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 Goldsmit	247/14	kevinfra94@gmail.com
Nicolás Bukovits	546/14	nicobuk@gmail.com

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

Índice

1. Módulo CampusSeguro

1.1. Interfaz

se explica con: CAMPUSSEGURO.

géneros: `campusSeguro`.

1.1.1. Operaciones básicas de CampusSeguro

COMENZARRASTRILLAJE(**in** c: `campus`, **in** d: `diccNat(agente,datosAgente)`) $\rightarrow$ res : `campusSeguro`
Pre $\equiv$ {($\forall a$: agente) (def?(a,d) $\Rightarrow_L$ (posVálida(obtener(a,d)) $\wedge$ $\neg$ocupada?(obtener(a,d,c))) $\wedge$ ($\forall a, a2$: agente) ((def?(a,d) $\wedge$ def?($a2$,d) $\wedge a \neq a2$) $\Rightarrow_L$ obtener(a,d) $\neq$ obtener($a2$,d))}
Post $\equiv$ {res $=_{obs}$ comenzarRastrillaje(c,d)}
Complejidad: $O(f * c)^2 + N_a^2)$
Descripción: Crea un nuevo campusSeguro tomando un campus y un diccionario con agentes.
Aliasing: Se genera aliasing campus y en el diccionario que se pasan por parametro.

INGRESARESTUDIANTE(**in** e: `nombre`, **in** p: `posición`, **in/out** cs: `campusSeguro`)
Pre $\equiv$ {$cs =_{obs} cs_o \wedge e \notin$ (estudiantes(cs) $\bigcup$ hippies(cs)) $\wedge$ esIngreso?(p,campus(cs)) $\wedge$ $\neg$estaOcupada?(p,cs)}
Post $\equiv$ {$cs =_{obs}$ ingresarEstudiante(e,p,cs_o)}
Complejidad: $O(|n_m|)$
Descripción: Ingresa un nuevo estudiante al campusSeguro

INGRESARHIPPIE(**in** h: `nombre`, **in** p: `posición`, **in/out** cs: `campusSeguro`)
Pre $\equiv$ {$cs =_{obs} cs_o \wedge h \notin$ (estudiantes(cs) $\bigcup$ hippies(cs)) $\wedge$ esIngreso?(p,campus(cs)) $\wedge$ $\neg$estaOcupada?(p,cs)}
Post $\equiv$ {$cs =_{obs}$ ingresarHippie(h,p,cs_o)}
Complejidad: $O(|n_m|)$
Descripción: Ingresa un nuevo hippie el campusSeguro

MOVERESTUDIANTE(**in** e: `nombre`, **in** dir: `dirección`, **in/out** cs: `campusSeguro`)
Pre $\equiv$ {$cs =_{obs} cs_o \wedge e \in$ estudiantes(cs) $\wedge$ (seRetira(e,dir,cs)) $\vee$ (posValida(proxPosicion(posEstudianteYHippie(e,cs),dir,campus(cs)), campus(cs)) $\wedge$ $\neg$estaOcupada?(proxPosicion(posEstudianteYHippie(e,cs),dir,campus(cs)),cs))}
Post $\equiv$ {$cs =_{obs}$ 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).

MOVERHIPPIE(**in** h: `nombre`, **in/out** cs: `campusSeguro`)
Pre $\equiv$ {$cs =_{obs} cs_o \wedge h \in$ hippies(cs) $\wedge$ $\neg$todasOcupadas?(vecinos(posEstudianteYHippie(h,cs),campus(cs)), cs)}
Post $\equiv$ {$cs =_{obs}$ 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).

MOVERAGENTE(**in** a: `agente`, **in/out** cs: `campusSeguro`)
Pre $\equiv$ {$cs =_{obs} cs_o \wedge a \in$ agentes(cs) $\wedge_L$ cantSanciones(a,cs) $\leq 3 \wedge$ $\neg$todasOcupadas?(vecinos(posEstudianteYHippie(h,cs),campus(cs)),cs)}
Post $\equiv$ {$cs =_{obs}$ 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 : `campusSeguro`) $\rightarrow$ res : `campus`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ campus(cs)}
Complejidad: $O(1)$
Descripción: Devuelve el campus del campusSeguro.
Aliasing: res es una referencia no modificable.

ESTUDIANTES(**in** cs : `campusSeguro`) $\rightarrow$ res : `itConj(nombre)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ estudiantes(cs)}
Complejidad: $O(1)$
Descripción: Devuelve el conjunto de los estudiantes que estan en el campus.
Aliasing: res es un iterador no modificable.

HIPPIES(**in** cs : `campusSeguro`) $\rightarrow$ res : `itConj(nombre)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ hippies(cs)}
Complejidad: $O(1)$
Descripción: Devuelve el conjunto de los hippies que estan en el campus.
Aliasing: res es un iterador no modificable.

AGENTES(**in** cs : `campusSeguro`) $\rightarrow$ res : `itConj(agente)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ agentes(cs)}
Complejidad: $O(1)$
Descripción: Devuelve un iterador al diccionario de los agentes que estan en el campus.
Aliasing: res es un interador no modificable.

POSESTUDIANTEYHIPPIE(**in** id : `nombre`, **in** cs : `campusSeguro`) $\rightarrow$ res : `posición`
Pre $\equiv$ {$id \in$ (estudiantes(cs) $\bigcup$ hippies(cs)}
Post $\equiv$ {$res =_{obs}$ posEstudianteYHippie(id,cs)}
Complejidad: $O(|n_m|)$, donde $|m_n|$ es la longitud mas larga entre todos los nombres.
Descripción: Devuelve la posicion del estudiante o hippie.

POSAGENTE(**in** a : `agente`, **in** cs : `campusSeguro`) $\rightarrow$ res : `posición`
Pre $\equiv$ {$a \in$ agentes(cs)}
Post $\equiv$ {$res =_{obs}$ posAgente(a,cs)}
Complejidad: $O(1)$ en caso promedio.
Descripción: Devuelve la posicion del agente pasado como parametro.

CANTSANCIONES(**in** a : `agente`, **in** cs : `campusSeguro`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$a \in$ agentes(cs)}
Post $\equiv$ {$res =_{obs}$ cantSanciones(a,cs)}
Complejidad: $O(1)$ en caso promedio.
Descripción: Devuelve la cantidad de sanciones que posee el agente pasado como parametro.

CANTHIPPIESATRAPADOS(**in** a : `agente`, **in** cs : `campusSeguro`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$a \in$ agentes(cs)}
Post $\equiv$ {$res =_{obs}$ cantHippiesAtrapados(a,cs)}
Complejidad: $O(1)$ en caso promedio.
Descripción: Devuelve la cantidad de hippies que atrapo el agente pasado como parametro.

MÁSVIGILANTE(**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.

CONMISMASSANCIONES(**in** a : `agente`, **in** cs : `campusSeguro`) $\rightarrow$ res : `conj(agentes)`
Pre $\equiv$ {$a \in$ agentes(cs)}
Post $\equiv$ {$res =_{obs}$ conMismasSanciones(a,cs)}
Complejidad: $O(1)$ en el caso promedio.
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.

CONKSANCIONES(**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 conjuto de agenes que tienen k sanciones.
Aliasing: res es una referencia no modificable.

1.1.2. Representación de campusSeguro

`campusSeguro` **se representa con** `estr`

donde `estr` es `tupla`(*campus*: `campus`,
personalAS: `diccNat(agente,datosAgente)`,
agentesOrdenados: `vector(As)` ,
posicionesAgente: `vector(As)`,
masVigilante: `As`,
listaMismasSanc: `lista(kSanc)`,
arregloMismasSanc: `arreglo(itLista(kSanc))` ,
mismasSancModificado: `bool` ,
hippies: `dicString(nombre,posicion)`,
estudiantes: `dicString(nombre, posicion)`,
posicionesHippies: `vector(nombre)` ,
posicionesEstudiantes: `vector(nombre)`)

donde `datosAgente` es `tupla`(*posicion*: `posicion`,
cantSanc: `nat`,
cantAtrapados: `nat`,
itMismasSanc: `itLista(kSanc)`,
itConjMismasSanc: `itConj(agente)`)

donde `As` es `tupla`(*agente*: `agente`, *datos*: `itDiccNat(agente,datosAgente)`)

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

1.1.3. Invariante de Representación

(I) Las posiciones de todos los agentes son posiciones validas del campus.

(II) Las posiciones de los hippies son posiciones validas del campus.

(III) Las posiciones de los estudiantes son posiciones validas 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 esta definido en personalAs y tiene la mayor cantidad de sanciones.

(VIII) Para todo nombre que esta 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 el campus.

(X) Los agentes de posicionesAgente son los mismos que los de personalAs y viceversa.

(XI) La posicion de todos los agentes de posicionesAgente se mapea con la posicion que tienen en personalAs y viceversa.

(XII) Las posiciones en posicionesAgente que no tienen agente tienen un agente tienen un iterador en el que ver si hay siguiente da false.

(XIII) La dimension del campus es mayor que la cantidad de estudiantes, hippies, agentes y obstaculos.

(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) Cada agente que esta en personalAS esta en el conjunto de un nodo que tiene su misma cantidad de sanciones de la lista listaMismasSanciones y viceversa.

(XIX) Cada agente de personalAS tiene un iterador valido, itMismasSanc, que apunta al nodo de la lista listaMismas-Sanc que tiene la misma cantidad de sanciones que ese agente, y otro iterador valido, itConjMismasSanc, que apunta si mismo en el conjunto agentes del nodo de listaMismasSanc que posee su misma cantidad de sanciones.

(XX) Si mismasSancModificado es falso, arregloMismasSanc tiene misma cantidad de elementos que listaMismasSanc, y cada elemento apunta a un nodo de listaMismasSanc, respetando el orden de listaMismasSanc.

(XXI) Los hippies de posicionesHippies son los mismos que los del dicString hippies.

(XXII) Las posiciones en posicionesHippies que no tienen un hippie poseen el caracter de espacio .

(XXIII) La posiciones de todos los hippies en posicionesHippies se mapea con la posicion que tienen en el diccString hippies y viceversa

(XXIV) Los estudiantes de posicionesEstudiantes son los mismos que los del dicString estudiantes.

(XXV) Las posiciones en posicionesEstudiantes que no tienen un estudiante poseen el caracter de espacio .

(XXVI) La posiciones de todos los estudiantes en posicionesEstudiantes se mapea con la posicion que tienen en el diccString estudiantes y viceversa

(XXVII) agentesOrdenados tiene todas las claves de la tabla, el largo es igual a la cantidad de claves de personalAS y los iteradores van al significado de la clave correspondiente en la tabla

(XXVIII) agentesOrdenados está ordenado por placa

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ ($\forall a$: nat)(def?(a, e.personalAS) $\Rightarrow_L$ posValida(Obtener(a,e.personalAS).posicion, e.campus) $\wedge$ $\neg$ocupada?(Obtener(a,e.personalAS).posicion),e.campus)
$\wedge$
($\forall h$: string)(def?(h, e.hippies) $\Rightarrow_L$ posValida(Obtener(h,e.hippies), e.campus) $\wedge$ $\neg$ocupada?(Obtener(h, e.hippies)),e.campus)
$\wedge$
($\forall est$: string)(def?(est, e.estudiantes) $\Rightarrow_L$ posValida(Obtener(est, e.estudiantes), e.campus) $\wedge$ $\neg$ocupada?(Obtener(est, e.estudiantes)), e.campus)
$\wedge$
($\forall a$: nat)($\forall h$: string) def?(a, e.personalAS) $\wedge$ def?(h, e.hippies) $\Rightarrow_L$ Obtener(a, e.personalAS).posicion $\neq$ Obtener(h, e.hippies)
$\wedge$
($\forall est$: string)($\forall h$: string) def?(est,e.estudiantes) $\wedge$ def?(h,e.hippies) $\Rightarrow_L$ Obtener(est,e.estudiantes) $\neq$ Obtener(h, e.hippies)
$\wedge$
($\forall a$: nat)($\forall est$: string) def?(a, e.personalAS) $\wedge$ def?(est, e.estudiantes) $\Rightarrow_L$ Obtener(a, e.personalAS).posicion $\neq$ Obtener(est, e.estudiantes)
$\wedge$
def?(masVigilante.agente, e.personalAS) $\wedge_L$ HaySiguiente(masVigilante.datos) $\wedge_L$ Obtener(masVigilante.agente, e.personalAS) = Siguiente(masVigilante.datos)
$\wedge$
($\forall a$: nat)(def?(a, e.personalAS) $\wedge$ HaySiguiente(masVigilante.datos)) $\Rightarrow_L$ Obtener(a, e.personalAS).cantSanc $\leq$ Siguiente(masVigilante.datos).cantSanc
$\wedge$
($\forall h$: string) def?(h, e.hippies) $\Rightarrow$ $\neg$def?(h, e.estudiantes)
$\wedge$
($\forall est$: string) def?(est, e.estudiantes) $\Rightarrow$ $\neg$def?(est, e.hippies)
$\wedge$
longitud(e.posicionesAgente) = filas(e.campus)$*$columnas(e.campus)
$\wedge$
longitud(e.posicionesHippies) = filas(e.campus)$*$columnas(e.campus)
$\wedge$
longitud(e.posicionesEstudiantes) = filas(e.campus)$*$columnas(e.campus)
$\wedge$
($\forall a$: nat) def?(a, e.personalAS) $\Leftrightarrow_L$ ($\exists i$: nat) i < longitud(e.posicionesAgente) $\wedge_L$ e.posicionesAgente[i].agente = a $\wedge$ Siguiente(e.posicionesAgente[i].datos) = Obtener(a, e.personalAS) $\wedge$ (i / columnas(e.campus)) = Obtener(a, e.personalAS).posicion.y $\wedge$ (i *mod* columnas(e.campus)) = Obtener(a, e.personalAS).posicion.x
$\wedge$
($\forall i$: nat) i < longitud(e.posicionesAgente) $\Leftrightarrow_L$ $\neg$HaySiguiente(e.posicionesAgente[i].datos) $\vee$ ($\exists!a$: nat) def?(a, e.personalAS) $\wedge_L$ Siguiente(e.posicionesAgente[i].datos) = Obtener(a, e.personalAS)
$\wedge$
#claves(e.personalAS) + #claves(e.hippies) + #claves(e.estudiantes) + #Obstaculos(e.campus) $\leq$ filas(e.campus)*columnas(e.campus)
$\wedge$
long(e.listaMismasSanc) = #SancionesDistintas(e.personalAS)
$\wedge$
($\forall$ sanc: nat)(sanc $\in$ conjSanciones(claves(e.personalAS), e.personalAS) $\Rightarrow$ ($\exists$ nodo: kSanc)(esta?(nodo, e.listaMismasSanc) $\wedge$ nodo.sanc = sanc))
$\wedge$
ordenada?(e.listaMismasSanc)

$\wedge$
($\forall$ nodo: kSanc)(esta?(nodo, e.listaMismasSanc) $\Rightarrow_L$ agentes(nodo) = agentesKSanc(e.personalAS, claves(e.personalAS), sanc(kSanc)))
$\wedge$
($\forall$ a: nat)(def?a, e.personalAS) $\Rightarrow_L$ (HaySiguiente(Obtener(a, e.personalAS).itMismasSanc) $\wedge_L$ Siguiente(Obtener(a,

e.personalAS).itMismasSanc).sanc = Obtener(a, e.personalAS).cantSanc) $\wedge$ (HaySiguiente(Obtener(a, e.personalAS).itConjMismasSanc) $\wedge_L$ Siguiente(Obtener(a, e.personalAS).itConjMismasSanc) = a) $\wedge$ ($\exists!$ nodo: kSanc) (esta?(nodo, e.listaMismasSanc) $\wedge_L$ Siguiente(Obtener(a, e.personalAS).itMismasSanc = nodo $\wedge$ Pertenece?(nodo, Siguiente(Obtener(a, e.personalAS).itConjMismasSanc)) = Pertenece?(nodo, a)
$\wedge$
$\neg e$.mismasSancModificado $\Rightarrow$ (long(e.listaMismasSanc) = long(vectorMismasSanc) $\wedge_L$ mismosNodos(e.listaMismasSanc, e.vectorMismasSanc))
$\wedge$
($\forall$ i: nat)(i <longitud(e.posicionesHippies) $\Leftrightarrow_L$ posicionesHippies[i] = $\vee$ ($\exists!$ h: string)(def?(h, e.hippies) $\wedge_L$ h = posicionesHippies[i]
$\wedge$
($\forall$ h: string)(def?(h, e.hippies) $\Leftrightarrow_L$ ($\exists!$ i: nat)(i <longitud(e.posicionesHippies) $\wedge_L$ e.posicionesHippies[i] = obtener(h, e.hippies) $\wedge$ (i / columnas(e.campus) = obtener(h, e.hippies).x) $\wedge$ (i mod columnas(e.campus) = obtener(h, e.hippies).y)))
$\wedge$
($\forall$ i: nat)(i <longitud(e.posicionesEstudiantes) $\Leftrightarrow_L$ posicionesEstudiantes[i] = $\vee$ ($\exists!$ e: string)(def?(e, e.estudiantes) $\wedge_L$ e = posicionesEstudiantes[i]
$\wedge$
($\forall$ est: string)(def?(est, e.estudiantes) $\Leftrightarrow_L$ ($\exists!$ i: nat)(i <longitud(e.posicionesEstudiantes) $\wedge_L$ e.posicionesEstudiantes[i] = Obtener(est, e.estudiantes) $\wedge$ (i / columnas(e.campus) = Obtener(est, e.estudiantes).x) $\wedge$ (i mod columnas(e.campus) = Obtener(est, e.estudiantes).y)))
$\wedge$
long(e.agentesOrdenados) = cantClaves(e.personalAS) $\wedge$ ($\forall$ placa : agente)(definido?(placa, e.personalAS)
$\Rightarrow_L$ ($\exists$ a : As)(placa $=_{obs}$ a.agente $\wedge$ obtener(placa, e.personalAS) $=_{obs}$ a.datos)
$\wedge$
ordenadoPorPlaca?(e.agentesOrdenados)

#Obstaculos : campus $\longrightarrow$ nat

#ObstaculosAux : nat $\times$ nat $\times$ campus $\longrightarrow$ nat

#SancionesDistintas : diccNat(agente $\times$ datosAgente) $\longrightarrow$ nat

conjSanciones : conj(nat) $\times$ diccNat(agente $\times$ datosAgente) $\longrightarrow$ conj(nat)

cantAgentes : secu(kSanc) $\longrightarrow$ nat

ordenada? : secu(kSanc) $\longrightarrow$ bool

agentesKSanc : diccNat(agente $\times$ datosAgente) $\times$ conj(agente) $\times$ nat $\longrightarrow$ conj(agente)

mismosNodos : secu(kSanc) *lista* $\times$ secu(itLista(kSanc)) *vec* $\longrightarrow$ bool {long(lista) = long(vec)}

ordenadoPorPlaca? : secu(As) $\longrightarrow$ bool

#Obstaculos(c) $\equiv$ #ObstaculosAux(filas(c)-1, columnas(c)-1, c)

#ObstaculosAux(f,col,c) $\equiv$ **if** ($f = 0 \wedge col = 0$) **then**
β(ocupada?(<f,col>, c))
else
if ($f \neq 0 \wedge col = 0$) **then**
#ObstaculosAux($f - 1$, columnas(c)-1) + β(ocupada?(<f,col>, c))
else
if ($f = 0 \wedge col \neq 0$) **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
fi

#SancionesDistintas(d) $\equiv$ #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?(fin(lista)) **then**
true
else
if sanc(prim(lista)) <sanc(prim(fin(lista))) **then** ordenada?(fin(lista)) **else** false **fi**
fi
fi

agentesKSanc(dicc, claves, k) ≡ **if** ∅?(claves) **then**
∅
else
if cantSanc(obtener(prim(claves), dicc)) = k **then**
Ag(prim(claves), agentesKSanc(dicc, fin(claves), k))
else
agentesKSanc(dicc, fin(claves), k)
fi
fi

mismosNodos(lista, vec) ≡ **if** vacia?(lista) **then**
true
else
if prim(lista) = Siguiente(prim(vec)) **then**
mismosNodos(fin(lista), fin(vec))
else
false
fi
fi

ordenadoPorPlaca?(vec) ≡ **if** vacia?(vec) **then**
true
else
if vacia?(fin(vec)) **then**
true
else
if agente(prim(vec)) <agente(prim(fin(vec))) **then**
ordenadoPorPlaca?(fin(vec))
else
false
fi
fi
fi

1.1.4. Función de Abstracción

Abs : estr e ⟶ campusSeguro {Rep(e)}

Abs(e) =$_{obs}$ cs: campusSeguro | e.campus = campus(cs)
∧
e.estudiantes = estudiantes(cs)
∧
e.hippies = hippies(cs)
∧
e.personalAS = agentes(cs)
∧

($\forall a$: nat) (def?(a, e.personalAS) $\Rightarrow_L$ posAgente(a, cs) = Obtener(a, e.personalAS).posicion $\wedge$ cantSanciones(a, cs) = Obtener(a, e.personalAS).cantSanc $\wedge$cantHippiesAtrapados(a, cs) = Obtener(a, e.personalAS).cantAtrapados)
$\wedge$
($\forall id$: string) (def?(id, e.estudiantes) $\wedge_L$ Obtener(id, e.estudiantes) = posEstudianteYHippie(cs)) $\vee$ (def?(id, e.hippies) $\wedge_L$ Obtener(id, e.hippies) = posEstudianteYHippie(cs))

1.2. Algoritmos

```
icomenzarRastrillaje (in c: campus, in d: diccNat(agente, datosAgente)) → res: estr

    res.campus ← c (genera aliasing)                                          O(1)
    res.listaMismasSanc ← generarListaMismasSanc(d) (c1)                      O(N_a)
    res.personalAS ← d (genera aliasing)                                      O(1)
    res.posicionesAgente ← vectorizarPos(d,filas(c),columnas(c)) (c2)         O((f*c)^2 + N_a)
    res.masVigilante ← menorPlaca(d) (c3)                                     O(N_a)
    res.mismasSancModificado ← true                                           O(1)
    res.hippies ← Vacio()                                                     O(1)
    res.estudiantes ← Vacio()                                                 O(1)
    res.posicionesHippies ← Vacio()                                           O(1)
    res.posicionesEstudiantes ← Vacio()                                       O(1)

    nat: i ← 0                                                                O(1)
    while i < filas(c)*columnas(c) do                                         O(1)
        AgregarAtras(res.posicionesHippies, " ")  O(k, siendo k la cantidad de elementos del vector)
        AgregarAtras(res.posicionesEstudiantes, " ")                  O(k, idem anterior)
        i ← i+1                                                               O(1)
    end while (c4)                                                            O(∑_{k=0}^{f*c-1}(2k))

    itDiccNat(agente, datosAgente) it ← CrearIt(d)                            O(1)
    res.ordenadoPorPlaca ← Vacia()                                            O(1)
    while HaySiguiente?(it) do                                                O(1)
        i ← 0                                                                 O(1)
        bool ordenado ← true                                                  O(1)
        while i < Longitud(res.ordenadoPorPlaca) ∧ ordenado do                O(1)
            if res.ordenadoPorPlaca[i].agente > Siguiente(it).agente then     O(1)
                ordenado ← false                                              O(1)
            end if
        end while                                                             O(N_a)
        Agregar(res.ordenadoPorPlaca, i, tupla(Siguiente(it).agente, it))     O(N_a)
    end while (c5)                                                            O(N_a^2)
```

Complejidad : $O(N_a)+O(N_a)+O((f*c)^2+N_a)+O(\sum_{k=0}^{f*c-1}(2k))+O(N_a^2) = max\{2O(N_a), O((f*c)^2+N_a)\}+O(2*(f*c-1)*(f*c)/2)+O(N_a^2) = O((f*c)^2+N_a)+O(2*(f*c-1)*(f*c)/2)+O(N_a^2) < O((f*c)^2+N_a)+O((f*c)^2)+O(N_a^2) = O((f*c)^2+N_a^2)$

Justificación Complejidad: ***(c1)*** Complejidad de generarListaMismasSanc es $O(N_a)$.
(c2) Complejidad de vectorizarPos es $O(f*c)^2 + N_a)$.
(c3) Complejidad de menorPlaca es $O(N_a)$.
(c4) AgregarAtras del vector cuesta $\Theta(f(\text{long}(v) + \text{copy(elemento)})$ entonces, por cada iteración hay un elemento más en el vector, esto genera que sea $\sum_{k=0}^{n}(2k)$lo que tarda en agregarse un elemento, siendo $0 \leq n \leq f*c-1$. Por lo tanto, por todas las iteraciones tarda $O(\sum_{k=0}^{f*c-1}(2k)) = O(2*(f*c-1)*(f*c)/2) < O((f*c)^2)$.
(c5) Recorro todas las claves del diccionario, lo cual cuesta $O(N_a)$, y para cada una recorro el vector, que es lineal en la cantidad de elementos del vector, la cual va de 0 hasta la cantidad de agentes, y tambien se agrega en el vector hasta la posicion recorrida, esto cuesta $O(N_a)$. Finalmente, la complejidad es $O(N_a^2)$.

VECTORIZARPOS(**in** d: `diccNat(agente, datosAgente)`, **in** f: `nat`, **in** c: `nat`) → res : `vector(As)`
Pre ≡ {true}
Post ≡ {res va a ser el vector donde cada posicion va a matchear con una posicion del campus. Además, en cada posicion, va a haber un agente (si lo hay en el campus).}

```
vectorizarPos (in d: diccNat(agente, datosAgente), in f: nat, in c: nat) → res: vector(As)
    res ← Vacio()                                                             O(1)
```

```
nat: i ← 0                                                                    O(1)
itDiccNat(agente, datosAgente): it ← CrearIt(d)                               O(1)

vector(tupla(agente, datosAgente)) vVacio ← Vacio()                           O(1)
diccNat(agente, datosAgente) dVacio ← crearDiccionario(vVacio)                O(1)
itDiccNat(agente, datosAgente): itVacio ← CrearIt(dVacio)                     O(1)

while i < f*c do                                                              O(1)
    AgregarAtras(res, tupla(0,itVacio))    O(k, siendo k la cantidad de elementos en el vector.)
    i ← i+1                                                                   O(1)
end while (c1)                                                                O(∑_{k=0}^{f*c-1}(k)))

while HaySiguiente(it) do                                                     O(1)
    itDiccNat(agente, datosAgente): itPos ← it por copia                      O(1)
    res[Siguiente(it).significado.posicion.y * c +
            Siguiente(it).significado.posicion.x] ←
        ← tupla(Siguiente(it).clave, itPos)                                   O(1)
end while (c2)                                                                O(N_a)
```

Complejidad : $O(\sum_{k=0}^{f*c-1}(k)) + N_a) = O((f*c-1)*(f*c)/2) + N_a) < O((f*c)^2 + N_a)$

Justificación Complejidad: ***(c1)*** AgregarAtras del vector cuesta $\Theta(f(\text{long}(v) + \text{copy}(\text{elemento}))$ entonces, por cada iteración hay un elemento más en el vector, esto genera que sea $\sum_{k=0}^{n}(k)$lo que tarda en agregarse un elemento, siendo $0 \leq n \leq f*c-1$. Por lo tanto, por todas las iteraciones tarda $O(\sum_{k=0}^{f*c-1}(k)) = O((f*c-1)*(f*c)/2) \leq O((f*c)^2)$. ***(c2)*** En cada iteración del iterador del diccionario de agentes me crea una copia del iterador (que tarda $O(1)$) y va agregando en el vector de posiciones en la posicion que corresponde al agente que apunta el iterador, con la copia del iterador en esa posición. Esto lo hace por todos los agentes por eso la complejidad de este ciclo es $O(N_a)$.
Como estos dos ciclos son los que predominan en complejidad (el resto de las funciones es $O(1)$) entonces la complejidad queda como $O((f*c)^2 + N_a)$.

menorPlaca(**in** d: `diccNat(agente,datosAgente)`) → res : `As`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ agenteDeMenorPlaca(d)}

```
menorPlaca (in d: diccNat(agente, datosAgente)) → res: As
    itDiccNat(agente, datosAgente): it ← CrearIt(d)                           O(1)
    nat: placaMenor ← Siguiente(it).clave                                     O(1)
    while HaySiguiente(d) do                                                  O(1)
        if Siguiente(it).clave < placaMenor then                              O(1)
            placaMenor ← Siguiente(it).clave                                  O(1)
            itDiccNat(agente, datosAgente) itMenor ← it por copia             O(1)
        end if                                                                O(1)
        Avanzar(it)                                                           O(1)
    end while (c1)                                                            O(N_a)

    res.agente ← placaMenor                                                   O(1)
    res.datos ← itMenor                                                       O(1)
```

Complejidad : $O(N_a)$

Justificación Complejidad: ***(c1)*** Va iterando sobre todos los agentes y buscando el que tiene menor placa, cuando lo encuentra crea una copia del iterador que apunta a ese agente y de la placa. Como itera sobre todos los agentes para buscarlo toma $O(N_a)$.

generarListaMismasSanc(**in/out** d: `diccNat(agente, datosAgente)`) → res : `lista(kSanc)`

Pre ≡ {true}
Post ≡ {res $=_{obs}$ generarListaMismasSanc ∧ (∀ a: nat)(def?(a,d) $\Rightarrow_L$ HaySiguiente(Obtener(a,d).itMismasSanc) ∧ HaySiguiente(Obtener(a,d).itConjMismasSanc) $\wedge_L$ Siguiente(Obtener(a,d).itMismasSanc).sanc = Obtener(a,d).cantSanc ∧ Pertenece(a,Siguiente(Obtener(a,d).itMismasSanc).agentes) ∧ Siguiente(Obtener(a,d).itConjMismasSanc) = a}

```
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 (c1)                                                              O(N_a)
```

Complejidad : $O(N_a)$

Justificación Complejidad: ***(c1)*** Por cada agente, va agregandolo al conjunto de agentes que tiene el nodo de la lista y va agregando los iteradores correspondientes al agente. Como es por cada agente esto toma $O(N_a)$.

```
iIngresarEstudiante (in e: nombre, in pos: posicion, in/out cs: estr )
    if todasOcupadas?(vecinos(pos, cs.campus), cs) AND
    AND AlMenosUnAgente(vecinos(pos, cs.campus)) then (c1)                      O(2*4)
        conj(As): conjAgParaSanc ←
            ← AgParaPremSanc(vecinos(pos, cs.campus), cs) (c2)                  O(4)
        SancionarAgentes(conjAgParaSanc, cs) (c3)                               O(4)
    end if                                                                      O(3*4)

    if CantHippiesVecinos(vecinos(pos, cs.campus), cs)<2 (c4) then              O(4*|n_m|)
        Definir(cs.estudiantes, e, pos)                                         O(|n_m|)
        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)
    end if                                                                      O(5*|n_m|)


    conj(nombre, posicion): conjHippiesRodEst ←
        ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs) (c5)          O(4)

    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 (f1)                                                               O(4*2|n_m|)
end if                                                                           O(4*2|n_m|)


//Las capturas se actualizan en HippiesRodeadosAs
conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) (c6)                        O(4)

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] ← " "                                                  O(1)
        Avanzar(itHAs)                                                           O(1)
    end while (f2)                                                               O(4*|n_m|)
end if                                                                           O(4*|n_m|)


conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) (c7)               O(4)

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)
    end while (f3)                                                               O(4*2|n_m|)
end if                                                                           O(4*2|n_m|)


conj(posicion): conjEstRodAs ←
    ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) (c8)                    O(4)

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 (c9)     O(2*4)
            conj(As): conjAgParaSanc ←
                ← AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus),
                    cs) (c10)                                                    O(4)
```

```
            SancionarAgentes(conjAgParaSanc, cs) (c11)                    O(4)
        end if                                                            O(1)
    end while (f4)                                                        O(4*4*4)

end if                                                                    O(4*4*4)
```

Complejidad : $O(3*4)+O(5*|n_m|)+O(4)+O(4*2|n_m|)+O(4*|n_m|)+O(4*2|n_m|)+O(4*4*4) = O(80)+O(25|n_m|) = O(1) + O(|n_m|) = O(|n_m|)$

Complejidad: ***(c1)*** Las funciones todasOcupadas? y AlMenosUnAgente tienen como complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro. Como el conjunto que les estoy pasando solo tiene cuatro elementos (porque la función vecinos devuelve un conjunto de cuatro posiciones adyacentes) entonces la complejidad final de esa guarda es $O(2*4)$ que es lo mismo que $O(1)$.
(c2) AgParaPremSanc idem que el anterior punto, la complejidad es $O(\#c)$ pero le estoy pasando un conjunto de 4 elementos la complejidad es $O(4)$ que es lo mismo que $O(1)$.
(c3) SancionarAgentes idem que el anterior, su complejidad es $O(\#c)$ pero como le paso un conjunto que como maximo tiene 4 elementos la complejidad es $O(4)$, o sea $O(1)$.
(c4) CantHippiesVecinos tiene como complejidad $O(\#c + |n_m|)$ siendo c el conjunto que se le pasa como parametro pero como le estoy pasando un conjunto que maximo tiene 4 elementos la complejidad termina siendo $O(4*|n_m|)$ que es lo mismo que $O(|n_m|)$.
(c5) HippiesRodeadosEstudiantes tiene como complejidad $O(\#c)$ siendo c el conjunto que se le pasa como parametro pero como le estoy pasando un conjunto que maximo tiene 4 elementos la complejidad termina siendo $O(4)$ que es lo mismo que $O(1)$.
(c6) HippiesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c7) EstudiantesRodeadosHippies idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c8) EstudiantesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c9) idem que el primer punto, todasOcupadas? y AlMenosUnAgente tienen complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro, pero como este conjunto tiene como maximo 4 elementos (porque vecinos devuelve un conjunto con maximo 4 elementos), las complejidades de ambas funciones es $O(4)$ que es lo mismo que $O(1)$.
(c10) AgParaPremSanc idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c11) SancionarAgentes idem que el punto anterior, termina teniendo complejidad $O(1)$.
(f1) El ciclo itera sobre el conjunto conjHippiesRodEst que tiene como máximo 4 elementos y dentro de él se define un estudiante ($O(|n_m|)$) y se elimina un hippie ($O(|n_m|)$), por lo tanto la complejidad del mismo es $O(4*2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f2) Idem que el punto anterior, el conjunto conjHippiesRodAs tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4*|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f3) Idem que el punto anterior, el conjunto conjEstRodHip tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4*2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f4) Idem que el anterior pero como las operaciones están acotadas a cuatro elementos la complejidad queda como $O(4*4*4)$ que es lo mismo que $O(1)$.

EstudiantesRodeadosAs(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `conj(posición)`
Pre ≡ {true}
Post ≡ {res es el conjunto de posiciones en el cual hay estudiantes rodeados por As.}

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

    while HaySiguiente(itC) do                                            O(1)
        if TodasOcupadas?(vecinos(Siguiente(itC), cs), cs) AND
        AND AlMenosUnAgente(vecinos(Siguiente(itC), cs), cs) then (c1)    O(2*4)
        AgregarRapido(res, Siguiente(itC))                                O(1)
        end if                                                            O(1)
        Avanzar(itC)                                                      O(1)
    end while                                                             O((2*4)*#c)
```

Complejidad : $O((2*4)*\#c) = O(\#c)$

Complejidad: ***(c1)*** Las funciones todasOcupadas? y AlMenosUnAgente tienen como complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro. Como el conjunto que les estoy pasando solo tiene cuatro elementos (porque la función vecinos devuelve un conjunto de cuatro posiciones adyacentes) entonces la complejidad final de esa guarda es $O(2*4)$ que es lo mismo que $O(1)$.
La función va iterando por el conjunto de posiciones que se le pasa como parametro buscando los estudiantes rodeados por agentes. Como en el ciclo las complejidades son constantes la complejidad final da $O((2*4+8)*\#c)$ que es lo mismo que $O(\#c)$.

EstudiantesRodeadosHippies(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `conj(nombre, posición)`
Pre ≡ {true}
Post ≡ {res es el conjunto que indica donde hay posiciones en las que haya estudiantes rodeados de hippies.}

```
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*Columnnas(cs.campus) +
        + Siguiente(itC).x] ≠ " " AND
        AND TodasOcupadas?(vecinos(Siguiente(itC), cs.campus), cs) AND
        AND HippiesAtrapando(vecinos(Siguiente(itC), cs.campus), cs) then (c1)  O(2*4)
        AgregarRapido(res,
            tupla(cs.posicionesEstudiantes[Siguiente(itC).y*Columnas(cs.campus) +
            + Siguiente(itC).x], Siguiente(itC))                                O(1)

    end while                                                                   O((2*4)*#c)
```

Complejidad : $O((2*4*\#c)) = O(\#c)$

Complejidad: ***(c1)*** Las funciones todasOcupadas? y HippiesAtrapando tienen como complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro. Como el conjunto que les estoy pasando solo tiene cuatro elementos (porque la función vecinos devuelve un conjunto de cuatro posiciones adyacentes) entonces la complejidad final de esa guarda es $O(2*4)$ que es lo mismo que $O(1)$.
La función va iterando por el conjunto de posiciones que se le pasa como parametro buscando los estudiates rodeados por hippies. Como en el ciclo las complejidades son constantes la complejidad final da $O((2*4+8)*\#c)$ que es lo mismo que $O(\#c)$.

HippiesAtrapando(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `bool`
Pre ≡ {true}
Post ≡ {res va a ser true si y solo si la cantidad de hippies que hay en el campus, que ademas ocupan las posiciones por parámetro es mayor a dos}

```
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                                          O(1)
            i ← i+1                                                             O(1)
        end if                                                                  O(1)
        Avanzar(itC)                                                            O(1)
```

```
    end while                                                    O(#c)

    res ← i ≥ 2                                                  O(1)
```

Complejidad : $O(\#c)$

Complejidad: La función itera sobre el conjunto de posiciones buscando si hay hippies en cada una de ellas. Como las operaciones son triviales, la complejidad final es la cantidad de elementos que tiene el conjunto pasado por parametro, o sea $O(\#c)$.

SancionarAgentes(**in** c: `conj(As)`, **in/out** cs: `campusseguro`)
Pre $\equiv$ {true}
Post $\equiv$ {Modifica los As pasados por parametro sumándoles uno al número de sanciones respectivo y actualiza las lista listaMismasSanc del campusSeguro}

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

    if Cardinal(c) > 0 then                                      O(1)
        cs.mismasSancModificado ← true                           O(1)
    end if

    while HaySiguiente(itC) do                                   O(1)
        Siguiente(Siguiente(itC).datos).cantSanc ←
            ← Siguiente(Siguiente(itC).datos).cantSanc+1         O(1)

            itLista(kSanc): itLis ← Siguiente(Siguiente(itC).datos).itMismasSanc  O(1)

        if HaySiguiente(Siguiente(Siguiente(itC).datos).itMismasSanc) then  O(1)
            Avanzar(Siguiente(Siguiente(itC).datos).itMismasSanc)

            if Siguiente(Siguiente(Siguiente(itC).datos).itMismasSanc).sanc ≠
            ≠ Siguiente(Siguiente(itC).datos).cantSanc then      O(1)
                AgregarComoAnterior(Siguiente(Siguiente(itC).datos).itMismasSanc,
                    tupla(Siguiente(Siguiente(itC).datos).cantSanc, Vacio()))  O(1)
                Retroceder(Siguiente(Siguiente(itC).datos).itMismasSanc)  O(1)
            end if

        else
            AgregarComoSiguiente(Siguiente(Siguiente(itC).datos).itMismasSanc,
                tupla(Siguiente(Siguiente(itC).datos).cantSanc, Vacio())  O(1)
            Avanzar(Siguiente(Siguiente(itC).datos).itMismasSanc)  O(1)
        end if

        EliminarSiguiente(Siguiente(Siguiente(itC).datos).itConjMismasSanc)  O(1)
        Siguiente(Siguiente(itC).datos).itConjMismasSanc ←
            ← AgregarAdelante(Siguiente(Siguiente(
                Siguiente(itC).datos).itMismasSanc).agentes,
                    Siguiente(itC).agente)                       O(1)
    end while                                                    O(#c)
```

Complejidad : $O(\#c)$

Complejidad: Itera una vez sobre el conjunto de agentes que toma como parametro sumandoles un uno en las sanciones de cada uno y reacomodandolos en la lista listaMismasSanc. Como todas las operaciones del ciclo tienen complejidad $O(1)$ la complejidad total es las veces que lo hace por cada elemento del conjunto. Por lo tanto, la

complejidad es $O(\#c)$.

HippiesRodeadosAs(**in** c: `conj(posición)`, **in/out** cs: `campusseguro`) → res : `conj(nombre, posición`
Pre ≡ {(∀ p ∈ c) posValida?(p)}
Post ≡ {res es el conjunto de hippies que estan rodeados por agentes y estan en alguna de las posiciones pasadas por parametro. Ademas, actualiza los agentes de compus seguro premiandolos cuando sea pertinente.}

```
HippiesRodeadosAs (in c: conj(posición), in cs: estr) → res: conj(nombre, posición)
    itConj(posicion): itC ← CrearIt(c)                                          O(1)
    res ← Vacio()                                                               O(1)

    while HaySiguiente(itC) do                                                  O(1)
        if cs.posicionesHippies[Siguiente(itC).y*Columnas(cs.campus) +
        + Siguiente(itC).y] ≠ " " AND
        todasOcupadas?(vecinos(Siguiente(itDiccS), cs.campus)) AND
        AlMenosUnAgente(vecinos(Siguiente(itDiccS), cs.campus)) then (c1)       O(2*4)
            AgregarRapido(res,
            tupla(cs.posicionesHippies[Siguiente(itC).y*Columnas(cs.campus) +
            + Siguiente(itC).x].agente,
                Siguiente(itC))                                                 O(1)

            conj(As): conjAgPremiar ←
            ← AgParaPremSanc(vecinos(Siguiente(itDiccS), cs), cs) (c2)          O(4)
            PremiarAgentes(conjAgPremiar, cs) (c3)                              O(4)

        end if                                                                  O(1)
        Avanzar(itC)                                                            O(1)
    end while                                                                   O((2*4+8)*#c)
```

Complejidad : $O((2*4+8)*\#c) = O(\#c)$

Complejidad: ***(c1)*** Las funciones todasOcupadas? y AlMenosUnAgente tienen como complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro. Como el conjunto que les estoy pasando solo tiene cuatro elementos (porque la función vecinos devuelve un conjunto de cuatro posiciones adyacentes) entonces la complejidad final de esa guarda es $O(2*4)$ que es lo mismo que $O(1)$.
(c2) AgParaPremSanc idem que lo anterior.
(c3) PremiarAgentes tieme como complejidad $O(\#c)$ siendo c el conjunto que se le pasa como parametro. El conjunto que le estoy pasando como parametro tiene como maximo cuatro elementos porque sale delconjunto que devuelve AgParaPremSanc que toma otro conjunto de cuatro elementos. Por ende, la complejidad en peor caso es de $O(4)$ que es lo mismo que $O(1)$.
La función va iterando por el conjunto de posiciones que se le pasa como parametro buscando los hippies rodeados por agentes. Como en el ciclo las complejidades son constantes la complejidad final da $O((2*4+8)*\#c)$ que es lo mismo que $O(\#c)$.

AgParaPremSanc(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `conj(As)`
Pre ≡ {true}
Post ≡ {res va a a ser el conjunto de As tales que esten en alguna de las posiciones pasadas por parametro}

```
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 HaySiguiente(cs.posicionesAgente[Siguiente(itC).y *
            * Columnas(cs.campus) + Siguiente(itC).x].datos) then               O(1)
            AgregarRapido(res,
```

```
                cs.posicionesAgente[Siguiente(itC).y * Columnas(cs.campus) +
                + Siguiente(itC).x])                                    O(1)
        end if                                                          O(1)
        Avanzar(itC)                                                    O(1)
    end while                                                           O(#c)
```

Complejidad : $O(\#c)$

Complejidad: Se itera sobre el conjunto de posiciones que se pasa como parametro una vez buscando las posiciones que tienen agente y se los va agregando, si los hay. Como se itera una vez y el resto toma $O(1)$ la complejidad de la función es $O(\#c)$.

PremiarAgentes(**in** c: `conj(As)`, **in/out** cs: `campusseguro`)
Pre $\equiv$ {cs $=_{obs}$ cs$_o$}
Post $\equiv$ {Se incrementa la cantidad de atrapados de todos los As pasados como parámetro}

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

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

        if Siguiente(cs.masVigilante.datos).cantAtrapados <
            < Siguiente(itC).cantAtrapados then                         O(1)
            cs.masVigilante ← Siguiente(itC)                            O(1)
        end if

        Avanzar(itC)                                                    O(1)
    end while                                                           O(#c)
```

Complejidad : $O(\#c)$

Complejidad: La función itera sobre el conjunto de As que se le pasa por parametro y modifica todos los As sumandoles un uno en la cantidad de atrapados. Por ende, la complejidad de la función es la cantidad de agentes que tiene el conjunto, o sea $O(\#c)$.

CantHippiesVecinos(**in** c: `conj(posición)`, **in** cs: `campusseguro`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res es igual a la cantidad de hippies que hay ocupando las posiciones pasadas por parametro}

```
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)
        itDiccString(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)
            Avanzar(itDic)                                              O(1)
        end while                                                       O(|n_m|)
```

```
        Avanzar(itC)                                                    O(1)
    end while                                                           O(#c * |n_m|)
```

Complejidad : $O(\#c * |n_m|)$

Complejidad: Itera sobre el conjunto de posiciones que se pasa por parametro y se fija por cada una de ellas si esta en el diccString. Por lo tanto la complejidad es la cantidad de elementos del conjunto por la longitud de la palabra mas larga, $O(\#c * |n_m|)$.

HippiesRodeadosEstudiantes(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `conj(nombre, posición)`
Pre ≡ {true}
Post ≡ {res es igual a un conjunto de tuplas que contienen el nombre y posicion de los hippies de c que hayan sido rodeados por estudiantes}

```
HippiesRodeadosEstudiantes (in c: conj(posición), in cs: estr) → res: conj(nombre, posición)
    itConj(posicion): itC ← CrearIt(c)                                  O(1)
    res ← Vacio()                                                       O(1)

    while HaySiguiente(itC) do                                          O(1)
        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 (c1)  O(4*2)
            AgregarRapido(res,
                tupla(cs.posicionesHippies[Siguiente(itC).y*Columnas(cs.campus) +
        + Siguiente(itC).x], Siguiente(itC))                            O(1)
        end if                                                          O(1)
        Avanzar(itC)                                                    O(1)
    end while                                                           O(4 * 2 * #c)
```

Complejidad : $O(4 * 2 * \#c) = O(\#c)$

Complejidad: ***(c1)*** Las funciones todasOcupadas? y TodosEstudiantes tienen como complejidad $O(\#c)$ siendo c el conjunto que le paso como parametro pero como les estoy pasando un conjunto de exactamente 4 elementos (vecinos me devuelve el conjunto que tiene las cuatro posiciones contiguas a la que le estoy pasando) la complejidad termina siendo $O(4)$ por cada función.
En la función va iterando cada posición del conjunto que se le pasa como parametro y va buscando si hay hippies atrapados. Como por cada ciclo se llama a todasOcupadas? y TodosEstudiantes la complejidad termina siendo $O(4 * 2 * \#c)$ que es lo mismo que $O(\#c)$.

todasOcupadas?(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `bool`
Pre ≡ {true}
Post ≡ {res $=_{obs}$ (∃ p: posicion) Pertenece?(p,c) ∧ ((∃ a: nat) def?(a, agentes(cs)) $\wedge_L$ Obtener(a, agentes(cs)).posición = p) ∨ ((∃ h: string) def?(h, hippies(cs)) $\wedge_L$ Obtener(h, hippies(cs)) = p) ∨ ((∃ e: string) def?(e, estudiantes(cs)) $\wedge_L$ Obtener(e, estudiantes(cs)) = p) ∨ Ocupada?(campus(cs),p)}

```
todasOcupadas? (in c: conj(posicion), in cs: estr) → res: bool
    res ← false                                                         O(1)
    itConj(posicion): itC ← CrearIt(c)                                  O(1)

    while HaySiguiente(itC) AND ¬ res do                                O(1)
        if cs.posicionesHippies[Siguiente(itC)*Columnas(cs.campus) +
        + Siguiente(itC).y] ≠ " " then                                  O(1)
            res ← true                                                  O(1)
```

```
        end if

        if cs.posicionesEstudiantes[Siguiente(itC)*Columnas(cs.campus) +
        + Siguiente(itC).y] ≠ " " then                                    O(1)
            res ← true                                                    O(1)
        end if
        Avanzar(itC)                                                      O(1)
    end while (c1)                                                        O(#c)

    itC ← CrearIt(c)                                                      O(1)
    while HaySiguiente(itC) AND ¬res do                                   O(1)
        if HaySiguiente(cs.posicionesAgente[Siguiente(itC).y *
            * Columnas(cs.campus) + x].datos) then                        O(1)
            res ← true                                                    O(1)
        end if                                                            O(1)
    end while (c2)                                                        O(#c)

    itC ← CrearIt(c)                                                      O(1)
    while HaySiguiente(itC) AND ¬res do                                   O(1)
        if Ocupada?(cs.campus, Siguiente(itC))                            O(1)
            res ← true
        end if                                                            O(1)
        Avanzar(itC)                                                      O(1)
    end while (c3)                                                        O(#c)
```

Complejidad : $O(\#c)$

Justificación de Complejidad: *(c1) (c2) (c3)* En cada ciclo itera sobre la cantidad de elementos del conjunto fijandose si hay un elemento en cada una de las posiciones que tiene el conjunto. Por ende la complejidad es $O(\#c)$.

AlMenosUnAgente(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `bool`
Pre ≡ {true}
Post ≡ {res $=_{obs}$ (∃ p: posición) Pertenece?(p,c) ∧ ((∃ a: nat) def?(a, agentes(cs)) $\wedge_L$ Obtener(a, agentes(cs)).posición = p)}

```
AlMenosUnAgente (in c: conj(posicion), in cs: estr ) → res: bool
    itConj(posicion): itC ← CrearIt(c)                                    O(1)
    res ← false                                                           O(1)

    while HaySiguiente(itC) AND ¬res do                                   O(1)
        if HaySiguiente(cs.posicionesAgente[Siguiente(itC).y *
            * Columnas(cs.campus) + x].datos) then                        O(1)
            res ← true                                                    O(1)
        end if                                                            O(1)
    end while                                                             O(#c)
```

Complejidad : $O(\#c)$

Justificación de Complejidad: Revisa en cada posición del conjunto que se pasa por parametro si alguna tiene un agente. Como lo itera hasta que encuentra una posición en que haya agente puede que no haya o sea el ultimo elemento y da como peor caso que lo tenga que iterar todo. Por lo tanto, la complejidad es $O(\#c)$.

TodosEstudiantes(**in** c: `conj(posición)`, **in** cs: `campusseguro`) → res : `bool`
Pre ≡ {true}
Post ≡ {res $=_{obs}$ (∀ p: posición) Pertenece?(p,c) ⇒ (∃! e: string) def?(e, estudiantes(cs)) $\wedge_L$ Obtener(e, estudian-

tes(cs)) = p}

```
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 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(#c)
```

Complejidad : $O(\#c)$

Justificación de Complejidad: Itera una vez el conjunto buscando si las posiciones que hay en él pertenecen a un estudiante. Como las operaciones del interior del ciclo son triviales la complejidad esta dada por la cantidad de posiciones que tiene el conjunto que se esta iterando. Por lo tanto, la complejidad es $O(\#c)$.

```
iIngresarHippie (in h: nombre, in pos: posicion, in/out cs: estr )
    if todasOcupadas?(vecinos(pos, cs.campus), cs) AND
    AND AlMenosUnAgente(vecinos(pos, cs.campus)) then (c1)                      O(2*4)
        conj(As): conjAgParaPrem ← AgParaPremSanc(vecinos(pos, cs.campus),
            cs) (c2)                                                            O(4)
        PremiarAgentes(conjAgParaPrem, cs) (c3)                                 O(4)
    else if todasOcuapadas?(vecinos(pos, cs.campus), cs) AND
    AND TodosEstudiantes(vecinos(pos, cs.campus), cs) then (c4)                 O(2*4)
        Definir(cs.estudiantes, h, pos)                                         O(|n_m|)
        cs.posicionesEstudiantes[pos.y*Columnas(cs.campus) + pos.x] ← h
    else
        Definir(cs.hippies, h, pos)                                             O(|n_m|)
        cs.posicionesHippies[pos.y*Columnas(cs.campus) + pos.x] ← h
    end if                                                                      O(2*4+|n_m|)


    conj(nombre, posicion): conjHippiesRodEst ←
        ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs )(c5)          O(4)

    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 (f1)                                                    O(4 * 2|n_m|)
end if                                                                O(4 * 2|n_m|)


//Las capturas se actualizan en HippiesRodeadosAs
conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) (c6)             O(4)

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] ← " "                                       O(1)
        Avanzar(itHAs)                                                O(1)
    end while (f2)                                                    O(4|n_m|)
end if                                                                O(4|n_m|)


conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) (c7)    O(4)

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)
    end while (f3)                                                    O(4 * 2|n_m|)
end if                                                                O(4 * 2|n_m|)


conj(posicion): conjEstRodAs ←
    ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) (c8)         O(4)

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 (c9) O(2 * 4)
            conj(As): conjAgParaSanc ←
                ← AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus),
                    cs) (c10)                                         O(4)
            SancionarAgentes(conjAgParaSanc, cs) (c11)                O(4)
        end if                                                        O(1)
    end while (f4)                                                    O(4 * 4 * 4)
```

```
    end if                                                          O(4 * 4 * 4)
```

Complejidad : $O(2*4+|n_m|)+O(4*2*|n_m|)+O(4*|n_m|)+O(4*2|n_m|)+O(4*4*4) = O(8)+O(21|n_m|) = O(1)+O(|n_m|) = O(|n_m|)$

Justificación de Complejidad: ***(c1)*** Las funciones todasOcupadas? y AlMenosUnAgente tienen como complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro. Como el conjunto que les estoy pasando solo tiene cuatro elementos (porque la función vecinos devuelve un conjunto de cuatro posiciones adyacentes) entonces la complejidad final de esa guarda es $O(2*4)$ que es lo mismo que $O(1)$.
(c2) AgParaPremSanc idem que el anterior punto, la complejidad es $O(\#c)$ pero le estoy pasando un conjunto de maximo 4 elementos la complejidad es $O(4)$ que es lo mismo que $O(1)$.
(c3) PremiarAgentes idem que el anterior punto, la complejidad es $O(\#c)$ pero le estoy pasando un conjunto de maximo 4 elementos la complejidad es $O(4)$ que es lo mismo que $O(1)$.
(c4) todasOcupadas? y TodosEstudiantes es lo mismo que el primer item, tienen ambas complejidad $O(\#c)$ pero se les pasa de parametro un conjunto de 4 elementos por ende su complejidad termina siendo $O(4)$ que es lo mismo que $O(1)$.
(c5) HippiesRodeadosEstudiantes tiene como complejidad $O(\#c)$ siendo c el conjunto que se le pasa como parametro pero como le estoy pasando un conjunto que maximo tiene 4 elementos la complejidad termina siendo $O(4)$ que es lo mismo que $O(1)$.
(c6) HippiesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c7) EstudiantesRodeadosHippies idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c8) EstudiantesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c9) idem que el primer punto, todasOcupadas? y AlMenosUnAgente tienen complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro, pero como este conjunto tiene como maximo 4 elementos (porque vecinos devuelve un conjunto con maximo 4 elementos), las complejidades de ambas funciones es $O(4)$ que es lo mismo que $O(1)$.
(c10) AgParaPremSanc idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c11) SancionarAgentes idem que el punto anterior, termina teniendo complejidad $O(1)$.
(f1) El ciclo itera sobre el conjunto conjHippiesRodEst que tiene como máximo 4 elementos y dentro de él se define un estudiante ($O(|n_m|)$) y se elimina un hippie ($O(|n_m|)$), por lo tanto la complejidad del mismo es $O(4*2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f2) Idem que el punto anterior, el conjunto conjHippiesRodAs tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4*|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f3) Idem que el punto anterior, el conjunto conjEstRodHip tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4*2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f4) Idem que el anterior pero como las operaciones están acotadas a cuatro elementos la complejidad queda como $O(4*4*4)$ que es lo mismo que $O(1)$.

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 = 0 OR pos.y = cd.campus.filas+1)) then           O(1)
        if CantHippiesVecinos(vecinos(pos, cs.campus), cs)<2 then (c1)   O(4*|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(4*|n_m|+2*|n_m|)


conj(nombre, posicion): conjHippiesRodEst ←
    ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs) (c2)               O(4)

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 (f1)                                                               O(4*2|n_m|)
end if                                                                           O(4*2|n_m|)


//Las capturas se actualizan en HippiesRodeadosAs
conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) (c3)                        O(4)

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] ← " "                                                  O(1)
        Avanzar(itHAs)                                                           O(1)
    end while (f2)                                                               O(4*|n_m|)
end if                                                                           O(4*|n_m|)


conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) (c4)               O(4)

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)
        end while (f3)                                                        O(4 * 2|n_m|)
    end if                                                                    O(4 * 2|n_m|)


    conj(posicion): conjEstRodAs ←
        ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) (c5)             O(4)

    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 (c6)  O(2 * 4)
                conj(As): conjAgParaSanc ←
                    ← AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus),
                        cs) (c7)                                              O(4)
                SancionarAgentes(conjAgParaSanc, cs) (c8)                     O(4)
            end if                                                            O(1)
        end while (f4)                                                        O(4 * 4 * 4)

    end if                                                                    O(4 * 4 * 4)
```

Complejidad : $O(|n_m|) + O(4 * |n_m| + 2 * |n_m|) + O(4 * 2|n_m|) + O(4 * |n_m|) + O(4 * 2|n_m|) + O(4 * 4 * 4) = O(27|n_m|) + O(64) = O(|n_m|) + O(1) = O(|n_m|)$

Justificación de Complejidad: ***(c1)*** CantHippiesVecinos tiene como complejidad $O(\#c * |n_m|)$ siendo c el conjunto que se le pasa como parametro. Como el conjunto que le paso como parametro tiene como maximo 4 posicines (porque la funcion vecinos devuelve un conjunto de las 4 posiciones adyacentes a la que le paso por parametro) la complejidad que da es $O(4 * |n_m|)$ que es lo mismo que $O(|n_m|)$.
(c2) HippiesRodeadosEstudiantes tiene como complejidad $O(\#c)$ siendo c el conjunto que se le pasa como parametro pero como le estoy pasando un conjunto que maximo tiene 4 elementos (porque la función vecinos devuelve un conjunto de las 4 posiciones adyacentes a la que le paso por parametro) la complejidad termina siendo $O(4)$ que es lo mismo que $O(1)$.
(c3) HippiesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c4) EstudiantesRodeadosHippies idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c5) EstudiantesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c6) idem que el primer punto, todasOcupadas? y AlMenosUnAgente tienen complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro, pero como este conjunto tiene como maximo 4 elementos (porque vecinos devuelve un conjunto con maximo 4 elementos), las complejidades de ambas funciones es $O(4)$ que es lo mismo que $O(1)$.
(c7) AgParaPremSanc idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c8) SancionarAgentes idem que el punto anterior, termina teniendo complejidad $O(1)$.
(f1) El ciclo itera sobre el conjunto conjHippiesRodEst que tiene como máximo 4 elementos y dentro de él se define un estudiante ($O(|n_m|)$) y se elimina un hippie ($O(|n_m|)$), por lo tanto la complejidad del mismo es $O(4 * 2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f2) Idem que el punto anterior, el conjunto conjHippiesRodAs tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4 * |n_m|)$ que es lo mismo que $O(|n_m|)$.
(f3) Idem que el punto anterior, el conjunto conjEstRodHip tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4 * 2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f4) Idem que el anterior pero como las operaciones están acotadas a cuatro elementos la complejidad queda como

$O(4 * 4 * 4)$ que es lo mismo que $O(1)$.

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

```
posicion : actualPos ← obtener(h, cs.hippies)                                  O(|n_m|)
posicion : pos ← ProxPos(actualPos, cs.estudiantes, cs) (c1)                   O(N_e)
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)
end if                                                                         O(|n_m|)

conj(nombre, posicion): conjHippiesRodEst ←
    ← HippiesRodeadosEstudiantes(vecinos(pos, cs.campus), cs) (c2)             O(4)

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 (f1)                                                             O(4 * 2|n_m|)
end if                                                                         O(4 * 2|n_m|)


//Las capturas se actualizan en HippiesRodeadosAs
conj(nombre, posicion): conjHippiesRodAs ←
    ← HippiesRodeadosAs(vecinos(pos, cs.campus), cs) (c3)                      O(4)

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] ← " "                                                O(1)
        Avanzar(itHAs)                                                         O(1)
    end while (f2)                                                             O(4 * |n_m|)
end if                                                                         O(4 * |n_m|)


conj(nombre, posicion): conjEstRodHip ←
    ← EstudiantesRodeadosHippies(vecinos(pos, cs.campus), cs) (c4)             O(4)

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)
        end while (f3)                                                         O(4*2|n_m|)
    end if                                                                     O(4*2|n_m|)


    conj(posicion): conjEstRodAs ←
        ← EstudiantesRodeadosAs(vecinos(pos, cs.campus), cs) (c5)              O(4)

    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 (c6)  O(2*4)
                conj(As): conjAgParaSanc ←
                    ← AgParaPremSanc(vecinos(Siguiente(itEstAs), cs.campus),
                        cs) (c7)                                               O(4)
                SancionarAgentes(conjAgParaSanc, cs) (c8)                      O(4)
            end if                                                             O(1)
        end while (f4)                                                         O(4*4*4)

    end if                                                                     O(4*4*4)
```

Complejidad : $O(|n_m|) + O(N_e) + O(|n_m|) + O(4*2|n_m|) + O(4|n_m|) + O(4*2|n_m|) + O(4*4*4) = O(22|n_m|) + O(N_e) + O(64) = O(|n_m|) + O(N_e) + O(1) = O(|n_m|) + O(N_e)$

Justificación de Complejidad: ***(c1)*** ProxPos tiene como complejidad $O(\#Claves(d))$ siendo d el diccionario que se le pasa como parametro. Como el diccionario que le paso como parametro en este caso es el diccionario de los estudiantes la complejidad es la cantidad de estudiantes que hay, o sea, $O(N_e)$.
(c2) HippiesRodeadosEstudiantes tiene como complejidad $O(\#c)$ siendo c el conjunto que se le pasa como parametro pero como le estoy pasando un conjunto que maximo tiene 4 elementos (porque la función vecinos devuelve un conjunto de las 4 posiciones adyacentes a la que le paso por parametro) la complejidad termina siendo $O(4)$ que es lo mismo que $O(1)$.
(c3) HippiesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c4) EstudiantesRodeadosHippies idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c5) EstudiantesRodeadosAs idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c6) idem que el segundo punto, todasOcupadas? y AlMenosUnAgente tienen complejidad $O(\#c)$ siendo c el conjunto que se les pasa como parametro, pero como este conjunto tiene como maximo 4 elementos (porque vecinos devuelve un conjunto con maximo 4 elementos), las complejidades de ambas funciones es $O(4)$ que es lo mismo que $O(1)$.
(c7) AgParaPremSanc idem que el punto anterior, termina teniendo complejidad $O(1)$.
(c8) SancionarAgentes idem que el punto anterior, termina teniendo complejidad $O(1)$.
(f1) El ciclo itera sobre el conjunto conjHippiesRodEst que tiene como máximo 4 elementos y dentro de él se define un estudiante ($O(|n_m|)$) y se elimina un hippie ($O(|n_m|)$), por lo tanto la complejidad del mismo es $O(4*2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f2) Idem que el punto anterior, el conjunto conjHippiesRodAs tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4*|n_m|)$ que es lo mismo que $O(|n_m|)$.

(f3) Idem que el punto anterior, el conjunto conjEstRodHip tiene 4 elementos e itera sobre ellos, en este caso la complejidad queda $O(4 * 2|n_m|)$ que es lo mismo que $O(|n_m|)$.
(f4) Idem que el anterior pero como las operaciones están acotadas a cuatro elementos la complejidad queda como $O(4 * 4 * 4)$ que es lo mismo que $O(1)$.

ProxPos(**in** *pos*: `posicion`, **in** *d*: `dicString(nombre,posicion)`, **in** *cs*: `estr`) → *res* : `posicion`
Pre ≡ {posValida?(pos)}
Post ≡ {*res* es la posicion a la cual deberia moverse quien esté en *pos* dependiendo de como estén distribuidos los hippies o estudiantes en *d*}

```
ProxPos (in pos: posicion, in d: dicString(nombre,posicion), in cs: estr) → res: posicion
    int: distCorta ← DistanciaMasCorta(pos, d) (c1)                          O(#Claves(d))

    conj(posicion): conjDondeIr ← DondeIr(pos, distCorta, d) (c2)            O(#Claves(d))
    conj(posicion): conjLugaresPosibles ←
        ← LugaresPosibles(pos,conjDondeIr, cs) (c3)                          O(#Claves(d))

    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 : $O(\#Claves(d)) + O(\#Claves(d)) + O(\#Claves(d)) = O(\#Claves(d))$

Complejidad: ***(c1)*** La función DistanciaMasCorta tiene como complejidad $O(\#Claves(d))$ porque itera sobre ellas buscando la distancia mas corta a la posición que tambien tiene como parametro.
(c2) DondeIr tiene como complejidad $O(\#Claves(d))$ porque itera sobre las claves del diccionario que se le pasa por parametro.
(c3) LugaresPosibles tiene como complejidad en realidad $O(\#c)$ siendo c el conjunto que se le pasa como parametro, pero como la cantidad de elementos del conjunto que devuelve no puede ser mayor que la cantidad de claves del diccionario se lo puede acotar estas. Por lo tanto, la complejidad es $O(\#Claves(d))$.

LugaresPosibles(**in** *pos*: `posicion`, **in** *c*: `conj(posicion)`, **in** *cs*: `estr`) → *res* : `conj(posicion)`
Pre ≡ {posValida?(pos)}
Post ≡ {*res* es el conjunto de posiciones a las cuales es posible moverse desde *pos*}

```
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              O(1)
            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         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         O(1)
            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         O(1)
            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))                        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           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           O(1)
        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           O(1)
        AgregarRapido(res, tupla(pos.x+1, pos.y))                           O(1)
    end if                                                                  O(1)

    Avanzar(itC)                                                            O(1)
end while (c1)                                                              O(#c)

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

while HaySiguiente(itPosibles) do                                           O(1)
    if HayAlgoEnPos(Siguiente(itPosibles), cs) then (c2)                    O(1)
        EliminarSiguiente(itPosibles)                                       O(1)
    end if                                                                  O(1)
    Avanzar(itPosibles)                                                     O(1)
end while (c3)                                                              O(#c)
```

Complejidad : $O(\#c) + O(\#c) = O(\#c)$

Complejidad: ***(c1)*** El ciclo itera sobre la cantidad de elementos que tiene el conjunto que se pasa por parametro. Como las operaciones que hay dentro de él tienen complejidad $O(1)$ la complejidad del ciclo termina siendo la cantidad de veces que itera que es $O(\#c)$.
(c2) La función HayAlgoEnPos se fija si hay algo en la posición que le pasamos como parametro y cuesta $O(1)$ por las estructuras que tenemos.
(c3) Idem que el primer punto, la complejidad termina siendo $O(\#c)$.

HayAlgoEnPos(**in** *pos*: `posicion`, **in** *cs*: `estr`) → *res* : `bool`
Pre ≡ {posValida?(pos, cs)}
Post ≡ {*res* es true si y solo si hay algo, ya sea hippie, estudiante, agente u obstáculo, en *pos*}

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

    if HaySiguiente(cs.posicionesAgente[pos.y*Columnas(cs.campus) + pos.x].datos) 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 : $O(1)$

Complejidad: Trivial, se fija si hay algo en la posición y como usamos vectores, ver en una posición de un vector cuesta $O(1)$.

DondeIr(**in** *pos*: `posicion`, **in** *dist*: `nat`, **in** *d*: `diccString(nombre,posicion)`) → *res* : `conj(posicion)`
Pre ≡ {posValida(pos)}
Post ≡ {*res* el el conjunto de posiciones, que representan individuos de *d*, cuya distancia desde *pos* hasta ellas es *dist*}

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 (c1)         O(1)
        AgregarRapido(res, Siguiente(itDicc))                    O(1)
    end if                                                       O(1)
    Avanzar(itDicc)                                              O(1)
end while (c2)                                                   O(#Claves(d))
```

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

Complejidad: ***(c1)*** La función Distancia devuelve la distancia que hay entre las dos posiciones que paso como parametro, lo hace en $O(1)$.
(c2) Como las operaciones dentro del ciclo tienen complejidad $O(1)$ la complejidad del mismo termina siendo la cantidad de claves que tiene el diccionario por el cual estoy iterando que es $O(\#Claves(d))$.

DistanciaMasCorta(**in** *pos*: `posicion`, **in** *d*: `dicString(nombre, posicion)`) → *res* : `nat`
Pre ≡ {#Claves(d))>0}
Post ≡ {Para toda posicion definida en d, res es menor o igual que la distancia entre pos y la posicion.}

DistanciaMasCorta (**in** *pos*: `posicion`, **in** *d*: `dicString(nombre, posicion)`) → res: nat

```
itDiccString(nombre,posicion): itDicc ← CrearIt(d)               O(1)
res ← Distancia(pos, Siguiente(itDicc)) (c1)                     O(1)
Avanzar(itDicc)                                                  O(1)

while HaySiguiente(itDicc) do                                    O(1)
    if res > Distancia(pos, Siguiente(itDicc))                   O(1)
        res ← Distancia(pos, Siguiente(itDicc))                  O(1)
    end if                                                       O(1)
    Avanzar(itDicc)                                              O(1)
end while (c2)                                                   O(#Claves(d))
```

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

Complejidad: ***(c1)*** La función Distancia devuelve la distancia que hay entre las dos posiciones que paso como parametro, lo hace en $O(1)$.
(c2) Como las operaciones dentro del ciclo tienen complejidad $O(1)$ la complejidad del mismo termina siendo la cantidad de claves que tiene el diccionario por el cual estoy iterando que es $O(\#Claves(d))$.

Distancia(**in** *pos*1, *pos*2: `posicion`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res $=_{\text{obs}}$ $|pos2.x - pos1.x| + |pos2.y - pos1.x|$}

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

```
res ← |pos2.x − pos1.x| + |pos2.y − pos1.x|                    O(1)
```

Complejidad : $O(1)$

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

```
itDiccNat(agente, datosAgente) it
    ← busqBinPorPlaca(cs.ordenadoPorPlaca, a)                  O(log(N_a))

// Actualizo la posicion del agente
posicion nuevaPos ← ProxPos(Siguiente(it).posicion, cs.hippies, cs)   O(N_h)
posicionesAgentes[Siguiente(it).posicion.y * columnas(cs.campus)
    + Siguiente(it).posicion.x].datos ← CrearIt(Vacio())       O(1)
posicionesAgentes[nuevaPos.y * columnas(cs.campus) + nuevaPos.x]
    ← tupla(a, it)                                             O(1)
Siguiente(it).posicion ← nuevaPos                              O(1)

// Me fijo a quienes atrapa
posicion posArr ← moverDir(cs.campus, nuevaPos, arriba)        O(1)
actualizarAgente(cs, posArr, a, it)                            O(|n_m|)

posicion posAba ← moverDir(cs.campus, nuevaPos, abajo)         O(1)
actualizarAgente(cs, posAba, a, it)                            O(|n_m|)

posicion posDer ← moverDir(cs.campus, nuevaPos, der)           O(1)
actualizarAgente(cs, posDer, a, it)                            O(|n_m|)

posicion posIzq ← moverDir(cs.campus, nuevaPos, izq)           O(1)
actualizarAgente(cs, posIzq, a, it)                            O(|n_m|)
```

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

Complejidad: Hace busqueda binaria sobre el vector de agentes ordenado por placa. Esto cuesta $O(log(N_a)$. Luego muevo al agente, para lo cual debo calcular primero la posición. Esto cuesta $O(N_H)$ ya que debo preguntar por las posiciones de todos los hippies para decidir. Finalmente actualizo el estado del agente, fijandome a quienes atrapa. En caso de atrapar un hippie debo borrarlo del diccString, que cuesta $O(|n_m|)$.

ACTUALIZARAGENTE(**in/out** cs : `estr`, **in** $posicion$: `pos`, **in** a : `agente`, **in** it : `itDiccNat(agente, datosAgente)`)
Pre ≡ {pos debe ser una posicion contigua y sin obstáculos}
Post ≡ {Se fija en pos si hay alguien, si hay un hippie y fue atrapado, lo borra e incrementa la cantidad de atrapados del agente, si hay un estudiante y fue atrapado, incrementa la cantidad de sanciones}

actualizarAgente (**in/out** cs : `estr`, **in** $posicion$: `pos`, **in** a : `agente`, **in** it : `itDiccNat(agente, datosAgente)`)

```
if(posValida?(cs.campus, pos) then                             O(1)

    if(posicionesHippies[pos.y * columnas(cs.campus) + pos.x] ≠ "") then
        if(atrapado?(cs, pos)) then                            O(1)
            // Le sumo uno a sus capturas, actualizo masVigilante y mato al hippie
            Siguiente(it).cantAtrapados++;                     O(1)
            if(Siguiente(it).cantAtrapados >
                (*cs.masVigilante.datos).cantAtrapados) then
                 cs.masVigilante ← tupla(a, it)                O(1)
```

```
            end if
            borrar(cs.hippies, posicionesHippies[pos.y *
                   columnas(cs.campus) + pos.x])                    O(|n_m|)
            posicionesHippies[pos.y * columnas(cs.campus) + pos.x]
                ← ""                                                O(1)
        end if
    end if

    if(posicionesEstudiantes[pos.y * columnas(cs.campus) + pos.x] ≠ "") then
        if(atrapado?(cs, pos)) then                                 O(1)
            // Actualizo las sanciones y las estructuras relacionadas
            cs.mismasSancModificado ← true                          O(1)

            itConj(agente) iterConj ← Siguiente(it).itConjMismasSanc O(1)
            EliminarSiguiente(iterConj)                             O(1)

            itLista(kSanc) iterLista ← Siguiente(it).itMismasSanc   O(1)
            // Me guardo un iterador para borrar el nodo de la lista
            // si es que queda sin agentes
            itLista(kSanc) iterListaAnterior ← Siguiente(it).itMismasSanc O(1)
            if(HaySiguiente?(iterLista) then                        O(1)
                // Me fijo si el siguiente es la siguiente sancion
                nat : sanciones ← Siguiente(iterLista).sanc         O(1)
                Avanzar(iterLista)                                  O(1)

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

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

            Siguiente(it).cantSanc++;                               O(1)
        end if
    end if
```

```
    end if
```
Complejidad : $O(|n_m|)$

Complejidad: Se fija a quién atrapa en la dirección pasada como parametro. Todas las operaciones son de tiempo constante con excepción de la de borrar un hippie del diccString, que se ejecuta en caso de haber atrapado un hippie. Esta operación cuesta $O(|n_m|)$.

```
iCampus (in cs: estr ) → res: campus
    res ← cs.campus                                                    O(1)
```
Complejidad : $O(1)$

Complejidad: Devuelve una referencia a campus en tiempo constante.

```
iEstudiantes (in cs: estr ) → res: itConj(nombre)
    res ← CrearItClaves(cs.estudiantes)                                O(1)
```
Complejidad : $O(1)$

Complejidad: Crea un iterador de estudiantes y lo devuelve en tiempo constante.

```
iHippies (in cs: estr ) → res: itConj(nombre)
    res ← CrearItClaves(cs.hippies)                                    O(1)
```
Complejidad : $O(1)$

Complejidad: Crea un iterador de hippies y lo devuelve en tiempo constante.

```
iAgentes (in cs: estr ) → res: itConj(agente)
    res ← CrearItClaves(cs.personalAS)                                 O(1)
```
Complejidad : $O(1)$

Complejidad: Crea un iterador de agentes y lo devuelve en tiempo constante.

```
iPosEstudianteYHippie (in id: nombre, in cs: estr ) → res: posición
    if Definido?(id, cs.hippies) then                                  O(|n_m|)
        res ← Obtener(id, cs.hippies)                                  O(|n_m|)
    else
        res ← Obtener(id, cs.estudiantes)                              O(|n_m|)
    end if
```
Complejidad : $O(|n_m|) + O(4 * 2|n_m|) = O(|n_m|)$

Complejidad: En el peor de los casos llama a la funcion definido del diccString que itera sobre los caracteres de la palabra y despues hace una llamada al obtener

```
iPosAgente (in a: agente, in cs: estr ) → res: posición
    res ← Obtener(a, cs.personalAS).posicion                           O(1 caso promedio)
```
Complejidad : $O(1)$ caso promedio

```
iCantSanciones (in a: agente, in cs: estr )
    res ← Obtener(a, cs.personalAS).cantSanc                    O(1 caso promedio)
```
Complejidad : $O(1)$ caso promedio

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

```
iConMismasSanciones (in a: agente, in cs: estr ) → res: conj(agente)
    res ← Siguiente(Obtener(a, cs.personalAS).itMismasSanc).agentes   O(1 caso promedio)
```
Complejidad : $O(1)$ caso promedio

```
iConKSanciones (in k: nat, in cs: estr ) → res: conj(agente)
    if cs.mismasSancModificado == true then                     O(1)
        hacerArregloMismasSanc(cs)                              O(N_a)
        cs.mismasSancModificado ← false
    end if

    nat: i ← 0
    bool: esta ← busqBinAgente(k, i, cs.arregloMismasSanc)      O(logN_a)
    if esta = true then                                         O(1)
        res ← *cs.arregloMismasSanc[i].agentes                  O(1)
    else
        res ← Vacia()                                           O(1)
    end if
```
Complejidad : Si las sanciones no fueron modificadas: $O(N_a)$. Si no $O(logN_a)$

HACERARREGLOMISMASSANC(**in/out** *cs*: estr)
Pre ≡ {true}
Post ≡ {El arreglo arregloMismasSanc de cs tiene en sus posiciones punteros a ksanc}

```
hacerArregloMismasSanc (in/out cs: estr)
    arreglo(puntero(kSanc)): arregloNuevo ← CrearArreglo(Longitud(cs.listaMismasSanc))
                                                                O(N_a)
    itLista(kSanc): it ← CrearIt(cs.listaMismasSanc)            O(1)
    nat: i ← 0                                                  O(1)

    while HaySiguiente(it) do                                   O(1)
        puntero(kSanc): p ← puntero(Siguiente(it))              O(1)
        arregloNuevo[i] ← p                                     O(1)
        i ← i +1                                                O(1)
        Avanzar(it)                                             O(1)
    end while                                                   O(N_a)

    cs.arregloMismasSanc ← arregloNuevo                         O(1)
```

Complejidad : $O(N_a) + O(N_a) = O(N_a)$ Como máximo va a ser la cantidad de agentesla lista listaMismasSanc ya que no se pueden tener mas conjuntos con mismas sanciones que cantidad de agentes.

BUSQBINAGENTE(**in** k: `nat`, **in/out** i: `nat`, **in** v: `arreglo(puntero(kSanc))`) → res : `bool`
Pre ≡ {Longitud(v)>0}
Post ≡ {res $=_{obs}$ esta?(k, v) ∧ i va a ser la posicion en V en la que se encuentra k. Si k no esta, i se queda igual que como entro por parametro}

```
busqBinAgente (in k: nat, in/out i: nat, in v: arreglo(puntero(kSanc))) → res: bool
    nat: n ← 0                                                          O(1)
    nat: m ← Longitud(v)                                                O(1)
    nat: med

    while n ≠ m−1 do                                                    O(1)
        med ← (n+m)/2                                                   O(1)
        if med ≤ k then                                                 O(1)
            n ← med                                                     O(1)
        else
            m ← med                                                     O(1)
        end if
    end while                                                           O(log(N_a))

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

Complejidad : $O(logN_a)$ Ya que en cada iteracion se queda con la mitad del arreglo, tal como esta pensada la busqueda binaria.

```
Atrapado? (in c: campus, in pos: Posicion) → res: bool
    res ← todasOcupadas?(vecinos(pos, cs.campus))                       O(1)
```

BUSQBINPORPLACA(**in** a: `agente`, **in** v: `vector(As)`) → res : `itDiccNat(agente, datosAgente)`
Pre ≡ {(∃ elem : As)(esta?(elem, v) ∧ agente(elem) = a) ∧ ordenadoPorPlaca?(v)}
Post ≡ {res es igual al *datos* del elemento de v que tiene a a como *agente*}

```
busqBinPorPlaca (in a: agente, in v: vector(As)) → res: itDiccNat(agente, datosAgente)
    nat: inf ← 0                                                        O(1)
    nat: sup ← Longitud(v)                                              O(1)
    nat: med ← (inf+sup)/2                                              O(1)

    while inf ≠ sup−1 do                                                O(1)
        med ← (inf+sup)/2                                               O(1)
        if v[med].agente ≤ a then                                       O(1)
            inf ← med + 1                                               O(1)
        else
            sup ← med                                                   O(1)
        end if
    end while                                                           O(log(N_a))
```

```
    res ← v[inf].datos
```

Complejidad : $O(logN_a)$ idem busqueda binaria anterior.

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: $O(alto * ancho)$
Descripción: Crea un nuevo campus tomando un alto y un ancho

AGREGAROBSTACULO(**in/out** *c*: `campus`, **in** *pos*: `posicion`)
Pre ≡ {$c =_{obs} c_0 \wedge$ posValida?(pos,c) $\wedge_{\text{L}}$ ¬ocupada?(pos,c)}
Post ≡ {$c =_{obs}$ agregarObstaculo(pos,c_0)}
Complejidad: $O(1)$
Descripción: Agrega un obstaculo 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*: `posicion`) → *res* : `bool`
Pre ≡ {posValida(pos,c)}
Post ≡ {res =$_{obs}$ ocupada(pos,c)}
Complejidad: $O(1)$
Descripción: Devuelve true si la posicion esta ocupada por un obstaculo

POSVALIDA?(**in** *c*: `campus`, **in** *pos*: `posicion`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {$res =_{obs}$ posValida?(c,pos)}
Complejidad: $O(1)$
Descripción: Devuelve true si la posicion es valida

ESINGRESO?(**in** *c*: `campus`, **in** *pos*: `posicion`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {$res =_{obs}$ esIngreso?(pos,c)}
Complejidad: $O(1)$
Descripción: Devuelve true si la posicion es un ingreso. No tiene en cuenta su validez

INGRESOSUPERIOR?(**in** c: `campus`, **in** pos: `posicion`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{obs}$ 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`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{obs}$ 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`) → res : `conj(posicion)`
Pre ≡ {posValida?(pos,c)}
Post ≡ {$res =_{obs}$ 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`) → res : `nat`
Pre ≡ {true}
Post ≡ {$res =_{obs}$ distancia($p1$,$p2$,c)}
Complejidad: $O(1)$
Descripción: Devuelve la distancia entre dos posiciones

MOVERDIR(**in** c: `campus`, **in** pos: `posicion`, **in** dir: `direccion`) → res : `posicion`
Pre ≡ {posValida(pos,c)}
Post ≡ {$res =_{obs}$ 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`) → res : `conj(posicion)`
Pre ≡ {posValida(pos,c)}
Post ≡ {$res =_{obs}$ 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 $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ long(obstaculos) $\leq$ filas * columnas

2.1.4. Función de Abstracción

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

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

2.2. Algoritmos

iCrearCampus (**in** *alto*: `nat`, **in** *ancho*: `nat`) $\rightarrow$ res: estr
{**Pre** $\equiv$ true}

```
res.filas ← alto                                          O(1)
res.columnas ← ancho                                      O(1)

nat : pos ← 0                                             O(1)
res.obstaculos ← Vacio()                                  O(1)
while(pos < alto * ancho) do                              O(1)
    AgregarAtras(res.obstaculos, false)                   O(1)
    pos ← pos + 1                                         O(1)
end while                                                 O(alto * ancho)
```

{**Post** $\equiv$ res $=_{obs}$ crearCampus(alto , ancho)}}
Complejidad : $O(alto * ancho)$ Justificacion: Realiza alto*ancho veces un agregarAtras en una lista vacia

iAgregarObstaculo (**in/out** *cs*: `campus`, **in** *pos*: `posicion`)
{**Pre** $\equiv$ $cs =_{obs} cs_0 \wedge$ posValida?(pos,cs) $\wedge_L$ $\neg$ocupada?(pos,cs)}

```
cs.obstaculos[pos.y * cs.columnas + pos.x] ← true         O(1)
```

{**Post** $\equiv$ $cs =_{obs}$ agregarObstaculo(pos,cs_0)}
Complejidad : $O(1)$ Justificacion: Pone en true el valor almacenado en la lista de obstaculos en la posicionque se resuelve en tiempo constante con una multiplicacion y una suma

iFilas (**in** *cs*: `campus`) → res: nat
{**Pre** ≡ true}

```
res ← cs.filas                                                    O(1)
```

{**Post** ≡ res $=_{obs}$ filas(cs)}
Complejidad : $O(1)$ Justificacion: Retorna el campo filas del campus

iColumnas (**in** *cs*: `campus`) → res: nat
{**Pre** ≡ true}

```
res ← cs.columnas                                                 O(1)
```

{**Post** ≡ res $=_{obs}$ columnas(cs)}
Complejidad : $O(1)$ Justificacion: Retorna el campo columnas del campus

iOcupada? (**in** *cs*: `campus`, **in** *pos*: `posicion`) → res: bool
{**Pre** ≡ posValida?(c, pos)}

```
res ← cs.obstaculos[pos.y * cs.columnas + pos.x]                  O(1)
```

{**Post** ≡ res $=_{obs}$ ocupada?(pos, cs)}
Complejidad : $O(1)$ Justificacion: Solo devuelve el campo almcenado en la lista de obstaculosque contiene el booleano que indica si hay o no un obstaculo

iPosValida? (**in** *cs*: `campus`, **in** *pos*: `posicion`) → res: bool
{**Pre** ≡ true}

```
res ← (pos.x ≥ 0) ∧ (pos.x < cs.columnas) ∧
      (pos.y ≥ 0) ∧ (pos.y < cs.filas)                            O(1)
```

{**Post** ≡ res $=_{obs}$ posValida?(pos, cs)}
Complejidad : $O(1)$ Justificacion: Solo compara los valores x e y de la posicion con los valoresalto y ancho del campus

iEsIngreso (**in** *cs*: `campus`, **in** *pos*: `posicion`) → res: bool
{**Pre** ≡ true}

```
res ← esIngresoSuperior(cs, pos) ∨ esIngresoInferior(cs, pos)     O(1)
```

{**Post** ≡ res $=_{obs}$ esIngreso?(pos, cs)}
Complejidad : $O(1)$ Justificacion: Hace una operacion OR sobre el resultado de las llamadas a lasfunciones esIngresoSuperior e esIngresoInferior que son O(1)

iEsIngresoSuperior (**in** *cs*: `campus`, **in** *pos*: `posicion`) → res: bool
{**Pre** ≡ true}

```
res ← pos.y == 0                                                  O(1)
```

{**Post** ≡ res $=_{obs}$ esIngresoSuperior?(pos, cs)}
Complejidad : $O(1)$ Justificacion: Solo compara el valor y de la posicion con 0

iEsIngresoInferior (**in** *cs*: `campus`, **in** *pos*: `posicion`) → res: bool
{**Pre** ≡ `true`}

```
res ← pos.y == cs.filas - 1                                   O(1)
```

{**Post** ≡ `res` $=_{obs}$ `esIngresoInferior?(pos, cs)`}
Complejidad : $O(1)$ Justificacion: Solo compara el valor y de la posicion con el valor defilas -1 del campus

iVecinos (**in** *cs*: `campus`, **in** *pos*: `posicion`) → res: conj(posicion)
{**Pre** ≡ `posValida?(c, pos)`}

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

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

{**Post** ≡ `res` $=_{obs}$ `vecinos(pos, cs)`}
Complejidad : $O(1)$ Justificacion: Llama cuatro veces a la funcion posValida y en el peor casorealiza cuantro AgregarRapido a una lista

iDistancia (**in** *cs*: `campus`, **in** pos_1: `posicion`, **in** pos_2: `posicion`) → res: nat
{**Pre** ≡ `true`}

```
res ← |pos1.x - pos2.x| + |pos1.y - pos2.y|                   O(1)
```

{**Post** ≡ `res` $=_{obs}$ `distancia(p1, p2, cs)`}
Complejidad : $O(1)$ Justificacion: Realiza dos restas y modulos y una suma de enteros

iMoverDir (**in** *cs*: `campus`, **in** *pos*: `posicion`, **in** *dir*: `direccion`) → res: posicion
{**Pre** ≡ `posValida?(pos, cs)`}

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

{**Post** ≡ `res` $=_{obs}$ `proxPosicion(pos, dir, cs)`}
Complejidad : $O(1)$ Justificacion: Realiza cuatro comparaciones y una asignacion

iIngresosMasCercanos (**in** *cs*: `campus`, **in** *pos*: `posicion`) → res: conj(posicion)
{**Pre** ≡ posValida?(pos, cs)}

```
res ← Vacio()                                                          O(1)

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)
```

{**Post** ≡ res $=_{obs}$ ingresosMasCercanos(pos, cs)}
Complejidad : $O(1)$ Justificacion: En el peor caso realiza dos comparaciones y un solo AgregarRapido

3. Módulo Diccionario Nat Fijo

3.1. Interfaz

se explica con: DICCIONARIO(NAT, α).

géneros: `DiccNat(`α`)`, `itDiccNat(`α`)`.

3.1.1. Operaciones básicas de DiccNat(α)

CREARDICCIONARIO(**in** v: `vector(tupla(clave : nat, significado :` α`)))` $\rightarrow$ res : `DiccNat(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {((∀ t : tupla(nat, α)) esta?(t, v)) $\Rightarrow$ ((definido?(t.clave, res)) $\wedge_L$ obtener(t.clave,res) $=_{obs}$ t.significado) $\wedge$ cantClaves(res) $=_{obs}$ longitud(v)}
Complejidad: $O(copy(\alpha) * n)$ 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

REDEFINIR(**in/out** d: `DiccNat(`α`), in` n `: nat, in` a `:` α)
Pre $\equiv$ {definido?(n,d)}
Post $\equiv$ {obtener(n,d) $=_{obs}$ a}
Complejidad: $O(1)$ en caso promedio, $O(\#claves)$ en peor caso

OBTENER(**in** n: `nat`, **in** d: `DiccNat(`α`)`) $\rightarrow$ res : `puntero(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ obtener(n,d)}
Complejidad: $O(1)$ en caso promedio, $O(\#claves)$ 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

DEFINIDO?(**in** n: `nat`, **in** d: `DiccNat(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ def?(n,d)}
Complejidad: $O(1)$ en caso promedio, $O(\#claves)$ en peor caso
Descripción: Dice si está definida una clave en el diccionario

CANTCLAVES(**in** d: `DiccNat(`α`)`) $\rightarrow$ res : `Nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ $\#claves(d)$}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de claves definidas en el diccionario.

CREARITCLAVES(**in** d: `DiccNat(`α`)`) $\rightarrow$ res : `itConj(nat)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), c)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $O(1)$
Descripción: crea un iterador bidireccional del conjunto, de forma tal que HayAnterior evalúe a false (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente). Luego, se pueden utilizar todas las funciones del iterador de conjunto sobre res.
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 c sin utilizar las funciones del iterador. Esto funciona tal como se indica en la interfaz del iterador de conjunto.

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 : `DiccNat(`α`)`) $\rightarrow$ res : `itDiccNat(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), d)) $\wedge$ 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 : `itDiccNat(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ haySiguiente?(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 $\equiv$ {haySiguiente?(it)}
Post $\equiv$ {alias(res $=_{obs}$ Siguiente(it))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente del iterador.
Aliasing: res.significado 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 $\equiv$ {haySiguiente?(it)}
Post $\equiv$ {alias(res $=_{obs}$ Siguiente(it).significado)}
Complejidad: $\Theta(1)$
Descripción: devuelve el significado del elemento siguiente del iterador
Aliasing: res es modificable si y sólo si it es modficable.

Avanzar(**in/out** it : `itDiccNat(`α`)`)
Pre $\equiv$ {it = it$_0$ $\wedge$ haySiguiente?(it)}
Post $\equiv$ {it $=_{obs}$ Avanzar(it$_0$)}
Complejidad: $\Theta(1)$
Descripción: avanza a la posicion siguiente del iterador.

3.1.3. Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD DiccNat(α) extendido

extiende Diccionario(nat,α)

otras operaciones (no exportadas)

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

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

axiomas

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

secuADiccNat(s) $\equiv$ **if** vacia?(s) **then**
vacio
else
definir(prim(s).clave, prim(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)))`)

(I) No existe dos veces el mismo nat n en dos posiciones distintas del vector y ese nat va a estar en la posicion n mod longitud(tabla)

(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.

3.2.1. Invariante de Representación

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
($\forall$ l : lista(tupla(nat, α))) (($\exists$ k : nat) ((e.tabla[k] = l)
$\Rightarrow$ ($\nexists$ q : nat) (k $\neq$ q $\wedge$ e.tabla[q] = l)) $\wedge$ ($\forall$ $t1$, $t2$: tupla(nat, α)) (esta?($t1$,l) $\wedge$ esta?($t2$,l)
$\Rightarrow$ ($\forall$ n,m : nat) ($t1$.clave = n $\wedge$ $t2$.clave = m
$\Rightarrow$ n $\neq$ m $\wedge$ (n mod longitud(e.tabla) = k $\wedge$ m mod longitud(e.tabla) = k))))
$\wedge$
largosDeListas(e.tabla) = longitud(e.tabla)
$\wedge$
($\forall$ l : lista(tupla(nat, α)))(esta?(l, e.tabla)
$\Rightarrow$ ($\forall$ t : tupla(nat, α))(esta?(t, l)
$\Rightarrow$ ($\exists$ p : puntero(α)(&t = p $\wedge$ esta?(p, e.listaIterable)))))
$\wedge$
long(e.tabla) $=_{obs}$ long(e.listaIterable)

largosDeListas : secu(secu(tupla(nat $\times$ α))) $\longrightarrow$ nat

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

3.2.2. Función de Abstracción

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

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

3.3. Representación del iterador de DiccNat

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

3.4. Algoritmos

icrearDiccionario (**in** v: `vector(tupla(clave : nat, significado : α))`) → res: estr
{**Pre** ≡ true}

```
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(α))
    nat : q ← longitud(res.tabla[k])                                 O(longitud(res.tabla[k]))
    AgregarAtras(res.listaIterable, puntero(res.tabla[k][q-1]))      O(q)
    i++                                                              O(1)
end while                                                            O(longitud(v) * (copy(α) + q))
```

Post ≡ $((\forall\ t : \text{tupla(nat, } \alpha))\ \text{esta?}(t, \text{v})) \Rightarrow ((\text{definido?(t.clave, res)}) \wedge_L \text{obtener(t.clave,res)} =_{obs} \text{t.significado}) \wedge \text{cantClaves(res)} =_{obs} \text{longitud(v)}$
Complejidad : $O(longitud(v) * (copy(\alpha) + q))$ donde q es la cantidad de elementos que pueden encontrarse en una misma posicion de la tabla
Justificacion: las veces que se va a realizar la operacion de copia del elemento α es equivalente al largo del vector, ya que va a hacer una vez cada una. El valor de q va a ser 1 en caso promedio gracias a la funcion de Hash, pero en el peor caso va a ser igual a la longitud del vector de entrada.

iredefinir (**in/out** d: `estr`, **in** n: `nat`, **in** a: α)
{**Pre** ≡ definido?(n,d)}

```
nat : k ← n mod longitud(d)                                          O(1)
itLista(α) : it ← crearIt(d.tabla[k])                                O(1)
while haySiguiente?(it) do                                           O(1)
    IF siguiente(it).clave = n THEN                                  O(1)
        siguiente(it).significado ← a                                O(1)
    FI
    avanzar(it)                                                      O(1)
end while                                                            O(longitud(d.tabla[k]))
```

{**Post** ≡ obtener(n,d) $=_{obs}$ a}
Complejidad : $O(1)$ en caso promedio, $O(\#claves(d))$ en peor caso
Justificacion: En el peor caso, van a estar todos los elementos en la misma posicion de la tabla y el elemento que queremos redefinir va a estar en la ultima posicion. Ademas, cada acceso a una posicion de una lista, es O(i) siendo i la posicion y vamos a recorrer todos los elementos de la tabla (o sea de la lista) hasta encontrar el que queremos. Entonces, en el peor caso, i va a ser igual al numero de claves del diccionario, pero en caso promedio, i va a ser 1 gracias a la funcion de hash.

iobtener (**in** n: `nat`, **in** d: `estr`) → res: puntero(α)
{**Post** ≡ true}

```
nat : k ← n mod longitud(d)                                          O(1)
itLista(α) : it ← crearIt(d.tabla[k])                                O(1)
res ← NULL                                                           O(1)
while haySiguiente?(it) do                                           O(1)
    IF Siguiente(it).clave = n THEN                                  O(1)
        res ← Siguiente(it).significado                              O(1)
    FI
    Avanzar(it)                                                      O(1)
end for                                                              O(i)
```

{**Post** ≡ res $=_{obs}$ obtener(n,d)}
Complejidad : $O(1)$ en caso promedio, $O(\#claves(d))$ en peor caso
Justificacion: Misma justificacion que para redefinir(d,n,a)

idefinido? (**in** n: `nat`, **in** d: `estr`) → res: bool
{**Pre** ≡ true}
```
    nat : k ← n mod longitud(d)                                  O(1)
    itLista(α) : it ← crearIt(d.tabla[k])                        O(1)
    res ← false                                                  O(1)
    while haySiguiente?(it) do                                   O(1)
        IF Siguiente(it).clave = n THEN                          O(i)
            res ← true                                           O(1)
        FI
        Avanzar(it)                                              O(1)
    end while                                                    O(i)
```
{**Post** ≡ res $=_{obs}$ def?(n,d)}
Complejidad : $O(1)$ en caso promedio, $O(\#claves(d))$ en peor caso
Justificacion: Misma justificacion que para redefinir(d,n,a)

icantClaves (**in** d: `estr`) → res: nat
{**Pre** ≡ true}
```
    res ← longitud(d.tabla)                                      O(1)
```
{**Post** ≡ res $=_{obs}$ #claves(d)}
Complejidad : $O(1)$
Justificacion: Saber la longitud de un vector es O(1) y por el InvRep, sabemos que el largo del vector representa la cantidad de claves del dicc.

iCrearIt (**in** d: `DiccNat(`α`)`) → res: itDiccNat(α)
{**Pre** ≡ true}
```
    res ← crearIt(d.listaIterable)                               O(1)
```
{**Post** ≡ alias(esPermutacion(SecuSuby(res), d)) ∧ vacia?(Anteriores(res))}
Complejidad : $\Theta(1)$
Justificacion: De acuerdo a la interfaz de Lista, crear un itLista(α) es $\Theta(1)$.

iHaySiguiente (**in** it: `itDiccNat(`α`)`) → res: bool
{**Pre** ≡ true}
```
    res ← HaySiguiente(it)
```
{**Pre** ≡ res $=_{obs}$ haySiguiente?(it)}
Complejidad : $\Theta(1)$
Justificacion: De acuerdo a la interfaz de Lista, HaySiguiente(it) es $\Theta(1)$.

iSiguiente (**in** it: `itDiccNat(`α`)`) → res: tupla(nat,α)
{**Pre** ≡ HaySiguiente?(it)}
```
    res ← Siguiente(it)
```
{**Pre** ≡ alias(res $=_{obs}$ Siguiente?(it))}
Complejidad : $\Theta(1)$
Justificacion: De acuerdo a la interfaz de Lista, Siguiente(it) es $\Theta(1)$.

iSiguienteSignificado (**in** it: `itDiccNat(`α`)`) → res: α
{**Pre** ≡ HaySiguiente?(it)}
```
    res ← Siguiente(it)
```
{**Pre** ≡ alias(res $=_{obs}$ Siguiente(it).significado)}
Complejidad : $\Theta(1)$
Justificacion: De acuerdo a la interfaz de Lista, Siguiente(it) es $\Theta(1)$.

iAvanzar (**in**/**out** *it*: `itDiccNat(`α`)`)
{**Pre** $\equiv$ it $=_{obs}$ it_0 $\wedge$ res $=_{obs}$ haySiguiente?(it)}
    res $\leftarrow$ Avanzar(it)
{**Pre** $\equiv$ it $=_{obs}$ Avanzar?(it_0)}
Complejidad : $\Theta(1)$
Justificacion: De acuerdo a la interfaz de Lista, Avanzar(it) es $\Theta(1)$.

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: α) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ s}
Complejidad: $O(copy(s))$
Descripción: funcion 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 $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ vacío()}
Complejidad: $O(1)$ Justificación: Sólo crea un arreglo de 27 posiciones inicializadas con null y una lista vacía
Descripción: Crea un diccionario vacío.

DEFINIDO?(**in** d: `diccString`(α), **in** c: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ def?(d, c)}
Complejidad: $O(|n_m|)$ Justificación: Debe acceder a la clave c, recorriendo una por una las partes de la clave (caracteres)
Descripción: Devuelve true si la clave está definida en el diccionario y false en caso contrario.

DEFINIR(**in/out** d: `diccString`(α), **in** c: `string`, **in** s: α)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(c, s, d_0)}
Complejidad: $O(|n_m| + copy(s))$ Justificación: Debe definir la clave c, recorriendo una por una las partes de la clave y después copiar el contenido del significado.
Descripción: Define la clave c con el significado s
Aliasing: Almacena una copia de s.

OBTENER(**in** d: `diccString`(α), **in** c: `string`) $\rightarrow$ res : α
Pre $\equiv$ {def?(c, d)}
Post $\equiv$ {alias($res =_{\text{obs}}$ obtener(c, d))}
Complejidad: $O(|n_m|)$ Justificación: Debe acceder a la clave c, recorriendo una por una las partes de la clave (caracteres)
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(`α`)`, **in** c: `string`)

Pre $\equiv \{d =_{\text{obs}} d_0 \wedge \text{def?}(d, c)\}$
Post $\equiv \{d =_{\text{obs}} \text{borrar}(d_0, c)\}$
Complejidad: $O(|n_m|)$ Justificación: Debe acceder a la clave c, recorriendo una por una las partes de la clave (caracteres) e invalidar su significado
Descripción: Borra la clave c del diccionario y su significado.

CrearItClaves(**in** d: `diccString(`α`)`) $\rightarrow$ res : `itConj(String)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), c)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $O(1)$
Descripción: Crea un Iterador de Conjunto en base a la interfaz del iterador de Conjunto Lineal

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 listaIterable por lo que sus operaciones son identicas a ella.

CrearIt(**in** d: `diccString(`α`)`) $\rightarrow$ res : `itDiccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), d)) $\wedge$ 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: `itDiccString(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ haySiguiente?(it)}
Complejidad: $\Theta(1)$
Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

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

SiguienteSignificado(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {haySiguiente?(it)}
Post $\equiv$ {alias(res $=_{obs}$ haySiguiente?(it).significado)}
Complejidad: $\Theta(1)$
Descripción: devuelve el significado del elemento siguiente del iterador
Aliasing: res es modificable si y sólo si it es modficable.

AnteriorSignificado(**in** it: `itDiccString(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {hayAnterior?(it)}
Post $\equiv$ {alias(res $=_{obs}$ haySiguiente?(it).significado)}
Complejidad: $\Theta(1)$
Descripción: devuelve el significado del elemento anterior del iterador
Aliasing: res es modificable si y sólo si it es modficable.

AVANZAR(**in/out** *it* : `itDiccString(`α`)`)
Pre $\equiv$ {it = $\text{it}_0 \wedge$ haySiguiente?(it)}
Post $\equiv$ {it $=_{obs}$ avanzar(it_0)}
Complejidad: $\Theta(1)$
Descripción: avanza a la posicion siguiente del iterador.

RETROCEDER(**in/out** *it* : `itDiccString(`α`)`)
Pre $\equiv$ {it = $\text{it}_0 \wedge$ hayAnterior?(it)}
Post $\equiv$ {it $=_{obs}$ hayAnterior?(it_0)}
Complejidad: $\Theta(1)$
Descripción: retrocede a la posicion 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.

(∀c: diccString((α)))()

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
longitud(e.raiz)==27 $\wedge_{\text{L}}$
(∀i $\in$ [0..longitud(e.raiz)))
(((¬ e.raiz[i] == NULL) $\Rightarrow_{\text{L}}$ nodoValido(raiz[i])) $\wedge$ (∗e.raiz[i].infoValida == true $\Rightarrow_{\text{L}}$
iteradorValido(raiz[i]))) $\wedge$
listaValida(e.listaIterable)

nodoValido : puntero(Nodo) *nodo* $\longrightarrow$ bool

iteradorValido : puntero(Nodo) *nodo* $\longrightarrow$ bool

nodoValido(*nodo*) $\equiv$ longitud(*nodo.arbolTrie) == 27 $\wedge_{\text{L}}$
(∀i $\in$ [0..longitud(*nodo.arbolTrie)))
((¬ *nodo.arbolTrie[i] == NULL) $\Rightarrow_{\text{L}}$ nodoValido(*nodo.arbolTrie[i]))

iteradorValido(*nodo*) $\equiv$ PunteroValido(nodo) $\wedge_{\text{L}}$
(∀i $\in$ [0..longitud(*nodo.arbolTrie)))
((*nodo.arbolTrie[i].infoValida == true) $\Rightarrow_{\text{L}}$ 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

$$\text{Abs} : \text{estr } e \longrightarrow \text{diccString}(\alpha) \quad \{\text{Rep}(e)\}$$

$$\begin{aligned}\text{Abs}(e) =_{\text{obs}} \text{d: diccString}(\alpha) \mid\ & (\forall s\text{: string})(\text{def?}(d,\ s) =_{\text{obs}} \\ & \text{Definido?(d,s)} \ \wedge \\ & \text{def?}(d,\ s) \Rightarrow_{\text{L}} \text{obtener}(s,d) =_{\text{obs}} \\ & \text{Obtener}(d,s) \\ &)\end{aligned}$$

4.2. Algoritmos

iCrearDiccionario () → res: estr
{**Pre** ≡ true}

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

{**Post** ≡ res $=_{obs}$ vacio()}
Complejidad : $O(1)$ Justificación: Crea un arreglo de 27 posiciones y lo recorre inicializándolo en NULL. Luego crea una lista vacía

iDefinido? (**in** d: `estr`), **in** c: `string`) → res: bool
{**Pre** ≡ true}

```
nat: i ← 0                                                  O(1)
nat: letra ← ord(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 ← ord(c[i])                                       O(1)
    arr ← (*arr).arbolTrie[letra]                           O(1)
end while                                                   O(|n_m|)
if(i=longitud(c)) then                                      O(1)
    res ← (*arr).infoValida                                 O(1)
else
    res ← false                                             O(1)
end if
```

{**Post** ≡ res $=_{obs}$ def?(d,c)}
Complejidad : $O(|n_m|)$ Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego itera sobre los caracteres restantes hasta el final del String c, por lo que hace $|n_m|$ operaciones. Finalmente pregunta si el significado encontrado es válido o no

iDefinir (**in/out** d: `estr`, **in** c: `string`, **in** s: α)
{**Pre** ≡ d $=_{obs}$ d_0}

```
nat: i ← 0                                                  O(1)
```

```
nat: letra ← ord(c[0])                                                    O(1)
if (d.raiz[letra] = NULL) then                                            O(1)
    Nodo: nuevo                                                           O(1)
    arreglo(puntero(Nodo)): nuevo.arbolTrie ← CrearArrelgo(27)            O(1)
    nuevo.infoValida ← false                                              O(1)
    d.raiz[letra] ← puntero(nuevo)                                        O(1)
end if
puntero(Nodo): arr ← d.raiz[letra]                                        O(1)
while(i < longitud(c)) do                                                 O(1)
    i ← i + 1                                                             O(1)
    letra ← ord(c[i])                                                     O(1)
    if (arr.arbolTrie[letra] = NULL) then                                 O(1)
        Nodo: nuevoHijo                                                   O(1)
        arreglo(puntero(Nodo)): nuevoHijo.arbolTrie ← CrearArrelgo(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                                                                 O(|n_m|)
(*arr).info ← s                                                           O(copy(s))
if(not (*arr).infoValida = true) then                                     O(1)
    itLista(puntero(Nodo)) it ← AgregarAdelante(d.listaIterable,NULL)     O(1)
    (*arr).infoValida ← true                                              O(1)
    (*arr).infoEnLisa ← it                                                O(1)
    siguiente(it) ← puntero(*arr)                                         O(1)
end if
```

{**Post** ≡ d $=_{obs}$ definir(c,s,d_0)}
Complejidad : $O(|n_m| + copy(s))$ Justificación: Itera sobre la cantidad de caracteres del String c y en caso de que algún caracter no esté definido crea un arrelglo de 27 posiciones, por lo que realiza $|n_m|$ operaciones. Luego copia el significado pasado por parámetro en O(copy(s)) y finalmente agrega en la lista un puntero al nodo creado

iObtener (**in** d: `estr`, **in** c: `string`) → res: α
{**Pre** ≡ def?(c,d)}

```
nat: i ← 0                                                                O(1)
nat: letra ← ord(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 ← ord(c[i])                                                     O(1)
    arr ← (*arr).arbolTrie[letra]                                         O(1)
end while                                                                 O(|n_m|)
res ← (*arr).info                                                         O(1)
```

{**Post** ≡ alias(res $=_{obs}$ obtener(c,d)}
Complejidad : $O(|n_m|)$ Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego itera sobre los caracteres restantes hasta el final del String c, por lo que hace $|n_m|$ operaciones. Finalmente retorna el significado almacenado. Todas las demás operaciones se realizan en O(1) porque son comparaciones o asignaciones de valores enteros o de punteros

iEliminar (**in/out** d: `estr`, **in** c: `string`)
{**Pre** ≡ d $=_{obs}$ d_0 ∧ def?(d,c)}

```
nat: i ← 0                                                                  O(1)
nat: letra ← ord(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 ← ord(c[i])                                                       O(1)
    arr ← (*arr).arbolTrie[letra]                                           O(1)
    Apilar(pil,arr)                                                         O(1)
end while                                                                   O(|n_m|)
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)                                                      O(1)
        i ← i + 1                                                           O(1)
    end while                                                               O(|n_m|)
    if(i = longitud(c)) then                                                O(1)
        d.raiz[ord(c[0])] ← NULL                                            O(1)
    end if
end if
```

{**Post** ≡ d $=_{obs}$ borrar(d_0,c)}
Complejidad : $O(|n_m|)$ Justificación: Toma el primer caracter y encuentra su posición en el arreglo raíz. Luego crea una pila en O(1). Recorre el resto de los caracteres del String c y apila cada uno de los Nodos encontrado en la pila (0(1)) por lo que en total realiza $|n_m|$ operaciones. Llama a la función tieneHermanos y le pasa por parámetro el nodo encontrado O(1) (ver Algoritmo "tieneHermanos"). Luego recorre todos los elementos apilados preguntando si hay alguno que no tiene hermanos para en cuyo caso eliminarlo, realizando en el peor caso $|n_m|$ operaciones porque puede ser que sea necesario eliminar todo hasta la raiz.

tieneHermanos (**in** *nodo*: `puntero(Nodo)`) → res: bool
{**Pre** ≡ nodo!=NULL}

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

{**Post** ≡ res $=_{obs}$ ($\exists i \in [0..$ longitud(*nodo.arbolTrie)) (*nodo.arbolTrie[i] != NULL))}
Complejidad : $O(1)$ Justificación: Recorre el arreglo de 27 posiciones en caso de que todas las posiciones del mismo tengan NULL. Como es una constante ya que en el peor caso siempre recorre a lo sumo 27 posiciones entonces es O(1)

tieneHermanosEInfo (**in** *nodo*: `puntero(Nodo)`) → res: bool
{**Pre** ≡ nodo!=NULL}

```
res ← tieneHermanos(nodo) and (*nodo).infoValida = true                    O(1)
```

{**Post** ≡ res $=_{obs}$
(∃*i* ∈ [0..longitud(*nodo.arbolTrie))
(*nodo.arbolTrie[i] != NULL)) ∧ (*nodo).infoValida = true}
Complejidad : $O(1)$ Justificación: Llama a la funcion tieneHermanos que es O(1) y verifica además que el nodo contenga información válida