Algoritmos y Estructura de Datos 2

Trabajo Práctico 3

Diseño de *Sokoban Extendido*

Alumno: Leandro Carreira
LU: 669/18
Grupo: 15

Documento online con tabla lateral de navegación:
https://docs.google.com/document/d/1is1tpikY0xtDMoD8iQjyyUsS28ioniRCDL9DRQMz0lo/edit?usp=sharing

Coordenada

Coordenada es solo un renombre de una tupla de números naturales.

`Coord` **se representa con** `tupla < x: nat, y: nat >`

Módulo Dirección

Interfaz

se explica con: Dirección
géneros: dirección

Operaciones básicas de dirección

`Norte() → res: dir`
Pre ≡ {true}
Post ≡ {ord(res) = 0}
Complejidad: Θ(1)
Descripción: genera un objeto del tipo dirección hacia el norte

`Este() → res: dir`
Pre ≡ {true}
Post ≡ {ord(res) = 1}
Complejidad: Θ(1)
Descripción: genera un objeto del tipo dirección hacia el este

`Sur() → res: dir`
Pre ≡ {true}
Post ≡ {ord(res) = 2}
Complejidad: Θ(1)
Descripción: genera un objeto del tipo dirección hacia el sur

`Oeste() → res: dir`
Pre ≡ {true}
Post ≡ {ord(res) = 3}
Complejidad: Θ(1)
Descripción: genera un objeto del tipo dirección hacia el oeste

`Ord(`**in**`: dir d) → res: nat`
Pre ≡ {true}

Post ≡ {res = ord(d)}
Complejidad: θ(1)
Descripción: devuelve la representación de la dirección como un número entre 0 y 3 inclusives.

⊕(**in**: coord c, **in**: dir d) → res: coord
Pre ≡ {true}
Post ≡ {res = coord ⊕ d}
Complejidad: θ(1)
Descripción: modifica el valor de la coordenada del personaje con un paso en la dirección de movimiento.

Representación

Representación de Dirección
Se utiliza una tupla de enteros para representar este módulo, donde:
Norte ≡ <0, 1>
Este ≡ <1, 0>
Sur ≡ <0, -1>
Oeste ≡ <-1, 0>

Dirección **se representa con** tupla < x: nat, y: nat >

Algoritmos

---

iOrd(**in** d: dir) → res : nat
if d.x = 0 ∧ d.y = 1 **then**
res ← 0
else if d.x = 1 ∧ d.y = 0 **then**
res ← 1
else if d.x = 0 ∧ d.y = -1 **then**
res ← 2
else if d.x = -1 ∧ d.y = 0 **then**
res ← 3
end if

Complejidad: O(1)

---

iSuma(**in** c: coord, **in** d: dirección) → res : coord
    res ← < c.x + d.x, c.y + d.y >

Complejidad: O(1)

Módulo Mapa

se explica con: Mapa
géneros: mapa

Operaciones básicas de mapa

hayPared?(**in**: mapa m, **in**: coord c) → res: bool
Pre ≡ {true}
Post ≡ {res = hayPared?(m, c)}
Complejidad: O(B + log P)
Descripción: devuelve true si hay una pared en la coordenada de entrada.

hayDepósito?(**in**: mapa m, **in**: coord c) → res: bool
Pre ≡ {true}
Post ≡ {res = hayDepósito?(m, c)}
Complejidad: O(log D)
Descripción: devuelve true si hay un depósito en la coordenada de entrada.

agPared(**in/out**: mapa m, **in**: coord c)
Pre ≡ { ¬hayPared?(m, c) ∧ ¬hayDepósito?(m, c) }
Post ≡ { hayPared?(res, c) ∧ ¬hayDepósito?(res, c) }
Complejidad: O(log P)
Descripción: agrega una pared al mapa en la coordenada de entrada
Aliasing: modifica el mapa de entrada

agDepósito(in/out: mapa m, in: coord c)
Pre ≡ { ¬hayDepósito?(m, c) ∧ ¬hayPared?(m, c) }
Post ≡ { hayDepósito?(res, c) ∧ ¬hayPared?(res, c) }

Complejidad: O(log D)
Descripción: agrega un depósito al mapa en la coordenada de entrada
Aliasing: modifica el mapa de entrada

tirarBomba(**in/out**: mapa m, **in**: coord c)
Pre ≡ {true}
Post ≡ La coordenada de entrada es guardada en la lista de explosiones. Los diccionarios de la estructura no se modifican.
Complejidad: O(1)
Descripción: Las paredes con alguna de sus coordenadas coincidentes con la posición en que se tira la bomba, se considerarán destruídas. Concretamente, *hayPared?* devolverá *false* para todas las posiciones con coordenada x ó y igual a c.x ó x.y.

depósitos(**in**: mapa m) → res: conj(coord)
Pre ≡ {true}
Post ≡ Todos los elementos c del conjunto res devuelven true al usar hayDepósito?(m, c).
Complejidad: O(D)
Descripción: Devuelve un conjunto con las coordenadas correspondientes a los depósitos en el mapa, implementado sobre un Conjunto Lineal.

Representación

Representación de Mapa

En el mapa se encuentran las posiciones de las paredes y depósitos.
Las paredes pueden ser destruídas al tirar una bomba.
Los depósitos son inmutables.
Ambos se representan con un diccionario sobre AVL para cumplir con la complejidad requerida.
Las coordenadas de las explosiones ocurridas se representan con una lista enlazada.

Mapa **se representa con** estr

```
donde estr es tupla < paredes:     diccAVL(nat, coord),
                      depósitos:   diccAVL(nat, coord),
                      explosiones: lista(coord) >
```

Algoritmos

```
iHayPared?(in e: estr, in c: coord) → res : bool
      res ← false
      it  ← CrearIt(e.paredes)
      explotó ← false // Variable auxiliar
      while HaySiguiente?(it) ∧ res = false ∧ explotó = false do
            pared ← obtener( Siguiente(it), e.paredes )
            if c = pared then
                  res ← true
                  // Verifico que no haya sido explotada previamente
                  itEx ← CrearIt(e.explosiones)
                  while HaySiguiente?(itEx) ∧ res = true do
                        explosión ← Siguiente(itEx)
                        if explosión.x = pared.x or explosión.y = pared.y then
                              res ← false
                              explotó ← true
                        end if
                        Avanzar(itEx)
                  end while
            end if
            Avanzar(it)
      end while
```

Complejidad: O(B + log D) dado por el peor caso de búsqueda en un diccionario sobre AVL más buscar sobre la lista enlazada que contiene las coordenadas de todas las explosiones ocurridas hasta el momento.

```
iHayDepósito?(in e: estr, in c: coord) → res : bool
      res ← false
      it  ← CrearIt(e.depósitos)
      while HaySiguiente?(it) ∧ res = false do
            if c = obtener( Siguiente(it), e.depósitos ) then
                  res ← true
            end if
            Avanzar(it)
      end while
```

Complejidad: O(log D) dado por el peor caso de búsqueda en un diccionario sobre AVL.

```
iAgPared(in/out e: estr, in c: coord)
      it  ← CrearIt(e.paredes)
      // Defino nueva entrada en el diccionario de paredes
      // usó cantidad de elementos como clave (comienza en 0)
      definir(#Claves(e.paredes), c, e.paredes)
```

Complejidad: O(log D) dado por el peor caso de inserción en un diccionario sobre AVL.

```
iAgDepósito(in/out e: estr, in c: coord)
      // Defino nueva entrada en el diccionario de depósitos
      // usó cantidad de elementos como clave (comienza en 0)
      definir(#Claves(e.depósitos), c, e.depósitos)
```

Complejidad: O(log D) dado por el peor caso de inserción en un diccionario sobre AVL.

```
iTirarBomba(in/out e: estr, in c: coord)
      agregarAdelante(c, e.explosiones)
```

Complejidad: O(1) dado la complejidad de insertar al comienzo de una lista enlazada.

```
iDepósitos(in e: estr) → res : conj(coord)
      it  ← CrearIt(e.depósitos)
      // Creo Conjunto Lineal Vacío
      res ← Vacío()
      while HaySiguiente?(it) do
            depósito ← obtener( Siguiente(it), e.depósitos )
            res ← AgregarRápido(depósito, res)
            Avanzar(it)
      end while
```

Complejidad: O(D) dado la complejidad de buscar cada uno de los D depósitos e insertar sus coordenadas en un conjunto lineal, que representa el conjunto de coordenadas de salida. Agregarlas a un conjunto lineal tomará O(1) para cada elemento: O(D) + D * O(1) ≡ O(2*D) ≡ O(D)

Módulo Nivel

Interfaz

se explica con: Nivel
géneros: nivel

Operaciones básicas de nivel

mapaN(**in**: nivel n) → res: mapa
Pre ≡ {true}
Post ≡ {res = mapaN(n)}
Complejidad: O(1)
Descripción: devuelve el mapa de un nivel dado.

personaN(**in**: nivel n) → res: coord
Pre ≡ {true}
Post ≡ {res = personaN(n)}
Complejidad: O(1)
Descripción: devuelve la coordenada de la persona en el mapa de un nivel dado.

cajasN(**in**: nivel n) → res: conj(coord)
Pre ≡ {true}
Post ≡ {res = cajasN(n)}
Complejidad: O(C) dado que asume el peor caso donde el conjunto de entrada de las coordenadas de las cajas al crear el mapa fue de una estructura de lista enlazada, y agregarlas a un conjunto lineal tomará O(1) para cada elemento: O(C) + C * O(1) ≡ O(2*C) ≡ O(C)
Descripción: devuelve conjunto de coordenadas de las cajas de un nivel dado, implementado sobre un Conjunto Lineal.

#bombasN(**in**: nivel n) → res: nat
Pre ≡ {true}
Post ≡ {res = #bombasN(n)}
Complejidad: O(1)
Descripción: devuelve la cantidad de bombas disponibles de un nivel dado.

nuevoN(**in**: mapa m, **in**: coord p, **in**: conj(coord) cs, **in**: nat b) → res: nivel
Pre ≡ { p ∉ cs ∧
¬hayPared?(m, p) ∧

```
            #(depósitos(m)) = #(cs)    ∧
            (∀c: coord)(c ∈ cs ⇒ ¬hayPared?(m, c))
        }
Post ≡ {    mapaN(res)      = m   ∧
            personaN(res)   = p   ∧
            cajasN(res)     = cs  ∧
            #bombasN(res)   = b
        }
```

Complejidad: O(1) dado por cada una de las operaciones O(1) que resultan de apuntar a los módulos de Mapa y Conjunto de cajas, y copiar la coordenada de la persona y la cantidad de bombas.

Aliasing: Produce aliasing sobre el módulo de Mapa y sobre el conjunto de cajas.

Representación

Representación de Nivel

En el nivel se encuentra el mapa, las posiciones de la persona y las cajas, y la cantidad de bombas disponibles.

```
Nivel se representa con estr

donde estr es tupla < mapa:     Mapa,
                      persona:  coord,
                      cajas:    lista(coord),
                      bombas:   nat>
```

Algoritmos

```
iMapaN(in e: estr) → res : mapa
    res ← e.mapa
```

Complejidad: O(1)

```
iPersonaN(in e: estr) → res : coord
     res ← e.persona
```

Complejidad: O(1)

```
iCajasN(in e: estr) → res : conj(coord)
     it  ← CrearIt(e.cajas)
     // Creo un Conjunto Lineal vacío
     res ← Vacío()
     while HaySiguiente?(it) do
          caja ← Siguiente(it)
          res  ← AgregarRápido(caja, res)
          Avanzar(it)
     end while
```

Complejidad: O(C) dado que asume el peor caso donde el conjunto de entrada de las coordenadas de las cajas al crear el mapa fue de una estructura de lista enlazada, y agregarlas a un conjunto lineal tomará O(1) para cada elemento: O(C) + C * O(1) ≡ O(2*C) ≡ O(C)

```
i#Bombas(in e: estr) → res : nat
     res ← e.bombas
```

Complejidad: O(1)

```
iNuevoN(in: mapa m, in: coord p, in: conj(coord) cs, in: nat b) → res: nivel
     res.mapa        ← CrearIt(m)
     res.persona     ← p
     res.cajas       ← CrearIt(cs)
     res.bombas      ← b
```

Complejidad: O(1) dado por cada una de las operaciones O(1) que resultan de apuntar a los módulos de Mapa y Conjunto de cajas, y copiar la coordenada de la persoa y la cantidad de bombas.

Aliasing: Produce aliasing sobre el módulo de Mapa y sobre el conjunto de cajas.

Módulo Sokoban

Interfaz

se explica con: Sokoban
géneros: soko

Operaciones básicas de soko

mapa(**in**: soko s) → res: mapa
Pre ≡ {true}
Post ≡ {res = mapa(s)}
Complejidad: O(1) dada por agregar la referencia al nivel de entrada a una lista enlazada vacía, y crear una lista enlazada vacía para estr.acciónFueTirarBomba.
Descripción: devuelve el mapa de un soko dado.

persona(**in**: soko s) → res: coord
Pre ≡ {true}
Post ≡ {res = persona(s)}
Complejidad: O(1)
Descripción: devuelve la coordenada de la persona en el mapa de un soko dado.

hayCaja?(**in**: soko s, **in**: coord c) → res: bool
Pre ≡ {true}
Post ≡ {res = hayCaja?(s, c)}
Complejidad: O(C)
Descripción: devuelve true si y sólo si existe una caja en el nivel en la coordenada c.

#bombas(**in**: soko s) → res: nat
Pre ≡ {true}
Post ≡ {res = #bombas(s)}
Complejidad: O(1)
Descripción: devuelve la cantidad de bombas disponibles de un soko dado.

deshacer(**in/out**: soko s)
Pre ≡ {true}

Post ≡ Si se han realizado acciones desde el inicio del soko, el estado actual del soko se elimina, y el nuevo estado actual será el último en la secuencia de estados del módulo.

De no haberse realizado acciones, el soko de entrada y salida son los mismos.

Complejidad: O(1)

Descripción: modifica un soko dado deshaciendo la última acción realizada.

nuevoS(**in**: nivel n) → res: soko

Pre ≡ {true}

Post ≡ { mapa(res) = mapaN(n) ∧
persona(res) = personaN(n) ∧
#bombas(res) = #bombasN(n) ∧
(∀c: coord)(c ∈ cajasN(n) ⇒ hayCaja?(res, c))
}

Complejidad: O(1)

Descripción: genera un nuevo soko a partir del nivel dado.

mover(**in/out**: soko s, **in**: dir d)

Pre ≡ {puedeMover?(s,d)}

Post ≡ La posicion actual de la persona se modificará en 1 unidad en alguna de las componentes x ó y. De haber interactuado con una caja, ésta también se modificará en 1 unidad en la misma componente.

Complejidad: O(B + C + log P + log D) dada por crear una copia del *nivel* correspontiente al último estado de la lista de estados, donde:

los dos diccionarios sobre AVL del *mapa* del *nivel* (paredes y depósitos) se copian en O(log P) y O(log D),

la lista enlazada de las explosiones del *mapa* del *nivel* ocurridas se copian en O(B),

el conjunto de cajas del *mapa* representado con una lista enlazada se copia en O(C),

y la coordenada de la persona del *nivel* se copia en O(1), ignorada en la complejidad total ya que 1 es constante y menor o igual a cualquiera de las otras complejidades.

Descripción: mueve el personaje de un soko dado en la dirección ingresada.

tirarBomba(**in/out**: soko s)

Pre ≡ {#bombas(s) > 0}

Post ≡ La cantidad de bombas disponibles se reduce en 1 al valor de antes de tirar la bomba.

La coordenada de la persona al momento de tirar la bomba será agregada a la lista de explosiones del mapa (usada para considerar paredes destruídas).

Complejidad: O(1)

Descripción: destruye todas las paredes en la misma fila o columna que la coordenada de la persona al momento de tirar la bomba.

noHayParedNiCaja?(**in**: soko s, **in**: coord c) → res: bool
Pre ≡ {true}
Post ≡ {res = ¬hayPared?(mapa(s), c) ∧ ¬hayCaja(s, c)}
Complejidad: O(B + log P + C) dado por las dos operaciones que se necesitan para responeder la pregunta.
Descripción: devuelve true si y sólo si en la coordenada c no existen paredes en el mapa del soko, y tampoco cajas en el nivel.

puedeMover?(**in**: soko s, **in**: dir d) → res: bool
Pre ≡ {true}
Post ≡ res será true solo en los casos donde no hay objetos (paredes o cajas) en la coordenada del personaje siguiendo la dirección d, o si hay una caja y delante de ella no hay objetos.
Complejidad: O(B + log P + C) dado por la pregunta noHayParedNiCaja? realizada dos veces: 1 para saber si el personaje tiene el camino libre, y de haber una caja, preguntar si noHayParedNiCaja delante de ésta.
Descripción: devuelve true si y sólo si el personaje se puede mover en la dirección d.

ganó?(**in**: soko s) → res: bool
Pre ≡ {true}
Post ≡ res es true si y sólo si todas las cajas coinciden con las coordenadas de todos los depósitos en el nivel actual del soko.
Complejidad: $O(C^2)$ dado por el peor caso de tener que comparar las coordenadas de todas las cajas con las coordenadas de todos los depósitos O(C*D), y como C=D por la Pre condición al crear el nivel, se puede escribir como $O(C^2)$.
Descripción: devuelve true si ganó el nivel actual del soko.

hayCajas?(**in**: soko s, **in**: conj(coord) cs) → res: bool
Pre ≡ {true}
Post ≡ res es true si y sólo si en cada coordenada del conjunto cs existe una caja en el nivel actual del soko.
Complejidad: $O(C^2)$ dado por el peor caso de tener que comparar las coordenadas de todas las cajas del mapa con las coordenadas de todas las cajas del conjunto de entrada cs.
Descripción: devuelve true si todas las cajas de cs existen en el nivel actual.

Representación

Representación de Sokoban

El módulo Sokoban permiten interactuar con el nivel actual manteniendo un historial de los estados observados luego de cada una de las acciones tomadas Se agrega la opción de deshacer acciones.

```
Sokoban se representa con estr

donde estr es tupla < estados: lista(itNivel),
                      acciónFueTirarBomba: lista(Bool)
                      >
```

Algoritmos

```
iMapa(in e: estr) → res : mapa
     res ← e.nivel.mapa
```

Complejidad: O(1)
Aliasing: Devuelve el mapa del nivel actual por refencia. La lista de niveles a la cual apunta pertenece al módulo de Juego.

```
iPersona(in e: estr) → res : coord
     res ← e.nivel.persona
```

Complejidad: O(1)

```
iHayCaja?(in e: estr, in c: coord) → res : bool
     it  ← CrearIt(e.nivel.cajas)
     res ← false
     while HaySiguiente(it) ∧ res = false do
          if c = Siguiente(it) then
               res ← true
          end if
          Avanzar(it)
     end while
```

Complejidad: O(C) dado por buscar en el conjunto de cajas implementado con una lista enlazada en el módulo Nivel.

```
iBombas(in e: estr) → res : nat
    res ← e.nivel.bombas
```

Complejidad: O(1) dado por devolver el valor del Nat bombas en el módulo Nivel.

```
iDeshacer(in/out e: estr)
    if longitud(e.acciónFueTirarBomba) > 0 then
        // El nivel original fue modificado con alguna acción
        if Último(e.acciónFueTirarBomba) then
            // Solo borro la explosión y recupero la bomba
            it ← CreatItUlt(e.acciónFueTirarBomba)
            EliminarSiguiente(it)
            it ← CreatItUlt(Último(e.estados).mapa.explosiones)
            EliminarSiguiente(it)
            Último(e.estados).#bombasN ← Último(e.estados).#bombasN + 1
        else
            // Borro último estado pues me había movido
            it  ← CreatItUlt(e.estados)
            EliminarSiguiente(it)
        end if
    end if
```

Complejidad: O(1) dado por borrar el último elemento de una lista en el caso de deshacer un movimiento, y el último elemento de 2 listas y sumar y copiar un Nat en el caso de deshacer una explosion.

```
iNuevoS(in n: nivel) → res : soko
    nuevaListaEstados ← Vacía()
    it ← CrearIt(n)
    res.estados ← AgregarAtrás(nuevaListaEstados, it)
    res.acciónFueTirarBomba ← Vacía()
```

Complejidad: O(1) dada por agregar la referencia al nivel de entrada a una lista enlazada vacía, y crear una lista enlazada vacía para estr.acciónFueTirarBomba.

```
iMover(in/out s: soko, in d: dir)
    // Pre: puedoMover?(s, d) es true
    nuevoEstado   ← Último(s.estados).persona ⊕ d
```

```
    itNuevoEstado ← CreatIt(nuevoEstado)
    AgregarAtrás(s.estados, itNuevoEstado)
    // registro las bombas tiradas en cada acción para deshacerlas en O(1)
    AgregarAtrás(acciónFueTirarBomba, false)
```

Complejidad: O(B + C + log P + log D) dada por crear una copia del *nivel* correspontiente al último estado de la lista de estados, donde:

los dos diccionarios sobre AVL del *mapa* del *nivel* (paredes y depósitos) se copian en O(log P) y O(log D),

la lista enlazada de las explosiones del *mapa* del *nivel* ocurridas se copian en O(B),

el conjunto de cajas del *mapa* representado con una lista enlazada se copia en O(C),

y la coordenada de la persona del *nivel* se copia en O(1), ignorada en la complejidad total ya que 1 es constante y menor o igual a cualquiera de las otras complejidades.

```
iTirarBomba(in/out s: soko)
    // Pre: #bombas(s) > 0
    posExplosión  ← Último(s.estados).persona
    Último(s.estados).mapa.tirarBomba(posExplosión)
    // registro las bombas tiradas en cada acción para deshacerlas en O(1)
    AgregarAtrás(s.acciónFueTirarBomba, true)
```

Complejidad: O(1)

```
iNoHayParedNiCaja?(in e: estr, in c: coord) → res : bool
    // Uso operación de Mapa
    noHayPared ← ¬hayPared?(Último(e.estados).mapa, c)
    // Uso operación de Nivel
    noHayCaja  ← ¬hayCaja?(Último(e.estados), c)
    res ← noHayPared ∧ noHayCaja
```

Complejidad: O(log P + C) dado por el diccionario sobre AVL sobre el cual se representan las paredes, y la lista enlazada que representa al conjunto de cajas.

```
iPuedoMover?(in e: estr, in d: dir) → res : bool
    // Dos casos: O está libre, o hay una caja que puedo
    posPersona ← e.Último(e.estados).mapa.persona
```

```
    nuevaPos ← posPersona ⊕ d
    res ← false
    if noHayParedNiCaja?(e, nuevaPos) then
        res ← true
    else
        if hayCaja?(Último(e.estados), nuevaPos) then
            res ← noHayParedNiCaja?(e, nuevaPos ⊕ d)
        end if
    end if
```

Complejidad: O(log P + C) dado por el uso de noHayParedNiCaja dos veces, concretamente: 2 * O(log P + C) ≡ O(2 * (log P + C)) ≡ O(log P + C).

```
iGanó?(in e: estr) → res : bool
    // Uso operación de igualdad de conjuntos lineales
    res ← e.Último(e.estados).cajas = e.Último(e.estados).mapa.depósitos
```

Complejidad: $O(C^2)$ dado por el peor caso de tener que comparar las coordenadas de todas las cajas con las coordenadas de todos los depósitos O(C*D), y como C=D por la Pre condición al crear el nivel, se puede escribir como $O(C^2)$.

```
iHayCajas?(in e: estr, in cs: conj(cajas)) → res : bool
    // Uso operación de igualdad de conjuntos lineales
    res ← cs = e.Último(e.estados).cajas
```

Complejidad: $O(C^2)$ dado por el peor caso de tener que comparar las coordenadas de todas las cajas del mapa con las coordenadas de todas las cajas del conjunto de entrada cs.

Módulo Juego

Interfaz

se explica con: Juego
géneros: juego

Operaciones básicas de juego

nivelActual(**in**: juego j) → res: soko
Pre ≡ {true}
Post ≡ {res = nivelActual(j)}
Complejidad: O(1)
Descripción: devuelve el soko de un juego dado.

nivelesPendientes(**in**: juego j) → res: secu(nivel)
Pre ≡ {true}
Post ≡ {true}
Complejidad: O(1)
Descripción: devuelve una secuencia de los niveles pendientes.

nuevoJ(**in**: secu(nivel) ns) → res: juego
Pre ≡ {¬vacía?(ns)}
Post ≡ nivelActual(res) = prim(ns) ∧
nivelesPendientes(res) coincidirá en cada nivel y el mismo orden que el fin de la secuencia de entrada ns.
Complejidad: O(1)
Descripción: genera un nuevo juego a partir de una secuencia de niveles de entrada.

mover(**in/out**: juego j, **in**: dir d)
Pre ≡ {puedeMover?(nivelActual(j), d)}
Post ≡ La posicion actual de la persona en el mapa actual se modificará en 1 unidad en alguna de las componentes x ó y. De haber interactuado con una caja, ésta también se modificará en 1 unidad en la misma componente.
Complejidad: O(B + C + log P + log D)
Descripción: mueve el personaje de un soko del juego dado en la dirección ingresada.

tirarBomba(**in/out**: juego j)
Pre ≡ {#bombas(s) > 0}
Post ≡ La cantidad de bombas del nivel actual disponibles se reduce en 1 al valor de antes de tirar la bomba.
La coordenada de la persona al momento de tirar la bomba será agregada a la lista de explosiones del mapa (usada para considerar paredes destruídas).
Complejidad: O(1)
Descripción: destruye todas las paredes en la misma fila o columna que la coordenada de la persona al momento de tirar la bomba.

tirarBomba(**in/out**: juego j)
Pre ≡ {#bombas(s) > 0}
Post ≡ La cantidad de bombas del nivel actual disponibles se reduce en 1 al valor de antes de tirar la bomba.

La coordenada de la persona al momento de tirar la bomba será agregada a la lista de explosiones del mapa (usada para considerar paredes destruídas).
Complejidad: O(1)
Descripción: destruye todas las paredes en la misma fila o columna que la coordenada de la persona al momento de tirar la bomba.

deshacer(**in/out**: juego j)
Pre ≡ {true}
Post ≡ Si se han realizado acciones desde el inicio del soko del juego, el estado actual del soko se elimina, y el nuevo estado actual será el último en la secuencia de estados del módulo.
De no haberse realizado acciones, el juego de entrada y salida son los mismos.
Complejidad: O(1)
Descripción: modifica un soko de un juego actual dado, deshaciendo la última acción realizada.

Representación

Representación de Juego
Juego es el módulo de más alto nivel, encargado de llevar un orden en los Niveles que se van jugando y ganando, y gando acceso a las operaciones de módulo de menor nivel como Sokoban, Nivel y Mapa.
Su estructura consiste en una Lista Enlazada de niveles, una referencia al nivel actualmente en curso, y una referencia al Sokoban más reciente, correspondiente al mismo nivel al que apunta itNivelActual.

Juego **se representa con** estr

donde estr es tupla < niveles: listaEnlazada(nivel) ,
itNivelActual: itListaEnlazada(nivel),
itSokoActual : itSokoban >

Algoritmos

```
iNuevoJ(in ns: secu(nivel)) → res : juego
      // Guardo la referencia a la secuencia de niveles en res.niveles
      it ← CrearItUni(ns)
      res.niveles ← it
      // Mantengo una referencia al nivel actual
      res.itNivelActual ← it
      // Creo un nuevo Sokoban con el primer nivel de la secuencia
      // Guardo una referencia al mismo en res.itSokoActual
      res.itSokoActual  ← CrearIt(nuevoS(Siguiente(it)))
```

Complejidad: O(1) dado por guardar referencias a los elementos de la lista de niveles.
Aliasing: Utiliza la secuencia de niveles de entrada por referencia.

```
iNivelActual(in e: estr) → res : soko
      res ← e.itSokoActual
```

Complejidad: O(1)
Aliasing: Devuelve referencia al módulo soko utilizado actualmente por juego.

```
iNivelesPendientes(in e: estr) → res : secu(nivel)
      res  ← e.itNivelActual
```

Complejidad: O(1)
Aliasing: Devuelvo iterador unidireccional a la secuencia de niveles que utiliza el módulo juego, a partir del nivel que se está jugando actualmente.

```
iMover(in/out e: estr, in d: dir)
      // Uso mover y ganó? de módulo Sokoban
      mover(Siguiente(e.itSokoActual), d)
      if ganó?(Siguiente(e.itSokoActual)) ∧ HaySiguiente(e.itNivelActual) then
            Avanzar(e.itNivelActual)
            // Creo un nuevo Sokoban con el siguienet nivel de la secuencia
            // Guardo una referencia al mismo en e.itSokoActual
            e.itSokoActual  ← CrearIt(nuevoS(Siguiente(e.itNivelActual)))
      end if
```

Complejidad: O(B + C + log P + log D) dada por crear una copia del *nivel* correspontiente al último estado de la lista de estados, donde:

los dos diccionarios sobre AVL del *mapa* del *nivel* (paredes y depósitos) se copian en O(log P) y O(log D),

la lista enlazada de las explosiones del *mapa* del *nivel* ocurridas se copian en O(B),

el conjunto de cajas del *mapa* representado con una lista enlazada se copia en O(C),

y la coordenada de la persona del *nivel* se copia en O(1), ignorada en la complejidad total ya que 1 es constante y menor o igual a cualquiera de las otras complejidades.

Aliasing: Devuelvo iterador unidireccional a la secuencia de niveles que utiliza el módulo juego, a partir del nivel que se está jugando actualmente.

---

iTirarBomba(**in/out** e: estr)

```
// Uso operación tirarBomba de Sokoban
tirarBomba(e.itSokoActual)
```

Complejidad: O(1) dada por la operación *tirarBomba* de Sokoban

---

iDeshacer(**in/out** e: estr)

```
// Uso operación deshacer de Sokoban
deshacer(e.itSokoActual)
```

Complejidad: O(1) dada por la operación *deshacer* de Sokoban

---