DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 2

Algoritmos y Estructuras de Datos II

Grupo: 21

Integrante	LU	Correo electrónico
Langberg, Andrés	249/14	andreslangberg@gmail.com
Walter, Nicolás	272/14	nicowalter25@gmail.com
Sticco, Patricio Bernardo	337/14	pbsticco@hotmail.com
Len, Julián	467/14	julianlen@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. **TAD** POSICION **ES** TUPLA(X:NAT, Y:NAT)
2. **TAD** DIRECCION **ES** ENUM{ IZQ,DER,ARRIBA,ABAJO}
3. **TAD** AGENTE **ES** NAT
4. **TAD** NOMBRE **ES** STRING
5. Suponemos que contamos con el TAD DiccionarioM, donde la funcion vacio() toma como parámetro un 'k', cuyo valor acota superiormente a la cantidad de claves.
6. Asumimos a $|Nm|$ como la longitud más larga entre todos los nombres del campusSeguro, Na la cantidad de agentes y Ne la cantidad de estudiante en el momento donde será usado y Nh la cantidad de hippies, en el momento donde va a ser usado.
7. Por consigna, se desestiman los costos de eliminación de elementos, con lo cual se pueden ignorar en el cálculo de complejidades.

1. Diseño del Tipo Campus

1.1. Especificación

Se usa el Tad Campus especificado por la cátedra.

1.2. Aspectos de la interfaz

1.2.1. Interfaz

Se explica con especificación de Campus

Género `campus`

Operaciones básicas de Campus

CrearCampus(**in** c: nat, **in** f: nat) $\longrightarrow res : campus$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} crearCampus(c, f)$ }
Complejidad: $\mathcal{O}(\text{f}^2 * c^2)$
Descripción: Crea un campus de c columnas y f filas.

Filas?(**in** c: $campus$) $\longrightarrow res : nat$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} filas(c)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de filas en el campus.

Columnas?(**in** c: $campus$) $\longrightarrow res : nat$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} columnas(c)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de columnas en el campus.

Ocupada?(**in** c: $campus$, **in** p: $posicion$) $\longrightarrow res : bool$
Pre $\equiv$ { $posValida(p, c)$ }
Post $\equiv$ { $res =_{\text{obs}} ocupada?(p, c)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve $true$ sii p esta ocupada por un obstaculo.

AgregarObstaculo(**in/out** c: $campus$, **in** p: $posicion$) $\longrightarrow$
Pre $\equiv$ { $c =_{\text{obs}} \text{c}_0 \wedge$ posValida(p,c) $\wedge_{\text{L}} \neg$ocupada?(p,c)}
Post $\equiv$ { $c =_{\text{obs}} agregarObstaculo(p, \text{c}_0)$}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve $true$ sii p esta ocupada por un obstaculo.

PosValida?(**in** c: $campus$, **in** p: $posicion$) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} posValida?(p, c)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve $true$ sii p es parte del mapa.

EsIngreso?(**in** c: $campus$, **in** p: $posicion$) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} esIngreso?(p, c)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve $true$ sii p es un ingreso.

VECINOS(**in** c: $campus$, **in** p: $posicion$) $\longrightarrow res : conj(posicion)$
Pre $\equiv$ { $posValida(p, c)$ }
Post $\equiv$ { $res =_{\text{obs}} vecinos(p, c)$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el conjunto de posiciones vecinas a p.

VECINOSCOMUNES(**in** c: $campus$, **in** p: $posicion$, **in** $p2$: $posicion$) $\longrightarrow res : conj(posicion)$
Pre $\equiv$ { $posValida(p, c) \wedge$ posValida(p2,c)}
Post $\equiv$ { $res =_{\text{obs}} vecinos(p, c) \cap$vecinos(p2,c)}

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el conjunto de vecinos comunes entre dos posiciones. La complejidad es O(1) dado que los vecinos son a lo sumo 4, o sea, constantes.

PROXPOSICION(**in** c: $campus$, **in** dir: $direccion$, **in** p: $posicion$) $\longrightarrow res : posicion$
Pre $\equiv$ { $posValida(p, c)$ }
Post $\equiv$ { $res =_{\text{obs}} proxPosicion(p, d, c)$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve la posicion vecina a p que esta en la direccion dir.

INGRESOSMASCERCANOS(**in** c: $campus$, **in** p: $posicion$) $\longrightarrow res : conj(posicion)$
Pre $\equiv$ { $posValida(p, c)$ }
Post $\equiv$ { $res =_{\text{obs}} ingresosMasCercanos(p, c)$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el conjunto de ingresos mas cercanos a p.

1.3. Pautas de implementación

1.3.1. Estructura de representación

campus **se representa con** *estr*
donde *estr* **es**
tupla(
filas: ***nat*** $\times$
columnas: ***nat*** $\times$
mapa: ***vector(vector(bool))***
)

1.3.2. Justificación

1.3.3. Invariante de Representación

Informal

1. El mapa debe tener tantas filas como indica la estructura, lo mismo con las columnas.

Formal

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ ($true \iff$
(1) e.filas = longitud(e.mapa) $\wedge_{\text{L}}$ ($\forall$ i : nat)(i $\leq$ e.filas $\Rightarrow$ longitud(e.mapa[i])= e.columnas))

1.3.4. Función de Abstracción

Abs : estr e $\longrightarrow$ campus {Rep(e)}
($\forall$ e:estr) Abs(e) $=_{\text{obs}}$ c : campus /

$$\Big(\text{filas(c)} = \text{e.filas} \wedge \text{columnas(c)} = \text{e.columnas} \wedge_{\text{L}} (\forall\ \text{p} : \text{posicion})(\text{p.X} \leq \text{e.filas} \wedge$$
$$\text{p.Y} \leq \text{e.columnas} \Rightarrow_{\text{L}} \text{ocupada?(p,c)} \Leftrightarrow (\text{e.mapa[f])[c]}\Big)$$

1.3.5. Algoritmos

```
1: function iCrearCampus(in c: nat, in f: nat) ⟶ res : estr                ▷ O(f² * c²)
2:     var vector(vector(bool)) mapa ← vacia(vacia())                       ▷ O(1)
3:     var nat i ← 0                                                        ▷ O(1)
4:     while i≤f do                                                         ▷ O(f)
5:         var vector(bool) nuevo ← vacia()                                 ▷ O(1)
6:         var nat j ← 0                                                    ▷ O(1)
7:         while j≤c do                                                     ▷ O(c)
8:             AgregarAtras(nuevo, false)                                   ▷ O(c)
9:             j++                                                          ▷ O(1)
10:        end while
11:        AgregarAtras(mapa, nuevo)                                        ▷ O(f)
12:        i++                                                              ▷ O(1)
13:    end while
14:    res ← < f, c, mapa>                                                  ▷ O(1)
15: end function
```

```
1: function iAgregarObstaculo(in/out e: estr, in p: posicion) ⟶ res : estr   ▷ O(longitud(e.mapa[p.X]))
2:     Agregar(e.mapa[p.X], p.Y, true)                                       ▷ O(longitud(e.mapa[p.X]))
3: end function
```

```
1: function iFilas?(in e: estr) ⟶ res : nat                                 ▷ O(1)
2:     res ← e.filas                                                        ▷ O(1)
3: end function
```

```
1: function iColumnas?(in e: estr) ⟶ res : nat                              ▷ O(1)
2:     res ← e.columnas                                                     ▷ O(1)
3: end function
```

```
1: function iOcupada?(in e: estr, in p: posicion) ⟶ res : bool              ▷ O(1)
2:     res ← (e.mapa[p.X])[p.Y]                                             ▷ O(1)
3: end function
```

```
1: function iPosValida?(in e: estr, in p: posicion) ⟶ res : bool            ▷ O(1)
2:     res ← (0 < p.X) ∧ (p.X ≤ e.filas) ∧ (0 < p.Y) ∧ (p.Y ≤ e.columnas)    ▷ O(1)
3: end function
```

```
1: function iEsIngreso?(in e: estr, in p: posicion) ⟶ res : bool            ▷ O(1)
2:     res ← (p.Y = 1) ∨ (p.Y = e.filas)                                    ▷ O(1)
3: end function
```

```
1: function iVecinos(in e: estr, in p: posicion)⟶ res : conj(posicion)                ▷ O(1)
2:     var conj(posicion) nuevo ← vacio()                                              ▷ O(1)
3:     Agregar(nuevo, (p.X+1,p.Y))
4:     Agregar(nuevo, (p.X-1,p.Y))
5:     Agregar(nuevo, (p.X,p.Y+1))
6:     Agregar(nuevo, (p.X,p.Y-1))
7:     var itConj(posicion) it ← crearIt(nuevo)
8:     while haySiguiente(it) do                                                       ▷ O(c)
9:         if iPosValida?(e,siguiente(it)) then                                        ▷ O(1)
10:            avanzar(it)                                                          ▷ O(1)
11:        else
12:            eliminarSiguiente(it)                                                ▷ O(1)
13:        end if
14:    end while
15:    res ← nuevo                                                                     ▷ O(1)
16: end function
```

```
1: function iVecinosComunes(in e: estr, in p: posicion, in p2: posicion)⟶ res : conj(posicion)   ▷ O(1)
2:     var conj(posicion) v ← vecinos(e,p)                                             ▷ O(1)
3:     var conj(posicion) v2 ← vecinos(e,p2)                                           ▷ O(1)
4:     var conj(posicion) nuevo ← vacio()                                              ▷ O(1)
5:     var itConj(posicion) it ← crearIt(v)                                            ▷ O(1)
6:     while haySiguiente(it) do                                                       ▷ O(1)
7:         if Pertenece?(v2,Siguiente(it)) then                                        ▷ O(1)
8:             Agregar(nuevo, Siguiente(it))                                           ▷ O(1)
9:         end if
10:        Avanzar(it)                                                             ▷ O(1)
11:    end while
12:    res ← nuevo                                                                     ▷ O(1)
13: end function
```

```
1: function iVecinosValidos(in e: estr, in ps: conj(posicion)) ⟶ res : conj(posicion)   ▷ O(1)
2:     var conj(posicion) nuevo ← vacio()                                              ▷ O(1)
3:     var itConj(posicion) it ← crearIt(ps)                                           ▷ O(1)
4:     while haySiguiente(it) do                                                       ▷ O(1)
5:         if PosValida?(e,siguiente(it)) then                                         ▷ O(1)
6:             Agregar(nuevo, siguiente(it))                                           ▷ O(1)
7:         end if
8:         avanzar(it)                                                             ▷ O(1)
9:     end while
10:    res ← nuevo                                                                     ▷ O(1)
11: end function
```

```
1: function iDistancia(in e: estr, in p: posicion, in p2: posicion)⟶ res : nat         ▷ O(1)
2:     res ← |p.X - p2.X| + |p.Y - p2.Y|                                               ▷ O(1)
3: end function
```

```
1: function iProxPosicion(in e: estr, in d: direccion, in p: posicion)⟶ res : posicion                ▷ O(1)
2:     var posicion p2 ← p
3:     if d==izq then
4:         p2 ← <p2.X+1, p2.Y>                                                                       ▷ O(1)
5:     else
6:         if d==der then
7:             p2 ← <p2.X, p2.Y>                                                                     ▷ O(1)
8:         else
9:             if d==arriba then                                                                     ▷ O(1)
10:                p2 ← <p2.X, p2.Y-1>                                                               ▷ O(1)
11:            else
12:                p2 ← <p2.X, p2.Y+1>                                                               ▷ O(1)
13:            end if
14:        end if
15:    end if
16:    res ← p2                                                                                      ▷ O(1)
17: end function
```

```
1: function iIngresosMasCercanos(in e: estr, in p: posicion)⟶ res : conj(posicion)                   ▷ O(1)
2:     var conj(posicion) nuevo ← Vacio()                                                            ▷ O(1)
3:     if distancia(e, p, <p.x,1>) < distancia(e, p, <p.x,e.filas>) then                             ▷ O(1)
4:         Agregar(nuevo, <p.x,1>)                                                                   ▷ O(1)
5:     else
6:         if distancia(e, p, <p.x,1>) > distancia(e, p, <p.x,filas(e)>) then                        ▷ O(1)
7:             Agregar(nuevo, <p.x,e.filas>)                                                         ▷ O(1)
8:         else
9:             Agregar(nuevo, <p.x,1>)                                                               ▷ O(1)
10:            Agregar(nuevo, <p.x,e.filas>)                                                         ▷ O(1)
11:        end if
12:    end if
13:    res ← nuevo                                                                                   ▷ O(1)
14: end function
```

2. Diseño del Tipo Rastrillaje

2.1. Especificación

Se usa el Tad CampusSeguro especificado por la cátedra.

2.2. Aspectos de la interfaz

2.2.1. Interfaz

Se explica con especificación de CampusSeguro

Género `rastr`

Operaciones básicas de Rastrillaje

Campus(**in** r: $rastr$) $\longrightarrow res$: $campus$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} campus(r)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el campus.

Estudiantes(**in** r: $rastr$) $\longrightarrow res$: $conj(nombre)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} estudiantes(r)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de estudiantes presentes en el campus.

Hippies(**in** r: $rastr$) $\longrightarrow res$: $conj(nombre)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} hippies(r)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de hippies presentes en el campus.

Agentes(**in** r: $rastr$) $\longrightarrow res$: $conj(agente)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} agentes(r)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de agentes presentes en el campus.

PosEstudiantesYHippie(**in** r: $rastr$, **in** id: $nombre$) $\longrightarrow res$: $posicion$
Pre $\equiv$ { $id \in$ (estudiantes(r) $\cup$ hippies(cs))}
Post $\equiv$ { $res =_{\text{obs}} posEstudianteYHippie(id, r)$ }
Complejidad: $\mathcal{O}(|N_m|)$
Descripción: Devuelve la posición del estudiante/hippie pasado como parámetro.

PosAgente(**in** r: $rastr$, **in** a: $agente$) $\longrightarrow res$: $posicion$
Pre $\equiv$ { $a \in$ posAgente(a,r)}

Post $\equiv$ { $res =_{\text{obs}} posAgente(a, r)$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve la posición del agente pasado como parámetro. La complejidad se da en el caso promedio.

CantSanciones(**in** r: $rastr$, **in** a: $agente$) $\longrightarrow res : nat$

Pre $\equiv$ { $a \in$ cantSanciones(a,r)}

Post $\equiv$ { $res =_{\text{obs}} cantSanciones(a, r)$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve la cantidad de sanciones recibidas por el agente pasado como parámetro. La complejidad se da en el caso promedio.

CantHippiesAtrapados(**in** r: $rastr$, **in** a: $agente$) $\longrightarrow res : nat$

Pre $\equiv$ { $a \in$ agentes(r)}

Post $\equiv$ { $res =_{\text{obs}} cantHippiesAtrapados(a, r)$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve la cantidad de hippies atrapados por el agente pasado como parámetro. La complejidad se da en el caso promedio.

ComenzarRastrillaje(**in** c: $campus$, **in** d: $dicc(agente, posicion)$) $\longrightarrow res : rastr$

Pre $\equiv$ { $(\forall$ a : agente)(def?(a,d) $\Rightarrow_{\text{L}}$ (posValida?(obtener(a,d))) $\wedge$ $\neg$ocupada?(obtener(a,d),c)) $\wedge$ ($\forall$ a, a_2 : agente)((def?(a,d) $\wedge$ def?(a_2,d) $\wedge$ a $\neq a_2$) $\Rightarrow_{\text{L}}$ obtener(a,d)$\neq$ obtener(a_2,d))}

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

Complejidad: $\mathcal{O}(1)$

Descripción: Crea un Rastrillaje.

IngresarEstudiante(**in/out** r: $rastr$, **in** e: $nombre$, **in** p: $posicion$) $\longrightarrow$

Pre $\equiv$ { $r =$r$_0 \wedge$ e $\notin$ (estudiantes(r)$\cup$hippies(r)) $\wedge$ esIngreso?(p,campus(r)) $\wedge$ $\neg$estaOcupada?(p,r)}

Post $\equiv$ { $r =_{\text{obs}} ingresarEstudiante(e, p,r_0)$}

Complejidad: $\mathcal{O}(|\ Nm\ |)$

Descripción: Modifica el rastrillaje, ingresando un estudiante al campus.

IngresarHippie(**in/out** r: $rastr$, **in** h: $nombre$, **in** p: $posicion$) $\longrightarrow$

Pre $\equiv$ { $r =$r$_0 \wedge$ h $\notin$ (estudiantes(r)$\cup$hippies(r)) $\wedge$ esIngreso?(p,campus(r)) $\wedge$ $\neg$estaOcupada?(p,r)}

Post $\equiv$ { $r =_{\text{obs}} ingresarHippie(h, p,r_0)$}

Complejidad: $\mathcal{O}(|Nm|)$

Descripción: Modifica el rastrillaje, ingresando un hippie al campus.

MoverEstudiante(**in/out** r: $rastr$, **in** e: $nombre$, **in** dir: $direccion$) $\longrightarrow$

Pre $\equiv$ { $r =$r$_0 \wedge$ e $\in$ estudiantes(r) $\wedge$ (seRetira(e,dir,r) $\vee$ (posValida?(proxPosicion(posEstudianteYHippie(e,r),dir,campus(r)),campus(r)) $\wedge$ $\neg$ estaOcupada?(proxPosicion(posEstudianteYHippie(e,r),dir,campus(r)),r)))}

Post $\equiv$ { $r =_{\text{obs}} moverEstudiante(e, d,r_0)$}

Complejidad: $\mathcal{O}(|Nm|)$

Descripción: Modifica el rastrillaje, al mover un estudiante del campus.

MoverHippie(**in/out** r: $rastr$, **in** h: $nombre$) $\longrightarrow$

Pre $\equiv$ { $r =$r$_0 \wedge$ h $\in$ hippies(r) $\wedge$ $\neg$todasOcupadas?(vecinos(posEstudianteYHippie(h,r),campus(r)),r) }

Post $\equiv$ { $r =_{\text{obs}} moverHippie(r, r_0)$}

Complejidad: $\mathcal{O}(|Nm|)$ + O(Ne)

Descripción: Modifica el rastrillaje, al mover un hippie del campus.

MoverAgente(**in/out** r: $rastr$, **in** a: $agente$) $\longrightarrow$

Pre $\equiv$ { $r =_0$ $\wedge$ a $\in$ agentes(r) $\wedge_{\text{L}}$ cantSanciones(a,r) $\leq$ 3 $\wedge$ $\neg$todasOcupadas?(vecinos(posAgente(a,r),campus(r)),r)}

Post $\equiv$ { $r =_{\text{obs}} moverAgente(a, r_0)$}

Complejidad: $\mathcal{O}(|Nm|)$ + O($logNa$) + O(Ne)

Descripción: Modifica el rastrillaje, al mover un agente del campus.

MasVigilante(**in** r: $rastr$) $\longrightarrow res : agente$

Pre $\equiv$ { $true$ }

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

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el agente con mas capturas.

ConKSanciones(**in** r: $rastr$, **in** k: nat) $\longrightarrow res : conj(agente)$

Pre $\equiv$ { $true$ }

Post $\equiv$ { $res =_{\text{obs}} conKSanciones(k, r)$ }

Complejidad: $\mathcal{O}(Na) / \mathcal{O}(logNa)$

Descripción: Devuelve el agente con mas capturas. La primera vez que se llama será O(Na) luego mientras no haya sanciones, O($logNa$).

ConMismasSanciones(**in** r: $rastr$, **in** a: $agente$) $\longrightarrow res : conj(agente)$

Pre $\equiv$ { $a \in$ agentes(r)}

Post $\equiv$ { $res =_{\text{obs}} conMismasSanciones(a, r)$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el conjunto de agentes con la misma cantidad de sanciones que a.

2.3. Pautas de implementación

2.3.1. Estructura de representación

campus **se representa con** *estr*

donde *estr* **es**

tupla(

campo: ***campus*** $\times$

agentes: ***diccPromedio(agente ; datosAg)*** $\times$

posAgentesLog: ***arreglo(tupla(placa;posicion))*** $\times$

hippies: ***conjLineal(datosHoE)*** $\times$

estudiantes: ***conjLineal(datosHoE)*** $\times$

posCiviles: ***diccString(nombre;posicion)*** $\times$

posRapida: ***diccLineal(nombre;posicion)*** $\times$

quienOcupa: ***vector(vector(datosPos))*** $\times$

masVigilante: ***itConj(agente)*** $\times$

agregoEn1: ***lista(datosK))*** $\times$

hayNuevas: ***bool*** $\times$

buscoEnLog: ***vector(datosK)***
)
donde *datosAg* **es**
tupla(
QSanciones: ***nat*** ×
premios: ***nat*** ×
posActual: ***posicion*** ×
grupoSanciones: ***itConj(agente)*** ×
verK: ***itLista(datosK)***
)
donde *datosHoE* **es**
tupla(
ID: ***nombre*** ×
posActual: ***itDicc(nombre;posicion)***
)
donde *datosPos* **es**
tupla(
ocupada?: ***bool*** ×
queHay: ***clases*** ×
hayCana: ***itDicc(agente)*** ×
hayHoE: ***itConj(nombre)***
)
donde *clases* **es** enum{"agente","estudiante","hippie","obstaculo","nada"}
donde *datosK* **es**
tupla(
K: ***nat*** ×
grupoK: ***conjLineal(agente)***
)

2.3.2. Justificación

2.3.3. Invariante de Representación

Informal

1. Todos los agentes tienen distinta posicion.

2. La cantidad de sanciones se ve reflejada dos veces en la tupla DatosAg y debe ser la misma.

3. Si dos agentes tienen la misma cantidad de sanciones, pertenecen al mismo grupo. En caso contrario, sus grupos son disjuntos.

4. Todas las posiciones estan dentro del rango permitido en el campus.

5. El conjunto que contiene a todas las placas de posAgentesLog es igual al conjunto de claves de agentes.

Formal

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ ($true \iff$
(1)(2)(3)(4) ($\forall$ a,a2: Agente)(a $\neq$ a2 $\wedge$ definido?(a,e.agentes) $\wedge$ definido?(a2,e.agentes)
$\wedge_{\text{L}}$ PosValida(e.campo,obtener(a,e.agentes).PosActual) $\wedge$ PosValida(e.campo,obtener(a2,e.agentes).PosActual))) $\Rightarrow_{\text{L}}$
obtener(a,e.agentes).PosActual $\neq$ obtener(a2,e.agentes).PosActual
$\wedge$ (obtener(a,e.agentes).Qsanciones=siguiente(obtener(a,e.agentes).verK).K
$\wedge$ obtener(a,e.agentes).grupoSanciones=siguiente(obtener(a,e.agentes).verK).grupoK
$\wedge$ (a2 $\in$ obtener(a,e.agentes).grupoSanciones) $\iff$ (obtener(a,e.agentes).Qsanciones = obtener(a2,e.agentes).Qsanciones)
$\wedge$ (5) TodasLasPlacas(e,e.posAgentesLog)= claves(e.agentes)
$\wedge$ (6) ($\forall$ a3:agente, t: tupla(agente,posicion))(t $\in$ e.posAgentesLog $\wedge$ a3= Π_1(t) $\wedge_{\text{L}}$ definido?(a3,e.agentes) $\Rightarrow_{\text{L}}$ obtener(a3,e.agentes)= Π_2(t))
$\wedge$ enOrden(e.posAgentesLog) $\wedge$ enOrden(e.buscoEnLog)
$\wedge$ (7) UnionConjuntos(e,e.lista)=claves(e.agentes)
$\wedge$ ($\forall$ h,h1:tupla(nombre,itDicc(nombre;posicion)))(h $\in$ e.hippies $\wedge$ h1 $\in$ e.hippies $\wedge$ Π_1(h) $\neq$ Π_1(h1)) $\Rightarrow_{\text{L}}$ (Π_2(h) $\neq$ Π_2(h1))
$\wedge$ e.posCiviles=e.posRapida $\wedge$ ($\forall$ hi:nombre,e:nombre)((definido?(hi,e.posCiviles) $\wedge$ definido?(e,e.posCiviles)) $\Rightarrow_{\text{L}}$ obtener(e,e.posCiviles) $\neq$ obtener(hi,e.posCiviles)
$\wedge$ ($\forall$ a:agente, civ:nombre)(definido?(a,e.agentes) $\wedge$ definido?(civ,e.posCiviles))
$\Rightarrow_{\text{L}}$ (obtener(a,e.agentes) $\neq$ obtener(civ,e.posCiviles)) $\wedge$ (e.hippies $\cap$ e.estudiantes)= $\emptyset$
$\wedge$ JuntarIDS(e.estudiantes)$\cup$JuntarIDS(e.hippies)=claves(e.posCiviles))
$\wedge$ ($\forall$ i:nat, j:nat)(i$\geq$0 $\wedge$ i<e.campo.filas $\wedge$ j$\geq$0 $\wedge$ j<e.campo.columnas) $\Rightarrow_{\text{L}}$ **if** Π_1(e.quienOcupa[i][j])=false
then Π_2(e.quienOcupa[i][j]) = "*nada*"
else if Π_2(e.quienOcupa[i][j]) = "hippie"$\vee$ Π_2(e.quienOcupa[i][j]) = "*estudiante*" **then**
Π_3(e.quienOcupa[i][j])=itvacio
else
Π_4(e.quienOcupa[i][j])=itvacio **fi**
fi
$\wedge$ ($\forall$ k:nat)(($\exists$ i:nat)(i$\geq$0 $\wedge$ i<longitud(e.agregoEn1) $\Rightarrow_{\text{L}}$ e.agregoEn1[i].K=k) $\iff$ ($\exists$ ag:agente)(definido?(ag,e.agentes)
$\Rightarrow_{\text{L}}$ obtener(ag,e.agentes).Qsanciones=k $\wedge$ ag $\in$ e.agregoEn1[i].grupoK))
$\wedge$ ($\forall$ k:nat)(($\exists$ i:nat)(i$\geq$0 $\wedge$ i<longitud(e.buscoEnLog) $\Rightarrow_{\text{L}}$ e.buscoEnLog[i].K=k) $\iff$ ($\exists$ ag:agente)(definido?(ag,e.agentes)
$\Rightarrow_{\text{L}}$ obtener(ag,e.agentes).Qsanciones=k $\wedge$ ag $\in$ e.buscoEnLog[i].grupoK)))

2.3.4. Función de Abstracción

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

$(\forall\ e$:estr) Abs$(e) =_{obs} c$: rastrillaje /

$\Big($campus(r) = e.campo $\wedge$ estudiantes(r) = e.estudiantes $\wedge$ hippies(r)=e.hippies $\wedge$ agentes(r)=e.agentes
$\wedge$ ($\forall$ n:nombre)((definido?(n,posEstudianteYHippie(n,r)) $\Longleftrightarrow$ definido?(n,e.diccString))
$\Rightarrow_{L}$ obtener(n,e.diccString)=obtener(n,posEstudianteYHippie(n,r))) $\wedge$

2.3.5. Algoritmos

```
1: function iCampus(in e: estr)⟶ res : campus                                   ▷ O(1)
2:     res ← e.campo
3: end function
```

```
1: function iEstudiantes(in e: estr)⟶ res : itConj(nombre)                      ▷ O(1)
2:     res ← crearIt (e.estudiantes)
3: end function
```

```
1: function iHippies(in e: estr)⟶ res : itConj(nombre)                          ▷ O(1)
2:     res ← crearIt (e.hippies)
3: end function
```

```
1: function iAgentes(in e: estr)⟶ res : itConj(agente)                          ▷ O(1)
2:     res ← claves(e.agentes)
3: end function
```

```
1: function iposEstudiantesYHippie(in e: estr, in n: nombre)⟶ res : posicion    ▷ O(|N_m|)
2:     res ← obtener(n,e.posCiviles)
3: end function
```

```
1: function iposAgente(in e: estr in a: agente)⟶ res : posicion                 ▷ O(1)(promedio)
2:     res ← obtener(a,e.agentes).posActual
3: end function
```

```
1: function icantSanciones(in e: estr, in a: agente)⟶ res : nat                 ▷ O(1)(promedio)
2:     res ← obtener(a,e.agentes).Qsanciones
3: end function
```

```
1: function icantHippiesAtrapados(in e: estr, in a: agente)⟶ res : nat          ▷ O(1)(promedio)
2:     res ← obtener(a,e.agentes).premios
3: end function
```

```
1: function imasVigilante(in e: estr)⟶ res : agente                             ▷ O(1)
2:     res ← siguiente(e.masVigilante)
3: end function
```

```
1: function iconMismasSanciones(in e: estr in a: agente)⟶ res : conj(agente)    ▷ O(1)
2:     res ← siguiente(obtener(e.agentes,a).grupoSanciones)
3: end function
```

```
1: function iconKSanciones(in e: estr in k: nat)⟶ res : conj(agente)   ▷ O(Na) la primera vez, luego mientras no haya sanciones O(logNa)
2:     if ¬e.hayNuevas then                                                ▷ O(1)
3:         var nat i ← BusquedaBin(e.buscoEnLog, k)                        ▷ O(logNa)
4:         res ← e.buscoEnLog[i].grupoK                                    ▷ O(1)
5:     else
6:         var itLista(datosK) itK ← crearIt(e.agregoEn1)                  ▷ O(1)
7:         while haySiguiente(it) do                                       ▷ O(1)
8:             buscoEnLog [i] ← siguiente(itK)                             ▷ O(1)
9:             avanzar(itK)
10:        end while
11:        var nat i ← BusquedaBin (e.buscoEnLog, k)                       ▷ O(logNa)
12:        res ← e.buscoEnLog[i].grupoK                                    ▷ O(1)
13:        e.hayNuevas ← false                                             ▷ O(1)
14:    end if
15: end function
```

```
1: function iINGRESARESTUDIANTE(in/out e: estr, in n: nombre, in p: posicion)                    ▷ O(| Nm |)
2:     if esHippizable(e,p) then                                                                  ▷ O(1)
3:         if esCapturable(e,p) then                                                              ▷ O(1)
4:             var conj(posicion) v ← vecinos(e.campus, p)                                        ▷ O(1)
5:             var itConj(posicion) it ← crearIt(v)                                               ▷ O(1)
6:             while haySiguiente(it) do
7:                 if e.quienOcupa[siguiente(it).X] [siguiente(it).Y].queHay == "agente" then     ▷ O(1)
8:                     recompensar(e, siguiente(it))                                              ▷ O(1)
9:                 end if
10:                avanzar(it)
11:            end while
12:        else
13:            definir(e.posCiviles, n, p)                                                        ▷ O(| Nm |)
14:            var itDicc(nombre, posicion) iterPos ← definirRapido(e.posRapida,n,p)              ▷ O(1)
15:            e.quienOcupa[p.X] [p.Y] ← < true,"hippie",crearIt(), agregarRapido(e.hippies,<n,iterPos>)>   ▷ O(1)
16:            var conj(posicion) Ps ← vecinos(e.campus, p)                                       ▷ O(1)
17:            var itConj(posicion) it ← crearIt(Ps)                                              ▷ O(1)
18:            while haySiguiente(it) do
19:                if esEstudiante(e,siguiente(it)) ∧ esHippizable(e,siguiente(it)) then          ▷ O(1)
20:                    Hippizar(e, siguiente(it))                                                 ▷ O(1)
21:                    if esCapturable(e,siguiente(it)) then                                      ▷ O(1)
22:                        capturarHippie(e,siguiente(it))                                        ▷ O(| Nm |)
23:                    end if
24:                else
25:                    if esEstudiante(e,siguiente(it)) ∧ esCapturable(e, siguiente(it)) then     ▷ O(1)
26:                        var itConj(posicion) itAg ← vecinos(e.campus, siguiente(it))           ▷ O(1)
27:                        while haySiguiente(itAg) do                                            ▷ O(1)
28:                            if esAgente(e, siguiente(itAg)) then                               ▷ O(1)
29:                                sancionar(e,siguiente(itAg))                                   ▷ O(1)
30:                            end if
31:                            avanzar(itAg)
32:                        end while
33:                    else
34:                        if esHippie(e, siguiente(it)) ∧ esCapturable(e,siguiente(it)) then     ▷ O(1)
35:                            capturarHippie(e,siguiente(it))                                    ▷ O(| Nm |)
36:                        end if
37:                    end if
38:                end if
39:                avanzar(it)
40:            end while
41:        end if
42:    else
43:        definir(e.posCiviles, n, p)                                                            ▷ O(| Nm |)
44:        var itDicc(nombre, posicion) iterPos ← definirRapido(e.posRapida, n, p)                ▷ O(1)
45:        e.quienOcupa[p.X] [p.Y] ← < true,"estudiante",crearIt(), agregarRapido(e.estudiantes,<n,iterPos>)>
46:        var conj(posicion) Ps ← vecinos(e.campus, p)                                           ▷ O(1)
47:        var itConj(posicion) it ← crearIt(ps)                                                  ▷ O(1)
48:        while haySiguiente(it) do
49:            if esHippie(e,siguiente(it)) ∧ esEstudiantizable(e,siguiente(it)) then             ▷ O(1)
50:                Estudiantizar(e,siguiente(it))                                             ▷ O(1)
51:            else
52:                if esEstudiante(e,siguiente(it)) ∧ esCapturable(e,siguiente(it)) then      ▷ O(1)
53:                    var itConj(posicion) itAg ← vecinos(e.campus, siguiente(it))           ▷ O(1)
54:                    while haySiguiente(itAg) do
55:                        if esAgente(e,siguiente(itAg)) then                                ▷ O(1)
56:                            Sancionar(e,siguiente(itAg))                                   ▷ O(1)
57:                        end if
58:                        avanzar(itAg)
59:                    end while
60:                else
61:                    if esHippie(e,siguiente(it) ∧ esCapturable(e,siguiente(it)) then       ▷ O(1)
62:                        capturarHippie(e,siguiente(it))                                    ▷ O(| Nm |)
63:                    end if
64:                end if
65:            end if
```

```
1: function iINGRESARHIPPIE(in/out e: estr in p: posicion in h : nombre: )                                   ▷ O(|Nm|)
2:     definir(e.posCiviles, h,p)                                                                            ▷ O(|Nm|)
3:     var itDicc(nombre,posicion) iterPos ← definirRapido(e.posRapida,h,p)                                  ▷ O(1)
4:     e.quienOcupa[p.X] [p.Y] ← < true, "hippie", crearIt(), agregarRapido(e.hippies,<h,iterPos>)>          ▷ O(1)
5:     var conj(posicion) Ps ← vecinos(e.campus,p)                                                           ▷ O(1)
6:     var itConj(posicion) ← crearIt(ps)                                                                    ▷ O(1)
7:     if esCapturable(e,p) then                                                                             ▷ O(1)
8:         capturarHippie(e,p)                                                                               ▷ O(|Nm|)
9:     else
10:        while haySiguiente(it) do                                                                         ▷ O(1)
11:            if ocupada(e.campus, siguiente(it)) ∨ ¬e.quienOcupa[siguiente(it).X] [siguiente(it).Y].ocupada? then  ▷ O(1)
12:                avanzar(it)                                                                               ▷ O(1)
13:            else
14:                if esEstudiante(e,siguiente(it)) ∧ esHippizable(e,siguiente(it)) then                     ▷ O(1)
15:                    Hippizar(e, siguiente(it))                                                            ▷ O(1)
16:                    if esCapturable(e,siguiente(it) then                                                  ▷ O(1)
17:                        capturarHippie(e,siguiente(it))                                                   ▷ O(|Nm|)
18:                    end if
19:                else
20:                    if esEstudiante(e,siguiente(it)) ∧ esCapturable(e,siguiente(it)) then                 ▷ O(1)
21:                        var itConj(posicion) itAg ← vecinos(e.campus, siguiente(it))                      ▷ O(1)
22:                        while haySiguiente(itAg) do                                                       ▷ O(1)
23:                            if esAgente(siguiente(itAg) then                                              ▷ O(1)
24:                                sancionar(e,siguiente(itAg))                                              ▷ O(1)
25:                            end if
26:                            avanzar(itAg)
27:                        end while
28:                    end if
29:                end if
30:            end if
31:            avanzar(it)
32:        end while
33:    end if
34: end function
```

```
1: function iMOVERESTUDIANTE(in/out e: estr, in d: direccion, in s: estudiante)                ▷ O(|Nm|)
2:     var posicion actual ← obtener(e.posCiviles,s)                                          ▷ O(1)
3:     var posicion prx ← proxPosicion(e.campus, d, actual)                                   ▷ O(1)
4:     if seFue?(e.campus,actual, prx) then
5:         borrar(e.posCiviles, s)                                                            ▷ O(|Nm|)
6:         var itConj(datosHoE) dat ← copia(e.quienOcupa[actual.X] [actual.Y].hayHoE)         ▷ O(1)
7:         eliminarSiguiente(dat, posActual)                                                  ▷ O(1)
8:         eliminarSiguiente(dat)                                                             ▷ O(1)
9:         e.quienOcupa[actual.X] [actual.Y] ← < false, "nada", crearIt(), crearIt() >        ▷ O(1)
10:    else
11:        var itConj(datosHoE) iterAHOI ← copia(e.quienOcupa[actual.X] [actual.Y].hayHoE)    ▷ O(1)
12:        eliminarSiguiente(siguiente(iterAHOI).posActual)                                   ▷ O(1)
13:        siguiente(iterAHOI).posActual ← definirRapido(e.posRapida,s,prx)                   ▷ O(1)
14:        e.quienOcupa[prx.X] [prx.Y] ← <true, "estudiante", crearIt(), iterAHOI>            ▷ O(1)
15:        e.quienOcupa[actual.X] [actual.Y] ← <false, "nada", crearIt(), crearIt()>          ▷ O(1)
16:        definir(e.posCiviles, s, prx) O(|Nm|)
17:        var conj(posicion) vc ← vecinos(e.campus, prx)                                     ▷ O(1)
18:        var itConj(posicion) it ← crearIt(vc)                                              ▷ O(1)
19:        if esHippizable(e,prx) then                                                        ▷ O(1)
20:            hippizar(e, prx)                                                               ▷ O(1)
21:            while haySiguiente(it do                                                       ▷ O(1)
22:                if ocupada(e.campus, siguiente(it)) ∨ ¬e.quienOcupa[siguiente(it).X] [siguiente(it).Y].ocupada? then
23:                    avanzar(it)
24:                else
25:                    if esEstudiante(e,siguiente(it)) ∧ esHippizable(e,siguiente(it)) then  ▷ O(1)
26:                        Hippizar(e, siguiente(it))                                         ▷ O(1)
27:                        if esCapturable(e, siguiente(it)) then                             ▷ O(1)
28:                            capturarHippie(e,siguiente(it))                                ▷ O(|Nm|)
29:                        end if
30:                    else
31:                        if esEstudiante(e,siguiente(it) ∧ esCapturable(e,siguiente(it) then ▷ O(1)
32:                            var itConj(posicion) itAg ← vecinos(e.campus, siguiente(it))   ▷ O(1)
33:                            while haySiguiente(itAg) do                                    ▷ O(1)
34:                                if esAgente(e,siguiente(itAg) then                         ▷ O(1)
35:                                    sancionar(e,siguiente(itAg))                           ▷ O(1)
36:                                end if
37:                                avanzar(itAg)
38:                            end while
39:                        else
40:                            if esHippie(e,siguiente(it)) ∧ esCapturable(s,siguiente(it)) then ▷ O(1)
41:                                capturarHippie(e,siguiente(it))                            ▷ O(|Nm|)
42:                            end if
43:                        end if
44:                    end if
45:                end if
46:            end while
47:        else
48:            while haySiguiente(it) do
49:                if ocupada(e.campus, siguiente(it))∨ ¬e.quienOcupa[siguiente(it).X] [siguiente(it).Y].ocupada? then ▷
   O(1)
50:                    avanzar(it)
51:                else
52:                    if esHippie(e, siguiente(it)) ∧ esEstudiantizable(e,siguiente(it)) then ▷ O(1)
53:                        Estudiantizar(e, siguiente(it))                                    ▷ O(1)
54:                    else
55:                        if esEstudiante(e,siguiente(it)) ∧ esCapturable(e, siguiente(it)) then ▷ O(1)
56:                            var itConj(posicion) itAg2 ← vecinos(e.campus, siguiente(it))  ▷ O(1)
57:                            while haySiguiente(itAg2) do                                   ▷ O(1)
58:                                if esAgente(e,siguiente(itAg2)) then                       ▷ O(1)
59:                                    sancionar(e,siguiente(itAg2))                          ▷ O(1)
60:                                end if
61:                                avanzar(itAg2)
62:                            end while
63:                        else
64:                            if esHippie(e, siguiente(it)) ∧ esCapturable(e,siguiente(it)) then ▷ O(1)
```

```
1: function iMOVERAGENTE(in/out e: estr in a: agente)                    ▷ O(|Nm|) + O(logNa) + O(Ne)
2:     var nat j ← BusquedaBin(e.AgentesLog,a)                           ▷ O(logNa)
3:     var posicion actual ← e.AgentesLog[j]                             ▷ O(1)
4:     var direccion d ← proxPosicionA(e,a)                              ▷ O(Ne)
5:     var posicion prx ← proxPosicion(e.campus, d, actual)              ▷ O(1)
6:     var datosAg datAux ← obtener(e.agentes, a)                        ▷ O(1)
7:     datAux.posActual ← prx                                            ▷ O(1)
8:     var itDicc(placa,datosAg) itA ← copia(e.quienOcupa[actual.X] [actual.Y].hayCana)   ▷ O(1)
9:     e.quienOcupa[actual.X] [actual.Y] ← <false, "nadie", crearIt(), crearIt()>         ▷ O(1)
10:    e.quienOcupa[prx.X] [prx.Y] ← <true, "agente", itA, crearIt()>    ▷ O(1)
11:    var itConj(posicion) ← crearIt(vecinos(e.campus, prx))            ▷ O(1)
12:    while haySiguiente(it) do                                         ▷ O(1)
13:        if ocupada(e.campus, siguiente(it)) ∨ ¬e.quienOcupa[siguiente(it).X] [siguiente(it).Y].ocupada? then
14:            avanzar(it)
15:        else
16:            if esEstudiante(e, siguiente(it)) ∧ esCapturable(e, siguiente(it)) then   ▷ O(1)
17:                var itConj(posicion) itAg ← vecinos(e.campus, siguiente(it))          ▷ O(1)
18:                while haySiguiente(itAg) do                           ▷ O(1)
19:                    if esAgente(e,siguiente(itAg)) then               ▷ O(1)
20:                        sancionar(e, siguiente(itAg))                 ▷ O(1)
21:                    end if
22:                    avanzar(itAg)
23:                end while
24:            else
25:                if esHippie(e,siguiente(it)) ∧ esCapturable(e,siguiente(it)) then   ▷ O(1)
26:                    capturarHippie(e, siguiente(it))                  ▷ O(|Nm|)
27:                end if
28:            end if
29:        end if
30:        avanzar(it)
31:    end while
32: end function=0
```

```
1: function iMOVERHIPPIE(in/out e: estr in h: nombre)                                   ▷ O(|Nm|) + O(Ne)
2:     var posicion actual ← obtener(e.posCiviles, h)                                   ▷ O(|Nm|)
3:     var direccion d ← proxPosicionH(e,h)                                             ▷ O(Ne)
4:     var posicion prx ← proxPosicion(e.campus, d, obtener(e.posCiviles,h))            ▷ 1
5:     definir(e.posCiviles, h, prx)                                                    ▷ O(|Nm|)
6:     var itConj(nombre) itR ← e.quienOcupa[actual.X] [actual.Y].hayHoe                ▷ O(1)
7:     eliminarSiguiente(siguiente(itR).posActual)                                      ▷ O(1)
8:     siguiente(itR).posActual ← definirRapido(e.posRapida, h, prx)                    ▷ O(1)
9:     e.quienOcupa[prx.X] [prx.Y] ← <true, "hippie", crearIt(), itR>                   ▷ O(1)
10:    e.quienOcupa[actual.X] [actual.Y] ← <false, "nadie", crearIt(), crearIt()>       ▷ O(1)
11:    while haySiguiente(it) do
12:        if ocupada(e.campus, siguiente(it)) ∨ ¬e.quienOcupa[siguiente(it).X] [siguiente(it).Y].ocupada? then   ▷ O(1)
13:            avanzar(it)
14:        else
15:            if esEstudiante(e,siguiente(it)) ∧ esHippizable(e, siguiente(it)) then   ▷ O(1)
16:                hippizar(e, siguiente(it))                                           ▷ O(1)
17:                if esCapturable(e,siguiente(it) then
18:                    capturarHippie(e,siguiente(it))                                  ▷ O(|Nm|)
19:                end if
20:            else
21:                if esEstudiante(e, siguiente(it)) ∧ esCapturable(e, siguiente(it)) then   ▷ O(1)
22:                    var itConj(posicion) itAg ← vecinos(e.campus, siguiente(it))     ▷ O(1)
23:                    while haySiguiente(itAg) do
24:                        if esAgente(e,siguiente(itAg)) then
25:                            sancionar(e, siguiente(itAg))
26:                        end if
27:                        avanzar(itAg)
28:                    end while
29:                else
30:                    if esHippie(e,siguiente(it)) ∧ esCapturable(e,siguiente(it)) then   ▷ O(1)
31:                        capturarHippie(e,siguiente(it))                              ▷ O(|Nm|)
32:                    end if
33:                end if
34:            end if
35:        end if
36:        avanzar(it)
37:    end while
38: end function
```

```
1: function iESESTUDIANTE(in e: estr in p: posicion)⟶ res : bool                        ▷ O(1)
2:     res ← e.quienOcupa[p.X] [p.Y].queHay == "estudiante"
3: end function
```

```
1: function iESHIPPIE(in e: estr in p: posicion)⟶ res : bool                            ▷ O(1)
2:     res ← e.quienOcupa[p.X] [p.Y].queHay == "hippie"
3: end function
```

```
1: function iESAGENTE(in e: estr in p: posicion)⟶ res : bool                            ▷ O(1)
2:     res ← e.quienOcupa[p.X] [p.Y].queHay == "agente"
3: end function
```

```
1: function iEstudiantizar(in/out e: estr in p: posicion)                                   ▷ O(1)
2:     var datosHoE dat ← <Siguiente(e.quienOcupa[p.X] [p.Y].hayHoE).ID, Siguiente(e.quienOcupa[p.X]
   [p.Y].hayHoe).posActual>                                                                 ▷ O(1)
3:     eliminarSiguiente(e.quienOcupa[p.X] [p.Y].hayHoe)                                    ▷ O(1)
4:     var itConj(nombre) it ← agregarRapido(e.estudiantes, dat)                            ▷ O(1)
5:     e.quienOcupa[p.X] [p.Y] ← <true, "estudiante", crearIt(), it>                        ▷ O(1)
6: end function
```

```
1: function iHippizar(in/out e: estr in p: posicion)                                       ▷ O(1)
2:     var datosHoE dat ← <Siguiente(e.quienOcupa[p.X] [p.Y].hayHoE).ID, Siguiente(e.quienOcupa[p.X]
   [p.Y].hayHoe).posActual>                                                                 ▷ O(1)
3:     eliminarSiguiente(e.quienOcupa[p.X] [p.Y].hayHoe)                                    ▷ O(1)
4:     var itConj(nombre) it ← agregarRapido(e.hippies, dat)                                ▷ O(1)
5:     e.quienOcupa[p.X] [p.Y] ← <true, "hippie", crearIt(), it>                            ▷ O(1)
6: end function
```

```
1: function iesCapturable(in e: estr in p: posicion)⟶ res : bool                           ▷ O(1)
2:     var itConj(posicion) it ← crearIt(vecinos(e.campus, p))                              ▷ O(1)
3:     var nat Contador ← 0                                                                 ▷ O(1)
4:     bool hayGuardia ← false                                                              ▷ O(1)
5:     while haySiguiente(it) do
6:         if e.quienOcupa[p.X] [p.Y].ocupada? then                                         ▷ O(1)
7:             contador+ +                                                                  ▷ O(1)
8:         end if
9:         if e.quienOcupa[siguiente(it).X] [siguiente(it).Y].quienOcupa== "agente" then    ▷ O(1)
10:            hayGuardia ← true                                                            ▷ O(1)
11:            avanzar(it)
12:
13:            res ← contador == 4 ∧ hayGuardia                                             ▷ O(1)
14:
```

```
1: function iesHippizable(in/out e: estr in p: posicion)⟶ res : bool                       ▷ O(1)
2:     var itConj(posicion) it ← crearIt(vecinos(e.campus, p))                              ▷ O(1)
3:     var nat Contador ← 0                                                                 ▷ O(1)
4:     while haySiguiente(it) do                                                            ▷ O(1)
5:         if e.quienOcupa[siguiente(it).X] [siguiente(it).Y].quienOcupa == "hippie" then   ▷ O(1)
6:             contador + +                                                                 ▷ O(1)
7:         end if
8:         avanzar(it)
9:     end while
10:    res ← contador ⩾ 2                                                                   ▷ O(1)
11: end function=0
```

```
1: function iCAPTURARHIPPIE(in/out e: estr in p: posicion)                                  ▷ O(|Nm|)
2:     var nombre n ← siguiente(e.quienOcupa[p.X] [p.Y].hayHoE).ID                          ▷ O(1)
3:     eliminarSiguiente(siguiente(e.quienOcupa[p.X] [p.Y].hayHoE).posActual)               ▷ O(1)
4:     eliminarSiguiente(siguiente(e.quienOcupa[p.X] [p.Y].hayHoE)                          ▷ O(1)
5:     borrar(n, e.posCiviles)                                                              ▷ O(|Nm|)
6:     e.quienOcupa[p.X] [p.Y] ← <false, nadie, crearIt(), crearIt() >                      ▷ O(1)
7:     while haySiguiente(it) do                                                            ▷ O(1)
8:         if e.quienOcupa[siguiente(it).X] [siguiente(it).Y].quienOcupa == "agente" then   ▷ O(1)
9:             recompensar(e, siguiente(it))                                                ▷ O(1)
10:        end if
11:        avanzar(it)
12:    end while
13: end function
```

```
1: function iTODASOCUPADAS(in e: estr, in p: conj(posicion)) res:bool                       ▷ O(1)
2:     var itConj(posicion) it ← crearIt(p)                                                 ▷ O(1)
3:     var contador ← 0                                                                     ▷ O(1)
4:     while haySiguiente(it) do
5:         if e.quienOcupa[siguiente(it).X] [siguiente(it).Y].ocupada? then                 ▷ O(1)
6:             contador + +                                                                 ▷ O(1)
7:         end if
8:     end while
9:     res ← contador == 4
10: end function
```

```
function iRECOMPENSAR(in/out e: estr, in a: posicion)                                       ▷ O(1)
    var placa p ← siguienteClave(quienOcupa[a.X] [a.Y].hayCana                              ▷ O(1)
    var datosAgente dat ← obtener(e.agentes,p)                                              ▷ O(1)
    dat.premios ← dat.premios+1                                                             ▷ O(1)
    if dat.premios > obtener(e.agentes, siguienteClave(e.masVigilante)).premios then        ▷ O(1)
        e.masVigilante ← quienOcupa[a.X] [a.Y].hayCana                                      ▷ O(1)
    else
        if dat.premios == obtener(e.agentes, siuienteClave(e.masVigilante)).premios then    ▷ O(1)
            if p < siguienteClave(e.masVigilante)) then                                     ▷ O(1)
                e.masVigilante ← quienOcupa[a.X] [a.Y].hayCana                              ▷ O(1)
            end if
        end if
    end if
end function
```

```
function iSANCIONAR(in/out e: estr, in a: posicion)                                         ▷ O(1)
    var placa p ← siguienteClave(quienOcupa[a.X] [a.Y].hayCana                              ▷ O(1)
    var datosAgente dat ← obtener(e.agentes,p)                                              ▷ O(1)
    dat.Qsanciones ← dat.Qsanciones+1                                                       ▷ O(1)
    eliminarSiguiente(dat.grupoSanciones)                                                   ▷ O(1)
    avanzar(dat.verK)                                                                       ▷ O(1)
    e.hayNuevas ← true                                                                      ▷ O(1)
    if Siguiente(dat.verK).K == dat.Qsanciones then                                         ▷ O(1)
        dat.grupoSanciones ← Agregar(siguiente(dat.verK).grupoK, p)                         ▷ O(1)
    else
        dat.grupoSanciones ← AgregarComoAnterior(dat.verK, <dat.Qsanciones,Agregar(Vacio(),p)>)  ▷ O(1)
    end if
end function
```

```
1: function iProxPosicionH(in/out e: estr in h: nombre)                                  ▷ O(N_e)
2:     var itConj(datosHoE) it← crearIt(e.estudiantes)                                   ▷ O(1)
3:     var posicion menorD ← obtener(e.posRapida,h)                                      ▷ O(N_e)
4:     var direccion direcc                                                              ▷ O(1)
5:     if (¬ haySiguiente(it)) then                                                      ▷ O(1)
6:         if (menorD.Y ≤ e.campus.filas/2) then                                         ▷ O(1)
7:             if (¬ ocupadaD(e,p,abajo)) then                                           ▷ O(1)
8:                 res←Abajo
9:             else                                                                      ▷ O(1)
10:                if (¬ ocupadaD(e,p,derecha)) then                                     ▷ O(1)
11:                    res←Derecha
12:                else                                                                  ▷ O(1)
13:                    if (¬ ocupadaD(e,p,izquierda)) then                               ▷ O(1)
14:                        res←izquierda
15:                    else                                                              ▷ O(1)
16:                        res←arriba                                                    ▷ O(1)
17:                    end if
18:                end if
19:            end if
20:        else
21:            if (¬ ocupadaD(e,p,Arriba)) then                                          ▷ O(1)
22:                res←Arriba
23:            else                                                                      ▷ O(1)
24:                if (¬ ocupadaD(e,p,derecha)) then                                     ▷ O(1)
25:                    res←Derecha
26:                else                                                                  ▷ O(1)
27:                    if (¬ ocupadaD(e,p,izquierda)) then                               ▷ O(1)
28:                        res←izquierda
29:                    else                                                              ▷ O(1)
30:                        res←Abajo                                                     ▷ O(1)
31:                    end if
32:                end if
33:            end if
34:        end if
35:    else
36:        menorD←SiguienteSignificado(siguiente(it).posActual)                          ▷ O(1)
37:        var posicion otraPos                                                          ▷ O(1)
38:        while haySiguiente(it) do                                                     ▷ O(N_e)
39:            otraPos←SiguienteSignificado(siguiente(it).posActual)                     ▷ O(1)
40:            if (distancia(e,p,otraPos) < distancia(e,p,menorD)) then                  ▷ O(1)
41:                menorD←otraPos                                                        ▷ O(1)
42:            end if
43:        end while
44:        res←VecinoMasCercanoA(e,p,menorD)                                             ▷ O(1)
45:    end if
46: end function
```

```
1: function iProxPosicionA(in/out e: estr in a: placa)                                  ▷ O(N_h)
2:     var itConj(datosHoe) it ← crearIt(e.hippies)                                     ▷ O(1)
3:     var posicion menorD ← obtener(e.posRapida,a)                                     ▷ O(N_h)
4:     var direccion direcc                                                             ▷ O(1)
5:     if (¬ haySiguiente(it)) then                                                     ▷ O(1)
6:         if (menorD.Y ≤ e.campus.filas/2) then                                        ▷ O(1)
7:             if (¬ ocupadaD(e,p,abajo)) then                                          ▷ O(1)
8:                 res ← Abajo
9:             else                                                                     ▷ O(1)
10:                if (¬ ocupadaD(e,p,derecha)) then                                    ▷ O(1)
11:                    res ← Derecha
12:                else                                                                 ▷ O(1)
13:                    if (¬ ocupadaD(e,p,izquierda)) then                              ▷ O(1)
14:                        res ← izquierda
15:                    else                                                             ▷ O(1)
16:                        res ← arriba                                                 ▷ O(1)
17:                    end if
18:                end if
19:            end if
20:        else
21:            if (¬ ocupadaD(e,p,Arriba)) then                                         ▷ O(1)
22:                res ← Arriba
23:            else                                                                     ▷ O(1)
24:                if (¬ ocupadaD(e,p,derecha)) then                                    ▷ O(1)
25:                    res ← Derecha
26:                else                                                                 ▷ O(1)
27:                    if (¬ ocupadaD(e,p,izquierda)) then                              ▷ O(1)
28:                        res ← izquierda
29:                    else                                                             ▷ O(1)
30:                        res ← Abajo                                                  ▷ O(1)
31:                    end if
32:                end if
33:            end if
34:        end if
35:    else
36:        menorD ← siguiente(it)                                                       ▷ O(1)
37:        var posicion otraPos                                                         ▷ O(1)
38:        while haySiguiente(it) do                                                    ▷ O(N_h)
39:            otraPos ← SiguienteSignificado(siguiente(it).posActual)                  ▷ O(1)
40:            if (distancia(e,p,otraPos)<distancia(e,p,menorD)) then                   ▷ O(1)
41:                menorD ← otraPos                                                     ▷ O(1)
42:            end if
43:        end while
44:        res ← VecinoMasCercanoA(e,p,menorD)                                          ▷ O(1)
45:    end if
46: end function
```

```
1: function iVecinoMasCercano(in e: estr in p: posicion in p2: posicion ) ⟶ res : direccion        ▷ O(1)
2:     var conj(posicion) Ps ← vecinos(e.campus, p)                                               ▷ O(1)
3:     var itConj(posicion) it ← crearIt(Ps)                                                      ▷ O(1)
4:     var posicion destino← siguiente(it)                                                        ▷ O(1)
5:     while haySiguiente(it) do                                                                  ▷ O(1)
6:         if (¬e.quienOcupa[siguiente(it).X][siguiente(it).Y].ocupada?) then                     ▷ O(1)
7:             if (distancia(e,p,siguiente(it))<distancia(e,p,destino)) then                      ▷ O(1)
8:                 destino← siguiente(it)                                                         ▷ O(1)
9:             end if
10:        end if
11:        avanzar(it)                                                                            ▷ O(1)
12:    end while
13:    if (destino.X≠p.X) then                                                                    ▷ O(1)
14:        if (destino.y>p.Y) then                                                                ▷ O(1)
15:            res← Arriba
16:        else                                                                                   ▷ O(1)
17:            res← Abajo                                                                         ▷ O(1)
18:        end if
19:    else
20:        if (destino.x>p.x) then                                                                ▷ O(1)
21:            res← Derecha
22:        else                                                                                   ▷ O(1)
23:            res← Izquierda                                                                     ▷ O(1)
24:        end if
25:    end if
26: end function
```

```
1: function iSeFue(in e: estr in p: posicion in destino: posicion ) ⟶ res : bool                 ▷ O(1)
2:     res ← (p.Y==e.campus.alto −1 ∧ destino.y==e.campus.alto) ∨(p.Y==0  destino.y== −1)         ▷ O(1)
3: end function
```

```
1: function iBusquedaBin(in v: vector(datosK) in obj: nat) ⟶ res : nat                  ▷ O(log_2(longitud(v)))
2:     var int i← 0                                                                              ▷ O(1)
3:     var int d← longitud(v)-1                                                                  ▷ O(1)
4:     while (i +1 < d) do                                                                       ▷ O(log_2(longitud(v)))
5:         var int m← (i+d)/2                                                                    ▷ O(1)
6:         if (v[m].k<obj) then                                                                  ▷ O(1)
7:             i← m
8:         else                                                                                  ▷ O(1)
9:             d← m                                                                              ▷ O(1)
10:        end if
11:    end while
12:    if (v[i].k==obj) then                                                                     ▷ O(1)
13:        res← i
14:    else                                                                                      ▷ O(1)
15:        res← d                                                                                ▷ O(1)
16:    end if
17: end function
```

```
1: function iOCUPADAD(in e: rastr in p: posicion in dir: direccion) ⟶ res : bool          ▷ O(1)
2:     if dir== " Arriba" then                                                          ▷ O(1)
3:         res←e.quienOcupa[p.X][p.Y +1].ocupada?
4:     else                                                                             ▷ O(1)
5:         if dir== " Abajo" then                                                       ▷ O(1)
6:             res←e.quienOcupa[p.X][p.Y −1].ocupada?
7:         else                                                                         ▷ O(1)
8:             if dir== " izquierda" then                                               ▷ O(1)
9:                 res←e.quienOcupa[p.X−1][p.Y].ocupada?
10:            else                                                                     ▷ O(1)
11:                res←e.quienOcupa[p.X+1][p.Y].ocupada?                                ▷ O(1)
12:            end if
13:        end if
14:    end if
15: end function
```

3. Diseño del Tipo DiccionarioString(σ)

3.1. Especificación

Se usa el Tad Diccionario(κ, σ) especificado en el apunte de Tads básicos.

3.2. Aspectos de la interfaz

3.2.1. Interfaz

parámetros formales

género κ, σ

función $\bullet = \bullet$(**in** a_1: κ, **in** a_2: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} (a_1 = a_2)$ }
Complejidad: $\Theta(equals(a_1, a_2))$
Descripción: función de igualdad de κ's

función **COPIAR**(**in** k: κ) $\longrightarrow res : \kappa$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} k$ }
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

función **COPIAR**(**in** s: σ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} s$ }
Complejidad: $\Theta(copy(s))$
Descripción: función de copia de σ's

Se explica con especificación de Diccionario(κ, σ), Iterador Bidireccional(Tupla(κ, σ))

Género `diccString(`κ`,`σ`)`

Operaciones básicas de diccionario

Definido?(**in** d: $diccString(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} def?(d, k)$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*
Descripción: Devuelve true si y sólo si k está definido en el diccionario.

Obtener(**in** d: $diccString(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $def?(d, k)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} obtener(d, k))$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*
Descripción: Devuelve el significado de la clave k en d.
Aliasing: res no es modificable.

Vacio() $\longrightarrow res : diccString(\kappa, \sigma)$
Pre $\equiv$ { $true$ }

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

Complejidad: $\mathcal{O}(1)$

Descripción: Genera un diccionario vacío.

DEFINIR(**in/out** d: $diccString(\kappa, \sigma)$, **in** k: κ, **in** s: σ)

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

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

Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*

Descripción: Define la clave k con el significado s en el diccionario.

BORRAR(**in/out** d: $diccString(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : bool$

Pre $\equiv$ { $d =_{\text{d}_0} \wedge def?(k, d)$ }

Post $\equiv$ { $d =_{\text{obs}} borrar(k, \text{d}_0)$}

Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*

Descripción: Elimina la clave k del diccionario.

Operaciones básicas del iterador

CREARIT(**in** d: $diccString(\kappa, \sigma)$) $\longrightarrow res : itdiccString(\kappa, \sigma)$

Pre $\equiv$ { $true$ }

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

Complejidad: $\mathcal{O}(n)$ n *es la cantidad de claves.*

Descripción: Crea un iterador del diccionario de forma tal que se puedan recorrer sus elementos aplicando iterativamente SIGUIENTE(no ponemos la operacion SIGUIENTE en la interfaz pues no la usamos).

HAYSIGUIENTE(**in** it: $itdiccString(\kappa, \sigma)$) $\longrightarrow res : bool$

Pre $\equiv$ { $true$ }

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

Complejidad: $\mathcal{O}(1)$

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

SIGUIENTESIGNIFICADO(**in** it: $itdiccString(\kappa, \sigma)$) $\longrightarrow res : \sigma$

Pre $\equiv$ { $HaySiguiente?(it)$ }

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

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el significado del elemento siguiente del iterador.

Aliasing: res no es modificable.

AVANZAR(**in/out** it: $itdiccString(\kappa, \sigma)$)

Pre $\equiv$ { $it =_{\text{obs}} it_0 \wedge$ HaySiguiente?(it)}

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

Complejidad: $\mathcal{O}(1)$

Descripción: Avanza a la posicion siguiente del iterador.

3.3. Pautas de implementación

3.3.1. Estructura de representación

$diccString(\kappa, \sigma)$ **se representa con** $puntero(nodo)$

donde *nodo* **es**
tupla(
significado: ***Puntero(σ)*** ×
caracteres: ***arreglo[256]*** *de puntero(nodo)* ×
padre: ***Puntero(nodo)***
)

3.3.2. Justificación

3.3.3. Invariante de Representación

Informal

- Todas las posiciones del arreglo de caracteres están definidas.
- No hay claves de 0 caracteres. El significado de la raíz es NULL.
- No hay ciclos en la estructura. Es decir, existe una cota superior sobre la cantidad de niveles posibles del árbol.
- Dado un nodo cualquiera del trie, existe un único camino desde la raíz hasta el nodo.

Formal

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ ($true \iff$
(1)($\forall i : nat$)($i < 256 \Rightarrow$ definido?(e $\rightarrow$ caracteres,i)) $\wedge_{\text{L}}$
(2)(e $\rightarrow$ $significado = NULL$) $\wedge_{\text{L}}$
(2)($\exists$ n:nat)(finaliza(e,n)) $\wedge_{\text{L}}$
(3)($\forall$ p,q: puntero(nodo))(p $\in$ punteros(e) $\wedge q \in (punteros(e) - \{p\}) \Rightarrow$ p$\neq$ q) $\wedge_{\text{L}}$
)

3.3.4. Función de Abstracción

Abs : roseTree(estrDato) r $\longrightarrow$ dicc_trie(σ) {Rep(r)}
($\forall$ r:roseTree(estrDato)) Abs(r) $=_{\text{obs}}$ d : dicc_trie(σ) /
($\forall$ k : secu($letra$))(def?(k, d) $=_{\text{obs}}$ esta?(k, r)) $\wedge$ (def?(c, d) $\Rightarrow$ (obtener(k, d) $=_{\text{obs}}$ buscar(k, r)))

Funciones Auxiliares

3.3.5. Algoritmos

```
1: function iVacio( )⟶ res : estr                                          ▷ O(1)
2:     var arreglo(puntero(nodo)) letras ← crearArreglo[256]
3:     for i ← 0 to 255 do                                                 ▷ O(1)
4:     letras[i] ← NULL                                                    ▷ O(1)
5:     end for
6:     var nodo nuevo ← <NULL,letras,NULL>                                 ▷ O(1)
7:     res ← &nuevo                                                        ▷ O(1)
8: end function
```

```
1: function iDefinir(in/out d: estr, in k: string, in s: σ)
2:     nat i ← 0                                                           ▷ O(1)
3:     puntero(nodo) actual ← d                                            ▷ O(1)
4:     while (i < |k|) do                                                  ▷ O(|k|)
5:         if actual ⟶ caracteres[ord(k[i])] = NULL then                   ▷ O(1)
6:             puntero(nodo) anterior ← actual
7:             actual ⟶ caracteres[ord(k[i])] ← iVacio()                   ▷ O(1)
8:             actual ⟶ padre ← anterior
9:         else
10:            actual ← (actual ⟶ caracteres[ord(k[i])])                   ▷ O(1)
11:        end if
12:        i ← i + 1                                                       ▷ O(1)
13:    end while
14:    actual ⟶ significado ← &copiar(s)                                   ▷ O(1)
15: end function
```

```
1: function iObtener(in d: estr, in k: string)⟶ res : σ
2:     nat i ← 0                                                           ▷ O(1)
3:     puntero actual ← d                                                  ▷ O(1)
4:     while i < |k| do                                                    ▷ O(|k|)
5:         actual ← (actual ⟶ caracteres[ord(k[i])])                       ▷ O(1)
6:         i ← i + 1
7:     end while
8:     res ← *(actual ⟶ significado)
9: end function
```

```
1: function iBorrar(in/out d: estr, in k: string)
2:     puntero(nodo) actual ← d
3:     for i ← 0 to |k|                                                    ▷ O(1)
4:     actual ← (actual⟶caracteres[ord(k[i])])                             ▷ O(1)
5:     end for
6:     (actual⟶significado) ← NULL var puntero(nodo) camino ← NULL
7:     while (actual⟶significado = NULL) or todosNULL(actual⟶caracteres) do ▷ O(|k|)
8:         camino ← actual                                                 ▷ O(1)
9:         actual ← (actual ⟶padre)
10:        delete camino
11:    end while
```

```
1: function IDEFINIDO?(in d: estr, in k: string)⟶ res : bool
2:     nat i ← 0                                                          ▷ O(1)
3:     puntero actual ← d                                                 ▷ O(1)
4:     bool def ← true                                                    ▷ O(1)
5:     while (i < |k| and def) do                                         ▷ O(|k|)
6:         if actual ⟶ caracteres[ord(k[i])] = NULL then                  ▷ O(1)
7:             def ← false                                                ▷ O(1)
8:         else
9:             actual ← actual ⟶ caracteres[ord(k[i])]                    ▷ O(1)
10:            i ← i + 1                                                  ▷ O(1)
11:        end if
12:    end while
13:    res ← def ∧ ¬(actual ⟶ significado(NULL))                          ▷ O(1)
14: end function=0
```

3.4. Servicios Usados

Requerimientos sobre el Tipo

- La función $|\mathbf{x}|$ debe tener complejidad $\mathcal{O}(1)$ en el caso peor.
- La función $|\mathbf{x}|$ debe tener complejidad $\mathcal{O}(1)$ en el caso peor.
- Las operaciones deben realizarse por referencia.
- Debe proveer una operación **Copia** que devuelve una nueva instancia de la secuencia pero que es independiente de la actual, con complejidad $\mathcal{O}(n)$ en el caso peor.
- Debe proveer un **iterador** para avanzar que comienza en el primero elemento de la secuencia.
- Debe proveer un **iterador** para retroceder que comienza en el último elemento de la secuencia.
- Las operaciones **CrearIt, Siguiente, Anterior, TieneSiguiente, TieneAnterior** deben tener complejidad $\mathcal{O}(1)$ en el caso peor.

Donde n es la longitud de la palabra.

4. Diseño del Tipo DiccionarioProm

4.1. Especificación

Se usa el Tad DiccionarioM (Nota al corrector: leer observaciones).

4.2. Aspectos de la interfaz

4.2.1. Interfaz

Se explica con especificación de DiccionarioM(κ, σ)

Género `diccProm(`κ`,`σ`)`

Operaciones básicas de diccionario

Definido?(**in** d: $diccProm(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} def?(d, k)$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de claves.*
Descripción: Devuelve true si y sólo si k está definido en el diccionario.

Obtener(**in** d: $diccProm(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $def?(d, k)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} obtener(d, k))$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de claves.*
Descripción: Devuelve el significado de la clave k en d.
Aliasing: se devuelve una referencia al significado de la clave.

Claves(**in** d: $diccProm(\kappa, \sigma)$) $\longrightarrow res : itConj(\kappa)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} claves(d)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: *Devuelve el conjunto con las claves definidas en d.*

Vacio(**in** *n: nat*) $\longrightarrow res : diccProm(\kappa, \sigma)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} vacio(n)$ }
Complejidad: $\mathcal{O}(n)$
Descripción: *Genera un diccionario vacío, donde n acota superiormente a la cantidad de claves.*

Definir(**in/out** *d: diccProm*(κ, σ), **in** *k:* κ, **in** *s:* σ)
Pre $\equiv$ { $d =_{\text{obs}} d_0$ }
Post $\equiv$ { $d =_{\text{obs}} definir(k, s, d_0)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: *Define la clave k con el significado s en el diccionario.*

4.3. Pautas de implementación

4.3.1. Estructura de representación

$diccProm(\kappa, \sigma)$ **se representa con** $estr$

donde *estr* **es**
tupla(
Cclaves: ***conjLineal***(κ) $\times$
clavesMax: ***nat*** $\times$
tabla: ***arreglo*** *de lista*(*datos*)
)
donde *datos* **es**
tupla(
clave: κ $\times$
significado: σ
)

4.3.2. Justificación

4.3.3. Invariante de Representación

Informal

- *clavesMax es mayor que cero.*
- *La longitud del arreglo es igual a clavesMax.*
- *Todas las posiciones del arreglo estan definidas.*
- *Todos los elementos de Cclaves estan definidos en la tabla y viceversa.*
- *Todas las claves de la tabla estan definidos en Cclaves.*

Formal

$Rep : estr \longrightarrow$ boolean
$(\forall$ e $: estr)$
$Rep($e$) \equiv (true \iff$
$(1)\ e.clavesMax > 0 \wedge_{\text{L}}$
$(2)\ longitud(e.tabla) == e.clavesMax \wedge$
$(3)\ (\forall\ i: nat)(i \leq e.clavesMax \Rightarrow_{\text{L}} definido?(e.tabla,i)) \wedge$
$(3)\ (\forall\ k: \kappa)(k \in e.Cclaves\ \Rightarrow\ (\exists\ j : nat)(estaEn?(e.tabla[j],k))) \wedge$
$(4)\ (\forall\ i: nat)(\forall\ k: \kappa)(i\ \ e.clavesMax \wedge_{\text{L}} estaEn?(e.tabla[i],k)\ \Rightarrow\ k \in e.Ccclaves\ )\ )$

Funciones Auxiliares

$estaEn?\ :\ lista(datos) \times \kappa\ \longrightarrow\ bool$
$estaEn?(l,k)\ \equiv\ (\exists\ i : nat)(i<longitud(l) \Rightarrow_{\text{L}} l[i].clave == k)$

4.3.4. Función de Abstracción

$Abs : estr\ e \longrightarrow DiccionarioProm(\kappa, \sigma)$ $\{Rep(e)\}$
$(\forall\ e{:}estr)\ Abs(e) =_{\text{obs}} d : DiccionarioProm(\kappa, \sigma)\ /$
cla

Funciones Auxiliares

4.3.5. Algoritmos

```
1: function iVacio(in n: nat )⟶ res : estr                          ▷ O(clavesMax)
2:     var arreglo(lista(datos)) tabla ← crearArreglo[n]            ▷ O(clavesMax)
3:     for i ← 0 to n do                                            ▷ O(clavesMax)
4:     tabla[i] ← Vacia()                                           ▷ O(1)
5:     end for
6:     res ← <n,tabla>                                              ▷ O(1)
7: end function
```

```
1: function iDefinir(in/out d: estr, in k: nat, in s: σ)            ▷ O(1)
2:     nat i ← fHash(k, e.clavesMax)                                ▷ O(1)
3:     e.tabla[i] ← AgregarAtras(e.tabla[i],<k,s>)                  ▷ O(1)
4: end function
```

```
1: function iObtener(in d: estr, in k: nat)⟶ res : σ                ▷ O(longitud(tabla[i]))
2:     nat i ← fHash(k, e.clavesMax)                                ▷ O(1)
3:     var itLista(datos) it ← crearIt(tabla[i])
4:     while haySiguiente(it) do
5:         if siguiente(it).clave = k then
6:             res ← siguiente(it).significado
7:         end if
8:     end while
9: end function
```

```
1: function iDefinido?(in d: estr, in k: nat)⟶ res : bool            ▷ O(longitud(tabla[i]))
2:     nat i ← fHash(k, e.clavesMax)                                ▷ O(1)
3:     var itLista(datos) it ← crearIt(tabla[i])
4:     bool aux ← false
5:     while haySiguiente(it) do
6:         if siguiente(it).clave = k then
7:             aux ← true
8:         end if
9:     end while
10:    res ← aux
11: end function
```

```
1: function fHash(in k: nat, in clavesMax: nat)⟶ res : nat           ▷ O(1)
2:     res ← k mod clavesMax                                        ▷ O(1)
3: end function
```

```
1: function iClaves(in d: estr )⟶ res : itConj(κ)                    ▷ O(clavesMax)
2:     res ← crearIt(e.Cclaves)
3: end function
```

4.4. Servicios Usados