Algoritmos y Estructuras de Datos II

Diseño

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

Trabajo Práctico 2: Diseño

Alias del grupo:
WOWOANERNCTYDOHJTMEA

Integrante	LU	Correo electrónico
Barragán, Jerónimo	1472/21	barragan.jeronimo123@gmail.com
García Alurralde, Jorge	437/22	jalurralde@dc.uba.ar
Mamani Meza, Carlos Ezequiel	496/16	mamanimezacarlos@gmail.com
Sarkissian, Ralph Ezequiel	1698/21	ralphsarkissian.b@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Definiciones

puestoId es nat
personaId es nat
ítem es nat
cant es nat

2. Lollapatuza

2.1. Interfaz

se explica con: LOLLAPATUZA.

géneros: `lolla`.

Operaciones básicas de Lollapatuza

CREARLOLLA(**in** *ps*: `diccLog(puestoId, puesto)`, **in** *as*: `conjLineal(personaId)`) → *res* : `lolla`
Pre ≡ {vendenAlMismoPrecio(ps) ∧ novendieronAun(significados(ps)) ∧ ¬∅?(as) ∧ ¬∅?(claves(ps)) }
Post ≡ {*res* =$_{\text{obs}}$ crearLolla(ps, as)}
Complejidad: O(P*log(P) + A*log(A)), donde P es la cantidad de claves de ps, y A es el cardinal de as.
Descripción: Creamos un Lollapatuza a partir de un diccionario válido de puestos y un conjunto valido de personas.
Aliasing: Cualquier modificacion realizada a res afectara los significados de ps.

VENDER(**in/out** *l*: `lolla`, **in** *pi*: `puestoId`, **in** *a*: `personaId`, **in** *i*: `ítem`, **in** *c*: `cant`)
Pre ≡ {l =$_{\text{obs}}$ L_0 ∧ a ∈ personas(l) ∧ def?(pi, puestos(l)) ∧$_{\text{L}}$
(haySuficiente?(obtener(pi, puestos(l)), i, c) ∧ i ∈ menu(obtener(pi, puestos(l))))}
Post ≡ {l =$_{\text{obs}}$ vender(L_0, pi, a, i, c)}
Complejidad: O(log(A) + log(I) + log(P) + log(cant))
Descripción: Registra la compra de una cantidad de un ítem particular, realizada por una persona en un puesto.
Aliasing: No genera aliasing.

HACKEAR(**in/out** *l*: `lolla`, **in** *a*: `personaId`, **in** *i*: `ítem`)
Pre ≡ {l =$_{\text{obs}}$ L_0 ∧ consumioSinPromoEnAlgunPuesto(l, a, i)}
Post ≡ {l =$_{\text{obs}}$ hackear(L_0, a, i)}
Complejidad: O(log(A) + log(I)). En caso de que el puesto correspondiente deje de ser hackeable para esa persona e ítem luego de esta operación, la complejidad sera O(log(A) + log(I) + log(P))
Descripción: Realiza acción de hackeo de sistema, el cual remueve una unidad del ítem i si la compra fue realizada sin promoción.
Aliasing: No genera aliasing.

GASTOPERSONA(**in** *l*: `lolla`, **in** *a*: `personaId`) → *res* : `dinero`
Pre ≡ {a ∈ personas(l)}
Post ≡ {res =$_{\text{obs}}$ gastoTotal(l, a)}
Complejidad: O(log(A))
Descripción: Devuelve el gasto total de la persona.
Aliasing: No genera aliasing.

MASGASTÓ(**in** *l*: `lolla`) → *res* : `personaId`
Pre ≡ {¬Vacío(l.personas)}
Post ≡ {res =$_{\text{obs}}$ másGastó(l)}
Complejidad: O(1)
Descripción: Dada un festival, se devuelve el ID de la persona que mas dinero gastó. Si hay mas de una persona con el mismo gasto, se devuelve la de menor ID.
Aliasing: No genera aliasing.

PERSONAS(**in** *l*: `lolla`) → *res* : `conjLineal(personaId)`
Pre ≡ {true}
Post ≡ {res =$_{\text{obs}}$ personas(l)}
Complejidad: O(1)
Descripción: Devuelve el conjunto de personas del festival.
Aliasing: Se devuelve una referencia no modificable.

PUESTOS(**in** *l*: `lolla`) → *res* : `diccLog(puesto)`
Pre ≡ {true}
Post ≡ {res =$_{\text{obs}}$ puestos(l)}
Complejidad: O(1)
Descripción: Se devuelve un diccionario con los puestos
Aliasing: Se devuelve una referencia no modificable.

PUESTOCONMENORSTOCKDEÍTEM(**in** l : `lolla`, **in** i : `ítem`) → res : `puestoId`
Pre ≡ {¬Vacío(l.puestos)}
Post ≡ {res =$_{obs}$ menorStock(l, i)}
Complejidad: O(P * log(I))
Descripción: Devuelve el ID del puesto de menor stock para un item dado. Si hay más de un puesto que tiene stock mínimo, devuelve el de menor ID.
Aliasing: No genera aliasing.

PERTENECEPERSONA(**in** l : `lolla`, **in** a : `personaId`) → res : `bool`
Pre ≡ {True}
Post ≡ {res =$_{obs}$ (a ∈ personas(l))}
Complejidad: O(log(A))
Descripción: Indica si la persona está en el festival.
Aliasing: No genera aliasing.

PERTENECEPUESTO(**in** l : `lolla`, **in** pi : `puestoId`) → res : `bool`
Pre ≡ {True}
Post ≡ {res =$_{obs}$ def?(puestos(l), pi)}
Complejidad: O(log(P))
Descripción: Indica si el puesto está en el festival.
Aliasing: No genera aliasing.

VENDENALMISMOPRECIO(**in** ps : `diccLog(puestoId, puesto)`) → res : `bool`
Pre ≡ {True}
Post ≡ {res =$_{obs}$ true ⟺ vendenAlMismoPrecio(significados(ps))}
Complejidad: O(P× I × log(I))
Descripción: Dado un diccionario de puestos, verifico si cada ítem que vende alguno, tiene el mismo precio que en los otros que lo también venden.
Aliasing: No genera aliasing.

2.2. Representación

Representación de Lollapatuza

`lolla` **se representa con** `estr`

donde `estr` es `tupla`(*itPuestos*: `diccLog(puestoId, itDicc(puestoId, puesto))` ,
puestos: `diccLog(puestoId, puesto)` ,
conjPersonas: `conjLineal(personaId)` ,
personas: `diccLog(personaId, infoPersona)` ,
másGastadores: `diccLog(tupla<gasto : int, personaId>, personaId)`)

donde `infoPersona` es `tupla`(*gastoPersona*: `nat` ,
comprasSinDescuento: `diccLog(ítem, diccLog(puestoId, tupla<cantidad:nat, itPuesto : itDicc(puestoId, puesto)>))`)

Rep : estr ⟶ bool
Rep(e) ≡ true ⟺ 1 ∧$_L$ 2 ∧$_L$ 3 ∧$_L$ 4 ∧$_L$ 5 ∧$_L$ 6 ∧$_L$ 7 ∧$_L$ 8 ∧$_L$ 9

1. ¬∅(e.personas)
2. claves(e.personas) = e.conjPersonas
3. claves(e.puestos) = claves(e.itPuestos)
4. (∀ personaId : nat)(personaId ∈ e.conjPersonas ⇒$_L$ sumaDeGastos(personaId, significados(e.puestos)) = obtener(personaId, e.personas).gastoPersona)
5. (∀ puestoId : nat)(def?(puestoId, e.itPuestos) ⇒$_L$ (π_1(Siguiente(obtener(puestoId, e.itPuestos))) = puestoId ∧ π_2(Siguiente(obtener(puestoId, e.itPuestos))) = obtener(puestoId, e.puestos)))
6. (∀ t : tupla(int, nat))(def?(t, e.masGastadores) ⇒$_L$ (t.personaId ∈ e.conjPersonas ∧$_L$ obtener(t.personaId, e.personas).gastoPersona = -t.gasto))
7. #claves(e.masGastadores) = #e.conjPersona

8. (∀a : persona)(∀i : item)(∀p : puestoId)
(a ∈ e.conjPersonas ∧ def?(p, e.puestos) $\wedge_{\text{L}}$ i ∈ menu(obtener(p, e.puestos)) $\Rightarrow_{\text{L}}$
(#unidadesSinDescuento(a, i, obtener(p, e.puestos), ventas(obtener(p, e.puestos), a)) > 0 ⇒
(def?(i, obtener(a, e.personas).comprasSinDescuento) $\wedge_{\text{L}}$ def?(p, obtener(i, obtener(a, e.personas).comprasSinDescuento))
$\wedge_{\text{L}}$ π_1(obtener(p, obtener(i, obtener(a, e.personas).comprasSinDescuento))) = #unidadesSinDescuento(a, i, obtener(p, e.puestos), ventas(p, a)) ∧ Siguiente(π_2(obtener(p, obtener(i, obtener(a, e.personas).comprasSinDescuento))))
= obtener(p, e.itPuestos))))

9. (∀ a : persona)(∀ i : item)(∀ p : puestoid)
(a ∈ e.conjPersonas $\wedge_{\text{L}}$ def?(i, obtener(a, e.personas).comprasSinDescuento) $\wedge_{\text{L}}$ def?(p, obtener(i, obtener(a, e.personas).comprasSinDescuento)) $\Rightarrow_{\text{L}}$ (def?(p, e.puestos) $\wedge_{\text{L}}$ i ∈ menu(obtener(p, e.puestos)) ∧
obtener(p, obtener(i, obtener(a, e.personas).comprasSinDescuento)) =
⟨unidadesSinDescuento(a,i,obtener(p,e.puestos),ventas(obtener(p,e.puestos),a)) ,obtener(p, e.itPuestos)⟩)))))

sumaDeGastos : persona p × multiconj(puesto) c ⟶ nat
sumaDeGastos(p, c) ≡ **if** ∅?(c) **then** 0 **else** gastosDe(dameUno(c), p) + sumaDeGastos(p, sinUno(c)) **fi**

#unidadesSinDescuento : persona *per* × item i × puesto p × multiconj(⟨ítem × cant⟩) m ⟶ nat
{i ∈ menu(p) ∧ m ⊂ ventas(p, per)}
#unidadesSinDescuento(per, i, p, m) ≡ **if** ∅?(m) **then** 0
else if consumióSinPromo1Venta?(p,i, dameUno(m))
then π_2(dameUno(m))+#unidadesSinDescuento(per, i, p, sinUno(m))
else #unidadesSinDescuento(per, i, p, sinUno(m))
fi
fi

Abs : estr e ⟶ lolla {Rep(e)}
Abs(e) ≡ l : lolla / puestos(l) = e.puestos ∧ personas(l) = e.conjPersonas

2.3. Algoritmos

Algoritmos del módulo

Definimos una relación de orden entre tupla<int, nat> aprovechando que al crear un iterador a un diccLog, la siguiente clave es la mínima. Aclaración: Por cómo se construye la relación de orden, la mínima clave se corresponde con la persona que más gastó.

---

i• < **•(in** A : `tupla<`a_1`: int, `a_2`: nat>`, **in** B : `tupla<`b_1`: int, `b_2`: nat>`) → res : bool
1: $res \leftarrow a_1 < b_1 \parallel (a_1 = b_1$ && $a_2 < b_2)$ ▷ $O(1)$

Complejidad: $O(1)$

---

iCrearLolla(**in** ps : `diccLog(puestoId, puesto)`, **in** as : `conj(personaId)`) $\rightarrow$ res : estr

1: $itPuestos \leftarrow VACIO()$ ▷ $O(1)$
2: $puestos \leftarrow VACIO()$ ▷ $O(1)$
3: $it1 \leftarrow crearIt(ps)$ ▷ $O(1)$
4: **while** $haySiguiente(it1)$ **do** ▷ costo ciclo: $O(P * log(P))$
5: $it2 \leftarrow definir(puestos,\ siguienteClave(it1),\ siguienteSignificado(it1))$
6: $definir(itPuestos,\ siguienteClave(it1),\ it2)$
7: $Avanzar(it1)$ ▷ $O(1)$
8: **end while**
9: $personas \leftarrow VACIO()$ ▷ $O(1)$
10: $masGastadores \leftarrow VACIO()$ ▷ $O(1)$
11: $It3 \leftarrow crearIt(as)$ ▷ $O(1)$
12: **while** $haySiguiente(it3)$ **do** ▷ costo ciclo: $O(A * log(A))$
13: $definir(personas,\ siguiente(it3),\ \langle 0,\ VACIO()\rangle)$
14: $definir(masGastadores,\ \langle 0,\ siguiente(it3)\rangle,\ siguiente(it3))$
15: $Avanzar(it3)$
16: **end while**
17: $res \leftarrow \langle itPuestos,\ puestos,\ as,\ personas,\ masGastadores\rangle$

Complejidad: $O(P \times log(P) + A \times log(A))$
Justificación: Donde P es la cantidad de claves de ps, y A es el cardinal de as.

---

iVender(**in/out** l : `estr`, **in** pi : `puestoId`, **in** a : `personaId`, **in** i : `ítem`, **in** c : `cant`)

1: $puesto \leftarrow significado(estr.puestos,\ pi)$ ▷ Referencia al puesto; $O(log(P))$
2: $gastoVenta \leftarrow Vender(puesto,\ a,\ i,\ cant)$ ▷ $O(log(A)) + O(log(I)) + O(log(cant))$
3: $infoPersona \leftarrow significado(estr.personas,\ a)$ ▷ Referencia modificable; $O(log(A))$
4: $gastoActual \leftarrow infoPersona.gastoPersona$ ▷ $O(1)$
5: $nuevoGasto \leftarrow infoPersona.gastoPersona + gastoVenta$ ▷ $O(1)$
6: $BORRAR(estr.masGastadores, < -gastoActual,\ a >)$ ▷ $O(log(A))$
7: $DEFINIR(estr.masGastadores, < -nuevoGasto,\ a >,\ a)$ ▷ Actualización de másGastadores $O(log(A))$
8: $infoPersona.gastoPersona \leftarrow nuevoGasto$ ▷ Actualizar info persona; $O(1)$
9: $descuento \leftarrow obtenerDescuento(p, i, cant)$ ▷ $O(log(I)) + O(log(cant))$
10: **if** $descuento = 0$ ▷ Modificar comprasSinDescuento
11: **then if** $\neg definido?(infoPersona.comprasSinDescuento,\ i)$ ▷ $O(log(I))$
12: **then** $definir(infoPersona.comprasSinDescuento,\ i,\ Vacío())$ ▷ $O(log(I))$
13: **fi**
14: $comprasHackeables \leftarrow significado(infoPersona.comprasSinDescuento,\ i)$ ▷ $O(log(I)$
15: **if** $\neg definido?(comprasHackeables, pi)$
16: **then** $itPuesto \leftarrow obtener(estr.itPuestos,\ pi)$ ▷ $O(log(P))$
17: $definir(comprasHackeables,\ pi,\ < 0,\ itPuesto >)$ ▷ $O(log(P))$
18: **else** $cantidad \leftarrow significado(comprasHackeables,\ pi).cantidad$ ▷ Ref. mod.; $O(log(P))$
19: $cantidad \leftarrow cantidad + cant$ ▷ $O(1)$
20: **fi**
21: **fi**
22: **else** $cantidad \leftarrow significado(comprasHackeables,\ pi).cantidad$ ▷ $O(log(P))$
23: $cantidad \leftarrow cantidad + cant$ ▷ $O(1)$
24: **fi**

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

iHackear(**in/out** l : `estr`, **in** a : `personaId`, **in** i : `ítem`)

1: $infoPersona \leftarrow significado(estr.personas,\ a)$ ▷ Ref modificable; $O(log(A))$
2: $puestosHackeables \leftarrow significado(infoPersona.comprasSinDescuento,\ i)$ ▷ $O(log(I))$
3: $itMin \leftarrow crearIt(puestosHackeables)$ ▷ (clave, significado); $O(1)$
4: $id\_puesto \leftarrow siguienteClave(itMin)$ ▷ $O(1)$
5: $itPuesto \leftarrow siguienteSignificado(itMin).itPuesto$ ▷ obtengo: it al dicc (clave, significado); $O(1)$
6: $puesto \leftarrow siguienteSignificado(itPuesto)$ //O(1)
7: $precioItem \leftarrow precio(puesto,\ i)$ ▷ Actualizo infoPersona y masGastadores; $O(log(I))$
8: $gastoActual \leftarrow infoPersona.gastoPersona$ ▷ Tipo primitivo en tupla por referencia se pasa por copia; $O(1)$
9: $BORRAR(masGastadores, < -gastoActual,\ a >)$ ▷ $O(log(A))$
10: $gastoActual \leftarrow infoPersona.gastoPersona - precioItem$ ▷ $O(1)$
11: $DEFINIR(masGastadores, < -gastoActual,\ a >,\ a)$ ▷ Agrego; $O(log(A-1)) = O(log(A))$
12: $infoPersona.gastoPersona \leftarrow gastoActual$ ▷ $O(1)$
13: $OlvidarItem(puesto,\ a,\ i)$ ▷ $O(log(A)) + O(log(I))$
14: $itemsHackeables \leftarrow siguienteSignificado(itMin).cantidad - 1$ ▷ $O(1)$
15: **if** $itemsHackeables > 0$ ▷ ¿Sigue siendo hackeable el puesto?
16: **then** $siguienteSignificado(itMin) \leftarrow < itemsHackeables,\ itPuesto >$ ▷ $O(1)$
17: **else** $BORRAR(puestosHackeables,\ id\_puesto)$ ▷ $O(log(P))$
18: **fi**

Complejidad: $O(log(A) + log(I)) \vee O(log(A) + log(I) + log(P))$
Justificación: Antes del if tenemos complejidad O(log(A) + log(I)); si la guarda es verdadera, O(if) = O(1) → O(log(A) + log(I)) + O(1) = O(log(A) + log(I)); si no, O(if) = O(log(P)) → O(log(A) + log(I)) + O(log(P)) = O(log(A) + log(I) + log(P))

---

igastoPersona(**in** l : `estr`, **in** a : `personaId`) → res : nat

1: $dinero \leftarrow Significado(estr.personas,\ a).gastoPersona$ ▷ $O(log(A)) + O(1)$

Complejidad: $O(log(A))$

---

imásGastó(**in** l : `estr`) → res : personaId

1: $itMin \leftarrow crearIt(estr.masGastadores)$ ▷ $O(1)$
2: $res \leftarrow Copiar(SiguienteSignificado(itMin))$ ▷ $O(1)$

Complejidad: $O(1)$

ipuestoConMenorStockDeÍtem(**in** l : `estr`, **in** i : `íten`) → res : puestoId

```
1: it ← crearIt(estr.puestos)
2: id_puestoMenorStock ← siguienteClave(it)                          ▷ puestoId
3: puestoMenorStock ← siguienteSignificado(it)
4: menorStock ← obtenerStock(puestoMenorStock, i)                    ▷ O(log(I))[*]
5: avanzar(it)
6: while haySiguiente(it) do                                         ▷ P veces → P * O(log(I)) = O(P × log(I))
7:     id_puestoActual ← siguienteClave(it)
8:     puestoActual ← siguienteSignificado(it)
9:     if estaEnMenu?(puestoActual, i)                               ▷ O(log(I))
10:      then stockActual ← obtenerStock(puestoActual, i)            ▷ O(log(I))
11:           if stockActual < menorStock
12:             then menorStock ← stockActual
13:                  id_puestoMenorStock ← siguienteClave(it)
14:           if stockActual = menorStock && id_puestoActual < id_puestoMenorStock
15:             then menorStock ← stockActual
16:                  id_puestoMenorStock ← id_puestoActual
17:    avanzar(it)
18: end while
19: res ← copiar(id_puestoMenorStock)                                ▷ O(1)
```

Complejidad: $O(P \times log(I))$
Justificación: [∗] Para consultar stock, i debe pertenecer al puesto (pasa lo mismo cuando inicializo menorStock).

---

ipertenecePersona(**in** l : `estr`, **in** a : `personaId`) → res : bool

```
1: res ← pertenece?(estr.personas, a)                                ▷ O(log(A))
```

Complejidad: $O(log(A))$

---

ipertenecePuesto(**in** l : `estr`, **in** pi : `puestoId`) → res : bool

```
1: res ← definido?(estr.puestos, pi)                                 ▷ O(log(P))
```

Complejidad: $O(log(P))$

---

iPersonas(**in** l : `estr`) → res : conjLineal(personaId) no modificable

```
1: res ← estr.conjPersonas                                           ▷ Referencia no modificable; O(1)
```

Complejidad: $O(1)$

---

iPuestos(**in** l : `estr`) → res : diccLog(puestoId, puesto) no modificable

```
1: res ← estr.puestos                                                ▷ Referencia no modificable; O(1)
```

Complejidad: $O(1)$

ivendenAlMismoPrecio(**in** ps : `DiccLog(puestoId, puesto)`) $\rightarrow$ res : bool

```
1: it ← crearIt(ps)                                                          ▷ Θ(1)
2: id_puestoActual ← siguienteClave(it)
3: puestoActual ← siguienteSignificado(it)
4: diccLog : todosLosItems ← vacio()
5: agregarItemsNoContenidos(todosLosItems, puestoActual.menu)
6: avanzar(it)
7: res ← true
8: while haySiguiente(it) &&res do          ▷ P veces → P × 2 · O(I × log(I)) = O(P × I × log(I))
9:     id_puestoActual ← siguienteClave(it)
10:    puestoActual ← siguienteSignificado(it)
11:    res ← res&&aMismoItemMismoPrecio(todosLosItems, puestoActual.menu)
12:    agregarItemsNoContenidos(todosLosItems, puestoActual.menu)
13:    avanzar(it)
14: end while
```

Complejidad: $O(P \times I \times log(I))$

Esta es una operación privada que recibe dos menús, agregando al primer menú los del segundo que no están definidos el primero:

iagregarItemsNoContenidos(**in/out** $granMenu$: `diccLog(ÍtemId, nat)`, **in** $menuMenor$: `diccLog(ÍtemId, nat)`)

```
1: it_menuMenor ← crearIt(menuMenor)                                         ▷ O(1)
2: id_itemMenuMenor ← siguienteClave(it_menuMenor)
3: precioMenuMenor ← siguienteSignificado(it_menuMenor)
4: while haySiguiente(it_menuMenor) do      ▷ Peor caso, recorro todos los ítems del festival O(I × log(I))
5:     if ¬definido?(granMenu, id_itemMenuMenor)
6:         then definir(granMenu, id_itemMenuMenor, precioMenuMenor)         ▷ O(log(I))
7:     fi
8:     avanzar(it_menuMenor)
9: end while
```

Complejidad: $O(I \times log(I))$

Esta es una operación privada que recibe dos menús, verificando si todos los items del segundo cuestan lo mismo que en el primero:

iaMismoItemMismoPrecio(**in** $granMenu$: `diccLog(ÍtemId, nat)`, **in** $menuMenor$: `diccLog(ÍtemId, nat)`) $\rightarrow$ res : bool

```
1: it_menuMenor ← crearIt(menuMenor)                                         ▷ O(1)
2: id_itemMenuMenor ← siguienteClave(it_menuMenor)
3: precioMenuMenor ← siguienteSignificado(it_menuMenor)
4: res← true
5: while haySiguiente(it_menuMenor) && res do  ▷ Peor caso, recorro todos los ítems del festival O(I × log(I))
6:     if definido?(granMenu, id_itemMenuMenor)
7:         then res ← res && (significado(granMenu, id_itemMenuMenor) = precioMenuMenor)   ▷ O(log(I))
8:     fi
9:     avanzar(it_menuMenor)
10: end while
```

Complejidad: $O(I \times log(I))$

3. Puesto De Comidas

3.1. Interfaz

se explica con: PuestoDeComida.

géneros: `puesto`.

Operaciones básicas de Puesto De Comida

CrearPuesto(**in** *menu*: `diccLog(ítem, nat)`, **in** *stock*: `diccLog(ítem, nat)`, **in** *promos*: `diccLog(ítem, diccLog(nat, nat))`) → *res* : `puesto`
Pre ≡ {claves(menu) =$_{obs}$ claves(stock) ∧ claves(promos) ⊂ claves(menu)}
Post ≡ {res =$_{obs}$ crearPuesto(menu, stock, promos)}
Complejidad: O(I × (D × log(N) + N × log(N))). Donde N es la maxima cantidad para la que esta definida un descuento de un item en promos. D es el maximo cardinal de los descuentos pasados por parametro, definidos para un item.
Descripción: Crea nuevo puesto de comida.
Aliasing: Devuelve una referencia modificable.

Vender(**in/out** p: `puesto`, **in** a: `idPersona`, **in** i: `item`, **in** *cant*: `nat`) → *res* : `nat`
Pre ≡ { $p =_{obs} p_0 \wedge$ haySuficiente?$(p_0, i, cant)$}
Post ≡ {$p =_{obs}$ vender$(p_0, a, i, cant)$}
Complejidad: O(log(A) + log(I) + log(cant))
Descripción: Realiza la venta de una cantidad determinada del item, para una persona. Se devuelve como resultado el gasto de la venta.
Aliasing: No genera aliasing.

Precio(**in** p: `puesto`, **in** i: `ítem`) → *res* : `nat`
Pre ≡ {i ∈ menú(p)}
Post ≡ {res =$_{obs}$ precio(p, i)}
Complejidad: O(log(I))
Descripción: Devuelve el precio del item en el puesto p.
Aliasing: No genera aliasing.

ObtenerStock(**in** p: `puesto`, **in** i: `ítem`) → *res* : `nat`
Pre ≡ {i ∈ menú(p)}
Post ≡ {res =$_{obs}$ stock(p, i)}
Complejidad: O(log(I))
Descripción: Devuelve el stock de un ítem en un puesto determinado.
Aliasing: No genera aliasing.

ObtenerDescuento(**in** p: `puesto`, **in** i: `ítem`, **in** c: `cant`) → *res* : `nat`
Pre ≡ {i ∈ menú(p)}
Post ≡ {res =$_{obs}$ descuento(p, i, c)}
Complejidad: O(log(cant) + log(I))
Descripción: Devuelve el descuento a aplicarse sobre la compra de una cierta cantidad de ítems en un puesto. Si no hay un descuento definido para esa cantidad, se devuelve 0.
Aliasing: No genera aliasing.

ObtenerGasto(**in** p: `puesto`, **in** a: `personaId`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ gastosDe(p, a)}
Complejidad: O(log(A))
Descripción: Devuelve el menú del puesto de comida.
Aliasing: No genera aliasing.

OlvidarItem(**in/out** p: `puesto`, **in** a: `personaId`, **in** i: `ítem`)
Pre ≡ {p =$_{obs}$ $P_0 \wedge_L$ i ∈ menu(P_0) $\wedge_L$ consumioSinPromo?(P_0, a, i)}
Post ≡ {p =$_{obs}$ olvidarÍtem(P_0, a, i) }
Complejidad: O(log(A) + log(I))
Descripción: Dados un puesto, persona e ítem, borra un venta que involucra a los tres.
Aliasing: No genera aliasing.

ConsumióSinPromo?(**in** p: `puesto`, **in** a: `personaId, in` i : `ítem`) → res : `bool`
Pre ≡ {i ∈ menú(p)}
Post ≡ {res =$_{obs}$ consumióSinPromo?(p, a, i)}
Complejidad: O(log(A) + log(I))
Descripción: Devuelve un valor de verdad respecto si hubo una venta sin promoción dados un puesto, persona e ítem.
Aliasing: No genera aliasing.

EstáEnMenú?(**in** p: `puesto`, **in** i: `ítem`) → res : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ i ∈ menú(p)}
Complejidad: O(log(I))
Descripción: Devuelve true si el ítem se encuentra en el menú, falso en otro caso.
Aliasing: No genera aliasing.

3.2. Representación

Representación del Puesto De Comidas

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

donde `estr` es `tupla`(*menu*: `diccLog(ítemId, nat)` ,
stock: `diccLog(ítemId, nat)` ,
promociones: `diccLog(ítemId, infoPromos)` ,
gastos: `diccLog(personaId, nat)` ,
ventas: `diccLog(personaId, diccLog(item, infoVentas))`)
donde `infoPromo` es `tupla`(*minN*: `nat`,
maxN: `nat`,
descuentos: `diccDig(cantidad, descuento)`)
donde `infoVentas` es `tupla`(*ventasConDescuento*: `listaEnlazada(cant)`
,
ventasSinDescuento: `listaEnlazada(cant)`)

Rep : estr ⟶ bool
Rep(e) ≡ true ⟺ 1 $\wedge_L$ 2 $\wedge_L$ 3 $\wedge_L$ 4 $\wedge_L$ 5 $\wedge_L$ 6 $\wedge_L$ 7 $\wedge_L$ 8

1. claves(e.menu) = claves(e.stock)
2. claves(e.gastos) = claves(e.ventas)
3. claves(e.promociones) ⊂ claves(e.menu)
4. (∀ i : item)(def?(i, e.promociones) $\Rightarrow_L$
   (def?(obtener(i, e.promociones).minN, obtener(i, e.promociones).descuentos) ∧
   def?(obtener(i, e.promociones).maxN, obtener(i,e.promociones).descuentos) $\wedge_L$
   ¬ (∃ o : cant)(def?(o, obtener(i,e.promociones)) ∧ (o < obtener(i, e.promociones).minN ∨
   o > obtener(i, e.promociones).maxN)))))
5. (∀ i : item)(def?(i, e.promociones) $\Rightarrow_L$
   (∀ k : nat)(obtener(i, e.promociones).minN ≤ k ≤ obtener(i, e.promociones).maxN ⇒
   def?(k, obtener(i, e.promociones).descuentos))
6. (∀ a : persona)(∀ i : item) ((a ∈ claves(e.ventas) $\wedge_L$ def?(i, obtener(a, e.ventas)) $\Rightarrow_L$ def?(i, e.menu)))
7. (∀ a : persona)(∀ i : item)(a ∈ claves(e.ventas) $\wedge_L$ def?(i, obtener(a, e.ventas)) $\Rightarrow_L$
   (∀ $cant_1$: nat)(esta?($cant_1$, obtener(i, obtener(a, e.ventas)).ventasConDescuento) $\Rightarrow_L$
   $cant_1 > 0 \wedge cant_1 \geq$ obtener(i, obtener(a, e.ventas)).minN) ∧
   (∀ $cant_2$: nat) (esta?($cant_2$, obtener(i, obtener(a, e.ventas)).ventasSinDescuento) $\Rightarrow_L$
   $cant_2 > 0 \wedge cant_2 <$ obtener(i, obtener(a, e.ventas)).minN))
8. (∀ a : persona)(a ∈ claves(e.gastos) $\Rightarrow_L$
   (obtener(a, e.gastos) = gastoTotal(e.menu, obtener(a, e.ventas), e.promociones)))

gastoTotal : {dicc(item, nat) *precios* × dicc(item, tupla(secu(nat), secu(nat))) *ventas* × dicc(item, tupla(nat, nat, dicc(nat, nat))) $info$} ⟶ nat
{claves(ventas) ⊂ claves(precios) ∧ claves(info) = claves(precios)}

gastoTotal(precios, venta, info) $\equiv$ **if** ventas = vacío **then** 0
else if def?(dameUno(claves(ventas)), info)
then gastoSinDescuento(obtener(dameUno(claves(ventas)), precios),
π_2 (obtener(dameUno(claves(ventas)), ventas))) +
gastoConDescuento(obtener(dameUno(claves(ventas)), precios),
π_3(obtener(dameUno(claves(ventas))), info),
π_1 (obtener(dameUno(claves(ventas)), ventas))) +
gastoTotal(precios, sinUnaClave(ventas), info)
else gastoSinDescuento(obtener(dameUno(claves(ventas)), precios),
π_2(obtener(dameUno(claves(ventas)), ventas))) +
gastoTotal(precios, sinUnaClave(ventas), info)
fi
fi

gastoSinDescuento : nat *precio* $\times$ secu(nat) s $\longrightarrow$ nat
gastoSinDescuento(precio, s) $\equiv$ **if** Vacía?(s) **then** 0 **else** prim(s) $\times$ precio + gastoSinDescuento(precio, fin(s)) **fi**

gastoConDescuento : nat *precio* $\times$ dicc(nat, nat) d $\times$ secu(nat) s $\longrightarrow$ nat
{($\forall$ n: nat)(Está?(n, s) $\Rightarrow$ def?(n, d))}
gastoConDescuento(precio, d, s) $\equiv$ **if** Vacía?(s) **then** 0 **else** aplicarDescuento(precio $\times$ prim(s), obtener(prim(s), d))
+ gastoConDescuento(precio, d, fin(s)) **fi**

Abs : estr e $\longrightarrow$ puesto {Rep(e)}
Abs(e) $\equiv$ p : puesto /
menu(p) = claves(e.menu) $\wedge_{\text{L}}$ ($\forall$ i : item)(i $\in$ menu(p) $\Rightarrow_{\text{L}}$
precio(p, i) = obtener(i, e.menu) $\wedge$ stock(p, i) = obtener(i, e.stock) $\wedge$ ($\neg$def?(i, e.promociones) $\Rightarrow$
($\forall$ c : cant)(descuento(p, i, c) = 0)) $\wedge$ (def?(i, e.promociones) $\Rightarrow$
($\forall$ c : cant)((c<obtener(i, e.promociones).mínN descuento(p,i,c) = 0)) $\vee$
(obtener(i, e.promociones).mínN $\leq$ c $\leq$ obtener(i, e.promociones).maxN $\wedge$
descuento(p, i, c) = obtener(c, obtener(i, e.promociones).descuentos)) $\vee$
(c > obtener(i, e.promociones).máxN $\wedge$ descuento(p, i,c) = obtener(obtener(i, e.promociones).máxN,
obtener(i, e.promociones).descuentos))) $\wedge_{\text{L}}$
($\forall$ a : persona)(($\neg$def?(a, e.personas) $\Rightarrow_{\text{L}}$
ventas(p, a) = $\emptyset$) $\wedge$ (def?(a, e.personas) $\Rightarrow_{\text{L}}$
ventas(p, a) = unirVentas(obtener(a, e.ventas))))

unirVentas : dicc(item, $\langle$secu(nat), secu(nat)$\rangle$) d $\longrightarrow$ multiconj($\langle$ítem, nat$\rangle$)
unirVentas(d) $\equiv$ **if** d $=_{\text{obs}}$ vacío() **then** $\emptyset$
else SecuAMulticonj(dameUno(claves(d)), π_1(obtener(dameUno(claves(d)), d))) $\cup$
SecuAMulticonj(dameUno(claves(d)), π_2(obtener(dameUno(claves(d)), d))) $\cup$
unirVentas(sinUnaClave(d))
fi
SecuAMulticonj : ítem i $\times$ secu(nat) s $\longrightarrow$ multiconj($\langle$ítem, nat$\rangle$)
SecuAMulticonj(i, s) $\equiv$ **if** Vacía?(s) **then** $\emptyset$ **else** Ag($\langle$i, prim(s)$\rangle$, SecuAMulticonj(i, fin(s))) **fi**

3.3. Algoritmos

Algoritmos del módulo

iCrearPuesto(**in** *menu*: `diccLog(item, nat)`, **in** *stock*: `diccLog(item, nat)`, **in** *promos*: `diccLog(item, diccLog(nat, nat))`) $\rightarrow$ *res* : puesto

```
1: MenuCopia ← copiar(menu)                                                    ▷ O(I)
2: StockCopia ← copiar(stock)                                                  ▷ O(I)
3: Gastos ← VACIO()                                                            ▷ O(1)
4: Ventas ← VACIO()                                                            ▷ O(1)
5: promociones ← VACIO()                                                       ▷ O(1)
6: it1 ← crearIt(promos)                                                       ▷ O(1)
7: while HaySiguiente(it1) do          ▷ I veces el primer while; I × O((D× log(N) + N × log(N)))
8:     item ← Siguiente(it1)                                                   ▷ O(1)
9:     descuentos ← VACIO()                                                    ▷ O(1)
10:    It2 ← crearIt(obtener(promos, item))                                    ▷ O(1)
11:    minN ← SiguienteClave(it2)                                              ▷ O(1)
12:    maxN ← 0                                                                ▷ O(1)
13:    while HaySiguiente(it2) do          ▷ costo de ciclo: D veces × O(log(N))
14:        definir(descuentos, SiguienteClave(it2), SiguienteSignificado(it2)) ▷ O(log(N))
15:        maxN ← SiguienteClave(it2)                                          ▷ O(1)
16:        Avanzar(it2)                                                        ▷ O(1)
17:    end while
18:    ultimoDescuento ← obtener(descuentos, minN)                             ▷ O(1)
19:    for int j = minN; j <= maxN; j++ do   ▷ N veces como mucho; O(N × log(N))
20:        if !definido?(descuentos, j)
21:          then definir(descuentos, j, ultimoDescuento)          ▷ diccDigital; O(log(N))
22:          else ultimoDescuento ← obtener(descuentos, j)
23:        fi
24:    end for
25:    definir(promociones, item, ⟨mnN, mxN, descuentos⟩)                      ▷ O(log(I))
26:    Avanzar(it1)
27: end while
28: res ← ⟨MenuCopia, StockCopia, promociones, Gastos, Ventas⟩
```

Complejidad: $O(I \times (D \times log(N) + N \times log(N)))$
Justificación: [$*$] N es el máxima cantidad para la que esta definida un descuento. D es el máximo cardinal de los descuentos pasados por parámetro, definidos para un ítem.

---

iObtenerDescuento(**in** p: `estr`, **in** i: `ítem`, **in** c: `nat`) $\rightarrow$ *res* : nat

```
1: res ← 0                                                                     ▷ O(1)
2: if definido?(p.promociones, i)                                              ▷ O(log(I))
3:   then infoP ← obtener(p.promociones, i)                                    ▷ O(log(I))
4:        if infoPromo → maxN ≤ cant                                           ▷ O(1)
5:          then res ← obtener(infoPromo → descuentos, infoPromo → maxN)       ▷ O(log(cant))
6:          else if infoPromo → minN ≤ cant                                    ▷ O(1)
7:                 then res ← obtener(infoPromo → descuentos, infoPromo → cant) ▷ O(log(cant))
8:               fi
9:        fi
10: fi
```

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

---

iObtenerGasto(**in** p: `estr`, **in** a: `personaId`) $\rightarrow$ *res* : nat

```
1: gasto ← obtener(p.gastos, a)                                                ▷ O(log(A))
2: res ← copiar(gasto)                                                         ▷ O(1)
```

Complejidad: $O(log(A))$

---

iVender(**in/out** p: `estr`, **in** a: `personaId`, **in** i: `ítem`, **in** c: `nat`) $\rightarrow$ res : nat

1: $precio \leftarrow significado(p.menu,\ i)$ ▷ $O(log(I))$
2: $precioSinDescuento \leftarrow precio * cant$ ▷ $O(1)$
3: $desc \leftarrow obtenerDescuento(p,\ i,\ cant)$ ▷ $O(log(I) + log(cant))$
4: $gasto \leftarrow aplicarDescuento(precioSinDescuento,\ desc)$ ▷ $O(1)$
5: **if** $\neg definido?(p.gastos, a)$ **then** ▷ $O(log(A))$
6: $definir(p.gastos, a, 0)$ ▷ $O(log(A))$
7: $gastoActual \leftarrow significado(p.gastos,\ a)$ ▷ Referencia modificable; $O(log(A))$
8: $gastoActual \leftarrow gastoActual + gasto$ ▷ $O(1)$
9: $stock \leftarrow significado(p.stock,\ i)$ ▷ Referencia modificable; $O(log(I))$
10: $stock \leftarrow stock - cant$ ▷ $O(1)$
11: //Actualizo historial.
12: **if** $\neg definido?(p.ventas, a)$ **then** ▷ $O(log(A))$
13: $definir(p.ventas, a, Vacio())$ ▷ $O(log(A))$
14: $historial \leftarrow significado(p.ventas,\ a)$ ▷ $O(log(A))$
15: **if** $\neg definido?(historial,\ i)$ ▷ $O(log(I))$
16: **then** $definir(historial,\ i,\ tupla\langle VACÍA,\ VACÍA\rangle)$ ▷ $O(log(I))$
17: **fi**
18: $historial\_item \leftarrow significado(historial,\ i)$ ▷ O(log(I))
19: **if** $desc = 0$
20: **then** $AgregarAdelante(historial\_item.ventasSinDescuento,\ cant)$ ▷ $O(1)$
21: **else** $AgregarAdelante(historial\_item.ventasConDescuento,\ cant)$ ▷ $O(1)$
22: **fi**
23: $res \leftarrow gasto$

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

---

iOlvidarItem(**in** p: `estr`, **in** a: `personaId`, **in** i: `ítem`) $\rightarrow$ res : nat

1: $precio \leftarrow significado(p.menu,\ i)$ ▷ $O(log(I))$ //
2: $gasto \leftarrow significado(p.gastos,\ a)$ ▷ Referencia modificable; $O(log(A))$
3: $gasto \leftarrow gasto - precio$
4: $stock \leftarrow significado(p.stock,\ i)$ ▷ Referencia modificable; $O(log(I))$
5: $stock \leftarrow stock + 1$ ▷ $O(1)$
6: $historial \leftarrow significado(p.ventas,\ a)$ ▷ $O(log(A))$
7: $historial\_item \leftarrow significado(historial,\ i)$ ▷ $O(log(I))$
8: $cantidadConsumida \leftarrow primero(historial\_item.ventasSinDescuento)$ ▷ Referencia modificable
9: **if** $cantidadConsumida = 1$
10: **then** $fin(historial\_item.ventasSinDescuento)$ ▷ $O(1)$
11: **else** $cantidadConsumida \leftarrow cantidadConsumida - 1$ ▷ $O(1)$
12: **fi**

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

---

iConsumioSinPromo?(**in** p: `estr`, **in** a: `personaId`, **in** i: `ítem`) $\rightarrow$ res : bool

1: $historialVentas \leftarrow significado(p.ventas,\ a)$ ▷ $O(log(A))$
2: $ventasSinDesc \leftarrow significado(historialVentas,\ i).ventasSinDescuento$ ▷ $O(log(I))$
3: $res \leftarrow EsVacia?(ventasSinDesc)$ ▷ $O(1)$

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

---

iPrecio(**in** p: `estr`, **in** i: `ítem`) $\rightarrow$ res : nat

1: $res \leftarrow significado(p.menú,\ i)$ ▷ $O(log(I))$

Complejidad: $O(log(I))$

iObtenerStock(**in** p: `estr`, **in** i: `ítem`) $\rightarrow$ res : nat

1: $res \leftarrow significado(p.stock,\ i)$ ▷ $O(log(I))$

Complejidad: $O(log(I))$

EstaEnMenu?(**in** p: `estr`, **in** i: `ítem`) $\rightarrow$ res : bool

1: $res \leftarrow definido?(p.menú,\ i)$ ▷ $O(log(I))$

Complejidad: $O(log(I))$

4. Diccionario Digital

4.1. Interfaz

parámetros formales
géneros σ
función COPIAR(**in** s: α) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ s}
Complejidad: Θ(copy(s))
Descripción: función de copia de σ's COPIAR(**in** a: α) $\rightarrow$ res : α función de copia, con costo temporal $\Theta(copy(a))$.

se explica con: DICCIONARIO(NAT, σ).

géneros: `diccDig(nat, σ)`.

Operaciones básicas de Diccionario Digital

VACÍO() $\rightarrow$ res : `diccDig(nat, σ)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ vacío}
Complejidad: O(1)
Descripción: Crea diccionario vacío.
Aliasing: No genera aliasing.

DEFINIR(**in/out** d: `diccDig(nat, σ)`, **in** k: `nat`, **in** s: σ)
Pre $\equiv$ {d = D_0}
Post $\equiv$ {d $=_{\text{obs}}$ definir(k, s, D_0)}
Complejidad: O(log(k))
Descripción: Definimos s en la clave k
Aliasing: Define una referencia a s en el diccionario.

DEFINIDO?(**in/out** d: `diccDig(nat, σ)`, **in** k: `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ def?(k, d)}
Complejidad: O(log(k))
Descripción: Revisa si k está definido.
Aliasing: No genera aliasing.

SIGNIFICADO(**in/out** d: `diccDig(nat, σ)`, **in** k: `nat`, **in** s: σ) $\rightarrow$ res : σ
Pre $\equiv$ {def?(k, d)}
Post $\equiv$ {res $=_{\text{obs}}$ obtener(k, d)}
Complejidad: O(log(k))
Aliasing: Devuelve una referencia modificable.

4.2. Representación

Representación del Diccionario Digital

`diccDig` **se representa con** `estr`

donde `estr` es `tupla`(*raíz*: `puntero (nodo)`)
donde `nodo` es `tupla`(*hijos*: `arreglo_estático[10] de puntero(nodo)` ,
valor: `puntero(σ)`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ (raíz = NULL $\vee_{\text{L}}$ (raíz[0] = NULL)) $\wedge$ invarianteDigital(*raíz)

Abs : estr e $\longrightarrow$ diccDig(nat. σ) {Rep(e)}
Abs(e) $\equiv$ d : diccDig(nat, σ) / ($\forall$ k : nat)(estáDefinida?(k, *e.raíz, dígitos(k)) $\iff$ def?(j, d) $\wedge_{\text{L}}$
($\forall$ h : nat)(def?(h, d) $\Rightarrow_{\text{L}}$ significado(h, *e.raíz) $=_{\text{obs}}$ obtener(h, d)))

invarianteDigital : nodo n $\times$ secu(puntero(nodo)) $hijos$ $\longrightarrow$ bool

invarianteDigital(n, hijos) $\equiv$ n.valor = NULL $\Rightarrow$ tieneHijos(n) $\wedge$ invarianteSobreHijos(n.hijos)

invarianteSobreHijos : secu(puntero(nodo)) s $\longrightarrow$ bool
invarianteSobreHijos(s) $\equiv$ **if** $\neg$vacía(s)
then (prim(s) = NULL $\vee_{\text{L}}$ invarianteDigital(*prim(s), prim(s)→hijos)) $\wedge$
invarianteSobreHijos(fin(s))
else true
fi

estáDefinida? : nat k × nodo n × nat c) $\longrightarrow$ bool {c $\geq$ 1}
estáDefinida?(k, n, c) $\equiv$ **if** c = 1
then if n = 0
then n.valor $\neq$ NULL
else nÉsimo(primerDígito(K), n.hijos) $\neq$ NULL $\wedge_{\text{L}}$
nÉsimo(primerDígito(K), n.hijos) → valor = NULL
fi
else if nÉsimo(primerDígito(K), n.hijos) = NULL
then false
else estáDefinida?(sinPrimerDígito(k), *nÉsimo(primerDígito(k), n.hijos), c-1)
fi
fi

significado : nat k × nodo n × nat c $\longrightarrow$ bool {estáDefinida(k, n)}
significado(k, n, c) $\equiv$

dígitos : nat n $\longrightarrow$ nat
dígitos(n) $\equiv$ dígitosAux(n, 1)

dígitosAux : nat n × nat m $\longrightarrow$ nat
dígitosAux(n, m) $\equiv$ **if** n mod 10 ^ m = n **then** m **else** dígitosAux(n, m+1) **fi**

primerDígito : nat n $\longrightarrow$ nat
primerDígito(n) $\equiv$ n div 10 ^ digitos(n)-1

sinPrimerDígito : nat n $\longrightarrow$ nat {dígitos(n) > 1}
sinPrimerDígito(n) $\equiv$ n mod 10 ^ digitos(n)-1

nÉsimo : nat n × secu(α s) $\longrightarrow$ α {n < long(s)}
nÉsimo(n, s) $\equiv$ **if** n = 0 **then** prim(s) **else** nÉsimo(n-1, fin(s)) **fi**

div : nat n × nat k $\longrightarrow$ nat
div(n, k) $\equiv$ **if** n < k **then** 0 **else** 1+ div(n-k, k) **fi**

• mod • : nat n × nat k $\longrightarrow$ nat {k > 1}
n mod m $\equiv$ n − div(n, k) × k

• ^ • : nat n × nat p $\longrightarrow$ nat {n = 0 $\Rightarrow$ k > 0}
n ^ k $\equiv$ **if** k = 0 **then** 1 **else** n × (n^(k-1)) **fi**

4.3. Algoritmos

Algoritmos del módulo

iVacío() → *res* : estr

1: $varAE : arreglo\_estático[10]\ de\ puntero(nodo)$ ▷ $O(1)$
2: **for** $int\ i = 0;\ i < 10;\ i++$ **do** ▷ $10 \times O(1) = O(1)$
3: $AE[i] \leftarrow NULL$ ▷ $O(1)$
4: **end for**
5: $res \leftarrow \langle \&\langle AE, NULL\rangle\rangle$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: $O(1) + O(1) + O(1) = O(1)$.

Esta es una operación privada que recibe un natural n cualquiera y devuelve en res su cantidad de dígitos:

iDígitos(**in** n : `nat`) → *res* : nat

1: $res \leftarrow 1$ ▷ $O(1)$
2: **while** $n\ mod\ 10\hat{}res\ \neq n$ **do** ▷ $[*]$
3: $res++$ ▷ $O(1)$
4: **end while**

Complejidad: $O(log(n))$
Justificación: Basta notar que n mod 10 ^ res = n $\iff$ n <10 ^ res $\iff$ log(n) < res. Luego, como res aumenta en 1 en cada iteración, en el peor caso el while se ejecuta log(n) veces. Por álgebra de órdenes, $O(1) + O(log(n)) = O(log(n))$.

Esta es una operación privada que recibe un natural n cualquiera, su cantidad de dígitos c >0 y devuelve en res el dígito "más a la izquierda":

iPrimerDígito(**in** n : `nat`, **in** c : `nat`) → *res* : nat

1: $res \leftarrow n\ div\ 10\hat{}(c-1)$ ▷ $O(1)$

Complejidad: $O(1)$

Esta es una operación privada que recibe un natural n cualquiera, su cantidad de dígitos c >0, y devuelve otro natural res que es el resultado de ponerle en cero el primer dígito al pasado como parámetro:

iSinPrimerDígito(**in** n : `nat`, **in** c : `nat`) → *res* : nat

1: $res \leftarrow n\ mod\ 10\hat{}(c-1)$ ▷ $O(1)$

Complejidad: $O(1)$

iDefinir(**in/out** d : `estr`, **in** k : `nat`, **in** s : σ)

1: $nat\ i \leftarrow Dígitos(k)$ ▷ $O(log(k))$
2: $puntero(nodo)\ actual \leftarrow estr.raíz$ ▷ $O(1)$
3: **while** $i > 0$ **do** ▷ $[*]$
4: **if** $actual \rightarrow hijos[PrimerDigito(k,i)] = NULL$ ▷ $O(1)$
5: **then** $varAE : arreglo\_estático[10]\ de\ puntero(nodo)$ ▷ $O(1)$
6: **for** int j = 0; j <10; j++ **do** ▷ $10 \times O(1) = O(1)$
7: $AE[j] \leftarrow NULL$ ▷ $O(1)$
8: **end for**
9: $actual \rightarrow hijos[PrimerDígito(k,\ i)] \leftarrow \&< AE,\ NULL >$ ▷ $O(1)$
10: $actual \leftarrow actual \rightarrow hijos[PrimerDígito(k,\ i)]$ ▷ $O(1)$
11: $k \leftarrow sinPrimerDígitp(k,\ i)$ ▷ $O(1)$
12: $i--$ ▷ $O(1)$
13: **end while**
14: $actual \rightarrow valor \leftarrow \&s$ ▷ $O(1)$

Complejidad: $O(log(k))$ $[*]$
Justificación: El while se ejecuta log(k) veces en el peor caso, por el mismo argumento de la complejidad de Dígitos $O(log(k)) + O(1) + O(log(k)) + O(1) = O(log(k))$.

iDefinido?(**in** d : `estr`, **in** k : `nat`,) $\rightarrow$ res : bool

1: $nat\ i \leftarrow Dígitos(k)$ ▷ $O(log(k))$
2: $puntero(nodo)\ actual \leftarrow estr.raíz$ ▷ $O(1)$
3: **while** $i > 0$ && $actual \rightarrow hijos[PrimerDigito(k,i)] \neq NULL$ **do** ▷ $O(log(k))$
4: $actual \leftarrow actual \rightarrow hijos[PrimerDígito(k,\ i)]$ ▷ $O(1)$
5: $k \leftarrow sinPrimerDígitp(k,\ i)$ ▷ $O(1)$
6: $i--$ ▷ $O(1)$
7: **end while**
8: $res \leftarrow (i = 0$ && $actual \rightarrow valor \neq NULL)$ ▷ $O(1)$

Complejidad: $O(log(k))$
Justificación: Estructura similar a la anterior.

---

iSignificado?(**in** d : `estr`, **in** k : `nat`,) $\rightarrow$ res : σ

1: $nat\ i \leftarrow Dígitos(k)$ ▷ $O(log(k))$
2: $puntero(nodo)\ actual \leftarrow estr.raíz$ ▷ $O(1)$
3: **while** $i > 0$ **do** ▷ $O(log(k))$
4: $actual \leftarrow actual \rightarrow hijos[PrimerDígito(k,\ i)]$ ▷ $O(1)$
5: $k \leftarrow sinPrimerDígitp(k,\ i)$ ▷ $O(1)$
6: $i--$ ▷ $O(1)$
7: **end while**
8: $res \leftarrow *actual \rightarrow valor$ ▷ $O(1)$

Complejidad: $O(log(k))$
Justificación: Estructura similar a la anterior.