

Algoritmos y Estructura de Datos II

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

Trabajo Práctico Diseño

Grupo 1

Integrante	LU	Correo electrónico
Bálsamo, Facundo	874/10	facundobalsamo@gmail.com
Lasso, Nicolás	763/10	lasso.nico@gmail.com
Rodriguez, Agustín	120/10	agustinrodriguez90@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo ArbolCategorias	4
1.1. Interfaz	4
1.2. Representación	6
1.2.1. Invariante de Representación	6
1.2.1.1. El Invariante Informalmente	6
1.2.1.2. El Invariante Formalmente	7
1.2.2. Función de Abstracción	8
1.2.2.1. Funciones auxiliares	8
1.3. Algoritmos	8
1.4. Analisis de complejidades	10
1.5. Iterador de Categorías	13
1.5.1. Interfaz	13
1.5.2. Representación	13
1.5.3. Invariante de Representación	14
1.5.3.1. El Invariante Formalmente	14
1.5.4. Función de Abstracción	14
1.5.5. Algoritmos	14
1.5.6. Analisis de complejidades	14
1.6. Iterador de Familia	15
1.6.1. Interfaz	15
1.6.2. Representación	16
1.6.3. Invariante de Representación	16
1.6.3.1. El Invariante Formalmente	16
1.6.4. Algoritmos	16
1.6.5. Analisis de complejidades	17
1.7. Iterador de Hijos	17
1.7.1. Interfaz	17
1.7.2. Representación	18
1.7.3. Invariante de Representación	18
1.7.3.1. El Invariante Formalmente	18
1.7.4. Función de Abstracción	19
1.7.5. Algoritmos	19
1.7.6. Analisis de complejidades	19
2. Módulo LinkLinkIt	21
2.1. Interfaz	21
2.2. Representación	23
2.2.1. Invariante de Representación	23
2.2.1.1. El Invariante Informalmente	23
2.2.1.2. El Invariante Formalmente	24
2.2.2. Función de Abstracción	24
2.2.2.1. Funciones auxiliares	25
2.3. Algoritmos	25
2.4. Analisis de complejidades	29
2.5. Iterador de Links	31
2.5.1. Interfaz	31
2.5.2. Representación	32
2.5.3. Invariante de Representación	32

2.5.3.1.	El Invariante Formalmente	32
2.5.4.	Función de Abstracción	32
2.5.5.	Algoritmos	32
2.5.6.	Análisis de complejidades	33
2.6.	Iterador de Punteros a DatosLink	33
2.6.1.	Interfaz	33
2.6.2.	Representación	34
2.6.3.	Invariante de Representación	35
2.6.3.1.	El Invariante Formalmente	35
2.6.4.	Función de Abstracción	35
2.6.5.	Algoritmos	35
2.6.6.	Análisis de complejidades	37
2.7.	Iterador de Accesos	39
2.7.1.	Representación	39
2.7.2.	Invariante de Representación	39
2.7.2.1.	El Invariante Formalmente	39
2.7.3.	Función de Abstracción	39
2.7.4.	Algoritmos	39
2.7.5.	Análisis de complejidades	40
3.	Módulo <code>diccTrie(clave,significado)</code>	41
3.1.	Interfaz	41
3.2.	Representación	41
3.2.1.	Invariante de Representación	42
3.2.1.1.	El Invariante Informalmente	42
3.2.1.2.	El Invariante Formalmente	42
3.2.2.	Función de Abstracción	42
3.2.2.1.	Funciones auxiliares	42
3.3.	Algoritmos	44
3.4.	Análisis de complejidades	45

1. Módulo ArbolCategorias

1.1. Interfaz

parámetros formales

géneros `acat`

se explica con: `ArbolDeCategorias`

Operaciones

`CATEGORIASAC(in ac: acat) → res: itCategorias`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} categorias(ac)\}$

Complejidad: $O(1)$

Aliasing: No se debe modificar nada de lo iterado por `res`.

`RAIZAC(in ac: acat) → res: Categoria`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} raiz(ac)\}$

Complejidad: $O(1)$

Aliasing: El nombre de la categoría raíz se pasa por referencia, no debe ser modificado.

`IDAC(in ac: acat, in c: Categoria) → res: nat`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} id(ac, c)\}$

Complejidad: $O(|c|)$

Aliasing: No tiene.

`ALTURACATAC(in ac: acat, in c: Categoria) → res: nat`

Pre $\equiv \{esta?(c, ac)\}$

Post $\equiv \{res =_{\text{obs}} alturaCategoria(ac, c)\}$

Complejidad: $O(|c|)$

Aliasing: No tiene.

`HIJOSAC(in ac: acat, in c: Categoria) → res: itHijos`

Pre $\equiv \{esta?(c, ac)\}$

Post $\equiv \{res =_{\text{obs}} hijos(ac, c)\}$

Complejidad: $O(|c|)$

Aliasing: No se debe modificar nada de lo iterado por `res`.

`PADREAC(in ac: acat, in c: Categoria) → res: Categoria`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} padre(ac, c)\}$

Complejidad: $O(|c|)$

Aliasing: El nombre de la categoría padre se pasa por referencia, no debe ser modificado.

ALTURAAC(**in** *ac*: **acat**) → *res*: **nat**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} altura(ac)\}$

Complejidad: $O(1)$

Aliasing: No tiene.

NUEVOAC(**in** *c*: **Categoria**) → *res*: **acat**

Pre $\equiv \{\neg vacia?(c)\}$

Post $\equiv \{res =_{\text{obs}} nuevo(c)\}$

Complejidad: $O(|c|)$

Aliasing: No tiene.

AGREGARAC(**in/out** *ac*: **acat**, **in** *c*: **categoria**, **in** *h*: **categoria**)

Pre $\equiv \{esta?(c, ac) \wedge \neg esta?(h, ac) \wedge \neg vacia?(h) \wedge ac_0 =_{\text{obs}} ac\}$

Post $\equiv \{ac =_{\text{obs}} agregar(ac_0, c, h)\}$

Complejidad: $O(|c| + |h|)$

Aliasing: No hay alias ya que no devuelve nada.

ESTA?(**in** *c*: **categoria**, **in** *ac*: **acat**) → *res*: **bool**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} esta?(c, ac)\}$

Complejidad: $O(|c|)$

Aliasing: No tiene.

ESSUBCATEGORIA(**in** *ac*: **acat**, **in** *c*: **categoria**, **in** *h*: **categoria**) → *res*: **bool**

Pre $\equiv \{esta?(c, ac) \wedge esta?(h, ac)\}$

Post $\equiv \{res =_{\text{obs}} esSubCategoria(ac, c, h)\}$

Complejidad: $O(|h| + |c| + alturaAC(ac))$

Aliasing: No tiene.

fin interfaz

1.2. Representación

ArbolCategorias **se representa con** *estrAC*, donde *estrAC* es tupla<
 raiz: puntero(datosCat),
 cantidad: nat,
 alturaMax: nat,
 familia: diccTrie(Categoria, puntero(datosCat)),
 categorias: Lista(datosCat)>

datosCat es tupla<
 categoria: Categoria,
 id: nat,
 altura: nat,
 hijos: Conj(puntero(datosCat)),
 padre: puntero(datosCat)>

Arbol de Categorias guarda en su estructura una Lista de datosCat(*categorias*), que cada uno

guarda todos los datos de una categoria.

Guardamos en un `dicTrie(familia)` para cada categoria, un puntero a su `datosCat` correspondiente de la lista `categorias` para acceder a esos datos en $O(\text{longitud de la categoria})$.

En `raiz` guardamos un puntero a `datosCat` de la categoria raiz del arbol para accederla en $O(1)$
`cantidad` es la cantidad de categorias que tiene el arbol y nos permite en $O(1)$ saber cual va a ser el id para una categoría que estemos agregando.

`alturaMax` es la altura del arbol de categorias.

1.2.1. Invariante de Representación

1.2.1.1. El Invariante Informalmente

1. Para cada clave de '*familia*' obtener el significado devolvera un puntero(`datosCat`) donde '*categoria*' es igual a la clave.
2. Toda clave de '*familia*' debera ser raiz o pertenecer a algun conjunto de punteros de '*hijos*' de alguna otra clave.
3. Todos los significados de '*familia*' apuntan a un nodo de '*categorias*' y cada nodo de '*categorias*' es significado de alguna clave de '*familia*'.
4. Todos los elementos de '*hijos*' de una clave de '*familia*', tendrá como '*padre*' a esa clave.
5. '*cantidad*' sera igual a la longitud de la lista '*categorias*'.
6. Cuando la clave es igual a '*raiz*' su '*altura*' e '*id*' es 1.
7. La '*altura*' de cada clave es menor o igual a '*alturaMax*' del sistema.
8. Existe una clave en la cual '*altura*' es igual a '*alturaMax*'.
9. Los '*hijos*' de una clave tienen '*altura*' igual a $1 + \text{'altura de la clave'}$.
10. Los '*id*' de cada clave deberan ser menor o igual a '*cant*'.
11. No hay '*id*' repetidos en '*familia*'.

1.2.1.2. El Invariante Formalmente

$\text{Rep} : \text{estrAC} \rightarrow \text{boolean}$

$(\forall ac : \text{estrAC}) \text{Rep}(ac) \equiv \text{true} \iff$

1. $(\forall c : \text{Categoria})(\text{def?}(c, e.familia)) \iff (*\text{obtener}(c, e.familia)).categoria = c \wedge_L$
2. $(\forall c_1 : \text{Categoria})(\text{def?}(c_1, e.familia)) \iff ((c_1 == e.raiz) \vee ((\exists c_2 : \text{Categoria})(\text{def?}(c_2, e.familia)) \wedge_L c_1 \in (*\text{obtener}(c_2, e.familia)).hijos)) \wedge_L$
3. $(\forall c : \text{Categoria})(\text{def?}(c, e.familia)) \iff (((\exists d : \text{datosCat})\text{esta?}(d, e.categorias) \wedge d.categoria == c) \wedge_L d == \text{obtener}(c, e.familia))) \wedge_L$
4. $(\forall c_1, c_2 : \text{string})(\text{def?}(c_1, e.familia)) \wedge (\text{def?}(c_2, e.familia)) \Rightarrow_L c_2 \in *((\text{obtener}(c_1, e.familia)).hijos) \iff ((*(\text{obtener}(c_2, e.familia))).padre).categoria = c_1 \wedge_L$
5. $e.cantidad = \text{longitud}(e.categorias) \wedge_L$

6. $(\forall c: \text{categoria})(\text{def?}(c, e.familia)) \wedge c = e.raiz \Rightarrow_L$
 $(*(\text{obtener}(c, e.familia))).altura = 1 \wedge (*(\text{obtener}(c, e.familia))).id = 1 \wedge_L$
7. $(\forall c: \text{Categoria})(\text{def?}(c, e.familia)) \Rightarrow_L (*(\text{obtener}(c, e.familia))).altura \leq e.alturaMax$
 \wedge_L
8. $(\exists c: \text{Categoria})(\text{def?}(c, e.familia)) \wedge_L (*(\text{obtener}(c, e.familia))).altura = e.alturaMax$
 \wedge_L
9. $(\forall c_1, c_2: \text{string})(\text{def?}(c_1, e.familia)) \wedge (\text{def?}(c_2, e.familia)) \wedge_L$
 $((\exists d: \text{datosCat})d \in (*(\text{obtener}(c_1, e.familia))).hijos \wedge d.categoria == c_2) \Rightarrow_L$
 $(*(\text{obtener}(c_2, e.familia))).altura = 1 + (*(\text{obtener}(c_1, e.familia))).altura \wedge_L$
10. $(\forall c: \text{Categoria})(\text{def?}(c, e.familia)) \Rightarrow_L (*(\text{obtener}(c, e.familia))).id \leq e.cant \wedge_L$
11. $(\forall c_1, c_2: \text{Categoria})(\text{def?}(c_1, e.familia)) \wedge (\text{def?}(c_2, e.familia)) \wedge c_1 \neq c_2 \Rightarrow_L$
 $(*(\text{obtener}(c_1, e.familia))).id \neq (*(\text{obtener}(c_2, e.familia))).id$

1.2.2. Función de Abstracción

$\text{Abs} : e : \text{estrAC} \rightarrow \text{acat}$

$\text{Rep}(e)$

$(\forall e : \text{estrAC}) \text{ Abs}(e) =_{\text{obs}} \text{ac} : \text{acat} \mid$

1. $\text{categorias}(\text{ac}) =_{\text{obs}} \text{todasLasCategorias}(e.\text{categorias}) \wedge_L$
2. $\text{raiz}(\text{ac}) =_{\text{obs}} (*e.\text{raiz}).\text{categoria} \wedge_L$
3. $(\forall c : \text{Categoria}) \text{esta?}(c, \text{ac}) \wedge c \neq \text{raiz}(\text{ac}) \Rightarrow_L$
 $\text{padre}(\text{ac}, c) = (*(\text{obtener}(c, e.\text{familia})).\text{padre}).\text{categoria} \wedge_L$
4. $(\forall c : \text{Categoria}) \text{esta?}(c, \text{ac}) \Rightarrow_L \text{id}(\text{ac}, c) = (*(\text{obtener}(c, e.\text{familia}))).\text{id}$

1.2.2.1. Funciones auxiliares

$\text{todasLasCategorias} : \text{secu}(\text{datosCat}) \rightarrow \text{conj}(\text{categoria})$

$\text{todasLasCategorias}(\text{cs}) \equiv \text{if vacia?}(\text{cs}) \text{ then}$
 \emptyset
 else
 $\text{Ag}((\text{prim}(\text{cs})).\text{categoria}, \text{todasLasCategorias}(\text{fin}(\text{cs})))$
 fi

1.3. Algoritmos

Algoritmo 1 iCategoriasAC

```
1: function ICATEGORIASAC(in  $\text{ac} : \text{estrAC}$ )  $\rightarrow$   $\text{res} : \text{itCategorias}$   
2:    $\text{res} \leftarrow \text{crearItCategorias}(\text{ac})$  //O(1)  
3: end function  
Complejidad: O(1)
```

Algoritmo 2 iRaizAC

```
1: function IRAIZ(in  $\text{ac} : \text{estrAC}$ )  $\rightarrow$   $\text{res} : \text{Categoria}$   
2:    $\text{res} \leftarrow (*\text{ac.raiz}).\text{categoria}$  //O(1)  
3: end function  
Complejidad: O(1)
```

Algoritmo 3 iDameCantidad

```
1: function IDAMECANTIDAD(in  $\text{ac} : \text{estrAC}$ )  $\rightarrow$   $\text{res} : \text{nat}$   
2:    $\text{res} \leftarrow \text{ac.cantidad}$  //O(1)  
3: end function  
Complejidad: O(1)
```

Algoritmo 4 iIdAC

```
1: function IID(in  $\text{ac} : \text{estrAC}$ , in  $c : \text{Categoria}$ )  $\rightarrow$   $\text{res} : \text{nat}$   
2:    $\text{res} \leftarrow ((*\text{obtener}(c, \text{ac.familia})).\text{id})$  //O(|c|)  
3: end function  
Complejidad: O(|c|)
```

Algoritmo 5 iAlturaCatAC

```
1: function IALTURACATAC(in ac: estrAC, in c: Categoria) → res: nat  
2:   res ← (*obtener(c, ac.familia)).altura //O(|c|)  
3: end function
```

Complejidad: $O(|c|)$

Algoritmo 6 iHijosAC

```
1: function IHIJOSAC(in ac: estrAC, in c: Categoria) → res: itHijos  
2:   res ← crearItHijos(ac, c) //O(|c|)  
3: end function
```

Complejidad: $O(|c|)$

Algoritmo 7 iPadreAC

```
1: function IPADREAC(in ac: estrAC, in c: Categoria) → res: Categoria  
2:   res ← (*(*obtener(c, ac.familia)).padre).categoria //O(|c|)  
3: end function
```

Complejidad: $O(|c|)$

Algoritmo 8 iAlturaAC

```
1: function IALTURAAC(in ac: estrAC) → res: nat  
2:   res ← ac.alturaMax //O(1)  
3: end function
```

Complejidad: $O(1)$

Algoritmo 9 iNuevoAC

```
1: function INUEVOAC(in c: Categoria) → res: estrAC  
2:   res.cantidad ← 1 //O(1)  
3:   datosCat tuplaA //O(1)  
4:   tuplaA ← tupla(c, 1, 1, vacio(), Null) //O(|c|)  
5:   puntero(datosCat) punt ← &tuplaA //O(1)  
6:   res.raiz ← punt //O(1)  
7:   res.alturaMax ← 1 //O(1)  
8:   definir(c, punt, res.familia) //O(|c|)  
9:   agregarAtras(tuplaA, res.categorias) //O(1)  
10: end function
```

Complejidad: $O(|c|)$

Algoritmo 10 iAgregarAC

```
1: function IAGREGARAC(in/out ac: estrAC, in c: Categoria, in h: Categoria)
2:   puntero(datosCat) puntPadre ← obtener(c, ac.familia) //O(|c|)
3:   if (*puntPadre).altura == ac.alturaMax then //O(1)
4:     ac.alturaMax++ //O(1)
5:   end if
6:   datosCat tuplaA ← (h, ac.cantidad+1, (*puntPadre).altura+1, vacio(), puntPadre) //O(|h|)
7:   puntero(datosCat) punt ← &tuplaA //O(1)
8:   Agregar((*puntPadre).hijos, punt) //O(1)
9:   definir(h, punt, ac.familia) //O(|h|)
10:  ac.cantidad++ //O(1)
11:  agregarAtras(tuplaA, ac.categorias) //O(1)
12: end function
```

Complejidad: $O(|c| + |h|)$

Algoritmo 11 iEsta?

```
1: function IESTA?(in ac: estrAC, in c: Categoria) → res: bool
2:   res ← def?(c, ac.familia) //O(|c|)
3: end function
```

Complejidad: $O(|c|)$

Algoritmo 12 iEsSubCategoria

```
1: function IESSUBCATEGORIA(in ac: estrAC, in c: Categoria, in h: Categoria) → res: bool
2:   res ← false //O(1)
3:   if h == c then //O(|h|)
4:     res ← true //O(1)
5:   else
6:     if h == raizAC(ac) then //O(|h|)
7:       res ← false //O(1)
8:     else
9:       puntero(datosCat) actual ← (*obtener(h, ac.familia)).padre //O(|h|)
10:      puntero(datosCat) puntC ← (*obtener(c, ac.familia)) //O(|c|)
11:      while res == false ∧ actual ≠ NULL do //O(alturaAC(ac))
12:        if puntC.Id == actual.Id then //O(1)
13:          res ← true //O(1)
14:        else
15:          actual ← (*actual).padre //O(1)
16:        end if
17:      end while
18:    end if
19:  end if
20: end function
```

Complejidad: $O(|h| + |c| + \text{alturaAC}(\text{ac}))$

1.4. Analisis de complejidades

1. iCategoriasAC

Se devuelve un iterador de la lista **categorias** del arbol de categorias en $O(1)$. El iterador muestra sólo los nombres de las categorías.

Orden Total: $O(1)$

2. **iRaiz**

Se devuelve una referencia al nombre de la categoria raiz del arbol de categorias en $O(1)$.

Orden Total: $O(1)$

3. **idameCantidad**

Se devuelve en $O(1)$ el natural almacenado en el campo cantidad del arbol de categorias.

Orden Total: $O(1)$

4. **iIdAC**

Dada la categoria c , se obtiene en $O(|c|)$ el **datosCat** de dicha categoría y en $O(1)$ se devuelve el id que tiene el **datosCat** obtenido.

Orden Total: $O(|c|)$

5. **iAlturaCatAC**

Dada la categoria c , se obtiene en $O(|c|)$ el **datosCat** de dicha categoría y en $O(1)$ se devuelve la altura que tiene el **datosCat** obtenido.

Orden Total: $O(|c|)$

6. **iHijosAC**

Dada la categoria c , se obtiene en $O(|c|)$ el **datosCat** de dicha categoría y en $O(1)$ se devuelve un iterador al conjunto **hijos** del **datosCat** obtenido.

Orden Total: $O(|c|)$

7. **iPadreAC**

Dada la categoria c , se obtiene en $O(|c|)$ el **datosCat** de dicha categoría y en $O(1)$ se devuelve por referencia en $O(1)$ el nombre de la categoria del puntero **padre** que tiene el **datosCat** obtenido.

Orden Total: $O(|c|)$

8. **iAlturaAC**

Devuelve en $O(1)$ la **alturaMax** del arbol de categorias.

Orden Total: $O(1)$

9. **iNuevoAC**

A **res.cantidad** le asignamos 1, que tarda $O(1)$. Creamos una nueva variable **tuplaA**, que es **datosCat**. Esto tarda $O(1)$.

Creamos la variable **punt**, que es un puntero a **datosCat** y le asignamos la referencia de **tuplaA**. Y esto tarda $O(1)$. A **tuplaA** le asignamos una nueva tupla **datosCat**, que en uno de sus componentes es el string c , y copiarse tarda $O(|c|)$. Los demas componentes de la tupla tardan en copiarse $O(1)$.

A **res.raiz** le asignamos **punt**, y tarda $O(1)$. A **res.alturaMax** le asignamos 1, y tarda $O(1)$. A **res.familia** le asignamos el **diccTrie** que nos da la operacion **definir**, a la cual le pasamos como clave el string c . Entonces **definir** tarda $O(|c|)$.

A res.categorias le asignamos la lista que nos da la operacion AgregarAtras, que tarda $O(1)$.

Orden Total: $O(1)+O(1)+O(1)+O(|c|)+O(1)+O(1)+O(|c|)+O(1) = O(|c|)$

10. iAgregarAC

Obtenemos un puntero de datosCat de la categoria c usando la operacion obtener del diccTrie ac.familia, y lo asignamos a la variable puntPadre. Esto tarda $O(|c|)$.

Comparamos la altura de la tupla que apunta puntPadre con ac.alturaMax, y esto tarda $O(1)$. En caso que valga la guarda del if hacemos una suma y una asignacion, que cuesta $O(1)$.

Luego creamos y asignamos una tupla de datosCat tuplaA, que se le asigna una tupla con valores que tardan $O(1)$ en copiarse, excepto por la categoria h que es string. Entonces la asignacion y creacion de esa tupla tarda $O(|h|)$.

Creamos la variable punt que es un puntero a datosCat, y le asignamos la referencia de tuplaA. Esto tarda $O(1)$. Agregamos al conjunto de punteros hijos que apunta puntPadre, el puntero punt, que tarda $O(1)$. Definimos la clave h, con el significado punt al diccTrie ac.familia. Esto tarda $O(|h|)$.

Incrementamos ac.cantidad, tardando $O(1)$. Finalmente agregamos atras tuplaA a la lista ac.categorias. Esto tarda $O(1)$

Orden Total: $O(|c|)+O(1)+O(1)+O(|h|)+O(1)+O(1)+O(|h|)+O(1)+O(1) = O(|c| + |h|)$

11. iEsta?

Para ver si una categoria c esta en nuestro arbolCategorias, vemos si esta definida la clave c en el diccTrie ac.familia. Y esto tarda $O(|c|)$.

Orden Total: $O(|c|)$

12. iEsSubCategoria

Le asignamos a res un valor booleano igual a false, demorando $O(1)$. Comparamos las dos categorias si son iguales o no. Demorando $O(|h|)$. En caso afirmativo cambiamos el valor de res por true, demorando $O(1)$.

En caso negativo, consultamos si h es igual a raizAC(ac) demorando $O(|h|)$, en caso positivo le asignamos a res el valor false, tardando $O(1)$. En caso negativo: creamos un puntero a datosCat denominado actual al cual le asignamos la tupla obtenida por la operacion obtener del diccTrie pasandole la categoria h y pidiendo padre de la tupla obtenida por esta operacion, esto demora $O(|h|)$. Creamos un puntero a datosCat denominado puntC al cual le asignamos la tupla obtenida por la operacion obtener del diccTrie pasandole la categoria c y pidiendo padre de la tupla obtenida por esta operacion, esto demora $O(|c|)$. Luego, se ingresa a un ciclo con la condicion de que res sea igual a false y actual distinto de NULL. Se compara puntC con actual. En caso afirmativo se asigna a res el valor true, demorando $O(1)$, en caso negativo, se modifica actual asignandole el puntero a padre de la tupla a la que estaba apuntando anteriormente. Luego de realizar alturaAC(ac) iteraciones se sale del ciclo.

Orden Total:

$O(1)+O(|h|)+O(1)+O(|h|)+O(1)+O(|h|)+O(|c|)+(alturaAC(ac)*(O(1)+O(1)+O(1))) =$
 $O(|h|+|c|+alturaAC(ac))$

1.5. Iterador de Categorías

1.5.1. Interfaz

parámetros formales

géneros `itCategorías`

se explica con: Iterador Bidireccional

Operaciones

`CREARITCATEGOTIAS(in ac: acat) → res: itCategorías`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{crearIt}(ac.categorías)\}$

Complejidad: $O(1)$

Aliasing: No debe modificarse ningún elemento iterado por `res`.

`HAYSIGUIENTE?(in it: itCategorías) → res: bool`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{haySiguiente?}(it)\}$

Complejidad: $O(1)$

Aliasing: No tiene.

`SIGUIENTE(in it: itCategorías) → res: Categoría`

Pre $\equiv \{\text{haySiguiente?}(it)\}$

Post $\equiv \{res =_{\text{obs}} \pi_1(\text{siguiente}(it))\}$

Complejidad: $O(1)$

Aliasing: La categoría se pasa por referencia, no debe ser modificada.

`AVANZAR(in/out it: itCategorías)`

Pre $\equiv \{it =_{\text{obs}} it_0 \wedge \text{haySiguiente?}(it)\}$

Post $\equiv \{it =_{\text{obs}} \text{avanzar}(it_0)\}$

Complejidad: $O(1)$

Aliasing: No tiene.

fin interfaz

1.5.2. Representación

`itCategorías` se representa con `itLista(datosCat)`

`datosCat` es tupla<
 `categoría: Categoría`,
 `id: nat`,
 `altura: nat`,
 `hijos: Conj(puntero(datosCat))`,
 `padre: puntero(datosCat)`>

`itCategorías` es un iterador de lista común. Sus complejidades nos alcanzan para iterar una

Lista(datosCat).

1.5.3. Invariante de Representación

1.5.3.1. El Invariante Formalmente

$\text{Rep} : \text{estrITC} \rightarrow \text{boolean}$

$(\forall it: \text{estrITC}) \text{Rep}(it) \equiv \text{true}$

1.5.4. Función de Abstracción

$\text{Abs} : e: \text{estrITC} \rightarrow itBi(\alpha) \quad \text{Rep}(e)$

$(\forall e: \text{estrITC}) \text{Abs}(e) =_{\text{obs}} it: itBi(\alpha) \mid$

1. $\text{anteriores}(e.\text{iterador}) =_{\text{obs}} \text{anteriores}(it) \wedge$
 $\text{siguientes}(e.\text{iterador}) =_{\text{obs}} \text{siguientes}(it)$

1.5.5. Algoritmos

Algoritmo 13 iCrearItCategorias

```
1: function ICREARITCATEGORIAS(in  $ac: \text{estrAC}$ )  $\rightarrow res: \text{estrITC}$ 
2:    $res \leftarrow \text{crearIt}(ac.\text{categorias})$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 14 iHaySiguiente?

```
1: function IHAYSIGUIENTE?(in  $e: \text{estrITC}$ )  $\rightarrow res: \text{bool}$ 
2:    $res \leftarrow \text{haySiguiente}(e)$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 15 iSiguiente

```
1: function ISIGUIENTE(in  $e: \text{estrITC}$ )  $\rightarrow res: \text{Categoria}$ 
2:    $res \leftarrow (\text{siguiente}(e)).\text{categoria}$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 16 iAvanzar

```
1: function IAVANZAR(in/out  $e: \text{estrITC}$ )
2:    $\text{avanzar}(e)$  //O(1)
3: end function
Complejidad: O(1)
```

1.5.6. Analisis de complejidades

1. iCrearItCategorias

Crea un `itCategorias` con la lista del árbol de categorías que se pasa como parámetro y se la asigna a `res`, esto demora $O(1)$.

Orden Total: $O(1)$

2. `iHaySiguiente?`

Se llama a `HaySiguiente` del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

3. `iSiguiente`

Se llama a `Siguiente` del Iterador de Lista en $O(1)$. Se devuelve una referencia al valor categoría de la tupla `DatosCat`.

Orden Total: $O(1)$

4. `iAvanzar`

Se llama a `Avanzar` del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

1.6. Iterador de Familia

1.6.1. Interfaz

parámetros formales

géneros `itFamilia`

Operaciones

`CREARITFAMILIA(in ac: estrAC, in c: Categoria) → res: itFamilia`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{obtener}(ac, c)\}$

Complejidad: $O(|c|)$

Aliasing: Los elementos iterados por `res` no deben ser modificados.

`HAYSIGUIENTE?(in it: itFamilia) → res: bool`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} (it \neq NULL)\}$

Complejidad: $O(1)$

Aliasing: No tiene.

`SIGUIENTECAT(in it: itFamilia) → res: Categoria`

Pre $\equiv \{\text{haySiguiente?}(it)\}$

Post $\equiv \{res =_{\text{obs}} (*it).catgoria\}$

Complejidad: $O(1)$

Aliasing: la Categoría se pasa por referencia y no debe ser modificado.

`SIGUIENTEID(in it: itFamilia) → res: int`

Pre $\equiv \{\text{haySiguiente?}(it)\}$

Post $\equiv \{res =_{\text{obs}} (*it).id\}$

Complejidad: $O(1)$

Aliasing: El ID se pasa por referencia y no debe ser modificado.

AVANZAR(**in/out** *it*: itFamilia)
Pre $\equiv \{it =_{\text{obs}} it_0 \wedge \text{haySiguiente?}(it)\}$
Post $\equiv \{it =_{\text{obs}} (*it_0).padre\}$
Complejidad: $O(1)$
Aliasing: No tiene.

fin interfaz

1.6.2. Representación

itFamilia se representa con puntero(DatosCat)

datosCat es tupla<
categoria: Categoria,
id: nat,
altura: nat,
hijos: Conj(puntero(datosCat)),
padre: puntero(datosCat)>

itFamilia es un iterador de puntero a datoscat que al hacer siguiente va al puntero datoscat padre. Al manejarse con punteros sus complejidades son $O(1)$.

1.6.3. Invariante de Representación

1.6.3.1. El Invariante Formalmente

$\text{Rep} : \text{estrITF} \rightarrow \text{boolean}$
 $(\forall it : \text{estrITF}) \text{Rep}(it) \equiv \text{true}$

1.6.4. Algoritmos

Algoritmo 17 iCrearItFamilia

```

1: function ICREARITFAMILIA(in ac: estrAC,in c: Categoria)→ res: estrITA
2:   res  $\leftarrow$  obtener(ac.familia,c)                                     //O(|c|)
3: end function
Complejidad:  $O(|c|)$ 

```

Algoritmo 18 iHaySiguiente?

```

1: function IHAYSIGUIENTE?(in e: estrITF)→ res: bool
2:   res  $\leftarrow e \neq \text{NULL}$                                            //O(1)
3: end function
Complejidad:  $O(1)$ 

```

Algoritmo 19 iSiguienteCat

```

1: function ISIGUIENTE(in e: estrITF)→ res: Categoria
2:   res  $\leftarrow (*e).categoria$                                          //O(1)
3: end function
Complejidad:  $O(1)$ 

```

Algoritmo 20 iSiguienteId

```
1: function ISIGUIENTE(in  $e$ : estrITF)  $\rightarrow$   $res$ : int  
2:    $res \leftarrow (*e).id$  //O(1)  
3: end function
```

Complejidad: $O(1)$

Algoritmo 21 iAvanzar

```
1: function IAVANZAR(in/out  $e$ : estrITF)  
2:    $e \leftarrow (*e).padre$  //O(1)  
3: end function
```

Complejidad: $O(1)$

1.6.5. Análisis de complejidades

1. iCrearItFamilia

Crea un itFamilia obteniendo en $O(|c|)$ los datos de esa categoría y asignandoselo a si mismo en $O(1)$.

Orden Total: $O(1)$

2. iHaySiguiente?

Chequea que el puntero no sea Null.

Orden Total: $O(1)$

3. iSiguienteCat

Devuelve en $O(1)$ una referencia a la Categoría del puntero que tiene almacenado el iterador.

Orden Total: $O(1)$

4. iSiguienteId

Devuelve en $O(1)$ el Id del puntero que tiene almacenado el iterador.

Orden Total: $O(1)$

5. iAvanzar

en $O(1)$ el iterador se guarda el puntero Padre del puntero que tenia guardado.

Orden Total: $O(1)$

1.7. Iterador de Hijos

1.7.1. Interfaz

parámetros formales

géneros itHijos

se explica con: Iterador Bidireccional

Operaciones

CREARITHIJOS(**in** *ac*: **estrAC**, **in** *c*: **Categoria**) → *res*: **itHijos**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} crearIt((\ast Obtener(ac, c)).hijos)\}$

Complejidad: $O(|c|)$

Aliasing: No se pueden modificar los datosCat del iterador.

HAYSIGUIENTE?(**in** *it*: **itHijos**) → *res*: **bool**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} haySiguiente?(it)\}$

Complejidad: $O(1)$

Aliasing: No tiene.

SIGUIENTE(**in** *it*: **itHijos**) → *res*: **Categoria**

Pre $\equiv \{haySiguiente?(it)\}$

Post $\equiv \{res =_{\text{obs}} \pi_1(\ast siguiente(it))\}$

Complejidad: $O(1)$

Aliasing: *res* no es modificable.

AVANZAR(**in/out** *it*: **itHijos**)

Pre $\equiv \{it =_{\text{obs}} it_0 \wedge haySiguiente?(it)\}$

Post $\equiv \{it =_{\text{obs}} avanzar(it_0)\}$

Complejidad: $O(1)$

Aliasing: No tiene.

fin interfaz

1.7.2. Representación

itHijos se representa con itConj(puntero(datosCat))

datosCat es tupla<
 categoria: **Categoria**,
 id: **nat**,
 altura: **nat**,
 hijos: **Conj**(puntero(datosCat)),
 padre: puntero(datosCat)>

itHijos es un iterador de conjunto. Sus complejidades nos alcanzan para iterar un Conj(puntero(datosCat)).

1.7.3. Invariante de Representación

1.7.3.1. El Invariante Formalmente

$Rep : \text{estrITH} \rightarrow \text{boolean}$

$(\forall it : \text{estrITH}) \text{Rep}(it) \equiv \text{true}$

1.7.4. Función de Abstracción

$Abs : e : \text{estrITC} \rightarrow \text{itBi}(\alpha)$

$\text{Rep}(e)$

$(\forall e : \text{estrITC}) \text{ Abs}(e) =_{\text{obs}} \text{it} : \text{itBi}(\alpha) \mid$

1. $\text{anteriores}(e.\text{iterador}) =_{\text{obs}} \text{anteriores}(\text{it}) \wedge$
 $\text{siguientes}(e.\text{iterador}) =_{\text{obs}} \text{siguientes}(\text{it})$

1.7.5. Algoritmos

Algoritmo 22 iCrearItHijos

```
1: function ICLEARITHIJOS(in  $ac : \text{acat}$ , in  $c : \text{Categoria}$ )  $\rightarrow res : \text{estrITH}$ 
2:    $res \leftarrow \text{crearIt}((\text{*obtener}(ac.familia, c)).\text{hijos})$  //O(|c|)
3: end function
Complejidad: O(1)
```

Algoritmo 23 iHaySiguiente?

```
1: function IHAYSIGUIENTE?(in  $e : \text{estrITH}$ )  $\rightarrow res : \text{bool}$ 
2:    $res \leftarrow \text{haySiguiente}(e)$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 24 iSiguiente

```
1: function ISIGUIENTE(in  $e : \text{estrITH}$ )  $\rightarrow res : \text{Categoria}$ 
2:    $res \leftarrow (\text{*siguiente}(e)).\text{categoria}$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 25 iAvanzar

```
1: function IAVANZAR(in/out  $e : \text{estrITH}$ )
2:    $\text{avanzar}(e)$  //O(1)
3: end function
Complejidad: O(1)
```

1.7.6. Análisis de complejidades

1. iCrearItHijos

Crea un itHijos con el conjunto Hijos del puntero que se obtiene en $O(|c|)$ del diccionario Familia con la categoria pasada por parametro.

Orden Total: O(1)

2. iHaySiguiente?

Se llama a HaySiguiente del Iterador de Conjunto en $O(1)$.

Orden Total: O(1)

3. **iSiguiente**

Se llama a Siguiente del Iterador de Conjunto en $O(1)$. A eso se le aplica la operacion dameCat que también cuesta $O(1)$ y se devuelve una referencia a la categoria resultante.

Orden Total: $O(1)$

4. **iAvanzar**

Se llama a Avanzar del Iterador de Conjunto en $O(1)$.

Orden Total: $O(1)$

2. Módulo LinkLinkIt

2.1. Interfaz

parámetros formales

géneros **linkLinkIt**

se explica con: TAD **linkLinkIt**

Operaciones

DAMEACAT(**in** *lli*: **linkLinkIt**) → *res*: **acat**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} lli.arbolCategorias\}$

Complejidad: $O(1)$

Aliasing: *res* es una referencia a *lli.arbolCategorias*, no debe modificarse.

FECHAACTUAL(**in** *lli*: **linkLinkIt**) → *res*: **Fecha**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} fechaActual(lli)\}$

Complejidad: $O(1)$

Aliasing: No tiene

LINKS(**in** *lli*: **linkLinkIt**) → *res*: **itLinks**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} links(lli)\}$

Complejidad: $O(1)$

Aliasing: No deben modificarse los elementos iterados por *res*.

CATEGORIALINK(**in** *lli*: **linkLinkIt**, **in** *l*: **Link**) → *res*: **Categoria**

Pre $\equiv \{l \in links(lli)\}$

Post $\equiv \{res =_{\text{obs}} categoriaLink(lli, l)\}$

Complejidad: $O(|l|)$

Aliasing: La categoria se devuelve por referencia, no debe modificarse.

FECHAULTIMOACCESO(**in** *lli*: **linkLinkIt**, **in** *l*: **Link**) → *res*: **Fecha**

Pre $\equiv \{l \in links(lli)\}$

Post $\equiv \{res =_{\text{obs}} fechaUltimoAcceso(lli, l)\}$

Complejidad: $O(|l|)$

Aliasing: No tiene

ACCESOSRECIENTESDIA(**in** *lli*: **linkLinkIt**, **in** *l*: **Link**, **in** *f*: **Fecha**) → *res*: **nat**

Pre $\equiv \{l \in links(lli)\}$

Post $\equiv \{res =_{\text{obs}} accesosRecientesDia(lli, l, f)\}$

Complejidad: $O(|l|)$

Aliasing: No tiene

INICIARLLI(**in** *ac*: *acat*) → *res*: **linkLinkIt**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} iniciar(ac)\}$

Complejidad: $O(\#categorias(ac))$

Aliasing: No tiene.

NUEVOLINKLLI(**in/out** *lli*: **linkLinkIt**, **in** *l*: **Link**, **in** *c*: **Categoria**)

Pre $\equiv \{c \in categorias(lli) \wedge l \notin links(lli) \wedge \neg vacia?(l) \wedge lli_0 = lli\}$

Post $\equiv \{lli = nuevoLink(lli_0, l, c)\}$

Complejidad: $O(|l| + |c| + altura(lli.arbolCategorias))$

Aliasing: No hay alias ya que no devuelve nada.

ACCEDERLLI(**in/out** *lli*: **linkLinkIt**, **in** *l*: **Link**, **in** *f*: **Fecha**)

Pre $\equiv \{l \in links(lli) \wedge f \geq fechaActual(lli) \wedge lli_0 = lli\}$

Post $\equiv \{lli = acceso(lli_0, l, f)\}$

Complejidad: $O(|l|)$

Aliasing: No hay alias ya que no devuelve nada.

CANTLINKS(**in** *lli*: **linkLinkIt**, **in** *c*: **Categoria**) → *res*: **nat**

Pre $\equiv \{c \in categorias(lli)\}$

Post $\equiv \{res =_{\text{obs}} cantLinks(lli, c)\}$

Complejidad: $O(|c|)$

Aliasing: No tiene.

LINKSORDENADOSPORACCESOS(**in** *lli*: **linkLinkIt**, **in** *c*: **Categoria**) → *res*: **itPuntLinks**

Pre $\equiv \{c \in categorias(lli)\}$

Post $\equiv \{res =_{\text{obs}} linksOrdenadosPorAccesos(lli, c)\}$

Complejidad: $O((longitud(lli.arrayCatLinks[id]))^2 + |c|)$

Aliasing: Se devuelve un iterador a los links relacionados con esa categoría. No debe ser modificado.

ESRECIENTE?(**in** *lli*: **linkLinkIt**, **in** *l*: **Link**, **in** *f*: **Fecha**) → *res*: **bool**

Pre $\equiv \{l \in links(lli)\}$

Post $\equiv \{res =_{\text{obs}} esReciente?(s, l, f)\}$

Complejidad: $O(|l|)$

Aliasing: No tiene.

fin interfaz

2.2. Representación

```
LinkLinkIt se representa con estrLLI, donde estrLLI es tupla<
    arbolCategorias: acat,
    actual: Fecha,
    linkInfo: diccTrie(Link, puntero(datosLink)),
    listaLink: Lista(datosLink),
    arrayCatLinks: arreglo(linksFamilia)>

datosLink es tupla<
    link: Link,
    catDLink: Categoria,
    accesosRecientes: Lista(acceso),
    cantAccesosRecientes: nat>

acceso es tupla<
    dia: Fecha,
    cantAccesos: nat>

linksFamilia es Lista(puntero(datosLink))
```

Un linkLinkIt guarda en su estructura el arbol de categorias con el que fue creado. La fecha actual, para poder accederla en $O(1)$.

Tiene también una lista de datosLink(*listaLink*), que guarda un datosLink para cada Link con sus datos: nombre(*link*), una referencia al nombre de su categoría relacionada(*catDLink*) para accederla en $O(1)$, la cantidad de accesos recientes(*cantAccesosRecientes*) y su lista de accesos recientes, es decir sus ultimos tres días(*accesosRecientes*).

En el diccTrie *linkInfo*, tomando como claves los nombres de los links, guardamos un puntero al datoLink correspondiente de *listaLink*, para poder acceder a esos datos en $O(\text{longitud del link})$.

En el arreglo *arrayCatLink* se guarda en cada posición, la lista de links relacionados para la categoria cuyo id es esa posicion-1 (Incluye a los links de las categorias hijas.).

2.2.1. Invariante de Representación

2.2.1.1. El Invariante Informalmente

1. Para todo '*link*' que exista en '*linkInfo*' la '*catDLink*' de la tupla apuntada en el significado debiera existir en '*arbolCategorias*'.
2. Para todo '*link*' que exista en '*linkInfo*', todos los '*dia*' de la lista '*accesosRecientes*' deberan ser menor o igual a *actual*, estan ordenados, no hay dias repetidos y la longitud de la lista es menor o igual a 3.
3. Para todo '*link*' que exista en '*linkInfo*' su significado deberá existir en '*listaLinks*' y viceversa.
4. Para todo '*link*' que exista en '*linkInfo*' su significado deberá aparecer en '*arrayCatLinks*' en la posicion igual al id de '*catDLink*' y en las posiciones de los predecesores de esa categoria y en ninguna otra.
5. No hay 2 claves que existan en '*linkInfo*' y devuelvan el mismo significado.
6. No existen '*link*' repetidos en las tuplas de '*listaLinks*'.

7. No hay elementos repetidos en ninguna lista '*linksFamilia*'.
8. Para todo '*link*' que exista en '*linkInfo*', '*cantAccesosRecientes*' es igual a la suma de '*cantAccesos*' de cada elemento de la lista '*accesosRecientes*'

2.2.1.2. El Invariante Formalmente

Rep : *estrLLI* → boolean

($\forall lli: \text{estrLLI}$) Rep(*lli*) \equiv true \iff

1. ($\forall l: \text{Link}$)(*def?*(*l*, *lli.linkInfo*)) \Rightarrow_L
 $(\text{*obtener}(l, lli.linkInfo)).catDLink \in \text{todasLasCategorias}(lli.arbolCategorias.categorias)$
 \wedge_L
2. ($\forall l: \text{Link}$)(*def?*(*l*, *lli.linkInfo*)) \Rightarrow_L
 $\text{long}(\text{*obtener}(l, lli.linkInfo).accesosRecientes) \leq 3 \wedge$
 $\text{accesoOrdenadoNoRepetido}(\text{*obtener}(l, lli.linkInfo).accesosRecientes) \wedge_L$
 $\text{fechasCorrectas}(lli.actual, (\text{*obtener}(l, lli.linkInfo).accesosRecientes)) \wedge_L$
3. ($\forall l: \text{Link}$)(*def?*(*l*, *e.linkInfo*) \Leftrightarrow
 $((\exists d: \text{datosLink}) \text{esta?}(d, e.listaLinks) \wedge d.link == l) \wedge_L d == \text{obtener}(l, e.linkInfo))$
 \wedge_L
4. ($\forall l: \text{Link}$)(*def?*(*l*, *lli.linkInfo*)) \Rightarrow_L
 $(\forall c: \text{Categoria}) c \in \text{todasLasCategorias}(lli.arbolCategorias.categorias) \Rightarrow_L$
 $(\text{esta?}(\text{obtener}(l, lli.linkInfo), \text{arrayCatLinks}[\text{idAC}(lli.arbolCategorias, c)]) \Leftrightarrow$
 $\text{esSubCategoria}(lli.arbolCategorias, c, (\text{*obtener}(l, lli.linkInfo).catDLink))) \wedge_L$
5. ($\forall l, l': \text{Link}$) $l \neq l' \wedge (\text{def?}(l, lli.linkInfo) \wedge (\text{def?}(l', lli.linkInfo)) \Rightarrow_L$
 $(\text{*obtener}(l, lli.linkInfo) \neq (\text{*obtener}(l', lli.linkInfo)) \wedge_L$
6. ($\forall i, i': \text{nat}$) $i < \text{long}(lli.listaLinks) \wedge i' < \text{long}(lli.listaLinks) \Rightarrow_L$
 $lli.listaLinks_i.link = lli.listaLinks_{i'}.link \Leftrightarrow i = i' \wedge_L$
7. ($\forall i: \text{nat}$) $i < \text{tam}(lli.arrayCatLinks) \Rightarrow_L \text{sinRepetidos}(\text{arrayCatLinks}[i]) \wedge_L$
8. ($\forall l: \text{Link}$)(*def?*(*l*, *lli.linkInfo*)) \Rightarrow_L
 $(\text{*obtener}(l, lli.linkInfo).cantAccesosRecientes ==$
 $\text{cantidadDeAccesos}(\text{*obtener}(l, lli.linkInfo).accesosRecientes)$

2.2.2. Función de Abstracción

Abs : *e*: *estrLLI* → linkLinkIt

Rep(*e*)

($\forall e: \text{estrLLI}$) Abs(*e*) =_{obs} *lli*: linkLinkIt |

1. *categorias*(*lli*) = *categorias*(*e.arbolCategorias*) \wedge
2. *links*(*lli*) = *todosLosLinks*(*e.listaLinks*) \wedge_L
3. ($\forall l: \text{Link}$) *def?*(*l*, *e.linkInfo*) \Rightarrow_L
 $\text{categoriaLink}(lli, l) = (\text{*obtener}(l, e.linkInfo).catDLink) \wedge$
4. *fechaActual*(*lli*) = *e.actual* \wedge
5. ($\forall l: \text{Link}$) $l \in \text{links}(e) \Rightarrow_L$
 $\text{fechaUltimoAcceso}(lli, l) = (\text{ultimo}(\text{*obtener}(l, e.linkInfo).accesosRecientes)).dia$
 \wedge
6. ($\forall l: \text{Link}$)($\forall f: \text{Fecha}$) $l \in \text{links}(lli) \wedge_L \text{esReciente?}(e, l, f) \Rightarrow_L$
 $\text{accesosRecientesDia}(lli, l, f) =$
 $\text{cantidadPorDia}(f, (\text{*obtener}(l, e.linkInfo).accesosRecientes)$

2.2.2.1. Funciones auxiliares

```
cantidadPorDia : estrLLI  $\times$  Fecha  $\times$  Lista(acceso)  $\longrightarrow$  nat
cantidadPorDia(e,f,ls)  $\equiv$  if f==prim(ls).dia then
    prim(ls).cantAccesos
else
    cantidadPorDia(e,f,fin(ls))
fi

todosLosLinks : secu(datosLink)  $\longrightarrow$  conj(Link)
todosLosLinks(s)  $\equiv$  if  $\emptyset?(s)$  then  $\emptyset$  else Ag(prim(s).link,todosLosLinks(fin(s))) fi

sinRepetidos : secu( $\alpha$ )  $\longrightarrow$  bool
sinRepetidos(ls)  $\equiv$  if vacia?(ls) then
    true
else
    if esta?(prim(ls),fin(ls)) then false else sinRepetidos(fin(ls)) fi
fi

fechasCorrectas : fecha  $\times$  secu(acceso)  $\longrightarrow$  bool
sinRepetidos(f,ls)  $\equiv$  if vacia?(ls) then
    true
else
    if prim(ls).dia > f then false else fechasCorrectas(f,fin(ls)) fi
fi

accesoOrdenadoNoRepetido : secu(acceso)  $\longrightarrow$  bool
sinRepetidos(ls)  $\equiv$  if long(ls)  $\leq$  1 then
    true
else
    if prim(ls).dia  $\geq$  prim(fin(ls)).dia then
        false
    else
        accesoOrdenadoNoRepetido(fin(ls))
    fi
fi

cantidadDeAccesos : secu(acceso)  $\longrightarrow$  nat
cantidadDeAccesos(ls)  $\equiv$  if vacia?(ls) then
    0
else
    prim(ls).cantAccesos + cantidadDeAccesos(fin(ls))
fi
```

2.3. Algoritmos

Algoritmo 26 idameACat

```
1: function IDAMEACAT(in lli: estrLLI)  $\rightarrow$  res: acat
2:   res  $\leftarrow$  lli.arbolCategorias
3: end function
```

Complejidad: O(1)

//O(1)

Algoritmo 27 iFechaActual

```
1: function IFECHAACtual(in lli: estrLLI)→ res: Fecha
2:   res ← lli.actual //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 28 iLinks

```
1: function ILinks(in lli: estrLLI)→ res: itLinks
2:   res ← crearItLinks(lli) //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 29 iCategoriaLink

```
1: function ICATEGORIALINK(in lli: estrLLI, in l: Link)→ res: Categoria
2:   res ← (*obtener(l,lli.linksInfo)).catDLink //O(|l|)
3: end function
```

Complejidad: O(|l|)

Algoritmo 30 iFechaUltimoAcceso

```
1: function IFECHAULTIMOACCESO(in lli: estrLLI, in l: Link)→ res: Fecha
2:   res ← (ultimo((*obtener(l,lli.linkInfo)).accesosRecientes)).dia //O(|l|)
3: end function
```

Complejidad: O(|l|)

Algoritmo 31 iAccesosRecientesDia

```
1: function IACCESOSRECIENTESDIA(in lli: linkLinkIt, in l: Link, in f: Fecha)→ res: nat
2:   itAccesos accesos ← crearItAccesos(lli,l) //O(|l|)
3:   while haySiguiente(accesos) do //O(|accesos|) = O(1)
4:     if siguiente(accesos).dia == f then //O(1)
5:       res ← siguiente(accesos).cantAccesos //O(1)
6:     end if
7:     avanzar(accesos) //O(1)
8:   end while
9: end function
```

Complejidad: O(|l|)

Algoritmo 32 iIniciarLLI

```
1: function iINICIARLLI(in ac: acat) → res: estrLLI
2:   res.actual ← 1 //O(1)
3:   res.arbolCategorias ← ac //O(1)
4:   nat c ← 0 //O(1)
5:   res.arrayCatLinks ← crearArreglo(dameCantidad(ac))
6:                                     //O(dameCantidad(ac))
7:   res.listaLinks ← vacia() //O(1)
8:   res.linksInfo ← vacio() //O(1)
9:   while c < dameCantidad(res.arbolCategorias) do
10:                                     //O(dameCantidad(res.arbolCategorias))
11:     linksFamilia llist ← vacia() //O(1)
12:     res.arrayCatLinks[c] ← llist //O(1)
13:     c++ //O(1)
14:   end while
15: end function
```

Complejidad: $O(\text{dameCantidad}(\text{res.arbolCategorias}))$

Algoritmo 33 iNuevoLink

```
1: function iNUEVOLINK(in/out lli: estrLLI, in l: Link, in c: Categoria)
2:   itFamilia itF ← crearItFamilia(lli.arbolCategorias, c) //O(|c|)
3:   Lista(acceso) accesoDeNuevoLink ← vacia() //O(1)
4:   datosLink nuevoLink ← <l, c, accesoDeNuevoLink, 0> //O(|l|)
5:   puntero(datosLink) puntLink ← nuevoLink //O(1)
6:   definir(l, puntLink, lli.linkInfo) //O(|l|)
7:   agregarAtras(lli.listaLinks, nuevoLink) //O(1)
8:   while haySiguiente(itF) do //O(alturaAC(ac))
9:     agregarAtras(lli.arrayCatLinks[SiguienteId(itF)-1], puntLink) //O(1)
10:    Avanzar(itF) //O(1)
11:   end while
12: end function
```

Complejidad: $O(|c| + |l| + \text{alturaAC}(\text{ac}))$

Algoritmo 34 iAccederLLI

```
1: function IACCEDERLLI(in/out lli: estrLLI, in l: Link, in f: Fecha)
2:   if lli.actual  $\neq$  f then //O(1)
3:     lli.actual  $\leftarrow$  f //O(1)
4:   end if
5:   puntero(datosLink) puntLink  $\leftarrow$  obtener(l, lli.linkInfo) //O(|l|)
6:   if ultimo((*puntLink).accesos).dia == f then //O(1)
7:     ultimo((*puntLink).accesos).cantAccesos ++ //O(1)
8:   else
9:     agregarAtras((*puntLink).accesos, <f, 1>) //O(1)
10:  end if
11:  if longitud((*puntLink).accesos) == 4 then //O(1)
12:    (*puntLink).cantAccesosRecientes -= prim((*puntLink).accesos).cantAccesos //O(1)
13:    fin((*puntLink).accesos) //O(1)
14:  end if
15:  (*puntLink).cantAccesosRecientes++ //O(1)
16: end function
Complejidad: O(|l|)
```

Algoritmo 35 iCantLinks

```
1: function ICANTLINKS(in lli: estrLLI, c: Categoria)  $\rightarrow$  res: nat
2:   res  $\leftarrow$  longitud(lli.arrayCatLinks[id(lli.arbolCategorias, c)-1]) //O(|c|)
3: end function
Complejidad: O(|c|)
```

Algoritmo 36 iLinksOrdenadosPorAccesos

```
1: function ILINKSORDENADOSPORACCESOS(in lli: estrLLI, in c: Categoria)  $\rightarrow$ 
2:   res: itPuntLinks
3:   nat id  $\leftarrow$  id(lli.arbolCategorias, c) //O(|c|)
4:   id  $\leftarrow$  id-1 //O(1)
5:   Fecha f  $\leftarrow$  1 //O(1)
6:   itPuntLinks itParaFecha  $\leftarrow$  crearItPuntLins(lli, id, f) //O(1)
7:   Fecha fecha  $\leftarrow$  ultFecha(itParaFecha) //O(longitud(lli.arrayCatLinks[id]))
8:   Lista(puntero(datosLink)) listaOrdenada  $\leftarrow$  vacia() //O(1)
9:   if  $\neg$ estaOrdenada?(crearItPuntLins(lli, id, fecha)) then //O(longitud(lli.arrayCatLinks[id]))
10:
11:     while  $\neg$ vacia?(lli.arrayCatLinks[id]) do
12:       itPuntLinks itMax  $\leftarrow$  crearItPuntLins(lli, id, fecha) //O(1)
13:       itMax  $\leftarrow$  buscarMax(itMax, fecha) //O(longitud(lli.arrayCatLinks[id]))
14:       agregarAtras(listaOrdenada, siguiente(itMax)) //O(1)
15:       eliminarSiguiente(itMax) //O(1)
16:     end while
17:     lli.arrayCatLinks[id]  $\leftarrow$  listaOrdenada //O(1)
18:   end if
19:   res  $\leftarrow$  crearItPuntLins(lli, id, fecha) //O(1)
20: end function
Complejidad: O((longitud(lli.arrayCatLinks[id]))2 + |c|)
```

Algoritmo 37 iEsReciente

```
1: function IESRECIENTE(in lli: estrLLI, in l: Link, in f: Fecha)  $\rightarrow$  res: bool
2:   res  $\leftarrow$  f  $\geq$  (fechaUltimoAcceso(lli,l)-2)  $\wedge$  f  $\leq$  fechaUltimoAcceso(lli,l) //O(|l|)
3: end function
Complejidad: O(|l|)
```

2.4. Analisis de complejidades

1. iDameACatLLI

Se devuelve por referencia el arbol del sistema pasado como parametro, esto demora O(1).

Orden Total: O(1)

2. iFechaActual

Devuelve la fecha actual del sistema, esto cuesta O(1).

Orden Total: O(1)

3. iLinksLLI

Devuelve en O(1) un itLinks que itera los nombres de todos los links de nuestro linkLinkIt.

Orden Total: O(1)

4. iCategoriaLink

Dado un link l, se busca en O(|l|) los datos del mismo y, de la tupla obtenida se devuelve por referencia el nombre de la categoria relacionada a ese link en O(1).

Orden Total: O(|l|)

5. iFechaUltimoAcceso

Dado un link l, se busca en O(|l|) los datos del mismo y, de la tupla obtenida se saca el dia del ultimo elemento de la lista de accesos recientes en O(1).

Orden Total: O(|l|)

6. iAccesosRecientesDia

Dado un link y una fecha, se crea en O(|l|) un itAccesos y, se la itera mientras haya siguiente preguntando en O(1) si el dia de siguiente(it) es el mismo que la fecha. En caso de ser cierto, en O(1) se le asigna ese valor al resultado. Iterar la lista os cuesta la longitud de la lista. Pero como a lo sumo tiene 3 elementos, podemos asumir que su complejidad es O(1).

Orden Total: O(|l|)

7. iIniciarLLI

Se le asigna una referencia del arbol de categorias pasado como parametro al arbolCategorias del linkLinkIt en O(1). A actual se le asigna en O(1) un 1, que será la fecha actual del nuevo linkLinkIt. Se crea una lista vacia y un diccTrie vacio, ambos en O(1) y se los asigna a listaLinks y linkInfo respectivamente en O(1). Luego se crea un array cuyo tamaño es la cantidad de categorias de del arbol pasado como parámetro, por lo cual su complejidad es O(cantidad de categorias del arbol) y a cada posicion del array se le asigna una lista vacia que cuesta O(1). Como lo hago para cada posicion, nos cuesta en total O(cantidad de categorias del arbol). En total nos costaria O(2* cantidad de categorias del arbol)= O(cantidad de categorias del arbol). (Sea cant = cantidad de categorias del arbol)

Orden Total: O(cant)

8. **inuevoLink**

Se crea un puntero a datosCat cat donde se le pasa el puntero obtenido por la operacion obtener del modulo arbolCategorias, esto cuesta $O(|c|)$. Se crea una lista de acceso inicializada vacia, que cuesta $O(1)$.

Se crea una tupla datosLink, a la cual se le pasa una tupla con el link dado, el puntero a datosCat y la lista de acceso, la cual tarda $O(|l|)$. Se crea un puntero a datosLink y se le pasa la tupla datosLink, esto cuesta $O(1)$. Se utiliza la operacion definir del diccTrie en la cual se agrega el link dado al diccionario accesosXLink, lo cual tarda $O(|l|)$.

Se utiliza la operacion agregarAtras que agrega el puntero a datosLink a la lista listaLinks, esto demora $O(1)$. Se ingresa a un ciclo si cat es distinto de la operacion puntRaiz de arbolCategorias, esto tarda $O(1)$. Se utiliza la operacion agregarAtras que agrega el puntero a datosLink a la lista que esta en la posicion (*cat).id del arreglo arrayCatLinks, lo cual tarda $O(1)$.

Se modifica el puntero a datosCat y se guarda cat.padre, lo cual tarda $O(1)$. Una vez que no se cumple la condicion del ciclo se del mismo habiendo tardado $O(h)$. Se utiliza la operacion agregarAtras que agrega el puntero a datosLink a la lista que esta en la posicion (*cat).id del arreglo arrayCatLinks, lo cual tarda $O(1)$.

Aclaracion h es igual a la altura de la categoria c.

Orden Total: $O(|c|)+O(1)+O(|l|)+O(1)+O(1)+O(1)+O(h*(O(1)+O(1)))+O(1)=O(|l|+|c|+h)$

9. **iAccederLLI**

Se pregunta si la fecha actual del sistema es igual a f, esto demora $O(1)$, en caso verdadero se deja actual como esta, en caso negativo se modifica a y se guarda f como fecha actual, esto tarda $O(1)$.

Se crea un puntero a datosLink puntLink que se le pasa un puntero obtenido por medio de la operacion obtener del diccionario accesosXLink dando el link que se quiere ingresar al sistema, esto demora $O(|l|)$.

Se pregunta si el dia de la tupla del ultimo elemento de la lista accesosRecientes de la tupla apuntada por el puntero puntLink es igual al f dado, esto cuesta $O(1)$, en caso positivo, se modifica cantAccesos de la misma tupla del elemento sumandole uno, esto demora $O(1)$ en caso negativo se utiliza la operacion agregarAtras y se agrega una tupla acceso con la fecha f y cantAccesos igual a 1 a la lista de accesosRecientes, lo cual demora $O(1)$.

Por ultimo, se consulta por la longitud de la lista accesosRecientes, consultando si la nueva longitud es igual a 4, esto demora $O(1)$, en caso positivo se modificara la lista sacando el primer elemento de la misma. Esto demora $O(1)$.

Orden Total: $O(1)+O(1)+O(1)+O(|l|)+O(1)+O(1)+O(1)+O(1)+O(1)=O(|l|)$

10. **iCantLinks**

Dada una categoria c, obtener su id en el arbol de categorias nos cuesta $O(|c|)$. Luego accedemos en arrayCatLinks a la posicion correspondiente en $O(1)$. Y en $O(1)$ conseguimos la longitud de la lista que allí encontramos.

Orden Total: $O(|c|)$

11. **iLinksOrdenadosPorAccesos**

Dada la categoria c pasada como parametro, obtenemos su id en $O(|c|)$. Luego con coste $O(1)$ creamos un iterador de punteros a datosLink para la lista l alojada en la posicion correspondiente de arrayCatLinks que tiene longitud n. Obtener la ultima fecha de esa lista nos cuesta

$O(n)$ con la funcion `ultFecha`. Creamos una lista de punteros a `datosLink` que al finalizar sera la lista ordenada.

Creamos otro iterador a la lista `l` y llamamos a la funcion `estaOrdenada?` con un costo de $O(n)$. Si ya esta ordenada, devolvemos un puntero a esa lista en $O(1)$, teniendo un costo total de $O(|c|) + 2 * O(n) = O(|c| + n)$. Si no lo esta, creamos una lista de punteros a `datosLink` que al finalizar sera la lista `Ordenada` y luego se entra en el ciclo.

El ciclo se realiza mientras la lista `l` no esté vacia. Por lo que vamos a hacer n veces lo siguiente: Generamos un `itPunLinks` a la lista `l` en $O(1)$, llamamos a la funcion `buscarMax` que nos cuesta $O(n)$ y nos deja un iterador apuntando al link con mas accesos recientes para esa categoria. Agregamos ese puntero a la lista `Ordenada` en $O(1)$ y lo eliminamos de la lista vieja con `eliminarSiguiente` en $O(1)$.

El ciclo nos cuesta en total $O(n) * (O(n) + 3 * O(1)) = O(n^2)$

Finalmente, en $O(1)$ le pasamos a la posicion de `arrayCatLinks` una referencia a la nueva lista `Ordenada`.

Orden Total: $O(|c| + n^2)$

12. iEsReciente

Dado un link `l` y una fecha `f`, llamamos a la funcion `fechaUltimoAcceso` para ese link en $O(|l|)$ y vemos que `f` este en el rango $[fechaUltimoAcceso, fechaUltimoAcceso - 2]$.

Orden Total: $O(|l|)$

2.5. Iterador de Links

2.5.1. Interfaz

parámetros formales

géneros `itLinks`

se explica con: Iterador Bidireccional

Operaciones

`CREARITLINKS(in l: estrLLI) → res: itLinks`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{crearIt}(l)\}$

Complejidad: $O(1)$

Aliasing: No se deben modificar los elementos iterados por `res`.

`HAYSIGUIENTE?(in it: itLinks) → res: bool`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{haySiguiente?}(it)\}$

Complejidad: $O(1)$

Aliasing: No tiene.

`SIGUIENTE(in it: itLinks) → res: Link`

Pre $\equiv \{\text{haySiguiente?}(it)\}$

Post $\equiv \{res =_{\text{obs}} \pi_1(\text{siguiente}(it))\}$

Complejidad: $O(1)$

Aliasing: El link se pasa por referencia, no debe ser modificado.

AVANZAR(**in/out** *it*: itLinks)
Pre $\equiv \{it =_{\text{obs}} it_0 \wedge \text{haySiguiente?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{avanzar}(it_0)\}$
Complejidad: $O(1)$
Aliasing: No tiene.

fin interfaz

2.5.2. Representación

itLinks se representa con itLista(datosLink)

datosLink es tupla<
link: Link,
catDLink: Categoria,
accesosRecientes: Lista(acceso),
cantAaccesosRecientes: nat>

itLinks es un iterador de lista común. Sus complejidades nos alcanzan para iterar una Lista(datosLink).

2.5.3. Invariante de Representación

2.5.3.1. El Invariante Formalmente

Rep : estrITL \rightarrow boolean

$(\forall it: \text{estrITL}) \text{Rep}(it) \equiv \text{true}$

2.5.4. Función de Abstracción

Abs : *e*: estrITL \rightarrow itBi(α)

Rep(*e*)

$(\forall e: \text{estrITL}) \text{Abs}(e) =_{\text{obs}} it: \text{itBi}(\alpha) \mid$

1. $\text{anteriores}(e.\text{iterador}) =_{\text{obs}} \text{anteriores}(it) \wedge$
 $\text{siguientes}(e.\text{iterador}) =_{\text{obs}} \text{siguientes}(it)$

2.5.5. Algoritmos

Algoritmo 38 iCrearItLinks

```
1: function ICLEARITLINKS(in l: lli)  $\rightarrow$  res: estrITL
2:   res  $\leftarrow$  crearIt(lli.listaLink)
3: end function
```

//O(1)

Complejidad: $O(1)$

Algoritmo 39 iHaySiguiente?

```
1: function IHAYSIGUIENTE?(in e: estrITL)  $\rightarrow$  res: bool
2:   res  $\leftarrow$  haySiguiente(e)
3: end function
```

//O(1)

Complejidad: $O(1)$

Algoritmo 40 iSiguiente

```
1: function ISIGUIENTE(in  $e$ : estrITL)  $\rightarrow$   $res$ : Link  
2:    $res \leftarrow (\text{siguiente}(e)).\text{link}$  //O(1)  
3: end function
```

Complejidad: $O(1)$

Algoritmo 41 iAvanzar

```
1: function IAVANZAR(in/out  $e$ : estrITL)  
2:   avanzar( $e$ ) //O(1)  
3: end function
```

Complejidad: $O(1)$

2.5.6. Análisis de complejidades

1. iCrearItLinks

Crea un itLinks con la listaLink del linklinkit que se pasa como parametro y se la asigna a res, esto demora $O(1)$.

Orden Total: $O(1)$

2. iHaySiguiente?

Se llama a HaySiguiente del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

3. iSiguiente

Se llama a Siguiente del Iterador de Lista en $O(1)$. A eso se le aplica la operacion dameLink que también cuesta $O(1)$ y se devuelve una referencia al link resultante.

Orden Total: $O(1)$

4. iAvanzar

Se llama a Avanzar del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

2.6. Iterador de Punteros a DatosLink

2.6.1. Interfaz

parámetros formales

géneros itPuntLinks

se explica con: Iterador Bidireccional

Operaciones

CREARITPUNTLINKS(**in** lli : **estrLLI**, **in** id : **nat**, **in** f : **Fecha**) \rightarrow res : **itPuntLinks**

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{crearIt}(lli, id, f)\}$

Complejidad: $O(1)$

Aliasing: No se deben modificar los elementos iterados por res.

HAYSIGUIENTE?(**in** *it*: itPuntLinks) → *res*: bool

Pre ≡ {*true*}

Post ≡ {*res* =_{obs} *haySiguiente?(it)*}

Complejidad: O(1)

Aliasing: No tiene.

SIGUIENTELINK(**in** *it*: itPuntLinks) → *res*: Link

Pre ≡ {*haySiguiente?(it)*}

Post ≡ {*res* =_{obs} (**siguiente(it)*).link}

Complejidad: O(1)

Aliasing: El link se pasa por referencia, no debe ser modificado.

SIGUIENTECAT(**in** *it*: itPuntLinks) → *res*: Categoria

Pre ≡ {*haySiguiente?(it)*}

Post ≡ {*res* =_{obs} (**siguiente(it)*).catDLink}

Complejidad: O(1)

Aliasing: La categoria se pasa por referencia, no debe ser modificada.

SIGUIENTECANTACCESOSDELLINK(**in** *it*: itPuntLinks) → *res*: nat

Pre ≡ {*haySiguiente?(it)*}

Post ≡ {*res* =_{obs} *cantAccesosDesde(it)*}

Complejidad: O(1)

Aliasing: No tiene

AVANZAR(**in/out** *it*: itPuntLinks)

Pre ≡ {*it* =_{obs} *it*₀ ∧ *haySiguiente?(it)*}

Post ≡ {*it* =_{obs} *avanzar(it)*₀}

Complejidad: O(1)

Aliasing: No tiene.

ELIMINARSIGUIENTE(**in/out** *it*: itPuntLinks)

Pre ≡ {*it* =_{obs} *it*₀ ∧ *haySiguiente?(it)*}

Post ≡ {*it* =_{obs} *eliminarSiguiente(it)*₀}

Complejidad: O(1)

Aliasing: No tiene.

fin interfaz

2.6.2. Representación

itPuntLinks se representa con itLista(puntero(datosLink))

datosLink es tupla<
 link: Link,
 catDLink: Categoria,
 accesosRecientes: Lista(acceso),
 cantAaccesosRecientes: nat>

itPuntLinks es un iterador de lista común. Sus complejidades nos alcanzan para iterar una Lista(puntero(datosLink)).

2.6.3. Invariante de Representación

2.6.3.1. El Invariante Formalmente

$\text{Rep} : \text{estrITPL} \rightarrow \text{boolean}$

$(\forall it : \text{estrITPL}) \text{Rep}(it) \equiv \text{true}$

2.6.4. Función de Abstracción

$\text{Abs} : e : \text{estrITPL} \rightarrow \text{itBi}(\alpha)$

$\text{Rep}(e)$

$(\forall e : \text{estrITPL}) \text{Abs}(e) =_{\text{obs}} \text{it} : \text{itBi}(\alpha) \mid$

1. $\text{anteriores}(e.\text{iterador}) =_{\text{obs}} \text{anteriores}(it) \wedge$
 $\text{siguientes}(e.\text{iterador}) =_{\text{obs}} \text{siguientes}(it)$

2.6.5. Algoritmos

Algoritmo 42 iCrearItPuntLinks

```
1: function ICREARITLINKS(in  $e : \text{estrLLI}$ , in  $id : \text{nat}$ )  $\rightarrow res : \text{estrITPL}$ 
2:    $res \leftarrow \text{crearIt}(e.\text{arrayCatLinks}[id])$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 43 iHaySiguiente?

```
1: function IHAYSIGUIENTE?(in  $e : \text{estrITPL}$ )  $\rightarrow res : \text{bool}$ 
2:    $res \leftarrow \text{haySiguiente}(e)$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 44 iSiguiente

```
1: function ISIGUIENTE(in  $e : \text{estrITPL}$ )  $\rightarrow res : \text{DatosLink}$ 
2:    $res \leftarrow \text{siguiente}(e)$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 45 iSiguienteLink

```
1: function ISIGUIENTELINK(in  $e : \text{estrITPL}$ )  $\rightarrow res : \text{Link}$ 
2:    $res \leftarrow (*\text{siguiente}(e)).\text{link}$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 46 iSiguienteCat

```
1: function ISIGUIENTECAT(in  $e : \text{estrITPL}$ )  $\rightarrow res : \text{Categoria}$ 
2:    $res \leftarrow (*\text{siguiente}(e)).\text{catDLink}$  //O(1)
3: end function
Complejidad: O(1)
```

Algoritmo 47 iSiguienteCantidadAccesosDelLink

```
1: function ISIGUIENTECANTIDADACCESOSDELLINK(in  $e$ : estrITPL)  $\rightarrow$   $res$ : int
2:    $res \leftarrow \text{cantAccesosDesde}(e.\text{fecha})$  //O(1)
3: end function
```

Complejidad: $O(1)$

Algoritmo 48 iAvanzar

```
1: function IAVANZAR(in/out  $e$ : estrITPL)
2:   avanzar( $e$ ) //O(1)
3: end function
```

Complejidad: $O(1)$

Algoritmo 49 iEliminarSiguiente

```
1: function IELIMINARSIGUIENTE(in/out  $e$ : estrITPL)
2:   eliminarSiguiente( $e$ ) //O(1)
3: end function
```

Complejidad: $O(1)$

Algoritmo 50 iBuscarMax

```
1: function IBUSCARMAX(in  $it$ : estrITPL, in  $f$ : Fecha)  $\rightarrow$   $res$ : itPuntLinks
2:    $res \leftarrow \text{copiarIt}(it)$  //O(1)
3:   while haySiguiente( $it$ ) do //O(longitud(siguientes(it)))
4:     if  $\text{cantAccesosDesde}(it, f) > \text{cantAccesosDesde}(res, f)$  then //O(1)
5:        $res \leftarrow it$  //O(1)
6:     end if
7:     avanzar( $it$ ) //O(1)
8:   end while
9: end function
```

Complejidad: $O(\text{longitud}(\text{siguientes}(it)))$

Algoritmo 51 iUltFecha

```
1: function IULTFECHA(in  $it$ : estrITPL)  $\rightarrow$   $res$ : Fecha
2:    $res \leftarrow (\text{ultimo}(*\text{siguiente}(it).\text{accesos})).\text{dia}$  //O(1)
3:   while haySiguiente( $it$ ) do //O(longitud(siguientes(it)))
4:     if  $(\text{ultimo}(*\text{siguiente}(it).\text{accesos})).\text{dia} > res$  then //O(1)
5:        $res \leftarrow (\text{ultimo}(*\text{siguiente}(it).\text{accesos})).\text{dia}$  //O(1)
6:     end if
7:     avanzar( $it$ ) //O(1)
8:   end while
9: end function
```

Complejidad: $O(\text{longitud}(\text{siguientes}(it)))$

Algoritmo 52 iCantAccesosDesde

```
1: function ICANTACCESOSDESDE(in it: estrITPL, in f: Fecha)  $\rightarrow$  res: nat
2:   itAccesos itAcc  $\leftarrow$  (*siguiente(it)).accesos //O(1)
3:   res  $\leftarrow$  0 //O(1)
4:   while haySiguiente(itAcc) do //O(1)
5:     if (siguiente(itAcc)).dia  $\leq$  f  $\wedge$  (siguiente(itAcc)).dia  $\leq$  f-2 then //O(1)
6:       res  $\leftarrow$  res + (siguiente(it)).cantA //O(1)
7:     end if
8:     avanzar(itAcc) //O(1)
9:   end while
10: end function
Complejidad: O(1)
```

Algoritmo 53 iEstaOrdenada?

```
1: function IESTAORDENADA?(in it: estrITPL, in f: Fecha)  $\rightarrow$  res: bool
2:   res  $\leftarrow$  true //O(1)
3:   nat aux  $\leftarrow$  cantAccesosDesde(it,f) //O(1)
4:   avanzar(it) //O(1)
5:   while haySiguiente(it) do //O(longitud(siguientes(it)))
6:     if cantAccesosDesde(it,f) > aux then //O(1)
7:       res  $\leftarrow$  false //O(1)
8:     end if
9:     aux  $\leftarrow$  cantAccesosDesde(it,f) //O(1)
10:    avanzar(it) //O(1)
11:  end while
12: end function
Complejidad: O(longitud(siguientes(it)))
```

2.6.6. Analisis de complejidades

1. iCrearItLinks

Crea un itPuntLinks con la lista que se pasa como parametro y se la asigna a res, esto demora O(1).

Orden Total: O(1)

2. iHaySiguiente?

Se llama a HaySiguiente del Iterador de Lista en O(1).

Orden Total: O(1)

3. iSiguiente

Se llama a Siguiente del Iterador de Lista en O(1). Se devuelve una referencia del elemento link de la tupla DatosLink.

Orden Total: O(1)

4. iSiguienteCat

Se llama a Siguiente del Iterador de Lista en O(1). Se devuelve una referencia del elemento catDLink de la tupla DatosLink.

Orden Total: O(1)

5. **iSiguienteCantidadAccesosDelLink**

Se llama a `cantAccesosDesde` del Iterador en $O(1)$. Se devuelve el valor entero de la cantidad de accesos del link en la posicion actual del iterador.

Orden Total: $O(1)$

6. **iAvanzar**

Se llama a `Avanzar` del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

7. **iEliminarSiguiente**

Se llama a `eliminarSiguiente` del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

8. **iBuscarMax**

Dado un `itPuntLinks` y una fecha, iteramos llamando cada vez a `cantAccesosDesde` para el iterador y la fecha. Cada vez nos cuesta $O(1)$ y lo hacemos una vez para cada iteracion. En total nos cuesta $O(\text{longitud}(\text{siguientes}(\text{it})))$.

En $O(1)$ copiamos el iterador a `res` sólo si la llamada a `cantAccesosDesde` nos dio mayor a la que resultaba del anterior valor de `res`.

finalmente avanzar el iterador nos cuesta $O(1)$ tambien.

Orden Total: $O(\text{longitud}(\text{siguientes}(\text{it})))$

9. **iUltFecha**

Dado un `itPuntLinks`, iteramos pidiendo el día al acceso mas nuevo, para eso generamos un `itAccesos` a la ultima posicion de la lista de accesos en $O(1)$. Luego pedimos la fecha de ese acceso tambien en $O(1)$.

Y evaluamos en $O(1)$ si es mayor a la fecha que teniamos guardada en el resultado. Cambiandola en caso de ser necesario en $O(1)$.

Como lo hacemos para cada link de la primera lista, nos cuesta $O(\text{longitud}(\text{siguientes}(\text{it})))$

Orden Total: $O(\text{longitud}(\text{siguientes}(\text{it})))$

10. **iCantAccesosDesde**

Dado un `itPuntLinks` y una fecha, obtengo en $O(1)$ un `itAccesos` para la lista de accesos del siguiente del iterador.

Luego voy iterando `itAccesos` y si la fecha se encuentra dentro de la fecha pasada y la fecha pasada menos dos, sumo la cantidad de accesos para ese acceso, todo en $O(1)$.

Como la lista de accesos iterada tiene a lo sumo 3 elementos, podemos considerar que iterarla nos lleva tiempo constante, o sea $O(1)$.

Orden Total: $O(1)$

11. **iEstaOrdenada?**

Dado un `itPuntLinks` y una fecha, iteramos llamando cada vez a `cantAccesosDesde` para el iterador y la fecha. Cada vez nos cuesta $O(1)$ y lo hacemos una vez para cada iteracion. En total nos cuesta $O(\text{longitud}(\text{siguientes}(\text{it})))$.

Comparamos `cantAccesosDesde` con la variable `aux` en $O(1)$ y si `aux` es menor, cambiamos `res` a `false` en $O(1)$

Actualizar el aux con cantAccesosDesde y avanzar el iterador nos cuesta $O(1)$ en ambos casos.

Orden Total: $O(\text{longitud}(\text{siguientes}(\text{it})))$

2.7. Iterador de Accesos

2.7.1. Representación

itAccesos se representa con itLista(acceso)

acceso es tupla<
dia: Fecha,
cantAccesos: nat>

itAccesos es un iterador de lista común. Sus complejidades nos alcanzan para iterar una Lista(acceso).

2.7.2. Invariante de Representación

2.7.2.1. El Invariante Formalmente

$\text{Rep} : \text{estrITA} \rightarrow \text{boolean}$

$(\forall it : \text{estrITA}) \text{Rep}(it) \equiv \text{true}$

2.7.3. Función de Abstracción

$\text{Abs} : e : \text{estrITA} \rightarrow \text{itBi}(\alpha) \quad \text{Rep}(e)$

$(\forall e : \text{estrITA}) \text{Abs}(e) =_{\text{obs}} it : \text{itBi}(\alpha) \mid$

1. $\text{anteriores}(e.\text{iterador}) =_{\text{obs}} \text{anteriores}(it) \wedge$
 $\text{siguientes}(e.\text{iterador}) =_{\text{obs}} \text{siguientes}(it)$

2.7.4. Algoritmos

Algoritmo 54 iCrearItAccesos

```
1: function ICREARITACCESOS(in l: Lista(acceso))  $\rightarrow$  res: estrITA
2:   res  $\leftarrow$  crearIt(l) //O(1)
3: end function
```

Complejidad: $O(1)$

Algoritmo 55 iHaySiguiente?

```
1: function IHAYSIGUIENTE?(in e: estrITA)  $\rightarrow$  res: bool
2:   res  $\leftarrow$  haySiguiente(e) //O(1)
3: end function
```

Complejidad: $O(1)$

Algoritmo 56 iSiguiente

1: **function** ISIGUIENTE(**in** e : **estrITA**) \rightarrow res : **Acceso**

2: $res \leftarrow \text{siguiente}(e)$ //O(1)

3: **end function**

Complejidad: $O(1)$

Algoritmo 57 iAvanzar

1: **function** IAVANZAR(**in/out** e : **estrITA**)

2: $\text{avanzar}(e)$ //O(1)

3: **end function**

Complejidad: $O(1)$

2.7.5. Análisis de complejidades

1. iCrearItAccesos

Crea un `itAccesos` con la lista que se pasa como parametro y se la asigna a `res`, esto demora $O(1)$.

Orden Total: $O(1)$

2. iHaySiguiente?

Se llama a `HaySiguiente` del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

3. iSiguiente

Se llama a `Siguiente` del Iterador de Lista en $O(1)$. A eso se le aplica la operacion `dameCat` que también cuesta $O(1)$ y se devuelve una referencia a la categoria resultante.

Orden Total: $O(1)$

4. iAvanzar

Se llama a `Avanzar` del Iterador de Lista en $O(1)$.

Orden Total: $O(1)$

3. Módulo `diccTrie(clave, significado)`

3.1. Interfaz

parámetros formales

géneros `diccTrie(α)`

usa: `bool`, puntero, `arreglo(α)`, `conj(α)`

se explica con: `Diccionario(string, α)`

Operaciones

`VACIO()` $\rightarrow res: \text{diccTrie}(c, s)$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} vacio()\}$

Complejidad: $O(1)$

Aliasing: No tiene

`DEFINIR(in $c: \text{string}$, in $s: \alpha$, in/out $d: \text{diccTrie}(s)$)`

Pre $\equiv \{d =_{\text{obs}} d_0\}$

Post $\equiv \{d =_{\text{obs}} definir(c, s, d_0) \wedge alias(significado(d, c), s)\}$

Complejidad: $O(|c|)$

Aliasing: Se genera alias con s en el significado de c . Si se modifica s , se modifica el significado de c .

`DEF?(in $c: \text{string}$, in $d: \text{diccTrie}(s)$) $\rightarrow res: \text{bool}$`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} def?(c, d)\}$

Complejidad: $O(|c|)$

Aliasing: No tiene

`OBTENER(in $c: \text{string}$, in $d: \text{diccTrie}(s)$) $\rightarrow res: \alpha$`

Pre $\equiv \{def?(c, d)\}$

Post $\equiv \{res =_{\text{obs}} obtener(c, d) \wedge esAlias(res, significado(d, c))\}$

Complejidad: $O(|c|)$

Aliasing: res es modificable.

fin interfaz

3.2. Representación

`DiccTrie(α)` se representa con `estrDT`, donde `estrDT` es `Puntero(Nodo)`

`Nodo` es `tupla< arreglo: arreglo(Puntero(Nodo))[256], significado: Puntero(α) >`

La estructura es un puntero a `Nodo` en la cual cada nodo es una tupla entre un arreglo y un significado para el dicc. Cada posición del arreglo representa una letra y su contenido es un puntero al nodo de la letra siguiente o a `Null`.

3.2.1. Invariante de Representación

3.2.1.1. El Invariante Informalmente

1. No hay repetidos en arreglo de Nodo salvo por Null. Todas las posiciones del arreglo están definidas.
2. No se puede volver al Nodo actual siguiendo alguno de los punteros hijo del actual o de alguno de los hijos de estos.
3. O bien el Nodo es una hoja, o todos sus punteros hijo no-nulos llevan a hojas siguiendo su recorrido.

3.2.1.2. El Invariante Formalmente

$\text{Rep} : \text{estrDT} \rightarrow \text{boolean}$

$(\forall e : \text{estrDT}) \text{Rep}(e) \equiv \text{true} \iff$

1. $(\forall i, j : \text{nat}) 0 \leq i \leq 255 \wedge 0 \leq j \leq 255 \Rightarrow$
 $\text{Definido?}((*)e.\text{Arreglo}, i) \wedge \text{Definido?}((*)e.\text{Arreglo}, j) \wedge$
 $(i = j) \vee$
 $(i \neq j \wedge ((*)e.\text{Arreglo}[i] = \text{null} \wedge (*)e.\text{Arreglo}[j] = \text{null} \vee$
 $(*)e.\text{Arreglo}[i] \neq (*)e.\text{Arreglo}[j]) \wedge_L$
2. $(\neg \exists n : \text{nat}) \text{EncAEstrDTEnNMov}(e, e, n) \wedge_L$
3. $\text{SonTodosNullOLosHijosLoSon}(e)$

3.2.2. Función de Abstracción

$\text{Abs} : e : \text{estrDT} \rightarrow \text{diccT}(c, \alpha)$

$\text{Rep}(e)$

$(\forall e : \text{estrDT}) \text{Abs}(e) =_{\text{obs}} d : \text{diccT}(c, \alpha) \mid$

1. $(\forall c : \text{clave}) \text{def?}(c, d) =_{\text{obs}} \text{estaDefinido?}(c, e) \wedge_L$
2. $(\forall h : \text{clave}) \text{def?}(h, d) \Rightarrow \text{obtener}(h, d) =_{\text{obs}} \text{ObtenerS}(h, *(e))$

3.2.2.1. Funciones auxiliares

$\text{EncAEstrDTEnNMov} : \text{estrDT} \times \text{estrDT} \times \text{Nat} \rightarrow \text{Bool}$

$\text{EncAEstrDTEnNMov}(\text{buscado}, \text{actual}, n) \equiv \text{if } (n = 0) \text{ then}$

$\text{EstaEnElArregloActual?}(\text{buscado}, \text{actual}, 255)$

else

$\text{RecurrenciaConLosHijos}(\text{buscado}, \text{actual}, n-1, 255)$

fi

$\text{EstaEnElArregloActual?} : \text{estrDT} \times \text{estrDT} \times \text{nat} \rightarrow \text{Bool}$

$\text{EstaEnElArregloActual?}(\text{buscado}, \text{actual}, n) \equiv \text{if } (n=0) \text{ then}$

$((*)\text{actual}.\text{Arreglo}[0] = \text{buscado})$

else

$((*)\text{actual}.\text{Arreglo}[n] = \text{buscado}) \vee (\text{EstaEnElArregloActual?}(\text{buscado}, \text{actual}, n-1))$

fi

RecurrenciaConLosHijos : $\text{estrDT} \times \text{estrDT} \times \text{nat} \times \text{nat} \longrightarrow \text{Bool}$
 RecurrenciaConLosHijos(buscado,actual,n,i) \equiv **if** (i = 0) **then**
 EncAEstrDTEnNMov(buscado,(*actual).Arreglo[0],n)
 else
 EncAEstrDTEnNMov(buscado,
 (*actual).Arreglo[i],n) \vee
 (RecurrenciaConLosHijos(buscado,actual,n,i-
 1)
 fi

SonTodosNullOLosHijosLoSon : $\text{estrDT} \longrightarrow \text{Bool}$
 SonTodosNullOLosHijosLoSon(e) \equiv Los256SonNull(e,255) \vee BuscarHijosNull (e, 255)

Los256SonNull : $\text{estrDT} \times \text{nat} \longrightarrow \text{Bool}$
 Los256SonNull(e,i) \equiv **if** (i = 0) **then**
 ((*e).Arreglo[0] = null)
 else
 ((*e).Arreglo[i] = null) \wedge Los256SonNull(e, i-1)
 fi

BuscarHijosNull : $\text{estrDT} \times \text{nat} \longrightarrow \text{Bool}$
 BuscarHijosNull(e,i) \equiv **if** (i = 0) **then**
 ((*e).Arreglo[0] = null) \vee SonTodosNullOLosHijosLoSon((*e).Arreglo[0])
 else
 (((e).Arreglo[i] = null) \vee SonTodosNullOLosHijosLo-
 Son((*e).Arreglo[i])) \wedge BuscarHijosNull(e,i-1)
 fi

estaDefinido? : $\text{string} \times \text{estrDT} \longrightarrow \text{bool}$
 estaDefinido?(c,e) \equiv **if** (e==Null) **then** false **else** NodoDef?(c,*(e)) **fi**

NodoDef? : $\text{string} \times \text{Nodo} \longrightarrow \text{bool}$
 NodoDef?(c,n) \equiv **if** (vacía?(c)) **then**
 true
 else
 if (n.arreglo[ord(prim(c))] \neq Null) **then**
 NodoDef?(fin(c),*(n.arreglo[ord(prim(c))]))
 else
 false
 fi
 fi

ObtenerS : $\text{string} \times \text{Nodo} \longrightarrow \alpha$
 ObtenerS(c,n) \equiv **if** (vacía?(c)) **then**
 *(n.significado)
 else
 ObtenerS(fin(c),*(n.arreglo[ord(prim(c))]))
 fi

3.3. Algoritmos

Algoritmo 58 iVacio

```
1: function IVACIO  $\rightarrow res: \text{estrDT}$ 
2:   var  $n: \text{Puntero}(\text{Nodo})$  //O(1)
3:    $n \leftarrow \text{Null}$  //O(1)
4:    $res \leftarrow n$  //O(1)
5: end function
```

Complejidad: $O(1)$

Algoritmo 59 iDefinir

```
1: function IDEFINIR(in  $c: \text{string}$ , in  $s: \alpha$ , in/out  $e: \text{estrDT}$ )
2:   if ( $e = \text{Null}$ ) then //O(1)
3:     var  $n: \text{Nodo}$  //O(1)
4:      $n \leftarrow \text{iNuevoNodo}$  //O(1)
5:      $e \leftarrow \&(n)$  //O(1)
6:   end if
7:   var  $n_1: \text{Nodo}$  //O(1)
8:    $n_1 \leftarrow *(e)$  //O(1)
9:   var  $i: \text{nat}$  //O(1)
10:   $i \leftarrow 0$  //O(1)
11:  while ( $i < |c|$ ) do //O( $|c|$ )
12:    if ( $n_1.\text{arreglo}[\text{ord}(c[i])] = \text{Null}$ ) then //O(1)
13:      var  $n_2: \text{Nodo}$  //O(1)
14:       $n_2 \leftarrow \text{iNuevoNodo}$  //O(1)
15:       $n_1.\text{arreglo}[\text{ord}(c[i])] \leftarrow \&(n_2)$  //O(1)
16:    end if
17:     $n_1 \leftarrow *(n_1.\text{arreglo}[\text{ord}(c[i])])$  //O(1)
18:     $i++$  //O(1)
19:  end while
20:   $n_1.\text{significado} \leftarrow s$  //O(1)
21: end function
```

Complejidad: $O(|c|)$

Algoritmo 60 iNuevoNodo

```
1: function INUEVONODO  $\rightarrow res: \text{Nodo}$ 
2:   var  $n: \text{Nodo}$  //O(1)
3:    $n.\text{significado} \leftarrow \text{Null}$  //O(1)
4:   for ( $i: \text{nat} \leftarrow 0; i < 256; i++$ ) do //O( $256*1$ )
5:      $n.\text{arreglo}[i] \leftarrow \text{Null}$  //O(1)
6:   end for
7:    $res \leftarrow n$  //O(1)
8: end function
```

Complejidad: $O(1)$

Algoritmo 61 iDef?

```
1: function IDEF?(in  $c$ : string, in  $e$ : estr)  $\rightarrow$   $res$ : bool
2:   if ( $e \neq \text{Null}$ ) then //O(1)
3:     var  $n$ : Nodo //O(1)
4:      $n \leftarrow *(e)$  //O(1)
5:     var  $i$ : nat //O(1)
6:     var  $i \leftarrow 0$  //O(1)
7:      $res \leftarrow \text{true}$  //O(1)
8:     while ( $i < |c|$ ) do //O(|c|)
9:       if ( $n.\text{arreglo}[\text{ord}(c[i])] \neq \text{Null}$ ) then //O(1)
10:         $n \leftarrow *(n.\text{arreglo}[\text{ord}(c[i])])$  //O(1)
11:      else
12:         $res \leftarrow \text{false}$  //O(1)
13:         $i \leftarrow \text{long}(c)$  //O(1)
14:      end if
15:       $i++$  //O(1)
16:    end while
17:  else
18:     $res \leftarrow \text{false}$  //O(1)
19:  end if
20: end function
Complejidad:  $O(|c|)$ 
```

Algoritmo 62 iObtener

```
1: function IOBTENER(in  $c$ : string, in  $e$ : estr)  $\rightarrow$   $res$ :  $\alpha$ 
2:   var  $n$ : Nodo //O(1)
3:    $n \leftarrow *(e)$  //O(1)
4:   var  $i$ : nat //O(1)
5:   var  $i \leftarrow 0$  //O(1)
6:   while ( $i < |c|$ ) do //O(|c|)
7:      $n \leftarrow *(n.\text{arreglo}[\text{ord}(c[i])])$  //O(1)
8:      $i++$  //O(1)
9:   end while
10:   $res \leftarrow n.\text{significado}$  //O(1)
11: end function
Complejidad:  $O(|c|)$ 
```

3.4. Analisis de complejidades

1. iVacio

Se crea la variable p de tipo Puntero a Nodo en $O(1)$, luego se le asigna “Null” en $O(1)$ y finalmente se le asigna a res .

Orden Total: $O(1)+O(1)+O(1)=O(1)$

2. iDefinir

Se evalua si no nada definido y se crea un nuevo Nodo en caso afirmativo, luego se le asigna el puntero a este Nodo a la $estrDT$. Esto se logra en $O(1)$. Posteriormente se crean algunas variables y se le asignan valores en $O(1)$ y se hace un loop con la longitud del string en

$O(|\text{string}| * O(\text{operaciones dentro del loop}))$. En el loop se hace un if para evaluar si ya esta definida esa letra y en caso negativo se crea un nuevo nodo y se asigna el puntero a ese nodo. Todo esto se hace en $O(1)$. Luego se asigna al nodo el nodo al cual este apunta en la posición de la letra evaluada y se incrementa el contador del loop. Esto se hace en $O(1)$. Finalmente se asigna al ultimo nodo iterado el significado

Orden Total: $O(1+1+1+1)+O(1+1+1+1)+O(|\text{string}|*(O(1+1+1+1)+O(1+1)))+O(1) = O(1)+O(|\text{string}|)+O(1) = \mathbf{O(|string|)}$

3. iNuevoNodo

Se crea una variable de tipo Nodo y se le asigna “Null” en $O(1)$. Luego se realiza un For iterando entre 0 y 256 y asignandole a cada posicion del Nodo “Null” en $O(1)$ dando un total para el For de $O(256)$. Finalmente se asigna el nodo a res en $O(1)$.

Orden Total: $O(1+1)+O(256*(O(1)))+O(1)=\mathbf{O(1)}$

4. iDef?

Se evalua si hay algo definido en $O(1)$. En caso afirmativo se crean variables y se le asignan valor en $O(1)$ y luego se realiza un loop iterando la longitud del string en $O(|\text{string}|*(\text{operaciones dentro del loop}))$. Dentro del loop se evalua si esta definido el char correspondiente a la iteración en y se le asigna al nodo el nodo apuntado en la posición iterada en $O(1)$, caso contrario se asigna “false” a res en $O(1)$. Finalmente incrementa el iterador en $O(1)$.

Orden Total: $O(1)+O(1+1+1+1+1)+O(|\text{string}|*(O(1+1)+O(1)))=\mathbf{O(|string|)}$

5. iObtener

Se crean variables y se les asigna valor en $O(1)$. Luego se realiza un loop iterando la longitud del string en $O(|\text{string}|*O(\text{operaciones dentro del loop}))$. Dentro del loop se asigna al nodo el nodo apuntado en la posición iterada y avanza el iterador en $O(1)$. Finalmente se asigna a res el significado en el ultimo nodo asignado en $O(1)$.

Orden Total: $O(1+1+1+1)+O(|\text{string}|*O(1+1))+O(1)=\mathbf{O(|string|)}$