7. Tablas hash

7.1 Introducción

En nuestro camino por el mundo de las estructuras de datos, uno de los objetivos que nos ha hecho estudiar diferentes estructuras es la eficiencia que conlleva la operación de buscar un elemento en un conjunto. Hasta el momento la mejor eficiencia que hemos obtenido es $O(log_2(n))$. Esta eficiencia la hemos obtenido con dos estructuras de datos:

- En un vector ordenado aplicando una búsqueda binaria
- En un árbol binario de búsqueda equilibrado (p.e AVL)

Cabe destacar el interés a lo largo de la literatura sobre la mejora de la operación de búsqueda en un conjunto cuando el contexto en el que se trabaja son con bases de datos muy grandes.
Con tal fin vamos a estudiar una nueva estructura de datos, **Tablas Hash**, que mejoran la eficiencia de la operación de búsqueda hasta $O(1)$. Esta estructura se caracteriza porque intenta asignar a cada elemento del conjunto una *única* posición dentro de una tabla y a cada posición de la tabla un único elemento. Al tener la tabla un espacio finito habrá elementos a los que se le asignan una misma posición en la tabla y por lo tanto se producirá una colisión. Esta estructura prevee esta posibilidad y por lo tanto veremos distintos mecanismos para resolver estas colisiones.

7.2 Objetivo

Dado un conjunto de datos identificados por una clave (k) se quiere obtener una función $h(k)$ (*función hash*) que nos dé la posición dentro de una tabla (*tabla hash*) en la que almacenamos un par con la clave y la dirección en un fichero donde se encuentra la información asociada a esa

clave.

Ejemplo 7.2.1

	Tabla hash	
h(k)	k	Posición dentro del fichero
0	239	i
1	500	n
	.	
	.	
	.	
n	733	1

	Fichero	
	k	Contenido
	.	
1	733	bla bla bla bla bla
	.	
	.	
	.	

Para obtener la información asociada a la clave $k = 733$ sólo tendríamos que aplicar la función hash para obtener la dirección del fichero en la que se guarda su información, es decir $h(733) = n$. Vamos a la tabla a la posición n y en esta posición consultamos la dirección en el fichero donde tenemos el resto de la información. Así para la clave $k = 733$, aplicamos $h(733) = n$. En la posición n de la tabla si consultamos la posición dentro del fichero nos reconduce a la posición 1 del fichero. En la posición 1 del fichero tenemos toda la información del regitro con valor $k = 733$. □

La función hash cuesta calcularla $O(1)$, lo que hace que nuestra operación de búsqueda sea $O(1)$. El problema de las tablas hash es que a veces diferentes claves tiene el mismo valor hash y por lo tanto se produce colisiones. Y en segundo lugar se debe activar mecanismos para no generar tablas muy descompensadas con respecto al número de datos que queremos almacenar.

7.2.1 Colisiones

Se dan cuando para dos claves diferentes la función hash devuelve el mismo valor, es decir, para las claves k_1 y k_2 siendo $k_1 \neq k_2$ ocurre colisión si $h(k_1) = h(k_2)$. Debemos buscar funciones hash que produzcan el menor número de colisiones. Además las funciones hash deben devolver valores en el rango de índices de mi tabla. Otro requisito es que la tabla tenga un tamaño que sea idóneo para los datos que queremos almacenar. Por un lado para que ocurran menos colisiones se exigiría que la tabla fuese muy grande, pero desperdiciamos mucho espacio si el número de datos es pequeño frente a la dimensión de la tabla. Por otro lado si la tabla es muy pequeña frente al número de datos que queremos almacenar ocurrirán muchas colisiones.

7.2.2 Características de las tablas hash

1. La función hash debe ser rápida de calcular
2. La función hash debe producir el menor número de colisiones
3. El tamaño de la tabla hash debe ser adecuado

Ejemplo 7.2.2

Tenemos el siguiente fichero de datos:

	Fichero de Datos	
	k	Nombre
0	12	Abad Ruiz
1	21	Bernabe Pérez
2	68	Carrasco Ruiz
3	38	Domingo Coca
4	52	Fdez Sánchez
5	70	Juan Ruiz
6	44	Martín Pérez
7	18	Pérez Galiano

Sobre estos 8 registros, vamos a construir una tabla hash con 11 posiciones, ya que es recomendable que el tamaño de la tabla hash sea un número primo para evitar colisiones. Por tanto tenemos lo siguiente:

1. $M = 11$ posiciones en mi tabla hash que van en el rango $[0, M-1]$
2. $h(k) = k\%M$ función hash

La función módulo ($\%$) hace que el rango de valores siempre este entre $[0, M-1]$ para cualquier valor de k. A continuación vamos a calcular las distintas posiciones asociadas a cada clave (k) y a rellenar la tabla hash:

1. $h(12) = 12\%11 = 1$
2. $h(21) = 21\%11 = 10$
3. $h(68) = 68\%11 = 2$
4. $h(38) = 38\%11 = 5$
5. $h(52) = 52\%11 = 8$
6. $h(70) = 70\%11 = 4$
7. $h(44) = 44\%11 = 0$
8. $h(18) = 18\%11 = 7$

Una vez calculadas las posiciones dentro de la tabla hash, el resultado sería este:

	k	Dirección
0	44	6
1	12	0
2	68	2
3		
4	70	5
5	38	3
6		
7	18	7
8	52	4
9		
10	21	1

Para buscar, por ejemplo, $k = 52$ aplicamos $h(52) = 52\%11 = 8$. Nos da la dirección 4 y accedemos al disco. La búsqueda se realiza en $O(1)$. Si el resultado es una casilla vacía, no tendríamos nada en dicha casilla y por lo tanto no accedemos al fichero. □ Cuando ocurra

una colisión deberemos resolverla. Los mecanismos para resolver colisiones se engloblan en las técnicas de *hashing abierto* ó *hashing cerrado*.

7.2.3 Hashing abierto

Para arreglar los problemas de colisión, asociamos a cada entrada $h(k)$ una lista donde se ponen todas las claves con igual valor hash $h(k)$. Se debe procurar que las listas no sean muy grandes, cuando lo son debemos redimensionar la tabla. Y esto nos llevará colocar de nuevo todas la claves ya almacenadas en la nueva tabla. Para establecer cuando redimensionar debes atender al *Factor de Carga*.
Factor de Carga:*Razón entre el numero de elementos total y el número de listas. En general las listas se le denominan cubetas. Y puede ser lista u otra estructura contenedora*.
Para agilizar los procesos de búsqueda entre las claves con igual función hash las listas deben estar ordenadas por valor de clave.

Ejemplo 7.2.3
Siguiendo el ejemplo anterior, pero con $M = 5$ y $h(k) = k\%5$ (podemos reducir el tamaño de la tabla siempre que el factor de carga no sea grande), la tabla hash sería de la siguiente forma:

$h(k)$	
0	$\longrightarrow$ $< 70 \| 5 \| X >$
1	$\longrightarrow$ $< 2 \| 1 \| X >$
2	$\longrightarrow$ $< 12 \| 0 \| > \longrightarrow < 52 \| 4 \| X >$
3	$\longrightarrow$ $< 18 \| 7 \| > \longrightarrow < 33 \| 8 > \longrightarrow < 38 \| 3 > \longrightarrow < 68 \| 2 \| X >$
4	$\longrightarrow$ $< 44 \| 6 \| X >$

Podemos tener cualquier estructura para guardar datos en vez de una lista, por ejemplo, un AVL. □

Implementación con un vector de la STL

```
#include <vector>
#include <list>

class Celda {
private:
      int k;      // valor de la clave
      int d;      // direccion en el fichero de datos
public:
      Celda (): k(-1), d(-1) {}
      Celda (int c, int p): k(c), d(p) {}
      int & clave () {return k;}
      int & posicion () {return d;}
};

class TH {
```

```
16 private:
17       vector<list<Celda> > tabla;
18       int fhash (int clave) {
19             return clave % tabla.size();
20       }
21
22       pair<bool,list<Celda>::iterator> Esta (int clave) {
23             list<Celda>::iterator it;
24             int pos = fhash (clave);
25             for (it = tabla[pos].begin(); it != tabla[pos].end()
26                   && (*it).clave() != clave; ++it);
27
28             // al salir del for tenemos dos posibilidades: haber encontrado la
29             // clave o no.
30             bool find = false;
31             if (it != tabla[pos].end()) find = true;
32             return pair<bool, list<Celda>::iterator> (find, it);
33       }
34 public:
35       TH (int tam) {
36             assert (tam > 0);
37             tabla.resize (tam);
38       }
39
40       bool Existe (int clave) {
41             pair<bool,list<Celda>::iterator> a;
42             a = Esta (clave);
43             return a.first;
44       }
45
46       bool Insertar (int clave, int d) {
47             pair<bool,list<Celda>::iterator> a;
48             a = Esta (clave);
49             if (!a.first) {
50                   int pos = fhash (clave);
51                   tabla[pos].push_back (Celda(clave,d));
52                   return true;
53             }
54             else
55                   return false;
56       }
57
58       bool CambiarDir (int clave, int nuevadir) {
```

```
59              pair<bool,list<Celda>::iterator> a;
60              a = Esta(clave);
61              if (a.first) {
62                    // podemos hacer esto porque posicion() devuelve una referencia
63                    (*(a.second)).posicion() = nuevadir;
64                    return true;
65              }
66              else
67                    return false;
68        }
69
70        int ObtenDir (int clave) {
71              pair<bool,list<Celda>::iterator> a = Esta (clave);
72              if (a.first)
73                    return (*(a.second)).posicion ();
74              else
75                    return -1;
76        }
77
78        bool Borrar (int clave) {
79              pair<bool,list<Celda>::iterator> a = Esta (clave);
80              if (a.first) {
81                    int pos = fhash (clave);
82                    tabla[pos].erase (a.second);
83                    return true;
84              }
85              else
86                    return false;
87        }
88
89        friend ostream & operator<< (ostream & os, const TH & T) {
90              vector<list<Celda> >::iterator it1; int pos = 0;
91              for (it1 = T.tabla.begin(); it1 != T.tabla.end(); ++it1; ++pos) {
92                    os << "Datos en posicion " << pos << ':';
93                    list<Celda>::iterator it2;
94                    for (it2 = (*it1).begin(); it2 != (*it1).end(); ++it2)
95                          os << (*it2).clave() << ' ' << (*it2).posicion() << ' ';
96                    os << endl;
97              }
98              return os;
99        }
100 };
```

Ejemplo 7.2.4
Suponed que tenemos el T.D.A. Tabla Hash abierta (unordered_set) (Class TH), en la que la resolución de colisiones se hace utilizando para cada cubo una lista.

```
#include <vector>
#include <list>
using namespace std;

class TH {
private:
      struct info {
            int key; //clave
            int di;  //direccion
      };
      vector<list<info> > data;
      int fhash (int k) const;
      bool recolocar () const;
public:
...
      /* k es la clave, d la direccion asociada para esa clave*/
      void insertar (int k, int d);
...
      class iterator {
      private:
            list<info>::iterator it_cub;
            vector<list<info> >::iterator it;
            ...
      };
      iterator begin ();
      iterator end();
...
};
```

1. Implementar la función **insertar** suponiendo que tenemos implementada la función hash (**fhash**). Después de añadir la nueva clave k y su dirección asociada d, la función **insertar** debe comprobar si es necesario redimensionar la tabla hash. Para ello se supone que tenemos implementada la función **recolocar**. Esta función devuelve verdadero en el caso de que sea necesario pasar todos los datos a una nueva tabla hash y false en caso contrario. El tamaño de la nueva tabla tiene que ser el primo más cercano a $2 \cdot M$ por exceso, siendo M el tamaño original.

```
void TH::insertar (int k, int d) {
      int pos = fhash (k);
      info aux = {k, d};
      data[pos].push_back (aux);
      if (recolocar()) {
```

```
            int nuevo_tam = nextprimo (data.size()*2);
            vector<list<info> > nuevodata (nuevo_tam);
            for (int i=0; i<data.size(); i++) {
                  list<info>::iterator it;
                  for (it = data[i].begin(); it != data[i].end(); ++it) {
                        // hacemos la funcion fhash a mano pues debemos
                        //hacerla con el nuevo tamanio calculado
                        int npos = (*it).key % nuevo_tam;
                        nuevodata[npos].push_back(*it);
                  }
            }
            data = nuevodata;
      }
}
```

2. Implementar la **clase iterator** (un iterador sobre todos los elementos de la tabla hash) de la tabla hash, así como las funciones **begin** y **end** de la clase TH.

```
class iterator {
private:
      list<info>::iterator it_cub;
      vector<list<info> >::iterator it;
      vector<list<info> >::iterator final_tabla;//para implementar ++
      vector<list<info> >::iterator comienzo_tabla;//para implementar --

public:
      iterator & operator++ () {
            // avanzamos it_cub
            ++it_cub;
            // si it_cub es el fin de la lista de la posicion actual
            //de la tabla  avanzamos el iterador que recorre la
            //tabla y ponemos it_cub
            // al inicio de la lista de esa posicion
            while (it_cub == it->end() && it != final_tabla) {
                  ++it;
                  it_cub = it->begin();
            }

            return *this;
      }

      iterator & operator-- () {
           if (it_cub==it->begin() && it==comienzo_tabla){
                it=final_tabla;
                return *this;
```

```
28              }
29              if (it_cub==it->begin() &&it!=comienzo_tabla){
30                  --it;
31                  while (it!=comienzo_tabla && it->begin()==it->end()){
32                      --it;
33                  }
34                  if (it==comienzo_tabla){
35                   if (it->begin()==it->end()){
36                     it=final_tabla;
37                     return *this;
38                   }
39                   else{
40                      it_cub= it->end(); --it_cub;
41                      return *this;
42                   }
43                 }
44                 else{
45                      it_cub= it->end(); --it_cub;
46                      return *this;
47                 }
48             }
49             else{
50               --it_cub;
51               return *this;
52             }
53
54
55
56          }
57
58          info & operator* () {
59                return *it_cub;
60          }
61
62          bool operator == (iterator & i)const {
63                return it == i.it && it_cub == i.it_cub;
64          }
65
66          bool operator != (iterator & i)const {
67                return it != i.it && it_cub != i.it_cub;
68          }
69
70          friend class TH;
```

```
71  };
72
73  iterator begin () {
74        iterator i;
75        i.it = datos.begin();
76        if (i.it!=datos.end())
77          i.it_cub = it->begin();
78        i.final_tabla = datos.end();
79        i.comienzo_tabla= datos.begin();
80        if (i.it_cub == i.it->end())
81              ++i;  // si no hay elementos debemos avanzar a la siguiente
82                    // lista valida
83        return i;
84  }
85
86  iterator end () {
87        iterator i;
88        i.it = datos.end();
89        //i.it_cub = it->end(); No se poden porque it->end no existe
90        i.final_tabla = datos.end();
91        i.comienzo_tabla= datos.begin();
92        return i;
93  }
```

□

7.2.4 Hashing cerrado

Sólo se usa una tabla donde cada elemento se dispone en una única entrada. Lo que se almacena en dicha entrada es:

1. La clave
2. La posición de dicha información dentro del fichero
3. El estado de la casilla, que puede ser libre u ocupado.

El proceso de relleno es igual al de las tablas abiertas, la diferencia está en cómo tratamos las colisiones: en el hashing cerrado se usa el *rehashing*. Las distintas estrategias de rehashing que vamos a ver son:

1. Lineal
2. Doble
3. Sondeo aleatorio

Lineal

Los pasos que se dan son :

1. Aplicar $h(k)$ y, si existe colisión,
2. Aplicar $h_i(k) = (h(k) + (i-1))\,\%M$ siendo $i = 2,3,4...$ hasta encontrar una casilla libre.

Ejemplo 7.2.5
Usando los datos del ejemplo 7.2.2, $h(k) = k\,\%M$, $M = 11$ y $h(k) = (h(k) + (i-1))\,\%M$:

1. $h(12) = 12\,\%11 = 1$
2. $h(21) = 21\,\%11 = 10$
3. $h(68) = 68\,\%11 = 2$
4. $h(32) = 32\,\%11 = 10 \longleftarrow$ Colisión
   a) $h_1(32) = 10$ Este se corresponde con el ya calculado
   b) $h_2(32) = (10+1)\,\%11 = 0$ Como el 0 está libre, insertamos la clave 32 en esa posición.
5. $h(56) = 56\,\%11 = 1 \longleftarrow$ Colisión
   a) $h_2(56) = (1+1)\,\%11 = 2$
   b) $h_3(56) = (1+2)\,\%11 = 3$
6. $h(77) = 77\,\%11 = 0 \longleftarrow$ Colisión
   a) $h_2(77) = (0+1)\,\%11 = 1$
   b) $h_3(77) = (0+2)\,\%11 = 2$
   c) $h_4(77) = (0+3)\,\%11 = 3$
   d) $h_5(77) = (0+4)\,\%11 = 4$
7. $h(91) = 91\,\%11 = 3 \longleftarrow$ Colisión
   a) $h_2(91) = (3+1)\,\%11 = 4$
   b) $h_3(91) = (3+2)\,\%11 = 5$
8. $h(18) = 18\,\%11 = 7$

El rendimiento sería 17, el número de intentos para almacenar todas las claves.
La tabla final quedaría así:

$h(k)$[1]	k	d_i	estado
0	32	3	ocupada
1	12	0	ocupada
2	68	2	ocupada
3	56	4	ocupada
4	77	5	ocupada
5	91	6	ocupada
6			libre
7	18	7	ocupada
8			libre
9			libre
10	21	1	ocupada

A la hora de consultar debemos tener en cuenta si una casilla está libre por no haberle insertado ningún valor aún o porque su valor ha sido borrado. Si vemos una casilla que ha sido borrada en una consulta, la tomaremos como ocupada pues si no, nuestra búsqueda concluirá en ese punto.
□

[1] Recordad que h(k) es la función hash aplicada a la clave k que se corresponde con el índice de la tabla

El rehashing lineal produce agrupamientos primarios[2], lo suyo, sin embargo, es ir dando saltos para encontrar más huecos libres. Para poder hacer esto está el rehashing doble.

Estados de las casillas

1. *Ocupada*
2. *Vacía*
3. *Borrada*

El estado *ocupada* y *borrada* significan lo mismo para el proceso de consulta.
El estado *vacía* y *borrada* significan lo mismo para el proceso de intersección.

Algoritmo para buscar una clave

Algorithm 7.2.1: SEARCHKEY(T,k)

comment: Buscar la clave k en la tabla T

h(c)$\leftarrow$ *Calcular*(h(c))
if *Estado*(h(c))$!=$ *borrada* **and** *clave*(h(c))$==c$
then { *posicion* $\leftarrow$ *registro*(h(c))
comment: la función registro devuelve la posición en disco de la información asociada a a c
comment: llamaremos n a $h_{i-1}(c)$

else if *Estado*(n(c))$!=$ *borrado* **and** (n(c))$==c$ **or** *Estado*(n(c))$==$ *vacia*
then {
comment: las dos condiciones unidas por el and se dan cuando se encuentra la clave
comment: la que viene después del or, se da cuando la clave no está en la tabla
if *Estado*(n(c))$!=$ *vacia*
then { **if** *clave*(n(c))$==c$
then {*posicion* $\leftarrow$ *registro*(n(c))
else {*posicion* $\leftarrow -1$
return (*posicion*)

Rehashing doble

La ventaja respecto al lineal es que los agrupamientos no son primarios, es decir, a partir de una posición no tendremos los demás sino que haremos un salto entre cada uno.
La función hash sería la de siempre:

$$h(k) = k\%M$$

Pero ahora, la función de rehashing sería la siguiente:

$$h_i(k) = (h_{i-1}(k) + h_0(k))\%M, \qquad \text{siendo } i = 2,3\ldots, \text{teniendo en cuenta que } h_1(k) = h(k)$$

[2]Se dan cuando para encontrar una casilla libre vamos avanzando al siguiente y al final quedan todos los elementos uno detrás de otro

$h_0(k)$ se calcularía de la siguiente forma:

$$h_0(k) = 1 + (k\%(M-2)), \qquad M \text{ y } M-2 \text{ se toman como primos relativos}$$

Ejemplo 7.2.6

Queremos crear una tabla hash cerrada con $M = 13$. El fichero de datos contiene 12 registros:

Clave	119	85	43	141	72	91	109	147	38	137	148	101
Registro	0	1	2	3	4	5	6	7	8	9	10	11
$h(k)$	2	7	4	11	7	0	5	4	12	7	5	10

Hemos calculado en una tabla, el $h_0(k)$ y el $h_1(k)$ de cada uno:

k	119	85	43	141	72	91	109	147	38	137	148	101
$h_1(k)$	2	7	4	11	7	0	5	4	12	7	5	10
$h_0(k)$	10	9	11	10	7	4	11	5	6	6	6	3

Al rellenar la tabla nos queda así:

$h(k)$	k	registro
0	91	5
1	72	4
2	119	0
3	101	11
4	43	2
5	109	6
6	137	9
7	85	1
8		
9	147	7
10	148	10
11	141	3
12	38	8

1. $h(119) = 119\%13 = 2$
2. $h(85) = 85\%13 = 7$
3. $h(43) = 43\%13 = 4$
4. $h(141) = 141\%13 = 11$
5. $h(72) = 72\%13 = 7 \longleftarrow$ Colisión
   a) $h_0(72) = 1 + (72\%11) = 7$
   b) $h_1(72) = h(72) = 7$
   c) $h_2(72) = (7+7)\%13 = 1$
6. $h(38) = 38\%13 = 12$
7. $h(91) = 91\%13 = 0$
8. $h(109) = 109\%13 = 5$
9. $h(147) = 147\%13 = 4 \longleftarrow$ Colisión
   a) $h_0(147) = 1 + (147\%11) = 5$
   b) $h_2(147) = (4+5)\%13 = 9$
10. $h(137) = 137\%13 = 7 \longleftarrow$ Colisión
    a) $h_0(137) = 1 + (137\%11) = 6$
    b) $h_2(137) = (7+6)\%13 = 0$
    c) $h_3(137) = (0+6)\%13 = 6$
11. $h(148) = 148\%13 = 5 \longleftarrow$ Colisión
    a) $h_0(148) = 1 + (148\%11) = 6$
    b) $h_2(148) = (5+6)\%13 = 11$
    c) $h_3(148) = (11+6)\%13 = 4$
    d) $h_4(148) = (4+6)\%13 = 10$
12. $h(101) = 101\%13 = 10 \longleftarrow$ Colisión
    a) $h_0(101) = 1 + (101\%11) = 3$
    b) $h_2(101) = (10+3)\%13 = 0$
    c) $h_3(101) = (3+0)\%13 = 3$

□

7.2.5 Sondeo aleatorio

La función hash a utilizar igual que en el rehashing lineal y doble sería: $h(k) = k\,\%M$ Cuando existe colisión, se aplica la siguiente ecuación:

$$h_i(k) = (h_{i-1}(k) + c)\,\%M \text{ para } i = 2,3\ldots$$

Donde:

1. M es el tamaño de la tabla. Suele ser el siguiente primo mayor al número de claves a insertar en la tabla.
2. c es una constante mayor que uno ($c > 1$). Debe ser primo relativo de M (no tener factores en común).
3. $h_1(k) = h(k)$

Ejemplo 7.2.7

Usando los datos del 7.2.6 vamos a insertar los elementos con sondeo aleatorio:
Tenemos $h(k) = k\,\%13$ y $c = 5$, por tanto $h_i(k) = (h_{i-1}(k) + 5)\,\%13$.

$h(k)$	k	registro
0	91	5
1	38	8
2	119	0
3	101	11
4	43	2
5	109	6
6	137	9
7	85	1
8		
9	147	7
10	148	10
11	141	3
12	72	4

1. $h(119) = 119\,\%13 = 2$
2. $h(85) = 85\,\%13 = 7$
3. $h(43) = 43\,\%13 = 4$
4. $h(141) = 141\,\%13 = 11$
5. $h(72) = 72\,\%13 = 7 \longleftarrow$ Colisión
   a) $h_2(72) = (7+5)\,\%13 = 12$
6. $h(91) = 91\,\%13 = 0$
7. $h(109) = 109\,\%13 = 5$
8. $h(147) = 147\,\%13 = 4 \longleftarrow$ Colisión
   a) $h_2(147) = (4+5)\,\%13 = 9$
9. $h(38) = 38\,\%13 = 12 \longleftarrow$ Colisión
   a) $h_2(38) = (12+5)\,\%13 = 4$
   b) $h_3(38) = (4+5)\,\%13 = 9$
   c) $h_4(38) = (9+5)\,\%13 = 1$
10. $h(137) = 137\,\%13 = 7 \longleftarrow$ Colisión
    a) $h_2(137) = (7+5)\,\%13 = 12$
    b) $h_3(137) = (12+5)\,\%13 = 4$
    c) $h_4(137) = (4+5)\,\%13 = 9$
    d) $h_5(137) = (9+5)\,\%13 = 1$
    e) $h_6(137) = (1+5)\,\%13 = 6$
11. $h(148) = 148\,\%13 = 5 \longleftarrow$ Colisión
    a) $h_0(148) = 1 + (148\,\%11) = 10$
12. $h(101) = 101\,\%13 = 10 \longleftarrow$ Colisión
    a) Ahora, iríamos dando saltos de 5 en 5 hasta encontrar un hueco libre. Para ello tendríamos que calcular hasta $h_{10} = 3$

En el sondeo aleatorio, para encontrar una casilla libre vamos dando saltos de tamaño c.

□

7.2.6 Definir una función hash de una cadena de caracteres.

Si queremos **definir una función hash a partir de un vector de caracteres**, tenemos que encontrar una función que nos de una posicion (índice) dentro de una tabla. Una funcion trivial seria coger el código ASCII de cada carácter:

$$h(k) = (k[0] + k[1] + \cdots + k[n-1])\,\%M$$

Otra posibilidad función hash es usar solamente un numero de caracteres (digamos los l primeros) de la cadena para obtener la función hash. Siguiendo esta idea la función hash podría ser:

$$h(k) = (L^0 * k[0] + L^1 * k[1] + \cdots + L^l * k[l-1])\,\%M$$

Por ejemplo si nuestro alfabeto tiene $L = 28$ caracteres, y tenemos una tabla hash con 5 posiciones y queremos asignar una posición a la cadena *Hola*. Vamos a usar un prefijo de 3 caracteres para la función hash y teniendo en cuenta que los códigos ASCII son:

- H tiene código ASCII 72
- o tiene código ASCII 111
- l tiene código ASCII 108

$$h(k) = (28^0 * 72 + 28^1 * 111 + 28^2 * 108)\,\%5 = 2$$

7.2.7 Otras funciones hash

Lo ideal sería encontrar una función hash inyectiva (sin colisiones), pero no es fácil encontrar una tabla con tal característica.
Por ejemplo, si tenemos una tabla hash con 40 posiciones y 30 claves para insertar:

1. ¿Cuántas funciones hash podemos definir? Cada casilla podemos rellenarla de 30 formas posibles, por tanto podríamos definir unas 40^{30} funciones.
2. De todas esas, son inyectivas $\binom{40}{30} = \frac{40!}{30!10!}$

La función hash más usada es $h(k) = k\,\%M$, que se llama **División-resto (módulo)**. Donde M se puede estimar en base del número de claves a insertar, como el siguiente primo a dicho número de claves. Este valor M puede cambiar en el proceso de inserción y borrado de elementos de la tabla hash, para hacer más eficiente el espacio que ocupa la tabla hash.
A continuación veremos otras funciones hash que se pueden derivar de la clave.
Truncamiento : consiste en quedarse con unos dígitos determinados de la clave. Por ejemplo:

$$h(123456789) = 123$$

En este ejemplo nos quedaríamos con los tres primeros dígitos. El inconveniente que tiene es que las tablas deben tener un tamaño potencia de 10 y por tanto, éste se dispara rápidamente. Una alternativa para resolver esto es hacer un **truncamiento a nivel de bit**, donde el tamaño sería una potencia de dos y el número se truncaría en binario.

Plegado : consiste en dividir la parte en, al menos, dos partes y sumar dichas partes. Por ejemplo:

$$h(123456) = 123 + 456 = 579$$

Si el resultado fuese un número muy grande se podría aplicar truncamiento. El tamaño debe ser potencia de 10 y como alternativa, se podría hacer en binario.

Multiplicación : igual que el plegado, pero en vez de sumar se multiplica.

Cuadrado del centro : consiste en quedarse con la parte central del número y calcular su cuadrado. Por ejemplo:

$$h(123456789) = 456^2 = 207936$$

Se podría aplicar truncamiento tras calcularlo.

Centro del cuadrado : Igual que antes, pero primero se aplica primero el cuadrado y luego se coge el número del centro:

$$h(1234) = 1234^2 = 1522756 = 2275$$

7.3 Tablas Hash en la STL

La STL define las tablas hash como las clases:

- *unordered_set,unordered_multiset*: Se usan cuando se quiere alamacenar un conjunto de claves. Los accesos por clave se hacen muy rápidos. Si se admiten claves repetidas se usará un *unordered_multiset*, en caso de que no se admita claves repetidas se usará un *unordered_set*.
- *unordered_map,unordered_multimap*: Se usan para almacenar de nuevo un conjunto de claves que tiene una información asociada a la clave. Si se admite claves repetidas se debe usar *unordered_multimap* en otros caso *unordered_map*.

Estas clases ya fueron introducidas en la sección 5.2.3. Aqui vamos a ver un ejemplo de uso.

Ejemplo 7.3.1

Algoritmo Karp-Rabin.- Este algoritmo pretende encontrar si un texto contiene una cadena. Un algoritmo basado en la fuerza bruta, haría todas las comparaciones entre el texto y la cadena de entrada de la siguiente forma:

```
1  int Fuerza_Bruta(const string & texto, const string &cadena){
2     int n = texto.size();
3     int m = cadena.size();
4     for (int i=0;i<n-m+1;i++){
5      bool seguir=true;
6      for (int j=0;j<m && !seguir; j++){
7        if (texto[i+j]!=cadena[j])
8          seguir =false;
9       }
10      if (seguir)
```

```
11        return i;
12      }
13      return -1;
14  }
```

Este algoritmo devuelve la posición en texto donde comienza la cadena que se busca, en caso de que exista. Y en otro caso devuelve -1, indicando que cadena no aparece en texto.
El algoritmo de Karp-Rabin se basa en el hecho de comparar el trozo del texto correspondiente y la cadena. Si el trozo del texto que se analiza y la cadena tienen la misma función hash se pasa a analizar sin son iguales carácter a carácter. En caso de que no sea así no se pierde tiempo haciendo el for que recorre para *j* en el algoritmo de la fuerza bruta. Veamos el código a continuación:

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <unordered_set>
5  using namespace std;
6  typedef unordered_set<string> stringset;
7
8  int Karp_Rabin(const string & texto, const string & cadena){
9     stringset myconj;
10    //obtenemos la funcion hash para string
11    stringset::hasher fn = myconj.hash_function();
12    int n= texto.size();
13    int m =cadena.size();
14    int hp = fn(cadena); //obtenemos el valor hash de cadena
15    int hs= fn(texto.substr(0,m)); //funcion hash del trozo de texto
16    for (int i=0;i<n-m+1;++i){
17        hs =fn(texto.substr(i,m));
18       if (hp==hs){ //ahora comparamos
19          if (texto.substr(i,m)==cadena)
20            return i;
21       }
22    }
23    return -1;
24 }
```

Al principio del código se redefine el tipo *unordered_set<string>* como *stringset*. En la línea 10 del código se puede ver que se obtiene la función hash de un conjunto de string en la variable *fn*. Dentro del bucle, líneas 16-22, se hace la comparación de las cadenas cuando ambas tienen la misma función hash.

□

Ejemplo 7.3.2

Redefinamos la representación del T.D.A Diccionario, visto en el ejemplo 5.1.1, usando un orderer_map. Recordad que un diccionario en términos generales es una colección de pares, en la que cada par se conforma por una clave y un conjunto de informaciones asociadas. En nuestra primera aproximación usando tablas hash vamos a definir Diccionario como:

```
#include <string>
#include <unordered_map>
#include <iostream>
using namespace std;

template <class T,class U>
class Diccionario{
          private:
                    unordered_map<T,U> datos;
          ...
};
```

Hay que tener en cuenta que los datos alamacenados en un unordered_map no están ordenados ni por su clave ni por su valor asociado, ya que se organizan en cubetas dependiendo de sus valores hash. De esta forma es muy rápido acceder a elementos directamente por su valor clave. Así que el TDA unordered_map es más rápido que el map para acceder a elementos individuales por su clave. No obstante se muestran menos eficientes cuando se quiere acceder a un rango de valores, p.e todos los elementos con valor clave comprendidos entre $[a,b]$.
A continuación veamos las operaciones de Diccionario:

- *Constructores*
- *Destructores*
- *Consultas*: número de entradas, comprobar si una clave ya esta en el diccionario, conseguir la información asociada a una clave
- *Modificadores*: insertar una nueva clave con su definición, modificar la información asociada a una clave

Antes de ver los métodos, vaeamos que dentro de la clase vamos a definir dos iteradores: **iterator y const_iterator** .

```
#include <string>
#include <unordered_map>
#include <iostream>
using namespace std;

template <class T,class U>
class Diccionario{
private:
        unordered_map<T,U> datos;
public:
...
  class const_iterator;
```

```
13    class iterator{
14    private:
15          typename unordered_map<T,U> ::iterator punt;
16    public:
17              iterator(){}
18              iterator & operator ++(){
19                        punt++;
20                        return *this;
21              }
22              iterator & operator --(){
23                        punt--;
24                        return *this;
25              }
26              bool operator ==(const iterator & it){
27                        return it.punt==punt;
28              }
29              bool operator !=(const iterator & it){
30                        return it.punt!=punt;
31              }
32              pair<const T,U> & operator *(){
33                        return *punt;
34              }
35              friend class Diccionario;
36              friend class const_iterator;
37    };
38    //redefinimos iterator
39
40
41    class const_iterator{
42              private:
43                  typename unordered_map<T,U> ::const_iterator punt;
44              public:
45                        const_iterator(){}
46                        const_iterator(const iterator &it){
47                                  punt = it.punt;
48                        }
49                        const_iterator & operator ++(){
50                                  punt++;
51                                  return *this;
52                        }
53
54                        const_iterator & operator --(){
55                                  punt--;
```

```
56                              return *this;
57                      }
58                      bool operator ==(const const_iterator & it){
59                              return it.punt==punt;
60                      }
61                      bool operator !=(const const_iterator & it){
62                              return it.punt!=punt;
63                      }
64                      const pair<const T,U> & operator *()const{
65                              return *punt;
66                      }
67                      friend class Diccionario;
68
69      };
70
71
72      iterator   begin(){
73                  iterator it;
74                  it.punt =datos.begin();
75                  return it;
76      }
77      iterator end(){
78                  iterator it;
79                  it.punt =datos.end();
80                  return it;
81      }
82
83      const_iterator begin()const{
84                  const_iterator it;
85                  it.punt =datos.begin();
86                  return it;
87      }
88      const_iterator end()const {
89                  const_iterator it;
90                  it.punt =datos.end();
91                  return it;
92      }
93
94  };
```

Como se puede observar la *clase iterator* se representa como un iterador de unordered_map. Los métodos asociados los podemos ver a continuación:

```
1   class Diccionario{
2   public:
```

```
3   ....
4    Diccionario(){}
5    Diccionario(const Diccionario &D):datos(D.datos){
6
7    }
8    ~Diccionario(){
9      Borrar();
10   }
11   void clear(){
12     Borrar();
13   }
14
15   int size()const{
16             return datos.size();
17   }
18
19  //redefinimos iterator como iterdiccio
20  typedef Diccionario<T,U>::iterator iterdiccio;
21
22
23  //Averigua si una clave ya esta insertada.
24  pair<bool,iterdiccio> Esta_Clave(const T &p){
25           pair<bool,iterdiccio> res;
26           if (datos.size()>0){
27
28               typename unordered_map<T,U> ::iterator it;
29               it = datos.find(p);
30
31               if (it==datos.end()){
32                   res = {false,end()};
33               }
34               else {
35                   iterdiccio i;
36                   i.punt=it;
37                   res = {true,i};
38               }
39
40               return res;
41           }
42           res ={false,end()};
43           return res;
44
45   }
```

```
46    //Inserta un nuevo par clave informacion asociada
47    void Insertar(const T& clave,const  U &info){
48
49
50        pair<bool,iterdiccio> res = Esta_Clave(clave);
51
52        if (!res.first){
53                  pair<T,U> p(clave,info);
54                  datos.insert(p);
55
56        }
57
58    }
59    // el tipo U de la informacion asociada a la clave requiere que tenga
60    //implementada la funcion push_back
61    template <class signi>
62    void AddSignificado(const T &p,const signi &s){
63            pair<bool,iterdiccio> res = Esta_Clave(p);
64              if (!res.first){
65                      U aso;
66                      aso.push_back(s);
67                      Insertar(p,aso);
68              }
69              else
70              //Insertamos el siginificado al final
71              (*(res.second)).second.push_back(s);
72
73
74    }
75    //Modifica el significado
76    void UpdateSignificado_Palabra(const T &p,const U & s ){
77              pair<bool,iterdiccio> res = Esta_Clave(p);
78              if (!res.first){
79
80                      Insertar(p,s);
81              }
82              else
83              //Insertamos el siginificado al final
84              (*(res.second)).second=s;
85    }
86    //Obtiene la informacion asociada a una clave.
87    U  getInfo_Asoc(const T & p) {
88              pair<bool,iterdiccio> res = Esta_Clave(p);
```

```
89              if (!res.first){
90                  return U();
91              }
92              else{
93
94                  return ((*(res.second)).second);
95              }
96    }
```

Hemos redeclarado el iterador de diccionario como *iterdiccio* con la siguiente sentencia:

```
typedef Diccionario<T,U>::iterator iterdiccio;
```

Un ejemplo de uso de nuestro diccionario podría ser el siguiente:

```
1  #include <iostream>
2  #include "diccionario.h"
3  #include <list>
4  #include <fstream>
5
6  //redefinimos un iterator
7  typedef Diccionario<string,list<string> >::iterator iter;
8  //redefinimos un const_iterator
9  typedef Diccionario<string,list<string> >::const_iterator const_iter;
10
11
12 ostream & operator<<(ostream & os, const Diccionario<string,list<string> > & D){
13   const_iter it;
14   for (it=D.begin(); it!=D.end(); ++it){
15
16             list<string>::const_iterator it_s;
17             os<<endl<<(*it).first<<endl<<" informacion asociada:"<<endl;
18             for (it_s=(*it).second.begin();it_s!=(*it).second.end();++it_s){
19                       os<<(*it_s)<<endl;
20             }
21             os<<"************************************"<<endl;
22   }
23
24   return os;
25 }
26 //EL formato de la entrada es:
27 //   numero de claves en la primera linea
28 //   clave-iseima retorno de carro
29 //   numero de informaciones asociadas en la siguiente linea
30 //   en cada linea informacion asociada
31
```

```
32 istream & operator >>(istream & is,Diccionario<string,list<string> > &D){
33   int np;
34   is>>np;
35   is.ignore();//quitamos \n
36   Diccionario<string,list<string> > Daux;
37   for (int i=0;i<np; i++){
38             string clave;
39
40             getline(is,clave);
41
42             int ns;
43             is>>ns;
44             is.ignore();//quitamos \n
45             list<string>laux;
46             for (int j=0;j<ns; j++){
47                       string s;
48                       getline(is,s);
49                       laux.insert(laux.end(),s);
50             }
51             Daux.Insertar(clave,laux);
52   }
53   D=Daux;
54   return is;
55 }
56 void EscribeSigni(const list<string>&l){
57   list<string>::const_iterator it_s;
58   for (it_s=l.begin();it_s!=l.end();++it_s){
59             cout<<*it_s<<endl;
60   }
61 }
62 int main(int argc, char * argv[]){
63  if (argc!=2){
64     cout<<"Los parametros son:"<<endl;
65     cout<<"1.- El nombre del fichero con las definiciones"<<endl;
66     return 0;
67  }
68
69
70   Diccionario<string,list<string> > D;
71   ifstream f (argv[1]);
72   f>>D;
73   f.close();
74   cout<<D;
```

```
75    string a;
76
77    cout<<"Introduce una palabra"<<endl;
78    cin>>a;
79    list<string>l=D.getInfo_Asoc(a);
80    if (l.size()>0)
81              EscribeSigni(l);
82    cout<<"Dime un nuevo siginificado de la palabra "<<a<<endl;
83    string new_signi;
84    cin>>new_signi;
85
86
87    D.AddSignificado(a,new_signi);
88    cout<<"Despues de anhadir significado ************"<<endl;
89    cout<<D<<endl;
90    //usando iterdiccio
91    iter myiter;
92    myiter=D.begin();
93    while (myiter!=D.end()){
94       cout<<(*myiter).first<<endl;
95       ++myiter;
96    }
97
98 }
```

□

Ejemplo 7.3.3
Redefinir la función hash al TDA Diccionario dado en el ejemplo 7.3.2. De forma que se aplique la función hash sobre los primeros *len* elementos de la clave.
Para llevar a cabo este ejercicio vamos a redefinir una función hash en Diccionario y para ello construiremos la clase my_hash, de tal forma:

```
1  template <class T,class U>
2  class Diccionario{
3    private:
4      //funcion hash
5      class my_hash{
6        private:
7            //numero de elementos sobre los que calcular la funcion hash
8            unsigned int len;
9            //razon para pasar de un elemento a otro
10           unsigned int factor;
11       public:
12         //Constructor
```

```
13          my_hash(unsigned int l=3,unsigned int f=28):len(l),factor(f){}
14
15          //modifica len y factor
16          void set(int l,int f){
17              len=l; factor=f;
18
19          }
20          //devuelve el valor hash de la clave usando solamente los len primero
21          //valores de clave
22          size_t operator()(const T & clave) const{
23            size_t s=0;
24            int ff=1;
25            for (int i=0;i<len;i++){
26              s=(int)(s+ff*clave[i]);
27              ff*=factor;
28            }
29            return s;
30          }
31
32      };
33
34      unordered_map<T,U,my_hash> datos;
35      ....
36  };
```

La definición de *datos* (línea 34) además de los tipos del unordered_map se da un tercer elemento que indica la clase que implementa la función hash. Esta clase tiene que tener sobrecargado el operador () que se aplica sobre una clave. Así la redefinición de la función hash se obtiene mediante la implementación del operador () de my_hash, que se define como:

$$fh(clave) = factor^0 * clave[0] + factor^1 * clave[1] + ... + factor^{len-1} * clave[len-1]$$

□