DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP Diseño

Exorcismo Extremo

05 de Junio de 2019 — Algoritmos y Estructuras de Datos II

Grupo 3

Integrante	LU	Correo electrónico
Gianatiempo, Octavio	280/10	ogianatiempo@gmail.com
Gómez, Bruno	428/18	brunolml99@outlook.es
Tropea, Tomás	115/18	tomastropea@hotmail.com
Zylber, Julián	21/18	jzylber@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Exorcismo Extremo

Interfaz

se explica con: JUEGO

géneros: `Juego`.

TP DiseñoOperaciones básicas de Exorcismo Extremo

NUEVOJUEGO(**in** *habitacion* : `Habitacion`,**in** *nombreJ* : `conj(string)`, **in** *fant*: `lista(Evento)`) $\rightarrow$ *res* : `Juego`
Pre $\equiv$ $\{\#jug > 0 \wedge long(fantasma) > 0\}$
Post $\equiv$ $\{(\forall j : jugador)$
$(j \in nombreJ \Rightarrow_{\text{L}} j \in jugadores(res) \wedge posJugador(j, res) = \Pi_0(obtener(j, localizarJugadores(res))) \wedge$
$(\#fantasmas(res) = 1 \wedge (\exists f : fantasma)(f \in fantasma(juego) \wedge_{\text{L}} posFantasma(f, res) = \Pi_0(prim(fant))))\ \}$
Complejidad: $O(localizar\_jugadores + \#nombresJ * max\{long(n) : n \in nombresJ\} + long(fantasma) + setearMapa)$
Descripción: inicia el juego con todos sus jugadores posicionados en la habitacion y el fantasma.

JUGADORES(**in** *juego*: `Juego`) $\rightarrow$ *res* : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{obs} jugadores(juego)\ \}$
Complejidad: $O(1)$
Descripción: esta operacion nos devuelve el conjunto de los nombres de los jugadores.
Aliasing: No aplica.

JUGADORESVIVOS(**in** *juego*: `Juego`) $\rightarrow$ *res* : `lista(Tupla`$\langle$`string, pos , dir`$\rangle$`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{noHayRepetidos(res) \wedge (\forall\ nom, pos, dir)\langle nom, pos, dir\rangle \in res \leftrightarrow \exists nom \in jugadores(\overset{\wedge}{juego}) \wedge_L jugadorVivo(nom, \overset{\wedge}{juego}) \wedge pos = posJugador(nom, \overset{\wedge}{juego}) \wedge dir = dirJugador(nom, \overset{\wedge}{juego})\}$
Complejidad: $O(1)$
Descripción: Devuelve jugadores vivos. Esta es la operacion que obedece con la complejidad pedida en el enunciado inciso 1.
Aliasing: Se devuelve la lista por referencia inmutable

FANTASMAS(**in** *juego*: `Juego`) $\rightarrow$ *res* : `lista(fantasma)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = fantasmas(juego)\}$
Complejidad: O(longitud(fantasmas)*max{long(l) : l $\in$ juego.accionF})
Descripción: Nos permite conocer la posicion y direccion actual de todos los fantasmas vivos.
Aliasing: No aplica.

FANTASMASVIVOS(**in** *juego*: `Juego`) $\rightarrow$ *res* : `lista(Tupla`$\langle$`id,pos , dir`$\rangle$`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{noHayRepetidos(res) \wedge (\forall f \in fantasmas(\overset{\wedge}{juego}))\ fantasmaVivo(\overset{\wedge}{juego}, f) \leftrightarrow (\exists\langle pos, dir\rangle \in res) \wedge_L posFantasma(f, \overset{\wedge}{juego}) = pos \wedge dirFantasma(f, \overset{\wedge}{juego}) = dir\}$
Complejidad: $O(1)$
Descripción: Posición y dirección de fantasmas vivos. Esta es la operacion que obedece con la complejidad pedida en el enunciado inciso 2.
Aliasing: Se devuelve la lista por referencia inmutable

FANTASMAESPECIAL(**in** *juego*: `Juego`) $\rightarrow$ *res* : `fantasma`
Pre $\equiv$ $\{True\}$
Post $\equiv$ {res = fantasmaEspecial(juego)}
Complejidad: $O(1)$
Descripción: nos devuelve el fantasma "Especial" de la Ronda actual.
Aliasing: Devuelve una referencia inmutable al fantasmaEspecial que tenemos en el final de la lista de fantasmas-Vivos. Esta es la operacion que obedece con la complejidad pedida en el enunciado inciso 3.

FantasmasQueDispararon(**in** *juego*: `Juego`) → *res* : `lista(fantasma)`
Pre ≡ {true}
Post ≡ $\{(\forall f : fantasma)(esta?(f, res) \Rightarrow_{\text{L}} fantasmaVivo(f, juego) \wedge \Pi_2(recorrer(f, step(juego) - 1)) = true)\}$
Complejidad: $O(long(juego.fantasmasV))$
Descripción: Nos permite conocer la posicion y direccion de todos los Fantasmas Vivos que dispararon en el ultimo paso del juego. Esta es la operacion que obedece a la complejidad pedida en la consigna del tp, en el inciso 4
Aliasing: Se devuelve la lista por referencia inmutable

EstaVivo?(**in** *juego*: `Juego`,**in** *jug*: `string`) → *res* : `bool`
Pre ≡ {true}
Post ≡ $\{res = true \iff jugadorVivo(jug, juego)\}$
Complejidad: $O(|jug|)$
Descripción: Nos permite preguntar por la muerte de un jugador. Esta es la operacion que obedece a la complejidad pedida en la consigna del tp, en el inciso 5.
Aliasing: No aplica.

Step(**in/out** *jug*: `String`, **in/out** *accion*: `Accion`, **in/out** *juego*: `estr`)
Pre ≡ $\{juego =_{obs} JUEGO_0 \wedge jug \in jugadores(juego) \wedge \neg esPasar(accion)\}$
Post ≡ $\{\text{juego} =_{obs} step(jug, accion, JUEGO_0)\}$
Complejidad:$O(|jug| + long(fantasmaV) * m + long(jugadoresV))$
Descripción: actualiza el turno en base a la accion de un jugador, se actualiza dicho jugador, los otros ejecutan la accion esperar y luego actuan todos los fantasmas. Esta es la operacion que obedece a la complejidad pedida en la consigna del tp, en el inciso 6.
Aliasing: No aplica.

Pasar(**in/out** *juego*: `Juego`)
Pre ≡ $\{juego =_{obs} JUEGO_0\}$
Post ≡ $\{\text{juego} =_{obs} pasar(JUEGO_0)\}$
Complejidad:$O(long(juego.fantasmaV) * tamaño(juego.habitacion) + long(jugadoresV))$
Descripción: Actualiza el turno con un Jugador que ejecuta la accion pasar, a su vez en ese mismo turno todos los otros jugadores tambien "pasan" y luego ejecutan todos los fantasmas las acciones siguientes en su lista de acciones correspondiente. Esta es la operacion que obedece a la complejidad pedida en la consigna del tp, en el inciso 7.
Aliasing: No aplica.

PosPorDisparosEnRonda(**in** *juego*: `Juego`) → *res* : `conj(posicion)`
Pre ≡ {true}
Post ≡ $\{(\forall p : pos)(p \in res \Leftrightarrow (\exists f : fantasma)(\exists i : Nat)(f \in fantasmas(juego) \wedge_{\text{L}} i \leq step(juego) \wedge_{\text{L}} \Pi_2(recorrer(f, i)) \wedge_{\text{L}} p \in alcanceDisparo(\Pi_0(recorrer(f, i)), \Pi_1(recorrer(f, i)), habitacion(juego))))\}$
Complejidad:$O(long(fantasmaV) * tamaño(juego.habitacion) + long(jugadoresV))$
Descripción: Devuelve un conjunto de posiciones, las cuales fueron atravezadas por un disparo de fantasma, durante la UltimaRonda. Esta es la operacion que obedece a la complejidad pedida en la consigna del tp, en el inciso 8.
Aliasing: No aplica.

Habitacion(**in** *juego*: `Juego`) → *res* : `Habitacion`
Pre ≡ {True}
Post ≡ $\{res =_{obs} habitacion(juego)\}$
Complejidad: $O(1)$
Descripción: Devolvemos una refencia inmutable a la habitacion que tenemos en nuestra estructura.

acciones(**in** *jug*: `String`,**in** *juego*: `Juego`) → *res* : `lista(Evento)`
Pre ≡ {true}
Post ≡ $\{res =_{obs} acciones(jug, juego)\}$
Complejidad: $O(1)$
Descripción: Devolvemos la lista de acciones asociada al jugador jug.
Aliasing: Se devuelve la lista por referencia inmutable

Fórmulas Lógicas Auxiliares

noHayRepetidos(lista(α)) $\equiv (\forall i_1, i_2 : Nat)(i_1 < long(lista(\alpha)) \wedge i_2 < long(lista(\alpha)) \wedge i_1 \neq i_2 \Rightarrow_{\text{L}} lista(\alpha)[i_1] \neq lista(\alpha)[i_2])$

Representación

TP DiseñoRepresentación de Exorcismo Extremo El Modulo Exorcismo Extremo va a estar representado, por la siguiente estructura que detallamos.

`Exorcismo Extremo` **se representa con** `estr`

donde `estr` es `tupla`(*Habitacion*: `habitacion`
, *jugadoresPorNombre*: `dicc(nombre : string , puntero(dataJ))`
, *jugadoresV*: `lista(puntero(dataJ))`
, *jugadores*: `arregloDimensionable(dataJ)`
, *fantasmaV*: `lista(puntero(dataF))`
, *fantasmas*: `lista(dataF)`
, *accionesF*: `lista (lista (evento))`
, *accionesJ*: `arregloDimensionable` ⟨`lista(evento)`⟩)
, *jugadoresVivosObs*: `lista(Tupla`⟨`nombre , pos , dir`⟩)
, *fantasmasVivosObs*: `lista(Tupla`⟨`pos , dir`⟩)
)

donde `dataJ` es `tupla`(*nombre*: `string` , *pos*: `pos` , *dir*: `dir` , *vivo*: `bool` , *accionesJ*: `puntero(lista(evento)` , *jugadorObs*: `itLista(Tupla`⟨`nombre , pos , dir`⟩))

donde `dataF` es `tupla`(*id*: `Nat` , *pos*: `pos` , *dir*: `dir` , *accionActual*: `itLista(lista(evento))` , *accionInicial*: `itLista(lista(evento))` , *accionFinal*: `itLista(lista(evento))`)

donde `evento` es `tupla`(*posicion*: `pos` , *direccion*: `dir` , *dispara*: `bool`)

donde `pos` es `tupla`(x: `Nat` , y: `Nat`)

donde **dir** es un tipo enumerado basado en string que sólo puede tomar los valores Norte, Sur, Este y Oeste

Invariante de Representación

- El dicc "jugadoresPorNombre" contiene en todas sus claves los nombres de todos los jugadores del juego, y tiene la misma cantidad de claves que el tamaño del arreglo dimensionable llamado "jugadores"
- jugadoresPorNombre tiene en su significado un puntero a dataJ que esta contenido en jugadores y ademas en la primera posicion de la tupla dataJ posee el mismo string que el nombre que utilizamos como clave.
- jugadoresV es una lista de punteros a dataJ, de tamaño menor o igual al tamaño del arreglo jugadores, y cada dataJ al que apunta esta contenido jugadores y ese jugador está vivo.Ademas no hay punteros que apunten al mismo
- jugadoresV durante el transcurso del juego debe tener siempre como minimo un jugador, en el que caso de que quede vacia luego de determinado turno eso solo sucede cuando el juego termina
- jugadores es un arreglo dimensionable de dataJ siendo la tupla que contiene la informacion necesaria para distinguir a un jugador de otro, es decir, su nombre, posicion actual, direccion, un valor booleano para indicarnos si dicho jugador esta vivo y ademas un puntero a una lista de eventos perteneciente al arreglo dimensionable accionesJ el cual contiene para cada jugador su lista de acciones correspondiente. DataJ tambièn contiene dataJ.jugadorObs que es un iterador a jugadoresVivosObs. La información de la tupla a la que apunta coincide con la informaciòn de dataJ de ese jugador.
- jugadores no contiene tuplas dataJ repetidas, es decir no hay dos tuplas que contengan el mismo nombre o su punto de acciones apunte al mismo. Además todas las tuplas son apuntadas por una entrada de jugadoresPorNombre cuya clave coincide con el string y si el jugadore està vivo es también apuntado por un nodo de la lista jugadoresV.
- fantasmasV es una lista de punteros a dataF contenidos en fantasmas, no hay dos de los punteros contenidos en fantasmasV que apunten a la misma tupla en fantasmas y ademas su tamaño siempre es menor o igual al tamaño de la lista fantasmas. el fantasma que representa está vivo

- mientras ocurre una ronda la lista de fantasmasV debe tener al menos al fantasmaEspecial, el cual, siempre esta posicionado al final de la lista en caso de que dicho fantasma sea extraido de la lista fantasmasV termina la ronda, pues se supone que fue asesinado.
- fantasmas es una lista de dataF, es decir de tuplas que contienen la informacion necesaria para distinguir un fantasma de otro, su posicion actual, la direccion en la que esta mirando, la accion actual, que seria la que esta ejecutando en el momento la cual es representada con un iterador a la lista asociada de eventos y ademas con un iterador adiccional que señala a la lista de evento que tiene asociada dicho fantasma. También tiene un iterador que apunta al final de la lista de acciones. Si está vivo, existe un puntero en fantasmasV que apunta a esta data.
- fantasmas no contiene dataF repetidas y ademas su tamaño es igual al numero de la ronda que se esta ejecutando
- cada elemento de la lista de fantasmas tiene como propiedad que el iterador a la lista de acciones asociada a cada fantasma, nos devuelve una lista de eventos la cual fue ejecutada o llevada a cabo por el jugador que mato a dicho fantasma en una ronda determinada (menos el primero cuyas acciones se obtuvieron externamente)
- accionesF es uns lista de listas de eventos donde cada lista esta asociada a un fantasma, es decir la cantidad de listas que contiene accionesF es igual a la longitud de la lista fantasmas y ademas no hay dos listas que esten asociadas a mismo fantasma.
- cada lista en accionesF fue ejecutada durante una ronda entera por un jugador y termina en el momento en el que el jugador mata al fantasma al que le es asociada dicha lista eventos, por lo tanto, cada lista tiene al menos un evento (nuevamente, el primero no cuenta)
- accionesJ es un arreglo dimensionable del mismo tamaño que el arreglo jugadores, donde cada lista esta asociada a un jugador y ademas se van llenando a medida que un jugador ejecuta acciones.
- AccionesJ no tiene listas que esten asociadas a mismo jugador, ademas cuando empieza una ronda todas las listas son vacias, y antes de terminar la ronda al menos existe una de esas que contenga un evento
- jugadoresVivosObs es una lista de Tuplas que contienen nombre, posicion y direccion de manera que contiene a todos los jugadores que estan vivos en esta ronda, la finalidad de dicha estructura aunque redundante es que nos asegura que cuando devolvemos los datos en dicha estructura nos asegura que no estariamos revelando demas
- jugadoresVivosObs es una lista que no tiene tuplas repetidas y tiene una longitud menor o igual a la longitud del arreglo de jugadores
- fantasmasVivosObs es una lista de tuplas con contiene la posicion y direccion de cada fantasmaVivo, al igual que jugadoresVivosObs esta estructura nos asegura que cuando nos soliciten la informacion pedida de los FantasmasVivos no revelariamos informacion demas para los usuarios de dicha interfaz
- No existen dos jugadores que tienen un iterador a la misma tupla de jugadoresVivosObs

Función de Abstracción

Una estructura nuestra es igual al TAD si:

- la habitacion que contiene el juego es igual a la Habitacion de nuestra estructura
- El conjunto de fantasmas que nos proporciona el juego en nuestra estructura es igual a los fantasmas del tad juego
- IfantasmaEspecial es el fantasma especial del TAD
- Ijugadores es igual a jugadores del TAD
- Iacciones es igual a las acciones para cada jugador en jugadores

Algoritmos

iPasar(**in/out** *juego*: `estr`)

1: accionarDemasJugadoresYFantasmas(true,0,juego)

Complejidad:O(longitud(juego.jugadoresV)+longitud(juego.fantasmasV) * ancho(juego.mapa))

Justificación:

1)La complejidad esta dada por la función "accionarDemasJugadoresYFantasmas" la cual va a ejecutar un turno suponiendo que todos los jugadores vivos esperan y los fantasmas vivos ejecutan sus respectivas acciones, generando la complejidad equivalente a la suma de jugadores vivos y a la de fantasmas vivos por el ancho del mapa , esto ultimo sera en el peor cas suponiendo que todos disparan

---

iStep(**in/out** *nombreJ*: `String`,**in/out** *accion*: `Accion`,**in/out** *juego*: `estr`)

```
1: punteroJugador ← jugadoresPorNombre[nombreJ]                                   ▷ O(|nombreJ|)
2: if esMover(accion) then                                                        ▷ O(1)
3:     if EsMovValido(juego.habitacion,punteroJugador→pos,accion.dir) then        ▷ O(1)
4:         mover(true,punteroJugador→pos,accion.dir,juego.habitacion)             ▷ O(1)
5:         punteroJugador→pos ← punteroJugador→ady(juego.habitacion,punteroJugador→pos,accion.dir)  ▷ O(1)
6:         punteroJugador→dir ← accion.dir                                        ▷ O(1)
7:         punteroJugador→acciones→agregarAtras(⟨ punteroJugador→pos,punteroJugador→dir,false⟩)     ▷ O(1)
8:     else
9:         punteroJugador→dir ← accion.dir                                        ▷ O(1)
10:        punteroJugador→acciones→agregarAtras(⟨ punteroJugador→pos,punteroJugador→dir,false⟩)     ▷ O(1)
11:    end if
12: else
13:    if esDisparar(accion) then                                                 ▷ O(1)
14:        disparar(juego.habitacion,true,punteroJugador→pos,it→dir)              ▷ O(ancho(juego.mapa))
15:        itFanV ← creatIt(fantasmasV)                                           ▷ O(1)
16:        while HaySiguiente(itFanV) do                                          ▷ O(1)
17:            if !estaVivo(juego.habitacion,false,siguiente(itFanV).pos) then    ▷ O(1)
18:                if Siguiente(itFanV).id == (longitud(juego.fantasmas)-1) then  ▷ O(1)
19:                    siguienteRonda(punteroJugador,juego)                       ▷ O(siguienteRonda)
20:                    return
21:                end if
22:                eliminarSiguiente(itFanV)                                      ▷ O(1)
23:            end if
24:        end while
25:        (punteroJugador→acciones)→agregarAtras(⟨ punteroJugador→pos,punteroJugador→dir,true⟩)     ▷ O(1)
26:    else
27:        (punteroJugador→acciones)→agregarAtras(⟨ punteroJugador→pos,punteroJugador→dir,false⟩)     ▷ O(1)
28:    end if
29: end if
30: accionarDemasJugadoresYFantasmas(false,punteroJugador→nombre,juego)           ▷
```

O(longitud(juego.jugadoresV)+longitud(juego.fantasmasV)* ancho(juego.habitacion))

Complejidad: $O(|nombreJ|$ + longitud(juego.jugadoresV)+longitud(juego.fantasmasV) * ancho(juego.mapa))

Justificación:

1)Se busca al jugador que va a realizar la accion,para ello se lo busca en el diccionario por su nombre, teniendo una complejidad equivalente al largo de su nombre

2)Luego se realiza la accion correspondiente para el jugador, la cual puede ser esperar ,mover o disparar, y en el peor de los casos sera disparar y este producira una linea de disparo igual al largo del mapa, lo cual generaria una complejidad equivalente al largo del mapa, pero esto es un multiplo de uno de los factores en la complejidad que aparece mas adelante en "AccionarDemasJugadoresYFantasmas" entonces no aparece como un factor aparte

3)Si se llegara a resetear la ronda, en lugar de sumarse la complejidad de "accionarDemasJugadoresYFantasmas" se le suma la complejidad de "siguienteRonda".

AccionarDemasJugadoresYFantasmas(**in** *pasarJugadores* : `Bool`,**in** *nombreJ* : `String`,**in/out** *juego* : `estr`)

```
1: for all puntero(jugadores) p : jugadoresV do
2:     if  p→nombre != nombreJ ∨ pasarJugadores == true then                          ▷ O(1)
3:         p→acciones→agregarAtras(⟨ p→pos,p→dir,false⟩)                              ▷ O(1)
4:     end if
5: end for
6: fantasmasVivosObs ← vacio()
7: for all punteroF : juego.fantasmasV do
8:     if !HaySiguiente(punteroF→accionActual) then                                   ▷ O(1)
9:         punteroF→accionActual ← punteroF→accionInicial                             ▷ O(1)
10:    end if
11:    accion ← Siguiente(punteroF→accionActual)                                      ▷ O(1)
12:    Avanzar(punteroF→accionActual)                                                 ▷ O(1)
13:    if accion.pos != punteroF→pos then                                             ▷ O(1)
14:        mover(false,punteroF→pos,accion.dir,juego.habitacion)                      ▷ O(1)
15:    else if accion.disparar == true then                                           ▷ O(1)
16:        disparar(false,accion.pos,accion.dir)                                      ▷ O(ancho(juego.mapa))
17:    end if
18:    punteroF→pos ← accion.pos                                                      ▷ O(1)
19:    punteroF→dir ← accion.dir                                                      ▷ O(1)
20:    agregarAtras(fantasmasVivosObs,⟨punteroF→pos,punteroF→dir⟩)                    ▷ O(1)
21: end for
22: itJugV ← creatIt(jugadoresV)                                                      ▷ O(1)
23: while HaySiguiente(itJugV) do
24:    if !estaVivo(juego.habitacion,true,Siguiente(itJugV)→pos) then                 ▷ O(1)
25:        Siguiente(itJugV)→vivo ← false                                             ▷ O(1)
26:        eliminarSiguiente(Siguiente(itJugV)→jugadorObs)                            ▷ O(1)
27:        eliminarSiguiente(itJugV)                                                  ▷ O(1)
28:    else
29:        Siguiente(Siguiente(itJugV)→jugadorObs).dir ← Siguiente(itJugV)→dir        ▷ O(1)
30:        Siguiente(Siguiente(itJugV)→jugadorObs).pos ← Siguiente(itJugV)→pos        ▷ O(1)
31:    end if
32: end while
```

Complejidad: O(longitud(juego.jugadoresV)+longitud(juego.fantasmasV)* ancho(juego.mapa))

Justificación:

1)Se recorren todos los jugadores vivos que no hayan disparado y se les aplica la accion pasar, esto tiene complejidad equivalente a la cantidad de jugadores vivos

2)Por cada uno de los fantasmas vivos se ejecuta su accion,ya sea pasar,mover o disparar, en el peor de los casos todos van a disparar y ademas todos van a disparar en una linea con longitud equivalente al largo del mapa, por lo tanto la complejidad de esto va a ser cantidad de fantasmas vivos por ancho del mapa

siguienteRonda(**in** *punteroJugador* : `puntero(dataJ)`,**in/out** *juego* : `estr`)

1: accionesFantasma ← *(punteroJugador→accion) ▷ $O(1)$
2: i ← 5
3: **while** i > 0 **do**
4: agregarAtras(accionesFantasma,⟨punteroJugador→pos,punteroJugador→dir,false⟩) ▷ $O(1)$
5: i ← i-1 ▷ $O(1)$
6: **end while**
7: accionesInvertidas ← inversa(*(punteroJugador→accion)) ▷ $O(\text{invertir})$
8: itAccionInv ← crearIt(accionesInvertidas) ▷ $O(1)$
9: **while** HaySiguiente(itAccionInv) **do**
10: agregarAtras(accionesFantasma,Siguiente(itAccionInv)) ▷ $O(1)$
11: avanzar(itAccionInv) ▷ $O(1)$
12: **end while**
13: i ← 5
14: **while** i > 0 **do**
15: agregarAtras(accionesFantasma,
⟨primero(*(punteroJugador→accion))→pos,primero(*(punteroJugador→accion))→dir,false⟩) ▷ $O(1)$
16: i ← i-1 ▷ $O(1)$
17: **end while**
18: itFantasmas ← crearIt(juego.fantasmas) ▷ $O(1)$
19: agregarAtras(juego.accionesF,accionesFantasma) ▷ $O(\text{long(accionesFantasma)})$
20: agregarAtras(juego.fantasmas, ⟨longitud(juego.fantasmas),punteroJugador→pos, punteroJugador→dir, crearIt(Ultimo(juego.accionesF)), crearIt(Ultimo(juego.accionesF)), crearItUlt(Ultimo(res.accionesF)) ⟩) ▷ $O(1)$
21: itAccionesJ ← creatIt(juego.accionesJ) ▷ $O(1)$
22: **while** HaySiguiente(itAccionesJ) **do**
23: Siguiente(itAccionesJ) ← vacio() ▷ $O(\text{longitud(juego.jugadores)* (juego.accionesJ[0])})$
24: avanzar(itAccionesJ) ▷ $O(1)$
25: **end while**
26: itFan ← crearIt(juego.fantasmas) ▷ $O(1)$
27: fantasmasVivosObs ← vacio()
28: **while** HaySiguiente(itFan) **do**
29: Siguiente(itFan).accionActual ← Siguiente(itFan).accionInicial ▷ $O(1)$
30: Siguiente(itFan).pos ← Siguiente(Siguiente(itFan).accionActual).pos ▷ $O(1)$
31: Siguiente(itFan).dir ← Siguiente(Siguiente(itFan).accionActual).dir ▷ $O(1)$
32: Avanzar(Siguiente(itFan).accionActual) ▷ $O(1)$
33: agregarAtras(juego.fantasmasVivos,&(Siguiente(itFan))) ▷ $O(1)$
34: agregarAtras(fantasmasVivosObs,⟨Siguiente(itFan).pos,Siguiente(itFan).dir⟩)
35: avanzar(itFan) ▷ $O(1)$
36: **end while**
37: pos_dir ← localizar_jugadores(juego) ▷ $O(\text{localizar\_jugadores})$
38: itJug ← crearIt(juego.jugadores) ▷ $O(1)$
39: jugadoresVivosObs ← vacio() ▷ $O(1)$
40: **while** HaySiguiente(itJug) **do**
41: Siguiente(itJug).vivo ← true ▷ $O(1)$
42: Siguiente(itJug).pos ← pos_dir[Siguiente(itJug).nombre].pos ▷ $O(1)$
43: Siguiente(itJug).dir ← pos_dir[Siguiente(itJug).nombre].dir ▷ $O(1)$
44: agregarAtras(juego.jugadoresVivos,&(Siguiente(itJug))) ▷ $O(1)$
45: agregarAtras(jugadoresVivosObs,⟨Siguiente(itJug).nombre , Siguiente(itJug).pos , Siguiente(itJug).dir⟩) ▷ $O(1)$
46: Siguiente(itJug).jugadorObs ← crearItUltimo(jugadoresVivosObs) ▷ $O(1)$
47: avanzar(itJug) ▷ $O(1)$
48: **end while**
49: setearMapa(juego) ▷ $O(\text{setearMapa})$

Complejidad:$O(\text{longitud(juego.accionesJ[0])+longitud(juego.fantasmas)+longitud(juego.jugadores)* (juego.accionesJ[0])+localizar\_jugadores)+setearMapa)}$

Justificación:

1. Se toman las acciones del jugador que mato al fantasma especial, se les agrega 5 veces la accion esperar y luego se le agregan las mismas acciones invertidas, es decir, la secuencia de acciones invertida y las direcciones de cada una tambien, todo esto termina teniendo como complejidad la cantidad de pasos que paso en la ultima ronda

2. Se cambia el id de cada fantasma por id+1, lo cual termina teniendo como complejidad la cantidad de fantasmas de la ronda, que es el numero de ronda

3. Se resetea la lista de acciones de todos los jugadores, lo cual implica recorrer todos y setear a la lista vacia, eliminando el contenido anterior, el cual era una lista de longitud cantidad de turnos en la ronda anterior, lo cual hace que haya una complejidad de cantidad de jugadores * cantidad de turnos de ronda anterior

4. Finalmente se le suma la complejidad de localizar_jugadores y la de setearMapa,vale notar que se recorren todos los jugadores una ultima vez para setear sus valores iniciales pero eso es proporcional a la cantidad de jugadores, que ya esta contenido en las complejidades anteriores.

---

setearMapa(**in/out** *juego*: `estr`)

```
1:  resetear(juego.mapa)                                              ▷ O(resetear)
2:  itJugadores ← crearIt(juego.jugadores)                            ▷ O(1)
3:  listaPosJugadores ← Vacio()                                       ▷ O(1)
4:  while HaySiguiente(itJugadores) do
5:      agregarAtras(Siguiente(itJugadores).pos,listaPosJugadores)    ▷ O(1)
6:  end while
7:  itFantasmas ← crearIt(juego.fantasmas)                            ▷ O(1)
8:  listaPosFantasmas ← Vacio()                                       ▷ O(1)
9:  while HaySiguiente(itFantasmas) do
10:     agregarAtras(Siguiente(itFantasmas).pos,listaPosFantasmas)    ▷ O(1)
11: end while
12: agregarFantasmas(listaPosFantasmas,juego.mapa)                    ▷ O(agregarFantasmas)
13: agregarJugadores(listaPosJugadores,juego.mapa)                    ▷ O(agregarJugadores)
```

Complejidad:O(resetear + long(jugadores) + long(fantasma) + agregarFantasmas + agregarJugadores)

Justificación:

1. Se agrega la complejidad de resetear el mapa que esta dada por la complejidad que provee el modulo habitacion para esta operacion

2. Se agregan todas las posiciones de los jugadores a un conjunto, el cual puede tener repetidos ya que varios jugadores pueden estar en la misma posicion, entonces se utiliza el agregarRapido , obteniendo una complejidad igual a la cantidad de jugadores

3. Fantasmas pueden estar en la misma posicion, entonces se utiliza el agregarRapido , obteniendo una complejidad igual a la cantidad de fantasmas

4. Finalmente se agregan los fantasmas y los jugadores al mapa, con lo cual se suma la complejidad que dice el modulo habitacion acerca de las funciones agregarFantasmas y agregarJugadores

---

iNuevoJuego(**in** *habitacion* : `Habitacion`,**in** *nombresJ* : `conj(String)`,**in** *fantasma* : `lista(Evento)`) → *res* : *estr*

1: res.jugadores ← vacia() ▷ $O(1)$
2: res.fantasmas ← vacia() ▷ $O(1)$
3: res.jugadoresVivosObs ← vacia() ▷ $O(1)$
4: res.fantasmasVivosObs ← vacia() ▷ $O(1)$
5: res.jugadoresPorNombre ← vacia() ▷ $O(1)$
6: res.accionesF ← vacia() ▷ $O(1)$
7: res.accionesJ ← vacia() ▷ $O(1)$
8: res.jugadoresVivos ← vacia() ▷ $O(1)$
9: res.fantasmasVivos ← vacia() ▷ $O(1)$
10: pos_dir ← localizar_jugadores(*this) ▷ O(localizar_jugadores)
11: **for all** nombre : nombresJ **do**
12: AgregarAtras(res.jugadoresVivosObs,⟨nombre,pos_dir[nombre].pos,pos_dir[nombre].dir⟩)
13: AgregarAtras(res.accionesJ,vacio())
14: AgregarAtras(res.jugadores, ⟨nombre, pos_dir[nombre].pos, pos_dir[nombre].dir, true, &Ultimo(res.accionesJ), crearItUltimo(res.jugadoresVivosObs)⟩) ▷ $O(1)$
15: AgregarAtras(res.jugadoresVivos,&Ultimo(res.jugadores)) ▷ $O(1)$
16: Definir(res.jugadoresPorNombre,nombre,&Ultimo(res.jugadores)) ▷ $O(|nombre|)$
17: **end for**
18: agregarAtras(res.accionesF,fantasma) ▷ O(longitud(fantasma))
19: agregarAtras(res.fantasmas, ⟨0,fantasma[0].pos, fantasma[0].dir, crearIt(Ultimo(res.accionesF)), crearIt(Ultimo(res.accionesF)), crearItUlt(Ultimo(res.accionesF))⟩) ▷ $O(1)$
20: AgregarAtras(res.fantasmasVivosObs,⟨fantasma[0].pos,fantasma[0].dir⟩)
21: agregarAtras(res.fantasmasVivos,&primero(res.fantasmas)) ▷ $O(1)$
22: setearMapa(res) ▷ $O(setearMapa)$

Complejidad:O(localizar_jugadores + #nombresJ * max{long(n) : n ∈ nombresJ} + long(fantasma) + setearMapa)

Justificación:
1)Inicializar todos los contenedores de juego es constante en tiempo porque se esta creando el juego, por lo tanto no tenian datos previos
2)Se suma la complejidad de localizar jugadores
3)Luego por cada nombre en el conjunto de jugadores se lo agrega al diccionario y tambien se agregan sus datos a la lista de jugadores, lo cual termina teniendo como complejidad la cantidad de elementos en el conjunto de jugadores por el nombre mas largo dentro del mismo
4)Se agrega la lista de acciones del primer fantasma a la lista de listas de acciones de fantasmas, lo cual cuesta la longitud de la lista de acciones del primer fantasma
5)Al momento de generar la lista de jugadores vivos se recorre toda esa lista, cuyo largo es el mismo que el de nombresJ, por lo tanto solo se le suma un multiplo de la complejidad de agregar todos los nombres al diccionario(2)
6)Finalmente se suma la complejidad de setearMapa

Ihabitacion(**in/out** *juego* : `estr`) → *res* : *Habitacion*

1: res ← juego.habitacion

Complejidad:$O(1)$

Justificación:Se devuelve una referencia a la habitacion ,por lo tanto la complejidad es constante.

Ifantasmas(**in** *juego* : `estr`) → *res* : *lista*(*lista*(*evento*))

1: res ← juego.accionesF

Complejidad:O(longitud(juego.accionesF)*max{long(l) : l ∈ juego.accionesF})

Justificación:El algoritmo consiste de recorrer la lista que contiene las acciones de cada fantasma y copiar cada una de esas listas de acciones al conjunto de fantasmas, por lo tanto se termino recorriendo la cantidad de fantasmas y por cada uno se copia su lista de acciones, entonces la complejidad sera cantidad de fantasmas por la longitud de la lista mas larga de acciones que exista

IfantasmaEspecial(**in** *juego*: `estr`) → *res* : *lista*(*evento*)

1: res ← Ultimo(juego.accionesF)

Complejidad:$O(1)$

Justificación:Devuelvo una referencia al ultimo elemento de una lista, lo cual es constante.

Ijugadores(**in** *juego*: `estr`) → *res* : *conj*(*string*)

1: res ← iClaves(juego.jugadoresPorNombre)

Complejidad:$O(1)$

Justificación:Como iClaves devuelve una referencia de las claves del diccionario y se devuelve luego la referencia a este ultimo en la funcion Ijugadores, no se produce ninguna copia de datos y todas estas operaciones son constantes.

Iacciones(**in/out** *jugador*: `String` **in/out** *juego*: `estr`) → *res* : *secu*(*Evento*)

1: res ← *juego.jugadoresPorNombre[jugador].accionesJ

Complejidad:$O(1)$

Justificación: Al devolver una referencia a la lista de acciones de un jugador no se realiza una copia de datos por lo que termina siendo constante la complejidad de devolverlas.

IfantasmasVivos(**in** *juego*: `estr`) → *res* : *lista*($Tupla < id, pos, dir >$)

1: res ← juego.fantasmasVivosObs

Complejidad:$O(1)$

Justificación:

IjugadoresVivos(**in** *juego*: `estr`) → *res* : *lista*($Tupla < string, pos, dir >$)

1: res ← juego.jugadoresVivosObs

Complejidad:$O(1)$

Justificación:

IFantasmasQueDispararon(**in** *juego*: `estr`)→ *res* : *lista*(*pos*)

```
1: itfan ← crearIt(juego.fantasmasV)
2: res ← vacia()
3: while HaySiguiente(itfan) do
4:     if ¬(HayAnterior(siguiente(itfan).accionactual)) then
5:         agregarAtrás(res,anterior(siguiente(itfan).accionfinal).pos)
6:     else
7:         if anterior(siguiente(itfan) → accionactual).disparo = True then
8:             agregarAtrás(res,siguiente(itfan).pos)
9:         end if
10:    end if
11: end while
```

Complejidad:$O(longitud(juego.fantasmasV))$

Justificación: la complejidad se justifica con la operacion realizada en el Modulo Habitacion.

2. Acción

Este módulo representa las acciones que pueden realizar los personajes del juego.

Interfaz

se explica con: ACCIÓN.

géneros: *Accion*.

TP DiseñoOperaciones básicas de diccionario

MOVER(**in** *dir* : `string`) → *res* : *Accion*
Pre ≡ {true}
Post ≡ {res =$_{obs}$ mover(dir)}
Complejidad: $O(1)$
Descripción: genera la acción mover en la direccion dir.

PASAR() → *res* : *Accion*
Pre ≡ {true}
Post ≡ {res =$_{obs}$ pasar}
Complejidad: $O(1)$
Descripción: genera la acción pasar.

DISPARAR() → *res* : *Accion*
Pre ≡ {true}
Post ≡ {res =$_{obs}$ disparar}
Complejidad: $O(1)$
Descripción: genera la acción disparar.

ESDISPARAR(**in** *a* : `Acción`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ esDisparar(a)}
Complejidad: $O(1)$
Descripción: Chequea si la acción es disparar.

ESPASAR(**in** *a* : `Acción`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ esPasar(a)}
Complejidad: $O(1)$
Descripción: Chequea si la acción es pasar.

ESMOVER(**in** *a* : `Acción`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ esMover(a)}
Complejidad: $O(1)$
Descripción: Chequea si la acción es mover.

DIRECCION(**in** *a* : `Acción`) → *res* : `string`
Pre ≡ {esMover(a)}
Post ≡ {res =$_{obs}$ direccion(a)}
Complejidad: $O(1)$
Descripción: Devuelve la dirección.

INVERSA(**in** *acciones* : `secu(Evento)`) → *res* : `secu(Evento)`
Pre ≡ {true}
Post ≡ {longitud(acciones) = longitud(res) $\wedge_{\text{L}}(\forall i : nat)(i < longitud(res) \Rightarrow_{\text{L}} res[i] = invertir(acciones[i]))$}
Complejidad:O(long(acciones))
Descripción: Devuelve la lista de acciones invertida con cada direccion de cada evento invertida

INVERTIR(**in** *evento* : `Evento`) → *res* : `Evento`
Pre ≡ {true}

Post $\equiv$ {(evento.direccion = "NORTE"$\wedge res.direccion =$ "SUR") $\vee$
($evento.direccion =$ "SUR" $\wedge$ $res.direccion =$ "$NORTE$") $\vee$
($evento.direccion =$ "$OESTE$" $\wedge$ $res.direccion =$ "$ESTE$") $\vee$
($evento.direccion =$ "$ESTE$" $\wedge$ $res.direccion =$ "$OESTE$")}
Complejidad:O(1)
Descripción: Invierte la direccion de un evento

Representación

TP DiseñoRepresentación de Accion Explicacion

`Accion` se representa con `estr`

donde `estr` es `tupla`(*tipo*: `string`, *dirección*: `string`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ TipoVálido $\wedge_{\text{L}}$SiEsMoverHayDireccion

TipoVálido $\equiv$ El tipo puede ser: "Mover", "Pasar.º "Disparar"
SiEsMoverHayDireccion $\equiv$ Si el tipo es mover, entonces la dirección es "NORTE", "SUR", .ESTEü .OESTE".

Abs : estr e $\longrightarrow$ Acción {Rep(e)}
Abs(e) $=_{\text{obs}}$ a: Acción | (esDisparar(a) = esDisparar(e) $\wedge esPasar(a) = esPasar(e) \wedge esMover(a) = esMover(e)$)
$\wedge_{\text{L}}$ ($esMover(a) \Rightarrow_{\text{L}} direccion(a) = direccion(e)$)

Algoritmos

Mover(**in** *dir* : `String`) $\rightarrow$ *res* : *Accion*
1: res $\leftarrow$ $\langle$ "Mover",dir $\rangle$
Complejidad:$O(1)$
Justificación:

Pasar() $\rightarrow$ *res* : *Accion*
1: res $\leftarrow$ $\langle$ "Pasar","NORTE"$\rangle$
Complejidad:$O(1)$
Justificación:

Disparar() $\rightarrow$ *res* : *Accion*
1: res $\leftarrow$ $\langle$ "Disparar","NORTE"$\rangle$
Complejidad:$O(1)$
Justificación:

esDisparar(**in** *accion* : `Accion`) → *res* : *Bool*

```
1: if accion.tipo == "Disparar" then
2:     res ← true
3: else
4:     res ← false
5: end if
```

Complejidad:$O(1)$

Justificación:

---

esPasar(**in** *accion* : `Accion`) → *res* : *Bool*

```
1: if accion.tipo == "Pasar" then
2:     res ← true
3: else
4:     res ← false
5: end if
```

Complejidad:$O(1)$

Justificación:

---

esMover(**in** *accion* : `Accion`) → *res* : *Bool*

```
1: if accion.tipo == "Mover" then
2:     res ← true
3: else
4:     res ← false
5: end if
```

Complejidad:$O(1)$

Justificación:

Direccion(**in** *accion* : `Accion`) → *res* : *String*

```
1: res ← accion.direccion
```

Complejidad:$O(1)$

Justificación:

Inversa(**in** *acciones* : `secu(Evento)`) → *res* : *secu*(*Evento*)

```
1: res ← vacia()
2: itAcc ← crearUlt(acciones)
3: while hayAnterior(itAcc) do
4:     agregarAdelante(res,Invertir(Anterior(itAcc))
5:     retroceder(itAcc)
6: end while
```

Complejidad:$O(long(acciones))$

Justificación:Hay que recorrer toda la lista de acciones para invertir cada direccion , lo cual toma la longitud de las acciones

---

Invertir(**in** *evento*: `Evento`) → *res* : *Evento*

```
1: if accion.direccion == "NORTE" then
2:     res ← ⟨ evento.pos,"SUR",evento.disparo⟩
3: else
4:     if accion.tipo == "SUR" then
5:         res ← ⟨ evento.pos,"NORTE",evento.disparo⟩
6:     else
7:         if accion.tipo == "ESTE" then
8:             res ← ⟨ evento.pos,"OESTE",evento.disparo⟩
9:         else
10:            if accion.tipo == "OESTE" then
11:                res ← ⟨ evento.pos,"ESTE",evento.disparo⟩
12:            end if
13:        end if
14:    end if
15: end if
```

Complejidad:$O(1)$

Justificación:

3. Diccionario de Trie(*string*, σ)

El módulo Diccionario de Trie provee un diccionario en el que se puede definir, borrar, y testear si una clave está definida en tiempo proporcional al largo de la clave.

Para describir la complejidad de las operaciones, vamos a llamar $copy(k)$ al costo de copiar el elemento $k \in \kappa \cup \sigma$ y $equal(k_1, k_2)$ al costo de evaluar si dos elementos $k_1, k_2 \in \kappa$ son iguales.

Interfaz

parámetros formales
- **géneros** $string, \sigma$
- **función** COPIAR(**in** s: σ) $\rightarrow$ res : σ
  - **Pre** $\equiv$ {true}
  - **Post** $\equiv$ {$res =_{\text{obs}} s$}
  - **Complejidad:** $O(copy(s))$
  - **Descripción:** función de copia de σ's

se explica con: DICCIONARIO(κ, σ).

géneros: `diccStr`($string, \sigma$), `itDiccStr`($string, \sigma$).

TP DiseñoOperaciones básicas de diccionario

VACÍO() $\rightarrow$ res : `diccStr`($string, \sigma$)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacio}
Complejidad: $O(1)$
Descripción: genera un diccionario vacío.

DEFINIR(**in/out** d: `diccStr`($string, \sigma$), **in** k: $string$, **in** s: σ)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(d_0, k, s) $\wedge$ alias(esPermutación(SecuSuby(res), d))}
Complejidad: ($|k| + copy(s)$), donde $|k|$ es la longitud de la clave k.
Descripción: define la clave k con el significado s en el diccionario.
Aliasing: Los elementos k y s se definen por copia.

DEFINIDO?(**in** d: `diccStr`($string, \sigma$), **in** k: $string$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(d, k)}
Complejidad: $O(|k|)$, donde $|k|$ es la longitud de la clave k.
Descripción: devuelve `true` si y sólo k está definido en el diccionario.

SIGNIFICADO(**in** d: `diccStr`($string, \sigma$), **in** k: $string$) $\rightarrow$ res : σ
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Significado(d, k))}
Complejidad: $O(|k|)$, donde $|k|$ es la longitud de la clave k.
Descripción: devuelve el significado de la clave k en d.
Aliasing: res es una referencia modificable si y sólo si d es modificable.

BORRAR(**in/out** d: `diccStr`($string, \sigma$), **in** k: $string$)
Pre $\equiv$ {$d = d_0 \wedge$ def?(d, k)}
Post $\equiv$ {$d =_{\text{obs}}$ borrar(d_0, k)}
Complejidad: $O(|k| + copy(significado(d, k))$, donde $|k|$ es la longitud de la clave k.
Descripción: elimina la clave k y su significado de d.

CLAVES(**in** d: `diccStr`($string, \sigma$)) $\rightarrow$ res : `conj`($string$)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} d$}
Complejidad: (1)
Descripción: devuelve una referencia no modificable al conjunto de claves.

COPIAR(**in** d: `diccStr`($string, \sigma$)) $\rightarrow$ res : `dicc`($string, \sigma$)

Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} claves(d)\}$
Complejidad: $\left(\sum_{k \in K} (|k| + copy(k) + copy(\text{significado}(d,k)))\right)$, donde $|k|$ es el largo de la clave k y $K = \text{claves}(d)$)
Descripción: genera una copia nueva del diccionario.

TAD Diccionario Extendido(κ, σ)

extiende DICCIONARIO(κ, σ)

otras operaciones (no exportadas)
esPermutacion? : secu(tupla(κ, σ)) $\times$ dicc(κ, σ) $\longrightarrow$ bool
secuADicc : secu(tupla(κ, σ)) $\longrightarrow$ dicc(κ, σ)

axiomas
esPermutacion?(s, d) $\equiv$ $d = \text{secuADicc}(s) \wedge \#\text{claves}(d) = \text{long}(s)$
secuADicc(s) $\equiv$ **if** vacia?(s) **then** vacio **else** definir($\Pi_1(\text{prim}(s))$, $\Pi_2(\text{prim}(s))$, secuADict(fin(s))) **fi**

Fin TAD

Representación

TP DiseñoRepresentación del diccionario Para representar el diccionario vamos a usar un conjunto de claves y un puntero a la raíz un árbol compuesto por nodos donde las conexiones entre nodos representan los caracteres de las claves. Los nodos son tuplas de un puntero al significado y un arreglo de punteros a nodo de 256 posiciones. Cada una de estas posiciones corresponde a un caracter según la codificación ASCII. Además, si el nodo tiene un significado asociado, entonces tiene un puntero a su clave en el conjunto de claves para poder borrarla del conjunto en $O(1)$ luego de haberla encontrado en el árbol.

`diccStr`($string, \sigma$) **se representa con** `trie`

donde `trie` es `tupla`(*claves*: `conj(string)`, *raíz*: `puntero(nodo)`)
donde `nodo` es `tupla`(*clave*: *itConj*(*string*), *significado*: *puntero*(σ), *siguientes*: `arreglo[256](puntero(nodo))`)

Rep : trie t $\longrightarrow$ bool
Rep(t) $\equiv$ true $\iff$ NoHayCiclos $\wedge CaminosNoConvergen \wedge CaminosConSig \wedge NoHayAliasing \wedge ClavesCoinciden$

NoHayCiclos $\equiv$ No hay forma de recorrer los nodos a traves de los punteros en cada uno de ellos de manera que se pase 2 veces por el mismo nodo.
CaminosNoConvergen $\equiv$ No hay 2 nodos que tengan un puntero cada uno que apunte al mismo nodo (no existen 2 caminos distintos a traves de nodos que llegan al mismo nodo).
CaminosConSig $\equiv$ Si existe un nodo que tenga su significado igual a NULL, entonces tiene por lo menos 1 puntero a nodo en el array "siguientes"que no apunta a NULL(tiene como minimo 1 hijo).
NoHayAliasing $\equiv$ No hay 2 nodos que tengan sus significados que apunten al mismo lugar.
ClavesCoinciden $\equiv$ Para cada clave del conjunto de claves, existe un camino que lleva a un nodo con significado. El string codificado por el camino a dicho nodo coincide con la clave y el iterador al conjunto de claves apunta a la clave.
Por otro lado, para cada camino que lleva a un nodo con significado, la clave codificada por el camino está en el conjunto de claves y el iterador al conjunto de claves del nodo apunta a dicha clave.

Abs : trie t $\longrightarrow$ diccStr(string,σ) {Rep(t)}
Abs(t) $=_{\text{obs}}$ d: diccStr(string,σ)| Para todo camino que empieza en la raiz hasta un nodo con significado, si yo recostruyo el string de manera que cada acceso al array siguientes de cada nodo, respresente su valor en la codificacion ASCII voy a obtener la clave que esta definida en el trie y esta debe estar definida en el diccionario d,además, debe tener el mismo significado asociado en dicho diccionario y viceversa.

Algoritmos

En esta seccion nos dedicamos a justificar mediante pseudocodigo, y argumentos en lenguaje natural, la manera en

la que capa operacion de nuestro modulo es realizada y su respectiva complejidad expresada en la seccion previamente redactada de Interfaz.

TP DiseñoAlgoritmos del módulo

---

iVacío() $\rightarrow res$: trie

1: $nodo \leftarrow \langle clave : NULL, siginificado : NULL, siguientes : arreglo(256, NULL)\rangle$ ▷ $O(1)$
2: $res \leftarrow \langle claves : Vacío(), raíz : \&nodo\rangle$ ▷ $O(1)$

Complejidad: $O(1)$

---

iDefinir(**in/out** t: `trie`, **in** k: $string$, **in** s: σ)

1: $actual \leftarrow t.raíz$ ▷ $O(1)$
2: **for** c in k **do**
3: **if** $actual \rightarrow siguientes[int(c)] = NULL$ **then** ▷ $O(1)$
4: $nuevo \leftarrow \langle siginificado : NULL, arreglo(256, NULL)\rangle$ ▷ $O(1)$
5: $actual \rightarrow siguientes[int(c)] \leftarrow \&nuevo$ ▷ $O(1)$
6: **end if**
7: $actual \leftarrow actual \rightarrow siguientes[int(c)]$ ▷ $O(1)$
8: **end for**
9: **if** $actual \rightarrow significado = NULL$ **then**
10: $actual \rightarrow clave \leftarrow AgregarRápido(t.claves, k)$ ▷ $O(copy(k))$
11: **end if**
12: $sig \leftarrow s$ ▷ $O(copy(s))$
13: $actual \rightarrow significado \leftarrow \&sig$ ▷ $O(1)$

Complejidad: $O(|k| + copy(s))$
Justificación: El algoritmo tiene un ciclo que se ejecuta $|k|$ veces y que contiene operaciones $O(1)$, por lo tanto esto es $O(|k|)$. Luego llama a la función AgregarRápido cuyo costo es $O(copy(k))$ y como k es string esto es $O(|k|)$. Finalmente copia el significado s, con costo $O(copy(s))$. Aplicando álgebra de órdenes: $O(|k|)+O(|k|))+O(copy(s))$ $= O(|k| + copy(s))$.

---

iDefinido?(**in** t: `trie`, **in** k: $string$) $\rightarrow res$: $bool$

1: $actual \leftarrow t.raíz$ ▷ $O(1)$
2: $i \leftarrow 0$
3: **while** $i < k.size()$ and $actual \rightarrow siguientes[int(k[i])] \neq NULL$ **do**
4: $actual \leftarrow actual \rightarrow siguientes[int(k[i])]$ ▷ $O(1)$
5: $i \leftarrow i + 1$ ▷ $O(1)$
6: **end while**
7: **if** $i = k.size()$ and $actual \rightarrow significado \neq NULL$ **then**
8: $res \leftarrow true$ ▷ $O(1)$
9: **else**
10: $res \leftarrow false$ ▷ $O(1)$
11: **end if**

Complejidad: $O(|k|)$
Justificación: El algoritmo tiene un ciclo que se ejecuta como máximo $|k|$ veces y que contiene operaciones $O(1)$, por lo tanto esto es $O(|k|)$.

---

iSignificado(**in** t : `trie`, **in** k : $string$) $\rightarrow$ res : σ

```
1: actual ← t.raíz                                                    ▷ O(1)
2: for c in k do
3:     actual ← actual → siguientes[int(c)]                           ▷ O(1)
4: end for
5: res ← actual → significado
```

Complejidad: $O(|k|)$
Justificación: El algoritmo tiene un ciclo que se ejecuta como máximo $|k|$ veces y que contiene operaciones $O(1)$, por lo tanto esto es $O(|k|)$.

---

iBorrar(**in/out** t : `trie`, **in** k : $string$)

```
1: actual ← t.raíz                                                    ▷ O(1)
2: ultimoRelevante ← NULL                                             ▷ O(1)
3: borrarDesde ← t.raíz                                               ▷ O(1)
4: for c in k do
5:     if nodoRelevante(actual) then
6:         ultimoRelevante ← actual                                   ▷ O(1)
7:         indiceArreglar ← int(c)                                    ▷ O(1)
8:         actual ← actual → siguientes[int(c)]                       ▷ O(1)
9:         borrarDesde ← actual                                       ▷ O(1)
10:    else
11:        actual ← actual → siguientes[int(c)]                       ▷ O(1)
12:    end if
13: end for
14: if tieneHijos(actual) then
15:    delete actual → significado                                    ▷ O(copy(actual → significado))
16:    eliminarSiguiente(actual → clave)                              ▷ O(1)
17:    actual.significado ← NULL                                      ▷ O(1)
18: else
19:    borradoRecursivo(borrarDesde)                                  ▷ O(|k|)
20: end if
21: if ultimoRelevante ≠ NULL then
22:    ultimoRelevante → siguientes[indiceArreglar] ← NULL            ▷ O(1)
23: else
24:    t.raiz ← NULL                                                  ▷ O(1)
25: end if
```

Complejidad: $O(|k| + copy(significado(d, k))$
Justificación: El algoritmo tiene un ciclo que busca el nodo a borrar y se ejecuta $|k|$ veces. Como contiene operaciones $O(1)$, el costo de todo es $O(|k|)$. Luego tiene que borrar el significado s asociado a la clave k. Como el significado puede ser complejo, consideramos que borrarlo tiene un costo proporcional a realizar una copia. Además, si el nodo a borrar no tiene hijos, debe borrar todos los nodos innecesarios (no tienen definición ni hijos). Esto en el peor caso es $O(|k|)$, cuando solo queda una clave o la clave a borrar no comparte prefijo con otra. Aplicando álgebra de órdenes: $O(|k|)+O(|k|))+O(copy(s)) = O(|k| + copy(s))$.

---

iClaves(**in** t : `trie`) $\rightarrow$ res : $conj(string)$

```
1: res ← &t.claves                                                    ▷ O(1)
```

Complejidad: $O(1)$

iCopiar(**in** t : `trie`) $\rightarrow$ res : $trie$

1: $res \leftarrow \langle claves : Vacío(), raíz : NULL) \rangle$
2: $res.claves \leftarrow copiadoRecursivo(res, t.raíz)$

Complejidad: $\left(\sum_{k \in K} (|k| + copy(k) + copy(\text{significado}(d, k)))\right)$, donde $|k|$ es el largo de la clave k y $K = \text{claves}(d)$
Justificación: En el peor caso, las claves no comparten prefijos y el copiado recursivo tiene que crear $|k|$ nodos por cada clave, además copiar la clave en el conjunto de claves y copiar el significado de la clave.

---

nodoRelevante(**in** p : `puntero(nodo)`) $\rightarrow$ res : bool

1: $res \leftarrow p \rightarrow significado \neq NULL$ ▷ $O(1)$
2: $hijos \leftarrow 0$ ▷ $O(1)$
3: **for** $(i \leftarrow 0; i < 256; i++)$ **do**
4: **if** $p \rightarrow siguientes[i] \neq NULL$ **then**
5: $res \leftarrow true$ ▷ $O(1)$
6: **end if**
7: **end for**

Complejidad: $O(1)$

---

tieneHijos(**in** p : `puntero(nodo)`) $\rightarrow$ res : bool

1: $res \leftarrow false$ ▷ $O(1)$
2: **for** $(i \leftarrow 0; i < 256; i++)$ **do**
3: **if** $p \rightarrow siguientes[i] \neq NULL$ **then**
4: $res \leftarrow true$ ▷ $O(1)$
5: **end if**
6: **end for**

Complejidad: $O(1)$

---

borradoRecursivo(**in/out** p : `puntero(nodo)`)

1: **if** $p \neq NULL$ **then**
2: $borradoRecursivoSiguientes(p \rightarrow siguientes)$
3: $delete\ p \rightarrow significado$
4: $delete\ p$
5: **end if**

---

borradoRecursivoSiguientes(**in/out** a : `arreglo(puntero(nodo))`)

1: **for** p in a **do**
2: $borradoRecursivo(p)$
3: **end for**

---

copiadoRecursivo(**in/out** t : `trie`, **in** p : `puntero(nodo)`) $\rightarrow$ res : puntero(nodo)

```
1: if p ≠ NULL then
2:     nodo ← ⟨clave : NULL, siginificado : NULL, siguientes : arreglo(256, NULL)⟩
3:     if p ← clave ≠ NULL then
4:         nodo → clave ← AgregarRápido(t.claves, siguiente(p → clave))
5:     end if
6:     if p ← definicion ≠ NULL then
7:         nuevaDef ← p → definicion
8:         nodo → definicion ← nuevaDef
9:     end if
10:    for (i ← 0; i < 256; i++) do
11:        if p → siguientes[i] ≠ NULL then
12:            nuevo → siguientes[i] ← copiadoRecursivo(p → siguientes[i])
13:        end if
14:    end for
15:    res ← &nuevo
16: else
17:    res ← NULL
18: end if
```

4. Módulo Habitación

se explica con: HABITACION

POSICION es TUPLA(NAT,NAT)

CELDA es TUPLA(*obstaculo:* BOOL,*jugadores:* NAT,*fantasmas:* NAT,*disparada:* BOOL)

(Las Postcondiciones tienen también elementos por fuera del TAD para mayor claridad sobre las funciones)

Interfaz

TP DiseñoOperaciones básicas de Habitación NUEVOMAPA(**in** t : `nat`) $\rightarrow$ res : `Habitación`
Pre $\equiv$ {true}
Post $\equiv$ $\{[(\forall i, j : nat) 0 \leq i, j < t \rightarrow_L res.tablero[i][j] = \langle False, 0, 0, False \rangle] \wedge res.tamano = t \wedge res.pos\_disparadas = \emptyset\}$
Complejidad: $O(t^2)$
Descripción: generador vacío de habitación

AGREGARJUGADORES(**in/out** m : `Habitación`, **in** jug : `lista(posicion)`)
Pre $\equiv$ $\{m = m_0 \ \wedge \ (\forall p : posicion) p \in jug \rightarrow p \in casilleros(\hat{m}) \ \wedge_L \ libre(\hat{m}, p)\}$
Post $\equiv$ $\{\forall (p : posicion) p \in jug \rightarrow_L (\sum_{i=0}^{longitud(jug)} if \ jug[i] = p \ then \ 1 \ else \ 0) + m_0.tablero[p].fantasmas = m.tablero[p].jugadores \ \wedge \ m.tamano = m_0.tamano \ \wedge \ m.pos\_disparadas = m_0.pos\_disparadas \ \wedge$
$[(\forall i, j : nat) 0 \leq i, j < m.tamano \rightarrow_L (m.tablero[i][j].obstaculo = m_0.tablero[i][j].obstaculo) \wedge (m.tablero[i][j].fantasmas = m_0.tablero[i][j].fantasmas) \ \wedge \ (m.tablero[i][j].disparada = m_0.tablero[i][j].disparada)]\}$
Complejidad: $O(longitud(jug))$
Descripción: agrega un numero de jugadores al mapa

AGREGARFANTASMAS(**in/out** m : `Habitación`, **in** fan : `lista(posicion)`)
Pre $\equiv$ $\{m = m_0 \ \wedge \ (\forall p : posicion) p \in fan \rightarrow p \in casilleros(\hat{m}) \ \wedge_L \ libre(\hat{m}, p)\}$
Post $\equiv$ $\{\forall (p : posicion) p \in fan \rightarrow_L (\sum_{i=0}^{longitud(fan)} if \ fan[i] = p \ then \ 1 \ else \ 0) + m_0.tablero[p].fantasmas = m.tablero[p].fantasmas \ \wedge \ m.tamano = m_0.tamano \ \wedge \ m.pos\_disparadas = m_0.pos\_disparadas \ \wedge$
$[(\forall i, j : nat) 0 \leq i, j < m.tamano \rightarrow_L (m.tablero[i][j].obstaculo = m_0.tablero[i][j].obstaculo) \wedge (m.tablero[i][j].jugadores = m_0.tablero[i][j].jugadores) \ \wedge \ (m.tablero[i][j].disparada = m_0.tablero[i][j].disparada)]\}$
Complejidad: $O(longitud(fan))$
Descripción: agrega un numero de fantasmas al mapa

AGREGAROBSTACULOS(**in/out** m : `Habitación`, **in** obs : `lista(posicion)`)

Pre $\equiv$ { $m = m_0 \ \wedge \ (\forall p : posicion)p \in obs \rightarrow alcanzan(libres(\hat{m}) - p, libres(\hat{m}) - p) \ \wedge \ [(\forall i, j)0 < i, j < m.tamano \rightarrow_L m.tablero[i][j] = \langle False, 0, 0, False \rangle]$ }

Post $\equiv \{\forall (p : posicion)p \in obs \rightarrow_L m.tablero[p].obstaculo \ \wedge \ m.tamano = m_0.tamano \ \wedge \ m.pos\_disparadas = m_0.pos\_disparadas \ \wedge$
$[(\forall i, j : nat)0 \leq i, j < m.tamano \rightarrow_L (m.tablero[i][j].jugadores = m_0.tablero[i][j].jugadores) \wedge (m.tablero[i][j].fantasmas$
$m_0.tablero[i][j].fantasmas) \ \wedge \ (m.tablero[i][j].disparada = m_0.tablero[i][j].disparada)]\}$

Complejidad: $O(longitud(obs))$

Descripción: agrega un numero de obstáculos al mapa

ESOBSTACULO(**in** m : `Habitación`,**in** pos : `posicion`) $\rightarrow res$: `bool`

Pre $\equiv \{\forall p \in pos \rightarrow_L p \in casilleros(\hat{m})\}$

Post $\equiv$ {res $=_{\text{obs}}$ m.tablero[pos].obstaculo}

Complejidad: $O(1)$

Descripción: ve si una posicion del mapa tiene un obstaculo

POSDISPARADASFANTASMA(**in** m : `Habitación`) $\rightarrow res$: `conj(pos)`

Pre $\equiv$ {true}

Post $\equiv \{(\forall p : posicion)p \in res \rightarrow m.tablero[p].disparada\}$

Complejidad:$O(1)$

Descripción: Las posiciones disparadas por fantasmas en la ronda anterior. Lo creimos un poco extraño este pedido, pero la consigna al día 4/6/2019, día anterior a la entrega.

Aliasing: Se devuelve un conjunto de posiciones como referencia constante (inmutable) a la lista guardada en la estructura

ESTAVIVO(**in** m : `Habitación`,**in** jug : `bool`,**in** pos : `posicion`) $\rightarrow res$: `bool`

Pre $\equiv$ {true}

Post $\equiv \{res = ((m.tablero[pos].jugadores = 0) \ \wedge \ jug) \ \vee \ ((m.tablero[pos].fantasmas = 0) \ \wedge \ \neg jug)\}$

Complejidad: $O(1)$

Descripción: Establece si un fantasma o jugador está vivo, chequeando si hay jugadores o fantasmas en esa posición o no. Solo tiene sentido esta operación inmediatamente después de disparar, sino la información se pierde

ADYACENTE(**in** pos : `posicion`,**in** dir : `direccion`) $\rightarrow res$: `posicion`

Pre $\equiv$ {true}

Post $\equiv \{res = (dir =_{\text{obs}} norte \ \wedge \ res = pos + \langle 1, 0 \rangle) \ \vee \ (dir =_{\text{obs}} sur \ \wedge \ res = pos + \langle -1, 0 \rangle) \ \vee \ (dir =_{\text{obs}} este \ \wedge \ res = pos + \langle 0, 1 \rangle) \ \vee \ (dir =_{\text{obs}} norte \ \wedge \ res = pos + \langle 0, -1 \rangle)\}$

Complejidad: $O(1)$

Descripción: Da la siguiente posición en dirección de

DISPARAR(**in/out** m : `Habitación`,**in** jug : `bool`,**in** pos : `posicion`,**in** dir : `direccion`)

Pre $\equiv$ { $m = m_0 \ \wedge \ p \in casilleros(\hat{m}) \ \wedge_L \ libre(\hat{m}) \ \wedge_L \ [(m.tablero[pos].jugadores > 0 \ \wedge \ jug) \ \vee \ (m.tablero[pos].fantasmas > 0 \ \wedge \ \neg jug)]\}$

Post $\equiv$ { $[(\forall p : posicion)p \in alcanceDisparo(pos, dir, \hat{m}) \rightarrow_L$
$m.tablero[p] = \langle False,$
$if \ jug \ then \ m_0.tablero[p].jugadores \ else \ 0, if \ jug \ then \ 0 \ else \ m_0.tablero[p].jugadores, True \rangle]$
$\wedge \ [(\forall p : posicion)p \notin alcanceDisparo(pos, dir, \hat{m}) \ \wedge \ p \in casilleros(\hat{m}) \rightarrow_L$
$m.tablero[p] = m_0.tablero[p]] \ \wedge \ [m.tamano = m_0.tamano]$
$\wedge [m.pos\_disparadas = m_0.pos\_disparadas \cup alcanceDisparo(pos, dir, \hat{m})]\}$

Complejidad: $O(m.tamano)$

Descripción: Realiza el proceso de disparo, poniendo a 0 las categorías contrarias a la que disparó

MOVER(**in/out** m : `Habitación`,**in** jug : `bool`,**in** pos : `posicion`,**in** dir : `direccion`)

Pre $\equiv \{m = m_0 \ \wedge \ pos \in casilleros(\hat{m}) \ \wedge_L \ libre(\hat{m}, pos) \ \wedge \ adyacente(pos, \hat{m}) \in casilleros(\hat{m}) \ \wedge_L \ libre(\hat{m}, adyacente(pos, \hat{m})) \wedge [(m.tablero[pos].jugadores > 0 \wedge jug) \vee (m.tablero[pos].fantasmas > 0 \wedge \neg jug)]\}$

Post $\equiv \{[m.tablero[pos] = \langle False, m_0.tablero[pos].jugadores - if \ jug \ then \ 1 \ else \ 0, m_0.tablero[pos].fantasmas - if \ jug \ then \ 0 \ else \ 1, m_0.tablero[pos].disparada \rangle]$
$\wedge \ [m.tablero[adyacente(pos, dir)] = \langle False, m_0.tablero[adyacente(pos, dir)].jugadores + if \ jug \ then \ 1 \ else \ 0,$
$m_0.tablero[adyacente(pos, dir)].fantasmas + if \ jug \ then \ 0 \ else \ 1, m_0.tablero[adyacente(pos, dir)].disparada \rangle]$

$\wedge$ $[(\forall p : posicion)p \in casilleros(\hat{m}) \wedge p \notin \{pos, adyacente(pos, dir)\} \rightarrow_L m.tablero[p] = m_0.tablero[p]$
$\wedge$ $[m.tamano = m_0.tamano]$ $\wedge$ $[m.pos\_disparadas = m_0.pos\_disparadas]$ }
Complejidad: $O(1)$
Descripción: Actualiza el número de jugadores o fantasmas en base a un movimiento válido. **Se le debe pasar la posición vieja y la dirección a la cual se mueve**

esMovValido(**in** m : `Habitación`,**in** pos : `posicion`,**in** dir : `direccion`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = $(dir = norte \wedge hayAdy \uparrow (pos, \hat{m}))$
$\vee$ $(dir = sur \wedge hayAdy \downarrow (pos, \hat{m}))$
$\vee$ $(dir = este \wedge hayAdy \rightarrow (pos, \hat{m}))$
$\vee$ $(dir = oeste \wedge hayAdy \leftarrow (pos, \hat{m}))$ }
Complejidad: $O(1)$
Descripción: Determina si un movimiento puede hacerse

resetear(**in/out** m : `Habitación`)
Pre $\equiv$ $\{m = m_0\}$
Post $\equiv$ $\{(\forall p : posicion)p \in casilleros(\hat{m}) \rightarrow_L m.tablero[p] = \langle m_0.tablero[p].obstaculo, 0, 0, False\rangle \wedge m.tamano = m_0.tamano \wedge m.pos\_disparadas = \emptyset\}$
Complejidad: $O((m.tamano)^2)$
Descripción: Vuelve el mapa a un estado sin jugadores, fantasmas o posiciones disparadas pero conserva obstáculos y tamaño

TP DiseñoEspecificación de las operaciones auxiliares utilizadas en la interfaz

iEsPosValida(**in/out** m : `Habitación`,**in** pos : `posicion`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = pos $\in casilleros(\hat{m}) \wedge_L libre(pos, \hat{m})$}
Complejidad:$O(1)$
Descripción: Determina si una posición está dentro del tablero y no tiene obstáculos

Representación

TP DiseñoRepresentación de la Habitación
La habitación se representa como una tupla, que contiene:

- tablero: un arreglo de arreglos de celdas, ambos de longitud m ya que es cuadrado. Las celdas tienen un booleano que representa si es obstaculo o no, un natural con la cantidad de jugadores en ella, otro natural con la cantidad de fantasmas en ella y un booleano que representa si pasó un disparo por esa posición en la ronda.
- tamano: un natural que representa la longitud m
- pos_disparadas: un conjunto lineal que contiene las posiciones en las que hubo disparos

`Habitación` **se representa con** `habitación`

donde `habitación` es `tupla`(*tablero*: `arreglo_dimensionable de (arreglo_dimensionable de celda)` , *tamano*: `nat` , *pos_disparadas*: `conjlineal(posicion)`)

Invariante de Representación

El invariante de represantación se compone de las siguientes condiciones:

- Los arreglos de m.tablero son de tamaño m.tamano
- Si una posición es obstáculo, no tiene ni jugadores ni fantasmas y nunca fue disparada
- Si una posicion está en m.pos_disparadas entonces está marcada como tal en el tablero (m.tablero(pos).disparada = True)
- Todas las posiciones son alcanzables.

Función de Abstracción

El Módulo Habitación toma parte de las modalidades del TAD Habitación, cuya instancia llamaremos Hab:

- m.tamano = tam(Hab)
- Para toda p:posicion, p $\in casilleros(\hat{m}), m.tablero[p].obstaculo = \neg$libre(p,Hab)

Algoritmos

---

nuevoMapa(**in** t : `nat`) $\rightarrow$ res : Habitación
  $res.tablero \leftarrow crearArreglo(t)$
  $i \leftarrow 0$
  $celda\_vacia \leftarrow \langle False, 0, 0.False \rangle$
  while $i < t$ **do**
    $res.tablero[i] = crearArreglo(t)$
    $j \leftarrow 0$
    while $j < t$ **do**
      $res.tablero[i][j] = celda\_vacia$
      $j++$
    end while
    $i++$
  end while
  $res.tamano = t$
  $res.pos\_disparadas = \emptyset$
  Complejidad: $O(t^2)$

---

iEsPosValida(**in/out** m : `Habitación`,**in** pos : `posicion`) $\rightarrow$ res : $bool$
  $res \leftarrow [(0 \leq \pi_1(pos) < m.tamano) \wedge (0 \leq \pi_2(pos) < m.tamano) \wedge_L \neg(m.tablero[\pi_1(pos)][\pi_2(pos)].obstaculo)]$
  Complejidad: $O(1)$

---

agregarJugadores(**in/out** m : `Habitación`, **in** jug : `lista(posicion)`)
  $t \leftarrow \&(m.tablero)$
  $conj \leftarrow craerIt(jug)$
  while $conj \neq NULL$ **do**
    $pos \leftarrow conj^*$
    $t[\pi_1(pos)][\pi_2(pos)] \leftarrow \langle t[\pi_1(pos)][\pi_2(pos)].obstaculo, t[\pi_1(pos)][\pi_2(pos)].jugadores + 1,$
  $t[\pi_1(pos)][\pi_2(pos)].fantasmas,\ t[\pi_1(pos)][\pi_2(pos)].disparada \rangle$
    $conj++$
  end while
  Complejidad: $O(longitud(jug))$

---

agregarFantasmas(**in/out** m : `Habitación`, **in** fan : `lista(posicion)`)
  $t \leftarrow \&(m.tablero)$
  $conj \leftarrow craerIt(fan)$
  while $conj \neq NULL$ **do**
    $pos \leftarrow conj^*$
    $t[\pi_1(pos)][\pi_2(pos)] \leftarrow \langle t[\pi_1(pos)][\pi_2(pos)].obstaculo, t[\pi_1(pos)][\pi_2(pos)].jugadores,$
  $t[\pi_1(pos)][\pi_2(pos)].fantasmas + 1,\ t[\pi_1(pos)][\pi_2(pos)].disparada \rangle$
    $conj++$
  end while
  Complejidad: $O(longitud(fan))$

---

agregarObstaculos(**in/out** m : `Habitación`, **in** obs : `lista(posicion)`)

$t \leftarrow \&(m.tablero)$
$conj \leftarrow craerIt(obs)$
while $conj \neq NULL$ **do**
  $pos \leftarrow conj^*$
  $t[\pi_1(pos)][\pi_2(pos)] \leftarrow \langle True, 0, 0, False \rangle$
  $conj++$
end while
Complejidad: $O(longitud(obs))$

---

esObstaculo(**in** m : `Habitación`, **in** pos : `posicion`) $\rightarrow$ res : $bool$

$res \leftarrow m[\pi_1(pos)][\pi_2(pos)].obstaculo$
Complejidad: $O(1)$

---

posDisparadasFantasma(**in** m : `Habitación`) $\rightarrow$ res : $conj(pos)$

$res \leftarrow \&(m.pos\_disparadas)$
Complejidad: $O(1)$

---

estaVivo(**in** m : `Habitación`, **in** jug : `bool`, **in** pos : `posicion`) $\rightarrow$ res : $bool$

if jug **then**
  $res \leftarrow m.tablero[\pi_1(pos)][\pi_2(pos)].jugadores = 0$
else
  $res \leftarrow m[\pi_1(pos)][\pi_2(pos)].fantasmas = 0$
end if
Complejidad: $O(1)$

---

adyacente(**in** pos : `posicion`,**in** dir : `direccion`) $\rightarrow$ res : $posicion$

if $dir = norte$ **then**
  $res \leftarrow \langle \pi_1(pos), \pi_2(pos) + 1 \rangle$
else
  if $dir = sur$ **then**
    $res \leftarrow \langle \pi_1(pos), \pi_2(pos) - 1 \rangle$
  else
    if $dir = este$ **then**
      $res \leftarrow \langle \pi_1(pos) + 1, \pi_2(pos) \rangle$
    else
      $res \leftarrow \langle \pi_1(pos) - 1, \pi_2(pos) \rangle$
    end if
  end if
end if
Complejidad: $O(1)$

disparar(**in/out** m : `Habitación`,**in** jug : `bool`,**in** pos : `posicion`,**in** dir : `direccion`)

$pos\_actual = adyacente(pos, dir)$
while iEsPosValida(m,pos_actual) **do**
  if jug **then**
    $m.tablero[pos\_actual].fantasmas = 0$
  else
    if $\neg(m.tablero[pos\_actual].disparada)$ **then**
      $agregarRapido(pos\_actual, m.pos\_disparadas)$
      $m.tablero[pos\_actual].disparada = True$
    end if
    $m.tablero[pos\_actual].jugadores = 0$
  end if
  $pos\_actual = adyacente(pos\_actual, dir)$
end while
Complejidad: $O(m.tamano)$

---

mover(**in/out** m : `Habitación`,**in** jug : `bool`,**in** pos : `posicion`,**in** dir : `direccion`)

$pos\_nueva = adyacente(pos, dir)$
if jug **then**
  $m.tablero[pos].jugadores - -$
  $m.tablero[pos\_nueva].jugadores + +$
else
  $m.tablero[pos].fantasmas - -$
  $m.tablero[pos\_nueva].fantasmas + +$
end if
Complejidad: $O(1)$

---

esMovValido(**in** m : `Habitación`, **in** pos : `posicion`, **in** dir : `direccion`) $\rightarrow res : bool$

$res \leftarrow iEsPosValida(m, adyacente(pos, dir))$
Complejidad: $O(1)$

---

resetear(**in/out** m : `Habitación`)

$i \leftarrow 0$
while $i < m.tamano$ **do**
  $j \leftarrow 0$
  while $j < m.tamano$ **do**
    $m.tablero[i][j] = \langle m.tablero[i][j].obstaculo, 0, 0, False \rangle$
    $j + +$
  end while
  $i + +$
end while
$m.pos\_disparadas = \emptyset$
Complejidad: $O((m.tamano)^2)$
Borrar el conjunto es $O(cardinal)$ pero como este contiene posiciones del tablero sin repetidos, tiene como mucho $(m.tamano)^2$ posiciones y no cambia entonces la complejidad.

Servicios Usados

Se usaron los siguientes servicios:

- Bool

- Nat
- Arreglo (redimiensionables, pero en este caso se mantiene constante su longitud). En este caso, usando arreglos de arreglos, y una posición *pos*, el acceso debería ser $arreglo[\pi_1(pos)][\pi_2(pos)]$ pero usamos $arreglo[pos]$ para simplificar la lectura
- Conjunto lineal, con posibilidad de repetidos, inserción en $O(1)$ y búsqueda en $O(n)$ con su iterador correspondiente
- Lista enlazada, con inserción en $O(1)$ y búsqueda en $O(n)$ con su iterador correspondiente