

# 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

<b>1. Informe</b>	<b>3</b>
1.1. General	3
1.2. Algoritmos privados	3
1.3. Juego	3
1.4. IterDiccString	3
1.5. Cola de Prioridad	3
<b>2. Juego</b>	<b>4</b>
2.1. Interfaz	4
2.2. Representacion	6
2.3. Algoritmos	12
2.4. Servicios usados	26
<b>3. Mapa</b>	<b>28</b>
3.1. Interfaz	28
3.2. Representacion	29
3.3. Algoritmos	31
3.4. Servicios usados	34
<b>4. Coordenada</b>	<b>36</b>
4.1. Interfaz	36
4.2. Representacion	37
4.3. Algoritmos	37
<b>5. DiccString(<math>\alpha</math>)</b>	<b>39</b>
5.1. Interfaz	39
5.2. Representacion	40
5.3. Algoritmos	40
5.4. Servicios usados	43
<b>6. iterDiccString(<math>\alpha</math>)</b>	<b>45</b>
6.1. Interfaz	45
6.2. Representacion del iterDiccString	45
6.3. Algoritmos	46
6.4. Servicios usados	46
<b>7. Cola de Entrenadores</b>	<b>48</b>
7.1. Interfaz de Cola de Entrenadores	48
7.2. Interfaz del iterador	48
7.3. Representacion de Cola de Entrenadores	49
7.4. Representacion del iterador	49
7.5. Algoritmos Cola de Entrenadores	50
7.6. Algoritmos del iterador	53
7.7. Funciones auxiliares	59
7.8. Servicios usados	59
<b>8. TAD Iterador Cola</b>	<b>60</b>

## 1. Informe

### 1.1. General

- Intentamos dar una explicación de las estructuras que fuimos construyendo, que pueden leerse en la sección “representación” de cada módulo. En esos lugares contamos un poco mas sobre las decisiones particulares de cada estructura.

### 1.2. Algoritmos privados

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

### 1.3. Juego

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

### 1.4. 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)

### 1.5. Cola de Prioridad

- Sabemos que no era la única forma de mantener a los entrenadores que esperan, podiamos usar un AVL y las complejidades seguirían valiendo. Elegimos implementarlo con un Heap porque era mas fácil de implementar, o eso creímos..
- En la cola de entrenadores, se usan nodos y punteros para la estructura. Los nodos se mantienen “fijos” una vez que se agregan, 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.

## 2. Juego

### Interfaz

#### 2.1. Interfaz

se explica con: JUEGO.

géneros: juego.

#### Operaciones básicas de Juego

**CREARJUEGO**(in  $m$ : map)  $\rightarrow res$  : juego

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{crearJuego}(m)\}$

**Complejidad**:  $O((\text{tam}(m))^2)$

**Descripción**: Genera una juego con el mapa  $m$  y sin jugadores.

**AGREGARPOKÉMON**(in  $p$ : pokemon, in  $c$ : coord, in/out  $j$ : juego)

**Pre**  $\equiv \{j_0 =_{\text{obs}} j \wedge \text{posExistente}(c, \text{mapa}(j)) \wedge p \notin \text{pokemones}(j) \wedge \text{PuedoAgregarPokemon}(c, j)\}$

**Post**  $\equiv \{j =_{\text{obs}} \text{agregarPokemon}(p, c, j_0) \wedge p \in \text{pokemones}(j)\}$

**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**  $\equiv \{res =_{\text{obs}} \text{ProxId}(j_0) \wedge j =_{\text{obs}} \text{agregarJugador}(j_0)\}$

**Complejidad**:  $O(J)$

**Descripción**: Agrega un jugador al juego  $j$  con id igual a  $\text{ProxId}(j)$ .

**CONECTARSE**(in  $e$ : jugador, in  $c$ : coor, in/out  $j$ : juego)

**Pre**  $\equiv \{j_0 =_{\text{obs}} j \wedge e \in \text{jugadores}(j) \wedge_{\text{L}} \neg \text{estaConectado}(e, j) \wedge \text{posExistente}(c, \text{mapa}(j))\}$

**Post**  $\equiv \{j =_{\text{obs}} \text{conectarse}(e, c, j_0)\}$

**Complejidad**:  $O(\log(EC))$

**Descripción**: Conecta al jugador  $e$  en la posicion  $c$ .

**DESCONECTARSE**(in  $e$ : jugador, in/out  $j$ : juego)

**Pre**  $\equiv \{j_0 =_{\text{obs}} j \wedge e \in \text{jugadores}(j) \wedge_{\text{L}} \text{estaConectado}(e, j)\}$

**Post**  $\equiv \{j =_{\text{obs}} \text{desconectarse}(e, j_0)\}$

**Complejidad**:  $O(\log(EC))$

**Descripción**: Desconecta al jugador  $e$  del juego.

**MOVESE**(in  $e$ : jugador, in  $c$ : coor, in/out  $j$ : juego)

**Pre**  $\equiv \{j_0 =_{\text{obs}} j \wedge e \in \text{jugadores}(j) \wedge_{\text{L}} \text{estaConectado}(e, j) \wedge \text{posExistente}(c, \text{mapa}(j))\}$

**Post**  $\equiv \{j =_{\text{obs}} \text{moverse}(e, c, j_0)\}$

**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**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res = \text{mapa}(j)\}$

**Complejidad**:  $O(1)$

**Descripción**: Devuelve el mapa del juego.

**Aliasing**: Es por referencia, produce aliasing.

**JUGADORES**(in  $j$ : juego)  $\rightarrow res$  : itConj(jugador)

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{jugadores}(j)\}$

**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**  $\equiv \{e \in \text{jugadores}(j)\}$

**Post**  $\equiv \{res = \text{estaConectado}(e, j)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve true si el jugador esta conectado.

SANCIONES(**in**  $e$ : jugador, **in**  $j$ : juego)  $\rightarrow res$  : nat

**Pre**  $\equiv \{e \in \text{jugadores}(j)\}$

**Post**  $\equiv \{res = \text{sanciones}(e, j)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la cantidad de sanciones de un jugador.

POSICION(**in**  $j$ : juego, **in**  $e$ : jugador)  $\rightarrow res$  : coor

**Pre**  $\equiv \{e \in \text{jugadores}(j) \wedge_L \text{estaConectado}(e, j)\}$

**Post**  $\equiv \{res = \text{posicion}(e, j)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la posicion actual de un jugador.

POKEMONS(**in**  $j$ : juego, **in**  $e$ : jugador)  $\rightarrow res$  : iterDiccString(nat)

**Pre**  $\equiv \{e \in \text{jugadores}(j)\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{crearIt}(j.\text{jugadores}[e].\text{pokemons})\}$

**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**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res = \text{expulsados}(j)\}$

**Complejidad:**  $O(J)$

**Descripción:** Devuelve un conjunto con los jugadores expulsados.

POSCONPOKEMONS(**in**  $j$ : juego)  $\rightarrow res$  : conj(coor)

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{posConPokemons}(j))\}$

**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**  $\equiv \{c \in \text{posConPokemons}(j)\}$

**Post**  $\equiv \{res = \text{pokemonEnPos}(c, j)\}$

**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**  $\equiv \{c \in \text{posConPokemons}(j)\}$

**Post**  $\equiv \{res = \text{cantMovimientosParaCaptura}(c, j)\}$

**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**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res = \text{puedoAgregarPokemon}(c, j)\}$

**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**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res = \text{hayPokemonCercano}(c, j)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve verdadero si hay algun pokemon en el territorio.

**POSPOKEMONCERCANO**(in  $c$ : **coor**, in  $j$ : **juego**)  $\rightarrow res$  : **coor**

**Pre**  $\equiv \{\text{hayPokemonCercano}(c, j)\}$

**Post**  $\equiv \{res = \text{posPokemonCercano}(c, j)\}$

**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**  $\equiv \{\text{hayPokemonCercano}(c, j) \wedge es \subset \text{jugadoresConectados}(j)\}$

**Post**  $\equiv \{res = \text{entrenadoresPosibles}(c, es, j)\}$

**Complejidad:**  $O(\text{Longitud}(ec) * EC)$

**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**  $\equiv \{p \in \text{todosLosPokemons}(j)\}$

**Post**  $\equiv \{res = \text{indiceRareza}(p, j)\}$

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

**Descripción:** Devuelve el indice de rareza del pokemon dado.

**CANTPOKEMONSTOTALES**(in  $j$ : **juego**)  $\rightarrow res$  : **nat**

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res = \text{cantPokemonsTotales}(p, j)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la cantidad de pokemons totales del juego.

**CANTMISMAESPECIE**(in  $p$ : **pokemon**, in  $j$ : **juego**)  $\rightarrow res$  : **nat**

**Pre**  $\equiv \{p \in \text{todosLosPokemons}(j)\}$

**Post**  $\equiv \{res = \text{cantMismaEspecie}(p, j)\}$

**Complejidad:**  $O(|p|)$

**Descripción:** Devuelve la cantidad de pokemons totales del juego.

## Representación

### 2.2. 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 preocuparnos 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 posición del jugador en ese conjunto: “iterAJuego”.

Los pokemons que capturó están representados por un diccionario, donde para cada pokemon capturado dice la cantidad capturada de esa especie. “cantCapt” es la cantidad total de pokemons capturados. Puede obtenerse sumando la cantidad capturada de cada pokemon, pero decidimos mantenerla así para poder armar rápidamente 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 da 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 habría 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 habría 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 posición 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 jugador en ese indice (de esto se desprende que los ids son unicos)

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

(1) El jugador  $e$  esta en `j.jugadoresNoEliminados` sii no esta eliminado

$(\forall e: \text{jugador})(e \in j.\text{jugadoresNoEliminados} \iff (e < \text{longitud}(j.\text{jugadores}) \wedge j.\text{jugadores}[e].\text{sanciones} < 5))$

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

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

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

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

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

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

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

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

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

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



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

$$(\forall jug : jugStruc) ( esJugadorConectado(jug, j) \Rightarrow_L Siguiente(jug.iteradorAPos) = jug.id )$$

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

$$(\forall jug : jugStruc) (esJugadorNoEliminado(jug, j) \Rightarrow_L jug.cantCapt = sumaSignif(jug.pokemons))$$

(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)

$$(\forall p : pokemon) ((p \in Claves(j.cantPokemon)) \Rightarrow_L$$

$$Obtener(p, j.cantPokemon) = cantSalvajes(p, j) + sumaPokesCapturados(p, j))$$

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

$$(\forall p : pokemon) ((p \in Claves(j.cantPokemon)) \Rightarrow_L$$

$$(cantSalvajes(p, j) = Obtener(p, j.cantPokemon) - sumaPokesCapturados(p, j)))$$

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

$$longitud(j.pokenodos) = tam(j.mapa) \wedge_L (\forall i: nat) ((i < Logitud(j.pokenodos)) \Rightarrow Longitud(j.pokenodos[i]) = Tam(j.mapa))$$

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

$$(\forall x, y: nat) ((enRango(x, y, j.mapa) \wedge_L j.pokenodos[x][y] \neq NULL)$$

$$\Rightarrow_L (posExistente(crearCoordenada(x, y), j.mapa) \wedge crearCoordenada(x, y) \in j.posPokemons)) \wedge$$

$$(\forall c: coord) ((c \in j.posPokemons) \Rightarrow_L (enRango(latitud(c), longitud(c), j.mapa)$$

$$\wedge_L j.pokenodos[latitud(c)][longitud(c)] \neq NULL))$$

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

$$(\forall x, y: nat) ((enRango(x, y, j.mapa) \wedge_L j.pokenodos[x][y] \neq NULL) \wedge$$

$$(\forall z, w: nat) ((enRango(z, w, j.mapa) \wedge_L j.pokenodos[z][w] \neq NULL) \wedge$$

$$(x \neq z \wedge y \neq w) \Rightarrow_L distEuclidea(crearCoordenada(x, y), crearCoordenada(z, w)) > 5)$$

(15) El contador de todo pokenodo es < 10

$$(\forall x, y: nat) ((enRango(x, y, j.mapa) \wedge_L j.pokenodos[x][y] \neq NULL)$$

$$\Rightarrow_L (* (j.pokenodos[x][y])).contador < 10)$$

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

(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

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

(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

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

$\text{esJugadorConectado} : \text{jugStruc } jug \times \text{juego } j \longrightarrow \text{bool}$   
 $\text{esJugadorConectado}(jug, j) \equiv jug \in j.\text{jugadores} \wedge_L jug.\text{conectado}$

$\text{esJugadorNoEliminado} : \text{jugStruc } jug \times \text{juego } j \longrightarrow \text{bool}$   
 $\text{esJugadorNoEliminado}(jug, j) \equiv jug \in j.\text{jugadores} \wedge_L jug.\text{sanciones} < 5$

$\text{enRango} : \text{nat } x \times \text{nat } y \times \text{juego } j \longrightarrow \text{bool}$   
 $\text{enRango}(x, y, j) \equiv \text{posExistente}(\text{crearCoordenada}(x, y), j.\text{mapa})$

$\text{sumaSignif} : \text{dicc}(\text{string} \times \text{nat}) \longrightarrow \text{nat}$   
 $\text{sumaSignif}(d) \equiv \text{sumaSignifAux}(\text{claves}(d), d)$

$\text{sumaSignifAux} : \text{conj}(\text{string}) \times \text{dicc}(\text{string} \times \text{nat}) \longrightarrow \text{nat}$   
 $\text{sumaSignifAux}(cs, d) \equiv \text{if } \emptyset?(cs) \text{ then } 0 \text{ else } \text{obtener}(\text{dameUno}(cs), d) + \text{sumaSignifAux}(\text{sinUno}(cs), d) \text{ fi}$

$\text{cantSalvajes} : \text{pokemon } p \times \text{juego } j \longrightarrow \text{nat}$   
 $\text{cantSalvajes}(p, j) \equiv \#(p, \text{pokemonsSalvajes}(\text{posConPokemons}(j)))$

sumaPokesCapturados : pokemon  $p \times$  juego  $j \rightarrow \text{nat}$

sumaPokesCapturados( $p, j$ )  $\equiv$  sumaPokesCapturadosAux( $p, j, \text{jugadoresConectados}(j)$ )

sumaPokesCapturadosAux : pokemon  $p \times$  juego  $j \times \text{conj}(\text{jugador}) \text{ } js \rightarrow \text{nat}$

```

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

```

Rep : juego  $j \rightarrow \text{bool}$

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

Abs : juego  $j \rightarrow \text{Juego}$

{Rep( $j$ )}

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

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

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

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

## Algoritmos

### 2.3. 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 ← 0 to Tam(map) − 1 do                                    ▷  $O((Tam(m))^2)$ 
8:   vector(lista(jugador)) vectorInterno ← Vacio()              ▷  $O(1)$ 
9:   for j ← 0 to Tam(map) − 1 do                                    ▷  $O((Tam(m)))$ 
10:    lista(jugador) jugsVacía ← Vacía()                          ▷  $O(1)$ 
11:    AgregarAtras(vectorInterno, jugsVacía)                      ▷  $O(1)$  amortizado
12:  end for
13:  AgregarAtras(jugs, vectorInterno)                            ▷  $O(1)$  amortizado
14: end for

14: vector(vector(pokeStruc)) pokenodos ← Vacio()                ▷  $O(1)$ 
15: for i ← 0 to Tam(map) − 1 do                                    ▷ Se repite Tam(map) veces  $O(1)$ 
16:   vector(puntero(pokeStruc)) vectorInterno ← Vacio()          ▷  $O(1)$ 
17:   for j ← 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:  $O((Tam(map))^2)$ 
Justificación: 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 vacíos 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 ← Vacía()                              ▷  $O(1)$ 
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:  $j.jugadores[j].conectado \leftarrow \text{true}$   $\triangleright O(1)$ 
2:  $j.jugadores[e].iteradorAPos \leftarrow \text{AgregarAtras}(j.\text{grillaJugs}[\text{latitud}(c)][\text{longitud}(c)], e)$   $\triangleright O(1)$ 
3:  $j.jugadores[e].pos \leftarrow c$ 
4: if HayPokemonCercano( $c, j$ ) then
5:   if HayCamino( $c, \text{PosPokemonCercano}(c, j), \text{Mapa}(j)$ ) then  $\triangleright O(1)$ 
6:      $\text{nat } latPok \leftarrow \text{latitud}(\text{PosPokemonCercano}(c, j))$   $\triangleright O(1)$ 
7:      $\text{nat } lonPok \leftarrow \text{longitud}(\text{PosPokemonCercano}(c, j))$   $\triangleright O(1)$ 
8:      $(j.\text{pokenodos}[latPok][lonPok] \rightarrow \text{contador}) \leftarrow 0$   $\triangleright O(1)$ 
9:      $\text{tupla } \langle \text{nat}, \text{nat} \rangle t \leftarrow \langle e, j.jugadores[e].cantCap \rangle$ 
10:     $j.jugadores[e].iteradorAEntrenadores \leftarrow \text{Encolar}((j.\text{pokenodos}[latPok][lonPok] \rightarrow \text{entrenadores}), t)$   $\triangleright$ 
     $O(\log(EC))$ 
11:   end if
12: end if

```

13: Complejidad:  $O(\log(EC))$

14: Justificación: 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:  $j.jugadores[e].conectado \leftarrow \text{false}$   $\triangleright O(1)$ 
2: EliminarSiguiente( $j.jugadores[e].iteradorAPos$ )  $\triangleright O(1)$ 
3:  $c \leftarrow j.jugadores[e].pos$ 
4: if HayPokemonCercano( $c, j$ )  $\wedge$  HayCamino( $c, \text{PosPokemonCercano}(c, j), \text{Mapa}(j)$ ) then  $\triangleright O(1)$ 
5:    $\text{nat } latPok \leftarrow \text{latitud}(\text{PosPokemonCercano}(c, j))$   $\triangleright O(1)$ 
6:    $\text{nat } lonPok \leftarrow \text{longitud}(\text{PosPokemonCercano}(c, j))$   $\triangleright O(1)$ 
7:   Borrar( $j.jugadores[e].iteradorAEntrenadores, (j.\text{pokenodos}[latPok][lonPok] \rightarrow \text{entrenadores})$ )  $\triangleright O(\log(EC))$ 
8: end if

```

9: Complejidad:  $O(\log(EC))$

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

```

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

22: Justificación: Las operaciones de asignación, acceso a posiciones y desreferenciación de punteros son  $O(1)$ . La funciones 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 iteración 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  $EC \leq EC * \log(EC)$ )  $= O(|P| + EC * \log(EC))$

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

```

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

```

```

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

24:   if hayPokDesp then
25:     if ¬ hayPokAntes then
26:       CasoMov3(e, posAntes, c, j)                                          ▷ O((PS * |P|) + log(EC))
27:     else
28:       if PosPokemonCercano(posAntes, j) = PosPokemonCercano(c, j) then    ▷ O(1)
29:         CasoMov1(e, posAntes, c, j)                                       ▷ O(PS * |P|)
30:       else
31:         CasoMov5(e, posAntes, c, j)                                       ▷ O((PS * |P|) + log(EC))
32:       end if
33:     end if
34:   else
35:     if hayPokAntes then
36:       CasoMov2(e, posAntes, c, j)                                          ▷ O((PS * |P|) + log(EC))
37:     else
38:       CasoMov4(e, posAntes, c, j)                                          ▷ O(PS * |P|)
39:     end if
40:   end if

```

```

41:   Borrar(j.jugadores[e].iterAPos)                                          ▷ O(1)
42:   j.jugadores[e].iterAPos ← AgregarAtras(j.grillaJugs[Latitud(c)][Longitud(c)]) ▷ O(1)
43:   j.jugadores[e].pos ← c                                                  ▷ O(1)
44: end if

```

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

46: 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:  $j = j_0 \wedge \text{hayPokemonCercano}(\text{ant}, j) \wedge \text{hayPokemonCercano}(\text{desp}, j) \wedge_L \text{posPokemonCercano}(\text{ant}, j) = \text{posPokemonCercano}(\text{desp}, j)$

Post:  $j_0 = \text{move}(\text{e}, \text{desp}, j)$

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 HaySiguiente(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( $j.pokenodos[x][y] \rightarrow poke$ ,  $Proximo(j.pokenodos[x][y] \rightarrow entrenadores)$ )  $\triangleright O(|P|)$ 
11: EliminarSiguiente( $iterPos$ )  $\triangleright O(1)$ 
12:  $j.pokenodos[x][y] \leftarrow NULL$   $\triangleright$  Libero la memoria  $O(1)$ 
13: else
14: Avanzar( $iterPos$ )  $\triangleright O(1)$ 
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( $j.jugadores[e].iterAEntrenadores$ )  $\triangleright O(\log(EC))$ 
2:  $itConj\ iterPos \leftarrow CrearIter(j.posPokemons)$   $\triangleright O(1)$ 
3: while HaySiguiente( $iterPos$ ) do  $\triangleright O(PS * |P|)$ 
4:    $nat\ x \leftarrow Latitud(Siguiente(iterPos))$   $\triangleright O(1)$ 
5:    $nat\ y \leftarrow Longitud(Siguiente(iterPos))$   $\triangleright O(1)$ 
6:    $(j.pokenodos[x][y] \rightarrow contador \leftarrow ((j.pokenodos[x][y] \rightarrow contador) + 1)$   $\triangleright O(1)$ 
7:   if  $j.pokenodos[x][y] \rightarrow contador = 10$  then
8:     SumarUnoEnJug( $j.pokenodos[x][y] \rightarrow poke$ ,  $Proximo(j.pokenodos[x][y] \rightarrow entrenadores)$ )  $\triangleright O(|P|)$ 
9:     EliminarSiguiente( $iterPos$ )  $\triangleright O(1)$ 
10:     $j.pokenodos[x][y] \leftarrow NULL$   $\triangleright$  Libero la memoria  $O(1)$ 
11:   else
12:     Avanzar( $iterPos$ )  $\triangleright O(1)$ 
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 HaySiguiente(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:  $j = j_0 \wedge \neg \text{hayPokemonCercano}(ant, j) \wedge \neg \text{hayPokemonCercano}(desp, j)$

Post:  $j_0 = \text{moverse}(e, desp, j)$

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

Aliasing: Produce aliasing, el juego es modificable

```

1: itConj iterPos ← CrearIter(j.posPokemons)                                ▷  $O(1)$ 
2: while HaySiguiente(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 hacia las cercanias de otro pokemon distinto

Pre:  $j = j_0 \wedge \text{hayPokemonCercano}(ant, j) \wedge \neg \text{hayPokemonCercano}(desp, j) \wedge_L \text{posPokemonCercano}(ant, j) \neq \text{posPokemonCercano}(desp, j)$

Post:  $j_0 = \text{moverse}(e, desp, j)$

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

Aliasing: Produce aliasing, el juego es modificable

```

1: Eliminar( $j$ .jugadores[ $e$ ].iterAEntrenadores)                                ▷  $O(\log(EC))$ 
2: coor  $pokePos \leftarrow \text{PosPokemonCercano}(desp, j)$                         ▷  $O(1)$ 
3:  $j$ .jugadores[ $e$ ].iterAEntrenadores ← Encolar(
4:      $j$ .pokenodos[ $\text{Latitud}(pokePos)$ ][ $\text{Longitud}(pokePos)$ ],  $\langle e, j$ .jugadores[ $e$ ].cantCapt >
5: )                                                                            ▷  $O(\log(EC))$ 
6: itConj  $iterPos \leftarrow \text{CrearIter}(j$ .posPokemons)                        ▷  $O(1)$ 
7: while HaySiguiente( $iterPos$ ) do                                           ▷  $O(PS * |P|)$ 
8:   nat  $x \leftarrow \text{Latitud}(\text{Siguiente}(iterPos))$                         ▷  $O(1)$ 
9:   nat  $y \leftarrow \text{Longitud}(\text{Siguiente}(iterPos))$                       ▷  $O(1)$ 
10:  if Siguiente( $iterPos$ )  $\neq pokepos$  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:  $res \leftarrow j$ .cantPokemonsTotales                                     ▷  $O(1)$ 
2: Complejidad:  $O(1)$ 

```

---

---



---

**iIndiceRareza**(in  $p$ : pokemon, in  $j$ : juego)  $\rightarrow res$ : nat

- 1: nat  $pokecant \leftarrow \text{Obtener}(j.\text{cantPokemon}, p)$   $\triangleright O(|P|)$
  - 2:  $res \leftarrow 100 - (100 * pokecant / j.\text{cantPokemonsTotales})$   $\triangleright O(1)$
  - 3: Complejidad:  $O(|P|)$
  - 4: Justificación:  $j.\text{cantPokemon}$  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  $O(|P|)$ .  $j.\text{cantPokemonsTotales}$  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)  $\rightarrow res$ : conj(coor)

- 1:  $res \leftarrow j.\text{posPokemons}$   $\triangleright$  Por referencia  $O(1)$
  - Complejidad:  $O(1)$
- 

---



---

**iPokemonEnPos**(in  $j$ : juego, in  $c$ : coor)  $\rightarrow res$ : pokemon

- 1:  $res \leftarrow ((j.\text{pokenodos}[\text{Latitud}(c)][\text{Longitud}(c)]) \rightarrow \text{poke})$   $\triangleright O(1)$
  - Complejidad:  $O(1)$
- 

---



---

**iPosicion**(in  $j$ : juego, in  $e$ : jugador)  $\rightarrow res$ : coor

- 1:  $res \leftarrow j.\text{jugadores}[e].\text{pos}$   $\triangleright O(1)$
  - 2: Complejidad:  $O(1)$
  - 3: Justificación: Todas las operaciones son  $O(1)$
- 

---



---

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

- 1:  $res \leftarrow j.\text{jugadores}[e].\text{sanciones}$   $\triangleright O(1)$
  - 2: Complejidad:  $O(1)$
  - 3: Justificación: Todas las operaciones son  $O(1)$
- 

---



---

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

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

**iHayPokemonCercano**(in  $c$ : coor, in  $j$ : juego)  $\rightarrow res$ : coor

```

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

```

```

55:   end if
56:   if  $y < tam(m) - 1$  then
57:       if  $j.pokenodos[x + 1][y + 1] \neq \text{NULL}$  then  $\triangleright O(1)$ 
58:            $hayPokemon \leftarrow \text{true}$ 
59:       end if
60:   end if
61: end if
62: if  $tam(m) > 1 \wedge x < tam(m) - 2$  then
63:     if  $j.pokenodos[x + 2][y] \neq \text{NULL}$  then  $\triangleright O(1)$ 
64:          $hayPokemon \leftarrow \text{true}$ 
65:     end if
66: end if
67:  $res \leftarrow hayPokemon$ 
68: Complejidad:  $O(1)$ 
69: Justificacion: Reviso 13 posiciones  $O(1)$ 

```

**iPosPokemonCercano**(in  $c$ : coor, in  $j$ : juego)  $\rightarrow res$ : coor

```

1: nat  $x \leftarrow \text{latitud}(c)$   $\triangleright O(1)$ 
2: nat  $y \leftarrow \text{longitud}(c)$   $\triangleright O(1)$ 
3: coor  $coorConPokemon$   $\triangleright O(1)$ 
4: if  $j.pokenodos[x][y] \neq \text{NULL}$  then  $\triangleright O(1)$ 
5:      $coorConPokemon \leftarrow \text{CrearCoor}(x, y)$   $\triangleright O(1)$ 
6: end if
7: if  $x > 0$  then
8:     if  $j.pokenodos[x - 1][y] \neq \text{NULL}$  then  $\triangleright O(1)$ 
9:          $coorConPokemon \leftarrow \text{CrearCoor}(x - 1, y)$   $\triangleright O(1)$ 
10:    end if
11:    if  $y > 0$  then
12:        if  $j.pokenodos[x - 1][y - 1] \neq \text{NULL}$  then  $\triangleright O(1)$ 
13:             $coorConPokemon \leftarrow \text{CrearCoor}(x - 1, y - 1)$   $\triangleright O(1)$ 
14:        end if
15:    end if
16:    if  $y < tam(m) - 1$  then
17:        if  $j.pokenodos[x - 1][y + 1] \neq \text{NULL}$  then  $\triangleright O(1)$ 
18:             $coorConPokemon \leftarrow \text{CrearCoor}(x - 1, y + 1)$   $\triangleright O(1)$ 
19:        end if
20:    end if
21:    if  $x - 1 > 0$  then
22:        if  $j.pokenodos[x - 2][y] \neq \text{NULL}$  then  $\triangleright O(1)$ 
23:             $coorConPokemon \leftarrow \text{CrearCoor}(x - 1, y)$   $\triangleright O(1)$ 
24:        end if
25:    end if
26: end if
27: if  $y > 0$  then
28:     if  $j.pokenodos[x][y - 1] \neq \text{NULL}$  then  $\triangleright O(1)$ 
29:          $coorConPokemon \leftarrow \text{CrearCoor}(x, y - 1)$   $\triangleright O(1)$ 
30:     end if
31:     if  $y - 1 > 0$  then
32:         if  $j.pokenodos[x][y - 2] \neq \text{NULL}$  then  $\triangleright O(1)$ 
33:              $coorConPokemon \leftarrow \text{CrearCoor}(x, y - 2)$   $\triangleright O(1)$ 
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:       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)  $\rightarrow res$ : conj(jugador)

```

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

```

5: Complejidad:  $O(J)$

6: Justificación: 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)  $\rightarrow res$ : iterDiccString(nat)

```

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

```

2: Complejidad:  $O(1)$

3: Justificación: Devuelvo un iterador en  $O(1)$

---



---

**iCantMismaEspecie**(in  $p$ : pkemon, in  $j$ : juego)  $\rightarrow res$ : nat

```

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

```

Complejidad:  $O(|p|)$

Justificación: La unica operacion es  $O(|p|)$

---



---

**iExpulsados**(in  $j$ : juego)  $\rightarrow res$ : conj(jugador)

```

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

```

Complejidad:  $O(J)$

Justificación: Siendo  $J$  la cantidad total de jugadores, se los recorre todos buscando los expulsados.

---



---

**iMovValido**(in  $e$ : jugador, in  $c$ : coor, in  $j$ : juego)  $\rightarrow res$ : bool

MovValido : Función privada

Descripción : Dice si el movimiento entre la posicion del jugador y la nueva coordenada  $c$ , es valido

Pre:  $e \in jugadoresConctados(j) \wedge posExistente(c, j.mapa)$

Post:  $res = hayCamino(c, j.jugadores[e].pos, j.mapa) \wedge$   
 $DistEuclidea(c, j.jugadores[e].pos) \leq 100$

Complejidad:  $O(1)$

```

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

```

Complejidad:  $O(1)$

Justificación: 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 jugadorPre:  $j = j_0 \wedge e \in \text{jugadores}(j)$ Post:  $\#pokemons(p, j_0.\text{jugadores}[e].pokemons) = \#pokemons(p, j.\text{jugadores}[e].pokemons) + 1$ Complejidad:  $O(|P|)$ 

```

1: if Def?( $p, j.\text{cantPokemon}$ ) then  $\triangleright O(|P|)$ 
2:   Definir( $j.\text{cantPokemon}, p, \text{Obtener}(p, j.\text{cantPokemon}) + 1$ )  $\triangleright O(|P|)$ 
3: else
4:   Definir( $j.\text{cantPokemon}, p, 1$ )  $\triangleright O(|P|)$ 
5: end if

```

Complejidad:  $O(|P|)$ Justificación: 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:  $\text{posValida}(c, j.\text{mapa})$ Post:  $res =_{\text{obs}} \text{entrenadoresPosibles}(c, \text{jugadores}(j), j)$ Complejidad:  $O(EC)$ Aliasing: Devuelve el vector por referencia

```

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

```



```

31:   end if
32:   if  $y - 1 > 0$  then
33:       if  $j.pokenodos[x][y - 2] \neq \text{NULL}$  then  $\triangleright O(1)$ 
34:           AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada( $x, y - 2$ ),  $c, j$ )  $\triangleright O(EC)$ 
35:       end if
36:   end if
37: end if
38: if  $y < tam(m) - 1$  then
39:     if  $j.pokenodos[x][y + 1] \neq \text{NULL}$  then  $\triangleright O(1)$ 
40:         AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada( $x, y + 1$ ),  $c, j$ )  $\triangleright O(EC)$ 
41:     end if
42:     if  $tam(m) > 1 \wedge y < tam(m) - 2$  then
43:         if  $j.pokenodos[x][y + 2] \neq \text{NULL}$  then  $\triangleright O(1)$ 
44:             AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada( $x, y + 2$ ),  $c, j$ )  $\triangleright O(EC)$ 
45:         end if
46:     end if
47: end if
48: if  $x < tam(m) - 1$  then
49:     if  $j.pokenodos[x + 1][y] \neq \text{NULL}$  then  $\triangleright O(1)$ 
50:         AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada( $x + 1, y$ ),  $c, j$ )  $\triangleright O(EC)$ 
51:     end if
52:     if  $y > 0$  then
53:         if  $j.pokenodos[x + 1][y - 1] \neq \text{NULL}$  then  $\triangleright O(1)$ 
54:             AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada( $x + 1, y - 1$ ),  $c, j$ )  $\triangleright O(EC)$ 
55:         end if
56:     end if
57:     if  $y < tam(m) - 1$  then
58:         if  $j.pokenodos[x + 1][y + 1] \neq \text{NULL}$  then  $\triangleright O(1)$ 
59:             AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada( $x + 1, y + 1$ ),  $c, j$ )  $\triangleright O(EC)$ 
60:         end if
61:     end if
62: end if
63: if  $tam(m) > 1 \wedge x < tam(m) - 2$  then
64:     if  $j.pokenodos[x + 2][y] \neq \text{NULL}$  then  $\triangleright O(1)$ 
65:         AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada( $x + 2, y$ ),  $c, j$ )  $\triangleright O(EC)$ 
66:     end if
67: end if
68:  $res \leftarrow jugsRadio$   $\triangleright$  Por referencia  $O(1)$ 
69: Complejidad:  $O(EC)$ 
70: Justificación: En peor caso agrego jugadores de 13 posiciones, donde ese agregar me cuesta  $O(EC)$  cada vez.  $O(EC)$ 
    +  $O(EC) + \dots + O(EC) = O(EC)$ 

```

---

**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 posición, al vector pasado por parametros,

Pre:  $jugs = jugs_0 \wedge posValida(posJug, j.mapa) \wedge posValida(posPoke, j.mapa)$

Post:  $jugs = jugs_0 \text{ con los jugadores en condiciones de capturar agregados atras}$

Complejidad:  $O(EC)$

```

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

Complejidad:  $O(EC)$

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

---

## 2.4. 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( $\alpha$ )) debe ser  $O(1)$
- Avanzar(itLista( $\alpha$ )) debe ser  $O(1)$

De Vector

- AgregarAtras( vector( $\alpha$ ),  $\alpha$ ) debe ser  $O(f(long(v)) + copy(a))$

De Cola

- Encolar(colaEntr, entrenador) debe ser  $O(\log(EC))$

De Diccionario

- iVacio() debe ser  $O(1)$  - Borrar( string, diccString( $\alpha$ )) debe ser  $O(|P|)$
- Def?(string, diccString( $\alpha$ )) debe ser  $O(|P|)$
- Definir(diccString( $\alpha$ ), string,  $\alpha$ ) debe ser  $O(|P|)$
- Obtener(string, diccString( $\alpha$ )) debe ser  $\alpha$   $O(|P|)$

De Conjunto Lineal

- AgregarRapido( conj( $\alpha$ ),  $\alpha$ ) debe ser  $O(copy(a))$
- EliminarSiguiente(itConj( $\alpha$ )) debe ser  $O(1)$
- HaySiguiente( itConjAcotado) debe ser  $O(1)$

- Avanzar( itConj( $\alpha$ )) debe ser  $O(1)$
- Siguiente(tConj( $\alpha$ )) debe ser  $O(1)$

### 3. Mapa

#### Interfaz

##### 3.1. Interfaz

se explica con: MAPA.

géneros: map.

#### Operaciones básicas de Mapa

CREARMAPA()  $\rightarrow res : \text{map}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{crearMapa}\}$

**Complejidad:**  $O(1)$

**Descripción:** Genera una mapa vacío.

AGREGARCOOR( **in**  $c : \text{coord}$ , **in/out**  $m : \text{map}$  )

**Pre**  $\equiv \{m_0 =_{\text{obs}} m \wedge \neg \text{posExistente}(c, m_0)\}$

**Post**  $\equiv \{m =_{\text{obs}} \text{agregarCoor}(c, m_0)\}$

**Complejidad:**  $O(\max(\text{latitud}(c), \text{longitud}(c), \text{tam}(m))^4)$

**Descripción:** Agrega la coordenada  $c$  al mapa  $m$ .

COORDENADAS(**in**  $m : \text{map}$ )  $\rightarrow res : \text{conj}(\text{coord})$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{coordenadas}(m)\}$

**Complejidad:**  $O((\text{tam}(m))^2)$

**Descripción:** Devuelve el conjunto de todas las coordenadas del mapa  $m$ .

POSEXISTENTE( **in**  $c : \text{coord}$ , **in**  $m : \text{map}$  )  $\rightarrow res : \text{bool}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{posExistente}(c, m)\}$

**Complejidad:**  $O(1)$

**Descripción:** Verifica si la coordenada  $c$  existe en el mapa  $m$ .

HAYCAMINO( **in**  $c_1 : \text{coord}$ , **in**  $c_2 : \text{coord}$ , **in**  $m : \text{map}$  )  $\rightarrow res : \text{bool}$

**Pre**  $\equiv \{\text{posExistente}(c_1, m) \wedge \text{posExistente}(c_2, m)\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{hayCamino}(c_1, c_2, m)\}$

**Complejidad:**  $O(1)$

**Descripción:** verifica si existe una forma de llegar desde  $c_1$  a  $c_2$ .

TAM( **in**  $m : \text{map}$  )  $\rightarrow res : \text{nat}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{tam}(m)\}$

**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

$\text{tam} : \text{mapa } m \longrightarrow \text{nat}$

$\text{max} : \text{nat } x \times \text{nat } y \longrightarrow \text{nat}$

```

maxLatitud : conj(coor) cs  → nat                                {¬∅?(cs)}
maxLongitud : conj(coor) cs  → nat                                {¬∅?(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
            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
            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 así para que no haya que distinguir entre tamaño de ancho o de alto, así es más sencillo escribir los algoritmos y los cálculos 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 útil 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  $\text{grilla}[x][y][z][w]$  vale tambien  $\text{grilla}[z][w][x][y]$   
 (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)  $((\forall i: \text{nat})(i < \text{long}(\text{m.grilla})) \Rightarrow_L \text{long}(\text{m.grilla}[i]) = \text{long}(\text{m.grilla})) \wedge_L$   
 $((\forall i, j: \text{nat})(i < \text{long}(\text{m.grilla}) \wedge j < \text{long}(\text{m.grilla})) \Rightarrow_L \text{long}(\text{m.grilla}[i][j]) = \text{long}(\text{m.grilla})) \wedge_L$   
 $((\forall i, j, k: \text{nat})(i < \text{long}(\text{m.grilla}) \wedge j < \text{long}(\text{m.grilla}) \wedge k < \text{long}(\text{m.grilla})) \Rightarrow_L$   
 $\text{long}(\text{m.grilla}[i][j][k]) = \text{long}(\text{m.grilla}))$
- (2)  $\text{m.tam} = \text{long}(\text{e.grilla})$
- (3)  $((\forall x, y, z, w: \text{nat})(\text{enRango}(\text{m}, x, y) \wedge \text{enRango}(\text{m}, z, w)) \Rightarrow_L (\text{grilla}[x][y][z][w] = \text{grilla}[z][w][x][y]))$
- (4)  $((\forall x, y: \text{nat})(\text{enRango}(\text{m}, x, y)) \Rightarrow_L$   
 $(\neg \text{grilla}[x][y][x][y] \Rightarrow (\forall z, w: \text{nat})(\text{enRango}(\text{m}, z, w)) \Rightarrow_L \neg \text{grilla}[x][y][z][w]))))$
- (5)  $((\forall x, y, z, w: \text{nat})$   
 $((\text{enRango}(\text{m}, x, y) \wedge_L \text{m.grilla}[x][y][x][y]) \wedge$   
 $(\text{enRango}(\text{m}, z, w) \wedge_L \text{m.grilla}[z][w][z][w]) \wedge_L$   
 $\text{esContigua}(x, y, z, w) \Rightarrow_L \text{m.grilla}[x][y][z][w]))$
- (6)  $((\forall a, b, c, d, e, f: \text{nat})$   
 $(\text{enRango}(\text{m}, a, b) \wedge_L \text{m.grilla}[a][b][a][b]) \wedge$   
 $(\text{enRango}(\text{m}, c, d) \wedge_L \text{m.grilla}[c][d][c][d]) \wedge$   
 $(\text{enRango}(\text{m}, e, f) \wedge_L \text{m.grilla}[e][f][e][f]) \wedge$   
 $(a \neq c \vee b \neq d) \wedge (c \neq e \vee d \neq f))$   
 $\Rightarrow_L ($   
 $((\text{m.grilla}[a][b][c][d] \wedge \text{m.grilla}[c][d][e][f]) \Rightarrow \text{m.grilla}[a][b][e][f]) \wedge$   
 $((\text{m.grilla}[a][b][c][d] \wedge \neg \text{m.grilla}[c][d][e][f]) \Rightarrow \neg \text{m.grilla}[a][b][e][f])$   
 $)$

$\text{enRango} : \text{estr } m \times \text{nat } x \times \text{nat } y \longrightarrow \text{bool}$

$\text{enRango}(m, x, y) \equiv (x < \text{m.tam} \wedge y < \text{m.tam})$

$\text{esContigua} : \text{nat } x \times \text{nat } y \times \text{nat } z \times \text{nat } w \longrightarrow \text{bool}$

$\text{esContigua}(x, y, z, w) \equiv (x = z \wedge y = w + 1) \vee$   
 $(x = z + 1 \wedge y = w) \vee$   
 $(\text{if } z > 0 \text{ then } (x = z - 1 \wedge y = w) \text{ else false fi}) \vee$   
 $(\text{if } w > 0 \text{ then } (x = z \wedge y = w - 1) \text{ else false fi})$

$\text{Rep} : \text{estr } m \longrightarrow \text{bool}$

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

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

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

## Algoritmos

### 3.3. Algoritmos

---

---

**iCrearMapa()**  $\rightarrow res : \text{map}$

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

Complejidad:  $O(1)$

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

---



---

---

**iCoordenadas(in m : map)**  $\rightarrow res : \text{conj}(\text{coord})$

1:  $\text{conj}(\text{coord}) \text{ coors} \leftarrow \text{Vacio}()$   $\triangleright$  Crear conjunto vacio es  $O(1)$  //  $O(1)$   
 2: **for**  $i = 0$  to  $m.\text{tam}$  **do**  
 3:   **for**  $j = 0$  to  $m.\text{tam}$  **do**  
 4:     **if**  $\text{iPosExistente}(i, j)$  **then**  $\triangleright O(1)$   
 5:        $c \leftarrow \text{CrearCoor}(i, j)$   $\triangleright O(1)$   
 6:        $\text{AgregarRapido}(\text{coors}, c)$   $\triangleright O(1)$   
 7:     **end if**  
 8:   **end for**  
 9: **end for**  
 10:  $res \leftarrow \text{coors}$   $\triangleright O(1)$

Complejidad:  $O((\text{tam}(m))^2)$

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

---

---

**iPosExistente**(in  $m$  : map, in  $c$  : coor)  $\rightarrow res$  : bool

```

1: bool existe
2: if latitud(c)  $\geq m.tam \vee$  longitud(c)  $\geq m.tam$  then  $\triangleright O(1)$ 
3:   existe  $\leftarrow$  false  $\triangleright O(1)$ 
4: else
5:   nat x  $\leftarrow$  Latitud(c)  $\triangleright O(1)$ 
6:   nat y  $\leftarrow$  Longitud(c)  $\triangleright O(1)$ 
7:   existe  $\leftarrow m.grilla[x][y][x][y]$   $\triangleright O(1)$ 
8: end if
9: res  $\leftarrow$  existe  $\triangleright O(1)$ 

```

Complejidad:  $O(1)$ 

Justificación: 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)  $\rightarrow res$  : bool

```

1: res  $\leftarrow m.grilla[Latitud(c_1)][Longitud(c_1)][Latitud(c_2)][Longitud(c_2)]$   $\triangleright O(1)$ 

```

Complejidad:  $O(1)$ 

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

---

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

```

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

```



```

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:  $O(\max(\text{Latitud}(c), \text{Longitud}(c), m.\text{tam}))^4$

Justificación: Trabajo con matrices de 4 dimensiones, en peor caso hay que redimensionar, en donde se crea una nueva grilla con el nuevo tamaño. CrearGrilla tiene complejidad  $O(n^4)$ , por lo que en este caso es  $O(\max(\text{Latitud}(c), \text{Longitud}(c))^4)$ . Si se redimensiona, es porque:  $\max(\text{Latitud}(c), \text{Longitud}(c)) > m.\text{tam}$ . Entonces en ese caso,  $\max(\text{Latitud}(c), \text{Longitud}(c)) = O(\max(\text{Latitud}(c), \text{Longitud}(c), m.\text{tam}))^4$ .

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  $O(\max(\text{Latitud}(c), \text{Longitud}(c), m.\text{tam}))^2$ , donde *m.tam* es el tamaño original de la grilla. Entonces la complejidad en peor caso es la de mayor exponente por álgebra de ordenes:  $O(\max(\text{Latitud}(c), \text{Longitud}(c), m.\text{tam}))^4$

---

**iTam**(in  $m : \text{map}$ )  $\rightarrow res : \text{nat}$ 
1:  $res \leftarrow m.tam$  $\triangleright O(1)$ Complejidad:  $O(1)$ 


---

**iCrearGrilla**(in  $n : \text{nat}$ )  $\rightarrow res : \text{vector}(\text{vector}(\text{vector}(\text{vector}(\text{bool}))))$ 

Funcion privada

Pre:  $n > 0$ Post:  $res$  es una grilla de tam  $n$ 1:  $\text{vector}(\text{vector}(\text{vector}(\text{vector}(\text{bool})))) \text{ nGrilla} \leftarrow \text{Vacio}()$  $\triangleright O(1)$ 2: **for** nat  $i \leftarrow 0$  to  $n - 1$  **do** $\triangleright O(n^4)$ 3:    $\text{vector}(\text{vector}(\text{vector}(\text{bool}))) \text{ nGrilla2} \leftarrow \text{Vacio}()$  $\triangleright O(1)$ 4:   **for** nat  $j \leftarrow 0$  to  $n - 1$  **do** $\triangleright O(n^3)$ 5:      $\text{vector}(\text{vector}(\text{bool})) \text{ nGrilla3} \leftarrow \text{Vacio}()$  $\triangleright O(1)$ 6:     **for** nat  $k \leftarrow 0$  to  $n - 1$  **do** $\triangleright O(n^2)$ 7:        $\text{vector}(\text{bool}) \text{ nGrilla4} \leftarrow \text{Vacio}()$  $\triangleright O(1)$ 8:       **for** nat  $l \leftarrow 0$  to  $n - 1$  **do** $\triangleright O(n)$ 9:          $\text{AgregarAtras}(\text{nGrilla4}, \text{false})$  $\triangleright O(1)$ 10:       **end for**11:        $\text{AgregarAtras}(\text{nGrilla3}, \text{nGrilla4})$ 12:     **end for**13:      $\text{AgregarAtras}(\text{nGrilla2}, \text{nGrilla3})$ 14:   **end for**15:    $\text{AgregarAtras}(\text{nGrilla}, \text{nGrilla2})$ 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  $\text{ nGrilla: vector}(\text{vector}(\text{vector}(\text{vector}(\text{bool}))))$ , in  $\text{ vGrilla: vector}(\text{vector}(\text{vector}(\text{vector}(\text{bool}))))$ )
CopiarCoordenadas : Función privadaDescripción : Asigno las coordenadas existentes de la vieja grilla en la nueva grillaPre:  $\text{longitud}(\text{vGrilla}) \leq \text{longitud}(\text{nGrilla})$ Post:  $(\forall i, j : \text{nat})(i < \text{longitud}(\text{vGrilla}) \wedge j < \text{longitud}(\text{vGrilla}) \Rightarrow_{\text{L}}$  $\text{nGrilla}[i][j][i][j] =_{\text{obs}} \text{vGrilla}[i][j][i][j])$ Complejidad:  $O(\text{Longitud}(\text{vGrilla})^2)$ 1: **for** nat  $i \leftarrow 0$  to  $\text{Longitud}(\text{vGrilla}) - 1$  **do** $\triangleright O(\text{Longitud}(\text{vGrilla})^2)$ 2:   **for** nat  $j \leftarrow 0$  to  $\text{Longitud}(\text{vGrilla}) - 1$  **do** $\triangleright O(\text{Longitud}(\text{vGrilla}))$ 3:      $\text{nGrilla}[i][j][i][j] \leftarrow \text{vGrilla}[i][j][i][j]$  $\triangleright O(1)$ 4:   **end for**5: **end for**Complejidad: $\triangleright O(\text{Longitud}(\text{vGrilla})^2)$ Justificacion: Son 2 fors anidados, donde en cada uno hago  $\text{Longitud}(\text{vGrilla})$  iteraciones.  $O(\text{Longitud}(\text{vGrilla}))$  $* O(\text{Longitud}(\text{vGrilla})) = O(\text{Longitud}(\text{vGrilla})^2)$ 


---

### 3.4. Servicios usados

De Vector

-  $\text{AgregarAtras}(\text{vector}(\alpha), \alpha)$  debe ser  $O(\text{f}(\text{long}(\text{v})) + \text{copy}(\text{a}))$

- Vacía() debe ser  $O(1)$

De Conjunto Lineal

- AgregarRapido( conj( $\alpha$ ),  $\alpha$ ) debe ser  $O(\text{copy}(a))$

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  $O(\text{Longitud}(v\text{Grilla})^2)$

De Cola

- Encolar(colaEntr, entrenador) debe ser  $O(\log(EC))$
- Proximo(colaEntr) debe ser  $O(1)$
- Desencolar(colaEntr) debe ser  $O(1)$

## 4. Coordenada

### Interfaz

#### 4.1. Interfaz

se explica con: COORDENADA.

géneros: `coor`.

#### Operaciones básicas de Coordenada

**CREARCOOR**(`in`  $n_1 : \text{nat}$ , `in`  $n_2 : \text{nat}$ )  $\rightarrow res : \text{coor}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{crearCoor}\}$

**Complejidad:**  $O(1)$

**Descripción:** genera una coordenada nueva.

**LATITUD**(`in`  $c : \text{coor}$ )  $\rightarrow res : \text{nat}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{latitud}(c)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve la latitud de la coordenada  $c$ .

**LONGITUD**(`in`  $c : \text{coor}$ )  $\rightarrow res : \text{nat}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{longitud}(c)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve la longitud de la coordenada  $c$ .

**DISTEUCLIDEA**(`in`  $c_1 : \text{coor}$ , `in`  $c_2 : \text{coor}$ )  $\rightarrow res : \text{nat}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{distEuclidea}(c_1, c_2)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve la distancia entre la coordenadas  $c_1$  y  $c_2$ .

**COORDENADAARRIBA**(`in`  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{distEuclidea}(c_1, c_2)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve la coordenadas .

**COORDENADAABAJO**(`in`  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$

**Pre**  $\equiv \{\text{Latitud}(c) > 0\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{distEuclidea}(c_1, c_2)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve la coordenadas .

**COORDENADAALADERECHA**(`in`  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{distEuclidea}(c_1, c_2)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve la coordenadas .

**COORDENADAALAIZQUIERDA**(`in`  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$

**Pre**  $\equiv \{\text{Longitud}(c) > 0\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{distEuclidea}(c_1, c_2)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve la coordenadas .

## Representación

### 4.2. Representacion

**coor se representa con estr**

donde **estr** es  $\text{tupla}(\text{latitud: nat}, \text{longitud: nat})$

$\text{Rep} : \text{estr } e \longrightarrow \text{bool}$

$\text{Rep}(e) \equiv \text{true}$

$\text{Abs} : \text{estr } e \longrightarrow \text{coor}$

$\{\text{Rep}(e)\}$

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

## Algoritmos

### 4.3. Algoritmos

---

---

**iCrearCoor**(in  $n_1 : \text{nat}$ , in  $n_2 : \text{nat}$ )  $\rightarrow res : \text{coor}$

1:  $res \leftarrow \langle n_1, n_2 \rangle$

$\triangleright O(1)$

Complejidad:  $O(1)$

---



---

---

**iLatitud**(in  $c : \text{coor}$ )  $\rightarrow res : \text{nat}$

1:  $res \leftarrow c.\text{latitud}$

$\triangleright O(1)$

Complejidad:  $O(1)$

---



---

---

**iLongitud**(in  $c : \text{coor}$ )  $\rightarrow res : \text{nat}$

1:  $res \leftarrow c.\text{longitud}$

$\triangleright O(1)$

Complejidad:  $O(1)$

---

---



---

**iDistEuclidea**(in  $c_1 : \text{coor}$ , in  $c_2 : \text{coor}$ )  $\rightarrow res : \text{nat}$ 

```

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

```

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


---



---

**iCoordenadaArriba**(in  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$ 

```

1:  $res \leftarrow \langle \text{Latitud}(c) + 1, \text{Longitud}(c) \rangle$   $\triangleright O(1)$ 

```

Complejidad:  $O(1)$ 


---



---

**iCoordenadaAbajo**(in  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$ 

```

1:  $res \leftarrow \langle \text{Latitud}(c) - 1, \text{Longitud}(c) \rangle$   $\triangleright O(1)$ 

```

Complejidad:  $O(1)$ 


---



---

**iCoordenadaALaDerecha**(in  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$ 

```

1:  $res \leftarrow \langle \text{Latitud}(c), \text{Longitud}(c) + 1 \rangle$   $\triangleright O(1)$ 

```

Complejidad:  $O(1)$ 


---



---

**iCoordenadaALaIzquierda**(in  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$ 

```

1:  $res \leftarrow \langle \text{Latitud}(c), \text{Longitud}(c) - 1 \rangle$   $\triangleright O(1)$ 

```

Complejidad:  $O(1)$

## 5. DiccString( $\alpha$ )

### Interfaz

#### 5.1. Interfaz

**parámetros formales**

**géneros**  $\alpha$

**se explica con:** DICC(String,  $\alpha$ ), CONJ(String).

**géneros:** diccString( $\alpha$ ).

VACIO()  $\rightarrow res : \text{diccString}(\alpha)$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{vacío}\}$

**Complejidad:**  $O(1)$

**Descripción:** genera un diccionario vacío.

DEFINIR(**in/out**  $d : \text{diccString}(\alpha)$ , **in**  $s : \text{string}$ , **in**  $a : \alpha$ )

**Pre**  $\equiv \{d =_{\text{obs}} d_0\}$

**Post**  $\equiv \{d =_{\text{obs}} \text{definir}(p, a, d)\}$

**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**  $p : \text{string}$ , **in**  $d : \text{diccString}(\alpha)$ )  $\rightarrow res : \text{bool}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{def?}(p, d)\}$

**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**  $p : \text{string}$ , **in**  $d : \text{diccString}(\alpha)$ )  $\rightarrow res : \alpha$

**Pre**  $\equiv \{\text{def?}(s, d)\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{obtener}(p, d))\}$

**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**  $p : \text{string}$ , **in/out**  $d : \text{diccString}(\alpha)$ )

**Pre**  $\equiv \{d =_{\text{obs}} d_0 \wedge \text{def?}(s, d)\}$

**Post**  $\equiv \{d =_{\text{obs}} \text{borrar}(p, d_0)\}$

**Complejidad:**  $O(|P|)$  siendo P la clave mas larga.

**Descripción:** borra la clave  $p$  y su significado.

CLAVES(**in**  $d : \text{diccString}(\alpha)$ )  $\rightarrow res : \text{conj}(\text{string})$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{claves}(d))\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve del conjunto de claves de  $d$ . Aliasing:  $res$  se modifica sii  $\text{claves}(d)$  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 así la complejidad en peor caso.

`diccString( $\alpha$ )` se representa con `estr`

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

donde `nodo` es `tupla(definicion: puntero( $\alpha$ ), 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

$\text{Abs} : \text{estr } d \longrightarrow \text{dicc}(\text{string}, \alpha) \quad \{\text{Rep}(d)\}$

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

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

## Algoritmos

### 5.3. Algoritmos

---

**iVacio()**  $\rightarrow res : \text{diccString}(\alpha)$

- |    |   |                       |
|----|---|-----------------------|
| 1: | puntero(nodo) <i>iRaiz</i> $\leftarrow NULL$            | $\triangleright O(1)$ |
| 2: | conj(string) <i>iClaves</i> $\leftarrow \text{Vacio}()$ | $\triangleright O(1)$ |
| 3: | $res \leftarrow \langle iRaiz, iClaves \rangle$         | $\triangleright O(1)$ |
| 4: | Complejidad: $O(1)$                                     |                       |
| 5: | Justificación: $O(1) + O(1) + O(1)$                     |                       |
- 

---

**iClaves(in  $d : \text{diccString}(\alpha)$ )**  $\rightarrow res : \text{conj}(\alpha)$

- |    |                                  |                       |
|----|----------------------------------|-----------------------|
| 1: | res $\leftarrow d.\text{claves}$ | $\triangleright O(1)$ |
| 2: | Complejidad: $O(1)$              |                       |
-



---

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

```

1: puntero(nodo)  $n \leftarrow raiz$   $\triangleright O(1)$ 
2: nat  $i \leftarrow 0$   $\triangleright O(1)$ 
3: while  $i < Longitud(p)$  do  $\triangleright$  Se repite  $|p|$   $O(1)$ 
4:    $actual \leftarrow (*actual).siguientes[ord(p[i])]$   $\triangleright O(1)$ 
5:    $i \leftarrow i + 1$   $\triangleright O(1)$ 
6: end while
7:  $res \leftarrow (*actual).definicion$   $\triangleright O(1)$ 
8: Complejidad:  $O(|P|)$ 
9: Justificación: 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( $\alpha$ ), in  $p$ : string, in  $a$ :  $\alpha$ )

```

1: Puntero(nodo)  $actual \leftarrow raiz$   $\triangleright O(1)$ 
2: Nat  $i \leftarrow 0$   $\triangleright O(1)$ 
3: bool  $esNueva \leftarrow true$   $\triangleright O(1)$ 
4: while  $i < Longitud(p)$  do  $\triangleright$  Se repite  $|p|$   $O(1)$ 
5:   if  $(*actual).siguientes[ord(p[i])] = NULL$  then  $\triangleright O(1)$ 
6:      $(*actual).siguientes[ord(p[i])] \leftarrow \& \langle NULL, arreglo[256](NULL), NULL \rangle$   $\triangleright O(1)$ 
7:      $esNueva \leftarrow false$   $\triangleright O(1)$ 
8:   end if
9:    $actual \leftarrow (*actual).siguientes[ord(p[i])]$   $\triangleright O(1)$ 
10:   $i \leftarrow i + 1$ 
11: end while
12: if  $(*actual).definicion \neq NULL$  then  $\triangleright O(1)$ 
13:    $(*actual).definicion \leftarrow NULL$   $\triangleright$  se libera la memoria acupada por definicion  $O(1)$ 
14: end if
15:  $(*actual).definicion \leftarrow \& a$   $\triangleright O(1)$ 
16: if  $esNueva$  then  $\triangleright O(1)$ 
17:    $itConj(string) it \leftarrow claves.AgregarRapido(s)$   $\triangleright O(1)$ 
18:    $(*actual).itClave \leftarrow \& it$   $\triangleright O(1)$ 
19: end if
20: Complejidad:  $O(|P|)$ 
21: Justificación: 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  $i \leftarrow 0$   $\triangleright O(1)$ 
2: bool  $pertenece \leftarrow true$   $\triangleright O(1)$ 
3: puntero(nodo)  $actual \leftarrow raiz$   $\triangleright O(1)$ 
4: while  $i < Longitud(p) \wedge pertenece$  do  $\triangleright$  Se repite  $|p|$   $O(1)$ 
5:   if  $(*actual).siguientes[ord(p[i])] = NULL$  then  $\triangleright O(1)$ 
6:      $pertenece \leftarrow false$   $\triangleright O(1)$ 
7:   end if
8:    $actual \leftarrow (*actual).siguientes[ord(p[i])]$   $\triangleright O(1)$ 
9:    $i \leftarrow i + 1$   $\triangleright O(1)$ 
10: end while
11: if  $(*actual).significado = NULL$  then  $\triangleright O(1)$ 
12:    $pertenece \leftarrow false$   $\triangleright O(1)$ 
13: end if
14:  $res \leftarrow pertenece$   $\triangleright O(1)$ 
15: Complejidad:  $O(|P|)$ 
16: Justificación: 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( $\alpha$ ))

1: bool $borrarRaiz \leftarrow d.Claves() = 1$	$\triangleright O(1)$
2: puntero(nodo) $reserva \leftarrow raiz$	$\triangleright O(1)$
3: nat $rindex \leftarrow 0$	$\triangleright O(1)$
4: puntero(nodo) $actual \leftarrow raiz$	$\triangleright O(1)$
5: nat $i \leftarrow 0$	$\triangleright O(1)$
6: <b>while</b> $i < Longitud(p)$ <b>do</b>	$\triangleright$ Se repite $ p $ $O(1)$
7: $actual \leftarrow (*actual).siguientes[ord(p[i])]$	$\triangleright O(1)$
8:     bool $definido \leftarrow i \neq  p  - 1 \wedge (*actual).definicion \neq NULL$	$\triangleright O(1)$
9: <b>if</b> CuentaHijos( $actual$ ) $> 1 \vee definido$ <b>then</b>	$\triangleright O(1)$
10: $reserva \leftarrow actual$	$\triangleright O(1)$
11: $rindex \leftarrow i + 1$	$\triangleright O(1)$
12: <b>end if</b>	
13: $i \leftarrow i + 1$	$\triangleright O(1)$
14: <b>end while</b>	
15: EliminarSiguiente( $(*actual).(*itClave)$ )	$\triangleright O(1)$
16: $(*actual).itClave \leftarrow NULL$	$\triangleright$ Se libera la memoria ocupada por $(*actual).itClave$ $O(1)$
17: <b>if</b> CuentaHijos( $actual$ ) $> 1$ <b>then</b>	$\triangleright O(1)$
18: $(*actual).definicion \leftarrow NULL$	$\triangleright$ se libera la memoria ocupada por la definicion $O(1)$
19: <b>end if</b>	
20: <b>if</b> CuentaHijos( $actual$ ) $= 0$ <b>then</b>	$\triangleright O(1)$
21:     BorrarDesde( $reserva, rindex$ )	$\triangleright O( P )$
22: <b>end if</b>	
23: <b>if</b> borrarRaiz <b>then</b>	$\triangleright O(1)$
24: $d.raiz \leftarrow NULL$	$\triangleright$ se libera la memoria ocupada por raiz $O(1)$
25: <b>end if</b>	
26: <u>Complejidad:</u> $O( P )$	
27: <u>Justificación:</u> 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**  $\equiv \{desde \text{ y } desde.siguientes[a] \text{ no nulos}\}$

**Post**  $\equiv \{\text{Borra la rama a partir de } desde.siguientes[a]\}$

---

**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: Justificación: 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**  $\equiv \{desde \text{ no nulo}\}$

**Post**  $\equiv \{\text{deveuelve la cantidad de punteros no nulos en } desde.siguientes\}$

---

**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: Justificación: 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( $\alpha$ ),  $\alpha$  ) debe ser O(copy(a))
- EliminarSiguiente(itConj( $\alpha$ )) debe ser O(1)

De String

- Longitud(string) debe ser  $O(1)$

De Char

- ord(char) debe ser  $O(1)$

## 6. iterDiccString( $\alpha$ )

### Interfaz

#### 6.1. Interfaz

**parámetros formales**

**géneros**  $\alpha$

**se explica con:** ITERADORUNIDIRECCIONAL( $\alpha$ ), DICCSTRING( $\alpha$ ).

**géneros:** iterDiccString( $\alpha$ ).

**CREARIT**(in  $d$ : diccString( $\alpha$ ))  $\rightarrow res$  : iterDiccString( $\alpha$ )

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{alias}(\text{esPermutación}(\text{SecuSuby}(res), \text{claves}(d)))\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve un iterador al diccionario.

**Aliasing:** El iterador se invalida si se modifican claves del diccionario.

**HAYMAS?**(in  $it$ : iterDiccString( $\alpha$ ))  $\rightarrow res$  : bool

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res = \text{hayMas?}(it)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve true si hay mas claves por recorrer.

**ACTUAL**(in  $it$ : iterDiccString( $\alpha$ ))  $\rightarrow res$  : tupla<string,  $\alpha$ >

**Pre**  $\equiv \{\text{hayMas?}(it)\}$

**Post**  $\equiv \{res = \text{actual}(it)\}$

**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( $\alpha$ ))

**Pre**  $\equiv \{it = it_0 \wedge \text{hayMas?}(it)\}$

**Post**  $\equiv \{it = \text{avanzar}(it_0)\}$

**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( $\alpha$ ) se representa con estr

donde estr es tupla( $itClave$ : itConj(string),  $dicc$ : diccString( $\alpha$ ))

### Invariante de representacion

Rep : estr  $e \rightarrow$  bool

$$\text{Rep}(e) \equiv \text{Rep}(e.\text{itClave}) \wedge_L ((\text{siguiente}(e.\text{itClave}) = \text{NULL} \vee_L \text{siguiente}(e.\text{itClave}) \in \text{claves}(e.\text{dicc}))$$

## Funcion de abstraccion

$$\text{Abs} : \text{estr } e \longrightarrow \text{itUni}(\alpha)$$

$$\{\text{Rep}(d)\}$$

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

## Algoritmos

### 6.3. Algoritmos

---



---

**iCrearIt**(in  $d : \text{diccString}(\alpha) \rightarrow res : \text{iterDiccString}(\alpha)$ 

 1:  $res \leftarrow \langle d, \text{CrearIt}(\text{Claves}(d)) \rangle$ 
 $\triangleright$  Por referencia  $O(1)$ 
Complejidad:  $O(1)$ 
Justificación: 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  $iter : \text{iterDiccString}(\alpha) \rightarrow res : \text{bool}$ 

 1:  $res \leftarrow \text{HaySiguiente}(iter.\text{itClave})$ 
 $\triangleright O(1)$ 
Complejidad:  $O(1)$ 


---



---



---

**iActual**(in  $iter : \text{iterDiccString}(\alpha) \rightarrow res : \text{tupla}\langle \text{string}, \alpha \rangle$ 

 1:  $res \leftarrow \langle \text{Siguiente}(iter.\text{iClave}), \text{Obtener}(\text{Siguiente}(iter.\text{iClave}), iter.\text{dicc}) \rangle$ 
 $\triangleright O(|P|)$ 
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 larga. Acceder al Siguiente del conjunto es  $O(1)$ . En total, el algoritmo cuesta  $O(|P|)$ 


---



---



---

**iAvanzar**(in/out  $iter : \text{iterDiccString}(\alpha)$ 

 1: Avanzar( $iter.\text{itClave}$ )

 $\triangleright O(1)$ 
Complejidad:  $O(1)$ 


---

### 6.4. Servicios usados

 De  $\text{conj}(\alpha)$ 

 -  $\text{CrearIt}(\text{conj}(\alpha))$  debe ser  $O(1)$ 

 De  $\text{itConj}(\alpha)$ 

 -  $\text{HaySiguiente}(\text{itConj}(\alpha))$  debe ser  $O(1)$ 

 -  $\text{Siguiente}(\text{itConj}(\alpha))$  debe ser  $O(1)$

- Avanzar(itConj( $\alpha$ )) debe ser  $O(1)$

De diccString( $\alpha$ )

- Claves(diccString( $\alpha$ )) debe ser  $O(1)$
- Obtener(string, diccString( $\alpha$ )) debe ser  $O(|P|)$

## 7. Cola de Entrenadores

### Interfaz

#### 7.1. Interfaz de Cola de Entrenadores

se explica con: COLA DE PRIORIDAD( $\alpha$ ), ITERADOR COLA DE PRIORIDADES( $\alpha$ ).

géneros: colaEntr, itcolaEntrenador).

#### Operaciones básicas de Cola de Entrenadores

VACIA()  $\rightarrow res : \text{colaEntr}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{vacía}\}$

**Complejidad:**  $O(1)$

**Descripción:** genera una nueva cola de entrenadores

ENCOLAR(in/out  $h : \text{colaEntr}$ , in  $j : \text{entrenador}$ )  $\rightarrow res : \text{itcolaEntrenador}$

**Pre**  $\equiv \{h =_{\text{obs}} h_0\}$

**Post**  $\equiv \{h =_{\text{obs}} \text{encolar}(h_0, j)\}$

**Complejidad:**  $O(\log(EC))$

**Descripción:** encola  $j$  a  $h$

**Aliasing:** el elemento  $j$  se encola por copia

ESVACIA?(in  $h : \text{colaEntr}$ )  $\rightarrow res : \text{bool}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{vacía?}(h)\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve true si la cola es vacía

PROXIMO(in  $h : \text{colaEntr}$ )  $\rightarrow res : \text{itcolaEntrenador}$

**Pre**  $\equiv \{\neg \text{vacía?}(h)\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{proximo}(h))\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve el próximo de la cola

**Aliasing:** el elemento se devuelve por copia

DESENCOLAR(in/out  $h : \text{colaEntr}$ )

**Pre**  $\equiv \{h =_{\text{obs}} h_0 \wedge \neg \text{vacía}(h)\}$

**Post**  $\equiv \{h =_{\text{obs}} \text{desencolar}(h_0)\}$

**Complejidad:**  $O(\log(EC))$

**Descripción:** desencola el próximo de  $h$

#### Operaciones del iterador

#### 7.2. Interfaz del iterador

BORRAR(in  $it : \text{itcolaEntrenador}$ , in  $h : \text{colaEntr}$ )

**Pre**  $\equiv \{\text{siguiente}(it) \in \text{elementos}(h)\}$

**Post**  $\equiv \{\text{siguiente}(it) \notin \text{elementos}(h)\}$

**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  $\text{tupla}(\text{raiz: puntero(Nodoheap)}, \text{ultimo: puntero(Nodoheap)})$

donde **Nodoheap** es  $\text{tupla}(\text{elemento: entrenador}, \text{padre: puntero(Nodoheap)}, \text{izq: puntero(Nodoheap)}, \text{der: puntero(Nodoheap)})$

donde **entrenador** es  $\text{tupla}(\text{id: nat}, \text{cantCapt: nat})$

#### 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.

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

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

## Representación

### 7.4. Representacion del iterador

**itcolaEntrenador se representa con estr**

donde **estr** es  $\text{tupla}(\text{siguiente: puntero(Nodoheap)}, \text{estructura: puntero(colaEntr)})$

$\text{Rep} : \text{estr } e \longrightarrow \text{bool}$

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

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

$\text{Nodoheap}(j, k, i, d, p) \equiv \langle \langle i, j \rangle, p, i, d \rangle$

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

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

## Algoritmos

## 7.5. Algoritmos Cola de Entrenadores

---



---

**iVacio()**  $\rightarrow res : colaEntr$

1:  $res \leftarrow \langle NULL, NULL \rangle$   $\triangleright O(1)$

Complejidad:  $O(1)$

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

---



---



---

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

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

Complejidad:  $O(1)$

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

---



---



---

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

1:  $Nodoheap\ n \leftarrow \langle j, NULL, NULL, NULL \rangle$   $\triangleright O(1)$

2:  $puntero(Nodoheap)\ p \leftarrow \&n$   $\triangleright O(1)$

3: **if**  $EsVacia?(h)$  **then**  $\triangleright$  Insertamos el nodo al final de la cola  $O(1)$

4:      $h.raiz \leftarrow p$   $\triangleright O(1)$

5:      $h.ultimo \leftarrow p$   $\triangleright O(1)$

6: **else**

7:     **if**  $h.raiz = h.ultimo$  **then**  $\triangleright O(1)$

8:          $(p \rightarrow padre) \leftarrow h.raiz$   $\triangleright O(1)$

9:          $(h.raiz \rightarrow izq) \leftarrow p$   $\triangleright O(1)$

10:          $h.ultimo \leftarrow p$   $\triangleright O(1)$

11:     **else**

12:         **if**  $(h.raiz \rightarrow izq) = h.ultimo$  **then**  $\triangleright O(1)$

13:              $(p \rightarrow padre) \leftarrow h.raiz$   $\triangleright O(1)$

14:              $(h.raiz \rightarrow der) \leftarrow p$   $\triangleright O(1)$

15:              $h.ultimo \leftarrow p$   $\triangleright O(1)$

16:         **else**

17:             **if**  $EsHijoIzquierdo?(h.ultimo)$  **then**  $\triangleright O(1)$

18:                  $(p \rightarrow padre) \leftarrow (h.ultimo \rightarrow padre)$   $\triangleright O(1)$

19:                  $(h.ultimo \rightarrow padre \rightarrow der) \leftarrow p$   $\triangleright O(1)$

20:                  $h.ultimo \leftarrow p$   $\triangleright O(1)$

21:             **else**

22:                  $puntero(Nodoheap)\ i \leftarrow h.ultimo$   $\triangleright O(1)$

23:                 **while**  $i \neq NULL \wedge \neg EsHijoIzquierdo?(j)$  **do**  $\triangleright O(\log(EC))$

24:                      $i \leftarrow (i \rightarrow padre)$   $\triangleright O(1)$

25:                 **end while**

26:                 **if**  $i = NULL$  **then**  $\triangleright O(1)$

27:                      $puntero(Nodoheap)\ ultimoizq \leftarrow h.raiz$   $\triangleright O(1)$

28:                     **while**  $(ultimoizq \rightarrow izq) \neq NULL$  **do**  $\triangleright O(\log(EC))$

29:                          $ultimoizq \leftarrow (ultimoizq \rightarrow izq)$   $\triangleright O(1)$

30:                     **end while**

31:                      $(p \rightarrow padre) \leftarrow ultimoizqu$   $\triangleright O(1)$

32:                      $(ultimoizq \rightarrow izq) \leftarrow p$   $\triangleright O(1)$

33:                      $h.ultimo \leftarrow p$   $\triangleright O(1)$

34:                 **else**

35:                      $i \leftarrow (i \rightarrow padre \rightarrow der)$   $\triangleright O(1)$

36:                     **while**  $(i \rightarrow izq) \neq NULL$  **do**  $\triangleright O(\log(EC))$

37:                          $i \leftarrow (i \rightarrow izq)$   $\triangleright O(1)$

38:                     **end while**

39:                      $(p \rightarrow padre) \leftarrow i$   $\triangleright O(1)$

40:                      $(j \rightarrow izq) \leftarrow p$   $\triangleright O(1)$

```

41:          $h.ultimo \leftarrow p$   $\triangleright O(1)$ 
42:     end if
43: end if
44: end if
45: end if
46: end if
47:  $puntero(colaEntr)q \leftarrow h$ 
48:  $it \leftarrow CrearItCola(h.ultimo, q)$   $\triangleright O(1)$ 
49: if  $h.raiz \neq h.ultimo$  then  $\triangleright O(1)$ 
50:      $puntero(Nodo) \text{ nuevoNodo} \leftarrow h.ultimo$   $\triangleright O(1)$ 
51:     while  $((nuevoNodo \rightarrow elemento.cantCapt) < (nuevoNodo \rightarrow padre \rightarrow elemento.cantCapt)) \vee (((nuevoNodo \rightarrow$   

 $elemento.cantCapt) = (nuevoNodo \rightarrow padre \rightarrow elemento.cantCapt)) \wedge ((nuevoNodo \rightarrow elemento.id) < (nuevoNodo \rightarrow$   

 $padre \rightarrow elemento.id)))$  do  $\triangleright O(\log(EC))$ 
52:          $puntero(Nodoheap) aSwapear \leftarrow (nuevoNodo \rightarrow padre)$ 
53:         if  $h.ultimo = nuevoNodo$  then
54:              $h.ultimo \leftarrow aSwapear$ 
55:         end if
56:         if  $EsHijoIzquierdo?(nuevoNodo)$  then  $\triangleright O(1)$ 
57:             if  $(nuevoNodo \rightarrow izq) = NULL$  then  $\triangleright O(1)$ 
58:                  $(aSwapear \rightarrow izq) \leftarrow NULL$   $\triangleright O(1)$ 
59:             else
60:                  $(aSwapear \rightarrow izq) \leftarrow (nuevoNodo \rightarrow izq)$   $\triangleright O(1)$ 
61:                  $(nuevoNodo \rightarrow izq \rightarrow padre) \leftarrow aSwapear$   $\triangleright O(1)$ 
62:             end if
63:             if  $(nuevoNodo \rightarrow der) = NULL$  then  $\triangleright O(1)$ 
64:                  $(nuevoNodo \rightarrow der) \leftarrow (aSwapear \rightarrow der)$   $\triangleright O(1)$ 
65:                 if  $(aSwapear \rightarrow der) \neq NULL$  then  $\triangleright O(1)$ 
66:                      $(aSwapear \rightarrow der \rightarrow padre) \leftarrow nuevoNodo$   $\triangleright O(1)$ 
67:                 end if
68:                  $(aSwapear \rightarrow der) \leftarrow NULL$   $\triangleright O(1)$ 
69:             else
70:                 if  $(aSwapear \rightarrow der) = NULL$  then  $\triangleright O(1)$ 
71:                      $(aSwapear \rightarrow der) \leftarrow (nodoNuevo \rightarrow der)$   $\triangleright O(1)$ 
72:                      $(nodoNuevo \rightarrow der \rightarrow padre) \leftarrow aSwapear$   $\triangleright O(1)$ 
73:                      $(nodoNuevo \rightarrow der) \leftarrow NULL$   $\triangleright O(1)$ 
74:                 else
75:                      $(nodoNuevo \rightarrow der \rightarrow padre) \leftarrow aSwapear$   $\triangleright O(1)$ 
76:                      $(aSwapear \rightarrow der \rightarrow padre) \leftarrow nuevoNodo$   $\triangleright O(1)$ 
77:                      $puntero(Nodoheap) auxiliar \leftarrow (nuevoNodo \rightarrow der)$   $\triangleright O(1)$ 
78:                      $(nuevoNodo \rightarrow der) \leftarrow (aSwapear \rightarrow der)$   $\triangleright O(1)$ 
79:                      $(aSwapear \rightarrow der) \leftarrow auxiliar$   $\triangleright O(1)$ 
80:                 end if
81:             end if
82:              $(nuevoNodo \rightarrow izq) \leftarrow aSwapear$   $\triangleright O(1)$ 
83:         else
84:             if  $(nuevoNodo \rightarrow der) = NULL$  then  $\triangleright O(1)$ 
85:                  $(aSwapear \rightarrow der) \leftarrow NULL$   $\triangleright O(1)$ 
86:             else
87:                  $(aSwapear \rightarrow der) \leftarrow (nuevoNodo \rightarrow der)$   $\triangleright O(1)$ 
88:                  $(nuevoNodo \rightarrow der \rightarrow padre) \leftarrow aSwapear$   $\triangleright O(1)$ 
89:             end if
90:             if  $(nuevoNodo \rightarrow izq) = NULL$  then  $\triangleright O(1)$ 
91:                  $(nuevoNodo \rightarrow izq) \leftarrow (aSwapear \rightarrow izq)$   $\triangleright O(1)$ 
92:                 if  $(aSwapear \rightarrow izq) \neq NULL$  then  $\triangleright O(1)$ 
93:                      $(aSwapear \rightarrow izq \rightarrow padre) \leftarrow nuevoNodo$   $\triangleright O(1)$ 
94:                 end if
95:                  $(aSwapear \rightarrow izq) \leftarrow NULL$   $\triangleright O(1)$ 
96:             else

```

```

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

```

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 raíz (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:  $res \leftarrow (h.raiz \rightarrow elemento)$                                 ▷  $O(1)$ 
```

Complejidad:  $O(1)$

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

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

```
1:  $puntero(colaEntr)q \leftarrow h$ 
2:  $it \leftarrow CrearItCola(h.raiz, q)$                                 ▷  $O(1)$ 
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 → raiz) = (it.estructura → ultimo) then                                ▷ O(1)
2:   it.siguiente ← NULL                                                                ▷ Libera memoria O(1)
3:   (it.estructura → raiz) ← NULL                                                        ▷ O(1)
4:   (it.estructura → ultimo) ← NULL                                                    ▷ O(1)
5: else
6:   puntero(Nodo) cambiado ← (it.estructura → ultimo)                                ▷ O(1)
7:   puntero(Nodo) aPonerUltimo ← it.siguiente
8:   if (it.estructura → ultimo → padre) ≠ aPonerUltimo then                            ▷ Swapeamos con el último de la cola. O(1)
9:     (it.estructura → ultimo → izq) ← (aPonerUltimo → izq)                            ▷ O(1)
10:    (it.estructura → ultimo → der) ← (aPonerUltimo → der)                            ▷ O(1)
11:    if (aPonerUltimo → izq) ≠ NULL then                                                ▷ O(1)
12:      (aPonerUltimo → izq → padre) ← (it.estructura → ultimo)                      ▷ O(1)
13:    end if
14:    if (aPonerUltimo → der) ≠ NULL then                                                ▷ O(1)
15:      (aPonerUltimo → der → padre) ← it.estructura → ultimo                        ▷ O(1)
16:    end if
17:    if aPonerUltimo ≠ it.estructura → raiz then                                    ▷ O(1)
18:      padreaPonerUltimo ← (aPonerUltimo → padre)                                    ▷ O(1)
19:      padreUltimo ← (it.estructura → ultimo → padre)                                ▷ O(1)
20:      if EsHijoIzquierdo?(aPonerUltimo) ∧ EsHijoIzquierdo?(it.estructura → ultimo) then ▷ O(1)
21:        (padreaPonerUltimo → izq) ← it.estructura → ultimo                        ▷ O(1)
22:        (it.estructura → ultimo → padre) ← padreaPonerUltimo                      ▷ O(1)
23:        (aPonerUltimo → padre) ← padreUltimo                                        ▷ O(1)
24:        (padreUltimo → izq) ← aPonerUltimo                                          ▷ O(1)
25:      else
26:        if EsHijoIzquierdo?(aPonerUltimo) ∧ ¬EsHijoIzquierdo?(it.estructura → ultimo) then ▷ O(1)
27:          (padreaPonerUltimo → izq) ← it.estructura → ultimo                        ▷ O(1)
28:          (aPonerUltimo → padre) ← padreUltimo                                    ▷ O(1)
29:          (it.estructura → ultimo → padre) ← padreaPonerUltimo                      ▷ O(1)
30:          (padreUltimo → der) ← aPonerUltimo                                        ▷ O(1)
31:        else
32:          if ¬EsHijoIzquierdo?(aPonerUltimo) ∧ EsHijoIzquierdo?(it.estructura → ultimo) then ▷
33:            (padreaPonerUltimo → izq) ← it.estructura → ultimo                        ▷ O(1)
34:            (aPonerUltimo → padre) ← padreUltimo                                    ▷ O(1)
35:            (it.estructura → ultimo → padre) ← padreaPonerUltimo                      ▷ O(1)
36:            (padreUltimo → izq) ← aPonerUltimo                                        ▷ O(1)
37:          else
38:            if ¬EsHijoIzquierdo?(aPonerUltimo) ∧ ¬EsHijoIzquierdo?(it.estructura → ultimo) then
39:              (padreaPonerUltimo → der) ← (it.estructura → ultimo)                    ▷ O(1)
40:              (it.estructura → ultimo → padre) ← padreaPonerUltimo                  ▷ O(1)
41:              (aPonerUltimo → padre) ← padreUltimo                                ▷ O(1)
42:              (padreUltimo → der) ← 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:     else buscador ← (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) < (cambiado → elemento.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) ← nuevoNodo                                ▷ O(1)
233:         (nuevoNodo → padre) ← abuelo                                ▷ O(1)
234:         (aSwapear → padre) ← nuevoNodo                            ▷ O(1)
235:     else
236:         (abuelo → der) ← nuevoNodo                                ▷ 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) ← nuevoNodo                                ▷ O(1)
313:             (nuevoNodo → padre) ← abuelo                              ▷ O(1)
314:             (aSwapear → padre) ← nuevoNodo                            ▷ O(1)
315:         else
316:             (abuelo → der) ← nuevoNodo                                ▷ 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 raíz (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:  $res \leftarrow \langle p, q \rangle$  ▷  $O(1)$

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( $\alpha$ )) debe ser  $O(1)$

## 8. TAD Iterador Cola

**TAD ITERADOR COLA DE ENTRENADORES( $\alpha$ )**

**géneros**       $itcola(\alpha)$

**igualdad observacional**

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

**exporta**       $itcola(\alpha)$ , generadores, observadores

**usa**             $\text{CONJUNTO}(\alpha)$ ,  $\text{COLA DE PRIORIDAD}(\alpha)$

$colaPrior(\alpha)$

**observadores básicos**

$siguiente : itcola \longrightarrow \alpha$

**generadores**

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

**otras operaciones**

$borrar : itcola(\alpha)it \times colaPrior(\alpha)cp \longrightarrow colaPrior(\alpha) \quad \{siguiente(it) \in elementos(cp)\}$

$elementos : colaPrior(\alpha) \longrightarrow \text{conj}(\alpha)$

$agregarSin : \alpha \times \text{conj}(\alpha) \longrightarrow colaPrior(\alpha)$

**axiomas**       $\forall cp, sp: colaPrior(\alpha), \forall e: \alpha, \forall con: \text{Conj}(\alpha)$

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

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

$elementos(sp) \equiv \text{if } vacia?(sp) \text{ then } \emptyset \text{ else } Ag(proximo(sp), elementos(desencolar(sp))) \text{ fi}$

$agregarSin(e, con) \equiv \text{if } \emptyset?(con) \text{ then } vacia \text{ else } \text{if } dameUno(con) = e \text{ then } agregarSin(e, sinUno(con)) \text{ else } encolar(dameUno(con), agregarSin(e, sinUno(con))) \text{ fi}$

**Fin TAD**