Ejercicio 1: Diseño

Ejercicio

El sindicato de piratas de Mêlée Island™ desea un sistema para controlar las peleas que ocurren en la isla. En ella conviven varios piratas, que liberan sus tensiones bebiendo grog y peleándose entre ellos. El grog viene en distintas variedades, tales como *Grog Clásico*, *Grog XD*, etc. Cada variedad de grog tiene una graduación alcohólica distinta. Como todo el mundo sabe, la habilidad con la espada es secundaria en las peleas. Lo que más importa es cuántos insultos sabe cada pirata. Los insultos tienen distinta potencia: es clarísimo que decir "Peleas como un granjero" tiene menor potencia que insultar diciendo "Ordeñaré tu sangre hasta la última gota".

En una pelea, gana el pirata que sume la mayor potencia entre todos los insultos que conoce. En caso de empate, gana el pirata que esté más borracho, es decir aquel cuyo último grog bebido haya sido más fuerte. Si aun así hubiese empate, ninguno de ellos gana. Siempre que un pirata no gana, aprende algún nuevo insulto tomado de aquellos que no conoce (si es que no conoce todos ya)[1]. Los piratas recién llegados no conocen ningún insulto, y se sabe que el último grog que bebieron antes de llegar a Mêlée es el *Grog Free*, que tiene 0 % de alcohol.

TAD INSULTO **ES** TUPLA(STRING, NAT)
TAD GROG **ES** TUPLA(STRING, NAT)
TAD PIRATA **ES** NAT

TAD MÊLÉE

generadores

empezar	:	conj(pirata) × conj(insultos)	⟶	melee	
agregarPirata	:	melee m × pirata p	⟶	melee	$\{p \notin \text{piratas}(m)\}$
agregarInsulto	:	melee m × insulto i	⟶	melee	$\{i \notin \text{insultos}(m)\}$
beberGrog	:	melee m × pirata p × grog	⟶	melee	$\{p \in \text{piratas}(m)\}$
pelear	:	melee m × pirata p_1 × pirata p_2	⟶	melee	$\{p_1 \neq p_2 \wedge \{p_1, p_2\} \subseteq \text{piratas}(m)\}$

observadores básicos

piratas	:	melee	⟶	conj(pirata)	
insultos	:	melee	⟶	conj(insulto)	
insultosQueConoce	:	melee m × pirata p	⟶	conj(insulto)	$\{p \in \text{piratas}(m)\}$
últimoGrogQueBebió	:	melee m × pirata p	⟶	grog	$\{p \in \text{piratas}(m)\}$
peleas	:	melee	⟶	dicc(pirata, dicc(pirata, secu(bool)))	

Fin TAD

Diseñe el TAD MÊLÉE respetando los siguientes requerimientos de complejidad temporal en **peor caso**:

- INSULTOS(m) debe resolverse en $O(1)$
- PIRATAS(m) debe resolverse en $O(1)$
- ÚLTIMOGROGQUEBEBIÓ(m, p) debe resolverse en $O(\log(P))$
- INSULTOSQUECONOCE(m, p) debe resolverse en $O(\log(P))$
- PELEAR(m, p_1, p_2) debe resolverse en $O(\log(P))$
- AGREGARINSULTO(m, i) debe resolverse en $O(P \times Ins)$

donde P es la cantidad de piratas en m e Ins es la cantidad de insultos en m. Puede suponer que la longitud de cualquier insulto estará acotada por 200 caracteres.

a) Muestre la estructura de representación propuesta. Explique para qué sirve cada parte de la estructura, o utilice nombres autoexplicativos.

b) Escriba el pseudocódigo del algoritmo PELEAR(**inout** m : estr, **in** p_1 : pirata, **in** p_2 : pirata).

Justifique claramente cómo y por qué los algoritmos, la estructura y los tipos soporte permiten satisfacer los requerimientos pedidos. No es necesario diseñar los módulos soporte, **pero sí describirlos, justificando por qué pueden (y cómo logran)** exportar los órdenes de complejidad que su diseño supone.

[1] HINT: el dameUno de conjunto debería ser determinístico.

Solución

A continuación se presenta una posible solución al ejercicio de diseño. Es importante aclarar que podría haber algunas explicaciones incompletas y que de ninguna manera es un resumen de todos los temas analizados en la clase práctica. No lo tomen como la solución l ejercicio sino como una guía de la solución.

Representación

Representación del módulo Mêlée:

`Mêlée` **se representa con** `estr`

donde `estr` es `tupla`(*insultos*: `conj(insulto)`,
piratas: `conj(pirata)`,
datos_piratas: `dicc(pirata, info_pirata)`,
peleas: `dicc(pirata, dicc(pirata, secu(bool)))`)

donde `info_pirata` es `tupla`(*ult_grog*: `grog`,
insultos_conocidos: `conj(insulto)`,
insultos_no_conocidos: `conjOrd(insulto)`,
potencia: `Nat`)

Explicación de la estructura

- El campo *insultos* representa el conjunto de todos los insultos que se conocen en Mêlée. El conjunto está diseñado sobre *conj* que es el módulo conjunto lineal del apunte de módulos básicos.

  Si se quiere agregar un insulto nuevo al conjunto, se puede utilizar la operación *AgregarRapido* de conjunto lineal, el cual posee complejidad O(1).

- El campo *piratas* representa el conjunto de todos los piratas que están en Mêlée. El conjunto está diseñado sobre *conj* que es el módulo conjunto lineal del apunte de módulos básicos.

- El campo *datos_piratas* es un diccionario. El mismo posee como claves todos los piratas de Mêlée y como significado *info_pirata* que brinda cierta información de cada clave. El diccionario *dicc* está diseñado sobre el módulo *dicc*.

  info_pirata es una tupla que posee:

  - *ult_grog*: el último grog que bebió el pirata.
  - *insultos_conocidos*: el conjunto de todos los insultos que conoce un pirata. El conjunto está diseñado sobre *conj* que es el módulo conjunto lineal del apunte de módulos básicos. Si se quiere agregar un insulto nuevo al conjunto, se puede utilizar la operación AgregarRapido de conjunto lineal, el cual posee complejidad O(1).
  - *insultos_no_conocidos*: el conjunto de todos los insultos de Mêlée que el pirata no conoce. Es importante ver que $insultos\_no\_conocidos \cup insultos\_conocidos = insultos$.

    El conjunto está diseñado sobre *conjOrd* que es una lista ordenada. Los elementos están ordenados en orden creciente según el orden lexicográfico del string de los insultos y la lista es una lista enlazada. Como la lista está ordenada se puede pedir el minimo del conjunto con la operación *min* y se puede eliminar el mismo con $eliminarMin$; ambos poseen costo O(1). AgregarOrdenado un elemento tiene costo O(n), donde n es la cantidad de elementos del conjunto.
  - *potencia*: la suma de las potencias de todos los insultos que se encuentran en *insultos_conocidos*.

- El campo *peleas* guarda el historial de las peleas entre los piratas y el resultado de las mismas. Ambos diccinarios poseen como claves todos los piratas de Mêlée. Ambos diccionarios están diseñados sobre el módulo *dicc*.

  Dado dos piratas $p1$ y $p2$, $obtener(p2, obtener(p1, peleas))$ nos devuelve una $secu(bool)$. La longitud de la secuencia nos indica cuantas veces $p1$ peleó con $p2$ y su contenido nos indica el resultado de cada batalla; el resultado de una batalla es true si $p1$ fue el ganador y $false$ en caso de haber perdido o en caso de empate. Si $p1$ pelea nuevamente con $p2$ el resultado se agrega al final de la secuencia. La secuencia está disañada sobre una lista

enlazada con puntero al primer y al último elemento. Por lo tanto agregar un bool al final de está secuencia posee consto O(1).

- Módulo *dicc*: está diseñado sobre un AVL.

  Como los piratas son Nat, comparar dos claves tiene costo O(1); por este motivo buscar o agregar una clave cuesta O($log\,(P)$). Es por esta razón que podemos justificar las complejidades de algunas operaciones del diccionario:

  - La operación obtener consiste en buscar una clave y devuelver su significado por referencia, posee complejidad O($log\,(P)$).
  - La operación definir busca la clave en el diccionario, si esta está ya estaba definida previamente, o en caso contrario, la agrega. En ambos casos su significado se agrega al diccionario por referencia. Por lo tanto su complejidad es O($log\,(P)$).

  Este diccionario posee un iterador que permite recorrer las claves del diccionario, necesario para la operación *agregarInsulto*. Las operaciones que debe poseer son *crearIt*, *haySiguiente*, *siguiente* que devuelve la clave correspondiente y *Avanzar* el iterador; todas con costo es O(1). Además debe poseer una operación que me devuelva el significado de la clave indicada por el iterador en costo O(1); por ejemplo se puede llamar *significado_siguiente*. Faltaría explicar brevemente cómo estarían diseñadas estas operaciones; al menos es necesario describir cómo se crea el iterador y como avanza.

Justificación de la complejidad pedida

Para justificar la complejidad de las operaciones pedidas se puede explicar en castellano el por qué se cumplen estas complejidades o se puede escribir el código del algoritmo correspondiente. Recuerden que en este ejercicio sólo se pide el pseudocódico de la operación pelear. Para que quede más claro, yo voy a escribir todas las operaciones.

Considero en todos los casos que la asignación $\leftarrow$ se hace por referencia, por lo cual posee costo O(1). También asumo que devolver un campo de una tupla se hace por referencia, por lo cual la complejidad es O(1).

---

iInsultos(in m: estr) $\rightarrow$ *res* : conj(insulto) ▷ $O(1)$

1: $res \leftarrow m.insultos$ ▷ $O(1)$ ya que devuelve el conjunto por referencia

---

iPiratas(in m: estr) $\rightarrow$ *res* : conj(pirata) ▷ $O(1)$

1: $res \leftarrow m.piratas$ ▷ $O(1)$ ya que devuelve el conjunto por referencia

---

iUltimoGrogQueBebio(in m: estr, in p:pirata) $\rightarrow$ *res* : grog ▷ $O(log(P))$

1: $datos \leftarrow obtener(p, m.datos\_piratas)$ ▷ $O(log(P))$ por todo lo explicado previamente.
2: $res \leftarrow datos.grog$ ▷ $O(1)$ ya que devuelve el grog por referencia

---

iInsultosQueConoce(in m: estr, in p:pirata) $\rightarrow$ *res* : conj(insulto) ▷ $O(log(P))$

1: $datos \leftarrow obtener(p, m.datos\_piratas)$ ▷ $O(log(P))$ por todo lo explicado previamente
2: $res \leftarrow datos.insultos\_conocidos$ ▷ $O(1)$ ya que devuelve el conjunto por referencia

---

iAgregarInsulto(in/out m: estr, in insult:insulto) ▷ $O(P * Ins)$

1: $agregarRapido(insult, m.insultos)$ ▷ $O(1)$ ya que la estructura de $m.insultos$ es *conj*
2: $it \leftarrow CrearIt(m.datos\_piratas)$ ▷ $O(1)$ Falta explicar
3: **while** $haySiguiente(it)$ **do** ▷ $O(P * Ins)$. El ciclo se repite P veces.
4: $datos \leftarrow siguiente\_significado(it)$ ▷ $O(1)$ Falta explicar. La asignación genera alising
5: $agregaOrdenado(insult, datos.insultos\_no\_conocidos)$ ▷ $O(Ins)$. Como hay aliasing, también se modifica $m.datos\_piratas$.
6:
7: $avanzar(it)$ ▷ $O(1)$ Falta explicar
8: **end while**

ipelear(in/out m: estr, in p1:pirata, in p2:pirata) ▷ $O(log(P)$

```
1: datos_p1 ← obtener(p1, m.datos_piratas)                ▷ O(log(P)) por todo lo explicado previamente
2: datos_p1 ← obtener(p2, m.datos_piratas)                ▷ O(log(P)) por todo lo explicado previamente
3: potencia_p1 ← datos_p1.potencia                        ▷ O(1)
4: potencia_p2 ← datos_p2.potencia                        ▷ O(1)
5: grog_p1 ← datos_p1.ult_grog                            ▷ O(1) ya que devuelve el grog por referencia
6: grog_p2 ← datos_p2.ult_grog                            ▷ O(1) ya que devuelve el grog por referencia
7:
8: peleas_p1 ← obtener(p2, obtener(p1, m.peleas))         ▷ O(log(P) + log(P)) = O(log(P)) Se genera aliasing
9: peleas_p2 ← obtener(p1, obtener(p2, m.peleas))         ▷ O(log(P) + log(P)) = O(log(P)) Se genera aliasing
10:
11: if ((potencia_p1 > potencia_p2) ∨ (potencia_p1 = potencia_p2 ∧ Π2(grog_p1) > Π2(grog_p2)) then   ▷ O(1)
12:     agregarAtras(True, peleas_p1)        ▷ O(1) ya que peleas_p1 es una lista con puntero al último elemento.
13:                                          ▷ Como hay aliasing, se modifica también m.peleas
14:     agregarAtras(False, peleas_p2)       ▷ O(1) ya que peleas_p1 es una lista con puntero al último elemento.
15:                                          ▷ Como hay aliasing, se modifica también m.peleas
16:     aprenderInsulto(p2, m)                                                    ▷ O(log(P))
17: else
18:     if ((potencia_p2 > potencia_p1) ∨ (potencia_p2 = potencia_p1 ∧ Π2(grog_p2) > Π2(grog_p1)) then ▷ O(1)
19:         agregarAtras(False, peleas_p1)   ▷ O(1) ya que peleas_p1 es una lista con puntero al último elemento.
20:                                          ▷ Como hay aliasing, se modifica también m.peleas
21:         agregarAtras(True, peleas_p2)    ▷ O(1) ya que peleas_p1 es una lista con puntero al último elemento.
22:                                          ▷ Como hay aliasing, se modifica también m.peleas
23:         aprenderInsulto(p1, m)                                                ▷ O(log(P))
24:     else
25:         agregarAtras(False, peleas_p1)   ▷ O(1) ya que peleas_p1 es una lista con puntero al último elemento.
26:                                          ▷ Como hay aliasing, se modifica también m.peleas
27:         agregarAtras(False, peleas_p2)   ▷ O(1) ya que peleas_p1 es una lista con puntero al último elemento.
28:                                          ▷ Como hay aliasing, se modifica también m.peleas
29:         aprenderInsulto(p1, m)                                                ▷ O(log(P))
30:         aprenderInsulto(p2, m)                                                ▷ O(log(P))
31:     end if
32: end if
```

iaprenderInsulto(in p1:pirata, in/out m: estr) ▷ $O(log(P))$

```
1: datos_p1 ← obtener(p1, m.datos_piratas)                ▷ O(log(P)) por todo lo explicado previamente
2: unInsulto ← min(datos_p1.insultos_no_conocido)         ▷ O(1) ya que pido el min de un conjOrd
3: agregarRapido(unInsulto, insultos_conocido))           ▷ O(1) ya que la estructura de m.insultos_conocido es conj
4:                                                        ▷ se modifica además datos_p1 porque hay aliasing
5: unInsulto ← eliminarMin(datos_p1.insultos_no_conocido)  ▷ O(1) ya que borro el min de un conjOrd
6:                                                        ▷ se modifica además datos_p1 porque hay aliasing
7: nueva_pot ← datos_p1.potencia + Π2(unInsulto)          ▷ O(1)
8: nuevos_datos ←< datos_p1.ult_grog, datos_p1.insultos_conocido, datos_p1.insultos_no_conocido, nueva_pot >
9:                                                        ▷ O(1). Cada campo es una referencia a una estructura
10: definir(p1, nuevos_datos, m)                          ▷ O(log(P) por todo lo explicado previamente
```