

Tema 9- La Tabla de Dispersión (*Hash*)

- ❑ Duración: 2 semanas aprox.
- ❑ Índice general:
 1. Implementación de las operaciones de un Diccionario en tiempo constante: ideas básicas
 2. Función de dispersión (*hashing*): características. El método hashCode de Object
 3. Resolución de Colisiones: métodos de Exploración Lineal y Cuadrática
 4. Implementación en Java de una Tabla Hash con Exploración Cuadrática: las clases EntradaTablaHash y TablaHashCuadratica

1

❑ Objetivos:

- ✓ Presentar la Tabla Hash como una Representación eficiente de la EDA Diccionario. En concreto, se estudiarán:
 - Los conceptos relacionados con su definición: función de dispersión (*hashing*), Conflictos y su Resolución mediante los Métodos de Exploración Lineal y Cuadrático
 - El Análisis de su Eficiencia, medida como eMC, en función del Método de Exploración que emplea y su Factor de Carga
 - Las clases que intervienen en su implementación en Java: EntradaTablaHash y TablaHashCuadratica
- ✓ Estudiar las ventajas e inconvenientes que supone representar un Diccionario mediante una Tabla Hash o un ABB, lo que se ejemplificará implementando el interfaz Java Diccionario tanto con la clase TablaHashCuadratica como ABBDiccionario

2

❑ Bibliografía básica:

- ✓ Weiss, M.A. **Estructuras de datos en Java**. Addison-Wesley, 2000.
 - Capítulo 6, apartado 7 para la introducción de la Tabla de Dispersión
 - Capítulo 19 para la implementación en Java de la Tabla de Dispersión

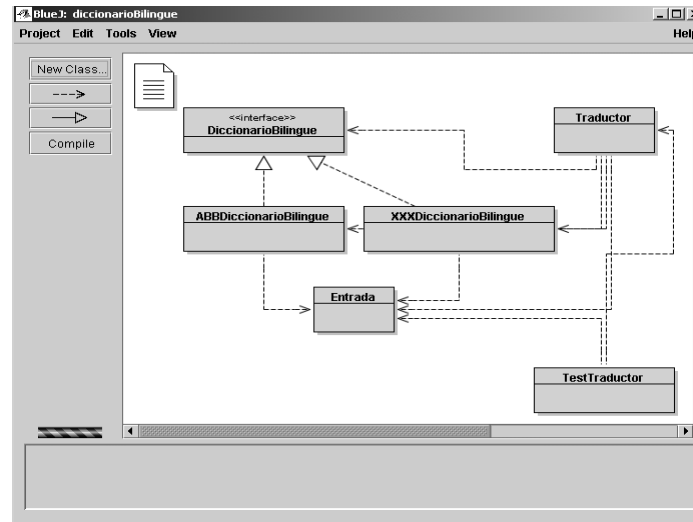
3

Aplicaciones Orientadas a la Búsqueda: Modelo Diccionario

- ✓ La operación básica del Diccionario es la Búsqueda Dinámica Por Nombre o Clave en una Colección de Entrada's
- ✓ Una Implementación del Diccionario mediante un ABB Equilibrado (ABBDiccionario) consigue que sus operaciones se ejecuten en un tiempo logarítmico con su número de Entrada's
- ✓ Una Implementación del Diccionario ¿adecuada? consigue que sus operaciones se ejecuten en un tiempo independiente de su número de Entrada's

4

Traductor palabra a palabra



La clase Entrada

```

public class Entrada implements Comparable {
    String clave, valor;
    public Entrada(String c, String v){ clave = c; valor = v;}
    public Entrada(String c){ this(c, "");}
    public String clave(){ return this.clave;}
    public String valor(){ return this.valor;}
    public int compareTo(Object x){
        return this.clave.compareTo(((Entrada)x).clave);
    }
    public String toString(){ return clave+"\t"+valor;}
}
    
```

Se quiere implementar un Diccionario de Enteros de 16 bits (entre 0 y 65.535) mediante una Representación que permita la Búsqueda Dinámica, Por Nombre y en tiempo constante

eliminar(65.535)

Tabla implements Diccionario

0	0
0	1
0	2
.	.
.	.
.	.
.	.
.	.
.	.
...	.
0	65.535

* array de tamaño máximo 65.536

* cada posición de la Tabla representa un Entero del Diccionario

* Operaciones básicas, menos inicializar(), del orden de una constante

PROBLEMAS

1. el Diccionario es de Enteros de 32 bits

!!!! la memoria es finita !!!!

2. el Diccionario **NO** es de Enteros

¿ cómo se indexan sus Entrada's ?

Función de Dispersión (*hashing*)

□ **DEFINICIÓN:** función que convierte una Entrada en un Entero (**valor *hash***) adecuado para indexar la Tabla en la que dicha Entrada se quiere almacenar (**índice *hash***)

Estándar de Java para obtener valores *hash*:

public **int** hashCode() de la clase Object

Para obtener el **valor *hash*** de una Entrada de un Diccionario se deberá rescribir hashCode() de Object en la clase Entrada

¿?

Aplicando a su Clave el método hashCode() de la clase String

La nueva clase Entrada: rescritura del método hashCode() de Object

```
public class Entrada implements Comparable {  
    String clave, valor;  
    public Entrada(String c, String v){ clave = c; valor = v;}  
    ....  
    public int compareTo(Object x){  
        return this.clave.compareTo(((Entrada)x).clave);  
    }  
    public int hashCode(){  
        int valorHash = this.clave.hashCode();  
        return valorHash;  
    }  
    public boolean equals(Object x) {  
        return this.clave.equals(((Entrada)x).clave);  
    }  
}
```

Método hashCode() de la clase Object

public int hashCode() returns a hash code **value** for the object ...

The general contract of **hashCode** is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified...
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables ...

Método hashCode() de la clase String

public int hashCode()

Returns a hash code for this string. The hash code for a **String** object is computed as

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$$

using **int** arithmetic, where **s[i]** is the *i*th character of the string, **n** is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

Overrides: hashCode in class **Object**

CONVERSIÓN String-int

- Un String **s** es un **array** de **char** (método charAt(**int** n))
- Un **char** **i** se puede codificar con nbits en una cierta base 2^{nbits} como un Entero pequeño entre 0 y $2^{nbits}-1$

$$\text{valorHash} = \text{charAt}(0)*\text{base}^{\text{length}()-1} + \dots + \text{charAt}(\text{s.length}()-1)*\text{base}^0$$

Función de Dispersión (*hashing*)

□ **DEFINICIÓN:** función que convierte una Entrada en un Entero (**valor hash**) ¿adecuado? para indexar la Tabla en la que dicha Entrada se quiere almacenar (**índice hash**)

Para obtener el **valor hash** de una Entrada de un Diccionario se deberá rescribir hashCode() de Object en la clase Entrada aplicando a su Clave el método hashCode() de la clase String

¿ valor hash == índice hash ?

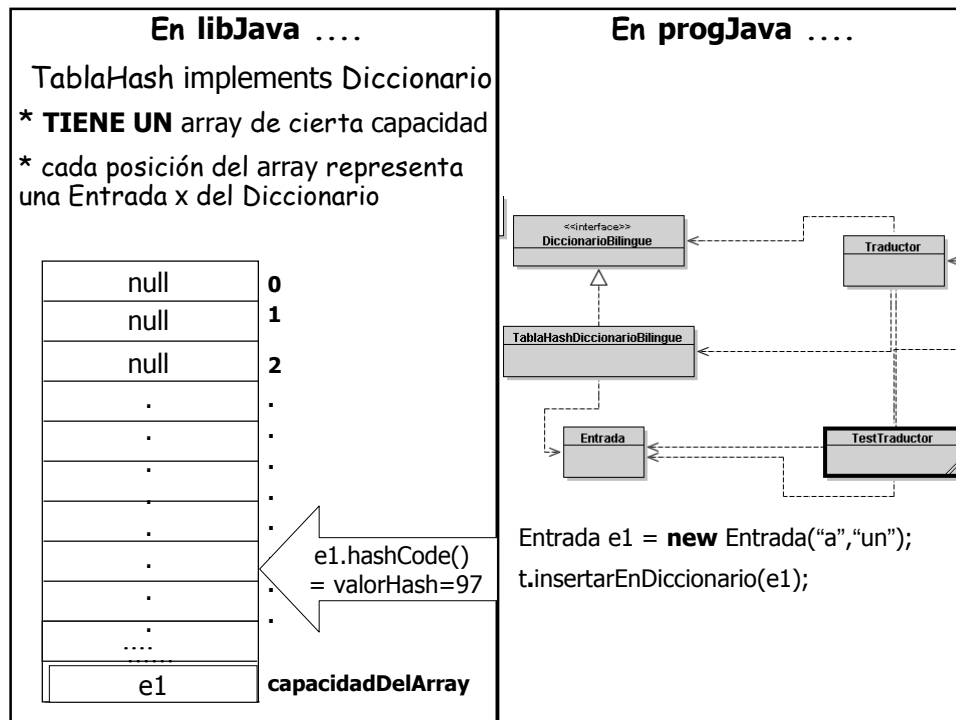
Función de Dispersión (*hashing*): de valor *hash* a índice *hash*

PROBLEMAS EN LA CONVERSIÓN String-int:

- valorHash puede ser un número muy grande
- **!!!! la memoria es finita !!!!**
- **!!!! la longitud de las Claves NO se puede reducir !!!!**

¿ **Cómo** convertir un **valor hash** (valorHash) en un **índice hash** (indiceHash), un **int** adecuado para indexar una **Tabla Hash** de capacidad dada ?

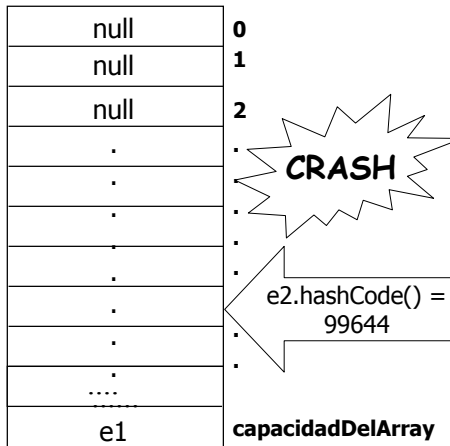
13



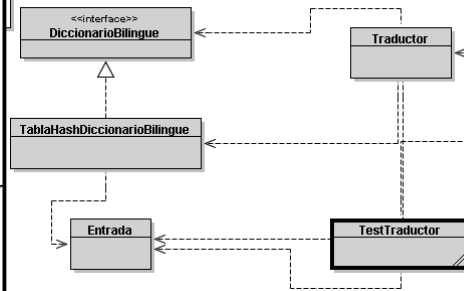
En libJava

TablaHash implements Diccionario

- * **TIENE UN** array de cierta capacidad
- * cada posición del array representa una Entrada x del Diccionario



En progJava



```
Entrada e2 = new Entrada("dog", "perro");  
t.insertarEnDiccionario(e2);
```

Función de Dispersión (*hashing*):
de valor *hash* a índice *hash*

¿ Cómo convertir un valor *hash* en un índice *hash* ? (solución al Problema **iiii** la memoria es finita !!!)

Si valorHash es un Entero no negativo arbitrario entonces
 $\text{indiceHash} = \text{valorHash} \% \text{capacidadDelArray}$ es un Entero
entre 0 y capacidadDelArray-1

```
if ( indiceHash < 0 ) indiceHash += capacidadDelArray;
```


Colisiones que provoca la función de Dispersión (*hashing*)

Fijada la función de Dispersión de una Tabla Hash, la Búsqueda Dinámica (buscar, insertar o eliminar) de una Entrada e1 en ella puede conducir a una posición ya ocupada por otra Entrada e2

$e1 \neq e2$ && $\text{indiceHashE1} == \text{indiceHashE2}$

Se resuelve **iiii Colisión !!!!** vía

- Exploración Lineal
- Exploración Cuadrática
- Otros

17

Implementación de la función de Dispersión (*hashing*)

Para realizar la Búsqueda Dinámica de una Entrada x en una Tabla Hash el **PRIMER PASO** es determinar su posición en el **array** de capacidadDelArray componentes que **TIENE** la Tabla

```
int buscarPos(Object x) {  
    /** función de hashing, que transforma x en un índice del array */  
    int valorHashX = x.hashCode();  
    int indiceHashX = valorHashX % capacidadDelArray;  
    if ( indiceHashX < 0 ) indiceHashX += capacidadDelArray;  
    /** si hay Colisión, resolverla */  
    int posicionFinalX = resolverColision(indiceHashX);  
    return posicionFinalX;  
}
```

Relación entre la calidad de la función de Dispersión, la capacidadDelArray de una Tabla Hash y la talla de un Diccionario

DEFINICIÓN: El Grado de Ocupación o Factor de Carga de una Tabla Hash es la relación entre el número de posiciones ocupadas del **array** que **TIENE** la Tabla y capacidadDelArray

$$FC = \text{ocupadasDelArray} / \text{capacidadDelArray}$$

- Para una talla dada de Diccionario, una función de Dispersión perfecta, sin Colisiones, conseguiría $FC = 1$
- Una buena función de Dispersión logra aproximarse a él

19

Relación entre la calidad de la función de Dispersión, la capacidadDelArray de una Tabla Hash y la talla de un Diccionario

Para examinar los efectos que tiene la calidad de la función de Dispersión y el Factor de Carga de una Tabla Hash sobre el número de Colisiones se propone el siguiente experimento de simulación:

Sea 109.580 el número de Entradas o talla del Diccionario representado por una Tabla Hash y sea la longitud de clave máxima 28. Se calculará el número de Colisiones que se producen

- al utilizar las 4 funciones de Dispersión descritas en la página siguiente: hashCode, Weiss, Mala y MacKenzie
- al incrementar en cada paso de la simulación capacidadDelArray en 109.580 unidades, desde 109.580 hasta 1.095.800

20

En la clase Entrada se rescribe hashCode

```
public int hashCode(){ return this.clave.hashCode();}

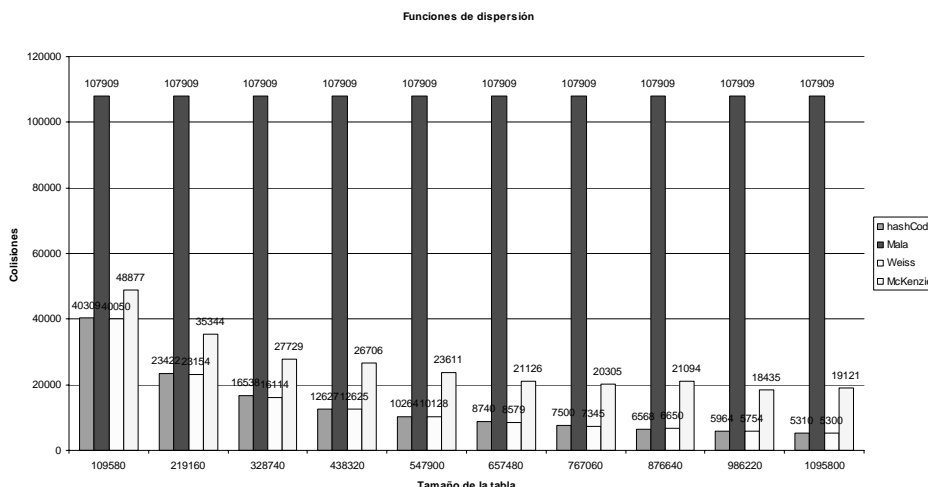
public int hashCode(){
    /** Weiss: en Weiss, capítulo 19 apartado 2, figura 19.2 */
    int valorHash = 0;
    for ( int i = 0 ; i < clave.length() ; i++ )
        valorHash = 37 * valorHash + clave.charAt(i);
    return valorHash;
}

public int hashCode(){
    /** Mala: en Weiss, capítulo 19 apartado 2, figura 19.3 */
    int valorHash = 0;
    for ( int i = 0 ; i < clave.length() ; i++ ) valorHash +=clave.charAt(i);
    return valorHash;
}

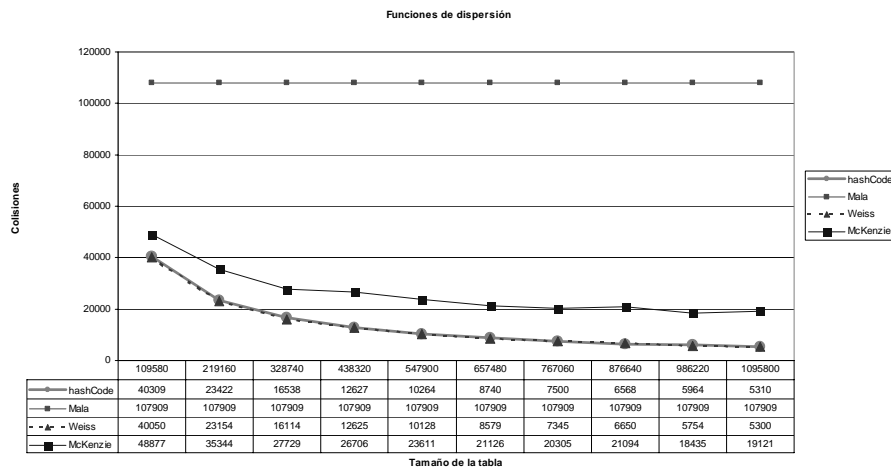
public int hashCode(){
    /** McKenzie: substituir 37 por 4 en Weiss
```

Calidad y Eficiencia se deben combinar en el diseño de la función de dispersión, PERO debe prevalecer el de Calidad

Dado un Grado de Ocupación de la Tabla, cuanto mejor es la función de dispersión menor número de colisiones provoca



- El nº de Colisiones disminuye conforme capacidadDelArray aumenta
- El nº de Colisiones disminuye conforme FC disminuye



Cuestión propuesta: ¿ Por qué en la gráfica anterior la función de dispersión Mala provoca SIEMPRE el mismo número de Colisiones, independientemente de la capacidad de la Tabla

Evaluación de una Tabla Hash

- ✓ Su rendimiento se medirá, como en un ABB, en términos del **esfuerzo Medio de Comparación**

$$eMC = \frac{\text{intentosInsercion}}{\text{inserciones}} \neq 1$$

Número Medio de Intentos

$$\text{intentosInsercion} = \sum_{i=1..n} 1 + \text{Colisiones_insercion_i}$$

- ✓ Su rendimiento dependerá, por tanto de:
 1. la calidad de su función de dispersión
 2. su Factor de Carga o Grado de Ocupación
 3. su método de Resolución de Colisiones

25

Representación de un Diccionario mediante una Tabla Hash

- una Tabla Hash TIENE UN array de Object (**elArray**) de

1. **int** capacidadDelArray componentes, que representa una estimación de la talla máxima del Diccionario particular que implemente. Inicialmente están a null pero conforme se van insertando Entradas en el Diccionario
2. **int** ocupadasDelArray, que representa el número actual de Entradas insertadas

$$FC = \text{ocupadasDelArray} / \text{capacidadDelArray}$$

- una Tabla Hash TIENE UN intentosInsercion e inserciones tal que $eMC = \text{intentosInsercion} / \text{inserciones}$

26

Representación de un Diccionario mediante una Tabla Hash

- una Tabla Hash TIENE UN array de Object (**elArray**)
- una Tabla Hash TIENE UNA capacidadDelArray
- una Tabla Hash TIENE UN ocupadasDelArray
- una Tabla Hash TIENE UN intentosInsercion e inserciones

27

Resolución de Colisiones mediante Exploración Lineal

Una Colisión se resuelve **buscando** secuencialmente a partir de indiceHashX la siguiente posición libre de la Tabla

i.e. se consultan sucesivamente las posiciones indiceHashX+1, indiceHashX+2, ... , indiceHashX+i, hasta posicionFinalX, implementando circularidad

```
int buscarPos(Object x) {  
    int valorHashX = x.hashCode();  
    int indiceHashX = valorHashX % capacidadDelArray;  
    if ( indiceHashX < 0 ) indiceHashX += capacidadDelArray;  
    int posicionFinalX = resolverColisionExplLineal(indiceHashX);  
    return posicionFinalX;  
}
```

28

Ejemplo: sea una Tabla Hash de **int** vacía, capacidadDelArray=10, función de dispersión tal que $\text{indiceHashX} = x \% 10$ y Método de Exploración Lineal. Insertar en ella 89, 18, 49, 58 y 9

insertar(89) insertar(18) insertar(49) insertar(58) insertar(9)

null	0		0		0		0		0
null	1		1		1		1		1
null	2		2		2		2		2
.	3		3		3		3		3
.	4		4		4		4		4
.	5		5		5		5		5
.	6		6		6		6		6
.	7		7		7		7		7
.	8		8		8		8		8
null	9		9		9		9		9

29

Evaluación de una Tabla Hash con Exploración Lineal

$$eMC = \frac{\text{intentosInsercion} / \text{inserciones}}{\text{Número Medio de Intentos}} \neq 1$$

- En TEORÍA, si $FC = \lambda$, la probabilidad de encontrar una celda vacía es $1-\lambda$, por lo que el valor del eMC esperado, Número Medio de Intentos independientes hasta encontrar una celda vacía es $1/(1-\lambda)$
- En la PRÁCTICA, eMC para FC elevado difiere mucho del **teórico**

Cada Intento NO es independiente del anterior ya que:

- ⇒ La secuencia de intentos realizada para buscarPos de una Clave afecta a posteriores buscarPos
- ⇒ Para resolver un conflicto muchas Claves requieren un número excesivo de intentos tras lo cual terminan añadiéndose al final de uno de los bloques existentes

Evaluación de una Tabla Hash con Exploración Lineal

Para examinar el efecto que el **Factor de Carga** ejerce sobre el comportamiento de una Tabla Hash con Exploración Lineal se propone el siguiente experimento de simulación:

Sea una Tabla Hash con $\text{capacidadDelArray} = 1000$ y función de dispersión $\text{nextInt}(\text{capacidadDelArray})$ (lo que asegura su bondad) sobre la que se insertan valores **int** entre 0 y 999. En cada paso de simulación, su **Factor de Carga** se incrementará un % por inserción de nuevas Entradas y se calculará el emC de la Tabla resultante. Específicamente, el **Factor de Carga** se incrementa como sigue:

- **En una primera fase:** un 10%, desde 0.1 hasta 0.90 \Rightarrow ocupadasDelArray se incrementa de 100 en 100
- **En una segunda fase:** un 1%, desde 0.91 hasta 0.99 \Rightarrow ocupadasDelArray se incrementa de 10 en 10

- Hasta FC del 70% el comportamiento es muy bueno (≈ 2)
- Si $\text{FC} \geq 80\%$ el comportamiento se degrada rápidamente

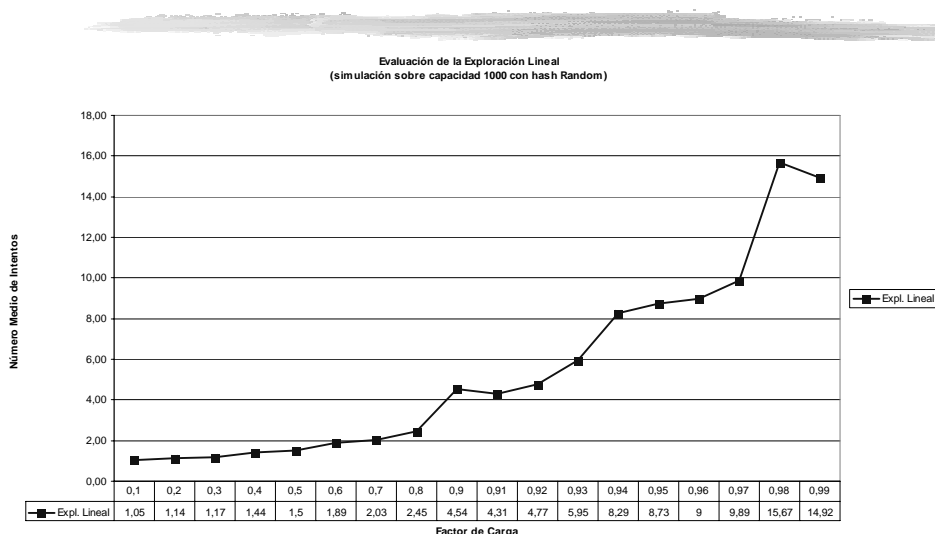


Tabla Hash con Exploración Lineal VS Árbol Binario de Búsqueda

- ✓ $eMC \approx 9$
- de un ABB Equilibrado con 1000 Nodos,
- de una Tabla Hash con Exploración Lineal y FC del 95%
- ✓ Una Tabla Hash con FC por debajo del 60% tiene un comportamiento que es cuatro veces más rápido que un ABB Equilibrado
- ✓ Si se quiere imprimir un Diccionario o realizar en él Búsquedas según un Orden, el ABB Equilibrado es más rápido

33

Ejemplo: eliminar, buscar e insertar 9 en la siguiente Tabla Hash

eliminar(9)
buscar(9)
insertar(9)

49	0
58	1
null	2
.	3
.	4
.	5
.	6
null	7
18	8
89	9

eliminar(9)
buscar(9)
insertar(9)

49
58
9
null
.
.
.
null
18
89

▪ $posicionFinalX$ es aquella tal que
 $elArray[posicionFinalX] == null$ OR
 $elArray[posicionFinalX].equals(x)$

▪ Si $elArray[posicionFinalX] == null$

1. insertar(x):

2. buscar(x) o eliminar(x):

▪ Si $elArray[posicionFinalX].equals(x)$

1. insertar(x):

2. buscar(x):

3. eliminar(x):

buscar(x), eliminar(x) e insertar(x) en la Tabla Hash del ejemplo

eliminar(89) insertar(49) buscar(49)

49	0
58	1
9	2
null	3
.	4
.	5
.	6
null	7
18	8
89	9

49	0
58	1
9	2
null	3
.	4
.	5
.	6
null	7
18	8
null	9

49	0
58	1
9	2
null	3
.	4
.	5
.	6
null	7
18	8
null	9

Operaciones de búsqueda, eliminación e inserción de Datos en una Tabla Hash

- `posicionFinalX` es aquella tal que `elArray[posicionFinalX] == null` **OR** `elArray[posicionFinalX].equals(x)`

⇒ la eliminación de `x` es **PEREZOSA**, i.e. `x` no se elimina físicamente de la Tabla sino que se marca como eliminado

⇒ componente de la Tabla == (Object dato, **boolean** activa)

EntradaTablaHash

- Si `elArray[posicionFinalX] == null`
 1. `insertar(x)`:
 2. `buscar(x)` o `eliminar(x)`:
- Si `elArray[posicionFinalX].equals(x)`
 1. `insertar(x)`:
 2. `buscar(x)`:
 3. `eliminar(x)`:

Repercusiones de la eliminación perezosa

1. una Tabla Hash TIENE UN array de EntradaTablaHash (**elArray**)
2. ocupadasDelArray **NO** representa el número de Entradas del Diccionario insertadas en la Tabla Hash que lo implementa
⇒ Añadir a su Representación **int** activasDelArray
 - a) Inicialmente ocupadasDelArray = activasDelArray = 0
 - b) Al insertar(x) NO DUPLICADO en posicionFinalX de **elArray**
activasDelArray++
SII elArray[posicionFinalX] == null ocupadasDelArray++
 - c) Al eliminar(x) de posicionFinalX de **elArray** activasDelArray--
PERO NO VARÍA ocupadasDelArray **NI** su FC
 - Con la definición y cálculo de eMC propuestos, ¿en qué momento de la manipulación de la Tabla es válido plenamente?

Resolución de Colisiones por Exploración Cuadrática

- ✓ **REHASHING** cuando FC es suficientemente alto
- ✓ Utilizar un método de Resolución de Colisiones que evite la Agrupación primaria pero con coste similar, i.e. utilizar el método de Exploración Cuadrática

Resolución de Colisiones por Exploración Cuadrática

Una colisión se resuelve consultando sucesivamente las posiciones $\text{indiceHashX} + 1^2$, $\text{indiceHashX} + 2^2, \dots, \text{indiceHashX} + i^2$, hasta posicionFinalX , implementando circularidad

```
int buscarPos(Object x) {
    int valorHashX = x.hashCode();
    int indiceHashX = valorHashX % capacidadDelArray;
    if ( indiceHashX < 0 ) indiceHashX += capacidadDelArray;
    int posicionFinalX = resolverColisionExplCuadratica(indiceHashX);
    return posicionFinalX;
}
```

39

Ejemplo: sea una Tabla Hash de **int** vacía, $\text{capacidadDelArray} = 10$, función de dispersión tal que $\text{indiceHashX} = x \% 10$ y Método de Exploración Cuadrática. Insertar en ella 89, 18, 49, 58 y 9

insertar(89)	insertar(18)	insertar(49)	insertar(58)	insertar(9)
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5
6	6	6	6	6
7	7	7	7	7
8	8	8	8	8
9	9	9	9	9

40

Evaluación de una Tabla Hash con Exploración Cuadrática

$$eMC = \text{intentosInsercion} / \text{numInserciones} \neq 1$$

- En TEORÍA,
- En la PRÁCTICA, se elimina la Agrupación Primaria PERO los elementos con el mismo valorHashX probarán las mismas celdas
 - o cada Intento NO es independiente del anterior
 - o la Exploración Cuadrática mantiene la Agrupación Secundaria

RECORDAR el gasto de memoria que supone mantener la Tabla medio vacía (menor en Java que en otros lenguajes)

41

Comparación de los métodos de Exploración Lineal y Cuadrática

Para examinar el efecto que el método de Exploración ejerce sobre el comportamiento de una Tabla Hash para distintos valores del **Factor de Carga** se propone el siguiente experimento de simulación:

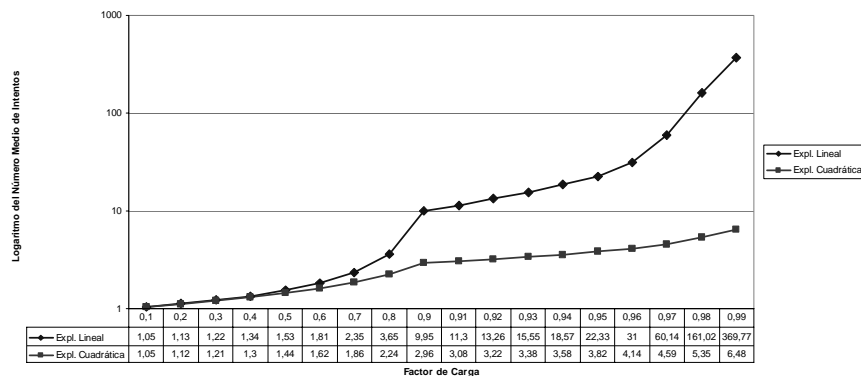
Sea una Tabla Hash con $\text{capacidadDelArray} = 110.017$ (el siguiente primo de 109.580) y función de dispersión `hashCode` de `String` sobre la que se insertan `String`. En cada paso de simulación, su **Factor de Carga** se incrementará un % por inserción de nuevas Entradas y se calculará el `emC` de la Tabla resultante.

Específicamente, el **Factor de Carga** se incrementa como sigue:

- En una **primera fase**: un 10%, desde 0.1 hasta 0.90 \Rightarrow ocupadasDelArray se incrementa desde 10.958 hasta 98.621
- En una **segunda fase**: un 1%, desde 0.91 hasta 0.99 \Rightarrow ocupadasDelArray se incrementa desde 99.717 hasta 108.484

Exploración Lineal VS Exploración Cuadrática

Exploración Lineal VS Cuadrática
(sobre distinct.txt, con hashCode())



43

Exploración Cuadrática VS Lineal

1. Provoca menos colisiones que la Lineal al eliminar la Agrupación Primaria **PERO** al buscarPos de una Clave x,

- i. ¿ se garantiza que una celda se examina sólo una vez ?
- ii. si queda sitio, ¿ se garantiza la inserción de x ?

SÍ, siempre que capacidadDelArray sea un número primo y la Tabla se encuentra medio vacía, $FC \leq 0.5$ (**REHASHING**)

2. ¿ Es más ineficiente que la Exploración lineal ?

NO, porque la Exploración Cuadrática se puede implementar sin necesidad de usar los operadores * y %

44

Cuestión propuesta: se quieren insertar como máximo 10.000 Entradas en una Tabla Hash ¿Cuál sería su capacidad óptima si se utiliza el método de Exploración Cuadrática?

45

La clase EntradaTablaHash

```
package noLineales;
class EntradaTablaHash {
    Object dato;
    boolean activa;
    EntradaTablaHash(Object dato) { this(dato, true);}
    EntradaTablaHash(Object d, boolean a){ dato = d; activa = a; }
}
```

46

La clase TablaHashCuadratica: atributos y constructora

```
package noLineales;
import modelos.*; import excepciones.*;

public class TablaHashCuadratica implements Diccionario{
    protected EntradaTablaHash elArray[];
    protected int capacidadDelArray, ocupadasDelArray, activasDelArray;
    protected int intentosInsercion, colisiones, inserciones;
    public TablaHash(int capacidad){
        capacidadDelArray = siguientePrimo(2*capacidad);
        elArray = new EntradaTablaHash[capacidadDelArray];
        for ( int i = 0; i < elArray.lenght; i++) elArray[i] = null;
        ocupadasDelArray=activasDelArray=intentosInsercion=inserciones = 0;
    }
    protected final static int          siguientePrimo(int n){...}
    protected final static boolean      esPrimo(int n){...}
}
```

La clase TablaHashCuadratica: métodos para implementar Diccionario

```
public final double eMC(){ return ((double)intentosInsercion)/inserciones;}
public final boolean esVacio() {return ( this.activasDelArray == 0 );}
protected final int buscarPos(Object x);

public final Object buscar(Object x)          throws ElementoNoEncontrado {...}
public final void eliminar(Object x)          throws ElementoNoEncontrado {...}
public final void insertar(Object x)          throws ElementoDuplicado{...}
```


La clase TablaHashCuadratica: métodos para implementar DiccionarioBilingue

```
public final int tamanyo () {...}
public final Object[] toArray() {...}
public final String toString() {
    String res = "*";
    for ( int i = 0; i < elArray.length; i++ )
        if ( elArray[i] != null && elArray[i].activa )
            res += elArray[i].dato.toString()+"\n";
    return res;
}
```

49

La clase TablaHashCuadratica: el método buscarPos

```
protected final int buscarPos(Object x){
    int indiceHashX = x.hashCode() % elArray.lenght;
    if ( indiceHashX < 0 ) indiceHashX += elArray.lenght;
    colisiones = 0;
    while ( elArray[indiceHashX] != null
        && !elArray[indiceHashX].dato.equals(x) ){
        indiceHashX += 2 * ++colisiones - 1 ;
        if ( indiceHashX >= elArray.lenght ) indiceHashX -= elArray.lenght;
    }
    return indiceHashX;
}
```

50

La clase TablaHashCuadratica: el método buscar

```
public final Object buscar(Object x) throws ElementoNoEncontrado{  
    int posicionFinalX = buscarPos(x);  
    if ( elArray[posicionFinalX] == null || !elArray[posicionFinalX].activa )  
        throw new ElementoNoEncontrado ("...");  
    return elArray[posicionFinalX].dato;  
}
```

Ejercicio propuesto: realizar las modificaciones oportunas al método buscar para obtener el método eliminar

51

La clase TablaHashCuadratica: el método insertar

```
public final void insertar(Object x) throws ElementoDuplicado {  
    int posicionFinalX = buscarPos(x);  
    intentosInsercion += 1 + colisiones; inserciones++;  
    if ( elArray[posicionFinalX] != null && elArray[posicionFinalX].activa )  
        throw new ElementoDuplicado ("...");  
    else {  
        activasDelArray++;  
        if ( elArray[posicionFinalX] == null ) ocupadasDelArray++;  
        elArray[posicionFinalX] = new EntradaTablaHash(x);  
    }  
    /** SII FC es MAYOR O IGUAL al 50%: REHASHING */  
    ....  
}
```

52

Detalles del REHASHING en inserción

```
public final void insertar(Object x) throws ElementoDuplicado {  
    /** SII FC es MAYOR O IGUAL al 50%: REHASHING */  
    if ( ocupadasDelArray >= elArray.length/2 ) {  
        EntradaTablaHash elArrayAntiguo[] = elArray;  
        elArray = new EntradaTablaHash[siguientePrimo(2*elArrayAntiguo.length)];  
        ocupadasDelArray = activasDelArray = 0;  
        inserciones = 0;  
        for ( int i = 0; i < elArrayAntiguo.length; i++ )  
            if ( elArrayAntiguo[i] != null && elArrayAntiguo[i].activa )  
                insertar(elArrayAntiguo[i].dato);  
    }  
}
```

Ejercicios propuestos

1.- Reutilizando **TablaHashCuadratica** al máximo, impleméntese **TablaHashLineal**

2.- En la práctica 6, **diccionarioBilingue**, se quiere añadir al Traductor un nuevo método **modificarEntradaDiccionario**, que dada una Entrada modifica su valor; si la Entrada no está en el Diccionario Bilingüe actual la inserta.

Diséñese este nuevo método utilizando única y exclusivamente las operaciones de **DiccionarioBilingue** ¿Cuál es su coste? ¿Cómo se podría mejorar?