1. Módulo Campus

Interfaz

se explica con: CAMPUS

géneros: `campus`

Operaciones básicas de campus

CREARCAMPUS(**in** f : `nat`, **in** c : `nat`) $\rightarrow$ res : `campus`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{crearCampus}(f, c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un nuevo campus sin obstáculos.
Aliasing: No hay aspectos de aliasing.

AGREGAROBSTÁCULO(**in** p : `posicion`, **in/out** c : `campus`)
Pre $\equiv$ $\{c =_{\text{obs}} c_0 \wedge \text{posVálida?}(p, c) \wedge_{\text{L}} \neg\text{ocupada?}(p, c)\}$
Post $\equiv$ $\{c =_{\text{obs}} \text{agregarObstáculo}(p, c_0)\}$
Complejidad: $\mathcal{O}(N_o)$
Descripción: Agrega el obstáculo al campus. N_o denota la cantidad de obstáculos actuales.
Aliasing: No hay aspectos de aliasing.

FILAS(**in** c : `campus`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{filas}(c)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve la cantidad de filas que tiene el campus.
Aliasing: No hay aspectos de aliasing.

COLUMNAS(**in** c : `campus`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{columnas}(c)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve la cantidad de columnas que tiene el campus.
Aliasing: No hay aspectos de aliasing.

OCUPADA?(**in** p : `posicion`, **in** c : `campus`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{\text{posVálida?}(p, c)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{ocupada?}(p, c)\}$
Complejidad: $\mathcal{O}(N_o)$
Descripción: Revisa si la posición p esta ocupada. N_o denota la cantidad de obstáculos actuales.
Aliasing: No hay aspectos de aliasing.

POSVÁLIDA?(**in** p : `posicion`, **in** c : `campus`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{posVálida?}(p, c)\ \}$
Complejidad: $\mathcal{O}(1)$
Descripción: Chequea que las componentes X e Y de la posición p sean menores a las columnas y a las filas del campus, respectivamente.
Aliasing: No hay aspectos de aliasing.

ESINGRESO?(**in** p : `posición`, **in** c : `campus`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{esIngreso?}(p, c)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve el resultado de evaluar si la posición pasada como argumento pertenece a las posiciones consideradas como ingresos al campus.
Aliasing: No hay aspectos de aliasing.

ingresoSuperior?(**in** p: `posición`, **in** c: `campus`) → res : `bool`
Pre ≡ {$true$}
Post ≡ {$res =_{\text{obs}}$ ingresoSuperior?(p, c)}
Complejidad: Θ(1)
Descripción: Devuelve el resultado de evaluar si la posición pasada como argumento pertenece a las posiciones consideradas como ingresos superiores al campus.
Aliasing: No hay aspectos de aliasing.

ingresoInferior?(**in** p: `posición`, **in** c: `campus`) → res : `bool`
Pre ≡ {$true$}
Post ≡ {$res =_{\text{obs}}$ ingresoInferior?(p, c)}
Complejidad: Θ(1)
Descripción: Devuelve el resultado de evaluar si la posición pasada como argumento pertenece a las posiciones consideradas como ingresos inferiores al campus.
Aliasing: No hay aspectos de aliasing.

vecinos(**in** p: `posición`, **in** c: `campus`) → res : `conj(posición)`
Pre ≡ {posVálida(p, c)}
Post ≡ {$res =_{\text{obs}}$ vecinos(p, c)}
Complejidad: Θ(1)
Descripción: Devuelve el conjunto de posiciones adyacentes de una posición del campus.
Aliasing: No hay aspectos de aliasing.

distancia(**in** p: `posición`, **in** p_2: `posición`, **in** c: `campus`) → res : `nat`
Pre ≡ {$true$}
Post ≡ {$res =_{\text{obs}}$ distancia(p, p_2, c)}
Complejidad: Θ(1)
Descripción: Devuelve la distancia que hay entre la posición p y la posición p_2 dentro del campus.
Aliasing: No hay aspectos de aliasing.

proxPosición(**in** p: `posición`, **in** d: `dirección`, **in** c: `campus`) → res : `posición`
Pre ≡ {posVálida(p, c)}
Post ≡ {$res =_{\text{obs}}$ proxPosición(p, d, c)}
Complejidad: Θ(1)
Descripción: Devuelve la posición resultante de moverse desde la posición p hacia la dirección d.
Aliasing: No hay aspectos de aliasing.

ingresosMásCercanos(**in** p: `posición`, **in** c: `campus`) → res : `conj(posición)`
Pre ≡ {posVálida(p, c)}
Post ≡ {$res =_{\text{obs}}$ ingresosMásCercanos(p, c)}
Complejidad: Θ(1)
Descripción: Devuelve un conjunto que contiene las posiciones de los ingresos más cercanos.
Aliasing: No hay aspectos de aliasing.

posicionMinimaDistancia(**in** p: `posición`,**in** cp: `conj(posición)`, **in** c: `campus`) → res : `posición`
Pre ≡ {¬ EsVacio?(cp)}
Post ≡ {(∀ p_2 : posicion) posVálida?(p_2, c) $\wedge_{\text{L}}$ distancia(p, p_2, c) ≥ distancia(p, res, c)}
Complejidad: $\mathcal{O}$(Cardinal(cp))
Descripción: Devuelve una posición de cp que está a la mínima distancia de todas las de cp con p.
Aliasing: No hay aspectos de aliasing.

enQuéDireccionesVoy(**in** p: `posición`,**in** p_2: `posición`, **in** c: `campus`) → res : `conj(posición)`
Pre ≡ {posVálida?(p, c) ∧ posVálida?(p_2, c)}
Post ≡ {(∀ d:direccion) ($d \in cd \Rightarrow$ (distancia(p, p_2, c) ≥ distancia(proxPosición(p, res, c), p_2, c) ∧ proxPosición(p, res, c) ∈ vecinosValidos?(vecinos(p, c),c)))}
Complejidad: $\mathcal{O}$(1)
Descripción: Devuelve un conjunto que contiene las direcciones en las que hay que moverse para llegar de p a p_2 con el camino más corto.
Aliasing: No hay aspectos de aliasing.

enQuéDirecciónVoy(**in** p: `posición`, **in** p_2: `posición`, **in** c: `campus`) $\rightarrow$ res : `direccion`
Pre $\equiv$ {posVálida?(p, c) $\wedge$ posVálida?(p_2, c)}
Post $\equiv$ {distancia(p, p_2, c) $\geq$ distancia(proxPosición(p, res, c), p_2, c) $\wedge$
proxPosición(p, res, c) $\in$ vecinosValidos?(vecinos(p, c),c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve una dirección en la que hay que moverse para llegar de p a p_2 con el camino más corto.
Aliasing: No hay aspectos de aliasing.

objetivosDeMínimaDistancia(**in** p: `posición`, **in** cp: `conj(posicion)`, **in** $dist$: `nat` **in** c: `campus`) $\rightarrow$ res : `conj(posición)`
Pre $\equiv$ {posVálida?(p, c) $\wedge$ ($\forall$ t:posicion) (t $\in$ cp $\Rightarrow$ posVálida?(t, c)}
Post $\equiv$ {($\forall$ t:posicion) ($t \in res \Rightarrow$ (distancia(p, t, c) = dist)}
Complejidad: $\mathcal{O}(\#cp)$
Descripción: Devuelve un conjunto que contiene las posiciones de cp que están a distancia dist de p.
Aliasing: No hay aspectos de aliasing.

Representación

Representación del Campus

`Campus` **se representa con** `estr`

donde `estr` es `tupla`($filas$: `nat`, $columnas$: `nat`, $ocupadas$: `conj(posición)`)
donde `posicion` es `tupla`(X: `nat`, Y: `nat`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ (($\forall$ p:posición) $p \in e.ocupadas \Rightarrow_{\text{L}}$
$(0 < p.X \wedge p.X \leq e.columnas \wedge 0 < p.Y \wedge p.Y \leq e.filas)$)

Abs : estr e $\longrightarrow$ campus {Rep(e)}
Abs(e) $\equiv$ c : campus | ($e.filas =_{\text{obs}}$ filas(c) $\wedge$ $e.columnas =_{\text{obs}}$ columnas(c) $\wedge_{\text{L}}$
($\forall$ p:posición) posVálida?(p, c) $\Rightarrow_{\text{L}}$($p \in e.ocupadas =_{\text{obs}}$ ocupada?(p, c)))

PosValida? : posicion p $\times$ campus c $\longrightarrow$ bool
PosVálida?(p, c) $\equiv$ (0 < p.X) $\wedge$ (p.X $\leq$ filas(c)) $\wedge$ (0 < $p.Y$) $\wedge$ ($p.Y \leq$ columnas(c))

Algoritmos

Algorithm 1 crearCampus

1: **procedure** iCrearCampus(**in** $filas$: `nat`, **in** $columnas$: `nat`) $\rightarrow$ res : `estr`
2: $res.filas \leftarrow filas$ ▷ $\mathcal{O}(1)$
3: $res.columnas \leftarrow columnas$ ▷ $\mathcal{O}(1)$
4: $res.ocupadas \leftarrow$ Vacío() ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: $\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$ (por álgebra de órdenes)

Algorithm 2 AgregarObstáculo

1: **procedure** iAgregarObstáculo(**in** p: `posición`, **in/out** e: `estr`)
2: Agregar($e.ocupadas$, p) ▷ $\mathcal{O}(N_o)$

Complejidad: $\mathcal{O}(N_o)$
Justificación: Agregar en un conjunto lineal tiene orden $\mathcal{O}(N_o)$, siendo N_o la cantidad actual de obstáculos.

Algorithm 3 Filas

procedure iFILAS(**in** e : `estr`) $\rightarrow$ res : `nat`
$res \leftarrow e.filas$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Tenemos guardado en memoria el natural. Recuperarlo es $\mathcal{O}(1)$ y asignarlo también.

Algorithm 4 Columnas

procedure iCOLUMNAS(**in** e : `estr`) $\rightarrow$ res : `nat`
$res \leftarrow e.columnas$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Tenemos guardado en memoria el natural. Recuperarlo es $\mathcal{O}(1)$ y asignarlo también.

Algorithm 5 Ocupada?

1: **procedure** iOCUPADA?(**in** p : `posición`, **in** e : `estr`) $\rightarrow$ res : `bool`
2: $res \leftarrow$ pertenece?(p, $e.ocupadas$) ▷ $\mathcal{O}(N_o)$

Complejidad: $\mathcal{O}(N_o)$
Justificación: Complejidad exportada del apunte de módulos básicos (módulo conjunto lineal), tomando la comparación de naturales como $\mathcal{O}(1)$, y la cantidad de elementos del conjunto = N_o

Algorithm 6 posVálida?

1: **procedure** iPOSVÁLIDA?(**in** p : `posición`, **in** e : `estr`) $\rightarrow$ res : `bool`
2: $res \leftarrow (0 < p.X) \wedge (p.X \leq e.columnas) \wedge (0 < p.Y) \wedge (p.Y \leq e.filas)$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Todas las comparaciones entre números naturales son $\mathcal{O}(1)$ (y la asignación también)

Algorithm 7 esIngreso?

1: **procedure** iESINGRESO?(**in** p : `posición`, **in** e : `estr`) $\rightarrow$ res : `bool`
2: $res \leftarrow$ ingresoSuperior?(p, c) $\vee$ ingresoInferior?(p, c) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Las funciones que utiliza son $\mathcal{O}(1)$

Algorithm 8 ingresoSuperior?

1: **procedure** iINGRESOSUPERIOR?(**in** p : `posición`, **in** e : `estr`) $\rightarrow$ res : `bool`
2: $res \leftarrow (p.Y = 1)$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: La comparación es $\mathcal{O}(1)$ y la asignación también.

Algorithm 9 ingresoInferior?

1: **procedure** iINGRESOINFERIOR?(**in** p : `posición`, **in** e : `estr`) $\rightarrow$ res : `bool`
2: $res \leftarrow (p.Y = e.filas)$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: La comparación es $\mathcal{O}(1)$ y la asignación también.

Algorithm 10 Vecinos

1: **procedure** iVECINOS(**in** p : `posición`, **in** e : `estr`) $\rightarrow$ res : `conj(posición)`
2: c : conj(posición) $\leftarrow$ Vacio() ▷ $\mathcal{O}(1)$
3: AgregarRapido(c, $< p.X + 1, p.Y >$) ▷ $\mathcal{O}(1)$
4: AgregarRapido(c, $< p.X - 1, p.Y >$) ▷ $\mathcal{O}(1)$
5: AgregarRapido(c, $< p.X, p.Y + 1 >$) ▷ $\mathcal{O}(1)$
6: AgregarRapido(c, $< p.X, p.Y - 1 >$) ▷ $\mathcal{O}(1)$
7: res $\leftarrow$ vecinosVálidos(c, e) ▷ $\mathcal{O}(4)$

Complejidad: $\mathcal{O}(1)$
Justificación: $\mathcal{O}(4) = \mathcal{O}(1)$ (por álgebra de órdenes)

Algorithm 11 vecinosVálidos?

```
1: procedure iVECINOSVÁLIDOS(in cp: conj(posición), in e: estr ) → res : conj(posición)
2:     nuevoConj : conj(posición) ← Vacío()                                  ▷ O(1)
3:     it : itConj ← CrearIt(cp)                                             ▷ O(1)
4:     while HaySiguiente?(it) do                                            ▷ O(1) * O(N_p)
5:         if posVálida(Siguiente(it)) then                                  ▷ O(1)
6:             AgregarRapido(nuevoConj, Siguiente(it))                       ▷ O(1)
7:         Avanzar(it)                                                       ▷ O(1)
8:     res ← nuevoConj                                                       ▷ O(N_p)
```

Complejidad: $\mathcal{O}(N_p) + \mathcal{O}(N_p) = \mathcal{O}(N_p)$ (álgebra de órdenes)
Justificación: Hay que iterar todas las posiciones del conjunto *cp*. N_p denota la cantidad de posiciones del conjunto *cp*
Pre = $\{true\}$
Post = $\{res =_{\text{obs}} \text{vecinosVálidos}(cp, c)\}$

Algorithm 12 Distancia

```
1: procedure iDISTANCIA(in p: posición, in p_2 : posición, in e: estr ) → res : nat
2:     res ← |p.X − p_2.X| + |p.Y − p_2.Y |                                  ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificación: Las operaciones entre naturales son $\mathcal{O}(1)$

Algorithm 13 proxPosición

```
1: procedure iPROXPOSICIÓN(in p: posición, in d: dirección, in e: estr ) → res : posición
2:     int movEnX ← 0                                                        ▷ O(1)
3:     int movEnY ← 0                                                        ▷ O(1)
4:     if d = izq then                                                       ▷ O(1)
5:         movEnX ← -1                                                       ▷ O(1)
6:     else
7:         if d = der then                                                   ▷ O(1)
8:             movEnX ← 1                                                    ▷ O(1)
9:         else
10:            if d = arriba then                                            ▷ O(1)
11:                movEnY ← -1                                               ▷ O(1)
12:            else
13:                movEnY ← 1                                                ▷
14:    posicion posResultante ← <posActual.X + movEnX, posActual.Y + movEnY>  ▷ O(1)
15:    return posResultante                                                  ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificación: Las asignaciones y las guardas son $\mathcal{O}(1)$

Algorithm 14 ingresosMásCercanos

```
1: procedure iINGRESOSMÁSCERCANOS(in p: posición, in e: estr ) → res : conj(posición)
2:     c : conj(posición) ← Vacio()                                          ▷ O(1)
3:     if distancia(p, < p.X, 1 >, c) < distancia(p, < p.X, filas(c) >, c) then   ▷ O(1)
4:         AgregarRapido(c, < p.X, 1 >)                                      ▷ O(1)
5:     else
6:         if distancia(p, < p.X, 1 >, c) > distancia(p, < p.X, e.filas >, c) then   ▷ O(1)
7:             AgregarRapido(c, < p.X, e.filas >)                            ▷ O(1)
8:         else
9:             AgregarRapido(c, < p.X, 1 >)                                  ▷ O(1)
10:            AgregarRapido(c, < p.X, e.filas >)                            ▷ O(1)
11:    res ← c                                                               ▷ O(2)
```

Complejidad: $\mathcal{O}(2) = \mathcal{O}(1)$ (por álgebra de órdenes)
Justificación: Los agregar rápido del conjunto lineal son $\mathcal{O}(1)$

Algorithm 15 posicionMinimaDistancia

```
1: procedure iPosicionMinimaDistancia(in p : posición, in cp : conj(posición), in e : estr) → res : posición
2:     it : itConj(posicion) ← CrearIt(cp)                                          ▷ O(1)
3:     minima : posicion ← Siguiente(it)                                            ▷ O(1)
4:     Avanzar(it)                                                                  ▷ O(1)
5:     while HaySiguiente(it) do                                                    ▷ O(1) * O(Cardinal(cp))
6:         if Distancia(p, Siguiente(it)) < minima then                             ▷ O(1)
7:             minima ← Siguiente(it)                                               ▷ O(1)
8:         Avanzar(it)                                                              ▷ O(1)
9:     res ← minima                                                                 ▷ O(1)
```

Complejidad: $\mathcal{O}(\text{Cardinal}(cp))$
Justificación: Se recorre todo el conjunto cp con un iterador. En cada iteración se hacen pasos $\mathcal{O}(1)$, por lo que al final termina dando una complejidad lineal en la cantidad de elementos del conjunto cp.
Pre = $\{\neg$ EsVacio?$(cp)\}$
Post = $\{(\forall\ p_2$: posicion) posVálida?$(p_2, c) \wedge_{\text{L}}$ distancia$(p, p_2, c) \geq$ distancia$(p, res, c)\}$

Algorithm 16 enQuéDireccionesVoy

```
1: procedure iEnQuéDireccionesVoy(in p : posición, in p2 : posición, in e : estr) → res : conj(direccion)
2:     cd : conj(direccion) ← Vacio()                                               ▷ O(1)
3:     if p.X < p2.X then                                                           ▷ O(1)
4:         AgregarRapido(cd, der)                                                   ▷ O(1)
5:     if p.X > p2.X then                                                           ▷ O(1)
6:         AgregarRapido(cd, izq)                                                   ▷ O(1)
7:     if p.Y < p2.Y then                                                           ▷ O(1)
8:         AgregarRapido(cd, abajo)                                                 ▷ O(1)
9:     if p.Y > p2.Y then                                                           ▷ O(1)
10:        AgregarRapido(cd, arriba)                                                ▷ O(1)
11:    return cd                                                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificación: Todas las operaciones son $\mathcal{O}(1)$.
Pre = $\{$posVálida?$(p, e) \wedge$ posVálida?$(p_2, e)\}$
Post = $\{$ $(\forall\ d$:direccion$)$ $(d \in cd \Rightarrow$ (distancia$(p, p_2, e) \geq$ distancia(proxPosición$(p, res, e), p_2, e) \wedge$ proxPosición$(p, res, e) \in$ vecinosValidos?(vecinos(p, e),e)))$\}$

Algorithm 17 enQuéDirecciónVoy

```
1: procedure iEnQuéDirecciónVoy(in p : posición, in p2 : posición, in e : estr) → res : direccion
2:     itConj(direccion) it ← CrearIt(EnQuéDireccionesVoy(p,p2,e))          ▷ O(1)
3:     direccion res ← Siguiente(it)                                        ▷ O(1)
4:     return res                                                           ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificación: Todas las operaciones son $\mathcal{O}(1)$.
Pre = {posVálida?(p, e) ∧ posVálida?(p_2, e)}
Post = {distancia(p, p_2, e) ≥ distancia(proxPosición(p, res, e), p_2, e) ∧ proxPosición(p, res, e) ∈ vecinosValidos?(vecinos(p, e),e)}

Algorithm 18 objetivosDeMínimaDistancia

```
1: procedure iObjetivosDeMínimaDistancia(in p : posición, in cp : conj(posicion), in dist : nat, in e : estr)
   → res : conj(posicion)
2:     conj(posicion) res ← Vacio()
3:     itConj(posicion) it ← CrearIt(cp)                                    ▷ O(1)
4:     while HaySiguiente(it) do                                            ▷ O(1) x #cp
5:         if distancia(p,Siguiente(it),e) = dist then                      ▷ O(1)
6:             AgregarRapido(res,Siguiente(it))                             ▷ O(1)
7:         Avanzar(it)                                                      ▷ O(1)
8:     return res                                                           ▷ O(1)
```

Complejidad: $\mathcal{O}(\#cp)$
Justificación: Todas las operaciones son $\mathcal{O}(1)$ y se repiten a lo sumo #cp veces.
Pre = {posVálida?(p, e) ∧ (∀ t:posicion) (t ∈ cp ⇒ posVálida?(t, e)}
Post = {(∀ t:posicion) (t ∈ res ⇒ (distancia(p, t, e) = dist)}

2. Módulo CampusSeguro

Interfaz

parámetros formales
géneros α
función Copiar(**in** a: α) → res : α
Pre ≡ {true}
Post ≡ {res =$_{\text{obs}}$ a}
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's

se explica con: CampusSeguro

géneros: `campusSeg`

usa: Nombre : `String`, agente : `nat`, hippie : `String`, estudiante : `String`, dirección : Enum(arriba, abajo, der, izq), posicion : `tupla(X : nat, Y : nat)`

Operaciones básicas de campusSeg

campus(**in** c: `campusSeg`) → res : `campus`
Pre ≡ {$true$}
Post ≡ {res =$_{\text{obs}}$ campus(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el campus.

estudiantes(**in** c: `campusSeg`) → res : `itdiccT(Nombre)`
Pre ≡ {$true$}
Post ≡ {res =$_{\text{obs}}$ estudiantes(c)}
Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve los estudiantes del campus.
Aliasing: Retorna un iterador del conjunto de estudiantes del Campus.

HIPPIES(**in** c: `campusSeg`) $\rightarrow$ res : `itdiccT(Nombre)`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{hippies}(c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los hippies del campus.
Aliasing: Retorna un iterador del conjunto de hippies del Campus.

AGENTES(**in** c: `campusSeg`) $\rightarrow$ res : `itConj(agente)`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{agentes}(c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los agentes del campus.
Aliasing: Retorna un iterador del conjunto de agentes del Campus.

POSESTYHIP(**in** n: `nombre` **in** c: `campusSeg`) $\rightarrow$ res : `posicion`
Pre $\equiv$ $\{n \in (\text{estudiantes}(c) \cup \text{hippies}(c))\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{posEstudianteYHippie}(n, c)\}$
Complejidad: $\mathcal{O}(|N_m|)$
Descripción: Devuelve la posición del estudiante o el hippie n.

POSAGENTE(**in** a: `agente` **in** c: `campusSeg`) $\rightarrow$ res : `posicion`
Pre $\equiv$ $\{a \in \text{agentes}(c)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{posAgente}(a, c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la posición del agente a.

CANTSANCIONES(**in** a: `agente` **in** c: `campusSeg`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{a \in \text{agentes}(c)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{cantSanciones}(a, c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve las sanciones del agente a.

CANTHIPPIESATRAPADOS(**in** a: `agente` **in** c: `campusSeg`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{a \in \text{agentes}(c)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{cantHippiesAtrapados}(a, c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de hippies atrapados por el agente a.

COMENZARRASTRILLAJE(**in** c: `campus` **in** d: `dicc(agente,posicion)`) $\rightarrow$ res : `CampusSeg`
Pre $\equiv$ $\{(\forall\ a$:agente) (def?(a, d) $\Rightarrow_{\text{L}}$(posVálida(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 $\equiv$ $\{res =_{\text{obs}} \text{comenzarRastrillaje}(c,d)\}$
Complejidad: $\mathcal{O}(copy(c)) + \mathcal{O}(N_a) + \mathcal{O}(\#e.campus.ocupadas) + \mathcal{O}(e.campus.columnas) * \mathcal{O}(e.campus.filas)$
Descripción: Comienza el rastrillaje de los agentes.

INGRESARESTUDIANTE(**in** n: `nombre` **in** p: `posicion` **in/out** c: `CampusSeg`)
Pre $\equiv$ $\{c =_{\text{obs}} c_0 \wedge n \notin (\text{estudiantes}(c) \cup \text{hippies}(c)) \wedge \text{esIngreso?}(p,\text{campus}(c)) \wedge \neg\text{estaOcupada?}(p,c)\ \}$
Post $\equiv$ $\{res =_{\text{obs}} \text{ingresarEstudiante}(n,p,c_0)\}$
Complejidad: $\mathcal{O}(|N_m|)$
Descripción: Ingresa el estudiante en el campus.

INGRESARHIPPIE(**in** n: `nombre` **in** p: `posicion` **in/out** c: `CampusSeg`)
Pre $\equiv$ $\{c =_{\text{obs}} c_0 \wedge n \notin (\text{estudiantes}(c) \cup \text{hippies}(c)) \wedge \text{esIngreso?}(p,\text{campus}(c)) \wedge \neg\text{estaOcupada?}(p,c)\ \}$
Post $\equiv$ $\{res =_{\text{obs}} \text{ingresarHipppie}(n,p,c_0)\}$
Complejidad: $\mathcal{O}(|N_m|)$
Descripción: Ingresa el hippie en el campus.

MOVERESTUDIANTE(**in** n: `nombre` **in** d: `direccion` **in/out** c: `CampusSeg`)

Pre $\equiv \{c =_{\text{obs}} c_0 \land n \in \text{estudiantes}(c) \land (\text{seRetira}(n,dir,c) \lor (\text{posVálida}(\text{proxPosición}(\text{posEstYHip}(n,c),d,\text{campus}(c)),\text{campus}(c))))\}$
Post $\equiv \{res =_{\text{obs}} \text{moverEstudiante}(n,d,c_0)\}$
Complejidad: $\mathcal{O}(|N_m|)$
Descripción: El estudiante se mueve en la direccion d.

MOVERHIPPIE(**in** n : `nombre` **in/out** c : `CampusSeg`)
Pre $\equiv \{c =_{\text{obs}} c_0 \land n \in \text{hippies}(c) \land \neg\text{todasOcupadas?}(\text{vecinos}(\text{posEstYHip}(n,c),\text{campus}(c)),c)\}$
Post $\equiv \{res =_{\text{obs}} \text{moverHippie}(n,d,c_0)\}$
Complejidad: $\mathcal{O}(|N_m|) + \mathcal{O}(|N_e|)$
Descripción: El hippie se mueve hacia el estudiante más cercano.

MOVERAGENTE(**in** a : `agente` **in/out** c : `CampusSeg`)
Pre $\equiv \{c =_{\text{obs}} c_0 \land a \in \text{agentes}(c) \land_{\text{L}} \text{cantSanciones}(a,c) \leq 3 \land \neg\text{todasOcupadas?}(\text{vecinos}(\text{posAgente}(a,c),\text{campus}(c)),c)\}$
Post $\equiv \{res =_{\text{obs}} \text{moverAgente}(n,d,c_0)\}$
Complejidad: $\mathcal{O}(|N_m|) + \mathcal{O}(|N_h|)$
Descripción: El agente se mueve hacia el hippie más cercano.

CANTHIPPIES(**in** c : `CampusSeg`) $\rightarrow$ res : `nat`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} \text{cantHippies}(c)\}$
Complejidad: $\mathcal{O}(|N_h|)$
Descripción: Retorna la cantidad de hippies presentes en el campus.

CANTESTUDIANTES(**in** c : `CampusSeg`) $\rightarrow$ res : `nat`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} \text{cantEstudiantes}(c)\}$
Complejidad: $\mathcal{O}(|N_e|)$
Descripción: Retorna la cantidad de estudiantes presentes en el campus.

MÁSVIGILANTE(**in** c : `CampusSeg`) $\rightarrow$ res : `nat`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} \text{másVigilante}(c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Retorna el as que mas hippies capturó.

CONMISMASSANCIONES(**in** a : `agente` **in** c : `CampusSeg`) $\rightarrow$ res : `conj(agente)`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} \text{conMismasSanciones}(a,c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Retorna los ases con la misma cantidad de sanciones que las del pasado por parámetro.

CONKSANCIONES(**in** k : `nat` **in** c : `CampusSeg`) $\rightarrow$ res : `itConj(agente)`
Pre $\equiv \{true\}$
Post $\equiv \{\text{SecuAConj}(\text{SecuSuby}(res)) =_{\text{obs}} \text{conKSanciones}(k,c)\}$
Complejidad: $\mathcal{O}(N_a)$ en el caso que haya actualizar, en otro caso $\mathcal{O}(log(N_a))$
Descripción: Retorna los ases con la cantidad de sanciones pasadas por parámetro.

Representación

Representación del CampusSeguro

`campusSeg` se representa con `estr`

donde `estr` es `tupla`(*campus*: `Campus`
, *Estudiantes*: `DiccTrie(estudiante,posición)`
, *Hippies*: `DiccTrie(hippie, posición)`
, *AgentesD*: `DiccHash(agente,<agente,posicion,capturados:nat,sanciones:nat,mSanc:itConj(agente),pos:itLista(<conj(agente),nat>)>)`
, *AgentesC*: `Conj(agente)`
, *MasVigilante*: `<ag:agente,cant:nat>`
, *sanciones*: `vector(<itConj(agente),sanc:nat>))`
, *CantSaciones*: `<Lista(<conj:Conj(agente),sanc:nat>),actualizar:bool>`
, *Mapa*: `vector(vector(<est:<oc:bool,n:nombre>,hip:<oc:bool,n:nombre>,agente: <oc:bool,p:puntero(<agente,posicion,capturados:nat,sanciones:nat,mSanc:itConj(agente),pos:itLista(<conj(agente),nat>)>)>,obstaculo:bool>)))`

2.0.1. Invariante de Representación de CampusSeguro

(I) la intersección de los estudiantes y los hippies es disjunta.

(II) AgentesC tiene los mismos agentes que las claves del hash

(III) la posición de cada estudiante de e.estudiantes esta colocada como true en el mapa y en el índice correspondiente (lo mismo para los hippies,agentes(y estos tienen los punteros apuntando al agente dentro del agente en el hash) y obstáculos) y no puede haber ninguna posición en el mapa que tenga true en una componente y la posición tenga algo distinto o este vacía (ni tampoco puede haber alguna que tenga dos o màs componentes en true).

(IV) Las posiciones de los hippies, agentes, estudiantes y obstaculos son distintas.

(V) MasVigilante es uno de los agentes con mas capturados, pero el de menor placa.

(VI) Los agentes de los conjuntos de cantSanciones pertenecen a AgentesC y su posicion en la lista depende de la cantidad de sanciones (la lista està ordenada) y si actualizar es true, entonces cada iterador del arreglo e.sanciones apunta al mismo conjunto que la lista.

(VII) Cada posiciòn de los estudiantes, hippies y agentes debe cumplir que està dentro del rango del mapa.

(VIII) Para cada elemento definido en el hash, el agente debe pertenecer a AgentesC, su posición debe ser la correcta respecto del mapa,el agente debe ser el mismo que el de la clave sus capturados debe ser menor o igual a e.masvigilantes, mSanc debe estar hecho de tal forma que apunte al conjunto donde pertenezca el agente y al hacer siguiente se lo obtenga y pos es un iterador de lista que al hacer siguiente se obtiene el conjunto al que pertenece el agente y sus sanciones deben ser las mismas que el sanc de cantSanciones.

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
($claves(e.Estudiantes)$∩ claves(e.Hippies)) $=_{obs}$ ∅ ∧ e.AgentesC $=_{obs}$ claves(e.AgentesD)
∧ (∀ i: nat) ((i < longitud (e.Mapa) $\Rightarrow_L$ (∀ j: nat) (j < longitud ((e.Mapa)[i]))) $\Rightarrow_L$
((e.Mapa)[i])[j]) $=_{obs}$
$<\Big($ ($\exists es : nombre$)
$\big(definido?(e.Estudiantes, es) \Rightarrow_L$ ($(obtener(e.Estudiantes, es) =_{obs} < i+1, j+1 >)$) $\big)$,
$\big($ ($\exists hip : nombre$)$(definido?(e.Hippies, hip) \Rightarrow_L$ (($obtener(e.Hippies, hip) =_{obs} < i+1, j+1 >$))) $\big)$,
$< \big($($\exists ag : nat$)
$(definido?(e.AgentesD, ag) \Rightarrow_L$ ((($obtener(e.AgentesD, ag)).posicion) =_{obs} < i+1, j+1 >$)) $\big), >$,
$ocupada?(< i+1, j+1 >, e.campus)\Big) >$
∧ (∀ p: posicion) (((∀ es: nombre) (obtener(e.Estudiantes, es) $=_{obs}$ p)) $\Rightarrow_L$((e.Mapa)[(p.X)-1])[(p.Y)-1] $=_{obs}$
< <true,es>, <false,>, <false,NULL>, false>) ∨($\exists hip : nombre$)
$(obtener(e.Hippies, es) =_{obs} p)) \Rightarrow_L ((e.Mapa)[(p.X)-1])[(p.Y)-1] =_{obs} << false,"" >, < true, hip >, < false, NULL >, false >)$∨ (∀ ag: agente) (obtener(e.agentesD, ag).posicion $=_{obs}$ p)) $\Rightarrow_L$($(e.Mapa)[(p.X)-1])[(p.Y)-1] =_{obs} << false,"" >, < false,"" >, <$ true,&(obtener(e.agentesD, ag)>, false>) ∨ (∀ obs: obstaculo)
(obs ∈ e.campus.ocupadas)) $\Rightarrow_L$($(e.Mapa)[(p.X)-1])[(p.Y)-1] =_{obs}$ < <false,"">, <false,"">, <true,NULL>, true>) ∧ e.MasVigilante.ag ∈ e.AgentesC ∧ e.MasVigilante.ag ∈ ultimo(e.CantSanciones)
∧ (∀ ag ∈ ultimo(e.cantSanciones)) e.MasVigilante.ag ≤ ag ∧ e.MasVigilante.cant = obtener(e.AgentesD,e.Masvigilante.ag).capturados
∧ (∀ es: nombre) (definido?(e.Estudiantes, es) $\Rightarrow_L$ (posVálida?(obtener(e.Estudiantes, es), e.campus) $\wedge_L$ ¬ ocupada?(obtener(e.Estudiantes, es),e.campus))
∧ (∀ hip: nombre) (definido?(e.Hippies, hip) $\Rightarrow_L$ (posVálida?(obtener(e.Hippies, hip), e.campus) $\wedge_L$ ¬ ocupada?(obtener(e.Hippies, hip), e.campus))
∧ (∀ ag: agente) (definido?(e.AgentesD, ag) $\Rightarrow_L$ (posVálida?((obtener(e.AgentesD, es)).posicion, e.campus $\wedge_L$ ¬ ocupada?((obtener(e.AgentesD, ag)).posicion, e.campus))) ∧ (∀ cag:conj(<agente,nat>)) (esta?(cag,e.cantSanciones) $\Rightarrow_L$cag.conj ⊂ e.agentesC) ∧ (∀ a,a':agente) a ∈ claves(e.agentesD) $\Rightarrow_L$(obtener(a,e.agentesD).sanciones >obtener(a',e.agentesD) → (∀ ag:agente) (obtener(a,e.agentesD).sanciones = obtener(ag,e.agentesD).sanciones ∧ obtener(ag,e.agentesD).pos = obtener(a,e.agentesD).pos)) ∧ obtener(a,e.agentesD).agente = a e.cantsanciones.actualizar = true $\Rightarrow_L$((∀ i:nat) e.sanciones[i]= crearIt(cantsanciones.Lista[i])))) ∧ (∀ a:agente) def?(a,e.agentesD) $\Rightarrow_L$e.mapa[obtener(a,e.agentesD).posicion.X-1][obtener(a,e.agentesD).posicion.Y-1] = < <false,"">,<false,"">,<true,&obtener(a,e.agentesD)>,false> ∧ siguiente(obtener(a,e.agentesD).mSanc) = a ∧ siguiente(obtener(a,e.agentesD).pos).sanc = obtener(a,e.agentesD).sanciones)))

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

Abs(e) $\equiv$ cs : campusSeg |
campus(cs) $=_{obs}$ e.campus
∧ estudiantes(cs) $=_{obs}$ claves(e.Estudiantes)
∧ hippies(cs) $=_{obs}$ claves(e.Hippies)
∧ agentes(cs) $=_{obs}$ e.AgentesC
∧ (∀ a:agente) a ∈ agentes(cs) $\Rightarrow_L$(posAgente(a,cs) $=_{obs}$ obtener(a,e.AgentesD).posicion
∧ cantSanciones(a,cs) $=_{obs}$ obtener(a,e.AgentesD).sanciones
∧ cantHippiesAtrapados(a,cs) $=_{obs}$ obtener(a,e.AgentesD).capturados)
∧ (∀ id: nombre) ((id ∈ estudiantes(cs) $\Rightarrow_L$ posEstudianteYHippie(id,cs) $=_{obs}$ obtener(e.Estudiantes, id))
∧ (id ∈ hippies(cs) $\Rightarrow_L$ posEstudianteYHippie(id,cs) $=_{obs}$ obtener(e.Hippies, id)))

SecuAConj : secu(α) $\longrightarrow$ conj(α)
SecuAConj(s) $\equiv$ **if** vacia?(s) **then** ∅ **else** Ag(prim(s), SecuAConj(fin(s))) **fi**

Aclaraciones:

1. En algunos algoritmos se usaron ciertos reemplazos para facilitar la lectura:

Si se tiene un iterador a un conjunto de posiciones, it,
Siguiente(it) y Siguiente(it).pos son sinónimos Siguiente(it).tipo = Agente (por ejemplo) devuelve verdadero si y sólo si e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].agente.oc = true.
Siguiente(it).nombre devuelve e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].est.n si y sólo si Siguiente(it).tipo = Estudiante, por ejemplo.

$(e.Mapa[p.X-1])[p.Y-1] \leftarrow$ (Estudiante, n) significa
(e.Mapa[p.X-1])[p.Y-1] $\leftarrow \langle$ <true,n>, <false, >, <false, NULL>, false $\rangle$, y el nombre n se pasa por copia así que la complejidad es $\mathcal{O}(|n|)$, que igual no afecta porque esta asignación se realiza cuando se define un estudiante en esa posición.
Lo mismo para (Hippie, n).
$(e.Mapa[p.X-1])[p.Y-1] \leftarrow$ Libre significa
(e.Mapa[p.X-1])[p.Y-1] $\leftarrow \langle$ <false, >, <false, >, <false, NULL>, false $\rangle$.

2. identificacion es tipo enumerado de $\{ nombre, placa \}$.

3. Algunas líneas de pseudocógido tendrían comentarios de complejidad muy largos que molestarían la lectura del algoritmo y harían más difícil verificar que tuviera la complejidad afirmada, así que se escribió la complejidad que tiene esa línea y las justificaciones en la parte de Justificación.
Un ejemplo de esto se puede ver cuando se llama a vecinosVálidos?(vecinos(p,e.campus)) que tendría complejidad $\mathcal{O}(\#vecinos(p, e.campus))$, pero como ese cardinal está acotado por 4 se escribió $\mathcal{O}(1)$.

4. Si |n| es la longitud de un nombre, por ejemplo de un estudiante de IngresarEstudianteMágicamente, y |Nm| era la longitud más larga de todos los nombres de acuerdo con el enunciado hasta antes de su ingreso, la complejidad de este algoritmo sería el máximo de las dos longitudes, pero pensamos que en el enunciado se tomó que |n| $\leq$ |Nm|.

Algoritmos

Algorithm 19 ConMismasSanciones

1: **procedure** *i*ConMismasSanciones(**in** a : `agente` ,**in** e : `estr`) $\rightarrow$ res : `itConj(agente)`
2: it : itLista(<conj(agente), nat>) $\leftarrow$ <Obtener($e.AgentesD$, a).pos, Obtener($e.AgentedD$, a).$sanciones$> ▷ $\mathcal{O}(1)$
3: $res \leftarrow$ Siguiente(it) ▷ $\mathcal{O}(1)$

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

Algorithm 20 Campus

1: **procedure** *i*Campus(**in** e : `estr`) $\rightarrow$ res : `campus`
2: $res \leftarrow e.campus$ ▷ $\mathcal{O}(1)$

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

Algorithm 21 Estudiantes

1: **procedure** *i*Estudiantes(**in** e : `estr`) $\rightarrow$ res : `itdiccT(nombre)`
2: $res \leftarrow$ crearItClaves($e.Estudiantes$) ▷ $\mathcal{O}(1)$

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

Algorithm 22 Hippies

1: **procedure** *i*HIPPIES(**in** e : `estr`) $\rightarrow$ res : `itdiccT(nombre)`
2: $res \leftarrow$ crearItClaves($e.Hippies$) ▷ $\mathcal{O}(1)$

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

Algorithm 23 Agentes

1: **procedure** *i*AGENTES(**in** e : `estr`) $\rightarrow$ res : `itConj(nat)`
2: $res \leftarrow$ crearIt(e.agentesC) ▷ $\mathcal{O}(1)$

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

Algorithm 24 PosEstYHip

1: **procedure** *i*POSESTYHIP(**in** n : `nombre`,**in** e : `estr`) $\rightarrow$ res : `posición`
2: **if** definido?(e.estudiantes,n) **then** ▷ $O(|Nm|)$
3: $res \leftarrow$ obtener(e.estudiantes,n) ▷ $O(|Nm|)$
4: **else**
5: $res \leftarrow$ obtener(e.hippies,n) ▷ $O(|Nm|)$

Complejidad: $\mathcal{O}(|Nm|)$
Justificación: Evaluar la guarda del if se hace una vez, y para cualquier camino que se tome se tiene la misma complejidad, por lo que la complejidad de evaluar la guarda y llevar a cabo el cuerpo del if es $\mathcal{O}(|Nm|)+\mathcal{O}(|Nm|)$, que es igual a $\mathcal{O}(|Nm|)$ por álgebra de órdenes.

Algorithm 25 PosAgente

1: **procedure** *i*POSAGENTE(**in** a : `agente` , **in** e : `estr`) $\rightarrow$ res : `posición`
2: $res \leftarrow$ significado($e.agentesD$, a).$posicion$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Se usa la operación significado del Diccionario Hash, que tiene complejidad $\mathcal{O}(1)$ en caso promedio.

Algorithm 26 CantSanciones

1: **procedure** *i*CANTSANCIONES(**in** a : `agente`, **in** e : `estr`) $\rightarrow$ res : `nat`
2: $res \leftarrow$ significado($e.agentesD$, a).$sanciones$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Se usa la operación significado del Diccionario Hash, que tiene complejidad $\mathcal{O}(1)$ en caso promedio.

Algorithm 27 CantHippiesAtrapados

1: **procedure** *i*CANTHIPPIESATRAPADOS(**in** a : `agente`, **in** e : `estr`) $\rightarrow$ res : `nat`
2: $res \leftarrow$ significado($e.agentesD$, a).$capturados$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Se usa la operación significado del Diccionario Hash, que tiene complejidad $\mathcal{O}(1)$ en caso promedio.

Algorithm 28 ComenzarRastrillaje

1: **procedure** *i*ComenzarRastrillaje(**in** c: `campus`, **in** d: `diccLineal(agente, posicion)`) $\rightarrow$ e : `estr`
2: $e.campus \leftarrow c$ ▷ $\mathcal{O}(copy(c))$
3: $e.Estudiantes \leftarrow$ Vacío() ▷ $\mathcal{O}(1)$
4: $e.Hippies \leftarrow$ Vacío() ▷ $\mathcal{O}(1)$
5: it : itDicc(agente, posicion) $\leftarrow$ CrearIt(d) ▷ $\mathcal{O}(1)$
6: $n \leftarrow$ #Claves(d) ▷ $\mathcal{O}(1)$
7: $lista$: Lista(<conj:Conj(agente),sanc:nat>) $\leftarrow$ Vacia() ▷ $\mathcal{O}(1)$
8: $conj$: conj(agente) $\leftarrow$ Vacio() ▷ $\mathcal{O}(1)$
9: $posX \leftarrow 0$ ▷ $\mathcal{O}(1)$
10: $posY \leftarrow 0$ ▷ $\mathcal{O}(1)$
11: **while** $posX$ <e.campus.columnas **do** ▷ $\mathcal{O}(e.campus.columnas)$
12: **while** $posY$ <e.campus.filas **do** ▷ $\mathcal{O}(e.campus.filas)$
13: (e.Mapa[posX])[posY] $\leftarrow$ Libre ▷ $\mathcal{O}(1)$
14: $posY \leftarrow posY + 1$ ▷ $\mathcal{O}(1)$
15: $posX \leftarrow posX + 1$ ▷ $\mathcal{O}(1)$
16: AgregarAdelante($lista$, <$conj$, 0>) ▷ $\mathcal{O}(1)$
17: $itlista \leftarrow$ CrearIt($lista$) ▷ $\mathcal{O}(1)$
18: $diccH$:diccHash(agente,<agente,posicion,nat,nat,itConj(agente),itLista(<conj(agente),nat>)>) $\leftarrow$ Crear(n) ▷ $\mathcal{O}(N_a)$
19: $menorPlaca$: agente $\leftarrow$ Siguiente(it) ▷ $\mathcal{O}(1)$
20: **while** HaySiguiente(it) **do** ▷ $\mathcal{O}(N_a)$
21: $tupla$: <agente,posicion,nat,nat,itConj(agente),itLista(<conj(agente),nat>)> ▷ $\mathcal{O}(1)$
22: $\Pi_1(tupla) \leftarrow$ SiguienteClave(it) ▷ $\mathcal{O}(1)$
23: $\Pi_2(tupla) \leftarrow$ SiguienteSignificado(it) ▷ $\mathcal{O}(1)$
24: $\Pi_3(tupla) \leftarrow 0$ ▷ $\mathcal{O}(1)$
25: $\Pi_4(tupla) \leftarrow 0$ ▷ $\mathcal{O}(1)$
26: $\Pi_5(tupla) \leftarrow$ AgregarRapido(Siguiente($itlista$).$conj$, SiguienteClave(it)) ▷ $\mathcal{O}(1)$
27: $\Pi_6(tupla) \leftarrow itlista$ ▷ $\mathcal{O}(1)$
28: Definir($diccH$, SiguienteClave(it), $tupla$) ▷ $\mathcal{O}(1)$ en caso promedio
29: $puntTup \leftarrow$ & Obtener($diccH$, SiguienteClave(it)) ▷ $\mathcal{O}(1)$ en caso promedio
30: ($e.Mapa$[SiguienteSignificado($it2$).X - 1])[SiguienteSignificado($it2$).Y - 1] $\leftarrow$
31: <<false,"">,<false,"">,<true,puntTup>,false> ▷ $\mathcal{O}(1)$
32: **if** SiguienteClave(it) < $menorPlaca$ **then** ▷ $\mathcal{O}(1)$
33: $menorPlaca \leftarrow$ SiguienteClave(it) ▷ $\mathcal{O}(1)$
34: Avanzar(it) ▷ $\mathcal{O}(1)$
35: $e.AgentesD \leftarrow diccH$ ▷ $\mathcal{O}(N_a)$
36: $e.AgentesC \leftarrow$ Claves(d) ▷ $\mathcal{O}(N_a)$
37: $e.MasVigilante \leftarrow menorPlaca$ ▷ $\mathcal{O}(1)$
38: $\Pi_1(e.CantSanciones) \leftarrow lista$ ▷ $\mathcal{O}(N_a)$
39: $\Pi_2(e.CantSanciones) \leftarrow$ false ▷ $\mathcal{O}(1)$
40: $array$: Vector(<itConj(agente),nat>) ▷ $\mathcal{O}(1)$
41: $\Pi_1(array) \leftarrow$ CrearIt(Siguiente($itlista$).$conj$) ▷ $\mathcal{O}(1)$
42: $\Pi_2(array) \leftarrow 0$ ▷ $\mathcal{O}(1)$
43: $e.Sanciones \leftarrow array$ ▷ $\mathcal{O}(1)$
44: $itc \leftarrow$ itConj($e.campus.ocupadas$) ▷ $\mathcal{O}(1)$
45: **while** HaySiguiente(itc) **do** ▷ $\mathcal{O}(\#e.campus.ocupadas)$
46: ($e.Mapa$[Siguiente(itc).X - 1])[Siguiente(itc).Y - 1] $\leftarrow$ <<false,"">,<false,"">,<false,NULL>,true> ▷ $\mathcal{O}(1)$
47: Avanzar(itc) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(copy(c)) + \mathcal{O}(N_a) + \mathcal{O}(\#e.campus.ocupadas) + \mathcal{O}(e.campus.columnas) * \mathcal{O}(e.campus.filas)$
Justificación: Se utilizó álgebra de órdenes para reducir la expresión. Los $\mathcal{O}(N_a)$ se suman todos, y queda $\mathcal{O}(4 * N_a) = \mathcal{O}(N_a)$

Algorithm 29 IngresarEstudiante

1: **procedure** *i*IngresarEstudiante(**in** n : `nombre`,**in** p : `posición`,**in**/**out** e : `estr`)
2: IngresarEstudianteMágicamente(n, p, e) ▷ $\mathcal{O}(|n|)$

Complejidad: $\mathcal{O}(|n|)$
Justificación: $\mathcal{O}(|n|)$

Algorithm 30 IngresarHippie

1: **procedure** *i*IngresarHippie(**in** n : `nombre`,**in** p : `posición`, **in**/**out** e : `estr`)
2: IngresarHippieMágicamente(n,p,e) ▷ $\mathcal{O}(|Nm|)$

Complejidad: $\mathcal{O}(|Nm|)$
Justificación: $\mathcal{O}(|Nm|)$
Idea: Igual a la de IngresarEstudiante.

Algorithm 31 ConKSanciones

1: **procedure** *i*ConKSanciones(**in** k : `nat` ,**in** e : `estr`) → res : `conj(agente)`
2: **if** $e.CantSanciones.actualizar$ **then** ▷ $\mathcal{O}(1)$
3: $e.CantSanciones.actualizar \leftarrow$ false ▷ $\mathcal{O}(1)$
4: i : nat $\leftarrow 0$ ▷ $\mathcal{O}(1)$
5: it : itLista(<conj(agente),nat>) $\leftarrow$ CrearIt($\Pi_1(e.CantSanciones)$) ▷ $\mathcal{O}(1)$
6: **while** HaySiguiente(it) **do** ▷ $\mathcal{O}(n)$
7: **if** $i <$ longitud($e.sanciones$) **then** ▷ $\mathcal{O}(1)$
8: $e.sanciones[i] \leftarrow$ <CrearIt(Π_1(Siguiente(it))), Siguiente(it).$sanc$> ▷ $\mathcal{O}(1)$
9: **else**
10: AgregarAtras($e.sanciones$, <CrearIt(Π_1(Siguiente(it))), Siguiente(it).$sanc$>) ▷ $\mathcal{O}(1)$
11: $i \leftarrow i + 1$ ▷ $\mathcal{O}(1)$
12: Avanzar(it) ▷ $\mathcal{O}(1)$
13: $res \leftarrow e.sanciones[busquedaBinaria(e.sanciones, k)].conj$ ▷ $\mathcal{O}(log(n))$

Complejidad: $\mathcal{O}(n)$ en el caso que haya actualizar, en otro caso $\mathcal{O}(log(n))$
Justificación: $\mathcal{O}(n)$ *o* $\mathcal{O}(log(n))$

Algorithm 32 IngresarHippieMágicamente

```
1:  procedure iIngresarHippieMágicamente(in n : nombre,in p : posición, in/out e : estr )
2:      conj(posicion) vecinasLegales ← vecinosVálidos?(Vecinos(p, e.campus), e.campus)        ▷ O(1)
3:      itConj(posicion) it ← CrearIt(vecinasLegales)                                          ▷ O(1)
4:      bool sinEscapatoria ← todasLasVecinasOcupadas(vecinasLegales,e)                        ▷ por copia, O(1)
5:      bool seNerdiza ← (TodosEstudiantes(vecinasLegales,e)                                   ▷ por copia, O(1)
6:      bool muere ← ( sinEscapatoria ∧ alMenos1Agente(vecinasLegales,e)))                     ▷ por copia, O(1)
7:      bool noSeNerdizaYQueda ← (¬ seNerdiza ∧ ¬ muere)                                       ▷ por copia, O(1)
8:      SeHaceHippie(n, p, e)                                                                  ▷ O(|n|)
9:      while HaySiguiente(it) do                                          ▷ O(1) x #vecinasLegales
10:         conj(posicion) vecinasDeEstaVecina ← vecinosVálidos?
11:         (Vecinos(Siguiente(it), e.campus), e.campus)                                       ▷ O(1)
12:         if Siguiente(it).tipo = Hippie then                                                ▷ O(1)
13:             if todasLasVecinasOcupadas(vecinasDeEstaVecina, e) then                        ▷ O(1)
14:                 if alMenos1Agente(vecinasDeEstaVecina), e) then                            ▷ O(1)
15:                     muerteDelHippie(Siguiente(it).nombre, e)                               ▷ O(|Nm|)
16:                     OtorgarCapturas(Siguiente(it).pos,e)                ▷                  ▷ O(1)
17:         else
18:             if Siguiente(it).tipo = Estudiante then                                        ▷ O(1)
19:                 if alMenos2Hippies(vecinasDeEstaVecina,e) then                             ▷ O(1)
20:                     SeHaceHippie(Siguiente(it).nombre, Siguiente(it).pos, e)               ▷ O(|Nm|)
21:                     if todasLasVecinasOcupadas(vecinasDeEstaVecina, e) then                ▷ O(1)
22:                         OtorgarSanciones(Siguiente(it).pos, e)                             ▷ O(1)
23:                         if alMenos1Agente(vecinasDeEstaVecina, e) then                     ▷ O(1)
24:                             muerteDelHippie(Siguiente(it).nombre, e)                       ▷ O(|Nm|)
25:                             OtorgarCapturas(Siguiente(it).pos,e)                           ▷ O(1)
26:                 else
27:                     if todasLasVecinasOcupadas(vecinasDeEstaVecina, e) then                ▷ O(1)
28:                         OtorgarSanciones(Siguiente(it).pos, e)                             ▷ O(1)
29:         Avanzar(it)                                                                        ▷ O(1)
30:     if seNerdiza then                                                                      ▷ O(1)
31:         Definir(e.Estudiantes,n,p)                                                         ▷ O(|n|)
32:         (e.Mapa[p.X-1])[p.Y-1] ← (Estudiante, n)                                           ▷ O(|n|)
33:     else
34:         if ¬ noSeNerdizaYQueda then                                                        ▷ O(1)
35:             MuerteDelHippie(n,e)                                                           ▷ O(|n|)
36:             OtorgarCapturas(p, e)                                                          ▷ O(1)
```

Complejidad: $\mathcal{O}(|Nm|)$
Justificación:
Líneas 4 y 5: #vecinasLegales ≤ 4.
Idea: Puede o no convertir en hippies a los de su alrededor (y estos nuevos hippies pueden o no ser capturados, pero no volver a estudiantes porque significaría que no tenían otro hippie más alrededor) y (morir o ser convertido en estudiante o quedar hippie)

Algorithm 33 IngresarEstudianteMágicamente

```
1: procedure iIngresarEstudianteMágicamente(in n : nombre,in p : posición, in/out e : estr )
2:     conj(posicion) vecinasLegales ← vecinosVálidos?(Vecinos(p, e.campus), e.campus)                ▷ O(1)
3:     itConj(posicion) it ← CrearIt(vecinasLegales)                                                  ▷ O(1)
4:     bool sinEscapatoria ← todasLasVecinasOcupadas(vecinasLegales,e)                                ▷ por copia, O(1)
5:     bool muere ← ( sinEscapatoria ∧ alMenos1Agente(vecinasLegales,e)))                             ▷ por copia, O(1)
6:     bool seEnhippiza ← (alMenos2Hippies(vecinasLegales,e) )                                        ▷ por copia, O(1)
7:     bool seEnhippizaYQueda ← (seEnhippiza ∧ ¬ muere)                                               ▷ por copia, O(1)
8:     Definir(e.Estudiantes,n,p)                                                                     ▷ O(|n|)
9:     (e.Mapa[p.X − 1])[p.Y − 1] ← (Estudiante, n)                                                   ▷ O(|n|)
10:    while HaySiguiente(it) do                                                  ▷ O(1) x #vecinasLegales
11:        conj(posicion) vecinasDeEstaVecina ← vecinosVálidos?(Vecinos(Siguiente(it).pos, e.campus), e.campus) ▷ O(1)
12:        if Siguiente(it).tipo = Hippie then                                                        ▷ O(1)
13:            if todasLasVecinasOcupadas(vecinasDeEstaVecina, e) then                                ▷ O(1)
14:                if alMenos1Agente(vecinasDeEstaVecina, e) then                                     ▷ O(1)
15:                    muerteDelHippie(Siguiente(it).nombre, e)                                       ▷ O(|Nm|)
16:                    OtorgarCapturas(Siguiente(it).pos,e)                                           ▷ O(1)
17:                else
18:                    if TodosEstudiantes(vecinasDeEstaVecina, e) then
19:                        Definir(e.Estudiantes, Siguiente(it).nombre, Siguiente(it).pos)            ▷ O(|Nm|)
20:                        (e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1] ← (Estudiante, Siguiente(it).nombre) ▷ O(|Nm|)
21:                        muerteDelHippie(Siguiente(it).nombre, e)                                   ▷ O(|Nm|)
22:        else
23:            if Siguiente(it).tipo = Estudiante then                                                ▷ O(1)
24:                if todasLasVecinasOcupadas(vecinasDeEstaVecina, e) then                            ▷ O(1)
25:                    OtorgarSanciones(Siguiente(it).pos, e)                                         ▷ O(1)
26:        Avanzar(it)                                                                                ▷ O(1)
27:    if ¬ seEnhippiza then                                                                          ▷ O(1)
28:        if sinEscapatoria then                                                                     ▷ O(1)
29:            OtorgarSanciones(p, e)                                                                 ▷ O(1)
30:    else
31:        if seEnhippizaYQueda then                                                                  ▷ O(1)
32:            if sinEscapatoria then                                                                 ▷ O(1)
33:                OtorgarSanciones(p, e)                                                             ▷ O(1)
34:            SeHaceHippie(n,p,e)                                                                    ▷ O(|n|)
35:        else
36:            OtorgarSanciones(p, e)                                                                 ▷ O(1)
37:            OtorgarCapturas(p, e)                                                                  ▷ O(1)
38:            SeHaceHippie(n,p,e)                                                                    ▷ O(|n|)
39:            MuerteDelHippie(n,e)                                                                   ▷ O(|n|)
```

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

Justificación:

Línea 2: es la suma de Vecinos que es $\mathcal{O}(1)$, y vecinosVálidos?(Vecinos(p, e.campus), e.campus) que es $\mathcal{O}(\#Vecinos(p, e.campus)$=$\mathcal{O}(4)$.

Líneas 4, 5 y 6: #vecinasLegales ≤ 4.

Algorithm 34 IngresarEstudianteMágicamente2

1: **procedure** *i*IngresarEstudianteMágicamente2(**in** n : `nombre`,**in** p : `posición`, **in**/**out** e : `estr`)
2: Idea: Se chequea en el Mapa qué hay en las posiciones vecinas de la posición en la que estaría el estudiante. Como a lo sumo son 3 posiciones y acceder en el mapa es $\mathcal{O}(1)$, esto se hace en $\mathcal{O}(3)$. Al chequear esas posiciones, se anota en un booleano qué le pasaría al estudiante (convertirse en hippie y quedar hippie o desaparecer por haber agentes, o quedar estudiante) y si hubiera que agregar sanciones, capturas. En caso de que quedara estudiante o hippie habría que definirlo en el trie de estudiantes o de hippies según corresponda, en $\mathcal{O}(|Nm|)$; si se convirtiera en hippie y desapareciera habría que sumarle capturas a los agentes y sanciones porque si se convirtió en hippie fue porque estaba encerrado el estudiante (de las 3 posiciones vecinas, desaparece el hippie recién convertido porque una es de un agente y se convierte el estudiante en hippie porque 2 eran de hippies), y tenía un agente alrededor.
3: A la vez se chequea en el Mapa qué hay en las posiciones vecinas de las vecinas de p para determinar qué le pasaría a las cosas de las vecinas de p y a las vecinas de esas vecinas con la aparición del estudiante en p, con complejidad $\mathcal{O}(3x4)$. De todas formas, borrar o agregar hippies o estudiantes puede darse una cantidad de veces que es constante, y estas operaciones toman $\mathcal{O}(|Nm|)$.
4: Descripción: El estudiante n pasa a estar solamente en la posición p: se define en el diccionario de estudiantes de la estructura e al estudiante n en la posición p, ocupa la posición p en el mapa (si el estudiante ya estaba definido se obtiene su posición primero y después se desocupa en el mapa), y hace todos los cambios que tenga que hacer sobre el resto de la estructura por posibles transformaciones, mira qué les pasa a las cosas de las posiciones vecinas de p cuando un lugar más es bloqueado: un estudiante podria hacerse hippie, un hippie puede capturarse y habria que agregar capturas o hacerse estudiante (podria haber que modificar sanciones), etc..

Pre: $e =_{\text{obs}} e_0$
Post: ($\neg$ EsIngreso?(p, e.campus) $\wedge$ e = moverEstudiante(n, p, e_0)) $\vee$ (EsIngreso?(p, e.campus) $\wedge$ e = ingresarEstudiante(n, p, e_0)).

Algorithm 35 alMenos1Agente

1: **procedure** *i*AlMenos1Agente(**in** ps : `conj(posicion)`, **in** e : `estr`) $\rightarrow$ res : `bool`
2: itConj(posicion) it $\leftarrow$ CrearIt(ps) ▷ $\mathcal{O}(1)$
3: nat cantAgentes $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
4: **while** HaySiguiente(it) **do** ▷ $\mathcal{O}(1)$ x #ps
5: **if** (e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].tipo = Agente **then** ▷ $\mathcal{O}(1)$
6: cantAgentes $\leftarrow$ cantAgentes + 1 ▷ $\mathcal{O}(1)$
7: Avanzar(it) ▷ $\mathcal{O}(1)$
8: res $\leftarrow$ (cantAgentes $\geq$ 1) ▷ $\mathcal{O}(1)$
9: return res ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(\#ps)$
Justificación: $\mathcal{O}(\#ps)$
Descripción: Devuelve true si hay al menos un agente en el conjunto de posiciones ps.
Pre = {($\forall$ p : posición)($p \in ps \Rightarrow_{\text{L}}$posVálida?($p$,campus($c$))}
Post = {res = alMenos1Agente(ps,c)}

Algorithm 36 alMenos2Hippies

1: **procedure** *i*AlMenos2Hippies(**in** ps : `conj(posicion)`, **in** e : `estr`) $\rightarrow$ res : `bool`
2: itConj(posicion) it $\leftarrow$ CrearIt(ps) ▷ $\mathcal{O}(1)$
3: nat cantHippies $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
4: **while** HaySiguiente(it) **do** ▷ $\mathcal{O}(1)$ x #ps
5: **if** (e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].tipo = Hippie **then** ▷ $\mathcal{O}(1)$
6: cantHippies $\leftarrow$ cantHippies + 1 ▷ $\mathcal{O}(1)$
7: Avanzar(it) ▷ $\mathcal{O}(1)$
8: res $\leftarrow$ (cantHippies $\geq$ 1) ▷ $\mathcal{O}(1)$
9: return res ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(\#ps)$
Justificación: $\mathcal{O}(\#ps)$
Descripción: Devuelve true si hay al menos dos hippies en el conjunto de posiciones ps.
Pre = {($\forall$ p : posición)($p \in ps \Rightarrow_{\text{L}}$posVálida?($p$,campus($c$))}
Post = {res = (#posConHippies(ps,c) $\geq$ 2)}

Algorithm 37 todasLasVecinasOcupadas

```
1: procedure iTodasLasVecinasOcupadas(in ps : conj(posicion), in e : estr ) → res : bool
2:     itConj(posicion) it ← CrearIt(ps))                                   ▷ O(1)
3:     nat posLibres ← 0                                                    ▷ O(1)
4:     while HaySiguiente(it) do                                            ▷ O(1) x #ps
5:         if (e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].tipo = Libre then   ▷ O(1)
6:             posLibres ← posLibres + 1                                    ▷ O(1)
7:         Avanzar(it)                                                      ▷ O(1)
8:     res ← (posLibres = 0)                                                ▷ O(1)
9:     return res                                                           ▷ O(1)
```

Complejidad: $\mathcal{O}(\#ps)$
Justificación: $\mathcal{O}(\#ps)$
Descripción: Devuelve true si todas las posiciones de ps están ocupadas.
Pre = $\{(\forall\ p : \text{posición})(p \in ps \Rightarrow_{\text{L}} \text{posVálida?}(p,\text{campus}(c))\}$
Post = res = todasOcupadas?(ps,e).

Algorithm 38 todosEstudiantes

```
1: procedure iTodosEstudiantes(in ps : conj(posicion), in e : estr ) → res : bool
2:     itConj(posicion) it ← CrearIt(ps)                                    ▷ O(1)
3:     nat cantEstudiantes ← 0                                              ▷ O(1)
4:     while HaySiguiente(it) do                                            ▷ O(1) x #ps
5:         if (e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].tipo = Estudiante then   ▷ O(1)
6:             cantEstudiantes ← cantEstudiantes + 1                        ▷ O(1)
7:         Avanzar(it)                                                      ▷ O(1)
8:     res ← (cantEstudiantes = #ps)                                        ▷ O(1)
9:     return res                                                           ▷ O(1)
```

Complejidad: $\mathcal{O}(\#ps)$
Justificación: $\mathcal{O}(\#ps)$
Descripción: Devuelve true si todas las posiciones de ps están ocupadas por estudiantes.
Pre = $(\forall\ p : \text{posición})(p \in ps \Rightarrow_{\text{L}} \text{posVálida?}(p,\text{campus}(c))$
Post =
res = todosEstudiantes(ps,e)

Algorithm 39 muerteDelHippie

```
1: procedure iMuerteDelHippie(in n : nombre, in/out e : estr )
2:     posicion pos ← Obtener(e.Hippies, n)                          ▷ por copia, O(|n|)
3:     Borrar(e.Hippies,n)                                           ▷ O(|n|)
4:     (e.Mapa[pos.X-1])[pos.Y-1] ← Libre                            ▷ O(1)
```

Complejidad: $\mathcal{O}(|n|)$
Justificación: $\mathcal{O}(|n|)$
Descripción: Mata al hippie n.
Pre = $\{n \in \text{hippies}(c)\}$
Post = $\{n \notin \text{hippies}(c)\}$

Algorithm 40 seHaceHippie

```
1: procedure iseHaceHippie(in n : nombre, in p : posicion, in/out e : estr )
2:     if Definido(e.Estudiantes,n) then                             ▷ O(|n|)
3:         Borrar(e.Estudiantes,n)                                   ▷ O(|n|)
4:     Definir(e.Hippies,n,p)                                        ▷ O(|n|)
5:     (e.Mapa[p.X-1])[p.Y-1] ← (Hippie, n)                          ▷ O(|n|)
```

Complejidad: $\mathcal{O}(|n|)$
Justificación: $\mathcal{O}(|n|)$
Descripción: Si existe el estudiante n lo convierte en hippie, si no define al hippie n con la posición p.
Pre = {posVálida?(p, $e.campus$)}
Post = $\{n \in e.hippies \wedge$ posEstYHip(n,e) $= p\}$

Algorithm 41 otorgarSanciones

```
1: procedure iOtorgarSanciones(in p : posicion, in/out e : estr)
2:     it:itConj(posicion) ← CrearIt(vecinosVálidos?(Vecinos(p, e.campus), e.campus))   ▷ O(1)
3:     while HaySiguiente(it) do                                     ▷ O(1) * 4
4:         if (e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].agente.oc then   ▷ O(1)
5:             ag : agente ← ((e.Mapa[Siguiente(it).X-1])[Siguiente(it).Y-1].agente.p)− > agente   ▷ O(1)
6:             tupla:<agente,posicion,nat,nat,itConj(agente),itLista(<conj(agente),nat>)>←
   Significado(e.AgentesD, ag)                                       ▷ O(1)
7:             Π4(tupla) ← Π4(tupla) + 1                             ▷ O(1)
8:             EliminarSiguiente(Π5(tupla))                          ▷ O(1)
9:             if Siguiente(Π6(tupla)).sanc = Π4(tupla) then         ▷ O(1)
10:                Π5(tupla) ← AgregarRapido(Siguiente(Π6(tupla)).conj, ag)   ▷ O(1)
11:                Avanzar(Π6(tupla))                                ▷ O(1)
12:            else
13:                nuevoNodo:<conj(agente),nat>← <Vacio(), Π4(tupla)>   ▷ O(1)
14:                Π5(tupla) ← AgregarRapido(Π1(nuevoNodo), ag)      ▷ O(1)
15:                AgregarComoSiguiente(Π6(tupla), nuevoNodo)        ▷ O(1)
16:                Avanzar(Π6(tupla))                                ▷ O(1)
17:        Avanzar(it)                                               ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificación: Dentro del while se realizan operaciones con costo $\mathcal{O}(1)$, y se realizan en el peor de los casos 4 veces (una vez por cada posición adyacente a p). Por álgebra de órdenes, $\mathcal{O}(1)$ * 4 = $\mathcal{O}(4)$ = $\mathcal{O}(1)$
Descripción: Otorga una sanción a cada agente adyacente a la posición p.

Pre = $\{e =_{\text{obs}} e_0 \wedge$ posVálida(p, campus(c))$\}$
Post = $\{(\forall\ t$:posicion)((t ∈ vecinosValidos?(vecinos(p,$e.campus$),$e.campus$) ∧ ($\exists$ a:agente) (posAgente(a, c) = t)) $\Rightarrow_{\text{L}}$ cantSanciones(buscarAgente(t, agentes(e), e), e) = cantSanciones(buscarAgente(t, agentes(e_0), e_0), e_0) + 1)$\}$

Algorithm 42 otorgarCapturas

1: **procedure** *i*OTORGARCAPTURAS(**in** p : `posicion`, **in/out** e : `estr`)
2: it:itConj(posicion) ← CrearIt(vecinosVálidos?(Vecinos(p, $e.campus$), $e.campus$)) ▷ $\mathcal{O}(1)$
3: **while** HaySiguiente(it) **do** ▷ $\mathcal{O}(1)$ * 4
4: **if** ($e.Mapa$[Siguiente(it).X-1])[Siguiente(it).Y-1].$agente.oc$ **then** ▷ $\mathcal{O}(1)$
5: ag : agente ← (($e.Mapa$[Siguiente(it).X-1])[Siguiente(it).Y-1].$agente.p$)− > $agente$ ▷ $\mathcal{O}(1)$
6: $tupla$:<posicion,nat,nat,itConj(agente),itLista(<conj(agente),nat>)>← Significado($e.AgentesD$, ag) ▷ $\mathcal{O}(1)$
7: $\Pi_3(tupla) \leftarrow \Pi_3(tupla) + 1$ ▷ $\mathcal{O}(1)$
8: **if** $\Pi_3(tupla) > e.masVigilante.cant$ **then** ▷ $\mathcal{O}(1)$
9: $e.masVigilante.cant \leftarrow \Pi_3(tupla)$ ▷ $\mathcal{O}(1)$
10: $e.masVigilante.ag \leftarrow ag$ ▷ $\mathcal{O}(1)$
11: **else**
12: **if** $\Pi_3(tupla) = e.masVigilante.cant \wedge ag < e.masVigilante.ag$ **then** ▷ $\mathcal{O}(1)$
13: $e.masVigilante.ag \leftarrow ag$ ▷ $\mathcal{O}(1)$
14: Avanzar(it) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: Dentro del while se realizan operaciones con costo $\mathcal{O}(1)$, y se realizan en el peor de los casos 4 veces (una vez por cada posición adyacente a p). Por álgebra de órdenes, $\mathcal{O}(1)$ * 4 = $\mathcal{O}(4)$ = $\mathcal{O}(1)$
Descripción: Otorga 1 captura a cada agente adyacente a la posición p.

Pre = $\{c =_{\text{obs}} c_0 \wedge$ posVálida(p, campus(c))$\}$
Post = $\{(\forall\ t$:posicion)(($t \in$ vecinosValidos?(vecinos(p,$e.campus$),$e.campus$) $\wedge$ ($\exists\ a$:agente) (posAgente(a, e) = t)) $\Rightarrow_{\text{L}}$ cantHippiesAtrapados(buscarAgente(t, agentes(e), e), e) = cantHippiesAtrapados(buscarAgente(t, agentes(e_0), e_0), c_0) + 1)$\}$

Algorithm 43 MoverEstudiante

1: **procedure** *i*MOVERESTUDIANTE(**in** n : `nombre`,**in** d : `dirección`,**in/out** e : `estr`)
2: posicion posActual ← obtener(e.estudiantes,n) ▷ por copia $\mathcal{O}(|Nm|)$
3: Borrar(e.Estudiantes, n) ▷ $\mathcal{O}(|n|)$
4: (e.Mapa[posActual.X-1])[posActual.Y-1] ← Libre ▷ $\mathcal{O}(1)$
5: **if** posVálida?(proxPosición(posActual,d,e.campus), e.campus) **then** ▷ $\mathcal{O}(1)$
6: IngresarEstudianteMágicamente(n, posResultante, e) ▷ $\mathcal{O}(|n|)$

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

Algorithm 44 posicionesDeObjetivos

```
1:  procedure iPosicionesDeObjetivos(in i : identificacion,in e : estr ) → res : conj(posicion)
2:      conj(posicion) res ← vacio()                                                    ▷ O(1)
3:      if Definido?(e.Hippies,i) then                                                  ▷ O(|i|) ≤ O(|Nm|)
4:          itConj(nombre) it ← CrearItClaves(e.Estudiantes)                            ▷ O(1)
5:          while HaySiguiente(it) do                  ▷ O(1) x #Claves(e.Estudiantes) = O(Ne)
6:              AgregarRapido(res,SiguienteSignificado(it))                             ▷ O(1)
7:              Avanzar(it)                                                             ▷ O(1)
8:      else
9:          itConj(nombre) it ← CrearItClaves(e.Hippies)                                ▷ O(1)
10:         while HaySiguiente(it) do                  ▷ O(1) x #Claves(e.Hippies) = O(Nh)
11:             AgregarRapido(res,SiguienteSignificado(it))                             ▷ O(1)
12:             Avanzar(it)                                                             ▷ O(1)
13:     return res                                                                      ▷ O(1)
```

Complejidad: Si i es un hippie $\mathcal{O}(|Nm|) + \mathcal{O}(N_e)$, si i es un agente $\mathcal{O}(|Nm|) + \mathcal{O}(N_h)$
Justificación: Si i es un hippie, tiene que recorrer todo el conjunto de estudiantes, y eso es $\mathcal{O}(N_e)$. Por el contrario, si i es un agente, el algoritmo recorre todos los hippies, y eso es $\mathcal{O}(N_h)$.
AgregarRapido los significados, que son posiciones, o sea duplas, es $\mathcal{O}(copy(significado)) = \mathcal{O}(1)$ porque las posiciones se copian en $\mathcal{O}(1)$, y SiguienteSignificado(it) es $\mathcal{O}(1)$.
Pre = $\{i \in$ hippies$(c) \vee i \in$ agentes$(c)\}$
Post = $\{(i \in$ hippies$(c) \wedge (\forall$ p:posicion$)$ $(p \in res \Leftrightarrow (\exists$ n:nombre$)$ (Definido$(e.Estudiantes,n) \Rightarrow_{\text{L}}$ Obtener$(e.Estudiantes,n) = p))) \vee$
$(i \in$ agentes$(c) \wedge (\forall$ p:posicion$)$ $(p \in res \Leftrightarrow (\exists$ n:nombre$)$ (Definido$(e.Hippies,n) \Rightarrow_{\text{L}}$ Obtener$(e.Hippies,n) = p)))\}$

Algorithm 45 MoverHippie

```
1:  procedure iMoverHippie(in n : nombre,in/out e : estr )
2:      posicion posActual ← Significado(e.Hippies,n)                          ▷ O(|n|) ≤ O(|Nm|)
3:      if HaySiguiente?(CrearItClaves(e.Estudiantes)) then                    ▷ O(1)
4:          nat mínimaDistancia ← distancia(posActual,PosicionMinimaDistancia(posActual,
5:          PosicionesDeObjetivo(n,e),e.campus),e.campus)                      ▷ O(|Nm|) + O(Ne)
6:          itConj(posicion) it3 ← CrearIt(objetivosDeMínimaDistancia(posActual,
7:          PosicionesDeObjetivo(n,e),mínimaDistancia,e.campus))               ▷ O(|Nm|) + O(Ne)
8:          itConj(direccion) it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))   ▷ O(1)
9:          posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)          ▷ O(1)
10:         while HaySiguiente(it3) ∧ Ocupada?(posResultante,e.campus) do      ▷ O(1) x O(Ne)
11:             it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))   ▷ O(1)
12:             while HaySiguiente(it) ∧ Ocupada?(ProxPosicion(posActual,Siguiente(it),e.campus)) do   ▷ O(1)
13:                 Avanzar(it)                                                ▷ O(1)
14:             posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)   ▷ O(1)
15:             Avanzar(it3)                                                   ▷ O(1)
16:     else
17:         conj(posicion) ingresosConEsteX ← Vacio()                          ▷ O(1)
18:         AgregarRapido(ingresosConEsteX, <p.X,1>)                           ▷ O(1)
19:         AgregarRapido(ingresosConEsteX, <p.X,e.campus.filas>)              ▷ O(1)
20:         nat mínimaDistancia ← distancia(posActual,PosicionMinimaDistancia(posActual,
21:         ingresosConEsteX,e.campus),e.campus)                               ▷ O(1)
22:         itConj(posicion) it3 ← CrearIt(objetivosDeMínimaDistancia(posActual,
23:         ingresosConEsteX,mínimaDistancia,e.campus))                        ▷ O(1)
24:         itConj(direccion) it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))   ▷ O(1)
25:         posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)          ▷ O(1)
26:         while HaySiguiente(it3) ∧ Ocupada?(posResultante,e.campus) do      ▷ O(1)
27:             it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))   ▷ O(1)
28:             while HaySiguiente(it) ∧ Ocupada?(ProxPosicion(posActual,Siguiente(it),e.campus)) do   ▷ O(1)
29:                 Avanzar(it)                                                ▷ O(1)
30:             posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)   ▷ O(1)
31:             Avanzar(it3)                                                   ▷ O(1)
32:     if ¬ Ocupada?(posResultante,e.campus) then                             ▷ O(1)
33:         MuerteDelHippie(n,e)                                               ▷ O(|n|) ≤ O(|Nm|)
34:         IngresarHippieMagicamente(n,posResultante,e)                       ▷ O(|Nm|)
```

Complejidad: $\mathcal{O}(|Nm|)+ \mathcal{O}(|Ne|)$.
Justificación:
Línea 8: |n| ≤ |Nm| porque el hippie n ya existía y |Nm| era considerando al nombre n.
PosicionesDeObjetivo(n,e) es $\mathcal{O}(|Nm|) + \mathcal{O}(N_e)$ dado que n es un hippie.
PosicionMinimaDistancia(posActual,PosicionesDeObjetivo(n,e),e.campus) es $\mathcal{O}(\#PosicionesDeObjetivo(n,e)) + (\mathcal{O}(|Nm|) + \mathcal{O}(N_e)) = (\mathcal{O}(|Nm|) + \mathcal{O}(N_e))$ dado que hay N_e objetivos, que son los estudiantes.

Algorithm 46 MoverAgente

```
1: procedure iMoverAgente(in a : agente, in/out e : estr )
2:     posicion posActual ← posAgente(a,e)                                   ▷ O(1) en caso promedio
3:     if HaySiguiente?(CrearItClaves(e.Hippies)) then                       ▷ O(1)
4:         nat mínimaDistancia ← distancia(posActual,PosicionMinimaDistancia(posActual,
5:         PosicionesDeObjetivo(a,e),e.campus),e.campus)                     ▷ O(|Nm|) + O(Nh)
6:         itConj(posicion) it3 ← CrearIt(objetivosDeMínimaDistancia(posActual,
7:         PosicionesDeObjetivo(a,e),mínimaDistancia,e.campus))              ▷ O(|Nm|) + O(Nh)
8:         itConj(direccion) it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))   ▷ O(1)
9:         posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)          ▷ O(1)
10:        while HaySiguiente(it3) ∧ Ocupada?(posResultante,e.campus) do     ▷ O(1) x O(Nh)
11:            it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))  ▷ O(1)
12:            while HaySiguiente(it) ∧ Ocupada?(ProxPosicion(posActual,Siguiente(it),e.campus)) do   ▷ O(1)
13:                Avanzar(it)                                                ▷ O(1)
14:            posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)   ▷ O(1)
15:            Avanzar(it3)                                                   ▷ O(1)
16:    else
17:        conj(posicion) ingresosConEsteX ← Vacio()                          ▷ O(1)
18:        AgregarRapido(ingresosConEsteX, <p.X,1>)                           ▷ O(1)
19:        AgregarRapido(ingresosConEsteX, <p.X,e.campus.filas>)              ▷ O(1)
20:        nat mínimaDistancia ← distancia(posActual,PosicionMinimaDistancia(posActual,
21:        ingresosConEsteX,e.campus),e.campus)                               ▷ O(1)
22:        itConj(posicion) it3 ← CrearIt(objetivosDeMínimaDistancia(posActual,
23:        ingresosConEsteX,mínimaDistancia,e.campus))                        ▷ O(1)
24:        itConj(direccion) it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))   ▷ O(1)
25:        posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)          ▷ O(1)
26:        while HaySiguiente(it3) ∧ Ocupada?(posResultante,e.campus) do      ▷ O(1)
27:            it ← CrearIt(EnQuéDireccionesVoy(p,Siguiente(it3),e.campus))   ▷ O(1)
28:            while HaySiguiente(it) ∧ Ocupada?(ProxPosicion(posActual,Siguiente(it),e.campus)) do   ▷ O(1)
29:                Avanzar(it)                                                ▷ O(1)
30:            posicion posResultante ← ProxPosicion(posActual,Siguiente(it),e.campus)   ▷ O(1)
31:            Avanzar(it3)                                                   ▷ O(1)
32:    if ¬ Ocupada?(posResultante,e.campus) then                             ▷ O(1)
33:        punterito ← e.Mapa[posActual.X-1])[posActual.Y-1].agente.p         ▷ O(1)
34:        (e.Mapa[posActual.X-1])[posActual.Y-1] ← Libre                     ▷ O(1)
35:        punterito → posicion ← posResultante                               ▷ O(1)
36:        (e.Mapa[posResultante.X-1])[posResultante.Y-1] ← (a, punterito)    ▷ O(1)
37:        conj(posicion) vecinasLegales ← vecinosVálidos?(Vecinos(posResultante, e.campus), e.campus)   ▷ O(1)
38:        itConj(posicion) it2 ← CrearIt(vecinasLegales)                     ▷ O(1)
39:        while HaySiguiente(it2) do          ▷ O(1) x #vecinasLegales ≤ O(1) x 4 = O(1)
40:            conj(posicion) vecinasDeEstaVecina ← vecinosVálidos?
41:            (Vecinos(Siguiente(it2), e.campus), e.campus)                  ▷ O(1)
42:            if Siguiente(it2).tipo = Hippie then                           ▷ O(1)
43:                if todasLasVecinasOcupadas(vecinasDeEstaVecina, e) then    ▷ O(1)
44:                    if alMenos1Agente(vecinasDeEstaVecina, e) then         ▷ O(1)
45:                        muerteDelHippie(Siguiente(it2).nombre, e)          ▷ O(|Nm|)
46:                        OtorgarCapturas(Siguiente(it2).pos,e)              ▷ O(1)
47:            else
48:                if Siguiente(it2).tipo = Estudiante then                   ▷ O(1)
49:                    if todasLasVecinasOcupadas(vecinasDeEstaVecina, e) then   ▷ O(1)
50:                        OtorgarSanciones(Siguiente(it2), e)                ▷ O(1)
51:            Avanzar(it)                                                    ▷ O(1)
```

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

Justificación:

PosicionesDeObjetivo(a,e) es $\mathcal{O}(|Nm|) + \mathcal{O}(N_h)$ dado que a es un agente.

PosicionMinimaDistancia(posActual,PosicionesDeObjetivo(n,e),e.campus) es $\mathcal{O}(\#PosicionesDeObjetivo(a, e)) + (\mathcal{O}(|Nm|) + \mathcal{O}(N_h)) = (\mathcal{O}(|Nm|) + \mathcal{O}(N_h))$ dado que hay N_h objetivos, que son los hippies.

Algorithm 47 MásVigilante

1: **procedure** *i*MÁSVIGILANTE(**in** e: `estr`) → res : `nat`
2: $res \leftarrow$ e.MásVigilante.ag ▷ $\mathcal{O}(1)$

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

Algorithm 48 cantEstudiantes

1: **procedure** *i*CANTESTUDIANTES(**in** e: `estr`) → res : `nat`
2: $res \leftarrow$ Cardinal(Claves($e.estudiantes$)) ▷ $\mathcal{O}(|N_e|)$

Complejidad: $\mathcal{O}(|N_e|)$
Justificación: La función Claves del diccTrie tiene complejidad $\mathcal{O}(|N_e|)$

Algorithm 49 cantHippies

1: **procedure** *i*CANTHIPPIES(**in** e: `estr`) → res : `nat`
2: $res \leftarrow$ Cardinal(Claves($e.hippies$)) ▷ $\mathcal{O}(|N_h|)$

Complejidad: $\mathcal{O}(|N_h|)$
Justificación: La función Claves del diccTrie tiene complejidad $\mathcal{O}(|N_h|)$

Algorithm 50 todasOcupadas?

1: **procedure** *i*TODASOCUPADAS?(**in** cp: `conj(posición)` ,**in** e: `estr`) → res : `bool`
2: it:itConj(posicion) ← crearIt(cp) ▷ $\mathcal{O}(1)$
3: chequeo:bool ← true ▷ $\mathcal{O}(1)$
4: pos : posición ▷ $\mathcal{O}(1)$
5: **while** haySiguiente(it) **do** ▷ $\mathcal{O}(n)$
6: pos ← e.mapa[Siguiente(it).X][Siguiente(it).Y] ▷ $\mathcal{O}(1)$
7: **if** ¬(pos.est ∨ pos.hip ∨ π_1(pos.agente) ∨ pos.obstaculo) **then** ▷ $\mathcal{O}(1)$ chequeo ← false ▷ $\mathcal{O}(1)$
res ← chequeo ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(n)$
Justificación: Recorre todo el conjunto. La complejidad es $\mathcal{O}(n)$, donde n es la cantidad de elementos del conjunto cp
Pre = {(∀ p : posición)(p ∈ cp ⇒$_{\text{L}}$ posVálida?(p, e.campus)}
Post = {$res =_{\text{obs}}$ todasOcupadas?(cp, ê)}

Algorithm 51 busquedaBinaria

1: **procedure** *i*BUSQUEDABINARIA(**in** a: `vector(<itConj(agente),nat>)` **in** e: `nat`)→ res : `nat`
2: res ← busquedaDC(a, e, 0, longitud(a)-1) ▷ $\Theta(log(n))$

Complejidad: $\Theta(log(n))$
Justificación: La complejidad de busquedaDC es $\Theta(log(n))$

Algorithm 52 busquedaDC

1: **procedure** *i*BUSQUEDADC(**in** a: `vector(<itConj(agente),nat>)` **in** e: `nat`,**in** $bajo$: `nat` **in** $alto$: `nat`)→ res : `nat`
2: **if** alto=bajo **then** ▷ $\mathcal{O}(1)$
3: **if** a[alto].sanc=e **then** ▷ $\mathcal{O}(1)$
4: res ← alto ▷ $\mathcal{O}(1)$
5: medio ← (bajo+alto)/2 ▷ $\mathcal{O}(1)$
6: **if** a[medio].sanc<e **then** ▷ $\mathcal{O}(1)$
7: res← busquedaDC(a,e,medio+1,alto) ▷ $T(n/2)$
8: **else**
9: res← busquedaDC(a,e,bajo,medio) ▷ $T(n/2)$

Complejidad: $\mathcal{O}(log(n))$
Justificación: La justificación de la complejidad de la búsqueda binaria fue dada en clase. Básicamente parte la entrada en dos partes iguales, y se llama recursivamente sólo en una de ellas, entonces tenemos a = 1 y c = 2. Dividir y juntar los resultados son $\mathcal{O}(1)$. T(n) = T(n/2) + $\mathcal{O}(1)$. Usando el segundo caso del teorema maestro, nos queda que la complejidad de la búsqueda binaria es $\Theta(n^{log_2(1)} * log(n)) = \Theta(log(n))$

3. Módulo DiccTrie

Interfaz

parámetros formales
géneros σ

se explica con: DICCIONARIO($string, \sigma$).

géneros: `diccT(`σ`)`, `itClavesdiccT(`σ`)`.

Operaciones básicas Diccionario Trie(σ)

VACÍO() $\rightarrow$ *res* : `diccT(`σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacío()}
Complejidad: $\mathcal{O}(1)$
Descripción: genera un diccionario vacío.

DEFINIR(**in/out** d: `diccT(`σ`)`, **in** k: *string*, **in** s: σ)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(k, s, d_0)}
Complejidad: $\mathcal{O}(|k|)$
Descripción: define la clave k con el sinificado s.
Aliasing: s se define por referencia.

DEFINIDO?(**in** d: `diccT(`σ`)`, **in** k: *string*) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ def?(k, d) }
Complejidad: $\mathcal{O}(|k|)$
Descripción: devuelve *true* si la clave k está definida en el diccionario.

OBTENER(**in** d: `diccT(`σ`)`, **in** k: *string*) $\rightarrow$ *res* : σ
Pre $\equiv$ {def?(k, d)}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ obtener(k, d))}
Complejidad: $\mathcal{O}(|k|)$,
Descripción: devuelve el significado de la clave k en d.
Aliasing: *res* es modificable si y sólo si d lo es

CLAVES(**in** d: `diccT(`σ`))` $\rightarrow$ *res* : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ claves(d)}
Complejidad: $\Theta(|k|)$
Descripción: devuelve un conjunto con todas las claves del diccionario.

BORRAR(**in/out** d: `diccT(`σ`)`, **in** k: *string*)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge$ def?(k, d_0) }
Post $\equiv$ {$d =_{\text{obs}}$ borrar(k, d_0)}
Complejidad: $\mathcal{O}(|k|)$
Descripción: elimina la entrada k del diccionario.

4. Operaciones del iterador

CREARITCLAVES(**in** d: `diccT(`σ`))` $\rightarrow$ *res* : `itdiccT(`σ`)`
Pre $\equiv$ {true}

Post ≡ {alias(esPermutación?(SecuSuby(res), clavesExtendidas(d))) ∧ vacia?(Anteriores(res))}
Complejidad: $\mathcal{O}(1)$
Descripción: crea un iterador del diccionario, de forma que pueda ser recorrido completamente aplicando iterativamente Siguiente.

HaySiguiente?(**in** it : `itdiccT(`σ`)`) → res : `bool`
Pre ≡ {true}
Post ≡ {res =$_{\text{obs}}$ haySiguiente(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve *true* si y sólo si quedan elementos para iterar.

SiguienteClave(**in** it : `itdiccT(`σ`)`) → res : `string`
Pre ≡ {haySiguiente?(it)}
Post ≡ {alias(res =$_{\text{obs}}$ siguiente(it).clave)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve la clave de la entrada del diccionario apuntada por el iterador.
Aliasing: res no es modificable.

SiguienteSignificado(**in** it : `itdiccT(`σ`)`) → res : σ
Pre ≡ {haySiguiente?(it)}
Post ≡ {alias(res =$_{\text{obs}}$ siguiente(it).significado)}
Complejidad: $\mathcal{O}(1)$
Descripción: devuelve el significado de la entrada del diccionario apuntada por el iterador.
Aliasing: res no es modificable.

Avanzar(**in/out** it : `itdiccT(`σ`)`)
Pre ≡ {it =$_{\text{obs}}$ it_0∧ haySiguiente?(it_0)}
Post ≡ {it =$_{\text{obs}}$ avanzar(it_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: avanza a la posición siguiente del iterador.

HayMas?(**in** d : `itClaves(`*string*`)`) → res : *bool*
Pre ≡ {true}
Post ≡ {res =$_{\text{obs}}$ hayMas?(it)}
Complejidad: O(*longitud*(*secuSuby*(d)))
Descripción: Informa si hay más elementos por iterar.

Actual(**in** d : `itClaves(`*string*`)`) → res : *string*
Pre ≡ {true}
Post ≡ {res =$_{\text{obs}}$ actual(it)}
Complejidad: O(*longitud*(*secuSuby*(d)))
Descripción: Devuelve la clave de la posición actual.

Avanzar(**in/out** it : `itClaves(`*string*`)`)
Pre ≡ {hayMas?(it) ∧ $it = it_0$}
Post ≡ {it =$_{\text{obs}}$ avanzar(it_0)}
Complejidad: O(*longitud*(*secuSuby*(d)))
Descripción: Avanza a la próxima clave.

4.1. Representación

4.1.1. Representación del Diccionario Trie(α)

`diccT`(α) **se representa con** `estr`

donde `estrDic` es `tupla`(*raiz*: `puntero(nodo)` , *claves*: `lista`(*string*))

`Nodo` **se representa con** `estrNodo`

donde `estr` es `tupla`(*valor*: `puntero`(α)
, *clave*: `itLista`(*string*)
, *hijos*: `arreglo_estático[256](puntero(nodo))`)

4.1.2. Invariante de Representación de diccT

(I) Existe un único camino entre cada nodo y el nodo raíz (no hay ciclos).

(II) Todos los nodos hojas, es decir, todos los que tienen su arreglo hijos con todas sus posiciones en NULL, tienen que tener un valor distinto de NULL.

(III) Raíz es distinto de NULL

(IV) En claves está el camino que se recorre desde la raź hasta cada nodo hoja.

Rep : estrDic $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$
(raíz = NULL $\wedge$ esVacia?($e.claves$)) $\vee_{\text{L}}$raíz $\neq NULL \wedge_{\text{L}}$ $noHayCiclos(e) \wedge$ todasLasHojasTienenValor(e) $\wedge$
hayHojas(e) $\Leftrightarrow$ |e.claves|>0 $\wedge$
($\forall$ c : string)(c $\in$ caminosANodos(e)) $\Rightarrow$ $\big((\exists$ i: nat)(i $\in \{0..|\text{e.claves}|\}) \Leftrightarrow$ (e.claves[i] = c)$\big)$

4.1.3. Operaciones auxiliares del invariante de Representación

noHayCiclos : puntero(nodo) $\longrightarrow$ bool
noHayCiclos(n, p) $\equiv$ ($\exists$ n:nat)(($\forall$ c: string)(|s| = n $\Rightarrow$ leer(p, s) = NULL))
leer : puntero(nodo) $\times$ string $\longrightarrow$ bool
leer(p, s) $\equiv$ **if** vacia?(s) **then**
p $\rightarrow$ valor
else
if p $\rightarrow$ hijos[prim(s)] = NULL **then** NULL **else** leer(p $\rightarrow$ hijos[prim(s)], fin(s)) **fi**
fi
todosNull : arreglo(puntero(nodo)) $\longrightarrow$ bool
todosNull(a) $\equiv$ auxTodosNull($a, 0$)
auxTodosNull : arreglo(puntero(nodo)) $\times$ nat $\longrightarrow$ bool
auxTodosNull(a, i) $\equiv$ **if** i <|a| **then** a[i] == NULL $\wedge$ auxTodosNull($a, i+1$) **else** a[i].valor == NULL **fi**
esHoja : puntero(nodo) $\longrightarrow$ bool
esHoja(p) $\equiv$ **if** p == NULL **then** false **else** todosNull(p.hijos) **fi**
todasLasHojas : puntero(nodo) $\times$ nat $\longrightarrow$ conj(nodo)
todasLasHojas(p, n) $\equiv$ **if** p == NULL **then**
false
else
if esHoja(p) **then** Ag(*p, vacio) **else** auxTodasLasHojas((*p).hijos, 256) **fi**
fi
auxTodasLasHojas : arreglo(puntero(nodo)) $\times$ nat $\longrightarrow$ conj(nodo)
auxTodasLasHojas(a, n) $\equiv$ hojasDeHijos($a, n, 0$)
hojasDeHijos : arreglo(puntero(nodo)) $\times$ nat $\times$ nat $\longrightarrow$ conj(nodo)
hojasDeHijos(a, n, i) $\equiv$ **if** i = n **then** $\emptyset$ **else** todasLasHojas(a[i]) $\cup$ hojasDeHijos($a, n, (i+1)$) **fi**
todasLasHojasTienenValor : puntero(nodo) $\longrightarrow$ bool
todasLasHojasTienenValor(p) $\equiv$ auxTodasLasHojasTienenValor(todasLasHojas($p, 256$))

auxTodasLasHojasTienenValor : arreglo(puntero(nodo)) $\longrightarrow$ bool
auxTodasLasHojasTienenValor(a) $\equiv$ **if** |a| = 0 **then**
true
else
dameUno(a).valor != NULL $\wedge$ auxTodasLasHojasTienenValor(sinUno(a))
fi
hayHojas : puntero(nodo) $\longrightarrow$ bool
Dada una estructura, indica si en la misma existe algún nodo cuyo arreglo hijos tenga todas las posiciones NULL.
caminosANodos : puntero(nodo) $\longrightarrow$ conj(string)
Dada una estructura, devuelve un conjunto con el único camino existente entre la raiz y cada hoja. El camino se obtiene de agregar la posición del arreglo hijos por la cual hay que bajar en cada nivel de la estructura hasta llegar a la hoja, conviritendo en cada paso esa posición en un CHAR y juntándolos en un STRING.

4.1.4. Función de abstracción de diccT

Abs : estrDicc e $\longrightarrow$ dicc(string,α) {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: dicc(string,α) | ($\forall$ c:string)(definido?(c, d)) = ($\exists$ n: nodo)(n $\in$ todasLasHojas(e)) n.valor != NULL
$\wedge$ ($\exists$ i:nat)(i $\in$ {0..|e.claves|}) $\Rightarrow$ e.claves[i] = c $\wedge_{\text{L}}$ significado(c, d) = leer($e.clave$).valor

4.1.5. Representación del iterador de Claves del Diccionario Trie(α)

`itClaves(`*string*`)` **se representa con** `puntero(nodo)`

Su Rep y Abs son los de itSecu(α) definido en el apunte de iteradores.

4.2. Algoritmos

4.2.1. Algoritmos de Diccionario Trie

```
1: function iVacia → res: estrDicc(α)
2:     res.claves ← Vacia()                                   ▷ O(1)
3:     Nodo n ← crearNodo(res)                                ▷ O(1)
4:     res.raiz ← *n                                          ▷ O(1)
   Complejidad: 3*O(1) = O(1)
```

```
1: function iCrearNodo(in e : estr) → res: nodo
2:     d : arreglo_estático[256]                              ▷ O(1)
3:     i ← 0                                                  ▷ O(1)
4:     while i <256 do                                        ▷ 256*
5:         d[i] ← NULL                                        ▷ O(1)
6:         i++
7:     res.hijos ← d                                          ▷ O(1)
8:     res.valor ← NULL                                       ▷ O(1)
9:     res.clave ← CrearItUl(e.claves)                        ▷ O(1)
10: Complejidad:2*O(1) + 256*O(1) + 2*O(1) = O(1)
```

```
1: function iDefinir(in/out d: estrDicc(α), in c: String, in s: α)
2:     i ← 0                                                          ▷ O(1)
3:     p ← d.raiz                                                     ▷ O(1)
4:     while i <(longitud(c)) do                                      ▷ O(|c|)
5:         if p.hijos[ord(c[i])] == NULL then                         ▷ O(1)
6:             n: nodo ← crearNodo()                                  ▷ O(1)
7:             p.hijos[ord(c[i])] ← *n                                ▷ O(1)
8:             p.valor ← NULL                                         ▷ O(1)
9:         p ← p.hijos[ord(c[i])]                                     ▷ O(1)
10:        i++                                                        ▷ O(1)
11:    p.valor ← &s                                                   ▷ O(1)
12:    agregarAtras(d.claves, c)                                      ▷ O(1)
13:    p.clave ← CrearItUl(d.claves)                                  ▷ O(1)
14: Complejidad: 2*O(1) + |c|*5*O(1) + O(1) + O(|c|) = O(|c|)
```

```
1: function iSignificado(in d: estrDicc(α), in c: string) → res: α
2:     i ← 0                                                          ▷ O(1)
3:     p ← d.raiz                                                     ▷ O(1)
4:     while i <(longitud(s)) do                                      ▷ |s|*
5:         p ← p.hijos[ord(s[i])]                                     ▷ O(1)
6:         i++                                                        ▷ O(1)
7:     res ← p.valor                                                  ▷ O(1)
8: Complejidad: 2*O(1) + |s|*2*O(1) + O(1) = O(|s|)
```

```
1: function iDef?(in d: estrDicc(α), in c: string) → res: bool
2:     i ← 0                                                          ▷ O(1)
3:     p ← d.raiz                                                     ▷ O(1)
4:     while i <(longitud(s)) do                                      ▷ |s|*
5:         if (p.hijos[ord(s[i])] != NULL) then                       ▷ O(1)
6:             p ← p.hijos[ord(s[i])]                                 ▷ O(1)
7:             i++                                                    ▷ O(1)
8:         else
9:             return false                                           ▷ O(1)
10:    res ← p.valor != NULL                                          ▷ O(1)
11: Complejidad: 2*O(1) + |s|*3*O(1) + O(1) = O(|s|)
```

```
1: function iClaves(in d: estrDicc(α)) → res: itClaves(string)
2:     res ← CrearIt(d.claves)                                        ▷ O(1)
3: Complejidad: O(1)
```

1: **function** iBORRAR(**in**/**out** d: `estrDicc(`α`)`, **in** k: `string`)
2: i : nat $\leftarrow 0$ ▷ $\mathcal{O}(1)$
3: p : puntero(nodo) $\leftarrow d.raiz$ ▷ $\mathcal{O}(1)$
4: **while** $i <$ longitud(k) **do** ▷ $\mathcal{O}(|k|)$
5: $p \leftarrow p.hijos$[ord(k[i])] ▷ $\mathcal{O}(1)$
6: $i \leftarrow i + 1$ ▷ $\mathcal{O}(1)$
7: $p.valor \leftarrow$ NULL ▷ $\mathcal{O}(1)$
8: EliminarAnterior($p.clave$) ▷ $\mathcal{O}(1)$
9: **Complejidad:** $\mathcal{O}(|k|)$
10: **Justificación:** El while recorre toda la palabra, y las operaciones que hace son todas $\mathcal{O}(1)$, así que la complejidad queda $\mathcal{O}(|k|)$, donde $|k|$ denota la longitud de la palabra k

4.2.2. Algoritmos del iterador de claves del Diccionario String

Utiliza los mismos algoritmos que itSecu(α) definido en el apunte de iteradores.

4.3. Servicios Usados

Módulo	**Operación**	**Complejidad Requerida**
arreglo_estático	AgregarElemento	O(1)
arreglo_estático	•[•]	O(1)
lista	AgregarAdelante	O(copy(α))
lista	•[•]	O(1)

5. Modulo DiccHash(nat,α)

Interfaz

se explica con: DICCIONARIO(NAT, σ)

generos: `diccHash(nat, α)`

CREAR(**in** n: `nat`) $\rightarrow$ res : `diccHash(nat, α)`
Pre $\equiv$ $\{n > 0\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{vacío}()\}$
Complejidad: $\mathcal{O}(n)$
Descripción: Crea un diccionario vacío de n posiciones.
Aliasing: No hay aspectos de aliasing.

DEFINIR(**in/out** d: `diccHash(nat,α)`, **in** n: `nat`, **in** a: α)
Pre $\equiv$ $\{d =_{\text{obs}} d_0 \wedge \neg\ \text{def?}(a, d)\}$
Post $\equiv$ $\{d =_{\text{obs}} \text{definir}(n, a, d_0)\}$
Complejidad: $\mathcal{O}(1)$ en caso promedio
Descripción: Define la clave n con el significado a en el diccionario d.
Aliasing: No hay aspectos de aliasing.

DEFINIDO?(**in** d: `diccHash(nat,α)`, **in** n: `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{def?}(n, d)\}$
Complejidad: $\mathcal{O}(N)$
Descripción: Evalúa si la clave n pertenece al diccionario d. N es el tamano del diccionario
Aliasing: No hay aspectos de aliasing.

SIGNIFICADO(**in** d: `diccHash(nat,α)`, **in** n: `nat`) $\rightarrow$ res : α
Pre $\equiv$ {def?(n, d)}
Post $\equiv$ {alias($res =_{\text{obs}}$ obtener(n, d))}
Complejidad: $\mathcal{O}(1)$ en caso promedio
Descripción: Devuelve el significado de la clave n en el diccionario pasado como argumento de la función.
Aliasing: El significado lo devuelve por referencia. Los cambios que se hagan en res afectarán a los valores de la estructura.

Representación

`diccHash(nat, α)` **se representa con** `estr`

donde `estr` es `tupla`(*tamano*: `nat`
, *array*: `arregloDimensionable(tupla(`*clave* `: nat,` *significado* `: α))`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ $e.tamano =_{\text{obs}}$ tam($e.array$) $\wedge_{\text{L}}$ ($\forall$ j, k : nat) $j \neq k \wedge j <$ $e.tamano \wedge k <$ $e.tamano \wedge$ definido?($e.array, j$) $\wedge$ definido?($e.array$, k) $\Rightarrow_{\text{L}}$ $e.array[j] \neq e.array[k]$

Abs : estr e $\longrightarrow$ diccHash(nat, α) {Rep(e)}
Abs(e) $\equiv$ d : diccHash(nat, α) | ($\forall$ c : nat) (def?(c, d) $\Leftrightarrow$ ($\exists$ *indice* : nat) (definido?($e.array$, *indice*) $\wedge$ $0 \leq$ *indice* $<$$e.tamano \wedge_{\text{L}}$ $e.array$[*indice*].*clave* $= c$)) $\wedge_{\text{L}}$ obtener(c, d) $= e.array$[obtenerIndice(c mod $e.tamano$, c, e) mod $e.tamano$].*significado*

obtenerIndice : nat *indice* × nat *clave* × estr e $\longrightarrow$ nat {Rep(e)}
obtenerIndice(*indice*, *clave*, e) $\equiv$ **if** $e.array[indice].clave \neq clave$ **then**
1 + obtenerIndice((*indice* + 1) mod $e.tamano$, *clave*, e)
else
0
fi

Algoritmos

---

*i***Definir**(**in/out** e : `estr`, **in** *clave* : `nat`, **in** $significado : \alpha$)

1: *indice* : nat ← *clave* mod $e.tamano$ ▷ $\mathcal{O}(1)$
2: i : nat ← 0 ▷ $\mathcal{O}(1)$
3: **while** definido?($e.array$, *indice*) **do** ▷ $\mathcal{O}(1)$ (en caso promedio)
4: $i \leftarrow i + 1$ ▷ $\mathcal{O}(1)$
5: *indice* ← (*clave* + i) mod $e.tamano$ ▷ $\mathcal{O}(1)$
6: *valor* : tupla(nat, α) ← $< clave, significado >$ ▷ $\mathcal{O}(1)$
7: $e.array[indice] \leftarrow valor$ ▷ $\mathcal{O}(1)$
8: Complejidad: $\mathcal{O}(1)$
9: Justificación: En caso promedio, asumiendo que las claves están uniformemente distribuidas, la mayoría de las veces entra directamente a un bucket vacío, por lo que el algoritmo termina siendo $\mathcal{O}(1)$

---

*i***Definido?**(**in** e : `estr`, **in** *clave* : `nat`)

1: $i \leftarrow 0$ ▷ $\mathcal{O}(1)$
2: *indice* : nat ← *clave* mod $e.tamano$ ▷ $\mathcal{O}(1)$
3: **while** e.array[indice].clave ≠ clave ∧ $i <$ $e.tamano$ **do** ▷ $\mathcal{O}(N)$
4: $i \leftarrow i + 1$ ▷ $\mathcal{O}(1)$
5: *indice* ← (*clave* + i) mod $e.tamano$ ▷ $\mathcal{O}(1)$
6: $res \leftarrow (i \neq e.tamano) \vee (i = e.tamano \wedge e.array[indice].clave = clave)$ ▷ $\mathcal{O}(1)$
7: Complejidad: $\mathcal{O}(N)$
8: Justificación: Se recorre todo el arreglo buscando el elemento. En peor caso es O(N), siendo N el tamaño del diccionario

---

*i***Significado**(**in** e : `estr`, **in** *clave* : `nat`) → $res : \alpha$

1: *indice* : nat ← *clave* mod $e.tamano$ ▷ $\mathcal{O}(1)$
2: i : nat ← 0 ▷ $\mathcal{O}(1)$
3: **while** $e.array[indice].clave \neq clave$ **do** ▷ $\mathcal{O}(1)$ (en caso promedio)
4: $i \leftarrow i + 1$ ▷ $\mathcal{O}(1)$
5: *indice* ← (*clave* + i) mod $e.tamano$ ▷ $\mathcal{O}(1)$
6: $res \leftarrow e.array[indice].significado$ ▷ $\mathcal{O}(1)$
7: Complejidad: $\mathcal{O}(1)$
8: Justificación: En caso promedio, asumiendo que las claves están uniformemente distribuidas, la mayoría de las veces entra directamente al bucket del agente, por lo que el algoritmo termina siendo $\mathcal{O}(1)$ (en caso promedio)

Algorithm 53 Crear

```
1: procedure iCrear(in n : nat)→ res : estr
2:     res.tamano ← n                                   ▷ O(1)
3:     res.array ← crearArreglo(n)                      ▷ O(n)
```

Complejidad: $\mathcal{O}(n)$
Justificación: Crear un arreglo dimensionable de tamaño n cuesta $\mathcal{O}(n)$.