```
                                //MAPA//
Representación:
  mapa se representa con estr donde
   estr es tupla( dimensiones: Tupla( ancho:  N, alto: N),
                  Tablero: vector(vector( TipoCasillero ))
                  rs: diccTrie(color, pos))
TipoCasillero es enum { PISO, RAMPA, PARED }

Solución Informal:
- La longitud del primer vector tiene que ser igual a e.dimensiones.ancho - 1 y la de todos los vectores dentro de éste deben ser iguales a e.dimensiones.alto - 1.
- Toda rampa de la estructura debe tener al menos un piso y una pared (en este caso, con que no sea rampa o pared nos alcanza) adyacente a sí misma.
- Las posiciones de todos los significados de e.rs deben estar en rango
- No se pueden repetir posiciones en e.rs para ningún color distinto de otro.
---
Invariante de representación:
Rep: estr e -> bool
(∀e: estr) Rep(e) ≡ True ⇔ ((1) ∧ (2) ∧ (3) ∧ (4))
donde:
    (1): Longitud(e.Tablero) = e.dimensiones.ancho - 1 ∧ (∀i: Z) (0 ≤ i < e.dimensiones.ancho ⇒L Longitud(e.Tablero[i]) = e.dimensiones.alto - 1)
    (2): (∀i, j: Z) (0 ≤ i < e.dimensiones.ancho ∧ 0 ≤ j < e.dimensiones.alto ∧L e.Tablero[i][j] = RAMPA ⇒ (2') ∧ (2''))
    (2'): (∃i', j': Z) (0 ≤ i' < e.dimensiones.ancho-1 ∧ 0 ≤ j' < e.dimensiones.alto-1 ∧
    (i' = i+1 ∧ j' = j) ∨ (i' = i-1 ∧ j' = j) ∨ (i' = i ∧ j' = j-1) ∨ (i' = i ∧ j' = j+1) ∧L e.Tablero[i'][j'] = PARED)
    (2''):(∃i'', j'': Z) (0 ≤ i'' < e.dimensiones.ancho-1 ∧ 0 ≤ j'' < e.dimensiones.alto-1 ∧
    (i'' = i+1 ∧ j'' = j) ∨ (i'' = i-1 ∧ j'' = j) ∨ (i'' = i ∧ j'' = j-1) ∨ (i'' = i ∧ j'' = j+1) ∧L e.Tablero[i''][j''] = PISO)
    (3): (∀c: color) (c ∈ claves(e.rs) ⇒L 1 ≤ π1(obtener(c, e.rs)) ≤ e.dimensiones.ancho ∧ 1 ≤ π2(obtener(c, e.rs)) ≤ e.dimensiones.alto)
    (4): (∀c1,c2: color) ((def?(c1, e.rs) ∧ def?(c2, e.rs) ∧ c1 ≠ c2) ⇒L obtener(c1, e.rs) ≠ obtener(c2, e.rs))

---

Función de abstracción:
Abs: estr e -> mapa
(∀e: estr)
Abs(e) =obs m: mapa | e.dimensiones = (ancho(m), alto(m)) ∧
                      e.rs = receptáculos(m) ∧
                      (∀i, j: Z) (0 ≤ i < e.dimensiones.ancho ∧ 0 ≤ j < e.dimensiones.alto ∧L e.Tablero[i][j] = PARED ⇒ esPared(m, (i, j)) ∧
                      (∀i, j: Z) (0 ≤ i < e.dimensiones.ancho ∧ 0 ≤ j < e.dimensiones.alto ∧L e.Tablero[i][j] = RAMPA ⇒ esRampa(m, (i, j))
----
 Interfaz:
          Parametros formales:
          Géneros: mapa

Operaciones básicas:
  ancho(in m: mapa) -> res: N
    Pre ≡ {True}
    Post ≡ {res =obs ancho(m)}
  Complejidad: O(1)
  Descripción: Devuelve un natural igual al ancho del mapa
---
  alto(in m: mapa) -> res: N
    Pre ≡ {True}
    Post ≡ {res =obs alto(m)}
  Complejidad: O(1)
  Descripción: Devuelve un natural igual al alto del mapa
---
  receptáculos(in m: mapa) -> res: dicc(color, pos)
    Pre ≡ {True}
    Post ≡ {res =obs receptáculos(m)}
  Complejidad: O(1)
  Descripción: Devuelve un diccionario con los colores de las casillas y su respectiva posicion.
  Aliasing: Referencia no modificable
---
  esPared(in m: mapa, in p: pos) -> res: bool
    Pre ≡ {enRango(m,p)}
    Post ≡ {res =obs esPared(m, p)}
  Complejidad: O(1) //Ya que uso la funcion [] de la interfaz vector teniendo su posicion, en nuestro caso [][](vector,p1),p2), esto es O(1+1) = O(1)
  Descripción: Devuelve true si la posicion p es una pared
---
  esRampa(in m: mapa, in p: pos) -> res: bool
    Pre ≡ {enRango(m,p)}
    Post ≡ {res =obs esRampa(m, p)}
  Complejidad: O(1) //idem esPared
  Descripción: Devuelve true si la posicion p es una rampa
---
  nuevoMapa(in alto, ancho: N, in rs: dicc(color, pos)) -> res: mapa
    Pre ≡ {((∀c: color) def?(c, rs) ⇒L (enRango(m,obtener(c, res)))
  ∧ ((∀ c1,c2 : color) (def?(c1,rs) ∧ def?(c2,rs) ∧ c1 ≠ c2) ⇒L obtener(c1,rs) ≠ obtener(c2,rs))}
    Post ≡ {res =obs nuevoMapa(alto, ancho, rs)}
  Complejidad: O(|C| * |tamaño del dicc| + |alto|*(|ancho|*|c|))
  Descripción: Crea un mapa
---
  agregarPared(in m: mapa, in p: pos) -> res: mapa
    Pre ≡ {enRango(m,p) ∧L esPiso(m, p) ∧
           ((∀p': pos)(enRango(m, p') ∧L dist(p, p') = 1 ∧ esRampa(m, p')) ⇒L
           ((∃p": pos) p ≠ p" ∧ enRango(m, p") ∧L dist(p', p") = 1 ∧ esPiso(m, p")))}
    Post ≡ {res =obs agregarPared(m, p)}
  Complejidad: O(1) //Idem a esPared, solamente que ahora asigno ademas de preguntar y eso es o(1)
  Descripción: Agrega una pared en la posicion p
---
```

```
agregarRampa(in m: mapa, in p: pos) -> res: mapa
  Pre ≡ {enRango(m,p) ∧
         ((∃ p': pos) enRango(m, p') ∧L dist(p', p) = 1 ∧ esPared(m, p')) ∧
         ((∃ p': pos) enRango(m, p') ∧L dist(p', p) = 1 ∧ esPiso(m, p')) ∧
         ((∀p': pos)(enRango(m, p') ∧L dist(p, p') = 1 ∧ esRampa(m, p')) ⇒L
         ((∃p": pos) p ≠ p" ∧ enRango(m, p") ∧L dist(p', p") = 1 ∧ esPiso(m, p")))}}
  Post ≡ {res =obs agregarRampa(m, p)}
Complejidad: O(1) //Idem agregarPared
Descripción: Agrega una rampa en la posicion p
---
esPiso(in m: mapa, in p: pos) -> res: bool
  Pre ≡ {enRango(m,p)}
  Post ≡ {res =obs esPiso(m, p)}
Complejidad: O(1) //Idem esPared
Descripción: Devuelve true si la posicion p es un piso
---
enRango(in m: mapa, in p: pos) -> res: bool
  Pre ≡ {True}
  Post ≡ {res =obs enRango(m, p)}
Complejidad: O(1)
Descripción: Devuelve true si la posicion p se encuentra en los limites del mapa
---
dist(in p1, p2: pos) -> res: N
  Pre ≡ {True}
  Post ≡ {res =obs dist(p1, p2)}
Complejidad: O(1)
Descripción: Devuelve el calculo de la distancia entre p₁ y p₂
---
Algoritmos del módulo:
iancho(in e: estr) -> res: N
res <- e.dimensiones.ancho                          //O(1) (copia de N = O(1))
---
ialto(in e: estr) -> res: N
res <- e.dimensiones.alto                           //O(1)
---
ireceptáculos(in e: estr) -> res: dicc(color, pos)
res <- e.rs                                         //O(1)//ref no modificable para no pagar costo copia
---
iesPared(in e: estr, in p: pos) -> res: bool
res <- e.Tablero[ π₁(p) ][ π₂(p) ] = PARED          //O(1)
---
iesRampa(in e: estr, in p: pos) -> res: bool
res <- e.Tablero[ π₁(p) ][ π₂(p) ] = RAMPA          //O(1)
---
inuevoMapa(in alto, ancho: N, in rs: dicc(color, pos)) -> res: estr
res.dimensiones <- ⟨alto, ancho⟩                    //O(1)
res.rs <- rs                                        //O(|C| * |tamaño del dicc)
v <- Vector::Vector::Vacia()                        //O(1)
i <- 0                                              //O(1)
j <- 0                                              //O(1)
while i < alto do                                   //O(|alto)
fila <- Vector::Vacia()                             //O(1)
v <- AgregarAtras(v,fila)                           //O(1)
  while j < ancho do                                //O(|ancho|)
    Vector::AgregarAtras(v[i],PISO)                 //O(|c|)
    j <- j + 1                                      //O(1)
  end while
  i <- i + 1                                        //O(1)
end while
res.Tablero <- v                                    //O(|alto|*(|ancho|*|c|))
---
iagregarPared(in e: estr, in p: pos) -> res: estr
res.Tablero[ π₁(p) ][ π₂(p) ] <- PARED              //O(1)
---
iagregarRampa(in e: estr, in p: pos) -> res: estr
res.Tablero[ π₁(p) ][ π₂(p) ] <- RAMPA              //O(1)
---
iesPiso(in e: estr, in p: pos) -> res: bool
res <- e.Tablero[ π₁(p) ][ π₂(p) ] = PISO           //O(1)
---
ienRango(in e: estr, in p: pos) -> res: bool
res <- (first.p >= 1 ∧ first.p <= e.dimensiones.ancho) ∧  //O(1)
       (second.p >= 1 ∧ second.p <= e.dimensiones.alto);  //O(1)
---
idist(in p1, p2: pos) -> res: N
res <- abs(first.p1 - first.p2) +                   //O(1)
       abs(second.p1 - second.p2);                  //O(1)

---

Servicios usados:
No utiliza ningún otro módulo para cumplir con su interfaz.
```

```
                    //SIMULACION//
Representación:
sim se representa con estr donde
 estr es tupla( Mapa: mapa,
                posAgente: pos,
                ObjetosDicc: DiccTrie(color, pos),
                ObjetosMatriz: vector (vector (Tupla( Objeto:color,Destino:color ))),
                cantMovimientos: N,
                cantObjetivosCompletados: N,
                objetivosConj: conjLineal(Objetivo)
                objetivosDicc: DiccTrie(color, DiccTrie(color, itConjLineal(Objetivo)))
                )

Solución Informal:
- La posición del agente se encuentra contenida dentro de las posiciones del mapa, y no se corresponde con la posición de un objeto.
- Solo hay un objeto por posicion en e.ObjetosDicc
- Todos los objetos definidos en e.ObjetosDicc aparecen en e.ObjetosMatriz, en la misma posición de su significado.
- Toda posición de e.ObjetosMatriz que no tenga un objeto o receptáculo, tendrá como valor (en su respectiva componente de la tupla) un string vacío.
- No se pueden repetir colores de objetos ni de destinos en e.ObjetosMatriz
- Todas las posiciones de e.ObjetosMatriz deben estar en rango del mapa.
- e.cantMovimientos y e.cantObjetivosCompletados no tienen restricción
- Los colores de los objetivos dentro de e.objetivosConj deben estar incluidos en e.ObjetosMatriz
- Los conjuntos de claves de e.objetivosDicc están incluidos en las claves de e.ObjetosDicc -
- Las claves de los significados de e.objetivosDicc deben estar incluidas en las claves de receptáculos(e.Mapa), y el iterador apunta a un objetivo (dentro de
e.objetivosConj) que tiene como colorObjeto a la clave de e.objetivosDicc y como colorDestino a la clave del significado de e.objetivosDicc.

---
Invariante de representación:
Rep: estr e -> bool
(∀e: estr) (Rep(e) ≡ True ⇔ (1) ∧ (2) ∧ (3) ∧ (4) ∧ (5))
donde:
  (1): (enRango(e.Mapa, e.posAgente)) ∧ (∀c: color)(def?(c, e.ObjetosDicc) ⇒L obtener(c, e.objetosDicc) ≠ e.posAgente)
  (2): (∀c1, c2: color) (def?(c1, e.ObjetosDicc) ∧ def?(c2, e.ObjetosDicc) ∧ c1 ≠ c2 ⇒L obtener(c1, e.ObjetosDicc) ≠ obtener(c2, e.ObjetosDicc))
  (3): (∀c: color) (def?(c, e.ObjetosDicc) <=> π1(e.ObjetosMatriz[π1(obtener(c, e.ObjetosMatriz))][π2(obtener(c, e.ObjetosMatriz))]) = c)
  (4): (∀c: color)(∀i, j: Z) (enRango(e.Mapa, <i, j>) ∧ ¬def?(c, e.ObjetosDicc) ⇒L π1(e.ObjetosMatriz[i][j]) = ""))
  (4'): (∀c: color)(∀i, j: Z) (enRango(e.Mapa, <i, j>) ∧ ¬def?(receptáculos(e.Mapa)) ⇒L π2(e.ObjetosMatriz[i][j]) = ""))
  (5): (∀i, j, i', j': Z) (enRango(e.Mapa, <i, j>) ∧ enRango(e.Mapa, <i', j'>) ∧ <i, j> ≠ <i', j'> ⇒L
   ( π1(e.ObjetosMatriz[i][j]) ≠ π1(e.ObjetosMatriz[i'][j']) ∧ π2(e.ObjetosMatriz[i][j]) ≠ π2(e.ObjetosMatriz[i'][j']) ) ∨
   (e.ObjetosMatriz[i][j] = <"", "">))
  (6): Longitud(e.ObjetosMatriz) = ancho(e.Mapa)-1 ∧ (∀i: Z) (0 ≤ i < ancho(e.Mapa) ⇒L Longitud(e.ObjetosMatriz[i]) = alto(e.Mapa) - 1)
  (7): (∀obj: objetivo) (obj ∈ e.ObjetivosConj ⇒L def?(π1(obj), e.ObjetosDicc) ∧ def?(π2(obj), receptáculos(e.Mapa)))
  (8): (∀c: color) (def?(c, e.objetivosDicc) ⇒L def?(c, e.ObjetosDicc) ∧ (∀c': color) (def?(c', obtener(c, e.objetivosDicc)) ⇒L def?(c', receptáculos(e.Mapa)) ∧
       ColorObjeto(Siguiente(obtener(c', obtener(c, e.objetivosDicc)))) = c ∧ ColorDestino(Siguiente(obtener(c', obtener(c, e.objetivosDicc)))) = c')

Función de abstracción:
Abs: estr e -> sim
(∀e: estr)
Abs(e) =obs s: sim |  mapa(s) = e.Mapa ∧
                      posJugador(s) = e.posAgente ∧
                      cantMovimientos(s) = e.cantMovimientos ∧
                      objetivosDisponibles(s) = e.objetivosConj ∧
                      #objetivosRealizados(s) = e.cantObjetivosCompletados ∧
                      coloresObjetos(s) = Claves(e.objetosDicc) ∧
                      (∀c: color) (def?(c,e.objetosDicc) ⇒L (posObjeto(s,c) = obtener(c,e.objetosDicc))

---
Interfaz:
        Parametros formales:
        Géneros: sim

Operaciones básicas:
mapa(in s: sim) -> res: mapa
  Pre ≡ {True}
  Post ≡ {res =obs mapa(s)}
Complejidad: O(1)
Descripción: Devuelve el mapa de la simulacion
Aliasing: Devuelve una referencia no modificable
---
posJugador(in s: sim) -> res: pos
  Pre ≡ {True}
  Post ≡ {res =obs posJugador(s)}
Complejidad: O(1)
Descripción: Pos actual del jugador en el mapa
---
cantMovimientos(in s: sim) -> res: nat
  Pre ≡ {True}
  Post ≡ {res =obs cantMovimientos(s)}
Complejidad: O(1)
Descripción: Cantidad de movimientos hechos del jugador en la simulación
---
objetivosDisponibles(in s: sim) -> res: conj(objetivo)
  Pre ≡ {True}
  Post ≡ {res =obs objetivosDisponibles(s)}
Complejidad: O(1)
Descripción: Conjunto de los objetivos disponibles no realizados
Aliasing: Devuelve una referencia no modificable
---
```

```
#objetivosRealizados(in s: sim) -> res: nat
  Pre ≡ {True}
  Post ≡ {res =obs #objetivosRealizados(s)}
Complejidad: O(1)
Descripción: Cantidad de objetivos realizados por el jugador
---
coloresObjetos(in s: sim) -> res: conj(color)
  Pre ≡ {True}
  Post ≡ {res =obs coloresObjetos(s)}
Complejidad: O(|alto|*(ancho*|c|))
Descripción: Conjunto de colores de todos los objetos presentes en la simulacion
---
posObjeto(in s: sim, in c: color) -> res: pos
  Pre ≡ {c ∈ coloresObjetos(s)}
  Post ≡ {res =obs posObjeto(s,c)}
Complejidad:  O(|C|)
Descripción: Devuelve la posicion del objeto
---
nuevaSimulacion(in m: mapa, in p: pos, in objetos: dicc(color,pos)) -> res: sim
  Pre ≡ {(enRango(m, p) ∧ ((∀c: color) (def?(c, objetos)) ⇒L
         (enRango(m, obtener(c, objetos)) ∧ obtener(c, objetos) ≠ p)) ∧
         (∀c1, c2: color) (def?(c, objetos) ∧ c1 ≠ c2 ⇒L obtener(c1, objetos) ≠ obtener(c2, objetos))) }
  Post ≡ {res =obs nuevaSimulacion(m,p,objetos)}
Complejidad: O(|tamaño dicc objetos|*|c| + |tamaño dicc receptaculos|*|c| + (|ancho|*|c|)*|alto|)
Descripción: Crea una simulación
---
mover(in s: sim, in d: dir) -> res: sim
  Pre ≡ { enRango(m,siguientePosición(posJugador(s),d))}
  Post ≡ {res =obs mover(s,d) }
Complejidad: O(|C|)
Descripción: Devuelve true si se pudo mover al jugador una posicion en una direccion
---
agObjetivo(in s: sim, in o: objetivo) -> res: sim
  Pre ≡ {colorObjeto(o) ∈ coloresObjetos(s) ∧ def?(colorDestino(o), receptáculos(mapa(s))}
  Post ≡ {res =obs agObjetivo(s,d)}
Complejidad: O(|C|)
Descripción: Agrega el objeto a la simulacion
---
hayMovimiento(in p: pos, in d: dir, in m: mapa) -> res: bool
  Pre ≡ {True}
  Post ≡ {res =obs hayMovimiento(p,d,m)}
Complejidad: O(1)
---
siguientePosición(in p: pos, in d: dir) -> res: pos
  Pre ≡ {True}
  Post ≡ {res =obs siguientePosición(p,d)}
Complejidad: O(1)
Descripción: Depende la dir, suma/resta 1 a la posicion
---
filtrarRealizados(in co: conj(objetivo), in s: sim) -> res: conj(objetivo)
  Pre ≡ {True}
  Post ≡ {res =obs filtrarRealizados(co,s)}
Complejidad: O(|co| * |c|)
Descripción: Filtra los objetivos realizados
---
hayObjeto(in p: pos, in s: sim) -> res: bool
  Pre ≡ {enRango(mapa(s),p))}
  Post ≡ {(res =obs hayObjeto(p,s))}
Complejidad: O((ancho*|c|)*alto)
---
hayAlgúnObjeto(in p: pos, in cs: conj(color), in s: sim) -> res: bool
  Pre ≡ {enRango(mapa(s),p) ∧ cs ⊆ coloresObjetos(s)) }
  Post ≡ {(res =obs hayObjeto(p,s))}
Complejidad: O(1)
---
Algoritmos del módulo:
imapa(in e: estr) -> res: mapa
res <- e.Mapa                                          //O(1)//Devuelto como ref no modif para no pagar costo de copia
---
iposJugador(in e: estr) -> res: pos
res <- e.posAgente                                     //O(1)
---
icantMovimientos(in e: estr) -> res: nat
res <- e.cantMovimientos                               //O(1)
---
iobjetivosDisponibles(in e: estr) -> res: conj(objetivo)
res <- e.objetivosConj                                 //O(1)//Devuelto como ref no modif para no pagar costo de copia
---
i#objetivosRealizados(in e: estr) -> res: nat
res <- e.cantObjetivosCompletados                      //O(1)
---
```

```
icoloresObjetos(in e: estr) -> res: conj(color)
res <- ConjLineal::Vacio()                                          //O(1)
i <- 0                                                              //O(1)
j <- 0                                                              //O(1)
while i < Mapa::Alto(e.mapa) do                                     //O(|alto|)       Costo Ciclo: O(|alto|*(ancho*|c|))
  while j < Mapa::Ancho(e.mapa)  do                                 //O(ancho)
    if e.ObjetosMatriz [i] [j] != "" then                           //O(1)
      ConjLineal::AgregarRapido(res,e.ObjetosMatriz [i] [j].Objeto)//O(|c|)
      j <- j +1                                                     //O(1)
    else
      j <- j+1                                                      //O(1)
    fi
  end while
  i <- i+1                                                          //O(1)
end while
---
iposObjeto(in e: estr, in c: color) -> res: pos
res <- DiccTrie::Significado(e.ObjetosDicc,c)                       //O(|C|)
---
inuevaSimulacion(in m: mapa, in p: pos, in objetos: dicc(color,pos)) -> res: estr
res.Mapa <- m                                                        //O((tamaño de Receptaculos * |c|) + (alto*ancho))
res.posAgente <- p                                                   //O(1)
res.ObjetosDicc <- objetos                                           //O(tamaño del diccLineal Objetos * |c|)

while i < alto do                                                    //O((ancho*|c|)*alto)
fila <- Vector::Vacia()                                              //O(1)
AgregarAtras(res.ObjetosMatriz,fila)                                 //O(1)
  while j < ancho do                                                 //O(ancho)
    Vector::AgregarAtras(res.ObjetosMatriz[i],<"","">)               //O(|c|)
    j <- j + 1                                                       //O(1)
  end while
  i <- i + 1                                                         //O(1)
end while

it <- itDiccLineal::CrearIt(objetos)                                 //O(1)
while itDiccLineal::HaySiguiente(it) do                              //O(tamaño del dicclineal objetos)
  posicion <- itDiccLineal::SiguienteClave(it)                       //O(1)
  color <- itDiccLineal::SiguienteSignificado(it)                    //O(1)
  e.ObjetosMatriz [ π1(posicion ] [ π2(posicion) ].Objeto <- color   //O(|c|)
  itDiccLineal::Siguiente(it)                                        //O(1)
end while

it <- itDiccLineal::CrearIt(Mapa::Receptaculos(e.mapa))              //O(1)
while itDiccLineal::HaySiguiente(it) do                              //O(tamaño del dicctrie receptaculos)
  posicion <- itDiccLineal::SiguienteClave(it)                       //O(1)
  color <- itDiccLineal::SiguienteSignificado(it)                    //O(1)
  e.ObjetosMatriz [ π1(posicion ] [ π2(posicion) ].destino <- color //O(|c|)
  itDiccLineal::Siguiente(it)                                        //O(1)
end while

res.cantMovimientos <- 0                                             //O(1)
res.cantObjetivosCompletados <- 0                                    //O(1)
res.objetivosConj <- ∅                                               //O(1)
res.objetivosDicc <- diccTrie::Vacio()                               //O(1)
---
```

```
imover(inout e: estr, in d: dir)
sigPosAgente <- siguientePosición(e.posAgente, d)                                   //O(1)
if Simulacion::hayMovimiento(e.posAgente,d,e.Mapa) then                             //O(1) || CASO 1(AGENTE):Se puede mover
    if e.ObjetosMatriz [ π1(sigPosAgente) ] [ π2(sigPosAgente) ].Objeto != ""       //O(1) || CASO 1(OBJETO): Hay un objeto donde se quiere mover
     then
      sigPosObjeto <- siguientePosición(siguientePosición(e.posAgente, d),d)        //O(1)
      colorObjeto <- e.ObjetosMatriz [ π1(sigPosAgente) ] [ π2(sigPosAgente) ].Objeto   //O(|C|)
      if Simulacion::hayMovimiento(sigPosAgente,d,e.Mapa) ∧                         //O(1) || El objeto se puede mover
          e.ObjetosMatriz [ π1(sigPosObjeto) ] [ π2(sigPosObjeto) ].Objeto = ""     //O(1) || No hay dos objetos contiguos
      then
        e.posAgente <- sigPosAgente                                                 //O(1)
        e.ObjetosMatriz [ π1(sigPosObjeto) ] [ π2(sigPosObjeto) ].Objeto <- colorObjeto //O(|C|)  || Actualizo e.ObjetosMatriz moviendo el objeto
        e.ObjetosMatriz [ π1(sigPosAgente) ] [ π2(sigPosAgente) ].Objeto <- ""      //O(|C|)  ||
        DiccTrie::definir(e.ObjetosDicc,colorObjeto,sigPosObjeto)                   //O(|C|)  || Actualizo e.ObjetosDicc redefiniendo el objeto con la nueva posicion

        //Seccion Objetivos
        colorDestino <- e.ObjetosMatriz [ π1(sigPosObjeto) ] [ π2(sigPosObjeto) ].Destino  //O(1)
        if Dicctrie::def?(e.ObjetivosDicc,colorObjeto) then                         //O(|C|) || Pregunto si ese objeto tiene un objetivo
           if Objetivos::Realizado?(<colorObjeto,colorDestino>)                     //O(|C|+|C|) || Como tiene, armo una tupla con su Color Destino, Color Objeto y pregu
            then
            res.cantObjetivosCompletados <- res.cantObjetivosCompletados + 1                                    //O(1)
            itConjLineal::EliminarSiguiente(Dicctrie::significado(DiccTrie::significado(e.ObjetivosDicc,colorObjeto),colorDestino)  //O(1)  || Actualizo e.objetivosConj con el
            DiccTrie::eliminar(DiccTrie::significado(e.ObjetivosDicc,colorObjeto),colorDestino)                 //O(|C|)||Actualizo e.ObjetivosDicc
           else
            -
           fi
        else
          -
        fi
      else
        -
      fi
    else                                                                            //     || CASO 2(OBJETO): NO hay un objeto donde se quiere mover
      e.posAgente <- sigPosAgente                                                   //O(1)
    fi
  else                                                                              //     || CASO 2(AGENTE): NO se pudo mover
  -
fi
e.cantMovimientos <- e.cantMovimientos + 1                                          // O(1)


---
iagObjetivo(inout e: estr, in o: objetivo)
if DiccTrie::def?(e.ObjetivosDicc,Objetivo::ColorObjeto(o)))                                        //O(|c|)
 then                                                                                               //CASO 1:Objetivo existe(hay objeto en e.ObjetivosDicc pero puede
  if DiccTrie::def?(DiccTrie::significado(e.ObjetivosDicc,Objetivo::ColorObjeto(o)),Objetivo::ColorDestino(o))  // O(|c|)
  then                                                                                              //CASO 1: Objetivo con destino ya existente(no hacer nada)
    -
  else                                                                                               //CASO 2: Objetivo con destino no existente(hay que actualizar
  it <- conjLineal::AgregarRapido(e.objetivosConj,o)                                                 //O(1)     ||Como se que no existe agrego rapido y me guardo e
  DiccTrie::definir(DiccTrie::significado(e.ObjetivosDicc,Objetivo::ColorObjeto(o)),Objetivo::ColorDestino(o),it) //O(|c|)   ||Defino ColorDestino con su it a e.ObjetivosConj,
  fi
else                                                                                    //CASO 2:Objetivo no existe(No hay objeto en e.ObjetivosDicc)
  it <- conjLineal::AgregarRapido(e.objetivosConj,o)                                    //O(1)     ||Como se que no existe agrego rapido y me guardo el it
  DiccDestinoIt <- DiccTrie::Vacio()                                                    //O(1)     ||Creo nuevo dicctrie para ese objeto ya que no existia
  DiccTrie::definir(DiccDestinoIt,Objetivo::ColorDestino(o),it)                         //O(|c|)   ||Defino ColorDestino con su it a e.ObjetivosConj
  ObjetoyDiccDest <- <Objetivo::ColorDestino(o),DiccDestinoIt>                          //O(|c|)   ||Me hago una tupla <ColorDestino,DiccDestinoIt> para definirla
  DiccTrie::definir(e.ObjetivosDicc,Objetivo::ColorObjeto(o),ObjetoyDiccDest)           //O(|c|)   ||Actualizo e.ObjetivosDicc
fi
---
ihayMovimiento(in p: pos, in d: dir, in m: mapa) -> res: bool
res <- Mapa::enRango(m, siguientePosición(p, d)) ∧L              // O(1)
(Mapa::esPared(m, siguientePosición(p, d)) ⇒ Mapa::esRampa(m, p)) // O(1)
---
isiguientePosición(in p: pos, in d: dir) -> res: pos
res <- p                                                        //O(1)
if dir = ARRIBA then
  res.first <- res.first + 1
else
  if dir = ABAJO then
    res.first <- res.first - 1
  else
    if dir = DER then
      res.second <- res.second + 1
    else
      res.second <- res.second - 1
    fi
  fi
fi

---
```

```
ifiltrarRealizados(inout co: conj(objetivo), in e: estr)
it <- itConjLineal::CrearIt(co)                                  //O(1)
while itConjLineal::HaySiguiente(it)  do                         //O(|co|)
  if Objetivo::realizado?(itConjLineal::Siguiente(it))           //O(|c|)
  then
    itConjLineal::EliminarSiguiente(it)                          //O(1)
  else
    -
  fi
  itConjLineal::Avanzar(it)                                      //O(1)
end while
---
ihayObjeto(in p: pos, in e: estr) -> res: bool
res <- hayAlgúnObjeto(p,Simulacion::coloresObjetos(e),e)                                   //O((ancho*|c|)*alto)
---
ihayAlgúnObjeto(in p: pos, in cs: conj(color), in e: estr) -> res: bool
res <- e.ObjetosMatriz [ π1(p) ] [ π2(p) ].Objeto != ""                                    //O(1)
---


Servicios usados:
El módulo Simulación utiliza el módulo mapa, específicamente sus operaciones "receptáculos", "esPared", "esRampa" y "enRango",
además del módulo Objetivo, del cuál hace uso de las operaciones "colorObjeto" y "colorDestino".

---

                      //Objetivo//
Representación:
Objetivo se representa con estr donde

 estr es tupla ( obj: color,
                 dest: color )

---
Invariante de representación:
Rep: estr e -> bool
(∀e: estr) (Rep(e) ≡ True)
---

Función de abstracción:
Abs: estr e -> Objetivo
(∀e: estr)  Abs(e) =obs o: objetivo | e.obj = colorObjeto(o) ∧
                                      e.dest = colorDestino(o)
---

Interfaz:
        Parametros formales:
        Géneros: objetivo

Operaciones básicas:
colorObjeto(in o: objetivo) -> res: color
  Pre ≡ {True}
  Post ≡ {res =obs colorObjeto(o)}
Complejidad: O(|c|)
Descripción: Devuelve el color del objeto
---
colorDestino(in o: objetivo) -> res: color
  Pre ≡ {True}
  Post ≡ {res =obs colorDestino(o)}
Complejidad: O(|c|)
Descripción: Devuelve el color de la casilla destino
---
nuevoObjetivo(in c1: color, in c2: color) -> res: objetivo
  Pre ≡ {True}
  Post ≡ {res =obs nuevoObjetivo(c1,c2)}
Complejidad: O(|C|)
Descripción: Crea un objetivo
---
realizado?(in o: objetivo, in s: sim ) -> res: bool
    Pre ≡ {colorObjeto(o) ∈ coloresObjetos(s) ∧
           def?(colorDestino(o), receptáculos(mapa(s)))}
    Post ≡ {res =obs realizado?(o,s)}
Complejidad: O(|c|)
Descripción: Devuelve true si el objetivo se realizo(si la posicion del objeto esta en la posicion destino)
---
Algoritmos del módulo:
icolorObjeto(in e: estr) -> res: color
res <- e.obj                                                 //O(|c|)
---
icolorDestino(in e: estr) -> res: color
res <- e.dest                                                //O(|c|)
---
inuevoObjetivo(in c1: color, in c2: color) -> res: estr
res.obj <- c1                                                //O(|C|)
res.dest <- c2                                               //O(|C|)
---
```

```
irealizado?(in e: estr, in s: sim ) -> res: bool
res <- DiccTrie::Significado(Mapa::receptáculos(Simulacion::Mapa(s)),e.dest) = Simulacion::posObjeto(s,e.obj);  //O(|c|+|c|)
---

Servicios usados:
En la operación básica "realizado?" utiliza el módulo Simulación, que
a su vez utiliza el módulo Mapa, accediendo a las operaciones "receptáculos",
"coloresObjetos" y "posObjeto".

------------
//Modulos Auxiliares(DiccTrie y ConjTrie)
          //Dicctrie
Interfaz:
Parámetros formales:
Función:
•=•(in k₁ : κ, in k₂ : κ) -> res: bool
  Pre ≡ {true}
  Post ≡ {res =obs (k₁ = k₂)}
Complejidad: Θ(equal(k₁, k₂))
Descripción: función de igualdad de κ's
---
Función:
Copiar(in k : κ) -> res: κ
  Pre ≡ {true}
  Post ≡ {res =obs k}
Complejidad: Θ(copy(k))
Descripción: función de copia de κ's
---
Función:
Copiar(in s : σ) -> res: σ
  Pre ≡ {true}
  Post ≡ {res =obs s}
Complejidad: Θ(copy(s))
Descripción: función de copia de σ's
---
Se explica con: TAD Diccionario
Géneros: diccTrie(κ, σ)

Operaciones básicas:

Vacío() -> res: diccTrie
  Pre ≡ {true}
  Post ≡ {res =obs vacío()}
Complejidad: O(1)
Descripción: genera un diccionario vacío.
---
Def?(in dicc: diccTrie, in k : β) -> res: bool
  Pre ≡ {true}
  Post ≡ {res =obs def?(k, dicc) }
Complejidad: O(|c|)
Descripción: Devuelve true si y solo si k está definida en el diccionario dicc.
---
Definir(inout dicc: diccTrie, in k: β, in s : α) -> res: itDiccTrie
  Pre ≡ {dicc = dicc₀}
  Post ≡ {dicc =obs definir(dicc₀, k, s) ∧ haySiguiente(res) ∧L Siguiente(res) == ⟨k, s⟩ ∧
          alias(esPermutación(SecuSuby(res), d))}
Complejidad: O(|c|)
Descripción: Define la clave k con el significado s en el diccionario dicc.
Retorna un iterador al elemento recién agregado.
---
Significado(in dicc: diccTrie, in k: β) -> res: α
  Pre ≡ {def?(k, dicc) }
  Post ≡ {res =obs obtener(k, dicc) }
Complejidad: O(|c|)
Descripción: Devuelve el significado de la clave k en el diccionario dicc.
---
Borrar(in dicc: diccTrie, in k : β) -> res: diccTrie
  Pre ≡ {def?(k, clave)}
  Post ≡ {res =obs borrar(k, dicc)}
Complejidad: O(|c|)
Descripción: Elimina del diccionario dicc, la clave k y su significado.
---
Operaciones Del Iterador:

CrearIt(in d : diccTrie) -> res: itDicc
  Pre ≡ {true}
  Post ≡ {alias(esPermutación(SecuSuby(res), d)) ∧ vacia?(Anteriores(res))}
Complejidad: O(1)
Descripción: Crea un iterador bidireccional del diccionario, de forma tal que
HayAnterior evalúe a false (i.e., que se pueda recorrer los elementos aplicando
iterativamente Siguiente)
---
HaySiguiente(in it : itDiccTrie) -> res: bool
  Pre ≡ {true}
  Post ≡ {res =obs haySiguiente?(it)}
Complejidad: O(1)
Descripción: Devuelve true sii en el iterador todavía quedan elemento para avanzar.
---
```

```
HayAnterior(in it : itDiccTrie) -> res: bool
  Pre ≡ {true}
  Post ≡ {res =obs hayAnterior?(it)}
Complejidad: O(1)
Descripción: Devuelve true sii en el iterador todavía quedan elementos para retroceder.
---
Siguiente(in it : itDiccTrie(k, σ)) -> res: tupla(k, σ)
  Pre ≡ {HaySiguiente?(it)}
  Post ≡ {alias(res =obs Siguiente(it))}
Complejidad: O(1)
Descripción: Devuelve el elemento siguiente del iterador.
---
SiguienteClave(in it: itDiccTrie(k, σ)) -> res: k
  Pre ≡ {HaySiguiente?(it)}
  Post ≡ {alias(res =obs Siguiente(it).clave)}
Complejidad: O(1)
Descripción: Devuelve la clave siguiente del iterador.
---
SiguienteSignificado(in it: itDiccTrie(k, σ)) -> res: σ
  Pre ≡ {HaySiguiente?(it)}
  Post ≡ {alias(res =obs Siguiente(it).significado)}
Complejidad: O(1)
Descripción: Devuelve el significado siguiente del iterador.
---
Avanzar(inout it : itDiccTrie)
  Pre ≡ {it = it₀ ∧ HaySiguiente?(it)}
  Post ≡ {it =obs Avanzar(it₀)}
Complejidad: O(1)
Descripción: Avanza a la posición siguiente del iterador.
---
EliminarSiguiente(inout it : itDiccTrie)
  Pre ≡ {it = it₀ ∧ HaySiguiente?(it)}
  Post ≡ {it =obs EliminarSiguiente(it₀)}
Complejidad: O(1)
Descripción: Elimina del diccionario la clave del elemento que se encuentra en la posición siguiente.
---
```