



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Grupo Número 1

06 de Septiembre de 2015

Algoritmos y Estructuras de Datos II

Integrante	LU	Correo electrónico
Joel Esteban Camera	257/14	joel.e.camera@gmail.com
Manuel Mena	313/14	manuelmena1993@gmail.com
Kevin Frachtenberg Goldsmith	247/14	kevinfra94@gmail.com
Nicolás Bukovits	546/14	nicobuk@gmail.com

Buen trabajo!
Felicidades

(I) corrigió la dñ
(ver atas)



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - Pabellón 1

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Argentina

Tel/Fax: (54 11) 4576-3359

<http://exactas.uba.ar>

Índice

1. Módulo CampusSeguro	8
1.1. Interfaz	8
1.1.1. Operaciones básicas de CampusSeguro	8
1.1.2. Representación de campusSeguro	9
1.1.3. Invariante de Representación	9
1.1.4. Función de Abstracción	10
1.2. Algoritmos	10
2. Módulo Campus	29
2.1. Interfaz	29
2.1.1. Operaciones básicas de Campus	29
2.1.2. Representación de campus	31
2.1.3. Invariante de Representación	31
2.1.4. Función de Abstracción	31
2.2. Algoritmos	31
3. Módulo Diccionario Nat Fijo	34
3.1. Interfaz	34
3.1.1. Operaciones básicas de DiccNat(α)	34
3.1.2. Operaciones básicas del iterador	35
3.1.3. Especificación de las operaciones auxiliares utilizadas en la interfaz	35
3.2. Representación de DiccNat(α)	37
3.2.1. Invariante de Representación	37
3.2.2. Función de Abstracción	37
3.3. Representación del iterador de DiccNat	38
3.4. Algoritmos	38
4. Módulo Diccionario String(α)	41
4.1. Interfaz	41
4.1.1. Operaciones básicas de Diccionario String(α)	41
4.1.2. Operaciones básicas del iterador	42
4.1.3. Representación de Diccionario String(α)	44
4.1.4. Invariante de Representación	44
4.1.5. Función de Abstracción	45
4.2. Algoritmos	45



- faltan ejemplos de los algoritmos auxiliares
- falta la especificación de los errores de comportamiento (el error en las representaciones no válidas) y cómo se les manejan
- errores de importancia menor
- errores de diseño

1. Módulo CampusSeguro

1.1. Interfaz

se explica con: CAMPUSSEGURO.

géneros: campusSeguro.

1.1.1. Operaciones básicas de CampusSeguro

COMENZAR RASTRILLAJE(in c : campus, in d : dicc(placa_AS)) \rightarrow res : campusSeguro

Pre $\equiv \{(\forall a : \text{agente}) (\text{def?}(a,d) \Rightarrow_L (\text{posVálida}(\text{obtener}(a,d)) \wedge \neg\text{ocupada?}(\text{obtener}(a,d,c))) \wedge (\forall a, a2 : \text{agente}) ((\text{def?}(a,d) \wedge \text{def?}(a2,d) \wedge a \neq a2) \Rightarrow_L \text{obtener}(a,d) \neq \text{obtener}(a2,d))\}$

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

Complejidad: $O((f * c)^2 + N_a)$

Descripción: Crea un nuevo campusSeguro tomando un campus y un diccionario con agentes.

INGRESAR ESTUDIANTE(in e : nombre, in p : posición, in/out cs : campusSeguro)

Pre $\equiv \{cs =_{obs} cs_o \wedge e \notin (\text{estudiantes}(cs) \cup \text{hippies}(cs)) \wedge \text{esIngreso?}(p, \text{campus}(cs)) \wedge \neg\text{estaOcupada?}(p, cs)\}$

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

Complejidad: $O(|n_m|) + O(\log(N_a))$

Descripción: Ingrasa un nuevo estudiante al campusSeguro.

INGRESAR HIPPIE(in h : nombre, in p : posición, in/out cs : campusSeguro)

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

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

Complejidad: $O(|n_m|)$

Descripción: Ingrasa un nuevo hippie el campusSeguro

MOVER ESTUDIANTE(in e : nombre, in dir : dirección, in/out cs : campusSeguro)

Pre $\equiv \{cs =_{obs} cs_o \wedge e \in \text{estudiantes}(cs) \wedge (\text{seRetira}(e, dir, cs) \vee$

campus(cs))

$(\text{posValida}(\text{proxPosición}(\text{posEstudianteYHippie}(e, cs), dir, \text{campus}(cs)),$

$\neg\text{estaOcupada?}(\text{proxPosición}(\text{posEstudianteYHippie}(e, cs), dir, \text{campus}(cs)), cs))\}$

Post $\equiv \{cs =_{obs} \text{moverEstudiante}(e, dir, cs_o)\}$

Complejidad: $O(|n_m|)$

Descripción: Mueve un estudiante dentro del campus o lo hace salir y se actualizan los atrapados, sanciones (si es que las hay) y hippies atrapados (si es que los hay).

MOVER HIPPIE(in h : nombre, in/out cs : campusSeguro)

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

Post $\equiv \{cs =_{obs} \text{moverHippie}(h, d, cs_o)\}$

Complejidad: $O(|n_m|) + O(N_e)$

Descripción: Mueve un hippie dentro del campus y se actualizan los atrapados, sanciones (si es que las hay) y hippies atrapados (si es que los hay).

MOVER AGENTE(in a : agente, in/out cs : campusSeguro)

Pre $\equiv \{cs =_{obs} cs_o \wedge a \in \text{agentes}(cs) \wedge \text{cantSanciones}(a, cs) \leq 3 \wedge$

$\neg\text{todasOcupadas?}(\text{vecinos}(\text{posEstudianteYHippie}(h, cs), \text{campus}(cs)), cs)\}$

Post $\equiv \{cs =_{obs} \text{moverAgente}(a, cs_o)\}$

Complejidad: $O(|n_m|) + O(\log(N_a)) + O(N_h)$

Descripción: Mueve un agente dentro del campus y se actualizan los atrapados, sanciones (si es que las hay) y hippies atrapados (si es que los hay).

CAMPUS(in $cs : \text{campusSeguro}$) $\rightarrow res : \text{campus}$

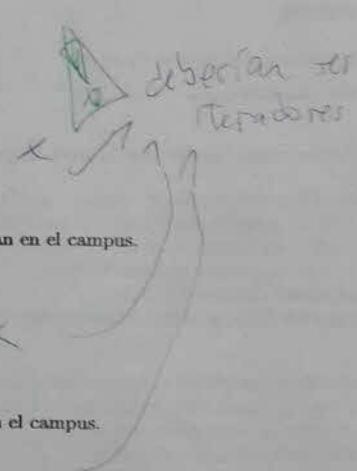
Pre $\equiv \{\text{true}\}$

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

Complejidad: $O(1)$

Descripción: Devuelve el campus del campusSeguro.

Aliasing: res es una referencia no modificable.



ESTUDIANTES(in $cs : \text{campusSeguro}$) $\rightarrow res : \text{conj}(\text{nombre})$

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

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

Complejidad: $O(1)$

Descripción: Devuelve el conjunto de los estudiantes que están en el campus.

Aliasing: res es una referencia no modificable.

HIPPIES(in $cs : \text{campusSeguro}$) $\rightarrow res : \text{conj}(\text{nombre})$

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

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

Complejidad: $O(1)$

Descripción: Devuelve el conjunto de los hippies que están en el campus.

Aliasing: res es una referencia no modificable.

AGENTES(in $cs : \text{campusSeguro}$) $\rightarrow res : \text{conj}(\text{agentes})$

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

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

Complejidad: $O(1)$

Descripción: Devuelve el conjunto de los agentes que están en el campus.

Aliasing: res es una referencia no modificable.

POSESTUDIANTEYHIPPIE(in $id : \text{nombre}$, in $cs : \text{campusSeguro}$) $\rightarrow res : \text{posición}$

Pre $\equiv \{id \in (\text{estudiantes}(cs) \cup \text{hippies}(cs))\}$

Post $\equiv \{res =_{obs} \text{posEstudianteYHippie}(id, cs)\}$

Complejidad: $O(|n_m|)$, donde $|m_n|$ es la longitud más larga entre todos los nombres.

Descripción: Devuelve la posición del estudiante o hippie.

POSAGENTE(in $a : \text{agente}$, in $cs : \text{campusSeguro}$) $\rightarrow res : \text{posición}$

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

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

Complejidad: $O(1)$ en caso promedio.

Descripción: Devuelve la posición del agente pasado como parámetro.

CANTSANCIONES(in $a : \text{agente}$, in $cs : \text{campusSeguro}$) $\rightarrow res : \text{nat}$

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

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

Complejidad: $O(1)$ en caso promedio.

Descripción: Devuelve la cantidad de sanciones que posee el agente pasado como parámetro.

CANTHIPPIESATRAPADOS(in $a : \text{agente}$, in $cs : \text{campusSeguro}$) $\rightarrow res : \text{nat}$

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

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

Complejidad: $O(1)$ en caso promedio.

Descripción: Devuelve la cantidad de hippies que atrapó el agente pasado como parámetro.

MÁS VIGILANTE(in cs: campusSeguro) \rightarrow res : agente

Pre \equiv {true}

Post \equiv {res =_{obs} másVigilante(cs)}

Complejidad: $O(1)$

Descripción: Devuelve la placa del agente que ha atrapado mas hippies.

CONMISMASSANCIJES(in a: agente, in cs: campusSeguro) \rightarrow res : conj(agentes)

Pre \equiv {a \in agentes(cs)}

Post \equiv {res =_{obs} conMismasSanciones(a,cs)}

Complejidad: $O(1)$ ~~PROBLEMA~~

Descripción: Devuelve el conjunto de los agentes que tienen el mismo numero de sanciones que el agente pasado como parametro.

Aliasing: res es una referencia no modificable.

CONKSANCIJES(in k: nat, in cs: campusSeguro) \rightarrow res : conj(agentes)

Pre \equiv {true}

Post \equiv {res =_{obs} conKSanciones(k,cs)}

Complejidad: $O(N_a)$ la primera vez que se la llama y $O(\log(N_a))$ en futuras llamadas mientras no ocurran sanciones.

Descripción: Devuelve el conjunto de agenes que tienen k sanciones.

Aliasing: res es una referencia no modificable.

1.1.2. Representación de campusSeguro

campusSeguro se representa con estr

dónde estr es tupla(*campus*: campus,
personalAS: diccNat(agente, datosAgente),
posicionesAgente: vector(As),
masVigilante: As,
listaMismasSanc: lista(kSanc),
arregloMismasSanc: arreglo(puntero(kSanc)),
mismasSancModificado: bool,
hippies: dicString(nombre, posición),
estudiantes: dicString(nombre, posición),
posicionesHippies: vector(nombre),
posicionesEstudiantes: vector(nombre))

dónde datosAgente es tupla(*posición*: posición,
cantSanc: nat,
cantAtrapados: nat,
itMismasSanc: itLista(kSanc),
itConjMismasSanc: itConj(agente))

dónde As es tupla(*agente*: nat, *datos*: puntero(datosAgente))

dónde kSanc es tupla(*sanc*: nat, *agentes*: conj(agente))

1.1.3. Invariante de Representación

- (I) Las posiciones de todos los agentes son posiciones válidas del campus.
- (II) Las posiciones de los hippies son posiciones válidas del campus.
- (III) Las posiciones de los estudiantes son posiciones válidas del campus.
- (IV) Las posiciones de los agentes son distintas a las posiciones de los hippies.
- (V) Las posiciones de los estudiantes son distintas a las posiciones de los hippies.
- (VI) Las posiciones de los agentes son distintas a las posiciones de los estudiantes.
- (VII) El agente masVigilante está definido en personalAS y tiene la mayor cantidad de sanciones.
- (VIII) Para todo nombre que está definido en el diccionario de hippies no puede estar definido en el diccionario de estudiantes y viceversa.
- (IX) La cantidad de elementos que tiene posicionesAgente, posicionesHippies y posicionesEstudiantes es la cantidad de coordenadas que tiene la grilla.
- (X) Los agentes de posicionesAgente son los mismos que los de personalAS y viceversa.
- (XI) La posición de todos los agentes de posicionesAgente se mapea con la posición que tienen en personalAS y viceversa.
- (XII) Las posiciones en posicionesAgente que no tienen agente tienen un puntero a NULL.
- (XIII) La dimensión del campus es mayor que la cantidad de estudiantes, hippies, agentes y obstáculos.
- (XIV) La longitud de listaMismasSanc es la cantidad de sanciones diferentes que hay.
- (XV) Para cada cantidad de sanciones distinta hay un nodo en listaMismasSanc con esa cantidad de sanciones.

- (XVI) listaMismasSanc esta ordenada por cantidad de sanciones.
- (XVII) En cada nodo de listaMismasSanc va a estar el conjunto de todos los agentes que tienen la cantidad de sanciones indicada en el nodo.
- (XVIII) Si mismasSancModificado es falso, vectorMismasSanc tiene misma cantidad de elementos que listaMismasSanc, y cada elemento apunta a un nodo de listaMismasSanc, respetando el orden de listaMismasSanc
- (XIX) La posiciones de todos los hippies en posicionesHippies se mapea con la posicion que tienen en el diccString hippies y viceversa
- (XX) La posiciones de todos los estudiantes en posicionesEstudiantes se mapea con la posicion que tienen en el diccString estudiantes y viceversa

Rep : estr \rightarrow bool

Rep(e) = true \iff

$$\begin{aligned} & (\forall a: \text{nat})(\text{def?}(a, e.\text{personalAS}) \Rightarrow_L \text{posValida}(\text{Obtener}(a.e.\text{personalAS}).\text{posicion}, e.\text{campus}) \wedge \\ & \neg\text{ocupada}(\text{Obtener}(a.e.\text{personalAS}).\text{posicion}), e.\text{campus})) \\ & \wedge \\ & (\forall h: \text{string})(\text{def?}(h, e.\text{hippies}) \Rightarrow_L \text{posValida}(\text{Obtener}(h.e.\text{hippies}), e.\text{campus}) \wedge \neg\text{ocupada}(\text{Obtener}(h.e.\text{hippies})), e.\text{campus})) \\ & \wedge \\ & (\forall est: \text{string})(\text{def?}(est, e.\text{estudiantes}) \Rightarrow_L \text{posValida}(\text{Obtener}(est, e.\text{estudiantes}), e.\text{campus}) \wedge \\ & \neg\text{ocupada}(\text{Obtener}(est, e.\text{estudiantes})), e.\text{campus})) \\ & \wedge \\ & (\forall a: \text{nat})(\forall h: \text{string}) \text{ def?}(a, e.\text{personalAS}) \wedge \text{def?}(h, e.\text{hippies}) \Rightarrow_L \text{Obtener}(a, e.\text{personalAS}).\text{posicion} \neq \\ & \text{Obtener}(h, e.\text{hippies})) \\ & \wedge \\ & (\forall est: \text{string})(\forall h: \text{string}) \text{ def?}(est, e.\text{estudiantes}) \wedge \text{def?}(h, e.\text{hippies}) \Rightarrow_L \text{Obtener}(est, e.\text{estudiantes}) \neq \\ & \text{Obtener}(h, e.\text{hippies})) \\ & \wedge \\ & (\forall a: \text{nat})(\forall est: \text{string}) \text{ def?}(a, e.\text{personalAS}) \wedge \text{def?}(est, e.\text{estudiantes}) \Rightarrow_L \text{Obtener}(a, e.\text{personalAS}).\text{posicion} \neq \\ & \text{Obtener}(est, e.\text{estudiantes})) \\ & \wedge \\ & \text{def?}(\text{masVigilante.agente}, e.\text{personalAS}) \wedge_L \text{Obtener}(\text{masVigilante.agente}, e.\text{personalAS}) = * \text{masVigilante.datos.cantSanc} \\ & \wedge \\ & (\forall a: \text{nat}) \text{ def?}(a, e.\text{personalAS}) \Rightarrow_L \text{Obtener}(a, e.\text{personalAS}).\text{cantSanc} \leq * \text{masVigilante.datos.cantSanc} \\ & \wedge \\ & (\forall h: \text{string}) \text{ def?}(h, e.\text{hippies}) \Rightarrow \neg\text{def?}(h, e.\text{estudiantes})) \\ & \wedge \\ & (\forall est: \text{string}) \text{ def?}(est, e.\text{estudiantes}) \Rightarrow \neg\text{def?}(est, e.\text{hippies})) \\ & \wedge \\ & \text{longitud}(e.\text{posicionesAgente}) = \text{filas}(e.\text{campus}) * \text{columnas}(e.\text{campus})) \\ & \wedge \\ & \text{longitud}(e.\text{posicionesHippies}) = \text{filas}(e.\text{campus}) * \text{columnas}(e.\text{campus})) \\ & \wedge \\ & \text{longitud}(e.\text{posicionesEstudiantes}) = \text{filas}(e.\text{campus}) * \text{columnas}(e.\text{campus})) \\ & \wedge \\ & (\forall a: \text{nat}) \text{ def?}(a, e.\text{personalAS}) \Leftrightarrow_L (\exists i: \text{nat}) i < \text{longitud}(e.\text{posicionesAgente}) \wedge_L \\ & e.\text{posicionesAgente}[i].\text{agente} = a \wedge *e.\text{posicionesAgente}[i].\text{datos} = \text{Obtener}(a, e.\text{personalAS}) \wedge (i / \\ & \text{columnas}(e.\text{campus})) = \text{Obtener}(a, e.\text{personalAS}).\text{posicion.y} \wedge (i \bmod \text{columnas}(e.\text{campus})) = \text{Obtener}(a, e.\text{personalAS}).\text{posicion.x}) \\ & \wedge \\ & (\forall i: \text{nat}) i < \text{longitud}(e.\text{posicionesAgente}) \Leftrightarrow_L *e.\text{posicionesAgente}[i].\text{datos} = \text{NULL} \vee (\exists a: \text{nat}) \text{ def?}(a, e.\text{personalAS}) \wedge_L \\ & *e.\text{posicionesAgente}[i].\text{datos} = \text{Obtener}(a, e.\text{personalAS})) \\ & \wedge \\ & \#claves(e.\text{personalAS}) + \#claves(e.\text{hippies}) + \#claves(e.\text{estudiantes}) + \#\text{Obstaculos}(e.\text{campus}) \leq \\ & \text{filas}(e.\text{campus}) * \text{columnas}(e.\text{campus}) \end{aligned}$$

```

^
long(e.listaMismasSanc) = #SancionesDistintas(e.personalAS)
^
(∀ sanc : nat)(sanc ∈ conjSanciones(claves(e.personalAS), e.personalAS) ⇒ (∃ nodo : kSanc)(esta?(kSanc, e.listaMismasSanc) ∧ kSanc.sanc = sanc))
^
ordenada?(e.listaMismasSanc)
^
(∀ nodo : kSanc)(esta?(kSanc, e.listaMismasSanc) ⇒ agentes(kSanc) =obs agentesKSanc(e.personalAS, claves(e.personalAS), sanc(kSanc)))
^
e.mismasSancModificado ⇒ (long(e.listaMismasSanc) = long(vectorMismasSanc)) ∧L
mismosNodos(e.listaMismasSanc, e.vectorMismasSanc)
^
(∀ h : string)(def?(h, e.hippies) ⇔L (∃ i : nat)(i < longitud(e.posicionesHippies) ∧L e.posicionesHippies[i] = obtener(h, e.hippies) ∧ (i / columnas(e.campus) = obtener(j, e.hippies).y) ∧ (i mod columnas(e.campus) = obtener(j, e.hippies).y)))
^
(∀ h : string)(def?(h, e.estudiantes) ⇔L (∃ i : nat)(i < longitud(e.posicionesEstudiantes) ∧L e.posicionesEstudiantes[i] = obtener(h, e.estudiantes) ∧ (i / columnas(e.campus) = obtener(j, e.estudiantes).y) ∧ (i mod columnas(e.campus) = obtener(j, e.estudiantes).y)))
^

#Obstaculos : campus → nat
#ObstaculosAux : nat × nat × campus → nat
#SancionesDistintas : diccNat(agente × datosAgente) → nat
conjSanciones : conj(nat) × diccNat(agente × datosAgente) → conj(nat)
cantAgentes : secu(kSanc) → nat
ordenada? : secu(kSanc) → bool
agentesKSanc : diccNat(agente × datosAgente) × conj(agente) × nat → conj(agente)
mismosNodos : secu(kSanc) lista × secu(puntero(kSanc)) vec → bool {long(lista) = long(vec)}
{long(lista) = long(vec)}

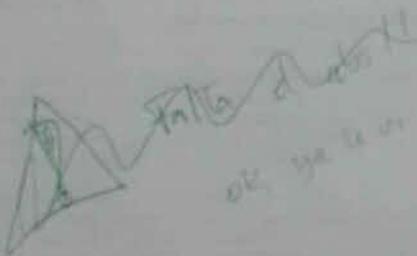
#Obstaculos(c) ≡ #ObstaculosAux(filas(c)-1, columnas(c)-1, c)
#ObstaculosAux(f,col,c) ≡ if (f = 0 ∧ col = 0) then
    β(ocupada?(<f,col>, c)
else
    if (f ≠ 0 ∧ col = 0) then
        #ObstaculosAux(f - 1, columnas(c)-1) + β(ocupada?(<f,col>, c)
    else
        if (f = 0 ∧ col ≠ 0) then
            #ObstaculosAux(filas(c), col - 1, c) + β(ocupada?(<f,col>, c)
        else
            #ObstaculosAux(f - 1, col - 1, c) + β(ocupada?(<f,col>, c)
        fi
    fi
#SancionesDistintas(d) ≡ #conjSanciones(Claves(d),d)
conjSanciones(c,d) ≡ if ~∅(c) then
    ∅
else
    Ag(Obtener(DameUno(c).d).cantSanc, conjSanciones(SinUno(c,d)))
fi
cantAgentes(s) ≡ if s = <> then 0 else Long(prim(s).vectorAgente) + cantAgentes(fin(s)) fi

```

```

ordenada?(lista) = IF vacia?(lista) Then
    true
  else
    IF vacia?([n](lista)) Then
      true
    else
      IF sonprimos(prim(lista)) <= sonprimos(prim([n](lista))) Then ordenada?([n](lista)) else false
    fi
  fi
agenteKSane(dicc, claves, k) = IF P?(claves) Then
    j
  else
    IF existSane(obtener(prim(claves), dicc)) = k Then
      Ag(prm([claves]), agenteKSane(dicc, fin([claves]), k))
    else
      agenteKSane(dicc, fin([claves]), k)
    fi
  fi
fin
numerosNodos(lista, vec) = IF vacia?(lista) Then
    true
  else
    IF lprim(lista) = prim(vec) Then numerosNodos(lm(lista), fin(vec)) else false
  fi

```



1.1.4. Función de Abstracción

$\text{Abs} : \text{estr } e \rightarrow \text{campusSeguro}$

$$\text{Abs}(e) =_{\text{obs}} \text{campusSeguro} \mid \begin{array}{l} e.\text{campus} = \text{campus}(cs) \\ \wedge \\ e.\text{estudiantes} = \text{estudiantes}(cs) \\ \wedge \\ e.\text{hippies} = \text{hippies}(cs) \\ \wedge \\ e.\text{personalAS} = \text{agentes}(cs) \\ \wedge \\ (\forall a: \text{nat}) \ (def?(a, e.\text{personalAS}) \Rightarrow_L \ posAgente(a, cs) = \text{Obtener}(a, e.\text{personalAS}).\text{posicion} \wedge \ cantSanciones(a, cs) = \text{Obtener}(a, e.\text{personalAS}).cantSanc \wedge \ cantHippiesAtrapados(a, cs) = \text{Obtener}(a, e.\text{personalAS}).cantAtrapados) \\ \wedge \\ (\forall id: \text{string}) \ (def?(id, e.\text{estudiantes}) \wedge_L \ Obtener(id, e.\text{estudiantes}) = posEstudianteYHippie(cs)) \vee (def?(id, e.\text{hippies}) \wedge_L \ Obtener(id, e.\text{hippies}) = posEstudianteYHippie(cs)) \end{array}$$

1.2. Algoritmos

icomenzarRastrillaje (in $c: \text{campus}$, in $d: \text{diccNat}(\text{agente}, \text{datosAgente})$) \rightarrow res: estr

res.campus $\leftarrow c$	O(1)
res.listaMismasSanc \leftarrow generarListaMismasSanc(d)	O(N_a)
res.personalAS $\leftarrow d$	O(1)
res.posicionesAgente \leftarrow vectorizarPos(d, filas(c), columnas(c))	$O((f * c)^2 + N_a)$
res.masVigilante \leftarrow menorPlaca(d)	O(N_a)
res.mismasSancModificado \leftarrow true	O(1)
res.hippies \leftarrow Vacio()	O(1)
res.estudiantes \leftarrow Vacio()	O(1)
res.posicionesHippies \leftarrow Vacio()	O(1)
res.posicionesEstudiantes \leftarrow Vacio	O(1)
nat: i $\leftarrow 0$	O(1)
while i < filas(c)*columnas(c) do	O(1)
AgregarAtras(res.posicionesHippies, " ")	O(1)
AgregarAtras(res.posicionesEstudiantes, " ")	O(1)
i \leftarrow i+1	O(1)
end while	$O(f * c)$

Complejidad: $O(N_a) + O(N_a) + O((f * c)^2 + N_a) + O((f * c)^2) = O((f * c)^2 + N_a)$

vectorizarPos (in $d: \text{diccNat}(\text{agente}, \text{datosAgente})$, in $f: \text{nat}$, in $c: \text{nat}$) \rightarrow res: vector(AS)

res \leftarrow Vacio()	O(1)
nat: i $\leftarrow 0$	O(1)
itDiccNat(agente, datosAgente): it \leftarrow CrearIt(d)	O(1)
while i < f*c do	O(1)
AgregarAtras(res, tupla(0, NULL))	$O(f * c)$
i \leftarrow i+1	O(1)
end while	$O((f * c)^2)$
while HaySiguiente(it) do	O(1)

```

res [ Siguiente(it).significado.posicion.y * c +
      Siguiente(it).significado.posicion.x ] ←
      ← tupla(Siguiente(it).clave, puntero(Siguiente(it).significado)) O(1)
end while O(Na)
Complejidad : O((f * c)2 + Na)

```

Pre 1.1.1?

```

menorPlaca (in d: diccNat(agente, datosAgente)) → res: AS
  itDiccNat(agente, datosAgente): it ← CrearIt(d) O(1)
  nat: placaMenor ← Siguiente(it).clave O(1)
  puntero(datosAgente): punt ← puntero(Siguiente(it).datosAgente) O(1)
  while HaySiguiente(d) do O(1)
    if Siguiente(it).clave < placaMenor then O(1)
      placaMenor ← Siguiente(it).clave O(1)
      punt ← puntero(Siguiente(it).datosAgente) O(1)
    end if O(1)
    Avanzar(it) O(1)
  end while O(1)
  res.agente ← placaMenor O(1)
  res.datos ← punt O(1)

```

Complejidad : O(N_a)

Pre 1.1.2?

```

generarListaMismasSanc (in/out d: diccNat(agente, datosAgente)) → res: lista(kSanc)
  itDiccNat(agente, datosAgente): itDic ← CrearIt(d) O(1)
  res ← Vacia() O(1)
  AgregarAdelante(res, tupla(0, Vacio())) O(1 (esta vacío el vector))
  itLista(kSanc): itL ← CrearIt(res) O(1)

  while HaySiguiente(itDic) do O(1)
    itConj(agente): itC ← AgregarRapido(res.agentes, Siguiente(itDic).clave) O(1)
    Siguiente(itDic).significado.itConjMismasSanc ← itC O(1)
    Siguiente(itDic).significado.itMismasSanc ← itL O(1)
    Avanzar(itDic) O(1)
  end while O(Na)

```

Complejidad : O(N_a)

Pre 1.2.1?

```

IngresarEstudiante (in e: nombre, in pos: posicion, in/out cs, estr)
  if todasOcupadas?(vecinos(pos, cs.campus), cs) AND O(1)
    AND AIMenosUnAgente(vecinos(pos, cs.campus)) then O(1)
    conj(As): conjAgParaSanc ← AgParaPremSanc(vecinos(pos, cs.campus), cs) O(1)
    SancionarAgentes(conjAgParaSanc, cs) O(1)
  end if O(1)

  if CANTHippiesVecinos(vecinos(pos, cs.campus), cs) < 2 then O(|nm|)
    Definir(cs.estudiantes, e, pos) O(|nm|)
    cs.posicionesEstudiantes[pos.y * Columnas(cs.campus) + pos.x] ← e O(1)
  else
    Definir(cs.hippies, e, pos) O(|nm|)
    cs.posicionesHippies[pos.y * Columnas(cs.campus) + pos.x] ← e O(1)
  end if O(|nm| + |nm|)

```

```

conj(nombre, posicion): conjHippiesRodEst ←
    ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjHippiesRodEst) > 0 then O(1)
    itConj(nombre, posicion): itHEst ← CrearIt(conjHippiesRodEst) O(1)

    while HaySiguiente(itHEst) do O(1)
        Definir(cs.estudiantes, Siguiente(itHEst).nombre, Siguiente(itHEst).posicion) O(|nm|)
        Eliminar(cs.hippies, Siguiente(itHEst).nombre) O(|nm|)

        cs.posicionesEstudiantes[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
        + Siguiente(itHEst).posicion.x] ← " "
        ← cs.posicionesHippies[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
        + Siguiente(itHEst).posicion.x] ← " " O(1)

        cs.posicionesHippies[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
        + Siguiente(itHEst).posicion.x] ← " " O(1)

        Avanzar(itHEst) O(1)
    end while O(2|nm|)
end if O(4 * 2|nm|)

conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjHippiesRodAs) > 0 then O(1)
    itConj(nombre, posicion): itHAs ← CrearIt(conjHippiesRodAs) O(1)

    while HaySiguiente(itHAs) do O(1)
        Eliminar(cs.hippies, Siguiente(itHAs).nombre) O(|nm|)

        cs.posicionesHippies[Siguiente(itHAs).y * Columnas(cs.campus) +
        + Siguiente(itHAs).x] ← " "
        Avanzar(itHAs) Faltan actualizar las capturas y el mas visible O(1)
    end while O(4|nm|)
end if O(4|nm|)

conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjEstRodHip) > 0 then O(1)
    itConj(nombre, posicion): itEstH ← CrearIt(conjEstRodHip) O(1)

    while HaySiguiente(itEstH) do O(1)
        Eliminar(cs.estudiantes, Siguiente(itEstH).nombre) O(|nm|)

        cs.posicionesEstudiantes[Siguiente(itEstH).y * Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← " " O(1)

        Definir(cs.hippies, Siguiente(itEstH).nombre, Siguiente(itEstH).posicion) O(|nm|)

        cs.posicionesHippies[Siguiente(itEstH).y * Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← Siguiente(itEstH).nombre O(1)

        Avanzar(itEstH) O(1)
    end while O(4 * 2|nm|)
end if O(4 * 2|nm|)

conj(posicion): conjEstRodAs ←

```

```

    ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) O(1)
if Cardinal(conjEstRodAs) > 0 then O(1)
    itConj(posicion): itEstAs ← CrearIt(conjEstRodAs) O(1)

    while HaySiguiente(itEstAs) do O(1)
        if todasOcupadas?(vecinos(Siguiente(itEstAs), cs.campus), cs AND O(1)
            AND AlMenosUnAgente(vecinos(Siguiente(itEstAs), cs.campus)) then O(1)
                conj(as): conjAgParaSanc ←
                    ← AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus), cs) O(1)
                    SancionarAgentes(conjAgParaSanc, cs) O(1)
            end if O(1)
        end while O(4)

    end if O(4)

```

Complejidad: $O(2|n_m|) + O(4 * 2|n_m|) + O(4|n_m|) + O(4 * 2|n_m|) = O(22|n_m|) = O(|n_m|)$

EstudiantesRodeadosAs (in c: conj(posicion), in cs: estr) → res: conj(posicion)

```

itConj(posicion): itC ← CrearIt(c) O(1)
res ← Vacio() O(1)

while HaySiguiente(itC) do O(1)
    if TodasOcupadas?(vecinos(Siguiente(itC), cs), cs) AND O(1)
        AND AlMenosUnAgente(vecinos(Siguiente(itC), cs), cs) then O(1)
            AgregarRapido(res, Siguiente(itC)) O(1)
        end if O(1)
        Avanzar(itC) O(1)
    end while O(4)

```

Complejidad: $O(4) = O(1)$

*NO! A lo mejor la función de
Avanzar no es constante
(aunque es constante) → O(HC)*

EstudiantesRodeadosHippies (in c: conj(posicion), in cs: estr) → res: conj[nombre, posicion]

```

itConj(posicion): itC ← CrearIt(c) O(1)
res ← Vacio() O(1)

while HaySiguiente(itC) do O(1)
    if cs.posicionesEstudiantes[Siguiente(itC).y * Columnas(cs.campus) +
        Siguiente(itC).x] ≠ " " AND O(1)
        AND TodasOcupadas?(vecinos(Siguiente(itC), cs.campus), cs) AND O(1)
        AND HippiesAtrapando(vecinos(Siguiente(itC), cs.campus), cs) then O(1)
            AgregarRapido(res,
                tupla(cs.posicionesEstudiantes[Siguiente(itC).y * Columnas(cs.campus) +
                    Siguiente(itC).x], Siguiente(itC))) O(1)
    end while O(4)

```

Complejidad: ~~O(HC)~~ O(HC)

HippiesAtrapando (in c: conj(posicion), in cs: estr) → res: bool

```

nat: i ← 0 O(1)
itConj(posicion): itC ← CrearIt(c) O(1)

while HaySiguiente(itC) do O(1)

```

```

if cs.posicionesHippies[ Siguiente(itC).y * Columnas(cs.campus) +
+ Siguiente(itC).x ] ≠ " " then
    i ← i+1
end if
Avanzar(itC)
end while

res ← i ≥ 2

```

$O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(4)$
 $O(1)$

Complejidad : $O(1)$

```

SancionarAgentes (in c: conj(as), in/out cs: estr)
itConj(as): itC ← CrearIt(c)

while HaySiguiente(itC) do
    * Siguiente(itC).datos.cantSanc ←
        ← * Siguiente(itC).datos.cantSanc + 1
    itLista(kSanc): itLis ← * Siguiente(itC).datos.itMismasSanc

    if HaySiguiente(* Siguiente(itC).datos.itMismasSanc) then
        Avanzar(* Siguiente(itC).datos.itMismasSanc)
    if Siguiente(* Siguiente(itC).datos.itMismasSanc).sanc =
        * Siguiente(itC).datos.cantSanc then
            [ ]
        else
            AgregarComoAnterior(* Siguiente(itC).datos.itMismasSanc,
                tupla(* Siguiente(itC).datos.cantSanc, Vacio()))
            Retroceder(* Siguiente(itC).datos.itMismasSanc)
        end if
    else
        AgregarComoSiguiente(* Siguiente(itC).datos.itMismasSanc,
            tupla(* Siguiente(itC).datos.cantSanc, Vacio()))
        Avanzar(* Siguiente(itC).datos.itMismasSanc)
    end if

    EliminarSiguiente(* Siguiente(itC).datos.itConjMismasSanc)
    * Siguiente(itC).datos.itConjMismasSanc ←
        ← AgregarAdelante(* Siguiente(itC).datos.itMismasSanc, agentes
            * Siguiente(itC).agente)

```

$O(1)$
 $O(1)$

Complejidad : $O(1)$

```

HippiesRodeadosAs (in c: conj(posicion), in cs: estr) → res: conj(nombre, posicion)
itConj(posicion): itC ← CrearIt(c)
res ← Vacio()

while HaySiguiente(itC) do
    if cs.posicionesHippies[ Siguiente(itC).y * Columnas(cs.campus) +
        + Siguiente(itC).x ] ≠ " " AND
        todasOcupadas?(vecinos(Siguiente(itDiccS), cs.campus)) AND
        AlMenosUnAgente(vecinos(Siguiente(itDiccS), cs.campus)) then
            AgregarRapido(res,
                tupla(cs.posicionesHippies[ Siguiente(itC).y * Columnas(cs.campus) +
                    + Siguiente(itC).x ],

```

$O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$

```

Siguiente(itC)) O(1)
conj(As): conjAgPremiar ←
← AgParaPremSanc(vecinos(Siguiente(itDiceS), cs), cs) O(1)
PremiarAgentes(conjAgPremiar, cs) O(1)

end if
Avanzar(itC) O(1)
end while O(1)
end O(4)

```

No. Esto es a fuerza de función

Complejidad : $O(4) = O(1)$. La cantidad maxima del conjunto de posiciones es 4. $O(Hc)$

```

AgParaPremSanc (in c: conj(posicion), in cs: estr) → res: conj(As)
itConj(posicion): itC ← CrearIt(c) O(1)
res ← Vacio() O(1)

while HaySiguiente(itC) do O(1)
    if cs.posicionesAgente[Siguiente(itC).y * Columnas(cs.campus) + x].datos ≠
        # NULL then O(1)
        AgregarRapido(res,
            cs.posicionesAgente[Siguiente(itC).y * Columnas(cs.campus) + x]) O(1)
    end if O(1)
    Avanzar(itC) \ote{1} end while \ote{4} ofi {O(1) + O(1) + O(4) = O(1)}

```

No. $O(\#(c)) \rightarrow$ este algoritmo

```

PremiarAgentes (in c: conj(As), in/out cs: estr)
itConj(As): itC ← CrearIt(As) O(1)

while HaySiguiente(itC) do O(1)
    *Siguiente(itC).datos.cantAtrapados ←
    ← *Siguiente(itC).datos.cantAtrapados+1 O(1)
end while O(1)

```

Complejidad : $O(4) = O(1)$. La cantidad maxima de agentes va a ser 4.

```

CantHippiesVecinos (in c: conj(posicion), in cs: estr) → res: nat
itConj(posicion): itC ← CrearIt(c) O(1)
res ← 0

while HaySiguiente(itC) do O(1)
    itDicString(nombre, posicion): itDic ← CrearIt(cs.hippies) O(1)

    while HaySiguiente(itDic) do O(1)
        if Siguiente(itDic) = Siguiente(itC) O(1)
            res ← res+1 O(1)
        end if O(1)
    end while Avanzar O(|nm|)

    Avanzar(itC) O(1)
end while O(|nm|)

```

Complejidad : $O(|n_m|)$. Como vecinos como maximo tiene longitud 4, la complejidad seria $O(4 * |n_m|) = O(|n_m|^2)$

HippiesEscodeadosEstudiantes (in c: conj(posicion), in cs: estr) → res: conj(nombre, posicion)

itConj(posicion): itC ← CrearIt(c)
res ← Vacio()

O(1)
O(1)

```

while HaySiguiente(itC) do
    if cs.posicionesHippies[Siguiente(itC).y * Columnas(cs.campus) +
    + Siguiente(itC).x] ≠ " " AND
        todasOcupadas?(vecinos(Siguiente(itC), cs.campus)) AND
        TodosEstudiantes(vecinos(Siguiente(itC), cs.campus)) then
            AgregarRapido(res,
                tupla(., cs.posicionesHippies[Siguiente(itC).y * Columnas(cs.campus) +
                + Siguiente(itC).x]
                    Siguiente(itC)))
        end if
        Avanzar(itC)
    end while

```

O(1)
O(1)

Complejidad: $O(4) = O(1)$. Como el conjunto c tiene como maximo longitud 4, la complejidad sería $O(1)$

todasOcupadas? (in c: conj(posicion), in cs: estr) → res: bool

res ← false
itConj(posicion): itC ← CrearIt(c)

O(1)
O(1)

```

while HaySiguiente(itC) do
    if cs.posicionesHippies[Siguiente(itC).y * Columnas(cs.campus) +
    + Siguiente(itC).x] ≠ " " then
        res ← true
    if cs.posicionesEstudiantes[Siguiente(itC).y * Columnas(cs.campus) +
    + Siguiente(itC).x] ≠ " " then
        res ← true
    Avanzar(itC)
end while

```

O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)

```

itC ← CrearIt(c)
while HaySiguiente(itC) do
    if cs.posicionesAgente[Siguiente(itC).y * Columnas(cs.campus) + x].datos ≠
    # NULL then
        res ← true
    end if
end while

```

O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)

```

itC ← CrearIt(c)
while HaySiguiente(itC) do
    if Ocupada?(cs.campus, Siguiente(itC))
        res ← true
    end if
    Avanzar(itC)
end while

```

O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)

Complejidad: $O(1)$. Como el conjunto c tiene como maximo longitud 4, la complejidad sería $O(1)$

AlMenosUnAgente (in c: conj(posicion), in cs: estr) → res: bool

```

itConj(posicion): itC ← CrearIt(c) O(1)
res ← false O(1)

while HaySiguiente(itC) do O(1)
    if cs.posicionesAgente[Siguiente(itC).y * Columnas(cs.campus) + x].datos ≠
        ≠ NULL then O(1)
        res ← true O(1)
    end if O(1)
end while O(1)

```

Complejidad: $O(4) = O(1)$, Como el conjunto c tiene maximo 4 posiciones, solo hace el ciclo 4 veces maximo.

TodosEstudiantes (in c: conj(posicion), in cs: estr) → res: bool

```

itConj(posicion): itC ← CrearIt(c) O(1)
res ← true O(1)

while HaySiguiente(itC) AND res = true do O(1)
    if cs.posicionesEstudiantes[Siguiente(itC).y * Columnas(cs.campus) +
        + Siguiente(itC).x] = " " then O(1)
        res ← false O(1)
    end if O(1)

    Avanzar(itC) O(1)
end while O(1)

```

Complejidad: $O(1)$, Como el conjunto c tiene como maximo longitud 4, la complejidad sería $O(1)$.

```

ingresarHippie (in h: nombre, in pos: posicion, in/out cs: estr)
    if todasOcupadas?(vecinos(pos, cs.campus), cs) AND O(1)
        AND AlMenosUnAgente(vecinos(pos, cs.campus)) then O(1)
            conj(As): conjAgParaPrem ← AgParaPremSanc(vecinos(pos, cs.campus), cs) O(1)
            PremiarAgentes(conjAgParaPrem, cs) O(1)

            else if todasOcupadas?(vecinos(pos, cs.campus), cs) AND O(1)
                AND TodosEstudiantes(vecinos(pos, cs.campus), cs) then O(1)
                    Definir(cs.estudiantes, h, pos) O(|n_m|)
                    cs.posicionesEstudiantes[pos.y * Columnas(cs.campus) + pos.x] ← h O(|n_m|)

            else
                Definir(cs.hippies, h, pos) O(|n_m|)
                cs.posicionesHippies[pos.y * Columnas(cs.campus) + pos.x] ← h O(|n_m|)

        end if
    end if

```

```

conj(nombre, posicion): conjHippiesRodEst ← O(1)
    ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjHippiesRodEst) > 0 then O(1)
    itConj(nombre, posicion): itHEst ← CrearIt(conjHippiesRodEst) O(1)

    while HaySiguiente(itHEst) do O(1)
        Definir(cs.estudiantes, Siguiente(itHEst).nombre,
            Siguiente(itHEst).posicion) O(|n_m|)
        Eliminar(cs.hippies, Siguiente(itHEst).nombre) O(|n_m|)

        cs.posicionesEstudiantes[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
            + Siguiente(itHEst).posicion.x] ←
            ← cs.posicionesHippies[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
            + Siguiente(itHEst).posicion.x]

```

```

+ Siguiente(itHest).posicion.x] O(1)
cs.posicionesHippies[Siguiente(itHest).posicion.y*Columnas(cs.campus) +
+ Siguiente(itHest).posicion.x] ← " " O(1)

Avanzar(itHest) O(1)
end while O(4 * 2|nm|)
end if O(4 * 2|nm|)

conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) O(1)
if Cardinal(conjHippiesRodAs) > 0 then O(1)
    itConj(nombre, posicion): itHAs ← CrearIt(conjHippiesRodAs) O(1)
    while HaySiguiente(itHAs) do O(1)
        Eliminar(cs.hippies, Siguiente(itHAs).nombre) O(|nm|)

        cs.posicionesHippies[Siguiente(itHAs).y*Columnas(cs.campus) +
        + Siguiente(itHAs).x] ← " " O(1)
        Avanzar(itHAs) O(1)
    end while O(4|nm|)
end if O(4|nm|)

conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) O(1)
if Cardinal(conjEstRodHip) > 0 then O(1)
    itConj(nombre, posicion): itEstH ← CrearIt(conjEstRodHip) O(1)
    while HaySiguiente(itEstH) do O(1)
        Eliminar(cs.estudiantes, Siguiente(itEstH).nombre) O(|nm|)

        cs.posicionesEstudiantes[Siguiente(itEstH).y*Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← " " O(1)

        Definir(cs.hippies, Siguiente(itEstH).nombre,
            Siguiente(itEstH).posicion) O(|nm|)

        cs.posicionesHippies[Siguiente(itEstH).y*Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← Siguiente(itEstH).nombre O(1)

        Avanzar(itHAs) O(1)
    end while O(4 * 2|nm|)
end if O(4 * 2|nm|)

conj(posicion): conjEstRodAs ←
    ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) O(1)
if Cardinal(conjEstRodAs) > 0 then O(1)
    itConj(posicion): itEstAs ← CrearIt(conjEstRodAs) O(1)

    while HaySiguiente(itEstAs) do O(1)
        if todasOcupadas?(vecinos(Siguiente(itEstAs), cs.campus), cs AND
            AND AlMenosUnAgente(vecinos(Siguiente(itEstAs), cs.campus)) then O(1)
            conj(As): conjAgParaSanc ←
                ← AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus), cs) O(1)
                SancionarAgentes(conjAgParaSanc, cs) O(1)
        end if O(1)
    end while O(4)
end if O(4)

```

Diagrama:
Un cuadro triangular que parece ser un diagrama de flujo o un organigrama. Tiene una flecha que apunta hacia la derecha con la palabra "Avanzar". A su lado, en la parte superior, dice "y la siguiente?", y debajo de la flecha, "se cumplió?".

Complejidad : $O(|n_m|)$

iMoverEstudiante (in e: nombre, in d: dirección, in/out cs: estr)

```

Posicion: actualPos ← Obtener(cs.estudiantes, e) O(|n_m|)
Posicion: pos ← actualPos O(1)
if (d=Izquierda) then O(1)
    pos.x ← pos.x - 1 O(1)
else if (d=derecha) then O(1)
    pos.x ← pos.x + 1 O(1)
else if (d=Arriba) then O(1)
    pos.y ← pos.y + 1 O(1)
else if (d=Abajo) then O(1)
    pos.y ← pos.y - 1 O(1)
end if

if (not (pos.y = 1 OR pos.y = cd.campus.filas)) then O(1)
    if CantHippiesVecinos(vecinos(pos, cs.campus), cs) < 2 then O(|n_m|)
        Definir(cs.estudiantes, e, pos) O(|n_m|)
        cs.posicionesEstudiantes[actualPos.y * Columnas(cs.campus) + actualPos.x] ← "" O(1)
        cs.posicionesEstudiantes[pos.y * Columnas(cs.campus) + pos.x] ← e O(1)
    else
        Definir(cs.hippies, e, pos) O(|n_m|)
        cs.posicionesHippies[pos.y * Columnas(cs.campus) + pos.x] ← e O(1)
        cs.posicionesEstudiantes[actualPos.y * Columnas(cs.campus) + actualPos.x] ← "" O(1)
        Borrar(cs.estudiantes, e) O(|n_m|)
    end if O(|n_m| + |n_m|)

conj(nombre, posicion): conjHippiesRodEst ←
    ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjHippiesRodEst) > 0 then O(1)
    itConj(nombre, posicion): itHEst ← CrearIt(conjHippiesRodEst) O(1)

    while HaySiguiente(itHEst) do O(1)
        Definir(cs.estudiantes, Siguiente(itHEst).nombre,
            Siguiente(itHEst).posicion) O(|n_m|)
        Eliminar(cs.hippies, Siguiente(itHEst).nombre) O(|n_m|)

        cs.posicionesEstudiantes[Siguiente(itHEst).posicion.y * Columnas(cs.campus) + Siguiente(itHEst).posicion.x] ← cs.posicionesHippies[Siguiente(itHEst).posicion.y * Columnas(cs.campus) + Siguiente(itHEst).posicion.x] O(1)

        cs.posicionesHippies[Siguiente(itHEst).posicion.y * Columnas(cs.campus) + Siguiente(itHEst).posicion.x] ← "" O(1)

        Avanzar(itHEst) O(1)
    end while O(2|n_m|)
end if O(4 * 2|n_m|)

conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) O(1)
if Cardinal(conjHippiesRodAs) > 0 then O(1)
    itConj(nombre, posicion): itHAS ← CrearIt(conjHippiesRodAs) O(1)

```

```

while HaySiguiente(itHAs) do
    Eliminar(cs.hippies, Siguiente(itHAs).nombre) O(1)
    cs.posicionesHippies[Siguiente(itHAs).y * Columnas(cs.campus) +
    + Siguiente(itHAs).x] ← "" O(1)
    Avanzar(itHAs) O(1)
end while O(4 * |nm|)
end if O(4 * |nm|)

conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) O(1)
if Cardinal(conjEstRodHip) > 0 then O(1)
    itConj(nombre, posicion): itEstH ← CrearIt(conjEstRodHip) O(1)
    while HaySiguiente(itEstH) do O(1)
        Eliminar(cs.estudiantes, Siguiente(itEstH).nombre) O(|nm|)
        cs.posicionesEstudiantes[Siguiente(itEstH).y * Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← "" O(1)

        Definir(cs.hippies, Siguiente(itEstH).nombre,
        Siguiente(itEstH).posicion) O(|nm|)

        cs.posicionesHippies[Siguiente(itEstH).y * Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← Siguiente(itEstH).nombre O(1)

        Avanzar(itHAs) O(1)
    end while O(4 * 2|nm|)
end if O(4 * 2|nm|)

conj(posicion): conjEstRodAs ←
    ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) O(1)
if Cardinal(conjEstRodAs) > 0 then O(1)
    itConj(posicion): itEstAs ← CrearIt(conjEstRodAs) O(1)

    while HaySiguiente(itEstAs) do O(1)
        if todasOcupadas?(vecinos(Siguiente(itEstAs), cs.campus), cs AND
            AND AlMenosUnAgente(vecinos(Siguiente(itEstAs), cs.campus)) then O(1)
            conj(As): conjAgParaSanc ←
                AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus), cs) O(1)
                SancionarAgentes(conjAgParaSanc, cs) O(1)
            end if O(1)
        end while O(4)

        end if O(4)
    else
        Borrar(cs.estudiantes, e) O(|nm|)
        cs.posicionesEstudiantes[actualPos.y * Columnas(cs.campus) + actualPos.x] ← "" O(1)
    end if O(1)

```

Complejidad : $O(|n_m|)$

iMoverHippie (in h: nombre, in/out cs: estr)

```

posicion : actualPos ← obtener(h, cs.hippies) O(|nm|)
posicion : pos ← porxPos(actualPos, cs.estudiantes, cs) O(Nc)
if actualPos ≠ pos then O(1)

```

```

definir(cs.hippies, h, pos) O(|n_m|)

cs.posicionesHippies[actualPos.y * columnas(cs.campus)
+ actualPos.x] ← " "
O(1)

cs.posicionesHippies[pos.y * columnas(cs.campus) + pos.x] ← h
O(1)

conj(nombre, posicion): conjHippiesRodEst ←
    ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjHippiesRodEst) > 0 then O(1)
    itConj(nombre, posicion): itHEst ← CrearIt(conjHippiesRodEst) O(1)

    while HaySiguiente(itHEst) do O(1)
        Definir(cs.estudiantes, Siguiente(itHEst).nombre,
        Siguiente(itHEst).posicion) O(|n_m|)

        Eliminar(cs.hippies, Siguiente(itHEst).nombre) O(|n_m|)

        cs.posicionesEstudiantes[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
        + Siguiente(itHEst).posicion.x] ← " "
        ← cs.posicionesHippies[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
        + Siguiente(itHEst).posicion.x] ← " " O(1)

        cs.posicionesHippies[Siguiente(itHEst).posicion.y * Columnas(cs.campus) +
        + Siguiente(itHEst).posicion.x] ← " "
        O(1)

        Avanzar(itHEst) O(1)

    end while
end if O(2|n_m|) O(4 * 2|n_m|)

conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjHippiesRodAs) > 0 then O(1)
    itConj(nombre, posicion): itHAS ← CrearIt(conjHippiesRodAs) O(1)

    while HaySiguiente(itHAS) do O(1)
        Eliminar(cs.hippies, Siguiente(itHAS).nombre) O(|n_m|)

        cs.posicionesHippies[Siguiente(itHAS).y * Columnas(cs.campus) +
        + Siguiente(itHAS).x] ← " "
        Avanzar(itHAS) O(1)

    end while
end if O(4|n_m|) O(4|n_m|)

conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) O(1)

if Cardinal(conjEstRodHip) > 0 then O(1)
    itConj(nombre, posicion): itEstH ← CrearIt(conjEstRodHip) O(1)

    while HaySiguiente(itEstH) do O(1)
        Eliminar(cs.estudiantes, Siguiente(itEstH).nombre) O(|n_m|)

        cs.posicionesEstudiantes[Siguiente(itEstH).y * Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← " "
        O(1)

        Definir(cs.hippies, Siguiente(itEstH).nombre,
        Siguiente(itEstH).posicion) O(|n_m|)

        cs.posicionesHippies[Siguiente(itEstH).y * Columnas(cs.campus) +
        + Siguiente(itEstH).x] ← Siguiente(itEstH).nombre O(1)

        Avanzar(itHAS) O(1)

```

*Falta averiguar las SS
a evaluar las SS
de la otra parte*

```

    end while
end if

conj(posicion): conjEstRodAs ←
    ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) O(4 * 2|nm|)

if Cardinal(conjEstRodAs) > 0 then O(4 * 2|nm|)
    itConj(posicion): itEstAs ← CrearIt(conjEstRodAs) O(1)

    while HaySiguiente(itEstAs) do O(1)
        if todasOcupadas?(vecinos(Siguiente(itEstAs), cs.campus), cs AND
            AND AlMenosUnAgente(vecinos(Siguiente(itEstAs), cs.campus)) then O(1)
            conj(As): conjAgParaSanc ←
                ← AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus), cs) O(1)
                SancionarAgentes(conjAgParaSanc, cs) O(1)
        end if O(1)
    end while O(4)

end if O(4)
end if O(4)

```

Complejidad: $O(N_c + |n_m|)$

ProxPos (in pos: posicion, in d: dicString(nombre, posicion), in cs: estr) → res: posicion

```

itDicString(nombre, posicion): itDic ← CrearIt(d) O(1)
int: distCorta ← DistanciaMasCorta(pos, d) O(#Claves(d))

conj(posicion): conjDondeIr ← DondeIr(pos, distCorta, d) O(#Claves(d))
conj(posicion): conjLugaresPosibles ←
    ← LugaresPosibles(pos, conjDondeIr, cs) O(Cardinal(c))

if Cardinal(conjLugaresPosibles) = 0 then O(1)
    res ← pos O(1)
else
    itConj(posicion): itC ← CrearIt(conjLugaresPosibles) O(1)
    res ← Siguiente(itC) O(1)
end if O(1)

```

Complejidad: $O(\#Claves(d)) + Cardinal(c) = O(\#Claves(d))$. El Cardinal(c) depende de las claves del diccionario.

LugaresPosibles (in pos: posicion, in c: conj(posicion), in cs: estr) → res: conj(posicion)

```

itConj(posicion): itC ← CrearIt(c) O(1)
res ← Vacio() O(1)

while HaySiguiente(itC) do O(1)
    if Siguiente(itC).x > pos.x AND Siguiente(itC).y > pos.y
        AgregarRapido(res, tupla(pos.x+1, pos.y)) O(1)
        AgregarRapido(res, tupla(pos.x, pos.y+1)) O(1)
    else if Siguiente(itC).x = pos.x AND Siguiente(itC).y > pos.y
        AgregarRapido(res, tupla(pos.x, pos.y+1)) O(1)
    else if Siguiente(itC).x < pos.x AND Siguiente(itC).y > pos.y
        AgregarRapido(res, tupla(pos.x-1, pos.y)) O(1)
        AgregarRapido(res, tupla(pos.x, pos.y+1)) O(1)
    else if Siguiente(itC).x < pos.x AND Siguiente(itC).y = pos.y
        AgregarRapido(res, tupla(pos.x-1, pos.y)) O(1)

```

```

else if Siguiente(itC).x < pos.x AND Siguiente(itC).y < pos.y O(1)
    AgregarRapido(res, tupla(pos.x-1, pos.y))
    AgregarRapido(res, tupla(pos.x, pos.y-1))
else if Siguiente(itC).x = pos.x AND Siguiente(itC).y < pos.y O(1)
    AgregarRapido(res, tupla(pos.x, pos.y-1))
else if Siguiente(itC).x < pos.x AND Siguiente(itC).y < pos.y O(1)
    AgregarRapido(res, tupla(pos.x-1, pos.y))
    AgregarRapido(res, tupla(pos.x, pos.y-1))
else if Siguiente(itC).x > pos.x AND Siguiente(itC).y = pos.y O(1)
    AgregarRapido(res, tupla(pos.x+1, pos.y))
end if O(1)

Avanzar(itC) O(1)
end while O(Cardinal(c))

itConj(posicion) itPosibles ← CrearIt(res) O(1)

while HaySiguiente(itPosibles) do O(1)
    if HayAlgoEnPos(Siguiente(itPosibles), cs) then O(1)
        EliminarSiguiente(itPosibles) O(1)
    end if O(1)
    Avanzar(itPosibles) O(1)
end while O(Cardinal(c))

```

Complejidad : $Cardinal(c)$

```

HayAlgoEnPos (in pos: posicion, in cs: estr) → res: bool
    res ← false O(1)

    if cs.posicionesAgente[pos.y*Columnas(cs.campus) + pos.x].datos ≠ NULL then O(1)
        res ← true O(1)
    end if O(1)

    if cs.posicionesHippies[pos.y*Columnas(cs.campus) + pos.x].datos ≠ " " then O(1)
        res ← true O(1)
    end if O(1)

    if cs.posicionesEstudiantes[pos.y*Columnas(cs.campus) + pos.x].datos ≠ " " then O(1)
        res ← true O(1)
    end if O(1)

    if Ocupada?(pos, cs.campus) then O(1)
        res ← true O(1)
    end if O(1)

```

Complejidad : $O(1)$

DondeIr (in pos: posicion, in dist: nat, in d: diccString(nombre, posicion)) → res: conj(posicion)

```

res ← Vacio() O(1)
itDiccString(nombre, posicion): itDicc ← CrearIt(d) O(1)

while HaySiguiente(itDicc) do O(1)
    if dist = Distancia(pos, Siguiente(itDicc)) then O(1)
        AgregarRapido(res, Siguiente(itDicc)) O(1)

```

```

    end if
    Avanzar(itDicc)
end while

```

$O(1)$
 $O(1)$
 $O(\#Claves(d))$

Complejidad: $O(\#Claves(d))$

DistanciaMasCorta (in pos: posicion, in d: dicString(nombre, posicion)) → res: int

```

itDiccString(nombre, posicion): itDicc ← CrearIt(d)
res ← Distancia(pos, Siguiente(itDicc))
Avanzar(itDicc)

while HaySiguiente(itDicc) do
    if res > Distancia(pos, Siguiente(itDicc))
        res ← Distancia(pos, Siguiente(itDicc))
    end if
    Avanzar(itDicc)
end while

```

$O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(\#Claves(d))$

Complejidad: $O(\#Claves(d))$

Distancia (in pos1, pos2: posicion) → res: nat

```

res ← |pos2.x - pos1.x| + |pos2.y - pos1.y|

```

$O(1)$

Complejidad: $O(1)$

iMoverAgente (in a: agente, in/out cs: estr)

```

vector(tupla(agente, puntero(datosAgente)))
placas ← ordenadoPorClave(cs.personalAS)
puntero(datosAgente) as ← busqBinPorPlaca(placas, a)

```

$O(1)$
 $O(\log(N_a))$

// Actualizo la posicion del agente

```

posicion nuevaPos ← ProxPos((*as).posicion, cs.hippies, cs)
posicionesAgentes[((*as).posicion.y * columnas(cs.campus) + (*as).posicion.x), datos
    ← NULL
posicionesAgentes[nuevaPos.y * columnas(cs.campus) + nuevaPos.x]
    ← tupla(a, as)
(*as).posicion ← nuevaPos

```

$O(N_h)$
 $O(1)$
 $O(1)$
 $O(1)$

// Me fijo a quienes atrapa

```

posicion posArr ← moverDir(cs.campus, nuevaPos, arriba)
actualizarAgente(cs, posArr, a, as)

```

$O(1)$
 $O(\log(N_a) + |n_m| + N_h)$

```

posicion posAba ← moverDir(cs.campus, nuevaPos, abajo)
actualizarAgente(cs, posAba, a, as)

```

$O(1)$
 $O(\log(N_a) + |n_m| + N_h)$

```

posicion posDer ← moverDir(cs.campus, nuevaPos, der)
actualizarAgente(cs, posDer, a, as)

```

$O(1)$
 $O(\log(N_a) + |n_m| + N_h)$

```

posicion posIzq ← moverDir(cs.campus, nuevaPos, izq)
actualizarAgente(cs, posIzq, a, as)

```

$O(1)$
 $O(\log(N_a) + |n_m| + N_h)$

Complejidad : $O(\log(N_a) + |n_m| + N_h)$

```

actualizarAgente (in/out cs: estr, in posicion: pos, in a: agente, in puntero(datosAgente): as)
if (posValida?(cs.campus, pos) then
    if (posicionesHippies[pos.y * columnas(cs.campus) + pos.x] ≠ "") then
        if (atrapado?(cs, pos)) then
            // Le sumo uno a sus capturas, actualizo masVigilante y mato al hippie
            (*as).cantAtrapados++;
            if ((*as).cantAtrapados > (*cs.masVigilante.datos).cantAtrapados) then
                cs.masVigilante ← tupla(a, as)
            end if
            borrar(cs.hippies, posicionesHippies[pos.y *
                columnas(cs.campus) + pos.x])
            posicionesHippies[pos.y * columnas(cs.campus) + pos.x]
            ← ""
        end if
    end if
end if

if (posicionesEstudiantes[pos.y * columnas(cs.campus) + pos.x] ≠ "") then
    if (atrapado?(cs, pos)) then
        // Actualizo las sanciones y las estructuras relacionadas
        cs.mismasSancModificado ← true
        itConj(agente) iterConj ← (*as).itConjMismasSanc
        EliminarSiguiente(iterConj)

        itLista(kSanc) iterLista ← (*as).itMismasSanc
        // Me guardo un iterador para borrar el nodo de la lista
        // si es que queda sin agentes
        itLista(kSanc) iterListaAnterior ← (*as).itMismasSanc
        if (HaySiguiente?(iterLista)) then
            // Me fijo si el siguiente es la siguiente sancion
            nat : sanciones ← Siguiente(iterLista).sanc
            Avanzar(iterLista)

            if (Siguiente(iterLista).sanc = sanciones + 1) then
                // Lo agrego al conjunto
                (*as).itConjMismasSanc ←
                    AgregarRapido(a, Siguiente(iterLista).agentes)
            else
                // Creo un nuevo nodo en el medio
                conj(agente) conj ← Vacio()
                (*as).itConjMismasSanc ←
                    AgregarRapido(a, conj)
                kSanc nodo ← tupla(sanciones + 1, conj)
                AgregarComoAnterior(iterLista, nodo)
                Retroceder(iterLista)
                (*as).itMismasSanc ← iterLista
            end if
        else
            // Creo un nuevo nodo
            conj(agente) conj ← Vacio()
            (*as).itConjMismasSanc ←
                AgregarRapido(a, conj)
            kSanc nodo ← tupla(sanciones + 1, conj)
    end if
end if

```

```

    AgregarComoSiguiente(iterLista, nodo)           O(1)
        Avanzar(iterLista)
        (*as.).itMismasSanc  $\leftarrow$  iterLista
    end if

    if (!HaySiguiente?(iterConj)) then             O(1)
        // Borro el nodo anterior de la lista porque no tiene agentes
        EliminarSiguiente(iterListaAnterior)
    end if

    (*as.).cantSanc++;
end if
end if
Complejidad :  $O(\log(N_n) + |n_m| + N_b)$ 

```

iCampus (in $cs: estr$) \rightarrow res: campus
 $res \leftarrow cs.campus$

O(1)

iEstudiantes (in $cs: estr$) \rightarrow res: itDicString(nombre, posición)
 $res \leftarrow CrearIt(cs.estudiantes)$

O(1)

iHippies (in $cs: estr$) \rightarrow res: itDicString(nombre, posición)
 $res \leftarrow CrearIt(cs.hippies)$

O(1)

iAgentes (in $cs: estr$) \rightarrow res: itDicNat(agente, datosAgente)
 $res \leftarrow CrearIt(cs.personalAS)$

O(1)

iPosEstudianteYHippie (in $id: nombre$, in $cs: estr$) \rightarrow res: posición
if Definido?($id, cs.hippies$) then
 $res \leftarrow Obtener(id, cs.hippies)$
else
 $res \leftarrow Obtener(id, cs.estudiantes)$
end if

 $O(|n_m|)$ $O(|n_m|)$ $O(|n_m|)$

Complejidad : $O(|n_m|) + O(4 * 2|n_m|) = O(|n_m|)$

O(1 caso promedio)

iPosAgente (in $a: agente$, in $cs: estr$) \rightarrow res: posición
 $res \leftarrow Obtener(a, cs.personalAS).posicion$

O(1 caso promedio)

Complejidad : $O(1)$ caso promedio

O(1 caso promedio)

iCantHippiesAtrapados (**in** a : agente, **in** cs : estr)
 $res \leftarrow$ Obtener(a , cs . personalAS).cantAtrapados
Complejidad: $O(1)$ caso promedio

O(1) caso promedio

iConMismasSanciones (**in** a : agente, **in** cs : estr) \rightarrow res: lista(agente)
 $res \leftarrow$ Obtener(a , cs . personalAS).itMismasSanc.conjAgente
Complejidad: $O(1)$ caso promedio

O(1) caso promedio

iConKSanciones (**in** k : nat, **in** cs : estr) \rightarrow res: lista(agente)
if cs . mismasSancModificado = true **then**
 hacerArregloMismasSanc(cs)
 cs . mismasSancModificado \leftarrow false
end if

 nat: $i \leftarrow 0$
 bool: $esta \leftarrow$ busqBinAgente(k , i , cs . vectorMismasSanc)
if $esta = \text{true}$ **then**
 $res \leftarrow *cs$. vectorMismasSanc[i]. agentes
else
 $res \leftarrow$ Vacia()
end if

*O(1)**O(N_a)**O(1)**O(1)**O(1)**O(1)**O(1)**O(1)**O(1)**O(1)*

Complejidad: Si las sanciones no fueron modificadas: $O(N_a)$. Si no $O(\log N_a)$

hacerArregloMismasSanc (**in/out** cs : estr)
 $arreglo$ (**puntero**(kSanc)): $arregloNuevo \leftarrow$ CrearArreglo (Longitud(cs . listaMismasSanc))
 $O(N_a)$
 $itLista$ (kSanc): $it \leftarrow$ CrearIt(cs . listaMismasSanc)
 nat: $i \leftarrow 0$

while HaySiguiente(it) **do**
 p \leftarrow puntero(Siguiente(it))
 $arregloNuevo[i] \leftarrow p$
 $i \leftarrow i + 1$
 Avanzar(it)
end while

 cs . arregloMismasSanc \leftarrow arregloNuevo

*O(1)**O(1)**O(1)**O(1)**O(1)**O(1)**O(N_a)**O(1)*

Complejidad: $O(N_a) + O(N_a) = O(N_a)$ Como máximo va a ser la cantidad de agentes la lista listaMismasSanc

busqBinAgente (**in** k : nat, **in/out** i : nat, **in** v : arreglo(puntero(kSanc))) \rightarrow res: bool
 nat: $n \leftarrow 0$
 nat: $m \leftarrow$ Longitud(v)
 nat: med

while $n \neq m - 1$ **do**
 $med \leftarrow \frac{n+m}{2}$
 if $med \leq k$ **then**
 $n \leftarrow med$
 else
 $m \leftarrow med$

*O(1)**O(1)**O(1)**O(1)**O(1)**O(1)**O(1)**O(1)*

```

    m ← med
end if
end while
if v[n] = k then
    i ← n
    res ← true
else
    res ← false
end if

```

Complejidad : $O(\log N_s)$ $O(1)$ $O(\log(N_s))$

Atrapado? (in c: campus, in pos: Posicion) → res: bool
 $\text{res} \leftarrow \text{todasOcupadas?}(\text{vecinos}(\text{pos}, \text{cs}, \text{campus}))$

 $O(1)$

busqBinPorPlaca (in a: agente, in v: vector(tupla(clave : agente, p : puntero(datosAgente))) → res: puntero(datosAgente)

```

nat: inf ← 0
nat: sup ← Longitud(v)
nat: med ←  $\frac{inf+sup}{2}$ 
while inf ≠ sup-1 do
    med ←  $\frac{inf+sup}{2}$ 
    if v[med].clave ≤ a then
        inf ← med
    else
        sup ← med
    end if
end while
res ← v[inf].p

```

Complejidad : $O(\log N_s)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(\log(N_s))$

2. Módulo Campus

2.1. Interfaz

se explica con: CAMPUS.

géneros: campus.

2.1.1. Operaciones básicas de Campus

CREARCAMPUS(in *alto*: nat, in *ancho*: nat) → *res* : campus

Pre ≡ {true}

Post ≡ {*res* =_{obs} crearCampus(*alto*, *ancho*)}

Complejidad: $O((alto * ancho)^2)$

Descripción: Crea un nuevo campus tomando un alto y un ancho

AGREGAROBSTACULO(in/out *c*: campus, in *pos*: posición)

Pre ≡ {*c* =_{obs} *c*₀ ∧ posValida?(*pos*, *c*) ∧_L ¬ocupada?(*pos*, *c*)}

Post ≡ {*c* =_{obs} agregarObstaculo(*pos*, *c*₀)}

Complejidad: $O(1)$

Descripción: Agrega un obstáculo al campus

FILAS(in *c*: campus) → *res* : nat

Pre ≡ {true}

Post ≡ {*res* =_{obs} filas(*c*)}

Complejidad: $O(1)$

Descripción: Devuelve la cantidad de filas del campus

COLUMNAS(in *c*: campus) → *res* : nat

Pre ≡ {true}

Post ≡ {*res* =_{obs} columnas(*c*)}

Complejidad: $O(1)$

Descripción: Devuelve la cantidad de columnas del campus

Ocupada?(in *c*: campus, in *pos*: posición) → *res* : bool

Pre ≡ {posValida(*pos*, *c*)}

Post ≡ {*res* =_{obs} ocupada(*pos*, *c*)}

Complejidad: $O(1)$

Descripción: Devuelve true si la posición está ocupada por un obstáculo

POSVALIDA?(in *c*: campus, in *pos*: posición) → *res* : bool

Pre ≡ {true}

Post ≡ {*res* =_{obs} posValida(*c*, *pos*)}

Complejidad: $O(1)$

Descripción: Devuelve true si la posición es válida

ESINGRESO?(in *c*: campus, in *pos*: posición) → *res* : bool

Pre ≡ {true}

Post ≡ {*res* =_{obs} esIngreso(*pos*, *c*)}

Complejidad: $O(1)$

Descripción: Devuelve true si la posición es un ingreso. No tiene en cuenta su validez

INGRESOSUPERIOR?(in c : campus, in pos : posicion) $\rightarrow res$: bool

Pre \equiv {true}

Post \equiv { $res =_{obs}$ ingresoSuperior?(pos, c)}

Complejidad: $O(1)$

Descripción: Devuelve true si la posicion es un ingreso superior. No tiene en cuenta su validez

INGRESOINFERIOR?(in c : campus, in pos : posicion) $\rightarrow res$: bool

Pre \equiv {true}

Post \equiv { $res =_{obs}$ ingresoInferior?(pos, c)}

Complejidad: $O(1)$

Descripción: Devuelve true si la posicion es un ingreso inferior. No tiene en cuenta su validez

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

Pre \equiv {posValida?(pos, c)}

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

Complejidad: $O(1)$

Descripción: Devuelve el conjunto posiciones validas adyacentes a pos

Aliasing: res es una referencia no modificable

DISTANCIA(in c : campus, in $pos1$: posicion, in $pos2$: posicion) $\rightarrow res$: nat

Pre \equiv {true}

Post \equiv { $res =_{obs}$ distancia($pos1, pos2, c$)}

Complejidad: $O(1)$

Descripción: Devuelve la distancia entre dos posiciones

MOVERDIR(in c : campus, in pos : posicion, in dir : direccion) $\rightarrow res$: posicion

Pre \equiv {posValida(pos, c)}

Post \equiv { $res =_{obs}$ proxPosicion(pos, dir, c)}

Complejidad: $O(1)$

Descripción: Devuelve la posicion resultante al avanzar en la direccion pasada como parametro. No tiene en cuenta su validez

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

Pre \equiv {posValida(pos, c)}

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

Complejidad: $O(1)$

Descripción: Devuelve un conjunto con las posiciones de los ingresos mas cercanos

Aliasing: res es una referencia no modificable

2.1.2. Representación de campus

campus se representa con estr

donde estr es tupla(filas: nat, columnas: nat, obstaculos: vector(bool))

2.1.3. Invariante de Representación

- (I) El largo del vector es igual al producto de filas por columnas

Rep : estr → bool

Rep(e) ≡ true ⇔ long(obstaculos) ≤ filas * columnas

2.1.4. Función de Abstracción

Abs : estr e → campus

{Rep(e)}

Abs(e) =_{obs} cs: campus |

filas(cs) = e.filas ∧
 columnas(cs) = e.columnas ∧
 $(\forall pos: \text{posicion})(\text{posValida?}(pos, cs) \Rightarrow_L$
 $(\text{ocupada?}(pos, cs) = e.\text{obstaculos}[pos.y * e.\text{columnas} + pos.x]))$

2.2. Algoritmos

crearCampus (in alto: nat, in ancho: nat) → res: estr

```

res.filas ← alto
res.columnas ← ancho

nat : pos ← 0
while(pos < alto * ancho) do
    AgregarAtras(res.obstaculos, false)
end while
res.obstaculos ← Vacio()

```

Complejidad : $O(1)$

O(1)
 O(1)
 O(1)
 O(1)
 $O(alto * ancho)$
 $O((alto * ancho)^2)$
 O(1)

agregarObstaculo (in/out cs: campus, in pos: posicion)

```
cs.obstaculos[pos.y * cs.columnas + pos.x] ← true
```

O(1)

Complejidad : $O(1)$

filas (in cs: campus) → res: nat

```
res ← cs.filas
```

O(1)

Complejidad : $O(1)$

columnas (in *cs*: campus) → res: nat

res ← *cs*.columnas

O(1)

Complejidad: $O(1)$

ocupada? (in *cs*: campus, in *pos*: posicion) → res: bool

res ← *cs*.obstaculos [*pos*.y * *cs*.columnas + *pos*.x]

O(1)

Complejidad: $O(n)$

posValida? (in *cs*: campus, in *pos*: posicion) → res: bool

res ← (*pos*.x ≥ 0) ∧ (*pos*.x < *cs*.columnas) ∧
(*pos*.y ≥ 0) ∧ (*pos*.y < *cs*.filas)

O(1)

Complejidad: $O(1)$

esIngreso (in *cs*: campus, in *pos*: posicion) → res: bool

res ← esIngresoSuperior(*cs*, *pos*) ∨ esIngresoInferior(*cs*, *pos*)

O(1)

Complejidad: $O(1)$

esIngresoSuperior (in *cs*: campus, in *pos*: posicion) → res: bool

res ← *pos*.y = 0

O(1)

Complejidad: $O(1)$

esIngresoInferior (in *cs*: campus, in *pos*: posicion) → res: bool

res ← *pos*.y = *cs*.filas - 1

O(1)

Complejidad: $O(1)$

vecinos (in *cs*: campus, in *pos*: posicion) → res: conj(posicion)

res ← Vacio()

O(1)

if (*posValida*(*cs*, tupla(*pos*.x + 1, *pos*.y)))

O(1)

AgregarRapido(*res*, tupla(*pos*.x + 1, *pos*.y))

O(1)

if (*posValida*(*cs*, tupla(*pos*.x - 1, *pos*.y)))

O(1)

AgregarRapido(*res*, tupla(*pos*.x - 1, *pos*.y))

O(1)

```

if (posValida(cs, tupla(pos.x, pos.y + 1)))
    AgregarRapido(res, tupla(pos.x, pos.y + 1)) O(1)
if (posValida(cs, tupla(pos.x, pos.y - 1)))
    AgregarRapido(res, tupla(pos.x, pos.y - 1)) O(1)

```

Complejidad : $O(1)$

distancia (in cs: campus, in pos₁: posicion, in pos₂: posicion) → res: nat

```
res ← |pos1.x - pos2.x| + |pos1.y - pos2.y|
```

$O(1)$

Complejidad : $O(1)$

moverDir (in cs: campus, in pos: posicion, in dir: direccion) → res: posicion

```

if (dir = izq)
    res ← tupla(pos.x - 1, pos.y)
if (dir = der)
    res ← tupla(pos.x + 1, pos.y)
if (dir = arriba)
    res ← tupla(pos.x, pos.y - 1)
if (dir = abajo)
    res ← tupla(pos.x, pos.y + 1)

```

$O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$

Complejidad : $O(1)$

ingresosMasCercanos (in cs: campus, in pos: posicion) → res: conj(posicion)

```

res ← Vacio()
if (distancia(cs, pos, tupla(pos.x, 0)) <
    distancia(cs, pos, tupla(pos.x, cs.filas - 1))) O(1)
    AgregarRapido(res, tupla(pos.x, 0)) O(1)
else
    if (distancia(cs, pos, tupla(pos.x, 0)) >
        distancia(cs, pos, tupla(pos.x, cs.filas - 1))) O(1)
        AgregarRapido(res, tupla(pos.x, cs.filas - 1)) O(1)
    else
        AgregarRapido(res, tupla(pos.x, 0)) O(1)

```

Complejidad : $O(1)$

3. Módulo Diccionario Nat Fijo

3.1. Interfaz

se explica con: $\text{DICCCIONARIO}(\text{NAT}, \alpha)$.

géneros: $\text{DiccNat}(\alpha)$, $\text{itDiccNat}(\alpha)$.

3.1.1. Operaciones básicas de $\text{DiccNat}(\alpha)$

$\text{CREARDICCIONARIO}(\text{in } v: \text{vector}(\text{tupla}(\text{clave : nat}, \text{significado : } \alpha))) \rightarrow res : \text{DiccNat}(\alpha)$

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

Post $\equiv \{((\forall t : \text{tupla}(\text{nat}, \alpha)) \text{ esta?}(t, v)) \Rightarrow ((\text{definido?}(t.\text{clave}, res)) \wedge_L \text{obtener}(t.\text{clave}, res) =_{obs} t.\text{significado}) \wedge \text{cantClaves}(res) =_{obs} \text{longitud}(v)\}$

Complejidad: $O(n^2 + \text{copy}(\alpha) * n)$ donde n es el largo del vector

Descripción: Agrega las tuplas de clave-significado pasadas por parametro al diccionario d

Aliasing: Los elementos pasados por parámetros se copian al diccionario

$\text{REDEFINIR}(\text{in/out } d: \text{DiccNat}(\alpha), \text{ in } n: \text{nat}, \text{ in } a: \alpha)$

Pre $\equiv \{\text{definido?}(n, d)\}$

Post $\equiv \{\text{obtener}(n, d) =_{obs} a\}$

Complejidad: $O(1)$ donde i es 1 en el caso promedio y $\text{longitud}(d)$ en el peor caso.

$\text{OBTENER}(\text{in } n: \text{nat}, \text{ in } d: \text{DiccNat}(\alpha)) \rightarrow res : \text{puntero}(\alpha)$

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

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

Complejidad: $O(1)$ en caso promedio, $O(n)$ en peor caso

Descripción: Devuelve un puntero al significado de la clave pasada por parametro. Si no está definido, devuelve NULL

Aliasing: El puntero va ser una referencia al significado almacenado en el diccionario

$\text{DEFINIDO?}(\text{in } n: \text{nat}, \text{ in } d: \text{DiccNat}(\alpha)) \rightarrow res : \text{bool}$

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

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

Complejidad: $O(1)$ en caso promedio, $O(n)$ en peor caso

Descripción: Dice si está definida una clave en el diccionario

$\text{CANTCLAVES}(\text{in } d: \text{DiccNat}(\alpha)) \rightarrow res : \text{Nat}$

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

Post $\equiv \{res =_{obs} \#\text{claves}(d)\}$

Complejidad: $O(1)$

Descripción: Devuelve la cantidad de claves definidas en el diccionario.

for que?
No justifican
los otros 2
complejidades

$\text{ORDENADOPORCLAVE}(\text{in } d: \text{DiccNat}(\alpha)) \rightarrow res : \text{vector}(\text{tupla}(\text{nat}, \text{puntero}(\alpha)))$

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

Post $\equiv \{\forall n : \text{nat} (\text{def?}(n, d) \Leftarrow \text{esta?}((n, \text{obtener}(n, d)), res)) \wedge \text{estaOrdenado}(res)\}$

Complejidad: $O(1)$

Descripción: Devuelve el vector ordenado de manera creciente según el natural de la clave

Aliasing: La información a la que apunta la segunda parte de la tupla es una referencia al significado correspondiente de la primera parte de la tupla

No!!

triangle
error de
no balanceamiento
este no debe
ir acá

3.1.2. Operaciones básicas del iterador

Este iterador permite recorrer la tabla de hash sobre la que está implementado el diccionario para obtener cada clave con su respectivo significado sin modificar ningún dato del diccionario.

CREARIT(in $d: \text{DiccNat}(\alpha)$) \rightarrow res : itDiccNat(α)

Pre \equiv {true}

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

Complejidad: $\Theta(1)$

Descripción: crea un iterador bidireccional del diccionario, de forma tal que HayAnterior evalúe a false (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).

Aliasing: El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique d sin utilizar las funciones del iterador.

HAYSIGUIENTE(in it : itDiccNat(α)) \rightarrow res : bool

Pre \equiv {true}

Post \equiv {res =_{def} haySigiente?(it)}

Complejidad: $\Theta(1)$

Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

SIGUIENTE(in it : itDiccNat(α)) \rightarrow res : tupla(nat, α)

Pre \equiv {haySigiente?(it)}

Post \equiv {alias(res =_{def} Siguiente(it))}

Complejidad: $\Theta(1)$

Descripción: devuelve el elemento siguiente del iterador.

Aliasing: res.sígnificado es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable.

SIGUIENTESIGNIFICADO(in it : itDiccNat(α)) \rightarrow res : α

Pre \equiv {haySigiente?(it)}

Post \equiv {alias(res =_{def} haySigiente?(it).significado)}

Complejidad: $\Theta(1)$

Descripción: devuelve el significado del elemento siguiente del iterador.

Aliasing: res es modificable si y sólo si it es modificable.

AVANZAR(in/out it : itDiccNat(α))

Pre \equiv {it = $i_0 \wedge$ haySigiente?(it)}

Post \equiv {it =_{def} avanzar(i_0)}

Complejidad: $\Theta(1)$

Descripción: avanza a la posición siguiente del iterador.

3.1.3. Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD DICC N AT(α) EXTENDIDO

extiende Diccionario(nat, α)

otras operaciones (no exportadas)

esPermutación? : secu(tupla(nat \times α)) \times diccNat(α) \longrightarrow bool

secuADiccNat : secu(tupla(nat \times α)) \longrightarrow diccNat(α)

axiomas

esPermutación?(s, d) \equiv d = secuADiccNat \wedge #claves = long(s)

```
secuADiccNat(s) ≡ if vacia?(s) then
    vacio
else
    definir(ptim(s).clave, primo(s).significado; secuADiccNat(fin(s)))
fi
```

Fin TAD

3.2. Representación de DiccNat(α)

DiccNat se representa con estr

donde estr es tupla(tabla: vector(lista(tupla(clave : nat, significado : α))),
 listaIterable: lista(puntero(tupla(clave : nat, significado : α))),
 vectorOrdenClaves: vector(tupla(clave : nat, significado : puntero(α))))

- (I) No existe dos veces el mismo nat en dos posiciones distintas del vector.
- (II) La suma del largo de todas las listas enlazadas que salen del vector, tiene que ser igual al largo del vector.
- (III) Toda tupla de la tabla es apuntado por un elemento de listaIterable.
- (IV) El largo de listaIterable es igual al largo del vector.
- (V) El vectorOrdenClaves tiene todas las claves de la tabla, el largo es igual al largo de la lista y los punteros van al significado de la clave correspondiente en la tabla

3.2.1. Invariante de Representación

Rep : estr \rightarrow bool

Rep(e) = true \Leftrightarrow

$$\begin{aligned}
 & (\forall l_1, l_2 : \text{lista}(\text{tupla}(\text{nat}, \alpha))) ((\text{esta?}(l_1, e.\text{tabla}) \wedge \text{esta?}(l_2, e.\text{tabla})) \\
 & \Rightarrow (\forall t_1, t_2 : \text{tupla}(\text{nat}, \alpha)) (\text{esta?}(t_1, l_1) \wedge \text{esta?}(t_2, l_2)) \\
 & \Rightarrow_L (\forall n_1, n_2 : \text{nat}) n_1 =_{obs} t_1.\text{clave} \wedge n_2 =_{obs} t_2.\text{clave} \wedge t_1 \neq t_2 \Rightarrow n_1 \neq n_2 \\
 & \wedge \\
 & \text{largosDeListas}(e.\text{tabla}) =_{obs} \text{longitud}(e.\text{tabla}) \\
 & \wedge \\
 & (\forall l : \text{lista}(\text{tupla}(\text{nat}, \alpha))) (\text{esta?}(l, e.\text{tabla}) \\
 & \Rightarrow (\forall t : \text{tupla}(\text{nat}, \alpha)) (\text{esta?}(t, l)) \\
 & \Rightarrow (\exists p : \text{puntero}(\alpha)) (\& t =_{obs} p \wedge \text{esta?}(p, e.\text{listaIterable}))) \\
 & \wedge \\
 & \text{long}(e.\text{tabla}) =_{obs} \text{long}(e.\text{listaIterable}) \\
 & \wedge \\
 & \text{long}(e.\text{vectorOrdenClaves}) =_{obs} \text{long}(e.\text{tabla}) \wedge (\forall l : \text{lista}(\text{tupla}(\text{nat}, \alpha))) (\text{esta?}(l, e.\text{tabla}) \\
 & \Rightarrow (\forall t : \text{tupla}(\text{nat}, \alpha)) (\text{esta?}(t, l)) \\
 & \Rightarrow (\forall n : \text{nat}) (n =_{obs} t.\text{clave})) \\
 & \Rightarrow_L (\exists p : \text{puntero}(\alpha)) (\& t.\text{significado} =_{obs} p \wedge \text{esta?}((t.\text{clave}, p), e.\text{vectorOrdenClaves}))
 \end{aligned}$$

largosDeListas : secu(secu(secu(tupla(nat \times α))) \rightarrow nat

largoDeListas(vector) \equiv if vacia?(vector) then 0 else longitud(prim(vector)) + largoDeLista(fin(vector)) fi

3.2.2. Función de Abstracción

Abs : estr e \rightarrow Diccionario

Abs(e) =_{obs} dicc: Diccionario | $(\forall l : \text{lista}(\text{tupla}(\text{nat}, \alpha))) ((\text{esta?}(l, e.\text{tabla})$
 $\Rightarrow (\forall t : \text{tupla}(\text{nat}, \alpha)) \text{esta?}(t, l))$
 $\Rightarrow_L (\forall n : \text{nat}) n =_{obs} t.\text{clave} \Leftrightarrow$
 $\text{def?}(n, \text{dicc}) \wedge_L \text{obtener}(n, \text{dicc}) =_{obs} t.\text{significado}))$

{Rep(e)}

on los de
la mano
la fue de
bien.

3.3. Representación del iterador de DiccNat

`itDiccNat(α) se representa con itDiccNat donde itDiccNat es
itLista(tupla(nat, α))`

3.4. Algoritmos

`crearDiccionario (in v: vector(tupla(clave : nat, significado : α))) → res: estr`

```

nat : i ← 0                                O(1)
while i < longitud(v) do                   O(1)
    AgregarAtras(res.tabla, vacia())        O(1)
end while                                     O(1)
i ← 0                                         O(1)
while i < longitud(v) do                   O(1)
    nat : k ← v[i].clave mod longitud(v)   O(1)
    AgregarAtras(res.tabla[k], v[i])         O(copy( $\alpha$ ))
    nat : q ← longitud(res.tabla[k])          O(i)
    AgregarAtras(res.listaliterable, puntero(res.tabla[k][q-1]))  O(1)
    i++                                         O(1)
end while                                     O( $n * \text{copy}(\alpha)$ )
i ← 0                                         O(1)
while i < longitud(res.vectorOrdenClaves) do  O(1)
    nat : j ← i
    while j > 0 ∧ res.vectorOrdenClaves[j] <
        res.vectorOrdenClaves[j-1] do          O(1)
            tupla(nat, puntero( $\alpha$ )) : temp ← res.vectorOrdenClaves[j]  O(1)
            res.vectorOrdenClaves[j] ← res.vectorOrdenClaves[j-1]        O(1)
            res.vectorOrdenClaves[j-1] ← temp                          O(1)
            j--                                           O(1)
        end while                                     O(i)
        i++                                         O(1)
    end while                                     O( $\sum_{i=0}^{\text{longitud(res.vectorOrdenClaves)}} i$ )

```

Complejidad : $O(n^2 + n * \text{copy}(\alpha))$

redefinir (in/out d: DiccNat(α), in n: nat, in a: α)

```

nat : k ← n mod longitud(d)                O(1)
nat : i ← 0                                O(1)
while i < longitud(d[k]) do               O(1)
    IF d.tabla[k][i].clave =  $\alpha$  THEN          O(1)
        d.tabla[k][i].significado ← a      O(1)
        i ← longitud(d[k])                O(1)
    ELSE                                     O(1)
        i++                                 O(1)
    FI                                     O(i)
end while                                     O(i)

```

Complejidad : $O(i)$ $O((\log \max \text{lista})^2)$

```

obtener (in n: nat, in d: DiccNat( $\alpha$ ) → res: puntero( $\alpha$ ))
    nat : i ← 0
    nat : k ← n mod longitud(d)
    res ← NULL
    while i < longitud(d[k]) do
        IF d.tabla[k][i].clave =abs n THEN
            res ← *d.tabla[k][i].significado
            i ← longitud(d[k])
        ELSE
            i++
        FI
    end for

```

Complejidad: $O(i)$

definido? (in n: nat, in d: DiccNat(α) → res: bool

```

    nat : k ← n mod longitud(d)
    nat : i ← 0
    res ← false
    while i < longitud(d[k]) do
        IF d.tabla[k][i].clave =abs n THEN
            res ← true
            i ← longitud(d.tabla[k])
        FI
        i++
    end while

```

Complejidad: $O(i)$

cantClaves (in d: DiccNat(α) → res: nat

```
    res ← longitud(d)
```

Complejidad: $O(1)$

OrdenadoPorClave (in d: DiccNat(α) → res: vector(tupla(nat, α))

```
    res ← d.vectorOrdenClaves
```

$O(i)$

Complejidad: $O(1)$

CrearIt (in d: DiccNat(α) → res: itDiccNat(α)

```
    res ← crearIt(d.listable)
```

$O(1)$

Complejidad: $O(1)$

HaySiguiente (in it: itDiccNat(α) → res: bool

```
res ← HaySiguiente(it)
```

Complejidad : $O(1)$

Siguiente (in $it: \text{itDiccNat}(\alpha)$) \rightarrow res: tupla(nat, α)

```
res ← Siguiente(it)
```

Complejidad : $O(1)$

Avanzar (in/out $it: \text{itDiccNat}(\alpha)$)

```
res ← Avanzar(it)
```

Complejidad : $O(1)$

4. Módulo Diccionario String(α)

Se representa mediante un árbol n-ario con invariante de trie. Las claves son strings y permite acceder a un significado en un tiempo en el peor caso igual a la longitud de la palabra (string) más larga y definir un significado en el mismo tiempo más el tiempo de copy(s) ya que los significados se almacenan por copia.

4.1. Interfaz

parametros formales

géneros: α .

funcion: COPIAR(in $s: \alpha$) → $res : \alpha$

Pre ≡ {true}

Post ≡ { $res =_{obs} s$ }

Complejidad: $O(copy(s))$

Descripción: función de copia de α .

se explica con: DICCIONARIO(STRING, α).

géneros: diccString(α), itDiccString(α).

4.1.1. Operaciones básicas de Diccionario String(α)

CREARDICCIONARIO()

Pre ≡ {true}

Post ≡ { $res =_{obs}$ vacío()}

Complejidad: $O(1)$

Descripción: Crea un diccionario vacío.

DEFINIDO?(in $d: diccString(\alpha)$, in $c: string$) → $res : bool$

Pre ≡ {true}

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

Complejidad: $O(|n_m|)$

Descripción: Devuelve true si la clave está definida en el diccionario y false en caso contrario.

DEFINIR(in/out $d: diccString(\alpha)$, in $c: string$, in $s: \alpha$)

Pre ≡ { $d =_{obs} d_0$ }

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

Complejidad: $O(|n_m| + copy(s))$

Descripción: Define la clave c con el significado s .

Aliasing: Almacena una copia de s .

OBTENER(in $d: diccString(\alpha)$, in $c: string$) → $res : \alpha$

Pre ≡ { $\text{def?}(c, d)$ }

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

Complejidad: $O(|n_m|)$

Descripción: Devuelve el significado correspondiente a la clave c .

Aliasing: Devuelve el significado almacenado en el diccionario, por lo que res es modificable si y sólo si d lo es.

ELIMINAR(in/out $d: diccString(\alpha)$, in $c: string$)

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

Post ≡ { $d =_{obs} \text{borrar}(d_0, c)$ }

Complejidad: $O(|n_m|)$

Descripción: Borra la clave c del diccionario y su significado.

4.1.2. Operaciones básicas del iterador

Este iterador permite recorrer el trie sobre el que está implementado el diccionario para obtener cada clave los significados. Las claves de los elementos iterados no pueden modificarse nunca por cuestiones de implementación. El iterador es un iterador de lista, que recorre listalterable por lo que sus operaciones son idénticas a ella.

CREARIT(in $d: \text{dictString}(\alpha) \rightarrow res : \text{itDictString}(\alpha)$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{alias(esPermutación(SecuSuby(res), } d) \wedge \text{vacia?}(Anteriores(res))\}$
Complejidad: $\Theta(1)$

Descripción: crea un iterador bidireccional del diccionario, de forma tal que HayAnterior evalúe a false (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).

Aliasing: El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique d sin utilizar las funciones del iterador.

HAYSIGUIENTE(in $it: \text{itDictString}(\alpha) \rightarrow res : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{obs} \text{haySigiente?}(it)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

HAYANTERIOR(in $it: \text{itDictString}(\alpha) \rightarrow res : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{obs} \text{hayAnterior?}(it)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para retroceder.

SIGUIENTESIGNIFICADO(in $it: \text{itDictString}(\alpha) \rightarrow res : \alpha$
Pre $\equiv \{\text{haySigiente?}(it)\}$
Post $\equiv \{\text{alias}(res =_{obs} \text{haySigiente?}(it).\text{significado})\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el significado del elemento siguiente del iterador
Aliasing: res es modificable si y sólo si it es modificable.

ANTERIORSIGNIFICADO(in $it: \text{itDictString}(\alpha) \rightarrow res : \alpha$
Pre $\equiv \{\text{hayAnterior?}(it)\}$
Post $\equiv \{\text{alias}(res =_{obs} \text{haySigiente?}(it).\text{significado})\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el significado del elemento anterior del iterador
Aliasing: res es modificable si y sólo si it es modificable.

AVANZAR(in/out $it: \text{itDictString}(\alpha)$
Pre $\equiv \{it = it_0 \wedge \text{haySigiente?}(it)\}$
Post $\equiv \{it = it_0 \wedge \text{avanzar}(it_0)\}$
Complejidad: $\Theta(1)$
Descripción: avanza a la posición siguiente del iterador.

RETROCEDER(in/out $it: \text{itDictString}(\alpha)$
Pre $\equiv \{it = it_0 \wedge \text{hayAnterior?}(it)\}$

Post = {it = obs , hayAnterior?(it₀)}

Complejidad: $\Theta(1)$

Descripción: retrocede a la posición anterior del iterador.

4.1.3. Representación de Diccionario String(α)

Diccionario String(α) se representa con estr

donde estr es tupla(raiz: arreglo(puntero(Nodo)), listaIterable: lista(puntero(Nodo)))

donde Nodo es tupla(arbolTrie: arreglo(puntero(Nodo)),
 info: α ,
 infoValida: bool,
 infoEnLista: iterador(listaIterable))

4.1.4. Invariante de Representación

- (I) Raiz es la raiz del arbol con invariante de trie y es un arreglo de 27 posiciones.
- (II) Cada uno de los elementos de la lista tiene que ser un puntero a un Nodo del trie.
- (III) Nodo es una tupla que contiene un arreglo de 27 posiciones con un puntero a otro Nodo en cada posicion ,un elemento info que es el alfa que contiene esa clave del arbol, un elemento infoValida y un elemento iterador que es un puntero a un nodo de la lista enlazada.
- (IV) El iterador a la lista enlazada de cada nodo tiene que apuntar al elemento de la lista que apunta al mismo Nodo.
- (V) Cada uno de los nodos de la lista apunta a un nodo del arbol cuyo infoEnLista apunta al mismo nodo de la lista.

$(\forall c: \text{diccString}((\alpha)))()$

Rep : estr \rightarrow bool

Rep(e) \equiv true \iff

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

nodoValido : puntero(Nodo) nodo \rightarrow bool

iteradorValido : puntero(Nodo) nodo \rightarrow bool

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

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

PunteroValido(nodo) \equiv El iterador perteneciente al nodo (infoEnLista) apunta a un nodo de listaIterable (lista(puntero(Nodo))) cuyo puntero apunta al mismo nodo pasado por parámetro. Es decir se trata de una referencia circular.

listaValida(lista) \equiv Cada nodo de la lista tiene un puntero a un nodo de la estructura cuyo infoEnLista (iterador) apunta al mismo nodo. Es decir se trata de una referencia circular.

4.1.5. Función de Abstracción

```

Abs : estr e → diccString(α) {Rep(e)}
Abs(e) =obs d: diccString(α) | (vs: string)(def?(d, s) =obs
Definido?(d, s) ∧
def?(d, s) ⇒L obtener(s, d) =obs
Obtener(d, s)
)

```

4.2. Algoritmos

crearDiccionario () → res: estr

```

arreglo(puntero(Nodo)): res.raiz ← CrearArreglo(27) O(1)
nat: i ← 0 O(1)
while i < longitud(res.raiz) do O(1)
    res.raiz[i] ← NULL O(1)
end while O(1)
res.listaIterable ← Vacia() O(1)

```

Complejidad: $O(1)$

definido? (in d: diccString(α), in c: string) → res: bool

```

nat: i ← 0 O(1)
nat: letra ← posicion(c[0]) O(1)
puntero(Nodo): arr ← d.raiz[letra] O(1)
while(i < longitud(c) and not arr = NULL) do O(1)
    i ← i + 1 O(1)
    letra ← posicion(c[i]) O(1)
    arr ← (*arr).arbolTrie[letra] O(1)
end while O(|nm|)
if(i=longitud(c)) then O(1)
    res ← (*arr).infoValida O(1)
else
    res ← false O(1)
end if

```

Complejidad: $O(|n_m|)$

definir (in/out d: diccString(α), in c: string, in s: α)

```

nat: i ← 0 O(1)
nat: letra ← posicion(c[0]) O(1)
if (d.raiz[letra] = NULL) then O(1)
    Nodo: nuevo O(1)
    arreglo(puntero(Nodo)): nuevo.arbolTrie ← CrearArreglo(27) O(1)
    nuevo.infoValida ← false O(1)
    d.raiz[letra] ← puntero(nuevo) O(1)
end if O(1)
puntero(Nodo): arr ← d.raiz[letra] O(1)
while(i < longitud(c)) do O(1)
    i ← i + 1 O(1)
    letra ← posicion(c[i]) O(1)
    if (arr.arbolTrie[letra] = NULL) then O(1)

```

```

Nodo: nuevoHijo
arreglo(puntero(Nodo)): nuevoHijo, arbolTrie ← CrearArreglo(27) O(1)
nuevoHijo.infoValida ← false O(1)
arr.arbolTrie[letra] ← puntero(nuevoHijo) O(1)
end if
arr ← (*arr).arreglo[letra] O(1)
O(1)

end while
(*arr).info ← s O(|nm|)
if(not (*arr).infoValida = true) then O(copy(s))
    itLista(puntero(Nodo)) it ← AgregarAdelante(d.listaIterable, NULL) O(1)
    (*arr).infoValida ← true O(1)
    (*arr).clave ← c O(|nm|)
    (*arr).infoEnLisita ← it O(1)
    siguiente(it) ← puntero(*arr) O(1)
end if

```

Complejidad: $O(|n_m| + \text{copy}(s))$

obtener (in d: diccString(α), in c: string) → res: α

```

nat: i ← 0 O(1)
nat: letra ← posicion(c[0]) O(1)
puntero(Nodo): arr ← d.raiz[letra] O(1)
while(i < longitud(c)) do O(1)
    i ← i + 1 O(1)
    letra ← posicion(c[i]) O(1)
    arr ← (*arr).arbolTrie[letra] O(1)
end while O(|nm|)
res ← (*arr).info O(1)


```

Complejidad: $O(|n_m|)$

eliminar (in/out d: diccString(α), in c: string)

```

nat: i ← 0 O(1)
nat: letra ← posicion(c[0]) O(1)
puntero(Nodo): arr ← d.raiz[letra] O(1)
pila(puntero(Nodo)): pil ← Vacia() O(1)
while(i < longitud(c)) do O(1)
    i ← i + 1 O(1)
    letra ← posicion(c[i]) O(1)
    arr ← (*arr).arbolTrie[letra] O(1)
    Apilar(pil, arr) O(1)
end while O(|nm|)
if (tieneHermanos(arr)) then O(1)
    (*arr).infoValida ← false O(1)
else
    i ← 1 O(1)
    puntero(Nodo): del ← tope(pil) O(1)
    del ← NULL O(1)
    Desapilar(pil) O(1)
    while(i < longitud(c) and not(tieneHermanosEInfo(*tope(pil)))) do O(1)
        del ← tope(pil) O(1)
        del ← NULL O(1)


```

```

        Desapilar(pil)
        i ← i + 1
    end while
    if (i = longitud(c)) then
        d.raiz[posicion(c[0])] ← NULL
    end if
end if

```

$O(1)$
 $O(1)$
 $O(n_m)$
 $O(1)$
 $O(1)$

Complejidad : $O(|n_m|)$

tieneHermanos (in nodo: puntero(Nodo)) → res: bool

```

nat: i ← 0
nat: l ← longitud((*nodo).arbolTrie))
while(i < l and not((*nodo).arbolTrie[i] = NULL)) do
    i ← i + 1
end while
res ← i < l

```

$O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$
 $O(1)$

Complejidad : $O(1)$

tieneHermanosEInfo (in nodo: puntero(Nodo)) → res: bool

```
res ← tieneHermanos(nodo) and ((*nodo).infoValida = true)
```

$O(1)$

Complejidad : $O(1)$

posicion (in ch: char) → res: nat

```

if(ch=a) then
    res ← 0
else if (ch=b) then
    res ← 1
else if(ch=c) then
    res ← 2
else if(ch=d) then
    res ← 3
else if(ch=e) then
    res ← 4
else if(ch=f) then
    res ← 5
else if(ch=g) then
    res ← 6
else if(ch=h) then
    res ← 7
else if(ch=i) then
    res ← 8
else if(ch=j) then
    res ← 9
else if(ch=k) then
    res ← 10
else if(ch=l) then
    res ← 11

```

$O(1)$
 $O(1)$

```
else if (ch=m) then O(1)
    res ← 12
else if(ch=n) then O(1)
    res ← 13
else if(ch='n) then O(1)
    res ← 14
else if(ch=o) then O(1)
    res ← 15
else if(ch=p) then O(1)
    res ← 16
else if(ch=q) then O(1)
    res ← 17
else if(ch=r) then O(1)
    res ← 18
else if(ch=s) then O(1)
    res ← 19
else if(ch=t) then O(1)
    res ← 20
else if(ch=u) then O(1)
    res ← 21
else if(ch=v) then O(1)
    res ← 22
else if(ch=w) then O(1)
    res ← 23
else if(ch=x) then O(1)
    res ← 24
else if(ch=y) then O(1)
    res ← 25
else
    res ← 26
end if
```

Complejidad : $O(1)$