# Algoritmos y Estructuras de Datos II

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

## Trabajo Práctico 2 - Diseño

PokemonGOArgentina

**Grupo SEG1V1ENTATION FAULT**



| Integrante | LU | Correo electrónico |
|---|---|---|
| Vigali, Leandro Ezequiel | 951/12 | leandrovigali@yahoo.com.ar |
| Gayol, Patricio Nahuel | 805/13 | patriciogayol@hotmail.com |
| Mosqueira Caballero, Edgardo Ramon | 808/13 | edgarcab666@hotmail.com |
| Romero, Mariano Oscar | 661/09 | marianoromero.11@gmail.com |



**Reservado para la cátedra**

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

# Índice

# 1. Módulo Coordenada

## 1.1. Interfaz de Coordenada

# Interfaz

**usa**: NAT, BOOL.

**se explica con**: COORDENADA.

**géneros**: coord.

### Operaciones básicas de Coordenada

CREARCOORD(**in** *lat* : `nat`, **in** *long* : `nat`) → *res* : `coord`
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}} crearCoor(lat, long)$}
**Complejidad:** $O(1)$
**Descripción:** crear una coordenada.

LATITUD(**in** *coordenada* : `coord`) → *res* : `nat`
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}} latitud(c)$}
**Complejidad:** $O(1)$
**Descripción:** devuelve la primera componente de la coordenada.

LONGITUD(**in** *coordenada* : `coord`) → *res* : `nat`
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}} longitud(c)$}
**Complejidad:** $O(1)$
**Descripción:** devuelve la segunda componente de la coordenada.

DISTANCIA(**in** $c_1$ : `coord`, **in** $c_2$ : `coord`) → *res* : `nat`
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}} distEuclidea(c_1, c_2)$}
**Complejidad:** $O(1)$
**Descripción:** dado dos coordenadas, devuelve la distancia euclidiana entre estas.

## 1.2. Representación de Coordenada

# Representación

`coord` **se representa con** `tupla`(*latitud* : `nat`, *longitud* : `nat`)

Rep : tupla(*latitud* : *nat* × *longitud* : *nat*)*c* ⟶ bool
Rep(*c*) ≡ true ⟺ true

Abs : tupla(*latitud* :nat × *longitud* :nat) *c* ⟶ coor {Rep(*c*)}
Abs(*c*) ≡ *e* : coor | c.latitud =$_{\text{obs}}$ latitud(e) ∧ c.longitud =$_{\text{obs}}$ longitud(e)

### 1.3. Algoritmos de Coordenada

## Algoritmos

---

**icrearCoord**(**in** *lat* : `nat`, **in** *long* : `nat`) → *res* : tupla(*latitud* : nat, *longitud* : nat)

1: $res \leftarrow \langle lat, long \rangle$ ▷ $O(copy(lat) + O(copy(long))$

Complejidad: $O(1)$
Justificación: lat y long son nat, tipos primitivos, se pasan por copia, y se consideran operaciones elementales, por lo tanto copiarlos cuesta $O(1) + O(1) = O(1)$

---

**iLatitud**(**in** $c$ : `tupla`(*latitud* : `nat`, *longitud* : `nat`)) → *res* : nat

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

Complejidad: $O(1)$
Justificación: copiar un nat cuesta $O(1)$

---

**iLongitud**(**in** $c$ : `tupla`(*latitud* : `nat`, *longitud* : `nat`)) → *res* : nat

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

Complejidad: $O(1)$
Justificación: copiar un nat cuesta $O(1)$

---

**iDistancia**(**in** $c_1$ : `tupla`(*latitud* : `nat`, *longitud* : `nat`), **in** $c_2$ : `tupla`(*latitud* : `nat`, *longitud* : `nat`)) → *res* : nat

1: $n_1 : nat$ ▷ $O(1)$
2: $n_2 : nat$ ▷ $O(1)$
3: **if** $latitud(c_1) > latitud(c_2)$ **then** ▷ $O(1)$
4: $\quad n_1 \leftarrow (latitud(c_1) - latitud(c_2))$ x $(latitud(c_1) - latitud(c_2))$ ▷ $O(1)$
5: **else**
6: $\quad n_1 \leftarrow (latitud(c_2) - latitud(c_1))$ x $(latitud(c_2) - latitud(c_1))$ ▷ $O(1)$
7: **end if**
8: **if** $longitud(c_1) > longitud(c_2)$ **then**
9: $\quad n_2 \leftarrow (longitud(c_1) - longitud(c_2))$ x $(longitud(c_1) - longitud(c_2))$ ▷ $O(1)$
10: **else**
11: $\quad n_2 \leftarrow (longitud(c_2) - longitud(c_1))$ x $(longitud(c_2) - longitud(c_1))$ ▷ $O(1)$
12: **end if**
13: $res \leftarrow n_1 + n_2$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: todas son operaciones elementales

---

# 2. Módulo Mapa

## 2.1. Interfaz de Mapa

# Interfaz

**Usa**: CONJ, BOOL, COORDENADA.

**se explica con**: MAPA.

**generos**: mapa

### Operaciones basicas de Mapa

COORDENADAS(**in** $m$ : `mapa`) $\rightarrow$ $res$ : `Conj(coord)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} coordenadas(m)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve las coordenadas validas del mapa.
**Aliasing:** res no es modificable.

CREARMAPA() $\rightarrow$ $res$ : `mapa`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} crearMapa()\}$
**Complejidad:** $O(1)$
**Descripción:** Genera un mapa sin coordenadas.

AGREGARCOOR(**in** $c$ : `coord`, **in/out** $m$ : `mapa`)
**Pre** $\equiv$ $\{m = m_0 \wedge c \notin coordenadas(m_0)\}$
**Post** $\equiv$ $\{m =_{\text{obs}} agregarCoor(c, m_0)\}$
**Complejidad:** $O((m.maxLat * m.maxLong * max(nLat, m.maxLat) * nLong) + \#m.coordenadas)$
**Descripción:** Agrega la coordenada c al mapa.

POSEXISTENTE(**in** $c$ : `coord`, **in** $m$ : `mapa`) $\rightarrow$ $res$ : `Bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} posExistente(c, m)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve TRUE, si la coordenada es válida.

HAYCAMINO(**in** $c1$ : `coord`, **in** $c2$ : `coord`, **in** $m$ : `mapa`) $\rightarrow$ $res$ : `Bool`
**Pre** $\equiv$ $\{\{c1, c2\} \subseteq coordenadas(m)\}$
**Post** $\equiv$ $\{res =_{\text{obs}} hayCamino(c1, c2, m)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve TRUE si y solo si hay camino que conecte a c1 con c2.

## 2.2. Representacion de Mapa

`mapa` **se representa con** `estr_mapa`

donde `estr_mapa` es `tupla`(*matriz*: `Vector(Vector(infoCoor))`,
*coordenadas*: `conj(coord)`,
*maxLat*: `nat` , *maxLong*: `nat` )

donde `infoCoor` es `tupla`(*caminos*: `puntero(Vector( Vector(bool) ))` ,
*cValida*: `bool`)

Rep : estr_mapa m $\longrightarrow$ bool

$\text{Rep}(m) \equiv \text{true} \iff ((m.\text{maxLat} = \text{Longitud}(m.\text{matriz}) \wedge_{\text{L}} (\forall i\text{: nat})\ i \leq m.\text{maxLat} \Rightarrow_{\text{L}} m.\text{maxLong} = \text{Longitud}(m.\text{matriz[i]})\ ) \wedge ((\forall i, j\text{: nat})\ i \leq m.\text{maxLat} \wedge t \leq m.\text{maxLong} \Rightarrow_{\text{L}} ((m.\text{matriz[i][t].cValida} = \text{false}) \iff (m.\text{matriz[i][t].caminos} = \text{NULL})) \wedge ((m.\text{matriz[i][t].cValida} = \text{true}) \iff (m.\text{matriz[i][t].caminos} \neq \text{NULL}) \wedge_{\text{L}} *(m.\text{matriz[i][t].caminos})[i][t] = \text{true})) \wedge (\forall c\text{: coord},\ i, t\text{: nat})\ *(m.\text{matriz[Latitud(c)][Longitud(c)].caminos})[i][t] = \text{true} \iff *(m.\text{matriz[i][t].caminos})[\text{Latitud(c)}][\text{Longitud(c)}] = \text{true})) \wedge (\forall c\text{: coord})\ c \in e.\text{coordenadas} \Rightarrow \text{Latitud(c)} \leq m.\text{maxLat} \wedge \text{Longitud(c)} \leq m.\text{maxLong} \wedge_{\text{L}} m.\text{matriz[Latitud(c)][Longitud(c)].cValida} = \text{true}))$

$\text{Abs} : \text{estr\_mapa}\ m \longrightarrow \text{map} \qquad \{\text{Rep}(m)\}$

$\text{Abs}(m) \equiv e : \text{map} \mid \text{m.coordenadas} =_{\text{obs}} \text{coordenas(e)}$

### 2.3. Algoritmos de Mapa

# Algoritmos

---

**iCoordenadas**(**in** $m$ : `estr_mapa`) $\rightarrow$ $res$ : conj($coord$)

1: $res \leftarrow m.coordenadas$ ▷ $O(1)$

Complejidad: $O(1)$

---

---

**iCrearMapa**() $\rightarrow$ $res$ : estr_mapa

1: $res.matriz \leftarrow Vacio()$ ▷ $O(1)$

2: $res.coordenadas \leftarrow Vacio()$ ▷ $O(1)$

3: $res.maxLat \leftarrow 0$ ▷ $O(1)$

4: $res.maxLong \leftarrow 0$ ▷ $O(1)$

Complejidad: $O(1)$

---

**iAgregarCoor**(**in** $c$: `coord`, **in/out** $m$: `estr_mapa`)

1: **if** $c \notin m.coordenadas$ **then** ▷ $O(\#m.coordenadas)$
2: bool $resize \leftarrow false$ ▷ $O(1)$
3: nat $nLat \leftarrow Latitud(c)$ ▷ $O(1)$
4: nat $nLong \leftarrow Longitud(c)$ ▷ $O(1)$
5: **if** $nLat > maxLat \vee nLong > maxLong$ **then** ▷ $O(1)$
6: $redimensionMapa(m, nLat, nLong)$ ▷ $O(max(nLat, m.maxLat) * nLong)$
7: $resize \leftarrow true$ ▷ $O(1)$
8: **end if**
9: puntero(Vector(Vector(bool))) $nuevoCamino \leftarrow \&Vacio()$ ▷ $O(1)$
10: $dimensionarVector(*nuevoCamino, maxLat, maxLong, nLat, nLong)$ ▷ $O(max(nLat, m.maxLat) * max(nLong, m.maxLong))$
11: **if** $nLat > 0 \wedge m.matriz[nLat-1][nLong].cValida$ **then** ▷ $O(1)$
12: $recorrerCaminos(m, nLat-1, nLong, nuevoCamino)$ ▷ $O(m.maxLat * m.maxLong)$
13: **end if**
14: **if** $nLong > 0 \wedge m.matriz[nLat][nLong-1].cValida \wedge \neg(*nuevoCamino[nLat][nLong-1])$ **then** ▷ $O(1)$
15: $recorrerCaminos(m, nLat, nLong-1, nuevoCamino)$ ▷ $O(m.maxLat * m.maxLong)$
16: **end if**
17: **if** $m.matriz[nLat+1][nLong].cValida \wedge \neg(*nuevoCamino[nLat+1][nLong])$ **then** ▷ $O(1)$
18: $recorrerCaminos(m, nLat+1, nLong, nuevoCamino)$ ▷ $O(m.maxLat * m.maxLong)$
19: **end if**
20: **if** $m.matriz[nLat][nLong+1].cValida \wedge \neg(*nuevoCamino[nLat][nLong+1])$ **then** ▷ $O(1)$
21: $recorrerCaminos(m, nLat, nLong+1, nuevoCamino)$ ▷ $O(m.maxLat * m.maxLong)$
22: **end if**
23: $m.matriz[nLat][nLong].cValida \leftarrow true$ ▷ $O(1)$
24: $m.matriz[nLat][nLong].caminos \leftarrow nuevoCamino$ ▷ $O(1)$
25: $*nuevoCamino[nLat][nLong] \leftarrow true$ ▷ $O(1)$
26: $nuevoCamino \leftarrow NULL$ ▷ $O(1)$
27: **if** $resize$ **then** ▷ $O(1)$
28: $redimensionarCaminos(m, nLat, nLong)$ ▷ $O(m.maxLat * m.maxLong * max(nLat, m.maxLat) * nLong)$
29: **end if**
30: **if** $nLat > maxLat$ **then** ▷ $O(1)$
31: $maxLat \leftarrow nLat$ ▷ $O(1)$
32: **end if**
33: **if** $nLong > maxLong$ **then** ▷ $O(1)$
34: $maxLong \leftarrow nLong$ ▷ $O(1)$
35: **end if**
36: $AgregarRapido(m.coordenadas, c)$ ▷ $O(1)$
37: **end if**

Complejidad: $O((m.maxLat * m.maxLong * max(nLat, m.maxLat) * nLong) + \#m.coordenadas)$
Justificación: esto es porque, puede que todas las coordenadas sean validas, y porque si maxLat, maxLong nLat y nLong fueron iguales a N, $O(m.maxLat * m.maxLong * max(nLat, m.maxLat) * nLong)$ sería mayor a $O(max(nLat, m.maxLat) * nLong) + O(max(nLat, m.maxLat) * max(nLong, m.maxLong)) + O(m.maxLat * m.maxLong) + O(m.maxLat * m.maxLong) + O(m.maxLat * m.maxLong * max(nLat, m.maxLat) * nLong)$ $O(m.maxLat * m.maxLong * max(nLat, m.maxLat) * nLong)$, que es $N^4$

---

**iposExistente**(**in** $c$: `coord`, **in** $m$: `estr_mapa`) → $res$ : Bool

1: $res \leftarrow m.matriz[Latitud(c)][Longitud(c)].cValida$ ▷ $O(1)$

Complejidad: $O(1)$

**ihayCamino**(**in** $c1$ : `coord`, **in** $c2$ : `coord`, **in** $m$ : `estr_mapa`) $\rightarrow$ $res$ : Bool

1: $res \leftarrow false$ ▷ $O(1)$
2: **if** $*(m.matriz[Latitud(c1)][Longitud(c1)].caminos)$ $[Latitud(c2)][Longitud(c2)]$ **then** ▷ $O(1)$
3: $res \leftarrow true$ ▷ $O(1)$
4: **end if**

Complejidad: $O(1)$

**Funciones Auxiliares**

---

**redimensionMapa**(**in/out** $m$ : `estr_mapa`, **in** $x$ : `nat`,**in** $y$ : `nat`)

Pre: $\{m = m_0 \wedge m.maxLat < x \wedge m.maxLong < y\}$

Post: $\{m.maxLat = x \wedge m.maxLong = y\}$

```
1:  for i : nat ← m.maxLat to x do                                   ▷ O(x − m.maxLat) = O(x)
2:      AgregarAtras(m.matriz, Vacio())                              ▷ O(1)
3:      for n : nat ← 0 to y do                                      ▷ O(y)
4:          AgregarAtras(m.matriz[i], < NULL, false >)               ▷ O(1)
5:      end for
6:  end for
7:  for i : nat ← 0 to m.maxLat do                                   ▷ O(m.maxLat)
8:      for n : nat ← m.maxLong to y do                              ▷ O(y − m.maxLong) = O(y)
9:          AgregarAtras(m.matriz[n], < NULL, false >)               ▷ O(1)
10:     end for
11: end for
```

Complejidad: $O(x) * O(y) + O(m.maxLat) * O(y) = O(x + m.maxLat) * O(y) = O(max(x, m.maxLat)) * O(y)$

Justificación: consideramos el peor caso cuando m.maxLat y m.maxLong son cero y x e y mayores a 0

---

**dimensionarVector**(**in/out** $v$ : `Vector(Vector(bool))`, **in** $x_1$ : `nat`, **in** $y_1$ : `nat`, **in** $x_2$ : `nat`, **in** $y_2$ : `nat`)

Pre: $\{true\}$

Post: $\{longitud(v) = if(x_1 > x_2)then x_1 else x_2 \wedge paratodob\ b < longitud(v) longitud(v[b]) = if(y_1 > y_2)then y_1 else y_2\}$

```
1:  nat latitud ← x1                                                 ▷ O(1)
2:  nat longitud ← y1                                                ▷ O(1)
3:  if x1 < x2 then                                                  ▷ O(1)
4:      longitud ← x2                                                ▷ O(1)
5:  end if
6:  if y1 < y2 then                                                  ▷ O(1)
7:      nat longitud ← y2                                            ▷ O(1)
8:  end if
9:  for i : nat ← 0 to latitud do                                    ▷ O(latitud)
10:     AgregarAtras(v, Vacio())                                     ▷ O(1)
11:     for l : nat ← 0 to longitud do                               ▷ O(longitud)
12:         AgregarAtras(v[i], false)                                ▷ O(1)
13:     end for
14: end for
```

Complejidad: $O(latitud * longitud)$

Justificación:

---

**recorrerCaminos**(**in** $m$ : `estr_mapa`, **in** $x$ : `nat`, **in** $y$ : `nat`, **in/out** $p$ : `puntero(Vector(Vector(bool)))`)

Pre: $\{true\}$

Post: $\{v[i][b] = if(m.matriz[i][b]thentrueelsev[i][b])\}$

```
1:  puntero Vector(Vector(bool)) camino ← m.matriz[x][y].caminos                ▷ O(1)
2:  for i : nat ← 0 to m.maxLat do                                              ▷ O(m.maxLat)
3:      for l : nat ← 0 to m.maxLong do                                         ▷ O(m.maxLong)
4:          if x ≠ i ∧ y ≠ l then                                               ▷ O(1)
5:              if *(camino)[i][l] then                                         ▷ O(1)
6:                  *(p)[i][l] ← true                                           ▷ O(1)
7:                  m.matriz[i][l].caminos ← p                                  ▷ O(1)
8:              end if
9:          end if
10:     end for
11: end for
```

Complejidad: $O(m.maxLat * m.maxLong)$

Justificación: consideramos el peor caso cuando m.maxLat y m.maxLong son cero y x e y mayores a 0

---

**redimensionarCaminos**(**in** $m$ : `estr_mapa`, **in** $x$ : `nat`, **in** $y$ : `nat`)

Pre: $\{true\}$

Post: $\{longitud(v) = if(x_1 > x_2)thenx_1elsex_2 \wedge paratodob\ b < longitud(v)longitud(v[b]) = if(y_1 > y_2)theny_1elsey_2\}$

```
1:  for i : nat ← 0 to m.maxLat do                                              ▷ O(m.maxLat)
2:      for l : nat ← 0 to m.maxLong do                                         ▷ O(m.maxLong)
3:          if m.matriz[i][l].cValida ∧ (m.matriz[x][y].caminos ≠ m.matriz[i][l].caminos) then   ▷ O(1)
4:              for n : nat ← m.maxLat to x do                                  ▷ O(x − m.maxLat) = O(x)
5:                  AgregarAtras(*(m.matriz[i][l].caminos), Vacio())            ▷ O(1)
6:                  for t : nat ← 0 to y do                                     ▷ O(y)
7:                      AgregarAtras(*(m.matriz[i][l].caminos)[n], false)       ▷ O(1)
8:                  end for
9:              end for
10:             for b : nat ← 0 to m.maxLat do                                  ▷ O(m.maxLat)
11:                 for c : nat ← m.maxLong to y do                             ▷ O(y − m.maxLong) = O(y)
12:                     AgregarAtras(*(m.matriz[i][l].caminos)[b], false)       ▷ O(1)
13:                 end for
14:             end for
15:         end if
16:     end for
17: end for
```

Complejidad: $O(m.maxLat * m.maxLong) * O(x * y) + O(m.maxLat * y) = O(m.maxLat * m.maxLong) * (O(x + m.maxLat) * O(y)) = O(m.maxLat * m.maxLong * max(x, m.maxLat) * y)$

Justificación: consideramos como el peor caso cuando, x e y son mayores que m.maxLat y m.maxLong respectivamente

# 3. Modulo Juego

## 3.1. Interfaz de Juego

# Interfaz

**usa**: NAT, BOOL, CONJUNTO($\alpha$), MULTICONJUNTO($\alpha$), JUGADOR, POKEMON, COORDENADA, MAPA

**se explica con**: JUEGO

**géneros**: juego

### Operaciones básicas de Juego

CREARJUEGO(**in** $m$: `mapa`) $\rightarrow$ $res$ : `juego`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$j =_{\text{obs}}$ crearJuego($m$)}
**Complejidad:** $O(copy(m) + \#(coordenadas(m)) + maxLat(coordenadas(m)) * maxLong(coordenadas(m)))$
**Descripción:** Inicia el juego con el mapa $m$.

AGREGARPOKEMON(**in** $p$: `pokemon`, **in** $c$: `coord`, **in/out** $j$: `juego`)
**Pre** $\equiv$ {puedoAgregarPokemon($c, j$)}
**Post** $\equiv$ {$j =_{\text{obs}}$ agregarPokemon($p, c, j$)}
**Complejidad:** $O(|P|+EC * log(EC))$
**Descripción:** Agrega el pokemon $p$ al juego, en la coordenada $c$.

AGREGARJUGADOR(**in/out** $j$: `juego`) $\rightarrow$ $res$ : `jugador`
**Pre** $\equiv$ {$j =_{\text{obs}} j_0$}
**Post** $\equiv$ {$j =_{\text{obs}}$ agregarJugador($j$) $\wedge$ $res =_{\text{obs}}$ ProxID($j_0$)}
**Complejidad:** $\Theta(J)$, donde $J = \#$(jugadores($j$))
**Descripción:** Agrega un jugador al juego y devuelve su ID.

CONECTARSE(**in** $e$: `jugador`, **in** $c$: `coord`, **in/out** $j$: `juego`)
**Pre** $\equiv$ {$j =_{\text{obs}} j_0 \wedge e \in$ jugadores($j$) $\wedge_{\text{L}}$ $\neg$estaConectado($e$, $j$) $\wedge$ posExistente($c$, mapa($j$))}
**Post** $\equiv$ {$j =_{\text{obs}}$ conectarse($e$, $c$, $j_0$)}
**Complejidad:** $O(log(EC))$
**Descripción:** Conecta al jugador $e$ en la coordenada $c$.

DESCONECTARSE(**in** $e$: `jugador`, **in/out** $j$: `juego`)
**Pre** $\equiv$ {$j =_{\text{obs}} j_0 \wedge e \in$ jugadores($j$) $\wedge_{\text{L}}$ estaConectado($e$, $j$)}
**Post** $\equiv$ {$j =_{\text{obs}}$ desconectarse($e$, $j_0$)}
**Complejidad:** $O(log(EC))$
**Descripción:** Desconecta al jugador $e$ del juego.

MOVERSE(**in** $e$: `jugador`, **in** $c$: `coord`, **in/out** $j$: `juego`)
**Pre** $\equiv$ {$j =_{\text{obs}} j_0 \wedge e \in$ jugadores($j$) $\wedge_{\text{L}}$ estaConectado($e$, $j$) $\wedge$ posExistente($c$, mapa($j$))}
**Post** $\equiv$ {$j =_{\text{obs}}$ moverse($e$, $c$, $j_0$)}
**Complejidad:** $O((PS + PC)*|P|+log(EC))$
**Descripción:** Mueve al jugador $e$ hacia la coordenada $c$.

MAPA(**in** $j$: `juego`) $\rightarrow$ $res$ : `mapa`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ mapa($j$)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve el mapa del juego por referencia.
**Aliasing:** $res$ no es modificable.

JUGADORES(**in** $j$: `juego`) $\rightarrow$ $res$ : `itJugadores`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {alias(esPermutacion?(SecuSuby($res$), jugadores($j$)))}
**Complejidad:** $O(1)$

**Descripción:** Devuelve un iterador para recorrer el conjunto de jugadores del juego. La operación Avanzar del iterador no es $\Theta(1)$.
**Aliasing:** *res* no es modificable.

EstaConectado(**in** $e$: `jugador`, **in** $j$: `juego`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv \{e \in \text{jugadores}(j)\}$
**Post** $\equiv \{res =_{\text{obs}} \text{estaConectado}(e, j)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve un bool que indica si el jugador $e$ esta o no conectado en el juego.

Sanciones(**in** $e$: `jugador`, **in** $j$: `juego`) $\rightarrow$ *res* : `nat`
**Pre** $\equiv \{e \in \text{jugadores}(j)\}$
**Post** $\equiv \{res =_{\text{obs}} \text{sanciones}(e, j)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve la cantidad de sanciones del jugador $e$ en el juego.

Posicion(**in** $e$: `jugador`, **in** $j$: `juego`) $\rightarrow$ *res* : `coord`
**Pre** $\equiv \{e \in \text{jugadores}(j) \wedge_{\text{L}} \text{estaConectado}(e, j)\}$
**Post** $\equiv \{res =_{\text{obs}} \text{posicion}(e, j)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve la posicion del jugador $e$ en el juego.

Pokemons(**in** $e$: `jugador`, **in** $j$: `juego`) $\rightarrow$ *res* : `itDiccTrie`
**Pre** $\equiv \{e \in \text{jugadores}(j)\}$
**Post** $\equiv \{\text{alias}(\text{esPermutacion?}(\text{SecuSuby}(res), \text{ConjSuby}(\text{pokemons}(e, j))))\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve un iterador de tupla (pokemon, cantidad) para recorrer los pokemons capturados por el jugador $e$. [*res* no es modificable.

Expulsados(**in** $j$: `juego`) $\rightarrow$ *res* : `conj(jugador)`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{expulsados}(j)\}$
**Complejidad:** $O(\#(jugadores(j)) + \#(expulsados(j)))$
**Descripción:** Devuelve el conjunto de jugadores expulsados del juego. *res* se devuelve por copia.

PosConPokemons(**in** $j$: `juego`) $\rightarrow$ *res* : `conj(coord)`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{posConPokemons}(j)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve el conjunto de coordenadas del juego que tienen un pokemon. El conjunto se devuelve por referencia.
**Aliasing:** *res* no es modificable.

PokemonEnPos(**in** $c$: `coord`, **in** $j$: `juego`) $\rightarrow$ *res* : `pokemon`
**Pre** $\equiv \{c \in \text{posConPokemons}(j)\}$
**Post** $\equiv \{res =_{\text{obs}} \text{pokemonEnPos}(c, j)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve el pokemon que se encuentra en la coordenada $c$ del juego. *res* se devuelve por copia.

CantMovimientosParaCaptura(**in** $c$: `coord`, **in** $j$: `juego`) $\rightarrow$ *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 para capturar al pokemon de la coordenada $c$.

PuedoAgregarPokemon(**in** $c$: `coord`, **in** $j$: `juego`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{puedoAgregarPokemon}(c, j)\}$
**Complejidad:** $O(\#(PosConPokemons(j)))$
**Descripción:** Devuelve un bool que indica si es posible agregar un pokemon en la coordenada $c$.

HayPokemonCercano(**in** $c$: `coord`, **in** $j$: `juego`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ hayPokémonCercano($c$, $j$)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve un bool que indica si hay un pokemon cerca de la coordenada $c$.

PosPokemonCercano(**in** $c$: `coord`, **in** $j$: `juego`) $\rightarrow$ $res$ : `coord`
**Pre** $\equiv$ {hayPokemonCercano($c$, $j$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ posPokemonCercano($c$, $j$)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve la coordenada del pokemon cercano a la coordenada $c$.

EntrenadoresPosibles(**in** $c$: `coord`, **in** $es$: `conj(jugador)`, **in** $j$: `juego`) $\rightarrow$ $res$ : `conj(jugador)`
**Pre** $\equiv$ {hayPokemonCercano($c$, $j$) $\wedge$ $es \subseteq$ jugadoresConectados($j$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ entrenadoresPosibles($c$, $j$)}
**Complejidad:** $\Theta(EC)$
**Descripción:** Devuelve el conjunto de jugadores que pueden atrapar al pokemon cercano a la coordenada $c$.

IndiceRareza(**in** $p$: `pokemon`, **in** $j$: `juego`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {$p \in$ todosLosPokemons($j$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ indiceRareza($p$, $j$)}
**Complejidad:** $\Theta(|P|)$, donde $P$ es el tipo de pokemon mas largo definido en el juego.
**Descripción:** Devuelve el indice de rareza del tipo de pokemon $p$.

CantPokemonsTotales(**in** $j$: `juego`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ cantPokemonsTotales($j$)}
**Complejidad:** $\Theta(1)$
**Descripción:** Devuelve la cantidad de pokemons del juego.

CantMismaEspecie(**in** $p$: `pokemon`, **in** $j$: `juego`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ cantMismaEspecie($p$, $ps$)}
**Complejidad:** $\Theta(|P|)$, donde $P$ es el tipo de pokemon mas largo definido en el juego.
**Descripción:** Devuelve la cantidad de pokemons del tipo $p$ del juego.

### 3.2. Representacion de Juego

# Representación

`juego` **se representa con** `estr`

donde `estr` es `tupla`(*mapa*: `mapa`,
*jugadores*: `vector(infoJugador)`,
*jugadoresValidos*: `vector(bool)`,
*pokemons*: `diccTrie(pokemon, infoPokemon)`,
*posConPokemons*: `conj(coord)`,
*infoDePos*: `arreglo_dimensionable(arreglo_dimensionable(infoPos))`,
*cantTotalPokemons*: `nat`)

donde `infoJugador` es `tupla`(*conectado*: `bool`, *sanciones*: `nat`, *posicion*: `coord`,
*capturados*: `diccTrie(pokemon, nat)`, *cantCapturados*: `nat`,
*enRangoDe*: `itColaPr(nat, nat)`,
*enPos*: `itConj(nat)`)

donde `infoPokemon` es `tupla`(*cantSalvajes*: `nat`, *cantCapturados*: `nat`)

donde `infoPos` es `tupla`(*hayPokemon*: `bool`, *pokemon*: `pokemon`,
*jugadoresEnPos*: `conj(nat)`,
*cantMovsEsperando*: `nat`,
*jugadoresEsperando*: `colaPr(nat, nat)`)

Rep : estr $\longrightarrow$ bool
Rep($j$) $\equiv$ true $\Longleftrightarrow$

1. *jugadores* y *jugadoresValidos* tienen la misma longitud.
2. Todo *jugador* en *jugadoresValidos*, tiene true sii *sanciones* en *jugadores* es menor que 5.
3. Todo *jugador* en *jugadoresValidos*, tiene false sii *sanciones* en *jugadores* es mayor o igual que 5.
4. Las claves de todos los diccionarios *capturados* de *jugadores*, estan incluidas en las claves de *pokemons*.
5. Toda posicion de *jugadores* pertenece a coordenadas del *mapa*.
6. *posConPokemons* esta incluido en coordenadas del *mapa*.
7. La matriz $infoDePos$ tiene tamano $maxLatitud \times maxLongitud$, donde maxLatitud y maxLongitud son los valores maximos que toma una coordenada del mapa para su componente latitud y longitud respectivamente.
8. Toda coordenada de *posConPokemons* tiene true en la componente *hayPokemon* de la matriz $infoDePos$ (accediendo con latitud, longitud) (y viceversa).
9. Todo *jugador* de *jugadores* pertenece al conjunto *jugadoresEnPos* de la matriz $infoDePos$, accediendo con latitud, longitud de la *posicion* del jugador. Y ademas, el iterador *enPos* apunta a dicho conjunto.
10. Para todo pokemon del dicc *pokemons*, *cantCapturados* es la suma de los significados de los dicc *capturados* de todos los jugadores de *jugadores*.
11. Para todo pokemon del dicc *pokemons*, *cantSalvajes* es la cantidad de apariciones en la matriz $infoDePos$.
12. *cantTotalPokemons* es la suma de *cantCapturados* y *cantSalvajes* de todos los pokemons del dicc *pokemons*.

Abs : estr $e$ $\longrightarrow$ juego {Rep($e$)}
Abs($e$) $\equiv$ $j$ : juego |
mapa($j$) = e.mapa $\wedge$
jugadores($j$) = armarJugadores(e.jugadoresValidos) $\wedge$
expulsados($j$) = armarExpulsados(e.jugadoresValidos) $\wedge$
((∀$g$: jugador) $g \in$ jugadores($j$) $\Rightarrow$
(sanciones($g$, $j$) = e.jugadores[g].sanciones $\wedge$
pokemons($g$, $j$) = armarCapturados(e.jugadores[g].capturados, claves(e.jugadores[g].capturados)) $\wedge$
estaConectado($g$, $j$) = e.jugadores[g].conectado $\wedge$
posicion($g$, $j$) = e.jugadores[g].posicion)) $\wedge$
posConPokemons($j$) = e.posConPokemons $\wedge$
((∀$c$: coord) $c \in$ posConPokemons($j$) $\Rightarrow$
(pokemonEnPos($c$, $j$) = e.infoDePos[Latitud(c)][Longitud(c)].pokemon $\wedge$
cantMovimientosParaCaptura($c$, $j$) = e.jugadores[Latitud(c)][Longitud(c)].cantMovsEsperando))

### 3.3. Algoritmos de Juego

## Algoritmos

**iCrearJuego**(**in** $m$ : `mapa`) $\rightarrow$ $res$ : estr

1: $res.mapa \leftarrow m$ ▷ $O(copy(m))$
2: $res.jugadores \leftarrow Vacia()$ ▷ $\Theta(1)$
3: $res.jugadoresValidos \leftarrow Vacia()$ ▷ $\Theta(1)$
4: $res.pokemons \leftarrow Vacio()$ ▷ $\Theta(1)$
5: $res.posConPokemons \leftarrow Vacio()$ ▷ $\Theta(1)$
6: $res.cantTotalPokemons \leftarrow 0$ ▷ $\Theta(1)$
7:
8: $maxLat, maxLong : nat$ ▷ $\Theta(1)$
9: $maxLat \leftarrow 0$ ▷ $\Theta(1)$
10: $maxLong \leftarrow 0$ ▷ $\Theta(1)$
11: $itPos : itConj(coord)$ ▷ $\Theta(1)$
12: $itPos \leftarrow CrearIt(Coordenadas(m))$ ▷ $\Theta(1)$
13:
14: **while** $HaySiguiente(itPos)$ **do** ▷ $O(\#(coordenadas(m)))$
15: **if** $Latitud(Siguiente(itPos)) > maxLat$ **then** ▷ $\Theta(1)$
16: $maxLat \leftarrow Latitud(Siguiente(itPos))$ ▷ $\Theta(1)$
17: **end if**
18: **if** $Longitud(Siguiente(itPos)) > maxLong$ **then** ▷ $\Theta(1)$
19: $maxLong \leftarrow Longitud(Siguiente(itPos))$ ▷ $\Theta(1)$
20: **end if**
21: $Avazar(itPos)$ ▷ $\Theta(1)$
22: **end while**
23:
24: $i : nat$ ▷ $\Theta(1)$
25: $i \leftarrow 0$ ▷ $\Theta(1)$
26:
27: $res.infoDePos \leftarrow CrearArreglo(maxLat)$ ▷ $O(maxLatitud(coordenadas(m)))$
28: **while** $i < maxLat$ **do** ▷ $O(maxLatitud(coordenadas(m)))$
29: $res.infoDePos[i] \leftarrow CrearArreglo(maxLong)$ ▷ $O(maxLongitud(coordenadas(m)))$
30: $i++$ ▷ $\Theta(1)$
31: **end while**

Complejidad: $O(copy(m) + \#(coordenadas(m)) + maxLat(coordenadas(m)) * maxLong(coordenadas(m)))$
Justificación: Se recorre el conjunto de coordenadas del mapa. Se crea la matriz de infoDePos con maxLat arreglos de tamano maxLong, donde maxLat y maxLong son los maximos valores que toma una coordenada del mapa para la latitud y longitud respectivamente. Y se copia el mapa con un costo copy(m).

**iAgregarPokemon**(**in** $p$: `pokemon`, **in** $c$: `coord`, **in/out** $j$: `estr`)

```
1:  jug : jugador                                                          ▷ Θ(1)
2:  capt : nat                                                             ▷ Θ(1)
3:  lat, long : nat                                                        ▷ Θ(1)
4:  cj : conj(jugador)                                                     ▷ Θ(1)
5:  itCJ : itConj(jugador)                                                 ▷ Θ(1)
6:  cp : colaPr(nat, nat)                                                  ▷ Θ(1)
7:  itCP : itColaPr(jugador)                                               ▷ Θ(1)
8:
9:  cj ← EntrenadoresPosibles(c, Vacio(), j)                               ▷ Θ(1)
10: itCJ ← CrearIt(cj)                                                     ▷ Θ(1)
11: while HaySiguiente(itCJ) do                                            ▷ O(EC)
12:     jug ← Siguiente(itCJ)                                              ▷ Θ(1)
13:     capt ← j.jugadores[jug].cantCapturados                             ▷ Θ(1)
14:     itCP ← Encolar(⟨capt, jug⟩, cp)                                    ▷ O(logEC)
15:     j.jugadores[jug].enRangoDe ← itCP                                  ▷ Θ(1)
16:     Avazar(itCJ)                                                       ▷ Θ(1)
17: end while
18: lat ← Latitud(c)                                                       ▷ Θ(1)
19: long ← Longitud(c)                                                     ▷ Θ(1)
20: j.infoDePos[lat][long].hayPokemon ← true                               ▷ Θ(1)
21: j.infoDePos[lat][long].pokemons ← p                                    ▷ Θ(1)
22: j.infoDePos[lat][long].jugadoresEsperando ← cp                         ▷ Θ(1)
23:
24: cantS, cantC : nat                                                     ▷ Θ(1)
25: cantS ← 0                                                              ▷ Θ(1)
26: cantC ← 0                                                              ▷ Θ(1)
27: if Definido(p, j.pokemons) then                                        ▷ O(|P|)
28:     cantS ← Obtener(p, j.pokemons).cantSalvajes                        ▷ O(|P|)
29:     cantC ← Obtener(p, j.pokemons).cantCapturados                      ▷ O(|P|)
30: end if
31: cantS++                                                                ▷ Θ(1)
32: cantC++                                                                ▷ Θ(1)
33: Definir(p, ⟨cantS, cantC⟩, j.pokemons)                                 ▷ O(|P|)
34:
35: AgregarRapido(j.posConPokemons, c)                                     ▷ Θ(1)
36:
37: j.cantTotalPokemons++                                                  ▷ Θ(1)
```

Complejidad: $O(|P|+EC*log(EC))$
Justificación: Se agrega un pokemon al dicc sobre trie, con costo $|P|$. Luego, los jugadores en rango a $c$ se agregan a la cola de prioridad del pokemon.

---

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

```
1: jug : nat                                                                                          ▷ Θ(1)
2: jug ← Longitud(j.jugadores)                                                                        ▷ Θ(1)
3:
4: AgregarAtras(j.jugadores, ⟨false, 0, CrearCoord(0,0), Vacio(), CrearIt(Vacia()), CrearIt(Vacio())⟩)   ▷ Θ(1)
5:
6: res ← jug                                                                                          ▷ Θ(1)
```

Complejidad: $O(J)$
Justificación: Se agrega un elemento al vector jugadores, en el peor caso se debe redimensionar el vector.

**iConectarse**(**in** $e$ : `jugador`, **in** $c$: `coord`, **in/out** $j$ : `estr`)

```
1:  cpk : coord                                                                          ▷ Θ(1)
2:  lat, long, capt : nat                                                                ▷ Θ(1)
3:  itC : itColaPr(jugador)                                                              ▷ Θ(1)
4:
5:  if HayMas(j.jugadores[e].enRangoDe) then                                             ▷ Θ(1)
6:      Eliminar(j.jugadores[e].enRangoDe)                                               ▷ O(log(EC))
7:  end if
8:
9:  j.jugadores[e].enRangoDe ← CrearIt(Vacia())                                          ▷ Θ(1)
10: if HayPokemonCercano(c, j) then                                                      ▷ Θ(1)
11:     cpk ← PosPokemonCercano(c, j)                                                    ▷ Θ(1)
12:     lat ← Latitud(cpk)                                                               ▷ Θ(1)
13:     long ← Longitud(cpk)                                                             ▷ Θ(1)
14:     capt ← j.jugadores[e].cantCapturados                                             ▷ Θ(1)
15:     itC ← Encolar(⟨capt, e⟩ , j.infoDePos[lat][long].jugadoresEsperando)             ▷ O(log(EC))
16:     j.infoDePos[lat][long].cantMovsEsperando ← 0                                     ▷ Θ(1)
17:     j.jugadores[e].enRangoDe ← itC                                                   ▷ Θ(1)
18: end if
19:
20: itJug : itConj(jugador)                                                              ▷ Θ(1)
21: itJug ← AgregarRapido(j.infoDePos[Latitud(c)][Longitud(c)].jugadoresEnPos, e)        ▷ Θ(1)
22: j.jugadores[e].enPos ← itJug                                                         ▷ Θ(1)
23:
24: j.jugadores[e].posicion ← c                                                          ▷ Θ(1)
25: j.jugadores[e].conectado ← true                                                      ▷ Θ(1)
```

Complejidad: $O(log(EC))$
Justificación: Si el jugador estaba en el rango de un pokemon, eliminarlo de la cola de prioridad cuesta log EC. De la misma forma, si entra en el rango de un pokemon, agregar a la nueva cola cuesta log EC.

---

**iDesconectar**(**in** $e$ : `jugador`, **in/out** $j$ : `estr`)

```
1:  if HayMas(j.jugadores[e].enRangoDe) then                                             ▷ Θ(1)
2:      Eliminar(j.jugadores[e].enRangoDe)                                               ▷ O(log(EC))
3:  end if
4:  EliminarSiguiente(j.jugadores[e].enPos)                                              ▷ O(1)
5:
6:  j.jugadores[e].enRangoDe ← CrearIt(Vacia())                                          ▷ Θ(1)
7:  j.jugadores[e].enPos ← CrearIt(Vacio())                                              ▷ Θ(1)
8:
9:  j.jugadores[e].conectado ← false                                                     ▷ Θ(1)
```

Complejidad: $O(log(EC))$
Justificación: Si el jugador estaba en el rango de un pokemon, eliminarlo de la cola de prioridad cuesta log EC.

**iMoverse**(**in** $e$ : `jugador`, **in** $c$ : `coord`, **in/out** $j$ : `estr`)

1: $itPC : itDiccTrie(pokemon, nat)$ ▷ $\Theta(1)$
2: $itPS : itConj(coord)$ ▷ $\Theta(1)$
3: $itC : itColaPr(jugador)$ ▷ $\Theta(1)$
4: $pk : pokemon$ ▷ $\Theta(1)$
5: $capt, tcapt : nat$ ▷ $\Theta(1)$
6: $cpk : coord$ ▷ $\Theta(1)$
7: $jugCapt : jugador$ ▷ $\Theta(1)$
8: $lat, long, movsEsp, cantC : nat$ ▷ $\Theta(1)$
9: **if** $HayMas(j.jugadores[e].enRangoDe)$ **then** ▷ $\Theta(1)$
10: $Eliminar(j.jugadores[e].enRangoDe)$ ▷ $O(log(EC))$
11: $j.jugadores[e].enRangoDe \leftarrow CrearIt(Vacia())$ ▷ $\Theta(1)$
12: **end if**
13: $j.jugadores[e].posicion \leftarrow c$ ▷ $\Theta(1)$
14: **if** $HayPokemonCercano(c, j)$ **then** ▷ $\Theta(1)$
15: $cpk \leftarrow PosPokemonCercano(c, j)$ ▷ $\Theta(1)$
16: $lat \leftarrow Latitud(cpk)$ ▷ $\Theta(1)$
17: $long \leftarrow Longitud(cpk)$ ▷ $\Theta(1)$
18: $capt \leftarrow j.jugadores[e].cantCapturados$ ▷ $\Theta(1)$
19: $itC \leftarrow Encolar(\langle capt, e \rangle, j.infoDePos[lat][long].jugadoresEsperando)$ ▷ $O(log(EC))$
20: $j.infoDePos[lat][long].cantMovsEsperando \leftarrow 0$ ▷ $\Theta(1)$
21: $j.jugadores[e].enRangoDe \leftarrow itC$ ▷ $\Theta(1)$
22: **end if**
23: **if** $debeSancionarse(e, c, j)$ **then** ▷ $O(1)$
24: $j.jugadores[e].sanciones++$ ▷ $\Theta(1)$
25: **if** $j.jugadores[e].sanciones >= 5$ **then** ▷ $O(1)$
26: $itPC \leftarrow CreatIt(j.jugadores[e].capturados)$ ▷ $\Theta(1)$
27: **while** $HaySiguiente(itPC)$ **do** ▷ $O(PC)$
28: $pk \leftarrow SiguienteClave(itPC)$ ▷ $\Theta(1)$
29: $capt \leftarrow SiguienteSignificado(itPC)$ ▷ $\Theta(1)$
30: $tcapt \leftarrow Obtener(pk, j.pokemons).cantCapturados$ ▷ $O(|P|)$
31: $Obtener(pk, j.pokemons).cantCapturados \leftarrow tcapt - capt$ ▷ $O(|P|)$
32: $j.cantTotalPokemons \leftarrow j.cantTotalPokemons - capt$ ▷ $\Theta(1)$
33: $Avazar(itPC)$ ▷ $\Theta(1)$
34: **end while**
35: **end if**
36: **end if**
37: $itPS \leftarrow CreatIt(j.posConPokemons)$ ▷ $\Theta(1)$
38: **while** $HaySiguiente(itPS)$ **do** ▷ $O(PS)$
39: $cpk \leftarrow Siguiente(itPS)$ ▷ $\Theta(1)$
40: $lat \leftarrow Latitud(cpk)$ ▷ $\Theta(1)$
41: $long \leftarrow Longitud(cpk)$ ▷ $\Theta(1)$
42: $pk \leftarrow j.infoDePos[lat][long].pokemon$ ▷ $\Theta(1)$
43: $movsEsp \leftarrow j.infoDePos[lat][long].cantMovsEsperando$ ▷ $\Theta(1)$
44: **if** $movsEsp >= 9$ **then** ▷ $O(1)$
45: $jugCapt \leftarrow Proximo(j.infoDePos[lat][long].jugadoresEsperando)$ ▷ $\Theta(1)$
46: $cantC \leftarrow 0$ ▷ $\Theta(1)$
47: **if** $Definido(pk, j.jugadores[jugCapt].capturados)$ **then** ▷ $O(|P|)$
48: $cantC \leftarrow Obtener(pk, j.jugadores[jugCapt].capturados)$ ▷ $O(|P|)$
49: **end if**
50: $cantC++$ ▷ $\Theta(1)$
51: $Definir(pk, cantC, j.jugadores[jugCapt].capturados)$ ▷ $O(|P|)$
52: $j.jugadores[jugCapt].cantCapturados++$ ▷ $\Theta(1)$
53: $Obtener(pk, j.pokemons).cantSalvajes--$ ▷ $O(|P|)$
54: $Obtener(pk, j.pokemons).cantCapturados++$ ▷ $O(|P|)$
55: **end if**
56: $Avazar(itPS)$ ▷ $\Theta(1)$
57: **end while**

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

**iMapa**(**in** $j$ : `estr`) $\rightarrow$ $res$ : mapa

1: $res \leftarrow j.mapa$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Se devuelve el mapa por referencia. $res$ no es modificable.

---

**iJugadores**(**in** $j$ : `estr`) $\rightarrow$ $res$ : itJugadores

1: $res \leftarrow CrearIt(j.jugadoresValidos)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Se devuelve un iterador a los jugadores. La operación Avanzar() del iterador no es O(1). El iterador no es modificable.

---

**iEstaConectado**(**in** $e$ : `jugador`, **in** $j$ : `estr`) $\rightarrow$ $res$ : bool

1: $res \leftarrow j.jugadores[e].conectado$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Se accede al vector con la informacion del jugador usando el indice $e$ en $\Theta(1)$.

---

**iSanciones**(**in** $e$ : `jugador`, **in** $j$ : `estr`) $\rightarrow$ $res$ : nat

1: $res \leftarrow j.jugadores[e].sanciones$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Se accede al vector con la informacion del jugador usando el indice $e$ en $\Theta(1)$.

---

**iPosicion**(**in** $e$ : `jugador`, **in** $j$ : `estr`) $\rightarrow$ $res$ : coord

1: $res \leftarrow j.jugadores[e].posicion$ ▷ $O(1)$

Complejidad: $\Theta(1)$
Justificación: Se accede al vector con la informacion del jugador usando el indice $e$ en $\Theta(1)$.

---

**iPokemons**(**in** $e$ : `jugador`, **in** $j$ : `estr`) $\rightarrow$ $res$ : itDiccTrie

1: $res \leftarrow CrearIt(j.jugadores[e].capturados)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Se devuelve un iterador a los pokemons capturados por el jugador $e$. El iterador no es modificable.

---

**iExpulsados**(**in** $j$ : `estr`) $\rightarrow$ $res$ : conj(jugador)

```
1:  eliminados : conj(jugador)                                          ▷ Θ(1)
2:  i : nat                                                             ▷ Θ(1)
3:  longitud : nat                                                      ▷ Θ(1)
4:
5:  eliminados ← Vacio()                                                ▷ Θ(1)
6:  i ← 0                                                               ▷ Θ(1)
7:  longitud ← Longitud(j.jugadoresValidos)                             ▷ Θ(1)
8:
9:  while i < longitud do                                               ▷ O(#(jugadores(j)) + #(expulsados(j)))
10:     if ¬j.jugadoresValidos[e] then                                  ▷ Θ(1)
11:         AgregarRapido(i, eliminados)                                ▷ Θ(1)
12:     end if
13:     i++                                                             ▷ Θ(1)
14: end while
15:
16: res ← eliminados                                                    ▷ Θ(1)
```

Complejidad: $O(\#(jugadores(j)) + \#(expulsados(j)))$
Justificación: Se recorre el vector de jugadores para identificar los eliminados. El conjunto de eliminados se devuelve por copia.

---

**iPosConPokemons**(**in** $j$ : `estr`) $\rightarrow$ $res$ : conj(coord)

```
1:  res ← j.posConPokemons                                              ▷ O(1)
```

Complejidad: $O(1)$
Justificación: Se devuelve el conjunto de coordenadas con pokemons por referencia. $res$ no es modificable.

---

**iPokemonEnPos**(**in** $c$: `coord`, **in** $j$ : `estr`) $\rightarrow$ $res$ : pokemon

```
1:  res ← j.infoDePos[Latitud(c)][Longitud(c)].pokemon                  ▷ Θ(1)
```

Complejidad: $\Theta(1)$
Justificación: Se accede a la matriz (arreglo de arreglo) con la informacion de la posicion en $\Theta(1)$. Se devuelve el pokemon por copia.

---

**iCantMovimientosParaCaptura**(**in** $c$: `coord`, **in** $j$ : `estr`) $\rightarrow$ $res$ : nat

```
1:  res ← j.infoDePos[Latitud(c)][Longitud(c)].cantMovsEsperando        ▷ Θ(1)
```

Complejidad: $\Theta(1)$
Justificación: Se accede a la matriz (arreglo de arreglo) con la informacion de la posicion en $\Theta(1)$.

---

**iPuedoAgregarPokemon**(**in** $c$: `coord`, **in** $j$: `estr`) $\rightarrow$ $res$ : bool

```
1:  hayPK : bool                                                          ▷ Θ(1)
2:  hayPK ← false                                                         ▷ Θ(1)
3:  it : itConj(coord)                                                    ▷ Θ(1)
4:  it ← CrearIt(j.posConPokemons)                                        ▷ Θ(1)
5:
6:  while HaySiguiente(it) do                                             ▷ O(#(posConPokemons(j)))
7:      if Distancia(c, Siguiente(it)) <= 25 then                         ▷ Θ(1)
8:          hayPK ← true                                                  ▷ Θ(1)
9:      end if
10:     Avazar(it)                                                        ▷ Θ(1)
11: end while
12:
13: res ← ¬hayPK                                                          ▷ Θ(1)
```

Complejidad: $O(\#(posConPokemons(j)))$
Justificación: Se recorre el conjunto de posiciones con pokemons.

---

**iHayPokemonCercano**(**in** $c$: `coord`, **in** $j$: `estr`) $\rightarrow$ $res$ : bool

```
1:  hayPK : bool                                                          ▷ Θ(1)
2:  hayPK ← false                                                         ▷ Θ(1)
3:  lat, long : nat                                                       ▷ Θ(1)
4:  pc : conj(coord)                                                      ▷ Θ(1)
5:  pc ← PosCercanas(c, j.mapa)                                           ▷ Θ(1)
6:  it : itConj(coord)                                                    ▷ Θ(1)
7:  it ← CrearIt(pc)                                                      ▷ Θ(1)
8:
9:  while HaySiguiente(it) ∧ ¬hayPk do                                    ▷ O(cantPosCercanas)
10:     lat ← Latitud(Siguiente(it))                                      ▷ Θ(1)
11:     long ← Longitud(Siguiente(it))                                    ▷ Θ(1)
12:     if j.infoDePos[lat][long].hayPokemon then                         ▷ Θ(1)
13:         hayPK ← true                                                  ▷ Θ(1)
14:     end if
15:     Avazar(it)                                                        ▷ Θ(1)
16: end while
17:
18: res ← hayPK                                                           ▷ Θ(1)
```

Complejidad: $O(cantPosCercanas) = O(13) = O(1)$
Justificación: Se recorre el conjunto de posiciones cercanas a $c$, donde por cercana se entiende a distancia menor a 4. Luego este conjunto esta acotado por 13, que es el mayor numero de coordenadas existentes posibles que este a distancia menor a 4. Luego se puede decir que la complejidad es O(1).

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

```
1:  hayPK : bool                                                          ▷ Θ(1)
2:  hayPK ← false                                                         ▷ Θ(1)
3:  lat, long : nat                                                       ▷ Θ(1)
4:  pc : conj(coord)                                                      ▷ Θ(1)
5:  pc ← PosCercanas(c, j.mapa)                                           ▷ Θ(1)
6:  it : itConj(coord)                                                    ▷ Θ(1)
7:  it ← CrearIt(pc)                                                      ▷ Θ(1)
8:
9:  while HaySiguiente(it) ∧ ¬hayPk do                                    ▷ O(cantPosCercanas)
10:     lat ← Latitud(Siguiente(it))                                      ▷ Θ(1)
11:     long ← Longitud(Siguiente(it))                                    ▷ Θ(1)
12:     if j.infoDePos[lat][long].hayPokemon then                         ▷ Θ(1)
13:         hayPK ← true                                                  ▷ Θ(1)
14:     end if
15:     Avazar(it)                                                        ▷ Θ(1)
16: end while
17:
18: res ← Siguiente(it)                                                   ▷ Θ(1)
```

Complejidad: $O(cantPosCercanas) = O(13) = O(1)$

Justificación: Se recorre el conjunto de posiciones cercanas a $c$, donde por cercana se entiende a distancia menor a 4. Luego este conjunto esta acotado por 13, que es el mayor numero de coordenadas existentes posibles que este a distancia menor a 4. Luego se puede decir que la complejidad es O(1).

**iEntrenadoresPosibles**(**in** $c$: `coord`, **in** $es$: `conj(jugador)`, **in** $j$: `juego`) → $res$ : conj(jugador)

```
1:  cj : conj(jugador)                                                          ▷ Θ(1)
2:  cj ← Vacio()                                                                ▷ Θ(1)
3:  lat, long : nat                                                             ▷ Θ(1)
4:  pc : conj(coord)                                                            ▷ Θ(1)
5:  pc ← PosCercanas(c, j.mapa)                                                 ▷ Θ(1)
6:  itJug : itConj(jugador)                                                     ▷ Θ(1)
7:  itPos : itConj(coord)                                                       ▷ Θ(1)
8:  itPos ← CrearIt(pc)                                                         ▷ Θ(1)
9:
10: while HaySiguiente(itPos) do                                                ▷ O(cantPosCercanas)
11:     if HayCamino(c, Siguiente(itPos), j.mapa) then                          ▷ Θ(1)
12:         lat ← Latitud(Siguiente(itPos))                                     ▷ Θ(1)
13:         long ← Longitud(Siguiente(itPos))                                   ▷ Θ(1)
14:         itJug ← CrearIt(j.infoDePos[lat][long].jugadoresEnPos)              ▷ Θ(1)
15:
16:         while HaySiguiente(itJug) do                                        ▷ O(cantJugEnPos)
17:             if j.jugadores[Siguiente(itJug)].conectado then                 ▷ Θ(1)
18:                 AgregarRapido(Siguiente(itJug), cj)                         ▷ Θ(1)
19:             end if
20:             Avazar(itJug)                                                   ▷ Θ(1)
21:         end while
22:     end if
23:     Avazar(itPos)                                                           ▷ Θ(1)
24: end while
25:
26: res ← cj                                                                    ▷ Θ(1)
```

Complejidad: $O(cantPosCercanas * SUM(cantJugEnPos)) = O(13 * EC) = O(EC)$
Justificación: Se recorre el conjunto de posiciones cercanas a $c$, donde por cercana se entiende a distancia menor a 4. Luego este conjunto esta acotado por 13, que es el mayor numero de coordenadas existentes posibles que este a distancia menor a 4. Luego se puede decir que un ciclo de *cantPosCercanas* esta acotado por O(1). Por otro lado, SUM(*cantJugEnPos*) es, en el peor caso, igual a $EC$ dado que la union de los conjuntos de jugadores en posiciones cercanas sera el conjunto de jugadores esperando capturar.

---

**iIndiceRareza**(**in** $p$: `pokemon`, **in** $j$: `estr`) → $res$ : nat

```
1:  cantMismaEsp : nat                                                          ▷ Θ(1)
2:  cantTotal : nat                                                             ▷ Θ(1)
3:
4:  cantMismaEsp ← CantMismaEspecie(p, j)                                       ▷ Θ(|P|)
5:  cantTotal ← CantPokemonsTotales(j)                                          ▷ Θ(1)
6:
7:  res ← 100 − (100 * cantMismaEsp/cantTotal)                                  ▷ Θ(1)
```

Complejidad: $\Theta(|P|)$
Justificación: La complejidad está dada por la llamanda a la función CantMismaEspecie$(p, j)$, la cual tiene un costo de $\Theta(|P|)$.

---

**iCantPokemonsTotales**(**in** $p$: `pokemon`, **in** $j$: `estr`) → $res$ : nat

```
1:  res ← j.cantTotalPokemon                                                    ▷ Θ(1)
```

Complejidad: $\Theta(1)$
Justificación: Se accede a un campo de la estructura.

---

**iCantMismaEspecie**(**in** $p$: `pokemon`, **in** $j$: `estr`) $\rightarrow$ $res$ : nat

1: $res \leftarrow Obtener(p, j.pokemons).cantSalvajes + Obtener(p, j.pokemons).cantCapturados$ ▷ $\Theta(|P|+|P|)$

Complejidad: $\Theta(|P|)$, donde $P$ es el tipo de pokemon mas largo definido en el juego.
Justificación: Se accede al diccionario sobre trie en $\Theta(|P|)$, donde $P$ es el tipo de pokemon mas largo definido en el juego.

---

**PosCercanas**(**in** $c$: `coord`, **in** $m$: `mapa`) $\rightarrow$ $res$ : conj(coord)

1: $cj : conj(coord)$ ▷ $\Theta(1)$
2: $cj \leftarrow Vacio()$ ▷ $\Theta(1)$
3:
4: $res \leftarrow cj$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificación: Se crean un numero constante de coordenadas (13) y se agregan al conjunto resultado. La operacion *posExistente* de mapa tiene costo O(1).

# 4. Módulo Cola de Prioridad$(nat, nat)$

## 4.1. Interfaz de Cola de Prioridad

# Interfaz

**parámetros formales**
**género** $Tupla < nat, nat >$
**función** $\bullet < \bullet$(**in** $a$: $Tupla < nat, nat >$, **in** $b$: $Tupla < nat, nat >$) $\rightarrow res$ : $bool$
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} \pi_1(a) < \pi_1(b) \vee (\pi_1(a) = \pi_1(b) \wedge \pi_2(a) \leq \pi_2(b))\}$
**Complejidad:** $\Theta(1)$

**se explica con**: Cola de prioridad $(\alpha)$ .

**géneros**: `colaPr`.

### Operaciones básicas de Cola de Prioridad

Vacia() $\rightarrow res$ : `colaPr`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}}$ vacia$\}$
**Complejidad:** $O(1)$
**Descripción:** genera una cola de prioridad vacia.

Encolar(**in** $a$: `<nat,nat>`, **in/out** $h$: `colaPr`) $\rightarrow res$ : `iTcolaPr`
**Pre** $\equiv \{h =_{\text{obs}} h_0\}$
**Post** $\equiv \{h =_{\text{obs}} encolar(a, h_o)\}$
**Complejidad:** $O(log(H)), siendo\ H =\ Total\ de\ elementos\ en\ la\ cola$
**Descripción:** agrega el elemento $a$ en la cola de prioridad
**Aliasing:** el elemento $a$ se agrega por copia.

Proximo(**in** $h$: `colaPr`) $\rightarrow res$ : `<nat,nat>`
**Pre** $\equiv \{\neg$ vacio?(h)$\}$
**Post** $\equiv \{h =_{\text{obs}} proximo(a, h_o)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve el proximo elemento en la cola de prioridad.
**Aliasing:** Res es devuelto por copia

Vacia?(**in** $h$: `colaPr` ) $\rightarrow res$ : `bool`
**Pre** $\equiv$ {True}
**Post** $\equiv \{res =_{\text{obs}}$ vacia?$(h)\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve true si y solo si $h$ es vacia.

Desencolar(**in/out** $h$: `colaPr`)
**Pre** $\equiv \{h =_{\text{obs}} h_0 \wedge \neg vacio?(h)\}$
**Post** $\equiv \{h =_{\text{obs}}$ desencolar$(h_0)\}$
**Complejidad:** $O(log(H))$
**Descripción:** Devuelve una referencia, no modificable, al elemento con mas prioridad de la cola de prioridad.

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

**TAD** Cola de Prioridad$(\alpha)$

**otras operaciones**

sacar : $\alpha\ e \times$ ColaPrior$(\alpha)\ c \longrightarrow$ ColaPrior$(\alpha)$ {pertenece(e,h)}

pertenece : $\alpha\ e \times$ ColaPrior($\alpha$) $c$ $\longrightarrow$ Bool

**axiomas**

pertenece(e,h) $\equiv$

```
if vacio?(h) then
    false
else
    if e = proximo(h)  then
        True
    else
        pertenece(e,desencolar(h))
    end if
end if
```

sacar(e,encolar(a,h)) $\equiv$

```
if e = a then
    h
else

    if e = proximo(h) then
        encolar(a, desencolar(h))
    else
        encolar(a, sacar(e,h))
    end if
end if
```

**Fin TAD**

## Operaciones basicas del Iterador

Se provee un Iterador unidireccional modificable (implementadas solo las funciones que se utilizaran)

CrearIt(**in** $c$: `colaPr`) $\rightarrow$ $res$ : `iTcolaPr`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{*(res.cola) =_{\text{obs}} c \wedge *(res.nodo) =_{\text{obs}} *(c.arbol) \wedge alias(EsPermutacion(SecuSubY(res), c)\}$
**Complejidad:** $O(1)$
**Descripción:** Crea un iterador de manera que el actual es el elemento de mas prioridad de la cola.

Actual(**in** $it$: `iTcolaPr`) $\rightarrow$ $res$ : `nodo`
**Pre** $\equiv$ {HayMas?($it$)}
**Post** $\equiv$ $\{alias(res =_{\text{obs}} (actual(it))\}$
**Complejidad:** $O(1)$
**Descripción:** Devuelve el elemento siguiente del iterador

Agregar(**in/out** $it$: `iTcolaPr`, **in** $a$: `<nat,nat>`)
**Pre** $\equiv$ {HayMas?($it$)}
**Post** $\equiv$ $\{res =_{\text{obs}} (Agregar(it, a)))\}$
**Complejidad:** $O(Log(H)), H$ = cantidad de nodos en $it.cola$
**Descripción:** Agrega , por copia, el elemento a la cola indicada por el iterador. El elemento se agrega segun su prioridad y queda referenciado por el iterador.

Eliminar(**in/out** $it$: `iTcolaPr`)
**Pre** $\equiv$ {HayMas?($it$)}
**Post** $\equiv$ $\{res =_{\text{obs}} (Eliminar(it))\}$
**Complejidad:** $O(Log(H)), H$ = cantidad de nodos en $it.cola$
**Descripción:** Elimina el actual de la cola iterada.

### 4.2. Representacion de Cola de prioridad

# Representación

`colaPr` **se representa con** `estr`

donde `estr` es `tupla`(*arbol*: `Puntero(nodo)` , *nodos*: `nat`
)
donde `nodo` es `tupla`(*raiz*: `clave`, *izq*: `puntero(nodo)`
, *der*: `puntero(nodo)`
, *padre*: `puntero(nodo)`
)
donde `clave` es `tupla`(*valor*: `nat`, *num*: `nat`)

Invariante de Representacion: Sea e estr.
1. Que la estructura sea balanceada. Esto es un arbol binario lleno en el que todas las hojas están a profundidad n o n-1, para algun n.
2. Todo subarbol es un Heap. Vale Rep para cada subarbol
3. Es izquierdista, osea el ultimo nivel esta lleno desde la izquierda.
4. La clave (raiz) de cada nodo es menor que la de sus hijos, si los tiene, y en caso de igualdad el nodo(padre) tiene menor num que el de sus hijos , si los tiene.

Abs : estr $c$ $\longrightarrow$ colaPr $\{\text{Rep}(c)\}$
Abs($c$) $\equiv$ $c_0$ : colaPrior |
vacia?($c_0$) = (c.nodos = 0 $\wedge$ c.arbol = NULL) $\wedge_{\text{L}}$
($\neg$ vacia?($c_0$) $\Rightarrow_{\text{L}}$proximo($c_0$) = *(c.arbol).raiz $\wedge$
desencolar($c_0$) = (.$^{\text{En}}$ la estructura c, quita la coneccion del nodo h.arbol (osea, intercambia la ultima hoja con ese nodo y por ultimo quita la coneccion de ese mismo nodo), que es el elemento con mas prioridad y ordena las conecciones para mantener el InvRep. Osea, pone la ultima hoja como el nodo con mas prioridad (lo que ahora apunta h.arbol), y la va bajando con algunos de sus hijos (el de mas prioridad en caso de tener dos) hasta que no tiene mas hijos o hasta que todos sus hijos sean con menos prioridad que el.")

## Representación de Iterador de Cola de Prioridad

`iTcolaPr` **se representa con** `estr`

donde `estr` es `tupla`(*cola*: `Puntero(colaPr)` , *nodo*: `Puntero(nodo)`)

Invariante de Representacion: Sea it un iTcolaPr.
1. El nodo de it.nodo es null o bien peretenece a colaPr. 2. Vale el REp de colaPr en el elemento apuntado por iTcolaPr.cola
Abstraccion:

### 4.3. Algoritmos de Cola de prioridad

# Algoritmos

**Algorithm 1** iVacia() $\rightarrow$ *res* : `estr`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ vacio}

1: *res.arbol* $\leftarrow$ *NULL* ▷ $O(1)$
2: *res.nodos* $\leftarrow$ 0 ▷ $O(1)$
3: *complejidad total* = $O(1)$

**Algorithm 2** iEncolar(**in** $a$: `<nat,nat>`, **in/out** $h$: `estr`) $\rightarrow$ $res$ : `iTcolaPr`
**Pre** $\equiv$ $\{m =_{\text{obs}} m_0\}$
**Post** $\equiv$ $\{m =_{\text{obs}} \text{agregar}(e, m_0)\}$

1: $it \leftarrow CrearIt(h)$ ▷ $O(1)$
2: $res \leftarrow Agregar(it, a)$ ▷ $O(log(h.nodos))$
3: *complejidad total* = $O(log(h.nodos))$

**Algorithm 3** iVacia?(h) $\rightarrow$ $res$ : `estr`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{vacio}\}$

1: $res \leftarrow (h.nodos = 0)$ ▷ $O(1)$
2: *complejidad total* = $O(1)$

**Algorithm 4** Proximo(**in** $h$: `cola`) $\rightarrow$ $res$ : `<nat,nat>`
**Pre** $\equiv$ $\{\neg$ vacio?(h)$\}$
**Post** $\equiv$ $\{h =_{\text{obs}} proximo(a, h_o)\}$

1: $res \leftarrow copy(*(h.arbol).raiz)$ ▷ $O(1)$
2: *complejidad total* = $O(1)$

**Algorithm 5** IDESENCOLAR(**in**/**out** $h$ : `colaPr`)
**Pre** $\equiv \{h =_{\text{obs}} h_0 \wedge \neg$ vacio?(h)$\}$
**Post** $\equiv \{m =_{\text{obs}}$ Desencolar$(h_0)\}$

1: **if** [ **then** ▷ $O(1)$] $h.nodos = 1$
2: $h \leftarrow NULL$ ▷ $O(1)$
3: **else**
4: $p : Puntero(nodo)$ ▷ $O(1)$
5: $*p \leftarrow *(PadreUltimaHoja(h, h.nodos))$ ▷ $O(log(h.nodos))$
6: **if** [ **then** ▷ $O(1)$]$p.der = NULL$
7: $Swap(*((*p).izq), *(h.arabol), h)$ ▷ $O(1)$
8: $*(a.padre).izq \leftarrow NULL$ ▷ $O(1)$
9: **else**
10: $Swap(*((*p).der), *(h.arabol), h)$ ▷ $O(1)$
11: $*(a.padre).der \leftarrow NULL$ ▷ $O(1)$
12: **end if**
13: $q : Puntero(Nodo)$ ▷ $O(1)$
14: $q \leftarrow \&(h.arbol)$ ▷ $O(1)$
15: **while** [ **do** ▷ $O(1)$] $(*(q).izq \neq NULL \vee *(q).der \neq NULL) \wedge ((*q).raiz.valor > *(*(q).izq).raiz.valor \vee (*q).raiz.valor = *(*(q).izq).raiz.valor \wedge *(q).raiz.num > *(*(q).izq).raiz.num) \vee (*(q).raiz.valor > *(*(q).der).raiz.valor \vee *(q).raiz.valor = *(*(q).der).raiz.valor \wedge *(q).raiz.num > *(*(q).der).raiz.num)$
16: **if** [ **then** ▷ $O(1)$] $*(q).izq = NULL$
17: $Swap(*q, *((*q).der), h)$ ▷ $O(1)$
18: **else**
19: **if** [ **then** ▷ $O(1)$] $*(q).der = NULL$
20: $Swap(*q, *((*q).izq), h)$ ▷ $O(1)$
21: **else**
22: **if** [ **then** ▷ $O(1)$] $(*(*(q).izq).raiz.valor < *(*(q).der).raiz.valor \vee *(*(q).izq).raiz.valor = *(*(q).der).raiz.valor \wedge *(*(q).izq).raiz.num < *(*(q).der).raiz.num)$
23: $Swap(*q, *((*q).izq), h)$ ▷ $O(1)$
24: **else**
25: $Swap(*q, *((*q).der), h)$ ▷ $O(1)$
26: **end if**
27: **end if**
28: **end if**
29: **end while**
30: $*(h).nodos \leftarrow *(h).nodos - 1$ ▷ $O(1)$
31: **end if**
32: *complejidad del ciclo* $= O(1) * O(log(h.nodos)) = O(1 * log(h.nodos)) = O(log(h.nodos))$
33: *complejidad total* $= O(log(h.nodos)) + O(log(h.nodos)) + O(1) = 2 * O(log(h.nodos)) = O(log(h.nodos))$
34:

**Algorithm 6** IPADREULTIMAHOJA(**in** $h$: `estr`, **in** $n$: `nat`) $\rightarrow$ $res$ : `nodo`
**Pre** $\equiv$ {Existe al menos un elemento en h y 1<=n<=h.nodos+1}
**Post** $\equiv$ {Devuelve el nodo padre al que se le insertara la proxima hoja}

```
1:  iT : itLista(nat)                                                    ▷ O(1)
2:  iT ← CrearIt(binario(n))                                             ▷ O(log(n))
3:  actual : puntero(Nodo)                                               ▷ O(1)
4:  padre : puntero(Nodo)                                                ▷ O(1)
5:  padre ← NULL                                                         ▷ O(1)
6:  actual ← h.arbol                                                     ▷ O(1)
7:  Avanzar(iT)                                                          ▷ O(1)
8:  while [ do                                          ▷ O(1)] HaySiguiente(iT)
9:      padre ← actual                                                   ▷ O(1)
10:     if [ then                                       ▷ O(iT)] siguiente(j) = 0
11:         actual ← actual.izq                                          ▷ O(1)
12:     else
13:         actual ← actual.der                                          ▷ O(1)
14:     end if
15:     Avanzar(iT)                                                      ▷ O(1)
16: end while
17: res ← *padre                                                         ▷ O(1)
18: complejidad del ciclo = O(1) * O(log(n)) = O(1 * log(n)) = O(log(n))
19: complejidad total = O(log(n)) + O(log(n)) + O(1) =2 * O(log(n)) = O(log(n))
20:
```

**Algorithm 7** BINARIO(**in** $a$: `nat`) $\rightarrow$ $res$ : $lista(nat)$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {
}

```
1:  ls : lista(nat)                                                      ▷ O(1)
2:  ls ← Vacia()                                                         ▷ O(1)
3:  i : nat                                                              ▷ O(1)
4:  i ← a                                                                ▷ O(1)
5:  while [ do                                          ▷ O(1)] (i div 2 ≠ 1)
6:      AgregarAdelante(ls, i mod 2)                                     ▷ O(1)
7:      i ← i div 2                                                      ▷ O(1)
8:  end while
9:  AgregarAdelante(ls, i mod 2)                                         ▷ O(1)
10: AgregarAdelante(ls, 1)                                               ▷ O(1)
11: complejidad del ciclo = O(1) * O(log(a)) = O(1 * log(a)) = O(log(a))
12: complejidad total = O(log(a)) + O(1) = O(log(a))
13:
```

**Algorithm 8** iSacar(**in** $a$: `Nodo`, **in/out** $h$: `colaPr`)
**Pre** $\equiv \{h =_{\text{obs}} h_0 \wedge \text{pertenece(a,h)}\}$
**Post** $\equiv \{h =_{\text{obs}} sacar(a, h_0)\}$

1: **if** [ **then** ▷ $O(1)$] $h.nodos = 1$ $O(1)$
2: $h.arbol \leftarrow NULL$ ▷ $O(1)$
3: **else**
4: $p : Puntero(nodo)$ ▷ $O(1)$
5: $p \leftarrow \&(PadreUltimaHoja(h, h.nodos))$ ▷ $O(log(h.nodos))$
6: **if** [ **then** ▷ $O(1)$]$p.der = NULL$
7: $Swap(*((*p).izq), a, h)$ ▷ $O(1)$
8: $*(a.padre).izq \leftarrow NULL$ ▷ $O(1)$
9: **else**
10: $Swap(*((*p).der), a, h)$ ▷ $O(1)$
11: $*(a.padre).der \leftarrow NULL$ ▷ $O(1)$
12: **end if**
13: $q : Puntero(Nodo)$ ▷ $O(1)$
14: $q \leftarrow \&(p)$ ▷ $O(1)$
15: **while** [ **do** ▷ $O(1)$] $(*(q).izq \neq NULL \vee *(q).der \neq NULL) \wedge ((*q).raiz.valor > *(*(q).izq).raiz.valor \vee (*q).raiz.valor = *(*(q).izq).raiz.valor \wedge *(q).raiz.num > *(*(q).izq).raiz.num) \vee (*(q).raiz.valor > *(*(q).der).raiz.valor \vee *(q).raiz.valor = *(*(q).der).raiz.valor \wedge *(q).raiz.num > *(*(q).der).raiz.num)$
16: **if** [ **then** ▷ $O(1)$] $*(q).izq = NULL$
17: $Swap(*q, *((*q).der), h)$ ▷ $O(1)$
18: **else**
19: **if** [ **then** ▷ $O(1)$] $*(q).der = NULL$
20: $Swap(*q, *((*q).izq), h)$ ▷ $O(1)$
21: **else**
22: **if** [ **then** ▷ $O(1)$] $(*(*(q).izq).raiz.valor < *(*(q).der).raiz.valor \vee *(*(q).izq).raiz.valor = *(*(q).der).raiz.valor \wedge *(*(q).izq).raiz.num < *(*(q).der).raiz.num)$
23: $Swap(*q, *((*q).izq), h)$ ▷ $O(1)$
24: **else**
25: $Swap(*q, *((*q).der), h)$ ▷ $O(1)$
26: **end if**
27: **end if**
28: **end if**
29: **end while**
30: $*(h).nodos \leftarrow *(h).nodos - 1$ ▷ $O(1)$
31: **end if**
32: *complejidad del ciclo* $= O(1) * O(log(h.nodos)) = O(1 * log(h.nodos)) = O(log(h.nodos))$
33: *complejidad total* $= O(log(h.nodos)) + O(log(h.nodos)) + O(1) = 2 * O(log(h.nodos)) = O(log(h.nodos))$
34:

**Algorithm 9** Swap(**in/out** $a$: `Nodo`, **in/out** $b$: `Nodo`, **in/out** $h$: `colaPr`)
**Pre** $\equiv \{\}$
**Post** $\equiv$ {Intercambia los dos nodos adecuadamente, manteniendo correctos los iteradores que los referencian. Modifica la raiz de la cola}

1: $hijoIzq : Puntero(Nodo)$ ▷ $O(1)$
2: $hijoDer : Puntero(Nodo)$ ▷ $O(1)$
3: $father : Puntero(Nodo)$ ▷ $O(1)$
4: **if** [ **then** ▷ $O(1)$] $*(h.arbol) = a$
5: $*(h.arbol) \leftarrow b$ ▷ $O(1)$
6: **else**
7: **if** [ **then** ▷ $O(1)$] $*(h.arbol) = b$
8: $*(h.arbol) \leftarrow b$ ▷ $O(1)$
9: **end if**
10: **end if**

```
1: if [ then▷ O(1)] *(a.izq) = b
2:     hijoDer ← a.der                                              ▷ O(1)
3:     father ← a.padre                                             ▷ O(1)
4:     a.izq ← b.izq                                                ▷ O(1)
5:     a.der ← b.der                                                ▷ O(1)
6:     a.padre ← b                                                  ▷ O(1)
7:     b.izq ← a                                                    ▷ O(1)
8:     b.der ← hijoDer                                              ▷ O(1)
9:     b.padre ← father                                             ▷ O(1)
10: else
11:     if [ then                                                   ▷ O(1)] *(a.der) = b
12:         hijoIzq ← a.izq                                         ▷ O(1)
13:         father ← a.padre                                        ▷ O(1)
14:         a.izq ← b.izq                                           ▷ O(1)
15:         a.der ← b.der                                           ▷ O(1)
16:         a.padre ← b                                             ▷ O(1)
17:         b.der ← a                                               ▷ O(1)
18:         b.izq ← hijoIzq                                         ▷ O(1)
19:         b.padre ← father                                        ▷ O(1)
20:     else
21:         if [ then                                               ▷ O(1)] *(b.izq) = a
22:             hijoDer ← b.der                                     ▷ O(1)
23:             father ← b.padre                                    ▷ O(1)
24:             b.izq ← a.izq                                       ▷ O(1)
25:             b.der ← a.der                                       ▷ O(1)
26:             b.padre ← a                                         ▷ O(1)
27:             a.izq ← b                                           ▷ O(1)
28:             a.der ← hijoDer                                     ▷ O(1)
29:             a.padre ← father                                    ▷ O(1)
30:         else
31:             if [ then                                           ▷ O(1)] *(b.der) = a
32:                 hijoIzq ← b.izq                                 ▷ O(1)
33:                 father ← b.padre                                ▷ O(1)
34:                 b.izq ← a.izq                                   ▷ O(1)
35:                 b.der ← a.der                                   ▷ O(1)
36:                 b.padre ← a                                     ▷ O(1)
37:                 a.der ← b                                       ▷ O(1)
38:                 a.izq ← hijoIzq                                 ▷ O(1)
39:                 a.padre ← father                                ▷ O(1)
40:             else
41:                 hijoIzq ← a.izq                                 ▷ O(1)
42:                 hijoDer ← a.der                                 ▷ O(1)
43:                 father ← a.padre                                ▷ O(1)
44:                 a.izq ← b.izq                                   ▷ O(1)
45:                 a.der ← b.der                                   ▷ O(1)
46:                 a.padre ← b.padre                               ▷ O(1)
47:                 b.izq ← hijoIzq                                 ▷ O(1)
48:                 b.der ← hijoDer                                 ▷ O(1)
49:                 b.padre ← father                                ▷ O(1)
50:             end if
51:         end if
52:     end if
53: end if
54: complejidad total = O(1)
55:
```

## 4.4. Algoritmos de iterador

**Algorithm 10** CrearIt(**in** $c$: `colaPr`) $\rightarrow$ $res$ : `iTcolaPr`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{*(res.cola) =_{\text{obs}} c \wedge *(res.nodo) =_{\text{obs}} *(c.arbol) \wedge alias(EsPermutacion(SecuSubY(res), c)\}$

1: $res \leftarrow < \&c, (c.arbol) >$ ▷ $O(1)$

**Algorithm 11** Actual(**in** $it$: `iTcolaPr`) $\rightarrow$ $res$ : `nodo u <it.cola, it.nodo>`
**Pre** $\equiv$ {HayMas?$(it)$}
**Post** $\equiv$ $\{alias(res =_{\text{obs}} (actual(it))\}$

1: $res \leftarrow < it.cola, it.nodo >$ ▷ $O(1)$
2: *complejidad total* $= O(1)$

**Algorithm 12** Agregar(**in/out** $it$: `iTcolaPr`, **in** $a$: `<nat,nat>`)
**Pre** $\equiv$ {HayMas?$(it)$}
**Post** $\equiv$ $\{res =_{\text{obs}} (Agregar(it, a)))\}$
**Complejidad:** $\theta(Log(H)), H =$ *cantidad de nodos en it.cola*
**Descripción:** Agrega , por copia, el elemento a la cola indicada por el iterador. El elemento es agrega segun su prioridad y queda referenciado por el iterador.

1: **if** [ **then**▷ $O(1)$] $*(it.cola).arbol = NULL$
2: $*(it.cola).arbol \leftarrow \& < copiar(a), null, null, null >$ ▷ $O(1)$
3: $*(it.cola).nodos \leftarrow 1$ ▷ $O(1)$
4: $it.nodo \leftarrow *(it.cola).arbol$
5: **else**
6: $p : Puntero(nodo)$ ▷ $O(1)$
7: $p \leftarrow \&(PadreUltimaHoja(*(it.cola), *(it.cola).nodos + 1))$ ▷ $O(log(*(it.cola)).nodos + 1)) = O(log(*(it.cola).nodos))$
8: $aux : nodo$ ▷ $O(1)$
9: $aux \leftarrow < copy(a), NULL, NULL, p >$ ▷ $O(1)$
10: **if** [ **then** ▷ $O(1)]p.izq = NULL$
11: $(*p).izq \leftarrow \&aux$ ▷ $O(1)$
12: **else**
13: $(*p).der \leftarrow \&aux$ ▷ $O(1)$
14: **end if**
15: $q : Puntero(Nodo)$ ▷ $O(1)$
16: $q \leftarrow \&aux$ ▷ $O(1)$
17: **while** [ **do** ▷ $O(1)$] $((*q).padre \neq NULL) \wedge ((*q).raiz.valor < (*q).padre.valor \vee ((*q).raiz.valor = (*q).padre.valor \wedge (*q).raiz.num < (*q).padre.num))$
18: $Swap(*q, *(*q).padre), *(it.cola))$
19: **end while**
20: $*(it.cola).nodos \leftarrow *(it.cola).nodos + 1$ ▷ $O(1)$
21: $(it.nodo) \leftarrow \&q$
22: **end if**
23: *complejidad del ciclo* $= O(1) * O(log(*(it.cola).nodos)) = O(1 * log(*(it.cola).nodos)) = O(log(*(it.cola).nodos))$
24: *complejidad total* $= O(log(*(it.cola).nodos)) + O(log(*(it.cola).nodos)) + O(1) = 2 * O(log(*(it.cola).nodos)) = O(log(*(it.cola).nodos))$

**Algorithm 13** ELIMINAR(**in/out** $it$ : `iTcolaPr`)
**Pre** $\equiv$ {HayMas?($it$)}
**Post** $\equiv$ {$res =_{\text{obs}} (Eliminar(it))$}
**Complejidad:** $\theta(Log(H)), H =$ *cantidad de nodos en it.cola*
**Descripción:** Elimina el actual del iterador. Se indefinen las referencias al mismo.

1: $Sacar(*(it.cola), *(it.nodo))$ ▷ $O(log(*(it.cola).nodos))$
2: *complejidad total* = $O(log(*(it.cola).nodos))$

# 5. Módulo Diccionario Trie(string, $\alpha$)

## 5.1. Interfaz

**parámetros Formales**
**géneros** $\alpha$
**función** COPIAR(**in** $a$: $\alpha$) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} a$}
**Complejidad:** $O(copy(a))$
**Descripción:** función de copia de $\alpha$'s
**se explica con**: DICCIONARIO(STRING, $\alpha$).
**géneros**: **diccTrie(string, $\alpha$), itDiccTrie(string,$\alpha$).**

### Operaciones básicas de diccionario trie

CREARDICC() $\rightarrow$ $res$ : `diccTrie(string, α)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ vacío()}
**Complejidad:** $O(1)$
**Descripción:** crea un nuevo diccionario vacío.

DEFINIR(**in** $c$: `string`, **in** $s$: $\alpha$, **in/out** $d$: `diccTrie(string, α)`)
**Pre** $\equiv$ {$d =_{\text{obs}} d_0$}
**Post** $\equiv$ {$d =_{\text{obs}}$ definir($d_0$, $c$, $s$)}
**Complejidad:** $O(|c| + copy(s))$
**Descripción:** define la clave $c$ con el significado $s$ en el diccionario.
**Aliasing:** el significado $s$ se agrega por copia.

DEFINIDO?(**in** $c$: `string`, **in** $d$: `estr_diccTrie`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {res $=_{\text{obs}}$ def?($c$, $d$)}
**Complejidad:** O(|c|)
**Descripción:** Devuelve true si y solo si $c$ está definido en el diccionario

OBTENER(**in** $c$: `string`, **in** $d$: `diccTrie(string, α)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {def?($c$, $d$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ obtener($c$, $d$))}
**Complejidad:** $O(|c|)$
**Descripción:** devuelve el significado de la clave $c$ en $d$.
**Aliasing:** $res$ es modificable si y sólo si $d$ es modificable.

BORRAR(**in** $c$: `string`, **in/out** $d$: `diccTrie(string, α)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {$d =_{\text{obs}} d_0$, def?($c$, $d$)}
**Post** $\equiv$ {$d =_{\text{obs}}$ borrar($c$, $d$))}
**Complejidad:** $O(|c|)$
**Descripción:** elimina la clave $c$ y su significado de $d$.

CLAVES(**in** $d$: `diccTrie(string, α)`) $\rightarrow$ $res$ : `itDiccTrieTrie(string, nat)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ CrearItMod(d))}
**Complejidad:** $O(1)$
**Descripción:** devuelve un iterador a las claves y el significado de cada uno

### Operaciones del iterador

El iterador que presentamos permite modificar el diccionario recorrido, eliminando elementos. Sin embargo, si el diccionario es no modificable, no se pueden utilizar las funciones de eliminar.

CrearIt(**in** $d$: `DiccTrie(String, ` $\alpha$`)`) $\rightarrow$ $res$ : `itDiccTrie(`$String, \alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$true$} alias(esPermutacion(SecuSuby($res$), $d$)) $\wedge$ vacia?(Anteriores($res$)) crea un iterador bidireccional del diccionario, que apunta al primer elemento del mismo. $O(CL * long(k))$ Donde $CL$ es la cantidad de claves de d y $k$ la palabra mas larga de d hay aliasing entre los significados en el iterador y los del diccionario
HaySiguiente(**in** $it$: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ $res$ : $bool$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$true$} $res =_{\text{obs}}$ haySiguiente?($it$) devuelve `true` si y solo si en el iterador todavia quedan elementos para avanzar. $O(1)$
HayAnterior(**in** $it$: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ $res$ : $bool$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$true$} $res =_{\text{obs}}$ hayAnterior?($it$) devuelve `true` si y solo si en el iterador todavia quedan elementos para retroceder. $O(1)$
Siguiente(**in** $it$: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ $res$ : `tupla(String, `$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {HaySiguiente?($it$)} alias($res =_{\text{obs}}$ Siguiente($it$)) devuelve el elemento siguiente del iterador. $O(1)$ $res$.significado es modificable si y solo si $it$ es modificable, por aliasing. En cambio, $res$.clave no es modificable.
SiguienteClave(**in** $it$: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ $res$ : $String$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {HaySiguiente?($it$)} alias($res =_{\text{obs}}$ Siguiente($it$).clave) devuelve la clave del elemento siguiente del iterador. $O(1)$ $res$ no es modficable.

SiguienteSignificado(**in** *it*: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ *res* : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {HaySiguiente?(*it*)} alias(*res* $=_{\text{obs}}$ Siguiente(*it*).significado) devuelve el significado del elemento siguiente del iterador. $O(1)$ *res* es modificable si y solo si *it* es modficable, por aliasing.
Anterior(**in** *it*: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ *res* : `tupla(clave:` *String*`, significado:` $\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {HayAnterior?(*it*)} alias(*res* $=_{\text{obs}}$ Anterior(*it*)) devuelve el elemento anterior del iterador. $O(1)$ *res*.significado es modificable si y solo si *it* es modificable, por aliasing. En cambio, *res*.clave no es modificable.
AnteriorClave(**in** *it*: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ *res* : *String*
**Pre** $\equiv$ {true}
**Post** $\equiv$ {HayAnterior?(*it*)} alias(*res* $=_{\text{obs}}$ Anterior(*it*).clave) devuelve la clave del elemento anterior del iterador. $O(1)$ *res* no es modficable.
AnteriorSignificado(**in** *it*: `itDiccTrie(String,`$\alpha$`)`) $\rightarrow$ *res* : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {HayAnterior?(*it*)} alias(*res* $=_{\text{obs}}$ Anterior(*it*).significado) devuelve el significado del elemento anterior del iterador. $O(1)$ *res* es modificable si y solo si *it* es modficable, por aliasing.
Avanzar(**in**/**out** *it*: `itDiccTrie(String,`$\alpha$`)`)
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$it = it_0 \wedge$ HaySiguiente?(*it*)} *it* $=_{\text{obs}}$ Avanzar($it_0$) avanza a la posicion siguiente del iterador. $O(1)$
Retroceder(**in**/**out** *it*: `itDiccTrie(String,`$\alpha$`)`)
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$it = it_0 \wedge$ HayAnterior?(*it*)} *it* $=_{\text{obs}}$ Retroceder($it_0$) retrocede a la posicion anterior del iterador. $O(1)$

## 5.2. Representacion del DiccTrie

Representamos cada nodo del árbol con una tupla que contiene un puntero a su significado, que podrá ser NULL si no hay un significado asociado a ese nodo, y un arreglo de 256 punteros a nodos hijos, en el que elemento de índice $i$ representa al nodo hijo correspondiente a un char, el cual será NULL si no se definió ninguna clave.
El diccionario se representa con una tupla que mantiene una referencia al nodo raíz.

`diccTrie(string,` $\alpha$ **se representa con** `estr`

donde `estr` es `tupla`(*raíz*: `puntero(nodo)`
, *claves*: `lista(string)`
, *significado*: `lista`($\alpha$) )
donde `nodo` es `tupla`(*significado*:: `itLista`($\alpha$),
, *hijos*: *arreglo*$_e$*statico*[256]*depuntero*(*nodo*) )

**Invariante de representación:**

1. El árbol no debe tener ciclos, ni nodos con dos padres.
2. El árbol no debe tener nodos repetidos.
3. La cantidad de claves deben coincidir con la cantidad de significado
4. Si la raiz es NULL entonces #claves == #significados == 0
5. Los nodos terminales tienen obtener no nulo.

**Función de abstracción:**

1. El diccionario tiene la misma cantidad de claves que la estructura.
2. Para cada clave del diccionario:
   *a*) La clave está definida en la estructura.
   *b*) El obtener de la clave en el diccionario es el mismo que en la estructura.

Abs : estr_dicctrie $e$ $\longrightarrow$ dicc(string, $\alpha$) {Rep($e$)}
Abs($e$) $\equiv$ d
#(claves($d$)) $=_{\text{obs}}$ $e$.#Claves $\wedge_{\text{L}}$ 1.
($\forall c$: string)(def?($c$, $d$) $\Leftrightarrow$ ( 2.

$\neg$(ObtenerDeLaEstructura($c$, $e$.raíz) $=_{\text{obs}}$ NULL) $\wedge_{\text{L}}$ 2. a)

obtener($c$, $d$) $=_{\text{obs}}$ *(ObtenerDeLaEstructura($c$, $e$.raíz)) 2. b)

))

ObtenerDeLaEstructura : string $\times$ estr_nodo $\longrightarrow$ puntero($\alpha$)

ObtenerDeLaEstructura($c, n$) $\equiv$ **if** vacia?($c$) **then**
$n$.obtener
**else**
**if** $n$.hijos[(prim($c$))] $=_{\text{obs}}$ NULL **then**
NULL
**else**
ObtenerDeLaEstructura(fin($c$), $n$.hijos[(prim($c$))])
**fi**
**fi**

### 5.3. Representacion del itDiccTrie

El iterador del diccionario es simplemente un par de iteradores a las listas correspondientes. Lo unico que hay que pedir es que se satisfaga el Rep de este para de listas. Se explica con el iterador unidireccional modificable itDiccTrie(*String*, $\alpha$)itdt)

`itDicTrie(`*String*`, `$\alpha$`)` **se representa con** `itdt`

donde `itdt` es `tupla(`*claves*: `itLista(`*String*`)`, *significados*: `puntero(`$\alpha$`))`

## Algoritmos

### Algoritmos del DiccionarioTrie

---

**iCrearDicc** $\rightarrow$ *res* : estr_diccTrie

1: $res \leftarrow \langle raiz : iNuevoNodo(), claves.vacio(), significados.vacio() \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(copy(a))$

---

---

**iNuevoNodo** $\rightarrow$ *res* : estr_diccTrie

1: $res \leftarrow \langle Significado : NULL, hijos : CrearArreglo() \rangle$ ▷ $\Theta(1)$
2: **for** $i : nat \leftarrow 0$ to 255 **do** ▷ $\Theta(255)$
3: $res$.hijos[$i$] $\leftarrow$ NULL ▷ $\Theta(1)$
4: **end for**

Complejidad: $\Theta(copy(a))$
Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(256)$. Aplicando álgebra de órdenes:
5: $\Theta(1) + \Theta(255) + \Theta(1) = \Theta(1)$

---

**iDefinir**(**in**/**out** $d$: `dicTrie`, **in** $c$: *String*, **in** $s$: $\alpha$)

1: ***Nodo** *actual* $\leftarrow$ &($d$.raiz) ▷ $\Theta(1)$
2: **for** $i$ : *nat* $\leftarrow$ 0 to **Longitud**($c$) **do**
3: **if** *actual*→hijos[($c[i]$)] $=_{\text{obs}}$ NULL **then**
4: *actual*→hijos[($c[i]$)] $\leftarrow$ &(**iNuevoNodo**()) ▷ $\Theta(1)$
5: **end if**
6: *actual* $\leftarrow$ *actual*→hijos[($c[i]$)] ▷ $\Theta(1)$
7: **end for**
8: **if** ( **thenHaySiguiente?**(*actual*.significado))
9: **Siguiente**(*actual*.significado) $\leftarrow$ $s$
10: **else**
11: *actual*.significado $\leftarrow$ **AgregarAtras**(d.significados, s)
12: **end if**
13: *Complejidad del For :* O(longitud(c))
14: *actual*→obtener $\leftarrow$ &(Copiar($s$)) ▷ $\Theta(copy(s))$

Complejidad del For: $\Theta(longitud(c))$
Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(copy(a))$. Aplicando álgebra de órdenes:
15: $O(1) + |c| * (O(1) + O(1) + (O(1)$ ó $0) + O(1)) + O(copy(s)) + O(1) =$
16: $O(1) + |c| * O(1) + O(copy(s)) + O(1) =$
17: $|c| * O(1) + O(copy(s)) =$
18: $O(|c|) + O(copy(s)) =$
19: $O(|c| + copy(s))$

---

**iObtener**(**in** $d$: `dicTrie`, **in** $c$: *String*)→ $res$ : $\alpha$

1: ***Nodo** *actual* $\leftarrow$ &($d$.raiz) ▷ $\Theta(1)$
2: **for** $i$ : *nat* $\leftarrow$ 0 to **Longitud**($c$) **do**
3: *actual* $\leftarrow$ *actual*→hijos[($c[i]$)] ▷ $\Theta(1)$
4: **end for**
5: *Complejidad del For :* $\Theta(1)$
6: $res \leftarrow$ *(*actual*→*obtener*) ▷ $\Theta(1)$

Complejidad: $\Theta(long(c))$
Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(long(c))$. Aplicando álgebra de órdenes:
7: $O(1) + |c| * (O(1) + O(1)) + O(1) =$
8: $|c| * O(1) = O(|c|)$

---

**iClaves**(**in** $d$: `estr_dicTrie`)→ $res$ : conj(*string*)

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

**iDefinido?**(**in** $d$: `estr_dicTrie`, **in** $c$: `clave`)→ $res$ : $bool$

1: ***Nodo** $actual \leftarrow \&(d.\text{raiz})$
2: **bool** $existe \leftarrow$ true ▷ $\Theta(1)$
3: **for** $i : nat \leftarrow 0$ to **Longitud**($c$) **do** ▷ $\Theta(long(c))$
4: **if** $actual \rightarrow$hijos$[(c[i])] =_{\text{obs}}$ NULL **then**
5: $existe \leftarrow$ false ▷ $\Theta(1)$
6: **end if**
7: **end for**
8: *Complejidad del For :* O(longitud(c))
9: $res \leftarrow existe$ ▷ $\Theta(1)$

Complejidad: $\Theta(long(c))$
Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(long(c))$. Aplicando álgebra de órdenes:
10: $O(1) + O(1) + |c| * (O(1)0) + O(1) =$
11: $2 * O(1) + |c| * O(1) =$
12: $O(1) + |c| * O(1) = O(|c|)$

### 5.3.1. Algoritmos del iterador

**iCrearIt**(**in/out** $d$: `diccTrie`) → $res$ : $itDicctrie$

1: $itc \leftarrow d.claves.CrearIt()$ ▷ $\Theta(1)$
2: $its \leftarrow d.significados.CrearIt()$ ▷ $\Theta(1)$
3: $res.claves \leftarrow itc$
4: $res.significados \leftarrow its$

Complejidad: $\Theta(longitud(k) * tam(d))$

**iHaySiguiente**(**in/out** $it$: `itDiccTrie`) → $res$ : $bool$

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

**iHayAnterior**(**in/out** $it$: `itDiccTrie`) → $res$ : $bool$

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

**iSiguiente**(**in/out** $it$: `itDiccTrie`) → $res$ : $tupla(string, \alpha)$

1: $res.clave \leftarrow it.claves.siguienteClave$ ▷ $\Theta(1)$ (es una referencia)
2: $res.significado \leftarrow it.significados.siguienteSignificado$ ▷ $\Theta(1)$(es una referencia)

**iSiguienteClave**(**in/out** $it$: `itDiccTrie`) → $res$ : $String$

1: $res \leftarrow it.claves.siguiente * .dato$ ▷ $\Theta(1)$(es una referencia)

**iSiguienteSignificado**(**in/out** $it$: `itDiccTrie`) → $res$ : $\alpha$

1: $res \leftarrow it.significados.siguiente * .dato$ ▷ $\Theta(1)$(es una referencia)

**iAnterior**(**in/out** $it$: `itDiccTrie`) → $res$ : $tupla(string, \alpha)$

1: $res.clave \leftarrow it.claves.anteriorClave$ ▷ $\Theta(1)$(es una referencia)
2: $res.significado \leftarrow it.significados.anteriorSignificado$ ▷ $\Theta(1)$(es una referencia)

**iAnteriorClave**(**in**/**out** *it* : `itDiccTrie`) → *res* : *String*

1: *res* ← *it.claves.anterior* ∗ *.dato* Θ(1)(es una referencia)

**iAnteriorSignificado**(**in**/**out** *it* : `itDiccTrie`) → *res* : $\alpha$

1: *res* ← *it.significados.anterior* ∗ *.dato* Θ(1)(es una referencia)

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

1: $it.claves \leftarrow it.claves.siguiente$ $\Theta(1)$(es una referencia)
2: $it.significados \leftarrow it.significados.siguiente$ $\Theta(1)$(es una referencia)

**iRetroceder**(**in**/**out** *it* : `itDiccTrie`)

1: $it.claves \leftarrow it.claves.anterior$ $\Theta(1)$
2: $it.significados \leftarrow it.significados.anterior$ $\Theta(1)$

**TAD** ITERADOR UNIDIRECCIONAL BOOL(BOOL)

**parámetros formales**

**gÃ©neros** bool

**géneros** itUniBool(bool)

**igualdad observacional**

$$(\forall it_1, it_2 : \text{itUniBool(bool)}) \left( it_1 =_{\text{obs}} it_2 \iff \begin{pmatrix} \text{Anteriores}(it_1) =_{\text{obs}} \text{Anteriores}(it_2) \wedge \\ \text{Siguientes}(it_1) =_{\text{obs}} \text{Siguientes}(it_2) \end{pmatrix} \right)$$

**observadores básicos**

Anteriores : itUniBool(bool) $\longrightarrow$ secu(bool)
Siguientes : itUniBool(bool) $\longrightarrow$ secu(bool)

**generadores**

CrearItUniBool : secu(bool) $\times$ secu(bool) $\longrightarrow$ itUniBool(bool)

**otras operaciones**

| | | | |
|---|---|---|---|
| SecuSuby : itUniBool(bool) | $\longrightarrow$ secu(bool) | | |
| HayMas? : itUniBool(bool) | $\longrightarrow$ bool | | |
| Actual : itUniBool(Bool) *it* | $\longrightarrow$ nat | | {HayMas?(*it*)} |
| Avanzar : itUniBool(bool) *it* | $\longrightarrow$ itUniBool(bool) | | {HayMas?(*it*)} |

**axiomas**

| | | |
|---|---|---|
| Anteriores(CrearItMod(*i*, *d*)) | $\equiv$ | *i* |
| Siguientes(CrearItMod(*i*, *d*)) | $\equiv$ | *d* |
| HayMas?(<>) | $\equiv$ | false |
| HayMas?(*it*) | $\equiv$ | **if**(Prim(fin(*it*))==true) **then** true **else** HayMas?(Avanzar(*it*)) |
| Actual(*it*) | $\equiv$ | **if**(Prim(Fin(Siguientes(it)))==true) **then** long(Anteriores(*it*))+1 **else** Actual(Avanzar(Fin(Siguientes(*it*)))) |
| Avanzar(*it*) | $\equiv$ | **if**(Prim(Fin(Siguientes(it)))==true)**then** CrearItUniBool(Anteriores(*it*)•Actual(*it*)), Fin(Siguientes(*it*))**else** Avanzar(Fin(Siguientes(*it*))) |

**Fin TAD**

# 6. Módulo ItJugadores

## 6.1. Operaciones del ItJugadores

CREARIT(**in** $v$: `vectorBool`) $\rightarrow$ $res$ : `itJugadores`
**Pre** $\equiv$ {True}
**Post** $\equiv$ {res.vectorBool $=_{\text{obs}}$ v && res.posicion = 0}
**Complejidad:** O(1)
**Descripción:** Crea un iterador cuyo vectorBool es dado por parámetro y lo sitúa al principio del vector.
**Aliasing:** res no es modificable.

HAYMAS(**in** $itJug$: `itJugadores`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {True}
**Post** $\equiv$ {res $=_{\text{obs}}$ haySiguente?(it)}
**Complejidad:** O(n)
**Descripción:** Devuelve true si y solo si en el iterador todavia quedan elementos válidos para avanzar.

ACTUAL(**in** $itJug$: `itJugadores`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {haySiguente?(it)}
**Post** $\equiv$ {alias(res $=_{\text{obs}}$ Siguiente(it))}
**Complejidad:** O(n)
**Descripción:** Devuelve el siguiente elemento válido a la posicion del iterador.
AVANZAR(**in/out** $itJug$: `itJugadores`)
**Pre** $\equiv$ {it $= it_0 \wedge$ haySiguente?(it)}
**Post** $\equiv$ {it $=_{\text{obs}}$ avanzar(it)}
**Complejidad:** O(n)
**Descripción:** Avanza el iterador a la posicion siguiente.

## 6.2. Representacion de itJugadores

**`itJugadores` se representa con `estr`**

donde `estr` es `tupla`($VectorBool$: $Vector$(*bool*),
, $posicion$: `Nat` )

**Invariante de representación:**

**REP en Castellano:**

1. El campo posicion es mayor que cero y menor a la longitud del vectorBool.

Rep : estr $\longrightarrow$ bool
Rep($e$) $\equiv$ true $\Longleftrightarrow$ (0 $\leqslant$ estr.posicion $\leqslant$ longitud(iter.VectorBool)
**Función de abstracción:**
Abs : $estr$ $\longrightarrow$ itUni($\alpha$) {Rep($estr$)}
Abs($estr$) $\equiv$ b: itUni($\alpha$) | Siguientes(b) = Abs(<estr.VectorBoolIt)

## 6.3. Algoritmos De itJugadores

---

**iCoordenadas**(**in** $m$: `estr_mapa`) $\rightarrow$ $res$ : conj($coord$)

1: res.VectorBool $\leftarrow$ v ▷ O(1)
2: res.posicion $\leftarrow$ 0 ▷ O(1)

---

**iCoordenadas**(**in** $m$ : `estr_mapa`) $\rightarrow$ $res$ : conj($coord$)

1: Nat pos $\leftarrow$ itJug.posicion ▷ O(1)
2: **while** [ **do**$O(1)$]$pos < Longitud(itJug.vectorBool)$ && $not(itJug.vectorBool_{[pos]})$
3: pos++ ▷ O(1)
4: **end while**
5: \\ Complejidad del WHILE: O(n) donde n es la cantidad de elementos del vectorBool.
6: res $\leftarrow$ pos

**iCoordenadas**(**in** $m$ : `estr_mapa`) $\rightarrow$ $res$ : conj($coord$)

1: Nat pos $\leftarrow$ itJug.posicion ▷ O(1)
2: **while** [ **do**$O(1)$]$not(itJug.vectorBool_{[pos]})$
3: pos++ ▷ O(1)
4: **end while**
5: \\ Complejidad del WHILE: O(n) donde n es la cantidad de elementos del vectorBool.
6: res $\leftarrow$ pos ▷ O(1)

**iCoordenadas**(**in** $m$ : `estr_mapa`) $\rightarrow$ $res$ : conj($coord$)

1: Nat pos $\leftarrow$ itJug.posicion + 1 ▷ O(1)
2: **while** [ **do**$O(1)$]$pos < Longitud(itJug.vectorBool_{[pos]})$ && $not(itJug.vectorBool_{[pos]}$
3: pos++ ▷ O(1)
4: **end while**
5: \\ Complejidad del WHILE: O(n) donde n es la cantidad de elementos del vectorBool.
6: itJug.posicion $\leftarrow$ pos ▷ O(1)