Apunte de Mï¿½dulos Bï¿½sicos (v. 0.3α)

Algoritmos y Estructuras de Datos II, DC, UBA.

1er cuatrimestre de 2019

Índice

1. Diccionario Trie (α)

El mï¿½dulo Diccionario Trie provee un diccionario bï¿½sico montado sobre un trie.
Solo se definen e implementan las operaciones que serï¿½n utilizadas.

Interfaz

parï¿½metros formales
  gï¿½neros α
  funciï¿½n COPIAR(**in** s: α) $\rightarrow$ res : α
    Pre $\equiv$ {true}
    Post $\equiv$ {$res =_{\text{obs}} s$}
    Complejidad: $\Theta(copy(s))$
    Descripciï¿½n: funciï¿½n de copia de α

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

gï¿½neros: `diccTrie`($string, \alpha$).

Operaciones bï¿½sicas de diccionario

VACIï¿½O() $\rightarrow$ res : `diccTrie`($string, \alpha$)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacio}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera un diccionario vacï¿½o.

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: $\Theta(|k| + copy(s))$
Descripciï¿½n: 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: $string$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(d, k)}
Complejidad: $\mathcal{O}(|k|)$
Descripciï¿½n: devuelve `true` si y sï¿½lo k estï¿½ definido en el diccionario.

SIGNIFICADO(**in** d: `diccTrie(`$string$`, `α`)`, **in** k: $string$) $\rightarrow$ res : σ
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ obtener(d, k))}
Complejidad: $\Theta(|k|)$
Descripciï¿½n: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y gusï¿½lo si d es modificable.

Representaciï¿½n

Representaciï¿½n del diccionario

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

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

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

Rep : diccTrie $\longrightarrow$ 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.
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 finalemente 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

Abs : estr e $\longrightarrow$ diccTrie($string$, α) {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: diccTrie($string$, α) |
$(\forall$ :: s string)(def?(s,d) $=_{\text{obs}}$ estaDefinido(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

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

---

iVacï¿½a() $\rightarrow$ res : estr
1: // Le asigna un nuevo nodo a la raiz
2: $res \leftarrow \langle raiz : nuevoNodo() \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificacië¿½n: La complejidad de crear un nuevo nodo es $\Theta(1)$

---

iSignificado(**in**/**out** d: `estr`, **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: $\Theta(|k|)$
Justificacië¿½n: 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 serï¿½ $\Theta(|k|)$.

---

iDefinido?(**in/out** d: `estr`, **in** k: *string*) → 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: $\mathcal{O}(|k|)$

Justificaciï¿½n: 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 mï¿½ximo. Luego la complejidad serï¿½ $\mathcal{O}(|k|)$.

---

iDefinir(**in/out** d: `estr`, **in** k: *string*, **in** s: α) → res : 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 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: // Asigno al significado del nodo el puntero creado con s.
14: (actual → significado) ← &sig                                          ▷ Θ(1)
15: // Agrego la nueva clave al conjunto de claves
16: // Como precondicion, se que no existe asi que la agrego rapido
17: AgregarRapido(e.claves, k)                                             ▷ Θ(copy(k))
18: // Devuelvo por referencia el significado.
19: res ← sig                                                              ▷ Θ(1)
```

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

Justificaciï¿½n: Siempre se recorre toda la palabra para definirla, entonces el *for* siempre tiene $|k|$ ciclos. La dereferenciaciï¿½n y comparaciï¿½n de punteros, e indexaciï¿½n en arreglos estï¿½ticos son $\Theta(1)$.

---

inuevoNodo() → res : puntero(nodo) ▷ Funciï¿½n privada que crea un nuevo nodo

```
1: // Reserva la memoria para un nuevo nodo con significado null y siguientes vacï¿½os
2: res ← &⟨significado : NULL, siguientes : arreglo_estatico[256] de puntero(Nodo)⟩   ▷ Θ(1)
```

Complejidad: $\Theta(1)$

Justificaciï¿½n: El tiempo de creaciï¿½n de un array de 255 posiciones es $\mathcal{O}(255) \in \mathcal{O}(1)$

---

iClaves(**in** d: `estr`)→ res : conj(string)

```
1: res ← e.claves                                                          ▷ Θ(1)
```

Complejidad: $\Theta(1)$

2. Mï¿½dulo Juego

Aqui va la descripciï¿½n

Interfaz

generos: `juego`.

se explica con: JUEGO.

Operaciones bï¿½sicas de Juego

INICIAR(**in** m : `mapa`,ï¿½**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
Descripciï¿½n: 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: $\Theta(?)$
Descripciï¿½n: ejecuta un paso de tiempo cuando ningï¿½n jugador realiza una acciï¿½n.

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: $\Theta(?)$
Descripciï¿½n: actualiza con la acciï¿½n 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: $\Theta(1)$
Descripciï¿½n: devuelve un conjunto con punteros a la informaciï¿½n de los personajes que estï¿½n vivos.
Aliasing: res es no modificable.

FANTASMASVIVOS(**in** j : `juego`) $\rightarrow$ res : `conj(infoFan)`
Pre $\equiv$ {true}
Post $\equiv$ $\{fantasmaValido(j, res)\}$
Complejidad: $\Theta(1)$
Descripciï¿½n: devuelve un conjunto referencias a la informaciï¿½n de los fantasmas que estï¿½n 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: $\Theta(1)$
Descripciï¿½n: 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: $O(\#fv)$
Descripciï¿½n: devuelve un conjunto con punteros a la informaciï¿½n de los fantasmas que estï¿½n 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: $O(|j|)$
Descripciï¿½n: devuelve si un jugador estï¿½ vivo

POSOCUPADASPORDISPAROS(**in** j : `juego`) $\rightarrow$ res : `conj(posicion)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} alcanceDisparosFantasmas(fantasmas(j), j)\}$
Complejidad: $O(\#fv * m)$
Descripciï¿½n: devuelve un conjunto de las posiciones afectadas por disparos de fantasmas en la ï¿½ltima *ronda* (TODO: ronda o paso?).

Predicados auxiliares:

fantasmaValido(j, fs):
$(\forall f : infoFan)(f \in res \Rightarrow_{\text{L}}$
$(f.eventos \in fantasmas(j)) \wedge_{\text{L}}$
$(fantasmaVivo(f.eventos, j)) \wedge$
$((\forall e : evento)(e \in f.eventos \Rightarrow_{\text{L}}$
$(e.pos =_{\text{obs}} posFantasma(f.eventos, j)) \wedge$
$(e.dir =_{\text{obs}} dirFantasma(f.eventos, j))))$

Representaciï¿½n

Representaciï¿½n 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 secciï¿½n se hace abuso de notaciï¿½n en los cï¿½lculos de ï¿½lgebra de ï¿½rdenes presentes en la justificaciones de los algoritmos. La operaciï¿½n de suma "+" denota secuencializaciï¿½n de operaciones con determinado orden de complejidad, y el sï¿½mbolo de igualdad "=" denota la pertenencia al orden de complejidad resultante.

Algoritmos del mï¿½dulo

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 tamaï¿½o 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 vacï¿½as
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: // Suponemos la existencia de la funciï¿½n
25: // dict(jugador, tupla(pos, dir)) localizarJugadores(m, conj(jugador) pjs)
26:
27: // Obtengo las posiciones y direcciones de jugadores
28: localPJs ← localizarJugadores(m, pjs)
29:
30: // Lleno las estructuras de jugadores
31: for (j, localizacion : localPJs) do
32:     // Creo la infoActual y la agrego a su conjunto
33:     infoActual ← ⟨identidad : j, posicion : localizacion.pos, direccion : localizacion.dir⟩
34:     itInfoActual ← AgregarRapido(res.infoActualJugadoresVivos, infoActual)
35:
36:     // Creo la infoPJ con la actual
37:     info ← iNuevaInfoPJ(j, localizacion, itInfoActual)
38:     // La agrego al trie y me guardo el puntero a la info guardada
39:     infoPtr ← &Definir(res.infoJugadores, j, info)
40:
41:     // Agrego al conjunto de jugadores vivos el puntero a la info del PJ
42:     AgregarRapido(res.infoJugadoresVivos, infoPtr)
43: end for
44:
45: // Lleno las estructuras de fantasmas
46: // Creo la infoActual y la agrego a su conjunto
47: infoActualFan ← ⟨posicion : eventosFan[0].pos, direccion : eventosFan[0].dir⟩
48: itInfoActualFan ← AgregarRapido(infoActualFan, res.infoActualFantasmasVivos)
49:
50: // Hago que el fantasma especial sea este
51: res.infoFantasmaEspecial ← itInfoActualFan
52:
53: // Creo la infoFan con la actual
54: infoFan ← ⟨infoActual : itInfoActualFan, eventos : eventosFan⟩
55: // La agrego al conjunto de informaciï¿½n de fantasmas y me guardo su iterador
56: itInfoFan ← AgregarRapido(infoFan, res.infoFantasmas)
57:
58: // Agrego al conjunto de fantasmas vivos el interador a la info del Fan
59: AgregarRapido(itInfoFan, res.infoFantasmasVivos)
```

Complejidad: Θ(?)

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

1: // Aumentas paso // 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

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

1:

3. Mï¿½dulo Mapa

Aqui va la descripciï¿½n

Interfaz

generos: `mapa`.

se explica con: HABITACIÏ¿½N.

Operaciones bï¿½sicas del mapa

NUEVOMAPA(**in** n : `nat`) $\rightarrow$ res : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} nuevaHab(n)\}$
Complejidad: $\Theta(n^2)$
Descripciï¿½n: genera un mapa de tamano 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: $\Theta(1)$
Descripciï¿½n: ocupa una posicion del mapa siempre y cuando este no deje de ser conexo.

TAM(**in** m : `mapa`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} tam(m)\}$
Complejidad: $\Theta(1)$
Descripciï¿½n: devuelve el tamano 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: $\Theta(1)$
Descripciï¿½n: devuelve si un elemento esta ocupado.

Representaciï¿½n

Representaciï¿½n del mapa

El objetivo de este mï¿½dulo es implementar una lista doblemente enlazada con punteros al principio y al fin. Para simplificar un poco el manejo de la estructura, vamos a reemplazarla por una lista circular, donde el siguiente del ï¿½ltimo apunta al primero y el anterior del primero apunta al ï¿½ltimo. La estructura de representaciï¿½n, su invariante de representaciï¿½n y su funciï¿½n de abstraccii¿½n son las siguientes.

`mapa` se representa con `m`

donde `m` es `tupla`(*tamano*: `nat`,*casilleros*: `vec(vec(bool))`,)

Rep : mapa $\longrightarrow$ bool
Rep(m) $\equiv$ true $\iff$ La longitud de m.casilleros es igual a tamano $\wedge$
La longitud del vector m.casilleros es igual a la de todo otro vector dentro de el) $\wedge$
Es conexa
Abs : mapa m $\longrightarrow$ hab $\{\text{Rep}(m)\}$
Abs(m) $=_{\text{obs}}$ h: hab | $m.tamano =_{\text{obs}} tam(h) \wedge_{\text{L}}$
($\forall$t: tuple(nat,nat))($0 \leq \Pi_1(t), \Pi_2(t) < m.tamano - 1 \Rightarrow_{\text{L}}$
libre(m, t) $=_{\text{obs}}$ $m.casilleros[\Pi_1(t)][\Pi_2(t)]$)

Algoritmos

En esta secciï¿½n se hace abuso de notaciï¿½n en los cï¿½lculos de ï¿½lgebra de ï¿½rdenes presentes en la justifi-

caciones de los algoritmos. La operaciï¿½n de suma "+" denota secuencializaciï¿½n de operaciones con determinado orden de complejidad, y el sï¿½mbolo de igualdad "=" denota la pertenencia al orden de complejidad resultante.

Algoritmos del mï¿½dulo

---

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

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

Complejidad: $\Theta(1)$

---

iOcupar(**in**/**out** m : `mapa`, **in** c : `tupla(int, int)`)

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

Complejidad: $\Theta(1)$

Justificaciï¿½n: El acceso a una posiciï¿½n de un vector y su modificaciï¿½n es $\Theta(1)$

---

iLibre(**in** m : `mapa`, **in** c : `tupla(int, int)`) $\rightarrow$ res : bool

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

Complejidad: $\Theta(1)$

Justificaciï¿½n: El acceso a una posiciï¿½n de un vector es $\Theta(1)$

---

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

1: $m.tamano \leftarrow n$ ▷ $\Theta(1)$
2: $v \leftarrow Vacia()$ ▷ $\Theta(1)$
3: $i \leftarrow 0$ ▷ $\Theta(1)$
4: **while** $i < n$ **do** ▷ $O(n)$
5: $v.AgregarAtras(false)$
6: $i \leftarrow i + 1$
7: **end while**
8: $i \leftarrow 0$
9: **while** $i < n$ **do** ▷ $O(n^2)$
10: $res.AgregarAtras(v.Copiar())$ ▷ $O(n)$
11: $i \leftarrow i + 1$ ▷ $O(1)$
12: **end while**

Complejidad: $\Theta(n^2)$

Justificaciï¿½n: Copiar un vector de n booleanos es $O(n * copy(bool))$ y copiar un bool es $\Theta(1)$. Luego, agregar n veces la copia del vector es $O(n^2)$

---

4. Mï¿½dulo Direcciï¿½n

Aqui va la descripciï¿½n

Interfaz

generos: `dir`.

se explica con: DIRECCIÏ¿½N.

Operaciones bï¿½sicas de Direcciï¿½n

ARRIBA() $\rightarrow$ *res* : `dir`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}\uparrow$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera la direcciï¿½n arriba.

ABAJO() $\rightarrow$ *res* : `dir`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}\downarrow$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera la direcciï¿½n abajo.

IZQUIERDA() $\rightarrow$ *res* : `dir`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}\leftarrow$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera la direcciï¿½n izquierda.

DERECHA() $\rightarrow$ *res* : `dir`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}\rightarrow$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera la direcciï¿½n derecha.

INVERTIR(**in**/**out** d: `dir`)
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ *invertir*(d)}
Complejidad: $\Theta(1)$
Descripciï¿½n: invierte la direcciï¿½n.

Representaciï¿½n

Representaciï¿½n de Direcciï¿½n

`dir` se representa con `string`

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

Algoritmos

Algoritmos del mï¿½dulo

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. Mï¿½dulo Acciï¿½n

Aqui va la descripciï¿½n

Interfaz

generos: `accion`.

se explica con: Acciï¿½n.

Operaciones bï¿½sicas de Acciï¿½n

MOVER(**in** d: `dir`) $\rightarrow$ res : `accion`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} mover(d)$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera una acciï¿½n de mover en la direcciï¿½n especificada.

PASAR() $\rightarrow$ res : `accion`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} pasar$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera la acciï¿½n de pasar.

DISPARAR() $\rightarrow$ res : `accion`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} disparar$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera la acciï¿½n de disparar.

APLICAR() $\rightarrow$ res : `tupla()`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} disparar$}
Complejidad: $\Theta(1)$
Descripciï¿½n: genera la acciï¿½n de disparar.

Representaciï¿½n

Representaciï¿½n de Acciï¿½n

El objetivo de este mï¿½dulo es implementar una lista doblemente enlazada con punteros al principio y al fin. Para simplificar un poco el manejo de la estructura, vamos a reemplazarla por una lista circular, donde el siguiente del ï¿½ltimo apunta al primero y el anterior del primero apunta al ï¿½ltimo. La estructura de representaciï¿½n, su invariante de representaciï¿½n y su funciï¿½n de abstraccii¿½n son las siguientes.

`mapa` **se representa con** `m`

donde `m` es `tupla`($tamano$: `nat`,$casilleros$: `vec(vec(bool))`,)

Rep : mapa $\longrightarrow$ bool
Rep(m) $\equiv$ true $\iff$ La longitud de m.casilleros es igual a tamano $\wedge$
La longitud del vector m.casilleros es igual a la de todo otro vector dentro de el) $\wedge$
Es conexa
Abs : mapa m $\longrightarrow$ hab {Rep(m)}
Abs(m) $=_{\text{obs}}$ h: hab | $m.tamano =_{\text{obs}} tam(h)$ $\wedge_{\text{L}}$
($\forall t$: tuple(nat,nat))($0 \leq \Pi_1(t), \Pi_2(t) < m.tamano - 1 \Rightarrow_{\text{L}}$
libre(m, t) $=_{\text{obs}}$ $m.casilleros[\Pi_1(t)][\Pi_2(t)]$)

Algoritmos

En esta seccii¿½n se hace abuso de notacii¿½n en los cï¿½lculos de ï¿½lgebra de ï¿½rdenes presentes en la justifi-

caciones de los algoritmos. La operaciï¿½n de suma "+" denota secuencializaciï¿½n de operaciones con determinado orden de complejidad, y el sï¿½mbolo de igualdad "=" denota la pertenencia al orden de complejidad resultante.

Algoritmos del mï¿½dulo

---

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

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

Complejidad: $\Theta(1)$
