Práctica 3, Ejercicio 5

Alejandro Santos - Programación III - UNLP

Cursada 2013 (10 de mayo de 2013)

Índice

1. Enunciado

Se define el valor de trayectoria pesada de una hoja de un árbol binario como la suma del contenido de todos los nodos desde la raíz hasta la hoja multiplicado por el nivel en el que se encuentra. Implemente un método que, dado un árbol binario, devuelva el valor de la trayectoria pesada de cada una de sus hojas. Considere que el nivel de la raíz es 1.

Para el ejemplo de la figura: trayectoria pesada de la hoja 4 es: $(4 * 3) + (1 * 2) + (7 * 1) = 21$.

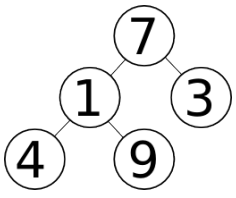

Figura 1: Ejercicio 5

2. Forma de encarar el ejercicio

El ejercicio se puede dividir en tres partes:

1. Recorrido del árbol.
2. Procesamiento de los nodos.
3. Almacenamiento del resultado.

2.1. Recorrido del árbol

Existen diferentes maneras de recorrer árboles, cada cual con sus ventajas y desventajas. Un recorrido recursivo puede ser interesante, en especial el preorden o inorden. Cualquiera de éstos se puede aplicar al ejercicio. Por ejemplo, a continuación1 se puede ver el recorrido preorden.

Algorithm 1 versión en pseudocódigo de un recorrido preorden.

```
function ℝ(A)
    if A no es vacío then
        Procesar(A.dato)
        if A tiene hijo izquierdo then
            ℝ(A.Izq)
        end if
        if A tiene hijo derecho then
            ℝ(A.Der)
        end if
    end if
end function
```

2.2. Procesamiento de los nodos

El el dibujo de ejemplo se pueden ver tres hojas. El valor de la trayectoria de cada hoja es:

- Hoja 4: $7 \times 1 + 1 \times 2 + 4 \times 3$.
- Hoja 9: $7 \times 1 + 1 \times 2 + 9 \times 3$.
- Hoja 3: $7 \times 1 + 3 \times 2$.

En todas las hojas, como parte del valor de la trayectoria se encuentra la suma parcial de 7×1, y en todos los nodos hijos del nodo 1 se encuentra la suma parcial de 1×2. En general, para un nodo cualquiera del árbol todos sus hijos van a tener como parte de su trayectoria la suma parcial del camino que existe desde la raíz hasta ese nodo.

Esto significa que, recursivamente, podemos acumular el valor del nodo actual multiplicado por el número de nivel y pasarlo por parámetro en el llamado recursivo de ambos hijos.

Actualizando el pseudocódigo2 tenemos el recorrido preorden $\mathbb{R}$ con tres parámetros: el árbol (A), el número de nivel actual (N), y la suma acumulada desde la raíz hasta el nodo actual (V):

Algorithm 2 versión en pseudocódigo del procesamiento de nodos.

function $\mathbb{R}(A, N, V)$
  if A no es vacío **then**
    Procesar(A.dato, N, V)
    if A tiene hijo izquierdo **then**
      $\mathbb{R}(A.\text{Izq}, N + 1, V + N \times A.\text{dato})$
    end if
    if A tiene hijo derecho **then**
      $\mathbb{R}(A.\text{Der}, N + 1, V + N \times A.\text{dato})$
    end if
  end if
end function

2.3. Almacenamiento del resultado

En ambas versiones de pseudocódigo anteriores se encuentra el llamado a la función "`Procesar`". El enunciado pide devolver el valor de cada una de las hojas del árbol, por lo que hace falta utilizar una estructura que permite almacenar y devolver múltiples elementos. En nuestra materia, la estructura más adecuada que tenemos se llama `ListaGenerica<T>`, que nos permite almacenar objetos de cualquier tipo que corresponda con el parámetro de comodín de tipo. Otras opciones pueden ser `ListaDeEnteros` o `ColaDeEnteros`.

Hace falta hacer un pequeño análisis acerca de qué debe hacerse en "`Procesar`". El ejercicio pide almacenar el valor de la trayectoria, donde éste es un valor que se calcula a partir del dato propio de cada nodo y del nivel, teniendo en cuenta que el dato de la hoja también debe ser almacenado. Esto significa dos cosas, por un lado, al hacer el recorrido recursivo el dato no puede almacenarse hasta no visitar una hoja, y por el otro el valor de la hoja debe ser parte del resultado final.

En Java, "`Procesar`" puede ser una llamada a un método separado, o puede ser simplemente el código mismo. Por ejemplo:

```
1 private void Procesar(ArbolBinario<T> A, int N, int V, ListaGenerica<
    Integer> lista) {
2   if (A.esHoja()) {
3
4     lista.agregar(V + (Integer)A.getDatoRaiz() * N);
5   }
6 }
```

2.4. Detalles de implementación

2.4.1. Modularización

Modularizar y separar el código en diferentes funciones o métodos suele ser una buena idea. Hasta ahora vimos que el ejercicio se puede separar en dos métodos. Sin embargo, se pide devolver los valores de todas las hojas, por lo que hace falta que el método principal del ejercicio tenga como tipo de retorno `ListaGenerica<T>` (o la estructura elegida). Esto puede complicar la implementación del recorrido preorden, por lo que puede ser de mucha ayuda tener otro método aparte que se encargue del recorrido recursivo. Una buena forma de modularizar este ejercicio puede ser como el siguiente ejemplo:

```
1  class EjercicioCinco {
2    public static ListaGenerica<Integer> CalcularTrayectoria(
       ArbolBinario<T> A) {
3      ListaGenerica<Integer> lista = new ListaEnlazadaGenerica<
         Integer>();
4
5      Recorrido(A, 1, 0, lista);
6
7      return lista;
8    }
9
10   private static void Recorrido(ArbolBinario<T> A, int N, int V,
       ListaGenerica<Integer> lista) {
11     // Metodo del recorrido "R" preorden recursivo.
12     if (!A.esVacio()) {
13       // ...
14     }
15   }
16
17   private static void Procesar(ArbolBinario<T> A, int N, int V,
       ListaGenerica<Integer> lista) {
18     // ...
19   }
20 }
```

2.4.2. `esHoja()`

La definición del método `esHoja()` de la clase `ArbolBinario<T>` puede ser:

```
1  class ArbolBinario<T> {
2    public boolean esHoja() {
3      return !tieneHijoIzquierdo() && !tieneHijoDerecho();
4    }
5
6    public boolean tieneHijoIzquierdo() {
7      return getRaiz().getHijoIzquierdo() != null;
8    }
9
10   public boolean tieneHijoDerecho() {
11     return getRaiz().getHijoDerecho() != null;
12   }
13 }
```

2.4.3. `public` y `private`

En Java, la forma que se tiene de abstraer y encapsular código y datos es mediante los **especificadores de acceso**. Dos de ellos son `public` y `private`, que permiten definir desde qué parte del código se puede invocar un método o acceder a una variable.

En el caso de este ejercicio, es importante que el método principal del recorrido sea `public` a fin

de que los usuarios del `ArbolBinario` lo puedan utilizar. Por otro lado, tener los métodos auxiliares como `private` permite que los detalles de implementación permanezcan ocultos, dejando la libertad que en un futuro sea posible modificar el código y los detalles de implementación teniendo el menor impacto en el mantenimiento del resto del código donde se utilizan estas llamadas.

2.4.4. Clases abstractas

El operador "`new`" solo puede ser usado para instanciar clases que no sean `abstract`. Por ejemplo, el siguiente programa no es correcto, ya que el operador `new` se está usando para instanciar una clase `abstract`.

```
1 public abstract class X {
2   public X() {
3     // ...
4   }
5
6   public static void main(String[] args) {
7     X y = new X(); // Error!
8   }
9 }
```

En particular, no es posible crear una instancia de `ListaGenerica<T>`, ya que esta es una clase `abstract`. Es necesario instanciar una subclase que no sea `abstract`. Por ejemplo, `ListaEnlazadaGenerica<T>`.

```
1 public void f() {
2     ListaGenerica<Integer> L = new ListaEnlazadaGenerica<Integer>();
3 }
```

3. Errores comunes en las entregas

1. No devolver los valores. El enunciado pide devolver los valores, por lo que imprimirlos mediante `System.out.print();` no sería correcto. Hace falta utilizar una estructura que permita almacenar una cantidad variable de elementos para luego devolver la estructura completa.

2. No sumar el valor de la hoja como parte de la trayectoria. Al momento de agregar el valor de la trayectoria hace falta incluir el valor de la hoja multiplicado por su nivel. Manteniendo la idea explicada hasta ahora, el siguiente resultado es incorrecto:

```
1 private static void Procesar(ArbolBinario<T> A, int N, int V,
    ListaGenerica<Integer> lista) {
2   if (A.esHoja()) {
3     lista.agregar(V); // Incorrecto, falta calcular hoja
4   }
5 }
```

3. No preguntar inicialmente si el árbol es vacío. No solo hace falta preguntar si el arbol tiene hijo izquierdo o hijo derecho, sino también si el arbol original inicial tiene algún dato. Eso se logra preguntando si la `raíz` de `ArbolBinario` es `null`, o mediante el uso del método `esVacio()`.

```
1 private static void R(ArbolBinario<T> A) {
2   if (!A.esVacio()) {
3     // ...
4   }
5 }
```

Donde el método `esVacio()` se define dentro de la clase `ArbolBinario` como:

```
1 class ArbolBinario<T> {
2   public boolean esVacio() {
3     return getRaiz() == null;
4   }
5 }
```

4. Uso del operador "`++`" a derecha para incrementar de nivel. El operador "`++`" a derecha, por ejemplo en el caso de "`n++`", hace que el valor de la variable `n` se modifique, pero este incremento se realiza después de devolver el valor original. Si se usa `n++` como incremento de nivel, y esta expresión está directamente en el llamado recursivo, el efecto deseado no se cumplirá. En cada llamada recursiva, el parámetro `n` siempre tomará el mismo valor, y no se estará incrementando el nivel.

```
1 public static void R(ArbolBinario<T> A, int N) {
2   if (A.esHoja()) {
3     // El parámetro N nunca llega al llamado recursivo con el valor
        incrementado.
4   } else {
5     if (A.tieneHijoIzquierdo()) R(A.getHijoIzquierdo(), N++);
6     if (A.tieneHijoDerecho()) R(A.getHijoDerecho(), N++);
7   }
8 }
```

5. Pasar por parámetro el árbol en un método no `static` de la clase `ArbolBinario` genera que se esten recorriendo dos árboles a la vez. Por un lado, el árbol que llega por parámetro, y por el otro el arbol de "`this`".

```
1 class ArbolBinario<T> {
2   public ListaGenerica<Integer> Trayectoria(ArbolBinario<T> A) {
3     // Acá hay dos árboles. "this" y "A".
4   }
5 }
```

La solución a esto puede ser:

a) Indicar los métodos de recorrido como `static`,

b) Quitar el parámetro y hacer el recorrido del arbol con "`this`".

c) Implementar los métodos en una clase diferente a `ArbolBinario`, por ejemplo dentro de la clase "`EjercicioCinco`".

6. Preguntar por la existencia del hijo izquierdo o derecho a partir del valor devuelto por `getHijo Izquierdo()` de `ArbolBinario`.

```
1 class ArbolBinario<T> {
2   public ArbolBinario<T> getHijoIzquierdo() {
3     return new ArbolBinario<T>(getRaiz().getHijoIzquierdo());
4   }
5 }
```

De acuerdo a la definición vista en clase de `ArbolBinario`, los métodos `getHijoIzquierdo` y `getHijoDerecho` de `ArbolBinario` nunca devuelven `null`, y por lo tanto la siguiente pregunta nunca va a ser `True`, porque el operador "`new`" nunca devuelve `null`.

```
1 if (arbol.getHijoIzquierdo()==null) {
2   // Nunca puede ser True.
3 }
```

Lo ideal en este caso es utilizar los métodos `tieneHijoIzquierdo` y `tieneHijoDerecho`, implementados dentro de la clase `ArbolBinario` tal como se muestra en la sección *Detalles de implementación*.

7. Variables en `null`. El siguiente código dispara siempre un error de `NullPointerException`. Si la condición del **if** da `True` entonces "`lista`" es `null`, y de ninguna forma es posible utilizar la variable "`lista`" porque justamente su valor es `null`.

```
1 if(lista==null)
2 {
3   lista.agregar(sum);
4 }
```

8. Preguntar si `this` es `null`. El lenguaje Java garantiza que "`this`" jamás pueda ser `null`, ya que al momento de invocar un método de instancia en una variable con referencia `null`, se dispara un error de `NullPointerException`.