

Algoritmos y Estructuras de Datos II

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

Trabajo Práctico II

Diseño

Grupo De TP Algo2

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

Reservado para la cdra

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

Índice

| | |
|--|-----------|
| 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 | 6 |
| 3. Modulo Juego | 8 |
| 3.0.7. Representación de Mapa | 10 |
| 3.0.8. Invariante de Representación | 10 |
| 3.0.9. Función de Abstracción | 11 |
| 4. Modulo Diccionario Acotado(<i>coordenada</i>, σ) | 12 |
| 4.0.10. Especificacion de las operaciones auxiliares utilizadas en la interfaz | 13 |
| 5. Módulo Cola de mínima prioridad(α) | 14 |
| 5.1. Especificación | 14 |
| 5.2. Interfaz | 15 |
| 5.2.1. Operaciones básicas de Cola de mínima prioridad | 15 |
| 5.3. Representación | 16 |
| 5.3.1. Representación de colaMinPrior | 16 |
| 5.3.2. Invariante de Representación | 16 |
| 5.3.3. Función de Abstracción | 16 |
| 5.4. Algoritmos | 16 |
| 6. Módulo Diccionario String(α) | 18 |
| 6.1. Interfaz | 18 |
| 6.1.1. Operaciones básicas de Diccionario String(α) | 18 |
| 6.1.2. Operaciones Básicas Del Iterador | 19 |
| 6.1.3. Representación de Diccionario String(α) | 21 |
| 6.1.4. Invariante de Representación | 21 |
| 6.1.5. Función de Abstracción | 22 |
| 6.2. Algoritmos | 22 |

1. Modulo Coordenada

Interfaz

usa: NAT, BOOL.

se explica con: COORDENADA.

generos: `coor`.

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

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

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

Complejidad: $\Theta(1)$

Descripción: Crea una nueva coordenada

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve la latitud de la coordenada pasada por parametro

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve la longitud de la coordenada pasada por parametro

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve la distancia euclidea entre las dos coordenadas

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de arriba

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de abajo

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de la derecha

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

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

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

Complejidad: $O(1)$

Descripción: Devuelve la coordenada de la izquierda

Representación

1.0.1. Representación de Mapa

Coordenada se representa con `estr`

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

1.0.2. Invariante de Representación

`Rep : estr → bool`

`Rep(e) ≡ true ⇔ true`

1.0.3. Función de Abstracción

`Abs : estr e → coor`

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

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

Algoritmos

Trabajo Práctico II Algoritmos del modulo

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

1: `res.la ← x`

▷ $\Theta(1)$

2: `res.lo ← y`

▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iLatitud(`in c: coor`) → `res: Nat`

1: `res ← c.la`

▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iLongitud(`in c: coor`) → `res: Nat`

1: `res ← c.lo`

▷ $\Theta(1)$

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

```

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

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

```

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

```

Complejidad: $\Theta(1)$

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

```

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

```

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

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

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

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

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

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

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

Representación

2.0.4. Representación de Mapa

Mapa se representa con **estr**

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

2.0.5. Invariante de Representación

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

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

$\text{Rep}(e) \equiv \text{true} \iff (e.\text{ancho} = \text{Max}(\Pi_1(\text{coordenadas})))$

2.0.6. Función de Abstracción

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

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

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

Algoritmos

Trabajo Práctico II Algoritmos del modulo

iCrearMapa() $\rightarrow res : \text{Mapa}$ 1: $res.coordenadas \leftarrow \text{Vacio}()$ \triangleright La complejidad es la de crear el Conjunto Lineal vacio $\Theta(1)$ Complejidad: $\Theta(1)$

iAgregarCoordenada(in/out $m : \text{map}$, in $c : \text{coor}$) $\rightarrow res : \text{itConj}(\text{coor})$ 1: $res \leftarrow \text{Agregar}(m.coordenadas, c)$ \triangleright La complejidad es la de agregar un elemento al conjunto lineal $\Theta(1)$ Complejidad: $\Theta(1)$

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

3. Modulo Juego

Interfaz

usa: MAPA, COORDENADA.

se explica con: JUEGO.

generos: juego.

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

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

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

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

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

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

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

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

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

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

Complejidad: $O(J)$

Descripción: Agrega el jugador en el conjLineal, 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 $\equiv \{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge \neg \text{estaConectado}(id, j_0) \wedge \text{posExistente}(c, \text{mapa}(j_0))\}$

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

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

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

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

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

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

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

Descripción: Desconecta al jugador pasado por parametro

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

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

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

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

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

MAPA(in j : juego) $\rightarrow res$: Mapa

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

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

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

Descripción: Devuelve el mapa del juego

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

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

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

Complejidad: $\Theta(1)$

Descripción: Devuelve un iterador al conjunto de jugadores del juego

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

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

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

Complejidad: $\Theta(1)$

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

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

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

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

Complejidad: $\Theta(1)$

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

POKEMONES(in j : juego, in id : Nat) $\rightarrow res$: itLista(puntero(pokemon))

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

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

Complejidad: $\Theta(1)$

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

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

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

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

Complejidad: $\Theta(1)$

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

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

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

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

Complejidad: $O(1)$

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

POKEMONENPOS(in j : juego, in c : Coor) $\rightarrow res$: 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) $\rightarrow res$: Nat

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

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

Complejidad: $\Theta(1)$

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

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

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

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

Complejidad: $\Theta(???)$

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

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

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

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

Complejidad: $\Theta(???)$

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

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

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

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

Complejidad: $\Theta(???)$

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

ENTRENADORESPOSIBLES(*in j* : juego, *in c* : Coor) $\rightarrow res$: itColaPrior(itJugador)
Pre $\equiv \{hayPokemonCercano(c, j) \wedge_L pokemonEnPos(posPokemonCercano(c, j), j).jugadoresEnRango \subseteq jugadoresConectados(c, j)\}$
Post $\equiv \{res =_{obs} entrenadoresPosibles(c, pokemonEnPos(posPokemonCercano(c, j), j).jugadoresEnRango, j)\}$
Complejidad: $\Theta(???)$
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: $\Theta(???)$
Descripción: Devuelve el indice de rareza del pokemon del juego ingresado

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

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

Representación

3.0.7. Representación de Mapa

Juego se representa con juego

donde juego es tupla(*pokemons*: diccString, *jugadores*: conjLineal(jugador), *expulsados*: conjLineal(jugador), *jugadoresPorID*: Vector(<itConj(jugador), itColaPrior(jugador)>), *posicionesPokemons*: DiccAc(coor, <itDiccString(pokemon), Bool>), *mapa*: Mapa, *pokemonsTotales*: Nat)

Jugador se representa con jug

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

Pokemon se representa con poke

donde poke es tupla(*tipo*: String, *contador*: Nat, *jugadoresEnRango*: diccHeap, *salvaje*: Bool)

3.0.8. Invariante de Representación

1. La suma de las cantidades de cada pokemon es igual a pokemonsTotales.
2. 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 coordenada 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. $(\forall c : \text{coord}) \text{def?}(c, e.posicionesPokemons) \Rightarrow_L j.\text{posicion} \in e.\text{mapa.coordenadas}$
 4. $(\forall j : \text{jug}) j \in e.jugadores \wedge j.\text{estaConectado} \Rightarrow_L j.\text{posicion} \in e.\text{mapa.coordenadas}$
 5. $(\forall p : \text{poke}) (\text{def?}(p, e.pokemons) \wedge p.\text{salvaje}) \Rightarrow_L (\forall it : \text{itJug}) \text{HayMas?}(it) \wedge_L \text{Actual}(it) \in p.jugadoresEnRango \Rightarrow_L \text{Actual}(it) \in e.jugadores$
 6. $(\forall j : \text{jug}) j \in e.jugadores \wedge j.\text{estaConectado} \wedge_L \text{estaParaAtrapar}(j) \Rightarrow_L (\forall p : \text{poke}) \text{def?}(p, e.pokemons) \wedge_L j \in p.jugadoresEnRango$
 7. $(\forall j : \text{jug}) (j \in e.jugadores \Rightarrow_L j \notin e.expulsados) \vee (j \in e.expulsados \Rightarrow_L j \notin e.jugadores)$

3.0.9. Función de Abstracción

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

4. Modulo Diccionario Acotado(*coordenada*, σ)

El modulo Diccionario Acotado provee un diccionario por posiciones en el que se puede definir, borrar, y testear si hay un valor en una posicion en tiempo $O(1)$.

El principal costo de paga al crear la estructura, dado de cuesta tiempo lineal *ancho* por *largo*.

Interfaz

parametros formales

generos *coordenada*, σ

se explica con: DICCACOTADO(Nat , σ),

generos: DiccAc(*coordenada*, σ).

Trabajo Práctico II Operaciones basicas de tabla

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

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

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

Complejidad: $\Theta(ancho * largo)$

Descripción: genera un diccionario vacia.

DEFINIR(**in/out** $t : DiccAc(coordenada, \sigma)$, **in** $c : coordenada$, **in** $s : \sigma$)

Pre $\equiv \{t =_{obs} t_0\}$

Post $\equiv \{t =_{obs} definir(t, c, s)\}$

Complejidad: $\Theta(1)$

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

Aliasing: Hay alising, pero no se como explicarlo TODO

DEFINIDO?(**in** $t : tabla(coordenada, \sigma)$, **in** $c : coordenada$) $\rightarrow res : bool$

Pre $\equiv \{true\}$

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

Complejidad: $\Theta(1)$

Descripción: devuelve **true** si y solo c tiene un valor en la tabla.

SIGNIFICADO(**in** $t : tabla(coordenada, \sigma)$, **in** $c : coordenada$) $\rightarrow res : \sigma$

Pre $\equiv \{def?(t, c)\}$

Post $\equiv \{alias(res =_{obs} Significado(t, c))\}$

Complejidad: $\Theta(1)$

Descripción: devuelve el valor en la posicion c de t .

BORRAR(**in/out** $t : tabla(coordenada, \sigma)$, **in** $c : coordenada$)

Pre $\equiv \{t = t_0 \wedge def?(t, c)\}$

Post $\equiv \{t =_{obs} borrar(t_0, c)\}$

Complejidad: $\Theta(1)$

Descripción: elimina el valor en la posicion c en t .

4.0.10. Especificacion de las operaciones auxiliares utilizadas en la interfaz

TAD DICCACOTADO(NAT,SIGNIFICADO)

géneros $\text{diccAc}(\text{Nat}, \text{Significado})$ **exporta** $\text{diccAc}(\text{Nat}, \text{Significado})$, generadores, observadores, borrar, claves**usa** NAT , BOOL , $\text{CONJ}(\text{NAT})$

,

igualdad observacional

$$(\forall d, d' : \text{Dicc}(\text{Nat}, \sigma)) \left(d =_{\text{obs}} d' \iff \left(\begin{array}{l} (\forall c : \text{Nat}) (\text{enRango}(c, d) =_{\text{obs}} \text{enRango}(c, d') \wedge_L) \\ \text{def?}(c, d) =_{\text{obs}} \text{def?}(c, d') \wedge_L \\ \text{def?}(c, d) \Rightarrow_L \text{obtener}(c, d) =_{\text{obs}} \text{obtener}(c, d') \end{array} \right) \right)$$

observadores básicos $\text{enRango} : \text{Nat} \times \text{diccAc}(\text{Nat} \times \text{significado}) \longrightarrow \text{Bool}$ $\text{def?} : \text{Nat } c \times \text{diccAc}(\text{Nat} \times \text{significado}) d \longrightarrow \text{Bool} \quad \{\text{enRango}(c, d)\}$ $\text{obtener} : \text{Nat } c \times \text{diccAc}(\text{Nat} \times \text{significado}) d \longrightarrow \text{significado} \quad \{\text{enRango}(c, d) \wedge_L \text{def?}(c, d)\}$ **generadores** $\text{vacío} : \text{Nat } r \longrightarrow \text{diccAc}(\text{Nat}, \text{significado}) \quad \{r > 0\}$ $\text{definir} : \text{Nat } c \times \text{significado } s \times \text{diccAc}(\text{Nat} \times \text{significado}) d \longrightarrow \text{diccAc}(\text{Nat}, \text{significado}) \quad \{\text{enRango}(c, d)\}$ **otras operaciones** $\text{borrar} : \text{Nat } c \times \text{diccAc}(\text{Nat} \times \text{significado}) d \longrightarrow \text{diccAc}(\text{Nat}, \text{significado}) \quad \{\text{enRango}(c, d) \wedge_L \text{def?}(c, d)\}$ $\text{claves} : \text{diccAc}(\text{Nat} \times \text{significado}) \longrightarrow \text{conj}(\text{Nat})$ **axiomas** $\forall c, k : \text{Nat} \forall d : \text{diccAc}(\text{Nat}, \text{significado}) \forall s : \text{significado}$ $\text{enRango}(c, \text{vacío}(r)) \equiv c < r$ $\text{def?}(c, \text{vacío}(r)) \equiv \text{false}$ $\text{enRango}(c, \text{definir}(k, s, d)) \equiv \text{enRango}(c, d)$ $\text{def?}(c, \text{definir}(k, s, d)) \equiv c = k \vee \text{def?}(c, d)$ $\text{obtener}(c, \text{definir}(k, s, d)) \equiv \text{if } c = k \text{ then } s \text{ else obtener}(c, d) \text{ fi}$

$$\begin{aligned} \text{borrar}(c, \text{definir}(k, s, d)) \equiv & \text{if } c = k \text{ then} \\ & \text{if } \text{def?}(c, d) \text{ then borrar}(c, d) \text{ else } d \text{ fi} \\ & \text{else} \\ & \text{definir}(k, s, \text{borrar}(c, d)) \\ & \text{fi} \end{aligned}$$
 $\text{claves}(\text{vacío}) \equiv \emptyset$ $\text{claves}(\text{definir}(c, s, d)) \equiv \text{Ag}(c, \text{claves}(d))$ **Fin TAD**

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

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

parámetros formales

géneros α

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

Relación de orden total estricto¹

géneros $\text{colaMinPrior}(\alpha)$

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

usa **BOOL**

observadores básicos

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

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

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

generadores

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

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

otras operaciones

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

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

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

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

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

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

Fin TAD

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: colaMinPrior(α).

5.2.1. Operaciones básicas de Cola de mínima prioridad

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

VACÍA?(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

PRÓXIMO(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

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

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

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

Complejidad: O(log(tamaño(c)))

Descripción: Quita el elemento más prioritario

Aliasing: Se devuelve el elemento por copia

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

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

Post $\equiv \{c =_{\text{obs}} \text{encolar}(p, c_0)\}$

Complejidad: O(log(tamaño(c)))

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

Aliasing: Se agrega el elemento por copia

• = •(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(tamaño(c), tamaño(c')))

Descripción: Indica si c es igual c'

5.3. Representación

5.3.1. Representación de colaMinPrior

`colaMinPrior(α)` se representa con `estr`

donde `estr` es `diccLog(nodoEncolados)`

donde `nodoEncolados` es `tupla(encolados: cola(α), prioridad: nat)`

5.3.2. Invariante de Representación

- (I) Todos los significados del diccionario tienen como clave el valor de *prioridad*
- (II) Todos los significados del diccionario no pueden tener una cola vacía

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

$\text{Rep}(e) \equiv \text{true} \iff (\forall n : \text{nat}) \text{def?}(n, e) \Rightarrow_L ((\text{obtener}(n, e).\text{prioridad} = n) \wedge \neg \text{vacía?}(\text{obtener}(n, e).\text{encolados}))$

5.3.3. Función de Abstracción

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

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

$\text{Abs}(e) =_{\text{obs}} \text{cmp} : \text{colaMinPrior} \mid (\text{vacía?}(\text{cmp}) \Leftrightarrow (\# \text{claves}(e) = 0)) \wedge$
 $\neg \text{vacía?}(\text{cmp}) \Rightarrow_L$
 $((\text{próximo}(\text{cmp}) = \text{próximo}(\text{mínimo}(e).\text{encolados})) \wedge$
 $(\text{desencolar}(\text{cmp}) = \text{desencolar}(\text{mínimo}(e).\text{encolados})))$

5.4. Algoritmos

`iVacía () → res: colaMinPrior(α)`

`res ← CrearDicc()`

$O(1)$

Complejidad : $O(1)$

`iVacía? (in c: colaMinPrior(α)) → res: bool`

`res ← Vacío?(c)`

$O(1)$

Complejidad : $O(1)$

`iPróximo (in/out c: colaMinPrior(α)) → res: α`

`res ← Proximo(Minimo(c).encolados)`

$O(1)$

Complejidad : $O(1)$

iDesencolar (**in/out** c : colaMinPrior(α)) \rightarrow res: α

```

res  $\leftarrow$  Copiar(Proximo(Minimo( $c$ ).encolados))           O(copy( $\alpha$ ))
Desencolar(Minimo( $c$ ).encolados)                          O(log(tamaño( $c$ )))
if EsVacia?(Minimo( $c$ ).encolados) then                     O(1)
    Borrar( $c$ , Minimo( $c$ ).prioridad)                       O(log(tamaño( $c$ )))
end if

```

Complejidad : $O(\log(\text{tamano}(c)) + O(\text{copy}(\alpha)))$

iEncolar (**in/out** c : colaMinPrior(α), **in** p : nat, **in** a : α)

```

if Definido?( $c$ ,  $p$ ) then                                   O(log(tamaño( $c$ )))
    Encolar(Obtener( $c$ ,  $p$ ).encolados,  $a$ )                 O(log(tamaño( $c$ )) + copy( $\alpha$ ))
else
    nodoEncolados nuevoNodoEncolados                     O(1)
    nuevoNodoEncolados.encolados  $\leftarrow$  Vacía()         O(1)
    nuevoNodoEncolados.prioridad  $\leftarrow$   $p$                O(1)
    Encolar(nuevoNodoEncolados.encolados,  $a$ )             O(copy( $a$ ))
    Definir( $c$ ,  $p$ , nuevoNodoEncolados)                   O(log(tamaño( $c$ )) + copy(nodoEncolados))
end if

```

Complejidad : $O(\log(\text{tamano}(c)) + O(\text{copy}(\alpha)))$

$\bullet = \bullet$ (**in** c_0 : colaMinPrior(α), **in** c_1 : colaMinPrior(α)) \rightarrow res: bool

```

res  $\leftarrow$   $c_0 = c_1$                                      O(min(tamano( $c_0$ ), tamaño( $c_1$ )))

```

Complejidad : $O(\min(\text{tamano}(c_0), \text{tamano}(c_1)))$

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: funcion de copia de α .

se explica con: DICCIONARIO(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: $O(|n_m|)$ 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: $O(|n_m| + \text{copy}(s))$ 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: $O(|n_m|)$ Justificación: Debe acceder a la clave c , recorriendo una por una las partes de la clave (caracteres)

Descripción: Devuelve el significado correspondiente a la clave c .

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

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

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

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

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

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

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

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

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

Complejidad: $O(1)$

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

6.1.2. Operaciones Básicas Del Iterador

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

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

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

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

Complejidad: $O(1)$

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

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

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

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

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

Complejidad: $O(1)$

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

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

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

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

Complejidad: $O(1)$

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

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

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

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

Complejidad: $O(1)$

Descripción: devuelve el significado del elemento siguiente del iterador

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

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

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

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

Complejidad: $O(1)$

Descripción: devuelve el significado del elemento anterior del iterador

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

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

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

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

Complejidad: $O(1)$

Descripción: avanza a la posición 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 posición anterior del iterador.

6.1.3. Representación de Diccionario String(α)

Diccionario String(α) se representa con estr

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

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

6.1.4. Invariante de Representación

- (I) Raiz es la raiz del arbol con invariante de trie y es un arreglo de 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.

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

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

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

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

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

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

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

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

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

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

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

6.2. Algoritmos

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

Pre \equiv true

$\text{arreglo}(\text{puntero}(\text{Nodo})) : res.raiz \leftarrow \text{CrearArreglo}(27) \quad \triangleright O(1)$
 $nat : i \leftarrow 0 \quad \triangleright O(1)$
while $i < \text{long}(res.raiz)$ **do** $\triangleright O(1)$
 $\quad res.raiz[i] \leftarrow \text{NULL} \quad \triangleright O(1)$
end while
 $res.listaIterable \leftarrow \text{Vacía}() \quad \triangleright O(1)$

Complejidad: $O(1)$

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

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

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

Pre \equiv true

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

Complejidad: $O(|n_m|)$

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

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

iDefinir(in/out d : **estr**, in c : **string**, in s : α)

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

```

  nat : i  $\leftarrow$  0  $\triangleright O(1)$ 
  nat : letra  $\leftarrow$  ord( $c[0]$ )  $\triangleright O(1)$ 
  if  $d.raiz[letra] = NULL$  then  $\triangleright O(1)$ 
    Nodo : nuevo  $\triangleright O(1)$ 
    arreglo(puntero(Nodo)) : nuevo.arbolTrie  $\leftarrow$  CrearArreglo(27)  $\triangleright O(1)$ 
    nuevo.infoValida  $\leftarrow$  false  $\triangleright O(1)$ 
     $d.raiz[letra] \leftarrow$  puntero(nuevo)  $\triangleright O(1)$ 
  end if
  puntero(Nodo) : arr  $\leftarrow$   $d.raiz[letra]$   $\triangleright O(1)$ 
  while  $i < longitud(c)$  do  $\triangleright O(|n_m|)$ 
    i  $\leftarrow$  i + 1  $\triangleright O(1)$ 
    letra  $\leftarrow$  ord( $c[i]$ )  $\triangleright O(1)$ 
    if  $arr.arbolTrie[letra] = NULL$  then  $\triangleright O(1)$ 
      Nodo : nuevoHijo  $\triangleright O(1)$ 
      arreglo(puntero(Nodo)) : nuevoHijo.arbolTrie  $\leftarrow$  CrearArreglo(27)  $\triangleright O(1)$ 
      nuevoHijo.infoValida  $\leftarrow$  false  $\triangleright O(1)$ 
       $arr.arbolTrie[letra] \leftarrow$  puntero(nuevoHijo)  $\triangleright O(1)$ 
    end if
    arr  $\leftarrow$  ( $*arr$ ).arbolTrie[letra]  $\triangleright O(1)$ 
  end while
  ( $*arr$ ).info  $\leftarrow$  s  $\triangleright O(copy(s))$ 
  if  $\neg(*arr).infoValida$  then  $\triangleright O(1)$ 
    itLista(puntero(Nodo))it  $\leftarrow$  AgregarAdelante( $d.listaIterable, NULL$ )  $\triangleright O(1)$ 
    ( $*arr$ ).infoValida  $\leftarrow$  true  $\triangleright O(1)$ 
    ( $*arr$ ).infoEnLista  $\leftarrow$  it  $\triangleright O(1)$ 
    siguiente(it)  $\leftarrow$  puntero( $*arr$ )  $\triangleright O(1)$ 
  end if

```

Complejidad: $O(|n_m| + copy(s))$
Justificación: Itera sobre la cantidad de caracteres del String c y en caso de que algún caracter no esté definido crea un arreglo de 27 posiciones, por lo que realiza $|n_m|$ operaciones. Luego copia el significado pasado por parámetro en $O(copy(s))$ y finalmente agrega en la lista un puntero al nodo creado.

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

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

```

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

```

Complejidad: $O(|n_m|)$
Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego itera sobre los caracteres restantes hasta el final del String c , por lo que hace $|n_m|$ 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 $\equiv alias(res =_{obs} obtener(c,d))$

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

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

```

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

```

 Complejidad: $O(|n_m|)$

Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego crea una pila en $O(1)$. Recorre el resto de los caracteres del String c y apila cada uno de los Nodos encontrado en la pila ($O(1)$) por lo que en total realiza $|n_m|$ 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 $|n_m|$ 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 $\equiv nodo \neq \text{NULL}$

```

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

```

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

tieneHermanosEInfo(in $nodo : \text{puntero}(\text{Nodo}) \rightarrow res : \text{bool}$ **Pre** $\equiv nodo \neq \text{NULL}$ $res \leftarrow \text{tieneHermanos}(nodo) \wedge (*nodo).infoValida = \text{true} \quad \triangleright O(1)$ Complejidad: $O(1)$ Justificación: Llama a la función `tieneHermanos` que es $O(1)$ y verifica además que el nodo contenga información válida.**Post** $\equiv res =_{obs} (\exists i \in [0..longitud(*nodo.arbolTrie)) (*nodo.arbolTrie[i] \neq \text{NULL})) \wedge (*nodo).infoValida = \text{true}$
