

Apunte de Módulos Básicos (v. 0.3 α)

Algoritmos y Estructuras de Datos II, DC, UBA.

1^{er} cuatrimestre de 2019

Índice

1. Diccionario Trie (α)	2
2. Módulo Juego	5
3. Módulo Mapa	13
4. Módulo Dirección	15
5. Módulo Acción	17

1. Diccionario Trie (α)

El módulo Diccionario Trie provee un diccionario básico montado sobre un trie. Solo se definen e implementan las operaciones que serán utilizadas.

Interfaz

parámetros formales

géneros α
función $\text{COPIAR}(\text{in } s : \alpha) \rightarrow \text{res} : \alpha$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} s\}$
Complejidad: $\Theta(\text{copy}(s))$
Descripción: función de copia de α

se explica con: $\text{DICCIONARIO}(\text{string}, \alpha)$.

géneros: $\text{diccTrie}(\text{string}, \alpha)$.

Operaciones básicas de diccionario

$\text{VACÍO}() \rightarrow \text{res} : \text{diccTrie}(\text{string}, \alpha)$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{vacío}\}$
Complejidad: $\Theta(1)$
Descripción: genera un diccionario vacío.

$\text{DEFINIR}(\text{in/out } d : \text{diccTrie}(\text{string}, \alpha), \text{in } k : \text{string}, \text{in } s : \alpha)$
Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}} \text{definir}(d, k, s)\}$
Complejidad: $\Theta(|k| + \text{copy}(s))$
Descripción: define la clave $k \notin \text{claves}(d)$ con el significado s en el diccionario.
Aliasing: los elementos k y s se definen por copia.

$\text{DEFINIDO?}(\text{in } d : \text{diccTrie}(\text{string}, \alpha), \text{in } k : \text{string}) \rightarrow \text{res} : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{def?}(d, k)\}$
Complejidad: $\mathcal{O}(|k|)$
Descripción: devuelve **true** si y sólo k está definido en el diccionario.

$\text{SIGNIFICADO}(\text{in } d : \text{diccTrie}(\text{string}, \alpha), \text{in } k : \text{string}) \rightarrow \text{res} : \sigma$
Pre $\equiv \{\text{def?}(d, k)\}$
Post $\equiv \{\text{alias}(\text{res} =_{\text{obs}} \text{obtener}(d, k))\}$
Complejidad: $\Theta(|k|)$
Descripción: devuelve el significado de la clave k en d .
Aliasing: res es modificable si y sólo si d es modificable.

Representación

Representación del diccionario

$\text{diccTrie}(\text{string}, \alpha)$ se representa con **estr**

donde **estr** es $\text{tupla}(\text{raiz: puntero(nodo)},$
 $\text{claves: conj(string)})$

donde **nodo** es $\text{tupla}(\text{significado: } \alpha,$
 $\text{siguientes: arreglo(puntero(nodo))[256]})$

Rep : $\text{diccTrie} \rightarrow \text{bool}$

$\text{Rep}(d) \equiv \text{true} \iff$

(Los nodos del diccionario (excepto la raíz) tienen un unico padre. Es decir, no hay dos Nodos en la estructura que tengan punteros iguales en los siguientes del Nodo. \wedge

La raíz no tiene padre. Es decir, no hay un camino de hijos por el cual se llegue a dicho Nodo. \wedge

Todas las hojas tienen un significado distinto de NULL.

Un s string pertenece a d .claves si y solo si se puede seguir un camino de nodos en el diccionario con los caracteres de s (en orden) llegando finalmente al ultimo nodo (correspondiente a la ultima letra) teniendo este significado distinto de NULL)

// La primer condicion implica que no hay ciclos ni Nodos con hijos de menor nivel

$\text{Abs} : \text{estr } e \longrightarrow \text{diccTrie}(\text{string}, \alpha)$

$\{\text{Rep}(e)\}$

$\text{Abs}(e) =_{\text{obs}} d : \text{diccTrie}(\text{string}, \alpha) \mid$

$(\forall :: s \text{ string})(\text{def?}(s,d) =_{\text{obs}} \text{estaDefinido}(e.\text{raiz}, s)) \wedge$

$(\forall :: s \text{ string})(\text{def?}(s,d) \Rightarrow_{\text{L}} \text{obtener}(s, d) =_{\text{obs}} \text{significado}(e.\text{raiz}, s)) \wedge$

$\text{claves}(d) =_{\text{obs}} e.\text{claves}$

$\text{estaDefinido}(r, s) \equiv \text{if } \text{vacía?}(s)$

then $r \rightarrow \text{significado} \neq \text{NULL}$

else $r \rightarrow \text{siguientes}[\text{int}(\text{prim}(s))] \neq \text{NULL} \wedge_{\text{L}} \text{estaDefinido}(r.\text{siguientes}[\text{int}(\text{prim}(s))], \text{fin}(s))$ **fi**

$\text{significado}(r,s) \equiv \text{if } \text{vacía?}(s)$

then $r \rightarrow \text{significado}$

else $\text{significado}(r.\text{siguientes}[\text{int}(\text{prim}(s))], \text{fin}(s))$ **fi**

Algoritmos

iVacía() $\rightarrow res : \text{estr}$

1: // Le asigna un nuevo nodo a la raíz

2: $res \leftarrow \langle \text{raiz} : \text{nuevoNodo}() \rangle$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

Justificación: La complejidad de crear un nuevo nodo es $\Theta(1)$

iSignificado(in/out $d : \text{estr}$, in $k : \text{string}$) $\rightarrow res : \alpha$

1: $\text{Nodo actual} \leftarrow d.\text{raiz}$

$\triangleright \Theta(1)$

2: **for** ($\text{char } c : k$) **do**

$\triangleright \mathcal{O}(|k|)$

3: $\text{actual} \leftarrow (\text{actual} \rightarrow \text{siguientes}[\text{toInt}(c)])$

$\triangleright \Theta(1)$

4: **end for**

5: $res \leftarrow *(\text{actual} \rightarrow \text{significado})$

$\triangleright \Theta(1)$

Complejidad: $\Theta(|k|)$

Justificación: Los accesos y las asignaciones de punteros son $\Theta(1)$. Como el ciclo se ejecuta $|k|$ veces, se ejecutaran dichas asignaciones $|k|$ veces. Luego la complejidad será $\Theta(|k|)$.

iDefinido?(in/out d : **estr**, in k : *string*) $\rightarrow res$: **bool**

```

1: Nodo actual  $\leftarrow d.raiz$   $\triangleright \Theta(1)$ 
2: for (char  $c$  :  $k$ ) do  $\triangleright \mathcal{O}(|k|)$ 
3:   if (actual  $\rightarrow siguientes[toInt(c)] \neq NULL$ )  $\triangleright \Theta(1)$ 
4:     then actual  $\leftarrow (actual \rightarrow siguientes[toInt(c)])$   $\triangleright \Theta(1)$ 
5:     else  $res \leftarrow false$   $\triangleright \Theta(1)$ 
6:   end if
7: end for
8:  $res \leftarrow ((actual \rightarrow significado) \neq NULL)$   $\triangleright \Theta(1)$ 

```

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

Justificación: Los accesos y las asignaciones de punteros son $\Theta(1)$. Como el ciclo se ejecuta a lo sumo $|k|$ veces, se ejecutarán dichas asignaciones $|k|$ veces como máximo. Luego la complejidad será $\mathcal{O}(|k|)$.

iDefinir(in/out d : **estr**, in k : *string*, in s : α)

```

1: Nodo actual  $\leftarrow d.raiz$ 
2: for (char  $c$  :  $k$ ) do  $\triangleright \Theta(|k|)$ 
3:   // Si no tengo siguiente, lo creo
4:   if (actual  $\rightarrow siguientes[toInt(c)] == NULL$ ) then  $\triangleright \Theta(1)$ 
5:     actual  $\rightarrow siguientes[toInt(c)] = nuevoNodo()$   $\triangleright \Theta(1)$ 
6:   end if
7:   actual  $\leftarrow (actual \rightarrow siguientes[toInt(c)])$   $\triangleright \Theta(1)$ 
8: end for
9:
10: // Estoy parado en el nodo que va a tener el puntero al significado.
11: // Reservo un lugar en memoria y hago una copia del provisto en dicho lugar.
12: sig  $\leftarrow s$   $\triangleright \Theta(copy(s))$ 
13: // Asigno al significado del nodo el puntero creado con s.
14: (actual  $\rightarrow significado$ )  $\leftarrow \&sig$   $\triangleright \Theta(1)$ 
15: // Agrego la nueva clave al conjunto de claves
16: // Como precondition, se que no existe así que la agrego rapido
17: AgregarRapido(e.claves,  $k$ )  $\triangleright \Theta(copy(k))$ 
18: // Devuelvo por referencia el significado.
19:  $res \leftarrow sig$ 

```

Complejidad: $\Theta(|k| + copy(s))$

Justificación: Siempre se recorre toda la palabra para definirla, entonces el *for* siempre tiene $|k|$ ciclos. La dereferenciación y comparación de punteros, e indexación en arreglos estáticos son $\Theta(1)$.

nuevoNodo() $\rightarrow res$: puntero(nodo)

\triangleright Función privada que crea un nuevo nodo

```

1: // Reserva la memoria para un nuevo nodo con significado null y siguientes vacios
2:  $res \leftarrow \&\langle significado : NULL, siguientes : arreglo\_estatico[256] \text{ de puntero}(Nodo) \rangle$   $\triangleright \Theta(1)$ 

```

Complejidad: $\Theta(1)$

Justificación: El tiempo de creación de un array de 255 posiciones es $\mathcal{O}(255) \in \mathcal{O}(1)$

iClaves(in d : **estr**) $\rightarrow res$: conj(*string*)

```

1:  $res \leftarrow e.claves$   $\triangleright \Theta(1)$ 

```

Complejidad: $\Theta(1)$

2. Módulo Juego

Aquí va la descripción

Interfaz

generos: juego.

se explica con: JUEGO.

Operaciones básicas de Juego

INICIAR(**in** m : mapa, **in** pjs : conj(jugador), **in** $eventosFan$: vector(evento)) $\rightarrow res$: juego

Pre $\equiv \{\neg vacio(pjs) \wedge (\forall e : evento)(est?(e, eventosFan) \Rightarrow_L e.pos \in libres(m))\}$

Post $\equiv \{res =_{obs} nuevo.Juego(m, pjs, eventosFan)\}$

Complejidad: $\Theta(?)$ TODO

Descripción: crea un nuevo juego con el mapa dado, un conjunto de jugadores, y los eventos de un fantasma.

PASARTIEMPO(**in** j : juego) $\rightarrow res$: juego

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} pasar(j)\}$

Complejidad: $\Theta(?)$

Descripción: ejecuta un paso de tiempo cuando ningún jugador realiza una acción.

EJECUTARACCION(**in** j : juego, **in** a : accion, **in** pj : jugador) $\rightarrow res$: juego

Pre $\equiv \{pj \in jugadores(j) \wedge_L jugadorVivo(pj, j) \wedge \neg esPasar(a)\}$

Post $\equiv \{res =_{obs} step(j, a, pj)\}$

Complejidad: $\Theta(?)$

Descripción: actualiza con la acción a del jugador pj .

JUGADORESVIVOS(**in** j : juego) $\rightarrow res$: conj(puntero(infoPJ))

Pre $\equiv \{true\}$

Post $\equiv \{(\forall p : puntero(infoPJ))(p \in res \Rightarrow_L$
 $(p \rightarrow id \in jugadores(j)) \wedge_L$
 $(p \rightarrow vivo? \wedge jugadorVivo(p \rightarrow id, j)) \wedge$
 $((\forall e : evento)(e \in p \rightarrow eventos \Rightarrow_L$
 $(e.pos =_{obs} posJugador(p \rightarrow id, j)) \wedge$
 $(e.dir =_{obs} dirJugador(p \rightarrow id, j))))\}$

Complejidad: $\Theta(1)$

Descripción: devuelve un conjunto con punteros a la información de los personajes que están vivos.

Aliasing: res es no modificable.

FANTASMASVIVOS(**in** j : juego) $\rightarrow res$: conj(infoFan)

Pre $\equiv \{true\}$

Post $\equiv \{fantasmaValido(j, res)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve un conjunto referencias a la información de los fantasmas que están vivos.

Aliasing: las referencias son no modificables.

FANTASMAESPECIAL(**in** j : juego) $\rightarrow res$: infoFan

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} fantasmaEspecial(j)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve el fantasma especial.

Aliasing: res es una referencia no modificable.

FANTASMASVIVOSQUEDISPARAN(**in** j : juego) $\rightarrow res$: conj(infoFan)

Pre $\equiv \{true\}$

Post $\equiv \{fantasmaValido(j, res) \wedge_L$
 $((\forall f : infoFan)(f \in res \Rightarrow_L disparando(f.eventos, step(j))))\}$

Complejidad: $O(\#fv)$

Descripción: devuelve un conjunto con punteros a la información de los fantasmas que están vivos y disparan en el ultimo paso ejecutado en el juego.

Aliasing: res es un conjunto de referencias no modificables.

VIVO?(in j : juego, in pj : string) $\rightarrow res$: bool

Pre $\equiv \{pj \in jugadores(j)\}$

Post $\equiv \{res =_{\text{obs}} jugadorVivo(pj, j)\}$

Complejidad: $O(|j|)$

Descripción: devuelve si un jugador está vivo

POSOCUPADASPORDISPAROS(in j : juego) $\rightarrow res$: conj(posicion)

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

Post $\equiv \{res =_{\text{obs}} alcanceDisparosFantasmas(fantasmas(j), j)\}$

Complejidad: $O(\#fv * m)$

Descripción: devuelve un conjunto de las posiciones afectadas por disparos de fantasmas en la última *ronda* (TODO: ronda o paso?).

Predicados auxiliares:

fantasmaValido(j , fs):

$$(\forall f : infoFan)(f \in res \Rightarrow_L$$

$$(f.eventos \in fantasmas(j)) \wedge_L$$

$$(fantasmaVivo(f.eventos, j)) \wedge$$

$$((\forall e : evento)(e \in f.eventos \Rightarrow_L$$

$$(e.pos =_{\text{obs}} posFantasma(f.eventos, j)) \wedge$$

$$(e.dir =_{\text{obs}} dirFantasma(f.eventos, j))))$$

Representación

Representación de Juego

juego se representa con estr

donde j es tupla(// General

$paso$: nat,
 $ronda$: nat,
 $mapa$: m,

// Disparos

$mapaDisparos$: arreglo(arreglo(tupla(nat, nat))),
 $disparosUltimoPaso$: conj(posicion),

// Jugadores

$infoJugadores$: diccTrie(string, infoPJ),
 $infoActualJugadoresVivos$: conj(infoActualPJ),
 $infoJugadoresVivos$: conj(puntero(infoPJ)),

// Fantasmas

$infoFantasmas$: conj(infoFan),
 $infoActualFantasmasVivos$: conj(infoActualFan),
 $infoFantasmasVivos$: conj(itConj(infoFan)),
 $infoFantasmaEspecial$: itConj(infoActualFan))

donde infoPJ es tupla($eventos$: vector(evento),

$vivo?$: bool,

$infoActual$: itConj(infoActualPJ))

donde infoActualPJ es tupla($identidad$: string,

$posicion$: pos,

$direccion$: dir)

donde infoFan es tupla($infoActual$: itConj(infoActualFan),

$eventos$: vector(evento))

donde **infoActualFan** es **tupla**(*posicion*: **pos**,
direccion: **dir**)

Rep : mapa \rightarrow bool
Rep(*m*) \equiv true \iff

Abs : mapa *m* \rightarrow hab {**Rep**(*m*)}
Abs(*m*) =_{obs} h: hab |

Algoritmos

En esta sección se hace abuso de notación en los cálculos de álgebra de órdenes presentes en la justificaciones de los algoritmos. La operación de suma “+” denota secuencialización de operaciones con determinado orden de complejidad, y el símbolo de igualdad “=” denota la pertenencia al orden de complejidad resultante.

Algoritmos del módulo

```

iIniciar(in m : mapa, in pjs : conj(jugador), in eventosFan : vector(evento))  $\rightarrow$  res : estr
1: // Inicializo la estructura
2: res : {
3:   // Inicializo contadores
4:   paso : 0,  $\triangleright \Theta(1)$ 
5:   ronda : 0,  $\triangleright \Theta(1)$ 
6:
7:   // Seteo el mapa
8:   mapa : m,  $\triangleright \Theta(1)$ 
9:
10:  // Inicializo el mapa de disparos con el mismo tamaño que el mapa
11:  mapaDisparos : arreglo(arreglo(tupla(nat, nat))[Tam(m)])[Tam(m)],  $\triangleright \Theta(Tam(m)^2)$ 
12:  disparosUltimoPaso : Vacio(),  $\triangleright \Theta(1)$ 
13:
14:  // Inicializo estructuras de jugadores y fantasmas como vacías
15:  infoActualJugadoresVivos : Vacio(),
16:  infoJugadoresVivos : Vacio(),
17:  infoJugadores : Vacia(),
18:  infoFantasmas : Vacio(),
19:  infoActualFantasmasVivos : Vacio(),
20:  infoFantasmasVivos : Vacia(),
21:  infoFantasmaEspecial : CrearIt(Vacio())
22: }
23:
24: // Inicializo los jugadores
25: iIniciarJugadores(res, m, pjs)
26:
27: // Inicializo los fantasmas
28: iIniciarFantasmas(res, eventosFan)

```

Complejidad: $\Theta(?)$

```

iIniciarJugadores(in  $j$ : estr, in  $m$ : mapa, in  $pjs$ : conj(jugador)) ▷ Función privada
1: // Suponemos la existencia de la función
2: //  $dict(jugador, tupla(pos, dir))$  localizarJugadores( $m$ , conj(jugador)  $pjs$ )
3:
4: // Obtengo las posiciones y direcciones de jugadores
5:  $localPJs \leftarrow localizarJugadores(m, pjs)$ 
6:
7: // Lleno las estructuras de jugadores
8: for ( $j$ , localizacion :  $localPJs$ ) do
9:   // Creo la infoActual y la agrego a su conjunto
10:   $infoActual \leftarrow \langle identidad : j, posicion : localizacion.pos, direccion : localizacion.dir \rangle$ 
11:   $itInfoActual \leftarrow AgregarRapido(j.infoActualJugadoresVivos, infoActual)$ 
12:
13:  // Creo la infoPJ con la actual
14:   $info \leftarrow iNuevaInfoPJ(localizacion, itInfoActual)$ 
15:  // La agrego al trie y me guardo el puntero a la info guardada
16:   $infoPtr \leftarrow \&Definir(j.infoJugadores, j, info)$ 
17:
18:  // Agrego al conjunto de jugadores vivos el puntero a la info del PJ
19:   $AgregarRapido(j.infoJugadoresVivos, infoPtr)$ 
20: end for

Complejidad:  $\Theta(?)$ 

```

```

iNuevaInfoPJ(in localizacion: tupla(pos, dir), in itInfoActual : itConj(infoActualPJ)  $\rightarrow res$  : infoPJ ▷
Función privada
1: // Creo el evento
2:  $evento \leftarrow \langle$  ▷  $\Theta(1)$ 
3:    $pos : localizacion.pos,$ 
4:    $dir : localizacion.dir,$ 
5:    $disparo? : false$ 
6:  $\rangle$ 
7:
8: // Creo una lista con él
9:  $evts \leftarrow Vacia()$  ▷  $\Theta(1)$ 
10:  $AgregarAtras(evts, evento)$  ▷  $\Theta(1)$ 
11:
12: // Armo la infoPJ
13:  $res \leftarrow \langle$  ▷  $\Theta(1)$ 
14:    $eventos : evts$ 
15:    $vivo? : true$ 
16:    $infoActual : itInfoActual$ 
17:  $\rangle$ 

Complejidad:  $\Theta(1)$ 

```

iIniciarFantasmas(in j : **estr**, in $eventosFan$: **vector**(evento))

▷ Función privada

```

1: // Lleno las estructuras de fantasmas
2: // Creo la infoActual y la agrego a su conjunto
3:  $infoActualFan \leftarrow \langle posicion : eventosFan[0].pos, direccion : eventosFan[0].dir \rangle$ 
4:  $itInfoActualFan \leftarrow AgregarRapido(infoActualFan, j.infoActualFantasmasVivos)$ 
5:
6: // Hago que el fantasma especial sea este
7:  $j.infoFantasmaEspecial \leftarrow itInfoActualFan$ 
8:
9: // Creo la infoFan con la actual
10:  $infoFan \leftarrow \langle infoActual : itInfoActualFan, eventos : eventosFan \rangle$ 
11: // La agrego al conjunto de información de fantasmas y me guardo su iterador
12:  $itInfoFan \leftarrow AgregarRapido(infoFan, j.infoFantasmas)$ 
13:
14: // Agrego al conjunto de fantasmas vivos el interador a la info del Fan
15:  $AgregarRapido(itInfoFan, j.infoFantasmasVivos)$ 

```

Complejidad: $\Theta(?)$

iPasarTiempo(in/out j : **estr**)

```

1: // Incremento el paso
2:  $j.paso \leftarrow j.paso + 1$ 
3:
4: // Actualizo el mapa de disparos con los fantasmas vivos
5:  $iActualizarMapaDisparosConFan(j)$ 
6:
7: // Veo que jugadores mueren
8:  $iChequearMuerteJugadores(j)$ 

```

▷ $\Theta(1)$ ▷ $\mathcal{O}(\#fv * m)$ ▷ $\Theta(\#jv)$
Complejidad: $\mathcal{O}(\#fv * m + \#jv)$

iActualizarMapaDisparosConFan(in/out j : **estr**)

```

1: // Recorro los fantasmas vivos
2: for ( $itInfoFan : j.infoFantasmasVivos$ ) do
3:     // Obtengo la información del fantasma
4:      $infoFan \leftarrow Siguiente(itInfoFan)$ 
5:
6:     // Obtengo el evento actual
7:      $eventoActual \leftarrow iEventoActualFan(infoFan, j.paso)$ 
8:
9:     // Si dispara, agrego su disparo a los del paso
10:    if ( $eventoActual.dispara?$ )
11:        then  $iAgregarDisparoFan(j, infoActual.pos, infoActual.dir)$ 
12:    end if
13: end for

```

▷ $\Theta(\#fv)$ ▷ $\Theta(1)$ ▷ $\Theta(1)$ ▷ $\mathcal{O}(m)$
Complejidad: $\mathcal{O}(\#fv * m)$

```

iChequearMuerteJugadores(in/out j : estr) ▷ Funcion privada
1: // Recorro los jugadores vivos con un iterador
2: itPJVivos ← CrearIt(j.infoJugadoresVivos) ▷  $\Theta(1)$ 
3: while HaySiguiente(itPJVivos) do ▷  $\Theta(\#jv)$ 
4:   // Obtengo su evento actual
5:   ptrInfoPJ ← Siguiente(itPJVivos) ▷  $\Theta(1)$ 
6:   eventoActual ← iEventoActualPJ(*ptrInfoPJ) ▷  $\Theta(1)$ 
7:
8:   if iPJAfectadoPorDisparo?(j, eventoActual.pos) ▷  $\Theta(1)$ 
9:     then iMuerePJ(j, itPJVivos) ▷  $\Theta(1)$ 
10:  end if
11:
12:  // Avanzo el iterador
13:  Avanzar(itPJVivos) ▷  $\Theta(1)$ 
14: end while

Complejidad:  $\Theta(\#jv)$ 
15:

```

```

iPJAfectadoPorDisparo?(in j : estr, in pos : pos) → res : bool ▷ Funcion privada
1: // El PJ estará afectado si en la posición en la que está hay un disparo de un fantasma
2: // Indexo por su posición en el mapa de disparos para obtener el paso en el que hubo un disparo del fantasma
3: pasoDispFan ← j.mapaDisparos[pos.x][pos.y].pasoDispFan ▷  $\Theta(1)$ 
4:
5: // Estará afectado si el paso del disparo del fantasma es igual al actual
6: afectado? ← (pasoDispFan == j.paso) ▷  $\Theta(1)$ 
7: res ← afectado?

Complejidad:  $\Theta(1)$ 

```

```

iMuerePJ(in/out j : estr, in/out itPJVivos : itConj(puntero(infoPJ))) ▷ Funcion privada
1: // Obtengo su información
2: infoPJ ← *Siguiente(itPJVivos) ▷  $\Theta(1)$ 
3:
4: // Lo seteo como muerto
5: infoPJ.vivo? ← false ▷  $\Theta(1)$ 
6:
7: // Lo borro del conjunto infoActualJugadoresVivos
8: EliminarSiguiente(infoPJ.infoActual) ▷  $\Theta(1)$ 
9:
10: // Lo borro del conjunto infoJugadoresVivos
11: EliminarSiguiente(itPJVivos) ▷  $\Theta(1)$ 

Complejidad:  $\Theta(1)$ 

```

```

iAgregarDisparoFan(in/out  $j$ : estr, in  $pos$ : pos, in  $dir$ : dir) ▷ Funcion privada
1: // Parado desde  $pos$  en  $mapaDisparos$ , recorro hacia  $dir$ 
2: // hasta que me choco con un obstaculo o la pared.
3: while Valida?( $j.mapa$ ,  $pos$ )  $\wedge_L$  Libre( $j.mapa$ ,  $pos$ ) do ▷  $\mathcal{O}(Tam(j.mapa))$ 
4:   // Si no pasé ya por está posición con otro fantasma
5:   // (i.e si en el mapa de disparos no está ya este paso)
6:   if  $mapaDisparos[pos.x][pos.y].pasoDispFan \neq j.paso$  ▷  $\Theta(1)$ 
7:   then
8:     // Agrego la posición al conjunto de disparos y seteo el paso al actual
9:      $AgregarRapido(j.disparosUltimoPaso, pos)$  ▷  $\Theta(1)$ 
10:     $mapaDisparos[pos.x][pos.y].pasoDispFan \leftarrow j.paso$  ▷  $\Theta(1)$ 
11:   end if
12:
13:   // Avanzo la posición en esa dirección
14:    $pos \leftarrow Avanzar(pos, dir)$  ▷  $\Theta(1)$ 
15: end while

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

```

```

iEventoActualFan(in  $info$ : infoFan, in  $paso$ : nat)  $\rightarrow res$ : evento ▷ Funcion privada
1:  $idx \leftarrow mod(j.paso, Longitud(info.eventos))$ 
2:  $res \leftarrow info.eventos[idx]$ 

Complejidad:  $\Theta(1)$ 

```

```

iEventoActualPJ(in  $info$ : infoPJ)  $\rightarrow res$ : evento ▷ Funcion privada
1:  $res \leftarrow Ultimo(info.eventos)$ 

Complejidad:  $\Theta(1)$ 

```

iEjecutarAccion(in j : estr, in a : accion, in pj : jugador) $\rightarrow res$:estr

```

1: // Incremento el paso
2:  $j.paso \leftarrow j.paso + 1$   $\triangleright \Theta(1)$ 
3:
4: // Obtengo la información del PJ
5:  $infoPJ \leftarrow Obtener(j.infoJugadores, pj)$   $\triangleright \mathcal{O}(|j|)$ 
6:
7: // Genero un evento con la acción
8:  $evtPJ \leftarrow Aplicar(a, j, iEventoActualPJ(infoPJ))$   $\triangleright \Theta(1)$ 
9:
10: // Agrego el evento al jugador
11:  $AgregarAtras(infoPJ.eventos, evtPJ)$   $\triangleright \Theta(1)$ 
12:
13: // Actualizo el mapa de disparos con el jugador
14:  $iActualizarMapaDisparosConPJ(j, evtPJ)$   $\triangleright \mathcal{O}(m)$ 
15:
16: // Veo que fantasmas mueren
17:  $iChequearMuerteFantasmas(j)$   $\triangleright \Theta(\#fv)$ 
18:
19: // Si murió el fantasma especial, cambio de ronda
20: if..
21:  $iNuevaRonda(j, infoPJ)$ 
22: else,
23: // Actualizo el mapa de disparos con los fantasmas vivos
24:  $iActualizarMapaDisparosConFan(j)$   $\triangleright \mathcal{O}(\#fv * m)$ 
25:
26: // Veo que jugadores mueren
27:  $iChequearMuerteJugadores(j)$   $\triangleright \Theta(\#jv)$ 

```

Complejidad: Sin cambiar de ronda $\mathcal{O}(|j| + m + \#fv + \#fv * m + \#jv) \in \mathcal{O}(|j| + \#fv * m + \#jv)$, cambiando $\mathcal{O}(?)$

3. Módulo Mapa

El módulo Mapa provee una habitación en la que se puede ocupar y consultar por una posición en $\Theta(1)$.

Interfaz

generos: mapa.

se explica con: HABITACIÓN.

Operaciones básicas del mapa

NUEVOMAPA(**in** $n : \text{nat}$) $\rightarrow res : \text{mapa}$

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

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

Complejidad: $\Theta(n^2)$

Descripción: genera un mapa de tamaño $n \times n$.

OCUPAR(**in/out** $m : \text{mapa}$, **in** $c : \text{tupla}(\text{int}, \text{int})$)

Pre $\equiv \{m =_{\text{obs}} m_0 \wedge c \in \text{casilleros}(m) \wedge_L \text{libre}(m, c) \wedge \text{alcanzan}(\text{libres}(m) - c, \text{libres}(m) - c, m)\}$

Post $\equiv \{m =_{\text{obs}} \text{ocupar}(c, m_0)\}$

Complejidad: $\Theta(1)$

Descripción: ocupa una posición del mapa siempre y cuando éste no deje de ser conexo.

TAM(**in** $m : \text{mapa}$) $\rightarrow res : \text{nat}$

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

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

Complejidad: $\Theta(1)$

Descripción: devuelve el tamaño del mapa.

LIBRE(**in** $m : \text{mapa}$, **in** $c : \text{tupla}(\text{int}, \text{int})$) $\rightarrow res : \text{bool}$

Pre $\equiv \{c \in \text{casilleros}(m)\}$

Post $\equiv \{res =_{\text{obs}} \text{libre}(c, m)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve si una posición está ocupada.

Representación

Representación del mapa

El objetivo de este módulo es implementar una matriz de tamaño n con vectores de booleanos que indican si una posición está ocupada. La estructura de representación, su invariante de representación y su función de abstracción son las siguientes.

mapa se representa con map

donde **map** es $\text{tupla}(\text{tamano} : \text{nat}, \text{casilleros} : \text{vec}(\text{vec}(\text{bool})),)$

Rep : $\text{mapa} \rightarrow \text{bool}$

Rep(*map*) $\equiv \text{true} \iff$ La longitud de *map.casilleros* es igual a *tamano* \wedge

La longitud del vector *m.casilleros* es igual a la de todo otro vector dentro de *el*) \wedge

Toda posición libre debe ser alcanzable por todo el resto de las posiciones libres a través de un camino de posiciones libres (conexo).

Abs : $\text{mapa map} \rightarrow \text{hab}$

$\{\text{Rep}(\text{map})\}$

Abs(*map*) $=_{\text{obs}} h : \text{hab} \mid m.\text{tamano} =_{\text{obs}} \text{tam}(h) \wedge_L$

$(\forall t : \text{tupla}(\text{nat}, \text{nat}))(0 \leq \Pi_1(t), \Pi_2(t) < \text{map.tamano} - 1 \Rightarrow_L$

$\text{libre}(h, t) =_{\text{obs}} \text{map.casilleros}[\Pi_1(t)][\Pi_2(t)])$

Algoritmos

Algoritmos del módulo

iTam(in $m : \text{map}$) $\rightarrow res : \text{nat}$

1: $res \leftarrow m.tamano$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iOcupar(in/out $m : \text{map}$, in $p : \text{pos}$)

1: $m[\Pi_1(p)][\Pi_2(p)] \leftarrow true$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

Justificación: El acceso a una posición de un vector y su modificación es $\Theta(1)$

iLibre(in $m : \text{map}$, in $p : \text{pos}$) $\rightarrow res : \text{bool}$

1: $res \leftarrow \neg m[\Pi_1(p)][\Pi_2(p)]$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

Justificación: El acceso a una posición de un vector es $\Theta(1)$

iNuevoMapa(in $n : \text{nat}$) $\rightarrow res : \text{map}$

1: // Inicializo el tamaño, el vector y el mapa.

2: $res \leftarrow \langle tamano : n, casilleros : Vacía() \rangle$

$\triangleright \Theta(1)$

3:

4: // Genero un vector de booleanos en falso con n posiciones.

5: $i \leftarrow 0$

$\triangleright \Theta(1)$

6: **while** $i < n$ **do**

$\triangleright \mathcal{O}(n^2)$

7: $v.AgregarAtras(false)$

$\triangleright \mathcal{O}(n)$

8: $i \leftarrow i + 1$

9: **end while**

10:

11: // Genero la matriz de n x n posiciones haciendo n copias del vector de booleanos antes creado.

12: $i \leftarrow 0$

13: **while** $i < n$ **do**

$\triangleright \mathcal{O}(n^2)$

14: $res.AgregarAtras(v.Copiar())$

$\triangleright \mathcal{O}(n)$

15: $i \leftarrow i + 1$

$\triangleright \Theta(1)$

16: **end while**

Complejidad: $\mathcal{O}(n^2)$

Justificación: Copiar un vector de n booleanos es $\mathcal{O}(n * copy(bool))$ y copiar un bool es $\Theta(1)$. Luego, agregar n veces la copia del vector es $\mathcal{O}(n^2)$, puesto que AgregarAtras es $\mathcal{O}(n)$ y copiarlo es $\mathcal{O}(n)$ por lo antes visto. Luego la complejidad de la operación de la línea 10 es $\mathcal{O}(n)$ y, por lo tanto, todo el while es $\mathcal{O}(n^2)$.

4. Módulo Dirección

El módulo Dirección provee una dirección y una función que permite invertir las mismas.

Interfaz

generos: `dir`.

se explica con: DIRECCIÓN.

Operaciones básicas de Dirección

`ARRIBA()` $\rightarrow res : dir$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} \uparrow\}$

Complejidad: $\Theta(1)$

Descripción: genera la dirección arriba.

`ABAJO()` $\rightarrow res : dir$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} \downarrow\}$

Complejidad: $\Theta(1)$

Descripción: genera la dirección abajo.

`IZQUIERDA()` $\rightarrow res : dir$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} \leftarrow\}$

Complejidad: $\Theta(1)$

Descripción: genera la dirección izquierda.

`DERECHA()` $\rightarrow res : dir$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} \rightarrow\}$

Complejidad: $\Theta(1)$

Descripción: genera la dirección derecha.

`INVERTIR(in/out $d : dir$)`

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} invertir(d)\}$

Complejidad: $\Theta(1)$

Descripción: invierte la dirección.

Representación

El objetivo de este módulo es implementar una dirección utilizando strings. La estructura de representación, su invariante de representación y su función de abstracción son las siguientes.

Representación de Dirección

`dir` se representa con `string`

`Rep` : `dir` \rightarrow `bool`

`Rep(d)` $\equiv true \iff$

$d =_{obs} "arriba" \vee$

$d =_{obs} "abajo" \vee$

$d =_{obs} "izquierda" \vee$

$d =_{obs} "derecha"$

`Abs` : `dir d` \rightarrow `dir`

`Abs(d)` $=_{obs} d_{tad} : dir \mid (d =_{obs} "arriba" \wedge d_{tad} =_{obs} \uparrow) \vee$
 $(d =_{obs} "abajo" \wedge d_{tad} =_{obs} \downarrow) \vee$
 $(d =_{obs} "izquierda" \wedge d_{tad} =_{obs} \leftarrow) \vee$
 $(d =_{obs} "derecha" \wedge d_{tad} =_{obs} \rightarrow)$

$\{Rep(d)\}$

Algoritmos

Algoritmos del módulo

iArriba() $\rightarrow res : \text{dir}$ 1: $res \leftarrow \text{"arriba"}$ $\triangleright \Theta(1)$ Complejidad: $\Theta(1)$

iAbajo() $\rightarrow res : \text{dir}$ 1: $res \leftarrow \text{"abajo"}$ $\triangleright \Theta(1)$ Complejidad: $\Theta(1)$

iIzquierda() $\rightarrow res : \text{dir}$ 1: $res \leftarrow \text{"izquierda"}$ $\triangleright \Theta(1)$ Complejidad: $\Theta(1)$

iDerecha() $\rightarrow res : \text{dir}$ 1: $res \leftarrow \text{"derecha"}$ $\triangleright \Theta(1)$ Complejidad: $\Theta(1)$

iInvertir(in/out $d : \text{dir}$)1: $\text{switch}(d)$ $\triangleright \Theta(1)$ 2: case "arriba" : 3: $d \leftarrow \text{"abajo"}$ 4: case "abajo" : 5: $d \leftarrow \text{"arriba"}$ 6: $\text{case "izquierda" :}$ 7: $d \leftarrow \text{"derecha"}$ 8: case "derecha" : 9: $d \leftarrow \text{"izquierda"}$ Complejidad: $\Theta(1)$

5. Módulo Acción

El módulo Acción provee una acción y una funciones que permiten operar con acciones y eventos.

Interfaz

generos: accion.

se explica con: ACCIÓN.

Operaciones básicas de Acción

MOVER(in d : dir) $\rightarrow res$: accion

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} mover(d)\}$

Complejidad: $\Theta(1)$

Descripción: genera una acción de mover en la dirección especificada.

PASAR() $\rightarrow res$: accion

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} pasar\}$

Complejidad: $\Theta(1)$

Descripción: genera la acción de pasar.

DISPARAR() $\rightarrow res$: accion

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} disparar\}$

Complejidad: $\Theta(1)$

Descripción: genera la acción de disparar.

APLICAR(in a : acción, in j : juego, in e : evento) $\rightarrow res$: evento

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} aplicar(a, j, e)\}$

Complejidad: $\Theta(1)$

Descripción: genera el evento a partir de la acción a realizar.

INVERTIR(in e : evento) $\rightarrow res$: evento

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} invertir(e)\}$

Complejidad: $\Theta(1)$

Descripción: invierte un evento.

INVERSA(in/out es : vector(evento))

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} inversa(es)\}$

Complejidad: $\Theta(1)$

Descripción: genera una secuencia que contiene a la inicial, le suma 5 pasos de espera y le agrega la secuencia original invertida.

Representación

Representación de Acción

El objetivo de este módulo es implementar una acción utilizando una tupla de string y dirección. La estructura de representación, su invariante de representación y su función de abstracción son las siguientes.

acción se representa con a

donde a es `tupla(acción: string, dir: dir)`

$Rep : acción \rightarrow bool$

$$\begin{aligned}
\text{Rep}(a) &\equiv \text{true} \iff \\
&\quad \text{a.acción} =_{\text{obs}} \text{"disparar"} \vee \\
&\quad \text{a.acción} =_{\text{obs}} \text{"pasar"} \vee \\
&\quad \text{a.acción} =_{\text{obs}} \text{"mover"} \\
\text{Abs} : \text{acción } a &\longrightarrow \text{acción} && \{\text{Rep}(a)\} \\
\text{Abs}(a) =_{\text{obs}} a_{\text{tad}} : \text{acción} &| (\text{a.acción} =_{\text{obs}} \text{"disparar"} \wedge \text{esDisparar}(a_{\text{tad}})) \vee \\
&\quad (\text{a.acción} =_{\text{obs}} \text{"pasar"} \wedge \text{esPasar}(a_{\text{tad}})) \vee \\
&\quad ((\text{a.acción} =_{\text{obs}} \text{"mover"} \wedge \text{esMover}(a_{\text{tad}})) \wedge_{\text{L}} \text{a.dir} =_{\text{obs}} \text{direccion}(a_{\text{tad}}))
\end{aligned}$$

Algoritmos

Para las acciones que no tienen dirección, les definimos la dirección *Arriba()*.
Esto no importa ya que la dirección es ignorada en general para esas acciones.

Algoritmos del módulo

iPasar() $\rightarrow res : \text{acción}$
1: $res \leftarrow \langle \text{accion} : \text{"pasar"}, \text{dir} : \text{Arriba}() \rangle$ $\triangleright \Theta(1)$
Complejidad: $\Theta(1)$

iDisparar() $\rightarrow res : \text{acción}$
1: $res \leftarrow \langle \text{accion} : \text{"disparar"}, \text{dir} : \text{Arriba}() \rangle$ $\triangleright \Theta(1)$
Complejidad: $\Theta(1)$

iMover(in d: dir) $\rightarrow res : \text{acción}$
1: $res \leftarrow \langle \text{accion} : \text{"mover"}, \text{dir} : d \rangle$ $\triangleright \Theta(1)$
Complejidad: $\Theta(1)$

iInvertir(in e: evento) $\rightarrow res : \text{evento}$
1: $res \leftarrow \langle \text{pos} : e.\text{pos}, \text{dir} : \text{Invertir}(e.\text{dir}), \text{disparo?} : e.\text{disparo?} \rangle$ $\triangleright \Theta(1)$
Complejidad: $\Theta(1)$
)

iInversa(in/out es: vector(evento))

```

1: // El resultado deseado es el siguiente
2: // es + nada + nada + nada + nada + nada + inversa(es)
3:
4: // Me guardo la long original ya que lo voy a modificar por referencia
5: longOriginal ← Longitud(es) ▷ Θ(1)
6:
7: // Creo un evento que sea pasar y lo agrego 5 veces
8: eventoPasar ← ⟨pos : Ultimo(es).pos, dir : Ultimo(es).dir, disparo? : false⟩ ▷ Θ(1)
9: for (i = 0, ... , 4) do ▷ O(long(es) * 5)
10:   AgregarAtras(es, eventoPasar) ▷ O(long(es))
11: end for
12:
13: // Recorro los eventos de la secuencia original de atrás para adelante,
14: // invirtiendolos y agregándolos al final
15: for (i = longOriginal - 1, ... , 0) do ▷ O(long(es)2)
16:   AgregarAtras(es, invertir(es[i])) ▷ O(long(es))
17: end for

```

Complejidad: $O(\text{long}(\text{es})^2)$

Justificación: Crear una tupla y acceder al vector es $\Theta(1)$. $O(\text{long}(\text{es}) * 5) + O(\text{long}(\text{es})^2) = O(\text{long}(\text{es})^2)$.

iAplicar(in a: acción, in j: juego, in e: evento) → res: evento

```

1: if (a.accion = disparar) ▷ Θ(1)
2:   then res ← ⟨pos : e.pos, dir : e.dir, disparo? : true⟩ ▷ Θ(1)
3: end if
4:
5: if (a.accion = pasar) ▷ Θ(1)
6:   then res ← ⟨pos : e.pos, dir : e.dir, disparo? : false⟩ ▷ Θ(1)
7: end if
8:
9: if (a.accion = mover) then ▷ Θ(1)
10:   if (a.dir = Arriba() ∧ Π1(e.pos) + 1 < Tam(j.mapa) ∧L
11:     Libre(j.mapa, ⟨Π1(e.pos) + 1, Π2(e.pos)⟩) ) ▷ Θ(1)
12:     then res ← ⟨pos : ⟨Π1(e.pos) + 1, Π2(e.pos)⟩, dir : a.dir, disparo? : false⟩ ▷ Θ(1)
13:     else res ← ⟨pos : e.pos, dir : a.dir, disparo? : false⟩
14:   end if
15:   if (a.dir = Abajo() ∧ Π1(e.pos) - 1 < Tam(j.mapa) ∧L
16:     Libre(j.mapa, ⟨Π1(e.pos) - 1, Π2(e.pos)⟩) ) ▷ Θ(1)
17:     then res ← ⟨pos : ⟨Π1(e.pos) - 1, Π2(e.pos)⟩, dir : a.dir, disparo? : false⟩ ▷ Θ(1)
18:     else res ← ⟨pos : e.pos, dir : a.dir, disparo? : false⟩
19:   end if
20:   if (a.dir = Derecha() ∧ Π2(e.pos) + 1 < Tam(j.mapa) ∧L
21:     Libre(j.mapa, ⟨Π1(e.pos), Π2(e.pos) + 1⟩) ) ▷ Θ(1)
22:     then res ← ⟨pos : ⟨Π1(e.pos), Π2(e.pos) + 1⟩, dir : a.dir, disparo? : false⟩ ▷ Θ(1)
23:     else res ← ⟨pos : e.pos, dir : a.dir, disparo? : false⟩
24:   end if
25:   if (a.dir = Izquierda() ∧ Π2(e.pos) - 1 < Tam(j.mapa) ∧L
26:     Libre(j.mapa, ⟨Π1(e.pos), Π2(e.pos) - 1⟩) ) ▷ Θ(1)
27:     then res ← ⟨pos : ⟨Π1(e.pos), Π2(e.pos) - 1⟩, dir : a.dir, disparo? : false⟩ ▷ Θ(1)
28:     else res ← ⟨pos : e.pos, dir : a.dir, disparo? : false⟩
29:   end if
30: end if

```

Complejidad: $\Theta(1)$

Justificación: Crear una tupla, comparar sus elementos y las operaciones del mapa son $\Theta(1)$.
