DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Especificación

16 de octubre de 2016

Algoritmos y Estructuras de Datos II
Segundo Cuatrimestre de 2016

Grupo "Algo Habrán Hecho (por las Estructuras de Datos)"

Integrante	LU	Correo electrónico
Barylko, Roni Ariel	750/15	rbarylko@dc.uba.ar
Giudice, Carlos	694/15	cgiudice@dc.uba.ar
Szperling, Sebastián Ariel	763/15	sszperling@dc.uba.ar
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Módulo Juego

La cantidad de cada pokémon se guarda en un diccionario basado en trie, usando los nombres como claves. Sus posiciones se guardan en un Conjunto Lineal con inserción rápida, y en el mismo mapa se guarda el pokémon que se ubica en dicha posición (este valor se invalida si la posición no está en el conjunto, como cuando se captura el pokémon).

Utilizaremos la misma notación para complejidades que en el enunciado:

- J es la cantidad total de jugadores que fueron agregados al juego.
- $|P|$ es el nombre más largo para un pokémon en el juego.
- EC es la máxima cantidad de jugadores esperando para atrapar un pokémon.
- PS es la cantidad de pokémon salvajes.
- PC es la máxima cantidad de pokémon capturados por un jugador.

Interfaz

se explica con: Juego.

géneros: `juego`, `itJugadores`.

servicios usados: Coordenada, Mapa, Vector(α), DiccionarioString(α), DiccionarioPrioridad, Conjunto Lineal(α), ConjuntoJugadores

CrearJuego(**in** m : `mapa`) $\rightarrow$ res : `juego`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearJuego(m)}
Complejidad: $O(alto(m) \times ancho(m))$
Descripción: crea un juego vacío usando el mapa provisto.
Aliasing: se guarda una copia de m.

AgregarPokémon(**in/out** j : `juego`, **in** pk : `pokemon`, **in** c : `coor`)
Pre $\equiv$ {$j =_{\text{obs}} j_0 \wedge$ puedoAgregarPokémon(c, j_0)}
Post $\equiv$ {$j =_{\text{obs}}$ agregarPokémon(pk, c, j_0)}
Complejidad: $O(|P|)$
Descripción: agrega un pokemon al juego.

AgregarJugador(**in/out** j : `juego`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$j =_{\text{obs}} j_0$}
Post $\equiv$ {$j =_{\text{obs}}$ agregarJugador(j)}
Complejidad: $O(J)$
Descripción: registra un nuevo jugador y devuelve su ID.

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

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

Moverse(**in/out** j : `juego`, **in** e : `jugador`, **in** c : `coor`)
Pre $\equiv$ {$j =_{\text{obs}} j_0 \wedge e \in$ jugadores(j_0) $\wedge_{\text{L}}$ conectado(e, j_0) $\wedge$ posExistente(c, mapa(j_0))}
Post $\equiv$ {$j =_{\text{obs}}$ moverse(e, c, j_0)}
Complejidad: $O((PS + PC)|P| + log(EC))$

Descripción: mueve el jugador a la posición elegida. Si el movimiento es ilegal, sanciona al jugador o lo expulsa si excede el límite de sanciones. Aumenta los contadores de captura para otros jugadores donde corresponde y puede provocar la captura de pokémons.

MAPA(**in** j : `juego`) $\rightarrow$ res : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ {alias(res = mapa(j))}
Complejidad: $O(1)$
Descripción: devuelve el mapa del juego.
Aliasing: res no es modificable.

JUGADORES(**in** j : `juego`) $\rightarrow$ res : `itUni(jugador)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearItUni(jugadores(j))}
Complejidad: $O(1)$
Descripción: devuelve un iterador del conjunto de jugadores registrados (no expulsados).
Aliasing: el iterador se invalida si se expulsa al siguiente jugador del iterador, o si se agrega un nuevo jugador y el iterador había llegado al final. Además, siguientes(res) podría cambiar completamente ante cualquier operación que modifique la lista de jugadores.

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 si el jugador está conectado al juego.

SANCIONES(**in** j : `juego`, **in** e : `jugador`) $\rightarrow$ 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 que el jugador posee.

POSICION(**in** j : `juego`, **in** e : `jugador`) $\rightarrow$ res : `coor`
Pre $\equiv$ {$e \in$ jugadores(j)}
Post $\equiv$ {$res =_{\text{obs}}$ posicion(e, j)}
Complejidad: $O(1)$
Descripción: devuelve la posición actual del jugador.

POKÉMONS(**in** j : `juego` **in** e : `jugador`) $\rightarrow$ res : `itBi(tupla(pokemon, nat))`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearItBi(<>, pokemons(j)) $\wedge$ alias(esPermutacion(SecuSuby(res), pokemons(j)))}
Complejidad: $O(1)$
Descripción: devuelve un iterador al conjunto de los pokémons capturados por un jugador, y la cantidad de los mismos.
Aliasing: el iterador se invalida si y sólo si se elimina el elemento siguiente del iterador.

EXPULSADOS(**in** j : `juego`) $\rightarrow$ res : `itUni(jugador)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearItUni(expulsados(j))}
Complejidad: $O(1)$
Descripción: devuelve un iterador del conjunto de jugadores expulsados.
Aliasing: el iterador se invalida si se expulsa al siguiente jugador del iterador, o si se agrega un nuevo jugador y el iterador había llegado al final. Además, siguientes(res) podría cambiar completamente ante cualquier operación que modifique la lista de jugadores.

POSCONPOKÉMONS(**in** j : `juego`) $\rightarrow$ res : `conj(coor)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(res = posConPokémons(j))}
Complejidad: $O(1)$
Descripción: devuelve el conjunto de posiciones que contienen un pokémon.

Aliasing: res no es modificable.

PokémonEnPos(**in** j : `juego`, **in** c : `coor`) → res : `pokemon`
Pre ≡ {$c \in$ posConPokemon(j)}
Post ≡ {$res =_{\text{obs}}$ pokémonEnPos(c, j)}
Complejidad: $O(1)$
Descripción: devuelve el pokémon que se encuentra en la posición.

PuedoAgregarPokémon(**in** j : `juego`, **in** c : `coor`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ puedoAgregarPokémon(c, j)}
Complejidad: $O(PS)$
Descripción: devuelve si un nuevo pokémon puede ser agregado a esa posición (no debe haber ningún pokémon cerca).

HayPokémonCercano(**in** j : `juego`, **in** c : `coor`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ hayPokémonCercano(c, j)}
Complejidad: $O(PS)$
Descripción: devuelve si hay algún pokémon cerca de la posición.

PosPokémonCercano(**in** j : `juego`, **in** c : `coor`) → res : `coor`
Pre ≡ {hayPokémonCercano(c, j)}
Post ≡ {$res =_{\text{obs}}$ posPokémonCercano(c, j)}
Complejidad: $O(PS)$
Descripción: devuelve el pokémon que se encuentra cerca de la posición.

EntrenadoresPosibles(**in** j : `juego`, **in** c : `coor`) → res : `conj(jugador)`
Pre ≡ {hayPokémonCercano(c, j)}
Post ≡ {$res =_{\text{obs}}$ entrenadoresPosibles(c, j)}
Complejidad: $O(EC)$
Descripción: devuelve el conjunto de jugadores esperando a capturar el pokémon que se encuentra en la posición.
Aliasing: res no es modificable.

IndiceRareza(**in** j : `juego`, **in** pk : `pokemon`) → res : `nat`
Pre ≡ {$pk \in$ pokemons(j)}
Post ≡ {$res =_{\text{obs}}$ indiceRareza(pk, j)}
Complejidad: $O(|P|)$
Descripción: calcula el índice de rareza de un pokémon.

CantPokémonsTotales(**in** j : `juego`) → res : `nat`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ cantPokémonsTotales(j)}
Complejidad: $O(1)$
Descripción: devuelve la cantidad de pokémons en juego.

CantMismaEspecie(**in** j : `juego`, **in** pk : `pokemon`) → res : `nat`
Pre ≡ {$pk \in$ pokemons(j)}
Post ≡ {$res =_{\text{obs}}$ cantMismaEspecie(j)}
Complejidad: $O(|P|)$
Descripción: devuelve la cantidad de pokémons de la especie especificada en juego.

Operaciones del iterador de jugadores

El iterador que presentamos permite recorrer tanto los jugadores registrados válidos como los expulsados de forma unidireccional. El iterador es solo un contador (devuelve las IDs de los jugadores, no su detalle).

CrearIt(**in** j : `juego`, **in** $elim?$: `bool`) → res : `itJugadores`
Pre ≡ {true}
Post ≡ {true}
Complejidad: $O(1)$

Descripción: crea un iterador unidireccional de los jugadores validos o expulsados.
Aliasing: el iterador se invalida si se expulsa al siguiente jugador del iterador, o si se agrega un nuevo jugador y el iterador había llegado al final. Además, siguientes(res) podría cambiar completamente ante cualquier operación que modifique la lista de jugadores.

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

Actual(**in** it : `itJugadores`) $\rightarrow$ res : `jugador`
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {$res =_{\text{obs}}$ Actual(it)}
Complejidad: $O(J)$)
Descripción: devuelve el elemento siguiente a la posición del iterador.

Avanzar(**in/out** it : `itJugadores`)
Pre $\equiv$ {$it = it_0 \wedge$ HayMas?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $O(J)$
Descripción: avanza a la posición siguiente del iterador.

Representación

Representación de Juego

`Juego` **se representa con** `pokego`

donde `pokego` es `tupla`(*mapa*: `mapa` , *pokemons*: `diccString(nat)` , *posConPokemons*: `conj(coor)` , *jugadores*: `vector(puntero(infoJugador))` , *grillaPos*: `arreglo_dimensionable de (arreglo_dimensionable de infoPos)` , *cantPokemons*: `nat`)
donde `infoJugador` es `tupla`(*sanciones*: `nat` , *conectado?*: `bool` , *posicion*: `coor` , *cantPokemons*: `nat` , *pokemonsCapturados*: `diccString(nat)`)
donde `infoPos` es `tupla`(*hayPokemon?*: `bool` , *pokemon*: `pokemon` , *contadorCaptura*: `nat` , *jugEsperandoCaptura*: `dicPrior` , *jugsEnPos*: `conjJugs`)

Invariante de Representación del Juego

1. La grilla tiene el alto y ancho del mapa
2. Los jugadores conectados estan en la grilla en la posicion en la que se encuentra
3. En los jugadores que se encuentran en la grilla solo estan los que realmente se encuentran ahi
4. Solo las posiciones con pokemon tienen pokemons en la grilla
5. CantPokemons es igual a la suma de todas los significados del diccionario
6. Los significados del diccionario de pokemons es igual a la cantidad de pokemons que hay capturados mas los salvajes
7. Los jugadores tienen menos de 5 sanciones
8. No hay dos coordenadas pertenecientes a posConPokemons a menos de 5 de distancia
9. Para cada una de las claves de jugsEsperandoCaptura, el jugador es un jugador válido y se encuentra en posición de captura del pokemon correspondiente
10. Para cada una de las claves de jugsEsperandoCaptura, su significado es la cantidad de pokemons

Rep : pokego $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$
long(e.grillaPos) = alto(e.mapa) $\wedge_{\text{L}}$
(($\forall n$: nat) n $\leq$ alto(e.mapa) $\Rightarrow$ long(e.grillaPos[n]) = ancho(e.mapa)) $\wedge_{\text{L}}$
jugadoresConectadosEnPosValidas(e) $\wedge_{\text{L}}$
jugadoresConectadosEstanEnLaPos(e) $\wedge$
jugadoresEnPosEstanConectados(e) $\wedge$
pokemonsEnGrillaSonSalvajes(e) $\wedge$
pokemonesCapturados(e.jugadores) + #(e.posConPokemons) = e.cantPokemons $\wedge$
diccionarioCorrecto(e) $\wedge$
jugadores5sanciones(e.jugadores) $\wedge$
pksAlejados(e.posConPokemons) $\wedge$
jugadoresEsperandoCaptura(e, e.posConPokemon)

jugadoresConectadosEnPosValidas : pokego $\longrightarrow$ bool
jugadoresConectadosEstanEnLaPos : pokego $\longrightarrow$ bool
jugadoresEnPosEstanConectados : pokego $\longrightarrow$ bool
jugadoresConectadosAux : secu(infoJugador) $\longrightarrow$ conj(jugador)
pokemonsEnGrillaSonSalvajes : pokego $\longrightarrow$ nat
pokemonesCapturados : secu(infoJugador) $\longrightarrow$ nat
diccionarioCorrecto : pokego $\longrightarrow$ bool
cantPkAfuera : pokemon $\times$ pokego $\longrightarrow$ nat
juntarPkEnGrilla : pokemon $\times$ conj(coor) $\times$ pokego $\longrightarrow$ nat
juntarPkCapturados : pokemon $\times$ secu(infoJugador) $\longrightarrow$ nat
jugadores5sanciones : secu(infoJugador) $\longrightarrow$ bool
pksAlejados : conj(coor) $\longrightarrow$ bool
compararCoors : coor $\times$ conj(coor) $\longrightarrow$ bool
jugadoresEsperandoCaptura : pokego $\times$ conj(coor) $\longrightarrow$ bool
jugsEsperandoEnPosValido : pokego $\times$ dicc(jugador $\times$ nat) $\longrightarrow$ bool
jugsEsperandoClavesValidas : pokego $\times$ conj(jugador) $\longrightarrow$ bool
jugsEsperandoDefsValidas : pokego $\times$ dicc(jugador $\times$ nat) $\times$ conj(coor) $\longrightarrow$ bool

jugadoresConectadosEnPosValidas(e) $\equiv$ ($\forall i$: nat) ((i $\leq$ long(e.jugadores) $\wedge$ e.jugadores[i]$\rightarrow$conectado?) $\Rightarrow$ (posExistente(e.jugadores[i]$\rightarrow$posicion, e.mapa)))

jugadoresConectadosEstanEnLaPos(e) $\equiv$ ($\forall i$: nat) ((i $\leq$ long(e.jugadores) $\wedge$ e.jugadores[i]$\rightarrow$conectado?) $\Rightarrow$ (i $\in$ e.grillaPos[longitud(e.jugadores[i]$\rightarrow$posicion)][latitud(e.jugadores[i]$\rightarrow$posicion)].ju

jugadoresEnPosEstanConectados(e) $\equiv$ ($\forall c$: coor) posExistente(c, e.mapa) $\Rightarrow$ e.grillaPos[longitud(c)][latitud(c)].jugsEnPos $\subseteq$ jugadoresConectadosAux(e.jugadores)

jugadoresConectadosAux(js) $\equiv$ **if** vacia?(js) **then**
$\emptyset$
else
if ult(js).sanciones $<$ 5 $\wedge_{\text{L}}$ ult(js).conectado? **then**
Ag((long(js) - 1), jugadoresConectadosAux(com(js)))
else
jugadoresConectadosAux(com(js))
fi
fi

pokemonsEnGrillaSonSalvajes(e) $\equiv$ ($\forall c$: coor) posExistente(c, e.mapa) $\Rightarrow$ (c $\in$ e.posConPokemons $\Longleftrightarrow$ e.grillaPos[longitud(c)][latitud(c)].hayPokemon?)

pokemonesCapturados(js) $\equiv$ **if** vacia?(js) **then**
0
else
prim(js).cantPokemons + pokemonesCapturados(fin(js))
fi

diccionarioCorrecto(e) $\equiv$ ($\forall pk$: pokemon)
((¬def?(pk, e.pokemons) $\wedge$ cantPkAfuera(pk, e) = 0) $\vee_{\text{L}}$
obtener(pk, e.pokemons) = cantPkAfuera(pk, e))

cantPkAfuera(pk, e) $\equiv$ juntarPkEnGrilla(pk, coordenadas(e.mapa), e) + juntarPkCapturados(pk, e.jugadores)

juntarPkEnGrilla(pk, coors, e) $\equiv$ **if** vacio?(coors) **then**
0
else
if e.grillaPos[longitud(dameUno(coors))][latitud(dameUno(coors))].hayPokemon? $\wedge_{\text{L}}$ e.grillaPos[longitud(dameUno(coors))][latitud(dameUno(coors))].pokemon = pk **then**
1
else
0
fi + juntarPkEnGrilla(pk, coors, e)
fi

juntarPkCapturados(pk, js) $\equiv$ **if** vacia?(js) **then**
0
else
if def?(pk, prim(js).pokemonsCapturados) **then**
obtener(pk, prim(js).pokemonsCapturados)
else
0
fi + pokemonesCapturados(fin(js))
fi

jugadores5sanciones(js) $\equiv$ ($\neg$ vacia?(js)) $\Rightarrow_{\text{L}}$ (prim(js).sanciones $\leq 5 \wedge$ jugadores5sanciones(fin(js)))

pksAlejados(coors) $\equiv$ ($\neg$ vacio?(dameUno(coors))) $\Rightarrow_{\text{L}}$ compararCoors(dameUno(coors), sinUno(coors))

compararCoors(c, coors) $\equiv$ ($\neg$ vacio?(coors)) $\Rightarrow_{\text{L}}$ (distEuclidea(c,dameUno(coors)) $\leq$ 25 $\wedge$ compararCoors(c, sinUno(coors)))

jugadoresEsperandoCaptura(e, ps) $\equiv$ vacio?(ps) $\vee_{\text{L}}$ (jugsEsperandoEnPosValido(e, e.grilla[latitud(dameUno(ps))][longitud(dameUno(ps))].jugEsperandoCaptura) $\wedge$ jugadoresEsperandoCaptura(sinUno(ps)))

jugsEsperandoEnPosValido(e, dic) $\equiv$ jugsEsperandoClavesValidas(e, claves(dic)) $\wedge$ jugsEsperandoDefsValidas(e, dic, claves(dic))

jugsEsperandoClavesValidas(e, js) $\equiv$ vacio?(js) $\vee_{\text{L}}$ jugsEsperandoClavesValidas(e, sinUno(js)) $\wedge$ dameUno(js) $<$ long(e.jugadores) $\wedge_{\text{L}}$ (e.jugadores[dameUno(js)].sanciones $< 5 \wedge_{\text{L}}$ distEuclidea(e.jugadores[dameUno(js)].posicion, crearCoor(i,j)) ≤ 4)

jugsEsperandoDefsValidas(e, dic, js) $\equiv$ vacio?(js) $\vee_{\text{L}}$ (jugsEsperandoDefsValidas(e, dic, sinUno(js)) $\wedge$ obtener(dic, dameUno(js)) = (*e.jugadores[dameUno(js)]).cantPokemons)

Función de Abstracción

Abs : pokego e $\longrightarrow$ juego {Rep(e)}

Abs(e) $=_{\text{obs}}$ j: juego | mapa(j) = e.mapa $\wedge$ long(e.jugadores) = ProxID(j) $\wedge_{\text{L}}$ ($\forall p$: jugador)
(long(e.jugadores) $>$ p $\Rightarrow_{\text{L}}$ (sanciones(p, j) = e.jugadores[p]$\rightarrow$sanciones $\wedge$
sanciones(p, j) $< 5 \Rightarrow_{\text{L}}$ (j $\in$ jugadores(j) $\wedge_{\text{L}}$ estaConectado(p, j) = e.jugadores[p]$\rightarrow$conectado? $\wedge$ (estaConectado(p, j) $\Rightarrow_{\text{L}}$ posicion(p, j) = e.jugadores[p]$\rightarrow$posicion) $\wedge$
pokemons(j) = listaPoke(claves(e.pokemons), e.pokemons)) $\wedge$
(e.jugadores[p]$\rightarrow$sanciones = 5) = (p $\in$ expulsados(j)))) $\wedge$ ($\forall c$: coor)
(c $\in$ posConPokemons(j) = c $\in$ e.posConPokemons $\wedge_{\text{L}}$ c $\in$ posConPokemons(j) $\Rightarrow_{\text{L}}$ (pokemonEnPos(c, j) = j.grillaPos[Latitud(c)][Longitud(c)].pokemon $\wedge$ cantMovimientosParaCaptura(c, j) = j.grillaPos[Latitud(c)][Longitud(c)].contadorCaptura))

listaPoke : conj(pokemon) cs $\times$ dicc(String,Nat) dic $\longrightarrow$ multiconj(pokemon) {cs $\subseteq$ claves(dic)}

agregarPoke : pokemon $\times$ nat $\longrightarrow$ multiconj(pokemon)

listasPoke(cs, dic) ≡ **if** vacia?(cs) **then**
∅
else
agregarPoke(dameUno(cs),obtener(dameUno(cs),dic)) ⋃ listasPoke(sinUno(cs), dic)
fi
agregarPoke(p, n) ≡ **if** n = 0 **then** ∅ **else** Ag(p, agregarPoke(p, n-1)) **fi**

Representación del iterador de jugadores

`itJugadores` **se representa con** `itJug`

donde `itJug` es `tupla`(*listaJugadores*: `puntero(vector(puntero(infoJugador)))` , *contador*: `nat` , *eliminados?*: `bool`)

Invariante de Representación del iterador

1. La lista de jugadores no es nula

Rep : itJug ⟶ bool
Rep(*it*) ≡ true ⟺ it.listaJugadores ≠ NULL

Función de Abstracción del iterador

Abs : itJug *it* ⟶ itUni(nat) {Rep(*it*)}
Abs(*it*) =$_{obs}$ b: itUni(nat) | Siguientes(b) = seleccionar(ultimos(*it.listaJugadores, it.contador), it.eliminados?)

selecionar : secu(infoJugador) *js* × bool *elim*? ⟶ secu(nat)
ultimos : secu(infoJugador) *js* × nat *n* ⟶ secu(infoJugador)

selecionar(js, elim?) ≡ **if** vacia?(js) **then**
<>
else
if (ult(js).sanciones < 5) = elim? **then**
seleccionar(com(js), elim?) ∘ (long(js) - 1))
else
seleccionar(com(js), elim?)
fi
fi
ultimos(js, n) ≡ **if** 0?(n) ∨ vacia?(js) **then** <> **else** ultimos(com(js), n-1) ∘ ult(js) **fi**

Algoritmos

Algoritmos de Juego

iCrearJuego (**in** *m* : `mapa`) → res: pokego	
res.mapa ← m;	$O(1)$
res.pokemons ← CrearDiccionario();	$O(1)$
res.posConPokemons ← Vacio();	$O(1)$
res.jugadores ← Vacio();	$O(1)$
res.cantPokemons ← 0;	$O(1)$
res.grillaPos ← CrearArreglo(Alto(m));	$O(Alto(m))$
for $i \leftarrow 0$ **to** *Alto(m)* **do**	$O(Alto(m) \times Ancho(m))$
res.grillaPos[i] ← CrearArreglo(Ancho(m));	$O(Ancho(m))$
for $j \leftarrow 0$ **to** *Ancho(m)* **do**	$O(Ancho(m))$
res.grillaPos[i][j] ← CrearInfoPos();	$O(1)$
end for	
end for	

Complejidad: $O(Alto(m) \times Ancho(m))$
Justificación: se debe reservar memoria para la grilla que contiene información del juego de cada posición.

iAgregarPokémon (**in/out** j : `pokego`, **in** pk : `pokemon`, **in** c : `coor`)	
j.cantPokemons ← j.cantPokemons + 1;	$O(1)$
if *Definido(j.pokemons, pk)* **then**	
nat: nuevaCant ← Obtener(j.pokemons, pk) + 1;	$O(\|pk\|)$
Definir(j.pokemons, pk, nuevaCant);	$O(\|pk\|)$
else	
Definir(j.pokemons, pk, 1);	$O(\|pk\|)$
end if	
j.grillaPos[Latitud(c)][Longitud(c)].hayPokemon ← true;	$O(1)$
j.grillaPos[Latitud(c)][Longitud(c)].pokemon ← pk;	$O(1)$
j.grillaPos[Latitud(c)][Longitud(c)].contadorCaptura ← 0;	$O(1)$
`/* Se desestima la complejidad de borrar el diccionario de prioridad anterior`	`*/`
j.grillaPos[Latitud(c)][Longitud(c)].jugEsperandoCaptura ← Vacio();	$O(1)$
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, 2);	$O(1)$
itBi(coor) : itCoor ← CrearIt(coorEnRango);	$O(1)$
while *HaySiguiente(itCoor)* **do**	$O(EC \times log(EC))$
coor: d ← Siguiente(itCoor);	$O(1)$
itConjJugs : it ← CrearIterador(j.grillaPos[Latitud(d)][Longitud(d)].jugsEnPos);	$O(1)$
while *HayMas(it)* **do**	$O(EC \times log(EC))$
jugador: jug ← Actual(it);	$O(1)$
Definir(j.grillaPos[Latitud(c)][Longitud(c)].jugEsperandoCaptura, jug, (*j.jugadores[jug]).cantPokemons);	$O(log(EC))$
Avanzar(it);	$O(1)$
end while	
Avanzar(itCoor);	$O(1)$
end while	

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

Justificación: todos los jugadores que se encuentran en el area (EC) deben agregarse a la lista de espera, que los ordena por prioridad (inserción en log(EC)). Al mismo tiempo debe definirse este nuevo pokémon en el diccionario global ($|P|$).

iAgregarJugador (**in/out** j : `pokego`) → res: nat	
res ← Longitud(j.jugadores);	$O(1)$
AgregarAtras(j.jugadores, CrearInfoJugador());	$O(J)$

Complejidad: $O(J)$

Justificación: en el peor caso se debe redimensionar el vector. Hacerlo requiere copiar el arreglo interno, pero al tratarse de punteros la copia es gratuita ($\Theta(1)$ por posición, o $\Theta(J)$ en su totalidad).

iConectarse (**in/out** j : `pokego`, **in** e : `jugador`, **in** c : `coor`)	
(*j.jugadores[e]).conectado? ← true;	$O(1)$
(*j.jugadores[e]).posicion ← c;	$O(1)$
Agregar(j.grillaPos[Latitud(c)][Longitud(c)].jugsEnPos, e);	$O(log(EC))$
AgregarACola(j, e);	$O(log(EC))$
ReseteарContadores(j, e);	$O(1)$

Complejidad: $O(log(EC))$

Justificación: al conectarse, el jugador debe agregarse al conjunto de jugadores en c, y unirse a la cola de espera de captura de haber un pokemón cerca. De ser ese el caso, la cantidad de jugadores en la cola de espera será siempre mayor a la cantidad de jugadores en c (todos los jugadores en c están en la cola de espera).

iDesconectarse (**in/out** j : `pokego`, **in** e : `jugador`)	
(*j.jugadores[e]).conectado? ← false;	$O(1)$
Borrar(j.grillaPos[Latitud(c)][Longitud(c)].jugsEnPos, e);	$O(log(EC))$
RemoverDeCola(j, e);	$O(log(EC))$

Complejidad: $O(log(EC))$
Justificación: inversamente, al desconectarse, el jugador debe salir del conjunto de jugadores en c y de la cola de espera de captura (de haber un pokemón cerca).

iMoverse (**in/out** j : `pokego`, **in** e : `jugador`, **in** c : `coor`)

```
coor : posAnterior ← Posicion(j,e);                                                    O(1)
/* Removemos al jugador del conjunto de su posición anterior y de la cola de espera   */
Borrar(j.grillaPos[Latitud(c)][Longitud(c)].jugsEnPos, e);                             O(log(EC))
RemoverDeCola(j, e);                                                                   O(log(EC))
if not HayCamino(j.mapa, posAnterior, c) or DistEuclidea(posAnterior, c) > 100 then
    (*j.jugadores[e]).sanciones ← (*j.jugadores[e]).sanciones + 1;                     O(1)
end if
if (*j.jugadores[e]).sanciones = 5 then
    /* Si el jugador debe ser eliminado, borramos sus pokémons                        */
    itDiccString(nat) : pokesABorrar ← CrearIt((*j.jugadores[e]).pokemonsCapturados);  O(1)
    while HaySiguiente(pokesABorrar) do                                                O(PC × |P|)
        tupla(calve: String, significado: Nat) : sig ← Siguiente(pokesABorrar);        O(|P|)
        nat: nuevaCant ← Obtener(j.pokemons, sig.clave) - sig.significado;             O(|P|)
        Definir(j.pokemons, pk, nuevaCant);                                            O(|P|)
        j.cantPokemons ← j.cantPokemons - sig.significado;                             O(1)
    end while
else
    /* Si el jugador sigue siendo válido, lo agregamos al conjunto de su nueva posición y a la
       cola de espera                                                                  */
    (*j.jugadores[e]).posicion ← c;                                                    O(1)
    AgregarACola(j, e, c);                                                             O(log(EC))
    Agregar(j.grillaPos[Latitud(c)][Longitud(c)].jugsEnPos, e);                        O(log(EC))
end if
it ← CrearIt(j.posConPokemons);                                                        O(1)
/* Iteramos las posiciones con pokémons                                               */
while HaySiguiente?(it) do                                                             O(PS × |P|)
    coor: coorConPk ← Siguiente(it);                                                   O(1)
    if DistEuclidea(c, coorConPk) > 4 then
        /* Si el movimiento es lejano, sumar al contador e manejar captura si corresponde */
        j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)].contadorCaptura ←
          j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)].contadorCaptura +1;     O(1)
        infoPos: posPk ← j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)];         O(1)
        if posPk.contadorCaptura = 10 then
            pokemon: pk ← posPk.pokemon;                                               O(1)
            jugador: captor ← Menor(posPk.jugEsperandoCaptura);                        O(1)
            (*j.jugadores[captor]).cantPokemons ← (*j.jugadores[captor]).cantPokemons + 1;  O(1)
            if Definido((*j.jugadores[captor]).pokemonsCapturados, pk) then            O(log(|P|))
                nat: nuevaCant ← Obtener((*j.jugadores[captor]).pokemonsCapturados, pk) + 1;  O(log(|P|))
                Definir((*j.jugadores[captor]).pokemonsCapturados, pk, nuevaCant);     O(log(|P|))
            else
                Definir((*j.jugadores[captor]).pokemonsCapturados, pk, 1);             O(log(|P|))
            end if
            /* Si se elimina esa posición, no hay que avanzar el iterador             */
            EliminarSiguiente(it);                                                     O(1)
        else
            Avanzar(it);                                                               O(1)
        end if
    else
        if DistEuclidea(posAnterior, coorConPk) > 4 then
            /* Si el jugador entró a una nueva área de captura, reiniciar el contador */
            j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)].contadorCaptura ← 0;  O(1)
        end if
        Avanzar(it);                                                                   O(1)
    end if
end while
```

Complejidad: $O((PS + PC) \times |P| + log(EC))$
Justificación: las validaciones del movimiento son gratuitas gracias a la implementación por grupos de mapa. Por otro lado, si el jugador queda eliminado, debe eliminarse del diccionario del sistema ($|P|$) cada pokémon que había capturado (PC).

En cada movimiento, el jugador que se mueve debe cambiar de conjunto y tal vez salir de o entrar en un grupo de espera de captura de un pokémon (log(EC)).

Por otro lado, para posiciones lejanas se deben procesarse las potenciales capturas (PS), agregando el pokémon al diccionario del jugador que lo captura ($|P|$).

iMapa (**in** j : `pokego`) → res: mapa	
res ← j.mapa;	$O(1)$

Complejidad: $O(1)$
Justificación: se devuelve el mapa por referencia.

iPokémonEnPos (**in** j : `pokego`, **in** c : `coor`) → res: pokemon	
res ← j.grillaPos[Latitud(c)][Longitud(c)].pokemon;	$O(1)$

iEstaConectado (**in** j : `pokego`, **in** e : `jugador`) → res: bool	
res ← (*j.jugadores[e]).conectado?;	$O(1)$

iSanciones (**in** j : `pokego`, **in** e : `jugador`) → res: nat	
res ← (*j.jugadores[e]).sanciones;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se accede a un vector y se desreferncia un puntero y/o se accede a un miembro.

iPosicion (**in** j : `pokego`, **in** e : `jugador`) → res: coor	
res ← CrearCoor(Latitud((*j.jugadores[e]).posicion), Longitud((*j.jugadores[e]).posicion));	$O(1)$

Complejidad: $O(1)$
Justificación: solo se accede a un vector, se desreferencia un puntero y se crea una copia de la coordenada.

iPokémons (**in** j : `pokego`, **in** e : `jugador`) → res: itBi(tupla(pokemon, nat))	
res ← CrearIterador((*j.jugadores[e]).pokemonsCapturados);	$O(1)$

iJugadores (**in** j : `pokego`) → res: itJugadores	
res ← CrearIt(j, false);	$O(1)$

iExpulsados (**in** j : `pokego`) → res: itJugadores	
res ← CrearIt(j, true);	$O(1)$

Complejidad: $O(1)$
Justificación: solo se inicializa el iterador.

iPosConPokémons (**in** j : `pokego`) → res: conj(coor)	
res ← j.posConPokemons;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se devuelve el conjunto por referencia.

iPuedoAgregarPokémon (**in** j : `pokego`, **in** c : `coor`) → res: bool	
res ← PosExistente(j.mapa, c) **and** HayPokémonEnDistancia(j, c, 5);	$O(1)$

iHayPokémonCercano (**in** j : `pokego`, **in** c : `coor`) → res: bool	
res ← HayPokémonEnDistancia(j, c, 2);	$O(1)$

iPosPokémonCercano (**in** j : `pokego`, **in** c : `coor`) → res: coor	
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, 2);	$O(1)$
itBi(coor) : itCoor ← CrearIt(coorEnRango);	$O(1)$
while *HaySiguiente(itCoor)* **do**	$O(n^2)$
coor : siguiente ← Siguiente(itCoor);	$O(1)$
if *HayPokémonEnPos(j, siguiente)* $\leq n$ **then**	
res ← siguiente;	$O(1)$
end if	
Avanzar(itCoor);	$O(1)$
end while	

Complejidad: $O(1)$
Justificación: solo debe iterar todas las posiciones de un cuadrado de lado predeterminado. Al ser predeterminado se considera constante.

iEntrenadoresPosibles (**in** j : `pokego`, **in** c : `coor`) → res: conj(jugador)
res ← Claves(j.grillaPos[Latitud(c)][Longitud(c)].jugEsperandoCaptura)

Complejidad: $O(EC)$
Justificación: el diccionario debe recuperar las claves iterando.

iCantPokémonsTotales (**in** j : `pokego`) → res: nat	
res ← j.cantPokemons;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se retorna un valor almacenado.

iIndiceRareza (**in** j : `pokego`, **in** pk : `pokemon`) → res: nat	
nat : cantPk ← CantMismaEspecie(j, pk);	$O(\|P\|)$
res ← 100 - (cantPk × 100 ÷ j.cantPokemons);	$O(1)$

iCantMismaEspecie (**in** j : `pokego`, **in** pk : `pokemon`) → res: nat	
res ← Obtener(j.pokemones, pk);	$O(\|P\|)$

Complejidad: $O(|P|)$
Justificación: se debe acceder al diccionario para averiguar cuántos pokemons de esa especie.

Algoritmos auxiliares

CrearInfoPos () → res: infoPos	
Pre: `true`	
Post: la posición nueva no contiene pokémon ni jugadores	
res.hayPokemon? ← false;	$O(1)$
res.contadorCaptura ← 0;	$O(1)$
res.jugEsperandoCaptura ← Vacio();	$O(1)$
res.jugsEnPos ← Vacio();	$O(1)$

CrearInfoJugador () → res: puntero(infoJugador)

Pre: `true`
Post: el jugador nuevo no tiene sanciones ni pokémons y no está conectado

infoJugador : nuevo;	$O(1)$
nuevo.sanciones ← 0;	$O(1)$
nuevo.conectado? ← false;	$O(1)$
nuevo.cantPokemons ← 0;	$O(1)$
nuevo.pokemonsCapturados ← CrearDiccionario();	$O(1)$
res ← &nuevo;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se inicializan las variables.

AgregarACola (**in/out** j : `pokego`, **in** e : `jugador`)

Pre: $e \in$ jugadores(j) $\wedge_{\text{L}}$ conectado?(e, j)
Post: agrega al jugador a la lista de espera del pokémon mas cercano a él (si hay uno en rango de captura)

coor: c ← (*j.jugadores[e]).posicion;	$O(1)$
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, 2);	$O(1)$
itBi(coor) : itCoor ← CrearIt(coorEnRango);	$O(1)$
while *HaySiguiente(itCoor)* **do**	$O(log(EC))$
coor: siguiente ← Siguiente(itCoor);	$O(1)$
if *HayPokémonEnPos(j, siguiente).hayPokemon* **then**	
Definir(j.grillaPos[Latitud(siguiente)][Longitud(siguiente)].jugEsperandoCaptura, (*j.jugadores[e]).cantPokemons);	$O(log(EC))$
end if	
Avanzar(itCoor);	$O(1)$
end while	

Complejidad: $O(log(EC))$
Justificación: el algoritmo recorre una cantidad menor a una constante de coordenadas alrededor del jugador, de encontrar un Pokemon, agrega al jugador al diccionarioDePrioridad que tiene insercion en $O(log(EC))$, ademas por la especificacion sabemos que un jugador puede estar capturando un solo pokemon, por lo que definir el jugador se hace una vez.

RemoverDeCola (**in/out** j : `pokego`, **in** e : `jugador`)

Pre: $e \in$ jugadores(j) $\wedge_{\text{L}}$ conectado?(e, j)
Post: remueve al jugador de la lista de espera del pokémon mas cercano a él (si hay uno en rango de captura)

coor: c ← (*j.jugadores[e]).posicion;	$O(1)$
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, 2);	$O(1)$
itBi(coor) : itCoor ← CrearIt(coorEnRango);	$O(1)$
while *HaySiguiente(itCoor)* **do**	$O(log(EC))$
coor: siguiente ← Siguiente(itCoor);	$O(1)$
if *HayPokémonEnPos(j, siguiente).hayPokemon* **then**	
Borrar(j.grillaPos[Latitud(siguiente)][Longitud(siguiente)].jugEsperandoCaptura, e);	$O(log(EC))$
end if	
Avanzar(itCoor);	$O(1)$
end while	

Complejidad: $O(log(EC))$
Justificación: debe iterar todas las posiciones de un cuadrado de lado predeterminado y agregar al jugador a la cola de espera de captura más cercana. Si la misma existe.

ResetearContadores (**in/out** j : `pokego`, **in** e : `jugador`)

Pre: $e \in$ jugadores(j) $\wedge_{\text{L}}$ conectado?(e, j)
Post: los contadores de captura las posiciones aledañas al jugador vuelven a 0

```
coor: c ← (*j.jugadores[e]).posicion;                                              O(1)
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, 2);                             O(1)
itBi(coor) : itCoor ← CrearIt(coorEnRango);                                        O(1)
while HaySiguiente(itCoor) do                                                      O(1)
    coor: siguiente ← Siguiente(itCoor);                                           O(1)
    if HayPokémonEnPos(j, siguiente).hayPokemon then
        j.grillaPos[Latitud(siguiente)][Longitud(siguiente)].contadorCaptura ← 0;   O(1)
    end if
    Avanzar(itCoor);                                                               O(1)
end while
```

Complejidad: $O(1)$
Justificación: debe iterar todas las posiciones de un cuadrado de lado predeterminado.

PosicionesEnRango (**in/out** j : `pokego`, **in** c : `coor, in` n : `nat`) → res: conj(coor)

Pre: `true`
Post: res es igual al conjunto de posiciones válidas cerca de c, con distancia euclidiana máxima de n^2.[1]

```
res ← Vacia();                                                                     O(1)
for i ← 0 to n do                                                                  O(n²)
    for j ← 0 to n do                                                              O(n)
        coor : ne ← CrearCoor(Latitud(c) + i, Longitud(c) + j);                    O(1)
        if DistEuclidea(c,ne) ≤ n² and PosExistente(ne, j.mapa) then               O(1)
            AgregarRapido(res, ne);                                                O(1)
        end if
        if Longitud(c) > j then
            coor : no ← CrearCoor(Latitud(c) + i, Longitud(c) - j);                O(1)
            if DistEuclidea(c,no) ≤ n² and PosExistente(no, j.mapa) then           O(1)
                AgregarRapido(res, no);                                            O(1)
            end if
        end if
        if Latitud(c) > i then
            coor : se ← CrearCoor(Latitud(c) - i, Longitud(c) + j);                O(1)
            if DistEuclidea(c,se) ≤ n² and PosExistente(se, j.mapa) then           O(1)
                AgregarRapido(res, se);                                            O(1)
            end if
        end if
        if Latitud(c) > i and Longitud(c) > j then
            coor : so ← CrearCoor(Latitud(c) - i, Longitud(c) - j);                O(1)
            if DistEuclidea(c,so) ≤ n² and PosExistente(so, j.mapa) then           O(1)
                AgregarRapido(res, so);                                            O(1)
            end if
        end if
    end for
end for
```

[1] Se usa la definición de Coordenada de distancia euclidiana, que no implica la raíz cuadrada.

HayPokémonEnDistancia (**in** j : `pokego`, **in** c : `coor`, **in** n : `nat`) → res: bool

Pre: `true`
Post: res es `true` si hay un pokémon en una posición en el conjunto de posiciones válidas cerca de c, con distancia euclidiana máxima de n^2.

```
res ← false;                                                          O(1)
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, n);                O(1)
itBi(coor) : itCoor ← CrearIt(coorEnRango);                           O(1)
while HaySiguiente(itCoor) do                                         O(n^2)
    coor : siguiente ← Siguiente(itCoor);                             O(1)
    if HayPokémonEnPos(j, siguiente) ≤ n then
        res ← true;                                                   O(1)
    end if
    Avanzar(itCoor);                                                  O(1)
end while
```

Complejidad: $O(n^2)$
Justificación: itera n veces por latitud, multiplicado por n veces por la longitud.

HayPokémonEnPos (**in/out** j : `pokego`, **in** c : `coor`) → res: bool

Pre: posExistente(c, mapa(j))
Post: res $=_{\text{obs}}$ c ∈ posConPokémons(j)

```
res ← j.grillaPos[Latitud(c)][Longitud(c)].hayPokemon;                O(1)
```

Complejidad: $O(1)$
Justificación: solo se accede al arreglo.

Algoritmos del iterador

CrearIt (**in** j : `pokego`, **in** *elim?* : `bool`) → res: itJugadores

```
res.listaJugadores ← &(j.jugadores);                                  O(1)
res.contador ← 0;                                                     O(1)
res.eliminados ← elim?;                                               O(1)
```

Complejidad: $O(1)$
Justificación: solo se inicializa el iterador.

HayMas (**in** *it* : `itJugadores`) → res: bool

```
res ← false;                                                          O(1)
for i ← it.contador to long(*it.listaJugadores) do                    O(J)
    if ((*it.listaJugadores)[i].sanciones < 5) = it.eliminados? then
        res ← true;                                                   O(1)
        break;
    end if
end for
```

Actual (**in** *it* : `itJugadores`) → res: jugador

```
for i ← it.contador to long(*it.listaJugadores) do                    O(J)
    if ((*it.listaJugadores)[i].sanciones < 5) = it.eliminados? then
        res ← i;                                                      O(1)
        break;
    end if
end for
```

```
Avanzar (in/out it : itJugadores)
for i ← it.contador to long(*it.listaJugadores) do                    O(J)
    if ((*it.listaJugadores)[i].sanciones < 5) = it.eliminados? then
        it.contador ← i + 1;                                          O(1)
        break;
    end if
end for
```

Complejidad: $O(J)$

Justificación: como la lista de jugadores y la de eliminados son la misma, debe iterarla en su totalidad hasta encontrar otro elemento válido (si es que existe).

Cabe aclarar que la complejidad de iterar la lista en su totalidad es $\Theta(J)$, ya que la parte de la lista que se debe iterar en cada iteración es distinta, hasta que se la recorre de manera completa.

2. Módulo Mapa

Las posiciones se almacenan en una grilla dinámica. El mapa asegura la posibililidad de encontrar conexiones rápidamente al costo de agregado lento. En cada posición se almacena el grupo del conexiones al que pertenece: se considera que dos posiciones están conectadas si pertenecen al mismo grupo. Al ser agregada una posición, las nuevas conexiones se calculan automáticamente.

Usaremos lat_{max} y $long_{max}$ para denotar las máximas dimensiones entre las posiciones existentes del mapa.

Interfaz

se explica con: MAPA.

géneros: `map`.

servicios usados: VECTOR(α)

Operaciones básicas de Mapa

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

AGREGARCOOR(**in** c: `coor`, **in/out** m: `map`)
Pre $\equiv$ $\{m =_{\text{obs}} m_0\}$
Post $\equiv$ $\{m =_{\text{obs}} agregarCoor(c, m_0)\}$
Complejidad: $O(lat_{max} \times long_{max})$
Descripción: agrega la coordenada al mapa.

COORDENADAS(**in** m: `map`) $\rightarrow$ res : `conj(coor)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} coordenadas(m)\}$
Complejidad: $O(lat_{max} \times long_{max})$
Descripción: devuelve las coordenadas de un mapa

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

HAYCAMINO(**in** $c1$: `coor`, **in** $c2$: `coor`, **in** m: `map`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{c1 \in$ coordenadas(m) $\wedge$ $c2 \in$ coordenadas(m)$\}$
Post $\equiv$ $\{res =_{\text{obs}} hayCamino(c1, c2, m)\}$
Complejidad: $O(1)$
Descripción: verifica si hay un camino entre dos coordenadas.

ALTO(**in** m: `map`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} alto(m, coordenadas(m))\}$
Complejidad: $O(1)$
Descripción: devuelve el alto del mapa.

ANCHO(**in** m: `map`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} ancho(m, coordenadas(m))\}$
Complejidad: $O(1)$

Descripción: devuelve el ancho del mapa.

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Mapa Extendido

extiende MAPA

otras operaciones (exportadas)

alto : mapa m × conj(coor) c ⟶ nat {c ⊆ coordenadas(m)}
ancho : mapa m × conj(coor) c ⟶ nat {c ⊆ coordenadas(m)}

axiomas

```
alto(m, c) ≡ if vacio?(c) then
                0
             else
                if alto(m, sinUno(c)) < Latitud(dameUno(c)) then
                   Latitud(dameUno(c))
                else
                   alto(m, sinUno(c))
                fi
             fi
ancho(m, c) ≡ if vacio?(c) then
                 0
              else
                 if ancho(m, sinUno(c)) < Longitud(dameUno(c)) then
                    Longitud(dameUno(c))
                 else
                    ancho(m, sinUno(c))
                 fi
              fi
```

Fin TAD

Representación

Representación de Mapa

`Mapa` **se representa con** `eMap`

donde `eMap` es `tupla`(*alto*: `nat` , *ancho*: `nat` , *posiciones*: `vector(vector(dataPos))` , *proxGrupo*: `nat`)

donde `dataPos` es `tupla`(*existe*: `bool` , *grupo*: `nat`)

Invariante de Representación

1. La longitud del vector posiciones es igual al alto del mapa.
2. La longitud de cada uno de los vectores dentro del vector posiciones es igual o menor al ancho del mapa.
3. Existe al menos un vector dentro del vector posiciones cuya longitud es igual al ancho del mapa.
4. Las posiciones existentes adyacentes entre sí deben tener el mismo grupo.
5. Si dos posiciones no tienen un camino de posiciones adyacentes de mismo grupo que las una, deben tener grupos distintos.[2]

```
Rep : eMap ⟶ bool
Rep(e) ≡ true ⟺ e.alto = Longitud(e.posiciones) ∧ₗ
          (∃i: nat) (i < e.alto ⇒ Longitud(e.posiciones[i]) = e.ancho) ∧
          (∀i: nat) (i < e.alto ⇒ Longitud(e.posiciones[i]) ≤ e.ancho) ∧
          (∃i, j: nat) (i = e.alto - 1 ∧ posValida?(i,j,e)) ∧
          (∃i, j: nat) (j = e.ancho - 1 ∧ posValida?(i,j,e)) ∧
          (∀i, j: nat) (posValida?(i,j,e) ⇒ₗ gruposValidos?(i,j,e))
```

[2] No pudimos expresar esto en rep, lo dejamos en castellano

posValida? : nat i × nat j × eMap e $\longrightarrow$ bool
posValida?(i, j, e) $\equiv$ i < e.alto $\wedge_{\text{L}}$ j < long(e.posiciones[i]) $\wedge_{\text{L}}$ e.posiciones[i][j].existe

gruposValidos? : nat i × nat j × eMap e $\longrightarrow$ bool {posValida?(i,j,e)}
gruposValidos?(i, j, e) $\equiv$ mismoGrupoArriba(i,j,e) $\wedge$ mismoGrupoAbajo(i,j,e)
$\wedge$ mismoGrupoALaDer(i,j,e) $\wedge$ mismoGrupoALaIzq(i,j,e)
mismoGrupoArriba : nat i × nat j × eMap e $\longrightarrow$ bool {posValida?(i,j,e)}
mismoGrupoArriba(i, j, e) $\equiv$ $\neg$posValida?(i+1,j,e) $\vee_{\text{L}}$ e.posiciones[i][j+1].grupo = e.posiciones[i][j].grupo

mismoGrupoAbajo : nat i × nat j × eMap e $\longrightarrow$ bool {posValida?(i,j,e)}
mismoGrupoArriba(i, j, e) $\equiv$ j = 0 $\vee_{\text{L}}\neg$posValida?(i-1,j,e) $\vee_{\text{L}}$ e.posiciones[i][j-1].grupo = e.posiciones[i][j].grupo

mismoGrupoALaDer : nat i × nat j × eMap e $\longrightarrow$ bool {posValida?(i,j,e)}
mismoGrupoArriba(i, j, e) $\equiv$ $\neg$posValida?(i,j+1,e) $\vee_{\text{L}}$ e.posiciones[i+1][j].grupo = e.posiciones[i][j].grupo

mismoGrupoAbajo : nat i × nat j × eMap e $\longrightarrow$ bool {posValida?(i,j,e)}
mismoGrupoALaIzq(i, j, e) $\equiv$ i = 0 $\vee_{\text{L}}\neg$posValida?(i,j-1,e) $\vee_{\text{L}}$ e.posiciones[i][j-1].grupo = e.posiciones[i][j].grupo

Función de Abstracción

Abs : eMap e $\longrightarrow$ map {Rep(e)}
Abs(e) $=_{\text{obs}}$ m: map | ($\forall c$: coor) (c $\in$ coordenadas(m) $\iff$ posValida?(latitud(c), longitud(c), e))

Algoritmos

Algorítmos de Mapa

iCrearMapa () → res: map	
res.alto ← 0;	$O(1)$
res.ancho ← 0;	$O(1)$
res.posiciones ← Vacia();	$O(1)$
res.proxGrupo ← 1;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se inicializan las variables y se crea el vector vacío.

iAgregarCoor (**in** c: coor, **in/out** m: map)	
/* se agregan los vectores vacíos necesarios */	
while *Longitud(m.posiciones)* $\leq$ *Latitud(c)* **do**	$O(lat_{max})$
AgregarAtras(m.posiciones, Vacia());	$O(f(lat_{max}))$
end while	
if *m.alto* $\leq$ *Latitud(c)* **then**	
m.alto $\leftarrow$ Latitud(c);	$O(1)$
end if	
/* se agregan las posiciones inexistentes necesarias */	
while *Longitud(m.posiciones[Latitud(c)])* $\leq$ *Longitud(c)* **do**	$O(long_{max})$
AgregarAtras(m.posiciones, CrearDataPos());	$O(f(long_{max}))$
end while	
if *m.ancho* $\leq$ *Longitud(c)* **then**	
m.ancho $\leftarrow$ Longitud(c);	$O(1)$
end if	
/* se modifica la posición a existente y se unen los grupos pertinentes */	
m.posiciones[Latitud(c)][Longitud(c)].existe $\leftarrow$ true;	$O(1)$
m.posiciones[Latitud(c)][Longitud(c)].grupo $\leftarrow$ m.proxGrupo;	$O(1)$
Unir(c, CoordenadaArriba(c), m);	$O(lat_{max} \times long_{max})$
if *Latitud(c)* $>$ *0* **then**	
Unir(c, CoordenadaAbajo(c), m);	$O(lat_{max} \times long_{max})$
end if	
Unir(c, CoordenadaALaDerecha(c), m);	$O(lat_{max} \times long_{max})$
if *Longitud(c)* $>$ *0* **then**	
Unir(c, CoordenadaALaIzquierda(c), m);	$O(lat_{max} \times long_{max})$
end if	
/* si no se une con nada, se aumenta el próximo grupo */	
if *m.posiciones[Latitud(c)][Longitud(c)].grupo* $=$ *m.proxGrupo* **then**	
m.proxGrupo $\leftarrow$ m.proxGrupo + 1;	$O(1)$
end if	

Complejidad: $O(lat_{max} \times long_{max})$

Justificación: más allá de si la nueva posición tiene la mayor longitud o latitud, se debe iterar el conjunto entero para unir las posiciones que correspondan. Los costos redimensionar los vectores son menores (si se considera el costo de copiar coordenadas $\Theta(1)$) ya que luego se deben iterar todos los vectores del mapa.

iCoordenadas (**in** m: map) $\rightarrow$ res: conj(coor)	
res $\leftarrow$ Vacío();	$O(1)$
for $i \leftarrow 0$ **to** *m.alto* **do**	$O(lat_{max} \times long_{max})$
for $j \leftarrow 0$ **to** *Longitud(m.posiciones[i])* **do**	$O(long_{max})$
if *m.posiciones[i][j].existe* **then**	$O(1)$
coor: pos $\leftarrow$ CrearCoor(i,j);	$O(1)$
AgregarRapido(res, pos);	$O(1)$
end if	
end for	
end for	

Complejidad: $O(lat_{max} \times long_{max})$

Justificación: se deben leer todas las posiciones de los vectores para agregar las existentes al conjunto. En el peor caso, el mapa es rectangular y todos los vectores tienen la misma longitud. Cada agregado individual es constante (se sabe que esa coordenada no estaba anteriormente).

iPosExistente (**in** c: coor, **in** m: map) $\rightarrow$ res: bool	
res $\leftarrow$ Latitud(c) $<$ m.alto **and** Longitud(c) $<$ Long(m.posiciones[Latitud(c)]) **and** m.posiciones[Latitud(c1)][Longitud(c1)].existe;	$O(1)$

Complejidad: $O(1)$

Justificación: solo se comparan números y se accede a posiciones de los vectores.

iHayCamino (**in** $c1$: `coor`, **in** $c2$: `coor`, **in** m : `map`) → res: bool	
res ← m.posiciones[Latitud(c1)][Longitud(c1)].grupo = m.posiciones[Latitud(c2)][Longitud(c2)].grupo;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se accede a posiciones de los vectores. Las uniones se calculan al agregar las posiciones.

iAncho (**in** m : `map`) → res: nat	
res ← m.ancho;	$O(1)$

iAlto (**in** m : `map`) → res: nat	
res ← m.alto;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se accede a variables del mapa.

Algoritmos auxiliares

CrearDataPos () → res: dataPos	
Pre: `true`	
Post: *res* es una posición no existente	
res.existe ← false;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se crea una posición no existente.

Unir (**in** $c1$: `coor`, **in** $c2$: `coor`, **in** m : `map`) → res: bool	
Pre: `true`	
Post: ($c1 \in$ coordenadas(m) $\wedge$ $c2 \in$ coordenadas(m)) $\Rightarrow_{\text{L}}$ m.posiciones[latitud(c1)][longitud(c1)].grupo = m.posiciones[latitud(c2)][longitud(c2)].grupo	
if *PosExistente(c1)* ***and*** *PosExistente(c2)* ***and not*** *HayCamino(c1, c2, m)* **then**	$O(lat_{max} \times long_{max})$
nat : grupoViejo ← m.posiciones[Latitud(c1)][Longitud(c1)].grupo;	$O(1)$
nat : grupoUnido ← m.posiciones[Latitud(c2)][Longitud(c2)].grupo;	$O(1)$
itConj(coor) : it ← CrearIt(Coordenadas(m));	$O(lat_{max} \times long_{max})$
while *HaySiguiente(it)* **do**	$O(lat_{max} \times long_{max})$
coor : pos ← Siguiente(it);	$O(1)$
if *m.posiciones[Latitud(pos)][Longitud(pos)].grupo = grupoViejo* **then**	
m.posiciones[Latitud(pos)][Longitud(pos)].grupo ← grupoUnido;	$O(1)$
end if	
end while	
end if	

Complejidad: $O(lat_{max} \times long_{max})$
Justificación: para poder unir se debe crear un iterador del conjunto de las coordenadas existentes. La máxima cantidad de iteraciones a realizar con ese iterador es la misma cantidad de coordenadas existentes. En el mejor caso no se deben unir las posiciones, y no se realiza ninguna operación ($\Theta(1)$).

3. Módulo Coordenada

Interfaz

se explica con: COORDENADA.

géneros: `coor`.

Operaciones básicas de Coordenada

CREARCOOR(**in** x, y : `nat`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ crearCoor(y,x)}
Complejidad: $O(1)$
Descripción: creación de una coordenada con latitud x y longitud y.

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, d: `coor`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ distEuclidea(c, d)]}
Complejidad: $O(1)$
Descripción: devuelve la distancia euclideana entre las coordenadas c y d.

COORDENADAARRIBA(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ coordenadaArriba(c)}
Complejidad: $O(1)$
Descripción: devuelve una coordenada con latitud de c + 1, y longitud de c.

COORDENADAABAJO(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {latitud(c) > 0}
Post $\equiv$ {*res* $=_{\text{obs}}$ coordenadaAbajo(c)}
Complejidad: $O(1)$
Descripción: devuelve una coordenada con latitud de c - 1, y longitud de c.

COORDENADAALADERECHA(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ coordenadaALaDerecha(c)}
Complejidad: $O(1)$
Descripción: devuelve una coordenada con latitud de c, y longitud de c + 1.

COORDENADAALAIZQUIERDA(**in** c: `coor`) $\rightarrow$ *res* : `coor`
Pre $\equiv$ {longitud(c) > 0}
Post $\equiv$ {*res* $=_{\text{obs}}$ coordenadaALaIzquierda(c)}
Complejidad: $O(1)$

Descripción: devuelve una coordenada con latitud de c, y longitud de c - 1.

Representación

Representación de Coordenada

`Coordenada` **se representa con** `eCoor`

donde `eCoor` es `tupla`(*lat*: `nat` , *long*: `nat`)

Invariante de Representación

Rep : eCoor $\longrightarrow$ bool
Rep(e) $\equiv$ true

Función de Abstracción

Abs : eCoor e $\longrightarrow$ coor $\{\text{Rep}(e)\}$
Abs(e) $=_{\text{obs}}$ c: coor | latitud(c) = e.lat $\wedge$ longitud(c) = e.long

Algoritmos

iCrearCoor (**in** y : `nat`, **in** x : `nat`) $\rightarrow$ res: eCoor	
res.latitud $\leftarrow$ y;	$O(1)$
res.longitud $\leftarrow$ x;	$O(1)$

Complejidad: $O(1)$
Justificación: todos los algortimos de coordenadas consisten en asignaciones, accesos u operaciones aritméticas básicas en $\Theta(1)$

iLatitud (**in** e : `eCoor`) $\rightarrow$ res: nat	
res $\leftarrow$ e.latitud;	$O(1)$

iLongitud (**in** e : `eCoor`) $\rightarrow$ res: nat	
res $\leftarrow$ e.longitud;	$O(1)$

iCoordenadaArriba (**in** e : `eCoor`) $\rightarrow$ res: eCoor	
res $\leftarrow$ CrearCoor(Latitud(e)+1, Longitud(e));	$O(1)$

iCoordenadaAbajo (**in** e : `eCoor`) $\rightarrow$ res: eCoor	
res $\leftarrow$ CrearCoor(Latitud(e)-1, Longitud(e));	$O(1)$

iCoordenadaALaDerecha (**in** e : `eCoor`) $\rightarrow$ res: eCoor	
res $\leftarrow$ CrearCoor(Latitud(e), Longitud(e)+1);	$O(1)$

iCoordenadaALaIzquierda (**in** e : `eCoor`) $\rightarrow$ res: eCoor	
res $\leftarrow$ CrearCoor(Latitud(e), Longitud(e)-1);	$O(1)$

iDistEuclidea (**in** *e*1 : `eCoor`, **in** *e*2 : `eCoor`) → res: nat	
nat: la ← 0;	$O(1)$
nat: lo ← 0;	$O(1)$
if *Latitud(e1) > Latitud(e2)* **then**	
| la ← (Latitud(e1) - Latitud(e2)) × (Latitud(e1) - Latitud(e2));	$O(1)$
else	
| la ← (Latitud(e2) - Latitud(e1)) × (Latitud(e2) - Latitud(e1));	$O(1)$
end if	
if *Longitud(e1) > Longitud(e2)* **then**	
| lo ← (Longitud(e1) - Longitud(e2)) × (Longitud(e1) - Longitud(e2));	$O(1)$
else	
| lo ← (Longitud(e2) - Longitud(e1)) × (Longitud(e2) - Longitud(e1));	$O(1)$
end if	
res ← la + lo;	$O(1)$

4. Módulo ConjuntoJugadores

Este conjunto aprovecha el orden absoluto de los jugadores (nats) para almacenar los datos en un arbol de búsqueda autobalanceado (AVL). Esto permite agregar, buscar y remover cualquier elemento en tiempo logarítmico.

Usaremos $\#j$ para denotar la cantidad de entradas del diccionario.

Interfaz

se explica con: Conjunto(Jugador), Iterador Unidireccional(Jugador).

géneros: `conjJugs`, `itConjJugs`.

Operaciones básicas de ConjuntoJugadores

Vacio() $\rightarrow$ *res* : `conjJugs`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ $\emptyset$ }
Complejidad: $O(1)$
Descripción: Crea el conjunto vacío

Agregar(**in/out** c: `conjJugs`, **in** j: `jugador`)
Pre $\equiv$ $\{c =_{\text{obs}} c_0\}$
Post $\equiv$ $\{c =_{\text{obs}} \text{Ag}(j, c_0)\}$
Complejidad: $O(log(\#j))$
Descripción: Agrega al conjunto j
Aliasing: se almacenan copias de j.

Pertenece?(**in** c: `conjJugs`, **in** j: `jugador`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} j \in c)]\}$
Complejidad: $O(log(\#j))$
Descripción: devuelve `true` si el jugador esta en el conjunto.

Borrar(**in/out** c: `conjJugs`, **in** j: `jugador`)
Pre $\equiv$ $\{c =_{\text{obs}} c_0)\}$
Post $\equiv$ $\{c =_{\text{obs}} \text{borrar}(j, c_0)\}$
Complejidad: $O(log(\#j))$
Descripción: borra un jugador del conjunto.

Operaciones básicas del iterador

El iterador que presentamos no permite modificar el conjunto recorrido.

CrearIt(**in** c: `conjJugs`) $\rightarrow$ *res* : `itConjJugs`
Pre $\equiv$ {true}
Post $\equiv$ {true}
Complejidad: $O(1)$
Descripción: crea un iterador unidireccional de los jugadores validos o expulsados.
Aliasing: el iterador se invalida si y solo si se elimina el siguiente elemento del iterador. Además, siguientes(res) podría cambiar completamente ante cualquier operación que modifique el conjunto.

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

Actual(**in** it: `itConjJugs`) $\rightarrow$ *res* : `jugador`
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ $\{res =_{\text{obs}} \text{Actual}(it)\}$
Complejidad: $O(1)$)

Descripción: devuelve el elemento siguiente a la posición del iterador.

AVANZAR(**in/out** it : `itConjJugs`)
Pre $\equiv \{it = it_0 \wedge \text{HayMas?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{Avanzar}(it_0)\}$
Complejidad: $O(log(\#j))$
Descripción: avanza a la posición siguiente del iterador.

Representación

Representación de conjJugs

`conjJugs` **se representa con** `eJugs`

donde `eJugs` es `tupla`(*raiz*: `puntero(Nodo)`)

donde `Nodo` es `tupla`(*hijoIzq*: `puntero(Nodo)` , *hijoDer*: `puntero(Nodo)` , *id*: `jugador` , *alto*: `nat`)

Invariante de Representación de ConjuntoJugadores

1. No hay jugadores repetidos
2. Para cualquier nodo del arbol de jugadores, su hijo izquierdo tiene un id menor que él y su hijo derecho uno mayor
3. La altura de un Nodo es la altura del hijo con mayor altura mas 1
4. El factor de balanceo es ≤ 1 (donde el factor de balanceo es el modulo de la diferencia de las alturas de los hijos)

Rep : eJugs $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ sinRepetidos(e.raiz, $\emptyset$) $\wedge$
ABB?(e.raiz) $\wedge$ altoValido?(e.raiz)

Función de Abstracción

Abs : eJugs e $\longrightarrow$ conjJugs {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: conjJugs | ($\forall j$: jugador) (def?(j, d) = definido?(j, e.raiz) $\wedge_{\text{L}}$
(def?(j, d) $\Rightarrow_{\text{L}}$ obtener(j, d) = obtener(j, e.raiz)))

Operaciones auxiliares

sinRepetidos : puntero(nodo) $\times$ conj(nat) $\longrightarrow$ bool
ABB? : puntero(Nodo) *nodo* $\longrightarrow$ bool
menor? : puntero(Nodo) *padre* $\times$ puntero(Nodo) *hijo* $\longrightarrow$ bool {padre $\neq$ NULL}
altoValido? : puntero(nodo) $\longrightarrow$ bool
mayorAltura : nat $\times$ nat $\longrightarrow$ nat
factorDeBalanceo : nat $\times$ nat $\longrightarrow$ nat
cantidadHijos : puntero(Nodo) $\longrightarrow$ nat
definido? : jugador $\times$ puntero(nodo) $\longrightarrow$ bool
obtener : jugador j $\times$ puntero(nodo) p $\longrightarrow$ nat {definido?(j,p)}

sinRepetidos(padre, ids) $\equiv$ padre = NULL $\vee_{\text{L}}$(padre.id $\notin$ ids $\wedge$
sinRepetidos(padre.hijoIzq, Ag(padre.id, c)) $\wedge$
sinRepetidos(padre.hijoDer, Ag(padre.id, c)))

ABB?(nodo) $\equiv$ (nodo = NULL) $\vee_{\text{L}}$
(menor?(nodo, (*nodo).hijoDer) $\wedge$ $\neg$menor?(nodo, (*nodo).hijoIzq) $\wedge$
ABB?((*nodo).hijoIzq) $\wedge$ ABB?((*nodo).hijoDer))

menor?(padre, hijo) $\equiv$ (hijo = NULL) $\vee_{\text{L}}$ padre.id < hijo.id)

altoValido?(nodo) $\equiv$ nodo = NULL $\vee_{\text{L}}$
((*nodo).alto = mayorAltura((*nodo).hijoIzq, (*nodo).hijoDer) + 1 $\wedge$
factorDeBalanceo(cantidadHijos((*nodo).hijoIzq), cantidadHijos((*nodo).hijoDer)) ≤ 1

mayorAltura(izq, der) $\equiv$ **if** cantidadHijos(izq) < cantidadHijos(der) **then**
cantidadHijos(izq)
else
cantidadHijos(der)
fi

cantidadHijos(nodo) $\equiv$ **if** nodo = NULL **then**
0
else
β((*nodo).hijoIzq $\neq$ NULL) + cantidadHijos((*nodo).hijoIzq) +
β((*nodo).hijoDer $\neq$ NULL) + cantidadHijos((*nodo).hijoDer)
fi

factorDeBalanceo(izq, der) $\equiv$ **if** izq < der **then** der - izq **else** izq - der **fi**

definido?(j , p) $\equiv$ p $\neq$ NULL $\wedge_{\text{L}}$ (p.id = j $\vee$ definido?(j, p.hijoIzq) $\vee$ definido?(j, p.hijoDer))

obtener(j , p) $\equiv$ **if** p.id = j **then**
p.cantPokemons
else
if definido?(j, p.hijoIzq) **then** obtener(j, p.hijoIzq) **else** obtener(j, p.hijoDer) **fi**
fi

Representación del iterador

`itConjJugs` **se representa con** `itJugs`

donde `itJugs` es `tupla`(*conjunto*: `puntero(conjJugs)` , *pila*: `pila(puntero(Nodo))`)

Invariante de Representación del iterador

1. El conjunto no es nulo
2. Ningún elemento de la pila es nulo
3. Todo elemento de la pila es menor al anterior

Rep : itJugs $\longrightarrow$ bool
Rep(it) $\equiv$ true $\Longleftrightarrow$ it.conjunto $\neq$ NULL $\wedge$ PilaValida?(it.pila)

Función de Abstracción del iterador

Abs : itJugs it $\longrightarrow$ itUni(nat) {Rep(it)}
Abs(it) $=_{\text{obs}}$ b: itUni(nat) | Siguientes(b) = Sigs(it.pila)
PilaValida? : pila(puntero(Nodo)) $\longrightarrow$ bool
Sigs : pila(puntero(Nodo)) $\longrightarrow$ secu(jugador)
Hijos : puntero(Nodo) n $\longrightarrow$ secu(jugador) {n $\neq$ NULL}
HijosDer : pila(puntero(Nodo)) p $\longrightarrow$ secu(jugador)

PilaValida?(p) $\equiv$ vacia?(p) $\vee_{\text{L}}$ (PilaValida?(desapilar(p)) $\wedge$ tope(p) $\neq$ NULL $\wedge_{\text{L}}$
(vacia?(desapilar(p)) $\vee_{\text{L}}$ (*tope(p)).id < (*tope(desapilar(p))).id))
Sigs(p) $\equiv$ **if** vacia?(p) **then** <> **else** Hijos(tope(p)) & HijosDer(desapilar(p)) **fi**
Hijos(n) $\equiv$ **if** n = NULL **then** <> **else** (*n).id $\bullet$ (Hijos((*n).hijoIzq) & Hijos((*n).hijoDer)) **fi**

HijosDer(n) $\equiv$ **if** vacia?(p) **then** <> **else** Hijos((*tope(p)).hijoDer) & HijosDer(desapilar(p)) **fi**

Algoritmos

Algoritmos de ConjuntoJugadores

iVacio () → res: eJugs	
res.raiz ← NULL;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se inicializan los punteros como nulos.

iPertenece? (**in** c: `eJugs`, **in** j: `jugador`) → res: bool	
res ← Buscar(p.raiz) ≠ NULL;	$O(log(\#j))$

iAgregar (**in/out** c: `eJugs`, **in** j: `jugador`)	
p.raiz ← Insertar(j, p.raiz);	$O(log(\#j))$

iBorrar (**in/out** c: `eJugs`, **in** j: `jugador`)	
p.raiz ← Remover(j, p.raiz);	$O(log(\#j))$

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

Justificación: la complejidad de buscar, agregar o eliminar un elemento es de orden de la altura del arbol, que por invariante de AVL sabemos que es $log(\#j)$

Algoritmos auxiliares

Buscar (**in** j: `jugador`, **in** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: `true`

Post: res es igual al nodo hijo o el que viene por parametro que tenga el id del jugador a menos que no este en cuyo caso en NULL

```
if nodo = NULL or (*nodo).id = jugador then
    res ← nodo;                                              O(1)
else if j < (*nodo).id then
    res ← Buscar(j, (*nodo).hijoIzq);                        O(h)
else
    res ← Buscar(j, (*nodo).hijoDer);                        O(h)
end if
```

Complejidad: $O(h)$

Insertar (**in** j: `jugador`, **in/out** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: `true`

Post: res es igual a la raiz del subarbol del nodo pasado por parámetro pero con el nuevo elemento y balanceado

```
if nodo = NULL then
    res ← CrearNodo(j);                                      O(1)
else
    if j = nodo.id then
        res ← nodo;                                          O(1)
    else
        if j < nodo.id then
            (*nodo).hijoIzq ← Insertar(j, (*nodo).hijoIzq);  O(h)
        else
            (*nodo).hijoDer ← Insertar(j, (*nodo).hijoDer);  O(h)
        end if
        res ← Balancear(nodo);                               O(1)
    end if
end if
```

Complejidad: $O(h)$

Remover (**in** *j* : `jugador`, **in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: entre los nodo que desprenden del nodo del parametro se encuentra el id j.
Post: res es el mismo nodo pero ahora con el hijo removido en esa rama excepto que sea el nodo que hay que remover el cual se convierte en null

```
if nodo = NULL then
    res ← nodo;                                                          O(1)
else if j < (*nodo).id then
    (*nodo).hijoIzq ← Remover(j, (*nodo).hijoIzq);                       O(h)
    res ← Balancear(nodo);                                               O(1)
else if j > (*nodo).id then
    (*nodo).hijoDer ← Remover(j, (*nodo).hijoDer);                       O(h)
    res ← Balancear(nodo);                                               O(1)
else
    puntero(Nodo): i ← (*nodo).hijoIzq;                                  O(1)
    puntero(Nodo): d ← (*nodo).hijoDer;                                  O(1)
    if d = NULL then
        res ← i;                                                         O(1)
    else
        puntero(Nodo): minimo ← BuscarMinimo(d);                         O(h)
        minimo.hijoDer ← RemoverMinimo(d);                               O(h)
        minimo.hijoIzq ← i;                                              O(1)
        res ← Balancear(minimo);                                         O(1)
    end if
end if
```

Complejidad: $O(h)$

BuscarMinimo (**in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: true
Post: res es el puntero al minimo hijo del nodo(o el si es el menor)

```
if (*nodo).hijoIzq = NULL then
    res ← nodo;                                                          O(1)
else
    res ← BuscarMinimo((*nodo).hijoIzq);                                 O(h)
end if
```

RemoverMinimo (**in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: true
Post: res es el el mismo nodo pero con el nodo de mas a la izquierda eliminado(al padre del menor, se le pasan los hijos mayores si es que tiene)

```
if (*nodo).hijoIzq = NULL then
    res ← (*nodo).hijoDer;                                               O(1)
else
    (*nodo).hijoIzq = RemoverMinimo((*nodo).hijoIzq);                    O(h)
    res ← Balancear(nodo);                                               O(1)
end if
```

Complejidad: $O(h)$, donde h es la altura del nodo

Justificación: como sabemos la altura de un nodo es la mayor altura entre sus hijos + 1, como en cada iteración estamos haciendo una llamada recursiva a uno de sus hijos hasta que ese hijo sea nulo, en el peor de los casos llamamos recursivamente al hijo con mayor altura y en cada llamada estamos disminuyendo en 1 la altura.

La complejidad es equivalente a $log(n)$, donde n es la cantidad de hijos del nodo, ya que al estar ordenado como ABB y balanceado con invariante de AVL, cada llamada recursiva a cada subarbol reduce el tamaño de la entrada aproximadamente por la mitad.

CrearNodo (**in** j : `jugador`) → res: puntero(Nodo)

Pre: `true`
Post: *res* apunta a un nodo de alto 1 con id j

Nodo: nuevo;	$O(1)$
nuevo.hijoIzq ← NULL;	$O(1)$
nuevo.hijoDer ← NULL;	$O(1)$
nuevo.alto ← 1;	$O(1)$
nuevo.id ← j;	$O(1)$
res ← &nuevo;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se crea e inicializa un (*nodo). Como inicialmente se agrega como hoja del arbol, su altura es 1.

ArreglarAlto (**in/out** *nodo* : `puntero(Nodo)`)

Pre: *nodo* ≠ NULL
Post: el módulo del factor de balanceo de *res* es menor a 2

nat: alturaIzq ← Altura((*nodo).hijoIzq);	$O(1)$
nat: alturaDer ← Altura((*nodo).hijoDer);	$O(1)$
if *alturaIzq* < *alturaDer* **then**	
| (*nodo).altura ← alturaDer + 1;	$O(1)$
else	
| (*nodo).altura ← alturaIzq + 1;	$O(1)$
end if	

Complejidad: $O(1)$
Justificación: solo se realizan comparaciones para las alturas de los hijos

Altura (**in** *nodo* : `puntero(Nodo)`) → res: nat

Pre: `true`
Post: *res* es 0 si *nodo* es nulo, o igual a la altura del nodo al que apunta en caso contrario

if *nodo* = *NULL* **then**	
| res ← 0;	$O(1)$
else	
| res ← (*nodo).alto;	$O(1)$
end if	

Complejidad: $O(1)$
Justificación: solo se verifica si es nulo o se lee un componente.

Balancear (**in/out** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: *nodo* ≠ NULL
Post: el módulo del factor de balanceo de *res* es menor a 2

ArreglarAlto(nodo);	$O(1)$
nat: alturaIzq ← Altura((*nodo).hijoIzq);	$O(1)$
nat: alturaDer ← Altura((*nodo).hijoDer);	$O(1)$
if *alturaDer > alturaIzq* ***and*** *alturaDer - alturaIzq = 2* **then**	
if *Altura((*nodo).hijoDer.hijoIzq) > Altura((*nodo).hijoDer.hijoDer)* **then**	
(*nodo).hijoDer ← rotarDer((*nodo).hijoDer;	$O(1)$
end if	
res ← rotarIzq(nodo);	$O(1)$
else if *alturaIzq > alturaDer* ***and*** *alturaIzq - alturaDer = 2* **then**	
if *Altura((*nodo).hijoIzq.hijoDer) > Altura((*nodo).hijoIzq.hijoIzq)* **then**	
(*nodo).hijoIzq ← rotarIzq((*nodo).hijoIzq);	$O(1)$
end if	
res ← rotarDer(nodo);	$O(1)$
else	
res ← nodo;	$O(1)$
end if	

rotarDer (**in** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: *nodo* tiene un hijo izquierdo
Post: *res* es el hijo izquierdo de nodo, y arbol se rota a la derecha

puntero(nodo) : aux ← (*nodo).hijoIzq;	$O(1)$
(*nodo).hijoIzq ← aux.hijoDer;	$O(1)$
aux.hijoDer ← nodo;	$O(1)$
ArreglarAlto(nodo);	$O(1)$
ArreglarAlto(aux);	$O(1)$
res ← aux;	$O(1)$

rotarIzq (**in** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: *nodo* tiene un hijo derecho
Post: *res* es el hijo derecho de nodo, y arbol se rota a la izquierda

puntero(nodo) : aux ← (*nodo).hijoDer;	$O(1)$
(*nodo).hijoDer ← aux.hijoIzq;	$O(1)$
aux.hijoIzq ← nodo;	$O(1)$
ArreglarAlto(nodo);	$O(1)$
ArreglarAlto(aux);	$O(1)$
res ← aux;	$O(1)$

Complejidad: $O(1)$
Justificación: las rotaciones son una serie de comparasiones y asignaciones de punteros ($\Theta(1)$). El invariante de orden del arbol de búsqueda (ABB) se preserva luego de cada rotación. El invariante de balanceo de AVL se restaura para cada subarbol al final de `Balancear`, pero no luego de cada rotación individual (puede ser necesario rotar un subarbol de manera temporalmente imbalanceada para restaurar el balance de un arbol).

Algoritmos del iterador

CrearIt (**in** *c*: `eJugs`) → res: itJugs

res.conjunto ← &(c);	$O(1)$
res.pila ← Vacía();	$O(1)$
if ***not*** *c.raiz = NULL* **then**	
Apilar(res.pila, c.raiz);	$O(1)$
end if	

Complejidad: $O(1)$
Justificación: solo se inicializa el iterador.

HayMas (**in** *it* : `itJugs`) → res: bool	
res ← EsVacía?(it.pila);	$O(1)$

Actual (**in** *it* : `itJugs`) → res: jugador	
res ← (*Tope(it.pila)).id;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se utilizan operaciones básicas de pila.

Avanzar (**in/out** *it* : `itJugs`)	
while *not* *EsVacía(it.pila)* **do**	$O(1)$
puntero(Nodo) : tope ← Tope(it.pila);	$O(1)$
if *not (*tope).hijoIzq = NULL* **then**	
Apilar(it.pila, (*tope).hijoIzq);	$O(1)$
break;	$O(1)$
else	
Desapilar(it.pila);	$O(1)$
if *not (*tope).hijoDer = NULL* **then**	
Apilar(it.pila, (*tope).hijoDer);	$O(1)$
break;	$O(1)$
end if	
end if	
end while	

Complejidad: $O(1)$
Justificación: la pila contiene aquellas posiciones para las cuales todavía no se visitó el nodo derecho, y en el caso del tope, tampoco se revisó el nodo izquierdo, además de estar ordenadas por posición en el arbol (las primeras que se agregan correpsonden a posiciones superiores).

El peor caso posible para un ABB normal es que la pila contenga todos los elementos del conjunto (ningún nodo del arbol tiene nodo derecho).

En el caso particular de los AVL la complejidad es constante porque, por el invariante de factores de balance, la cantidad de posiciones que tenemos que recorrer "hacia arriba"para encontrar un nodo derecho es como máximo 2. Por ende, la cantidad de ciclos a realizar tiene máximo constante y no depende del tamaño del arbol.

5. Módulo DiccionarioString(α)

El diccionario se representa con un trie, que permite lectura, inserción y modificación en $\Theta(|clave|)$, donde `clave` es la clave consultar o modificar.

Las claves se guardan al mismo tiempo en un Conjunto Lineal, siempre con inserción rápida ya que al insertar en el trie podemos saber si la clave existía de antemano.

Al tener que mantener las copias de las claves, remover un elemento cuesta $O(|c_{max}| * \#c)$ (ya que debe removerse del conjunto de claves).

Usaremos $copy(s)$ para denotar el costo de copiar el elemento $s \in \alpha$. Llamaremos $|c_{max}|$ a la longitud de la clave más larga, y $\#c$ a la cantidad de claves definidas. Se asume que la complejidad de comparar dos Strings es $O(|c_{max}|)$.

Interfaz

parámetros formales
  géneros α
  función COPIAR(**in** s: α) $\rightarrow$ res : α
  Pre $\equiv$ {true}
  Post $\equiv$ {$res =_{\text{obs}} s$}
  Complejidad: $\Theta(copy(s))$
  Descripción: función de copia de α's.

se explica con: DICCIONARIO(κ,σ), ITERADOR BIDIRECCIONAL(α).

géneros: `diccString(`α`), itDiccString(`α`)`.

servicios usados: CONJUNTO LINEAL(α)

Operaciones básicas de DiccionarioString(α)

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

DEFINIR(**in/out** d: `diccString(`α`)`, **in** c: `string`, **in** s: α)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(c, s, d_0)}
Complejidad: $O(|c_{max}| + copy(s))$
Descripción: define una clave en el diccionario.
Aliasing: se almacenan copias de c y s.

DEFINIDO?(**in** d: `diccString(`α`)`, **in** c: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(c, d)]}
Complejidad: $O(|c_{max}|)$
Descripción: devuelve `true` si la clave esta definida en el diccionario.

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

BORRAR(**in/out** d: `diccString(`α`)`, **in** c: `string`)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge$ def?(c, d_0)}
Post $\equiv$ {$d =_{\text{obs}}$ borrar(c, d_0)}
Complejidad: $O(|c_{max}| * \#c)$
Descripción: borra una clave y su significado del diccionario.

Claves(**in** d: `diccString(`α`)`) $\rightarrow$ res : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(res = claves(d))}
Complejidad: $O(1)$
Descripción: devuelve el conjunto de las claves del diccionario.
Aliasing: res no es modificable.

Operaciones del iterador

El iterador que presentamos no permite modificar el diccionario recorrido.

CrearIt(**in** d: `DiccString(`α`)`) $\rightarrow$ res : `itDiccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby($res.itClaves$), $d.claves$)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $O(1)$
Descripción: crea un iterador bidireccional del diccionario usando el iterador del conjunto de sus claves.
Aliasing: el iterador se invalida si y sólo si se elimina el elemento siguiente del iterador. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique d.

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

HayAnterior(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ hayAnterior?(it)}
Complejidad: $O(1)$
Descripción: devuelve `true` si y sólo si en el iterador todavía quedan elementos para retroceder.

Siguiente(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : `tupla(String, `α`)`
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it))}
Complejidad: $O(|c_{max}|))$
Descripción: devuelve el elemento siguiente a la posición del iterador, como tupla clave-valor.
Aliasing: res no es modificable.

Anterior(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : `tupla(String, `α`)`
Pre $\equiv$ {HayAnterior?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Anterior(it))}
Complejidad: $O(|c_{max}|)$
Descripción: devuelve el elemento anterior a la posición del iterador, como tupla clave-valor.
Aliasing: res no es modificable.

Avanzar(**in/out** it: `itDiccString(`α`)`)
Pre $\equiv$ {$it = it_0 \wedge$ HaySiguiente?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $O(1)$
Descripción: avanza a la posición siguiente del iterador.

Retroceder(**in/out** it: `itDiccString(`α`)`)
Pre $\equiv$ {$it = it_0 \wedge$ HayAnterior?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Retroceder(it_0)}
Complejidad: $O(1)$

Descripción: retrocede a la posición anterior del iterador.

Representación

Representación de DiccionarioString(α)

`DiccionarioString(`α`)` **se representa con** `dicStr`

donde `dicStr` es `tupla(`*raiz*: `puntero(Nodo)` , *claves*: `conj(String)`)

donde `Nodo` es `tupla(`*hijos*: `arreglo[256] de puntero(Nodo)` , *valor*: α , *contieneValor*: `bool`)

Invariante de Representación del Diccionario

1. La raíz del trie es un nodo válido no nulo que no guarda valor
2. Los punteros son únicos (se referencian desde un único punto del trie)[3]

Rep : dicStr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ (e.raiz $\neq$ NULL) $\wedge_{\text{L}}\neg$(e.raiz.contieneValor)

Función de Abstracción del Diccionario

Abs : dicStr e $\longrightarrow$ diccString(α) {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: diccString(α) | ($\forall s$: string) (def?(c, d) = definido?(e.raiz, c, 0) $\wedge_{\text{L}}$
(def?(c, d) $\Rightarrow_{\text{L}}$ obtener(c, d) = obtener(e.raiz, c, 0)))

definido? : puntero(Nodo) n × string c × nat i $\longrightarrow$ bool {n $\neq$ NULL $\wedge$ i $\leq$ longitud(c)}
definido?(nodo, clave, i) $\equiv$ **if** i < Longitud(clave) **then**
¬(nodo.hijos[ord(clave[i])] = NULL) $\wedge_{\text{L}}$
definido?(nodo.hijos[ord(clave[i])], clave, i+1)
else
nodo.contieneValor
fi

obtener : puntero(Nodo) n × string c × nat i $\longrightarrow$ α {n $\neq$ NULL $\wedge$ i $\leq$ longitud(c) $\wedge_{\text{L}}$ definido?(n, c, i)}
obtener(nodo, clave, i) $\equiv$ **if** i < Longitud(clave) **then**
obtener(nodo.hijos[ord(clave[i])], clave, i+1)
else
nodo.valor
fi

Representación del iterador

`Iterador DiccionarioString(`α`)` **se representa con** `itDicStr`

donde `itDicStr` es `tupla(`*itClaves*: `itConj(String)` , *dic*: `puntero(dicStr)`)

Invariante de Representación del iterador

1. El diccionario no es nulo
2. El iterador de las claves corresponde a las claves del diccionario

Rep : itDicStr $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ (it.dic $\neq$ NULL) $\wedge_{\text{L}}$ esPermutacion(SecuSuby(it.itClaves), (*it.dic).claves)

Función de Abstracción del iterador

Abs : itDicStr it $\longrightarrow$ itBi(tupla(String, α)) {Rep(it)}
Abs(it) $=_{\text{obs}}$ b: itBi(tupla(String, α)) | Anteriores(b) = Tuplas(Anteriores(it.itClaves), *it.dic) $\wedge$
Siguientes(b) = Tuplas(Siguientes(it.itClaves), *it.dic)

Tuplas : secu(String) cs × dicc(String, α) dic $\longrightarrow$ secu(tupla(String, α))

[3] No pudimos expresar esto en rep, lo dejamos en castellano

Tuplas(cs, dic) $\equiv$ **if** vacia?(cs) **then** <> **else** $\langle$ prim(cs), obtener(prim(cs), dic) $\rangle$ • Tuplas(fin(cs), dic) **fi**

Algoritmos

Algorítmos de DiccionarioString(α)

iCrearDiccionario () → res: dicStr	
res.raiz ← CrearNodo();	$O(1)$
res.claves ← Vacio();	$O(1)$

Complejidad: $O(1)$
Justificación: solo se crea un nodo vacío y el conjunto vacío de claves

CrearNodo () → res: puntero(Nodo)	
Pre: `true`	
Post: Se crea un puntero a nodo vacio	
Nodo: nuevo;	$O(1)$
nuevo.contieneValor ← false;	$O(1)$
for $i \leftarrow 0$ **to** 255 **do**	$O(1)$
⎸ nuevo.hijos[i] ← NULL;	$O(1)$
end for	
res ← &nuevo;	$O(1)$

Complejidad: $O(1)$
Justificación: se asigna NULL a una cantidad fija de posiciones

iDefinir (**in/out** *dic*: `dicStr`, **in** *c*: `String`, **in** *s*: α)	
puntero(Nodo): entrada ← dic.raiz;	$O(1)$
for $i \leftarrow 0$ **to** *Longitud(c)* **do**	$O(\|c_{max}\|)$
⎸ **if** *(*entrada).hijos[ord(c[i])] = NULL* **then**	$O(1)$
⎸ ⎸ (*entrada).hijos[ord(c[i])] ← CrearNodo();	$O(1)$
⎸ **end if**	
⎸ entrada ← (*entrada).hijos[ord(c[i])];	$O(1)$
end for	
if *not (*entrada).contieneValor* **then**	$O(1)$
⎸ AgregarRapido(dic.claves, c);	$O(1)$
⎸ (*entrada).contieneValor ← true;	$O(1)$
end if	
(*entrada).valor ← s;	$O(copy(s))$

Complejidad: $O(|c_{max}| + copy(s))$
Justificación: el ciclo itera hasta el largo de la clave a insertar y luego la copia para insertarla; la clave solo se agrega al conjunto si no estaba definida antes

iDefinido? (**in/out** *dic*: `dicStr`, **in** *c*: `String`) → res: bool	
puntero(Nodo): entrada ← dic.raiz;	$O(1)$
for $i \leftarrow 0$ **to** *Longitud(c)* **do**	$O(\|c_{max}\|)$
⎸ **if** *entrada = NULL* **then** break;	$O(1)$
⎸ entrada ← (*entrada).hijos[ord(c[i])];	$O(1)$
end for	
res ← not entrada = NULL **and** (*entrada).contieneValor;	$O(1)$

Complejidad: $O(|c_{max}|)$
Justificación: el ciclo como máximo itera hasta el largo de la clave buscada

iObtener (**in/out** *dic*: `dicStr`, **in** *c*: `String`) → res: α	
puntero(Nodo): entrada ← dic.raiz;	$O(1)$
for $i \leftarrow 0$ **to** *Longitud(c)* **do**	$O(\|c_{max}\|)$
⎸ entrada ← (*entrada).hijos[ord(c[i])];	$O(1)$
end for	
res ← (*entrada).valor;	$O(copy(s))$

Complejidad: $O(|c_{max}| + copy(s))$
Justificación: el ciclo itera hasta el largo de la clave a obtener y luego la copia para retornarla

iBorrar (**in/out** *dic*: `dicStr`, **in** *c*: `String`)	
puntero(Nodo): entrada ← dic.raiz;	$O(1)$
for $i \leftarrow 0$ **to** *Longitud(c)* **do**	$O(\|c_{max}\|)$
⎸ entrada ← (*entrada).hijos[ord(c[i])];	$O(1)$
end for	
(*entrada).contieneValor ← false;	$O(1)$
Eliminar(dic.claves, c);	$O(\|c_{max}\| * \#c)$

Complejidad: $O(|c_{max}| * \#c)$
Justificación: el ciclo itera hasta el largo de la clave a borrar y luego asigna un booleano, y borra la clave del conjunto

iClaves (**in** *dic*: `dicStr`) → res: conj(String)	
res ← dic.claves;	$O(1)$

Complejidad: $O(1)$
Justificación: el conjunto de claves se devuelve por referencia

Algorítmos del iterador

iCrearIt (**in** *dic*: `dicStr`) → res: itDicStr	
res.itClaves ← CrearIt(dic.claves);	$O(1)$
res.dic ← &dic;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se guarda un puntero al diccionario y se crea el iterador de las claves

iHaySiguiente (**in** *it*: `itDicStr`) → res: bool	
res ← HaySiguiente(it.itClaves);	$O(1)$

Complejidad: $O(1)$
Justificación: comparte la complejidad del iterador de las claves

iHayAnterior (**in** *it*: `itDicStr`) → res: bool	
res ← HayAnterior(it.itClaves);	$O(1)$

Complejidad: $O(1)$
Justificación: comparte la complejidad del iterador de las claves

iSiguiente (**in** *it*: `itDicStr`) → res: tupla(calve: String, significado: α)	
String: clave ← Siguiente(it.itClaves);	$O(1)$
res ← ⟨ clave, Obtener(*it.dic, clave) ⟩;	$O(\|c_{max}\|)$

Complejidad: $O(|c_{max}|)$
Justificación: comparte la complejidad del iterador de las claves, y luego consulta el diccionario a través de su función de obtener

iAnterior (**in** *it* : `itDicStr`) → res: tupla(calve: String, significado: α)	
String: clave ← Anterior(it.itClaves);	$O(1)$
res ← ⟨ clave, Obtener(*it.dic, clave) ⟩;	$O(\|c_{max}\|)$

Complejidad: $O(|c_{max}|)$
Justificación: comparte la complejidad del iterador de las claves, y luego consulta el diccionario a través de su función de obtener

iAvanzar (**in/out** *it* : `itDicStr`)	
it.itClaves ← Avanzar(it.itClaves);	$O(1)$

Complejidad: $O(1)$
Justificación: comparte la complejidad del iterador de las claves

iRetroceder (**in/out** *it* : `itDicStr`)	
it.itClaves ← Retroceder(it.itClaves);	$O(1)$

Complejidad: $O(1)$
Justificación: comparte la complejidad del iterador de las claves

6. Módulo DiccionarioPrioridad

Este diccionario utiliza dos criterios de ordenamiento: por la clave y por el significado. Esto permite definir, obtener y remover cualquier elemento al igual que buscar el mínimo elemento en tiempo logarítmico.

En el ordenamiento por significado, para los casos en los que hay dos significados iguales, se utiliza nuevamente la clave.

También se almacenan los valores del mínimo elemento para retornarlos en tiempo constante.

Usaremos $\#j$ para denotar la cantidad de entradas del diccionario.

Interfaz

se explica con: DICCIONARIO(JUGADOR, NAT).

géneros: `prior`.

Operaciones básicas de DiccionarioPrioridad

VACIO() $\rightarrow$ *res* : `prior`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ vacío}
Complejidad: $O(1)$
Descripción: Crea el diccionario vacío

DEFINIR(**in/out** p: `prior`, **in** j: `jugador`, **in** c: `nat`)
Pre $\equiv$ $\{p =_{\text{obs}} p_0\}$
Post $\equiv$ $\{p =_{\text{obs}} \text{definir}(j, c, p_0)\}$
Complejidad: $O(log(\#j))$
Descripción: define la cantidad de pokemons de un jugador en el diccionario.
Aliasing: se almacenan copias de j y c.

DEFINIDO?(**in** p: `prior`, **in** j: `jugador`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{def?}(j, p)]\}$
Complejidad: $O(log(\#j))$
Descripción: devuelve `true` si el jugador esta definido en el diccionario.

ESVACIO?(**in** p: `prior`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{esVacio?}(p)]\}$
Complejidad: $O(1)$
Descripción: devuelve `true` si el diccionario está vacio

BORRAR(**in/out** p: `prior`, **in** j: `jugador`)
Pre $\equiv$ $\{p =_{\text{obs}} p_0 \wedge \text{def?}(j, d_0)\}$
Post $\equiv$ $\{p =_{\text{obs}} \text{borrar}(j, p_0)\}$
Complejidad: $O(log(\#j))$
Descripción: borra un jugador del diccionario.

CLAVES(**in** p: `prior`) $\rightarrow$ *res* : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(res = claves(p))}
Complejidad: $O(\#j)$
Descripción: devuelve el conjunto de los jugadores del diccionario.
Aliasing: *res* no es modificable.

MENOR(**in/out** p: `prior`) $\rightarrow$ *res* : `jugador`
Pre $\equiv$ $\{\neg\emptyset?(\text{claves}(p))\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{menor}(p, \text{claves}(p))\}$
Complejidad: $O(log(\#j))$

Descripción: devuelve el jugador con menos pokemons en el diccionario

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Diccionario(Jugador, Nat) Extendido

extiende DICCIONARIO(JUGADOR, NAT)

otras operaciones (exportadas)

menor : dicc(nat × nat) d × conj(nat) c ⟶ nat {¬∅?(c) ∧ c ⊆ claves(d)}
esVacio? : dicc(nat × nat) ⟶ bool

axiomas

menor(d,c) ≡ **if** ∅?(dameUno(c)) **then**
sinUno(c)
else
if obtener(dameUno(c), d) < menor(d, sinUno(c)) **then**
dameUno(c)
else
menor(d, sinUno(c))
fi
fi
esVacio?(d) ≡ ∅?(claves(d))

Fin TAD

Representación

Representación de prior

`prior` **se representa con** `dicPri`

donde `dicPri` es `tupla`(*raizJugadores*: `puntero(Nodo)` , *raizCantidad*: `puntero(Nodo)` , *menorID*: `jugador` , *menor*: `nat`)

donde `Nodo` es `tupla`(*hijoIzq*: `puntero(Nodo)` , *hijoDer*: `puntero(Nodo)` , *id*: `jugador` , *cantPokemons*: `nat` , *alto*: `nat`)

Invariante de Representación

1. No hay jugadores repetidos
2. Un nodo cualquiera del arbol de jugadores su hijo izquierdo tiene menor id y su hijo derecho tiene mayor id
3. Un nodo cualquiera del arbol de cantidades su hijo izquierdo tiene menor o igual cantidad y su hijo derecho tiene mayor o igual cantidad
4. Un nodo cualquiera del arbol de cantidades su hijo izquierdo si tiene la misma cantidad tiene menor id y lo mismo para su hijo derecho pero mayor id
5. La altura de un Nodo es la altura del hijo con mayor altura mas 1
6. El factor de balanceo es ≤ 1 (donde el factor de balanceo es el modulo de la diferencia de las alturas de los hijos)
7. Los arboles tienen los mismos elementos
8. El menor es realmente el menor

Rep : dicPri ⟶ bool
Rep(e) ≡ true ⟺ sinRepetidos(e.raizJugadores, ∅) ∧ sinRepetidos(e.raizCantidad, ∅) ∧
ABB?(e.raizJugadores) ∧ altoValido?(e.raizJugadores) ∧
ABBespecial?(e.raizCantidad) ∧ altoValido?(e.raizCantidad) ∧
(∀ j: jugador) (definido?(j, e.raizJugadores) = definido?(j, e.raizCantidad) ∧$_{L}$
definido?(j, e.raizJugadores) ⇒$_{L}$ obtener(j, e.raizJugadores) = obtener(j, e.raizCantidad)) ∧
(e.raizCantidad ≠ NULL) ⇒$_{L}$ (e.menorID = menor(e.raizCantidad).id ∧ e.menor = menor(e.raizCantidad).cantPokemons)

Función de Abstracción

Abs : dicPri e $\longrightarrow$ prior {Rep(e)}

Abs(e) $=_{\text{obs}}$ d: prior | ($\forall j$: jugador) (def?(j, d) = definido?(j, e.raizJugadores) $\wedge_{\text{L}}$
(def?(j, d) $\Rightarrow_{\text{L}}$ obtener(j, d) = obtener(j, e.raizJugadores)))

Operaciones auxiliares

sinRepetidos : puntero(nodo) $\times$ conj(nat) $\longrightarrow$ bool

ABB? : puntero(Nodo) *nodo* $\longrightarrow$ bool

menor? : puntero(Nodo) *padre* $\times$ puntero(Nodo) *hijo* $\longrightarrow$ bool {padre $\neq$ NULL}

ABBespecial? : puntero(Nodo) *nodo* $\longrightarrow$ bool

menorEspecial? : puntero(Nodo) *padre* $\times$ puntero(Nodo) *hijo* $\longrightarrow$ bool {padre $\neq$ NULL}

altoValido? : puntero(nodo) $\longrightarrow$ bool

mayorAltura : nat $\times$ nat $\longrightarrow$ nat

factorDeBalanceo : nat $\times$ nat $\longrightarrow$ nat

cantidadHijos : puntero(Nodo) $\longrightarrow$ nat

definido? : jugador $\times$ puntero(nodo) $\longrightarrow$ bool

obtener : jugador j $\times$ puntero(nodo) p $\longrightarrow$ nat {definido?(j,p)}

menor : puntero(nodo) p $\longrightarrow$ puntero(nodo) {p $\neq$ NULL}

sinRepetidos(padre, ids) $\equiv$ padre = NULL $\vee_{\text{L}}$(padre.id $\notin$ ids $\wedge$
sinRepetidos(padre.hijoIzq, Ag(padre.id, c)) $\wedge$
sinRepetidos(padre.hijoDer, Ag(padre.id, c)))

ABB?(nodo) $\equiv$ (nodo = NULL) $\vee_{\text{L}}$
(menor?(nodo, (*nodo).hijoDer) $\wedge$ $\neg$menor?(nodo, (*nodo).hijoIzq) $\wedge$
ABB?((*nodo).hijoIzq) $\wedge$ ABB?((*nodo).hijoDer))

ABBespecial?(nodo) $\equiv$ (nodo = NULL) $\vee_{\text{L}}$
(menorEspecial?(nodo, (*nodo).hijoDer) $\wedge$ $\neg$menorEspecial?(nodo, (*nodo).hijoIzq) $\wedge$
ABBespecial?((*nodo).hijoIzq) $\wedge$ ABBespecial?((*nodo).hijoDer))

menor?(padre, hijo) $\equiv$ (hijo = NULL) $\vee_{\text{L}}$ padre.id < hijo.id)

menorEspecial?(padre, hijo) $\equiv$ (hijo = NULL) $\vee_{\text{L}}$ padre.cantPokemons < hijo.cantPokemons $\vee$
(padre.cantPokemons = hijo.cantPokemons $\wedge$ padre.id < hijo.id)

altoValido?(nodo) $\equiv$ nodo = NULL $\vee_{\text{L}}$
((*nodo).alto = mayorAltura((*nodo).hijoIzq, (*nodo).hijoDer) + 1 $\wedge$
factorDeBalanceo(cantidadHijos((*nodo).hijoIzq), cantidadHijos((*nodo).hijoDer)) $\leq$ 1

mayorAltura(izq, der) $\equiv$ **if** cantidadHijos(izq) < cantidadHijos(der) **then**
cantidadHijos(izq)
else
cantidadHijos(der)
fi

cantidadHijos(nodo) $\equiv$ **if** nodo = NULL **then**
0
else
β((*nodo).hijoIzq $\neq$ NULL) + cantidadHijos((*nodo).hijoIzq) +
β((*nodo).hijoDer $\neq$ NULL) + cantidadHijos((*nodo).hijoDer)
fi

factorDeBalanceo(izq, der) $\equiv$ **if** izq < der **then** der - izq **else** izq - der **fi**

definido?(j , p) $\equiv$ p $\neq$ NULL $\wedge_{\text{L}}$ (p.id = j $\vee$ definido?(j, p.hijoIzq) $\vee$ definido?(j, p.hijoDer))

obtener(j , p) $\equiv$ **if** p.id = j **then**
p.cantPokemons
else
if definido?(j, p.hijoIzq) **then** obtener(j, p.hijoIzq) **else** obtener(j, p.hijoDer) **fi**
fi

menor(p) $\equiv$ **if** p.hijoIzq = NULL **then** p **else** menor(p.hijoIzq) **fi**

Algoritmos

Algoritmos de DiccionarioPrioridad

iVacio () → res: dicPri	
res.raizJugadores ← NULL;	$O(1)$
res.raizCantidad ← NULL;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se inicializan los punteros como nulos.

iDefinido? (**in** p : `dicPri`, **in** j : `jugador`) → res: bool	
res ← BuscarJ(p.raizJugadores) ≠ NULL;	$O(log(\#j))$

Complejidad: $O(log(\#j))$
Justificación: buscar el elemento nos cuesta la altura del arbol, y por ser un AVL es $log(\#j)$

iEsVacio? (**in** p : `dicPri`) → res: bool	
res ← p.raizJugadores = NULL;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se ingresa a la raiz del arbol

iDefinir (**in/out** p : `dicPri`, **in** j : `jugador`, **in** n : `nat`)	
if *Definido?(p, j)* **then**	
Borrar(p, j);	$O(log(\#j))$
end if	
if *res.raizJugadores = NULL* **or** $n <$ *p.menor* **or** $(n =$ *p.menor* $\wedge j <$ *p.menorId)* **then**	
p.menor ← n;	$O(1)$
p.menorId ← j;	$O(1)$
end if	
p.raizJugadores ← InsertarJ(j, n, p.raizJugadores);	$O(log(\#j))$
p.raizCantidad ← InsertarN(j, n, p.raizCantidad);	$O(log(\#j))$

Complejidad: $O(log(\#j))$
Justificación: la complejidad de definir un elemento es. como máximo, recorrer todo el arbol, que por invariante de AVL sabemos que es $log(\#j)$

iBorrar (**in/out** p : `dicPri`, **in** j : `jugador`)	
puntero(Nodo): nodo ← BuscarJ(j, p.raizJugadores);	$O(log(\#j))$
p.raizCantidad ← RemoverN(j, (*nodo).cantPokemons, p.raizCantidad);	$O(log(\#j))$
p.raizJugadores ← RemoverJ(j, p.raizJugadores);	$O(log(\#j))$
if $j =$ *p.menorId* $\wedge$ *p.raizCantidad* $\neq$ *NULL* **then**	
puntero(Nodo): minimo ← BuscarMinimo(p.raizCantidad);	$O(log(\#j))$
p.menor ← (*minimo).cantPokemons;	$O(1)$
p.menorId ← (*minimo).id;	$O(1)$
end if	

Complejidad: $O(log(\#j))$
Justificación: la complejidad de buscar o eliminar un elemento es de orden de la altura del arbol, que por invariante de AVL sabemos que es $log(\#j)$

iMenor (**in** p : `dicPri`) → res: jugador	
res ← p.menorId;	$O(1)$

Complejidad: $O(1)$
Justificación: solo tenemos que acceder a la variable correspondiente

iClaves (**in** p : `dicPri`) → res: conj(Jugador)	
res ← Vacio();	$O(1)$
AgregarClaves(p.raizJugadores, res);	$O(\#j)$

Complejidad: $O(\#j)$

Justificación: el arbol no almacena las claves en una estructura separada, así que las mismas deben obtenerse manualmente recorriendo todo el arbol.

Algoritmos auxiliares

BuscarJ (**in** j : `jugador`, **in** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: `true`

Post: res es igual al nodo hijo o el que viene por parametro que tenga el id del jugador a menos que no este en cuyo caso en NULL

if *nodo = NULL* ***or*** *(*nodo).id = jugador* **then**	
res ← nodo;	$O(1)$
else if $j <$ *(*nodo).id* **then**	
res ← BuscarJ(j, (*nodo).hijoIzq);	$O(h)$
else	
res ← BuscarJ(j, (*nodo).hijoDer);	$O(h)$
end if	

Complejidad: $O(h)$

InsertarJ (**in** j : `jugador`, **in** n : `nat`, **in/out** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: `true`

Post: res es igual al mismo nodo que viene por parametro pero con el nuevo nodo(id:j y n) y balanceado

if *nodo = NULL* **then**	
res ← CrearNodo(j, n);	$O(1)$
else	
if $j <$ *n.id* **then**	
(*nodo).hijoIzq ← InsertarJ(j, n, (*nodo).hijoIzq);	$O(h)$
else	
(*nodo).hijoDer ← InsertarJ(j, n, (*nodo).hijoDer);	$O(h)$
end if	
res ← Balancear(nodo);	$O(1)$
end if	

Complejidad: $O(h)$

RemoverJ (**in** *j* : `jugador`, **in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: entre los nodo que desprenden del nodo del parametro se encuentra el id j.
Post: res es el mismo nodo pero ahora con el hijo removido en esa rama excepto que sea el nodo que hay que remover el cual se convierte en null

```
if nodo = NULL then
    res ← nodo;                                                          O(1)
else if j < (*nodo).id then
    (*nodo).hijoIzq ← RemoverJ(j, (*nodo).hijoIzq);                      O(h)
    res ← Balancear(nodo);                                               O(1)
else if j > (*nodo).id then
    (*nodo).hijoDer ← RemoverJ(j, (*nodo).hijoDer);                      O(h)
    res ← Balancear(nodo);                                               O(1)
else
    puntero(Nodo): i ← (*nodo).hijoIzq;                                  O(1)
    puntero(Nodo): d ← (*nodo).hijoDer;                                  O(1)
    if d = NULL then
        res ← i;                                                         O(1)
    else
        puntero(Nodo): minimo ← BuscarMinimo(d);                         O(h)
        minimo.hijoDer ← RemoverMinimo(d);                               O(h)
        minimo.hijoIzq ← i;                                              O(1)
        res ← Balancear(minimo);                                         O(1)
    end if
end if
```

Complejidad: $O(h)$

BuscarMinimo (**in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: true
Post: res es el puntero al minimo hijo del nodo(o el si es el menor)

```
if (*nodo).hijoIzq = NULL then
    res ← nodo;                                                          O(1)
else
    res ← BuscarMinimo((*nodo).hijoIzq);                                 O(h)
end if
```

RemoverMinimo (**in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: true
Post: res es el el mismo nodo pero con el nodo de mas a la izquierda eliminado(al padre del menor, se le pasan los hijos mayores si es que tiene)

```
if (*nodo).hijoIzq = NULL then
    res ← (*nodo).hijoDer;                                               O(1)
else
    (*nodo).hijoIzq = RemoverMinimo((*nodo).hijoIzq);                    O(h)
    res ← Balancear(nodo);                                               O(1)
end if
```

BuscarN (**in** *j* : `jugador`, **in** *n* : `nat` **in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: true
Post: res es el puntero al nodo con la clave j y el significado n si no lo encuetnra es null

```
if nodo = NULL or ((*nodo).id = jugador and (*nodo).cantPokemons = n) then
    res ← nodo;                                                              O(1)
else
    if n < (*nodo).cantPokemons or (n = (*nodo).cantPokemons and j < (*nodo).id) then
        res ← BuscarN(j, n, (*nodo).hijoIzq);                                O(h)
    else
        res ← BuscarN(j, n, (*nodo).hijoDer);                                O(h)
    end if
end if
```

InsertarN (**in** *j* : `jugador`, **in** *n* : `nat`, **in/out** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: true
Post: res es igual al mismo nodo que viene por parametro pero con el nuevo nodo(id:j y n) y balanceado

```
if nodo = NULL then
    res ← CrearNodo(j, n);                                                   O(1)
else
    if n < (*nodo).cantPokemons or (n = (*nodo).cantPokemons and j < (*nodo).id) then
        (*nodo).hijoIzq ← InsertarN(j, n, (*nodo).hijoIzq);                  O(h)
    else
        (*nodo).hijoDer ← InsertarN(j, n, (*nodo).hijoDer);                  O(h)
    end if
    res ← Balancear(nodo);                                                   O(1)
end if
```

RemoverN (**in** *j* : `jugador`, **in** *n* : `nat`, **in** *nodo* : `puntero(Nodo)`) → res: puntero(Nodo)

Pre: entre los nodo que desprenden del nodo del parametro se encuentra el id j con significado n.
Post: res es el mismo nodo pero ahora con el hijo removido en esa rama excepto que sea el nodo que hay que remover el cual se convierte en null

```
if nodo = NULL then
    res ← nodo;                                                              O(1)
else if n < (*nodo).cantPokemons or (n = (*nodo).cantPokemons and j < (*nodo).id) then
    (*nodo).hijoIzq ← RemoverN(j, n, (*nodo).hijoIzq);                       O(h)
    res ← Balancear(nodo);                                                   O(1)
else if n > (*nodo).cantPokemons or (n = (*nodo).cantPokemons and j > (*nodo).id) then
    (*nodo).hijoDer ← RemoverN(j, n, (*nodo).hijoDer);                       O(h)
    res ← Balancear(nodo);                                                   O(1)
else
    puntero(Nodo): i ← (*nodo).hijoIzq;                                      O(1)
    puntero(Nodo): d ← (*nodo).hijoDer;                                      O(1)
    if d = NULL then
        res ← i;                                                             O(1)
    else
        puntero(Nodo): minimo ← BuscarMinimo(d);                             O(h)
        minimo.hijoDer ← RemoverMinimo(d);                                   O(h)
        minimo.hijoIzq ← i;                                                  O(1)
        res ← Balancear(minimo);                                             O(1)
    end if
end if
```

Complejidad: $O(h)$, donde h es la altura del nodo

Justificación: como sabemos la altura de un nodo es la mayor altura entre sus hijos + 1, como en cada iteración estamos haciendo una llamada recursiva a uno de sus hijos hasta que ese hijo sea nulo, en el peor de los casos llamamos

recursivamente al hijo con mayor altura y en cada llamada estamos disminuyendo en 1 la altura.

La complejidad es equivalente a $log(n)$, donde n es la cantidad de hijos del nodo, ya que al estar ordenado como ABB y balanceado con invariante de AVL, cada llamada recursiva a cada subarbol reduce el tamaño de la entrada aproximadamente por la mitad.

AgregarClaves (**in** *nodo*: `Nodo`, **in/out** *c*: `conj(Jugador)`)

Pre: c es un subconjunto de las claves del arbol y nodo es un nodo del arbol
Post: c es igual a todas las claves del arbol

if *nodo* $\neq$ *NULL* **then**	
AgregarRapido(c, (*nodo).id);	$O(1)$
AgregarClaves((*nodo).hijoIzq, c);	$O(\#j)$
AgregarClaves((*nodo).hijoDer, c);	$O(\#j)$
end if	

Complejidad: $O(\#j)$
Justificación: el arbol no almacena las claves en una estructura separada, así que las mismas deben obtenerse manualmente recorriendo todo el arbol.

CrearNodo (**in** j: `jugador`, **in** n: `nat`) → res: puntero(Nodo)

Pre: `true`
Post: *res* apunta a un nodo de alto 1 con id j y cantPokemons n

Nodo: nuevo;	$O(1)$
nuevo.hijoIzq ← NULL;	$O(1)$
nuevo.hijoDer ← NULL;	$O(1)$
nuevo.alto ← 1;	$O(1)$
nuevo.id ← j;	$O(1)$
nuevo.cantPokemons ← n;	$O(1)$
res ← &nuevo;	$O(1)$

Complejidad: $O(1)$
Justificación: solo se crea e inicializa un nodo.

ArreglarAlto (**in/out** *nodo*: `puntero(Nodo)`)

Pre: la altura de los hijos de *nodo* es correcta
Post: la altura de *nodo* es igual a la altura máxima de sus hijos + 1

nat: alturaIzq ← Altura((*nodo).hijoIzq);	$O(1)$
nat: alturaDer ← Altura((*nodo).hijoDer);	$O(1)$
if *alturaIzq* $<$ *alturaDer* **then**	
(*nodo).altura ← alturaDer + 1;	$O(1)$
else	
(*nodo).altura ← alturaIzq + 1;	$O(1)$
end if	

Complejidad: $O(1)$
Justificación: solo se realizan comparaciones para las alturas de los hijos

Altura (**in** *nodo*: `puntero(Nodo)`) → res: nat

Pre: `true`
Post: *res* es 0 si *nodo* es nulo, o igual a la altura del nodo al que apunta en caso contrario

if *nodo* = *NULL* **then**	
res ← 0;	$O(1)$
else	
res ← (*nodo).alto;	$O(1)$
end if	

Complejidad: $O(1)$
Justificación: solo se verifica si es nulo o se lee un componente.

Balancear (**in/out** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: *nodo* ≠ NULL
Post: el módulo del factor de balanceo de *res* es menor a 2

ArreglarAlto(nodo);	$O(1)$
nat: alturaIzq ← Altura((*nodo).hijoIzq);	$O(1)$
nat: alturaDer ← Altura((*nodo).hijoDer);	$O(1)$
if *alturaDer* > *alturaIzq* ***and*** *alturaDer* - *alturaIzq* = *2* **then**	
if *Altura((*nodo).hijoDer.hijoIzq)* > *Altura((*nodo).hijoDer.hijoDer)* **then**	
(*nodo).hijoDer ← rotarDer((*nodo).hijoDer;	$O(1)$
end if	
res ← rotarIzq(nodo);	$O(1)$
else if *alturaIzq* > *alturaDer* ***and*** *alturaIzq* - *alturaDer* = *2* **then**	
if *Altura((*nodo).hijoIzq.hijoDer)* > *Altura((*nodo).hijoIzq.hijoIzq)* **then**	
(*nodo).hijoIzq ← rotarIzq((*nodo).hijoIzq);	$O(1)$
end if	
res ← rotarDer(nodo);	$O(1)$
else	
res ← nodo;	$O(1)$
end if	

rotarDer (**in** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: *nodo* tiene un hijo izquierdo
Post: *res* es el hijo izquierdo de nodo, y arbol se rota a la derecha

puntero(nodo) : aux ← (*nodo).hijoIzq;	$O(1)$
(*nodo).hijoIzq ← (*aux).hijoDer;	$O(1)$
(*aux).hijoDer ← nodo;	$O(1)$
ArreglarAlto(nodo);	$O(1)$
ArreglarAlto(aux);	$O(1)$
res ← aux;	$O(1)$

rotarIzq (**in** *nodo*: `puntero(Nodo)`) → res: puntero(Nodo)

Pre: *nodo* tiene un hijo derecho
Post: *res* es el hijo derecho de nodo, y arbol se rota a la izquierda

puntero(nodo) : aux ← (*nodo).hijoDer;	$O(1)$
(*nodo).hijoDer ← aux.hijoIzq;	$O(1)$
aux.hijoIzq ← nodo;	$O(1)$
ArreglarAlto(nodo);	$O(1)$
ArreglarAlto(aux);	$O(1)$
res ← aux;	$O(1)$

Complejidad: $O(1)$

Justificación: las rotaciones son una serie de comparasiones y asignaciones de punteros ($\Theta(1)$). El invariante de orden del arbol de búsqueda (ABB) se preserva luego de cada rotación. El invariante de balanceo de AVL se restaura para cada subarbol al final de `Balancear`, pero no luego de cada rotación individual (puede ser necesario rotar un subarbol de manera temporalmente imbalanceada para restaurar el balance de un arbol).

7. Consideraciones

- Se asume lógica de cortocircuito para todos los algoritmos.
- Algunas complejidades y sus justificaciones son compartidas (ya que la función es prácticamente idéntica, solo llama a una o dos auxiliares o solo realiza alguna otra acción en $\Theta(1)$), así que en algunos casos se omiten y se asume que se usa la próxima complejidad y justificación disponible
- La aridad de `EntrenadoresPosibles` se modifica ya que tomaba un conjunto de jugadores por parámetro. Se asume que el funcionamiento es igual a llamar a esa función con todos los jugadores válidos.
- Para ser consistente con el cambio a iteradores en la funcion `Jugadores`, también se devuelve un iterador en `Expulsados`.