Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2016

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

Trabajo Práctico 2

Diseño

Grupo: "ITerador, el payaso asesino"

Integrante	LU	Correo electrónico
Ocles Garcia, Nestor Dario	633/15	dario.ocles@gmail.com
Ansaldi, Nicolas	128/14	nansaldi611@gmail.com
Pawlow, Dante	449/12	dante.pawlow@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Módulos Simples

1.1 Pokemon

Servicios usados: String

Representación

1.1.1 Representacion

`Pokemon` **se representa con** `String`

1.2 Jugador

Servicios usados: Nat

Representación

1.2.1 Representacion

`Jugador` **se representa con** `Nat`

2 Coordenada

2.1 Interfaz

Género Coordenada
se explica con: NAT, BOOL

CrearCoordenada(**in** n : `nat`, **in** m : `nat`) → res : `coordenada`
Pre ≡ {true}
Post ≡ {res = crearCoor(n, m)}
Complejidad: O(1)
Descripción: Crea una nueva coordenada

DistEuclidea(**in** c_1 : `coordenada`, **in** c_2 : `coordenada`) → res : `nat`
Pre ≡ {true}
Post ≡ {res = distEuclidea(c_1, c_2)}
Complejidad: O(1)
Descripción: Devuelve la distancia entre 2 coordenadas

CoordenadaArriba(**in** c : `coordenada`) → res : `coordenada`
Pre ≡ {true}
Post ≡ {res = coordenadaArriba(c)}
Complejidad: O(1)
Descripción: Crea una coordenada arriba de la pasada por parámetro

CoordenadaAbajo(**in** c : `coordenada`) → res : `coordenada`
Pre ≡ {latitud(c)>0}
Post ≡ {res = coordenadaAbajo(c)}
Complejidad: O(1)
Descripción: Crea una coordenada abajo de la pasada por parámetro

CoordenadaIzquierda(**in** c : `coordenada`) → res : `coordenada`
Pre ≡ {longitud(c)>0}
Post ≡ {res = coordenadaIzquierda(c)}
Complejidad: O(1)
Descripción: Crea una coordenada a la izquierda de la pasada por parámetro

CoordenadaDerecha(**in** c : `coordenada`) → res : `coordenada`
Pre ≡ {true}
Post ≡ {res = coordenadaDerecha(c)}
Complejidad: O(1)
Descripción: Crea una coordenada a la derecha de la pasada por parámetro

TieneCoordenadaAbajo(**in** c : `coordenada`) → res : `bool`
Pre ≡ {true}
Post ≡ {res = latitud(c) > 0}
Complejidad: O(1)
Descripción: Dice si tiene coordenada abajo

TieneCoordenadaIzquierda(**in** c : `coordenada`) → res : `bool`
Pre ≡ {true}
Post ≡ {res = longitud(c) > 0}
Complejidad: O(1)
Descripción: Dice si tiene coordenada a la izquierda

Latitud(**in** c : `coordenada`) → res : `Nat`
Pre ≡ {true}
Post ≡ {res = latitud(c)}

Complejidad: O(1)
Descripción: Devuelve latitud de la coordenada

LONGITUD(**in** c: `coordenada`) → res : `Nat`
Pre ≡ {true}
Post ≡ {res = longitud(c)}
Complejidad: O(1)
Descripción: Devuelve longitud de la coordenada

Representación

2.1.1 Justificación

casillero representa una Coordenada. Y guardamos latitud y longitud de cada coordenada.

`Coordenada` **se representa con** `casillero`

donde `casillero` es `tupla`(*latitud*: `Nat` , *longitud*: `Nat`)

2.1.2 Invariante de representación

Informal

Vale para todo par de natruales
Formal

Rep : casilleroe $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$ true

2.1.3 Predicado de abtraccion

Abs : casillero e $\longrightarrow$ Coordenada {Rep(e)}
Abs(e) $\equiv$ ($\forall$ s:casillero)(Abs(s) $=_{obs}$ c:Coordenada) | (s.latitud = latitud(c) $\wedge$ s.longitud = longitud(c))

2.2 Algoritmos

Algoritmos

---

iCrearCoordenada(**in** n: `nat`, **in** m: `nat`) → res: casillero

1: $res \leftarrow < n, m >$ ▷ O(1)

Complejidad: O(1)
Justificacion: Sólo realiza una asignación

---

iDistEuclidea(**in** c_1: `casillero`, **in** c_2: `casillero`) → res: nat

1: $res \leftarrow ((c_1.latitud - c_2.latitud)^2 + (c_1.longitud - c_2.longitud)^2))$ ▷ O(1)

Complejidad: O(1)
Justificacion: Sólo realiza operaciones básicas

---

iCoordenadaArriba(**in** c: `casillero`) → res: casillero

1: $res \leftarrow < c.latitud + 1, c.longitud >$ ▷ O(1)

Complejidad: O(1)
Justificacion: Sólo realiza una asignación y una suma

iCoordenadaAbajo(**in** c: `casillero`) → res: casillero

1: $res \leftarrow < c.latitud - 1, c.longitud >$ ▷ O(1)

Complejidad: O(1)
Justificacion: Sólo realiza una asignación y una resta

iCoordenadaIzquierda(**in** c: `casillero`) → res: casillero

1: $res \leftarrow < c.latitud, c.longitud - 1 >$ ▷ O(1)

Complejidad: O(1)
Justificacion: Sólo realiza una asignación y una resta

iCoordenadaDerecha(**in** c: `casillero`) → res: casillero

1: $res \leftarrow < c.latitud, c.longitud + 1 >$ ▷ O(1)

Complejidad: O(1)
Justificacion: Sólo realiza una asignación y una suma

iTieneCoordenadaAbajo(**in** c: `casillero`) → res: bool

1: $res \leftarrow c.latitud > 0$ ▷ O(1)

Complejidad: O(1)
Justificacion: Si latitud es mayor a 0 tiene coordenada abajo

iTieneCoordenadaIzquierda(**in** c: `casillero`) → res: bool

1: $res \leftarrow c.longitud > 0$ ▷ O(1)

Complejidad: O(1)
Justificacion: Si longitud es mayor a 0 tiene coordenada izquierda

iLatitud(**in** c: `casillero`) → res: Nat

1: $res \leftarrow c.latitud$ ▷ O(1)

Complejidad: O(1)
Justificacion: Solo hacemos una asignación de un valor guardado en la estructura interna

iLongitud(**in** c: `casillero`) → res: Nat

1: $res \leftarrow c.longitud$ ▷ O(1)

Complejidad: O(1)
Justificacion: Solo hacemos una asignación de un valor guardado en la estructura interna

3 Mapa

3.1 Interfaz

Género mapa
se explica con: CONJ(σ), BOOL, COORDENADA

CREARMAPA() $\rightarrow$ *res* : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ {res = crearMapa()}
Complejidad: O(1)
Descripción: Crea un nuevo mapa

COORDENADAS(**in** *m*: `mapa`) $\rightarrow$ *res* : `Conj(Coordenada)`
Pre $\equiv$ {true}
Post $\equiv$ {res = coordenadas(m)}
Complejidad: O(1)
Descripción: Devuelve todas las coordenadas del mapa

AGREGARCOORDENADA(**in** *c*: `Coordenada`, **in/out** *m*: `mapa`)
Pre $\equiv$ {m $=_{obs}$ m_0}
Post $\equiv$ {m = agregarCoor(c, m_0)}
Complejidad: O(max(n^3, T^2))
Descripción: Agrega la coordenada a mapa. Donde T es el tamaño de la grilla de todo el mapa (ancho * alto) y n es el cardinal de coordenadas en el Mapa. Donde n representa la cantidad de coordenadas en el mapa

HAYCAMINO(**in** c_1: `Coordenada`, **in** c_2: `Coordenada`, **in** *m*: `mapa`) $\rightarrow$ *res* : `Bool`
Pre $\equiv$ {$c_1 \in coordenadas(m) \wedge c_2 \in coordenadas(m)$}
Post $\equiv$ {res $=_{obs}$ hayCamino(c_1, c_2)}
Complejidad: O(1)
Descripción: Te dice si dos coordenadas estan conectadas

POSEXISTENTE(**in** *c*: `Coordenada`,**in** *m*: `mapa`) $\rightarrow$ *res* : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ posExistente(c, m)}
Complejidad: O(1)
Descripción: Devuelve true si existe esa coordenada

Representación

3.1.1 Justificación

infomapa representa el tad Mapa. La componente coordenadas representa todas las coordenadas que fueron agregadas al mapa. relacionCoordenadas representa que coordenadas estan conectadas con cuales. Es una matriz donde la primer dimension representa cada coordenada y la segunda dimensión representa las coordenadas con las que están relacionadas. La idea es guardar en la intercepción entre dos coordenadas (cada una en una dimensión distinta) un True en caso si están relacionadas. Esta matriz nos permite verificar en O(1) si dos coordenadas estan relacionadas o no. ancho y alto representan el tamaño del mapa, para calcularlo buscamos la coordenada con la longitud/latitud mayor y lo guardamos. Fue necesario guardar estos datos ya que los necesitamos para calcular la posición de cada coordenada en relacionCoordenadas en cada dimensión.

`Mapa` **se representa con** `infomapa`

donde `infomapa` es `tupla`(*coordenadas*: `conj(Coordenada)`
, *relacionCoordenadas*: `arreglo(arreglo(Bool))` , *ancho*: `Nat` , *alto*: `Nat`)

3.1.2 Invariante de representación

Informal
(1)Existe una coordenada en coordenadas tal que, dicha coordenada toma como su latitud al alto, y Existe una coordenada en coordenadas que su longitud es el ancho del mapa (puede ser la misma coordenada)
(2)relacionCoordenadas tiene como dimension el ancho*alto de alto y tambien de largo, ademas para toda celda de esta matriz se tiene la relacion de camino entre 2 coordenadas, dichas coordenadas tienen que existir en coordenadas del mapa

Formal

Rep : infoMapa M $\longrightarrow$ bool
Rep(M) $\equiv$ true $\Longleftrightarrow$ (1)($\exists$ c_1 :Coordenada)($c_1 \in$ M.coordenadas) $\Rightarrow$ latitud(c_1) = M.alto $\wedge$ ($\exists$ c_2 :Coordenada)($c_2 \in$ M.coordenadas) $\Rightarrow$ longitud(c_2) = M.ancho $\wedge$ (2) ($forall$ i, j: nat)(Definido?(M.coordenadas, [i, j]))$\Rightarrow$ ($forall$ k, l: nat)(k <i $\wedge$ l <j)$\Rightarrow$ Definido?(M.coordenadas, [k, l]) $\wedge$ ($\forall$ i, j: nat)(Definido?(M.relacionCoordenadas, [i, j])) $\Rightarrow$ <i, j> $\in$ M.coordenadas

3.1.3 Predicado de abtraccion

Abs : infoMapa M $\longrightarrow$ Mapa {Rep(M)}
Abs(M) $\equiv$ mapa :Mapa | M.coordenadas = Coordenadas(mapa)

3.2 Algoritmos

No exportable, operaciones auxiliares

CoordenadasConectadasA(**in** c: `Coordenada`, **in** m: `mapa`) $\rightarrow$ res : `Conj(Coordenada)`
Pre $\equiv$ {$c \in coordenadas(m)$}
Post $\equiv$ {$(\forall c_1 : Coordenada) c_1 \in res \wedge c_1 \in coordenadas(m) \Rightarrow_{\text{L}} hayCamino(c, c_1, m)$}
Complejidad: O(n^2)
Descripción: Devuelve un conjunto de coordenadas a las cuales hayCamino

Algoritmos

---

iCrearMapa() $\rightarrow$ res: mapainfo
1: $res \leftarrow < Vacio(), arreglo[0], 0, 0 >$ ▷ O(1)

Complejidad: O(1)
Justificacion: Sólo realiza una asignación y las funciones de Vacío() de módulo Conjunto Lineal y Diccionario Lineal son O(1)

---

iHayCamino(**in** $c1$: `coordenada`, **in** $c2$: `coordenada`, **in** m: `infomapa`) $\rightarrow$ res: bool
1: pos1 $\leftarrow$ m.ancho * Longitud(c1) + m.alto * Latitud(c1) ▷ O(1)
2: pos2 $\leftarrow$ m.ancho * Longitud(c2) + m.alto * Latitud(c2) ▷ O(1)
3: res $\leftarrow$ m.relacionCoordenadas[pos1][pos2] ▷ O(1)

Complejidad: O(1)
Justificacion: Son solamente 3 asignaciones y un acceso de orden de 1 en un arreglo estatico

---

iPosExistente(**in** c: `coordenada`, **in** m: `infomapa`) → res: bool

```
1: res ← False                                                          ▷ O(1)
2: if Latitud(c) < m.alto ∧ Longitud(c) < m.ancho then                  ▷ O(1)
3:     pos ← m.ancho * Longitud(c) + m.alto * Latitud(c)                ▷ O(1)
4:     res ← m.relacionCoordenadas[pos][pos] == True                    ▷ O(1)
5: else
6: end if
```

Complejidad: O(1)
Justificacion: Son 3 asignaciones y un acceso de orden 1 a un arreglo. Esto funciona porque cuando calculo las relaciones entre las coordenas siempre definimos que una coordenada esta relacionada consigo misma.

iAgregarCoordenada(**in** c: `coordenada`, **in** m: `infomapa`)

```
1: if Longitud(c) > infomapa.ancho then m.ancho ← Longitud(c) else fi                ▷ O(1)
2: if Latitud(c) > infomapa.alto then m.alto ← Latitud(c) else fi                    ▷ O(1)
3: Agregar(m.coordenadas, c)                                                         ▷ O(#m.coordenadas)
4: m.relacionCoordenadas ← arreglo[m.ancho*m.alto] de arreglo[m.ancho*m.alto] de Bool ▷ O((m.ancho * m.alto)^2)
5: iter ← CrearIt(m.coordenadas)                                                     ▷ O(1)
6: while HaySiguiente(iter) do                                                       ▷ O(#m.coordenadas^3)
7:     coor ← Siguiente(iter)                                                        ▷ O(1)
8:     Avanzar(iter)                                                                 ▷ O(1)
9:     conectadas ← iCoordenadasConectadas(coor, m)                                  ▷ O(#m.coordenadas^2)
10:    iterConectadas ← CrearIt(conectadas)                                          ▷ O(1)
11:    while HaySiguiente(iterConectadas) do                                         ▷ O(#m.coordenadas)
12:        coor_2 ← Siguiente(iterConectadas)                                        ▷ (1)
13:        Avanzar(iterConectadas)                                                   ▷ O(1)
14:        pos1 ← m.ancho * Longitud(coor) + m.alto * Altitud(coor)                  ▷ O(1)
15:        pos2 ← m.ancho * Longitud(coor_2) + m.alto * Altitud(coor_2)              ▷ O(1)
16:        m.relacionCoordenadas[pos1][pos2] ← True                                  ▷ O(1)
17:        m.relacionCoordenadas[pos2][pos1] ← True                                  ▷ O(1)
18:    end while
19: end while
```

Complejidad: O(max(n^3, T^2))
Justificacion: Donde T es el tamaño de la grilla de todo el mapa (ancho * alto) y n es el cardinal de coordenadas en el Mapa. Ya que la creación de los arreglos no es gratis, tiene un costo que es el tamaño del ancho*alto del Mapa. También ejecutamos un While de n iteraciones donde ejecutamos operaciones que cuestan como máximo n^2 por lo cual el While tiene un costo del orden de n^3. Dado que la creación podría tomar más tiempo que ejecutar el While debemos tomar el máximo valor de ambos como la complejidad del algoritmo.

iCoordenadasConectadasA(**in** c: `coordenada`, **in** m: `infomapa`) → res: Conj(coordenada)

```
1:  visitadas ← Vacío()                                                          ▷ O(1)
2:  aVisitar ← Encolar(Vacía(), c)                                               ▷ O(1)
3:  res ← Agregar(Vacío(), c)                                                    ▷ O(1)
4:  while ¬ EsVacía(aVisitar) do                                                 ▷ O(#m.coordenadas^2)
5:      coor ← Proximo(aVisitar)                                                 ▷ O(1)
6:      Desencolar(aVisitar)
7:      Agregar(visitadas, coor)                                                 ▷ O(#m.coordenadas)
8:      if Latitud(coor) > 0 then                                                ▷ O(1)
9:          coorAbajo ← CoordenadaAbajo(coor)                                    ▷ O(1)
10:         if ¬ Pertenece?(visitadas, coorAbajo) ∧ Pertenece?(m.coordenadas, coorAbajo) then ▷
    O(#m.coordenadas)
11:             Agregar(res, coorAbajo)                                          ▷ O(#m.coordenadas)
12:             Encolar(aVisitar, coorAbajo)                                     ▷ O(copy(coordenada))
13:         else
14:         end if
15:     else
16:     end if
17:     if longitud(coor) > 0 then                                               ▷ O(1)
18:         coorIzq ← CoordenadaIzquierda(coor)                                  ▷ O(1)
19:         if ¬ Pertenece?(visitadas, coorIzq) ∧ Pertenece?(m.coordenadas, coorIzq) then ▷ O(#m.coordenadas)
20:             Agregar(res, coorIzq)                                            ▷ O(#m.coordenadas)
21:             Encolar(aVisitar, coorIzq)                                       ▷ O(copy(coordenada))
22:         else
23:         end if
24:     else
25:     end if
26:     coorDer ← CoordenadaDerecha(coor)                                        ▷ O(1)
27:     if ¬ Pertenece?(visitadas, coorDer) ∧ Pertenece?(m.coordenadas, coorDer) then ▷ O(#m.coordenadas)
28:         Agregar(res, coorDer)                                                ▷ O(#m.coordenadas)
29:         Encolar(aVisitar, coorDer)                                           ▷ O(copy(coordenada))
30:     else
31:     end if
32:     coorArriba ← CoordenadaDerecha(coor)                                     ▷ O(1)
33:     if ¬ Pertenece?(visitadas, coorArriba) ∧ Pertenece?(m.coordenadas, coorArriba) then ▷ O(#m.coordenadas)
34:         Agregar(res, coorArriba)                                             ▷ O(#m.coordenadas)
35:         Encolar(aVisitar, coorArriba)                                        ▷ O(copy(coordenada))
36:     else
37:     end if
38: end while
```

Complejidad: $O(n^2)$
Justificacion: Dado un mapa y una coordenada te devuelve todas las coordenadas conectadas a esa coordenada inicial. Tomando como **n** el cardinal de infomapa.coordenadas nos da $O(n^2)$.

4 Juego

4.1 Interfaz

Género juego, itJugadores, itPokemon
se explica con: Mapa, Conjunto (σ) , Secuencia(σ), Bool, Pokemon, Jugador, Nat

Operaciones básicas

CrearJuego(**in** m: `mapa`) $\rightarrow$ res : `juego`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ crearJuego(m)}
Complejidad: O(TM), donde TM es el tamaño del mapa (ancho*alto)
Descripción: Creo un nuevo juego tomando un mapa

AgregarPokemon(**in** p: `pokemon`, **in** c: `coordenada`, **in/out** g: `juego`)
Pre $\equiv$ {puedoAgregarPokemon(c, j) $\wedge$ g $=_{obs}$ g_0}
Post $\equiv$ {g = agregarPokemon(p, c, g_0)}
Complejidad: O(|P| + EC*log(EC)), siendo |P| es el nombre más largo para un pokemon y EC es la máxima cantidad de jugadores esperando capturar un pokemon
Descripción: Agrego un pokemon al juego

AgregarJugador(**in** g: `juego`) $\rightarrow$ res : `it(jugador)`
Pre $\equiv$ {g $=_{obs}$ g_0}
Post $\equiv$ {g = agregarJugador(g_0)}
Complejidad: O(J), Siendo J la cantidad de jugadores que fueron agregados al juego
Descripción: Agrega un jugador al juego

Conectarse(**in** j: `jugador`, **in** c: `coordenada`, **in/out** g: `juego`)
Pre $\equiv$ {g $=_{obs}$ g_0 $\wedge$ j $\in$ jugadores(g) $\wedge_{\text{L}}$ $\neg$estaConectado(j, g) $\wedge$ posExistente(c, mapa(g))}
Post $\equiv$ {g = conectarse(g_0)}
Complejidad: O(log(EC)), siendo EC la máxima cantidad de jugadoes esperando capturar un pokémon
Descripción: Conecta un jugador al juego, con todo lo que esto implica

Desconectarse(**in** j: `jugador`, **in/out** g: `juego`)
Pre $\equiv$ {g $=_{obs}$ g_0 $\wedge$ j $\in$ jugadores(g) $\wedge_{\text{L}}$ estaConectado(j, g)}
Post $\equiv$ {g = desconectarse(j, g_0)}
Complejidad: O(log(EC)), siendo EC la máxima cantidad de jugadoes esperando capturar un pokémon
Descripción: Desconecta al jugador j del juego

Moverse(**in** j: `jugador`, **in** c: `coordenada`, **in/out** g: `juego`)
Pre $\equiv$ {g $=_{obs}$ g_0 $\wedge$ j $\in$ jugadores(g) $\wedge_{\text{L}}$ estaConectado(j, g) $\wedge$ posExistente(c, mapa(g))}
Post $\equiv$ {g = moverse(j, c, g_0)}
Complejidad: O((PS + PC)*|P|+log(EC)), siendo PS la cantidad de pokemons salvajes, PC la máxima cantidad de pokemon capturados por un jugador
Descripción: Mueve un jugador en el mapa, verifica si hay una captura de pokémon, y para el jugador movido verifica si cometió alguna infracción

Mapa(**in** g: `juego`) $\rightarrow$ res : `Mapa`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ mapa(g)}
Complejidad: O(1)
Descripción: Devuelve la instancia de mapa que tenemos guardada

Jugadores(**in** g: `juego`) $\rightarrow$ res : `Conj(Jugador)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ jugadores(g)}
Complejidad: O(1)

Descripción: Devuelve la instancia de mapa que tenemos guardada

EstaConectado(**in** j : `Jugador`, **in** g : `juego`) → res : `Bool`
Pre ≡ {j ∈ jugadores(g)}
Post ≡ {res =$_{obs}$ estaConectado(g)}
Complejidad: O(1)
Descripción: Dice si un jugador esta conectado o no

Sanciones(**in** j : `Jugador`, **in** g : `juego`) → res : `Nat`
Pre ≡ {j ∈ jugadores(g)}
Post ≡ {res =$_{obs}$ sanciones(g)}
Complejidad: O(1)
Descripción: La cantidad de sanciones que tiene un jugador

Posicion(**in** j : `Jugador`, **in** g : `juego`) → res : `Coordenada`
Pre ≡ {j ∈ jugadores(g) ⇒$_{L}$ estaConectado(j, g)}
Post ≡ {res =$_{obs}$ posicion(j, g)}
Complejidad: O(1)
Descripción: Posición actual del jugador cuando se encuentra conectado

Pokemons(**in** j : `Jugador`, **in** g : `juego`) → res : `ItPokemon`
Pre ≡ {j ∈ jugadores(g)}
Post ≡ {res =$_{obs}$ CrearIt(deMulticonjAConj(pokémons(j, g)))}
Complejidad: O(1)
Descripción: Devuelve un iterador <Pokemon, Nat>
Aliasing: Crea un iterador a la primera posicion del conjunto, el iterador es no modificable por ende si se borra en el conjunto se invalida.

deMulticonjAConj : multiconj(pokemon) ⟶ conj(<pokemon, nat>)
deMulticonjAConj(mp) ≡ **if** ∅(mp) **then**
∅
else
Ag(<DameUno(mp), #(DameUno(mp), mp)>, deMulticonjAConj(SinUno(mp)))
fi

Expulsados(**in** g : `juego`) → res : `Conj(Jugador)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ expulsados(g)}
Complejidad: O(J)
Descripción: Conjunto de jugadores expulsados del juego

PosConPokemons(**in** g : `juego`) → res : `Conj(coor)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ posConPokémons(g)}
Complejidad: O(1)
Descripción: Conjunto de coordenadas con Pokémones

PokemonEnPos(**in** c : `Coordenada`, **in** g : `juego`) → res : `Pokemon`
Pre ≡ {c ∈ posConPokemons(g)}
Post ≡ {res =$_{obs}$ pokemonEnPos(c, g)}
Complejidad: O(1)
Descripción: Devolvemos el Pokemon en la Coordenada

CantMovimientosParaCaptura(**in** c : `Coordenada`, **in** g : `juego`) → res : `Nat`
Pre ≡ {c ∈ posConPokemons(g)}
Post ≡ {res =$_{obs}$ cantMovimientosParaCaptura(c, g)}
Complejidad: O(1)
Descripción: Cantidad de movimientos restantes para que un Pokemon sea capturado

JugadoresConectados(**in** g: `juego`) → res : `Conj(Jugador)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ jugadoresConectados(g)}
Complejidad: O(J)
Descripción: Conjunto de jugadores conectados

PuedoAgregarPokemon(**in** c: `Coordenada`, **in** g: `juego`) → res : `Bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ puedoAgregarPokemon(g)}
Complejidad: O(PS), donde PS es la cantidad de pokemons salvajes
Descripción: Devuelve True si se puede agregar un Pokemon en la coordenada

HayPokemonCercano(**in** c: `Coordenada`, **in** g: `juego`) → res : `Bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ hayPokemonCercano(c, g)}
Complejidad: O(1)
Descripción: Devuelve True sí y solo sí hay un Pokemon a radio 4 de la Coordenada

IndiceRareza(**in** p: `Pokemon`, **in** g: `Juego`) → res : `nat`
Pre ≡ {p ∈ pokemons}
Post ≡ {res =$_{obs}$ indiceRareza(p, g)}
Complejidad: O(|s|)
Descripción: Devuelve el índice de rareza del pokemon

CantPokemonsTotales(**in** g: `Juego`) → res : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ cantPokemonsTotales(g)}
Complejidad: O(1)
Descripción: Devuelve la cantidad de pokemons existentes, tanto salvajes como capturados.

PosPokemonCercano(**in** c: `Coordenada`, **in** g: `Juego`) → res : `Coordenada`
Pre ≡ {hayPokemonCercano(c,g)}
Post ≡ {res =$_{obs}$ posPokemonCercano(c, g)}
Complejidad: O(1)
Descripción: Devuelve True sí y solo sí hay un Pokemon a radio 4 de la Coordenada

Operaciones del iterador Jugador

CrearIt(**in** g: `Juego`) → res : `itJugador`
Pre ≡ {true}
Post ≡ {res = CrearItUni(v)}
Complejidad: O(1)
Descripción: Crea un iterador unidireccional no modificable al principio del vector, no necesariamente es un elemento válido por ende no se puede usar Actual sin Avanzar

HayMas?(**in** it: `itJugador`) → res : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ HayMas?(it)}
Complejidad: O(n), siendo n la cantidad de elementos del vector
Descripción: Devuelve true si y solo si quedan elementos para avanzar

Avanzar(**in/out** it: `itJugador`)
Pre ≡ {HayMas?(it) ∧ it =$_{obs}$ it$_0$}
Post ≡ {it =$_{obs}$ Avanzar(it$_0$)}
Complejidad: O(n), siendo n la cantidad de elementos del vector
Descripción: Avanza el iterador al próximo elemento del vector

Actual(**in** it: `itJugador`) → res : `Nat`

Pre ≡ {HayMas?(it)}
Post ≡ {res =$_{obs}$ Actual(it)}
Complejidad: O(1)
Descripción: Devuelve el Id apuntado por el iterador
Aliasing: res no es modificable porque el iterador no es modificable

SIGUIENTES(**in** *it*: `itJugador`)) → *res* : `lista(`σ`)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ Siguientes(it)}
Complejidad: O(n), siendo n la cantidad de elementos del vector
Descripción: Devuelve los elementos del vector posteriores al iterador, puede no haber ninguno

Operaciones del iterador Pokemons

CREARIT(**in** *g*: `Juego`, **in** *j*: `jugador`) → *res* : `itPokemon`
Pre ≡ {j ∈ jugadores(g)}
Post ≡ {res =$_{obs}$ CrearItUni(g)}
Complejidad: O(1)
Descripción: Crea un iterador no modificable al principio de los pokemons del jugador j
Aliasing: Como es un iterador no modificable puede invalidarse si se borra en la estructura

ACTUAL(**in** *it*: `itPokemon`) → *res* : `<Pokemon, Cantidad>`
Pre ≡ {HayMas?(it)}
Post ≡ {res =$_{obs}$ Actual(it)}
Complejidad: O(|P|), siendo P la clave mas larga del diccionario
Descripción: Devuelve una tupla con un pokemon del jugador y su cantidad

HAYMAS?(**in** *it*: `itPokemon`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ HayMas?(it)}
Complejidad: O(1)
Descripción: Chequea si hay mas elementos para recorrer

AVANZAR(**in/out** *it*: `itPokemon`)
Pre ≡ {it =$_{obs}$ it$_0$ ∧ HayMas?(it)}
Post ≡ {res =$_{obs}$ Avanzar(it)}
Complejidad: O(1)
Descripción: Avanza el iterador a la siguiente clave del diccionario

SIGUIENTES(**in** *it*: `itPokemon`) → *res* : `lista(`σ`)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ Siguientes(it)}
Complejidad: O(n), siendo n la cantidad de claves del diccionario
Descripción: Devuelve una lista con las claves que aun no se recorrierron

Representación del modulo

4.1.1 Justificación

Game representa un Juego.
En pokemons guardamos un diccionario sobre Trie y la clave es el Pokemon (string). Esto nos permite encontrar un pokemon en |P|. En el diccionario guardamos la cantidad de pokemones salvajes que hay y cuantos hubieron de ese tipo en total para poder calcular la rareza del pokemon en O(|P|) junto con cantidadTotPokemons que es el total de los Pokemons en el juego. En mapa guardamos el mapa con el que se crea el Juego.
coordenadasConPokemons guardamos un conjunto de Coordenadas donde hay pokemones para poder devolver en

posConPokémons en O(1). En jugadores de game guardamos un vector de InfoJugador (ver más abajo). Como en InfoCoordenada guardamos una lista de jugadores en cada coordenada y cuando expulsamos un jugador debemos borrar de forma eficiente en esa lista guardamos un iterador al jugador en diccionario para poder cumplir con los ordenes de mover.
mapainfo es una matriz de InfoCoordenadas donde cada dimensión es la latitud y longitud de una coordenada.
cantidadTotPokemones guardamos el total de pokemones del juego.
coordenaadasPokemons es un diccionario <Coordenada, Pokemon> donde guardamos el Pokemon que cada coordenada del Mapa tiene.
pokemonsDeJugadores es una lista donde guardamos los Pokemones que atrapó cada jugador. (**) Esto lo tuvimos que guardar fuera de InfoJugador porque teníamos un problema de complejidad al agregar un jugador nuevo. Como usabamos un Vector de jugadores el agregar es O(J + copy(α)) y guardar los Pokemones que atrapo el jugador hacía que el copy no fuera O(1), entonces guardamos un iterador a esta lista en InfoJugador.
InfoCoordenada representa la información perteneciente a cada coordenada en el Mapa. Guardamos si en esta posición hay pokemon y que pokemon, si ya se capturo un pokemon en esta coordenada que lo usamos para saber si quedo un heap no valido. En jugEspe usamos una colaDePrioriada(σ), donde σ es tupla <cantPokemons, Jugador> ordenamos por la primera parte de la tupla de forma que la tupla mas prioritaria sea la del jugador que menor cantidad de pokemons tenga, sirve para cuando se capture un Pokemon poder seleccionar el jugador de forma eficiente. Era necesario esta complejidad para poder cumplir la complejidad de mover. En MovimientosRestantes guardamos la cantidad de movimientos restantes para atrapar el Pokemon.
InfoJugador representa la información de cada jugador. Aquí guardamos al jugador (para no depender solamente del la posición del vector que algunas veces lo recorremos pero no lo tenemos), si esta conectado, expulsado, sanciones, en pos la posición actual, pokemons atrapados, la posición en el mapa, cantTotalPoke es cantidad total de pokemones que atrapo. En pokemons guardamos un iterador de lista que se guarda en Game, se explicó en (**). En posicionMapa guardamos un iterador a la cola de prioridad (heap) de la coordenada en caso de estar esperando para atrapar un pokemon, esto lo hicimos para poder eliminarlo de forma fácil del heap (cuando se mueve, se elimina, etc) y poder cumplir las complejidades de mover.

`Juego` **se representa con** `Game`

donde `Game` es `tupla`(*pokemons*: `diccString(pokemon, tupla <cantSalvaje: Nat, cantTotal: Nat>)` ,
mapa: `Mapa` ,
jugadores: `Vector(InfoJugador)` ,
mapaInfo: `Arreglo de Arreglo de InfoCoordenada` ,
cantidadTotPokémons: `Nat` ,
coordenadasConPokemons: `Conj(Coordenada)` ,
pokemonsDeJugadores: `Lista(DiccString(pokemon: string, cant: nat))`)

donde `InfoJugador` es `tupla`(*jug*: `jugador` ,
conectado: `Bool` ,
expulsado: `Bool` ,
sanciones: `Nat` ,
pos: `Coordenada` ,
pokemons: `itLista(Dicc(pokemon: string, cant: nat))` ,
posicionMapa: `itDicc(jugador: Nat, EsperandoCapturar: itColaDePrioridad(T, menorT))` ,
cantTotalPoke: `Nat`)

donde `InfoCoordenada` es `tupla`(*pokemon*: `Pokemon` ,
jugEspe: `ColaDePrioridad(T, menorT)` ,
hayPokemon: `Bool` ,
yaSeCapturo: `Bool` ,
jugadores: `Dicc(jugador: nat, EsperandoCapturar: itColaDePrioridad(T, menorT))` ,
MovimientosRestantes: `Nat`)

donde `T` es `tupla`(*pokemon*: `Pokemon` ,
cantidadPokemonesCapturados: `Nat`)

Donde **menorT** es una función:
$menorT(T_1, T_2) \rightarrow Bool$
$\pi_2(T_1) < \pi_2(T_2)$.

4.1.2 Invariante de representación

Informal

(1)Para todos los pokemons de un jugador dado, estos existen en el diccionario pokemons del juego y ademas la cantidad de los mismos es menor o igual a la cantidad total definida en el diccionario de pokemon del juego

(2)CantidadTotalPokemons del juego es igual a la suma de la cantTotal de cada clave del dicc Pokemons

(3)El tamaño de mapaInfo es igual al del mapa, y tienen las mismcas coordenadas

(4)Para toda coordenada en coordenadaConPokemon, existe un pokemon cuya cantSalvaje es 1. Dicho de otra manera la suma de la cantidad de coordenadas es igual a la suma de pokemons libres del dicc Pokemons del juego, para toda coordenada del conj existe una relacion con un pokemon del dicc

(5)Toda coordenada donde esta un jugador (infoJugador.pos) esa coordenada existe en mapa y en mapaInfo

(6)La parte de la tupla jug en InfoJugador corresponde con el índice del vector jugadores por el cual accedo a esa tupla

(7)Para todo elemento de la colaDePrioridad de infoCoordenada, para todo nodo de la cola existe un elemento en el dicc Jugadores de la misma estructura o alguna otra coordenada dentro del radio de captura. Además para cada elemento en el dicc jugadores de infoCoordenada, para cualquier coordenada, existe un elemento en el vector Jugadores (jugador no expulsado), los jugadores en la cola estan ordenados de forma creciente por la cantidad de pokemons que tienen, es decir que el jugador que esta primero en la cola es el jugador que menos pokemons tiene de toda la cola. Esos pokemons se corresponden con la cantTotal de pokemons que tiene dicho jugador en infoJugador.

(8)Para toda infoCoordenada el pokemon existe en el diccPokemon del juego y ademas la coordenada por la cual se filtra el arreglo de arreglos mapaInfo esta en coordenadaConPokemons

(9)Para todo elemento de la lista de pokemons de jugadores existe un jugador (no expulsado) para el cual se corresponde, ademas la suma de la cant de todas las claves del diccionario es igual a CantTotalPoke de infoJugador para ese jugador

Formal

Rep : Game G $\longrightarrow$ bool

Rep(G) $\equiv$ true $\iff$ (1)((∀ j: Jugador)(0 $\leq$ j $\leq$ Longitud(G.jugadores)) $\Rightarrow_L$ (∀ p: Pokemons)(Definido?(p, Siguiente(G.jugadores[j].pokemons))) $\Rightarrow$ Definido?(p, G.pokemons) $\wedge_L$ Significado(p, Siguiente(G.jugadores[j].pokemons)) $\leq$ Significado(p, G.pokemons).cantTotal)∧ (2) (SumarDicc(claves(G.pokemons), G.pokemons) = G.cantidadTotPokemons) ∧ (3) ((∀ i, j :nat)(Definido?(G.mapaInfo, [i, j])) $\Rightarrow$ (∃ c :Coordenada)(c $\in$ Coordenadas(G.mapa)) $\Rightarrow$ i = Latitud(c) ∧ j = Longitud(c)) ∧ (4) (#(G.coordenadaConPokemons) = (pokemonsSalvajes(claves(G.pokemons), G.pokemons))) ∧ (5) ((∀ i :Jugador)(0 $\leq$ j $<$ Longitud(G.jugadores)) $\Rightarrow_L$ PosExistente(G.jugadores[i].pos, G.mapa) ∧ Definido?(G.mapaInfo, G.jugadores[i].pos)) ∧ (6) ((∀ i :Jugador)(0 $\leq$ j $<$ Longitud(G.jugadores)) $\Rightarrow_L$ i = G.jugadores[i].jug) ∧ (7) (∀ c: Coordenada)(c $\in$ G.coordenadasConPokemons) $\Rightarrow_L$ (∀ x: G.mapaInfo[c].jugEspe)(∃ j:jugadores)(0 $\neq$ j $<$ Longitud(G.jugadores) $\wedge_L$ distEuclidea(c, G.jugadores[j].pos) $<$ 25 ∧ hayCamino(c, G.jugadores[j].pos)) $\Rightarrow_L$ x = <G.jugadores[j].cantTotalPoke, j> ∧ (8) ((∀ c: Coordenada)(Definido?(c, G.mapaInfo)) $\Rightarrow_L$ G.mapaInfo[c].hayPokemon $\Rightarrow_L$ Definido(G.mapaInfo[c].pokemon, G.pokemons) ∧ c $\in$ G.coordenadasConPokemons) ∧ (9) ((∀ j: jugador)(0 $\leq$ j $<$ Longitud(G.jugadores) $\wedge_L$ G.jugadores[j].expulsado = false) $\Rightarrow_L$ SumaPokemon(claves(Siguiente(G.jugadores[j].pokemons)), Siguiente(G.jugadores[j].pokemons)) = G.jugadores[j].cantTotalPoke)

pokemonsSalvajes : conj(pokemons)cp × dicc(pokemons × <nat × nat>d) $\longrightarrow$ nat
{(∀ p: Pokemon) (p $\in$ cp) $\Rightarrow$ (Definido?(x, d))}

pokemonsSalvajes(cp, d) $\equiv$ **if** ∅(cp) **then**
0
else
π_1(Significado(DameUno(cp), d)) + pokemonsSalvajes(SinUno(cp), d)
fi

SumarDicc : Conj(string) cs × Dicc(stirng × <nat × nat>) d $\longrightarrow$ nat
{(∀ s: Sting)(s ∈ cs) ⇒ (Definido?(x, d))}
SumarDicc(cs, d) ≡ **if** ∅?(cs) **then** 0 **else** π_2(Significado(DameUno(cs), d)) + SumarDicc(SinUno(cs), d) **fi**
SumarPokemon : Conj(string) cs × Dicc(stirng × nat) d $\longrightarrow$ nat {(∀ s: Sting)(s ∈ cs) ⇒ (Definido?(x, d))}
SumarPokemon(cs, d) ≡ **if** ∅?(cs) **then** 0 **else** Significado(DameUno(cs), d) + SumarPokemon(SinUno(cs), d) **fi**

4.1.3 Predicado de abtraccion

Abs : Game g $\longrightarrow$ juego {Rep(g)}
Abs(g) ≡ j :Juego| (g.mapa = mapa(j)) ∧ (g.jugadores = jugadores(j)∪expulsados(j) $\wedge_L$ (∀ x :Jugador)(x ∈ j.jugadores) $\Rightarrow_L$ estaConectado(x, j) = g.jugadores[x].conectado ∧ sanciones(x, j) = g.jugadores[x].sanciones ∧ posicion(x, j) = g.jugadores[x].pos ∧ deDiccAMulti(claves(Siguiente(g.jugadores[x].pokemons)), Siguiente(g.jugadores[x].pokemons)) = pokemons(x, j)) ∧ (g.coordenadasConPokemons = j.posConPokemon $\wedge_L$ (∀ c: coordenada)(c *in* j.posConPokemons) $\Rightarrow_L$ g.mapaInfo[c].pokemon = pokemonEnPos(c, j) ∧ g.mapaInfo[c].MovimientosRestantes = cantMovimientosParaCapturar(c, j))
deDiccAMulti : conj(pokemon) cp × dicc(pokemon × cantidad)d $\longrightarrow$ multiconj(pokemon)
{(∀ p: Pokemon) (p ∈ cp) ⇒ (Definido?(x, d))}
deDiccAMulti(cp, d) ≡ **if** ∅(cp) **then**
∅
else
agregarATodos(Significado(DameUno(cp), d), DameUno(cp)) ∪ deDiccAMulti(SinUno(cp), d)
fi
agregarATodos : nat × pokemon $\longrightarrow$ multiconj(pokemon)
agregarATodos(n, p) ≡ **if** 0?(n) **then** ∅ **else** Ag(p, agregarATodos(n-1, p)) **fi**

Representacion del iterador jugador

4.2 Justificacion

El objetivo del iterador era poder devolver un iterador a la lista de jugadores en O(1) pudiendo siguiente no ser O(1). Creamos el iterados ya que en la lista de jugadores guardamos los jugadores que están jugando como los expulsados y con el iterador recorremos los jugadores no expulsado. Guardamos la ultima posición donde estamos paradon con posicion y el puntero al Vector real de los jugadores

`itJugador` **se representa con** `iter`

donde `iter` es `tupla`(*posicion*: `nat` , *vector*: `puntero(Vector(infoJugador))`)

4.3 Invariante de representacion

Informal
(1)Posicion existe como indice del vector
Formal

Rep : iterI $\longrightarrow$ bool
Rep(I) ≡ true $\Longleftrightarrow$ (1) 0 ≤ I.posicion < Longitud(*I.vector)

4.4 Predicado de abstraccion

Abs : iter I $\longrightarrow$ itJugador {Rep(I)}
Abs(I) ≡ it :Iterador Unidireccional| Siguientes(I) = Siguientes(it)

Representacion del iterador pokemon

4.5 Justificacion

La idea del iterador es ocultar (encapsular) al iterador del DiccString

`itPokemon` se representa con `iter`

donde `iter` es `tupla`(*iterador*: `itString`)

4.6 Invariante de representacion

Formal

true

Informal

Rep : iterI $\longrightarrow$ bool
Rep(I) $\equiv$ true $\Longleftrightarrow$ true

4.7 Predicado de abstraccion

Abs : iter I $\longrightarrow$ itPokemon {Rep(I)}
Abs(I) $\equiv$ it :Iterador Unidireccional| Siguientes(I) = Siguientes(it)

No exportable, operaciones auxiliares

CeldasValidas(**in** g: `juego`, **in** c: `coordenada`) $\rightarrow$ res : `lista(coordenadas)`
Pre $\equiv$ {c $\in$ coordenadas(mapa(g))}
Post $\equiv$ {($\forall c_1$: coordenada)(esta?(c_1, res)$\Rightarrow_{\text{L}}$(distEuclidea(c_1, c)$\leq$2 $\wedge$ posExistente(c_1, mapa(g)))}
Complejidad: O(1)
Descripción: Devuelve una lista con las coordenas a una distancia no mayor de 2 de la coordenada c y que ademas existan en el mapa del juego

4.8 Algoritmos

Algoritmos del Modulo

iCrearJuego(**in** m: `Mapa`) $\rightarrow$ res: Game

```
1:  coords ← Coordenadas(m)                                              ▷ O(1)
2:  iter ← CrearIt(coords)                                               ▷ O(1)
3:  ancho ← 0                                                            ▷ O(1)
4:  alto ← 0                                                             ▷ O(1)
5:  while HaySiguiente(iter) do                                          ▷ O(#coords)
6:      c ← Siguiente(iter)                                              ▷ O(1)
7:      Avanzar(iter)                                                    ▷ O(1)
8:      if Altitud(c) > alto then alto ← Altitud(c) else fi              ▷ O(1)
9:      if Longitud(c) > ancho then ancho ← Longitud(c) else fi          ▷ O(1)
10: end while
11: infocoor ← arreglo[ancho] de arreglo[alto] de <Vacía(), Bool, Vacía(), 10>   ▷ O(m.ancho * m.alto)
12: res ← <Vacio(), m, Vacía(), infocoor, 0, Vacía()>                    ▷ O(1)
```

Complejidad: O(TM)
Justificacion: Donde TM es el tamaño del mapa (alto $\times$ ancho)

iAgregarJugador(**in** g : `Game`) → res: itJuego(σ)

```
1: Dicc(pokemon, nat) dicc ← Vacio()                                          ▷ O(1)
2: itLista(Dicc(pokemon, cantidad)) it ← AgregarAtras(g.pokemonsDeJugadores, dicc)   ▷ O(1)
3: AgregarAtras (g.jugadroes, <false, false, 0, <0, 0>, it, NULL, 0>)   ▷ O(longitud(g.jugadores) + copy(tupla))
4: itJuego(σ) it ← CrearIt(g)                                                 ▷ O(1)
5: while it.posicion < Longitud(g.jugadores) do                               ▷ O(longitud(g.jugadores))
6:     it.posicion ← it.posicion + 1                                          ▷ O(1)
7: end while
8: res ← it                                                                   ▷ O(1)
```

Complejidad: O(longitud(g.jugadores))
Justificacion:Agrega un jugador al juego, el costo de copiar la tupla es O(1) porque todas las componentes están vacías, después crea un iterador al principio del vector y lo avanza hasta la última posición donde fue agregado el jugador y lo devuelve. Para hacer esto último tengo que recorrer todo el vector entonces la complejidad final es O(longitud(g.jugadores) + (longitud(g.jugadores) + copiar(tupla))), como dijimos el costo de copiar

iAgregarPokemon(**in** p: `string`, **in** c: `coordenada`, **in/out** g: `game`)

```
1: AgregarRapido(g.coordenadasPokemons, c)                                   ▷ O(copiar(c)) = O(1)
2: if ¬Definido(p, g.pokemons) then                                          ▷ O(2*|p|) = O(|p|)
3:     Definir(p, <1, 1>, g.pokemons)                                        ▷ O(|P|)
4: else
5:     Definir(p, <Significado(p, g.pokemons).cantSalvaje +1, Significado(p, g.pokemons).cantTotales +1>,
   g.pokemons)                                                               ▷ O(|p|)
6: end if
7: g.mapaInfo[c].hayPokemon ← true                                           ▷ O(1)
8: g.mapaInfo[c].jugEspe ← Vacio()                                           ▷ O(1)
9: g.mapaInfo[c].yaSeCapturo ← flase                                         ▷ O(1)
10: g.mapaInfo[c].movimientosRestantes ← 0                                   ▷ O(1)
11: g.mapaInfo[c].pokemon ← p                                                ▷ O(1)
12: lista(coordenada) lc ← Vacia()                                           ▷ O(1)
13: lc ← CeldasValidas(g, c)                                                 ▷ O(1)
14: AgregarAtras(lc, c)                                                      ▷ O(1)
15: itLista(coordenadas) itCoordenadas ← CrearIt(lc)                         ▷ O(1)
16: while HaySiguiente(itCoordenadas) do                                     ▷
   O(#(jugadoresEnRadioDeCaptura)*log(#(jugadoresEnRadioDeCaptura))
17:     itDicc(jugador, EsperandoCapturar) itJugadores ← CrearIt(g.mapaInfo[Siguiente(itCoordenadas)].jugadores)
   ▷ O(1)
18:     while HaySiguiente(itJugadores) do                                   ▷
   O(#(jugadoresEnRadioDeCaptura)*log(#(jugadoresEnRadioDeCaptura))
19:         if SiguienteSignificado(itJugadores) ≠ NULL then
20:             Borrar(SiguienteSignificado(itJugadores))
21:         end if
22:         itColaPrioridad itCola ← Encolar(g.mapaInfo[c].jugEspe, g.jugadores[SiguienteClave(itJugadores)].cantTotPoke,
   SiguienteClave(itJugadores))   ▷ O(log (n), siendo n la cantidad de elementos en el arbol)
23:         SiguienteSignficado(itJugadores) ← itCola                        ▷ O(1)
24:         Avanzar(itJugadores)                                             ▷ O(1)
25:     end while
26:     if EsVacio?(g.mapaInfo[Siguiente(itCoordena)].jugEspe) then
27:         g.mapaInfo[Siguiente(itCoordena)].yaSeCapturo ← false            ▷ O(1)
28:     end if
29:     Avanzar(itCoordenada)                                                ▷ O(1)
30: end while
```

Complejidad: O((#(jugadoresEnRadioDeCaptura)*log(#(jugadoresEnRadioDeCaptura))) + |P|)
Justificacion:Primero defino el pokemon en el diccString, si ya estaba sumo un 1 en la cant de pokemons salvajes, sino lo defino con un 1 en cant salvajes y 0 en atrapados, esto me toma |P| siendo P la máxima longitud de una clave del diccionario. Después me armo una lista con las coordenadas en el radio de captura esto toma tiempo constate porque son finitas cordenadas. Luego creo un iterador a esta lista para recorrerla, como tengo finitos elementos recorrerla es constante. Por cada coordenada me creo un it al diccionario de los jugadores en esa coordenada. Los recorro, recorrer a todos los jugadores en el radio es EC siendo EC la máxima cantidad de jugadores esperando capturar un pokemon, y por cada jugador pregunto si su iter a cola de prioridad esta definido (no supimos como hacerlo asi que lo consideramos como un puntero) si lo esta borra lo que esta apuntando, despues encolo el elemento a la cola de prioridad del pokemon, me guardo el iterador que me devuelve encolar, y asi para todos los jugadores de la coordena, despues de salir de este while que toma la cantidad de los jugadores de la coordenada, O(#jugadores en la coordenada), y por cada uno lo encolo a la cola eso me toma O(log n)(n la cantidad de elementos de la cola), entonces la complejidad final es O(#jugadores en la coordenada * log n). Esto lo hago para todas las coordenadas entonces me queda O(#JugadoresEsperandoCapturar * log(#JugadoresEsperandoCapturar)), luego antes de avanzar de coordenada, pregunto si el la que estoy parado su cola de prioridad esta vacía, si lo está pongo a false un booleano, este es el caso de que hubo alguna vez un pokemon en esa coordenada. Y despues de salir del while general termine todo entonces la complejidad final es O((#(jugadoresEnRadioDeCaptura)*log(#(jugadoresEnRadioDeCaptura))) + |P|)

iCeldasValidas(**in** g: `Game`, **in** c: `coordenada`) → res: lista(coordenada)

```
1: lista(coordenada) ls ← Vacia()                                                    ▷ O(1)
2: nat i ← 4                                                                         ▷ O(1)
3: while i > 0 do                                                                    ▷ O(1)
4:     AgregarAtras(ls, <latitud(c)+i, longitud(c)>)                                 ▷ O(1)
5:     if latitud(c)-i>0 then
6:         AgregarAtras(ls, <latitud(c)-i, longitud(c)>)                             ▷ O(1)
7:     end if
8:     AgregarAtras(ls, <latitud(c), longitud(c)+i>)                                 ▷ O(1)
9:     if longitud(c)-i >0 then
10:        AgregarAtras(ls, <latitud(c), longitud(c)-i>)                             ▷ O(1)
11:    end if
12:    i ← i - 1                                                                     ▷ O(1)
13: end while
14: i ← 3
15: while i > 0 do                                                                   ▷ O(1)
16:    if longitud(c)-(i-1)>0 then
17:        AgregarAtras(ls, <latitud(c)+3, longitud(c)-(i-1)>)                       ▷ O(1)
18:    end if
19:    if latitud(c)-(i-1)>0 then
20:        AgregarAtras(ls, <latitud(c)-(i-1), longitud(c)+3>)                       ▷ O(1)
21:    end if
22:    if latitud(c)-3>0 ∧ longitud(c)-(i-1) then
23:        AgregarAtras(ls, <latitud(c)-3, longitud(c)-(i-1)>)                       ▷ O(1)
24:    end if
25:    if latitud(c)-(i-1) > 0 ∧ longitud(c)-3 > 0 then
26:        AgregarAtras(ls, <latitud(c)-(i-1), longitud(c)-3>)                       ▷ O(1)
27:    end if
28:    if latitud(c)-3 > 0 then
29:        AgregarAtras(ls, <latitud(c)-3, longitud(c)+(i-1)>)                       ▷ O(1)
30:    end if
31:    if longitud(c)-3 > 0 then
32:        AgregarAtras(ls, <latitud(c)+(i-1), longitud(c)-3>)                       ▷ O(1)
33:    end if
34:    AgregarAtras(ls, <latitud(c)+3, longitud(c)+(i-1)>)                           ▷ O(1)
35:    AgregarAtras(ls, <latitud(c)+(i-1), longitud(c)+3>)                           ▷ O(1)
36:    AgregarAtras(ls, <latitud(c)+(i-1), longitud(c)+2>)                           ▷ O(1)
37:    AgregarAtras(ls, <latitud(c)+(i-1), longitud(c)+1>)                           ▷ O(1)
38:    if longitud(c)-2 > 0 then
39:        AgregarAtras(ls, <latitud(c)+(i-1), longitud(c)-2>)                       ▷ O(1)
40:    end if
41:    if longitud(c)-1 > 0 then
42:        AgregarAtras(ls, <latitud(c)+(i-1), longitud(c)-1>)                       ▷ O(1)
43:    end if
44:    if latitud(c)-(i-1) > 0 ∧ longitud(c)-2 >0 then
45:        AgregarAtras(ls, <latitud(c)-(i-1), longitud(c)-2>)                       ▷ O(1)
46:    end if
47:    if latitud(c)-(i-1)>0 ∧ longitud(c)-1 >0 then
48:        AgregarAtras(ls, <latitud(c)-(i-1), longitud(c)-1>)                       ▷ O(1)
49:    end if
50:    if latitud(c)-(i-1) > 0 then
51:        AgregarAtras(ls, <latitud(c)-(i-1), longitud(c)+2>)                       ▷ O(1)
52:    end if
53:    if latitud(c)-(i-1) >0 then
54:        AgregarAtras(ls, <latitud(c)-(i-1), longitud(c)+1>)                       ▷ O(1)
55:    end if
56:    i ← i -1                                                                      ▷ O(1)
57: end while
58: itLista(coordenada) it ← CrearIt(ls)                                             ▷ O(1)
59: while HaySiguiente(it) do                                                        ▷ O(1)
60:    if PosExistente (Siguiente(it), g.mapa) then                                  ▷ O(1)
61:        Avanzar(it)                                                               ▷ O(1)
62:    else
63:        EleminarSiguiente(it)                                                     ▷ O(1)
64:    end if
65: end while
66: res ← ls                                                                         ▷ O(1)
67: if  then
68: end if
```

Complejidad: O(1))

Justificacion:Como me estoy fijando un numero finito de coordenadas, y la cantidad que veo no varia porque no depende de la entrada, puedo decir que toma O(1) ver todas las celdas, luego recorro la lista para ver cuales son válidas y cuales no, que como son una cantidad constatnte de celdas recorrer la lista tambien es constante

iMoverse(**in** j : `jugador`, **in** c : `coordenada`, **in/out** g : `Game`)

1: VerCapturas(j, c, g) ▷ O(#(elementosEnConj) * |PenJ|)
2: ActualizarJugador(j, c, g) ▷ O(log #(elementosEnColaDeC) + #(pokemonsDiccDeJugador)*|P|)

Complejidad:O(#(elementosEnConj) * |PenJ| + log #(elementosEnColaDeC) + #(pokemonsDiccDeJugador)*|P|) = O((#(pokemonsDiccDeJugador + #(elementosEnConj)) * |P|) + log #(elementosEnColaDeC)) = O((PC + PS) * |P| + log(EC))

Justificacion:Primero veo que #(elementosEnConj) son los pokemons salvajes que quedan en el juego porque esa complejidad viene de recorrer el conjunto de coordenadas de pokemons en el mapa, entonces #(elementosEnConj) = PS, luego #(pokemonsDiccDeJugador) es la complejidad de recorrer todas las claves del diccionario donde guardo la informacion de los pokemons de cada jugador, entonces esta complejidad esta dada por PC que es la máxima cantidad de pokemons atrapados por un jugador. Luego tengo log #(elementosEnColaDeC) que viene de la parte de mover el jugador y actualizar los heap, en este caso esta complejidad se engloba por log(EC) siendo EC la máxima cantidad de jugadores esperando capturar un pokemon y por último quedan |PenJ| y |P| estas complejidad se engloban por |P′| siendo P′ la máxima longitud de un nombre de un pokemon en el juego. Luego la complejidad final queda O((PC + PS)*|P′| + log(EC)), aprovechando que |PenJ| y |P| estan contenidas en |P′|

iActualizarJugadorYCoordenadas(**in** j: `jugador`, **in** c: `coordenada`, **in/out** g: `Game`)

```
1: if distEuclidia(c, g.jugadores[j].pos) > 100 ∨ ¬(hayCamino(c, g.jugadores[j].pos, g.mapa)) then ▷ O(1)
2:     g.jugadores[j].sanciones ← g.jugadores[j].sanciones +1 ▷ O(1)
3:     if ( thening.jugadores[j].sanciones ≥ 4) ▷ O(1)
4:         g.jugadores[j].expulsado ← true ▷ O(1)
5:     end if
6: end if
7: if g.jugadores[j].expulsado = true then ▷ O(#(pokemonsDicc)*|P|)
8:     g.cantidadTotPokemons ← g.cantidadTotPokemons - g.jugadores[j].cantTotalPoke ▷ O(1)
9:     itLista(string) itPokemons ← CrearIt(Claves(Siguiente(g.jugadores[j].pokemons))) ▷ O(1)
10:    while HayMas?(itPokemons) do ▷ O(#(pokemonsDicc)*|P|)
11:        Significado(g.pokemons, Siguiente(itPokemons).cantTotal - Significado(g.jugadores[j].pokemons, Siguiente(itPokemons)) ▷ O(|P|+|P|) = O(|P|)
12:        EliminarSiguiente(itPokemons) ▷ O(1)
13:    end while
14:    EliminarSiguiente(g.jugadores[j].pokemons) ▷ O(1)
15:    g.jugadores[j].pokemons ← NULL ▷ O(1)
16:    g.jugadores[j].cantTotalPoke ← 0 ▷ O(1)
17: else
18:    if hayPokemonCercano(g.jugadores[j].pos, g) then ▷ O(1)
19:        if hayPokemonCercano(c, g) then ▷ O(1)
20:            if posPokemonCercano(g.jugadores[j].pos, g) ≠ posPokemonCercano(c, g) then ▷ O(1)
21:                g.mapaInfo[g.jugadores[j].pos].MovimientosRestantes ← 0 ▷ O(1)
22:            end if
23:        else
24:            g.mapaInfo[g.jugadores[j].pos].MovimientosRestantes ← 0 ▷ O(1)
25:        end if
26:    else
27:        if hayPokemonCercano(c, g) then ▷ O(1)
28:            g.mapaInfo[c].MovimientosRestantes ← 0 ▷ O(1)
29:        end if
30:    end if
31: end if
32: g.jugadores[j].pos ← c ▷ O(1)
33: if HayPokemonCerca(c, g) then ▷ O(log #(elementosEnColaDeC))
34:    itDicc(jugador, itColaPrioridad(cantPokemon)) itPosicion ← DefinirRapido(g.mapaInfo[c].jugadores, j, Encolar(g.mapaInfo[posPokemonCerca(c, g)].jugEspe, g.jugadores[j].cantTotalPoke, j)) ▷ O(log #(elementosEnColaDeC))
35:    g.jugadores[j].posicionMapa ← itPosicion ▷ O(1)
36:    g.jugadores[j].posicionMapa ← itPosicion ▷ O(1)
37: else
38:    itDicc(jugador, itColaPrioridad(cantPokemon)) itPosicion ← DefinirRapido(g.mapaInfo[c].jugadores, j, NULL) ▷ O(1)
39:    g.jugadores[j].posicionMapa ← itPosicion ▷ O(1)
40: end if
```

Complejidad: O(log #(elementosEnColaDeC) + #(pokemonsDicc)*|P|), donde P es la máxima longitud de las claves de diccionario

Justificacion:Primero chequeo si el jugador cometió una infracción, si lo hizo le sumo una sanción. Entonces veo si lo tengo que expulsar, si lo hago resto su cantidad de pokemons al total de pokemons en el juego porque desaparecen, despues por cada pokemon del diccionario del jugador, resto su cantidad a su respectivo pokemon en el diccString (resto a su cantidadTotal en el diccString), esto me toma O(#(elementos del dicc) * |P|). Si no lo tengo que expulsar simplemente defino su nueva coordenada y veo si tengo que agregarlo a una nueva cola de prioridad o no, si tengo que hacerlo tengo que revalanciar la cola eso me cuesta O(log #(elementos de la cola en la coordenada c)), sino es constante. Entonces la complejidad final es O(log #(elementosEnColaDeC) + #(pokemonsDicc)*|P|). Para chequear que la complejidad cumple la pedida (La función la uso en moverse) veo que #(elementos del diccionario de pokemons del jugador) es a lo sumo PC (máxima cantidad de pokemons atrapados por un jugador), cuando lo multiplico por |P|, veo que la complejidad está contenida en O((PS + PC) * |P|). Para la otra parte de la complejidad veo que log #(elementos de la cola de C) se acota por log (EC) (donde EC es la máxima cantidad de jugadores esperando capturar un pokemon) esto pasa porque la cola está contemplada en la complejidad, pasaría, en peor caso, O(log(EC)), entonces la complejidad de esta función está contenida en la pedida.

iVerCapturas(**in** j : `jugadores`, **in** c : `coordenadas`, **in/out** g : `Game`)

```
1: coordenada coordeJ ← g.jugadores[j].pos                                                    ▷ O(1)
2: itConj(coordenadas) itPokeCoordenadas ← g.coordenadasPokemons                              ▷ O(1)
3: while HaySiguiente(itPokeCoordenadas) do                          ▷ O(#(elementosEnConj) * |PenJ|)
4:     if Siguiente(itPokeCoordenadas) neq coordeJ then                                       ▷ O(1)
5:         g.mapaInfo[Siguiente(itPokeCoordenadas)].MovimientosRestantes ← g.mapaInfo[Siguiente(itPokeCoordenadas)].Mov
   +1                                                                                         ▷ O(1)
6:         if g.mapaInfo[Siguiente(itPokeCoordenadas)].MovimientosRestantes ≥ 10 then         ▷ O(1)
7:             g.mapaInfo[Siguiente(itPokeCoordenadas)].hayPokemon ← false                    ▷ O(1)
8:             g.mapaInfo[Siguiente(itPokeCoordenadas)].yaSeCapturo ← true                    ▷ O(1)
9:             g.mapaInfo[Siguiente(itPokeCoordenadas)].MovimientosRestantes ← 0              ▷ O(1)
10:            Significado(g.pokemons, g.mapaInfo[Siguiente(itPokeCoordenadas)].pokemon).cantSalvaje ← Signifi-
   cado(g.pokemons, g.mapaInfo[Siguiente(itPokeCoordenadas)].pokemon).cantSalvaje -1          ▷
   O(|P|)
11:            jugador jug ← Proximo(g.mapaInfo[Siguiente(itPokeCoordenada)].jugEspe)         ▷ O(1)
12:            if Definido?(Siguiente(g.jugadores[jug].pokemons), g.mapaInfo[Siguiente(itPokeCoordenadas)].pokemon)
   then                                                          ▷ O(3*|PenJ|) = O(|PenJ|)
13:                Significado(Siguiente(g.jugadores[jug].pokemons), g.mapaInfo[Siguiente(itPokeCoordenadas)].pokemon)
   ← Significado(Siguiente(g.jugadores[jug].pokemons), g.mapaInfo[Siguiente(itPokeCoordenadas)].pokemon) +1   ▷
   O(|PenJ|)
14:            else
15:                Definir(Siguiente(g.jugadores[jug].pokemons), g.mapaInfo[Siguiente(itPokeCoordenadas)].pokemon,
   1)                                                                                         ▷ O(|PenJ|)
16:            end if
17:        end if
18:        EleminarSiguiente(itPokeCoordenada)                                                ▷ O(1)
19:    else
20:        Avanzar(itPokeCoordenada)                                                          ▷ O(1)
21:    end if
22: end while
```

Complejidad:O(#(elementosEnConj) * |PenJ|)
Justificacion:Recorro todas las coordenadas en donde hay pokemons y por cada una veo si se produce una captura (movimientosRestantes >= 10), si se podruce busco el jugador que lo captura (Pido tope a la colaDePrioridad) en O(log #(elementosCola))) y Defino al pokemon en el diccionario de pokemons del jugador (O(|PenJ|)) y si ya estaba definido le modifico el significado en ambos casos me toma lo mismo en complejidad temporal, despues pongo el booleano yaSeAtrapo en true. Entonces la complejidad final es O(#(elementosEnConj) * |PenJ|). Para verla con el enunciado queda O(PS * |P|)

iConectarse(**in** j : `jugador`, **in** c : `coordenada`, **in/out** g : `Game`)

```
1: g.jugadores[j].conectado ← true                                                    ▷ O(1)
2: g.jugadores[j].pos ← c                                                             ▷ O(1)
3: if HayPokemonCerca(c, g) then                                                      ▷ O(log #(elementosEnColaDeC))
4:     itDicc(jugador, itColaPrioridad(cantPokemon)) itPosicion ← DefinirRapido(g.mapaInfo[c].jugadores, j, Encolar(g.mapaInfo[posPokemonCerca(c, g)].jugEspe, g.jugadores[j].cantTotalPoke, j))   ▷ O(log #(elementosEnColaDeC))
5:     g.jugadores[j].posicionMapa ← itPosicion                                       ▷ O(1)
6:     g.jugadores[j].posicionMapa ← itPosicion                                       ▷ O(1)
7: else
8:     itDicc(jugador, itColaPrioridad(cantPokemon)) itPosicion ← DefinirRapido(g.mapaInfo[c].jugadores, j, NULL)   ▷ O(1)
9:     g.jugadores[j].posicionMapa ← itPosicion                                       ▷ O(1)
10: end if
```

Complejidad: O(log #(elementosEnColaDeC))
Justificacion:A la hora de conectarse, primero veo si en la coordenada c hay un pokemon cerca, si lo hay agrego el jugador a la cola de prioridad de la coordenada del pokemon eso me cuesta O(log #(elementosEnCola)), si no hay pokemon cerca simplemente no existe el iterado. Luego cargo los datos necesarios para el jugador

iDesconectarse(**in** j : `jugador`, **in/out** g : `Game`)

```
1: g.jugadores[j].conectado ← false                                                   ▷ O(1)
2: if SiguienteSignificado(g.jugadores[j].posicionMapa) ≠ NULL then                   ▷ O(log #(elementosEnCola))
3:     Borrar(SiguienteSignificado(g.jugadores[j].posicionMapa))                      ▷ O(log #(elementosEnCola))
4:     EleminarSiguiente(g.jugadores[j].posicionMapa)                                 ▷ O(1)
5:     g.jugadores[j].posicionMapa ← NULL                                             ▷ O(1)
6: else
7:     EleminarSiguiente(g.jugadores[j].posicionMapa)                                 ▷ O(1)
8:     g.jugadores[j].posicionMapa ← NULL                                             ▷ O(1)
9: end if
```

Complejidad: O(log #(elementosEnCola))
Justificacion:Cuando un jugador se desconecta, primero actualizo su flag de conectado a false y despues veo si esta en alguna cola de prioridad, si lo está elimino el nodo al que apunta (reordeno la cola) y despues borro el diccionario en donde está definido y pongo a null el iterador. Si está en una cola la complejidad (para reordenoar) toma O(log #(elementosEnCola)) sino toma O(1)

iMapa(**in** g : `Game`) → res: mapa

```
1: res ← g.mapa                                                                       ▷ O(1)
```

Complejidad: O(1)
Justificacion: Devuelve la instancia de mapa que tenemos guardada

iJugadores(**in** g : `Game`) → res: conj(jugador)

```
1: res ← CrearIt(g)                                                                   ▷ O(1)
```

Complejidad: O(1)
Justificacion: Crea un iterador a jugadores que tiene orden de 1 y lo devuelve

iEstaConectado(**in** j : `jugador`, **in** g : `Game`) → res: conj(jugador)

1: res ← g.jugadores[j].conectado ▷ O(1)

Complejidad: O(1)
Justificacion: Es una asignación, un acceso de O(1) a un vector por el id del jugador y ahí guardamos una tupla con información del jugador, en particular si está ó no conectado

iSanciones(**in** j : `jugador`, **in** g : `Game`) → res: nat

1: res ← g.jugadores[j].sanciones ▷ O(1)

Complejidad: O(1)
Justificacion: Es una asignación, un acceso de O(1) a un vector por el id del jugador y ahí guardamos una tupla con información del jugador, en particular la cantidad de sanciones que tiene

iPosicion(**in** j : `jugador`, **in** g : `Game`) → res: coordenada

1: res ← g.jugadores[j].pos ▷ O(1)

Complejidad: O(1)
Justificacion: Es una asignación, un acceso de O(1) a un vector por el id del jugador y ahí guardamos una tupla con información del jugador, en particular la posición actual del jugador cuando esta conectado

iPokemons(**in** j : `jugador`, **in** g : `Game`) → res: itDicc($< Pokemon, cantidad >$)

1: res ← CrearIt(Siguiente(g.jugadores[j].pokemons)) ▷ O(1)

Complejidad: O(1)
Justificacion: Dentro de jugadores guardamos un iterador a Dicc(<Pokemon, Cantidad>) que está guardado en Game. De esta forma en el vector de jugadores guardamos estructuras simples de copiar, esto era necesario por la complejidad del agregar jugador (ver AgregarJugador). Tanto la asignación y la creación del iterador y el siguiente del iterador de lista son todos O(1)

iExpulsados(**in** g : `Game`) → res: conj(jugador)

1: res ← Vacío() ▷ O(1)
2: tam ← Longitud(g.jugadores) ▷ O(1)
3: **for** n ← 0 **to** tam ▷ O(J)
4: AgregarRapido(res, g.jugadores[n].jug) ▷ O(copy(jugador))

Complejidad: O(J)
Justificacion: El for se ejecuta J veces y como jugador es un nat, el costo de copiarlo es O(1) entonces la complejidad es O(J)

iPosConPokemons(**in** g : `Game`) → res: conj(coordenada)

1: res ← g.coordenadasConPokemons ▷ O(1)

Complejidad: O(1)
Justificacion: Devolvemos el conjunto de coordenadas con Pokemones. La asignación al ser de un tipo **no** primitivo tanto res como g.coordenadasConPokemons referencian a la misma estructura física (del apunte de diseño)

iPokemonEnPos(**in** c: `coordenada`, **in** g: `Game`) → res: pokemon

1: res ← g.mapainfo[Altitud(c)][Longitud(c)].pokemon ▷ O(1)

Complejidad: O(1)
Justificacion: El acceso a mapainfo que es una matriz es O(1) y tenemos solo una asignación

iCantMovimientosParaCaptura(**in** c: `coordenada`, **in** g: `Game`) → res: nat

1: res ← 10 - g.mapainfo[Altitud(c)][Longitud(c)].MovimientosRestantes ▷ O(1)

Complejidad: O(1)
Justificacion: El acceso a mapainfo que es una matriz es O(1), tenemos solo una asignación y una resta

iJugadoresConectados(**in** g: `Game`) → res: conj(jugador)

1: res ← Vacío() ▷ O(1)
2: tam ← Longitud(g.jugadores) ▷ O(1)
3: **for** n ← 0 **to** tam ▷ O(J)
4: **if** g.jugadores[n].conectado **then** AgregarRapido(res, g.jugadores[n].jug) **else** **fi** ▷ O(copy(α))
5: **endfor**

Complejidad: O(J)
Justificacion: El for se ejecuta J veces el agregar rapido tiene el costo de copiar el elemento J veces, pero al ser jugador un natural, el costo es O(1) luego el costo de la función es O(J)

iPuedoAgregarPokemon(**in** c: `Coordenada`, **in** g: `Game`) → res: Bool

1: res ← True ▷ O(1)
2: **if** PosExistente(c, g.mapa) **then** ▷ O(1)
3: coordPokemons ← PosConPokemons(g) ▷ O(1)
4: iter ← CrearIt(coordPokemons) ▷ O(1)
5: **while** HaySiguiente(iter) **do** ▷ O(PS)
6: **if** DistEuclidea(Siguiente(iter), c) < 25 **then** res ← False **else** **fi** ▷ O(1)
7: Avanzar(iter) ▷ O(1)
8: **end while**
9: **else**
10: **end if**

Complejidad: O(PS)
Justificacion: Son todas operaciones de O(1) pero el while se ejecuta PS veces ejecutando código O(1) por lo tanto todo tiene complejidad O(PS)

iHayPokemonCercano(**in** c: `Coordenada`, **in** g: `Game`) → res: Bool

```
1:  res ← False                                                                  ▷ O(1)
2:  if PosExistente(c, g.mapa) then                                              ▷ O(1)
3:      coordCercanas ← CeldasValidas(g, c)                                      ▷ O(1)
4:      iter ← CrearIt(coordPokemons)                                            ▷ O(1)
5:      while HaySiguiente(iter) do                                              ▷ O(1)
6:          coor ← Siguiente(iter)                                               ▷ O(1)
7:          if g.mapainfo[Altitud(coor)][Longitud(coor)].hayPokemon then res ← True else fi   ▷ O(1)
8:          Avanzar(iter)                                                        ▷ O(1)
9:      end while
10: end if
```

Complejidad: O(1)
Justificacion: Son todas asignaciones y operaciones en O(1). El único detalle es la cantidad de coordenadas que devuelve CeldasValidas. Como la función devuelve todas las coordenadas a radio 4 de la coordenada que nos pasan, sabemos que devuelve una cantidad constante de coordenadas (salvo en el borde del mapa que devuelve menos pero que igual está acotada por el radio 4) y al ser constante se puede despreciar y es O(1)

iIndiceRareza(**in** p: `pokemon`, **in** g: `Game`) → res: nat

```
1:  res ← 100                                                                    ▷ O(1)
2:  nat aux ← significado(j.gokemons, p).cantTotal                               ▷ O(|s|)
3:  aux ← aux / g.cantTotPokemons                                                ▷ O(1)
4:  res ← res - aux ∗ 100                                                        ▷ O(1)
```

Complejidad: O(|s|)
Justificacion: Devuelve el índice de rareza del pokemon pedido. La única operación que tiene complejidad mayor que O(1) es buscar en el trie de pokemons, la complejidad de esta operación es en peor caso buscar el nombre de pokemon más largo, por lo tanto la complejidad es O(|s|).

iCantPokemonsTotales(**in** g: `Game`) → res: nat

```
1:  res ← g.cantTotPokemons                                                      ▷ O(1)
```

Complejidad: O(1)
Justificacion: Devuelve el valor de un elemento almacenado en la estructura, es una referencia y una asignación, operaciones de complejidad de orden constante.

iPosPokemonCercano(**in** c: `Coordenada`, **in** g: `Game`) → res: Coordenada

```
1:  if PosExistente(c, g.mapa) then                                              ▷ O(1)
2:      coordCercanas ← CeldasValidas(g, c)                                      ▷ O(1)
3:      iter ← CrearIt(coordPokemons)                                            ▷ O(1)
4:      while HaySiguiente(iter) do                                              ▷ O(1)
5:          coor ← Siguiente(iter)                                               ▷ O(1)
6:          if g.mapainfo[Altitud(coor)][Longitud(coor)].hayPokemon then
7:              res ← coor FI                                                    ▷ O(1)
8:          end if
9:          Avanzar(iter)                                                        ▷ O(1)
10:     end while
11: end if
```

Complejidad: O(1)
Justificacion: Son todas asignaciones y operaciones en O(1). El único detalle es la cantidad de coordenadas que devuelve CeldasValidas. Como la función devuelve todas las coordenadas a radio 4 de la coordenada que nos pasan, sabemos que devuelve una cantidad constante de coordenadas (salvo en el borde del mapa que devuelve menos pero que igual está acotada por el radio 4) y al ser constante se puede despreciar y es O(1)

Algoritmos del iterador jugadores

iCrearIt(**in** g : `game`) → res: iter))

```
1: res ← <0, puntero(g.jugadores)>                                  ▷ O(1)
```

Complejidad: O(1)
Justificacion:Crea un iterador al princio del vector, solo realiza una asignación

---

iHayMas?(**in** it : `iter`) → res: bool))

```
1: bool b ← false
2: nat i ← it.posicion                                              ▷ O(1)
3: while i < it.vector→ longitud ∧ ¬b do                            ▷ O(longitud(v))
4:     if it.vector→ elementos[i].expulsado = false then            ▷ O(1)
5:         b ← true                                                 ▷ O(1)
6:     end if
7:     i ← i +1                                                     ▷ O(1)
8: end while
9: res ← b
```

Complejidad: O(longitud(v))
Justificacion:En el peor de lo casos hay que recorrer todo el vector para saber si existe otro elemento

---

iAvanzar(**in/out** it : `iter`)

```
1: bool b ← false
2: nat i ← it.posicion                                              ▷ O(1)
3: while i < it.vector→ longitud ∧ ¬b do                            ▷ O(longitud(v))
4:     if it.vector→ elementos[i].expulsado = false then            ▷ O(1)
5:         b ← true                                                 ▷ O(1)
6:     end if
7:     i ← i +1                                                     ▷ O(1)
8: end while
9: it.posicion ← (i-1)                                              ▷ O(1)
```

Complejidad: O(longitud(v))
Justificacion:Recorre el vector y para en el primer elemento válido, que existe por la precondición de la función, luego actualiza la posición del iterador

---

iActual(**in** it : `iter`) → res: nat

```
1: res ← it.posicion                                                ▷ O(1)
```

Complejidad: O(1)
Justificacion:Devuelve la id del jugador que apunta el iterador

iSiguientes(**in** *it* : `iter`) → res: lista(jugadores)))

```
1: lista(nat) ls ← Vacia()                                   ▷ O(1), crea una lista vacia
2: nat i ← it.posicion                                       ▷ O(1)
3: while i < it.vector→ longitud do                          ▷ O(longitud(v))
4:     if it.vector→ elementos[i].expulsado = false then     ▷ O(1)
5:         AgregarAtras(ls, it.posicion)                     ▷ O(1)
6:     end if
7:     i ← i +1                                              ▷ O(1)
8: end while
9: res ← ls                                                  ▷ O(1)
```

Complejidad: O(longitud(v))
Justificacion:Para devolver una lista con los elementos que quedan por recorrer, recorro todo el vector viendo que elementos son válidos y si un elemento es válido lo agrego a una lista.

Algoritmos del iterador pokemons

iCrearIt(**in** *g* : `game`, **in** *j* : `jugador`) → res:iter

```
1: res ← CrearIt(Siguiente(g.jugadores[j].pokemons))         ▷ O(1)
```

Complejidad: O(1)
Justificacion:Crea un iterador al diccionario que contiene los pokemons del jugadro pasado por parametros

iHayMas?(**in** *it* : `itPokemon`) → res:bool

```
1: res ← HayMas?(it.iterador)                                ▷ O(1)
```

Complejidad: O(1)
Justificacion:Chequea si quedan elementos por recorrer

iActual(**in** *it* : `itPokemon`) → res:<string, nat>

```
1: res ← Actual(it.iterador)                                 ▷ O(|P|)
```

Complejidad: O(|P|)
Justificacion:Devuelve la clave y el significado en una tupla. Para el obtener el significado, busca en el arbol a travez de la clave

iAvanzar(**in/out** *it* : `itPokemon`)

```
1: Avanzar(it.iterador)                                      ▷ O(1)
```

Complejidad: O(1)
Justificacion:Avanza al siguiente elemento a recorrer

iSiguientes(**in** *it* : `itPokemon`) → res:lista(string)

```
1: res ← Siguientes(it.iterador)                             ▷ O(#(elementosDeClaves))
```

Complejidad: O(#(elementosDeClaves))
Justificacion:Devuelve los elementos que quedan por recorrer, O(2*#(elementosDeClaves)) = O(#(elementosDeClaves))

5 DiccString(σ)

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

5.1 Interfaz

Género diccString, itString
se explica con: String, Bool, Lista(String), Nat

Operaciones del modulo
Vacio() $\rightarrow$ res : `diccString(`σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {res = Vacio()}
Complejidad: O(1)
Descripción: Crea un diccionario vacio

Significado(**in** s: `string`, **in** d: `diccString(`σ`)`) $\rightarrow$ res : σ
Pre $\equiv$ {Definido(s, d)}
Post $\equiv$ {res = Significado(s, d)}
Complejidad: O(|L|), siendo L la máxima longitud de las claves en el dicc
Descripción: Devuelve el significado del diccionario

Definido(**in** s: `string`, **in** d: `diccString(`σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = Definido(s, d)}
Complejidad: O(|L|), siendo L la máxima longitud de las claves en el dicc
Descripción: Verifica si una clave existe o no en el diccionario

Vacio?(**in** d: `diccString(`σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = Vacio?(d)}
Complejidad: O(1)
Descripción: Chequea si un diccionario es vacio

Definir(**in** s: `string`, **in** c: σ, **in/out** d: `diccString(`σ`)`)
Pre $\equiv$ {d $=_{\text{obs}}$ d_0}
Post $\equiv$ {d = Definir(s, c, d_0)}
Complejidad: O(|L|), siendo L la máxima longitud de las claves del diccionario
Descripción: Define una clave en el diccionario

Borrar(**in** s: `string`, **in/out** d: `diccString(`σ`)`)
Pre $\equiv$ {d = d_0}
Post $\equiv$ {d = Borrar(s, d_0)}
Complejidad: O(|L|+ k), siendo L la máxima longitud de las claves en el diccionario y k la cant de claves en el dicc
Descripción: Borra la clave, si es prefijo de alguna otra clave solo borra el significado

Claves(**in** d: `diccString(`σ`)`) $\rightarrow$ res : `lista(string)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ claves(d)}
Complejidad: O(1)

Descripción: Devuelve un conjunto con las claves del diccionario

Operaciones del iterador
CrearIt(in d**:** `diccString(`σ`)`**)** $\rightarrow$ res **:** `itString`
Pre $\equiv$ **{true}**
Post $\equiv$ **{res** $=_{obs}$ **CrearItUni(claves(d))}**
Complejidad: O(1)
Descripción: Crea un iterador no modificable al principio de las claves del diccionario

Actual(in it**:** `itString`**)** $\rightarrow$ res **:** `<string, nat>`
Pre $\equiv$ **{HayMas?(it)}**
Post $\equiv$ **{res** $=_{obs}$ **Actual(it)}**
Complejidad: O(|P|), siendo P la clave mas larga del diccionario
Descripción: Devuelve una tupla <clave, significado>

HayMas?(in it**:** `itString`**)** $\rightarrow$ res **:** `bool`
Pre $\equiv$ **{true}**
Post $\equiv$ **{res** $=_{obs}$ **HayMas?(it)}**
Complejidad: O(1)
Descripción: Chequea si hay mas elementos para recorrer

Avanzar(in/out it**:** `itString`**)**
Pre $\equiv$ **{it** $=_{obs}$ **it**$_0$ $\wedge$ **HayMas?(it)}**
Post $\equiv$ **{res** $=_{obs}$ **Avanzar(it)}**
Complejidad: O(1)
Descripción: Avanza el iterador a la siguiente clave del diccionario

Siguientes(in it**:** `itString`**)** $\rightarrow$ res **:** `lista(string)`
Pre $\equiv$ **{true}**
Post $\equiv$ **{res** $=_{obs}$ **Siguientes(it)}**
Complejidad: O(n), siendo n la cantidad de claves del diccionario
Descripción: Devuelve una lista con las claves que aun no se recorrieron

Representación

5.2 Justificacion

En el diccionario sobre Trie lo representamos con Nodo apuntando a nodo donde guardamos cada char de la palabra en cada nodo. Y tenemos una estructura extra llamada Arbolito que su única finalidad es tener un Puntero a la raíz de Trie y guardamos una lista de las claves creadas. La lista de claves la creamos para poder iterar sobre el Trie de forma más conveniente, donde nos permite poder devolver un iterador en O(1), avanzar en las claves en O(1) y cuando piden el actual hacemos la busqueda sobre el Trie con la clave actual teniendo una complejidad de O(|clave mas larga|).

`diccString` **se representa con** `Arbolito`

donde `Arbolito` **es** `tupla(`***raiz***`: puntero(nodo(`σ`)) ,` ***claves***`: lista(string) )`
donde `nodo` **es** `tupla(`***hijos***`: arreglo[256] de puntero(nodo(`σ`) ,` ***significado***`: puntero(`σ`) )`

5.3 Invariante de representación

Informal
(1)Para todo nodo, sus hijos no pueden tenerlo como hijo (2)Todas las hojas tienen significado (3)Para todo elem en claves, elem esta definido en el diccionario y viceversa
Formal

Rep : Arbolito $\longrightarrow$ **bool**

Rep(A) $\equiv$ true $\iff$ (2) ($\forall$ n: nodo)(EstaEnElDicc(n, A) $\Rightarrow$ EsHoja(n)) $\Rightarrow$ n.significado $\neq$ NULL $\wedge$ (3) ($\forall$ s: string)(esta?(s, A.claves)) $\Rightarrow$ Definido(s, A) $\wedge$ ($\forall$ s: string)(Definido(s, A)) $\Rightarrow$ esta?(s, A.claves)
EsHoja : nodo $\longrightarrow$ bool
EsHoja(n) $\equiv$ Vacio?(n.hijos)

5.4 Predicado de abtraccion

Abs : Arbolito A $\longrightarrow$ diccString **{Rep(A)}**
Abs(A) $\equiv$ dicc :Diccionario(Clave, Significado)|
Abs : puntero(nodo) p $\longrightarrow$ dicc(string, σ) **{Rep(p)}**
Abs(p) $\equiv$ d : dicc(string,σ) | $(\forall s : string)\Big(\big(Def?(s,d) \iff (encontrarPalabra(s,p) \neq NULL \wedge_{\text{L}} encontrarPalabra(s,p) \rightarrow significado \neq NULL)\big) \wedge \big(*(encontrarPalabra(s,p) \rightarrow significado) = obtener(s,p)\big)\Big)$

encontrarPalabra : string $\times$ puntero(nodo) $\longrightarrow$ puntero(nodo)
encontrarPalabra(s, p) $\equiv$ if $vacia(s) \vee p = NULL$ **then**
p
else
encontrarPalabra($fin(s)$**,** $p \rightarrow caracteres$**[**$ord(prim(s))$**])**
fi

Representacion del iterador

5.5 Justificacion

El iterador del diccionario utiliza un iterador a lista porque desde el Trie no podemos acceder a la representación interna de claves. Entonces usamos un iterador para poder recorrer esa lista de forma eficiente.

`itString` se representa con `iter`

donde `iter` es `tupla`(*siguiente*: `itLista(string)` , *arbol*: `puntero(arbolito)`)

5.6 Invariante de representacion

Informal (1) El iterador debe corresponderse con la lista de claves del Dicc
Formal
Rep : iterI $\longrightarrow$ bool
Rep(I) $\equiv$ true $\iff$ (1) I.siguiente = CrearIt(*(I.arbol).claves)

5.7 Predicado de abtraccion

Abs : iter I $\longrightarrow$ itString **{Rep(I)}**
Abs(I) $\equiv$ it :Iterador Unidireccional| Siguientes(I) = Siguientes(it)

5.8 Algoritmos

Algoritmos del modulo

---

iVacio() → res:Arbolito)

1: a ← arreglo[256] de puntero(nodo(σ)) ▷ O(256), creo un arreglo vacio de 256 posiciones, que guarda punteros a nodo
2: $res \leftarrow << a, NULL >, Vacio() >$ ▷ O(1)

Complejidad: O(1)
Justificacion: Crea un arreglo vacio, como siempre va a tener 256 posiciones para cualquier entrade la complejidad queda constante, asigna y crea un conjunto vacio

---

iSignificado(**in** s: `string`, **in** d: `Arbolito`) → res: σ)

1: nat i ← Longitud(s) ▷ O(1)
2: puntero(nodo(σ)) p ← d ▷ O(1)
3: **while** i < Longitud(s) **do** ▷ O(|s|), longitud de s
4: p ← p → hijos[ord(s[i])] ▷ O(1)
5: i ← i +1 ▷ O(1)
6: **end while**
7: res ← *(p → significado) ▷ O(1)

Complejidad: O(|s|)
Justificacion: Recorre toda la clave, usando un puntero. Cuando este llega al final devuelve el significado, ya que recorrio toda la clave por el DiccString.

---

iDefinido(**in** s: `string`, **in** d: `Arbolito`) → res: bool

1: nat i ← Longitud(s) ▷ O(1)
2: puntero(nodo(σ)) p ← d ▷ O(1)
3: **while** i < Longitud(s) ∧ p → hijos[ord(s[i])] ≠ NULL **do** ▷ O(|s|), longitud de s
4: p ← p → hijos[ord(s[i])] ▷ O(1)
5: i ← i +1 ▷ O(1)
6: **end while**
7: res ← (i = Longitud(s)) ∧ (p→ significado ≠ NULL) ▷ O(1)

Complejidad: O(|s|)
Justificacion: En el peor de los casos recorre toda la clave para verificar si esta definida o no.

---

iVacio?(**in** d: `Arbolito`) → res: bool

1: bool b ← false ▷ O(1)
2: nat i ← 0 ▷ O(1)
3: **while** i < 256 ∧ ¬b **do** ▷ O(256), termina siendo O(1)
4: **if** d → hijos[i] ≠ NULL **then** ▷ O(1)
5: b ← true ▷ O(1)
6: **end if**
7: **end while**
8: res ← (d = NULL) ∨ (i = 256) ▷ O(1)

Complejidad: O(1)
Justificacion:Chequea si el DiccString es vacio, para esto verifica si la raiz es null o que ningun existe.

---

iDefinir(**in** s: `string`, **in** c: σ, **in/out** d: `Arbolito`)

```
1:  puntero(nodo(σ)) p ← d                                          ▷ O(1)
2:  nat i ← 0                                                       ▷ O(1)
3:  while i < Longitud(s) do                                        ▷ O(|s|), longitud de s
4:      if p→ hijos[ord(s[i])] = NULL then                          ▷ O(1)
5:          puntero(nodo(σ)) p_2 gets NULL                          ▷ O(1)
6:          p→ hijos[ord(s[i])] ← p_2                               ▷ O(1)
7:          p ← p→ hijos[ord(s[i])]                                 ▷ O(1)
8:          i ← i + 1                                               ▷ O(1)
9:      else
10:         p ← p→ hijos[ord(s[i])]                                 ▷ O(1)
11:         i ← i + 1                                               ▷ O(1)
12:     end if
13: end while
14: p→ significado ← c                                              ▷ O(1)
15: AgregarAtras(d.claves, s)                                       ▷ O(1)
```

Complejidad: O(|s|)
Justificacion:Recorre la clave, si existe el nodo solo avanza a este sino crea una nuevo y avanza. Al final asigna el significado y lo agrega al conjunto, como recorre la clave una vez la complejidad final es O(|s|)

iClaves(**in** d: `Arbolito`) → res:lista(string))

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

Complejidad: O(1)
Justificacion:Devuelve el conjunto que es parte de la estructura.

iBorrar(**in** s: `string`, **in/out** d: `Arbolito`)

```
1: puntero(nodo(σ)) aux ← d                                                          ▷ O(1)
   ▷ d apunta a la raíz.
2: puntero(nodo(σ)) padreAux ← d                                                     ▷ O(1)
   ▷ Usamos padreAux y Aux para recorrer la estructura
3: puntero(nodo(σ)) hastaDondeEliminar ← NULL                                        ▷ O(1)
   ▷ hastaDondeEliminar, padreHDE y iHDE se usan para determinar desde dónde hay que eliminar.
4: puntero(nodo(σ)) padreHDE ← NULL                                                  ▷ O(1)
5: nat iHDE ← 0                                                                      ▷ O(1)
6: nat i ← 0                                                                         ▷ O(1)
7: while i < longitud(s) do                                                          ▷ O(|s|)
8:     if aux→significado = NULL ∧ cantidadHijos(aux) = 1 ∧ hastaDondeEliminar = NULL then   ▷ O(1)
9:         hastaDondeEliminar ← aux                                                  ▷ O(1)
10:        padreHDE ← padreAux                                                       ▷ O(1)
11:        iHDE ← i - 1                                                              ▷ O(1)
12:    else
13:        if (cantidadHijos(aux) > 1 ∨ aux→significado ≠ NULL) ∧ (i ≠ longitud(s) -1) then   ▷ O(1)
14:            hastaDondeEliminar ← NULL                                             ▷ O(1)
15:            padreHDE ← NULL                                                       ▷ O(1)
16:            iHDE ← 0                                                              ▷ O(1)
17:        end if
18:    end if
19:    padreAux ← aux                                                                ▷ O(1)
20:    if cantidadHijos(padreAux) > 1 ∧ i = longitud(s)-1 then                       ▷ O(1)
21:        padreHDE ← padreAux                                                       ▷ O(1)
22:        hastaDondeEliminar ← aux→hijos[ord(s[i])]                                 ▷ O(1)
23:        iHDE ← i                                                                  ▷ O(1)
24:    end if
25:    aux ← aux→hijos[ord(s[i])]                                                    ▷ O(1)
26:    i ← i + 1                                                                     ▷ O(1)
27: end while
28: if hastaDondeEliminar = d ∧ cantidadHijos(aux) = 0 then                          ▷ O(1)
29:    borrarNodo(d)                                                                 ▷ O(|s|)
30:    d ← NULL                                                                      ▷ O(1)
31: end if
32: if cantidadHijos(aux) > 0 then                                                   ▷ O(1)
33:    aux→significado ← NULL                                                        ▷ O(1)
34: else
35:    borrarNodo(hastaDondeEliminar)                                                ▷ O(|s|)
36:    padreHDE→hijos[ord(s[iHDE])] ← NULL                                           ▷ O(1)
37: end if
38: itLista(string) itlis ← CrearIt(d.Arbolito)                                      ▷ O(1)
39: while HaySiguiente(itlis) ∧ Siguiente(itlis) ≠ s do                              ▷ O(#(ClavesDicc))
40:    Avanzar(it)                                                                   ▷ O(1)
41: end while
42: if HaySiguiente(it) then                                                         ▷ O(1)
43:    EliminarSiguiente(itlis)                                                      ▷ O(1)
44: end if
```

Complejidad: O(|s| + #(ClavesDicc))
Justificacion:Recorre la clave, guardando cuál es el último nodo "útil" del camino, para eliminar toda la parte de la palabra que vaya a quedar sin una definición asociada.
Si la palabra tiene hijos, sólo se le elimina el significado.
Como en peor caso como máximo se recorre la palabra más larga (varias veces, pero de manera independiente, entonces el orden de la complejidad no aumenta), la complejidad final es O(|s|). Luego busco y elimino la clave de la lista de calves del diccionario

cantidadHijos(**in** p: `puntero(nodo(`σ`))`) → res: nat

```
1: nat res ← 0                                                    ▷ O(1)
2: nat i ← 0                                                      ▷ O(1)
3: while i < 256 do                                               ▷ O(256) ∈ O(1)
4:     if p→hijos[ord(s[i])] ≠ NULL then                          ▷ O(1)
5:         res ← res + 1                                          ▷ O(1)
6:     end if
7:     i ← i + 1                                                  ▷ O(1)
8: end while
```

Complejidad: O(1)
Justificacion: Como el arreglo tiene longitud constante, que no depende del input, recorrerlo siempre va a llevar la misma cantidad de operaciones, entonces es O(1).

borrarNodo(**in** p: `puntero(nodo(`σ`))`) → res: nat

```
1: if p ≠ NULL then                                               ▷ O(1)
2:     nat i ← 0                                                  ▷ O(1)
3:     while i < 256 do                                           ▷ O(256) ∈ O(1)
4:         if p→hijos[ord(s[i])] ≠ NULL then                      ▷ O(1)
5:             borrarNodo(p→hijos[ord(s[i])])                     ▷ O(|s|-1)
6:         end if
7:         i ← i + 1                                              ▷ O(1)
8:     end while
9:     p ← NULL                                                   ▷ O(1)
10: end if
```

Complejidad: O(|s|)
Justificacion: Como el arreglo tiene longitud constante, que no depende del input, recorrerlo siempre va a llevar la misma cantidad de operaciones, es O(1).
La cantidad de veces que llamamos recursivamente a la función es en peor caso la longitud de la palabra más larga del diccionario, porque borrarNodo se llama cuando sólo hay una cadena por borrar. Por lo tanto la complejidad es O(|s|*1) ∈ -O(|s|).

Algoritmos del iterador

iCrearIt(**in** d: `Arbolito`) → res:itString

```
1: itLista(string) it ← CrearIt(*(d.claves))
2: res ← <it, d>                                                  ▷ O(1)
```

Complejidad: O(1)
Justificacion:Crea un iterador al primer elemento de las claves

iHayMas?(**in** it: `itString`) → res:bool

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

Complejidad: O(1)
Justificacion:Chequea si quedan elementos por recorrer

iActual(**in** *it* : `itString`) → res:<string, nat>

1: *res* ← <Siguiente(it.siguiente), Significado(Siguiente(it.siguiente), *(it.arbol)) ▷ O(|P|)

Complejidad: O(|P|)
Justificacion:Devuelve la clave y el significado en una tupla. Para el obtener el significado, busca en el arbol a travez de la clave

iAvanzar(**in/out** *it* : `itString`)

1: Avanzar(it.siguiente) ▷ O(1)

Complejidad: O(1)
Justificacion:Avanza al siguiente elemento a recorrer

iSiguientes(**in** *it* : `itString`) → res:lista(string)

1: itLista(string) itLis ← CrearIt(it.arbol→ claves) ▷ O(1)
2: **while** Siguiente(itLis) ≠ (it.siguiente) **do** ▷ O(#(elementosDeClaves))
3: Avanzar(itLis) ▷ O(1)
4: **end while**
5: lista(string) li ← Vacia() ▷ O(1)
6: **while** (**do**HaySiguiente(itLis)) ▷ O(#(elementosDeClaves))
7: AgregarAtras(li, Siguiente(itLis)) ▷ O(1)
8: Avanzar(itLis) ▷ O(1)
9: **end while**
10: res ← li ▷ O(1)

Complejidad: O(#(elementosDeClaves))
Justificacion:Devuelve los elementos que quedan por recorrer, O(2*#(elementosDeClaves)) = O(#(elementosDeClaves))

6 colaPrioridadMin(σ)

parámetros formales

géneros σ

función $\bullet{=}\bullet$**(in** $s1, s2$**:** σ**)** $\rightarrow$ res **:** σ
Pre $\equiv$ **{true}**
Post $\equiv$ **{**$res =_{\text{obs}} s1 = s2$**}**
Complejidad: $O(1)$
Descripción: función de igualdad de σ's

Se explica con: Cola de Prioridad(α), Iterador Unidireccional Modificable (α).
Géneros: colaPrioridadMin(σ), colaMin(σ), iterCola(σ).

6.1 Interfaz

Operaciones del modulo

Vacia() $\rightarrow$ res **:** `colaPrioridadMin(`σ`)`
Pre $\equiv$ **{true}**
Post $\equiv$ **{res = vacia}**
Complejidad: O(1)
Descripción: Crea una cola de prioridad vacia

EsVacia?(in c**:** `colaPrioridadMin(`σ`)`**)** $\rightarrow$ res **:** `bool`
Pre $\equiv$ **{true}**
Post $\equiv$ **{**$res =_{\text{obs}} vacia?(c)$**}**
Complejidad: O(1)
Descripción: Devuelve verdadero si la lista es vacia.

Proximo(in c**:** `colaPrioridadMin(`σ`)`**)** $\rightarrow$ res **:** σ
Pre $\equiv$ **{**$\neg EsVacia?(x)$**}**
Post $\equiv$ **{**$res =_{\text{obs}} proximo(c)$**}**
Complejidad: O(1)
Descripción: Devuelve el valor del primer elemento de la cola de prioridad

Encolar(in/out c**:** `colaPrioridadMin(`σ`)`**, in** k**:** κ**, in** s**:** β**)** $\rightarrow$ res **:** `iterColaMin`
Pre $\equiv$ **{true}**
Post $\equiv$ **{**$res =_{\text{obs}} encolar(x, c)$**}**
Complejidad: O(log(n))
Descripción: Agrega un elemento a la cola de prioridad. Devuelve un iterador apuntando al elemento insertado

Desencolar(in/out c**:** `colaPrioridadMin(`σ`)`**)**
Pre $\equiv$ **{**$\neg EsVacia?(x)$**}**
Post $\equiv$ **{**$res =_{\text{obs}} desencolar(c)$**}**
Complejidad: O(log(n))
Descripción: Elimina el proximo elemento de la cola de prioridad.

Borrar(in/out c**:** `colaPrioridadMin(`σ`)`**, in** it**:** `iterCola`**)**
Pre $\equiv$ **{c** $=_{\text{obs}}$ **c**$_0$ **}**
Post $\equiv$ **{c** $=_{\text{obs}}$ **Borrar(c, it)}**
Complejidad: O(log n), siendo n la cantidad de elementos de la cola
Descripción: Borrar el elemento apuntado por el iterador, y rearma la cola

Operaciones del iterador

CrearIt(in c: colaMin) $\rightarrow$ res : iterCola
Pre $\equiv$ **{true}**
Post $\equiv$ **{res** $=_{obs}$ **crearItMod(<>, c)** $\wedge$ **alias(SecuSuby(it)** $=$ **l)}**
Complejidad: O(1)
Descripción: Crea un iterador a la cola a la raiz de la cola

HayMas?(in it: iterCola) $\rightarrow$ res : bool
Pre $\equiv$ **{true}**
Post $\equiv$ **{res** $=_{obs}$ **HayMas?(it)}**
Complejidad: O(1)
Descripción: Chequea que el iterador este apuntando a un elemento

Actual(in it: iterCola) $\rightarrow$ res : nat
Pre $\equiv$ **{HayMas?(it)}**
Post $\equiv$ **{res** $=_{obs}$ **Actual(it)}**
Complejidad: O(1)
Descripción: Devuelve el jugador apuntado por el iterador

Representación

6.2 Justificacion

Representacion del modulo

Implementamos la cola de prioridad sobre minHeap para poder tener acceso en O(1) al primer de elemento de la cola y tener las operaciones de encolar y desencolar en O(log(n)).
Como queremos poder agregar y quitar una arbitraria cantidad de elementos, no podemos usar un arreglo para representar el heap, dado que para extender el arreglo tendríamos una complejidad mayor a la pedida.
Lo representamos sobre un árbol binario completo izquierdista, cumpliendo la complejidad de heap.
El principal uso de la cola de prioridad será para determinar qué jugador captura al pokemon, la prioridad será: aquel que tenga la menor cantidad de pokemons tendrá la mayor prioridad.
No nos interesa buscar en la estructura de manera eficiente (la complejidad de la búsqueda es O(n)) pero sí nos interesa obtener el primer elemento en O(1).

colaPrioridadMin(σ) **se representa con** colaMin

donde colaMin **es** tupla(***raiz***: puntero(nodoHeap(σ)) , ***ultimoAgregdo***: puntero(nodoHeap(σ))))
donde nodoHeap **es** tupla(***padre***: puntero(nodoHeap(σ)) , ***izq***: puntero(nodoHeap(σ)) , ***der***: puntero(nodoHeap(σ)) , ***clave***: κ , ***significado***: β)

6.3 Invariante de representación

Informal
(1)Para todo nodo, sus hijos no pueden tenerlo como hijo.
(2)La raíz no tiene padre.
(3)Árbol binario perfectamente balanceado e izquierdista.
(4)La clave de cada nodo es menor o igual a la de sus hijos (si los tiene).
(5)Todo subárbol es un heap.

Formal
Rep : colaMinC $\longrightarrow$ **bool**

Rep(C) $\equiv$ true $\iff$ C.raiz = NULLL $\vee$ ((2) C.raiz->padre = NULL $\wedge$ (3) ($\forall$ p: nodoHeap(σ)) p.der $\neq$ NULL $\Rightarrow$ p.izq $\neq$ NULL $\wedge$ (4) ($\forall$ p: nodoHeap(σ))(((p.izq $\neq$ NuLL) $\Rightarrow_{\text{L}}$ p.clave > p.izq->clave) $\wedge$ ((p.der $\neq$ NULL) $\Rightarrow_{\text{L}}$ p.clave > p.der->clave)) $\wedge$ (5) Rep(C.raiz->izq) $\wedge$ Rep(C.raiz->der))

6.4 Predicado de abstracción

Abs : colaM C $\longrightarrow$ colaMin **{Rep(C)}**
Abs(C) $\equiv$ cola :Cola(σ)| EsVacia?(C) = vacia(cola) $\vee_{\text{L}}$ (Proximo(C) = proximo(cola) $\wedge$ Desencolar(C) = desencolar(cola))

6.5 Representación del iterador

6.6 Justificación

El iterador nos permite acceder a cualquier elemento del heap en tiempo constante, sin necesidad de hacer búsqueda exhaustiva. En particular nos interesa eliminar un elemento (cuando un jugador se aleja del pokemon) en tiempo O(log(n)). Este iterador en particular no tiene función avanzar porque sólo nos interesa que apunte siempre al mismo nodo.

`iterCola` **se representa con** `iter`

donde `iter` **es** `tupla(`***Siguiente***`: puntero(nodoHeap) )`

6.7 Invariante de representación

Informal

(1) True

Formal

Rep : iterColaIt $\longrightarrow$ bool
Rep(It) $\equiv$ true $\iff$ true

6.8 Predicado de abstraccion

Abs : iter I $\longrightarrow$ iterCola **{Rep(I)}**
Abs(I) $\equiv$ it :iterador Unidireccional| Actual(I) $\bullet$ <> = Siguientes(it)

6.9 Algoritmos

Funciones Auxiliares, no exportadas

BuscarPadreInsertar(in/out c: `colaPrioridadMin`)
Pre $\equiv$ {true}
Post $\equiv$ {Conserva la propiedad de heap}
Complejidad: O(log(n))
Descripción: Encuentra el nodo al cual hay que insertarle el próximo hijo

SiftUp(in/out c: `colaPrioridadMin`, in p: `puntero`)
Pre $\equiv$ {$\neg EVacia?(Desencolar(c))$}
Post $\equiv$ {No necesariamente conserva la propiedad de heap}
Complejidad: O(log(n))
Descripción: Mueve hacia arriba en la cola de prioridad a p si su padre es de menor prioridad

SiftDown(in/out c: `colaPrioridadMin`, in p: `puntero`)
Pre $\equiv$ {$\neg EVacia?(Desencolar(c))$}

Post ≡ {No necesariamente conserva la propiedad de heap}
Complejidad: O(log(n))
Descripción: Mueve hacia abajo en la cola de prioridad a p si alguno de sus hijos es de mayor prioridad

Swap(in/out c**:** `colaPrioridadMin`**, in** p**:** `puntero`**, in** q**:** `puntero`**)**
Pre ≡ {$\neg EVacia?(Desencolar(c))$**}**
Post ≡ {No necesariamente conserva la propiedad de heap}
Complejidad: O(1)
Descripción: Toma dos nodos del heap e intercambia sus posiciones.

TieneQueSubir?(in/out c**:** `colaPrioridadMin`**, in** p**:** `puntero`**)** $\rightarrow$ res **:** `bool`
Pre ≡ {$\neg EVacia?(Desencolar(c))$**}**
Post ≡ {No necesariamente conserva la propiedad de heap}
Complejidad: O(1)
Descripción: Verifica si hay que hacer siftUp para reestablecer la propiedad del heap

TieneQueBajar?(in/out c**:** `colaPrioridadMin`**, in** p**:** `puntero`**)** $\rightarrow$ res **:** `bool`
Pre ≡ {$\neg EVacia?(Desencolar(c))$**}**
Post ≡ {No necesariamente conserva la propiedad de heap}
Complejidad: O(1)
Descripción: Verifica si hay que hacer siftDown para reestablecer la propiedad del heap

Borrar(in/out c**:** `colaPrioridadMin`**, in** p**:** `puntero`**)**
Pre ≡ {$\neg EVacia?(c)$**}**
Post ≡ {No necesariamente conserva la propiedad de heap}
Complejidad: O(1)
Descripción: Elimina un nodo del árbol

Eliminar(in/out it**:** `iterCola`**)**
Pre ≡ {HayMas?(it) $\wedge$ **it = it**$_0$**}**
Post ≡ {it = Eliminar(it$_0$**)}**
Complejidad: O(1)
Descripción: Borra el nodo apuntado por el iterador y pone el iterador a NULL

Algoritmos

---

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

Complejidad: O(1)
Justificacion: Crea un heap vacio, como siempre va a tener que crear una raíz vacía, la complejidad queda constante.

---

iEsVacia?(**in** r**:** `colaMin`) $\rightarrow$ res: bool)
1: $res \leftarrow r.raiz = NULL$ ▷ O(1)

Complejidad: O(1)
Justificacion: Hace una comparación con la raíz para verificar que la cola de prioridad esté vacía.

---

iEncolar(**in/out** c: `colaMin`, **in** k: κ, **in** s: β) → res: iterColaMin

```
1: puntero(nodoHeap(σ)) insertado
2: if c.raiz = NULL then
3:     c.raiz← insertado
4:     c.ultimo ← insertado
5: end if
6: dondeAgregar ← buscarPadreInsertar(c, ultimo)                ▷ O(log(n))
7: if dondeAgregar→izq = NULL then
8:     dondeAgrega→izq ← insertado
9: else
10:    if dondeAgregar→der = NULL then
11:        dondeAgregar→der ← insertado
12:    end if
13: end if
14: insertado→padre ← dondeAgregar
15: c.ultimo ← insertado
16: SiftUp(c, insertado)                                        ▷ O(log(n))
17: res ← iterador(insertado)
```

Complejidad: O(log(n))
Justificacion: Se agrega un elemento al fondo del heap, luego se utiliza la función sift-up para volver a la propiedad de heap. Ocurren varias operaciones independientes que tienen complejidad O(log(n)), por lo tanto tiene una complejidad O(log(n)).

iDesencolar(**in/out** c: `colaMin`)

```
1: if c.raiz→izq = NULL then
2:     c.raiz = NULL
3:     c.ultimo = NULL
4: else
5:     puntero(nodoHeap(σ)) nuevoUltimo ← buscarUltimo(c.ultimo)
6:     puntero(nodoHeap(σ)) aBorrar ← c.raiz
7:     swap(c.raiz, c.ultimo)                                   ▷ O(1)
8:     if c.ultimo→padre→der = aBorrar then
9:         c.ultimo→padre→der ← NULL
10:    else
11:        c.ultimo→padre→izq ← NULL
12:    end if
13:    c.ultimo ← nuevoUltimo
14:    SiftDown(c.raiz)                                         ▷ O(log(n))
15: end if
```

Complejidad: O(log(n))
Justificacion: Se elimina el primer elemento del heap, se reemplaza la raiz por el último elemento, luego se utiliza la función sift-down para volver a tener la propiedad de heap. El ciclo itera como máximo log(n) veces, que es la altura del heap (sólo se puede llevar hasta el fondo).

iProximo(**in** c: `colaMin`) → res: σ

```
1: res←c→raiz.significado
```

Complejidad: O(1)
Justificación: Se obtiene el significado de la raiz del heap.

iBorrar(**in/out** c: `colaPrioridadMin(`σ`)`, **in/out** p: `iterCola`

```
1:  puntero(nodoHeap(σ)) ultimo ← NULL                                          ▷ O(1)
2:  if c.padreAgregar→der = NULL then                                           ▷ O(1)
3:      ultimo← c.padreAgregar→izq                                              ▷ O(1)
4:      c.padreAgregar→izq←NULL                                                 ▷ O(1)
5:  else
6:      ultimo ← c.padreAgregar→der                                             ▷ O(1)
7:      c.padreAgregar→der←NULL                                                 ▷ O(1)
8:  end if
9:  ultimo→izq←p→izq                                                            ▷ O(1)
10: ultimo→der←p→der                                                            ▷ O(1)
11: ultimo→padre←p→padre                                                        ▷ O(1)
12: while ultimo→der ≠ NULL ∧ ultimo→izq ≠ NULL ∧ (ultimo→der.clave < ultimo.clave ∨ ultimo→izq.clave <
    ultimo.clave)) do                                                           ▷ O(log(n))
13:     siftDown(ultimo, c)                                                     ▷ O(1)
14: end while
15: Eleminar(p)                                                                 ▷ O(1)
```

Complejidad: O(log(n))
Justificacion: Muy similar a iDesencolar(), pero eliminando el puntero del parámetro.
Se elimina el elemento apuntado por el iterador, se reemplaza por el último elemento, luego se utiliza la función sift-down para volver a tener la propiedad de heap. El ciclo itera como máximo log(n) veces, que es la altura del heap (sólo se puede llevar hasta el fondo).

tieneQueSubir(**in** c: `colaPrioridadMin(`σ`)`, **in** n: `puntero(nodoHeap(`σ`))`) → res: Bool

```
1: if n→padre = NULL then
2:     res ← False
3: else
4:     res ← n→valor < n→padre→valor
5: end if
```

Complejidad: O(1)
Justificacion:Una pequeña función para determinar si hay que hacer siftUp.

tieneQueBajar(**in** c: `colaPrioridadMin(`σ`)`, **in** n: `puntero(nodoHeap(`σ`))`) → res: Bool

```
1: if n→izq = NULL ∧ n→der = NULL then
2:     res ← False
3: else
4:     if n→der != NULL ∧ n→der→valor < n→valor then
5:         res ← True
6:     else
7:         res ← n→izq→valor < n→valor
8:     end if
9: end if
```

Complejidad: O(1)
Justificacion:Una pequeña función para determinar si hay que hacer siftDown.

swap(in c: `colaPrioridadMin(`σ`)`, **in** $nodo1$: `puntero(nodoHeap(`σ`))`, **in** $nodo2$: `puntero(nodoHeap(`σ`))`)

```
 1: n1 ← nodo1                                                          ▷ Swap es O(1)
 2: n1Padre ← nodo1→padre
 3: n1Der ← nodo1→der
 4: n1Izq ← nodo1→izq
 5: n2 ← nodo2
 6: n2Padre ← nodo2→padre
 7: n2Der ← nodo2→der
 8: n2Izq ← nodo2→izq
                                                                        ▷ Cambiamos los padres
 9: if nodo1→padre = nodo2→padre then
10:     No hacer nada, dejarlos con el mismo padre
11: else
12:     if nodo1→padre != nodo2 ∧ nodo2→padre != nodo1 then
13:         nodo1→padre ← n2Padre
14:         nodo2→padre ← n1Padre
15:         if nodo1→izq != NULL then
16:             nodo1→der→padre ← nodo2
17:         end if
18:         if nodo1→der != NULL then
19:             nodo1→izq→padre ← nodo2
20:         end if
21:         if nodo2→izq != NULL then
22:             nodo1→izq→padre ← nodo1
23:         end if
24:         if nodo2→der != NULL then
25:             nodo1→der→padre ← nodo1
26:         end if
27:     else
28:         if nodo2→padre = nodo1 then
29:             nodo1→padre ← n2
30:             nodo2→padre ← n1Padre
31:             if nodo1→izq != NULL ∧ nodo1→izq != nodo2) then
32:                 nodo1→izq→padre ← nodo2
33:             end if
34:             if nodo1→der != NULL ∧ nodo1→der != nodo2 then
35:                 nodo1→der→padre ← nodo2
36:             end if
37:             if nodo2→izq != NULL then
38:                 nodo2→izq→padre ← nodo1
39:             end if
40:             if nodo2→der != NULL then
41:                 nodo2→der→padre ← nodo1
42:             end if
43:         else
44:             if nodo1→padre = nodo2 then
45:                 nodo1→padre ← n2Padre
46:                 nodo2→padre = n1
47:                 if nodo1→izq != NULL then
48:                     nodo1→izq→padre ← nodo2
49:                 end if
50:                 if nodo1→der != NULL then
51:                     nodo1→der→padre ← nodo2
52:                 end if
53:                 if nodo2→izq != NULL then
54:                     nodo2→izq→padre ← nodo1
55:                 end if
56:                 if nodo2→der != NULL then
57:                     nodo2→der→padre ← nodo1
58:                 end if
59:             end if
60:         end if
                                                                        ▷ Apuntamos los nuevos hijos de cada nodo
61:         nodo1→izq ← n2Izq
62:         if nodo1→izq != NULL ∧ nodo1→izq = nodo1 then
63:             nodo1→izq ← nodo2
64:         end if
65:         nodo1→der ← n2Der
66:         if nodo1→der != NULL ∧ nodo1→der = nodo1 then
67:             nodo1→der ← nodo2
68:         end if
69:         nodo2→izq ← n1Izq
70:         if nodo2→izq != NULL ∧ nodo2→izq = nodo2 then
71:             nodo2→izq ← nodo1
72:         end if
73:         nodo2→der ← n1Der
74:         if nodo2→der != NULL ∧ nodo2→der = nodo2 then
75:             nodo2→der ← nodo1
76:         end if
                                                                        ▷ Tenemos que actualizar las referencias de los padres
77:         if nodo1→padre = nodo2→padre then
78:             izqAux ← nodo1→padre→izq
79:             nodo1→padre→izq ← nodo1→padre→der
80:             nodo1→padre→der ← izqAux
81:         else
82:             if n1Padre != NULL ∧ n1Padre→izq = n1 then
83:                 n1Padre→izq ← n2
84:             end if
85:             if n1Padre != NULL ∧ n1Padre→der = n1 then
86:                 n1Padre→der ← n2
87:             end if
88:             if n2Padre != NULL ∧ n2Padre→izq = n2 then
```

BuscarPadreInsertar(**in** c: `colaPrioridadMin(`σ`)`, **in** n: `puntero(nodoHeap(`σ`))`) → res :puntero(nodoHeap(σ))

```
1: if n = c→raiz then
2:     res ← n
3: else
4:     if nodo→padre→der = NULL then
5:         res ← n→padre
6:     else
7:         aux ← n→padre
8:         while aux != c→raiz ∧ aux→padre→der = aux do                ▷ O(log(n))
9:             aux ← aux→padre
10:        end while
11:        if aux != c→raiz then
12:            aux ← aux→padre→der
13:        end if
14:        while aux→izq != NULL do                                    ▷ O(log(n))
15:            aux ← aux→izq
16:        end while
17:        res ← aux
18:    end if
19: end if
```

Complejidad: O(log(n))
Justificacion: En el peor caso se recorre dos veces la altura del heap, es decir 2log(n) operaciones, dando una complejidad O(log(n)).

---

buscarUltimoAlDesencolar(**in** c: `colaPrioridadMin(`σ`)`, **in** n: `puntero(nodoHeap(`σ`))`) → res :puntero(nodoHeap(σ))

```
1: if n→padre→der = n then
2:     res ← n→padre→izq
3: elseaux ← n→padre
4:     while aux != c→raiz ∧ aux→padre→izq = aux do                    ▷ O(log(n))
5:         aux ← aux→padre
6:     end while
7:     if aux != c→raiz then
8:         aux ← aux→padre→izq
9:     end if
10:    while aux→der != NULL do                                        ▷ O(log(n))
11:        aux ← aux→der
12:    end while
13:    res ← aux
14: end if
```

Complejidad: O(log(n))
Justificacion:Similar a BuscarPadreInsertar. En el peor caso se recorre dos veces la altura del heap, es decir 2log(n) operaciones, dando una complejidad O(log(n)).

---

siftUp(**in** c: `colaPrioridadMin(`σ`)`, **in** n: `puntero(nodoHeap(`σ`))`)

```
1: while tieneQueSubir(n) do                                           ▷ O(log(n))
2:     swap(n, n→padre)                                                ▷ O(1)
3: end while
```

Complejidad: O(log(n))
Justificacion:Se intercambia un nodo con uno de sus hijos que tenga clave menor. Como mucho puede recorrer toda la altura del árbol, que es log(n).

siftDown(**in** c: `colaPrioridadMin(`σ`)`, **in** n: `puntero(nodoHeap(`σ`))`)

```
1: aBajar ← nodoEvaluado
2: while tieneQueBajar(aBajar) do                                   ▷ O(log(n))
3:     if aBajar→der = NULL then
4:         swap(aBajar→izq, aBajar)
5:     else
6:         if aBajar→izq != NULL ∧ aBajar→der != NULL then
7:             if aBajar→izq→valor < aBajar→der→valor then
8:                 swap(aBajar→izq, aBajar)
9:             else
10:                swap(aBajar→der, aBajar)
11:            end if
12:        end if
13:    end if
14: end while
```

Complejidad: O(log(n))
Justificacion:Se intercambia un nodo con su padre. Como mucho puede recorrer toda la altura del árbol, que es log(n).

Algoritmos del iterador

iCrearIt(**in** c: `colaMin`) → res: iter

```
1: res← <c.raiz>                                                    ▷ O(1)
```

Complejidad: O(1)
Justificación:Creo un iterador que apunta a raiz de la cola

iEliminar(**in/out** it: `iter`)

```
1: it.siguiente→ padre ← NULL                                       ▷ O(1)
2: it.siguiente→ izq ← NULL                                         ▷ O(1)
3: it.siguiente→ der ← NULL                                         ▷ O(1)
4: it.siguiente ← NULL                                              ▷ O(1)
```

Complejidad: O(1)
Justificación:Primero me con el iterador borro la relación con el padre y de los hijos, despues pongo a NULL al iterador para eliminar un nodo, nota esta funcion nunca sucede por si sola, siempre es llamada desde Borrar de la cola

iHayMas?(**in** it: `iter`) → res: bool

```
1: res ← it.siguiente ≠ NULL                                        ▷ O(1)
```

Complejidad: O(1)
Justificación: Veo si el iterador esta apuntando a algo valido

iActual(**in** it: `iter`) → res: nat

```
1: res ← it.siguiente->significado                                  ▷ O(1)
```

Complejidad: O(1)
Justificación:Devuelve la ID del jugador al que apunta el iterador