TP 1: Diseño

Juego de Palabras

28 de Octubre de 2022

Algoritmos y Estructuras de Datos II

Integrante	LU	Correo electrónico
Quintiero, Manuel	71/20	manu_quintiero@hotmail.com
Sarco, Sebastian	1708/21	sebastiansarco20@gmail.com
Bruzzo,Bruno	1377/21	bruzzo.bruno@gmail.com
Soria,Stephanie	1634/21	stepsoria@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

MODULO Juego[1]

Interfaz

Parámetros Formales

SE EXPLICA CON: juego

géneros: juego

Operaciones básicas de juego:

VARIANTE (**in** j: juego) -> res : variante
Pre ≡ {True}
Post ≡ {$\hat{res} =_{obs}$ variante($\hat{j}$)}
Complejidad: O(1)
Descripción: Devuelve la variante del juego.
Aliasing: Juego se recibe por referencia no modificable.

TURNO (**in** j: juego) -> res: nat
Pre ≡ {True}
Post ≡ {$\hat{res} =_{obs}$ turno($\hat{j}$)}
Complejidad: O(1)
Descripción: Devuelve a qué jugador le toca jugar.
Aliasing: Juego se recibe por referencia no modificable.

NUEVOJUEGO(**in** k: nat, **in** v: variante, **in** r: cola(letra) -> j : juego
Pre ≡ {tamaño($\hat{r}$) ≥ tamañoTablero($\hat{v}$)^2 + k ∗ #fichas($\hat{v}$)}
Post ≡ {$\hat{j} =_{obs}$ nuevoJuego($\hat{k}$,$\hat{v}$,$\hat{r}$)}
Complejidad: O(N^2 + |Σ|K +FK)
Descripción: Genera un nuevo juego de variante v, cantidad de jugadores k y repositorio de fichas r.
Aliasing: El variante se pasa por referencia no modificable y el repositorio por referencia modificable.

UBICAR (**in/out** j:juego, **in** o:ocurrencia[2])
Pre ≡ { j $=_{obs} j_0 \wedge$ jugadaValida?($\hat{j}$,$\hat{o}$)}
Post ≡ {$\hat{j} =_{obs}$ ubicar ($\hat{j_0}$,$\hat{o}$)}
Complejidad: O(m), donde m es el número de fichas que se ubican, es decir, long(o).
Descripción: Ubica la ocurrencia dada en el tablero.
Aliasing: Juego se recibe por referencia modificable, y la ocurrencia por referencia no modificable.

JUGADAVALIDA? (**in** j: juego, **in** o: ocurrencia) -> res: bool
Pre ≡ {True}
Post ≡ {$\hat{res}$ =$_{obs}$ jugadaValida?($\hat{j}$,$\hat{o}$)}
Complejidad: O($Lmax^2$)
Descripción: Chequea si luego de poner las fichas de la ocurrencia “o”, todas las palabras del tablero son válidas.
Aliasing: Juego se recibe por referencia no modificable, y la ocurrencia por referencia no modificable.

CONTENIDOCASILLA (**in** j: juego, **in** i: nat, **in** j: nat) → res: letra
Pre ≡ { enTablero?($j.\hat{tablero}$,$\hat{i}$,$\hat{j}$) }
Post ≡ { $\hat{res}$ =$_{obs}$ If ¬hayLetra?($j.\hat{tablero}$,$\hat{i}$,$\hat{j}$) then LETRAVACIA[4] else letra($j.\hat{tablero}$,$\hat{i}$,$\hat{j}$) }
Complejidad: O(1)
Descripción: Devuelve el contenido del tablero del juego en la coordenada (i,j) que debe ser una posicion válida. Sí no hay letra devuelve la letra vacía
Aliasing: Juego se recibe por referencia no modificable.

PUNTAJE(**in** j: juego, **in** id: nat) → res : nat
Pre ≡ {0 ≤ id < j.#jugadores}
Post ≡ {$\hat{res}$ = $_{obs}$ puntaje($\hat{j}$, $\hat{id}$)}
Complejidad: O(1 + m ∗ LMax)
Descripción: Actualiza el puntaje del jugador id desde la última vez que se invocó esta operación y devuelve su puntaje actualizado.
Aliasing: Juego entra como referencia modificable, donde UltimasFichasxJugador y puntaje[id] actualiza el valor de sus índices.

CUANTASFICHASTIENE?(**in** j: juego, **in** jug: nat, **in** x: letra) → res: nat
Pre ≡ {True}
Post ≡ {$\hat{res}$ = $_{obs}$ #($\hat{x}$, $fichas\hat{}(jug, j)$) }
Complejidad: O(1)
Descripción: Devuelve cuantas letras “x” tiene el jugador “jug” en su mano.
Aliasing: Juego se recibe por referencia no modificable.

REPOSITORIO(**in** j: juego) → res: cola<letra>
Pre ≡ {True}
Post ≡ {$\hat{res}$ =$_{obs}$ $repositorio(\hat{j})$}
Complejidad: O(1)
Descripción: Devuelve el repositorio del juego.
Aliasing: Juego se recibe por referencia no modificable.

ULTIMAREPOSICION(**in** j: juego, **in**: id:nat) → res: vector<letra>
Pre ≡ {0 ≤ id < j.#jugadores}
Post ≡ {(∀ i : nat)(0 ≤ i < $longitud(res)$ →$_L$ $\hat{res}[i]$ ∈ fichas($\hat{j}$, $\hat{id}$)}
Complejidad: O(1)
Descripción: Devuelve la última reposición realizada a ese jugador.
Aliasing: Juego se recibe por referencia no modificable.

Representación

Estructura

Juego se representa con jgo
donde jgo es tupla< tablero: tablero,
variante: variante,
turno: nat,
fichasEnMano: vector<vector<nat>>,
puntaje: vector<nat>,
repositorio: cola(letra),
ultimasFichasxJugador[3]: vector<vector<vector<ficha>>,
#Jugadores: nat,
ronda: nat,
ultimaReposicionxJugador: vector<vector<letra>>
>

donde ficha es tupla<letra, ronda, i, j>
ronda, i, j es nat

Invariante de Representación

Rep: $\hat{jgo}$ -> bool
(∀ j : jgo)
Rep(j) =
tamañoTablero(j.variante) = tamaño(j.tablero) ∧
OcurrenciasenTablero?Validas ∧
0 ≤ j.turno < j.#Jugadores ∧
LongitudEsCorrectaFichasEnMano ∧
#FichasEnMano ∧
long(j.puntaje) = j.#jugadores ∧
long(j.UltimasFichasXJugador) = j.#Jugadores ∧
UltimasFichasEsValida ∧
ultimaReposicionxJugadorValida

Donde:

- **OcurrenciasenTablero?Validas** = Para toda ocurrencia presente en el tablero, la palabra que forma debe pertenecer al conjunto de palabras válidas del variante.

- **LongitudEsCorrectaFichasEnMano** = long(j.fichasEnMano) = j.#Jugadores ∧ (∀ i:nat) (i < j.#Jugadores $\rightarrow_L$ long(j.fichasEnMano[i]) =|Σ|)

- **#FichasEnMano** = (∀ k :nat)(k < j.#jugadores)($\sum_{i=0}^{longitud(X)} X[i]$ = #fichas(j.variante)

  donde X = j.fichasEnMano[k]

- **UltimasFichasEsValida** = La ronda de cada ficha debe ser menor o igual que la ronda actual(j.ronda). Luego, por cada ficha en cada vector de UltimasFichasXJugador, cada ficha debe seguir estando en el tablero actual, en su posición correspondiente.

- **ultimaReposicionxJugadorValida** = long(j.ultimaReposicionxJugador) = j.#Jugadores ∧ (∀ i :nat) (i < j.#Jugadores $\rightarrow_L$ long(j.ultimaReposicionxJugador[i]) ≤ LMax)

Función de abstracción

Abs : $\hat{jgo}$ j → juego {Rep(j)}

Abs(j) = g : juego /

j.tablero = tablero(g) ∧
j.variante = variante(g) ∧
j.turno = turno(g) ∧
j.#Jugadores = #jugadores(g) ∧
CantidadDeFichasEnManoCorrectas ∧
PuntajesPorJugadorCorrectos ∧
j.repositorio = repositorio(g)

Donde:

- **CantidadDeFichasEnManoCorrectas** = (∀ i:nat) (i < #Jugadores(g) $\rightarrow_L$ (Vj: nat)(j < |Σ| → j.fichasEnMano[i][j] = #(ord^{-1}(j),fichas(g,i))

- **PuntajesPorJugadorCorrectos** = Para una instancia del tad, el puntaje de un jugador “x” es igual al elemento de puntaje correspondiente dentro de la estructura (j.puntaje[x]) sumado al puntaje que recibiria el jugador por las palabras formadas por las fichas pertenecientes al vector j.ultimasFichasXJugador[x], dado el estado del tablero en el que se pusieron dichas fichas.

Algoritmos

```
iVariante(in j: jgo) → res : vrt
1    res← j.variante
```

Complejidad: O(1)
Justificación: La variante se encuentra presente en la estructura de juego, por lo tanto es hacer una referencia.

```
iTurno(in j: jgo) → res: nat
1    res ← j.turno
```

Complejidad: O(1)
Justificación: El turno se encuentra presente en la estructura de juego, por lo tanto es hacer una referencia.

```
iNuevoJuego(in k: nat, in v: vrt, in r: cola(letra) → j: jgo
1    tablero t ← nuevoTablero(tamañoTablero(v))
2    vector<vector<nat>> fichas ← IniciarVectorDimensionado(k,
  IniciarVectorDimensionado(|Σ|,0))
     vector<vector<letra>> ultimaRep ← IniciarVectorDimensionado(k, vacía())
3    vector<nat> puntaje ← IniciarVectorDimensionado(k,0))
4    i ← 0
5    while (i < k)
6        j ← 0
7        while (j < #fichas(v))
8            fichas[i][ord(próximo(r)] = fichas[i][ord(próximo(r)] + 1
9            desencolar(r)
10        endwhile
11    endwhile      // A cada jug se le reparte tantas f como diga el var.
12    vector<vector<ficha>> h ← IniciarVectorDimensionado(k, vector.vacia())
13    nat ronda ← 0
14    j ← tupla(t, v, 0, fichas, puntaje, r, h, k, ronda, ultimaRep)
```

Complejidad: O(N^2 +|Σ|K +FK)
Justificación: La creación del tablero tiene complejidad O(N^2). Inicializar el vector que registrará la cantidad de fichas por jugador tiene complejidad O(k*|Σ|). Repartir las fichas tiene complejidad O(KF). El resto de las complejidades son acotadas.

```
iubicar (in/out j: jgo, in o: ocurrencia)
1  i ← 0
2   j.ultimaReposicionxJugador[j.turno] ← vacía()
```

```
3  while (i < longitud(o))
4     // Agrega la ficha al tablero
5     ponerLetra(j.tablero, o[i][0], o[i][1], o[i][2], j.ronda)
6
7     agregarAtras(j.ultimasFichasxJugador[j.turno], tupla(o[i][2], o[i][0],
8     o[i][1], j.ronda))
9     // Le sacamos la ficha al jugador.
10    j.fichasEnMano[j.turno][ord(o[i][2])] = j.fichasEnMano[j.turno][ord(o[i][2])]-1
11     // Le repone 1 ficha al jugador
12    AgregarAtras(j.ultimaReposicionxJugador[j.turno],proximo(j.repositorio))
13     j.fichasEnMano[j.turno][ord(proximo(j.repositorio))]++
14     desencolar(j.repositorio)
15     i = i + 1
16  endwhile
17  j.ronda++
18  j.turno = j.turno + 1 % j.#jugadores
```

Complejidad = O(m), donde m es la cantidad de fichas que ubica el jugador en esa jugada.
Justificación = O(1) el costo de inicializar variables y las asignaciones. El ciclo se repite m veces, y todas sus operaciones tienen costo O(1). Luego O(m) es el costo total del ciclo. El costo total del algoritmo es: O(1) + O(m) * O(1) + O(1) = O(m)

---

contenidoCasilla (in j: jgo, in i: nat, in j: nat) → res: letra

```
1    res ← j.tablero[i][j]_0[5]
```

Complejidad: O(1)
Justificación: Indexar la fila tiene complejidad O(1), indexar la celda en la misma tiene complejidad O(1).

---

ijugadaValida? (in j:juego, in o:ocurrencia) -> res: bool

```
1  res ← true
2  if (LMax < longitud(o)) //Evitar ocurrencias mayores a LMax
3    res ← false
4  endif
5  vector<letra> palabra = vacia()
6  i ← 0
7  while (i < longitud(o) ∧ res) // Para cada elemento de la ocurrencia
8    f ← o[i][0]
9    c ← o[i][1]
10    while ((hayLetra?(j.tablero, f, c - 1) && enTablero?(j.tablero, f, c - 1) &&
  res) // Obtener primer letra de palabra horizontal.
11      c- -
```

```
12    endwhile
13    while((hayLetra?(j.tablero,f,c) || c=o[i][1]) && enTablero?(j.tablero,f,c) &&
  res) // Avanzamos hasta la última para conseguir la palabra horizontal.
14       if (c = o[i][1])
15         agregarAtras(palabra, o[i][2])
16       else
17         agregarAtras(palabra, Letra(j.tablero, f, c)_0)
18       endif
19       c++
20    endwhile
21    if !(palabraLegitima?(j.variante, palabra)) //Chequear validez de palabra.
22      res ← false
23    endif
24    palabra = vacia()
25    c ← o[i][1]
26    while ((hayLetra?(j.tablero, f - 1, c) && enTablero?(j.tablero, f -1, c) &&
  res) // Analogamente a buscar la palabra horizontal pero vertical
27       f--
28    endwhile
29    while((hayLetra?(j.tablero, f, c) || f=o[i][0] ) && enTablero?(j.tablero, f,
  c) && res)
30      if (f = o[i][0])
31        agregarAtras(palabra, o[i][2])
32      else
33        agregarAtras(palabra, Letra(j.tablero, f, c)_0)
34     endif
35     f++
36     endwhile
37     if !(palabraLegitima?(j.variante, palabra))
38       res ← false
39     endif
40     palabra = vacia()
41     i++
42  endwhile
```

Complejidad: O(LMax^2)
Justificación: El algoritmo tiene 4 ciclos O(LMax) y 2 chequeos de la palabra que cuestan O(LMax) dentro de otro ciclo que también es O(LMax), ademas de varias operaciones O(1). Usando álgebra de órdenes:

kO(1) + O(LMax) * 6O(LMax) = O(LMax^2).

---

iCuantasFichasTiene?(in j: juego, in jug: nat, in x: letra) → res: nat

```
1    res ← j.fichasEnMano[jug][ord(x)]
```

Complejidad: O(1)
Justificación: La estructura de juego almacena la cantidad de fichas que posee un jugador en un vector. Luego el acceso a una posición de este vector es O(1)

---

iPuntaje(in j:jgo, in id:nat) → res: nat

```
1    for(i = 0; i  < longitud(j.ultimasFichasXJug[id]); i++)
2    //iteras sobre las rondas/jugadas
3        vector<ficha> jugada_actual = j.ultimasFichasXJug[id][i]
4        bool ocurrencia_comun_vertical = (longitud(jugada_actual) > 1 &&
  jugada_actual[0]_3 == jugada_actual[1]_3)
5        bool ocurrencia_comun_horizontal = !ocurrencia_comun_vertical
6
7        ronda = jugada_actual[0]_1
8        f = jugada_actual[0]_2 // fila
9        c = jugada_actual[0]_3 // columna
10        j.puntaje[id] += puntajeLetra(j.variante, jugada_actual[0]_0)
11//se suma el puntaje de la primera letra. Esta letra se usa como “pivot” para
  sumar el resto del puntaje de la ocurrencia comun.
12
13        if(ocurrencia_comun_horizontal){
14          while (hayLetra?(j.tablero, f, c - 1) && enTablero?(j.tablero, f, c-1)
   && Letra(j.tablero, f, c - 1)_1 ≤ ronda)
15            j.puntaje[id] += puntajeLetra(j.variante, Letra(j,tablero, f, c-1)_0)
16            c−
17            endwhile
18
19          c = jugada_actual[0]_3 //reinicias la columna
20          while (hayLetra?(j.tablero, f, c + 1) && enTablero?(j.tablero, f, c+1)
21                 && Letra(j.tablero, f, c + 1)_1 ≤ ronda)
22            j.puntaje[id] += puntajeLetra(j.variante, Letra(j,tablero, f, c+1)_0)
23            c++
24          endwhile
25 /* ya sumamos los puntos de la ocurrencia común, incluyendo los de las letras de
26 las fichas que puso el jugador
27 falta sumar los puntos en vertical de cada ficha. */
28
29          j = 0 //j-esima ficha de la jugada (jugada_actual[j])
30          while(j < longitud(jugada_actual)){
31          f = jugada_actual[j]_2 // fila
32          c = jugada_actual[j]_3 // columna
33
34          while (hayLetra?(j.tablero, f-1,c) && enTablero?(j.tablero,f-1,c)
   && Letra(j.tablero,f-1,c)_1 ≤ ronda))
35              j.puntaje[id] +=puntajeLetra(j.variante,Letra(j,tablero,f-1,c)_0)
36              f--
```

```
37          endwhile
38
39          f = jugada_actual[j]_2 //se reinicia la fila
40           while (hayLetra?(j.tablero, f + 1, c) && enTablero?(j.tablero, f+1, c)
                        && Letra(j.tablero, f + 1, c)_1 ≤ ronda)
41              j.puntaje[id] +=puntajeLetra(j.variante,Letra(j,tablero, f+1, c)_0)
42              f++
43          endwhile
44
45        else //si la ocurrencia común es vertical y no horizontal.
46          while (hayLetra?(j.tablero, f - 1, c) && enTablero?(j.tablero, f-1, c)
  && Letra(j.tablero, f - 1, c)_1 ≤ ronda)
47             j.puntaje[id] += puntajeLetra(j.variante,Letra(j,tablero, f-1, c)_0)
48             f--
49          endwhile
50
51          f = jugada_actual[0]_3 //reinicias la fila
52
53          while (hayLetra?(j.tablero, f + 1, c) && enTablero?(j.tablero, f+1, c)
  && Letra(j.tablero, f + 1, c)_1 ≤ ronda)
54              j.puntaje[id] += puntajeLetra(j.variante,Letra(j,tablero,f+1, c)_0)
55              f++
56          endwhile
57
58     /* ya sumamos los puntos de la ocurrencia común, incluyendo los de las
    letras de las fichas que puso el jugador
59     falta sumar los puntos en horizontal de cada ficha (de esto se encarga el
60   proximo while) /*
61
62          j = 0 // Esta j es para iterar por sobre las fichas de la jugada
63          while(j < longitud(jugada_actual))
64              f = jugada_actual[j]_2 // fila
65              c = jugada_actual[j]_3 // columna
66              while(hayLetra?(j.tablero, f,c+1) && enTablero?(j.tablero,f,c+1)
  && Letra(j.tablero, f, c + 1)_1 ≤ ronda)
67                  j.puntaje[id] += Letra(j.tablero, f, c + 1)_0
68                  c++
69              endwhile
70
71              c = jugada_actual[j]_2 //se reinicia la columna
72
73              while(hayLetra?(j.tablero,f,c-1) && enTablero?(j.tablero,f,c-1)
  && Letra(j.tablero,f,c-1)_1 ≤ ronda)
74                    j.puntaje[id] += Letra(j.tablero, f, c - 1)_0
75                    c--
76              endwhile
77       endif
```

```
78    ultimasFichasXJug[id] = vacia()
79    res ← j.puntaje[id]
```

Complejidad: O(1 + m * LMax)
Justificación: O(1) para definir las variables al principio del “for” principal y para vaciar el vector ultimasFichasxJugador del jugador, O(LMax) para obtener el puntaje de la ocurrencia común y O(m * LMax) para obtener el puntaje de cada palabra que formó cada ficha, en sentido contrario al sentido de la ocurrencia común. Usando álgebra de órdenes:

$$0(1) + O(LMax) + O(m * LMax) \subseteq O(1 + m*LMax)$$

---

iRepositorio((in j: juego) → res: cola<letra>

```
1      res ← j.repositorio
```

Complejidad: O(1)
Justificación: El repositorio se encuentra presente en la estructura de juego, por lo tanto acceder a él es O(1).

---

iUltimaReposicion((in j: juego, in id: nat) → res: vector<letra>

```
1      res ← j.ultimaReposicionxJugador[id]
```

Complejidad: O(1)
Justificación: ultimaReposicionxJugador se encuentra presente en la estructura de juego, por lo tanto acceder a él es O(1).

MÓDULO Variante

Interfaz

Parámetros Formales

SE EXPLICA CON: variante

géneros: variante

Operaciones básicas de variante

TAMAÑOTABLERO (**in** v: variante) -> res : nat
Pre ≡ {True}
Post ≡ {res $=_{obs}$ tamañoTablero(v)}
Complejidad: O(1)
Descripcion: Da el tamaño de una fila o columna del tablero como referencia no modificable.
Aliasing: Recibe el variante como referencia no modificable.

#FICHAS (**in** v: variante) -> res : nat
Pre ≡ {True}
Post ≡ {res $=_{obs}$ #fichas(v)}
Complejidad: O(1)
Descripcion: Devuelve la cantidad de fichas que tendrá cada jugador en la mano de la variante del juego como referencia no modificable.
Aliasing: Recibe el variante como referencia no modificable.

PUNTAJELETRA (**in** v: variante, **in** l: letra) -> res : nat
Pre ≡ {True}
Post ≡ {res $=_{obs}$ puntajeLetra(v, l) }
Complejidad: O(1)
Descripcion: Devuelve el puntaje que suma poner una letra como referencia no modificable.
Aliasing: Recibe el variante como referencia no modificable.

PALABRALEGITIMA? (**in** v: variante, **in** s: vector(letra)) -> res : bool
Pre ≡ {True}
Post ≡ {res $=_{obs}$ palabraLegítima?(v, s)}
Complejidad: O(LMax)
Descripcion: Verifica que la palabra s sea una palabra válida de la variante del juego.
Aliasing: Recibe el variante y la palabra como referencia no modificable.

NUEVAVARIANTE (**in** n: nat, **in** f: nat, **in** d: dicc(letra, nat), **in** c: conj(vector(letra)) -> res : variante
Pre ≡ {n > 0 ∧ f > 0}
Post ≡ {res =$_{obs}$ nuevaVariante(n, f, d, c)}
Complejidad: O(LMax * longitud(c) + |∑|)
Descripcion: Crea una nueva variante de juego no modificable.
Aliasing: Recibe el diccionario d y el conjunto c como referencias no modificables.

Representación

Estructura

variante se representa con vrt donde vrt es
tupla<tamaño: nat,
#fichas: nat,
puntajesLetraVec: vector<nat>,
palabrasValidas: conjDigital(vector(letra)),
>

Invariante de Representación

Rep: $\hat{vrt}$ -> bool
(∀ e: $\hat{vrt}$)
Rep(e) = e.tamaño > 0 ∧ e.#fichas > 0 ∧ longitud(e.puntajesLetraVec) = |∑| ∧
(∀ i : nat)(0 ≤ i < |∑| $\rightarrow_L$ puntajesLetraVec[i] > 0) ∧
(∀ p : vector<letra>)(p ∈ e.palabrasValidas $\rightarrow_L$ p ≤ $LMax$)

Función de abstracción

Abs: $\hat{vrt}$ e → variante {Rep(e)}
Abs(e) =$_{obs}$ v: variante /
e.tamaño = tamañoTablero(v) ∧ e.#fichas = #fichas(v) ∧
(∀ i: nat)(i < |∑| $\rightarrow_L$ puntajesLetraVec[i] = puntajeLetra(v, ord^{-1}(i))) ∧
(∀ p : vector<letra>)(p ∈ e.palabraValidas ⇔ palabraLegitima?(v, p))

Algoritmos

---

iTamañoTablero(in v: vrt) → res : nat
1 res ← v.tamaño

Complejidad: O(1)
Justificación: El tamaño del tablero está almacenado en la estructura de la variante, luego acceder por referencia es O(1).

---

i#fichas(in v: vrt) -> res : nat
1 res ← v.#fichas

Complejidad: O(1)
Justificación: La cantidad de fichas que puede tener cada jugador en la mano está almacenado en la estructura de la variante, luego acceder por referencia es O(1).

---

iPuntajeLetra(in v: vrt, in l : letra) → res : nat
1 res ← v.puntajesLetraVec[ord(l)]

Complejidad: O(1)
Justificación: El puntaje de cada letra se encuentra almacenado en un vector en la estructura de la variante. Luego acceder a una posición de este vector es O(1).

---

ipalabraLegitima?(in v: vrt, in p: vector(letra)) → res : bool
1 res ← pertenece?(p, v.palabrasValidas)

Complejidad: O(LMax)
Justificación: Chequear pertenencia en un conjunto digital (trie) está acotado por la palabra legítima más larga definida por la variante, LMax.

---

iNuevaVariante(in n:nat,in f:nat,in d:dicc(letra,nat),in c:conj(vector(letra)) → res: variante

```
1    it ← crearIt(c)
2    PuntajesLetraVec ← IniciarVectorDimensionado(|∑|, 1) // O(|∑|)
3    itDic ←creatIt(d)
4    while(haySiguiente(itDic)) // O(|∑|)
5        PuntajesLetraVec[ord(siguienteClave(itDic)] = siguienteSignificado(itDic)
6// Esto se puede hacer porque PuntajesLetraVec es modificable en esta instancia.
7
8        Avanzar(it)
```

```
9    endwhile
10    conjTrie ← conjDigital(letra).vacía()
11    while(haySiguiente(it))
12        agregar(conjDigital, siguiente(it)) //conjunto hecho con tries.
13        if longitud(siguiente(it) > Lmax)
14            Lmax = longitud(siguiente(it))
15          it ← avanzar(it)
16        endif
17    endwhile
18    res ← tupla(n, f, PuntajesLetraVec, conjDigital)
```

Complejidad: O(LMax * #palabras + |∑|)
Justificación: Crear el vector de puntajes por letra tiene costo O(|∑|). Cambiar sus valores también tiene costo O(|∑|). El costo de guardar las palabras validas en un conjunto digital es O(LMax * #palabras).

MÓDULO Conjunto Digital(letra)

Interfaz

Parámetros formales:

SE EXPLICA CON: conjunto(α)
géneros: conjDigital(letra)

Operaciones básicas de conjunto digital:

VACÍA() → res: conjDigital(letra)
Pre ≡ {True}
Post ≡ {res $=_{obs}$ ∅}
Descripción: Devuelve un conjunto digital vacío.
Complejidad: O(|∑|)

INSERTAR(**in** p: vector<letra>, **in/out** t: conjDigital(letra))
Pre ≡ {t $=_{obs}$ t_0}
Post ≡ {t $=_{obs}$ Ag(p, t_0) }
Descripción: Inserta el elemento a al conjunto digital t.
Complejidad: O(LMax)

PERTENECE?(**in** p: vector<letra>, **in** t: conjDigital(letra)) → res: bool
Pre ≡ {True}
Post ≡ {res $=_{obs}$ pertenece?(a, t)}
Descripción: Devuelve si el elemento a pertenece al conjunto digital t.
Complejidad: O(LMax)

Representación

Estructura

conjDigital se representa con trie
donde trie es puntero(nodo)
dónde nodo es tupla(hijos: vector<trie>, finPalabra: bool)

Invariante de Representación

Rep: $\hat{trie}$ -> bool
(∀ t: $\hat{trie}$) Rep(t) = ¬(t == NULL) → longitud(t→hijos) == |∑|

Función de abstracción

Abs: $\hat{trie}$ t → conjunto ($\hat{palabra}$) **{ Rep(t) }**

Abs(t) $=_{obs}$ c : conjunto /

($\forall\ p : \hat{palabra}$) (pertenece?(p, t) ⇔ p ∈ c)

dónde palabra es vector<letra>

Algoritmos

```
Vacía() → res: trie
1    res <-- puntero(nodo(IniciarVectorDimensionado(|∑|, null), false)) //El
  array debe ser inicializado todo en NULL
```

Complejidad: O(|∑|)
Justificación: Inicializar el vector de longitud |∑| cuesta O(|∑|).

```
Insertar(in p: vector<letra>, in/out t: trie)
1    nodoActual ← trie //va a servir para avanzar sobre los nodos del trie.
2    for(i ← 0; i < longitud(p); i++)
3        if(nodoActual→hijos[ord(p[i])]) == NULL)
4            nodoActual→hijos[ord(p[i])] = trie.vacio()
5        end if
6        nodoActual ← nodoActual→hijos[ord(p[i])]
7     //Avanzas nodoActual, ya sabiendo que la posicion en hijos no es NULL.
8     endfor
9
10    nodoActual-->finPalabra = true; //nodoActual es la ultima letra.
```

Complejidad: O(LMax)
Justificación: En el peor caso la palabra a insertar tiene longitud Lmax, y luego el for realiza tantas iteraciones como longitud tenga esa palabra.

```
Pertenece?(in p: vector<letra>, in t: trie) --> res: bool
1    nodoActual ← t
2    for(i ← 0; i < longitud(p) && nodoActual->hijos[ord(p[i])] != NULL; i++)
3       nodoActual ← nodoActual→hijos[ord(p[i])]
4    endfor
5    res ← (i == longitud(palabra) && nodoActual→finPalabra)
```

Complejidad: O(LMax)
Justificación: El peor caso es que busquemos pertenencia de la palabra legítima con longitud máxima. En ese caso habría que iterar por esa longitud (LMax).

MÓDULO Tablero

Interfaz

Parámetros formales:

SE EXPLICA CON: tablero
géneros: tablero

Operaciones básicas de tablero:

TAMAÑO (**in** t: tablero) → res: nat
Pre ≡ {True}
Post ≡ {res $=_{obs}$ tamaño(t) }
Descripción: Devuelve el tamaño del tablero t.
Complejidad: O(1)
Aliasing: Tablero se recibe como referencia no modificable.

HAYLETRA? (**in** t: tablero, **in** i: nat, **in** j: nat) → res: bool
Pre ≡ {enTablero?(t, i, j)}
Post ≡ {res $=_{obs}$ hayLetra(t, i, j)}
Descripción: Devuelve true sí y solo sí en la posición (i,j) del tablero t hay una letra, donde (i,j) debe ser una posición válida.
Complejidad: O(1)
Aliasing: Tablero se recibe como referencia no modificable.

LETRA (**in** t: tablero, **in** i: nat, **in** j: nat) → res: tupla(letra, nat)
Pre ≡ {enTablero?(t, i , j) $\wedge_L$ hayLetra?(t, i, j)}
Post ≡ {res_0 $=_{obs}$ letra(t, i, j)}
Descripción: Devuelve la letra ubicada en la posición (i,j) de t siempre y cuando sea una posición válida.
Complejidad: O(1)
Aliasing: Tablero se recibe como referencia no modificable.

NUEVOTABLERO (**in** n: nat) → res: tab
Pre ≡ {n > 0}
Post ≡ {nuevoTablero(n) $=_{obs}$ res}
Descripción: Genera un nuevo tablero de tamaño n.
Complejidad: $O(n^2)$

PONERLETRA (**in**/**out** t: tablero, **in** i:nat, **in** j:nat, **in** l: letra, **in** r: nat)
Pre ≡ {t_0 = t ∧ ¬hayLetra?(t, i, j) $\wedge_L$ enTablero?(t, i, j)}
Post ≡ {t =$_{obs}$ ponerLetra(t_0, i , j, l)}
Descripción: Modifica el tablero t ubicando (l,r), donde r es la ronda actual en la posición (i,j) siempre y cuando sea una posición válida en ese tablero y no haya ninguna letra antes.
Complejidad: O(1)
Aliasing: Tablero se recibe como referencia modificable.

ENTABLERO?(**in** t: tab, **in** i: nat, **in** j: nat) → res: bool
Pre ≡ {True}
Post ≡ {res =$_{obs}$ enTablero?(t, i, j)}
Descripción: Devuelve true si la posición (i,j) está dentro del rango del tablero.
Complejidad: O(1)
Aliasing: Tablero se recibe como referencia no modificable.

OCURRENCIASDEPALABRAS(in t:tab) -> res: conj(ocurrencia)[6]
Pre {True}
Post {$res =_{obs}$ ocurrenciaDePalabras(t)}
Descripción: Devuelve todas las ocurrencias presentes en el tablero.
Complejidad: O(N^2)
Aliasing: Tablero se recibe como referencia no modificable.

Representación

Estructura

tablero se representa con tab dónde tab es vector<vector<tupla<letra, nat>>>

Invariante de Representación

Rep: tab → bool
(∀ t : tab)
Rep(t) = (∀ i :nat) (0≤ i< long(t)) →$_L$ long(t) = long (t[i])

Función de abstracción

Abs: t:tab → tablero {Rep(t)}
Abs(t) = x : tablero /
tamaño(x) = long(t) ∧ (∀ i, j : nat)((0 ≤ i,j < long(t) →$_L$ (hayLetra?(x,i,j) ⇔ (t[i][j]_0 != LETRAVACIA) ∧ hayLetra?(x,i,j) →$_L$(Letra(x,i,j) == t[i][j]_0))

Algoritmos

---

iTamaño(in t: tab) → res : nat
1 res ← longitud(t)

Complejidad: O(1)
Justificación: Calcular la longitud de un vector es O(1).

---

ihayLetra?(in t: tab, i : nat, j : nat) → res : bool
1 res ← t[i][j]_0 != LETRAVACIA

Complejidad: 0(1)
Justificación: El acceso a un elemento de un vector por indice es O(1)

---

iLetra(in t: tab, in i: nat, in j: nat) → res : tupla(letra, nat)
1 res ← t[i][j]

Complejidad: 0(1)
Justificación: El acceso a un elemento de un vector por indice es O(1)

---

iNuevoTablero(in n : nat) -> res : tab
1 res ←
iniciarVectorDimensionado(n,iniciarVectorDimensionado(n,tupla(LETRAVACIA,0)))

Complejidad: $O(N^2)$
Justificación: Es el costo de inicializar un vector de tamaño N, donde cada posición es a la vez otro vector de tamaño N.

---

iponerLetra(inout t: tab, i : nat, j : nat, let : letra, r: nat)
1 t[i][j] = tupla(let,r));

Complejidad: 0(1)
Justificación: El acceso a un elemento de un vector por indice es O(1). Luego, crear una tupla ya dados los elementos es O(1)

---

ienTablero?(in t : tab, i : nat, j : nat) → res : bool

```
1   res ← (0 ≤ i < longitud(t) && 0 ≤ j < longitud(t))
```

Complejidad: O(1)
Justificación: El costo de las comparaciones y la asignación es O(1)

---

iocurrenciasDePalabras(in t: tab) → res : conjuntoLineal(ocurrencias)

```
1 res ← vacio()
2 for(int i = 0; i < longitud(t): i++)
3    vector<ocurrencia> ocurrenciaHor = vacia()
4    for( int j = 0: j < longitud(t); j++)
5        if (t[i][j]_0 != LETRAVACIA)
6           AgregarAtras(ocurrenciaHor, (i, j, t[i][j]_0))
7       else
8            Agregar(res, ocurrenciaHor)
9           ocurrenciaHor = vacia()
10        end if
11    end for
12     for(int j = 0; j < longitud(t), j++)
13     vector<ocurrencia> ocurrenciaVer = vacia()
14       for(int i = 0; i < longitud(t), i++)
15           if (t[i][j]_0 != LETRAVACIA)
16               AgregarAtras(ocurrenciaVer, (i, j, t[i][j]_0))
17             else
18                 Agregar(res, ocurrenciaVer)
19                 ocurrenciaVer = { }
20             end if
21    end for
22 end for
```

MODULO Vector(α) EXTENSIÓN[7]

Interfaz

INICIARVECTORDIMENSIONADO(**in** n: nat, **in** a : α) →res : Vector(α)
Pre ≡ { True}
Post ≡ { longitud(res) = n ∧ (∀ i: nat) (i < longitud(res)) $\rightarrow_L$ res[i]=a) }
Complejidad: O(n * copy(a))
Descripcion: Inicializa un vector de n elementos con valor a.

Algoritmos

```
iIniciarVectorDimensionado(in n:nat, in a:α) →res: vector(α)
1    j ←0
2    res ←vacía()
3    while(j < n)
4        agregarAtras(res, a)      // O(1) amortizado
5        j ← j + 1
6    endwhile

Complejidad: O(n * copy(a))
Justificación: El costo será el de copiar el elemento a n veces.
```

MÓDULO SERVIDOR

Interfaz

Parámetros formales:

SE EXPLICA CON: servidor

géneros: servidor

Operaciones básicas de servidor

NUEVOSERVIDOR (**in** v: variante, **in** k: nat, **in** c: cola(letra)) -> res: servidor
Pre ≡ {long(r) ≥ tamañoTablero(v) * tamañoTablero(v) + k * #fichas(v)}
Post ≡ {res = $_{obs}$ nuevoServidor(k, v, c) }
Complejidad: O(N2 + |Σ|K + FK)
Descripción: Inicializa un servidor.
Aliasing: El variante se recibe como referencia no modificable. c se recibe como referencia modificable (es modificada por la operacion nuevoJuego.)

CONECTARCLIENTE (**in/out**: s servidor)
Pre ≡ {$s_0 =_{obs} s \wedge$ ¬empezó?(s_0)}
Post ≡ {s = $_{obs}$ conectarCliente(s_0) }
Complejidad: O(1)
Descripción: Conecta un cliente al servidor.
Aliasing: El servidor se recibe como referencia modificable.

CONSULTARNOTIFICACIONES (**in/out** s: servidor, **in** id: nat)
Pre ≡ { $s =_{obs} s_0 \wedge$ id < #conectados(s)}
Post ≡ {s = consultar(s_0, id)}
Complejidad: O(n)
Descripción: Vacía la cola de notificaciones del jugador id.
Aliasing: El servidor se recibe como referencia modificable.

RECIBIRMENSAJE (**in/out** s: servidor, **in** id: nat, **in** o: ocurrencia)
Pre ≡ { $s =_{obs} s_0 \wedge$ id < #conectados(s)}
Post ≡ {s = recibirMensaje(s_0, id, o)}
Complejidad: O($Lmax^2 + m * LMax$)
Descripción: Recibe un mensaje de un cliente.
Aliasing: El servidor se recibe como referencia modificable.

#ESPERADOS(**in** s: servidor) → res : nat
Pre ≡ {True}
Post ≡ { res $=_{obs}$ #esperados(s)}
Complejidad: O(1)
Descripción: Devuelve la cantidad de jugadores esperados del servidor.
Aliasing: El servidor se recibe como referencia no modificable.

#CONECTADOS(**in** s: servidor) → res : nat
Pre ≡ {True}
Post ≡ { res $=_{obs}$ #conectados(s)}
Complejidad: O(1)
Descripción: Devuelve la cantidad de jugadores conectados al servidor.
Aliasing: El servidor se recibe como referencia no modificable.

JUEGO(**in** s: servidor) → res : juego
Pre ≡ { empezó?(s) }
Post ≡ { res $=_{obs}$ juego(s)}
Complejidad:O(1)
Descripción: Devuelve el juego correspondiente al servidor.
Aliasing: Servidor se pasa por referencia no modificable.

Representación

Estructura

servidor se representa con srv donde srv es
    tupla<j: juego,
    CantEsperados: nat,
    jugadores: vector<nat>,
    timeStamp: nat,
    notificacionesxJugador: vector<tupla(notifs: cola(tupla(notif, nat)), índice: nat)>,
    notificacionesBroadcast: vector<tupla(notif, nat)>

Invariante de Representación

Rep: srv →bool
(∀ s: srv)
Rep(s) = longitud(s.jugadores) ≤ s.CantEsperados ∧
longitud(s.notificacionesxJugador) = s.CantEsperados ∧
(∀i :nat) (0≤ i < longitud(s.jugadores) $\Rightarrow_L$ s.jugadores[i] = i) ∧
timeStampValido ∧
indiceValido ∧
notificacionesxJugadorValidas ∧
notificacionesBroadcastValidas

donde:

timestampValido: En una misma instancia, en notificacionesxJugador y notificacionesBroadcast, los timestamp de las notificaciones de cada jugador son estrictamente menores que el timeStamp correspondiente a la instancia.

indiceValido: (∀i :nat) (0≤ i < longitud(s.notificacionesxJugador) $\Rightarrow_L$ $notificacionesxJugador_{índice}$ ≤ longitud(notificacionesBroadcast)

notificacionesxJugadorValidas: Si existen notificaciones en alguna de las colas del vector, las notificaciones pueden ser IDCliente, Reponer, Mal. No pueden existir notificaciones Empezar, TurnoDe, Ubicar, SumaPuntos.

notificacionesBroadcastValidas: Si existen notificaciones en alguna de las colas del vector, las notificaciones pueden ser Empezar, TurnoDe, Ubicar, SumaPuntos. No pueden existir notificaciones IDCliente, Reponer, Mal.

Función de Abstracción

Abs: e:srv → servidor {Rep(e)}
Abs(e) = s: servidor/
juego(s) = e.j ∧ #esperados(s) = e.CantEsperados ∧ #conectados(s) = longitud(e.jugadores)
∧ configuracion(s) = tupla(variante(s.j), repositorio(s.j)) ∧
seCorrespondenLasColasDeNotificaciones

donde:

seCorrespondenLasColasDeNotificaciones: Las notificaciones de un jugador “x” de una instancia del tad son iguales a la cola de notificaciones correspondientes al jugador “x” (es decir, la que le corresponde en el vector notificacionesxJugador) combinada con el vector de notificaciones de broadcast desde el elemento referenciado por el índice del jugador, hasta el final del vector. La secuencia combinada está ordenada de menor a mayor con respecto a el timeStamp.

Algoritmos

```
inuevoServidor (in v: variante, in k: nat, in c: cola(letra)) → res: srv
1    juego j ← nuevoJuego(k,v,c)
2    vector<nat> jugadores ← vacio()
3    vector<tupla(cola(tupla(notif, nat)), nat)> notificaciones ←
4    IniciarVectorDimensionado(k, tupla(cola.vacia(),0))
5    vector<tupla(notif, nat)> notificacionesBroadcast ← vector.vacia()
6    res ← tupla(j, k, jugadores, 0, notificaciones, notificacionesBroadcast)
```

Complejidad: $O(N^2 + |\Sigma|K + FK)$
Justificación: El costo de la operación nuevoJuego del modulo Juego es $O(N^2 + |\Sigma|K + FK)$. Luego el costo de inicializar un vector vacio es O(1). El costo de inicializar el vector de notificaciones es O(k) * O(1) (costo de inicializar una tupla y una cola vacia). Finalmente el costo total del algoritmo es: $O(N^2 + |\Sigma|K + FK) + O(1) + O(k) * O(1) \subseteq O(N^2 + |\Sigma|K + FK)$

```
iconectarCliente (in/out: s srv)
 1    nat id ← longitud(s.jugadores)
 2    agregarAtras(s.jugadores, id) // se agrega el jugador
 3    encolar(notificacionesXJugador[id]_notifs, tupla(idCliente(id),s.timeStamp))
 4    s.timeStamp++ // se encola la notificacion del id del jugador junto al
  timeStamp y luego se incrementa en 1.
 5
 6    if(longitud(s.jugadores) = s.CantEsperados) // si empieza el juego
 7        agregarAtras(s.notificacionesBroadcast,
 8        tupla(Empezar(tamañoTablero(variante(s.j))), s.timeStamp)
 9        s.timeStamp++
 10       agregarAtras(s.notificacionesBroadcast, tupla(turnoDe(0), s.timeStamp))
 11       s.timeStamp++
 12   endif
```

Complejidad: O(1)
Justificación: Todas las operaciones de este algoritmo son O(1) u O(1) amortizado, pues es encolar elementos, incrementar valores o agregar atras a un vector.

```
i#Conectados(in s: srv) → res : nat
1    res ← longitud(s.jugadores)
```

Complejidad: O(1)
Justificación: Devuelve un elemento de la tupla srv.

---

iCantEsperados(in s: srv) → res : nat

```
1     res ← s.CantEsperados;
```

Complejidad: O(1)
Justificación: Devuelve un elemento de la tupla srv.

---

ijuego(in s:servidor) → res : jgo

```
1     res ← s.j
```

Complejidad: O(1)
Justificación: Devuelve un elemento de la tupla srv.

---

iconsultarNotificaciones(in/out s:servidor, in c: nat)

```
1    while(¬esVacia?(s.notificacionesxJugador[c]_notifs)
  || s.notificacionesxJugador[c]_índice < longitud(s.notificacionesBroadcast))
2  // mientras haya notificaciones disponibles.
3        if (esVacia?(s.notificacionesxJugador[c]_notifs))
4  //Si solo tiene pendientes notificaciones generales.
5            s.notificacionesxJugador[c]_índice++
6        else  //Tiene notif pendientes en ambas colas.
7            timeStampJug ← proximo(s.notificacionesxJugador[c]_notifs)_1
8            timeStampBC ←
  s.notificacionesBroadcast[s.notificacionesxJugador[c]_índice]_1
9            if (timeStampJug < timeStampBC) //La prox notif es del jugador.
10                desencolar(s.notificacionesxJugador[c]_notifs)
11           else //La prox notif es general.
12               s.notificacionesxJugador[c]_índice++
13           endif
14       endif
15    endwhile
```

Complejidad: O(n)
Justificación: Recorrer la lista de notificacionesXJugador y desencolar es O(n) y recorrer notificacionesBroadcast es O(n).

```
iRecibirMensaje(in/out s: servidor, in id: nat, in o: ocurrencia)
1    if (jugadaValida?(s.j, o))
2        puntaje_antes_de_ubicar ← puntaje(s.j, id)
3
4       ubicar(s.j, o)
5    //Comunica a todxs la ocurrencia jugada.
6       agregarAtras(s.notificacionesBroadcast, tupla(ubicar(id, o), s.timeStamp))
7       s.timeStamp++
8
9       puntaje_actual ← puntaje(s.j, id)
10       agregarAtras(s.notificacionesBroadcast, tupla(sumaPuntos(id,
  puntaje_actual - puntaje_antes_de_ubicar), s.timeStamp)) //Comunica a todxs los
  puntos sumados.
11       s.timeStamp++
12       //Comunica al jugador las fichas que le fueron repuestas
13       encolar(notificacionesxJugador[id]_notif,
  tupla(reponer(ultimaReposicion(s.j, id)), s.timeStamp)
14       s.timeStamp++
15
16       agregarAtras(s.notificacionesBroadcast, tupla(turnoDe(id + 1 % K),
  s.timeStamp))  //Comunica a todxs el siguiente turno.
17       s.timeStamp++
18    else
19       encolar(notificacionesxJugador[id]_notif, tupla(mal(), s.timeStamp)
20       s.timeStamp++ //Comunica al jugador jugada inválida.
21    endif
```

Complejidad: $O(LMax^2) + O(m * Lmax)$
Justificación: El costo de chequear sí la jugada es válida es $O(LMax^2)$ (heredado del módulo Juego). El ciclo realiza m iteraciones de operaciones O(1) amortizado, donde m es la cantidad de fichas que se ubican en la jugada. Luego el costo del ciclo es O(m). El costo de ubicar la jugada es O(m) y el de acceder al puntaje de un jugador es O(m * Lmax) (heredados ambos del módulo Juego). Luego el resto de todas las operaciones son O(1). Finalmente el costo total del algoritmo es: $O(LMax^2) + O(m) + O(m) + O(m * Lmax) + O(1) \subseteq O(LMax^2 + m * LMax)$

MÓDULO NOTIFICACIÓN

Interfaz

Parámetros formales:

SE EXPLICA CON: notificación
géneros: notif

Operaciones básicas de notificación

IDCLIENTE(**in** id: nat) → res: notif
Pre ≡ {True}
Post ≡ { res $=_{obs}$ IdCliente(id) }
Complejidad: O(1)
Descripción: Crea la notificación idCliente.

EMPEZAR(**in** n: nat) → res: notif
Pre ≡ {True}
Post ≡ {res $=_{obs}$ Empezar(n) }
Complejidad: O(1)
Descripción: Crea la notificación empezar.

TURNODE(**in** id: nat) → res: notif
Pre ≡ {True}
Post ≡ {res $=_{obs}$ turnoDe(id) }
Complejidad: O(1)
Descripción: Crea la notificación turnoDe

UBICAR(**in** id: nat, **in** o: ocurrencia) → res: notif
Pre ≡ {True}
Post ≡ {res $=_{obs}$ Ubicar(id, o) }
Complejidad: O(1)
Descripción: Crea la notificación ubicar.

REPONER(**in** f: vector<letra>) → res: notif
Pre ≡ {True}
Post ≡ { res $=_{obs}$ Reponer(f)}
Complejidad: O(1)
Descripción: Crea la notificación reponer

SUMAPUNTOS(**in** cid: nat, **in** n: nat) → res: notif
Pre ≡ {True}
Post ≡ {res $=_{obs}$ SumaPuntos(cid,n)}
Complejidad: O(1)
Descripción: Crea la notificación sumaPuntos

MAL() →res : notif
Pre ≡ {True}
Post ≡ {res =$_{obs}$ Mal}
Complejidad: O(1)
Descripción: Crea la notificación MAL.

Representación

Estructura

notificación se representa con notif donde notif es

tupla<t : tipoNotif, id: nat, m: nat, f: vector(letra),o: ocurrencia>

donde tipoNotif es Enum(IdCliente, Empezar, TurnoDe, Ubicar, Reponer, SumaPuntos, Mal)

Invariante de Representación

Rep: notif →bool
(∀ t: notif)
Rep(t) = (IdCliente, cid, 0, ∅, ∅) V (Empezar, 0, n, ∅, ∅) V (TurnoDe, cid, 0, ∅, ∅) V(Ubicar, cid, 0, ∅, o) V(Reponer, 0, 0, f , ∅) V (SumaPuntos, cid, n, ∅, ∅) V (Mal, 0, 0, ∅, ∅)

donde cid, n son nat. f es vector<letra>

Función de abstracción

Abs: n:notif → notificacion {Rep(n)}
Abs(n) = n : notificacion /
datos(notif) = notif

Algoritmos

---

iIdCliente(in cid: nat) → res : notif
1 res ← tupla(IdCliente, cid, 0, vector.vacia(), vector.vacia())

---

iempezar(in n: nat) → res: notif
1 res ← tupla(Empezar, 0, n, vector.vacia(), vector.vacia())

---

iTurnoDe(in cid: nat) → res: notif
1 res ← tupla(TurnoDe, cid, 0, vector.vacia(), vector.vacia())

---

iUbicar(in cid: nat, in o: ocurrencia) → res: notif
1 res ← tupla(Ubicar, cid, 0, vector.vacia(), o)

---

iReponer(in f: cola(letra)) → res: notif
1 res ←tupla(Reponer, 0, 0, f , vector.vacia())

---

isumaPuntos(in cid: nat, in n: nat) → res: notif
1 res ←tupla(SumaPuntos, cid, n, vector.vacia(), vector.vacia())

---

iMal() → res: notif
1 res ←tupla(Mal, 0, 0, vector.vacia(), vector.vacia())

Complejidad de todos los algoritmos del módulo: O(1)
Justificación: En todos los algoritmos se devuelve una tupla de longitud constante con elementos constantes (o de generacion constante, como por ejemplo vector.vacia()).

Decisiones Tomadas

(1) Usamos la notación “sombrero” para abstraer las variables y operaciones de la interfaz y estructura del módulo juego. Hacemos un abuso de notación en el resto de los módulos, dando a entender que se espera el mismo comportamiento que en la notación de módulo juego.

(2) ocurrencia es vector(tupla(nat,nat,letra))

(3) Hay determinadas operaciones que implementamos en los módulos que siguen un patrón de tener una estructura para cada jugador, para poder diferenciar entre que jugador realizó cada jugada. De esta forma, por ejemplo, al sumar las últimas fichas jugadas, podemos operar de una forma sencilla solamente sobre las fichas de determinado jugador, y sumar exclusivamente esas. Luego, al implementar el módulo servidor, notamos que la forma en la cuál opera el mismo para hacer las jugadas, termina recurriendo a ciertos comportamientos automáticos. Es decir, por ejemplo, cada vez que un jugador tiene la intención de realizar una jugada, sí la misma es válida, el servidor habilitará la ubicación de las mismas, y en el proceso de notificar al jugador su puntaje, habrá actualizado el puntaje del mismo. De esta manera, en cada turno se actualizarán estos valores y no habrá una estructura de tipo “historial” en las mismas.

De todas formas, decidimos mantener nuestra implementación ya que nuestro módulo juego sigue siendo compatible con el módulo servidor y el mismo seguirá siendo compatible con otro tipo de módulos de servidor que excedan este TP que no necesariamente realizan los mismos comportamientos automáticos. Por ejemplo, un servidor que no notifique los puntajes, y los jugadores decidan sí quieren actualizar su puntaje de forma manual, y esta operación puede ocurrir en cualquier turno.

(4) Asignamos el valor LETRAVACIA a la letra VACIO para poner sobre un casillero libre.

(5) Forma de indexar tuplas: Sea A = tupla(x1,x2) ⇒ A_0 = x1 y A_1 = x2.

(6) Para el módulo tablero decidimos implementar OcurrenciaDePalabras a pesar de que no la utilizamos ya que la misma nos es util para definir el invariante de representación de juego, y está exportada por el TAD tablero.
Además, en el módulo variante, implementamos la operación nuevoVariante, a pesar de que nunca creamos un nuevo variante, la misma es el único generador de variante. Es exportada por el TAD y la consideramos necesaria para la completitud del módulo.

(7) Decidimos extender el modulo Vector con la operación IniciarVectorDimensionado para facilitar la comprensión del TP, ya que lo usamos en repetidas ocasiones.