

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 2o.

| 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 Tuso | 792/14 | btuso.95@gmail.com |

Reservado para la cdra

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

IMPORTANTE .

- Con respecto a la corrección: Tienen en cuenta que hay correcciones sobre cosas que tienen que corregir y comentarlos. Esos comentarios, no es necesario ni deberían modificar el TP, son puntos de intercambio de opiniones y otras maneras de resolver lo mismo. Pero la solución que hicieron es correcta.

"Para recuperar, hacer las modificaciones sobre las cosas a corregir , correcciones Ver ~~comentarios~~ + importante dentro.

Modelos Juegos

- Agregar Jugador: no hace lo que deberia seguir especificacion y no cumple o está mal calculada la complejidad.
→ ver donde se setea la complejidad del jugador según especificación.
- Conectarse: no cumple la complejidad pedida.
- Desconectarse: no cumple la complejidad pedida.
- Bloquear: no cumple la complejidad pedida.
- Entre estructuras, no queda claro si es el conjunto de jugadores o/ y el vector. Quizás quieran ambos o no. Analizar. Pensar como hacer el it de jugadores, tener en cuenta que no habrá sobreexpulsados. Ver si tienen un rango es posible? como lo implementan? les cuesta la complejidad?
- Fijarse la sintaxis (los parámetros) de las funciones en juego. Cuando implementan la pasan la estructura ~~de juego~~ que esté bien, pero jugador y pokemón no cambian. jugadores es int y pokemón es string.
Si hacer otra cosa tendrían un conflicto con la sintaxis declararon en la interfaz.
Además, van a tener problemas con las complejidades.
(avisarles si se les presenta alguna dificultad con esto!).
- El Heap implementado sobre vector tiene aprender $O(n)$ en lugar de $\log(n)$. Van a tener que usar un árbol binario (punteros) para cumplir la complejidad pedida.
Ver resto de correcciones en TP.

Índice

| | |
|---|-----------|
| 1. Modulo Coordenada | 3 |
| 1.0.1. Representación de Mapa | 4 |
| 1.0.2. Invariante de Representación | 4 |
| 1.0.3. Función de Abstracción | 4 |
| 2. Modulo Mapa | 6 |
| 2.0.4. Representación de Mapa | 6 |
| 2.0.5. Invariante de Representación | 6 |
| 2.0.6. Función de Abstracción | 7 |
| 3. Modulo Juego | 10 |
| 3.0.7. Representación de Juego | 12 |
| 3.0.8. Invariante de Representación | 12 |
| 3.0.9. Función de Abstracción | 13 |
| 3.1. Algoritmos | 13 |
| 4. Modulo Diccionario Matriz($coor, \sigma$) | 20 |
| 4.0.1. Especificacion de las operaciones auxiliares utilizadas en la interfaz | 22 |
| 5. Módulo Cola de mínima prioridad(α) | 25 |
| 5.1. Especificación | 25 |
| 5.2. Interfaz | 26 |
| 5.2.1. Operaciones básicas de Cola de mínima prioridad | 26 |
| 5.3. Representación | 27 |
| 5.3.1. Representación de colaMinPrior | 27 |
| 5.3.2. Invariante de Representación | 27 |
| 5.3.3. Función de Abstracción | 27 |
| 5.4. Algoritmos | 27 |
| 6. Módulo Diccionario String(α) | 31 |
| 6.1. Interfaz | 31 |
| 6.1.1. Operaciones básicas de Diccionario String(α) | 31 |
| 6.1.2. Operaciones Básicas Del Iterador | 32 |
| 6.1.3. Representación de Diccionario String(α) | 34 |
| 6.1.4. Invariante de Representación | 34 |
| 6.1.5. Función de Abstracción | 35 |
| 6.2. Algoritmos | 35 |

1. Modulo Coordenada

Interfaz

usa: NAT, BOOL.

se explica con: COORDENADA.

generos: coor.

CREARCOOR(in x : Nat, **in** y : Nat) \rightarrow res : coor

Pre \equiv {true}

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

Complejidad: $O(1)$

Descripción: Crea una nueva coordenada

LATITUD(in c : coor) \rightarrow res : Nat

Pre \equiv {true}

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

Complejidad: $O(1)$

Descripción: Devuelve la latitud de la coordenada pasada por parametro

LONGITUD(in c : coor) \rightarrow res : Nat

Pre \equiv {true}

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

Complejidad: $O(1)$

Descripción: Devuelve la longitud de la coordenada pasada por parametro

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

Pre \equiv {true}

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

Complejidad: $O(1)$

Descripción: Devuelve la distancia euclidea entre las dos coordenadas

COORDENADAARRIBA(in c : coor) \rightarrow res : coor

Pre \equiv {true}

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de arriba

COORDENADAABAJO(in c : coor) \rightarrow res : coor

Pre \equiv {latitud(c) > 0}

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de abajo

COORDENADALADERECHA(in c : coor) \rightarrow res : coor

Pre \equiv {true}

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de la derecha

COORDENADALAIZQUIERDA(in c : coor) \rightarrow res : coor

Pre \equiv {longitud(c) > 0}

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de la izquierda

Representación

1.0.1. Representación de Mapa

Coordenada se representa con estr

donde estr es tupla($la: \text{Nat}$, $lo: \text{Nat}$)

1.0.2. Invariante de Representación

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

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

1.0.3. Función de Abstracción

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

$\text{Abs}(e) \equiv (\forall c : \text{coor}) e.\text{la} = \text{latitud}(c) \wedge e.\text{lo} = \text{longitud}(c)$

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

Algoritmos

Trabajo Práctico II Algoritmos del modulo

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

1: $res.\text{la} \leftarrow x$

2: $res.\text{lo} \leftarrow y$

$\triangleright \Theta(1)$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

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

1: $res \leftarrow c.\text{la}$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

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

1: $res \leftarrow c.\text{lo}$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

2. Modulo Mapa

Interfaz

usa: NAT, BOOL, COORDENADA, CONJ(α).

se explica con: MAPA.

generos: map.

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

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

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

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

Representación

2.0.4. Representación de Mapa

Mapa se representa con estr

donde estr es tupla(*coordenadas*: ConjLineal, *ancho*: Nat , *secciones*: DiccMat(coor, Nat))

2.0.5. Invariante de Representación

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

Rep : estr \rightarrow bool

$\text{Rep}(e) \equiv \text{true} \iff (\text{e.ancho} = \text{Max}(\text{campo}_2(\text{coordenadas})) \wedge (\forall c : \text{coor}) c \in \text{e.coordena} \Rightarrow_L \text{def?}(c, e.\text{secciones}))$

2.0.6. Función de Abstracción

Abs : estr $e \rightarrow$ mapa

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

{Rep(e)}

Algoritmos

Trabajo Práctico II Algoritmos del modulo

iCrearMapa() $\rightarrow res : \text{Mapa}$

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

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

Complejidad: $\Theta(1)$ $\cancel{\text{ancho} \leftarrow 0}$

$\cancel{\text{secciones} \leftarrow \text{Vacio}()}.$ (o NULL si no queremos crecer).

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

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

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

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

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

3: $m.\text{secciones} \leftarrow \text{CrearArreglo}(largo * ancho)$

$\triangleright \Theta(largo * ancho)$ ✓

4: $res \leftarrow \text{Aregar}(m.\text{coordena}, c)$

No
vacío (largo, ancho).

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

5: $seccion \leftarrow 0$

$\triangleright \Theta(1)$ ✓

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

$\triangleright \Theta(1)$ ✓

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

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

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

$\triangleright \Theta(1)$ ✓

9: $\text{Avanzar}(it)$

$\triangleright \Theta(1)$ ✓

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

$\triangleright \Theta(1)$ ✓

11: $\text{DefinirSeccion}(m, coord, seccion)$

$\triangleright \Theta(1)$ ✓

12: $seccion \leftarrow seccion + 1$

$\triangleright \Theta(1)$ ✓

13: **end if**

$\triangleright \Theta(1)$ ✓

14: **end while**

$\triangleright \Theta(1)$ ✓

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

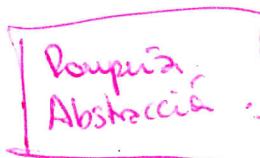
Justificación: $\Theta(\text{Ancho}(m) * \text{Largo}(m))$ es mayor o igual que $\Theta(\text{Cardinal}(m.\text{coordena}))$ y el costo de Agregar un elemento a un conjunto lineal. El While tiene complejidad $\Theta(\text{Cardinal}(m.\text{coordena}))$ dentro, y dentro se llama a una función con la misma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

Secciones es de tipo DicctNat

Usar la interfaz de DicctNat.

Vacio (ancho, largo, ancho).

→ Si usan crearArreglo este mal!
eso es porz exceso y si hubiese implementado DicctNat en un arreglo tanlo co



podrían usarlo pq estén trabajando sobre la estructura que está fuera del módulo DicctNat.

iDefinirSeccion(in/out m : map, in c : coor, in i : nat)

```

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

```

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

$\triangleright \Theta(1)$

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

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

$\triangleright \Theta(1)$

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

$\triangleright \Theta(1)$

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

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

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 $\Theta(4 * \text{Cardinal}(m.coordenadas))$ que se puede simplificar, ya que pertenece a la misma clase. Esta función no es cuadrática, ya que usa el diccionario para chequear que no este recorriendo una posición más de una vez.

↓ comentarios
 En realidad, no es exponencial (sería). → 4^n }
 Tiene una "poda" que corta las recursiones }
 Año 3.11

iCoordenadas(in m : map) → res : itConj(coor)

```

1: res ← CrearIt(m.coordenadas)           ▷ La complejidad es la de crear un iterador a un conjunto lineal  $\Theta(1)$ 
    Complejidad:  $\Theta(1)$ 

```

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

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

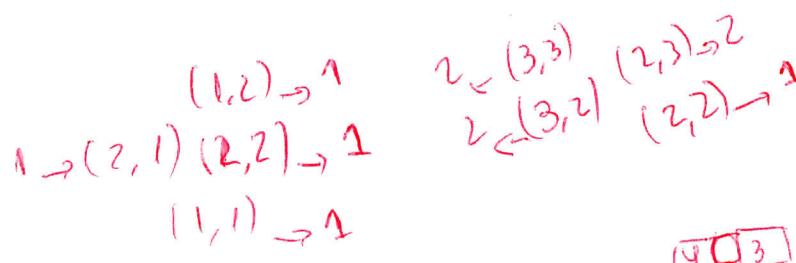
Justificación: La complejidad es la fijarse que un elemento pertenezca al conjunto lineal.

iHayCamino(in c1 : coor, in c2 : coor, in m : map) → res : bool

```

1: res ←  $(Definido?(m.secciones, c1) \wedge Definido?(m.secciones, c2)) \wedge_L (Significado(m.secciones, c1) =$ 
    $Significado(m.secciones, c2))$            ▷  $\Theta(1)$ 
    Complejidad:  $\Theta(1)$ 

```



403

iAncho(in $m : \text{map}$) $\rightarrow res : \text{nat}$

```

1:  $it \leftarrow \text{CrearIt}(m.\text{coordenadas})$ 
2:  $max \leftarrow 0$ 
3: while  $\text{doHaySiguiente}(it)$ 
4:   if ( then  $max < \text{campo}_2(it \rightarrow \text{siguiente})$ )  $\Rightarrow$  Siguiente(it)
5:      $max \leftarrow \text{campo}_2(it \rightarrow \text{siguiente})$ 
6:   end if
7:    $\text{Avanzar}(it)$ 
8: end while

```

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

$\triangleright \Theta(1)$
 $\triangleright \Theta(1)$
 $\triangleright (\#m.\text{coordenadas})$
 $\triangleright \Theta(1)$
 $\triangleright \Theta(1)$
 $\triangleright \Theta(1)$
 $\triangleright \Theta(1)$

iLargo(in $m : \text{map}$) $\rightarrow res : \text{nat}$

```

1:  $it \leftarrow \text{CrearIt}(m.\text{coordenadas})$ 
2:  $max \leftarrow 0$ 
3: while  $\text{doHaySiguiente}(it)$ 
4:   if ( then  $max < \text{campo}_1(it \rightarrow \text{siguiente})$ )  $\Rightarrow$  Siguiente(it)
5:      $max \leftarrow \text{campo}_1(it \rightarrow \text{siguiente})$ 
6:   end if
7:    $\text{Avanzar}(it)$ 
8: end while

```

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

$\triangleright \Theta(1)$
 $\triangleright \Theta(1)$
 $\triangleright (\#m.\text{coordenadas})$
 $\triangleright \Theta(1)$
 $\triangleright \Theta(1)$
 $\triangleright \Theta(1)$
 $\triangleright \Theta(1)$

3. Modulo Juego

Interfaz

usa: MAPA, COORDENADA.

se explica con: JUEGO.

generos: juego.

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

Pre \equiv {true}

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

Complejidad: $\Theta(MUCHO)$

Descripción: Crea el nuevo juego, revisar la complejidad

AGREGARPOKEMON(in/out j : juego, in c : coor, in p : pokemon) \rightarrow res : itPokemon

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

Post \equiv { j =_{obs} 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 : Nat

Pre \equiv { j =_{obs} j_0 }

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

Complejidad: $O(J)$

Descripción: Agrega el jugador en el conjLineal, el iterador que devuelve el agregar se guarda en un vector donde la posicion es el id del jugador que voy a devolver

CONECTARSE(in/out j : juego, in id : Nat, in c : coor)

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

Post \equiv { j =_{obs} 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 : Nat)

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

Post \equiv { j =_{obs} desconectarSe(id, j_0)}

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

Descripción: Desconecta al jugador pasado por parametro

MOVERSE(in/out j : juego, in id : Nat, in c : coor)

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

Post \equiv { j =_{obs} 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 : Mapa

Pre \equiv {true}

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

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

Descripción: Devuelve el mapa del juego

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

Pre \equiv {true}

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

Complejidad: $\Theta(1)$

Descripción: Devuelve un iterador al conjunto de jugadores del juego

ESTACONECTADO(in j : juego, in id : Nat) \rightarrow res : Bool

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

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

Complejidad: $\Theta(1)$

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

POSICION(**in** j : juego, **in** id : Nat) $\rightarrow res : \text{coor}$

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 : Nat) $\rightarrow res : \text{itConj(itDiccString)}$

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) $\rightarrow res : \text{itConj(Jugador)}$

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) $\rightarrow res : \text{itConj(Coor)}$

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) $\rightarrow res : \text{itPokemon}$

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) $\rightarrow res : \text{Nat}$

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) $\rightarrow res : \text{Bool}$

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

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

Complejidad: $\Theta(\text{??})$

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

HAYPOKEMONCERCANO(**in** j : juego, **in** c : Coor) $\rightarrow res : \text{Bool}$

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

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

Complejidad: $\Theta(\text{??})$

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

POSPOKEMONCERCANO(**in** j : juego, **in** c : Coor) $\rightarrow res : \text{Coor}$

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

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

Complejidad: $\Theta(\text{??})$

O(D) x enunciado

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

Si no
cuenta lo veas

no existe itPokemon
↓
Pokemon no es módulo
hay un conflicto de
tipos
↳ tener de usar
Pokemon como String
↓

itColaPrior(jugador)

ENTRENADORESPOSIBLES(in c : coor, in es : conjLineal(jugador), in j : juego) $\rightarrow res : itColaPrior(itJugador)$

Pre $\equiv \{hayPokemonCercano(c, j) \wedge_{\perp} pokemonEnPos(posPokemonCercano(c, j), j).jugadoresEnRango \subseteq jugadoresConectados(c, j)\}$

Post $\equiv \{res =_{obs} entrenadoresPosibles(c, pokemonEnPos(posPokemonCercano(c, j), j).jugadoresEnRango, j)\}$

Complejidad: $\Theta(\text{???})$

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

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

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

✓ ojo con
lo que transforma
el jugador
a jugadores
Nat

Representación

3.0.7. Representación de Juego

Juego se representa con juego

donde juego es tupla($pokemones: diccString(Nat)$, $todosLosPokemons: conjLineal(pokemon)$, $jugadores: conjLineal(jugador)$, $expulsados: conjLineal(jugador)$, $jugadoresPorID: Vector<itConj(jugador), itColaPrior(jugador)>$, $posicionesPokemons: DiccAc(coor, itConjLineal(pokemon))$, $mapa: Mapa$)

esta ok! continuo.
sean bien la complejidad del módulo
vector el agregar - (esto los va a ayudar
a calcular la complejidad de
agregar Jugador).
poke

Jugador se representa con jug

que estructura es DicAc?

donde jug es tupla($id: Nat$, $posicion: Coordenada$, $estaConectado: Bool$, $sanciones: Nat$, $pokeCapturados: ConjLineal(itDiccString(pokemon))$)

Nat?

Rever este dato

Pokemon se representa con poke

Pokemon

donde poke es tupla($tipo: String$, $contador: Nat$, $jugadoresEnRango: diccHeap<Nat, itConjLineal>$, $salvaje: Bool$)

ocupada? se representa con Bool

jugador es string
Pokemon es nat
y luego pueden definir
los tipos

3.0.8. Invariante de Representación

- La suma de todos los significados de pokemones es igual al cardinales de todosLosPokemons.
- La suma de la cantidad de jugadores y expulsados es igual a la longitud del vector jugadoresPorID.

3. Para toda coordenada, si esta definida en posicionesPokemons entonces la coordeanda pertenece al mapa.
 4. La posicion de todo jugador que pertenezca al conjunto jugadores y este conectado pertenece al mapa.
 5. Para todo pokemon que exista en pokemons y sea salvaje, el conjunto de jugadores que esta esperando para atraparlo pertenece al conjunto jugadores.
 6. Todo jugador que pertenezca a jugadores, este conectado y este esperando para atrapar, esta incluido en el conjunto de jugadores en rango del pokemon al que quiere atrapar.
 7. Los conjuntos jugadores y expulsados son disjuntos.
1. Checkear con significado de trie
 2. $\# e.jugadores + \# e.expulsados = \text{long}(e.jugadoresPorID)$
 3. $(\forall c : coor) \text{ def?}(c, e.posicionesPokemons) \Rightarrow_L j.posicion \in e.mapa.coordenadas$
 4. $(\forall j : jug) j \in e.jugadores \wedge j.estConectado \Rightarrow_L j.posicion \in e.mapa.coordenadas$
 5. $(\forall p : poke) (\text{def?}(p, e.pokemons) \wedge p.salvaje) \Rightarrow_L (\forall it : itJug) \text{ HayMas?}(it) \wedge_L \text{Actual}(it) \in p.jugadoresEnRango \Rightarrow_L \text{Actual}(it) \in e.jugadores$
 6. $(\forall j : jug) j \in e.jugadores \wedge j.estConectado \wedge_L \text{estaParaAtrapar}(j) \Rightarrow_L (\forall p : poke) \text{ def?}(p, e.pokemons) \wedge_L j \in p.jugadoresEnRango$
 7. $(\forall j : jug) (j \in e.jugadores \Rightarrow_L j \notin e.expulsados) \vee (j \in e.expulsados \Rightarrow_L j \notin e.jugadores)$

3.0.9. Función de Abstracción

Abs(e): estre - > 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 impliego
 $j.sanciones = sanciones(j, pGo) ((j \text{ pertenece expulsados}(pGo) \wedge j.sanciones \geq 10) \vee (j.pokesCapturados = pokemons(j, pGo) \wedge j.estConectado = estConectado(j, pGo) \wedge j.estConectado \text{ impliego } j.pos = posicion(j, pGo)))$
 (Para todo p : pokemon) p pertenece e.pokemons impliego (Para todo j : Jugador)
 $j \text{ pertenece e.jugadores y luego } p \text{ pertenece pokemons}(j, pGo) \text{ o } [(\text{Para todo } c : coord) c \text{ pertenece e.mapa.coordenadas y luego } p = pokemonEnPos(c, pGo) \wedge \text{cantMovParaCap}(c, pGo) \wedge p.contador]$

3.1. Algoritmos

iCrearJuego(in m : Mapa) → res : juego

Pre ≡ true

Juego : j
 $j.pokemons \leftarrow \text{crearDiccionario}(\text{Vacio}())$
 $j.posicionesPokemon \leftarrow \text{VACIO}()$
 $j.jugadores \leftarrow \text{vacio}()$
 $j.expulsados \leftarrow \text{vacio}()$
 $j.jugadoresPorID \leftarrow \text{vacia}()$
 $j.mapa \leftarrow m$
 $j.pokemonTotales \leftarrow 0$
 $res \leftarrow j$

Complejidad: $O(\text{ancho} * \text{largo})$ ~~seguro~~ (X)

Justificación: la unica operacion que no es O(1) tiene esa complejidad. (???)

↑ todos los Pokemons?
 ↑ de un solo?
 ↑ no ejerce estructura?

▷ O(1)
 ▷ O(1)
 ▷ O(ancho * largo) ??
 ▷ O(1)
 ▷ O(1)

en el TAD es String en el modulo?

iAgregarPokemon(in/out j: juego), in c: coor, in p: pokemon) → res : itPokemon

ItColaPrior(itJugador) it ← entrenadoresPosibles(j, c) $\triangleright O(|1|)$
 while it.HaySiguiente() do $\triangleright O(1)$
 Definir(p.jugadoresEnRango, it.Siguiente().id, it.Siguiente()) $\triangleright O(\log(|entrenadoresPosibles|))$
 end while
 p.salvaje ← TRUE $\triangleright O(|1|)$
 p.contador ← 0 $\triangleright O(|1|)$
 j.pokemonsTotales ← j.pokemonsTotales + 1 $\triangleright O(|1|)$
 if Definido?(pokemones, p.tipo) then $\triangleright O(|1|)$
 ItPokemon poke ← Obtener(j.pokemones, p.tipo) $\triangleright O(|p.tipo|)$
 else $\triangleright O(|p.tipo|)$
 ItPokemon poke ← Definir(j.pokemones, p.tipo, p) $\triangleright O(|p.tipo|)$
 end if $\triangleright O(|p.tipo|)$
 res ← Definir(j.posicionesPokemon, coord, < poke, true >) $\triangleright O(|1|)$

Complejidad: $O(|p.tipo| + |entrenadoresPosibles| * \log(|entrenadoresPosibles|))$ esta mal creo, pero no se que meterle

Justificación: definir, preguntar si esta definido y obtener el pokemon son la longitud del tipo ya que representan una insercion o busqueda en un trie, el ciclo recorre todos los entrenadores posibles, los cuales pertenecen a un conjunto acotado por el rango del pokemon, hay tantos ciclos como entrenadores posibles y por cada uno de ellos hay que definirlo en un heap

Jerarquía y TAD recibe solo juego

iAgregarJugador(in/out j: juego), in c: coor, in jug: Jugador) → res : Nat

jug.pokeCapturados ← Vacio() $\triangleright O(1)$
 jug.posicion ← coor \triangleright coor es tipo y no se especifica coordenadas $\triangleright O(1)$
 jug.estaConectado ← TRUE \triangleright jug se lleva los datos al iniciar $\triangleright O(1)$
 jug.sanciones ← 0 $\triangleright O(1)$
 ItConLinealitConj ← agregarRapido(j.jugadores, jug) $\triangleright O(\text{copy(jug)})$
 ItColaPrioritCola ← NULL \triangleright quien es coor? $\triangleright O(1)$
 if HayPokemonCercano(j, coor) then \triangleright de AgregarJugador de FALSE $\triangleright O(|P|)$
 ItPokemonpoke ← pokemonEnPos(j, posPokemonCercano(j, c)) $\triangleright O(1)$
 poke.contador ← 0 $\triangleright O(1)$
 itCola ← definir(poke.siguiente.jugadoresEnRango, jug.id, jug) $\triangleright O(\text{copy(jug)})$
 end if \triangleright Siguiente → AgregarAtos si es vector \triangleright no ejerce en jugadores? $\triangleright O(\text{copy(jug)})$?
 res ← (agregarRapido(j.jugadoresPorID, coord, < itConj, itCola >)).id \triangleright no ejerce en jugadores? $\triangleright O(\text{copy(jug)})$?
Complejidad: $O(jug) \neq 2$ lo pedido \rightarrow debe dar $O(jug)$ \triangleright cantidad de jugadores.

Justificación: Seria el peor caso, ya que este se da cuando se tiene que armar el heap de jugadoresEnRango de pokemon

no aver las complejidades entre subfunciones → ver bien el tema de las coordenadas

iConectarse(in/out j: juego), in id: Nat, in c: Coor)

Siguiente \triangleright $\pi_2(j.jugadoresPorID[id]).Siguiente.estaConectado \leftarrow \text{true}$ $\triangleright O(1)$
 Siguiente \triangleright $\pi_2(j.jugadoresPorID[id]).Siguiente.posicion \leftarrow coord$ $\triangleright O(1)$
 if HayPokemonCercano(j, coor) then \triangleright (coord es de tipo) \triangleright $\pi_2(j.jugadoresPorID[id]).Siguiente.estaConectado \leftarrow \text{true}$ $\triangleright O(1)$
 ItPokemonpoke ← pokemonEnPos(j, posPokemonCercano(j, c)) $\triangleright O(1)$
 poke.contador ← 0 $\triangleright O(1)$
 definir(poke.siguiente.jugadoresEnRango, id, $\pi_1(j.jugadoresPorID[id])$) \triangleright $\pi_2(j.jugadoresPorID[id]).Siguiente.estaConectado \leftarrow \text{true}$ $\triangleright O(1)$
 end if \triangleright mover? conectar? $\triangleright O(\text{copy(jug)})$
Complejidad: $O(1)$ \rightarrow debe ser $O(\log(Ec))$

→ acá hay algo que no avanza -

primero AgregarJugador no recibe coordenadas, sino que las coordenadas estén al conectarse → la parte de actualizar los pokemones no avanza en j...

iDesconectarse(in/out j : juego, in id : Nat)

Siguientes: $\pi_2(j.jugadoresPorID[id])$.Siguiente.estaNectado $\leftarrow \text{false}$ $\triangleright O(1)$
if HayPokemonCercano(j , coor) **then** $\triangleright O(|p|)$
 ItPokemonpoke \leftarrow pokemonEnPos(j , posPokemonCercano(j , c)) $\triangleright O(1)$
 definir(poke.siguienre.jugadoresEnRango, id, NULL) $\triangleright O(\text{copy}(jug))$
end if

Complejidad: $O(jug)$ → cumple la complejidad pedida.
Justificación:

Moverse(in/out j : juego, in id : Nat, in c : coor)

if debeSancionarse?(Siguiente(jugadoresPorID[id]), c, j) **then** $\triangleright O(|P|)$
 if campo₁(Siguiente(jugadoresPorID[id])).sanciones < 4 **then**
 campo₁(jugadoresPorID[id] → eliminarSiguiente)
 if hayPokemonCercano(j , c) **then**
 (PokemonEnPos → siguiente).jugadoresEnRango.Eliminar(campo₂(jugadoresPorID[id]))
 campo₂(jugadoresPorID[id] → eliminarSiguiente)
 else
 campo₁(jugadoresPorID[id] → siguiente).sanciones \leftarrow campo₁(jugadoresPorID[id] → siguiente).sanciones + 1
 end if
 end if
end if

Complejidad: $O()$? → no cumple la complejidad pedida.
Justificación:

iDebeSancionarse(in e : jugador, in c : coor, in j : juego) \rightarrow res : Mapa

Pre $\equiv e \in \text{Jugadores}(j)$

res $\leftarrow \neg \text{HayCamino}(e.\text{posicion}, c, j.\text{mapa}) \vee \text{DistEuclidea}(e.\text{posicion}, c, \text{mapa}) > 100$ $\triangleright O(\text{copy}(\text{mapa}(j)))$

Complejidad: $O(\text{copy}(\text{mapa}(j)))$ complejidad?
Justificación: Devuelve el mapa del juego por copia.

iMapa(in j : juego) \rightarrow res : Mapa

res $\leftarrow j.\text{mapa}$ $\triangleright O(\text{copy}(\text{mapa}(j)))$

Complejidad: $O(\text{copy}(\text{mapa}(j)))$ ✓
Justificación: Devuelve el mapa del juego por copia.

iJugadores(in j : juego) \rightarrow res : itConj(jugador)

res $\leftarrow \text{CrearIt}(j.\text{jugadores})$ $\triangleright O(1)$

Complejidad: $O(1)$
Justificación: Devuelve el mapa del juego.

iEstaNectado(in j : juego, in id : Nat) \rightarrow res : Bool

res $\leftarrow \text{Siguiente}(j.\text{jugadoresPorID}[id_0]).\text{estaNectado}$ $\triangleright O(1)$

Complejidad: $O(1)$
Justificación: Devuelve si el jugador esta conectado.

iPosicion(in j : juego, in id : Nat) \rightarrow res : coor
 $res \leftarrow Siguiente(j.jugadoresPorID[id]_0).posicion$

 $\triangleright O(1)$ Complejidad: $O(1)$ Justificación: Devuelve si el jugador ~~esta conectado.~~*posicion*

iPokemones(in j : juego, in id : Nat) \rightarrow res : itConj(itDiccString)

 $res \leftarrow CrearIt(Siguiente(j.jugadoresPorID[id]_0).pokeCapturados)$ *el enunciado pide*Complejidad: $O(1)$ Justificación: Devuelve un iterador al conjunto de pokemones atrapados por el jugador. $\triangleright O(1)$

iExpulsados(in j : juego) \rightarrow res : itConj(Jugador)

 $res \leftarrow CrearIt(j.expulsados)$ $\triangleright O(1)$ Complejidad: $O(1)$ Justificación: Devuelve un iterador al conjunto de jugadores expulsados.

iPosConPokemones(in j : juego) \rightarrow res : itConj(Coor)

 $res \leftarrow CrearIt(Coordenadas(j.posicionesPokemons))$ $\triangleright O(1)$ Complejidad: $O(1)$ Justificación: Devuelve las posiciones con pokemones.*Pide devolver el resultado + el resultado*

iPokemonEnPos(in j : juego, in c : coor) \rightarrow res : itConj(Pokemon)

 $res \leftarrow Significado(j.posicionesPokemons, c)$ $\triangleright O(1)$ Complejidad: $O(1)$ Justificación: Devuelve el mapa del juego.*pedir que devuelva el conjunto referente.*

iCantMovimientosParaCaptura(in j : juego, in c : coor) \rightarrow res : Nat

 $res \leftarrow 10 - Siguiente(Significado(j.posicionesPokemons, c)).contador$ $\triangleright O(1)$ Complejidad: $O(1)$ Justificación: Devuelve cuantos movimientos faltan para capturar al pokemon.

iPosPokemonCercano(in j : juego, in c : Coor) $\rightarrow res : coor$

```

i  $\leftarrow 0$   $\triangleright O(1)$ 
latC  $\leftarrow$  Latitud(c)  $\triangleright O(1)$ 
if latC > 2 then  $\triangleright O(1)$ 
    i  $\leftarrow$  latC - 2  $\triangleright O(1)$ 
else  $\triangleright O(1)$ 
    if latC > 1 then  $\triangleright O(1)$ 
        i  $\leftarrow$  latC - 1  $\triangleright O(1)$ 
    else  $\triangleright O(1)$ 
        i  $\leftarrow$  latC  $\downarrow i = 1$   $\triangleright O(1)$ 
    end if  $\triangleright O(1)$ 
end if  $\triangleright O(1)$ 
j  $\leftarrow 0$   $\triangleright O(1)$ 
longC  $\leftarrow$  Longitud(c)  $\triangleright O(1)$ 
if longC > 2 then  $\triangleright O(1)$ 
    j  $\leftarrow$  longC - 2  $\triangleright O(1)$ 
else  $\triangleright O(1)$ 
    if longC > 1 then  $\triangleright O(1)$ 
        j  $\leftarrow$  longC - 1  $\triangleright O(1)$ 
    else  $\triangleright O(1)$ 
        j  $\leftarrow$  longC  $\triangleright O(1)$ 
    end if  $\triangleright O(1)$ 
end if  $\triangleright O(1)$ ;  $i + 2 = 3 \rightarrow$  max vale 3
while i < latC + 2 do  $\triangleright O(1)$  Vale porque estoy recorriendo un conjunto acotado de coordenadas
    while j < longC + 2 do  $\triangleright O(1)$  Vale porque estoy recorriendo un conjunto acotado de coordenadas
        if Definido?(j.posicionesPokemons,  $\langle i, j \rangle$ ) then  $\triangleright O(1)$ 
            res  $\leftarrow \langle i, j \rangle$   $\triangleright O(1)$ 
        end if  $\triangleright O(1)$ 
        j  $\leftarrow$  j + 1  $\triangleright O(1)$ 
    end while  $\triangleright O(1)$ 
    i  $\leftarrow$  i + 1  $\triangleright O(1)$ 
end while  $\triangleright O(1)$ 

```

*No función revisar -
 c = (i,j)
 posición pokem
 (3,1)
 distancia ≤ 1
 2 el espacio
 m 2 divisible*

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 : juego, in c : coor) $\rightarrow res : bool$

```

res  $\leftarrow$  PosExistente(c, j.mapa)  $\wedge \neg(\text{Definido?}(j.\text{posicionesPokemons}, c)) \wedge \neg(\text{HayPokemonCercano}(j, c)) \triangleright O(1)$ 

```

Complejidad: $\Theta(1)$
Justificación:

HayPokemonCercano(in j : juego, in c : coor) $\rightarrow res : bool$

```

res ← false                                ▷ O(1)
i ← 0                                     ▷ O(1)
latC ← Latitud(c)                         ▷ O(1)
if latC > 5 then                          ▷ O(1)
    i ← latC - 5                           ▷ O(1)
else
    if latC > 4 then                      ▷ O(1)
        i ← latC - 4                           ▷ O(1)
    else
        if latC > 3 then                      ▷ O(1)
            i ← latC - 3                           ▷ O(1)
        else
            if latC > 2 then                      ▷ O(1)
                i ← latC - 2                           ▷ O(1)
            else
                if latC > 1 then                      ▷ O(1)
                    i ← latC - 1                           ▷ O(1)
                else
                    i ← latC [ ] i = 1               ▷ O(1)
                end if
            end if
        end if
    end if
end if
j ← 0                                     ▷ O(1)
longC ← Longitud(c)                        ▷ O(1)
if longC > 5 then                          ▷ O(1)
    j ← longC - 5                           ▷ O(1)
else
    if longC > 4 then                      ▷ O(1)
        j ← longC - 4                           ▷ O(1)
    else
        if longC > 3 then                      ▷ O(1)
            j ← longC - 3                           ▷ O(1)
        else
            if longC > 2 then                      ▷ O(1)
                j ← longC - 2                           ▷ O(1)
            else
                if longC > 1 then                      ▷ O(1)
                    j ← longC - 1                           ▷ O(1)
                else
                    j ← longC                         ▷ O(1)
                end if
            end if
        end if
    end if
end if
 $i < 1 + 6 = 6 \Rightarrow$  anivele  $\langle 6, * \rangle$  ▷ no lo encuentra
end if
while i < latC + 5 do                      ▷ O(1) Vale porque estoy recorriendo un conjunto acotado de coordenadas
    while j < longC + 5 do                  ▷ O(1) Vale porque estoy recorriendo un conjunto acotado de coordenadas
        if Definido?(j.posicionesPokemons,  $\langle i, j \rangle$ ) then
            res ← true                         ▷ O(1)
        end if
        j ← j + 1                            ▷ O(1)
    end while
    i ← i + 1                            ▷ O(1)
end while

```

ca (1,1)
ca (6,1)
posicionPokemon
= (6,1)

distancia
 $(6-1) \times (6-1)$
 $\approx (1-1) \times (4-1)$
 $= 5 \times 5 = 25$
pide TAD.

revisar algunos
ejemplos más

Complejidad: $\Theta(1)$

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

resmt

iEntrenadoresPosibles(in c : coor, in es : conjLineal(jugador), in j : juego) $\rightarrow res : conjLineal(itConj)$

```

ePosibles  $\leftarrow$  Vacia()
if Cardinal(es)  $\neq 0$  then
    itE  $\leftarrow$  CrearIt(es)
    while HaySiguiente(itE) do
        posJugador  $\leftarrow$  Posicion(Siguiente(itE))
        if (iHayPokemonCercano(posJugador, j)  $\wedge$ 
            iPosPokeCercano(posJugador, j) == c  $\wedge$ 
            iHayCamino(c, posJugador, Mapa(j))) then
            AgregarRapido(ePosibles, Siguiente(itE))
        end if
        Avanzar(itE)
    end while
end if
res  $\leftarrow$  ePosibles

```

Complejidad: $O(Cardinal(es) * copy(Siguiente(itE)))$

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

esto cambia si jugador es mt

b

6. jugadores por ID [siguiente(itE)]. prior

↓ O(1).

estoy no lo tendría gg! deriva los ids de los jugadores.

$\triangleright O(1)$

$\triangleright O(1)$

$\triangleright O(Cardinal(es))$

$\triangleright O(1)$

es String (no cambien el tipo!).

iIndiceRareza(in j : juego, in p : pokemon) $\rightarrow res : Nat$

```

cuantosP  $\leftarrow$  iCantMismaEspecie(j, p)
res  $\leftarrow$  100 - (100  $\times$  cuantosP / iCantPokemonesTotales(j))

```

Complejidad: $O(|P|)$

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

cuantos cambian el tipo de Pokemon

no coincidir

$\triangleright O(|p.tipo|)$

$\triangleright O(1)$

iCantPokemonesTotales(in j : juego) $\rightarrow res : Nat$

```

res  $\leftarrow$  cardinal(j.todosLosPokemones)

```

Complejidad: $O(1)$

Justificación: Pide el cardinal de un conjunto.

✓ es string

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

```

if Definido?(j.pokemones, p.tipo) then
    res  $\leftarrow$  Obtener(j.pokemones, p.tipo)
else
    res  $\leftarrow$  0
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.

$\triangleright O(|P|)$

$\triangleright O(|P|)$

$\triangleright O(1)$

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 $O(\text{copy}(\sigma))$. 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: $\text{DICCMAT}(Nat, Nat, \sigma)$,

generos: $\text{diccMat}(coor, \sigma)$.

VACIO(in $Nat : 1$ argo, in $Nat : a$ ncho) $\rightarrow res : \text{diccMat}(coor, \sigma)$

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

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

Complejidad: $\Theta(\text{ancho} * \text{largo})$

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

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

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

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

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

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

Aliasing: Hay aliasing, pero no se como explicarlo TODO

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

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

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

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

El (1) (ver algoritmo).

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

SIGNIFICADO(in $d : \text{diccMat}(coor, \sigma)$, in $c : coor$) $\rightarrow res : \sigma$

Pre $\equiv \{\text{enRango}(c_1, c_2, d) \wedge \text{def?}(c_1, c_2, d)\}$

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

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

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

BORRAR(in/out $d : \text{diccMat}(coor, \sigma)$, in $c : coor$)

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

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

Complejidad: $\Theta\left(\sum_{c' \in d.\text{claves}} \text{equal}(c, c')\right)$

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

COORDENADAS(in $d : \text{diccMat}(coor, \sigma)$) $\rightarrow res : \text{itConj}(coor)$

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

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

Complejidad: $\Theta(1)$

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

ANCHO(in $d : \text{diccMat}(coor, \sigma)$) $\rightarrow res : Nat$

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

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

Complejidad: $\Theta(1)$

Descripción: Devuelve el ancho de *d*.

LARGO(in $d : \text{diccMat}(coor, \sigma)$) $\rightarrow res : Nat$

No hay aliasing.

Aliasing se genera cuando quedas apuntando a la misma posición de memoria.

En este caso no hay porque vez es una copia de s cuando agregás en tu estructura.

entonces, cualquier modificación que hagas sobre el valor guardado en el diccionario no va a modificar el valor original.

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

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

Complejidad: $\Theta(1)$

Descripción: Devuelve el largo de d

4.0.1. Especificación de las operaciones auxiliares utilizadas en la interfaz

| TAD DICCMatrix(NAT, NAT, σ) | | <i>comenzó quizás hubiera usado tupla(Nat, Nat) como clave - de hecho claves devuelven un conjunto o los mismos.</i> |
|--|--|---|
| géneros | $\text{diccMat}(\text{Nat}, \text{Nat}, \sigma)$ | <i>Fuerte esta muy bien lo que hicieron! (no hicieron tanto, en cambio, el diccionario con las series pero también que habrá extendido más)</i> |
| exporta | $\text{diccMat}(\text{Nat}, \text{Nat}, \sigma)$, generadores, observadores, borrar, claves | <i>↓ el diccionario con las series pero también que habrá extendido</i> |
| usa | NAT , BOOL , $\text{CONJ}(\text{TUPLA}(\text{NAT}, \text{NAT}))$ | <i>↓ series pero también que habrá extendido</i> |
| igualdad observacional | | <i>↓ es igual que tienen! (no hicieron tanto, en cambio, el diccionario con las series pero también que habrá extendido)</i> |
| $(\forall d, d' : \text{DiccMat}(\text{Nat}, \text{Nat}, \sigma))$ | $d =_{\text{obs}} d' \iff \left(\begin{array}{l} (\text{ancho}(d) =_{\text{obs}} \text{ancho}(d') \wedge \text{largo}(d) =_{\text{obs}} \text{largo}(d')) \\ \wedge_L (\forall x, y : \text{Nat}) (\text{def?}(x, y, d) =_{\text{obs}} \text{def?}(x, y, d')) \\ \wedge_L \text{def?}(x, y, d) \Rightarrow_L \text{obtener}(x, y, d) =_{\text{obs}} \text{obtener}(x, y, d') \end{array} \right)$ | |

observadores básicos

| | | | |
|----------------|--|---------------------------|----------|
| largo | $: \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma)$ | $\rightarrow \text{Nat}$ | <i>✓</i> |
| ancho | $: \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma)$ | $\rightarrow \text{Nat}$ | <i>✓</i> |
| def? | $: \text{Nat} x \times \text{Nat} y \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d$ | $\rightarrow \text{Bool}$ | <i>✓</i> |
| obtener | $: \text{Nat} x \times \text{Nat} y \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d$ | $\rightarrow \sigma$ | <i>✓</i> |

{enRango(x,y,d)}

generadores

| | | | | |
|--------------|----------------|---------------------------------|--|----------|
| vacio | $: \text{Nat}$ | $largo \times \text{Nat} ancho$ | $\rightarrow \text{diccMat}(\text{Nat}, \text{Nat}, \sigma)$ | <i>✓</i> |
|--------------|----------------|---------------------------------|--|----------|

*{largo * ancho > 0}*

| | | | |
|----------------|--|--|----------|
| definir | $: \text{Nat} x \times \text{Nat} y \times \sigma s \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d$ | $\rightarrow \text{diccMat}(\text{Nat}, \text{Nat}, \sigma)$ | <i>✓</i> |
|----------------|--|--|----------|

{enRango(x,y,d)}

otras operaciones

| | | | |
|---------------|--|---|----------|
| borrar | $: \text{Nat} x \times \text{Nat} y \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma) d$ | $\rightarrow \text{diccMat}(\text{Nat}, \text{Nat}, \sigma)$ | <i>✓</i> |
| claves | $: \text{diccMat}(\text{Nat} \times \text{Nat} \times \sigma)$ | $\rightarrow \text{conj}(\text{tupla}(\text{Nat}, \text{Nat}))$ | <i>✓</i> |

otras operaciones (no exportadas)

| | | | |
|----------------|---|---------------------------|----------|
| enRango | $: \text{Nat} \times \text{Nat} \times \text{diccMat}(\text{Nat} \times \text{Nat} \times \text{sinificado})$ | $\rightarrow \text{Bool}$ | <i>✓</i> |
|----------------|---|---------------------------|----------|

axiomas $\forall x, y, m, n : \text{Nat} \forall d : \text{diccMat}(\text{Nat}, \text{Nat}, \sigma) \forall s : \sigma$

$$\text{largo}(\text{vacio}(m, n)) \equiv m$$

$$\text{ancho}(\text{vacio}(m, n)) \equiv n$$

$$\text{def?}(x, y, \text{vacio}(m, n)) \equiv \text{false}$$

$$\text{largo}(\text{definir}(x, y, s, d)) \equiv \text{largo}(d)$$

$$\text{ancho}(\text{definir}(x, y, s, d)) \equiv \text{ancho}(d)$$

$$\text{def?}(x, y, \text{definir}(m, n, s, d)) \equiv (x = m \wedge y = n) \vee \text{def?}(x, y, d)$$

$$\text{obtener}(x, y, \text{definir}(m, n, s, d)) \equiv \text{if } (x = m \wedge y = n) \text{ then } s \text{ else } \text{obtener}(x, y, d) \text{ fi}$$

```

borrar(x,y, definir(m,n,s,d)) ≡ if (x = m ∧ 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) ≡ ∅

claves(definir(x,y,s,d)) ≡ Ag((x,y), claves(d))

enRango(x,y, d) ≡ x < largo(d) ∧ y < ancho(d)

```

COMENTARIO

Este es mejor ejemplo sobre Obs. Basico. Tengo esto correcto, en este caso no modifiqué.

Fin TAD

Representación

Diccionario Matriz se representa con dicc

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

Rep : estr \rightarrow boolRep(e) \equiv true \iff longitud(e.posiciones) = cardinal(e.claves)Abs : estr e \rightarrow diccMat

Abs(e) =_{obs} d: diccMat | ($\forall d : \text{diccMat}$) e.claves = coordenadas(d) \wedge e.ancho = ancho(d) \wedge e.largo = largo(d) \wedge ($\forall c \leftarrow e.\text{claves}$) e.posiciones[c.campo₁ * e.ancho + c.campo₂] = significado(c,d)

Algoritmos

Trabajo Práctico II Algoritmos del modulo

iVacio(in l: Nat, in a: Nat) \rightarrow res : diccMat

- 1: res.largo \leftarrow l $\triangleright \Theta(1)$
- 2: res.ancho \leftarrow a $\triangleright \Theta(1)$
- 3: res.posiciones \leftarrow CrearArreglo(a * l) $\triangleright \Theta(a * l)$
- 4: res.coordenadas \leftarrow Vacio() $\triangleright \Theta(1)$

Complejidad: $\Theta(a * l)$ iDefinir(in/out d: diccMat, in c: coor, in s: σ)

- 1: if $\neg(\text{Definido?}(d, \text{Aplanar}(d, c)))$ then $\triangleright \Theta(\text{copy}(s))$
- 2: AgregarRapido(d.claves, c) $\triangleright \Theta(\text{copy}(s))$
- 3: end if
- 4: d.posiciones[Aplanar(d, c)] \leftarrow s $\triangleright \Theta(\text{copy}(s))$

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

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

iDefinido?(in d : diccMat, in c : coor) \rightarrow res : bool

1: $res \leftarrow Definido?(d.posiciones, Aplanar(d, c)) \wedge_L d.posiciones[Aplanar(d, c)]_1$ ▷ Si no esta definido o esta marcado como borrado, se devuelve que no esta definido $\Theta(1)$
 Complejidad: $\Theta(1)$

Justificacion: $Aplanar$ 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 : diccMat, in c : coor) \rightarrow res : σ

1: $res \leftarrow d.posiciones[Aplanar(d, c)]$ ▷ $\Theta(1)$
 Complejidad: $\Theta(1)$

iBorrar(in/out d : diccMat, in c : coor)

1: $Eliminar(d.claves, c)$ ▷ $\Theta\left(\sum_{c' \in d.claves} equal(c, c')\right)$
 2: $d.posiciones[Aplanar(d, c)] \leftarrow < false, d.posiciones[Aplanar(d, c)]$ ▷ $\Theta(1)$
 Complejidad: $\Theta\left(\sum_{c' \in d.claves} equal(c, c')\right)$

iCoordenadas(in d : diccMat) \rightarrow res : itConj(coor)

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

iAplanar(in d : diccMat, in c : coor) \rightarrow res : nat

1: $res \leftarrow c.campo_1 * d.ancho + c.campo_2$ ▷ $\Theta(1)$
 Complejidad: $\Theta(1)$
Justificacion: Son operaciones matematicas de Nat

iLargo(in d : diccMat) \rightarrow res : nat

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

iAncho(in d : diccMat) \rightarrow res : nat

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

5. Módulo Cola de mínima prioridad(α)

El módulo cola de mínima prioridad consiste en una cola de prioridad de elementos del tipo α cuya prioridad está determinada por un *nat* de forma tal que el elemento que se ingrese con el menor *nat* será el de mayor prioridad.

5.1. Especificación

TAD COLA DE MÍNIMA PRIORIDAD(α)

igualdad observacional

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

parámetros formales

géneros α

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

Relación de orden total estricto¹

géneros $\text{colaMinPrior}(\alpha)$

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

usa BOOL

observadores básicos

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

$\{\neg \text{vacía?}(c)\}$

próximo : $\text{colaMinPrior}(\alpha) c \rightarrow \alpha$

$\{\neg \text{vacía?}(c)\}$

desencolar : $\text{colaMinPrior}(\alpha) c \rightarrow \text{colaMinPrior}(\alpha)$

generadores

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

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

otras operaciones

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

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

vacía?(vacía) $\equiv \text{true}$

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

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

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

Fin TAD

axiomas
de tamaño?

¹Una relación es un orden total estricto cuando se cumple:

Antirreflexividad: $\neg a < a$ para todo $a : \alpha$

Antisimetría: $(a < b \Rightarrow \neg b < a)$ para todo $a, b : \alpha, a \neq b$

Transitividad: $((a < b \wedge b < c) \Rightarrow a < c)$ para todo $a, b, c : \alpha$

Totalidad: $(a < b \vee b < a)$ para todo $a, b : \alpha$

5.2. Interfaz

parámetros formales

géneros α

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

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

5.2.1. Operaciones básicas de Cola de mínima prioridad

$\text{VACÍA}() \rightarrow res : \text{colaMinPrior}(\alpha)$

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

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

Complejidad: $O(1)$

Descripción: Crea una cola de prioridad vacía

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

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

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

Complejidad: $O(1)$

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

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve el próximo elemento a desencolar

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

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

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

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

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

Descripción: Quitar el elemento más prioritario

$\text{ENCOLAR}(\text{in/out } c : \text{colaMinPrior}(\alpha), \text{in } p : \text{nat}, \text{in } a : \alpha) \rightarrow res : \text{itLista}(<\text{nat}, \alpha)$

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

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

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

Descripción: Agrega al elemento a con prioridad p a la cola

Aliasing: Se agrega el elemento por copia

la implementación
no de lopar binico
sino lineal → cambiar
estructura
o punteros.

$\text{ELIMINAR}(\text{in/out } c : \text{colaMinPrior}(\alpha), \text{in } i : \text{nat})$

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

Post $\equiv \{c =_{\text{obs}} \text{Alias}(\text{EsPermutacion}(\text{deColaASecu}(c)))\}$

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

Descripción: Quitar el elemento más prioritario

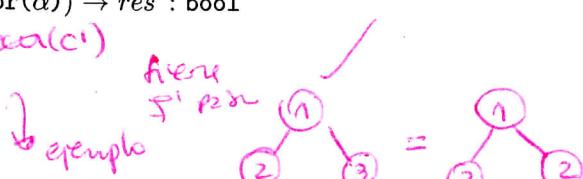
• = •(in $c : \text{colaMinPrior}(\alpha)$, in $c' : \text{colaMinPrior}(\alpha)$) $\rightarrow res : \text{bool}$

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

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

Complejidad: $O(\min(\text{tamaño}(c), \text{tamaño}(c')))$

Descripción: Indica si c es igual c'



5.3. Representación

5.3.1. Representación de colaMinPrior

[enfasis] $\text{colaMinPrior}(\alpha)$ se representa con estr

donde estr es tupla(elementos: Lista(Nodo), proximos: Vector(encolado))

donde nodo es tupla(id: nat, elem: α)

donde encolado es tupla(prior: nat, elemCola: itLista(Nodo))

④ Vector es $O(n)$ peor caso al agregar.
Implementar Heaps sobre arreglos es utilizando
la cantidad de elementos a insertar es acotada o
fija de forma que se crea ya con el tamaño

→ Hay un tipo de tipos → y las inserciones
en encolar usar puntero y esto es log.

④ es lista(Nodo) → continuo?

↓ No tiene complejidad de ser lista(puntero
continuo).

Debería ser List(puntero
(Nodo)).

por que no usamos todo punteros?

5.3.2. Invariante de Representación

(I) Todos ids de cada nodo en elementos son menores que el largo del vector proximos

(II) Si ids de nodos en elementos son diferentes, evaluados en la posición correspondiente de proximos, sus prioridades mantienen la diferencia.

Rep : estr \rightarrow bool

Rep(e) \equiv true \iff

$(\forall i : \text{nat}) (i < \text{longitud}(e.\text{proximos})) \Rightarrow_L$
 $((\forall j : \text{nat}) j \leq i \Rightarrow_L (e.\text{proximos}[j].\text{prioridad} \leq e.\text{proximos}[i].\text{prioridad}) \wedge$
 $((\forall n : \text{nat}) n < \text{longitud}(e.\text{proximos}) \Rightarrow_L e.\text{proximos}[e.\text{elementos}[n].\text{id}].\text{elemCola} \rightarrow \text{siguiente.elem} =_{\text{obs}}$
 $e.\text{elementos}[n].\text{elem})$

④ Tendrán que usar un símbolo

binario (para bajar los punteros
para que quede logarítmico).

5.3.3. Función de Abstracción

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

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

$\text{Abs}(e) =_{\text{obs}} \text{cmp} : \text{colaMinPrior} \mid (\text{vacía?}(\text{cmp}) \Leftrightarrow \text{tamano}(e) =_{\text{obs}} 0) \wedge$
 $\neg \text{vacia?}(\text{cmp}) \Rightarrow_L$
 $(\text{próximo}(\text{cmp}) =_{\text{obs}} \text{próximo}(e) \wedge$
 $\text{desencolar}(\text{cmp}) =_{\text{obs}} \text{desencolar}(e))$

5.4. Algoritmos

iVacia() \rightarrow res : colaMinPrior(α)

1: $res \leftarrow \text{Vacia}(), \text{Vacia}()$

▷ La complejidad es la de crear una lista vacía y un vector vacío $\Theta(1)$

Complejidad: $\Theta(1)$

iVacia?(in c: colaMinPrior(α)) \rightarrow res : Bool

1: $res \leftarrow \text{EsVacia?}(c.\text{elementos})$

▷ La complejidad es la de preguntarle a una lista si es vacía $\Theta(1)$

Complejidad: $\Theta(1)$

iPróximo(in c: colaMinPrior(α)) \rightarrow res : α

1: $res \leftarrow c.\text{proximos}[0].\text{elemCola} \rightarrow \text{siguiente}$ \rightarrow siguiente \triangleright La complejidad acceder por posición a un vector y pedirle al iterador de una lista su siguiente $\Theta(1)$

Complejidad: $\Theta(1)$

iDesencolar(in/out c: colaMinPrior(α))

- 1: *Eliminar(c, 0)* \triangleright La complejidad es la eliminar un elemento en particular (ver iEliminar) $\Theta(\log(\text{tamano}(c)))$
Complejidad: $\Theta(\log(\text{tamano}(c)))$

iEncolar(in/out c: colaMinPrior(α), in prioridad : nat, in a : α) \rightarrow res : iter

- 1: *puntero(Nodo) : nuevo* $\triangleright \Theta(1)$
2: *Nodo $\rightarrow id \leftarrow longitud(c.\text{encolados})$* $\triangleright \Theta(1)$
3: *Nodo $\rightarrow elem \leftarrow a$* $\triangleright \Theta(\text{copy}(\alpha))$
4: *it $\leftarrow AgregarAtras(c.\text{elementos}, Nodo)$* \triangleright La complejidad es la de agregar atras en una lista enlazada $\Theta(\text{copy}(\alpha))$
5: *AgregarAtras(c.proximos, < prioridad, Nodo >)* \triangleright La complejidad es la de agregar atras en un vector es amortizada sumado al copiar $\Theta(f(\text{long}(v)) + \text{copy}(\alpha))$ *→ tienen que tener peek (caso -)*
6: *res $\leftarrow it$* $\triangleright \Theta(1)$
7: **if** *longitud(c.elementos) > 1 then* $\triangleright O(1)$
8: *siftUp(c, Nodo $\rightarrow id)$* $\triangleright O(\log(\text{tamano}(c)))$
9: **end if**
- Complejidad: $\Theta(\log(\text{tamano}(c)) + \text{copy}(\alpha))$ *→ amortizado f peek* *redimensionar el vector* *en peek caso crece más de logarítmico* *(está mal la estructura?)*

iEliminar(in/out c: colaMinPrior(α), in i : nat)

- 1: *swapCola(c, i, longitud(c.proximos) - 1)* $\triangleright \Theta(1)$
2: *c.proximos[longitud(c.proximos) - 1].elemCola $\rightarrow EliminarSiguiente$* $\triangleright \Theta(1)$
3: *c.proximos $\leftarrow Comienzo(c.proximos)$* $\triangleright \Theta(1)$
4: *siftDown(c, i)* $\triangleright O(\log(\text{tamano}(c)))$
- Complejidad: $O(\log(\text{tamano}(c)))$

```

siftDown(in/out c: colaMinPrior( $\alpha$ ), in i: nat) → res : nat
1: nat : posNodo ← i                                ▷  $\Theta(1)$ 
2: nat : posHijo ← i                               ▷  $\Theta(1)$ 
3: if longitud(c.proximos) > 2 then               ▷  $\Theta(1)$ 
4:   bool : swap ← true                            ▷  $\Theta(1)$ 
5:   while posNodo ≤ longitud(c.proximos) - 1 ∧ swap do    ▷ La variable índice (posNodo) siempre avanza en
   forma exponencial  $O(\log(\text{tamano}(c)))$ 
6:     if c.proximos[posNodo].prior < c.proximos[2 * posNodo + 1].prior then      ▷  $\Theta(1)$ 
7:       if c.proximos[posNodo].prior < c.proximos[2 * posNodo + 2].prior then      ▷  $\Theta(1)$ 
8:         swap ← false                           ▷  $\Theta(1)$ 
9:       else
10:         posHijo ← (2 * posNodo + 1)           ▷  $\Theta(1)$ 
11:         swapCola(c, posNodo, posHijo)        ▷  $\Theta(1)$ 
12:       end if
13:     else
14:       if c.proximos[posNodo].prior > c.proximos[2 * posNodo + 2].prior then      ▷  $\Theta(1)$ 
15:         if c.proximos[posNodo].prior > c.proximos[2 * posNodo + 1].prior then      ▷  $\Theta(1)$ 
16:           posHijo ← (2 * posNodo + 2)          ▷  $\Theta(1)$ 
17:           swapCola(c, posNodo, posHijo)        ▷  $\Theta(1)$ 
18:         else
19:           posHijo ← (2 * posNodo + 1)           ▷  $\Theta(1)$ 
20:           swapCola(c, posNodo, posHijo)        ▷  $\Theta(1)$ 
21:         end if
22:       else
23:         posHijo ← (2 * posNodo + 1)           ▷  $\Theta(1)$ 
24:         swapCola(c, posNodo, posHijo)        ▷  $\Theta(1)$ 
25:       end if
26:     end if
27:     posNodo ← posHijo                          ▷  $\Theta(1)$ 
28:   end while
29: else
30:   if longitud(c.proximos) > 1 then          ▷  $\Theta(1)$ 
31:     if c.proximos[posNodo].prior > c.proximos[2 * posNodo + 1].prior then      ▷  $\Theta(1)$ 
32:       posHijo ← (2 * posNodo + 1)          ▷  $\Theta(1)$ 
33:       swapCola(c, posNodo, posHijo)        ▷  $\Theta(1)$ 
34:     end if
35:   end if
36: end if
37: res ← posNodo                                ▷  $\Theta(1)$ 

```

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

No pueden hacer el Heap sobre
el vector —

```
siftUp(in/out c: colaMinPrior( $\alpha$ ) , in i : nat) → res : nat
1: nat : posNodo ← i                                     ▷  $\Theta(1)$ 
2: if longitud(c.proximos) > 1 then                  ▷  $\Theta(1)$ 
3:   bool : swap ← true                                ▷  $\Theta(1)$ 
4:   nat : posPadre                                    ▷  $\Theta(1)$ 
5:   while posNodo ≠ 0 ∧ swap do                      ▷ La variable índice (posNodo) siempre avanza en forma exponencial
   O(log(tamano(c)))                                 ▷  $O(\log(\text{tamano}(c)))$ 
6:     if posNodo mod 2 = 1 then                         ▷  $\Theta(1)$ 
7:       posPadre ← (posNodo - 1)/2                     ▷  $\Theta(1)$ 
8:     else
9:       posPadre ← (posNodo - 2)/2                     ▷  $\Theta(1)$ 
10:    end if
11:    if c.proximos[posPadre].prior > c.proximos[posNodo].prior then ▷  $\Theta(1)$ 
12:      swapCola(c, posNodo, posHijo)                  ▷  $\Theta(1)$ 
13:      posNodo ← posPadre                            ▷  $\Theta(1)$ 
14:    else
15:      swap ← false                                ▷  $\Theta(1)$ 
16:    end if
17:  end while
18: end if
19: res ← posNodo                                      ▷  $\Theta(1)$ 
```

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

```
swapCola(in/out c: colaMinPrior( $\alpha$ ) , in i : nat, in j : nat)
1: encolado : aux ← c.proximos[i]                      ▷  $\Theta(1)$ 
2: c.proximos[i] ← c.proximos[j]                      ▷  $\Theta(1)$ 
3: (c.proximos[i].elemCola → siguiente).id ← i      ▷  $\Theta(1)$ 
4: c.proximos[j] ← aux                                ▷  $\Theta(1)$ 
5: (c.proximos[j].elemCola → siguiente).id ← j      ▷  $\Theta(1)$ 
Complejidad:  $\Theta(1)$ 
```

No pueden hacer el
Heap sobre el vector
(No es $\Theta(1)$ complejidad
logarítmica) —

6. Módulo Diccionario String(α)

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: α .

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

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

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

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

Descripción: función de copia de α .

se explica con: DICCCIONARIO(STRING, α).

géneros: diccString(α), itDiccString(α).

6.1.1. Operaciones básicas de Diccionario String(α)

CREARDICCIONARIO()

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

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

Complejidad: $O(1)$ Justificación: Sólo crea un arreglo de 27 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 $\equiv \{d =_{obs} d_0 \wedge \text{def?}(d, c)\}$

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

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

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

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

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 identicas a ella.

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

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

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

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 $\equiv \{res =_{obs} \text{haySigiente?}(it)\}$

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

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 $\equiv \{\text{haySigiente?}(it)\}$

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

Complejidad: $O(1)$

Descripción: devuelve el significado del elemento siguiente del iterador

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

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

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

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

Complejidad: $O(1)$

Descripción: devuelve el significado del elemento anterior del iterador

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

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

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

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

Complejidad: $O(1)$

Descripción: avanza a la posicion siguiente del iterador.

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

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

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

Complejidad: $O(1)$

Descripción: retrocede a la posicion anterior del iterador.

6.1.3. Representación de Diccionario String(α)

Diccionario String(α) se representa con estr

donde estr es tupla(raiz: arreglo(puntero(Nodo)), listaIterable: lista(puntero(Nodo)))

donde Nodo es tupla(arbolTrie: arreglo(puntero(Nodo)),
 info: α ,
 infoValida: bool,
 infoEnLista: iterador(listaIterable))

6.1.4. Invariante de Representación

- (I) Raiz es la raiz del arbol con invariante de trie y es un arreglo de 27 posiciones.
- (II) Cada uno de los elementos de la lista tiene que ser un puntero a un Nodo del trie.
- (III) Nodo es una tupla que contiene un arreglo de 27 posiciones con un puntero a otro Nodo en cada posicion ,un elemento info que es el alfa que contiene esa clave del arbol, un elemento infoValida y un elemento iterador que es un puntero a un nodo de la lista enlazada.
- (IV) El iterador a la lista enlazada de cada nodo tiene que apuntar al elemento de la lista que apunta al mismo Nodo.
- (V) Cada uno de los nodos de la lista apunta a un nodo del arbol cuyo infoEnLista apunta al mismo nodo de la lista.

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

Rep : estr \rightarrow bool

Rep(e) \equiv true \iff 256
 $\text{longitud}(e.\text{raiz}) == 27 \wedge_L$
 $(\forall i \in [0.. \text{longitud}(e.\text{raiz}))$
 $((\neg e.\text{raiz}[i] == \text{NULL}) \Rightarrow_L \text{nodoValido}(e.\text{raiz}[i])) \wedge (*e.\text{raiz}[i].\text{infoValida} == \text{true} \Rightarrow_L$
 $\text{iteradorValido}(e.\text{raiz}[i])) \wedge$
 $\text{listaValida}(e.\text{listaIterable})$

nodoValido : puntero(Nodo) nodo \rightarrow bool

iteradorValido : puntero(Nodo) nodo \rightarrow bool

nodoValido(nodo) \equiv 256
 $\text{longitud}(*\text{nodo}.arbolTrie) == 27 \wedge_L$
 $(\forall i \in [0.. \text{longitud}(*\text{nodo}.arbolTrie)))$
 $((\neg *\text{nodo}.arbolTrie[i] == \text{NULL}) \Rightarrow_L \text{nodoValido}(*\text{nodo}.arbolTrie[i]))$

iteradorValido(nodo) \equiv
 $\text{PunteroValido}(\text{nodo}) \wedge_L$
 $(\forall i \in [0.. \text{longitud}(*\text{nodo}.arbolTrie)))$
 $((*\text{nodo}.arbolTrie[i].\text{infoValida} == \text{true}) \Rightarrow_L \text{iteradorValido}(*\text{nodo}.arbolTrie[i]))$

PunteroValido(nodo) \equiv

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

listaValida(lista) \equiv

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

6.1.5. Función de Abstracción

Abs : estr $e \rightarrow \text{diccString}(\alpha) \quad \{\text{Rep}(e)\}$

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

6.2. Algoritmos

iCrearDiccionario() $\rightarrow res : \text{estr}$

Pre \equiv true

arreglo(puntero(Nodo)) : $res.raiz \leftarrow \text{CrearArreglo}(27)$

nat : $i \leftarrow 0$

while $i < \text{long}(res.raiz)$ do

$res.raiz[i] \leftarrow \text{NULL}$

end while

$res.listaIterable \leftarrow \text{Vacia}()$

Complejidad: $O(27)$

Justificación: Crea un arreglo de 27 posiciones y lo recorre inicializándolo en NULL. Luego crea una lista vacía.

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

$O(27)$

$\triangleright O(1)$

$\triangleright O(1)$

$\triangleright O(1)$

$\triangleright O(1)$

$\triangleright O(1)$

$\triangleright O(1)$

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

Pre \equiv true

nat : $i \leftarrow 0$

nat : letra $\leftarrow \text{ord}(c[0])$

puntero(Nodo) : arr $\leftarrow d.raiz[letra]$

while $i < \text{longitud}(c) \wedge \neg \text{arr} = \text{NULL}$ do

$i \leftarrow i + 1$

 letra $\leftarrow \text{ord}(c[i])$

 arr $\leftarrow (*arr).\text{arbolTrie}[letra]$

end while

if $i = \text{longitud}(c)$ then

$res \leftarrow (*arr).\text{infoValida}$

else

$res \leftarrow \text{false}$

end if

Complejidad: $O(|c|)$

Justificación: Toma el primer carácter 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.

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

$\triangleright O(1)$

$\triangleright O(1)$

$\triangleright O(1)$

$\triangleright O(|c|)$

$\triangleright O(1)$

```

iDefinir(in/out d: estr, in c: string, in s: α)
Pre ≡ d =obs d0
  nat : i ← 0                                ▷ O(1)
  nat : letra ← ord(c[0])                      ▷ O(1)
  if d.raiz[letra] = NULL then                ↗
    Nodo : nuevo                                ▷ O(1)
    arreglo(puntero(Nodo)) : nuevo.arbolTrie ← CrearArreglo(27) ↗
    nuevo.infoValida ← false                   ↗
    d.raiz[letra] ← puntero(nuevo)             ↗
  end if
  puntero(Nodo) : arr ← d.raiz[letra]          ▷ O(1)
  while i < longitud(c) do                  ↗
    i ← i + 1                                  ▷ O(|c|) ↗
    letra ← ord(c[i])                         ▷ O(1) ↗
    if arr.arbolTrie[letra] = NULL then       ↗
      Nodo : nuevoHijo                         ▷ O(1) ↗
      arreglo(puntero(Nodo)) : nuevoHijo.arbolTrie ← CrearArreglo(27) ↗
      nuevoHijo.infoValida ← false            ↗
      arr.arbolTrie[letra] ← puntero(nuevoHijo) ↗
    end if
    arr ← (*arr).arbolTrie[letra]              ▷ O(1) ↗
  end while
  (*arr).info ← s                            ▷ O(copy(s))
  if ¬(*arr).infoValida then                 ↗
    itLista(puntero(Nodo))it ← AgregarAdelante(d.listaIterable, NULL) ↗
    (*arr).infoValida ← true                  ↗
    (*arr).infoEnLista ← it                  ↗
    siguiente(it) ← puntero(*arr)           ↗
  end if

```

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

Justificación: Itera sobre la cantidad de caracteres del String c y en caso de que algún carácter no esté definido crea un arreglo de 27 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.

Post ≡ d =_{obs} definir(c,s,d₀)

iObtener(in d: estr, in c: string) → res : α

Pre ≡ def?(c,d)

```

  nat : i ← 0                                ▷ O(1)
  nat : letra ← ord(c[0])                      ▷ O(1)
  puntero(Nodo) : arr ← d.raiz[letra]          ↗
  while i < longitud(c) do                  ↗
    i ← i + 1                                  ▷ O(|c|) ↗
    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 carácter 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.

Post ≡ alias(res =_{obs} obtener(c,d))

iEliminacion(in/out d: estr, in c: string)

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

```

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                                     ▷ O(1)
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)  $\wedge$   $\neg$ tieneHermanos(*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                                     ▷ O(1)
if i = longitud(c) then                    ▷ O(1)
    d.raiz[ord(c[0])]  $\leftarrow$  NULL            ▷ O(1)
end if                                         ▷ O(1)
end if                                         ▷ O(1)

```

Complejidad: $O(|c|)$

Justificación: Toma el primer carácter 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.

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

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

Pre \equiv nodo != NULL

```

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                                     ▷ O(1)
res  $\leftarrow$  i < l                                ▷ O(1)

```

Complejidad: $O(1)$

Justificación: Recorre el arreglo de 27 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 27 posiciones entonces es $O(1)$.

Post \equiv res =_{obs} ($\exists i \in [0..longitud(*nodo.arbolTrie)]$ $(*nodo.arbolTrie[i] \neq$ NULL))

tieneHermanosEInfo(in $nodo : \text{puntero}(\text{Nodo})$) $\rightarrow res : \text{bool}$

Pre $\equiv nodo \neq \text{NULL}$

$res \leftarrow \text{tieneHermanos}(nodo) \wedge (*nodo).\text{infoValida} = \text{true}$

$\triangleright O(1)$

Complejidad: $O(1)$

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

Post $\equiv res =_{obs} (\exists i \in [0..\text{longitud}(*nodo.\text{arbolTrie})) (*nodo.\text{arbolTrie}[i] \neq \text{NULL}) \wedge (*nodo).\text{infoValida} = \text{true}$

Hermanos?