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) $\longrightarrow res : campus$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} crearCampus(c, f)$ }
Complejidad: $\mathcal{O}(1)$
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 $\equiv$ { $c =_{\text{obs}} c_0 \wedge$ posValida(p,c) $\wedge_{\text{L}}$ $\neg$ocupada?(p,c)}
Post $\equiv$ { $c =_{\text{obs}} agregarObstaculo(p, c_0)$}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve $true$ sii p esta ocupada por un obstaculo.

PosValida?(**in** c: $campus$, **in** p: $posicion$) $\longrightarrow res : bool$
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. La complejidad es O(1) dado que los vecinos son a lo sumo 4, o sea, constantes

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

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: $agente$) $\longrightarrow res : nat$

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.

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.

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

Post $\equiv$ { $r =_{\text{obs}} ingresarEstudiante(e, p, \text{r}_0)$}

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

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

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

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

Post $\equiv$ { $r =_{\text{obs}} ingresarHippie(h, p, \text{r}_0)$}

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

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

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

Pre $\equiv$ { $r =_{\text{r}_0} \wedge$ 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 $\equiv$ { $r =_{\text{obs}} moverEstudiante(e, d, \text{r}_0)$}

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

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

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

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

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

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

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

MoverAgente(**in/out** r: $rastr$, **in** a: $agente$) $\longrightarrow$
Pre $\equiv$ { $r =_{r_0} \wedge$ 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: $\mathcal{O}(1)$

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: $\mathcal{O}(1)$

Descripción: Devuelve el agente con mas capturas.

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))*** $\times$
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

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ ($true \iff$
(1)(2)(3) ($\forall$ a,a2: Agente)(def?(a,e.agentes) $\wedge$ def?(a2,e.agentes)) $\Rightarrow_{\text{L}}$
(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)
$\wedge$ (4) PosValida(obtener(a,e.agentes).PosActual))

2.3.4. Función de Abstracción

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

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 ← crearIt (e.agentes)
3: end function
```

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

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

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

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

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: $\mathcal{O}(Na)$ *Na es la cantidad de agentes.*
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: $\mathcal{O}(Na)$ *Na es la cantidad de agentes.*
Descripción: Devuelve el significado de la clave k en d.
Aliasing: res no es modificable.

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

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

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

Descripción: Genera un diccionario vacío.

DEFINIR(**in/out** d: $diccProm(\kappa, \sigma)$, **in** k: κ, **in** s: σ)
Pre $\equiv$ { $d =_{\text{obs}} d_0$ }
Post $\equiv$ { $d =_{\text{obs}} definir(k, s, d_0)$ }
Complejidad: $\mathcal{O}(1)$
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(
CantClaves: ***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

- Todas las posiciones del arreglo de caracteres estÃ¡n definidas.
- No hay claves de 0 caracteres. El significado de la raÃz es NULL.
- No hay ciclos en la estructura. Es decir, existe una cota superior sobre la cantidad de niveles posibles del Ã¡rbol.
- Dado un nodo cualquiera del trie, existe un Ãºnico camino desde la raÃz 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) $\wedge_{\text{L}}$
)

4.3.4. Función de Abstracción

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

Funciones Auxiliares

4.3.5. Algoritmos

```
1: function iVacio(in n: nat )⟶ res : estr                          ▷ O(cantClaves)
2:     var arreglo(lista(datos)) tabla ← crearArreglo[n]            ▷ O(cantClaves)
3:     for i ← 0 to n do                                            ▷ O(cantClaves)
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.cantClaves)                               ▷ 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.cantClaves)                               ▷ 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.cantClaves)                               ▷ 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 cantClaves: nat)⟶ res : nat          ▷ O(1)
2:     res ← k mod cantClaves                                       ▷ 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 Ãºltimo 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.