

Diccionarios con Árboles

Hasta ahora hemos visto 3 implementaciones de la clase diccionario:

- con un arreglo desordenado
- con un arreglo ordenado
- usando una lista enlazada

La primera era ineficiente en memoria y tiempo (salvo para agregar definiciones). La segunda era eficiente para buscar definiciones, pero ineficiente en todo el resto, y la última, era eficiente en memoria pero lenta en búsqueda y eliminación.

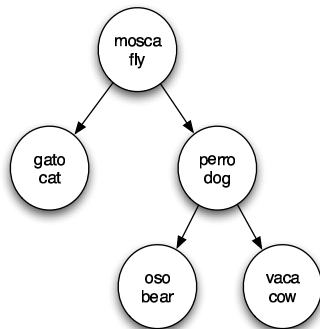
1 Árboles

Existe una implementación que es tan eficiente en memoria como las listas, pero que además es eficiente en tiempo (para todas las operaciones del diccionario!). Se trata de los árboles de búsqueda binaria.

Al igual que las listas, se construye en base a nodos, con la diferencia que los nodos tienen dos hijos en vez de un nodo "siguiente".

```
class Nodo {  
    String llave, valor;  
    Nodo izq, der;  
}
```

Estos nodos se combinan como se muestra en la siguiente figura.

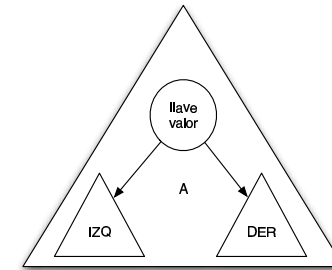


Este árbol tiene dos características:

1. Cada nodo tiene hasta 2 hijos, por lo tanto se trata de un árbol binario

2. La llave en un nodo es lexicográficamente menor que todas las llaves de su subárbol izquierdo y mayor que todas las llaves de su subárbol derecho. Esta es una propiedad de los árboles de búsqueda.

La naturaleza de los árboles es recursiva, como se muestra en el siguiente esquema:



Es decir, un árbol es o un árbol vacío, o un nodo con un subárbol izquierdo y uno derecho.

2 La clase Diccionario usando árboles

La clase diccionario tiene como única variable de instancia una referencia al nodo superior del árbol (la raíz). Inicialmente esta referencia es null (árbol vacío):

```
class Diccionario {  
    Nodo raiz;  
    Diccionario() {  
        raiz=null;  
    }  
}
```

3 El método get

Como los árboles son estructuras recursivas, en general los algoritmos que trabajan sobre ellos también lo son.

Basándose en el esquema anterior, suponiendo que n es la referencia al nodo representado con un círculo, se tiene que el algoritmo para get es

- Si llave=n.llave=>get(llave,n) vale n.valor
- Si llave<n.llave=>get(llave,n) vale get(llave,IZQ)
- Si llave>n.llave=>get(llave,n) vale get(llave,DER)

```
String get(int llave, Nodo n) {
    if (n==null)
        return null;
    else if (llave.equals(n.llave))
        return n.valor;
    else if (llave.compareTo(n.llave)<0)
        return get(llave, n.izq);
    else
        return get(llave, n.der);
}
```

Estos métodos recursivos suelen tener un método auxiliar cuya única misión es invocarlo con la raíz:

```
String get(String llave) {
    return get(String llave, raiz);
}
```

En Java, dos métodos pueden llamarse igual, siempre que sus parámetros difieran. Por lo tanto `get(llave)` y `get(llave,nodo)` pueden coexistir.

4 El método put

Nuevamente tenemos un método `put` que invoca a un método `put` más general:

```
void put(String llave, String valor) {
    raiz=put(llave, val, raiz);
}
```

El algoritmo de `put` es

- Si el árbol está vacío, se devuelve un nuevo nodo con los valores deseados
- Si `llave<n.llave`, se devuelve el árbol original (`n`) con la definición insertada en su subárbol izquierdo
- Si `llave>n.llave`, se devuelve el árbol original (`n`) con la definición insertada en su subárbol derecho

```
Nodo put(String llave, String valor, Nodo n) {
    if (n==null)
        return new Nodo(llave, valor, null, null);
    if (llave.compareTo(n.llave)<0)
        n.izq=put(llave, valor, n.izq);
    else
        n.der=put(llave, valor, n.der);
    return n;
}
```

5 El método remove

`Remove` es considerablemente más complejo:

```
void remove(String llave) {
    raiz=remove(llave, raiz);
}
```

El algoritmo de `remove`, que devuelve el árbol (o subárbol) `n` sin la llave, es

- si la llave está en la raíz, devuelve la unión del subárbol izquierdo con el derecho
- si no, se devuelve el árbol original, con el subárbol izquierdo o derecho con la llave eliminada

```
Nodo remove(int llave, Nodo n) {
    if (n==null)
        return null;
    if (n.llave.equals(llave))
        return unir(n.izq, n.der);
    if (llave.compareTo(n.llave)<0)
        n.izq=remove(llave, n.izq);
    else
        n.der=remove(llave, n.der);
    return n;
}
```

6 Problema resuelto

Escribe el método `unir`

6.1 Solución

```
Nodo unir(Nodo izq, Nodo der) {
    if (izq==null)
        return der;
    if (der==null)
        return izq;
    Nodo centro=unir(izq.der, der.izq);
    izq.der=centro;
    der.izq=izq;
    return der;
}
```