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

1.1. Interfaz

parámetros formales

géneros acat
se explica con: ArbolDeCategorias

Operaciones

CATEGORIASAC(**in** ac: `acat`)→ res: **itCategorias**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} categorias(ac)\}$
Complejidad: O(1)
Aliasing: No se debe modificar nada de lo iterado por res.

RAIZAC(**in** ac: `acat`)→ res: **Categoria**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} raiz(ac)\}$
Complejidad: O(1)
Aliasing: El nombre de la categoría raiz se pasa por referencia, no debe ser modificado.

RAIZAC(**in** ac: `acat`)→ res: **nat**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} \#categorias(ac)\}$
Complejidad: O(1)
Aliasing: No tiene

IDAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **nat**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} id(ac, c)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

ALTURACATAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **nat**
Pre ≡ $\{esta?(c, ac)\}$
Post ≡ $\{res =_{\text{obs}} alturaCategoria(ac, c)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

HIJOSAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **itHijos**
Pre ≡ $\{esta?(c, ac)\}$
Post ≡ $\{res =_{\text{obs}} hijos(ac, c)\}$
Complejidad: O(|c|)
Aliasing: No se debe modificar nada de lo iterado por res.

PADREAC(**in** ac: `acat`, **in** c: `Categoria`)$\rightarrow$ res: **Categoria**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\mathrm{obs}} padre(ac, c)\}$
Complejidad: O(|c|)
Aliasing: El nombre de la categoría padre se pasa por referencia, no debe ser modificado.

ALTURAAC(**in** ac: `acat`)$\rightarrow$ res: **nat**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\mathrm{obs}} altura(ac)\}$
Complejidad: O(1)
Aliasing: No tiene.

NUEVOAC(**in** c: `Categoria`)$\rightarrow$ res: **acat**
Pre $\equiv \{\neg vacia?(c)\}$
Post $\equiv \{res =_{\mathrm{obs}} nuevo(c)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

AGREGARAC(**in/out** ac: `acat`, **in** c: `categoria`, **in** h: `categoria`)
Pre $\equiv \{esta?(c, ac) \wedge \neg esta?(h, ac) \wedge \neg vacia?(h) \wedge ac_0 =_{\mathrm{obs}} ac\}$
Post $\equiv \{ac =_{\mathrm{obs}} agregar(ac_0, c, h)\}$
Complejidad: O(|c|+|h|)
Aliasing: No hay alias ya que no devuelve nada.

ESTA?(**in** c: `categoria`, **in** ac: `acat`)$\rightarrow$ res: **bool**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\mathrm{obs}} esta?(c, ac)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

ESSUBCATEGORIA(**in** ac: `acat`, **in** c: `categoria`, **in** h: `categoria`)$\rightarrow$ res: **bool**
Pre $\equiv \{esta?(c, ac) \wedge esta?(h, ac)\}$
Post $\equiv \{res =_{\mathrm{obs}} esSubCategoria(ac, c, h)\}$
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 diccTrie(*familia*) para cada categoria, un puntero a su datosCat correspondiente de la lista *categorias* para acceder a esos datos en O(longitud de la categoria).

En *raiz* guardamos un puntero a datosCat de la categoria raiz del arbol para accedrla en O(1)

cantidad es la cantidad de links 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 alutra 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*' 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 + '*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

Rep : estrAC $\rightarrow$ boolean

$(\forall\, ac \colon \texttt{estrAC})$ Rep(ac) $\equiv$ true $\Longleftrightarrow$

1. $(\forall\, c \colon \texttt{Categoria})(def?(c, e.familia)) \Leftrightarrow (*obtener(c, e.familia)).categoria = c \wedge_{\text{L}}$
2. $(\forall\, c_1 \colon \texttt{Categoria})(def?(c_1, e.familia)) \Leftrightarrow ((c_1 == e.raiz) \vee$ $((\exists\, c_2 \colon \texttt{Categoria})(def?(c_2, e.familia)) \wedge_{\text{L}} c_1 \in (*obtener(c_2, e.familia)).hijos)) \wedge_{\text{L}}$
3. $(\forall\, c \colon \texttt{Categoria})(def?(c, e.familia) \Leftrightarrow$ $(((\exists\, d \colon \texttt{datosCat})esta?(d, e.categorias) \wedge d.categoria == c) \wedge_{\text{L}} d == obtener(c, e.familia)))$ $\wedge_{\text{L}}$
4. $(\forall\, c_1, c_2 \colon \texttt{string})(def?(c_1, e.familia)) \wedge (def?(c_2, e.familia)) \Rightarrow_L$ $c_2 \in *((obtener(c_1, e.familia))).hijos \Leftrightarrow$ $(*(*(obtener(c_2, e.familia))).padre).categoria = c_1 \wedge_{\text{L}}$
5. $e.cantidad = longitud(e.categorias) \wedge_{\text{L}}$
6. $(\forall\, c \colon \texttt{categoria})(def?(c, e.familia)) \wedge c = e.raiz \Rightarrow_L$ $(*(obtener(c, e.familia))).altura = 1 \wedge_{\text{L}}$
7. $(\forall\, c \colon \texttt{Categoria})(def?(c, e.familia)) \Rightarrow_L (*obtener(c, e.familia)).altura \leq e.alturaMax$ $\wedge_{\text{L}}$
8. $(\exists\, c \colon \texttt{Categoria})(def?(c, e.familia)) \wedge_{\text{L}} *((obtener(c, e.familia))).altura = e.alturaMax$ $\wedge_{\text{L}}$
9. $(\forall\, c_1, c_2 \colon \texttt{string})(def?(c_1, e.familia)) \wedge (def?(c_2, e.familia)) \wedge_{\text{L}}$ $((\exists\, d \colon \texttt{datosCat})d \in (*(obtener(c_1, e.familia))).hijos \wedge d.categoria == c_2) \Rightarrow_L$ $(*(obtener(c_2, e.familia))).altura = 1 + (*(obtener(c_1, e.familia))).altura \wedge_{\text{L}}$
10. $(\forall\, c \colon \texttt{Categoria})(def?(c, e.familia)) \Rightarrow_L (*(obtener(c, e.familia))).id \leq e.cant \wedge_{\text{L}}$
11. $(\forall\, c_1, c_2 \colon \texttt{Categoria})(def?(c_1, e.familia)) \wedge (def?(c_2, e.familia)) \wedge c_1 \neq c_2 \Rightarrow_L$ $(*(obtener(c_1, e.familia))).id \neq (*(obtener(c_2, e.familia))).id$

1.2.2. Función de Abstracción

Abs : e: `estrAC` $\rightarrow$ acat Rep(e)

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

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

1.2.2.1. Funciones auxiliares

todasLasCategorias : secu(datosCat) $\longrightarrow$ conj(categoria)
todasLasCategorias(cs) $\equiv$ **if** vacia?(cs) **then**
  $\emptyset$()
else
  Ag((prim(cs)).categoria,todasLasCategorias(fin(cs)))
fi

1.3. Algoritmos

Algoritmo 1 iCategoriasAC

```
function iCategoriasAC(in ac: estrAC)→ res: itCategorias
    res ← crearItCategorias(ac)                                    //O(1)
end function
```

Complejidad: O(1)

Algoritmo 2 iRaizAC

```
function iRaiz(in ac: estrAC)→ res: Categoria
    res ← (*(ac.raiz)).categoria                                   //O(1)
end function
```

Complejidad: O(1)

Algoritmo 3 iDameCantidad

```
function iDameCantidad(in ac: estrAC)→ res: nat
    res ← ac.cantidad                                              //O(1)
end function
```

Complejidad: O(1)

Algoritmo 4 iIdAC

```
function iId(in ac: estrAC, in c: Categoria)→ res: nat
    res ← ((*obtener(c,ac.familia)).id                             //O(|c|)
end function
```

Complejidad: O(|c|)

Algoritmo 5 iAlturaCatAC

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

Complejidad: O(|c|)

Algoritmo 6 iHijosAC

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

Complejidad: O(|c|)

Algoritmo 7 iPadreAC

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

Complejidad: O(|c|)

Algoritmo 8 iAlturaAC

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

Complejidad: O(1)

Algoritmo 9 iNuevoAC

```
function iNuevoAC(in c: Categoria) → res: estrAC
    res.cantidad ← 1                                               //O(1)
    datosCat tuplaA                                                //O(1)
    tuplaA ← tupla(c,1,1,vacio(),Null)                             //O(|c|)
    puntero(datosCat) punt ← &tuplaA                               //O(1)
    res.raiz ← punt                                                //O(1)
    res.alturaMax ← 1                                              //O(1)
    definir(c, punt, res.familia)                                  //O(|c|)
    agregarAtras(tuplaA, res.categorias)                           //O(1)
end function
```

Complejidad: O(|c|)

Algoritmo 10 iAgregarAC

```
function iAgregarAC(in/out ac: estrAC, in c: Categoria, in h: Categoria)
    puntero(datosCat) puntPadre ← obtener(c,ac.familia)                                    //O(|c|)
    if (*puntPadre).altura == ac.alturaMax then                                            //O(1)
        ac.alturaMax++                                                                     //O(1)
    end if
    datosCat tuplaA ← (h,ac.cantidad+1,(*puntPadre).altura+1,vacio(),puntPadre)             //O(|h|)
    puntero(datosCat) punt ← &tuplaA                                                       //O(1)
    Agregar((*puntPadre).hijos,punt)                                                       //O(1)
    definir(h,punt,ac.familia)                                                             //O(|h|)
    ac.cantidad++                                                                          //O(1)
    agregarAtras(tuplaA,ac.categorias)                                                     //O(1)
end function
```

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

Algoritmo 11 iEsta?

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

Complejidad: O(|c|)

Algoritmo 12 iEsSubCategoria

```
function iEsSubCategoria(in ac: estrAC, in c: Categoria, in h: Categoria)→ res: bool
    res ← false                                                                            //O(1)
    if h == c then                                                                         //O(|h|)
        res ← true                                                                         //O(1)
    else
        if h == raizAC(ac) then                                                            //O(|h|)
            res ← false                                                                    //O(1)
        else
            puntero(datosCat) actual ← (*obtener(h,ac.familia)).padre                      //O(|h|)
            puntero(datosCat) puntC ← (*obtener(c,ac.familia))                             //O(|c|)
            while res == false ∧ actual ≠ NULL do                                          //O(alturaAC(ac))
                if puntC.Id == actual.Id then                                              //O(1)
                    res ← true                                                             //O(1)
                else
                    actual ← (*actual).padre                                               //O(1)
                end if
            end while
        end if
    end if
end function
```

Complejidad: O(|h| + |c| + alturaAC(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 asginamos 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 Categorias

1.5.1. Interfaz

parámetros formales

géneros itCategorias
se explica con: Iterador Bidireccional

Operaciones

crearItCategotias(**in** ac: `acat`)$\rightarrow$ res: **itCategorias**
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} crearIt(ac.categorias)\}$
Complejidad: O(1)
Aliasing: No debe modificarse ningún elemento iterado por res.

haySiguiente?(**in** it: `itCategorias`)$\rightarrow$ res: **bool**
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} haySiguiente?(it)\}$
Complejidad: O(1)
Aliasing: No tiene.

siguiente(**in** it: `itCategorias`)$\rightarrow$ res: **Categoria**
Pre $\equiv$ $\{haySiguiente?(it)\}$
Post $\equiv$ $\{res =_{\text{obs}} \pi_1(siguiente(it))\}$
Complejidad: O(1)
Aliasing: La categoría se pasa por referencia, no debe ser modificada.

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

fin interfaz

1.5.2. Representación

itCategorias se representa con itLista(datosCat)

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

itCategorias 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

Rep : estrITC → boolean

($\forall\, it$: `estrITC`) Rep(it) ≡ true

1.5.4. Función de Abstracción

Abs : e: `estrITC` → itBi(α) Rep(e)

($\forall\, e$: `estrITC`) Abs(e) $=_{\text{obs}}$ it: itBi(α) |

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

1.5.5. Algoritmos

Algoritmo 13 iCrearItCategorias

```
function iCrearItCategorias(in ac: estrAC)→ res: estrITC
    res ← crearIt(ac.categorias)                    //O(1)
end function
```

Complejidad: O(1)

Algoritmo 14 iHaySiguiente?

```
function iHaySiguiente?(in e: estrITC)→ res: bool
    res ← haySiguiente(e)                           //O(1)
end function
```

Complejidad: O(1)

Algoritmo 15 iSiguiente

```
function iSiguiente(in e: estrITC)→ res: Categoria
    res ← (siguiente(e)).categoria                  //O(1)
end function
```

Complejidad: O(1)

Algoritmo 16 iAvanzar

```
function iAvanzar(in/out e: estrITC)
    avanzar(e)                                      //O(1)
end function
```

Complejidad: O(1)

1.5.6. Analisis de complejidades

1. **iCrearItCategorias**

Crea un itCategorias con la lista del arbol de categorias 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 al valor categoria 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`)$\rightarrow$ res: **itFamilia**
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} obtener(ac, c)\}$
Complejidad: O(|c|)
Aliasing: Los elementos iterados por res no deben ser modificados.

HAYSIGUIENTE?(**in** it: `itFamilia`)$\rightarrow$ res: **bool**
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} (it \neq NULL)\}$
Complejidad: O(1)
Aliasing: No tiene.

SIGUIENTECAT(**in** it: `itFamilia`)$\rightarrow$ res: **Categoria**
Pre $\equiv$ $\{haySiguiente?(it)\}$
Post $\equiv$ $\{res =_{\text{obs}} (*it).catgoria\}$
Complejidad: O(1)
Aliasing: la Categoria se pasa por referencia y no debe ser modificado.

SIGUIENTEID(**in** it: `itFamilia`)$\rightarrow$ res: **int**
Pre $\equiv$ $\{haySiguiente?(it)\}$
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 $\equiv \{it =_{\text{obs}} it_0 \wedge haySiguiente?(it)\}$

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

Rep : estrITF → boolean

$(\forall\, it$: `estrITF`$)$ Rep(it) $\equiv$ true

1.6.4. Algoritmos

Algoritmo 17 iCrearItFamilia

```
function iCrearItFamilia(in ac: estrAC, in c: Categoria) → res: estrITA
    res ← obtener(ac.familia,c)                                    //O(|c|)
end function
```

Complejidad: O(|c|)

Algoritmo 18 iHaySiguiente?

```
function iHaySiguiente?(in e: estrITF) → res: bool
    res ← e ≠ NULL                                                 //O(1)
end function
```

Complejidad: O(1)

Algoritmo 19 iSiguienteCat

```
function iSiguiente(in e: estrITF) → res: Categoria
    res ← (*e).categoria                                           //O(1)
end function
```

Complejidad: O(1)

Algoritmo 20 iSiguienteId

```
function ISIGUIENTE(in e: estrITF)→ res: int
    res ← (*e).id                                   //O(1)
end function
```

Complejidad: O(1)

Algoritmo 21 iAvanzar

```
function IAVANZAR(in/out e: estrITF)
    e ← (*e).padre                                  //O(1)
end function
```

Complejidad: O(1)

1.6.5. Analisis 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 Categoria 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`)$\rightarrow$ res: **itHijos**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} crearIt((*Obtener(ac, c)).hijos)\}$
Complejidad: O(|c|)
Aliasing: No se pueden modificar los datosCat del iterador.

HAYSIGUIENTE?(**in** it: `itHijos`)$\rightarrow$ res: **bool**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} haySiguiente?(it)\}$
Complejidad: O(1)
Aliasing: No tiene.

SIGUIENTE(**in** it: `itHijos`)$\rightarrow$ res: **Categoria**
Pre $\equiv \{haySiguiente?(it)\}$
Post $\equiv \{res =_{\text{obs}} \pi_1(*siguiente(it))\}$
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 : estrITh $\rightarrow$ boolean

($\forall$ it: `estrITH`) Rep(it) $\equiv$ true

1.7.4. Función de Abstracción

Abs : e: `estrITC` $\rightarrow$ itBi(α) Rep(e)

($\forall\, e$: `estrITC`) Abs(e) $=_{\text{obs}}$ it: itBi(α) |

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

1.7.5. Algoritmos

Algoritmo 22 iCrearItHijos

```
function iCrearItHijos(in ac: acat,in c: Categoria)→ res: estrITH
    res ← crearIt((*obtener(ac.familia,c)).hijos)        //O(|c|)
end function
```

Complejidad: O(1)

Algoritmo 23 iHaySiguiente?

```
function iHaySiguiente?(in e: estrITH)→res: bool
    res ← haySiguiente(e)        //O(1)
end function
```

Complejidad: O(1)

Algoritmo 24 iSiguiente

```
function iSiguiente(in e: estrITH)→ res: Categoria
    res ← (*siguiente(e)).categoria        //O(1)
end function
```

Complejidad: O(1)

Algoritmo 25 iAvanzar

```
function iAvanzar(in/out e: estrITH)
    avanzar(e)        //O(1)
end function
```

Complejidad: O(1)

1.7.6. Analisis 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 tambíen 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 ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} lli.arbolCategorias\}$
Complejidad: O(1)
Aliasing: res es una referencia a lli.arbolCategorias, no debe modificarse.

FECHAACTUAL(**in** lli: `linkLinkIt`)→ res: **Fecha**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} fechaActual(lli)\}$
Complejidad: O(1)
Aliasing: No tiene

LINKS(**in** lli: `linkLinkIt`)→ res: **itLinks**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} links(lli)\}$
Complejidad: O(1)
Aliasing: No deben modificarse los elementos iterados por res.

CATEGORIALINK(**in** lli: `linkLinkIt`, **in** l: `Link`)→ res: **Categoria**
Pre ≡ $\{l \in links(lli)\}$
Post ≡ $\{res =_{\text{obs}} categoriaLink(lli, l)\}$
Complejidad: O(|l|)
Aliasing: La categoria se devuelve por referencia, no debe modificarse.

FECHAULTIMOACCESO(**in** lli: `linkLinkIt`, **in** l: `Link`)→ res: **Fecha**
Pre ≡ $\{l \in links(lli)\}$
Post ≡ $\{res =_{\text{obs}} fechaUltimoAcceso(lli, l)\}$
Complejidad: O(|l|)
Aliasing: No tiene

ACCESOSRECIENTESDIA(**in** lli: `linkLinkIt`, **in** l: `Link`, **in** f: `Fecha`)→ res: **nat**
Pre ≡ $\{l \in links(lli)\}$
Post ≡ $\{res =_{\text{obs}} accesosRecientesDia(lli, l, f)\}$
Complejidad: O(|l|)
Aliasing: No tiene

INICIARLLI(**in** ac: `acat`)→ res: **linkLinkIt**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} iniciar(ac)\}$
Complejidad: O(#categorias(ac))
Aliasing: No tiene.

NUEVOLINKLLI(**in/out** lli: `linkLinkIt`, **in** l: `Link`, **in** c: `Categoria`)
Pre ≡ $\{c \in categorias(lli) \wedge l \notin links(lli) \wedge \neg vacia?(l) \wedge lli_0 = lli\}$
Post ≡ $\{lli = nuevoLink(lli_0, l, c)\}$
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 ≡ $\{l \in links(lli) \wedge f \geq fechaActual(lli) \wedge lli_0 = lli\}$
Post ≡ $\{lli = acceso(lli_0, l, f)\}$
Complejidad: O(|l|)
Aliasing: No hay alias ya que no devuelve nada.

CANTLINKS(**in** lli: `linkLinkIt`, **in** c: `Categoria`)→ res: **nat**
Pre ≡ $\{c \in categorias(lli)\}$
Post ≡ $\{res =_{\text{obs}} cantLinks(lli, c)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

LINKSORDENADOSPORACCESOS(**in** lli: `linkLinkIt`, **in** c: `Categoria`)→ res: **itPuntLinks**
Pre ≡ $\{c \in categorias(lli)\}$
Post ≡ $\{res =_{\text{obs}} linksOrdenadosPorAccesos(lli, c)\}$
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 ≡ $\{l \in links(lli)\}$
Post ≡ $\{res =_{\text{obs}} esReciente?(s, l, f)\}$
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)`,
$cantAaccesosRecientes$: `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(*carDLink*) 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(longitud del link).

arrayCatLink 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 debera 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

$(\forall\, lli\colon \texttt{estrLLI})$ Rep(lli) ≡ true ⟺

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

2.2.2. Función de Abstracción

Abs : $e\colon \texttt{estrLLI}$ → linkLinkIt   Rep(e)

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

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

2.2.2.1. Funciones auxiliares

cantidadPorDia : estrLLI × Fecha × Lista(acceso) ⟶ nat
cantidadPorDia(e,f,ls) ≡ **if** f==prim(ls).dia **then**
prim(ls).cantAccesos
else
cantidadPorDia(e,f,fin(ls))
fi

todosLosLinks : secu(datosLink) ⟶ conj(Link)
todosLosLinks(s) ≡ **if** ∅?(s) **then** ∅ **else** Ag(prim(s).link,todosLosLinks(fin(s))) **fi**

sinRepetidos : secu(α) ⟶ bool
sinRepetidos(ls) ≡ **if** vacia?(ls) **then**
true
else
if esta?(prim(ls),fin(ls)) **then** false **else** sinRepetidos(fin(ls)) **fi**
fi

fechasCorrectas : fecha × secu(acceso) ⟶ bool
sinRepetidos(f,ls) ≡ **if** vacia?(ls) **then**
true
else
if prim(ls).dia >f **then** false **else** fechasCorrectas(f,fin(ls)) **fi**
fi

accesoOrdenadoNoRepetido : secu(acceso) ⟶ bool
sinRepetidos(ls) ≡ **if** long(ls)≤ 1 **then**
true
else
if prim(ls).dia≥ prim(fin(ls)).dia **then**
false
else
accesoOrdenadoNoRepetido(fin(ls))
fi
fi

cantidadDeAccesos : secu(acceso) ⟶ nat
cantidadDeAccesos(ls) ≡ **if** vacia?(ls) **then**
0
else
prim(ls).cantAccesos + cantidadDeAccesos(fin(ls))
fi

2.3. Algoritmos

Algoritmo 26 idameACat

function IDAMEACAT(**in** *lli*: `estrLLI`)→ *res*: `acat`
  res ← lli.arbolCategorias //O(1)
end function

Complejidad: O(1)

Algoritmo 27 iFechaActual

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

Complejidad: O(1)

Algoritmo 28 iLinks

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

Complejidad: O(1)

Algoritmo 29 iCategoriaLink

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

Complejidad: O(|l|)

Algoritmo 30 iFechaUltimoAcceso

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

Complejidad: O(|l|)

Algoritmo 31 iAccesosRecientesDia

```
function iAccesosRecientesDia(in lli: linkLinkIt, in l: Link, in f: Fecha)→ res: nat
    itAccesos accesos ← crearItAccesos(lli,l)                           //O(|l|)
    while haySiguiente(accesos) do                      //O(|accesos|) = O(1)
        if siguiente(accesos).dia == f then                             //O(1)
            res ← siguiente(accesos).cantAccesos                        //O(1)
        end if
        avanzar(accesos)                                                //O(1)
    end while
end function
```

Complejidad: O(|l|)

Algoritmo 32 iIniciarLLI

```
function iINICIARLLI(in ac: acat)→ res: estrLLI
    res.actual ← 1                                              //O(1)
    res.arbolCategorias ← ac                                    //O(1)
    nat c ← 0                                                   //O(1)
    res.arrayCatLinks ← crearArreglo(dameCantidad(ac))
                                                  //O(dameCantidad(ac))
    res.listaLinks ← vacia()                                    //O(1)
    res.linksInfo ← vacio()                                     //O(1)
    while c <dameCantidad(res.arbolCategorias) do
                                 //O(dameCantidad(res.arbolCategorias)))
        linksFamilia llist ← vacia()                            //O(1)
        res.arrayCatLinks[c] ← llist                            //O(1)
        c++                                                     //O(1)
    end while
end function
```

Complejidad: O(dameCantidad(res.arbolCategorias))

Algoritmo 33 iNuevoLink

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

Complejidad: O(|c|+|l|+alturaAC(ac))

Algoritmo 34 iAccederLLI

```
function iAccederLLI(in/out lli: estrLLI, in l: Link, in f: Fecha)
    if lli.actual ≠ f then                                                    //O(1)
        lli.actual ← f                                                        //O(1)
    end if
    puntero(datosLink) puntLink ← obtener(l,lli.linkInfo)                     //O(|l|)
    if ultimo((*puntLink).accesos).dia == f then                              //O(1)
        ultimo((*puntLink).accesos).cantAccesos ++                            //O(1)
    else
        agregarAtras((*puntLink).accesos,<f,1>)                               //O(1)
    end if
    if longitud((*puntLink).accesos) == 4 then                                //O(1)
        (*puntLink).cantAccesosRecientes -= prim((*puntLink).accesos).cantAccesos    //O(1)
        fin((*puntLink).accesos)                                              //O(1)
    end if
    (*puntLink).cantAccesosRecientes++                                        //O(1)
end function
```

Complejidad: O(|l|)

Algoritmo 35 iCantLinks

```
function iCantLinks(in lli: estrLLI, c: Categoria) → res: nat
    res ← longitud(lli.arrayCatLinks[id(lli.arbolCategorias,c)-1])           //O(|c|)
end function
```

Complejidad: O(|c|)

Algoritmo 36 iLinksOrdenadosPorAccesos

```
function iLinksOrdenadosPorAccesos(in lli: estrLLI,in c: Categoria) →
                                                                  res: itPuntLinks
    nat id ← id(lli.arbolCategorias,c)                                        //O(|c|)
    id ← id-1                                                                 //O(1)
    Fecha f ← 1                                                               //O(1)
    itPuntLinks itParaFecha ← crearItPuntLins(lli,id,f)                       //O(1)
    Fecha fecha ← ultFecha(itParaFecha)            //O(longitud(lli.arrayCatLinks[id]))
    Lista(puntero(datosLink)) listaOrdenada ← vacia()                         //O(1)
    if ¬estaOrdenada?(crearItPuntLins(lli,id,fecha)) then
                                                   //O(longitud(lli.arrayCatLinks[id]))
        while ¬vacia?(lli.arrayCatLinks[id]) do
            itPuntLinks itMax ← crearItPuntLins(lli,id,fecha)                 //O(1)
            itMax ← buscarMax(itMax,fecha)         //O(longitud(lli.arrayCatLinks[id]))
            agregarAtras(listaOrdenada,siguiente(itMax))                      //O(1)
            eliminarSiguiente(itMax)                                          //O(1)
        end while
        lli.arrayCatLinks[id] ← listaOrdenada                                 //O(1)
    end if
    res ← crearItPuntLins(lli,id,fecha)                                       //O(1)
end function
```

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

Algoritmo 37 iEsReciente

```
function iEsReciente(in lli: estrLLI, in l: Link, in f: Fecha)→ res: bool
    res ← f ≥ (fechaUltimoAcceso(lli,l)-2) ∧ f ≤ fechaUltimoAcceso(lli,l)        //O(|l)
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 correspondiende 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 listaOrdenada 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 listaOrdenada 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 listaOrdenada.

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 ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} crearIt(l)\}$
Complejidad: O(1)
Aliasing: No se deben modificar los elementos iterados por res.

HAYSIGUIENTE?(**in** it: `itLinks`)→ res: **bool**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} haySiguiente?(it)\}$
Complejidad: O(1)
Aliasing: No tiene.

SIGUIENTE(**in** it: `itLinks`)→ res: **Link**
Pre ≡ $\{haySiguiente?(it)\}$
Post ≡ $\{res =_{\text{obs}} \pi_1(siguiente(it))\}$
Complejidad: O(1)
Aliasing: El link se pasa por referencia, no debe ser modificado.

AVANZAR(**in/out** it: `itLinks`)

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

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

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 → boolean

($\forall\, it$: `estrITL`) Rep(it) ≡ true

2.5.4. Función de Abstracción

Abs : e: `estrITL` → itBi(α) Rep(e)

($\forall\, e$: `estrITL`) Abs(e) $=_{\text{obs}}$ it: itBi(α) |

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

2.5.5. Algoritmos

Algoritmo 38 iCrearItLinks

```
function iCrearItLinks(in l: lli)→ res: estrITL
    res ← crearIt(lli.listaLink)                    //O(1)
end function
```

Complejidad: O(1)

Algoritmo 39 iHaySiguiente?

```
function iHaySiguiente?(in e: estrITL)→ res: bool
    res ← haySiguiente(e)                    //O(1)
end function
```

Complejidad: O(1)

Algoritmo 40 iSiguiente

function ISIGUIENTE(**in** e: `estrITL`)→ res: `Link`
  res ← (siguiente(e)).link //O(1)
end function

Complejidad: O(1)

Algoritmo 41 iAvanzar

function IAVANZAR(**in/out** e: `estrITL`)
  avanzar(e) //O(1)
end function

Complejidad: O(1)

2.5.6. Analisis 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 tambíen 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`)→ res: **itPuntLinks**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} crearIt(lli, id, f)\}$
Complejidad: O(1)
Aliasing: No se deben modificar los elementos iterados por res.

HAYSIGUIENTE?(**in** it: `itPuntLinks`)→ res: **bool**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} haySiguiente?(it)\}$
Complejidad: O(1)
Aliasing: No tiene.

SIGUIENTELINK(**in** it: `itPuntLinks`)→ res: **Link**
Pre ≡ $\{haySiguiente?(it)\}$
Post ≡ $\{res =_{\text{obs}} (*siguiente(it)).link\}$
Complejidad: O(1)
Aliasing: El link se pasa por referencia, no debe ser modificado.

SIGUIENTECAT(**in** it: `itPuntLinks`)→ res: **Categoria**
Pre ≡ $\{haySiguiente?(it)\}$
Post ≡ $\{res =_{\text{obs}} (*siguiente(it)).catDLink\}$
Complejidad: O(1)
Aliasing: La categoria se pasa por referencia, no debe ser modificada.

SIGUIENTECANTACCESOSDELLINK(**in** it: `itPuntLinks`)→ res: **nat**
Pre ≡ $\{haySiguiente?(it)\}$
Post ≡ $\{res =_{\text{obs}} cantAccesosDesde(it)\}$
Complejidad: O(1)
Aliasing: No tiene

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

ELIMINARSIGUIENTE(**in/out** it: `itPuntLinks`)
Pre ≡ $\{it =_{\text{obs}} it_0 \wedge haySiguiente?(it)\}$
Post ≡ $\{it =_{\text{obs}} eliminarSiguiente(it_0)\}$
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

Rep : estrITPL $\rightarrow$ boolean

$(\forall\, it \colon$ `estrITPL`$)$ Rep(it) $\equiv$ true

2.6.4. Función de Abstracción

Abs : e: `estrITPL` $\rightarrow$ itBi(α) Rep(e)

$(\forall\, e \colon$ `estrITPL`$)$ Abs(e) $=_{\text{obs}}$ it: itBi(α) |

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

2.6.5. Algoritmos

Algoritmo 42 iCrearItPuntLinks

```
function iCrearItLinks(in e: estrLLI, in id: nat) → res: estrITPL
    res ← crearIt(e.arrayCatLinks[id])                    //O(1)
end function
```

Complejidad: O(1)

Algoritmo 43 iHaySiguiente?

```
function iHaySiguiente?(in e: estrITPL) → res: bool
    res ← haySiguiente(e)                                 //O(1)
end function
```

Complejidad: O(1)

Algoritmo 44 iSiguiente

```
function iSiguiente(in e: estrITPL) → res: DatosLink
    res ← siguiente(e)                                    //O(1)
end function
```

Complejidad: O(1)

Algoritmo 45 iSiguienteLink

```
function iSiguienteLink(in e: estrITPL) → res: Link
    res ← (*siguiente(e)).link                            //O(1)
end function
```

Complejidad: O(1)

Algoritmo 46 iSiguienteCat

```
function iSiguienteCat(in e: estrITPL) → res: Categoria
    res ← (*siguiente(e)).catDLink                        //O(1)
end function
```

Complejidad: O(1)

Algoritmo 47 iSiguienteCantidadAccesosDelLink

```
function iSiguienteCantidadAccesosDelLink(in e: estrITPL)→ res: int
    res ← cantAccesosDesde(e.fecha)                                   //O(1)
end function
```

Complejidad: O(1)

Algoritmo 48 iAvanzar

```
function iavanzar(in/out e: estrITPL)
    avanzar(e)                                                        //O(1)
end function
```

Complejidad: O(1)

Algoritmo 49 iEliminarSiguiente

```
function ieliminarSiguiente(in/out e: estrITPL)
    eliminarSiguiente(e)                                              //O(1)
end function
```

Complejidad: O(1)

Algoritmo 50 iBuscarMax

```
function ibuscarMax(in it: estrITPL, in f: Fecha)→ res: itPuntLinks
    res ← copiarIt(it)                                                //O(1)
    while haySiguiente(it) do                     //O(longitud(siguientes(it)))
        if cantAccesosDesde(it,f) >cantAccesosDesde(res,f) then       //O(1)
            res ← it                                                  //O(1)
        end if
        avanzar(it)                                                   //O(1)
    end while
end function
```

Complejidad: O(longitud(siguientes(it)))

Algoritmo 51 iUltFecha

```
function iultFecha(in it: estrITPL)→ res: Fecha
    res ← (ultimo((*siguiente(it).accesos)).dia                       //O(1)
    while haySiguiente(it) do                     //O(longitud(siguientes(it)))
        if (ultimo((*siguiente(it).accesos)).dia >res then            //O(1)
            res ← (ultimo((*siguiente(it).accesos)).dia               //O(1)
        end if
        avanzar(it)                                                   //O(1)
    end while
end function
```

Complejidad: O(longitud(siguientes(it)))

Algoritmo 52 iCantAccesosDesde

```
function ICANTACCESOSDESDE(in it: estrITPL, in f: Fecha)→ res: nat
    itAccesos itAcc ← (*siguiente(it)).accesos                                //O(1)
    res ← 0                                                                   //O(1)
    while haySiguiente(itAcc) do                                              //O(1)
        if (siguiente(itAcc)).dia ≤ f ∧ (siguiente(itAcc)).dia ≤ f-2 then     //O(1)
            res ← res + (siguiente(it)).cantA                                 //O(1)
        end if
        avanzar(itAcc)                                                        //O(1)
    end while
end function
```

Complejidad: O(1)

Algoritmo 53 iEstaOrdenada?

```
function IESTAORDENADA?(in it: estrITPL, in f: Fecha)→ res: bool
    res ← true                                                                //O(1)
    nat aux ← cantAccesosDesde(it,f)                                          //O(1)
    avanzar(it)                                                               //O(1)
    while haySiguiente(it) do                              //O(longitud(siguientes(it)))
        if cantAccesosDesde(it,f) >aux then                                   //O(1)
            res ← false                                                       //O(1)
        end if
        aux ← cantAccesosDesde(it,f)                                          //O(1)
        avanzar(it)                                                           //O(1)
    end while
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(longitud(siguientes(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(longitud(siguientes(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(longitud(siguientes(it)))

   Orden Total: O(longitud(siguientes(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 costante, 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(longitud(siguientes(it))).

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

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

Orden Total: O(longitud(siguientes(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

Rep : estrITA → boolean

($\forall\, it$: `estrITA`) Rep(it) ≡ true

2.7.3. Función de Abstracción

Abs : e: `estrITA` → itBi(α) Rep(e)

($\forall\, e$: `estrITA`) Abs(e) $=_{\text{obs}}$ it: itBi(α) |

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

2.7.4. Algoritmos

Algoritmo 54 iCrearItAccesos

```
function iCrearItAccesos(in l: Lista(acceso))→ res: estrITA
    res ← crearIt(l)                                        //O(1)
end function
```

Complejidad: O(1)

Algoritmo 55 iHaySiguiente?

```
function iHaySiguiente?(in e: estrITA)→ res: bool
    res ← haySiguiente(e)                                   //O(1)
end function
```

Complejidad: O(1)

Algoritmo 56 iSiguiente

```
function ISIGUIENTE(in e: estrITA)→ res: Acceso
    res ← siguiente(e)                                //O(1)
end function
```

Complejidad: O(1)

Algoritmo 57 iAvanzar

```
function IAVANZAR(in/out e: estrITA)
    avanzar(e)                                        //O(1)
end function
```

Complejidad: O(1)

2.7.5. Analisis 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 tambíen 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*: **diccTrie(c,s)**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} vacio()\}$
Complejidad: O(1)
Aliasing: No tiene

DEFINIR(**in** c: `string`, **in** s: α, **in/out** d: `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: `string`, **in** d: `diccTrie(s)`)$\rightarrow$ *res*: **bool**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} def?(c, d)\}$
Complejidad: O(|c|)
Aliasing: No tiene

OBTENER(**in** c: `string`, **in** d: `diccTrie(s)`)$\rightarrow$ *res*: α
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

Rep : estrDT → boolean

$(\forall\, e\colon \texttt{estrDT})$ Rep(e) $\equiv$ true $\Longleftrightarrow$

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

3.2.1.3. Funciones auxiliares

EncAEstrDTEnNMov : estrDT × estrDT × Nat ⟶ Bool
EncAEstrDTEnNMov(buscado,actual,n) ≡ **if** (n = 0) **then**
  EstaEnElArregloActual?(buscado,actual,255)
else
  RecurrenciaConLosHijos(buscado,actual, n-1,255)
fi

EstaEnElArregloActual? : estrDT × estrDT × nat ⟶ Bool
EstaEnElArregloActual?(buscado,actual,n) ≡ **if** (n=0) **then**
  ((*actual).Arreglo[0] = buscado)
else
  ((*actual).Arreglo[n] = buscado) ∨ (EstaEnElArregloActual? (buscado,actual,n-1))
fi

RecurrenciaConLosHijos : estrDT × estrDT × nat × nat ⟶ Bool

RecurrenciaConLosHijos(buscado,actual,n,i) ≡ **if** (i = 0) **then**
EncAEstrDTEnNMov(buscado,(*actual).Arreglo[0],n)
else
EncAEstrDTEnNMov(buscado,
(*actual).Arreglo[i],n) ∨
(RecurrenciaConLosHijos(buscado,actual,n,i-
1)
fi

SonTodosNullOLosHijosLoSon : estrDT ⟶ Bool
SonTodosNullOLosHijosLoSon(e) ≡ Los256SonNull(e,255) ∨ BuscarHijosNull (e, 255)

Los256SonNull : estrDT × nat ⟶ Bool
Los256SonNull(e,i) ≡ **if** (i = 0) **then**
((*e).Arreglo[0] = null)
else
((*e).Arreglo[i] = null) ∧ Los256SonNull(e, i-1)
fi

BuscarHijosNull : estrDT × nat ⟶ Bool
BuscarHijosNull(e,i) ≡ **if** (i = 0) **then**
((*e).Arreglo[0] = null) ∨ SonTodosNullOLosHijosLoSon((*e).Arreglo[0])
else
(((*e).Arreglo[i] = null) ∨ SonTodosNullOLosHijosLo-Son((*e).Arreglo[i])) ∧ BuscarHijosNull(e,i-1)
fi

3.2.2. Función de Abstracción

Abs : e: `estrDT` → diccT(c,α) Rep(e)

($\forall\, e$: `estrDT`) Abs(e) $=_{\text{obs}}$ d: diccT(c,α) |

1. $(\forall\, c\colon \texttt{clave}) def?(c,d) =_{\text{obs}} estaDefinido?(c,e) \wedge_{\text{L}}$
2. $(\forall\, c\colon \texttt{clave}) def?(c,d) \Rightarrow obtener(c,d) =_{\text{obs}} ObtenerS(c, *(e))$

3.2.2.1. Funciones auxiliares

estaDefinido? : string × estrDT ⟶ bool
estaDefinido?(c,e) ≡ **if** (e==Null) **then** false **else** NodoDef?(c,*(e)) **fi**

NodoDef? : string × Nodo ⟶ bool
NodoDef?(c,n) ≡ **if** (vacia?(c)) **then**
true
else
if (n.arreglo[numero(prim(c))] ≠ Null) **then**
NodoDef?(fin(c),*(n.arreglo[numero(prim(c))]))
else
false
fi
fi

numero : char ⟶ nat
numero(char) ≡ char - a

ObtenerS : string × Nodo ⟶ α
ObtenerS(c,n) ≡ **if** (vacia?(c)) **then**
*(n.significado)
else
ObtenerS(fin(c),*(n.arreglo[numero(prim(c))]))
fi

3.3. Algoritmos

Algoritmo 58 iVacio

```
function IVACIO→ res: estrDT
    var n: Puntero(Nodo)                                   //O(1)
    n ← Null                                               //O(1)
    res ← n                                                //O(1)
end function
```

Complejidad: O(1)

Algoritmo 59 iDefinir

```
function IDEFINIR(in c: string, in s: α, in/out e: estrDT)
    if (e = Null) then                                     //O(1)
        var n: Nodo                                        //O(1)
        n ← iNuevoNodo                                     //O(1)
        e ← &(n)                                           //O(1)
    end if
    var n1: Nodo                                           //O(1)
    n1 ← *(e)                                              //O(1)
    var i: nat                                             //O(1)
    i ← 0                                                  //O(1)
    while (i <|c|) do                                      //O(|c|)
        if (n1.arreglo[iNumero(c[i])] = Null) then         //O(1)
            var n2: Nodo                                   //O(1)
            n2 ← iNuevoNodo                                //O(1)
            n1.arreglo[iNumero(c[i])] ← &(n2)              //O(1)
        end if
        n1 ← *(n1.arreglo[iNumero(c[i])])                  //O(1)
        i++                                                //O(1)
    end while
    n1.significado ← s                                     //O(1)
end function
```

Complejidad: O(|c|)

Algoritmo 60 iNuevoNodo

```
function INUEVONODO→ res: Nodo
    var n: Nodo                                    //O(1)
    n.significado ← Null                           //O(1)
    for (i: nat←0; i<256; i++) do                  //O(256*1)
        n.arreglo[i] ← Null                        //O(1)
    end for
    res ← n                                        //O(1)
end function
```

Complejidad: O(1)

Algoritmo 61 iNumero

```
function INUMERO(c1: char)→ res: nat
    var c2: char                                   //O(1)
    c2 ← a                                         //O(1)
    res ← (c1 - c2)                                //O(1)
end function
```

Complejidad: O(1)

Nota: Se le resta el char “a” para que al tomar la representación ASCII de los char evaluen como “a=0”, “b=1”, “c=2”, etc.

Algoritmo 62 iDef?

```
function IDEF?(in c: string, in e: estr)→ res: bool
    if (e ≠ Null) then                             //O(1)
        var n: Nodo                                //O(1)
        n ← *(e)                                   //O(1)
        var i: nat                                 //O(1)
        var i ← 0                                  //O(1)
        res ← true                                 //O(1)
        while (i <|c|) do                          //O(|c|)
            if (n.arreglo[numero(c[i])] ≠ Null) then   //O(1)
                n ← *(n.arreglo[numero(c[i]))]     //O(1)
            else
                res ← false                        //O(1)
                i ← long(c)                        //O(1)
            end if
            i++                                    //O(1)
        end while
    else
        res ← false                                //O(1)
    end if
end function
```

Complejidad: O(|c|)

Algoritmo 63 iObtener

```
function iObtener(in c: string, in e: estr) → res: α
    var n: Nodo                                   //O(1)
    n ← *(e)                                      //O(1)
    var i: nat                                    //O(1)
    var i ← 0                                     //O(1)
    while (i <|c|) do                             //O(|c|)
        n ← *(n.arreglo[numero(c[i]))]            //O(1)
        i++                                       //O(1)
    end while
    res ← n.significado                           //O(1)
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(|string|*O(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(|string|*(O(1+1+1+1+)+O(1+1))+O(1) = O(1)+O(|string|)+O(1) = **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)=**O(1)**

4. **iNumero**

   Se crea una variable char y se le asigna un valor en O(1). Luego se asigna a res la resta de 2 chars en O(1). Como res es un nat se asigna el número que representan dichos char.

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

5. **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(|string|*(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(|string|*(O(1+1)+O(1))=**O(|string|)**

6. **iObtener**

   Se crean variables y se les asigna valor en O(1). Luego se realiza un loop iterando la longitud del string en O(|string|*O(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(|string|*O(1+1))+O(1)=**O(|string|)**