Algoritmos y Estructuras de Datos II

Trabajo Práctico 2

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

Lollapatuza (diseño)

Grupo *Tere*

Integrante	LU	Correo electrónico
Tercic, Magalí	581/21	tercicmagali@gmail.com
Piñeiro, Ariel	128/22	arielalhuepd@gmail.com
Schandin, Juan	960/21	jschandin@gmail.com
Vega, Martina	596/22	martina.sandra.vega@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Comentarios

- Adjuntamos en C++ la implementación de los algoritmos del heap con los casos de test relevantes (especialmente para el heap($gasto$)) solo para asegurarse de que la implementación funciona correctamente y facilitar en cualquier caso la lectura de los algoritmos. La única diferencia es que la implementación en C++ puede ser tanto un max como min heap, dependiendo de un flag. Eliminamos esa parte del TP porque no hacía falta y complicaba la extensión del TAD Cola de prioridad.
- Nos valemos de lo comunicado por foro sobre el diccLog: que crear un iterador y hacer siguiente sobre él nos da el elemento con la menor clave.
- En algunos casos (puntualizados) nos valemos de que las operaciones de "Siguiente" sobre el iterador de un diccLog pueden devolver una referencia modificable sobre el significado (no sobre la clave).
- Interpretamos que cuando se nos pide "Complejidades de todas las funciones públicas, cuando corresponda.", si para el enunciado no hay restricción de complejidad, entonces no corresponde darla. Completamos con un "-" en esos casos y en todas las funciones auxiliares que utiliza esa operación (siempre y cuando esa función auxiliar no se utilice en otra operación para la cual sí se pida complejidad).
- Como en ningún apunte tenemos el aspecto visual de la sección de "Servicios usados", ni podemos consultar qué clase de contenidos tiene que tener y con qué grado de formalidad, entendemos que con puntualizar qué otros módulos utilizamos y con qué complejidades alcanza.

2 Módulos

2.1 Gasto

Este módulo es simplemente una extensión de una tupla de dos elementos que define una operación de comparación. De esta manera podemos aprovecharnos del heap para resolver el problema de encontrar quién fue la persona que más gastó desempatando por menor *id*; también nos permite, si esa persona es hackeada, encontrar la siguiente que más gastó manteniendo tiempo logarítmico.

2.1.1 Interfaz

parámetros formales

se explica con: TUPLA($gasto : dinero, persona : nat$)

géneros: `gasto`.

Operaciones básicas de gasto

$\bullet < \bullet$(**in** g_1 : `gasto`, **in** g_2 : `gasto`) $\rightarrow$ res : `bool`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} g1 < g2\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Compara gastos entre sí. Como los vamos a querer tener en un max heap, pero, si el gasto es el mismo, desempatar por menor id, chequeamos primero el gasto y luego el id de la persona de ser necesario.

Especificación de las operaciones utilizadas en la interfaz

TAD GASTO

géneros gasto

extiende TUPLA(DINERO, PERSONA)

usa Tupla, Persona, Dinero

otras operaciones

$\bullet < \bullet$: gasto $\times$ gasto $\longrightarrow$ bool

axiomas $\forall g_1, g_2$: gasto

$g_1 < g_2$ $\equiv$ **if** $g_1.gasto \,!= g_2.gasto$ **then** $g_1.gasto < g_2.gasto$ **else** $g_1.persona > g_2.persona$ **fi**

Fin TAD

2.1.2 Algoritmos

$\bullet < \bullet$(**in** g_1 : `gasto`, **in** g_2 : `gasto`) $\rightarrow$ *res* : bool

if $g_1.gasto \,!=\, g_2.gasto$ **then**
  return $g_1.gasto < g_2.gasto$
end if
return $g_1.persona > g_2.persona$

Complejidad: $\mathcal{O}(1)$
Justificación: Son comparaciones entre naturales.

2.2 Heap

2.2.1 Representación

`heap`(α) **se representa con** `estr`

donde `estr` es `tupla`(*raíz*: `nodo*`, *ultimo*: `nat`, *tamaño*: `nat`, *mapping*: `diccLog`(α, $\langle$ *puntero* : `nodo*`, *freq* : *nat*$\rangle$))

donde `nodo` es `tupla`(*izq*: `nodo*`, *der*: `nodo*`, *padre*: `nodo*`, *val*: α*, *it*: `itDicc`(α, $\langle$ *puntero* : `nodo*`, *freq* : *nat*$\rangle$))

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ (
  // La cantidad de nodos es consistente.
  (e.ultimo = #claves($e.mapping$) = #conjNodosEnAB($e.raiz$) = #nodos($e.raiz$)) $\wedge_{\text{L}}$

  // La frecuencia de cada nodo es consistente.
  (e.tamaño = sumFreq(significados($e.mapping$)) $\wedge_{\text{L}}$

  // Los nodos en el árbol y en el dicc son los mismos.
  (conjNodosEnAB($e.raiz$) = conjNodosEnMapping(significados($e.mapping$)) $\wedge_{\text{L}}$

  // Cumple con el invariante de max heap.
  (esIzquierdista($e.raiz$, 0, #nodos($e.raiz$)) $\wedge$ esMaxHeap($e.raiz$)) $\wedge_{\text{L}}$

  // Los punteros del mapping están bien apuntados a los nodos.
  ($\forall a : \alpha$)(def?(a, $e.mapping$) $\Rightarrow_{\text{L}}$ obtener(a, $e.mapping$).puntero$\rightarrow$val = a) $\wedge_{\text{L}}$

  // Los iteradores están bien apuntados al mapping.
  ($\forall n : nodo*$)(n $\in$ conjNodosEnAB($e.raiz$) $\Rightarrow_{\text{L}}$ (
    $Siguiente(n \rightarrow it)_1$ = n$\rightarrow$val $\wedge$
    $Siguiente(n \rightarrow it)_2$ = obtener(n$\rightarrow$val, $e.mapping$)
) $\wedge_{\text{L}}$

  // Si hay raíz, no tiene padre
  (e.raiz = $NULL$ $\vee_{\text{L}}$ e.raiz$\rightarrow$padre = $NULL$)
)

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

Abs(e) $\equiv$ construirColaPrior(construirSecu($e.raiz$))

conjNodosEnAB : nodo* $\longrightarrow$ conj(nodo*)

conjNodosEnAB(n) $\equiv$ **if** $n = NULL$ **then**
  $\emptyset$
else
  {n} $\bigcup$ conjNodosEnAB($n \rightarrow izq$) $\bigcup$ conjNodosEnAB($n \rightarrow der$)
fi

conjNodosEnMapping : conj(⟨ *puntero*: nodo*, *freq*: nat ⟩) $\longrightarrow$ conj(nodo*)

conjNodosEnMapping(ct) $\equiv$ **if** $\emptyset?(ct)$ **then**
$\emptyset$
else
Ag(dameUno(ct).puntero, conjNodosEnMapping(sinUno(ct)))
fi

#nodos : nodo* $\longrightarrow$ nat

#nodos(n) $\equiv$ **if** $n = NULL$ **then** 0 **else** 1 + #nodos($n \rightarrow izq$) + #nodos($n \rightarrow der$) **fi**

sumFreq : conj(⟨ *puntero*: nodo*, *freq*: nat ⟩) $\longrightarrow$ nat

sumFreq(ct) $\equiv$ **if** $\emptyset?(ct)$ **then** 0 **else** dameUno(ct).freq + sumFreq(sinUno(ct)) **fi**

esMaxHeap : nodo* $\longrightarrow$ bool

esMaxHeap(n) $\equiv$ **if** n = $NULL$ **then**
$true$
else
$(n \rightarrow izq = NULL \vee_{\text{L}} (n \rightarrow val > n \rightarrow izq \rightarrow val \wedge n \rightarrow izq \rightarrow padre = n)) \wedge$
$(n \rightarrow der = NULL \vee_{\text{L}} (n \rightarrow val > n \rightarrow der \rightarrow val \wedge n \rightarrow der \rightarrow padre = n)) \wedge$
esMaxHeap($n \rightarrow izq$) $\wedge$ esMaxHeap($n \rightarrow der$)
fi

esIzquierdista : nodo* $\times$ nat $\times$ nat $\longrightarrow$ bool

esIzquierdista(n, i, c) $\equiv$ **if** $n = NULL$ **then**
$true$
else
if $i \geq c$ **then**
$false$
else
esIzquierdista($n \rightarrow izq$, $2 \times i + 1$, c) $\wedge$ esIzquierdista($n \rightarrow der$, $2 \times i + 2$, c)
fi
fi

construirSecu : nodo* $\longrightarrow$ secu(α)

construirSecu(n) $\equiv$ **if** $n = NULL$ **then**
$\langle\rangle$
else
n→val • (construirSecu($n \rightarrow izq$) ++ construirSecu($n \rightarrow der$))
fi

construirColaPrior : secu(α) $\longrightarrow$ colaPrior(α)

construirColaPrior(s) $\equiv$ **if** $vacia?(s)$ **then** vacía **else** encolar(prim(s), construirColaPrior(fin(s))) **fi**

2.2.2 Interfaz

parámetros formales

géneros α

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

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

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

se explica con: COLADEPRIORIDAD(α)

géneros: heap(α).

Operaciones básicas de heap

VACÍO() $\rightarrow$ *res* : heap
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} vacia\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un heap vacío.

ENCOLAR(**in/out** h: heap, **in** a: α)
Pre $\equiv \{h_0 =_{\text{obs}} h\}$
Post $\equiv \{h =_{\text{obs}} encolar(a, h_0)\}$
Complejidad: $\mathcal{O}(log(U) + copy(a))$
Descripción: Agrega un elemento al heap.
Aliasing: El elemento a se agrega por copia.

DESENCOLAR(**in/out** h: heap)
Pre $\equiv \{h_0 =_{\text{obs}} h \wedge \neg vacia?(h)\}$
Post $\equiv \{h =_{\text{obs}} desencolar(h_0)\}$
Complejidad: $\mathcal{O}(log(U))$
Descripción: Remueve el máximo del heap.

PRÓXIMO(**in** h: heap) $\rightarrow$ *res* : (α)
Pre $\equiv \{\neg vacia?(h)\}$
Post $\equiv \{res =_{\text{obs}} proximo(h)\}$
Complejidad: $\mathcal{O}(copy(\alpha))$
Descripción: Devuelve el próximo elemento del heap.
Aliasing: *res* es una copia.

TAMAÑO(**in** h: heap) $\rightarrow$ *res* : *nat*
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} long(h)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de elementos en el heap.

PERTENECE(**in** h: heap, **in** a: α) $\rightarrow$ *res* : bool
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} esta?(a, h)\}$
Complejidad: $\mathcal{O}(log(U))$
Descripción: Se fija si un elemento está en el heap.

BORRAR(**in/out** h: heap, **in** a: α)
Pre $\equiv \{h_0 =_{\text{obs}} h \wedge \neg esta?(a, h_0)\}$
Post $\equiv \{h =_{\text{obs}} borrar(a, h_0)\}$
Complejidad: $\mathcal{O}(log(U))$
Descripción: Borra el elemento pasado como parámetro.

Operaciones auxiliares de heap

INSERTAR(**in/out** h: heap, **in/out** n: puntero(*nodo*))
Pre $\equiv \{h =_{\text{obs}} h_0 \wedge n =_{\text{obs}} n_0\}$
Post $\equiv$ {el nodo n se insertó en la primera posición libre manteniendo el izquierdismo y actualizando los punteros correspondientes.}
Complejidad: $\mathcal{O}(log(U))$
Descripción: Es una operación no exportada cuya única función es simplificar la lectura. Al final de esta operación no vale el invariante, debe entenderse como un reemplazo puramente sintáctico. Inserta un nuevo nodo en la primer posición libre respetando el izquierdismo.

BUSCAR(**in** n: puntero(*nodo*), **in** *camino*: nat) $\rightarrow$ *res* : puntero(*nodo*)
Pre $\equiv \{true\}$

Post $\equiv$ {*res* es el último nodo no $NULL$ del camino.}
Complejidad: $\mathcal{O}(log(U))$
Descripción: Dado un número natural, lo descomponemos en binario y recorremos el árbol en la manera en que indica la representación binaria del número: hacia la izquierda si el dígito es 0, o hacia la derecha si es 1. Si nos tenemos que mover hacia la izquierda o la derecha pero llegamos a un $NULL$, entonces devolvemos el último nodo que no es $NULL$.

SiftUp(**in/out** h: `heap`, **in/out** n: `puntero(`*nodo*`)`)
Pre $\equiv \{h =_{\text{obs}} h_0 \wedge esta?(n, h_0)\}$
Post $\equiv$ {Rep(h) $\wedge$ mismosElementos(h, h_0)}
Complejidad: $\mathcal{O}(log(U))$
Descripción: Es una operación no exportada cuya única función es simplificar la lectura. Al comienzo de esta operación no vale el invariante, debe entenderse como un reemplazo puramente sintáctico. Luego de la inserción de un nodo, realiza el sift up hasta acomodarlo y que vuelva a valer el invariante de representación.

SiftDown(**in/out** h: `heap`)
Pre $\equiv \{h =_{\text{obs}} h_0\}$
Post $\equiv$ {Rep(h) $\wedge$ mismosElementos(h, h_0)}
Complejidad: $\mathcal{O}(log(U))$
Descripción: Es una operación no exportada cuya única función es simplificar la lectura. Al comienzo de esta operación no vale el invariante, debe entenderse como un reemplazo puramente sintáctico. Luego de que se removió el *proximo* nodo, realiza el sift down para acomodar la nueva raíz a donde corresponda y que vuelva a valer el invariante de representación.

Swap(**in/out** *padre*: `puntero(`*nodo*`)`, **in/out** *hijo*: `puntero(`*nodo*`)`)
Pre $\equiv \{padre =_{\text{obs}} p_0 \wedge hijo =_{\text{obs}} h_0\}$
Post $\equiv$ {Intercambia los valores y los iteradores entre nodos y actualiza el mapping para que cada clave apunte efectivamente al nodo en el que se encuentra su valor.}
Complejidad: $\mathcal{O}(1)$
Descripción: Intercambia los valores y los iteradores de dos nodos entre sí, manteniendo el invariante de que cada clave de *mapping* apunta al nodo que efectivamente contiene como valor dicha clave. Como diccLog implementa un iterador bidireccional (modificable), al hacer 'Siguiente' tenemos el dato en la misma dirección de memoria, con lo cual al modificarlo estamos modificando también a dónde apunta el iterador, que es lo que queremos, ya que al cambiar la clave de nodo queremos actualizar el mapping para que ahora la clave apunte al nuevo nodo; y esto lo queremos hacer en tiempo constante.

Especificación de las operaciones utilizadas en la interfaz

TAD Cola de Prioridad Extendida(α)

extiende Cola de Prioridad(α)

otras operaciones

long : colaPrior(α) $\longrightarrow$ nat

está? : $\alpha \times$ colaPrior(α) $\longrightarrow$ bool

borrar : $\alpha\ a \times$ colaPrior(α) c $\longrightarrow$ colaPrior(α) {está?(a, c)}

axiomas

long(c) $\equiv$ **if** vacía?(c) **then** 0 **else** 1 + long(desencolar(c)) **fi**

está?(a, c) $\equiv$ **if** vacía?(c) **then** false **else** próximo(c) = a $\vee$ está?(a, desencolar(c)) **fi**

borrar(a_2, encolar(a_1, c)) $\equiv$ **if** $a_1 = a_2$ **then** c **else** encolar(a_1, borrar(a_2, c)) **fi**

Fin TAD

2.2.3 Algoritmos

- Sea U el máximo entre 1 y la cantidad de nodos que tiene el heap.

---

iVacío() $\rightarrow$ *res* : estr

---

raíz $\leftarrow$ NULL ▷ $\mathcal{O}(1)$
ultimo $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
tamaño $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
mapping $\leftarrow$ vacío() ▷ diccLog modificable. $\mathcal{O}(1)$
return tupla(raiz, ultimo, tamaño, mapping)

Complejidad: $\mathcal{O}(1)$
Justificación: Es la suma de operaciones constantes

---

iEncolar(**in/out** h : `estr`, **in** a : α)

---

if Definido?($h.mapping$, a) **then** ▷ $\mathcal{O}(log(U))$
  def $\leftarrow$ Significado($h.mapping$, a) ▷ Referencia modificable. $\mathcal{O}(log(U))$
  def.freq $\leftarrow$ def.freq + 1 ▷ $\mathcal{O}(1)$
else
  // El iterador es basura, se pisa inmediatamente
  itBasura $\leftarrow$ CrearIt(vacío())
  $nodo*$ nuevo $\leftarrow$ &(tupla($NULL$, $NULL$, $NULL$, a, $itBasura$)) ▷ $\mathcal{O}$(copy(α))
  nuevo$\rightarrow$it $\leftarrow$ Definir($h.mapping$, val, $\langle$nuevo, 1$\rangle$) ▷ $\mathcal{O}(log(U))$
  itBasura $\leftarrow$ NULL
  Insertar(h, $nuevo$) ▷ $\mathcal{O}(log(U))$
  SiftUp(h, $nuevo$) ▷ $\mathcal{O}(log(U))$
end if
h.tamaño $\leftarrow$ h.tamaño + 1 ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(log(U) + copy(\alpha))$
Justificación: Es la suma de operaciones que tardan $log(U)$ más lo que tarda en copiarse el valor de tipo α. Si es un tipo primitivo, es constante.

---

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

```
def ← Significado(h.mapping, Proximo(h))          ▷ Referencia modificable. O(log(U))
if def.freq > 1 then                                                          ▷ O(1)
    // Se insertó más de una vez este valor, así que reducimos su frecuencia y listo.
    def.freq ← def.freq - 1                                                   ▷ O(1)
    h.tamaño ← h.tamaño - 1                                                   ▷ O(1)
    return
end if

nodo* ultimo ← buscar(h.raiz, h.ultimo)
if raiz = ultimo then
    EliminarSiguiente(raiz→it)                                                ▷ O(log(U))
    raiz→val ← NULL                                   ▷ Liberamos memoria. O(1)
    raiz ← NULL                                       ▷ Liberamos memoria. O(1)
else
    Swap(raiz, ultimo)                                                        ▷ O(1)

    if ultimo→padre→izq = ultimo then                                         ▷ O(1)
        ultimo→padre→izq ← NULL                                               ▷ O(1)
    else
        ultimo→padre→der ← NULL                                               ▷ O(1)
    end if
    EliminarSiguiente(ult→it)                                                 ▷ O(log(U))
    ultimo→val ← NULL                                 ▷ Liberamos memoria. O(1)
    ultimo ← NULL                                     ▷ Liberamos memoria. O(1)
    SiftDown(h)                                                               ▷ O(log(U))
end if
h.ultimo ← h.ultimo - 1                                                       ▷ O(1)
```

Complejidad: $\mathcal{O}(log(U))$
Justificación: Es la suma de operaciones constantes y otras que tardan $log(U)$.

---

iProximo(**in** h : `estr`) → res : α

```
res ← *(h.raiz→val)                                                           ▷ O(copy(α))
return res
```

Complejidad: $\mathcal{O}(copy(\alpha))$
Justificación: Es simplemente desreferenciar el nodo y devolver una copia de su valor.

---

iTamaño(**in** h : `estr`) → res : nat

```
return h.tamaño
```

Complejidad: $\mathcal{O}(1)$
Justificación: Es simplemente devolver un elemento de la estructura.

---

iPertenece(**in** h : `estr`, **in** a : α) → res : $bool$

```
return Definido?(h.mapping, a)
```

Complejidad: $\mathcal{O}(log(U))$
Justificación: Es simplemente chequear en un diccionario logarítmico si la clave está definida.

---

iBorrar(**in/out** h: `estr`, **in** a: α)

```
def ← Significado(h.mapping, a)                    ▷ Referencia modificable. O(log(U))
if def.freq > 1 then                                                          ▷ O(1)
    // Se insertó más de una vez este valor, así que reducimos su frecuencia y listo.
    def.freq ← def.freq - 1                                                   ▷ O(1)
    h.tamaño ← h.tamaño - 1                                                   ▷ O(1)
    return
end if
nodo* n ← def.puntero                                                         ▷ O(1)

// Lo mandamos a la raíz y lo desencolamos
while n→padre != NULL do                                                      ▷ O(1)
    Swap(n→padre, n)                                                          ▷ O(1)
    n ← n→padre                                                               ▷ O(1)
end while
Desencolar()                                                                  ▷ O(log(U))
```

Complejidad: $\mathcal{O}(log(U))$
Justificación: Es la suma de operaciones constantes o que tardan $log(U)$ más un ciclo que, en el peor caso, corre $log(U)$ veces.

2.2.4 Algoritmos auxiliares

iInsertar(**in/out** h: `estr`, **in/out** n: `puntero(`$nodo$`)`)

```
h.ultimo ← h.ultimo + 1                                                       ▷ O(1)

// Nos fijamos si vamos a insertar la raíz
if h.ultimo = 1 then                                                          ▷ O(1)
    h.raiz ← n                                                                ▷ O(1)
    return
end if

// Si no es la raíz, buscamos al padre recorriendo el camino desde la raíz hasta la posición en la que iría el nodo,
no incluyendo ninguna de las dos.
nodo* padre ← buscar(h.raiz, h.ultimo)                                        ▷ O(log(U))
n→padre ← padre                                                               ▷ O(1)

// Insertamos al hijo en la posición que corresponda
if padre→izq = NULL then                                                      ▷ O(1)
    padre→izq ← n                                                             ▷ O(1)
else
    padre→der ← n                                                             ▷ O(1)
end if
```

Complejidad: $\mathcal{O}(log(U))$
Justificación: Es la suma de operaciones constantes, una operación que tarda $\mathcal{O}(log(U))$ y un ciclo que corre $log(U)$ veces.

iBuscar(**in** n : `puntero(`*nodo*`)`, **in** *camino*: *nat*) $\rightarrow$ *res* : puntero(*nodo*)

```
if camino ≤ 1 then                                                          ▷ O(1)
    return n
end if
nodo* prox ← Buscar(n, camino/2)                                            ▷ División entera: O(1)
if camino % 2 = 1 then
    if prox→der != NULL then                                                ▷ O(1)
        return prox→der
    end if
    return prox
end if
if prox→izq != NULL then                                                    ▷ O(1)
    return prox→izq
end if
return prox
```

Complejidad: $\mathcal{O}(log(U))$
Justificación: Cada llamada sucesiva al procedimiento reducimos el camino a la mitad. En el peor caso, estamos siguiendo el camino al último elemento, luego, recorreríamos la altura del árbol, con lo cual nos queda $\mathcal{O}(log(U))$.

---

iSiftUp(**in/out** h: `estr`, **in/out** n: `puntero(`*nodo*`)`)

```
nodo* actual ← n                                                            ▷ O(1)
while (actual→padre != NULL) && *(actual→val) > *(actual→padre→val) do      ▷ O(1)
    Swap(actual→padre, actual)                                              ▷ O(1)
    actual ← actual→padre                                                   ▷ O(1)
end while
```

Complejidad: $\mathcal{O}(log(U))$
Justificación: El ciclo corre como máximo $log(U)$ veces, ya que el nodo se insertó en la siguiente posición libre y como mucho puede subir hasta la raíz.

---

iSiftDown(**in/out** h: `estr`)

```
nodo* actual ← h.raiz                                                       ▷ O(1)
while (actual→izq != NULL) ||(actual→der != NULL) do                        ▷ O(1)
    nodo* mayor ← actual→izq                                                ▷ O(1)
    if (actual→der) != NULL && *(actual→der→val) > *(actual→izq→val) then    ▷ O(1)
        mayor ← actual→der                                                  ▷ O(1)
    end if
    if *(mayor→val) > *(actual→val)) then                                   ▷ O(1)
        Swap(actual, mayor)                                                 ▷ O(1)
        actual ← mayor                                                      ▷ O(1)
    else
        break
    end if
end while
```

Complejidad: $\mathcal{O}(log(U))$
Justificación: El ciclo corre como máximo $log(U)$ veces ya que en el peor caso estamos haciendo sift down del menor valor de todo el heap, esto implica recorrer la altura del árbol, es decir, $\mathcal{O}(log(U))$.

---

iSwap(**in/out** *padre*: `puntero(`*nodo*`)`, **in/out** *hijo*: `puntero(`*nodo*`)`)

```
// Nos guardamos temporalmente el mapping, el valor y el iterador del padre.
node_tmp ← Siguient(padre→it).second.first          ▷ Almacena el puntero: O(1)
it_tmp ← padre→it                                   ▷ O(1)
val_tmp ← padre→val                                 ▷ Almacena el puntero: O(1)

// Actualizamos el nodo del mapping.
Siguiente(padre→it).second.first ← Siguiente(hijo→it).second.first   ▷ O(1)
Siguiente(hijo→it).second.first ← node_tmp          ▷ O(1)

// Swap valores
padre→val ← hijo→val                                ▷ O(1)
hijo→val ← val_tmp                                  ▷ O(1)

// Swap iteradores
padre→it ← hijo→it                                  ▷ O(1)
hijo→it ← it_tmp                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificación: Solamente almacenamos y movemos punteros de lugar.

2.3 Lollapatuza

2.3.1 Representación

`Lollapatuza` **se representa con** `estr`

donde `estr` es `tupla`(*puestos*: `diccLog`(*puestoid*, *puesto*),
personas: `conjLineal`(*persona*),
gastosPorPersona: `diccLog`(*persona*, *dinero*),
personaQueMásGastó: `heap`(*gasto*),
puestosHackeablesPorPersona: `diccLog`(
persona, `diccLog`(*item*, `diccLog`(*puestoid*, $\langle p : puesto, hacks : nat \rangle$))
))

Rep : estre $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ 1. $(\forall a$: persona)(def?(a,$e.gastosPorPersona$) $\Rightarrow$ def?(a,$e.personas$))
$\wedge$
2. $(\forall g$: gasto)(esta?(g,e,$personaQueMasGasto$) $\Leftrightarrow$ (def?($g.persona$,$e.gastosPorPersona$)
$\wedge_{\text{L}}$ $g.gasto$ = obtener($g.persona$,$e.gastosPorPersona$)))
$\wedge$
3. $(\forall a$: persona)(def?(a,$e.puestosHackeablesPorPersona$) $\Rightarrow_{\text{L}}$ def?(a,$e.gastosPorPersona$) $\wedge$
$(\forall i$: item)(def?(i,obtener(a,e.puestosHackeablesPorPersona)) $\Rightarrow_{\text{L}}$
$(\forall p$: puestoid)(def?(p,obtener(i,obtener(a,$e.puestosHackeablesPorPersona$))) $\Rightarrow_{\text{L}}$
def?(p,e,$puestos$) $\wedge_{\text{L}}$ $i \in$ menu(obtener(p,$e.puestos$)) $\wedge$
obtener(p,$e.puestos$) = obtener(p,obtener(i,obtener(a,$e.puestosHackeablesPorPersona$))).$p$ $\wedge$
ConsumioSinPromo?(obtener(p,$e.puestos$),a,i))))

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

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

2.3.2 Interfaz

se explica con: ColaDePrioridad(α) , Lollapatuza , Gasto

genero: lolla

Operaciones básicas de lolla

NuevoSistema(**in** ps: `dicc`(*puestoid*,*puesto*) , **in** as: `conjLineal(persona)`) $\rightarrow$ res : `lolla`
Pre $\equiv$ { vendenAlMismoPrecio(significados(ps)) $\wedge$ NoVendieronAun(significados(ps)) $\wedge$ $\neg\emptyset$ (as) $\wedge$ $\neg\emptyset$ (claves(ps))}
Post $\equiv$ { $res =_{\text{obs}} crearLolla(ps, as)$ }
Complejidad: -
Descripción: Crea un nuevo sistema. En particular, define las estructuras vacías necesarias y transforma las dadas para adecuarlas a la estructura que elegimos.

RegistrarCompra(**in/out** l: `lolla` , **in** p: `puestoid` , **in** a: `persona` , **in** i: `item` , **in** c: `cant`)
Pre $\equiv$ { $l =_{\text{obs}} l_0 \wedge def?(p.puestos(l)) \wedge_l haySuficiente?(obtener(p, puestos(l)), i, c)$ }
Post $\equiv$ { $l =_{\text{obs}} vender(l_0, p, a, i, c)$ }
Complejidad: $\mathcal{O}(log(A) + log(P) + log(I))$
Descripción: Registra la compra en el puesto correspondiente, se fija si la compra es hackeable y, en caso de serlo, registra la compra como hackeable para esa persona, item y puesto. Actualiza también el gasto total de esa persona y, finalmente, mantiene el max heap con los gastos de cada persona.

HackearSistema(**in/out** l: `lolla` ,**in** a: `persona` ,**in** i: `item`)
Pre $\equiv$ { $l =_{\text{obs}} l_0 \wedge consumioSinPromoEnAlgunPuesto(l, a, i)$ }

Post $\equiv$ { $l =_{\text{obs}} hackear(l_0, a, i)$ }
Complejidad: $\mathcal{O}(log(A) + log(I) \mid log(A) + log(I) + log(P))$
Descripción: Encuentra el puesto a hackear para esa persona y ese item y, valga la redundancia, lo hackea. Se encarga también de mantener el diccLog y el max heap con los gastos para cada persona.

GastoTotalPorPersona(**in** l: `lolla,in` a : $persona$) $\rightarrow$ res : `nat`
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}} gastoTotal(l, a)$}
Complejidad: $\mathcal{O}(log(A))$
Descripción: Dada una persona que puede o no estar en el festival devolvemos cuánto gastó. En particular, si no está en el festival el gasto es 0.

PersonaQueMasGasto(**in** l: $lolla$) $\rightarrow$ res : `persona`
Pre $\equiv$ { $\neg vacio?(personas(l)) \wedge (\exists a$: persona)(a $\in$ persona(l) $\wedge_{\text{L}}$ gastoTotal(l,a) > 0) }
Post $\equiv$ { $res =_{\text{obs}} masGasto(l)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la persona que más gastó consultando al próximo elemento del max heap.

PuestoConMenorStock(**in** l: `lolla` , **in** i: `item`) $\rightarrow$ res : `puestoid`
Pre $\equiv$ {$true$}
Post $\equiv$ { $res =_{\text{obs}} menorStock(l, i)$ }
Complejidad: -
Descripción: Devuelve el *puestoid* con menor stock para un item i. Si ningún puesto tenía ese ítem, simplemente devolvemos el puesto de menor id.

PersonasEnSistema(**in** l: `lolla`) $\rightarrow$ res : `conjLineal(persona)`
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} personas(l)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de personas en el sistema.
Aliasing: Res es una referencia no modificable

PuestosDeComidaEnSistema(**in** l: `lolla`) $\rightarrow$ res : `dicc(`$puestoid$`,`$puesto$`)`
Pre $\equiv$ { $true$ }
Post $\equiv$ {$res =_{\text{obs}} puestos(l)$}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los puestos en el sistema junto con sus ids.
Aliasing: Res es una referencia no modificable

Operaciones auxiliares de lolla

ConstruirDiccLog(**in** d: `dicc(`κ`, `σ`)`) $\rightarrow$ res : `diccLog(`κ`, `σ`)`
Pre $\equiv$ { true }
Post $\equiv$ { res $=_{\text{obs}}$ d }
Complejidad: -
Descripción: Dado un diccionario, construye un diccionario logarítmico

2.3.3 Algoritmos

iNuevoSistema(**in** ps : `dicc(`$puestoid$`,` $puesto$`)`, **in** as : `conjLineal(`$persona$`)`) $\rightarrow$ res : $estr$

puestos $\leftarrow$ ConstruirDiccLog(ps)
personas $\leftarrow$ as
gastosPorPersona $\leftarrow$ Vacio()
personaQueMásGastó $\leftarrow$ Vacio()
puestosHackeablesPorPersona $\leftarrow$ Vacio()
return $\langle$ puesto, personas, gastosPorPersona, personaQueMásGastó, puestosHackeablesPorPersona $\rangle$

Complejidad: -
Justificación: No se piden restricciones a la complejidad.

---

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

res $\leftarrow$ $Proximo(e.personaQueMasGasto).persona$
return res

Complejidad: $\mathcal{O}(1)$
Justificación: Es simplemente devolver el valor máximo del heap.

iRegistrarCompra(**in/out** e: $estr$, **in** a: $persona$, **in** p: $puestoid$ **in** i: $item$, **in** c: $cant$)

```
// Registramos la compra en el puesto
puesto ← Significado(e.puestos, p)                                          ▷ O(log(P))
iRegistrarCompraEnPuesto(puesto, a, i, c)                                   ▷ O(log(A) + log(I))

// Nos fijamos y registramos si la compra es hackeable
huboDescuento ← iGetDescuento(puesto, i, c) > 0                             ▷ O(log(I))
if ¬huboDescuento then                                                      ▷ O(1)
    itemsHackeables ← Vacio()                                               ▷ O(1)
    if Definido?(e.puestosHackeablesPorPersona, a) then                     ▷ O(log(A))
        itemsHackeables ← Significado(e.puestosHackeablesPorPersona, a)     ▷ O(log(A))
    end if
    puestosHackeables ← Vacio()                  ▷ Creamos un diccLog modificable. O(1)
    if Definido?(itemsHackeables, i) then                                   ▷ O(log(I))
        puestosHackeables ← Significado(itemsHackeables, i)                 ▷ O(log(I))
    end if
    hackeosPosibles ← 0
    if Definido?(puestosHackeables, p) then                                 ▷ O(log(P))
        hackeosPosibles ← Significado(puestosHackeables, p).hacks           ▷ O(log(P))
    end if
    Definir(puestosHackeables, p, ⟨puesto, hackeosPosibles + 1⟩)            ▷ O(log(P))
    Definir(itemsHackeables, i, puestosHackeables)                          ▷ O(log(I))
    Definir(e.puestosHackeablesPorPersona, a, itemsHackeables)              ▷ O(log(A))
end if

// Procesamos el gasto por persona y mantenemos el max heap de gastos
viejoGastoTotal ← 0
if Definido?(e.gastoPorPersona, a) then                                     ▷ O(log(A))
    viejoGastoTotal ← Significado(e.gastoPorPersona, a)                     ▷ O(log(A))
end if
nuevoGastoTotal ← iCalcularTotal(puesto, i, c) + viejoGastoTotal            ▷ O(log(I))
Definir(e.gastoPorPersona, a, nuevoGastoTotal)                              ▷ O(log(A))
if Pertenece(h.personaQueMasGasto, gasto(viejoGastoTotal, a)) then          ▷ O(log(A))
    Borrar(h.personaQueMasGasto, gasto(viejoGastoTotal, a))                 ▷ O(log(A))
end if
Encolar(h.personaQueMasGasto, gasto(nuevoGastoTotal, a))                    ▷ O(log(A))
```

Complejidad: $\mathcal{O}(log(P) + log(A) + log(I))$
Justificación: Si sumamos el peor caso posible (cuando hubo descuento y tanto la persona, como el item, como el puesto están definidos en la estructura; y además la persona ya había realizado una compra), nos quedaría $\mathcal{O}(4 \times log(P) + 10 \times log(A) + 6 \times log(I) + 3) = \mathcal{O}(log(P) + log(A) + log(I))$

---

iPersonasEnSistema(**in** e: $estr$) → res : conjLineal($persona$)

```
res ← e.personas
return res
```

Complejidad: $\mathcal{O}(1)$
Justificación: Es simplemente devolver el elemento de la estructura.

iGastoTotalPorPersona(**in** e : $estr$, **in** p : $persona$) $\rightarrow$ res : nat

if Definido?($e.gastosPorPersona$, p) **then** ▷ $\mathcal{O}(log(A))$
  res $\leftarrow$ Significado(p, $e.gastosPorPersona$) ▷ $\mathcal{O}(log(A))$
else
  res $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
end if
return res

Complejidad: $\mathcal{O}(log(A))$
Justificación: En el peor caso tenemos $\mathcal{O}(log(A) + log(A)) = \mathcal{O}(log(A))$

---

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

// Buscamos el puesto.
itemsHackeables $\leftarrow$ Significado($e.puestosHackeablesPorPersona$, a) ▷ $\mathcal{O}(log(A))$
puestosHackeables $\leftarrow$ Significado($itemsHackeables$, i) ▷ $\mathcal{O}(log(I))$

// El primer elemento del iterador de un diccLog es el de menor clave.
it $\leftarrow$ CrearIt(puestosHackeables) ▷ $\mathcal{O}(1)$

// Hackeamos el puesto y modificamos la cantidad de hackeos posibles.
puestoAHackear $\leftarrow$ SiguienteSignificado(it) ▷ Referencia modificable: $\mathcal{O}(1)$
HackearPuesto($puestoAHackear.p$, a, i) ▷ $\mathcal{O}(log(A) + log(I))$
puestoAHackear.hacks $\leftarrow$ puestoAHackear.hacks - 1 ▷ Referencia modificable: $\mathcal{O}(1)$
if puestoAHackear.hacks = 0 **then**
  EliminarSiguiente(it) ▷ Mantener el invariante de un diccLog luego de borrar: $\mathcal{O}(log(P))$
end if

// Mantenemos actualizado el max heap y el diccionario de gastos por persona
viejoGastoTotal $\leftarrow$ Significado($e.gastoPorPersona$, a) ▷ $\mathcal{O}(log(A))$
nuevoGastoTotal $\leftarrow$ viejoGastoTotal - iCalcularTotal($puestoAHackear$, i, 1) ▷ $\mathcal{O}(log(I))$
Definir($e.gastoPorPersona$, a, $nuevoGastoTotal$) ▷ $\mathcal{O}(log(A))$
Borrar($h.personaQueMasGasto$, gasto($viejoGastoTotal$, a)) ▷ $\mathcal{O}(log(A))$
Encolar($h.personaQueMasGasto$, gasto($nuevoGastoTotal$, a)) ▷ $\mathcal{O}(log(A))$

Complejidad: $\mathcal{O}(log(A) + log(I) | log(A) + log(I) + log(P))$
Justificación: Es la suma de $\mathcal{O}(6 \times log(A) + 3 \times log(I))$ y eventualmente $log(P)$ si es necesario borrarlo al puesto en el caso en que no es más hackeable. Luego queda $\mathcal{O}(log(A) + log(I) | log(A) + log(I) + log(P)$

iPuestoConMenorStock(**in** l: `lolla` , **in** i: `item`) $\rightarrow$ res : *puestoid*

```
// Nos armamos un diccLog con todos los puestos que tienen ese item en el menú
puestosConItemEnMenu ← Vacío()
it ← CrearIt(l.puestos)
while HaySiguiente(it) do
    pid ← SiguienteClave(it)
    puesto ← SiguienteSignificado(it)
    if Definido?(puesto.menu, i) then
        Definir(puestosConItemEnMenu, pid, puesto)
    end if
    Avanzar(it)
end while

// Si ningún puesto tiene el ítem en el menú, devolvemos el puesto de menor id.
if #Claves(puestosConItemEnMenu) = 0 then
    return SiguienteClave(CrearIt(l.puestos))
end if

// Si llegamos acá es que hay puestos con el ítem definido en su menú, luego buscamos el mínimo y desempatamos
por menor id.
it ← CrearIt(puestosConItemEnMenu)
min ← GetStock(SiguienteSignificado(it), i)
min_pid ← SiguienteClave(it)
Avanzar(it)
while HaySiguiente(it) do
    pid ← SiguienteClave(it)
    stock ← GetStock(SiguienteSignificado(it), i)
    if stock < min ∨ (stock = min ∧ pid < min_pid) then
        min ← stock
        min_pid ← pid
    end if
    Avanzar(it)
end while
return min_pid
```

Complejidad: -
Justificación: No tiene restricciones.

2.3.4 Algoritmos auxiliares

iConstruirDiccLog(**in** d: `dicc(`κ`, `σ`)`) $\rightarrow$ res : diccLog(κ, σ)

```
res ← Vacío()
it ← CreateIt(d)
while HaySiguiente(it) do
    sig ← Siguiente(it)
    Definir(sig_1, sig_2, res)
    Avanzar(it)
end while
return res
```

Complejidad: -
Justificación: No hay ninguna restricción.

2.3.5 Servicios usados

- Utiliza el módulo "Puesto".

- La operación de registrar compra en "Puesto" debe ser $\mathcal{O}(log(A) + log(I))$.
- La operación de hackear un "Puesto" debe ser $\mathcal{O}(log(A) + log(I))$.
- La operación de calcular el total de una venta debe ser $\mathcal{O}(log(I))$.
- Utiliza el módulo "Heap".
- Las operaciones de "Heap": encolar, desencolar, borrar y pertenece deben ser $\mathcal{O}(log(n))$.
- La operación de "Heap" de "Proximo" debe ser $\mathcal{O}(1)$

2.4 Puesto de comida

2.4.1 Representación

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

donde `estr` es `tupla`(*menu*: `diccLog`(*item*, *nat*),
stock: `diccLog`(*item*, *nat*),
promos: `diccLog`(*item*, `vector`(*nat*)),
ventas: `diccLog`(*persona*, `lista`($\langle$*item*, *cantidad*$\rangle$)),
ventasHackeables: `diccLog`(*persona*, `diccLog`(*item*, `lista`(*itLista*))),
gastosPorPersonaEnPuesto: `diccLog`(*persona*, *dinero*))

Rep : estre $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ 1. $(\forall i$: item)(def?$(i, e.stock) \Leftrightarrow$ def?$(i, e.menu)$)

$\wedge$

2. $(\forall i$: item)(def?$(, i, e.promos) \Rightarrow_{\text{L}}$ def?$(i, e.menu)$)

$\wedge$

3. $(\forall a$: persona)(def?$(a, e.ventas) \Leftrightarrow$ def?$(a, e.gastosPorPersonaEnPuesto)$
$\wedge$ (def?$(a, e.ventas) \Rightarrow_{\text{L}}$
$(\forall t$: tupla($item$,$cantidad$)(esta?(t,obtener(a,$e.ventas$)) $\Rightarrow_{\text{L}}$ def?($t.item$,$e.menu$) $\wedge$
totalVentas(obtener(a,$e.ventas$),$e.menu$,$e.promos$) = obtener(a,$e.gastosPorPersonaEnPuesto$)))))

$\wedge$

4. $(\forall a$: persona)$(\forall i$: item)(def?(a,$e.ventasHackeables$) $\wedge_{\text{L}}$ def?(i,obtener(a,$e.ventasHackeables$)
$\Rightarrow_{\text{L}}$ def?(i,$e.menu$) $\wedge$
$(\forall it$: itLista)(esta?(it,obtener(i,obtener(a,$e.ventasHackeables$))) $\Rightarrow_{\text{L}}$
esta?(Siguiente(it),obtener(a,$e.ventas$)) $\wedge$ Siguiente(it).item=i)))

$\wedge$

5. $(\forall a$: persona)(def?$(a, e.ventas) \Rightarrow_{\text{L}}$ $(\forall t$: tupla(item,cantidad))(esta?$(t, obtener(a, e.ventas))$
$\wedge$ def?$(t.item, e.menu) \wedge_{\text{L}}$ consumioSinPromo1Venta?$(e, t.item, t) \Rightarrow$
def?$(a, e.ventasHackeables) \wedge_{\text{L}}$ def?$(t.item, obtener(a, e.ventasHackeables)) \wedge_{\text{L}}$
hayIteradorAVenta($obtener(t.item, obtener(a, e.ventasHackaebles)), t$)))

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

Abs(e) $\equiv$ e $=_{\text{obs}}$ p : puesto |
menu(p) = claves($e.menu$)
$\wedge$
$(\forall i$: item)($i \in$ menu(p) $\Rightarrow_{\text{L}}$ precio(p, i) = obtener(i, $e.menu$) $\wedge$ stock(p, i) = obtener(i, $e.stock$) $\wedge$
$(\forall n$: nat)(
($n \geq$ Longitud(VectorPromos(e, i)) $\Rightarrow$
descuento(p, i, n) = VectorPromos($e.promos$, i)[Longitud(VectorPromos($e.promos$, i)) - 1])
$\wedge$
($n <$ Longitud(VectorPromos($e.promos$,i)) $\Rightarrow$
descuento(p, i, n) = VectorPromos($e.promos$,i)[n])))
$\wedge$
$(\forall a$: persona)(ventas(p, i) = ListaAMulticonjunto(obtener(a, $e.ventas$)))

VectorPromos : dicc($item, secu(nat)$) $\times$ item $\longrightarrow$ secu(nat)

VectorPromos(d,i) $\equiv$ obtener(i, d)

totalVentas : secu(tupla($item, cantidad$)) $\times$ dicc($item, nat$) $\times$ dicc($item, secu(nat)$) $\longrightarrow$ nat

totalVentas(s,d,p) $\equiv$ **if** $vacia?(s)$ **then** 0 **else** gastoPorVenta($prim(s), d, p$) + totalVentas($fin(s), d$) **fi**

gastoPorVenta : tupla($item$, $cantidad$) $\times$ dicc($item$, nat) $\times$ dicc($item$, secu(nat)) $\longrightarrow$ nat

gastoPorVenta(t, d, p) $\equiv$ **if** $def?(t.item, p)$ **then** gastoConDescuento(t, d, p) **else** obtener($t.item, d$) * $t.cant$ **fi**

gastoConDescuento : tupla($item$, $cantidad$) $\times$ dicc($item$, nat) $\times$ dicc($item$, secu(nat)) $\longrightarrow$ nat

gastoConDescuento $\equiv$ **if** $t.cant \geq Long(obtener(t.item, p))$ **then**
((100 - ult(obtener(t.item,p))) / 100) * obtener($t.item, d$) * $t.cant$
else
((100 - obtener($t.item, p$)[t.cant])/100) * obtener($t.item, d$) * $t.cant$
fi

hayIteradorAVenta(l, t) $\equiv$ ($\exists it$: itLista(tupla($item, cantidad$)))(esta?(it, l) $\wedge$ Siguiente(it) = t)

2.4.2 Interfaz

se explica con: PUESTO

genero: PUESTO

Operaciones básicas de puesto

NUEVOPUESTO(**in** m: `dicc(`$item$`, ` nat`)`, **in** s: `dicc(`$item$`, ` $cant$`)`, **in** d: `dicc(`$item$`, dicc(`$cant$`, ` nat`))`) $\rightarrow$ res : `puesto`
Pre $\equiv$ {$claves$(m) $=_{\text{obs}}$ $claves$(s) $\wedge$ $claves$(d) $\subseteq$ $claves$(m)}
Post $\equiv$ {$res =_{\text{obs}} crearPuesto(m, s, d)$}
Complejidad: -
Descripción: Crea una nueva estructura

GETSTOCK(**in** p: $puesto$, **in** i: $item$) $\rightarrow$ res : `cant`
Pre $\equiv$ {$i \in menu(p)$}
Post $\equiv$ {$res =_{\text{obs}} stock(p, i)$}
Complejidad: $\mathcal{O}(log(I))$
Descripción: Dado un puesto y un item, devuelve la cantidad de ese item en stock en el puesto.

GETDESCUENTO(**in** p: $puesto$, **in** i: $item$, **in** c: $cant$) $\rightarrow$ res : `nat`
Pre $\equiv$ {$i \in menu(p)$}
Post $\equiv$ {$res =_{\text{obs}} descuento(p, i, c)$}
Complejidad: $\mathcal{O}(log(I)) \subseteq \mathcal{O}(log(I) + log(cant))$
Descripción: Dado un puesto, un item y una cantidad, devuelve, de haberlo, el descuento correspondiente para esa cantidad del item en el puesto, o 0 de lo contrario.

GETGASTOPORPERSONAENPUESTO(**in** p: $puesto$, **in** a: $persona$) $\rightarrow$ res : `dinero`
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}} gastosDe(p, a)$}
Complejidad: $\mathcal{O}(log(A))$
Descripción: Dado un puesto y una persona, devuelve cuánto gastó esa persona en total en el puesto.

REGISTRARCOMPRAENPUESTO(**in/out** p: $puesto$, **in** a: $persona$, **in** i: $item$, **in** c: $cant$))
Pre $\equiv$ {$i \in menu(p) \wedge_{\text{L}} haySuficiente(p, i, c) \wedge p =_{\text{obs}} p_0$}
Post $\equiv$ {$p =_{\text{obs}} vender(p_0, a, i, c)$}
Complejidad: $\mathcal{O}(log(A) + log(I))$
Descripción: Dado un puesto, una persona, un item y una cantidad, modifica el puesto para registrar la venta; donde "registrar" implica actualizar el registro de ventas para esa persona, el gasto que realizó dicha persona en el puesto y, de haber sido una venta sin descuento, actualizar también las ventas hackeables para esa persona en el puesto.

HACKEARPUESTO(**in/out** p: $puesto$, **in** a: $persona$, **in** i: $item$)

Pre $\equiv \{i \in menu(p) \wedge_L p =_{\text{obs}} p_0 \wedge ConsumioSinPromo?(p, a, i)\}$
Post $\equiv \{p =_{\text{obs}} olvidarItem(p_0, a, i)\}$
Complejidad: $\mathcal{O}(log(A) + log(I))$
Descripción: Dado un puesto, una persona y un item, hackea una venta realizada a esa persona de ese item que no haya tenido descuento. Esto implica "olvidar el item" (ver *OlvidarItem*), actualizar el gasto de la persona y el stock del puesto.

CalcularTotal(**in** e: *estr*, **in** i: *item*, **in** c: *cant*) $\rightarrow$ *res* : `dinero`
Pre $\equiv$ { $i \in menu(e)$ }
Post $\equiv$ { Res es el total de la compra con el descuento correspondiente aplicado , si es que lo habia. }
Complejidad: $\mathcal{O}(log(I))$
Descripción: Calcula el total (con descuento) que gastó una persona al realizar una compra de una determinada cantidad de un item.

Operaciones auxiliares de puesto

OlvidarItem(**in/out** e: *estr*, **in** p: *persona*, **in** i: *item*)
Pre $\equiv \{i \in menu(e) \wedge_L e =_{\text{obs}} e_0 \wedge ConsumioSinPromo?(e, a, i)\}$
Post $\equiv$ {Se elimina la primera venta hackeable de e.ventas indicada por el iterador asociado en e.ventasHackeables para esa persona. Se borra el iterador que apunta a esa venta en e.ventasHackeables para esa persona. }
Complejidad: $\mathcal{O}(log(A) + log(I))$
Descripción: Dado un puesto, una persona y un item, disminuye la cantidad del item vendido a esa persona en 1 (o borra la venta si queda en 0).

ConstruirPromos(**in** d: `dicc(`*item*`, dicc(`*cant*`, `*nat*`))`) $\rightarrow$ *res* : `diccLog(`*item*`, vector(`*nat*`))`
Pre $\equiv$ { true }
Post $\equiv$ { Res es un diccionario donde cada clave es un item cuyo significado es un vector que en cada posicion tiene el mejor descuento para la cantidad indicada por la posicion }
Complejidad: -
Descripción: La idea es poder obtener el descuento para un ítem y una cantidad en tiempo constante. Para eso lo que hacemos es construir un vector donde cada índice representa una cantidad que apunta a cuál es el mejor descuento para esa cantidad. Si se compra más cantidad que el tamaño del vector, entonces devolvemos el último elemento, que será el mejor descuento aplicable para esa cantidad; sino, simplemente devolvemos lo que hay en el índice para esa cantidad. Esta función, entonces, toma las promociones para cada ítem y devuelve un diccionario donde cada item es una clave cuyo significado es el vector con las propiedades mencionadas anteriormente. De esta manera, buscar un descuento para una cantidad de un ítem puede hacerse en tiempo constante.

ConstruirPromosParaItem(**in** d: `dicc(`*cant*`, `*nat*`)`) $\rightarrow$ *res* : `vector(`*nat*`)`
Pre $\equiv$ { true }
Post $\equiv$ { Res es un vector que indica la mejor promocion para cada cantidad de un item representada por la posicion en tal vector }
Complejidad: -
Descripción: Dado un diccionario $\langle cant, nat \rangle$, construye el vector con las mejores promociones para cada cantidad.

MaxPromosParaItem(**in** d: `dicc(`*cant*`, `*nat*`)`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ { true }
Post $\equiv$ { Devolvemos de todas las promos definidas en el diccionario la maxima clave. }
Complejidad: -
Descripción: Dado un diccionario $\langle cant, nat \rangle$, devuelve la máxima clave.

ConstruirDiccLog(**in** d: `dicc(`κ`, `σ`)`) $\rightarrow$ *res* : `diccLog(`κ`, `σ`)`
Pre $\equiv$ { true }
Post $\equiv$ { res $=_{\text{obs}}$ d }
Complejidad: -
Descripción: Dado un diccionario, construye un diccionario logarítmico

AgregarVenta(**in/out** e: *estr*, **in** p: `persona`, **in** i: *item*, **in** c: *cant*) $\rightarrow$ *res* : `itLista`
Pre $\equiv \{e =_{\text{obs}} e_0 \wedge i \in menu(e) \wedge_{\text{L}} haySuficiente(e, i, c)$ }
Post $\equiv$ { Agrega la tupla $< item, cantidad >$ a la lista e.ventas para esa persona p ; y res es el iterador apuntando a esa venta. }

Complejidad: $\mathcal{O}(log(A))$
Descripción: Agrega una venta $\langle item, cant \rangle$ para una determinada persona en el diccionario de ventas.

AgregarVentaHackeable(**in/out** e: $estr$, **in** p: `persona`, **in** i: $item$, **in** it: $itLista$)
Pre $\equiv$ { $e =_{\text{obs}} e_0$ e it apunta a un elemento de e.ventas para la persona p, y ese elemento es una tupla cuyo item es i. Ademas , $i \in menu(e)$ }
Post $\equiv$ { it se agrega a e.ventasHackeables para esa persona p }
Complejidad: $\mathcal{O}(log(A) + log(I))$
Descripción: Agrega al diccionario de ventas hackeables un iterador que apunta a una venta sin descuento de un item para una persona en el diccionario de ventas.

Div(**in** n: $dinero$, **in** k: nat) $\rightarrow$ res : $dinero$
Pre $\equiv$ $\{true\}$
Post $\equiv$ { $res = n/k$ }
Complejidad: $\mathcal{O}(n/k)$
Descripción: Divisón entera.

AplicarDescuento(**in** p: $dinero$, **in** d: nat) $\rightarrow$ res : `dinero`
Pre $\equiv$ $\{true\}$
Post $\equiv$ { $res = p - p * (d//100)$ }
Complejidad: $\mathcal{O}(p/d)$
Descripción: Calcula el descuento.

2.4.3 Algoritmos

---

iNuevoPuesto(**in** m: `dicc(`$item$`,` nat`)`, **in** s: `dicc(`$item$`,` $cant$`)`, **in** d: `dicc(`$item$`,` `dicc(`$cant$`,` nat`))`) $\rightarrow$ res : $puesto$

---

menu $\leftarrow$ ConstruirDiccLog(m)
stock $\leftarrow$ ConstruirDiccLog(s)
promos $\leftarrow$ ConstruirPromos(d)
ventas $\leftarrow$ Vacío()
ventasHackeables $\leftarrow$ Vacío()
gastosPorPersonaEnPuesto $\leftarrow$ Vacío()
res $\leftarrow$ $\langle menu, stock, promos, ventas, ventasHackeables, gastosPorPersonaEnPuesto \rangle$
return res

Complejidad: -
Justificación: No hay ninguna restricción.

---

iGetStock(**in** e: $estr$, **in** i: $item$) $\rightarrow$ res : $cant$

---

res $\leftarrow$ Significado($e.stock$, i)
return res

Complejidad: $\mathcal{O}(log(I))$
Justificación: La complejidad de la operación $Significado$ sobre un $diccLog$ es $\mathcal{O}(log(n))$. En este caso n es la cantidad de items ya que en el peor caso el stock de este puesto tiene por todos los items del festival.

---

iGetDescuento(**in** e: $estr$, **in** i: $item$, **in** c: $cant$) $\rightarrow$ res : nat

```
if Definido?(e.promos, i) then                                          ▷ O(log(I))
    descuentos ← Significado(e.promos, i)                               ▷ O(log(I))
    if c > Longitud(descuentos) then                                    ▷ O(1)
        res ← Ultimo(descuentos)                                        ▷ O(1)
    else
        res ← descuentos[c]                                             ▷ O(1)
    end if
else
    res ← 0                                                             ▷ O(1)
end if
return res
```

Complejidad: $\mathcal{O}(log(I))$
Justificación: En primer lugar, no hay ningún ciclo. En segundo lugar, salvo $Definido$ y $Significado$, que son $\mathcal{O}(log(n))$ sobre un $diccLog$, las demás operaciones son $\mathcal{O}(1)$. Luego, en el peor caso esto es $\mathcal{O}(log(I)+log(I)+1+1) = \mathcal{O}(log(I))$.

---

iGetGastoPorPersonaEnPuesto(**in** e: $estr$, **in** p: $persona$) $\rightarrow$ res : nat

```
if Definido?(e.gastosPorPersonaEnPuesto, p) then                        ▷ O(log(A))
    res ← Significado(e.gastosPorPersonaEnPuesto, p)                    ▷ O(log(A))
else
    res ← 0                                                             ▷ O(1)
end if
return res
```

Complejidad: $\mathcal{O}(log(A))$
Justificación: En el peor caso queda $\mathcal{O}(log(A) + log(A)) = \mathcal{O}(log(A))$

---

iRegistrarCompraEnPuesto(**in/out** e: $estr$, **in** p: $persona$, **in** i: $item$, **in** c: $cant$)

```
// Agregamos la venta
it ← AgregarVenta(e, p, i, c)                                           ▷ O(log(A))

// Si es hackeable, la agregamos a ventas hackeables
if GetDescuento(e, i, c) = 0 then                                       ▷ O(log(I))
    AgregarVentaHackeable(e, p, i, it)                                  ▷ O(log(A) + log(I))
end if

// Calculamos el total y actualizamos el gasto por persona
total ← CalcularTotal(e, i, c)                                          ▷ O(log(I))
if Definido?(e.gastosPorPersonaEnPuesto, p) then                        ▷ O(log(A))
    total ← total + Significado(e.gastosPorPersonaEnPuesto, p)          ▷ O(log(A))
end if
Definir(e.gastosPorPersonaEnPuesto, p, total)                           ▷ O(log(A))

// Actualizamos el stock
stock ← Significado(e.stock, i) - c                                     ▷ O(log(I))
Definir(e.stock, i, stock)                                              ▷ O(log(I))
```

Complejidad: $\mathcal{O}(log(A) + log(I))$
Justificación: Si sumamos en el peor caso quedaría: $\mathcal{O}(5 \times log(A) + 5 \times log(I)) = \mathcal{O}(log(A) + log(I))$

iHackearPuesto(**in/out** e: $estr$, **in** p: $persona$, **in** i: $item$)

```
stock ← Significado(e.stock, i) + 1                                  ▷ O(log(I))
precio ← Significado(e.menu, i)                                      ▷ O(log(I))
gasto ← Significado(e.gastoPorPersonaEnPuesto, p) - precio           ▷ O(log(A))
Definir(e.stock, i, stock)                                           ▷ O(log(I))
Definir(e.gastoPorPersonaEnPuesto, p, gasto)                         ▷ O(log(A))
OlvidarItem(e, p, i)                                                 ▷ O(log(A) + log(I))
```

Complejidad: $\mathcal{O}(log(A) + log(I))$
Justificación: Si sumamos en el peor caso quedaría: $\mathcal{O}(3 \times log(A) + 4 \times log(I)) = \mathcal{O}(log(A) + log(I))$

iCalcularTotal(**in** e: $estr$, **in** i: $item$, **in** c: $cant$) → res : nat

```
descuento ← GetDescuento(e, i, c)                                    ▷ O(log(I))
precio ← Significado(e.menu, i)                                      ▷ O(log(I))
res ← AplicarDescuento(c × precio, descuento)                        ▷ O(1)
return res
```

Complejidad: $\mathcal{O}(log(I))$
Justificación: Queda $\mathcal{O}(log(I) + log(I) + 1) = \mathcal{O}(log(I))$.

2.4.4 Algoritmos auxiliares

iConstruirPromos(**in** d: `dicc(`$item$`, dicc(`$cant$`, `nat`))`) → res : diccLog($item$, vector(nat))

```
res ← Vacío()
it ← CrearIt(d)
while HaySiguiente(it) do
    sig ← Siguiente(it)
    Definir(sig_1, iConstruirPromosParaItem(sig_2), res)
    Avanzar(it)
end while
return res
```

Complejidad: -
Justificación: No hay ninguna restricción.

iConstruirPromosParaItem(**in** d: `dicc(`$cant$`, `nat`)`) → res : vector(nat)

```
res ← Vacío()
AgregarAtrás(res, 0)                          ▷ No nos interesa la posición 0.
max ← maxPromosParaItem(d)
i ← 1
mejorPromo ← 0
while i ≤ max do
    if Definido(i, d) then
        mejorPromo ← Significado(i, d)
    end if
    AgregarAtrás(res, mejorPromo)
end while
return res
```

Complejidad: -
Justificación: No hay ninguna restricción.

iMaxPromoParaItem(**in** d: `dicc(`$cant$`,` nat`)`) $\rightarrow$ res : nat

```
it ← CrearIt(d)
res ← 0
while HaySiguiente(it) do
    clave ← SiguienteClave(it)
    if clave > res then
        res ← clave
    end if
    Avanzar(it)
end while
return res
```

Complejidad: -
Justificación: No hay ninguna restricción.

iConstruirDiccLog(**in** d: `dicc(`κ`,` σ`)`) $\rightarrow$ res : diccLog(κ, σ)

```
res ← Vacío()
it ← CreateIt(d)
while HaySiguiente(it) do
    sig ← Siguiente(it)
    Definir(sig_1, sig_2, res)
    Avanzar(it)
end while
return res
```

Complejidad: -
Justificación: No hay ninguna restricción.

iAgregarVenta(**in/out** e: $estr$, **in** p: `persona`, **in** i: $item$, **in** c: $cant$) $\rightarrow$ res : itLista

```
venta ← ⟨i, c⟩                                          ▷ O(1)
if ¬Definido?(e.ventas, p) then                          ▷ O(log(A))
    Definir(e.ventas, p, Vacía())                        ▷ O(log(A))
end if
listaDeVentas ← Significado(e.ventas, p)                 ▷ O(log(A))
it ← AgregarAtrás(listaDeVentas, venta)                  ▷ O(1)
return it
```

Complejidad: $\mathcal{O}(log(A))$
Justificación: Queda $\mathcal{O}(3 \times log(A) + 2) = \mathcal{O}(log(A))$.

iAgregarVentaHackeable(**in/out** e: *estr*, **in** p: `persona`, **in** i: *item*, **in** it: *itLista*)

```
venta ← ⟨i, c⟩                                                    ▷ O(1)
if ¬Definido?(e.ventasHackeables, p) then                         ▷ O(log(A))
    Definir(e.ventasHackeables, p, Vacío())                       ▷ O(log(A))
end if
items ← Significado(e.ventasHackeables, p)                        ▷ O(log(A))
if ¬Definido?(items, i) then                                      ▷ O(log(I))
    Definir(items, i, Vacía())                                    ▷ O(log(I))
end if
itLista ← Significado(items, i)                                   ▷ O(log(I))
AgregarAtrás(itLista, it)                                         ▷ O(1)
```

Complejidad: $\mathcal{O}(log(A) + log(I))$
Justificación: Queda $\mathcal{O}(3 \times log(A) + 3 \times log(I) + 2) = \mathcal{O}(log(A) + log(I))$.

---

iOlvidarItem(**in/out** e: *estr*, **in** p: *persona*, **in** i: *item*)

```
items ← Significado(e.ventasHackeables, p)                        ▷ O(log(A))
ventas ← Significado(items, i)                                    ▷ O(log(I))
it ← Primero(ventas)                                              ▷ O(1)
venta ← Siguiente(it)                          ▷ Referencia modificable. O(1)
if venta.cant > 1 then                                            ▷ O(1)
    // Borramos un item de la venta
    venta.cant = venta.cant - 1                                   ▷ O(1)
else
    // Borramos la venta y el iterador
    EliminarSiguiente(it)                                         ▷ O(1)
    Comienzo(ventas)                                              ▷ O(1)
end if
```

Complejidad: $\mathcal{O}(log(A) + log(I))$
Justificación: Queda $\mathcal{O}(log(A) + log(I) + 5) = \mathcal{O}(log(A) + log(I))$.

---

iDiv(**in** n: *dinero*, **in** k: *nat*) $\rightarrow$ *res* : dinero

```
if n < k then
    return 0
else
    return 1 + div(n − k, k)
end if
```

Complejidad: $\mathcal{O}(n/k)$
Justificación: La cantidad de llamadas recursivas que hace es igual a la cantidad de veces que entra k en n.

---

iAplicarDescuento(**in** p: *dinero*, **in** d: *nat*) $\rightarrow$ *res* : dinero

```
return Div(p × (100 − d), 100)
```

Complejidad: $\mathcal{O}(n/k) \approx \mathcal{O}(1)$
Justificación: Es exactamente lo que tarda *div*, prácticamente constante para precios y descuentos sensatos.
