificación: Gracias a utilizar los iteradores que apuntan a una venta Hackeable dentro del historial, no tenemos que recorrer la lista completa hasta encontrar una venta Hackeable. Esto reduce drásticamente la complejidad, que solo viene de la verificación de las claves y obtención de los significados de los diccionarios logarítmicos. Esto queda como: $\Theta(log(I) + log(A))$

**iCopiar**(**in** $p$: `puesto`) $\rightarrow$ $res$ : $puesto$

1: //Copio las estructuras de Puesto que no utilizan iteradores
2: $diccPrecioCopia \leftarrow copiar(p.diccPrecio)$ ▷ $\Theta(I)$
3: $diccStockCopia \leftarrow copiar(p.diccStock)$ ▷ $\Theta(I)$
4: $diccDescuentosCopia \leftarrow copiar(p.infoDescuentos)$ ▷ $\Theta(\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item))$
5: $diccVentasCopia \leftarrow copiar(p.diccVentas)$ ▷ $\Theta(VentasTotales)$
6: //Creo iteradores que apunten a las ventas copiadas Hackeables
7: $diccHackeosCopia \leftarrow Vacio()$ ▷ $\Theta(1)$
8: $itHackeos \leftarrow CrearIt(e.diccHackeos)$ ▷ $\Theta(1)$
9: //Por cada persona y por cada item, busco las ventas hackeables originales y creo los nuevos iteradores
10: **while** ($HaySiguiente(itHackeos)$ ▷ $\Theta(1)$
11: $personaHackeada \leftarrow SiguienteClave(itHackeos)$ ▷ $\Theta(1)$
12: $Definir(diccHackeoCopia, personaHackeada, vacio())$ ▷ $\Theta(Log(P))$
13: $DiccItemsHackCopia \leftarrow Significado(diccHackeoCopia, personaHackeada)$ ▷ $\Theta(Log(P))$
14: $itItemsHack \leftarrow CrearIt(SiguienteSignificado(itHackeos))$ ▷ $\Theta(1)$
15: //Por cada item, busco las ventas hackeables
16: **while** (($HaySiguiente(itItemsHack)$) ▷ $\Theta(1)$
17: $itemHackeado \leftarrow SiguienteClave(itItemsHack)$ ▷ $\Theta(1)$
18: $Definir(DiccItemsHackCopia, itemHackeado, vaca())$ ▷ $\Theta(Log(I))$
19: $ListaVentasHackCopia \leftarrow Significado(DiccItemsHackCopia, itemHackeado)$ ▷ $\Theta(Log(I))$
20: //Recorro las ventas copiadas y guardo los iteradores que apunten a una Venta Hackeable
21: //( Ya habia un iterador apuntando a ella en el puesto original)
22: $itListaVentasHackeables \leftarrow CrearIt(SiguienteSignificado(itItemsHack))$ ▷ $\Theta(1)$
23: $itListaVentas \leftarrow CrearIt(Significado(diccVentasCopia, personaHackeada).historial)$ ▷ $\Theta(Log(P))$
24: **while** ($HaySiguiente(itListaVentasHackeables)$) ▷ $\Theta(1)$
25: //Cuando encuentro una venta que coincida con los iteradores de la lista original
26: //guardo el iterador de la lista copiada
27: $iventaHackeable \leftarrow Siguiente(Siguiente(itListaVentasHackeables))$ ▷ $\Theta(1)$
28: $ventaActual \leftarrow Siguiente(itListaVentas)$ ▷ $\Theta(1)$
29: **if** ($ventaHackeable = ventaActual$) ▷ $\Theta(1)$
30: $Agregar(ListaVentasHackCopia, itListaVentas)$ ▷ $\Theta(1)$
31: //Busco la siguiente venta Hackeable
32: $Avanzar(itListaVentasHackeables)$ ▷ $\Theta(1)$
33: $Avanzar(itListaVentas)$ ▷ $\Theta(1)$
34: $res \leftarrow < diccPrecioCopia, diccStockCopia, diccDescuentosCopia, diccVentasCopia, diccHackeosCopia >$ ▷ $\Theta(1)$

Complejidad: $\Theta(I) + \Theta(\sum(Cant\ De\ Descuento\ Max\ De\ Cada\ Item)) + \Theta(VentasTotales) + \Theta(1) + \Theta(LogP) + \Theta(LogI) = \Theta(I + Cantidad\ total\ de\ Descuentos + A * I * CantidadVentasActuales)$

Justificación: Copiar un diccionario que tanto sus claves como sus significados son números enteros cuesta igual que la cantidad de sus claves (en este caso sería I). Luego debemos copiar los vectores de Descuentos, que cada uno tiene una longitud igual a la cantidad máxima del descuento del item. Por último y lo más complejo, debemos de que crear nuevos iteradores que apunten a la lista copiada en vez de la original, para esto primero debemos iterar por todas las personas, luego por cada item de cada persona y buscar las ventas no hackeables en su historial para crear sus iteradores. Es por esto que la complejidad termina siendo $\Theta(I + Cantidad\ total\ de\ Descuentos + A * I * CantidadVentasActuales)$