Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2016

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

Trabajo Práctico 2

Diseño de tipos abstractos de datos

Grupo Estrellitas

Integrante	LU	Correo electrónico
Acuña Lai, Maria Sol	255/13	sol.acu@hotmail.com
Lasso, Andrés	714/14	lassoandres2@gmail.com
Berríos Verboven, Nicolás	046/12	nbverboven@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo Diccionario String(string, σ)

Este módulo implementa un diccionario en donde tanto definir, obtener el significado de una determinada clave y testear si la misma está definida puede hacerse en función de la longitud de dicha clave.

Por simplicidad se decidió restringir el dominio de las claves a únicamente elementos del tipo `string`

Para describir la complejidad de las operaciones, se llamará $copy(s)$ al costo de copiar un elemento $s \in \sigma$ e $equal(s_1, s_2)$ al costo de evaluar si dos elementos $s_1, s_2 \in \sigma$ son iguales.

Interfaz

usa: BOOL, NAT, STRING, LISTA ENLAZADA(STRING)

parámetros formales

géneros σ

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

función $\bullet = \bullet$(**in** s_1 : σ, **in** s_2 : σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (s_2 = s_2)$}
Complejidad: $\Theta(equal(s_1, s_2))$
Descripción: función de igualdad de σ's

se explica con: DICCIONARIO(STRING,σ)

generos: `diccString(string,`σ`)`

Operaciones básicas

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

DEFINIR(**in/out** d: `diccString(string,`σ`)`, **in** $clave$: `string`, **in** s: σ)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge \text{long}(clave) > 0$}
Post $\equiv$ {$d =_{\text{obs}} \text{definir}(clave, s, d_0)$}
Complejidad: $\Theta(K)$, donde K es la longitud de la clave más larga del diccionario
Descripción: recibe una clave con su significado y la define. Si la clave ya se encontraba definida, se reescribe.
Aliasing: los elementos $clave$ y s se definen por copia.

DEFINIDO?(**in** d: `diccString(string,`σ`)`, **in** $clave$: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \text{def?}(clave, d)$}
Complejidad: $\Theta(K)$, donde K es la longitud de la clave más larga del diccionario
Descripción: devuelve `true` sii k está definida en el diccionario.
Aliasing: $clave$ se pasa por referencia y no es modificable, como tampoco lo es d.

OBTENER(**in** d: `diccString(string,`σ`)`, **in** $clave$: `string`) $\rightarrow$ res : σ
Pre $\equiv$ {$\text{def?}(clave, d)$}
Post $\equiv$ {$res =_{\text{obs}} \text{obtener}(clave, d)$}
Complejidad: $\Theta(K)$, donde K es la longitud de la clave más larga del diccionario
Descripción: devuelve el significado de $clave$, siempre que esta se encuentre definida previamente en d.
Aliasing: $clave$ se pasa por referencia y res se devuelve del mismo modo. $clave$, res y d no son modificables.

BORRAR(**in/out** d: `diccString(string,`σ`)`, **in** $clave$: `string`)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge \text{def?}(clave, d)$}
Post $\equiv$ {$res =_{\text{obs}} \text{borrar}(clave, d_0)$}
Complejidad: $\Theta(\#(claves(d) + K)$, donde K es la longitud de la clave más larga del diccionario
Descripción: elimina tanto la clave como su significado de d.

#CLAVES(**in** d: `diccString(string,`σ`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}

Post $\equiv \{res =_{\text{obs}} \#\text{claves}(d)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de claves del diccionario.

Representación

Representación del diccionario string

La estructura elegida fue un trie, representado como un puntero a la raíz y un conjunto lineal donde se almacenan las claves. Este último se incluyó por comodidad, ya que reduce el tiempo insumido en el agregado, pudiendo realizarse en $\Theta(1)$.

Cada nodo del árbol contiene un puntero a su padre, un significado (en el caso de tenerlo) y un arreglo que contiene los 256 posibles caracteres que pueden formar parte de un string. De esta forma, a pesar del costo inicial que requiere la creación del arreglo se obtiene una ganancia debido a que el acceso a cada elemento es rápido, lo que permite recorrer el árbol de una forma eficiente.

`diccString(string,`σ`)` se representa con `dicT`

donde `dicT` es `tupla(`*raiz*: `puntero(nodo)` , *claves*: `conj(string)`)
donde `nodo` es `tupla(`*significado*: `puntero(`σ`)`, *anterior*: `puntero(nodo)`, *siguientes*: `arreglo_estático[256] de puntero(nodo))`

El invariante de representación puede escribirse como:

1. Todos las claves definidas en el trie están en $dicT.claves$.
2. Todos los elementos de $dicT.claves$ están definidos en el trie (esto es la vuelta del item anterior).

Rep : dicT $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$ (1) $\wedge$ (2)

Abs : dicT d $\longrightarrow$ dicc(string, σ) {Rep(d)}
Abs(d) $\equiv$ dic: dicc(string, σ) / $(\forall s : \sigma)((\text{def?}(s, dicc) \Leftrightarrow \text{esta?}(s, d.claves))) \ \wedge_{\text{L}}$
$(\forall s : \sigma)(\text{esta?}(s, d.claves) \Rightarrow_{\text{L}} (\text{obtener}(s, d) =_{\text{obs}} \text{obtener}(s, dic))$

Algoritmos

Algoritmos del módulo

---

iCrearDicc() $\rightarrow res : dicT$
1: $res \leftarrow \langle NULL, Vacio() \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iDefinir(**in/out** d: `dicT`, **in** $clave$: `string`, **in** $s: \sigma$) $\rightarrow$ $res: dicT$

1: $AgregarAtras(d.claves, clave)$ ▷ $\Theta(copy(clave))$
2: **if** $d.raiz = NULL$ **then**
3: $d.raiz \leftarrow \&NuevoNodo()$ ▷ $\Theta(1)$
4: **else**
5: $puntero(nodo)\ actual \leftarrow d.raiz$ ▷ $\Theta(1)$
6: $i \leftarrow 0$ ▷ $\Theta(1)$
7: **while** $i < Longitud(clave)$ **do**
8: $indice \leftarrow ord(clave[i])$ ▷ $\Theta(1)$
9: **if** (actual→siguientes)[indice] = NULL **then**
10: $(actual \rightarrow siguientes[indice]) \leftarrow \&NuevoNodo()$ ▷ $\Theta(1)$
11: $((actual \rightarrow siguientes[indice]) \rightarrow padre) \leftarrow actual$ ▷ $\Theta(1)$
12: **end if**
13: $actual \leftarrow (actual \rightarrow siguientes[indice])$ ▷ $\Theta(1)$
14: $i \leftarrow i + 1$ ▷ $\Theta(1)$
15: **end while**
16: **end if**
17: **if** $(actual \rightarrow significado) \neq NULL$ **then**
18: $(actual \rightarrow significado) \leftarrow NULL$ ▷ $\Theta(1)$
19: **end if**
20: $(actual \rightarrow significado) \leftarrow \&s$ ▷ $\Theta(1)$

Complejidad: $\Theta(K)$, donde $K = max\{Longitud(k), \forall k \in d.claves\}$ es la longitud de la clave más larga del diccionario.

Complejidad: El algoritmo tiene un ciclo *while* que se ejecuta tantas vaces como la cantidad de caracteres que tiene la clave, es decir, Longitud(clave) veces. Dentro del ciclo se ejecutan tres asignaciones con orden de complejidad $\Theta(1)$ y un if con orden de complejidad $\Theta(1)$. Fuera del ciclo, se ejecutan operaciones que tienen complejidad $\Theta(1)$ más una con costo $\Theta(copy(clave))$. Para un string, el costo de copiar es el costo de copiar cada uno de sus elementos. Asumiendo que esto, por ser los `char` tipo primitivos, cuesta $\Theta(1)$, termina arrojando un costo final de $\Theta(Longitud(clave))$

Luego la complejidad total dada una clave es:

$$\Theta(Longitud(clave))\Theta(1) + \left(\sum_{i=0}^{Longitud(clave)} \Theta(1)\right) + \Theta(1) + \Theta(1) = \Theta(Longitud(clave))$$

Suponiendo el peor caso (aquel en que se recorre la clave más larga para agregar *clave*), La complejidad se vuelve $\Theta(K)$, donde $K = max\{Longitud(k), \forall k \in d.claves\}$.

iDefinido?(**in** d : `dicT`, **in** *clave* : `string`) $\rightarrow$ res : *bool*

1: $esta\_definido \leftarrow false$ ▷ $\Theta(1)$
2: $i \leftarrow 0$ ▷ $\Theta(1)$
3: $puntero(nodo)$ $actual \leftarrow \&d.raiz$ ▷ $\Theta(1)$
4: **while** $i < longitud(clave) \wedge actual \neq NULL$ **do** ▷ $\Theta(Longitud(clave))$
5: $indice \leftarrow ord(clave[i])$ ▷ $\Theta(1)$
6: $actual \leftarrow (actual \rightarrow siguientes)[indice]$ ▷ $\Theta(1)$
7: $i \leftarrow i + 1$ ▷ $\Theta(1)$
8: **end while**
9: **if** $actual \neq NULL \wedge (actual \rightarrow significado) \neq NULL$ **then** ▷ $\Theta(1)$
10: $esta\_definido \leftarrow true$ ▷ $\Theta(1)$
11: **end if**
12: $res \leftarrow esta\_definido$ ▷ $\Theta(1)$

Complejidad: $\Theta(K)$, donde $K = max\{Longitud(k), \forall k \in d.claves\}$ es la longitud de la clave más larga del diccionario

Justificación: El algoritmo tiene un ciclo que se repite, en el peor caso, hasta que se termine de recorrer cada elemento de la clave. El costo de las operaciones realizadas en cada iteración, así como las realizadas en el resto del algoritmo, es $\Theta(1)$. El costo total resulta:

$$\Theta(1) + \Theta(1) + \Theta(1) + \left(\sum_{i=0}^{Longitud(clave)} \Theta(1)\right) + \Theta(1) + \Theta(1) = \Theta(Longitud(clave)).$$

La clave puede pertenecer al diccionario y ser la de mayor longitud o no estar, en cuyo caso la clave más larga podría ser un prefijo de esta. En cualquiera de los dos casos, el ciclo recorre todos los elementos de la mayor de las claves definidas en d.

Luego, la complejidad del algoritmo es $\Theta(K)$, donde K es la longitud de la clave más larga del diccionario.

---

iObtener(**in** d : `dicT`, **in** *clave* : `string`) $\rightarrow$ res : σ

1: $puntero(nodo)$ $actual \leftarrow \&d.raiz$ ▷ $\Theta(1)$
2: $i \leftarrow 0$ ▷ $\Theta(1)$
3: **while** $i < Longitud(clave)$ **do** ▷ $\Theta(Longitud(clave))$
4: $indice \leftarrow ord(clave[i])$ ▷ $\Theta(1)$
5: $actual \leftarrow (actual \rightarrow siguientes)[indice]$ ▷ $\Theta(1)$
6: $i \leftarrow i + 1$ ▷ $\Theta(1)$
7: **end while**
8: $res \leftarrow (actual \rightarrow significado)$ ▷ $\Theta(1)$

Complejidad: $\Theta(K)$, donde $K = max\{Longitud(k), \forall k \in d.claves\}$ es la longitud de la clave más larga del diccionario

Justificación: El algoritmo tiene un ciclo que, como es precondición que la clave esté definida en diccionario, iterará $Longitud(clave)$ veces hasta recorrer todos sus elementos. Dentro del ciclo, se realizan operaciones con costo $\Theta(1)$ así como en el resto del algoritmo. El costo total es:

$$\Theta(1) + \Theta(1) + \left(\sum_{i=0}^{Longitud(clave)} \Theta(1)\right) + \Theta(1) = \Theta(Longitud(clave))$$

El peor caso ocurre cuando la clave cuyo significado se desea obtener es la más larga de todas la que están definidas en el diccionario. Puedo llamar K a la longitud de dicha clave y, entonces la complejidad en el peor caso es $\Theta(K)$.

---

i#Claves(**in** d : `dicT`) $\rightarrow$ res : nat

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

Complejidad: $\Theta(1)$

Justificación: Obtener la longitud de un *string* es $\Theta(1)$, ya que se trata de un `areglo(char)`.

---

iBorrar(**in/out** d: `dicT`,**in** *clave*: `string`)

1: $it \leftarrow CrearIt(d.claves)$ ▷ $\Theta(1)$
2: **while** $Siguiente(it) \neq clave$ **do** ▷ $[*]$
3: $Avanzar(it)$ ▷ $\Theta(1)$
4: **end while**
5: $EliminarSiguiente(it)$ ▷ $\Theta(1)$
6: *puntero*(*nodo*) $actual \leftarrow \&d.raiz$ ▷ $\Theta(1)$
7: *puntero*(*nodo*) $hasta\_aca\_es\_prefijo \leftarrow \&d.raiz$ ▷ Si una parte de la clave se comparte con otras del diccionario, esta es la posición que marca dónde se diferencia la clave actual. // $\Theta(1)$
8: $borro\_desde\_esta\_letra \leftarrow clave[0]$
9: **for** $i \leftarrow 0$ **to** $i < Longitud(clave)$ **do** ▷ Busco el el trie hasta encontrar el lugar en donde se encuentra definida la clave.
10: **if** $caminosPosibles(actual) > 1 \vee (actual \rightarrow significado) \neq NULL$ **then**
11: $hasta\_aca\_es\_prefijo \leftarrow actual$ ▷ $\Theta(1)$
12: $borro\_desde\_esta\_letra \leftarrow clave[i]$ ▷ $\Theta(1)$
13: **end if**
14: $actual = (actual \rightarrow siguientes)[ord(clave[i])]$ ▷ $\Theta(1)$
15: **end for**
16: **if** $CaminosPosibles(actual) > 0$ **then** ▷ La clave es prefijo de otra. //$\Theta(1)$
17: $(actual \rightarrow significado) \leftarrow NULL$
18: **else** ▷ $\Theta(1)$ ▷ Caso contrario
19: $BorrarSubstring(d, hasta\_aca\_es\_prefijo, actual, borro\_desde\_esta\_letra)$ ▷ $\Theta(Longitud(clave))$
20: **end if**

Complejidad: $\Theta(\#Claves(d)K)$, donde $K = max\{Longitud(k), \forall k \in d.claves\}$ es la longitud de la clave más larga del diccionario
Justificación: Calcular la longitud de un *string* tiene complejidad $\Theta(1)$. El algoritmo tiene un ciclo for que recorrera los caracteres del *string* pasado por parametro. Dentro del ciclo realiza operaciones con complejidad $\Theta(1)$. Luego entra en un if que tiene complejidad $\Theta(Longitud(clave))$.

Antes de eso existe un ciclo que recorre, en el peor caso, toda la lista de claves para eliminar *clave*. En cada iteración realiza una comparación del string buscado con el siguiente del iterador. Nuevamente considerando el peor caso, esto correspondería a comparar en cada paso cadenas con la longitud máxima definida en el diccionario. La cantidad de comparaciones de este ciclo es

$$[*]\Theta(\sum_{i=0}^{Longitud(d.claves)-1} equal(s_1, s_2))$$

donde K es la longitud de la clave más larga del diccionario. Luego, la complejida para una clave dada está dada por:

$$\begin{aligned}\Theta(\#Claves(d)K) + \Theta(1) + \Theta(\sum_{i=0}^{Longitud(clave)} c) + \Theta(1) + \Theta(K) = \\ = \Theta(\#Claves(d)K) + \Theta(K) + \Theta(K) + \Theta(1) + \Theta(1) \\ = \Theta(\#Claves(d)K + K) = \Theta(\#Claves(d)K)\end{aligned}$$

Operaciones auxiliares (privadas)

NuevoNodo(**in** d: `dicT`, **in** *actual*: `puntero(nodo)`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \langle NULL, NULL, crearArreglo(256)\rangle\}$
Complejidad: $\Theta(1)$
Descripción: devuelve en número de hijos de un nodo dado.
Aliasing: d no es modificable.

CaminosPosibles(**in** d: `dicT`, **in** *actual*: `puntero(nodo)`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \sum_{i=0}^{255} \beta(actual.siguientes[i] \neq NULL)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve en número de hijos de un nodo dado.
Aliasing: d no es modificable.

BorrarSubstring(**in/out** d: `dicT` **in** *desde*: `puntero(nodo)` **in** *hasta*: `puntero(nodo)` **in** *ultimo_char_borrado*: `char`) $\rightarrow$ *res* : `nat`

Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}}$ borrar(secuencia comprendida entre *desde* y *hasta*, $d_0)\}$
Complejidad: $\Theta(K)$, donde K es la longitud de la clave más larga del diccionario.
Descripción: devuelve en número de hijos de un nodo dado.
Aliasing: tanto *desde* como *ultimo_char_borrado* no son modificables.

---

iNuevoNodo() $\rightarrow$ *res* : *nodo*

1: $res \leftarrow \langle NULL, NULL, CrearArreglo(256)\rangle$ ▷ $\Theta(1)$
2: $i \leftarrow 0$ ▷ $\Theta(1)$
3: **while** $i < 256$ **do**
4: $\quad res.siguientes[i] \leftarrow NULL$ ▷ $\Theta(1)$
5: **end while**

Complejidad: $\Theta(1)$
Justificación: El ciclo presente en el algoritmo siempre itera 256 veces. La operaciones realizada dentro tienen costo $\Theta(1)$. La complejidad se calcula, entonces, como:

$$\Theta(1) + \sum_{i=0}^{255} \Theta(1) = \Theta(1)$$

---

iCaminosPosibles(**in** *actual* : `puntero(nodo)`) $\rightarrow$ *res* : *nat*

1: $count \leftarrow 0$ ▷ $\Theta(1)$
2: **for** $i \leftarrow 0$ **to** 255 **do** ▷ $\Theta(\sum_{i=0}^{255} c)$
3: $\quad$ **if** $(actual \rightarrow siguientes)[i] \neq NULL$ **then** ▷ $\Theta(1) + \Theta(1)$
4: $\quad\quad count \leftarrow count + 1$ ▷ $\Theta(1)$
5: $\quad$ **end if**
6: **end for**
7: $res \leftarrow count$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: El ciclo presente en el algoritmo siempre itera 256 veces. Todas las operaciones realizadas dentro tienen costo $\Theta(1)$. La complejidad se calcula como:

$$c_1 + \sum_{i=0}^{255}(c_2 + c_3 + c_4) + c_5 = c_1 + 256 \times (c_2 + c_3 + c_4) + c_5 = \Theta(1)$$

---

iBorrarSubstring(**in/out** d: `dicT` **in** *desde*: `puntero(nodo)` **in** *hasta*: `puntero(nodo)` **in** *ultimo_char_borrado*: `char`)

1: **while** $desde \neq hasta$ **do** ▷ $[*]$
2: $\quad puntero(nodo)\ anterior \leftarrow (hasta \rightarrow padre)$ ▷ $\Theta(1)$
3: $\quad (hasta \rightarrow padre) \leftarrow NULL$ ▷ $\Theta(1)$
4: $\quad hasta \leftarrow NULL$ ▷ $\Theta(1)$
5: $\quad hasta \leftarrow anterior$ ▷ $\Theta(1)$
6: **end while**
7: $indice \leftarrow ord(ultimo\_char\_borrado)$ ▷ $\Theta(1)$
8: $(desde \rightarrow siguientes)[indice] \leftarrow NULL$ ▷ $\Theta(1) + \Theta(1)$

Complejidad: $\Theta(K)$, donde K es la longitud de la clave más larga del diccionario.
Justificación: $[*]$ Todas las operaciones que se realizan tanto dentro del ciclo como fuera de él tienen costo $\Theta(1)$. Por lo tanto, la complejidad del algoritmo estará determinada por la cantidad de iteraciones que realiza el ciclo.

Supongamos el peor caso, en que *desde* corresponde a la raíz del trie y *hasta* al nodo que contiene el significado de la clave más larga. El ciclo iterará una cantidad de veces igual a la distancia en nodos entra *hasta* y la raíz. En el peor caso, esto es la longitud de la clave más larga definida en en diccionario.

---

2. Módulo Cola De Prioridad(α)

Este módulo implementa una cola de prioridad que permite obtener el tope (léase elemento más prioritario) en $\Theta(1)$. Además, el costo de encolar y desencolar es $\Theta(\log(n))$, donde n corresponde al tamaño de la cola. En contraposición a esto, desencolar un elemento i distinto del tope es menos eficiente, acarreando un costo de $\Theta(n\ \log(n))$.

Siguiendo con la lógica del TAD Cola de prioridad(α), en donde solamente es posible acceder al miembro más prioritario, se provee un iterador que no permite recorrer todos los elementos encolados, sino únicamente al próximo. El iterador también permite eliminar en tiempo constante un elemento que no sea necesariamente el próximo. Para esto, al encolar se devuelve un iterador al elemento agregado.

Por cuestiones de implementación no se permite modificar el elemento obtenido mediante la operación Siguiente.

Para describir la complejidad de las operaciones que provee el módulo llamaremos $copy(a)$ al costo de copiar un elemento $a \in \alpha$, $equal(a_1, a_2)$ al de evaluar si dos elementos $a_1, a_2 \in \alpha$ son iguales y $ordered(a_1, a_2)$ al de evaluar la expresión $a_1 < a_2$.

Interfaz

usa: Bool, Nat

parámetros formales

géneros α

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

función $\bullet = \bullet$(**in** a_1 : κ, **in** a_2 : κ) $\to$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (a_1 = a_2)$}
Complejidad: $\Theta(equal(a_1, a_2))$
Descripción: función de igualdad de α's

función $\bullet < \bullet$(**in** a_1 : α, **in** a_2 : α) $\to$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (a_1 < a_2)$}
Complejidad: $\Theta(ordered(a_1, a_2))$
Descripción: relación de orden total estricto entre α's.

se explica con: Cola De Prioridad(α), Iterador Bidireccional Modificable (α)

generos: `colaPrior(`α`)`, `itColaPrior(`α`)`

Operaciones Básicas

Vacia() $\to$ res : `colaPrior(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia}
Complejidad: $\Theta(1)$
Descripción: genera una cola de prioridad vacía.

Encolar(**in/out** c: `colaPrior(`α`)`, **in** a: α) $\to$ res : `itColaPrior(`α`)`
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ encolar(a, c_0) $\wedge$ HaySiguiente?(res) $\wedge_{\text{L}}$ Siguiente(res) = a $\wedge$ alias(SecuSuby(res), c)}
Complejidad: $\Theta(\log(n)(copy(a) + ordered(a_1, a_2)))$, donde n es el número de elementos de la cola
Descripción: agrega un elemento a a la cola. Devuelve un iterador al elemento agregado.
Aliasing: el elemento a se encola por copia.

EsVacia?(**in** c: `colaPrior(`α`)`) $\to$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia?(c)}
Complejidad: $\Theta(1)$
Descripción: devuelve `true` sii la cola no posee elementos.

Proximo(**in** c: `colaPrior(`α`)`) $\to$ res : α
Pre $\equiv$ {$\neg$ vacia?(c)}
Post $\equiv$ {$res =_{\text{obs}}$ proximo(c)}
Complejidad: $\Theta(1)$

Descripción: devuelve el proximo de la cola.
Aliasing: *res* no es modificable.

DESENCOLAR(**in/out** c: `colaPrior(`α`)`)
Pre $\equiv \{c =_{\text{obs}} c_0 \ \wedge \ \neg\text{vacia?}(c)\}$
Post $\equiv \{c =_{\text{obs}} \text{desencolar}(a, c_0)\}$
Complejidad: $\Theta(\log(n)(ordered(a_1, a_2)))$, donde n es el número de elementos de la cola
Descripción: desencola al próximo de c.

TAMAÑO(**in** c: `colaPrior(`α`)`) $\rightarrow$ res : `nat`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{tamaño}(c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de elementos encolados en c.

Operaciones del iterador

CREARIT(**in** c: `colaPrior(`α`)`) $\rightarrow$ res : `itColaPrior(`α`)`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{crearItBi}(<>, c) \wedge \text{alias}(\text{esPermutacion}(\text{SecuSuby}(res), c))\}$
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 sii se elimiia el elemento siguiente del iterador sin utilizar la operación ELIMINARSIGUIENTE.

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

SIGUIENTE(**in** it: `itColaPrior(`α`)`) $\rightarrow$ res : α
Pre $\equiv \{\text{HaySiguiente}(it)\}$
Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it))\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente a la posición del iterador.
Aliasing: res no es modificable.

ELIMINARSIGUIENTE(**in/out** it: `itColaPrior(`α`)`)
Pre $\equiv \{it = it_0 \wedge \text{HaySiguiente?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{EliminarSiguiente}(it_0)\}$
Complejidad: $\Theta(\log(n))$, donde n es el número de elementos de la cola
Descripción: elimina de la cola iterada el valor que se encuentra en la posición siguiente del iterador.

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Cola de prioridad Extendida(α)

extiende COLA DE PRIORIDAD(α)

otras operaciones (exportadas)
tamaño : colaPrior(α) $\longrightarrow$ nat

axiomas
tamaño(c) $\equiv$ **if** vacia?(c) **then** 0 **else** 1 + tamaño(desencolar(c)) **fi**

Fin TAD

Representación

La cola de prioridad se representa como un puntero a la raíz y un `nat` que corresponde a la cantidad de elementos

encolados. La estructura, en general, puede verse como un árbol en donde cada nodo tiene un puntero tanto al siguiente como al anterior en la lista resultante de aplicar breadth-first search. Por comodidad, esta última es una lista circular en donde el anterior de la raíz es el último nodo.

Cada nodo tiene, además de los ya mencionados punteros al anterior y al siguiente, otros al dato que contiene, padre y a los hijos derecho e izquierdo.

La elección de esta estructura en particular se basó en que, en un árbol de punteros, es posible insertar y eliminar la raíz en tiempo logarítmico. Sin embargo, un inconveniente que se presenta es qué sucede cuando se pretende borrar un nodo intermedio ya que buscar en un árbol es lineal en la cantidad de elementos en el peor caso. Otra complicación está en encontrar cuál es la posición en donde se debe agregar un nuevo elemento para mantener la propiedad estructural del heap. Una solución naïve es buscarla, pero esto es $\Theta(n)$ en el peor caso.

Frente a este problema, la respuesta que surge naturalmente es la de dotar al heap de un iterador que permita recorrer el árbol como si se hiciera BFS. Para que encontrar el siguiente fuera rápido es que se agregaron los punteros *anterior* y *siguiente*. Luego, es posible realizar la inserción y el borrado de cualquier elemento de heap en $\Theta(\log(n))$, donde n es el tamaño de la cola de prioridad.

Representación de la cola de prioridad

`colaPrior(`α`)` **se representa con** `minHeap`

donde `minHeap` es `tupla(`*raiz*: `puntero(nodo)` , *tamaño*: `nat`)
donde `nodo` es `tupla(`*dato*: `puntero(`α`)` , *padre*: `puntero(nodo)` , *hijoI*: `puntero(nodo)` , *hijoD*: `puntero(nodo)`
, *siguiente*: `puntero(nodo)` , *anterior*: `puntero(nodo)`)

El invariante de representación puede escribirse como:

1. El tamaño de la cola de prioridad es 0 cuando no posee elementos (i.e. cuando la raíz es un puntero nulo).
   $(h.\text{raíz} = \text{NULL}) = (h.\text{tamaño} = 0)$
2. El heap puede recorrerse como una lista circular en donde a continuación del último elemento está el primero.
   $h.\text{tamaño} \neq 0 \Rightarrow_{\text{L}} (\text{Nodo}(h,h.\text{tamaño}) = h.\text{raíz})$
3. La raíz es el primer elemento del heap si se lo observa como una lista enlazada. Además, es el único que puede serlo.
   $h.\text{tamaño} \neq 0 \Rightarrow_{\text{L}} ((\forall i : nat)(1 \leq i < h.\text{tamaño} \Rightarrow (\text{Nodo}(h,i) \neq h.\text{raíz})))$
4. Todo nodo es el anterior de su siguiente, y visceversa.
   $h.\text{tamaño} \neq 0 \Rightarrow_{\text{L}}$
   $(\forall i : nat)(0 \leq i < h.\text{tamaño} \Rightarrow (\text{Nodo}(h,i)\rightarrow\text{siguiente} = \text{Nodo}(h,i+1) \wedge \text{Nodo}(h,i+1)\rightarrow\text{anterior} = \text{Nodo}(h,i)))$
5. La raíz no tiene padre.
   $h.\text{tamaño} \neq 0 \Rightarrow_{\text{L}} ((h.\text{raíz})\rightarrow\text{padre} = \text{NULL})$
6. Todos los nodos son padres de sus hijos.
   $h.\text{tamaño} \neq 0 \Rightarrow_{\text{L}}$
   $(\forall i : nat)(0 \leq i < h.\text{tamaño} \Rightarrow_{\text{L}}$
   $((2i+1 \geq h.\text{tamaño} \Rightarrow \text{Nodo}(h,i)\rightarrow\text{hijoI} = \text{NULL}) \wedge$
   $(2i+1 < h.\text{tamaño} \Rightarrow_{\text{L}} \text{Nodo}(h,i)\rightarrow\text{hijoI} = \text{Nodo}(h,2i+1) \wedge \text{Nodo}(h,2i+1)\rightarrow\text{padre} = \text{Nodo}(h,i)) \wedge$
   $(2i+2 \geq h.\text{tamaño} \Rightarrow \text{Nodo}(h,i)\rightarrow\text{hijoD} = \text{NULL}) \wedge$
   $(2i+1 < h.\text{tamaño} \Rightarrow_{\text{L}} \text{Nodo}(h,i)\rightarrow\text{hijoI} = \text{Nodo}(h,2i+2) \wedge \text{Nodo}(h,2i+2)\rightarrow\text{padre} = \text{Nodo}(h,i))))$
7. La estructura cumple la propiedad de heap minimal.
   $h.\text{tamaño} \neq 0 \Rightarrow_{\text{L}} (h.\text{tamaño} = 1 \vee_{\text{L}} (\forall i : nat)(1 \leq i < h.\text{tamaño} \Rightarrow_{\text{L}} (\text{Nodo}(h,i)\rightarrow\text{dato} \geq \text{Nodo}(h,\lfloor \frac{i-1}{2} \rfloor)\rightarrow\text{dato})))$

Rep : minHeap $\longrightarrow$ bool
Rep(h) $\equiv$ true $\iff$ (1) $\wedge_{\text{L}}$ (2) $\wedge_{\text{L}}$ (3) $\wedge$ (4) $\wedge$ (5) $\wedge$ (6) $\wedge_{\text{L}}$ (7)

Abs : minHeap h $\longrightarrow$ colaPrior(α) {Rep(h)}
Abs(h) $\equiv$ **if** $h.raiz = \text{NULL}$ **then** vacia **else** encolar($^*(h.raiz \rightarrow dato)$), Abs(FinLst($h$))) **fi**

Nodo : minHeap h × nat $\longrightarrow$ puntero(nodo) $\{h.\text{raíz} \neq \text{NULL} \wedge 0 \leq i \leq h.\textit{tamaño}\}$
Nodo(h,i) $\equiv$ **if** $i = 0$ **then** h.raíz **else** Nodo(FinLst(h), $i-1$) **fi**

FinLst : minHeap $\longrightarrow$ minHeap
FinLst(l) $\equiv$ $\langle h.\text{raíz}\rightarrow\text{siguiente}, h.\text{tamaño} - \text{mín}\{h.\text{tamaño}, 1\}\rangle$

Representación del iterador

El iterador es simplemente un puntero al nodo siguiente más otro al heap, de forma de poder modificarlo. Debido a que la estructura elegida para representar la cola de prioridad se asemeja a la lista enlazada del apunte de diseño, el iterador se construyó con el de dicho módulo en mente y, por lo tanto, tanto el invariante de representación como la función de abstracción del iterador son los mismos.

`itColaPrior(`α`)` **se representa con** `iter`

donde `iter` es `tupla`(*siguiente*: `puntero(nodo)` , *colaP*: `puntero(minHeap)`)

Rep : iter $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ Rep($^*(it.\text{heap})$) $\wedge_{\text{L}}$ (it.siguiente = NULL $\vee_{\text{L}}$ $(\exists i\text{: nat})$(Nodo($^*it$.heap, i) = it.siguiente)

Abs : iter it $\longrightarrow$ itBi(α) $\{\text{Rep}(it)\}$
Abs(it) $=_{\text{obs}}$ b: itBi(α) | Siguientes(b) = Abs(Sig(it.colaP, it.siguiente)) $\wedge$
Anteriores(b) = Abs(Ant(it.colaP, it.siguiente))

Sig : puntero(minHeap) h × puntero(nodo) p $\longrightarrow$ minHeap $\{\text{Rep}(\langle p, h\rangle)\}$
Sig(i,p) $\equiv$ $\langle p, h\rightarrow\text{tamaño} - \text{Pos}(*h, p)\rangle$

Ant : puntero(minHeap) h × puntero(nodo) p $\longrightarrow$ minHeap $\{\text{Rep}(\langle p, h\rangle)\}$
Ant(i,p) $\equiv$ $\langle$ **if** $p = h\rightarrow$raíz **then** NULL **else** $h\rightarrow$raíz **fi**, Pos($*h$, p) $\rangle$

Pos : minHeap h × puntero(nodo) p $\longrightarrow$ nat $\{\text{Rep}(\langle p, h\rangle)\}$
Pos(h,p) $\equiv$ **if** h.raíz = $p \vee h$.tamaño = 0 **then** 0 **else** 1 + Pos(FinLst(h), p) **fi**

Algoritmos

Algoritmos del módulo

---

iVacia() $\rightarrow$ res : $minHeap$
1: $res \leftarrow \langle NULL, 0\rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iEncolar(**in/out** h : `minHeap`, **in** a : α) $\rightarrow$ res : $iter$
1: $it \leftarrow CrearIt(h)$ ▷ $\Theta(1)$
2: $AgregarAlFinal(it, a)$ ▷ $\Theta(copy(a))$
3: $SiftUp(it)$ ▷ $\Theta(\log(n)\ (ordered(a_1, a_2)))$
4: $(it.colaP \rightarrow tamaño) \leftarrow (it.colaP \rightarrow tamaño) + 1$ ▷ Actualizo el tamaño de la cola de prioridad // $\Theta(1)$
5: $res \leftarrow it$

Complejidad: $\Theta(\log(n)(ordered(a_1, a_2)) + copy(a))$, $n = h.tamaño$
Justificación:
$$\Theta(1) + \Theta(copy(a)) + \Theta(\log(n)(ordered(a_1, a_2))) = \Theta(\log(n)(ordered(a_1, a_2)))$$

---

iEsVacia?(**in/out** h : `minHeap`) $\rightarrow$ res : $bool$
1: $res \leftarrow (h.raiz = NULL)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iProximo(**in/out** h : `minHeap`) $\rightarrow$ $res : \alpha$

1: $res \leftarrow Siguiente(CrearIt(h))$ ▷ $\Theta(1) + \Theta(1)$

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

---

iDesencolar(**in/out** h : `minHeap`)

1: $it \leftarrow CrearIt(h)$ ▷ $\Theta(1)$
2: $EliminarSiguiente(it)$ ▷ $\Theta(\log(n)(ordered(a_1, a_2))$
3: $(it.colaP) \rightarrow tamaño \leftarrow (it.colaP) \rightarrow tamaño - 1$ ▷ $\Theta(1)$

Complejidad: $\Theta(\log(n)(ordered(a_1, a_2))$, $n = h.tamaño$
Justificación:
$$2 \times \Theta(1) + \Theta(\log(h.tamaño)\ (ordered(a_1, a_2))) = \Theta(\log(h.tamaño)\ (copy(a) + ordered(a_1, a_2)))$$

---

iTamaño?(**in/out** h : `minHeap`) $\rightarrow$ $res : nat$

1: $res \leftarrow h.tamaño$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Algoritmos del iterador

---

iCrearIt(**in** h : `minHeap`) $\rightarrow$ $res : iter$

1: $res \leftarrow \langle h.raiz, h \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iHaySiguiente(**in** it : `iter`) $\rightarrow$ $res : bool$

1: $res \leftarrow (it.siguiente \neq NULL)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iSiguiente(**in** it : `iter`) $\rightarrow$ $res : \alpha$

1: $res \leftarrow *(it.siguiente \rightarrow dato)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iEliminarSiguiente(in/out it : `iter`)

1: **if** $(it.colaP \rightarrow tamaño) = 1$ **then** ▷ $\Theta(1)$
2: $puntero(nodo)tmp \leftarrow (it.colaP \rightarrow raiz)$ ▷ $\Theta(1)$
3: $(it.colaP \rightarrow raiz) \leftarrow NULL$ ▷ $\Theta(1)$
4: $tmp \leftarrow NULL$ ▷ $\Theta(1)$
5: **else**
6: $puntero(nodo)\ primero \leftarrow (it.colaP \rightarrow raiz)$ ▷ $\Theta(1)$
7: $puntero(nodo)\ ultimo \leftarrow ((it.colaP \rightarrow raiz) \rightarrow anterior)$ ▷ $\Theta(1)$
8: $Swap(it, it.siguiente, ultimo)$ ▷ $\Theta(1)$
9: $(((it.siguiente) \rightarrow anterior) \rightarrow siguiente) \leftarrow ((it.siguiente) \rightarrow siguiente)$ ▷ $\Theta(1)$
10: $(((it.siguiente) \rightarrow siguiente) \rightarrow anterior) \leftarrow ((it.siguiente) \rightarrow anterior)$ ▷ $\Theta(1)$
11: $puntero(nodo)\ ultPadre \leftarrow ((it.siguiente) \rightarrow padre)$ ▷ $\Theta(1)$
12: **if** (ultPadre→hijoD) = (it.siguiente) **then** ▷ $\Theta(1)$
13: $(ultPadre \rightarrow hijoD) \leftarrow NULL$ ▷ $\Theta(1)$
14: **else**
15: $(ultPadre \rightarrow hijoI) \leftarrow NULL$ ▷ $\Theta(1)$
16: **end if**
17: $(it.siguiente) \leftarrow NULL$ ▷ Elimino el nodo apuntado por $it.siguiente$. $\Theta(1)$
18: $(it.siguiente) \leftarrow ultimo$ ▷ Actualizo el siguiente del iterador con el nodo que ahora rompe el invariante del heap. // $\Theta(1)$
19: $SiftDown(it)$ ▷ $\Theta(\log(n)\ (ordered(a_1, a_2)))$
20: **end if**

Complejidad: $\Theta(\log(n)\ (ordered(a_1, a_2)))$, donde n es el tamaño de la cola.
Justificación: El peor caso ocurre cuando la cola de prioridad tiene más de un elemento, en cuyo caso se ingresa por la rama negativa del if. Todas las operaciones (salvo $SiftDown$) son asignaciones, desreferenciación de punteros y comparaciones que se realizan en $\Theta(1)$. La complejidad de la rama *else* es

$$9 \times \Theta(1) + \Theta(\log(n)\ (ordered(a_1, a_2))) = \Theta(\log(n)\ (ordered(a_1, a_2))).$$

Operaciones Auxiliares (privadas)

SiftUp(**in/out** it : `iter`)
Pre $\equiv$ {$it =_{\text{obs}} it_0 \wedge {}^*(it.colaP)$ posee, a lo sumo, un elemento menor que su padre o mayor que alguno de sus hijos}
Post $\equiv$ {Siguiente(it) $=_{\text{obs}}$ Siguiente(it_0) $\wedge\ {}^*(it.colaP)$ cumple la propiedad de heap minimal}
Complejidad: $\Theta(\log(n)\ (ordered(a_1, a_2)))$, donde n es la cantidad de elementos de ${}^*(it.colaP)$.
Descripción: dado un árbol en que, a lo sumo, una de sus posiciones rompe el invariante de heap, modifica dicho árbol para que lo cumpla.

SiftDown(**in/out** it : `iter`)
Pre $\equiv$ {$it =_{\text{obs}} it_0 \wedge {}^*(it.colaP)$ posee, a lo sumo, un elemento menor que su padre o mayor que alguno de sus hijos}
Post $\equiv$ {Siguiente(it) $=_{\text{obs}}$ Siguiente(it_0) $\wedge\ {}^*(it.colaP)$ cumple la propiedad de heap minimal}
Complejidad: $\Theta(\log(n)\ (ordered(a_1, a_2)))$, donde n es la cantidad de elementos de ${}^*(it.colaP)$.
Descripción: dado un árbol en que, a lo sumo, una de sus posiciones rompe el invariante de heap, modifica dicho árbol para que lo cumpla.

Swap(**in/out** it : `iter`, **in/out** n_1 : `puntero(nodo)`, **in/out** n_2 : `puntero(nodo)`)
Pre $\equiv$ {true}
Post $\equiv$ {}
Complejidad: $\Theta(1)$
Descripción: intercambia las posiciones de dos nodos del heap. Si alguno de los dos corresponde a la raiz, esta se actualiza.
Aliasing: los parámetros se pasan por copia.

AgregarAlFinal(**in/out** it : `iter`, **in** a : α)
Pre $\equiv$ {$it =_{\text{obs}} it_0$}
Post $\equiv$ {Siguiente(it) $=_{\text{obs}}$ $a \wedge$ SecucSuby(it) $=_{\text{obs}}$ SecuSuby(it_0) $\circ$ a}

Complejidad: $\Theta(copy(a))$
Descripción: inserta a en el heap de forma tal que quede como anterior de la raíz.
Aliasing: el elemento a se agrega por copia.

---

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

1: $der \leftarrow (it.siguiente \rightarrow hijoD)$ ▷ $\Theta(1)$
2: $izq \leftarrow (it.siguiente \rightarrow hijoI)$ ▷ $\Theta(1)$
3: **if** $izq \neq NULL \ \wedge \ *(it.siguiente \rightarrow dato) > *((izq \rightarrow dato)$ **then** ▷ Si existe un hijo izquierdo, lo comparo con el nodo actual y me quedo con el menor // $\Theta(1) + \Theta(ordered(a_1, a_2))$
4: $min \leftarrow izq$ ▷ $\Theta(1)$
5: **else**
6: $min \leftarrow (it.siguiente)$ ▷ $\Theta(1)$
7: **end if**
8: **if** $der \neq NULL \ \wedge \ *(min \rightarrow dato) > *((der \rightarrow dato)$ **then** ▷ $\Theta(1) + \Theta(ordered(a_1, a_2))$
9: $min \leftarrow der$ ▷ Si existe un hijo derecho, lo comparo con min y me quedo con el menor // $\Theta(1)$
10: **end if**
11: **if** $min \neq it.siguiente$ **then** ▷ Los casos en que la guarda es falsa son aquellos en que el nodo actual es una hoja o se reestableció la propiedad de heap // $\Theta(1)$
12: $Swap(it, (it.siguiente), min)$ ▷ $\Theta(1)$
13: $SiftDown(it)$ ▷ Restauro la propiedad de heap minimal // $\Theta(\log(n)\ (ordered(a_1, a_2)))$
14: **end if**

Complejidad: $\Theta(\log(n)\ (ordered(a_1, a_2)))$, donde n es la cantidad de elementos del heap.
Justificación: En primer lugar, vemos que en cada llamada recursiva se realizan operaciones con costo $\Theta(1)$, y $\Theta(ordered(a_1, a_2))$. El peor caso ocurre cuando la raiz debe intercambiarse con nodos intermedios hasta llegar a una hoja. En este caso, la cantidad de llamadas recursivas a $SiftDown$ será igual a la altura del árbol. Como este cumple la propiedad estructural de un heap, este valor será $\log n$, donde n es la cantidad total de elementos de h.

Luego, la complejidad del algoritmo es $\Theta(\log(n)\ (ordered(a_1, a_2)))$.

---

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

1: $puntero(nodo)\ tope \leftarrow (it.colaP \rightarrow raiz)$ ▷ $\Theta(1)$
2: **while** $it.siguiente \neq tope \ \wedge \ *(it.siguiente \rightarrow dato) < *((it.siguiente \rightarrow padre) \rightarrow dato)$ **do** ▷ $[*]$
3: $Swap(it, it.siguiente, (it.siguiente \rightarrow padre))$ ▷ $\Theta(1)$
4: **end while**

Complejidad: $\Theta(\log(n)(ordered(a_1, a_2)))$, donde n es la cantidad de elementos del heap.
Justificación: $[*]$ La única operación que se realiza fuera del `while` cuesta $\Theta(1)$, por lo que la complejidad del algoritmo estará determinada por la del ciclo. Dentro de este último se realizan dos operaciones: un intercambio y una asignación, ambas con costo $\Theta(1)$ respectivamente más la evaluación de la guarda, de costo $\Theta(ordered(a_1, a_2))$. Por lo tanto, resta ver la cantidad de iteraciones que se producen.

El peor caso ocurre cuando el elemento ubicado incorrectamente es una hoja, y debe intercambiarse hasta llegar a la raíz, recorriendo así la altura del heap. La cantidad de iteraciones del ciclo será $\log(n)$, donde n es la cantidad de elementos del heap.

La complejidad resulta $\Theta(\log(n)(ordered(a_1, a_2))) + \Theta(1) = \Theta(\log(n)(ordered(a_1, a_2)))$.

---

iSwap(in/out it : `iter`, **in/out** n_1 : `puntero(nodo)` **in/out** n_2 : `puntero(nodo)`)

1: $puntero(nodo)\ rz = (it.colaP \rightarrow raiz)$
2: **if** $rz = n_1$ **then** ▷ $\Theta(1)$
3: $\quad (it.colaP \rightarrow raiz) \leftarrow n_2$
4: **end if**
5: **if** $rz = n_2$ **then** ▷ $\Theta(1)$
6: $\quad (it.colaP \rightarrow raiz) \leftarrow n_1$
7: **end if**
8: $puntero(nodo)\ n_1Parent \leftarrow (n_1 \rightarrow padre)$ ▷ $\Theta(1)$
9: $puntero(nodo)\ n_1RChild \leftarrow (n_1 \rightarrow hijoD)$ ▷ $\Theta(1)$
10: $puntero(nodo)\ n_1LChild \leftarrow (n_1 \rightarrow hijoI)$ ▷ $\Theta(1)$
11: $puntero(nodo)\ n_1Prev \leftarrow (n_1 \rightarrow anterior)$ ▷ $\Theta(1)$
12: $puntero(nodo)\ n_1Next \leftarrow (n_1 \rightarrow siguiente)$ ▷ $\Theta(1)$
13: $((n1 \rightarrow anterior) \rightarrow siguiente) \leftarrow n_2$ ▷ $\Theta(1)$
14: $((n1 \rightarrow siguiente) \rightarrow anterior) \leftarrow n_2$ ▷ $\Theta(1)$
15: $((n2 \rightarrow anterior) \rightarrow siguiente) \leftarrow n_1$ ▷ $\Theta(1)$
16: $((n2 \rightarrow siguiente) \rightarrow anterior) \leftarrow n_1$ ▷ $\Theta(1)$
17: $(n_1 \rightarrow padre) \leftarrow (n_2 \rightarrow padre)$ ▷ $\Theta(1)$
18: $(n_1 \rightarrow hijoD) \leftarrow (n_2 \rightarrow hijoD)$ ▷ $\Theta(1)$
19: $(n_1 \rightarrow hijoI) \leftarrow (n_2 \rightarrow hijoI)$ ▷ $\Theta(1)$
20: $(n_1 \rightarrow anterior) \leftarrow (n_2 \rightarrow anterior)$ ▷ $\Theta(1)$
21: $(n_1 \rightarrow siguiente) \leftarrow (n_2 \rightarrow siguiente)$ ▷ $\Theta(1)$
22: $n_2 \rightarrow padre \leftarrow n_1Parent$ ▷ $\Theta(1)$
23: $n_2 \rightarrow hijoD \leftarrow n_1LChild$ ▷ $\Theta(1)$
24: $n_2 \rightarrow hijoI \leftarrow n_1RChild$ ▷ $\Theta(1)$
25: $n_2 \rightarrow padre \leftarrow n_1Parent$ ▷ $\Theta(1)$
26: $n_2 \rightarrow anterior \leftarrow n_1Prev$ ▷ $\Theta(1)$
27: $n_2 \rightarrow siguiente \leftarrow n_1Next$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Todas las operaciones que se realizan poseen costo $\Theta(1)$

iAgregarAlFinal(**in/out** it : `iter`, **in** a : α)

1: $puntero(nodo)\ sig \leftarrow (it.colaP \rightarrow raiz)$ ▷ $\Theta(1)$
2: $puntero(nodo)\ nuevo \leftarrow \&\langle a, NULL, NULL, NULL, NULL, NULL\rangle$ ▷ Reservo memoria para un nuevo nodo. // $\Theta(1)$
3: **if** $sig = NULL$ **then** ▷ Si el heap estaba vacío, agrego a a como raíz y también como primero de una lista circular. // $\Theta(1)$
4: $(nuevo \rightarrow anterior) \leftarrow nuevo$ ▷ $\Theta(1)$
5: $(nuevo \rightarrow siguiente) \leftarrow nuevo$ ▷ $\Theta(1)$
6: $(it.colaP \rightarrow raiz) \leftarrow nuevo$ ▷ $\Theta(1)$
7: **else** ▷ Si había por lo menos un elemento, tengo que insertar a como anterior de la raíz y como último elemento del árbol. // $\Theta(1)$
8: $(nuevo \rightarrow anterior) \leftarrow (sig \rightarrow anterior)$ ▷ $\Theta(1)$
9: $(nuevo \rightarrow siguiente) \leftarrow sig$ ▷ $\Theta(1)$
10: $puntero(nodo)\ nuevo\_parent \leftarrow (nuevo \rightarrow anterior \rightarrow padre)$
11: **if** $nuevo\_parent = NULL$ **then** ▷ En este caso, el heap tiene un solo elemento. Agrego el nuevo nodo como hijo izquierdo de la raíz.
12: $(nuevo \rightarrow anterior \rightarrow hijoI) \leftarrow nuevo$ ▷ $\Theta(1)$
13: $(nuevo \rightarrow padre) \leftarrow (nuevo \rightarrow anterior)$ ▷ $\Theta(1)$
14: **else** ▷ Hay más de un elemento en el heap. // $\Theta(1)$
15: **if** $nuevo\_parent \rightarrow hijoI \neq NULL$ **then** ▷ Inserto como hijo izquierdo del padre del nodo anterior. // $\Theta(1)$
16: $(nuevo\_parent \rightarrow hijoI) \leftarrow nuevo$ ▷ $\Theta(1)$
17: **else**
18: **if** $nuevo\_parent \rightarrow hijoD \neq NULL$ **then** ▷ Inserto como hijo derecho del padre del nodo anterior. // $\Theta(1)$
19: $(nuevo\_parent \rightarrow hijoD) \leftarrow nuevo$ ▷ $\Theta(1)$
20: **else**
21: $nuevo\_parent \leftarrow (nuevo\_parent \rightarrow siguiente)$ ▷ $\Theta(1)$
22: $(nuevo\_parent \rightarrow hijoI) \leftarrow nuevo$ ▷ $\Theta(1)$
23: **end if**
24: **end if**
25: $(nuevo \rightarrow padre) \leftarrow nuevo\_parent$ ▷ $\Theta(1)$
26: **end if**
27: **end if**
28: $(nuevo \rightarrow anterior \rightarrow siguiente) \leftarrow nuevo$ ▷ $\Theta(1)$
29: $(nuevo \rightarrow siguiente \rightarrow anterior) \leftarrow nuevo$ ▷ $\Theta(1)$
30: $it.siguiente \leftarrow nuevo$ ▷ Hago que el siguiente de it sea el nuevo nodo. // $\Theta(1)$

Complejidad: $\Theta(copy(a))$
Justificación: Todas las operaciones que se realizan tienen costo $\Theta(1)$ por lo que la complejidad total está dada por la de copiar el elemento que se desea agregar. Luego, esto es

$$\Theta(copy(a)) + \Theta(1) = \Theta(copy(a)).$$

3. Módulo Coordenada

Interfaz

usa: NAT

se explica con: COORDENADA.

géneros: `coordenada`.

Operaciones básicas

CREARCOORDENADA(**in** *lat*: `nat`, **in** *long*: `nat`) → *res* : `coordenada`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ crearCoor(*lat*, *long*)}
Complejidad: Θ(1)
Descripción: genera una coordenada con los parámetros especificados.

DISTEUCLIDEA(**in** c_1 : `coordenada`, **in** c_2 : `coordenada`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ distEuclidea(c_1, c_2)}
Complejidad: Θ(1)
Descripción: calcula la distancia euclidea entre las coordenadas c_1 y c_2.

COORDENADAARRIBA(**in** *c*: `coordenada`) → *res* : `coordenada`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ coordenadaArriba(c)}
Complejidad: Θ(1)
Descripción: Devuelve la coordenada de arriba mas proxima a la coordenada c

COORDENADAABAJO(**in** *c*: `coordenada`) → *res* : `coordenada`
Pre ≡ {*Latitud*(*c*) > 0}
Post ≡ {*res* =$_{\text{obs}}$ coordenadaAbajo(c)}
Complejidad: Θ(1)
Descripción: Devuelve la coordenada de abajo mas proxima a la coordenada c

COORDENADAALADERECHA(**in** *c*: `coordenada`) → *res* : `coordenada`
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ coordenadaAlaDerecha(c)}
Complejidad: Θ(1)
Descripción: Devuelve la coordenada de la derecha mas proxima a la coordenada c

COORDENADAALAIZQUIERDA(**in** *c*: `coordenada`) → *res* : `coordenada`
Pre ≡ {*Longitud*(*c*) > 0}
Post ≡ {*res* =$_{\text{obs}}$ coordenadaAlaIzquierda(c)}
Complejidad: Θ(1)
Descripción: Devuelve la coordenada de la izquierda mas proxima a la coordenada c

Representación

Representación de la coordenada

`coordenada` **se representa con** `crd`

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

Rep : crd ⟶ bool
Rep(*c*) ≡ true ⟺ true

Abs : crd *c* ⟶ coor {Rep(*c*)}

Abs(c) $\equiv$ crearCoor(e.latitud, e.longitud)

Algoritmos

Algoritmos del módulo

---

iCrearCoordenada(**in** lat : `nat`, **in** $long$: `nat`) $\rightarrow$ res : crd

1: $res \leftarrow \langle lat, long \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

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

1: $aux \leftarrow 0$ ▷ $\Theta(1)$
2: **if** $c_1.latitud \geq c_2.latitud$ **then**
3: $\quad aux \leftarrow (c_1.latitud - c_2.latitud) \times (c_1.latitud - c_2.latitud)$ ▷ $\Theta(1)$
4: **else**
5: $\quad aux \leftarrow (c_2.latitud - c_1.latitud) \times (c_2.latitud - c_1.latitud)$ ▷ $\Theta(1)$
6: **end if**
7: **if** $c_1.longititud \geq c_2.longititud$ **then**
8: $\quad aux \leftarrow aux + (c_1.longitud - c_2.longititud) \times (c_1.longitud - c_2.longitud)$ ▷ $\Theta(1)$
9: **else**
10: $\quad aux \leftarrow aux + (c_2.longitud - c_1.longititud) \times (c_2.longitud - c_1.longitud)$ ▷ $\Theta(1)$
11: **end if**
12: $res \leftarrow aux$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: El algoritmo tiene dos condicionales en los cuales en el peor caso realizan operaciones con costo $\Theta(1)$. Fuera de estos, se realizan dos asignaciones que tienen costo $\Theta(1)$. La complejidad del algoritmo es: $\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

---

iCoordArriba(**in** c : `crd`) $\rightarrow$ res : crd

1: $lat \leftarrow c.latitud + 1$ ▷ $\Theta(1)$
2: $long \leftarrow c.longitud$ ▷ $\Theta(1)$
3: $res \leftarrow CrearCoordenada(lat, long)$ ▷ $\Theta(1)$

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

---

iCoordAbajo(**in** c : `crd`) $\rightarrow$ res : crd

1: $lat \leftarrow c.latitud - 1$ ▷ $\Theta(1)$
2: $long \leftarrow c.longitud$ ▷ $\Theta(1)$
3: $res \leftarrow CrearCoordenada(lat, long)$ ▷ $\Theta(1)$

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

---

iCoordALaDerecha(**in** c : `crd`) $\rightarrow$ $res : crd$

1: $lat \leftarrow c.latitud$ ▷ $\Theta(1)$
2: $long \leftarrow c.longitud + 1$ ▷ $\Theta(1)$
3: $res \leftarrow CrearCoordenada(lat, long)$ ▷ $\Theta(1)$

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

iCoordALaIzquierda(**in** c : `crd`) $\rightarrow$ $res : crd$

1: $lat \leftarrow c.latitud$ ▷ $\Theta(1)$
2: $long \leftarrow c.longitud - 1$ ▷ $\Theta(1)$
3: $res \leftarrow CrearCoordenada(lat, long)$ ▷ $\Theta(1)$

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

4. Módulo Grilla(α)

TAD Grilla

TAD Grilla

igualdad observacional

$$(\forall g, g' : \text{grilla}) \left(g =_{\text{obs}} g' \iff \begin{pmatrix} \#Columnas(g) =_{\text{obs}} \#Columnas(g') \wedge \#Filas(g) =_{\text{obs}} \\ \#Filas(g') \wedge_{\text{L}} (\forall n : Nat)((0 \leq n < \#Filas) \Rightarrow_L (\forall m : \\ Nat)((0 \leq m < \#Columnas) \Rightarrow_L g[n][m] = g'[n][m] \end{pmatrix} \right)$$

géneros grilla

exporta grilla, observadores, generadores

usa VECTOR(α), NAT, BOOL, CONJUNTO(α)

observadores básicos

$\#Filas$	: grilla	$\longrightarrow$	Nat
$\#Columnas$	: grilla	$\longrightarrow$	Nat
$\bullet[\bullet][\bullet]$	: grilla g $\times$ Nat n $\times$ Nat m	$\longrightarrow$	Nat $\{0 \leq i < \#Filas(g) \wedge 0 \leq j < \#Columnas(g)\}$

generadores

CrearGrilla	:	$\longrightarrow$	grilla
AgColumnas	: grilla g $\times$ Nat n $\times$ α a	$\longrightarrow$	grilla
AgFilas	: grilla g $\times$ Nat n $\times$ α a	$\longrightarrow$	grilla

axiomas

$\#Filas(CrearGrilla())$	$\equiv$	0
$\#Columnas(CrearGrilla())$	$\equiv$	0
$\#Filas(AgColumnas(g,n,a))$	$\equiv$	$\#Filas(g)$
$\#Filas(AgFilas(g,n,a))$	$\equiv$	$\#Filas(g) + n$
$\#Columnas(AgFilas(g,n,a))$	$\equiv$	$\#Columnas(g)$
$\#Columnas(AgColumnas(g,n,a))$	$\equiv$	$\#Columnas(g) + n$
$AgColumnas(g,n,a)[i][j]$	$\equiv$	**if** $(0 \leq i < \#Filas(g) \wedge 0 \leq j < \#Columnas(g))$ **then** $g[i][j]$ **else** a **fi**
$AgFilas(g,n,a)[i][j]$	$\equiv$	**if** $(0 \leq i < \#Filas(g) \wedge 0 \leq j < \#Columnas(g))$ **then** $g[i][j]$ **else** a **fi**

Fin TAD

Interfaz

usa: BOOL, NAT,

parámetros formales

géneros: `grilla`. **se explica con:** GRILLA, VECTOR(α).

Operaciones básicas

CREARGRILLA() $\rightarrow$ *res* : `grilla`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$CrearGrilla()}
Complejidad: $\Theta(1)$
Descripción: Crea una grilla vacia.

•[•][•](**in** g : `grilla`,**in** n : `Nat`, **in** m : `Nat`) $\rightarrow$ res : `Nat`
Pre $\equiv$ $\{0 \leq i < \#Filas(g) \wedge 0 \leq j < \#Columnas(g)\}$
Post $\equiv$ $\{res =_{\text{obs}}$ g[n][m]$\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve por referencia el elemento que contiene la grilla en la n-ésima fila y m-ésima columna.

AgregarColumnas(**in** g : `grilla`, **in** n : `Nat`,**in** a : α) $\rightarrow$ res : `grilla`
Pre $\equiv$ {true}
Post $\equiv$ {res$=_{\text{obs}}$ AgColumnas(g,n,a)}
Complejidad: $\Theta(F.C)$
Descripción: Agrega n columnas a la matriz, donde cada elemento de las columnas agregadas sera a.

AgregarFilas(**in** g : `grilla`, **in** n : `Nat`,**in** a : α) $\rightarrow$ res : `grilla`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ AgFilas(g,n,a)}
Complejidad: $\Theta(1)$
Descripción: Agrega n filas a la matriz, donde cada elemento de las filas agregadas sera a.

#Filas(**in** g : `grilla`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ #Filas(g)}
Complejidad: $\Theta(1)$
Descripción: Indica la cantidad de filas de la grilla

#Columnas(**in** g : `grilla`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ #Columnas(g)}
Complejidad: $\Theta(1)$
Descripción: Indica la cantidad de columnas de la grilla

Representación

Representación de la grilla

La grilla se representa con un vector de vectores de α.

`grilla` **se representa con** `Vector(Vector(`α`))`

El invariante de representación es el siguiente:

1. Todos los vectores de la grilla que conforman el vector principal deben tener la misma longitud.
   $(\forall i : nat)(0 \leq i \leq longitud(grilla) \Rightarrow (\forall j : nat)(0 \leq j \leq longitud(grilla) \Rightarrow \text{longitud(grilla}[i]) = \text{longitud(grilla}[j])))$

Rep : grilla $\longrightarrow$ bool
Rep(g) $\equiv$ true $\Longleftrightarrow$ (1)

Abs : grl g $\longrightarrow$ grilla {Rep(g)}
Abs(g) $\equiv$ $g.longitud =_{\text{obs}} \#Filas(grilla) \wedge g[0].longitud =_{\text{obs}} \#Columnas(grilla) \wedge_{\text{L}} (\forall n : Nat)(0 \leq n < \#Filas(grilla) \Rightarrow_L (\forall m : Nat)(0 \leq m < \#Columnas(grilla) \Rightarrow_L g[n][m] =_{\text{obs}} grilla[n][m])$

Algoritmos

Algoritmos del módulo

iCrearGrilla() $\rightarrow res : grl$

1: $res \leftarrow Vacia()$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Se crea un vector(α) vacio. Según nos informa la interfaz de VECTOR(α) esto tiene costo $\Theta(1)$.

---

iAgregarColumnas(**in/out** g : `grl`, **in** n : `Nat`, **in** $a : \alpha$)

1: $i \leftarrow 0$ ▷ $\Theta(1)$
2: **while** $i < \#Filas(g) \vee EsVacio?(g)$ **do**
3: $j \leftarrow 0$ ▷ $\Theta(1)$
4: **while** $j < n$ **do**
5: $AgregarAtras(g[i], a)$ ▷ $O(\#Columnas(g) + copy(a))$
6: $j \leftarrow j + 1$ ▷ $\Theta(1)$
7: **end while**
8: $i \leftarrow i + 1$ ▷ $\Theta(1)$
9: **end while**

Complejidad: O((#Columnas(g)+copy(a))*n*#Filas(g))

Justificación: El algorítmo agrega n columnas a la grilla g pasada por parametro.Las columnas agregadas contendran en cada posición el elemento $a \in \alpha$ pasado por parametro. Para ésto se realiza un ciclo que se ejecuta $\#Filas(g)$ veces. La entrada i-ésima al ciclo, realiza otro ciclo que agrega atras n veces $a \in \alpha$ a la fila i-ésima. Cada inserción tiene costo O(#Columnas(g)+copy(a)), luego el costo total de éste ciclo es O((#Columnas(g)+copy(a))*n). Esto nos da una complejidad total de O((#Columnas(g)+copy(a))*n*#Filas(g)).

---

iAgregarFilas(**in/out** g : `grl`, **in** n : `Nat`, **in** $a : \alpha$)

1: $j \leftarrow 0$ ▷ $\Theta(1)$
2: **while** $j < n$ **do**
3: $AgregarAtras(g, Vacia())$ ▷ $O(\#Filas(g))$
4: $i \leftarrow 0$ ▷ $\Theta(1)$
5: **while** $i < \#Columnas(g)$ **do**
6: $AgregarAtras(Ultimo(g), a)$ ▷ $O(tamaño(Ultimo(g)) + copy(a))$
7: $i \leftarrow i + 1$ ▷ $\Theta(1)$
8: **end while**
9: $j \leftarrow j + 1$ ▷ $\Theta(1)$
10: **end while**

Complejidad: $O(n * max\{\#Filas(g), \#Columnas(g) * (\#Columnas(g) + copy(a))\})$

Justificación: El algorítmo agrega n filas a la grilla g pasada por parametro.Las filas agregadas contendran en cada posición el elemento $a \in \alpha$ pasado por parametro. Para esto el algoritmo realiza un ciclo que se ejecuta n veces, donde la entrada j-ésima al ciclo agrega atras un vector vacio v_j en el vector grilla cada entrada tiene costo O(tamaño(Ultimo(g)) + copy(a)). Puedo acotar tamaño(Ultimo(g)) por $\#Columnas(g)$. Luego, mediante otro ciclo agrega atras #Columnas(g) veces el elemento a $\in \alpha$ pasado por paramentro al vector v_j. Esto me da un costo total de
$O(n * (\#Filas(g) + \#Columnas(g) * (\#Columnas(g) + copy(a)))) = O(n * max\{\#Filas(g), \#Columnas(g) * (\#Columnas(g) + copy(a))\})$

---

i$\#Filas$(**in** g : `grl`, $\rightarrow res : Nat$

1: $res \leftarrow Longitud(g)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

i#*Columnas*(**in** g : `grl`) $\rightarrow$ res : Nat

1: $res \leftarrow Longitud(g[0])$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

i•[•][•](**in** g : `grl`,**in** n : `Nat`,**in** m : `Nat`) $\rightarrow$ res : Nat

1: $res \leftarrow g[n][m]$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

5. Módulo Mapa

Interfaz

usa: Bool, Nat, Coordenada, Conjunto Lineal(α).

se explica con: Mapa.

géneros: `mapa`.

Operaciones básicas

CrearMapa() $\rightarrow$ *res* : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ crearMapa()}
Complejidad: $\Theta(1)$
Descripción: Crea un mapa vacio.

AgregarCoordenada(**in** *c*: `coordenada`, **in** *m*: `mapa`) $\rightarrow$ *res* : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ agregarCoord(c,m)}
Complejidad: $\Theta(1)$
Descripción: Agrega una nueva coordenada al mapa.

Coordenadas(**in** *m*: `Mapa`) $\rightarrow$ *res* : `conj(coordenada)`
Pre $\equiv$ {true}
Post $\equiv$ {res$=_{\text{obs}}$ Coordenadas(m)}
Complejidad: $\Theta(1)$
Descripción: Devuelve el conjunto de coordenadas agregadas al mapa.

PosicionExistente(**in** *c*: `coordenada` **in** *m*: `Mapa`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ posExistente(c,m)}
Complejidad: $\Theta(1)$
Descripción: Indica se una posicion pertenece a las coordenadas del mapa.

HayCamino(**in** *c*1: `coordenada` **in** *c*2: `coordenada` **in** *m*: `Mapa`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {$c1 \in coordenadas(m) \land c2 \in coordenas(m)$}
Post $\equiv$ {res $=_{\text{obs}}$ hayCamino(c1,c2,m)}
Complejidad: $\Theta(1)$
Descripción: Indica si hay un camino entre la coordenada c1 y la coordenada c2

ExisteCamino(**in** *c*1: `coordenada`,**in** *c*2: `coordenada`,**in** *cs*: `conj(coordenada)`, **in** *m*: `Mapa`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {$c1 \in coordenadas(m) \land c2 \in coordenadas(m)) \land cs \subseteq coordenadas(m)$}
Post $\equiv$ {$res =_{\text{obs}} existeCamino(c1, c2, coordenadas(m) - \{c_1\}, m)$}
Descripción: Indica si existe un camino entre las coordenadas c_1 y c_2.

Representación

Representación del Mapa

`mapa` **se representa con** `map`

donde `map` es `tupla`(*grl*: `grilla(grillaStruct)`, *coordenadas*: `conj(coordenada)` , *maxLatitud*: `Nat` , *maxLongitud*: `Nat`)
donde `grillaStruct` es `tupla`(*caminos*: `grilla(Nat)` , *disponible*: `bool`)

El invariante de representación se escribe como

1. Todas las coordenadas del conjunto de coordenadas del mapa solo pueden existir dentro del mismo.

   ($\forall c$: *coordenada*)($c \in$ map.coordenadas $\Rightarrow_{\text{L}}$ (c.latitud $<$ #Filas(map.grl) $\land$ c.longitud $<$ #Columnas(map.grl) $\land$ c.latitud $\leq$ map.maxLatitud $\land$ c.longitud $\leq$ map.maxLongitud))

2. La grilla que se encuentra en cada una de las posiciones de *map.grl*, si es que no está vacía, posee las mismas dimensiones que esta.

   $(\forall n : nat)(0 \leq n < \#\text{Filas(map.grl)} \Rightarrow_{\text{L}} (\forall m : nat)(0 \leq m < \#\text{Columnas(map.grl)} \Rightarrow_{\text{L}}$
   $(\#\text{Filas(map.grl}[n][m]\text{.caminos)} = \#\text{Filas(map.grl)} \wedge \#\text{Columnas(map.grl}[n][m]\text{.caminos)} = \#\text{Columnas(map.grl))}$
   $\vee$ (map.grl$[n][m]$.caminos = CrearGrilla)))

3. La grilla mencionada en el punto anterior no es vacía sólo en el caso de la posición en la que se encuentra corresponda a una coordenada del mapa.

   $(\forall c : coordenada)(c \in \text{map.coordenadas} \Leftrightarrow \text{map.grl}[c\text{.latitud}][c\text{.longitud}] \neq \text{CrearGrilla})$

4. *map.grl*$[n][m]$.*caminos* es una grilla cuyas posiciones toman los valores 0 y 1. Habrá un 1 en *map.grl*$[n][m]$.*caminos*$[x][y]$ si y solo si hay un camino entre (n, m) y (x, y) en el mapa.

   $(\forall c : coordenada)($ $(\forall n : nat)(0 \leq n < \#\text{Filas(map.grl)} \Rightarrow_{\text{L}} (\forall m : nat)(0 \leq m < \#\text{Columnas(map.grl)} \Rightarrow_{\text{L}}$
   $(\text{map.grl}[c\text{.latitud}][c\text{.longitud}]\text{.caminos}[n][m] = 0 \vee \text{map.grl}[c\text{.latitud}][c\text{.longitud}]\text{.caminos}[n][m] = 1) \wedge_{\text{L}}$
   $(\text{map.grl}[c\text{.latitud}][c\text{.longitud}]\text{.caminos}[n][m] = 1) \Leftrightarrow \text{hayCamino}(c, \text{crearCoor}(n,m),\text{map}))))$

5. map.grl[n][m].disponible tomará el valor *true* si y solo sí una posicion (n, m) pertenece a las coordenadas del mapa.

   $(\forall c : coordenada)($ $(\forall n : nat)(0 \leq n < \#\text{Filas(map.grl)} \Rightarrow_{\text{L}} (\forall m : nat)(0 \leq m < \#\text{Columnas(map.grl)} \Rightarrow_{\text{L}}$
   $(\text{map.grl}[c\text{.latitud}][c\text{.longitud}]\text{.disponible} \Leftrightarrow c \in \text{map.coordenadas}))))$

Rep : map $\longrightarrow$ bool
Rep(map) $\equiv$ true $\Longleftrightarrow$ (1) $\wedge$ (2) $\wedge_{\text{L}}$ (3) $\wedge_{\text{L}}$ (4) $\wedge_{\text{L}}$ (5)

Abs : Mapa m $\longrightarrow$ map {Rep(m)}
Abs(m) $\equiv$ $m.coordenadas =_{\text{obs}} coordenadas(m)$

Algoritmos

Algoritmos del módulo

---

iCrearMapa $\rightarrow res : mapa$

1: $mapa.coordenadas \leftarrow Vacio()$ ▷ $\Theta(1)$
2: $mapa.grl \leftarrow CrearGrilla()$ ▷ $\Theta(1)$
3: $mapa.maxLatitud \leftarrow -1$ ▷ $\Theta(1)$
4: $mapa.maxLongitud \leftarrow -1$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Justificación:Información provista por las interfaces de Grilla y Conjunto Lineal (α)

---

iAgregarCoordenada(**in/out** m : `mapa` **in** c : `coordenada`)

1: **if** $longitud(c) > \#Columnas(g)$ **then** ▷ $\Theta(1)$
2: $difLongitud \leftarrow longitud(c) - \#Columnas(g)$ ▷ $\Theta(1)$
3: $AgregarColumnas(m.grl, difLongitud, \langle CrearGrilla(), false\rangle)$ ▷ $O((\#Col(g) * difLongitud * \#Filas(g))$
4: **end if**
5: **if** $latitud(c) > \#Filas(g)$ **then** ▷ $\Theta(1)$
6: $difLatitud \leftarrow latitud(c) - \#Filas(g)$ ▷ $\Theta(1)$
7: $AgregarFilas(m.grl, difLatitud, \langle CrearGrilla(), false\rangle)$ ▷ $O(difLatitud * max\{\#Filas(g), (\#Col(g))^2\})$
8: **end if**
9: **if** $EsVacio?(m.grl[latitud(c)][longitud(c)].caminos)$ **then** ▷ $\Theta(1)$
10: $AgregarRapido(m.coordenadas, c)$ ▷ $\Theta(1)$
11: $m.grl[latitud(c)][longitud(c)].disponible \leftarrow true$ ▷ $\Theta(1)$
12: $it \leftarrow CrearIt(m.coordenadas)$ ▷ $\Theta(1)$
13: **while** HaySiguiente(it) **do** ▷ $\Theta(1)$
14: $GrillaHayCamino(latitud(Siguiente(it)), longitud(Siguiente(it)), m)$ ▷ O(*)
15: $Avanzar(it)$
16: **end while**
17: $m.maxLongitud \leftarrow longitud(c)$ ▷ $\Theta(1)$
18: $m.maxLongitud \leftarrow latitud(c)$ ▷ $\Theta(1)$
19: **end if**

$* = O(max\{\#Columnas(m.grl)^2 * \#Filas(m.grl), \#Filas(m.grl)^2, \#m.coordenadas * (3^{\#m.coordenadas})\})$
Complejidad:
$O(max\{\#Filas(m.grl)^2, \#m.coordenadas * \#Columnas(m.grl)^2 * \#Filas(m.grl), \#m.coordenadas^2 * (3^{\#m.coordenadas})\})$
Justificación: Para agregar una coordenada al mapa miro si la latitud y longitud de la coordenada son menores a $\#Filas(g)$ y $\#Columnas(g)$ respectivamente, si no lo son, agrego la cantidad de filas y/o columnas necesarias para que esto se cumpla. Agregar filas tiene un costo $O(difLatitud * max\{\#Filas(g), (\#Col(g))^2\}) \subseteq O(max\{\#Filas(g)^2, \#Filas(g) * (\#Col(g))^2\})$, agregar columnas tiene un costo $O((\#Col(g) * difLongitud * \#Filas(g)) \subseteq O(\#Col(g)^2 * \#Filas(g))$.
Luego, me fijo si la coordenada c a agregar ya pertenecia al mapa. En el peor caso, c no pertenece a mapa y tengo que armar la grilla m.grl[latitud(c)][longitud(c)].caminos, cambiar el bool disponible a true y actualizar las grillas de las demas coordenadas del mapa. Para esto entro #m.coordenadas veces a un ciclo donde actualizo las grillas caminos de todas las coordenadas . Cada entrada tiene costo $O(max\{\#Columnas(m.grl)^2 * \#Filas(m.grl), \#Filas(m.grl)^2, \#m.coordenadas * (3^{\#m.coordenadas})\})$.
Concluimos que la complejidad de la funcion AgregarCoordenada es la siguiente:

$O(\#Col(g)^2 * \#Filas(g) + \#Filas(g) * max\{\#Filas(g), (\#Col(g))^2\} + max\{\#m.coordenadas * \#Columnas(m.grl)^2 * \#Filas(m.grl), \#Filas(m.grl)^2, \#m.coordenadas^2 * (3^{\#m.coordenadas})\}) =$

$$O(max\{\#Filas(m.grl)^2, \#m.coordenadas * \#Columnas(m.grl)^2 * \#Filas(m.grl), \#m.coordenadas^2 * (3^{\#m.coordenadas})\})$$

---

iPosExistente(**in** c : `coordenada`, **in** m : `mapa`) $\rightarrow$ res : $bool$

1: $res \leftarrow \neg(EsVacio?(m.grl[c.latitud][c.longitud].caminos))$ ▷ $\Theta(1)$

Complejidad:$\Theta(1)$

---

iHayCamino(**in** c_1 : `coordenada`,**in** c_2 : `coordenada`,**in** m : `mapa`) $\rightarrow$ res : $bool$

1: $res \leftarrow m.grl[latitud(c_1)][longitud(c_1)].caminos[latitud(c_2)][longitud(c_2)] == 1$

Complejidad: $\Theta(1)$

---

iExisteCamino(**in** c_1 : `coordenada`,**in** c_2 : `coordenada`,**in** cs : `conj(coordenada)`,**in** m : `mapa`) $\rightarrow$ res : $bool$

1: $res \leftarrow ExisteCaminoAux(c_1, c_2, m)$ ▷ $\mathcal{O}(3^{\#m.coordenadas})$
2: $RestaurarDisponibilidad(m)$ ▷ $\mathcal{O}(\#m.coordenadas)$

Complejidad: $\mathcal{O}(3^{\#m.coordenadas})$
Justificación: La complejidad de este algoritmo está dada por la complejidad de las funciones ExisteCaminoAux y RestaurarDisponibilidad. Luego la complejidad de ExisteCamino será :
$\mathcal{O}(3^{\#m.coordenadas}) + \mathcal{O}(\#m.coordenadas) = \mathcal{O}(3^{\#m.coordenadas})$

Operaciones Auxiliares (privadas)

RestaurarDisponibilidad(**in/out** m : `mapa`)
Pre $\equiv$ {true}
Post $\equiv$ $\{m.grl[i][j].disponible \leftrightarrow (i,j) \in m.coordenadas\}$
Complejidad: O($\#m.coordenadas$)
Descripción: Dado un mapa, recorre las posiciones de mapa.grl que representan las coordenadas pertenecientes al conjunto mapa.coordenadas y le asigna al booleano *disponible* el valor *true*.

GrillaHayCamino(**in** c: `coordenada`, **in/out** m : `mapa`)
Pre $\equiv$ $\{c \in m.coordenadas)\}$
Post $\equiv$ $\{(\forall i : nat)(0 \leq i < \#Filas(m.grl)) \Rightarrow (\forall j : nat)(0 \leq j < \#Columnas(m.grl)) \Rightarrow (m.grl[latitud(c)][longitud(c)].caminos[i][j] == 1 \vee m.grl[latitud(c)][longitud(c)].caminos[i][j] == 0) \wedge_{\text{L}} (m.grl[latitud(c)][longitud(c)].caminos[i][j] == 1 \leftrightarrow (i,j) \in m.coordenadas \wedge_{\text{L}} HayCamino(c,(i,j),m))\}$
Complejidad: O($max\{\#Columnas(m.grl)^2 * \#Filas(m.grl), \#Filas(m.grl)^2, \#m.coordenadas * (3^{\#m.coordenadas})\}$)
Descripción: Dada una coordenada c en m.coordenadas, le asigna a m.grl [$latitud(c)$ [$longitud(c)$].caminos una grilla de 0's y 1's del mismo tamaño que m.grl, un 1 indica que ExisteCamino desde la coordenada c hasta la coordenada que representa esa posicion.

ExisteCaminoAux(**in/out** m : `mapa`)
Pre $\equiv$ $\{c_1 \in coordenadas(m) \wedge c_2 \in coordenadas(m)\}$
Post $\equiv$ $\{res =_{\text{obs}} ExisteCamino(c_1, c_2, coordenadas(m) - c_1, m)\}$
Complejidad: $\mathcal{O}(3^{\#m.coordenadas})$
Descripción: Indica si existe un camino entre las coordenadas c_1 y c_2.

iRestaurarDisponibilidad(**in/out** m : `mapa`)

1: $it \leftarrow CrearIt(m.coordenadas)$ ▷ $\Theta(1)$
2: **while** HaySiguiente(it) **do**
3: $m.grl[latitud(Siguiente(it))][longitud(Siguiente(it))].disponible \leftarrow true$ ▷ $\Theta(1)$
4: Avanzar(it) ▷ $\Theta(1)$
5: **end while**

Complejidad: O($\#m.coordenadas$)
Justificación: Recorre todas las coordenadas del mapa para restituir el invariante de los booleanos *disponible* de cada posicion de m.grilla.

iGrillaHayCamino(**in** c: `coordenada`, **in/out** m: `mapa`)

1: $it \leftarrow CrearIt(m.coordenadas)$ ▷ $\Theta(1)$
2: $caminos \leftarrow puntero(m.grl[latitud(c)][longitud(c)].caminos)$ ▷ $\Theta(1)$
3: $difFilas \leftarrow \#Columnas(m.grl) - \#Columnas(caminos)$
4: $AgregarColumnas(caminos, difColumnas, 0)$ ▷ $O(\#Columnas(m.grl) * difColumnas * \#Filas(m.grl))$
5: $difFilas \leftarrow \#Filas(m.grl) - \#Filas(caminos)$
6: $AgregarFilas(caminos, difFilas, 0)$ ▷ $O(difFilas * max\{\#Filas(m.grl), \#Col(m.grl)^2\})$
7: $it \leftarrow CrearIt(m.coordenadas)$ ▷ $\Theta(1)$
8: **while** HaySiguiente(it) **do**
9: **if** $caminos[latitud(Siguiente(it))][longitud(Siguiente(it))] == 0$ **then** ▷ $\Theta(1)$
10: **if** ExisteCamino($c, Siguiente(it), m$) **then** ▷ $O(3^{\#m.coordenadas})$
11: $caminos[latitud(Siguiente(it))][longitud(Siguiente(it))] \leftarrow 1$ ▷ $\Theta(1)$
12: **end if**
13: **end if**
14: Avanzar(it)
15: **end while**

Complejidad:
$O(max\{\#Columnas(m.grl)^2 * \#Filas(m.grl), \#Filas(m.grl)^2, \#m.coordenadas * (3^{\#m.coordenadas})\})$
Justificación:
Dada una coordenada c y un mapa, GrillaHayCamino actualiza m.grl[latitud(c)][longitud(c)].caminos. Primero se fija si m.grl[latitud(c)][longitud(c)].caminos tiene la misma cantidad de filas y columnas que m.grl[latitud(c)][longitud(c)], si no es asi le agregar filas y/o columnas de ceros hasta que se cumpla esa condicion. Luego, creo un iterador al conjunto de las coordenadas del mapa y para cada coordenada c' del mapa me fijo si esa posicion en m.grl[latitud(c)][longitud(c)].caminos es 1 o 0. Si es 1 quiere decir que ya habia un camino entre c y c', en ese caso sigo iterando sin hacer nada. Si es un 0 llamo a la funcion ExisteCamino, llamar a esta funcion tiene cuesta $(3^{\#m.coordenadas})$.
Si ExisteCamino devuelve true, le asigno un 1 a la posicion m.grl[latitud(c)][longitud(c)].caminos[latitud(c')][longitud(c')]
.
$O(\#Columnas(m.grl) * difColumnas * \#Filas(m.grl) + difFilas * max\{\#Filas(m.grl), \#Col(m.grl)^2\} + \#m.coordenadas * (O(3^{\#m.coordenadas}) + \#m.coordenadas)) \subseteq$
$O(max\{\#Columnas(m.grl)^2 * \#Filas(m.grl), \#Filas(m.grl) * max\{\#Filas(m.grl), \#Columnas(m.grl)^2\}, \#m.coordenadas * (3^{\#m.coordenadas})\}) =$
$O(max\{\#Columnas(m.grl)^2 * \#Filas(m.grl), \#Filas(m.grl)^2, \#m.coordenadas * (3^{\#m.coordenadas})\})$

iExisteCaminoAux(**in** c_1 : `coordenada`,**in** c_2 : `coordenada`,**in** cs : `conj(coordenada)`,**in** m : `mapa`) $\rightarrow$ res : *bool*

1: **if** $c_1 = c_2$ **then** ▷ $\Theta(1)$
2: $res \leftarrow true$
3: **else**
4: $res \leftarrow ExisteCaminoPorArriba(c_1, c_2, m) \vee$
5: $ExisteCaminoPorAbajo(c_1, c_2, m) \vee$
6: $ExisteCaminoPorDerecha(c_1, c_2, m) \vee$
7: $ExisteCaminoPorIzquierda(c_1, c_2, m)$
8: **end if**

Complejidad: $\mathcal{O}(3^{\#m.coordenadas})$
Justificación: El algoritmo es un algoritmo recursivo con un condicional. Se comporta de la siguiente forma: Dadas dos coordenadas c_1 y c_2 quiero ver si estan conectadas. Para esto, uso la grilla m.grl. En la primer llamada a la funcion, las posiciones en m.grl correspondientes a todas las coordenadas pertenecientes a m.coordenadas, tendran el booleano *disponible* indicando true y las demas posiciones lo tendran en false.
En el peor caso c_1 y c_2 son distintas, por lo tanto entro en *else* y llamo a las funcion Auxiliar $ExisteCaminoPorArriba(c_1, c_2, m)$. Esta función tambien contiene un condicional. En el peor de los casos el booleano *disponible* de la coordenada de arriba de c_1 indicará true, en este caso voy a llamar a la funcion ExisteCamino($coordenadaArriba(c_1), c_2, m$). Para no perder la información de que esa coordenada ya fue visitada, antes de volver a llamar a ExisteCaminoAux, voy a marcar la coordenada cambiando el booleano *disponible* a false. De esta forma, me aseguro que por cada coordenada en m.coordenadas voy a mirar si existe camino hacia c_2 una unica vez.
$ExisteCaminoPorAbajo(c_1, c_2, m), ExisteCaminoPorDerecha(c_1, c_2, m), ExisteCaminoPorIzquierda(c_1, c_2, m)$ se comportan de la misma forma. El algoritmo termina cuando c_1 y c_2 son iguales o no hayan coordenadas disponibles para visitar.
En el peor de los casos voy a aplicar la funcion ExisteCamino a todas las coordenadas de m.coordenadas. En cada llamado a ExisteCamino sabemos que ExisteCaminoPor$LaDireccionDesdeDondeMeMovi$ tendrá el booleano *disponible* en false y no voy a llamar a ExisteCamino desde esa direccion (a menos que sea el primer llamado, podemos despreciar este caso). Por lo tanto, en el peor caso dada una coordenada en m.coordenadas voy a llamar 3 veces a la funcion ExisteCaminoAux. Esto me da una complejidad total $\mathcal{O}(3^{\#m.coordenadas})$ para la funcion ExisteCaminoAux.

---

iExisteCaminoPorArriba(**in** c_1 : `coordenada`,**in** c_2 : `coordenada`,**in** m : `mapa`) $\rightarrow$ res : *bool*

1: **if** $latitud(c_1) < \#Filas(m.grl) \wedge m.grl[latitud(coordenadaArriba(c_1))][longitud(c_1)].disponible$ **then** ▷ $\Theta(1)$
2: $m.grl[latitud(coordenadaArriba(c_1))][longitud(c_1)].disponible \leftarrow false$ ▷ $\Theta(1)$
3: $res \leftarrow ExisteCaminoAux(coordenadaArriba(c_1), c_2, m)$
4: **else**
5: $res \leftarrow false$ ▷ $\Theta(1)$
6: **end if**

---

iExisteCaminoPorAbajo(**in** c_1 : `coordenada`,**in** c_2 : `coordenada` **in** m : `mapa`) $\rightarrow$ res : *bool*

1: **if** $latitud(c_1) > 0 \wedge m.grl[latitud(coordenadaAbajo(c_1))][longitud(c_1)].disponible?$ **then** ▷ $\Theta(1)$
2: $m.grl[latitud(coordenadaAbajo(c_1))][longitud(c_1)].disponible \leftarrow false$ ▷ $\Theta(1)$
3: $res \leftarrow ExisteCaminoAux(coordenadaAbajo(c_1), c_2, m)$
4: **else**
5: $res \leftarrow false$ ▷ $\Theta(1)$
6: **end if**

---

iExisteCaminoPorDerecha(**in** c_1 : `coordenada`,**in** c_2 : `coordenada`,**in** m : `mapa`) $\rightarrow$ res : $bool$

1: **if** $m.grl[latitud(c_1)][longitud(coordenadaALaDerecha(c_1))].disponible$ **then** ▷ $\Theta(1)$
2: $m.grl[latitud(c_1)][longitud(coordenadaALaDerecha(c_1))].disponible \leftarrow false$ ▷ $\Theta(1)$
3: $res \leftarrow ExisteCaminoAux(coordenadaALaDerecha(c_1), c_2, m)$
4: **else**
5: $res \leftarrow false$ ▷ $\Theta(1)$
6: **end if**

iExisteCaminoPorIzquierda(**in** c_1 : `coordenada`,**in** c_2 : `coordenada` **in** m : `mapa`) $\rightarrow$ res : $bool$

1: **if** $longitud(c_1) > 0 \wedge m.grl[latitud(c_1)][longitud(coordenadaALaIzquierda(c_1))].disponible$ **then** ▷ $\Theta(1)$
2: $m.grl[latitud(c_1)][longitud(coordenadaALaIzquierda(c_1))].disponible \leftarrow false$ ▷ $\Theta(1)$
3: $res \leftarrow ExisteCaminoAux(coordenadaALaIzquierda(c_1), c_2, m)$
4: **else**
5: $res \leftarrow false$ ▷ $\Theta(1)$
6: **end if**

6. Módulo Juego

Interfaz

usa: BOOL, NAT, POKEMON, JUGADOR, COORDENADA, VECTOR(JUGADOR), COLAPRIOR(JUGADOR), CONJUNTO LINEAL(STRING), DICCIONARIO STRING(STRING,σ)) MAPA, GRILLA(α).

se explica con: JUEGO

generos: `Juego`

Operaciones básicas

CREARJUEGO(**in** *map*: `Mapa`) $\rightarrow$ *res* : `juego`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ crearJuego(*map*)}
Complejidad: Θ(*largo***ancho*)
Descripción: Genera un juego con el parámetro map como Mapa.

AGREGARPOKEMON(**in** *poke*: `pokemon`, **in** *coord*: `coordenada`, **in/out** *juego*: `juego`)
Pre $\equiv$ {puedoAgregarPokemon(coord,juego)}
Post $\equiv$ {*ret* $=_{\text{obs}}$ agregarPokemon(poke,coord,juego)}
Complejidad: $\Theta(|P| + EC{*}log(EC))$
Descripción: Si se cumple la precondición, agrega poke al juego en la coordenada coord.

AGREGARJUGADOR(**in/out** *juego*: `juego`)
Pre $\equiv$ {true}
Post $\equiv$ {ret $=_{\text{obs}}$ agregarJugador(juego)}
Complejidad: $\Theta(|jugadores(juego)|)$
Descripción: Agrega un jugador al juego.

CONECTARSE(**in** *jug*: `jugador`, **in** *coord*: `coordenada`, **in/out** *juego*: `juego`)
Pre $\equiv$ {jug $\in$ jugadores(juego) $\wedge_{\text{L}}\neg$estaConectado(jug,juego) $\wedge$ posExistente(coord,mapa(juego)) }
Post $\equiv$ {res $=_{\text{obs}}$ conectarse(jug, coord, juego)}
Complejidad: $\Theta(log(EC))$
Descripción: Siempre y cuando se cumpla la precondición, se conecta el jugador.

DESCONECTARSE(**in** *jug*: `jugador`, **in/out** *juego*: `juego`)
Pre $\equiv$ {jug $\in$ jugadores(juego) $\wedge_{\text{L}}$ estaConectado(jug,juego)}
Post $\equiv$ {res $=_{\text{obs}}$ desconectarse(jug, juego)}
Complejidad: $\Theta(log(EC))$
Descripción: Siempre y cuando se cumpla la precondición, se desconecta el jugador.

MOVERSE(**in** *jug*: `jugador`, **in** *coord*: `coordenada`, **in/out** *juego*: `juego`)
Pre $\equiv$ {jug $\in$ jugadores(juego) $\wedge_{\text{L}}$ estaConectado(jug,juego) $\wedge$ posExistente(coord,mapa(juego))}
Post $\equiv$ {res $=_{\text{obs}}$ moverse(jug, coord, juego)}
Complejidad: $\Theta((PS + PC){*}|P| + log(EC))$
Descripción: Siempre y cuando se cumpla la precondición, se mueve el jugador jug a la coordenada cord.

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

JUGADORES(**in** *juego*: `juego`) $\rightarrow$ *res* : `itJugador`
Pre $\equiv$ {true}
Post $\equiv$ {ret $=_{\text{obs}}$ jugadores(juego)}
Complejidad: $\Theta(1)$
Descripción: Devuelve los jugadores del juego.

ESTACONECTADO(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : `bool`
Pre ≡ {jug ∈ jugadores(juego)}
Post ≡ {ret =$_{obs}$ agregarJugador(juego)}
Complejidad: Θ(1)
Descripción: Devuelve el estado en booleano del jugador.

SANCIONES(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : `Nat`
Pre ≡ {jug ∈ jugadores(juego)}
Post ≡ {ret =$_{obs}$ sanciones(jug,juego)}
Complejidad: Θ(1)
Descripción: Devuelve el número de sanciones del jugador.

POSICION(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : `coordenada`
Pre ≡ {jug ∈ jugadores(juego) $\wedge_{L}$ estaConectado(jug,juego)}
Post ≡ {ret =$_{obs}$ posicion(jug, juego)}
Complejidad: Θ(1)
Descripción: Devuelve la posición del jugador.

POKEMONS(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : $vector(\langle pokemon, nat \rangle)$
Pre ≡ {jug ∈ jugadores(juego)}
Post ≡ {ret =$_{obs}$ pokemons(juego)}
Complejidad: Θ(1)
Descripción: Devuelve los pokémones capturados por el jugador.

EXPULSADOS(**in** *juego*: `juego`) → *res* : `vector(jugador)`
Pre ≡ {true}
Post ≡ {ret =$_{obs}$ expulsados(juego)}
Complejidad: Θ(J)
Descripción: Devuelve los jugadores expulsados.

POSCONPOKEMONS(**in** *juego*: `juego`) → *res* : `vector(coordenadas)`
Pre ≡ {true}
Post ≡ {ret =$_{obs}$ posConPokemons(juego)}
Complejidad: Θ(1)
Descripción: Devuelve las posiciones con pokémons.

POKEMONENPOS(**in** *coord*: `coordenada`, **in** *juego*: `juego`) → *res* : `pokemon`
Pre ≡ {coord ∈ posConPokemons(juego)}
Post ≡ {ret =$_{obs}$ pokemonEnPos(coord, juego)}
Complejidad: Θ(1)
Descripción: Devuelve el pokémon que se encuentra en la coordenada coord.

CANTMOVIMIENTOSPARACAPTURA(**in** *coord*: `coordenada`, **in** *juego*: `juego`) → *res* : `Nat`
Pre ≡ {coord ∈ posConPokemons(juego)}
Post ≡ {ret =$_{obs}$ cantMovimientosParaCaptura(coord, juego)}
Complejidad: Θ(1)
Descripción: Devuelve la cantidad de Movimientos que faltan para capturar el pokémon que se encuentra en la coordenada coord.

PROXID(**in** *juego*: `juego`) → *res* : `jugador`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ ProxId(juego)}
Complejidad: Θ(1)
Descripción: Devuelve el proximo jugador que se creará en el juego.

JUGADORESCONECTADOS(**in** *juego*: `juego`) → *res* : `lista(jugador)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ jugadoresConectados(juego)}
Complejidad: Θ(J)
Descripción: Devuelve los jugadores que se encuentran conectados

SOLOLOSCONECTADOS(**in** *jugadores*: `conj(jugador)`,**in** *juego*: `juego`) → *res* : `lista(jugadores)`
Pre ≡ {jugadores ⊆ jugadores(juego)}
Post ≡ {res =$_{obs}$ soloLosConectados(jugadores, juego)}
Complejidad: Θ(#*jugadores*)
Descripción: Siempre y cuando se cumpla la precondición, devuelve los jugadores que se encuentran conectados y en jugadores.

PUEDOAGREGARPOKEMON(**in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ puedoAgregarPokemon(coord, juego)}
Complejidad: Θ(1)
Descripción: Devuelve true si se puede agregar un pokémon en la coordenada coord.

HAYPOKEMONENTERRITORIO(**in** *coord*: `coordenada`, **in** *conjCoord*: `conj(coord)`, **in** *juego*: `juego`) → *res* : `conj(bool)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ hayPokemonEnTerritorio(coord, conjCoord, juego)}
Complejidad: Θ(#*conjCoord*)
Descripción: Devuelve el conjunto de booleanos de cada coordenada de conjCoord comparada con coord

DEBESANCIONARSE(**in** *jug*: `jugadore`, **in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : `bool`
Pre ≡ {jug ∈ jugadores(juego)}
Post ≡ {res =$_{obs}$ debeSancionarse(jug, coord, juego)}
Complejidad: Θ(1)
Descripción: Siempre y cuando se cumpla la precondición, devuelve true si se debe sancionar al jugador.

DEBEEXPULSARSE(**in** *jug*: `jugadore`, **in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : `bool`
Pre ≡ {jug ∈ jugadores(juego)}
Post ≡ {res =$_{obs}$ debeExpulsarse(jug, coord, juego)}
Complejidad: Θ(1)
Descripción: Siempre y cuando se cumpla la precondición, devuelve true si se debe expulsar al jugador.

HAYPOKEMONCERCANO(**in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : `coordenada`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ hayPokemonCercano(coord, juego)}
Complejidad: Θ(1)

POSPOKEMONCERCANO(**in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : `coordenada`
Pre ≡ {hayPokemonCercano(coord,juego)}
Post ≡ {res =$_{obs}$ posPokemonCercano(coord, juego)}
Complejidad: Θ(1)
Descripción: Siempre y cuando se cumpla la precondición, devuelve la coordenada del pokemon que se encuentra en radio de la coordenada

ENTRENADOREPOSIBLES(**in** *coord*: `coordenada`,**in** *jugadores*: `conj(jugador)`,**in** *juego*: `juego`) → *res* : `conj(jugador)`
Pre ≡ {hayPokemonCercano(coord,juego) ∧ jugadores ⊆ jugadores(juego)}
Post ≡ {res =$_{obs}$ entrenadoresPosibles(coord, jugadores, juego)}
Complejidad: Θ(#*jugadores*)
Descripción: Siempre y cuando se cumpla la precondición, devuelve los jugadores posibles a capturar el pokémon que se encuentra en la coordenada coord.

POSDEPOKÉMONSACAPTURAR(**in** *coord*: `coordenada`, **in** *conjCoord*: `conj(coordenada)`, **in** *juego*: `juego`) → *res* : `conj(coord)`
Pre ≡ {conjCoord ⊆ posConPokemons(juego)}
Post ≡ {res =$_{obs}$ posDePokémonsACapturar(coord, juego)}
Complejidad: Θ(#*conjCoord*)
Descripción: Siempre y cuando se cumpla la precondición, devuelve los pokemones a capturar que se encuentran en conjCoord.

SECAPTURO(**in** *coord1*: `coordenada`, **in** *coord2*: `coordenada`,**in** *juego*: `juego`) → *res* : `bool`
Pre ≡ {coord1 ∈ posConPokemons(juego)}

Post ≡ {res =$_{obs}$ seCapturo(coord1, coord2, juego)}
Complejidad: Θ(1)
Descripción: Siempre y cuando se cumpla la precondición, devuelve true si se capturó el pokemon que se encontraba en la coordenada coord1.

INDICERAREZA(**in** *poke*: `pokemon`,**in** *juego*: `juego`) → *res* : `nat`
Pre ≡ {poke ∈ todosLosPokemons(juego)}
Post ≡ {res =$_{obs}$ indiceRareza(poke, juego)}
Complejidad: Θ(|*P*|)
Descripción: Siempre y cuando se cumpla la precondición, devuelve el indice de rareza del pokémon poke.

CANTPOKÉMONSTOTALES(**in** *juego*: `juego`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ cantPokémonsTotales(juego)}
Complejidad: Θ(1)
Descripción: Devuelve la cantidad de pokemones totales del juego.

TODOSLOSPOKÉMONS(**in** *juego*: `juego`)) → *res* : `lista(pokemons)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ todosLosPokémons(juego)}
Complejidad: Θ((#*juego.jugNoExpulsados* ∗ *PCjug*) + (#*juego.posConPokemones* ∗ *PS*))
Descripción: Siempre y cuando se cumpla la precondición, devuelve todos los pokemones del juego (Salvejes y capturados).

POKEMONSSALVAJES(**in** *conjCoord*: `conj(coordenada)`,**in** *juego*: `juego`) → *res* : `lista(pokemon)`
Pre ≡ {conjCoord ⊆ posConPokemons(juego)}
Post ≡ {res =$_{obs}$ pokemonsSalvajes(conjCoord, juego)}
Complejidad: Θ(#*conjCoord* ∗ *PS*)
Descripción: Siempre y cuando se cumpla la precondición, devuelve los pokemones salvajes que se encuentran las coordenadas de conjCoord.

POKEMONSCAPTURADOS(**in** *jugadores*: `conj(jugador)`,**in** *juego*: `juego`) → *res* : `lista(pokemon)`
Pre ≡ {jugadores ⊆ jugadores(juego)}
Post ≡ {res =$_{obs}$ pokemonsCapturados(jugadores, juego)}
Complejidad: Θ(#*jugadores*∗*PCjug*)
Descripción: Siempre y cuando se cumpla la precondición, devuelve todos los pokemones que capturaron los jugadores de jugadores.

BUSCARPOKÉMONSCERCANOS(**in** *coord*: `coordenada`, **in** *conjCoord*: `conj(coordenada)`, **in** *juego*: `juego`) → *res* : `conj(cooord)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ buscarPokémonsCercanos(coord, conjCoord, juego)}
Complejidad: Θ(#*conjCoord*)
Descripción: Devuelve los pokemones cercanos a la coordenada coord que se encuentran en el conjCoord.

CANTMISMAESPECIE(**in** *poke*: `pokemon`,**in** *pokemones*: `conj(pokemon)`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ cantMismaEspecie(poke, pokemones)}
Complejidad: Θ(#*pokemones*)
Descripción: Siempre y cuando se cumpla la precondición, devuelve los poke de la misma especie que se encuentran en pokemones.

Representación

Representación del juego

`juego` **se representa con** `jgo`

donde `jgo` es `tupla`(
mapa: `Mapa` , *jugNoExpulsados*: `conj(jugador)` , *jugadores*: `vector(jugadorStruct)`
, *posConPokemones*: `conj(coordenada)`
, *pokemonesSalvajes*: `diccString(pokemon, nat)`
, *pokemonesCapturados*: `diccString(pokemon, nat)`
, *posiciones*: `arreglo( arreglo( posStruct ) )`
, *cantPokemonTotal*: `nat`)

donde `pokemonCapturado` es `tupla`(*pokemon*: `pokemon` , *cantidad*: `nat`)

donde `posStruct` es `tupla`(*jugadores*: `colaPrior(tupla(nat,jugador))`
, *pokemonACapturar*: `puntero(pokemonACapturar)`)

donde `pokemonACapturar` es `tupla`(*pokemon*: `pokemon` , *movAfuera*: `nat`
, *itCoord*: `itConj(coordenada)`
, *jugACapturalo*: `colaPrior(tupla(nat,jugador))`
)

donde `jugadorStruct` es `tupla`(*conectado*: `bool` , *sanciones*: `Nat` , *pos*: `coordenada`
, *pokemones*: `lista(tupla(pokemon,nat))`
, *itPokemones*: `diccString(pokemon,itLista(tupla(pokemon, Nat))))`
, *pokemonesTotales*: `nat`
, *itJugNoExpulsados*: `itConj(jugador)`
, *itPosJug*: `itColaPrior(tupla(nat,jugador))`
, *itCapturarPoke*: `itColaPrior(tupla(nat,jugador))`
)

)

El invariante de representación puede escribirse como:

1. Todos los jugadores no expulsados fueron agregados alguna vez al juego.
2. Todos los jugadores en el conjunto de j.jugNoExpulsados son los jugadores que no están expulsados, en otras palabras, tienen menos de 5 infracciones.
3. Todos las coordenadas en el conjunto de j.posConPokemons son los pokemones que fueron agregados al juego y todavía no fueron capturados.
4. Por cada jugador agregado al juego, el vector de jugadores se agranda en una posición, dejando el índice del vector como el número del jugador.
5. Por cada coordenada en el mapa, se genera una posición en el juego. La misma se representa una grilla (o lista de listas) de posStruct.
6. Cada posStruct tiene una cola de prioridad de jugadores que se encuentran en la posicion y un puntero hacia la representacion de un pokemon posicionado (el mismo puede ser NULL, indicando que no se encuentra ningún pokémon en dicha posición).
7. Por cada coordenada en j.posConPokemon la misma tiene una posicion en j.posiciones donde el pokemon agregado se representa con pokemonACapturar lo cual contine el mismo pokemon como: pokemon, los movimientos afuera del radio del pokemon con: movAfuera, el iterador apuntando a su representante en j.posConPokemons con: itCoord y una cola de prioridad de los jugadores que se encuentran posicionados en el radio del mismo para capturarlo con: jugACapturarlo.
8. Cada posicion del vector j.jugadores contiene un jugadorStruct el cual tiene: un estado que re representa como conectado, una cantidad de sanciones: sanciones, un diccionario de pokemones capturados por el jugador:

pokemones, una cantidad de pokemones totales capturados: pokemonesTotales, un iterador apuntando a su representante en j.jugNoExpulsados: itjugNoExpulsados, un iterador apuntando a su representante en la cola de prioridad que se encuentra en la posicion del jugador: itPosJug, un iterador apuntando a su representante en la cola de prioridad que se encuentra en el posible pokemon a capturar: itCapturarPoke.

9. Por cada pokemon capturado que se encuentra en el diccionario dentro de jugadorStruct.pokemones, tiene su equivalencia en los pokemones capturados de j.pokemonesCapturados, es decir que la sumatoria de todos los pokemones capturados dentro de cada jugadorStruct.pokemones es la misma que resulta de j.pokemonesCapturados.

Rep : jgo $\longrightarrow$ bool
Rep(j) $\equiv$ true $\Longleftrightarrow$ (1) $\wedge_{\text{L}}$ (2) $\wedge$ (3) $\wedge_{\text{L}}$ (4) $\wedge$ (5) $\wedge_{\text{L}}$ (6) $\wedge$ (7) $\wedge$ (8) $\wedge$ (9)

Abs : jgo j $\longrightarrow$ juego {Rep(j)}
Abs(j) $=_{\text{obs}}$ pG: juego |
mapa(pG) $=_{\text{obs}}$ j.mapa $\wedge$
jugadores(pG) $=_{\text{obs}}$ j.jugNoExpulsados $\wedge$
posConPokemons(pG) $=_{\text{obs}}$ j.posConPokemones $\wedge_{\text{L}}$
$(\forall c : coordenada)(c \in$ j.posConPokemones $\Rightarrow_{\text{L}}$
(pokemonEnPos(c, pG) $=_{\text{obs}}$ (j.posiciones[c.latitud][c.longitud].pokemonACapturar)$\rightarrow$pokemon
$\wedge$
cantMovimientosParaCaptura(c, pG) $=_{\text{obs}}$
(j.posiciones[c.latitud][c.longitud].pokemonACapturar)$\rightarrow$movAfuera)) $\wedge$
$(\forall e : jugador)(e \in j$.jugNoExpulsados $\Rightarrow_{\text{L}}$
(estaConectado(e, pG) $\Leftrightarrow$ j.jugadores[e].conectado $\wedge$
sanciones(e, pG) $=_{\text{obs}}$ j.jugadores[e].sanciones $\wedge$
posicion(e, pG) $=_{\text{obs}}$ j.jugadores[e].pos $\wedge$
pokemons(e, pG) $=_{\text{obs}}$ diccAMulticonj(j.jugadores[e].pokemones)))

diccAMulticonj : dicc(pokemon, nat) d $\longrightarrow$ multiConj(pokemon)
diccAMulticonj(d) $\equiv$ **if** vacío?(claves(d)) **then**
$\emptyset$
else
tuplaAMulticonj($\langle$ DameUno(claves(d)), obtener(DameUno(claves(d))) $\rangle$)
$\cup$ diccAMulticonj(borrar(DameUno(claves(d)), d))
fi

tuplaAMulticonj : tupla(pokemon, nat) t $\longrightarrow$ multiConj(pokemon)
tuplaAMulticonj(t) $\equiv$ **if** $\pi_2(t) = 0$ **then** $\emptyset$ **else** Ag($\pi_1(t)$, tuplaAMulticonj($\langle \pi_1(t), \pi_2(t) - 1 \rangle$)) **fi**

Algoritmos

Algoritmos del módulo

---

crearJuego(**in** map: mapa) $\rightarrow$ res : $juego$

1: $jugNoExpulsados \leftarrow vacio()$ ▷ $\Theta(1)$
2: $jugadores \leftarrow vacia()$ ▷ $\Theta(1)$
3: $posConPokemones \leftarrow vacio()$ ▷ $\Theta(1)$
4: $pokemonesSalvajes \leftarrow CrearDicc()$ ▷ $\Theta(1)$
5: $pokemonesCapturados \leftarrow CrearDicc()$ ▷ $\Theta(1)$
6: $posiciones \leftarrow CrearPosiciones(map)$ ▷ $\Theta(\#coordenadas + largo^{*}ancho)$
7: $cantPokemonTotal \leftarrow 0$
8: $res \leftarrow \langle map, jugNoExpulsados, jugadores, posConPokemones, pokemonesSalvajes, pokemonesCapturados, posiciones, cantPokemonTotal \rangle$ ▷ $\Theta(1)$
Complejidad: $\Theta(\#coordenadas + largo^{*}ancho)$, donde largo es la coordenada con la longitud mas grande del mapa y ancho es la coordenada con la latitud mas grande del mapa.
Justificación: $\Theta(1) + \Theta(1) + \Theta(1) * \Theta(\#coordenadas + largo^{*}ancho) + \Theta(1) = \Theta(\#coordenadas + largo^{*}ancho)$.

---

agregarPokemon(**in** *poke*: `pokemon`, **in** *coord*: `coordenada`,**in/out** *juego*: `juego`)

1: $nuevaCantPokemonTotal \leftarrow juego.cantPokemonTotal + 1$ ▷ $\Theta(1)$
2: $juego.cantPokemonTotal \leftarrow nuevaCantPokemonTotal$ ▷ $\Theta(1)$
3: $itConjPoke \leftarrow agregarRapido(juego.posConPokemones, coord)$ ▷ $\Theta(1)$
4: $posicionarPokemon(poke, coord, itConjCord, juego)$ ▷ $\Theta(EC^*log(EC))$
5: **if** definido(juego.pokemonesSalvajes, poke) **then** ▷ $\Theta(|P|)$
6: $cantidad \leftarrow obtener(juego.pokemonesSalvajes, poke)$ ▷ $\Theta(|P|)$
7: $definir(juego.pokemonesSalvajes, poke, cantidad + 1)$ ▷ $\Theta(|P|)$
8: **else**
9: $definir(juego.pokemonesSalvajes, poke, 1)$ ▷ $\Theta(|P|)$
10: **end if**
11: $res \leftarrow juego$ ▷ $\Theta(1)$

Complejidad: $\Theta(EC^*log(EC) + |P|)$, donde EC es la máxima cantidad de jugadores esperando para atrapar un pokémon y $|P|$ es el nombre más largo para un pokémon en el juego.
Justificación: $\Theta(EC^*log(EC)) + \Theta(|P|) + \Theta(1) = \Theta(EC^*log(EC) + |P|)$

---

agregarJugador(**in/out** *juego*: `juego`)

1: $jug \leftarrow Longitud(juego.jugadores)$ ▷ $\Theta(1)$
2: $AgregarAtras(juego.jugadores, jug)$ ▷ O(J)
3: $itNoExpulsados \leftarrow agregarRapido(juego.jugNoExpulsados, i)$ ▷ $\Theta(1)$
4: $jugadores[i] \leftarrow nuevaJugadorStruct(itNoExpulsados)$ ▷ $\Theta(1)$

Complejidad: O(J) , donde J es la cantidad total de jugadores que fueron agregados al juego.
Justificación: $\Theta(1)$+O(J) + $\Theta(1)$ + $\Theta(1)$ = O(J)

---

conectarse(**in** *jug*: `jugador`, **in** *coord*: `coordenada`, **in/out** *juego*: `juego`)

1: $juego.jugadores[jug].conectado \leftarrow true$ ▷ $\Theta(1)$
2: $juego.jugadores[jug].pos \leftarrow coord$ ▷ $\Theta(1)$
3: $posStruct \leftarrow \&juego.posiciones[longitud(coord)][latitud(coord)]$ ▷ $\Theta(1)$
4: $tupla \leftarrow \langle juego.jugadores[jug].pokemonesTotales, jug\rangle$ ▷ $\Theta(1)$
5: $juego.jugadores[jug].itPosJug \leftarrow encolar(posStruct.jugadores, tupla)$ ▷ $\Theta(log(\#tupla.jugadores))$
6: **if** $hayPokemonCercano(coord, juego)$ **then** ▷ $\Theta(1)$
7: $coordPoke \leftarrow posPokemonCercano(coord, juego)$ ▷ $\Theta(1)$
8: $pokeACapturar \leftarrow \&juego.posiciones[longitud(coordPoke)][latitud(coordPoke)]$ ▷ $\Theta(1)$
9: $juego.jugadores[jug].itCapturarPoke \leftarrow encolar(pokeACapturar \rightarrow jugACapturarlo, tupla)$ ▷ $\Theta(log(EC))$
10: **end if**

Complejidad: $\Theta(log(EC))$, donde EC es la máxima cantidad de jugadores esperando para atrapar un pokémon (en este caso #tupla.jugadores).
Justificación: $\Theta(log(EC))$ + $\Theta(log(\#tupla.jugadores))$ + $\Theta(1)$ $*5$ = $\Theta(log(EC))$, dado que EC $\geq$ #tupla.jugadores y puedo acotarlo por EC.

---

desconectase(**in** *jug*: `jugador`,**in/out** *juego*: `juego`)

1: $juego.jugadores[jug].conectado \leftarrow false$ ▷ $\Theta(1)$
2: $eliminarSiguiente(juego.jugadores[jug].itPosJug)$ ▷ $\Theta(log(\#tupla.jugadores))$
3: **if** $hayPokemonCercano(juego.jugadores[jug].pos, juego)$ **then** ▷ $\Theta(1)$
4: $eliminarSiguiente(juego.jugadores[jug].itCapturarPoke)$ ▷ $\Theta(log(EC))$
5: **end if**

Complejidad: $\Theta(log(EC))$, donde EC es la máxima cantidad de jugadores esperando para atrapar un pokémon (en este caso #tupla.jugadores).
Justificación: $\Theta(log(EC))$ + $\Theta(log(\#tupla.jugadores))$ + $\Theta(1)$ = $\Theta(log(EC))$, dado que EC $\geq$ #tupla.jugadores y puedo acotarlo por EC.

moverse(**in** *jug*: `jugador`, **in** *coord*: `coordenada`, **in/out** *juego*: `juego`)

1: $jugador \leftarrow juego.jugadores[jug]$ ▷ $\Theta(1)$
2: $posAnterior \leftarrow jugador.pos$ ▷ $\Theta(1)$
3: **if** ¬existeCamino(coord, posAnterior, juego.mapa) ∨ distEuclidea(coord, posAnterior) ≥ 100 **then** ▷ $\Theta(1)$
4: $jugador.sanciones \leftarrow jugador.sanciones + 1$ ▷ $\Theta(1)$
5: **end if**
6: $eliminarSiguiente(jugador.itPosJug)$ ▷ $\Theta(log(EC))$
7: **if** $hayPokemonCercano(posAnterior, juego) \wedge \neg hayPokemonCercano(coord, juego)$ **then** ▷ $\Theta(1)$
8: $eliminarSiguiente(jugador.itCapturarPoke)$ ▷ $\Theta(log(EC))$
9: **end if**
10: **if** jugador.sanciones < 5 **then** ▷ $\Theta(1)$
11: $jugador.pos \leftarrow coord$ ▷ $\Theta(1)$
12: $posStruct \leftarrow \&juego.posiciones[longitud(coord)][latitud(coord)]$ ▷ $\Theta(1)$
13: $tupla \leftarrow \langle jugador.pokemonesTotales, jug \rangle$ ▷ $\Theta(1)$
14: $jugador.itPosJug \leftarrow encolar(posStruct.jugadores, tupla)$ ▷ $\Theta(log(EC))$
15: **if** $hayPokemonCercano(coord, juego) \wedge \neg hayPokemonCercano(posAnterior, juego)$ **then** ▷ $\Theta(1)$
16: $coordPoke \leftarrow posPokemonCercano(coord, juego)$ ▷ $\Theta(1)$
17: $pokeACapturar \leftarrow \&juego.posiciones[longitud(coordPoke)][latitud(coordPoke)]$ ▷ $\Theta(1)$
18: $itCapturarPoke \leftarrow encolar(pokeACapturar \rightarrow jugACapturarlo, tupla)$ ▷ $\Theta(log(EC))$
19: $juego.jugadores[jug].itCapturarPoke \leftarrow itCapturarPoke$ ▷ $\Theta(1)$
20: $pokeACapturar \rightarrow movAfuera \leftarrow 0$ ▷ $\Theta(1)$
21: **end if**
22: $itPosConPoke \leftarrow CrearIt(juego.posConPokemones)$ ▷ $\Theta(1)$
23: **while** HaySiguiente(itPosConPoke) **do** ▷ $\Theta((PS + PC)*|P|)$
24: $posConPoke \leftarrow siguiente(itPosConPoke)$ ▷ $\Theta(1)$
25: **if** distEuclidea(posConPoke,coord) > 4 **then** ▷ $\Theta(1)$
26: $lat \leftarrow latitud(posConPoke)$ ▷ $\Theta(1)$
27: $long \leftarrow longitud(posConPoke)$ ▷ $\Theta(1)$
28: $pokemonACapturar \leftarrow juego.posiciones[lat][long].pokemonACapturar$ ▷ $\Theta(1)$
29: $pokemonACapturar \rightarrow movAfuera \leftarrow pokemonACapturar \rightarrow movAfuera + 1$ ▷ $\Theta(1)$
30: **end if**
31: **if** $pokemonACapturar \rightarrow$ movAfuera = 10 **then**
32: $jugACapt \leftarrow proximo(pokemonACapturar \rightarrow jugACapturarlo)$ ▷ $\Theta(1)$
33: $capturarPokemon(\pi_2(jugACapt), pokemonACapturar \rightarrow pokemon, juego)$ ▷ $\Theta(|P|)$
34: $eliminarSiguiente(pokemonACapturar \rightarrow itCoord)$ ▷ $\Theta(1)$
35: $juego.posiciones[lat][long].pokemonACapturar \leftarrow NULL$ ▷ $\Theta(1)$
36: **end if**
37: $avanzar(itPosConPoke)$ ▷ $\Theta(1)$
38: **end while**
39: **else**
40: $eliminarSiguiente(jugador.itJugNoExpulsados)$ ▷ $\Theta(1)$
41: $pokemones \leftarrow jugador.pokemones$ ▷ $\Theta(1)$
42: $i \leftarrow 0$ ▷ $\Theta(1)$
43: **while** $i < longitud(pokemones)$ **do** ▷ $O(PC * |P|)$
44: $poke \leftarrow \pi_1(pokemones[i])$ ▷ $\Theta(1)$
45: $cantPoke \leftarrow \pi_2(pokemones[i])$ ▷ $\Theta(1)$
46: $nuevaCantPokemonTotal \leftarrow juego.cantPokemonTotal - cantPoke$ ▷ $\Theta(1)$
47: $juego.cantPokemonTotal \leftarrow nuevaCantPokemonTotal$
48: $nuevaCantPoke \leftarrow obtener(juego.pokemonesCapturados, poke) - cantPoke$ ▷ $\Theta(|P|)$
49: $definir(juego.pokemonesCapturados, poke, nuevaCantPoke)$ ▷ $\Theta(|P|)$
50: **end while**
51: **end if**
52: $juego.jugadores[jug] \leftarrow jugador$ ▷ $\Theta(1)$

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

mapa(**in** *juego*: `juego`) → *res* : *mapa*

1: *res* ← *juego.mapa* ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

jugadores(**in** *j* : `juego`) → *res* : *itJugador*

1: *res* ← *CrearIt*(*j.jugNoExpulsados*) ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

estaConectado(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : *bool*

1: *res* ← *juego.jugadores*[*jug*].*conectado* ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

sanciones(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : *Nat*

1: *res* ← *juego.jugadores*[*jug*].*sanciones* ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

posicion(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : *coordenada*

1: *res* ← *juego.jugadores*[*jug*].*pos* ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

pokemons(**in** *jug*: `jugador`, **in** *juego*: `juego`) → *res* : *itDicc*(*pokemon*, *nat*)

1: *res* ← *CrearIt*(*juego.jugadores*[*jug*].*pokemones*) ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

expulsados(**in** *juego*: `juego` → *res* : *vector*(*jugador*))

res ← *vacia*() ▷ $\Theta(1)$
i ← 0 ▷ $\Theta(1)$
while *i* < *tamaño*(*juego.jugadores*) **do** ▷ $\Theta(\#jugadores)$
  if (juego.jugadores[i]).sanciones = 5 **then** ▷ $\Theta(1)$
    agregarAtras(res, i) ▷ $\Theta(1)$
  end if
end while

Complejidad: $\Theta(J)$, donde J es la cantidad total de jugadores que fueron agregados al juego.

---

posConPokemons(**in** *juego*: `juego`) → *res* : *conj*(*coord*)

1: *res* ← *juego.posConPokemones* ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

pokemonEnPos(**in** *coord*: `coordenada`, **in** *juego*: `juego`) → *res* : *pokemon*

1: $res \leftarrow juego.posiciones[longitud(coord)][latitud(coord)].PokemonACapturar \rightarrow pokemon$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

cantMovimientosParaCaptura(**in** *coord*: `coordenada`, **in** *juego*: `juego`) → *res* : *nat*

$res \leftarrow juego.posiciones[longitud(coord)][latitud(coord)].pokemonACapturar \rightarrow movAfuera$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

proxId(**in** *juego*: `juego`) → *res* : *jugador*

1: $res \leftarrow tamaño(jugadores)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

jugadoresConectados(**in** *juego*: `juego`) → *res* : *lista*(*jugador*)

1: $lista \leftarrow vacia()$ ▷ $\Theta(1)$
2: $i \leftarrow 0$ ▷ $\Theta(1)$
3: **while** $tamaño(juego.jugadores)¿i$ **do** ▷ $\Theta(J)$
4: **if** juego.jugadores[i].conectado **then** ▷ $\Theta(1)$
5: $agregarAtras(lista, i)$ ▷ $\Theta(1)$
6: **end if**
7: **end while**
8: $res \leftarrow lista$ ▷ $\Theta(1)$

Complejidad: $\Theta(J)$, donde J es la cantidad de jugadores que fueron agregados al juego.

---

soloLosConectados(**in** *jugadores*: `conj(jugador)`,**in** *juego*: `juego`) → *res* : *lista*(*jugador*)

1: $lista \leftarrow vacia()$ ▷ $\Theta(1)$
2: $it \leftarrow creatIt(jugadores)$ ▷ $\Theta(1)$
3: **while** haySiguiente(it) **do** ▷ $\Theta(\#jugadores)$
4: $i \leftarrow siguiente(it)$ ▷ $\Theta(1)$
5: **if** juego.jugadores[i].conectado **then** ▷ $\Theta(1)$
6: $agregarAtras(lista, i)$ ▷ $\Theta(1)$
7: **end if**
8: $avanzar(it)$ ▷ $\Theta(1)$
9: **end while**
10: $res \leftarrow lista$ ▷ $\Theta(1)$

Complejidad: $\Theta(\#jugadores)$

---

puedoAgregarPokemon(**in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : *bool*

1: $res \leftarrow \neg(hayPokemonCercano(coord, juego))$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

hayPokemonEnTerritorio(**in** *coord*: `coordenada`, **in** *conjCoord*: `conj(coord)`, **in** *juego*: `juego`) → *res* : *conj*(*bool*)

1: $conj \leftarrow vacio()$ ▷ $\Theta(1)$
2: $it \leftarrow creatIt(conjCoord)$ ▷ $\Theta(1)$
3: **while** haySiguiente(it) **do** ▷ $\Theta(\#conjCoord)$
4: $pCoord \leftarrow siguiente(it)$ ▷ $\Theta(1)$
5: $agregarRapido(conj, (distEuclidea(coord, pCoord) \leq 4))$ ▷ $\Theta(1)$
6: $avanzar(it)$ ▷ $\Theta(1)$
7: **end while**
8: $res \leftarrow conj$ ▷ $\Theta(1)$

Complejidad: $\Theta(\#conjCoord)$

---

idebeSancionarse(**in** j : `jugador` **in** c : `coordenada`, **in** *juego*: `juego`) → *res* : *bool*

$jugador \leftarrow juego.jugadores[j]$
$res \leftarrow \neg(HayCamino(jugador.pos, c, juego.mapa)) \vee distanciaEuclidea(jugador.pos, c, juego.mapa) > 100$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

idebeExpulsarse(**in** j : `jugador` **in** c : `coordenada`, **in** *juego*: `juego`) → *res* : *bool*

$jugador \leftarrow juego.jugadores[j]$
$res \leftarrow debeSancionarse(j, c, juego) \wedge jugador.sanciones \geq 4$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

hayPokemonCercano(**in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : *coordenada*

1: $ret \leftarrow false$ ▷ $\Theta(1)$
2: $lat \leftarrow latitud(coord)$ ▷ $\Theta(1)$
3: $long \leftarrow longitud(coord)$ ▷ $\Theta(1)$
4: $lat\_desde \leftarrow lat - 2$ ▷ $\Theta(1)$
5: **while** $lat\_desde \leq lat + 2$ **do** ▷ $\Theta(4 * 4)$
6: $long\_desde \leftarrow lat - 2$ ▷ $\Theta(1)$
7: **while** $long\_desde \leq lat + 2$ **do** ▷ $\Theta(4)$
8: **if** distEuclidea($\langle lat\_desde, long\_desde \rangle$,coord) ≤ 4 **then** ▷ $\Theta(1)$
9: $tupla \leftarrow \&juego.posiciones[longitud(coord)][latitud(coord)]$ ▷ $\Theta(1)$
10: **if** $tupla.pokemonACapturar \neq NULL$ **then** ▷ $\Theta(1)$
11: $ret \leftarrow true$ ▷ $\Theta(1)$
12: **end if**
13: **end if**
14: $long\_desde \leftarrow long\_desde + 1$ ▷ $\Theta(1)$
15: **end while**
16: $lat\_desde \leftarrow lat\_desde + 1$ ▷ $\Theta(1)$
17: **end while**
18: $res \leftarrow ret$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

posPokemonCercano(**in** *coord*: `coordenada`,**in** *juego*: `juego`) → *res* : *coordenada*

```
1:  lat ← latitud(coord)                                                    ▷ Θ(1)
2:  long ← longitud(coord)                                                  ▷ Θ(1)
3:  lat_desde ← lat − 2                                                     ▷ Θ(1)
4:  while lat_desde ≤ lat + 2 do                                            ▷ Θ(4 ∗ 4)
5:      long_desde ← lat − 2                                                ▷ Θ(1)
6:      while long_desde ≤ lat + 2 do                                       ▷ Θ(4)
7:          retCoord ← ⟨lat_desde, long_desde⟩                              ▷ Θ(1)
8:          if distEuclidea(retCoord, coord) ≤ 4 then                       ▷ Θ(1)
9:              tupla ← &juego.posiciones[longitud(coord)][latitud(coord)]  ▷ Θ(1)
10:             if tupla.pokemonACapturar ≠ NULL then                       ▷ Θ(1)
11:                 res ← retCoord                                          ▷ Θ(1)
12:             end if
13:         end if
14:         long_desde ← long_desde + 1                                     ▷ Θ(1)
15:     end while
16:     lat_desde ← lat_desde + 1                                           ▷ Θ(1)
17: end while
```

Complejidad: Θ(1)

---

entrenadoresPosibles(**in** *coord*: `coordenada`,**in** *jugadores*: `conj(jugador)`,**in** *juego*: `juego`) → *res* : *conj*(*jugador*)

```
1:  conj ← vacio()                                                          ▷ Θ(1)
2:  it ← creatIt(jugadores)                                                 ▷ Θ(1)
3:  while haySiguiente(it) do                                               ▷ Θ(#jugadores)
4:      jug ← siguiente(it)                                                 ▷ Θ(1)
5:      jCoord ← juego.jugadores[jug].pos                                   ▷ Θ(1)
6:      if distEuclidea(coord, jCoord) ≤ 4 then                             ▷ Θ(1)
7:          agregarRapido(conj, jug)                                        ▷ Θ(1)
8:      end if
9:      avanzar(it)                                                         ▷ Θ(1)
10: end while
11: res ← conj                                                              ▷ Θ(1)
```

Complejidad: Θ(#*jugadores*)

---

posDePokémonsACapturar(**in** *coord*: `coordenada`, **in** *conjCoord*: `conj(coordenada)`, **in** *juego*: `juego`) → *res* : *conj*(*coordenada*)

```
1:  conj ← vacio()                                                          ▷ Θ(1)
2:  it ← creatIt(conjCoord)                                                 ▷ Θ(1)
3:  while haySiguiente(it) do                                               ▷ Θ(#conjCoord)
4:      pCoord ← siguiente(it)                                              ▷ Θ(1)
5:      if seCapturo(pCoord, coord, juego) then                             ▷ Θ(1)
6:          agregarRapido(conj, jug)                                        ▷ Θ(1)
7:      end if
8:      avanzar(it)                                                         ▷ Θ(1)
9:  end while
10: res ← conj                                                              ▷ Θ(1)
```

Complejidad: Θ(#*conjCoord*)

seCapturo(**in** *coord1* : `coordenada`, **in** *coord2* : `coordenada`,**in** *juego* : `juego`) → *res* : *bool*

```
1:  ret ← false                                                                          ▷ Θ(1)
2:  tupla ← &juego.posiciones[longitud(coord1)][latitud(coord1)]                         ▷ Θ(1)
3:  if hayPokemonCercano(coord2,juego) then                                              ▷ Θ(1)
4:      if posPokemonCercano(coord2, juego) ≠ coord1 ∧ tupla.pokemonACapturar ≠ NULL then ▷ Θ(1)
5:          ret ← true                                                                   ▷ Θ(1)
6:      end if
7:  else
8:      if tupla.pokemonACapturar ≠ NULL then                                            ▷ Θ(1)
9:          ret ← true                                                                   ▷ Θ(1)
10:     end if
11: end if
12: res ← ret                                                                            ▷ Θ(1)
```

Complejidad: Θ(1)

indiceRareza(**in** *poke* : `pokemon`,**in** *juego* : `juego`) → *res* : *nat*

```
1:  cant ← 0                                                                             ▷ O(1)
2:  if Definido?(juego.pokemonesSalvajes, poke) then                                     ▷ O(P)
3:      cant ← Obtener(juego.pokemonesSalvajes, poke) + cant                             ▷ O(—P—)
4:  end if
5:  if Definido?(juego.pokemonesCapturados, poke) then                                   ▷ O(—P—)
6:      cant ← Obtener(juego.pokemonesCapturados, poke)                                  ▷ O(—P—)
7:  end if
8:  res ← 100 − (100 ∗ cant/cantPokemonsTotales(juego))                                  ▷ O(1)
```

Complejidad: Θ(|*P*|) Donde P es el nombre mas largo para un pokemon en el juego

cantPokémonsTotales(**in** *juego* : `juego`) → *res* : *nat*

```
1:  res ← juego.cantPokemonTotal
```

todosLosPokémons(**in** *juego* : `juego`) → *res* : *lista*(*pokemons*)

```
1:  lista ← pokemonsSalvajes(juego.posConPokemones, juego)                               ▷ Θ(#posConPokemones ∗ PS)
2:  it ← creatIt(pokemonsCapturados(juego.jugNoExpulsados, juego))                       ▷ Θ(#jugNoExpulsados ∗ PCjug)
3:  while haySiguiente(it) do                                                            ▷ Θ(#jugNoExpulsados ∗ PCjug)
4:      poke ← siguiente(it)                                                             ▷ Θ(1)
5:      agregarAtras(lista, poke)                                                        ▷ Θ(1)
6:      avanzar(it)                                                                      ▷ Θ(1)
7:  end while
8:  res ← lista                                                                          ▷ Θ(1)
```

Complejidad: Θ((#*juego.jugNoExpulsados* ∗ *PCjug*) + (#*juego.posConPokemones* ∗ *PS*))

pokemonsSalvajes(**in** *conjCoord*: `conj(coordenada)`,**in** *juego*: `juego`) → *res* : *lista*(*pokemon*)

```
1:  lista ← vacia()                                                          ▷ Θ(1)
2:  it ← creatIt(conjCoord)                                                  ▷ Θ(1)
3:  while haySiguiente(it) do                                                ▷ Θ(#conjCoord ∗ PS)
4:      pCoord ← siguiente(it)                                               ▷ Θ(1)
5:      if pertenece(juego.posConPokemones, pCoord) then                     ▷ Θ(PS)
6:          tupla ← juego.posiciones[longitud(pCoord)][latitud(pCoord)]      ▷ Θ(1)
7:          poke ← tupla.pokemonACapturar → pokemon                          ▷ Θ(1)
8:          agregarAtras(lista, poke)                                        ▷ Θ(1)
9:      end if
10:     avanzar(it)                                                          ▷ Θ(1)
11: end while
12: res ← lista                                                              ▷ Θ(1)
```

Complejidad: $\Theta(\#conjCoord * PS)$

---

pokemonsCapturados(**in** *jugadores*: `conj(jugador)`,**in** *juego*: `juego`) → *res* : *lista*(*pokemon*)

```
1:  lista ← vacia()                                                          ▷ Θ(1)
2:  it ← creatIt(jugadores)                                                  ▷ Θ(1)
3:  while haysiguiente(it) do                                                ▷ Θ(#jugadores ∗ PCjug)
4:      jug ← siguiente(it)                                                  ▷ Θ(1)
5:      itPoke ← pokemons(jug, juego)                                        ▷ Θ(1)
6:      while haySiguiente(itPoke) do                                        ▷ Θ(PCjug)
7:          tupPokemon ← siguiente(itPoke)                                   ▷ Θ(1)
8:          cantidadParaAgregar ← π2(tupPokemon)                             ▷ Θ(1)
9:          while cantidadParaAgregar > 0 do                                 ▷ Θ(π2(tupPokemon))
10:             agregarAtras(lista, π1(tupPokemon))                          ▷ Θ(1)
11:             cantidadParaAgregar ← cantidadParaAgregar − 1                ▷ Θ(1)
12:         end while
13:         avanzar(it)                                                      ▷ Θ(1)
14:     end while
15:     avanzar(it)                                                          ▷ Θ(1)
16: end while
17: res ← lista                                                              ▷ Θ(1)
```

Complejidad: $\Theta(\#jugadores * PCjug)$

---

buscarPokémonsCercanos(**in** *coord*: `coordenada`, **in** *conjCoord*: `conj(coordenada)`, **in** *juego*: `juego`) → *res* : *conj*(*coordenada*)

```
1:  conj ← vacio()                                                           ▷ Θ(1)
2:  it ← creatIt(conjCoord)                                                  ▷ Θ(1)
3:  while haySiguiente(it) do                                                ▷ Θ(#conjCoord)
4:      pCoord ← siguiente(it)                                               ▷ Θ(1)
5:      if distEuclidea(coord, pCoord) ≤ 4 then                              ▷ Θ(1)
6:          agregarRapido(conj, pCoord)                                      ▷ Θ(1)
7:      end if
8:      avanzar(it)                                                          ▷ Θ(1)
9:  end while
10: res ← conj                                                               ▷ Θ(1)
```

Complejidad: $\Theta(\#conjCoord)$

cantMismaEspecie(**in** *poke* : `pokemon`,**in** *pokemones* : `conj(pokemon)`) → *res* : *nat*

```
1:  cant ← 0                                                          ▷ Θ(1)
2:  it ← creatIt(pokemones)                                           ▷ Θ(1)
3:  while haySiguiente(it) do                                         ▷ Θ(#pokemones)
4:      poke2 ← siguiente(it)                                         ▷ Θ(1)
5:      if poke = poke2 then                                          ▷ Θ(1)
6:          cant ← cant + 1                                           ▷ Θ(1)
7:      end if
8:      avanzar(it)                                                   ▷ Θ(1)
9:  end while
10: res ← cant                                                        ▷ Θ(1)
```

Complejidad: $\Theta(\#pokemones)$

Operaciones Auxiliares (privadas)

crearPosiciones(**in** *map* : `puntero(mapa)`) → *res* : *posiciones*

```
1:  ancho ← map.maxLongitud                                           ▷ Θ(1)
2:  largo ← map.maxLatitud                                            ▷ Θ(1)
3:  posiciones ← crearArreglo(largo)                                  ▷ Θ(largo)
4:  lg ← 0                                                            ▷ Θ(1)
5:  while lg < largo do                                               ▷ Θ(largo ∗ ancho)
6:      arrAncho ← crearArreglo(ancho)                                ▷ Θ(ancho)
7:      an ← 0                                                        ▷ Θ(1)
8:      while an < ancho do                                           ▷ Θ(ancho)
9:          arrAncho[an] ← ⟨vacia(), NULL⟩                            ▷ Θ(1)
10:         an ← an + 1                                               ▷ Θ(1)
11:     end while
12:     posiciones[lg] ← arrAncho                                     ▷ Θ(1)
13:     lg ← lg + 1                                                   ▷ Θ(1)
14: end while
15: res ← posiciones                                                  ▷ Θ(1)
```

Complejidad: $\Theta(largo*ancho)$, donde largo es la coordenada con la longitud mas grande del mapa y ancho es la coordenada con la latitud mas grande del mapa.

Justificación: El algoritmo crea un arreglo con largo cantidad de posiciones y recorrer cada una para insertar otro arreglo de ancho cantidad de posiciones, dejando una complejidad de $(2 * largo * ancho)$ donde el 2 es una constante por crear cada vez los arreglos correspondientes. (Luego la constante se omite)

posicionarPokemon(**in** *poke*: `pokemon`, **in** *coord*: `coordenada`, **in** *itCoord*: `itConj(coordenada)`, **in/out** *juego*: `juego`)

1: $tupla \leftarrow \&juego.posiciones[longitud(coord)][latitud(coord)]$ ▷ $\Theta(1)$
2: $jugACapturarlo \leftarrow Vacia()$ ▷ $\Theta(1)$
3: $lat \leftarrow latitud(coord)$ ▷ $\Theta(1)$
4: $long \leftarrow longitud(coord)$ ▷ $\Theta(1)$
5: $lat\_desde \leftarrow lat - 2$ ▷ $\Theta(1)$
6: **while** $lat\_desde \leq lat + 2$ **do** ▷ $\Theta(4 * 4)$
7: $long\_desde \leftarrow long - 2$ ▷ $\Theta(1)$
8: **while** $long\_desde \leq lat + 2$ **do** ▷ $\Theta(4)$
9: **if** distEuclidea($\langle lat\_desde, long\_desde\rangle$,coord) ≤ 4 **then** ▷ $\Theta(1)$
10: $subTupla \leftarrow \&juego.posiciones[long\_desde][lat\_desde]$ ▷ $\Theta(1)$
11: $itJugadores \leftarrow CrearIt(subTupla.jugadores)$ ▷ $\Theta(1)$
12: **while** $haySiguiente(itJugadores)$ **do** ▷ $\Theta(EC{*}log(EC))$
13: $jugador \leftarrow Siguiente(itJugadores)$ ▷ $\Theta(1)$
14: $jugStruct \leftarrow \&juego.jugadores[\pi_2(jugador)]$ ▷ $\Theta(1)$
15: $jugStruct.itCapturarPoke \leftarrow encolar(jugACapturarlo, jugador)$ ▷ $\Theta(log(EC))$
16: $Avanzar(itJugadores)$ ▷ $\Theta(1)$
17: **end while**
18: **end if**
19: $long\_desde \leftarrow long\_desde + 1$ ▷ $\Theta(1)$
20: **end while**
21: $lat\_desde \leftarrow lat\_desde + 1$ ▷ $\Theta(1)$
22: **end while**
23: $puntero \leftarrow \&nuevoPokemonACapturar(poke, itCoord, jugACapturarlo)$ ▷ $\Theta(1)$
24: $tupla.pokemonACapturar \leftarrow puntero$ ▷ $\Theta(1)$

Complejidad: $\Theta(EC{*}log(EC))$, donde EC es la máxima cantidad de jugadores esperando para atrapar un pokémon.

Justificación: Recorro las posiciones que se encuentran dentro del radio del pokémon (las posiciones son constanstes dado que el radio siempre es el mismo) y por cada posición que agrego recorro los jugadores que están para capturar el pokémon (EC) y los agrego a la cola de prioridad para capturarlo (log(EC) por cada uno). Dejando así una complejidad de EC∗log(EC).

---

nuevoPokemonACapturar(**in** *poke*: `pokemon`, **in** *itCoord*: `itConj(coordenada)`,**in/out** *jugACapturarlo*: `colaPrior(tupla(nat,jugador))`) $\rightarrow$ *res* : *pokemonACapturar*

1: $res.pokemon \leftarrow poke$ ▷ $\Theta(1)$
2: $res.movAfuera \leftarrow 0$ ▷ $\Theta(1)$
3: $res.jugACapturarlo \leftarrow jugACapturarlo$ ▷ $\Theta(1)$
4: $res.itCoord \leftarrow itCoord$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

nuevaJugadorStruct(**in** *itNoExpulsados*: `itConj(jugador)`) $\rightarrow$ *res* : *jugadorStruct*

1: $res.estado \leftarrow desconectado$ ▷ $\Theta(1)$
2: $res.sanciones \leftarrow 0$ ▷ $\Theta(1)$
3: $res.pos \leftarrow (-1, -1)$ ▷ $\Theta(1)$
4: $res.pokemones \leftarrow Vacia()$ ▷ $\Theta(1)$
5: $res.pokemonesTotales \leftarrow 0$ ▷ $\Theta(1)$
6: $res.itPosJug \leftarrow CrearIt(Vacia())$ ▷ $\Theta(1)$
7: $res.itCapturarPoke \leftarrow CrearIt(Vacia())$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

capturarPokemon(**in** jug : `jugador`, **in** $poke$: `pokemon`, **in/out** $juego$: `juego`)

1: $jugador \leftarrow juego.jugadores[jug]$ ▷ $\Theta(1)$
2: $cantPoke \leftarrow obtener(juego.pokemonesSalvajes)$ ▷ $\Theta(1)$
3: $definir(juego.pokemonesSalvajes, poke, cantPoke - 1)$ ▷ $\Theta(|P|)$
4: **if** $definido(juego.pokemonesCapturados, poke)$ **then** ▷ $\Theta(|P|)$
5: $cantPoke \leftarrow obtener(juego.pokemonesCapturados, poke)$ ▷ $\Theta(|P|)$
6: $definir(juego.pokemonesCapturados, poke, cantPoke + 1)$ ▷ $\Theta(|P|)$
7: **else**
8: $definir(juego.pokemonesCapturados, poke, 1)$ ▷ $\Theta(|P|)$
9: **end if**
10: **if** $definido(jugador.itPokemones, poke)$ **then** ▷ $\Theta(|P|)$
11: $itPokemones \leftarrow obtener(jugador.itPokemones, poke)$
12: $cantPoke \leftarrow \pi_2(Siguiente(itPokemones))$ ▷ $\Theta(1)$
13: $\pi_2(Siguiente(itPokemones) \leftarrow cantPoke + 1$ ▷ $\Theta(1)$
14: **else**
15: $it \leftarrow CrearItUlt(jugador.pokemones)$ ▷ $\Theta(1)$
16: $itPoke \leftarrow AgregarComoSiguiente(it, \langle poke, 1 \rangle)$ ▷ $\Theta(1)$
17: $definir(jugador.itPokemones, poke, itPoke)$ ▷ $\Theta(|P|)$
18: **end if**
19: $jugador.pokemonesTotales \leftarrow jugador.pokemonesTotales + 1$ ▷ $\Theta(1)$
20: $juego.jugadores[jug] \leftarrow jugador$ ▷ $\Theta(1)$

Complejidad: $\Theta(|P|)$, donde $|P|$ es el nombre más largo para un pokémon en el juego.
Justificacion: