

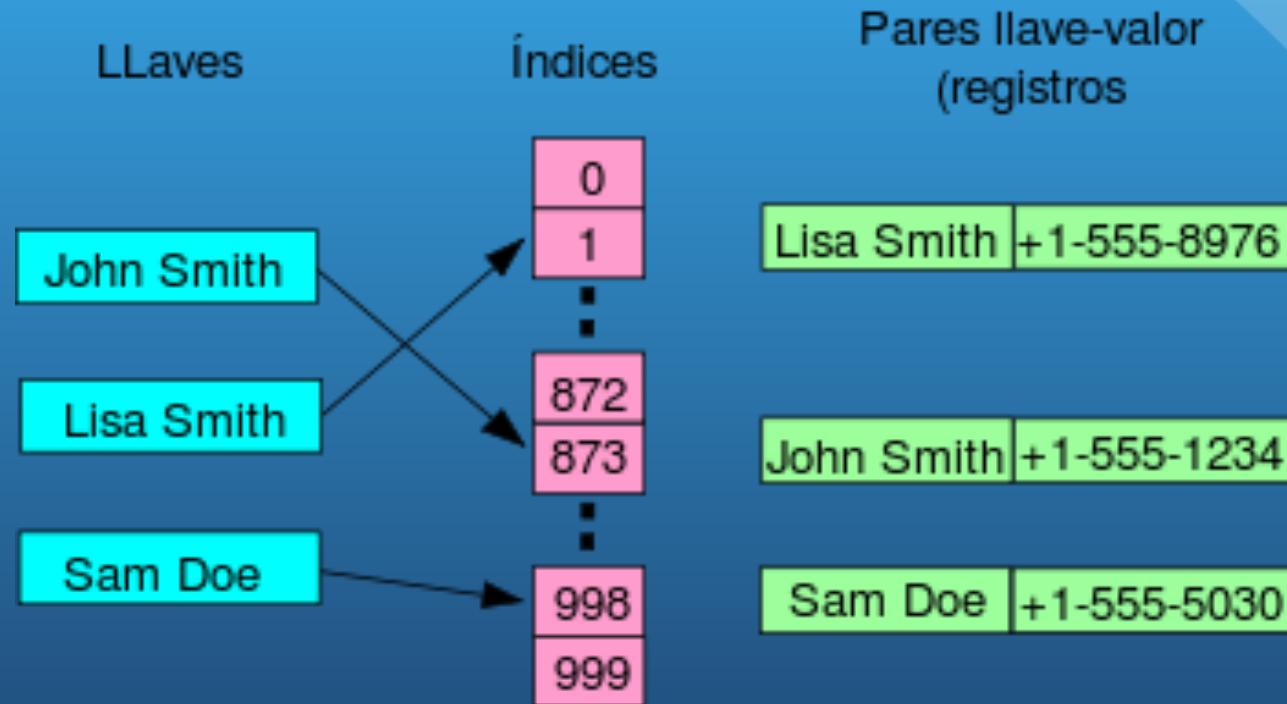


Estructuras de Datos

IIS262

Profesor: Patricio Galeas

CAPÍTULO 11 : Tablas Hash



Introducción

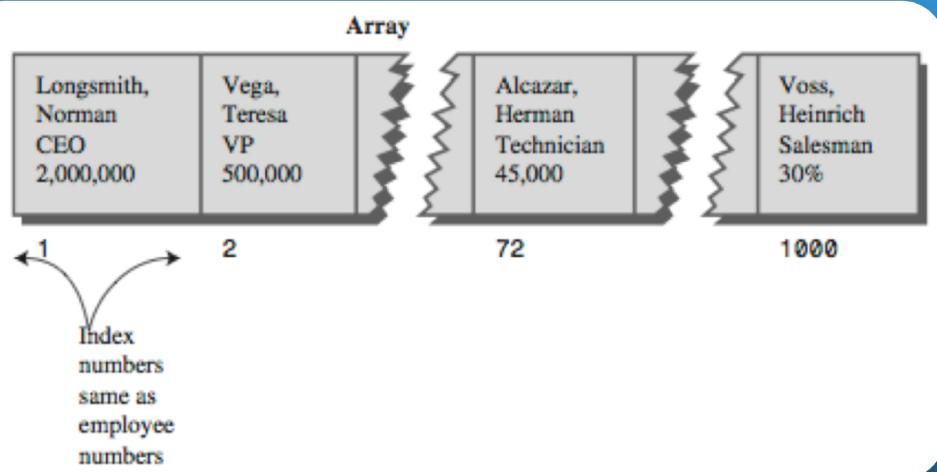
- Las tablas es una estructura de datos que permite una inserción y búsqueda muy rápida.
- No importando la cantidad de ítems en la tabla, la inserción, la búsqueda tienen un tiempo de ejecución de: $O(1)$.
- Las tablas Hash son más rápidas que los árboles que operan en un tiempo: $O(\log N)$.
- Aparte de ser rápidas, las tablas hash son fáciles de programar.
- Desventajas :
 - se basan en arreglos y se degradan cuando están muy llenas.
 - hay que tener muy claro cuantos datos voy a guardar en la tabla
 - De lo contrario hay que transferir datos a tablas más grandes (LENTO).

Que es Hashing

- Un concepto importante es como un rango de valores (palabras clave) es transformado en un rango de índices.
- En una tablas hash esto es logrado a través de una función.
- Sin embargo, para ciertos valores (claves) esta función no es necesaria. Ya que los valores pueden ser utilizados directamente como índices.
- Veamos este último caso, que es el más simple ...

Que es Hashing

- Posibilidad 1: INDICE == NÚMERO DE EMPLEADO



Esto funciona sólo si las claves están bien organizadas:

- Son secuenciales.
- No hay necesidad de eliminar.
- El número de elementos esta bien definido.

Acceder a un elemento :

```
empRecord rec = databaseArray[72];
```

Insertar un elemento :

```
databaseArray[totalEmployees++] = newRecord;
```

Que es Hashing

- Posibilidad 2 : UN DICCIONARIO
 - En muchos casos las claves no son tan manipulables como el caso anterior. El ejemplo clásico es un diccionario.
- Ejemplo:
 - Un diccionario con 50.000 palabras.
 - Cada palabra ocupa una celda en un arreglo de tamaño 50.000.
 - Se puede acceder a la palabra a través de su índice.
 - Pero, ¿cómo hago la relación entre el índice y la palabra que buscada?.



Que es Hashing



- Posibilidad 2 : UN DICCIONARIO
 - Necesitamos un método para pasar de una palabra a un índice determinado
 - Sumando los dígitos : $a b a c o = 1 + 2 + 1 + 3 + 15 = 22$
 - Con palabras de máx. 10 letras tenemos:
 - $a = 0+0+0+0+0+0+0+0+1 = 1$
 - $z z z z z z z z z z = 26+26+26+26+26+26+26+26+26+26 = 260$
 - Entonces en rango de los códigos : [1 - 260]
 - Pero tenemos 50.000 palabras
 - Entonces, cada elemento del arreglo almacenaría aprox. 192 palabras.
 - Crear un sub-arreglo o lista (de 192 elementos) en cada celda haría lenta la búsqueda.
 - ... Hay que aumentar la cantidad de índices...

Que es Hashing



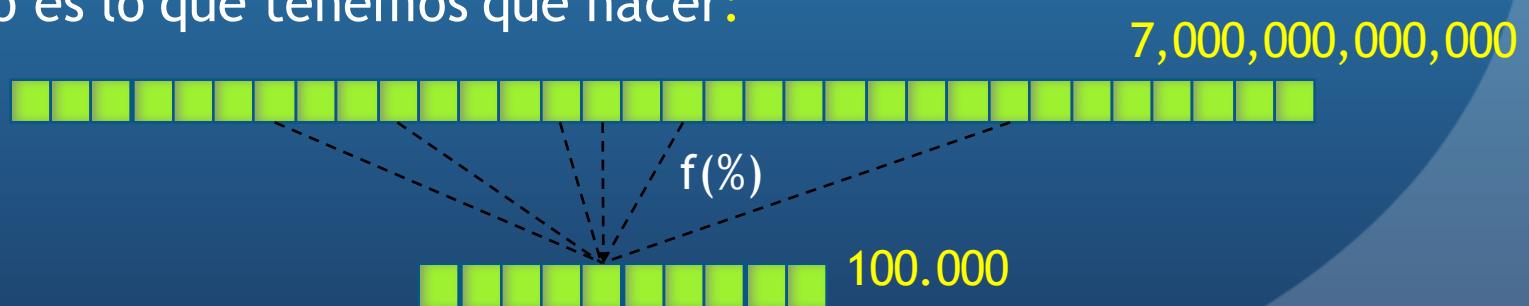
- Posibilidad 2 : UN DICCIONARIO
 - Multiplicando Potencias
 - Hay que garantizar que cada palabra del diccionario reciba un único índice/elemento en el arreglo.
 - Pensando en la composición de los número como potencias, donde los dígitos tienen 10 posibilidades: $7.546 = 7*10^3 + 5*10^2 + 4*10^1 + 6*10^0$
 - Análogamente : $abaco = 1*26^4 + 2*26^3 + 1*26^2 + 3*26^1 + 15*26^0 = 492.897$
 - Problema: para palabras muy grandes necesitamos indices muy grandes: Ejemplo: 10 letras requiere índices mayores que 7,000,000,000,000
 - Esto pasa porque este método asigna un índice a cada posible combinación de letras, a pesar de que esa combinación no exista como palabra: aaaaaaaaaaa, aaaaaaaaaaab, aaaaaaaaaac, ... , zzzzzzzzzz

Que es Hashing



- **Posibilidad 2 : UN DICCIONARIO**

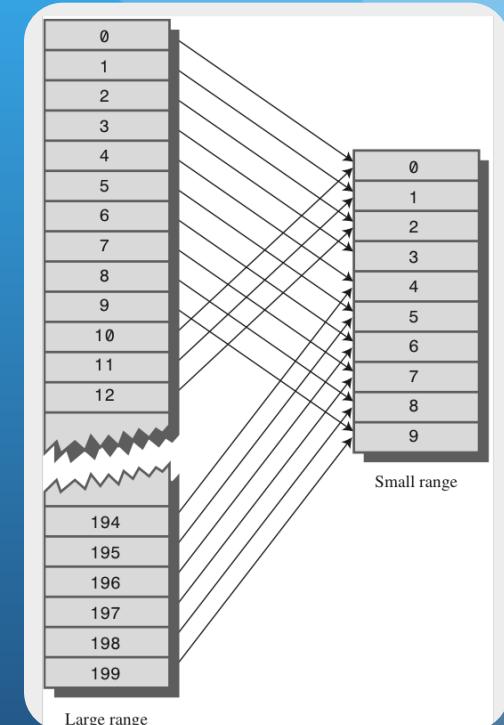
- El caso de las sumas de índices generaba muy pocos índices y el de las potencias generaba muchos índices. **¿qué hacemos?**
- Como tenemos 50.000 palabras necesitamos aprox. Un arreglo de igual tamaño.
- Asumamos que necesitaremos un arreglo con un tamaño que duplique la cantidad de palabras, es decir : 100.000 elementos (después veremos por qué).
- Esto es lo que tenemos que hacer:



Que es Hashing

- Posibilidad 2 : UN DICCIONARIO

- Ejemplo:
 - numeroGrande : de 0 a 199
 - rangoPequeño : de 0 a 9
 - numeroPequeño = numeroGrande % rangoPequeño
- Es decir, utilizando el operador módulo podemos comprimir un arreglo en un rango aproximado al doble de los datos que queremos almacenar:
 $\text{IndiceArreglo} = \text{numeroGrande \% rangoPequeño}$

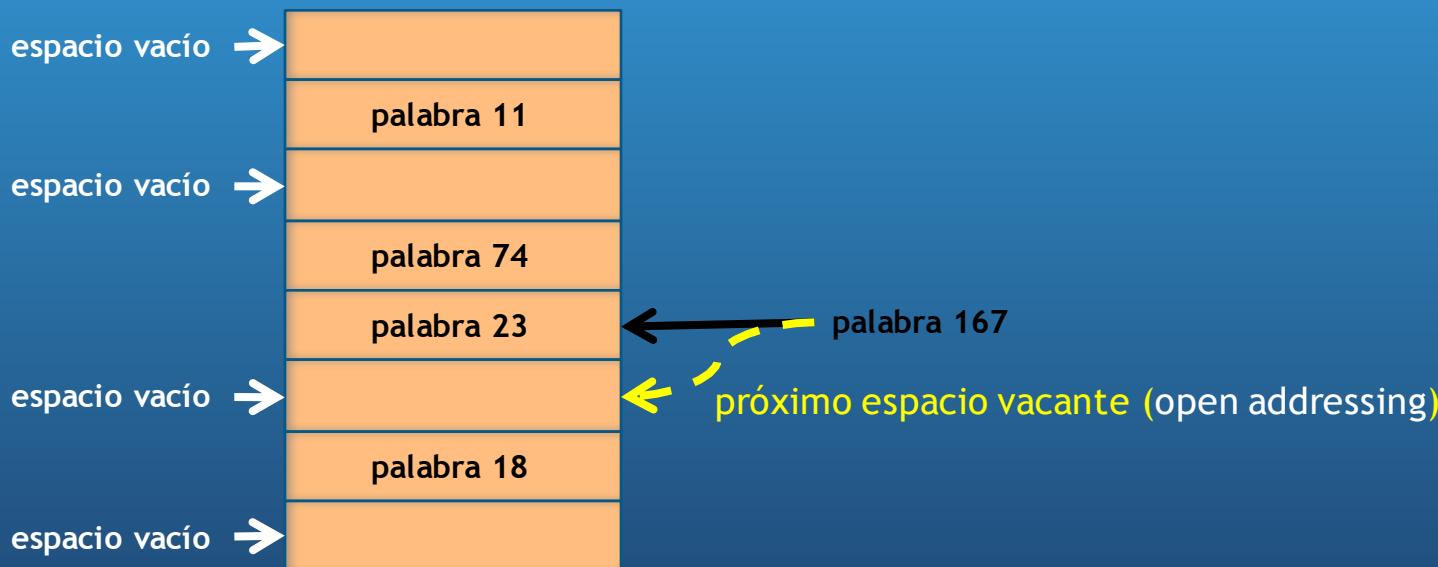
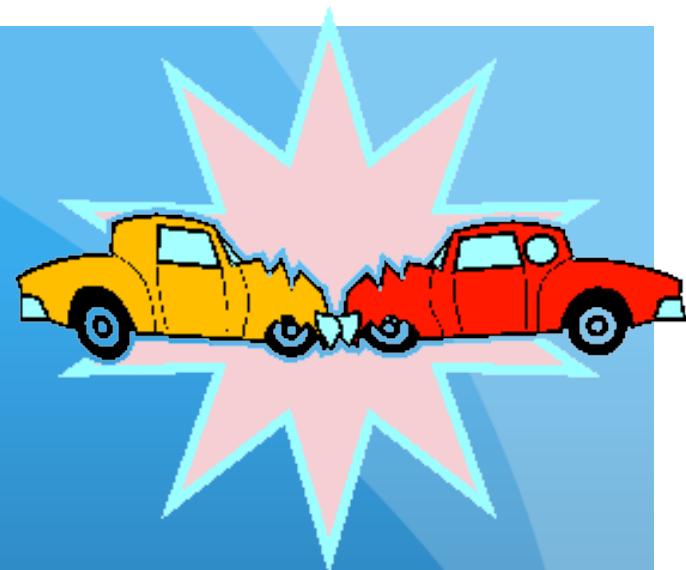


- En el caso de nuestro diccionario:
 $\text{tamañoArreglo} = \text{numeroPalabras} * 2;$
 $\text{indiceArreglo} = \text{numeroGrande \% tamañoArreglo};$



Colisiones

- Hay un precio por el hecho de comprimir un rango grande de número en otro más pequeño

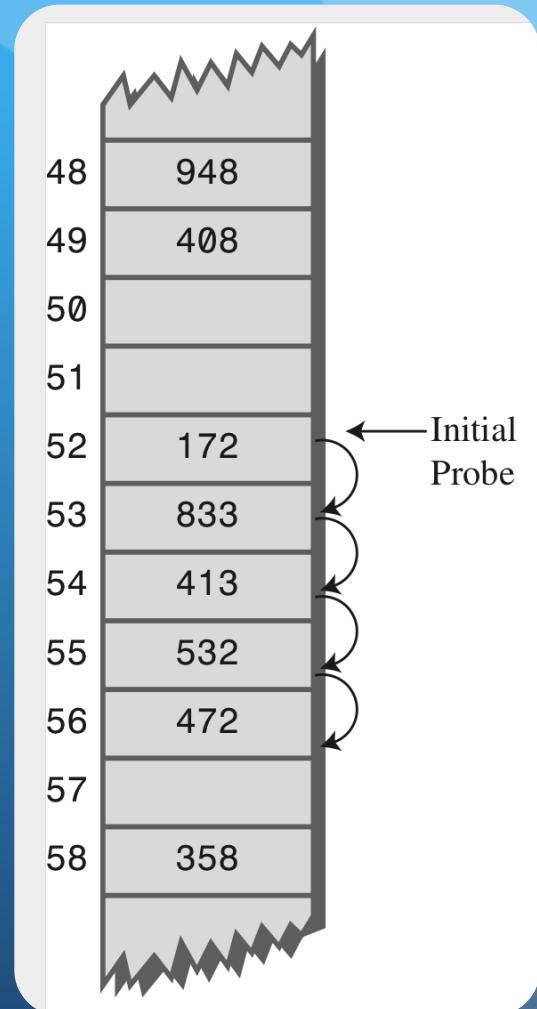


Open Addressing

Cuando un nuevo ítem no puede ser ubicado en el índice calculado por la función hash, se debe buscar una nueva posición disponible (open) para este valor en la tabla hash.

Exploraremos 3 métodos

- Prueba Lineal (Linear Probing):
- Prueba Cuadrática (Quadratic Probing)
- Hashing Doble.



Open Addressing -> Prueba Lineal

Linear probing busca secuencialmente espacios vacantes .



- Revisemos como funciona en el applet:

- Rango de valores : de 0 a 999
- Tamaño original del arreglo : 60
- Luego la función hash sería :
`indiceArreglo = valor % 60;`

cluster

AppletViewer: Hash.class

New Fill Ins Del Find Enter number:

Press any button

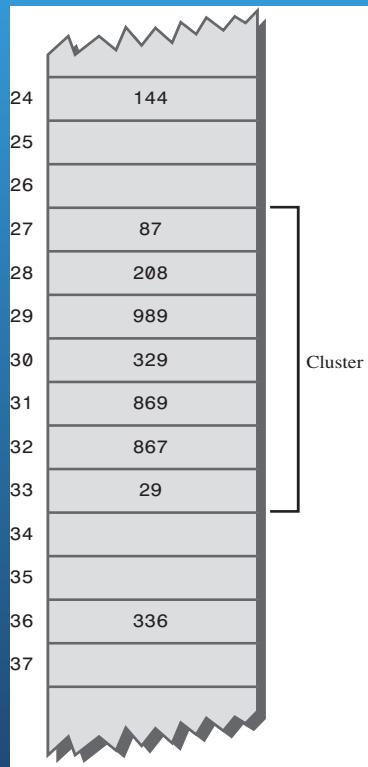
0	12	24	36	48
1	13	25	37	49
2	14	26	38	50
3	663	805	698	829
4	664	148	580	650
5	185	568	821	49
6	544	30	42	471
7	123	737	41	53
8	248	559	43	54
9	669	31	44	55
10	487	691	45	895
11	307	33	46	56
		393	47	57
		359		58
		755		898
				59

Subprograma iniciado.

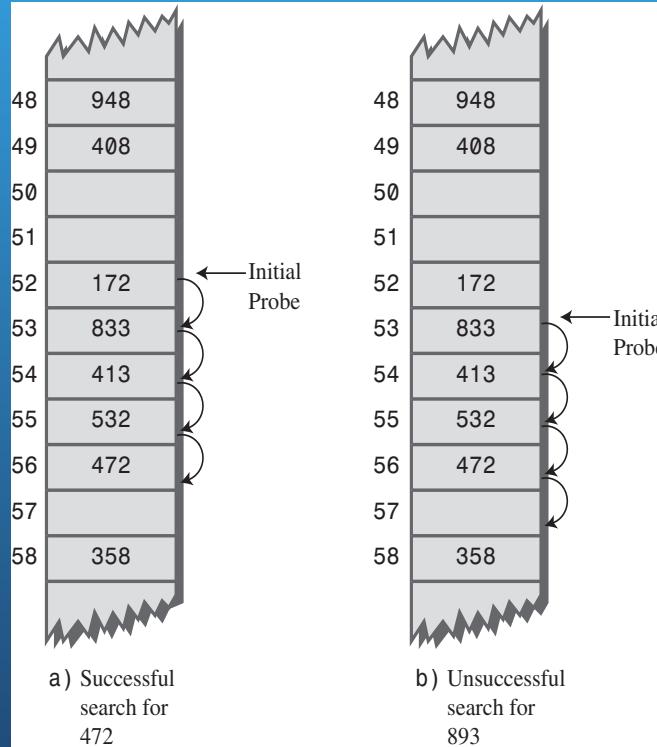
2015 - Patricio Galeas

Open Addressing -> Prueba Lineal->Insertar

Linear probing busca secuencialmente espacios vacantes .



Se van generando **clusters** de datos a medida que se va llenando el arreglo



El proceso de **buscar** la celda siguiente después de la colisión se llama “prueba”

Al **insertar** un elemento en un espacio ya ocupado (colisión), el “largo de la prueba” puede ser de varias celdas, si el arreglo esta muy lleno.

EJEMPLO: Si el arreglo es de tamaño 60, los valores 7, 67, 127, 187, 247, ..., 967. Ocuparan el mismo índice 7



Open Addressing -> Prueba Lineal -> Eliminar

Eliminar un elemento
no es dejar vacía la
celda.
¿Por qué?

- Recuerden que la inserción el proceso de “prueba” recorría una serie de celdas ocupadas buscando un espacio vacío.
- Si generamos un espacio vacío en el medio de este grupo de celdas ocupadas, ...
- ... la rutina de búsqueda terminará en el espacio generado y nunca encontrará el valor buscado.



SOLUCIÓN

Reemplazar el elemento borrado con un valor predeterminado (*Del*, -1, ...)

Luego

- El método *insertar* utilizará los espacios disponibles o aquellos ocupados por *Del*.
- El método *buscar* tratará a las celdas *Del* como ítems existentes

Esto no es muy eficiente!

Por esto muchas tablas Hash no permiten eliminación, o la eliminación de ítems es usada en forma muy esporádica.

Open Addressing -> Prueba Lineal->Duplicados

El applet sólo puede encontrar el primer elemento

- Este método podría ser modificado para encontrar todas las copias ...
- Pero esto tardaría bastante tiempo, ya que habría que buscar en todas las secuencias lineales que encuentre.



SOLUCIÓN

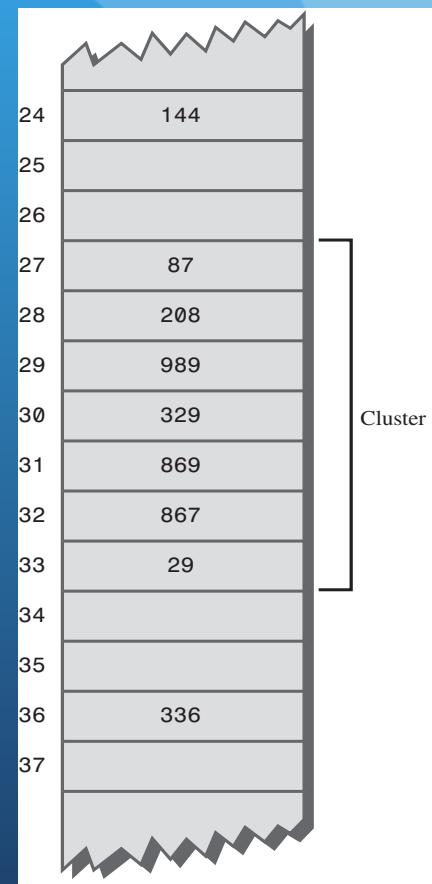
En la mayoría de las implementaciones de tablas hash se trata de evitar los duplicados.

Open Addressing -> Prueba Lineal -> Clustering

Cuando insertamos **muchos ítems** usando el applet, veremos que los clusters de ítems se van haciendo cada vez más grandes.

- Genera grandes secuencias del proceso de “prueba”.
- Se traduce en que el acceso a las celdas al final de la secuencia es muy lenta.

Proporción del arreglo lleno	Eficiencia de la Tabla Hash
1/2	buenas
2/3	medianas
Mayor a 2/3	mala



Open Addressing -> Prueba Lineal

Implementaciones en Java

método **find**

```
public DataItem find(int key)      // find item with key
// (assumes table not full)
{
    int hashVal = hashFunc(key);  // hash the key

    while(hashArray[hashVal] != null) // until empty cell,
    {
        if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal]; // yes, return item
        ++hashVal;                // go to next cell
        hashVal %= arraySize;     // wrap around if necessary
    }
    return null;
} // can't find item
```

calcula el índice

compara el valor buscado

Si no lo encuentra,
incrementa el índice

Evita salida del
límite del arreglo

¿Que pasa con este método si el arreglo se llena?



Open Addressing -> Prueba Lineal

Implementaciones en Java

método insert

```
public void insert(DataItem item) // insert a DataItem  
// (assumes table not full)  
{  
    int key = item.getKey();          // extract key  
    int hashVal = hashFunc(key);    // hash the key  
                                    // until empty cell or -1,  
    while(hashArray[hashVal] != null &&  
          hashArray[hashVal].iData != -1)  
    {  
        ++hashVal;                  // go to next cell  
        hashVal %= arraySize;       // wrap around if necessary  
    }  
    hashArray[hashVal] = item;        // insert item  
} // end insert()
```

- Muy similar al procedimiento que buscar (find), para encontrar la posición del nuevo ítem.
- Sólo se diferencia en la condición de búsqueda (ítem vacío o eliminado)



Open Addressing -> Prueba Lineal

Implementaciones en Java

método delete

```
public DataItem delete(int key) // delete a DataItem
{
    int hashVal = hashFunc(key); // hash the key

    while(hashArray[hashVal] != null) // until empty cell,
        {                                // found the key?
            if(hashArray[hashVal].getKey() == key)
            {
                DataItem temp = hashArray[hashVal]; // save item
                hashArray[hashVal] = nonItem;      // delete item
                return temp;                      // return item
            }
            ++hashVal;                      // go to next cell
            hashVal %= arraySize;          // wrap around if necessary
        }
    return null;                      // can't find item
} // end delete()
```

- Encuentra el ítem a eliminar usando algo muy similar a find.
- Luego sobreescribe el valor del ítem encontrado con el valor de “nonItem” especificado anteriormente con el valor -1.



Open Addressing -> Prueba Lineal

Implementaciones en Java

El programa hash.java



- DataItem contiene sólo un atributo: un entero
- El atributo principal es HashTable, el cual es una arreglo llamado hashArray.
- El main() en la clase HashTableApp contiene una interfaz de usuario que permite:
 - Ver el contenido de la tabla hash
 - Insertar un ítem
 - Borrar un ítem
 - Encontrar un ítem

Open Addressing -> Prueba Lineal

Expandir el Arreglo



Expandiendo
el Arreglo

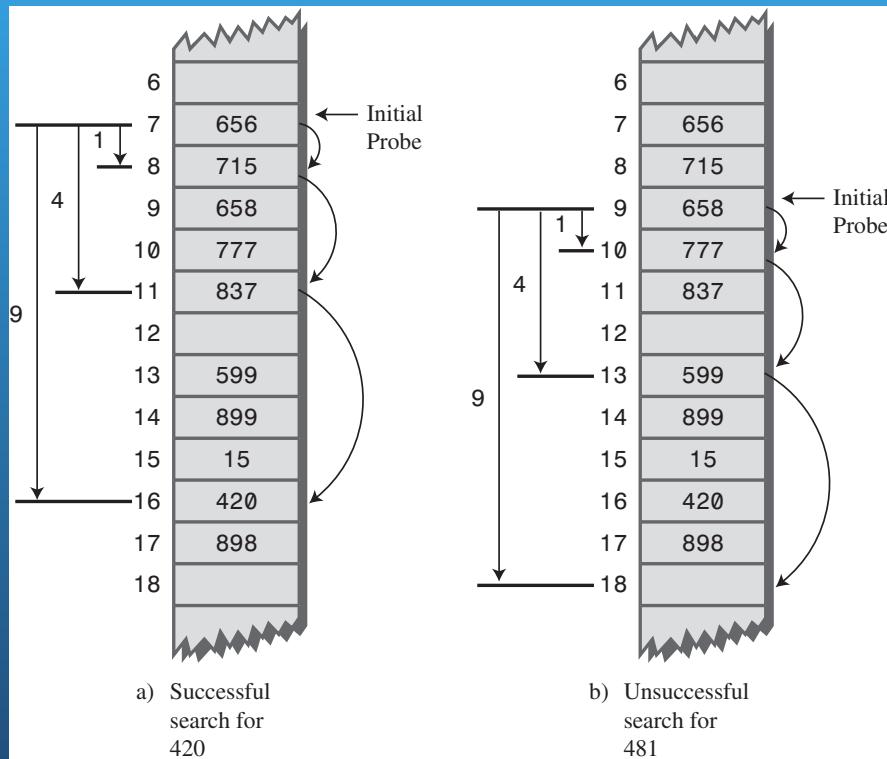
- Cuando la tabla hash comienza a llenarse una opción es expandir su arreglo.
- Hay que crear un nuevo arreglo más grande (doble) e insertar en él, el contenido del arreglo antiguo.
- Como la función hash calcula la posición de los elementos en función del tamaño de arreglo, estos no quedarán en la misma posición del arreglo original.
- Hay que copiar los ítems uno a uno, utilizando el método insert : **rehashing** (proceso lento)

Open Addressing -> Prueba Cuadrática

X²

- En la prueba lineal los clusters van creciendo constantemente.
- Mientras más grande el cluster, éste crece mas rápido.
- Los cluster reducen el rendimiento de la tabla hash.
- La Prueba Cuadrática es una forma de evitar la formación de clusters.
- La idea es probar en celdas más espaciadas, en vez de las adyacentes.
- El salto/paso es el cuadrado del paso original.
- Factor de Carga: es la relación entre el número de items de una tabla y el tamaño de esta:
$$\text{loadFactor} = \text{nItems} / \text{arraySize}$$

Open Addressing -> Prueba Cuadrática



- **Prueba Lineal:** $x+1, x+2, x+3, \dots$
- **Prueba Cuadrática:** $x+1^2, x+2^2, x+3^2, x+4^2, x+5^2, \dots$



Probar buscando en una tabla de 59 celdas con 47 ítems.

Nota: Si el tamaño del arreglo **no es** un número primo podría ocurrir una secuencia infinita en el cálculo de la prueba.

Open Addressing -> Prueba Cuadrática

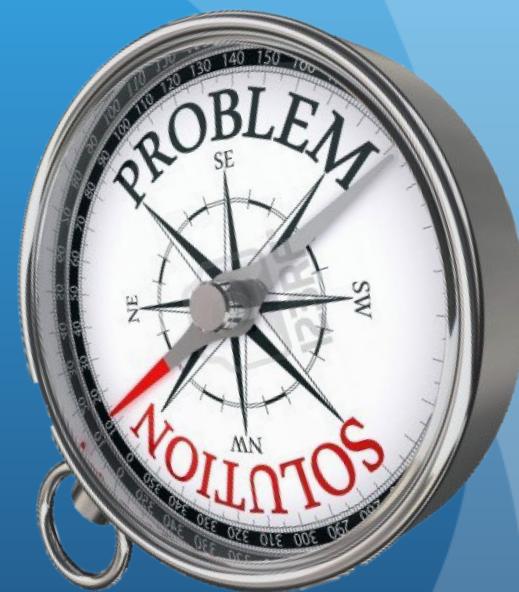
Problema con la Prueba Cuadrática

Todos los valores que son “hasheados” a una celda particular siguen la misma secuencia tratando de encontrar un espacio disponible :

Ejemplo: Supongamos que tenemos una tabla de tamaño 59, entonces si insertamos 184, 302, 420 y 544, tienen un hash de 7 -> Clustering Secundario

Luego:

- 184 se guarda en 7
- 302 requiere una prueba de 1 paso
- 420 requiere una prueba de 4 pasos
- 544 requiere una prueba de 9 pasos
- ... cualquier otro número cuyo hash es 7 requerirá una prueba mayor



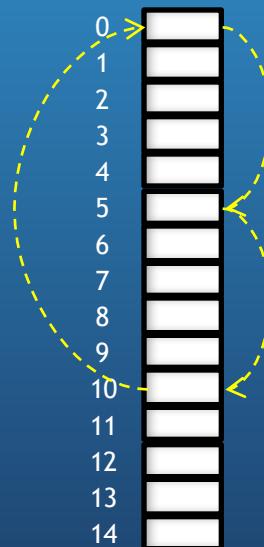
Solución:
Doble Hashing

Open Addressing -> Doble Hashing

¿Por qué el tamaño de la tabla debiera ser un numero primo?

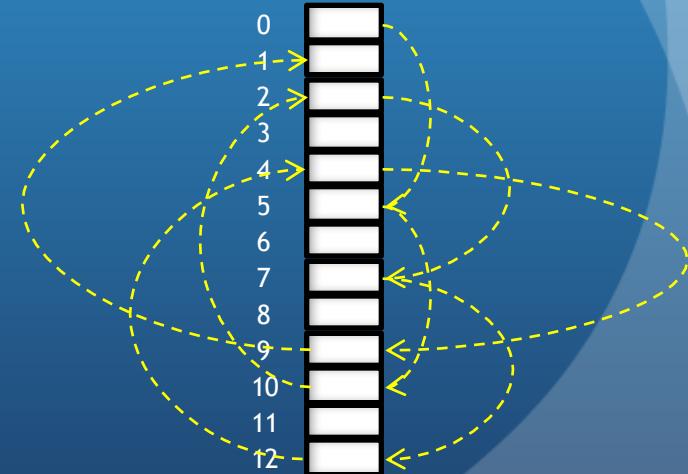
Indice inicial = 0
Step = 5

ciclo infinito



Tamaño 15 (no primo)

Indice inicial = 0
Step = 5



Tamaño 13 (primo)

Open Addressing -> Doble Hashing

- Para eliminar el problema del clustering primario y secundario necesitamos que las secuencias de prueba dependan del valor ingresado en vez de ser las mismas para todos los valores.
- Entonces los valores que tienen un mismo hash usaran secuencias de prueba distintas.
- SOLUCION:
 - Hashear el valor por segunda vez, usando diferentes funciones de hash y utilizar el resultado como el tamaño del paso.
- ENTONCES:
 - El tamaño del paso es constante para un valor determinado, pero distinto para valores diferentes.
- RESTRICCIONES de la función de hash secundaria:
 - No puede ser la misma que la función de hash primaria
 - Nunca debe dar un valor 0 (paso=0, cada prueba termina en la misma celda)

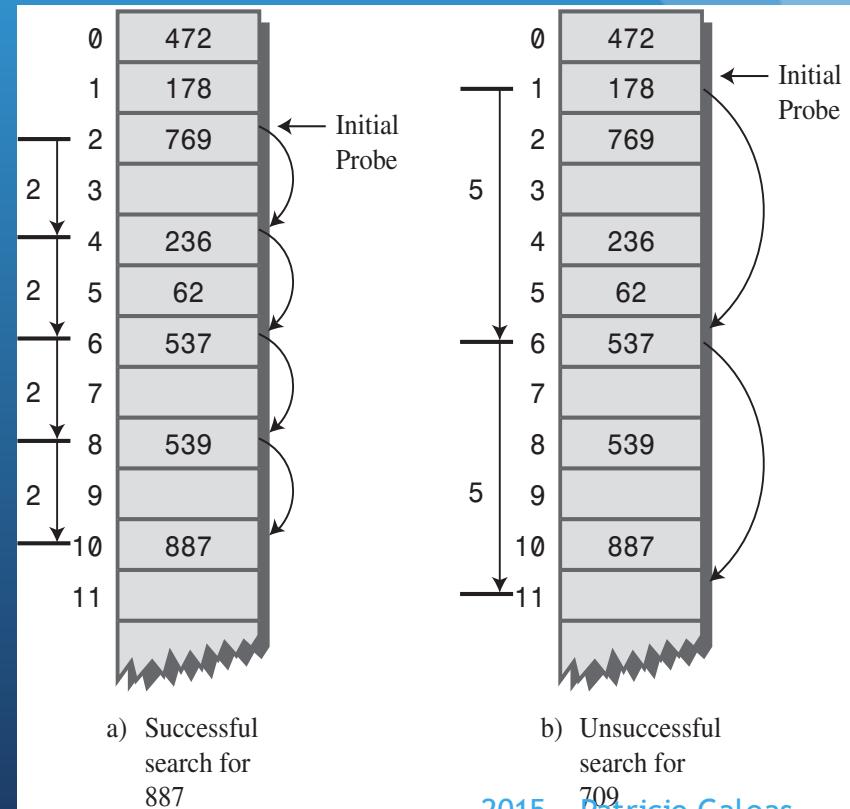
Open Addressing -> Doble Hashing

Ej. función hashing secundaria:

$$\text{paso} = \text{constante} - (\text{valor \% constante})$$

donde la *constante* es un valor primo menor que el tamaño del arreglo.

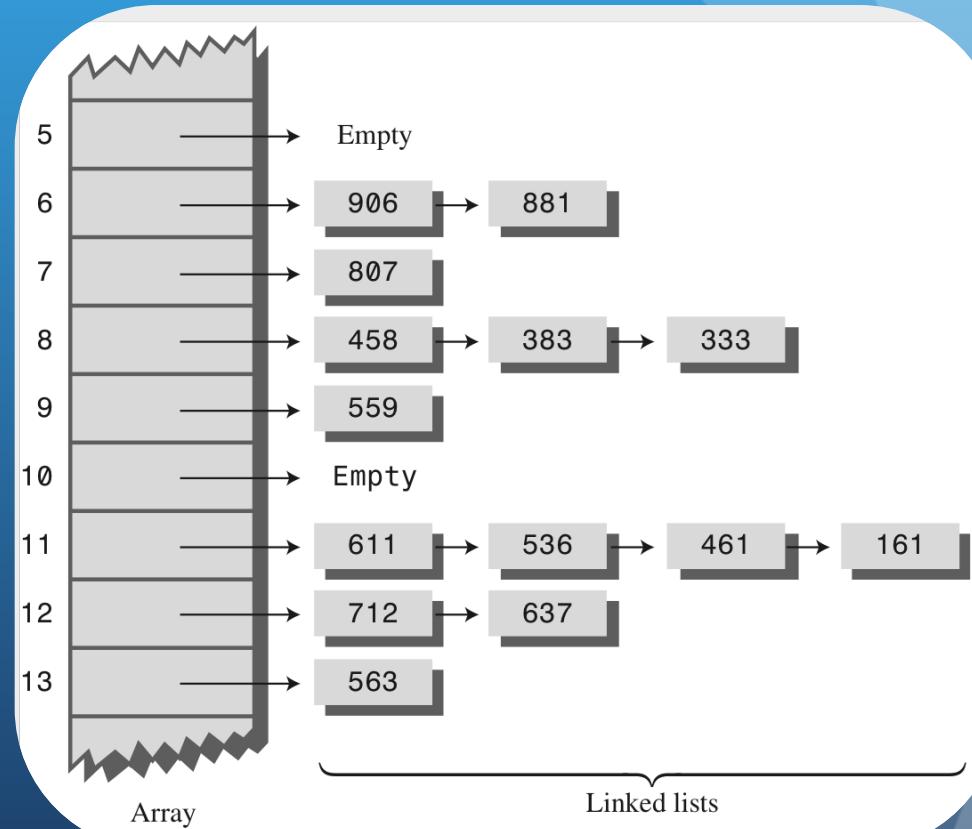
Ejemplo : $\text{paso} = 5 - (\text{valor \% } 5)$



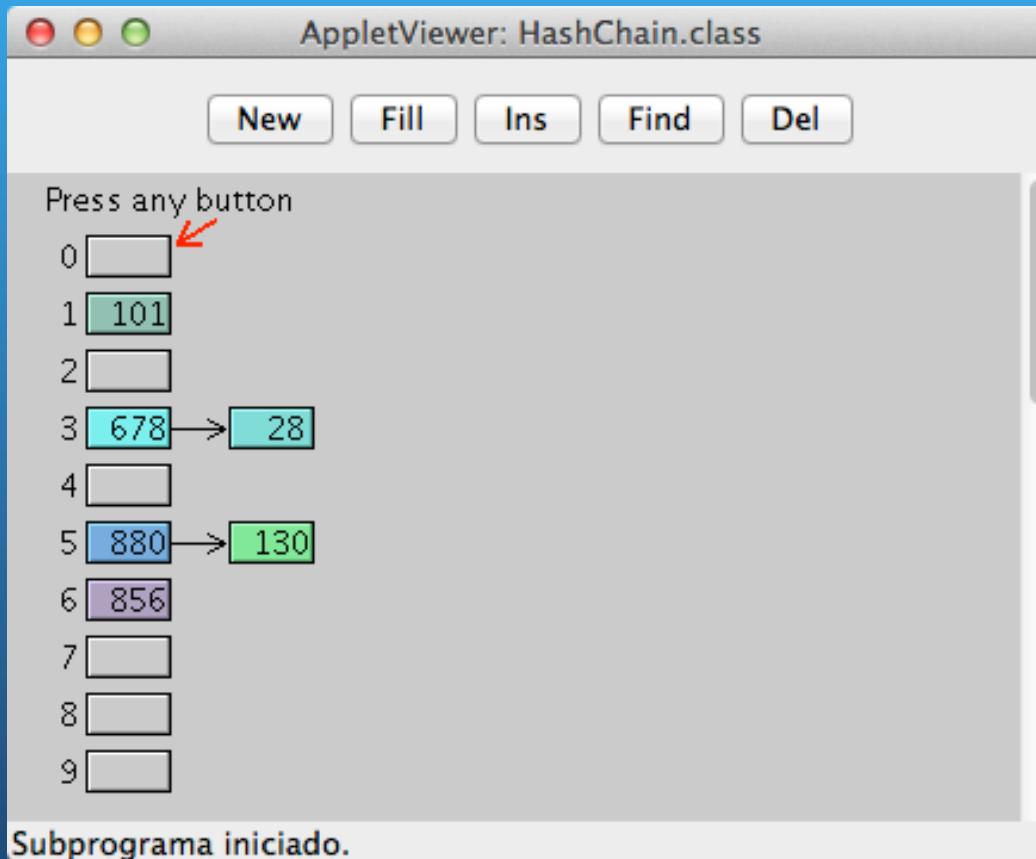
Separate Chaining

En el procedimiento anterior (Open Addressing), las colisiones eran resueltas buscando celdas disponibles (open) en la tabla hash.

- Otra solución es **instalar una Lista Enlazada** en cada índice de la tabla Hash.
- Los valores son indexados de manera usual, siendo **insertados en la lista enlazada** asociada al index.
- Otros ítems con el mismo índice **son simplemente agregados** a la lista enlazada.



Separate Chaining-> Applet



HashChain



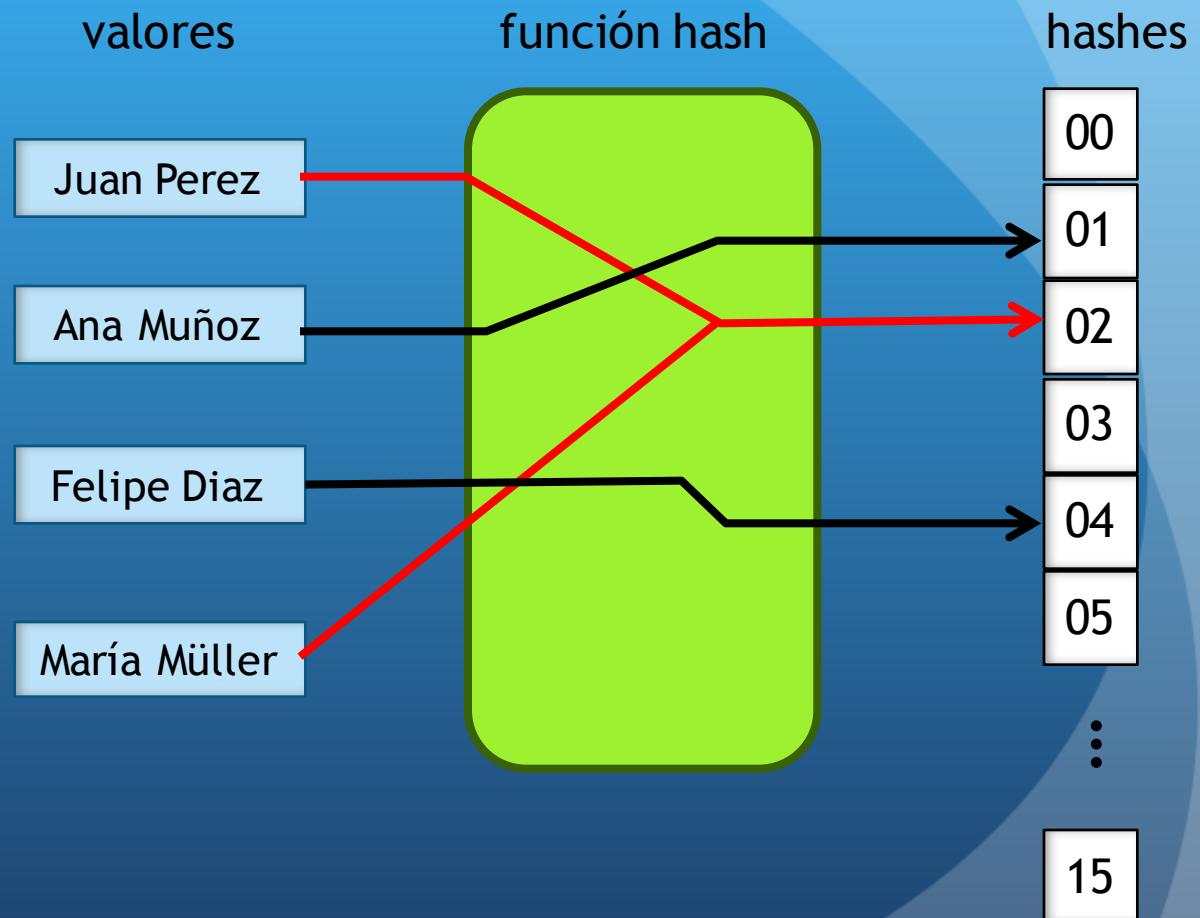
Applet

Separate Chaining

- **Factor de Carga** : puede ser > 1 sin afectar el rendimiento considerablemente.
- **Duplicados** : Están permitidos, siendo insertados en la misma lista (baja el rendimiento).
- **Borrado** : funciona similar al de Open Addressing, no importa si la lista queda vacía.
- **Tamaño de la Tabla** : Aquí no es tan importante usar valores primos como en Open Addressing y Doble Hashing, dado que no hay secuencias de pruebas.
- **Arreglos**: Otro forma de configurar Separate Chaining es utilizar arreglos en vez de listas enlazadas. Pero presenta **desventaja** del uso de memoria propio de los arreglos (tamaño predefinido).

Funciones Hash

Aquí exploraremos que hace a una buena función de hash y como mejorar el indexado de strings mencionado al principio de este capítulo

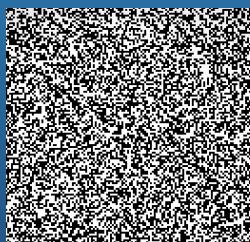


Funciones Hash



Calculo Rápido:

- Una buena función de hash es simple y rápida.
- La idea es tomar una serie de valores y transformarlos en índices, de manera que los valores queden aleatoriamente distribuidos en los índices del arreglo.



Valores Aleatorios:

- Una función hash perfecta mapea cada valor en diferentes partes de la tabla.
- Pero esto es casi imposible.
- La distribución de los valores en particular determina la función a utilizar. Por ejemplo, para datos aleatorios funciona bien: `index = valor % tamañoArreglo`

Funciones Hash



Valores No Aleatorios:

Lamentablemente la información es a menudo no aleatoria. Por ejemplo una BD de partes de vehículos.

033 - 400 - 03 - 94 - 05 - 0 - 535
nro. categoria mes año serie tóxico checksum
distrib. (bit)

Estos valores no están distribuidos aleatoriamente, ya que la mayoría de los valores entre 0 y 9.999.999.999.999 no serán ocupados. => habría que implementar algo de manera que los valores fueran distribuidos en forma más aleatoria.

- No usar datos sin información adicional
 - Comprimir: por ejemplo “categoría” sólo tiene 16 valores.
 - No incluir datos sin información adicional (checksum)
 - Utilice el máximo de datos en la función hash (no sólo un par de dígitos), para asegurar un rango mayor de índices.
 - Utilice números primos para el calculo del módulo.

Indexando Strings - método básico

$$\begin{array}{cccc}
 C & a & t & S \\
 3*27^3 & 1*27^2 & 20*27^1 & 19*27^0 \\
 \end{array} = 60337$$

Índice en la tabla = $60337 \% \text{arraySize}$

```

public static int hashFunc1(String key)
{
    int hashVal = 0;
    int pow27 = 1; // 1, 27, 27*27, etc

    for(int j=key.length()-1; j>=0; j--) // right to left
    {
        int letter = key.charAt(j) - 96; // get char code
        hashVal += pow27 * letter; // times power of 27
        pow27 *= 27; // next power of 27
    }
    return hashVal % arraySize;
} // end hashFunc1()

```

Pero, este método **no es muy eficiente**, ya que hay dos multiplicaciones y una suma en el loop.

Se puede mejorar usando la identidad matemática de **Horner**.

Indexando Strings - método mejorado

Horner dice:

$$\text{var4} \cdot n^4 + \text{var3} \cdot n^3 + \text{var2} \cdot n^2 + \text{var1} \cdot n^1 + \text{var0} \cdot n^0$$

puede ser escrito como:

$$(((\text{var4} \cdot n + \text{var3}) \cdot n + \text{var2}) \cdot n + \text{var1}) \cdot n + \text{var0}$$

```
public static int hashFunc2(String key)
{
    int hashVal = key.charAt(0) - 96;
    for(int j=1; j<key.length(); j++) // left to right
    {
        int letter = key.charAt(j) - 96; // get char code
        hashVal = hashVal * 27 + letter; // multiply and add
    }
    return hashVal % arraySize; // mod
} // end hashFunc2()
```

Pero, este método no puede manejar strings mayores a 7 caracteres, ya que sobrepasa el largo de los enteros (int).

Se puede mejorar la aplicando el módulo dentro del ciclo.

Indexando Strings - método mejorado

Aplicando el módulo dentro del ciclo ...

```
public static int hashFunc3(String key)
{
    int hashVal = 0;
    for(int j=0; j<key.length(); j++)      // left to right
    {
        int letter = key.charAt(j) - 96;   // get char code
        hashVal = (hashVal * 27 + letter) % arraySize; // mod
    }
    return hashVal;
} // end hashFunc3()
```

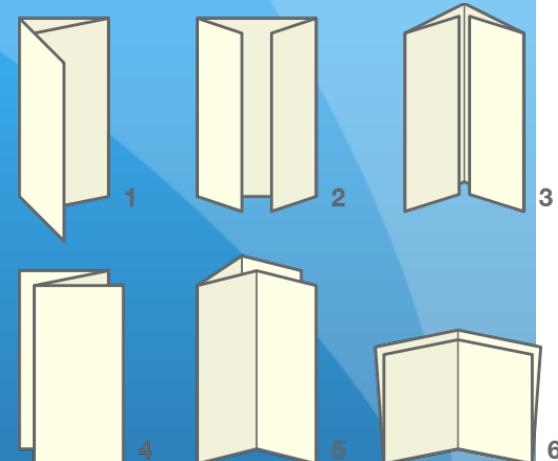
// no mod

Este es uno de los métodos normalmente usados para indexar cadenas de caracteres.

Folding

Otras funciones hash interesantes involucran el **separar el valor a indexar en grupos de dígitos**, y después **sumar los grupos**.

- Esto garantiza que todos los dígitos influencien el valor del índice.
- El **número de dígitos** en un grupo debe corresponder al **tamaño del arreglo**. Para un arreglo de 1.000 ítems se deben usar grupos de 3 dígitos.



EJEMPLO:

Supongamos que queremos indexar el Rut 12.345.678-9

- En un arreglo de tamaño 1000.

Hay que dividir el String en 3 grupos y sumar: $123+456+789 = 1368$

Luego usamos el módulo para hacer calzar la suma al índice más alto (999):

$$1368 \% 1000 = 368$$

- Si el tamaño del arreglo fuera de 100:

$$12+34+56+78+9 = 189 \% 100 = 89$$

Eficiencia del Hashing

- La **inserción** en una tabla hash tiene un **orden O(1)**, siempre y cuando **no existan colisiones**.
- Si hay **colisiones**, el tiempo de la inserción **depende del largo de la prueba**.
- Durante el acceso la celda debe ser chequeada para verificar si esta vacía o no. En el caso de la **búsqueda** y la **eliminación** se debe chequear si el ítem contiene el valor deseado.
- Por esto, la **búsqueda** también tienen un **tiempo proporcional al largo de la prueba**.



El **largo promedio de la prueba** depende **del factor de carga**: a mayor factor de carga mayor largo de prueba.

Eficiencia del Hashing para Open Addressing



La perdida de eficiencia para valores altos del factor de carga es más seria en el esquema Open Addressing que en el de Separate Chains.

En Open Addressing las búsquedas no exitosas toman más tiempo que las exitosas.

Eficiencia del Hashing

Open Addressing

Prueba Lineal

La siguiente ecuaciones (fórmulas de Knuth) muestran la relación entre el largo de la prueba (P) y el factor de carga (L).

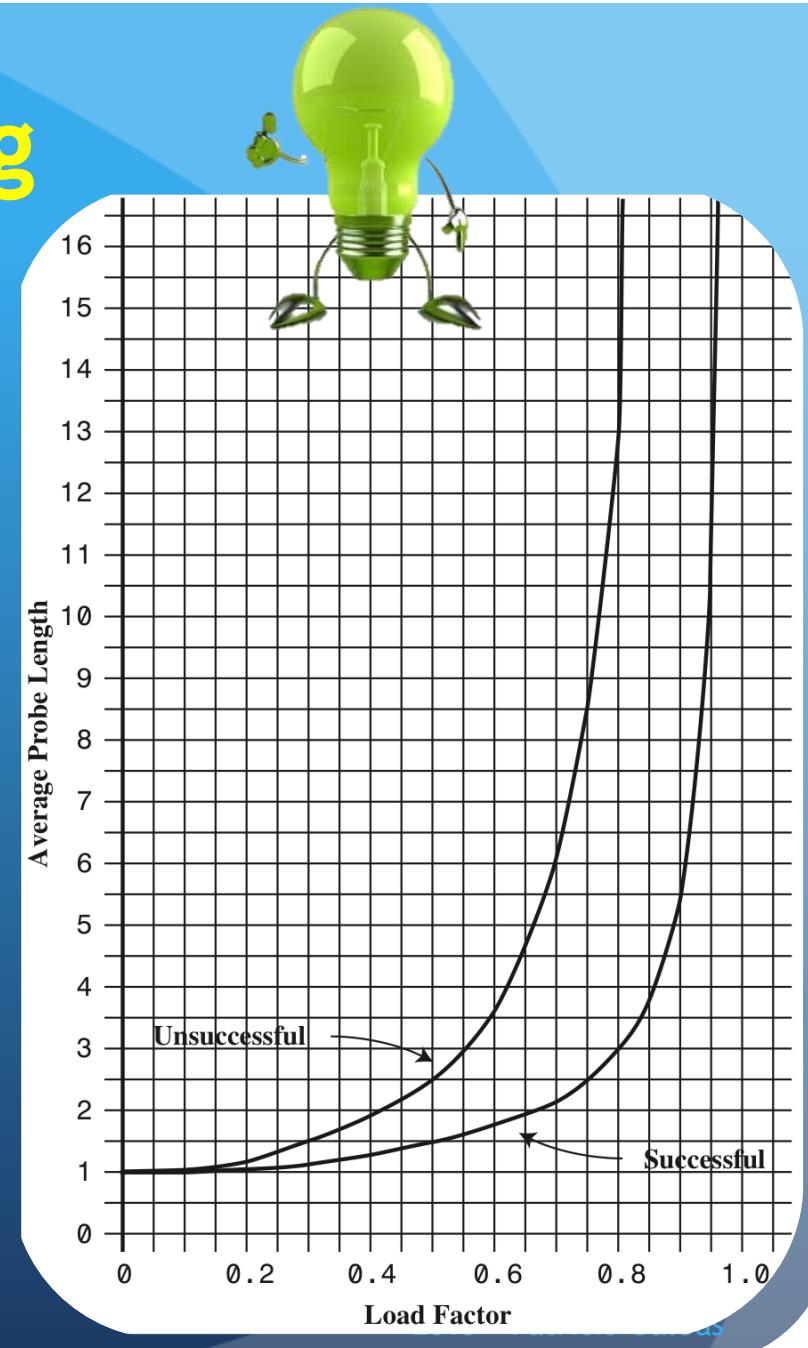
- Para una búsqueda exitosa:

$$P = (1 + 1 / (1 - L)^2) / 2$$
- Para una búsqueda no exitosa:

$$P = (1 + 1 / (1 - L)) / 2$$

Para factores de carga sobre 2/3 el largo de la prueba es muy grande.

Sin embargo, mientras menor sea el factor de carga mayor memoria necesita el sistema.



Eficiencia del Hashing

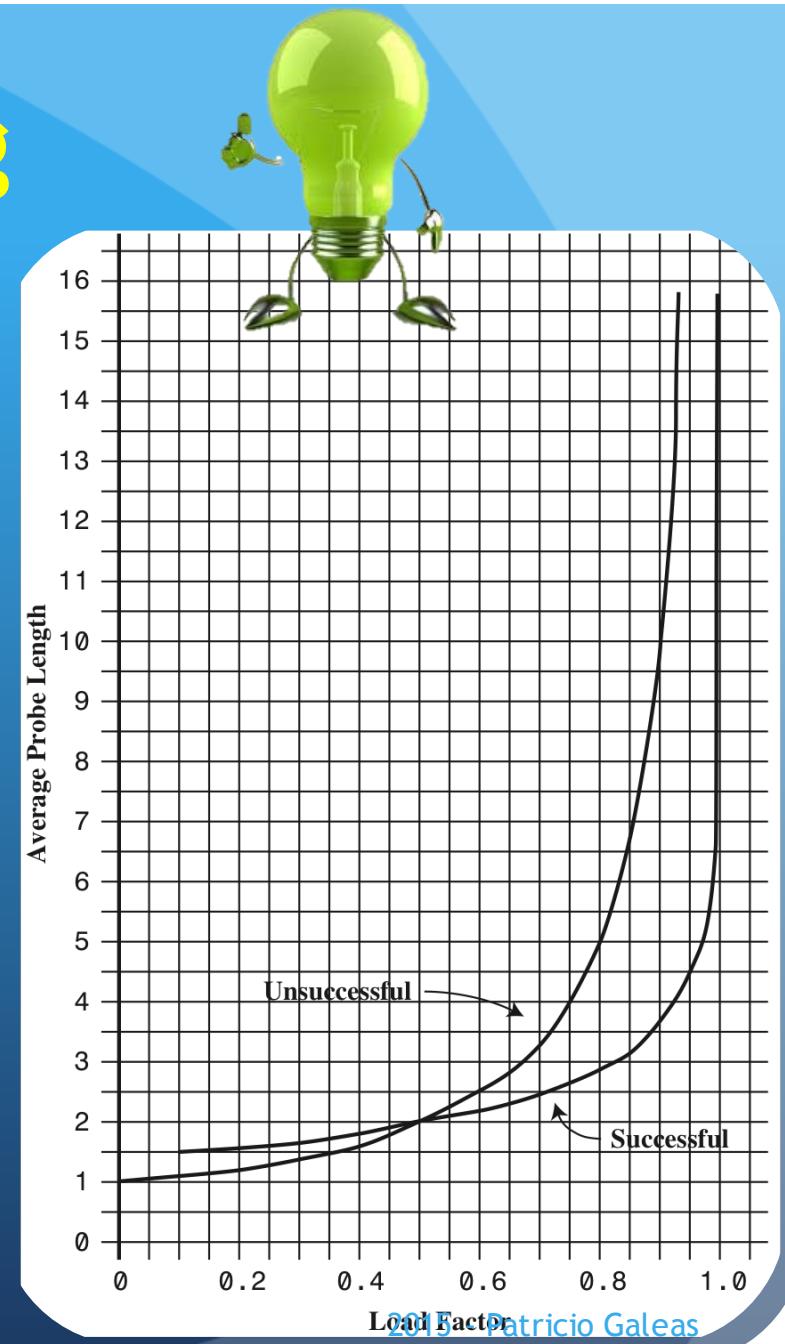
Open Addressing

Prueba Cuadrática y Doble Hashing

Comparten las mismas ecuaciones de rendimiento, las cuales presentan una leve superioridad sobre la Prueba Lineal:

- Para una búsqueda exitosa:
 $-\log_2(1-\text{loadFactor}) / \text{loadFactor}$
- Para una búsqueda no exitosa:
 $1 / (1-\text{loadFactor})$

Sin embargo, la Prueba Cuadrática y Doble Hashing toleran factores de carga superiores a los de la Prueba Lineal.



Eficiencia del Hashing

Separate Chaining

El cálculo de la eficiencia en este caso es distinto en incluso más fácil que el de Open Addressing.

- La idea es calcular el tiempo de búsqueda e inserción.
- Lo que más tarda es la comparación del valor buscado con los valores de los items.
- Además se estima que el tiempo para encontrar el índice (hash) o para determinar si el fin de la lista se ha alcanzado es equivalente a una comparación
- Entonces, todas las operaciones requieren: $O(1 + nComps)$

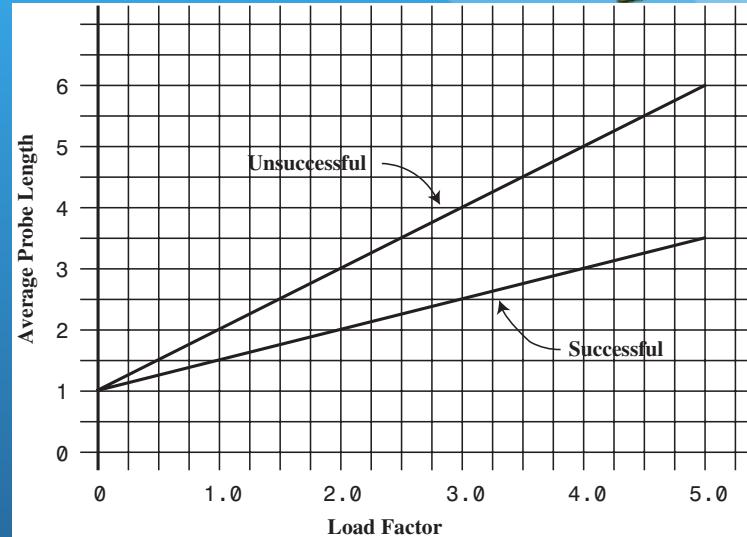


Eficiencia del Hashing

Separate Chaining



- Supongamos que la tabla hash cuenta con **arraySize** elementos.
- Cada uno de los cuales se asocia a una **lista**.
- Y se han insertado **N** items a la tabla.
- Luego en promedio cada lista tendrá **AverageListLength** elementos:
$$\text{AverageListLength} = N / \text{ArraySize}$$
- Formula que es identica al del Factor de Carga:
$$\text{LoadFactor} = N / \text{ArraySize}$$



- Búsqueda exitosa:
 $O(1 + \text{loadFactor} / 2)$
- Búsqueda no exitosa (lista no ordenada)
 $O(1 + \text{loadFactor})$
- Inserción (lista no ordenada)
 $O(1)$
- Inserción (lista ordenada)
 $O(1 + \text{loadFactor} / 2)$

Open Addressing v/s Separate Chaining



- Supongamos que la tabla hash cuenta con **arraySize** elementos.
- Cada uno de los cuales se asocia a una **lista**.
- Y se han insertado **N** items a la tabla.
- Luego en promedio cada lista tendrá **AverageListLength** elementos:
$$\text{AverageListLength} = N / \text{ArraySize}$$
- Formula que es identica al del Factor de Carga:
$$\text{LoadFactor} = N / \text{ArraySize}$$

Resumen

- Las tablas Hash se basan en arreglos
- El rango de valores a almacenar es normalmente mayor al tamaño del arreglo.
- Los valores se son indexados de acuerdo a una función hash.
- Los diccionarios tradicionales es un ejemplo típico de una base de datos que puede ser manejada en una tabla hash.
- La indexación de un valor en una posición ocupada del arreglo se denomina “colisión”.
- Las colisiones pueden ser manejadas principalmente de dos formas: Open Addressing y Encadenamiento Separado.
- En Open Addressing los valores en colisión son asociados a otra celda dentro del arreglo.
- En el Encadenamiento Separado, cada elemento del arreglo consiste en una lista enlazada y todos los elementos que son indexados en el mismo índice son insertados a la lista correspondiente.
- Existen 3 mecanismos de prueba en Open Addressing: Lineal, Cuadrático y Doble Hashing.
- En la prueba lineal el tamaño del paso es siempre 1.



Resumen

- El número de paso necesarios para encontrar un ítem determinado se denomina el Largo de la Prueba.
- En la prueba lineal, van apareciendo grupos de cedas con valores. A estos grupos se les denomina clusters, los cuales reducen la eficiencia.
- En la prueba cuadrática el tamaño de la prueba es el cuadrado del número del paso. Es decir la prueba va a $x, x+1, x+4, x+9$, etc.
- La prueba cuadrática elimina los clusters primarios pero sufre de los menos problemáticos clusters secundarios.
- Los clusters secundarios ocurren porque todos los valores que se indexan en el mismo índice siguen la misma secuencia de pasos durante la prueba.
- El hashing doble el tamaño del paso depende del valor indexado y se obtiene de una segunda función de hashing.
- Si en hashing doble, la segunda función de hashing retorna el valor “ s ” la prueba va a $x, x+s, x+2s, x+3s, x+4s$, etc. Lo que depende del valor indexado, pero se mantiene constante durante la prueba.



Resumen

- El Factor de Carga es la relación entre la cantidad de ítems en una tabla hash y el tamaño de arreglo de la tabla.
- El factor de carga máximo en Open Addressing debiera ser alrededor de 0,5.
- En Open Addressing el Factor de Carga el tiempo de búsqueda tiene a infinito si el factor de carga tiende a 1.
- Es crucial que en una tabla hash con Open Addressing no se llenen demasiado.
- El Factor de Carga 1 es apropiado para encadenamiento Separado.
- El largo de la prueba en Encadenamiento Separado crece linealmente al Factor de Carga.
- Un string puede ser indexado multiplicando cada carácter por la potencia de una constante, sumando el producto, y usando el operador módulo para reducir el resultado al tamaño de la tabla hash.
- El tamaño de las tablas hash debiera ser un numero primo. Esto es especialmente importante en la prueba cuadrática y doble hashing.



Experimentos

- Con el applet genere una pequeña tabla hash cuadrática donde su tamaño no es un número primo, por ej. 24. Llene la tabla con 16 items y luego busque por items no existentes, hasta que el algoritmo entre en un ciclo infinito.
- Modifique el programa hash.java (11.1) para que use prueba cuadrática.
- Implemente una tabla hash con prueba lineal que guarde strings.

