

Muchas veces Uds. mencionaron la Tabla de Hashing (map) como una alternativa para solución a varios problemas.

Y la hemos usado para algunas implementaciones.

Ej: guardar los operadores con su precedencia para la generación a notación postfija

Ej: guardar variables y valores para el parser de precedencia de operadores

Cada vez que había que hacer un lookup rápido de alguna componente, surgió la idea de que una Tabla de Hashing podía ser una estrategia superadora frente a usar un Arreglo ordenado o Lista ordenada...

Analizaremos: ¿Qué ventajas concreta tiene Hashing? ¿Qué desventajas? ¿Qué tipos de Hashing existen?

Problemas

Ejemplo 1: problemas de informática o matemática

- **Representar la clase Bag**
- **Representar la clase Set**

Para cualquiera de esos 2 escenarios un hashing parece ser Buena elección.

Problemas

Ejemplo 2: para implementar programación dinámica (caché).
Usamos un hashing para los valores de precalculo

```
import java.util.HashMap;

public class FastFibo {
    private static HashMap<Integer, Long> cache= new HashMap<>();

    public static long fibo(int N) {
        if (cache.containsKey(N))
            return cache.get(N);

        long rta;

        if (N <= 1)
            rta= N;
        else
            rta= fibo(N-1) + fibo(N-2);

        cache.put(N, rta);
    }

    public static void main(String[] args) {
        System.out.println( fibo(28));
    }
}
```

Problemas

Ejemplo 3: Shazam



- Representa de las canciones sus frecuencias características y los tiempos en que ocurren => audio fingerprint
- Hashea por audio fingerprint
- Representa lo que escucha => audio fingerprint y busca en el hashing.

Tabla de Hashing

¿Cómo resulta Hashing para dar soporte las operaciones de índices?, es decir:

```
package eda;

public interface IndexParametricService <T extends Comparable<? super T>>{

    void initialize(T [] elements);

    boolean search(T key);

    void insert(T key);

    void delete(T key);

    int occurrences(T key);

    T[] range(T leftKey, T rightKey, boolean leftIncluded, boolean rightIncluded);

    void sortedPrint();

    T getMax();

    T getMin();

}
```

Hashing (Hash Table)

Si se almacenan valores en un **arreglo ordenado tradicional**, la búsqueda tiene complejidad temporal ¿...?

Observación 1: La estructura (el lugar que ocupa una componente) **no depende del orden** en que llegaron las demás componentes.

Ej: inserto 10, 20, 30, 40, 50

Ej: inserto 50, 40, 30, 20, 10

¿El 40 dónde queda?

Observación 2: Pero la posición del elemento es **fuertemente dependiente de los valores de las otras componentes**.

Ej: inserto 90, 40, 100, 120, 130 ¿El 40 dónde queda?

Por eso es que se precisa "buscar" el elemento => $O(\log N)$

Hashing (Hash Table)

Tabla de Hashing (hashing a secas) es una estructura de datos **que utiliza** un **arreglo** (lookup table) para almacenar pares key/value de una forma muy especial. No mantiene **ni contigüidad, ni orden de las componentes.**

Si Ω es el universo de los ítems que queremos almacenar y tenemos un arreglo Lookup subyacente, entonces, utiliza una función de hashing hash: $\Omega \rightarrow [0, |\text{Lookup}|-1]$ (donde $|\text{Lookup}|$ es cantidad de ranuras)

Prioriza la búsqueda, tratando de que el algoritmo tenga complejidad cercana a $O(1)$ en la búsqueda.

Hashing (Hash Table)

Primer escenario: suficiente espacio, o sea $|\Omega| \ll |\text{Lookup}|$

Sabemos que los arreglos tienen el problema de la alocación del espacio y hay que re-alocar cuando el espacio es insuficiente.

En una primera aproximación vamos a suponer que hay espacio suficiente para almacenar los pares key/value.

Formalmente

Sea Ω el conjunto de claves a hashear, y sea **LookUp** el arreglo para albergar los pares key/value.

Asumimos $|\Omega| \leq |\text{Lookup}|$, es decir, **hay posibilidad** de que a cada key se le asigne una ranura de LookUp.

Tenemos definido hash: $\Omega \rightarrow [0, |\text{Lookup}|-1]$

Idealmente encontraríamos al valor asociado a un determinado key en $\text{LookUp}[\text{hash(key)}]$

Como los keys que proporciona el usuario son tipos opacos (TAD, Objectos) es bueno solicitarle que proporcione una función de

prehash: $\Omega \rightarrow \mathbb{N}$ (números naturales en general, no le vamos a pedir al usuario que acomode el número a la cantidad de ranuras que nosotros reservamos!).

Aclaración: Eso es lo que no hace definir Java con el método hashCode()

Entonces, el usuario define

prehash: $\Omega \rightarrow \mathbb{N}$

Y nosotros que tenemos que garantizar que **hash: $\Omega \rightarrow [0, |\text{Lookup}| - 1]$** (cantidad ranuras) porque la ranura tiene que ser válida, hacemos

hash(key) = prehash(key) % | LookUp |

Como hashing no resulta bien para operaciones típicas como :

sortedPrint(),

min()

max(),

range()

Definimos una nueva interface para índices que precisen operaciones solo lookup...

```
public interface IndexParametricService<K, V> {  
  
    // no acepta key ni data nulls=> lanza exception. Si el key está, realizar un update en el valor.  
    // Si no existia lo inserta. Si hace falta crece de a chunks  
    void insertOrUpdate(K key, V data);  
  
    // nunca nunca nunca debe tirar exception. Devuelve el valor asociado si lo encuentra o null si no está.  
    V find(K data);  
  
    // nunca nunca nunca debe tirar exception.  
    // Borra y devuelve true si el elemento estaba. Si no lo encuentra devuelve false .  
    boolean remove(K key);  
  
    // nunca nunca nunca debe tirar exception. Devuelve la cantidad de elementos presentes  
    int size();  
  
    // imprimir en cualquier orden  
    void dump();  
}
```

Ejemplo 1:

Supongamos que guardamos legajos y datos de alumnos.
La fn prehash es la identidad (debe ser rápida de calcular,
en lo posible $O(1)$).

$$\text{prehash}(n) = n$$

Usar la clase Hash (campus) y escribir la clase Test que proporcione el **prehash identidad**. Usar la clase **java.util.function.Function** o **lambda** para parametrizar una fn.

Caso de Uso Ideal

En un escenario ideal, si los pares key/value a insertar fueran (55, "Ana"), (44, "Juan"), (18, "Paula"), (19, "Lucas"), (21, "Sol") y considerando un arreglo de 10 componentes que es capaz de albergar a dichos pares, testear el código. ¿Qué se obtiene?

0	
1	(21, 'Sol')
2	
3	
4	(44, 'Juan')
5	(55, 'Ana')
6	
7	
8	(18, 'Paula')
9	(19, 'Lucas')

Caso de Uso problemático

En un escenario **no tan ideal**, si los pares key/value a insertar fueran (55, "Ana"), (29, "Victor"), (25, "Tomas"), (19, "Lucas"), (21, "Sol") re ejecutar el código. Explicar qué sucede.

0	
1	(21, 'Sol')
2	
3	
4	
5	(25, 'Tomas') Se perdió (55, 'Ana')
6	
7	
8	
9	(19, 'Lucas') Se perdió (29, 'Victor')

Es decir, aún teniendo lugar pueden haber colisiones entre elementos. Eso hay que resolverlo!

Definición "Colisión"

Se dice que 2 claves $\text{key1} \leftrightarrow \text{key2}$ **colisionan** si $\text{hash}(\text{key1}) = \text{hash}(\text{key2})$, es decir, se les asigna la misma ranura.

Definición "fn hash Perfecta"

Se dice que una **fn de hash es Perfecta** si no produce colisiones. Es decir, si $\text{key1} \leftrightarrow \text{key2} \Rightarrow \text{hash}(\text{key1}) \neq \text{hash}(\text{key2})$. Sería una fn inyectiva.

Una **fn hash perfecta** no es fácil de encontrar.

Además, se tiene un universo de claves posibles (aunque no se las precise hashear a todas) y por lo tanto, garantizar que nunca van a colisionar.... Habría que conocer mucho sobre la forma de los keys.

Sin embargo, hubo un caso que discutimos en la clase de Soundex, que fue un "hashing encubierto" con fn hash perfecta.

¿Por qué lo logramos?

Teníamos que mapear 26 letras en números. En vez de hacer "if anidados" pudimos representar un arreglo consecutivo (las letras están ordenadas!!!)

Sin embargo, hubo un caso que discutimos en la clase de Soundex, que fue un "hashing encubierto" con fn hash perfecta.

¿Por qué lo logramos?

Teníamos que mapear 26 letras en números. En vez de hacer "if anidados" podemos representar un arreglo consecutivo (las letras están ordenadas!!!)

0	1	2	3	0	1	...	2
26 Letras	Pesos fonéticos						
A, E, I, O, U, Y, W,	0 -- no se codifica						
H							
B, F, P, V	1						
C, G, J, K, Q, S, X, Z	2						
D, T	3						
L	4						
M, N	5						
R	6						

- Características:
- 1) Uso todo Ω (no hay altas y bajas. Siempre son las mismas). Conozco tamaño de Lookup.
Keys son los chars. Peso fonético el data
 - 2) Prehash(char)=
 $\text{Ascii}(\text{char}) - \text{Ascii}('A')$
No hay colisión



Objetivo: una fn de hash buena (aunque no perfecta) será aquella que minimiza la cantidad de colisiones posibles.

Más allá de todo esto, en algún momento el Lookup puede quedarse sin lugar y hay que incrementar su tamaño (de a chunks) y rehashear.

Definición Factor de Carga (current load factor)

$$| \text{Keys usadas} | / | \text{Lookup} |$$

El algoritmo de inserción provisorio (porque todavía no manejamos colisiones) consiste en ir a la ranura correspondiente y proceder según el caso:

- Si está ocupada por el mismo key => era un update. Lo actualiza
- Si está ocupada por otra key => era inserción con colisión. Esta inserción no es exitosa porque todavía no manejamos colisiones, por lo tanto, provisoriamente lanzamos excepción.
- Si está vacía => era inserción exitosa. Primero inserta allí. Luego, chequea si el factor de carga supera un cierto umbral predefinido (Load Factor Threshold) y si eso ocurre se duplica el espacio y se rehashean todas las claves.

Caso de Uso A (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- `myHash.insertOrUpdate(55, "Ana");`

ranura	Element<K,V>
0	
1	
2	
3	
4	

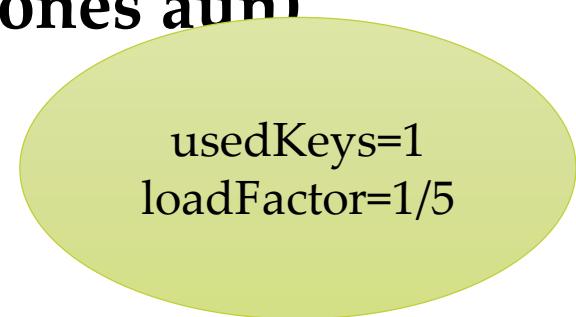
Caso de Uso A (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- `myHash.insertOrUpdate(55, "Ana");`



ranura	Element<K,V>
0	<55, "Ana">
1	
2	
3	
4	

Caso de Uso A (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(55, "Ana");
- **myHash.insertOrUpdate(29, "Victor");**

ranura	Element<K,V>
0	<55, "Ana">
1	
2	
3	
4	

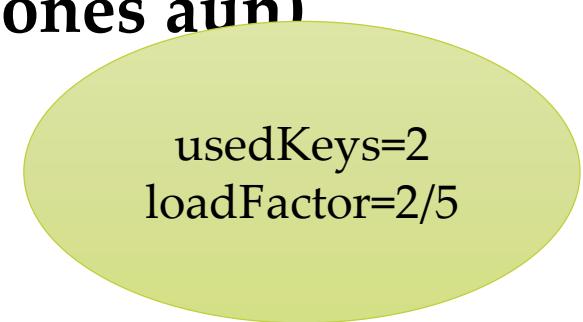
Caso de Uso A (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(55, "Ana");
- **myHash.insertOrUpdate(29, "Victor");**



ranura	Element<K,V>
0	<55, "Ana">
1	
2	
3	
4	<29, "Victor">

Caso de Uso A (sin manejo de colisiones)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(55, "Ana");
- myHash.insertOrUpdate(29, "Victor");
- **myHash.insertOrUpdate(25, "Tomas");**



ranura	Element<K,V>
0	<55, "Ana">
1	
2	
3	
4	<29, "Victor">

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- `myHash.insertOrUpdate(5, "E");`

ranura	Element<K,V>
0	
1	
2	
3	
4	

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

usedKeys=1
loadFactor=1/5

- **myHash.insertOrUpdate(5, "E");**

ranura	Element<K,V>
0	<5, "E">
1	
2	
3	
4	

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- **myHash.insertOrUpdate(4, "D");**

ranura	Element<K,V>
0	<5, "E">
1	
2	
3	
4	

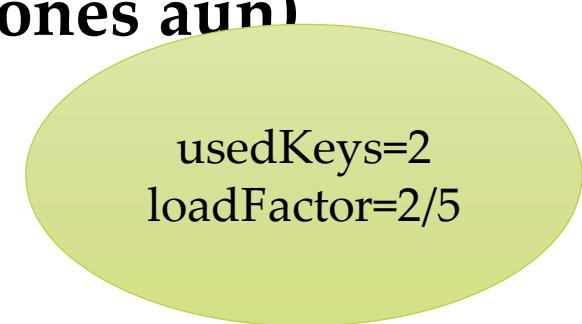
Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- **myHash.insertOrUpdate(4, "D");**



ranura	Element<K,V>
0	<5, "E">
1	
2	
3	
4	<4, "D">

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- **myHash.insertOrUpdate(1, "A");**

ranura	Element<K,V>
0	<5, "E">
1	
2	
3	
4	<4, "D">

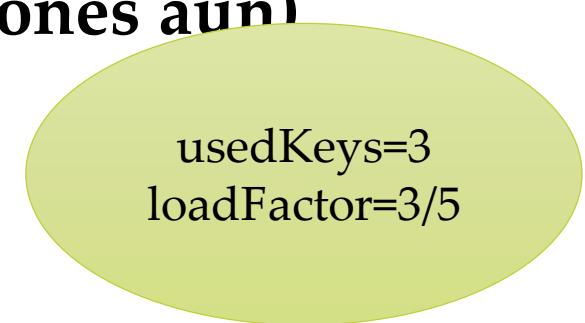
Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- **myHash.insertOrUpdate(1, "A");**



ranura	Element<K,V>
0	<5, "E">
1	<1, "A">
2	
3	
4	<4, "D">

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- myHash.insertOrUpdate(1, "A");
- **myHash.insertOrUpdate(2, "B");**

ranura	Element<K,V>
0	<5, "E">
1	<1, "A">
2	
3	
4	<4, "D">

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 5;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- myHash.insertOrUpdate(1, "A");
- **myHash.insertOrUpdate(2, "B");**

usedKeys=4
loadFactor=4/5
REHASH!!!

ranura	Element<K,V>
0	<5, "E">
1	<1, "A">
2	<2, "B">
3	
4	<4, "D">

ranura	Element<K,V>
0	<5, "E">
1	<1, "A">
2	<2, "B">
3	
4	<4, "D">



usedKeys=4
loadFactor=4/10

ranura	Element<K,V>
0	
1	<1, "A">
2	<2, "B">
3	
4	<4, "D">
5	<5, "E">
6	
7	
8	
9	

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 10;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- myHash.insertOrUpdate(1, "A");
- myHash.insertOrUpdate(2, "B");
- **myHash.insertOrUpdate(2, "otroB");**

ranura	Element<K,V>
0	
1	<1, "A">
2	<2, "B">
3	
4	<4, "D">
5	<5, "E">
6	
7	
8	
9	

NO CAMBIA
usedKeys=4
loadFactor=4/10

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 10;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- myHash.insertOrUpdate(1, "A");
- myHash.insertOrUpdate(2, "B");
- **myHash.insertOrUpdate(2, "otroB");**

ranura	Element<K,V>
0	
1	<1, "A">
2	<2, "otroB">
3	
4	<4, "D">
5	<5, "E">
6	
7	
8	
9	

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 10;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- myHash.insertOrUpdate(1, "A");
- myHash.insertOrUpdate(2, "B");
- myHash.insertOrUpdate(2, "otroB");
- **myHash.insertOrUpdate(3, "C");**

ranura	Element<K,V>
0	
1	<1, "A">
2	<2, "otroB">
3	
4	<4, "D">
5	<5, "E">
6	
7	
8	
9	

usedKeys=5
loadFactor=5/10

Caso de Uso B (sin manejo de colisiones aun)

currentLookUpSize= 10;

threshold= 0.75

La fn prehash es la identidad

- myHash.insertOrUpdate(5, "E");
- myHash.insertOrUpdate(4, "D");
- myHash.insertOrUpdate(1, "A");
- myHash.insertOrUpdate(2, "B");
- myHash.insertOrUpdate(2, "otroB");
- **myHash.insertOrUpdate(3, "C");**

ranura	Element<K,V>
0	
1	<1, "A">
2	<2, "B">
3	<3, "C">
4	<4, "D">
5	<5, "E">
6	
7	
8	
9	

TP 4- Ejer 3



Mejorar el método
insertOrUpdate según lo
explicado (sin manejo de
colisiones)

Ej: empezar probando con
factor de carga:
 $threshold=0.75$

Después probar con otros
valores.

Implementar el siguiente comportamiento:

- Si el **key existe** → **update** del valor.
- Si ya hay **otra key** en la ranura → **Exception**
- Se debe llevar la cuenta de la cantidad de ranuras ocupadas. En cada inserción se debe chequear si se supera el factor de carga si se supera → **Duplica tamaño** de la tabla y **Rehash**

TP 4- Ejer 4



Si la clave es string

¿Cual de los métodos
distribuye mejor,
considerando un archivo
con datos reales?

Bajar de campus el archivo amazon-categories30.txt (colocarlo en resources) el cual contiene un subconjunto de 30 productos que ofrece Amazon. Cada línea representa un producto con la siguiente información:

title#category#salesRank

Ej: Patterns of Preaching: A Sermon Sampler#Book#396585

Supongamos que la **key** es **title** (realmente son todos distintos).

- **Método 1:** escribir un pre-hash -> ASCII del primer elemento sobre title
- **Método 2:** escribir pre-hash -> suma de los ASCII sobre title

Para cada una de los métodos imprimir cuantos colisiones se obtuvo con el primer método y cuantas con el segundo.

La de la suma no es tan mala...

Java usa una muy parecida pero la dispersa con un número primo:

```
Function<String, Integer> fn = new Function<String, Integer>() {  
    public Integer apply(String t)  
    {  
        int sum = 0;  
        for (int rec = 0; rec < t.length(); rec++) {  
            sum = 31 * sum + t.codePointAt(rec);  
        }  
  
        return sum;  
    }  
}
```

Ayuda para leer el archivo:

```
String fileName= "amazon-categories30.txt";
InputStream is =
ClosedHashing.class.getClassLoader().getResourceAsStream(fileName);
```

```
Reader in = new InputStreamReader(is);
BufferedReader br = new BufferedReader(in);
```

```
String line;
while ((line = br.readLine()) != null) {
.....
}
```

Cuando las claves son numéricas existen varias fn posibles:

- **División ó Módulo:** $\text{hash}(X) = X \bmod m$. Donde m debe ser número primo .
- **Mid-Square:** se calcula el cuadrado de un número y se toman los bits centrales como lugar donde hashear.
Ejemplo: si $X = 14$, $X * X = 23420$ y hay que tomar los bits centrales para poder hashear. Si la tabla tuviera 11 elementos usaría el numero $34 \% 11$ (o $42 \% 11$).
- **Folding o Plegado:** se divide el numero en zonas de la misma longitud. Se las suma y se toman los bytes necesarios.
Ejemplo: si el numero es 20112241203123 y la tabla tiene 101 elementos, se arman grupos de a 3 dígitos, se obtiene $020 + 112 + 241 + 203 + 123 = 699$ y se lo hashea a $699 \% 101$.
- **Análisis del Dígito:** implica un conocimiento de antemano, de las características de la población. Se analiza los patrones de las claves, en busca de la información de la clave que menos se repite.
Ejemplo: si las claves para hashear los alumnos de una facultad en cierto año fuera su DNI, no convendría elegir sus primero dígitos porque todos los alumnos de un mismo año comienzan con las mismas cifras de DNI.

Colisiones

Existen 2 formas de resolver las colisiones:

- **Open Addressing or Closed Hashing:** dentro de la misma tabla de hashing se guardan los elementos que colisionaron
- **Open Hashing or Closed Addressing or Chaining**
=> fuera del hashing se almacenan los elementos que colisionaron.

Open Addressing or Closed Hashing

Cada ranura puede tener null (está vacío o baja física). Aunque la ranura no esté vacía puede ser que el elemento no esté, ya que hay que manejar el concepto de bajas lógicas (además de las físicas). Es decir, una ranura representa 3 estados: tiene un elemento o no tiene un elemento (dado por baja lógica o bien por baja física).

- Típica formas de resolver esa colisión:
 - Rehasheo Lineal (*linear probing*). Si hay colisión en la ranura i , entonces intentar con la ranura $i+1$, y así siguiendo hasta encontrar que el elemento (se hace update) o encontrar un lugar vacío (baja física) y se inserta allí. Con esta técnica si hay lugar lo encuentra seguro.

Ej: si cae en ranura 4 y está ocupado va a intentar ranura $4+1$, luego ranura $4+1+1$, luego ranura $4+1+1+1$, etc. Se suele tratar al arreglo como una lista circular.

- Resaheo Cuadrático (*quadratic probing*). El intervalo entre ranuras a usar, si hubiera colisión, será cuadrática. Ej: si le toca la ranura 4 y está ocupado, va intentar la ranura $4+1^2$, luego $4+2^2$, luego $4+3^2$, etc. Problema: podría haber lugar y no lo encuentra.
- Otra combinación predeterminada (determinística) de fn: siempre conviene que la última sea rehasheo lineal.

Supongamos que quiero rehasheo lineal

Open Addressing or Closed Hashing

- **Borrado:** No se puede reemplazar al lugar borrado por una ranura vacía porque la búsqueda de alguna clave puede necesitar "pasar sobre ella" si hubo colisión. Se debe manejar dos tipos de borrado: físico (realmente se elimina el elemento) y lógico (se lo marca como que no está, y si más tarde hay que insertar en esa ranura se la puede aprovechar).
 - El borrado físico se lo usa *cuando la ranura que le sigue* está también borrado físicamente
 - El borrado lógico se lo usa en caso contrario, o sea *cuando la ranura que le sigue* está ocupada o bien borrada lógicamente

Open Addressing or Closed Hashing

- **Búsqueda:** Por lo dicho en el punto anterior se comienza buscando la clave en la ranura calculada. Si el lugar está con baja física seguro que no está en otro lado. Caso contrario si está marcado como ocupado y coincide con el valor esperado, se ha encontrado!!!. Pero si el lugar está ocupado y no es el elemento buscado o bien está como baja lógica, **no se sabe si va aparecer más adelante** (en la aplicación de las sucesivas funciones de hashing). O sea que en ese caso hay que seguir buscando hasta encontrarlo (hallar una ranura ocupada que coincida con el elemento) o bien hallar una baja física.

Open Addressing or Closed Hashing

- **Inserción:** Si la ranura calculada está marcada como baja física, el elemento se inserta allí. Caso contrario (está ocupado y no es el elemento a insertar, o bien está marcado como baja lógica) hay que comenzar a navegar con las sucesivas celdas **hasta encontrar la primera baja física** (o el error porque el elemento ya existía). Atención que no se puede detenerse en la primera baja lógica y pretender insertarlo allí porque justamente puede estar más adelante. Una vez que se encuentra la primera baja física se lo puede insertar allí o en alguna de las bajas lógicas halladas en ese trayecto.

Open Addressing or Closed Hashing

Tip Importante:

Para no rebotar de más en las colisiones, lo mejor en el algoritmo de inserción es **no insertar en el primer espacio libre que se encuentra, sino en la primera baja lógica** que se encontró en el camino hasta descubrir que se podía insertar (se halló baja física).

Open Addressing or Closed Hashing

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- **myLookUp.insert(3, "Dick");**
- **myLookUp.insert(23, "Joe");**
- **myLookUp.insert(4, "Sue");**
- **myLookUp.insert(15, "Meg");**
- **myLookUp.delete(23); //Joe**
- **myLookUp.delete(15); //Meg**
- **myLookUp.insert(4, "Sue");**
- **myLookUp.insert(43, "Paul");**

ranura	Element<K,V>
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Open Addressing or Closed Hashing

usedKeys=1
loadFactor=1/10

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- **myLookUp.insert(3, "Dick");**
- **myLookUp.insert(23, "Joe");**
- **myLookUp.insert(4, "Sue");**
- **myLookUp.insert(15, "Meg");**
- **myLookUp.delete(23); //Joe**
- **myLookUp.delete(15); //Meg**
- **myLookUp.insert(4, "Sue");**
- **myLookUp.insert(43, "Paul");**

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	
5	
6	
7	
8	
9	

Open Addressing or Closed Hashing

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- **myLookUp.insert(23, "Joe");**
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	
5	
6	
7	
8	
9	

Open Addressing or Closed

usedKeys=2
loadFactor=2/10

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- **myLookUp.insert(23, "Joe");**
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> notdeleted
5	
6	
7	
8	
9	

Open Addressing or Closed Hashing

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- **myLookUp.insert(4, "Sue");**
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> notdeleted
5	
6	
7	
8	
9	

Open Addressing or Closed

usedKeys=3
loadFactor=3/10

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- **myLookUp.insert(4, "Sue");**
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> notdeleted
5	<4, "Sue"> notdeleted
6	
7	
8	
9	

Open Addressing or Closed Hashing

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- **myLookUp.insert(15, "Meg");**
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> notdeleted
5	<4, "Sue"> notdeleted
6	
7	
8	
9	

Open Addressing or Closed

usedKeys=4
loadFactor=4/10

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- **myLookUp.insert(15, "Meg");**
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> notdeleted
5	<4, "Sue"> notdeleted
6	<15, "Meg"> notdeleted
7	
8	
9	

Open Addressing or Closed Hashing

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- **myLookUp.delete(23); //Joe**
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> notdeleted
5	<4, "Sue"> notdeleted
6	<15, "Meg"> notdeleted
7	
8	
9	

Open Addressing or Closed

usedKeys=3
loadFactor=3/10

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- **myLookUp.delete(23); //Joe**
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> deleted
5	<4, "Sue"> notdeleted
6	<15, "Meg"> notdeleted
7	
8	
9	

Open Addressing or Closed Hashing

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- **myLookUp.delete(15); //Meg**
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> deleted
5	<4, "Sue"> notdeleted
6	<15, "Meg"> notdeleted
7	
8	
9	

Open Addressing or Closed

usedKeys=2
loadFactor=2/10

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- **myLookUp.delete(15); //Meg**
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> deleted
5	<4, "Sue"> notdeleted
6	
7	
8	
9	

Open Addressing or Closed

Fue update
usedKeys=2
loadFactor=2/10

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- **myLookUp.insert(4, "Sue");**
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> deleted
5	<4, "Sue"> notdeleted
6	
7	
8	
9	

Open Addressing or Closed Hashing

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- **myLookUp.insert(43, "Paul");**

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> deleted
5	<4, "Sue"> notdeleted
6	
7	
8	
9	

usedKeys=3
loadFactor=3/10

Open Addressing or Clos

Caso de Uso C:

initialLookUpSize= 10;

Threshold= 0.75

La fn pre hash el mismo key

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<43, "Paul"> notdeleted
5	<4, "Sue"> notdeleted
6	
7	
8	
9	

TP 4- Ejer 5



Ahora implementar Closed Hashing con resolucion de colisiones con rehasheo lineal (tratamiento lista circular)

Open Addressing or Closed Hashing

Linear Hashing es muy eficiente para implementar la resolución de colisiones (aprovecha la localidad de los componentes => elementos cercanos).

Si hay lugar lo encuentra seguro.

La desventaja se presenta cuando el factor de carga es alto ! Buscar qué es lo que se llama “**Primary Clustering**” (ej: Wikipedia)

Colisiones

Existen 2 formas de resolver las colisiones:

- **Open Addressing or Closed Hashing:** dentro de la misma tabla de hashing se guardan los elementos que colisionaron
- **Open Hashing or Closed Addressing or Chaining**
=> fuera del hashing se almacenan los elementos que colisionaron.

Open Hashing / Chaining / Overflow/ Closed Addressing

Las colisiones se resuelven en una estructura auxiliar (lista lineal, etc).

Cada ranura puede tener null, o bien una estructura auxiliar con las componentes que colisionaron en dicha ranura (zona overflow).

La zona de overflow se administra a demanda (no a priori).

Open Hashing / Chaining / Overflow/ Closed Addressing

- **remove:** Si la ranura está en null (sin zona de overflow habilitada), el elemento no existía. Si hay zona overflow, se lo navega para borrarlo (si estuviera). Si luego del borrado se obtiene una zona de overflow innecesaria, entonces la ranura vuelve a null.

Es una operación destructiva.

Open Hashing / Chaining / Overflow/ Closed Addressing

- **find:** Si la ranura está en null, el elemento no está. Si hay zona de overflow, se lo navega allí para ver si se lo encuentra. Operación read-only.

Open Hashing / Chaining / Overflow/ Closed Addressing

- **insertOrUpdate:** Si se le asigna una ranura vacía, se habilita la zona de overflow. La inserción se hace en la zona de overflow correspondiente, si es que el elemento no estaba, caso contrario es un update.

Es una operación destructiva.

Aclaración

En un hashing, no hay orden de los elementos.

No es razonable pedir que el “key” sea comparable para que pueda ordenarse...

No vamos a usar una lista ordenada simplemente encadenada, sino una lista. Pueden usar la que viene con Java: **LinkedList**.

Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan");
- myLookUp.insertOrUpdate(43, "Paul");

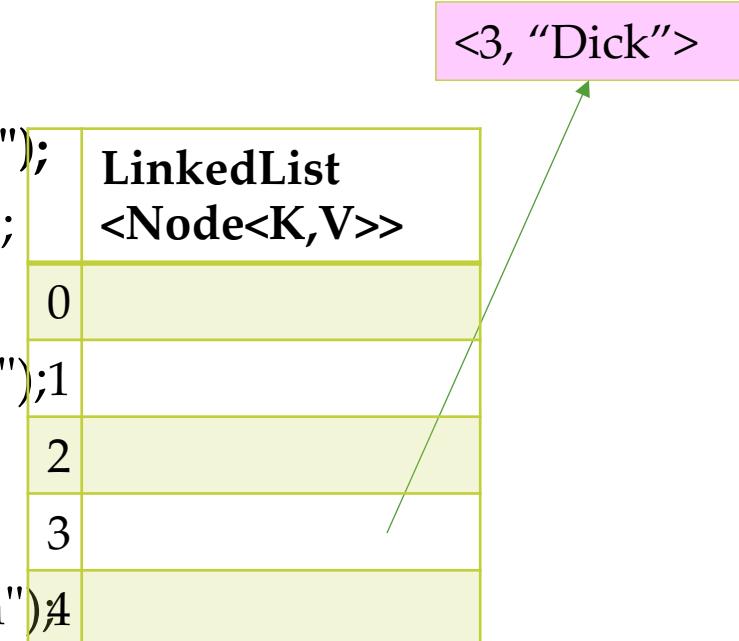
LinkedList <Node<K,V>>	
0	
1	
2	
3	
4	

Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan");
- myLookUp.insertOrUpdate(43, "Paul");

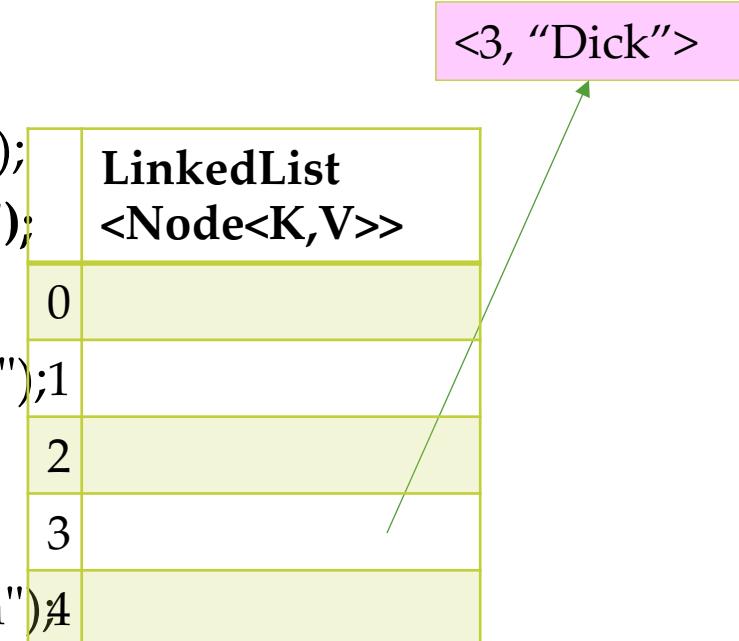


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- **myLookUp.insertOrUpdate(23, "Joe");**
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan");
- myLookUp.insertOrUpdate(43, "Paul");



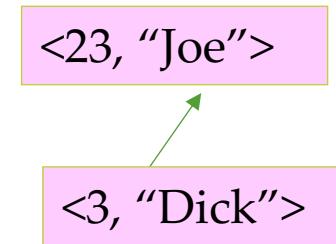
Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- **myLookUp.insertOrUpdate(23, "Joe");**
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan");
- myLookUp.insertOrUpdate(43, "Paul");

LinkedList <Node<K,V>>	
0	
1	
2	
3	
4	



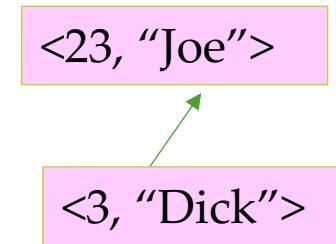
Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- **myLookUp.insertOrUpdate(4, "Sue");**
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan");
- myLookUp.insertOrUpdate(43, "Paul");

LinkedList <Node<K,V>>	
0	
1	
2	
3	
4	

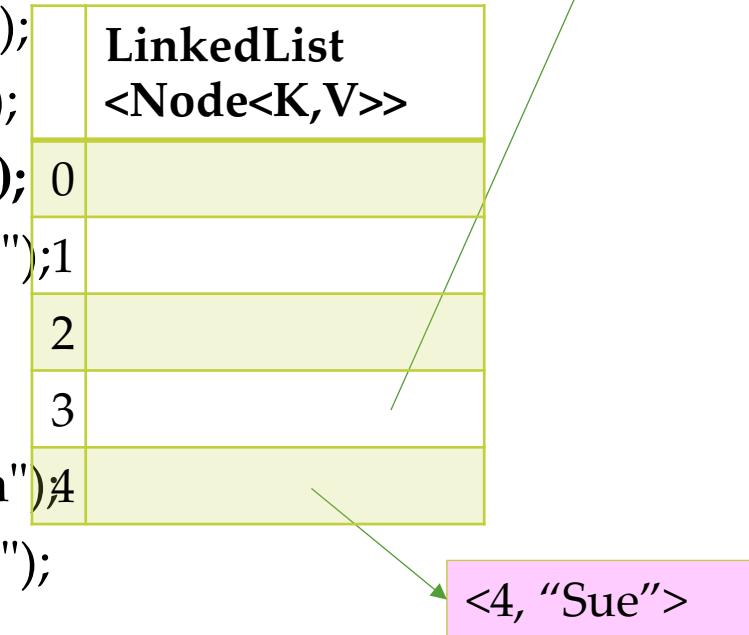


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- **myLookUp.insertOrUpdate(4, "Sue");**
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan")
- myLookUp.insertOrUpdate(43, "Paul");

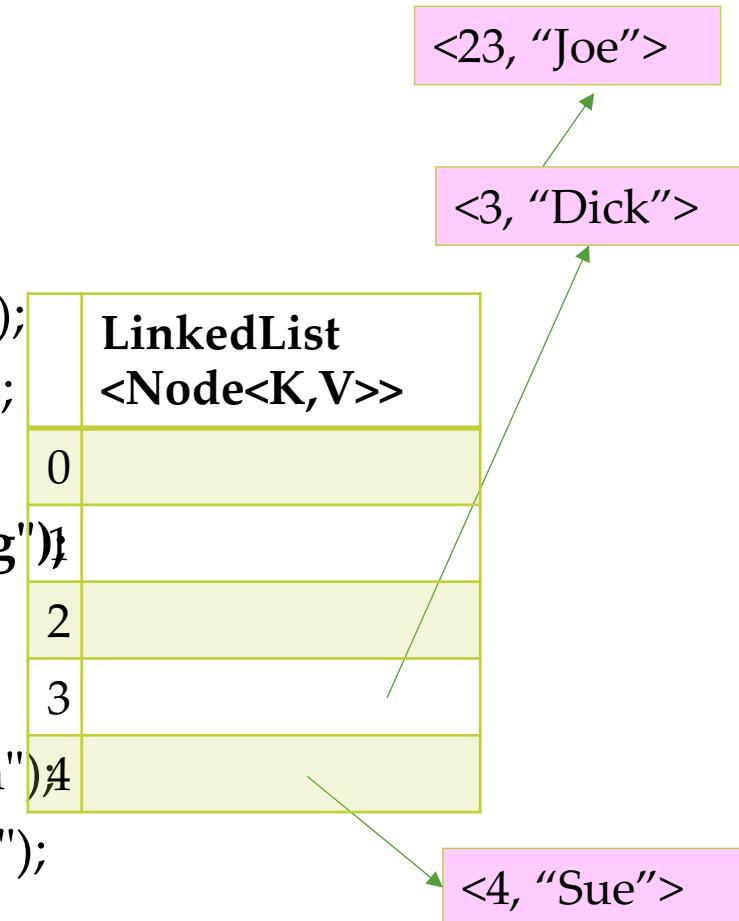


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- **myLookUp.insertOrUpdate(15, "Meg")};**
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan")};
- myLookUp.insertOrUpdate(43, "Paul");

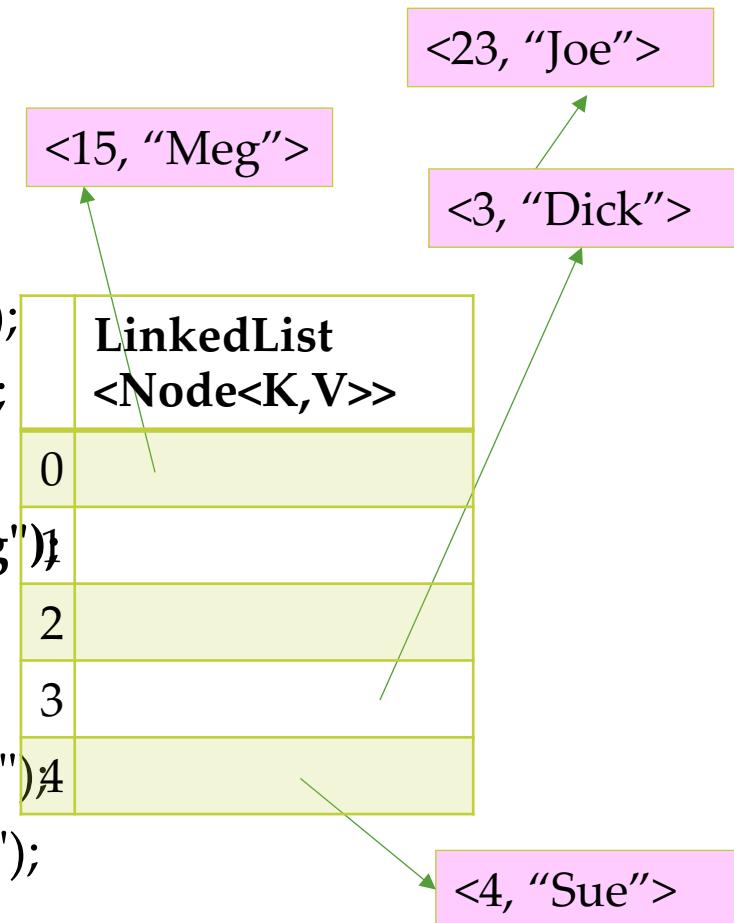


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- **myLookUp.insertOrUpdate(15, "Meg")};**
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan")};
- myLookUp.insertOrUpdate(43, "Paul");

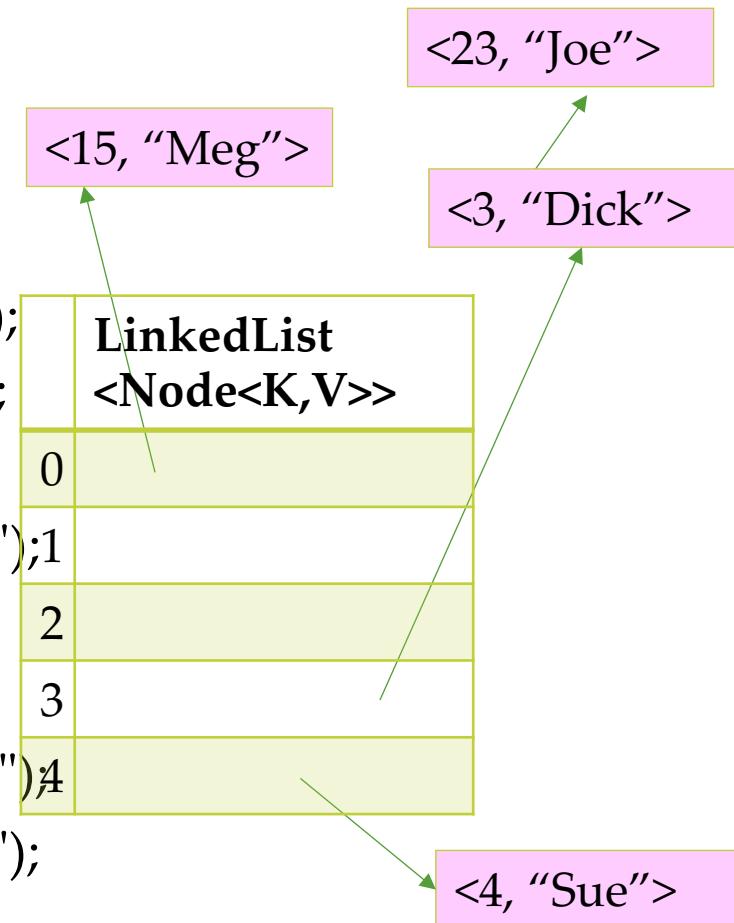


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- **myLookUp.remove(23); //Joe**
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan")
- myLookUp.insertOrUpdate(43, "Paul");

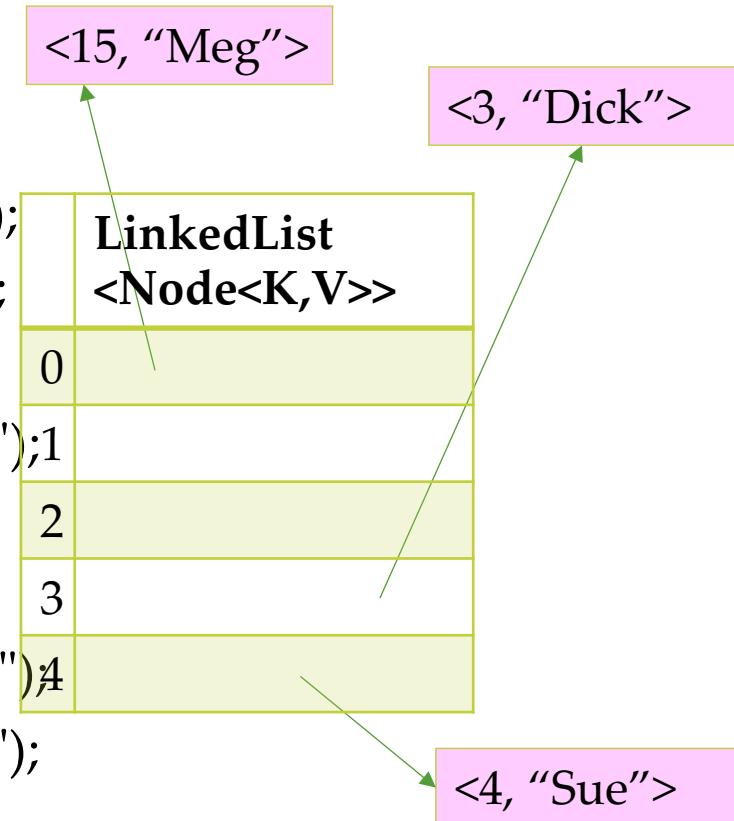


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- **myLookUp.remove(23); //Joe**
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan")
- myLookUp.insertOrUpdate(43, "Paul");

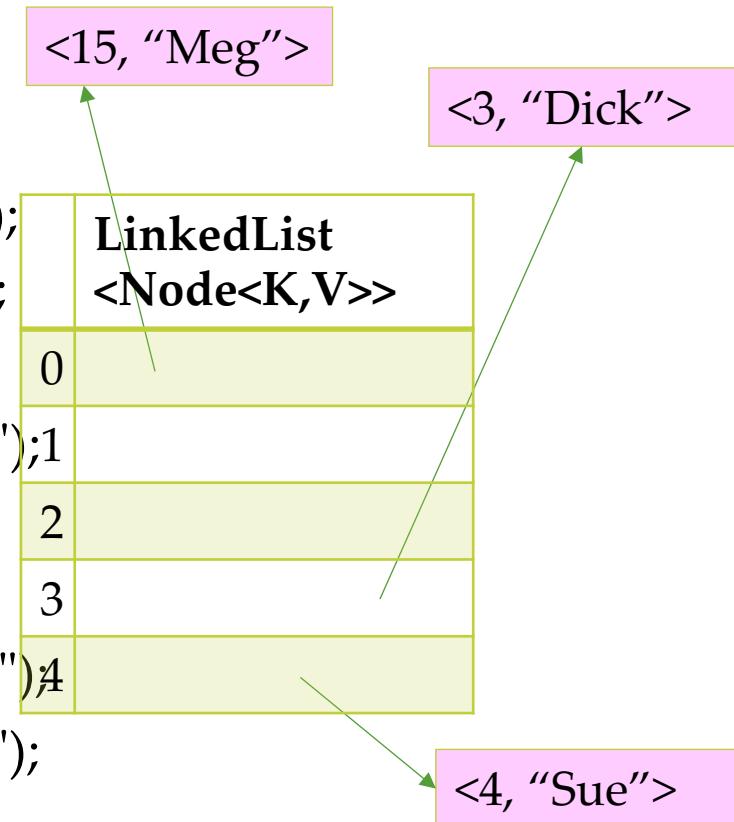


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- **myLookUp.remove(15); //Meg**
- myLookUp.insertOrUpdate(4, "Susan")
- myLookUp.insertOrUpdate(43, "Paul");

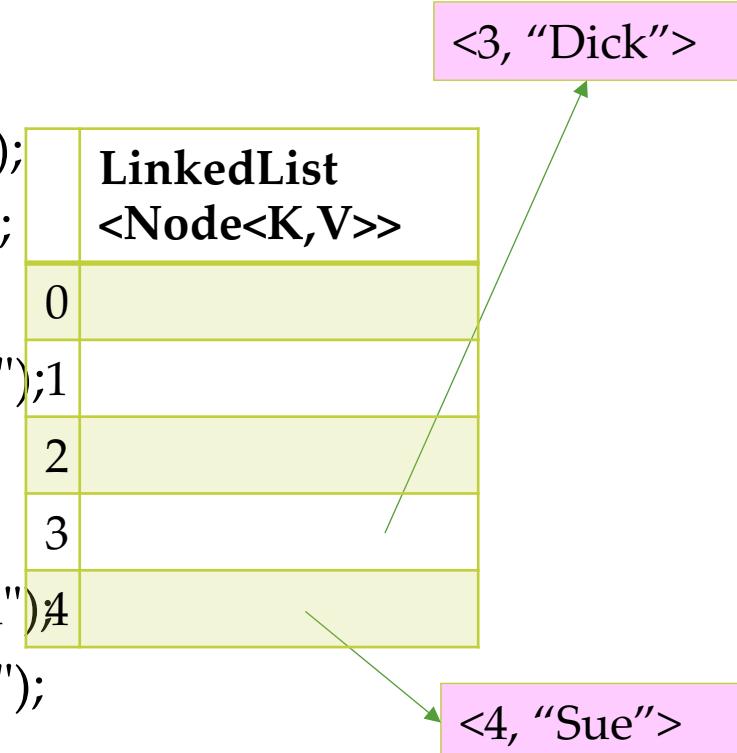


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- **myLookUp.remove(15); //Meg**
- myLookUp.insertOrUpdate(4, "Susan")
- myLookUp.insertOrUpdate(43, "Paul");

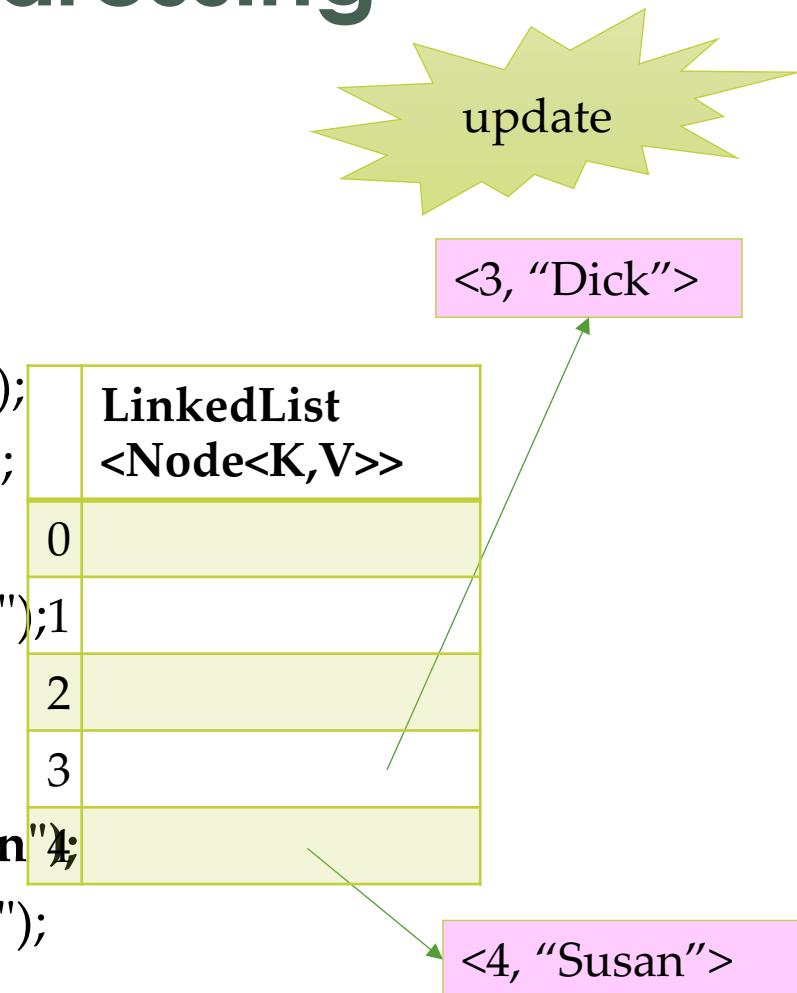


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- **myLookUp.insertOrUpdate(4, "Susan");**
- myLookUp.insertOrUpdate(43, "Paul");

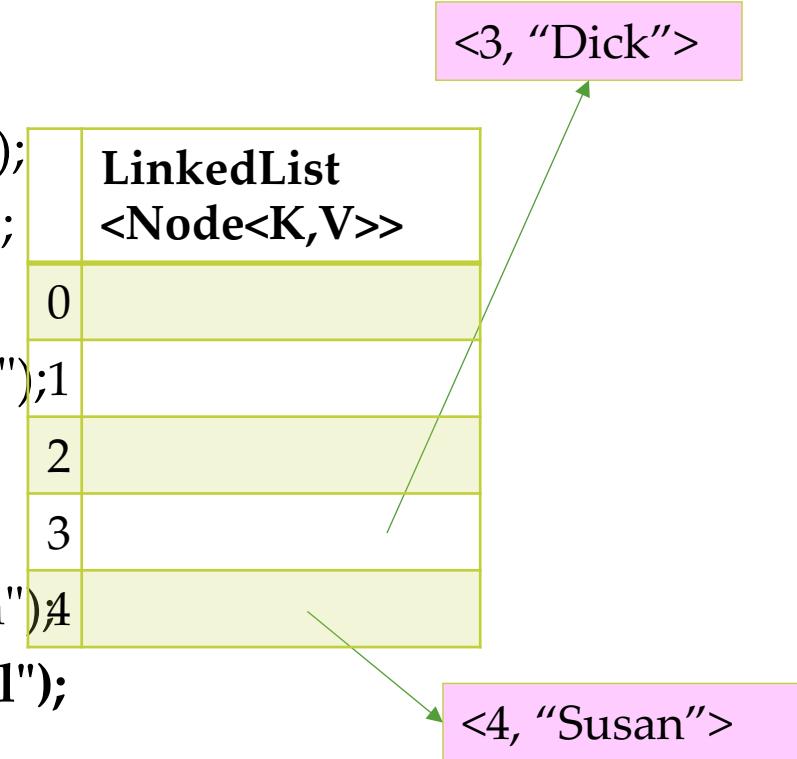


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan")
- myLookUp.insertOrUpdate(43, "Paul");

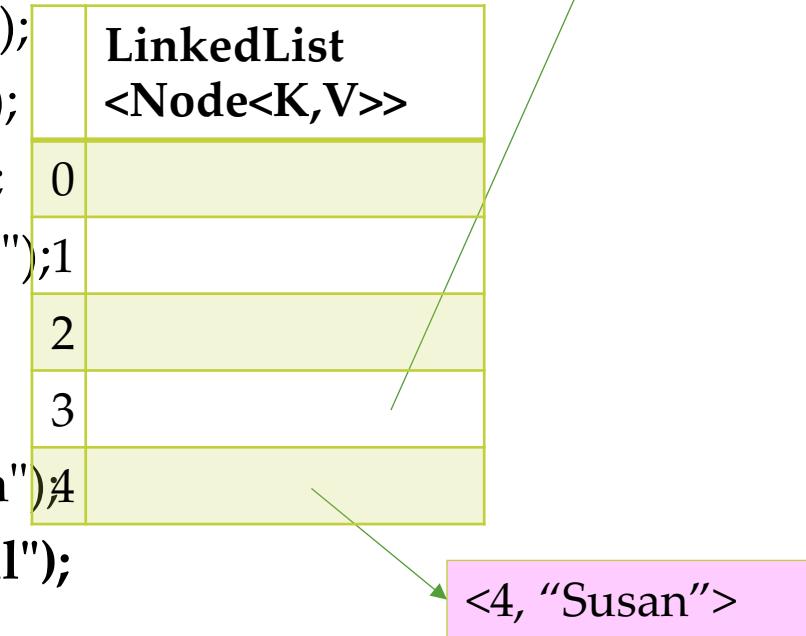


Open Hashing / Chaining / Overflow/ Closed Addressing

Caso de Uso A:

La fn pre hash el mismo key

- myLookUp.insertOrUpdate(3, "Dick");
- myLookUp.insertOrUpdate(23, "Joe");
- myLookUp.insertOrUpdate(4, "Sue");
- myLookUp.insertOrUpdate(15, "Meg");
- myLookUp.remove(23); //Joe
- myLookUp.remove(15); //Meg
- myLookUp.insertOrUpdate(4, "Susan")
- myLookUp.insertOrUpdate(43, "Paul");



TP 4- Ejer 6.1



Implementar estrategia
OpenHashing con zona de overflow version LinkedList.

No considerar aun factor de carga (lo discutimos en un rato)

Consideraciones

Súper importante redefinir `equals()`. Caso contrario, la lista no permitirá updates porque siempre considerará que los elementos son diferentes.

Manejar una lista en zona de overflow puede generar una complejidad mucho mayor que 1 en inserción/búsqueda/update/remove. ¿Por qué?

TP 4- Ejer 6.2



Mejorar la versión 6.1 con factor de carga:

Factor de carga global > threshold => duplica tabla y rehashea con la idea de acercarnos a O(1)

¿Un factor de carga global, soluciona totalmente el problema del desbalanceo?

Rta

No garantiza que hay alguna lista que esta muy sobrecargada.

Algunos usan factor de carga local y si una lista particular supera cierta longitude si hace split. Otros combinan las estrategias.

Hashing y Java

Hashing es una colección que viene implementada en Java.

Pregunta:

¿Cómo lo tiene implementado Java? En particular hubo un cambio muy importante **a partir de la versión 8**. Chequear esta versión.

<https://github.com/frohoff/jdk8u-jdk/blob/master/src/share/classes/java/util/HashMap.java>

Usa, al igual que como lo hicieron ustedes:

A) Arreglo de ranuras con zona de overflow

```
transient Node<K,V>[] table;
```

b) Manejo del espacio inicial y factor de carga

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16  
  
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

C) Si se acaba el espacio, lo duplica

```
++modCount;  
if (++size > threshold)  
    resize();  
    . . .
```

D) Tiene manejo de zona de overflow pero ¿qué coloca allí ?

Si no hay mucha ocupación en una ranura, entonces usa una lista

```
static class Node<K,V> implements Map.Entry<K,V> {  
    final int hash;  
    final K key;  
    V value;  
    Node<K,V> next;  
  
    Node(int hash, K key, V value, Node<K,V> next) {  
        this.hash = hash;  
        this.key = key;  
        this.value = value;  
        this.next = next; ←  
    }  
}
```

Pero si hay demasiados elementos allí, los transforma en un (red black) árbol !

```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent; // red-black tree links
    TreeNode<K,V> left;  ←
    TreeNode<K,V> right; ←
    TreeNode<K,V> prev;  // needed to unlink next upon deletion
    boolean red;
```

```

/**
 * Replaces all linked nodes in bin at index for given hash unless
 * table is too small, in which case resizes instead.
 */
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        TreeNode<K,V> hd = null, tl = null;
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            hd.treeify(tab);
    }
}

```

¿Y si mis Keys no son comparables? ¿Cómo maneja el orden del árbol?

En ese caso ordena por hashCode()



¡Muy elaborada la
nueva version!

Ya pueden terminar el TP 4 por cuenta de Uds.