Hashing

Hashing

- **Hashing**. Técnica para mapear de manera eficiente elementos de datos a índices en un arreglo para que puedan agregarse, eliminarse y buscarse rápidamente.
- Hashing es la técnica que se utiliza para implementar las clases HashSet y HashMap.
- Hashing hace posible tener una colección que puede agregar, eliminar y buscar datos en tiempo constante (también llamado O(1)).

TDA IntSet

- Suponer que se desea implementar el tipo de datos abstractos IntSet para representar conjuntos de números enteros no negativos.
- Las operaciones sobre el TDA son add (agregar un número), remove (borrar un número) y contains (revisa si un número está o no).
- Siguiendo el ejemplo del Java Collections Framework, el TDA se define mediante una interface.

TDA IntSet

```
public interface IntSet {
    void add(int value);
    boolean contains(int value);
    void clear();
    boolean isEmpty();
    void remove(int value);
    int size();
}
```

Implementación

- En los conjuntos no importa el orden en el que los números se guardan.
- Una forma de implementar el TDA es mediante un arreglo y un campo size.

Implementación con arreglos

```
public class ArrayIntSet implements IntSet {
    private int[] elementData;
    private int size;
    // Construye un conjunto vacío
    public ArrayIntSet()
    {
        elementData = new int[10];
        size = 0;
    }
```

Implementación con arreglos

```
public void add(int value)      // O(1)
{
    elementData[size++] = value;
}
public void clear()             // O(1)
{
    size = 0;
}
```

Implementación con arreglos

```
public boolean contains(int value)          // O(N)
{
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value)
            return true;
    }
    return false;
}
```

Implementación con arreglos

```
public boolean isEmpty()      // O(1)
{
    return size == 0;
}
public int size()             // O(1)
{
    return size;
}
```

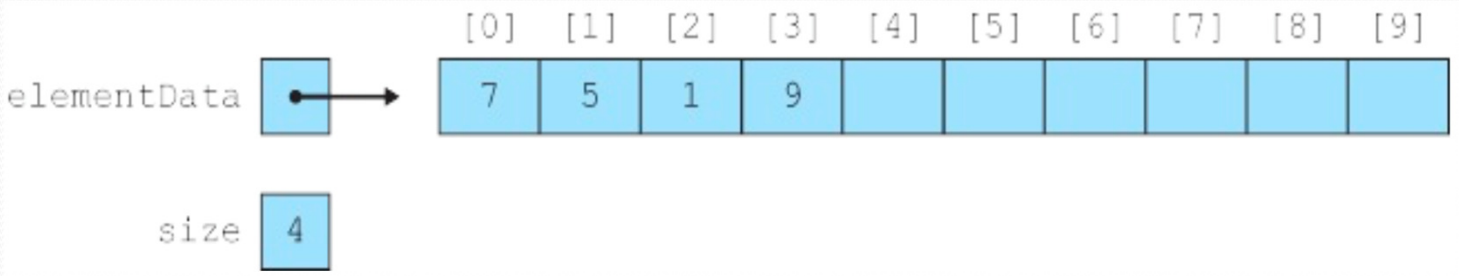
Implementación con arreglos

```
public void remove(int value)              // O(N)
{
    int index = 0;
    for (; index < size; index++) {
        if (elementData[index] == value)
            break;
    }
    if (index < size) {
        elementData[index] = elementData[size – 1]; size--;
    }
}
```

Implementación con arreglos

- Cada vez que se agrega un número se agrega al final.
- Suponer el siguiente código:

```
IntSet set = new ArrayIntSet();
set.add(7);
set.add(5);
set.add(1);
set.add(9);
```



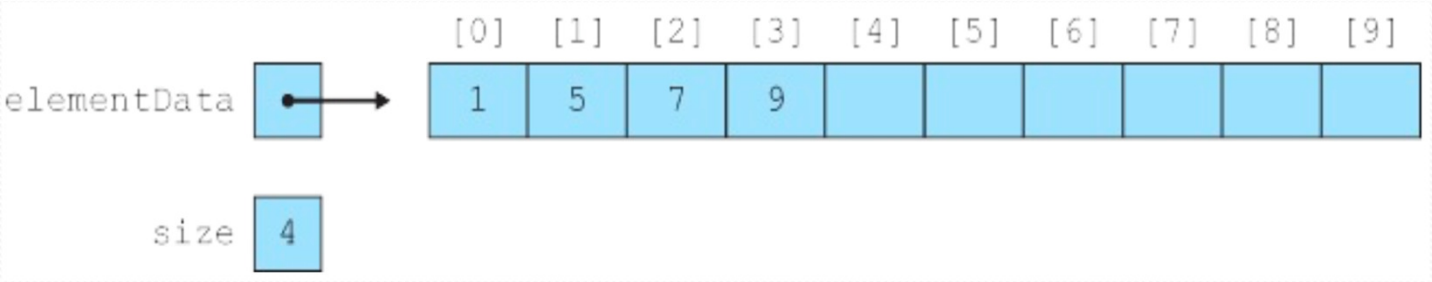
- Las casillas en blanco tienen un 0 (cero).

Eficiencia

- Agregar es $O(1)$.
- El elemento se agrega al final y $size$ se incrementa.
- Buscar es $O(N)$.
- En promedio hay que buscar en la mitad del arreglo por un elemento que si está.
- En el peor caso, si el elemento no está, se recorre todo el arreglo.
- Borrar es $O(N)$.
- Hay que buscar el elemento antes de borrarlo.
- El borrado es fácil, se reemplaza el elemento borrado por el último.

Eficiencia

- ¿Se puede mejorar esa eficiencia?
- Algo, si el arreglo se mantiene ordenado.



Eficiencia

- Ahora la búsqueda es $O(\log N)$.
- Es una búsqueda binaria. En cada iteración se descarta la mitad del arreglo.
- Pero ahora la inserción es $O(N)$.
- Se hace en dos pasos:
  - Buscar el índice correcto para mantener el orden.
  - Recorrer los elementos que son mayores a la derecha.

Eficiencia

- El borrado es O(N).
- Se hace en dos pasos:
  - Buscar el elemento.
  - Recorrer los elementos mayores a la izquierda.
- ¿Se puede mejorar esta eficiencia?
- Si.

Hashing

- Almacenar el valor k en el índice k.

- Agregar es O(1).

```
elementData[k] = k; size++;
```

- Buscar es O(1).

```
esta = elementData[k] == k ? true : false;
```

- Borrar es O(1).

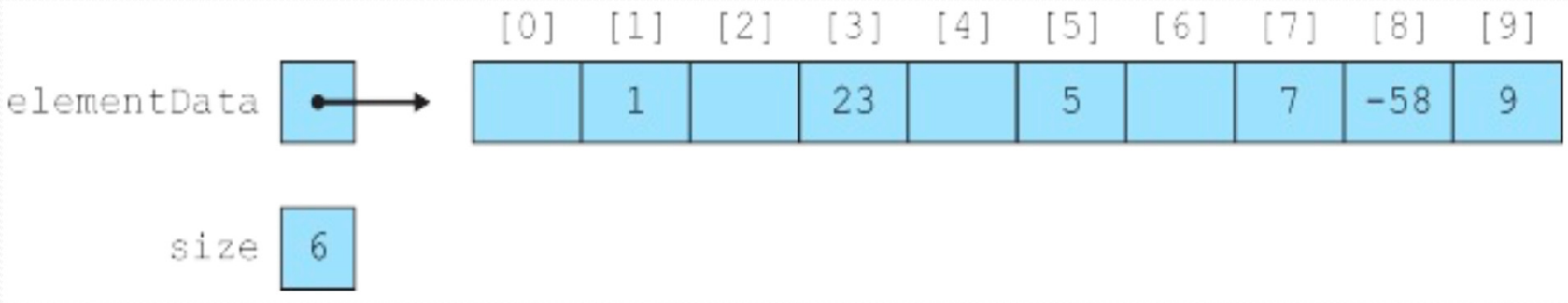
```
elementData[k] = 0; size--;
```

Consideraciones

- ¿Qué pasa si se inserta un número mayor que 9?
  - Agrandar el arreglo.
  - Aplicar la operación módulo. Por ejemplo, el 23 se guarda en el índice 23 % 10 = 3.

Consideraciones

- ¿Qué pasa si se desea extender el TDA a números negativos?
  - Usar el valor absoluto. Por ejemplo, el -58 se guarda en abs(-58) % 10 = 8.



Función hash

- Ahora se necesita un método que reciba un valor y regrese el índice correcto.
- A ese método se le llama **función hash**.
- Al arreglo que utiliza una función hash para gobernar la inserción y el borrado de sus elementos se le llama **tabla hash**.
- Hasta ahora, esta es la función de hash de ArrayIntSet:

```
private int hashFunction(int value)
{
    return Math.abs(value) % elementData.length;
}
```

Colisiones

- **Colisión**. Cuando la función de hash mapea dos valores al mismo índice.
- Por ejemplo:

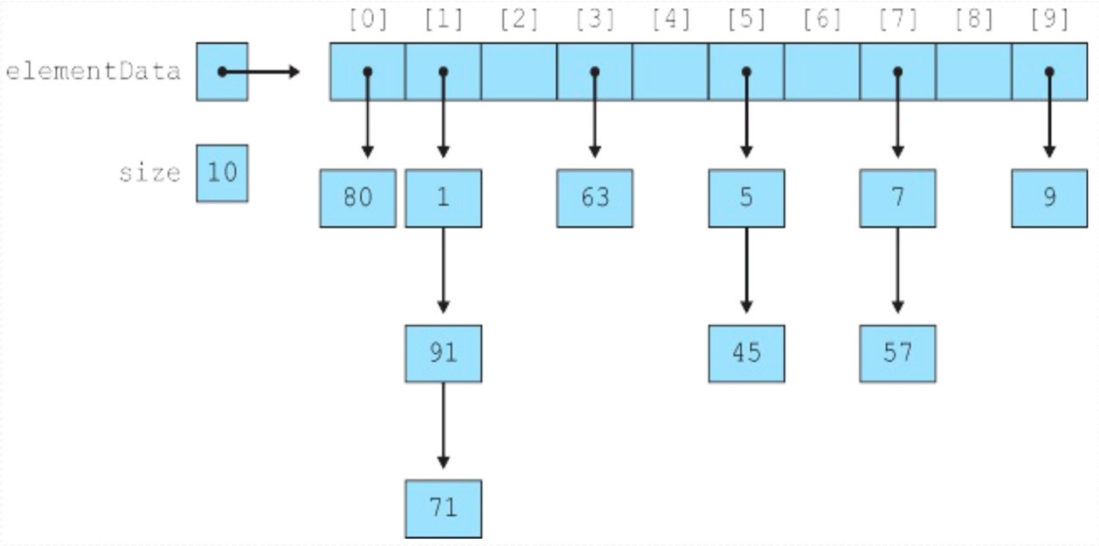
$$\text{hashFunction}(5) = 5$$

$$\text{hashFunction}(45) = 5$$

- Métodos para resolver colisiones:
- Hashing cerrado. Se usa algún tipo de *probing*.
- Hashing abierto. La tabla de hash es un arreglo de *listas ligadas*.
- Los detalles se estudian en Estructuras de Datos.

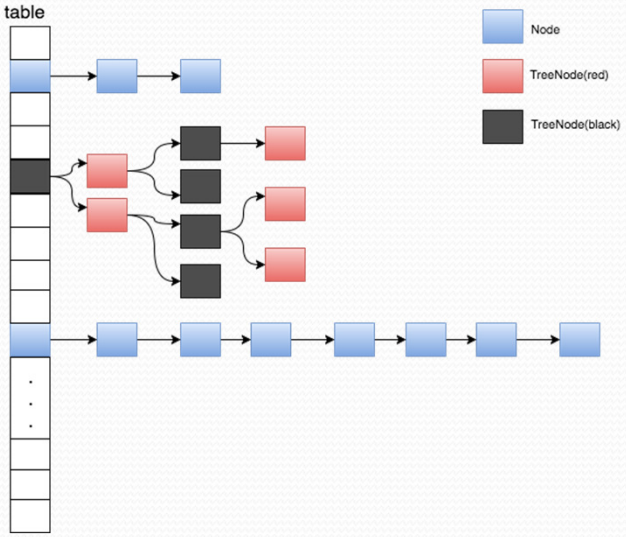
Colisiones en Java

- HashSet usa un HashMap para guardar sus elementos.
- HashMap utiliza una tabla de hash con, inicialmente, listas ligadas para resolver colisiones.



Colisiones en Java

- Si una lista tiene 8 elementos y se le agrega otro elemento se convierte en un árbol.



Fuente: https://medium.com/@harmeetkaur2793/hashmap-interenal-working-362c0ba7d854

Hashing de objetos

- La clase Object tiene un método llamado hashCode.
- hashCode regresa un entero en base a la dirección del objeto.
- No toma en cuenta el estado del objeto.
- Por esta razón muchas clases sobreponen hashCode.
- Ejemplos.

Clase Integer

```
public int hashCode()
{
    return Integer.hashCode(value);
}
public static int hashCode(int value)
{
    return value;
}
```

Clase Double

```
public int hashCode()
{
    return Double.hashCode(value);
}
public static int hashCode(float value)
{
    long bits = doubleToLongBits(value);
    return (int) (bits ^ (bits >>> 32));
}
```

Clase String

```
public int hashCode() {
    int h = hash;
    if (h == 0 && !hashIsZero) {
        h = isLatin1() ? StringLatin1.hashCode(value) : StringUTF16.hashCode(value);
        if (h == 0) {
            hashIsZero = true;
        } else {
            hash = h;
        }
    }
    return h;
}
```

Clase StringLatin1

```
public static int hashCode(byte[] value)
{
    int h = 0;
    for (byte v : value) {
        h = 31 * h + (v & 0xff);
    }
    return h;
}
```

Clase Point

- No define un hashCode propio, hereda el de java.awt.geom.Point2D.

```
public int hashCode()
{
    long bits = java.lang.Double.doubleToLongBits(getX());
    bits ^= java.lang.Double.doubleToLongBits(getY()) * 31;
    return (((int) bits) ^ ((int) (bits >> 32)));
}
```

Implementando hashCode

- Uno puede escribir su propio método hashCode en las clases que escribe.
- Todas las clases vienen con una versión predeterminada basada en la dirección de memoria.
- La versión sobrepuesta debe incluir de alguna manera el estado del objeto.

Buen comportamiento de hashCode

- *Consistente consigo mismo* (regresar el mismo valor en cada llamada):
- obj.hashCode() == obj.hashCode() si el estado no ha cambiado.
- *Consistente con la igualdad*:
- a.equals(b) debe implicar que a.hashCode() == b.hashCode().
- !a.equals(b) no implica que a.hashCode() != b.hashCode(), pero es deseable.
- *Buena distribución de códigos hash*:
- Para un conjunto grande de objetos con estados distintos, debe regresar códigos hash únicos.

Ejemplo

```
public class Point3D {
    private int x, y, z;
    ...
    public int hashCode() {
        // mejor que simplemente devolver (x + y + z)
        // distribuir los números, menos colisiones
        return 293 * x + 137 * y + 23 * z;
    }
}
```

Clase Arrays

```
public static int hashCode(Object a[]) {
    if (a == null)
        return 0;
    int result = 1;
    for (Object element : a)
        result = 31 * result + (element == null ? 0 : element.hashCode());
    return result;
}
```

Ejemplo revisitado

```
public class Point3D {
    private int x, y, z;
    ...
    public int hashCode() {
        int result = x;
        result = 31 * result + y;
        result = 31 * result + z;
        return result;
    }
}
```

Recomendaciones

- Si uno de los campos es un objeto, invocar su hashcode:

```
public int hashCode() {   // Estudiante
    return 531 * apellido.hashCode() + ...;
```

- Para incorporar un campo double o boolean, llamar al hashcode de las clases correspondientes Double y Boolean:

```
public int hashCode() {    // Cuenta de banco
    return 31337 * Double.valueOf(saldo).hashCode() +
            Boolean.valueOf(esCuentaCheques).hashCode();
```

Recomendaciones

- Java incluye el método Objects.hash(…) que recibe cualquier número de argumentos y los combina en un solo código hash:

```
public int hashCode()              // Point3D
{
    return Objects.hash(x, y, z);
}
```

- Objects.hash(…) invoca a su vez a Arrays.hashCode(Object[] a).

Reglas de una buena función hash

1. Simple.
2. Minimizar colisiones.
3. Distribución uniforme.
4. Considerar todos los bits de la clave.

Referencia

The Art of Computer Programming, Volume 3, Second Edition

Donald E. Knuth

Addison–Wesley, 1998