

# PAII-19: tablas de hash (2)

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Abril 2008



# Outline

- 1 Resolución de las colisiones
- 2 Tablas de hash dinámicas
- 3 Hash y STL



# Outline

- 1 Resolución de las colisiones
- 2 Tablas de hash dinámicas
- 3 Hash y STL



# Colisiones: resolución

*Open addressing*: cuando  $\alpha < 1$ , la filosofía es **aprovechar del espacio libre dentro del arreglo** para buscar lugares donde poner las llaves cuando hay colisión

- **seguir una ley dada para efectuar “sondeos”** (probing) para detectar lugares disponibles
- la ley tiene  $h^{(k)}(v)$  se toma generalmente como:

$$h^{(k)}(v) = (h(v) + c(k)) \mod N$$

donde  $c(k)$  satisface:

$$\begin{cases} c(0) = 0 \\ \text{los } c(k) \mod N \text{ permiten alcanzar todo entero en } [0, m-1] \end{cases}$$

- la ley de sondeo mas simple es una **ley lineal** ( $c(k)$  lineal)



# Colisiones: resolución

Para **manejar elementos quitados**:

- usar *flag* (interpretado diferentemente en inserción/búsqueda)
- si queremos evitar tener sobre-costos en memoria

```
void remove(Item x) {
    int i = hash(x.key(), M), j;
    while (!st[i].null())
        if (x.key() == st[i].key()) break;
        else i = (i+1) % M;
    if (st[i].null()) return; // No esta la llave
    st[i] = nullItem; N--;
    for (j=i+1; !st[j].null(); j=(j+1)%M, N--) {
        Item v = st[j]; st[j] = nullItem; insert(v); }
}
```

o sea reinsertar los elementos que vamos a encontrar a la derecha (incluso unos que no sean de mismo valor de *hash*)



# Colisiones: resolución

*Linear probing* sobre una llave  $v$  en un arreglo de tamaño  $N$ , con sondeos siguiendo la ley:

$$h^{(k)}(v) = (h(v) + k) \bmod N$$

hasta que el valor encontrado corresponda a un lugar disponible.

3,7,23,34,50,27,14,12

	23		3		27	50	7			
0	1	2	3	4	5	6	7	8	9	10



# Colisiones: resolución

*Linear probing* sobre una llave  $v$  en un arreglo de tamaño  $N$ , con sondeos siguiendo la ley:

$$h^{(k)}(v) = (h(v) + k) \bmod N$$

hasta que el valor encontrado corresponda a un lugar disponible.

3,7,23,34,50,27,14,12

	23	34	3		27	50	7			
0	1	2	3	4	5	6	7	8	9	10



# Colisiones: resolución

*Linear probing* sobre una llave  $v$  en un arreglo de tamaño  $N$ , con sondeos siguiendo la ley:

$$h^{(k)}(v) = (h(v) + k) \bmod N$$

hasta que el valor encontrado corresponda a un lugar disponible.

3,7,23,34,50,27,14,12

	23	34	3	14	27	50	7			
0	1	2	3	4	5	6	7	8	9	10





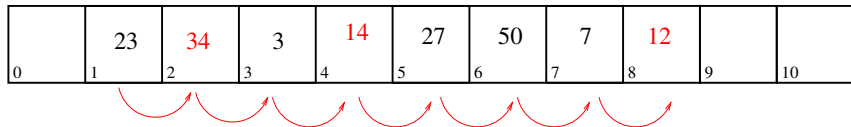
# Colisiones: resolución

*Linear probing* sobre una llave  $v$  en un arreglo de tamaño  $N$ , con sondeos siguiendo la ley:

$$h^{(k)}(v) = (h(v) + k) \bmod N$$

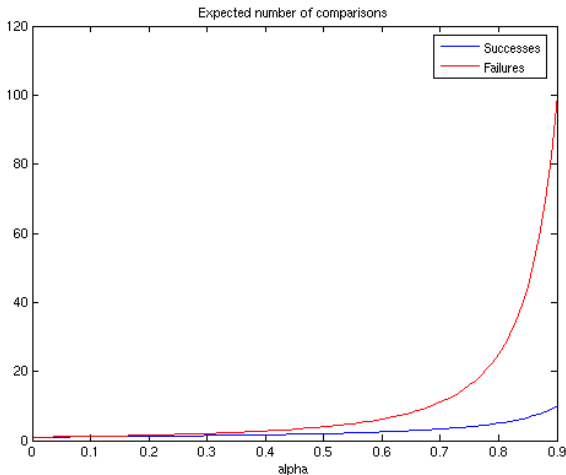
hasta que el valor encontrado corresponda a un lugar disponible.

3,7,23,34,50,27,14,12



# Colisiones: resolución

Evolución del numero promedio de comparaciones:  $\frac{1}{1-\alpha}$  para éxitos y  $\frac{1}{(1-\alpha)^2}$  para fracasos (o inserciones!)



# Colisiones: resolución

- Se puede aceptar o no llaves duplicadas, ambos están soportados por el mecanismo
- Notar los **casos peores**: por ejemplo al hacer inserción de  $N$  objetos, sería la complejidad cuadrática. Por qué?
- Ahora, con un uso razonable y  $\alpha < 1$  estamos en  $O(1)$



# Colisiones: resolución

- El problema mas importante es que **rompemos la filosofía fundamental de tablas de hash**: investigamos eventualmente objetos con varias valores de *hash* mientras hacemos el sondeo
- Además, la inserción de una llave de valor de *hash* dado puede tener consecuencia sobre el desempeño de otras valores de *hash*, por agrupación: **fenómeno de clustering** (y sería igual con saltos de 2,3...)
- Idea para sacarse de este problema: **sondeo no-linear**



# Colisiones: resolución

Por qué este  $\frac{1}{1-\alpha}$ ?



# Colisiones: resolución

Por qué este  $\frac{1}{1-\alpha}$ ?

- Probabilidad de tener un lugar ocupado:  $\alpha$
- Probabilidad de tener un lugar no ocupado:  $1 - \alpha$
- Numero promedio de sondeos antes de llegar a un no ocupado:  
 $\frac{1}{1-\alpha}$
- No sigue exactamente esa ley por el efecto de *clustering*! (difícil que estudiar)



# Colisiones: resolución

Primero mejoramiento:

- Usar una **constante  $K$**  al lugar del 1 y aumentar el índice de  $K$  en cada nuevo sondeo
- Cuidado! Si  $K$  tiene divisor común con  $M$  no se podrá recorrer todas las celdas del arreglo.
- $K$  tiene que ser **primo relativamente a  $M$ !**
- Se alivia el problema de *clustering* primario pero sigue presente, aunque disminuido! Las secuencias a partir de valores de **hash** espaciados de  $K$  son víctimas del fenómeno



# Colisiones: sondeo pseudo-aleatorio

Otra idea de mejoramiento: usar una secuencia pseudo-aleatoria

- generar una **permutación aleatoria** de  $[0 \dots N - 1]$ :  $[s_0 \dots s_{N-1}]$
- usar la ley

$$h^{(k)}(v) = (h(v) + s_k) \mod N$$

- desaparece el efecto de *clustering* primario pero hay el problema que las secuencias generadas son las mismas para todo valor de *hash* inicial





# Colisiones: sondeo cuadrático

La primera idea que vendría es la de usar una función no-lineal para determinar el paso, como una función cuadrática por ejemplo:

$$h^{(k)}(v) = (h(v) + k^2) \mod N$$

o mas generalmente

$$h^{(k)}(v) = (h(v) + ak + bk^2) \mod N$$

Claramente se puede tener  $c(0) = 0$ , pero es mas difícil ver lo de la segunda propiedad. De hecho esta propiedad no esta verificada en general!



# Colisiones: sondeo cuadrático

Un sondeo cuadrático en una tabla de *hash* de tamaño  $N$ ,  $N$  primo, es tal que los  $\lfloor \frac{N}{2} \rfloor$  primeros sondeos se hacen sobre índices distintos.

Se prueba por contradicción: si existe  $0 \leq k < l < \lfloor \frac{N}{2} \rfloor$  tal que

$$h^{(k)}(v) = h^{(l)}(v)$$

entonces

$$k^2 \mod N = l^2 \mod N$$

y

$$(k + l)(k - l) \mod N = 0 <$$

Eso (ya que  $N$  es primo) implica  $k = l$ , imposible porque  $k \neq l$  o  $k + l \mod N = 0$  que es otra vez imposible porque  $k$  y  $l$  son inferiores  $\lfloor \frac{N}{2} \rfloor$ .



# Colisiones: sondeo cuadrático

## Consecuencias:

- hay que limitar los sondeos a una búsqueda de tamaño total  $\lfloor \frac{N}{2} \rfloor$  si queremos evitar ciclos infinitos
- para que funcione eficientemente el sondeo en este caso, **sin fenómeno de *clustering***, queremos evitar de caer durante el sondeo sobre un elemento ocupado por otra llave: necesitamos

$$n < \lfloor \frac{N}{2} \rfloor$$

o sea

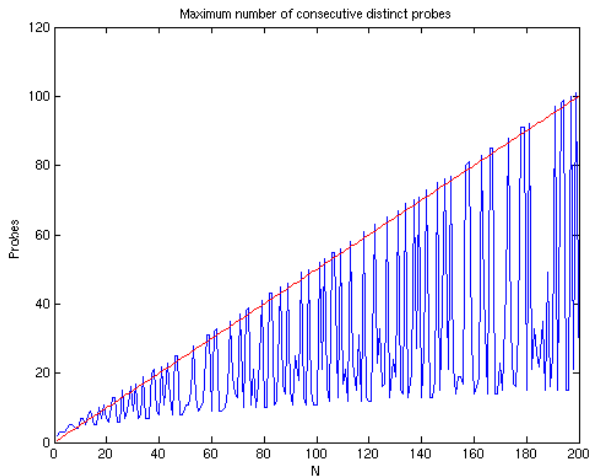
$$\alpha < \frac{1}{2}$$

- si  $N$  no es primo, no podemos garantizar el resultado



# Colisiones: sondeo cuadrático

Ese mismo fenómeno se puede ver claramente:



# Colisiones: sondeo cuadrático

- + **Mejor desempeño** que el sondeo linear (en numero de comparaciones que hacer en promedio)
- + Si el arreglo esta mitad vacío y el tamaño primo, garantizamos que podemos hacer la inserción de una llave sin repasar por el mismo índice
- + **Quita parcialmente el efecto de *clustering*** del sondeo linear (*clustering* “primario”)
  - Limita el espacio útil a la mitad del arreglo; sin esto el desempeño se degrada
  - El clustering no desaparece completamente ya que de todos modos los caminos siguen las mismas secuencia de índices



# Colisiones: sondeo cuadrático

Una remarca: se podría tener a priori problemas con el **calculo de  $i * i$** , tanto por costo que por riesgo de *overflow*. Pero hay una manera de evitarlo completamente!

$$\begin{aligned} h^{(k)}(v) &= (h(v) + k * k) \mod N \\ h^{(k-1)}(v) &= (h(v) + (k-1) * (k-1)) \mod N \end{aligned}$$

entonces

$$h^{(k)}(v) = (h^{(k-1)}(v) + 2 * k - 1) \mod N$$

Calcular los valores de *hash* sucesivos incrementalmente!



# Colisiones: sondeo cuadrático

Queda el problema de *clustering* secundario:

- *clustering* primario: el sondeo linear tiene como efecto de ir **agregando elementos a los *clusters* ya formados en cada nueva colisión**; creando posibilidad de mas colisión, y creando *clusters* mas grandes que se pueden fusionar
- *clustering* secundario: es el efecto (que existe en ambos, linear o cuadrático) de que para **llaves diferentes**  $v_1$  y  $v_2$  tal que  $h(v_1) = h(v_2)$  las secuencias que se siguen a partir del valor de *hash* son idénticas.

Ambos aumentan la complejidad de las operaciones basicas  
Para el segundo, nos ayudaría una función  **$c(k)$  que no solo dependería de  $k$  sino también de  $v$ !**



# Colisiones: doble hash

La idea de **doble hash** es basada en ese ultimo principio:

$$h^{(k)}(v) = (h(v) + k * h'(v)) \mod N$$

- Usar una **segunda función de hash**,  $h'(v)$  diferente de la primera, para determinar el incremento que usar para repartir las llaves
- Esta vez, el recorrido hecho por un par de llaves diferentes **no estará el mismo**, excepto en caso de doble colisión (para  $h$  y  $h'$ )
- Cuidado a los valores de  $h'(v)$ !





# Colisiones: doble hash

Elección de  $h'(v)$

- No se puede tolerar  $h'(v) = 0$ , por qué?
- Cada valor tiene que ser primo relativamente a  $N$ , por qué?
- para asegurarse de eso: por ejemplo elegir  $N$  primo y usar  $h'(v)$  que regrese valores inferiores a  $N$ ; otra solución es  $N = 2^m$  y  $h'(v)$  regresando números impares



# Colisiones: doble hash

## Elección de $h'(v)$

- No se puede tolerar  $h'(v) = 0$ , por qué?
- Cada valor tiene que ser primo relativamente a  $N$ , por qué? Si  $N = pq$  y  $h'(v) = pr$  entonces después de  $q < \frac{N}{2}$  sondeos

$$h^{(q)}(v) = (h(v) + q * h'(v)) \mod N = h(v)$$

- para asegurarse de eso: por ejemplo elegir  $N$  primo y usar  $h'(v)$  que regrese valores inferiores a  $N$ ; otra solución es  $N = 2^m$  y  $h'(v)$  regresando números impares



# Colisiones: doble hash

Elección de  $h'(v)$ : ejemplos

- $h'(v) = p - (v \bmod p)$ ,  $p < N$  primo
- $h'(v) = 1 + (v \bmod p)$ ,  $p < N$  primo

Tomar por ejemplo

- $N = 13$
- $h(v) = v \bmod 13$
- $h'(v) = 7 - v \bmod 7$

Insertar 18, 41, 22, 59, 32, 31, 73, 44



# Colisiones: doble hash

18, 41, 22, 59, 32, 31, 73, 44

		41			18	32	59		22			
0	1	2	3	4	5	6	7	8	9	10	11	12



# Colisiones: doble hash

18, 41, 22, 59, 32, 31, 73, 44

		41			18	32	59		22			
0	1	2	3	4	5	6	7	8	9	10	11	12


$$h'(31) = 4$$



# Colisiones: doble hash

18, 41, 22, 59, 32, 31, 73, 44

31		41			18	32	59		22			
0	1	2	3	4	5	6	7	8	9	10	11	12


$$h'(31) = 4$$

$$h'(31) = 4$$



# Colisiones: doble hash

18, 41, 22, 59, 32, 31, 73, 44

31		41			18	32	59	73	22			
0	1	2	3	4	5	6	7	8	9	10	11	12

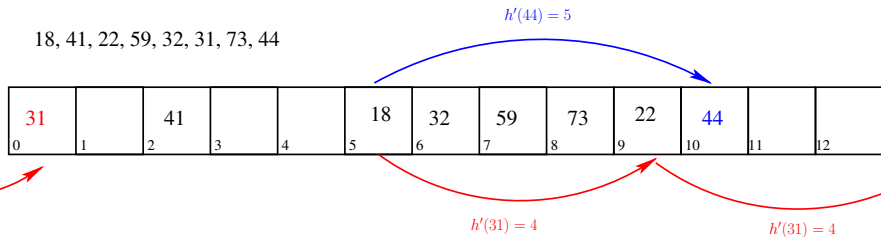

$$h'(31) = 4$$

$$h'(31) = 4$$



# Colisiones: doble hash

18, 41, 22, 59, 32, 31, 73, 44





# Colisiones: doble hash

Implementación igual a la del sondeo linear

```
void insert(Item item)
{
    Key v = item.key();
    int i = hash(v, M), k = hashtwo(v, M);
    while (!st[i].null()) i = (i+k) % M;
    st[i] = item; N++;
}

Item search(Key v) {
    int i = hash(v, M), k = hashtwo(v, M);
    while (!st[i].null())
        if (v == st[i].key()) return st[i];
        else i = (i+k) % M;
    return nullItem;
}
```



# Colisiones: doble hash

El **desempeño es mejor que en los otros casos** (sobre todo para  $\alpha > \frac{1}{2}$ ), se puede mostrar que el numero promedio de sondeos (comparaciones) que hacer en una búsqueda es:

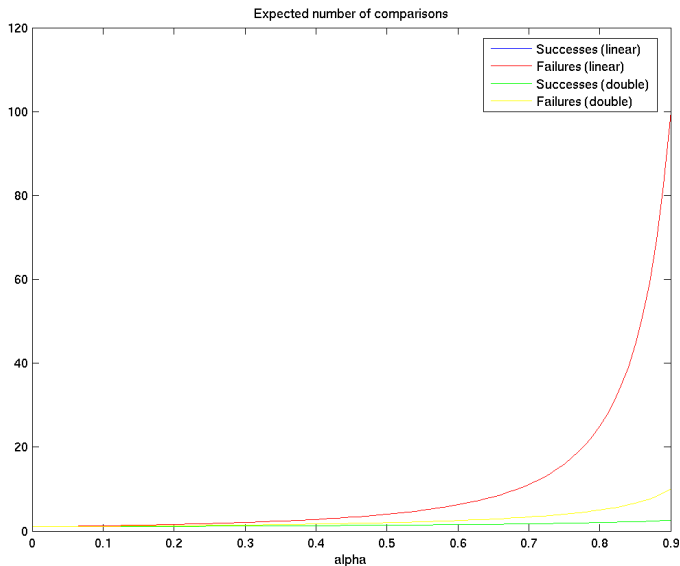
$$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right) \text{ si la búsqueda es exitosa}$$

y

$$\frac{1}{1-\alpha} \text{ si la búsqueda se acaba con fracaso}$$



# Colisiones: doble hash



# Colisiones: doble hash

Expresado de otra manera:

- se puede garantizar de que no haremos mas de  $t$  sondeos con

$$\alpha < 1 - \frac{1}{\sqrt{t}}$$

en el caso de **sondeo linear**

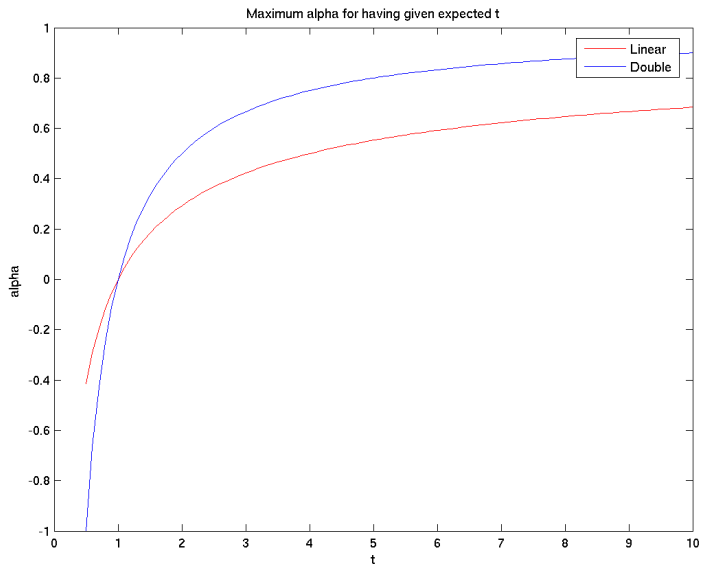
- se puede garantizar de que no harémos mas de  $t$  sondeos con

$$\alpha < 1 - \frac{1}{t}$$

en el caso de sondeo por **doble hash**



# Colisiones: doble hash



# Tablas de hash, en resumen

- Peor caso en  $O(n)$  para búsquedas, quitar nodo y inserciones
- Peor caso evitable por la elección de una buena función de *hash*
- En promedio, desempeño en  $O(1)$
- Muy rápido mientras  $\alpha \ll 1$
- Crecimiento de  $\alpha$  que penaliza el desempeño (eso es una diferencia con las otras estructuras)
- Largamente usado en sistemas reales, por ejemplo para verificar la existencia (por declaración previa) de variables para un compilador



# Outline

- 1 Resolución de las colisiones
- 2 Tablas de hash dinámicas**
- 3 Hash y STL



# Tablas dinámicas

- El **desempeño de las tablas de *hash* se degrada**, por las colisiones que ocurren, mientras añadimos objetos en la llave: con listas ligadas o con open addressing. . .
- Podemos llegar aun en un caso critico al rellenar la tabla, por ejemplo en el caso de re-hash por sondeo
- Una solución en este caso es **aumentar el tamaño de la tabla por multiplicación de esa por dos**: re-hash





# Tablas dinámicas

Cuando hacerlo?

- cuando esta la tabla a medio rellena?
- cuando una inserción fracasa?
- cuando un valor de  $\alpha$  esta alcanzado?



# Tablas dinámicas

- Es una operación costosa: **todas las llaves** tienen que estar bajo hash otra vez
- Análisis **amortizada** simple permite establecer que el costo ligado a los re-hash al momento de doblar el arreglo están amortizados por las otras operaciones, de inserción por ejemplo
- Como después de un doblamiento tenemos

$$\frac{1}{4} < \alpha < \frac{1}{2}$$

se deduce que el número de sondeos realizados se va a quedar en promedio abajo de 2 o 4 (éxitos/fracasos). Entonces, como hasta la próxima operación de este tipo vamos a insertar  $\frac{N}{4}$  objetos, y como la re-inserción implica  $\frac{N}{2}$  objetos, se puede dar un **costo amortizado doble del costo de cada inserción normal**



# Tablas dinámicas: implementación

Doblar cuando  $\alpha > \frac{1}{2}$

```
private:
    void expand() {
        Item *t = st; init(M+M);
        for (int i = 0; i < M/2; i++)
            if (!t[i].null()) insert(t[i]);
        delete t;
    }
public:
    ST(int maxN){ init(4); }
    void insert(Item item)
    { int i = hash(item.key(), M);
      while (!st[i].null()) i = (i+1) % M;
      st[i] = item;
      if (N++ >= M/2) expand();
    }
```



# Tablas dinámicas

Ejemplo practico: el utilitario rehash:

- en los sistemas Unix, el contenido de los directorios “visibles por default” (exec path) esta **organizado en una tabla de hash**:
  - llaves: nombre (corto) del ejecutable
  - objetos asociados: el camino (path) completo
- al añadir nuevos softwares el utilitario rehash re-lee este contenido y regenera la tabla de hash, eventualmente con aumentación del tamaño



# Outline

- 1 Resolución de las colisiones
- 2 Tablas de hash dinámicas
- 3 Hash y STL**



# Tablas de hash en la STL

Existe una implementación en la STL aunque no hace parte del “core” de la STL: esta considerado como extensión

- HASHTABLE: estructura básica
- HASH\_SET, HASH\_MULTISSET
- HASH\_MAP, HASH\_MULTIMAP

Los últimos son implementaciones de SET y MAP con tablas de *hash* (contrariamente a las versiones estándar que usan **arboles equilibrados**)



# Tablas de hash en la STL

- Están en **otro espacio de nombre** que STD:  
`using namespace __gnu_cxx ;`
- Pronto estarán **añadidos a la std** con nombres diferentes:  
`UNORDERED_SET, UNORDERED_MAP...` Ver por ejemplo el  
archivo `c++/4.0.0/tr1/unordered_map`
- Son contenedores clásicos **con iteradores**



# Tablas de hash en la STL: hashtable

Es la **estructura interna**, que es un *template*:

```
hashtable<tipo,llave,hash_fun,equalKey_fun,extractKey_f
```

- Los tres últimos parámetros son **objetos funciones** (clases con operador ())
- HASH\_FUN: función de *hash* (regresa un entero a partir de una llave cuyo tipo es parte del template)
- EQUALKEY\_FUN: define si dos llaves son iguales
- EXTRACTKEY\_FUN: extrae una llave a partir de un objeto de tipo TIPO
- Mejor no usarle directamente y usar las clases con interfaces MAP, SET





# Tablas de hash en la STL

Al dar un vistazo a HASHTABLE.H:

```
static const unsigned long __stl_prime_list[ __stl_num_
53ul , 97ul , 193ul , 389ul , 769ul ,
1543ul , 3079ul , 6151ul , 12289ul , 24593ul ,
49157ul , 98317ul , 196613ul , 393241ul , 786433ul ,
1572869ul , 3145739ul , 6291469ul , 12582917ul , 25165843ul ,
50331653ul , 100663319ul , 201326611ul , 402653189ul , 80530
1610612741ul , 3221225473ul , 4294967291ul
};

inline unsigned long __stl_next_prime(unsigned long __
const unsigned long* __first = __stl_prime_list;
const unsigned long* __last = __stl_prime_list +
    (int) __stl_num_primes;

...
```

Lista pre-computada de primos...



# Tablas de hash en la STL

Otra cosa que se puede remarcar:

```
iterator find(const key_type& __key) {  
    size_type __n = _M_bkt_num_key(__key);  
    _Node* __first;  
    for (__first = _M_buckets[__n];  
         __first && !_M_equals(_M_get_key(__first->_M  
         __first = __first->_M_next) {}  
    return iterator(__first, this);  
}
```

Qué se puede deducir de este pedazito?



# Tablas de hash en la STL: hash\_map

`hash_map<Key, Data, HashFcn, EqualKey, Alloc>`

- Contenedor asociativo que asocia objetos de tipo `SKEY` a objetos de tipo `DATA`
- No dos elementos tienen la misma llave
- Misma interfase que el `MAP`
- Usarle cuando el **orden de los datos no importa tanto**
- No `EXTRACTKEY`, ya que los objetos vienen por pares!



# Tablas de hash en la STL: hash\_map

```
#include <string>
#include <ext/hash_map>
using namespace std;
using namespace __gnu_cxx ;
class hash_fun{
public :
    size_t operator()(const string & v) const {
        return (v.size()); // Not very good!
    }
};
class equalKey_fun{
public :
    bool operator()(const string & v1, const string & v2)
        return (v1 == v2);
    }
};
```



# Tablas de hash en la STL: hash\_map

Anuario:

```
int main() {  
    hash_fun          f_hash ;  
    equalKey_fun      f_equal ;  
    hash_map<string , long , hash_fun , equalKey_fun> teltab ;  
    teltab [ " Pedro" ]      = 1175512 ;  
    teltab [ " Paco" ]       = 2395401 ;  
    teltab [ " Lupe" ]       = 7689134 ;  
    teltab [ " Yasmina" ]    = 3211169 ;  
}
```



# Tablas de hash en la STL: hash\_map

Notar que existen objetos funciones de hash para los tipos estandar:

```
struct eqstr {  
    bool operator()(const char* s1, const char* s2) const {  
        return strcmp(s1, s2) == 0;  
    }  
};
```

```
int main() {  
    hash_map<const char*, int, hash<const char*>, eqstr>  
        months;  
    months["january"] = 31; // Inserciones  
    months["february"] = 28;  
    months["march"] = 31;  
    months["april"] = 30;  
    months["may"] = 31;
```



# Tablas de hash en la STL: hash\_map

```
months["june"] = 30;  
months["july"] = 31;  
months["august"] = 31;  
months["september"] = 30;  
months["october"] = 31;  
months["november"] = 30;  
months["december"] = 31;
```

```
cout << "september_->_" << months["september"] << endl;  
cout << "april_----->_" << months["april"] << endl;  
cout << "june_----->_" << months["june"] << endl;  
cout << "november_->_" << months["november"] << endl;  
}
```



# Tablas de hash en la STL: hash

- template definido para tipos básicos
- implementación **muy sencilla** que hay que cambiar según las necesidades

```
inline size_t __stl_hash_string(const char* __s) {  
    unsigned long __h = 0;  
    for ( ; *__s; ++__s)  
        __h = 5*__h + *__s;  
    return size_t(__h);  
}
```

