ADSW – Tema 2 – Diccionarios

José A. Mañas

10.3.2013

Contenido

1 interface Diccionario

```
public interface Diccionario {

   // Asocia una clave a un valor.
   // El tratamiento de claves repetidas depende de la implementacion.
   public void put(Comparable clave, Object valor);

   // Devuelve el valor asociado a la clave.
   public Object get(Comparable clave);

   // Elimina la clave y el valor asociado.
   // El comportamiento si hubiera claves repetidas depende de la implementacion.
   public Object remove(Comparable clave);

   // Devuelve el numero de pares clave-valor.
   public int size();

   // Vacia el diccionario.
   // Es util para pruebas.
   void reset();
}
```

2 Implementaciones

2.1 con arrays

```
public class DiccionarioArray
      implements Diccionario {
   private Comparable[] claves;
   private Object[] valores;
   private int n;

   public DiccionarioArray(int size) {
      claves = new Comparable[size];
      valores = new Object[size];
      n = 0;
   }

   // no detecta duplicados
   public void put(Comparable clave, Object valor) {
      claves[n] = clave;
      valores[n] = valor;
      n++;
   }

   public Object get(Comparable clave) {
      for (int i = 0; i < n; i++) {
         if (claves[i].equals(clave))
            return valores[i];
```

```
        }
        return null;
    }

    public Object remove(Comparable clave) {
        for (int i = 0; i < n; i++) {
            if (claves[i].equals(clave)) {
                Object valor = valores[i];
                if (n > 1) {
                    claves[i] = claves[n - 1];
                    valores[i] = valores[n - 1];
                }
                n--;
                claves[n] = null;
                valores[n] = null;
                return valor;
            }
        }
        return null;
    }

    public int size() {
        return n;
    }

    public void reset() {
        Arrays.fill(claves, null);
        Arrays.fill(valores, null);
        n = 0;
    }
}
```

2.2 con arrays ordenados

```
public class DiccionarioArrayOrdenado
        implements Diccionario {
    private Comparable[] claves;
    private Object[] valores;
    private int n;

    public DiccionarioArrayOrdenado(int size) {
        claves = new Comparable[size];
        valores = new Object[size];
        n = 0;
    }

    /**
     * busca en el rango [a, z).
     *
     * @return la posición donde debería estar la clave buscada.
     */
```

```
// version iterativa
private int busca(Comparable clave, int a, int z) {
   while (a < z) {
      int m = (a + z) / 2;
      int cmp = claves[m].compareTo(clave);
      if (cmp == 0)
         return m;
      else if (cmp < 0)
         a = m + 1;
      else
         z = m;
   }
   return a;
}

// version recursiva
private int busca(Comparable clave, int a, int z) {
   if (a >= z)
      return a;
   int m = (a + z) / 2;
   int cmp = claves[m].compareTo(clave);
   if (cmp == 0)
      return m;
   else if (cmp < 0)
      return busca(clave, m + 1, z);
   else
      return busca(clave, a, m);
}

// detecta duplicados: reemplaza el valor
public void put(Comparable clave, Object valor) {
   int idx = busca(clave, 0, n);
   if (claves[idx] != null && claves[idx].equals(clave)) {
      valores[idx] = valor;
      return;
   }
   if (idx < n) {
      System.arraycopy(claves, idx, claves, idx + 1, n - idx);
      System.arraycopy(valores, idx, valores, idx + 1, n - idx);
   }
   claves[idx] = clave;
   valores[idx] = valor;
   n++;
}

public Object get(Comparable clave) {
   int idx = busca(clave, 0, n);
   if (claves[idx] != null && claves[idx].equals(clave))
      return valores[idx];
   else
      return null;
}
```

```
  public Object remove(Comparable clave) {
    int idx = busca(clave, 0, n);
    if (claves[idx] == null || !claves[idx].equals(clave))
      return null;
    Object valor = valores[idx];
    if (n > 1) {
      System.arraycopy(claves, idx + 1, claves, idx, n - idx - 1);
      System.arraycopy(valores, idx + 1, valores, idx, n - idx - 1);
    }
    n--;
    claves[n] = null;
    valores[n] = null;
    return valor;
  }

  public int size() {
    return n;
  }

  public void reset() {
    Arrays.fill(claves, null);
    Arrays.fill(valores, null);
    n = 0;
  }
}
```

2.3 árbol binario de búsqueda

English: Binary Search Tree (BST)

```
package dictionaries;

// arbol binario de busqueda
public class DiccionarioArbol
    implements Diccionario {
  // nodos del arbol
  private class Nodo {
    Comparable clave;
    Object valor;
    Nodo izq;
    Nodo der;
  }

  private Nodo raiz;
  private int n;      //no. de nodos en el arbol

  // detecta duplicados: reemplaza el valor
  public void put(Comparable clave, Object valor) {
    raiz = put(raiz, clave, valor);
  }
```

```
private Nodo put(Nodo nodo, Comparable clave, Object valor) {
   if (nodo == null) {
      nodo = new Nodo();
      nodo.clave = clave;
      nodo.valor = valor;
      n++;
      return nodo;
   }

   Comparable claveDelNodo = nodo.clave;
   int cmp = claveDelNodo.compareTo(clave);
   if (cmp == 0)
      nodo.valor = valor;       // reutilizamos el nodo
   else if (cmp > 0)
      nodo.izq = put(nodo.izq, clave, valor);
   else
      nodo.der = put(nodo.der, clave, valor);
   return nodo;
}

public Object get(Comparable clave) {
   return get(raiz, clave);
}

private Object get(Nodo nodo, Comparable clave) {
   if (nodo == null)
      return null;
   int cmp = nodo.clave.compareTo(clave);
   if (cmp == 0)
      return nodo.valor;
   else if (cmp > 0)
      return get(nodo.izq, clave);
   else
      return get(nodo.der, clave);
}

public Object remove(Comparable clave) {
   Object[] valor = new Object[1];
   raiz = remove(raiz, clave, valor);
   return valor[0];
}

/**
 * Necesitamos lo que devuelve para mantener la estructura del arbol.
 * El argumento auxiliar valor lo usamos para devolver el valor del nodo eliminado.
 */
private Nodo remove(Nodo nodo, Comparable clave, Object[] valor) {
   if (nodo == null)
      return null;
   int cmp = nodo.clave.compareTo(clave);
   if (cmp == 0) {
      valor[0] = nodo.valor;
```

```
            if (nodo.izq == null) {
               n--;
               nodo = nodo.der;
            } else if (nodo.der == null) {
               n--;
               nodo = nodo.izq;
            } else {
               Nodo nieto = getNieto(nodo.izq);
               nodo.clave = nieto.clave;
               nodo.valor = nieto.valor;
               nodo.izq = remove(nodo.izq, nieto.clave, new Object[1]);
            }
         } else if (cmp > 0)
            nodo.izq = remove(nodo.izq, clave, valor);
         else
            nodo.der = remove(nodo.der, clave, valor);
         return nodo;
      }

      private Nodo getNieto(Nodo nodo) {
         if (nodo.der == null)
            return nodo;
         else
            return getNieto(nodo.der);
      }

      public int size() {
         return n;
      }

      public void reset() {
         raiz = null;
         n = 0;
      }
   }
```

2.4 tabla hash con listas de desbordamiento

```
public class DiccionarioTablaHashListas
      implements Diccionario {
   private class Nodo {
      Comparable clave;
      Object valor;

      private Nodo(Comparable clave, Object valor) {
         this.clave = clave;
         this.valor = valor;
      }
   }

   private final List<Nodo>[] tabla;
   private int n;
```

```
public DiccionarioTablaHashListas(int size) {
   tabla = new List[size];
}

// no detecta duplicados
public void put(Comparable clave, Object valor) {
   int pos = Math.abs(clave.hashCode() % tabla.length);
   List<Nodo> lista = tabla[pos];
   if (lista == null) {
      lista = new ArrayList<Nodo>();
      tabla[pos] = lista;
   }
   Nodo nodo = new Nodo(clave, valor);
   lista.add(nodo);
   n++;
}

public Object get(Comparable clave) {
   int pos = Math.abs(clave.hashCode() % tabla.length);
   List<Nodo> lista = tabla[pos];
   if (lista == null)
      return null;
   for (Nodo nodo : lista) {
      if (nodo.clave.equals(clave))
         return nodo.valor;
   }
   return null;
}

public Object remove(Comparable clave) {
   int pos = Math.abs(clave.hashCode() % tabla.length);
   List<Nodo> lista = tabla[pos];
   if (lista == null)
      return null;
   int idx = -1;
   for (int i = 0; i < lista.size(); i++) {
      Nodo nodo = lista.get(i);
      if (nodo.clave.equals(clave)) {
         idx = i;
         break;
      }
   }
   if (idx < 0)
      return null;
   n--;
   Nodo nodo = lista.remove(idx);
   return nodo.valor;
}

public int size() {
   return n;
```

```
  }

  public void reset() {
    Arrays.fill(tabla, null);
    n = 0;
  }
}
```

2.5 tabla hash abierta: resolución interna de colisiones

```
public class DiccionarioTablaHashAbierta
    implements Diccionario {
  private final Item[] tabla;
  private int n;

  private static class Item {
    private Comparable clave;
    private Object valor;
    private boolean borrado;

    void set(Comparable clave, Object valor) {
      this.clave = clave;
      this.valor = valor;
      this.borrado = false;
    }

    void setBorrado() {
      this.clave = null;
      this.valor = null;
      this.borrado = true;
    }

    public Comparable getClave() {
      return clave;
    }

    public Object getValor() {
      return valor;
    }

    public boolean isBorrado() {
      return borrado;
    }
  }

  public DiccionarioTablaHashAbierta(int size) {
    tabla = new Item[size];
    reset();
  }

  public int size() {
    return n;
```

```
}

public void reset() {
   for (int i = 0; i < tabla.length; i++)
      tabla[i] = new Item();
   n = 0;
}

// int busca(clave)
// devuelve

// numero entre 0 y claves.length - 1:
// donde hay que meter el nuevo dato

// numero = -1
// cuando la tabla se ha saturado y no hay sitio

private int busca(Comparable clave) {
   int paso = getPaso(clave); // hash alternativo

   int primerHueco = -1;

   int idx = Math.abs(clave.hashCode() % tabla.length);
   for (int i = 0; i < tabla.length; i++) {

      // prueba lineal: +1, +2, +3, ...
      int prueba = (idx + i) % tabla.length;

      // prueba cuadratica: +1, +4, +9, ...
      // int prueba = (idx + i * i) % claves.length;

      // prueba segundo hash: +s, +2s, +3s, ...
      // int prueba = (idx + i * paso) % claves.length;

      if (tabla[prueba].getClave() == null) {
         if (primerHueco < 0)
            return prueba;
         else
            return primerHueco;
      }
      if (tabla[prueba].getClave().equals(clave))
         return prueba;
      if (tabla[prueba].isBorrado() && primerHueco < 0)
         primerHueco = prueba;
   }
   // tabla llena
   return -1;
}

// hash alternativo
private int getPaso(Comparable clave) {
   int constante = 29;
```

```
    int hc = clave.hashCode() + clave.toString().length();
    return constante - Math.abs(hc % constante);
  }

  // detecta duplicados: reemplaza el valor
  public void put(Comparable clave, Object valor) {
    int idx = busca(clave);
    if (idx < 0) {
      // no caben mas
      return;
    }
    // clave y valor van al sitio encontrado
    tabla[idx].set(clave, valor);
    n++;
  }

  public Object get(Comparable clave) {
    int idx = busca(clave);
    if (idx < 0)
      return null;
    if (tabla[idx].isBorrado())
      return null;
    return tabla[idx].getValor();
  }

  public Object remove(Comparable clave) {
    int idx = busca(clave);
    if (idx < 0)
      return null;
    if (tabla[idx].isBorrado())
      return null;

    Object valor = tabla[idx].getValor();
    tabla[idx].setBorrado();
    n--;
    return valor;
  }
}
```

2.6 recubrimiento de la biblioteca Map y HashMap

Esto sólo vale para hacer pruebas con la misma interface.

```
public class DiccionarioHashMap
    implements Diccionario {
  private final Map<Comparable, Object> datos;

  public DiccionarioHashMap(int max) {
    datos = new HashMap<Comparable, Object>(max);
  }

  // detecta duplicados: reemplaza el valor
```

```
  public void put(Comparable clave, Object valor) {
    datos.put(clave, valor);
  }

  public Object get(Comparable clave) {
    return datos.get(clave);
  }

  public Object remove(Comparable clave) {
    return datos.remove(clave);
  }

  public int size() {
    return datos.size();
  }

  public void reset() {
    datos.clear();
  }
}
```

3 Pruebas

3.1 Pruebas funcionales de Diccionario

```
public class DiccionarioTest {
  private Diccionario diccionario;

  @Before
  public void setUp() {
    diccionario = …

    diccionario.put("Cervantes", "Miguel de Cervantes");
    diccionario.put("America", "12 de octubre de 1492");
    diccionario.put("Revolucion", "1789");
    assertEquals(3, diccionario.size());
  }

  @Test
  public void test001() {
    assertNull(diccionario.get("kaka"));
    assertEquals("Miguel de Cervantes", diccionario.get("Cervantes"));
    assertEquals("12 de octubre de 1492", diccionario.get("America"));
    assertEquals("1789", diccionario.get("Revolucion"));
  }

  @Test
  public void test011() {
    assertEquals("Miguel de Cervantes", diccionario.remove("Cervantes"));
    assertNull(diccionario.remove("Cervantes"));
```

```
    assertEquals(2, diccionario.size());
  }

  @Test
  public void test012() {
    assertEquals("12 de octubre de 1492", diccionario.remove("America"));
    assertNull(diccionario.remove("America"));
    assertEquals(2, diccionario.size());
  }

  @Test
  public void test013() {
    assertEquals("1789", diccionario.remove("Revolucion"));
    assertNull(diccionario.remove("Revolucion"));
    assertEquals(2, diccionario.size());
  }

  @Test
  public void test015() {
    assertEquals("Miguel de Cervantes", diccionario.remove("Cervantes"));
    assertEquals("12 de octubre de 1492", diccionario.remove("America"));
    assertEquals("1789", diccionario.remove("Revolucion"));
    assertEquals(0, diccionario.size());
  }
}
```

3.2 Pruebas de tiempo de ejecución de búsqueda

```
public class BancoPruebas1 {
  public static final int CASOS = 100;
  public static final int MAX = 10000000;
  private static int[] datos;

  private static int algoritmo;

  public static void main(String[] args) {
    algoritmo = 0;  // lineal
//      algoritmo= 1;   // binario iterativo
//      algoritmo= 2;   // binario recursivo

    Random random = new Random();

    // datos de prueba
    datos = new int[MAX];
    for (int i = 0; i < MAX; i++) {
      int clave = random.nextInt(MAX);
      datos[i] = clave;
    }
    if (algoritmo > 0)
      Arrays.sort(datos);

    // forzamos al compilador a optimizar código
```

```
      for (int i = 0; i < 100; i++) {
         int clave = random.nextInt(100);
         busca(clave, 100);
      }

      // pruebas
      int[] tamanos = {
            1000, 2000, 5000,
            10000, 20000, 50000,
            100000, 200000, 500000,
            1000000, 2000000, 5000000,
            10000000,
      };
      for (int N : tamanos) {
//          long t0 = System.currentTimeMillis();
         long t0 = System.nanoTime();
         for (int i = 0; i < CASOS; i++) {
            int clave = random.nextInt(N);
            busca(clave, N);
         }
//          long t2 = System.currentTimeMillis();
         long t2 = System.nanoTime();
//          System.out.printf("N= %,10d: %,6dms%n", N, (t2 - t0));
         System.out.printf("%,10d: %,12d%n", N, (t2 - t0));
//          System.out.printf("%,10d: %d%n", N, (t2 - t0));
      }
   }

   private static int busca(int clave, int N) {
      if (algoritmo == 0)
         return busquedaLineal(clave, 0, N);
      else if (algoritmo == 1)
         return busquedaBinariaIterativa(clave, 0, N);
      else
         return busquedaBinariaRecursiva(clave, 0, N);
   }

   // busca en el rango [a, z)
   private static int busquedaLineal(int clave, int a, int z) {
      for (int i = a; i < z; i++) {
         int dato = datos[i];
         if (dato == clave)
            return i;
      }
      return -1;
   }

   // busca en el rango [a, z)
   private static int busquedaBinariaIterativa(int clave, int a, int z) {
      while (a < z) {
         int m = (a + z) / 2;
         int dato = datos[m];
```

```
            if (dato == clave)
                return m;
            else if (dato < clave)
                a = m + 1;
            else
                z = m;
        }
        return a;
    }

    // busca en el rango [a, z)
    private static int busquedaBinariaRecursiva(int clave, int a, int z) {
        if (a >= z)
            return a;
        int m = (a + z) / 2;
        int dato = datos[m];
        if (dato == clave)
            return m;
        else if (dato < clave)
            return busquedaBinariaRecursiva(clave, m + 1, z);
        else
            return busquedaBinariaRecursiva(clave, a, m);
    }
}
```

3.3 pruebas hash con diferentes tamaños

```
public class BancoPruebasHash1 {
    public static final int CASOS = 100;

    public static final double CARGA = 0.6;

    private static int algoritmo = 0;  // TablaHashListas
    private static int algoritmo = 1;  // TablaHashAbierto
    private static int algoritmo = 2;  // HashMap
    private static int algoritmo = 3;  // TreeMap

    private static Random random = new Random();

    public static void main(String[] args) {
        // datos de prueba
        Diccionario diccionario = crea(1000);
        // forzamos al compilador a optimizar código
        for (int i = 0; i < 1000; i++) {
            int clave = random.nextInt(100);
            diccionario.get(clave);
        }

        // pruebas
        int[] tamanos = {
              1000, 2000, 5000,
              10000, 20000, 50000,
```

```
            100000, 200000, 500000
        };
        for (int N : tamanos) {
            diccionario = crea(N);
//          long t0 = System.currentTimeMillis();
            long t0 = System.nanoTime();
            for (int i = 0; i < CASOS; i++) {
                int clave = random.nextInt(N);
                diccionario.get(clave);
            }
//          long t2 = System.currentTimeMillis();
            long t2 = System.nanoTime();
//          System.out.printf("N= %,10d: %,6dms%n", N, (t2 - t0));
//          System.out.printf("%,10d: %,12d%n", N, (t2 - t0));
            System.out.printf("%,10d: %d%n", N, (t2 - t0));
        }
    }

    private static Diccionario crea(int total) {
        Diccionario diccionario;
        if (algoritmo == 0)
            diccionario = new DiccionarioTablaHashListas(total);
        else if (algoritmo == 1)
            diccionario = new DiccionarioTablaHashAbierta(total);
        else if (algoritmo == 2)
            diccionario = new DiccionarioHashMap(total);
        else
            diccionario = new DiccionarioTreeMap();

        int ocupado = (int) (total * CARGA);
        for (int i = 0; i < ocupado; i++) {
            int clave = random.nextInt();
            diccionario.put(clave, "valor");
        }

        return diccionario;
    }
}
```

3.4 pruebas hash con diferente carga

```
public class BancoPruebasHash2 {
    public static final int CASOS = 100;

    public static final int N = 5000;

    private static int algoritmo = 0;  // TablaHashListas
    private static int algoritmo = 1;  // TablaHashAbierto

    private static Random random = new Random();

    public static void main(String[] args) {
```

```
      // datos de prueba
      Diccionario diccionario = crea(0.5);
      // forzamos al compilador a optimizar código
      for (int i = 0; i < 1000; i++) {
         int clave = random.nextInt(100);
         diccionario.get(clave);
      }

      // pruebas
      double[] cargas = {
            0.5, 0.6, 0.7, 0.8, 0.9,
            0.95, 0.96, 0.97, 0.98, 0.99,
            1.0, 2.0, 5.0, 10.0
      };
      for (double CARGA : cargas) {
         diccionario = crea(CARGA);
//          long t0 = System.currentTimeMillis();
         long t0 = System.nanoTime();
         for (int i = 0; i < CASOS; i++) {
            int clave = random.nextInt(N);
            diccionario.get(clave);
         }
//          long t2 = System.currentTimeMillis();
         long t2 = System.nanoTime();
         System.out.printf("%,6.2f: %8d%n", CARGA, (t2 - t0));
      }
   }

   private static Diccionario crea(double CARGA) {
      Diccionario diccionario;
      if (algoritmo == 0)
         diccionario = new DiccionarioTablaHashListas(N);
      else
         diccionario = new DiccionarioTablaHashAbierta(N);

      int ocupado = (int) (N * CARGA);
      for (int i = 0; i < ocupado; i++) {
         int clave = random.nextInt();
         diccionario.put(clave, "valor");
      }

      return diccionario;
   }
}
```