DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio del Trabajo Práctico II

PokemonGoArgentina Diseño

Algoritmos y Estructuras de Datos II
Segundo Cuatrimestre de 2016

Grupo: Wololobot

Integrante	LU	Correo electrónico
Alejandro Dario Echeverri	939/05	ale_echeverri@yahoo.com.ar
Lucas Monzon	785/14	lucasmonzon94@gmail.com
Arroyo Luis	913/13	luis.arroyo.90@gmail.com
Manuel Cafferata	592/14	manucaferacing@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1 Módulo Coordenada **3**

2 Módulo MulticonjStr **6**

3 Módulo ColaDePrioridad(α, $<_\alpha$) **10**

4 Módulo DiccString(α) **15**

5 Módulo DiccMatriz(α) **20**

6 Módulo Mapa **23**

7 Módulo Juego **26**

8 Observaciones **37**

1 Módulo Coordenada

Interfaz

se explica con: COORDENADA.

géneros: `coor`.

función: $\bullet = \bullet$(**in** c_1 : `coor`, **in** c_2 : `coor`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (c_1 = c_2$}
Complejidad: $\Theta(1)$
Descripción: Función de igualdad de coor

función: COPIAR(**in** c: `coor`) $\rightarrow$ res : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ c}
Complejidad: $\Theta(1)$
Descripción: Funcion copia de coor

Operaciones básicas de la coor

NUEVACOOR(**in** la : `nat`, **in** lo : `nat`) $\rightarrow$ res : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} crearCoor(la, lo)$}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea una nueva coordenada con latitud y longitud la y lo respectivamente.

LATITUD(**in** c: `coor`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ latitud(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la latitud de la coordenada c.

LONGITUD(**in** c: `coor`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ longitud(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la latitud de la coordenada c.

DISTEUCLIDEA(**in** $c1$: `coor`, **in** $c2$: `coor`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ distEuclidea(c1, c2)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el cuadrado de la distancia euclidea entre c1 y c2.

Representación

Representación de la coor

`coor` **se representa con** `estr`

donde `estr` es `tupla`(*latitud*: `nat` , *longitud*: `nat`)

Invariante en castellano

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ true

Abs : estr e $\longrightarrow$ coordenada {Rep(e)}

Abs(e) $\equiv$ c:coordenada |
e.latitud = latitud(c) $\wedge$
e.longitud = longitud(c)

Algoritmos

`InuevaCoor(`**`in`** lat `: nat, `**`in`** $long$ `: nat) → res:coor`

```
1: res.latitud <- lat                                   O(1)
2: res.longitud <- long                                 O(1)
```

Complejidad: $\mathcal{O}(1)$.

`ILatitud(`**`in`** c `: estr) → res:nat`

```
1: res <- c.latitud                                     O(1)
```

Complejidad: $\mathcal{O}(1)$.

`ILongitud(`**`in`** c `: estr) → res:nat`

```
1: res <- c.longitud                                    O(1)
```

Complejidad: $\mathcal{O}(1)$.

`IdistEuclidea(`**`in`** $c1$ `: estr, `**`in`** $c2$ `: estr) → res:nat`

```
1: res <- cuadRestaLat(c1,c2) + cuadRestaLong(c1,c2)    O(1)
```

Complejidad: $\mathcal{O}(1)$.

CUADRESTALAT(**in** $c1$: `coor`, **in** $c2$: `coor`) → res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ cuadRestaLat(c1, c2)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el cuadrado de la resta entre la latitud de c1 y la de c2.

IcuadRestaLat(**in** $c1$: estr, **in** $c2$: estr) $\rightarrow$ res:nat

```
1: if ( c1.latitud > c2.latitud ) then
2:     res <- ( c1.latitud - c2.latitud ) * ( c1.latitud - c2.latitud )        O(1)
3: else
4:     res <- ( c2.latitud - c1.latitud ) * ( c2.latitud - c1.latitud )        O(1)
5: end if
```

Complejidad: $\mathcal{O}(1)$.

CUADRESTALONG(**in** $c1$: coor, **in** $c2$: coor) $\rightarrow res$: nat
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ cuadRestaLong(c1, c2)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el cuadrado de la resta entre la longitud de c1 y la de c2.

IcuadRestaLong(**in** $c1$: estr, **in** $c2$: estr) $\rightarrow$ res:nat

```
1: if ( c1.longitud > c2.longitud ) then
2:     res <- ( c1.longitud - c2.longitud ) * ( c1.longitud - c2.longitud )    O(1)
3: else
4:     res <- ( c2.longitud - c1.longitud ) * ( c2.longitud - c1.longitud)     O(1)
5: end if
```

Complejidad: $\mathcal{O}(1)$.

Especificación de las operaciones auxiliares utilizadas que no se exportan

cuadRestaLat : coor $\times$ coor $\longrightarrow$ nat

cuadRestaLat(c_1, c_2) $\equiv$ **if** latitud(c_1) > latitud(c_2) **then**
(latitud(c_1) - latitud(c_2)) * (latitud(c_1) - latitud(c_2))
else
(latitud(c_2) - latitud(c_1)) * (latitud(c_2) - latitud(c_1))
fi

cuadRestaLong : coor $\times$ coor $\longrightarrow$ nat

cuadRestaLong(c_1, c_2) $\equiv$ **if** longitud(c_1) > longitud(c_2) **then**
(longitud(c_1) - longitud(c_2)) * (longitud(c_1) - longitud(c_2))
else
(longitud(c_2) - longitud(c_1)) * (longitud(c_2) - longitud(c_1))
fi

Servicios Usados: Nat

2 Módulo MulticonjStr

Interfaz

parámetros formales:

géneros: α

función: $\bullet = \bullet$(**in** m_1 : `multiconjStr`, **in** m_2 : `multiconjStr`) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} (m_1 = m_2)\}$
Complejidad: $\mathcal{O}(n * |ls|)$. Donde n es cantidad de string distintos de m1 y ls es la longitud del string más largo de m1
Descripción: Función de igualdad de Multiconjunto de String.

función: COPIAR(**in** m : `multiconj`(α)) $\rightarrow res$: `multiconj`(α)
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} d\}$
Complejidad: $\mathcal{O}(n * |ls|)$. Donde n es cantidad de string distintos de m y ls es la longitud del string más largo de m
Descripción: Función copia de Multiconjunto de String.

se explica con: MULTICONJUNTO($string$).

géneros: `multiconjStr`

Operaciones básicas del multiconjunto

REPETICIONES(**in** mc : `multiconjStr`, **in** a : `string`) $\rightarrow res$: `nat`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} \#(a, mc)\}$
Complejidad: $\mathcal{O}(|a|))$
Descripción: Devuelve la cantidad de repeticiones del elemento.

VACÍO( ) $\rightarrow res$: `multiconjStr`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}} \emptyset$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un multiconjunto de string vacío.

AGREGAR(**in/out** mc : `multiconjStr`, **in** a : `string`)
Pre $\equiv$ {mc = mc_0}
Post $\equiv$ {mc $=_{\text{obs}}$ Ag(a, mc_0)}
Complejidad: $\mathcal{O}(|a|))$
Descripción: Agrega el elemento.

CARDINAL(**in** mc : `multiconjStr`) $\rightarrow res$: `nat`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \#(mc)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad elementos que posee el multiconjunto de string.

Operaciones del iterador

El iterador diseñado permite recorrer los elementos unidireccionalmente.

CREARIT(**in** mc: `multiconjStr`) $\rightarrow$ res : `itMulticonjStr`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), mc))}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un iterador unidireccional del multiconjunto.

HAYSIGUIENTE(**in** it : `itMulticonjStr`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ haySiguiente?(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve **true** si y sólo si en el iterador todavía quedan elementos para avanzar.

SIGUIENTE(**in** it : `itMulticonjStr`) $\rightarrow$ res : `tupla`($string, nat$)
Pre $\equiv$ {haySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ siguiente(it))}
Complejidad: $\mathcal{O}(|a|)$. Con a = siguiente string
Descripción: Devuelve el elemento siguiente del iterador.
Aliasing: res no es modificable.

AVANZAR(**in/out** it : `itMulticonjStr`)
Pre $\equiv$ {$it = it_0 \wedge$ haySiguiente?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza a la posición siguiente del iterador.

Representación

Representación del multiconjStr

`multiconjstr` **se representa con** `estr`

donde `estr` es `tupla`(*cardinal*: `nat` , *elems*: `DiccString(repeticiones:nat)`)

Invariante en castellano

El cardinal del multiconjunto es igual a la suma de los significados del diccionario.

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ (e.cardinal = sumaCardinales(e.elems))

sumaCardinales : dicc($\alpha \times$ nat) $\longrightarrow$ nat

sumaCardinales(d) $\equiv$ sumarSignficados(d,claves(d))

sumarSiginificados : dicc($\alpha \times$ nat) $d \times$ conj(α) c $\longrightarrow$ nat {c $\subseteq$ claves(d)}

sumarSignificados(d,c) $\equiv$ **if** $\emptyset$(c) **then**
0
else
obtener(dameUno(c),d) + sumarSignficados(d, sinUno(claves(d)))
fi

Abs : estr e $\longrightarrow$ multiconjunto(α) {Rep(e)}

Abs(e) $\equiv$ mc : multiconjunto(α) | ($\forall$ x: α)(#(x,mc) = repeticiones(x,e.elems))

repeticiones : $\alpha \times$ dicc($\alpha \times$ nat) $\longrightarrow$ nat

repeticiones(x,d) $\equiv$ **if** def?(x,d) **then** obtener(x,d) **else** 0 **fi**

Representación del iterador

`ItMulticonjStr` **se representa con** `iterMs`

donde `iterMS` es `tupla`(*palabras*: `itConj(string)` , *cantApariciones*: `nat`)

Algoritmos

`Ivacío() → res:multiconj(`α`)`

```
1: res.elems <- vacío()                                   O(1)
2: res.cardinal <- 0                                      O(1)
```

Complejidad: $\mathcal{O}(1)$.

`Irepeticiones(`**`in`** e `: estr, `**`in`** a `: string) → res:nat`

```
1: if (definido?(e.elems,a))                              O(|a|)
2:     res <- significado(e.elems,a)                      O(|a|)
3: else
4:     res <- 0                                           O(1)
5: end if
```

Complejidad: $\mathcal{O}(|a|)$.

`Iagregar(`**`in/out`** mc `: estr, `**`in`** a `: string)`

```
1: if (definido?(e.elems,a))                              O(|a|)
2:     definir(e.elems,a, significado(e.elems,a) + 1)     O(|a|)
3: else
4:     definir(e.elems,a, 1)                              O(|a|)
5: e.cardinal <- e.cardinal + 1                           O(1)
```

Complejidad:$\mathcal{O}(|a|)$

`Icardinal(`**`in`** e `: estr) → res:nat`

```
1: res <- e.cardinal                                      O(1)
```

Complejidad: $\mathcal{O}(1)$.

IcreatIt(**in** e : estr) $\rightarrow$ res:itMulticonjStr

```
1: res.palabras <- clavesDicc(e.elems)                                  O(1)
2: res.cantApariciones <- 0                                             O(1)
```

Complejidad: $\mathcal{O}(1)$.

IhaySiguiente(**in** it : iterMS) $\rightarrow$ res:bool

```
1: res <- haySiguiente(it.palabras)                                     O(1)
```

Complejidad: $\mathcal{O}(1)$.

Isiguiente(**in** it : iterMS) $\rightarrow$ res:tupla(string,nat)

```
1: Π1(res) <-siguiente(it.palabras)                                     O(1)
2: Π2(res) <-significado(e.elems,siguiente(it.palabras))                O(|siguiente(it.palabras)|)
```

Complejidad: $\mathcal{O}(|siguiente(it.palabras)|)$.

Iavanzar(**in/out** it : iterMS)

```
1: avanzar(it.palabras)                                                 O(1)
```

Complejidad: $\mathcal{O}(1)$.

3 Módulo ColaDePrioridad(α, $<_\alpha$)

Interfaz

se explica con: COORDENADA.

géneros: `coor`.

función: • = •(**in** c_1 : `cola`, **in** c_2 : `cola`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} (c_1 = c_2$}
Complejidad: $\Theta(n + m)$ donde n y m son la cantidad de elementos de las colas
Descripción: Función de igualdad de cola

función: COPIAR(**in** c: `cola`) → res : `cola`
Pre ≡ {true}
Post ≡ {res $=_{\text{obs}}$ t}
Complejidad: $\Theta(n)$ donde n es la cnatidad de elementos de c
Descripción: Funcion copia de cola

Operaciones básicas de la coor

VACIA() → res : `cola`(α)
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} vacia$}
Complejidad: $\mathcal{O}(1)$
Descripción: Genera 1 cola vacía.

ENCOLAR(**in/out** c: `cola`(α), **in** a : α)
Pre ≡ {c = c0}
Post ≡ {c $=_{\text{obs}}$ encolar(c0,a)}
Complejidad: $\mathcal{O}(log(tam(c))$
Descripción: Encola a a c.
Aliasing: El elemento a se encola por copia.

ESVACIA?(**in** c: `cola`(α)) → res : `bool`
Pre ≡ {true}
Post ≡ {res $=_{\text{obs}}$ vacia?(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve true si y solo si la cola es vacia.

PROXIMO(**in** c: `cola`(α)) → res : α
Pre ≡ {¬vacia?(c)}
Post ≡ {alias(res $=_{\text{obs}}$ proximo(c))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el proximo de la cola.
Aliasing: res es modificable si y solo si α es modificable

DESENCOLAR(**in/out** c: `cola`(α))
Pre ≡ {c = c0 ∧ ¬vacia?(c)}
Post ≡ {c $=_{\text{obs}}$ desencolar(c0)}
Complejidad: $\mathcal{O}(log(tam(c)))$
Descripción: Desencola el proximo de c.

TAMAÑO(**in** c: `cola`(α)) → res : `nat`
Pre ≡ {true}

Post $\equiv$ {res $=_{\text{obs}}$ tamaño(c)}
Complejidad: $\mathcal{O}(1)$)
Descripción: Devuelve la cantidad de elementos de c.

Representación

Representación de la cola

`cola` **se representa con** `cola`

donde `cola` es `tupla`(*raiz*: `puntero(nodo)` , *prioridad*: $<_\alpha$))

donde `nodo` es `tupla`(*izq*: `puntero(nodo)` , *der*: `puntero(nodo)` , *padre*: `puntero(nodo)` , *altura*: `nat` , *tamIzq*: `nat` , *tamDer*: `nat` , *val*: α)

Invariante en castellano

1. El árbol no tiene ciclos.
2. No hay dos padres que apunten al mismo hijo.
3. La raiz es el nodo de mayor prioridad.
4. La prioridad de los hijos siempre es menor a la del padre.
5. Es izquierdista. (completo??)

Rep : coor $\longrightarrow$ bool

Rep(c) $\equiv$ true $\Longleftrightarrow$ true

Abs : coor c $\longrightarrow$ Coordenada {Rep(c)}

Abs(c) $\equiv$ co:Coordenada |
c.latitud = latitud(co) $\wedge$
c.longitud = longitud(co)

Algoritmos

`Ivacia(in` p : $<_\alpha$`)` $\rightarrow$ `res:cola(`α, $<_\alpha$`)`

```
1: res.raiz <- NULL                                              O(1)
2: res.prioridad <- p                                            O(1)
```

Complejidad: $\mathcal{O}(1)$.

`Iencolar(in/out` c : `cola(`α, $<_\alpha$`), in` a : α`)`

```
1: //en esta fase vamos a inicializar el nodo
2: encolado <- new Nodo(NULL,NULL,NULL,0,0,1,a)
3: //si es vacio lo encolamos como raiz, sino buscamos
4: //la posicion mas a la izquierda del arbol
5: if(c.raiz == NULL) then                                       O(1)
6:     c.raiz <- encolado
7: else
8:     dondeAgregar <- Buscarmodificable(c.raiz,encolado)        O(log(tam(c)))
9:     sift-UP (c.raiz,dondeagregar)                             O(log(tam(c)))
```

Complejidad: $\mathcal{O}(log(tam(c))$.

`Ieliminar(in/out` c : `cola(`α, $<_\alpha$`),in` $Aborrar$: `puntero(nodo))`

```
1: //si es el primer nodo, lo eliminamos
2: if(c.raiz == Aborrar) then                                              O(1)
3:     c.raiz <- NULL
4:     delete Aborrar
5: else
6:     ultimo <- UltimoAgregado(c.raiz)                                    O(log(tam(c)))
7:     Aborrar.val <- ultimo.val
8:     if (soyHijoDerecho(ultimo)) then
9:        ultimo.padre.der <- NULL
10:   else
11:       ultimo.padre.izq <- NULL
12:    reconstruir(ultimo)
13:    delete(ultimo)
14:    sift-DOWN (Aborrar)                                                 O(log(tam(c))
15:    sift-UP (Aborrar)                                                   O(log(tam(c))
```

Complejidad: $\mathcal{O}(log(tam(c))$.

IBuscarmodificable (**in** *buscador* : puntero(nodo), **in** a : α) $\rightarrow$ res:puntero(nodo)

```
1: // si no tiene hijos estamos en un caso base, osea agregamos a la izquierda
2: if(SinHijos(buscador) then
3:     buscador.izq <- new Nodo(NULL,NULL,NULL,0,0,1,a)
4:     buscador.tamIzq ++
5:    buscador.altura ++
6:     res <- buscador.izq
7: else
8:    if(UnHijo(buscador)) then                                            O(1)
9:        //si tiene un solo hijo necesariamente hay que agregar hacia la derecha
10:       buscador.tamDer ++
11:       buscador.der<- new Nodo(NULL,NULL,NULL,0,0,1,a)
12:       res <- buscador.der
13:   else
14:    si tiene dos hijos, entonces hay que buscar en cual de los dos subarboles hay que
   avanzar
15:       if(completo(buscador.der,buscador.altura-1) && completo(buscador.izq,buscador.altura-1)
   then
16:       buscador.altura++
17:       buscador.tamIzq++
18:       BuscarModificable(buscador.izq)
19:    else
20:       if(completo(buscador.tamizq,buscador.altura-1) then
21:          buscador.tamDer++
22:          BuscarModificable(buscador.der)
23:       else
24:         buscador.izq++
25:         BuscarModificable(buscador.izq)
```

Complejidad: $\mathcal{O}(log(tam(c))$. **justificacion:** debido a que es un algoritmo recursivo que en cada iteracion deja a la mitad de la cantidad de los nodos del arbol de lado, se demuestra por medio del teorema maestro con a=1 b = 2 y f(n) = $\mathcal{O}(1)$

IUltimoAgregado (**in** *buscador* : puntero(nodo), **in** a : α)

```
1: if(SinHijos(buscador) then
2:     res <- buscador
3: else
4:    if(UnHijo(buscador)) then                                            O(1)
```

```
5:        res <- buscador.izq
6:    else
7:        if(completo(buscador.der,buscador.altura-1) && completo(buscador.izq,buscador.altura-1)
   then
8:        UltimoAgregado(buscador.der)
9:     else
10: // el siguiente if sirve para saber si el arbol de la izquierda esta completo y el arbol de
   la derecha esta completamente "vacio" respecto del ultimo nivel, permitiendonos saber que
   debemos avanzar hacia la izquierda para llegar al ultimo que fue agregado
11:       if(completo(buscador.izq,buscador.altura-1) && completo(buscador.izq,buscador.altura-2)
   ) then
12:          ultimoAgregado(buscador.izq)
13:       else
14:         ultimoAgregado(buscador.der)
```

Complejidad: $\mathcal{O}(log(tam(c))$. **justificacion:** debido a que es un algoritmo recursivo que en cada iteracion deja a la mitad de la cantidad de los nodos del arbol de lado, se demuestra por medio del teorema maestro con a=1 b = 2 y f(n) = $\mathcal{O}(1)$

```
Ireconstruir (in armando: puntero(nodo), in a: α)
1: if(TieneHemano(armando) then
2:     armando.padre.tamDer--
3: else
4:    recorre <-armando
5:    recorre.izq--
6:    recorre.altura--
7:     while noHijoDerecho(recorre) do
8:       recorre.izq--
9:       recorre.altura--
10:       end while
```

Complejidad: $\mathcal{O}(log(tam(c))$. **justificacion:** ya que, en el peor de los casos, tiene que reconstruir el invariante hasta la raiz.

```
iSift-UP (in Nodo: puntero(nodo))
1: recorre <- Nodo
2: while recorre.padre != NULL && recorre.val < recorre.padre.val do
3:    swap <- recorre.padre.val
4:    recorre.padre <- recorre.val
5:    recorre.val <-swap
6:    recorre <- recorre.padre
7: endwhile
```

Complejidad: $\mathcal{O}(log(tam(c))$. **justificacion:** ya que, en el peor de los casos, tiene que swapear los valores desde una hoja hasta la raiz.

```
iSift-DOWN (in Nodo: puntero(nodo))
1: recorre <- Nodo
2: while TieneHijos(recorre) && HijosConPrioridad(recorre) do
3:     if (recorre.der.val < recorre.val) then
4:    swap <-recorre.val
5:    recorre <- recorre.der.val
6:    recorre.der.val <- swap
7:    recorre <- recorre.der
8: else
9:    swap <-recorre.val
```

```
10:    recorre <- recorre.izq.val
11:    recorre.izq.val <- swap
12:    recorre <- recorre.izq
13: endwhile
```

Complejidad: $\mathcal{O}(log(tam(c))$. **justificacion:** ya que, en el peor de los casos, tiene que swapear los valores desde la raiz hasta la hoja.

ihijosConPrioridad (**in** $Nodo$: puntero(nodo)) $\rightarrow$ res:Bool

```
1: if(Nodo.der.val < Nodo.val || Nodo.izq.val < Nodo.val) then
2:    res <- True
3: else
4:    res <- False
```

icompleto (**in** $tamao$: nat, **in** $altura$: nat) $\rightarrow$ res:Bool

```
1: if(tamaño == 2^altura -1) then
2:    res <- True
3: else
4:    res <- False
```

Complejidad: $\mathcal{O}(1)$.

isinHijos (**in** $Nodo$: puntero(nodo)) $\rightarrow$ res:Bool

```
1: if(Nodo.der == NULL && Nodo.izq == NULL) then
2:    res <- True
3: else
4:    res <- False
```

Complejidad: $\mathcal{O}(1)$.

iUnHijo (**in** $Nodo$: puntero(nodo)) $\rightarrow$ res:Bool

```
1: if(Nodo.der == NULL && Nodo.izq != NULL ⊻ Nodo.der != NULL && Nodo.izq == NULL ) then
2:    res <- True
3: else
4:    res <- False
```

Complejidad: $\mathcal{O}(1)$.

iDosHijos (**in** $Nodo$: puntero(nodo)) $\rightarrow$ res:Bool

```
1: if(¬ Unhijo(Nodo) && ¬ sinHijos(Nodo)) then
2:    res <- True
3: else
4:    res <- False
```

Complejidad: $\mathcal{O}(1)$.

inoHijoDerecho (**in** $Nodo$: puntero(nodo)) $\rightarrow$ res:Bool

```
1: if(Nodo.val != Nodo.padre.der.val) then
2:    res <- True
3: else
4:    res <- False
```

Complejidad: $\mathcal{O}(1)$.

4 Módulo DiccString(α)

Interfaz

parámetros formales:

géneros: α

función: $\bullet$ = $\bullet$(**in** d_1 : `diccString`, **in** d_2 : `diccString`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (d_1 = d_2$}
Complejidad: $\mathcal{O}(n * (m + k))$.Donde n y m son la cantidad de claves de d y d' respectivamente, y k la longitud del string mas largo de las claves.
Descripción: Función de igualdad de diccString.

función: COPIAR(**in** d : `DiccString(`α`)`) $\rightarrow$ res : `DiccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} d$}
Complejidad: $\mathcal{O}(n * |k|)$. Donde n es la cantidad de claves de d, y k la longitud del string mas largo de las claves.
Descripción: Funcion copia de d

se explica con: DICCIONARIO(STRING, α).

géneros: `diccString(`α`)`

Operaciones básicas del diccionario

VACIO() $\rightarrow$ res : `diccString(`α`)`
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}} vacio$}
Complejidad: $\Theta(1)$
Descripción: Genera un diccionario vacio

DEFINIR(**in/out** d : `diccString(`α`)`, **in** k : $string$, **in** v : α)
Pre $\equiv$ {d $=_{\text{obs}}$ d_0}
Post $\equiv$ {d $=_{\text{obs}}$ definir(d_0, k, v) }
Complejidad: $\Theta(|k|$
Descripción: Define la clave k con el significado v en el diccionario.

DEFINIDO?(**in** d : `diccString(`α`)`, **in** s : $string$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {d $=_{\text{obs}}$ def?(d, k, v) }
Complejidad: $\Theta(|k|)$
Descripción: Devuelve true si la clave k está definida, false de lo contrario.

SIGNIFICADO(**in** d : `diccString(`α`)`, **in** k : $string$) $\rightarrow$ res : α
Pre $\equiv$ {def?(d,k)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ significado(d,k))}
Complejidad: $\Theta(|k|)$
Descripción: Devuelve un puntero al significado de la clave k.
Aliasing: Da acceso a α y puede modificarlo.

BORRAR(**in/out** d : `diccString(`α`)`, **in** k : $string$)
Pre $\equiv$ {def?(d,k) $\wedge$ d = d_0}
Post $\equiv$ {d $=_{\text{obs}}$ borrar(d_0,k)}
Complejidad: $\Theta(|k|)$
Descripción: Borra el elemento.

CLAVESDICC(**in** d: `diccString(`α`)`) $\rightarrow res$: `itConj(string)`
Pre $\equiv$ {True}
Post $\equiv \{alias(res =_{\text{obs}} claves(d))\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve un iterador al conjunto de claves del diccionario.
Aliasing: El iterador sólo puede recorrer las claves.

Representación

Representación del diccionario

`diccString(`α`)` **se representa con** `estr`

donde `estr` es `tupla(`*raiz*: `puntero(nodo)`, *conjClaves*: `conj(string)`$)$

donde `nodo` es `tupla(`*chars*: `arreglo(puntero(nodo))`, *val*: `puntero(`α`)`, *nombre*: `itConj` $)$

Invariante en castellano

1. El arbol no tiene ciclos.
2. No hay dos nodos que apunten al mismo nodo.
3. El iterador nombre de un nodo apunta al string correspondiente a su valor val en conjClaves (si es que éste nodo está definido en el diccionario).
4. El tamaño de conjClaves es igual a la cantidad de valores distintos de NULL y viceversa.

Inviariante de representación:

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$

1. NoHayCiclos(e.raíz, $\emptyset$) $\wedge_{\text{L}}$
2. ($\forall\ n1, n2$: nodo)(($\nexists\ m$: nodo) (hayCamino($n1, m$) $\wedge$ hayCamino($n2, m$))) $\wedge$
3. ValoresEnConjunto(e.raíz, e.conjclaves)

NoHayCiclos : nodo $\times$ conj(nodo) $\longrightarrow$ bool

NoHayCiclos($e, padres$) $\equiv$ ($e \notin padres$) $\wedge_{\text{L}}$ NoHayCiclosEnLosChars(e, Ag($e, padres$), 0)

NoHayCiclosEnLosChars : nodo $\times$ conj(nodo) $\times$ nat $\longrightarrow$ bool

NoHayCiclosEnLosChars($e, padres, n$) $\equiv$ ((e.chars[n] $\neq_{\text{obs}}$ NULL) $\Rightarrow_{\text{L}}$ NoHayCiclos(e.chars[n], $padres$)) $\wedge_{\text{L}}$ ($n <$ 256 $\Rightarrow_{\text{L}}$ NoHayCiclosEnLosChars(e, $padres$, $n+1$))

ValoresEnConjunto : nodo $\times$ conj(string) $\longrightarrow$ bool

ValoresEnConjunto(n,cs) $\equiv$ (**if** n.val $\neq_{\text{obs}}$ NULL **then** siguiente(n.nombre) $\in$ cs **else** True **fi**) $\wedge$ ValoresEnConjuntoDeLosChars(n, cs, 0)

ValoresEnConjuntoDeLosChars : nodo $\times$ conj(string) $\times$ nat $\longrightarrow$ bool

ValoresEnConjuntoDeLosChars(n, cs, i) $\equiv$ (**if** n.chars[i] $\neq_{\text{obs}}$ NULL **then**
ValoresEnConjunto(n.chars[i], cs)
else
True
fi) $\wedge$
(**if** $i < 256$ **then**
ValoresEnConjuntoDeLosChars(n, cs, i + 1)
else
True
fi)

hayCamino : nodo $\times$ nodo $\longrightarrow$ bool

hayCamino(n, m) $\equiv$ **if** (n $\neq$ NULL) **then**
n == m $\vee$ ($\exists$ i: nat)((1$\leqslant$ i $\leqslant$ 256) $\Rightarrow_{\text{L}}$ hayCamino(n.chars[i], m))
else
False
fi

Funcion De Abstraccion

Abs : estrd e $\longrightarrow$ dicc(string, α) {Rep(e)}

Abs(e) $\equiv$ d: dicc(string, α) |

1. ($\forall c$: string)(def?(c, d) $\Leftrightarrow$
((ObtenerDeLaEstructura(c, e.raíz) $\neq_{\text{obs}}$ NULL) $\wedge_{\text{L}}$
obtener(c, d) $=_{\text{obs}}$ ObtenerDeLaEstructura(c, e.raíz)) $\wedge$
2. #(conjClaves(d)) $=_{\text{obs}}$ #(e.Claves)

ObtenerDeLaEstructura : string $\times$ nodo $\longrightarrow$ puntero(α)

ObtenerDeLaEstructura(c, n) $\equiv$ **if** vacía?(c) **then**
n.val
else
if n.chars[ord(prim(c))] $=_{\text{obs}}$ NULL **then**
NULL
else
ObtenerDeLaEstructura(fin(c), n.chars[ord(prim(c))])
fi
fi

Algoritmos

iVacio() $\rightarrow$ res : DiccString(α)

```
1: res.raiz ← nuevoNodo()                                   O(1)
2: res.conjclaves ← vacio()                                 O(1)
```

Complejidad: $\mathcal{O}(1)$.

iDefinir(**in/out** d : diccString(α), **in** k : string, **in** v : α)

```
1: nat i ← 0                                                O(1)
2: nat len ← |k|                                            O(|k|)
3: nodo tr ← d.raiz                                         O(1)
4: while i < len                                            O(|k|)..
5:     if tr.chars[ord(k[i])] = NULL                        O(1)
6:         tr.chars[ord(k[i])] ← nuevoNodo()                O(1)
7:     end if
```

```
 8:     tr ← tr.chars[ord(k[i])]                                   O(1)
 9:     i++                                                        O(1)
10: end while                                                      ....
11: tr.val ← v                                                     O(1)
12: tr.nombre ← AgregarRapido(d.conjclaves, k)                     O(Copy(k))
```

Complejidad: $|k| + \mathcal{O}(Copy(k)) = \mathcal{O}(|k|)$.

iDefinido?(**in** d : diccString(α), **in** k : string) → res : bool

```
1: nat i ← 0                                                       O(1)
2: nat len ← |k|                                                   O(|k|)
3: nodo tr ← d.raiz                                                O(1)
4: while i < len && tr.chars[ord(k[i])] != NULL                    O(|k|)..
5:     tr ← tr.chars[ord(k[i])]                                    O(1)
6:     i++                                                         O(1)
7: end while                                                       ....
8: res ← (i==len && tr.val!=NULL)                                  O(1)
```

Complejidad: $\mathcal{O}(|k|)$.

iSignificado(**in** d : diccString(α), **in** k : string) → res : α

```
1: nat i ← 0                                                       O(1)
2: nat len ← |k|                                                   O(|k|)
3: nodo tr ← d.raiz                                                O(1)
4: while i < len                                                   O(|k|)..
5:     tr ← tr.chars[ord(k[i])]                                    O(1)
6:     i++                                                         O(1)
7: end while                                                       ....
8: res ← tr.val                                                    O(1)
```

Complejidad: $\mathcal{O}(|k|)$.

iBorrar(**in/out** d : diccString(α), **in** k : string)

```
1: nat i ← 0                                                       O(1)
2: nat len ← |k|                                                   O(|k|)
3: nodo tr ← d.raiz                                                O(1)
4: while i < len                                                   O(|k|)..
5:     tr ← tr.chars[ord(k[i])]                                    O(1)
6:     i++                                                         O(1)
7: end while                                                       ....
8: eliminarSiguiente(tr.nombre)                                    O(1)
9: tr.val ← NULL                                                   O(1)
```

Complejidad: $\mathcal{O}(|k|)$.

iClavesDicc(**in** d : diccString(α)) → res : itConj(string)

```
1: res <- creatIt(d.conjClaves)                                    O(1)
```

Complejidad: $\mathcal{O}(1)$.

• = •(**in** d : diccString(α), **in** d' : diccString(α)) → res : bool

```
1: res <- True                                                     O(1)
2: if d.conjClaves = d'.conjClaves then                            O(n * m)
3:     it <- clavesDicc(d)                                         O(1)
```

```
4:     while haySiguiente(it) && res                                                    O(n)..
5:        res <- (significado(d, siguiente(it)) = significado(d', siguiente(it)) )     O(k)
6:        avanzar(it)                                                                  O(1)
7:     end while                                                                       ....
8: else
9:     res <- False                                                                    O(1)
10: end if
```

Complejidad: $\mathcal{O}(n * m) + \mathcal{O}(n * k) = \mathcal{O}(n * (m + k))$. Donde n y m son la cantidad de claves de d y d' respectivamente, y k la longitud del string mas largo de las claves.

```
iCopiar(in d : diccString(α))  → res : diccString(α)
1: res ← vacio()                                                                      O(1)
2: while haySiguiente(clavesDicc(d)) do                                               O(n * |k|))
3:     definir(res, siguiente(clavesDicc(d)), significado(d, siguiente(clavesDicc(d)))) O(|k|)
4: end while                                                                          ....
```

Complejidad: $\mathcal{O}(n * |k|)$. Donde n es la cantidad de claves de d, y k la longitud del string mas largo de las claves.

```
iNuevoNodo() → res : nodo
1: res.chars ← CrearArreglo(256)                                                      O(256)
2: res.val ← NULL                                                                     O(1)
3: res.nombre ← NULL                                                                  O(1)
4: for i=0 to 255 do                                                                  O(256)..
5:     res.chars[i] ← NULL                                                            O(1)
6: end for                                                                            ....
```

Complejidad: $\mathcal{O}(1)$.

Servicios Usados: Conj

5 Módulo DiccMatriz(α)

Interfaz

parámetros formales:

géneros: α

función: • = •(**in** d_1 : `diccMatriz`, **in** d_2 : `diccMatriz`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} (d_1 = d_2$}
Complejidad: $\mathcal{O}(n * m)$.Donde n es la cantidad maxima de columnas y m la cantidad maxima de filas
Descripción: Función de igualdad de diccString.

función: COPIAR(**in** d : `DiccMatriz(`α`)`) → res : `DiccMatriz(`α`)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} d$}
Complejidad: $\mathcal{O}(n * m)$.Donde n es la cantidad maxima de columnas y m la cantidad maxima de filas.
Descripción: Funcion copia de d

se explica con: DICCIONARIO(TUPLA(NAT, NAT), α).

géneros: `diccMatriz(`α`)`

Operaciones básicas del diccionario

VACIO() → res : `diccMatriz(`α`)`
Pre ≡ {$true$}
Post ≡ {$res =_{\text{obs}} vacio$}
Complejidad: $\Theta(1)$
Descripción: Genera un diccionario vacio

DEFINIR(**in/out** d : `diccMatriz(`α`)`, **in** k : `tupla(nat, nat)`, **in** v : α)
Pre ≡ {d =$_{\text{obs}}$ d_0}
Post ≡ {d =$_{\text{obs}}$ definir(d_0, k, v) }
Complejidad: $O(n * m)$
Descripción: Define la clave k con el significado v en el diccionario.

DEFINIDO?(**in** d : `diccMatriz(`α`)`, **in** k : `tupla(nat, nat)`) → res : `bool`
Pre ≡ {true}
Post ≡ {d =$_{\text{obs}}$ def?(d, k, v) }
Complejidad: $\Theta(1)$
Descripción: Devuelve true si la clave k está definida, false de lo contrario.

SIGNIFICADO(**in** d : `diccMatriz(`α`)`, **in** k : `tupla(nat, nat)`) → res : α
Pre ≡ {def?(d,k)}
Post ≡ {alias(res =$_{\text{obs}}$ significado(d,k))}
Complejidad: $\Theta(1)$
Descripción: Devuelve un puntero al significado de la clave k.
Aliasing: Da acceso a α y puede modificarlo.

BORRAR(**in/out** d : `diccMatriz(`α`)`, **in** k : `tupla(nat, nat)`)
Pre ≡ {def?(d,k) ∧ d = d_0}
Post ≡ {d =$_{\text{obs}}$ borrar(d_0,k)}
Complejidad: $\Theta(1)$
Descripción: Borra el elemento.

CLAVESDICC(**in** d: `diccMatriz(`α`)`) $\rightarrow$ res : `itConj(tupla(nat, nat))`
Pre $\equiv$ {True}
Post $\equiv$ {$alias(res =_{\text{obs}} claves(d))$}
Complejidad: $\Theta(1)$
Descripción: Devuelve un iterador al conjunto de claves del diccionario.
Aliasing: El iterador sólo puede recorrer las claves.

Representación

Representación del diccMatriz

`diccMatriz` **se representa con** `dm`

donde `dm` es `tupla`(*Matriz*: `vector(vector(`α`)` , *Claves*: `conj(tupla(nat,nat)` , *latitud*: `nat` , *longitud*: `nat`)

Invariante en castellano

la matriz es una matriz cuadrada con, cada lado igual al producto de la latitud con la longitud

Rep : diccMatriz $\longrightarrow$ bool

Rep(dm) $\equiv$ true $\iff$ Longitud dm.Matriz = latitud*longitud $\wedge_L$ ($\forall$ i:nat)((1 $\leq$ i $\leq$ dm.longitud))(longitud(dm.Matriz)[i] = long*lat

Abs : coor c $\longrightarrow$ Coordenada {Rep(c)}

Abs(c) $\equiv$ co:Coordenada |
c.latitud = latitud(co) $\wedge$
c.longitud = longitud(co)

Algoritmos

`Ivacio() → res:diccMatriz`

```
1: res.matriz <- vacia()                                              O(1)
```

Complejidad: $\mathcal{O}(1)$.

`Idefinir(`**`in/out`** mat `: diccMatriz,` **`in`** k `: tupla(nat,nat),` **`in`** v `:` α`)`

```
1: x <- π1(k)
2: y <- π2(k)
3: if (x < mat.lat && y < mat.long) then
4:     d.matriz[x][y] <- α                                            O(1)
5: else
6:     for(i = 0; i < x - mat.lat; i++) do                            O(x * y)
7:         for(j = 0; j < y - mat.long ; j++) do                      O(y)
8:             AgregarAtras(d[i],NULL)                                O(1)
9:         endfor
10:    endfor
11:    d[x][y] <- α                                                   O(1)
12: end if
```

Complejidad: $\mathcal{O}(\pi_1(k) * \pi_2(k))$.

`Idefinido(`**`in/out`** d `: diccMatriz,` **`in`** k `: tupla(nat,nat)) → res:bool`

```
1: res -> d.matriz[π1(k)][π2(k)] != NULL                              O(1)
```

π_1(k) **Complejidad:** $\mathcal{O}(1)$.

Isignificado(**in/out** d : diccMatriz, **in** k : tupla(nat,nat)) $\rightarrow$ res:$|alpha$

```
1: res -> d.matriz[π1(k)][π2(k)]
```
$\mathcal{O}(1)$

π_1(k) **Complejidad:** $\mathcal{O}(1)$.

Iborrar(**in/out** d : diccMatriz, **in** k : tupla(nat,nat))

```
1: d.matriz[π1(k)][π2(k)] <- NULL
```
$\mathcal{O}(1)$

π_1(k) **Complejidad:** $\mathcal{O}(1)$.

IclavesDicc(**in/out** d : diccMatriz) $\rightarrow$ res:itConj(tupla(nat,nat))

```
1: res <- d.claves.crearIt()
```

π_1(k) **Complejidad:** $\mathcal{O}(1)$.

6 Módulo Mapa

Interfaz

parámetros formales:

géneros: α

función: $\bullet = \bullet$(**in** d_1 : `diccMatriz`, **in** d_2 : `diccMatriz`) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} (d_1 = d_2\}$
Complejidad: $\mathcal{O}(n * m)$.Donde n es la cantidad maxima de columnas y m la cantidad maxima de filas
Descripción: Función de igualdad de diccString.

función: COPIAR(**in** d : `DiccMatriz(`α`)`) $\rightarrow res$: `DiccMatriz(`α`)`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} d\}$
Complejidad: $\mathcal{O}(n * m)$.Donde n es la cantidad maxima de columnas y m la cantidad maxima de filas.
Descripción: Funcion copia de d

se explica con: DICCMATRIZ(BOOL).

géneros: `diccMatriz(`α`)`

Operaciones básicas de mapa

CREARMAPA() $\rightarrow res$: `Mapa(Bool)`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} vacio\}$
Complejidad: $\Theta(1)$
Descripción: Genera un mapa vacio

HAYCAMINO(**in** c_1 : `coordenada` ,**in** c_2 : `coordenada`, **in** m : `Mapa`) $\rightarrow res$: `Bool`
Pre $\equiv$ {definido(c_1, m) $\wedge$ definido(c_1, m)}
Post $\equiv \{res =_{\text{obs}} HayCamino(c1, c2, m)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve verdadero si existe un camino entre la coordenada inicial y la final

COORDENADAS(**in** m : `Mapa`) $\rightarrow res$: `Bool`
Pre $\equiv$ {definido(c_1, m}
Post $\equiv \{res =_{\text{obs}} Adyacentes(c1, m)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de las coordenadas adyacentes a la coordenada dada

AGREGARCOORD(**in** d : `diccMatriz(bool)`, **in** k : `tupla(nat, nat))` $\rightarrow res$: α
Pre $\equiv$ {PosExistente(d,k)}
Post $\equiv$ {[}$\Theta(1)$] [Dice si el valor es verdadero o falso]

Representación

Representación del diccMatriz

`diccMatriz` **se representa con** `dm`

donde `dm` es `tupla`(*Matriz*: `vector(vector(`α`)` , *Claves*: `conj(tupla(nat,nat)` , *latitud*: `nat` , *longitud*: `nat`)

Invariante en castellano

la matriz es una matriz cuadrada con, cada lado igual al producto de la latitud con la longitud

Rep : diccMatriz $\longrightarrow$ bool

Rep(dm) $\equiv$ true $\iff$ Longitud dm.Matriz = latitud*longitud $\wedge_L$ ($\forall$ i:nat)((1 $\leq$ i $\leq$ dm.longitud))(longitud(dm.Matriz)[i] = long*lat

Abs : coor c $\longrightarrow$ Coordenada {Rep(c)}

Abs(c) $\equiv$ co:Coordenada |
c.latitud = latitud(co) $\wedge$
c.longitud = longitud(co)

Algoritmos

IAgregarCoord(**in** k : tupla(nat,nat), **in** m : mapa)

```
1: x <- π1(k)
2: y <- π2(k)
3: if (x < m.lat && y < m.long) then
4:     d <- posicion(x,y)
5:     m[d][d] <- True                                   O(1)
6: else
7:     for(i = 0; i < d; i++) do                         O(x * y)
8:         for(j = 0; d < y; j++) do                     O(y)
9:             AgregarAtras(m[i],NULL)                   O(1)
10:        endfor
11:    endfor
12:    m[d][d] <- true                                   O(1)
13: end if
```

Complejidad: $\mathcal{O}(x * y)$.

HayCamino(**in** c_1 : coordenada,**in** c_1 : coordenada, **in** m : mapa) $\rightarrow$ res:bool

```
1: d1 <- Posicion(c1)
2: d2 <- Posicion(c2)
3: if ExistePos(d1,d2) then
4:     res <- True
5: else
6:     res <- False
```

Complejidad: $\mathcal{O}(1)$.

Iconectadas(**in** k : tupla(nat,nat), **in** m : mapa)

```
1: x <- π1(k)
2: y <- π2(k)
3: PorVer <- Vacia
4: Visitadas <- Vacia
5: AgAtras (visitadas, <x,y>)
6: AgregarAdyacentes(Porver,Adyacentes(m,<x,y>))
7:     while (Tamaño(Porver) != 0 ) do
8:        A <- π1(Primero (porver))
9:        B <- π2(Primero (porver))
10:       AgregarAdyacentes(Primero(porver))
11:       EliminarVisitadas(Porver,visitadas)
12:       AgAtras(Visitadas,Primero(porver))
```

```
13:        EliminarPrimero(porver)
14:        endwhile
```

Complejidad: $\mathcal{O}(longitud(m) * longitud(m)$.

IEliminarVisitadas(**in/out** $porver$: ConjLineal(tupla<nat,nat>), **in** $Visitadas$: ConjLineal(tupla<nat,na

```
1: PV <- CrearIt(porver)
2: Vis <- CrearIt(visitadas)
3:     while (HaySiguiente(Vis) do
4:        while (HaySiguiente(PV) do
5:           if (Siguiente(PV) = Siguiente(Vis) then
6:              EliminarSiguiente(PV)
7: FI
8:        Avanzar(porver)
9:        endwhile
10:    avanzar(visitadas)
11:    endwhile
```

Complejidad: $\mathcal{O}(longitud(m) * longitud(m)$.

Iadyacentes(**in** c_1 : coordenada, **in** m : mapa) $\rightarrow$ res:conjuntoLineal

```
1: coord <- NuevaCoordenada(c1.longitud,c1.latitud + 1)
2:     agregar(adyacentes, coord)                                  O(1)
3: else                                                           O(1)
4: end if
5: if(Definido(c1.longitud,c1.latitud + 1,m)) then                O(1)
6: coord <- NuevaCoordenada(c1.longitud,c1.latitud + 1)
7:     agregar(adyacentes, coord)                                  O(1)
8: else                                                           O(1)
9: end if
10: if(Definido(c1.longitud +1,c1.latitud ,m)) then               O(1)
11: coord <- NuevaCoordenada(c1.longitud +1,c1.latitud)
12:    agregar(adyacentes, coord)                                  O(1)
13: if(Definido(c1.longitud -1,c1.latitud ,m)) then               O(1)
14: coord <- NuevaCoordenada((c1.longitud -1,c1.latitud)
15:    agregar(adyacentes, coord)                                  O(1)
16: if(Definido(c1.longitud ,c1.latitud -1 ,m)) then              O(1)
17: coord <- NuevaCoordenada((c1.longitud ,c1.latitud -1)
18:    agregarAtras(adyacentes, coord)                             O(1)
```

Complejidad: $\mathcal{O}(1)$.

7 Módulo Juego

Interfaz

se explica con: JUEGO.

géneros: vg.

Operaciones básicas de la vg

NUEVOJUEGO(**in** m : mapa) $\rightarrow res$: vg
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} crearJuego(m)$}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un nuevo juego.

AGREGARPOKÉMON(**in/out** v : vg, **in** c : coor, **in** p : pokemon)
Pre $\equiv$ {v = $v_0 \wedge$ puedoAgregarPokemon(c, v_0)}
Post $\equiv$ {v $=_{\text{obs}}$ agregarPokemon(p, c, v0)}
Complejidad: $\mathcal{O}(|P| + EC * log(EC))$
Descripción: Agrega el pokemon p al juego v.

AGREGARJUGADOR(**in/out** v : vg) $\rightarrow res$: jugador
Pre $\equiv$ {v = v_0}
Post $\equiv$ {v $=_{\text{obs}}$ agregarJugador(v_0) }
Complejidad: $\mathcal{O}(J)$
Descripción: Agrega un jugador al juego.

CONECTARSE(**in/out** v : vg, **in** j : jugador, **in** c : coor)
Pre $\equiv$ {v = $v_0 \wedge$ j $\in$ jugadores(v_0) $\wedge_L \neg$ estaConectado(j,v_0) $\wedge$ posExistente(c, Mapa(v_0))}
Post $\equiv$ {v $=_{\text{obs}}$ conectarse(j,c,v_0)}
Complejidad: $\mathcal{O}(log(EC))$
Descripción: Conecta al jugador.

DESCONECTARSE(**in/out** v : vg, **in** j : jugador)
Pre $\equiv$ {v = $v0 \wedge$ j $\in$ jugadores(v) $\wedge_L$ estaConectado(j, v_0)}
Post $\equiv$ {v $=_{\text{obs}}$ desconectarse(j,v_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Desconecta al jugador.

MOVERSE(**in/out** v : vg, **in** j : jugador, **in** c : coor)
Pre $\equiv$ {v = $v_0 \wedge$ j $\in$ jugadores(v0) $\wedge_L$ estaConectado(j,v) $\wedge$ posExistente(c, Mapa(j)) }
Post $\equiv$ {v $=_{\text{obs}}$ moverse(j, c, v_0)}
Complejidad: $\mathcal{O}((PS + PC)|P| + log(EC))$
Descripción: Mueve al jugador a la coordenada indicada en el parámetro.

MAPA(**in** v : vg) $\rightarrow res$: mapa
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ mapa(v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve una referencia no modificable del juego.

JUGADORES(**in** v : vg) $\rightarrow res$: itJugador
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ jugadores(v)}

Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador no modificable a los jugadores.

ESTÁCONECTADO(**in** v : vg, **in** j : jugador) $\rightarrow res$: bool
Pre $\equiv$ {j $\in$ jugadores(v)}
Post $\equiv$ {res $=_{\text{obs}}$ estaConectado(j,v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Dice si un jugador está conectado.

SANCIONES(**in** v : vg, **in** j : jugador) $\rightarrow res$: nat
Pre $\equiv$ {j $\in$ jugadores(v)}
Post $\equiv$ {res $=_{\text{obs}}$ sanciones(v, j)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve las sanciones que tiene el jugador j.

POSICIÓN(**in** v : vg, **in** j : jugador) $\rightarrow res$: coor
Pre $\equiv$ {j $\in$ jugadores(v) $\wedge_L$ estaConectado(j, v)}
Post $\equiv$ {res $=_{\text{obs}}$ posicion(v, j)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la posicion donde se encuentra el jugador j.

POKÉMONS(**in** v : vg, **in** j : jugador) $\rightarrow res$: ItMulticonjStr
Pre $\equiv$ {j $\in$ jugadores(v)}
Post $\equiv$ {res $=_{\text{obs}}$ pokemons(v, j)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador no modificable a los pokemon que tiene el jugador j.

EXPULSADOS(**in** v : vg) $\rightarrow res$: itExpulsado
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ expulsados(v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador a los expulsados.

POSCONPOKÉMONS(**in** v : vg) $\rightarrow res$: conj(coor)
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ posConPokemon(v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de las posiciones de los pokemons salvajes por referencia.

POKÉMONENPOS(**in** v : vg, **in** c : coor) $\rightarrow res$: pokemon
Pre $\equiv$ {c $\in$ posConPokemon(v)}
Post $\equiv$ {res $=_{\text{obs}}$ pokemonEnPos(v, c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el pokemon que se encuentra en la posición c.

CANTMOVIMIENTOSPARACAPTURA(**in** v : vg, **in** c : coor) $\rightarrow res$: nat
Pre $\equiv$ {c $\in$ posConPokemon(v)}
Post $\equiv$ {res $=_{\text{obs}}$ cantMovimientosParaCaptura(v, c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de moviemientos que faltan en la posición c para el pokemon sea capturado.

PUEDOAGREGARPOKEMON(**in** c : coor, **in** v : vg) $\rightarrow res$: bool
Pre $\equiv$ {c $\in$ coordenadas(mapa(v))}
Post $\equiv$ {res $=_{\text{obs}}$ puedoAgregarPokemon(c, v)}
Complejidad: $\mathcal{O}(1)$

Descripción: Dice si es posible agregar un pokemon en la coordenada c.

HAYPOKEMONCERCANO(**in** c : coor, **in** v : vg) $\rightarrow res$: bool
Pre $\equiv$ {c $\in$ coordenadas(mapa(v))}
Post $\equiv$ {res $=_{\text{obs}}$ hayPokemonCercano(c, v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Dice si hay algún pokemon a menos de 4 de distancia de la coordenada c.

POSPOKEMONCERCANO(**in** c : coor, **in** v : vg) $\rightarrow res$: coor
Pre $\equiv$ {c $\in$ coordenadas(mapa(v)) $\wedge_L$ hayPokemonCercano(c,j)}
Post $\equiv$ {res $=_{\text{obs}}$ posPokemonCercano(c, v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la coordenada donde se encuentra el pokemon cercano a c.

ENTRENADORESPOSIBLES(**in** c : coor, **in** js : conj(jugador), **in** v : vg) $\rightarrow res$: conj(jugador)
Pre $\equiv$ {c $\in$ coordenadas(mapa(v)) $\wedge_L$ hayPokemonCercano(c,v) $\wedge$ js $\subseteq$ jugadoresConectados(j) }
Post $\equiv$ {res $=_{\text{obs}}$ entrenadoresPosibles(c, js, v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de entrenadores que pueden atrapar al pokemon en la posición c.

INDICERAREZA(**in** p : pokemon, **in** v : vg) $\rightarrow res$: nat
Pre $\equiv$ {p $\in$ todosLosPokemon(v)}
Post $\equiv$ {res $=_{\text{obs}}$ indiceRareza(p, v)}
Complejidad: $\mathcal{O}(|P|)$
Descripción: Devuelve el índice de rareza del pokemon p.

CANTPOKEMONTOTALES(**in** v : vg) $\rightarrow res$: nat
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ cantPokemonTotales(v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad total de pokemon en el juego.

CANTMISMAESPECIE(**in** p : pokemon, **in** mp : multiconjunto(pokemon)) $\rightarrow res$: nat
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ cantMismaEspecie(p, mp)}
Complejidad: $\mathcal{O}(|P|)$
Descripción: Devuelve cuando pokemon del tipo p hay en el multiconjunto mp.

Operaciones del iterador de Jugador

El iterador diseñado permite recorrer los jugadores unidireccionalmente.

CREARIT(**in** v : vg) $\rightarrow res$: itJugador
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), jugadores(v)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un iterador unidireccional del conjunto de jugadores.

HAYSIGUIENTE(**in** it : itJugador) $\rightarrow res$: bool
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ haySiguiente?(it)}
Complejidad: $\mathcal{O}(n)$, n = cantidad de jugadores.
Descripción: devuelve **true** si y sólo si en el iterador todavía quedan elementos para avanzar.

SIGUIENTE(**in** it : itJugador) $\rightarrow res$: jugador
Pre $\equiv$ {haySiguiente?(it)}

Post $\equiv$ {alias($res =_{\text{obs}}$ siguiente(it))}
Complejidad: $\mathcal{O}(n)$, n = cantidad de jugadores.
Descripción: Devuelve el elemento siguiente del iterador.
Aliasing: res no es modificable.

AVANZAR(**in/out** it : `itJugador`)
Pre $\equiv$ {$it = it_0 \wedge$ haySiguiente?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $\mathcal{O}(n)$, n = cantidad de jugadores.
Descripción: Avanza a la posición siguiente del iterador.

Operaciones del iterador de Expulsado

El iterador diseñado permite recorrer los jugadores unidireccionalmente.

CREARIT(**in** v : `vg`) $\rightarrow res$: `itExpulsado`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), jugadores(v)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un iterador unidireccional del conjunto de expulsado.

HAYSIGUIENTE(**in** it : `itExpulsado`) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ haySiguiente?(it)}
Complejidad: $\mathcal{O}(n)$, n = cantidad de expulsados.
Descripción: devuelve **true** si y sólo si en el iterador todavía quedan elementos para avanzar.

SIGUIENTE(**in** it : `itExpulsado`) $\rightarrow res$: `jugador`
Pre $\equiv$ {haySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ siguiente(it))}
Complejidad: $\mathcal{O}(n)$, n = cantidad de expulsados.
Descripción: Devuelve el elemento siguiente del iterador.
Aliasing: res no es modificable.

AVANZAR(**in/out** it : `itJugador`)
Pre $\equiv$ {$it = it_0 \wedge$ haySiguiente?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $\mathcal{O}(n)$, n = cantidad de expulsados.
Descripción: Avanza a la posición siguiente del iterador.

Representación

Representación del juego

`juego` **se representa con** `estr`

donde `estr` es `tupla`(*mundo*: `mapa` , *jugadores*: `vector(puntero(infoJug))` , *posSalvajes*: `conjLineal(coor)` , *pokédex*: `diccString(totalEspecie:nat)` , *totalPokémons*: `nat` , *futurasCapturas*: `diccMatriz(puntero(infoPos))`)

donde `infoJug` es `tupla`(*sanciones*: `nat` , *conectado*: `bool` , *ubicación*: `coordenada` , *atrapados*: `multiconjStr` , *cazaActual*: `itCola(jugador)`)

donde `infoPos` es `tupla`(*turnos*: `nat` , *posiblesEntrenadores*: `colaDePrioridad(jugador)` , *bicho*: `pokémon`)

Invariante en castellano

1. Todos los jugadores cuyo puntero es distinto de NULL tienen 4 o menos sanciones.
2. Todos los pokémons atrapados del jugador pertenecen a las claves de pokedex
3. La ubicación de un jugador conectado pertenece a las coordenadas del mapa
4. Si el jugador está conectado y su ubicación cumple que tiene pokemon cercano y existe camino la misma y la posición del pokemon cercano entonces cazaActual apunta a una cola de prioridad
5. totalPokemons es igual a la suma de todos los significados de pokedex
6. Los turnos no pueden ser mayores a 10
7. Todos lo posibles entrenadores pertenecen a jugadores conectados, no estan expulsados y se encuentran a menos de 4 de DistEuclidea de la clave de futurasCapturas y hay camino entre dicha clave y su ubicación
8. El bicho pertenece a las claves de pokedex
9. Las posiciones de los pokémons salvajes son posiciones que pertenece al mundo

Rep : tbl $\longrightarrow$ bool

Rep(t) $\equiv$ true $\Longleftrightarrow$

1.

Abs : estr e $\longrightarrow$ vg {Rep(e)}

Abs(e) $\equiv$ jgo: vg |
e.mundo = mapa(jgo) $\wedge$
jugadores(jgo) = it2ConjJug(crearIt(e)) $\wedge$
expulsados(jgo) = it2ConjExp(crearIt(e)) $\wedge$
($\forall$ j: jugador) (j $\in$ jugadores(jgo) $\Rightarrow_L$ (estáConectado(j,jgo) = e.jugadores[j]->conectado $\wedge_L$
posición(j,jgo) = e.jugadores[J]->ubicación $\wedge$ sanciones(j,jgo) = e.jugadores[j]->sanciones $\wedge$
pokémons(j,jgo) = e.jugadores[j]->atrapados) $\wedge$
posConPokémons(jgo) = e.posSalvajes $\wedge$
($\forall$ c:coor) (c $\in$ posConPokémons(jgo) $\Rightarrow_L$ (pokémonEnPos(c,jgo) = obtener(c,e.futurasCapturas)->bicho $\wedge$ cantMovimientosParaCaptura(c,jgo) = obtener(c,e.futurasCapturas)-> turnos))

it2ConjJug : itJugador $\longrightarrow$ conj(jugador)

it2Conj(it) $\equiv$ **if** $\neg$ haySiguiente(it) **then** $\emptyset$ **else** Ag(siguiente(it), it2Conj(Avanzar(it))) **fi**

it2ConjExp : itExpulsado $\longrightarrow$ conj(jugador)

it2Conj(it) $\equiv$ **if** $\neg$ haySiguiente(it) **then** $\emptyset$ **else** Ag(siguiente(it), it2ConjExp(Avanzar(it))) **fi**

Algoritmos

```
InuevoJuego(in m : mapa) → res:estr
1: res.mundo <- m                                              O(1)
2: res.jugadores <- vacio()                                    O(1)
3: res.posSalvajes <- vacio()                                  O(1).
4: res.pokedex <- vacio()                                      O(1)
5: res.totalPokemon <- 0                                       O(1)
6: res.futurasCapturas <- vacio()                              O(1)
7: while m.clavesDicc.haySiguiente do                          O(#m.clavesDicc)
```

```
8:     futurasCapturas.definir(clavesDicc.Siguiente(), NULL)
9: end while
```

Complejidad: $\mathcal{O}(\#m.clavesDicc)$.

IagregarPokemon(**in/out** v : estr, **in** c : coor, **in** p : pokemon)

```
1: AgregarRapido(v.posSalvajes, c)                                                    O(1)
2: if(!Definido(v.pokedex, p)) then                                                   O(1)
3:     Definir(v.pokedex, p, 0)                                                       O(|p|)..
4: else                                                                               O(1)
5:     Definir(v.pokedex, p, significado(v.pokedex, p) + 1)                           O(|p|)
6: end if
7: info <- new infoPos                                                                O(1)
8: info.turnos <- 10                                                                  O(1)
9: info.pokemonEnPos <- p                                                             O(1)
10: nat cantAtrapados <- 0                                                            O(1)
11: bool online <- false                                                              O(1)
12: itJugador it <- v.jugadores.crearIt()                                             O(1)
13: while it.haySiguiente() do                                  O(|v.jugadores| * log(|v.jugadores|))
14:     online <- it.Siguiente().conectado                                            O(1)
15:     if online && v.puedeAtrapar(c, it.Siguiente.ubicacion) then                   O(1)
16:         cantAtrapados <- it.Siguiente().atrapados.cardinal()                      O(1)
17:         it.Siguiente().cazaActual <- info.posiblesEntrenadores.Encolar(it.Siguiente(), cantAtrapados)
    O(log(ec))
18:     end if
19:     it.Avanzar()                                                                  O(1)
20: end while
21: v.futurasCapturas.Definir(c, info)                                                O(1)
22: v.totalPokemon <- v.totalPokemon + 1                                              O(1)
```

Complejidad: $\mathcal{O}(|p| + |v.jugadores| * log(|v.jugadores|)$.

IpuedeAtrapar(**in** v : juego, **in** $c1$: coor, **in** $c2$: coor) $\rightarrow$ res:bool

```
1: res <- c1.distEuclidea(c2) <= 4 && v.mundo.hayCamino(c1, c2)                       O(1)
```

Complejidad: $\mathcal{O}(1)$.

IagregarJugador(**in/out** v : juego) $\rightarrow$ res:jugador

```
1: info <- new infoJug                                                                O(1)
2: info.sanciones <- 0                                                                O(1).
3: info.conectado <- false                                                            O(1)
4: info.ubicacion <- (0,0)                                                            O(1)
5: info.atrapados <- vacio()                                                          O(1)
6: info.cazaActual <- NULL                                                            O(1)
7: AgregarAtras(v.jugadores, info)                                                    O(1)
8: res <- v.jugadores.Ultimo()                                                        O(1)
```

Complejidad: $\mathcal{O}(1)$.

Iconectarse(**in/out** v : vg, **in** j : jugador, **in** c : coor)

```
1: v.jugadores[j].conectado <- true                                                   O(1)
2: v.jugadores[j].ubicacion <- c                                                      O(1)
3: if v.hayPokemonCercano(c) then
4:     infoPos pos <- v.futurasCapturas.significado(j.posPokemonCercano(c))           O(1)
5:     pos.turnos = 0                                                                 O(1)
6:     if v.mundo.hayCamino(v.posPokemonCercano(c), c)                                O(1)
7:         v.jugadores[j].cazaActual <- pos.posiblesEntrenadores.Encolar(j, v.jugadores[j].atrapados.card
```

$\mathcal{O}(log(|v.jugadores|)$

Complejidad: $\mathcal{O}(log(|v.jugadores|)$.

Idesconectarse(**in/out** v : vg, **in** j : jugador)

```
1: v.jugadores[j].conectado <- false                                                    O(1)
2: if v.jugadores[j].cazaActual != NULL then                                            O(1)
3:     v.jugadores[j].cazaActual.eliminarSiguiente()                                    O(log|v.jugadores|)
4: end if
5: if v.futurasCapturas.Significado(v.posPokemonCercano(v.jugadores[j].ubicacion)).posiblesEntrenadores.v
   then                                                                                 O(1)
6:     v.futurasCapturas.Significado(v.posPokemonCercano(v.jugadores[j].ubicacion)).turnos = 0
   O(1)
7: end if
```

Complejidad: $\mathcal{O}(log(|v.jugadores|))$.

Imoverse(**in/out** v : vg, **in** j : jugador, **in** c : coor)

```
1: jug <- v.jugadores[j]                                                                O(1)
2: hayPok <- false                                                                      O(1)
3: if v.hayPokemonCercano(jug.ubicacion) then                                           O(1)
4:     posPok <- v.posPokemonCercano(jug.ubicacion)                                     O(1)
5:     hayPok <- true                                                                   O(1)
6: end if
7: if v.debeSancionarse(j) && jug.sanciones == 4 then                                   O(1)
8:     v.expulsar(jug)                                                                  O(#jug.atrapados * |p|)
9: else if debeSancionarse(j) then                                                      O(1)
10:    jug.sanciones <- jug.sanciones + 1                                               O(1)
11: else
12:    jug.ubicacion <- c                                                               O(1)
13:    v.futurasCapturas.significado(posPok).turnos <- v.futurasCapturas.significado(posPok).turnos
   + 1                                                                                  O(1)
14:    if hayPok && c.distEuclidea(posPok) >= 4 then                                    O(1)
15:        jug.cazaActual.eliminarSiguiente()                                           O(log(|v.jugadores|))
16:    end if
17:    if c.distEuclidea(posPok) >= 4 && v.hayPokemonCercano(c) then                    O(1)
18:        jug.cazaActual <- v.futurasCapturas.Significado(v.posPokemonCercano(c)).futurosEntrenadores.En
   jug.atrapados.cardinal())                                                            O(log(|v.jugadores|)
19:        v.futurasCapturas.Significado(v.posPokemonCercano(c)).turnos = 0             O(1)
20:    end if
21:    if v.futurasCapturas.significado(posPok).turnos == 10 then                       O(1)
22:        capturador <- v.futurasCapturas.significado(posPok).entrenadoresPosibles.tope() O(1)
23:        v.jugadores[capturador].atrapados.agregarRapido(v.futurasCapturas.significado(posPok).bicho)
   O(1)
24:        v.futurasCapturas.eliminar(posPok)                                           O(1)
25:        it <- posSalvajes.crearIt()                                                  O(1)
26:        while(it.haySiguiente() && it.Siguiente() != posPok)                         O(#v.posSalvajes)
27:            it.avanzar()                                                             O(1)
28:        end while
29:    it.eliminarSiguiente()                                                           O(1)
30:    end if
31: end if
32: it2 <- posSalvajes.crearIt()
33: while (it2.haySiguiente())
34:    if it2.Siguiente() != c && it2.Siguiente() != posPok
35:        v.futurasCapturas.significado(it2.Siguiente()).turnos <- v.futurasCapturas.significado(it2.Sig
```

```
+ 1
```

Complejidad: $\mathcal{O}((pc + ps) * |p| + log(|v.jugadores|)$.

Imapa(**in** v : juego) $\rightarrow$ res:mapa

```
1: res <- v.mundo                                                    O(1)
```

Complejidad: $\mathcal{O}(1)$.

Ijugadores(**in** v : juego) $\rightarrow$ res:itJugador

```
1: res <- v.CrearIt()                                                O(1)
```

Complejidad: $\mathcal{O}(1)$.

IestaConectado(**in** v : juego, **in** j : jugador) $\rightarrow$ res:bool

```
1: res <- v.jugadores[j].conectado                                   O(1)
```

Complejidad: $\mathcal{O}(1)$.

Isanciones(**in** v : juego, **in** j : jugador) $\rightarrow$ res:nat

```
1: res <- v.jugadores[j].sanciones                                   O(1)
```

Complejidad: $\mathcal{O}(1)$.

Iposicion(**in** v : juego, **in** j : jugador) $\rightarrow$ res:coor

```
1: res <- v.jugadores[j].ubicación                                   O(1)
```

Complejidad: $\mathcal{O}(1)$.

Ipokémons(**in** v : juego, **in** j : jugador) $\rightarrow$ res:itMultiConjString

```
1: res <- v.jugadores[j].atrapados.CrearIt()                         O(1)
```

Complejidad: $\mathcal{O}(1)$.

Iexpulsados(**in** v : juego) $\rightarrow$ res:itExpulsado

```
1: res <- v.crearIt()                                                O(1)
```

Complejidad: $\mathcal{O}(1)$.

IposConPokémons(**in** v : juego) $\rightarrow$ res:conjLineal(coor)

```
1: res <- v.posSalvajes                                              O(1)
```

Complejidad: $\mathcal{O}(1)$.

IpokémonEnPos(**in** v : juego, **in** c : coor) $\rightarrow$ res:pokémon

```
1: res <- v.futurasCapturas.significado(c).bicho                     O(1)
```

Complejidad: $\mathcal{O}(1)$.

IcantMovimientosParaCaptura(**in** v : juego, **in** c : coor) $\rightarrow$ res:nat

```
1: res <- obtener(c, v.futurasCapturas)->turnos                      O(1)
```

Complejidad: $\mathcal{O}(1)$.

IpuedoAgregarPokémon(**in** v : juego, **in** c : coor) $\rightarrow$ res:bool

```
1: res <- posExistente(c,v.mundo) && ¬ hayPokemonEnTerritorio(c,v.posSalvajes)   O()
```

Complejidad: $\mathcal{O}()$.

IhayPokémonEnTerritorio(**in** c : coor, **in** cc : conj(coor)) $\rightarrow$ res:bool

```
1: itConj(coor) itPos <- crearIt(cc)
2: while ( haySiguiente(itPos) && distEuclidea(siguiente(itPos),c) <= 25 ) do   O(PS)
3:     avanzar(itPos)
4: end while
5: res <- ¬ haySiguiente(itPos)                                      O(1)
```

Complejidad: $\mathcal{O}(PS)$.

IobtenerPosicionesCercanas(**in** v : estr, **in** c : coor) $\rightarrow$ res:conj(coor)

```
1: menorLat <- latitud(c)
2: menorLong <- longitud(c)
3: while(menorLat > 1)
4:     menorLat --
5: end while
6: while(menorLong > 1)
7:     menorLong --
8: end while
9: for(i <- menorLong to menorLong + 5 ) do
10:     for(j <- menorLat to menorLat + 5 ) do
11:         if (i*i + j*j <= 4 && posExistente(nuevaCoor(i,j))) then
12:             AgregarRápido(res,nuevaCoor(i,j))
13:         end if
14:     end for
15: end for
```

Complejidad: $\mathcal{O}(1)$.

IhayPokémonCercano(**in** v : juego, **in** c : coor) $\rightarrow$ res:bool

```
1: itConj(coor) itC = crearIt(obtenerPosicionesCercanas(v,c))
2: while(haySiguiente(itC) && obtener(siguiente(itc),v.futurasCapturas) == NULL) do
3:     avanzar(itC)
4: end while
5: res <- haySiguiente(itC)
```

Complejidad: $\mathcal{O}(1)$.

IposPokémonCercano(**in** v : juego, **in** c : coor) $\rightarrow$ res:nat

```
1: itConj(coor) itC = crearIt(obtenerPosicionesCercanas(v,c))
2: while(haySiguiente(itC) && obtener(siguiente(itc),v.futurasCapturas) == NULL) do
3:     avanzar(itC)
4: end while
5: res <- siguiente(itC)
```

Complejidad: $\mathcal{O}(1)$.

IentrenadoresPosibles(**in** c : coor,**in** cj : conj(jugador), **in** v : juego) $\rightarrow$ res:conj(jugador)

```
1: itConj(jugador) itJ <- crearIt(cj)                                                    O(1)
2: while (haySiguiente(itJ)) do                                                          O(J)
3:     if ( hayPokémonCercano(v,posición(v,siguiente(itJ))) && posPokémonCercano(v, posición(v,
   siguiente(itJ))) == c && hayCamino(c,posicion(v,siguiente(itJ)),v.mundo) ) then       O(1)
4:         agregarRapido(res,siguiente(itJ))                                             O(1)
5:     end if
6:         avanzar(itJ)
7: end while
```

Complejidad: $\mathcal{O}(J)$.

IindiceRareza(**in** v : juego, **in** p : pokémon) $\rightarrow$ res:nat

```
1: res <- 100 - 100 * ( cantMismaEspecie(v,p) / cantPokémonsTotales(v) )                 O(|p|)
```

Complejidad: $\mathcal{O}(1)$.

IcantPokémonsTotales(**in** v : juego) $\rightarrow$ res:nat

```
1: res <- v.totalPokémons                                                                O(1)
```

Complejidad: $\mathcal{O}(1)$.

IcantMismaEspecie(**in** v : juego, **in** p : pokémon) $\rightarrow$ res:nat

```
1: if(definido(v.pokedex, p)) then                                  O(|p|)
2:     res <-significado(v.pokedex, p)                              O(|p|)
3: else
4:     res <- 0                                                     O(1)
5: end if
```

Complejidad: $\mathcal{O}(|p|)$.

IcreatIt(**in** v : juego) $\rightarrow$ res:itJugador

```
1: res.id <- 0                                                      O(1)
2: res.historial <- v.jugadores                                     O(1)
```

Complejidad: $\mathcal{O}(1)$.

IhaySiguiente(**in** it : itJugador) $\rightarrow$ res:bool

```
1: nat j <- it.id
2: while (j < longitud(it.historial) && it.historial[j] == NULL) do        O(n)
3:     j ++                                                                O(1)
4: end while
5: res <- j < longitud(it.historial)                                       O(1)
```

Complejidad: $\mathcal{O}(1)$.

Isiguiente(**in** it : itJugador) $\rightarrow$ res:Jugador

```
1: while( it.historial[it.id] == NULL) do                                  O(n)
2:     it.id ++                                                            O(1)
3: end while
4: res <- it.id
```

Complejidad: $\mathcal{O}(n)$ con n = cantidad jugadores que se agregaron al juego .

Iavanzar(**in/out** it : itJugador)

```
1: while( it.historial[it.id] == NULL) do                                  O(n)
2:     it.id ++                                                            O(1)
```

Complejidad: $\mathcal{O}(n)$ con n = cantidad jugadores que se agregaron al juego.

IcreatIt(**in** v : juego) $\rightarrow$ res:itExpulsado

```
1: res.id <- 0                                                             O(1)
2: res.historial <- v.jugadores                                            O(1)
```

Complejidad: $\mathcal{O}(1)$.

IhaySiguiente(**in** it : itExpulsado $\rightarrow$ res:bool

```
1: nat j <- it.id
2: while (j < longitud(it.historial) && it.historial[j] != NULL) do        O(n)
3:     j ++                                                                O(1)
4: end while
5: res <- j < longitud(it.historial)                                       O(1)
```

Complejidad: $\mathcal{O}(1)$.

Isiguiente(**in** it : itExpulsado) $\rightarrow$ res:nat

```
1: while( it.historial[it.id] != NULL) do                                  O(n)
2:     it.id ++                                                            O(1)
3: end while
4: res <- it.id
```

Complejidad: $\mathcal{O}(n)$ con n = cantidad jugadores que se agregaron al juego .

Iavanzar(**in/out** it : itJugador)

```
1: while( it.historial[it.id] != NULL) do                                  O(n)
2:     it.id ++                                                            O(1)
```

Complejidad: $\mathcal{O}(n)$ con n = cantidad jugadores que se agregaron al juego.

8 Observaciones

Pokémon es String.

Jugador es Nat.