# Algoritmos y Estructuras de Datos II

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

## Trabajo Práctico Nº2

**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 | 186/15 | juanmanuelbaldonado@gmail.com |
| Carbonelli, Lucas | 596/15 | luc92a@hotmail.com |



**Reservado para la cátedra**

| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |

# Índice

# 1. Módulo Coordenada

## Interfaz

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

**se explica con**: COORDENADA.

**generos**: `coordenada`.

### Operaciones básicas de Coordenada

CREARCOOR(**in** $n$: `Nat`, **in** $n'$: `Nat`) $\rightarrow$ $res$ : `Coordenada`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$longitud(res) = n \wedge latitud(res) = n'$ }
**Complejidad:** $\Theta(1)$
**Descripción:** genera una coordenada con longitud y latitud pasadas por parametro.

LONGITUD(**in** $c$: `Coordenada`) $\rightarrow$ $res$ : `Nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res = longitud(c)$}
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve el valor correspondiente a la longitud de la coordenada.

LATITUD(**in** $c$: `Coordenada`) $\rightarrow$ $res$ : `Nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res = latitud(c)$}
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve el valor correspondiente a la latitud de la coordenada.

DISTEUCLIDEA(**in** $c$: `Coordenada`, **in** $c'$: `Coordenada`) $\rightarrow$ $res$ : `Nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res = (latitud(c) - latitud(c')) * (latitud(c) - latitud(c')) + (longitud(c) - longitud(c')) * (longitud(c) - longitud(c'))$)}
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve la distancia euclidiana.

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

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

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

COORDENADAALAIZQUIERDA(**in** $c$: `Coordenada`) $\rightarrow$ $res$ : `Coordenada`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{latitud(res) = latitud(c) \wedge longitud(res) = longitud(c) - 1\}$
**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`)

Rep : coord $\longrightarrow$ bool
Rep($c$) $\equiv$ true $\Longleftrightarrow$

0
Abs : coord $c$ $\longrightarrow$ coordenada(Nat) {Rep($c$)}
Abs($c$) $\equiv$ crearCoord(c.latitud , c.longitud)

# Algoritmos

## Algoritmos del modulo

---

**iCrearCoord**(**in** $n$: `Nat`, **in** $n'$: `Nat`) $\rightarrow$ $res$ : Coordenada
1: $res \leftarrow \langle n, n' \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iLatitud**(**in** $c$: `coord`) $\rightarrow$ $res$ : $Nat$
$res \leftarrow c.Latitud$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iLongitud**(**in** $c$: `coord`) $\rightarrow$ $res$ : $Nat$
$res \leftarrow c.Longitud$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iDistEuclidea**(**in** $c$: `coord`,**in** $c'$: `coord`) $\rightarrow$ $res$ : $Nat$

**if** $c.Longitud > c'.Longitud$ **then** ▷ $\Theta(1)$
  $long \leftarrow (c.Longitud - c'.Longitud) * (c.Longitud - c'.Longitud)$ ▷ $\Theta(1)$

**else** $long \leftarrow (c'.Longitud - c.Longitud) * (c'.Longitud - c.Longitud)$ ▷ $\Theta(1)$
**end if**

**if** $c.Latitud > c'.Latitud$ **then** ▷ $\Theta(1)$
  $lat \leftarrow (c.Latitud - c'.Latitud) * (c.Latitud - c'.Latitud)$ ▷ $\Theta(1)$

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

Complejidad: $\Theta(1)$

---

**iCoordenadaArriba**(**in** $c$: `coord`) $\rightarrow$ $res$ : $coord$
$res \leftarrow \langle Latitud(c) + 1, Longitud(c) \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iCoordenadaAbajo**(**in** $c$: `coord`) $\rightarrow$ $res$ : $coord$
$res \leftarrow \langle Latitud(c) - 1, Longitud(c) \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iCoordenadaALaDerecha**(**in** $c$: `coord`) $\rightarrow$ $res$ : $coord$
$res \leftarrow \langle Latitud(c), Longitud(c) + 1 \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iCoordenadaALaIzquierda**(**in** $c$: `coord`) $\rightarrow$ $res$ : $coord$
$res \leftarrow \langle Latitud(c), Longitud(c) - 1 \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

# 2. Módulo Mapa

## Interfaz

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

**se explica con**: MAPA.

**generos**: `map`.

### Operaciones basicas de Mapa

CREARMAPA() $\rightarrow$ $res$ : `map`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$(\not\exists hayAnterior(coordenadas(res))) \wedge (\not\exists haySiguiente(coordenadas(res)))$ }
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve una instancia de mapa sin coordenadas

AGREGARCOOR(**in** $c$: `coord`,**in/out** $m$: `map` ) $\rightarrow$ $res$ : `map`
**Pre** $\equiv$ {$m =_{\text{obs}} m_0$}
**Post** $\equiv$ {$secuSuby(coordenadas(res)) =_{\text{obs}} c \bullet secuSuby(coordenadas(m_0))$}
**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** $m$: `map`) $\rightarrow$ $res$ : `itConj(coordenada)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} CrearIt(m)$}
**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** $c$: `coord`,**in/out** $m$: `map` ) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} esta?(c, secuSuby(coordenadas(m)))$}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve true si hay la coordenada se encuentra en el mapa y false de lo contrario.

HAYCAMINO(**in** $c$: `coord`,**in** $c'$: `coord`,**in** $m$: `map`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {$esta?(c, secuSuby(coordenadas(m))) \wedge esta?(c', secuSuby(coordenadas(m)))$}
**Post** $\equiv$ {$res =_{\text{obs}} hayCamino(c, c', m)$}
**Complejidad:** $\Theta(??)$
**Descripción:** Devuelve true si hay un camino entre las 2 corrdenadas.

EXISTECAMINO(**in** $c$: `coord`,**in** $c'$: `coord`,**in** $cs$: `conj(coord)`,**in** $m$: `map`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {$esta?(c, secuSuby(coordenadas(m))) \wedge esta?(c', secuSuby(coordenadas(m))) \wedge cs \subseteq coordenadas(m)$}
**Post** $\equiv$ {$res =_{\text{obs}} existeCamino(c, c', cs, m)$}
**Complejidad:** $\Theta(??)$

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

# Representación

## Representacion del Mapa

El modulo se representa como un vector de vectores .El vector externo coresponde a la latitud y los vectores dentro de este a la longitud. Detro de cada posicion del vector ïnterno"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 euclideana 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 euclideana es menor que 100 en tiempo constante , para el resto de los casos la complejidada 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 $\longrightarrow$ bool
Rep($m$) $\equiv$ true $\iff$ $vacio(m.coordenas) \iff longitud(m.columnas) = 0 \wedge \neg vacio(m.coordenas) \Rightarrow_{\text{L}}$ $((\forall c : coord)\ pertenece(m.Coordenadas, c) \iff (Longitud(c) \leq Longitud(m.columnas)) \wedge_{\text{L}} (Latitud(c) \leq Longitud(m.columnas[Longitud(c)])) \wedge_{\text{L}} m.columnas[Longitud(c)][Latitud(c)].esta = false) \wedge (\forall c' : coord) pertenece(m.Coordenadas, c') \wedge distEuclidea(c, c') < 10 \Rightarrow_{\text{L}} pertenece(m.coordenadas[Longitud(c)][Latitud(c)].adyacentes, c') \wedge pertenece(m.coordenadas[Longitud(c')][Latitud(c')].adyacentes, c)$
maximo : $conj(coord)\ cs$ $\longrightarrow$ coord
dameMapa(cs) $\equiv$ **if** $vacia(s)$ **then** $m$ **else** dameMapa($fin(s)$,$AgregarColumna(prim(s), m, 0)$ ,$i + 1$) **fi**

Abs : map $m$ $\longrightarrow$ Mapa {Rep($m$)}
Abs($m$) $\equiv$ $dameMapa(abs(abs(m.filas))$ , $crearMapa()$,0 )
dameMapa : $secu(secu(bool))\ s \times map\ m \times nat\ i$ $\longrightarrow$ 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 $j$ $\longrightarrow$ 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: $res \leftarrow \langle iVacia(), 0 \rangle$ ▷ $\mathcal{O}(1)$

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

---

**icoordenadas**(**in** $m$ : `map`) $\rightarrow$ $res$ : it Conj(coordendas)

1: $res \leftarrow \langle CrearIt(m.coordenadas) \rangle$ ▷ creo un iterador a conjunto de coordenadas$\mathcal{O}(1)$

Complejidad: $\Theta(1)$

**posExistente**(**in** $c$: `coord`,**in** $m$ : `map`) $\rightarrow$ $res$ : bool

1:
2: **if** Longitud(c) >m.longitud **then** ▷ $\mathcal{O}(1)$
3:
4: **if** Latitud(c) >(m.columnas[c.Longitud]).longitud **then** ▷ $\mathcal{O}(1)$
5: $res \leftarrow (m.columnas[Longitud(c)])[Latitud(c)].esta$ ▷ acceso a un arreglo de arreglos $\mathcal{O}(1)$
6: **end if**
7: **else**$res \leftarrow false$
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** $c$: `coord`,**in/out** $m$ : `map`) $\rightarrow$ $res$ : map

1:
2: **if** longitud(m.columnas) <Longitud(c) **then**
3: $i \leftarrow longitud(m.columnas)$ ▷ $\Theta(1)$
4: **while** $i \leq Longitud(c)$ **do** ▷ $\mathcal{O}(Longitud(c))$
5: $Agregar(m.columnas, i, iVacia())$ ▷ se agrega en la posicion i-esima del vector un arreglo vacio $\mathcal{O}(longitud(m.columnas))$
6: $i \leftarrow i + 1$ ▷ $\mathcal{O}(1)$
7: **end while**
8: **end if**
9:
10: **if** longitud(m.columnas[Longitud(c)]) <Latitud(c) **then**
11: $j \leftarrow longitud(m.columnas[Longitud(c)])$ ▷ acceso a un arreglo $\mathcal{O}(1)$
12: **while** $j \leq Latitud(c)$ **do** ▷ $\mathcal{O}(Latitud(c))$
13: $Agregar(m.columnas[Longitud(c)], j, < FALSE, iVacio())$ ▷ se agrega en la posicion j-esima del vector una tupla<bool, arreglo vacio>$\mathcal{O}(1)$
14: $j \leftarrow j + 1$ ▷ $\mathcal{O}(Latitud(c))$
15: **end while**
16: **end if**
17: $Agregar(m.columnas[Longitud(c)], Latitud(c), < True, iVacio() >)$ ▷ $\Theta(1)$
18: $it \leftarrow coordenadas(m)$ ▷ $\mathcal{O}(1)$
19: **while** haySiguiente(it) **do**
20:
21: **if** $HayCamino(c, Siguiente(it), m) \wedge (c \neq Siguiente(it)) \wedge distanciaEuclideana(c, Siguiente(it)) < 100$ **then** ▷ 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: $Agregar(m.columnas[longitud(Siguiente(it))][Latitud(Siguiente(it))].adyacentes, c)$ ▷ $\mathcal{O}(1)$
23: $Agregar(m.columnas[Longitud(c)][Latitud(c)].adyacentes, Siguiente(it)$ ▷ $\mathcal{O}(1)$
24: **end if**
25: $Avanzar(it)$ ▷ $\mathcal{O}(1)$
26: **end while**

Complejidad: La funcion realiza el el peor caso 2*Latitud(c)+2*Longitud(c) inserciones a un vector pues puede suceder que tanto la latidud 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 $\#(m.coordenadas)$. Tenemos entonces que la complejidad de nuestro algotirmo es$\mathcal{O}(Latitud(c) + Longitud(c) + n^2)$.

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

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

Complejidad: En el caso en el que la distancia euclidea entre las dos coordenanas pasadas por paramentro 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 euclideana sea mayor a 100 la funcion debe llamar a la funcion auxiliar hayCamino2 cuya complejidad es $\mathcal{O}(4^n n^2) dondenes \#(m.coordenadas)$. 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: $res \leftarrow iExisteCamino(c, c', Eliminar(coords, c), m)$ ▷ $\mathcal{O}(4^{\#Eliminar(coords,c)}(\#Eliminar(coords, c))^2)$

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

**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** $res \leftarrow iExisteCaminoPorArriba(c, c', cs, m)$ or $iExisteCaminoPorAbajo(c, c', cs, m)$ or $iExisteCaminoPorDerecha(c, c', cs, m)$ 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 el la ecuacion de recurrencia es la siguiente

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

Luego obtenemos que
$\mathcal{T}(n) = \sum_{i=1}^{n-1}(4^i + \mathcal{O}(n)) \leq 4^n \mathcal{O}(n^2) = \mathcal{O}(4^n n^2)$ donde n 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 ← iExisteCamino(iCoordenadaArriba(c), c', Remover(cs, iCoordenadaArriba(c)), m)
4: elseres ← 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 ← iExisteCamino(iCoordenadaAbajo(c), c', Remover(cs, iCoordenadaAbajo(c)), m)
6:     elseres ← false
7:     end if
8: elseres ← 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 ← iExisteCamino(iCoordenadaALaDerecha(c), c', Remover(cs, iCoordenadaALaDerecha(c)), m)
4: elseres ← 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 ← iExisteCamino(iCoordenadaALaIzquierda(c), c', Remover(cs, iCoordenadaALaIzquierda(c)), m)
6:     elseres ← false
7:     end if
8: elseres ← false
9: end if
```

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

# 3. Renombres de TADs

TAD JugadorHeap es Tupla(Nat,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 $\alpha$ que tenga una relación de orden total estricto. Esta implementación se limita al tipo jugadorHeap, que es tupla <Nat, Nat>.

## Interfaz

**se explica con**: Cola de prioridad, jugadorHeap.

**géneros**: `heapMod`.

### Operaciones básicas de Heap Modificable

Vacía?(**in** $c$: `heapMod`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{\text{obs}}$ vacía?(c)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve true si la cola de prioridad está vacia.

Próximo(**in** $c$: `heapMod`) $\rightarrow$ $res$ : `JugadorHeap`
**Pre** $\equiv$ {¬vacía?(c)}
**Post** $\equiv$ {alias($res$, 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** $c$: `heapMod`)
**Pre** $\equiv$ {¬vacía?(c) $\wedge$ $c =_{\text{obs}} c_0$}
**Post** $\equiv$ {c $=_{\text{obs}}$ desencolar($c_0$)}
**Complejidad:** $\Theta(log(\#heapSecu(c)))$
**Descripción:** Desencola el próximo elemento.

Vacía() $\rightarrow$ $res$ : `heapMod`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ vacía()}
**Complejidad:** $\Theta(1)$
**Descripción:** Retorna por referencia una cola vacía.

Encolar(**in** $a$: `JugadorHeap`, **in/out** $c$: `heapMod`) $\rightarrow$ $res$ : `itHeapMod`
**Pre** $\equiv$ {$c =_{\text{obs}} c_0$}
**Post** $\equiv$ {c $=_{\text{obs}}$ encolar($c_0$) $\wedge$ res $=_{\text{obs}}$ AgregarComoSiguiente(CrearIt($c$), $a$)}
**Complejidad:** $\Theta(log(\#heapSecu(c)))$
**Descripción:** Encola el elemento JugadorHeap. Devuelve un iterador de forma tal que al pedir Siguiente se obtenga el elemento agregado.

EsMayor?(**in** $a$: `JugadorHeap`, **in** $b$: `JugadorHeap`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{\text{obs}}$ 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** $\equiv$ {$res =_{\text{obs}}$ 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** $\equiv$ {$res =_{\text{obs}}$ 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($res =_{\text{obs}}$ 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** $\equiv$ {$it =_{\text{obs}}$ 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** $\equiv$ {$it =_{\text{obs}}$ 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 $\longrightarrow$ bool

**otras operaciones (no exportadas)**
heapSecu : heapMod $\longrightarrow$ secu(jugadorHeap)

**axiomas**
esMayor?(a, b) $\equiv$ **if** $\Pi_1(a) = \Pi_1(b)$ **then** $\Pi_2(a) < \Pi_2(b)$ **else** $\Pi_1(a) < \Pi_1(b)$ **fi**
heapSecu(c) $\equiv$ **if** vacía?(c) **then** <> **else** próximo(c) $\bullet$ heapSecu(desencolar(c)) **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 tando 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 rapidamente.

**`heapMod` se representa con `heapmod`**

donde `heapmod` es `tupla`(*tope*: `puntero(nodo)`)
donde `nodo` es `tupla`(*elemento*: `jugadorHeap`, *ramaMasCorta*: `nat`, *ramaMasLarga*: `nat`, *hijoIzq*: `puntero(nodo)`, *hijoDer*: `puntero(nodo)`, *padre*: `puntero(nodo)`)
donde `jugadorHeap` es `tupla`(*cantPokes*: `nat`, *id*: `nat`)

Rep : heapmod $\longrightarrow$ bool
Rep($c$) $\equiv$ 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 $< Nat, Nat >$
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 ($log(\#nodos)$).
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 vacia.

**`itHeapMod` se representa con `itHeapmod`**

donde `itHeapmod` es `tupla`(*siguiente*: `puntero(heapmod)`, *heap*: `puntero(heapmod)`)

Rep : itHeapmod $\longrightarrow$ bool
Rep($it$) $\equiv$ true $\iff 1 \wedge 2$

Invariante de representación:
1) *siguiente* apunta a $NULL$ si i solo si el *heap* apunta a una cola vacia.
2) Si la cola no está vacia, *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: $res \leftarrow c.tope = NULL$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iPróximo**(**in** $c$: `heapmod`) $\rightarrow$ $res$ : jugadorHeap

1: $res \leftarrow *(c.tope).elemento$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

**iDesencolar**(**in/out** $c$: `heapmod`)

1: **if** $*(c.tope).hijoIzq = NULL \wedge *(c.tope).hijoDer = NULL$ **then** ▷ $\Theta(1)$
2: $c.tope \leftarrow NULL$ ▷ Si hay un solo elemento, simplemente se elimina // $\Theta(1)$
3: **else**
4: $puntero(nodo)\ ultimoNodo \leftarrow iUltimoNodo(c)$ ▷ Se obtiene un puntero al último nodo // $\Theta(log(\#nodos(c))$
5: $puntero(nodo)\ padreUlt \leftarrow (*ultimoNodo).padre$ ▷ Se crea un puntero al padre del último nodo // $\Theta(1)$
6: **if** $(*padreUlt).hijoDer = ultimoNodo$ **then** ▷ Vemos si el último nodo es el derecho del padre // $\Theta(1)$
7: $(*padreUlt).hijoDer \leftarrow NULL$ ▷ Eliminamos la conexión al nodo // $\Theta(1)$
8: **else**
9: $(*padreUlt).hijoIzq \leftarrow NULL$ ▷ Eliminamos la conexión al nodo // $\Theta(1)$
10: **end if**
11: $iCorregirProfundidad(padreUlt, c)$ ▷ Corregimos la profundidad // $\Theta(log(\#nodos(c))$
12: $(*ultimoNodo).padre \leftarrow NULL$ ▷ $\Theta(1)$
13: $(*ultimoNodo).hijoIzq \leftarrow (*c.tope).hijoIzq$ ▷ $\Theta(1)$
14: $(*ultimoNodo).hijoDer \leftarrow (*c.tope).hijoDer$ ▷ $\Theta(1)$
15: $(*ultimoNodo).ramaMasCorta \leftarrow (*c.tope).ramaMasCorta$ ▷ $\Theta(1)$
16: $(*ultimoNodo).ramaMasLarga \leftarrow (*c.tope).ramaMasLarga$ ▷ $\Theta(1)$
17: $c.tope \leftarrow ultimoNodo$ ▷ $\Theta(1)$
18: $iSiftDown(c.tope, c)$ ▷ 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: $res \leftarrow \langle NULL \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

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

1: **if** $c.tope = NULL$ **then** ▷ La cola está vacia // $\Theta(1)$
2: $c.tope \leftarrow \langle a, 0, 0, NULL, NULL, NULL \rangle$ ▷ Se coloca el primer elemento // $\Theta(1)$
3: **else**
4: $puntero(nodo)\ futuroPadre \leftarrow iFuturoPadre(c)$ ▷ 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: $puntero(nodo)\ ultimoNodo \leftarrow c.tope$ ▷ 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: $puntero(nodo)\ ultimoNodo \leftarrow c.tope$ ▷ 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 // Θ(1)
2:      (*p).ramaMasCorta ← 0                                ▷ // Θ(1)
3:      (*p).ramaMasLarga ← 0                                ▷ // Θ(1)
4:  else
5:      if (*p).hijoIzq ≠ NULL ∧ (*p).hijoDer ≠ NULL then    ▷ Vemos si tiene dos hijos // Θ(1)
6:          (*p).ramaMasCorta ← 1                            ▷ // Θ(1)
7:          (*p).ramaMasLarga ← 1                            ▷ // Θ(1)
8:      else
9:          (*p).ramaMasCorta ← 0                            ▷ Tiene un solo hijo // Θ(1)
10:         (*p).ramaMasLarga ← 1                            ▷ // Θ(1)
11:     end if
12: end if
13: p ← (*p).padre                                           ▷ Subo un nivel // Θ(1)
14: while (*p).padre ≠ NULL do                               ▷ Recorre hasta el tope del arbol // Θ(1)
15:     (*p).ramaMasCorta ← (*(*p).hijoDer).ramaMasCorta + 1 ▷ // Θ(1)
16:     (*p).ramaMasLarga ← (*(*p).hijoIzq).ramaMasLarga + 1 ▷ // Θ(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 Θ(1)
2:  if iEsMayor?((*(*p).hijoIzq).elemento, (*p).elemento) then ▷ Vemos si el hijo izquierdo es mayor Θ(1)
3:      swap ← (*p).hijoIzq                                  ▷ Θ(1)
4:  end if
5:  if iEsMayor?((*(*p).hijoDer).elemento, (*swap).elemento) then ▷ Vemos si el hijo derecho es el mayor Θ(1)
6:      swap ← (*p).hijoDer                                  ▷ Θ(1)
7:  end if
8:  if p ≠ swap then                                         ▷ Vemos si es necesario reacomodar los nodos Θ(1)
9:      iIntercambio(p, swap)                                ▷ Intercambiamos el padre con el hijo Θ(1)
10:     iSiftDown(c, swap)                                   ▷ Llamamos a la funcion recursivamente Θ(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: $res \leftarrow \&\langle c.tope \rangle$ ▷ $\Theta(1)$

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*, $\sigma$)

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 esta definida en el diccionario, la definición se puede hacer en tiempo $O(|K|)$, donde $K \in string$ es la clave mas larga.

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

## Interfaz

**parámetros formales**
**géneros** $string, \sigma$

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

**se explica con**: DICCIONARIO($string, \sigma$)

**géneros**: `dicc`($string, \sigma$)

### Operaciones básicas de diccionario

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

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

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

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

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

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

[1] Nótese 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 costara 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** $d$: `dicc(`$string, \sigma$`)`) $\rightarrow$ $res$ : `dicc(`$string, \sigma$`)`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} d\}$
**Complejidad:** $\mathcal{O}(Longitud(c) + Cardinal(claves))$
**Descripción:** genera una copia nueva del diccionario.

$\bullet = \bullet$(**in** $d_1$ : `dicc(`$string, \sigma$`)`, **in** $d_2$ : `dicc(`$string, \sigma$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} c_1 = c_2\}$
**Complejidad:** $\mathcal{O}iLongitud(c) + Cardinal(claves))$
**Descripción:** compara $d_1$ y $d_2$ por igualdad, cuando $\sigma$ posee operación de igualdad.
**Requiere:** $\bullet = \bullet$(**in** $s_1 : \sigma$, **in** $s_2 : \sigma$) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} (s_1 = s_2)\}$
**Complejidad:** $\Theta(equal(s_1, s_2))$
**Descripción:** función de igualdad de $\sigma$'s

### Especificación de las operaciones auxiliares utilizadas en la interfaz

**TAD** Diccionario Extendido($string$, $\sigma$)

**extiende** CONJUNTO($string$, $\sigma$)

**otras operaciones (no exportadas)**
Definido : $string \times$ dNodo $\longrightarrow$ bool
Obtener : $string \times$ dNodo $\longrightarrow$ $\sigma$ {Definido($string$, dNodo)}

**axiomas**
Definido$(c, d)$ $\equiv$ **if** $\emptyset?(c) \wedge d$.Definicion $\neq$ NULL **then**
true
**else**
**if** Longitud$(c) > 0 \wedge_L d$.Siguientes[int$(c[0])$] $\neq$ NULL **then**
Definido(Fin$(c)$, $d$.Siguientes[int$(c[0])$])
**else**
false
**fi**
**fi**
Obtener$(c, d)$ $\equiv$ **if** $\emptyset?(c)$ **then** d.Definicion **else** Obtener(fin$(c)$, $d$.Siguientes[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 seran tuplas de $\sigma$, para guardar la definicion, 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 estan compuestos por chars, y tomamos la convencion de que habra 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 algun nodo agregado anteriormente a la instancia, es decir: "que un nodo no apunte a nodos anteriores".

`dicc(`$string, \sigma$`)` **se representa con** `dicCadena`

donde `dicCadena` es `tupla(`*dNodo*: `puntero(Nodo)` , *claves*: `conjuntoLineal(string)` )
donde `Nodo` es `tupla(`*definicion*: $\sigma$ , *siguientes*: `vector(puntero(dNodo))` )

Rep : dNodo $\longrightarrow$ bool

Rep($d$) $\equiv$ 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.

Abs : dicc $d$ $\longrightarrow$ dicc($string, \sigma$) {Rep($d$)}
Abs($d$) $\equiv$ def?(clave, d.dNodo) = Definido(clave, d.dNodo)
obtener(clave, d.dNodo) = Obtener(clave, 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**() $\rightarrow$ $res$ : dicc
1: $res.dNodo \leftarrow \langle Vacio(), NULL \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iDefinir**(**in**/**out** $d$: dicc, **in** $c$: *string*, **in** $s$: $\sigma$)

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

Complejidad: $\mathcal{O}(Longitud(c))$
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, $\mathcal{O}(Longitud(c))$. En el peor caso, el diccionario estará vacío, entonces entrará en los dos *while*, sumando $\mathcal{O}(Longitud(c))$ por cada uno (estarán sumando porque estos *while* no están anidados). Luego, como sabemos que, aplicando álgebra de ordenes, $\mathcal{O}(Longitud(c)) + \Theta(1) = \mathcal{O}(Longitud(c))$, podemos decir que sólo nos quedará: $\mathcal{O}(Longitud(c)) + \mathcal{O}(Longitud(c)) = \mathcal{O}(Longitud(c))$.

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

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

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

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

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

---

**iClaves**(**in/out** $d$: `dicc`) $\rightarrow$ $res$ : conjuntoLineal($\sigma$)

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

Complejidad: $\Theta(1)$

---

**iCopiar**(**in** $d$: `dic`)$\rightarrow$ $res$ : dicc($string, \sigma$)

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

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

---

$\bullet =_i \bullet$(**in** $d1$ : `dicc`, **in** $d2$ : `dicc`)$\rightarrow$ $res$ : $bool$

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

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

---

**buscarRuptura**(**in** $p$: `puntero(Nodo)`, **in** $c$: $string$, **in/out** $ruptura$: `puntero(Nodo)`) $\rightarrow$ $res$ : $Nat$

1: $actual \leftarrow p$ ▷ $\Theta(1)$
2: $i \leftarrow 0$ ▷ $\Theta(1)$
3: $letraRuptura \leftarrow 0$ ▷ $\Theta(1)$
4: **while** $i < Longitud(c)$ **do** ▷ $\mathcal{O}(Longitud(c))$
5: **if** $cantidadNoNull(actual) > 1 \vee actual.Definicion \leq NULL$ **then** ▷ $\Theta(1)$
6: $ruptura \leftarrow actual$ ▷ $\Theta(1)$
7: $letraRuptura \leftarrow i$ ▷ $\Theta(1)$
8: **end if**
9: $actual \leftarrow actual.Siguientes[int(c[i])]$ ▷ $\Theta(1)$
10: $i \leftarrow i + 1$ ▷ $\Theta(1)$
11: **end while**
12: $res \leftarrow letraRuptura$ ▷ $\Theta(1)$

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

---

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

1: $actual \leftarrow p$ ▷ $\Theta(1)$
2: $i \leftarrow 0$ ▷ $\Theta(1)$
3: **while** $i < Longitud(c)$ **do** ▷ $\mathcal{O}(Longitud(c))$
4: $actual \leftarrow actual.Siguientes[int(c[i])]$ ▷ $\Theta(1)$
5: $i \leftarrow i + 1$ ▷ $\Theta(1)$
6: **end while**
7: $res \leftarrow actual$ ▷ $\Theta(1)$

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

---

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

1: $contador \leftarrow 0$ ▷ $\Theta(1)$
2: $i \leftarrow 0$ ▷ $\Theta(1)$
3: **while** $i < 256$ **do** ▷ $\Theta(1)$
4: **if** $p.Siguientes[i] \neq NULL$ **then** ▷ $\Theta(1)$
5: $contador \leftarrow contador + 1$ ▷ $\Theta(1)$
6: **end if**
7: $i \leftarrow i + 1$ ▷ $\Theta(1)$
8: **end while**
9: $res \leftarrow contador$ ▷ $\Theta(1)$

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

**mismasDefiniciones**(**in** *d*1 : `dicc`, **in** *d*2 : `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: $Cardinal(claves) * \mathcal{O}(Longitud(c))$
Justificación: El algoritmo tiene llamadas a funciones $\Theta(1)$, un *while* que ejecuta siempre $Cardinal(claves)$ veces, y dentro de este fucniones $\mathcal{O}(Longitud(c))$. Luego, su complejidad debe ser $Cardinal(claves)$*$\mathcal{O}(Longitud(c))$.

# 6. Módulo Juego

1. pokemonesCapturados es Nat
2. matriz($\alpha$) es Vector(Vector($\alpha$))

# Interfaz

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

**se explica con**: JUEGO.

**generos**: `Juego`.

## Operaciones basicas de Juego

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

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

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

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

POKÉMONS(**in** $e$: `jugador`, **in** $j$: `juego`) $\rightarrow$ $res$ : `itConj(<pokémon, nat>)`
**Pre** $\equiv$ {$e \in jugadore(j)$}
**Post** $\equiv$ {res = CrearIt(clasificarPokemons(pokémons(e, j)))}
**Complejidad:** $\Theta(1)$
**Descripción:** Retorna un iterador al conjunto de tuplas <pokémons, cantidad>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** $\equiv$ {true}
**Post** $\equiv$ {alias($res$, expulsados(j))}
**Complejidad:** $\Theta(1)$
**Descripción:** Retorna el conjunto de jugador expulsados por referencia.

POSCONPOKÉMONS(**in** $j$ : `juego`) $\rightarrow$ $res$ : `conj(coor)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {alias($res$, posConPokémons(j))}
**Complejidad:** $\Theta(1)$
**Descripción:** Retorna el conjunto de coordenadas donde hay pokémons por referencia.

POKÉMONENPOS(**in** $c$: `coor`, **in** $j$ : `juego`) $\rightarrow$ $res$ : `pokémon`
**Pre** $\equiv$ {$c \in$ posConPokémons(j)}
**Post** $\equiv$ {$res$ = pokémonEnPos(c, j))}
**Complejidad:** $\Theta(1)$
**Descripción:** Retorna por copia el pokémon que se encuentra en la coordenada c.

CANTMOVIMIENTOSPARACAPTURA(**in** $c$: `coor`, **in** $j$ : `juego`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {$c \in$ posConPokémons(j)}
**Post** $\equiv$ {$res$ = cantMovimientosParaCaptura(c, j))}
**Complejidad:** $\Theta(1)$
**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** $\equiv$ {true}
**Post** $\equiv$ {$mapa(res) = m \wedge \emptyset?(jugadores(res)) \wedge \emptyset?(expulsados(res)) \wedge \emptyset?(posConPokemons(res))$}
**Complejidad:** $\bigcirc(\#coordenadas(m))$
**Descripción:** agrega un pokemon en una coordenada valida al juego.

AGREGARPOKEMON(**in** $p$ : `Pokemon`, **in** $c$: `Coordenada`, **in/out** $j$ : `Juego`)
**Pre** $\equiv$ {$puedoAgregarPokemon(c, j)$}
**Post** $\equiv$ {$Ag(c, posConPokemons(j))$}
**Complejidad:** $\bigcirc(|P| + |EC| * log(|EC|))$
**Descripción:** agrega un pokemon en una coordenada valida al juego.

AGREGARJUGADOR(**in/out** $j$ : `Juego`))
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$Ag(ProxID(j), jugadores(j))$}
**Complejidad:** $\bigcirc(\#jugadores(j))$
**Descripción:** agrega un nuevo jugador, designandole la proxima ID disponible.

CONECTARSE(**in** $e$: `jugador` **in** $c$: `coord` **in/out** $j$ : `Juego`)
**Pre** $\equiv$ {$j_0 =_{obs}$ j $\wedge$ e $\in$ j.jugadores $\wedge_L$ $\neg$estaConectado(e,j) $\wedge$ posExistente(c,e.mapa)}
**Post** $\equiv$ {j $=_{obs}$ conectarse(e,c,$j_0$)}
**Complejidad:** $\Theta$(log(EC))
**Descripción:** Conecta un jugador al juego.

DESCONECTARSE(**in** $e$: `jugador` **in/out** $j$ : `Juego`)
**Pre** $\equiv$ {$j_0 =_{obs}$ j $\wedge$ e $\in$ j.jugadores $\wedge_L$ estaConectado(e,j)}
**Post** $\equiv$ {j $=_{obs}$ 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** $\equiv$ {$j_0 =_{obs}$ j $\wedge$ e $\in$ j.jugadores $\wedge_L$ $\neg$estaConectado(e,j) $\wedge$ posExistente(c,e.mapa)}
**Post** $\equiv$ {j $=_{obs}$ 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** $\equiv$ {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** $\equiv$ {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** $\equiv$ $\{hayPokemonCercano(c, j) \land es \subseteq jugadoresConcectados(j)\}$
**Post** $\equiv$ $\{res = entrenadoresPosibles(c, es, j)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve un conjunto con los jugadores cercanos a un pokemon.

INDICEDEREZA(**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 <tipo, nat>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 juador 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 pokemones salvajes del juego.El significado contienene 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 nesesarios 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 $\mathcal{O}(|P|)$, 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(pokemonesCapturados,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 $\longrightarrow$ bool

$\text{Rep}(j) \equiv$ 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) $(\forall C : conjuntoLineal(jugador))((\exists c : coordenada)(j.matrizJugadores[longitud(c)][latitud(c)] = C) \Rightarrow (\forall e : nat)(Pertenece?(e, C) \Rightarrow 0 \leq e \leq Longitud(j.jugadores) - 1))$
1) $(\forall e : nat)(0 \leq e \leq Longitud(j.jugadores) - 1 \Rightarrow_L$ (2) $\wedge$ (3)

2) $(j.jugadores[e].sanciones < 5 \Leftrightarrow Pertenece?(j.jugadoresNoExpulsados, e)) \wedge \neg Pertenece?(j.expulsados, e) \wedge_L$ (2bis)
2bis) $(Pertenece?(j.jugadoresNoExpulsados, e) \Rightarrow$ (2a) $\wedge$ $(j.jugadores[e].conectado \Rightarrow (\exists c : coordenada)(posExistente(c, j.mapa) \Rightarrow_L$ (2b) $\wedge_L$ (2c) $\wedge$ (2d) $\wedge_L$ (2e)) $\wedge$ (2f) $\wedge$ 2g
2a) $j.jugadores[e].lugarNoExpulsado$ señala a la e correspondiente en $j.jugadoresNoExpulsados$
2b) $j.jugadores[e].posicion = c \wedge_L Pertenece(j.matrizJugadores[longitud(c)][latitud(c)], e)$
2c) $j.jugadores[e].posMatriz$ señala a la e correspondiente en $j.matrizJugadores[c.longitud][c.latitud]$
2d) $(\forall c' : coordenada)(Pertenece?(coordenadas(j.mapa), c') \wedge c' \neq c \Rightarrow \neg Pertenece?(j.matrizJugadores[c'.longitud][c'.latitud], e))$
2e) $HayPokemonCercano(c, j) \Rightarrow_L j.jugadores[e].prioridad$ señala al lugar correspondiente, donde esté la tupla en la que está e, en la cola de prioridad de $PosPokemonCercano(c, j))$
2f) $(\forall p : tipo)(Pertenece?(p, j.jugadores[e].pokesRapido) \Rightarrow Definido?(j.jugadores[e].pokesRapido, p) \wedge Obtener(j.jugadores[e].pokesRapido, p)$ señala a la tupla correspondiente en $j.jugadores[e].pokemons$ la cual como tipo tiene a p)

3) $(j.jugadores[e].sanciones \geq 5 \Leftrightarrow \neg Pertenece?(j.jugadoresNoExpulsados, e)) \wedge Pertenece?(j.expulsados, e) \wedge_L$ (3bis)
3bis) $\neg j.jugadores[e].conectado \wedge Vacio?(j.jugadores[e].pokemons) \wedge Vacio?(Claves(j.jugadores[e].pokesRap$
$j.jugadores[e].prioridad = NULL \wedge j.jugadores[e].posMatriz = NULL \wedge j.jugadores[e].lugarNoExpulsado = NULL$

4) $(\forall c : coord)esta?(j.matrizPokemon, Longitud(c)) \wedge_{\text{L}} \neg vacia?(j.matrizPokemon[Longitud(c)]) \wedge_{\text{L}} esta?(j.matrizPokemon[Longitud(c)], Latitud(c)) \Rightarrow_{\text{L}} (j.matrizPokemon[Longitud(c)][Latitud(c)].hayPoke \iff definido?(j.posPokemons, c)) \vee (j.matrizPokemon[Longitud(c)][Latitud(c)].hayPoke = false \wedge j.matrizPokemon[Longitud(c)][Latitud(c)].iterador = NULL)$
4b) $(\forall c : coord)esta?(j.matrizPokemon, Longitud(c)) \wedge_{\text{L}} \neg vacia?(j.matrizPokemon[Longitud(c)]) \wedge_{\text{L}} esta?(j.matrizPokemon[Longitud(c)], Latitud(c)) \wedge j.matrizPokemon[Longitud(c)][Latitud(c)].hayPoke \Rightarrow_{\text{L}} (Siguiente(j.matrizPokemon[Longitud(c)][Latitud(c)].iter) = obetener?(j.posPokemons, c))$
5a) $(\forall c : coord)definido?(j.posPokemons, c) \Rightarrow (pertenece?(coordenadas(m), c) \wedge (\nexists c' : coord)(definido?(j.posPokemons, c') \wedge distEuclideana(c, c') \leq 25))$
5b) $(\forall c : coord)definido?(j.posPokemons, c) \Rightarrow_{\text{L}} obtener(j.posPokemons, c).cantMovCapt \leq 10 \iff ((\exists e : jugador)pertenece(j.jugadoresNoEliminados, e) \wedge_{\text{L}} j.jugadores[e].conectado \wedge distEuclideana(c, j.jugadores[e].posicion) \leq 4)$
5c) $(\forall c : coord)definido?(j.posPokemons, c) \Rightarrow_{\text{L}} dameJugadoresHeap(obtener(j.posPokemons, c).colaDePr$
$entrenadoresPosibles(c, j)$

6) $(\forall p : pokemon)(definido?(j.pokemonesTotales, p) \iff ((\exists e : jugador)pertenece(j.jugadoresNoEliminados, e) \wedge_{\text{L}} definido?(j.jugadores[e].pokemones, p) \vee ((\exists c : coord)definido?(j.posPokemons, c) \wedge_{\text{L}} obtener(j.posPokemons, c).tipo = p))$
6b) $(\forall p : pokemon)(definido?(j.pokemonesTotales, p) \Rightarrow_{\text{L}} \#obtener(j.pokemonesTotales, p).pos = pokesSalvajesDeTipo(j.posPokemos, p)$

6c) $(\forall p : pokemon)(definido?(j.pokemonesTotales, p) \Rightarrow_{\text{L}} \#obtener(j.pokemonesTotales, p).cant = pokesSalvajesDeTipo(j.posPokemos, p) + cantTotalPokes(j.jugadores, p)$

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

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

dameJugadoresHeap : $colaDeProridad(jugadorHeap) : h \longrightarrow$ conj(nat)
dameJugadoresHeap($h$) $\equiv$ **if** vacio?(h) **then** $\emptyset$ **else** Ag(Proximo(h).id, dameJugadorHeap(desencolar(h))) **fi**
pokesSalvajesDeTipo : $dicc(coord \times$ infoCoord):d $\longrightarrow$ nat

$pokemon : p$

pokesSalvajesDeTipo($d$,$p$) $\equiv$ pokesSalvajesDeTipoAux(d,p,claves(d)
pokesSalvajesDeTipoAux : $dicc(coord \times$ infoCoord):d $\longrightarrow$ nat

$pokemon : p$

$\times\ conj(coord) : C$
pokesSalvajesDeTipoAux($d$,$p$,$C$) $\equiv$ **if** $\emptyset?(C)$ **then**
0
**else**
**if** $obtener(d, dameUno(C)).tipo = p$ **then** 1 **else** 0 **fi** +pokesSalvajesDeTipoAux(d,p,sinUno(C))
**fi**
sumaPokesJugadorIgual : $tipo : p \times vector(jugador) : v \times nat : d \times nat : n \times nat : c \times juego : j \longrightarrow$ bool
sumaPokesJugadorIgual($p$,$v$, $d$, $n$, $j$) $\equiv$ $c = 0 \wedge cantTotalPokes(p, v, n, j) = d$
sumaPokesJugadorMenor : $tipo : p \times vector(jugador) : v \times nat : d \times nat : n \times nat : c \times juego : j \longrightarrow$ bool
sumaPokesJugadorMenor($p$,$v$, $d$, $n$, $j$) $\equiv$ $c > 0 \wedge cantTotalPokes(p, v, n, j) = d$
cantTotalPokes : $tipo : p \times vector(jugador) : v \times nat : n \times juego : j \longrightarrow$ nat
cantTotalPokes($p$,$v$,$n$,$j$) $\equiv$ **if** $n = -1$ **then**
0
**else**
**if** $\neg Pertenece?(j.expulsados, j.jugadores[n])$ **then**
$Obtener(v[n].pokesRapido, p).cant + sumaPokesJugadorIgual(p, v, n-1, j)$
**else**
$sumaPokesJugadorIgual(p, v, n-1, j)$
**fi**
**fi**
Abs : juego $j$ $\longrightarrow$ juego {Rep($j$)}
Abs($j$) $\equiv$ $j : juego$ |
$e.mapa = mapa(j)$
$e.jugadores = jugadores(j) \wedge_{\text{L}}$
$(\forall i : jugador) i \in jugadores(j) \Rightarrow_{\text{L}} estaConectado(i, j) = e.jugadores[i].estaConectado \wedge sanciones(i, j = e.jugadores[i].sanciones) \wedge$
$posicion(i, j) = e.jugadores[i].pokemons \wedge estaConectado(i, j) \Rightarrow_{\text{L}} posicion(i, j) = e.jugadores[i].posicion$
$expulsados(j) = e.expulsados$
$posConPokemons(j) = e.posPokemons().Claves() \wedge_{\text{L}} (\forall c : coord) c \in posConPokemons(j) \Rightarrow_{\text{L}} pokemonEnPos(c, j) = e.posConPokemons.Obtener(c, j).tipo \wedge cantMovimientosParaCaptura(c, j) = e.posPokemons.Obtener(c).cantMovCapt$

# Algoritmos

## Algoritmos del modulo

---

**iMapa**(**in** $j$ : `juego`) $\rightarrow$ $res$:map

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

---

---

**iJugadores**(**in** $j$ : `juego`) $\rightarrow$ $res$:it ListaJug

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

---

**iestaConectado**(**in** $e$ : `jugador`,**in** $j$ : `juego`) → $res$:bool

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

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

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

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

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

**iPokemons**(**in** $e$ : `jugador`,**in** $j$ : `juego`) → $res$:it multiconjunto(pokemon)

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

**iExpulsados**(**in** $j$ : `juego`) → $res$:it conjunto(jugador)

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

**iPosConPokemons**(**in** $j$ : `juego`) → $res$:it conjunto(coord)

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

**iPokemonEnPos**(**in** $c$ : `coord`,**in** $j$ : `juego`) → $res$: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`) → $res$: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** $m$ : mapa) $\rightarrow$ $res$:juego

```
1:  res.mapa ← m
2:  res.jugadoresNoExpulsados ← Vacio()
3:  res.jugadores ← Vacio()
4:  res.expulsados ← Vacio()
5:  res.matrizJugadores ← iCrearMatrizJug(m) O(#(coordenadas(m))
6:  res.matrizJugadores ← iCrearMatrizPokes(m) O(#(coordenadas(m))
7:  res.pokemons ← iVacio()
8:  res.posPokemons ← iVacio()
9:  res.pokemonsTotales ← iVacio()
10: res.cantidadPokeTotal ← 0
```

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

**iCrearMatrizJug**(**in** $m$ : mapa) $\rightarrow$ $res$:matriz(conj(jugador))

```
1:  res ← Vacio()
2:  it ← coordenadas(m)
3:  while haySiguiente(it) do
4:      c ← Siguiente(it)
5:      if longitud(res) < Longitud(c) then
6:          i ← longitud(res)                                   ▷ Θ(1)
7:          while i ≤ Longitud(c) do                            ▷ Θ(i)
8:              Agregar(res, i, iVacia())                       ▷ Θ(Longitud(c))
9:              i ← i + 1                                       ▷ Θ(1)
10:         end while
11:         if i < Longitud(res) then
12:             while i ≤ Longitud(c) do
13:                 Agregar(res, i, Vacio())                    ▷ Θ(Longitud(c))
14:                 i ← i + 1                                   ▷ Θ(1)
15:             end while
16:         end if
17:     end if
18:     if longitud(res[Longitud(c)]) < Latitud(c) then
19:         j ← longitud(res[Longitud(c)])                      ▷ Θ(1)
20:         while j ≤ Latitud(c) do                             ▷ Θ(1)
21:             Agregar(res[Longitud(c)], j, Vacio())           ▷ Θ(Latitud(c))
22:             j ← j + 1                                       ▷ Θ(1)
23:         end while
24:         if i < Longitud(Longitud) then
25:             while i ≤ Longitud(c) do
26:                 Agregar(res, i, Vacio())                    ▷ Θ(Longitud(c))
27:                 j ← j + 1                                   ▷ Θ(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 ← Vacio()
2:  it ← coordenadas(m)
3:  while haySiguiente(it) do
4:      c ← Siguiente(it)
5:      if longitud(res) < Longitud(c) then
6:          i ← longitud(res)                                              ▷ Θ(1)
7:          while i ≤ Longitud(c) do                                       ▷ Θ(i)
8:              Agregar(res, i, iVacia())                                  ▷ Θ(Longitud(c))
9:              i ← i + 1                                                  ▷ Θ(1)
10:         end while
11:         if i < Longitud(res) then
12:             while i ≤ Longitud(c) do
13:                 Agregar(res, i, Vacio())                               ▷ Θ(Longitud(res))
14:                 i ← i + 1                                              ▷ Θ(1)
15:             end while
16:         end if
17:     end if
18:     if longitud(res[Longitud(c)]) <Latitud(c) then
19:         j ← longitud(res[Longitud(c)])                                 ▷ Θ(1)
20:         while j ≤ Latitud(c) do                                        ▷ Θ(1)
21:             Agregar(res[Longitud(c)], j, < false, NULL >)              ▷ Θ(Longitud(res.[Longitud(c)]))
22:             j ← j + 1                                                  ▷ Θ(1)
23:         end while
24:         if i < Longitud(Longitud) then
25:             while i ≤ Longitud(c) do
26:                 Agregar(res, j, < false, NULL >)                       ▷ Θ(Longitud(res))
27:                 j ← j + 1                                              ▷ Θ(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: $j.cantidadPokeTotal \leftarrow j.cantidadPokeTotal + 1$ ▷ Se aumenta la cantidad total de pokémons // $\Theta(1)$
2: $itHeapmod\ itNuevoHeap \leftarrow iCrearHeapPokemon(j, c)$ ▷ Se crea el heap que contiene a todos los jugadores en radio 2 del pokémon // $\Theta(EC * log(EC))$
3: $itDic\ dicLinealPoke \leftarrow iDefinirRapido(j.posPokemon, c, \langle p, 0, itNuevoHeap\rangle)$ ▷ Se define la coordenada del pokémon y se coloca su infoCoord // $\Theta(1)$
4: $j.matrizPokemons[iLongitud(c)][iLatitud(c)] \leftarrow \langle true, dicLinealPoke\rangle$ ▷ 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** ▷ Se verifica que esté definido el tipo pokémon $p$ // $\Theta(iLongitud(p))$
6: $iObtener(j.pokemonsTotales, p).cant \leftarrow iObtener(j.pokemonsTotales, p).cant + 1$ ▷ Se incremente la cantidad de pokemones del tipo agregado // $\Theta(iLongitud(p))$
7: $AgrgarRapido(iObtener(j.pokemonsTotales, p).pos, dicLinealPoke)$ ▷ Se agrega el iterador al conjunto de posiciones // $\Theta(iLongitud(p))$
8: **else**
9: $iDefinir(j.pokemonsTotales, p, \langle 1, iVacio()\rangle)$ ▷ Se define el nuevo tipo de pokemon // $\Theta(iLongitud(p))$
10: $AgrgarRapido(iObtener(j.pokemonsTotales, p).pos, dicLinealPoke)$ ▷ 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 otra 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($\alpha$)

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

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

Complejidad: La complejidad del algoritmo es $\mathcal{O}(J)$ donde J es la cantidada de jugaores 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: $itConj \leftarrow AgregarRapido(j.matrizJugadores[c.longitud][c.latitud], e)$ ▷ $\mathcal{O}(copy(e)) = \Theta(1)$, 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: $nueva \leftarrow < CantidadPokemons(j.jugadores[e].pokemonesCapturados, j), e >$
8: $itHeap \leftarrow encolar(nueva, j.matrizPokemon[pokePosicion.longitud][pokePosicion.latitud].colaPrioridad)$ ▷ $\mathcal{O}(log(\#nodos(heap)))$
9: $j.jugadores[e].posicionCola \leftarrow itHeap$
10: **end if**

Complejidad: La complejidad del algoritmo es $\mathcal{O}(log(EC))$ donde EC es la cantidada de jugaores 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: $EliminarSiguiente(j.jugadores[e].posMatriz)$ ▷ $\mathcal{O}(1)$
3: $j.jugadores[e].posMatriz \leftarrow NULL$
4: **if** $hayPokemonCercano(c, j)$ **then**
5: $EliminarSiguiente(j.jugadores[e].posicionCola)$ ▷ $\mathcal{O}(log(\#nodos(heap)))$
6: $j.jugadores[e].posicionCola \leftarrow NULL$
7: **end if**

Complejidad: La complejidad del algoritmo es $\mathcal{O}(log(EC))$ donde EC es la cantidada de jugaores 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 $\mathcal{O}(log(EC))$.

---

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

1: $I \leftarrow j.jugadores[e].posicion$ ▷ $\Theta(1)$I de coordenada inicial
2: $F \leftarrow c$ ▷ $\Theta(1)$ F de cordenada final
3: $poke \leftarrow crearIt(j.pokemonsTotales.claves)$ ▷ $\Theta(1)$
4: **while** $HaySiguiente(poke)$ **do** ▷ $\mathcal{O}(PS * |P|)$
5: $posPoke \leftarrow CrearIt(Obtener(j.pokemonsTotales, Siguiente(poke)).pos)$ ▷ $\mathcal{O}(|P|)$
6: **while** $HaySiguiente(posPoke)$ **do** ▷ $\mathcal{O}(PS)$
7: **if** $SiguienteClave(Siguiente(posPoke)) = I$ **then**
8: $LaCordenadaEsInicio(posPoke, I, F, e, j)$ ▷ $\Theta(1)$
9: **end if**
10: **if** $SiguienteClave(Siguiente(posPoke)) = F$ **then**
11: $LaCoordenadaEsFinal(posPoke, I, F, e, j)$ ▷ $log(EC)$
12: **end if**
13: **if** $SiguienteClave(Siguiente(posPoke)) \neq I \wedge SiguienteClave(Siguiente(posPoke)) \neq F$ **then**
14: $LaCoordenadaEsOtra(posPoke, I, F, e, j)$ ▷ $\mathcal{O}(|P|)$
15: **end if**
16: **end while**
17: $Avanzar(poke)$ ▷ $\Theta(1)$
18: **end while**
19: $ElminiarSiguiente(j.jugadores[e].posMatriz)$ ▷ $\Theta(1)$
20: $j.jugadores[e].posMatriz \leftarrow AgregarRapido(j.matrizJugadores[c.longitud][c.latitud], e)$ ▷ $copy(e) = \Theta(1)$ ya que e es un nat.
21: $j.jugadores[e].posicion \leftarrow c$ ▷ $\Theta(1)$
22: **if** $DistEuclidia(I, F) \geq 10 \vee_L !HayCamino(I, F, mapa(j))$ **then** ▷ $\Theta(1) + \Theta(1) = \Theta(1)$
23: $j.jugadores[e].sanciones \leftarrow j.jugadores[e].sanciones + 1$ ▷ $\Theta(1)$
24: **end if**
25: **if** $j.jugadores[e].sanciones \geq 5$ **then** ▷ $\Theta(1)$
26: $ExpulsarJugador(e)$ ▷ $\mathcal{O}(log(EC)) + \mathcal{O}(PC * |P|)$
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 apuntara 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: $k \leftarrow SiguienteClave(Siguiente(posPoke))$ ▷ $\Theta(1)$
2: **if** $HayPokemonCercano(k, j)$ **then** ▷ $\Theta(1)$
3: **if** $!HayPokemonCercano(F, j) \vee_L PosPokemonCercano(k, j) \neq PosPokemonCercano(F, j)$ **then** ▷ $\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$
4: $SiguienteSignificado(Siguiente(posPoke)).cantMovCapt \leftarrow 0$ ▷ $\Theta(1)$ 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: $k \leftarrow SiguienteClave(Siguiente(posPoke))$ ▷ $\Theta(1)$
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: $SiguienteSignificado(Siguiente(posPoke)).cantMovCap \leftarrow 0$ ▷ $\Theta(1)$
5: $jugadorHeap \leftarrow \langle cantidadPokemons(e, j), e \rangle$ ▷ $\Theta(1)$
6: $heap \leftarrow SiguienteSignificado(Siguiente(posPoke)).colaPrioridad$ ▷ $\Theta(1)$
7: $EliminarSiguiente(j.jugadores[e].prioridad)$ ▷ $\mathcal{O}(log(EC))$
8: $j.jugadores[e].prioridad \leftarrow Encolar(jugadorHeap, heap)$ ▷ $\mathcal{O}(log(EC))$
9: **end if**
10: **if** $!HayPokemonCercano(I, j)$ **then** ▷ $\Theta(1)$
11: $SiguienteSignificado(Siguiente(posPoke)).cantMovCap \leftarrow 0$ ▷ $\Theta(1)$
12: $jugadorHeap \leftarrow \langle cantidadPokemons(e, j), e \rangle$ ▷ $\Theta(1)$
13: $heap \leftarrow SiguienteSignificado(Siguiente(posPoke)).colaPrioridad$ ▷ $\Theta(1)$
14: $j.jugadores[e].prioridad \leftarrow Encolar(jugadorHeap, heap)$ ▷ $\mathcal{O}(log(EC))$
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: $k \leftarrow SiguienteClave(Siguiente(posPoke))$ ▷ $\Theta(1)$
2: **if** $HayPokemonCercano(k, j) \wedge SiguienteSignificado(posPoke).cantMovPCapt = 9$ **then** ▷ $\Theta(1) + \Theta(1) = \Theta(1)$
3: $CapturarPokemon(posPoke, j)$ ▷ $\mathcal{O}(|P|)$
4: **else**
5: **if** $HayPokemonCercano(k, j) \wedge SiguienteSignificado(posPoke).cantMovPCapt < 9$ **then**
6: $SiguienteSignificado(posPoke).cantMovPCapt \leftarrow SiguienteSignificado(posPoke).cantMovPCapt + 1$
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: $k \leftarrow SiguienteClave(Siguiente(poke))$ ▷ $\Theta(1)$
2: $posibles \leftarrow SiguienteSignificado(Siguiente(poke)).colaPrioridad$ ▷ $\Theta(1)$
3: $tipo \leftarrow SiguienteSignificado(Siguiente(poke)).tipo$ ▷ $\Theta(1)$
4: **if** $HaySiguiente(posibles)$ **then** ▷ $\Theta(1)$
5: $jugGanador \leftarrow Proximo(posibles).id$ ▷ $\Theta(1)$
6: $DarlePokemon(jugGanador, tipo, j)$ ▷ $\mathcal{O}(|P|)$
7: $j.matrizPokemons[k.longitud][k.latitud] \leftarrow \langle false, NULL \rangle$ ▷ $\Theta(1)$
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: $iter \leftarrow NULL$
2: **if** $Definido?(j.jugadores[e].pokesRapido, p)$ **then** ▷ $\mathcal{O}(|P|)$
3: $iter \leftarrow Obtener(j.jugadores[e].pokesRapido, p)$ ▷ $\mathcal{O}(|P|)$
4: $Siguiente(iter).cant \leftarrow Siguiente(iter).cant + 1$ ▷ $\Theta(1)$
5: **else**
6: $nuevoPokemon \leftarrow \langle p, 1 \rangle$ ▷ $\Theta(1)$
7: $iter \leftarrow AgregarRapido(j.jugadores[e].pokemons, nuevoPokemon)$ ▷ $copy(nuevoPokemon) = \Theta(|P|)$, ya que nuevoPokemon es un string, y será, como mucho, el nombre más largo para un pokémon en el juego. ▷ $copy(nuevoPokemon) = \mathcal{O}(|P|)$
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: $j.cantidadPokeTot \leftarrow j.cantidadPokeTotal - CantidadPokemons(e, j)$ ▷ $\mathcal{O}(PC)$
2: $EliminarPokemons(e, j)$ ▷ $\mathcal{O}(PC * |P|)$
3: $EliminarSiguiente(j.jugadores[e].posMatriz)$ ▷ $\Theta(1)$
4: **if** $j.jugadores[e].prioridad \neq NULL$ **then** ▷ $\Theta(1)$
5: $EliminarSiguiente(j.jugadores[e].prioridad)$ ▷ $\mathcal{O}(log(EC))$
6: **end if**
7: $j.jugadores[e].conectado \leftarrow false$ ▷ $\Theta(1)$
8: $expulsarJ \leftarrow j.jugadores[e].lugarNoExpulsado$ ▷ $\Theta(1)$
9: $j.jugadores[e].lugarNoExpulsado \leftarrow NULL$ ▷ $\Theta(1)$
10: $EliminarSiguiente(expulsarJ)$ ▷ $\Theta(1)$
11: $AgregarRapido(j.Expulsados, e)$ ▷ $copy(e) = \Theta(1)$ 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: $iter \leftarrow pokemons(e, j)$ ▷ $\Theta(1)$
2: **while** $HaySiguiente(iter)$ **do** ▷ $\mathcal{O}(PC)$
3: $res \leftarrow res + Siguiente(iter).cant$ ▷ $\Theta(1)$
4: $Avanzar(iter)$ ▷ $\Theta(1)$
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: $iter \leftarrow pokemons(e, j)$ ▷ $\Theta(1)$ Iteramos sobre los pokemons del Jugador.
2: **while** $HaySiguiente(iter)$ **do** ▷ $\mathcal{O}(PC * |P|)$
3: $pokemon \leftarrow Siguiente(iter).tipo$ ▷ $\Theta(1)$ Tipo pokemon a eliminar.
4: $Obtener(pokemonsTotales, pokemon).cant \leftarrow Obtener(pokemonsTotales, pokemon).cant - Siguiente(iter).cant$ ▷ $\mathcal{O}(|P|)$
5: **if** $Obtener(pokemonsTotales, pokemon).cant = 0$ **then** ▷ $\mathcal{O}(|P|)$ Si todos los pokemons de ese tipo los tenia el jugador.
6: $claveAEliminar \leftarrow pokemon$ ▷ $\Theta(1)$
7: $Borrar(j.pokemonsTotales, pokemon)$ ▷ $\mathcal{O}(|P|)$ Borramos al pokemon del Trie
8: $EliminarSiguiente(iter)$ ▷ $\Theta(1)$ 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 funcón 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 ← true                                                    ▷ O(1)
3:     posibles ← coordARadio(c, j, 25)          ▷ Da un it de conjunto de coord a radio 5 O(1)
4:     while haySiguiente(posibles) do
5:         if j.matrizPokemons[longitud(Siguiente(posibles))][latitud(siguiente(posibles))].hayPoke then
6:             res ← false
7:         end if
8:         Avanzar(it)
9:     end while
10: elseres ← 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 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 ← false                                                       ▷ O(1)
2: posibles ← coordARadio(c, j, 4)          ▷ Da un it de conjunto de coord a radio 2 O(1)
3: while haySiguiente(posibles) do
4:     if j.matrizPokemons[longitud(Siguiente(posibles))][latitud(siguiente(posibles)].hayPoke then
5:         res ← 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 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 ← false                                                       ▷ O(1)
2: posibles ← coordARadio(c, j, 4)          ▷ Da un it de conjunto de coord a radio 2 O(1)
3: while haySiguiente(posibles) do
4:     cor ← j.matrizPokemons[longitud(Siguiente(posibles))][latitud(siguiente(posibles))]
5:     if cor.hayPoke then
6:         res ← 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 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`) → $res$:it Conj(jugadores)

```
1: entrenadres ← vacio()                                              ▷ O(1)
2: posibles ← coordARadio(c, j, 4)
3: while haySiguiente(posibles) do
4:     it ← CrearIt(j.matrizJugadores[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 ← CrearIt(entrenadores)
```

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

**iIndiceRareza**(**in** $p$ : `pokemon`,**in/out** $j$ : `juego`) → $res$:nat

```
1: res ← 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`) → $res$:nat

```
1: it ← j.pokemonksTotales
2: while haySiguiente(it) do
3:     if SiguienteClave(it) = p then
4:         res ← 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`) → $res$:nat

```
1: res ← j.cantidadPokeTotal
```

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

**iCoordARadio**(**in** $c$ : `coord`,**in** $j$ : `juego`,**in** $r$ : `nat` ) $\rightarrow$ $res$:it Conj(coord)

```
1:  coordenadas ← vacio()                                              ▷ O(1)
2:  if r ≤ Latitud(c) then
3:      i ← Latitud(c) − r
4:  elsei ← 0
5:  end if
6:  if r ≤ Longitud(c) then
7:      j ← Longitud(c) − r
8:  elsej ← 0
9:  end if
10: while i ≤ Latitud(c) + r do
11:     while j ≤ Lonigitud(c) + r do
12:         if posExistente(crearCoord(i, j), j.mapa) then
13:             AgregarRapido(coordenadas, crearCoord(i, j))
14:         end if
15:         j ← j + 1
16:     end while
17:     i ← i + 1
18: end while
19: res ← CrearIt(coordenadas)
```

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