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
Almansi, Emilio	123/12	lalala@lalala.com
Chapresto, Matías	123/12	lalala@lalala.com
Vileriño, Silvio	123/12	lalala@lalala.com
Alguno más si dios quiere	123/12	lalala@lalala.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo ArbolCategorias **3**
1.1. Interfaz . . . 3
1.2. Representación . . . 5
1.2.1. Invariante de Representación . . . 5
1.2.1.1. El Invariante Informalmente . . . 5
1.2.1.2. El Invariante Formalmente . . . 6
1.2.2. Función de Abstracción . . . 7
1.2.2.1. Funciones auxiliares . . . 7
1.3. Algoritmos . . . 7
1.4. Analisis de complejidades . . . 10
1.5. Iterador de Categorias . . . 12
1.5.1. Interfaz . . . 12
1.5.2. Representación . . . 13
1.5.3. Invariante de Representación . . . 13
1.5.3.1. El Invariante Formalmente . . . 13
1.5.4. Función de Abstracción . . . 13
1.5.5. Algoritmos . . . 14
1.5.6. Analisis de complejidades . . . 14

1. Módulo ArbolCategorias

1.1. Interfaz

parámetros formales

géneros acat
se explica con: ArbolDeCategorias

Operaciones

CATEGORIASAC(**in** ac: `acat`)→ res: **itCategorias**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} crearItUni(categorias(ac))\}$
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.

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 ≡ $\{res =_{\text{obs}} crearItUni(hijos(ac, c))\}$
Complejidad: O(|c|)
Aliasing: No se debe modificar nada de lo iterado por res.

PADREAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **Categoria**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} padre(ac, c)\}$
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 ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} altura(ac)\}$
Complejidad: O(1)
Aliasing: No tiene.

PREDECESORES(**in** ac: `acat`, **in** c: `Categoria`)→ res: **itFamilia**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} crearItUni(predecesores(ac, c))\}$
Complejidad: O(|c|)
Aliasing: res itera referencias a las categorias correspondientes.

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

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

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

ESSUBCATEGORIA(**in** ac: `acat`, **in** c: `categoria`, **in** h: `categoria`)→ res: **bool**
Pre ≡ $\{esta?(c, ac) \land esta?(h, ac)\}$
Post ≡ $\{res =_{\text{obs}} esSubCategoria(ac, c, h)\}$
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 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 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 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*' 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 + '*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

$(\forall\, ac$: `estrAC`$)$ Rep(ac) $\equiv$ true $\Longleftrightarrow$

1. $(\forall\, c$: `Categoria`$)(def?(c, e.familia)) \Leftrightarrow (*obtener(c, e.familia)).categoria = c \wedge_{\text{L}}$
2. $(\forall\, c_1$: `Categoria`$)(def?(c_1, e.familia)) \Leftrightarrow ((c_1 == e.raiz) \vee$
   $((\exists\, c_2$: `Categoria`$)(def?(c_2, e.familia)) \wedge_{\text{L}} c_1 \in (*obtener(c_2, e.familia)).hijos)) \wedge_{\text{L}}$
3. $(\forall\, c$: `Categoria`$)(def?(c, e.familia) \Leftrightarrow$
   $(((\exists\, d$: `datosCat`$)esta?(d, e.categorias) \wedge d.categoria == c) \wedge_{\text{L}} d == obtener(c, e.familia)))$
   $\wedge_{\text{L}}$
4. $(\forall\, c_1, c_2$: `Categoria`$)(def?(c_1, e.familia)) \wedge (def?(c_2, e.familia)) \Rightarrow_L$
   $c_2 \in *((obtener(c_1, e.familia))).hijos \Leftrightarrow$
   $(*(*(obtener(c_2, e.familia))).padre).categoria = c_1 \wedge_{\text{L}}$
5. $e.cantidad = longitud(e.categorias) \wedge_{\text{L}}$
6. $(\forall\, c$: `categoria`$)(def?(c, e.familia)) \wedge c = e.raiz \Rightarrow_L$
   $(*(obtener(c, e.familia))).altura = 1 \wedge (*(obtener(c, e.familia))).id = 1 \wedge_{\text{L}}$
7. $(\forall\, c$: `Categoria`$)(def?(c, e.familia)) \Rightarrow_L (*obtener(c, e.familia)).altura \leq e.alturaMax$
   $\wedge_{\text{L}}$
8. $(\exists\, c$: `Categoria`$)(def?(c, e.familia)) \wedge_{\text{L}} *((obtener(c, e.familia))).altura = e.alturaMax$
   $\wedge_{\text{L}}$
9. $(\forall\, c_1, c_2$: `Categoria`$)(def?(c_1, e.familia)) \wedge (def?(c_2, e.familia)) \wedge_{\text{L}}$
   $((\exists\, d$: `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. $(\forall\, c$: `Categoria`$)(def?(c, e.familia)) \Rightarrow_L (*(obtener(c, e.familia))).id \leq e.cant \wedge_{\text{L}}$
11. $(\forall\, c_1, c_2$: `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` → 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) ⟶ conj(categoria)
todasLasCategorias(cs) ≡ **if** vacia?(cs) **then**
∅
else
Ag((prim(cs)).categoria,todasLasCategorias(fin(cs)))
fi

predecesores : arbolCategoriasac × Categoriac ⟶ Conj(categoria) {c ∈ categorias(ac)}
predecesores(ac,c) ≡ predecesoresAux(ac,categorias(ac),c)

predecesoresAux : arbolCategoriasac × Conj(Categoria)cc × Categoriac ⟶ Conj(categoria)
{c ∈ categorias(ac) ∧ cc ⊆ categorias(ac)}
predecesoresAux(ac,cc,c) ≡ **if** ∅? cc **then**
∅
else
if esSubCategoria(ac,dameUno(cc),c) **then**
Ag(dameUno(cc),predecesoresAux(ac,sinUno(cc),c))
else
predecesoresAux(ac,sinUno(cc),c)
fi
fi

1.3. Algoritmos

Algoritmo 1 iCategoriasAC

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

Complejidad: O(1)

Algoritmo 2 iRaizAC

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

Complejidad: O(1)

Algoritmo 3 iDameCantidad

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

Complejidad: O(1)

Algoritmo 4 iIdAC

```
1: function iId(in ac: estrAC, in c: Categoria)→ res: nat
2:     res ← ((*obtener(c,ac.familia)).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 iPredecesores

```
1: function iPredecesores(in ac: estrAC, in c: Categoria)→ res: itFamilia
2:     res ← crearItFamilia(ac,c)                           //O(|c|)
3: end function
```

Complejidad: O(|c|)

Algoritmo 10 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 11 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 12 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 13 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| + 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. **iPredecesores**

   Devuelve un iterador a los predecesores de la categoría, incluyendola, en O(|c|) que es lo que le cuesta conseguir el iterador.

   Orden Total: O(|c|)

10. **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 la categoria 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 la categoria 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|)**

11. **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 categoria. 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|)**

12. **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|)

13. **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 Unidireccional de Categoria

Operaciones

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

SIGUIENTE(**in** *it*: `itCategorias`)→ *res*: **Categoria**
Pre ≡ {$HayMas?(it)$}
Post ≡ {$res =_{\text{obs}} Actual(it)$}
Complejidad: O(1)
Aliasing: La categoría se pasa por referencia, no debe ser modificada.

AVANZAR(**in/out** *it*: `itCategorias`)
Pre ≡ {$it =_{\text{obs}} it_0 \wedge HayMas?(it)$}
Post ≡ {$it =_{\text{obs}} avanzar(it_0)$}
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

(∀ *it*: `estrITC`) Rep(it) ≡ true

1.5.4. Función de Abstracción

Abs : *e*: `estrITC` → itUni(Categoria) {Rep(e)}

(∀ *e*: `estrITC`) Abs(e) =$_{\text{obs}}$ it: itUni(Categoria) |

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

1.5.5. Algoritmos

Algoritmo 14 iCrearItCategorias

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

Complejidad: O(1)

Algoritmo 15 iHaySiguiente?

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

Complejidad: O(1)

Algoritmo 16 iSiguiente

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

Complejidad: O(1)

Algoritmo 17 iAvanzar

```
1: function IAVANZAR(in/out e: estrITC)
2:     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 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)