

Descripción: crea un iterador unidireccional de los jugadores validos o expulsados.

Aliasing: el iterador se invalida si se expulsa al siguiente jugador del iterador, o si se agrega un nuevo jugador y el iterador había llegado al final. Además, siguientes(*res*) podría cambiar completamente ante cualquier operación que modifique la lista de jugadores.

HAYMAS(in *it*: itJugadores) → *res* : bool

Pre ≡ {true}

Post ≡ {*res* =_{obs} hayMas?(*it*)}

Complejidad: $O(J)$

Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

ACTUAL(in *it*: itJugadores) → *res* : jugador

Pre ≡ {HayMas?(*it*)}

Post ≡ {*res* =_{obs} Actual(*it*)}

Complejidad: $O(J)$

Descripción: devuelve el elemento siguiente a la posición del iterador.

AVANZAR(in/out *it*: itJugadores)

Pre ≡ {*it* = *it*₀ ∧ HayMas?(*it*)}

Post ≡ {*it* =_{obs} Avanzar(*it*₀)}

Complejidad: $O(J)$

Descripción: avanza a la posición siguiente del iterador.

Representación

Representación de Juego

Juego se representa con pokego

donde pokego es tupla(*mapa*: mapa , *pokemons*: diccString(nat) , *posConPokemons*: conj(coor) ,
jugadores: vector(puntero(infoJugador)) ,
grillaPos: arreglo_dimensionable de (arreglo_dimensionable de infoPos) ,
cantPokemons: nat)

donde infoJugador es tupla(*sanciones*: nat , *conectado?*: bool , *posicion*: coor , *cantPokemons*: nat ,
pokemonsCapturados: diccString(nat))

donde infoPos es tupla(*hayPokemon?*: bool , *pokemon*: pokemon , *contadorCaptura*: nat , *jugEsperandoCaptura*: dicPrior , *jugsEnPos*: conjJugs)

Invariante de Representación del Juego

- cola !!
1. La grilla tiene el alto y ancho del mapa
 2. Los jugadores conectados están en la grilla en la posición en la que se encuentra
 3. En los jugadores que se encuentran en la grilla solo están los que realmente se encuentran ahí
 4. Solo las posiciones con pokemon tienen pokemons en la grilla
 5. CantPokemons es igual a la suma de todas los significados del diccionario
suma
 6. Los significados del diccionario de pokemons es igual a la cantidad de pokemons que hay capturados mas los salvajes
 7. Los jugadores tienen menos de 5 sanciones → warco...?
 8. No hay dos coordenadas pertenecientes a posConPokemons a menos de 5 de distancia
 9. Para cada una de las claves de jugsEsperandoCaptura, el jugador es un jugador válido y se encuentra en posición de captura del pokemon correspondiente
 10. Para cada una de las claves de jugsEsperandoCaptura, su significado es la cantidad de pokemons

Aliasing: res no es modificable.

POKÉMONENPOS($\text{in } j : \text{juego}, \text{in } c : \text{coor}$) $\rightarrow res : \text{pokemon}$

Pre $\equiv \{c \in \text{posConPokemon}(j)\}$

Post $\equiv \{res =_{\text{obs}} \text{pokemonEnPos}(c, j)\}$

Complejidad: $O(1)$

Descripción: devuelve el pokémon que se encuentra en la posición.

PUEDOAGREGARPOKÉMON($\text{in } j : \text{juego}, \text{in } c : \text{coor}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{puedoAregarPokémon}(c, j)\}$

Complejidad: $O(PS)$

Descripción: devuelve si un nuevo pokémon puede ser agregado a esa posición (no debe haber ningún pokémon cerca).

HAYPOKÉMONCERCANO($\text{in } j : \text{juego}, \text{in } c : \text{coor}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{hayPokémonCercano}(c, j)\}$

Complejidad: $O(PS)$

Descripción: devuelve si hay algún pokémon cerca de la posición.

POSPOKÉMONCERCANO($\text{in } j : \text{juego}, \text{in } c : \text{coor}$) $\rightarrow res : \text{coor}$

Pre $\equiv \{\text{hayPokémonCercano}(c, j)\}$

Post $\equiv \{res =_{\text{obs}} \text{posPokémonCercano}(c, j)\}$

Complejidad: $O(PS)$

Descripción: devuelve el pokémon que se encuentra cerca de la posición.

ENTRENADORESPOSIBLES($\text{in } j : \text{juego}, \text{in } c : \text{coor}$) $\rightarrow res : \text{conj}(\text{jugador})$

Pre $\equiv \{\text{hayPokémonCercano}(c, j)\}$

Post $\equiv \{res =_{\text{obs}} \text{entrenadoresPosibles}(c, j)\}$

Complejidad: $O(EC)$

Descripción: devuelve el conjunto de jugadores esperando a capturar el pokémon que se encuentra en la posición.

Aliasing: res no es modificable.

INDICRAREZA($\text{in } j : \text{juego}, \text{in } pk : \text{pokemon}$) $\rightarrow res : \text{nat}$

Pre $\equiv \{pk \in \text{pokemons}(j)\}$

Post $\equiv \{res =_{\text{obs}} \text{indiceRareza}(pk, j)\}$

Complejidad: $O(|P|)$

Descripción: calcula el índice de rareza de un pokémon.

CANTPOKÉMONSTOTALES($\text{in } j : \text{juego}$) $\rightarrow res : \text{nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{cantPokémonsTotales}(j)\}$

Complejidad: $O(1)$

Descripción: devuelve la cantidad de pokémons en juego.

CANTMISMAESPECIE($\text{in } j : \text{juego}, \text{in } pk : \text{pokemon}$) $\rightarrow res : \text{nat}$

Pre $\equiv \{pk \in \text{pokemons}(j)\}$

Post $\equiv \{res =_{\text{obs}} \text{cantMismaEspecie}(j)\}$

Complejidad: $O(|P|)$

Descripción: devuelve la cantidad de pokémons de la especie especificada en juego.

Operaciones del iterador de jugadores

El iterador que presentamos permite recorrer tanto los jugadores registrados válidos como los expulsados de forma unidireccional. El iterador es solo un contador (devuelve las IDs de los jugadores, no su detalle).

CREARIT($\text{in } j : \text{juego}, \text{in } \text{elim?} : \text{bool}$) $\rightarrow res : \text{itJugadores}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{true}\}$

Complejidad: $O(1)$

iAregarPokémon (**in/out** j : pokego, **in** pk : pokemon, **in** c : coor)

```

j.cantPokemons ← j.cantPokemons + 1;  $O(1)$ 
if Definido( $j.pokemons$ ,  $pk$ ) then
| nat: nuevaCant ← Obtener( $j.pokemons$ ,  $pk$ ) + 1;  $O(|pk|)$ 
| Definir( $j.pokemons$ ,  $pk$ , nuevaCant);  $O(|pk|)$ 
else
| Definir( $j.pokemons$ ,  $pk$ , 1);  $O(|pk|)$ 
end if
j.grillaPos[Latitud( $c$ )][Longitud( $c$ )].hayPokemon ← true;  $O(1)$ 
j.grillaPos[Latitud( $c$ )][Longitud( $c$ )].pokemon ←  $pk$ ;  $O(1)$ 
j.grillaPos[Latitud( $c$ )][Longitud( $c$ )].contadorCaptura ← 0;  $O(1)$ 
/* Se desestima la complejidad de borrar el diccionario de prioridad anterior */ */
j.grillaPos[Latitud( $c$ )][Longitud( $c$ )].jugEsperandoCaptura ← Vacio();  $O(1)$ 
conj(coor) : coorEnRango ← PosicionesEnRango( $j$ ,  $c$ , 2);  $O(1)$ 
itBi(coor) : itCoor ← CrearIt(coorEnRango);  $O(1)$ 
while HaySigiente(itCoor) do  $O(EC \times log(EC))$ 
| coor: d ← Sigiente(itCoor);  $O(1)$ 
| itConjJugs : it ← CrearIterador(j.grillaPos[Latitud( $d$ )][Longitud( $d$ )].jugsEnPos);  $O(1)$ 
| while HayMas(it) do  $O(EC \times log(EC))$ 
| | jugador: jug ← Actual(it);  $O(1)$ 
| | Definir(j.grillaPos[Latitud( $c$ )][Longitud( $c$ )].jugEsperandoCaptura, jug,  $O(log(EC))$ 
| | (* $j.jugadores[jug]$ ).cantPokemons);  $O(1)$ 
| | Avanzar(it);  $O(1)$ 
| end while
| Avanzar(itCoor);  $O(1)$ 
end while

```

Complejidad: $O(|P| + EC \times log(EC))$

Justificación: todos los jugadores que se encuentran en el area (EC) deben agregarse a la lista de espera, que los ordena por prioridad (inscripción en $\log(EC)$). Al mismo tiempo debe definirse este nuevo pokémon en el diccionario global ($|P|$). $|pk|$ que está acotado por $|P|$

iAregarJugador (**in/out** j : pokego) → res: nat

```

res ← Longitud( $j.jugadores$ );  $O(1)$ 
AregarAtras( $j.jugadores$ , CrearInfoJugador());  $O(J)$ 

```

Complejidad: $O(J)$

Justificación: en el peor caso se debe redimensionar el vector. Hacerlo requiere copiar el arreglo interno, pero al tratarse de punteros la copia es gratuita ($\Theta(1)$ por posición, o $\Theta(J)$ en su totalidad).

iConectarse (**in/out** j : pokego, **in** e : jugador, **in** c : coor)

```

(* $j.jugadores[e]$ ).conectado? ← true;  $O(1)$ 
(* $j.jugadores[e]$ ).posicion ←  $c$ ;  $O(1)$ 
Aregar(j.grillaPos[Latitud( $c$ )][Longitud( $c$ )].jugsEnPos,  $e$ );  $O(log(EC))$ 
AregarACola( $j$ ,  $e$ );  $O(1)$ 
ResetearContadores( $j$ ,  $e$ );

```

OK ~~OK~~ $O(log(EC))$

Complejidad: $O(log(EC))$

Justificación: al conectarse, el jugador debe agregarse al conjunto de jugadores en c , y unirse a la cola de espera de captura de haber un pokémon cerca. De ser ese el caso, la cantidad de jugadores en la cola de espera será siempre mayor a la cantidad de jugadores en c (todos los jugadores en c están en la cola de espera).

iDesconectarse (**in/out** j : pokego, **in** e : jugador)

```

(* $j.jugadores[e]$ ).conectado? ← false;  $O(1)$ 
Borrar(j.grillaPos[Latitud( $c$ )][Longitud( $c$ )].jugsEnPos,  $e$ );  $O(log(EC))$ 
RemoverDeCola( $j$ ,  $e$ );  $O(log(EC))$ 

```

iMoverse (in/out j : pokego, in e : jugador, in c : coor)

```

coor : posAnterior ← Posicion(j,e);                                O(1)
/* Removemos al jugador del conjunto de su posición anterior y de la cola de espera */ 
Borrar(j.grillaPos[Latitud(c)][Longitud(c)].jugsEnPos, e);          O(log(EC))
RemoverDeCola(j, e);                                                 O(log(EC))
if not HayCamino(j.mapa, posAnterior, c) or DistEuclidea(posAnterior, c) > 100 then
    (*j.jugadores[e]).sanciones ← (*j.jugadores[e]).sanciones + 1;      O(1)
end if
if (*j.jugadores[e]).sanciones = 5 then
    /* Si el jugador debe ser eliminado, borramos sus pokémons */
    itDiccString(nat) : pokesABorrar ← CrearIt((*j.jugadores[e]).pokemonsCapturados);   O(1)
    while HaySiguiente(pokesABorrar) do
        tupla(clave: String, significado: Nat) : sig ← Siguiente(pokesABorrar);
        nat: nuevaCant ← Obtener(j.pokemons, sig.clave) - sig.significado;
        Definir(j.pokemons, pk, nuevaCant);
        j.cantPokemons ← j.cantPokemons - sig.significado;           ↗ y si nuevaCant == 0 O(PC × |P|) →
    end while
else
    /* Si el jugador sigue siendo válido, lo agregamos al conjunto de su nueva posición y a la
       cola de espera */
    (*j.jugadores[e]).posicion ← c;                                     O(1)
    AgregarACola(j, e, c);                                            O(log(EC))
    Agregar(j.grillaPos[Latitud(c)][Longitud(c)].jugsEnPos, e);      O(log(EC))
end if
it ← CrearIt(j.posConPokemons);                                      O(1)
/* Iteramos las posiciones con pokémons
while HaySiguiente?(it) do
    coor: coorConPk ← Siguiente(it);                                 O(PS × |P|) →
    if DistEuclidea(c, coorConPk) > 4 then
        /* Si el movimiento es lejano, sumar al contador de manejar captura si corresponde */
        j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)].contadorCaptura ←
            j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)].contadorCaptura + 1;   O(1)
        infoPos: posPk ← j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)];         O(1)
        if posPk.contadorCaptura = 10 then
            pokemon: pk ← posPk.pokemon;                                         O(1)
            jugador: captor ← Menor(posPk.jugEsperandoCaptura);                  O(1)
            (*j.jugadores[captor]).cantPokemons ← (*j.jugadores[captor]).cantPokemons + 1; O(1)
            if Definido((*j.jugadores[captor]).pokemonsCapturados, pk) then
                nat: nuevaCant ← Obtener((*j.jugadores[captor]).pokemonsCapturados, pk) + 1; O(log(|P|)) →
                Definir((*j.jugadores[captor]).pokemonsCapturados, pk, nuevaCant);       O(log(|P|)) →
            else
                Definir((*j.jugadores[captor]).pokemonsCapturados, pk, 1);           O(log(|P|)) →
            end if
            /* Si se elimina esa posición, no hay que avanzar el iterador */
            EliminarSiguiente(it);                                              O(1)
        else
            Avanzar(it);                                                       O(1)
        end if
    else
        if DistEuclidea(posAnterior, coorConPk) > 4 then
            /* Si el jugador entró a una nueva área de captura, reiniciar el contador */
            j.grillaPos[Latitud(coorConPk)][Longitud(coorConPk)].contadorCaptura ← 0;   O(1)
        end if
        Avanzar(it);                                                       O(1)
    end if
end while

```

iHayPokémonCercano (**in** j : pokego, **in** c : coor) \rightarrow res: bool

res \leftarrow HayPokémonEnDistancia(j , c , 2);

$O(1)$

iPosPokémonCercano (**in** j : pokego, **in** c : coor) \rightarrow res: coor

conj(coor) : coorEnRango \leftarrow PosicionesEnRango(j , c , 2);

$O(1)$

itBi(coor) : itCoor \leftarrow CrearIt(coorEnRango);

$O(1)$

while $HaySiguiente(itCoor)$ **do**

$O(n^2)$

 coor : siguiente \leftarrow Siguiente(itCoor);

$O(1)$

if $HayPokémonEnPos(j, siguiente) \leq n$ **then**

 | res \leftarrow siguiente;

end if

 Avanzar(itCoor);

$O(1)$

end while

$O(1)$

Complejidad: $O(1)$

Justificación: solo debe iterar todas las posiciones de un cuadrado de lado predeterminado. Al ser predeterminado se considera constante.

iEntrenadoresPosibles (**in** j : pokego, **in** c : coor) \rightarrow res: conj(jugador)

res \leftarrow Claves(j .grillaPos[Latitud(c)][Longitud(c)].jugEsperandoCaptura)

Complejidad: $O(EC)$

Justificación: el diccionario debe recuperar las claves iterando.

iCantPokémonsTotales (**in** j : pokego) \rightarrow res: nat

res \leftarrow j .cantPokemons;

$O(1)$

Complejidad: $O(1)$

Justificación: solo se retorna un valor almacenado.

iIndiceRareza (**in** j : pokego, **in** pk : pokemon) \rightarrow res: nat

nat : cantPk \leftarrow CantMismaEspecie(j , pk);

$O(|P|)$

res \leftarrow $100 - (cantPk \times 100 \div j.cantPokemons)$;

$O(1)$

iCantMismaEspecie (**in** j : pokego, **in** pk : pokemon) \rightarrow res: nat

res \leftarrow Obtener(j .pokemon, pk);

$O(|P|)$

Complejidad: $O(|P|)$

Justificación: se debe acceder al diccionario para averiguar cuántos pokemons de esa especie.

Algoritmos auxiliares

CrearInfoPos () \rightarrow res: infoPos

Pre: true

Post: la posición nueva no contiene pokémon ni jugadores

res.hayPokemon? \leftarrow false;

$O(1)$

res.contadorCaptura \leftarrow 0;

$O(1)$

res.jugEsperandoCaptura \leftarrow Vacio();

$O(1)$

res.jugsEnPos \leftarrow Vacio();

$O(1)$

CrearInfoJugador () → res: puntero(infoJugador)

Pre: true

Post: el jugador nuevo no tiene sanciones ni pokémons y no está conectado

infoJugador : nuevo;	$O(1)$
nuevo.sanciones ← 0;	$O(1)$
nuevo.conectado? ← false;	$O(1)$
nuevo.cantPokemons ← 0;	$O(1)$
nuevo.pokemonsCapturados ← CrearDiccionario();	$O(1)$
res ← &nuevo;	$O(1)$

Complejidad: $O(1)$

Justificación: solo se inicializan las variables.

AgregarACola (in/out j : pokego, in e : jugador)

Pre: $e \in \text{jugadores}(j) \wedge_L \text{conectado?}(e, j)$

Post: agrega al jugador a la lista de espera del pokémon mas cercano a él (si hay uno en rango de captura)

coor: c ← (*j.jugadores[e]).posicion;	$O(1)$
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, 2);	$O(1)$
itBi(coor) : itCoor ← CrearIt(coorEnRango);	$O(1)$
while HaySiguiente(itCoor) do	$O(\log(EC))$
coor: siguiente ← Siguiente(itCoor);	$O(1)$
if HayPokemonEnPos(j, siguiente).hayPokemon then	
Definir(j.grillaPos[Latitud(siguiente)][Longitud(siguiente)].jugEsperandoCaptura,	
(*j.jugadores[e]).cantPokemons);	$O(\log(EC))$
end if	
Avanzar(itCoor);	$O(1)$
end while	

Complejidad: $O(\log(EC))$

Justificación: el algoritmo recorre una cantidad menor a una constante de coordenadas alrededor del jugador, de encontrar un Pokemon, agrega al jugador al diccionarioDePrioridad que tiene insercion en $O(\log(EC))$, ademas por la especificacion sabemos que un jugador puede estar capturando un solo pokemon, por lo que definir el jugador se hace una vez.

RemoverDeCola (in/out j : pokego, in e : jugador)

Pre: $e \in \text{jugadores}(j) \wedge_L \text{conectado?}(e, j)$

Post: remueve al jugador de la lista de espera del pokémon mas cercano a él (si hay uno en rango de captura)

coor: c ← (*j.jugadores[e]).posicion;	$O(1)$
conj(coor) : coorEnRango ← PosicionesEnRango(j, c, 2);	$O(1)$
itBi(coor) : itCoor ← CrearIt(coorEnRango);	$O(1)$
while HaySiguiente(itCoor) do	$O(\log(EC))$
coor: siguiente ← Siguiente(itCoor);	$O(1)$
if HayPokemonEnPos(j, siguiente).hayPokemon then	
Borrar(j.grillaPos[Latitud(siguiente)][Longitud(siguiente)].jugEsperandoCaptura, e);	$O(\log(EC))$
end if	
Avanzar(itCoor);	$O(1)$
end while	

Complejidad: $O(\log(EC))$

Justificación: debe iterar todas las posiciones de un cuadrado de lado predeterminado y agregar al jugador a la cola de espera de captura más cercana. Si la misma existe.

4. Módulo ConjuntoJugadores

Este conjunto aprovecha el orden absoluto de los jugadores (nats) para almacenar los datos en un arbol de búsqueda autabalancado (AVL). Esto permite agregar, buscar y remover cualquier elemento en tiempo logarítmico.

Usaremos $\#j$ para denotar la cantidad de entradas del diccionario.

Interfaz

se explica con: CONJUNTO(JUGADOR), ITERADOR UNIDIRECCIONAL(JUGADOR).

géneros: conjJugs, itConjJugs.

Operaciones básicas de ConjuntoJugadores

VACIO() $\rightarrow res : \text{conjJugs}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \emptyset\}$

Complejidad: $O(1)$

Descripción: Crea el conjunto vacío

AGREGAR(in/out $c : \text{conjJugs}$, in $j : \text{jugador}$)

Pre $\equiv \{c =_{\text{obs}} c_0\}$

Post $\equiv \{c =_{\text{obs}} \text{Ag}(j, c_0)\}$

Complejidad: $O(\log(\#j))$

Descripción: Agrega al conjunto j

Aliasing: se almacenan copias de j.

PERTENECE?(in $c : \text{conjJugs}$, in $j : \text{jugador}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} j \in c\}$

Complejidad: $O(\log(\#j))$

Descripción: devuelve true si el jugador esta en el conjunto.

BORRAR(in/out $c : \text{conjJugs}$, in $j : \text{jugador}$)

Pre $\equiv \{c =_{\text{obs}} c_0\}$

Post $\equiv \{c =_{\text{obs}} \text{borrar}(j, c_0)\}$

Complejidad: $O(\log(\#j))$

Descripción: borra un jugador del conjunto.

Operaciones básicas del iterador

El iterador que presentamos no permite modificar el conjunto recorrido.

CREARIT(in $c : \text{conjJugs}$) $\rightarrow res : \text{itConjJugs}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{true}\}$

Complejidad: $O(1)$

Descripción: crea un iterador unidireccional de los jugadores validos o expulsados.

Aliasing: el iterador se invalida si y solo si se elimina el siguiente elemento del iterador. Además, siguientes(res) podría cambiar completamente ante cualquier operación que modifique el conjunto.

HAYMAS(in $it : \text{itConjJugs}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{hayMas?}(it)\}$

Complejidad: $O(1)$

Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

ACTUAL(in $it : \text{itConjJugs}$) $\rightarrow res : \text{jugador}$

Pre $\equiv \{\text{HayMas?}(it)\}$

Post $\equiv \{res =_{\text{obs}} \text{Actual}(it)\}$

Complejidad: $O(1)$

2 se podría generalizar
facilmente

6. Módulo DiccionarioPrioridad

Este diccionario utiliza dos criterios de ordenamiento: por la clave y por el significado. Esto permite definir, obtener y remover cualquier elemento al igual que buscar el mínimo elemento en tiempo logarítmico.

En el ordenamiento por significado, para los casos en los que hay dos significados iguales, se utiliza nuevamente la clave.

También se almacenan los valores del mínimo elemento para retornarlos en tiempo constante.

Usaremos $\#j$ para denotar la cantidad de entradas del diccionario.

Interfaz

se explica con: DICCIONARIO(JUGADOR, NAT).

géneros: prior.

Operaciones básicas de DiccionarioPrioridad

VACIO() $\rightarrow res : \text{prior}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{vacío}\}$

Complejidad: $O(1)$

Descripción: Crea el diccionario vacío

DEFINIR(in/out $p : \text{prior}$, in $j : \text{jugador}$, in $c : \text{nat}$)

Pre $\equiv \{p =_{\text{obs}} p_0\}$

Post $\equiv \{p =_{\text{obs}} \text{definir}(j, c, p_0)\}$ → usar otra letra

Complejidad: $O(\log(\#j))$

Descripción: define la cantidad de pokemons de un jugador en el diccionario.

Aliasing: se almacenan copias de j y c .

DEFINIDO?(in $p : \text{prior}$, in $j : \text{jugador}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{def?}(j, p)\}$

Complejidad: $O(\log(\#j))$

Descripción: devuelve true si el jugador esta definido en el diccionario.

EsVACIO?(in $p : \text{prior}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{esVacio?}(p)\}$

Complejidad: $O(1)$

Descripción: devuelve true si el diccionario está vacío

BORRAR(in/out $p : \text{prior}$, in $j : \text{jugador}$)

Pre $\equiv \{p =_{\text{obs}} p_0 \wedge \text{def?}(j, d_0)\}$

Post $\equiv \{p =_{\text{obs}} \text{borrar}(j, p_0)\}$

Complejidad: $O(\log(\#j))$

Descripción: borra un jugador del diccionario.

CLAVES(in $p : \text{prior}$) $\rightarrow res : \text{conj(string)}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{alias}(res = \text{claves}(p))\}$

Complejidad: $O(\#j)$

Descripción: devuelve el conjunto de los jugadores del diccionario.

Aliasing: res no es modificable.

MENOR(in/out $p : \text{prior}$) $\rightarrow res : \text{jugador}$

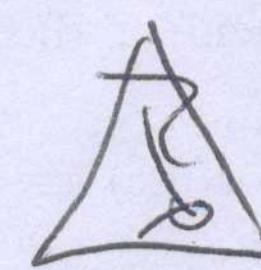
Pre $\equiv \{\neg \emptyset?(\text{claves}(p))\}$

Post $\equiv \{res =_{\text{obs}} \text{menor}(p, \text{claves}(p))\}$

Complejidad: $O(\log(\#j))$



Esto debería ser una
cola de prioridad y respetar
la interfaz de cola



Esto no
está bien



usar tuplas
desde afuera
del módulo

Descripción: devuelve el jugador con menos pokemons en el diccionario

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Diccionario(Jugador, Nat) Extendido

extiende DICIONARIO(JUGADOR, NAT)

otras operaciones (exportadas)

menor : dicc(nat × nat) d × conj(nat) c → nat
esVacio? : dicc(nat × nat) → bool

{ $\neg \emptyset(c) \wedge c \subseteq \text{claves}(d)$ }

axiomas

menor(d,c) ≡ if $\emptyset(\text{dameUno}(c))$ then
 sinUno(c)
 else
 if obtener(dameUno(c), d) < menor(d, sinUno(c)) then
 dameUno(c)
 else
 menor(d, sinUno(c))
 fi
esVacio?(d) ≡ $\emptyset(\text{claves}(d))$

Fin TAD

Representación

Representación de prior

prior se representa con dicPri

donde dicPri es tupla(raizJugadores: puntero(Nodo), raizCantidad: puntero(Nodo), menorID: jugador, menor: nat)

donde Nodo es tupla(hijoIzq: puntero(Nodo), hijoDer: puntero(Nodo), id: jugador, cantPokemons: nat, alto: nat)

Invariante de Representación

1. No hay jugadores repetidos
2. Un nodo cualquiera del arbol de jugadores su hijo izquierdo tiene menor id y su hijo derecho tiene mayor id
3. Un nodo cualquiera del arbol de cantidades su hijo izquierdo tiene menor o igual cantidad y su hijo derecho tiene mayor o igual cantidad
4. Un nodo cualquiera del arbol de cantidades su hijo izquierdo si tiene la misma cantidad tiene menor id y lo mismo para su hijo derecho pero mayor id
5. La altura de un Nodo es la altura del hijo con mayor altura mas 1
6. El factor de balanceo es ≤ 1 (donde el factor de balanceo es el modulo de la diferencia de las alturas de los hijos)
7. Los arboles tienen los mismos elementos
8. El menor es realmente el menor

NO! El modulo
no debería correr
sobre numeros
jugador

Rep : dicPri → bool

Rep(e) ≡ true ⇔ sinRepetidos(e.raizJugadores, \emptyset) \wedge sinRepetidos(e.raizCantidad, \emptyset) \wedge
• ABB?(e.raizJugadores) \wedge altoValido?(e.raizJugadores) \wedge
ABBespecial?(e.raizCantidad) \wedge altoValido?(e.raizCantidad) \wedge
 $(\forall j: \text{jugador}) (\text{definido?}(j, e.raizJugadores) = \text{definido?}(j, e.raizCantidad) \wedge_L$
 $\text{definido?}(j, e.raizJugadores) \Rightarrow_L \text{obtener}(j, e.raizJugadores) = \text{obtener}(j, e.raizCantidad)) \wedge$
 $(e.raizCantidad \neq \text{NULL}) \Rightarrow_L (e.menorID = \text{menor}(e.raizCantidad).id \wedge e.menor = \text{menor}(e.raizCantidad).cantPokemons)$

Función de Abstracción

$\text{Abs} : \text{dicPri } e \rightarrow \text{prior}$ $\{\text{Rep}(e)\}$
 $\text{Abs}(e) =_{\text{obs}} d : \text{prior} \mid (\forall j : \text{jugador}) (\text{def?}(j, d) = \text{definido?}(j, e.\text{raizJugadores}) \wedge_L$
 $(\text{def?}(j, d) \Rightarrow_L \text{obtener}(j, d) = \text{obtener}(j, e.\text{raizJugadores}))$

Operaciones auxiliares

$\text{sinRepetidos} : \text{puntero}(\text{nodo}) \times \text{conj}(\text{nat}) \rightarrow \text{bool}$	
$\text{ABB?} : \text{puntero}(\text{Nodo}) \text{ nodo} \rightarrow \text{bool}$	
$\text{menor?} : \text{puntero}(\text{Nodo}) \text{ padre} \times \text{puntero}(\text{Nodo}) \text{ hijo} \rightarrow \text{bool}$	$\{\text{padre} \neq \text{NULL}\}$
$\text{ABBespecial?} : \text{puntero}(\text{Nodo}) \text{ nodo} \rightarrow \text{bool}$	
$\text{menorEspecial?} : \text{puntero}(\text{Nodo}) \text{ padre} \times \text{puntero}(\text{Nodo}) \text{ hijo} \rightarrow \text{bool}$	$\{\text{padre} \neq \text{NULL}\}$
$\text{altoValido?} : \text{puntero}(\text{nodo}) \rightarrow \text{bool}$	
$\text{mayorAltura} : \text{nat} \times \text{nat} \rightarrow \text{nat}$	
$\text{factorDeBalanceo} : \text{nat} \times \text{nat} \rightarrow \text{nat}$	
$\text{cantidadHijos} : \text{puntero}(\text{Nodo}) \rightarrow \text{nat}$	
$\text{definido?} : \text{jugador} \times \text{puntero}(\text{nodo}) \rightarrow \text{bool}$	
$\text{obtener} : \text{jugador } j \times \text{puntero}(\text{nodo}) \text{ p} \rightarrow \text{nat}$	$\{\text{definido?}(j, p)\}$
$\text{menor} : \text{puntero}(\text{nodo}) \text{ p} \rightarrow \text{puntero}(\text{nodo})$	$\{p \neq \text{NULL}\}$

$\text{sinRepetidos}(\text{padre}, \text{ids}) \equiv \text{padre} = \text{NULL} \vee_L (\text{padre.id} \notin \text{ids} \wedge$
 $\text{sinRepetidos}(\text{padre.hijoIzq}, \text{Ag}(\text{padre.id}, c)) \wedge$
 $\text{sinRepetidos}(\text{padre.hijoDer}, \text{Ag}(\text{padre.id}, c)))$

$\text{ABB?}(\text{nodo}) \equiv (\text{nodo} = \text{NULL}) \vee_L$
 $(\text{menor?}(\text{nodo}, (*\text{nodo}).\text{hijoDer}) \wedge \neg \text{menor?}(\text{nodo}, (*\text{nodo}).\text{hijoIzq}) \wedge$
 $\text{ABB?}((*\text{nodo}).\text{hijoIzq}) \wedge \text{ABB?}((*\text{nodo}).\text{hijoDer}))$

$\text{ABBespecial?}(\text{nodo}) \equiv (\text{nodo} = \text{NULL}) \vee_L$
 $(\text{menorEspecial?}(\text{nodo}, (*\text{nodo}).\text{hijoDer}) \wedge \neg \text{menorEspecial?}(\text{nodo}, (*\text{nodo}).\text{hijoIzq}) \wedge$
 $\text{ABBespecial?}((*\text{nodo}).\text{hijoIzq}) \wedge \text{ABBespecial?}((*\text{nodo}).\text{hijoDer}))$

$\text{menor?}(\text{padre}, \text{hijo}) \equiv (\text{hijo} = \text{NULL}) \vee_L \text{padre.id} < \text{hijo.id})$

$\text{menorEspecial?}(\text{padre}, \text{hijo}) \equiv (\text{hijo} = \text{NULL}) \vee_L \text{padre.cantPokemons} < \text{hijo.cantPokemons} \vee$
 $(\text{padre.cantPokemons} = \text{hijo.cantPokemons} \wedge \text{padre.id} < \text{hijo.id})$

$\text{altoValido?}(\text{nodo}) \equiv \text{nodo} = \text{NULL} \vee_L$
 $((*\text{nodo}).\text{alto} = \text{mayorAltura}((*\text{nodo}).\text{hijoIzq}, (*\text{nodo}).\text{hijoDer}) + 1 \wedge$
 $\text{factorDeBalanceo}(\text{cantidadHijos}((*\text{nodo}).\text{hijoIzq}), \text{cantidadHijos}((*\text{nodo}).\text{hijoDer})) \leq 1)$

$\text{mayorAltura}(\text{izq}, \text{der}) \equiv \text{if } \text{cantidadHijos}(\text{izq}) < \text{cantidadHijos}(\text{der}) \text{ then}$
 $\quad \text{cantidadHijos}(\text{izq})$
 else
 $\quad \text{cantidadHijos}(\text{der})$
 fi

$\text{cantidadHijos}(\text{nodo}) \equiv \text{if } \text{nodo} = \text{NULL} \text{ then}$
 $\quad 0$
 else
 $\quad \beta((*\text{nodo}).\text{hijoIzq} \neq \text{NULL}) + \text{cantidadHijos}((*\text{nodo}).\text{hijoIzq}) +$
 $\quad \beta((*\text{nodo}).\text{hijoDer} \neq \text{NULL}) + \text{cantidadHijos}((*\text{nodo}).\text{hijoDer})$
 fi

$\text{factorDeBalanceo}(\text{izq}, \text{der}) \equiv \text{if } \text{izq} < \text{der} \text{ then } \text{der} - \text{izq} \text{ else } \text{izq} - \text{der} \text{ fi}$

$\text{definido?}(j, p) \equiv p \neq \text{NULL} \wedge_L (p.\text{id} = j \vee \text{definido?}(j, p.\text{hijoIzq}) \vee \text{definido?}(j, p.\text{hijoDer}))$

$\text{obtener}(j, p) \equiv \text{if } p.\text{id} = j \text{ then}$
 $\quad p.\text{cantPokemons}$
 else
 $\quad \text{if } \text{definido?}(j, p.\text{hijoIzq}) \text{ then } \text{obtener}(j, p.\text{hijoIzq}) \text{ else } \text{obtener}(j, p.\text{hijoDer}) \text{ fi}$
 fi

7. Consideraciones

- Se asume lógica de cortocircuito para todos los algoritmos.
- Algunas complejidades y sus justificaciones son compartidas (ya que la función es prácticamente idéntica, solo llama a una o dos auxiliares o solo realiza alguna otra acción en $\Theta(1)$), así que en algunos casos se omiten y se asume que se usa la próxima complejidad y justificación disponible *? a veces es poco claro*
- La aridad de **EntrenadoresPosibles** se modifica ya que tomaba un conjunto de jugadores por parámetro. Se asume que el funcionamiento es igual a llamar a esa función con todos los jugadores válidos.
- Para ser consistente con el cambio a iteradores en la función **Jugadores**, también se devuelve un iterador en **Expulsados**. *ok*

la idea era que
trabajar sobre ese
conjunto que recibir
y no el que está
en la estructura