Modulo Ranking

Representación:

fichin se representa con estr donde

estr es tupla ⟨ hayPersonaJugando: bool,
ranking: diccTrie(nombre, puntaje),
partidaActual: partida,
mapaRanking: mapa,
infoJugadorActual: tupla<nombre, nat>
⟩

Invariante de Representación - Solución Informal:

- mapaRanking es igual a mapa(partidaActual)
- Si el primer elemento de la tupla infoJugadorActual es distinto de “” entonces hayPersonaJugando= TRUE y si es igual a “”, entonces hayPersonaJugando= FALSE
- El segundo elemento de infoJugadorActual debe ser igual a CantMov(partidaActual)

Función de abstracción:

- El observador mapa está representado con la estructura mapaRanking
- El observador alguienJugando? está representado por el bool hayPersonaJugando.
- El observador jugadorActual está representado por pi_1(infoJugadorActual)
- El observador partidaActual está representado por partidaActual de la estructura.
- El observador ranking está representado por ranking de la estructura.

Interfaz:

Parámetros formales:

Géneros: rank

Se explica con: Fichin

Géneros: fichin, partida, mapa

Operaciones básicas:

MAPA(in f: fichin) -> res: mapa

Pre ≡ {true}

Post ≡ {res =obs mapa(f^)}

Descripción: Devuelve el mapa asociado al fichín dado.

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

ALGUIENJUGANDO?(in f: fichin) -> res: bool

Pre ≡ {true}

Post ≡ {res^ =obs alguienJugando?(f^)}

Descripción: Devuelve true si hay alguien jugando.

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

JUGADORACTUAL(in f: fichin) -> res: jugador

Pre ≡ {alguienJugando?(f)}

Post ≡ {res^ =obs jugadorActual(f^)}

Descripción: Retorna el jugador actual si hay una partida en curso.

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

PARTIDAACTUAL(in f: fichin) -> res: partida
Pre ≡ {alguienJugando?(f)}
Post ≡ {res^ =obs partidaActual(f^)}
Descripción: Si hay alguien jugando, retorna la partida actual.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

RANKING(in f: fichin) -> res: ranking
Pre ≡ {true}
Post ≡ {res^ =obs ranking(f^)}
Descripción: Dado un fichin devuelve su ranking asociado.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

NUEVOFICHÍN(in m: mapa) -> res: fichin
Pre ≡ {true}
Post ≡ {res^ =obs nuevoFichin(f^)}
Descripción: Genera un nuevo fichín a partir de un mapa.
Complejidad: Θ(1)
Aliasing: Devuelve una referencia modificable.

NUEVAPARTIDA(in/out f:fichin, in j: jugador) -> res: partida
Pre ≡ {f = f0 ∧ ¬alguienJugando?(f) }
Post ≡ {f =obs nuevaPartida(f0^, j) ∧L res = partidaActual(f^)}
Descripción: Siempre que no haya nadie jugando, a partir de un jugador y un fichín se crea una nueva partida, y la devuelve.
Complejidad: Θ(c)
Aliasing: Devuelve una referencia modificable.

MOVER(in/out f:fichin,in d:dirección) -> res: partida
Pre ≡ {f= f0 ∧ alguienJugando?(f)}
Post ≡ {f =obs mover(f0^, d^) ∧L res = partidaActual(f^) }
Descripción: Si hay alguien jugando, a partir de una dirección y un fichín actualiza la posición del jugador y devuelve la partida actualizada con el movimiento hecho.
Complejidad: Θ(|Jugador|)
Aliasing: No presenta aspectos de aliasing.

OTRAS OPERACIONES:

OBJETIVO(in f:fichin) -> res: tupla<jugador, nat>
Pre ≡ {alguienJugando?(f) ∧ definido?(jugadorActual(f), ranking(f))}
Post ≡ {res =obs objetivo(f)}
Descripción: Devuelve una tupla con el mejor puntaje inmediato al jugador actual y el nombre del jugador con este puntaje.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

Algoritmos del módulo:

iAlguienJugando(in f: fichin): -> res: bool
res ← f.hayPersonaJugando // O(1)

iMapa(in f: fichin) -> res: mapa
res ← f.mapaRanking // O(1)

iJugadorActual(in f: fichin) -> res: jugador
res ← pi_1(f.infoJugadorActual) // O(1)

```
iPartidaActual(in f: fichin)        -> res: partida
        res ← f.partidaActual  // O(1)

iRanking (in f: fichin)    -> res: ranking
        res ← f. ranking  // O(1)


iNuevoFichin(in m: mapa)-> res: fichin
        res ← <false, vacío, <m,conjVacio, 0,m.inicio,0>, m, <"", 0>>  // O(1)


iNuevaPartida(in/out f:fichin,in j:jugador)->res:<partida>
        f.infoJugadorActual←<j,0>  // O(1)
        f.hayPersonaJugando ← true            // O(1)
        res ←<f.mapaRanking, Chocolates(f.mapaRanking), 0, f.mapaRanking.inicio, 0> // O(1)



iMover(in/out f:fichin,in d:dirección)
        f.partida←f.partida.Mover(f.mapaRanking, d)              // O(1)
        pi_2(f.infoJugadorActual) ← pi_2(f.infoJugadorActual) +1 // O(1)
        if(Perdio?(f.partida)){
                f.hayPersonaJugando←false  // O(1)
                f.partidaActual←<f.mapaRanking,conjVacio, 0,f.mapaRanking.inicio,0>  // O(1)
                f.infoJugadorActual←<"",0>  // O(1)
        }

        if(Gano?(f.partida)){
                if( !definido?(f.ranking,pi_1(f.InfoJugadorActual))) ||
                (definido?(f.ranking,pi_1(f.InfoJugadorActual))) &&
```

```
            pi_1(f.InfoJugadorActual)) >
            significado(f.ranking,pi_1(f.InfoJugadorActual))){
                definir←(f.ranking,pi_1(f.InfoJugadorActual),pi_2(f.infoJugadorActual)
            }
        //Siendo f.ranking un dicc trie entonces tanto insertar como buscar si está definido es |N|.
        f.hayPersonaJugando←false
        f.partidaActual←<f.mapaRanking,conjVacio, 0,f.mapaRanking.inicio,0>
        f.infoJugadorActual←<"",0>
        }
```

ALGORITMOS AUXILIARES:

```
iObjetivo(in f:fichin) -> res: tupla<jugador, nat>
    res <- f.infoJugadorActual
    itDic <- crearIt(f.ranking)
    while(HaySiguiente(itDic)){
        posibleSiguiente <- siguiente(it)
        if(pi_2(posibleSiguiente) < (pi_2(f.infoJugadorActual)) ∧ (pi_2(res) < pi_2(posibleSiguiente))) then
            res <- posibleSiguiente
```

Servicios usados: Usa los módulos de **partida**, de **mapa**, y de módulos básicos el **DiccTrie** donde almacenamos el ranking para cumplir la complejidad del segundo punto (al ganar la partida, y tener que hacer una búsqueda en el diccionario donde el peor caso quedará con la longitud de la clave más larga) y un **DiccLineal** para la operación de objetivo.

Módulo Partida

Representación:

partida se representa con estr donde
estr es tupla ⟨ MapaPartida: mapa,
ChocolatesPartida: conjLineal(posición),
CantMovimientos: nat,
PosJugador: coordenada
movimientosInmunes: nat⟩

Invariante de Representación - Solución Informal:

-Para todo elemento de ChocolatesPartida en la coordenada que denota en el tablero de MapaPartida, la tercer componente debe estar en true.
-Por cada elemento del tablero con la tercer componente (de chocolate) en true, contiene un iterador en la cuarta componente que permite acceder siempre a algún elemento en ChocolatesPartida.
-Si CantMovimientos es 0, PosJugador es igual a la pos de inicio que se extrae de MapaPartida.
-En movimientosInmunes el valor que tiene debe estar entre 0 y 10.

Función de abstracción:

-El mapa es el elemento MapaPartida de la estructura.
-La posición del jugador está indicada en PosJugador.
-Los chocolates son el conjunto de posiciones que identifican a ChocolatesPartida
-La cantMov está indicada por cantMovimientos.
-La inmunidad que posee el jugador la indicamos con el elemento de la estructura movimientosInmunes.

Interfaz:

Parámetros formales:
Géneros: partida

Se explica con: Partida
Géneros: partida, mapa

Operaciones básicas:

MAPA(in p:partida) -> res: mapa
Pre ≡ {true}
Post ≡ {res^ =obs mapa(p^)}
Descripción: Devuelve el mapa asociado a la partida.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

JUGADOR(in p: partida) -> res: coordenada
Pre ≡ {true}
Post ≡ {res^ =obs jugador?(p^)}
Descripción: Devuelve la posición del jugador.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

CHOCOLATES(in p: partida) -> res: conj(coordenada)

Pre ≡ {true}

Post ≡ {res^ =obs chocolates(p^)}

Descripción: Retorna el conjunto de las posiciones de los chocolates de la partida.

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

CANTMOV(in p:partida) ->res:nat

Pre ≡ {true}

Post ≡ {res^ =obs cantMov(p^)}

Descripción: Retorna la cantidad de movimientos hasta ahora realizados en la partida actual.

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

INMUNIDAD(in p:partida) -> res: nat

Pre ≡ {true}

Post ≡ {res^ =obs inmunidad(p^)}

Descripción: Retorna cuantos movimientos de inmunidad le quedan al jugador.

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

NUEVAPARTIDA(in m: mapa) -> res: partida

Pre ≡ {true}

Post ≡ {res^ =obs nuevaPartida(m^)}

Descripción: Genera una nueva partida a partir de un mapa dado.

Complejidad: O(|c|) + O(|c|) + O(|c|) = O(|c|)

Aliasing: devuelve una referencia modificable.

MOVER(in/out p:partida, in d: direccion)

Pre ≡ {p = p0 ^ ¬perdio?(p) ^ ¬gano?(p)}

Post ≡ {p^ =obs mover(p0^,d^)}
Descripción: A partir de una partida y una dirección devuelve la partida modificada.
Complejidad: Θ(1)
Aliasing: Devuelve una referencia modificable.

OTRAS OPERACIONES:

GANÓ?(in p:partida) -> res: bool
Pre ≡ {true}
Post ≡ {res^ =obs ganó?(p^)}
Descripción: Retorna true si ganó la partida.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

PERDIÓ?(in p:partida) -> res: bool
Pre ≡ {true}
Post ≡ {res^ =obs perdio?(p^)}
Descripción: Devuelve un bool que es true si perdió la partida.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

IMPLEMENTACIÓN: ALGORITMOS DEL MÓDULO

```
iMAPA(in p:partida)  -> res: mapa
        res ← p.MapaPartida

iJUGADOR(in p: partida) ->  res: coordenada
        res ← p.PosJugador

iCHOCOLATES(in p: partida)  -> res: conj(coordenada)
        res ← p. ChocolatesPartida


iCANTMOV(in p:partida)        ->res:nat
        res ← p.CantMovimientos

iINMUNIDAD (in p:partida)     -> res: nat
        res ← p.movimientosInmunes


iNUEVAPARTIDA(in m: mapa) -> res: partida
   chocolatesP<-mapa.chocolates // O(|c|), copiar coordenada es O(1)
   inmunidad <- 0
  if(mapa.tablero[mapa.inicio.first][mapa.inicio.second].third == true)
        inmunidad <-10
        chocolatesP.Eliminar(mapa.inicio) // O(|c|)
        mapa.tablero[mapa.inicio.first][mapa.inicio.second].third<-false //O(1)
        itElem <- crearIt(chocolatesP)          //O(1)
   while(HaySiguiente?(chocolatesP.itConj(coordenada))) // O(|c|)
        elem <- Siguiente(chocolatesP.itConj(coordenada))  //O(1)
        mapa.tablero[elem.first][elem.second].forth <- itElem   //O(1)
        Avanzar(itElem)  //O(1)
```

```
        Avanzar(chocolatesP.itConj(coordenada)))  //O(1)
    mapa<- &mapa.tablero  //O(1)
    res<- <mapa, chocolatesP, 0, mapa.inicio, inmunidad > //O(1)


iMOVER(in/out p:partida, in d: direccion)
        mapa <- m.MapaPartida
        coordenada <- SIGUIENTEMOVIMIENTO(p,d)   // O(1)
        if(p.posJugador != coordenada)
            p.CantMovimientos <- p.CantMovimientos + 1   // O(1)
            if(mapa.tablero[coordenada.first][coordenada.second].third == true)
                p.movimientosInmunes <- 10   // O(1)
                mapa.tablero[coordenada.first][coordenada.second].forth.EliminarSiguiente()  // O(1)
                mapa.tablero[coordenada.first][coordenada.second].third <-  false   // O(1)
            else
                if(p.movimientosInmunes > 0)
                    p.movimientosInmunes <- p.movimientosInmunes  - 1  // O(1)
        p.posJugador <- coordenada // O(1)


iGanó?(in p:partida) -> res: bool
  res ← p.PosJugador= llegada(p.MapaPartida)

iPerdió??(in p:partida) -> res: bool
   res <- !Ganó?(p) && p.movimientosInmunes == 0 && tieneFantasmaCerca?(p)
```

IMPLEMENTACIÓN: OPERACIONES AUXILIARES DEL MÓDULO

SIGUIENTEMOVIMIENTO(in p:partida, d: direccion) -> res: coordenada
Pre ≡ {true}
Post ≡ {res ^ =obs siguienteMovimiento(p^,d^)}
Descripción: Retorna una coordenada que indica a dónde se moverá el jugador. Siempre y cuando el movimiento sea posible, retorna el movimiento. Caso contrario, retorna la misma coordenada que se le ingresó. (El movimiento no va a una pared, y tiene que estar dentro de los limites del mapa).
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

POSMOVIMIENTO(in c:coordenada, in d: dirección) -> res: coordenada
Pre ≡ {true}
Post ≡ {res^ =obs posMovimiento(p^,d^)}
Descripción: Retorna una coordenada donde caería el jugador al moverse en la dirección d.
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

TIENEFANTASMACERCA?(p:partida) -> res:bool
Pre ≡ {true}
Post ≡ {res =obs tieneFantasmaCerca?(p,d)}
Descripción: Retorna una coordenada donde caería el jugador al moverse en la dirección d
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

IMPLEMENTACIÓN: ALGORITMOS AUXILIARES DEL MÓDULO

```
iSIGUIENTEMOVIMIENTO(in p:partida, d: direccion) -> res: coordenada
      mapa <- p.MapaPartida
      alto<-mapa.tablero.tam()
      largo<-mapa.tablero[0].tam()
      posMovimiento <- POSMOVIMIENTO(p.posJugador,d)
      if(posMovimiento.first < alto && posMovimiento.second < largo&&
         mapa.tablero[posMovimiento.first][posMovimiento.second].second = false)
         res<- posMovimiento
       else
          res<- p.posJugador
```

/*COMO COORDENADA ES DE NATS, POR MÁS QUE SE VAYA MUY A LA IZQUIERDA O MUY ABAJO, LLEGANDO A TENER ALGUNA DE LAS DOS COMPONENTES NEGATIVAS, ASUMIMOS QUE SIEMPRE VA A DEVOLVER 0 EN LA CORRESPONDIENTE COMPONENTE EN ESTE CASO*/

```
iPOSMOVIMIENTO(in c:coordenada, in d: dirección) -> res: coordenada
    nuevaPos <- coordenada
    if (direccion == “IZQUIERDA”)
      nuevaPos <- nuevaPos.first -1
if (direccion == “DERECHA”)
     nuevaPos <- nuevaPos.first +1
if (direccion == “ARRIBA”)
    nuevaPos <- nuevaPos.second +1
if (direccion == “ABAJO”)
   nuevaPos <- nuevaPos.second -1
res<- nuevaPos


TIENEFANTASMACERCA?(p:partida) -> res:bool
    tablero<-p.mapaPartida.tablero
```

```
pos<-p.PosJugador
alto<-tablero.tam()
largo<-tablero[0].tam()
res<-false
for(i=max(0,pos.first-3)....i<min(pos.first+3, largo-1))
    for(j=max(0,pos.second-3)...j<min(pos.second+3,alto-1))
        if(tablero[i][j].first == true)
            res<- res || true
```

Complejidad: Explicación

Queremos saber si hay fantasma a distancia 3 Manhattan o menor, a la redonda del jugador. Entonces, vamos a tener que revisar todas las posiciones del tablero que se encuentren a esta distancia de él. A lo sumo, lo que va a ocurrir, es que vamos a tener que recorrer todas las posibles posiciones a la redonda que cumplen esto. Y como ya conocemos esta magnitud a la redonda, que es a lo sumo 3, esto nos deja con exactamente (2 * 3 + 1) ^ 2 = 49 posiciones que tendremos que revisar, como máximo.

Servicios usados: Usamos el módulo de **mapa** que está un nivel más abajo dentro de los módulos auxiliares para implementar dentro de la estructura de la partida, y el **ConjLineal** de módulos básicos del que podemos almacenar un iterador a una posición en la que haya un chocolate para que eventualmente la eliminación del chocolate de los chocolates de la partida, es decir, comer un chocolate, sea **O(1)**, como indica el punto c) de la consigna del TP.

Módulo Mapa

Representación:

mapa se representa con estr donde

estr es tupla ⟨ inicio: coordenada,
llegada: coordenada,
tablero: AD de AD de infoPosicion,
chocolates: conjLineal<coordenada> ⟩

donde AD es arreglo_dimensionable(infoPosicion)
donde infoPosicion es <fantasma: bool, pared: bool, chocolate:bool, itConj<coordenada>>

Invariante de Representación - Solución Informal:

-Todas las coordenadas de chocolates deben estar bien indexadas entre el alto y el largo del tablero, que es de a x l donde a es el tamaño de cada fila, y l es el tamaño de cada columna.
- Las posiciones de inicio y llegada también deben estar bien indexadas, además ser diferentes entre sí y no pueden ser fantasma ni pared.
-Todos los chocolates del conjunto de chocolates deben tener en true la tercer componente de su tupla en tablero y en false las otras dos.
-Ninguna posición con true en la tercer componente del arreglo bidimensional de tablero puede no estar en chocolates.
-Ninguna posición de infoPosicion en tablero pueden tener más de una componente en true, a lo sumo una.
-Todos los iteradores en la cuarta componente de cada infoPosicion en tablero, deben ser nulos.

Función de abstracción:

-El largo es el tamaño de cada fila del tablero.
-El alto es el tamaño[0], es decir, de cada columna, del tablero.
-La posición de inicio es la coordenada inicio de la estructura.
-La posición de llegada es la coordenada llegada de la estructura.
-El conjunto de paredes del mapa son todas las posiciones con true en la segunda componente de la tupla de infoPosicion del tablero.
-El conjunto de fantasmas del mapa son todas las posiciones con true en la primera componente de la tupla de infoPosicion del tablero.
-El conjunto de chocolates es el conjunto lineal chocolates de la estructura.

Interfaz:

Parámetros formales:

Géneros: mapa

Función:

Copiar(in m:mapa) -> res: mapa

Pre ≡ {true}

Post ≡ {res = m}

Complejidad: Θ(copy(m))

Descripción: Función copia de map's.

Se explica con: Mapa

Géneros: mapa

Operaciones básicas:

LARGO(in m: mapa) -> res: nat

Pre ≡ {true}

Post ≡ {res^ =obs largo(m^)}

Descripción: Devuelve el largo del mapa

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

ALTO(in m: mapa) -> res: nat

Pre ≡ {true}

Post ≡ {res^ =obs alto(m^)}

Descripción:Devuelve el alto del mapa

Complejidad: Θ(1)

Aliasing: No presenta aspectos de aliasing.

INICIO(in m: mapa) -> res: coordenada

Pre ≡ {true}

Post ≡ {res^ =obs inicio(m^)}

Descripción: Devuelve la coordenada de inicio del mapa
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

LLEGADA(in m: mapa) -> res: coordenada
Pre ≡ {true}
Post ≡ {res^ =obs llegada(m^)}
Descripción: Devuelve la coordenada de llegada del mapa
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

PAREDES(in m: mapa) -> res: conj(coordenada)
Pre ≡ {true}
Post ≡ {res^ =obs paredes(m^)}
Descripción: Devuelve el conjunto de las coordenadas correspondientes a las paredes que tiene el mapa
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

FANTASMAS(in m: mapa) -> res: conj(coordenada)
Pre ≡ {true}
Post ≡ {res^ =obs fantasmas(m^)}
Descripción: Devuelve el conjunto de las coordenadas correspondientes a los fantasmas que tiene el mapa
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

CHOCOLATES(in m: mapa) -> res: conj(coordenada)
Pre ≡ {true}
Post ≡ {res^ =obs chocolates(m^)}
Descripción: Devuelve el conjunto de las coordenadas correspondientes a los chocolates que tiene el mapa
Complejidad: Θ(1)
Aliasing: No presenta aspectos de aliasing.

NUEVOMAPA(in largo:nat, in alto:nat, in inic:coordenada, in lleg:coordenada, in cp:conj(coordenada), in cf:conj(coordenada), in cc:conj(coordenada)) -> res: mapa

Pre ≡ {(inicio ≠ llegada ∧ todosEnRango(paredes ∪ fantasmas ∪ chocolates ∪ {inicio, llegada}, largo, alto) ∧ {inicio, llegada} ∩
(fantasmas ∪ paredes) = ∅ ∧ disjuntosDeAPares(paredes, fantasmas, chocolates))}

Post ≡ {res^ = nuevoMapa(largo^, alto^, inic^, lleg^, cp^, cf^, cc^)}

Descripción: Devuelve un nuevo mapa, a partir de todos los elementos que lo componen.

Complejidad: O(|c|) + O(|f|) + O(|p|)

Aliasing: Devuelve referencia modificable

IMPLEMENTACIÓN: ALGORITMOS DEL MÓDULO

```
iLARGO(in m: mapa) -> res: nat
  res <- m.tablero.tam()

iALTO(in m: mapa) -> res: nat
  res <- m.tablero[0].tam()

iINICIO(in m: mapa)  -> res: coordenada
   res <- m.inicio

iLLEGADA(in m: mapa) -> res: coordenada
   res <- m.llegada

iFANTASMAS(in m: mapa) -> res: conj(coordenada)
   cf<- Vacio()
   for i=0... m.tablero.tam
      for j=0….m.tablero.tam[i]
```

```
        if(m.tablero[i][j].first == true)
            it <- cf.AGREGARRAPIDO(<i,j>)
            it.Avanzar()
  res<- cf

iPAREDES(in m: mapa) -> res: conj(coordenada)
    cp<- Vacio()
    for i=0... m.tablero.tam
        for j=0….m.tablero.tam[i]
          if(m.tablero[i][j].second == true)
            it <- cp.AGREGARRAPIDO(<i,j>)
            it.Avanzar()
     res <-  cp

iCHOCOLATES(in m: mapa) -> res: conj(coordenada)
      res<- m.chocolates

iNUEVOMAPA(in larg:nat, in alt:nat, in inic:coordenada, in lleg:coordenada, in cp:conj(coordenada), in cf:conj(coordenada), in
                    cc:conj(coordenada)) -> res: mapa
    ADexterno <- crearArreglo(alt) // O(1)
    ADinterno <- crearArreglo(larg) // O(1)
    tablero <- ADexterno[ADinterno[<false, false, false, NULL>]] // O(1), Definimos matriz de Alto * Largo, siendo cada una tripla de
                                                                    booleanos
    AGREGARELEMENTOSALTABLERO(tablero, cc, 'C') //O(|c|)
    AGREGARELEMENTOSALTABLERO(tablero, cf, 'F') //O(|f|)
    AGREGARELEMENTOSALTABLERO(tablero, cp, 'P') //O(|p|)
    inicio <- inic // O(1)
    inicio <- lleg // O(1)
    res <- <inicio, llegada, tablero, cc> // O(1)
```

IMPLEMENTACIÓN: AUXILIARES DEL MÓDULO

AGREGARELEMENTOSALTABLERO(in/out t: tablero, in c: conj(coordenada), in tipo: char)
Pre ≡ {Tipo es o ‘C’, o ‘F’, o ‘P’, cada coordenada de c es una coordenada válida (sus componentes están dentro de los límites del mapa)}
Post ≡ {Para cada posición del tablero correspondiente a la coordenada de cada elemento del conjunto del tipo pasado, esa posición tiene true en la componente correspondiente a ese tipo de la tupla de cada una de esas posiciones}
Descripción: “Agrega” los elementos al tablero, del conjunto del tipo correspondiente que se le pasa.
Aliasing: Modifica referencia que es modificable

```
AGREGARELEMENTOSALTABLERO(in/out t: tablero, in c: conj(coordenada), in tipo: char)
  if(!EsVacio?(c)) // O(1)
  while(HaySiguiente?(c.itConj(coordenada))) // O(|tipo|) * O(1),  tipo corresponde a chocolate, a fantasma o a pared
        elem <- Siguiente(c.itConj(coordenada)) // O(1)
        if(tipo == 'F') // O(1)
           tablero[elem.first][elem.second].fist <- true // O(1)
         if(tipo == 'P') // O(1)
           tablero[elem.first][elem.second].second <- true // O(1)
         if(tipo == 'C') // O(1)
           tablero[elem.first][elem.second].third <- true // O(1)
           Avanzar(c.itConj(coordenada))) // O(1)
```

Servicios usados: El mapa es el módulo de ‘más bajo nivel’ dentro de los auxiliares al ranking, por lo que se utiliza a sí mismo solamente dentro de los módulos del TP, y de módulos básicos utilizamos un Conjunto Lineal y sus iteradores