Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2016

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

Trabajo Práctico 2

Diseño

Grupo: 4 gigas de RAM

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

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Informe

1.1. General

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

1.2. Algoritmos privados

- Se inlcuye 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 parámetro un multiconjunto. Reemplazamos ese parámetro por un *juego*, porque usando el *juego* podemos obtener la cantidad de cada especie pokemon
- No teníamos claro como definir la función $<$ (menor) para tupla $\langle$id, cantidad$\rangle$. Por lo que decidimos crear una función publica en juego que toma dos de estas tuplas e indica cual es la menor.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

cantMismaEspecie(**in** p: `pokemon`, **in** j: `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$p \in$ todosLosPokemons(j)}
Post $\equiv$ {$res =_{\text{obs}}$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 procuparnos por ir filtrando los eliminados mientras el usuario usa el iterador. Un problema que surge es que cuando se elimina un jugador, necesitamos eliminar el jugador de esta lista (rápidamente). Es por esto que para cada

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

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

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

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

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

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

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

`Juego` se representa con `pokgo`

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

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

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

donde `jugYCantCapt` es `tupla`(*id*: `nat` ,
cant: `nat`)

Invariante de representacion

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(15) El contador de todo pokenodo es < 10

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Rep : juego j $\longrightarrow$ bool

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

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

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

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

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

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

Algoritmos

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

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

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

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

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

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

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

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

13: Complejidad: $O(log(EC))$
14: Justificaión: 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 $O(log(EC)+copiar(tupla < nat, nat >))$ en el peor caso, y como $copiar(tupla < nat, nat >)$ es $O(1)$ y $1 \leq log(EC)$, por álgebra de órdenes, sumando todos los costos, el costo final el algoritmo es $O(log(EC))$, donde EC es la cantidad máxima de jugadores esperando a capturar un pokemon, (la cantidad máxima de elementos de la cola).

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

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

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

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

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

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

```
    end if
end if
if j.jugadores[e].sanciones < 5 then
    coor posAntes ← Copiar(j.jugadores[e].pos)                                  ▷ O(1)
    bool hayPokAntes ← HayPokemonCercano(posAntes, j)                           ▷ O(1)
    bool hayPokDesp ← HayPokemonCercano(c, j)                                   ▷ O(1)

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

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

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

Justificación: Como copiar(Nat) es $O(1)$ entonces $O(|P| + copiar(Nat))$ es $O(|P|)$ Si el movimiento no fue valido, en el peor caso hay que eliminar al jugador. Esto cuesta $O((PC * |P|) + log(EC))$ pues se ejecuta el ciclo. La segunda parte del algoritmo no se ejecuta si el jugador fue eliminado (pues no entra en el if de sanciones menores a 5), sin embargo, en caso que no fue eliminado el peor caso cuesta $O((PS * |P|) + log(EC))$ si tomo la rama con mayor complejidad. Esto nos da que la complejidad total del algoritmo en peor caso, es la "rama" que mayor complejidad tenga, osea $O(max((PC * |P|) + log(EC), (PS * |P|) + log(EC)))$. Por álgebra de órdenes, usando que el máximo es la suma, esa complejidad es igual a $O((PC * |P|) + log(EC) + (PS * |P|) + log(EC)) = O((PC * |P|) + (PS * |P|) + log(EC)) = O((PC + PS) * |P|) + log(EC))$

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

CasoMov1 : Función privada

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

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

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

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

Aliasing: Produce aliasing, el juego es modificable

```
coor pokePos ← PosPokemonCercano(desp, j)                                       ▷ O(1)
itConj iterPos ← CrearIter(j.posPokemons)                                       ▷ O(1)
while HaySiguente(iterPos) do                                                   ▷ O(PS * |P|)
    nat x ← Latitud(Siguiente(iterPos))                                         ▷ O(1)
    nat y ← Longitud(Siguiente(iterPos))                                        ▷ O(1)
```

```
6:    if Siguiente(iterPos) ≠ pokepos then
7:        (j.pokenodos[x][y])→ contador ← ((j.pokenodos[x][y])→ contador) +1)        ▷ (O(1))
8:    end if
9:    if j.pokenodos[x][y])→ contador = 10 then
10:       SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y], j)→ entrenadores))   ▷ O(|P|)
11:       EliminarSiguiente(iterPos)                                                  ▷ O(1)
12:       j.pokenodos[x][y] ← NULL                                  ▷ Libero la memoria O(1)
13:   else
14:       Avanzar(iterPos)                                                            ▷ O(1)
15:   end if
16: end while
```

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

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

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

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

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

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

CasoMov3 : Función privada
Descripción : Este es el caso en donde el jugador no estaba cerca de ninguno y se movió dentro de algun radio de pokemon
Pre: $j = j_0 \wedge \neg$hayPokemonCercano(ant, j) $\wedge$ hayPokemonCercano($desp$, j)

Post: j_0 = moverse(e, $desp$, j)
Complejidad: $O(PS * |P|)$
Aliasing: Produce aliasing, el juego es modificable

```
1: coor pokePos ← PosPokemonCercano(desp, j)                                   ▷ O(1)
2: itConj iterPos ← CrearIter(j.posPokemons)                                   ▷ O(1)
3: j.jugadores[e].iterAEntrenadores ← Encolar(
4:                 j.pokenodos[Latitud(pokePos)][Longitud(pokePos)], <e, j.jugadores[e].cantCapt>
5:         )                                                                   ▷ O(log(EC) + copiar(tupla < nat, nat >))
6: while HaySiguente(iterPos) do                                               ▷ O(PS * |P|)
7:     nat x ← Latitud(Siguiente(iterPos))                                     ▷ O(1)
8:     nat y ← Longitud(Siguiente(iterPos))                                    ▷ O(1)
9:     if Siguiente(iterPos) = pokepos then
10:        (j.pokenodos[x][y])→ contador ← 0)                                  ▷ (O(1))
11:    end if
12:    if j.pokenodos[x][y])→ contador = 10 then
13:        SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y], j)→ 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: $O(log(EC) + copiar(tupla < nat, nat >))$, donde $copiar(tupla < nat, nat >)$ es $O(1)$. Recorro todos los pokemons salvajes: $O(PC)$, y por cada pokemon en peor caso el contador es mayor o igual a 10, por lo que hay que agregar el pokemon al diccionario del jugador: $O(|P|)$. Este ciclo en total cuesta $O(PC * |P|)$. El resto de las operaciones son $O(1)$ y no agregan complejidad. La complejidad en peor caso del algoritmo es $O((PS * |P|) + log(EC))$ por algebra de ordenes.

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

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

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

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

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

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

```
1:  Eliminar(j.jugadores[e].iterAEntrenadores)                                   ▷ O(log(EC))
2:  coor pokePos ← PosPokemonCercano(desp, j)                                    ▷ O(1)
3:  j.jugadores[e].iterAEntrenadores ← Encolar(
4:                  j.pokenodos[Latitud(pokePos)][Longitud(pokePos)], <e, j.jugadores[e].cantCapt>
5:          )                                                                    ▷ O(log(EC) + copiar(tupla < nat, nat >))
6:  itConj iterPos ← CrearIter(j.posPokemons)                                    ▷ O(1)
7:  while HaySiguente(iterPos) do                                                ▷ O(PS * |P|)
8:      nat x ← Latitud(Siguiente(iterPos))                                      ▷ O(1)
9:      nat y ← Longitud(Siguiente(iterPos))                                     ▷ O(1)
10:     if Siguiente(iterPos) = pokepos then
11:         (j.pokenodos[x][y])→ contador ← 0)                                   ▷ (O(1))
12:     end if
13:     (j.pokenodos[x][y])→ contador ← ((j.pokenodos[x][y])→ contador) +1)      ▷ (O(1))
14:     if j.pokenodos[x][y])→ contador = 10 then
15:         SumarUnoEnJug(j.pokenodos[x][y])→ poke, Proximo(j.pokenodos[x][y], j)→ 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: $O(log(EC) + copiar(tupla < nat, nat >))$ donde $copiar(tupla < nat, nat >)$ es $O(1)$. 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`) $\rightarrow$ 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* $\leftarrow$ Obtener(j.cantPokemon, p) ▷ $O(|P|)$

2: $res \leftarrow 100 - (100 * pokecant$ / j.cantPokemonsTotales) ▷ $O(1)$

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

4: Justificación: j.cantPokemon es un dicciString. La complejidad de buscar (y obtener el significado) en peor caso es la longitud de la string mas larga entre sus claves, eso es O($|P|$). j.cantPokemonsTotales es un dato guardado en la estructura del juego, y se accede en O(1). El resto son una resta, multiplicacion y division, que tambien son O(1). O($|P|$) + O(1) = O($|P|$)

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

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

Complejidad: $O(1)$

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

1: $res \leftarrow ((j$.pokenodos[Latitud(c)][Longitud(c)]) $\rightarrow$ poke) ▷ $O(1)$

Complejidad: $O(1)$

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

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

2: Complejidad: $O(1)$

3: Justificacion: Todas las operaciones son $O(1)$

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

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

2: Complejidad: $O(1)$

3: Justificacion: Todas las operaciones son $O(1)$

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

```
1: coor pokeCoor ← PosPokemonCercano(c, j)                          ▷ O(1)
2: vector(jugador) jugsEnNodo ←DameJugadoresEnPokerango(c, j)       ▷ O(EC)
3: itConj(jugador) itPosibles ← CrearIt(es)                         ▷ O(1)
4: while HaySiguiente(itPosibles) do                                ▷ O(cardinal(es) * EC)
5:     for i ← 0 → Longitud(jugsEnNodo) - 1 do                      ▷ O(EC)
6:         if Siguiente(itPosibles) = jugsEnNodo[i] then            ▷ O(1)
7:             AgregarRapido(res, Siguiente(itPosible))             ▷ O(1)
8:         end if
9:     end for
10:    Avanzar(itPosible)                                           ▷ O(1)
11: end while
```

12: Complejidad: O($cardinal(es)$ * EC)

13: Justificación: Son todas operaciones O(1) excepto aquella en la que se crea un vector con los jugadores en rango y el for que lo recorre que son O(EC). El while principal se itera O($cardinal(es)$) veces, y la operación más costosa dentro de este mismo es O(EC), por lo tanto el while entero cuesta O($cardinal(es)$*EC) operaciones elementales.

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

```
1: nat x ← latitud(c)                                               ▷ O(1)
2: nat y ← longitud(c)                                              ▷ O(1)
3: bool hayPokemon ← false                                          ▷ O(1)
4: if j.pokenodos[x][y] ≠ NULL then                                 ▷ O(1)
5:     hayPokemon ← true
6: end if
7: if x > 0 then
8:     if j.pokenodos[x − 1][y] ≠ NULL then                         ▷ O(1)
9:         hayPokemon ← true
10:    end if
11:    if y > 0 then
12:        if j.pokenodos[x − 1][y − 1] ≠ NULL then                 ▷ O(1)
13:            hayPokemon ← true
14:        end if
15:    end if
16:    if y < tam(m) − 1 then
17:        if j.pokenodos[x − 1][y + 1] ≠ NULL then                 ▷ O(1)
18:            hayPokemon ← true
19:        end if
20:    end if
21:    if x − 1 > 0 then
22:        if j.pokenodos[x − 2][y] ≠ NULL then                     ▷ O(1)
23:            hayPokemon ← true
24:        end if
25:    end if
26: end if
27: if y > 0 then
28:    if j.pokenodos[x][y − 1] ≠ NULL then                         ▷ O(1)
29:        hayPokemon ← true
30:    end if
31:    if y − 1 > 0 then
32:        if j.pokenodos[x][y − 2] ≠ NULL then                     ▷ O(1)
33:            hayPokemon ← true
34:        end if
35:    end if
```

end if
if $y < tam(m) - 1$ **then**
    if j.pokenodos[x][$y + 1$] $\neq$ NULL **then** ▷ $O(1)$
        $hayPokemon \leftarrow$ true
    end if
    if $tam(m) > 1 \wedge y < tam(m) - 2$ **then**
        if j.pokenodos[x][$y + 2$] $\neq$ NULL **then** ▷ $O(1)$
            $hayPokemon \leftarrow$ true
        end if
    end if
end if
if $x < tam(m) - 1$ **then**
    if j.pokenodos[$x + 1$][y] $\neq$ NULL **then** ▷ $O(1)$
        $hayPokemon \leftarrow$ true
    end if
    if $y > 0$ **then**
        if j.pokenodos[$x + 1$][$y - 1$] $\neq$ NULL **then** ▷ $O(1)$
            $hayPokemon \leftarrow$ true
        end if
    end if
    if $y < tam(m) - 1$ **then**
        if j.pokenodos[$x + 1$][$y + 1$] $\neq$ NULL **then** ▷ $O(1)$
            $hayPokemon \leftarrow$ true
        end if
    end if
end if
if $tam(m) > 1 \wedge x < tam(m) - 2$ **then**
    if j.pokenodos[$x + 2$][y] $\neq$ NULL **then** ▷ $O(1)$
        $hayPokemon \leftarrow$ true
    end if
end if
$res \leftarrow hayPokemon$
Complejidad: $O(1)$
Justificación: Reviso 13 posiciones O(1)

iPosPokemonCercano(**in** c : `coor`, **in** j : `juego`) $\rightarrow$ res : coor
nat $x \leftarrow$ latitud(c) ▷ $O(1)$
nat $y \leftarrow$ longitud(c) ▷ $O(1)$
coor $coorConPokemon$ ▷ $O(1)$
if j.pokenodos[x][y] $\neq$ NULL **then** ▷ $O(1)$
    $coorConPokemon \leftarrow$ CrearCoor(x, y) ▷ $O(1)$
end if
if $x > 0$ **then**
    if j.pokenodos[$x - 1$][y] $\neq$ NULL **then** ▷ $O(1)$
        $coorConPokemon \leftarrow$ CrearCoor($x - 1$, y) ▷ $O(1)$
    end if
    if $y > 0$ **then**
        if j.pokenodos[$x - 1$][$y - 1$] $\neq$ NULL **then** ▷ $O(1)$
            $coorConPokemon \leftarrow$ CrearCoor($x - 1$, $y - 1$) ▷ $O(1)$
        end if
    end if
    if $y < tam(m) - 1$ **then**
        if j.pokenodos[$x - 1$][$y + 1$] $\neq$ NULL **then** ▷ $O(1)$
            $coorConPokemon \leftarrow$ CrearCoor($x - 1$, $y + 1$) ▷ $O(1)$

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

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

iPuedoAgregarPokemon(**in** c: `coor`, **in** j: `juego`) $\rightarrow$ 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: Justificación: Todas las operaciones son O(1)

iCantMovimientosParaCaptura(**in** c: `coor`, **in** j: `juego`) $\rightarrow$ 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: Justificación: Todas las operaciones son O(1)

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

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

6: Complejidad: $O(J)$
7: 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 ← CrearIt(j.jugadores[e].pokemons)                                ▷ O(1)
```

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

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

```
1: res ← j.cantPokemon.Obtener(p)                                        ▷ O(|p|)
```

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

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

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

```
1: bool camino ← HayCamino(c, j.jugadores[e].pos, j.mapa)          ▷ O(1)
2: bool distancia ← (DistEuclidea(c, j.jugadores[e].pos) ≤ 100)    ▷ O(1)
3: res ← camino ∧ distancia                                        ▷ O(1)
```

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

---

iSumarUnoEnJug(**in** p : `pokemon`, **in** e : `jugador`, **in/out** j : `juego`)

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

```
1: if Def?(p, j.jugadores[e].pokemons) then                                  ▷ O(|P|)
2:     Definir(j.jugadores[e].pokemons, p, Obtener(p, j.jugadores[e].pokemons) + 1)   ▷ O(|P| + copiar(Nat))
3: else
4:     Definir(j.jugadores[e].pokemons, p, 1)                                ▷ O(|P| + copiar(Nat))
5: end if
```

Complejidad: $O(|P|)$
Justificaion: Como copiar(Nat) es $O(1)$ entonces $O(|P| + copiar(Nat))$ es $O(|P|)$. 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`) → res : vector(jugador)

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

```
1: vector(jugador) jugsRadio ← Vacio()                                       ▷ O(1)
2: nat x ← latitud(c)                                                        ▷ O(1)
3: nat y ← longitud(c)                                                       ▷ O(1)
4: coor coorConPokemon                                                       ▷ O(1)
5: if j.pokenodos[x][y] ≠ NULL then                                          ▷ O(1)
6:     AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x, y), c, j)   ▷ O(EC)
7: end if
8: if x > 0 then
9:     if j.pokenodos[x − 1][y] ≠ NULL then                                  ▷ O(1)
10:        AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x − 1, y), c, j)   ▷ O(EC)
11:    end if
12:    if y > 0 then
13:        if j.pokenodos[x − 1][y − 1] ≠ NULL then                          ▷ O(1)
14:            AgregarAtrasJugsQueEstanEnPos(jugsRadio, crearCoordenada(x − 1, y − 1), c, j)   ▷ O(EC)
```

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

Complejidad: $O(EC)$

Justificacion: Como copiar(Nat) es $O(1)$ entonces $O(|P| + copiar(Nat))$ es $O(|P|)$. En peor caso agrego jugadores de 13 posiciones, donde ese agregar me cuesta O(EC) cada vez. $O(EC) + O(EC) + ... + O(EC) = O(EC)$

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

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

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

Complejidad: $O(EC)$
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)

$<$(**in** *jc*1 : `jugYCantCapt`, **in** *jc*2 : `jugYCantCapt`) $\rightarrow$ *res* : bool

```
1: if jc1.cant = jc2.cant then                                                   ▷ O(1)
2:     res ← (jc1.id < jc2.id)                                                   ▷ O(1)
3: else
4:     res ← (jc1.cant < jc2.cant)                                               ▷ O(1)
5: end if
```

Complejidad: $O(1)$

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(α)) debe ser O(1)
- Avanzar(itLista(α)) debe ser O(1)

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

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

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

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

De Nat
- copiar(Nat) debe ser O(1)

3. Mapa

Interfaz

3.1. Interfaz

se explica con: MAPA.

géneros: `map`.

Operaciones básicas de Mapa

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

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

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

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

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

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

Especificacion de auxiliares usadas en la interfaz

TAD MAPA EXTENSION

extiende MAPA

Otras operaciones

tam : mapa m $\longrightarrow$ nat

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

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

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

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

Fin TAD

Representación

3.2. Representacion

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

`mapa` se representa con `map`

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

Invariante de representacion en castellano

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

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

Invariante de representacion en logica

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

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

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

(4) $((\forall x, y$: nat)(enRango(m, x, y)) $\Rightarrow_{\text{L}}$
($\neg$ grilla[x][y][x][y] $\Rightarrow$ $(\forall z, w$: nat)(enRango(m, z, w)) $\Rightarrow_{\text{L}}$ $\neg$ grilla[x][y][z][w])))

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

(6) $((\forall a, b, c, d, e, f$: nat)
(enRango(m, a, b) $\wedge_{\text{L}}$ m.grilla[a][b][a][b]) $\wedge$
(enRango(m, c, d) $\wedge_{\text{L}}$ m.grilla[c][d][c][d]) $\wedge$
(enRango(m, e, f) $\wedge_{\text{L}}$ m.grilla[e][f][e][f]) $\wedge$
(a $\neq$ c $\vee$ b $\neq$ d) $\wedge$ (c $\neq$ e $\vee$ d $\neq$ f))
$\Rightarrow_{\text{L}}$ ((m.grilla[a][b][c][d] $\wedge$ m.grilla[c][d][e][f]) $\Rightarrow$ m.grilla[a][b][e][f]))

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

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

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

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

Rep : estr m $\longrightarrow$ bool

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

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

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

Algoritmos

3.3. Algoritmos

iCrearMapa()$\rightarrow$ res : map

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

iTam(**in** m : `map`)→ res : nat

1: $res \leftarrow m.tam$ ▷ $O(1)$

Complejidad: $O(1)$

---

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

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

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

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

---

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

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

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

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

3.4. Servicios usados

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

- Vacía() debe ser O(1)

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

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

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

4. Coordenada

Interfaz

4.1. Interfaz

se explica con: COORDENADA.

géneros: `coor`.

Operaciones básicas de Coordenada

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

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

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

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

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

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

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

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

Descripción: devuelve la coordenadas .

Representación

4.2. Representacion

`coor` se representa con `estr`

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

Rep : estr e $\longrightarrow$ bool

Rep(e) $\equiv$ true

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

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

Algoritmos

4.3. Algoritmos

---

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

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

Complejidad: $O(1)$

---

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

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

Complejidad: $O(1)$

---

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

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

Complejidad: $O(1)$

---

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

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

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

---

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

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

Complejidad: $O(1)$

---

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

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

Complejidad: $O(1)$

---

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

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

Complejidad: $O(1)$

---

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

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

Complejidad: $O(1)$

5. DiccString(α)

Interfaz

5.1. Interfaz

parámetros formales
géneros α

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

géneros: `diccString(`α`)`.

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

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

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

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

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

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

Representación

5.2. Representacion

Representación del diccString

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

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

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

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

Invariante de representacion en castellano

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

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

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

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

Algoritmos

5.3. Algoritmos

---

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

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

---

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

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

---

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

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

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

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

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

23: Complejidad: $O(|P| + copiar(\alpha))$
24: Justificaion: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(|p|) = O(|P|)$ mas lo que cueste copiar a.

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

```
1: Nat i ← 0                                                                  ▷ O(1)
2: bool pertenece ← true                                                      ▷ O(1)
3: puntero(nodo) actual ← raiz                                                ▷ O(1)
4: while i < Longitud(p) ∧ pertenece do                                       ▷ Se repite |p| O(1)
5:     if (*actual).siguientes[ord(p[i])] = NULL then                         ▷ O(1)
6:         pertenece ← false                                                  ▷ O(1)
7:     end if
8:     actual ← (*actual).siguientes[ord(p[i])]                               ▷ O(1)
9:     i ← i + 1                                                              ▷ O(1)
10: end while
11: if (*actual).significado = NULL then                                      ▷ O(1)
12:     pertenece ← false                                                     ▷ O(1)
13: end if
14: res ← pertenece                                                           ▷ O(1)
15: Complejidad: O(|P|)
16: Justificaion: Siendo |P| el largo de la clave mas larga, sea cual sea p, |p| ≤ |P| entonces O(|p|) = O(|P|)
```

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

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

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

---

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

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

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

---

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

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

5.4. Servicios usados

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

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

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

6. iterDiccString(α)

Interfaz

6.1. Interfaz

parámetros formales
géneros α

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

géneros: `iterDiccString(`α`)`.

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

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

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

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

Representación

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

6.2. Representacion del iterDiccString

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

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

Invariante de representacion

Rep : estr e $\longrightarrow$ bool

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

Funcion de abstraccion

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

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

Algoritmos

6.3. Algoritmos

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

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

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

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

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

Complejidad: $O(1)$

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

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

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

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

1: Avanzar($iter$.itClave) ▷ $O(1)$

Complejidad: $O(1)$

6.4. Servicios usados

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

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

- Avanzar(itConj(α)) debe ser O(1)

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

7. Cola de Entrenadores

Interfaz

7.1. Interfaz de ColaPrioridad

parámetros formales
géneros α
función $\bullet <\bullet$ (**in** a: α, **in** b: α) $\rightarrow$ res : α
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} a < b\}$
Complejidad: $O(a < b)$
Descripción: Funcion menor de α's

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

géneros: `colaPrioridad`, `itcolaPrioridad`.

Operaciones básicas de ColaPrioridad

Vacia() $\rightarrow$ res : `colaPrioridad`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} vacia\}$
Complejidad: $O(1)$
Descripción: Genera una nueva cola de prioridades vacía

EsVacia?(**in** c: `colaPrioridad`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}}$ vacia?(c)$\}$
Complejidad: $O(1)$
Descripción: Devuelve verdadero si la cola está vacía

Próximo(**in** c: `colaPrioridad`) $\rightarrow$ res : α
Pre $\equiv \{\neg$ vacia?(c)$\}$
Post $\equiv \{res =_{\text{obs}}$ proximo(c)$\}$
Complejidad: $O(1)$
Descripción: Devuelve el próximo de la cola, sin eliminarlo

Encolar(**in/out** $cola$: `colaPrioridad(`α`)`, **in** e: α)
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}}$ encolar(e, $cola$)$\}$
Complejidad: O(log(#($cola$))+ $copiar(e)$)
Descripción: Encola el elemento en la cola de prioridad
Aliasing: El elemento se encola por copia

Desencolar(**in/out** c: `colaPrioridad`)
Pre $\equiv \{c =_{\text{obs}} c_0 \wedge \neg$ vacia(c) $\}$
Post $\equiv \{c =_{\text{obs}}$ desencolar(c_0)$\}$
Complejidad: O(log(#($cola$)))
Descripción: Desencola el próximo de la cola

Extension del tad

TAD Cola de Prioridades Extension

extiende Cola de Prioridades

Otras operaciones

$\#$: colaPrior(α) c $\longrightarrow$ nat

$\#(\text{vacia}) \equiv 0$

$\#(\text{encolar}(e, c)) \equiv 1 + \#(c)$

Fin TAD

Interfaz

7.2. Interfaz de itCola

Operaciones básicas de itCola

Siguiente(**in** it : `itCola(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {Siguiente(it) $\neq$ NULL}
Post $\equiv$ {$res =_{\text{obs}}$ Siguiente(it)}
Complejidad: $O(1)$
Descripción: Devuelve el elemento apuntado por el iterador

Borrar(**in** it : `itCola(`α`)`, **in/out** c : `colaPrioridad(`α`)`)
Pre $\equiv$ {siguiente(it) $\in$ elementos(c)}
Post $\equiv$ {siguiente(it) $\notin$ elementos(c)}
Complejidad: $O(log(EC))$
Descripción: Borra el elemento de la Cola y la reacomoda. El iterador quedará inválido.

Representación

7.3. Representacion de ColaPrioridad

`colaPrioridad` se representa con `estr`

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

donde `nodo` es `tupla`(*elem*: α,
padre: `puntero(nodo)`,
izq: `puntero(nodo)`,
der: `puntero(nodo)`)

Invariante de representación en castellano

- estr es completo a izquierda
- Para todo nodo, su valor es menor que todos los nodos de su izquierda y los de su derecha
- c.raiz no tiene padre, y ningun otro nodo tiene a la raiz como hijo
- Para todo a,b nodos ((a→izq = b) sii (b→padre = a)) y ((a→der = b) sii (b→padre = a))
- Desde cualquier nodo se puede llegar a la raiz yendo hacia arriba
- c.ultimo es el nodo de mas abajo a la derecha
- c.raiz es NULL sii c.ultimo es NULL
- No hay ciclos

Abs : estr c $\longrightarrow$ colaPrior(α) {Rep(c)}

Abs(c) $\equiv$ *cola* : colaPrior(α) /
vacia?(*cola*) $\iff$ (c.raiz = NULL) $\wedge_{\text{L}}$
proximo(*cola*) $=_{\text{obs}}$ (c.raiz $\rightarrow$ elem)

Representación

7.4. Representacion del iterador

`itCola` se representa con `estr`

donde `estr` es `tupla`(*siguiente*: `puntero(nodo)`, *estructura*: `puntero(colaPrioridad)`)

Rep : estr e $\longrightarrow$ bool

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

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

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

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

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

Algoritmos

7.5. Algoritmos ColaPrioridad

iVacia() → *res* : colaPrioridad
1: *res* ← ⟨ NULL, NULL ⟩ ▷ $O(1)$

Complejidad: $O(1)$

iEsVacia?(**in** *c*: `colaPrioridad(`α`)`) → *res* : bool
1: *res* ← (*c*.raiz = NULL) ▷ $O(1)$

Complejidad: $O(1)$

iPróximo(**in** *c*: `colaPrioridad(`α`)`) → *res* : α
1: *res* ← (*c*.raiz → elem) ▷ $O(1)$

Complejidad: $O(1)$

iEncolar(**in/out** *c*: `colaPrioridad(`α`)`, **in** *e*: α) → *res* : itCola(α)
1: **if** EsVacia?(*c*) **then**
2: nodo *nuevoNodo* ← ⟨ e, NULL, NULL, NULL ⟩ ▷ $O(copiar(e))$
3: *c*.raiz ← &*nuevoNodo* ▷ $O(1)$
4: *c*.ultimo ← &*nuevoNodo* ▷ $O(1)$
5: **else**
6: **if** EstaCompleto(*c*) **then** ▷ $O(log(\#(c)))$
7: nodo *nodoActual* ← ∗(*c*.raiz) ▷ $O(1)$
8: **while** *nodoActual* → izq ≠ NULL **do** ▷ $O(log(\#(c)))$
9: *nodoActual* ← (*nodoActual* → izq) ▷ $O(1)$
10: **end while**
11: nodo *nuevoNodo* ← ⟨ e, *nodoActual*, NULL, NULL ⟩ ▷ $O(1)$
12: (*nodoActual* → izq) ← &*nuevoNodo* ▷ $O(1)$
13: *c*.ultimo ← &*nuevoNodo* ▷ $O(1)$
14: **else**
15: **if** EsHijoIzquierdo(*c*.ultimo) **then** ▷ $O(1)$
16: nodo *nuevoNodo* ← ⟨ e, *c*.ultimo → padre, NULL, NULL ⟩ ▷ $O(1)$
17: (*c*.ultimo → padre → der) ← &*nuevoNodo* ▷ $O(1)$
18: *c*.ultimo ← &*nuevoNodo* ▷ $O(1)$
19: **else**
20: EncolarSiUltimoEsHijoDerechoNoCompleto(*c*, *e*) ▷ $O(log(\#(c)))$
21: **end if**
22: **end if**
23: **end if**
24: nodo m ← *(*c*.ultimo) ▷ $O(1)$
25: SiftUp(*m*, *c*) ▷ $O(log(\#(c)))$
26: puntero(nodo) *p* ← &*m* ▷ $O(1)$
27: res ← crearItCola(*p*, *c*) ▷ $O(1)$

Complejidad: $O(log(\#(c)) + copiar(e))$

Justificación: El While es O(log(#c)) porque como el nodo forma parte de árbol, yendo siempre hacia abajo a la izquierda recorro como máximo la altura del árbol: log(#c). La complejidad del algoritmo en peor caso es la suma de las complejidades: $O(1) + O(1) + ... + O(1) + O(copiar(e)) + O(log(\#(c))) + O(log(\#(c))) + O(log(\#(c)))$ $= O(log(\#(c)) + copiar(e))$ por álgebra de órdenes.

---

iDesencolar(**in/out** c: `colaPrioridad(`α`)`)

1: SwapNodos(*(c.raiz), *(c.ultimo), c) ▷ $O(1)$
2: EliminarUltimo(c) ▷ $O(log(\#c))$
3: SiftDown(*(c.raiz), c) ▷ $O(log(\#c))$

Complejidad: $O(1)$
Justificación: $O(1) + O(log(\#c)) + O(log(\#c)) = 2*O(log(\#c)) = O(log(\#c))$

Funcion privada: EstaCompleto
Descripción: devuelve *true* si el arbol esta completo, es decir, si el nivel mas profundo del arbol tiene todos los nodos posibles
Pre ≡ {¬ vacía?(c)}
Post ≡ {Res es true sii el ultimo es el de mas abajo a la derecha}

---

iEstaCompleto(**in** c: `colaPrioridad(`α`)`) → res : bool

1: puntero(nodo) $nodoActual \leftarrow c$.raiz ▷ $O(1)$
2: **while** ($nodoActual \rightarrow$ der) ≠ NULL **do** ▷ En peor caso recorro la altura del arbol $O(log(\#c))$
3: $nodoActual \leftarrow (nodoActual \rightarrow$ der) ▷ $O(1)$
4: **end while**
5: $res \leftarrow (nodoActual = c$.ultimo) ▷ $O(1)$

Complejidad: $O(log(\#c))$

Funcion privada: EsHijoIzquierdo
Descripción: Devuelve *true* si el nodo es el hijo izquierdo de su padre
Pre ≡ {true}
Post ≡ {Res es true sii n es hijo izquierdo}

---

iEsHijoIzquierdo(**in** n: `nodo`) → res : bool

1: **if** n.padre = NULL **then** ▷ $O(1)$
2: $res \leftarrow$ false ▷ $O(1)$
3: **else**
4: $res \leftarrow$ (*(n.padre → izq) = n) ▷ $O(1)$
5: **end if**

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

Funcion privada: EsHijoDerecho
Descripción: Devuelve *true* si el nodo es el hijo derecho de su padre

Pre $\equiv$ {true}
Post $\equiv$ {Res es true sii n es hijo derecho}

---

iEsHijoDerecho(**in** n : `nodo`) $\rightarrow$ res : bool

```
1: if n.padre = NULL then                                   ▷ O(1)
2:     res ← false                                          ▷ O(1)
3: else
4:     res ← (*(n.padre → der) = n)                         ▷ O(1)
5: end if
```

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

Funcion privada: EncolarSiUltimoEsHijoDerechoNoCompleto
Descripción: En el caso que el *c*ultimo es un hijo derecho (y el arbol no era completo), para buscar el nodo que va a ser el nuevoUltimo, es necesario hacer un recorrido del árbol. Esto se ve mas claramente cuando el heap tiene mas de 4 niveles, y el *c*.ultimo apunta a un hijo derecho del centro. Agrega en la posición correcta un nuevo nodo con el valor e
Pre $\equiv$ {c no esta completo y c.ultimo es hijo derecho}
Post $\equiv$ {c.ultimo es actualizado y agrega un nuevoNodo con el valor e al final del heap}

---

iEncolarSiUltimoEsHijoDerechoNoCompleto(**in/out** c : `colaPrioridad(`α`)`, **in** e : α)

```
1: puntero(nodo) nuevoUltimoPadre ← c.ultimo                                  ▷ O(1)
2: while ¬ EsHijoIzquierdo(*nuevoUltimoPadre) do      ▷ Peor caso recorro la altura O(log(#(c)))
3:     nuevoUltimoPadre ← (nuevoUltimoPadre → padre)                          ▷ O(1)
4: end while
5:
6: nuevoUltimoPadre ← (nuevoUltimoPadre → padre → der)                        ▷ O(1)
7:
8: while (nuevoUltimoPadre → izq) ≠ NULL do           ▷ Peor caso recorro la altura O(log(#(c)))
9:     nuevoUltimoPadre ← (nuevoUltimoPadre → izq)                            ▷ O(1)
10: end while
11:
12: nodo nuevoNodo ← ⟨ e, nuevoUltimoPadre , NULL, NULL ⟩                     ▷ O(1)
13: (nuevoUltimoPadre → izq) ← &nuevoNodo                                     ▷ O(1)
14: c.ultimo ← nuevoNodo                                                      ▷ O(1)
```

Complejidad: $O(log(\#(c)))$
Justificacion: En el peor caso tengo que recorrer hacia arriba toda la altura del arbol (log(#(c))) por ser completo) y luego bajar para llegar al nuevo ultimo. El resto de las operaciones son $O(1)$. En total la complejidad es: $O(1) + O(1) + .. + O(1) + O(log(\#(c))) + O(log(\#(c))) = O(log(\#(c)))$

Funcion privada: HayUnicoNodoEnUltimoNivel
Descripción: Devuelve *true* si el nivel mas profundo del arbol tiene un único nodo
Pre $\equiv$ {$\neg$ vacía?(c)}
Post $\equiv$ {Res es true sii el ultimo es el de mas abajo a la izquierda}

iHayUnicoNodoEnUltimoNivel(**in** c: `colaPrioridad(`α`)`) $\rightarrow$ res : bool

```
1: puntero(nodo) nodoActual ← c.raiz                                   ▷ O(1)
2: while (nodoActual → izq) ≠ NULL do   ▷ En peor caso recorro la altura del arbol O(log(#c))
3:     nodoActual ← (nodoActual → izq)                                 ▷ O(1)
4: end while
5: res ← (nodoActual = c.ultimo)                                       ▷ O(1)
```

Complejidad: $O(log(\#c))$

Funcion privada: SwapNodos
Descripción: Dados dos nodos cualesquiera que esten en el heap, los swappea reencadenando punteros (encargandose de reacomodar c.raiz y c.ultimo en caso de ser necesario)
Pre $\equiv$ $\{a =_{\text{obs}} a_0 \wedge b =_{\text{obs}} b_0 \wedge a \neq \text{NULL} \wedge b \neq \text{NULL} \wedge a \in c \wedge b \in c\}$
Post $\equiv$ {Los nodos a y b son swappeados reordenando los punteros}

iSwapNodos(**in/out** a: `nodo`, **in/out** b: `nodo`, **in/out** c: `colaPrioridad`)

```
1: if *(a.padre) = b then                                              ▷ O(1)
2:     SwapConHijo(a, b)                                               ▷ O(1)
3: else
4:     if *(b.padre) = a then                                          ▷ O(1)
5:         SwapConHijo(b, a)                                           ▷ O(1)
6:     else
7:         SwapDisjunto(a, b)                                          ▷ O(1)
8:     end if
9: end if
10: if *(c.raiz) = a then                                              ▷ O(1)
11:     c.raiz ← &b                                                    ▷ O(1)
12: else
13:     if *(c.raiz) = b then                                          ▷ O(1)
14:         c.raiz ← &a                                                ▷ O(1)
15:     end if
16: end if
17:
18: if *(c.ultimo) = a then                                            ▷ O(1)
19:     c.ultimo ← &b                                                  ▷ O(1)
20: else
21:     if *(c.ultimo) = b then                                        ▷ O(1)
22:         c.ultimo ← &a                                              ▷ O(1)
23:     end if
24: end if
```

Complejidad: $O(1)$

Funcion privada: SwapConHijo
Descripción: Realiza un swap entre los nodos a y b reencadenando punteros, en el caso que a es padre de b
Pre $\equiv$ {a es padre de b}
Post $\equiv$ {Los nodos a y b son swappeados reordenando los punteros}

iSwapConHijo(**in/out** a : `nodo`, **in/out** b : `nodo`)

```
1: if *(a.izq) = b then                                   ▷ O(1)
2:     SwapConHijoIzquierdo(a, b)                         ▷ O(1)
3: else
4:     if *(a.der) = b then                               ▷ O(1)
5:         SwapConHijoDerecho(a, b)                       ▷ O(1)
6:     end if
7: end if
```

Complejidad: $O(1)$

Funcion privada: SwapConHijoIzquierdo
Descripción: Realiza un swap entre los nodos a y b reencadenando punteros, en el caso de que b es el hijo izquierdo de a
Pre $\equiv$ {b es hijo izquierdo de a}
Post $\equiv$ {Los nodos a y b son swappeados reordenando los punteros}

iSwapConHijoIzquierdo(**in/out** a : `nodo`, **in/out** b : `nodo`)

```
1: puntero(nodo) tmpderA ← (a.der)                        ▷ O(1)
2: (a.der) ← (b.der)                                      ▷ O(1)
3: if a.der ≠ NULL then
4:     (a.der → padre) ← &a                               ▷ O(1)
5: end if
6: (b.der) ← tmpderA                                      ▷ O(1)
7: if b.der ≠ NULL then
8:     (b.der → padre) ← &b                               ▷ O(1)
9: end if
10:
11: if EsHijoIzquierdo(a) then
12:     (a.padre → izq) ← &b                               ▷ O(1)
13: else
14:     if EsHijoDerecho(a) then
15:         (a.padre → der) ← &b                           ▷ O(1)
16:     end if
17: end if
18:
19: (b.padre) ← (a.padre)                                 ▷ O(1)
20: if b.izq ≠ NULL then
21:     (b.izq → padre) ← &a                               ▷ O(1)
22: end if
23: (a.izq) ← (b.izq)                                     ▷ O(1)
24: (b.izq) ← &a                                          ▷ O(1)
25: (a.padre) ← &b                                        ▷ O(1)
```

Complejidad: $O(1)$

Funcion privada: SwapConHijoDerecho
Descripción: Realiza un swap entre los nodos a y b reencadenando punteros, en el caso de que b es el hijo derecho de a
Pre $\equiv$ {b es hijo derecho de a}
Post $\equiv$ {Los nodos a y b son swappeados reordenando los punteros}

---

iSwapConHijoDerecho(in/out a : `nodo`, **in/out** b : `nodo`)

```
1:  puntero(nodo) tmpizqA ← (a.izq)                                  ▷ O(1)
2:  (a.izq) ← (b.izq)                                                ▷ O(1)
3:  if a.izq ≠ NULL then
4:      (a.izq → padre) ← &a                                         ▷ O(1)
5:  end if
6:  (b.izq) ← tmpizqA                                                ▷ O(1)
7:  if b.izq ≠ NULL then
8:      (b.izq → padre) ← &b                                         ▷ O(1)
9:  end if
10:
11: if EsHijoIzquierdo(a) then
12:     (a.padre → izq) ← &b                                         ▷ O(1)
13: else
14:     if EsHijoDerecho(a) then
15:         (a.padre → der) ← &b                                     ▷ O(1)
16:     end if
17: end if
18:
19: (b.padre) ← (a.padre)                                            ▷ O(1)
20: if b.der ≠ NULL then
21:     (b.der → padre) ← &a                                         ▷ O(1)
22: end if
23: (a.der) ← (b.der)                                                ▷ O(1)
24: (b.der) ← &a                                                     ▷ O(1)
25: (a.padre) ← &b                                                   ▷ O(1)
```

Complejidad: $O(1)$

Funcion privada: SwapDisjunto
Descripción: Realiza un swap entre a y b, reencadenando punteros, en el caso en el que a no es hijo de b ni b es hijo de a
Pre ≡ {a distinto de b. a no es hijo de b. b no es hijo de a.}
Post ≡ {Los nodos a y b son swappeados reordenando los punteros}
iSwapDisjunto(**in**/**out** a : `nodo`, **in**/**out** b : `nodo`)

```
1:  if EsHijoIzquierdo(b) then
2:      (b.padre → izq) ← &a                                   ▷ O(1)
3:  else
4:      if EsHijoDerecho(b) then
5:          (b.padre → der) ← &a                               ▷ O(1)
6:      end if
7:  end if
8:  if b.der ≠ NULL then
9:      (b.der → padre) ← &a                                   ▷ O(1)
10: end if
11: if b.izq ≠ NULL then
12:     (b.izq → padre) ← &a                                   ▷ O(1)
13: end if
14:
15: if EsHijoIzquierdo(a) then
16:     (a.padre → izq) ← &b                                   ▷ O(1)
17: else
18:     if EsHijoDerecho(b) then
19:         (a.padre → der) ← &b                               ▷ O(1)
20:     end if
21: end if
22: if a.der ≠ NULL then
23:     (a.der → padre) ← &b                                   ▷ O(1)
24: end if
25: if a.izq ≠ NULL then
26:     (a.izq → padre) ← &b                                   ▷ O(1)
27: end if
28:
29: puntero(nodo) tmpPadreB ← (b.padre)                        ▷ O(1)
30: puntero(nodo) tmpIzqB ← (b.izq)                            ▷ O(1)
31: puntero(nodo) tmpDerB ← (b.der)                            ▷ O(1)
32:
33: (b.padre) ← (a.padre)                                      ▷ O(1)
34: (b.izq) ← (a.izq)                                          ▷ O(1)
35: (b.der) ← (a.der)                                          ▷ O(1)
36: (a.padre) ← tmpPadreB                                      ▷ O(1)
37: (a.izq) ← tmpIzqB                                          ▷ O(1)
38: (a.der) ← tmpDerB                                          ▷ O(1)
```

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

Funcion privada: EliminarUltimo
Descripción: Elimina el ultimo nodo del heap, actualizando c.ultimo
Pre ≡ { #c >0 }
Post ≡ {El nodo c.ultimo se elimina, y es actualizado}

iEliminarUltimo(in/out c: `colaPrioridad(`α`)`)

```
1: if c.ultimo = c.raiz then
2:     c.ultimo ← NULL                                          ▷ Libero memoria O(1)
3:     c.raiz ← NULL                                            ▷ O(1)
4: else
5:     if EsHijoDerecho(c.ultimo) then
6:         puntero(nodo) antiguoUltimo ← c.ultimo               ▷ O(1)
7:         c.ultimo ← (c.ultimo → padre → izq)                  ▷ O(1)
8:         (c.ultimo → padre → der) ← NULL                      ▷ O(1)
9:         antiguoUltimo ← NULL                                 ▷ O(1)
10:    else
11:        if HayUnicoNodoEnUltimoNivel(c) then                 ▷ O(log(#c))
12:            (c.ultimo → padre → izq) ← NULL                  ▷ O(1)
13:            c.ultimo ← NULL                                  ▷ Libero memoria O(1)
14:            puntero(nodo) nodoActual ← c.raiz                ▷ O(1)
15:            while (nodoActual → der) ≠ NULL do               ▷ O(log(#c))
16:                nodoActual ← (nodoActual → der)              ▷ O(1)
17:            end while
18:            c.ultimo ← nodoActual                            ▷ O(1)
19:        else
20:            puntero(nodo) nuevoUltimo ← c.ultimo             ▷ O(1)
21:            while ¬EsHijoDerecho(*nuevoUltimo) do            ▷ En peor caso recorro toda la altura O(log(#c))
22:                nuevoUltimo ← (nuevoUltimo → padre)          ▷ O(1)
23:            end while
24:            nuevoUltimo ← (nuevoUltimo → padre → izq)        ▷ O(1)
25:            while nuevoUltimo ≠ NULL do                      ▷ En peor caso recorro toda la altura O(log(#c))
26:                nuevoUltimo ← (nuevoUltimo → der)            ▷ O(1)
27:            end while
28:            c.ultimo ← NULL                                  ▷ Libero memoria O(1)
29:            c.ultimo ← nuevoUltimo                           ▷ O(1)
30:        end if
31:    end if
32: end if
```

Complejidad: O(log(#c))
Justificacion: Para la complejidad del peor caso, tomo la rama que mas complejidad tiene. Algunas ramas son O(1) (por ser suma de O(1)) y las demas son en peor caso: $O(1) + 2{*}O(log(\#c)) = O(log(\#c))$

Funcion privada: EsMenorNodos
Descripción: Devuelve *true* si el nodo apuntado por a tiene un elemento que es menor o igual al elemento apuntado por b
Pre ≡ {true}
Post ≡ {devuelve true si el nodo apuntado por a tiene un elemento y es menor o igual al elemento del nodo apuntado por b. Si ambos son null res es false}

iEsMenorNodos(**in** a : `puntero(nodo)`, **in** b : `puntero(nodo)`) $\rightarrow$ res : bool

```
1:  if a ≠ NULL ∨ b ≠ NULL then                                              ▷ O(1)
2:      if a = NULL ∧ b ≠ NULL then                                          ▷ O(1)
3:          res ← false                                                      ▷ O(1)
4:      else
5:          if a ≠ NULL ∧ b = NULL then                                      ▷ O(1)
6:              res ← true                                                   ▷ O(1)
7:          else
8:              if ((a → elem) < (b → elem)) ∨ (a → elem) = (b → elem) then  ▷ O(1)
9:                  res ← true                                               ▷ O(1)
10:             end if
11:         end if
12:     end if
13: else
14:     res ← false                                                          ▷ O(1)
15: end if
```

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

Funcion privada: SiftUp
Descripción: Hace siftUp con el nodo pasado por parámetro
Pre $\equiv$ {El nodo está en la estructura de C}
Post $\equiv$ {El nodo n se fue swapeando con su padre mientras su elemento era menor}

iSiftUp(**in/out** n : `nodo`, **in/out** c : `colaPrioridad(`α`)`)

```
1: while (n.padre ≠ NULL) ∧_L ((n.elem) < (n.padre → elem)) do   ▷ O(log(#c))
2:     SwapNodos(n, *(n.padre), c)                               ▷ O(1)
3: end while
```

Complejidad: O(log(#C))
Justificacion: En el peor caso se recorre toda la altura del arbol: O(log(#C))

Funcion privada: SiftDown
Descripción: Hace siftDown con el nodo pasado por parametros
Pre $\equiv$ {El nodo está en la estructura de C}
Post $\equiv$ {El nodo n se fue swapeando con sus hijos mientras era menor}

iSiftDown(**in/out** n : `nodo`, **in/out** c : `colaPrioridad(`α`)`)

```
1: bool sigueBajando ← true                                              ▷ O(1)
2: while sigueBajando do                                                 ▷ O(log(#c))
3:     if EsMenorNodos(n.izq, n.der) then                                ▷ O(1)
4:         if (n.izq → elem) <(n.elem) then                              ▷ O(1)
5:             SwapNodos(n, *(n.izq), c)                                 ▷ O(1)
6:         else
7:             sigueBajando ← false                                      ▷ O(1)
8:         end if
9:     else
10:        if EsMenorNodos(n.der, n.izq) then                            ▷ O(1)
11:            if (n.der → elem) <(n.elem) then                          ▷ O(1)
12:                SwapNodos(n, *(n.der), c)                             ▷ O(1)
13:            else
14:                sigueBajando ← false                                  ▷ O(1)
15:            end if
16:        else
17:            sigueBajando ← false                                      ▷ O(1)
18:        end if
19:    end if
20: end while
```

Complejidad: O(log(#c))
Justificacion: En el peor caso se recorre toda la altura del arbol: O(log(#c))

Algoritmos

7.6. Algoritmos itCola

iSiguiente(**in/out** it : `itCola(`α`)`) $\rightarrow$ res : α

```
1: res ← (it.siguiente → elem)                                           ▷ O(1)
```

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

iBorrar(**in** it : `itCola(`α`)`, **in/out** c : `colaPrioridad(`α`)`)

```
1: SwapNodos(*(it.siguiente), *(c.ultimo), c)                            ▷ O(1)
2: EliminarUltimo(c)                                                     ▷ O(log(EC))
3: SiftUp(*(it.siguiente), c)                                            ▷ O(log(EC))
```

Complejidad: $O(log(EC))$
Justificación $O(1) + O(log(EC)) + O(log(EC)) = O(log(EC))$

Función privada: crearItCola
Descripción: Crea un iterador cuyo siguiente es el nodo pasado por parámetro.
Pre $\equiv$ {$p \neq$ NULL $\wedge_{\text{L}}$ el nodo apuntado por p está en la estructura de c}
Post $\equiv$ {El siguiente del iterador es el puntero pasado por parámetro y la estructura es un puntero a la estructura pasada por parámetro. }

icrearItCola(**in** p: `puntero(nodo)`, **in** c: `colaPrioridad(`α`)`) $\rightarrow$ res : itCola(α)

1: puntero(colaPrioridad(α)) puntCola $\leftarrow$ &c ▷ $O(1)$
2: res $\leftarrow$ <p, puntCola> ▷ $O(1)$

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

8. TAD Iterador Cola

TAD ITERADOR COLA DE PRIORIDAD(α)

géneros itcola(α)

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

exporta itcola(α), generadores, observadores

usa CONJUNTO(α), COLA DE PRIORIDAD(α)

colaPrior(α)

observadores básicos

siguiente	: itCola	$\longrightarrow$	α

generadores

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

otras operaciones

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

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

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

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

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

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

Fin TAD