Algoritmos y Estructuras de Datos II

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

Trabajo Práctico de Especificación

Grupo 1

Integrante	LU	Correo electrónico
Bálsamo, Facundo	874/10	facundobalsamo@gmail.com
Lasso, Nicolás	892/10	lasso.nico@gmail.com
Rodríguez, 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		

1. TAD LinkLinkIT

TAD LinkLinkIT

gros **lli**

exporta generadores, categorias, links, categoriaLink, fechaActual, fechaUltimoAcceso, accesosRecientesDia, esReciente?, accesosRecientes, linksOrdenadosPorAccesos, cantLinks

usa BOOL, NAT, CONJUNTO, SECUENCIA, ARBOLCATEGORIAS

observadores bcos

categorias	: lli s	$\longrightarrow$ acat		
links	: lli s	$\longrightarrow$ conj(link)		
categoriaLink	: lli $\times$ link	$\longrightarrow$ categoria		
fechaActual	: lli	$\longrightarrow$ fecha		
fechaUltimoAcceso	: lli s $\times$ link l	$\longrightarrow$ fecha		$\{l \exists links(s)\}$
accesosRecientesDia	: lli s $\times$ link l $\times$ fecha f	$\longrightarrow$ nat		

generadores

iniciar	: acat ac	$\longrightarrow$ **lli**		
nuevoLink	: lli s $\times$ link l $\times$ categoria c	$\longrightarrow$ **lli**		$\{(l \exists links(s)) \wedge esta?(c, categorias(s))\}$
acceso	: lli s $\times$ link l $\times$ fecha f	$\longrightarrow$ **lli**		$\{l \exists links(s) \wedge f \geq fechaActual(s)\}$

otras operaciones

esReciente?	: lli s $\times$ link l $\times$ fecha f	$\longrightarrow$ bool		$\{l \exists links(s)\}$
accesosRecientes	: lli s $\times$ categoria c $\times$ link l	$\longrightarrow$ nat		$\{esta?(c, categorias(s)) \wedge l \exists links(s) \wedge esSubCategoria(categorias(s), c, categoriaLink(s, l))\}$
linksOrdenadosPorAccesos	: lli s $\times$ categoria c	$\longrightarrow$ secu(link)		$\{esta?(c, categorias(s))\}$
cantLinks	: lli s $\times$ categoria c	$\longrightarrow$ nat		$\{esta?(c, categorias(s))\}$
menorReciente	: lli s $\times$ link l	$\longrightarrow$ fecha		$\{l \exists links(s)\}$
diasRecientes	: lli s $\times$ link l	$\longrightarrow$ fecha		$\{l \exists links(s)\}$
diasRecientesDesde	: lli s $\times$ link l	$\longrightarrow$ fecha		$\{l \exists links(s)\}$
linksCategoriasOHijos	: lli s $\times$ categoria c	$\longrightarrow$ conj(link)		$\{esta?(c, categorias(s))\}$
filtrarLinksCategoriaOHijos	: lli s $\times$ categoria c $\times$ conj(link) ls	$\longrightarrow$ conj(link)		$\{esta?(c, categorias(s)) \wedge ls \subseteq links(s)\}$
diasRecientesParaCategorias	: lli s $\times$ categoria c	$\longrightarrow$ conj(fecha)		$\{esta?(c, categorias(s))\}$
linkConUltimoAcceso	: lli s $\times$ categoria c $\times$ conj(link) ls	$\longrightarrow$ link		$\{esta?(c, categorias(s)) \wedge \emptyset?(ls) \wedge ls \subseteq linksCategoriasOHijos(s, c)\}$
sumarAccesosRecientes	: lli s $\times$ link l $\times$ conj(fecha) fs	$\longrightarrow$ nat		$\{l \exists links(s) \wedge fs \subseteq diasRecientes(s, l)\}$
linksOrdenadosPorAccesosAux	: lli s $\times$ categoria c $\times$ conj(link) ls	$\longrightarrow$ secu(link)		$\{esta?(c, categorias(s)) \wedge ls \subseteq linksCategoriasOHijos(s, c)\}$
linkConMasAccesos	: lli s $\times$ categoria c $\times$ conj(link) ls	$\longrightarrow$ link		$\{esta?(c, categorias(s)) \wedge ls \subseteq linksCategoriasOHijos(s, c)\}$
β	: bool b	$\longrightarrow$ nat		

axiomas $\forall$linklinkITit, it'
$\forall$arbolDeCategoriasa
$\forall$categoriac
$\forall$linkl
$\forall$fechaf

∀conj(categoria)cc

categorias(iniciar(ac)) ≡ ac
categorias(nuevoLink(s,l,c)) ≡ categorias(ac)
categorias(acceso(s,l,f)) ≡ categorias(ac)

links(iniciar(ac)) ≡ ∅
links(nuevoLink(s,l,c)) ≡ Ag(l,links(s))
links(acceso(s,l,f)) ≡ links(s)

categoriaLink(nuevoLink(s,l,c),l') ≡ **if** $l == l'$ **then** c **else** $categoriaLink(s, l')$ **fi**
categoriaLink(acceso(s,l,f),l') ≡ categoriaLink(s,l')

fechaActual(iniciar(ac)) ≡ 0
fechaActual(nuevoLink(s,l,c)) ≡ fechaActual(s)
fechaActual(acceso(s,l,f)) ≡ f

fechaUltimoAcceso(nuevoLink(s,l,c),l') ≡ **if** l==l' **then** fechaActual(s) **else** fechaUltimoAcceso(s,l') **fi**
fechaUltimoAcceso(acceso(s,l,f),l') ≡ fechaUltimoAcceso(s,l')

menorReciente(s,l) ≡ max(fechaUltimoAcceso(s, l) + 1,diasRecientes) - diasRecientes
esReciente?(s,l,f) ≡ menorReciente(s,l) ≤ f ∧ f ≤ fechaUltimoAcceso(s,l)
accesoRecienteDia(nuevoLink(s,l,c),l',f) ≡ **if** l==l' **then** 0 **else** accesoRecienteDia(s,l',f) **fi**
accesoRecienteDia(acceso(s,l,f),l',f') ≡ $\beta(l == l' \wedge f == f')$ + **if** esReciente?(s,l,f') **then** accesoRecienteDia(s,l',f') **else** 0 **fi**
accesosRecientes(s, c, l) ≡ sumarAccesosRecientes(s, l, diasRecientesParaCategoria(s, c) ∩ diasRecientes(s, l))
linksOrdenadosPorAccesos(s, c) ≡ linksOrdernadosPorAccesosAux(s, c, linksCategoriaOHijos(s, c))
linksOrdenadosPorAccesosAux(s,c,ls) ≡ **if** ∅?(ls) **then**
∅
else
linkConMasAccesos(s, c, ls) • linksOrdernadosPorAccesosAux(s, c, ls - linkConMasAccesos(s, c, ls))
fi
linkConMasAccesos(s, c, ls) ≡ **if** #ls==1 **then**
dameUno(ls)
else
if accesosRecientes(s,c,dameUno(ls)) > accesosRecientes(s,c,linkConMasAccesos(s,c,sinUno(ls))) **then**
dameUno(ls)
else
linkConMasAccesos(s,c,sinUno(ls))
fi
fi

cantLinks(s, c) $\equiv$ #linksCategoriaOHijos(s, c)

diasRecientes(s, l) $\equiv$ diasRecientesDesde(s, l, menorReciente(s, l))

diasRecientesDesde(s, l, f) $\equiv$ **if** esReciente?(s, l, f) **then** Ag(f, diasRecientesDesde(s, l, f+1)) **else** $\emptyset$ **fi**

linksCategoriaOHijos(s, c) $\equiv$ filtrarLinksCategoriaOHijos(s, c, links(s))

filtrarLinksCategoriaOHijos(s, c, ls) $\equiv$ **if** $\emptyset$?(ls) **then**
$\emptyset$
else
(**if** esSubCategoria(categorias(s),c,categoriaLink(s,dameUno(ls)))
then
dameUno(ls)
else
$\emptyset$
fi) $\cup$ filtrarLinksCategoriaOHijos(s, c, siunUno(ls))
fi

diasRecientesParaCategoria(s, c) $\equiv$ **if** $\emptyset$?(linksCategoriaOHijos(s,c)) **then**
$\emptyset$
else
diasRecientes(s, linkConUltimoAcceso(s, c, linksCategoriaOHijos(s,c)))
fi

sumarAccesosRecientes(s, l, fs) $\equiv$ **if** $\emptyset$?(fs) **then**
0
else
accesosRecientesDia(s, l, dameUno(f)) + sumarAccesosRecientes(s, l, sinUno(fs))
fi

β(b) $\equiv$ **if** b **then** 1 **else** 0 **fi**

Fin TAD

1.0.1. Modulo de linkLinkIT

generos: *lli*
usa: bool, nat, conjunto, secuencia, arbolCategorias
se explica con: TAD linkLinkIT
géneros: lli

1.0.2. Operaciones Básicas

categorias (**in** s: estrLLI) $\longrightarrow$ res: ac

Pre $\equiv$ true
Post $\equiv$ res=$_{obs}$ categorias(s)
Complejidad : O(#categorias(s))
Descripción : Devuelve el arbol de categorias con todas las categorias del sistema
Aliasing: res es modificable si y solo si s es modificable.

links (**in** s: estrLLI) $\longrightarrow$ res: conj(link)

Pre $\equiv$ true
Post $\equiv$ res=$_{obs}$ links(s)
Complejidad : O(#links(s))
Descripción : Devuelve todos los links del sistema
Aliasing: res es modificable si y solo si s es modificable.

categoriaLink (**in** s: estrLLI, **in** l: link) $\longrightarrow$ res: categoria

Pre $\equiv$ true
Post $\equiv$ res$=_{obs}$ categoriaLink(s,l)
Complejidad : O(|l|)
Descripción : Devuelve la categoria del link ingresado
Aliasing: res es modificable si y solo si s es modificable.

fechaActual (**in** s: estrLLI) $\longrightarrow$ res: fecha

Pre $\equiv$ true
Post $\equiv$ res$=_{obs}$ fechaActual(s)
Complejidad : O(1)
Descripción : Devuelve la fecha actual
Aliasing: res es modificable si y solo si s es modificable.

fechaUltimoAcceso (**in** s: estrLLI, **in** l: link) $\longrightarrow$ res: fecha

Pre $\equiv$ l $\in$ links(s)
Post $\equiv$ res$=_{obs}$ fechaUltimoAcceso(s,l)
Complejidad : O(1)
Descripción : Devuelve la fecha de ultimo acceso al link
Aliasing: res es modificable si y solo si s es modificable.

accesosRecientesDia (**in** s: estrLLI, **in** l: link, **in** f: fecha) $\longrightarrow$ res: nat

Pre $\equiv$ l $\in$ links(s)
Post $\equiv$ res$=_{obs}$ accesosRecientesDia(s,l,f)
Complejidad : O(1)
Descripción : Devuelve la cantidad de accesos a un link un cierto dia

inicar (**in** ac: estrAC) $\longrightarrow$ res: lli

Pre $\equiv$ true
Post $\equiv$ res$=_{obs}$ iniciar(ac)
Complejidad : O(#categorias(ac))
Descripción : crea un sistema dado un arbol ac de categorias
Aliasing: res es modificable si y solo si ac es modificable.

nuevoLink (**in/out** s: estrLLI, **in** l: link , **in** c: categoria)

Pre $\equiv$ c $\in$ categorias(s) $\wedge$ s_0 $=_{obs}$ s $\wedge$ l $\notin$ links(s)
Post $\equiv$ s$=_{obs}$ nuevoLink(s_0,l,c)
Complejidad : O(|l|+|c|+h)
Descripción : Agregar un link al sistema

acceso (**in/out** s: estrLLI, **in** l: link , **in** f: fecha)

Pre $\equiv$ l $\in$ links(s) $\wedge$ f $\geq$ fechaActual(s) $\wedge$ s_0 $=_{obs}$ s
Post $\equiv$ s$=_{obs}$ acceso(s_0,l,f)
Complejidad : O(|l|)
Descripción : Acceder a un link del sistema

esReciente? (**in** s: estrLLI, **in** l: link , **in** f: fecha) $\longrightarrow$ res: bool

Pre $\equiv$ l $\in$ links(s)

Post ≡ res=$_{obs}$ esReciente?(s,l,f)
Complejidad : O(|l|)
Descripción : Chequea si el acceso fue reciente

accesosRecientes (**in** s: estrLLI, **in** c: categoria **in** l: link) ⟶ res: nat

Pre ≡ c ∈ categorias(s) ∧ l ∈ links(s)
Post ≡ res=$_{obs}$ accesosRecientes(s,c,l)
Complejidad : O(1)
Descripción : Devuelve la cantidad de accesos recientes del link ingresado

linksOrdenadosPorAccesos (**in** s: estrLLI, **in** c: categoria) ⟶ res: secu(link)

Pre ≡ c ∈ categorias(s)
Post ≡ res=$_{obs}$ linksOrdenadosPorAccesos(s,c)
Complejidad : O(n^2)
Descripción : Devuelve la cantidad de accesos recientes del link ingresado

cantlinks (**in** s: estrLLI, **in** c: categoria) ⟶ res: nat

Pre ≡ c ∈ categorias(s)
Post ≡ res=$_{obs}$ cantlinks(s,c)
Complejidad : O(|c|)
Descripción : Devuelve la cantidad de links de la categoria c

menorReciente (**in** s: estrLLI, **in** l: link) ⟶ res: fecha

Pre ≡ l ∈ links(s)
Post ≡ res=$_{obs}$ menorReciente(s,l)
Complejidad : O(1)
Descripción : Devuelve la fecha menor mas reciente

diasRecientes (**in** s: estrLLI, **in** l: link) ⟶ res: fecha

Pre ≡ l ∈ links(s)
Post ≡ res=$_{obs}$ diasRecientes(s,l)
Complejidad : O(|l|)
Descripción : Devuelve la fecha reciente del link

diasRecientesDesde (**in** s: lli, **in** l: link) ⟶ res: fecha

Pre ≡ l ∈ links(s)
Post ≡ res=$_{obs}$ diasRecientesDesde(s,l)
Complejidad : O(|l|)
Descripción : Devuelve la fecha reciente del link

diasRecientesParestrACegorias (**in** s: lli, **in** c: categoria) ⟶ res: conj(fecha)

Pre ≡ c ∈ categorias(s)
Post ≡ res=$_{obs}$ diasRecientesParaCategorias(s,c)
Complejidad : O(*|l|)
Descripción : Devuelve el conjunto de fechas recientes de la categoria c

linkConUltimoAcceso (**in** s: lli, **in** c: categoria,**in** ls: conj(link)) ⟶ res: link

Pre ≡ c ∈ categorias(s) ∧ esVacia??(ls) ∧ ls ⊆ linksCategoriasOHijos(s,c)
Post ≡ res=$_{obs}$ linkConUltimoAcceso(s,c,ls)

Complejidad : O(|(*max).link|)
Descripción : Devuelve el link que se accedio por ultima vez del conjunto ls

sumarAccesosRecientes (**in** s: lli, **in** l: link,**in** fs: conj(fecha)) $\longrightarrow$ res: nat

Pre $\equiv$ l $\in$ links(s) $\wedge$ fs $\subseteq$ diasRecientes(s,l)
Post $\equiv$ res=$_{obs}$ sumarAccesosRecientes(s,l,fs)
Complejidad : O(|l|)
Descripción : Devuelve la suma de todos los accesos recientes del link l

buscarMax(**in** ls:lista(puntero(datosLink)))$\longrightarrow$ res:itLista(puntero(datosLink))

pre $\equiv$ ls =$_{\text{obs}}$ ls$_0$
post$\equiv$ res =$_{\text{obs}}$ buscarMax(ls$_0$)
complejidad : $O(n)$
Descripción : *Devuelveellinkconmasaccesosrecientes*

estaOrdenada(**in** ls:lista(puntero(datosLink))) $\longrightarrow$ res:bool

pre $\equiv$ ls =$_{\text{obs}}$ ls$_0$
post$\equiv$ ls=$_{\text{obs}}$ estaOrdenada(ls$_0$)
complejidad : $O(n)$
Descripción : *Devuelveunvalorbooleanodefiniendosilalistaestaonoordenada*

1.1. Pautas de Implementación

1.1.1. Estructura de Representación

linkLinkIT **se representa con** estrILL **donde** estrILL **es**:
tupla (
arbolCategorias: acat,
actual:nat,
accesosXLink: diccTrie(*link*:string,puntero(datosLink)),
listaLinks:Lista(datosLink), *arrayCatLinks*:arreglo-dimen(linksFamilia))

Donde datosLink **es**:
tupla <*link*:link, *catDLink*puntero(datosCat),*accesosRecientes*:Lista(acceso),*cantAccesosRecientes*:nat>

Donde acceso **es**:
tupla <*dia*:nat, *cantAccesos*:nat>

Donde linksFamilia **es**:
lista (puntero(datosLink))

1.1.2. Invariante de Representación

1. Para todo '*link*' que exista en '*accesosXLink*' la '*catDLink*' de la tupla apuntada en el significado debera existir en *'arbolCategorias'*.
2. Para todo '*link*' que exista en '*accesosXLink*', 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 '*accesosXLink*' su significado deberá existir en '*listaLinks*' y viceversa.

4. Para todo '*link*' que exista en '*accesosXLink*' su significado deberá aparecer en '*arrayCantLinks*' 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 '*accesosXLink*' 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 '*accesosXLink*', '*cantAccesosRecientes*' es igual a la suma de '*cantAccesos*' de cada elemento de la lista '*accesosRecientes*'

Rep : estrLLI $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$

1. ($\forall$linkx) (def?(x,e.accesosXLink)) $\rightarrow_L$
(*obtener(x,e.accesosXLink)).catDLink $\in$ todasLasCategorias(e.arbolCategorias.categorias)
2. ($\forall$linkx) (def?(x,e.accesosXLink)) $\rightarrow_L$
long((*obtener(l,e.accesosXLink)).accesosRecientes) $\leq 3 \wedge$
accesoOrdenadoNoRepetido((*obtener(l,e.accesosXLink)).accesosRecientes) $\wedge_L$
fechasCorrectas(e.actual,((*obtener(l,e.accesosXLink)).accesosRecientes))
3. ($\forall$linkx) (def?(x,e.accesosXLink)) $\leftrightarrow$ (*obtener(x,e.accesosXLink)) $\in$ todosLosLinks(listaLinks)
4. ($\forall$linkx) (def?(x,e.accesosXLink)) $\rightarrow_L$
(($\forall$categoriac) c $\in$ todasLasCategorias(e.arbolCategorias.categorias) $\rightarrow_L$
(esta?((obtener(l,e.accesosXLink)),arrayCatLinks[id(c,e.arbolCategorias)]) $\leftrightarrow$ esPredecesor(c,(*obtener(l,e.accesosXLink))
5. ($\forall$linkx,x' l $\neq$ l') $\wedge$ (def?(x,e.accesosXLink)) $\wedge$ (def?(x,e.accesosXLink)) $\rightarrow_L$ (*obtener(x,e.accesosXLink)) $\neq$ (*obtener(x',e.accesosXLink))
6. ($\forall$nati,i') i <long(e.listaLinks) $\wedge$ i' <long(e.listaLinks) $\rightarrow_L$ e.listaLinks$_i$.link = e.listaLinks$'_i$.link $leftrightarrow$ i = i'
7. ($\forall$nati) i <tam(arrayCatLinks) $\rightarrow_L$ sinRepetidos(linksFamilia$_i$)
8. ($\forall$linkx) (def?(x,e.accesosXLink)) $\rightarrow_L$ (*obtener(x,e.accesosXLink)).cantAccesosRecientes == cantidadDeAccesos((*obtener(x,e.accesosXLink)).accesosRecientes)

1.1.3. Función de Abstraccion

Abs: estrLLI e $\rightarrow$ linkLinkIT
Abs(e) $=_{\text{obs}}$ s: linkLinkIT |

categorias(s) = e.arbolCategorias $\wedge$
links(s)= todosLosLinks(s.listaLinks) $\wedge_L$
($\forall$l:link)def?(l, e.accesosXLink)$\Rightarrow_L$ $categoriaLink(s,l) = *((obtener(l, e.accesosXLink))).catDLink \wedge$
fechaActual(s) = e.actual $\wedge$
($\forall$l:link) l $\in$ links(e) $\Rightarrow_L$ $fechaUltimoAcceso(s,l) = ultimo((*((obtener(s, e.accesosXLink))).accesos).dia) \wedge$
($\forall$l:link)($\forall$f:nat) l$\in$ links(e) $\wedge_L$ esReciente?(e,l,f) $\Rightarrow_L$
accesoRecienteDia(s,l,f) = cantidadPorDia(f,*((obtener(s,e.accesosXLink))).accesos)

Auxiliares

cantidadPorDia : estrLLI $\times$ fecha $\times$ lista(acceso) $\longrightarrow$ nat {esReciente?(e,prim(ls).link,f)}
cantidadPorDia(e,f,ls) $\equiv$ **if** f == (prim(ls)).dia **then** prim(ls).cantAccesos **else** cantidadPorDia(f,fin(ls)) **fi**
todosLosLinks : secu(datosLink) $\longrightarrow$ conj(link)
todosLosLinks(ls) $\equiv$ **if** $\emptyset$?(ls) **then** $\emptyset$ **else** Ag((prim(ls)).link,todosLosLinks(fin(ls))) **fi**

sinRepetidos : secu(α) $\longrightarrow$ bool

sinRepetidos(ls) $\equiv$ **if** vacia?(ls) **then**
true
else
if hayOtro(prim(ls),fin(ls)) **then** false **else** sinRepetidos(fin(ls)) **fi**
fi

hayOtro : $\alpha \times$ secu(α) $\longrightarrow$ bool

hayOtro(x,ls) $\equiv$ **if** vacia?(ls) **then** false **else if** x == prim(ls) **then** true **else** hayOtro(x,fin(ls)) **fi fi**

fechasCorrectas : nat $\times$ secu(acceso) $\longrightarrow$ bool

fechasCorrectas(x,ls) $\equiv$ **if** vacia?(ls) **then**
true
else
if prim(ls).dia >f **then** false **else** fechasCorrectas(x,fin(ls)) **fi**
fi

accesoOrdenadoNoRepetido : secu(acceso) $\longrightarrow$ bool

accesoOrdenadoNoRepetido(ls) $\equiv$ **if** long(ls) ≤ 1 **then**
true
else
if prim(ls).dia $\geq$ prim(fin(ls)).dia **then**
false
else
accesoOrdenadoNoRepetido(fin(ls))
fi
fi

cantidadDeAccesos : secu(acceso) $\longrightarrow$ nat

cantidad(ls) $\equiv$ **if** vacia?(ls) **then** 0 **else** (prim(ls)).cantAccesos + fin(ls) **fi**

1.1.4. Algoritmos

Algoritmo: 1

ICATEGORIAS (**in** s: lli) $\longrightarrow$ res: ac

res $\leftarrow$ s.arbolCategorias //O(1)

Complejidad: O(1)

Algoritmo: 2

ILINKS (**in** s: estrLLI) $\longrightarrow$ res: conj(link)

itLista iterador $\leftarrow$ crearIt(s.listaLinks) //O(1)

while(haySiguiente(iterador)) //O(|s.listaLinks|)

agregar(res,(*siguiente(iterador).link)) //O(|l|)

avanzar(iterador) //O(1)

end while

Complejidad: O($\sum_{i=1}^{longitud(s.listaLinks)}$)

Algoritmo: 3

ICATEGORIALINK (**in** s: estrLLI, **in** l: link) $\longrightarrow$ res: categoria

res ← *((obtener(l,s.accesosXLink))).catDLink //O(|l|)

Complejidad: O(|l|)

Algoritmo: 4

IFECHAACTUAL (**in** s: estrLLI) ⟶ res: fecha

res ← s.actual //O(1)

Complejidad: O(1)

Algoritmo: 5

IFECHAULTIMOACCESO (**in** s: estrLLI, **in** l: link) ⟶ res: fecha

res ← ultimo(*((obtener(l,s.accesosXLink))).accesosRecientes).dia //O(|l|)

Complejidad: O(|l|)

Algoritmo: 6

IACCESOSRECIENTESDIA (**in** s: estrLLI, **in** l: link, **in** f: fecha) ⟶ res: nat

lista(acceso) accesos ← vacia() //O(1)

res ← 0 //O(1)

accesos ← *((obtener(l,s.accesosXLink))).accesosRecientes //O(|l|)

while(esVacia?(accesos) ∧ res = 0) //O(|accesos|)

if (ultimo(accesos)).dia == f //O(1)

then res ← (ultimo(accesos)).cantAccesos //O(1)

else accesos ← fin(accesos) FI //O(1)

end while

Complejidad:O(|l|)

Algoritmo: 7

IINICIAR (**in** ac: acat) ⟶ res: estrLLI

res.actual ← 1 //O(1)

res.arbolCategorias ← &ac //O(1)

var c: nat //O(1)

c ← 1 //O(1)

res.arrayCantLinks ← crearArreglo(#categorias(ac)) //O(1)

res.listaLinks ← vacia() //O(1)

res.accesosXLink ← vacio() //O(1)

while (c ≤ #categorias(ac)) //O(#categorias(ac))

linksFamilia llist ← vacia() //O(1)

res.arrayCatLinks[c] ← llist //O(1)

c ++ //O(1)

end while

Complejidad: (#categorias(ac))

Algoritmo: 8

INUEVOLINK (**in/out** s: lli, **in** l: link , **in** c: categoria)

puntero(datosCat) cat ← obtener(c,s.arbolCategorias) //O(|c|)

lista(acceso) accesoDeNuevoLink ← vacia() //O(1)

datosLink nuevoLink ← <l,cat,accesoDeNuevoLink,0> //O(|l|)

puntero(datosLink) puntLink ← nuevoLink //O(1)

definir(l,puntLink,s.accesosXLink) //O(|l|)

agregarAtras(s.listaLinks,puntLink) //O(1)

while(cat ≠ puntRaiz(s.arbolCategorias)) //O(h)

agregarAtras(s.arrayCatLinks[(*cat).id],puntLink) //O(1)

cat ← cat.abuelo //O(1)

end while

agregarAtras(s.arrayCatLinks[(*cat).id],puntLink) //O(1)

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

Algoritmo: 9

IACCESO (**in/out** s: lli, **in** l: link , **in** f: fecha)

if s.actual == f //O(1)

then s.actual ← s.actual //O(1)

else s.actual ← f **fi** //O(1)

var puntero(datosLink) puntLink ← obtener(l,s.accesosXLink) //O(|l|)

if (ultimo((*puntLink).accesos)).dia == f //O(1)

then (ultimo((*puntLink).accesos)).cantAccesos++ //O(1)

else agregarAtras((*puntLink).accesos), f) **fi** //O(1)

if longitud((*puntLink).accesos) == 4 //O(1)

then fin((*puntLink).accesos) //O(1)

fi

(*puntLink).cantAccesosRecientes++ //O(1)

Complejidad: O(|l|)

Algoritmo: 10

IESRECIENTE? (**in** s: lli, **in** l: link , **in** f: fecha) ⟶ res: bool

res ← menorReciente(s,l) ≤ f ∧ f ≤ fechaUltimoAcceso(s,l) //O(|l|)

Complejidad: O(|l|)

Algoritmo: 11

IACCESOSRECIENTES (**in** s: lli, **in** c: categoria **in** l: link) ⟶ res: nat

res ← sumarAccesosRecientes(s, l, diasRecientesParaCategoria(s, c) ∩ diasRecientes(s, l)) //O(|l|)

Complejidad: O(|l|)

Algoritmo: 12

ILINKSORDENADOSPORACCESOS (**in** s: lli, **in** c: categoria) ⟶ res: itListaUni(lista(link))

nat id ← id(s.arbolCategorias,c) //O(|c|)

lista(puntero(datosLink)) listaOrdenada ← vacia() //O(1)

itLista(puntero(datosLink)) itMax ← crearIt(s.arrayCantLinks[id]) //O(1)

if iestaOrdenada?(s.arrayCantLinks[id])

//O(1)

then

while(haySiguiente?(s.arrayCantLinks[id])) //O(n)

itMax ← iBuscarMax(s.arrayCantLinks[id]) //O(n)

agregarAtras(listaOrdenada,siguiente(itMax)) //O(1)

eliminarSiguiente(itMax) //O(1)

end while

res ← crearIt(listaOrdenada) //O(1)

s.arrayCatLinks[id] ← listaOrdenada //O(1)

else

res ← crearIt(s.arrayCantLinks[id]) //O(1)

fi

Complejidad: O(n^2)

Algoritmo: 13

IBUSCARMAX (**in** ls: lista(puntero(datosLink))) ⟶ res: itLista(puntero(datosLink))

res ← crearIt(ls) //O(1)

itLista(puntero(datosLink)) itRecorre ← crearIt(ls) //O(1)

nat max ← (*siguiente(itRecorre)).cantAccesosRecientes //O(1)

while(haySiguiente(itRecorre)) //O(n)

if max <(*siguiente(itRecorre)).cantAccesosRecientes //O(1)

then //O(1)

max ← (*siguiente(itRecorre)).cantAccesosRecientes //O(1)

res ← itRecorre //O(1)

end while

avanzar(itRecorre) //O(1)

end while

Complejidad: O(n)

Algoritmo: 14

IESTAORDENADA (**in** ls: lista(puntero(datosLink))) ⟶ res: bool

res ← true //O(1)

itLista(puntero(datosLink)) itRecorre ← crearIt(ls) //O(1)

nat aux ← (*siguiente(itRecorre)).cantAccesosRecientes //O(1)

while(haySiguiente(itRecorre) ∧ res == true) //O(n)

avanzar(itRecorre) //O(1)

if aux <(*siguiente(itRecorre)).cantAccesosRecientes //O(1)

then //O(1)

res ← false //O(1)

fi //O(1)

aux ← (*siguiente(itRecorre)).cantAccesosRecientes //O(1)

end while

Complejidad: O(n)

Algoritmo: 15

ICANTLINKS (**in** s: lli, **in** c: categoria) ⟶ res: nat

puntero(datosCat) cat ← obtener(c,s.arbolCategorias) //O(|c|)

res ← longitud(arrayCantLinks[(*cat).id]) //O(1)

Complejidad: O(|c|)

Algoritmo: 16

IMENORRECIENTE (**in** s: lli, **in** l: link) ⟶ res: fecha

res ← max(fechaUltimoAcceso(s,l)+1,diasRecientes) - diasRecientes //O(|l|)

Complejidad: O(1)

Algoritmo: 17

IDIASRECIENTES (**in** s: lli, **in** l: link) ⟶ res: conj(fecha)

res ← diasRecientesDesde(s,l,menorReciente(s,l)) //O(|l|)

Complejidad:O(|l|)

Algoritmo: 18

IDIASRECIENTESDESDE (**in** s: lli, **in** l: link, **in** f: fecha) $\longrightarrow$ res: conj(fecha)

while(esReciente?(s,l,f)) //O(|l|)

Agregar(f,res) //O(1)

fecha++ //O(1)

end while

Complejidad: O(|l|)

Algoritmo: 19

IDIASRECIENTESPARACATEGORIAS (**in** s: lli, **in** c: categoria) $\longrightarrow$ res: conj(fecha)

itLista(puntero(datosLink)) links $\leftarrow$ crearIt(arrayCatLinks[id(s.arbolCategorias,c)] //O(1)

diasRecientes(s,linkConUltimoAcceso(s,c,links)) //O($\star$|l|)

Complejidad: O($\star$|l|)

Algoritmo: 20

ISUMARACCESOSRECIENTES (**in** s: lli, **in** l: link,**in** fs: conj(fecha)) $\longrightarrow$ res: nat

itConj iterador $\longleftarrow$ crearIt(fs) //O(1)

while(haySiguiente(iterador)) //O(1)

res $\longleftarrow$ accesosRecientesDia(s,l,siguiente(iterador)) //O(|l|)

avanzar(iterador) //O(1)

end while

Complejidad: O(|l|)

Algoritmo: 21

ILINKCONULTIMOACCESO (**in** s: lli, **in** c: categoria,**in** ls: itLista(puntero(datosLink)) $\longrightarrow$ res: link

while(haySiguiente(ls)) //O(|ls|)

if s.actual == (ultimo((*siguiente(ls)).accesosRecientes)).dia //O(1)

then res $\leftarrow$ (siguiente(ls))

//O(1)

fi //O(1)

avanzar(ls) //O(1)

end while

Complejidad: O(|(*max).link|)

1.2. Descripcion de Complejidades de Algoritmos

1. **ICATEGORIAS**:
   Devuelve el arbol de categorias del sistema, esto cuesta O(1).
   Orden Total:O(1)=**O(1)**

2. **ILINKS**:
   Se crea un conjunto vacio, esto tarda O(1). Se crea un itLista, esto tarda O(1).
   Se ingresa a un ciclo preguntando si haySiguiente, esto cuesta O(1), se le agrega link apuntado de cada tupla de datosLink de la lista listaLinks, esto tarda O(|l|), luego se avanza el it, esto cuesta O(1).
   Luego de recorrer toda la lista se sale del ciclo habiendo demorado finalmente O(|lista|), se devuelve el conjunto.
   Orden Total:O(1)+O(1)+O(1)+(suma O(|l|))+O(1)=**O(suma O(|l|))**

3. **ICATEGORIALINK**:
   Se utiliza la operacion obtener del diccionario accesosXLink, la cual devuelve un puntero a datosLink,
   se devuelve lo apuntado a catDLink, esto cuesta O(|l|).
   Orden Total:O(|l|)=**O(|l|)**

4. **IFECHAACTUAL**:
   Devuelve la fecha actual del sistema, esto cuesta O(1).
   Orden Total:O(1)=**O(1)**

5. **IFECHAULTIMOACCESO**:
   Se utiliza la operacion obtener del diccionario accesosXLink, la cual devuelve un puntero a datosLink,
   se accede a la lista accesosRecientes dentro de la tupla, se devuelve dia del ultimo elemento, esto cuesta O(|l|).
   Orden Total:O(|l|)=**O(|l|)**

6. **IACCESOSRECIENTESDIA**:
   Se crea una lista de acceso vacia, esto cuesta O(1). Se le guarda a la lista, la lista de accesosRecientes, la cual se obtiene con la operacion obtener del diccionario accesosXLink consultando por el link dado, esto cuesta O(|l|).
   Se ingresa a un ciclo, preguntando si no es vacia la lista, esto cuesta O(1).
   Se pregunta si dia del primer elemento de la lista es igual a f, esto cuesta O(1), en caso verdadero se devuelve cantAccesos de esa tupla, esto cuesta O(1), en caso falso se modifica la lista sacando el primer elemento, esto cuesta O(1). Una vez recorrida toda la lista se sale del ciclo demorando O(|lista|)
   Orden Total:O(1)+O(|l|)+O()=**O(|l|)**

7. **IINICIAR**:
   Se guarda en res.actual la fecha igual a 1, esto cuesta O(1). Se pasa por referencia el arbol dado y se lo guarda en res.arbolCategorias, estoy cuesta O(1). Se crea una variable del tipo nat, cuesta O(1) , se inicializa esta variable con 1, esto cuesta O(1), se crea un arreglo con tamaño igual a #categorias(ac) y se lo guarda en res.arrayCatLinks, esto cuesta O(1),
   se inicializa res.listaLinks como vacia, esto cuesta O(1), se inicializa con vacio el diccionario res.accesosXLink.
   Se ingresa a un ciclo consultando si c es menor o igual a la cantidad de categorias de ac, esto cuesta O(1). Se crea una lista linksFamilia inicializada con vacio, esto cuesta O(1).
   Se guarda en res.arrayCatLinks[c] la lista linksFamilia, esto cuesta O(1), se le suma 1 a c, esto cuesta O(1).Una vez que no se cumple la condicion del ciclo se sale del mismo habiendo demorado finalmente O((#categorias(ac)).
   Orden Total:O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+
   O(#categorias(ac)*(O(1)+O(1)+O(1)))=**O(#categorias(ac))**

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))

9. **IACCESO**:
   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. **IESRECIENTE**:
    Devuelve un bool dependiendo si el f pasado es mayor o igual a la fecha obtenida por la operacion menorReciente(s,l) la cual tarda O(|l|) y si es menor o igual a la fechaUltimoAcceso(s,l) la cual tambien tarda O(l). Tanto menorReciente como fechaUltimoAcceso son operaciones del modulo LinkLinkIT y se les pasa el sistema y un link.
    Orden Total:O(|l|)+O(|l|)=**O(|l|)**

11. **IACCESOSRECIENTES**:
    Devuelve un nat, el cual proviene de la operacion sumarAccesosRecientes que se le pasa el sistema, el link y la interseccion que demora O(1) de la operacion diasRecientesParaCategoria(s,c), que demora O(|⋆l|) con la operacion diasRecientes(s,l) que demora O(|l|).
    Aclaracion: ⋆l es el link obtenido de la operacion linkConMasAccesos el cual pertenece al conjunto de links de la categoria c.
    Orden Total:O(|l|)+(O(1)*(O(|l|)+O(|l|)))=**O(|l|)**

12. **ILINKSORDENADOSPORACCESOS**:
    Se crea un nat id al cual se le pasa el id de la categoria que ingresan por medio de la operacion del modulo de arbolCategorias, lo cual demora O(|c|). Se crea una lista de puntero a datosLink llamada listaOrdenada la cual se la inicializa en vacio, esto cuesta O(1).
    Se crea un itLista de puntero a datosLink nombrado itMax al cual se le pasa por referencia la lista del arreglo arrayCatLinks en la posicion del id de la categoria, esto demora O(1). Se pregunta por si la lista del arreglo arrayCatLinks en la posicion del id de la categoria no esta ordenada, esto cuesta O(n). En caso verdadero se ingresa a un ciclo, con la condicion de que haySiguiente? arrayCatLinks[id] sea verdadero, esto demora O(1). Se le pasa a itMax un iterador de la operacion BuscarMax a la cual se le pasa arrayCatLinks[id], esto demora O(n). Luego se utiliza la operacion agregarAtras demorando O(1), la cual agrega la posicion actual del iterador itMax en la lista listaOrdenada.
    Se usa la operacion eliminarSiguiente con la cual se elimina la posicion actual del iterador itMax, demorando O(1). Una vez recorrido todo el itLista de puntero a datosLink y que haySiguiente? sea false se sale del ciclo tardando O(n^2). Se le pasa a res un iterador unidireccional con la listaOrdenada en O(1).
    Se modifica arrayCatLinks[id] pasandole la listaOrdenada, esto demora O(1). Luego en la parte falsa del IF en el caso de que si este ordenada la arrayCatLinks[id] se le pasa a res un iterador unidireccional con arrayCatLinks[id]

demorando O(1).
Aclaracion: n es igual a la cantidad de elementos de la lista. **Orden Total:**O(|c|)+O(1)+O(1)+O(n)+[n*(O(n)+O(1)+O(1)

13. **IBUSCARMAX**:
    Se inicializa res pasandole un itLista de puntero a datosLink demorando O(1). Se crea un itLista de puntero a datosLink llamad itRecorre pasandole la lista que nos ingresa, demorando O(1).
    Se crea un nat llamado max el cual es inicializado pasandole el valor de cantAccesosRecientes de la tupla apuntada por la posicion actual del itRecorre, esto tarda O(1). Se ingresa a un ciclo con la condicion de que haySiguiente de itRecorre sea true, esto demora O(1). Se pregunta si max es menor al valor cantAccesosrecientes de la tupla apuntada por la posicion actual del itRecorre, esto demora O(1). En caso verdadero se guarda en max el valor cantAccesosrecientes de la tupla apuntada por la posicion actual del itRecorre, esto demora O(1).
    Se guarda en res el iterador itRecorre demorando O(1). No hay parte falsa. Se ultiza la operacion avanzar a la cual se le pasa itRecorre demorando O(1). Una vez recorrido toda la lista, y haySiguiente?(itRecorre) sea falso, se sale del ciclo habiendo demorado O(n). Aclaracion: n es igual a la cantidad de elementos de la lista. **Orden Total:**O(1)+O(1)+O(1)+[n*(O(1)+O(1)+O(1)+O(1))]=**O(n)**

14. **IESTAORDENADA**:
    Se inicializa res con true, esto demanda O(1). Se crea un itLista de puntero a datosLink itRecorre al cual se lo inicializa con una lista ls que pasan como parametro, esto cuesta O(1).
    Se crea un nat aux el cual es inicializado con el valor de cantAccesosRecientes apuntado en la posicion actual del iterador. Esto cuesta O(1). Se ingresa con la condicion de que haySiguiente del iterador sea verdadera y que res sea igual a true, esto cuesta O(1).
    Se avanza el iterador, lo que cuesta O(1). Se pregunta si el valor de aux es menor a el valor de cantAccesosRecientes apuntado en la posicion actual del iterador, esto demanda O(1), en caso afirmativo se modifica res por false, tardando O(1).
    Se modifica aux pasandole el valor de cantAccesosRecientes apuntado en la posicion actual del iterador. Luego de las iteraciones correspondientes, se sale del ciclo habiendo demorado en el peor de los casos O(n). Aclaracion: n es igual a la cantidad de elementos de la lista. **Orden Total:**O(1)+O(1)+O(1)+[n*(O(1)+O(1)+O(1)+O(1))]=**O(n)**

15. **ICANTLINKS**:
    Se crea un puntero a datosCat cat al cual se le guarda el puntero obtenido por la operacion obtener del modulo arbolCategorias, lo cual tarda O(|c|).
    Se devuelve la longitud de la lista del arreglo arrayCantLinks[(*cat).id], lo que demora O(1)
    Orden Total:O(|c|)+O(1)=**O(|c|)**

16. **IMENORRECIENTE**:
    Se devuelve la resta la cual demora O(1), del maximo que tarda O(1), de la operacion fechaUltimoAcceso(s,l) que demora O(|l|) + 1, con el valor constante diasRecientes.
    Orden Total:O(|l|)+O(1)+O(1)=**O(|l|)**

17. **IDIASRECIENTES**:
    Devuelve un conjunto de fechas dado por la operacion diasRecientesDesde que demora O(|l|), a la cual se le pasa el sistema, un link y la operacion menorReciente que tambien demora O(|l|).
    Orden Total:O(|l|)+O(|l|)=**O(|l|)**

18. **IDIASRECIENTESDESDE**:
    Se ingresa a un ciclo consultando por la operacion esReciente chequeando si la fecha es reciente, esta operacion tarda O(|l|), dentro del ciclo, se utiliza la operacion Agregar que agrega por copia la fecha al conjunto, esto demora O(1).
    Se modifica f y se le suma uno, esto demora tambien O(1).Se sale del ciclo.
    Orden Total:O(|l|)+O(1)+O(1)=**O(|l|)**

19. **IDIASRECIENTESPARACATEGORIAS**:
    Se crea un iterador itLista de puntero a datosLink links el cual se inicializa con arrayCatLinks[id(s.arbolCategorias,c)], o sea la lista listaLinks de la posicion id(s.arbolCategorias,c) del arreglo arrayCatLinks, esto demora O(1).
    Se devuelve un conjunto de fechas dado por la operacion diasRecientes que demora O($\star$|l|) a la cual se le pasan el sistema, y la operacion linkConMasAccesos que demora O($\star$|l|) a la cual se le pasan, el sistema, la categoria c y el itLista links.
    Aclaracion: $\star$l es el link obtenido de la operacion linkConMasAccesos el cual pertenece al conjunto de links de la categoria c.
    Orden Total:O(1)+O($\star$|l|)+O($\star$|l|)=**O($\star$|l|)**

20. **ISUMARACCESOSRECIENTES**:
    Se crea un itConj iterador al cual se le pasa un conjunto de fechas, lo que demora O(1). Se ingresa a un ciclo

consultando por si haySiguiente del itConj, esto demora O(1).
Se modifica res sumandole, la cual demora O(1), al valor anterior que tenia, el valor de la operacion accesosRecientesDia que demora O(|l|), pasandole el sistema, el link, y el valor de la posicion actual del iterador.
Luego se avanza el iterador que demora O(1). Una vez que la condicion del ciclo es falsa, se sale del ciclo habiendo demorado O(|fs|).
Orden Total:O(1)+O(1)+O(1)=**O(|l|)**

21. **ILINKCONULTIMOACCESO**:
Se ingresa a un ciclo consultando si hay siguiente del itLista, lo que demora O(1).
Se pregunta si el dia de la tupla del ultimo elemento de la lista de acceso de accesosRecientes de la tupla apuntada en la posicion del it es igual a la del sistema, esto demora O(1).
En caso afirmativo, se modifica res guardando la posicion actual del iterador, esto demora O(1).Se avanza el iterador, lo que cuesta O(1). Una vez recorrida toda la lista se sale del ciclo habiendo demorado O(|ls|). Se devuelve el link de la tupla apuntada por max. Esto demora O(|l|).
Orden Total:|ls|*(O(1)+O(1)+O(1))=**O(|ls|)**

2. TAD ArbolDeCategorias

TAD ArbolDeCategorias

gros acat

exporta generadores, categorias, raíz, padre, id, altura, está?, esSubCategoria, alturaCategoria, hijos

usa BOOL, NAT, CONJUNTO

observadores bcos

categorias : acat ac $\longrightarrow$ conj(categoria)

raiz : acat ac $\longrightarrow$ categoria

padre : acat ac $\times$ categoria h $\longrightarrow$ categoria $\{esta?(h, ac) \wedge raiz(ac) \neq h\}$

id : acat ac $\times$ categoria c $\longrightarrow$ nat $\{esta?(c, ac)\}$

generadores

nuevo : categoria c $\longrightarrow$ acat $\{\neg vacia?(c)\}$

agregar : acat ac $\times$ categoria c $\times$ categoria h $\longrightarrow$ acat $\{esta?(c, ac) \wedge vacia?(h) \wedge esta?(h, ac)\}$

otras operaciones

altura : acat ac $\longrightarrow$ nat

esta? : categoria c $\times$ acat ac $\longrightarrow$ bool

esSubCategoria : acat ac $\times$ categoria c $\times$ categoria h $\longrightarrow$ bool $\{esta?(c, ac) \wedge esta?(h, ac)\}$

alturaCategoria : acat ac $\times$ categoria c $\longrightarrow$ nat $\{esta?(c, ac)\}$

hijos : acat ac $\times$ categoria c $\longrightarrow$ conj(categoria) $\{esta?(c, ac)\}$

axiomas $\forall$arbolDeCategoriasa
$\forall$categoriac
$\forall$conj(arbolDeCategoria)ca
$\forall$conj(categoria)cc

categorias(nuevo(c)) $\equiv$ c

categorias(agregar(ac,c,h)) $\equiv$ Ag(h, categorias(ac))

raiz(nuevo(c)) $\equiv$ c

raiz(agregar(ac,c,h)) $\equiv$ raiz(ac)

padre(agregar(ac,c,h),h') ≡ **if** h == h' **then** c **else** padre(ac,c,h') **fi**

id(nuevo(c), c') ≡ 1
id(agregar(ac,c,h), h') ≡ **if** h==h' **then** #categorias(ac) + 1 **else** id(ac,h2) **fi**

altura(nuevo(c)) ≡ alturaCategoria(nuevo(c), c)
altura(agregar(ac,c,h)) ≡ max(altura(ac), alturaCategoria(agregar(ac,c,h), h))
alturaCategoria(ac, c) ≡ **if** c == raiz(ac) **then** 1 **else** 1 + alturaCategoria(ac, padre(ac, c)) **fi**

esta?(c,ac) ≡ c ∃ categorias(ac)

esSubCategoria(ac,c,h) ≡ c == h ∨L (h = raiz(ac) ∧L esSubCategoria(ac, c, padre(ac, h)))

hijos(nuevo(c1), c2) ≡ ∅
hijos(agregar(ac,c,h), c') ≡ **if** h == c' **then** ∅ **else** (**if** c==c' **then** h **else** ∅ **fi**) ∪ hijos(ac,c,c') **fi**

Fin TAD

2.0.1. Modulo de Arbol de Categorias

generos: *acat*
usa: bool, nat, conjunto
se explica con: TAD ArbolDeCategorias
géneros: acat

2.0.2. Operaciones Básicas

categorias (**in** ac: acat) ⟶ res: conj(categoria)
Pre ≡ true
Post ≡ res=$_{obs}$ categorias(ac)
Complejidad : O(#categorias(ac))
Descripción : Devuelve el conjunto de categorias de un ac

raiz (**in** ac: acat) ⟶ res: categoria

Pre ≡ true
Post ≡ res=$_{obs}$ raiz(ac)
Complejidad : O(|c|)
Descripción : Devuelve la raiz del arbol ac
Aliasing: res es modificable si y solo si ac es modificable.

padre (**in** ac: estrAC, **in** h: categoria) ⟶ res: categoria

Pre $\equiv$ h $\in$ ac $\wedge$ raiz(ac) $\neq$ h
Post $\equiv$ res=$_{obs}$ padre(ac,h)
Complejidad : O(|h|+|res|)
Descripción : Devuelve el padre de una categoria
Aliasing: res es modificable si y solo si ac es modificable.

id (**in** ac: estrAC, **in** c: categoria) $\longrightarrow$ res:nat

Pre $\equiv$ h $\in$ ac
Post $\equiv$ res=$_{obs}$ id(ac,c)
Complejidad : O(|c|)
Descripción : Devuelve el id de una categoria c en el arbol ac
Aliasing: res es modificable si y solo si ac es modificable.

nuevo (**in** c: categoria) $\longrightarrow$ res:estrAC

Pre $\equiv$ vacia?(c)
Post $\equiv$ res=$_{obs}$ nuevo(c)
Complejidad : O(|c|)
Descripción : Crea un arbol

agregar (**in/out** ac: estrAC,**in** c: categoria, **in** h: categoria)

Pre $\equiv$ c $\in$ ac $\wedge$ vacia?(h) $\wedge$ ac$_0$ =$_{obs}$ ac
Post $\equiv$ ac=$_{obs}$ agregar(ac$_0$,c,h)
Complejidad : O(|c|+|h|)
Descripción : Agrega una categoria hija a una padre

altura (**in** ac: estrAC) $\longrightarrow$ res:nat

Pre $\equiv$ true
Post $\equiv$ res=$_{obs}$ altura(ac)
Complejidad : O(1)
Descripción : Devuelve la altura del arbol ac
Aliasing: res es modificable si y solo si ac es modificable.

esta? (**in** c: categoria,**in** ac: estrAC) $\longrightarrow$ res:bool

Pre $\equiv$ true
Post $\equiv$ res=$_{obs}$ esta?(c,ac)
Complejidad : O(|ac|)
Descripción : Devuelve si esta o no en el arbol la categoria c

esSubCategoria (**in** ac: estrAC, **in** c: categoria,**in** h: categoria) $\longrightarrow$ res:bool

Pre $\equiv$ esta?(c,ac) $\wedge$ esta?(h,ac)
Post $\equiv$ res=$_{obs}$ esSubCategoria(ac,c,h)
Complejidad : O(|c|+|h|+*(h))
Descripción : Devuelve si c es descendiente de h

alturaCategoria (**in** ac: estrAC, **in** c: categoria) $\longrightarrow$ res:nat

Pre $\equiv$ esta?(c,ac)
Post $\equiv$ res=$_{obs}$ alturaCategoria(ac,c)
Complejidad : O(|c|)
descripción : Devuelve la altura de la categoria c
Aliasing: res es modificable si y solo si ac es modificable.

hijos (**in** ac: estrAC, **in** c: categoria) $\longrightarrow$ res:conj(categoria)

Pre $\equiv$ esta?(c,ac)
Post $\equiv$ res$=_{obs}$ hijos(ac,c)
Complejidad : O(|c|)
Descripción : Devuelve el conjunto de categorias hijos de c

2.1. Pautas de Implementación

2.1.1. Estructura de Representación

arbolDeCategorias **se representa con** estrAC **donde** estrAC **es**:
tupla <
raiz: puntero(datosCat),
cantidad: nat,
alturaMax: nat,
familia:diccTrie(*padre*:string,puntero(datosCat)),
categorias: Lista(datosCat)>
Donde datosCat **es**:
tupla <
categoria:string,
id:nat,
altura:nat,
hijos:conj(puntero(datosCat)),
abuelo:puntero(datosCat)>

2.1.2. Invariante de Representación

1. Para cada '*padre* obtener el significado devolvera un puntero(datosCat) donde '*categoria*' es igual a la clave
2. Para toda clave '*padre*' que exista en '*familia*' debera ser o raiz o pertenecer a algun conjunto de punteros de '*hijos*' de alguna clave '*padre*'
3. Todos los elementos de '*hijos* de una clave '*padre*', cada uno de estos hijos tendran como '*abuelo*' a ese '*padre*' cuando sean clave.
4. '*cantidad*' sera igual a la longitud de la lista '*categorias*'.
5. Cuando la clave es igual a '*raiz*' la '*altura* es 1.
6. La '*altura*' del puntero a datosCat de cada clave es menor o igual a '*alturaMax*'.
7. Existe una clave en la cual, la '*altura*' del significado de esta es igual a '*alturaMax*'.
8. Los '*hijos*' de una clave tienen '*altura*' igual a 1 + '*altura* de la clave.
9. Todos los '*id*' de significado de cada clave deberan ser menor o igual a '*cant*'.
10. No hay '*id*' repetidos en el '*familia*.
11. Todos los '*id*' son consecutivos.

Rep : estrAC $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$

1. ($\forall$stringx) (def?(x,e.familia)) $\leftrightarrow$ (*obtener(x,e.familia)).categoria = x

2. (∀stringx,y) (def?(x,e.familia)) ↔ (x == e.raiz) ∨ (def?(y,e.familia)) $\wedge_L$ x ∈ hijosDe(*((obtener(y,e.familia))).hijos)
3. (∀stringx,y) (def?(x,e.familia)) ∧ (def?(y,e.familia)) $\Rightarrow_L$ y ∈ *((obtener(x,e.familia))).hijos ⇔ (*(*(obtener(y,e.familia))).abuelo).categoria = x
4. e.cantidad = longitud(e.categorias)
5. (∀stringx) (def?(x,e.familia)) ∧ x = e.raiz $\Rightarrow_L$ *((obtener(x,e.familia))).altura = 1
6. (∀stringx) (def?(x,e.familia)) $\Rightarrow_L$ (*obtener(x,e.familia)).altura ≤ e.alturaMax
7. (∃x: string) (def?(x,e.familia)) $\wedge_L$ *((obtener(x,e.familia))).altura = e.alturaMax
8. (∀stringx,y) (def?(x,e.familia)) ∧ (def?(y,e.familia)) $\wedge_L$ y ∈ hijosDe((*(obtener(x,e.familia))).hijos) ⇒ (*(obtener(y,e.familia))).altura = 1 + (*(obtener(x,e.familia))).altura
9. (∀stringx) (def?(x,e.familia)) $\Rightarrow_L$ (*(obtener(x,e.familia))).id ≤ e.cant
10. (∀stringx,y) (def?(x,e.familia)) ∧ (def?(y,e.familia)) $\Rightarrow_L$ (*(obtener(x,e.familia))).id ≠ (*(obtener(y,e.familia))).id
11. (∀stringx) (def?(x,e.familia)) (∃ y: string) (def?(y,e.familia)) ⇔ (*(obtener(y,e.familia))).id ≤ e.cantidad ∧ (*(obtener(x,e.familia))).id < e.cantidad $\wedge_L$ (*(obtener(y,e.familia))).id = 1 + (*(obtener(x,e.familia))).id

2.1.3. Función de Abstraccion

Abs: estr e → arbolDeCategorias
Abs(e) $=_{obs}$ ac: arbolDeCategorias |

categorias(ac) = todasLasCategorias(e.categorias) $\wedge_L$
raiz(ac) = (*e.raiz).categoria $\wedge_L$
(∀categoriac) esta?(c,ac) ∧ c ≠ raiz(ac) $\Rightarrow_L$ padre(ac,c) = (*(*(obtener(c,e.familia))).abuelo).categoria $\wedge_L$
(∀categoriac) esta?(c,ac) $\Rightarrow_L$ id(ac,c) = (*(obtener(c,e.familia))).id

Auxiliares

todasLasCategorias : secu(datosCat) ⟶ conj(categoria)
Ag((prim(cs)).categoria,fin(cs)) ≡

2.1.4. Algoritmos

Algoritmo: 1

ICATEGORIAS (**in** ac: estrAC) ⟶ res: conj(categoria)

res ← vacio()	//O(1)
itLista iterador ← crearIt(ac.categorias)	//O(1)
while(haySiguiente(iterador))	//O(longitud(ac.categorias))
agregar(res, siguiente(iterador).categoria)	//O(|c|)
avanzar(iterador)	//O(1)
end while	//O(1)

Complejidad: sumatoria

Algoritmo: 2

IRAIZ (**in** ac: estrAC) $\longrightarrow$ res: categoria

res $\leftarrow$ (*ac.raiz).categoria //O(|c|)

Complejidad: O(|c|)

Algoritmo: 3

IPADRE (**in** ac: estrAC, **in** h: categoria) $\longrightarrow$ res: puntero(categoria)

res $\leftarrow$ (*(*(obtener(h,ac.familia))).abuelo).categoria //O(|h| + |res|)

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

Algoritmo: 4

IID (**in** ac: estrAC, **in** c: categoria) $\longrightarrow$ res:nat

res $\leftarrow$ (*(obtener(c,ac.familia))).id //O(|c|)

Complejidad: O(|c|)

Algoritmo: 5

INUEVO (**in** c: categoria) $\longrightarrow$ res:estrAC

res.cantidad $\leftarrow$ 1 //O(1)

datosCat tuplaA //O(1)

puntero(datosCat) punt $\leftarrow$ &tuplaA //O(1)

tuplaA $\leftarrow$ tupla(c,1,1,vacio(), punt) //O(|c|)

res.raiz $\leftarrow$ punt //O(1)

res.alturaMax $\leftarrow$ 1 //O(1)

res.familia $\leftarrow$ definir(c, punt, res.familia) //O(|c|)

res.categorias $\leftarrow$agregarAtras(tuplaA,res.categorias) //O(1)

Complejidad: O(|c|)

Algoritmo: 6

IAGREGAR (**in/out** ac: estrAC,**in** c: categoria, **in** h: categoria)

puntero(datosCat) puntPadre ← obtener(c,ac.familia) //O(|c|)

if (*puntPadre).altura == ac.alturaMax //O(1)

then ac.alturaMax ← ac.alturaMax + 1 //O(1)

ELSE ac.alturaMax ← ac.alturaMax FI //O(1)

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)

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

Algoritmo: 7

IALTURA (**in** ac: estrAC) ⟶ res:nat

res ← ac.alturaMax //O(1)

Complejidad: O(1)

Algoritmo: 8

IESTA? (**in** c: categoria,**in** ac: estrAC) ⟶ res:bool

res ← def?(c,ac.familia) //O(|c|)

Complejidad: O(|c|)

Algoritmo: 9

IESSUBCATEGORIA (**in** ac: estrAC, **in** c: categoria,**in** h: categoria) ⟶ res:bool

puntero(datosCat) puntPadre ← (obtener(c,ac.familia)) //O(|c|)

res ← false //O(1)

puntero(datosCat) actual //O(1)

if c == ac.raiz //O(|c|)

then res ← true //O(1)

ELSE actual ← (obtener(h,ac.familia)) //O(|h|)

while(res ≠ true ∧ actual ≠ ac.raiz) //O(altura de h)

if PERTENECE?((*puntPadre).hijos,actual) //O(cantidad((*puntPadre).hijos))

then res ← true //O(1)

ELSE actual ← (*actual).abuelo FI FI //O(1)

Complejidad:O(|c|+|h|+ sumatoria hasta la altura de h de cantidad de hijos que tenga c)

Algoritmo: 10

IALTURACATEGORIA (**in** ac: estrAC, **in** c: categoria) ⟶ res:nat

res ← (*(obtener(c,ac.familia))).altura //O(|c|)

Complejidad: O(|c|)

Algoritmo: 11

IHIJOS (**in** ac: estrAC, **in** c: categoria) ⟶ res:itConjUni(puntero(datosCat))

res ← crearIt((*obtener(c,ac.familia)).hijos) //O(|c|)

Complejidad:O(|c|)

Algoritmo 12

IOBTENER (**in** c: categoria, **in** ac: estrAC) ⟶ res:puntero(datosCat)

res ← obtener(c,ac.familia) //O(|c|)

Complejidad: O(|c|)

Algoritmo: 13

IPUNTRAIZ (**in** ac: estrAC) ⟶ res:puntero(datosCat)

res ← ac.raiz //O(1)

Complejidad: O(1)

2.2. Descripcion de Complejidades de Algoritmos

1. **ICATEGORIAS**:
   Se crea un conjunto vacio, esto tarda O(1). Se crea un itLista, esto tarda O(1).
   Se ingresa a un ciclo preguntando si haySiguiente a un iterador, esto cuesta O(1), se le agrega la categoria de cada tupla de datosCat de la lista ac.categorias, esto tarda O(|c|), luego se avanza el iterador, esto cuesta O(1). Salir del ciclo cuesta O(longitud(ac.caterorias))
   Orden Total:O(1)+O(1)+O(1)+(suma O(|c|))+O(1)=**O(suma O(|c|))** donde suma es una sumatoria hasta longitud(ac.caterorias) de O(|c|)

2. **IRAIZ**:
   Para que res sea la raiz necesitamos acceder a lo que apunta ac.raiz, que tarda O(1), y es una tupla que tiene el string categoria.
   Y copiar el string cuesta O(|c|) donde c es un string.
   Orden Total:O(|c|)

3. **IPADRE**:
   Para que res sea el padre, que es un puntero a categoria, necesitamos obtener el puntero de datosCat que lleva la clave h. Esto tarda O(|h|).
   Luego obtenemos lo que apunta, que nos da una tupla, para poder acceder a abuelo, que tambien es un puntero a datosCat. Obtenemos lo que apunta y accedemos a categoria para que sea copiada a res.
   Y copiar el string cuesta O(|c|) donde c es un string.
   Orden Total:O(|h| + |c|)

4. **IID**:
   Para que res sea el id necesitamos obtener la categoria c del diccTrie ac.familia, que tarda O(|c|) y como significado da un puntero a datosCat.
   Finalmente accedemos a lo apuntado para poder asignar a res el id de la tupla de datosCat
   Orden Total:O(|c|)

5. **INUEVO**:
   Res tiene que ser la tupla de la estructura.
   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|)**

6. **IAGREGAR**:
   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). En el caso contrario de la guarda, tambien hacemos una asignacion de nats, que tarda 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|)**

7. **IALTURA**:
   Para que res sea la altura, le asignamos ac.alturaMax, y al ser nat tarda O(1)
   Orden Total:O(1)

8. **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|)

9. **IESSUBCATEGORIA?**:
   Para ver si una categoria h es subcategoria de c en nuestra estructura ac, creamos un puntero de datosCat puntPadre, al cual le asignamos lo que nos da la operacion obtener, de la clave c en el diccTrie ac.familia. Esto tarda O(|c|). Luego inicializamos en falso res.
   Creamos la variable actual que es un puntero de datosCat. Esto tarda O(1). Luego comparamos el string c con la categoria que apunta ac.raiz. Esto tarda O(|c|). En caso que valga la guarda ponesmos res en true, que tarda O(1), en caso contrario asignamos a el puntero actual lo que nos da obtener la categoria h en ac.familia. Y esto tarda O(|h|).
   Seguimos con un ciclo en el que la guarda tarda O(1). Dentro preguntamos si actual pertenece al conjunto de punteros dado por los hijos de lo que apunta puntPadre. Esto tarda O(cantidad(hijos)). Si pertenece hacemos una asignacion que tarda O(1), sino asignamos a actual el abuelo de lo apuntado por el mismo puntero actual. Esto tarda O(1).
   El ciclo se ejecuta tantas veces como la altura de h. Quedando asi la complejidad del ciclo como la sumatoria hasta altura de h de la cantidad de hijos que tenga c, y eso es O(altura(h) * cantidad(hijos de c))
   Orden Total:O(|c| + |h| + (altura(h) * cantidad(hijos de c)))

10. **IALTURACATEGORIA?**:
    Para que res sea la altura de la categoria c en el arbolCategorias ac, le asignamos la altura de la tupla apuntada por el obtener de una categoria c en un diccTrie ac.familia. Y esto tarda O(|C|)
    Orden Total:O(|c|)

11. **IHIJOS**:
    Le asignamos a res un iterador de conjunto, creado por la operacion crearIt, que tarda O(1), pero se le pasa el conjunto de punteros dado por hijos de la tupla que apunta el obtener de una categoria c en un diccTrie ac.familia. Y esto tarda O(|c|)
    Orden Total:O(|c|)

12. **IOBTENER**:
    Le asignamos a res el conjunto de punteros datosCat que nos da la operacion de obtener de una categoria c en un diccTrie ac.familia. Y esto tarda O(|c|)
    Orden Total:O(|c|)

13. **IPUNTRAIZ**:
    Le asignamos a res el puntero de datosCat dado por nuestra estructura ac en ac.raiz. Esto tarda O(1)
    Orden Total:O(1)

3. Modulo DiccTrie

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 posicion del arreglo representa una letra y su contenido es un puntero al nodo de la letra siguiente o a Null.

3.0.1. Invariante de Representación

3.0.1.1. 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.

3.0.1.2. El Invariante Formalmente

Rep : estrDT $\rightarrow$ boolean

($\forall e$: `estrDT`) Rep(e) $\equiv$ true $\Longleftrightarrow$

1. $(\forall i, j$: `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. $(\neg\exists n$: `nat`$)EncAEstrDTEnNMov(e, e, n) \wedge_{\text{L}}$
3. $SonTodosNullOLosHijosLoSon(e)$

3.0.1.3. Funciones auxiliares

EncAEstrDTEnNMov : estrDT × estrDT × Nat $\longrightarrow$ Bool

EncAEstrDTEnNMov(buscado,actual,n) $\equiv$ **if** (n = 0) **then**
EstaEnElArregloActual?(buscado,actual,255)
else
RecurrenciaConLosHijos(buscado,actual, n-1,255)
fi

EstaEnElArregloActual? : estrDT × estrDT × nat $\longrightarrow$ Bool

EstaEnElArregloActual?(buscado,actual,n) $\equiv$ **if** (n=0) **then**
((*actual).Arreglo[0] = buscado)
else
((*actual).Arreglo[n] = buscado) $\vee$ (EstaEnElArregloActual?
(buscado,actual,n-1))
fi

RecurrenciaConLosHijos : estrDT × estrDT × nat × nat $\longrightarrow$ Bool

RecurrenciaConLosHijos(buscado,actual,n,i) $\equiv$ **if** (i = 0) **then**
EncAEstrDTEnNMov(buscado,(*actual).Arreglo[0],n)
else
EncAEstrDTEnNMov(buscado, (*actual).Arreglo[i],n) $\vee$
(RecurrenciaConLosHijos(buscado,actual,n,i-1)
fi

SonTodosNullOLosHijosLoSon : estrDT $\longrightarrow$ Bool

SonTodosNullOLosHijosLoSon(e) $\equiv$ Los256SonNull(e,255) $\vee$ BuscarHijosNull (e, 255)

Los256SonNull : estrDT $\times$ nat $\longrightarrow$ Bool

Los256SonNull(e,i) $\equiv$ **if** (i = 0) **then**
((*e).Arreglo[0] = null)
else
((*e).Arreglo[i] = null) $\wedge$ Los256SonNull(e, i-1)
fi

BuscarHijosNull : estrDT $\times$ nat $\longrightarrow$ Bool

BuscarHijosNull(e,i) $\equiv$ **if** (i = 0) **then**
((*e).Arreglo[0] = null) $\vee$ SonTodosNullOLosHijosLoSon((*e).Arreglo[0])
else
(((*e).Arreglo[i] = null) $\vee$ SonTodosNullOLosHijosLoSon((*e).Arreglo[i])) $\wedge$
BuscarHijosNull(e,i-1)
fi

3.0.2. Función de Abstracción

Abs : e: `estrDT` $\rightarrow$ diccT(c,α) Rep(e)

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

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

3.0.2.1. Funciones auxiliares

estaDefinido? : string $\times$ estrDT $\longrightarrow$ bool

estaDefinido?(c,e) $\equiv$ **if** (e==Null) **then** false **else** NodoDef?(c,*(e)) **fi**

NodoDef? : string $\times$ Nodo $\longrightarrow$ bool

NodoDef?(c,n) $\equiv$ **if** (vacia?(c)) **then**
true
else
if (n.arreglo[numero(prim(c))] $\neq$ Null) **then**
NodoDef?(fin(c),*(n.arreglo[numero(prim(c))]))
else
false
fi
fi

numero : char $\longrightarrow$ nat

numero(char) $\equiv$ char - a

ObtenerS : string × Nodo $\longrightarrow$ α

ObtenerS(c,n) $\equiv$ **if** (vacia?(c)) **then** *(n.significado) **else** ObtenerS(fin(c),*(n.arreglo[numero(prim(c))])) **fi**

EncAEstrDTEnNMov : estrDT × estrDT × Nat $\longrightarrow$ Bool

EncAEstrDTEnNMov(buscado,actual,n) $\equiv$ **if** (n = 0) **then**
EstaEnElArregloActual?(buscado,actual,255)
else
RecurrenciaConLosHijos(buscado,actual, n-1,255)
fi

EstaEnElArregloActual? : estrDT × estrDT × nat $\longrightarrow$ Bool

EstaEnElArregloActual?(buscado,actual,n) $\equiv$ **if** (n=0) **then**
((*actual).Arreglo[0] = buscado)
else
((*actual).Arreglo[n] = buscado) $\vee$ (EstaEnElArregloActual?
(buscado,actual,n-1))
fi

RecurrenciaConLosHijos : estrDT × estrDT × nat × nat $\longrightarrow$ Bool

RecurrenciaConLosHijos(buscado,actual,n,i) $\equiv$ **if** (i = 0) **then**
EncAEstrDTEnNMov(buscado,(*actual).Arreglo[0],n)
else
EncAEstrDTEnNMov(buscado, (*actual).Arreglo[i],n) $\vee$
(RecurrenciaConLosHijos(buscado,actual,n,i-1)
fi

SonTodosNullOLosHijosLoSon : estrDT $\longrightarrow$ Bool

SonTodosNullOLosHijosLoSon(e) $\equiv$ Los256SonNull(e,255) $\vee$ BuscarHijosNull (e, 255)

Los256SonNull : estrDT × nat $\longrightarrow$ Bool

Los256SonNull(e,i) $\equiv$ **if** (i = 0) **then**
((*e).Arreglo[0] = null)
else
((*e).Arreglo[i] = null) $\wedge$ Los256SonNull(e, i-1)
fi

BuscarHijosNull : estrDT × nat $\longrightarrow$ Bool

BuscarHijosNull(e,i) $\equiv$ **if** (i = 0) **then**
((*e).Arreglo[0] = null) $\vee$ SonTodosNullOLosHijosLoSon((*e).Arreglo[0])
else
(((*e).Arreglo[i] = null) $\vee$ SonTodosNullOLosHijosLoSon((*e).Arreglo[i])) $\wedge$
BuscarHijosNull(e,i-1)
fi

4. Modulo itLista

4.0.3. Interfaz

parámetros formales

géneros itListaUni
se explica con: Iterador Unidireccional

Operaciones

CREARIT(**in** l: `lista`)→ res: **itListaUni**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} crearItUni(<>, elems(l)) \wedge alias(secuSuby(res) =_{\text{obs}} elems(l))\}$
Complejidad: O(1)
Aliasing: No se pueden modificar los mensajes del iterador

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

SIGUIENTE(**in** it: `itListaOrd`)→ res: **mensaje**
Pre ≡ $\{haySiguiente?(it)\}$
Post ≡ $\{esAlias(res, siguiente(it))\}$
Complejidad: O(1)
Aliasing: res no es modificable

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

4.0.4. Representación

itListaUni se representa con itLista(puntero(α))
itListaOrdRecientes **se representa con** estrITR, donde estrITR es $iterador$: `itLista(puntero(`α`))`

4.0.5. Invariante de Representación

4.0.5.1. El Invariante Formalmente

Rep : estrITR → boolean

($\forall it$: `estrITR`) Rep(it) ≡ true

4.0.6. Función de Abstracción

Abs : e: `estrITR` → itUni(α) Rep(e)

($\forall e$: `estrITR`) Abs(e) =$_{\text{obs}}$ it: itUni(α) |

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

4.0.7. Algoritmos

Algoritmo 1 iCrearItL

```
1: ICREARITLISTAORD(in l: estrL)→ res: estrITR
2:     res ← crearIt(l)                                          //O(1)
```

Complejidad: O(1)

Algoritmo 2 iHaySiguiente?

```
1: IHAYSIGUIENTE?(in e: itLista(puntero(α)))→res: bool
2:     res ← haySiguiente(e)                                     //O(1)
```

Complejidad: O(1)

Algoritmo 3 iSiguiente

```
1: ISIGUIENTE(in e: itLista(puntero(α)))→ res: (puntero(α))
2:     res ← *(siguiente(e))                                     //O(1)
```

Complejidad: O(1)

Algoritmo 4 iAvanzar

```
1: IAVANZAR(in/out e: itLista(puntero(α)))
2:     avanzar(e)                                                //O(1)
```

Complejidad: O(1)

4.0.8. Analisis de complejidades

1. **iCrearIt**

   Se crea un Iterador de Lista en 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).

   Orden Total: O(1)

4. **iAvanzar**

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

   Orden Total: O(1)

5. Modulo itConj

5.0.9. Interfaz

parámetros formales

géneros itConjUni
se explica con: Iterador Unidireccional

Operaciones

CREARIT(**in** l: `Conj`)→ res: **itConjUni**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} crearItUni(<>, elems(l)) \wedge alias(secuSuby(res) =_{\text{obs}} elems(l))\}$
Complejidad: O(1)
Aliasing: No se pueden modificar los mensajes del iterador

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

SIGUIENTE(**in** it: `itListaOrd`)→ res: **mensaje**
Pre ≡ $\{haySiguiente?(it)\}$
Post ≡ $\{esAlias(res, siguiente(it))\}$
Complejidad: O(1)
Aliasing: res no es modificable

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

5.0.10. Representación

itConjUni se representa con itLista(puntero(α))
itConjUni **se representa con** estrITR, donde estrITR es *iterador*: `itConj(puntero(`α`))`

5.0.11. Invariante de Representación

5.0.11.1. El Invariante Formalmente

Rep : estrITR → boolean

($\forall$ it: `estrITR`) Rep(it) ≡ true

5.0.12. Función de Abstracción

Abs : e: `estrITR` → itUni(α) Rep(e)

($\forall$ e: `estrITR`) Abs(e) =$_{\text{obs}}$ it: itUni(α) |

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

5.0.13. Algoritmos

Algoritmo 5 iCrearItL

```
1: ICREARIT(in l: estrL)→ res: estrITR
2:     res ← crearIt(l)                                          //O(1)
```

Complejidad: O(1)

Algoritmo 6 iHaySiguiente?

```
1: IHAYSIGUIENTE?(in e: itLista(puntero(α)))→res: bool
2:     res ← haySiguiente(e)                                     //O(1)
```

Complejidad: O(1)

Algoritmo 7 iSiguiente

```
1: ISIGUIENTE(in e: itLista(puntero(α)))→ res: (puntero(α))
2:     res ← *(siguiente(e))                                     //O(1)
```

Complejidad: O(1)

Algoritmo 8 iAvanzar

```
1: IAVANZAR(in/out e: itLista(puntero(α)))
2:     avanzar(e)                                                //O(1)
```

Complejidad: O(1)

5.0.14. Analisis de complejidades

1. **iCrearIt**

   Se crea un Iterador de Conjunto Lineal en O(1).

   Orden Total: O(1)

2. **iHaySiguiente?**

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

   Orden Total: O(1)

3. **iSiguiente**

   Se llama a Siguiente del Iterador de Conjunto Lineal en O(1).

   Orden Total: O(1)

4. **iAvanzar**

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

   Orden Total: O(1)

6. Renombres

TAD Categoria
es String

Fin TAD

TAD Link
es String

Fin TAD

TAD Fecha
es Nat

Fin TAD