Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2016

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

Trabajo Práctico 2

Diseño

Grupo: 4 gigas de RAM

Integrante	LU	Correo electrónico
Jonathan Seijo	592/15	jon.seijo@gmail.com
Lucas Mauricio Córdoba	094/15	lmcordobaa@gmail.com
Lucas Gabriel De Bortoli	736/15	lu_cas_.97@hotmail.com.ar
Luciano Galli	534/15	lucianogalli@outlook.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Informe

1.1. General

- Intentamos dar una explicación de las estructuras que fuimos construyendo, que pueden leerse en la seccion "representacion" de cada modulo. En esos lugares contamos un poco mas sobre las decisiones particulares de cada estructura.

1.2. Juego

- La operación cantMismaEspecie de la especificación recive como parametro un multiconjunto. Reemplazamos ese parametro por un *juego*, porque usando el *juego* podemos obtener la cantidad de cada especie pokemon

1.3. Cola de Prioridad

- Sabemos que no era la unica forma de mantener a los entrenadores que esperan, podiamos usar un AVL y las complejidades seguirian valiendo. Elegimos implementarlo con un Heap porque era mas facil de implementar, o eso creimos..
- En la cola de entrenadores, se usan nodos y punteros para la estructura. Los nodos se mantienen "fijos" una vez que se agregar, hasta que son borrados. Cada vez que hay que hacer algún cambio o "swap" lo único que se modifican son los punteros "padre", "izq" y "der". Es decir que si hay algún puntero apuntando al nodo y se realiza un swap entre ese nodo y otro, dicho puntero seguirá apuntándolo.

1.4. Algoritmos privados

- Se inlcuye pre y post en castellano de los algoritmos privados que hacen manejo de memoria.

1.5. IterDiccString

- Si bien iterDiccString se explica con IteradorUnidireccional, hicimos un cambio en la aridad de "siguiente". En su TAD, el tipo que se devuelve es del mismo tipo que recibe en *crear*, pero nosotros cambiamos eso para que devuelva las tuplas que al usuario le interesan (particularmente en la funcion Pokemons() del juego)

2. Coordenada

Interfaz

2.1. Interfaz

se explica con: COORDENADA.

géneros: `coor`.

Operaciones básicas de Coordenada

CREARCOOR(**in** n_1 : `nat`, **in** n_2 : `nat`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} crearCoor$}
Complejidad: $O(1)$
Descripción: genera una coordenada nueva.

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

DISTEUCLIDEA(**in** c_1 : `coor`, **in** c_2 : `coor`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} distEuclidea(c_1, c_2)$}
Complejidad: $O(1)$
Descripción: devuelve la disntancia entre la coordenadas c_1 y c_2.

COORDENADAARRIBA(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} distEuclidea(c_1, c_2)$}
Complejidad: $O(1)$
Descripción: devuelve la coordenadas .

COORDENADAABAJO(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {Latitud(c) > 0}
Post $\equiv$ {$res =_{\text{obs}} distEuclidea(c_1, c_2)$}
Complejidad: $O(1)$
Descripción: devuelve la coordenadas .

COORDENADAALADERECHA(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} distEuclidea(c_1, c_2)$}
Complejidad: $O(1)$
Descripción: devuelve la coordenadas .

COORDENADAALAIZQUIERDA(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {Longitud(c) > 0}
Post $\equiv$ {$res =_{\text{obs}} distEuclidea(c_1, c_2)$}
Complejidad: $O(1)$

Descripción: devuelve la coordenadas .

Representación

2.2. Representacion

`coor` se representa con `estr`

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

Rep : estr e $\longrightarrow$ bool

Rep(e) $\equiv$ true

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

Abs(e) $\equiv$ c : coor / e.latitud = latitud(c) $\wedge$ e.longitud = longitud(c)

Algoritmos

2.3. Algoritmos

---

iCrearCoor(**in** n_1 : `nat`, **in** n_2 : `nat`) $\rightarrow$ res : coor

1: $res \leftarrow \langle\ n_1\ ,\ n_2\ \rangle$ ▷ $O(1)$

Complejidad: $O(1)$

---

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

1: $res \leftarrow c.latitud$ ▷ $O(1)$

Complejidad: $O(1)$

---

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

1: $res \leftarrow c.longitud$ ▷ $O(1)$

Complejidad: $O(1)$

---

iDistEuclidea(**in** c_1 : `coor`, **in** c_2 : `coor`) $\rightarrow$ res : nat

1: $a \leftarrow 0$ ▷ $O(1)$
2: **if** $c_1.latitud < c_2.latitud$ **then** ▷ $O(1)$
3: $a \leftarrow (c_1.latitud - c_2.latitud) \times (c_1.latitud - c_2.latitud)$ ▷ $O(1)$
4: **else**
5: $a \leftarrow (c_2.latitud - c_1.latitud) \times (c_2.latitud - c_1.latitud)$ ▷ $O(1)$
6: **end if**
7: $b \leftarrow 0$ ▷ $O(1)$
8: **if** $c_1.longitud < c_2.longitud$ **then** ▷ $O(1)$
9: $b \leftarrow (c_1.longitud - c_2.longitud) \times (c_1.longitud - c_2.longitud)$ ▷ $O(1)$
10: **else**
11: $b \leftarrow (c_2.longitud - c_1.longitud) \times (c_2.longitud - c_1.longitud)$ ▷ $O(1)$
12: **end if**
13: $res \leftarrow a + b$ ▷ $O(1)$

Complejidad: $O(1)$
Justificacion: $O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1)$

---

iCoordenadaArriba(**in** c : `coor`) $\rightarrow$ res : coor

1: $res \leftarrow \langle$ Latitud(c) + 1 , Longitud(c) $\rangle$ ▷ $O(1)$

Complejidad: $O(1)$

---

iCoordenadaAbajo(**in** c : `coor`) $\rightarrow$ res : coor

1: $res \leftarrow \langle$ Latitud(c) − 1 , Longitud(c) $\rangle$ ▷ $O(1)$

Complejidad: $O(1)$

---

iCoordenadaALaDerecha(**in** c : `coor`) $\rightarrow$ res : coor

1: $res \leftarrow \langle$ Latitud(c) , Longitud(c) + 1 $\rangle$ ▷ $O(1)$

Complejidad: $O(1)$

---

iCoordenadaALaIzquierda(**in** c : `coor`) $\rightarrow$ res : coor

1: $res \leftarrow \langle$ Latitud(c) , Longitud(c) − 1 $\rangle$ ▷ $O(1)$

Complejidad: $O(1)$

3. Mapa

Interfaz

3.1. Interfaz

se explica con: MAPA.

géneros: `map`.

Operaciones básicas de Mapa

CREARMAPA() $\rightarrow$ *res* : `map`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} crearMapa$}
Complejidad: $O(1)$
Descripción: Genera una mapa vacío.

AGREGARCOOR(**in** c: `coord`, **in/out** m: `map`)
Pre $\equiv$ {$m_0 =_{\text{obs}} m \wedge \neg$ posExistente(c, m_0)}
Post $\equiv$ {$m =_{\text{obs}}$ agregarCoor(c, m_0)}
Complejidad: $O(max(latitud(c), longitud(c), tam(m))^4)$
Descripción: Agrega la coordenada `c` al mapa `m`.

COORDENADAS(**in** m: `map`) $\rightarrow$ *res* : $conj(coord)$
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ coordenadas(m)}
Complejidad: $O((tam(m))^2)$
Descripción: Devuelve el conjunto de todas las coordenadas del mapa m.

POSEXISTENTE(**in** c: `coord`, **in** m: `map`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ posExistente(c, m)}
Complejidad: $O(1)$
Descripción: Verifica si la coordenada `c` existe en el mapa `m`.

HAYCAMINO(**in** c_1: `coord`, **in** c_2: `coord`, **in** m: `map`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {posExistente(c_1, m) $\wedge$ posExistente(c_2, m)}
Post $\equiv$ {$res =_{\text{obs}}$ hayCamino(c_1, c_2, m)}
Complejidad: $O(1)$
Descripción: verifica si existe una forma de llegar desde c_1 a c_2.

TAM(**in** m: `map`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {*true*}
Post $\equiv$ {$res =_{\text{obs}} tam(m)$}
Complejidad: $O(1)$
Descripción: Devuelve el máximo entre la longitud y la latitud más grandes

Especificacion de auxiliares usadas en la interfaz

TAD MAPA EXTENSION

extiende MAPA

Otras operaciones

tam : mapa m $\longrightarrow$ nat

max : nat x $\times$ nat y $\longrightarrow$ nat

maxLatitud : conj(coor) cs $\longrightarrow$ nat $\{\neg\emptyset?(cs)\}$

maxLongitud : conj(coor) cs $\longrightarrow$ nat $\{\neg\emptyset?(cs)\}$

```
tam(mapa) ≡ if #(coordenadas(mapa)) = 0 then
                0
            else
                max(maxLatitud(coordenadas(mapa)), maxLongitud(coordenadas(mapa)))
            fi
max(x ,y) ≡ if x ≥ y then x else y fi
maxLatitud(cs) ≡ if #cs = 1 then
                     latitud(dameUno(cs))
                 else
                     if latitud(dameUno(cs)) ≥ maxLatitud(sinUno(cs)) then
                         damUno(cs)
                     else
                         maxLatitud(sinUno(cs))
                     fi
                 fi
maxLongitud(cs) ≡ if #cs = 1 then
                      longitud(dameUno(cs))
                  else
                      if longitud(dameUno(cs)) ≥ maxLongitud(sinUno(cs)) then
                          damUno(cs)
                      else
                          maxLongitud(sinUno(cs))
                      fi
                  fi
```

Fin TAD

Representación

3.2. Representacion

La grilla con la que representamos el mapa es cuadrada, es decir, tiene la misma cantidad de coordenadas de alto que de ancho. m.tam es el tamaño, la cantidad de coordenadas de ancho (o alto) que tiene la grilla. Esto es asi para que no haya que distinguir entre tamaño de ancho o de alto, asi es mas sencillo escribir los algoritmos y los calculos de complejidades.
La grilla es un vector de 4 dimensiones de booleanos. Estos booleanos son los que nos dicen si hay un camino entre dos coordenadas. De esta manera, conseguimos que HayCamino sea O(1), que nos es muy util para usarlo en el Juego.
Si esta la coordenada C1=(x,y) y C2=(z,w), grilla[x][y][z][w] representa si hay camino entre C1 y C2. true significa que hay camino. Cuando agregamos una coordenada C=(i,j) a la grilla, la marcamos como existente poniendo en true el valor grilla[i][j][i][j] (Lo cual es coherente porque esto dice que C tiene camino consigo misma)

`mapa` se representa con `map`

donde `map` es `tupla`(*tam*: `nat` , *grilla*: `vector(vector(vector(vector(bool)))`)

Invariante de representacion en castellano

(1) Todos los vectores que forman la grilla son de la misma longitud
(2) tam es consistente con la longitud de la grilla

(3) Si vale grilla[x][y][z][w] vale tambien grilla[z][w][x][y]
(4) Si C no esta en la grilla, entonces no tiene camino con ninguna
(5) Si C y C' son contiguas y ambas estan en la grilla, entonces hay camino entre ellas
(6) Si hay camino entre C y C', y hay camino entre C' y C" entonces hay camino entre C y C"
Y Si hay camino entre C y C', y NO hay camino entre C' y C" entonces NO hay camino entre C y C"

Invariante de representacion en logica

(1) ((∀i: nat)(i < long(m.grilla)) $\Rightarrow_{\text{L}}$ long(m.grilla[i]) = long(m.grilla)) $\wedge_{\text{L}}$
((∀i, j: nat)(i < long(m.grilla) ∧ j < long(m.grilla)) $\Rightarrow_{\text{L}}$ long(m.grilla[i][j]) = long(m.grilla)) $\wedge_{\text{L}}$
((∀i, j, k: nat)(i < long(m.grilla) ∧ j < long(m.grilla) ∧ k < long(m.grilla)) $\Rightarrow_{\text{L}}$
long(m.grilla[i][j][k]) = long(m.grilla))

(2) m.tam = long(e.grilla)

(3) ((∀x, y, z, w: nat)(enRango(m, x, y) ∧ enRango(m, z, w)) $\Rightarrow_{\text{L}}$ (grilla[x][y][z][w] = grilla[z][w][x][y])

(4) ((∀x, y: nat)(enRango(m, x, y)) $\Rightarrow_{\text{L}}$
(¬ grilla[x][y][x][y] ⇒ (∀z, w: nat)(enRango(m, z, w)) $\Rightarrow_{\text{L}}$ ¬ grilla[x][y][z][w])))

(5) ((∀x, y, z, w: nat)
((enRango(m, x, y) $\wedge_{\text{L}}$ m.grilla[x][y][x][y]) ∧
(enRango(m, z, w) $\wedge_{\text{L}}$ m.grilla[z][w][z][w]) $\wedge_{\text{L}}$
esContigua(x,y,z,w) $\Rightarrow_{\text{L}}$ m.grilla[x][y][z][w]))

(6) ((∀a, b, c, d, e, f: nat)
(enRango(m, a, b) $\wedge_{\text{L}}$ m.grilla[a][b][a][b]) ∧
(enRango(m, c, d) $\wedge_{\text{L}}$ m.grilla[c][d][c][d]) ∧
(enRango(m, e, f) $\wedge_{\text{L}}$ m.grilla[e][f][e][f]) ∧
(a ≠ c ∨ b ≠ d) ∧ (c ≠ e ∨ d ≠ f))
$\Rightarrow_{\text{L}}$ (
((m.grilla[a][b][c][d] ∧ m.grilla[c][d][e][f]) ⇒ m.grilla[a][b][e][f]) ∧
((m.grilla[a][b][c][d] ∧ ¬m.grilla[c][d][e][f]) ⇒ ¬ m.grilla[a][b][e][f])
)

enRango : estr m × nat x × nat y $\longrightarrow$ bool

enRango(m, x, y) $\equiv$ (x < m.tam ∧ y < m.tam)

esContigua : nat x × nat y × nat z × nat w $\longrightarrow$ bool

esContigua(x, y, z, w) $\equiv$ (x = z ∧ y = w+1) ∨
(x = z+1 ∧ y = w) ∨
(**if** z >0 **then** (x = z-1 ∧ y = w) **else** false **fi**) ∨
(**if** w >0 **then** (x = z ∧ y = w-1) **else** false **fi**)

Rep : estr m $\longrightarrow$ bool

Rep(m) $\equiv$ (1) $\wedge_{\text{L}}$ (2) $\wedge_{\text{L}}$ (3) $\wedge$ (4) $\wedge$ (5) $\wedge$ (6)

Abs : estr m $\longrightarrow$ Mapa {Rep(m)}

Abs(m) $\equiv$ map : Mapa / ($\forall c$: coord)(c $\in$ coordenadas(map)) $\iff$
((latitud(c) $<$ m.tam) $\wedge$ (longitud(c) $<$ m.tam) $\wedge_{\text{L}}$
m.grilla[latitud(c)][longitud(c)][latitud(c)][longitud(c)] = true)

Algoritmos

3.3. Algoritmos

iCrearMapa()$\rightarrow$ res : map

1: vector(vector(vector(vector(bool)) $mapa$ $\leftarrow$ Vacio() ▷ Crear vector vacio es O(1) // $O(1)$
2: $res \leftarrow \langle$ 0 , $mapa$ $\rangle$ ▷ $O(1)$

Complejidad: $O(1)$
3: Justificaion: $O(1) + O(1)$

iCoordenadas(**in** m : `map`)$\rightarrow$ res : conj(coor)

```
1:  conj(coor) coors ← Vacio()                          ▷ Crear conjunto vacio es O(1) // O(1)
2:  for i = 0 to m.tam do
3:      for j = 0 to m.tam do
4:          if iPosExistente(i, j) then                  ▷ O(1)
5:              c ← CrearCoor(i, j)                      ▷ O(1)
6:              AgregarRapido(coors, c)                  ▷ O(1)
7:          end if
8:      end for
9:  end for
10: res ← coors                                          ▷ O(1)
```

Complejidad: $O((tam(m))^2)$
Justificacion: Las operaciones interiores son O(1). Hay dos for anidados, donde cada uno se ejecuta $tam(m)$ veces. En total son $(tam(m))^2$ iteraciones.

iPosExistente(**in** m : `map`, **in** c : `coor`)→ res : bool

```
1: bool existe
2: if latitud(c) ≥ m.tam ∨ longitud(c) ≥ m.tam then                    ▷ O(1)
3:     existe ← false                                                   ▷ O(1)
4: else
5:     nat x ← Latitud(c)                                               ▷ O(1)
6:     nat y ← Longitud(c)                                              ▷ O(1)
7:     existe ← m.grilla[x][y][x][y]                                    ▷ O(1)
8: end if
9: res ← existe                                                         ▷ O(1)
```

Complejidad: $O(1)$
Justificacion: Evaluar la guarda del if es O(1) por que son comparaciones de nats. Si es verdadera se produce una asignacion O(1). Si es falsa se ejecutan asignaciones O(1) y acceso directo a vector O(1). O(1) + O(1) + O(1) + O(1) = O(1). Es constante en cualquier caso.

iHayCamino(**in** m : `map`, **in** c_1 : `coor`, **in** c_2 : `coor`)→ res : bool

```
1: res ← m.grilla[Latitud(c1)][Longitud(c1)][Latitud(c2)][Longitud(c2)]   ▷ O(1)
```

Complejidad: $O(1)$
Justificacion: Latitud() y Longitud() son O(1). Accesos directos a vectores es O(1)

iAgregaCoor(**in/out** m : `map`, **in** c : `coor`)

```
1: nat maximo ← max(Latitud(c), Longitud(c))                            ▷ O(1)
2:
3: if maximo > m.tam then                                               ▷ O(1)
4:     vector(vector(vector(vector(bool)))) nGrilla
5:     nGrilla ← iCrearGrilla(maximo)                                   ▷ O(max(Latitud(c), Longitud(c))^4)
6:     iCopiarCoordenadas(nGrilla, m.grilla)                            ▷ O(max(Latitud(c), Longitud(c))^4)
7:     m.grilla ← nGrilla                                               ▷ O(max(Latitud(c), Longitud(c))^4)
8:     m.tam ← maximo                                                   ▷ O(1)
9: end if
10: m.grilla[Latitud(c)][Longitud(c)][Latitud(c)][Longitud(c)] ← true   ▷ O(1)
11:
12: vector(vector(coor))visitados ← Vacio()                             ▷ O(1)
13: for i = 0 to m.tam − 1 do   ▷ m.tam^2 iteraciones, pero m.tam pudo haber cambiado // O(max(Latitud(c), Longitud(c), m.tam)^2)
14:     vector(coor) visitadosAux ← Vacio()                             ▷ O(1)
15:     for j = 0 to m.tam − 1 do                                       ▷ Realiza m.tam iteraciones O(m.tam)
16:         visitadosAux.AgregarAtras(false)                            ▷ O(1)
17:     end for
18:     visitados.AgregarAtras(visitadosAux)                            ▷ O(1)
19: end for
20: cola(coor) aRecorrer ← Vacio()                                      ▷ O(1)
21: aRecorrer.Encolar(c)                                                ▷ O(1)
22:
23: while ¬ EsVacia(aRecorrer) do   ▷ Como maximo se recorren todas las coordenadas del mapa // O(max(Latitud(c), Longitud(c), m.tam))^2)
24:     coor act ← Proximo(aRecorrer)                                   ▷ O(1)
25:     Desencolar(aRecorrer)                                           ▷ O(1)
26:
27:     if Latitud(act) > 0 then                                        ▷ O(1)
28:         nat x ← Latitud(CoordenadaALaIzquierda(act))                ▷ O(1)
29:         nat y ← Longitud(CoordenadaALaIzquierda(act))               ▷ O(1)
30:         if ¬ visitados[x][y] then                                   ▷ O(1)
31:             visitados[x][y]← true                                   ▷ O(1)
```

```
32:             if Existe(coordenadaALaIzquierda(act)) then                          ▷ O(1)
33:                 m.Grilla[Latitud(c)][Longitud(c)][x][y] ← true                   ▷ O(1)
34:                 m.Grilla[x][y][Latitud(c)][Longitud(c)] ← true                   ▷ O(1)
35:                 Encolar(coordenadaALaIzquierda(act), aRecorrer)                  ▷ O(1)
36:             end if
37:         end if
38:     end if
39:
40:     if Longitud(act) > 0 then                                                    ▷ O(1)
41:         nat x ← Latitud(CoordenadaAbajo(act))                                    ▷ O(1)
42:         nat y ← Longitud(CoordenadaAbajo(act))                                   ▷ O(1)
43:         if ¬ visitados[x][y] then                                                ▷ O(1)
44:             visitados[x][y]← true                                                ▷ O(1)
45:             if Existe(CoordenadaAbajo(act)) then                                 ▷ O(1)
46:                 m.Grilla[Latitud(c)][Longitud(c)][x][y] ← true                   ▷ O(1)
47:                 m.Grilla[x][y][Latitud(c)][Longitud(c)] ← true                   ▷ O(1)
48:                 Encolar(CoordenadaAbajo(act), aRecorrer)                         ▷ O(1)
49:             end if
50:         end if
51:     end if
52:
53:     if Latitud(act) < m.Tam − 1 then                                             ▷ O(1)
54:         nat x ← Latitud(CoordenadaALaDerecha(act))                               ▷ O(1)
55:         nat y ← Longitud(CoordenadaALaDerecha(act))                              ▷ O(1)
56:         if ¬ visitados[x][y] then                                                ▷ O(1)
57:             visitados[x][y]← true                                                ▷ O(1)
58:             if Existe(CoordenadaALaDerecha(act)) then                            ▷ O(1)
59:                 m.Grilla[Latitud(c)][Longitud(c)][x][y] ← true                   ▷ O(1)
60:                 m.Grilla[x][y][Latitud(c)][Longitud(c)] ← true                   ▷ O(1)
61:                 Encolar(CoordenadaALaDerecha(act), aRecorrer)                    ▷ O(1)
62:             end if
63:         end if
64:     end if
65:
66:     if Longitud(act) < m.Tam−1 then                                              ▷ O(1)
67:         nat x ← Latitud(CoordenadaArriba(act))                                   ▷ O(1)
68:         nat y ← Longitud(CoordenadaArriba(act))                                  ▷ O(1)
69:         if ¬ visitados[x][y] then                                                ▷ O(1)
70:             visitados[x][y]← true                                                ▷ O(1)
71:             if Existe(CoordenadaArriba(act)) then                                ▷ O(1)
72:                 m.Grilla[Latitud(c)][Longitud(c)][x][y] ← true                   ▷ O(1)
73:                 m.Grilla[x][y][Latitud(c)][Longitud(c)] ← true                   ▷ O(1)
74:                 Encolar(CoordenadaArriba(act), aRecorrer)                        ▷ O(1)
75:             end if
76:         end if
77:     end if
78: end while
```

Complejidad: O(max(Latitud(c), Longitud(c), m.tam))^4)

Justificacion: Trabajo con matrices de 4 dimensiones, en peor caso hay que redimensionar, en donde se crea una nueva grilla con el nuevo tamanio. CrearGrilla tiene complejidad O(n^4), por lo que en este caso es O(max(Latitud(c), Longitud(c))^4). Si se redimensiona, es porque: max(Latitud(c), Longitud(c)) > m.tam. Entonces en ese caso, max(Latitud(c), Longitud(c)) = O(max(Latitud(c), Longitud(c), m.tam))^4).
En peor caso se recorren luego todas las coordenadas de la grilla "principal"(las primeras 2 dimensiones) La grilla pudo haber sido redimensionada, por lo que la complejidad del while es O(max(Latitud(c), Longitud(c), m.tam))^2), donde m.tam es el tamanio original de la grilla. Entonces la complejidad en peor caso es la de mayor exponente por algebra de ordenes: O(max(Latitud(c), Longitud(c), m.tam))^4)

iTam(**in** m : `map`)→ res : nat

1: $res \leftarrow m.tam$ ▷ $O(1)$

Complejidad: $O(1)$

---

iCrearGrilla(**in** n : `nat`)→ res : vector(vector(vector(vector(bool))))

Funcion privada
Pre: n > 0
Post: res es una grilla de tam n

1: vector(vector(vector(vector(bool)))) $nGrilla \leftarrow$ Vacio() ▷ $O(1)$
2: **for** nat $i \leftarrow 0$ to $n-1$ **do** ▷ $O(n^4)$
3: vector(vector(vector(bool))) $nGrilla2 \leftarrow$ Vacio() ▷ $O(1)$
4: **for** nat $j \leftarrow 0$ to $n-1$ **do** ▷ $O(n^3)$
5: vector(vector(bool)) $nGrilla3 \leftarrow$ Vacio() ▷ $O(1)$
6: **for** nat $k \leftarrow 0$ to $n-1$ **do** ▷ $O(n^2)$
7: vector(bool) $nGrilla4 \leftarrow$ Vacio() ▷ $O(1)$
8: **for** nat $l \leftarrow 0$ to $n-1$ **do** ▷ $O(n)$
9: AgregarAtras($nGrilla4$, false) ▷ $O(1)$
10: **end for**
11: AgregarAtras($nGrilla3$, $nGrilla4$)
12: **end for**
13: AgregarAtras($nGrilla2$, $nGrilla3$)
14: **end for**
15: AgregarAtras($nGrilla$, $nGrilla2$)
16: **end for**

Complejidad: $O(n^4)$
Justificacion: Son 4 fors anidados que se ejecutan n veces cada uno. $O(n) * O(n) * O(n) * O(n) = O(n^4)$

---

iCopiarCoordenadas(**in/out** $nGrilla$: `vector(vector(vector(vector(bool))))`, **in** $vGrilla$: `vector(vector(vector(vector(bool))))`)

CopiarCoordenadas : Función privada
Descripción : Asigno las coordenadas existentes de la vieja grilla en la nueva grilla
Pre: longitud($vGrilla$) ≤ longitud($nGrilla$)
Post: $(\forall i, j$: nat$)(i <$longitud($vGrilla$)$\wedge$ $j <$longitud($vGrilla$) $\Rightarrow_{\text{L}}$
nGrilla$[i][j][i][j] =_{\text{obs}}$ vGrilla$[i][j][i][j]$
Complejidad: O(Longitud($vGrilla$)^2)

1: **for** nat $i \leftarrow 0$ to Longitud($vGrilla$) -1 **do** ▷ O(Longitud($vGrilla$)^2)
2: **for** nat $j \leftarrow 0$ to Longitud($vGrilla$) -1 **do** ▷ O(Longitud($vGrilla$))
3: nGrilla$[i][j][i][j] \leftarrow$ vGrilla$[i][j][i][j]$ ▷ $O(1)$
4: **end for**
5: **end for**

Complejidad: ▷ O(Longitud($vGrilla$)^2)
Justificacion: Son 2 fors anidados, donde en cada uno hago Longitud($vGrilla$) iteraciones. O(Longitud($vGrilla$)) * O(Longitud($vGrilla$)) = O(Longitud($vGrilla$)^2)

3.4. Servicios usados

De Vector
- AgregarAtras(vector(α), α) debe ser O(f(long(v)) + copy(a))

- VacÃa() debe ser O(1)

De Conjunto Lineal
- AgregarRapido(conj(α), α) debe ser O(copy(a))

De Coordenada
- CrearCoor(nat, nat) debe ser O(1)
- longitud(coor) debe ser O(1)
- latitud(coor) debe ser O(1)
- CoordenadaArriba(coor) debe ser O(1)
- CoordenadaAbajo(coor) debe ser O(1)
- CoordenadaALaDerecha(coor) debe ser O(1)
- CoordenadaALaIzquierda(coor) debe ser O(1)
- CopiarCoordenadas(vector(vector(vector(vector(bool)))))), vector(vector(vector(vector(bool)))))
debe ser O(Longitud(vGrilla)^2)

De Cola
- Encolar(colaEntr, entrenador) debe ser O(log(EC))
- Proximo(colaEntr) debe ser O(1)
- Desencolar(colaEntr) debe ser O(1)

4. Juego

Interfaz

4.1. Interfaz

se explica con: JUEGO.

géneros: `juego`.

Operaciones básicas de Juego

CREARJUEGO(**in** m : `map`) $\rightarrow$ res : `juego`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} crearJuego(m)$}
Complejidad: $O((tam(m))^2)$
Descripción: Genera una juego con el mapa `m` y sin jugadores.

AGREGARPOKÉMON(**in** p: `pokemon`, **in** c: `coord`, **in/out** j : `juego`)
Pre $\equiv$ {$j_0 =_{\text{obs}} j \wedge$ posExistente(c, mapa(j)) $\wedge\ p \notin$ pokemones(j) $\wedge$ PuedoAgregarPokemon(c, j)}
Post $\equiv$ {$j =_{\text{obs}} agregarPokemon(p, c, j_0) \wedge p \in pokemones(j)$}
Complejidad: $O(J)$
Descripción: Agrega pokémon `p` al juego `j` en la coordenada `c`.

AGREGARJUGADOR(**in** j : `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$j_0 =_{\text{obs}} j$}
Post $\equiv$ {$res =_{\text{obs}}$ ProxId(j_0) $\wedge$ $j =_{\text{obs}}$ agregarJugador(j_0)}
Complejidad: $O(J)$
Descripción: Agrega un jugador al juego `j` con id igual a ProxId(j).

CONECTARSE(**in** e: `jugador`, **in** c: `coor`, **in/out** j : `juego`)
Pre $\equiv$ {$j_0 =_{\text{obs}} j \wedge e \in$ jugadores(j) $\wedge_{\text{L}} \neg$ estaConectado(e, j) $\wedge$ posExistente(c, mapa(j))}
Post $\equiv$ {$j =_{\text{obs}}$ conectarse(e, c, j_0)}
Complejidad: $O(log(EC))$
Descripción: Conecta al jugador e en la posicion c.

DESCONECTARSE(**in** e: `jugador`, **in/out** j : `juego`)
Pre $\equiv$ {$j_0 =_{\text{obs}} j \wedge e \in$ jugadores(j) $\wedge_{\text{L}}$ estaConectado(e, j)}
Post $\equiv$ {$j =_{\text{obs}}$ desconectarse(e, j_0)}
Complejidad: $O(log(EC))$
Descripción: Desconecta al jugador e del juego.

MOVERSE(**in** e: `jugador`, **in** c: `coor`, **in/out** j : `juego`)
Pre $\equiv$ {$j_0 =_{\text{obs}} j \wedge e \in jugadores(j) \wedge_{\text{L}} estaConectado(e, j) \wedge posExistente(c, mapa(j))$}
Post $\equiv$ {$j =_{\text{obs}}$ moverse(e, c, j_0)}
Complejidad: $O((PC + PS) * |P| + log(EC))$
Descripción: Mueve al jugador e en la posicion c si es valido y captura si debe, sino sanciona o expulsa.

MAPA(**in** j : `juego`) $\rightarrow$ res : `map`
Pre $\equiv$ {true}
Post $\equiv$ {$res =$ mapa(j)}
Complejidad: $O(1)$
Descripción: Devuelve el mapa del juego.
Aliasing: Es por referencia, produce aliasing.

JUGADORES(**in** j : `juego`) $\rightarrow$ res : `itConj(jugador)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ jugadores(j)}
Complejidad: $O(1)$

Descripción: Devuelve un iterador a los jugadores del juego
Aliasing: Modificar el conjunto que se devuelve modifica la estructura del juego.

EstaConectado(**in** j : `juego`, **in** e : `jugador`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$e \in$ jugadores(j)}
Post $\equiv$ {$res =$ estaConectado(e, j)}
Complejidad: $O(1)$
Descripción: Devuelve true si el jugador esta conectado.

Sanciones(**in** e : `jugador`, **in** j : `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$e \in$ jugadores(j)}
Post $\equiv$ {$res =$ sanciones(e, j)}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de sanciones de un jugador.

Posicion(**in** j : `juego`, **in** e : `jugador`) $\rightarrow$ res : `coor`
Pre $\equiv$ {$e \in$ jugadores(j) $\wedge_{\text{L}}$ estaConectado(e, j)}
Post $\equiv$ {$res =$ posicion(e, j)}
Complejidad: $O(1)$
Descripción: Devuelve la posicion actual de un jugador.

Pokemons(**in** j : `juego`, **in** e : `jugador`) $\rightarrow$ res : `iterDiccString(nat)`
Pre $\equiv$ {$e \in$ jugadores(j)}
Post $\equiv$ {$res =_{\text{obs}}$ crearIt(j.jugadores[e].pokemons)}
Complejidad: $O(1)$
Descripción: Devuelve un iterador a los pokemons capturados por el jugador.
Aliasing: El iterador se invalida si el conjunto de claves del DiccString (que contiene a los pokemons del jugador) cambia.

Expulsados(**in** j : `juego`) $\rightarrow$ res : `conj(jugador)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =$ expulsados(j)}
Complejidad: $O(J)$
Descripción: Devuelve un conjunto con los jugadores expulsados.

PosConPokemons(**in** j : `juego`) $\rightarrow$ res : `conj(coor)`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{\text{obs}}$ posConPokemons(j))}
Complejidad: $O(1)$
Descripción: Devuelve un conjunto con las posiciones del mapa que tienen pokemons.
Aliasing: El conjunto es devuelto por referencia.

PokemonEnPos(**in** j : `juego`, **in** c : `coor`) $\rightarrow$ res : `pokemon`
Pre $\equiv$ {$c \in$ posConPokemons(j)}
Post $\equiv$ {$res = pokemonEnPos(c, j)$}
Complejidad: $O(1)$
Descripción: Devuelve el pokemon que se encuentra en la posicion c.

CantMovimientosParaCaptura(**in** c : `coor`, **in** j : `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$c \in$ posConPokemons(j)}
Post $\equiv$ {$res =$ cantMovimientosParaCaptura(c, j)}
Complejidad: $O(1)$
Descripción: Devuelve el numero de movimientos que indican cuando se captura un pokemon.

PuedoAgregarPokemon(**in** c : `coor`, **in** j : `juego`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =$ puedoAgregarPokemon(c, j)}
Complejidad: $O(1)$
Descripción: Devuelve verdadero si la coordenada es valida y no hay ningun pokemon en el territorio.

HayPokemonCercano(**in** c : `coor`, **in** j : `juego`) $\rightarrow$ res : `bool`

Pre $\equiv$ {true}
Post $\equiv$ {res = hayPokemonCercano(c, j)}
Complejidad: $O(1)$
Descripción: Devuelve verdadero si hay algun pokemon en el territorio.

PosPokemonCercano(**in** c: `coor`, **in** j: `juego`) $\rightarrow$ res : `coor`
Pre $\equiv$ {hayPokemonCercano(c, j)}
Post $\equiv$ {res = posPokemonCercano(c, j)}
Complejidad: $O(1)$
Descripción: Devuelve la posicion del pokemon que esta en territorio.

EntrenadoresPosibles(**in** c: `coor`, **in** es: `conj(jugador)`, **in** j: `juego`) $\rightarrow$ res : `conj(jugador)`
Pre $\equiv$ {hayPokemonCercano(c, j) $\wedge$ es $\subset$ jugadoresConectados(j)}
Post $\equiv$ {res = entrenadoresPosibles(c, es, j)}
Complejidad: O(Longitud(ec) * EC)
Descripción: De todos los jugadores de la entrada ec, devuelve un conjunto con los entrenadores que estan en condiciones de capturar el pokemon que se encuentra en el rango de c. Que esten en condiciones de capturar significa que estan en rango2 del pokemon y que existe un camino hacia el.

IndiceRareza(**in** p: `pokemon`, **in** j: `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$p \in$ todosLosPokemons(j)}
Post $\equiv$ {$res = indiceRareza(p, j)$}
Complejidad: $O(|P|)$
Descripción: Devuelve el indice de rareza del pokemon dado.

cantPokemonsTotales(**in** j: `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res = cantPokemonsTotales(p, j)}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de pokemons totales del juego.

cantMismaEspecie(**in** p: `pokemon`, **in** j: `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$p \in$ todosLosPokemons(j)}
Post $\equiv$ {res = cantMismaEspecie(p, j)}
Complejidad: $O(|p|)$
Descripción: Devuelve la cantidad de pokemons totales del juego.

Representación

4.2. Representacion

"cantPokemonnos" nos dice, dado un pokemon, la cantidad total de ese pokemon que hay en el juego: salvajes y capturados por jugadores no eliminados.
La cantidad total de pokemons puede obtenerse sumando las cantidades de los pokemons individualmente, pero manteniéndolo de forma separada (cantPokemonsTotales) se puede tener un acceso rápido, útil para calcular el índice de rareza.

Los jugadores están representados por un vector de jugadores: "jugadores", aprovechando que agregar jugador tiene que ser O(J), y usamos que cada id del jugador se corresponde con el índice de su posición en el vector.
Los elementos en este vector de jugadores son tuplas que las llamamos "jugStruc", que contienen todos los datos que son relevantes para un jugador.
Ademas de este vector de jugadores, que contiene los datos de todos los jugadores (estén eliminados o no), tenemos tambien "jugadoresNoEliminados", que es un conjunto de, justamente, los jugadoresNoEliminados. De esta manera podemos devolver en O(1) un iterador a estos jugadores, usando el iterador ya existente de un conjunto, y no tenemos que procuparnos por ir filtrando los eliminados mientras el usuario usa el iterador. Un problema que surge es que cuando se elimina un jugador, necesitamos eliminar el jugador de esta lista (rápidamente). Es por esto que para cada

jugador, mantenemos un iterador a la posicion del jugador en ese conjunto: "iterAJuego".

Los pokemons que capturó estan representados por un diccionario, donde para cada pokemon capturado dice la cantidad capturada de esa especie. "cantCapt" es la cantidad total de pokemones capturados. Puede obtenerse sumando la cantidad capturada de cada pokemon, pero decidimos mantenerla así para poder armar rapidamente la cola de prioridades en la zona de captura.

En caso que esté conectado, podemos comprobar su posición, pero para cumplir complejidades pedidas, muchas veces necesitamos acceder rápidamente a los jugadores desde una posición. Recorrer todos los jugadores y filtrarlos por la coordenada buscada no es eficiente, y por este motivo existe "grillaJugs".

En "grillaJugs", dada una posición nos dá una lista con los jugadores que tienen esa posición. Cuando el jugador deja de estar en esa posición, queremos sacarlo de esa lista, pero puede pasar que en una misma posición haya muchos jugadores, y para eliminar al que queremos no sería barato, porque habria que recorrer la lista. Para solucionar esto, guardamos un iterador a la posición del jugador en esa lista ("iteradorAPos") aprovechando que borrar con este iterador de lista es O(1).

Algo similar sucede con los pokemons salvajes, queremos acceder a ellos por coordenada, para saber por ejemplo, si hay alguno en un rango cercano. Por este motivo los guardamos en "pokenodos", una grilla que los contiene. "pokenodos" es en realidad una grilla de punteros, donde el puntero es NULL si no hay pokemon en esa coordenada.
Si no es NULL, el puntero apunta a un "pokeStruc", que contiene el pokemon salvaje, un contador para saber cuanto falta para la captura, y una cola de entrenadores. Los entrenadores de la cola son todos aquellos jugadores en condiciones de capturar al pokemon. Es una cola de prioridades porque queremos obtener el mínimo de forma eficiente, para poder tener rápido al jugador que queremos actualizar en caso de captura.

Si el jugador elegido para que capture sale del radio, necesitamos el siguiente mínimo de forma eficiente, y esta es la principal razón por la que elegimos tener una cola de prioridad y no una variable el pokeStruc que indique quien es el elegido para capturar.

La situación no es tan feliz si un jugador cualquiera se va del radio, puesto que habria que buscarlo en la cola, y esto rompe las complejidades pedidas. Para resolver esto, en cada jugador (en caso que este en condiciones de capturar) mantenemos un iterador a su posicion correspondiente en esa cola ("itAEntrenadores"). De esta manera, dado un jugador podemos eliminarlo de los entrenadores del pokeStruc de forma eficiente (en tiempo logaritmico, por la forma en que implementamos la cola), cumpliendo las complejidades pedidas.

`Juego` se representa con `pokgo`

donde `pokgo` es `tupla`(*cantPokemon*: `diccString(nat)` ,
cantPokemonsTotales: `nat` ,
map: `mapa` ,
jugadores: `vector(jugStruc)` ,
jugadoresNoEliminados: `conj(jugador)` ,
grillaJugs: `vector(vector(lista(jugador)))` ,
pokenodos: `vector(vector(puntero(pokeStruc)))` ,
posPokemons: `conj(coor)`)

donde `pokeStruc` es `tupla`(*poke*: `pokemon` ,
contador: `nat`,
entrenadores: `colaEntr`)

donde `jugStruc` es `tupla`(*id*: `nat` ,
sanciones: `nat`,
conectado: `bool` ,
pos: `coor` ,
pokemons: `diccString(nat)` ,
iteradorAEntrenadores: `itcolaEntrenador` ,
iteradorAPos: `itLista(nat)` ,
iteradorAJuego: `itConj(jugador)` ,
cantCapt: `nat`)

Invariante de representacion

(0) El indice de la posicion del vector es igual al id del juegador en ese indice (de esto se desprende que los ids son unicos)

($\forall i$: nat)((i < Longitud(j.jugadores)) $\Rightarrow_{\text{L}}$ j.jugadores[i].id = i)

(1) El jugador e esta en j.jugadoresNoEliminados sii no esta eliminado
($\forall e$: jugador)($e \in j$.jugadoresNoEliminados $\iff$ (e <longitud(j.jugadores) $\wedge$ j.jugadores[e].sanciones <5)

(2) Todo jugador de j.jugadores que este conectado, tiene una posicion que es una coordenada existente en el mapa

($\forall jug$: jugStruc) ((esJugadorConectado(jug)) $\Rightarrow_{\text{L}}$ PosExistente(jug.pos, j.mapa))

(3) Dimensiones de la grillaJugs (vector de vectores) es igual al tamaño del mapa

Longitud(j.grillaJugs) = Tam(j.mapa) $\wedge_{\text{L}}$ ($\forall i$: nat) ((i < Logitud(j.grillaJugs)) Longitud(j.grillaJugs[i]) = Tam(j.mapa))

(4) No hay elementos repetidos en las listas de grillaJugs

($\forall x, y$: nat) ((enRango(x, y, j.mapa) $\Rightarrow_{\text{L}}$ sinRepetidos(j.grillaJugs[x][y]))

(5) Todo jugador que esta conectado tiene su id en la lista que se encuentra en grillaJugs para su posicion

($\forall jug$: jugStruc) (esJugadorConectado(jug, j) $\Rightarrow_{\text{L}}$
jug.id $\in$ j.grillaJugs[latitud(jug.pos)][longitud(jug.pos)])

(6) Toda id en toda lista de grillaJugs es un de un jugador del juego que este conectado

($\forall x, y$: nat) (enRango(x, y, j.mapa) $\Rightarrow_{\text{L}}$
($\forall i$: nat) ($i \leq$ Longitud(j.grillaJugs[x][y])
j.jugadores(j.grillaJugs[x][y][i]).conectado

(7) iteradorAPos apunta al elemento correcto (misma id) en la lista de grillaJugs correspondiente a su pos

$(\forall jug : \text{jugStruc})$ (esJugadorConectado(jug, j) $\Rightarrow_{\text{L}}$ Siguiente(jug.iteradorAPos) = jug.id)

(8) cantCapt es consistente con las cantidades de su lista de pokemons capturados

$(\forall jug : \text{jugStruc})$ (esJugadorNoEliminado(jug, j) $\Rightarrow_{\text{L}}$ jug.cantCapt = sumaSignif(jug.pokemons))

(9) cantPokemonTotales es igual a la sumatoria de todos los significados del diccionario
j.cantPokemonTotales = sumaSignif(j.cantPokemon)

(10) Para todo pokemon del diccionario, la cantidad que hay es igual a la suma de los salvajes mas los capturados por jugadores (no eliminados)
$(\forall p : \text{pokemon})$ $((p \in \text{Claves}(j.\text{cantPokemon})) \Rightarrow_{\text{L}}$
Obtener(p, j.cantPokemon) = cantSalvajes(p, j) + sumaPokesCapturados(p, j))

(11) Para todo pokemon salvaje, su cantidad es la resta entre la cantidad total en el diccionario menos los capturados por jugadores (no eliminados)

$(\forall p : \text{pokemon})$ $((p \in \text{Claves}(j.\text{cantPokemon})) \Rightarrow_{\text{L}}$
(cantSalvajes(p, j) = Obtener(p, j.cantPokemon) − sumaPokesCapturados(p, j)))

(12) Dimensiones de pokenodos (vector de vectores) es igual al tamaño del mapa

longitud(j.pokenodos) = tam(j.mapa) $\wedge_{\text{L}}$ $(\forall i\text{: nat})$ $((i <$ Logitud(j.pokenodos)) Longitud(j.pokenodos[i]) = Tam(j.mapa))

(13) Todo pokenodo que tenga un pokestruc, esta en una coordenada valida del mapa y es coherente con j.posPokemons

$(\forall x, y\text{: nat})$ ((enRango(x, y, $j.mapa$) $\wedge_{\text{L}}$ j.pokenodos[x][y] $\neg$ NULL)
$\Rightarrow_{\text{L}}$ (posExistente(crearCoordenada(x, y), j.mapa) $\wedge$ crearCoordenada(x, y) $\in$ j.posPokemons)) $\wedge$
$(\forall c\text{: coor})$ $((c \in j.\text{posPokemons}) \Rightarrow_{\text{L}}$ (enRango(latitud(c), longitud(c), $j.mapa$)
$\wedge_{\text{L}}$ j.pokenodos[latitud(c)][longitud(c)] $\neg$ NULL))

(14) No hay pokenodos con pokestrucs que esten a distancia menor a 5

$(\forall x, y\text{: nat})$ ((enRango(x, y, $j.mapa$) $\wedge_{\text{L}}$ j.pokenodos[x][y] $\neq$ NULL) $\wedge$
$(\forall z, w\text{: nat})$ ((enRango(z, w, $j.mapa$) $\wedge_{\text{L}}$ j.pokenodos[z][w] $\neq$ NULL) $\wedge$
$(x \neq z \wedge y \neq w) \Rightarrow_{\text{L}}$ distEuclidea(crearCoordenada(x, y), crearCoordenada(z, w)) >25)

(15) El contador de todo pokenodo es < 10

$(\forall x, y\text{: nat})$ ((enRango(x, y, $j.mapa$) $\wedge_{\text{L}}$ j.pokenodos[x][y] $\neq$ NULL)
$\Rightarrow_{\text{L}}$ (*(j.pokenodos[x][y])).contador <10)

(16) Todo pokestruc tiene un pokemon que esta bien definido
$(\forall x, y$: nat) ((enRango(x, y, $j.mapa$) $\wedge_{\text{L}}$ j.pokenodos[x][y] $\neq$ NULL)
$\Rightarrow_{\text{L}}$ Def?((*(j.pokenodos[x][y])).poke), j.cantPokemon)

(17) Para todas los pokenodos con pokemons, de todos jugadores validos, conectados, que esten en un radio menor a 2, con un camino a la posicion del pokemon, el que tiene menos cantidad de pokemons capturados (y menor id en caso de empate) se corresponde con el Proximo de la Cola de entrenadores

$(\forall x, y$: nat) ((enRango(x, y, $j.mapa$) $\wedge_{\text{L}}$ j.pokenodos[x][y] $\neq$ NULL)
(minJ(entrenadoresPosibles(crearCoordenada(x, y), jugadoresConectados(j), j) =
(proximo(*(j.pokenodos[x][y]).entrenadores)).id) $\wedge$
((proximo(*(j.pokenodos[x][y]).entrenadores)).cant = j.jugadores[proximo(*(j.pokenodos[x][y]).entrenadores)).id].cantcapt))

(18) Todo jugador valido conectado que tenga un pokemon cercano, si tiene un camino hacia ese pokemon entonces su iterador a entrenadores esta bien definido

$(\forall jug$: jugStruc) (esJugadorConectado(jug, j) $\wedge$ hayPokemonCercano(jug.pos, j) $\Rightarrow_{\text{L}}$
hayCamino(posPokemonCercano($j.pos$, j), j) $\Rightarrow$
siguiente(jug.iterAEntrenadores).id = jug.id $\wedge$ siguiente(jug.iterAEntrenadores).cant = j.cantCapt)

esJugadorConectado : jugStruc jug $\times$ juego j $\longrightarrow$ bool

esJugadorConectado(jug, j) $\equiv$ $jug \in j$.jugadores $\wedge_{\text{L}}$ jug.conectado

esJugadorNoEliminado : jugStruc jug $\times$ juego j $\longrightarrow$ bool

esJugadorNoEliminado(jug, j) $\equiv$ $jug \in j$.jugadores $\wedge_{\text{L}}$ jug.sanciones <5

enRango : nat x $\times$ nat y $\times$ juego j $\longrightarrow$ bool

enRango(x, y, j) $\equiv$ posExistente(crearCoordenada(x, y), j.mapa)

sumaSignif : dicc(string $\times$ nat) $\longrightarrow$ nat

sumaSignif(d) $\equiv$ sumaSignifAux(claves(d), d)

sumaSignifAux : conj(string) $\times$ dicc(string $\times$ nat) $\longrightarrow$ nat

sumaSignifAux(cs, d) $\equiv$ **if** $\emptyset$?(cs) **then** 0 **else** obtener(dameUno(cs), d) + sumaSignifAux(sinUno(cs), d) **fi**

cantSalvajes : pokemon p $\times$ juego j $\longrightarrow$ nat

cantSalvajes(p, j) $\equiv$ #(p, pokemonsSalvajes(posConPokemons(j)))

sumaPokesCapturados : pokemon p × juego j $\longrightarrow$ nat

sumaPokesCapturados(p, j) $\equiv$ sumaPokesCapturadosAux(p, j, jugadoresConectados(j))

sumaPokesCapturadosAux : pokemon p × juego j × conj(jugador) js $\longrightarrow$ nat

sumaPokesCapturadosAux(p, j, js) $\equiv$ **if** $\emptyset$?(js) **then**
0
else
if def?(p, dameUno(js).pokemons) **then**
Obtener(p, dameUno(js).pokemons)
else
0
fi + sumaPokesCapturadosAux(p, j, sinUno(js))
fi

Rep : juego j $\longrightarrow$ bool

Rep(j) $\equiv$ (0) $\wedge_{\text{L}}$ (1) $\wedge_{\text{L}}$ (2) $\wedge$ (3) $\wedge_{\text{L}}$ (4) $\wedge$ (5) $\wedge$ (6) $\wedge_{\text{L}}$ (7) $\wedge$ (8) $\wedge$ (9) $\wedge_{\text{L}}$
(10) $\wedge$ (11) $\wedge$ (12) $\wedge_{\text{L}}$ (13) $\wedge$ (14) $\wedge$ (15) $\wedge$ (16) $\wedge_{\text{L}}$ (17) $\wedge_{\text{L}}$ (18)

Abs : juego j $\longrightarrow$ Juego {Rep(j)}

Abs(j) $\equiv$ jue : Juego /
mapa(jue) $=_{\text{obs}}$ j.map $\wedge$
jugadores(jue) $=_{\text{obs}}$ j.jugadoresNoExpulsados $\wedge$

($\forall e$: jugador) (($e \in$ jugadores(jue)) $\Rightarrow_{\text{L}}$
estaConectado(e, jue) $=_{\text{obs}}$ j.jugadores[e].conectado $\wedge$
sanciones(e, jue) $=_{\text{obs}}$ j.jugadores[e].sanciones $\wedge$
pokemons(e, jue) $=_{\text{obs}}$ j.jugadores[e].pokemons $\wedge$
estaConectado(e, jue) $\Rightarrow$ posicion(e, jue) $=_{\text{obs}}$ j.jugadores[e].pos)) $\wedge$

expulsados(jue) $=_{\text{obs}}$ expulsadosAux(j.jugadores) $\wedge$
posConPokemon(jue) $=_{\text{obs}}$ j.posConPokemon $\wedge$

($\forall c$: coor) (($c \in$ posConPokemon(jue)) $\Rightarrow_{\text{L}}$
pokemonEnPos(c, jue) $=_{\text{obs}}$ ((j.pokenodos[latitud(c)][longitud(c)])$\rightarrow$ poke) $\wedge$
cantMovimientosParaCaptura(c, jue) $=_{\text{obs}}$ ((j.pokenodos[latitud(c)][longitud(c)])$\rightarrow$ contador))

Algoritmos

4.3. Algoritmos

iCrearJuego(**in** *map*: `mapa`) → *res* : juego

1: dictString *cantPokemon* ← Vacio() ▷ $O(1)$
2: nat *cantPokemonsTotales* ← 0 ▷ $O(1)$

3: vector(jugStruc) *jugs* ← Vacio() ▷ $O(1)$
4: conj(jugador) *jugsNoElim* ← Vacio() ▷ $O(1)$
5: conj(coor) *posPokes* ← Vacio() ▷ $O(1)$
6: vector(vector(lista(jugador))) *grillaJugs*

7: **for** $i \leftarrow 0$ to Tam(*map*) $- 1$ **do** ▷ $O((Tam(m))^2)$
8: vector(lista(jugador)) *vectorInterno* ← Vacio() ▷ $O(1)$
9: **for** $j \leftarrow 0$ to Tam(*map*) $- 1$ **do** ▷ $O((Tam(m)))$
10: lista(jugador) *jugsVacia* ← Vacia() ▷ $O(1)$
11: AgregarAtras(*vectorInterno*, *jugsVacia*) ▷ $O(1)$ amortizado
12: **end for**
AgregarAtras(*jugs*, *vectorInterno*) ▷ $O(1)$ amortizado
13: **end for**

14: vector(vector(pokeStruc)) *pokenodos* ← Vacio() ▷ $O(1)$
15: **for** $i \leftarrow 0$ to Tam(*map*) $- 1$ **do** ▷ Se repite Tam(map) veces $O(1)$
16: vector(puntero(pokeStruc)) *vectorInterno* ← Vacio() ▷ $O(1)$
17: **for** $j \leftarrow 0$ to Tam(*map*) $- 1$ **do** ▷ Se repite Tam(map) veces $O(1)$
18: puntero(pokeStruc) *pokePuntero* ← NULL ▷ $O(1)$
19: AgregarAtras(*vectorInterno*, *pokePuntero*) ▷ $O(1)$
20: **end for**AgregarAtras(*pokenodos*, *vectorInterno*) ▷ $O(1)$
21: **end for**
22: *res* ← ⟨*cantPokemon*, *cantPokemonsTotales*, *map*, *jugs*, *jugsNoElim*, *grillaJugs*, *pokenodos*, *posPokes* ⟩ ▷ $O(1)$

Complejidad: $O((Tam(map))^2)$
Justificaion: Se crean 2 vectores de vectores, de Tam(map) elementos tanto el vector interno como el externo $O((Tam(map))^2) + O((Tam(map))^2) = O((Tam(map))^2)$. Se crean varios contenedores vacios que cuestan O(1). El mapa lo pasamos por referencia. La tupla tiene una cantidad constante de elementos. $O(1) + .. + O(1) + O((Tam(map))^2) = O((Tam(map))^2)$

iAgregarJugador(**in** *j* : `juego`) → *res* : nat

1: nat *proxId* ← Longitud(*j*.jugadores) ▷ $O(1)$
2: coor *pos* ← CrearCoor(0,0) ▷ $O(1)$
3: diccString(nat) *pokes* ← Vacio() ▷ $O(1)$
4: itcolaEntrenador *itEntrenadores* ▷ $O(1)$
5: lista(nat) *listaDummy* ← Vacia() ▷ $O(1)$
6: itLista(nat) *iteradorAPos* ← CrearIt(*listaDummy*) ▷ $O(1)$
7: itConj(jugador) *iteradorAJuego* ← AgregarRapido(*proxId*, *j*.jugadoresNoEliminados) ▷ $O(1)$
8: AgregarRapido(*j*.jugadoresNoEliminados, *e*) ▷ $O(1)$
9: AgregarAtras(*j*.jugadores, $<$ *proxId*, 0, *false*, *pos*, *pokes*, *itEntregadores*, *iteradorAPos*, *iteradorAJuego*, 0 $>$) ▷ $O(J)$
10: *res* ← *proxId* ▷ $O(1)$

11: Complejidad: $O(J)$
12: Justificación: En el peor caso, hay que redimensionar el vector y eso cuesta O(cantElementosEnVector) = O(J) pues son todos los jugadores del juego.

iConectarse(**in/out** p: `juego`, **in** c: `coordenada`, **in** e: `jugador`)

1: j.jugadores[j].conectado ← true ▷ $O(1)$
2: j.jugadores[e].iteradorAPos ← it AgregarAtras(j.grillaJugs[latitud(c)][longitud(c)], e) ▷ $O(1)$
3: j.jugadores[e].pos ← c
4: **if** HayPokemonCercano(c, j) **then**
5: **if** HayCamino(c, PosPokemonCercano(c, j), Mapa(j)) **then** ▷ $O(1)$
6: nat $latPok$ ← latitud(PosPokemonCercano(c, j)) ▷ $O(1)$
7: nat $lonPok$ ← longitud(PosPokemonCercano(c, j)) ▷ $O(1)$
8: (j.pokenodos[$latPok$][$lonPok$] → contador) ← 0 ▷ $O(1)$
9: tupla <nat, nat>t ← <e, j.jugadores[e].cantCap>
10: j.jugadores[e].iteradorAEntrenadores ← Encolar((j.pokenodos[$latPok$][$lonPok$] → entrenadores) , t) ▷ $O(log(EC))$
11: **end if**
12: **end if**

13: Complejidad: $O(log(EC))$
14: Justificaion: Todas las operaciones de asignación, acceso a posiciones de vectores y desreferenciación de punteros son O(1). Las funciones "HayPokemonCercano", "HayCamino", "PosPokemonCercano", "Mapa" son O(1). La función AgregarAtras de lista enlazada es O(1). La función de Cola de entrenadores Encolar es $O(log(EC))$ en el peor caso, y como $1 \leq log(EC)$, por álgebra de órdenes, sumando todos los costos, el costo final el algoritmo es $O(log(EC))$, donde EC es la cantidad máxima de jugadores esperando a capturar un pokemon, (la cantidad máxima de elementos de la cola).

iDesconectarse(**in/out** j: `juego`, **in** e: `jugador`)

1: j.jugadores[e].conectado ← false ▷ $O(1)$
2: EliminarSiguiente(j.jugadores[e].iteradorAPos) ▷ $O(1)$
3: c ← j.jugadores[e].pos
4: **if** HayPokemonCercano(c, j) ∧ HayCamino(c, PosPokemonCercano(c, j), Mapa(j)) **then** ▷ $O(1)$
5: nat $latPok$ ← latitud(PosPokemonCercano(c, j)) ▷ $O(1)$
6: nat $lonPok$ ← longitud(PosPokemonCercano(c, j)) ▷ $O(1)$
7: Borrar(j.jugadores[e].iteradorAEntrenadores, (j.pokenodos[latPok][lonPok] → entrenadores)) ▷ $O(log(EC))$
8: **end if**

9: Complejidad: $O(log(EC))$
10: Justificaion: Todas las operaciones de asignación, acceso a posiciones de vectores y desreferenciación de punteros son O(1). Las funciones "HayPokemonCercano", "HayCamino", "PosPokemonCercano", "Mapa" son O(1). La función EliminarSiguiente del iterador de lista es $O(1)$. La función de itcolaEntrenadores Borrar es $O(log(EC))$ en el peor caso, y como $1 \leq log(EC)$, por álgebra de órdenes, sumando todos los costos da que el algoritmo es $O(log(EC))$. Donde EC es la cantidad máxima de jugadores esperando a capturar un pokemon, y por lo tanto, la cantidad máxima de elementos de la cola.

iAgregarPokemon(**in** p : `pokemon`, **in** c : `coord`, **in/out** j : `juego`)

1: j.cantPokemonsTtotales ← j.cantPokemonsTotales + 1 ▷ $O(1)$
2: AgregarRapido(j.posPokemons) ▷ Por precondicion puedoAgregarPokemon $O(1)$
3: **if** Def?(p, j.cantPokemon) **then** ▷ $O(|P|)$
4: Definir(j.cantPokemon, p, Obtener(p, j.cantPokemon) + 1) ▷ $O(|P|)$
5: **end if**
6: **if** ¬ Def?(p, j.cantPokemon) **then** ▷ $O(|P|)$
7: Definir(j.cantPokemon, p, 1) ▷ $O(|P|)$
8: **end if**
9: colaEntr h ← Vacia() ▷ $O(1)$
10: vector<jugador> posiblesCapturadores ← DameJugadoresEnPokerango(j, c) ▷ $O(EC)$
11: nat i ← 0 ▷ $O(1)$
12: **while** $i <$ Longitud($posiblesCapturadores$) **do** ▷ $O(EC * log(EC))$
13: $aInsertar$ ← <posiblesCapturadores[i], j.jugadores[posiblesCapturadores[i]].cantCapt> ▷ $O(1)$
14: itcolaEntrenador it ← Encolar(h, $aInsertar$) ▷ $O(log(EC))$
15: j.jugadores[posiblesCapturadores[i]].iteradorAEntrenadores ← it ▷ $O(1)$
16: i ← i + 1 ▷ $O(1)$
17: **end while**
18: pokeStruc pok ← <p, 0, h> ▷ $O(1)$
19: puntero(pokeStruc) puntPok ← pok ▷ $O(1)$
20: j.pokenodos[latitud(c)][longitud(c)] ← puntPok ▷ $O(1)$

21: Complejidad: $O(EC * log(EC) + |P|)$
22: Justificacion: Las operaciones de asignación, acceso a posiciones y desreferenciación de punteros son $O(1)$. La funciónes Def, Definir y Obtener son $O(|P|)$ donde |P| es la longitud del pokemon más largo. La operación Encolar de la cola de entrenadores es $O(log(EC))$ donde EC es la cantidad máxima de jugadores en un pokenodo, y por ende, la cantidad máxima de elementos de la cola. DameJugadoresEnPokerango es $O(EC)$ y devuelve un vector de EC elementos como máximo. En el while se realiza una iteación por cada elemento de dicho vector, por lo tanto en el peor caso se realizan EC iteraciones, y como dentro del ciclo hay 3 funciones $O(1)$ y una $O(log(EC))$, la cantidad de operaciones que realiza el while hasta terminar, en el peor caso es $O(EC * log(EC))$. Sumando los costos de las operaciones independientes, por álgebra de órdenes queda que el costo del algoritmo es de $O(1) + O(|P|) + O(1) + O(EC) + O(1) + O(EC * log(EC)) + O(1) + O(1) + O(1) = O(|P|) + O(EC) + O(EC * log(EC)) = O(|P|) + O(EC * log(EC))$ (ya que EC ≤ EC*log(EC)) $= O(|P| + EC * log(EC))$

---

iMoverse(**in** e : `jugador`, **in** c : `coord`, **in/out** j : `juego`)

1: **if** ¬MovValido(e, c, j) **then**
2: j.jugadores[e].sanciones ← j.jugadores[e].sanciones + 1 ▷ $O(1)$
3: **if** j.jugadores[e].sanciones ≥ 5 **then**
4: j.jugadores[e].conectado ← false ▷ $O(1)$
5: EliminarSiguiente(j.jugadores[e].iterAPos) ▷ $O(1)$
6: EliminarSiguiente(j.jugadores[e].iterAJuego) ▷ $O(1)$
7: itConj $pokesJug$ ← Claves(j.jugadores[e].pokemons) ▷ $O(1)$
8: **while** HaySiguiente($pokesJug$) **do** ▷ El jugador pudo haber capturado PC pokemons $O(PC * |P|)$
9: pokemon $pokeActual$ ← Siguiente($pokesJug$).poke ▷ $O(1)$
10: nat $cantActual$ ← Obtener(j.jugadores[e].pokemons, $pokeActual$) ▷ $O(|P|)$
11: Definir(j.cantPokemon, $pokeActual$, Obtener($pokeActual$, j.cantPokemon)−$cantActual$) ▷ $O(|P|)$
12: **end while**
13: j.cantPokemonTotales ← j.cantPokemonTotales - j.jugadores[e].cantCapt ▷ $O(1)$
14: **if** HayPokemonCerca(j.jugadores[e].pos) **then** ▷ $O(1)$
15: Borrar(j.jugadores[e].iterAEntrenadores) ▷ $O(log(EC))$
16: **end if**
17: Avanzar($pokesJug$) ▷ $O(1)$
18: **end if**
19: **end if**
20: **if** j.jugadores[e].sanciones < 5 **then**

```
    coor posAntes ← Copiar(j.jugadores[e].pos)                                   ▷ O(1)
    bool hayPokAntes ← HayPokemonCercano(posAntes, j)                            ▷ O(1)
    bool hayPokDesp ← HayPokemonCercano(c, j)                                    ▷ O(1)

    if hayPokDesp then
        if ¬ hayPokAntes then
            CasoMov3(e, posAntes, c, j)                                          ▷ O((PS ∗ |P|) + log(EC))
        else
            if PosPokemonCercano(posAntes, j) = PosPokemonCercano(c, j) then     ▷ O(1)
                CasoMov1(e, posAntes, c, j)                                      ▷ O(PS ∗ |P|)
            else
                CasoMov5(e, posAntes, c, j)                                      ▷ O((PS ∗ |P|) + log(EC))
            end if
        end if
    else
        if hayPokAntes then
            CasoMov2(e, posAntes, c, j)                                          ▷ O((PS ∗ |P|) + log(EC))
        else
            CasoMov4(e, posAntes, c, j)                                          ▷ O(PS ∗ |P|)
        end if
    end if


    Borrar(j.jugadores[e].iterAPos)                                              ▷ O(1)
    j.jugadores[e].iterAPos ← AgregarAtras(j.grillaJugs[Latitud(c)][Longitud(c)]) ▷ O(1)
    j.jugadores[e].pos ← c                                                       ▷ O(1)
end if
```

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

Justificación: Si el movimiento no fue valido, en el peor caso hay que eliminar al jugador. Esto cuesta $O((PC * |P|) + log(EC))$ pues se ejecuta el ciclo. La segunda parte del algoritmo no se ejecuta si el jugador fue eliminado (pues no entra en el if de sanciones menores a 5), sin embargo, en caso que no fue eliminado el peor caso cuesta $O((PS*|P|)+log(EC))$ si tomo la rama con mayor complejidad. Esto nos da que la complejidad total del algoritmo en peor caso, es la "rama" que mayor complejidad tenga, osea $O(max((PC*|P|)+log(EC), (PS*|P|)+log(EC)))$. Por álgebra de órdenes, usando que el máximo es la suma, esa complejidad es igual a $O((PC * |P|) + log(EC) + (PS * |P|) + log(EC)) = O((PC * |P|) + (PS * |P|) + log(EC)) = O((PC + PS) * |P|) + log(EC))$

iCasoMov1(**in** e : `jugador`, **in** ant : `coor` **in** $desp$: `coor`, **in/out** j : `juego`))

CasoMov1 : Función privada

Descripción : Este es el caso en donde el jugador estaba cerca de un pokemon y se movió dentro del mismo radio

Pre: $j = j_0 \wedge$ hayPokemonCercano(ant, j) $\wedge$ hayPokemonCercano($desp$, j) $\wedge_{\text{L}}$ posPokemonCercano(ant, j) = posPokemonCercano($desp$, j)

Post: j_0 = moverse(e, $desp$, j)

Complejidad: $O(PS * |P|)$

Aliasing: Produce aliasing, el juego es modificable

```
1: coor pokePos ← PosPokemonCercano(desp, j)                                     ▷ O(1)
2: itConj iterPos ← CrearIter(j.posPokemons)                                     ▷ O(1)
3: while HaySiguente(iterPos) do                                                 ▷ O(PS ∗ |P|)
4:     nat x ← Latitud(Siguiente(iterPos))                                       ▷ O(1)
5:     nat y ← Longitud(Siguiente(iterPos))                                      ▷ O(1)
6:     if Siguiente(iterPos) ≠ pokepos then
7:         (j.pokenodos[x][y])→ contador ← ((j.pokenodos[x][y])→ contador) +1)   ▷ (O(1))
8:     end if
9:     if j.pokenodos[x][y])→ contador = 10 then
```

```
10:         SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y])→ entrenadores))   ▷ O(|P|)
11:         EliminarSiguiente(iterPos)                                                          ▷ O(1)
12:         j.pokenodos[x][y] ← NULL                                          ▷ Libero la memoria O(1)
13:     else
14:         Avanzar(iterPos)                                                                    ▷ O(1)
15:     end if
16: end while
```

Complejidad: $O((PS * |P|) + log(EC))$
Justificación: Recorro todos los pokemons salvajes: $O(PC)$, y por cada pokemon en peor caso el contador es mayor o igual a 10, por lo que hay que agregar el pokemon al diccionario del jugador: $O(|P|)$. Este ciclo en total cuesta $O(PC * |P|)$. El resto de las operaciones son $O(1)$ y no agregan complejidad.

iCasoMov2(**in** e : `jugador`, **in** ant : `coor` **in** $desp$: `coor`, **in/out** j : `juego`))

CasoMov2 : Función privada
Descripción : Este es el caso en donde el jugador estaba cerca de un pokemon y se movió fuera del radio
Pre: $j = j_0 \wedge$ hayPokemonCercano(ant, j) $\wedge \neg$hayPokemonCercano($desp$, j)
Post: $j_0 =$ moverse(e, $desp$, j)
Complejidad: $O((PS * |P|) + log(EC))$
Aliasing: Produce aliasing, el juego es modificable

```
1:  Eliminar(j.jugadores[e].iterAEntrenadores)                                                  ▷ O(log(EC))
2:  itConj iterPos ← CrearIter(j.posPokemons)                                                   ▷ O(1)
3:  while HaySiguente(iterPos) do                                                               ▷ O(PS * |P|)
4:      nat x ← Latitud(Siguiente(iterPos))                                                     ▷ O(1)
5:      nat y ← Longitud(Siguiente(iterPos))                                                    ▷ O(1)
6:      (j.pokenodos[x][y])→ contador ← ((j.pokenodos[x][y])→ contador) +1)                     ▷ (O(1))
7:      if j.pokenodos[x][y])→ contador = 10 then
8:          SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y])→ entrenadores))  ▷ O(|P|)
9:          EliminarSiguiente(iterPos)                                                          ▷ O(1)
10:         j.pokenodos[x][y] ← NULL                                          ▷ Libero la memoria O(1)
11:     else
12:         Avanzar(iterPos)                                                                    ▷ O(1)
13:     end if
14: end while
```

Complejidad: $O((PS * |P|) + log(EC))$
Justificación: Se elimina al jugador de la cola de entrenadores del pokestruc en el que se encontraba: $O(log(EC))$. Luego recorro todos los pokemons salvajes: $O(PC)$, y por cada pokemon en peor caso el contador es mayor o igual a 10, por lo que hay que agregar el pokemon al diccionario del jugador: $O(|P|)$. Este ciclo en total cuesta $O(PC * |P|)$. El resto de las operaciones son $O(1)$ y no agregan complejidad. La complejidad en peor caso del algoritmo es $O((PS * |P|) + log(EC))$ por algebra de ordenes.

iCasoMov3(**in** e : `jugador`, **in** ant : `coor` **in** $desp$: `coor`, **in/out** j : `juego`))

CasoMov3 : Función privada
Descripción : Este es el caso en donde el jugador no estaba cerca de ninguno y se movió dentro de algun radio de pokemon
Pre: $j = j_0 \wedge \neg$hayPokemonCercano(ant, j) $\wedge$ hayPokemonCercano($desp$, j)
Post: $j_0 =$ moverse(e, $desp$, j)
Complejidad: $O(PS * |P|)$
Aliasing: Produce aliasing, el juego es modificable

```
1: coor pokePos ← PosPokemonCercano(desp, j)                                   ▷ O(1)
2: itConj iterPos ← CrearIter(j.posPokemons)                                   ▷ O(1)
3: j.jugadores[e].iterAEntrenadores ← Encolar(
4:             j.pokenodos[Latitud(pokePos)][Longitud(pokePos)], <e, j.jugadores[e].cantCapt>
5:     )                                                                       ▷ O(log(EC))
6: while HaySiguente(iterPos) do                                               ▷ O(PS * |P|)
7:     nat x ← Latitud(Siguiente(iterPos))                                     ▷ O(1)
8:     nat y ← Longitud(Siguiente(iterPos))                                    ▷ O(1)
9:     if Siguiente(iterPos) ≠ pokepos then
10:        (j.pokenodos[x][y])→ contador ← 0)                                  ▷ (O(1))
11:    end if
12:    if j.pokenodos[x][y])→ contador = 10 then
13:        SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y])→ entrenadores))   ▷ O(|P|)
14:        EliminarSiguiente(iterPos)                                          ▷ O(1)
15:        j.pokenodos[x][y] ← NULL                                            ▷ Libero la memoria O(1)
16:    else
17:        Avanzar(iterPos)                                                    ▷ O(1)
18:    end if
19: end while
```

Complejidad: $O((PS * |P|) + log(EC))$
Justificación: Modifico la cola de entrenadores del pokestruc al cual el jugador se mueve agregando el jugador a la cola: $O(log(EC))$. Recorro todos los pokemons salvajes: $O(PC)$, y por cada pokemon en peor caso el contador es mayor o igual a 10, por lo que hay que agregar el pokemon al diccionario del jugador: $O(|P|)$. Este ciclo en total cuesta $O(PC * |P|)$. El resto de las operaciones son $O(1)$ y no agregan complejidad. La complejidad en peor caso del algoritmo es $O((PS * |P|) + log(EC))$ por algebra de ordenes.

iCasoMov4(**in** e : `jugador`, **in** ant : `coor` **in** $desp$: `coor`, **in/out** j : `juego`))

CasoMov4 : Función privada
Descripción : Este es el caso en donde el jugador estaba lejos de todo pokemon y se movió lejos de todo pokemon
Pre: $j = j_0 \wedge \neg$hayPokemonCercano(ant, j) $\wedge \neg$hayPokemonCercano($desp$, j)
Post: j_0 = moverse(e, $desp$, j)
Complejidad: $O(PS * |P|)$
Aliasing: Produce aliasing, el juego es modificable

```
1: itConj iterPos ← CrearIter(j.posPokemons)                                   ▷ O(1)
2: while HaySiguente(iterPos) do                                               ▷ O(PS * |P|)
3:     nat x ← Latitud(Siguiente(iterPos))                                     ▷ O(1)
4:     nat y ← Longitud(Siguiente(iterPos))                                    ▷ O(1)
5:     (j.pokenodos[x][y])→ contador ← ((j.pokenodos[x][y])→ contador) +1)     ▷ (O(1))
6:     if j.pokenodos[x][y])→ contador = 10 then
7:         SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y])→ entrenadores))   ▷ O(|P|)
8:         EliminarSiguiente(iterPos)                                          ▷ O(1)
9:         j.pokenodos[x][y] ← NULL                                            ▷ Libero la memoria O(1)
10:    else
11:        Avanzar(iterPos)                                                    ▷ O(1)
12:    end if
13: end while
```

Complejidad: $O((PS * |P|))$
Justificación: Recorro todos los pokemons salvajes: $O(PC)$, y por cada pokemon en peor caso el contador es mayor o igual a 10, por lo que hay que agregar el pokemon al diccionario del jugador: $O(|P|)$. Este ciclo en total cuesta $O(PC * |P|)$. El resto de las operaciones son $O(1)$ y no agregan complejidad. La complejidad en peor caso del

algoritmo es $O((PS * |P|) + log(EC))$ por algebra de ordenes.

iCasoMov5(**in** e : `jugador`, **in** ant : `coor` **in** $desp$: `coor`, **in/out** j : `juego`))

CasoMov5 : Función privada
Descripción : Este es el caso en donde el jugador estaba cerca de un pokemon y se movió hacias las cercanias de otro pokemon distinto
Pre: $j = j_0 \wedge$ hayPokemonCercano(ant, j) $\wedge$ $\neg$hayPokemonCercano($desp$, j) $\wedge_{\text{L}}$ posPokemonCercano(ant, j) $\neq$ posPokemonCercano($desp$, j)
Post: j_0 = moverse(e, $desp$, j)
Complejidad: $O((PS * |P|) + log(EC))$
Aliasing: Produce aliasing, el juego es modificable

```
1: Eliminar(j.jugadores[e].iterAEntrenadores)                                   ▷ O(log(EC))
2: coor pokePos ← PosPokemonCercano(desp, j)                                    ▷ O(1)
3: j.jugadores[e].iterAEntrenadores ← Encolar(
4:             j.pokenodos[Latitud(pokePos)][Longitud(pokePos)], <e, j.jugadores[e].cantCapt>
5:     )                                                                        ▷ O(log(EC))
6: itConj iterPos ← CrearIter(j.posPokemons)                                    ▷ O(1)
7: while HaySiguente(iterPos) do                                                ▷ O(PS * |P|)
8:     nat x ← Latitud(Siguiente(iterPos))                                      ▷ O(1)
9:     nat y ← Longitud(Siguiente(iterPos))                                     ▷ O(1)
10:    if Siguiente(iterPos) ≠ pokepos then
11:        (j.pokenodos[x][y])→ contador ← 0)                                   ▷ (O(1))
12:    end if
13:    (j.pokenodos[x][y])→ contador ← ((j.pokenodos[x][y])→ contador) +1)      ▷ (O(1))
14:    if j.pokenodos[x][y])→ contador = 10 then
15:        SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y])→ entrenadores))   ▷ O(|P|)
16:        EliminarSiguiente(iterPos)                                           ▷ O(1)
17:        j.pokenodos[x][y] ← NULL                                             ▷ Libero la memoria O(1)
18:    else
19:        Avanzar(iterPos)                                                     ▷ O(1)
20:    end if
21: end while
```

Complejidad: $O((PS * |P|) + log(EC))$
Justificación: Se elimina al jugador de la cola de entrenadores del pokestruc en el que se encontraba: $O(log(EC))$. Inserto el jugador en la ola de entrenadores del nuevo pokenodo: $O(log(EC))$. En total hasta ahora cuesta $2 * O(log(EC)) = O(log(EC))$. Luego recorro todos los pokemons salvajes: $O(PC)$, y por cada pokemon en peor caso el contador es mayor o igual a 10, por lo que hay que agregar el pokemon al diccionario del jugador: $O(|P|)$. Este ciclo en total cuesta $O(PC * |P|)$. El resto de las operaciones son $O(1)$ y no agregan complejidad. La complejidad en peor caso del algoritmo, teniendo en cuenta la complejidad acumulada antes de ciclo, es $O((PS*|P|)+log(EC))$ por algebra de ordenes.

---

iCantPokemonTotales(**in** j : `juego`) → res : nat

```
1: res ← j.cantPokemonsTotales                                                  ▷ O(1)
```

2: Complejidad: $O(1)$

---

iIndiceRareza(**in** p : `pokemon`, **in** j : `juego`) → res : nat

1: nat *pokecant* ← Obtener(j.cantPokemon, p) ▷ $O(|P|)$
2: res ← 100 − (100 ∗ *pokecant* / j.cantPokemonsTotales) ▷ $O(1)$
3: Complejidad: $O(|P|)$
4: Justificacion: j.cantPokemon es un dicciString. La complejidad de buscar (y obtener el significado) en peor caso es la longitud de la string mas larga entre sus claves, eso es O($|P|$). j.cantPokemonsTotales es un dato guardado en la estructura del juego, y se accede en O(1). El resto son una resta, multiplicacion y division, que tambien son O(1). O($|P|$) + O(1) = O($|P|$)

iPosConPokemons(**in** j : `juego`) → res : conj(coor)

1: res ← j.posPokemons ▷ Por referencia $O(1)$

Complejidad: $O(1)$

iPokemonEnPos(**in** j : `juego`, **in** c : `coor`) → res : pokemon

1: res ← ((j.pokenodos[Latitud(c)][Longitud(c)]) → poke) ▷ $O(1)$

Complejidad: $O(1)$

iPosicion(**in** j : `juego`, **in** e : `jugador`) → res : coor

1: res ← j.jugadores[e].pos ▷ $O(1)$
2: Complejidad: $O(1)$
3: Justificacion: Todas las operaciones son $O(1)$

iSanciones(**in** e : `jugador`, **in** j : `juego`) → res : nat

1: res ← j.jugadores[e].sanciones ▷ $O(1)$
2: Complejidad: $O(1)$
3: Justificacion: Todas las operaciones son $O(1)$

iEntrenadoresPosibles(**in** c : `coor`, **in** $aRevisar$: `conj(jugador)`, **in** j : `juego`) → res : conj(jugador)

1: coor *pokeCoor* ← PosPokemonCercano(c, j) ▷ $O(1)$
2: puntero(pokeStruc) *pokePuntero* ← $j.pokeNodos[longitud(pokeCoor)][latitud(pokeCoor)]$ ▷ $O(1)$
3: itConj(jugador) *itPosibles* ← CrearIt($aRevisar$) ▷ $O(1)$
4: **while** HaySiguiente(*itPosibles*) **do** ▷ O($Longitud(aRevisar) * EC$)
5: vector(jugador) *jugsEnNodo* ← CrearIt(DameJugadoresEnPokerango(c, j)) ▷ $O(EC)$
6: **for** nat i ← 0 to Longitud(*jugsEnNodo* − 1) **do**
7: **if** Siguiente(*itPosibles*) = *itJugsEnNodo*[i] **then** ▷ $O(1)$
8: Agregar(res, Siguiente(*itPosible*))
9: **end if**
10: Avanzar(*itJugsEnNodo*))
11: **end for**
12: Avanzar(*itPosible*) ▷ ()
13: **end while**
14: Complejidad: O(Longitud($aRevisar$) * EC)
15: Justificacion: SOMEDAY

iHayPokemonCercano(**in** c: `coor`, **in** j: `juego`) → res : coor

```
1:  nat x ← latitud(c)                                                    ▷ O(1)
2:  nat y ← longitud(c)                                                   ▷ O(1)
3:  bool hayPokemon ← false                                               ▷ O(1)
4:  if j.pokenodos[x][y] ≠ NULL then                                      ▷ O(1)
5:      hayPokemon ← true
6:  end if
7:  if x > 0 then
8:      if j.pokenodos[x − 1][y] ≠ NULL then                              ▷ O(1)
9:          hayPokemon ← true
10:     end if
11:     if y > 0 then
12:         if j.pokenodos[x − 1][y − 1] ≠ NULL then                      ▷ O(1)
13:             hayPokemon ← true
14:         end if
15:     end if
16:     if y < tam(m) − 1 then
17:         if j.pokenodos[x − 1][y + 1] ≠ NULL then                      ▷ O(1)
18:             hayPokemon ← true
19:         end if
20:     end if
21:     if x − 1 > 0 then
22:         if j.pokenodos[x − 2][y] ≠ NULL then                          ▷ O(1)
23:             hayPokemon ← true
24:         end if
25:     end if
26: end if
27: if y > 0 then
28:     if j.pokenodos[x][y − 1] ≠ NULL then                              ▷ O(1)
29:         hayPokemon ← true
30:     end if
31:     if y − 1 > 0 then
32:         if j.pokenodos[x][y − 2] ≠ NULL then                          ▷ O(1)
33:             hayPokemon ← true
34:         end if
35:     end if
36: end if
37: if y < tam(m) − 1 then
38:     if j.pokenodos[x][y + 1] ≠ NULL then                              ▷ O(1)
39:         hayPokemon ← true
40:     end if
41:     if tam(m) > 1 ∧ y < tam(m) − 2 then
42:         if j.pokenodos[x][y + 2] ≠ NULL then                          ▷ O(1)
43:             hayPokemon ← true
44:         end if
45:     end if
46: end if
47: if x < tam(m) − 1 then
48:     if j.pokenodos[x + 1][y] ≠ NULL then                              ▷ O(1)
49:         hayPokemon ← true
50:     end if
51:     if y > 0 then
52:         if j.pokenodos[x + 1][y − 1] ≠ NULL then                      ▷ O(1)
53:             hayPokemon ← true
54:         end if
```

```
    end if
    if y < tam(m) − 1 then
        if j.pokenodos[x + 1][y + 1] ≠ NULL then                    ▷ O(1)
            hayPokemon ← true
        end if
    end if
end if
if tam(m) > 1 ∧ x < tam(m) − 2 then
    if j.pokenodos[x + 2][y] ≠ NULL then                            ▷ O(1)
        hayPokemon ← true
    end if
end if
res ← hayPokemon
```

Complejidad: $O(1)$
Justificacion: Reviso 13 posiciones O(1)

iPosPokemonCercano(**in** c: `coor`, **in** j: `juego`) → res : coor

```
nat x ← latitud(c)                                                  ▷ O(1)
nat y ← longitud(c)                                                 ▷ O(1)
coor coorConPokemon                                                 ▷ O(1)
if j.pokenodos[x][y] ≠ NULL then                                    ▷ O(1)
    coorConPokemon ← CrearCoor(x, y)                                ▷ O(1)
end if
if x > 0 then
    if j.pokenodos[x − 1][y] ≠ NULL then                            ▷ O(1)
        coorConPokemon ← CrearCoor(x − 1, y)                        ▷ O(1)
    end if
    if y > 0 then
        if j.pokenodos[x − 1][y − 1] ≠ NULL then                    ▷ O(1)
            coorConPokemon ← CrearCoor(x − 1, y − 1)                ▷ O(1)
        end if
    end if
    if y < tam(m) − 1 then
        if j.pokenodos[x − 1][y + 1] ≠ NULL then                    ▷ O(1)
            coorConPokemon ← CrearCoor(x − 1, y + 1)                ▷ O(1)
        end if
    end if
    if x − 1 > 0 then
        if j.pokenodos[x − 2][y] ≠ NULL then                        ▷ O(1)
            coorConPokemon ← CrearCoor(x − 1, y)                    ▷ O(1)
        end if
    end if
end if
if y > 0 then
    if j.pokenodos[x][y − 1] ≠ NULL then                            ▷ O(1)
        coorConPokemon ← CrearCoor(x, y − 1)                        ▷ O(1)
    end if
    if y − 1 > 0 then
        if j.pokenodos[x][y − 2] ≠ NULL then                        ▷ O(1)
            coorConPokemon ← CrearCoor(x, y − 2)                    ▷ O(1)
        end if
    end if
end if
if y < tam(m) − 1 then
```

```
38:     if j.pokenodos[x][y + 1] ≠ NULL then                                ▷ O(1)
39:         coorConPokemon ← CrearCoor(x, y + 1)                            ▷ O(1)
40:     end if
41:     if tam(m) > 1 ∧ y < tam(m) − 2 then
42:         if j.pokenodos[x][y + 2] ≠ NULL then                            ▷ O(1)
43:             coorConPokemon ← CrearCoor(x, y + 2)                        ▷ O(1)
44:         end if
45:     end if
46: end if
47: if x < tam(m) − 1 then
48:     if j.pokenodos[x + 1][y] ≠ NULL then                                ▷ O(1)
49:         coorConPokemon ← CrearCoor(x + 1, y)                            ▷ O(1)
50:     end if
51:     if y > 0 then
52:         if j.pokenodos[x + 1][y − 1] ≠ NULL then                        ▷ O(1)
53:             coorConPokemon ← CrearCoor(x + 1, y − 1)                    ▷ O(1)
54:         end if
55:     end if
56:     if y < tam(m) − 1 then
57:         if j.pokenodos[x + 1][y + 1] ≠ NULL then                        ▷ O(1)
58:             coorConPokemon ← CrearCoor(x + 1, y + 1)                    ▷ O(1)
59:         end if
60:     end if
61: end if
62: if tam(m) > 1 ∧ x < tam(m) − 2 then
63:     if j.pokenodos[x + 2][y] ≠ NULL then                                ▷ O(1)
64:         coorConPokemon ← CrearCoor(x + 2, y)                            ▷ O(1)
65:     end if
66: end if
67: res ← coorConPokemon

68: Complejidad: O(1)
69: Justificacion: Reviso 13 posiciones O(1)
```

---

iPuedoAgregarPokemon(**in** c: `coor`, **in** j: `juego`) → res : bool

```
1: bool puedo ← false                                                       ▷ O(1)
2: if PosExistente(c, j.mapa) then                                          ▷ O(1)
3:     if ¬ HayPokemonCercano(c, j) then                                    ▷ O(1)
4:         puedo ← true                                                     ▷ O(1)
5:     end if
6: end if
7: res ← puedo                                                              ▷ O(1)

8: Complejidad: O(1)
9: Justificacion: Todas las operaciones son O(1)
```

---

iCantMovimientosParaCaptura(**in** c: `coor`, **in** j: `juego`) → res : nat

```
1: puntero(pokeStruc) pokenodo ← j.pokenodos[latitud(c)][longitud(c)]       ▷ O(1)
2: res ← (*pokenodo).contador                                               ▷ O(1)

3: Complejidad: O(1)
4: Justificacion: Todas las operaciones son O(1)
```

---

iExpulsados(**in** j : `juego`) → res : conj(jugador)

1: **for** $nati \leftarrow 0$ to Longitud(j.jugadores) − 1 **do** ▷ $O(J)$
2: **if** j.jugadores[i].sanciones ≥ 5 **then** ▷ $O(1)$ AgregarRapido(res, j.jugadores[i].id) ▷ $O(1)$
3: **end if**
4: **end for**

5: Complejidad: $O(J)$
6: Justificacion: Aplico operaciones que son O(1) la cantidad de veces que ejecuto el ciclo. El ciclo se ejecuta J veces (porque j.jugadores tiene todos los jugadores que fueron agregados) Entonces es O(J), siendo J la cantidad de jugadores que fueron agregados.

---

iPokemons(**in** e : `jugador`, **in** j : `juego`) → res : iterDiccString(nat)

1: $res \leftarrow$ CrearIt(j.jugadores[e].pokemons) ▷ $O(1)$

2: Complejidad: $O(1)$
3: Justificacion: Devuelvo un iterador en O(1)

---

iCantMismaEspecie(**in** p : `pkemon`, **in** j : `juego`) → res : nat

1: $res \leftarrow j$.cantPokemon.Obtener(p) ▷ $O(|p|)$

Complejidad: $O(|p|)$
Justificaion: La unica operacion es $O(|p|)$

---

iExpulsados(**in** j : `juego`) → res : conj(jugador)

1: nat $i \leftarrow 0$ ▷ $O(1)$
2: conj(jugador) $js \leftarrow$ Vacío() ▷ $O(1)$
3: **while** $i <$ Longitud(j.jugadores) − 1 **do** ▷ se repite Longitud(j.jugadores) veces $O(1)$
4: **if** j.jugadores[i].sanciones ≥ 5 **then** ▷ $O(1)$
5: AgregarRapido(js, j.jugadores[i].id) ▷ $O(1)$
6: **end if**
7: **end while**
8: $res \leftarrow js$ ▷ $O(1)$

Complejidad: $O(J)$
Justificaion: Siendo J la cantidad total de jugadores, se los recorre todos buscando los expulsados.

---

iMovValido(**in** e : `jugador`, **in** c : `coor`, **in** j : `juego`) → res : bool

MovValido : Función privada
Descripción : Dice si el movimiento entre la posicion del jugador y la nueva coordenada c, es valido
Pre: $e \in$ jugadoresConctados(j) ∧ posExistente(c, j.mapa)
Post: res = hayCamino(c, j.jugadores[e].pos, j.mapa) ∧
DistEuclidea(c, j.jugadores[e].pos) ≤ 100
Complejidad: $O(1)$

1: bool *camino* ← HayCamino(c, j.jugadores[e].pos, j.mapa) ▷ $O(1)$
2: bool *distancia* ← (DistEuclidea(c, j.jugadores[e].pos) ≤ 100) ▷ $O(1)$
3: $res \leftarrow camino \wedge distancia$ ▷ $O(1)$

Complejidad: $O(1)$
Justificaion: Todas las operaciones usadas son O(1): O(1) + O(1) + O(1) = O(1)

iSumarUnoEnJug(**in** e : `jugador`, **in** c : `coor`, **in/out** j : `juego`)

SumarUnoEnJug : Función privada
Descripción : Suma uno a la cantidad de pokemons p que tiene el jugador
Pre: $j = j_0 \wedge e \in$ jugadores(j)
Post: #pokemons(p, j_0.jugadores[e].pokemons) = #pokemons(p, j.jugadores[e].pokemons) + 1
Complejidad: $O(|P|)$

```
1: if Def?(p, j.cantPokemon) then                                   ▷ O(|P|)
2:     Definir(j.cantPokemon, p, Obtener(p, j.cantPokemon) + 1)     ▷ O(|P|)
3: else
4:     Definir(j.cantPokemon, p, 1)                                 ▷ O(|P|)
5: end if
```

Complejidad: $O(|P|)$
Justificaion: Se ejecuta la guarda en $O(|P|)$ y en cualquier rama hay una operacion $O(|P|)$. $O(|P|) + O(|P|) = O(|P|)$

iDameJugadoresEnPokerango(**in** c : `coor`, **in** j : `juego`) $\rightarrow$ res : vector(jugador)

DameJugadoresEnPokerango : Función privada
Descripción : Devuelve un vector con todos los jugadores que pueden esperar captura que estan en el rango de c
Pre: posValida(c, j.mapa)
Post: $res =_{\text{obs}}$ entrenadoresPosibles(c, jugadores(j), j)
Complejidad: $O(EC)$
Aliasing: Devuelve el vector por referencia

```
1: vector(jugador) jugsRadio ← Vacio()                                          ▷ O(1)
2: nat x ← latitud(c)                                                           ▷ O(1)
3: nat y ← longitud(c)                                                          ▷ O(1)
4: coor coorConPokemon                                                          ▷ O(1)
5: if j.pokenodos[x][y] ≠ NULL then                                             ▷ O(1)
6:     AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x, y), c, j)    ▷ O(EC)
7: end if
8: if x > 0 then
9:     if j.pokenodos[x − 1][y] ≠ NULL then                                     ▷ O(1)
10:        AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x − 1, y), c, j)   ▷ O(EC)
11:    end if
12:    if y > 0 then
13:        if j.pokenodos[x − 1][y − 1] ≠ NULL then                             ▷ O(1)
14:            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x − 1, y − 1), c, j)   ▷ O(EC)
15:        end if
16:    end if
17:    if y < tam(m) − 1 then
18:        if j.pokenodos[x − 1][y + 1] ≠ NULL then                             ▷ O(1)
19:            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x − 1, y + 1), c, j)   ▷ O(EC)
20:        end if
21:    end if
22:    if x − 1 > 0 then
23:        if j.pokenodos[x − 2][y] ≠ NULL then                                 ▷ O(1)
24:            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x − 2, y), c, j)   ▷ O(EC)
25:        end if
26:    end if
27: end if
28: if y > 0 then
29:    if j.pokenodos[x][y − 1] ≠ NULL then                                     ▷ O(1)
30:        AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x, y − 1), c, j)   ▷ O(EC)
```

```
    end if
    if y − 1 > 0 then
        if j.pokenodos[x][y − 2] ≠ NULL then                                   ▷ O(1)
            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x, y − 2), c, j)   ▷ O(EC)
        end if
    end if
end if
if y < tam(m) − 1 then
    if j.pokenodos[x][y + 1] ≠ NULL then                                       ▷ O(1)
        AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x, y + 1), c, j)   ▷ O(EC)
    end if
    if tam(m) > 1 ∧ y < tam(m) − 2 then
        if j.pokenodos[x][y + 2] ≠ NULL then                                   ▷ O(1)
            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x, y + 2), c, j)   ▷ O(EC)
        end if
    end if
end if
if x < tam(m) − 1 then
    if j.pokenodos[x + 1][y] ≠ NULL then                                       ▷ O(1)
        AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x + 1, y), c, j)   ▷ O(EC)
    end if
    if y > 0 then
        if j.pokenodos[x + 1][y − 1] ≠ NULL then                               ▷ O(1)
            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x + 1, y − 1), c, j)   ▷ O(EC)
        end if
    end if
    if y < tam(m) − 1 then
        if j.pokenodos[x + 1][y + 1] ≠ NULL then                               ▷ O(1)
            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x + 1, y + 1), c, j)   ▷ O(EC)
        end if
    end if
end if
if tam(m) > 1 ∧ x < tam(m) − 2 then
    if j.pokenodos[x + 2][y] ≠ NULL then                                       ▷ O(1)
        AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x + 21, y), c, j)   ▷ O(EC)
    end if
end if
res ← jugsRadio                                                    ▷ Por referencia O(1)
```

Complejidad: $O(EC)$

Justificacion: En peor caso agrego jugadores de 13 posiciones, donde ese agregar me cuesta O(EC) cada vez. $O(EC) + O(EC) + ... + O(EC) = O(EC)$

iAgregarAtrasJugsQueEstanEnPos(**in/out** $jugs$: `vector(jugador)`, **in** $posJug$: `coor`, **in** $posPoke$: `coor`, **in** j: `juego`)

AgregarAtrasJugsQueEstanEnPos : Función privada
Descripción : Agrega los jugadores en condiciones de capturar que se encuentran en esa posicion, al vector pasado por parametros,
Pre: $jugs = jugs_0 \wedge$ posValida($posJug$, j.mapa) $\wedge$ posValida($posPoke$, j.mapa)
Post: $jugs = jugs_0 conlosjugadoresencondicionesdecapturaragregadosatras$
Complejidad: $O(EC)$

```
1: lista(jugador) jugsEnPos ← j.grillaJugs[Latitud(posJug)][Longitud(posJug)]   ▷ Por referencia O(1)
2: itLista iterJug ← CreatIt(jugsEnPos)                                          ▷ O(1)
3: while HaySiguiente(iterJug) do                                                ▷ O(longitud(jugsEnPos))
4:     nat e ← Siguiente(iterJug)                                                ▷ O(1)
5:     if Conectado(e) ∧ HayCamino(Posicion(e), posPoke) then
6:         AgregarAtras(jugs, e)                                                 ▷ O(1)amortizado
7:     end if
8:     Avanzar(iterJug)                                                          ▷ O(1)
9: end while
```

Complejidad: $O(EC)$
Justificaion: En el peor caso todos los jugadores que estan en la posición dada son todos los que tienen que esperar la captura. Como longitud($jugsEnPos$) $\leq$ EC, O(longitud($jugsEnPos$)) = O(EC)

4.4. Servicios usados

De Mapa
- iTam(map) debe ser O(1)
- HayCamino(coord, coord, map) debe ser O(1)
- PosExistente(coord, map) debe ser O(1)

De Coordenada
- longitud(coor) debe ser O(1)
- distEuclidea(coor, coor) debe ser O(1)

De Lista enlazada
- CrearIt(lista) debe ser O(1)
- EliminarSiguiente(itLista(α)) debe ser O(1)
- Avanzar(itLista(α)) debe ser O(1)

De Vector
- AgregarAtras(vector(α), α α) debe ser O(f(long(v)) + copy(a))

De Cola
- Encolar(colaEntr, entrenador) debe ser O(log(EC))

De Diccstring
iVacio() debe ser O(1) - Borrar(string, diccString(α)) debe ser O(|P|)
- Def?(string, diccString(α)) debe ser O(|P|)
- Definir(diccString(α), string, α α) debe ser O(|P|)
- Obtener(string, diccString(α)) debe ser α O(|P|)

De Conjunto Lineal
- AgregarRapido(conj(α), α α) debe ser O(copy(a))
- EliminarSiguiente(itConj(α)) debe ser O(1)
- HaySiguiente(itConjAcotado) debe ser O(1)

- Avanzar(itConj(α)) debe ser O(1)
- Siguiente(tConj(α)) debe ser O(1)

5. DiccString(α)

Interfaz

5.1. Interfaz

parámetros formales
géneros α

se explica con: DICC(STRING, α), CONJ(STRING).

géneros: `diccString(`α`)`.

VACIO() $\rightarrow$ res : `diccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacío}
Complejidad: $O(1)$
Descripción: genera un diccionario vacío.

DEFINIR(**in/out** d: `diccString(`α`)`, **in** s: `string`, **in** a: α)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(p, a, d)}
Complejidad: $O(|P|)$ siendo P la clave mas larga.
Descripción: define a en d con la clave s.
Aliasing: el elemento a se define por referencia.

DEF?(**in** p: `string`, **in** d: `diccString(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(p, d)}
Complejidad: $O(|P|)$ siendo P la clave mas larga.
Descripción: devuelve `true` si y sólo si la p tiene una definicion en d.

OBTENER(**in** p: `string`, **in** d: `diccString(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {def?(s, d)}
Post $\equiv$ {alias($res =_{\text{obs}}$ obtener(p, d))}
Complejidad: $O(|P|)$ siendo P la clave mas larga.
Descripción: devuelve el significado de la la clave p en d.
Aliasing: res es modificable si y sólo si d es modificable.

BORRAR(**in** p: `string`, **in/out** d: `diccString(`α`)`)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge$ def?(s, d)}
Post $\equiv$ {$d =_{\text{obs}}$ borrar(p, d_0)}
Complejidad: $O(|P|)$ siendo P la clave mas larga.
Descripción: borra la clave p y su significado.

CLAVES(**in** d: `diccString(`α`)`) $\rightarrow$ res : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{\text{obs}}$ claves(d))}
Complejidad: $O(1)$
Descripción: Devuelve del conjunto de claves de d. Aliasing: res se modifica sii claves(d) se modifica

Representación

5.2. Representacion

Representación del diccString

En este módulo usamos un trie para definir un diccionario cuyas claves son string.
La idea es que la complejidad de definir y obtener no dependa de la cantidad de claves, si no de la longitud de la clave. Bajando asi la complejidad en peor caso.

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

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

donde `nodo` es `tupla(`*definicion*: `puntero(`α`)`, *siguientes*: `arreglo[256](puntero(nodo))`, *itClave*: `puntero(itConj(string))`$)$

Invariante de representacion en castellano

- Si raiz es $NULL$ entonces el conjunto de claves es vacío.
- Si la raiz no es $NULL$ entonces el conjunto de claves es no vacío.
- Todos los elementos del conjunto de claves estan definidos en el diccString.
- Ningun nodo tiene entre sus siguientes a un nodo que esta "antes" que él

Abs : estr d $\longrightarrow$ dicc(string, α) {Rep(d)}

Abs(d) $\equiv$ dic : dicc(string, α) /
claves(dic) $=_{\text{obs}}$ d.claves $\wedge$

($\forall s$: string) ((def?(s, dic) $=_{\text{obs}}$ $s \in d$.claves) $\wedge_{\text{L}}$
(def?(s, dic) $\Rightarrow_{\text{L}}$ obtener(s, dic) $=_{\text{obs}}$ significado de s en la estructura))

Algoritmos

5.3. Algoritmos

---

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

1: puntero(nodo) $iRiaz \leftarrow NULL$ ▷ $O(1)$
2: conj($string$) $iClaves \leftarrow$ Vacio() ▷ $O(1)$
3: $res \leftarrow \langle\ iRaiz, iClaves\ \rangle$ ▷ $O(1)$
4: Complejidad: $O(1)$
5: Justificaion: $O(1) + O(1) + O(1)$

---

iClaves(**in** d: `diccString(`α`)`) $\rightarrow$ res : conj(α)

1: $res \leftarrow d.claves$ ▷ $O(1)$
2: Complejidad: $O(1)$

---

iObtener(**in** p: `string`, **in** d: `diccString(`α`)`) $\rightarrow$ res : α

```
1: puntero(nodo) n ← raiz                                                    ▷ O(1)
2: nat i ← 0                                                                 ▷ O(1)
3: while i < Longitud(p) do                                                  ▷ Se repite |p| O(1)
4:     actual ← (*actual).siguientes[ord(p[i])]                              ▷ O(1)
5:     i ← i + 1                                                             ▷ O(1)
6: end while
7: res ← (*actual).definicion                                                ▷ O(1)
```

8: Complejidad: $O(|P|)$
9: Justificaion: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(|p|) = O(|P|)$

iDefinir(**in/out** d: `diccString(`α`)`, **in** p: `string`, **in** a: α)

```
1: Puntero(nodo) actual ← raiz                                               ▷ O(1)
2: Nat i ← 0                                                                 ▷ O(1)
3: bool esNueva ← true                                                       ▷ O(1)
4: while i < Longitud(p) do                                                  ▷ Se repite |p| O(1)
5:     if (*actual).siguientes[ord(p[i])] = NULL then                        ▷ O(1)
6:         (*actual).siguientes[ord(p[i])] ← & ⟨ NULL, arreglo[256](NULL), NULL ⟩   ▷ O(1)
7:         esNueva ← false                                                   ▷ O(1)
8:     end if
9:     actual ← (*actual).siguientes[ord(p[i])]                              ▷ O(1)
10:    i ← i + 1
11: end while
12: if (*actual).definicion ≠ NULL then                                      ▷ O(1)
13:     (*actual).definicion ← NULL          ▷ se libera la memoria acupada por definicion O(1)
14: end if
15: (*actual).definicion ← & a                                               ▷ O(1)
16: if esNueva then                                                          ▷ O(1)
17:     itConj(string) it ← claves.AgregarRapido(s)                          ▷ O(1)
18:     (*actual).itClave ← & it                                             ▷ O(1)
19: end if
```

20: Complejidad: $O(|P|)$
21: Justificaion: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(|p|) = O(|P|)$

iDef?(**in** p: `string`) $\rightarrow$ res : $bool$

```
1: Nat i ← 0                                                                 ▷ O(1)
2: bool pertenece ← true                                                     ▷ O(1)
3: puntero(nodo) actual ← raiz                                               ▷ O(1)
4: while i < Longitud(p) ∧ pertenece do                                      ▷ Se repite |p| O(1)
5:     if (*actual).siguientes[ord(p[i])] = NULL then                        ▷ O(1)
6:         pertenece ← false                                                 ▷ O(1)
7:     end if
8:     actual ← (*actual).siguientes[ord(p[i])]                              ▷ O(1)
9:     i ← i + 1                                                             ▷ O(1)
10: end while
11: if (*actual).significado = NULL then                                     ▷ O(1)
12:     pertenece ← false                                                    ▷ O(1)
13: end if
14: res ← pertenece                                                          ▷ O(1)
```

15: Complejidad: $O(|P|)$
16: Justificaion: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(|p|) = O(|P|)$

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

```
1: bool borrarRaiz ← d.Claves() = 1                                                    ▷ O(1)
2: puntero(nodo) reserva ← raiz                                                        ▷ O(1)
3: nat rindex ← 0                                                                      ▷ O(1)
4: puntero(nodo) actual ← raiz                                                         ▷ O(1)
5: nat i ← 0                                                                           ▷ O(1)
6: while i < Longitud(p) do                                                            ▷ Se repite |p| O(1)
7:     actual ← (*actual).siguientes[ord(p[i])]                                        ▷ O(1)
8:     bool definido ← i ≠ |p| − 1 ∧ (*actual).definicion ≠ NULL                        ▷ O(1)
9:     if CuentaHijos(actual) > 1 ∨ definido then                                      ▷ O(1)
10:        reserva ← actual                                                            ▷ O(1)
11:        rindex ← i + 1                                                              ▷ O(1)
12:    end if
13:    i ← i + 1                                                                       ▷ O(1)
14: end while
15: EliminarSiguiente((*actual).(*itClave))                                            ▷ O(1)
16: (*actual).itClave ← NULL          ▷ Se libera la memoria ocupada por (*actual).itClave O(1)
17: if CuentaHijos(actual) > 1 then                                                    ▷ O(1)
18:    (*actual).definicion ← NULL      ▷ se libera la memoria ocupada por la definicion O(1)
19: end if
20: if CuentaHijos(actual) = 0 then                                                    ▷ O(1)
21:    BorrarDesde(reserva, rindex)                                                    ▷ O(|P|)
22: end if
23: if borrarRaiz then                                                                 ▷ O(1)
24:    d.raiz ← NULL                          ▷ se libera la memoria ocupada por raiz O(1)
25: end if
```

26: Complejidad: $O(|P|)$

27: Justificaion: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(2 * |p|) = O(|p|) = O(|P|)$

Pre ≡ {*desde* y *desde*.siguientes[*a*] no nulos}
Post ≡ {Borra la rama a partir de *desde*.siguientes[*a*]}

---

iBorrarDesde(**in/out** *desde* : `puntero(nodo)`, **in** *a* : `nat`)

```
1: puntero(nodo) temp ← raiz                                                          ▷ O(1)
2: desde ← (*desde).siguientes[a]                                                     ▷ O(1)
3: (*desde).siguientes[a] ← NULL          ▷ Se libera la memoria ocupada por (*desde).siguientes[a] O(1)
4: while desde ≠ NULL do                  ▷ se repite a los sumo |P| veces, siendo p la clave mas larga O(1)
5:     puntero(nodo) temp ← desde                                                     ▷ O(1)
6:     bool sigue ← false                                                             ▷ O(1)
7:     nat i ← 0
8:     O(1)
9:     while i < 256 do                                  ▷ se repite siempre 256 veces O(1)
10:        if (*desde).siguientes[i] ≠ NULL then                                      ▷ O(1)
11:            desde ← (*desde).siguientes[i]                                         ▷ O(1)
12:            sigue ← true                                                           ▷ O(1)
13:        end if
14:        i ← i + 1                                                                  ▷ O(1)
15:    end while
16: end while
17: if ¬ sigue then                                                                   ▷ O(1)
18:    desde ← NULL                       ▷ se libera la memoria ocupada por desde O(1)
19: end if
20: temp ← NULL                           ▷ se libera la memoria ocupada por temp O(1)
21: Complejidad: O(|P|)
22: Justificaion: Siendo |P| el largo de la clave mas larga, sea cual sea la rama que estamos borrando, es mas corta
    que la rama representada por la clave mas larga. Llamo p a la clave de la rama que estamos borrando y como |p|
    ≤ |P| entonces O(|p|) = O(|P|)
```

Pre ≡ {*desde* no nulo}
Post ≡ {deveulve la cantidad de punteros no nulos en *desde*.siguientes}

---

iCuentaHijos(**in** *desde* : `puntero(nodo)` → *res* : *nat*)

```
1: nat i ← 0                                                                          ▷ O(1)
2: nat hijos ← 0                                                                      ▷ O(1)
3: while i < 256 do                                   ▷ se repite siempre 256 veces O(1)
4:     if (*actutal).siguiente[i] ≠ NULL then
5:         O(1)
6:         suma ← suma + 1                                                            ▷ O(1)
7:         i ← i + 1                                                                  ▷ O(1)
8:     end if
9: end while
10: res ← hijos                                                                       ▷ O(1)
11: Complejidad: O(1)
12: Justificaion: O(1) + O(1) + O(256) + O(1) + O(1) + O(1) = O(261) = O(1)
```

5.4. Servicios usados

De Conjunto Lineal
- Vacio() debe ser O(1)
- AgregarRapido(conj(α), α α) debe ser O(copy(a))
- EliminarSiguiente(itConj(α)) debe ser O(1)

De String
- Longitud(string) debe ser O(1)

De Char
- ord(char) debe ser O(1)

6. iterDiccString(α)

Interfaz

6.1. Interfaz

parámetros formales
géneros α

se explica con: IteradorUnidireccional(α), DiccString(α).

géneros: `iterDiccString(`α`)`.

CrearIt(**in** d: `diccString(`α`)`) $\rightarrow$ res : `iterDiccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), claves(d)))}
Complejidad: $O(1)$
Descripción: Devuelve un iterador al diccionario.
Aliasing: El iterador se invalida si se modifican claves del diccionario.

HayMas?(**in** it: `iterDiccString(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = hayMas?(it)}
Complejidad: $O(1)$
Descripción: Devuelve true si hay mas claves por recorrer.

Actual(**in** it: `iterDiccString(`α`)`) $\rightarrow$ res : `tupla<string, `α`>`
Pre $\equiv$ {hayMas?(it)}
Post $\equiv$ {res = actual(it)}
Complejidad: $O(|P|)$, donde |P| es la longitud de la clave mas larga.
Descripción: Devuelve una tupla con el elemento actual y su significado.

Avanzar(**in/out** it: `iterDiccString(`α`)`)
Pre $\equiv$ {$it = it_0 \wedge$ hayMas?(it)}
Post $\equiv$ {it = avanzar(it_0)}
Complejidad: $O(1)$
Descripción: Avanza a la posicion siguiente del iterador.

Representación

Este es el iterador que se usa para recorrer el DiccString. Para recorrerlo, aprovechando que tenemos un conjunto de claves en nuestro diccionario, vamos a recorrer el conjunto de claves usando el iterador de conjunto que ya existe. Para obtener el elemento, buscamos la clave "actual" (según iterador de claves) en el diccionario. Esto implica que obtener el elemento sea un poco costoso, porque hay que buscarlo en el diccionario, pero decidimos hacerlo así ya que nos pareció la forma mas simple que cumplía los requisitos de complejidad.

6.2. Representacion del iterDiccString

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

donde `estr` es `tupla(`*itClave*: `itConj(string)`, *dicc*: `diccString(`α`)`)

Invariante de representacion

Rep : estr e $\longrightarrow$ bool

Rep(e) $\equiv$ Rep(e.itClave) $\wedge_{\text{L}}$ ((siguiente(e.itClave) = NULL $\vee_{\text{L}}$ siguiente(e.itClave) $\in$ claves(e.dicc))

Funcion de abstraccion

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

Abs(e) $\equiv$ uni : itUni(α) /
siguientes(uni) $=_{\text{obs}}$ siguientes(e.itClave)

Algoritmos

6.3. Algoritmos

iCrearIt(**in** d : `diccString(`α`)`) $\rightarrow$ res : iterDiccString(α)

1: $res \leftarrow \langle$ d , CrearIt(Claves(d)) $\rangle$ ▷ Por referencia $O(1)$

Complejidad: $O(1)$
Justificacion: Crear el iterador de conjunto es O(1), obtener las claves del diccionario es O(1), crear la tupla con los dos elementos es O(1). La complejidad total es O(1)

iHayMas?(**in** $iter$: `iterDiccString(`α`)`) $\rightarrow$ res : bool

1: $res \leftarrow$ HaySiguiente($iter$.itClave) ▷ $O(1)$

Complejidad: $O(1)$

iActual(**in** $iter$: `iterDiccString(`α`)`) $\rightarrow$ res : tupla<string, α >

1: $res \leftarrow \langle$ Siguiente($iter$.iClave), Obtener(Siguiente($iter$.iClave), $iter$.dicc) $\rangle$ ▷ $O(|P|)$

Complejidad: $O(|P|)$
Justificación: Para crear la tupla, necesito acceder al significado de la clave. Por las complejidades del DiccString, el peor caso se corresponde con la longitud de la clave mas larga: O(|P|), donde P es la clave mas large. Acceder al Siguiente del conjunto es O(1). En total, el algoritmo cuesta O(|P|)

iAvanzar(**in/out** $iter$: `iterDiccString(`α`)`)

1: Avanzar($iter$.itClave) ▷ $O(1)$

Complejidad: $O(1)$

6.4. Servicios usados

De conj(α)
- CrearIt(conj(α)) debe ser O(1)

De itConj(α)
- HaySiguiente(itConj(α)) debe ser O(1)
- Siguiente(itConj(α)) debe ser O(1)

- Avanzar(itConj(α)) debe ser O(1)

De diccString(α)
- Claves(diccString(α)) debe ser O(1)
- Obtener(string, diccString(α)) debe ser O(|P|)

7. Cola de Entrenadores

Interfaz

7.1. Interfaz de Cola de Entrenadores

se explica con: Cola de Prioridad(α), Iterador Cola de Prioridades(α).

géneros: `colaEntr`, `itcolaEntrenador)`.

Operaciones básicas de Cola de Entrenadores

Vacia() $\rightarrow$ *res* : `colaEntr`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} vacia$}
Complejidad: $O(1)$
Descripción: genera una nueva cola de entrenadores

Encolar(**in/out** h: `colaEntr`, **in** j: `entrenador`) $\rightarrow$ *res* : `itcolaEntrenador)`
Pre $\equiv$ {$h =_{\text{obs}} h_0$}
Post $\equiv$ {$h =_{\text{obs}}$ encolar(h_0, j)}
Complejidad: $O(log(EC))$
Descripción: encola j a h
Aliasing: el elemento j se encola por copia

EsVacia?(**in** h: `colaEntr`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia?(h)}
Complejidad: $O(1)$
Descripción: devuelve true si la cola es vacia

Proximo(**in** h: `colaEntr`) $\rightarrow$ *res* : `itcolaEntrenador)`
Pre $\equiv$ {$\neg$ vacia?(h)}
Post $\equiv$ {alias($res =_{\text{obs}}$ proximo(h))}
Complejidad: $O(1)$
Descripción: devuelve el próximo de la cola
Aliasing: el elemento se devuelve por copia

Desencolar(**in/out** h: `colaEntr`)
Pre $\equiv$ {$h =_{\text{obs}} h_0 \wedge \neg$ vacia(h) }
Post $\equiv$ {$h =_{\text{obs}}$ desencolar(h_0)}
Complejidad: $O(log(EC))$
Descripción: desencola el próximo de h

Operaciones del iterador

7.2. Interfaz del iterador

Borrar(**in** *it*: `itcolaEntrenador`, **in** h: `colaEntr`)
Pre $\equiv$ {$siguiente(it) \in elementos(h)$}
Post $\equiv$ {$siguiente(it) \notin elementos(h)$}
Complejidad: $O(log(EC))$
Descripción: Borra el elemento dado

Representación

7.3. Representacion de Cola de Entrenadores

`colaEntr` se representa con `estr`

donde `estr` es `tupla`(*raiz*: `puntero(Nodoheap)`, *ultimo*: `puntero(Nodoheap)`)

donde `Nodoheap` es `tupla`(*elemento*: `entrenador`, *padre*: `puntero(Nodoheap)`, *izq*: `puntero(Nodoheap)`, *der*: `puntero(Nodoheap)`)

donde `entrenador` es `tupla`(*id*: `nat`, *cantCapt*: `nat`)

Invariante de representación en castellano

- Si raiz es $NULL$ entonces ultimo tambien es $NULL$.
- Si la raiz no es $NULL$ entonces ultimo tampoco es $NULL$..
- Ningun nodo tiene como padre a ninguno de sus hijos ni a los hijos de sus hijos.

Abs : estr c $\longrightarrow$ colaPrior(Tupla(nat, nat)) {Rep(d)}

Abs(c) $\equiv$ $cola$: colaPrior(Tupla(nat, nat)) /
vacia?($cola$) $\iff$ $c.raiz = NULL$ $\wedge_{\text{L}}$
proximo($cola$) $=_{\text{obs}}$ $c.(*raiz)$.elemento

Representación

7.4. Representacion del iterador

`itcolaEntrenador` se representa con `estr`

donde `estr` es `tupla`(*siguiente*: `puntero(Nodoheap)`, *estructura*: `puntero(colaEntr)`)

Rep : estr e $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Leftrightarrow$ Rep(*(e.estructura)) $\wedge_{\text{L}}$ (it.siguiente = NULL) $\vee_{\text{L}}$ ($\exists$ j, k: nat)($\exists$ i, d, p: puntero(Nodoheap))(Nodohep(j, k, i, d, p) = it.siguiente)

Nodoheap : nat j $\times$ nat k $\times$ punt(Nodoheap) i $\times$ punt(Nodoheap) d $\times$ punt(Nodoheap) p $\longrightarrow$ punt(Nodoheap)

Nodoheap(j, k, i, d, p) $\equiv$ <<i, j>, p, i, d>

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

Abs(e) $\equiv$ it : itcolaEntr /
siguiente(it) $= NULL$ $\vee_{\text{L}}$
siguiente(it) $=_{\text{obs}}$ *(e.siguiente)

Algoritmos

7.5. Algoritmos Cola de Entrenadores

iVacio() $\rightarrow res$: colaEntr

1: $res \leftarrow < NULL, NULL >$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Todas las operaciones son $O(1)$

iEsVacia?(**in** h : `colaEntr`) $\rightarrow res$: bool

1: $res \leftarrow h$.raiz $\neq NULL$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Todas las operaciones son $O(1)$

iEncolarElem(**in/out** h : `colaEntr`, **in** j : `entrenador`) $\rightarrow res$: itcolaEntrenador)

1: Nodoheap $n \leftarrow < j, NULL, NULL, NULL >$ ▷ $O(1)$
2: puntero(Nodoheap) $p \leftarrow \& \; n$ ▷ $O(1)$
3: **if** $EsVacia?(h)$ **then** ▷ Insertamos el nodo al final de la cola $O(1)$
4: h.raiz $\leftarrow p$ ▷ $O(1)$
5: h.ultimo $\leftarrow p$ ▷ $O(1)$
6: **else**
7: **if** h.raiz $= h$.ultimo **then** ▷ $O(1)$
8: $(p \rightarrow padre) \leftarrow h$.raiz ▷ $O(1)$
9: $(h$.raiz $\rightarrow izq) \leftarrow p$ ▷ $O(1)$
10: h.ultimo $\leftarrow p$ ▷ $O(1)$
11: **else**
12: **if** $(h$.raiz $\rightarrow izq) = h$.ultimo **then** ▷ $O(1)$
13: $(p \rightarrow padre) \leftarrow h$.raiz ▷ $O(1)$
14: $(h$.raiz $\rightarrow der) \leftarrow p$ ▷ $O(1)$
15: h.ultimo $\leftarrow p$ ▷ $O(1)$
16: **else**
17: **if** EsHijoIzquierdo?(h.ultimo) **then** ▷ $O(1)$
18: $(p \rightarrow padre) \leftarrow (h$.ultimo $\rightarrow padre)$ ▷ $O(1)$
19: $(h$.ultimo $\rightarrow padre \rightarrow der) \leftarrow p$ ▷ $O(1)$
20: h.ultimo $\leftarrow p$ ▷ $O(1)$
21: **else**
22: puntero(Nodoheap) $i \leftarrow h$.ultimo ▷ $O(1)$
23: **while** $i \neq NULL \wedge \neg$ EsHijoIzquierdo?(j) **do** ▷ $O(log(EC))$
24: $i \leftarrow (i \rightarrow padre)$ ▷ $O(1)$
25: **end while**
26: **if** $i = NULL$ **then** ▷ $O(1)$
27: puntero(Nodoheap) $ultimoizq \leftarrow h$.raiz ▷ $O(1)$
28: **while** $(ultimoizq \rightarrow izq) \neq NULL$ **do** ▷ $O(log(EC))$
29: $ultimoizq \leftarrow (ultimoizq \rightarrow izq)$ ▷ $O(1)$
30: **end while**
31: $(p \rightarrow padre) \leftarrow ultimoizqu$ ▷ $O(1)$
32: $(ultimoizq \rightarrow izq) \leftarrow p$ ▷ $O(1)$
33: $h.ultimo \leftarrow p$ ▷ $O(1)$
34: **else**
35: $i \leftarrow (i \rightarrow padre \rightarrow der)$ ▷ $O(1)$
36: **while** $(i \rightarrow izq) \neq NULL$ **do** ▷ $O(log(EC))$
37: $i \leftarrow (i \rightarrow izq)$ ▷ $O(1)$
38: **end while**
39: $(p \rightarrow padre) \leftarrow i$ ▷ $O(1)$
40: $(j \rightarrow izq) \leftarrow p$ ▷ $O(1)$

```
                    h.ultimo ← p                                                    ▷ O(1)
                end if
            end if
        end if
    end if
end if
puntero(colaEntr)q ← h
it ← CrearItCola(h.ultimo, q)                                                       ▷ O(1)
if h.raiz ≠ h.ultimo then                                                           ▷ O(1)
    puntero(Nodo) nuevoNodo ← h.ultimo                                              ▷ O(1)
    while ((nuevoNodo → elemento.cantCapt) < (nuevoNodo → padre → elemento.cantCapt))∨(((nuevoNodo →
    elemento.cantCapt) = (nuevoNodo → padre → elemento.cantCapt))∧((nuevoNodo → elemento.id) < (nuevoNodo →
    padre → elemento.id))) do                                                       ▷ O(log(EC))
        puntero(Nodoheap) aSwapear ← (nuevoNodo → padre)
        if h.ultimo = nuevoNodo then
            h.ultimo ← aSwapear
        end if
        if EsHijoIzquierdo?(nuevoNodo) then                                         ▷ O(1)
            if (nuevoNodo → izq) = NULL then                                        ▷ O(1)
                (aSwapear → izq) ← NULL                                             ▷ O(1)
            else
                (aSwapear → izq) ← (nuevoNodo → izq)                                ▷ O(1)
                (nuevoNodo → izq → padre) ← aSwapear                                ▷ O(1)
            end if
            if (nuevoNodo → der) = NULL then                                        ▷ O(1)
                (nuevoNodo → der) ← (aSwapear → der)                                ▷ O(1)
                if (aSwapear → der) ≠ NULL then                                     ▷ O(1)
                    (aSwapear → der → padre) ← nuevoNodo                            ▷ O(1)
                end if
                (aSwapear → der) ← NULL                                             ▷ O(1)
            else
                if (aSwapear → der) = NULL then                                     ▷ O(1)
                    (aSwapear → der) ← (nodoNuevo → der)                            ▷ O(1)
                    (nodoNuevo → der → padre) ← aSwapear                            ▷ O(1)
                    (nodoNuevo → der) ← NULL                                        ▷ O(1)
                else
                    (nodoNuevo → der → padre) ← aSwapear                            ▷ O(1)
                    (aSwapear → der → padre) ← nuevoNodo                            ▷ O(1)
                    puntero(Nodoheap) auxiliar ← (nuevoNodo → der)                  ▷ O(1)
                    (nuevoNodo → der) ← (aSwapear → der)                            ▷ O(1)
                    (aSwapear → der) ← auxiliar                                     ▷ O(1)
                end if
            end if
            (nuevoNodo → izq) ← aSwapear                                            ▷ O(1)
        else
            if (nuevoNodo → der) = NULL then                                        ▷ O(1)
                (aSwapear → der) ← NULL                                             ▷ O(1)
            else
                (aSwapear → der) ← (nuevoNodo → der)                                ▷ O(1)
                (nuevoNodo → der → padre) ← aSwapear                                ▷ O(1)
            end if
            if (nuevoNodo → izq) = NULL then                                        ▷ O(1)
                (nuevoNodo → izq) ← (aSwapear → izq)                                ▷ O(1)
                if (aSwapear → izq) ≠ NULL then                                     ▷ O(1)
                    (aSwapear → izq → padre) ← nuevoNodo                            ▷ O(1)
                end if
                (aSwapear → izq) ← NULL                                             ▷ O(1)
            else
```

```
97:                 if (aSwapear → izq) = NULL then                                  ▷ O(1)
98:                     (aSwapear → izq) ← (nodoNuevo → izq)                         ▷ O(1)
99:                     (nodoNuevo → izq → padre) ← aSwapear                         ▷ O(1)
100:                    (nodoNuevo → izq) ← NULL                                     ▷ O(1)
101:                else
102:                    (nodoNuevo → izq → padre) ← aSwapear                         ▷ O(1)
103:                    (aSwapear → izq → padre) ← nuevoNodo                         ▷ O(1)
104:                    puntero(Nodoheap) auxiliar ← (nuevoNodo → izq)               ▷ O(1)
105:                    (nuevoNodo → izq) ← (aSwapear → izq)                         ▷ O(1)
106:                    (aSwapear → izq) ← auxiliar                                  ▷ O(1)
107:                end if
108:            end if
109:            (nuevoNodo → der) ← aSwapear                                         ▷ O(1)
110:        end if
111:        if aSwapear = h.raiz then
112:            (aSwapear → padre) ← nuevoNodo                                       ▷ O(1)
113:            h.raiz ← nuevoNodo                                                   ▷ O(1)
114:            (nuevoNodo → padre) ← NULL                                           ▷ O(1)
115:        else
116:            puntero(Nodoheap) abuelo ← (aSwapear → padre)                        ▷ O(1)
117:            if EsHijoIzquierdo?(aSwapear) then                                   ▷ O(1)
118:                (abuelo → izq) ← nuevoNuevo                                      ▷ O(1)
119:                (nuevoNodo → padre) ← abuelo                                     ▷ O(1)
120:                (aSwapear → padre) ← nuevoNodo                                   ▷ O(1)
121:            else
122:                (abuelo → der) ← nuevoNuevo                                      ▷ O(1)
123:                (nuevoNodo → padre) ← abuelo                                     ▷ O(1)
124:                (aSwapear → padre) ← nuevoNodo                                   ▷ O(1)
125:            end if
126:        end if
127:    end while
128: end if
129: res ← it                                                                        ▷ O(1)
```

Complejidad: $O(log(EC))$
Justificación: Todas las operaciones son $O(1)$ y en el ciclo se va recorriendo la cola desde una de las hojas hasta la raiz (o vice versa). Al ser la cola un árbol completo (ya que siempre se agregan elementos al último lugar disponible), si la cantidad de elementos es EC en el peor cas, entonces su altura es log(EC). Por lo tanto los ciclos se repiten log(EC) veces en el peor caso.

---

iProximo(**in** h : `colaEntr`) → res : entrenador

```
1: res ← (h.raiz → elemento)                                                         ▷ O(1)
```

Complejidad: $O(1)$
Justificación: Todas las operaciones son $O(1)$

---

iDesencolar(**in** h : `colaEntr`)

```
1: puntero(colaEntr)q ← h
2: it ← CrearItCola(h.raiz, q)                                                       ▷ O(1)
3: Borrar(it)                                                                        ▷ O(log(EC))
```

Complejidad: $O(log(EC))$
Justificación: Hay una operacion $O(1)$ y una $O(log(EC))$, por lo tanto, por álgebra de órdenes $O(1) + O(log(EC)) = O(log(EC))$

Algoritmos

7.6. Algoritmos del iterador

iBorrar(**in** it : `itcolaEntrenador`), **in** h : colaEntr)

1: **if** $(it.estructura \rightarrow raiz) = (it.estructura \rightarrow ultimo)$ **then** ▷ $O(1)$
2: $it.siguiente \leftarrow NULL$ ▷ Libera memoria $O(1)$
3: $(it.estructura \rightarrow raiz) \leftarrow NULL$ ▷ $O(1)$
4: $(it.estructura \rightarrow ultimo) \leftarrow NULL$ ▷ $O(1)$
5: **else**
6: $puntero(Nodo)$ cambiado $\leftarrow (it.estructura \rightarrow ultimo)$ ▷ $O(1)$
7: $puntero(Nodo)$ aPonerUltimo $\leftarrow it.siguiente$
8: **if** $(it.estructura \rightarrow ultimo \rightarrow padre) \neq aPonerUltimo$ **then** ▷ Swapeamos con el último de la cola. $O(1)$
9: $(it.estructura \rightarrow ultimo \rightarrow izq) \leftarrow (aPonerUltimo \rightarrow izq)$ ▷ $O(1)$
10: $(it.estructura \rightarrow ultimo \rightarrow der) \leftarrow (aPonerUltimo \rightarrow der)$ ▷ $O(1)$
11: **if** $(aPonerUltimo \rightarrow izq) \neq NULL$ **then** ▷ $O(1)$
12: $(aPonerUltimo \rightarrow izq \rightarrow padre) \leftarrow (it.estructura \rightarrow ultimo)$ ▷ $O(1)$
13: **end if**
14: **if** $(aPonerUltimo \rightarrow der) \neq NULL$ **then** ▷ $O(1)$
15: $(aPonerUltimo \rightarrow der \rightarrow padre) \leftarrow it.estructura \rightarrow ultimo$ ▷ $O(1)$
16: **end if**
17: **if** $aPonerUltimo \neq it.estructura \rightarrow raiz$ **then** ▷ $O(1)$
18: $padreaPonerUltimo \leftarrow (aPonerUltimo \rightarrow padre)$ ▷ $O(1)$
19: $padreUltimo \leftarrow (it.estructura \rightarrow ultimo \rightarrow padre)$ ▷ $O(1)$
20: **if** $EsHijoIzquierdo?(aPonerUltimo) \wedge EsHijoIzquierdo?(it.estructura \rightarrow ultimo)$ **then** ▷ $O(1)$
21: $(padreaPonerUltimo \rightarrow izq) \leftarrow it.estructura \rightarrow ultimo$ ▷ $O(1)$
22: $(it.estructura \rightarrow ultimo \rightarrow padre) \leftarrow padreaPonerUltimo$ ▷ $O(1)$
23: $(aPonerUltimo \rightarrow padre) \leftarrow padreUltimo$ ▷ $O(1)$
24: $(padreUltimo \rightarrow izq) \leftarrow aPonerUltimo$ ▷ $O(1)$
25: **else**
26: **if** $EsHijoIzquierdo?(aPonerUltimo) \wedge \neg EsHijoIzquierdo?(it.estructura \rightarrow ultimo)$ **then** ▷ $O(1)$
27: $(padreaPonerUltimo \rightarrow izq) \leftarrow it.estructura \rightarrow ultimo$ ▷ $O(1)$
28: $(aPonerUltimo \rightarrow padre) \leftarrow padreUltimo$ ▷ $O(1)$
29: $(it.estructura \rightarrow ultimo \rightarrow padre) \leftarrow padreaPonerUltimo$ ▷ $O(1)$
30: $(padreUltimo \rightarrow der) \leftarrow aPonerUltimo$ ▷ $O(1)$
31: **else**
32: **if** $\neg EsHijoIzquierdo?(aPonerUltimo) \wedge EsHijoIzquierdo?(it.estructura \rightarrow ultimo)$ **then** ▷ $O(1)$
33: $(padreaPonerUltimo \rightarrow izq) \leftarrow it.estructura \rightarrow ultimo$ ▷ $O(1)$
34: $(aPonerUltimo \rightarrow padre) \leftarrow padreUltimo$ ▷ $O(1)$
35: $(it.estructura \rightarrow ultimo \rightarrow padre) \leftarrow padreaPonerUltimo$ ▷ $O(1)$
36: $(padreUltimo \rightarrow izq) \leftarrow aPonerUltimo$ ▷ $O(1)$
37: **else**
38: **if** $\neg EsHijoIzquierdo?(aPonerUltimo) \wedge \neg EsHijoIzquierdo?(it.estructura \rightarrow ultimo)$ **then** ▷ $O(1)$
39: $(padreaPonerUltimo \rightarrow der) \leftarrow (it.estructura \rightarrow ultimo)$ ▷ $O(1)$
40: $(it.estructura \rightarrow ultimo \rightarrow padre) \leftarrow padreaPonerUltimo$ ▷ $O(1)$
41: $(aPonerUltimo \rightarrow padre) \leftarrow padreUltimo$ ▷ $O(1)$
42: $(padreUltimo \rightarrow der) \leftarrow aPonerUltimo$ ▷ $O(1)$
43: **end if**
44: **end if**
45: **end if**
46: **end if**
47: **else**

```
48:             (it.estructura → raiz → padre) ← (it.estructura → ultimo → padre)                ▷ O(1)
49:             if EsHijoIzquierdo?(it.estructura → ultimo) then                                 ▷ O(1)
50:                 (it.estructura → ultimo → padre → izq) ← it.estructura → raiz                ▷ O(1)
51:             else
52:                 (it.estructura → ultimo → padre → der) ← it.estructura → raiz                ▷ O(1)
53:             end if
54:             (it.estructura → ultimo → padre) ← NULL                                          ▷ O(1)
55:             (it.estructura → raiz) ← (it.estructura → ultimo)                                ▷ O(1)
56:         end if
57:     else
58:         puntero(Nodoheap)nuevoNodo ← (it.estructura → ultimo)                                ▷ O(1)
59:         puntero(Nodoheap) aSwapear ← (nuevoNodo → padre)
60:         if (it.estructura → ultimo) = nuevoNodo then
61:             (it.estructura → ultimo) ← aSwapear
62:         end if
63:         if EsHijoIzquierdo?(nuevoNodo) then                                                  ▷ O(1)
64:             if (nuevoNodo → izq) = NULL then                                                 ▷ O(1)
65:                 (aSwapear → izq) ← NULL                                                      ▷ O(1)
66:             else
67:                 (aSwapear → izq) ← (nuevoNodo → izq)                                         ▷ O(1)
68:                 (nuevoNodo → izq → padre) ← aSwapear                                         ▷ O(1)
69:             end if
70:             if (nuevoNodo → der) = NULL then                                                 ▷ O(1)
71:                 (nuevoNodo → der) ← (aSwapear → der)                                         ▷ O(1)
72:                 if (aSwapear → der) ≠ NULL then                                              ▷ O(1)
73:                     (aSwapear → der → padre) ← nuevoNodo                                     ▷ O(1)
74:                 end if
75:                 (aSwapear → der) ← NULL                                                      ▷ O(1)
76:             else
77:                 if (aSwapear → der) = NULL then                                              ▷ O(1)
78:                     (aSwapear → der) ← (nodoNuevo → der)                                     ▷ O(1)
79:                     (nodoNuevo → der → padre) ← aSwapear                                     ▷ O(1)
80:                     (nodoNuevo → der) ← NULL                                                 ▷ O(1)
81:                 else
82:                     (nodoNuevo → der → padre) ← aSwapear                                     ▷ O(1)
83:                     (aSwapear → der → padre) ← nuevoNodo                                     ▷ O(1)
84:                     puntero(Nodoheap) auxiliar ← (nuevoNodo → der)                           ▷ O(1)
85:                     (nuevoNodo → der) ← (aSwapear → der)                                     ▷ O(1)
86:                     (aSwapear → der) ← auxiliar                                              ▷ O(1)
87:                 end if
88:             end if
89:             (nuevoNodo → izq) ← aSwapear                                                     ▷ O(1)
90:         else
91:             if (nuevoNodo → der) = NULL then                                                 ▷ O(1)
92:                 (aSwapear → der) ← NULL                                                      ▷ O(1)
93:             else
94:                 (aSwapear → der) ← (nuevoNodo → der)                                         ▷ O(1)
95:                 (nuevoNodo → der → padre) ← aSwapear                                         ▷ O(1)
96:             end if
97:             if (nuevoNodo → izq) = NULL then                                                 ▷ O(1)
98:                 (nuevoNodo → izq) ← (aSwapear → izq)                                         ▷ O(1)
99:                 if (aSwapear → izq) ≠ NULL then                                              ▷ O(1)
100:                    (aSwapear → izq → padre) ← nuevoNodo                                      ▷ O(1)
101:                end if
102:                (aSwapear → izq) ← NULL                                                       ▷ O(1)
103:            else
104:                if (aSwapear → izq) = NULL then                                               ▷ O(1)
105:                    (aSwapear → izq) ← (nodoNuevo → izq)                                      ▷ O(1)
```

```
106:                (nodoNuevo → izq → padre) ← aSwapear                                   ▷ O(1)
107:                (nodoNuevo → izq) ← NULL                                               ▷ O(1)
108:            else
109:                (nodoNuevo → izq → padre) ← aSwapear                                   ▷ O(1)
110:                (aSwapear → izq → padre) ← nuevoNodo                                   ▷ O(1)
111:                puntero(Nodoheap) auxiliar ← (nuevoNodo → izq)                         ▷ O(1)
112:                (nuevoNodo → izq) ← (aSwapear → izq)                                   ▷ O(1)
113:                (aSwapear → izq) ← auxiliar                                            ▷ O(1)
114:            end if
115:        end if
116:        (nuevoNodo → der) ← aSwapear                                                   ▷ O(1)
117:    end if
118:    if aSwapear = (it.estructura → raiz) then
119:        (aSwapear → padre) ← nuevoNodo                                                 ▷ O(1)
120:        (it.estructura → raiz) ← nuevoNodo                                             ▷ O(1)
121:        (nuevoNodo → padre) ← NULL                                                     ▷ O(1)
122:    else
123:        puntero(Nodoheap) abuelo ← (aSwapear → padre)                                  ▷ O(1)
124:        if EsHijoIzquierdo?(aSwapear) then                                             ▷ O(1)
125:            (abuelo → izq) ← nuevoNuevo                                                ▷ O(1)
126:            (nuevoNodo → padre) ← abuelo                                               ▷ O(1)
127:            (aSwapear → padre) ← nuevoNodo                                             ▷ O(1)
128:        else
129:            (abuelo → der) ← nuevoNuevo                                                ▷ O(1)
130:            (nuevoNodo → padre) ← abuelo                                               ▷ O(1)
131:            (aSwapear → padre) ← nuevoNodo                                             ▷ O(1)
132:        end if
133:    end if
134: end if
135: (it.estructura → ultimo) ← aPonerUltimo                                               ▷ O(1)
136: if EsHijoIzquierdo?(it.estructura → ultimo) then                                      ▷ O(1)
137:    puntero(Nodoheap) buscador ← (it.estructura → ultimo)                              ▷ O(1)
138:    nat i ← 7                                                                          ▷ O(1)
139:    while (buscador → padre) ≠ NULL ∧i = 7 do                                          ▷ O(log(EC))
140:        if ¬EsHijoIzquierdo?(buscador) then                                            ▷ O(1)
141:            i ← 0                                                                      ▷ O(1)
142:        else buscador ← (buscador → padre)                                             ▷ O(1)
143:        end if
144:    end while
145:    if (buscador → padre) = NULL then                                                  ▷ O(1)
146:        while (buscador → der) ≠ NULL do                                               ▷ O(log(EC))
147:            buscador ← (buscador → der)                                                ▷ O(1)
148:        end while
149:        (it.estructura → ultimo) ← NULL                          ▷ Libera memoria O(1)
150:        (it.estructura → ultimo) ← buscador                                            ▷ O(1)
151:    elsebuscador ← (buscador → padre → izq)                                            ▷ O(1)
152:        while (buscador → der) ≠ NULL do                                               ▷ O(log(EC))
153:            buscador ← (buscador → der)                                                ▷ O(1)
154:        end while
155:        (it.estructura → ultimo) ← NULL                          ▷ Libera memoria O(1)
156:        (it.estructura → ultimo) ← buscador                                            ▷ O(1)
157:    end if
158: else
159:    (it.estructura → ultimo) ← (it.estructura → ultimo → padre → izq)                  ▷ O(1)
160:    (it.estructura → ultimo → padre → der) ← NULL   ▷ Libera memoria del nodo borrado O(1)
161: end if
162: while (cambiado → der) ≠ NULL do                                                      ▷ O(log(EC))
163:    puntero(Nodoheap)minimo ← Min(cambiado → izq, cambiado → der)                      ▷ O(1)
```

```
164:        if ((minimo → elemento.cantCapt) < (cambiado → elemento.cantCapt))∨(((minimo → elemento.cantCapt)
    = (cambiado → elemento.cantCapt)) ∧ ((minimo → elemento.id) < (cambiadoelemento.id))) then        ▷ O(1)
165:            puntero(Nodoheap)nuevoNodo ← minimo
166:            puntero(Nodoheap) aSwapear ← (nuevoNodo → padre)
167:            if (it.estructura → ultimo) = nuevoNodo then                          ▷ Swapea los Nodos O(1)
168:                it.estructura → ultimo ← aSwapear                                              ▷ O(1)
169:            end if
170:            if EsHijoIzquierdo?(nuevoNodo) then                                                ▷ O(1)
171:                if (nuevoNodo → izq) = NULL then                                               ▷ O(1)
172:                    (aSwapear → izq) ← NULL                                                    ▷ O(1)
173:                else
174:                    (aSwapear → izq) ← (nuevoNodo → izq)                                       ▷ O(1)
175:                    (nuevoNodo → izq → padre) ← aSwapear                                       ▷ O(1)
176:                end if
177:                if (nuevoNodo → der) = NULL then                                               ▷ O(1)
178:                    (nuevoNodo → der) ← (aSwapear → der)                                       ▷ O(1)
179:                    if (aSwapear → der) ≠ NULL then                                            ▷ O(1)
180:                        (aSwapear → der → padre) ← nuevoNodo                                   ▷ O(1)
181:                    end if
182:                    (aSwapear → der) ← NULL                                                    ▷ O(1)
183:                else
184:                    if (aSwapear → der) = NULL then                                            ▷ O(1)
185:                        (aSwapear → der) ← (nodoNuevo → der)                                   ▷ O(1)
186:                        (nodoNuevo → der → padre) ← aSwapear                                   ▷ O(1)
187:                        (nodoNuevo → der) ← NULL                                               ▷ O(1)
188:                    else
189:                        (nodoNuevo → der → padre) ← aSwapear                                   ▷ O(1)
190:                        (aSwapear → der → padre) ← nuevoNodo                                   ▷ O(1)
191:                        puntero(Nodoheap) auxiliar ← (nuevoNodo → der)                         ▷ O(1)
192:                        (nuevoNodo → der) ← (aSwapear → der)                                   ▷ O(1)
193:                        (aSwapear → der) ← auxiliar                                            ▷ O(1)
194:                    end if
195:                end if
196:                (nuevoNodo → izq) ← aSwapear                                                   ▷ O(1)
197:            else
198:                if (nuevoNodo → der) = NULL then                                               ▷ O(1)
199:                    (aSwapear → der) ← NULL                                                    ▷ O(1)
200:                else
201:                    (aSwapear → der) ← (nuevoNodo → der)                                       ▷ O(1)
202:                    (nuevoNodo → der → padre) ← aSwapear                                       ▷ O(1)
203:                end if
204:                if (nuevoNodo → izq) = NULL then                                               ▷ O(1)
205:                    (nuevoNodo → izq) ← (aSwapear → izq)                                       ▷ O(1)
206:                    if (aSwapear → izq) ≠ NULL then                                            ▷ O(1)
207:                        (aSwapear → izq → padre) ← nuevoNodo                                   ▷ O(1)
208:                    end if
209:                    (aSwapear → izq) ← NULL                                                    ▷ O(1)
210:                else
211:                    if (aSwapear → izq) = NULL then                                            ▷ O(1)
212:                        (aSwapear → izq) ← (nodoNuevo → izq)                                   ▷ O(1)
213:                        (nodoNuevo → izq → padre) ← aSwapear                                   ▷ O(1)
214:                        (nodoNuevo → izq) ← NULL                                               ▷ O(1)
215:                    else
216:                        (nodoNuevo → izq → padre) ← aSwapear                                   ▷ O(1)
217:                        (aSwapear → izq → padre) ← nuevoNodo                                   ▷ O(1)
218:                        puntero(Nodoheap) auxiliar ← (nuevoNodo → izq)                         ▷ O(1)
219:                        (nuevoNodo → izq) ← (aSwapear → izq)                                   ▷ O(1)
220:                        (aSwapear → izq) ← auxiliar                                            ▷ O(1)
```

```
221:                end if
222:            end if
223:            (nuevoNodo → der) ← aSwapear                                    ▷ O(1)
224:        end if
225:        if aSwapear = (it.estructura → raiz) then
226:            (aSwapear → padre) ← nuevoNodo                                  ▷ O(1)
227:            (it.estructura → raiz) ← nuevoNodo                              ▷ O(1)
228:            (nuevoNodo → padre) ← NULL                                      ▷ O(1)
229:        else
230:            puntero(Nodoheap) abuelo ← (aSwapear → padre)                   ▷ O(1)
231:            if EsHijoIzquierdo?(aSwapear) then                              ▷ O(1)
232:                (abuelo → izq) ← nuevoNuevo                                 ▷ O(1)
233:                (nuevoNodo → padre) ← abuelo                                ▷ O(1)
234:                (aSwapear → padre) ← nuevoNodo                              ▷ O(1)
235:            else
236:                (abuelo → der) ← nuevoNuevo                                 ▷ O(1)
237:                (nuevoNodo → padre) ← abuelo                                ▷ O(1)
238:                (aSwapear → padre) ← nuevoNodo                              ▷ O(1)
239:            end if
240:        end if
241:    end if
242: end while
243: if (cambiado → izq) ≠ NULL then                                            ▷ O(1)
244:    if ((cambiado → izq → elemento.cantCapt) < (cambiado → elemento.cantCapt)) ∨ (((cambiado →
    izq → elemento.cantCapt) = (cambiado → elemento.cantCapt)) ⇒ ((cambiado → izq → elemento.id) <
    (cambiadoelemento.id))) then                                                ▷ O(1)
245:        puntero(Nodoheap)nuevoNodo ← (cambiado → izq)
246:        puntero(Nodoheap) aSwapear ← (nuevoNodo → padre)
247:        if (it.estructura → ultimo) = nuevoNodo then                        ▷ Swapea los nodos O(1)
248:            it.estructura → ultimo ← aSwapear                               ▷ O(1)
249:        end if
250:        if EsHijoIzquierdo?(nuevoNodo) then                                 ▷ O(1)
251:            if (nuevoNodo → izq) = NULL then                                ▷ O(1)
252:                (aSwapear → izq) ← NULL                                     ▷ O(1)
253:            else
254:                (aSwapear → izq) ← (nuevoNodo → izq)                        ▷ O(1)
255:                (nuevoNodo → izq → padre) ← aSwapear                        ▷ O(1)
256:            end if
257:            if (nuevoNodo → der) = NULL then                                ▷ O(1)
258:                (nuevoNodo → der) ← (aSwapear → der)                        ▷ O(1)
259:                if (aSwapear → der) ≠ NULL then                             ▷ O(1)
260:                    (aSwapear → der → padre) ← nuevoNodo                    ▷ O(1)
261:                end if
262:                (aSwapear → der) ← NULL                                     ▷ O(1)
263:            else
264:                if (aSwapear → der) = NULL then                             ▷ O(1)
265:                    (aSwapear → der) ← (nodoNuevo → der)                    ▷ O(1)
266:                    (nodoNuevo → der → padre) ← aSwapear                    ▷ O(1)
267:                    (nodoNuevo → der) ← NULL                                ▷ O(1)
268:                else
269:                    (nodoNuevo → der → padre) ← aSwapear                    ▷ O(1)
270:                    (aSwapear → der → padre) ← nuevoNodo                    ▷ O(1)
271:                    puntero(Nodoheap) auxiliar ← (nuevoNodo → der)          ▷ O(1)
272:                    (nuevoNodo → der) ← (aSwapear → der)                    ▷ O(1)
273:                    (aSwapear → der) ← auxiliar                             ▷ O(1)
274:                end if
275:            end if
276:            (nuevoNodo → izq) ← aSwapear                                    ▷ O(1)
```

```
277:            else
278:                if (nuevoNodo → der) = NULL then                                  ▷ O(1)
279:                    (aSwapear → der) ← NULL                                       ▷ O(1)
280:                else
281:                    (aSwapear → der) ← (nuevoNodo → der)                          ▷ O(1)
282:                    (nuevoNodo → der → padre) ← aSwapear                          ▷ O(1)
283:                end if
284:                if (nuevoNodo → izq) = NULL then                                  ▷ O(1)
285:                    (nuevoNodo → izq) ← (aSwapear → izq)                          ▷ O(1)
286:                    if (aSwapear → izq) ≠ NULL then                               ▷ O(1)
287:                        (aSwapear → izq → padre) ← nuevoNodo                      ▷ O(1)
288:                    end if
289:                    (aSwapear → izq) ← NULL                                       ▷ O(1)
290:                else
291:                    if (aSwapear → izq) = NULL then                               ▷ O(1)
292:                        (aSwapear → izq) ← (nodoNuevo → izq)                      ▷ O(1)
293:                        (nodoNuevo → izq → padre) ← aSwapear                      ▷ O(1)
294:                        (nodoNuevo → izq) ← NULL                                  ▷ O(1)
295:                    else
296:                        (nodoNuevo → izq → padre) ← aSwapear                      ▷ O(1)
297:                        (aSwapear → izq → padre) ← nuevoNodo                      ▷ O(1)
298:                        puntero(Nodoheap) auxiliar ← (nuevoNodo → izq)            ▷ O(1)
299:                        (nuevoNodo → izq) ← (aSwapear → izq)                      ▷ O(1)
300:                        (aSwapear → izq) ← auxiliar                               ▷ O(1)
301:                    end if
302:                end if
303:                (nuevoNodo → der) ← aSwapear                                      ▷ O(1)
304:            end if
305:            if aSwapear = (it.estructura → raiz) then
306:                (aSwapear → padre) ← nuevoNodo                                    ▷ O(1)
307:                (it.estructura → raiz) ← nuevoNodo                                ▷ O(1)
308:                (nuevoNodo → padre) ← NULL                                        ▷ O(1)
309:            else
310:                puntero(Nodoheap) abuelo ← (aSwapear → padre)                     ▷ O(1)
311:                if EsHijoIzquierdo?(aSwapear) then                                ▷ O(1)
312:                    (abuelo → izq) ← nuevoNuevo                                   ▷ O(1)
313:                    (nuevoNodo → padre) ← abuelo                                  ▷ O(1)
314:                    (aSwapear → padre) ← nuevoNodo                                ▷ O(1)
315:                else
316:                    (abuelo → der) ← nuevoNuevo                                   ▷ O(1)
317:                    (nuevoNodo → padre) ← abuelo                                  ▷ O(1)
318:                    (aSwapear → padre) ← nuevoNodo                                ▷ O(1)
319:                end if
320:            end if
321:        end if
322:    end if
323: end if
```

Complejidad: $O(log(EC))$
Justificación: Todas las operaciones son $O(1)$ y en cada uno de los ciclos se va recorriendo la cola desde una de las hojas hasta la raiz (o vice versa). Al ser la cola un árbol completo (ya que siempre se agregan elementos al último lugar disponible), si la cantidad de elementos es EC en el peor cas, entonces su altura es log(EC). Por lo tanto estos ciclos se repiten log(EC) veces en el peor caso.

7.7. Funciones auxiliares

iEsHijoIzquierdo?(**in** p: `puntero(Nodoheap)`) $\rightarrow$ res : bool
▷ Esta función es privada y devuelve true si el nodo apuntado es hijo izquierdo de su padre. Pre: el padre del nodo tiene que ser $\neq$ NULL

1: $res \leftarrow (p = p \rightarrow padre \rightarrow izq)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Todas las operaciones son $O(1)$

---

iMin(**in** izq: `puntero(Nodoheap)`, **in** der: `puntero(Nodoheap)`) $\rightarrow$ res : puntero(Nodoheap)
▷ Esta función es privada y devuelve el nodo con menor cantidad de pokemones capturados, y en caso de empate, devuelve al que tiene menor id.

1: **if** $((izq \rightarrow elemento.cantCapt) < (der \rightarrow elemento.cantCapt)) \vee (((izq \rightarrow elemento.cantCapt) = (der \rightarrow elemento.cantCapt)) \wedge ((izq \rightarrow elemento.id) < (der \rightarrow elemento.id)))$ **then** ▷ $O(1)$
2: $res \leftarrow izq$ ▷ $O(1)$
3: **else**
4: $res \leftarrow der$ ▷ $O(1)$
5: **end if**

Complejidad: $O(1)$
Justificación: Todas las operaciones son $O(1)$

---

iCrearItCola(**in** h: `colaEntr`, **in** p: `puntero(Nodoheap)`) $\rightarrow$ res : itcolaEntrenador)
▷ Función privada que devuelve un iterador a un elemento de la cola de prioridad.

1: $puntero(crearEntr)q \leftarrow h$
2: $res \leftarrow < p, q >$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Todas las operaciones son $O(1)$ Pre: el puntero es $\neq$ NULL, tiene que apuntar a un nodo perteneciente a la estructura de la cola de entrenadores. La cola es no vacía Post: se devuelve un iterador a dicho nodo.

7.8. Servicios usados

De Cola
- EsVacia?(cola(α)) debe ser O(1)

8. TAD Iterador Cola

TAD ITERADOR COLA DE ENTRENADORES(α)

géneros itcola(α)

igualdad observacional
$(\forall it, it' : \text{itcola}(\alpha))\ (it =_{\text{obs}} it' \iff (siguiente(it) =_{\text{obs}} siguiente(it')))$

exporta itcola(α), generadores, observadores

usa CONJUNTO(α), COLA DE PRIORIDAD(α)

colaPrior(α)

observadores básicos

siguiente	: itcola	$\longrightarrow$	α

generadores

crearIt	: colaPrior(α) $\times$ α	$\longrightarrow$	itcola(α)

otras operaciones

borrar	: itcola(α)it $\times$ colaPrior(α)cp	$\longrightarrow$	colaPrior(α)	{siguiente(it) $\in$ elementos(cp)}
elementos	: colaPrior(α)	$\longrightarrow$	conj(α)	
agregarSin	: α $\times$ conj(α)	$\longrightarrow$	colaPrior(α)	

axiomas $\forall cp, sp$: colaPrior(α), $\forall e$: α, $\forall con$: Conj(α)

siguiente(crearIt(cp, e)) $\equiv$ e

borrar(crearIt(cp, e), sp) $\equiv$ agregarSin(e, elementos(sp))

elementos(sp) $\equiv$ **if** vacia?(sp) **then**
∅
else
Ag(proximo(sp), elementos(desencolar(sp)))
fi

agregarSin(e, con)) $\equiv$ **if** ∅?(con) **then**
vacia
else
if dameUno(con) = e **then**
agregarSin(e, sinUno(conj))
else
encolar(dameUno(con), agregarSin(e, sinUno(con)))
fi
fi

Fin TAD