1. Módulo ÁrbolCategorías

Interfaz

se explica con: ÁrbolCategorías, Iterador Unidireccional(Categoría).

géneros: `acat`, `datoscat`, `itercat`.

Operaciones básicas de árbol de categorías

CrearÁrbol(**in** *raiz*: `categoria`) → *res* : `acat`
Pre ≡ {¬vacía?(*raiz*)}
Post ≡ {*res* =$_{\text{obs}}$ nuevo(*raiz*)}
Complejidad: Θ(|*raiz*|)
Descripción: crea un árbol nuevo cuya categoría raíz es *raiz*.

NombreCategoríaRaíz(**in** *ac*: `acat`) → *res* : `categoria`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ raíz(*ac*)}
Complejidad: Θ(1)
Descripción: devuelve el nombre de la categoría raíz de *ac*.

AgregarCategoría(**in** *hija*: `categoria`, **in** *padre*: `categoria`, **in/out** *ac*: `acat`)
Pre ≡ {*ac* =$_{\text{obs}}$ ac_0 ∧ está?(*padre*, *ac*) ∧ ¬vacía?(*hija*) ∧ ¬está?(*hija*, *ac*)}
Post ≡ {*ac* =$_{\text{obs}}$ agregar(ac_0, *padre*, *hija*)}
Complejidad: Θ(|*padre*| + |*hija*|)
Descripción: agrega la categoría *hija* como hija de la categoría *padre*.

CrearIterCat(**in** *padre*: `categoria`, **in** *ac*: `acat`) → *res* : `itercat`
Pre ≡ {está?(*padre*, *ac*)}
Post ≡ {alias(esPermutacion?(SecuSuby(*res*), hijos(*ac*, *padre*))) ∧ vacia?(Anteriores(*res*))}
Complejidad: Θ(|*padre*|)
Descripción: devuelve un iterador unidireccional de las categorías hijas directas de la categoría *padre*.

DUDA: ¿Puedo tratar a *res* acá directamente como un itConj(α) en la expresión SecuSuby(*res*)?

DUDA: ¿Hay que extender el TAD ÁrbolCategorías como en el apunte de módulos básicos para poder especificar la operación esPermutacion??

IdCategoríaPorNombre(**in** *c*: `categoria`, **in** *ac*: `acat`) → *res* : `nat`
Pre ≡ {está?(*c*, *ac*)}
Post ≡ {*res* =$_{\text{obs}}$ id(*ac*, *c*)}
Complejidad: Θ(|*c*|)
Descripción: devuelve el *id* de la categoría *c*.

Operaciones de datos de categoría

DUDA: ¿Hace falta crear un TAD para el género datoscat para poder especificar las pre y postcondiciones de las funciones a continuación?

ObtenerId(**in** *dc*: `datoscat`) → *res* : `nat`
Pre ≡ {???}
Post ≡ {???}
Complejidad: Θ(1)
Descripción: devuelve el *id* de la categoría asociada a *dc*.

ObtenerNombre(**in** *dc*: `datoscat`) → *res* : `nat`
Pre ≡ {???}
Post ≡ {???}
Complejidad: Θ(1)
Descripción: devuelve el nombre de la categoría asociada a *dc*.

ObtenerPadre(**in** *dc*: `datoscat`) → *res* : `puntero(datoscat)`
Pre ≡ {???}
Post ≡ {???}
Complejidad: Θ(1)
Descripción: devuelve un puntero a los datos de la categoría padre asociada a *dc*.

ObtenerHijos(**in** *dc*: `datoscat`) → *res* : `conj(puntero(datoscat))`
Pre ≡ {???}
Post ≡ {???}
Complejidad: Θ(1)
Descripción: devuelve un conjunto de punteros a los datos de las categorías hijas directas asociadas a *dc*.

Operaciones de iterador de categorías

HayMás?(**in** *it*: `itercat`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ HayMás?(*it*)}
Complejidad: Θ(1)
Descripción: devuelve **true** si y sólo si en el iterador todavía quedan elementos para avanzar.

Actual(**in** *it*: `itercat`) → *res* : `puntero(datoscat)`
Pre ≡ {HayMás?(*it*)}
Post ≡ {*res* =$_{\text{obs}}$ Actual(*it*)}
Complejidad: Θ(1)
Descripción: devuelve el elemento actual del iterador.

Avanzar(**in/out** *it*: `itercat`)
Pre ≡ {*it* =$_{\text{obs}}$ it_0 ∧ HayMás?(*it*)}
Post ≡ {*res* =$_{\text{obs}}$ Avanzar(it_0)}
Complejidad: Θ(1)
Descripción: avanza el iterador a la posición siguiente.

Representación

Representación de árbol de categorías

`acat` **se representa con** `estr_acat`

donde `estr_acat` es `tupla`(*raíz*: `puntero(datoscat)`, *categorías*: `dicc_trie(datoscat)`)

Rep : lst ⟶ bool
Rep(l) ≡ true ⟺ (l.primero = NULL) = (l.longitud = 0) ∧$_{\text{L}}$ (l.longitud ≠ 0 ⇒$_{\text{L}}$
Nodo(l, l.longitud) = l.primero ∧
(∀i: nat)(Nodo(l,i)→siguiente = Nodo(l,$i+1$)→anterior) ∧
(∀i: nat)(1 ≤ i < l.longitud ⇒ Nodo(l,i) ≠ l.primero)

Abs : lst l ⟶ secu(α) {Rep(l)}
Abs(l) ≡ **if** l.longitud = 0 **then** <> **else** l.primero→dato • Abs(FinLst(l)) **fi**

Representación de datos de categoría

`datoscat` **se representa con** `estr_datoscat`

donde `estr_datoscat` es `tupla`(*id*: `nat`, *nombre*: `categori<`, *padre*: `puntero(datoscat)`, *hijos*: `conj(puntero(datoscat))`)

DUDA: ¿Hace falta crear un TAD nuevo para poder expresar el Rep y Abs del género datoscat? De ser así, ¿tengo que agregar dicho TAD en la lista 'se explica con'?

Representación de iterador de categorías

`itercat` **se representa con** `itConj(puntero(datoscat))`

DUDA: ¿Está bien la estructura de representación elegida?
DUDA: ¿Qué usamos para expresar el Rep y el Abs en este caso? Dado que se trata de un itConj(α), suena razonable pensar en usar el Rep y Abs de este iterador definidos en el módulo Conjunto Lineal(α).

Algoritmos

pendiente

2. Módulo Lista Enlazada(α)

Interfaz

parámetros formales
géneros α
función COPIAR(**in** a : α) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's

se explica con: SECUENCIA(α), ITERADOR BIDIRECCIONAL(α).

géneros: `lista`(α), `itLista`(α).

Operaciones básicas de lista

VACÍA() $\rightarrow$ res : `lista`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} <>$}
Complejidad: $\Theta(1)$
Descripción: genera una lista vacía.

AGREGARADELANTE(**in/out** l : `lista`(α), **in** a : α) $\rightarrow$ res : `itLista`(α
Pre $\equiv$ {$l =_{\text{obs}} l_0$}
Post $\equiv$ {$l =_{\text{obs}} a \bullet l_0 \wedge res =$ CrearItBi($<>$, l) $\wedge$ alias(SecuSuby(res) = l)}
Complejidad: $\Theta(copy(a))$
Descripción: agrega el elemento a como primer elemento de la lista. Retorna un iterador a l, de forma tal que Siguiente devuelva a.
Aliasing: el elemento a agrega por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE.

Operaciones del iterador

CREARIT(**in** l : `lista`(α)) $\rightarrow$ res : `itLista`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearItBi($<>$, l) $\wedge$ alias(SecuSuby(it) = l)}
Complejidad: $\Theta(1)$
Descripción: crea un iterador bidireccional de la lista, de forma tal que al pedir SIGUIENTE se obtenga el primer elemento de l.
Aliasing: el iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE.

CREARITULT(**in** l : `lista`(α)) $\rightarrow$ res : `itLista`(α)

Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearItBi(l, <>) $\wedge$ alias(SecuSuby(it) = l)}
Complejidad: $\Theta(1)$
Descripción: crea un iterador bidireccional de la lista, de forma tal que al pedir ANTERIOR se obtenga el último elemento de l.
Aliasing: el iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE.

Representación

Representación de la lista

`lista`(α) **se representa con** `lst`

donde `lst` es `tupla`(*primero*: `puntero(nodo)`, *longitud*: `nat`)
donde `nodo` es `tupla`(*dato*: α, *anterior*: `puntero(nodo)`, *siguiente*: `puntero(nodo)`)

Rep : lst $\longrightarrow$ bool
Rep(l) $\equiv$ true $\iff$ (l.primero = NULL) = (l.longitud = 0) $\wedge_{\text{L}}$ (l.longitud $\neq$ 0 $\Rightarrow_{\text{L}}$
Nodo(l, l.longitud) = l.primero $\wedge$
($\forall i$: nat)(Nodo(l,i)$\rightarrow$siguiente = Nodo(l,$i+1$)$\rightarrow$anterior) $\wedge$
($\forall i$: nat)($1 \leq i < l$.longitud $\Rightarrow$ Nodo(l,i) $\neq$ l.primero)

Nodo : lst l $\times$ nat $\longrightarrow$ puntero(nodo) {l.primero $\neq$ NULL}
Nodo(l,i) $\equiv$ **if** $i = 0$ **then** l.primero **else** Nodo(FinLst(l), $i-1$) **fi**

FinLst : lst $\longrightarrow$ lst
FinLst(l) $\equiv$ Lst(l.primero$\rightarrow$siguiente, l.longitud $-$ mín{l.longitud, 1})

Lst : puntero(nodo) $\times$ nat $\longrightarrow$ lst
Lst(p, n) $\equiv$ $\langle p, n \rangle$

Abs : lst l $\longrightarrow$ secu(α) {Rep(l)}
Abs(l) $\equiv$ **if** l.longitud = 0 **then** <> **else** l.primero$\rightarrow$dato $\bullet$ Abs(FinLst(l)) **fi**

Representación del iterador

`itLista`(α) **se representa con** `iter`

donde `iter` es `tupla`(*siguiente*: `puntero(nodo)`, *lista*: `puntero(lst)`)

Rep : iter $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ Rep($*$(it.lista)) $\wedge_{\text{L}}$ (it.siguiente = NULL $\vee_{\text{L}}$ ($\exists i$: nat)(Nodo($*it$.lista, i) = it.siguiente)

Abs : iter it $\longrightarrow$ itBi(α) {Rep(it)}
Abs(it) $=_{\text{obs}}$ b: itBi(α) | Siguientes(b) = Abs(Sig(it.lista, it.siguiente)) $\wedge$
Anteriores(b) = Abs(Ant(it.lista, it.siguiente))

Sig : puntero(lst) l $\times$ puntero(nodo) p $\longrightarrow$ lst {Rep($\langle l, p \rangle$)}
Sig(i, p) $\equiv$ Lst(p, $l\rightarrow$longitud $-$ Pos($*l$, p))

Ant : puntero(lst) l $\times$ puntero(nodo) p $\longrightarrow$ lst {Rep($\langle l, p \rangle$)}
Ant(i, p) $\equiv$ Lst(**if** $p = l\rightarrow$primero **then** NULL **else** $l\rightarrow$primero **fi**, Pos($*l$, p))

Nota: cuando p = NULL, Pos devuelve la longitud de la lista, lo cual está bien, porque significa que el iterador no tiene siguiente.
Pos : lst l $\times$ puntero(nodo) p $\longrightarrow$ puntero(nodo) {Rep($\langle l, p \rangle$)}
Pos(l,p) $\equiv$ **if** l.primero = $p \vee l$.longitud = 0 **then** 0 **else** 1 + Pos(FinLst(l), p) **fi**