Unidad VI - Árboles

- Organización Jerárquica con Árboles
- Recursión

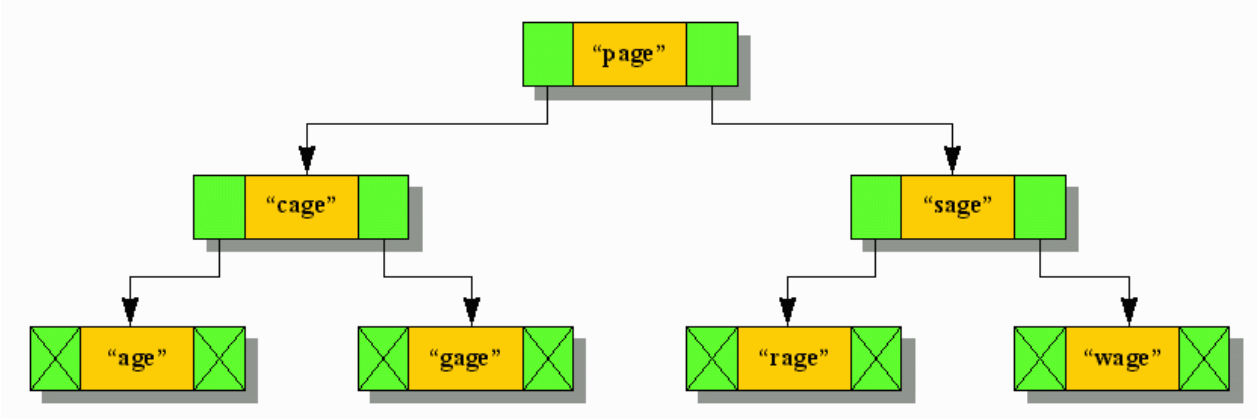
- Java en castellano (Tutorial Estructuras de Datos y Algoritmos en Java)
  http://www.programacion.com/java/tutorial/jap_data_alg/
- Tutorial original en Inglés
  http://www.javaworld.com/

Árboles

Organización Jerárquica con Árboles

Un *árbol* es un grupo finito de nodos, donde uno de esos nodos sirve como *raíz* y el resto de los nodos se organizan debajo de la raíz de una forma jerárquica. Un nodo que referencia un nodo debajo suyo es un *nodo padre*. De forma similar, un nodo referenciado por un nodo encima de él, es un *nodo hijo*. Los nodos sin hijos, son *nodos hoja*. Un nodo podría ser un padre e hijo, o un nodo hijo y un nodo hoja.

Un nodo padre podría referenciar tantos hijos como sea necesario. En muchas situaciones, los nodos padre sólo referencian un máximo de dos nodos hijos. Los árboles basados en dichos nodos son conocidos como **árboles binarios**. La siguiente figura representa un árbol binario que almacena siete palabras en orden alfabético.



Insertar nodos, borrar nodos, y atravesar los nodos en árboles binarios o de otros tipos se realiza mediante la recursión (vea el capítulo siguiente). Por brevedad, no entraremos en los algoritmos recursivos de inserción, borrado y movimiento por los nodos. En su lugar, presentaré el código fuente de una aplicación que cuenta palabras para demostrar la inserción y el movimiento por los nodos. Este código utiliza inserción de nodos para crear un árbol binario, donde cada nodo contiene una palabra y un contador de ocurrencias de esa palabra, y muestra estas palabras y contadores en orden alfabético mediante una variante del algoritmo de movimiento por árboles *move-left-examine-node-move-right*:

```
// WC.java

import java.io.*;

class TreeNode {
     String word;              // Word being stored.
     int count = 1;            // Count of words seen in text.
```

```
     TreeNode left;             // Left subtree reference.
     TreeNode right;            // Right subtree reference.

     public TreeNode (String word) {
            this.word = word;
            left = right = null;
     }

     public void insert (String word)  {
            int status = this.word.compareTo (word);

            if (status > 0) {    // word argument precedes current word

                    // If left-most leaf node reached, then insert new node as
                    // its left-most leaf node. Otherwise, keep searching left.
                    if (left == null)
                            left = new TreeNode (word);
                    else
                            left.insert (word);
            }
            else
                if (status < 0) {    // word argument follows current word

                    // If right-most leaf node reached, then insert new node as
                    // its right-most leaf node. Otherwise, keep searching right.

                    if (right == null)
                            right = new TreeNode (word);
                    else
                            right.insert (word);
                }
                else
                    this.count++;
     }
}

class WC {
     public static void main (String [] args) throws IOException  {
            int ch;

            TreeNode root = null;

            // Read each character from standard input until a letter
            // is read. This letter indicates the start of a word.

            while ((ch = System.in.read ()) != -1) {
                 // If character is a letter then start of word detected.

                 if (Character.isLetter ((char) ch)) {
                         // Create StringBuffer object to hold word letters.

                         StringBuffer sb = new StringBuffer ();

                        // Place first letter character into StringBuffer object.
                         sb.append ((char) ch);

                         // Place all subsequent letter characters into
StringBuffer
                         // object.
                         do {
                                ch = System.in.read ();
                                if(Character.isLetter ((char) ch))
                                        sb.append((char) ch);
                                else
                                        break;
                         }
                         while (true);
                         // Insert word into tree.
                         if (root == null)
                                root = new TreeNode (sb.toString ());
                         else
                                 root.insert (sb.toString ());
```

```
                }
            }
            display (root);
    }

    static void display (TreeNode root) {
            // If either the root node or the current node is null,
            // signifying that a leaf node has been reached, return.
            if (root == null)
                    return;

            // Display all left-most nodes (i.e., nodes whose words
            // precede words in the current node).
            display (root.left);

            // Display current node's word and count.
            System.out.println ("Word = " + root.word + ", Count = " +
                                                    root.count);

            // Display all right-most nodes (i.e., nodes whose words
            // follow words in the current node).
            display (root.right);
    }
}
```

Como tiene muchos cometarios no explicaré el código. En su lugar le sugiero que juegue con esta aplicación de esta forma: cuente el número de palabras de un fichero, lance una línea de comandos que incluya el símbolo de redirección `<`. Por ejemplo, cuente el número de palabras en `WC.java` lanzando `java WC <WC.java`. Abajo puede ver un extracto de la salida de este comando:

```
Word = Character, Count = 2
Word = Count, Count = 2
Word = Create, Count = 1
Word = Display, Count = 3
Word = IOException, Count = 1
Word = If, Count = 4
Word = Insert, Count = 1
Word = Left, Count = 1
Word = Otherwise, Count = 2
Word = Place, Count = 2
Word = Read, Count = 1
Word = Right, Count = 1
Word = String, Count = 4
Word = StringBuffer, Count = 5
```

Recursión

La ciencia de la computación hace tiempo que descubrió que se puede reemplazar a un método que utiliza un bucle para realizar un cálculo con un método que se llame repetidamente a sí mismo para realizar el mismo cálculo. El hecho de que un método se llame repetidamente a sí mismo se conoce como *recursión*.

La recursión trabaja dividiendo un problema en subproblemas. Un programa llama a un método con uno o más parámetros que describen un problema. Si el método detecta que los valores no representan la forma más simple del problema, se llama a sí mismo con valores de parámetros modificados que describen un subproblema cercano a esa forma. Esta actividad continúa hasta que el método detecta la forma más simple del problema, en cuyo caso el método simplemente retorna, posiblemente con un valor, si el tipo de retorno del método no es `void`. La pila de llamadas a método empieza a desbobinarse como una llamada a método anidada para ayudar a completar una evaluación de expresión. En algún punto, la llamada el método original se completa, y posiblemente se devuelve un valor.

Para entender la recursión, consideremos un método que suma todos los enteros desde 1 hasta algún límite superior:

```
static int sum (int limit) {
    int total = 0;
    for (int i = 1; i <= limit; i++)
```

```
        total += i;
    return total;
}
```

Este método es correcto porque consigue el objetivo. Después de crear una variable local `total` e inicializarla a cero, el método usa un bucle `for` para sumar repetidamente enteros a `total` desde 1 hasta el valor del parámetro `limit`. Cuando la suma se completa, `sum(int limit)` devuelve el total, mediante `return total;` a su llamador.

La recursión hace posible realizar está suma haciendo que `sum(int limit)` se llame repetidamente a sí mismo, como demuestra el siguiente fragmento de código:

```
static int sum (int limit) {
   if (limit == 1)
       return 1;
   else
       return limit + sum (limit - 1);
}
```

Para entender como funciona la recursión, considere los siguientes ejemplos:

1. `sum (1)`: El método detecta que `limit` contiene 1 y vuelve.
2. `sum (2)`: Como `limit` contiene 2, se ejecuta `return limit + sum (limit - 1);`. Lo que implica que se ejecute `return 2 + sum (1);`. La llamada a `sum (1)` devuelve 1, lo que hace que `return 2 + sum (1);` devuelva 3.
3. `sum (3)`: Como `limit` contiene 3, se ejecuta `return limit + sum (limit - 1);`. Esto implica que se ejecute `return 3 + sum (2);`. La llamada a `sum (2)` ejecuta `return 2 + sum (1);`. Luego, `sum (1)` devuelve 1, lo que hace que `sum (2)` devuelva 3, y al final `return 3 + sum (2);` devuelve 6.

Cuidado:

Asegúrese siempre que un método recursivo tiene una condición de parada (como `if (limit == 1) return 1;`). Por el contrario, la recursión continuará hasta que se sobrecargue la pila de llamadas a métodos.