



**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 PC ES NAT**
2. **TAD INTERFAZ ES NAT**
3. **TAD PRIORIDAD ES NAT**
4. **TAD PAQUETE ES TUPLA(NAT,PRIORIDAD,PC,PC)**

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*) \rightarrow *res: campus*

Pre $\equiv \{ true \}$

Post $\equiv \{ res =_{\text{obs}} \text{crearCampus}(c, f) \}$

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

Descripción: Crea un campus de c columnas y f filas.

FILAS?(*in c: campus*) \rightarrow *res: nat*

Pre $\equiv \{ true \}$

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

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

Descripción: Devuelve la cantidad de filas en el campus.

COLUMNAS?(*in c: campus*) \rightarrow *res: nat*

Pre $\equiv \{ true \}$

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

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

Descripción: Devuelve la cantidad de columnas en el campus.

OCUPADA?(*in c: campus, in p: posicion*) \rightarrow *res: bool*

Pre $\equiv \{ \text{posValida}(p, c) \}$

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

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

Descripción: Devuelve *true* sii p esta ocupada por un obstaculo.

AGREGAROBSTACULO(*in/out c: campus, in p: posicion*) \rightarrow

Pre $\equiv \{ c =_{\text{obs}} c_0 \wedge \text{posValida}(p, c) \wedge_L \neg \text{ocupada?}(p, c) \}$

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

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

Descripción: Devuelve *true* sii p esta ocupada por un obstaculo.

POSVALIDA?(*in c: campus, in p: posicion*) \rightarrow *res: bool*

Pre $\equiv \{ true \}$

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

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

Descripción: Devuelve *true* sii p es parte del mapa.

ESINGRESO?(*in c: campus, in p: posicion*) \rightarrow *res: bool*

Pre $\equiv \{ true \}$

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

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

Descripción: Devuelve *true* sii p es un ingreso.

VECINOS(**in** *c: campus*, **in** *p: posicion*) $\rightarrow res : conj(posicion)$

Pre $\equiv \{ posValida(p, c) \}$

Post $\equiv \{ res =_{obs} vecinos(p, c) \}$

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

Descripción: Devuelve el conjunto de posiciones vecinas a p.

VECINOSCOMUNES(**in** *c: campus*, **in** *p: posicion*, **in** *p2: posicion*) $\rightarrow res : conj(posicion)$

Pre $\equiv \{ posValida(p, c) \wedge posValida(p2, c) \}$

Post $\equiv \{ res =_{obs} vecinos(p, c) \cap vecinos(p2, c) \}$

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

Descripción: Devuelve el conjunto de vecinos comunes. La complejidad es $\mathcal{O}(1)$ dado que los vecinos son a lo sumo 4, o sea, constantes

VECINOSCOMUNES(**in** *c: campus*, **in** *p: posicion*, **in** *p2: posicion*) $\rightarrow res : conj(posicion)$

Pre $\equiv \{ posValida(p, c) \wedge posValida(p2, c) \}$

Post $\equiv \{ res =_{obs} vecinos(p, c) \cap vecinos(p2, c) \}$

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

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

PROXPOSICION(**in** *c: campus*, **in** *dir: direccion*, **in** *p: posicion*) $\rightarrow res : posicion$

Pre $\equiv \{ posValida(p, c) \}$

Post $\equiv \{ res =_{obs} proxPosicion(p, d, c) \}$

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

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

INGRESOSMASCERCANOS(**in** *c: campus*, **in** *p: posicion*) $\rightarrow res : conj(posicion)$

Pre $\equiv \{ posValida(p, c) \}$

Post $\equiv \{ res =_{obs} ingresosMasCercanos(p, c) \}$

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 ×
  columnas: nat ×
  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

$Rep : estr \rightarrow boolean$

$(\forall e : estr)$

$\text{Rep}(e) \equiv (\text{true} \iff$
 (1) $e.\text{filas} = \text{longitud}(e.\text{mapa}) \wedge_L (\forall i : \text{nat})(i \leq e.\text{filas} \Rightarrow \text{longitud}(e.\text{mapa}[i]) = e.\text{columnas}))$

1.3.4. Función de Abstracción

$\text{Abs} : \text{estr } e \longrightarrow \text{campus}$ $\{\text{Rep}(e)\}$
 $(\forall e : \text{estr}) \text{ Abs}(e) =_{\text{obs}} c : \text{campus} /$
 $\left(\text{filas}(c) = e.\text{filas} \wedge \text{columnas}(c) = e.\text{columnas} \wedge_L (\forall p : \text{posicion})(p.X \leq e.\text{filas} \wedge \right.$
 $\left. p.Y \leq e.\text{columnas} \Rightarrow_L \text{ocupada?}(p, c) \Leftrightarrow (e.\text{mapa}[p.X])[p.Y]) \right)$

1.3.5. Algoritmos

```

1: function iCREARCAMPUS(in c: nat, in f: nat) → res : estr                                ▷  $\mathcal{O}(f^2 * c^2)$ 
2:   var vector(vector(bool)) mapa ← vacia(vacia())                                     ▷  $\mathcal{O}(1)$ 
3:   var nat i ← 0                                                                    ▷  $\mathcal{O}(1)$ 
4:   while i ≤ f do                                                                    ▷  $\mathcal{O}(f)$ 
5:     var vector(bool) nuevo ← vacia()                                              ▷  $\mathcal{O}(1)$ 
6:     var nat j ← 0                                                                    ▷  $\mathcal{O}(1)$ 
7:     while j ≤ c do                                                                    ▷  $\mathcal{O}(c)$ 
8:       AgregarAtras(nuevo, false)                                                  ▷  $\mathcal{O}(c)$ 
9:       j++                                                                            ▷  $\mathcal{O}(1)$ 
10:    end while
11:    AgregarAtras(mapa, nuevo)                                                       ▷  $\mathcal{O}(f)$ 
12:    i++                                                                              ▷  $\mathcal{O}(1)$ 
13:  end while
14:  res ← < f, c, mapa >                                                            ▷  $\mathcal{O}(1)$ 
15: end function

```

```

1: function iAGREGAROBSTACULO(in/out e: estr, in p: posicion) → res : estr          ▷  $\mathcal{O}(\text{longitud}(e.\text{mapa}[p.X]))$ 
2:   Agregar(e.mapa[p.X], p.Y, true)                                                 ▷  $\mathcal{O}(\text{longitud}(e.\text{mapa}[p.X]))$ 
3: end function

```

```

1: function iFILAS?(in e: estr) → res : nat                                           ▷  $\mathcal{O}(1)$ 
2:   res ← e.filas                                                                    ▷  $\mathcal{O}(1)$ 
3: end function

```

```

1: function iCOLUMNAS?(in e: estr) → res : nat                                       ▷  $\mathcal{O}(1)$ 
2:   res ← e.columnas                                                                ▷  $\mathcal{O}(1)$ 
3: end function

```

```

1: function iOCUPADA?(in e: estr, in p: posicion) → res : bool                     ▷  $\mathcal{O}(1)$ 
2:   res ← (e.mapa[p.X])[p.Y]                                                       ▷  $\mathcal{O}(1)$ 
3: end function

```

```

1: function iPOSVALIDA?(in e: estr, in p: posicion) → res : bool                   ▷  $\mathcal{O}(1)$ 
2:   res ← (0 < p.X) ∧ (p.X ≤ e.filas) ∧ (0 < p.Y) ∧ (p.Y ≤ e.columnas)           ▷  $\mathcal{O}(1)$ 
3: end function

```

```

1: function iESINGRESO?(in e: estr, in p: posicion) → res : bool                   ▷  $\mathcal{O}(1)$ 
2:   res ← (p.Y = 1) ∨ (p.Y = e.filas)                                              ▷  $\mathcal{O}(1)$ 
3: end function

```

```

1: function iVECINOS(in e: estr, in p: posicion)  $\rightarrow$  res : conj(posicion)  $\triangleright \mathcal{O}(1)$ 
2:   var conj(posicion) nuevo  $\leftarrow$  vacio()  $\triangleright \mathcal{O}(1)$ 
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  $\leftarrow$  crearIt(nuevo)
8:   while haySiguiente(it) do  $\triangleright \mathcal{O}(c)$ 
9:     if iPosValida?(e,siguiente(it)) then  $\triangleright \mathcal{O}(1)$ 
10:      avanzar(it)  $\triangleright \mathcal{O}(1)$ 
11:     else
12:       eliminarSiguiente(it)  $\triangleright \mathcal{O}(1)$ 
13:     end if
14:   end while
15:   res  $\leftarrow$  nuevo  $\triangleright \mathcal{O}(1)$ 
16: end function

```

```

1: function iVECINOSCOMUNES(in e: estr, in p: posicion, in p2: posicion)  $\rightarrow$  res : conj(posicion)  $\triangleright \mathcal{O}(1)$ 
2:   var conj(posicion) v  $\leftarrow$  vecinos(e,p)  $\triangleright \mathcal{O}(1)$ 
3:   var conj(posicion) v2  $\leftarrow$  vecinos(e,p2)  $\triangleright \mathcal{O}(1)$ 
4:   var conj(posicion) nuevo  $\leftarrow$  vacio()  $\triangleright \mathcal{O}(1)$ 
5:   var itConj(posicion) it  $\leftarrow$  crearIt(v)  $\triangleright \mathcal{O}(1)$ 
6:   while haySiguiente(it) do  $\triangleright \mathcal{O}(1)$ 
7:     if Pertenece?(v2,Siguiente(it)) then  $\triangleright \mathcal{O}(1)$ 
8:       Agregar(nuevo, Siguiente(it))  $\triangleright \mathcal{O}(1)$ 
9:     end if
10:    Avanzar(it)  $\triangleright \mathcal{O}(1)$ 
11:  end while
12:  res  $\leftarrow$  nuevo  $\triangleright \mathcal{O}(1)$ 
13: end function

```

```

1: function iVECINOSVALIDOS(in e: estr, in ps: conj(posicion))  $\rightarrow$  res : conj(posicion)  $\triangleright \mathcal{O}(1)$ 
2:   var conj(posicion) nuevo  $\leftarrow$  vacio()  $\triangleright \mathcal{O}(1)$ 
3:   var itConj(posicion) it  $\leftarrow$  crearIt(ps)  $\triangleright \mathcal{O}(1)$ 
4:   while haySiguiente(it) do  $\triangleright \mathcal{O}(1)$ 
5:     if PosValida?(e,siguiente(it)) then  $\triangleright \mathcal{O}(1)$ 
6:       Agregar(nuevo, siguiente(it))  $\triangleright \mathcal{O}(1)$ 
7:     end if
8:     avanzar(it)  $\triangleright \mathcal{O}(1)$ 
9:   end while
10:  res  $\leftarrow$  nuevo  $\triangleright \mathcal{O}(1)$ 
11: end function

```

```

1: function iDISTANCIA(in e: estr, in p: posicion, in p2: posicion)  $\rightarrow$  res : nat  $\triangleright \mathcal{O}(1)$ 
2:   res  $\leftarrow$  |p.X - p2.X| + |p.Y - p2.Y|  $\triangleright \mathcal{O}(1)$ 
3: end function

```

```

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

```

```

1: function iINGRESOSMASCERCANOS(in e: estr, in p: posicion) → res : conj(posicion)           ▷  $\mathcal{O}(1)$ 
2:   var conj(posicion) nuevo ← Vacio()                             ▷  $\mathcal{O}(1)$ 
3:   if distancia(e, p, <p.x,1>) < distancia(e, p, <p.x,e.filas>) then   ▷  $\mathcal{O}(1)$ 
4:     Agregar(nuevo, <p.x,1>)                                         ▷  $\mathcal{O}(1)$ 
5:   else
6:     if distancia(e, p, <p.x,1>) > distancia(e, p, <p.x,filas(e)>) then   ▷  $\mathcal{O}(1)$ 
7:       Agregar(nuevo, <p.x,e.filas>)                                   ▷  $\mathcal{O}(1)$ 
8:     else
9:       Agregar(nuevo, <p.x,1>)                                         ▷  $\mathcal{O}(1)$ 
10:      Agregar(nuevo, <p.x,e.filas>)                                   ▷  $\mathcal{O}(1)$ 
11:    end if
12:  end if
13:  res ← nuevo                                                         ▷  $\mathcal{O}(1)$ 
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.

POSESTUDIANTEYHIPPIE(*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: $\mathcal{O}(1)$

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.

CANTSANCIONES(**in** $r: rastr$, **in** $a: agente$) $\rightarrow res: nat$

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.

CANTHIPPIESATRAPADOS(**in** $r: rastr$, **in** $a: agente$) $\rightarrow res: nat$

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.

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

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

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** $p: posicion$) \rightarrow

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: $\mathcal{O}(1)$

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

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

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: $\mathcal{O}(1)$

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

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

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: $\mathcal{O}(1)$

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

MOVERHIPPIE(**in/out** $r: rastr$, **in** $h: nombre$) \rightarrow

Pre $\equiv \{ r = r_0 \wedge h \in hippies(r) \wedge \neg todasOcupadas?(vecinos(posEstudianteYHippie(h, r), campus(r)), r) \}$

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

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

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

MOVERAGENTE(**in/out** r : *rastr*, **in** a : *agente*) \rightarrow

Pre $\equiv \{ r = r_0 \wedge a \in \text{agentes}(r) \wedge \text{cantSanciones}(a, r) \leq 3 \wedge \neg \text{todasOcupadas}(\text{vecinos}(\text{posAgente}(a, r), \text{campus}(r)), r) \}$

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

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

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

MASVIGILANTE(**in** r : *rastr*) $\rightarrow res$: *agente*

Pre $\equiv \{ true \}$

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

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

Descripción: Devuelve el agente con mas capturas.

CONKSANCIONES(**in** r : *rastr*, **in** k : *nat*) $\rightarrow res$: *conj(agente)*

Pre $\equiv \{ true \}$

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

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

Descripción: Devuelve el agente con mas capturas.

CONMISMASANCIONES(**in** r : *rastr*, **in** a : *agente*) $\rightarrow res$: *conj(agente)*

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

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

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

Formal

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

$(\forall e : \text{estr})$

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

$(1)(2)(3) (\forall a, a2 : \text{Agente})(\text{def?}(a, e.\text{agentes}) \wedge \text{def?}(a2, e.\text{agentes})) \Rightarrow_L$

$(\text{obtener}(a, e.\text{agentes}).\text{Qsanciones} = \text{siguiente}(\text{obtener}(a, e.\text{agentes}).\text{verK}).\text{K}$

$\wedge \text{obtener}(a, e.\text{agentes}).\text{grupoSanciones} = \text{siguiente}(\text{obtener}(a, e.\text{agentes}).\text{verK}).\text{grupoK}$

$\wedge (a2 \in \text{obtener}(a, e.\text{agentes}).\text{grupoSanciones}) \iff (\text{obtener}(a, e.\text{agentes}).\text{Qsanciones} = \text{obtener}(a2, e.\text{agentes}).\text{Qsanciones})$

$\wedge (4) \text{PosValida}(e, \text{obtener}(a, e.\text{agentes}).\text{PosActual})) \wedge (5) \text{TodasLasPlacas}(e, e.\text{PosAgentesLog}()))$

2.3.4. Función de Abstracción

$\text{Abs} : \text{estr } e \longrightarrow \text{campus}$

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

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

$\left(\text{filas}(c) = e.\text{filas} \wedge \text{columnas}(c) = e.\text{columnas} \wedge_L (\forall p : \text{posicion})(p.X \leq e.\text{filas} \wedge$

$p.Y \leq e.\text{columnas} \Rightarrow_L \text{ocupada?}(p, c) \Leftrightarrow (e.\text{mapa}[f])[c]) \right)$

2.3.5. Algoritmos

```

1: function iCAMPUS(in e: estr)  $\longrightarrow$  res : campus  $\triangleright \mathcal{O}(1)$ 
2:   res  $\leftarrow$  e.campo
3: end function

```

```

1: function iESTUDIANTES(in e: estr)  $\longrightarrow$  res : itConj(nombre)  $\triangleright \mathcal{O}(1)$ 
2:   res  $\leftarrow$  crearIt (e.estudiantes)
3: end function

```

```

1: function iHIPPIES(in e: estr)  $\longrightarrow$  res : itConj(nombre)  $\triangleright \mathcal{O}(1)$ 
2:   res  $\leftarrow$  crearIt (e.hippies)
3: end function

```

```

1: function iAGENTES(in e: estr)  $\longrightarrow$  res : itConj(agente)  $\triangleright \mathcal{O}(1)$ 
2:   res  $\leftarrow$  crearIt (e.agentes)
3: end function

```

```

1: function iPOSESTUDIANTESYHIPPIE(in n: nombre in e: estr)  $\longrightarrow$  res : posicion  $\triangleright \mathcal{O}(n_m)$ 
2:   res  $\leftarrow$  obtener(n,e.posCiviles)
3: end function

```

```

1: function iPOSAGENTE(in a: agente in e: estr)  $\longrightarrow$  res : posicion  $\triangleright \mathcal{O}(1)(promedio)$ 
2:   res  $\leftarrow$  obtener(a,e.agentes).posActual
3: end function

```

```

1: function iCANTSANCIONES(in a: agente in e: estr)  $\longrightarrow$  res : nat  $\triangleright \mathcal{O}(1)(promedio)$ 
2:   res  $\leftarrow$  obtener(a,e.agentes).Qsanciones
3: end function

```

```

1: function iCANTHIPPIESATRAPADOS(in a: agente in e: estr)  $\longrightarrow$  res : nat  $\triangleright \mathcal{O}(1)(promedio)$ 
2:   res  $\leftarrow$  obtener(a,e.agentes).premios
3: end function

```

```

1: function iMASVIGILANTE(in r: rastr)  $\longrightarrow$  res : agente  $\triangleright \mathcal{O}(1)$ 
2:   res  $\leftarrow$  siguiente(e.masVigilante)
3: end function

```

```

1: function iCONMISMASANCIONES(in r: rastr in a: agente)  $\longrightarrow$  res : conj(agente)  $\triangleright \mathcal{O}(1)$ 
2:   res  $\leftarrow$  siguiente(obtener(e.agentes,a).grupoSanciones)
3: end function

```

```

1: function iCONKSANCIONES(in r: rastr in k: nat)  $\rightarrow$  res : conj(agente)     $\triangleright \mathcal{O}(n)$  la primera vez, luego mientras no
   haya sanciones  $\mathcal{O}(\log n)$ 
2:   if  $\neg$ e.huboNuevas then                                                     $\triangleright \mathcal{O}(1)$ 
3:     var nat i  $\leftarrow$  BusquedaBin(e.buscoEnLog, k)                           $\triangleright \mathcal{O}(\log n)$ 
4:     res  $\leftarrow$  e.buscoEnLog[i].grupoK                                        $\triangleright \mathcal{O}(1)$ 
5:   else
6:     var itLista(datosK) itK  $\leftarrow$  crearIt(e.agregoEn1)                     $\triangleright \mathcal{O}(1)$ 
7:     while haySiguiente(it) do                                               $\triangleright \mathcal{O}(1)$ 
8:       buscoEnLog[i]  $\leftarrow$  siguiente(itK)                                   $\triangleright \mathcal{O}(1)$ 
9:       avanzar(itK)
10:    end while
11:    var nat i  $\leftarrow$  BusquedaBin (e.buscoEnLog, k)                           $\triangleright \mathcal{O}(\log n)$ 
12:    res  $\leftarrow$  e.buscoEnLog[i].grupoK                                        $\triangleright \mathcal{O}(1)$ 
13:    e.hayNuevas  $\leftarrow$  false                                               $\triangleright \mathcal{O}(1)$ 
14:  end if
15: end function

```

```

1: function iINGRESARHIPPIE(in/out r: rastr in p: posicion in h: nombre: )  $\triangleright \mathcal{O}(|Nm|)$ 
2:   definir(e.posCiviles, h,p)                                               $\triangleright \mathcal{O}(|Nm|)$ 
3:   var itDicc(nombre,posicion) iterPos  $\leftarrow$  definirRapido(e.posRapida,h,p)  $\triangleright \mathcal{O}(1)$ 
4:   e.quienOcupa[p.X] [p.Y]  $\leftarrow$  < true, "hippie", crearIt(), agregarRapido(e.hippies,<h,iterPos>)>  $\triangleright \mathcal{O}(1)$ 
5:   var conj(posicion) Ps  $\leftarrow$  vecinos(e.campus,p)                          $\triangleright \mathcal{O}(1)$ 
6:   var itConj(posicion)  $\leftarrow$  crearIt(ps)                                    $\triangleright \mathcal{O}(1)$ 
7:   if esCapturable(e,p) then                                               $\triangleright \mathcal{O}(1)$ 
8:     capturarHippie(e,p)                                                   $\triangleright \mathcal{O}(|Nm|)$ 
9:   else
10:    while haySiguiente(it) do                                               $\triangleright \mathcal{O}(1)$ 
11:      if ocupada(e.campus, siguiente(it))  $\vee \neg$ e.quienOcupa[siguiente(it).X] [siguiente(it).Y].ocupada? then  $\triangleright \mathcal{O}(1)$ 
12:        avanzar(it)                                                        $\triangleright \mathcal{O}(1)$ 
13:      else
14:        if esEstudiante(e,siguiente(it))  $\wedge$  esHippizable(e,siguiente(it)) then  $\triangleright \mathcal{O}(1)$ 
15:          Hippizar(e, siguiente(it))                                        $\triangleright \mathcal{O}(1)$ 
16:          if esCapturable(e,siguiente(it)) then                           $\triangleright \mathcal{O}(1)$ 
17:            capturarHippie(e,siguiente(it))                                $\triangleright \mathcal{O}(|Nm|)$ 
18:          end if
19:        else
20:          if esEstudiante(e,siguiente(it))  $\wedge$  esCapturable(e,siguiente(it)) then  $\triangleright \mathcal{O}(1)$ 
21:            var itConj(posicion) itAg  $\leftarrow$  vecinos(e.campus, siguiente(it))  $\triangleright \mathcal{O}(1)$ 
22:            while haySiguiente(itAg) do  $\triangleright \mathcal{O}(1)$ 
23:              if esAgente(siguiente(itAg)) then  $\triangleright \mathcal{O}(1)$ 
24:                sancionar(e,siguiente(itAg))  $\triangleright \mathcal{O}(1)$ 
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 r: rastr, in d: direccion, in s: estudiante) ▷  $\mathcal{O}(|Nm|)$ 
2:   actual ← obtener(e.posCiviles,s) ▷  $\mathcal{O}(1)$ 
3:   prx ← proxPosicion(e.campus, d, actual) ▷  $\mathcal{O}(1)$ 
4:   if seFue?(e.campus,actual, prx) then
5:     borrar(e.posCiviles, s) ▷  $\mathcal{O}(|Nm|)$ 
6:     var itConj(datosHoE) dat ← copia(e.quienOcupa[actual.X] [actual.Y].hayHoE) ▷  $\mathcal{O}(1)$ 
7:     eliminarSiguiente(dat, posActual) ▷  $\mathcal{O}(1)$ 
8:     eliminarSiguiente(dat) ▷  $\mathcal{O}(1)$ 
9:     e.quienOcupa[actual.X] [actual.Y] ← < false, "nada", crearIt(), crearIt() > ▷  $\mathcal{O}(1)$ 
10:  else
11:    var itConj(datosHoE) iterAHOI ← copia(e.quienOcupa[actual.X] [actual.Y].hayHoE) ▷  $\mathcal{O}(1)$ 
12:    eliminarSiguiente(siguiente(iterAHOI).posActual) ▷  $\mathcal{O}(1)$ 
13:    siguiente(iterAHOI).posActual ← definirRapido(e.posRApida,s,prx) ▷  $\mathcal{O}(1)$ 
14:    e.quienOcupa[prx.X] [prx.Y] ← <true, "estudiante", crearIt(), iterAHOI> ▷  $\mathcal{O}(1)$ 
15:    e.quienOcupa[actual.X] [actual.Y] ← <false, "nada", crearIt(), crearIt()> ▷  $\mathcal{O}(1)$ 
16:    definir(e.posCiviles, s, prx)  $\mathcal{O}(|Nm|)$ 
17:    var conj(posicion) vc ← vecinos(e.campus, prx) ▷  $\mathcal{O}(1)$ 
18:    var itConj(posicion) it ← crearIt(vc) ▷  $\mathcal{O}(1)$ 
19:    if esHippizable(e,prx) then ▷  $\mathcal{O}(1)$ 
20:      hippizar(e, prx) ▷  $\mathcal{O}(1)$ 
21:      while haySiguiente(it) do ▷  $\mathcal{O}(1)$ 
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 ▷  $\mathcal{O}(1)$ 
26:            Hippizar(e, siguiente(it)) ▷  $\mathcal{O}(1)$ 
27:            if esCapturable(e, siguiente(it)) then ▷  $\mathcal{O}(1)$ 
28:              capturarHippie(e,siguiente(it)) ▷  $\mathcal{O}(|Nm|)$ 
29:            end if
30:          else
31:            if esEstudiante(e,siguiente(it)) ∧ esCapturable(e,siguiente(it)) then ▷  $\mathcal{O}(1)$ 
32:              var itConj(posicion) itAg ← vecinos(e.campus, siguiente(it)) ▷  $\mathcal{O}(1)$ 
33:              while haySiguiente(itAg) do ▷  $\mathcal{O}(1)$ 
34:                if esAgente(e,siguiente(itAg)) then ▷  $\mathcal{O}(1)$ 
35:                  sancionar(e,siguiente(itAg)) ▷  $\mathcal{O}(1)$ 
36:                end if
37:                avanzar(itAg)
38:              end while
39:            else
40:              if esHippie(e,siguiente(it)) ∧ esCapturable(s,siguiente(it)) then ▷  $\mathcal{O}(1)$ 
41:                capturarHippie(e,siguiente(it)) ▷  $\mathcal{O}(|Nm|)$ 
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 ▷
50:          avanzar(it)
51:        else
52:          if esHippie(e, siguiente(it)) ∧ esEstudiantizable(e,siguiente(it)) then ▷  $\mathcal{O}(1)$ 
53:            Estudiantizar(e, siguiente(it)) ▷  $\mathcal{O}(1)$ 
54:          else
55:            if esEstudiante(e,siguiente(it)) ∧ esCapturable(e, siguiente(it)) then ▷  $\mathcal{O}(1)$ 
56:              var itConj(posicion) itAg2 ← vecinos(e.campus, siguiente(it)) ▷  $\mathcal{O}(1)$ 
57:              while haySiguiente(itAg2) do ▷  $\mathcal{O}(1)$ 
58:                if esAgente(e,siguiente(itAg2)) then ▷  $\mathcal{O}(1)$ 
59:                  sancionar(e,siguiente(itAg2)) ▷  $\mathcal{O}(1)$ 
60:                end if
61:                avanzar(itAg2)
62:              end while 16/??
63:            else
64:              if esHippie(e, siguiente(it)) ∧ esCapturable(e,siguiente(it)) then ▷  $\mathcal{O}(1)$ 

```

```

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

```

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 $\bullet = \bullet(\text{in } a_1: \kappa, \text{in } a_2: \kappa) \rightarrow res: bool$

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($\text{in } k: \kappa) \rightarrow res: \kappa$

Pre $\equiv \{ true \}$

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

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

Descripción: función de copia de κ 's

función COPIAR($\text{in } s: \sigma) \rightarrow res: \sigma$

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?($\text{in } d: \text{diccString}(\kappa, \sigma), \text{in } k: \kappa) \rightarrow res: bool$

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($\text{in } d: \text{diccString}(\kappa, \sigma), \text{in } k: \kappa) \rightarrow res: \sigma$

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() $\rightarrow res: \text{diccString}(\kappa, \sigma)$

Pre $\equiv \{ true \}$

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

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

Descripción: Genera un diccionario vacío.

DEFINIR(**in/out** $d: \text{diccString}(\kappa, \sigma)$, **in** $k: \kappa$, **in** $s: \sigma$)

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

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

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** $d: \text{diccString}(\kappa, \sigma)$, **in** $k: \kappa$) $\rightarrow res: \text{bool}$

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

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

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

Descripción: Elimina la clave k del diccionario.

Operaciones básicas del iterador

CREARIT(**in** $d: \text{diccString}(\kappa, \sigma)$) $\rightarrow res: \text{itdiccString}(\kappa, \sigma)$

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

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

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 operación SIGUIENTE en la interfaz pues no la usamos).

HAYSIGUIENTE(**in** $it: \text{itdiccString}(\kappa, \sigma)$) $\rightarrow res: \text{bool}$

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

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

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

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

SIGUIENTESIGNIFICADO(**in** $it: \text{itdiccString}(\kappa, \sigma)$) $\rightarrow res: \sigma$

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

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

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

Descripción: Devuelve el significado del elemento siguiente del iterador.

Aliasing: res no es modificable.

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

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

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

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

Descripción: Avanza a la posición siguiente del iterador.

3.3. Pautas de implementación

3.3.1. Estructura de representación

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

donde *nodo* es

tupla(
 significado: *Puntero*(σ) \times
 caracteres: *arreglo*[256] de *puntero*(*nodo*) \times
 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

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

$(\forall e : \text{estr})$

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

$(1)(\forall i : \text{nat})(i < 256 \Rightarrow \text{definido?}(e \rightarrow \text{caracteres}, i)) \wedge_L$

$(2)(e \rightarrow \text{significado} = \text{NULL}) \wedge_L$

$(2)(\exists n : \text{nat})(\text{finaliza}(e, n)) \wedge_L$

$(3)(\forall p, q : \text{puntero}(\text{nodo}))(p \in \text{punteros}(e) \wedge q \in (\text{punteros}(e) - \{p\}) \Rightarrow p \neq q) \wedge_L$

)

3.3.4. Función de Abstracción

$\text{Abs} : \text{roseTree}(\text{estrDato}) \ r \longrightarrow \text{dicc_trie}(\sigma)$

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

$(\forall r : \text{roseTree}(\text{estrDato})) \ \text{Abs}(r) =_{\text{obs}} d : \text{dicc_trie}(\sigma) \ /$

$(\forall k : \text{secu}(\text{letra}))(\text{def?}(k, d) =_{\text{obs}} \text{esta?}(k, r)) \wedge (\text{def?}(c, d) \Rightarrow (\text{obtener}(k, d) =_{\text{obs}} \text{buscar}(k, r)))$

Funciones Auxiliares

3.3.5. Algoritmos

```

1: function iVACIO( )  $\rightarrow$  res : estr  $\triangleright \mathcal{O}(1)$ 
2:   var arreglo(puntero(nodo)) letras  $\leftarrow$  crearArreglo[256]
3:   for i  $\leftarrow$  0 to 255 do  $\triangleright \mathcal{O}(1)$ 
4:     letras[i]  $\leftarrow$  NULL  $\triangleright \mathcal{O}(1)$ 
5:   end for
6:   var nodo nuevo  $\leftarrow$  <NULL,letras,NULL>  $\triangleright \mathcal{O}(1)$ 
7:   res  $\leftarrow$  &nuevo  $\triangleright \mathcal{O}(1)$ 
8: end function

```

```

1: function iDEFINIR(in/out d: estr, in k: string, in s:  $\sigma$ )
2:   nat i  $\leftarrow$  0  $\triangleright \mathcal{O}(1)$ 
3:   puntero(nodo) actual  $\leftarrow$  d  $\triangleright \mathcal{O}(1)$ 
4:   while (i < |k|) do  $\triangleright \mathcal{O}(|k|)$ 
5:     if actual  $\rightarrow$  caracteres[ord(k[i])] = NULL then  $\triangleright \mathcal{O}(1)$ 
6:       puntero(nodo) anterior  $\leftarrow$  actual
7:       actual  $\rightarrow$  caracteres[ord(k[i])]  $\leftarrow$  iVacio()  $\triangleright \mathcal{O}(1)$ 
8:       actual  $\rightarrow$  padre  $\leftarrow$  anterior
9:     else
10:      actual  $\leftarrow$  (actual  $\rightarrow$  caracteres[ord(k[i])])  $\triangleright \mathcal{O}(1)$ 
11:    end if
12:    i  $\leftarrow$  i + 1  $\triangleright \mathcal{O}(1)$ 
13:  end while
14:  actual  $\rightarrow$  significado  $\leftarrow$  &copiar(s)  $\triangleright \mathcal{O}(1)$ 
15: end function

```

```

1: function iOBTENER(in d: estr, in k: string)  $\rightarrow$  res :  $\sigma$ 
2:   nat i  $\leftarrow$  0  $\triangleright \mathcal{O}(1)$ 
3:   puntero actual  $\leftarrow$  d  $\triangleright \mathcal{O}(1)$ 
4:   while i < |k| do  $\triangleright \mathcal{O}(|k|)$ 
5:     actual  $\leftarrow$  (actual  $\rightarrow$  caracteres[ord(k[i])])  $\triangleright \mathcal{O}(1)$ 
6:     i  $\leftarrow$  i + 1
7:   end while
8:   res  $\leftarrow$  *(actual  $\rightarrow$  significado)
9: end function

```

```

1: function iBORRAR(in/out d: estr, in k: string)
2:   puntero(nodo) actual  $\leftarrow$  d
3:   for i  $\leftarrow$  0 to |k|  $\triangleright \mathcal{O}(1)$ 
4:     actual  $\leftarrow$  (actual  $\rightarrow$  caracteres[ord(k[i])])  $\triangleright \mathcal{O}(1)$ 
5:   end for
6:   (actual  $\rightarrow$  significado)  $\leftarrow$  NULL var puntero(nodo) camino  $\leftarrow$  NULL
7:   while (actual  $\rightarrow$  significado = NULL) or todosNULL(actual  $\rightarrow$  caracteres) do  $\triangleright \mathcal{O}(|k|)$ 
8:     camino  $\leftarrow$  actual  $\triangleright \mathcal{O}(1)$ 
9:     actual  $\leftarrow$  (actual  $\rightarrow$  padre)
10:    delete camino
11:  end while

```

```

1: function IDEFINIDO?(in d: estr, in k: string)  $\longrightarrow$  res : bool
2:   nat i  $\leftarrow$  0  $\triangleright \mathcal{O}(1)$ 
3:   puntero actual  $\leftarrow$  d  $\triangleright \mathcal{O}(1)$ 
4:   bool def  $\leftarrow$  true  $\triangleright \mathcal{O}(1)$ 
5:   while (i < |k| and def) do  $\triangleright \mathcal{O}(|k|)$ 
6:     if actual  $\longrightarrow$  caracteres[ord(k[i])] = NULL then  $\triangleright \mathcal{O}(1)$ 
7:       def  $\leftarrow$  false  $\triangleright \mathcal{O}(1)$ 
8:     else
9:       actual  $\leftarrow$  actual  $\longrightarrow$  caracteres[ord(k[i])]  $\triangleright \mathcal{O}(1)$ 
10:      i  $\leftarrow$  i + 1  $\triangleright \mathcal{O}(1)$ 
11:    end if
12:  end while
13:  res  $\leftarrow$  def  $\wedge$   $\neg$ (actual  $\longrightarrow$  significado(NULL))  $\triangleright \mathcal{O}(1)$ 
14: end function=0

```

3.4. Servicios Usados

Requerimientos sobre el Tipo

- La función $|x|$ debe tener complejidad $\mathcal{O}(1)$ en el caso peor.
- La función $|x|$ 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 DICCIONARIO_{PROM}(σ)

4.1. Especificación

Se usa el TAD DICCIONARIO(κ, σ) especificado en el apunte de Tads básicos.

4.2. Aspectos de la interfaz

4.2.1. Interfaz

parámetros formales

género κ, σ

función $\bullet = \bullet(\text{in } a_1: \kappa, \text{in } a_2: \kappa) \rightarrow res: bool$

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($\text{in } k: \kappa$) $\rightarrow res: \kappa$

Pre $\equiv \{ true \}$

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

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

Descripción: función de copia de κ 's

función COPIAR($\text{in } s: \sigma$) $\rightarrow res: \sigma$

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)

Género $\text{diccProm}(\kappa, \sigma)$

Operaciones básicas de diccionario

DEFINIDO?($\text{in } d: \text{diccProm}(\kappa, \sigma), \text{in } k: \kappa$) $\rightarrow res: bool$

Pre $\equiv \{ true \}$

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

Complejidad: $\mathcal{O}(Na)$ Na es la cantidad de agentes.

Descripción: Devuelve true si y sólo si k está definido en el diccionario.

OBTENER($\text{in } d: \text{diccString}(\kappa, \sigma), \text{in } k: \kappa$) $\rightarrow res: \sigma$

Pre $\equiv \{ \text{def?}(d, k) \}$

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

Complejidad: $\mathcal{O}(Na)$ Na es la cantidad de agentes.

Descripción: Devuelve el significado de la clave k en d .

Aliasing: res no es modificable.

VACIO($\text{in } \text{cantClaves}: nat$) $\rightarrow res: \text{diccString}(\kappa, \sigma)$

Pre $\equiv \{ true \}$

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

Complejidad: $\mathcal{O}(Na)$ Na es la cantidad de agentes.

Descripción: Genera un diccionario vacío.

DEFINIR(in/out d : $\text{diccProm}(\kappa, \sigma)$, **in** k : κ , **in** s : σ)

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

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

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

$\text{diccProm}(\kappa, \sigma)$ se representa con estr

donde estr es

$\text{tupla}(\text{CantClaves: } \text{nat} \times \text{tabla: } \text{arreglo de lista}(\text{datos}))$

donde datos es

$\text{tupla}(\text{clave: } \kappa \times \text{significado: } \sigma)$

4.3.2. Justificación

4.3.3. 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

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

$(\forall e : \text{estr})$

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

$(1)(\forall i : \text{nat})(i < 256 \Rightarrow \text{definido?}(e \rightarrow \text{caracteres}, i)) \wedge_L$

$(2)(e \rightarrow \text{significado} = \text{NULL}) \wedge_L$

$(2)(\exists n : \text{nat})(\text{finaliza}(e, n)) \wedge_L$

$(3)(\forall p, q : \text{puntero}(\text{nodo}))(p \in \text{punteros}(e) \wedge q \in (\text{punteros}(e) - \{p\}) \Rightarrow p \neq q) \wedge_L$

)

4.3.4. Función de Abstracción

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

4.3.5. Algoritmos

```

1: function IVACIO(in  $n: nat$ )  $\longrightarrow$   $res: estr$   $\triangleright \mathcal{O}(cantClaves)$ 
2:    $var$  arreglo(lista(datos))  $tabla \leftarrow$  crearArreglo[n]  $\triangleright \mathcal{O}(cantClaves)$ 
3:   for  $i \leftarrow 0$  to  $n$  do  $\triangleright \mathcal{O}(cantClaves)$ 
4:      $tabla[i] \leftarrow$  Vacía()  $\triangleright \mathcal{O}(1)$ 
5:   end for
6:    $res \leftarrow \langle n, tabla \rangle$   $\triangleright \mathcal{O}(1)$ 
7: end function

```

```

1: function IDEFINIR(in/out  $d: estr$ , in  $k: nat$ , in  $s: \sigma$ )  $\triangleright \mathcal{O}(1)$ 
2:    $nat\ i \leftarrow$  fHash( $k, e.cantClaves$ )  $\triangleright \mathcal{O}(1)$ 
3:    $e.tabla[i] \leftarrow$  AgregarAtras( $e.tabla[i], \langle k, s \rangle$ )  $\triangleright \mathcal{O}(1)$ 
4: end function

```

```

1: function IOBTENER(in  $d: estr$ , in  $k: nat$ )  $\longrightarrow$   $res: \sigma$   $\triangleright \mathcal{O}(longitud(tabla[i]))$ 
2:    $nat\ i \leftarrow$  fHash( $k, e.cantClaves$ )  $\triangleright \mathcal{O}(1)$ 
3:    $var\ itLista(datos)\ it \leftarrow$  crearIt( $tabla[i]$ )
4:   while haySiguiente(it) do
5:     if siguiente(it).clave =  $k$  then
6:        $res \leftarrow$  siguiente(it).significado
7:     end if
8:   end while
9: end function

```

```

1: function IDEFINIDO?(in  $d: estr$ , in  $k: nat$ )  $\longrightarrow$   $res: bool$   $\triangleright \mathcal{O}(longitud(tabla[i]))$ 
2:    $nat\ i \leftarrow$  fHash( $k, e.cantClaves$ )  $\triangleright \mathcal{O}(1)$ 
3:    $var\ itLista(datos)\ it \leftarrow$  crearIt( $tabla[i]$ )
4:    $bool\ aux \leftarrow$  false
5:   while haySiguiente(it) do
6:     if siguiente(it).clave =  $k$  then
7:        $aux \leftarrow$  true
8:     end if
9:   end while
10:   $res \leftarrow$  aux
11: end function

```

```

1: function FHASH(in  $k: nat$ , in  $cantClaves: nat$ )  $\longrightarrow$   $res: nat$   $\triangleright \mathcal{O}(1)$ 
2:    $res \leftarrow k \bmod cantClaves$   $\triangleright \mathcal{O}(1)$ 
3: end function

```

4.4. Servicios Usados

Requerimientos sobre el Tipo

- La función $|x|$ debe tener complejidad $\mathcal{O}(1)$ en el caso peor.
- La función $|x|$ 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 $\tilde{\text{último}}$ 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.