Apunte de Módulos Básicos (v. 0.3α)

Algoritmos y Estructuras de Datos II, DC, UBA.

1er cuatrimestre de 2019

Índice

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

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

géneros: `diccTrie`($string, \alpha$).

Operaciones básicas de diccionario

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

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

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

SIGNIFICADO(**in** d: `diccTrie(`$string$`, `α`)`, **in** k: $string$) $\rightarrow$ res : α
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ 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.

CLAVES(**in** d: `diccTrie(`$string$`, `α`)`) $\rightarrow$ res : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ claves(d)}
Complejidad: $\Theta(1)$
Descripción: devuelve las claves definidas en el diccionario

Representación

Representación del diccionario

`diccTrie`($string, \alpha$) **se representa con** `dic`

donde `dic` es `tupla`(*raiz*: `puntero(nodo)`,
claves: `conj(string)`)

donde `nodo` es `tupla`(*significado*: `puntero(`α`)`,
siguientes: `arreglo(puntero(nodo))[256]`)

Rep : dic $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$
(Los nodos del diccionario (excepto la raiz) tienen un unico padre. Es decir, no hay dos Nodos en la estructura que tengan punteros iguales en los siguientes del Nodo. $\wedge$
La raiz 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. $\wedge$
Un s string pertenece a d.claves $\Longleftrightarrow$ estáDefinido(s, d.claves))

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

Abs : dic e $\longrightarrow$ dicc(*string*, α) {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: dicc(*string*, α) |
$(\forall$ :: s string)(def?(s,d) $=_{\text{obs}}$ estáDefinido(e.raiz, s)) $\wedge$
$(\forall$ :: s string)(def?(s,d) $\Rightarrow_{\text{L}}$ obtener(s, d) $=_{\text{obs}}$ significado(e.raiz, s)) $\wedge$
claves(d) $=_{\text{obs}}$ e.claves

estáDefinido(r, s) $\equiv$ **if** vacia?(s)
then r $\rightarrow$ significado $\neq$ NULL
else r $\rightarrow$ sigueintes[int(prim(s))] $\neq$ NULL $\wedge_{\text{L}}$ estáDefinido(r.siguientes[int(prim(s))], fin(s)) **fi**

significado(r,s) $\equiv$ **if** vacia?(s)
then r $\rightarrow$ significado
else significado(r.siguientes[int(prim(s))], fin(s)) **fi**

Algoritmos

---

iVacía() $\rightarrow$ *res* : &dic

1: // Le asigna un nuevo nodo a la raiz
2: $res \leftarrow \langle raiz : nuevoNodo() \rangle$ ▷ $\Theta(1)$

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

---

iSignificado(**in**/**out** d: `dic`, **in** k: *string*)$\rightarrow$ *res* : &α

1: $Nodo\ actual \leftarrow d.raiz$ ▷ $\Theta(1)$
2: **for** ($char\ c\ :\ k$) **do** ▷ $\mathcal{O}(|k|)$
3: $actual \leftarrow (actual \rightarrow siguientes[toInt(c)])$ ▷ $\Theta(1)$
4: **end for**
5: $res \leftarrow *(actual \rightarrow significado)$ ▷ $\Theta(1)$

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: dic, **in** k: $string$) $\rightarrow$ res : bool

```
1: Nodo actual ← d.raiz                                                    ▷ Θ(1)
2: for (char c : k) do                                                     ▷ O(|k|)
3:     if (actual → siguientes[toInt(c)] ≠ NULL)                           ▷ Θ(1)
4:         then actual ← (actual → siguientes[toInt(c)])                   ▷ Θ(1)
5:         else res ← false                                                ▷ Θ(1)
6:     end if
7: end for
8: res ← ((actual → significado) ≠ NULL)                                   ▷ Θ(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 ejecutaran dichas asignaciones |k| veces como máximo. Luego la complejidad será $\mathcal{O}(|k|)$.

iDefinir(**in/out** d: dic, **in** k: $string$, **in** s: α) $\rightarrow$ res : $\&\alpha$

```
1: Nodo actual ← d.raiz
2: for (char c : k) do                                                     ▷ Θ(|k|)
3:     // Si no tengo siguiente, lo creo
4:     if (actual → siguientes[toInt(c)] == NULL) then                     ▷ Θ(1)
5:         actual → siguientes[toInt(c)] = nuevoNodo()                     ▷ Θ(1)
6:     end if
7:     actual ← (actual → siguientes[toInt(c)])                            ▷ Θ(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 ← s                                                                ▷ Θ(copy(s))
13:
14: // Si el significado no está definido, agrego la nueva clave al conjunto de claves
15: if (actual → significado == NULL) then                                 ▷ Θ(1)
16: // Como precondicion, se que no existe asi que la agrego rapido
17: AgregarRapido(e.claves, k)                                             ▷ Θ(copy(k))
18: end if
19:
20: // Asigno al significado del nodo el puntero creado con s y libero la memoria que contenía al valor anterior.
21: (actual → significado) ← &sig                                          ▷ Θ(1)
22:
23: // Devuelvo por referencia el significado.
24: res ← 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 $\Theta(1)$.

inuevoNodo() $\rightarrow$ res : puntero(nodo) ▷ Función privada que crea un nuevo nodo

```
1: // Reserva la memoria para un nuevo nodo con significado null y siguientes vacios
2: res ← &⟨significado : NULL, siguientes : arreglo_estatico[256] de puntero(Nodo)⟩   ▷ Θ(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 : `dic`)→ res : &conj(string)

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

Complejidad: $\Theta(1)$
Justificación: Devolver por referencia un conjunto es $\Theta(1)$.

2. Módulo Juego

Aqui 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 $\equiv$ $\{\neg vacio(pjs) \wedge (\forall e : evento)(est?(e, eventosFan) \Rightarrow_{\text{L}} e.pos \in libres(m))\}$
Post $\equiv$ $\{res =_{\text{obs}} nuevoJuego(m, pjs, eventosFan)\}$
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 $\equiv$ $\{res =_{\text{obs}} pasar(j)\}$
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 $\equiv$ $\{pj \in jugadores(j) \wedge_{\text{L}} jugadorVivo(pj, j) \wedge \neg esPasar(a)\}$
Post $\equiv$ $\{res =_{\text{obs}} step(j, a, pj)\}$
Complejidad: $\Theta(?)$
Descripción: actualiza con la acción a del jugador pj.

JUGADORESVIVOS(**in** j : `juego`) $\rightarrow$ res : `conj(puntero(infoPJ))`
Pre $\equiv$ {true}
Post $\equiv$ $\{(\forall p : puntero(infoPJ))(p \in res \Rightarrow_{\text{L}}$
$(p \rightarrow id \in jugadores(j)) \wedge_{\text{L}}$
$(p \rightarrow vivo? \wedge jugadorVivo(p \rightarrow id, j)) \wedge$
$((\forall e : evento)(e \in p \rightarrow eventos \Rightarrow_{\text{L}}$
$(e.pos =_{\text{obs}} posJugador(p \rightarrow id, j)) \wedge$
$(e.dir =_{\text{obs}} dirJugador(p \rightarrow id, j))))\}$
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 $\equiv$ $\{res =_{\text{obs}} fantasmaEspecial(j)\}$
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 $\equiv$ $\{fantasmaValido(j, res) \wedge_{\text{L}}$
$((\forall f : infoFan)(f \in res \Rightarrow_{\text{L}} disparando(f.eventos, step(j))))\}$
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 $\equiv$ {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):
$(\forall f : infoFan)(f \in res \Rightarrow_{\text{L}}$
$(f.eventos \in fantasmas(j)) \wedge_{\text{L}}$
$(fantasmaVivo(f.eventos, j)) \wedge$
$((\forall e : evento)(e \in f.eventos \Rightarrow_{\text{L}}$
$(e.pos =_{\text{obs}} posFantasma(f.eventos, j)) \wedge$
$(e.dir =_{\text{obs}} dirFantasma(f.eventos, j))))$

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)))`,
disparosFanUltimoPaso: `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*: `lista(evento)`,
vivo?: `bool`,
infoActual: `itConj(infoActualPJ)`)

donde `infoActualPJ` es `tupla`(*identidad*: `string`,
posicion: `pos`,
direccion: `dir`)

donde `infoFan` es `tupla`(*eventos*: `vector(evento)`,
vivo?: `bool`,
infoActual: `itConj(infoActualFan)`)

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

Rep : mapa $\longrightarrow$ bool
Rep(m) $\equiv$ true $\Longleftrightarrow$

Abs : mapa m $\longrightarrow$ hab {Rep(m)}
Abs(m) $=_{\text{obs}}$ 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, ▷ $\Theta(1)$
5: $ronda$: 0, ▷ $\Theta(1)$
6:
7: // Seteo el mapa
8: $mapa$: m, ▷ $\Theta(copy(m))$
9:
10: // Inicializo el mapa de disparos con el mismo tamaño que el mapa
11: $mapaDisparos$: $arreglo(arreglo(tupla(nat, nat))[Tam(m)])[Tam(m)]$, ▷ $\Theta(Tam(m)^2)$
12: $disparosFanUltimoPaso$: $Vacio()$, ▷ $\Theta(1)$
13:
14: // Inicializo estructuras de jugadores y fantasmas como vacías
15: $infoActualJugadoresVivos$: $Vacio()$, ▷ $\Theta(1)$
16: $infoJugadoresVivos$: $Vacio()$, ▷ $\Theta(1)$
17: $infoJugadores$: $Vacia()$, ▷ $\Theta(1)$
18: $infoFantasmas$: $Vacio()$, ▷ $\Theta(1)$
19: $infoActualFantasmasVivos$: $Vacio()$, ▷ $\Theta(1)$
20: $infoFantasmasVivos$: $Vacia()$, ▷ $\Theta(1)$
21: $infoFantasmaEspecial$: $CrearIt(Vacio())$ ▷ $\Theta(1)$
22: ⟩
23:
24: // Inicializo los jugadores
25: $iIniciarJugadores(res,\ m,\ pjs)$ ▷ $\Theta(\#pjs * |k| + locJugadores)$
26:
27: // Inicializo los fantasmas
28: $iIniciarFantasmas(res,\ eventosFan)$ ▷ $\Theta(long(eventosFan))$

Complejidad: $\Theta(?)$
Justificación: Copiar y generar iteradores, tuplas y conjuntos es $\Theta(1)$.

---

iIniciarJugadores(**in/out** 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, pjs)
3:
4:  // Obtengo las posiciones y direcciones de jugadores
5:  localPJs ← localizarJugadores(m, pjs)                                   ▷ Θ(locJugadores)
6:
7:  // Lleno las estructuras de jugadores
8:  for (pj, localizacion : localPJs) do                                    ▷ Θ(#pjs ∗ (|k| + copy(info)))
9:      // Creo la infoActual y la agrego a su conjunto
10:     infoActual ← ⟨identidad : pj, posicion : localizacion.pos, direccion : localizacion.dir⟩   ▷ Θ(1)
11:     itInfoActual ← AgregarRapido(j.infoActualJugadoresVivos, infoActual)   ▷ Θ(copy(infoActual))
12:
13:     // Creo la infoPJ con la actual
14:     info ← iNuevaInfoPJ(localizacion, itInfoActual)                    ▷ Θ(1)
15:     // La agrego al trie y me guardo el puntero a la info guardada
16:     infoPtr ← &Definir(j.infoJugadores, pj, info)                      ▷ Θ(|pj| + copy(info))
17:
18:     // Agrego al conjunto de jugadores vivos el puntero a la info del PJ
19:     AgregarRapido(j.infoJugadoresVivos, infoPtr)                       ▷ Θ(copy(infoActual))
20: end for
```

Complejidad: $\Theta(?)$
Justificación: Copiar y generar iteradores, tuplas y conjuntos es $\Theta(1)$. Definir es $\Theta(|pj|)$ ya que copiar la tupla de info es $\Theta(1)$. Luego, definir #pjs es $\Theta(\#pjs * |pj|)$. Finalmente, la complejidad de todo el algoritmo es $\Theta(\#pjs * |k| + locJugadores)$.

---

iNuevaInfoPJ(**in** $localizacion$: `tupla(pos, dir)`, `in` $itInfoActual$: `itConj(infoActualPJ)`) → res : infoPJ ▷ Función privada

```
1:  // Creo el evento
2:  evento ← ⟨                                                             ▷ Θ(1)
3:      pos : localizacion.pos,
4:      dir : localizacion.dir,
5:      disparo? : false
6:  ⟩
7:
8:  // Creo una lista con él
9:  evts ← Vacia()                                                         ▷ Θ(1)
10: AgregarAtras(evts, evento)                                             ▷ Θ(copy(evento))
11:
12: // Armo la infoPJ
13: res ← ⟨                                                                ▷ Θ(1)
14:     eventos : evts
15:     vivo? : true
16:     infoActual : itInfoActual
17: ⟩
```

Complejidad: $\Theta(1)$
Justificación: Copiar y generar iteradores, tuplas y conjuntos es $\Theta(1)$.

iIniciarFantasmas(**in/out** 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 ← ⟨posicion : eventosFan[0].pos, direccion : eventosFan[0].dir⟩        ▷ Θ(1)
4: itInfoActualFan ← AgregarRapido(infoActualFan, j.infoActualFantasmasVivos)   ▷ Θ(copy(infoActual))
5:
6: // Hago que el fantasma especial sea este
7: j.infoFantasmaEspecial ← itInfoActualFan                                          ▷ Θ(1)
8:
9: // Creo la infoFan con la actual
10: infoFan ← ⟨infoActual : itInfoActualFan, eventos : eventosFan⟩                   ▷ Θ(1)
11: // La agrego al conjunto de información de fantasmas y me guardo su iterador
12: itInfoFan ← AgregarRapido(infoFan, j.infoFantasmas)                  ▷ Θ(copy(infoFan))
13:
14: // Agrego al conjunto de fantasmas vivos el interador a la info del Fan
15: AgregarRapido(itInfoFan, j.infoFantasmasVivos)                  ▷ Θ(copy(itInfoFan))
```

Complejidad: $\Theta(long(eventosFan))$
Justificación: Copiar y generar iteradores, tuplas y conjuntos es $\Theta(1)$. Copiar infoActual implica copiar un vector de eventos, que es $\Theta(long(eventosFan) * copy(evento)) = \Theta(long(eventosFan))$

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

```
1: // Incremento el paso
2: j.paso ← j.paso + 1                                                               ▷ Θ(1)
3:
4: // Reinicio los disparos de fantasmas
5: iReiniciarDisparosFan(j)                                                  ▷ O(#fv * m)
6:
7: // Actualizo las acciones de los fantasmas,
8: // actualizando el mapa de disparos si disparan.
9: iActualizarFantasmas(j)                                                   ▷ O(#fv * m)
10:
11: // Veo que jugadores mueren
12: iChequearMuerteJugadores(j)                                               ▷ Θ(#jv)
```

Complejidad: $\mathcal{O}(2(\#fv * m) + \#jv) \in \mathcal{O}(\#fv * m + \#jv)$

iReiniciarDisparosFan(**in/out** j : `estr`) ▷ Funcion privada

```
1: // Vacío la lista de disparos del ultimo paso
2: // Al asignarle vacío, se libera la memoria que ocupaba anteriormente.
3: j.disparosFanUltimoPaso ← Vacio()                          ▷ Θ(Tam(j.disparosFanUltimoPaso))
```

Complejidad: $\Theta(\#fv * m)$
Justificación: Vaciar un arreglo de longitud n es $\Theta(n)$. Este arreglo en particular siempre tendrá longitud $\Theta(\#fv * m)$, ya que se llena con los disparos de los fantasmas que estén vivos en ese paso. Depende de que tan fino uno hile con el manejo de memoria, podría llegar a tomarse como $\Theta(1)$.

iActualizarFantasmas(**in/out** j : `estr`)

```
1:  // Recorro los fantasmas vivos
2:  for (itInfoFan : j.infoFantasmasVivos) do                          ▷ Θ(#fv)
3:      // Obtengo la información del fantasma
4:      infoFan ← Siguiente(itInfoFan)                                 ▷ Θ(1)
5:
6:      // Actualizo su información actual, obteniendo el evento actual
7:      eventoActual ← iActualizarFan(infoFan, j.paso)                 ▷ Θ(1)
8:
9:      // Si dispara, agrego su disparo a los del paso
10:     if (eventoActual.dispara?)
11:     then iAgregarDisparo(j, eventoActual.pos, eventoActual.dir, true)   ▷ O(m)
12:     end if
13: end for
```

Complejidad: $\mathcal{O}(\#fv * m)$
Justificación: Tomar referencia del elemento al que apunta un iterador y copiar un evento es $\Theta(1)$.

iActualizarFan(**in/out** $info$: `infoFan`, **in** $paso$: `nat`) $\rightarrow$ res : evento ▷ Funcion privada

```
1:  // Obtengo el evento actual
2:  eventoActual ← iEventoActualFan(infoFan, paso)                     ▷ Θ(1)
3:
4:  // Obtengo el iterador a la info actual
5:  itInfoActual ← info.infoActual                                     ▷ Θ(1)
6:
7:  // La actualizo con el eventoActual
8:  Siguiente(itInfoActual).posicion ← eventoActual.pos                ▷ Θ(1)
9:  Siguiente(itInfoActual).direccion ← eventoActual.dir               ▷ Θ(1)
10:
11: // Devuelvo el evento actual
12: res ← eventoActual
```

Complejidad: $\Theta(1)$
Justificación: Actualizar el iterador y generar el eventoActual es $\Theta(1)$.

iChequearMuerteJugadores(**in/out** j : `estr`) ▷ Funcion privada

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

Complejidad: $\Theta(\#jv)$
Justificación: Crear punteros, iteradores y tuplas es $\Theta(1)$.

15:

iPJAfectadoPorDisparo?(**in** j : `estr`, **in** pos : `pos`) $\rightarrow$ 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: $pasoDispFan \leftarrow j.mapaDisparos[pos.x][pos.y].pasoDispFan$ ▷ $\Theta(1)$
4:
5: // Estará afectado si el paso del disparo del fantasma es igual al actual
6: $afectado? \leftarrow (pasoDispFan == j.paso)$ ▷ $\Theta(1)$
7: $res \leftarrow afectado?$

Complejidad: $\Theta(1)$

---

iMuerePJ(**in/out** j : `estr`, **in/out** $itPJVivos$: `itConj(puntero(infoPJ))`) ▷ Funcion privada

1: // Obtengo su información
2: $infoPJ \leftarrow *Siguiente(itPJVivos)$ ▷ $\Theta(1)$
3:
4: // Lo seteo como muerto
5: $infoPJ.vivo? \leftarrow false$ ▷ $\Theta(1)$
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: $\Theta(1)$
Justificación: Las operaciones del iterador son $\Theta(1)$

iAgregarDisparo(**in/out** j : `estr`, **in** pos : `pos`, **in** dir : `dir`, **in** $esFan$: `bool`) ▷ Funcion privada

```
1:  // Copio pos para no modificar el original
2:  posCopy ← copy(pos)                                                    ▷ Θ(1)
3:
4:  // Parado desde posCopy en mapaDisparos, recorro hacia dir
5:  // hasta que me choco con un obstaculo o la pared.
6:  while Valida?(j.mapa, posCopy) ∧L Libre(j.mapa, posCopy) do            ▷ O(Tam(j.mapa))
7:      // Me guardo una referencia al pasoDisp correcto
8:      if esFan
9:      then pasoDisp ← mapaDisparos[pos.x][pos.y].pasoDispFan             ▷ Θ(1)
10:     else pasoDisp ← mapaDisparos[pos.x][pos.y].pasoDispPJ              ▷ Θ(1)
11:     end if
12:
13:     // Si no pasé ya por está posición con otro
14:     // (i.e si en el mapa de disparos no está ya el paso actual)
15:     if pasoDisp ≠ j.paso then
16:         // Le pongo el paso actual al paso en el que hubo un disparo
17:         pasoDisp ← j.paso                                              ▷ Θ(1)
18:
19:         // Si es un fantasma, agrego la posición al conjunto de disparos de fantasmas
20:         if esFan
21:         then AgregarRapido(j.disparosFanUltimoPaso, posCopy)           ▷ Θ(copy(posCopy))
22:         end if
23:     end if
24:
25:     // Avanzo la posición en esa dirección
26:     pos ← Avanzar(posCopy, dir)                                        ▷ Θ(1)
27: end while
```

Complejidad: $\mathcal{O}(m)$
Justificación: Copiar naturales, indexar y tomar referencia son $\Theta(1)$. Luego, como hacemos operaciones que son $\Theta(1)$ y lo hacemos a lo sumo Tam(j.mapa) veces, tenemos $\mathcal{O}(Tam(j.mapa))$.

---

iEventoActualFan(**in** $info$: `infoFan`, **in** $paso$: `nat`) → res : evento ▷ Funcion privada

```
1:  idx ← mod(j.paso, Longitud(info.eventos))                              ▷ Θ(1)
2:  res ← info.eventos[idx]                                                ▷ Θ(1)
```

Complejidad: $\Theta(1)$
Justificación: La operacions matemáticas y la indexación en un vector es $\Theta(1)$.

---

iEventoActualPJ(**in** $info$: `infoPJ`) → res : evento ▷ Funcion privada

```
1:  res ← Ultimo(info.eventos)                                             ▷ Θ(1)
```

Complejidad: $\Theta(1)$
Justificación: La operación Último sobre una lista es $\Theta(1)$

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

```
1:  // Incremento el paso
2:  j.paso ← j.paso + 1                                              ▷ Θ(1)
3:
4:  // Actualizo la información del jugador con la nueva acción,
5:  // y me guardo una referencia a su info modificada
6:  infoPJ ← iActualizarPJ(pj, a)                                    ▷ O(|j|)
7:  evtPJ ← iEventoActualPJ(infoPJ)                                  ▷ Θ(1)
8:
9:  // Reinicio los disparos de fantasmas
10: iReiniciarDisparosFan(j)                                         ▷ O(#fv ∗ m)
11:
12: // Modifico el mapa de disparos (solo si dispara)
13: iActualizarMapaDisparosConPJ(j, evtPJ)                           ▷ O(m)
14:
15: // Veo que fantasmas mueren, guardandome si murió el fantasma especial
16: murioFanEspecial ← iChequearMuerteFantasmas(j)                   ▷ Θ(#fv)
17:
18: // Si murió el fantasma especial, cambio de ronda
19: if murioFanEspecial then
20:     iNuevaRonda(j, infoPJ)                                       ▷ Θ(??)
21: else
22:     // Sigo en la misma ronda
23:     // Actualizo las acciones de los fantasmas,
24:     // actualizando el mapa de disparos si disparan.
25:     iActualizarFantasmas(j)                                      ▷ O(#fv ∗ m)
26:
27:     // Veo que jugadores mueren
28:     iChequearMuerteJugadores(j)                                  ▷ Θ(#jv)
29: end if
```

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

---

iActualizarMapaDisparosConPJ(**in/out** j : `estr`, **in** $evtPJ$: `evento`) ▷ Funcion privada

```
1: // Si dispara, agrego las posiciones afectadas por su disparo al mapa de disparos.
2: if evtPJ.dispara?
3: then iAgregarDisparo(j, evtPJ.pos, evtPJ.dir, false)              ▷ O(m)
4: end if
```

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

iChequearMuerteFantasmas(**in/out** j : `estr`) $\rightarrow$ res : bool ▷ Funcion privada

```
1:  // Me guardo si el fantasma especial muere
2:  muereFanEspecial ← false
3:
4:  // Recorro los fantasmas vivos con un iterador
5:  itFanVivos ← CrearIt(j.infoFantasmasVivos)                    ▷ Θ(#fv)
6:
7:  while HaySiguiente(itFanVivos) do
8:      // Obtengo su info
9:      infoFan ← Siguiente(Siguiente(itFanVivos))                ▷ Θ(1)
10:
11:     // Obtengo su evento actual
12:     eventoActual ← iEventoActualFan(infoFan, j.paso)          ▷ Θ(1)
13:
14:     if iFanAfectadoPorDisparo(j, eventoActual.pos)?           ▷ Θ(1)
15:     then muereFanEspecial ← iMuereFan(j, itFanVivos)          ▷ Θ(1)
16:     end if
17:
18:     // Avanzo el iterador
19:     Avanzar(itFanVivos)                                       ▷ Θ(1)
20: end while
21:
22: // Retorno si murio el fan especial
23: res ← muereFanEspecial
```

Complejidad: $\Theta(\#fv)$

iFanAfectadoPorDisparo(**in** j : `estr`, **in** pos : `pos`) $\rightarrow$ res : bool ▷ Funcion privada

```
1:  // El Fan estará afectado si en la posición en la que está hay un disparo de un PJ en el paso actual
2:  // Indexo por su posición en el mapa de disparos
3:  pasoDispPJ ← j.mapaDisparos[pos.x][pos.y].pasoDispPJ          ▷ Θ(1)
4:
5:  // Estará afectado si el paso del disparo del PJ es igual al actual
6:  afectado? ← (pasoDispPJ == j.paso)                            ▷ Θ(1)
7:  res ← afectado?
```

Complejidad: $\Theta(1)$

iMuereFan(**in**/**out** j : `estr`, **in**/**out** $itFanVivos$: `itConj(itConj(infoFan))`) $\rightarrow$ res : bool ▷ Funcion privada

```
1: // Obtengo la info
2: infoFan ← Siguiente(Siguiente(itFanVivos))                          ▷ Θ(1)
3:
4: // Lo seteo como muerto
5: infoFan.vivo? ← false                                               ▷ Θ(1)
6:
7: // Obtengo la info actual
8: itInfoActual ← infoFan.infoActual                                   ▷ Θ(1)
9:
10: // Veo si es el fantasma especial
11: eraFanEspecial ← (itInfoActual == j.infoFantasmaEspecial)          ▷ Θ(1)
12:
13: // Lo borro de infoActualFantasmasVivos
14: EliminarSiguiente(itInfoActual)                                    ▷ Θ(1)
15:
16: // Lo borro de infoFantasmasVivos
17: EliminarSiguiente(itFanVivos)                                      ▷ Θ(1)
18:
19: // Retorno si era el fantasma especial
20: res ← eraFanEspecial
```

Complejidad: $\Theta(1)$

iNuevaRonda(**in** j : `estr`, **in** $pjMatoFanEspecial$: `infoPJ`) ▷ Funcion privada

```
1: // Incremento la ronda
2: // Reinicio el paso
3:
4: // Reinicio el mapa de disparos y los disparos de los fantasmas
5:
6: //// Reinicio Fantasmas
7: // Creo un nuevo fantasma (secu de eventos) conviertiendo la lista enlazada de acciones del PJ que mató al
   fantasma especial en un vector.
8: // Hago que ese fantasma sea el nuevo especial
9:
10: // Recorro infoFantasmas, por cada fantasma
11:     // Obtengo infoFan
12:     // Lo seteo como vivo
13:
14:     // Creo su infoActual y la agrego a infoActualFantasmasVivos, guardandome su iterador
15:     // Se que infoActual será eventualmente consistente al realizarse un paso
16:
17:     // Le seteo el iterador a infoFan
18:
19:     // Agrego un iterador a su info a infoFantasmasVivos
20:
21: //// Reinicio Jugadores
22: // Obtengo sus localizaciones
23: // Por cada clave y valor de las localizaciones,
24:     // Obtengo infoPJ del trie
25:     // Reinicio los eventos
26:     // Agrego un primer evento con la localización obtenida
27:     // Lo seteo como vivo
28: g
29:     // Creo una info actual
30:     // La agrego a infoActualJugadoresVivos y me guardo itInfoActual
31:     // Le seteo infoActual a infoPJ con el iterador
32:
33:     // Agrego un puntero a la infoPJ a infoJugadoresVivos
```

Complejidad: $\Theta(??)$

iActualizarPJ(**in** pj : `jugador`, **in** a : `accion`) $\rightarrow$ res : infoPJ ▷ Funcion privada

1: // Busco la información del PJ
2: $infoPJ \leftarrow Obtener(j.infoJugadores,\ pj)$ ▷ $\mathcal{O}(|j|)$
3:
4: // Genero un evento con la acción y el evento anterior (el actual)
5: $evtPJ \leftarrow Aplicar(a,\ j,\ iEventoActualPJ(infoPJ))$ ▷ $\Theta(1)$
6:
7: // Agrego el evento al jugador
8: $AgregarAtras(infoPJ.eventos,\ evtPJ)$ ▷ $\Theta(1)$
9:
10: // Obtengo su información actual
11: $itInfoActual \leftarrow infoPJ.infoActual$ ▷ $\Theta(1)$
12:
13: // La actualizo
14: $Siguiente(itInfoActual).posicion \leftarrow evt.pos$ ▷ $\Theta(1)$
15: $Siguiente(itInfoActual).direccion \leftarrow evt.dir$ ▷ $\Theta(1)$
16:
17: // Devuelvo la info del pj
18: $res \leftarrow infoPJ$

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

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: `nat`) $\rightarrow$ res : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} nuevaHab(n)\}$
Complejidad: $\Theta(n^2)$
Descripción: genera un mapa de tamaño n x n.

OCUPAR(**in/out** m: `mapa`, **in** c: $tupla(int, int)$)
Pre $\equiv$ $\{m =_{\text{obs}} m_0 \wedge c \in casilleros(m) \wedge_{\text{L}} libre(m, c) \wedge alcanzan(libres(m) - c, libres(m) - c, m)\}$
Post $\equiv$ $\{m =_{\text{obs}} ocupar(c, m_0)\}$
Complejidad: $\Theta(1)$
Descripción: ocupa una posición del mapa siempre y cuando éste no deje de ser conexo.

TAM(**in** m: `mapa`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} tam(m)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el tamaño del mapa.

LIBRE(**in** m: `mapa`, **in** c: $tupla(int, int)$) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{c \in casilleros(m)\}$
Post $\equiv$ $\{res =_{\text{obs}} libre(c, m)\}$
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 `tupla`($tamano$: `nat`,$casilleros$: `vec(vec(bool))`,)

Rep : mapa $\longrightarrow$ bool
Rep(map) $\equiv$ true $\Longleftrightarrow$ 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 : mapa map $\longrightarrow$ hab {Rep(map)}
Abs(map) $=_{\text{obs}}$ h: hab | $m.tamano =_{\text{obs}} tam(h) \wedge_{\text{L}}$
$(\forall t$: tupla(nat,nat))$(0 \leq \Pi_1(t), \Pi_2(t) < map.tamano - 1 \Rightarrow_{\text{L}}$
libre(h, t) $=_{\text{obs}} map.casilleros[\Pi_1(t)][\Pi_2(t)])$

Algoritmos

Algoritmos del módulo

---

iTam(**in** m : `map`) $\rightarrow$ res : nat

1: $res \leftarrow m.tamano$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iOcupar(**in**/**out** m : `map`, **in** p : `pos`)

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

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

---

iLibre(**in** m : `map`, **in** p : `pos`) $\rightarrow$ res : bool

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

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

---

iNuevoMapa(**in** n : `nat`) $\rightarrow$ res : map

1: // Inicializo el tamaño, el vector y el mapa.
2: $res \leftarrow \langle tamano : n,\ casilleros : Vacia() \rangle$ ▷ $\Theta(1)$
3:
4: // Genero un vector de booleanos en falso con n posiciones.
5: $i \leftarrow 0$ ▷ $\Theta(1)$
6: **while** $i < n$ **do** ▷ $\mathcal{O}(n^2)$
7: $v.AgregarAtras(false)$ ▷ $\mathcal{O}(n)$
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** ▷ $\mathcal{O}(n^2)$
14: $res.AgregarAtras(v.Copiar())$ ▷ $\mathcal{O}(n)$
15: $i \leftarrow i + 1$ ▷ $\Theta(1)$
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 AgregarAtrás 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* $=_{\text{obs}}\uparrow$}
Complejidad: $\Theta(1)$
Descripción: genera la dirección arriba.

ABAJO() $\rightarrow$ *res* : `dir`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}\downarrow$}
Complejidad: $\Theta(1)$
Descripción: genera la dirección abajo.

IZQUIERDA() $\rightarrow$ *res* : `dir`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}\leftarrow$}
Complejidad: $\Theta(1)$
Descripción: genera la dirección izquierda.

DERECHA() $\rightarrow$ *res* : `dir`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}\rightarrow$}
Complejidad: $\Theta(1)$
Descripción: genera la dirección derecha.

INVERTIR(**in**/**out** *d*: `dir`)
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ *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`

$$
\begin{array}{l}
\text{Rep} : \text{dir} \longrightarrow \text{bool} \\
\text{Rep}(d) \equiv \text{true} \iff \\
\qquad \text{d} =_{\text{obs}} \text{"}arriba\text{"} \vee \\
\qquad \text{d} =_{\text{obs}} \text{"}abajo\text{"} \vee \\
\qquad \text{d} =_{\text{obs}} \text{"}izquierda\text{"} \vee \\
\qquad \text{d} =_{\text{obs}} \text{"}derecha\text{"}
\end{array}
$$

$$
\begin{array}{ll}
\text{Abs} : \text{dir } d \longrightarrow \text{dir} & \{\text{Rep}(d)\} \\
\text{Abs}(d) =_{\text{obs}} \text{d}_{tad}\text{: dir} \mid (\text{d} =_{\text{obs}} \text{"}arriba\text{"} \wedge \text{d}_{tad} =_{\text{obs}} \uparrow) \vee & \\
\qquad (\text{d} =_{\text{obs}} \text{"}abajo\text{"} \wedge \text{d}_{tad} =_{\text{obs}} \uparrow) \vee & \\
\qquad (\text{d} =_{\text{obs}} \text{"}izquierda\text{"} \wedge \text{d}_{tad} =_{\text{obs}} \leftarrow) \vee & \\
\qquad (\text{d} =_{\text{obs}} \text{"}derecha\text{"} \wedge \text{d}_{tad} =_{\text{obs}} \rightarrow) &
\end{array}
$$

Algoritmos

Algoritmos del módulo

iArriba() → *res* : dir
1: *res* ← "*arriba*" ▷ Θ(1)

Complejidad: Θ(1)

iAbajo() → *res* : dir
1: *res* ← "*abajo*" ▷ Θ(1)

Complejidad: Θ(1)

iIzquierda() → *res* : dir
1: *res* ← "*izquierda*" ▷ Θ(1)

Complejidad: Θ(1)

iDerecha() → *res* : dir
1: *res* ← "*derecha*" ▷ Θ(1)

Complejidad: Θ(1)

iInvertir(**in**/**out** *d* : `dir`)
1: *switch*(*d*) ▷ Θ(1)
2: *case* "*arriba*" :
3: *d* ← "*abajo*"
4: *case* "*abajo*" :
5: *d* ← "*arriba*"
6: *case* "*izquierda*" :
7: *d* ← "*derecha*"
8: *case* "*derecha*" :
9: *d* ← "*izquierda*"

Complejidad: Θ(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 $\equiv$ $\{res =_{\text{obs}} mover(d)\}$
Complejidad: $\Theta(1)$
Descripción: genera una acción de mover en la dirección especificada.

PASAR() $\rightarrow$ res : `accion`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} pasar\}$
Complejidad: $\Theta(1)$
Descripción: genera la acción de pasar.

DISPARAR() $\rightarrow$ res : `accion`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} disparar\}$
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 $\equiv$ $\{res =_{\text{obs}} aplicar(a,\ j,\ e)\}$
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 $\equiv$ $\{res =_{\text{obs}} invertir(e)\}$
Complejidad: $\Theta(1)$
Descripción: invierte un evento.

INVERSA(**in/out** es: `vector(evento)`)
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} inversa(es)\}$
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 $\longrightarrow$ bool

$\text{Rep}(a) \equiv \text{true} \iff$
a.acción $=_{\text{obs}}$ "*disparar*" $\vee$
a.acción $=_{\text{obs}}$ "*pasar*" $\vee$
a.acción $=_{\text{obs}}$ "*mover*"

Abs : acción a $\longrightarrow$ acción $\{\text{Rep}(a)\}$

$\text{Abs}(a) =_{\text{obs}}$ a_{tad}: acción | (a.acción $=_{\text{obs}}$ "*disparar*" $\wedge$ $esDisparar(a_{tad})$) $\vee$
(a.acción $=_{\text{obs}}$ "*pasar*" $\wedge$ $esPasar(a_{tad})$) $\vee$
((a.acción $=_{\text{obs}}$ "*mover*" $\wedge$ $esMover(a_{tad})$) $\wedge_{\text{L}}$ a.dir $=_{\text{obs}}$ $direccion(a_{tad})$)

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() $\to$ *res* : acción

1: $res \leftarrow \langle accion : \text{"}pasar\text{"},\ dir : Arriba() \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iDisparar() $\to$ *res* : acción

1: $res \leftarrow \langle accion : \text{"}disparar\text{"},\ dir : Arriba() \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iMover(**in** d : `dir`) $\to$ *res* : acción

1: $res \leftarrow \langle accion : \text{"}mover\text{"},\ dir : d \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iInvertir(**in** e : `evento`) $\to$ *res* : evento

1: $res \leftarrow \langle pos : e.pos,\ dir : Invertir(e.dir),\ disparo? : e.disparo? \rangle$ ▷ $\Theta(1)$

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

---

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 $\Theta(1)$.