Universidad Nacional de General Sarmiento

Programación II - Comisión 2 Noche

Trabajo Práctico

Programación II - TP1 1er Cuatrimestre 2018

Martha Semken
Agustín Nieto

APELLIDO Y NOMBRE	LEGAJO	EMAIL
Sánchez, Matías Alejandro	38.391.082/2015	mattisanchez94@gmail.com
Tula, Ignacio Mariano	35.226.620/2014	itula@logos.net.ar

JENGA

TAD JENGA (JUEGO)

Concepto

Un juego de Jenga consiste en dos ***jugadores***, y una ***torre de jenga***. Por turnos, los jugadores deben retirar elementos de los niveles de la torre y colocarlos en el nivel superior. Existiendo una probabilidad, por cada vez que un jugador retira un elemento, de que la torre caiga o quede destruida. Cuando esto ocurra en el turno de un jugador, el mismo pierde el juego, concediendo la victoria al oponente.

Atributos:

- **jugador1** // String con el nombre del jugador 1
- **jugador2** // String con el nombre del jugador 2
- **torre** // Torre de Jenga
- **siguienteTurno** // String que coincide con el nombre cuyo turno sigue.
- **jugando** // Booleano que indica si el juego terminó o no.

Operación	Especificación	Sintaxis
Constructor	Deja a disposición un TAD Juego sobre el que se podrá operar. Se debe entregar parámetro *jugador1* y *jugador2* que representan los nombres de los jugadores. Y por último un entero que representa los niveles que poseerá inicialmente la torre de jenga.	`void jenga(String jugador1, String jugador2, int niveles)`
Jugar	Intenta ejecutar la acción de cada turno. En un modo manual, solicita al usuario ingresar la ficha a mover, en el modo automático, ejecuta un random para decidir qué ficha mueve ese jugador.	`void jugar()`
Ganador	Devuelve el nombre del jugador que ha ganado, si es que existe un ganador. Caso contrario devuelve un string vacío.	`String ganador()`
Mostrar Cadena	Devuelve una Cadena con datos del juego. A que jugador le toca mover, y la torre.	`String toString()`
físicaDeCaidas privada	Calcula las probabilidades de que la torre se derrumbe, y ejecuta un random con dichas posibilidades. Devuelve True si la	`private boolean fisicaDeCaidas ( int`

	torre sigue en pie, o False si se derrumba. Hace uso de Bolillero, una vez definida la probabilidad correspondiente..	`pisosPorEncima, boolean esFichaMedio, boolean quedanDos)`
Bolillero	El parámetro indica la longitud de un subconjunto de 0 a 99 que es la posibilidad de perder. Esta función genera otro random interno, para desplazar el subconjunto que hace perder, para que pueda desplazarse dentro del conjunto 0 a 99. Esto permite generar, suponiendo una probabilidad del 25%, que haya subconjuntos perdedores que comiencen en 0 a 24, y otra ejecución de 74 a 99. Retorna verdadero cuando se gana, y falso cuando el bolillero salió dentro del subconjunto de números perdedores.	`private boolean bolillero(int probabilidadCaida)`
Mover	Mueve el elemento ubicado en la posición “pos” de nivel indicado. Lo desplaza hacia el nivel superior, o crea uno nuevo. Realiza movimiento si físicaDeCaidas lo permite. SI el movimiento se realiza exitosamente devuelve True, caso contrario False. Cambia el valor del estado de la torre a False en caso de que fisicaDeCaidas lo indique Sólo se puede realizar el movimiento si la ficha existe, de lo contrario se arroja excepción. Y además, sólo si hay movimiento posible y luego si la fisicaDeCaida lo permite. Se define como movimiento posible, un nivel con al menos 2 fichas. Un nivel con una ficha sola, donde se retira esta misma, produce caida de torre y pérdida del juego.	`boolean mover(int nivel, int pos)`

TAD TORRE

Concepto

Una torre de Jenga es una colección o agrupación de ***niveles*** no vacíos, en la cual existe una relación de orden entre cada uno de ellos. En dicha torre se pueden retirar elementos que no sean del nivel superior. Y sólo se pueden agregar elementos en el nivel superior.

Posee un estado inicial con una “n” cantidad definida de niveles completos y que puede aumentar a una cantidad máxima de “n * 3” niveles incompletos.

Dicho aumento se produce por el retiro de ***elementos*** de cualquier nivel (excepto del último) para ser colocados en el nivel final o superior.

Atributos:

- **estadoDePie** Indica si la torre permanece de pié, o se derrumbó por algún intento de Mover.
- **Niveles** Colección de niveles, ordenadada. Cada nivel tiene tres posiciones.

Operación	Especificación	Sintaxis
Constructor	Deja a disposición un TAD Torre sobre el que se podrá operar. Se debe entregar parámetro un entero que representa los niveles que poseerá inicialmente la torre de jenga.	`void torre(int niveles)`
Ultimo Nivel	Obtiene el índice del último nivel de la torre	`private int ultimoNivel()`
Es Ficha Central	Devuelve verdadero cuando la posición indicada como parámetro es 1 (Elemento central en TAD Nivel)	`static boolean esFichaCentral(int pos)`
Hay 3 en el Nivel	Indica si en el nivel dado hay 3 fichas. De lo contrario devuelve false.	`private boolean hay3EnNivel(int nivel)`

Quitar Ficha	Quita la ficha de la posición y nivel indicados. No calcula si es posible retirarla, eso se en Jenga->Mover(nivel,pos);	`private void quitarFicha(int nivel, int pos)`
Agregar Ficha	Agrega una ficha	`private void agregarFicha()`
Movimientos Posibles	Cuenta los elementos del nivel indicado. Si hay sólo un elemento, no hay movimiento posible (pues realizarlo rompería el invariante de esta torre, que sería lo mismo que una torre derrumbada). Si existen 2 o 3 elementos, entonces devuelve true	`public boolean hayMovimientoPosible(int nivel)`
Existe Ficha	Devuelve verdadero cuando existe un elemento (cuando el elemento es true), en el nivel indicado y en la posición indicada. Falso de lo contrario.	`public boolean existeFicha(int nivel, int pos)`
A Cadena	Devuelve una cadena que representa la torre, junto con el índice de cada nivel. La implementación de esta función utiliza iteradores y StringBuilder como indica la consigna	`public String toString()`

TAD NIVEL

Concepto

Es un conjunto de tres elementos, ordenados y cuyos valores pueden ser true o false. Pero no es posible que todos ellos sean false. Pueden ser creados de forma completa (sus tres posiciones en true) o pueden ser creados de forma incompleta (una posición en true, el resto en false).

Se menciona que existe un elemento, o hay un elemento cuando el valor en dicha posición del nivel es true.

Retirar un elemento es cambiar su valor a false.

Operación	Especificación	Sintaxis
Constructor	Deja a disposición un TAD Nivel sobre el que se podrá operar. Se crea un nivel con todos valores en true si completo es true. Se crea un nivel con sólo un elemento true si completo es false.	`void nivel(boolean completo)`
Retirar	Retira un elemento (colocando su valor en false) de la posición indicada. Lanza una excepción si esa posición ya es false.	`void retirar (int pos)`
Agregar	Agrega un elemento en la primera posición libre (valor false) y la coloca en true.	`void agregar()`
Obtener Elementos	Devuelve el valor indicado del elemento en la posición solicitada.	`boolean obtenerElementos(int pos)`

INVARIANTE DE REPRESENTACIÓN

Jenga

- No puede existir ganador si se está jugando.
- No se puede estar jugando si la torre está derrumbada.
- La cantidad de elementos de la torre debe permanecer constante. (Pueden variar de ubicación, pero ficha que es quitada, debe ser nuevamente agregada).
- Si n es la cantidad de niveles iniciales, entonces la torre sólo puede extenderse hasta tener una cantidad de niveles 3*n.

Torre

- Una torre sólo permanece de pié en los siguientes casos:
    - Todos los niveles son no-vacíos. Es decir todos los niveles poseen al menos un elemento. Intentar o realizar una operación de sustracción de un elemento único de nivel, derrumba la torre.
    - En los niveles que tengan un sólo elemento, dicho elemento único debe ser el central del nivel. Si el elemento único, se encuentra en los extremos del nivel, se derrumba la torre.
- El nivel n+1 sólo se puede agregar o existir cuando el nivel n tiene tres elementos (con valor true)

¿Cuántas piezas tiene que tener un nivel para ser elegible para quitar piezas?

Un nivel debe tener al menos dos piezas para que sea elegible para quitar piezas. En esta implementación, se le permite al usuario retirar la única pieza del nivel, pero eso le concede automáticamente la victoria al oponente.

Se optó por esta decisión de diseño para cumplir con el Junit ej1 test2. En donde un juego de jenga debe terminar derrumbado por quitarse todas las fichas de un mismo nivel.

Notar, que internamente, luego de quitar una pieza hay que agregarla en el último nivel o agregar un nivel nuevo. ¿Quien se debería encargar de esto?

Quien se encarga de mantener constante la cantidad de piezas es el juego Jenga junto con su reglamento. Donde cada ciclo de quitar una ficha, debe ser concluido con volver a agregarla.

La constancia en la cantidad de fichas y la obligación de mover una por turnos es el principio básico del juego que pone en riesgo el derrumbe de la torre y la victoria hacia el otro jugador.

La torre sólo limita por sus características a agregar elementos en el nivel superior. El Tad torre brinda elementos de operaciones sobre los niveles que ella posee, respetando niveles no vacíos y agregados sobre el último nivel.

Acoplamiento y Cohesión

Se ha decidido no modelizar las dimensiones física de la ficha puesto que esta no incide en la caída de la torre. La caída de la torre se produce por el reglamento del Jenga en el apartado de probabilidades. La existencia de cada ficha, es representada con valor true en la posición del nivel que ocupa.

Tampoco la disposición perpendicular de las fichas entre cada nivel del Jenga por el mismo motivo anterior.

Existe un acoplamiento en la clase Jenga generado en el proceso de implementación, en particular en las funciones Quitar (que ya existe en el Tad Torre) y Primer Nivel Posible). Esta decisión se tomó para que coincida con el Junit Ejer1 test2.

Puesto que Jenga sólo quita fichas usando la función de Torre quitar. Y sólo lo hace mediante la función mover, que sólo ejecuta movimientos cuando las probabilidades fueron favorables.

ACLARACIÓN:

Se agregó una quinta posibilidad de que se caiga la torre a las indicadas en el ejemplo de la consigna. Interpretando por el caso 4, que una torre con un solo elemento en el nivel, y que el mismo no se encuentra en el centro, produce la caída de la torre en el 100% de los casos.

Se detectó que en el caso 3, la ficha que se retira es la de un costado, pero que no necesariamente evalúa dónde se ubica la ficha restante. Si la ficha restante se encontrara en el centro, se produce el caso 3 sin inconveniente. Sin embargo si la ficha que se retira es la un costado, y la restante no es del centro (es decir que se ubica en el otro extremo). El resultado de realizar este movimiento es como quedaría un movimiento de caso 4.

Por lo que se decidió implementar los siguientes subcasos en la ejecución del caso3:

- Si la ficha a retirar es del costado, queda una sola ficha y la misma es del centro, se aplica el caso 3.
- Si la ficha a retirar es del costado, queda una sola ficha y la misma es del otro extremo, se aplica como si fuera caso 4 (100% de probabilidad de caída).

ÁRBOL BINARIO DE BÚSQUEDA

ÓRDENES DE COMPLEJIDAD

balanceado()

El orden de complejidad de aplicar la función balanceado se expresa como $O(n^2)$ donde *n* es la cantidad de nodos del árbol. Puesto que el balanceado al aplicarse recursivamente debe recorrer cada nodo (por derecha y por izquierda). Y cómo por cada vez que se recorre se debe calcular la *altura()* cuyo orden es $O(n)$ donde *n* es la cantidad de nodos del árbol.

El resultado por álgebra de órdenes $O(n) * O(n) = O(n * n) = O(n^2)$

Se considera que el orden de complejidad de calcular la altura es $O(n)$ puesto que en este caso el árbol no se encontraría balanceado. Si al árbol estuviera balanceado, el orden debería ser $O(log(n))$ con *n* representando la cantidad de nodos, porque por cada nodo, se descarta un subárbol y se procesa el restante.

rebalancear()
balancear(Nodo<Integer>)

El orden de complejidad de rebalancear se expresa como $O(n^2)$ considerando que la recursión requiere comprobar en el peor de los casos, por cada nodo, el *balance* de cada subárbol o la diferencia de *altura* entre los mismos.

En esta implementación, rebalancear sólo se ejecuta cuando balanceado() es false. Por lo que en el peor de los casos $O(n^2)$ se sumará al resto del orden de complejidad de esta función.

recambioIzquierda(Nodo<Integer>)
recambioDerecha(Nodo<Integer>)
Las funciones de recambio tienen un número constante de operaciones elementales por lo que su complejidad se define como orden $O(1)$

iesimo(int)
iesimo(Nodo<Integer>, int, int)

La función i-ésimo tiene un orden de complejidad que se puede expresar como $O(n)$ puesto que al calcular la cantidad de nodos del subárbol izquierdo para conocer el índice del nodo actual (y luego tomar la decisión sobre cuál subárbol moverse) se debe contar cada uno de ellos. El peor caso, sería el último i-ésimo del árbol, donde se debería recorrer cada nodo de los subárboles izquierdos.

FUNCIÓN REBALANCEAR

En primer lugar, la función rebalancear sólo ejecuta el grueso de dicho algoritmo, si el árbol no se encuentra balanceado. Cómo es posible que un balanceado requiera de varias operaciones, se espera que se ejecuten dichas operaciones desde el principio mientras el árbol no esté balanceado.

```
while (!this.balanceado())
```

Si la raíz no es nula, se hacen los siguientes cálculos y se toman las siguientes decisiones.

A)
Si la diferencia de la altura del subárbol izquierdo con respecto al derecho indica que el subárbol izquierdo es mayor en 2 unidades. Significa que el árbol está bastante desbalanceado por la izquierda, por lo que deben cambiarse elementos de la izquierda hacia la derecha.

```
if (altura(this.raiz.izq) - altura(this.raiz.der) > 2) {

                this.raiz = recambioDerecha(this.raiz);

            }
```

Si la diferencia es menor que -2. Significa lo opuesto. El árbol está bastante desbalanceado por la derecha, por lo deben cambiarse elementos de la derecha hacia la izquierda.

```
else if (altura(this.raiz.izq) - altura(this.raiz.der) < -2) {

                this.raiz = recambioIzquierda(this.raiz);
            }
```

Aclaración: Cambiar un elemento de la izquierda hacia la derecha (llamar a la función *recambioDerecha*). En el caso de ejecutarse sobre la raíz del árbol, significa que, el elemento *raiz.izq* pasará a ser la nueva raíz del árbol, y la ex-raíz pasará a ser un elemento hijo hacia derecha de la nueva raíz. En el caso del *recambioIzquierda*, es igual pero en sentido opuesto.

B)
En un siguiente estamento, se analizan los siguientes condicionales:

La diferencia entre el subárbol izquierdo y el derecho es de 2. Significa que el árbol está desbalanceado, con más elementos en la izquierda, pero solamente sobra un nivel.

Llegado este punto se analiza la probabilidad si el subárbol izquierdo se encuentra desbalanceado, si es así, se decide rebalancear sólo este subárbol

En caso de que el subárbol izquierdo esté balanceado, significa que el desbalance del árbol se produce en la raíz, por lo que se decide hacer un recambio a derecha sobre esta misma.

```
else if (altura(this.raiz.izq) - altura(this.raiz.der) == 2) {
      if (balanceado(this.raiz.izq)) {
            this.raiz = recambioDerecha(this.raiz);
      } else {
            this.raiz.izq = this.balancear(this.raiz.izq);
                              }
```

El caso opuesto es igual, con la diferencia del sentido opuesto y con una diferencia entre izquierdo y derecho de -2.

C)

Puede ocurrir, que la diferencia entre el subárbol izquierdo y derecho sea de 1, -1 ó 0. Que son casos no analizados en **A** y **B**. No son analizados porque en principio, estas diferencias indican que no existe desbalance en esta posición, sin embargo, que exista este resultado de la diferencia de subárboles en una función que se ejecuta sólo cuando el árbol general no está balanceado, tiene relevancia.

Estos casos ocurren cuando no existe balance dentro de alguno de los propios subárboles.

Por lo que si ocurre una diferencia de subárboles no contemplada en **A** y **B**, se solicita balancear el árbol que comienza en el nodo izquierdo de la raíz si el mismo no está balanceado, y lo respectivo con el nodo derecho de la raíz si fuese el caso.

```
} else {
      if (!balanceado(this.raiz.izq)) {
            this.raiz.izq = this.balancear(this.raiz.izq);
      } else {
            this.raiz.der = this.balancear(this.raiz.der);
      }
}
```

D) Con una lógica similar opera la función recursiva y privada balancear, que opera sobre los nodos.

Si el subárbol izquierdo y derecho del nodo actual poseen una diferencia mayor a 2, Se balancea el subárbol izquierdo. A excepción de que el subárbol derecho no exista, si ocurre esto último, se cambia hacia derecha sobre este mismo nodo. La lógica es inversa de derecha a izquierda si la diferencia es menor que -2.

```
if (altura(nodo.izq) - altura(nodo.der) > 2) {
      if (nodo.der != null) {
            nodo.izq = balancear(nodo.izq);
      } else {
            nodo = recambioDerecha(nodo);
      }
}
```

Si la diferencia entre subárboles es 2 se recambia a derecha. Y si la misma es -2 se recambia a izquierda.

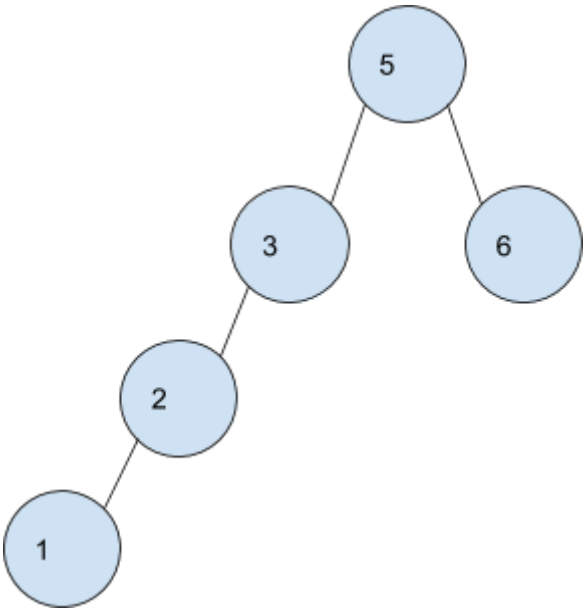
```
else if (altura(nodo.izq) - altura(nodo.der) == 2) {
      nodo = recambioDerecha(nodo);
            }
```

Si llegara a existir diferencias no comprendidas (1, -1, 0) entonces se analiza la existencia de cada subárbol y si se encuentra balanceado. En caso de no estarlo, se solicita se balancee.

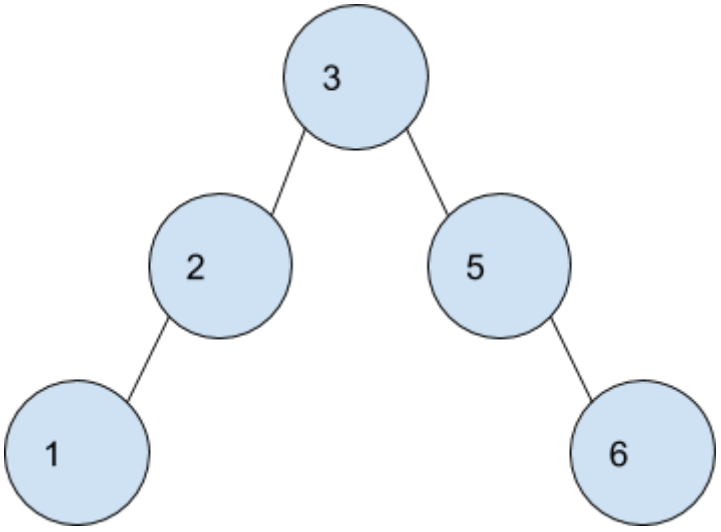
```
if (nodo.izq != null) {
      if (!balanceado(nodo.izq)) {
            nodo.izq = balancear(nodo.izq);
      }
}
```

E) Las funciones *recambioIzquierda* y *recambioDerecha* permite modificar la raíz o el nodo actual y colocar a un hijo en su lugar.

Supongamos el siguiente árbol.



Realizar un recambio a izquierda sobre el nodo 5 generaría el siguiente efecto.



Al aplicar recambio a izquierda.

FUNCIÓN IESIMO

La función i-ésimo requiere conocer cuántos nodos existen siempre en el subárbol izquierdo de la posición actual (sea la raíz o un nodo). Sabiendo que la posición actual es la cantidad de nodos.

La posición i-ésima del nodo actual es igual a la cantidad de nodos de su subárbol izquierdo. Siendo n la cantidad de nodos, y n-1 la posición del elemento anterior. Siempre ocurrirá lo indicado al principio de este párrafo.

SI la i-ésima posición actual es igual a la solicitada, se ha encontrado la solución. Si la posición solicitada es menor, se ejecuta esta función pero en el nodo izquierdo del actual.

Si la posición solicitada es mayor, se ejecuta esta función en el nodo derecho, indicando un offset (de posiciones ya recorridas, llamado contador).

Cuando se vuelva a ejecutar esta función para calcular la posición i-ésima actual de este nodo. Requerirá saber nuevamente cuántos nodos hay en el nuevo subárbol izquierdo y se les sumará al offset previamente indicado, el cual ayuda a mantener la cuenta.

Se seguirá llamando a la función, hasta encontrar la posición i-ésima solicitada.