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 Tuso	792/14	btuso.95@gmail.com

Reservado para la cdra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Modulo Coordenada

Interfaz

usa: Nat, Bool.

se explica con: Coordenada.

generos: `coor`.

CrearCoor(**in** x: `Nat`, **in** y: `Nat`) $\to$ res : `coor`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} crearCoor(x, y)\}$
Complejidad: $\Theta(1)$
Descripción: Crea una nueva coordenada

Latitud(**in** c: `coor`) $\to$ res : `Nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} latitud(c)\}$
Complejidad: $O(1)$
Descripción: Devuelve la latitud de la coordenada pasada por parametro

Longitud(**in** c: `coor`) $\to$ res : `Nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} longitud(c)\}$
Complejidad: $O(1)$
Descripción: Devuelve la longitud de la coordenada pasada por parametro

DistEuclidea(**in** $c1$: `coor`, **in** $c2$: `coor`) $\to$ res : `Nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} distEuclidea(c1, c2)\}$
Complejidad: $O(1)$
Descripción: Devuelve la distancia euclidea entre las dos coordenadas

CoordenadaArriba(**in** c: `coor`) $\to$ res : `coor`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} coordenadaArriba(c)\}$
Complejidad: $O(1)$
Descripción: Devuelve la coordenada de arriba

CoordenadaAbajo(**in** c: `coor`) $\to$ res : `coor`
Pre $\equiv$ $\{latitud(c) > 0\}$
Post $\equiv$ $\{res =_{\text{obs}} coordenadaAbajo(c)\}$
Complejidad: $O(1)$
Descripción: Devuelve la coordenada de abajo

CoordenadaALaDerecha(**in** c: `coor`) $\to$ res : `coor`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} coordenadaALaDerecha(c)\}$
Complejidad: $O(1)$
Descripción: Devuelve la coordenada de la derecha

CoordenadaALaIzquierda(**in** c: `coor`) $\to$ res : `coor`
Pre $\equiv$ $\{longitud(c) > 0\}$
Post $\equiv$ $\{res =_{\text{obs}} coordenadaALaIzquierda(c)\}$
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*: `Nat` , *lo*: `Nat`)

1.0.2. Invariante de Representación

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

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

1.0.3. Función de Abstracción

$\text{Abs} : \text{estr } e \longrightarrow \text{coor}$ $\{\text{Rep}(e)\}$

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

Algoritmos

Trabajo Práctico IIAlgoritmos del modulo

---

iCrearCoor(**in** x : `Nat`, **in** y : `Nat`) $\to res$: coor

$res.la \leftarrow x$ $\triangleright\ \Theta(1)$
$res.lo \leftarrow y$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

---

iLatitud(**in** c : `coor`) $\to res$: Nat

$res \leftarrow c.la$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

---

iLongitud(**in** c : `coor`) $\to res$: Nat

$res \leftarrow c.lo$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

---

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

rLa $\leftarrow$ 0 ▷ $\Theta(1)$
rLo $\leftarrow$ 0 ▷ $\Theta(1)$
if c1.la > c2.la **then** ▷ $\Theta(1)$
    rLa $\leftarrow$ ((c1.la - c2.la) x (c1.la - c2.la)) ▷ $\Theta(1)$
else
    rLa $\leftarrow$ ((c2.la - c1.la) x (c2.la - c1.la)) ▷ $\Theta(1)$
end if
if c1.lo > c2.lo **then** ▷ $\Theta(1)$
    rLo $\leftarrow$ ((c1.lo - c2.lo) x (c1.lo - c2.lo)) ▷ $\Theta(1)$
else
    rLo $\leftarrow$ ((c2.lo - c1.lo) x (c2.lo - c1.lo)) ▷ $\Theta(1)$
end if
$res \leftarrow (rLa + rLo)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iCoordenadaArriba(**in** c : `coor`) $\rightarrow$ res : coor

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

Complejidad: $\Theta(1)$

iCoordenadaAbajo(**in** c : `coor`) $\rightarrow$ res : coor

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

Complejidad: $\Theta(1)$

iCoordenadaALaDerecha(**in** c : `coor`) $\rightarrow$ res : coor

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

Complejidad: $\Theta(1)$

iCoordenadaALaIzquierda(**in** c : `coor`) $\rightarrow$ res : coor

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

Complejidad: $\Theta(1)$

2. Modulo Mapa

Interfaz

usa: Nat, Bool, Coordenada, Conj(α).

se explica con: Mapa.

generos: `map`.

CrearMapa() $\to$ *res* : `Mapa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} crearMapa()\}$
Complejidad: $O(1)$
Descripción: Crea un nuevo mapa

AgregarCoordenada(**in/out** m: `map`, **in** c: `coor`) $\to$ *res* : `itConj(coor)`
Pre $\equiv$ $\{m =_{\text{obs}} m_0\}$
Post $\equiv$ $\{m =_{\text{obs}} agregarCoor(c, m_0)\}$
Complejidad: $\Theta\left(\sum_{c' \in coordendas(m)} equal(c, c')\right)$
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: `map`) $\to$ *res* : `itConj(coor)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} coordenadas(m)\}$
Complejidad: $O(1)$
Descripción: Devuelve un iterador al conjunto de coordenadas del mapa

PosExistente(**in** c: `coor`, **in** m: `map`) $\to$ *res* : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} posExistente(c, m)\}$
Complejidad: $\Theta\left(\sum_{c' \in coordendas(m)} equal(c, c')\right)$
Descripción: Devuelve verdadero si la coordenada esta en el conjunto de coordenadas del mapa

HayCamino(**in** $c1$: `coor`, **in** $c2$: `coor`, **in** m: `map`) $\to$ *res* : `Bool`
Pre $\equiv$ $\{c1 \in coordenadas(m) \wedge c2 \in coordenadas(m)\}$
Post $\equiv$ $\{res =_{\text{obs}} hayCamino(c1, c2, m)\}$
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 $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ (e.ancho = Max($campo_2$(coordenadas)) $\wedge$ ($\forall c$: $coor$) c $\in$ e.coordenadas $\Rightarrow_L$ def?(c,e.secciones)

2.0.6. Función de Abstracción

Abs : estr e $\longrightarrow$ mapa {Rep(e)}

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

Algoritmos

Trabajo Práctico IIAlgoritmos del modulo

---

iCrearMapa() $\rightarrow$ res : Mapa

$res.coordenadas \leftarrow Vacio()$ ▷ La complejidad es la de crear el Conjunto Lineal vacio $\Theta(1)$

Complejidad: $\Theta(1)$

---

iAgregarCoordenada(**in/out** m: `map`, **in** c: `coor`) $\rightarrow$ res : itConj(coor)

$largo \leftarrow Largo(m)$ ▷ $\Theta(Cardinal(m.coordenadas))$
$ancho \leftarrow Ancho(m)$ ▷ $\Theta(Cardinal(m.coordenadas))$
$m.secciones \leftarrow CrearArreglo(largo * ancho)$ ▷ $\Theta(largo * ancho)$
$res \leftarrow Agregar(m.coordenadas, c)$ ▷ $\Theta\left(\sum_{c' \in coordendas(m)} equal(c, c')\right)$
$seccion \leftarrow 0$ ▷ $\Theta(1)$
$itCoor \leftarrow CrearIt(m.coordenadas)$ ▷ $\Theta(1)$
while $HaySiguiente(itCoor)$ **do** ▷ $\Theta(Cardinal(m.coordenadas))$
    $coord \leftarrow Siguiente(itCoor)$ ▷ $\Theta(1)$
    $Avanzar(it)$ ▷ $\Theta(1)$
    if $\neg(Definido?(m.secciones, coord))$ **then** ▷ $\Theta(1)$
        $DefinirSeccion(m, coord, seccion)$ ▷ $\Theta(Cardinal(m.coordenadas))$
        $seccion \leftarrow seccion + 1$ ▷ $\Theta(1)$
    end if
end while

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

---

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

if $\neg(Definido?(m.secciones, c)) \wedge PosExistente(c, m)$ **then** $\triangleright\ \Theta\left(\sum_{c' \in coordendas(m)} equal(c, c')\right)$
    $Definir(m.secciones, c, i)$ $\triangleright\ \Theta(1)$
    $DefinirSeccion(m, CoordenadaArriba(c), i)$ $\triangleright\ \Theta(Cardinal(m.coordenadas))$
    $DefinirSeccion(m, CoordenadaALaDerecha(c), i)$ $\triangleright\ \Theta(Cardinal(m.coordenadas))$
    if $Latitud(c) > 0$ **then** $\triangleright\ \Theta(1)$
        $DefinirSeccion(m, CoordenadaAbajo(c), i)$ $\triangleright\ \Theta(Cardinal(m.coordenadas))$
    end if
    if $Longitud(c) > 0$ **then** $\triangleright\ \Theta(1)$
        $DefinirSeccion(m, CoordenadaALaIzquierda(c), i)$ $\triangleright\ \Theta(Cardinal(m.coordenadas))$
    end if
end if

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

---

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

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

Complejidad: $\Theta(1)$

---

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

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

Complejidad: $\Theta\left(\sum_{c' \in coordendas(m)} equal(c, c')\right)$
Justificación: La complejidad es la fijarse que un elemento pertenezca al conjunto lineal.

---

iHayCamino(**in** $c1$: `coor`, **in** $c2$: `coor`, **in** m : `map`) $\rightarrow$ res : *bool*

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

Complejidad: $\Theta(1)$

iAncho(**in** m : `map`) $\rightarrow$ res : nat

$it \leftarrow CrearIt(m.coordenadas)$ $\triangleright\ \Theta(1)$
$max \leftarrow 0$ $\triangleright\ \Theta(1)$
while HaySiguiente(it) **do** $\triangleright\ (\sharp m.coordenadas)$
  if $max < campo_2(it \rightarrow siguiente)$ **then** $\triangleright\ \Theta(1)$
    $max \leftarrow campo_2(it \rightarrow siguiente)$ $\triangleright\ \Theta(1)$
  end if
  $Avanzar(it)$ $\triangleright\ \Theta(1)$
end while

Complejidad: $\Theta(\sharp m.coordenadas)$

iLargo(**in** m : `map`) $\rightarrow$ res : Nat

$it \leftarrow CrearIt(m.coordenadas)$ $\triangleright\ \Theta(1)$
$max \leftarrow 0$ $\triangleright\ \Theta(1)$
while $HaySiguiente(it)$ **do** $\triangleright\ (\sharp m.coordenadas)$
  if $max < campo_1(it \rightarrow siguiente)$ **then** $\triangleright\ \Theta(1)$
    $max \leftarrow campo_1(it \rightarrow siguiente)$ $\triangleright\ \Theta(1)$
  end if
  $Avanzar(it)$ $\triangleright\ \Theta(1)$
end while

Complejidad: $\Theta(\sharp m.coordenadas)$

3. Modulo Juego

Interfaz

usa: Mapa, Coordenada.

se explica con: Juego.

generos: `juego`.

CrearJuego(**in** m: `mapa`) $\rightarrow$ res : `juego`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{crearJuego}(m_0) \wedge \text{mapa}(res) =_{\text{obs}} m_0\}$
Complejidad: O((largo(m) x ancho(m)) + copy(m))
Descripción: Crea el nuevo juego

AgregarPokemon(**in/out** j: `juego`, **in** c: `coor`, **in** p: `pokemon`) $\rightarrow$ res : `itConj`
Pre $\equiv$ $\{j =_{\text{obs}} j_0 \wedge puedoAgregarPokemon(c, j_0)\}$
Post $\equiv$ $\{j =_{\text{obs}} agregarPokemon(p, c, j_0)\}$
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 $\equiv$ $\{j =_{\text{obs}} j_0\}$
Post $\equiv$ $\{j =_{\text{obs}} agregarJugador(j_0) \wedge res = \#jugadores(j_0) + \#expulsados(j_0)\}$
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 $\equiv$ $\{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_{\text{L}} \neg estaConectado(id, j_0) \wedge posExistente(c, mapa(j_0))\}$
Post $\equiv$ $\{j =_{\text{obs}} conectarse(id, c, j_0)\}$
Complejidad: $O(log(EC))$
Descripción: Conecta al jugador pasado por parametro en la coordenada indicada

Desconectarse(**in/out** j: `juego`, **in** id: `Nat`)
Pre $\equiv$ $\{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_{\text{L}} estaConectado(id, j_0)\}$
Post $\equiv$ $\{j =_{\text{obs}} desconectarse(id, j_0)\}$
Complejidad: $O(log(EC))$
Descripción: Desconecta al jugador pasado por parametro

Moverse(**in/out** j: `juego`, **in** id: `Nat`, **in** c: `coor`)
Pre $\equiv$ $\{j =_{\text{obs}} j_0 \wedge id \in jugadores(j_0) \wedge_{\text{L}} estaConectado(id, j_0) \wedge posExistente(c, mapa(j_0))\}$
Post $\equiv$ $\{j =_{\text{obs}} moverse(c, id, j_0)\}$
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 $\equiv$ $\{res =_{\text{obs}} mapa(j)\}$
Complejidad: $O(copy(mapa(j)))$
Descripción: Devuelve el mapa del juego

Jugadores(**in** j: `juego`) $\rightarrow$ res : `itConj(Jugador)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} jugadores(j)\}$
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 $\in$ jugadores(j)}
Post $\equiv$ $\{res =_{\text{obs}} estaConetado(id, j)\}$
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 $\in$ jugadores(j) $\wedge_{\text{L}}$ estaConectado(id,j)}
Post $\equiv$ $\{res =_{\text{obs}} posicion(id, j)\}$
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 : `itConj(itDiccString)`
Pre $\equiv$ {id $\in$ jugadores(j)}
Post $\equiv$ $\{res =_{\text{obs}} pokemons(id, j)\}$
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 $\equiv$ {True}
Post $\equiv$ $\{res =_{\text{obs}} expulsados(j)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve un iterador al conjunto de jugadores expulsados del juego

POSCONPOKEMONES(**in** j : `juego`) $\rightarrow$ res : `itConj(Coor)`
Pre $\equiv$ {True}
Post $\equiv$ $\{res =_{\text{obs}} posConPokemons(j)\}$
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 $\in$ posConPokemons(j)}
Post $\equiv$ $\{res =_{\text{obs}} pokemonEnPos(c, j)\}$
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 $\in$ posConPokemons(j)}
Post $\equiv$ $\{res =_{\text{obs}} cantMovimientosParaCaptura(c, j)\}$
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 $\equiv$ {True}
Post $\equiv$ $\{res =_{\text{obs}} puedoAgregarPokemon(c, j)\}$
Complejidad: $\Theta\left(\sum_{c' \in coordendas(mapa(j))} equal(c, c')\right)$
Descripción: Devuelve si la coordenada ingresada es valida para agregar un pokemon en ella

HAYPOKEMONCERCANO(**in** j : `juego`, **in** c : `Coor`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {True}
Post $\equiv$ $\{res =_{\text{obs}} hayPokemonCercano(c, j)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve si la coordenada ingresada pertenece al rango de un pokemon salvaje

POSPOKEMONCERCANO(**in** j : `juego`, **in** c : `Coor`) $\rightarrow$ res : `Coor`
Pre $\equiv$ $\{hayPokemonCercano(c, j)\}$
Post $\equiv$ $\{res =_{\text{obs}} posPokemonCercano(c, j)\}$
Complejidad: $\Theta(1)$

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

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

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

Representación

3.0.7. Representación de Juego

Juego se representa con `estr`

donde `estr` es `tupla`(*pokemones*: `diccString(String, ListaPorTipo(itConj(infoPokemon)))`, *todosLosPokemones*: `conjLineal(infoPokemon)` , *jugadores*: `conjLineal(infoJugador)` , *expulsados*: `conjLineal(Nat)` , *jugadoresPorID*: `Vector(<itConj(infoJugador), itColaPrior(jugador)>)` , *posicionesPokemons*: `DiccMat(coor, itConj(infoPokemon))` , *posicionesJugadores*: `DiccMat(coor, conjLineal(Nat))` , *mapa*: `Mapa`)

donde `infoJugador` es `tupla`(*id*: `Nat` , *estaConectado*: `Bool` , *sanciones*: `Nat` , *pokeCapturados*: `conjLineal(itConj(infoPokemon))`)

donde `infoPokemon` es `tupla`(*contador*: `Nat` , *jugadoresEnRango*: `colaPrior<Nat, itConj(jugadores)>` , *salvaje*: `Bool`)

3.0.8. Invariante de Representación

1. La suma de todos los significados de pokemones es igual al cardinales de todosLosPokemones.
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 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 $=$ long(e.jugadoresPorID)

3. $(\forall c : coor)$ def?(c, e.posicionesPokemons) $\Rightarrow_L$ j.posicion $\in$ e.mapa.coordenadas

4. $(\forall j : jug)$ j $\in$ e.jugadores $\wedge$ j.estaConectado $\Rightarrow_L$ j.posicion $\in$ e.mapa.coordenadas

5. $(\forall p : poke)$ (def?(p, e.pokemones) $\wedge$ p.salvaje) $\Rightarrow_L$ $(\forall it : itJug)$ HayMas?(it) $\wedge_L$ Actual(it) $\in$ p.jugadoresEnRango $\Rightarrow_L$ Actual(it) $\in$ e.jugadores

6. $(\forall j : jug)$ j $\in$ e.jugadores $\wedge$ j.estaConectado $\wedge_L$ estaParaAtrapar(j) $\Rightarrow_L$ $(\forall p : poke)$ def?(p, e.pokemones) $\wedge_L$ 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) yluego
(Para todo j : jugador) j pertenece e.jugadores impluego
j.sanciones = sanciones(j, pGo) ((j pertenece expulsados(pGo) y j.sanciones >= 10)
oluego (j.pokesCapturados = pokemones(j,pGo) y j.estaConectado = estaConectad(j,pGo)
y j.estaConectado impluego j.pos = posicion(j,pGo))) y
(Para todo p : pokemon) p pertenece c.pokemones impluego (Para todo j : Jugador)
j pertenece e.jugadores yluego p pertenece pokemones(j,pGo) o [(Para todo c : coord)
c pertenece e.mapa.coordenadas yluego p = pokemonEnPos(c,pGo) y cantMovParaCap(c,pGo)
p.contador]

3.1. Algoritmos

iCrearJuego(in m : `Mapa`)) $\rightarrow$ res : $Juego$

$Juego : j$	$\triangleright$ $O(1)$
j.pokemones $\leftarrow$ CrearDiccionario()	$\triangleright$ $O(1)$
j.todosLosPokemones $\leftarrow$ Vacio()	$\triangleright$ $O(1)$
j.jugadores $\leftarrow$ Vacio()	$\triangleright$ $O(1)$
j.expulsados $\leftarrow$ Vacio()	$\triangleright$ $O(1)$
j.jugadoresPorID $\leftarrow$ Vacia()	$\triangleright$ $O(1)$
j.posicionesPokemons $\leftarrow$ Vacio(largo(m), ancho(m))	$\triangleright$ O(largo(m) x ancho(m))
j.mapa $\leftarrow$ m	$\triangleright$ $O(copy(m))$
$res \leftarrow j$	$\triangleright$ $O(1)$

Complejidad: O((largo(m) x ancho(m)) + copy(m))
Justificación:

iAgregarPokemon(in/out j : `juego`)**, in** c : `coor`)**, in** p : `pokemon`)) $\rightarrow$ $res : itPokemon$

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

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

---

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

id ← Cardinal(j.jugadores) + Cardinal(j.expulsados) ▷ O(1)
infoJ ← <id, false, 0, Vacio()> ▷ O(1) El jugador es creado vacio, sin sanciones y sin pokemones atrapados
itJ ← AgregarRapido(j.jugadores, infoJ) ▷ O(copy(infoJ))
AgregarAtras(j.jugadoresPorID, <itJ, NULL>) ▷ O(J) Donde J es la cantidad total de jugadores que fueron agregados al juego
res ← id ▷ O(1)

Complejidad: $O(J)$
Justificación: O(copy(jug)) es igual a O(1) ya que solamente es copiar Nat, Bool y un conjunto vacio.

---

iConectarse(in/out j : `juego`**, in** e : `Nat`**, in** c : `Coor`)

tupJug ← j.jugadoresPorId[e] ▷ $O(1)$
itJug ← tupJug[1] ▷ $O(1)$
jug ← Siguiente(itJug) ▷ $O(1)$
jug.estaConectado ← true ▷ $O(1)$
jug.posicion ← c ▷ $O(1)$
if HayPokemonCercano(j, c) **then** ▷ $O(1)$
  p ← Siguiente(Significado(j.posicionesPokemons, PosPokemonCercano(j, c))) ▷ $O(1)$
  tupJug[2] ← Encolar(p.jugadoresEnRango, Cardinal(jug.pokeCapturados), itJug) ▷ $O(log(EC))$
  p.contador ← 0 ▷ $O(1)$
end if

Complejidad: $O(log(EC))$
Justificación: EC es la maxima cantidad de jugadores esperando para atrapar un pokemon. En el peor caso, el heap al que entra el jugador es el que mas jugadores esperando tiene.

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

tupJug ← j.jugadoresPorId[e] ▷ $O(1)$
if HayPokemonCercano(j, c) **then** ▷ $O(1)$
tupJug[2] ← EliminarSiguiente(tupJug[2]) ▷ $O(log(EC))$
end if
Siguiente(tupJug[1]).estaConectado ← false ▷ $O(1)$

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

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

if debeSancionarse?(Siguiente(jugadoresPorID[id]), c, j) **then** ▷ $O(|P|)$
if $campo_1(Siguiente(jugadoresPorID[id])).sanciones < 4$ **then**
$campo_1(jugadoresPorID[id]-> eliminarSiguiente)$
if hayPokemonCercano(j, c) **then**
$(PokemonEnPos \rightarrow siguiente).jugadoresEnRango.Eliminar(campo_2(jugadoresPorID[id])$
$campo_2(jugadoresPorID[id]-> eliminarSiguiente)$
else
$campo_1(jugadoresPorID[id]-> siguiente).sanciones \leftarrow campo_1(jugadoresPorID[id]-> siguiente).sanciones + 1$
end if
end if
end if

Complejidad: $O()$
Justificación:

---

iDebeSancionarse(**in** e: jugador, **in** c: coor, **in** j: juego) → res : $Bool$
Pre ≡ e ∈ Jugadores(j)

$res \leftarrow \neg HayCamino(e.posicion, c, j.mapa) \vee DistEuclidea(e.posicion, c, mapa) > 100$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Checkea si el jugador hizo un movimiento invalido.

---

iMapa(**in** j: juego) → res : $Mapa$

$res \leftarrow j.mapa$ ▷ $O(copy(mapa(j)))$

Complejidad: $O(copy(mapa(j)))$
Justificación: Devuelve el mapa del juego por copia.

---

iJugadores(**in** j: juego) → res : $itConj(jugador)$

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

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

---

iEstaConectado(**in** j: juego, **in** id: Nat) → res : Bool

$res \leftarrow Siguiente(j.jugadoresPorID[id]_0).estaConectado$ ▷ $O(1)$

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.

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

$res \leftarrow CrearIt(Siguiente(j.jugadoresPorID[id]_0).pokeCapturados)$ $\triangleright$ $O(1)$

Complejidad: $O(1)$
Justificación: Devuelve un iterador al conjunto de pokemones atrapados por el jugador.

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.

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.

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`) $\to$ *res* : *coor*

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

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

---

iPuedoAgregarPokemon(**in** j : `juego`, **in** c : `coor`) $\to$ *res* : *bool*

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

Complejidad: $[\Theta\left(\sum_{c' \in coordendas(mapa(j))} equal(c, c')\right)]$
Justificación: Tiene que ver si la posicion existe en el mapa, las demas operaciones son O(1)

HayPokemonCercano(**in** j : `juego`, **in** c : `coor`) $\to$ *res* : *bool*

res ← false ▷ $O(1)$
latC ← Latitud(c) ▷ $O(1)$
i ← DamePos(latC, 5) ▷ $O(1)$
longC ← Longitud(c) ▷ $O(1)$
j ← DamePos(longC, 5) ▷ $O(1)$
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, <i, j>) ∧ DistEuclidea(c, <i, j>) ≤ 25 **then** ▷ $O(1)$
      res ← true ▷ $O(1)$
    end if
    j ← j + 1 ▷ $O(1)$
  end while
  i ← i + 1 ▷ $O(1)$
end while

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

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

ePosibles ← Vacia() ▷ $O(1)$ Crea un conjunto de iteradores vacio
if Cardinal(es) ! = 0 **then** ▷ $O(1)$
  itE ← CrearIt(es) ▷ $O(1)$
  while HaySiguiente(itE) **do** ▷ $O(Cardinal(es))$ Es la cantidad de jugadores que haya en el conjunto es
    posJugador ← Siguiente(itE).posicion ▷ $O(1)$
    if (iHayPokemonCercano(posJugador, j) $\wedge_L$ ▷ $O(1)$
      iPosPokemonCercano(posJugador, j) == c $\wedge$ ▷ $O(1)$
      iHayCamino(c, posJugador, Mapa(j))) **then** ▷ $O(1)$
      AgregarRapido(ePosibles, Siguiente(itE)) ▷ $O(1)$ Copiar un iterador es O(1)
    end if
    Avanzar(itE) ▷ $O(1)$
  end while
end if
$res \leftarrow ePosibles$ ▷ $O(1)$

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

---

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

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

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

---

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

$res \leftarrow cardinal(j.todosLosPokemones)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Pide el cardinal de un conjunto.

---

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

if Definido?(j.pokemones, p.tipo) **then** ▷ $O(|P|)$
  $res \leftarrow Obtener(j.pokemones, p.tipo)$ ▷ $O(|P|)$
else
  $res \leftarrow 0$ ▷ $O(1)$
end if

Complejidad: $O(|P|)$
Justificación: En peor caso, el pokemon que se busca es el de nombre mas largo o no esta en el diccionario.

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

```
i ← p                                  ▷ O(1)
fin ← false                            ▷ O(1)
res ← i                                ▷ O(1)
while i ≠ 1 ∧¬ fin do                  ▷ O(1)
    i ← i - 1                          ▷ O(1)
    if i == p - step then              ▷ O(1)
        fin ← true                     ▷ O(1)
    end if
end while
```

Complejidad: $O(1)$
Justificación: Recorre un rango acotado.

4. Modulo Diccionario Matriz($coor$, σ)

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

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

Interfaz

parametros formales
generos $coor, \sigma$
se explica con: DiccMat(Nat, Nat, σ),

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

Vacio(**in** Nat: l argo, **in** Nat: a ncho) $\rightarrow res$: `diccMat`($coor, \sigma$)
Pre ≡ {largo $*$ ancho > 0}
Post ≡ {$res =_{\text{obs}}$ vacio(largo, ancho)}
Complejidad: $\Theta(ancho*largo)$
Descripción: Genera un diccionario vacio, de tamaño $ancho * largo$.

Definir(**in/out** d: `diccMat`($coor, \sigma$), **in** c: $coor$, **in** s: σ)
Pre ≡ {$d =_{\text{obs}} d_0 \wedge$ enRango(c_1, c_2, d)}
Post ≡ {$d =_{\text{obs}}$ definir(c_1, c_2, s, d_0)}
Complejidad: $\Theta(copy(s))$
Descripción: define el significado s en el diccMat, en la posicion representada por c.
Aliasing: Hay alising, pero no se como explicarlo TODO

Definido?(**in** d: `diccMat`($coor, \sigma$), **in** c: $coor$) $\rightarrow res$: `bool`
Pre ≡ {enRango(c_1, c_2, d)}
Post ≡ {$res =_{\text{obs}}$ def?(c_1, c_2, d)}
Complejidad: $\Theta(copy(s))$
Descripción: devuelve `true` si y solo si c tiene un valor en el diccMat.

Significado(**in** d: `diccMat`($coor, \sigma$), **in** c: $coor$) $\rightarrow res$: σ
Pre ≡ {enRango(c_1, c_2, d) $\wedge_L$ def?(c_1, c_2, d)}
Post ≡ {alias($res =_{\text{obs}}$ obtener(c_1, c_2, d)}
Complejidad: $\Theta(copy(s))$
Descripción: Devuelve el valor de d en la posicion c.

Borrar(**in/out** d: `diccMat`($coor, \sigma$), **in** c: $coor$)
Pre ≡ {$d = d_0 \wedge$ enRango(c_1, c_2, d) $\wedge_L$ def?(c_1, c_2, d)}
Post ≡ {$d =_{\text{obs}}$ borrar(c_1, c_2, d_0)}
Complejidad: $\Theta\left(\sum_{c' \in d.claves} equal(c, c')\right)$
Descripción: Elimina el valor en la posicion c en d.

Coordenadas(**in** d: `diccMat`($coor, \sigma$)) $\rightarrow res$: `itConj`($coor$)
Pre ≡ {true}
Post ≡ {alias(esPermutacion?(SecuSuby(res), $claves(d)$))}
Complejidad: $\Theta(1)$
Descripción: Devuelve un iterador al conjunto de claves de d.

Ancho(**in** d: `diccMat`($coor, \sigma$)) $\rightarrow res$: Nat
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ ancho(d)}
Complejidad: $\Theta(1)$
Descripción: Devuelve el ancho de d

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

Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{largo}(d)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve el largo de d

4.0.1. Especificacion de las operaciones auxiliares utilizadas en la interfaz

TAD DiccMatriz(Nat, Nat,σ)

géneros diccMat(Nat, Nat, σ)

exporta diccMat(Nat, Nat, σ), generadores, observadores, borrar, claves

usa Nat, Bool, Conj(tupla(Nat, Nat))

igualdad observacional

$$(\forall d, d' : \text{DiccMat(Nat, Nat, }\sigma)) \left(d =_{\text{obs}} d' \iff \begin{pmatrix} \text{(ancho(d) } =_{\text{obs}} \text{ ancho(d') } \land \text{ largo(d) } =_{\text{obs}} \\ \text{largo(d') } \land_L \\ (\forall \text{ x,y: Nat) (def?(x,y,d) } =_{\text{obs}} \text{ def?(x,y,d'))} \\ \land_L \\ \text{def?(x,y,d) } \Rightarrow_L \text{ obtener(x,y,d) } =_{\text{obs}} \text{ obte-} \\ \text{ner(x,y,d')} \end{pmatrix} \right)$$

observadores básicos

largo : diccMat(Nat $\times$ Nat $\times$ σ) $\longrightarrow$ Nat

ancho : diccMat(Nat $\times$ Nat $\times$ σ) $\longrightarrow$ Nat

def? : Nat x $\times$ Nat y $\times$ diccMat(Nat $\times$ Nat $\times$ σ) d $\longrightarrow$ Bool {enRango(x,y,d)}

obtener : Nat x $\times$ Nat y $\times$ diccMat(Nat $\times$ Nat $\times$ σ) d $\longrightarrow$ σ {enRango(x,y,d) $\land_L$ def?(x,y,d)}

generadores

vacio : Nat *largo* $\times$ Nat *ancho* $\longrightarrow$ diccMat(Nat,Nat,σ) {largo$*$ancho $>$ 0}

definir : Nat x $\times$ Nat y $\times$ σ s $\times$ diccMat(Nat $\times$ Nat $\times$ σ) d $\longrightarrow$ diccMat(Nat,Nat,σ) {enRango(x,y,d)}

otras operaciones

borrar : Nat x $\times$ Nat y $\times$ diccMat(Nat $\times$ Nat $\times$ σ) d $\longrightarrow$ diccMat(Nat,Nat,σ) {enRango(x,y,d) $\land_L$ def?(x,y,d)}

claves : diccMat(Nat $\times$ Nat $\times$ σ) $\longrightarrow$ conj(tupla(Nat,Nat))

otras operaciones (no exportadas)

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

axiomas $\forall$ x, y, m, n: Nat $\forall$ d: diccMat(Nat,Nat,σ) $\forall$ s: σ

largo(vacio(m,n)) $\equiv$ m

ancho(vacio(m,n)) $\equiv$ n

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

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

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

def?(x,y, definir(m,n,s,d)) $\equiv$ (x = m $\land$ y = n) $\lor$ def?(x,y,d)

obtener(x,y, definir(m,n,s,d)) $\equiv$ **if** (x = m $\land$ y = n) **then** s **else** obtener(x,y,d) **fi**

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

claves(vacio) $\equiv \emptyset$

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

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

Fin TAD

Representación

`Diccionario Matriz` **se representa con** `dicc`

donde `dicc` es `tupla`(*posiciones*: `arregloDimensionable de` $< bool, \sigma >$, *claves*: `conjLineal(coor)` , *ancho*: `Nat` , *largo*: `Nat`)

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ $longitud(e.posiciones) = cardinal(e.claves)$

Abs : estr e $\longrightarrow$ diccMat {Rep(e)}

Abs(e) $\equiv$ $=_{\mathrm{obs}}$ d: diccMat | $(\forall d : diccMat)$ e.claves = coordenadas(d) $\wedge$ e.ancho = ancho(d) $\wedge$ e.largo = largo(d) $\wedge$ $(\forall c \leftarrow e.claves)$ e.posiciones$[c.campo_1 * e.ancho + c.campo_2]$ = siginificado(c,d)

Algoritmos

Trabajo Práctico IIAlgoritmos del modulo

---

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

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

Complejidad: $\Theta(a * l)$

---

iDefinir(in/out d: `diccMat`, **in** c: `coor`, **in** s: σ)

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

Complejidad: $\Theta(copy(s))$
Justificacion: $Definido?$ y $Aplanar$ tienen costo $\Theta(1)$, $AgregarRapido$ y $Definir$ tienen costo $\Theta(copy(s))$. Aplicando algebra de ordenes: $\Theta(1) + \Theta(1) + \Theta(copy(s)) + \Theta(copy(s)) = \Theta(copy(s))$

---

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 consular 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`) $\rightarrow res : \sigma$

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

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

parámetros formales

géneros α

operaciones $\bullet < \bullet : \alpha \times \alpha \longrightarrow$ bool — Relación de orden total estricto[1]

géneros colaMinPrior(α)

exporta colaMinPrior(α), generadores, observadores

usa BOOL

observadores básicos

vacía?	: colaMinPrior(α)	$\longrightarrow$ bool	
próximo	: colaMinPrior(α) c	$\longrightarrow$ α	$\{\neg$ vacía?(c)$\}$
desencolar	: colaMinPrior(α) c	$\longrightarrow$ colaMinPrior(α)	$\{\neg$ vacía?(c)$\}$

generadores

vacía	:	$\longrightarrow$ colaMinPrior(α)
encolar	: $\alpha \times$ colaMinPrior(α)	$\longrightarrow$ colaMinPrior(α)

otras operaciones

tamaño	: colaMinPrior(α)	$\longrightarrow$ nat

axiomas $\forall$ c: colaMinPrior(α), $\forall$ e: α

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

Fin TAD

[1] Una 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: `colaMin`(α).

5.2.1. Operaciones básicas de Cola de mínima prioridad

VACÍA() $\rightarrow$ *res* : `colaMin`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacía}
Complejidad: O(1)
Descripción: Crea una cola de prioridad vacía

VACÍA?(**in** c: `colaMin`(α)) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacía?(c)}
Complejidad: O(1)
Descripción: Devuelve `true` si y sólo si la cola está vacía

PRÓXIMO(**in** c: `colaMin`(α)) $\rightarrow$ *res* : α
Pre $\equiv$ {$\neg$vacía?(c)}
Post $\equiv$ {alias($res =_{\text{obs}}$ próximo(c))}
Complejidad: O(1)
Descripción: Devuelve el próximo elemento a desencolar
Aliasing: *res* es modificable si y sólo si c es modificable

DESENCOLAR(**in/out** c: `colaMin`(α))
Pre $\equiv$ {$\neg$vacía?(c) $\wedge$ $c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ desencolar(c_0)}
Complejidad: O(log(tamaño(c)))
Descripción: Quita el elemento más prioritario

ENCOLAR(**in/out** c: `colaMinPrior`(α), **in** p: `nat`, **in** a: α) $\rightarrow$ *res* : `itColaMin`(*alpha*)
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ encolar(p,c_0) $\wedge$ $res =_{\text{obs}}$ CrearIt(ColaASecu(c_0), a) $\wedge$ alias(SecuSuby(res) = ColaASecu(c))}
Complejidad: O(log(|c|)) + *copy*(a)
Descripción: Agrega el elemento a de tipo α con prioridad p a la cola
Aliasing: Se agrega el elemento por copia

5.3. Representación

5.3.1. Representación de colaMinPrior

`colaMinPrior`(α) **se representa con** `colaMin`

donde `colaMin` es `tupla`(*proximo*: `puntero(Nodo)`, *tamao*: `nat`)

donde `Nodo` es `tupla`(*prior*: `Nat`, *elem*: α, *padre*: `puntero(Nodo)`, *izq*: `puntero(Nodo)`, *der*: `puntero(Nodo)`)

5.3.2. Invariante de Representación (Rehacer con nueva estructura)

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

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

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
$(\forall i : \text{nat})\ (i < longitud(e.proximos)) \Rightarrow_{\text{L}}$
$((\forall j : \text{nat})\ j \leqslant i \Rightarrow_{\text{L}}$ (e.proximos[j].prioridad $\leqslant$ e.proximos[i].prioridad) $\wedge$
$((\forall n : \text{nat})\ n < longitud(e.proximos) \Rightarrow_{\text{L}}$ e.proximos[$e.elementos[n].id$]$.elemCola \to siguiente.elem =_{\text{obs}}$ $e.elementos[n].elem$)

5.3.3. Función de Abstracción

Abs : estr e $\longrightarrow$ colaMinPrior {Rep(e)}

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

5.4. Algoritmos

---

iVacía() $\to$ res : colaMin(α)

1: $res \leftarrow < NULL, 0 >$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iVacía?(**in** c: `colaMin`(α)) $\to$ res : Bool

1: $res \leftarrow (c.proximo = NULL)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iPróximo(**in** c: `colaMinPrior`(α)) $\to$ res : α

1: $res \leftarrow CrearIt(c).Siguiente \to elem$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

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

```
it ← CrearIt(c)                                                        ▷ Θ(1)
A ← DecimalABinario(c.tamao)      ▷ Convertir un decimal a binario tiene complejidad logaritmica del largo de número Θ(log(|c|))
for i ← 1 to tam(A) - 1 do        ▷ Se empieza desde el segundo elemento porque el iterador ya está parado en el primero que por precondición no está invalidado Θ(|A|) = Θ(log(|c|))
    if A[i] = 0 then
        AvanzarIzq(it)                                                 ▷ Θ(1)
    else
        AvanzarDer(it)                                                 ▷ Θ(1)
    end if
end for
swapCola(c.proximo, it.siguiente)                                      ▷ Θ(1)
EliminarSiguiente(it)                                                  ▷ Θ(1)
if ( thenc.tamao > 1)
    siftDown(c)                                                        ▷ Θ(log(|c|))
end if
```

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

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

```
Nodo : nuevo ←< prioridad, a, NULL, NULL, NULL >                       ▷ Θ(1)
it ← CrearIt(c)                                                        ▷ Θ(1)
A ← DecimalABinario(c.tamao + 1) ▷ Convertir un decimal a binario tiene complejidad logaritmica del largo de número, buscamos la posición donde vamos a agregar el nuevo elemento Θ(log(|c|))
i ← 0                                                                  ▷ Θ(1)
while i <tam(A) - 1 do                                                 ▷ Θ(|A|) = Θ(log(|c|))
    if A[i] = 0 then                                                   ▷ Θ(1)
        AvanzarIzq(it)                                                 ▷ Θ(1)
    else
        AvanzarDer(it)                                                 ▷ Θ(1)
    end if
    i ← i + 1                                                          ▷ Θ(1)
end while
if A[i] = 0 then                                                       ▷ Θ(1)
    AgregarSiguienteIzq(it, nuevo)                                     ▷ Θ(1)
else
    AgregarSiguienteDer(it, nuevo)                                     ▷ Θ(1)
end if
siftUp(c)                                                              ▷ Θ(log(|c|))
```

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

siftDown(**in/out** c: `colaMin`(α))

```
it ← CrearIt(c)                                                              ▷ Θ(1)
swap ← true
while ¬esHoja(it) ∧ swap do      ▷ Lo máximo que se puede llegar a avanzar es la altura del árbol Θ(log(|c|))
    swap ← false                                                             ▷ Θ(1)
    if haySiguienteIzq(it) ∧ haySiguienteDer(it) then                        ▷ Θ(1)
        if it.siguiente → izq → prior < it.siguiente → der → prior then      ▷ Θ(1)
            if it.siguiente → izq → prior < it.siguiente → prior then        ▷ Θ(1)
                swapCola(it.siguiente → izq, it.siguiente)                   ▷ Θ(1)
                swap ← true                                                  ▷ Θ(1)
                AvanzarIzq(it)                                               ▷ Θ(1)
            end if
        else
            if it.siguiente → der → prior < it.siguiente → prior then        ▷ Θ(1)
                swapCola(it.siguiente → der, it.siguiente)                   ▷ Θ(1)
                swap ← true                                                  ▷ Θ(1)
                AvanzarDer(it)                                               ▷ Θ(1)
            end if
        end if
    else
        if haySiguienteIzq(it) then                                          ▷ Θ(1)
            if it.siguiente → izq → prior < it.siguiente → prior then        ▷ Θ(1)
                swapCola(it.siguiente → izq, it.siguiente)                   ▷ Θ(1)
                swap ← true                                                  ▷ Θ(1)
                AvanzarIzq(it)                                               ▷ Θ(1)
            end if
        else
            if it.siguiente → der → prior < it.siguiente → prior then        ▷ Θ(1)
                swapCola(it.siguiente → der, it.siguiente)                   ▷ Θ(1)
                swap ← true                                                  ▷ Θ(1)
                AvanzarDer(it)                                               ▷ Θ(1)
            end if
        end if
    end if
end while
```

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

siftUp(**in/out** c: `colaMin(α)`)

```
it ← CrearIt(c)                                   ▷ Θ(1)
swap ← true
A ← DecimalABinario(c.tamao)                      ▷ Convertir un decimal a binario tiene complejidad logaritmica del largo de número Θ(log(|c|))
for i ← 1 to tam(A) - 1 do                        ▷ Se empieza desde el segundo elemento porque el iterador ya está parado en el primero que por precondición no está invalidado, se avanza hasta el último elemento que es el que hay que reacomodar Θ(|A|) = Θ(log(|c|))
    if A[i] = 0 then
        AvanzarIzq(it)                            ▷ Θ(1)
    else
        AvanzarDer(it)                            ▷ Θ(1)
    end if
end for
while tienePadre(it) ∧ swap do                    ▷ Lo máximo que se puede llegar a avanzar es la altura del árbol Θ(log(|c|))
    swap → false                                  ▷ Θ(1)
    if it.siguiente → prior < it.siguiente → padre → prior then   ▷ Θ(1)
        swapCola(it.siguiente → padre, it.siguiente)              ▷ Θ(1)
        swap → true                               ▷ Θ(1)
        Retroceder(it)                            ▷ Θ(1)
    end if
end while
```

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

swapCola(**in/out** p: `puntero(Nodo)`, **in/out** q: `puntero(Nodo)`)

```
Nodo : aux ← (*p)                                 ▷ Θ(Copy(α))
p → padre ← q → padre                             ▷ Θ(1)
p → izq ← q → izq                                 ▷ Θ(1)
p → der ← q → der                                 ▷ Θ(1)
q → padre ← aux.padre                             ▷ Θ(1)
q → izq ← aux.izq                                 ▷ Θ(1)
q → der ← aux.der                                 ▷ Θ(1)
```

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

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 : α
Pre ≡ {true}
Post ≡ {res $=_{obs}$ s}
Complejidad: $O(copy(s))$
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 ≡ {true}
Post ≡ {res $=_{obs}$ 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: `diccString`(α), **in** c: `string`)) $\rightarrow$ res : `bool`
Pre ≡ {true}
Post ≡ {res $=_{\text{obs}}$ 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: `diccString`(α), **in** c: `string`, **in** $s: \alpha$)
Pre ≡ {$d =_{\text{obs}} d_0$}
Post ≡ {$d =_{\text{obs}}$ definir(c, s, d_0)}
Complejidad: $O(|c|+copy(s))$ 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: `diccString`(α), **in** c: `string`) $\rightarrow$ res : α
Pre ≡ {def?(c, d)}
Post ≡ {alias(res $=_{\text{obs}}$ 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: `diccString(`α`)`, **in** c: `string`)

Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge$ def?(d, c)}
Post $\equiv$ {$d =_{\text{obs}}$ 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: `diccString(`α`)`) $\rightarrow$ res : `itConj(String)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), c)) $\wedge$ vacia?(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: `diccString(`α`)`) $\rightarrow$ res : `itDiccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), d)) $\wedge$ vacia?(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: `itDiccString(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ haySiguiente?(it)}
Complejidad: $O(1)$
Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

HayAnterior(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ hayAnterior?(it)}
Complejidad: $O(1)$
Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para retroceder.

SiguienteSignificado(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {haySiguiente?(it)}
Post $\equiv$ {alias(res $=_{obs}$ haySiguiente?(it).significado)}
Complejidad: $O(1)$
Descripción: devuelve el significado del elemento siguiente del iterador
Aliasing: res es modificable si y sólo si it es modficable.

AnteriorSignificado(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {hayAnterior?(it)}
Post $\equiv$ {alias(res $=_{obs}$ haySiguiente?(it).significado)}
Complejidad: $O(1)$
Descripción: devuelve el significado del elemento anterior del iterador
Aliasing: res es modificable si y sólo si it es modficable.

AVANZAR(**in/out** it : `itDiccString(`α`)`)
Pre $\equiv$ {it = it_0 $\wedge$ haySiguiente?(it)}
Post $\equiv$ {it $=_{obs}$ avanzar(it_0)}
Complejidad: $O(1)$
Descripción: avanza a la posicion siguiente del iterador.

RETROCEDER(**in/out** it : `itDiccString(`α`)`)
Pre $\equiv$ {it = it_0 $\wedge$ hayAnterior?(it)}
Post $\equiv$ {it $=_{obs}$ 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.

($\forall c$: diccString((α)))()

Rep : estr $\longrightarrow$ bool

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

nodoValido : puntero(Nodo) *nodo* $\longrightarrow$ bool

iteradorValido : puntero(Nodo) *nodo* $\longrightarrow$ bool

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

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

Abs(e) $\equiv$ $=_{\text{obs}}$ d: diccString(α) | ($\forall s$: string)(def?(d, s) $=_{\text{obs}}$
Definido?(d,s) $\wedge$
def?(d, s) $\Rightarrow_{\text{L}}$ obtener(s,d) $=_{\text{obs}}$
Obtener(d,s)
)

6.2. Algoritmos

---

iCrearDiccionario() $\rightarrow$ *res* : *estr*
Pre $\equiv$ true

```
arreglo(puntero(Nodo)) : res.raiz ← CrearArreglo(27)      ▷ O(1)
nat : i ← 0                                                ▷ O(1)
while i < long(res.raiz) do                                ▷ O(1)
    res.raiz[i] ← NULL                                     ▷ O(1)
end while
res.listaIterable ← Vacia()                                ▷ 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 $=_{obs}$ vacio()

---

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

```
nat : i ← 0                                                ▷ O(1)
nat : letra ← ord(c[0])                                    ▷ O(1)
puntero(Nodo) : arr ← d.raiz[letra]                        ▷ O(1)
while i < longitud(c) ∧ ¬arr = NULL do                     ▷ O(|c|)
    i ← i + 1                                              ▷ O(1)
    letra ← ord(c[i])                                      ▷ O(1)
    arr ← (*arr).arbolTrie[letra]                          ▷ O(1)
end while
if i = longitud(c) then                                    ▷ O(1)
    res ← (*arr).infoValida                                ▷ O(1)
else
    res ← false                                            ▷ O(1)
end if
```

Complejidad: $O(|c|)$
Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego itera sobre los caracteres restantes hasta el final del String c, por lo que hace $|c|$ operaciones. Finalmente pregunta si el significado encontrado es válido o no.
Post $\equiv$ res $=_{obs}$ def?(d,c)

iDefinir(in/out d: `estr`, **in** c: `string`, **in** $s : \alpha$)
Pre ≡ 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**	$\triangleright$ $O(1)$
$Nodo : nuevo$	$\triangleright$ $O(1)$
$arreglo(puntero(Nodo)) : nuevo.arbolTrie \leftarrow CrearArrelgo(27)$	$\triangleright$ $O(1)$
$nuevo.infoValida \leftarrow false$	$\triangleright$ $O(1)$
$d.raiz[letra] \leftarrow puntero(nuevo)$	$\triangleright$ $O(1)$
end if	
$puntero(Nodo) : arr \leftarrow d.raiz[letra]$	$\triangleright$ $O(1)$
while $i < longitud(c)$ **do**	$\triangleright$ $O(\|c\|)$
$i \leftarrow i + 1$	$\triangleright$ $O(1)$
$letra \leftarrow ord(c[i])$	$\triangleright$ $O(1)$
if $arr.arbolTrie[letra] = NULL$ **then**	$\triangleright$ $O(1)$
$Nodo : nuevoHijo$	$\triangleright$ $O(1)$
$arreglo(puntero(Nodo)) : nuevoHijo.arbolTrie \leftarrow CrearArrelgo(27)$	$\triangleright$ $O(1)$
$nuevoHijo.infoValida \leftarrow false$	$\triangleright$ $O(1)$
$arr.arbolTrie[letra] \leftarrow puntero(nuevoHijo)$	$\triangleright$ $O(1)$
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: $O(|c| + copy(s))$
Justificación: Itera sobre la cantidad de caracteres del String c y en caso de que algún caracter no esté definido crea un arrelglo de 27 posiciones, por lo que realiza $|c|$ operaciones. Luego copia el significado pasado por parámetro en O(copy(s)) y finalmente agrega en la lista un puntero al nodo creado.
Post ≡ d $=_{obs}$ definir(c,s,d_0)

iObtener(in d: `estr`, **in** c: `string`) $\rightarrow res : \alpha$
Pre ≡ 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**	$\triangleright$ $O(\|c\|)$
$i \leftarrow i + 1$	$\triangleright$ $O(1)$
$letra \leftarrow ord(c[i])$	$\triangleright$ $O(1)$
$arr \leftarrow (*arr).arbolTrie[letra]$	$\triangleright$ $O(1)$
end while	
$res \leftarrow (*arr).info$	$\triangleright$ $O(1)$

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

iEliminar(**in/out** d: `estr`, **in** c: `string`)
Pre ≡ d $=_{obs}$ d_0 ∧ 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
if $tieneHermanos(arr)$ **then** ▷ $O(1)$
  $(*arr).infoValida \leftarrow false$ ▷ $O(1)$
else
  $i \leftarrow i + 1$ ▷ $O(1)$
  $puntero(Nodo) : del \leftarrow tope(pil)$ ▷ $O(1)$
  $del \leftarrow NULL$ ▷ $O(1)$
  $Desapilar(pil)$ ▷ $O(1)$
  while $i < longitud(c) \wedge \neg tieneHermanosEInfo(*tope(pil))$ **do** ▷ $O(|c|)$
    $del \leftarrow tope(pil)$ ▷ $O(1)$
    $del \leftarrow NULL$ ▷ $O(1)$
    $Desapilar(pil)$ ▷ $O(1)$
    $i \leftarrow i + 1$ ▷ $O(1)$
  end while
  if $i = longitud(c)$ **then** ▷ $O(1)$
    $d.raiz[ord(c[0])] \leftarrow NULL$ ▷ $O(1)$
  end if
end if

Complejidad: $O(|c|)$
Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego crea una pila en O(1). Recorre el resto de los caracteres del String c y apila cada uno de los Nodos encontrado en la pila (0(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 raiz.
Post ≡ d $=_{obs}$ borrar(d_0,c)

tieneHermanos(**in** $nodo$: `puntero(Nodo)`) → res : $bool$
Pre ≡ 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
$res \leftarrow i < l$ ▷ $O(1)$

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

tieneHermanosEInfo(**in** *nodo* : `puntero(Nodo)`) $\rightarrow$ *res* : *bool*
Pre $\equiv$ nodo!=NULL

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

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..longitud(*nodo.arbolTrie)) (*nodo.arbolTrie[i] != NULL)) $\wedge$ (*nodo).infoValida = true