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) $\to$ 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() $\to$ 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)
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 $k \notin$ claves(d) con el significado s en el diccionario.
Aliasing: los elementos k y s se definen por copia.

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

SIGNIFICADO(**in** d: `diccTrie(`$string$`, `α`)`, **in** k: $string$) $\to$ res : σ
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.

Representación

Representación del diccionario

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

donde `estr` es `tupla`(*raiz*: `puntero(nodo)`)

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

Rep : diccTrie $\longrightarrow$ bool
Rep(d) $\equiv$ true $\iff$ raiz no está contenido en ninguno de sus siguientes ni sus siguientes

Abs : diccTrie d $\longrightarrow$ diccTrie($string$, α) {Rep(d)}

Abs(d) $\equiv$ **if** vacía?(d.claves) **then** vacío **else** definir(prim(d).claves, prim(d).significado, Abs(fin(d))) **fi**

Algoritmos

---

iVacía() $\rightarrow$ res : estr

```
1: // Le asigna un nuevo nodo a la raiz
2: res ← ⟨raiz : nuevoNodo()⟩                                   ▷ Θ(1)
```

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

---

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

```
1: Nodo actual ← d.raiz                                         ▷ Θ(1)
2: for (char c : k) do                                          ▷ O(|k|)
3:     actual ← (actual → siguientes[toInt(c)])                 ▷ Θ(1)
4: end for
5: res ← (actual → significado)                                 ▷ Θ(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 ← 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 ← true                                                   ▷ 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: `estr`, **in** k: $string$, **in** s: α)

```
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 significado.
11: // Le asigno una copia del provisto.
12: actual → significado ← copy(s)                              ▷ Θ(copy(s))
```

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() → *res* : puntero(nodo) ▷ Función privada que crea un nuevo nodo

1: // Reserva la memoria para un nuevo nodo con significado null y siguientes vacíos
2: $res \leftarrow \&\langle significado : NULL,\ siguientes : arreglo\_estatico[256]\ de\ \alpha \rangle$ ▷ $\Theta(1)$

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

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)))`,
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 $\longrightarrow$ 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,                                                              ▷ Θ(1)
5:      ronda : 0,                                                             ▷ Θ(1)
6:
7:      // Seteo el mapa
8:      mapa : m,                                                              ▷ Θ(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)],      ▷ Θ(Tam(m)^2)
12:     disparosUltimoPaso : Vacio(),                                          ▷ Θ(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: Θ(?)

---

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 ← 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 ← ⟨identidad : j, posicion : localizacion.pos, direccion : localizacion.dir⟩
11:     itInfoActual ← AgregarRapido(j.infoActualJugadoresVivos, infoActual)
12:
13:     // Creo la infoPJ con la actual
14:     info ← iNuevaInfoPJ(localizacion, itInfoActual)
15:     // La agrego al trie y me guardo el puntero a la info guardada
16:     infoPtr ← &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)`) → *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(1)
11:
12: // Armo la infoPJ
13: res ← ⟨                                                       ▷ Θ(1)
14:     eventos : evts
15:     vivo? : true
16:     infoActual : itInfoActual
17: ⟩
```

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 ← ⟨posicion : eventosFan[0].pos, direccion : eventosFan[0].dir⟩
4: itInfoActualFan ← AgregarRapido(infoActualFan, j.infoActualFantasmasVivos)
5:
6: // Hago que el fantasma especial sea este
7: j.infoFantasmaEspecial ← itInfoActualFan
8:
9: // Creo la infoFan con la actual
10: infoFan ← ⟨infoActual : itInfoActualFan, eventos : eventosFan⟩
11: // La agrego al conjunto de información de fantasmas y me guardo su iterador
12: itInfoFan ← 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** j : `estr`)

```
1:
2: // Incremento el paso
3: j.paso ← j.paso + 1                                              ▷ Θ(1)
4:
5: // Actualizo el mapa de disparos con los fantasmas vivos
6: iActualizarMapaDisparos(j)
7:
8: // Veo que jugadores mueren
9: iChequearMuerteJugadores(j) // Por cada fantasma // Si dispara // Agregar disparo // Agregas el disparo al
   conjunto (inteligentemente) // Agregas las pos afectadas al mapa de disparos // // Actualizo la info actual //
   Por cada jugador // Te fijas si muere // Actualizas la info actual
```

iActualizarMapaDisparos(**in** j : `estr`)

```
1: // Recorro los fantasmas vivos
2: for (itInfoFan : j.infoFantasmasVivos) do
3:     // Obtengo la información del fantasma
4:     infoFan ← Siguiente(itInfoFan)
5:     // Si dispara, agrego el disparo
6:     eventoActual ← iEventoActualFan(infoFan, j.paso)
7:     if (eventoActual.dispara?)
8:         then iAgregarDisparo(j, eventoActual)
9: end for
```

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

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

iEjecutarAccion(**in** j : `estr`, **in** a : `accion`, **in** pj : `jugador`) → res :estr

```
1:
```

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`

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

Abs : dir *d* $\longrightarrow$ dir {Rep(*d*)}
Abs(*d*) $=_{\text{obs}}$ d_{tad}: dir | (d $=_{\text{obs}}$ "*arriba*" $\wedge$ d_{tad} $=_{\text{obs}}\uparrow$) $\vee$
(d $=_{\text{obs}}$ "*abajo*" $\wedge$ d_{tad} $=_{\text{obs}}\uparrow$) $\vee$
(d $=_{\text{obs}}$ "*izquierda*" $\wedge$ d_{tad} $=_{\text{obs}}\leftarrow$) $\vee$
(d $=_{\text{obs}}$ "*derecha*" $\wedge$ d_{tad} $=_{\text{obs}}\rightarrow$)

Algoritmos

Algoritmos del módulo

---

iArriba() → *res* : dir

1: $res \leftarrow$ "*arriba*" ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iAbajo() → *res* : dir

1: $res \leftarrow$ "*abajo*" ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iIzquierda() → *res* : dir

1: $res \leftarrow$ "*izquierda*" ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iDerecha() → *res* : dir

1: $res \leftarrow$ "*derecha*" ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iInvertir(**in**/**out** *d* : `dir`)

1: *switch*(*d*) ▷ $\Theta(1)$
2: *case* "*arriba*" :
3: $d \leftarrow$ "*abajo*"
4: *case* "*abajo*" :
5: $d \leftarrow$ "*arriba*"
6: *case* "*izquierda*" :
7: $d \leftarrow$ "*derecha*"
8: *case* "*derecha*" :
9: $d \leftarrow$ "*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 $\equiv$ $\{res =_{\text{obs}} mover(d)\}$
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*"

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

$\text{Abs}(a) =_{\text{obs}} a_{tad}$: 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 $\Theta(1)$.