Algoritmos y Estructuras de Datos II

Trabajo Práctico de Diseño

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

LinkLinkIt

Catálogo de rutas en internet

3

Integrante	LU	Correo electrónico
Barabas, Ariel	775/11	ariel.baras@gmail.com
Izcovich, Sabrina	550/11	sizcovich@gmail.com
Otero, Fernando	424/11	fergabot@gmail.com
Vita, Sebastián	149/11	sebastian_vita@yahoo.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo DiccString(α)

1.1. Interfaz

parámetro formal: α.

usa: `Nat`, `Bool`, `String`, `Puntero(`α`)`,`vector(`α`)`.

se explica con: `Diccionario(String,` α`)`.

géneros: `diccString(`α`)`.

Operaciones básicas

VACIO() $\rightarrow$ *res* : `diccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} vacio$}
Complejidad: $\Theta(1)$
Descripción: Genera un diccionario vacío.

DEFINIR(**in/out** d: `diccString(`α`)`, **in** k: `string`, **in** s: α)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}} definir(k, s, d_0)$}
Complejidad: O($|k|$)
Descripción: En el caso en el que k no pertenezca al diccionario, se lo agrega. Caso contrario, se reemplaza el significado anterior por s.
Aliasing: La clave se guarda por copia y el significado por referencia.

DEFINIDO?(**in** d: `diccString(`α`)`, **in** k: `string`) $\rightarrow$ *res* : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} def?(k, d)$}
Complejidad: O($|k|$)
Descripción: Devuelve `True` sí y sólo sí k es una clave del diccionario.

OBTENER(**in/out** d: `diccString(`α`)`, **in** k: `string`) $\rightarrow$ *res* : α
Pre $\equiv$ {$def?(k, d)$}
Post $\equiv$ {$alias(res =_{\text{obs}} obtener(k, d))$}
Complejidad: O($|k|$)
Descripción: Devuelve el significado por referencia de la clave k en el diccString d.

1.2. Representación

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

donde `estr` es `tupla(`*trie*: `puntero(nodo)`)

donde `nodo` es `tupla(`*hijo*: `puntero(nodo)` , *hermano*: `puntero(nodo)` , *elem*: `puntero(`α`)` , *letra*: `char`)

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff 1 \wedge 2 \wedge 3$

Donde:

1. Todos los nodos cuyo hijo sea NULL, tienen elem distinto de NULL.
   Formalmente: $(\forall p : puntero(nodo))(p \in ePunteros(e.trie))) \Rightarrow_{\text{L}} (((*p).hijo = NULL) \Rightarrow (*p).elem \neq NULL)$
2. No existen dos `punteros(nodo)` iguales.
   Formalmente: $sinRepetidos(ePunteros(e.trie))$
3. No existen dos hermanos conteniendo la misma letra.
   Formalmente: $(\forall p : puntero(nodo))p \in ePunteros(e.trie) \Rightarrow_{\text{L}} sinRepetidos(letrasHermanos(p))$

ePunteros : puntero(nodo) $\longrightarrow$ secu(puntero(nodo))

ePunteros(p) $\equiv$ **if** p = NULL **then** <> **else** p • ePunteros((*p).hermano) & ePunteros((*p).hijo) **fi**

letrasHermanos : puntero(nodo) $\longrightarrow$ secu(char)

letrasHermanos(p) $\equiv$ **if** p = NULL **then** <> **else** p • letrasHermanos((*p).hermano) **fi**

ePunteros : puntero(nodo) $\longrightarrow$ secu(puntero(nodo))

ePunteros(p) $\equiv$ **if** p = NULL **then** <> **else** p • ePunteros((*p).hermano) & ePunteros((*p).hijo) **fi**

sinRepetidos : secu(α) $\longrightarrow$ bool

sinRepetidos(a) $\equiv$ $\neg$esta?(prim(a),fin(a)) $\wedge$ sinRepetidos(fin(a))

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

Abs(e) $\equiv$ d: DiccString | ($\forall$ c: string)(def?(c, d) $\Leftrightarrow$ ((e.trie $\neq$ NULL)) $\Rightarrow_{L}$ (c $\in$ eClavesTrie(*(e.trie), c))) $\wedge$ ($\forall$ v: vector(char))def?(v, d)) $\Rightarrow_{L}$(obtener(v, d) $\Leftrightarrow$ dameSignif(v,e.trie))

eClavesTrie : nodo $\times$ string $\longrightarrow$ conj(string)

eClavesTrie(n, pal) $\equiv$ **if** n.elem = NULL **then** $\emptyset$ **else** Ag(pal $\circ$ n.letra, $\emptyset$) **fi** $\cup$
if n.hermano = NULL **then** $\emptyset$ **else** eClavesTrie(*n.hermano, pal) **fi** $\cup$
if n.hijo = NULL **then** $\emptyset$ **else** eClavesTrie(*n.hijo, pal $\circ$ n.letra) **fi**

En este auxiliar y en los auxiliares que llama, supongo que la clave existe.

dameSignif : vector(char) $\times$ puntero(nodo) $\longrightarrow$ puntero(α)

dameSignif(a, p) $\equiv$ dameSignifAux(a, 0, p)

dameSignifAux : vector(char) $\times$ nat $\times$ puntero(nodo) $\longrightarrow$ puntero(α)

dameSignifAux(a, n, p) $\equiv$ **if** n = long(a)-1 **then** (*p).elem **else** dameSignifAux(a, n+1, hijoChar(p, a[n])) **fi**

hijoChar : puntero(nodo) $\times$ char $\longrightarrow$ puntero(nodo)

hijoChar(p, c) $\equiv$ **if** (*p).letra = c **then** (*p).hijo **else** hijoChar((*p).hermano, c) **fi**

Justificación de la elección de la estructura

Para representar el DiccString, hemos elegido la estructura mostrada anteriormente (que representa a un trie) ya que nos permite acceder al contenido del mismo con la complejidad requerida para realizar los algoritmos de otros módulos. Dicha estructura se explica de la siguiente manera:

- *trie* es un puntero al primer char del primer string añadido.
- *hijo* y *hermano* son punteros que nos sirven para poder recorrer el trie e ir formando los strings que se van almacenando en el trie.

hermano nos dice que esos dos caracteres (tanto el que se representa con el nodo que lo contiene como el nodo al que éste apunta) comparten el mismo string como prefijo.

hijo nos dice que el string continúa (en el caso en el que *hijo* no es *NULL*) o que finaliza ahí (en el caso contrario).

- *elem* es un puntero al significado de la palabra, por lo que el *elem* de un char puede o no ser NULL (dependiendo de si la palabra llegó o no a su fin).
- *letra* es el char que representa ese nodo.

Operacion Auxiliar

CREARNODO(**in** c: `char`) $\rightarrow$ res : `nodo`
Pre $\equiv$ {true}
Post $\equiv$ {$res.hijo =_{\text{obs}}$ `NULL` $\wedge$ $res.hermano =_{\text{obs}}$ `NULL` $\wedge$ $res.elem =_{\text{obs}}$ `NULL` $\wedge$ $res.letra =_{\text{obs}} c$}
Complejidad: $\Theta(1)$
Descripción: Genera un nodo con todos los punteros a `NULL` y con el char que se pasa como parámetro almacenado en *letra*.

1.3. Algoritmos

iCrearNodo(**in** c: `char`) $\rightarrow$ res : `nodo`

```
1: res.elem ← NULL                                   ▷ Θ(1)
2: res.hijo ← NULL                                   ▷ Θ(1)
3: res.hermano ← NULL                                ▷ Θ(1)
4: res.letra ← c                                     ▷ Θ(1)
```

Cálculo de complejidad: $4\Theta(1) = \Theta(1)$

iVacio() $\rightarrow$ res : `diccString(`α`)`

```
1: res.trie ← NULL                                   ▷ Θ(1)
```

Cálculo de complejidad: $\Theta(1)$

iObtener(**in/out** d: `diccString(`α`)`, **in** k: `string` $\rightarrow$ res : α

```
1: nodo ← *d.trie                                    ▷ O(1)
2: i ← 0                                             ▷ O(1)

3: while i < LONGITUD(k) do                          ▷ 256 * O(|k|) * O(1) = O(|k|)
4:     if nodo.letra = k[i] then                     ▷ O(1)
5:         if i + 1 = LONGITUD(k) then               ▷ O(1)
6:             res ← *nodo.elem                      ▷ O(1)
7:         else
8:             nodo ← *nodo.hijo                     ▷ O(1)
9:         end if
10:        i ← i + 1                                 ▷ O(1)
11:    else
12:        nodo ← *nodo.hermano                      ▷ O(1)
13:    end if
14: end while
```

Cálculo de complejidad: $2O(1) + O(|k|) = O(|k|)$
La complejidad del bucle se encuentra explicada en el algoritmo iDefinir.

iDefinir(**in/out** d: `diccString(`α`)`, **in** k: `string`, **in** s: α)

```
1: cargoTodo ← d.trie = NULL                                   ▷ O(1)
2: i ← 0                                                       ▷ O(1)

3: nodo es nodo
4: if cargoTodo then
5:     nodo ← crearNodo(k[i])                                  ▷ O(1)
6:     i ← i + 1                                               ▷ O(1)
7: else
8:     nodo ← *d.trie                                          ▷ O(1)
9: end if
```

// Si al tomar el primer nodo, *cargoTodo* es *true*, entonces en el bucle se crearán todos los nodos necesarios hasta formar k. En el caso contrario, se irá recorriendo el árbol entero hasta encontrar el lugar donde colocar el resto de los caracteres de k que no se encuentran en el árbol.

```
10: while i < Longitud(k) do                                   ▷ 256 * O(|k|) * O(1) = O(|k|)
11:     if cargoTodo then
12:         nodoVacio ← crearNodo(k[i])                        ▷ O(1)
13:         nodo.hijo ← &nodoVacio                             ▷ O(1)
14:         nodo ← nodoVacio                                   ▷ O(1)
15:         i ← i + 1                                          ▷ O(1)
16:     else
17:         if nodo.letra = k[i] then                          ▷ O(1)
18:             if nodo.hijo = NULL then
19:                 cargoTodo ← True                           ▷ O(1)
20:                 i ← i + 1                                  ▷ O(1)
21:             else
22:                 nodo ← *nodo.hijo                          ▷ O(1)
23:                 i ← i + 1                                  ▷ O(1)
24:             end if
25:         else
26:             if nodo.hermano ≠ NULL then
27:                 nodo ← *nodo.hermano                       ▷ O(1)
28:             else
29:                 nodoVacio ← crearNodo(k[i])                ▷ O(1)
30:                 nodo.hermano ← &nodoVacio                  ▷ O(1)
31:                 nodo ← nodoVacio                           ▷ O(1)
32:                 i ← i + 1                                  ▷ O(1)
33:                 cargoTodo ← True                           ▷ O(1)
34:             end if
35:         end if
36:     end if
37: end while

38: nodo.elem ← &s                                             ▷ O(1)
```

Cálculo de complejidad: $3O(1) + O(|k|) = O(|k|)$

En el peor caso, que es en el que *cargoTodo* es $false$ y la letra a buscar es la única faltante en ese nivel, el bucle tiene 256 iteraciones y, al ser un valor constante, no altera la complejidad.

iDefinido?(**in** d: `diccString(`α`)`, **in** k: `string`) $\rightarrow$ res : `Bool`

```
1: nodo es nodo
2: i ← 0                                                          ▷ O(1)

3: if d.trie = NULL then
4:     res ← False                                                ▷ O(1)
5:     i ← Longitud(k)                                            ▷ O(1)
6: else
7:     nodo ← *d.trie                                             ▷ O(1)
8: end if

9: while i < Longitud(k) do                                       ▷ 256 * O(|k|) * O(1) = O(|k|)
10:    if nodo.letra = k[i] then                                  ▷ O(1)
11:        if i + 1 = Longitud(k) then                            ▷ O(1)
12:            res ← True                                         ▷ O(1)
13:            i ← i + 1                                          ▷ O(1)
14:        else
15:            if nodo.hijo = NULL then
16:                res ← False                                    ▷ O(1)
17:                i ← Longitud(k)                                ▷ O(1)
18:            else
19:                nodo ← *nodo.hijo                              ▷ O(1)
20:                i ← i + 1                                      ▷ O(1)
21:            end if
22:        end if
23:    else
24:        if nodo.hermano ≠ NULL then
25:            nodo ← *nodo.hermano                               ▷ O(1)
26:        else
27:            res ← False                                        ▷ O(1)
28:            i ← Longitud(k)                                    ▷ O(1)
29:        end if
30:    end if
31: end while
```

Cálculo de complejidad: $2O(1) + O(|k|) = O(|k|)$

La complejidad del bucle se encuentra explicada en el algoritmo anterior.

1.4. Servicios Usados

Los siguientes módulos deben cumplir con los compromisos pedidos

- `Puntero(`α`)`
  - * debe tener complejidad O(1).
  - & debe tener complejidad O(1).
- `Vector(`α`)`

Longitud debe tener complejidad O(1).

- $[\bullet]$ debe tener complejidad O(1).

2. Módulo ArbolCategorias

2.1. Interfaz

usa: `Nat`, `String`, `diccString(`α`)`, `lista(`α`)`, `iteradorUni(`α`)`, `puntero(`α`)`.

se explica con: `ArbolCategorias`, `Iterador Unidireccional(`α`)`, `Vector(`α`)`.

géneros: `abCat`, `itAbCat`.

Operaciones básicas

NUEVOAC(**in** *raiz*: `categoria`) $\rightarrow$ *res* : `abCat`
Pre $\equiv$ {$\neg$*vacia?*(*raiz*)}
Post $\equiv$ {*res* $=_{\text{obs}}$ *nuevo*(*raiz*)}
Complejidad: $\Theta(|raiz|)$
Descripción: Crea un árbol con categoría *raiz*.

RAIZ(**in** *ac*: `abCat`) $\rightarrow$ *res* : `categoria`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ *raiz*(*ac*)}
Complejidad: $\Theta(1)$
Descripción: Devuelve el nombre de la categoría raiz por referencia, con lo cual, si éste valor es modificado, se romperá el invariante de representación invalidando todo el árbol.

AGREGAR(**in/out** *ac*: `abCat`, **in** *padre*: `categoria`, **in** *hija*: `categoria`)
Pre $\equiv$ {$ac =_{\text{obs}} ac_0 \wedge$ *está?*$(padre, ac) \wedge \neg$*vacía?*$(hija) \wedge \neg$*está?*$(hija, ac)$}
Post $\equiv$ {$ac =_{\text{obs}}$ *agregar*$(ac_0, padre, hija)$}
Complejidad: $\Theta(|padre| + |hija|)$
Descripción: Agrega la categoría *hija* a *padre*.

HIJAS(**in** *ac*: `abCat`, **in** *padre*: `categoria`) $\rightarrow$ *res* : `iteradorUni(categoria)`
Pre $\equiv$ {*está?*$(padre, ac)$}
Post $\equiv$ {*res* $=_{\text{obs}}$ *crearItUni*(*conjuntoASecuencia*(*hijos*$(ac, padre)$))}
Complejidad: $\Theta(|padre|)$
Descripción: Devuelve un iterador que proyecta las hijas del padre de forma tal que, al pedir actual, el iterador devuelve el primer elemento de la lista.
Aliasing: El iterador se invalida sí y sólo sí se elimina el elemento siguiente del iterador.

PADRES(**in** *ac*: `abCat`, **in** *hija*: `categoria`) $\rightarrow$ *res* : `itAbCat`
Pre $\equiv$ {*está?*$(hija, ac)$}
Post $\equiv$ {*res* $=_{\text{obs}}$ *CrearItUni*(*padres*$(ac, hija)$)}
Complejidad: $\Theta(|hija|)$
Descripción: Devuelve un iterador que proyecta los IDs de los padres de forma tal que, al pedir ActualID, el iterador devuelve el id de la hija.

ID(**in** *ac*: `abCat`, **in** *c*: `categoria`) $\rightarrow$ *res* : `Nat`
Pre $\equiv$ {*está?*(c, ac)}
Post $\equiv$ {*res* $=_{\text{obs}}$ *id*(ac, c)}
Complejidad: $\Theta(|c|)$
Descripción: Devuelve el id de la categoría dada.

CANTCATEGORIAS(**in** *ac*: `abCat`) $\rightarrow$ *res* : `Nat`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ *#categorias*(ac)}
Complejidad: $\Theta(1)$
Descripción: Devuelve la cantidad de categorías existentes en el árbol.

Funciones auxiliares

conjuntoASecuencia : Conjunto(α) $\longrightarrow$ Secuencia(α)

conjuntoASecuencia(c) $\equiv$ **if** $\emptyset?(c)$ **then** $<>$ **else** dameUno(c) $\bullet$ conjuntoASecuencia(sinUno(c)) **fi**

padres : acat ac $\times$ categoria *hija* $\longrightarrow$ Secuencia(Nat) {está?($hija$, ac)}

padres(ac, *hija*) $\equiv$ **if** *hija* $=_{\text{obs}}$ raiz(ac) **then** $1 \bullet <>$ **else** id(ac, *hija*) $\bullet$ padres(ac, padre(ac, *hija*)) **fi**

Operaciones del iterador

HAYMASPADRES?(**in** *it*: `itAbCat`) $\rightarrow$ *res* : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ *HayMas?*(it)}
Complejidad: $\Theta(1)$

Descripción: Devuelve `True` sí y sólo sí en el iterador todavía quedan elementos por iterar.

SUBIR(**in/out** *it*: `itAbCat`)
Pre $\equiv \{it =_{\text{obs}} it_0 \wedge hayMas?(it)\}$
Post $\equiv \{it =_{\text{obs}} Avanzar(it_0)\}$
Complejidad: $\Theta(1)$
Descripción: Avanza el iterador a la siguiente posición.

ACTUALID(**in** *it*: `itAbCat`) $\rightarrow$ *res* : `Nat`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} Actual(it)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve el id del elemento al que está apuntando.

2.2. Representación

`abCat` se representa con `estr`

donde `estr` es `tupla`(*categorias*: `diccString(catInfo)` , *raiz*: `categoria` , *ultID*: `Nat`)

donde `catInfo` es `tupla`(*id*: `Nat` , *padre*: `puntero(catInfo)` , *hijas*: `lista(categoria)`)

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff 1 \wedge 2 \wedge 3 \wedge_{\text{L}} 4 \wedge_{\text{L}} 5 \wedge 6 \wedge 7 \wedge_{\text{L}} 8 \wedge 9 \wedge_{\text{L}} 10 \wedge 11$

Donde:

ACLARACIÓN: Llamaremos a la operación "esta?" (definida en el TAD secuencia) como "estaLista?" para no generar confusiones.

1. El id de la raíz del árbol es 1 (requerido por el enunciado del TP).
   Formalmente: $id(e.raiz) =_{obs} 1$
2. No existe ninguna categoría incluida en sus categorías hijas.
   Formalmente: $(\forall c : categoria)esta?(c, claves(e.categorias)) \Rightarrow_{\text{L}} \neg(estaLista?(c, eHijas(c)))$
3. Todas las categorías tienen padre, excluyendo la raíz.
   Formalmente: $(\forall c : categoria)(esta?(c, claves(e.categorias)) \wedge (c \neq e.raiz)) \Rightarrow_{\text{L}} ((\exists t : categoria)(esta?(t, claves(e.categorias)) \wedge t \neq c) \wedge_{\text{L}} estaLista?(c, eHijas(t)))$
4. Existe una categoría que no tiene padre y que es igual a raíz.
   Formalmente: $(\exists c : categoria)(esta?(c, claves(e.categorias)) \wedge_{\text{L}} c =_{obs} e.raiz) \Rightarrow_{\text{L}} (e.raiz.padre) = Null$
5. Ninguna categoría tiene más de un padre.
   Formalmente: $\neg(\exists t, c : categoria)((esta?(c, claves(e.categorias))) \wedge_{\text{L}} (esta?(t, claves(e.categorias))) \wedge_{\text{L}}(t \neq c) \wedge (t \neq e.raiz)) \wedge_{\text{L}} ((\forall k : categoria)esta?(k, claves(e.categorias)) \Rightarrow_{\text{L}} (estaLista?(k, eHijas(c)) \wedge_{\text{L}} estaLista?(k, eHijas(t)))$
6. La cantidad de categorías totales es igual a UltId.
   Formalmente: $\#claves(e.categorias) =_{obs} e.ultId$
7. No existen dos IDs iguales para categorías distintas.
   Formalmente: $(\forall c, t : categoria)(esta?(c, claves(e.categorias)) \wedge esta?(t, claves(e.categorias)) \wedge t \neq c) \Rightarrow_{\text{L}} id(c) \neq id(t)$
8. Las categorías hijas de cada categoría tienen un id mayor a ésta.
   Formalmente: $((\forall c : categoria)esta?(c, claves(e.categorias))) \Rightarrow_{\text{L}} ((\forall t : categoria)(esta?(c, claves(e.categorias)) \wedge_{\text{L}} estaLista?(t, eHijas(c))) \Rightarrow_{\text{L}} id(c) < id(t))$
9. Ningún padre es Null salvo el de la raíz.
   Formalmente: $(\forall t : categoria)(esta?(t, claves(e.categorias)) \wedge (t \neq e.raiz)) \Rightarrow_{\text{L}} ePadre(e, t) \neq Null$

10. El padre de la raíz es Null.
Formalmente: $ePadre(e, e.raiz) =_{obs} Null$

11. La altura del árbol está acotada por un natural. Esta propiedad nos asegurará que el árbol no sea cíclico.
Formalmente: $(\exists n : nat)acotado(e, e.raiz, n)$

acotado : estr × categoria × Nat ⟶ Bool

acotado(a, cat, n) ≡ **if** $n = 0$ **then** `False` **else** (∀ h:categoria)(h ∈ eHijas(a, cat))acotado(n-1, a, h) **fi**

eHijas : estr e × categoria c ⟶ secu(categoria) {def?(c, e.categorias)}

eHijas(e, c) ≡ obtener(e.categorias, c).hijas

ePadre : estr e × categoria c ⟶ puntero(catInfo) {def?(c, e.categorias)}

ePadre(e, c) ≡ obtener(e.categorias, c).padre

Abs : estr e ⟶ acat {Rep(e)}

Abs(e) ≡ a:acat | e.raiz = raiz(a) ∧ (∀ c:categoria)esta?(c,claves(e.categorias)) $\Rightarrow_L$ eId(e,c) = id(a,c) ∧ ((∀ c:categoria)esta?(c,claves(e.categorias)) $\Rightarrow_L$ ((∃ t: categoria)esta?(c,claves(e.categorias)) $\Rightarrow_L$ esSuId?(t,(ePadre(e,c).id)) $\wedge_L$ padre(a,c) = t) $\wedge_L$ categorias(a) = claves(e.categorias)

eId : estr e × categoria c ⟶ Nat {def?(c, e.categorias)}

eId(e, c) ≡ obtener(e.categorias, c).id

esSuId? : estr e × categoria c × Nat i ⟶ Bool {def?(c, e.categorias) ∧ i <= e.ultId}

esSuId?(e, i) ≡ obtener(e.categorias, c).id = i

Justificación de la representacion de la estructura

Para representar el árbol de categorías elegimos esta estructura por los siguientes motivos:

- El Diccionario String modulariza perfectamente un árbol de categorías dado que es posible conocer los hijos de cada una de las categorías con la complejidad requerida, para esto, se colocó en cada nodo una lista con los nombres de cada hijo.
- Esta estructura nos permite acceder a la raíz del arbol en O(1) y al ID de cada categoría en O($|c|$) donde c es el nombre de la categoría.
- Tener un puntero a la categoría padre nos permite luego hacer un iterador de padres el cual utilizaremos en linkLinkIt.

La estructura elegida se puede explicar de la siguiente manera:

- *categorias* es un DiccString que almacena cada categoría junto con su información.
- *raiz* es la categoría raiz del arbol de categorias.
- *ultId* es el *id* de la última categoría agregada al arbol.
- *id* es el id de la categoría cuya $catInfo$ está siendo observada.
- *padre* es un puntero a la información del padre de la categoría que está siendo observada.
- *hijas* es la lista de las hijas directas de un categoría.

2.3. Iterador

`itAbCat` se representa con `estr`

donde `estr` es `tupla`(*iterador*: `puntero(catInfo)`)

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ e.iterador $\neq$ NULL

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

Abs(e) $\equiv$ it:itAbCat | Siguientes(it) $=_{obs}$ ePadres(e.iterador)

ePadres : puntero(catInfo) $\longrightarrow$ secu(Nat)

ePadres(pci) $\equiv$ **if** pci = NULL **then** <> **else** (*pci).id • ePadres((*pci).padre) **fi**

Justificación de la elección de la estructura

El iterador necesita, únicamente, guardar un puntero a la categoría actual ya que con esto alcanza para acceder al padre y, de esta manera, subir en O(1). Éste se usará en el LinkLinkIt.

2.4. Algoritmos

iNuevoAC(**in** *raiz*: `categoria`) $\rightarrow$ *res* : `abCat`

1: $res.categorias \leftarrow$ VACIO() ▷ O(1)
2: $res.raiz \leftarrow raiz$ ▷ O($|raiz|$)
3: $res.ultID \leftarrow 1$ ▷ O(1)

4: $infoRaiz$ **es** `catInfo`
5: $infoRaiz.id \leftarrow res.ultID$ ▷ O(1)
6: $infoRaiz.padre \leftarrow$ `NULL` ▷ O(1)
7: $infoRaiz.hijas \leftarrow$ VACIA() ▷ O(1)
8: DEFINIR($res.categorias, raiz, infoRaiz$) ▷ O($|raiz|$)

Cálculo de complejidad: $5O(1) + 2O(|raiz|) = O(|raiz|)$

iRaiz(**in** *ac*: `abCat`) $\rightarrow$ *res* : `categoria`

1: $res \leftarrow ac.raiz$ ▷ O(1)

Cálculo de complejidad: $O(1)$

iPadres(**in** *ac*: `abCat`, **in** *hija*: `categoria`) $\rightarrow$ *res* : `itAbCat`

1: $nodoHija \leftarrow$ OBTENER($ac.categorias, hija$) ▷ O($|hija|$)
2: $iterador \leftarrow \&nodoHija$ ▷ O(1)
3: $res \leftarrow iterador$ ▷ O(1)

Cálculo de complejidad: $O(|hija|) + 2O(1) = O(|hija|)$

iAgregar(**in/out** ac: `abCat`, **in** $padre$: `categoria`, **in** $hija$: `categoria`)

1: $ac.ultID \leftarrow ac.ultID + 1$ ▷ O(1)
2: $infoPadre \leftarrow$ OBTENER$(ac.categorias, padre)$ ▷ O($|padre|$)

3: $infoHija$ **es** `catInfo`
4: $infoHija.id \leftarrow ac.ultID$ ▷ O(1)
5: $infoHija.padre \leftarrow \&infoPadre$ ▷ O(1)
6: $infoHija.hijas \leftarrow$ VACIA() ▷ O(1)
7: DEFINIR$(ac.categorias, hija, infoHija)$ ▷ O($|hija|$)

8: AGREGARATRAS$(infoPadre.hijas, hija)$ ▷ O($|hija|$)

Cálculo de complejidad: $4O(1) + 2O(|hija|) + O(|padre|) = O(|hija|) + O(|padre|) = O(|hija| + |padre|)$

iHijas(**in** ac: `abCat`, **in** $padre$: `categoria`) $\rightarrow$ res : `iteradorUni(categoria)`

1: $nodoHijo \leftarrow$ OBTENER$(ac.categorias, padre)$ ▷ O($|padre|$)
2: $res \leftarrow$ CREARITERADOR$(nodoHijo.hijas)$ ▷ O(1)

Cálculo de complejidad: $O(|padre|) + O(1) = O(|padre|)$

iID(**in** ac: `abCat`, **in** c: `categoria`) $\rightarrow$ res : `Nat`

1: $catInfo \leftarrow$ OBTENER$(ac.categorias, c)$ ▷ O($|c|$)
2: $res \leftarrow catInfo.id$ ▷ O(1)

Cálculo de complejidad: $O(|c|) + O(1) = O(|c|)$

iCantCategorias(**in** ac: `abCat`) $\rightarrow$ res : `Nat`

1: $res \leftarrow ac.ultId$ ▷ $O(1)$

Cálculo de complejidad: $O(1)$

2.4.1. Algoritmos del Iterador

iHayMasPadres?(**in** it: `itAbCat`) $\rightarrow$ res : `Bool`

1: $res \leftarrow (*it.iterador).padre \neq$ `NULL` ▷ $\Theta(1)$

Cálculo de complejidad: $\Theta(1)$

iSubir(**in/out** it: `itAbCat`)

1: $it \leftarrow (*it.iterador).padre$ ▷ $\Theta(1)$

Cálculo de complejidad: $\Theta(1)$

iActualID(**in** it: `itAbCat`) $\rightarrow$ res : `Nat`

1: $res \leftarrow (*it.iterador).id$ ▷ $\Theta(1)$

Cálculo de complejidad: $\Theta(1)$

2.5. Servicios Usados

Los siguientes módulos deben cumplir con los compromisos pedidos a continuación:

- `Lista Enlazada` (α)

Vacia debe tener complejidad O(1).
AgregarAtras debe tener complejidad O($copy(a)$), donde a es el tipo del elemento que se va a colocar en la lista.

- `Iterador Unidireccional (`α`)`

crearIterador debe tener complejidad O(1).

- `diccString(`α`)`

Vacio debe tener complejidad O(1).
Definir debe tener complejidad O($|k|$) donde k es el string que se va a colocar por definición.
Obtener debe tener complejidad O($|k|$) donde k es la clave.

- `Puntero(`α`)`

& debe tener complejidad O(1).
* debe tener complejidad O(1).

3. Módulo LinkLinkIt

3.1. Interfaz

usa: `Nat`, `Bool`, `String`, `Puntero(`α`)`, `abCat`, `itAbCat`, `diccString(`α`)`, `vector(`α`)`, `lista(`α`)`, `itLista(`α`)`.

se explica con: `linkLinkIt, ArbolCategorias, Iterador Unidireccional, tupla`.

géneros: `LinkLinkIt`.

Operaciones básicas

CREARLINKLINKIT(**in** ac: `abCat`) $\rightarrow$ res : `LinkLinkIt`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} iniciar(ac)\}$
Complejidad: $\Theta(\#categorias(ac))$
Descripción: Crea un sistema LinkLinkIt. El árbol de categorías se guarda por refencia.

AGREGARLINK(**in/out** s: `LinkLinkIt`, **in** l: `link`, **in** c: `categoria`)
Pre $\equiv$ $\{s =_{\text{obs}} s_0 \wedge l \notin links(s) \wedge está?(c, categorias(s))\}$
Post $\equiv$ $\{s =_{\text{obs}} nuevoLink(s_0, l, c)\}$
Complejidad: $O(|l| + |c| + h)$
Descripción: Agrega un link a la categoría señalada.

ACCEDERLINK(**in/out** s: `LinkLinkIt`, **in** l: `link`, **in** f: `fecha`)
Pre $\equiv$ $\{s =_{\text{obs}} s_0 \wedge l \in$ links$(s) \wedge f \geq$ fechaActual$(s)\}$
Post $\equiv$ $\{s =_{\text{obs}} acceso(s_0, l, f)\}$
Complejidad: $O(|l|)$
Descripción: Registra un acceso al link provisto.

CANTLINKS(**in** s: `LinkLinkIt`, **in** c: `categoria`) $\rightarrow$ res : `Nat`
Pre $\equiv$ $\{está?($c, categorias(s)$)\}$
Post $\equiv$ $\{res =_{\text{obs}} cantLinks(s, c)\}$
Complejidad: $O(|c|)$
Descripción: Devuelve la cantidad de links que tiene una categoría y sus hijas.

LINKSORDENADOSPORACCESOS(**in/out** s: `LinkLinkIt`, **in** c: `categoria`) $\rightarrow$ res : `itLinks`
Pre $\equiv$ $\{está?(c, categorias(s))\}$
Post $\equiv$ $\{res =_{\text{obs}} CrearItUni(secuInfoLinks(s, c))\}$
Complejidad: $O(|c| + n^2)$
Descripción: Devuelve la cantidad de links que tiene una categoría.

Funciones auxiliares

secuInfoLinks : lli s $\times$ categoria c $\longrightarrow$ secu(tupla(link, categoria, Nat)){está?(c, categorias(s))}
secuInfoLinks(s, c) $\equiv$ infoLinks(s, c, linksOrdenadosPorAccesos(s, c))

infoLinks : lli s × categoria c × secu(link) ls ⟶ secu(tupla(link, categoria, Nat))
$\left\{\begin{array}{l}\text{está?(c, categorias(s))} \wedge (\forall \text{ l:link})(\text{está?(l, ls)} \Rightarrow \text{l} \in \text{links(s)} \wedge \text{esSubCategoria(s.aCategorias, c, catego-}\\ \text{riaLink(s.aCategorias, l)))}\end{array}\right\}$

infoLinks(s, c, ls) ≡ **if** Vacia(ls) **then**
<>
else
tupla(prim(ls), categoriaLink(s, prim(ls)), accesosRecientes(s, c, prim(ls))) • infoLinks(s, c, fin(ls))
fi

Operaciones del iterador

CrearItLinks(**in** ls: `lista(puntero(infoLink))`, **in** f: `fecha`) → res : `itLinks`
Pre ≡ $\{(\forall\ l : infoLink)(está?(l, ls) \Rightarrow l.ultAcceso \leq f)\}$
Post ≡ $\{res =_{\text{obs}} CrearItUni(eInfoLinksAUniTupla(ls, f))\}$
Complejidad: $O(1)$
Descripción: Genera un iterador de links que permite verse a si mismo, observar su categoría y la cantidad de accesos recientes. eInfoLinksAUniTupla está axiomatizado en la página 16.

SiguienteLink(**in** it: `itLinks`) → res : `link`
Pre ≡ $\{HayMas?(it)\}$
Post ≡ $\{res =_{\text{obs}} \Pi_1(Actual(it))\}$
Complejidad: $O(1)$
Descripción: Devuelve el siguiente link.

SiguienteCategoria(**in** it: `itLinks`) → res : `categoria`
Pre ≡ $\{HayMas?(it)\}$
Post ≡ $\{res =_{\text{obs}} \Pi_2(Actual(it))\}$
Complejidad: $O(1)$
Descripción: Devuelve la categoría del siguiente link.

SiguienteAccesosRecientes(**in** it: `itLinks`) → res : `Nat`
Pre ≡ $\{HayMas?(it)\}$
Post ≡ $\{res =_{\text{obs}} \Pi_3(Actual(it))\}$
Complejidad: $O(1)$
Descripción: Devuelve los accesos recientes del siguiente link.

HaySiguiente(**in** it: `itLinks`) → res : `Bool`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} HayMas?(it)\}$
Complejidad: $O(1)$
Descripción: Informa si hay un link siguiente.

Avanzar(**in/out** it: `itLinks`)
Pre ≡ $\{it = it_0 \wedge HayMas?(it)\}$
Post ≡ $\{it =_{\text{obs}} Avanzar(it_0)\}$
Complejidad: $O(1)$
Descripción: Avanza al siguiente link.

3.2. Representación

`LinkLinkIt` **se representa con** `estr`

donde `estr` es `tupla`(*aCategorias*: `abCat` ,
linksPorCat: `vector(lista(puntero(infoLink)))` ,
infoLinks: `diccString(infoLink)`)

donde `infoLink` es `tupla`(*link*: `link` , *categoria*: `categoria` , *ultAcceso*: `fecha` , *accesos*: `vector(Nat)`)

Rep : estr ⟶ bool

$\text{Rep}(e) \equiv \text{true} \iff 1 \wedge 2 \wedge (\forall pil : puntero(infoLink))(pil \in aplanar(e.linksPorCat) \Rightarrow_{\text{L}} 3 \wedge_{\text{L}} 4)$

Donde:

1. La longitud de linksPorCat es igual a la cantidad de categorías existentes.
   Formalmente: $\#e.linksPorCat =_{\text{obs}} \#categorias(e.aCategorias)$
2. Todo infoLink definido en *infoLinks* aparece en la lista que le corresponde de *linksPorCat* según el ID de su categoría y las categorías padres a ella.
   Formalmente: $(\forall l : link, il : infoLink)(def?(e.infoLinks, link) \wedge_{\text{L}} obtener(e.infoLinks, link) =_{\text{obs}} il \Rightarrow (\forall c : Nat)(c \in eCategoriasPadres(e.aCategorias, il) \Rightarrow_{\text{L}} esta?(\&il, indice(e.linksPorCat, c))))$
3. Todo infoLink en linksPorCat debe obtenerse y ser el mismo en infoLinks con la clave igual a su link.
   Formalmente: $pil \neq NULL \wedge_{\text{L}} \&obtener(e.infoLinks, (*pil).link) =_{\text{obs}} pil$
4. El vector *accesos* de todo infoLink tiene longitud 3.
   Formalmente: $\#(*pil).accesos =_{\text{obs}} 3$

eCategoriasPadres : abCat ac × infoLink il $\longrightarrow$ conj(Nat) {il.categoria ∈ categorias(ac)}

eCategoriasPadres(ac, il) ≡ Ag(id(ac, il.categoria), **if** il.padre ≠ NULL **then** eCategoriasPadres(*il.padre) **else** ∅ **fi**)

aplanar : secu(secu(α)) $\longrightarrow$ secu(α)

aplanar(ss) ≡ **if** vacia?(ss) **then** <> **else** prim(ss) & aplanar(fin(ss)) **fi**

indice : secu(α) s × Nat i $\longrightarrow$ α {i <#s}

indice(s, i) ≡ **if** i = 0 **then** prim(s) **else** indice(fin(s), i - 1) **fi**

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

Abs(e) ≡ s : lli | categorias(s) = e.categorias ∧
links(s) = claves(s.infoLinks) ∧
(∀ l:link) l ∈ links(s) ⇒$_{\text{L}}$categoriaLink(s, l) = obtener(l, e.linksPorCat).categoria ∧
fechaActual(s) = mayorUltimoAcceso(indice(s.linksPorCat, 1)) ∧
(∀ k:link) k ∈ links(s) ⇒$_{\text{L}}$ fechaUltimoAcceso(s, k) = obtener(e.infoLinks, k).ultAcceso ∧
(∀ l:link) l ∈ links(s) ⇒$_{\text{L}}$(∀ f:fecha) accesosRecientesDia(s, l, f) = cantAccesosEnFecha(obtener(e.infoLinks, l), f)) ∧
(∀ k: link) k ∈ links(s) ⇒$_{\text{L}}$categoriaLink(s, k) = obtener(e.infoLinks, k).categoria

mayorUltimoAcceso : lista(puntero(infoLinik)) $\longrightarrow$ fecha

```
mayorUltimoAcceso(list) ≡ if vacia?(list) then
                              max(prim(list).ultAcceso, mayorUltimoAcceso(fin(list)))
                          else
                              0
                          fi
```

cantAccesosEnFecha : infoLink × fecha $\longrightarrow$ nat

```
cantAccesosEnFecha(info, f) ≡ if (f <info.ultAcceso -2) ∨ (f >info.ultAcceso) then
                                  0
                              else
                                  indice(info.accesos,f-info.ultAcceso)
                              fi
```

Justificación de la elección de la estructura

Para representar al sistema LinkLinkIt elegimos esta representación ya que relaciona los módulos necesarios para definirla respetando las complejidades requeridas. Dicha representación se explica de la siguiente manera:

- en estr
  - *aCategorias* es el árbol de categorías del sistema.
  - *linksPorCat* es un vector de lista de links. Los índices de dicho vector representan al ID de la categoría correspondiente a cada lista de links. Las listas de links no tienen únicamente sus links sino que también contienen los links de sus categorías hijas. Cuando hablamos de link, nos estamos refiriendo en realidad a un `puntero(InfoLink)`.
  - $infoLinks$ es un DiccString que guarda, para cada link, sus datos representados por un $infoLink$.
- en infoLink
  - *link* es el link del que queremos observar la información.
  - *categoria* es la categoría del link que estamos observando.
  - *ultAcceso* es la fecha de la última visita realizada a ese link.
  - *accesos* es un vector de 3 posiciones en el que se guarda la cantidad de visitas de los 3 últimos días (respectivamente) para el link observado.

3.3. Iterador

`itLinks` **se representa con** `estr`

donde `estr` es `tupla`(*it*: `itLista(puntero(infoLink))` , *ultAcceso*: `fecha`)

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ $(\forall p : puntero(infoLink))(p \in Siguientes(e.it) \Rightarrow_{\text{L}} (1 \wedge_{\text{L}} 2))$

Donde:

1. Ningún puntero puede ser NULL.
   Formalmente: $p \neq NULL$
2. ultAcceso debe representar el mayor último acceso de todos los links.
   Formalmente: $(*p).ultAcceso \leq e.ultAcceso$

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

Abs(e) $\equiv$ it:itUni | Siguientes(it) $=_{\text{obs}}$ eInfoLinksAUniTupla(Siguientes(e.it), e.ultAcceso)

eInfoLinksAUniTupla : secu(puntero(infoLink)) spil $\times$ fecha f $\longrightarrow$ secu(tupla(link, categoria, Nat))
$\{(\forall pil : puntero(infoLink))(esta?(pil, spil) \Rightarrow (pil \neq NULL \wedge_{\text{L}} (*pil).ultAcceso \leq f))\}$

eInfoLinksAUniTupla(spil, f) $\equiv$ **if** vacio?(spil) **then**
<>
else
tupla((*prim(spil)).link, (*prim(spil)).categoria, puntajeDelLink(*prim(spil), f))
• eInfoLinksAUniTupla(fin(spil), f)
fi

Justificación de la elección de la estructura

Esta estructura contiene una lista de links para los cuales se puede saber su categoría y su "accesosRecientes", uno por uno. Para este último dato, necesitamos tener una copia del sistema y saber la última fecha en la cual se accedió alguno de estos links, ésta la guardamos en *utlAcceso* para poder tenerla rápidamente.

Operacion Auxiliar

PUNTAJEDELLINK(**in** *il*: `infoLink`, **in** *ultAccesoCat*: `fecha`) $\rightarrow$ *res* : `Nat`
Pre $\equiv \{il.ultAcceso \leq f\}$

Post $\equiv \{res =_{\text{obs}} puntajeDelLink(il, ultAccesoCat)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve el puntaje del link. Esta función es privada ya que recibe una estructura interna.

puntajeDelLink : infoLink il × fecha f $\longrightarrow$ Nat {il.ultAcceso ≤ f}

```
puntajeDelLink(il, f) ≡ if il.ultAcceso + 3 > f then
                            0
                        else
                            indice(il.accesos, f - il.ultAcceso) + puntajeDelLink(il.ultAcceso, f + 1)
                        fi
```

3.4. Algoritmos

iCrearLinkLinkIt(**in** ac: `abCat`) → res : `LinkLinkIt`

```
1: res.aCategorias ← ac                                                          ▷ Θ(1)
2: res.linksPorCat ← VACIA()                                                     ▷ Θ(1)
3: res.infoLinks ← VACIO()                                                       ▷ Θ(1)

4: for i = 1 → CANTCATEGORIAS(ac) do   ▷ O(1) + #categorias(ac) * Θ(1) = Θ(#categorias(ac))
5:     AGREGARATRAS(res.linksPorCat, VACIA())                                    ▷ Θ(1)
6: end for
```

Cálculo de complejidad: $3\Theta(1) + \Theta(\#categorias(ac)) = \Theta(\#categorias(ac))$

iAgregarLink(**in/out** s: `LinkLinkIt`, **in** l: `link`, **in** c: `categoria`)

```
1: categoriaID ← ID(s.aCategorias, c)                                            ▷ O(|c|)

2: info es infoLink
3: info.link ← l                                                                 ▷ O(|l|)
4: info.categoria ← c                                                            ▷ O(|c|)
5: info.ultAcceso ← 0                                                            ▷ O(1)
6: info.accesos ← VACIA()                                                        ▷ O(1)
7: for i = 1 → 3 do                                                              ▷ 3O(1) = O(1)
8:     AGREGARATRAS(info.accesos, 0)                                             ▷ O(1)
9: end for
10: DEFINIR(s.infoLinks, l, info)                                                ▷ O(|l|)
11: AGREGARATRAS(s.linksPorCat[categoriaID − 1], &info)                          ▷ O(1)

12: it ← PADRES(s.aCategorias, c)                                                ▷ O(|c|)
13: while HAYMASPADRES?(it) do                                                   ▷ (O(h) + O(1)) * 3O(1) = O(h)
14:     SUBIR(it)                                                                ▷ O(1)
15:     categoriaID ← ACTUALID(it)                                               ▷ O(1)
16:     AGREGARATRAS(s.linksPorCat[categoriaID − 1], &info)                      ▷ O(1)
17: end while
```

Cálculo de complejidad: $4O(1) + 3O(|c|) + 2O(|l|) + O(h) = O(|c|) + O(|l|) + O(h) = O(|c| + |l| + h)$
El bucle tiene h iteraciones ya que inicia en un nodo y luego, en cada ciclo, se va subiendo hasta la raiz del árbol.

iAccederLink(**in/out** s: `LinkLinkIt`, **in** l: `link`, **in** f: `fecha`)

```
1: link ← OBTENER(s.infoLinks, l)                                   ▷ O(|l|)
2: diff ← f − link.ultAcceso                                        ▷ O(1)

   // Muevo hacia atrás los días que pasaron
3: i ← 0                                                            ▷ O(1)
4: while i < 3 − diff do                                            ▷ 3 * 2O(1) = O(1)
5:     link.accesos[i] ← link.accesos[i + diff]                     ▷ 2O(1) = O(1)
6:     i ← i + 1                                                    ▷ O(1)
7: end while

   // Pongo en 0 los nuevos días que llegaron
8: i ← 0                                                            ▷ O(1)
9: while i < diff ∧ i < 3 do                                        ▷ 3 * 2O(1) = O(1)
10:    link.accesos[2 − i] ← 0                                      ▷ O(1)
11:    i ← i + 1                                                    ▷ O(1)
12: end while

13: link.ultAcceso ← f                                              ▷ O(1)
14: link.accesos[2] ← link.accesos[2] + 1                           ▷ 2O(1) = O(1)
```

Cálculo de complejidad: $7O(1) + O(|l|) = O(|l|)$

iCantLinks(**in** s: `LinkLinkIt`, **in** c: `categoria`) → res : `Nat`

```
1: categoriaID ← ID(s.aCategorias, c)                               ▷ O(|c|)
2: res ← LONGITUD(s.linksPorCat[categoriaID − 1])                   ▷ 2O(1) = O(1)
```

Cálculo de complejidad: $O(|c|) + O(1) = O(|c|)$

iPuntajeDelLink(**in** il : `infoLink`, **in** $ultAccesoCat$: `fecha`) → res : `Nat`

```
1: res ← 0                                                          ▷ O(1)
2: i ← 3                                                            ▷ O(1)
3: while ultAccesoCat − il.ultAcceso < i do ▷ gracias a la Pre (ultAccesoCat − il.ultAcceso) > 0 ⇒ 3O(1) = O(1)
4:     i ← i − 1                                                    ▷ O(1)
5:     res ← res + il.accesos[i]                                    ▷ O(1)
6: end while
```

Cálculo de complejidad: $3O(1) = O(1)$

iLinksOrdenadosPorAccesos(**in**/**out** s: `LinkLinkIt`, **in** c: `categoria`) $\rightarrow$ res : `itLinks`

```
1:  categoriaID ← ID(s.aCategorias, c)                                   ▷ O(|c|)
2:  lista ← s.linksPorCat[categoriaID − 1]                               ▷ O(1)

    // Encuentro la fecha más alta de esta categoría.
3:  it ← crearIt(lista)                                                  ▷ O(1)
4:  f ← 0                                                                ▷ O(1)
5:  while HaySiguiente(it) do                                            ▷ nO(1) = O(n)
6:      f ← max(Siguiente(it).ultAcceso, f)                              ▷ O(1)
7:      Avanzar(it)                                                      ▷ O(1)
8:  end while

    // Ya está ordenado?
9:  it ← crearIt(lista)                                                  ▷ O(1)
10: estaOrdenada ← True                                                  ▷ O(1)
11: ultPuntaje ← −1                                                      ▷ O(1)
12: while HaySiguiente(it) ∧ estaOrdenada do                             ▷ (n + O(1)) ∗ 2O(1) = O(n)
13:     if ultPuntaje > −1 then
14:         estaOrdenada ← ultPuntaje ≥ puntajeDelLink(Siguiente(it), f) ▷ 2O(1)
15:     end if
16:     ultPuntaje ← puntajeDelLink(Siguiente(it), f)                    ▷ 2O(1)
17: end while
18: if estaOrdenada then
19:     return res ← crearItLinks(lista, f, s)                           ▷ O(1)
20: end if

    // Ordeno agregando n veces el de más accesos que encuentre.
21: listaOrdenada ← Vacia()                                              ▷ O(1)
22: itOrd ← crearIt(listaOrdenada)                                       ▷ O(1)
23: while ¬Vacia(lista) do                                               ▷ n ∗ (O(n) + 3O(1)) = O(n^2)
24:     itRes ← itMaxAccesos ← crearIt(lista)                            ▷ 2O(1) = O(1)
25:     while HaySiguiente(itRes) do                                     ▷ n ∗ 5O(1) = O(n)
26:         if puntajeDelLink(Siguiente(itRes), f) > puntajeDelLink(Siguiente(itMaxAccesos), f) then
27:             itMaxAccesos ← itRes
28:         end if
29:     end while

30:     AgregarComoAnterior(itOrd, Siguiente(itMaxAccesos))              ▷ O(1)
31:     EliminarSiguiente(itMaxAccesos)                                  ▷ O(1)

32: end while

33: s.linksPorCat[categoriaID − 1] ← listaOrdenada                       ▷ O(1)
34: res ← crearItLinks(listaOrdenada, f, s)                              ▷ O(1)
```

Cálculo de complejidad: $O(|c|) + 2O(n) + O(n^2) + 11O(1) = O(|c| + n^2)$

3.4.1. Algoritmos del Iterador

iCrearItLinks(**in** ls: `lista(infoLink)`, **in** $fecha$: `fecha`) $\rightarrow$ res : `itLinks`

```
1:  res.it ← crearIt(ls)                                                 ▷ O(1)
2:  res.ultAcceso ← fecha                                                ▷ O(1)
```

Cálculo de complejidad: $O(1)$

iSiguienteLink(**in** *itl* : `itLinks`) → *res* : `link`

1: *res* ← SIGUIENTE(*itl.it*).*link* ▷ O(1)

Cálculo de complejidad: $O(1)$

iSiguienteCategoria(**in** *itl* : `itLinks`) → *res* : `categoria`

1: *res* ← SIGUIENTE(*itl.it*).*categoria* ▷ O(1)

Cálculo de complejidad: $O(1)$

iSiguienteAccesosRecientes(**in** *itl* : `itLinks`) → *res* : `Nat`

1: *res* ← PUNTAJEDELLINK(*itl.s*, SIGUIENTE(*itl.it*), *itl.fecha*) ▷ 2O(1) = O(1)

Cálculo de complejidad: $O(1)$

iHaySiguiente?(**in** *itl* : `itLinks`) → *res* : `Bool`

1: *res* ← HAYSIGUIENTE?(*itl.it*) $\wedge_{\text{L}}$ PUNTAJEDELLINK(*itl.s*, SIGUIENTE(*itl.it*), *itl.utlAcceso*) > 0 ▷ 3O(1) = O(1)

Cálculo de complejidad: $O(1)$

iAvanzar(**in/out** *itl* : `itLinks`)

1: AVANZAR(*itl.it*)

Cálculo de complejidad: $O(1) = O(1)$

3.5. Servicios Usados

Los siguientes módulos deben cumplir los compromisos pedidos

- `Puntero(`α`)`

  & debe tener complejidad O(1).

- `abCat`

cantCategorias debe tener complejidad O(1).
ID debe tener complejidad O(|c|) donde c es la categoría de la que se desea saber el id.
padres debe tener complejidad O(|c|) donde c es la categoría de la que se desea conocer sus padres.

- `itAbCat`

hayMasPadres? debe tener complejidad O(1).
subir debe tener complejidad O(1).
actualID debe tener complejidad O(1).

- `diccString(`α`)`

Definir debe tener complejidad O(|k|) donde k es la clave.
Obtener debe tener complejidad O(|k|) donde k es la clave.
Vacio debe tener complejidad O(1).

- `Vector(`α`)`

Vacia debe tener complejidad O(1).
AgregarAtras debe tener complejidad O(n) sí y sólo sí se agregan n elementos de forma consecutiva.
• [•] debe tener complejidad O(1).

- `Lista Enlazada (`α`)`

Vacia debe tener complejidad O(1).
AgregarAtras debe tener complejidad O($copy(a)$), donde a es el tipo del elemento que va a ser colocado en la lista.
Longitud debe tener complejidad O(1).

- `itLista(`α`)`

crearIt debe tener complejidad O(1).
HaySiguiente? debe tener complejidad O(1).
Avanzar debe tener complejidad O(1).
Siguiente debe tener complejidad O(1).

- `itAbCat`

HayMasPadre? debe tener complejidad O(1).
Subir debe tener complejidad O(1).
ActualID debe tener complejidad O(1).

4. Módulo Iterador Unidireccional(α)

4.1. Interfaz

parámetro formal: α.

usa: `Bool, lista(`α`), itLista(`α`)`.

se explica con: `Iterador Unidireccional(`α`)`.

géneros: `iteradorUni(`α`)`

Operaciones básicas

CREARITERADOR(**in** l : `lista(`α`)`) $\rightarrow$ res : `iteradorUni(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} crearItUni(l)\}$
Complejidad: $\Theta(1)$
Descripción: Crea un iterador unidireccional a la lista de forma tal que al pedir actual se obtenga el primer elemento de l.

HAYMAS?(**in** it : `iteradorUni(`α`)`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} HayMas?(it)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve true sí y sólo sí en el iterador todavía quedan elementos por iterar.

SIGUIENTE(**in**/**out** it : `iteradorUni(`α`)`)
Pre $\equiv$ $\{hayMas?(it) \wedge it =_{\text{obs}} it_0\}$
Post $\equiv$ $\{it =_{\text{obs}} Avanzar(it_0)\}$
Complejidad: $\Theta(1)$
Descripción: Avanza el iterador a la siguiente posición.

ACTUAL(**in** it : `iteradorUni(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} Actual(it)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve el elemento al que está apuntando el iterador por referencia. Dicho elemento no debe ser modificado ya que alteraría la lista y eso no está permitido.

4.2. Representación

`iteradorUni(`α`)` se representa con `estr`

donde `estr` es `tupla(`*it*: `itLista(`α`) )`

Rep : estr $\longrightarrow$ bool

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

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

Abs(e) $\equiv$ it:iteradorUni(α) | Siguientes(it) $=_{\text{obs}}$ Siguientes(e.it)

Justificación

Para representar el iterador unidireccional de la lista, decidimos usar el iterador bidireccional que nos provee la lista enlazada y nos permite obtener, del iterador, las operaciones que necesitamos.

4.3. Algoritmos

iCrearIterador(**in** l: `lista(`α`)`) $\rightarrow$ *res* : `iteradorUni(`α`)`

1: *res.it* $\leftarrow$ CrearIt(l) ▷ $\Theta(1)$

Cálculo de complejidad: $\Theta(1)$

iHayMas?(**in** *iterador*: `iteradorUni(`α`)`) $\rightarrow$ *res* : `Bool`

1: *res* $\leftarrow$ HaySiguiente(*iterador.it*) ▷ $\Theta(1)$

Cálculo de complejidad: $\Theta(1)$

iSiguiente(**in/out** *iterador*: `iteradorUni(`α`)`)

1: Avanzar(*iterador.it*) ▷ $\Theta(1)$

Cálculo de complejidad: $\Theta(1)$

iActual(**in** *iterador*: `iteradorUni(`α`)`) $\rightarrow$ *res* : α

1: *res* $\leftarrow$ Siguiente(*iterador.it*) ▷ $\Theta(1)$

Cálculo de complejidad: $\Theta(1)$

4.4. Servicios Usados

Los siguientes módulos deben cumplir con los compromisos pedidos:

- `itLista(`α`)`

crearIt debe tener complejidad O(1).
HaySiguiente debe tener complejidad O(1).
Avanzar debe tener complejidad O(1).
Siguiente debe tener complejidad O(1).