

Algoritmos y Estructuras de Datos II

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

Trabajo Práctico N°2

Flanders y Asociados

Integrante	LU	Correo electrónico
Sabogal, Patricio	693/14	pato.sabogal@hotmail.com
Guralnik, Ivan	235/14	ivanstng@gmail.com
Baldonado, Juan Manuel	693/15	juanmanuelbaldonado@gmail.com
Carbonelli, Lucas	596/15	luc92a@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega	Matías	P-
Segunda entrega		

Índice

1. Módulo Coordenada	3
2. Módulo Mapa	6
3. Renombres de TADs	11
4. Módulo Heap Modificable	11
5. Módulo Diccionario en Cadena(<i>string</i> , σ)	20
6. Módulo Juego	28

1. Módulo Coordenada

Interfaz

parámetros formales

géneros α

función $\text{COPIAR}(\text{in } a : \alpha) \rightarrow res : \alpha$

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

Post $\equiv \{res =_{\text{obs}} a\}$

Complejidad: $\Theta(\text{copy}(a))$

Descripción: función de copia de α 's

se explica con: COORDENADA.

generos: coordenada.

Operaciones básicas de Coordenada

$\text{CREARCOOR}(\text{in } n : \text{Nat}, \text{in } n' : \text{Nat}) \rightarrow res : \text{Coordenada}$

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

Post $\equiv \{\text{longitud}(res) = n \wedge \text{latitud}(res) = n'\}$

Complejidad: $\Theta(1)$

Descripción: genera una coordenada con longitud y latitud pasadas por parametro.

$\text{LONGITUD}(\text{in } c : \text{Coordenada}) \rightarrow res : \text{Nat}$

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

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

Complejidad: $\Theta(1)$

Descripción: devuelve el valor correspondiente a la longitud de la coordenada.

$\text{LATITUD}(\text{in } c : \text{Coordenada}) \rightarrow res : \text{Nat}$

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

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

Complejidad: $\Theta(1)$

Descripción: devuelve el valor correspondiente a la latitud de la coordenada.

$\text{DISTEUCLIDEA}(\text{in } c : \text{Coordenada}, \text{in } c' : \text{Coordenada}) \rightarrow res : \text{Nat}$

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

Post $\equiv \{res = (\text{latitud}(c) - \text{latitud}(c')) * (\text{latitud}(c) - \text{latitud}(c')) + (\text{longitud}(c) - \text{longitud}(c')) * (\text{longitud}(c) - \text{longitud}(c'))\}$

Complejidad: $\Theta(1)$

Descripción: devuelve la distancia euclidiana.

$\text{COORDENADAARRIBA}(\text{in } c : \text{Coordenada}) \rightarrow res : \text{Coordenada}$

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

Post $\equiv \{\text{latitud}(res) = \text{latitud}(c) + 1 \wedge \text{longitud}(res) = \text{longitud}(c)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve la coordenada de arriba.

$\text{COORDENADAABAJO}(\text{in } c : \text{Coordenada}) \rightarrow res : \text{Coordenada}$

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

Post $\equiv \{\text{latitud}(res) = \text{latitud}(c) - 1 \wedge \text{longitud}(res) = \text{longitud}(c)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve la coordenada de arriba.

$\text{COORDENADAALADERECHA}(\text{in } c : \text{Coordenada}) \rightarrow res : \text{Coordenada}$

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

Post $\equiv \{\text{latitud}(res) = \text{latitud}(c) \wedge \text{longitud}(res) = \text{longitud}(c) + 1\}$

Complejidad: $\Theta(1)$

Descripción: devuelve la coordenada de arriba.

COORDENADAALAIZQUIERDA(**in** c : Coordenada) $\rightarrow res$: Coordenada
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{latitud}(res) = \text{latitud}(c) \wedge \text{longitud}(res) = \text{longitud}(c) - 1\}$
Complejidad: $\Theta(1)$
Descripción: devuelve la coordenada de arriba.

Representación

Representacion de la Coordenada

Coordenada se representa con coord

donde coord es `tupla(latitud: nat, longitud: nat)`

$\text{Rep} : \text{coord} \rightarrow \text{bool}$

$\text{Rep}(c) \equiv \text{true} \iff$

0

$\text{Abs} : \text{coord } c \rightarrow \text{coordenada}(\text{Nat})$

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

$\text{Abs}(c) \equiv \text{crearCoord}(c.\text{latitud}, c.\text{longitud})$

Algoritmos

Algoritmos del modulo

iCrearCoord(**in** n : Nat, **in** n' : Nat) $\rightarrow res$: Coordenada

1: $res \leftarrow \langle n, n' \rangle$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iLatitud(**in** c : coord) $\rightarrow res$: Nat

$res \leftarrow c.\text{Latitud}$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iLongitud(**in** c : coord) $\rightarrow res$: Nat

$res \leftarrow c.\text{Longitud}$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iDistEuclidea(in $c : \text{coord}$, in $c' : \text{coord}$) $\rightarrow res : \text{Nat}$

if $c.\text{Longitud} > c'.\text{Longitud}$ **then** $\triangleright \Theta(1)$
 $long \leftarrow (c.\text{Longitud} - c'.\text{Longitud}) * (c.\text{Longitud} - c'.\text{Longitud})$ $\triangleright \Theta(1)$

else $long \leftarrow (c'.\text{Longitud} - c.\text{Longitud}) * (c'.\text{Longitud} - c.\text{Longitud})$ $\triangleright \Theta(1)$
end if

if $c.\text{Latitud} > c'.\text{Latitud}$ **then** $\triangleright \Theta(1)$
 $lat \leftarrow (c.\text{Latitud} - c'.\text{Latitud}) * (c.\text{Latitud} - c'.\text{Latitud})$ $\triangleright \Theta(1)$

else $lat \leftarrow (c'.\text{Latitud} - c.\text{Latitud}) * (c'.\text{Latitud} - c.\text{Latitud})$ $\triangleright \Theta(1)$
end if
 $res \leftarrow lat + long$

Complejidad: $\Theta(1)$

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

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

Complejidad: $\Theta(1)$

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

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

Complejidad: $\Theta(1)$

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

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

Complejidad: $\Theta(1)$

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

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

Complejidad: $\Theta(1)$

2. Módulo Mapa

Interfaz

parámetros formales

géneros α

función **COPIAR**(in $a : \alpha$) $\rightarrow res : \alpha$

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

Post $\equiv \{res =_{\text{obs}} a\}$

Complejidad: $\Theta(\text{copy}(a))$

Descripción: función de copia de α 's

se explica con: MAPA.

generos: map.

Operaciones basicas de Mapa

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

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

Post $\equiv \{(\text{hayAnterior}(\text{coordenadas}(res))) \wedge (\text{haySiguiente}(\text{coordenadas}(res)))\}$

Complejidad: $\Theta(1)$

Descripción: Devuelve una instancia de mapa sin coordenadas

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

Pre $\equiv \{m =_{\text{obs}} m_0\}$

Post $\equiv \{\text{secuSuby}(\text{coordenadas}(res)) =_{\text{obs}} c \bullet \text{secuSuby}(\text{coordenadas}(m_0))\}$

Complejidad: $\Theta(1)$

Descripción: Agrega una nueva coordenada al mapa

Aliasing: El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin utilizar la funcion ELIMINARSIGUIENTE.

COORDENADAS(in/out $m : \text{map}$) $\rightarrow res : \text{itConj}(\text{coordenada})$

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

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

Complejidad: $\Theta(1)$

Descripción: Devuelve un iterador al conjunto de coordenadas del mapa.

Aliasing: El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin utilizar la funcion ELIMINARSIGUIENTE.

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

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

Post $\equiv \{res =_{\text{obs}} \text{esta?}(c, \text{secuSuby}(\text{coordenadas}(m)))\}$

Complejidad: $\Theta(1)$

Descripción: Devuelve true si hay la coordenada se encuentra en el mapa y false de lo contrario.

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

Pre $\equiv \{\text{esta?}(c, \text{secuSuby}(\text{coordenadas}(m))) \wedge \text{esta?}(c', \text{secuSuby}(\text{coordenadas}(m)))\}$

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

Complejidad: $\Theta(??)$

Descripción: Devuelve true si hay un camino entre las 2 coordenadas.

EXISTECAMINO(in $c : \text{coord}$, in $c' : \text{coord}$, in $cs : \text{conj}(\text{coord})$, in $m : \text{map}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{esta?}(c, \text{secuSuby}(\text{coordenadas}(m))) \wedge \text{esta?}(c', \text{secuSuby}(\text{coordenadas}(m))) \wedge cs \subseteq \text{coordenadas}(m)\}$

Post $\equiv \{res =_{\text{obs}} \text{existeCamino}(c, c', cs, m)\}$

Complejidad: $\Theta(??)$

Descripción: Devuelve true si hay un camino entre las 2 coordenadas.

Representación

Representacion del Mapa

El modulo se representa como un vector de vectores .El vector externo cooresponde a la latitud y los vectores dentro de este a la longitud. Detro de cada posicion del vector interno "se encuentra una tupla. El primer valor de la tupla corresponde a un booleano cuyo valor es true si y solo si la coordenada se encuentra en el mapa.El segundo valor de la tupla corresponde a un conjunto con las coordenadas del mapa las cuya distancia euclidean con la primera es menor que 100 y existe un camino que las conecta.Este conjunto nos permite determinar si hay un camino entre dos coordenadas cuya distancia euclidean es menor que 100 en tiempo constante , para el resto de los casos la complejidad depende del tamaño del mapa.Notar que por como lo se agregan coordenadas al conjunto en la funcion agregarCoordenada las coordenadas dentro de este pueden tener un camino cuya "distancia"sea mayor a 100 , es decir , el camino no necesariamente esta contenido dentro de un rango distancia 10 sino que puede ser cualquiera. El modulo cuenta ademas con un conjunto de coordenadas de contiene las coordenadas que se encuentran en el mapa.

Mapa se representa con map

donde map es tupla(columnas: vector(fila), Coordenadas: ConjuntoLineal(Coordenada))

donde fila es tupla(fila: vector(info))

donde info es tupla(esta: bool, adyacentes: conjuntoLineal(coord))

Rep : mapa \rightarrow bool

$$\text{Rep}(m) \equiv \text{true} \iff \text{vacío}(m.\text{coordenas}) \iff \text{longitud}(m.\text{columnas}) = 0 \wedge \neg \text{vacío}(m.\text{coordenas}) \Rightarrow_L$$

$$((\forall c : \text{coord}) \text{pertenece}(m.\text{Coordenadas}, c) \iff (\text{Longitud}(c) \leq \text{Longitud}(m.\text{columnas})) \wedge_L (\text{Latitud}(c) \leq \text{Longitud}(m.\text{columnas}[\text{Longitud}(c)]))$$

$$\wedge_L m.\text{columnas}[\text{Longitud}(c)][\text{Latitud}(c)].\text{esta} = \text{false}) \wedge (\forall c' : \text{coord}) \text{pertenece}(m.\text{Coordenadas}, c') \wedge$$

$$\text{distEuclidea}(c, c') < 10 \Rightarrow_L \text{pertenece}(m.\text{coordenadas}[\text{Longitud}(c)][\text{Latitud}(c)].\text{adyacentes}, c') \wedge$$

$$\text{pertenece}(m.\text{coordenadas}[\text{Longitud}(c')][\text{Latitud}(c')].\text{adyacentes}, c)$$

maximo : conj(coord) cs \rightarrow coord

dameMapa(cs) \equiv if vacia(s) then m else dameMapa(fin(s),AgregarColumna(prim(s),m,0) ,i + 1) fi

Abs : map m \rightarrow Mapa

{Rep(m)}

Abs(m) \equiv dameMapa(abs(abs(m.filas)) , crearMapa(),0)

dameMapa : secu(secu(bool)) s \times map m \times nat i \rightarrow map

dameMapa(s,m,i) \equiv if vacia(s) then m else dameMapa(fin(s), AgregarColumna(prim(s),m,0),i + 1) fi

AgregarColumna : secu(bool) s \times map m \times nat j \rightarrow map

AgregarColumna(s,m,j) \equiv if vacia(s) then

m

else

if prim(s) then

AgregarColumna(fin(s),AgregarCoord(crearCoord(i,j),m),j + 1)

else

AgregarColumna(fin(s),m,j + 1)

fi

fi

Algoritmos

Algoritmos del modulo

iCrearMapa() \rightarrow res : map

1: res \leftarrow <iVacía(),0>

$\triangleright \mathcal{O}(1)$

Complejidad: Solo se inicializa una lista vacia por lo cual la complejidad es $\mathcal{O}(1)$

icoordenadas(in $m : \text{map}$) $\rightarrow res : \text{it Conj}(\text{coordendas})$
1: $res \leftarrow \langle \text{CrearIt}(m.\text{coordendas}) \rangle$ \triangleright creo un iterador a conjunto de coordenadas $\mathcal{O}(1)$ Complejidad: $\Theta(1)$

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

1:

2: **if** Longitud(c) $> m.\text{longitud}$ **then** $\triangleright \mathcal{O}(1)$

3:

4: **if** Latitud(c) $> (m.\text{columnas}[c.\text{Longitud}]).\text{longitud}$ **then** $\triangleright \mathcal{O}(1)$ 5: $res \leftarrow (m.\text{columnas}[\text{Longitud}(c)])[\text{Latitud}(c)].\text{esta}$ \triangleright acceso a un arreglo de arreglos $\mathcal{O}(1)$ 6: **end if**7: **else** $res \leftarrow \text{false}$ 8: **end if**

Complejidad: acceder una posicion de un arreglo tiene complejidad $\mathcal{O}(1)$. Solo realizan 2 accesos a arreglos por lo cual la complejidad es $\mathcal{O}(1)$

iAgregarCoor(in $c : \text{coord}$, in/out $m : \text{map}$) $\rightarrow res : \text{map}$

1:

2: **if** longitud($m.\text{columnas}$) $< \text{Longitud}(c)$ **then**3: $i \leftarrow \text{longitud}(m.\text{columnas})$ $\triangleright \Theta(1)$ 4: **while** $i \leq \text{Longitud}(c)$ **do** $\triangleright \mathcal{O}(\text{Longitud}(c))$ 5: $\text{Agregar}(m.\text{columnas}, i, i\text{Vacio}())$ \triangleright se agrega en la posicion i-esima del vector un arreglo vacio $\mathcal{O}(\text{longitud}(m.\text{columnas}))$ 6: $i \leftarrow i + 1$ $\triangleright \mathcal{O}(1)$ 7: **end while**8: **end if**

9:

10: **if** longitud($m.\text{columnas}[\text{Longitud}(c)]$) $< \text{Latitud}(c)$ **then**11: $j \leftarrow \text{longitud}(m.\text{columnas}[\text{Longitud}(c)])$ \triangleright acceso a un arreglo $\mathcal{O}(1)$ 12: **while** $j \leq \text{Latitud}(c)$ **do** $\triangleright \mathcal{O}(\text{Latitud}(c))$ 13: $\text{Agregar}(m.\text{columnas}[\text{Longitud}(c)], j, < \text{FALSE}, i\text{Vacio}())$ \triangleright se agrega en la posicion j-esima del vectoruna tupla $\langle \text{bool}, \text{arreglo vacio} \rangle \mathcal{O}(1)$ 14: $j \leftarrow j + 1$ $\triangleright \mathcal{O}(\text{Latitud}(c))$ 15: **end while**16: **end if**17: $\text{Agregar}(m.\text{columnas}[\text{Longitud}(c)], \text{Latitud}(c), < \text{True}, i\text{Vacio}() >)$ $\triangleright \Theta(1)$ 18: $it \leftarrow \text{coordendas}(m)$ $\triangleright \mathcal{O}(1)$ 19: **while** haySiguiente(it) **do**

20:

21: **if** HayCamino($c, \text{Siguiente}(it), m$) $\wedge (c \neq \text{Siguiente}(it)) \wedge \text{distanciaEuclideana}(c, \text{Siguiente}(it)) < 100$ **then** \triangleright hay n llamados a la funcion hay camino $\mathcal{O}(n)$ donde n es la cantidad de coordenadas del mapa $\mathcal{O}(n^2)$ 22: $\text{Agregar}(m.\text{columnas}[\text{longitud}(\text{Siguiente}(it))][\text{Latitud}(\text{Siguiente}(it))].\text{adyacentes}, c)$ $\triangleright \mathcal{O}(1)$ 23: $\text{Agregar}(m.\text{columnas}[\text{Longitud}(c)][\text{Latitud}(c)].\text{adyacentes}, \text{Siguiente}(it))$ $\triangleright \mathcal{O}(1)$ 24: **end if**25: $\text{Avanzar}(it)$ $\triangleright \mathcal{O}(1)$ 26: **end while**

Complejidad: La funcion realiza el el peor caso $2 * \text{Latitud}(c) + 2 * \text{Longitud}(c)$ inserciones a un vector pues puede suceder que tanto la latitud como la longitud sean numero siguiente a alguna potencia de 2 con lo cual debemos duplicar el vector para insetarlos e inicializar correctamente todas las otras posiciones "sobrantes". Se realizan ademas n llamadas a la funcion hayCamino , cuya complejidad es tambien n, donde n es $\#(m.\text{coordendas})$. Tenemos entonces que la complejidad de nuestro algoritmo es $\mathcal{O}(\text{Latitud}(c) + \text{Longitud}(c) + n^2)$.

iHayCamino(in c : coord, in c' : coord, in m : map) $\rightarrow res$:bool

- 1: **if** $distEuclidea(c, c') < 100$ **then**
- 2: $res \leftarrow pertenece(m.columnas[Longitud(c)][Latitud(c)].adyacentes, c')$ \triangleright recorro un conjunto acotado $\mathcal{O}(1)$
- 3: **else** $res \leftarrow iHayCamino2(c, c', m)$ $\triangleright \mathcal{O}(\#(m.coordenadas)^2)$
- 4: **end if**

Complejidad: En el caso en el que la distancia euclidea entre las dos coordenadas pasadas por parametro es menor a 100 solo es necesario buscar en el conjunto de las coordenadas adyacentes de una de ellas que contiene a lo sumo 81 coordenadas y tenemos una complejidad de $\mathcal{O}(1)$. Sin embargo, para cualquier otro par de coordenadas cuya distancia euclidea sea mayor a 100 la funcion debe llamar a la funcion auxiliar `hayCamino2` cuya complejidad es $\mathcal{O}(4^n n^2)$ donde n es $\#(m.coordenadas)$. Por lo que la complejidad de la funcion resulta ser esta ultima.

iHayCamino2(in c : coord, in c' : coord, in m : map) $\rightarrow res$:bool

- 1: $coords \leftarrow iVacio()$
- 2: $it \leftarrow coordenadas(m)$
- 3: **while** `haySiguiente(it)` **do**
- 4: $iAgregarRapido(coords, siguiente(it))$
- 5: $Avanzar(it)$
- 6: **end while**
- 7: $res \leftarrow iExisteCamino(c, c', Eliminar(coords, c), m)$ $\triangleright \mathcal{O}(4^{\#Eliminar(coords, c)} (\#Eliminar(coords, c))^2)$

Complejidad: $\mathcal{O}(4^n n^2)$ donde n es $\#(m.coordenadas)$

iExisteCamino(in c : coord, in c' : coord, in cs : conj(coord), in m : map) $\rightarrow res$:bool

- 1:
- 2: **if** $c = c'$ **then**
- 3: $res \leftarrow true$
- 4: **else**
- 5: **if** `Vacio?(cs)` **then**
- 6: $res \leftarrow false$
- 7: **else** $res \leftarrow iExisteCaminoPorArriba(c, c', cs, m) \text{ or } iExisteCaminoPorAbajo(c, c', cs, m) \text{ or } iExisteCaminoPorDerecha(c, c', cs, m) \text{ or } iExisteCaminoPorIzquierda(c, c', cs, m)$
- 8: **end if**
- 9: **end if**

Complejidad: Si realizamos el arbol de ejecuciones de la funcion tendremos que en cada ejecucion se llama a la recursion 4 veces, donde los hijos reciben un conjunto que contiene un elemento menos que el padre. En cada nodo se llama a la funcion `pertenece` del conjunto. Luego tenemos que la ecuacion de recurrencia es la siguiente

$$\mathcal{T}(n) = \begin{cases} \mathcal{O}(1) & \text{si } n = 1 \\ 4\mathcal{T}(n-1) + \mathcal{O}(n) & \text{sin } >1 \end{cases}$$

Luego obtenemos que

$$\mathcal{T}(n) = \sum_{i=1}^{n-1} (4^i + \mathcal{O}(n)) \leq 4^n \mathcal{O}(n^2) = \mathcal{O}(4^n n^2) \text{ donde } n \text{ es } \#cs$$

iExisteCaminoPorArriba(in c : coord, in c' : coord, in cs : conj(coord), in m : map) $\rightarrow res$:bool

```

1:
2: if  $iPertenece(cs, iCoordenadaArriba(c))$  then
3:    $res \leftarrow iExisteCamino(iCoordenadaArriba(c), c', Remover(cs, iCoordenadaArriba(c)), m)$ 
4: elseres  $\leftarrow false$ 
5: end if

```

Complejidad: $\mathcal{O}(4^n n^2)$ donde n es #cs

iExisteCaminoPorAbajo(in c : coord, in c' : coord, in cs : conj(coord), in m : map) $\rightarrow res$:bool

```

1:
2: if  $latitud(c) > 0$  then
3:
4:   if  $iPertenece(cs, iCoordenadaAbajo(c))$  then
5:      $res \leftarrow iExisteCamino(iCoordenadaAbajo(c), c', Remover(cs, iCoordenadaAbajo(c)), m)$ 
6:   elseres  $\leftarrow false$ 
7:   end if
8: elseres  $\leftarrow false$ 
9: end if

```

Complejidad: $\mathcal{O}(4^n n^2)$ donde n es #cs

iExisteCaminoPorDerecha(in c : coord, in c' : coord, in cs : conj(coord), in m : map) $\rightarrow res$:bool

```

1:
2: if  $iPertenece(cs, iCoordenadaALaDerecha(c))$  then
3:    $res \leftarrow iExisteCamino(iCoordenadaALaDerecha(c), c', Remover(cs, iCoordenadaALaDerecha(c)), m)$ 
4: elseres  $\leftarrow false$ 
5: end if

```

Complejidad: $\mathcal{O}(4^n n^2)$ donde n es #cs

iExisteCaminoPorIzquierda(in c : coord, in c' : coord, in cs : conj(coord), in m : map) $\rightarrow res$:bool

```

1:
2: if  $longitud(c) > 0$  then
3:
4:   if  $iPertenece(cs, iCoordenadaALaIzquierda(c))$  then
5:      $res \leftarrow iExisteCamino(iCoordenadaALaIzquierda(c), c', Remover(cs, iCoordenadaALaIzquierda(c)), m)$ 
6:   elseres  $\leftarrow false$ 
7:   end if
8: elseres  $\leftarrow false$ 
9: end if

```

Complejidad: $\mathcal{O}(4^n n^2)$ donde n es #cs

3. Renombres de TADs

TAD **JUGADORHEAP** es $\text{TUPLA}(\text{NAT}, \text{NAT})$

4. Módulo Heap Modificable

Este módulo implementa una cola de prioridad. El TAD Cola de prioridad es paramétrico, es posible utilizarlo con cualquier tipo α que tenga una relación de orden total estricto. Esta implementación se limita al tipo jugadorHeap, que es tupla $\langle \text{Nat}, \text{Nat} \rangle$.

Interfaz

se explica con: COLA DE PRIORIDAD, JUGADORHEAP.

géneros: heapMod.

Operaciones básicas de Heap Modificable

VACÍA?(in $c : \text{heapMod}$) $\rightarrow res : \text{bool}$

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

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

Complejidad: $\Theta(1)$

Descripción: Devuelve true si la cola de prioridad está vacía.

PRÓXIMO(in $c : \text{heapMod}$) $\rightarrow res : \text{JugadorHeap}$

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

Post $\equiv \{\text{alias}(res, \text{próximo}())\}$

Complejidad: $\Theta(1)$

Descripción: Retorna por referencia el próximo valor en la cola.

Aliasing: La referencia es constante, el elemento a no puede modificarse

DESNCOLAR(in/out $c : \text{heapMod}$)

Pre $\equiv \{\neg \text{vacía?}(c) \wedge c =_{\text{obs}} c_0\}$

Post $\equiv \{c =_{\text{obs}} \text{desencolar}(c_0)\}$

Complejidad: $\Theta(\log(\# \text{heapSecu}(c)))$

Descripción: Desencola el próximo elemento.

VACÍA() $\rightarrow res : \text{heapMod}$

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

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

Complejidad: $\Theta(1)$

Descripción: Retorna por referencia una cola vacía.

ENCOLAR(in $a : \text{JugadorHeap}$, in/out $c : \text{heapMod}$) $\rightarrow res : \text{itHeapMod}$

Pre $\equiv \{c =_{\text{obs}} c_0\}$

Post $\equiv \{c =_{\text{obs}} \text{encolar}(c_0) \wedge res =_{\text{obs}} \text{AgregarComoSiguierte}(\text{CrearIt}(c), a)\}$

Complejidad: $\Theta(\log(\# \text{heapSecu}(c)))$

Descripción: Encola el elemento JugadorHeap. Devuelve un iterador de forma tal que al pedir SIGUIENTE se obtenga el elemento agregado.

ESMAYOR?(in $a : \text{JugadorHeap}$, in $b : \text{JugadorHeap}$) $\rightarrow res : \text{bool}$

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

Post $\equiv \{res =_{\text{obs}} \text{esMayor?}(a, b)\}$

Complejidad: $\Theta(1)$

Descripción: Devuelve true si a tiene mayor prioridad que b .

Operaciones del iterador

El iterador que presentamos permite modificar la cola de prioridad, pudiendo eliminar elementos que se encuentren en cualquier posición de la cola. No se incluyeron las funciones para avanzar y retroceder en la cola ya que no son

necesarias para el proposito de esta cola de prioridad.

CREARIT(in c : heapMod) $\rightarrow res$: itHeapMod

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} crearItBi(<>, c) \wedge alias(SecuSuby(it) = c)\}$

Complejidad: $\Theta(1)$

Descripción: crea un iterador bidireccional de la cola de prioridad, de forma tal que al pedir SIGUIENTE se obtenga el primer elemento de c .

Aliasing: el iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE.

HAYSIGUIENTE(in it : itHeapMod) $\rightarrow res$: bool

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} haySiguiente?(it)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

SIGUIENTE(in it : itHeapMod) $\rightarrow res$: JugadorHeap

Pre $\equiv \{HaySiguiente?(it)\}$

Post $\equiv \{alias(res =_{obs} Siguiente(it))\}$

Complejidad: $\Theta(1)$

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

Aliasing: res es modificable si y sólo si it es modificable.

ELIMINARSIGUIENTE(in/out it : itHeapMod)

Pre $\equiv \{it = it_0 \wedge HaySiguiente?(it)\}$

Post $\equiv \{it =_{obs} EliminarSiguiente(it_0)\}$

Complejidad: $\Theta(\log(\#heapSecu(c)))$

Descripción: elimina de la lista iterada el valor que se encuentra en la posición siguiente del iterador.

AGREGARCOMOSIGUIENTE(in/out it : itHeapMod, in a : JugadorHeap)

Pre $\equiv \{it = it_0\}$

Post $\equiv \{it =_{obs} AgregarComoSiguiente(it_0, a)\}$

Complejidad: $\Theta(\log(\#heapSecu(c)))$

Descripción: agrega el elemento a a la cola de prioridad, dejando al iterador posicionado de forma tal que al llamar a SIGUIENTE se obtenga a .

Aliasing: el elemento a se agrega por copia.

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Cola de Prioridad extendida

extiende COLA DE PRIORIDAD

otras operaciones

$esMayor?$: jugadorHeap \times jugadorHeap \rightarrow bool

otras operaciones (no exportadas)

$heapSecu$: heapMod $\rightarrow secu(jugadorHeap)$

axiomas

$esMayor?(a, b) \equiv \text{if } \Pi_1(a) = \Pi_1(b) \text{ then } \Pi_2(a) < \Pi_2(b) \text{ else } \Pi_1(a) < \Pi_1(b) \text{ fi}$

$heapSecu(c) \equiv \text{if } vacía?(c) \text{ then } <> \text{ else } próximo(c) \bullet heapSecu(desencolar(c)) \text{ fi}$

Fin TAD

Representación

Representación de la Cola de Prioridad

En este módulo vamos a utilizar un árbol binario para representar la cola, donde cada nodo contiene un elemento. La prioridad de los hijos es siempre menor a la del padre, por lo tanto la raíz del árbol tiene el elemento de mayor prioridad. La altura del árbol es igual al logaritmo de la cantidad de elementos ($\log(\#nodos)$). Cada nivel del árbol se

va llenando de izquierda a derecha. Cada nodo guarda la longitud de la rama más corta y la longitud de la rama más larga, lo cual se utiliza para poder buscar el último nodo(es decir, el nodo del último nivel que está más a la derecha) o donde ubicar un nuevo nodo que se agrega a la cola. Debido a como se van colocando los nodos, la rama derecha nunca puede ser más larga que la izquierda, por lo tanto la rama más larga del hijo izquierdo de un nodo va a ser igual o mayor en una unidad a la rama más larga del hijo derecho, mientras que lo contrario ocurre con la longitud de la rama más corta. Cada nodo, además de tener un puntero a cada hijo, también guarda un puntero al padre para poder acceder rápidamente.

heapMod se representa con heapmod

donde heapmod es $\text{tupla}(\text{tope: puntero(nodo)})$

donde nodo es $\text{tupla}(\text{elemento: jugadorHeap, ramaMasCorta: nat, ramaMasLarga: nat, hijoIzq: puntero(nodo), hijoDer: puntero(nodo), padre: puntero(nodo)})$

donde jugadorHeap es $\text{tupla}(\text{cantPokes: nat, id: nat})$

$\text{Rep} : \text{heapmod} \rightarrow \text{bool}$

$\text{Rep}(c) \equiv \text{true} \iff 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6 \wedge 7 \wedge 8 \wedge 9 \wedge 10 \wedge 11$

Invariante de representación:

- 1) *tope* es un puntero al nodo raíz del árbol.
- 2) Todos los nodos tienen un puntero al padre. En el caso de la raíz, apunta a *NULL*.
- 3) Todos los nodos tienen un elemento.
- 4) El elemento es una tupla $\langle \text{Nat}, \text{Nat} \rangle$
- 5) Los elementos de los hijos tienen menor prioridad que el del padre.
- 6) La altura del árbol es igual al logaritmo de la cantidad de nodos ($\log(\#\text{nodos})$).
- 7) Cada nivel del árbol se va llenando de izquierda a derecha, por lo tanto todas las hojas están en los últimos dos niveles del árbol y la diferencia de altura entre los dos subárboles de un nodo puede ser 0 o 1.
- 8) Cada nodo guarda la longitud de la rama más corta y la más larga.
- 9) La longitud de la rama más corta y la longitud de la rama más larga de una hoja es 0.
- 10) La longitud de la rama más corta de un nodo debe ser uno más que la longitud de la rama más corta del subárbol derecho.
- 11) La longitud de la rama más larga de un nodo debe ser uno más que la longitud de la rama más larga del subárbol izquierdo.

Representación del iterador

El iterador está formado simplemente por un puntero al nodo siguiente y un puntero a la cola, para poder modificarla. El puntero siguiente apunta a *NULL* en el caso en que la cola este vacía.

itHeapMod se representa con itHeapmod

donde itHeapmod es $\text{tupla}(\text{siguiente: puntero(heapmod), heap: puntero(heapmod)})$

$\text{Rep} : \text{itHeapmod} \rightarrow \text{bool}$

$\text{Rep}(it) \equiv \text{true} \iff 1 \wedge 2$

Invariante de representación:

- 1) *siguiente* apunta a *NULL* si i solo si el *heap* apunta a una cola vacía.
- 2) Si la cola no está vacía, *siguiente* apunta a un nodo que pertenece a la cola apuntada por *heap*.

Algoritmos

Algoritmos del módulo

iVacía?(in *c*: heapmod) \rightarrow *res*: bool

1: *res* \leftarrow *c.tope* = *NULL*

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iPróximo(in c : heapmod) $\rightarrow res$: jugadorHeap
1: $res \leftarrow *(c.tope).elemento$ $\triangleright \Theta(1)$ Complejidad: $\Theta(1)$

iDesencolar(in/out c : heapmod)
1: **if** $*(c.tope).hijoIzq = NULL \wedge *(c.tope).hijoDer = NULL$ **then** $\triangleright \Theta(1)$ 2: $c.tope \leftarrow NULL$ \triangleright Si hay un solo elemento, simplemente se elimina // $\Theta(1)$ 3: **else**4: $puntero(nodo) \ ultimoNodo \leftarrow iUltimoNodo(c)$ \triangleright Se obtiene un puntero al último nodo // $\Theta(\log(\#nodos(c)))$ 5: $puntero(nodo) \ padreUlt \leftarrow (*ultimoNodo).padre$ \triangleright Se crea un puntero al padre del último nodo // $\Theta(1)$ 6: **if** $(*padreUlt).hijoDer = ultimoNodo$ **then** \triangleright Vemos si el último nodo es el derecho del padre // $\Theta(1)$ 7: $(*padreUlt).hijoDer \leftarrow NULL$ \triangleright Eliminamos la conexión al nodo // $\Theta(1)$ 8: **else**9: $(*padreUlt).hijoIzq \leftarrow NULL$ \triangleright Eliminamos la conexión al nodo // $\Theta(1)$ 10: **end if**11: $iCorregirProfundidad(padreUlt, c)$ \triangleright Corregimos la profundidad // $\Theta(\log(\#nodos(c)))$ 12: $(*ultimoNodo).padre \leftarrow NULL$ $\triangleright \Theta(1)$ 13: $(*ultimoNodo).hijoIzq \leftarrow (*c.tope).hijoIzq$ $\triangleright \Theta(1)$ 14: $(*ultimoNodo).hijoDer \leftarrow (*c.tope).hijoDer$ $\triangleright \Theta(1)$ 15: $(*ultimoNodo).ramaMasCorta \leftarrow (*c.tope).ramaMasCorta$ $\triangleright \Theta(1)$ 16: $(*ultimoNodo).ramaMasLarga \leftarrow (*c.tope).ramaMasLarga$ $\triangleright \Theta(1)$ 17: $c.tope \leftarrow ultimoNodo$ $\triangleright \Theta(1)$ 18: $iSiftDown(c.tope, c)$ \triangleright Se repara el heap // $\Theta(\log(\#nodos(c)))$ 19: **end if**Complejidad: $\Theta(\log(\#nodos(c)))$ Justificación: El algoritmo llama a tres funciones con costo $\Theta(\log(\#nodos(c)))$ y el resto de las operaciones tienen costo $\Theta(1)$

iVacía() $\rightarrow res$: heapmod
1: $res \leftarrow \langle NULL \rangle$ $\triangleright \Theta(1)$ Complejidad: $\Theta(1)$

iEncolar(in a : jugadorHeap, in/out c : heapmod) $\rightarrow res$: itHeapmod

```

1: if  $c.tope = NULL$  then ▷ La cola está vacía //  $\Theta(1)$ 
2:    $c.tope \leftarrow \langle a, 0, 0, NULL, NULL, NULL \rangle$  ▷ Se coloca el primer elemento //  $\Theta(1)$ 
3: else
4:    $puntero(nodo) \text{ futuroPadre} \leftarrow iFuturoPadre(c)$  ▷ Puntero al futuro padre del nodo de  $a$  //  $\Theta(\log(\#nodos(c)))$ 
5:   if  $(*futuroPadre).hijoIzq = NULL$  then ▷ Vemos si tiene hijo izquierdo //  $\Theta(1)$ 
6:      $(*futuroPadre).hijoIzq \leftarrow \langle a, 0, 0, NULL, NULL, futuroPadre \rangle$  ▷ Creamos el nuevo nodo //  $\Theta(1)$ 
7:      $futuroPadre \leftarrow (*futuroPadre).hijoIzq$  ▷ Nos posicionamos en el nuevo elemento //  $\Theta(1)$ 
8:   else
9:      $(*futuroPadre).hijoDer \leftarrow \langle a, 0, 0, NULL, NULL, futuroPadre \rangle$  ▷ Creamos el nuevo nodo //  $\Theta(1)$ 
10:     $futuroPadre \leftarrow (*futuroPadre).hijoDer$  ▷ Nos posicionamos en el nuevo elemento //  $\Theta(1)$ 
11:   end if
12:    $iCorregirProfundidad((*futuroPadre).padre, c)$  ▷ Corregimos la profundidad //  $\Theta(\log(\#nodos(c)))$ 
13:    $iSiftUp(futuroPadre, c)$  ▷ Se repara el heap //  $\Theta(\log(\#nodos(c)))$ 
14: end if
15:  $res \leftarrow \langle futuroPadre, c \rangle$  ▷ Se devuelve el iterador //  $\Theta(1)$ 

```

Complejidad: $\Theta(\log(\#nodos(c)))$ Justificación: El algoritmo llama a tres funciones con costo $\Theta(\log(\#nodos(c)))$ y el resto de las operaciones tienen costo $\Theta(1)$

iUltimoNodo(in/out c : heapmod) $\rightarrow res$: puntero(nodo)

```

1:  $puntero(nodo) \text{ ultimoNodo} \leftarrow c.tope$  ▷ Se crea un puntero para buscar el último nodo //  $\Theta(1)$ 
2: while  $\neg((*ultimoNodo).hijoIzq = NULL \wedge (*ultimoNodo).hijoDer = NULL)$  do ▷ //  $\Theta(1)$ 
3:   if  $(*(*ultimoNodo).hijoIzq).ramaMasLarga = *(*ultimoNodo).hijoDer).ramaMasLarga$  then ▷ //  $\Theta(1)$ 
4:      $ultimoNodo \leftarrow (*ultimoNodo).hijoDer$  ▷ Si la profundidad es la misma, va a la derecha //  $\Theta(1)$ 
5:   else
6:      $ultimoNodo \leftarrow (*ultimoNodo).hijoIzq$  ▷ Si no, va a la izquierda //  $\Theta(1)$ 
7:   end if
8: end while
9:  $res \leftarrow ultimoNodo$  ▷ //  $\Theta(1)$ 

```

Complejidad: $\Theta(\log(\#nodos(c)))$ Justificación: El algoritmo cuenta con un ciclo que se repetirá $\log(\#nodos(c))$ (recorre el árbol de arriba a abajo), y el resto de las operaciones tienen costo $\Theta(1)$

iFuturoPadre(in/out c : heapmod) $\rightarrow res$: puntero(nodo)

```

1:  $puntero(nodo) \text{ ultimoNodo} \leftarrow c.tope$  ▷ Se crea un puntero para buscar la última posición //  $\Theta(1)$ 
2: while  $(*ultimoNodo).hijoIzq \neq NULL \wedge (*ultimoNodo).hijoDer \neq NULL$  do ▷ //  $\Theta(1)$ 
3:   if  $(*(*ultimoNodo).hijoIzq).ramaMasCorta = *(*ultimoNodo).hijoDer).ramaMasCorta$  then ▷ //  $\Theta(1)$ 
4:      $ultimoNodo \leftarrow (*ultimoNodo).hijoIzq$  ▷ Si la profundidad es la misma, va a la izquierda //  $\Theta(1)$ 
5:   else
6:      $ultimoNodo \leftarrow (*ultimoNodo).hijoDer$  ▷ Si no, va a la derecha //  $\Theta(1)$ 
7:   end if
8: end while
9:  $res \leftarrow ultimoNodo$  ▷ //  $\Theta(1)$ 

```

Complejidad: $\Theta(\log(\#nodos(c)))$ Justificación: El algoritmo cuenta con un ciclo que se repetirá $\log(\#nodos(c))$ (recorre el árbol de arriba a abajo), y el resto de las operaciones tienen costo $\Theta(1)$

iCorregirProfundidad(in p : puntero(nodo), in/out c : heapmod)

```

1: if (*p).hijoIzq = NULL ∧ (*p).hijoDer = NULL then           ▷ Vemos si no tiene hijos //  $\Theta(1)$ 
2:   (*p).ramaMasCorta ← 0                                       ▷ //  $\Theta(1)$ 
3:   (*p).ramaMasLarga ← 0                                       ▷ //  $\Theta(1)$ 
4: else
5:   if (*p).hijoIzq ≠ NULL ∧ (*p).hijoDer ≠ NULL then         ▷ Vemos si tiene dos hijos //  $\Theta(1)$ 
6:     (*p).ramaMasCorta ← 1                                       ▷ //  $\Theta(1)$ 
7:     (*p).ramaMasLarga ← 1                                       ▷ //  $\Theta(1)$ 
8:   else
9:     (*p).ramaMasCorta ← 0                                       ▷ Tiene un solo hijo //  $\Theta(1)$ 
10:    (*p).ramaMasLarga ← 1                                       ▷ //  $\Theta(1)$ 
11:   end if
12: end if
13:  $p \leftarrow (*p).padre$                                        ▷ Subo un nivel //  $\Theta(1)$ 
14: while (*p).padre ≠ NULL do                                     ▷ Recorre hasta el tope del arbol //  $\Theta(1)$ 
15:   (*p).ramaMasCorta ← ((*p).hijoDer).ramaMasCorta + 1         ▷ //  $\Theta(1)$ 
16:   (*p).ramaMasLarga ← ((*p).hijoIzq).ramaMasLarga + 1         ▷ //  $\Theta(1)$ 
17: end while

```

Complejidad: $\Theta(\log(\#nodos(c)))$ Justificación: El algoritmo cuenta con un ciclo que se repetirá $\log(\#nodos(c))$ (recorre el árbol de abajo hacia arriba), y el resto de las operaciones tienen costo $\Theta(1)$

iSiftDown(in/out c : heapmod, in p : puntero(nodo))

```

1: puntero(nodo) swap ← p                                       ▷ Creamos una variable para intercambio  $\Theta(1)$ 
2: if iEsMayor?((*p).hijoIzq).elemento, (*p).elemento) then     ▷ Vemos si el hijo izquierdo es mayor  $\Theta(1)$ 
3:   swap ← (*p).hijoIzq                                         ▷  $\Theta(1)$ 
4: end if
5: if iEsMayor?((*p).hijoDer).elemento, (swap).elemento) then   ▷ Vemos si el hijo derecho es el mayor  $\Theta(1)$ 
6:   swap ← (*p).hijoDer                                         ▷  $\Theta(1)$ 
7: end if
8: if  $p \neq swap$  then                                           ▷ Vemos si es necesario reacomodar los nodos  $\Theta(1)$ 
9:   iIntercambio( $p$ , swap)                                       ▷ Intercambiamos el padre con el hijo  $\Theta(1)$ 
10:  iSiftDown( $c$ , swap)                                          ▷ Llamamos a la funcion recursivamente  $\Theta(1)$ 
11: end if

```

Complejidad: $\Theta(\log(\#nodos(c)))$ Justificación: Todas las operaciones del algoritmo tienen costo $\Theta(1)$, por lo tanto una llamada a la función tiene costo $\Theta(1)$. El algoritmo recorre el árbol c recursivamente. En cada nueva llamada a función baja un nivel en el árbol, por lo tanto pueden haber como máximo $\log(\#nodos(c))$ llamadas a función, y cada una de las llamadas tiene costo $\Theta(1)$.

iSiftUp(in/out c : heapmod, in p : puntero(nodo))

```

1: puntero(nodo) swap  $\leftarrow p$                                 ▷ Creamos una variable para intercambio  $\Theta(1)$ 
2: if iEsMayor?(( $*p$ ).elemento, ( $*p$ ).padre).elemento then      ▷ Vemos si es mayor al padre  $\Theta(1)$ 
3:   swap  $\leftarrow (*p).padre$                                     ▷  $\Theta(1)$ 
4: end if
5: if  $p \neq swap$  then                                            ▷ Vemos si es necesario reacomodar los nodos  $\Theta(1)$ 
6:   iIntercambio(swap,  $p$ )                                       ▷ Intercambiamos el padre con el hijo  $\Theta(1)$ 
7:   iSiftUp( $c$ , swap)                                             ▷ Llamamos a la funcion recursivamente  $\Theta(1)$ 
8: end if

```

Complejidad: $\Theta(\log(\#nodos(c)))$

Justificación: Todas las operaciones del algoritmo tienen costo $\Theta(1)$, por lo tanto una llamada a la función tiene costo $\Theta(1)$. El algoritmo recorre el árbol c recursivamente. En cada nueva llamada a función sube un nivel en el árbol, por lo tanto pueden haber como máximo $\log(\#nodos(c))$ llamadas a función, y cada una de las llamadas tiene costo $\Theta(1)$.

iEsMayor?(in a : jugadorHeap, in b : jugadorHeap) $\rightarrow res$: bool

```

1: esMayor  $\leftarrow false$                                        ▷ //  $\Theta(1)$ 
2: if a.cantPokes = b.cantPokes                                  ▷ //  $\Theta(1)$ 
3:   esMayor  $\leftarrow a.id < b.id$  then                             ▷ //  $\Theta(1)$ 
4: else
5:   esMayor  $\leftarrow a.cantPokes < b.cantPokes$                   ▷ //  $\Theta(1)$ 
6: end if
7: res  $\leftarrow esMayor$                                            ▷ //  $\Theta(1)$ 

```

Complejidad: $\Theta(1)$

Justificación: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

iIntercambio(in/out padre: puntero(nodo), in/out hijo: puntero(nodo))

```

1: rmc  $\leftarrow (*hijo).ramaMasCorta$                              ▷ Intercambiamos las características  $\Theta(1)$ 
2: rml  $\leftarrow (*hijo).ramaMasLarga$                                ▷  $\Theta(1)$ 
3: ( $*hijo$ ).padre  $\leftarrow (*padre).padre$                            ▷  $\Theta(1)$ 
4: ( $*hijo$ ).ramaMasCorta  $\leftarrow (*padre).ramaMasCorta$              ▷  $\Theta(1)$ 
5: ( $*hijo$ ).ramaMasLarga  $\leftarrow (*padre).ramaMasLarga$              ▷  $\Theta(1)$ 
6: ( $*padre$ ).padre  $\leftarrow hijo$                                     ▷  $\Theta(1)$ 
7: ( $*padre$ ).ramaMasCorta  $\leftarrow rmc$                                ▷  $\Theta(1)$ 
8: ( $*padre$ ).ramaMasLarga  $\leftarrow rml$                                ▷  $\Theta(1)$ 
9: if ( $*padre$ ).hijoIzq = hijo then                                ▷ Vemos si era el hijo izquierdo o el derecho  $\Theta(1)$ 
10:  puntero(nodo) hDer  $\leftarrow (*padre).hijoDer$                  ▷  $\Theta(1)$ 
11:  ( $*padre$ ).hijoIzq  $\leftarrow (*hijo).hijoIzq$                    ▷  $\Theta(1)$ 
12:  ( $*padre$ ).hijoDer  $\leftarrow (*hijo).hijoDer$                      ▷  $\Theta(1)$ 
13:  ( $*hijo$ ).hijoIzq  $\leftarrow padre$                                 ▷  $\Theta(1)$ 
14:  ( $*hijo$ ).hijoDer  $\leftarrow hDer$                                  ▷  $\Theta(1)$ 
15: else
16:  puntero(nodo) hIzq  $\leftarrow (*padre).hijoIzq$                  ▷  $\Theta(1)$ 
17:  ( $*padre$ ).hijoIzq  $\leftarrow (*hijo).hijoIzq$                    ▷  $\Theta(1)$ 
18:  ( $*padre$ ).hijoDer  $\leftarrow (*hijo).hijoDer$                      ▷  $\Theta(1)$ 
19:  ( $*hijo$ ).hijoDer  $\leftarrow padre$                                 ▷  $\Theta(1)$ 
20:  ( $*hijo$ ).hijoIzq  $\leftarrow hIzq$                                  ▷  $\Theta(1)$ 
21: end if

```

Complejidad: $\Theta(1)$

Justificación: Todas las operaciones del algoritmo tienen costo $\Theta(1)$, por lo tanto una llamada a la función tiene costo $\Theta(1)$.

Algoritmos del iterador

iCrearIt(in c : heapmod) $\rightarrow res$: itHeapmod

1: $res \leftarrow \&(c.tope)$

▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iHaySiguiente(in it : itHeapmod) $\rightarrow res$: bool

1: $res \leftarrow c.tope \neq NULL$

▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iSiguiente(in it : itHeapmod) $\rightarrow res$: jugadorHeap

1: $res \leftarrow (*it.siguiente).elemento$

▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iEliminarSiguiente(in it : itHeapmod) $\rightarrow res$: jugadorHeap

```

1: puntero(nodo) ultimoNodo  $\leftarrow (*it.heap).tope$            ▷ Se crea un puntero para buscar el último nodo //  $\Theta(1)$ 
2: if (*ultimoNodo).hijoIzq = NULL  $\wedge$  (*ultimoNodo).hijoDer = NULL then           ▷  $\Theta(1)$ 
3:   (*it.heap).tope  $\leftarrow NULL$            ▷ Si hay un solo elemento, simplemente se elimina //  $\Theta(1)$ 
4: else
5:   ultimoNodo  $\leftarrow iUltimoNodo(c)$            ▷ Se obtiene un puntero al último nodo //  $\Theta(\log(\#nodos(c)))$ 
6:   puntero(nodo) padreUlt  $\leftarrow (*ultimoNodo).padre$            ▷ Se crea un puntero al padre del último nodo //  $\Theta(1)$ 
7:   if (*padreUlt).hijoDer = ultimoNodo then           ▷ Vemos si el último nodo es el derecho del padre //  $\Theta(1)$ 
8:     (*padreUlt).hijoDer  $\leftarrow NULL$            ▷ Eliminamos la conexión al nodo //  $\Theta(1)$ 
9:   else
10:    (*padreUlt).hijoIzq  $\leftarrow NULL$            ▷ Eliminamos la conexión al nodo //  $\Theta(1)$ 
11:   end if
12:   iCorregirProfundidad(padreUlt, c)           ▷ Corregimos la profundidad //  $\Theta(\log(\#nodos(c)))$ 
13:   (*ultimoNodo).padre  $\leftarrow (*it.siguiente).padre$            ▷ Colocamos el último en lugar del nodo a eliminar //  $\Theta(1)$ 
14:   (*ultimoNodo).hijoIzq  $\leftarrow (*it.siguiente).hijoIzq$            ▷  $\Theta(1)$ 
15:   (*ultimoNodo).hijoDer  $\leftarrow (*it.siguiente).hijoDer$            ▷  $\Theta(1)$ 
16:   (*ultimoNodo).ramaMasCorta  $\leftarrow (*it.siguiente).ramaMasCorta$            ▷  $\Theta(1)$ 
17:   (*ultimoNodo).ramaMasLarga  $\leftarrow (*it.siguiente).ramaMasLarga$            ▷  $\Theta(1)$ 
18:   if ((*it.siguiente).padre).hijoIzq = *it.siguiente then           ▷  $\Theta(1)$ 
19:     ((*it.siguiente).padre).hijoIzq  $\leftarrow ultimoNodo$            ▷  $\Theta(1)$ 
20:   else
21:     ((*it.siguiente).padre).hijoDer  $\leftarrow ultimoNodo$            ▷  $\Theta(1)$ 
22:   end if
23:   iSiftDown(ultimoNodo, c)           ▷ Se repara el heap //  $\Theta(\log(\#nodos(c)))$ 
24:   iSiftUp(ultimoNodo, c)           ▷ Se repara el heap //  $\Theta(\log(\#nodos(c)))$ 
25: end if

```

Complejidad: $\Theta(\log(\#nodos(c)))$

Justificación: El algoritmo llama a cuatro funciones con costo $\Theta(\log(\#nodos(c)))$ y el resto de las operaciones tienen costo $\Theta(1)$

AgregarComoSiguiente(in it : itHeapmod, in a : jugadorHeap)

1: $it \leftarrow iEncolar(*it.heap, a)$ ▷ Le pasamos el elemento a al heap al que pertenece el iterador //
 $\Theta(\log(\#nodos(it.heap)))$

Complejidad: $\Theta(\log(\#nodos(it.heap)))$

Justificación: El algoritmo simplemente llama $iEncolar$.

5. Módulo Diccionario en Cadena(*string*, σ)

El módulo Diccionario en Cadena provee un diccionario básico en el que se puede definir, borrar, y testear si una clave está definida en tiempo lineal. Cuando ya se sabe que la clave a definir no está definida en el diccionario, la definición se puede hacer en tiempo $O(|K|)$, donde $K \in \text{string}$ es la clave mas larga.

Para describir la complejidad de las operaciones, vamos a llamar $\text{copy}(k)$ al costo de copiar el elemento $k \in \text{string} \cup \sigma$ y $\text{equal}(k_1, k_2)$ al costo de evaluar si dos elementos $k_1, k_2 \in \text{string}$ son iguales (i.e., copy y equal son funciones de $\text{string} \cup \sigma$ y $\text{string} \times \text{string}$ en \mathbb{N} , respectivamente).¹

Interfaz

parámetros formales

géneros string, σ

función $\text{COPIAR}(\text{in } s : \sigma) \rightarrow \text{res} : \sigma$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} s\}$
Complejidad: $\Theta(\text{copy}(s))$
Descripción: función de copia de σ 's

se explica con: $\text{DICCIONARIO}(\text{string}, \sigma)$

géneros: $\text{dicc}(\text{string}, \sigma)$

Operaciones básicas de diccionario

$\text{VACÍO}() \rightarrow \text{res} : \text{dicc}(\text{string}, \sigma)$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{vacío}\}$
Complejidad: $\Theta(1)$
Descripción: genera un diccionario vacío.

$\text{DEFINIR}(\text{in/out } d : \text{dicc}(\text{string}, \sigma), \text{in } k : \text{string}, \text{in } s : \sigma)$
Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}} \text{definir}(d, k, s)\}$
Complejidad: $\mathcal{O}(\text{Longitud}(k))$
Descripción: define la clave k con el significado s en el diccionario.

$\text{DEFINIDO?}(\text{in } d : \text{dicc}(\text{string}, \sigma), \text{in } k : \text{string}) \rightarrow \text{res} : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{def?}(d, k)\}$
Complejidad: $\mathcal{O}(\text{Longitud}(k))$
Descripción: devuelve **true** si y sólo k está definido en el diccionario.

$\text{OBTENER}(\text{in } d : \text{dicc}(\text{string}, \sigma), \text{in } k : \text{string}) \rightarrow \text{res} : \sigma$
Pre $\equiv \{\text{def?}(d, k)\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{obtener}(d, k)\}$
Complejidad: $\mathcal{O}(\text{Longitud}(k))$
Descripción: devuelve el significado de la clave k en d .

$\text{BORRAR}(\text{in/out } d : \text{dicc}(\text{string}, \sigma), \text{in } k : \text{string})$
Pre $\equiv \{d = d_0 \wedge \text{def?}(d, k)\}$
Post $\equiv \{d =_{\text{obs}} \text{borrar}(d_0, k)\}$
Complejidad: $\mathcal{O}(\text{Longitud}(k))$
Descripción: elimina la clave k y su significado de d .

$\text{CLAVES}(\text{in } d : \text{dicc}(\text{string}, \sigma)) \rightarrow \text{res} : \text{Conj}(\text{string})$
Pre $\equiv \{\text{true}\}$

¹Nótese que este es un abuso de notación, ya que no estamos describiendo copy y equal en función del tamaño de k . A la hora de usarlo, habrá que realizar la traducción. De todas formas, notar que si hay que copiar strings, esto costará la longitud del string a copiar.

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

Complejidad: $\Theta(1)$

Descripción: devuelve las claves del diccionario.

COPIAR(in $d : \text{dicc}(string, \sigma) \rightarrow res : \text{dicc}(string, \sigma)$)

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

Post $\equiv \{res =_{\text{obs}} d\}$

Complejidad: $\mathcal{O}(\text{Longitud}(c) + \text{Cardinal}(\text{claves}))$

Descripción: genera una copia nueva del diccionario.

• = •(in $d_1 : \text{dicc}(string, \sigma)$, in $d_2 : \text{dicc}(string, \sigma) \rightarrow res : \text{bool}$)

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

Post $\equiv \{res =_{\text{obs}} c_1 = c_2\}$

Complejidad: $\mathcal{O}(\text{Longitud}(c) + \text{Cardinal}(\text{claves}))$

Descripción: compara d_1 y d_2 por igualdad, cuando σ posee operación de igualdad.

Requiere: **• = •**(in $s_1 : \sigma$, in $s_2 : \sigma \rightarrow res : \text{bool}$)

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

Post $\equiv \{res =_{\text{obs}} (s_1 = s_2)\}$

Complejidad: $\Theta(\text{equal}(s_1, s_2))$

Descripción: función de igualdad de σ 's

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Diccionario Extendido($string, \sigma$)

extiende CONJUNTO($string, \sigma$)

otras operaciones (no exportadas)

Definido : $string \times \text{dNodo} \rightarrow \text{bool}$

Obtener : $string \times \text{dNodo} \rightarrow \sigma$

$\{\text{Definido}(string, \text{dNodo})\}$

axiomas

Definido(c, d) \equiv if $\emptyset?(c) \wedge d.\text{Definicion} \neq \text{NULL}$ then true

else

if Longitud(c) $> 0 \wedge_L d.\text{Siguietes}[\text{int}(c[0])] \neq \text{NULL}$ then

Definido($\text{Fin}(c), d.\text{Siguietes}[\text{int}(c[0])]$)

else

false

fi

Obtener(c, d) \equiv if $\emptyset?(c)$ then $d.\text{Definicion}$ else Obtener($\text{fin}(c), d.\text{Siguietes}[\text{int}(c[0])]$) fi

Fin TAD

Representación

Representación del diccionario

Representamos al diccionario en cadena como una tupla de conjuntoLineal de string y puntero a nodo. Los nodos serán tuplas de σ , para guardar la definición, y vector de puntero a nodo. Como las claves son string, vamos a pedir que los vectores sean de 256 posiciones, ya que los strings están compuestos por chars, y tomamos la convención de que habrá 256 chars distintos para formar nuestros strings. Lo que le pediremos a cada instancia del diccionario en cadena, es que todo puntero en su vector de siguientes nunca apunte a algún nodo agregado anteriormente a la instancia, es decir: "que un nodo no apunte a nodos anteriores".

$\text{dicc}(string, \sigma)$ se representa con dicCadena

donde dicCadena es $\text{tupla}(\text{dNodo: puntero(Nodo)}, \text{claves: conjuntoLineal(string)})$

donde Nodo es $\text{tupla}(\text{definicion: } \sigma, \text{siguietes: vector(puntero(dNodo))})$

$\text{Rep} : \text{dNodo} \rightarrow \text{bool}$

$\text{Rep}(d) \equiv \text{true} \iff (1) \wedge_L (2) \wedge (3) \wedge_L (4)$

Donde 1,2,3 y 4 son:

- 1) Para todo nodo del diccionario en cadena, todo nodo en su descendencia no puede tener un puntero, en su vector de siguientes, que señale a dicho nodo. Donde descendencia son los nodos apuntados por su vector de siguientes, y los nodos apuntados por los vectores de siguientes de cada uno, y así recursivamente.
- 2) Para todo nodo n1 en el diccionario en cadena no existe otro nodo, distinto a el, en el diccionario, tal la definición de dicho nodo y la definición de n1 tengan aliasing.
- 3) Toda clave del conjunto de claves del diccionario en cadena, tiene, en el diccionario en cadena, una cadena de nodos que termina en una definición, y la cadena de letras que representa la cadena de nodos coincide con la clave, y además, la longitud de dicha clave es igual a la longitud de dicha cadena de nodos.
- 4) Para toda cadena que termine en una definición en el diccionario en cadena, existe una clave en el conjunto de claves del diccionario en cadena tal que la cadena de letras que representa la cadena de nodos, coincide con dicha clave.

$\text{Abs} : \text{dicc } d \longrightarrow \text{dicc}(\text{string}, \sigma)$

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

$\text{Abs}(d) \equiv \text{def?}(\text{clave}, \text{d.dNodo}) = \text{Definido}(\text{clave}, \text{d.dNodo})$
 $\text{obtener}(\text{clave}, \text{d.dNodo}) = \text{Obtener}(\text{clave}, \text{d.dNodo})$

Algoritmos En esta sección se hace abuso de notación en los cálculos de álgebra de órdenes presentes en la justificaciones de los algoritmos. La operación de suma “+” denota secuencialización de operaciones con determinado orden de complejidad, y el símbolo de igualdad “=” denota la pertenencia al orden de complejidad resultante.

Algoritmos del módulo

iVacío() $\rightarrow \text{res} : \text{dicc}$

1: $\text{res.dNodo} \leftarrow \langle \text{Vacío}(), \text{NULL} \rangle$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iDefinir(in/out d : dicc, in c : string, in s : σ)

```

1: if !Definido?( $d, c$ ) then  $\triangleright \mathcal{O}(\text{Longitud}(c))$ 
2:   AgregarRapido( $d.claves, c$ )  $\triangleright \Theta(1)$ 
3: end if
4:  $actual \leftarrow d.dNodo$   $\triangleright \Theta(1)$ 
5:  $i \leftarrow 0$   $\triangleright \Theta(1)$ 
6:  $guardoI \leftarrow i$   $\triangleright \Theta(1)$ 
7: if  $d = NULL$  then  $\triangleright \Theta(1)$  Si el diccionario estaba vacío, le agrego un nodo
8:    $inicio \leftarrow Nodo$   $\triangleright \Theta(1)$ 
9:    $d.dNodo = inicio$   $\triangleright \Theta(1)$ 
10: else  $\triangleright$  Busco el nodo donde aparesca la ultima letra de la clave c
11:   while  $i < \text{Longitud}(c)$  do  $\triangleright \mathcal{O}(\text{Longitud}(c))$ 
12:     if  $actual.Siguientes[\text{int}([i])] = NULL$  then  $\triangleright \Theta(1)$ 
13:        $guardoI \leftarrow i$   $\triangleright \Theta(1)$ 
14:        $i \leftarrow \text{Longitud}(c)$   $\triangleright \Theta(1)$ 
15:     end if
16:      $actual \leftarrow actual.Siguientes[\text{int}([i])]$   $\triangleright \Theta(1)$ 
17:      $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
18:   end while
19:    $i \leftarrow guardoI$   $\triangleright \Theta(1)$ 
20: end if
21: if  $i = \text{Longitud}(c)$  then  $\triangleright \Theta(1)$  Si la clave está, quizá tiene otra definición: la reemplazo
22:    $actual.Definicion \leftarrow NULL$   $\triangleright \Theta(1)$ 
23: else
24:   while  $i < \text{Longitud}(c)$  do  $\triangleright \mathcal{O}(i\text{Longitud}(c))$  Si no estaba la clave completa, agrego nodos para completarla
25:      $nuevo \leftarrow Nodo$   $\triangleright \Theta(1)$ 
26:      $actual.Siguientes[\text{int}([i])] \leftarrow nuevo$   $\triangleright \Theta(1)$ 
27:      $actual \leftarrow actual.Siguientes[\text{int}([i])]$   $\triangleright \Theta(1)$ 
28:      $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
29:   end while
30: end if
31:  $actual.Definicion \leftarrow s$   $\triangleright \Theta(1)$ . Ya estoy en el nodo correcto, agrego la definición

```

Complejidad: $\mathcal{O}(\text{Longitud}(c))$

Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$, en su mayoría asignar o inicializar, y dos *while*, cada uno con costo, en el peor caso, $\mathcal{O}(\text{Longitud}(c))$. En el peor caso, el diccionario estará vacío, entonces entrará en los dos *while*, sumando $\mathcal{O}(\text{Longitud}(c))$ por cada uno (estarán sumando porque estos *while* no están anidados). Luego, como sabemos que, aplicando álgebra de ordenes, $\mathcal{O}(\text{Longitud}(c)) + \Theta(1) = \mathcal{O}(\text{Longitud}(c))$, podemos decir que sólo nos quedará: $\mathcal{O}(\text{Longitud}(c)) + \mathcal{O}(\text{Longitud}(c)) = \mathcal{O}(\text{Longitud}(c))$.

iDefinido?(in/out d : dicc, in c : string) $\rightarrow res$: bool

```

1:  $actual \leftarrow d.dNodo$   $\triangleright \Theta(1)$ 
2:  $i \leftarrow 0$   $\triangleright \Theta(1)$ 
3: while  $i < Longitud(c)$  do  $\triangleright \mathcal{O}(Longitud(c))$ 
4:   if  $actual.Siguientes[int([i])] = NULL$  then  $\triangleright \Theta(1)$  Si falta tan sólo una letra, ya paro el ciclo.
5:      $i \leftarrow Longitud(c) + 1$   $\triangleright \Theta(1)$ 
6:   end if
7:    $actual \leftarrow actual.Siguientes[int([i])]$   $\triangleright \Theta(1)$ 
8:    $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
9: end while
10:  $res \leftarrow (i = Longitud(c) \wedge_L actual.Definicion \neq NULL)$   $\triangleright \Theta(1)$  Si se cumple es porque el while se ejecuto tantas veces como la longitud de la palabra, así que "la recorri" toda, y, además, su definición está donde debe estar.

```

Complejidad: $\mathcal{O}(Longitud(c))$

Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$, y un *while* de peor costo $\mathcal{O}(iLongitud(c))$. Por álgebra de ordenes sabemos que: $\mathcal{O}(Longitud(c)) + \Theta(1) = \mathcal{O}(Longitud(c))$, luego, en el caso en que recorra todas las posiciones de la clave, la complejidad será $\mathcal{O}(Longitud(c))$.

iObtener(in d : dicc, in c : string) $\rightarrow res$: σ

```

1:  $actual \leftarrow d.dNodo$   $\triangleright \Theta(1)$ 
2:  $i \leftarrow 0$   $\triangleright \Theta(1)$ 
3: while  $i < Longitud(c)$  do  $\triangleright \mathcal{O}(Longitud(c))$  Busco el final de la clave, para devolver su definición
4:    $actual \leftarrow actual.Siguientes[int(c[i])]$   $\triangleright \Theta(1)$ 
5:    $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
6: end while
7:  $res \leftarrow actual.Definicion$   $\triangleright \Theta(1)$ 

```

Complejidad: $\mathcal{O}(Longitud(c))$

Justificación: Como por restricción sé que la clave pasada debe estar en el diccionario, sé que tendré que iterar, en el *while*, hasta la longitud de la clave. Además, por álgebra de ordenes sé que: $\mathcal{O}(Longitud(c)) + \Theta(1) = \mathcal{O}(Longitud(c))$, y como todo, menos el *while*, es $\Theta(1)$, la complejidad es $\mathcal{O}(Longitud(c))$.

iBorrar(in/out d: dicc, in c: string)

```

1: ruptura  $\leftarrow$  d.dNodo  $\triangleright \Theta(1)$  La ruptura será el último nodo el cual o tiene una def, y no es la def de mi clave, o
   hay otra clave con el mismo prefijo que llega hasta la ruptura
2: letraRuptura  $\leftarrow$  buscarRuptura(d.dNodo, c, ruptura)  $\triangleright \mathcal{O}(\text{Longitud}(c))$  Acá lo busco, y además devuelvo en qué
   letra de la clave aparece dicho nodo
3: nodoDef  $\leftarrow$  buscarNodoDef(dNodo, c)  $\triangleright \mathcal{O}(\text{Longitud}(c))$  Busco el nodo que corresponde a la def de mi clave
4: if d.dNodo = ruptura then  $\triangleright \Theta(1)$  Si la raíz del dicc es la ruptura
5:   if cantidadNotNull(nodoDef) = 0 then  $\triangleright \Theta(1)$  Si la clave pasada no es prefijo de ninguna otra clave en mi
     dicc
6:     if cantidadNotNull(d.dNodo) = 1 then  $\triangleright \Theta(1)$  El caso en que tenga solo la clave pasada
7:       borrarDesde(d.dNodo, c, 0)  $\triangleright \mathcal{O}(\text{Longitud}(c))$ 
8:       d.dNodo = NULL  $\triangleright \Theta(1)$ 
9:     else  $\triangleright$  El caso en que tenga alguna clave mas
10:      borrarDesde(d.dNodo, c, 0)  $\triangleright \mathcal{O}(\text{Longitud}(c))$ 
11:    end if
12:  else  $\triangleright$  El caso en que la clave pasada sea prefijo de alguna otra clave en mi dicc
13:    nodoDef.Definicion = NULL  $\triangleright \Theta(1)$ 
14:  end if
15: else  $\triangleright$  Si la ruptura se ocasiona en un nodo posterior a la raíz
16:   if cantidadNotNull(nodoDef) = 0 then  $\triangleright \Theta(1)$  Si la clave pasada no es prefijo de ninguna otra
17:     borrarDesde(d.dNodo, c, letraRuptura)  $\triangleright \mathcal{O}(\text{Longitud}(c))$ 
18:   else
19:     nodoDef.Definicion = NULL  $\triangleright \Theta(1)$ 
20:   end if
21: end if
22: Eliminar(dicc.claves, c)  $\triangleright \Theta(1)$ 

```

Complejidad: $\mathcal{O}(\text{Longitud}(c))$

Justificación: El algoritmo tiene llamadas a funciones $\Theta(1)$, y a funciones $\mathcal{O}(\text{Longitud}(c))$. Notar que no tengo funciones que cuesten $\mathcal{O}(\text{Longitud}(c))$ anidadas. A la hora de buscar el nodo donde está la definición de la clave pasada, es decir *nodoDef*, ya sé que tengo complejidad $\mathcal{O}(\text{Longitud}(c))$, por lo tanto, por propiedades de álgebra de ordenes, la complejidad del algoritmo será $\mathcal{O}(\text{Longitud}(c))$, ya que: $\mathcal{O}(\text{Longitud}(c)) + \Theta(1) = \mathcal{O}(\text{Longitud}(c))$, y como mucho estaré sumando varias veces $\mathcal{O}(\text{Longitud}(c))$, por lo notado anteriormente, que dará $\mathcal{O}(\text{Longitud}(c))$.

iClaves(in/out d: dicc) \rightarrow res : conjuntoLineal(σ)

```

1: res  $\leftarrow$  d.claves  $\triangleright \Theta(1)$ 

```

Complejidad: $\Theta(1)$

iCopiar(in d: dic \rightarrow res : dicc(string, σ))

```

1: res.dNodo  $\leftarrow$  Vacio()  $\triangleright \Theta(1)$ 
2: claves  $\leftarrow$  Claves(d)  $\triangleright \Theta(1)$ 
3: while  $\neg$ EsVaco?(claves) do  $\triangleright \Theta(1)$ 
4:   Definir(res, primeraClave, iObtener(d, primeraClave))  $\triangleright \mathcal{O}(\text{Longitud}(c))$ 
5:   Eliminar(claves, primeraClave)  $\triangleright \mathcal{O}(\text{Longitud}(c))$ 
6: end while

```

Complejidad: $i\text{Cardinal}(\text{claves}) * \mathcal{O}(\text{Longitud}(c))$

Justificación: El algoritmo tiene llamadas a funciones $\Theta(1)$, un *while* que ejecuta siempre *Cardinal(claves)* veces, y dentro de este funciones $\mathcal{O}(\text{Longitud}(c))$. Luego, su complejidad debe ser $\text{Cardinal}(\text{claves}) * \mathcal{O}(\text{Longitud}(c))$.

$\bullet =_i \bullet(\text{in } d1: \text{dicc}, \text{in } d2: \text{dicc}) \rightarrow res: \text{bool}$

1: $res \leftarrow (d1.claves = d2.claves \wedge_L \text{mismasDefiniciones}(d1, d2))$ $\triangleright \mathcal{O}(\text{Longitud}(c) + \text{Cardinal}(claves))$

Complejidad: $\text{Cardinal}(claves) * \mathcal{O}(\text{Longitud}(c))$

Justificación: $\Theta(1) + \text{Cardinal}(claves) * \mathcal{O}(\text{Longitud}(c)) = \text{Cardinal}(claves) * \mathcal{O}(\text{Longitud}(c))$

buscarRuptura(in p : puntero(Nodo), in c : string, in/out $ruptura$: puntero(Nodo)) $\rightarrow res: \text{Nat}$

1: $actual \leftarrow p$ $\triangleright \Theta(1)$

2: $i \leftarrow 0$ $\triangleright \Theta(1)$

3: $letraRuptura \leftarrow 0$ $\triangleright \Theta(1)$

4: **while** $i < \text{Longitud}(c)$ **do** $\triangleright \mathcal{O}(\text{Longitud}(c))$

5: **if** $\text{cantidadNoNull}(actual) > 1 \vee actual.Definicion \leq \text{NULL}$ **then** $\triangleright \Theta(1)$

6: $ruptura \leftarrow actual$ $\triangleright \Theta(1)$

7: $letraRuptura \leftarrow i$ $\triangleright \Theta(1)$

8: **end if**

9: $actual \leftarrow actual.Siguientes[\text{int}(c[i])]$ $\triangleright \Theta(1)$

10: $i \leftarrow i + 1$ $\triangleright \Theta(1)$

11: **end while**

12: $res \leftarrow letraRuptura$ $\triangleright \Theta(1)$

Complejidad: $\mathcal{O}(\text{Longitud}(c))$

Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$, y un *while* de peor costo $\mathcal{O}(\text{Longitud}(c))$.

Por álgebra de ordenes sabemos que: $\mathcal{O}(\text{Longitud}(c)) + \Theta(1) = \mathcal{O}(\text{Longitud}(c))$, luego, en el caso en que recorra todas las posiciones de la clave, la complejidad será $\mathcal{O}(\text{Longitud}(c))$.

buscarNodoDef(in p : puntero(Nodo), in c : string) $\rightarrow res$: puntero(Nodo)

1: $actual \leftarrow p$ $\triangleright \Theta(1)$

2: $i \leftarrow 0$ $\triangleright \Theta(1)$

3: **while** $i < \text{Longitud}(c)$ **do** $\triangleright \mathcal{O}(\text{Longitud}(c))$

4: $actual \leftarrow actual.Siguientes[\text{int}(c[i])]$ $\triangleright \Theta(1)$

5: $i \leftarrow i + 1$ $\triangleright \Theta(1)$

6: **end while**

7: $res \leftarrow actual$ $\triangleright \Theta(1)$

Complejidad: $\mathcal{O}(\text{Longitud}(c))$

Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$, y un *while* de peor costo $\mathcal{O}(\text{Longitud}(c))$.

Por álgebra de ordenes sabemos que: $\mathcal{O}(\text{Longitud}(c)) + \Theta(1) = \mathcal{O}(\text{Longitud}(c))$, luego, en el caso en que recorra todas las posiciones de la clave, la complejidad será $\mathcal{O}(\text{Longitud}(c))$.

cantidadNoNull(in p : puntero(Nodo)) $\rightarrow res$: nat

1: $contador \leftarrow 0$ $\triangleright \Theta(1)$

2: $i \leftarrow 0$ $\triangleright \Theta(1)$

3: **while** $i < 256$ **do** $\triangleright \Theta(1)$

4: **if** $p.Siguientes[i] \neq \text{NULL}$ **then** $\triangleright \Theta(1)$

5: $contador \leftarrow contador + 1$ $\triangleright \Theta(1)$

6: **end if**

7: $i \leftarrow i + 1$ $\triangleright \Theta(1)$

8: **end while**

9: $res \leftarrow contador$ $\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$, y un *while* que será ejecutado 256 veces, y sabemos que $\Theta(256) = \Theta(1)$. Luego, la complejidad de este algoritmo será $\Theta(1)$.

borrarDesde(in *actual* : puntero(Nodo), in *c* : string, in *i* : nat)

```

1: letraRuptura ← i ▷ Θ(1)
2: while i < Longitud(c) do ▷ O(Longitud(c))
3:   if i = Longitud(c) − 1 then ▷ Θ(1)
4:     actual ← NULL ▷ Θ(1)
5:   else
6:     if i < Longitud(c) − 1 then ▷ Θ(1)
7:       actual ← actual.Siguientes[int(c[letraRuptura])].Siguientes[int(c[i + 1])] ▷ Θ(1)
8:     end if
9:   end if
10:  i ← i + 1 ▷ Θ(1)
11: end while

```

Complejidad: $\mathcal{O}(\text{Longitud}(c))$

Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$, y un *while* de peor costo $\mathcal{O}(\text{Longitud}(c))$. Por álgebra de ordenes sabemos que: $\mathcal{O}(\text{Longitud}(c)) + \Theta(1) = \mathcal{O}(\text{Longitud}(c))$, luego, en el caso en que recorra todas las posiciones de la clave, la complejidad será $\mathcal{O}(\text{Longitud}(c))$.

mismasDefiniciones(in *d1* : dicc, in *d2* : dicc) → *res* : bool

```

1: claves ← d1.claves ▷ Θ(1)
2: while ¬EsVaco?(claves) do ▷ Θ(Cardinal(claves))
3:   if Obetener(d1, primeraClave) = Obtener(d2, primeraClave) then ▷ O(Longitud(c))
4:     contador ← contador + 1 ▷ Θ(1)
5:   end if
6:   Eliminar(claves, primeraClave) ▷ Θ(1)
7: end while
8: res ← Cardinal(d1.claves) = contador ▷ Θ(1)

```

Complejidad: $\text{Cardinal}(\text{claves}) * \mathcal{O}(\text{Longitud}(c))$

Justificación: El algoritmo tiene llamadas a funciones $\Theta(1)$, un *while* que ejecuta siempre $\text{Cardinal}(\text{claves})$ veces, y dentro de este funciones $\mathcal{O}(\text{Longitud}(c))$. Luego, su complejidad debe ser $\text{Cardinal}(\text{claves}) * \mathcal{O}(\text{Longitud}(c))$.

6. Módulo Juego

1. pokemonesCapturados es Nat
2. matriz(α) es Vector(Vector(α))

Interfaz

parámetros formales

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

se explica con: JUEGO.

generos: Juego.

Operaciones basicas de Juego

MAPA(**in** $j : \text{Juego}$) $\rightarrow res : \text{mapa}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} j.\text{mapa}\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve el mapa del juego

JUGADORES(**in** $j : \text{Juego}$) $\rightarrow res : \text{itConj}(\text{jugador})$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{CrearIt}(e.\text{jugadores})\}$
Complejidad: $\Theta(1)$

Descripción: Devuelve un iterador al conjunto de jugadores.

Aliasing: El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin utilizar la funcion ELIMINARSIGUIENTE.

ESTACONECTADO(**in** $e : \text{Jugador}$, **in** $j : \text{Juego}$) $\rightarrow res : \text{Bool}$
Pre $\equiv \{e \in \text{jugadores}(j)\}$
Post $\equiv \{res =_{\text{obs}} \text{iesimo}(j, e).\text{conectado}\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve true si el jugador esta conectado

SANCIONES(**in** $e : \text{Jugador}$, **in** $j : \text{Juego}$) $\rightarrow res : \text{Nat}$
Pre $\equiv \{e \in \text{jugadores}(j)\}$
Post $\equiv \{res =_{\text{obs}} \text{iesimo}(j, e).\text{sanciones}\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve la posicion del jugador

POSICION(**in** $e : \text{Jugador}$, **in** $j : \text{Juego}$) $\rightarrow res : \text{coor}$
Pre $\equiv \{e \in \text{jugadores}(j) \wedge_{\text{L}} \text{estaConectado}(e, j)\}$
Post $\equiv \{res =_{\text{obs}} \text{iesimo}(j, e).\text{posicion}\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve la coordenada correspondiente a la posicion del jugador

POKÉMONS(**in** $e : \text{jugador}$, **in** $j : \text{juego}$) $\rightarrow res : \text{itConj}(\langle \text{pokémon}, \text{nat} \rangle)$
Pre $\equiv \{e \in \text{jugadore}(j)\}$
Post $\equiv \{res = \text{CrearIt}(\text{clasificarPokemons}(\text{pokemons}(e, j)))\}$
Complejidad: $\Theta(1)$

Descripción: Retorna un iterador al conjunto de tuplas $\langle \text{pokémon}, \text{cantidad} \rangle$ del jugador e .

Aliasing: El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE.

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

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

Post $\equiv \{\text{alias}(res, \text{expulsados}(j))\}$

Complejidad: $\Theta(1)$

Descripción: Retorna el conjunto de jugador expulsados por referencia.

Aliasing: ??

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

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

Post $\equiv \{\text{alias}(res, \text{posConPokémons}(j))\}$

Complejidad: $\Theta(1)$

Descripción: Retorna el conjunto de coordenadas donde hay pokémons por referencia.

Aliasing: ??

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

Pre $\equiv \{c \in \text{posConPokémons}(j)\}$

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

Complejidad: $\Theta(??)$

Descripción: Retorna por copia el pokémon que se encuentra en la coordenada c .

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

Pre $\equiv \{c \in \text{posConPokémons}(j)\}$

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

Complejidad: $\Theta(??)$

Descripción: Retorna por copia la cantidad de movimientos para capturar al pokémon que se encuentra en la coordenada c .

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

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

Post $\equiv \{\text{mapa}(res) = m \wedge \emptyset?(jugadores(res)) \wedge \emptyset?(\text{expulsados}(res)) \wedge \emptyset?(\text{posConPokemons}(res))\}$

Complejidad: $\mathcal{O}(\#coordenadas(m))$

Descripción: agrega un pokemon en una coordenada valida al juego.

AGREGARPOKEMON(**in** p : Pokemon, **in** c : Coordenada, **in/out** j : Juego)

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

Post $\equiv \{Ag(c, \text{posConPokemons}(j))\}$

Complejidad: $\mathcal{O}(|P| + |EC| * \log(|EC|))$

Descripción: agrega un pokemon en una coordenada valida al juego.

AGREGARJUGADOR(**in/out** j : Juego)

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

Post $\equiv \{Ag(\text{ProxID}(j), \text{jugadores}(j))\}$

Complejidad: $\mathcal{O}(\#jugadores(j))$

Descripción: agrega un nuevo jugador, designandole la proxima ID disponible.

CONECTARSE(**in** e : jugador **in** c : coord **in/out** j : Juego)

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

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

Complejidad: $\Theta(\log(EC))$

Descripción: Conecta un jugador al juego.

DESCONECTARSE(**in** e : jugador **in/out** j : Juego)

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

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

Complejidad: $\Theta(\log(EC))$

Descripción: Desconecta un jugador al juego.

MOVERSE(**in** e : jugador **in/out** c : coord **in/out** j : Juego)

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

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

Complejidad: $\Theta(\log(EC))$

Descripción: Desconecta un jugador al juego.

PUDOAGREGARPOKEMON(**in** c : Coordenada, **in/out** j : Juego) $\rightarrow res$: bool

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

Post $\equiv \{res = c \notin posConPokemons(j)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve verdadero si la coordenada pasada por parametro no está en el juego.

HAYPOKEMONCERCANO(**in** c : Coordenada, **in** j : Juego) $\rightarrow res$: bool

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

Post $\equiv \{res = \neg \phi?(buscarPokemonsCercano(c, posConPokemons(j), j))\}$

Complejidad: $\Theta(1)$

Descripción: devuelve verdadero si hay algún pokemon cercano a la coordenada pasada.

POSPOKEMONCERCANO(**in** c : Coordenada, **in** j : Juego) $\rightarrow res$: Coordenada

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

Post $\equiv \{res = dameUno(buscarPokemonsCercano(c, posConPokemons(j), j))\}$

Complejidad: $\Theta(1)$

Descripción: devuelve una coordenada, cercana a la coordenada pasada, donde hay un pokemon.

ENTRENADORESPOSIBLES(**in** c : Coordenada, **in** es : conj(jugador), **in** j : Juego) $\rightarrow res$: conj(jugador)

Pre $\equiv \{hayPokemonCercano(c, j) \wedge es \subseteq jugadoresConectados(j)\}$

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

Complejidad: $\Theta(1)$

Descripción: devuelve un conjunto con los jugadores cercanos a un pokemon.

INDICEDEAREZA(**in** p : pokemon **in** j : Juego) $\rightarrow res$: Nat

Pre $\equiv \{p \in TodosLosPokemons\}$

Post $\equiv \{res =_{\text{obs}} indiceDeRareza(p, j)\}$

Complejidad: $\Theta(1)$

Descripción: Devuelve la rareza de un pokemon.

Representación

Para la representacion del juego utilizamos una tupla que se compone de los siguientes elementos:

- **mapa:** El mapa pasado como parametro para la inicializacion del juego.
- **jugadores:** Un vector que contiene la informacion de los jugadores que fueron agregados al juego. En este vector estan tanto los jugadores en juego como los expulsado. El vector permite darnos acceso en tiempo constante a la informacion de los jugadores como sanciones, posicion y ver si esta conectado. En cada posicion del vector tenemos un tipo **infoJug** que es una tupla de:
 1. **conectado:** un valor booleano cuyo valor es true si esta conectado y false de lo contrario.
 2. **canciones:** un natural que corresponde al numero de sanciones acumuladas por el jugador.
 3. **posicion:** la coordenada correspondiente a la posicion del jugador en el mapa.
 4. **pokemons:** un conjunto lineal de pokes donde pokes es una tupla $\langle \text{tipo}, \text{nat} \rangle$ donde el primer valor corresponde al tipo de pokemon y el segundo a la cantidad de ese tipo que posee el jugador.
 5. **pokesRapido:** un diccionario implementado sobre un trie cuyas claves son pokemon y su significado es un iterador al conjunto **pokemons**. Esta estructura permite un algoritmo de busqueda sobre el conjunto **pokemons** cuya complejidad es $\mathcal{O}(|P|)$, donde P es el nombre mas largo de un pokemon en el conjunto.
 6. **prioridad:** si el jugador esta conectado, contiene un iterador a la posicion del jugador en la cola de prioridad del pokemon en rango de este, si es que hay uno.
 7. **posMatriz:** si el jugador esta conectado, contiene un iterador al conjunto de jugadores dentro de la posicion en la estructura **matrizJugadores** correspondiente con su posicion actual.
 8. **lugarNoExpulsado:** si es que el jugador no esta expulsado, contiene un iterador a su posicion en el conjunto **jugadoresNoExpulsados**.

- **jugadoresNoExpulsados:** un conjunto lineal que contiene solo los jugadores que fueron agregados al juego y acumularon menos de 5 sanciones.
- **expulsados:** un conjunto lineal que contiene solo los jugadores que fueron agregados al juego, y acumularon 5 o mas sanciones.
- **matrizDeJugadores:** una matriz en cuyas posiciones hay conjuntos dentro de los cuales solo se encuentran los jugadores conectados cuya posicion coincide con esta.
- **matrizPokemon:** una matriz en donde cada una de sus posiciones contiene una estructura de tipo **infoMatrizPoke** que es una tupla de:
 1. **hayPoke:** un valor booleano que indica cuyo valor es true si hay un pokemon en esa posicion del mapa.
 2. **iterador:** un iterador a la la posicion del diccionario posPokemons correspondiente a esa coordenda.
- **posPokemons:** un diccionario lineal cuyas claves corresponden a las coordenadas de los pokemons salvajes del juego.El significado contiene una estructura de tipo **infoCoord** que es tupla de:
 1. **tipo:** el pokemon que se encuentra en esa coordenada.
 2. **cantMovCapt:** un natural correspondiente al numero de movimientos fuera del rango del pokemon que son nesarios para la captura de este.
 3. **colaDePrioridad:** una cola de prioridad que contiene a todos los jugadores en rango del pokemon. El criterio de orden para esta cola es la cantidad de pokemons de un jugador, siendo el jugador con menos pokemons el que ocupa la primera posicion.
- **pokemonsTotales:** un diccionario cuyas claves corresponden a los pokemons en juego.Como significado contienen una estructura de tipo **infoPoke** que es una tupla de:
 1. **cant:** un natural corespondiente a la catidad de pokemons,capturados y/o salvajes de ese tipo en juego.
 2. **pos:** un conjunto lineal de iteradores. Cada uno de estos itradores apunta a un elemento del diccionario **posPokemons**, cuyo tipo es del tipo de la clave.

Esta estructura permite un algoritmo de busqueda sobre el diccionario **posPokemons** cuya complejidad es $O(|P|)$, donde P es el nombre mas largo de un pokemon en el conjunto.

- **cantidadPokeTotal:** un natural correspondiente a la cantidad de pokemons totales en juego , es decir que no fueron capturados por un entrenados y/o que si fueron capturados el jugador no fue eliminado.

Juego se representa con juego

```

donde juego es tupla(mapa: mapa
    , jugadores: vector(infoJug), jugadoresNoExpusados: conj(jugador) , expulsados:
    conj(jugador), matrizJugadores: matriz(conj(jugador))
    , matrizPokemons: matriz(infoMatrizPoke)
    , posPokemons: DiccionarioLineal(coordenada,infoCoord), pokemonsTotales:
    diccTrie(pokemon,infoPoke) , cantidadPokeTotal: nat
)
donde infoMatrizPoke es tupla(hayPoke: bool , iterador: itDicc(coordenada,infoCoord) )
donde infoJug es tupla(conectado: bool , sanciones: nat , posicion: coord , pokemons:
    ConjuntoLineal(Pokes) , pokesRapido: DiccTrie(pokemon,itConj(pokes)) ,
    prioridad: itHeapModificable(pokemonsCapturados,jugador)) , posMatriz:
    itConj(jugador) , lugarNoExpulsado: itConj(jugador)
)
donde pokes es tupla(tipo: pokemon , cant: nat )
donde infoCoord es tupla(tipo: pokemon , cantMovCapt: nat , colaPrioridad: Heap Modificable(jugador) )
donde infoPoke es tupla(cant: nat , pos: conj(it Dicc(coord,infoCoord)) )

```

Rep : juego \rightarrow bool

$\text{Rep}(j) \equiv \text{true} \iff (0) \wedge_L (1) \wedge (2) \wedge (3) \wedge (4) \wedge (5) \wedge (6) \wedge_L (7) \wedge_L ? (8)$

Donde (0), (1), (2), (3), (4), (5), (6), (7) y (8) son:

0) $(\forall C : \text{conjuntoLineal}(\text{jugador}))((\exists c : \text{coordenada})(j.\text{matrizJugadores}[\text{longitud}(c)][\text{latitud}(c)] = C) \Rightarrow (\forall e : \text{nat})(\text{Pertenece?}(e, C) \Rightarrow 0 \leq e \leq \text{Longitud}(j.\text{jugadores}) - 1))$

1) $(\forall e : \text{nat})(0 \leq e \leq \text{Longitud}(j.\text{jugadores}) - 1 \Rightarrow_L (2) \wedge (3))$

2) $(j.\text{jugadores}[e].\text{sanciones} < 5 \iff \text{Pertenece?}(j.\text{jugadoresNoExpulsados}, e) \wedge \neg \text{Pertenece?}(j.\text{expulsados}, e) \wedge_L (2\text{bis}))$

2bis) $(\text{Pertenece?}(j.\text{jugadoresNoExpulsados}, e) \Rightarrow (2a) \wedge (j.\text{jugadores}[e].\text{conectado} \Rightarrow (\exists c : \text{coordenada})(\text{posExistente}(c, j.\text{mapa}) \Rightarrow_L (2b) \wedge_L (2c) \wedge (2d) \wedge_L (2e)) \wedge (2f) \wedge 2g))$

2a) $j.\text{jugadores}[e].\text{lugarNoExpulsado}$ señala a la e correspondiente en $j.\text{jugadoresNoExpulsados}$

2b) $j.\text{jugadores}[e].\text{posicion} = c \wedge_L \text{Pertenece?}(j.\text{matrizJugadores}[\text{longitud}(c)][\text{latitud}(c)], e)$

2c) $j.\text{jugadores}[e].\text{posMatriz}$ señala a la e correspondiente en $j.\text{matrizJugadores}[c.\text{longitud}][c.\text{latitud}]$

2d) $(\forall c' : \text{coordenada})(\text{Pertenece?}(\text{coordenadas}(j.\text{mapa}), c') \wedge c' \neq c \Rightarrow \neg \text{Pertenece?}(j.\text{matrizJugadores}[c'.\text{longitud}][c'.\text{latitud}], e))$

2e) $\text{HayPokemonCercano}(c, j) \Rightarrow_L j.\text{jugadores}[e].\text{prioridad}$ señala al lugar correspondiente, donde esté la tupla en la que está e , en la cola de prioridad de $\text{PosPokemonCercano}(c, j)$

2f) $(\forall p : \text{tipo})(\text{Pertenece?}(p, j.\text{jugadores}[e].\text{pokesRapido}) \Rightarrow \text{Definido?}(j.\text{jugadores}[e].\text{pokesRapido}, p) \wedge \text{Obtener}(j.\text{jugadores}[e].\text{pokesRapido}, p)$ señala a la tupla correspondiente en $j.\text{jugadores}[e].\text{pokemons}$ la cual como tipo tiene a p)

3) $(j.\text{jugadores}[e].\text{sanciones} \geq 5 \iff \neg \text{Pertenece?}(j.\text{jugadoresNoExpulsados}, e) \wedge \text{Pertenece?}(j.\text{expulsados}, e) \wedge_L (3\text{bis}))$

3bis) $\neg j.\text{jugadores}[e].\text{conectado} \wedge \text{Vacio?}(j.\text{jugadores}[e].\text{pokemons}) \wedge \text{Vacio?}(\text{Claves}(j.\text{jugadores}[e].\text{pokesRapido}), j.\text{jugadores}[e].\text{prioridad} = \text{NULL} \wedge j.\text{jugadores}[e].\text{posMatriz} = \text{NULL} \wedge j.\text{jugadores}[e].\text{lugarNoExpulsado} = \text{NULL})$

4) $(\forall c : \text{coord})\text{esta?}(j.\text{matrizPokemon}, \text{Longitud}(c)) \wedge_L \neg \text{vacía?}(j.\text{matrizPokemon}[\text{Longitud}(c)]) \wedge_L \text{esta?}(j.\text{matrizPokemon}[\text{Longitud}(c)], \text{Latitud}(c)) \Rightarrow_L (j.\text{matrizPokemon}[\text{Longitud}(c)][\text{Latitud}(c)].\text{hayPoke} \iff \text{definido?}(j.\text{posPokemons}, c) \vee (j.\text{matrizPokemon}[\text{Longitud}(c)][\text{Latitud}(c)].\text{hayPoke} = \text{false} \wedge j.\text{matrizPokemon}[\text{Longitud}(c)][\text{Latitud}(c)].\text{iterador} = \text{NULL}))$

4b) $(\forall c : \text{coord})\text{esta?}(j.\text{matrizPokemon}, \text{Longitud}(c)) \wedge_L \neg \text{vacía?}(j.\text{matrizPokemon}[\text{Longitud}(c)]) \wedge_L \text{esta?}(j.\text{matrizPokemon}[\text{Longitud}(c)], \text{Latitud}(c)) \wedge j.\text{matrizPokemon}[\text{Longitud}(c)][\text{Latitud}(c)].\text{hayPoke} \Rightarrow_L (\text{Siguiente}(j.\text{matrizPokemon}[\text{Longitud}(c)][\text{Latitud}(c)].\text{iter}) = \text{obtener?}(j.\text{posPokemons}, c))$

5a) $(\forall c : \text{coord})\text{definido?}(j.\text{posPokemons}, c) \Rightarrow (\text{pertenece?}(\text{coordenadas}(m), c) \wedge (\nexists c' : \text{coord})(\text{definido?}(j.\text{posPokemons}, c') \wedge \text{distEuclidean}(c, c') \leq 25))$

5b) $(\forall c : \text{coord})\text{definido?}(j.\text{posPokemons}, c) \Rightarrow_L \text{obtener}(j.\text{posPokemons}, c).\text{cantMovCapt} \leq 10 \iff ((\exists e : \text{jugador})\text{pertenece}(j.\text{jugadoresNoEliminados}, e) \wedge_L j.\text{jugadores}[e].\text{conectado} \wedge \text{distEuclidean}(c, j.\text{jugadores}[e].\text{posicion}) \leq 4)$

5c) $(\forall c : \text{coord})\text{definido?}(j.\text{posPokemons}, c) \Rightarrow_L \text{dameJugadoresHeap}(\text{obtener}(j.\text{posPokemons}, c).\text{colaDePr} \text{ entrenadoresPosibles}(c, j))$

6) $(\forall p : \text{pokemon})(\text{definido?}(j.\text{pokemonsTotales}, p) \iff ((\exists e : \text{jugador})\text{pertenece}(j.\text{jugadoresNoEliminados}, e) \wedge_L \text{definido?}(j.\text{jugadores}[e].\text{pokemons}, p) \vee ((\exists c : \text{coord})\text{definido?}(j.\text{posPokemons}, c) \wedge_L \text{obtener}(j.\text{posPokemons}, c).\text{tipo} = p)))$

6b) $(\forall p : \text{pokemon})(\text{definido?}(j.\text{pokemonsTotales}, p) \Rightarrow_L \# \text{obtener}(j.\text{pokemonsTotales}, p).\text{pos} = \text{pokesSalvajesDeTipo}(j.\text{posPokemons}, p))$

6c) $(\forall p : \text{pokemon})(\text{definido?}(j.\text{pokemonsTotales}, p) \Rightarrow_L \# \text{obtener}(j.\text{pokemonsTotales}, p).\text{cant} = \text{pokesSalvajesDeTipo}(j.\text{posPokemons}, p) + \text{cantTotalPokes}(j.\text{jugadores}, p))$

7) $(\forall e : \text{nat})(0 \leq e \leq \text{Longitud}(j.\text{jugadores}) - 1 \Rightarrow (\forall p : \text{tipo})(\text{Pertenece?}(p, j.\text{jugadores}[e].\text{pokesRapido}) \Rightarrow \text{Pertenece?}(\text{Claves}(j.\text{pokemonsTotales}), p)))$

8) $(\forall e : \text{nat})(0 \leq e \leq \text{Longitud}(j.\text{jugadores}) - 1 \Rightarrow (\forall p : \text{pokemon})(\text{Pertenece?}(p, j.\text{jugadores}[e].\text{pokesRapido}) \Rightarrow (\exists p' : \text{pokemon})(\text{Definido?}(j.\text{pokemonsTotales}, p') \wedge_L p' = p \Rightarrow \text{sumaPokesJugadorIgual}(p, j.\text{jugadores}, \text{Obtener}(j.\text{pokemonsTotales}, p').\text{pos}), j)) \vee \text{sumaPokesJugadorMenor}(p, j.\text{jugadores}, \text{Obtener}(j.\text{pokemonsTotales}, p').\text{Cardinal}(\text{Obtener}(j.\text{pokemonsTotales}, p').\text{pos}), j)))$


```

dameJugadoresHeap : colaDeProridad(jugadorHeap) : h → conj(nat)
dameJugadoresHeap(h) ≡ if vacio?(h) then ∅ else Ag(Proximo(h).id, dameJugadorHeap(desencolar(h))) fi
pokesSalvajesDeTipo : dicc(coord × infoCoord):d → nat

pokemon : p

pokesSalvajesDeTipo(d,p) ≡ pokesSalvajesDeTipoAux(d,p,claves(d))
pokesSalvajesDeTipoAux : dicc(coord × infoCoord):d → nat

pokemon : p

× conj(coord) : C
pokesSalvajesDeTipoAux(d,p,C) ≡ if ∅?(C) then
  0
else
  if obtener(d,dameUno(C)).tipo = p then 1 else 0 fi +pokesSalvajesDe-
  TipoAux(d,p,sinUno(C))
fi

sumaPokesJugadorIgual : tipo : p × vector(jugador) : v × nat : d × nat : n × nat : c × juego : j → bool
sumaPokesJugadorIgual(p,v,d,n,j) ≡ c = 0 ∧ cantTotalPokes(p,v,n,j) = d
sumaPokesJugadorMenor : tipo : p × vector(jugador) : v × nat : d × nat : n × nat : c × juego : j → bool
sumaPokesJugadorMenor(p,v,d,n,j) ≡ c > 0 ∧ cantTotalPokes(p,v,n,j) = d
cantTotalPokes : tipo : p × vector(jugador) : v × nat : n × juego : j → nat
cantTotalPokes(p,v,n,j) ≡ if n = -1 then
  0
else
  if ¬Pertenece?(j.expulsados,j.jugadores[n]) then
    Obtener(v[n].pokesRapido,p).cant + sumaPokesJugadorIgual(p,v,n-1,j)
  else
    sumaPokesJugadorIgual(p,v,n-1,j)
  fi
fi

Abs : juego j → juego {Rep(j)}
Abs(j) ≡ j : juego |
  e.mapa = mapa(j)
  e.jugadores = jugadores(j) ∧L
  (∀i : jugador) i ∈ jugadores(j) ⇒L estaConectado(i,j) = e.jugadores[i].estaConectado ∧ sanciones(i,j) =
  e.jugadores[i].sanciones) ∧
  posicion(i,j) = e.jugadores[i].pokemons ∧ estaConectado(i,j) ⇒L posicion(i,j) = e.jugadores[i].posicion
  expulsados(j) = e.expulsados
  posConPokemons(j) = e.posPokemons().Claves() ∧L (∀c : coord) c ∈ posConPokemons(j) ⇒L
  pokemonEnPos(c,j) = e.posConPokemons.Obtener(c,j).tipo ∧ cantMovimientosParaCaptura(c,j) =
  e.posPokemons.Obtener(c).cantMovCapt

```

Algoritmos

Algoritmos del modulo

iMapa(in j : juego) → res:map

1: $res \leftarrow j.mapa$
Complejidad: $\mathcal{O}(1)$

iJugadores(in j : juego) → res:it ListaJug

1: $res \leftarrow CrearIt(e.jugadoresNoExpulsados)$
Complejidad: $\mathcal{O}(1)$

iEstaConectado(in e : jugador, in j : juego) $\rightarrow res:bool$

1: $res \leftarrow (j.jugadores[e]).conectado$
Complejidad: $\mathcal{O}(1)$

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

1: $res \leftarrow (j.jugadores[e]).sanciones$
Complejidad: $\mathcal{O}(1)$

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

1: $res \leftarrow (j.jugadores[e]).posicion$
Complejidad: $\mathcal{O}(1)$

iPokemons(in e : jugador, in j : juego) $\rightarrow res:it \text{ multiconjunto}(pokemon)$

1: $res \leftarrow crearIt((j.jugadores[e]).pokemons)$
Complejidad: $\mathcal{O}(1)$

iExpulsados(in j : juego) $\rightarrow res:it \text{ conjunto}(jugador)$

1: $res \leftarrow CrearIt(j.expulsados)$
Complejidad: $\mathcal{O}(1)$

iPosConPokemons(in j : juego) $\rightarrow res:it \text{ conjunto}(coord)$

1: $res \leftarrow CrearIt(j.posPokemons)$
Complejidad: $\mathcal{O}(1)$

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

1: $it \leftarrow (j.matrizPokemons[longitud(c)][latitud(c)]).iterador \ \Theta(1)$
 2: $res \leftarrow siguienteSignificado(it).tipo \ \Theta(1)$
Complejidad: $\mathcal{O}(1)$

iCantMovimientosParaCaptura(in c : coord, in j : juego) $\rightarrow res:nat$

1: $it \leftarrow (j.matrizPokemons[longitud(c)][latitud(c)]).iterador \ \Theta(1)$
 2: $res \leftarrow siguienteSignificado(it).tipo \ \Theta(1)$
Complejidad: $\mathcal{O}(1)$

iCrearJuego(in $m : \text{mapa}$) $\rightarrow res : \text{juego}$

```

1:  $res.mapa \leftarrow m$ 
2:  $res.jugadoresNoExpulsados \leftarrow Vacio()$ 
3:  $res.jugadores \leftarrow Vacio()$ 
4:  $res.expulsados \leftarrow Vacio()$ 
5:  $res.matrizJugadores \leftarrow iCrearMatrizJug(m) \mathcal{O}(\#(coordenadas(m)))$ 
6:  $res.matrizJugadores \leftarrow iCrearMatrizPokes(m) \mathcal{O}(\#(coordenadas(m)))$ 
7:  $res.pokemons \leftarrow iVacio()$ 
8:  $res.posPokemons \leftarrow iVacio()$ 
9:  $res.pokemonsTotales \leftarrow iVacio()$ 
10:  $res.cantidadPokeTotal \leftarrow 0$ 
    Complejidad:  $\mathcal{O}(\#(coordenadas(m)))$ 

```

iCrearMatrizJug(in $m : \text{mapa}$) $\rightarrow res : \text{matriz}(\text{conj}(\text{jugador}))$

```

1:  $res \leftarrow Vacio()$ 
2:  $it \leftarrow coordenadas(m)$ 
3: while  $haySiguiete(it)$  do
4:    $c \leftarrow Siguiete(it)$ 
5:   if  $longitud(res) < Longitud(c)$  then
6:      $i \leftarrow longitud(res)$   $\triangleright \Theta(1)$ 
7:     while  $i \leq Longitud(c)$  do  $\triangleright \Theta(i)$ 
8:        $Agregar(res, i, iVacia())$   $\triangleright \Theta(Longitud(c))$ 
9:        $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
10:    end while
11:    if  $i < Longitud(res)$  then
12:      while  $i \leq Longitud(c)$  do
13:         $Agregar(res, i, Vacio())$   $\triangleright \Theta(Longitud(c))$ 
14:         $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
15:      end while
16:    end if
17:  end if
18:  if  $longitud(res[Longitud(c)]) < Latitud(c)$  then
19:     $j \leftarrow longitud(res[Longitud(c)])$   $\triangleright \Theta(1)$ 
20:    while  $j \leq Latitud(c)$  do  $\triangleright \Theta(1)$ 
21:       $Agregar(res[Longitud(c)], j, Vacio())$   $\triangleright \Theta(Latitud(c))$ 
22:       $j \leftarrow j + 1$   $\triangleright \Theta(1)$ 
23:    end while
24:    if  $i < Longitud(Longitud)$  then
25:      while  $i \leq Longitud(c)$  do
26:         $Agregar(res, i, Vacio())$   $\triangleright \Theta(Longitud(c))$ 
27:         $j \leftarrow j + 1$   $\triangleright \Theta(1)$ 
28:      end while
29:    end if
30:  end if
31:   $Avanzar(it)$ 
32: end while
    Complejidad:  $\mathcal{O}(\#(coordenadas(m)))$ 

```

iCrearMatrizPokes(in m : mapa) \rightarrow res:matriz(infoMatrizPoke)

```

1:  $res \leftarrow Vacio()$ 
2:  $it \leftarrow coordenadas(m)$ 
3: while haySiguiente(it) do
4:    $c \leftarrow Siguiente(it)$ 
5:   if longitud(res) < Longitud(c) then
6:      $i \leftarrow longitud(res)$   $\triangleright \Theta(1)$ 
7:     while  $i \leq Longitud(c)$  do  $\triangleright \Theta(i)$ 
8:       Agregar(res, i, iVacia())  $\triangleright \Theta(Longitud(c))$ 
9:        $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
10:    end while
11:    if  $i < Longitud(res)$  then
12:      while  $i \leq Longitud(c)$  do
13:        Agregar(res, i, Vacio())  $\triangleright \Theta(Longitud(res))$ 
14:         $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
15:      end while
16:    end if
17:  end if
18:  if longitud(res[Longitud(c)]) < Latitud(c) then
19:     $j \leftarrow longitud(res[Longitud(c)])$   $\triangleright \Theta(1)$ 
20:    while  $j \leq Latitud(c)$  do  $\triangleright \Theta(1)$ 
21:      Agregar(res[Longitud(c)], j, < false, NULL >)  $\triangleright \Theta(Longitud(res.[Longitud(c)]))$ 
22:       $j \leftarrow j + 1$   $\triangleright \Theta(1)$ 
23:    end while
24:    if  $i < Longitud(Longitud)$  then
25:      while  $i \leq Longitud(c)$  do
26:        Agregar(res, j, < false, NULL >)  $\triangleright \Theta(Longitud(res))$ 
27:         $j \leftarrow j + 1$   $\triangleright \Theta(1)$ 
28:      end while
29:    end if
30:  end if
31:  Avanzar(it)
32: end while

```

Complejidad: $\mathcal{O}(\#(coordenadas(m)))$

iAgregarPokémon(in p : pokémon, in c : coor, in/out j : juego)

- 1: $j.cantidadPokeTotal \leftarrow j.cantidadPokeTotal + 1$ \triangleright Se aumenta la cantidad total de pokémons // $\Theta(1)$
- 2: $itHeapmod \leftarrow iCrearHeapPokemon(j, c)$ \triangleright Se crea el heap que contiene a todos los jugadores en radio 2 del pokémon // $\Theta(EC * \log(EC))$
- 3: $itDic \leftarrow iDefinirRapido(j.posPokemon, c, \langle p, 0, itNuevoHeap \rangle)$ \triangleright Se define la coordenada del pokémon y se coloca su infoCoord // $\Theta(1)$
- 4: $j.matrizPokemons[iLongitud(c)][iLatitud(c)] \leftarrow \langle true, dicLinealPoke \rangle$ \triangleright Se cambia *true* la posición donde está el pokémon y se pone un iterador al diccionario de coordenadas // $\Theta(1)$
- 5: **if** $iDefinido?(j.pokemonsTotales, p)$ **then** \triangleright Se verifica que esté definido el tipo pokémon p // $\Theta(iLongitud(p))$
- 6: $iObtener(j.pokemonsTotales, p).cant \leftarrow iObtener(j.pokemonsTotales, p).cant + 1$ \triangleright Se incrementa la cantidad de pokémones del tipo agregado // $\Theta(iLongitud(p))$
- 7: $AgregarRapido(iObtener(j.pokemonsTotales, p).pos, dicLinealPoke)$ \triangleright Se agrega el iterador al conjunto de posiciones // $\Theta(iLongitud(p))$
- 8: **else**
- 9: $iDefinir(j.pokemonsTotales, p, \langle 1, iVacio() \rangle)$ \triangleright Se define el nuevo tipo de pokemon // $\Theta(iLongitud(p))$
- 10: $AgregarRapido(iObtener(j.pokemonsTotales, p).pos, dicLinealPoke)$ \triangleright Se agrega el iterador al conjunto de posiciones // $\Theta(iLongitud(p))$
- 11: **end if**

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

Justificación: El algoritmo cuenta con una operación que tiene costo $\Theta(EC * \log(EC))$, tres operaciones con costo $\Theta(|P|)$ y otras operaciones con costo $\Theta(1)$. Luego sumando las complejidades se obtiene que la complejidad del algoritmo es $\Theta(|P| + EC * \log(EC))$.

iCrearHeapPokemon(in j : juego, in c : coor) $\rightarrow res$: heapMod(α)

- 1: $heapmod \leftarrow pokeHeap \leftarrow Vacía()$ \triangleright // $\Theta(1)$
- 2: $i \leftarrow -2$ \triangleright // $\Theta(1)$
- 3: **while** $i \leq 2$ **do** \triangleright // $\Theta(1)$
- 4: $j \leftarrow -2$ \triangleright // $\Theta(1)$
- 5: **while** $j \leq 2$ **do** \triangleright // $\Theta(1)$
- 6: $coor \leftarrow \langle longitud(c) + i, latitud(c) + j \rangle$ \triangleright // $\Theta(1)$
- 7: **if** $posExistente(coor, j.mapa) \wedge DistEuclidea(c, coor) \leq 2$ **then** \triangleright Vemos que la coordenada exista en el mapa y esté en rango del pokémon // $\Theta(1)$
- 8: $itConj(Jugador) \leftarrow iCrearIt(j.matrizJugadores[i][j])$ \triangleright // $\Theta(1)$
- 9: **while** $HaySiguiente(itJugHeap)$ **do** \triangleright Analizamos los jugadores que hay en esa coordenada // $\Theta(1)$
- 10: $id \leftarrow Siguiente(itJugHeap)$ \triangleright $\Theta(1)$
- 11: $itHeapmod \leftarrow Encolar(\langle \#(jugadores[id].pokemons), id \rangle, pokeHeap)$ \triangleright Agregamos al jugador a la cola de prioridad del pokémon y creamos un iterador al heap // $\Theta(\log(EC))$
- 12: $jugadores[id].prioridad \leftarrow itPokeHeap$ \triangleright Colocamos el iterador info del jugador // $\Theta(1)$
- 13: $Avanzar(itJugHeap)$ \triangleright // $\Theta(1)$
- 14: **end while**
- 15: **end if**
- 16: $j \leftarrow j + 1$ \triangleright // $\Theta(1)$
- 17: **end while**
- 18: $i \leftarrow i + 1$ \triangleright // $\Theta(1)$
- 19: **end while**

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

Justificación: Todas las operaciones tienen costo $\Theta(1)$, excepto *Encolar* que tiene costo $\Theta(\log(EC))$, donde *EC* es la cantidad de elementos en la cola. El algoritmo tiene dos ciclos, pero tienen una cantidad predeterminada de iteraciones, tanto *i* como *j* toma valores en el intervalo $[-2, 2]$. Luego la complejidad depende de la cantidad de elementos (*jugadores*) que se coloquen en la cola. Denominamos *EC* a la cantidad de jugadores que se encuentran en rango del pokémon y deben ser colocados en la cola de prioridad. Ya que *Encolar* un jugador que tiene costo $\Theta(\log(EC))$ y hay *EC* jugadores, la complejidad es $\Theta(EC * \log(EC))$.

iAgregarJugador(in/out j : juego) \rightarrow res : Nat

- 1: $res \leftarrow j.jugadores.Longitud()$ \triangleright El id del nuevo jugador agregado es la cantidad de elemento en el vector
- 2: $estaConectado \leftarrow false$ \triangleright Al momento de ser agregado el jugador todavía no está conectado
- 3: $sanciones \leftarrow 0$
- 4: $pos \leftarrow CrearCoord(42, 42)$ \triangleright Posición Hardcodeada arbitraria
- 5: $pokes \leftarrow Vacio()$ \triangleright No tiene pokemones capturados
- 6: $pokesRapido \leftarrow vacio()$ \triangleright No tiene pokemones capturados
- 7: $posicioneCola \leftarrow NULL$
- 8: $posMatriz \leftarrow NULL$ \triangleright Se le coloca una posición arbitraria pero no se lo coloca en la matriz de jugadores
- 9: $itNoExpulsado \leftarrow AgregarRapido(j.jugadoresNoExpulsados, res)$
- 10: $nuevo \leftarrow \langle estaConectado, sanciones, pos, pokes, pokesRapido, posicioneCola, posMatriz, itNoExpulsado \rangle$
- 11: $j.jugadores.AgregarAtras(nuevo)$ $\triangleright \mathcal{O}(Longitud(j.jugadores))$

Complejidad: La complejidad del algoritmo es $\mathcal{O}(J)$ donde J es la cantidad de jugadores agregados al juego pues todas las operaciones excepto la ultima tienen complejidad $\mathcal{O}(1)$ y la ultima operacion agregar atras del modulo vector que tiene complejidad $\mathcal{O}(n)$ donde n es la longitud del vector. En este caso la longitud del vector corresponde a la cantidad de jugadores agregados al juego hasta el momento por lo tanto la complejidad es $\mathcal{O}(J)$.

iConectarse(in e : jugador, in c : coord in/out j : juego)

- 1: $j.jugadores[e].estado.estaConectado \leftarrow true$
- 2: $j.jugadores[e].estado.posicion \leftarrow c$
- 3: $itConj \leftarrow AgregarRapido(j.matrizJugadores[c.longitud][c.latitud], e)$ $\triangleright \mathcal{O}(copy(e)) = \Theta(1)$, ya que e es nat
- 4: $j.jugadores[e].posMatriz \leftarrow itConj$
- 5: **if** hayPokemonCercano(c, j) **then**
- 6: $pokePosicion \leftarrow posPokemonCercano(c, j)$
- 7: $nueva \leftarrow \langle CantidadPokemons(j.jugadores[e].pokemonesCapturados, j), e \rangle$
- 8: $itHeap \leftarrow encolar(nueva, j.matrizPokemon[pokePosicion.longitud][pokePosicion.latitud].colaPrioridad)$ $\triangleright \mathcal{O}(\log(\#nodos(heap)))$
- 9: $j.jugadores[e].posicionCola \leftarrow itHeap$
- 10: **end if**

Complejidad: La complejidad del algoritmo es $\mathcal{O}(\log(EC))$ donde EC es la cantidad de jugadores esperando atrapar un pokemon pues todas las operaciones excepto la ultima tienen complejidad $\mathcal{O}(1)$ y la ultima es la operacion encolar del modulo heap que tiene complejidad $\mathcal{O}(\log(\#nodos(heap)))$. En este caso la cantidad de elementos en el heap corresponde a la cantidad de jugadores esperando atrapar un pokemon, por lo tanto la complejidad de la funcion es $\mathcal{O}(\log(EC))$.

iDesconectarse(in e : jugador in/out j : juego)

- 1: $j.jugadores[e].estado.estaConectado \leftarrow false$
- 2: $EliminarSiguierte(j.jugadores[e].posMatriz)$ $\triangleright \mathcal{O}(1)$
- 3: $j.jugadores[e].posMatriz \leftarrow NULL$
- 4: **if** hayPokemonCercano(c, j) **then**
- 5: $EliminarSiguierte(j.jugadores[e].posicionCola)$ $\triangleright \mathcal{O}(\log(\#nodos(heap)))$
- 6: $j.jugadores[e].posicionCola \leftarrow NULL$
- 7: **end if**

Complejidad: La complejidad del algoritmo es $\mathcal{O}(\log(EC))$ donde EC es la cantidad de jugadores esperando atrapar un pokemon pues todas las operaciones excepto la ultima tienen complejidad $\mathcal{O}(1)$ y la ultima es la operacion desencolar del modulo heap que tiene complejidad $\mathcal{O}(\log(\#nodos(heap)))$. En este caso la cantidad de elementos en el heap corresponde a la cantidad de jugadores esperando atrapar un pokemon, por lo tanto la complejidad de la funcion es $\mathcal{O}(\log(EC))$.

iMove(in e : jugador, in c : coordenada, in/out j : juego)

```

1:  $I \leftarrow j.jugadores[e].posicion$   $\triangleright \Theta(1)$  I de coordenada inicial
2:  $F \leftarrow c$   $\triangleright \Theta(1)$  F de cordenada final
3:  $poke \leftarrow crearIt(j.pokemonsTotales.claves)$   $\triangleright \Theta(1)$ 
4: while  $HaySiguiente(poke)$  do  $\triangleright \mathcal{O}(PS * |P|)$ 
5:    $posPoke \leftarrow CrearIt(Obtener(j.pokemonsTotales, Siguiente(poke)).pos)$   $\triangleright \mathcal{O}(|P|)$ 
6:   while  $HaySiguiente(posPoke)$  do  $\triangleright \mathcal{O}(PS)$ 
7:     if  $SiguienteClave(Siguiente(posPoke)) = I$  then
8:        $LaCordenadaEsInicio(posPoke, I, F, e, j)$   $\triangleright \Theta(1)$ 
9:     end if
10:    if  $SiguienteClave(Siguiente(posPoke)) = F$  then
11:       $LaCoordenadaEsFinal(posPoke, I, F, e, j)$   $\triangleright \log(EC)$ 
12:    end if
13:    if  $SiguienteClave(Siguiente(posPoke)) \neq I \wedge SiguienteClave(Siguiente(posPoke)) \neq F$  then
14:       $LaCoordenadaEsOtra(posPoke, I, F, e, j)$   $\triangleright \mathcal{O}(|P|)$ 
15:    end if
16:  end while
17:   $Avanzar(poke)$   $\triangleright \Theta(1)$ 
18: end while
19:  $ElminiarSiguiente(j.jugadores[e].posMatriz)$   $\triangleright \Theta(1)$ 
20:  $j.jugadores[e].posMatriz \leftarrow AgregarRapido(j.matrizJugadores[c.longitud][c.latitud], e)$   $\triangleright copy(e) = \Theta(1)$  ya que e es un nat.
21:  $j.jugadores[e].posicion \leftarrow c$   $\triangleright \Theta(1)$ 
22: if  $DistEuclidia(I, F) \geq 10 \vee_L !HayCamino(I, F, mapa(j))$  then  $\triangleright \Theta(1) + \Theta(1) = \Theta(1)$ 
23:    $j.jugadores[e].sanciones \leftarrow j.jugadores[e].sanciones + 1$   $\triangleright \Theta(1)$ 
24: end if
25: if  $j.jugadores[e].sanciones \geq 5$  then  $\triangleright \Theta(1)$ 
26:    $ExpulsarJugador(e)$   $\triangleright \mathcal{O}(\log(EC)) + \mathcal{O}(PC * |P|)$ 
27: end if

```

Complejidad: $\mathcal{O}(\log(EC)) + \mathcal{O}(|P| * (PS + PC)) = \mathcal{O}(|P| * (PS + PC) + \log(EC))$, donde EC es la máxima cantidad de jugadores esperando para atrapar un pokémon, PC es la máxima cantidad de pokémon capturados por un jugador, $|P|$ es el nombre más largo para un pokémon en el juego, y PS es la cantidad de pokemones no capturados en juego. Tomo la variable PS , ya que tendré que recorrer efectivamente todos los pokemones salvajes.

Justificación: El algoritmo llama a funciones $\Theta(1)$, $\mathcal{O}(\log(EC))$, $\mathcal{O}(|P| * (PS + PC))$, $\mathcal{O}(PC)$, $\mathcal{O}(PS)$, y $\mathcal{O}(|P|)$. Por álgebra de ordenes, sé que cualquiera de las complejidades nombradas anteriormente sumadas a $\Theta(1)$, permanecen igual (se podría decir que $\Theta(1)$ es neutro aditivo). El algoritmo tiene dos *while*, anidados, que paso a analizar:

- 28: El primer *while* se ejecuta la cantidad de claves que haya en $j.pokemonsTotales$, a su vez, por cada vez que se ejecuta, se ejecuta el segundo *while* como tantas "coordenadas" (son iteradores a) haya en su conjunto de pos (es decir $\#Obtener(j.pokemonsTotales, Siguiente(poke)).pos$). Cada "coordenada" de este conjunto representa un pokemon no capturado, por lo tanto, tengo que por cada tipo de pokemon, veo todos los de ese tipo no capturados, esto es, PS ejecuciones en total.
- 29: Dentro del *while* interno tengo tres llamados a auxiliares ($LaCoordenadaEsInicio$, $LaCoordenadaEsFinal$, $LaCoordenadaEsOtra$). En el recorrido de las coordenadas, entrará en los primeros dos auxiliares una unica vez cada uno, ya que el iterador $posPoke$ no apuntará dos veces a la misma coordenada. El resto de veces, entrara en el tercer auxiliar, que tiene como complejidad $\mathcal{O}(|P|)$. Por lo tanto, se puede decir que en todas las ejecuciones del ciclo anidado, menos 2, ejecutara una función de complejidad $\mathcal{O}(|P|)$, luego esto nos deja una complejidad de: $\mathcal{O}(PS * |P|)$.
- 30: Luego, tengo $\mathcal{O}(\log(EC)) + \mathcal{O}(PC * |P|) + \mathcal{O}(PS * |P|)$. Afirmo que, por álgebra de ordenes: $\mathcal{O}(PC * |P|) + \mathcal{O}(PS * |P|) = \mathcal{O}(|P| * (PS + PC))$. Esto es, por definición, existe un c real tal que: $PC * |P| + PS * |P| \leq c * |P| * (PS + PC)$, y este c puede ser, por ejemplo, 1.
- 31: Entonces, la complejidad de este algoritmo es: $\mathcal{O}(\log(EC)) + \mathcal{O}(|P| * (PS + PC))$.
-

LaCordenadaEsInicio(in *posPoke*: it conj(it Dicc(coord,infoCoord)), in *I*: coord, in *F*: coord, in *e*: jugador in/out *j*: juego)

```

1: k ← SiguienteClave(Siguiente(posPoke))                                ▷  $\Theta(1)$ 
2: if HayPokemonCercano(k, j) then                                       ▷  $\Theta(1)$ 
3:   if  $\neg$ HayPokemonCercano(F, j)  $\vee_L$  PosPokemonCercano(k, j)  $\neq$  PosPokemonCercano(F, j) then ▷  $\Theta(1) +$ 
    $\Theta(1) + \Theta(1) = \Theta(1)$ 
4:     SiguienteSignificado(Siguiente(posPoke)).cantMovCapt ← 0      ▷  $\Theta(1)$  No toco la cola de prioridad,
   porque la toco cuando k=final
5:   end if
6: end if
7: Avanzar(posPoke)                                                    ▷  $\Theta(1)$ 

```

Complejidad: $\Theta(1)$

Justificación: El algoritmo sólo llama a funciones $\Theta(1)$, y no tiene ningun ciclo.

LaCoordenadaEsFinal(in *posPoke*: it conj(it Dicc(coord,infoCoord)), in *I*: coord, in *F*: coord, in *e*: jugador, in/out *j*: juego)

```

1: k ← SiguienteClave(Siguiente(posPoke))                                ▷  $\Theta(1)$ 
2: if HayPokemonCercano(k, j) then                                       ▷  $\Theta(1)$ 
3:   if HayPokemonCercano(I, j)  $\wedge_L$  PosPokemonCercano(k, j)  $\neq$  PosPokemonCercano(I, j) then ▷  $\Theta(1) +$ 
    $\Theta(1) + \Theta(1) = \Theta(1)$ 
4:     SiguienteSignificado(Siguiente(posPoke)).cantMovCap ← 0      ▷  $\Theta(1)$ 
5:     jugadorHeap ←  $\langle$ cantidadPokemons(e, j), e $\rangle$                     ▷  $\Theta(1)$ 
6:     heap ← SiguienteSignificado(Siguiente(posPoke)).colaPrioridad    ▷  $\Theta(1)$ 
7:     EliminarSiguiente(j.jugadores[e].prioridad)                    ▷  $\mathcal{O}(\log(EC))$ 
8:     j.jugadores[e].prioridad ← Encolar(jugadorHeap, heap)          ▷  $\mathcal{O}(\log(EC))$ 
9:   end if
10:  if  $\neg$ HayPokemonCercano(I, j) then                                       ▷  $\Theta(1)$ 
11:    SiguienteSignificado(Siguiente(posPoke)).cantMovCap ← 0      ▷  $\Theta(1)$ 
12:    jugadorHeap ←  $\langle$ cantidadPokemons(e, j), e $\rangle$                     ▷  $\Theta(1)$ 
13:    heap ← SiguienteSignificado(Siguiente(posPoke)).colaPrioridad    ▷  $\Theta(1)$ 
14:    j.jugadores[e].prioridad ← Encolar(jugadorHeap, heap)          ▷  $\mathcal{O}(\log(EC))$ 
15:  end if
16: end if
17: Avanzar(posPoke)

```

Complejidad: $\mathcal{O}(\log(EC))$, donde *EC* es la máxima cantidad de jugadores esperando para atrapar un pokémon. Tomo esta variable, ya que el peor caso será si la cola de prioridad del pokemon que está situado en rango con la coordenada final (teniendo en cuenta que haya pokemon), tenga la máxima cantidad de jugadores esperando.

Justificación: El algoritmo llama a funciones $\Theta(1)$ y $\log(EC)$. No tiene ningún ciclo, así que resta sumar todas las complejidades: por propiedades de álgebra de ordenes sé que: $\log(EC) + \Theta(1) = \log(EC)$ y $\log(EC) + \log(EC) = \log(EC)$. Luego, la complejidad es $\log(EC)$.

LaCoordenadaEsOtra(in *posPoke*: it conj(it Dicc(coord,infoCoord)), in *I*: coord, in *F*: coord, in/out *j*: juego)

```

1: k ← SiguienteClave(Siguiente(posPoke))                                ▷  $\Theta(1)$ 
2: if HayPokemonCercano(k, j) ∧ SiguienteSignificado(posPoke).cantMovPCapt = 9 then    ▷  $\Theta(1) + \Theta(1) = \Theta(1)$ 
3:   CapturarPokemon(posPoke, j)                                         ▷  $\mathcal{O}(|P|)$ 
4: else
5:   if HayPokemonCercano(k, j) ∧ SiguienteSignificado(posPoke).cantMovPCapt < 9 then
6:     SiguienteSignificado(posPoke).cantMovPCapt ← SiguienteSignificado(posPoke).cantMovPCapt + 1
7:   end if
8:   Avanzar(posPoke)                                                    ▷  $\Theta(1)$ 
9: end if

```

Complejidad: $\mathcal{O}(|P|)$, donde $|P|$ es el nombre más largo para un pokémon en el juego. Tomo esta variable ya que, el peor caso será si el pokemon a capturar es el que tiene el nombre mas largo.

Justificación: El algoritmo llama a funciones $\Theta(1)$ y $\mathcal{O}(|P|)$, y no entra a ningún ciclo. Luego, por álgebra de ordenes sé que: $\mathcal{O}(|P|) + \Theta(1) = \mathcal{O}(|P|)$.

CapturarPokemon(in *poke*: it conj(it Dicc(coord,infoCoord)), in/out *j*: juego)

```

1: k ← SiguienteClave(Siguiente(poke))                                ▷  $\Theta(1)$ 
2: posibles ← SiguienteSignificado(Siguiente(poke)).colaPrioridad    ▷  $\Theta(1)$ 
3: tipo ← SiguienteSignificado(Siguiente(poke)).tipo                ▷  $\Theta(1)$ 
4: if HaySiguiente(posibles) then                                       ▷  $\Theta(1)$ 
5:   jugGanador ← Proximo(posibles).id                                  ▷  $\Theta(1)$ 
6:   DarlePokemon(jugGanador, tipo, j)                                ▷  $\mathcal{O}(|P|)$ 
7:   j.matrizPokemons[k.longitud][k.latitud] ←  $\langle false, NULL \rangle$     ▷  $\Theta(1)$ 
8:   EliminarSiguiente(Siguiente(poke))                             ▷  $\Theta(1)$  Eliminamos la coordenada del diccionario de coords
9:   EliminarSiguiente(poke)                                           ▷  $\Theta(1)$  Eliminamos el it dentro del trie de pokes
10: end if

```

Complejidad: $\mathcal{O}(|P|)$, donde $|P|$ es el nombre más largo para un pokémon en el juego.

Justificación: El algoritmo llama a funciones $\Theta(1)$ y $\mathcal{O}(|P|)$, y no entra a ningún ciclo. Luego, por álgebra de ordenes sé que: $\mathcal{O}(|P|) + \Theta(1) = \mathcal{O}(|P|)$.

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

```

1: iter ← NULL
2: if Definido?(j.jugadores[e].pokesRapido, p) then                    ▷  $\mathcal{O}(|P|)$ 
3:   iter ← Obtener(j.jugadores[e].pokesRapido, p)                     ▷  $\mathcal{O}(|P|)$ 
4:   Siguiente(iter).cant ← Siguiente(iter).cant + 1                 ▷  $\Theta(1)$ 
5: else
6:   nuevoPokemon ←  $\langle p, 1 \rangle$                                           ▷  $\Theta(1)$ 
7:   iter ← AgregarRapido(j.jugadores[e].pokemons, nuevoPokemon)    ▷ copy(nuevoPokemon) =  $\Theta(|P|)$ , ya que nuevoPokemon es un string, y será, como mucho, el nombre más largo para un pokémon en el juego.
8:   Definir(j.jugadores[e].pokesRapido, p, iter)                     ▷  $\mathcal{O}(|P|)$ 
9: end if

```

Complejidad: $\mathcal{O}(|P|)$, donde $|P|$ es el nombre más largo para un pokémon en el juego.

Justificación: El algoritmo llama a funciones $\Theta(1)$ y $\mathcal{O}(|P|)$, y no entra a ningún ciclo. Luego, por álgebra de ordenes sé que: $\mathcal{O}(|P|) + \Theta(1) = \mathcal{O}(|P|)$.

ExpulsarJugador(in e: jugador, in/out j: juego)

```

1:  $j.cantidadPokeTot \leftarrow j.cantidadPokeTotal - CantidadPokemons(e, j)$   $\triangleright \mathcal{O}(PC)$ 
2:  $EliminarPokemons(e, j)$   $\triangleright \mathcal{O}(PC * |P|)$ 
3:  $EliminarSiguiente(j.jugadores[e].posMatriz)$   $\triangleright \Theta(1)$ 
4: if  $j.jugadores[e].prioridad \neq NULL$  then  $\triangleright \Theta(1)$ 
5:    $EliminarSiguiente(j.jugadores[e].prioridad)$   $\triangleright \mathcal{O}(\log(EC))$ 
6: end if
7:  $j.jugadores[e].conectado \leftarrow false$   $\triangleright \Theta(1)$ 
8:  $expulsarJ \leftarrow j.jugadores[e].lugarNoExpulsado$   $\triangleright \Theta(1)$ 
9:  $j.jugadores[e].lugarNoExpulsado \leftarrow NULL$   $\triangleright \Theta(1)$ 
10:  $EliminarSiguiente(expulsarJ)$   $\triangleright \Theta(1)$ 
11:  $AgregarRapido(j.Expulsados, e)$   $\triangleright copy(e) = \Theta(1)$  ya que e es un nat.

```

Complejidad: $\mathcal{O}(\log(EC)) + \mathcal{O}(PC * |P|)$, donde EC es la máxima cantidad de jugadores esperando para atrapar un pokémon, PC es la máxima cantidad de pokémon capturados por un jugador, y $|P|$ es el nombre más largo para un pokémon en el juego.

Justificación: El algoritmo llama a funciones $\Theta(1)$, $\mathcal{O}(\log(EC))$, y $\mathcal{O}(PC * |P|)$. No tiene ningún ciclo, así que resta sumar todas las complejidades: por propiedades de álgebra de ordenes sé que: $\log(EC) + \Theta(1) = \log(EC)$, $\log(EC) + \log(EC) = \log(EC)$, $\Theta(1) + \mathcal{O}(|P|) = \mathcal{O}(|P|)$, $\Theta(1) + \mathcal{O}(PC) = \mathcal{O}(PC)$, y $\mathcal{O}(PC * |P|) + \mathcal{O}(|P|) = \mathcal{O}(PC * |P|)$. Luego, la complejidad debe ser $\mathcal{O}(\log(EC)) + \mathcal{O}(PC * |P|)$.

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

```

1:  $iter \leftarrow pokemons(e, j)$   $\triangleright \Theta(1)$ 
2: while  $HaySiguiente(iter)$  do  $\triangleright \mathcal{O}(PC)$ 
3:    $res \leftarrow res + Siguiente(iter).cant$   $\triangleright \Theta(1)$ 
4:    $Avanzar(iter)$   $\triangleright \Theta(1)$ 
5: end while

```

Complejidad: $\mathcal{O}(PC)$, donde PC es la máxima cantidad de pokémon capturados por un jugador. Tomo esta variable ya que, en el peor caso, el jugador a expulsar será el que mas pokemones tenga (antes de expulsarlo).

Justificación: El algoritmo llama a funciones $\Theta(1)$ y $\mathcal{O}(PC)$, y no entra a ningún ciclo. Luego, por álgebra de ordenes sé que: $\Theta(1) + \mathcal{O}(PC) = \mathcal{O}(PC)$.

EliminarPokemons(in e: jugador, in/out j: juego)

```

1:  $iter \leftarrow pokemons(e, j)$   $\triangleright \Theta(1)$  Iteramos sobre los pokemons del Jugador.
2: while  $HaySiguiente(iter)$  do  $\triangleright \mathcal{O}(PC * |P|)$ 
3:    $pokemon \leftarrow Siguiente(iter).tipo$   $\triangleright \Theta(1)$  Tipo pokemon a eliminar.
4:    $Obtener(pokemonsTotales, pokemon).cant \leftarrow Obtener(pokemonsTotales, pokemon).cant - Siguiente(iter).cant$   $\triangleright \mathcal{O}(|P|)$ 
5:   if  $Obtener(pokemonsTotales, pokemon).cant = 0$  then  $\triangleright \mathcal{O}(|P|)$  Si todos los pokemons de ese tipo los tenia el jugador.
6:      $claveAEliminar \leftarrow pokemon$   $\triangleright \Theta(1)$ 
7:      $Borrar(j.pokemonsTotales, pokemon)$   $\triangleright \mathcal{O}(|P|)$  Borramos al pokemon del Trie
8:      $EliminarSiguiente(iter)$   $\triangleright \Theta(1)$  Borramos al pokemon del conjunto de pokemons del jugador.
9:   end if
10: end while

```

Complejidad: $\mathcal{O}(PC * |P|)$, donde PC es la máxima cantidad de pokémon capturados por un jugador, y $|P|$ es el nombre más largo para un pokémon en el juego.

Justificación: El algoritmo llama a funciones $\Theta(1)$, funciones $\mathcal{O}(PC)$ y $\mathcal{O}(|P|)$. Tiene un ciclo que ejecutara como mucho $\mathcal{O}(PC)$, donde llama funciones de $\Theta(1)$ y una función de $\mathcal{O}(|P|)$. Entonces, este ciclo costara, como mucho, $\mathcal{O}(PC * |P|)$. Por álgebra de ordenes sé: $\Theta(1) + \mathcal{O}(|P|) = \mathcal{O}(|P|)$, $\Theta(1) + \mathcal{O}(PC) = \mathcal{O}(PC)$, y $\mathcal{O}(PC * |P|) + \mathcal{O}(|P|) = \mathcal{O}(PC * |P|)$. Luego, su complejidad debe ser $\mathcal{O}(PC * |P|)$.

iPuedoAgregarPokemon(in c : coord, in j : juego) $\rightarrow res$:bool

```

1: if  $posExistente(c, m)$  then
2:    $res \leftarrow true$   $\triangleright \mathcal{O}(1)$ 
3:    $posibles \leftarrow coordARadio(c, j, 25)$   $\triangleright$  Da un it de conjunto de coord a radio 5  $\mathcal{O}(1)$ 
4:   while  $haySiguiente(posibles)$  do
5:     if  $j.matrizPokemon[longitud(Siguiente(posibles))][latitud(siguiente(posibles))].hayPoke$  then
6:        $res \leftarrow false$ 
7:     end if
8:      $Avanzar(it)$ 
9:   end while
10: else  $res \leftarrow false$ 
11: end if

```

Complejidad: Se recorre un conjunto acotado de coordenadas (las coordenadas que se encuentran a un radio menor que 5 de la pasada por parametro) y para cada una de ellas solo se hace un acceso a un vector de vectores y la complejidad de esto ultimo es $\mathcal{O}(1)$ por lo cual la complejidad del algoritmo es $\mathcal{O}(1)$

iHayPokemonCercano(in c : coord, in j : juego) $\rightarrow res$:bool

```

1:  $res \leftarrow false$   $\triangleright \mathcal{O}(1)$ 
2:  $posibles \leftarrow coordARadio(c, j, 4)$   $\triangleright$  Da un it de conjunto de coord a radio 2  $\mathcal{O}(1)$ 
3: while  $haySiguiente(posibles)$  do
4:   if  $j.matrizPokemon[longitud(Siguiente(posibles))][latitud(siguiente(posibles))].hayPoke$  then
5:      $res \leftarrow true$ 
6:   end if
7:    $Avanzar(posibles)$ 
8: end while

```

Complejidad: Se recorre un conjunto acotado de coordenadas (las coordenadas que se encuentran a un radio menor o igual que 2 de la pasada por parametro) y para cada una de ellas solo se hace un acceso a un vector de vectores y la complejidad de esto ultimo es $\mathcal{O}(1)$ por lo cual la complejidad del algoritmo es $\mathcal{O}(1)$

iPosPokemonCercano(in c : coord, in j : juego) $\rightarrow res$:pokemon

```

1:  $res \leftarrow false$   $\triangleright \mathcal{O}(1)$ 
2:  $posibles \leftarrow coordARadio(c, j, 4)$   $\triangleright$  Da un it de conjunto de coord a radio 2  $\mathcal{O}(1)$ 
3: while  $haySiguiente(posibles)$  do
4:    $cor \leftarrow j.matrizPokemon[longitud(Siguiente(posibles))][latitud(siguiente(posibles))]$ 
5:   if  $cor.hayPoke$  then
6:      $res \leftarrow SiguienteSignificado(cor.iterador).tipo$ 
7:   end if
8:    $Avanzar(posibles)$ 
9: end while

```

Complejidad: Se recorre un conjunto acotado de coordenadas (las coordenadas que se encuentran a un radio menor o igual que 2 de la pasada por parametro) y para cada una de ellas solo se hace un acceso a un vector de vectores y la complejidad de esto ultimo es $\mathcal{O}(1)$ por lo cual la complejidad del algoritmo es $\mathcal{O}(1)$

iEntrenadoresPosibles(in c : coord, in j : juego) $\rightarrow res:it$ Conj(jugadores)
 $\triangleright \mathcal{O}(1)$

```

1:  $entrenadres \leftarrow vacio()$ 
2:  $posibles \leftarrow coordARadio(c, j, 4)$ 
3: while  $haySiguiente(posibles)$  do
4:    $it \leftarrow CrearIt(j.matriz.Jugadores[longitud(Siguiente(posibles))][latitud(siguiente(posibles))])$ 
5:   while  $haySiguiente(it)$  do
6:      $AgregarRapido(entrenadores, Siguiente(it))$ 
7:      $Avanzar(it)$ 
8:   end while
9:    $Avanzar(posibles)$ 
10: end while
11:  $res \leftarrow CrearIt(entrenadores)$ 

```

Complejidad: $\mathcal{O}(|J'|)$ donde J' es la cantidad total de jugadores no eliminados

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

```

1:  $res \leftarrow 100 - (100 * cantMismaEspecie(p, j) / j.cantPokemonsTotales)$ 

```

Complejidad: $\mathcal{O}(|P|)$ donde P es el nombre del Pokemon mas Largo

iCantMismaEspecie(in p : pokemon, in/out j : juego) $\rightarrow res:nat$

```

1:  $it \leftarrow j.pokemonsTotales$ 
2: while  $haySiguiente(it)$  do
3:   if  $SiguienteClave(it) = p$  then
4:      $res \leftarrow SiguienteSignificado(it).cant$ 
5:   end if
6:    $Avanzar(it)$ 
7: end while

```

Complejidad: $\mathcal{O}(|P|)$ donde P es el nombre del Pokemon mas Largo

iCantPokemonsTotales(in p : pokemon, in/out j : juego) $\rightarrow res:nat$

```

1:  $res \leftarrow j.cantidadPokeTotal$ 

```

Complejidad: $\mathcal{O}(1)$

iCoordARadio(**in** c : coord, **in** j : juego, **in** r : nat) $\rightarrow res$: it Conj(coord)
 $\triangleright \mathcal{O}(1)$

```

1:  $coordenadas \leftarrow vacio()$ 
2: if  $r \leq Latitud(c)$  then
3:    $i \leftarrow Latitud(c) - r$ 
4: else  $i \leftarrow 0$ 
5: end if
6: if  $r \leq Longitud(c)$  then
7:    $j \leftarrow Longitud(c) - r$ 
8: else  $j \leftarrow 0$ 
9: end if
10: while  $i \leq Latitud(c) + r$  do
11:   while  $j \leq Longitud(c) + r$  do
12:     if  $posExistente(crearCoord(i, j), j.mapa)$  then
13:        $AgregarRapido(coordenas, crearCoord(i, j))$ 
14:     end if
15:      $j \leftarrow j + 1$ 
16:   end while
17:    $i \leftarrow i + 1$ 
18: end while
19:  $res \leftarrow CrearIt(coordenas)$ 

```

Complejidad: $\mathcal{O}(|J'|)$ donde J' es la cantidad total de jugadores no eliminados
