Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2016

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

Trabajo Practico 2: Diseño

PokemonGOArgentina

noRep

Integrante	LU	Correo electrónico
Maidanik Ezequiel	935/12	maidaeze@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo Coordenada

El módulo Coordenada provee una representación del Tad coordenada en la que se puede acceder a la latitud y longitud. Y algunas funciones auxiliares.

Interfaz

se explica con: COORDENADA.

géneros: `coordenada`.

CREAR(**in** n_1, n_2 : `nat`) $\rightarrow$ *res* : `coordenada`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ crearCoor(n_1, n_2) }
Complejidad: $\Theta(1)$
Descripción: Genera una coordenada a partir de 2 naturales

LATITUD(**in** *c*: `coordenada`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ latitud(c)}
Complejidad: $\Theta(1)$
Descripción: Devuelve el primer componente de *c*

LONGITUD(**in** *c*: `coordenada`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ longitud(c)}
Complejidad: $\Theta(1)$
Descripción: Devuelve la segunda componente de *c*

DISTANCIA(**in** c_1, c_2 : `coordenada`)$\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ distEuclidea(c_1, c_2)}
Complejidad: $\Theta(1)$
Descripción: devuelve la distancia entre c1 y c2

Representación

`Coordenada` **se representa con** `co`

donde `co` es `tupla`(*Latitud*: `nat`, *Longitud*: `nat`)

Rep : co $\longrightarrow$ bool
Rep(c) $\equiv$ true

Abs : co c $\longrightarrow$ coor {Rep(c)}
Abs(c) $\equiv$ crearCoor(c.Latitud, c.Longitud)

Algoritmos

En esta sección se hace abuso de notación en los cálculos de álgebra de órdenes presentes en la justificaciones de los algoritmos. La operación de suma "+" denota secuencialización de operaciones con determinado orden de complejidad, y el símbolo de igualdad "=" denota la pertenencia al orden de complejidad resultante.

Algoritmos del módulo

---

iCrear(**in** *Latitud*, *Longitud*: `nat`) $\rightarrow$ *res* : co
1: *res* $\leftarrow$ $\langle res.Latitud, res.Longitud \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iLatitud(**in** c : co) $\rightarrow$ res : nat

1: $res \leftarrow c.Latitud$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iLongitud(**in** c : co) $\rightarrow$ res : nat

1: $res \leftarrow c.Longitud$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iDistancia(**in** c_1,c_2 : co)$\rightarrow$ res : nat

1: $longAux \leftarrow 0$ ▷ $\Theta(1)$
2: $latiAux \leftarrow 0$ ▷ $\Theta(1)$
3: **if** $c_1.Latitud < c_2.Latitud$ **then** ▷ $\Theta(1)$
4: $\quad latiAux \leftarrow (c_2.Latitud - c_1.Latitud)$
5: **else**
6: $\quad latiAux \leftarrow (c_1.Latitud - c_2.Latitud)$
7: **end if**
8: **if** $c_1.Longitud < c_2.Longitud$ **then** ▷ $\Theta(1)$
9: $\quad longAux \leftarrow (c_2.Longitud - c_1.Longitud)$
10: **else**
11: $\quad longAux \leftarrow (c_1.Longitud - c_2.Longitud)$
12: **end if**
13: $longAux \leftarrow (longAux * longAux)$ ▷ $\Theta(1)$
14: $latiAux \leftarrow (latiAux * latiAux)$ ▷ $\Theta(1)$
15: $dist \leftarrow (longAux + latiAux)$ ▷ $\Theta(1)$
16: $res \leftarrow dist$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

2. Módulo DiccionarioRapido(Nat, α)

El módulo DiccionarioRapido provee un diccionario con tiempo constante de obtener y de redefinir, no provee la operacion de borrar.

Para describir la complejidad de las operaciones, vamos a llamar *copy*(a) al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una función de α en $\mathbb{N}$), y vamos a utilizar

$$f(n,l) = \begin{cases} n & \text{si } n = l \wedge l = 2^k \text{ para algun k} \\ 1 & \text{en caso contrario} \end{cases}$$

para describir el costo de definir una clave.

Especificacion del dicc rapido

TAD DiccionarioR(nat, significado)

igualdad observacional

$$(\forall d, d' : \text{dicc}(\kappa, \sigma)) \left(d =_{\text{obs}} d' \iff \begin{pmatrix} (\forall c : \kappa)(\text{def?}(c,d) =_{\text{obs}} \text{def?}(c,d') \wedge_{\text{L}} \\ (\text{def?}(c,d) \Rightarrow_{\text{L}} \text{obtener}(c,d) =_{\text{obs}} \text{obtener}(c,d'))) \end{pmatrix} \right)$$

parámetros formales
géneros nat, significado

géneros diccR(significado)

exporta diccR(significado), generadores, observadores, claves

usa Bool, Nat, Conjunto(nat)

observadores básicos

def?	: clave × diccR(significado)	$\longrightarrow$ bool	
obtener	: nat c × diccR(significado) d	$\longrightarrow$ significado	{def?(c, d)}

generadores

vacío	:	$\longrightarrow$ diccR(significado)	
definir	: nat n × significado × diccR(significado) d	$\longrightarrow$ diccR(significado)	{n≤#(claves(d))}

otras operaciones

claves : diccR(significado) $\longrightarrow$ conj(clave)

axiomas $\forall d$: diccR(significado), $\forall c, k$: nat, $\forall s$: significado

def?(c,vacío)	$\equiv$	false
def?(c, definir(k, s, d))	$\equiv$	$c = k \vee$ def?(c, d)
obtener(c, definir(k, s, d))	$\equiv$	**if** $c = k$ **then** s **else** obtener(c, d) **fi**
claves(vacío)	$\equiv$	$\emptyset$
claves(definir(c,s,d))	$\equiv$	Ag(c, claves(d))

Fin TAD

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: $DiccionarioR(Nat, \alpha)$.

géneros: `diccR(`α`)`.

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

DEFINIR(**in** c: Nat,**in** s: α, **in/out** d: diccR(α))
Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}} definir(c, s, d_0)\}$
Complejidad: $\Theta(f(n, \#(claves(d))) * copy(a))$
Descripción: Agrega al diccionario el significado dada la clave

ESTADEFINIDO(**in** n: Nat, **in** d: diccR(α)) $\rightarrow$ res : bool
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} def?(n, d)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve **true** si y sólo si d esta definido para la clave n

OBTENER(**in** n: Nat, **in** d: diccR(α)) $\rightarrow$ res : α
Pre $\equiv$ {def?(n, d)}
Post $\equiv$ {alias($res = obtener(c, d)$)}
Complejidad: $\Theta(1)$
Descripción: devuelve por referencia el significado de la clave c

CLAVES(**in** d: diccR(α)) $\rightarrow$ res : itConj(nat)
Pre $\equiv$ {true}
Post $\equiv$ {esPermutacion?($SecuSuby(res)$, $claves(d)$)$\wedge vacia?(Anteriores(res))$}
Complejidad:$\Theta(1)$
Descripción: Devuelve un iterador a el conjunto de las claves dado el diccionario, el conjunto no es modificable.

Representación

diccR(α) **se representa con** di

donde di es tupla(*vec*: vector(α), *claves*: conj(α))

Rep : di $\longrightarrow$ bool
Rep(d) $\equiv$ true $\iff$ longitud(d.vec) = #(d.claves) $\wedge$ ($\forall$ n : nat)n $<$ longitud(d.vec) $\Rightarrow_{\text{L}}$ n $\in$ d.claves
Abs : di d $\longrightarrow$ diccR(α) {Rep(d)}
Abs(d) $\equiv$ **if** $longitud(d.vec) = 0$ **then**
$vacio$
else
$definir(longitud(d.vec) - 1, ultimo(d.vec), Abs(\langle comienzo(d.vec), d.claves \setminus \{\#(d.claves) - 1\}\rangle))$
fi

Algoritmos

---

iVacio() $\rightarrow$ res : di
1: $res \leftarrow \langle vacia(), vacio()\rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iDefinir(**in** n: nat,**in** a: α, **in/out** d: di)
1: **if** $n \geq longitud(d.vec)$ **then** ▷ $\Theta(f(n, longitud(d.vector)) + copy(a))$
2: $agregarRapido(d.claves, n)$
3: $agregarAtras(d.vec, a)$
4: **else**
5: $d.vec[n] \leftarrow a$
6: **end if**

Complejidad: $\Theta(f(n, \#claves(d)) + copy(a))$

---

iEstaDefinido(**in** n : `nat`, **in** d : `di`) $\rightarrow$ res : $bool$

1: **if** $n < longitud(d.vec)$ **then** ▷ $\Theta(1)$
2: $\quad res \leftarrow true$
3: **else**
4: $\quad res \leftarrow false$
5: **end if**

Complejidad: $\Theta(1)$

---

iObtener(**in** n : `nat`, **in** d : `di`) $\rightarrow$ res : α

1: $res \leftarrow (d.Vector)[n]$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iClaves(**in** d : `di`) $\rightarrow$ res : $itConj(\alpha)$

1: $res \leftarrow CrearIt(d.claves)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

3. Módulo Cola De Prioridad(α)

El módulo Cola de Prioridad provee una cola en la que se puede acceder al proximo de la misma y borrar cualquiera de sus elementos (si se tiene un iterador a el).

Para describir la complejidad de las operaciones, vamos a llamar *copy*(a) al costo de copiar el elemento $a \in \alpha$.

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: COLA DE PRIORIDAD EXTENDIDA(α), ITERADOR UNIDIRECCIONAL MODIFICABLE(α).

géneros: `colaPrior(`α`)`, `itCola(`α`)`.

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

ENCOLAR(**in/out** c: `colaPrior(`α`)`, **in** a: α) $\rightarrow$ res : `itCola(`α`)`
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ encolar(c, a) $\wedge$ alias(secuenciaACola(SecuSuby(res)) = c) $\wedge$ Actual(res) = a}
Complejidad: $\Theta(copy(a) + \log_2(tam(c)))$
Descripción: encola a en c. Retorna un iterador al elemento recien agregado.
Aliasing: El elemento a se encola por copia. El iterador se invalida si y sólo si se elimina el elemento actual del iterador. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique c.

ESVACIA?(**in** c: `colaPrior(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia?(c)}
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si la cola es vacía.

PROXIMO(**in** c: `colaPrior(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {$\neg$vacía?(c)}
Post $\equiv$ {alias($res =_{\text{obs}}$ proximo(c))}
Complejidad: $\Theta(1)$
Descripción: devuelve el proximo de la cola.

Operaciones del iterador

El iterador que presentamos permite unicamente eliminar, saber si existe un elemento actual y en caso de existir leerlo.

CREARIT(**in** c: `colaPriori(`α`)`) $\rightarrow$ res : `itCola(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {actual(res)=$_{\text{obs}}$proximo(c)}
Complejidad: $\Theta(1)$
Descripción: crea un iterador que devuelve el primer elemento de la cola

HAYMÁS?(**in** it: `colaPriori(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{\text{obs}}$ HayMas?(it))}
Complejidad: $\Theta(1)$

Descripción: devuelve el elemento actual a la posición del iterador por referencia.
Aliasing: *res* es modificable si y sólo si *it* es modificable.

ACTUAL(**in** *it* : `colaPriori(`α`)`) $\rightarrow$ *res* : α
Pre $\equiv$ {HayMas?(*it*)}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ Actual(*it*))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento actual a la posición del iterador por referencia.
Aliasing: *res* es modificable si y sólo si *it* es modificable.

ELIMINAR(**in/out** *it* : `colaPriori(`α`)`)
Pre $\equiv$ {$it = it_0 \wedge HayMas?$(it)}
Post$\equiv$ {*it* $=_{\text{obs}}$ Eliminar(it_0)}
Complejidad: $\Theta(log_2(long(SecuSuby(it))))$
Descripción: elimina de la cola de prioridad el valor que se encuentra en la posición actual del iterador y queda apuntando a nada

TAD Cola de Prioridad Extendida(α)

extiende COLA DE PRIORIDAD(α)

otras operaciones (no exportadas)
secuenciaACola : secu(α) $\longrightarrow$ colaPrior(α)

axiomas
secuenciaACola(s) $\equiv$ **if** $vacia?(s)$ **then** $vacia$ **else** $encolar(prim(s), secuenciaACola(fin(s)))$ **fi**

Fin TAD

Representación

El objetivo de este módulo es implementar una cola lo más eficientemente posible, que permita el agregado y el borrado en orden logaritmico. Para garantizar estas complejidades, se representa con un arbol(si usaramos un vector tendiramos O(EC) para el agregado).

`colaPriori` **se representa con** `estr`

donde `estr` es `tupla(`*proximo*: `puntero(nodo)`, *ultimo*: `puntero(nodo)`)
donde `nodo` es `tupla(`*padre*: `puntero(nodo)`, *hijoIzquierdo*: `puntero(nodo)` , *hijoDerecho*: `puntero(nodo)` , *raiz*: `jugadorCercano`)

Rep:

1. Proximo es NULL si y solo si ultimo es NULL.
2. Si proximo no es NULL, el padre del proximo es NULL.
3. Para todos los nodos del arbol, que no son el proximo, el nodo es hijo izquierdo o hijo derecho de su padre y su raiz es mayor o igual que la de su padre.
4. Si el arbol formado por la cola no es completo, los nodos faltantes estan en el ultimo piso. Los nodos del ultimo piso del arbol son contiguos y estan alineados a la izquierda, osea, el ultimo hijo izquierdo del arbol esta presente en el ultimo piso(resultado de aplicar sucesivamente hijo izquierdo desde el nodo proximo) y entre 2 nodos del ultimo piso no pueden faltar nodos.
5. Si ultimo no es NULL, es el nodo del ultimo piso del arbol que esta mas a la derecha.

Abs : estr e $\longrightarrow$ colaPriori(α) {Rep(e)}
Abs(e) $\equiv$ **if** $e.proximo = NULL$ **then** vacía **else** $encolarTodos(vacia, hacerSecuencia(e.proximo))$ **fi**

encolarTodos : colaPriori(α) $\times$ secu(puntero(nodo)) $\longrightarrow$ colaPriori(α)
encolarTodos(c, sp) $\equiv$ **if** vacia?(sp) **then** c **else** $encolarTodos(encolar(prim(sp) \rightarrow raiz), fin(sp))$ **fi**

hacerSecuencia : puntero(nodo) $\longrightarrow$ secu(puntero(nodo))

hacerSecuencia(p) $\equiv$ **if** NULL = p **then**
<>
else
$hacerSecuencia(p \rightarrow hijoIzquierdo)$ & $(p \rightarrow raiz \bullet <>)$& $hacerSecuencia(p \rightarrow hijoDerecho)$
fi

Representación del iterador:

El iterador es simplemente un puntero al nodo actual, que fue recien agregado. Como el objetivo del iterador es unicamente eliminar elementos, no se puede avanzar con el.

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

donde `iter` es `tupla(`*actual*: `puntero(nodo)`, *cola*: `puntero(colaPrior)`$)$

Rep : iter $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ Rep($*(it$.cola) $\wedge_{\text{L}}$ (actual forma parte de la cola de prioridad)

Abs : iter it $\longrightarrow$ itBi(α) {Rep(it)}
Abs(it) $=_{\text{obs}}$ b: itBi(α) | actual(b) = $it.actual$)

Algoritmos

---

iVacia(**in** j : `pg`) $\rightarrow$ res : $colaPriori$
$res \leftarrow \langle NULL, DiccJugadores(j), NULL\rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(copy(a))$

---

iEncolar(**in/out** c: `colaPriori`, **in** j : `jugadorCercano`) $\rightarrow$ res : $itCola$
if $(c.ultimo = NULL)$ **then** ▷ $\Theta(1)$
$c.ultimo \leftarrow \&\langle NULL, NULL, NULL, j\rangle$
else ▷ $\Theta(1)$
$nuevo \leftarrow \&\langle e.ultimo, NULL, NULL, j\rangle$
if $(c.ultimo \rightarrow hijoIzquierdo = NULL)$ **then** ▷ $\Theta(1)$
$c.ultimo \rightarrow hijoIzquierdo \leftarrow nuevo$
else ▷ $\Theta(1)$
$c.ultimo \rightarrow hijoDerecho \leftarrow nuevo$ ▷ $\Theta(1)$
while $c.ultimo \neq NULL \wedge c.ultimo \neq c.ultimo \rightarrow hijoDerecho$ **do** ▷ $\Theta(i)$
$Avanzar(it)$ ▷ $\Theta(1)$
$indice \leftarrow indice + 1$ ▷ $\Theta(1)$
end while
end if
end if
$res \leftarrow CrearIt(c)$ ▷ $\Theta(1)$

Complejidad: $\Theta(copy(a))$
Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$

---

iSwap(**in/out** n_1 : `puntero(nodo)`, **in/out**n_2 : `puntero(nodo)`) ▷ Esta es una operación privada que intercambia 2

$temp \leftarrow \&\langle n_1 \rightarrow padre, n_1 \rightarrow hijoIzquierdo, n_1 \rightarrow hijoDerecho, n_1 \rightarrow raiz\rangle$ ▷ nodos de lugar en el arbol // $\Theta(1)$

$n_1 \rightarrow padre \leftarrow n_2 \rightarrow padre$ ▷ $\Theta(1)$

$n_1 \rightarrow hijoDerecho \leftarrow n_2 \rightarrow hijoDerecho$ ▷ $\Theta(1)$

$n_1 \rightarrow hijoIzquierdo \leftarrow n_2 \rightarrow hijoIzquierdo$ ▷ $\Theta(1)$

$n_2 \rightarrow padre \leftarrow temp \rightarrow padre$ ▷ $\Theta(1)$

$n_2 \rightarrow hijoDerecho \leftarrow temp \rightarrow hijoDerecho$ ▷ $\Theta(1)$

$n_2 \rightarrow hijoIzquierdo \leftarrow temp \rightarrow hijoIzquierdo$ ▷ $\Theta(1)$

if $n_1 \rightarrow padre = n_1$ **then** ▷ $\Theta(1)$

  $n_1 \rightarrow padre \leftarrow n_2$

  if $n_2 \rightarrow hijoDerecho = n_2$ **then** ▷ $\Theta(1)$

    $n_2 \rightarrow hijoDerecho \leftarrow n_1$

  else ▷ $\Theta(1)$

    $n_2 \rightarrow hijoIzquierdo \leftarrow n_1$

  end if

end if

Complejidad: $\Theta(1)$

Justificación: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

---

iUpHeap(**in/out** n_1 : `puntero(nodo)`)

while $n \rightarrow padre \neq NULL \wedge n \rightarrow raiz > n \rightarrow padre \rightarrow raiz$ **do** ▷

  $swap(padre, n)$ ▷ $\Theta(1)$

  $n \leftarrow n \rightarrow padre$

end while

Complejidad: $\Theta(\log_2(tam(c)))$

Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y el ciclo se repite $\log_2(tam(c))veces$

---

iUpHeap(**in/out** n_1 : `puntero(nodo)`)

if $n \rightarrow padre \neq NULL$ **then** ▷ $\Theta(1)$

  $npadre \leftarrow n \rightarrow padre$ ▷ $\Theta(1)$

  if $n \rightarrow raiz > npadre \rightarrow raiz$ **then** ▷ $\Theta(1)$

    $swap(padre, n)$ ▷ $\Theta(1)$

    $Upheap(n)$ ▷ $\Theta(1)$

  end if

end if

Complejidad: $\Theta(copy(a))$

---

4. Módulo JugadorCercano

El módulo JugadorCercano provee una tupla que tiene un operador menor.

TAD TuplaOrdenada(Nat,Nat)

extiende TUPLA(NAT,NAT)

otras operaciones

$\bullet < \bullet$: tupla(nat,nat) $t \times$ tupla(nat,nat) $p \longrightarrow$ bool

axiomas

$c_1 < c_2 \equiv$ **if** $\Pi_1(c_1) < \Pi_1(c_2) \vee (\Pi_1(c_1) = \Pi_1(c_2) \wedge \Pi_2(c_1) < \Pi_2(c_2))$ **then** $true$ **else** $false$ **fi**

Fin TAD

Interfaz

se explica con: TUPLAORDENADA(NAT,NAT).

géneros: `jugadorCercano`.

CREARTUPLA(**in** $CantCapturados, JugadorId$: `Nat`) $\rightarrow res$: `jugadorCercano`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \langle CantCapturados, JugadorId \rangle\}$
Complejidad: $\Theta(1)$
Descripción: Genera una Tupla a partir de 2 Nat

JUGADOR(**in** j : `jugadorCercano`) $\rightarrow res$: `nat`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \Pi_2(j)\}$
Complejidad:$\Theta(1)$
Descripción: Devuelve la primera componente de la tupla, en este caso la cantidad de pokemon que capturó

CANTCAPTURADOS(**in** j : `jugadorCercano`) $\rightarrow res$: `nat`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \Pi_1(j)\}$
Complejidad:$\Theta(1)$
Descripción: Devuelve la segunda componente de la tupla, en este caso el jugador

$\bullet < \bullet$(**in** j_1, j_2 : `jugadorCercano`) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} j_1 < j_2\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve true si y solo si la primera componente de t es menor que la primera componente de p y si son iguales si la segunda componente de t es menor que la segunda componente de p

Representación

`JugadorCercano` **se representa con** `jc`

donde `jc` es `tupla`(*CantCapturados*: `nat`, *JugadorId*: `nat`)

Rep : jc $\longrightarrow$ bool
Rep(j) $\equiv$ true

Abs : jc j $\longrightarrow$ TuplaOrdenada(nat,nat) {Rep(j)}
Abs(j) $\equiv \langle jc.CantCapturados, jc.JugadorId \rangle$

Algoritmos

Algoritmos del módulo

iCrearTupla(**in** $CantCapturados, JugadorId$: `nat`) $\rightarrow$ res : jc

1: $res \leftarrow \langle CantCapturados, JugadorId \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iJugador(**in** j : `jc`) $\rightarrow$ res : nat

1: $res \leftarrow j.JugadorId$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iCantCapturados(**in** j : `jc`) $\rightarrow$ res : nat

1: $res \leftarrow j.CantCapturados$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

$\bullet < \bullet$(**in** j_1, j_2 : `JugadorCercano`)$\rightarrow$ res : bool

if $((j_1.CantCapturados < j_2.CantCapturados) \vee$
$((j_1.CantCapturados == j_2.canCapturados) \wedge (j_1.JugadorId < j_2.JugadorId)))$ **then** ▷ $\Theta(1)$
    $res \leftarrow true$
else
    $res \leftarrow false$
end if

Complejidad: $\Theta(1)$

5. Módulo Mapa

El módulo Mapa es usado para la construccion de una instancia del modulo pokemon go.

Interfaz

se explica con: MAPA.

géneros: `map`.

CREARMAPA() $\rightarrow$ *res* : `map`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ crearMapa()}
Complejidad: $\Theta(1)$
Descripción: Crea un mapa vacio

AGREGARCOOR(**in** *c*: `coordenada`, **in/out** *m*: `map`)
Pre $\equiv$ {m $=_{\text{obs}}$ m_0}
Post $\equiv$ {*m* $=_{\text{obs}}$ agregarCoor(c,m_0)}
Complejidad: $\Theta(long(coordenadas(m)))$
Descripción: agrega la coordenada c al mapa

COORDENADAS(**in** *m*: `map`) $\rightarrow$ *res* : `conj(coor)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ itLista(α)}
Complejidad: $\Theta(1)$
Descripción: devuelve el conjunto de coordenadas del mapa m

POSEXISTENTE(**in** *c*: `coor`, **in** *m*: `map`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ $posExistente(c, m)$}
Complejidad: $\Theta(long(coordenadas(m)))$
Descripción: Devuelve true si la coordenada c pertenece al mapa m

HAYCAMINO(**in** $c1, c2$: `coor`, **in** *m*: `map`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ $\{c1 \in coordenadas(m) \land c2 \in coordenadas(m)\}$
Post $\equiv$ {*res* $=_{\text{obs}}$ $hayCamino(c1, c2, m)$}
Complejidad: $\Theta(long(coordenadas(m)))$
Descripción: Devuelve true si existe un camino entre las coordenadas $c1$ y $c2$

Representación

`map` **se representa con** `mp`

donde `mp` es `tupla`(*coordenadas*: `conj(coor)`, *conexiones*: `lista(lista(coor))`)

Rep : mp $\longrightarrow$ bool
Rep(m) $\equiv$ true $\iff$ $esPermutacion?(concatenar(m.conexiones), m.coordenadas) \land_{\text{L}}$
$(\forall c_1 : coor)(\forall c_2 : coor)(c_1 \in m.coordenadas \land c_2 \in m.coordenadas) \land_{\text{L}}$
$existeCamino(c_1, c_2, m.coordenadas) \Rightarrow (\exists l : secu(coor))esta?(l, m.conexiones) \land esta?(c_1, l) \land esta?(c_2, l)$

concatenar : secu(secu(α)) $\longrightarrow$ secu(α)
concatenar(s) $\equiv$ **if** $vacia?(s)$ **then** <> **else** $prim(s)$ & $concatenar(fin(s))$ **fi**

existeCamino : coor $\times$ coor $\times$ conj(coor) $\longrightarrow$ bool

existeCamino(c_1, c_2, cs) $\equiv$ **if** $c_1 = c_2$ **then**
$true$
else
if $\emptyset?(cs)$ **then**
$false$
else
$existeCaminoPorArriba(c_1, c_2, cs) \vee$
$existeCaminoPorAbajo(c_1, c_2, cs) \vee$
$existeCaminoPorDerecha(c_1, c_2, cs) \vee$
$existeCaminoPorIzquierda(c_1, c_2, cs)$
fi
fi

existeCaminoPorArriba : coor $\times$ coor $\times$ conj(coor) $\longrightarrow$ bool
existeCaminoPorArriba(c_1, c_2, cs) $\equiv$ coordenadaArriba(c_1) $\in$ cs $\wedge_{\text{L}}$ existeCamino(coordenadaArriba(c_1), c_2, cs - {coordenadaArriba(c_1)})

existeCaminoPorAbajo : coor $\times$ coor $\times$ conj(coor) $\longrightarrow$ bool
existeCaminoPorArriba(c_1, c_2, cs) $\equiv$ latitud(c_1) $> 0 \wedge_{\text{L}}$coordenadaAbajo(c_1) $\in$ cs $\wedge_{\text{L}}$
existeCamino(coordenadaAbajo(c_1), c_2, cs - {coordenadaAbajo(c_1)})

existeCaminoPorDerecha : coor $\times$ coor $\times$ conj(coor) $\longrightarrow$ bool
existeCaminoPorDerecha(c_1, c_2, cs) $\equiv$ coordenadaDerecha(c_1) $\in$ cs $\wedge_{\text{L}}$ existeCamino(coordenadaDerecha(c_1), c_2, cs - {coordenadaDerecha(c_1)})

existeCaminoPorIzquierda : coor $\times$ coor $\times$ conj(coor) $\longrightarrow$ bool
existeCaminoPorIzquierda(c_1, c_2, cs) $\equiv$ longitud(c_1) $> 0 \wedge_{\text{L}}$coordenadaIzquierda(c_1) $\in$ cs $\wedge_{\text{L}}$
existeCamino(coordenadaIzquierda(c_1), c_2, cs - {coordenadaIzquierda(c_1)})

Abs : mp $tupla$ $\longrightarrow$ map {Rep($tupla$)}
Abs($tupla$) $=_{\text{obs}}$ m: map | $coordenadas(m) = tupla.coordenadas$

Algoritmos

---

iCrearMapa $\rightarrow res$: mp
1: $res \leftarrow \langle Vacio(), Vacio() \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iAgregarCoor(**in** c_1 : coor, **in/out** m : mp)

1: $AgregarRapido(c_1, m.coordenadas)$ ▷ $\Theta(1)$
2: $AgregarAdelante(AgregarAdelante(c_1, Vacia()), m.conexiones)$ ▷ $\Theta(1)$
3: $pnueva \leftarrow (Primero(m.coordenadas))$ ▷ $\Theta(1)$
4: $plista \leftarrow BuscarLista(Crear(Latitud(c_1)+1, Longitud(c_1)), m.conexiones)$ ▷ $O(1ong(coordenadas(m)))$
5: **if** $plista \neq NULL$ **then** ▷ $O(1ong(coordenadas(m)))$
6: $UnirListas(pnueva, plista)$
7: **end if**
8: $plista \leftarrow BuscarLista(Crear(Latitud(c_1), Longitud(c_1)+1), m.conexiones)$ ▷ $O(1ong(coordenadas(m)))$
9: **if** $plista \neq NULL$ **then** ▷ $O(1ong(coordenadas(m)))$
10: $UnirListas(pnueva, plista)$
11: **end if**
12: **if** $Latitud(c_1 > 0)$ **then**
13: $plista \leftarrow BuscarLista(Crear(Latitud(c_1)-1, Longitud(c_1)), m.conexiones)$ ▷ $O(1ong(coordenadas(m)))$
14: **if** $plista \neq NULL$ **then** ▷ $O(1ong(coordenadas(m)))$
15: $UnirListas(pnueva, plista)$
16: **end if**
17: **end if**
18: **if** $Longitud(c_1 > 0)$ **then**
19: $plista \leftarrow BuscarLista(Crear(Latitud(c_1), Longitud(c_1)-1), m.conexiones)$ ▷ $O(1ong(coordenadas(m)))$
20: **if** $plista \neq NULL$ **then** ▷ $O(1ong(coordenadas(m)))$
21: $UnirListas(pnueva, plista)$
22: **end if**
23: **end if**
24: $iterador \leftarrow CrearIt(m.conexiones)$ ▷ $\Theta(1)$
25: **while** $HaySiguiente(iterador)$ **do** ▷ $O(long(coordenadas(m)))$
26: **if** EsVacia?(Siguiente(iterador)) **then** ▷ Eliminas las listas que quedaron vacias
27: $EliminarSiguiente(iterador)$
28: **else**
29: $Avanzar(iterador)$
30: **end if**
31: **end while**

Complejidad: $4 * \Theta(1) + 5 * O(coordenadas(m)) = O(coordenadas(m))$
Justificación: Lo peor que puede pasar es que no haya ninguna conexion y ninguna de las coordenadas buscadas este, causando que los 4 buscar lista y el while tomen O(coordenadas(m)).

---

iCoordenadas(**in** m : mp) $\rightarrow$ res : itConj(coor)

1: $res \leftarrow crearIt(m \rightarrow coordenadas)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iPosExistente(**in** c_1 : coor, **in** m : mp) $\rightarrow$ res : Bool

1: **if** $c_1 < Cardinal(m.coordenadas)$ **then** ▷ $\Theta(1)$
2: $res \leftarrow true$
3: **else**
4: $res \leftarrow false$
5: **end if**

Complejidad: $\Theta(1)$

iHayCamino(**in** c_1, c_2 : `coor`, **in** m : `mp`) $\rightarrow$ res : Bool

1: $iterador \leftarrow CrearIt(m.conexiones)$ ▷ $O(\#coordenadas(m))$
2: **while** $\neg Esta(c_1, Siguiente(iterador))$ **do**
3: $\quad Avanzar(iterador)$ ▷ $\Theta(1)$
4: **end while**
5: **if** $Esta(c_2, Siguiente(iterador))$ **then** ▷ $O(\#coordenadas(m))$
6: $\quad res \leftarrow true$
7: **else**
8: $\quad res \leftarrow false$
9: **end if**

Complejidad: $\#coordenadas(m) + \#coordenadas(m) = \#coordenadas(m)$
Justificación: El tamanio de la lista de conexiones es igual al de el conjunto de coordenadas, entonces toma lo mismo recorrerlas en el peor caso.

---

iEsta(**in** a : α, **in** l : `lista(`α`)`) $\rightarrow$ res : bool ▷ Operacion privada usada para averiguar pertenencia en una lista

1: $iterador \leftarrow CrearIt(l)$ ▷ $\Theta(1)$
2: **while** $HaySiguiente(iterador) \wedge Siguiente(iterador) \neq a$ **do** ▷ $O(long(l))$
3: $\quad Avanzar(iterador)$
4: **end while**
5: **if** $HaySiguiente(iterador)$ **then** ▷ $\Theta(1)$
6: $\quad res \leftarrow true$
7: **else**
8: $\quad res \leftarrow false$
9: **end if**

Complejidad: $O(long(l)) + \Theta(1) = O(long(l))$
Justificación: El tamanio de la lista de conexiones es igual al de el conjunto de coordenadas, entonces toma lo mismo recorrerlas en el peor caso.

---

iBuscarLista(**in** a : α, **in** l : `lista(lista(`α`))`) $\rightarrow$ res : puntero(lista(α)) ▷ Operacion privada usada para crear los

1: $iterador \leftarrow CrearIt(l)$ ▷ $\Theta(1)$ grupos de equivalencia de conexiones del mapa
2: **while** $HaySiguiente(iterador) \wedge \neg Esta(a, Siguiente(iterador))$ **do** ▷ $O(long(l))$
3: $\quad Avanzar(iterador)$
4: **end while**
5: **if** $HaySiguiente(iterador)$ **then** ▷ $\Theta(1)$
6: $\quad res \leftarrow (Siguiente(iterador))$
7: **else**
8: $\quad res \leftarrow NULL$
9: **end if**

Complejidad: $O(long(l)) + \Theta(1) + \Theta(1) = O(long(l))$
Justificación: El tamanio de la lista de conexiones es igual al de el conjunto de coordenadas, entonces toma lo mismo recorrerlas en el peor caso.

---

iUnirListas(**in/out** l_1, l_2 : `puntero(lista(`α`))`) ▷ Operacion privada usada para averiguar pertenencia en una lista

1: **if** $l_1 \neq l_2$ **then** ▷ $\Theta(1)$
2: **while** $\neg EsVacia?(l_2)$ **do** ▷ $\Theta(long(l_2))$ grupos de equivalencia de conexiones del mapa
3: $AgregarAdelante(*l_1, Primero(l_2))$
4: $Fin(l_2)$
5: **end while**
6: **end if**

Complejidad: $O(long(l_2))$
Justificación: Se agregan uno a uno los elementos de l_2 a l_1 si son diferentes, pero si son iguales no se hace nada(osea la funcion es omega de 1 pero no es omega de long($*l_2$)).

6. Módulo DiccionarioAlfabetico(α)

El módulo DiccionarioAlfabetico provee un diccionario con claves de tipo String y acceso en $\Theta(L)$, donde L es el largo máximo de las claves.

Para describir la complejidad de las operaciones, vamos a llamar *copy*(a) al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una función de α en $\mathbb{N}$) para describir el costo de inserción de un elemento. Por lo tanto el costo de inserción sería O(L+copy(a)). El costo de obtener el conjunta de claves seria $\Theta(1)$.

Interfaz

parámetros formales
géneros *DiccA*(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: *DiccA*(α).

géneros: `diccA(`α`)`.

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

DEFINIR(**in** n: `String`,**in** a: α, **in/out** d: `diccA(`α`)`)
Pre $\equiv$ {true}
Post $\equiv$ {$d =_{\text{obs}} definir(c, s, d_0)$}
Complejidad: O($L + copy(a)$)
Descripción: Agrega al diccionario el significado dada la clave

ESTADEFINIDO(**in** n: `String`, **in** d: `diccA(`α`)`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} def?(c, d)$}
Complejidad: $\Theta(L)$
Descripción: devuelve `true` si y sólo si d esta definido para la clave c

OBTENER(**in** n: `String`, **in** d: `diccA(`α`)`) $\rightarrow$ *res* : `ITERADOR`
Pre $\equiv$ {def?(n)}
Post $\equiv$ {$res =_{\text{obs}} obtener(c, d)$}
Complejidad: $\Theta(L)$
Descripción: devuelve el significado de la clave c

CLAVES(**in** d: `diccA(`α`)`) $\rightarrow$ *res* : `conj(String)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} claves(d)$}
Complejidad: $\Theta(1)$
Descripción: Devuelve el conjunto de las claves dado el diccionario

Representación

`DiccAlfabetico` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero(nodo)`, *claves*: `lista(`*string*`)`)
donde `nodo` es `tupla`(*siguientes*: `arregloEstatico(nodo)`, *significado*: α))

Rep : diccA(α) $\longrightarrow$ bool

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

1. Existe un unico camino entra cada nodo y la raiz, es decir no hay ciclos en el árbol representado
2. Todos los nodos hojas, es decir, todos lo que tienen su arreglo siguientes con todas sus posiciones en NULL, tienen que tener un valor distinto de NULL
3. Una palabra pertenece al conjunto de claves si y solo si pertenece al árbol. Se dirá que un string S pertenece al árbol si empezando desde el nodo raiz y en cada paso, yendo al nodo que se encuentra en la posicion ord(S[i]) del arreglo de siguientes, donde I es el número de paso, se llega a un signficado distinto de NULL para cuando se termina de hacer una cantidad de pasos igual al tamaño de la palabra.

Abs : estre d $\longrightarrow$ diccA(α) {Rep(d)}
Abs(d) $=_{\text{obs}}$ e: diccA(α) | ($\#claves(d) = longitud(e.claves)$) $\wedge$
($\forall i$: claves(d)) (def?(i, d) $\Leftrightarrow iEstaDefinido(i, e)$) $\wedge_{\text{L}}$ ($obtener(i, d) iObtener(i, e)$)

Algoritmos

---

iVacio() $\rightarrow$ res : DiccA(α)
1: $res \leftarrow \langle NULL, Vacia() \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iDefinir(**in** n : `String`,**in** a : α, **in/out** d : `diccA`(α)
1: **if** $iEstaDefinido(n, d) == false$ **then** ▷ $\Theta(1)$
2: $agregarAtras((d.claves), a)$ ▷ $\Theta(copy(a))$
3: **end if**
4: **if** $d.raiz == NULL$ **then** ▷ $\Theta(1)$
5: $arreglo \leftarrow CrearArreglo(256)$ ▷ $\Theta(1)$
6: $nuevo \leftarrow \langle arreglo, NULL \rangle$ ▷ $\Theta(1)$
7: $d.raiz \leftarrow nuevo$ ▷ $\Theta(1)$
8: **end if**
9: $nodoAux \leftarrow d.raiz$ ▷ $\Theta(1)$
10: $indice \leftarrow 0$ ▷ $\Theta(1)$
11: **while** $nodoAux \neq NULL \wedge indice < longitud(n)$ **do** ▷ L veces como máximo
12: **if** $(*nodoAux).siguientes(n[i]) \neq NULL$ **then** ▷ $\Theta(1)$
13: $nodoAux \leftarrow (*nodoAux).siguientes(n[i])$
14: **else**
15: $arreglo \leftarrow CrearArreglo(256)$ ▷ $\Theta(1)$
16: $nuevo \leftarrow \langle arreglo, NULL \rangle$ ▷ $\Theta(1)$
17: $(*nodoAux).siguientes(n[i]) \leftarrow nuevo$ ▷ $\Theta(1)$
18: $nodoAux \leftarrow (*nodoAux).siguientes(n[i])$ ▷ $\Theta(1)$
19: **end if**
20: $indice \leftarrow indice + 1$ ▷ $\Theta(1)$
21: **end while**
22: $(*nodoAux).significado \leftarrow a$ ▷ $\Theta(1)$

Complejidad: O($copy(a) + L$)
Justificación: Todos los pasos se hacen en $\Theta(1)$ exceptuando el proceso de agregar una palabra al conjunto en caso de no estar definido aun el cual se hace en $\Theta(copy(a))$. Además se repite un loop en el cuál todo en su interior se hace en $\Theta(1)$ a lo sumo L veces por lo que se obtiene en el peor caso la complejidad seria O($copy(a) + L$)

---

iEstaDefinido(**in** n : `String`, **in** d : `diccA(`α`)`) $\rightarrow$ res : $bool$

```
1: indice ← 0                                                          ▷ Θ(1)
2: nodoAux ← d.raiz                                                    ▷ Θ(1)
3: while indice < longitud(n) do                                       ▷ longitud(n)veces
4:     if nodoAux = NULL then                                          ▷ Θ(1)
5:         res ← false                                                 ▷ Θ(1)
6:     end if
7:     nodoAux ← (∗nodoAux).siguientes(n[i])                           ▷ Θ(1)
8:     indice ← indice + 1                                             ▷ Θ(1)
9: end while
10: if (nodoAux == NULL) ∨ ((∗nodoAux).significado = NULL) then        ▷ Θ(1)
11:     res ← false                                                    ▷ Θ(1)
12: else
13:     res ← true                                                     ▷ Θ(1)
14: end if
```

Complejidad: $\Theta(L)$
Justificación: Todos las instrucciones tienen complejidad $\Theta(1)$ y hay un loop que se repite L veces, cuyas instrucciones tienen complejidad total $\Theta(1)$ por lo que la complejidad resultante sería L* $\Theta(1)$ es decir $\Theta(L)$

---

iObtener(**in** n : `String`, **in** d : `diccA(`α`)`) $\rightarrow$ res : α

```
1: indice ← 0                                                          ▷ Θ(1)
2: nodoAux ← d.raiz                                                    ▷ Θ(1)
3: while indice < longitud(n) do                                       ▷ longitud(n)veces
4:     nodoAux ← (∗nodoAux).siguientes(n[i])                           ▷ Θ(1)
5:     indice ← indice + 1                                             ▷ Θ(1)
6: end while
7: res ← (∗nodoAux).significado                                        ▷ Θ(1)
```

Complejidad: $\Theta(L)$
Justificación: Todos las instrucciones tienen complejidad $\Theta(1)$ y hay un loop que se repite L veces, cuyas instrucciones tienen complejidad total $\Theta(1)$ por lo que la complejidad resultante sería L* $\Theta(1)$ es decir $\Theta(L)$

---

iClaves(**in** d : `diccA(`α`)`) $\rightarrow$ res : $lista()$

```
1: res ← (d.claves)                                                    ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

7. Módulo PokemonGo

Interfaz

se explica con: JUEGO

géneros: pg.

CREARJUEGO(**in** *mapa* : map) → *res* : pg
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} creaarJuego(mapa)$}
Complejidad: $\Theta(maxLatitud(coordenadas(m)) * maxLongitud(coordenadas(m))$
Descripción: crea un juego sin jugadores ni pokemones

AGREGARPOKEMON(**in** p : pokemon, **in** c : coordenada, **in/out** j : pg)
Pre ≡ {j $=_{\text{obs}}$ j_0 ∧puedoAgregarPokemon(c, j_0)}
Post ≡ {j $=_{\text{obs}}$ $agregarPokemon(p, c, j_0)$}
Complejidad: $O(|P| + EC * log(EC))$
Descripción: agrega un pokemon de tipo p al juego j en la coordenada c

AGREGARJUGADOR(**in/out** j : pg)
Pre ≡ {j $=_{\text{obs}}$ j_0}
Post ≡ {agregarJugador(j_0)}
Complejidad: $O(J)$
Descripción: agrega un nuevo jugador al juego y le asigna un id

CONECTARSE(**in** e : jugador, **in** c : coordenada, **in/out** j : pg)
Pre ≡ {j $=_{\text{obs}}$ $j_0 \wedge e \in jugadores(j_0) \wedge_{\text{L}} \neg estaConectado?(e, j_0) \wedge posExistente(c, mapa(j_0))$}
Post ≡ {j $=_{\text{obs}}$ agregarJugador(e, c, j_0)}
Complejidad: $O(log(EC))$
Descripción: conecta a un jugador desconectado del juego en una posicion del mapa

DESCONECTARSE(**in** e : jugador, **in/out** j : pg)
Pre ≡ {j $=_{\text{obs}}$ $j_0 \wedge e \in jugadores(j_0) \wedge_{\text{L}} estaConectado?(e, j_0)$}
Post ≡ {j $=_{\text{obs}}$ desconectar(e, j_0)}
Complejidad: $O(log(EC))$
Descripción: desconecta del juego a un jugador conectado

MOVERSE(**in** e : jugador, **in** c : coordenada, **in/out** j : pg)
Pre ≡ {j $=_{\text{obs}}$ $j_0 \wedge e \in jugadores(j_0) \wedge_{\text{L}} estaConectado?(e, j_0) \wedge posExistente(c, mapa(j_0))$}
Post ≡ {j $=_{\text{obs}}$ moverse(e, c, j_0)}
Complejidad: $O((PS + PC) * |P| + log(EC))$
Descripción: resuelve en el juego el movimiento de un jugador, capturando los pokemones que deben ser capturados, actualizando los turnos para captura de todos los pokemones y, si realizo un movimiento invalido, sanciona y elimina del juego al jugador que se movio

MAPA(**in** j : pg) → res : map
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ mapa(j)}
Complejidad: $\Theta(1)$
Descripción: devuelve un el mapa asociado al juego por referencia

JUGADORES(**in** j : pg) → res : itConj(jugador)
Pre ≡ {true}
Post ≡ {alias(esPermutacion?(SecuSuby(res), jugadores(j)))}
Complejidad: $\Theta(1)$
Descripción: devuelve un iterador no modificable al conjunto de jugadores del juego

ESTACONECTADO(**in** e : jugador, **in** j : pg) → res : bool
Pre ≡ {$e \in jugadores(j)$}
Post ≡ {res $=_{\text{obs}}$ estaConectado(e, j)}

Complejidad: $\Theta(1)$
Descripción: retorna true si el jugador esta conectado al juego y false en caso contrario

SANCIONES(**in** e : `jugador`, **in** j : `pg`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{e \in jugadores(j)\}$
Post $\equiv$ {res $=_{\text{obs}}$ sanciones(e, j)}
Complejidad: $\Theta(1)$
Descripción: retorna la cantidad de sanciones del jugador e

POSICION(**in** e : `jugador`, **in** j : `pg`) $\rightarrow$ res : `coordenada`
Pre $\equiv$ $\{e \in jugadores(j) \wedge_{\text{L}} estaConectado(e, j)\}$
Post $\equiv$ {res $=_{\text{obs}}$ posicion(e, j)}
Complejidad: $\Theta(1)$
Descripción: retorna la posicion del jugador e en el juego

POKEMONS(**in** e : `jugador`, **in** j : `pg`) $\rightarrow$ res : `itConj(tupla⟨ pokemon, nat ⟩))`
Pre $\equiv$ $\{e \in jugadores(j)\}$
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), aMuliconj(pokemons(e, j))))}
Complejidad: $\Theta(1)$
Descripción: retorna un iterador no modificable a un conjunto de tuplas $\langle pokemon, cantidad \rangle$

EXPULSADOS(**in** j : `pg`) $\rightarrow$ res : `itConj(jugador)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), expulsados(j))}
Complejidad: $\Theta(J)$
Descripción: retorna un iterador no modificable al conjunto de jugadores expulsados del juego

POSCONPOKEMONS(**in** j : `pg`) $\rightarrow$ res : `itConj(coordenada)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ posConPokemons(j)}
Complejidad: $\Theta(1)$
Descripción: retorna un iterador al conjunto de pokemones salvajes

POKEMONENPOS(**in** c : `coordenada`, **in** j : `pg`) $\rightarrow$ res : `pokemon`
Pre $\equiv$ {c $\in$ posConPokemons(j)}
Post $\equiv$ {res $=_{\text{obs}}$ pokemonEnPos(c, j)}
Complejidad: $\Theta(1)$
Descripción: retorna el pokemon en la coordenada c del juego

CANTMOVIMIENTOSPARACAPTURA(**in** c : `coordenada`, **in** j : `pg`) $\rightarrow$ res : `nat`
Pre $\equiv$ {c $\in$ posConPokemons(j)}
Post $\equiv$ {res $=_{\text{obs}}$ cantMovimientosParaCaptura(c, j)}
Complejidad: $\Theta(1)$
Descripción: retorna los turnos que faltan para capturar a un pokemon en la posicion c del mapa del juego

PUEDOAGREGARPOKEMON(**in** c : `coordenada`, **in** j : `pg`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ puedoAgregarPokemon(c, j)}
Complejidad: $\Theta(1)$
Descripción: retorna los true si la coordenada c pertenece al mapa del juego y no hay ningun pokemon a distancia 5 o menos de c en el juego

HAYPOKEMONCERCANO(**in** c : `coordenada`, **in** j : `pg`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ hayPokemonCercano(c, j)}
Complejidad: $\Theta(1)$
Descripción: retorna los true si hay un pokemon en el juego a distancia 2 o menos de c, la coordenada c es pasada por referencia y deja la posicion del pokemon en c si retorna true

POSPOKEMONCERCANO(**in** c : `coordenada`, **in** j : `pg`) $\rightarrow$ res : `coordenada`
Pre $\equiv$ {hayPokemonCercano(c, j)}

Post $\equiv$ {res $=_{\text{obs}}$ posPokemonCercano(c, j)}
Complejidad: $\Theta(1)$
Descripción: retorna la posicion del pokemon que esta a distancia 2 o menos

ENTRENADORESPOSIBLES(**in** c: `coordenada`, **in** es: `conj(jugador)`, **in** j: `pg`) $\rightarrow$ res : `conj(jugador)`
Pre $\equiv$ {hayPokemonCercano(c, j) $\wedge$ es $\subseteq$ jugadoresConectados(j)}
Post $\equiv$ {res $=_{\text{obs}}$ entrenadoresPosibles(c, es, j)}
Complejidad: $\Theta(\#es)$
Descripción: retorna el conjunto de jugadores a distancia menor o igual a 2 de la posicion c, el conj es es pasado como referencia constante

INDICEDERAREZA(**in** p: `pokemon`, **in** j: `pg`) $\rightarrow$ res : `nat`
Pre $\equiv$ {p $\in$ todosLosPokemon(j)}
Post $\equiv$ {res $=_{\text{obs}}$ inidiceDeRareza(p, j)}
Complejidad: $O(|P|)$
Descripción: retorna indice de rareza del pokemon p

CANTPOKEMONSTOTALES(**in** j: `pg`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ cantPokemonsTotales(j)}
Complejidad: $\Theta(1)$
Descripción: retorna la cantidad de pokemons totales del juego

CANTMISMAESPECIE(**in** p: `pokemon`, **in** j: `pg`) $\rightarrow$ res : `nat`
Pre $\equiv$ {p $\in$ todosLosPokemon(j)}
Post $\equiv$ {res $=_{\text{obs}}$ cantMismaEspecie(p, j)}
Complejidad: $O(|P|)$
Descripción: retorna la cantidad de pokemons de tipo p en el juego

Representación

`pg` **se representa con** `estr`

donde `estr` es `tupla`(*mapa*: `map` , *grilla*: `diccR(diccR(infoMapa))`
, *pokemonesSalvajes*: `conj(coordenada)` , *datosPokemones*: `diccA(nat)`
, *jugadores*: `conj(jugador)` , *datosJugadores*: `diccR(infoJugador)`
, *pokemonesTotales*: `nat`)

donde `infoMapa` es `tupla`(*cordenadaPresente*: `bool` , *jugadoresEnCoordenada*: `conj(jugador)`
, *hayPokemon*: `bool` , *pokemonEnCoordenada*: `pokemon`
, *jugadoresCercanos*: `colaPrior(jugadorCercano)` , *movimientosParaCaptura*: `nat`)

donde `infoJugador` es `tupla`(*conectado*: `bool` , *coordenada*: `coordenada` , *sanciones*: `nat`
, *pokemones*: `conj(tupla(`$\langle pokemon, nat\rangle$`))` , *cantCapturados*: `nat`
, *iteradoresAPokemon*: `diccA(itConj(tupla(`$\langle pokemon, nat\rangle$`)))`
, *iteradorAPosicion*: `itConj(jugador)`
, *posicionEnCola*: `itCola(jugadorCercano)`
, *iteradorAJugadoresJuego*: `itConj(jugador)`)

Rep : estr e $\longrightarrow$ bool

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

1. El cardinal de las claves de la grilla - 1 es el maximo de las latitudes de las coordenadas del mapa y para todo significado en la grilla, el cardinal de sus claves - 1 es el maximo de las longitudes de las claves de las coordenadas del mapa y las coordenadas presentes en la grilla (resultado de obtener(longitud(coordenadas), obtener(latitud(coordenada), grilla))) estan en las coordenadas del mapa y las coordenadas del mapa estan presentes en la grilla(coordenada presente es true)

2. pokemones salvajes esta incluido en coordenadas del mapa

3. si una coordenada no esta en coordenadas mapa y su latitud y su longitud son menores o iguales a la maxima latitud y la maxima longitud de las coordenadas del mapa, esta coordenada no esta presente en la grilla(coordenada presente es false)

4. jugadores de juego esta incluido en claves de datosJugadores y sus sanciones son menores a 5, si un jugador esta en claves de datosJugadores y no esta en el conjunto de jugadores, sus sanciones son iguales a 5 y conectado es false

5. para todas las claves de jugadores, si estan conectados, estan en jugadoresEnCoordenada de obtenerlos en la grilla.

6. para todas las coordenadas c del mapa, los jugadores en coordenada de obtener Longitud de obtener latitud de la grilla, son claves de jugadores de estr, estan conectados y su cordenada es c

7. para todo significado de todo significado de la grilla, si la cooerdenada esta presente y hay un pokemon, el resultado de pasar a conjunto los jugadores de la cola jugadoresCercanos es la union de todos los jugadoresEnCola de las posiciones de la grilla que estan a distancia 2 o menor y estan presentes, sino la cola esta vacia

8. las coordenadas de pokemonesSalvajes estan incluidas en las coordenadas del mapa y para cada posicion correspondiente en la grilla hayPokemon en esa coordenada es true

9. la suma de los significados de datos de pokemones mas el resultado de sumar todos los segundos terminos de los pokemones capturados de los significados de datosJugadores es igual a pokemones totales

10. para cada clave de datosPokemones, su significado es igual a la cantidad de coordenadas en pokemones salvajes tales que su pokemonEnCoordenada en la grilla es igual a la clave mas la suma de las cantidades de pokemonesCapturados para cada jugador en datosJugador tales que el pokemon es igual la clave

11. si armas un conjunto con los nombres de los pokemones en la grilla que sus coordenadas estan en pokemones salvajes y los nombres de los pokemones de los pokemones capturados de todos los significados de datosDeJugadores, este conjunto esta incluido en las claves de datosPokemones y para todas las claves de datosPokemones que no esten en este conjunto, su significado es 0

12. los jugadores del juego corresponden a las claves de datosJugadores tal que las sanciones de obtenerlo en datosJugadores son menores a 5

13. la suma de los significados de datos de pokemones es igual a pokemones totales

14. para todo jugador en claves, si el jugador tiene menos de 5 sanciones y esta conectado y hay un pookemon a distancia menor o igual a 2 de su coordenada, su posicion en Cola es un iterador a la cola que esta en posPokemonCercano(c, j),

1. maxLatitud(coordenadas(e.mapa)) = #claves(e.grilla)-1 $\wedge(\forall n : nat)$n < #claves(grilla) $\Rightarrow_{\text{L}}$ #claves(obtener(n, e.grilla)) - 1 = maxLongitud(coordenadas(e.mapa)) $\wedge_{\text{L}}$ ($\forall$ c: coor) (c $\in$coordenadas(e.mapa) $\Rightarrow_{\text{L}}$ obtener(Longitud(c), obtener(Latitud(c))).coordenadaPresente) $\wedge$ (c $\notin$coordenadas(e.mapa) $\Rightarrow_{\text{L}}$ $\neg$obtener(Longitud(c), obtener(Latitud(c))).coo

2. e.pokemonesSalvajes $\subseteq$ coordenadas(e.mapa)

3. ($\forall$j:jugador)j$\in$ claves(e.datosJugadores) $\wedge_{\text{L}}$ obtener(j, e.datosJugadores).conectado$\Rightarrow_{\text{L}}$ obtener(j, e.datosJugadores).coorden $\in$ coordenadas(e.mapa)

4. (∀c : coordenada)(c ∈ coordenadas(e.mapa) ∧ obtener(Longitud(c), obtener(Latitud(c))).hayPokemon) $\Rightarrow_{\text{L}}$ (c∈ e.pokemonesSalvajes ∧$obtener(Longitud(c), obtener(Latitud(c))).movimientosParaCaptura¿9\Rightarrow_{\text{L}} esVacia(obtener(Longa$: coordenada)(c ∈ coordenadas(e.mapa) $\Rightarrow_{\text{L}}$

Abs : estr e $\longrightarrow$ pg {Rep(e)}
Abs(e) $=_{\text{obs}}$ j: pg | mapa(j) = e.map ∧ e.jugadore = jugadores(j) ∧
expulsados(j) = claves(e.datosJugadores) - e.jugadores $\wedge_{\text{L}}$ (∀p:jugador)p ∈ jugadores(j) $\Rightarrow_{\text{L}}$ (
estaConectado(p, j) = obtener(p, e.datosJugadores).conectado ∧ sanciones(p, j) = obtener(p, e.datosJugadores).sanciones ∧ pokemons(p, j) =
tuplasAMulticonj(obtener(p, e.datosJugadores).pokemons) $\wedge_{\text{L}}$ estaConectado(p, j)$\Rightarrow_{\text{L}}$
posicion(p, j) = obtener(p, e.datosJugadores).posicion) ∧
posConPokemon(j) = e.pokemonesSalvajes $\wedge_{\text{L}}$ (∀ c : coordenada) c ∈ posConPokemon(j) $\Rightarrow_{\text{L}}$
pokemonEnPos(c, j) = obtener(longitud(c), obtener(latitud(c), e.grilla)).pokemon ∧
cantMovimientosParaCaptura(c, j) = obtener(longitud(c), obtener(latitud(c), e.grilla)).movimientosParaCaptura

tuplasAMulticonj : conj(tupla(pokemon × nat)) $\longrightarrow$ multiconj(pokemon)
tuplasAMulticonj(ts) $\equiv$ **if** $\emptyset(ts)$ **then**
$\emptyset$
else
$agregarKVeces(\Pi_1(dameUno(ts)), \Pi_2(dameUno(ts)), tuplasAMulticonj(sinUno(ts)))$
fi
agregarKVeces : pokemon × nat × multiconj(pokemon) $\longrightarrow$ multiconj(pokemon)
agregarKVeces(p, k, mc) $\equiv$ **if** $k = 0$ **then** mc **else** $agregarKVeces(p, k-1, Ag(p, mc))$ **fi**

Algoritmos

Algoritmos del módulo

iCrearJuego(**in** m : `map`) $\rightarrow$ res : estr

1: $iterador \leftarrow coordenadas(m)$ ▷ $\Theta(1)$
2: $maxLatitud \leftarrow 0$ ▷ $\Theta(1)$
3: $maxLongitud \leftarrow 0$ ▷ $\Theta(1)$
4: **while** haySiguiente(iterador) **do** ▷ $\Theta(\#(coordenadas(m)))$
5: **if** Latitud(Siguiente(iterador)) $>$ $maxLatitud$ **then**
6: $maxLatitud \leftarrow Latitud(Siguiente(iterador)$ ▷ $\Theta(1)$
7: **end if**
8: **if** longitud(Siguiente(iterador)) $>$ $maxLongitud$ **then**
9: $maxLongitud \leftarrow Longitud(Siguiente(iterador)$
10: **end if**
11: $avanzar(iterador)$
12: **end while**
13: $i \leftarrow 0$ ▷ $\Theta(1)$
14: $j \leftarrow 0$ ▷ $\Theta(1)$
15: $grilla \leftarrow vacio()$
16: **while** i $<$ maxLatitud + 1 **do** ▷ $\Theta(maxLatitud(coordenadas(m)) * maxLongitud(coordenadas(m)))$
17: $definir(i, vacio(), grilla)$
18: $fila \leftarrow obtener(i, grilla)$
19: **while** j $<$ maxLongitud + 1 **do**
20: $definir(j, \langle false, vacio(), false, vacia(), vacia(), 0\rangle, fila)$
21: $j \leftarrow j + 1$
22: **end while**
23: $i \leftarrow i + 1$
24: **end while**
25: $iterador \leftarrow coordenadas(map)$
26: **while** haySiguiente(iterador) **do**
27: $obtener(longitud(siguiente(iterador)), obtener(latitud(siguiente(iterador)), grilla)).coordenadaPresente \leftarrow true$
28: $avanzar(iterador)$
30: **end while**

Complejidad: $\Theta(maxLatitud(coordenadas(m)) * maxLongitud(coordenadas(m)) + \#coordenadas(m)) = \Theta(maxLatitud(coordenadas(m)) * maxLongitud(coordenadas(m))$

iAgregarPokemon(**in** p : `pokemon`, **in** c : `coordenada`, `in/out` j : `estr`)

31: $AgregarRapido(j.pokemonesSalvajes, c)$ ▷ $\Theta(1)$
2: **if** $\neg estaDefinido(p, j.datosPokemones)$ **then** ▷ $\Theta(|P|)$
3: $definir(p, 0, j.datosPokemones)$
4: **end if**
5: $temp \leftarrow obtener(p, j.datosPokemones)$ ▷ $\Theta(|P|)$
6: $temp \leftarrow temp + 1$ ▷ temp es una referencia a la
7: $j.pokemonesTotales \leftarrow j.pokemonesTotales + 1$ ▷ posicion de memoria en el diccionario
8: $coordenadaMapa \leftarrow obtener(longitud(c), obtener(latitud(c), j.grilla))$ ▷ $\Theta(1)$
9: $coordenadaMapa.hayPokemon \leftarrow true$ ▷ $\Theta(1)$
10: $coordenadaMapa.pokemonEnCoordenada \leftarrow p$ ▷ $\Theta(1)$
11: $coordenadaMapa.movimientosParaCaptura \leftarrow 0$ ▷ $\Theta(1)$
12: $cola \leftarrow vacia()$ ▷ $\Theta(1)$
13: $encolarEC(cola, c, 0, 2, j)$ ▷ Toman $O(k * log(EC))$ cada una
14: $encolarEC(cola, c, 1, 2, j)$ ▷ La suma de todas es $O(EC * log(EC))$
15: $encolarEC(cola, c, 2, 2, j)$
16: $encolarEC(cola, c, 3, 2, j)$
17: $encolarEC(cola, c, 4, 2, j)$
18: $encolarEC(cola, c, 1, 3, j)$
19: $encolarEC(cola, c, 2, 3, j)$
20: $encolarEC(cola, c, 3, 3, j)$
21: $encolarEC(cola, c, 1, 1, j)$
22: $encolarEC(cola, c, 2, 1, j)$
23: $encolarEC(cola, c, 3, 1, j)$
24: $encolarEC(cola, c, 2, 0, j)$
25: $encolarEC(cola, c, 2, 4, j)$
26: $coordenadaMapa.jugadoresCercanos \leftarrow cola$

Complejidad: $O(|P| + EC * log(EC))$

---

iAgregarJugador(**in/out** j : `estr`)

1: $jugador \leftarrow cardinal(claves(j.datosJugadores))$
2: $iterador \leftarrow agregarRapido(j.jugadores, jugador)$
3: $corInvalida \leftarrow crear(0, 0)$
4: $info \leftarrow \langle false, corInvalida, 0, vacio(), 0, crearIt(Vacio()), crearIt(Vacio()), crearIt(vacia()), iterador \rangle$
5: $definir(jugador, info, j.datosJugadores)$ ▷ esto puede tardar J o 1

Complejidad: $O(J)$

---

iConectarse(**in** e : `jugador`, **in** c : `coordenada`, **in/out** j : `estr`)

1: $info \leftarrow obtener(e, j.datosJugadores)$
2: $info.conectado \leftarrow true$
3: $info.coordenada \leftarrow c$ ▷ esto es por copia
4: $iterador \leftarrow agregarRapido(obtener(longitud(c), obtener(latitud(c), j.grilla)).jugadoresEnCoordenada, e)$
5: $info.iteradorAPosicion \leftarrow iterador$
6: **if** hayPokemonCercano(c, j) **then** ▷ c es modificado si hay, sino no
7: $info.movimientosParaCaptura \leftarrow 0$
8: $info.posicionEnCola \leftarrow encolar(obtener(Longitud, obtener(latitud, j.grilla)).jugadoresCercanos, crearTupla(e, info.cantCapturados))$
9: **end if**

Complejidad: $O(log(EC))$
Justificación: todas las operaciones realizadas son O(1) con la posible esepcion de una sola de las encolar que es O(log(EC)) si hay un pokemon cerca a la coordenada pasada como parametro

---

iDesonectarse(**in** e : `jugador`, **in/out** j : `estr`)

1: $info \leftarrow obtener(e, j.datosJugadores)$
2: $info.conectado \leftarrow false$
3: $info.coordenada \leftarrow crear(0, 0)$
4: $eliminarSiguiente(info.iteradorAPosicion)$
5: **if** hayMas(info.posicionEnCola) **then**
6: $\quad eliminar(info.posicionEnCola)$
7: **end if**
8: $info.posicionEnCola \leftarrow crearIt(vacia())$
9: $info.iteradorAPosicion \leftarrow crearIt(vacio())$

Complejidad: $O(log(EC))$
Justificación: todas las operaciones realizadas son O(1) menos eliminar del iterador de la cola que es O(log(EC)) si hay un pokemon cercano al jugador

iMoverse(**in** e : `jugador`, **in** c : `coordenada`, **in/out** j : `estr`)

1: $info \leftarrow obtener(e, j.datosJugadores)$
2: $c_2 \leftarrow info.coordenada$
3: $desconectarse(e, j)$ ▷ $\Theta(log(EC))$
4: **if** distancia(c, c_2) ¿100 ∨¬hayCamino(c, c_2, j.mapa) **then**
5: $info.sanciones \leftarrow info.sanciones + 1$
6: **end if**
7: **if** info.sanciones ¡5 **then**
8: $conectarse(e, c, j)$ ▷ $\Theta(log(EC))$
9: **else**
10: $pokeIterador \leftarrow crearIt(info.pokemones)$
11: **while** haySiguiente(pokeiterador) **do**
12: $cantidad \leftarrow obtener(primera(siguiente(pokeiterador)), j.datosPokemones)$
13: $cantidad \leftarrow cantidad - segunda(siguiente(pokeIterador))$
14: $eliminarSiguiente(pokeiterador)$
15: **end while**
16: $j.pokemonesTotales \leftarrow j.pokemonesTotales - info.cantCapturados$
17: $info.cantCapturados \leftarrow 0$
18: $eliminarSiguiente(info.iteradorAJugadoresJuego)$
19: $info.iteradorAJugadoresJuego \leftarrow crearIt(vacio())$
20: $info.iteradoresAPokemon \leftarrow vacio()$
21: **end if**
22: $iterador \leftarrow crearIt(j.pokemonesSalvajes)$
23: **while** haySiguiente(iterador) **do**
24: $posPoke \leftarrow siguiente(iterador)$
25: $posMapa \leftarrow obtener(longitud(posPoke), obtener(latitud(posPoke), j.grilla))$
26: **if** distancia(posPoke, c) $< 5 \vee$ distancia(posPoke, c_2) > 4 **then**
27: $posMapa.movimientosParaCaptura \leftarrow 0$
28: **else**
29: $posMapa.movimientosParaCaptura \leftarrow posMapa.movimientosParaCaptura + 1$
30: **end if**
31: **if** posMapa.movimientosParaCaptura = 10∧¬esVacia(posMapa.jugadoresCercanos) **then**
32: $elQueCaptura \leftarrow jugador(proximo(posMapa.jugadoresCercanos))$
33: $infoEntrenador \leftarrow obtener(elQueCaptura, j.datosJugadores)$
34: $posMapa.hayPokemon \leftarrow false$
35: $tipo \leftarrow posMapa.pokemonEnCoordenada$
36: $posMapa.pokemonEnCoordenada \leftarrow vacia()$
37: $posMapa.jugadoresCercanos \leftarrow vacia()$
38: $posMapa.movimientosParaCaptura \leftarrow 0$
39: $infoEntrenador.cantCapturados \leftarrow infoEntrenador.cantCapturados + 1$
40: **if** estaDefinido(tipo, infoEntrenador.iteradoresAPokemon) **then** ▷ $O(|P|$ porque en ambos casos busca en el diccionario Alfabetico(o define))
41: $pokeIterador \leftarrow obtener(tipo, infoEntrenador.iteradoresAPokemon)$
42: $siguiente(pokeIterador) \leftarrow \langle tipo, segundo(siguiente(pokeIterador)) + 1\rangle$
43: **else**
44: $pokeIterador \leftarrow agregarRapdio(infoEntrenador.pokemones, \langle tipo, 1\rangle)$
45: $definir(tipo, pokeIterador, infoEntrenador.iteradoresAPokemon)$
46: **end if**
47: $eliminarSiguiente(iterador)$
48: **else**
49: $avanzar(iterador)$
50: **end if**
51: **end while**

Complejidad: $O((PS + PC)|P| + log(EC))$
Justificación: desconectar al jugador toma log(EC), reconectarlo toma lo mismo, si el jugador hay que eliminarlo del juego, toma PC * $|P|$ y siempre hay que resolver la situacion de todos los pokemones, los que fueron capturados toma $|P|$ resolverlos, los que no toma O(1) osea el peor caso es cuando todos los pokemones salvajes fueron capturados $|P|$ * PS, esto da 2 posibles resultados dependiendo de si se expulso al jugador o no. O(2 * log(EC) +$|P|$ * PS) si no fue expulsado y O(log(EC) + $|P|$(PS + PC)) ambos estan en O((PS + PC)$|P|$ + log(EC))

iMapa(**in** j : `estr`) $\rightarrow$ res : map

1: $res \leftarrow j.mapa$

Complejidad: $\Theta(1)$

iJugadores(**in** j : `estr`) $\rightarrow$ res : itConj(jugador)

1: $res \leftarrow crearIt(j.jugadores)$

Complejidad: $\Theta(1)$

iEstaConectado(**in** e : `jugador`, **in** j : `estr`) $\rightarrow$ res : bool

1: $res \leftarrow obtener(e, j.datosJugadores).conectado$

Complejidad: $\Theta(1)$

iSanciones(**in** e : `jugador`, **in** j : `estr`) $\rightarrow$ res : nat

1: $res \leftarrow obtener(e, j.datosJugadores).sanciones$

Complejidad: $\Theta(1)$

iPosicion(**in** e : `jugador`, **in** j : `estr`) $\rightarrow$ res : coordenada

1: $res \leftarrow obtener(e, j.datosJugadores).posicion$

Complejidad: $\Theta(1)$

iPokemones(**in** e : `jugador`, **in** j : `estr`) $\rightarrow$ res : itConj(tupla$\langle$ pokemon, nat $\rangle$)

1: $res \leftarrow crearIt(obtener(e, j.datosJugadores).pokemones)$

Complejidad: $\Theta(1)$

iExpulsados(**in** j : `estr`) $\rightarrow$ res : itConj(jugador)

1: $iterador \leftarrow claves(j.datosJugadores)$
2: $expulsados \leftarrow vacio()$
3: **while** haySiguiente(iteraador) **do** ▷ $\Theta(J)$
4: **if** obtener(Siguiente(iterador), d.datosJugadores).sanciones = 5 **then**
5: $agregarRapido(expulsados, siguiente(iterador))$
6: **end if**
7: $avanzar(iterador)$
8: **end while**
9: $res \leftarrow crearIt(expulsados)$

Complejidad: $\Theta(J)$

iPosConPokemon(**in** j : `estr`) $\rightarrow$ res : itConj(coordenada)

1: $res \leftarrow crearIt(j.pokemonesSalvajes)$

Complejidad: $\Theta(1)$

iPokemonEnPos(**in** c: `coordenada`, **in** j: `estr`) $\rightarrow$ res : pokemon

1: $res \leftarrow obtener(longitud, obtener(latitud, j.grilla)).pokemonEnCoordenada$

Complejidad: $\Theta(1)$

---

iCantMovimientosParaCaptura(**in** c: `coordenada`, **in** j: `estr`) $\rightarrow$ res : nat

1: $res \leftarrow obtener(longitud, obtener(latitud, j.grilla)).movimientosParaCaptura$

Complejidad: $\Theta(1)$

---

iPuedoAgregarPokemon(**in** c: `coordenada`, **in** j: `estr`) $\rightarrow$ res : bool

1: **if** estaDefinido(Latitud(c), j.grilla) $\wedge$
2: estaDefinido(Longitud(c), Obtener(Latitud(c), j.grilla)) $\wedge\neg$ hayPokemonCercano(c, j) **then**
3: $res \leftarrow true$
4: **else**
5: $res \leftarrow false$
6: **end if**

Complejidad: $\Theta(1)$

---

iHayPokemonCercano(**in** c: `coordenada`, **in** j: `estr`) $\rightarrow$ res : bool

1: $buscarPokemon(c, 0, 2, j)$
2: $buscarPokemon(c, 1, 2, j)$
3: $buscarPokemon(c, 2, 2, j)$
4: $buscarPokemon(c, 3, 2, j)$
5: $buscarPokemon(c, 4, 2, j)$
6: $buscarPokemon(c, 1, 3, j)$
7: $buscarPokemon(c, 2, 3, j)$
8: $buscarPokemon(c, 3, 3, j)$
9: $buscarPokemon(c, 1, 1, j)$
10: $buscarPokemon(c, 2, 1, j)$
11: $buscarPokemon(c, 3, 1, j)$
12: $buscarPokemon(c, 2, 0, j)$
13: $buscarPokemon(c, 2, 4, j)$
14: **if** estaDefinido(Latitud(c), j.grilla) $\wedge$
15: estaDefinido(Longitud(c), Obtener(Latitud(c), j.grilla)) **then**
16: **if** obtener(longitud(c), obtener(latitud(c), j.grilla)).hayPokemon **then**
17: $res \leftarrow true$
18: **else**
19: $res \leftarrow false$
20: **end if**
21: **else**
22: $res \leftarrow false$
23: **end if**

Complejidad: $\Theta(1) * 13 = \Theta(1)$

---

iPosPokemonCercano(**in** c: `coordenada`, **in** j: `estr`) $\rightarrow$ res : coordenada

1: $hayPokemonCercano(c, j)$
2: $res \leftarrow c$

Complejidad: $\Theta(1)$

---

iEntrenadoresPosibles(**in** c : `coordenada`, **in** es : `conj(jugador)`, **in** j : `estr`) → res : conj(jugador)

```
1: c ← posPokemonCercano(c, j)                                          ▷ Θ(1)
2: iterador ← crearIt(cs)                                               ▷ Θ(1)
3: res ← vacio()
4: while haySiguiente(iterador) do                                      ▷ Θ(#cs)
5:     jugador ← obtener(siguiente(iterador), j.datosJugadores)
6:     if jugador.conectado ∧ distancia(c, jugador.coordenada) ≤ 4 then
7:         agregarRapido(res, siguiente(iterador))
8:     end if
9:     avanzar(iterador)
10: end while
```

Complejidad: $\Theta(\#cs)$
Justificación: $recorreselconjuntocs, todaslasotrasoperacionesnosoncostosas$

---

iIndiceDeRareza(**in** p : `pokemon`, **in** j : `estr`) → res : nat

```
1: mismos ← cantMismaEspecie(p, j)                                      ▷ Θ(|p|)
2: res ← 100 − (100 ∗ mismos/cantPokemonsTotales(j))                    ▷ Θ(|1|)
```

Complejidad: $\Theta(|p|) \in O(|P|)$
Justificación: $|p| \leq |P|$

---

iCantPokemonsTotales(**in** j : `estr`) → res : nat

```
1: res ← j.pokemonesTotales                                             ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

iCantMismaEspecie(**in** p : `pokemon`, **in** j : `estr`) → res : nat

```
1: res ← obtener(p, j.datosPokemon)                                     ▷ Θ(|p|)
```

Complejidad: $\Theta(|p|) \in O(|P|)$
Justificación: $|p| \leq |P|$

---

iEncolarEC(**in/out** $cola$: `colaPrior(jugadorCercano)`, **in** c : `coordenada`, **in** x : `nat`, **in** y : `nat`, **in** j : `estr`) ▷ Operacion privada usada para crear las colas de jugadoresCercanos, la cola y el juego se pasan por referencia

```
1: if latitud(c) + y > 1 ∧ longitud(c) + y > 1 then
2:     nuevac ← Coordenada(Latitud(c) + y − 2, Longitud(c) + x − 2)     ▷ Θ(1)
3:     if estaDefinido(Latitud(nuevac), j.grilla) ∧
4: estaDefinido(Longitud(nuevac), Obtener(Latitud(nuevac), j.grilla)) then
5:         iterador ← crearIt(obtener(longitud, obtener(latitud, j.grilla)).jugadoresEnCoordenada)
6:         while haySiguiente(iterador) do
7:             jugador ← obtener(siguiente(iterador), j.datosJugadores)
8:             it ← encolar(cola, crearTupla(jugador.cantCapturados, siguiente(iterador)))
9:             jugador.posicionEnCola ← it
10:            avanzar(iterador)
11:        end while
12:    end if
13: end if
```

Complejidad: $O(EC * log(EC))$
Justificación: Se agre/gan uno a uno los jugadores cercanos a la cola, si y solo si la coordenada esta definida en la grilla

---

iBuscarEC(**in/out** *cola* : `colaPrior(jugadorCercano)`, **in** c : `coordenada`, **in** e : `jugador`, **in** x : `nat`, **in** y : `nat`, **in** j : `estr`) ▷ Operacion privada usada para buscar la cola de jugadores cercanos, si la encuentra, agrega al jugador, y actualiza el iterador en su informacion, y llamadas a esta funcion con la cola se vuelven inefectivas(deja de ser vacia)

```
1: if cola = vacia() ∧ latitud(c) + y > 1 ∧ longitud(c) + y > 1 then
2:     nuevac ← Coordenada(Latitud(c) + y − 2, Longitud(c) + x − 2)                ▷ Θ(1)
3:     if estaDefinido(Latitud(nuevac), j.grilla) ∧
4: estaDefinido(Longitud(nuevac), Obtener(Latitud(nuevac), j.grilla)) then
5:         if obtener(longitud(c), obtener(latitud(c), j.grilla)).hayPokemon then
6:             cola ← obtener(longitud(c), obtener(latitud(c), j.grilla)).jugadoresCercanos
7:             iterador ← encolar(cola, crearTupla(obtener(e, j.datosJugadores).cantCapturados, e))
8:             obtener(e, j.datosJugadores).posicionEnCola ← iterador
9:         end if
10:    end if
11: end if
```

Complejidad: $O(log(EC))$
Justificación: La unica operacion costosa es encolar al heap

---

iBuscarPokemon(**in/out** c : `coordenada`, **in** x : `nat`, **in** y : `nat`, **in** j : `estr`) ▷ Operacion privada usada para buscar la coordenada con pokemon cercana, la coordenada c es pasada por referencia y alli se almacena el resultado. si se aplica esta funcion luego de haber encontrado la coordenada con cualquiera de las 13 combinaciones de x, y que observan las coordenadas en rango de c, el valor de c no va a cambiar(porque no pueden haber 2 pokemon a distancia menor a 3).

```
1: if latitud(c) + y > 1 ∧ longitud(c) + y > 1 then
2:     nuevac ← Coordenada(Latitud(c) + y − 2, Longitud(c) + x − 2)                ▷ Θ(1)
3:     if estaDefinido(Latitud(nuevac), j.grilla) ∧
4: estaDefinido(Longitud(nuevac), Obtener(Latitud(nuevac), j.grilla)) then
5:         if obtener(longitud(c), obtener(latitud(c), j.grilla)).hayPokemon then
6:             c ← nuevac
7:         end if
8:     end if
9: end if
```

Complejidad: $\Theta(1)$
Justificación: todas las operaciones son $\Theta(1)$