

# Algoritmos y Estructuras de Datos II

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

## Trabajo Práctico II

### Diseño

#### Grupo De TP Algo2

Integrante	LU	Correo electrónico
Fernando Castro	627/12	fernandoarielcastro92@gmail.com
Philip Garrett	318/14	garrett.phg@gmail.com
Gabriel Salvo	564/14	gabrielsalvo.cap@gmail.com
Bernardo Tusó	792/14	btuso.95@gmail.com

#### Reservado para la cdra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

# Índice

<b>1. Modulo Coordenada</b>	<b>4</b>
1.0.1. Representación de Coordenada . . . . .	5
1.0.2. Invariante de Representación . . . . .	5
1.0.3. Función de Abstracción . . . . .	5
<b>2. Modulo Mapa</b>	<b>7</b>
2.0.1. Representación de Mapa . . . . .	8
2.0.2. Invariante de Representación . . . . .	8
2.0.3. Función de Abstracción . . . . .	8
<b>3. Modulo Juego</b>	<b>11</b>
3.0.1. Representación de Juego . . . . .	13
3.0.2. Invariante de Representación . . . . .	13
3.0.3. Función de Abstracción . . . . .	14
3.1. Algoritmos . . . . .	15
<b>4. Modulo Diccionario Matriz(<math>coord</math>, <math>\sigma</math>)</b>	<b>24</b>
4.0.1. Especificacion de las operaciones auxiliares utilizadas en la interfaz . . . . .	26
<b>5. Módulo Cola de mínima prioridad(<math>\alpha</math>)</b>	<b>30</b>
5.1. Especificación . . . . .	30
5.2. Interfaz . . . . .	30
5.2.1. Operaciones básicas de ColaMinPrior . . . . .	31
5.2.2. Operaciones del Iterador . . . . .	31
5.3. Representación . . . . .	32
5.3.1. Representación de ColaMinPrior( $\alpha$ ) . . . . .	32
5.3.2. Invariante de Representación (Rehacer con nueva estructura) . . . . .	32
5.3.3. Función de Abstracción . . . . .	32
5.3.4. Representación del Iterador Cola de Prioridad . . . . .	32
5.3.5. Invariante de Representación . . . . .	32
5.3.6. Función de Abstracción . . . . .	33
5.4. Algoritmos . . . . .	33
5.4.1. Algoritmos del Modulo . . . . .	33
5.4.2. Algoritmos del Iterador . . . . .	36
<b>6. Módulo Diccionario String(<math>\alpha</math>)</b>	<b>38</b>
6.1. Interfaz . . . . .	38
6.1.1. Operaciones básicas de Diccionario String( $\alpha$ ) . . . . .	38
6.1.2. Operaciones Básicas Del Iterador . . . . .	39
6.1.3. Representación de Diccionario String( $\alpha$ ) . . . . .	41
6.1.4. Invariante de Representación . . . . .	41
6.1.5. Función de Abstracción . . . . .	42

---

6.2. Algoritmos . . . . .	42
---------------------------	----

## 1. Modulo Coordenada

### Interfaz

**usa:** NAT, BOOL.

**se explica con:** COORDENADA.

**generos:** `coor`.

**CREARCOOR**(**in**  $x : \text{Nat}$ , **in**  $y : \text{Nat}$ )  $\rightarrow res : \text{coor}$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{crearCoor}(x, y)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Crea una nueva coordenada

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

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

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

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la latitud de la coordenada pasada por parametro

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

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

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

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la longitud de la coordenada pasada por parametro

**DISTEUCLIDEA**(**in**  $c1 : \text{coor}$ , **in**  $c2 : \text{coor}$ )  $\rightarrow res : \text{Nat}$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{distEuclidea}(c1, c2)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la distancia euclidea entre las dos coordenadas

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

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

**Post**  $\equiv \{res =_{\text{obs}} \text{coordenadaArriba}(c)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la coordenada de arriba

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

**Pre**  $\equiv \{\text{latitud}(c) > 0\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{coordenadaAbajo}(c)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la coordenada de abajo

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

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

**Post**  $\equiv \{res =_{\text{obs}} \text{coordenadaALaDerecha}(c)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la coordenada de la derecha

**COORDENADAALAIZQUIERDA**(**in**  $c : \text{coor}$ )  $\rightarrow res : \text{coor}$

**Pre**  $\equiv \{\text{longitud}(c) > 0\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{coordenadaALaIzquierda}(c)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve la coordenada de la izquierda

### Representación

**1.0.1. Representación de Coordenada**

Coordenada se representa con `estr`

donde `estr` es `tupla(la: Nat , lo: Nat )`

**1.0.2. Invariante de Representación**

`Rep : estr → bool`

`Rep(e) ≡ true ⇔ true`

**1.0.3. Función de Abstracción**

`Abs : estr e → coor`

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

`Abs(e) ≡ (∀c: coor) e.la = latitud(c) ∧ e.lo = longitud(c)`

**Algoritmos**

Trabajo Práctico II Algoritmos del modulo

---

---

**iCrearCoor**(`in x: Nat, in y: Nat`) → `res: estr`

1: `res.la ← x`

▷  $\Theta(1)$

2: `res.lo ← y`

▷  $\Theta(1)$

Complejidad:  $\Theta(1)$

---



---

---

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

1: `res ← c.la`

▷  $\Theta(1)$

Complejidad:  $\Theta(1)$

---



---

---

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

1: `res ← c.lo`

▷  $\Theta(1)$

Complejidad:  $\Theta(1)$

---

---



---

**iDistEuclidea**(in  $c1 : \text{estr}$ , in  $c2 : \text{estr}$ )  $\rightarrow res : \text{Nat}$ 

```

1: rLa  $\leftarrow$  0  $\triangleright \Theta(1)$ 
2: rLo  $\leftarrow$  0  $\triangleright \Theta(1)$ 
3: if  $c1.la > c2.la$  then  $\triangleright \Theta(1)$ 
4:   rLa  $\leftarrow ((c1.la - c2.la) \times (c1.la - c2.la))$   $\triangleright \Theta(1)$ 
5: else
6:   rLa  $\leftarrow ((c2.la - c1.la) \times (c2.la - c1.la))$   $\triangleright \Theta(1)$ 
7: end if
8: if  $c1.lo > c2.lo$  then  $\triangleright \Theta(1)$ 
9:   rLo  $\leftarrow ((c1.lo - c2.lo) \times (c1.lo - c2.lo))$   $\triangleright \Theta(1)$ 
10: else
11:   rLo  $\leftarrow ((c2.lo - c1.lo) \times (c2.lo - c1.lo))$   $\triangleright \Theta(1)$ 
12: end if
13:  $res \leftarrow (rLa + rLo)$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---



---



---

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

```

1:  $res \leftarrow iCrearCoor(c.la + 1, c.lo)$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---



---



---

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

```

1:  $res \leftarrow iCrearCoor(c.la - 1, c.lo)$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---



---



---

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

```

1:  $res \leftarrow iCrearCoor(c.la, c.lo + 1)$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---



---



---

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

```

1:  $res \leftarrow iCrearCoor(c.la, c.lo - 1)$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---

## 2. Modulo Mapa

### Interfaz

**usa:** NAT, BOOL, COORDENADA, CONJ( $\alpha$ ).

**se explica con:** MAPA.

**generos:** map.

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

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

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

**Complejidad:**  $O(1)$

**Descripción:** Crea un nuevo mapa

AGREGARCOORDENADA(**in/out**  $m : \text{map}$ , **in**  $c : \text{coor}$ )  $\rightarrow res : \text{itConj}(\text{coor})$

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

**Post**  $\equiv \{m =_{\text{obs}} \text{agregarCoor}(c, m_0)\}$

**Complejidad:**  $\Theta \left( \sum_{c' \in \text{coordenadas}(m)} \text{equal}(c, c') \right)$

**Descripción:** Agrega una coordenada al mapa y devuelve el iterador a la coordenada agregada. Su complejidad es la de agregar un elemento al conjunto lineal.

COORDENADAS(**in**  $m : \text{map}$ )  $\rightarrow res : \text{itConj}(\text{coor})$

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

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

**Complejidad:**  $O(1)$

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

POSEXISTENTE(**in**  $c : \text{coor}$ , **in**  $m : \text{map}$ )  $\rightarrow res : \text{Bool}$

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

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

**Complejidad:**  $\Theta \left( \sum_{c' \in \text{coordenadas}(m)} \text{equal}(c, c') \right)$

**Descripción:** Devuelve verdadero si la coordenada esta en el conjunto de coordenadas del mapa

HAYCAMINO(**in**  $c1 : \text{coor}$ , **in**  $c2 : \text{coor}$ , **in**  $m : \text{map}$ )  $\rightarrow res : \text{Bool}$

**Pre**  $\equiv \{c1 \in \text{coordenadas}(m) \wedge c2 \in \text{coordenadas}(m)\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{hayCamino}(c1, c2, m)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve verdadero si existe un camino entre ambas coordenadas

ANCHO(**in**  $m : \text{map}$ )  $\rightarrow res : \text{Nat}$

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

**Post**  $\equiv \{(\forall c : \text{coor}) c \in \text{coordenadas}(m) \Rightarrow_L \text{longitud}(res) \geq \text{longitud}(c) \}$

**Complejidad:**  $\Theta(\#\text{coordenadas}(m))$

**Descripción:** Devuelve el ancho del mapa

LARGO(**in**  $m : \text{map}$ )  $\rightarrow res : \text{Nat}$

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

**Post**  $\equiv \{(\forall c : \text{coor}) c \in \text{coordenadas}(m) \Rightarrow_L \text{latitud}(res) \geq \text{latitud}(c) \}$

**Complejidad:**  $\Theta(\#\text{coordenadas}(m))$

**Descripción:** Devuelve el largo del mapa

### Representación

### 2.0.1. Representación de Mapa

Mapa se representa con *estr*

donde *estr* es  $\text{tupla}(\text{coordenadas: ConjLineal}(\text{coor}), \text{secciones: DiccMat}(\text{coor}, \text{Nat}))$

### 2.0.2. Invariante de Representación

1. El ancho del mapa es igual al maximo del primer elemento de las coordenadas

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

$\text{Rep}(e) \equiv \text{true} \iff (\forall c : \text{coor}) c \in e.\text{coordenadas} \Rightarrow_L \text{def?}(c, e.\text{secciones})$

### 2.0.3. Función de Abstracción

$\text{Abs} : \text{estr } e \rightarrow \text{Mapa}$

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

$\text{Abs}(e) \equiv (\forall m : \text{Mapa}) e.\text{coordenadas} = \text{coordenadas}(m)$

## Algoritmos

Trabajo Práctico II Algoritmos del modulo

---

**iCrearMapa()**  $\rightarrow res : \text{estr}$

1:  $res.\text{coordenadas} \leftarrow \text{Vacio}()$

$\triangleright$  La complejidad es la de crear el Conjunto Lineal vacio  $\Theta(1)$

2:  $res.\text{secciones} \leftarrow \text{NULL}$

$\triangleright O(1)$

Complejidad:  $\Theta(1)$

---



---

**iAgregarCoordenada(in/out m : estr, in c : coor)**  $\rightarrow res : \text{itConj}(\text{coor})$

1:  $\text{largo} \leftarrow \text{Largo}(m)$

$\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$

2:  $\text{ancho} \leftarrow \text{Ancho}(m)$

$\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$

3:  $m.\text{secciones} \leftarrow \text{Vacio}(\text{largo}, \text{ancho})$

$\triangleright \Theta(\text{largo} * \text{ancho})$

4:  $res \leftarrow \text{Agregar}(m.\text{coordenadas}, c)$

$\triangleright \Theta \left( \sum_{c' \in \text{coordenadas}(m)} \text{equal}(c, c') \right)$

5:  $\text{seccion} \leftarrow 0$

$\triangleright \Theta(1)$

6:  $\text{itCoor} \leftarrow \text{CrearIt}(m.\text{coordenadas})$

$\triangleright \Theta(1)$

7: **while** *HaySiguiente(itCoor)* **do**

$\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$

8:    $\text{coord} \leftarrow \text{Siguiente}(\text{itCoor})$

$\triangleright \Theta(1)$

9:   *Avanzar(it)*

$\triangleright \Theta(1)$

10:   **if**  $\neg(\text{Definido?}(m.\text{secciones}, \text{coord}))$  **then**

$\triangleright \Theta(1)$

11:     *DefinirSeccion(m, coord, seccion)*

$\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$

12:      $\text{seccion} \leftarrow \text{seccion} + 1$

$\triangleright \Theta(1)$

13:   **end if**

14: **end while**

Complejidad:  $\Theta(\text{Ancho}(m) * \text{Largo}(m) + \text{Cardinal}(m.\text{coordenadas})^2)$

Justificación:  $\Theta(\text{Ancho}(m) * \text{Largo}(m))$  es mayor o igual que  $\Theta(\text{Cardinal}(m.\text{coordenadas}))$  y el costo de Agregar un elemento a un conjunto lineal. El While tiene complejidad  $\Theta(\text{Cardinal}(m.\text{coordenadas}))$  dentro, y dentro se llama a una funcion con la misma complejidad, luego, por algebra de complejidad, es  $\Theta(\text{Ancho}(m) * \text{Largo}(m) + \text{Cardinal}(m.\text{coordenadas})^2)$

---



---

**iDefinirSeccion**(in/out  $m$  : **estr**, in  $c$  : **coor**, in  $i$  : **Nat**)

```

1: if  $\neg(\text{Definido?}(m.\text{secciones}, c)) \wedge \text{PosExistente}(c, m)$  then
2:    $\text{Definir}(m.\text{secciones}, c, i)$ 
3:    $\text{DefinirSeccion}(m, \text{CoordenadaArriba}(c), i)$ 
4:    $\text{DefinirSeccion}(m, \text{CoordenadaALaDerecha}(c), i)$ 
5:   if  $\text{Latitud}(c) > 0$  then
6:      $\text{DefinirSeccion}(m, \text{CoordenadaAbajo}(c), i)$ 
7:   end if
8:   if  $\text{Longitud}(c) > 0$  then
9:      $\text{DefinirSeccion}(m, \text{CoordenadaALaIzquierda}(c), i)$ 
10:  end if
11: end if

```

$\triangleright \Theta \left( \sum_{c' \in \text{coordendas}(m)} \text{equal}(c, c') \right)$   
 $\triangleright \Theta(1)$   
 $\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$   
 $\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$   
 $\triangleright \Theta(1)$   
 $\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$   
 $\triangleright \Theta(1)$   
 $\triangleright \Theta(\text{Cardinal}(m.\text{coordenadas}))$

Complejidad:  $\Theta(\text{Cardinal}(m.\text{coordenadas}))$

Justificación DefinirSeccion se llama a si misma recursivamente recorriendo las coordenadas, en el peor caso, recorre todas las coordenadas una vez, luego su complejidad es  $\Theta(4^{\text{Cardinal}(m.\text{coordenadas})})$  que se puede simplificar, ya que pertenece a la misma clase. Esta funcion no es cuadratica, ya que usa el diccionario para chequear que no este recorriendo una posicion mas de una vez.

---



---

**iCoordenadas**(in  $m$  : **estr**)  $\rightarrow res$  : **itConj**(**coor**)

```

1:  $res \leftarrow \text{CrearIt}(m.\text{coordenadas})$   $\triangleright$  La complejidad es la de crear un iterador a un conjunto lineal  $\Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---



---

**iPosExistente**(in  $c$  : **coor**, in  $m$  : **estr**)  $\rightarrow res$  : **Bool**

```

1:  $res \leftarrow \text{pertenece?}(m.\text{coordenadas}, c)$   $\triangleright \Theta \left( \sum_{c' \in \text{coordendas}(m)} \text{equal}(c, c') \right)$ 

```

Complejidad:  $\Theta \left( \sum_{c' \in \text{coordendas}(m)} \text{equal}(c, c') \right)$

Justificación: La complejidad es la fijarse que un elemento pertenezca al conjunto lineal.

---



---

**iHayCamino**(in  $c1$  : **coor**, in  $c2$  : **coor**, in  $m$  : **estr**)  $\rightarrow res$  : **Bool**

```

1:  $res \leftarrow (\text{Definido?}(m.\text{secciones}, c1) \wedge \text{Definido?}(m.\text{secciones}, c2)) \wedge_L (\text{Significado}(m.\text{secciones}, c1) = \text{Significado}(m.\text{secciones}, c2))$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---

---



---

**iAncho**(in  $m : \text{estr}$ )  $\rightarrow res : \text{Nat}$ 

```

1:  $it \leftarrow \text{CrearIt}(m.coordenadas)$   $\triangleright \Theta(1)$ 
2:  $max \leftarrow 0$   $\triangleright \Theta(1)$ 
3: while HaySiguiente(it) do  $\triangleright \Theta(\text{Cardinal}(m.coordenadas))$ 
4:   if  $max < \text{Longitud}(\text{Siguiente}(it))$  then  $\triangleright \Theta(1)$ 
5:      $max \leftarrow \text{Longitud}(\text{Siguiente}(it))$   $\triangleright \Theta(1)$ 
6:   end if
7:    $\text{Avanzar}(it)$   $\triangleright \Theta(1)$ 
8: end while

```

Complejidad:  $\Theta(\text{Cardinal}(m.coordenadas))$

---



---



---

**iLargo**(in  $m : \text{estr}$ )  $\rightarrow res : \text{Nat}$ 

```

1:  $it \leftarrow \text{CrearIt}(m.coordenadas)$   $\triangleright \Theta(1)$ 
2:  $max \leftarrow 0$   $\triangleright \Theta(1)$ 
3: while HaySiguiente(it) do  $\triangleright \Theta(\text{Cardinal}(m.coordenadas))$ 
4:   if  $max < \text{Latitud}(\text{Siguiente}(it))$  then  $\triangleright \Theta(1)$ 
5:      $max \leftarrow \text{Latitud}(\text{Siguiente}(it))$   $\triangleright \Theta(1)$ 
6:   end if
7:    $\text{Avanzar}(it)$   $\triangleright \Theta(1)$ 
8: end while

```

Complejidad:  $\Theta(\text{Cardinal}(m.coordenadas))$

---

### 3. Modulo Juego

#### Interfaz

**usa:** MAPA, COORDENADA.

**se explica con:** JUEGO.

**generos:** juego.

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

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{crearJuego}(m_0) \wedge \text{mapa}(res) =_{\text{obs}} m_0\}$

**Complejidad:**  $O((\text{largo}(m) \times \text{ancho}(m)) + \text{copy}(m))$

**Descripción:** Crea el nuevo juego

**AGREGARPOKEMON**(in/out  $j$ : juego, in  $c$ : coor, in  $p$ : Pokemon)  $\rightarrow res$ : itConj(Pokemon)

**Pre**  $\equiv \{j =_{\text{obs}} j_0 \wedge \text{puedoAgregarPokemon}(c, j_0)\}$

**Post**  $\equiv \{j =_{\text{obs}} \text{agregarPokemon}(p, c, j_0)\}$

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

**Descripción:** EC es la maxima cantidad de jugadores esperando para atrapar un pokemon.  $|P|$  es el nombre mas largo para un pokemon en el juego

**AGREGARJUGADOR**(in/out  $j$ : juego)  $\rightarrow res$ : Jugador

**Pre**  $\equiv \{j =_{\text{obs}} j_0\}$

**Post**  $\equiv \{j =_{\text{obs}} \text{agregarJugador}(j_0) \wedge res = \#jugadores(j_0) + \#expulsados(j_0)\}$

**Complejidad:**  $O(J)$

**Descripción:** Agrega el jugador en el conjLineal, y devuelve su identificador

**CONECTARSE**(in/out  $j$ : juego, in  $id$ : Jugador, in  $c$ : coor)

**Pre**  $\equiv \{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge \neg \text{estaConectado}(id, j_0) \wedge \text{posExistente}(c, \text{mapa}(j_0))\}$

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

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

**Descripción:** Conecta al jugador pasado por parametro en la coordenada indicada

**DESCONECTARSE**(in/out  $j$ : juego, in  $id$ : Jugador)

**Pre**  $\equiv \{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge \text{estaConectado}(id, j_0)\}$

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

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

**Descripción:** Desconecta al jugador pasado por parametro

**MOVERSE**(in/out  $j$ : juego, in  $id$ : Jugador, in  $c$ : coor)

**Pre**  $\equiv \{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge \text{estaConectado}(id, j_0) \wedge \text{posExistente}(c, \text{mapa}(j_0))\}$

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

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

**Descripción:** Mueve al jugador pasado por parametro a la coordenada indicada

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

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{mapa}(j)\}$

**Complejidad:**  $O(\text{copy}(\text{mapa}(j)))$

**Descripción:** Devuelve el mapa del juego

**JUGADORES**(in  $j$ : juego)  $\rightarrow res$ : itConj(Jugador)

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{res =_{\text{obs}} jugadores(j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve un iterador al conjunto de jugadores del juego

**ESTACONECTADO**(in  $j$ : juego, in  $id$ : Jugador)  $\rightarrow res$ : Bool

**Pre**  $\equiv \{id \in jugadores(j)\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{estaConetado}(id, j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve si el jugador con id ingresado esta conectado o no

**POSICION**(in  $j$ : juego, in  $id$ : Jugador)  $\rightarrow res$  : coor

**Pre**  $\equiv \{id \in \text{jugadores}(j) \wedge_L \text{estaConectado}(id, j)\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{posicion}(id, j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve la posicion actual del jugador con id ingresado si esta conectado

**POKEMONES**(in  $j$ : juego, in  $id$ : Jugador)  $\rightarrow res$  : itConj(<Pokemon, Nat>)

**Pre**  $\equiv \{id \in \text{jugadores}(j)\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{pokemons}(id, j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve un iterador a la estructura que almacena los punteros a pokemons del jugador del id ingresado

**EXPULSADOS**(in  $j$ : juego)  $\rightarrow res$  : itConj(Jugador)

**Pre**  $\equiv \{\text{True}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{expulsados}(j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve un iterador al conjunto de jugadores expulsados del juego

**POSCONPOKEMONES**(in  $j$ : juego)  $\rightarrow res$  : itConj(coor)

**Pre**  $\equiv \{\text{True}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{posConPokemons}(j)\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve un iterador al conjunto de coordenadas en donde hay pokemons

**POKEMONENPOS**(in  $j$ : juego, in  $c$ : coor)  $\rightarrow res$  : Pokemon

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

**Post**  $\equiv \{res =_{\text{obs}} \text{pokemonEnPos}(c, j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve un iterador al pokemon de la coordenada dada

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

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

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

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve la cantidad de movimientos acumulados hasta el momento, para atrapar al pokemon de la coordenada dada

**PUEDOAGREGARPOKEMON**(in  $j$ : juego, in  $c$ : coor)  $\rightarrow res$  : Bool

**Pre**  $\equiv \{\text{True}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{puedoAgregarPokemon}(c, j)\}$

**Complejidad:**  $\Theta\left(\sum_{c' \in \text{coordendas}(\text{mapa}(j))} \text{equal}(c, c')\right)$

**Descripción:** Devuelve si la coordenada ingresada es valida para agregar un pokemon en ella

**HAYPOKEMONCERCANO**(in  $j$ : juego, in  $c$ : coor)  $\rightarrow res$  : Bool

**Pre**  $\equiv \{\text{True}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{hayPokemonCercano}(c, j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve si la coordenada ingresada pertenece al rango de un pokemon salvaje

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

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

**Post**  $\equiv \{res =_{\text{obs}} \text{posPokemonCercano}(c, j)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve la coordenada mas del pokemon salvaje del rango siempre y cuando haya uno

**ENTRENADORESPOSIBLES**(**in**  $c$ : **coor**, **in**  $es$ : **conjLineal**(**itConj**(**Jugador**)), **in**  $j$ : **juego**)  $\rightarrow res$  : **itConj**(**Jugador**)  
**Pre**  $\equiv \{hayPokemonCercano(c, j) \wedge_L pokemonEnPos(posPokemonCercano(c, j), j).jugadoresEnRango \subseteq jugadoresConectados(c, j)\}$   
**Post**  $\equiv \{res =_{obs} entrenadoresPosibles(c, pokemonEnPos(posPokemonCercano(c, j), j).jugadoresEnRango, j)\}$   
**Complejidad:**  $O(Cardinal(es))$   
**Descripción:** Devuelve un iterador a los jugadores que estan esperando para atrapar al pokemon mas cercano a la coordenada ingresada

**INDICERAREZA**(**in**  $j$ : **juego**, **in**  $p$ : **Pokemon**)  $\rightarrow res$  : **Nat**  
**Pre**  $\equiv \{p \in todosLosPokemons(j)\}$   
**Post**  $\equiv \{res =_{obs} indiceRareza(p, j)\}$   
**Complejidad:**  $O(|P|)$   
**Descripción:** Devuelve el indice de rareza del pokemon del juego ingresado

**CANTPOKEMONESTOTALES**(**in**  $j$ : **juego**)  $\rightarrow res$  : **Nat**  
**Pre**  $\equiv \{true\}$   
**Post**  $\equiv \{res =_{obs} cantPokemonsTotales(p)\}$   
**Complejidad:**  $\Theta(1)$   
**Descripción:** Devuelve la cantidad de pokemones que hay en el juego

**CANTMISMAESPECIE**(**in**  $j$ : **Juego**, **in**  $p$ : **Pokemon**)  $\rightarrow res$  : **Nat**  
**Pre**  $\equiv \{true\}$   
**Post**  $\equiv \{res =_{obs} cantMismaEspecie(p, pokemons(j), j)\}$   
**Complejidad:**  $O(|P|)$   
**Descripción:** Devuelve la cantidad de pokemones de la especie ingresada hay en el juego

## Representación

### 3.0.1. Representación de Juego

Jugador se representa con **Nat**

Pokemon se representa con **String**

Juego se representa con **estr**

donde **estr** es tupla( $pokemones$ : **diccString**(**Pokemon**, **Nat**),  $todosLosPokemones$ : **conjLineal**(**infoPokemon**) ,  $idsJugadores$ : **conjLineal**(**Jugador**) ,  $jugadores$ : **conjLineal**(**infoJugador**) ,  $expulsados$ : **conjLineal**(**Jugador**) ,  $jugadoresPorID$ : **Vector**(<**info**: **itConj**(**infoJugador**), **encolado**: **itColaPrior**(**Jugador**)>) ,  $posicionesPokemons$ : **DiccMat**(**coor**, **itConj**(**infoPokemon**)) ,  $posicionesJugadores$ : **DiccMat**(**coor**, **puntero**(**conjLineal**(**Jugador**))) ,  $mapa$ : **map** )

donde **infoJugador** es tupla( $id$ : **itConj**(**Jugador**) ,  $estaConectado$ : **Bool** ,  $sanciones$ : **Nat** ,  $pokemonesCapturados$ : **conjLineal**(**itConj**(**infoPokemon**)) ,  $posicion$ : **coor** ,  $itPosicion$ : **itConj**(**Jugador**) )

donde **infoPokemon** es tupla( $tipo$ : **Pokemon** ,  $posicion$ : **coor** ,  $contador$ : **Nat** ,  $jugadoresEnRango$ : **colaPrior**<**Nat**, **itConj**(**Jugador**)> ,  $salvaje$ : **Bool** )

### 3.0.2. Invariante de Representación

1. La suma de todas las cantidades de cada tipo de pokemon es igual al cardinal del conjunto **todosLosPokemones**.
2. Para cada **infoPokemon** definido en **todosLosPokemones**, si es **salvaje**, su tipo esta definido en **pokemones**, su

coordenada esta definida en posicionesPokemones y el significado en posicionesPokemones es igual a infoPokemon, su contador es un Nat menor estricto que 10 y cada jugador en rango esta en jugadores.

3. El cardinal de idsJugadores es igual a la longitud de jugadoresPorID, que es igual al cardinal de jugadores + el cardinal de expulsados
4. Cada jugador en idsJugadores pertenece a expulsados, o su numero de id representa la posicion en jugadoresPorID donde esta guardada su informacion. Esta informacion refleja lo definido en jugadores. Si el jugador esta encolado, entonces existe un infoPokemon en todos los pokemones, tal que ese jugador pertenece a los jugadoresEnRango del pokemon.
5. Cada coordenada valida en el mapa, si esta definida en posicionesPokemones, su significado pertenece a todosLosPokemones y es salvaje
6. Cada coordenada valida en el mapa, si esta definida en posicionesJugadores, su significado esta contenido en jugadores
7. Para cada infoJugador en jugadores, su id pertenece a idsJugadores. Si esta conectado, su itPosicion esta contenido en el significado de su posicion en posicionesJugadores. Sus sanciones son menores a 5. Sus pokemones capturados estan contenidos en todosLosPokemones
8. Los conjuntos jugadores y expulsados son disjuntos.

1. 
$$\left( \sum_{c \in \text{claves}(e.\text{pokemones})} \text{Obtener}(e.\text{pokemones}, c) = \#e.\text{todosLosPokemons} \right)$$
2.  $(\forall i : \text{infoPokemon}) \text{pertenece?}(e.\text{todosLosPokemones}, i) \Rightarrow_L (\text{definido?}(e.\text{pokemones}, i.\text{tipo}) \wedge \text{definido?}(e.\text{posicionesPokemones}, i.\text{posicion}) \wedge_L \text{Significado}(e.\text{posicionesPokemones}, i.\text{posicion}) = i \wedge i.\text{contador} < 10 \wedge (\forall j : \text{Jugador}) j \in i.\text{jugadoresEnRango} \Rightarrow_L j \in e.\text{jugadores})$
3.  $\#e.\text{idsJugadores} = \text{long}(e.\text{jugadoresPorID}) \wedge \#e.\text{idsJugadores} = (\#e.\text{jugadores} + \#e.\text{expulsados})$
4.  $(\forall j : \text{jugador}) j \in e.\text{idsJugadores} \Rightarrow_L (j \in e.\text{expulsados} \vee_L e.\text{jugadoresPorID}[j].\text{info} \in e.\text{jugadores} \wedge \text{Siguiete}(e.\text{jugadoresPorID}[j].\text{encolado}) \neq \text{NULL} \Rightarrow_L ((\forall i : \text{infoPokemon}) i \in e.\text{todosLosPokemones} \wedge_L j \in i.\text{jugadoresEnRango}))$
5.  $(\forall c : \text{coord}) c \in e.\text{mapa.coordenadas} \Rightarrow_L (\text{definido?}(e.\text{posicionesPokemones}, c) \Rightarrow_L \text{Siguiete}(\text{Significado}(e.\text{posicionesPokemones}, c)) \in e.\text{todosLosPokemones} \wedge \text{Siguiete}(\text{Significado}(e.\text{posicionesPokemones}, c)).\text{esSalvaje})$
6.  $(\forall c : \text{coord}) c \in e.\text{mapa.coordenadas} \Rightarrow_L (\text{definido?}(e.\text{posicionesJugadores}, c) \Rightarrow_L *(\text{Significado}(e.\text{posicionesPokemones}, c)) \subseteq e.\text{jugadores})$
7.  $(\forall i : \text{infoJugador}) i \in e.\text{jugadores} \Rightarrow_L (i.\text{id} \in e.\text{idsJugadores} \wedge i.\text{estaConectado} \Rightarrow_L (i.\text{itPosicion} \in *\text{Significado}(e.\text{posicionesJugadores}, i.\text{posicion})) \wedge i.\text{sanciones} < 5 \wedge i.\text{pokemonesCapturados} \subseteq e.\text{todosLosPokemones})$
8.  $e.\text{jugadores} \cap e.\text{expulsados} = \emptyset$

Rep : estr  $\rightarrow$  bool

Rep(e)  $\equiv$  true  $\iff$  1  $\wedge_L$  2  $\wedge_L$  3  $\wedge_L$  4  $\wedge_L$  5  $\wedge_L$  6  $\wedge_L$  7  $\wedge_L$  8

### 3.0.3. Función de Abstracción

Abs(e): estre  $\rightarrow$  Jugo Rep(e) pGo: Juego tq e.mapa = mapa(pGo) y e.jugadores = jugadores(pGo) y luego  
 (Para todo j : jugador) j pertenece e.jugadores impluego  
 j.sanciones = sanciones(j, pGo) ((j pertenece expulsados(pGo) y j.sanciones  $\geq$  10)  
 oluego (j.pokesCapturados = pokemones(j, pGo) y j.estaConectado = estaConectad(j, pGo)  
 y j.estaConectado impluego j.pos = posicion(j, pGo))) y  
 (Para todo p : pokemon) p pertenece e.pokemones impluego (Para todo j : Jugador)  
 j pertenece e.jugadores y luego p pertenece pokemones(j, pGo) o [(Para todo c : coord)  
 c pertenece e.mapa.coordenadas y luego p = pokemonEnPos(c, pGo) y cantMovParaCap(c, pGo)  
 p.contador]

### 3.1. Algoritmos

---



---

**iCrearJuego**(in  $m : \text{map}$ )  $\rightarrow$  res : estr

```

estr : j                                     ▷ O(1)
j.pokemones ← CrearDiccionario()             ▷ O(1)
j.todosLosPokemones ← Vacio()                 ▷ O(1)
j.idsJugadores ← Vacio()                     ▷ O(1)
j.jugadores ← Vacio()                       ▷ O(1)
j.expulsados ← Vacio()                      ▷ O(1)
j.jugadoresPorID ← Vacia()                   ▷ O(1)
j.posicionesPokemons ← Vacio(Largo(m), Ancho(m)) ▷ O(largo(m) x ancho(m))
j.posicionesJugadores ← Vacio(Largo(m), Ancho(m)) ▷ O(largo(m) x ancho(m))
j.mapa ← m                                   ▷ O(copy(m))
res ← j                                     ▷ O(1)

```

Complejidad:  $O((\text{largo}(m) \times \text{ancho}(m)) + \text{copy}(m))$

Justificación:

---



---



---

**iAgregarPokemon**(in/out  $j : \text{estr}$ , in  $c : \text{coor}$ , in  $p : \text{Pokemon}$ )  $\rightarrow$  res : itConj(Pokemon)

```

infoP ← <p, c, 0, Vacio(), True>                                     ▷ O(|P|)
itPokemon ← AgregarRapido(j.todosLosPokemones, infoP)             ▷ O(copy(infoP)) Copiar los elementos de la tupla es
O(1)
desdeLat ← iDamePos(iLatitud(c), 2)                                ▷ O(1)
desdeLong ← iDamePos(iLongitud(c), 2)                              ▷ O(1)
while desdeLat ≤ Latitud(c) + 2 do                                  ▷ O(1) Recorro un conjunto acotado de coordenadas
  while desdeLong ≤ Longitud(c) + 2 do                              ▷ O(1) Recorro un conjunto acotado de coordenadas
    if DistEuclidea(CrearCoor(desdeLat, desdeLong), c) ≤ 4 then    ▷ O(1)
      if Definido?(j.posicionesJugadores, CrearCoor(desdeLat, desdeLong)) then ▷ O(1)
        itJugadores ← iCrearIt(*(Significado(j.posicionesJugadores, CrearCoor(desdeLat, desdeLong)))) ▷
O(1)
        while HaySiguiente(itJugadores) do                          ▷ O(EC)
          e ← Siguiente(itJugadores)                                ▷ O(1)
          tupJugId ← j.jugadoresPorID[e]                            ▷ O(1)
          cantPokemonesJug ← Cardinal(Siguiente(tupJugId1).pokemonesCapturados) ▷ O(1)
          itCola ← iEncolar(Siguiente(itPokemon).jugadoresEnRango, cantPokemonesJug, e) ▷ O(log(EC))
+ copy(e)
          tupJugId2 ← itCola                                         ▷ O(1)
          Avanzar(itJugadores)                                       ▷ O(1)
        end while
      end if
    end if
  end while
end while
if Definido?(j.pokemones, p) then                                  ▷ O(|P|)
  cViejo ← Obtener(j.pokemones, p)                                ▷ O(|P|)
  Eliminar(j.pokemones, p)                                         ▷ O(|P|)
  Definir(j.pokemones, p, cViejo + 1)                              ▷ O(|P|)
else
  Definir(j.pokemones, p, 1)                                       ▷ O(|P|)
end if
res ← itPokemon                                                    ▷ O(1)

```

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

Justificación:

---

---

**iAgregarJugador**(in/out  $j$  : **estr**)  $\rightarrow$  res : Jugador

id  $\leftarrow$  Cardinal(j.jugadores) + Cardinal(j.expulsados)  $\triangleright O(1)$   
itConjIds  $\leftarrow$  AgregarRapido(j.idsJugadores, id)  $\triangleright O(1)$   
infoJ  $\leftarrow$  <itConjIds, false, 0, Vacio(), CrearCoor(0, 0), CrearIt(Vacio())>  $\triangleright O(1)$  El jugador es creado  
vacío, sin sanciones y sin pokemones atrapados. La coordenada es una cualquiera, ya que solo se va a acceder a ella cuando el jugador este conectado, y al conectarse, se le asigna una válida  
itJ  $\leftarrow$  AgregarRapido(j.jugadores, infoJ)  $\triangleright O(\text{copy}(\text{infoJ}))$  Se puede utilizar AgregarRapido porque infoJ contiene un iterador al id del jugador, que es unívoco  
AgregarAtras(j.jugadoresPorID, <itJ, NULL>)  $\triangleright O(J)$  Donde J es la cantidad total de jugadores que fueron agregados al juego  
res  $\leftarrow$  id  $\triangleright O(1)$

Complejidad:  $O(J)$  donde J es la cantidad de jugadores

Justificación:  $O(\text{copy}(\text{Jugador}))$  es igual a  $O(1)$  ya que solamente es copiar Nat, Bool y un conjunto vacío.

---



---

**iConectarse**(in/out  $j$  : **estr**, in  $e$  : Jugador, in  $c$  : coor)

itJug  $\leftarrow$  j.jugadoresPorId[e]<sub>1</sub>  $\triangleright O(1)$   
infoJ  $\leftarrow$  Siguiente(itJug)  $\triangleright O(1)$   
infoJ.estaConectado  $\leftarrow$  true  $\triangleright O(1)$   
AgregarJugadorEnPos(j.posicionesJugadores, infoJ, c)  $\triangleright O(1)$   
infoJ.estaConectado  $\leftarrow$  true  $\triangleright O(1)$   
**if** HayPokemonCercano(j, c) **then**  $\triangleright O(1)$   
    p  $\leftarrow$  Siguiente(Significado(j.posicionesPokemons, PosPokemonCercano(j, c)))  $\triangleright O(1)$   
    itJug  $\leftarrow$  Encolar(p.jugadoresEnRango, Cardinal(jug.pokeCapturados), itJug)  $\triangleright O(\log(EC))$   
    p.contador  $\leftarrow$  0  $\triangleright O(1)$   
**end if**

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

Justificación: EC es la máxima cantidad de jugadores esperando para atrapar un pokemon. En el peor caso, el heap al que entra el jugador es el que mas jugadores esperando tiene.

---



---

**iAgregarJugadorEnPos**(in/out  $d$  : DiccMat(coor, puntero(conjLineal(Jugador))), in/out  $j$  : infoJugador, in  $c$  : coor)

**Pre**  $\equiv \{ d = d_0 \wedge \text{longitud}(c) < \text{ancho}(d) \wedge \text{latitud}(c) < \text{largo}(d) \}$

conLineal(Jugador) : jugsEnPos  $\triangleright O(1)$   
**if**  $\neg$  Definido?(d, c) **then**  $\triangleright O(1)$   
    Definir(d, c, &Vacio())  $\triangleright O(1)$   
**end if**  
jugsEnPos  $\leftarrow$  \*(Significado(d, c))  $\triangleright O(1)$   
j.itPosicion  $\leftarrow$  AgregarRapido(jugsEnPos, j.id)  $\triangleright O(1)$   
j.posicion  $\leftarrow$  c  $\triangleright O(1)$   
Complejidad:  $O(1)$

**Post**  $\equiv \{ d = \text{definir}(\text{longitud}(c), \text{latitud}(c), \text{Ag}(j, \text{significado}(\text{longitud}(c), \text{latitud}(c)))) \}$

---



---



---

**iDesconectarse**(in/out  $j$ : estr, in  $id$ : Jugador)

tupJug $\leftarrow$ j.jugadoresPorId[e]	$\triangleright O(1)$
<b>if</b> HayPokemonCercano(j, c) <b>then</b>	$\triangleright O(1)$
tupJug <sub>2</sub> $\leftarrow$ EliminarSiguiente(tupJug <sub>2</sub> )	$\triangleright O(\log(EC))$
<b>end if</b>	
infoJ $\leftarrow$ Siguiente(tupJug <sub>1</sub> )	$\triangleright O(1)$
infoJ.estaConectado $\leftarrow$ false	$\triangleright O(1)$
EliminarSiguiente(infoJ.itPosicion)	$\triangleright O(1)$
infoJ.itPosicion $\leftarrow$ CrearIt(Vacio())	$\triangleright O(1)$

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

Justificación: EC es la maxima cantidad de jugadores esperando para atrapar un pokemon. En el peor caso, el heap del que sale el jugador es el que mas jugadores esperando tiene.

---

---

```

iMoverse(in/out  $j$ : estr, in  $id$ : Jugador, in  $c$ : coor)
  tupJug  $\leftarrow$  j.jugadoresPorID[id]  $\triangleright O(1)$ 
  infoJ  $\leftarrow$  Siguiente(tupJug1)  $\triangleright O(1)$ 
  if iHayPokemonCercano( $c$ ,  $j$ ) then  $\triangleright O(1)$  Pregunto si me estoy moviendo a donde hay un pokemon
     $p \leftarrow$  iPosPokemonCercano( $j$ ,  $c$ )  $\triangleright O(1)$ 
    if  $\neg$  iHayPokemonCercano(infoJ.posicion,  $j$ ) then  $\triangleright O(1)$  Pregunto si no estaba en rango de un pokemon antes
      de moverme
        infoP  $\leftarrow$  Siguiente(Significado(j.posicionesPokemons,  $p$ ))  $\triangleright O(1)$ 
        infoP.contador  $\leftarrow$  0  $\triangleright O(1)$ 
        cantPokemonesJug  $\leftarrow$  Cardinal(infoJ.pokemonesCapturados)  $\triangleright O(1)$ 
        tupJugId2  $\leftarrow$  iEncolar(infoP.jugadoresEnRango, cantPokemonesJug, id)  $\triangleright O(\log(EC))$ 
      else
        if  $p \neq$  iPosPokemonCercano( $j$ , infoJ.posicion) then  $\triangleright O(1)$  Pregunto si el pokemon donde estaba es distinto
          del que voy a estar
            infoP  $\leftarrow$  Siguiente(Significado(j.posicionesPokemons,  $p$ ))  $\triangleright O(1)$ 
            infoP.contador  $\leftarrow$  0  $\triangleright O(1)$ 
            cantPokemonesJug  $\leftarrow$  Cardinal(infoJ.pokemonesCapturados)  $\triangleright O(1)$ 
            EliminarSiguiente(tupJugId2)  $\triangleright O(\log(EC))$ 
            tupJugId2  $\leftarrow$  iEncolar(infoP.jugadoresEnRango, cantPokemonesJug, id)  $\triangleright O(\log(EC))$ 
          end if
        end if
        iActualizarMenos( $p$ ,  $j$ )  $\triangleright O(PS)$ 
      else
        if iHayPokemonCercano(infoJ.posicion,  $j$ ) then  $\triangleright O(1)$  Ya se que no hay pokemon donde voy a estar, si hay
          pokemon donde estoy ahora es que salgo de un rango
            EliminarSiguiente(tupJugId2)  $\triangleright O(\log(EC))$  Elimina al jugador del heap donde estaba esperando
          end if
          iActualizarTodos( $j$ )  $\triangleright O(PS)$ 
        end if
        if debeSancionarse?(id,  $c$ ,  $j$ ) then
          if infoJ.sanciones  $<$  4 then  $\triangleright O(1)$ 
            infoJ.sanciones  $\leftarrow$  infoJ.sanciones + 1  $\triangleright O(1)$ 
            EliminarSiguiente(infoJ.itPosicion)  $\triangleright O(1)$ 
            AgregarJugadorEnPos(j.posicionesJugadores, infoJ,  $c$ )  $\triangleright O(1)$ 
          else
            itPCapturados  $\leftarrow$  CrearIt(infoJ.pokemonesCapturados)  $\triangleright O(1)$ 
            while HaySiguiente(itPCapturados) do  $\triangleright O(PC)$ 
              infoP  $\leftarrow$  Siguiente(Siguiente(itPCapturados))  $\triangleright O(1)$ 
              pViejos  $\leftarrow$  Obtener(j.pokemones, infoP.tipo)  $\triangleright O(|P|)$ 
              Borrar(j.pokemones, infoP.tipo)  $\triangleright O(|P|)$ 
              if  $pViejos \neq 1$  then  $\triangleright O(1)$ 
                Definir(j.pokemones, infoP.tipo,  $pViejos - 1$ )  $\triangleright O(|P|)$ 
              end if
              EliminarSiguiente(Siguiente(itPCapturados))  $\triangleright O(1)$ 
            end while
            EliminarSiguiente(Siguiente(tupJug1)1)  $\triangleright O(1)$  Elimina el id del jugador del conjunto de jugadores
            iEliminarSiguiente(tupJug1)  $\triangleright O(1)$  Saca al jugador del conjunto de jugadores
            AgregarRapido(j.expulsados, id)  $\triangleright O(\text{copy}(\text{id}))$  Copiar un nat es  $O(1)$ 
            if iHayPokemonCercano( $j$ ,  $c$ ) then  $\triangleright O(1)$ 
              iEliminarSiguiente(tupJug2)  $\triangleright O(\log(EC))$  Elimina al jugador del heap donde estaba esperando
            end if
          end if
        end if
      else
        EliminarSiguiente(infoJ.itPosicion)  $\triangleright O(1)$ 
        AgregarJugadorEnPos(j.posicionesJugadores, infoJ,  $c$ )  $\triangleright O(1)$ 
      end if

```

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

Justificación:

---

**iActualizarMenos**(in/out  $j$ : *estr*, in  $c$ : *coor*)

```

p ← iPosPokemonCercano(j, c)                                ▷ O(1)
itCoor ← Coordenadas(j.posicionesPokemons)                 ▷ O(1)
while HaySiguiente(itCoor) do                                ▷ O(PS)
  if Siguiente(itCoor) ≠ p then                                ▷ O(1)
    itPokemon ← Significado(j.posicionesPokemons, Siguiente(itCoor)) ▷ O(1)
    ActualizarPokemon(itPokemon)                               ▷ O(1)
  end if
end while

```

Complejidad: O(PS)

Justificación: Actualiza todos las colas de prioridad excepto por la que este en la posicion que paso por parametro.

---

**iActualizarTodos**(in/out  $j$ : *estr*)

```

itCoor ← Coordenadas(j.posicionesPokemons)                 ▷ O(1)
while HaySiguiente(itCoor) do                                ▷ O(PS)
  itPokemon ← Significado(j.posicionesPokemons, Siguiente(itCoor)) ▷ O(1)
  ActualizarPokemon(itPokemon)                               ▷ O(1)
end while

```

Complejidad: O(PS)

Justificación: Actualiza todos las colas de prioridad. invalido.

---

**iActualizarPokemon**(in/out  $itPokemones$ : *itConj*(*infoPokemon*), in/out  $j$ : *estr*)

```

infoP ← Siguiente(itPokemones)                               ▷ O(1)
infoP.contador ← infoP.contador + 1                           ▷ O(1)
if infoP.contador = 10 then                                    ▷ O(1)
  e ← Proximo(infoP.jugadoresEnRango)                         ▷ O(1)
  infoP.esSalvaje ← false                                       ▷ O(1)
  infoJ ← Siguiente(j.jugadoresPorID[e]1)                     ▷ O(1)
  AgregarRapido(infoJ.pokemonsCapturados, itPokemones)     ▷ O(copy(itPokemones)) Copiar el iterador es O(1)
  Borrar(j.posicionesPokemons, infoP.posicion)                ▷ O(1)
end if

```

Complejidad: O(1)

Justificación: Actualiza la cola de prioridad del pokemon que paso por parametro. Suma uno al contador en caso de que no tenga que atraparse, si tiene que ser atrapado, lo agrega al conjunto de pokemons atrapados del jugador y lo elimina del diccionario de pokemons por posicion.

---

**iDebeSancionarse**(in  $e$ : *Jugador*, in  $c$ : *coor*, in  $j$ : *estr*) → *res*: *Bool*

```

tupJug ← j.jugadoresPorID[e]                                  ▷ O(1)
infoJ ← Siguiente(tupJug1)                                    ▷ O(1)
res ← ¬HayCamino(infoJ.posicion, c, j.mapa) ∨ DistEuclidea(infoJ.posicion, c, j.mapa) > 100 ▷ O(1)

```

Complejidad: O(1)

Justificación: Checkea si el jugador hizo un movimiento invalido.

---

**iMapa**(in  $j$ : *estr*) → *res*: *map*

```

res ← j.mapa                                                  ▷ O(copy(mapa(j)))

```

Complejidad: O(copy(mapa(j)))

Justificación: Devuelve el mapa del juego por copia.

---

**iJugadores**(in  $j$  : **estr**)  $\rightarrow$  res : itConj(Jugador)

 res  $\leftarrow$  CrearIt(j.idsJugadores)

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve el mapa del juego.

---

**iEstaConectado**(in  $j$  : **estr**, in  $id$  : Jugador)  $\rightarrow$  res : Bool

 res  $\leftarrow$  Siguiente(j.jugadoresPorID[id]<sub>0</sub>).estaConectado

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve si el jugador esta conectado.

---

**iPosicion**(in  $j$  : **estr**, in  $id$  : Jugador)  $\rightarrow$  res : coor

 res  $\leftarrow$  Siguiente((j.jugadoresPorID[id]).info).posicion

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve la posicion del jugador.

---

**iPokemones**(in  $j$  : **estr**, in  $id$  : Jugador)  $\rightarrow$  res : itConj(<Pokemon, Nat>)

 res  $\leftarrow$  CrearIt(Siguiente((j.jugadoresPorID[id]<sub>1</sub>).pokeCapturados)

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve un iterador al conjunto de pokemones atrapados por el jugador.

---

**iExpulsados**(in  $j$  : **estr**)  $\rightarrow$  res : itConj(Jugador)

 res  $\leftarrow$  CrearIt(j.expulsados)

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve un iterador al conjunto de jugadores expulsados.

---

**iPosConPokemones**(in  $j$  : **estr**)  $\rightarrow$  res : itConj(coor)

 res  $\leftarrow$  CrearIt(Coordenadas(j.posicionesPokemons))

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve las posiciones con pokemones.

---

**iPokemonEnPos**(in  $j$  : **estr**, in  $c$  : coor)  $\rightarrow$  res : Pokemon

 res  $\leftarrow$  Significado(j.posicionesPokemons, c)<sub>1</sub>

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve el mapa del juego.

---

**iCantMovimientosParaCaptura**(in  $j$  : **estr**, in  $c$  : coor)  $\rightarrow$  res : Nat

 res  $\leftarrow$  10 - Siguiente(Significado(j.posicionesPokemons, c)).contador

▷ O(1)

Complejidad: O(1)

Justificación: Devuelve cuantos movimientos faltan para capturar al pokemon.

---



---

**iPosPokemonCercano**(in  $j$ : estr, in  $c$ : coor)  $\rightarrow$  res : coor

```

i ← 0                                ▷ O(1)
latC ← Latitud(c)                     ▷ O(1)
i ← DamePos(latC, 2)                   ▷ O(1)
longC ← Longitud(c)                   ▷ O(1)
j ← DamePos(longC, 2)                  ▷ O(1)
while i ≤ latC + 2 do                 ▷ O(1) Vale porque estoy recorriendo un conjunto acotado de coordenadas
  while j ≤ longC + 2 do               ▷ O(1) Vale porque estoy recorriendo un conjunto acotado de coordenadas
    if Definido?(j.posicionesPokemons, <i, j>) ∧ DistEuclidea(c, <i, j>) ≤ 4 then   ▷ O(1)
      res ← <i, j>                      ▷ O(1)
    end if
    j ← j + 1                           ▷ O(1)
  end while
  i ← i + 1                             ▷ O(1)
end while

```

Complejidad: O(1)

Justificación: Como el rango a recorrer es una constante, se puede decir que es de la clase  $\Theta(1)$

---



---



---

**iPuedoAgregarPokemon**(in  $j$ : estr, in  $c$ : coor)  $\rightarrow$  res : Bool

```

res ← PosExistente(c, j.mapa) ∧ ¬(Definido?(j.posicionesPokemons, c)) ∧ ¬(HayPokemonEnTerritorio(j, c))  ▷
 $\Theta \left( \sum_{c' \in coordendas(mapa(j))} equal(c, c') \right)$ 

```

Complejidad:  $\left[ \Theta \left( \sum_{c' \in coordendas(mapa(j))} equal(c, c') \right) \right]$

Justificación: Tiene que ver si la posicion existe en el mapa, las demas operaciones son O(1)

---



---



---

**iHayPokemonCercano**(in  $j$ : estr, in  $c$ : coor)  $\rightarrow$  res : Bool

```

res ← false                            ▷ O(1)
latC ← Latitud(c)                       ▷ O(1)
i ← DamePos(latC, 2)                     ▷ O(1)
longC ← Longitud(c)                     ▷ O(1)
j ← DamePos(longC, 2)                    ▷ O(1)
while i ≤ latC + 2 do                   ▷ O(1) Vale porque estoy recorriendo un conjunto acotado de coordenadas
  while j ≤ longC + 2 do                 ▷ O(1) Vale porque estoy recorriendo un conjunto acotado de coordenadas
    if Definido?(j.posicionesPokemons, <i, j>) ∧ DistEuclidea(c, <i, j>) ≤ 4 then   ▷ O(1)
      res ← true                          ▷ O(1)
    end if
    j ← j + 1                             ▷ O(1)
  end while
  i ← i + 1                             ▷ O(1)
end while

```

Complejidad:  $\Theta(1)$

Justificación: Como el rango a recorrer es una constante, se puede decir que es de la clase  $\Theta(1)$

---

---

**iHayPokemonEnTerritorio**(in  $j$ : estr, in  $c$ : coor)  $\rightarrow$  res : Bool

```

res  $\leftarrow$  false  $\triangleright O(1)$ 
latC  $\leftarrow$  Latitud(c)  $\triangleright O(1)$ 
i  $\leftarrow$  DamePos(latC, 5)  $\triangleright O(1)$ 
longC  $\leftarrow$  Longitud(c)  $\triangleright O(1)$ 
j  $\leftarrow$  DamePos(longC, 5)  $\triangleright O(1)$ 
while i  $\leq$  latC + 5 do  $\triangleright O(1)$  Vale porque estoy recorriendo un conjunto acotado de coordenadas
  while j  $\leq$  longC + 5 do  $\triangleright O(1)$  Vale porque estoy recorriendo un conjunto acotado de coordenadas
    if Definido?(j.posicionesPokemons, <i, j>)  $\wedge$  DistEuclidea(c, <i, j>)  $\leq$  25 then  $\triangleright O(1)$ 
      res  $\leftarrow$  true  $\triangleright O(1)$ 
    end if
    j  $\leftarrow$  j + 1  $\triangleright O(1)$ 
  end while
  i  $\leftarrow$  i + 1  $\triangleright O(1)$ 
end while

```

Complejidad:  $\Theta(1)$

Justificación: Como el rango a recorrer es una constante, se puede decir que es de la clase  $\Theta(1)$

---



---

**iEntrenadoresPosibles**(in  $c$ : coor, in  $es$ : conjLineal(Jugador), in  $j$ : estr)  $\rightarrow$  res : conjLineal(itConj(Jugador))

```

ePosibles  $\leftarrow$  Vacia()  $\triangleright O(1)$  Crea un conjunto de iteradores vacio
if Cardinal(es)  $\neq$  0 then  $\triangleright O(1)$ 
  itE  $\leftarrow$  CreaIt(es)  $\triangleright O(1)$ 
  while HaySiguiente(itE) do  $\triangleright O(Cardinal(es))$  Es la cantidad de jugadores que haya en el conjunto es
    e  $\leftarrow$  Siguiente(itE)  $\triangleright O(1)$ 
    infoJ  $\leftarrow$  Siguiente(j.jugadoresPorID[e].info)  $\triangleright O(1)$ 
    if infoJ.estaConectado then  $\triangleright O(1)$ 
      posJugador  $\leftarrow$  infoJ.posicion  $\triangleright O(1)$ 
      if iHayPokemonCercano(posJugador, j) then  $\triangleright O(1)$ 
        if iPosPokemonCercano(posJugador, j) = c then  $\triangleright O(1)$ 
          if iHayCamino(c, posJugador, Mapa(j)) then  $\triangleright O(1)$ 
            AgregarRapido(ePosibles, e)  $\triangleright O(copy(e))$  Copiar un Nat es  $O(1)$ 
          end if
        end if
      end if
    end if
    Avanzar(itE)  $\triangleright O(1)$ 
  end while
end if
res  $\leftarrow$  ePosibles  $\triangleright O(1)$ 

```

Complejidad:  $O(\#(es))$

Justificación: Se itera por completo el conjunto de jugadores 'es'. En peor caso, todos los elementos de 'es' deben ser agregados al resultado.

---



---

**iIndiceRareza**(in  $j$ : estr, in  $p$ : Pokemon)  $\rightarrow$  res : Nat

```

cuantosP  $\leftarrow$  iCantMismaEspecie(j, p)  $\triangleright O(|p.tipo|)$ 
res  $\leftarrow$  100 - (100 x cuantosP / iCantPokemonesTotales)  $\triangleright O(1)$ 

```

Complejidad:  $O(|P|)$

Justificación: Siendo  $|P|$  el nombre mas largo para un pokemon en el juego.

---

---



---

**iCantPokemonesTotales**(in  $j$  : **estr**)  $\rightarrow$  res : Nat

 res  $\leftarrow$  Cardinal(j.todosLosPokemones)
 $\triangleright O(1)$ Complejidad:  $O(1)$ Justificación: Pide el cardinal de un conjunto.

---



---

**iCantMismaEspecie**(in  $j$  : **estr**, in  $p$  : **Pokemon**)  $\rightarrow$  res : Nat

 if Definido?(j.pokemones, p.tipo) **then**
 $\triangleright O(|P|)$ 
 res  $\leftarrow$  Longitud(Obtener(j.pokemones, p.tipo))
 $\triangleright O(|P|)$ **else**
 res  $\leftarrow$  0
 $\triangleright O(1)$ **end if**Complejidad:  $O(|P|)$ Justificación: En peor caso, el pokemon que se busca es el de nombre mas largo o no esta en el diccionario.

---



---

**DamePos**(in  $p$  : Nat, in  $step$  : Nat)  $\rightarrow$  res : Nat

 if  $p \geq step$  **then**
 $\triangleright O(1)$ 
 res  $\leftarrow$   $p - step$ 
 $\triangleright O(1)$ **else**
 res  $\leftarrow$  0
 $\triangleright O(1)$ **end if**Complejidad:  $O(1)$ Justificación: Aritmetica de naturales.

## 4. Modulo Diccionario Matriz( $coor, \sigma$ )

El modulo Diccionario Matriz provee un diccionario por posiciones en el que se puede definir, y consultar si hay un valor en una posicion en tiempo  $O(copy(\sigma))$ . Ademas, se puede borrar en tiempo lineal sobre las dimensiones de la matriz, y obtener un iterador a un conjunto lineal de claves.

El principal costo se paga al crear la estructura o borrar un dato, dado que cuesta tiempo lineal *ancho* por *largo*.

### Interfaz

#### parametros formales

**generos**  $coor, \sigma$

**se explica con:**  $DICCMAT(Nat, Nat, \sigma)$ ,

**generos:**  $diccMat(coor, \sigma)$ .

**VACIO**(**in**  $Nat$ :  $l$  argo, **in**  $Nat$ :  $a$  ncho)  $\rightarrow res : diccMat(coor, \sigma)$

**Pre**  $\equiv \{largo * ancho > 0\}$

**Post**  $\equiv \{res =_{obs} vacio(largo, ancho)\}$

**Complejidad:**  $\Theta(ancho * largo)$

**Descripción:** Genera un diccionario vacio, de tamaño  $ancho * largo$ .

**DEFINIR**(**in/out**  $d : diccMat(coor, \sigma)$ , **in**  $c : coor$ , **in**  $s : \sigma$ )

**Pre**  $\equiv \{d =_{obs} d_0 \wedge enRango(c_1, c_2, d)\}$

**Post**  $\equiv \{d =_{obs} definir(c_1, c_2, s, d_0)\}$

**Complejidad:**  $\Theta(copy(s))$

**Descripción:** define el significado  $s$  en el  $diccMat$ , en la posicion representada por  $c$ .

**DEFINIDO?**(**in**  $d : diccMat(coor, \sigma)$ , **in**  $c : coor$ )  $\rightarrow res : bool$

**Pre**  $\equiv \{enRango(c_1, c_2, d)\}$

**Post**  $\equiv \{res =_{obs} def?(c_1, c_2, d)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** devuelve **true** si y solo si  $c$  tiene un valor en el  $diccMat$ .

**SIGNIFICADO**(**in**  $d : diccMat(coor, \sigma)$ , **in**  $c : coor$ )  $\rightarrow res : \sigma$

**Pre**  $\equiv \{enRango(c_1, c_2, d) \wedge_L def?(c_1, c_2, d)\}$

**Post**  $\equiv \{alias(res =_{obs} obtener(c_1, c_2, d))\}$

**Complejidad:**  $\Theta(copy(s))$

**Descripción:** Devuelve el valor de  $d$  en la posicion  $c$ .

**BORRAR**(**in/out**  $d : diccMat(coor, \sigma)$ , **in**  $c : coor$ )

**Pre**  $\equiv \{d = d_0 \wedge enRango(c_1, c_2, d) \wedge_L def?(c_1, c_2, d)\}$

**Post**  $\equiv \{d =_{obs} borrar(c_1, c_2, d_0)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Elimina el valor en la posicion  $c$  en  $d$ .

**COORDENADAS**(**in**  $d : diccMat(coor, \sigma)$ )  $\rightarrow res : itConj(coor)$

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{alias(esPermutacion?(SecuSuby(res), claves(d)))\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve un iterador al conjunto de claves de  $d$ .

**ANCHO**(**in**  $d : diccMat(coor, \sigma)$ )  $\rightarrow res : Nat$

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{res =_{obs} ancho(d)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve el ancho de  $d$

**LARGO**(**in**  $d : diccMat(coor, \sigma)$ )  $\rightarrow res : Nat$

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{res =_{obs} largo(d)\}$

**Complejidad:**  $\Theta(1)$



**Descripción:** Devuelve el largo de  $d$

## 4.0.1. Especificacion de las operaciones auxiliares utilizadas en la interfaz

TAD DICCMATRIZ(NAT, NAT,  $\sigma$ )**géneros**       $\text{diccMat}(\text{Nat}, \text{Nat}, \sigma)$ **exporta**       $\text{diccMat}(\text{Nat}, \text{Nat}, \sigma)$ , generadores, observadores, borrar, claves**usa**          NAT, BOOL, CONJ(TUPLA(NAT, NAT))**igualdad observacional**

$$(\forall d, d' : \text{DiccMat}(\text{Nat}, \text{Nat}, \sigma)) \left( d =_{\text{obs}} d' \iff \left( \begin{array}{l} (\text{ancho}(d) =_{\text{obs}} \text{ancho}(d') \wedge \text{largo}(d) =_{\text{obs}} \text{largo}(d')) \wedge_{\text{L}} \\ (\forall x, y : \text{Nat}) (\text{def?}(x, y, d) =_{\text{obs}} \text{def?}(x, y, d')) \wedge_{\text{L}} \\ \text{def?}(x, y, d) \Rightarrow_{\text{L}} \text{obtener}(x, y, d) =_{\text{obs}} \text{obte-} \\ \text{ner}(x, y, d') \end{array} \right) \right)$$

**observadores básicos**

$\text{largo} : \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) \longrightarrow \text{Nat}$   
 $\text{ancho} : \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) \longrightarrow \text{Nat}$   
 $\text{def?} : \text{Nat } x \times \text{Nat } y \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d \longrightarrow \text{Bool} \quad \{\text{enRango}(x, y, d)\}$   
 $\text{obtener} : \text{Nat } x \times \text{Nat } y \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d \longrightarrow \sigma \quad \{\text{enRango}(x, y, d) \wedge_{\text{L}} \text{def?}(x, y, d)\}$

**generadores**

$\text{vacío} : \text{Nat } \text{largo} \times \text{Nat } \text{ancho} \longrightarrow \text{diccMat}(\text{Nat}, \text{Nat}, \sigma) \quad \{\text{largo} * \text{ancho} > 0\}$   
 $\text{definir} : \text{Nat } x \times \text{Nat } y \times \sigma s \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d \longrightarrow \text{diccMat}(\text{Nat}, \text{Nat}, \sigma) \quad \{\text{enRango}(x, y, d)\}$

**otras operaciones**

$\text{borrar} : \text{Nat } x \times \text{Nat } y \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d \longrightarrow \text{diccMat}(\text{Nat}, \text{Nat}, \sigma) \quad \{\text{enRango}(x, y, d) \wedge_{\text{L}} \text{def?}(x, y, d)\}$   
 $\text{claves} : \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) \longrightarrow \text{conj}(\text{tupla}(\text{Nat}, \text{Nat}))$

**otras operaciones (no exportadas)**

$\text{enRango} : \text{Nat} \times \text{Nat} \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \text{sinificado}) \longrightarrow \text{Bool}$

**axiomas**       $\forall x, y, m, n : \text{Nat} \forall d : \text{diccMat}(\text{Nat}, \text{Nat}, \sigma) \forall s : \sigma$  $\text{largo}(\text{vacío}(m, n)) \equiv m$  $\text{ancho}(\text{vacío}(m, n)) \equiv n$  $\text{def?}(x, y, \text{vacío}(m, n)) \equiv \text{false}$  $\text{largo}(\text{definir}(x, y, s, d)) \equiv \text{largo}(d)$  $\text{ancho}(\text{definir}(x, y, s, d)) \equiv \text{ancho}(d)$  $\text{def?}(x, y, \text{definir}(m, n, s, d)) \equiv (x = m \wedge y = n) \vee \text{def?}(x, y, d)$  $\text{obtener}(x, y, \text{definir}(m, n, s, d)) \equiv \text{if } (x = m \wedge y = n) \text{ then } s \text{ else obtener}(x, y, d) \text{ fi}$

```

borrar(x,y, definir(m,n,s,d))  $\equiv$  if (x = m  $\wedge$  y = n) then
    if def?(x,y,d) then borrar(x,y,d) else d fi
    else
        definir(m,n,s, borrar(x,y,d))
    fi

claves(vacio)  $\equiv \emptyset$ 

claves(definir(x,y,s,d))  $\equiv$  Ag((x,y),claves(d))

enRango(x,y, d)  $\equiv$  x < largo(d)  $\wedge$  y < ancho(d)

```

**Fin TAD**

## Representación

Diccionario Matriz se representa con dicc

donde dicc es  $\text{tupla}(\text{posiciones: arregloDimensionable de } \langle \text{bool}, \text{itConj}(\sigma), \sigma \rangle, \text{claves: conjLineal}(\text{coor}), \text{ancho: Nat}, \text{largo: Nat})$

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

$\text{Rep}(d) \equiv \text{true} \iff (\text{tam}(d.\text{posiciones}) =_{\text{obs}} (d.\text{ancho} \times d.\text{largo})) \wedge_L ((\forall c : \text{coor}) (c \in d.\text{claves} \iff \text{definido?}(d.\text{posiciones}, \text{latitud}(c) \times d.\text{ancho} + \text{longitud}(c)) \wedge_L (d.\text{posiciones}[\text{latitud}(c) \times d.\text{ancho} + \text{longitud}(c)]_1 =_{\text{obs}} \text{true})) \wedge_L ((\forall c : \text{coor}) (c \in d.\text{claves} \iff (\text{Siguiete}(d.\text{posiciones}[\text{latitud}(c) \times d.\text{ancho} + \text{longitud}(c)]_2) =_{\text{obs}} c)))$

$\text{Abs} : \text{dicc } d \rightarrow \text{diccMat} \quad \{\text{Rep}(d)\}$

$\text{Abs}(d) \equiv =_{\text{obs}} d' : \text{diccMat} \mid d.\text{claves} = \text{claves}(d') \wedge d.\text{ancho} = \text{ancho}(d') \wedge d.\text{largo} = \text{largo}(d') \wedge (\forall c \leftarrow d.\text{claves}) d.\text{posiciones}[\text{longitud}(c) * d.\text{ancho} + \text{latitud}(c)] = \text{significado}(\text{longitud}(c), \text{latitud}(c), d')$

## Algoritmos

Trabajo Práctico II Algoritmos del modulo

---

**iVacio**(in  $l : \text{Nat}$ , in  $a : \text{Nat}$ )  $\rightarrow res : \text{dicc}$

```

1: res.largo  $\leftarrow l$   $\triangleright \Theta(1)$ 
2: res.ancho  $\leftarrow a$   $\triangleright \Theta(1)$ 
3: res.posiciones  $\leftarrow \text{CrearArreglo}(a * l)$   $\triangleright \Theta(a * l)$ 
4: res.claves  $\leftarrow \text{Vacio}()$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(a * l)$

---

---

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

```

1: sig ← tupla                                ▷ O(1)
2: sig1 ← true                                ▷ O(1)
3: sig3 ← s                                    ▷ Θ(copy(s))
4: if ¬ (Definido?(d, Aplanar(d,c))) then
5:   sig2 ← AgregarRapido(d.claves, c)          ▷ Θ(copy(s))
6: else
7:   sig2 ← d.posiciones[Aplanar(d, c)]2      ▷ O(1)
8: end if
9: d.posiciones[Aplanar(d, c)] ← sig           ▷ Θ(copy(s))

```

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

Justificación: *Definido?* y *Aplanar* tienen costo  $\Theta(1)$ , *AgregarRapido* y *Definir* tienen costo  $\Theta(\text{copy}(s))$ . Aplicando algebra de ordenes:  $\Theta(1) + \Theta(1) + \Theta(\text{copy}(s)) + \Theta(\text{copy}(s)) = \Theta(\text{copy}(s))$

---



---

**iDefinido?**(in  $d$ : dicc, in  $c$ : coor) →  $res$ : bool

```

1: res ← Definido?(d.posiciones, Aplanar(d, c)) ∧L d.posiciones[Aplanar(d, c)]1    ▷ Si no esta definido o esta
   marcado como borrado, se devuelve que no esta definido Θ(1)

```

Complejidad:  $\Theta(1)$

Justificación: *Aplanar* tiene costo  $\Theta(1)$ , luego, como *Definido?* y consultar una posicion de un arreglo tienen costo  $\Theta(1)$ . Aplicando algebra de ordenes:  $\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

---



---

**iSignificado**(in  $d$ : dicc, in  $c$ : coor) →  $res$ :  $\sigma$ 

```

1: res ← d.posiciones[Aplanar(d, c)]                                ▷ Θ(1)

```

Complejidad:  $\Theta(1)$

---



---

**iBorrar**(in/out  $d$ : dicc, in  $c$ : coor)

```

1: EliminarSiguiete(d.posiciones[Aplanar(d, c)]2)                    ▷ Θ(1)
2: d.posiciones[Aplanar(d, c)] ←< false, CrearIt(Vacio()), d.posiciones[Aplanar(d, c)] >    ▷ Θ(1)

```

Complejidad:  $\Theta(1)$

---



---

**iCoordenadas**(in  $d$ : dicc) →  $res$ :  $itConj(\text{coor})$ 

```

1: res ← CrearIt(d.claves)                                            ▷ Θ(1)

```

Complejidad:  $\Theta(1)$

---



---

**iAplanar**(in  $d$ : dicc, in  $c$ : coor) →  $res$ : nat

```

1: res ← Longitud(c) * d.ancho + Latitud(c)                          ▷ Θ(1)

```

Complejidad:  $\Theta(1)$

Justificación: Son operaciones matematicas de Nat

---



---

**iLargo**(in  $d$ : dicc) →  $res$ : nat

```

1: res ← d.largo                                                      ▷ Θ(1)

```

Complejidad:  $\Theta(1)$

---

---

**iAncho**(in  $d$ : *dicc*)  $\rightarrow res$ : *nat*1:  $res \leftarrow d.ancha$  $\triangleright \Theta(1)$ Complejidad:  $\Theta(1)$ 

---

## 5. Módulo Cola de mínima prioridad( $\alpha$ )

### 5.1. Especificación

**TAD COLA DE MÍNIMA PRIORIDAD( $\alpha$ )**

**igualdad observacional**

$$(\forall c, c' : \text{colaMinPrior}(\alpha)) \left( c =_{\text{obs}} c' \iff \left( \begin{array}{l} \text{vacía?}(c) =_{\text{obs}} \text{vacía?}(c') \wedge_L \\ (\neg \text{vacía?}(c) \Rightarrow_L (\text{próximo}(c) =_{\text{obs}} \text{próximo}(c') \wedge \\ \text{desencolar}(c) =_{\text{obs}} \text{desencolar}(c'))) \end{array} \right) \right)$$

**parámetros formales**

**géneros**  $\alpha$

**operaciones**  $\bullet < \bullet : \alpha \times \alpha \longrightarrow \text{bool}$

**géneros**  $\text{colaMinPrior}(\alpha)$

**exporta**  $\text{colaMinPrior}(\alpha)$ , generadores, observadores

**usa** **BOOL**

**observadores básicos**

$\text{vacía?} : \text{colaMinPrior}(\alpha) \longrightarrow \text{bool}$

$\text{próximo} : \text{colaMinPrior}(\alpha) \ c \longrightarrow \alpha \quad \{\neg \text{vacía?}(c)\}$

$\text{desencolar} : \text{colaMinPrior}(\alpha) \ c \longrightarrow \text{colaMinPrior}(\alpha) \quad \{\neg \text{vacía?}(c)\}$

**generadores**

$\text{vacía} : \longrightarrow \text{colaMinPrior}(\alpha)$

$\text{encolar} : \alpha \times \text{colaMinPrior}(\alpha) \longrightarrow \text{colaMinPrior}(\alpha)$

**otras operaciones**

$\text{tamaño} : \text{colaMinPrior}(\alpha) \longrightarrow \text{nat}$

**axiomas**  $\forall c : \text{colaMinPrior}(\alpha), \forall e : \alpha$

$\text{vacía?}(\text{vacía}) \equiv \text{true}$

$\text{vacía?}(\text{encolar}(e, c)) \equiv \text{false}$

$\text{próximo}(\text{encolar}(e, c)) \equiv \text{if } \text{vacía?}(c) \vee_L \text{próximo}(c) > e \text{ then } e \text{ else } \text{próximo}(c) \text{ fi}$

$\text{desencolar}(\text{encolar}(e, c)) \equiv \text{if } \text{vacía?}(c) \vee_L \text{próximo}(c) > e \text{ then } c \text{ else } \text{encolar}(e, \text{desencolar}(c)) \text{ fi}$

**Fin TAD**

### 5.2. Interfaz

**parámetros formales**

**géneros**  $\alpha$

**se explica con:** COLA DE MÍNIMA PRIORIDAD(NAT).

**géneros:**  $\text{colaMinPrior}(\alpha)$ .

### 5.2.1. Operaciones básicas de ColaMinPrior

VACÍA()  $\rightarrow res : colaMinPrior(\alpha)$

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{res =_{obs} vacía\}$

**Complejidad:**  $O(1)$

**Descripción:** Crea una cola de prioridad vacía

VACÍA?(in  $c : colaMinPrior(\alpha)$ )  $\rightarrow res : bool$

**Pre**  $\equiv \{true\}$

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

**Complejidad:**  $O(1)$

**Descripción:** Devuelve true si y sólo si la cola está vacía

PRÓXIMO(in  $c : colaMinPrior(\alpha)$ )  $\rightarrow res : \alpha$

**Pre**  $\equiv \{\neg vacía?(c)\}$

**Post**  $\equiv \{alias(res =_{obs} próximo(c))\}$

**Complejidad:**  $O(1)$

**Descripción:** Devuelve el próximo elemento a desencolar

**Aliasing:**  $res$  es modificable si y sólo si  $c$  es modificable

DESENCOLAR(in/out  $c : colaMinPrior(\alpha)$ )

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

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

**Complejidad:**  $O(\log(\text{tamaño}(c)))$

**Descripción:** Quita el elemento más prioritario

ENCOLAR(in/out  $c : colaMinPrior(\alpha)$ , in  $p : nat$ , in  $a : \alpha$ )  $\rightarrow res : itColaMin(\alpha)$

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

**Post**  $\equiv \{c =_{obs} encolar(p, c_0) \wedge res =_{obs} CrearIt(ColaASecu(c_0), a) \wedge alias(SecuSuby(res) = ColaASecu(c))\}$

**Complejidad:**  $O(\log(|c|)) + copy(a)$

**Descripción:** Agrega el elemento  $a$  de tipo  $\alpha$  con prioridad  $p$  a la cola

**Aliasing:** Se agrega el elemento por copia

### 5.2.2. Operaciones del Iterador

CREARIT(in  $c : colaMinPrior(\alpha)$ )  $\rightarrow res : itColaMin(\alpha)$

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{Alias(EsPermutacion(SecuSuby(res), c))\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Crea un iterador de Cola Mínima de Prioridad( $\alpha$ )

**Aliasing:** El iterador se invalida si y solo si se elimina el elemento siguiente del iterador

HAYSIGUIENTE?(in  $it : itColaMin(\alpha)$ )  $\rightarrow res : bool$

**Pre**  $\equiv \{true\}$

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

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve true si y solo si en el iterador todavía quedan elementos para avanzar

SIGUIENTE(in  $it : itColaMin(\alpha)$ )  $\rightarrow res : \alpha$

**Pre**  $\equiv \{haySiguiente?(it)\}$

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

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve el elemento de la siguiente posición del iterador

**Aliasing:**  $res$  es modificable si y solo si  $it$  es modificable

ELIMINARSIGUIENTE(in/out  $it : itColaMin(\alpha)$ )

**Pre**  $\equiv \{it =_{obs} it_0 \wedge haySiguiente?(it)\}$

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

**Complejidad:**  $\Theta(1)$

**Descripción:** Elimina de la cola el valor que se encuentra en la posición siguiente del iterador.

### 5.3. Representación

#### 5.3.1. Representación de ColaMinPrior( $\alpha$ )

ColaMinPrior( $\alpha$ ) se representa con *estr*

donde *estr* es *tupla*(*proximo*: puntero(nodo), *tamano*: nat)

donde *nodo* es *tupla*(*prior*: Nat, *elem*:  $\alpha$ , *padre*: puntero(nodo), *izq*: puntero(nodo), *der*: puntero(nodo))

#### 5.3.2. Invariante de Representación (Rehacer con nueva estructura)

- (I) Si la cola esta vacía el primer elemento es nulo.
- (II) Si no esta vacía, si su elemento izquierdo esta definido la prioridad de la raíz es mayor y la raíz es el padre del elemento.
- (III) Si no esta vacía, si su elemento derecho esta definido la prioridad de la raíz es mayor y la raíz es el padre del elemento.
- (IV) El subarbol derecho e izquierdo cumplen con el invariante.

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

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

$(e.\text{proximo} = \text{NULL}) = (e.\text{tamaño} = 0) \wedge_L$

$(e.\text{proximo} \neq \text{NULL}) \Rightarrow_L ($

$(e.\text{proximo} \rightarrow \text{izq} \neq \text{NULL} \wedge_L ((e.\text{proximo} \rightarrow \text{prior} > (e.\text{proximo} \rightarrow \text{izq}) \rightarrow \text{prior}) \wedge (e.\text{proximo} \rightarrow$

$(e.\text{proximo} \rightarrow \text{izq}) \rightarrow \text{padre})))$

$(e.\text{proximo} \rightarrow \text{der} \neq \text{NULL} \wedge_L ((e.\text{proximo} \rightarrow \text{prior} > (e.\text{proximo} \rightarrow \text{der}) \rightarrow \text{prior}) \wedge (e.\text{proximo} \rightarrow$

$(e.\text{proximo} \rightarrow \text{der}) \rightarrow \text{padre}))) \wedge$

$\text{Rep}(\text{SubArbolIzq}(e)) \wedge \text{Rep}(\text{SubArbolDer}(e)))$

$\text{SubArbolIzq} : \text{estr} \rightarrow \text{estr}$

$\text{SubArbolIzq}(c) \equiv$

$\text{estr}(c.\text{proximo} \rightarrow \text{izq}, c.\text{tamaño}-1)$

$\text{SubArbolDer} : \text{estr} \rightarrow \text{estr}$

$\text{SubArbolDer}(c) \equiv$

$\text{estr}(c.\text{proximo} \rightarrow \text{der}, c.\text{tamaño}-1)$

#### 5.3.3. Función de Abstracción

$\text{Abs} : \text{estr } e \rightarrow \text{colaMinPrior}(\alpha)$

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

$\text{Abs}(e) \equiv =_{\text{obs}} c: \text{colaMinPrior}(\alpha) \mid (\text{if vacía?}(e) \text{ then vacia else encolar}(\text{proximo}(c), \text{Abs}(\text{desencolar}(c))) \text{ fi})$

#### 5.3.4. Representación del Iterador Cola de Prioridad

ItColaMin( $\alpha$ ) se representa con *iter*

donde *iter* es *tupla*(*siguiente*: puntero(nodo), *arbol*: puntero(ColaMinPrior( $\alpha$ )))

#### 5.3.5. Invariante de Representación

- (I) Tiene que valer el rep de la cola.
- (II) El siguiente es invalido o esta en la cola



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

$\text{Rep}(it) \equiv \text{true} \iff$   
 $\text{Rep}(*(\text{it.col}) \wedge_L$   
 $(\text{it.siguiente} = \text{NULL} \vee_L (\text{Esta?}(\text{it.siguiente}, *(\text{it.col})))$

$\text{Esta?} : \text{puntero}(\text{Nodo}) \times \text{estr} \rightarrow \text{bool}$

$\text{Esta?}(p, c) \equiv$   
 $p = c.\text{proximo} \vee_L (\text{Esta?}(p, \text{SubArbolIzq}(c))) \vee_L (\text{Esta?}(p, \text{SubArbolDer}(c)))$

### 5.3.6. Función de Abstracción

$\text{Abs} : \text{estr } it \rightarrow \text{itColaMin}$

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

$\text{Abs}(it) \equiv =_{\text{obs}} u : \text{itColaMin} \mid (\text{Siguietes}(u) = \text{deColaASecu}(\text{it.col}))$

▷ La funcion deColaASecu convierte todos los elementos de la cola en una secu( $\alpha$ )

## 5.4. Algoritmos

### 5.4.1. Algoritmos del Modulo

---

---

**iVacía()**  $\rightarrow res : \text{estr}$

1:  $res \leftarrow \langle \text{NULL}, 0 \rangle$

▷  $\Theta(1)$

Complejidad:  $\Theta(1)$

---



---

---

**iVacía?(in c: estr)**  $\rightarrow res : \text{Bool}$

1:  $res \leftarrow (c.\text{proximo} = \text{NULL})$

▷  $\Theta(1)$

Complejidad:  $\Theta(1)$

---



---

---

**iPróximo(in c: estr)**  $\rightarrow res : \alpha$

1:  $res \leftarrow \text{CrearIt}(c).\text{Siguiete} \rightarrow \text{elem}$

▷  $\Theta(1)$

Complejidad:  $\Theta(1)$

---



---

---

**iUltimoNodo(in/out c: estr)**  $\rightarrow res : \text{puntero}(\text{Nodo})$

1:  $A : \text{arreglo}_{\text{dimensionable}} \text{denat}$

▷  $\Theta(1)$

2:  $A \leftarrow i\text{DecimalABinario}(c.\text{tamao})$  ▷ Convertir un decimal a binario tiene complejidad logaritmica del largo de número  $\Theta(\log(|c|))$

3:  $\text{puntero}(\text{Nodo})n \leftarrow c.\text{proximo}$

▷  $\Theta(1)$

4: **for**  $i \leftarrow 1$  **to**  $\text{tam}(A) - 1$  **do** ▷ Se empieza desde el segundo elemento porque ya está posicionado en el primer elemento que por precondition no es nulo  $\Theta(|A|) = \Theta(\log(|c|))$

5:   **if**  $A[i] = 0$  **then**

6:      $n \leftarrow (n \rightarrow \text{izq})$

▷  $\Theta(1)$

7:   **else**

8:      $n \leftarrow (n \rightarrow \text{der})$

▷  $\Theta(1)$

9:   **end if**

10: **end for**

11:  $res \leftarrow n$

▷  $\Theta(1)$

Complejidad:  $\Theta(\log(|c|))$

---

**iDesencolar**(in/out  $c$ : **estr**)

```

1:  $\text{puntero}(\text{Nodo})n \leftarrow i\text{UltimoNodo}(c)$   $\triangleright \Theta(\log(|c|))$ 
2:  $i\text{SwapCola}(c.\text{proximo}, n)$   $\triangleright \Theta(1)$ 
3: if  $c.\text{tamano} > 1$  then
4:   if  $c.\text{tamanomod}2 = 0$  then
5:      $n \leftarrow \text{padre} \leftarrow \text{izq} \leftarrow \text{NULL}$   $\triangleright \Theta(1)$ 
6:   else
7:      $n \leftarrow \text{padre} \leftarrow \text{der} \leftarrow \text{NULL}$   $\triangleright \Theta(1)$ 
8:   end if
9:    $c.\text{tamao} \leftarrow c.\text{tamao} - 1$   $\triangleright \Theta(1)$ 
10:   $i\text{Bajar}(c, n)$   $\triangleright \Theta(\log(|c|))$ 
11: else
12:   $c.\text{proximo} \leftarrow \text{NULL}$   $\triangleright \Theta(1)$ 
13:   $c.\text{tamao} \leftarrow 0$   $\triangleright \Theta(1)$ 
14: end if

```

Complejidad:  $\Theta(\log(|c|))$

**iPadreNuevoNodo**(in/out  $c$ : **estr**)  $\rightarrow res$  :  $\text{puntero}(\text{Nodo})$ 

```

1:  $A : \text{arreglo}_{\text{dimensionable}} \text{denat}$   $\triangleright \Theta(1)$ 
2:  $A \leftarrow i\text{DecimalABinario}(c.\text{tamao} + 1)$   $\triangleright$  Convertir un decimal a binario tiene complejidad logaritmica del largo de número  $\Theta(\log(|c|))$ 
3:  $\text{puntero}(\text{Nodo})n \leftarrow c.\text{proximo}$   $\triangleright \Theta(1)$ 
4: for  $i \leftarrow 1$  to  $\text{tam}(A) - 2$  do  $\triangleright$  Se empieza desde el segundo elemento porque ya está posicionado en el primer elemento que por precondition no es nulo  $\Theta(|A|) = \Theta(\log(|c|))$ 
5:   if  $A[i] = 0$  then
6:      $n \leftarrow (n \rightarrow \text{izq})$   $\triangleright \Theta(1)$ 
7:   else
8:      $n \leftarrow (n \rightarrow \text{der})$   $\triangleright \Theta(1)$ 
9:   end if
10: end for
11:  $res \leftarrow n$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(\log(|c|))$

**iEncolar**(in/out  $c$ : **estr**, in  $\text{prioridad}$ : **nat**, in  $a$ :  $\alpha$ )  $\rightarrow res$  :  $iter$ 

```

1:  $\text{puntero}(\text{Nodo}) : \text{padre} \leftarrow i\text{PadreNuevoNodo}(c)$   $\triangleright \Theta(\log(|c|))$ 
2:  $\text{Nodo} : \text{nuevo} \leftarrow \langle \text{prioridad}, a, \text{padre}, \text{NULL}, \text{NULL} \rangle$   $\triangleright \Theta(1)$ 
3: if  $\text{padre} \rightarrow \text{izq} = \text{NULL}$  then  $\triangleright \Theta(1)$ 
4:    $\text{padre} \rightarrow \text{izq} = \&\text{nuevo}$   $\triangleright \Theta(1)$ 
5: else
6:    $\text{padre} \rightarrow \text{der} = \&\text{nuevo}$   $\triangleright \Theta(1)$ 
7: end if
8:  $c.\text{tamao} \leftarrow c.\text{tamao} + 1$   $\triangleright \Theta(1)$ 
9:  $i\text{Subir}(c, \text{nuevo})$   $\triangleright \Theta(\log(|c|))$ 
10:  $res \leftarrow \text{CrearItEncolado}(\text{nuevo}, c)$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(\log(|c|)) + \text{copy}(\alpha)$

---

**iBajar**(in/out  $c$ : *estr*, in  $n$ : puntero(Nodo))

```

1:  $bool : swap \leftarrow true$   $\triangleright \Theta(1)$ 
2:  $bool : esHoja \leftarrow (n \rightarrow izq = NULL \wedge n \rightarrow der = NULL)$   $\triangleright \Theta(1)$ 
3: while  $\neg esHoja \wedge swap$  do  $\triangleright$  Lo máximo que se puede llegar a avanzar es la altura del árbol  $\Theta(\log(|c|))$ 
4:    $swap \leftarrow false$   $\triangleright \Theta(1)$ 
5:   if  $n \rightarrow der = NULL$  then  $\triangleright \Theta(1)$ 
6:     if  $n \rightarrow izq \rightarrow prior < n \rightarrow prior$  then  $\triangleright \Theta(1)$ 
7:        $iSwapCola(n \rightarrow izq, n)$   $\triangleright \Theta(1)$ 
8:        $swap \leftarrow true$   $\triangleright \Theta(1)$ 
9:     end if
10:  else
11:    if  $n \rightarrow izq = NULL$  then  $\triangleright \Theta(1)$ 
12:      if  $n \rightarrow der \rightarrow prior < n \rightarrow prior$  then  $\triangleright \Theta(1)$ 
13:         $iSwapCola(n \rightarrow izq, n)$   $\triangleright \Theta(1)$ 
14:         $swap \leftarrow true$   $\triangleright \Theta(1)$ 
15:      end if
16:    else
17:      if  $n \rightarrow der \rightarrow prior < n \rightarrow izq \rightarrow prior$  then  $\triangleright \Theta(1)$ 
18:        if  $n \rightarrow der \rightarrow prior < n \rightarrow prior$  then  $\triangleright \Theta(1)$ 
19:           $iSwapCola(n \rightarrow izq, n)$   $\triangleright \Theta(1)$ 
20:           $swap \leftarrow true$   $\triangleright \Theta(1)$ 
21:        else
22:          if  $n \rightarrow izq \rightarrow prior < n \rightarrow prior$  then  $\triangleright \Theta(1)$ 
23:             $iSwapCola(n \rightarrow izq, n)$   $\triangleright \Theta(1)$ 
24:             $swap \leftarrow true$   $\triangleright \Theta(1)$ 
25:          end if
26:        end if
27:      else
28:        if  $n \rightarrow izq \rightarrow prior < n \rightarrow prior$  then  $\triangleright \Theta(1)$ 
29:           $iSwapCola(n \rightarrow izq, n)$   $\triangleright \Theta(1)$ 
30:           $swap \leftarrow true$   $\triangleright \Theta(1)$ 
31:        else
32:          if  $n \rightarrow der \rightarrow prior < n \rightarrow prior$  then  $\triangleright \Theta(1)$ 
33:             $iSwapCola(n \rightarrow izq, n)$   $\triangleright \Theta(1)$ 
34:             $swap \leftarrow true$   $\triangleright \Theta(1)$ 
35:          end if
36:        end if
37:      end if
38:    end if
39:  end if
40:   $esHoja \leftarrow (n \rightarrow izq = NULL \wedge n \rightarrow der = NULL)$   $\triangleright \Theta(1)$ 
41: end while

```

Complejidad:  $\Theta(\log(|c|))$ 


---

---



---

**iSubir**(in/out  $c$ : **estr**, in  $n$ : puntero(Nodo))

```

1:  $bool : swap \leftarrow true$   $\triangleright \Theta(1)$ 
2:  $bool : esRaiz \leftarrow (n \rightarrow padre = NULL)$   $\triangleright \Theta(1)$ 
3: while  $\neq esRaiz \wedge swap$  do  $\triangleright$  Lo máximo que se puede llegar a avanzar es la altura del árbol  $\Theta(\log(|c|))$ 
4:    $swap \rightarrow false$   $\triangleright \Theta(1)$ 
5:   if  $n \rightarrow prior < n \rightarrow padre \rightarrow prior$  then  $\triangleright \Theta(1)$ 
6:      $swapCola(n \rightarrow padre, n)$   $\triangleright \Theta(1)$ 
7:      $swap \rightarrow true$   $\triangleright \Theta(1)$ 
8:   end if
9:    $esRaiz \leftarrow (n \rightarrow padre = NULL)$   $\triangleright \Theta(1)$ 
10: end while

```

Complejidad:  $\Theta(\log(|c|))$

---



---



---

**iSwapCola**(in/out  $p$ : puntero(Nodo), in/out  $q$ : puntero(Nodo))

```

1:  $Nodo : aux \leftarrow (*p)$   $\triangleright \Theta(Copy(\alpha))$ 
2:  $p \rightarrow padre \leftarrow q \rightarrow padre$   $\triangleright \Theta(1)$ 
3:  $p \rightarrow izq \leftarrow q \rightarrow izq$   $\triangleright \Theta(1)$ 
4:  $p \rightarrow der \leftarrow q \rightarrow der$   $\triangleright \Theta(1)$ 
5:  $q \rightarrow padre \leftarrow aux.padre$   $\triangleright \Theta(1)$ 
6:  $q \rightarrow izq \leftarrow aux.izq$   $\triangleright \Theta(1)$ 
7:  $q \rightarrow der \leftarrow aux.der$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(Copy(\alpha))$

---



---



---

**iDecimalABinario**(in  $d$ : nat)  $\rightarrow res$ : arreglo(nat)

```

1:  $lista(\alpha) : temp \leftarrow Vacia()$   $\triangleright \Theta(1)$ 
2: while  $d > 1$  do  $\triangleright$  Lo máximo que se puede llegar a iterar es  $\log(d)$   $\Theta(\log(d))$ 
3:    $AgregarAdelante(temp, dmod2)$   $\triangleright \Theta(1)$ 
4:    $d \leftarrow d/2$   $\triangleright$  División entera  $\Theta(1)$ 
5: end while
6:  $AgregarAdelante(temp, d)$   $\triangleright \Theta(1)$ 
7:  $it \leftarrow CrearIt(temp)$   $\triangleright \Theta(1)$ 
8:  $nati \leftarrow 0$   $\triangleright \Theta(1)$ 
9:  $arreglo(nat)bin \leftarrow CrearArreglo(Longitud(temp))$   $\triangleright \Theta(1)$ 
10: while  $HaySiguiente(it)$  do  $\triangleright$  El largo de la lista es  $\log(d)$   $\Theta(|temp|) = \Theta(\log(d))$ 
11:    $bin[i] \leftarrow Siguiente(it)$   $\triangleright \Theta(1)$ 
12:    $i \leftarrow i + 1$   $\triangleright \Theta(1)$ 
13: end while
14:  $res \leftarrow bin$   $\triangleright \Theta(1)$ 

```

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

---

#### 5.4.2. Algoritmos del Iterador

---



---

**iCrearIt**(in  $c$ : **estr**)  $\rightarrow res$ : iter

```

1:  $res \leftarrow \langle c.proximo, c \rangle$   $\triangleright \Theta(1)$ 

```

Complejidad:  $\Theta(1)$

---

---

**iCrearItEncolado**(in  $c: \text{estr}$ , in  $nodo: \text{puntero}(\text{Nodo})$ )  $\rightarrow res: \text{iter}$   $\triangleright$  Esta es una operación privada, dado a que no hay un avanzar se crea un iterador con un elemento que pertenezca a la cola

**Pre**  $\equiv nodo \neq \text{NULL} \wedge_L nodo =_{obs} (\exists n \in [0..c.tamaño-1])$

1:  $res \leftarrow \langle n, c \rangle$   $\triangleright \Theta(1)$

Complejidad:  $\Theta(1)$

**Post**  $\equiv res =_{obs} (\exists i \in [0..c.tamaño-1])$

---



---

**iHaySiguiete?**(in  $it: \text{iter}$ )  $\rightarrow res: \text{bool}$

1:  $res \leftarrow it.siguiete \neq \text{NULL}$   $\triangleright \Theta(1)$

Complejidad:  $\Theta(1)$

---



---

**iSiguiete**(in  $it: \text{iter}$ )  $\rightarrow res: \alpha$

1:  $res \leftarrow (it.siguiete \rightarrow elem)$   $\triangleright \Theta(1)$

Complejidad:  $\Theta(1)$

---



---

**iEliminarSiguiete**(in/out  $it: \text{iter}$ )

1: **if**  $arbol.tamaño > 1$  **then**  $\triangleright \Theta(\log(|arbol|))$

2:  $\text{puntero}(\text{Nodo})ult \leftarrow iUltimoNodo(arbol)$   $\triangleright \Theta(1)$

3:  $iSwapCola(ult, it.siguiete)$   $\triangleright \Theta(1)$

4: **if**  $arbol.tamaño \bmod 2 = 0$  **then**

5:  $it.siguiete \rightarrow padre \rightarrow izq \leftarrow \text{NULL}$   $\triangleright \Theta(1)$

6: **else**  $\triangleright \Theta(1)$

7:  $it.siguiete \rightarrow padre \rightarrow der \leftarrow \text{NULL}$   $\triangleright \Theta(1)$

8: **end if**

9:  $arbol.tamao \leftarrow arbol.tamao - 1$   $\triangleright \Theta(1)$

10:  $iBajar(arbol, ult)$   $\triangleright \Theta(\log(|arbol|))$

11: **else**

12:  $it.siguiete \leftarrow \text{NULL}$   $\triangleright \Theta(1)$

13:  $arbol.tamao \leftarrow 0$   $\triangleright \Theta(1)$

14: **end if**

15:  $(it.siguiete \rightarrow padre) \leftarrow \text{NULL}$   $\triangleright \Theta(1)$

Complejidad:  $\Theta(1)$

---

## 6. Módulo Diccionario String( $\alpha$ )

Se representa mediante un árbol n-ario con invariante de trie. Las claves son strings y permite acceder a un significado en un tiempo en el peor caso igual a la longitud de la palabra (string) más larga y definir un significado en el mismo tiempo más el tiempo de copy(s) ya que los significados se almacenan por copia.

### 6.1. Interfaz

**parametros formales**

**géneros:**  $\alpha$ .

**funcion:** COPIAR(in  $s : \alpha$ )  $\rightarrow res : \alpha$

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

**Post**  $\equiv \{res =_{obs} s\}$

**Complejidad:**  $O(\text{copy}(s))$

**Descripción:** funcion de copia de  $\alpha$ .

**se explica con:** DICCIONARIO(STRING, $\alpha$ ).

**géneros:** diccString( $\alpha$ ), itDiccString( $\alpha$ ).

#### 6.1.1. Operaciones básicas de Diccionario String( $\alpha$ )

CREARDICCIONARIO()

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

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

**Complejidad:**  $O(1)$  Justificación: Sólo crea un arreglo de 256 posiciones inicializadas con null y una lista vacía

**Descripción:** Crea un diccionario vacío.

DEFINIDO?(in  $d : \text{diccString}(\alpha)$ , in  $c : \text{string}$ )  $\rightarrow res : \text{bool}$

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

**Post**  $\equiv \{res =_{obs} \text{def?}(d, c)\}$

**Complejidad:**  $O(|c|)$  Justificación: Debe acceder a la clave  $c$ , recorriendo una por una las partes de la clave (caracteres)

**Descripción:** Devuelve true si la clave está definida en el diccionario y false en caso contrario.

DEFINIR(in/out  $d : \text{diccString}(\alpha)$ , in  $c : \text{string}$ , in  $s : \alpha$ )

**Pre**  $\equiv \{d =_{obs} d_0\}$

**Post**  $\equiv \{d =_{obs} \text{definir}(c, s, d_0)\}$

**Complejidad:**  $O(|c| + \text{copy}(s))$  Justificación: Debe definir la clave  $c$ , recorriendo una por una las partes de la clave y después copiar el contenido del significado.

**Descripción:** Define la clave  $c$  con el significado  $s$

**Aliasing:** Almacena una copia de  $s$ .

OBTENER(in  $d : \text{diccString}(\alpha)$ , in  $c : \text{string}$ )  $\rightarrow res : \alpha$

**Pre**  $\equiv \{\text{def?}(c, d)\}$

**Post**  $\equiv \{\text{alias}(res =_{obs} \text{obtener}(c, d))\}$

**Complejidad:**  $O(|c|)$  Justificación: Debe acceder a la clave  $c$ , recorriendo una por una las partes de la clave (caracteres)

**Descripción:** Devuelve el significado correspondiente a la clave  $c$ .

**Aliasing:** Devuelve el significado almacenado en el diccionario, por lo que  $res$  es modificable si y sólo si  $d$  lo es.

ELIMINAR(in/out  $d : \text{diccString}(\alpha)$ , in  $c : \text{string}$ )

**Pre**  $\equiv \{d =_{\text{obs}} d_0 \wedge \text{def?}(d, c)\}$

**Post**  $\equiv \{d =_{\text{obs}} \text{borrar}(d_0, c)\}$

**Complejidad:**  $O(|c|)$  Justificación: Debe acceder a la clave  $c$ , recorriendo una por una las partes de la clave (caracteres) e invalidar su significado

**Descripción:** Borra la clave  $c$  del diccionario y su significado.

**CREARITCLAVES**(in  $d: \text{diccString}(\alpha)$ )  $\rightarrow res: \text{itConj}(\text{String})$

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

**Post**  $\equiv \{\text{alias}(\text{esPermutacion?}(\text{SecuSuby}(res), c)) \wedge \text{vacía?}(\text{Anteriores}(res))\}$

**Complejidad:**  $O(1)$

**Descripción:** Crea un Iterador de Conjunto en base a la interfaz del iterador de Conjunto Lineal

### 6.1.2. Operaciones Básicas Del Iterador

Este iterador permite recorrer el trie sobre el que está implementado el diccionario para obtener de cada clave los significados. Las claves de los elementos iterados no pueden modificarse nunca por cuestiones de implementación. El iterador es un iterador de lista, que recorre listaIterable por lo que sus operaciones son idénticas a ella.

**CREARIT**(in  $d: \text{diccString}(\alpha)$ )  $\rightarrow res: \text{itDiccString}(\alpha)$

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

**Post**  $\equiv \{\text{alias}(\text{esPermutación}(\text{SecuSuby}(res), d)) \wedge \text{vacía?}(\text{Anteriores}(res))\}$

**Complejidad:**  $O(1)$

**Descripción:** crea un iterador bidireccional del diccionario, de forma tal que HayAnterior evalúe a false (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).

**Aliasing:** El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique  $d$  sin utilizar las funciones del iterador.

**HAYSIGUIENTE**(in  $it: \text{itDiccString}(\alpha)$ )  $\rightarrow res: \text{bool}$

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

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

**Complejidad:**  $O(1)$

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

**HAYANTERIOR**(in  $it: \text{itDiccString}(\alpha)$ )  $\rightarrow res: \text{bool}$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{hayAnterior?}(it)\}$

**Complejidad:**  $O(1)$

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

**SIGUIENTESIGNIFICADO**(in  $it: \text{itDiccString}(\alpha)$ )  $\rightarrow res: \alpha$

**Pre**  $\equiv \{\text{haySiguiente?}(it)\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{haySiguiente?}(it).\text{significado})\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve el significado del elemento siguiente del iterador

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

**ANTERIORESIGNIFICADO**(in  $it: \text{itDiccString}(\alpha)$ )  $\rightarrow res: \alpha$

**Pre**  $\equiv \{\text{hayAnterior?}(it)\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{hayAnterior?}(it).\text{significado})\}$

**Complejidad:**  $O(1)$

**Descripción:** devuelve el significado del elemento anterior del iterador

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

AVANZAR(**in/out**  $it$ : itDiccString( $\alpha$ ))

**Pre**  $\equiv \{it = it_0 \wedge \text{haySiguiente?}(it)\}$

**Post**  $\equiv \{it =_{obs} \text{avanzar}(it_0)\}$

**Complejidad:**  $O(1)$

**Descripción:** avanza a la posición siguiente del iterador.

RETROCEDER(**in/out**  $it$ : itDiccString( $\alpha$ ))

**Pre**  $\equiv \{it = it_0 \wedge \text{hayAnterior?}(it)\}$

**Post**  $\equiv \{it =_{obs} \text{hayAnterior?}(it_0)\}$

**Complejidad:**  $O(1)$

**Descripción:** retrocede a la posición anterior del iterador.



### 6.1.3. Representación de Diccionario String( $\alpha$ )

Diccionario String( $\alpha$ ) se representa con estr

donde estr es  $\text{tupla}(\text{raiz: arreglo}(\text{puntero}(\text{Nodo})), \text{listaIterable: lista}(\text{puntero}(\text{Nodo})))$

donde  $\text{Nodo}$  es  $\text{tupla}(\text{arbolTrie: arreglo}(\text{puntero}(\text{Nodo})),$   
 $\text{info: } \alpha,$   
 $\text{info Valida: bool},$   
 $\text{infoEnLista: iterador}(\text{listaIterable})$  )

### 6.1.4. Invariante de Representación

- (I) Raiz es la raiz del arbol con invariante de trie y es un arreglo de 256 posiciones.
- (II) Cada uno de los elementos de la lista tiene que ser un puntero a un Nodo del trie.
- (III)  $\text{Nodo}$  es una tupla que contiene un arreglo de 256 posiciones con un puntero a otro  $\text{Nodo}$  en cada posicion ,un elemento  $\text{info}$  que es el  $\alpha$  que contiene esa clave del arbol, un elemento  $\text{info Valida}$  y un elemento  $\text{iterador}$  que es un puntero a un nodo de la lista enlazada.
- (IV) El  $\text{iterador}$  a la lista enlazada de cada nodo tiene que apuntar al elemento de la lista que apunta al mismo  $\text{Nodo}$ .
- (V) Cada uno de los nodos de la lista apunta a un nodo del arbol cuyo  $\text{infoEnLista}$  apunta al mismo nodo de la lista.

$(\forall c: \text{diccString}((\alpha)))()$

$\text{Rep} : \text{estr} \longrightarrow \text{bool}$

$\text{Rep}(e) \equiv \text{true} \iff$   
 $\text{longitud}(e.\text{raiz}) == 256 \wedge_L$   
 $(\forall i \in [0..\text{longitud}(e.\text{raiz})])$   
 $((\neg e.\text{raiz}[i] == \text{NULL}) \Rightarrow_L \text{nodoValido}(\text{raiz}[i])) \wedge (*e.\text{raiz}[i].\text{info Valida} == \text{true} \Rightarrow_L$   
 $\text{iteradorValido}(\text{raiz}[i])) \wedge$   
 $\text{listaValida}(e.\text{listaIterable})$

$\text{nodoValido} : \text{puntero}(\text{Nodo}) \text{ nodo} \longrightarrow \text{bool}$

$\text{iteradorValido} : \text{puntero}(\text{Nodo}) \text{ nodo} \longrightarrow \text{bool}$

$\text{nodoValido}(\text{nodo}) \equiv$   
 $\text{longitud}(*\text{nodo.arbolTrie}) == 256 \wedge_L$   
 $(\forall i \in [0..\text{longitud}(*\text{nodo.arbolTrie})])$   
 $((\neg *\text{nodo.arbolTrie}[i] == \text{NULL}) \Rightarrow_L \text{nodoValido}(*\text{nodo.arbolTrie}[i]))$   
 $\text{iteradorValido}(\text{nodo}) \equiv$   
 $\text{PunteroValido}(\text{nodo}) \wedge_L$   
 $(\forall i \in [0..\text{longitud}(*\text{nodo.arbolTrie})])$   
 $((*\text{nodo.arbolTrie}[i].\text{info Valida} == \text{true}) \Rightarrow_L \text{iteradorValido}(*\text{nodo.arbolTrie}[i]))$

$\text{PunteroValido}(\text{nodo}) \equiv$

El  $\text{iterador}$  perteneciente al nodo ( $\text{infoEnLista}$ ) apunta a un nodo de  $\text{listaIterable}$  ( $\text{lista}(\text{puntero}(\text{Nodo}))$ ) cuyo puntero apunta al mismo nodo pasado por parámetro. Es decir se trata de una referencia circular.

$\text{listaValida}(\text{lista}) \equiv$

Cada nodo de la lista tiene un puntero a un nodo de la estructura cuyo  $\text{infoEnLista}$  ( $\text{iterador}$ ) apunta al mismo nodo. Es decir se trata de una referencia circular.

### 6.1.5. Función de Abstracción

$Abs : estr\ e \longrightarrow diccString(\alpha) \quad \{Rep(e)\}$   
 $Abs(e) \equiv =_{obs} d: diccString(\alpha) \mid (\forall s: string)(def?(d, s) =_{obs}$   
 $\quad Definido?(d, s) \wedge$   
 $\quad def?(d, s) \Rightarrow_L obtener(s, d) =_{obs}$   
 $\quad Obtener(d, s)$   
 $\quad )$

## 6.2. Algoritmos

---

**iCrearDiccionario()**  $\rightarrow res : estr$

$arreglo(puntero(Nodo)) : res.raiz \leftarrow CrearArreglo(256) \quad \triangleright O(256)$   
 $nat : i \leftarrow 0 \quad \triangleright O(1)$   
**while**  $i < long(res.raiz)$  **do**  $\triangleright O(1)$   
 $\quad res.raiz[i] \leftarrow NULL \quad \triangleright O(1)$   
**end while**  
 $res.listaIterable \leftarrow Vacía() \quad \triangleright O(1)$

Complejidad:  $O(1)$

Justificación: Crea un arreglo de 256 posiciones y lo recorre inicializándolo en NULL. Luego crea una lista vacía.

---



---

**iDefinido?(in d: estr, in c: string)**  $\rightarrow res : bool$

$nat : i \leftarrow 0 \quad \triangleright O(1)$   
 $nat : letra \leftarrow ord(c[0]) \quad \triangleright O(1)$   
 $puntero(Nodo) : arr \leftarrow d.raiz[letra] \quad \triangleright O(1)$   
**while**  $i < longitud(c) \wedge \neg arr = NULL$  **do**  $\triangleright O(|c|)$   
 $\quad i \leftarrow i + 1 \quad \triangleright O(1)$   
 $\quad letra \leftarrow ord(c[i]) \quad \triangleright O(1)$   
 $\quad arr \leftarrow (*arr).arbolTrie[letra] \quad \triangleright O(1)$   
**end while**  
**if**  $i = longitud(c)$  **then**  $\triangleright O(1)$   
 $\quad res \leftarrow (*arr).infoValida \quad \triangleright O(1)$   
**else**  
 $\quad res \leftarrow false \quad \triangleright O(1)$   
**end if**

Complejidad:  $O(|c|)$

Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego itera sobre los caracteres restantes hasta el final del String c, por lo que hace  $|c|$  operaciones. Finalmente pregunta si el significado encontrado es válido o no.

---

---

**iDefinir**(in/out  $d$ : estr, in  $c$ : string, in  $s$ :  $\alpha$ )

```

  nat : i ← 0                                     ▷ O(1)
  nat : letra ← ord(c[0])                         ▷ O(1)
  if d.raiz[letra] = NULL then                    ▷ O(1)
    Nodo : nuevo                                  ▷ O(1)
    arreglo(puntero(Nodo)) : nuevo.arbolTrie ← CrearArreglo(256) ▷ O(1)
    nuevo.infoValida ← false                      ▷ O(1)
    d.raiz[letra] ← puntero(nuevo)                ▷ O(1)
  end if
  puntero(Nodo) : arr ← d.raiz[letra]             ▷ O(1)
  while i < longitud(c) do                         ▷ O(|c|)
    i ← i + 1                                     ▷ O(1)
    letra ← ord(c[i])                             ▷ O(1)
    if arr.arbolTrie[letra] = NULL then            ▷ O(1)
      Nodo : nuevoHijo                           ▷ O(1)
      arreglo(puntero(Nodo)) : nuevoHijo.arbolTrie ← CrearArreglo(256) ▷ O(1)
      nuevoHijo.infoValida ← false                ▷ O(1)
      arr.arbolTrie[letra] ← puntero(nuevoHijo)   ▷ O(1)
    end if
    arr ← (*arr).arbolTrie[letra]                 ▷ O(1)
  end while
  (*arr).info ← s                                 ▷ O(copy(s))
  if ¬(*arr).infoValida then                      ▷ O(1)
    itLista(puntero(Nodo))it ← AgregarAdelante(d.listaIterable, NULL) ▷ O(1)
    (*arr).infoValida ← true                      ▷ O(1)
    (*arr).infoEnLista ← it                      ▷ O(1)
    siguiente(it) ← puntero(*arr)                ▷ O(1)
  end if

```

Complejidad:  $O(|c| + \text{copy}(s))$

Justificación: Itera sobre la cantidad de caracteres del String  $c$  y en caso de que algún caracter no esté definido crea un arreglo de 256 posiciones, por lo que realiza  $|c|$  operaciones. Luego copia el significado pasado por parámetro en  $O(\text{copy}(s))$  y finalmente agrega en la lista un puntero al nodo creado.

---



---

**iObtener**(in  $d$ : estr, in  $c$ : string)  $\rightarrow res$ :  $\alpha$ 

```

  nat : i ← 0                                     ▷ O(1)
  nat : letra ← ord(c[0])                         ▷ O(1)
  puntero(Nodo) : arr ← d.raiz[letra]             ▷ O(1)
  while i < longitud(c) do                         ▷ O(|c|)
    i ← i + 1                                     ▷ O(1)
    letra ← ord(c[i])                             ▷ O(1)
    arr ← (*arr).arbolTrie[letra]                 ▷ O(1)
  end while
  res ← (*arr).info                               ▷ O(1)

```

Complejidad:  $O(|c|)$

Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego itera sobre los caracteres restantes hasta el final del String  $c$ , por lo que hace  $|c|$  operaciones. Finalmente retorna el significado almacenado. Todas las demás operaciones se realizan en  $O(1)$  porque son comparaciones o asignaciones de valores enteros o de punteros.

---

---

**iEliminar**(in/out  $d$ : **estr**, in  $c$ : **string**)

```

  nat :  $i \leftarrow 0$  ▷  $O(1)$ 
  nat :  $letra \leftarrow ord(c[0])$  ▷  $O(1)$ 
  puntero(Nodo) :  $arr \leftarrow d.raiz[letra]$  ▷  $O(1)$ 
  pila(puntero(Nodo)) :  $pil \leftarrow Vacia()$  ▷  $O(1)$ 
  while  $i < longitud(c)$  do ▷  $O(|c|)$ 
     $i \leftarrow i + 1$  ▷  $O(1)$ 
     $letra \leftarrow ord(c[i])$  ▷  $O(1)$ 
     $arr \leftarrow (*arr).arbolTrie[letra]$  ▷  $O(1)$ 
    Apilar( $pil, arr$ ) ▷  $O(1)$ 
  end while
  if tieneHermanos( $arr$ ) then ▷  $O(1)$ 
    ( $*arr$ ).infoValida  $\leftarrow false$  ▷  $O(1)$ 
  else
     $i \leftarrow i + 1$  ▷  $O(1)$ 
    puntero(Nodo) :  $del \leftarrow tope(pil)$  ▷  $O(1)$ 
     $del \leftarrow NULL$  ▷  $O(1)$ 
    Desapilar( $pil$ ) ▷  $O(1)$ 
    while  $i < longitud(c) \wedge \neg tieneHermanosEInfo(*tope(pil))$  do ▷  $O(|c|)$ 
       $del \leftarrow tope(pil)$  ▷  $O(1)$ 
       $del \leftarrow NULL$  ▷  $O(1)$ 
      Desapilar( $pil$ ) ▷  $O(1)$ 
       $i \leftarrow i + 1$  ▷  $O(1)$ 
    end while
    if  $i = longitud(c)$  then ▷  $O(1)$ 
       $d.raiz[ord(c[0])] \leftarrow NULL$  ▷  $O(1)$ 
    end if
  end if

```

Complejidad:  $O(|c|)$ 

Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego crea una pila en  $O(1)$ . Recorre el resto de los caracteres del String  $c$  y apila cada uno de los Nodos encontrado en la pila ( $O(1)$ ) por lo que en total realiza  $|c|$  operaciones. Llama a la función `tieneHermanos` y le pasa por parámetro el nodo encontrado  $O(1)$  (ver Algoritmo "tieneHermanos"). Luego recorre todos los elementos apilados preguntando si hay alguno que no tiene hermanos para en cuyo caso eliminarlo, realizando en el peor caso  $|c|$  operaciones porque puede ser que sea necesario eliminar todo hasta la raíz.

---



---

**tieneHermanos**(in  $nodo$ : puntero(Nodo))  $\rightarrow res$ : **bool**

```

  nat :  $i \leftarrow 0$  ▷  $O(1)$ 
  nat :  $l \leftarrow longitud((*nodo).arbolTrie)$  ▷  $O(1)$ 
  while  $i < l \wedge \neg ((*nodo).arbolTrie[i] = NULL)$  do ▷  $O(1)$ 
     $i \leftarrow i + 1$  ▷  $O(1)$ 
  end while
   $res \leftarrow i < l$  ▷  $O(1)$ 

```

Complejidad:  $O(1)$ 

Justificación: Recorre el arreglo de 256 posiciones en caso de que todas las posiciones del mismo tengan NULL. Como es una constante ya que en el peor caso siempre recorre a lo sumo 256 posiciones entonces es  $O(1)$ .

---



---

**tieneHermanosEInfo**(in  $nodo$ : puntero(Nodo))  $\rightarrow res$ : **bool**

```

   $res \leftarrow tieneHermanos(nodo) \wedge (*nodo).infoValida = true$  ▷  $O(1)$ 

```

Complejidad:  $O(1)$ 

Justificación: Llama a la función `tieneHermanos` que es  $O(1)$  y verifica además que el nodo contenga información válida.

---